

Level 0 Training

Session 4

STL, Part 1

Vector, pair, deque, queue,
priority queue, and stack

Choose the right tool. Trace its state.
Understand its operations and their cost.

How to use this reference

This is a reference to study **after the session**, not a list of commands to memorize. Every new tool is connected to a problem: what must we store, which item do we need next, and how much work will each operation take?

A practical study routine

1. **Read one idea.** Describe the tool's job without using its C++ name. "The person who arrived first is served first" is more useful than memorizing a declaration.
2. **Draw the state.** Write the stored values after each operation. Label the front, back, or top. Predict what will be removed next.
3. **Type and change the example.** Use an empty input where allowed, one item, equal items, and values arriving in a different order. Explain each new result.
4. **Solve independently.** Attempt the practice before opening the hint. After studying a solution, close it and write your own version from the idea.

How examples are labelled

A **complete program** has its own headers and `main`. Compile one file at a time. A **fragment** is a smaller illustration; its placement is stated. A **wrong example** is deliberately incorrect and must not be copied as a template.

Pause and predict questions have explained answers nearby; cover the answer first. **Optional extensions** and the final two practice challenges can wait until the core tools feel familiar.

C++23 and the editable source package

The examples use standard headers and readable C++23. Printed complete listings come directly from the supplied `.cpp` files. The standalone handouts and collected book share their chapter sources, so a correction does not need to be made twice.

Read in stages

You should be able to read input, use loops and arrays, write a helper, distinguish a copy from a reference, and recognize $O(1)$, $O(n)$, and $O(n \log n)$. Return to Sessions 1–3 whenever one of these foundations feels unclear.

Do not read the whole chapter in one sitting. Study vectors first, then pairs and deques, then queues, priority queues, and stacks. Use the revision tables *after* working through the 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 4: STL, Part 1**. Section numbers begin with 4 to match the collected book. The updated collected edition contains Sessions 1–4.

What Session 4 will help you do

Build a growable sequence, keep two related values together, work at both ends of a line, serve arrivals fairly, repeatedly select the largest or smallest value, and remember unfinished work in reverse order. Explain not only which operation you use, but why its order and cost fit the problem.

What can wait

Writing templates, lambdas, allocators, custom class types, advanced heap algorithms, monotonic queues, graphs, and later-session containers are not prerequisites. Small optional sections introduce a few useful conveniences and a bounded-heap challenge.

Conventions

Input follows each stated contract. All text exercises use ordinary ASCII characters. Sequence indices are zero-based unless a statement explicitly asks for a one-based position. Complexity assumes fixed-width numbers and constant-cost element operations; amortized costs are labelled. Storage that a program explicitly creates is included unless an alternative convention is stated.

Contents

How to use this reference	i
The Level 0 learning path	ii
4 STL, Part 1	1
4.1 From arrays to standard-library tools	2
4.2 Headers, type arguments, and common operations	3
4.3 Vector construction: empty, sized, and filled	5
4.4 Reading and building vectors	7
4.5 Traversal, access, and safe boundaries	9
4.6 Size, capacity, reserve, and resize	11
4.7 Iterator positions and half-open ranges	13
4.8 Inserting, erasing, and clearing vectors	15
4.9 Useful algorithms with vectors	17
4.10 Vectors and functions: copies versus references	20
4.11 Two-dimensional and nested vectors	22
4.12 Vector growth, amortized cost, and invalidation	24
4.13 Pair: two values travel together	26
4.14 Vectors of pairs, comparison, and tie rules	28
4.15 Deque: two ends and indexed access	30
4.16 A deque simulation: keep only the latest values	32
4.17 Queue: first in, first out	34
4.18 Queue simulations: commands and repeated turns	36
4.19 Priority queue: select the greatest value first	39
4.20 Min-heaps and priority pairs	42
4.21 Stack: last in, first out	45
4.22 Stack patterns: brackets, cancellation, and undo	47
4.23 Copies, fresh states, and multiple test cases	51
4.24 Choose the tool and count the actual operations	53
4.25 Optional extensions: convenient syntax and algorithms	56
4.26 Optional extension: keep the largest k while streaming	60
4.27 Debugging and testing STL-based solutions	62
4.28 Practice problems: choose, implement, and explain	64
4.29 Trace, tool-choice, and complexity drills	75
4.30 Hints before solutions	81
4.31 Explained practice solutions	85
4.32 Explained trace and complexity drills	108
4.33 Quick reference, glossary, and readiness checklist	114
4.34 Technical notes and references	118

SESSION 4

STL, Part 1

Vector, pair, deque, queue, priority queue, and stack

In Session 2 you stored values in arrays and strings. In Session 3 you learned to give a job a function name and to count its work. Now you will learn ready-made tools for storing and processing values in different ways.

The important question is not “Which container has the most functions?” It is “**Which operations does this problem need?**” A list of scores, a waiting line, and a pile of unfinished actions have different rules. Choosing a matching tool makes those rules easier to express and test.

Six tools, six useful jobs

Tool	Think of it as	Typical need
vector	A sequence that can grow.	Store scores and access an index.
pair	Two labelled fields together.	Keep a score with a student ID.
deque	A line accessible at both ends.	Add or remove at either end.
queue	A first-arrival waiting line.	Serve the oldest waiting item.
priority_queue	A collection with a priority rule.	Select the greatest or smallest item.
stack	A pile with one accessible top.	Finish or undo the newest action.

Your route through the chapter

Stage A: build sequences. Learn vector construction, safe access, iteration, useful algorithms, copies, and growth. Then group values with pairs and work at both ends with a deque.

Stage B: control processing order. Compare queue, priority queue, and stack using traces and short simulations. An access rule is part of the algorithm, not merely a C++ detail.

Stage C: practise and explain. Choose a tool, state its invariant, write the operations, test empty states and ties, and count total work. An **invariant** is a statement that remains true after every step, such as “the front is the next person to serve.”

Do not memorize all six declarations at once

Start with one small example per tool. Draw its contents after each operation. Once you can predict the next item correctly, the syntax becomes much easier to remember.

4.1 From arrays to standard-library tools

Reuse a tested operation instead of rebuilding it

Suppose a task gives you scores, but only some pass a condition. A fixed array can hold them, yet you must separately manage its capacity, active length, and insertion position. A vector keeps track of its own current length and provides an operation that appends an element.

You still design the algorithm. The library does not decide which scores pass, how ties should be resolved, or what the answer means. It supplies operations that you can combine into that algorithm.

What does STL mean here?

STL expands to **Standard Template Library**. In this training, “STL tools” means the standard-library containers, adaptors, iterators, algorithms, and related utilities that we use in problem solving. It is not a claim that every feature of the C++ standard library is a container.

A **container** holds elements. A **sequence container**, such as vector or deque, keeps them in a sequence. A **container adaptor**, such as queue or stack, uses another container internally but exposes a more restricted interface. A pair is a **utility type holding two values**, not a sequence container.^{[4-1][4-3][4-12][4-16][4-19]}

The operations explain the difference

If values arrive as 4 9 2, a queue will remove 4 next. A stack will remove 2. A default integer priority queue will remove 9. The same input leads to different behavior because the processing rules differ.

Question	Why it matters
Do I need an index?	Vector and deque provide indexed access; the three adaptors do not.
Which item leaves next?	Arrival order, last-in order, and priority order require different interfaces.
Do I change the front of-ten?	Shifting a vector’s remaining elements repeatedly may be expensive.
Must the original re-main?	Copying a container costs both work and additional storage.

Pause and predict – 1

Three jobs arrive in this order: 8 3 6. Which value is removed first by a queue, a stack, and a default integer priority queue?

Check your prediction. Queue: 8. Stack: 6. Priority queue: 8, because it is the greatest value, not because it arrived first. Try 3 8 6 to see that difference.

4.2 Headers, type arguments, and common operations

Include the header for the tool you use

Header	Used here for
<code><vector></code>	<code>std::vector</code> .
<code><utility></code>	<code>std::pair</code> and <code>std::make_pair</code> .
<code><deque></code>	<code>std::deque</code> .
<code><queue></code>	Both <code>std::queue</code> and <code>std::priority_queue</code> .
<code><stack></code>	<code>std::stack</code> .
<code><algorithm></code>	Algorithms such as <code>sort</code> , <code>reverse</code> , and <code>find</code> .
<code><numeric></code>	<code>std::accumulate</code> .
<code><functional></code>	Comparison tools such as <code>std::greater</code> .

The supplied programs include the headers for their own operations. They do not depend on a compiler-specific “include everything” header or on one standard header accidentally including another. The declarations of these facilities specify their owning headers.^{[4-1][4-12][4-14][4-16][4-17][4-19][4-8][4-11][4-28]}

Read a declaration from the outside in

Fragment – inside a function:

```
std::vector<int> scores;
std::vector<long long> totals;
std::vector<std::string> names;
```

`std::vector` names a standard-library template. The type between angle brackets specifies the **element type**. The variable name comes after the complete type. Every element in `scores` is an `int`; every element in `names` is a string. Include `<string>` for the third declaration.

You are *using* a template, not writing one. Think of it as choosing the type of item for a ready-made storage tool. A vector can grow, but its element type does not change while the program runs.

The dot calls an operation on an object

`scores.size()` asks *this vector* for its element count. `scores.push_back(90)` appends a value to *this vector*. These are member-function calls, just like the string operations used earlier.

Size, emptiness, and a common misunderstanding

All five containers/adaptors in this chapter provide `size()` and `empty()`. A pair does not: it has two named fields rather than a variable-length sequence.^{[4-4][4-12][4-16][4-17][4-19]}

```
std::vector<int> scores;
std::cout << scores.size() << '\n';    // 0
std::cout << scores.empty() << '\n';   // 1: true
scores.push_back(90);
std::cout << scores.empty() << '\n';   // 0: false
```

`empty()` **checks** whether there are no elements. It does not erase them. For a vector or deque, `clear()` removes all elements; for the three adaptors, we will use repeated removal or create a fresh object.

A small introduction to auto

`auto` asks the compiler to deduce a variable's type from its initializer. It does not mean the variable can later hold an unrelated type.^[4-29]

```
auto count = scores.size(); // Use the returned size type.
auto copy = scores;         // A separate vector<int>.
auto& same = scores;        // A reference to scores.
```

The `&` still matters. `auto copy = scores;` copies the vector; `auto& same = scores;` refers to the existing vector. We will use `auto` when iterator or nested types would otherwise distract from the idea.

Lengths and integer indices

Container sizes use an unsigned size type. When an example converts a size to `int`, its contract limits the length to at most 200000, which fits the tested training environment's `int`. This is not permission to narrow an arbitrarily large container length.

```
int n = static_cast<int>(scores.size());
for (int i = n - 1; i >= 0; --i) {
    std::cout << scores[i] << '\n';
}
```

Converting the length *before* subtracting avoids performing `size() - 1` in an unsigned type when the vector is empty. The signed loop starts at `-1` for an empty vector and performs no accesses.

Pause and predict – 2

Does `waiting.empty();` remove every item from `waiting`? What is the difference between `auto copy = values;` and `auto& copy = values;`?

Check your prediction. No: `empty()` returns a truth value and leaves the object unchanged. The first declaration creates a separate object; the second creates another way to refer to the existing object.

4.3 Vector construction: empty, sized, and filled

A vector owns a growable sequence

A `std::vector<int>` stores integers in order and manages the sequence's storage. For these ordinary element types, its elements occupy contiguous storage. Changing the vector's size is supported; accessing an index outside its size is not.^[4-1]

Declaration	Initial elements
<code>std::vector<int> a;</code>	No elements; size 0.
<code>std::vector<int> b(3);</code>	0 0 0; size 3.
<code>std::vector<int> c(3, 7);</code>	7 7 7; size 3.
<code>std::vector<int> d{3, 7};</code>	Two listed values: 3 7.
<code>std::vector<int> e = c;</code>	An independent copy of c.

For the standard allocator and `int`, the size-only constructor initializes the created elements to zero. This differs from an uninitialized built-in local array such as `int a[3];`.^[4-2]

Parentheses and braces do different jobs

Fragment – inside a function:

```
std::vector<int> a(5);    // Five zero-valued elements.
std::vector<int> b{5};   // One element whose value is 5.
std::vector<int> c(2, 9); // Two copies of 9.
std::vector<int> d{2, 9}; // The values 2 and 9.
```

Both spellings are valid, which makes this mistake easy to miss: the program may compile while constructing the wrong data. Before using a vector, say its intended contents aloud.

A length read at runtime is allowed

```
int n = 0;
std::cin >> n;
std::vector<int> values(n);
```

With valid input $0 \leq n \leq 200000$, this creates exactly `n` elements. A vector of size zero is valid. Unlike a variable-length built-in array extension, this is ordinary standard C++.

Pause and predict – 3

What are the sizes and contents of `std::vector<int> a(4, 2);` and `std::vector<int> b{4, 2};`?

Check your prediction. `a` has four elements: 2 2 2 2. `b` has two elements: 4 2. The braces list actual values.

Run the construction examples

This program takes no input. Its output records a size and, for nonempty vectors, the first stored value.

Complete program – four different initial states

```
#include <iostream>
#include <vector>

int main() {
    std::vector<int> empty;
    std::vector<int> zeros(3);
    std::vector<int> copies(3, 7);
    std::vector<int> listed{3, 7};
    std::cout << empty.size() << '\n';
    std::cout << zeros.size() << ' ' << zeros[0] << '\n';
    std::cout << copies.size() << ' ' << copies[0] << '\n';
    std::cout << listed.size() << ' ' << listed[0] << '\n';
}
```

Sample input

(no input)

Sample output

0
3 0
3 7
2 3

Read the output line by line

The first vector has no elements, so we only print its size. The second vector has three zero-valued integers. The third has three copies of seven. The fourth has two explicitly listed values, and the first of them is three.

Notice that `empty[0]` would be invalid even though the object `empty` itself exists. Creating a vector object and creating elements inside it are separate actions.

A vector cannot ignore the memory limit

Automatic storage management does not make memory unlimited. Choose a size that fits the problem constraints and memory limit. Large allocation failures are not handled by the contest examples; the input and environment must satisfy their stated contracts.

4.4 Reading and building vectors

Strategy A: create the elements, then fill them

When the number of values is known, construct n elements and read into each existing index. This is close to the array pattern from Session 2.

Task. Read $0 \leq n \leq 200000$ integers, each between -10^9 and 10^9 , and print them in reverse input order. A zero-length input produces a blank output line.

Complete program – sized vector input

```
#include <iostream>
#include <vector>

int main() {
    int n = 0;
    std::cin >> n;
    std::vector<int> values(n);
    for (int i = 0; i < n; ++i) {
        std::cin >> values[i];
    }
    for (int i = n - 1; i >= 0; --i) {
        if (i != n - 1) std::cout << ' ';
        std::cout << values[i];
    }
    std::cout << '\n';
}
```

Sample input

4
8 3 9 1

Sample output

1 9 3 8

After the input loop, `values[i]` contains the $(i + 1)$ -th input. The output loop visits indices $n-1$ down to 0, so it prints exactly those values in reverse. It does not modify the stored sequence.

Cost. Creating, reading, and printing n integers takes $O(n + 1)$ total time and $O(n + 1)$ storage. The $+1$ includes fixed work when n is zero; in ordinary positive-size discussion, we write $O(n)$.

Pause and predict – 4

You need to store only even inputs. Should the result vector begin with n zero-valued elements or begin empty?

Check your prediction. Begin empty, then append each accepted input. Otherwise the initial zeros become part of the stored result even though they were not accepted inputs.

Strategy B: begin empty, then append accepted values

When only some input values belong in the answer, the final count is not known yet. Start empty and use `push_back` once for every accepted value.

Task. Under the same bounds, keep the nonnegative inputs in their original order. Print their count and then the kept values; the second line is blank when none are kept.

Complete program – build a filtered sequence

```
#include <iostream>
#include <vector>

int main() {
    int n = 0;
    std::cin >> n;
    std::vector<int> kept;
    for (int i = 0; i < n; ++i) {
        int x = 0;
        std::cin >> x;
        if (x >= 0) kept.push_back(x);
    }
    std::cout << kept.size() << '\n';
    for (int i = 0; i < static_cast<int>(kept.size()); ++i) {
        if (i != 0) std::cout << ' ';
        std::cout << kept[i];
    }
    std::cout << '\n';
}
```

Sample input

5
-4 0 7 -1 2

Sample output

3
0 7 2

Why it works. After each input, the vector contains exactly the nonnegative values seen so far, in arrival order. A negative input changes nothing; an accepted input is appended after all earlier accepted inputs.

Wrong combination:

```
std::vector<int> values(n);
for (int i = 0; i < n; ++i) {
    int x = 0;
    std::cin >> x;
    values.push_back(x); // Appends AFTER the n existing zeros.
}
```

This produces $2 \cdot n$ elements, not n . Choose either “create and fill” or “start empty and append” for this job. Appending does not overwrite an existing zero.

4.5 Traversal, access, and safe boundaries

The same zero-based indexing rule still applies

For a vector of size n , the element indices are 0 through $n-1$. The expression `values[n]` is not an extra slot for the next value. Use `push_back` to append or `resize` to create more elements.^[4-3]

Operation	Required condition / meaning
<code>values[i]</code>	i identifies an existing element. No required bounds exception in C++23.
<code>values.at(i)</code>	Returns that element; throws <code>std::out_of_range</code> if the index is outside the size.
<code>values.front()</code>	The first element; vector must be nonempty.
<code>values.back()</code>	The last element; vector must be nonempty.

These access rules come from the sequence-container requirements. The examples prevent invalid calls rather than teaching exception handling. An uncaught exception from `at` stops the program; it does not silently fix the index.^[4-3]

Choose an indexed loop when the position matters

```
int n = static_cast<int>(values.size());
for (int i = 0; i < n; ++i) {
    std::cout << i << ' ' << values[i] << '\n';
}
```

Choose a range-based loop when you only need each element. The loop variable's type determines whether you read a copy or refer to the actual element.^[4-20]

```
for (int x : values) std::cout << x << '\n'; // Read copies.
for (int& x : values) x += 2;                // Change elements.
for (const int& x : values) {
    std::cout << x << '\n';                // Read-only access.
}
```

For a small `int`, copying is cheap. For a vector of strings or other larger objects, `const auto&` can avoid copying each element when you only want to read it.

Pause and predict – 5

A vector contains 1 2 3. After `for (int x : values) { x += 10; }`, what are its elements?

Check your prediction. Still 1 2 3. Each `x` is a separate integer copy. Use `int& x` to modify the actual elements.

Use references for input and updates

Task. Read n , an increment delta , and n integers. Add delta to every stored value and print their total. Here $0 \leq n \leq 200000$, $|a_i| \leq 10^6$, and $|\text{delta}| \leq 1000$.

Complete program – modify elements, then read them

```
#include <iostream>
#include <vector>

int main() {
    int n = 0, delta = 0;
    std::cin >> n >> delta;
    std::vector<int> values(n);
    for (int& x : values) std::cin >> x;
    for (int& x : values) x += delta;
    long long total = 0;
    for (int x : values) total += x;
    std::cout << total << '\n';
}
```

Sample input

```
3 2
4 1 7
```

Sample output

```
18
```

The first loop needs references because input must change the stored elements. The second also needs references because each stored value is updated. The third only reads integer values, so copies are sufficient.

The element values fit `int` under this contract, but their sum may not. Therefore `total` is `long long`. The container's element type and the result's type need not be the same.

Check the state before accessing an end

```
if (!values.empty()) {
    std::cout << values.front() << ' ' << values.back() << '\n';
}
```

The guard must run *before* access. Testing `values.back()` and then checking `empty()` is too late. An empty vector has no first or last element.

Do not grow or shrink a vector in a range-based loop over it

The loop remembers positions into that sequence. Appending or erasing may invalidate those positions. Use a separate result vector, or a deliberately designed indexed/iterator loop. We return to this issue in Section 4.12.

4.6 Size, capacity, reserve, and resize

Existing elements versus allocated room

Size is the number of elements that currently exist. **Capacity** is the number the vector can hold without reallocating its storage. Capacity can exceed size, but the extra room is not a collection of accessible elements.^[4-5]

Imagine a notebook with room for ten records but only three written records. The room can make future writing easier; it does not make a fourth record already exist. Similarly, valid indices depend on `size()`, never on `capacity()`.

Call	What changes?
<code>reserve(k)</code>	Requests capacity at least <code>k</code> ; does not change size.
<code>resize(k)</code>	Changes size to <code>k</code> ; added <code>int</code> elements are zero-valued.
<code>resize(k, value)</code>	Changes size; new elements receive <code>value</code> . Existing retained elements are not overwritten.
<code>clear()</code>	Removes every element; capacity is retained.
<code>shrink_to_fit()</code>	Requests less unused capacity; the request is not binding.

The capacity and resize rules are specified separately from the sequence operations. The exact capacity and growth factor are not promised; do not write expected output that assumes doubling.^{[4-5][4-3]}

Wrong example – reserve is not resize:

```
std::vector<int> values;
values.reserve(100);
values[0] = 7; // WRONG: size is still 0.
```

Use `values.push_back(7);` to append one element. To create a hundred elements instead, call `values.resize(100);` before assigning to index zero.

Pause and predict – 6

A vector contains 4 5. What does `resize(4, 9)` produce? What happens after a following `reserve(20)`?

Check your prediction. The elements become 4 5 9 9. Reserving more room does not change those four values or their count; it only ensures enough capacity for at least twenty elements.

Trace several size changes

This program takes no input. It prints 1 for a true comparison rather than printing an implementation-dependent capacity.

Complete program – room is not the same as content

```
#include <iostream>
#include <vector>

int main() {
    std::vector<int> values;
    values.reserve(5);
    std::cout << values.size() << ' '
              << (values.capacity() >= 5) << '\n';
    values.resize(3, 8);
    values.push_back(2);
    std::cout << values.size() << ' ' << values.back() << '\n';
    values.resize(2);
    std::cout << values.size() << ' ' << values.back() << '\n';
    values.clear();
    std::cout << values.empty() << ' '
              << (values.capacity() >= 5) << '\n';
}
```

Sample input	Sample output
(no input)	0 1
	4 2
	2 8
	1 1

After the call	Size	Elements
reserve(5)	0	None.
resize(3, 8)	3	8 8 8.
push_back(2)	4	8 8 8 2.
resize(2)	2	8 8.
clear()	0	None; capacity is still at least 5.

Use reserve as an optimization, not as a ritual

When you plan to append many values and already know a sensible upper count, reserving once can avoid repeated reallocations. When you use `vector<int> values(n)`, the `n` elements already exist and input should fill them.

Calling `reserve(size()+1)` before every append can interfere with efficient growth and lead to repeated copying. Correctness should not depend on reserving, except that existing-element references can become invalid when a reallocation occurs.

4.7 Iterator positions and half-open ranges

A position that algorithms can use

An **iterator** is a value representing a position in a sequence. For a vector, `begin()` refers to its first element when it is nonempty, and `end()` is the position *just after* its last element. For an empty vector, these two positions are equal.^{[4-4][4-7]}

Element index	0	1	2	3	Past the end
Value	10	20	30	40	No element

`begin()+2` identifies the element with value 30. `end()` is useful as a boundary, but `*end()` is not a valid element access. For these vectors, `*it` means “the element at iterator `it`.”

The interval includes the first position, excludes the last

A range `[first, last)` includes **first** and stops *before last*. This is called a **half-open range**. The whole vector is `[begin(), end())`.^[4-7]

```
// Inside a function: values contains at least four elements.
std::reverse(values.begin() + 1, values.begin() + 4);
```

This reverses indices 1, 2, and 3; index 4 is excluded. An empty valid range has equal endpoints. Its algorithm performs no element work.

Iterate without inventing a separate index

```
for (auto it = values.begin(); it != values.end(); ++it) {
    std::cout << *it << '\n';
}
```

`++it` advances to the next position. It is checked against `end()` before being dereferenced. This is another way to traverse the same vector, not a faster algorithm than an ordinary indexed scan.

Pause and predict – 7

A vector is 10 20 30 40 50. Which values belong to `[begin()+1, begin()+4)`? Can `end()` be dereferenced?

Check your prediction. The values are 20 30 40. The last endpoint is excluded. No: `end()` is a boundary position with no element to read.

Reverse only the requested part

Task. Read n , $left$, $right$, then n integers. Reverse the half-open index range $[left, right)$. The input guarantees $0 \leq left \leq right \leq n \leq 200000$, with values in $[-10^9, 10^9]$.

Complete program – pass a precise range

```
#include <algorithm>
#include <iostream>
#include <vector>

int main() {
    int n = 0, left = 0, right = 0;
    std::cin >> n >> left >> right;
    std::vector<int> values(n);
    for (int& x : values) std::cin >> x;
    std::reverse(values.begin() + left, values.begin() + right);
    for (auto it = values.begin(); it != values.end(); ++it) {
        if (it != values.begin()) std::cout << ' ';
        std::cout << *it;
    }
    std::cout << '\n';
}
```

Sample input

5 1 4
10 20 30 40 50

Sample output

10 40 30 20 50

The five values in the sample do not all move. Index zero and index four remain unchanged. Reversing a range does not require the values to be sorted first; it simply changes their order.

Cost. Reversing the requested range takes time proportional to its length. Reading and printing the full vector still takes $O(n + 1)$, so that is the program's overall time. Storage is $O(n + 1)$.

A valid endpoint does not always identify an element

`begin()+n` is a valid end position, but reading from it is invalid. Never build an iterator beyond the end or before the beginning. Both endpoints must belong to the same sequence and occur in a valid order.

4.8 Inserting, erasing, and clearing vectors

Append at the end; insert before a position

`push_back(x)` appends an element. `insert(position, x)` inserts before the specified iterator. The insertion position can be `end()`, meaning append. Insertion changes size; assigning to an existing element does not.^{[4-3][4-6]}

Removal changes indices

`erase(position)` removes one existing element. `erase(first, last)` removes a valid half-open range. The remaining elements close the gap, so later indices change. `pop_back()` removes the last element of a nonempty vector.^{[4-3][4-6]}

Complete program – follow the changing positions

```
#include <iostream>
#include <vector>

int main() {
    std::vector<int> values{10, 20, 30};
    values.insert(values.begin() + 1, 99);
    values.erase(values.begin() + 2);
    values.push_back(40);
    values.pop_back();
    for (int i = 0; i < static_cast<int>(values.size()); ++i) {
        if (i != 0) std::cout << ' ';
        std::cout << values[i];
    }
    std::cout << '\n';
}
```

Sample input

(no input)

Sample output

10 99 30

Step

Current elements

Initial vector	10 20 30
Insert 99 before index 1	10 99 20 30
Erase the element now at index 2	10 99 30
Append 40, then remove the last element	10 99 30

The erased value is 20, not 30, because insertion has already changed the positions. Trace the current vector rather than reusing old index meanings.

Removing is not the same as returning

The vector's `pop_back` returns no element value. When you need the removed value, first copy it, then remove it.

```
if (!values.empty()) {
    int removed = values.back();
    values.pop_back();
    std::cout << removed << '\n';
}
```

Wrong example: `int removed = values.pop_back();` attempts to use a `void` result as an integer. The same “read first, remove second” pattern applies to the three adaptors later in this chapter.^{[4-3][4-16][4-17][4-19]}

Erase a position, not a value

`values.erase(7)` does not mean “remove the value seven.” The member function expects an iterator. To remove the first occurrence of seven, find its position and erase it only if it was found. To remove every occurrence, an optional later section introduces the free function `std::erase`.

The iterator returned by `erase` points to the next surviving element, or `end()` if none follows. The old erased position must not be reused as though it still identified the same item.^{[4-3][4-6]}

Read the shift cost, not just the line count

Inserting or erasing near the front may move many elements. For n values, repeatedly erasing the first one can shift roughly

$$(n-1) + (n-2) + \dots + 1 = \frac{n(n-1)}{2}$$

elements. That is quadratic work even though each source line looks simple. A deque or queue is a better match when the algorithm repeatedly removes the front.

Safe removal does not automatically mean efficient removal

An iterator loop that uses the iterator returned by `erase` can be correct, yet repeated middle erasures can still take $O(n^2)$. Building a separate filtered vector often expresses the same task in linear total time.

Pause and predict – 8

Can you call `values.erase(values.end())`? What about erasing the empty range `[values.end(), values.end())`?

Check your prediction. Erasing one element at `end()` is invalid because there is no element there. Erasing that empty range is valid and removes nothing.

4.9 Useful algorithms with vectors

A member operation and a free algorithm are different

`values.size()` is a member call on one vector. `std::sort(values.begin(), values.end())` is a free algorithm that receives a range. A vector has no `sort()` member.^{[4-1][4-8]}

For integer vectors, `std::sort` puts values in nondecreasing order. Equal values remain present. Sorting is not deduplication, and sorting modifies the supplied sequence.^[4-8]

Complete program – sort, then reverse the sorted sequence

```
#include <algorithm>
#include <iostream>
#include <vector>

void print(const std::vector<int>& values) {
    for (int i = 0; i < static_cast<int>(values.size()); ++i) {
        if (i != 0) std::cout << ' ';
        std::cout << values[i];
    }
    std::cout << '\n';
}

int main() {
    int n = 0;
    std::cin >> n;
    std::vector<int> values(n);
    for (int& x : values) std::cin >> x;
    std::sort(values.begin(), values.end());
    print(values);
    std::reverse(values.begin(), values.end());
    print(values);
}
```

Sample input

Sample output

5	-2 1 3 4 4
4 1 4 -2 3	4 4 3 1 -2

Contract. $0 \leq n \leq 200000$, with values in $[-10^9, 10^9]$. The `print` helper reads a vector by `const` reference; Section 4.10 revisits why this avoids a copy.

Sorting and reversal answer different questions

Reversing 4 1 3 gives 3 1 4; it does not sort the values. Sorting gives 1 3 4. Reversing *that sorted result* gives 4 3 1, a descending sequence.

For n integers, sorting uses $O(n \log n)$ comparisons for nontrivial sizes. Reversal performs a linear number of swaps. Thus the previous program's total time is $O(1 + n \log(n + 1))$. It explicitly stores $O(n + 1)$ elements; a sorting implementation also has its own workspace, discussed in the technical notes.^{[4-8][4-9]}

Find returns a position, not a Boolean or an index

`std::find(first, last, target)` scans for the first match and returns its iterator. If it finds no match, it returns `last`. No sorting is required.^[4-10]

```
auto it = std::find(values.begin(), values.end(), target);
if (it != values.end()) {
    std::cout << *it << '\n';
    std::cout << (it - values.begin()) << '\n';
}
```

For vector iterators, subtracting `begin()` gives the zero-based index. Compare against `end()` first; otherwise a not-found result can be confused with a legitimate location. A linear search still costs $O(n)$ in the worst case even when called through a library function.

Accumulate needs the correct starting type

```
long long total = std::accumulate(values.begin(), values.end(), 0LL);
```

`std::accumulate` starts with the supplied initial value and adds elements in sequence. That initial value determines its accumulator type. `0LL` starts a `long long`; `0` starts an `int`. A `long long` variable on the left does not retroactively widen earlier overflowing additions.^[4-11]

For an empty range, the result is the initial value. For the sums used here, zero is the correct starting value. Choosing a different initial number changes the answer.

Pause and predict – 9

You sum three values of 10^9 . Does assigning `std::accumulate(..., 0)` to a `long long` guarantee safe addition on a 32-bit-int environment?

Check your prediction. No. The accumulator is still an `int` because the starting value is 0. Start with `0LL`; then the additions are accumulated in `long long`.

A search result and a wide sum

Task. Read n , a target, and n integers. Print the target's first zero-based index or -1 , then print the sum. Here $0 \leq n \leq 200000$ and all supplied integers lie in $[-10^9, 10^9]$.

Complete program – search and accumulate

```
#include <algorithm>
#include <iostream>
#include <numeric>
#include <vector>

int main() {
    int n = 0, target = 0;
    std::cin >> n >> target;
    std::vector<int> values(n);
    for (int& x : values) std::cin >> x;
    auto it = std::find(values.begin(), values.end(), target);
    if (it == values.end()) std::cout << -1 << '\n';
    else std::cout << (it - values.begin()) << '\n';
    long long total = std::accumulate(values.begin(),
                                      values.end(), 0LL);
    std::cout << total << '\n';
}
```

Sample input

Sample output

4 7	1
5 7 7 -2	17

The target appears twice in the sample, but the first matching index is one. The sum is independent of whether the target was found. Both passes are linear, so their costs *add*: the total is $O(n + 1)$, not $O(n^2)$.

Useful algorithm questions before writing a call

Does the algorithm change the sequence? What range does it receive? Is that range valid? Does the result represent a value, an iterator, or a count? Does it require a sorted range? What is the cost of its comparisons or arithmetic?

These questions matter more than memorizing a long algorithm list. Binary-search algorithms, for example, have ordering requirements; they are not substitutes for `find` on an arbitrary unsorted vector.

4.10 Vectors and functions: copies versus references

The vector carries its own size

Unlike a built-in array parameter, a vector parameter is an object with a `size()` operation. A helper normally does not need a separate `n` parameter unless it intentionally works on only part of the vector.

The reference rules from Session 3 still apply:

```
// Function declarations, above main:
long long sum(const std::vector<int>& values); // Read the original.
void addToAll(std::vector<int>& values, int delta); // Change it.
void inspectCopy(std::vector<int> values); // Receive a separate value.
```

Passing an existing vector by value normally copies its elements into the parameter. Passing by reference does not create that element copy. A const reference prevents modifying the vector through that parameter.^{[4-2][4-30]}

Complete program – one read-only helper and one modifier

```
#include <iostream>
#include <vector>

long long sum(const std::vector<int>& values) {
    long long total = 0;
    for (int x : values) total += x;
    return total;
}

void addToAll(std::vector<int>& values, int delta) {
    for (int& x : values) x += delta;
}

int main() {
    int n = 0, delta = 0;
    std::cin >> n >> delta;
    std::vector<int> values(n);
    for (int& x : values) std::cin >> x;
    std::cout << sum(values) << '\n';
    addToAll(values, delta);
    std::cout << sum(values) << '\n';
}
```

Sample input

3 2
4 1 7

Sample output

12
18

Contract. $0 \leq n \leq 200000$, $|a_i| \leq 10^6$, and $|\text{delta}| \leq 1000$. The result is wide enough for the total.

Trace the boundary between caller and helper

For the sample, `sum` first reads 4 1 7 and returns 12. `addToAll` then changes the caller's elements to 6 3 9. The second `sum` returns 18. No second vector is needed.

Changing `addToAll`'s parameter to a by-value vector would make it change only its own copy. The caller would still have 4 1 7. This is the same distinction you learned for strings, now applied to a sequence of integers.

When an intentional copy is useful

To sort a sequence while preserving its original order, make a copy and sort that copy:

```
std::vector<int> sorted = original;
std::sort(sorted.begin(), sorted.end());
```

The n element copies add $O(n)$ time and storage. That can be exactly what the task needs. Avoiding a copy is not an absolute rule; the goal is to know when one occurs and why it is justified.

Similarly, `a == b` compares the contents of two compatible vectors, not their identities. Equal sizes and corresponding values make them equal. Lexicographic ordering compares the first differing element, with a shorter equal prefix ordered first; an optional revision note covers this pattern.^{[4-4][4-31]}

Returning a vector

A function can build a vector locally and return it by value. The returned object remains valid; it is not a reference to destroyed local storage. C++ can elide or move this return, so do not assume every return necessarily duplicates all elements.^[4-30]

Returning a reference to a local vector is different and invalid after that local object has been destroyed. For beginners, use a value return for a newly built result, and a `const` reference parameter to read an existing input.

Pause and predict – 10

A helper receives `std::vector<int> a` by value and sorts it. Will the caller's vector become sorted? What interface would allow the helper to sort the caller's vector?

Check your prediction. The caller is unchanged. Use `std::vector<int>& a` when changing the caller is part of the helper's contract.

4.11 Two-dimensional and nested vectors

A vector can contain vectors

```
std::vector<std::vector<int>>> grid(rows, std::vector<int>(cols));
```

Read the type from the inside out: each row is a `vector<int>`; `grid` is a vector of those rows. The constructor creates `rows` separate rows, each containing `cols` zero-valued integers. One row is not an alias of every other row.^[4-2]

`grid[i][j]` means element `j` of row `i`. The row index must be valid for `grid`; the column index must be valid for that particular row.

Complete program – read a grid and sum each row

```
#include <iostream>
#include <vector>

int main() {
    int rows = 0, cols = 0;
    std::cin >> rows >> cols;
    std::vector<std::vector<int>>> grid(rows, std::vector<int>(cols));
    for (auto& row : grid) {
        for (int& x : row) std::cin >> x;
    }
    for (int i = 0; i < rows; ++i) {
        long long total = 0;
        for (int x : grid[i]) total += x;
        if (i != 0) std::cout << ' ';
        std::cout << total;
    }
    std::cout << '\n';
}
```

Sample input

```
2 3
1 2 3
4 -2 8
```

Sample output

```
6 10
```

Contract. $1 \leq \text{rows}, \text{cols} \leq 300$, with each value in $[-10^9, 10^9]$. The totals use `long long`. Reading and summing takes $O(\text{rows} \cdot \text{cols})$ time and stored cells.

Why the outer loop also needs a reference

```
for (auto& row : grid) {  
    for (int& x : row) std::cin >> x;  
}
```

The outer `&` refers to an existing row. Without it, each row variable would be a copied vector, and input would modify that copy instead of the grid. The inner `&` refers to an actual integer in that row. For reading without modification, write `const auto& row`. Copying an entire row just to inspect it creates avoidable work and temporary memory.

Not every nested vector is a rectangle

```
std::vector<std::vector<int>>> groups(3);  
groups[0] = {4, 7};  
groups[1] = {9};  
groups[2] = {};
```

These are three independent groups with lengths two, one, and zero. A valid traversal uses each row's own size, or a nested range-based loop. Accessing `groups[1][1]` is invalid even though another row has two elements.

Each ordinary integer row is contiguous internally. The collection of all rows is not one guaranteed contiguous rectangular integer block. You do not need raw pointers to use this representation safely.

Count both cells and row objects

With positive rectangular dimensions, storing r rows of c integers takes $O(rc)$ space. More generally, a ragged collection with r rows and m stored integers needs $O(r + m)$ logical storage: even an empty row still has a vector object.

Pause and predict – 11

What changes when `for (auto row : grid)` is replaced by `for (auto& row : grid)`?
Can every row be indexed using `grid[0].size()`?

Check your prediction. The first loop copies each row; the second refers to it. Only use a shared column count when a rectangular shape is guaranteed. Ragged rows must use their own lengths.

4.12 Vector growth, amortized cost, and invalidation

A single append is not always constant-time

Most appends use spare capacity. An append that needs more room may allocate new storage and relocate the current elements, making that particular operation linear in the vector's size. Across a sequence of appends, vector guarantees **amortized constant-time** append behavior for these fixed-cost elements.^{[4-1][4-6]}

Amortized describes total cost spread over an operation sequence. It is not a claim about random inputs, and it does not mean every individual operation has the same cost.

A simple model for understanding the guarantee

Imagine an illustrative implementation that grows capacity geometrically, for example from 1 to 2 to 4 to 8. Relocations would involve groups of sizes

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

Before n appends have finished, the total number of such relocations is proportional to n , not n times the largest relocation. Add the n direct insertions and the total remains linear.

This is an explanation of amortization, **not a required capacity sequence**. A different growth factor can satisfy the same guarantee. Tests should check behavior and valid bounds, not a particular capacity history.

Reallocation can invalidate remembered positions

A saved iterator, pointer, or reference to a vector element may become invalid when the storage moves. With reallocation, all such element handles and the old end iterator are invalidated.^{[4-5][4-6]}

Unsafe fragment – do not use after an append that may reallocate:

```
std::vector<int> values{10, 20};
int& saved = values[0];
values.push_back(30);
// saved may now be invalid; do not use it here.
```

To keep the value, store `int saved = values[0];`. To refer to the current element again after a possible move, obtain it again from the vector using a valid current index.

In-place shifting can also invalidate handles

Even without reallocation, insertion invalidates handles at or after the insertion position, including the old end. Erasure invalidates handles at or after its erase position. Handles strictly before a non-reallocating insertion remain valid.^[4-6]

For newcomers, a practical rule is: **do not retain element references or iterators across structural vector changes unless you have checked the exact rule**. Value copies are easier to reason about when you need only the old data.

Indices do not identify permanent people

An integer index does not dangle when storage reallocates, but insertion or erasure can make that index refer to a different logical item. After removing the first element, yesterday's index one is today's index zero.

This distinction matters when a problem gives stable identifiers. Store an ID with the value, rather than assuming the current vector index is a permanent identity. A pair can help with that, as the next section explains.

Do not append while using a cached end position

A range-based loop remembers its range boundaries. Even when `reserve` prevents element relocation, appending invalidates the old past-the-end iterator. Reserving room does not make a range-based loop that appends to its own vector a safe general pattern.

To transform input into output, use separate objects:

```
std::vector<int> doubled;
for (int x : original) {
    doubled.push_back(2 * x); // Assume the product fits int.
}
```

To append based on an original prefix of the same vector, a deliberately designed indexed loop can snapshot the original length and copy each needed element before appending. Do not carry a reference through that append.

Pause and predict – 12

For n calls to `push_back` starting from an empty integer vector, is the total append cost $O(n^2)$ just because one call may take $O(n)$?

Check your prediction. No. The overall sequence of appends has $O(n)$ amortized total cost. Multiplying the worst cost of a single call by every call would ignore the growth guarantee and overestimate the tight total behavior.

4.13 Pair: two values travel together

Keep a relationship explicit

A student name and a score belong together. So do a coordinate's x and y , or an event's time and ID. A `std::pair<A, B>` stores two values, potentially of different types, in fields named `first` and `second`.^[4-12]

```
std::pair<int, int> point{2, 7};
std::pair<std::string, int> student{"Mona", 85};
```

`point.first` is two; `point.second` is seven. These are **data members**, not function calls: write `point.first`, not `point.first()`.

Complete program – named fields inside a pair

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

int main() {
    std::pair<std::string, int> student{"Mona", 85};
    student.second += 5;
    std::pair<int, int> point = std::make_pair(2, 7);
    std::cout << student.first << ' ' << student.second << '\n';
    std::cout << point.first << ' ' << point.second << '\n';
}
```

Sample input

(no input)

Sample output

Mona 90

2 7

`std::make_pair(2, 7)` is another way to construct a pair; its element types are inferred from the arguments. Direct braces after an explicit pair type are usually easiest to read while learning. A pair is not an array: `point[0]` is not its normal interface.^{[4-12][4-13]}

Name the meaning, not only the type

A `pair<int, int>` does not explain whether it stores (`score`, `ID`) or (`ID`, `score`). State that convention in your explanation and use it consistently. Reversing the fields can change sorting and priority order even though the program still compiles.

Pause and predict – 13

For `std::pair<int, int> result{90, 4}`; storing (`score`, `ID`), which field is the ID? Does calling `result.size()` make sense?

Check your prediction. The ID is `result.second`. A pair has two named fields, not a variable-size sequence; it has no `size()` member.

Return two related results from one helper

Task. Read a nonempty vector and print its minimum and maximum. Here $1 \leq n \leq 200000$, with values in $[-10^9, 10^9]$.

Complete program – return a pair of answers

```
#include <iostream>
#include <utility>
#include <vector>

std::pair<int, int> minMax(const std::vector<int>& values) {
    int low = values[0], high = values[0];
    for (int x : values) {
        if (x < low) low = x;
        if (x > high) high = x;
    }
    return {low, high};
}

int main() {
    int n = 0;
    std::cin >> n;
    std::vector<int> values(n);
    for (int& x : values) std::cin >> x;
    std::pair<int, int> answer = minMax(values);
    std::cout << answer.first << ' ' << answer.second << '\n';
}
```

Sample input

5
8 -3 5 -3 7

Sample output

-3 8

The helper's contract says **first** is the minimum and **second** is the maximum. It initializes both from an actual element, which handles all-negative inputs correctly. Nonemptiness is essential because it accesses index zero.

Why it works. After each visited value, **low** is the smallest seen so far and **high** is the largest seen so far. The returned pair keeps those two results together. Scanning uses $O(n)$ time and $O(1)$ extra working variables beyond the input vector.

Assignment and comparison

You can assign a compatible pair to another pair or assign a new brace-initialized pair value. For integer pairs, equality requires both corresponding fields to match. Ordering compares the first field first, then the second only on a tie.^{[4-12][4-13]}

That last rule deserves its own trace: it is useful for sorting and heaps, but it is also the source of many tie-handling bugs.

4.14 Vectors of pairs, comparison, and tie rules

A sequence of records with two fields each

```
std::vector<std::pair<int, int>> events;
events.push_back({3, 2});
events.push_back({1, 9});
```

The vector's element type is a pair. `events[0].first` reads the first field of the first record; `events[0].second` reads its second field. Access still requires index zero to exist.

Lexicographic order means first field, then second

For integer pairs, (1, 9) comes before (3, 1) because one is less than three. The second fields do not matter in that comparison. Between (3, 1) and (3, 2), the first fields tie, so the second fields decide.^[4-13]

Left pair	Right pair	Which comes first?
(1, 9)	(3, 1)	(1, 9)
(3, 1)	(3, 2)	(3, 1)
(2, 5)	(2, 5)	Neither is less; they are equal.

This is not a comparison of sums. For example, (1, 100) still comes before (2, 0) in ascending lexicographic order.

Tie policy belongs in the problem statement

If a record is (time, ID), ascending pair order means earlier time first, then smaller ID. It does *not* automatically mean earlier arrival among tied times. Store an arrival index as the second field when that is the required tie rule.

Likewise, (score, ID) sorted ascending puts smaller scores first. A task asking for larger scores first and smaller IDs on ties needs a different comparison. The optional section on named comparators shows how to express that rule.

Pause and predict – 14

Sort (2, 8), (1, 50), (2, 3), (1, 4) in ascending pair order. Is (1, 50) after (2, 3) because its sum is larger?

Check your prediction. The order is (1, 4), (1, 50), (2, 3), (2, 8). No: the first differing field decides. Pair ordering does not add the fields.

Sort events by their two-field key

Task. Read $0 \leq n \leq 200000$ pairs (time, ID) and print them in ascending pair order. For this ordering exercise, both fields may be any integers in $[-10^9, 10^9]$; duplicate records are allowed.

Complete program – the fields stay together during sorting

```
#include <algorithm>
#include <iostream>
#include <utility>
#include <vector>

int main() {
    int n = 0;
    std::cin >> n;
    std::vector<std::pair<int, int>> events(n);
    for (auto& event : events) {
        std::cin >> event.first >> event.second;
    }
    std::sort(events.begin(), events.end());
    for (const auto& event : events) {
        std::cout << event.first << ' ' << event.second << '\n';
    }
}
```

Sample input	Sample output
4	1 4
3 2	1 9
1 9	3 1
3 1	3 2
1 4	

The range-based input loop uses `auto&` so it fills the stored pairs. The output loop uses `const auto&` because it only reads them. Sorting moves each *whole pair*; a time does not become accidentally attached to a different ID.

What equality does not guarantee

`std::sort` is not a stable-sort guarantee: equivalent keys do not generally preserve their original order. If the complete integer pairs are identical, they look identical in this output, so no distinction is needed. If additional hidden identity matters, make the required tie rule explicit.^[4-8]

Cost. Each integer-pair comparison examines at most two fixed-width integers, so it has constant cost. Sorting therefore needs $O(n \log n)$ comparisons for positive nontrivial sizes. A pair containing long strings can have more expensive comparisons; include the character work when relevant.

4.15 Deque: two ends and indexed access

A double-ended sequence

A `std::deque<int>` supports indexed access and efficient insertion/removal at both ends. It is useful when “front” and “back” both matter. It is commonly pronounced “deck.”^[4-14]

Operation	Meaning
<code>push_front(x)</code>	Add <code>x</code> before the current first element.
<code>push_back(x)</code>	Add <code>x</code> after the current last element.
<code>pop_front()</code>	Remove the first element; requires nonempty.
<code>pop_back()</code>	Remove the last element; requires nonempty.
<code>front()</code> , <code>back()</code>	Access an end element; require nonempty.
<code>d[i]</code> , <code>d.at(i)</code>	Access an existing indexed element; <code>at</code> checks bounds.
<code>size()</code> , <code>empty()</code>	Inspect the current state.

The core end operations have constant-time guarantees in the standard container-operation model; middle insertion/erasure is linear in the worst case. Random indexed access is constant-time.^{[4-14][4-15]}

Trace the two ends

Start with 20 30. After `push_front(10)`, the sequence is 10 20 30. After `push_back(40)`, it is 10 20 30 40. A front removal then removes ten; a back removal removes forty.

A deque is not a priority structure. Its front is not necessarily the smallest value, and its back is not necessarily the largest. Their meanings come from your insertions and removals.

Pause and predict – 15

A deque is 5 8. Apply `push_front(3)`, `push_back(1)`, then `pop_front()`. What are the elements, front, and back?

Check your prediction. The elements are 5 8 1. The front is five and the back is one. Numerical size does not determine either end.

A complete deque example

This program takes no input. It demonstrates both ends, followed by an indexed update.

Complete program – add and remove at both ends

```
#include <deque>
#include <iostream>

int main() {
    std::deque<int> line{20, 30};
    line.push_front(10);
    line.push_back(40);
    std::cout << line.front() << ' ' << line.back() << '\n';
    line.pop_front();
    line.pop_back();
    line[1] = 99;
    std::cout << line[0] << ' ' << line[1] << '\n';
}
```

Sample input

(no input)

Sample output

10 40

20 99

After the removals, `line` contains 20 30. Assigning `line[1] = 99` changes the second existing element; it does not append another one.

Deque is not just vector with one more member

A vector's ordinary elements are contiguous. A deque does not promise one contiguous element block; implementations can use multiple storage blocks. Do not treat it as a single raw array. Deque also has no vector-style `capacity()` or `reserve()` member.^{[4-1][4-14]}

Its iterators support random access, so algorithms such as `sort(d.begin(), d.end())` are available when their other requirements hold. You do not need to sort a deque merely to use its ends.

Iterator and reference rules differ

End insertions invalidate deque iterators but do not invalidate references to existing elements. This differs from vector reallocation. Middle changes have stronger invalidation effects. For ordinary beginner simulations, obtain `front()` or `back()` when needed, copy a value before removing it, and avoid keeping iterators across modifications.^[4-15]

Same logical item, new index

Pushing at the front shifts the logical indices: the old first item is now at index one. An element reference can remain valid while its index changes. Index, value, and identity are different ideas.

4.16 A deque simulation: keep only the latest values

Maintain a small recent-history buffer

Suppose values arrive one at a time and only the latest k matter. Keep the oldest retained value at the front and the newest at the back. On each arrival, append it. If the buffer becomes too large, remove the oldest retained value.

For $k = 3$, the arrivals 8 2 7 4 9 produce:

Arrival	Retained values, oldest to newest
8	8
2	8 2
7	8 2 7
4	2 7 4
9	7 4 9

Complete program – append the new value, discard the oldest

```
#include <deque>
#include <iostream>

int main() {
    int n = 0, k = 0;
    std::cin >> n >> k;
    std::deque<int> recent;
    for (int i = 0; i < n; ++i) {
        int x = 0;
        std::cin >> x;
        recent.push_back(x);
        if (static_cast<int>(recent.size()) > k) {
            recent.pop_front();
        }
    }
    for (int i = 0; i < static_cast<int>(recent.size()); ++i) {
        if (i != 0) std::cout << ' ';
        std::cout << recent[i];
    }
    std::cout << '\n';
}
```

Sample input

5 3
8 2 7 4 9

Sample output

7 4 9

Contract. $0 \leq n, k \leq 200000$; values are in $[-10^9, 10^9]$. If k is zero, the output is a blank line. If k exceeds the number of arrivals, all arrivals remain.

Why one removal is enough

Before a new arrival, the deque contains at most k values. One append can make it contain at most $k+1$. Therefore one conditional removal is enough to restore the bound. A `while` loop would also work, but repeated removals cannot be necessary after this single append.

The invariant is precise: after processing i arrivals, the deque contains the last $\min(i, k)$ of them, in the order they arrived. This statement proves both the contents and the order of the final answer.

Handle $k = 0$ deliberately

With $k = 0$, each append creates one element and the following removal immediately removes it. No empty access occurs: `pop_front()` happens only after that append and only when the size is too large.

This boundary case is a useful test of the algorithm's logic. Do not special-case it unless doing so makes the code clearer; the ordinary update already handles it.

Count arrivals and removals

There are n appends, at most n front removals, and at most $\min(n, k)$ printed values. The total is $O(n + 1)$ time and $O(\min(n, k) + 1)$ storage, including the temporary extra value before removal.

Using a vector and repeatedly erasing its first element expresses the same logical result, but each erase can shift the retained sequence. With a large k , that can do much more work. The deque matches the required end operations directly.

Pause and predict – 16

Five values arrive, but $k = 0$. Is calling `pop_front()` in this algorithm invalid because the final deque is always empty?

Check your prediction. No. The algorithm first appends one value, so the deque is nonempty when it removes that value. The empty final state is intentional.

4.17 Queue: first in, first out

Serve the oldest waiting item

A queue follows **FIFO**: first in, first out. New items join at the back; the next item to remove is at the front. The default `std::queue<T>` uses a deque internally, but deliberately hides general indexing and iteration.^[4-16]

Operation	Meaning
<code>push(x)</code>	Add an item at the back.
<code>front()</code>	Read/access the next item to leave. Requires nonempty.
<code>back()</code>	Read/access the newest queued item. Requires nonempty.
<code>pop()</code>	Remove one front item. Requires nonempty; returns no value.
<code>size()</code> , <code>empty()</code>	Inspect the item count or emptiness.

There is no `q[0]`, `q.begin()`, or `q.top()` in this interface. Use `front` for a queue. The restricted interface is useful because it makes the intended service rule explicit.^[4-16]

Complete program – arrival order controls removal

```
#include <iostream>
#include <queue>

int main() {
    std::queue<int> waiting;
    waiting.push(4);
    waiting.push(9);
    waiting.push(2);
    std::cout << waiting.front() << ' ' << waiting.back() << '\n';
    waiting.pop();
    std::cout << waiting.front() << ' ' << waiting.size() << '\n';
}
```

Sample input

(no input)

Sample output

4 2

9 2

Four arrived first, so it leaves first even though nine is larger and two arrived more recently. The default queue operations used here take constant time for fixed-width elements, following the underlying deque operations.^{[4-14][4-16]}

Draw the front and back, not an imaginary sorted list

Action	Front to back	Next to leave
push(4)	4	4
push(9)	4 9	4
push(2)	4 9 2	4
pop()	9 2	9
push(7)	9 2 7	9

A queue stores duplicate values as separate items. If two customers happen to have the same numeric label, two pushes still add two items. A data structure does not decide whether duplicate IDs are meaningful; the problem's contract does.

Read first, then remove

```
if (!waiting.empty()) {
    int next = waiting.front();
    waiting.pop();
    std::cout << next << '\n';
}
```

Do not write `auto& next = waiting.front();` and then access that reference after `pop()`. The referenced front element has been removed. A value copy, as above, remains usable.

Processing a whole queue consumes it

```
while (!waiting.empty()) {
    std::cout << waiting.front() << '\n';
    waiting.pop();
}
```

After this loop, `waiting` is empty. Printing does not require removal by itself, but using this interface to reach each later element does. Copy the queue first when the original contents must remain.

Pause and predict – 17

A queue contains 7 1 7, front to back. After one `pop()`, how many sevens remain, and what is the front?

Check your prediction. One seven remains. The front becomes one. One removal deletes exactly one queued item, not every equal value.

4.18 Queue simulations: commands and repeated turns

Translate an action into one state change

Task. Process $0 \leq q \leq 200000$ commands. 1 id joins the line. 2 prints and removes the front ID. 3 prints the front without removing it. For either request on an empty line, print **EMPTY**. IDs are integers in $[-10^9, 10^9]$ in this mechanical simulation.

Complete program – a service-line command processor

```
#include <iostream>
#include <queue>

int main() {
    int q = 0;
    std::cin >> q;
    std::queue<int> waiting;
    while (q--) {
        int type = 0;
        std::cin >> type;
        if (type == 1) {
            int id = 0;
            std::cin >> id;
            waiting.push(id);
        } else if (waiting.empty()) {
            std::cout << "EMPTY\n";
        } else {
            std::cout << waiting.front() << '\n';
            if (type == 2) waiting.pop();
        }
    }
}
```

Sample input

```
7
2
1 10
1 20
3
2
2
3
```

Sample output

```
EMPTY
10
10
20
EMPTY
```

The first branch handles an arrival. The second branch handles *all empty requests before access*. Only the final branch reads `front()`. The input guarantees that command types are one, two, or three.

A command can put unfinished work back in line

Imagine each job needs several one-unit turns. Take the front job, perform one unit of work, and either finish it or move it to the back. A pair keeps its ID with its remaining work.

For workloads 2 1 3, the initial queue is (1,2), (2,1), (3,3). Job two finishes on its first turn. Job one finishes on its second turn. Job three finishes last. Completion order is 2 1 3, not simple arrival order.

Complete program – requeue unfinished jobs

```
#include <iostream>
#include <queue>
#include <utility>

int main() {
    int n = 0;
    std::cin >> n;
    std::queue<std::pair<int, int>> jobs;
    for (int id = 1; id <= n; ++id) {
        int work = 0;
        std::cin >> work;
        jobs.push({id, work});
    }
    bool first = true;
    while (!jobs.empty()) {
        auto job = jobs.front();
        jobs.pop();
        --job.second;
        if (job.second == 0) {
            if (!first) std::cout << ' ';
            std::cout << job.first;
            first = false;
        } else {
            jobs.push(job);
        }
    }
    std::cout << '\n';
}
```

Sample input

3
2 1 3

Sample output

2 1 3

Contract. $0 \leq n \leq 1000$; each workload is between one and one hundred. IDs are the one-based input positions. A zero-job input prints a blank line. Positive workloads are required: decrementing a zero workload would break this termination rule.

Prove that the repeated-turn loop finishes

The total remaining work is a nonnegative integer. Each iteration reduces it by exactly one. Finished jobs are not reinserted. Therefore the loop eventually empties the queue after exactly the sum of the initial workloads in iterations.

The queue maintains the order of jobs waiting for their *next* turn. A job is copied before removal, its copy is updated, and that updated value is inserted at the back only if more work remains. No removed-element reference is retained.

A while loop over a queue is not automatically $O(n)$

Let $W = \sum_i w_i$, where w_i is the initial work of job i . The simulation takes $O(n + W + 1)$ time and $O(n + 1)$ storage. The number of original jobs alone does not determine how many turns occur.

This is the same analysis habit introduced in Session 3: count actual progress. One loop can be expensive when it repeatedly reintroduces work; nested loops can be efficient when each item is processed only a bounded number of times.

Do not compare a counter with a shrinking size

Wrong draining pattern:

```
for (int i = 0; i < static_cast<int>(waiting.size()); ++i) {  
    waiting.pop();  
}
```

With four items, the first removal leaves three, and the second leaves two. The loop then tests $2 < 2$ and stops with two items still present. Use `while (!waiting.empty())` to drain a queue, or snapshot an original count when exactly one original layer of work is intended.

Pause and predict – 18

Two jobs need 100 and 1 one-unit turns. How many iterations does the repeated-turn loop perform? Is “two jobs, so two iterations” correct?

Check your prediction. It performs 101 iterations: one per unit of required work. The second job finishes early, but the first still needs all one hundred turns.

4.19 Priority queue: select the greatest value first

Priority, not arrival order

A default `std::priority_queue<int>` exposes the greatest stored integer at `top()`. Pushing a new value updates that priority relation. Removing the top reveals the next greatest remaining value. Duplicate values are kept as separate items.^{[4-17][4-18]}

Operation	Meaning
<code>push(x)</code>	Insert a value and maintain the priority rule.
<code>top()</code>	Read the highest-priority value; requires <code>nonempty</code> .
<code>pop()</code>	Remove one highest-priority value; requires <code>nonempty</code> .
<code>size()</code> , <code>empty()</code>	Inspect count or emptiness.

The interface has no indexed access, normal iteration, `front()`, or `back()`. `top()` returns read-only access: you cannot directly assign a new priority into the top element through it.^[4-17]

A heap is not a completely sorted array

The adaptor maintains a **heap** ordering internally. For our purposes, the essential guarantee is that the next extreme value is available at the top. The remaining stored elements are not promised to be arranged as a completely sorted sequence.

A useful mental picture is a competition where the current winner is easy to retrieve. When a winner is removed or a competitor is added, the structure repairs enough relationships to find the new winner; it does not sort everything from scratch.

Pause and predict – 19

Insert 4, then 9, then 2 into a default integer priority queue. What is at the top? After removing it, what is at the top?

Check your prediction. Nine is initially at the top. After removing that nine, four becomes the top. The smallest value, two, waits longer regardless of when it arrived.

No arbitrary deletion or priority updates in this interface

A priority queue does not provide “erase this middle item” or “change the priority of item ID seven.” Do not invent such a member. Those tasks need an additional technique or a different data structure, beyond this session’s core examples.

Repeated extraction gives descending order

Task. Read $0 \leq n \leq 200000$ integers in $[-10^9, 10^9]$ and print them from greatest to smallest by extracting from a max-heap.

Complete program – a default max-priority queue

```
#include <iostream>
#include <queue>

int main() {
    int n = 0;
    std::cin >> n;
    std::priority_queue<int> values;
    for (int i = 0; i < n; ++i) {
        int x = 0;
        std::cin >> x;
        values.push(x);
    }
    bool first = true;
    while (!values.empty()) {
        if (!first) std::cout << ' ';
        std::cout << values.top();
        values.pop();
        first = false;
    }
    std::cout << '\n';
}
```

Sample input

5
4 9 2 9 -1

Sample output

9 9 4 2 -1

Every extraction removes exactly one copy of the largest remaining value. The sample contains two nines, so both are printed. When no values were inserted, the loop is skipped and the final output line is blank.

Understand the cost carefully

`top()` is constant-time. Heap repair for an insertion or removal uses logarithmic comparisons. The default backing vector can reallocate on an individual push, adding linear movement work; over a sequence of pushes the usual logarithmic bound is amortized.^{[4-1][4-17][4-18]}

Building by n individual pushes and then extracting all n integers takes $O(1 + n \log(n + 1))$ total time and $O(n + 1)$ stored elements in this model. This is not automatically better than sorting once. The main advantage appears when insertions and requests for the next extreme are interleaved.

Use a priority queue when the set keeps changing

Task. Commands have the same format as the earlier line simulation, but 2 prints and removes the *largest* stored bid, and 3 prints that bid without removing it. 1 x adds a bid. Use `EMPTY` for an empty request. The bounds are $0 \leq q \leq 200000$, $|x| \leq 10^9$.

Complete program – arrivals interleaved with maximum requests

```
#include <iostream>
#include <queue>

int main() {
    int q = 0;
    std::cin >> q;
    std::priority_queue<int> bids;
    while (q-- > 0) {
        int type = 0;
        std::cin >> type;
        if (type == 1) {
            int x = 0;
            std::cin >> x;
            bids.push(x);
        } else if (bids.empty()) {
            std::cout << "EMPTY\n";
        } else {
            std::cout << bids.top() << '\n';
            if (type == 2) bids.pop();
        }
    }
}
```

Sample input

```
7
1 4
1 9
3
2
1 7
2
2
```

Sample output

```
9
9
7
4
```

Only the required processing order changed: a FIFO queue would return the oldest bid, while this adaptor returns the largest. The surrounding command parser is intentionally similar so the data-structure difference is visible.

The total time is $O(1 + q \log(q + 1))$ as a safe amortized upper bound, with $O(q + 1)$ storage. There can be fewer live items than commands, but q bounds the maximum possible count.

4.20 Min-heaps and priority pairs

Select the smallest value instead

A min-heap exposes the smallest value at `top()`. Its declaration explicitly supplies the comparison type:

```
std::priority_queue<int, std::vector<int>, std::greater<int>> values;
```

The first type is the element type. The second is the backing sequence type. The third is the comparator type. `std::greater<int>` changes the ordering so that the smallest integer has highest priority. Include `<functional>`, `<queue>`, and `<vector>`.^{[4-17][4-28]}

Complete program – smallest value first

```
#include <functional>
#include <iostream>
#include <queue>
#include <vector>

int main() {
    int n = 0;
    std::cin >> n;
    std::priority_queue<int, std::vector<int>, std::greater<int>> values;
    for (int i = 0; i < n; ++i) {
        int x = 0;
        std::cin >> x;
        values.push(x);
    }
    bool first = true;
    while (!values.empty()) {
        if (!first) std::cout << ' ';
        std::cout << values.top();
        values.pop();
        first = false;
    }
    std::cout << '\n';
}
```

Sample input

5
4 9 2 9 -1

Sample output

-1 2 4 9 9

Contract. $0 \leq n \leq 200000$, values in $[-10^9, 10^9]$. The code differs from the max-heap example chiefly in its declaration; `top`, `push`, and `pop` keep their interface names.

Choose a smaller time, then a smaller ID

A pair in a priority queue follows pair comparison. A min-heap of (time, ID) therefore exposes the earliest time and, on a tie, the smaller ID.^{[4-13][4-17]}

using Event = std::pair<int, int>; gives the existing pair type a shorter name; it does not define a new structure or change its comparison rules.^[4-32]

Complete program – a min-heap of two-field priorities

```
#include <functional>
#include <iostream>
#include <queue>
#include <utility>
#include <vector>

int main() {
    int n = 0;
    std::cin >> n;
    using Event = std::pair<int, int>;
    std::priority_queue<Event, std::vector<Event>,
                      std::greater<Event>> events;
    for (int i = 0; i < n; ++i) {
        int time = 0, id = 0;
        std::cin >> time >> id;
        events.push({time, id});
    }
    while (!events.empty()) {
        Event event = events.top();
        events.pop();
        std::cout << event.first << ' ' << event.second << '\n';
    }
}
```

Sample input

4
5 3
2 8
5 1
2 4

Sample output

2 4
2 8
5 1
5 3

Contract. $0 \leq n \leq 200000$, $0 \leq \text{time} \leq 10^9$, and unique IDs in $[1, 200000]$. Each pair is read as time ID. The output contains both fields. IDs four and eight tie on time in the sample, so ID four is extracted first.

Contrast the default max-heap of pairs

The default max-heap exposes the lexicographically greatest pair: **larger first field, then larger second field**. It does not automatically choose a smaller ID on tied scores.^{[4-13][4-17]}

Complete program – both fields follow descending pair order

```
#include <iostream>
#include <queue>
#include <utility>

int main() {
    int n = 0;
    std::cin >> n;
    std::priority_queue<std::pair<int, int>> values;
    for (int i = 0; i < n; ++i) {
        int first = 0, second = 0;
        std::cin >> first >> second;
        values.push({first, second});
    }
    while (!values.empty()) {
        auto item = values.top();
        values.pop();
        std::cout << item.first << ' ' << item.second << '\n';
    }
}
```

Sample input

4
5 3
2 8
5 1
2 4

Sample output

5 3
5 1
2 8
2 4

Contract. $0 \leq n \leq 200000$; both fields are in $[-10^9, 10^9]$. Compare this output with the previous min-heap output using the same four pairs.

For a max-priority task that wants greater score but *smaller* positive ID on ties, one possible bounded numeric key is **(score, -ID)**; recover the original ID by negating the second field. Only use such transformations when negation is representable and you have verified the tie direction. They are not universal replacements for understanding comparison.

Pause and predict – 20

A default max-heap contains (90, 2) and (90, 5). Which is at the top? Which one would a min-heap of the same pair type expose?

Check your prediction. The max-heap exposes (90, 5). The min-heap exposes (90, 2). When the first fields tie, the second field decides in the same direction as the pair order.

4.21 Stack: last in, first out

The newest unfinished item is on top

A stack follows **LIFO**: last in, first out. New items are pushed onto the top; the next removal is from that same top. The default `std::stack<T>` uses a deque internally.^[4-19]

Operation	Meaning
<code>push(x)</code>	Place a new value on top.
<code>top()</code>	Access the newest remaining value; requires nonempty.
<code>pop()</code>	Remove that top value; requires nonempty and returns no value.
<code>size()</code> , <code>empty()</code>	Inspect count or emptiness.

There is no general indexing or normal iteration in the stack interface. There are also no `front()` or `back()` members. All the default stack operations above have constant cost for fixed-width elements, inherited from the deque's back operations.^{[4-14][4-19]}

Complete program – the last arrival leaves first

```
#include <iostream>
#include <stack>

int main() {
    std::stack<int> plates;
    plates.push(4);
    plates.push(9);
    plates.push(2);
    std::cout << plates.top() << ' ' << plates.size() << '\n';
    plates.pop();
    std::cout << plates.top() << '\n';
}
```

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

(no input)	2 3 9
------------	----------

The newest value is two, so it is on top. Removing it exposes nine. This is not a greatest-value rule: had the last pushed value been negative, it would still be on top.

Pause and predict – 21

Push 6, 1, and 8, then pop once and push 3. What is the top? What is the next value after one more pop?

Check your prediction. The top is three. Removing it exposes one. Six is still below that one.

A stack naturally reverses removal order

Task. Read $0 \leq n \leq 200000$ integers in $[-10^9, 10^9]$, push them in input order, and print them while removing them.

Complete program – use LIFO to reverse a sequence

```
#include <iostream>
#include <stack>

int main() {
    int n = 0;
    std::cin >> n;
    std::stack<int> values;
    for (int i = 0; i < n; ++i) {
        int x = 0;
        std::cin >> x;
        values.push(x);
    }
    bool first = true;
    while (!values.empty()) {
        if (!first) std::cout << ' ';
        std::cout << values.top();
        values.pop();
        first = false;
    }
    std::cout << '\n';
}
```

Sample input

4
8 3 9 1

Sample output

1 9 3 8

Every value is pushed once and popped once. Therefore time and stored elements are both $O(n + 1)$. The program consumes the stack while printing; no values remain afterward.

Choose an interface, not only a possible implementation

A vector can also act like a stack using `push_back`, `back`, and `pop_back`. This is useful when you need indexed access or a directly printable result as well as stack-like behavior. A `std::stack` makes the LIFO restriction explicit.

Neither choice means that reversing a known stored vector requires an extra stack. For that simpler job, a backward loop or `std::reverse` may be enough. Use the stack when the algorithm needs to remember and resolve the newest unfinished work.

The program call stack and `std::stack` are different objects

Session 3's optional recursion discussion described the implementation's call stack. `std::stack` is a container adaptor that your program creates explicitly. The common LIFO idea does not make them the same object or give one access to the other.

4.22 Stack patterns: brackets, cancellation, and undo

Brackets must close in reverse opening order

For $([{}])$, the last opening bracket seen is the first that must close. A stack stores those unmatched opening brackets. An opening bracket is pushed; a closing bracket must match the top and then remove it.

Counting how many opening and closing brackets occur is insufficient. $([])$ has matching totals for both types, yet its nesting is wrong. The top-of-stack rule detects the mismatch when $)$ tries to close $[$.

Character	Unmatched openings, bottom to top	Action
(	(	Push.
[	([	Push.
]	(	Match the top, then pop.
)	Empty	Match the top, then pop.

Three checks define correctness

A closing bracket cannot appear when the stack is empty. Its type must match the latest unmatched opening bracket. At the end, the stack must be empty so that no openings remain unclosed.

The invariant is: after a valid prefix, the stack stores exactly the unmatched opening brackets in their opening order. The top is therefore the only opening that the next close is allowed to match.

Pause and predict – 22

Why is $((()$ rejected even though no closing bracket ever has the wrong type? Why is $)()$ rejected?

Check your prediction. $((()$ leaves one unmatched opening at the end. $)()$ tries to close before anything is open. Both prefix validity and final emptiness are necessary.

Check three bracket types

Task. Read one full line consisting only of `()`, `[]`, and `{}` characters, of length zero through 200000. Print YES if it is balanced, otherwise NO. The empty line is balanced.

Complete program – match the most recent unmatched opening

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

bool matches(char open, char close) {
    return (open == '(' && close == ')') ||
           (open == '[' && close == ']') ||
           (open == '{' && close == '}');
}

bool balanced(const std::string& text) {
    std::stack<char> pending;
    for (char c : text) {
        if (c == '(' || c == '[' || c == '{') {
            pending.push(c);
        } else {
            if (pending.empty() || !matches(pending.top(), c)) {
                return false;
            }
            pending.pop();
        }
    }
    return pending.empty();
}

int main() {
    std::string text;
    std::getline(std::cin, text);
    std::cout << (balanced(text) ? "YES\n" : "NO\n");
}
```

Sample input

```
{[()]}
```

Sample output

```
YES
```

The condition uses short-circuit evaluation: when `pending.empty()` is true, the right side of `||` is not evaluated, so `top()` is not called on an empty stack.

Each character causes at most one push or pop, so time is $O(n + 1)$. The worst-case stack size is n , as in a prefix of only openings. The helper therefore uses $O(n + 1)$ additional storage beyond the input string.

Cancel adjacent equal characters

Task. Read a lowercase ASCII line of length zero through 200000. Repeatedly remove adjacent equal pairs using a left-to-right scan. Print the remaining text, or **EMPTY** if nothing remains.

Keep the reduced prefix on a stack. When the next character equals the top, remove that top instead of storing the new character. Otherwise, push the character.

Complete program – reduce a prefix as each character arrives

```
#include <algorithm>
#include <iostream>
#include <stack>
#include <string>

int main() {
    std::string text;
    std::getline(std::cin, text);
    std::stack<char> kept;
    for (char c : text) {
        if (!kept.empty() && kept.top() == c) kept.pop();
        else kept.push(c);
    }
    std::string answer;
    while (!kept.empty()) {
        answer.push_back(kept.top());
        kept.pop();
    }
    std::reverse(answer.begin(), answer.end());
    std::cout << (answer.empty() ? "EMPTY" : answer) << '\n';
}
```

Sample input

abbaca

Sample output

ca

A stack prints its contents in reverse when popped. The program therefore builds a temporary answer from top to bottom, then reverses it to restore the required left-to-right order.

Why cancellation can reveal another cancellation

Trace `abbaca`:

Next character	Reduced prefix, bottom to top	Reason
a	a	Nothing to match.
b	ab	Different from top.
b	a	Cancel the two b's.
a	Empty	Newly adjacent a's cancel.
c	c	Nothing to match.
a	ca	Different from top.

The reduced prefix has no adjacent equal pair. Appending a different character preserves that property; an equal character removes the only possible new boundary pair. This invariant explains why the algorithm does not need to restart a full scan after every deletion.

Each character is pushed at most once and popped at most once. Building and reversing the answer also takes linear total work. Time and peak extra storage are $O(n + 1)$, even though several stages appear in the code.

Undo uses the same newest-first rule

Suppose commands add amounts to a running total and an undo must cancel the most recent uncanceled addition. A stack can hold the added amounts. Adding pushes an amount and increases the total. Undo reads the top amount, subtracts it, and then pops it.

A negative addition is still an action: undoing an addition of `-3` increases the total by three. The practice problem on undo checks this case and specifies that undo on an empty history changes nothing.

Pause and predict – 23

What remains after reducing `abba`? A program only removes the first `bb` and stops with `aa`. What did it miss?

Check your prediction. Nothing remains, so the required output is `EMPTY`. Removing the b's creates a new adjacent a-pair. The stack's reduced-prefix state handles that next cancellation naturally.

4.23 Copies, fresh states, and multiple test cases

Copy before consuming when the original must remain

Copying a queue copies its contained values; popping from the copy does not pop from the original. The same value-copy idea applies to the other containers and adaptors used here.^{[4-2][4-4][4-16][4-17][4-19]}

Task. Read $0 \leq n \leq 200000$ integers in $[-10^9, 10^9]$. Print them by draining a *copy* of the queue. Then print the original queue's size and its front value, or **EMPTY** when it has none.

Complete program – preserve the original queue

```
#include <iostream>
#include <queue>

int main() {
    int n = 0;
    std::cin >> n;
    std::queue<int> original;
    for (int i = 0; i < n; ++i) {
        int x = 0;
        std::cin >> x;
        original.push(x);
    }
    auto copy = original;
    while (!copy.empty()) {
        if (copy.size() != original.size()) std::cout << ' ';
        std::cout << copy.front();
        copy.pop();
    }
    std::cout << '\n' << original.size() << '\n';
    if (original.empty()) std::cout << "EMPTY\n";
    else std::cout << original.front() << '\n';
}
```

Sample input

3
5 2 8

Sample output

5 2 8
3
5

The copy takes $O(n)$ time and extra storage for n fixed-width elements. Draining it takes another $O(n)$ time. Printing a priority queue by copying and draining would also pay the heap-removal costs; a copy does not make them disappear.

Pause and predict – 24

A queue is declared outside the test-case loop and never emptied. What can happen in the next case? Does `q.empty()`; repair the problem?

Check your prediction. Old items may be mixed with the new case's items. `empty()` only inspects the state; it does not remove anything. Use a fresh per-case queue or deliberately remove the old contents.

A local object starts fresh on each call

An easy way to prevent one test case from leaking into the next is to declare its container inside `solve()`. Every call creates a new object; the old call's local object is destroyed when the call ends.

Complete program – independent per-case state

```
#include <iostream>
#include <vector>

void solve() {
    int n = 0;
    std::cin >> n;
    std::vector<int> values(n);
    long long total = 0;
    for (int& x : values) {
        std::cin >> x;
        total += x;
    }
    std::cout << values.size() << ' ' << total << '\n';
}

int main() {
    int tests = 0;
    std::cin >> tests;
    while (tests--) solve();
}
```

Sample input

3
2
5 7
0
1
-4

Sample output

2 12
0 0
1 -4

Contract. $0 \leq T \leq 200000$, $0 \leq n_i \leq 200000$, $\sum n_i \leq 200000$, and each element is in $[-10^9, 10^9]$. Print `size sum` for each test case. Zero-length cases still produce a line.

Resetting is work, not magic

`clear()` removes vector/deque elements but can leave allocated room available for reuse. The three adaptors have no `clear()` member. You can repeatedly pop until empty, assign a fresh empty object, or simply give each case a local object. Destruction or clearing may visit the old elements; do not call it constant-time merely because the source statement is short.

The sample program takes $O(T + \sum n_i)$ total time and $O(\max n_i + 1)$ peak logical container storage. Sequential test cases do not keep all their local vectors alive simultaneously.

4.24 Choose the tool and count the actual operations

Start with the next required action

What the problem repeatedly asks you to do	A natural first choice
Read or change arbitrary indexed values; append at the end.	<code>vector</code> .
Keep two related fields together when moving, returning, or sorting data.	<code>pair</code> , often as another tool's element type.
Insert/remove at either end, possibly with indexed access.	<code>deque</code> .
Take the oldest waiting item next.	<code>queue</code> .
Take the greatest or smallest priority next while values arrive.	<code>priority_queue</code> .
Resolve or undo the most recent unfinished item.	<code>stack</code> .
Print all fixed input values sorted, without interleaved updates.	A vector plus <code>sort</code> is often simpler than a heap.

These are algorithm-design starting points, not claims that only one implementation is possible. A vector can act as a stack; a deque can act as a queue. Prefer the representation that expresses the required rule without unnecessary operations.

State the invariant before the loop

For a queue: “the front is the oldest unserved item.” For the recent-history deque: “these are the last at most k arrivals.” For the bracket stack: “these are the unmatched openings.” For a priority queue: “the top is the best currently stored key under the required comparison.”

An invariant ties each operation to correctness. Without it, a program may use the correct container name but still pop at the wrong time, keep the wrong tie order, or preserve items that should already have been removed.

Pause and predict – 25

A task says “remove the smallest number seen so far” after every few arrivals. Would a FIFO queue work simply because values arrive over time?

Check your prediction. No. Arrival order is not the requested removal rule. A min-priority queue matches repeated minimum selection; a FIFO queue only knows which item arrived first.

Operation costs for the types used in this chapter

Here n is the number of currently stored fixed-cost elements. These bounds assume valid operations and the default backing containers. “Amortized” refers to a whole operation sequence, not to random input.^{[4-1][4-3][4-4][4-6][4-14][4-15][4-16][4-17][4-18][4-19]}

Tool	Operation	Cost
Vector	Index access; valid front/back; size/empty	$O(1)$
Vector	Append one element	$O(1)$ amortized; $O(n + 1)$ for one reallocating call
Vector	Remove last element	$O(1)$
Vector	Insert/erase at an arbitrary position	$O(n + 1)$ worst case
Vector	Copy, scan, or clear n elements	$O(n + 1)$ upper bound
Deque	Index access; single-element end operations	$O(1)$
Deque	Middle insertion/erasure	$O(n + 1)$ worst case
Queue	Push, pop, front, back, size, empty	$O(1)$
Stack	Push, pop, top, size, empty	$O(1)$
Priority queue	Top, size, empty	$O(1)$
Priority queue	Pop	$O(\log(n + 1))$
Priority queue	Push	$O(\log(n + 1))$ amortized; a vector reallocation can make one call linear
Integer pair	Field access or pair comparison	$O(1)$

Read the assumptions behind a table

Copying a very long string is not a constant-cost element operation. Comparing two strings can inspect many characters. A container of strings or vectors therefore needs a more detailed analysis than blindly substituting n into an integer-container table.

Similarly, `reserve(1000000)` can allocate much more storage than a current size of three suggests. State capacity or chosen allocation bounds when they dominate the memory used. Clearing a vector does not necessarily release that storage.

Three comparisons that prevent common time-limit mistakes

Front removal. Repeatedly erase index zero of a vector of n elements: up to $O(n^2)$ shifted elements. Remove from a queue's front n times: $O(n)$ operations with the default deque.

Greatest item. Scan every remaining vector value before each of n removals: up to quadratic scanning. Use a max-heap with individual insertions and removals: $O(n \log n)$ total amortized time. If all data is known and only sorted output is needed, sort once and traverse instead.

Undo. Recompute a total by scanning the full history after every command: potentially quadratic. Maintain a running total and a stack of uncanceled additions: constant work per command, hence linear total time.

Printing and copying still count

One call to `top()` is constant-time, but printing all heap values requires repeated removal and costs $O(n \log n)$. Calling a helper q times with a by-value vector of n integers can add $O(qn)$ copy work even when the helper only asks for `size()`.

A reference avoids that copy, but it does not make a scan inside the helper constant-time. Separate parameter-passing cost from body cost, just as in Session 3.

Peak live storage is the relevant quantity

A vector of n inputs, a copied vector of n inputs, and a stack holding k values together use $O(n + k)$ logical storage. When sequential test cases destroy their local containers, their sizes do not all add to peak storage.

For problems that stream values and never keep the full input, the container may dominate all memory. The optional largest- k example next uses only $O(k)$ stored values rather than $O(n)$.

4.25 Optional extensions: convenient syntax and algorithms

Emplace and structured bindings

The core examples use `push_back({id, score})` to append a pair. `emplace_back(id, score)` constructs the new pair from those arguments in the vector. Both express a valid append; emplacement is not an automatic guarantee of a faster program.^{[4-3][4-6]}

A **structured binding** gives names to the two fields. `auto [id, score] = pair;` binds names to a copied pair value; `auto& [id, score] = pair;` refers to the existing pair. A const-reference binding gives read-only access without an element copy.^[4-21]

Complete program – copies and references still differ

```
#include <iostream>
#include <utility>
#include <vector>

int main() {
    std::vector<std::pair<int, int>> results;
    results.emplace_back(7, 80);
    auto [id, score] = results[0];
    score += 5;
    std::cout << id << ' ' << score << '\n';
    std::cout << results[0].second << '\n';
    auto& [storedId, storedScore] = results[0];
    storedScore += 10;
    std::cout << storedId << ' ' << storedScore << '\n';
}
```

Sample input

(no input)

Sample output

7 85

80

7 90

The copied score becomes eighty-five while the stored score remains eighty. The reference binding then changes the stored score to ninety. Do not keep those bound references across a vector operation that invalidates element references.

A named comparator for a different tie rule

Task. Sort integer pairs (`score`, `ID`) by decreasing score, then increasing `ID`. Here $0 \leq n \leq 200000$ and each field is in $[-10^9, 10^9]$; for a real student list, IDs would normally be unique.

The comparator answers “Should `a` appear before `b`?” for `std::sort`. It must be a strict, consistent ordering. In particular, comparing an item with itself must return false.^[4-22]

Complete program – use a helper to express the tie policy

```
#include <algorithm>
#include <iostream>
#include <utility>
#include <vector>

bool higherScore(const std::pair<int, int>& a,
                 const std::pair<int, int>& b) {
    if (a.first != b.first) return a.first > b.first;
    return a.second < b.second;
}

int main() {
    int n = 0;
    std::cin >> n;
    std::vector<std::pair<int, int>> students(n);
    for (auto& student : students) {
        std::cin >> student.first >> student.second;
    }
    std::sort(students.begin(), students.end(), higherScore);
    for (const auto& student : students) {
        std::cout << student.first << ' ' << student.second << '\n';
    }
}
```

Sample input

```
4
90 3
80 1
90 2
80 4
```

Sample output

```
90 2
90 3
80 1
80 4
```

Wrong comparator idea: return `a.first >= b.first`. It returns true when both values are the same and does not express the required ID tie rule. Do not replace strict comparison with a non-strict one to “include equality.”

Removing every match and keeping unique sorted values

`std::erase(values, target)` removes every matching value and returns the number removed. It is a free function declared for vectors in `<vector>`; it is available in C++23. This is different from the iterator-based member `values.erase(position)`.^[4-23]

`std::unique(first, last)` compacts consecutive equal values and returns a new *logical* end. It does not itself shrink the vector. Sorting first groups all equal values together; erasing the tail then leaves one of each value.^[4-24]

Complete program – remove a target, then deduplicate

```
#include <algorithm>
#include <iostream>
#include <vector>

int main() {
    int n = 0, target = 0;
    std::cin >> n >> target;
    std::vector<int> values(n);
    for (int& x : values) std::cin >> x;
    auto removed = std::erase(values, target);
    std::cout << removed << '\n';
    std::sort(values.begin(), values.end());
    auto newEnd = std::unique(values.begin(), values.end());
    values.erase(newEnd, values.end());
    for (int i = 0; i < static_cast<int>(values.size()); ++i) {
        if (i != 0) std::cout << ' ';
        std::cout << values[i];
    }
    std::cout << '\n';
}
```

Sample input

```
7 4
4 2 1 2 4 3 1
```

Sample output

```
2
1 2 3
```

Contract. $0 \leq n \leq 200000$, values and target in $[-10^9, 10^9]$. Print how many target values were removed, then the distinct remaining values in ascending order. The last line is blank for an empty result.

The logical end is not automatically the container's end

After `unique`, only the prefix before the returned iterator is the intended deduplicated result. The tail still consists of vector elements, but their values are not a meaningful answer to preserve. Use the returned iterator to erase that tail.^[4-24]

```
std::sort(values.begin(), values.end());
auto newEnd = std::unique(values.begin(), values.end());
values.erase(newEnd, values.end());
```

For a vector initially containing 1 1 2 2 3, the result prefix contains 1 2 3. The vector's size changes to three only after the final erase.

Without sorting, `unique` only removes consecutive duplicates: 1 2 1 has no consecutive equal pair and remains unchanged. Sorting first changes original order, so use this technique only when sorted distinct output is acceptable.

A safe erase-while-iterating pattern

```
for (auto it = values.begin(); it != values.end(); ) {
    if (*it == target) it = values.erase(it);
    else ++it;
}
```

The next iterator comes from `erase` when an element is removed. Incrementing the old iterator after erasure would use an invalidated iterator. This loop is useful for understanding iterator validity, but repeated vector shifts can make it quadratic. Prefer `std::erase` or a separate filtered result for this whole-vector task.

Pause and predict – 26

You call `unique` on sorted values 2 2 5 5 5. Is the vector's `size()` immediately two? What must happen next?

Check your prediction. No. The vector still has five elements until the tail is erased. Keep the returned logical-end iterator and call `erase(newEnd, end())`.

4.26 Optional extension: keep the largest k while streaming

Keep the weakest retained candidate easy to remove

Task. Read n values and keep the largest k , including duplicate occurrences. Print them in descending order. Here $1 \leq k \leq n \leq 200000$, with values in $[-10^9, 10^9]$. Do not store the entire input merely to select the result.

Use a **min-heap** for the retained candidates. Why a min-heap when the goal says “largest”? Because the smallest retained candidate is the one to discard when too many candidates are present.

For $k = 3$ and arrivals 5 1 9 9 2 7, the retained multisets are:

After arrival	Retained values shown sorted only for this trace
5	5
1	1 5
9	1 5 9
9	5 9 9
2	5 9 9
7	7 9 9

The trace is a sorted view for explanation, not a claim about the heap’s internal array order.

Why discarding the minimum is safe

Assume the heap already contains the largest at most k values from the processed prefix. Add the next value. If there are now $k+1$ candidates, their smallest cannot belong to the largest k of that candidate group, so remove it.

Previously rejected values cannot become more valuable when a new candidate arrives. Thus the invariant remains true. Equal values are separate occurrences: for example, the three largest values may include two equal nines.

Pause and predict – 27

For the largest three values seen so far, should the retained heap expose its largest or its smallest value for removal? Why?

Check your prediction. Its smallest. When a fourth candidate arrives, the least valuable retained candidate must be the one removed. A max-heap would discard the wrong end.

Implement the bounded candidate set

Complete program – a min-heap retaining at most k candidates

```
#include <algorithm>
#include <functional>
#include <iostream>
#include <queue>
#include <vector>

int main() {
    int n = 0, k = 0;
    std::cin >> n >> k;
    std::priority_queue<int, std::vector<int>, std::greater<int>>> best;
    for (int i = 0; i < n; ++i) {
        int x = 0;
        std::cin >> x;
        best.push(x);
        if (static_cast<int>(best.size()) > k) best.pop();
    }
    std::vector<int> answer;
    while (!best.empty()) {
        answer.push_back(best.top());
        best.pop();
    }
    std::reverse(answer.begin(), answer.end());
    for (int i = 0; i < k; ++i) {
        if (i != 0) std::cout << ' ';
        std::cout << answer[i];
    }
    std::cout << '\n';
}
```

Sample input

6 3
5 1 9 9 2 7

Sample output

9 9 7

At the end, popping the min-heap produces the retained values in ascending order. Reversing that output vector gives the requested descending order. Since $k \leq n$, the final heap contains exactly k values.

Cost. The heap temporarily reaches at most $k+1$ values. Each of n arrivals does a logarithmic-size heap update, and final extraction fits within the same overall $O(n \log(k+1))$ amortized time bound. Heap plus result vector use $O(k)$ logical storage. For $k = 1$, this is linear work.

Alternative. When all input is already stored and extra streaming memory is irrelevant, sorting and taking the last k values may be simpler. The benefit here is the smaller maintained set and the ability to process arrivals without retaining the full input.

4.27 Debugging and testing STL-based solutions

Check the operation's precondition first

Mistake	What to check instead
Accessing <code>v[0]</code> after only <code>reserve</code> .	Elements must exist within <code>size()</code> .
Filling <code>vector(n)</code> with <code>push_back</code> .	Decide whether to fill existing cells or append to an empty result.
Using <code>top</code> , <code>front</code> , or <code>pop</code> on empty.	Check state before the call, or prove nonemptiness from the invariant.
Writing <code>int x = s.pop();</code>	Copy the exposed value, then call removal separately.
Using <code>empty()</code> as a clearing command.	It is a query; choose actual removals or a fresh object.
Using <code>queue.top()</code> or <code>stack.front()</code> .	Check the tool's actual interface.
Remembering a reference across removal.	Copy the value before its element is destroyed.
Sorting pair priorities in the wrong direction.	Test equal first fields and different second fields.
Assuming a heap is a fully sorted array.	Only its top-priority access is exposed.
Using a by-value range variable for input.	Input must refer to an actual stored element.

Test a small state transition, not only a full sample

For command simulations, test requests before any insertion, insertion followed by one removal, repeated removals until empty, two equal values, and a new insertion after the structure becomes empty.

For vector algorithms, test zero length where allowed, one element, all equal values, a missing target, a target at each end, and negative values. For pair ordering, always test a tie. For bracket algorithms, test an early close, a wrong type, leftover openings, and the empty line.

Trace enough state to explain a failure

Write columns for the next command, the current sequence, the accessible end or top, and the produced output. For a heap, recording a sorted view of the stored values can help you predict the top, but label that view as a trace aid rather than the physical layout.

A failed test is useful evidence. Ask whether the error is in input parsing, state updates, the selection rule, a boundary condition, or the complexity assumption. Changing random container types until the sample works is not a reliable debugging method.

A local compiler command

Compile one supplied source file at a time. For example, from the source-package root:

```
g++ -std=c++23 -Wall -Wextra -Wshadow -O2 \
    examples/session_04/20_max_heap.cpp -o example
./example
```

The executable waits for the input described in that example. Select C++23 in an online judge's language menu when submitting. Availability and menu names vary by platform; the supplied build command records the intended language mode.^[4-25]

Debug builds help, but are not runtime benchmarks

The verification scripts also use undefined-behavior instrumentation and libstdc++ checks. Iterator-debug mode can catch invalid iterator use and range mistakes; it can also perform additional linear scans when checking heap preconditions. A debug executable therefore need not have the same observed speed as a normal contest build.^{[4-26][4-27]}

The large-boundary verification runs keep undefined-behavior checks and library assertions, but disable the extra iterator/heap debug mode. The exact flags and results are recorded with the sources. Passing tests is evidence about the tested paths, not a proof for every possible input.

Pause and predict – 28

A sample with distinct priorities passes. What additional case is essential before trusting a priority queue of pairs? Why is a test on an empty structure also useful?

Check your prediction. Use equal first fields with different second fields to expose the tie direction. Empty requests expose invalid accesses that normal nonempty samples cannot reveal.

4.28 Practice problems: choose, implement, and explain

Try each problem before using its hint or solution. For each answer, write the chosen tool, the statement that stays true after every step, and the total time and storage cost. There is often more than one valid implementation; obey any tool explicitly requested for practice.

Common bounds and output conventions

Unless a problem overrides them, $0 \leq n, q \leq 200000$ and each integer value lies in $[-10^9, 10^9]$. Inputs follow the stated format and valid command types. Print sequence values separated by single spaces, then a newline. A stated blank line means no values on that line, not the text “blank line.” Do not print labels or prompts. Problems 21 and 22 are optional challenges.

1. Reverse a vector

Read n , then n integers. Store them in a vector and print them in reverse input order on one line. A zero-length sequence produces a blank line. You may use a backward loop or the reversal algorithm.

Sample input

4
6 2 8 1

Sample output

1 8 2 6

Hint: page 81. Explained solution: page 86.

2. Keep only strictly positive values

Read n , then n integers. Keep only values strictly greater than zero, preserving input order and duplicates. Print the number kept, then their values on the next line. That second line is blank when none are kept.

Sample input

6
-2 5 0 7 -1 5

Sample output

3
5 7 5

Hint: page 81. Explained solution: page 87.

3. A growable list with queries

Read q , then q commands on an initially empty integer vector. 1 x : append x . 2: print and remove the last value, or print `EMPTY`. 3 i : print the value at zero-based index i , or `INVALID` if it does not exist. 4: print the size. Requested indices may be between -200000 and 200000 . Only commands 2, 3, and 4 produce output.

Sample input	Sample output
8	7
1 7	4
1 4	1
3 0	INVALID
2	7
4	EMPTY
3 1	
2	
2	

Hint: page 81. Explained solution: page 88.

4. Remove the first match

Read n , `target`, then n integers. Remove only the first occurrence of `target`; if absent, change nothing. Print the new size, then the resulting sequence on the next line. Preserve the relative order of all surviving values.

Sample input	Sample output
5 3	4
3 8 3 2 3	8 3 2 3

Hint: page 81. Explained solution: page 89.

5. Sort without losing the original order

Read n , then n integers. Print the values in ascending order on the first line. Print their original input order on the second line. Both lines are blank when $n=0$. Use a separate vector to preserve the original.

Sample input

4

4 1 4 2

Sample output

1 2 4 4

4 1 4 2

Hint: page 81. Explained solution: page 90.

6. Column totals in a matrix

Read positive `rows` and `cols`, each at most 300, followed by the matrix in row-major order. Store it as a nested vector and print each column's sum, left to right, on one line. Entries lie in $[-10^9, 10^9]$.

Sample input

2 3

1 2 3

4 5 6

Sample output5 7 9

Hint: page 81. Explained solution: page 91.

7. The first highest score

Read $1 \leq n \leq 200000$, then n scores in $[0, 100]$. Print the greatest score and the one-based position of its first occurrence. Write a helper that returns these two results as a pair.

Sample input	Sample output
5 70 95 95 60 90	95 2

Hint: page 81. Explained solution: page 92.

8. Order coordinate pairs

Read n , followed by n integer pairs x y . Print the points ordered by smaller x first and then smaller y on ties. Print one pair per line. Duplicate pairs must remain; $n=0$ produces no output lines.

Sample input	Sample output
4 2 3 -1 7 2 1 -1 2	-1 2 -1 7 2 1 2 3

Hint: page 81. Explained solution: page 93.

9. Keep the last k arrivals

Read n , k , then n integers, with $0 \leq k \leq 200000$. Keep only the latest k arrivals and print them from oldest retained to newest. When $k=0$, print a blank line; when $k>n$, retain all values. Use a deque without retaining the whole input elsewhere.

Sample input

6 2
1 8 2 9 3 7

Sample output

3 7

Hint: page 82. Explained solution: page 94.

10. Commands at both ends

Read q commands for an empty deque. L x : insert x at the front. R x : insert it at the back. F: print and remove the front. B: print and remove the back. A removal on an empty deque prints EMPTY. Insertions produce no output.

Sample input

7
L 3
R 8
L 1
B
F
F
B

Sample output

8
1
3
EMPTY

Hint: page 82. Explained solution: page 95.

11. Two players take end cards

Read n , then n card values in $[0, 10^9]$, ordered left to right. Two players alternate, starting with player one. On each turn the player *must* take the larger end value; on a tie, take the left end. Print player one's total and player two's total. An empty sequence gives 0 0. This is a prescribed simulation, not a claim that this strategy is optimal.

Sample input

4
4 1 2 10

Sample output

12 5

Hint: page 82. Explained solution: page 96.

12. Serve customers in arrival order

Read q commands. 1 id: join the back of a waiting line. 2: print and remove the next customer. 3: print the next customer without removing them. Print EMPTY for requests on an empty line. A request always refers to the oldest waiting item, not the smallest ID.

Sample input

6
1 7
1 4
3
2
2
2

Sample output

7
7
4
EMPTY

Hint: page 82. Explained solution: page 97.

13. Discard, then move to the back

Read $1 \leq n \leq 200000$. Start with cards $1, 2, \dots, n$ from front to back. While at least two cards remain, print and discard the front card, then move the next front card to the back. Print discarded cards on the first line, then the final remaining card on the second. For $n=1$, the first line is blank.

Sample input	Sample output
5	1 3 5 4 2

Hint: page 82. Explained solution: page 98.

14. Always take the highest bid

Read q commands. 1 x : add a bid. 2: print and remove the greatest stored bid. 3: print the greatest bid without removing it. Print **EMPTY** for requests when no bids remain. Each equal bid is a separate occurrence. Bid values may be in $[-10^9, 10^9]$ for this mechanical exercise.

Sample input	Sample output
6	50
1 20	40
1 50	20
2	
1 40	
2	
3	

Hint: page 82. Explained solution: page 99.

15. Print the k largest occurrences

Read n , k , then n integers, where $0 \leq k \leq n \leq 200000$. Print the k greatest occurrences in descending order. Duplicates count separately. A zero k gives a blank line. For this exercise, use a max-priority queue containing the input values.

Sample input

5 3
4 9 9 1 7

Sample output

9 9 7

Hint: page 82. Explained solution: page 100.

16. Earliest event, smaller ID on ties

Read n , then n pairs `time ID`. Times are in $[0, 10^9]$; IDs are unique positive integers at most 200000. Process earlier times first, then smaller IDs on tied times. Print only the processed IDs, one per line. Use a min-priority queue of pairs. A zero-event input prints no lines.

Sample input

4
5 3
2 8
5 1
2 4

Sample output

4
8
1
3

Hint: page 82. Explained solution: page 101.

17. A newest-first pile

Read q commands. 1 x : put x on top of a pile. 2: print and remove the top. 3: print the top without removing it. Print **EMPTY** for a request on an empty pile. The top is the newest remaining item, not the greatest number.

Sample input	Sample output
6	4
1 7	4
1 4	7
3	EMPTY
2	
2	
2	

Hint: page 83. Explained solution: page 102.

18. Check bracket nesting

Read one full line containing only bracket characters $()$, $[]$, and $\{\}$, of length zero through 200000. Print **YES** exactly when the sequence is balanced, otherwise **NO**. The empty line is balanced. A match must have the correct type and nesting order.

Sample input	Sample output
$([])\{\}\}$	YES

Hint: page 83. Explained solution: page 103.

19. Reduce adjacent equal pairs

Read a lowercase ASCII line of length zero through 200000. Scan left to right, repeatedly cancelling equal adjacent pairs in the reduced prefix. Print the remaining characters in their original left-to-right order, or `EMPTY` if the result has no characters.

Sample input

azxxzy

Sample output

ay

Hint: page 83. Explained solution: page 104.

20. Undo the most recent addition

Read q commands with an initial total of zero. 1 x : add x and remember that action. 2: cancel the most recent uncanceled addition; if there is none, do nothing. 3: print the current total. Only command 3 prints. Amounts are in $[-10^9, 10^9]$.

Sample input

8
1 10
1 -3
3
2
3
2
2
3

Sample output

7
10
0

Hint: page 83. Explained solution: page 105.

21. Challenge: reverse each group

Read n , k , then n integers, with $1 \leq k \leq 200000$. Split input into consecutive groups of size k , with a shorter final group if needed. Reverse each group but preserve group order. Print one output line. Use a queue for waiting values and a temporary stack for one group. This is an optional combination challenge.

Sample input

7 3
1 2 3 4 5 6 7

Sample output

3 2 1 6 5 4 7

Hint: page 83. Explained solution: page 106.

22. Challenge: the k th largest with limited storage

Read n , k , then n integers, where $1 \leq k \leq n \leq 200000$. Print the k -th largest occurrence; duplicates count separately. Process values as they arrive using $O(k)$ logical container storage rather than saving all n values. This optional challenge uses Section 4.26.

Sample input

6 3
5 1 9 9 2 7

Sample output

7

Hint: page 83. Explained solution: page 107.

4.29 Trace, tool-choice, and complexity drills

These drills are not complete programs to submit. Trace the fragments on paper and state the assumptions behind each cost. Unless stated otherwise, all sizes and indices fit the types used, element operations have constant cost, and a container operation must satisfy its preconditions.

Drill 1 – Construction and reserved room

Consider these independent declarations inside a function:

```
std::vector<int> a(3, 7);  
std::vector<int> b{3, 7};  
std::vector<int> c;  
c.reserve(8);
```

Give each vector's size and contents. Is `c[0] = 5`; valid? Give one valid operation that appends a five without creating eight elements.

Drill 2 – Copies and range references

Trace this fragment. What are the final contents of `a` and `b`?

```
std::vector<int> a{1, 2, 3};  
auto b = a;  
for (int x : a) { x += 10; }  
for (int& x : b) { x *= 2; }
```

Explain both why `a` did not change and why changing `b` did not change `a`.

Drill 3 – Half-open ranges and shifting indices

Trace the sequence below. List the vector after each operation.

```
std::vector<int> a{10, 20, 30, 40, 50};  
std::reverse(a.begin() + 1, a.begin() + 4);  
a.insert(a.begin() + 2, 99);  
a.erase(a.begin() + 3);
```

Is `a.erase(a.end())` a valid next operation? Explain the difference between that call and erasing an empty range at the end.

Drill 4 – Pair order and tie direction

The pairs are (2, 8), (1, 50), (2, 3), (1, 4). Give their order under ascending `sort`, complete extraction from a default max-priority queue, and complete extraction from a min-priority queue.

Now suppose pairs store (`score`, `ID`) and the required rule is *higher score, smaller ID*. Does either default ascending or default descending pair order implement that mixed rule? Explain.

Drill 5 – Two ends of a deque

Start with an empty deque and apply:

```
push_back(4)
push_front(7)
push_back(2)
pop_front()
push_front(9)
pop_back()
```

List the remaining sequence from front to back, then give its front, back, and size. Did any operation sort the values?

Drill 6 – Queue order and a shrinking bound

A queue receives 8, 3, 5, 1, in that order. What remains after this fragment?

```
for (int i = 0; i < static_cast<int>(q.size()); ++i) {
    q.pop();
}
```

Trace the loop counter and current size before each test. Rewrite the loop to remove every item safely.

Drill 7 – Max versus min priority

Push 4, 9, 2, and 9 into both a default integer priority queue and an integer min-heap. In each, record `top()`, pop once, and record `top()` again. Then insert 1 into each and record the new tops.

Explain why one max-heap pop does not remove both nines. Is `top()` an operation that removes the returned item?

Drill 8 – Bracket order and stack output

Decide whether each line is balanced: `([])`, `([]]`, `((()`, the empty line, and `)()`. Give the first reason for each rejection.

For adjacent-pair cancellation, trace `azxxzy`. What is the left-to-right reduced result? What order would a raw top-and-pop loop print without reversing the collected characters?

Drill 9 – Choose a first tool

Choose a natural tool for each job and justify the required access order: arbitrary score updates by index; keeping (`time`, `ID`) together; arrivals served first-in-first-out; adding/removing at either end; repeated minimum selection after insertions; undoing the newest uncanceled action.

For a fixed input that only needs fully sorted output once, explain why a vector plus `sort` may be preferable to building and draining a heap.

Drill 10 – Total work, not operation names

For each independent scenario, give a tight conventional growth classification under the stated model:

1. Append n integers to an initially empty vector, without manual reserve calls.
2. Repeatedly erase the first element of a vector until its n elements are gone.
3. Repeatedly pop a default queue containing n items until empty.
4. Build a max-heap by n pushes, then pop all values. Use a worst-case bound over integer inputs, not the special case of all equal keys.
5. Run the one-unit-turn job simulation with positive workloads $w_1, \dots, w_n$.

Assume n is positive for asymptotic shorthand. Distinguish amortized guarantees from per-call worst cases.

Drill 11 – A hidden copy inside a cheap query

A helper is called `q` times with an existing vector of `n` integers:

```
std::size_t countItems(std::vector<int> values) {  
    return values.size();  
}
```

Include parameter copying in the total time and peak additional storage. Then change the parameter to `const std::vector<int>&`. What changes? Would the const-reference version be constant-time per call if its body instead scanned all values to compute their sum?

Drill 12 – Check four tempting shortcuts

Evaluate these claims separately. For each false claim, explain the missing rule.

1. Reserving enough vector capacity makes it safe to append to that vector during a range-based loop over it.
2. `auto& next = waiting.front(); waiting.pop();` keeps a safe saved copy for later use.
3. A sort comparator using `a.first >= b.first` is valid because it handles ties.
4. Keeping only the largest `k` values in a min-heap can use $O(k)$ storage and $O(n \log(k + 1))$ amortized time for $1 \leq k \leq n$.

4.30 Hints before solutions

A hint should help you take the next step, not replace the whole attempt. Return to the problem after reading just its hint. The separate solutions begin in Section 4.31.

1. Reverse a vector

Reverse the entire half-open range, or print indices from $n-1$ down to zero. A size-zero vector still has a valid empty range.

2. Keep only strictly positive values

Start with an empty vector, not `vector<int>(n)`. Append only when $x > 0$; zero is not positive.

3. A growable list with queries

Separate append, remove-last, indexed access, and size queries. Guard removal with `empty()` and validate both the lower and upper index bound.

4. Remove the first match

Use `find` to get the first matching iterator. Erase it only when it is not `end()`; `std::erase` would remove all matches, which is a different task.

5. Sort without losing the original order

A vector assigned to another vector by value becomes an independent copy. Sort the copy, not the original. A printing helper can take a const reference.

6. Column totals in a matrix

Keep the column index fixed while scanning all rows for that column. Reset a `long long` total once per column, not once per cell.

7. The first highest score

Initialize from the first actual score. Update only for a strictly greater score so a later tie does not replace the earliest position.

8. Order coordinate pairs

The required order is the default ascending order of integer pairs. Keep each point together as one vector element.

9. Keep the last k arrivals

Append the new value at the back. If the size exceeds k , remove the front value. The deque temporarily contains at most $k+1$ items.

10. Commands at both ends

Use the matching push/pop member for each end. An empty check must happen before either front or back access.

11. Two players take end cards

Compare `front()` and `back()`, copy the chosen value, remove it, add to the current player's wide total, and flip whose turn it is.

12. Serve customers in arrival order

Use a FIFO queue. The parser resembles the pile and bid problems, but the accessible item must be `front()`, not `top()`.

13. Discard, then move to the back

Use a queue. After the discard, at least one card remains because the iteration began with at least two. Copy that next card before removing and reinserting it.

14. Always take the highest bid

Use a default integer priority queue. Read `top()` before popping and do not replace priority order with arrival order.

15. Print the k largest occurrences

Build the max-heap, then perform exactly k top-and-pop operations. The constraint $k \leq n$ guarantees each requested extraction exists.

16. Earliest event, smaller ID on ties

Store `(time, ID)`, not the reversed fields. A min-heap of that pair type applies the correct rule to both fields.

17. A newest-first pile

Use a stack. The command grammar is the same as the queue exercise, but LIFO changes which stored value is exposed.

18. Check bracket nesting

Store unmatched openings. On a close, reject an empty stack or a mismatched top; otherwise pop. Accept only when no unmatched opening remains at the end.

19. Reduce adjacent equal pairs

The reduced prefix acts like a stack. A matching top cancels; a different top receives a push. When using `std::stack`, reverse the popped output before printing.

20. Undo the most recent addition

Keep uncanceled amounts on a stack and maintain a `long long` total. Undo subtracts the top amount before popping, even when the amount is negative.

21. Challenge: reverse each group

Take at most `k` front items into a fresh stack, then drain that stack completely. Never create a group larger than the values still waiting.

22. Challenge: the kth largest with limited storage

Maintain a min-heap of the largest at most `k` values. If an append makes it too large, pop the minimum. At the end, its minimum is the `k`th largest overall.

Hints for the twelve drills

Drill	One useful starting point
1	Separate element count from listed values and reserved storage.
2	Mark each declaration as a value copy or a reference before tracing assignments.
3	The right endpoint is excluded; then recompute indices after the insertion.
4	Compare first fields before second fields. Reversing pair order reverses both tie directions.
5	Label front on the left and back on the right after every operation.
6	The loop counter increases while <code>size()</code> decreases.
7	Equal pushes create separate occurrences. A top query does not change size.
8	A close must match the latest unmatched open; popping a stack reverses its order.
9	Ask which item is needed next, not merely which tool can store the data.
10	Sum the shifts, count queue removals, and count the total units of job work.
11	The vector parameter exists before the helper's body asks for its size.
12	Check the old end iterator, removed-element lifetime, strict comparison, and the weakest retained candidate.

Before opening an answer

For a trace, write the states yourself. For a cost, name the input sizes and count the repeated work. For a tool choice, state the access rule. Those three habits will help with every later session, not only STL.

4.31 Explained practice solutions

Each solution starts on its own page and is a complete C++23 program. Input contracts and sample output are in the problem section; links there lead to the relevant hint and solution.

Read the idea and correctness explanation before studying the code. A different program is equally valid if it satisfies the stated contract and required practice tool, handles the boundaries, and meets the time and memory limits.

Use the answers in three passes

First identify the container's job. Then trace the sample, naming the accessible end or priority at each step. Finally, close the answer and write your own implementation.

How costs are reported

Time includes input, output, and container work. Storage counts the explicitly stored input and auxiliary containers, unless the text specifically separates them. For zero-length inputs, +1 records fixed setup work. Priority-queue insertion sequences use the amortized bound discussed in Section 4.19.

Equivalent code can use different memory

A vector-plus-index solution and a queue solution may both run in linear time but express different access rules. When this practice explicitly requests a tool, use that tool to learn its interface before experimenting with alternatives.

Solution 1 – Reverse a vector

Problem: page 64. **Idea.** Read into a vector that already has n elements. Reverse its whole range, then print the resulting sequence.

Complete solution – C++23

```
#include <algorithm>
#include <iostream>
#include <vector>

int main() {
    int n = 0;
    std::cin >> n;
    std::vector<int> a(n);
    for (int& x : a) std::cin >> x;
    std::reverse(a.begin(), a.end());
    for (int i = 0; i < n; ++i) {
        if (i != 0) std::cout << ' ';
        std::cout << a[i];
    }
    std::cout << '\n';
}
```

Why it works. Reversal exchanges the first and last elements, the second and second-last, and so on. Every original index i ends at $n-1-i$. Printing the vector left to right therefore produces exactly the requested reverse order.

Cost and boundaries. $O(n+1)$ time and $O(n+1)$ input storage, with constant extra work space for reversal. For $n=0$, no element is accessed and one blank line is printed.

Solution 2 – Keep only strictly positive values

Problem: page 64. **Idea.** Start with an empty result. Append each input exactly when it is positive, without changing the order of arrivals.

Complete solution – C++23

```
#include <iostream>
#include <vector>

int main() {
    int n = 0;
    std::cin >> n;
    std::vector<int> positive;
    for (int i = 0; i < n; ++i) {
        int x = 0;
        std::cin >> x;
        if (x > 0) positive.push_back(x);
    }
    std::cout << positive.size() << '\n';
    for (int i = 0; i < static_cast<int>(positive.size()); ++i) {
        if (i != 0) std::cout << ' ';
        std::cout << positive[i];
    }
    std::cout << '\n';
}
```

Why it works. After each input, the vector contains precisely the positive values in the processed prefix. Rejecting a nonpositive value leaves that fact unchanged; appending a positive value extends the correct result by one element.

Cost and boundaries. $O(n+1)$ total amortized time and $O(n+1)$ worst-case storage. All-negative and all-zero inputs leave an empty vector; the count is zero and the second output line is blank.

Solution 3 – A growable list with queries

Problem: page 65. **Idea.** Translate each command into one vector operation. An indexed request is valid only when its index is nonnegative and smaller than the current size.

Complete solution – C++23

```
#include <iostream>
#include <vector>

int main() {
    int q = 0;
    std::cin >> q;
    std::vector<int> values;
    while (q-- > 0) {
        int type = 0, x = 0;
        std::cin >> type;
        if (type == 1) {
            std::cin >> x;
            values.push_back(x);
        } else if (type == 2) {
            if (values.empty()) std::cout << "EMPTY\n";
            else {
                std::cout << values.back() << '\n';
                values.pop_back();
            }
        } else if (type == 3) {
            std::cin >> x;
            if (x < 0 || x >= static_cast<int>(values.size())) {
                std::cout << "INVALID\n";
            } else std::cout << values[x] << '\n';
        } else {
            std::cout << values.size() << '\n';
        }
    }
}
```

Why it works. The vector is exactly the requested list after every command. Appending adds the new last value; a valid removal deletes only that value; queries leave the list unchanged. Guards prevent reading nonexistent elements.

Cost. $O(q + 1)$ total amortized time, including growth, and $O(q + 1)$ worst-case storage. Testing only $i < \text{size}$ would miss negative indices.

Solution 4 – Remove the first match

Problem: page 65. **Idea.** Find the first matching iterator, erase that one element if found, and print the remaining sequence.

Complete solution – C++23

```
#include <algorithm>
#include <iostream>
#include <vector>

int main() {
    int n = 0, target = 0;
    std::cin >> n >> target;
    std::vector<int> a(n);
    for (int& x : a) std::cin >> x;
    auto it = std::find(a.begin(), a.end(), target);
    if (it != a.end()) a.erase(it);
    std::cout << a.size() << '\n';
    for (int i = 0; i < static_cast<int>(a.size()); ++i) {
        if (i != 0) std::cout << ' ';
        std::cout << a[i];
    }
    std::cout << '\n';
}
```

Why it works. `find` returns the earliest matching position. One member-`erase` removes only that element and closes the gap without reordering the surviving values. If the search returns `end()`, skipping erasure preserves the original vector.

Cost and boundaries. Search, one shift, and output each take at most linear work: $O(n + 1)$ overall. Storage is $O(n + 1)$. Check an empty input, a missing target, and repeated targets.

Solution 5 – Sort without losing the original order

Problem: page 66. **Idea.** Copy the input vector, sort the copy, then print the sorted vector and the unchanged original.

Complete solution – C++23

```
#include <algorithm>
#include <iostream>
#include <vector>

void print(const std::vector<int>& a) {
    for (int i = 0; i < static_cast<int>(a.size()); ++i) {
        if (i != 0) std::cout << ' ';
        std::cout << a[i];
    }
    std::cout << '\n';
}

int main() {
    int n = 0;
    std::cin >> n;
    std::vector<int> original(n);
    for (int& x : original) std::cin >> x;
    auto sorted = original;
    std::sort(sorted.begin(), sorted.end());
    print(sorted);
    print(original);
}
```

Why it works. The copy initially contains the same values. Sorting changes only that separate object, so it supplies the ordered output while the original supplies the input-order output. The read-only helper cannot reorder either vector.

Cost. $O(1 + n \log(n + 1))$ time. The two explicit vectors use $O(n + 1)$ stored elements; the extra copy alone costs $O(n + 1)$ space. Sorting also has implementation workspace; see the technical notes.

Solution 6 – Column totals in a matrix

Problem: page 66. **Idea.** Read each stored row, then use a column loop outside a row loop when computing totals.

Complete solution – C++23

```
#include <iostream>
#include <vector>

int main() {
    int rows = 0, cols = 0;
    std::cin >> rows >> cols;
    std::vector<std::vector<int>> a(rows, std::vector<int>(cols));
    for (auto& row : a) {
        for (int& x : row) std::cin >> x;
    }
    for (int j = 0; j < cols; ++j) {
        long long total = 0;
        for (int i = 0; i < rows; ++i) total += a[i][j];
        if (j != 0) std::cout << ' ';
        std::cout << total;
    }
    std::cout << '\n';
}
```

Why it works. For a fixed column j , the inner loop visits $a[0][j]$ through $a[\text{rows}-1][j]$ exactly once. Its total is therefore the requested column sum. The outer loop prints columns in their natural left-to-right order.

Cost and boundaries. $O(\text{rows} \cdot \text{cols})$ time and stored cells, with constant additional working variables. Wide totals are required: three hundred entries of 10^9 do not fit a 32-bit `int`.

Solution 7 – The first highest score

Problem: page 67. **Idea.** Keep a pair (best score, first position). Replace it only when a strictly better score appears.

Complete solution – C++23

```
#include <iostream>
#include <utility>
#include <vector>

std::pair<int, int> best(const std::vector<int>& scores) {
    std::pair<int, int> answer{scores[0], 1};
    for (int i = 1; i < static_cast<int>(scores.size()); ++i) {
        if (scores[i] > answer.first) answer = {scores[i], i + 1};
    }
    return answer;
}

int main() {
    int n = 0;
    std::cin >> n;
    std::vector<int> scores(n);
    for (int& x : scores) std::cin >> x;
    auto answer = best(scores);
    std::cout << answer.first << ' ' << answer.second << '\n';
}
```

Why it works. The stored score is always the greatest in the scanned prefix. On an equal score, not updating preserves the earlier position. On a greater score, the current position is the first occurrence of that new greatest value in the prefix.

Cost and boundaries. $O(n)$ time and $O(n)$ input storage; the helper needs $O(1)$ additional space. Nonempty input makes index zero valid. Test all-equal scores to verify the one-based tie rule.

Solution 8 – Order coordinate pairs

Problem: page 67. **Idea.** Store every point as `pair<int, int>` and sort the vector using default pair order.

Complete solution – C++23

```
#include <algorithm>
#include <iostream>
#include <utility>
#include <vector>

int main() {
    int n = 0;
    std::cin >> n;
    std::vector<std::pair<int, int>> points(n);
    for (auto& point : points) {
        std::cin >> point.first >> point.second;
    }
    std::sort(points.begin(), points.end());
    for (const auto& point : points) {
        std::cout << point.first << ' ' << point.second << '\n';
    }
}
```

Why it works. Pair comparison first compares `x`. Only equal `x`-coordinates lead to a `y`-coordinate comparison. This is exactly the required order, and sorting whole pairs never disconnects a point's coordinates.

Cost and boundaries. $O(1 + n \log(n + 1))$ time for fixed-width pairs and $O(n + 1)$ explicit storage, plus sorting workspace. Negative coordinates and duplicate full pairs are valid and are not removed.

Solution 9 – Keep the last k arrivals

Problem: page 68. **Idea.** Keep recent values in arrival order, remove the oldest only when a new arrival exceeds the allowed count.

Complete solution – C++23

```
#include <deque>
#include <iostream>

int main() {
    int n = 0, k = 0;
    std::cin >> n >> k;
    std::deque<int> recent;
    for (int i = 0; i < n; ++i) {
        int x = 0;
        std::cin >> x;
        recent.push_back(x);
        if (static_cast<int>(recent.size()) > k) recent.pop_front();
    }
    for (int i = 0; i < static_cast<int>(recent.size()); ++i) {
        if (i != 0) std::cout << ' ';
        std::cout << recent[i];
    }
    std::cout << '\n';
}
```

Why it works. Each processed prefix leaves exactly its last at most k values in the deque. Appending adds the newest, and any necessary front removal discards exactly the oldest excess value. Final indexed output preserves retained arrival order.

Cost. $O(n + 1)$ time and $O(\min(n, k) + 1)$ storage, including the temporary extra arrival. The ordinary update safely handles $k=0$ without accessing an empty deque.

Solution 10 – Commands at both ends

Problem: page 68. **Idea.** Keep the sequence in a deque. Each letter maps directly to one specified end operation.

Complete solution – C++23

```
#include <deque>
#include <iostream>

int main() {
    int q = 0;
    std::cin >> q;
    std::deque<int> line;
    while (q--> {
        char command = ' ';
        std::cin >> command;
        if (command == 'L' || command == 'R') {
            int x = 0;
            std::cin >> x;
            if (command == 'L') line.push_front(x);
            else line.push_back(x);
        } else if (line.empty()) {
            std::cout << "EMPTY\n";
        } else if (command == 'F') {
            std::cout << line.front() << '\n';
            line.pop_front();
        } else {
            std::cout << line.back() << '\n';
            line.pop_back();
        }
    }
}
```

Why it works. L and R insert at exactly the requested end. For F and B, the value is printed before that same end element is removed. The empty branch prevents either end from being accessed without an element.

Cost. $O(q + 1)$ total time and $O(q + 1)$ worst-case storage. Test front and back removals from a one-item deque: either operation must remove that single item only.

Solution 11 – Two players take end cards

Problem: page 69. **Idea.** Store the remaining cards in a deque. Follow the specified end-choice rule and alternate the recipient of each value.

Complete solution – C++23

```
#include <deque>
#include <iostream>

int main() {
    int n = 0;
    std::cin >> n;
    std::deque<int> cards;
    for (int i = 0; i < n; ++i) {
        int x = 0;
        std::cin >> x;
        cards.push_back(x);
    }
    long long first = 0, second = 0;
    bool firstTurn = true;
    while (!cards.empty()) {
        int chosen = 0;
        if (cards.front() >= cards.back()) {
            chosen = cards.front();
            cards.pop_front();
        } else {
            chosen = cards.back();
            cards.pop_back();
        }
        if (firstTurn) first += chosen;
        else second += chosen;
        firstTurn = !firstTurn;
    }
    std::cout << first << ' ' << second << '\n';
}
```

Why it works. The deque always contains precisely the unclaimed contiguous row of cards. Each turn removes the required end, and the turn flag awards its value to the correct player. One card disappears per iteration, so all cards are eventually awarded.

Cost and boundaries. $O(n + 1)$ time and storage. Totals use `long long`. A one-card deque has equal front and back; the left branch removes it once. No optimal-play theorem is needed.

Solution 12 – Serve customers in arrival order

Problem: page 69. **Idea.** A FIFO queue represents the waiting line. New arrivals join at the back; requests use its front.

Complete solution – C++23

```
#include <iostream>
#include <queue>

int main() {
    int q = 0;
    std::cin >> q;
    std::queue<int> waiting;
    while (q-- > 0) {
        int type = 0;
        std::cin >> type;
        if (type == 1) {
            int id = 0;
            std::cin >> id;
            waiting.push(id);
        } else if (waiting.empty()) {
            std::cout << "EMPTY\n";
        } else {
            std::cout << waiting.front() << '\n';
            if (type == 2) waiting.pop();
        }
    }
}
```

Why it works. At all times, the queue stores the unserved arrivals in arrival order. The front is therefore the next required customer. A peek leaves the state unchanged, while a service command removes exactly that front customer.

Cost. $O(q + 1)$ time and worst-case storage for the default queue. Two identical numeric IDs are still two separate queued items unless the task forbids such input.

Solution 13 – Discard, then move to the back

Problem: page 70. **Idea.** Repeat the two specified queue actions while the size exceeds one; then print the sole survivor.

Complete solution – C++23

```
#include <iostream>
#include <queue>

int main() {
    int n = 0;
    std::cin >> n;
    std::queue<int> cards;
    for (int i = 1; i <= n; ++i) cards.push(i);
    bool first = true;
    while (cards.size() > 1) {
        if (!first) std::cout << ' ';
        std::cout << cards.front();
        cards.pop();
        int next = cards.front();
        cards.pop();
        cards.push(next);
        first = false;
    }
    std::cout << '\n' << cards.front() << '\n';
}
```

Why it works. Each loop iteration discards exactly the current front and rotates exactly the next card. The queue remains in the specified order. The size decreases by one per iteration, so after $n-1$ iterations exactly one card remains.

Cost and boundaries. $O(n)$ time and storage: a constant number of queue operations per discarded card. For $n=1$, the loop is skipped and the initial card is the survivor.

Solution 14 – Always take the highest bid

Problem: page 70. **Idea.** Maintain bids in a max-priority queue so the next requested greatest value is always exposed.

Complete solution – C++23

```
#include <iostream>
#include <queue>

int main() {
    int q = 0;
    std::cin >> q;
    std::priority_queue<int> bids;
    while (q-- > 0) {
        int type = 0;
        std::cin >> type;
        if (type == 1) {
            int x = 0;
            std::cin >> x;
            bids.push(x);
        } else if (bids.empty()) {
            std::cout << "EMPTY\n";
        } else {
            std::cout << bids.top() << '\n';
            if (type == 2) bids.pop();
        }
    }
}
```

Why it works. After every insertion or removal, the adaptor restores the maximum-at-top rule. A request therefore prints the greatest currently stored bid. Pop removes one copy only; the remaining equal copies are still eligible.

Cost. $O(1 + q \log(q + 1))$ total amortized time and $O(q + 1)$ storage. A peek alone is constant-time. The queue does not promise FIFO behavior for equal-priority records with extra identity.

Solution 15 – Print the k largest occurrences

Problem: page 71. **Idea.** Insert all values into a max-heap and extract its top exactly k times.

Complete solution – C++23

```
#include <iostream>
#include <queue>

int main() {
    int n = 0, k = 0;
    std::cin >> n >> k;
    std::priority_queue<int> values;
    for (int i = 0; i < n; ++i) {
        int x = 0;
        std::cin >> x;
        values.push(x);
    }
    for (int i = 0; i < k; ++i) {
        if (i != 0) std::cout << ' ';
        std::cout << values.top();
        values.pop();
    }
    std::cout << '\n';
}
```

Why it works. The first extraction is the greatest input occurrence. Removing it leaves a heap of the remaining occurrences, so the next extraction is the next greatest. Repeating this argument yields the required k values in descending order.

Cost and boundaries. $O(1 + n \log(n + 1))$ total amortized time for this build-and-extract implementation, with $O(n + 1)$ storage. $k=0$ performs no heap access; $k=n$ prints every occurrence.

Solution 16 – Earliest event, smaller ID on ties

Problem: page 71. **Idea.** The priority key is (time, ID). Use ascending pair priority and output the selected event's ID.

Complete solution – C++23

```
#include <functional>
#include <iostream>
#include <queue>
#include <utility>
#include <vector>

int main() {
    int n = 0;
    std::cin >> n;
    using Event = std::pair<int, int>;
    std::priority_queue<Event, std::vector<Event>,
                      std::greater<Event>>> events;
    for (int i = 0; i < n; ++i) {
        int time = 0, id = 0;
        std::cin >> time >> id;
        events.push({time, id});
    }
    while (!events.empty()) {
        int id = events.top().second;
        events.pop();
        std::cout << id << '\n';
    }
}
```

Why it works. Lexicographic minimum first minimizes time. Among events with that time, it minimizes ID. Removing the selected event leaves exactly the unprocessed events, so the same rule applies to each next output.

Cost. $O(1 + n \log(n + 1))$ total amortized time and $O(n + 1)$ storage. Copy the needed ID before removing its pair. Sorting all events once is another valid approach when interleaving is unnecessary.

Solution 17 – A newest-first pile

Problem: page 72. **Idea.** Represent the pile with a stack. Push arrivals and use `top()` for either kind of request.

Complete solution – C++23

```
#include <iostream>
#include <stack>

int main() {
    int q = 0;
    std::cin >> q;
    std::stack<int> pile;
    while (q-- > 0) {
        int type = 0;
        std::cin >> type;
        if (type == 1) {
            int x = 0;
            std::cin >> x;
            pile.push(x);
        } else if (pile.empty()) {
            std::cout << "EMPTY\n";
        } else {
            std::cout << pile.top() << '\n';
            if (type == 2) pile.pop();
        }
    }
}
```

Why it works. Each push places the newest item above the older items. Each removal exposes the next newest remaining item. Thus the stack's accessible value always matches the prescribed pile rule.

Cost and boundaries. $O(q + 1)$ time and worst-case storage for the default stack. Read the value before popping. Test a new push after the pile has become empty again.

Solution 18 – Check bracket nesting

Problem: page 72. **Idea.** Store the unmatched openings in a stack. For a closing bracket, determine the required opening type and compare it with the top.

Complete solution – C++23

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

bool balanced(const std::string& text) {
    std::stack<char> open;
    for (char c : text) {
        if (c == '(' || c == '[' || c == '{') open.push(c);
        else {
            if (open.empty()) return false;
            char wanted = '(';
            if (c == ']') wanted = '[';
            if (c == '}') wanted = '{';
            if (open.top() != wanted) return false;
            open.pop();
        }
    }
    return open.empty();
}

int main() {
    std::string text;
    std::getline(std::cin, text);
    std::cout << (balanced(text) ? "YES\n" : "NO\n");
}
```

Why it works. The latest unmatched opening is the only one that may close next. Every accepted close removes exactly that opening. Rejecting a missing or wrong opening enforces prefix validity; final emptiness ensures that all openings were closed.

Cost. $O(n + 1)$ time and worst-case additional stack storage for a line of length n . The input contract excludes other characters, so the closing-type logic needs only the three specified cases.

Solution 19 – Reduce adjacent equal pairs

Problem: page 73. **Idea.** Maintain the already reduced prefix on a stack. Cancel a matching top or push a different next character.

Complete solution – C++23

```
#include <algorithm>
#include <iostream>
#include <stack>
#include <string>

int main() {
    std::string text;
    std::getline(std::cin, text);
    std::stack<char> kept;
    for (char c : text) {
        if (!kept.empty() && kept.top() == c) kept.pop();
        else kept.push(c);
    }
    std::string answer;
    while (!kept.empty()) {
        answer.push_back(kept.top());
        kept.pop();
    }
    std::reverse(answer.begin(), answer.end());
    std::cout << (answer.empty() ? "EMPTY" : answer) << '\n';
}
```

Why it works. Before each character, the retained prefix has no equal adjacent pair. The only possible new pair is between its last character and the incoming one. Cancelling that pair or appending a different character preserves the invariant. Popped output is reversed to restore left-to-right order.

Cost. $O(n + 1)$ total time and storage: each character is pushed and popped at most once. Test a chain such as `abba`, which becomes empty after one cancellation reveals another.

Solution 20 – Undo the most recent addition

Problem: page 73. **Idea.** A stack holds uncanceled additions, and a running total equals their sum.

Complete solution – C++23

```
#include <iostream>
#include <stack>

int main() {
    int q = 0;
    std::cin >> q;
    std::stack<long long> changes;
    long long total = 0;
    while (q-- > 0) {
        int type = 0;
        std::cin >> type;
        if (type == 1) {
            long long x = 0;
            std::cin >> x;
            changes.push(x);
            total += x;
        } else if (type == 2) {
            if (!changes.empty()) {
                total -= changes.top();
                changes.pop();
            }
        } else {
            std::cout << total << '\n';
        }
    }
}
```

Why it works. Adding an amount updates both the history and its sum. Undo removes the newest stored amount and subtracts that same amount, restoring the previous sum. An empty undo changes neither state, and a query only reads the sum.

Cost and boundaries. $O(q + 1)$ time and storage. The maximum absolute total is at most $2 \cdot 10^{14}$, so use `long long`. Cancelling a negative addition increases the total, as subtraction requires.

Solution 21 – Challenge: reverse each group

Problem: page 74. **Idea.** The queue supplies each next group in input order; a temporary stack emits that group in reverse order.

Complete solution – C++23

```
#include <iostream>
#include <queue>
#include <stack>

int main() {
    int n = 0, k = 0;
    std::cin >> n >> k;
    std::queue<int> waiting;
    for (int i = 0; i < n; ++i) {
        int x = 0;
        std::cin >> x;
        waiting.push(x);
    }
    bool first = true;
    while (!waiting.empty()) {
        std::stack<int> group;
        for (int i = 0; i < k && !waiting.empty(); ++i) {
            group.push(waiting.front());
            waiting.pop();
        }
        while (!group.empty()) {
            if (!first) std::cout << ' ';
            std::cout << group.top();
            group.pop();
            first = false;
        }
        std::cout << '\n';
    }
}
```

Why it works. The outer loop always begins at the next unprocessed group. Its first inner loop removes exactly that group from the queue. Its second inner loop reverses only those removed values. Groups are completed one at a time, so their overall order is unchanged.

Cost. $O(n + 1)$ time: every value enters and leaves each structure once. Total stored data is $O(n + 1)$; the group stack holds at most $\min(n, k)$ values. Positive k guarantees progress.

Solution 22 – Challenge: the kth largest with limited storage

Problem: page 74. **Idea.** Retain only the largest k candidates in a min-heap. The weakest retained candidate is the requested threshold.

Complete solution – C++23

```
#include <functional>
#include <iostream>
#include <queue>
#include <vector>

int main() {
    int n = 0, k = 0;
    std::cin >> n >> k;
    std::priority_queue<int, std::vector<int>, std::greater<int>>> best;
    for (int i = 0; i < n; ++i) {
        int x = 0;
        std::cin >> x;
        best.push(x);
        if (static_cast<int>(best.size()) > k) best.pop();
    }
    std::cout << best.top() << '\n';
}
```

Why it works. After each arrival, removing the minimum of $k+1$ candidates preserves exactly the largest k seen so far. At the end there are exactly k retained occurrences, so their smallest is the k -th largest occurrence of the whole input.

Cost and boundaries. $O(n \log(k + 1))$ total amortized time and $O(k)$ storage. $k=1$ returns the maximum; $k=n$ returns the minimum. Duplicate values are not deduplicated.

4.32 Explained trace and complexity drills

Compare these explanations with your written trace, not only with your final number. A correct guess does not reveal whether the intermediate reasoning was reliable.

Drill 1 – Construction and reserved room

`a` has size three and values 7 7 7. `b` has size two and values 3 7. `c` has size zero; reserving room for at least eight elements does not create any elements.

Thus `c[0] = 5;` is invalid. Use `c.push_back(5);` to create one appended value. The resulting size is one. Its capacity may exceed eight; the exact chosen capacity is not the answer being tested.

Drill 2 – Copies and range references

The final `a` is 1 2 3. The final `b` is 2 4 6. In the first loop, `x` is an integer copy, so adding ten to it does not update an element of `a`.

The declaration `auto b = a;` created a separate vector. In the second loop, `int& x` refers to an element of that separate vector, so doubling it changes `b` but not `a`. Two different copy/reference boundaries are involved: vector construction and element traversal.

Drill 3 – Half-open ranges and shifting indices

After reversal, the vector is 10 40 30 20 50: only indices one through three were reversed. Inserting ninety-nine before the current index two gives 10 40 99 30 20 50. Erasing the current index three then removes thirty, leaving 10 40 99 20 50.

`erase(end())` is invalid because the single-element overload needs an existing element. The range call `erase(end(), end())` is valid and removes no elements. A boundary can be a valid iterator without being dereferenceable.

Drill 4 – Pair order and tie direction

Ascending sort and min-heap extraction both produce (1,4), (1,50), (2,3), (2,8). Max-heap extraction produces (2,8), (2,3), (1,50), (1,4).

Neither natural ascending nor natural descending pair order implements “higher score, smaller ID.” Ascending gives the wrong score direction; descending gives the wrong ID tie direction. Use a named sort comparator for that rule, or a carefully justified transformed priority key for a heap.

Drill 5 – Two ends of a deque

The states are 4, then 7 4, then 7 4 2, then 4 2, then 9 4 2, and finally 9 4. Front is nine, back is four, and size is two.

No sorting occurred. Front and back describe sequence positions, not numeric extrema. On each push or pop, identify which specific end changes and leave the other elements in their current relative order.

Drill 6 – Queue order and a shrinking bound

i	Size before test	Test	Action
0	4	0 < 4: true	Remove 8.
1	3	1 < 3: true	Remove 3.
2	2	2 < 2: false	Stop.

The remaining queue is 5 1, front to back. To drain it, use:

```
while (!q.empty()) {
    q.pop();
}
```

The condition now directly asks whether there is more work to remove. It is checked before each potentially invalid empty pop.

Drill 7 – Max versus min priority

The max-heap tops are nine before the pop and nine after it, because two separate nines were inserted. After adding one, its top is still nine.

The min-heap tops are two before the pop and four after it. After adding one, its top becomes one. In both structures, `pop()` removes one occurrence, while `top()` only exposes the current selected value without changing size.

Drill 8 – Bracket order and stack output

`([])` is balanced. `([])` fails at `)` because the latest unmatched opening is `[`. `((` fails the final check because an opening remains. The empty line is balanced. `)``(` fails immediately because a closing bracket has no opening to match.

For `azxxzy`, the reduced prefixes are `a`, `az`, `azx`, `az`, `a`, and `ay`. The required result is `ay`. Raw top-and-pop output would be `ya`, so the collected result must be reversed before printing left to right.

Drill 9 – Choose a first tool

Indexed score updates naturally use a vector. Two related values use a pair, potentially inside a vector or heap. FIFO service uses a queue; end operations use a deque; repeated minimum selection uses a min-priority queue; newest-action undo uses a stack.

A fixed input requiring one complete sorted output is often easiest with a vector and one sort call. A priority queue is particularly helpful when insertions and extreme-value requests are interleaved. It is not an automatic speed improvement over sorting a fixed sequence.

Drill 10 – Total work, not operation names

Part	Growth	Reason
1	$\Theta(n)$ amortized total	Vector growth spreads relocation over the append sequence.
2	$\Theta(n^2)$ shifts	The remaining suffixes have lengths $n - 1, n - 2, \dots, 1$.
3	$\Theta(n)$	One constant-time front removal per original item.
4	$\Theta(n \log n)$ worst case	Integer heap comparisons across insertion and extraction; top queries alone would be cheaper.
5	$\Theta(n + \sum w_i)$	Each iteration completes exactly one unit of job work.

Here Θ states the tight growth class under the stated operation model. The corresponding O upper bounds are also valid answers when accompanied by the same reasoning. A single reallocating append can be linear even though part 1 is linear in total.

Drill 11 – A hidden copy inside a cheap query

With positive n and q , ordinary by-value passing from an existing vector adds $O(qn)$ copy work. Only one parameter copy is alive per sequential call, so peak additional container storage is $O(n)$, not $O(qn)$. The `size()` query inside the body is itself constant-time.

A const-reference parameter avoids the element copy, so q size-only calls take $O(q)$ modeled time and $O(1)$ extra container storage. A summing helper still needs $O(n)$ scanning work per call, even with a const reference. For empty vectors, include fixed call/setup work with $n + 1$.

This counts source-level operations rather than relying on a compiler to remove a whole computation. Complexity analysis should explain the algorithm's work, not depend on a particular optimization eliminating unused results.

Drill 12 – Check four tempting shortcuts

Claim 1 is false. `reserve` may prevent relocation, but appending still invalidates the old past-the-end iterator used by the range-based loop. Use a separate output or a carefully designed indexed loop.

Claim 2 is false. `auto&` creates a reference, not a copy. Removing the front destroys the element it refers to. Use `auto next = waiting.front();` before removal when a saved value is required.

Claim 3 is false. `>=` returns true for equal first fields, including comparing an item with itself. This violates the strict ordering required by `sort` and does not supply a valid tie rule.

Claim 4 is true under the stated fixed-cost model. Retain a min-heap of at most k candidates, temporarily $k+1$ during an update. Discarding its smallest excess candidate preserves the largest k . The result has $O(k)$ storage and $O(n \log(k + 1))$ total amortized time.

4.33 Quick reference, glossary, and readiness checklist

These pages are for revision after you understand the examples. When a declaration looks unfamiliar, return to the explanation rather than memorizing it without a purpose.

The six declarations

Fragments – declarations inside a function; include the named headers above main:

```
// <vector>: growable indexed sequence
std::vector<int> values;
std::vector<int> scores(5, 0);

// <utility>: two related values
std::pair<int, int> point{3, 7};

// <deque>: indexed sequence with two accessible ends
std::deque<int> recent;

// <queue>: first in, first out
std::queue<int> waiting;

// <queue>: largest value first
std::priority_queue<int> largest;

// <queue>, <vector>, <functional>: smallest value first
std::priority_queue<int, std::vector<int>,
                    std::greater<int>> smallest;

// <stack>: last in, first out
std::stack<int> history;
```

The type inside angle brackets describes an element. A pair has two element types; a vector of pairs uses `std::vector<std::pair<int, int>>`. You do not have to write a template definition to use these types.

The four questions before choosing a tool

Access: do I need an index, either end, the earliest item, the latest item, or the highest-priority item?

Change: what will be inserted or removed, and where? What remains true after that change?

Safety: what happens when the collection is empty or a requested index does not exist?

Cost: how many operations happen in total, including copies, shifts, heap repairs, and reinsertions?

Read the interface, not just the type name

Tool	Read	Add / remove	Reminder
Vector	<code>v[i]</code> , <code>front()</code> , <code>back()</code>	<code>at(i)</code> , <code>push_back(x)</code> , <code>pop_back()</code> , <code>insert</code> , <code>erase</code>	Indices refer only to existing elements.
Pair	<code>p.first</code> , <code>p.second</code>	Assign either field or the whole pair.	No <code>size()</code> or <code>push()</code> .
Deque	<code>d[i]</code> , <code>front()</code> , <code>back()</code>	<code>push_front(x)</code> , <code>push_back(x)</code> , <code>pop_front()</code> , <code>pop_back()</code>	Front and back are different ends.
Queue	<code>front()</code> , <code>back()</code>	<code>push(x)</code> , <code>pop()</code>	Pop removes the front.
Priority queue	<code>top()</code>	<code>push(x)</code> , <code>pop()</code>	Default integer top is the maximum.
Stack	<code>top()</code>	<code>push(x)</code> , <code>pop()</code>	Pop removes the newest item.

All five containers/adaptors provide `empty()` and `size()`. Vector and deque also provide iterators and `clear()`; the three adaptors do not provide public indexed access, iteration, or `clear()`.

Safe removal pattern

```
// Inside a function; waiting is a queue<int>.
if (!waiting.empty()) {
    int next = waiting.front(); // Save the value first.
    waiting.pop();             // Removes; returns no value.
    std::cout << next << '\n';
}
```

For a stack or priority queue, use `top()` instead of `front()`. Checking emptiness is necessary before accessing or removing an element, unless a previously established condition already guarantees one exists.

A few algorithm patterns

```
// <algorithm>, inside a function; v is vector<int>.
std::sort(v.begin(), v.end());
std::reverse(v.begin(), v.end());
auto found = std::find(v.begin(), v.end(), target);
if (found != v.end()) v.erase(found);

// <numeric>: the initial value also selects the sum type.
long long total = std::accumulate(v.begin(), v.end(), 0LL);
```

Sorting changes order, not length. Finding returns an iterator, not an index or a truth value. Erasing one found element does not erase every equal element.

Short glossary

Term	Meaning in this chapter
Container	An object that stores a collection of elements.
Element	One stored value, such as an integer or pair.
Adaptor	A restricted interface built over another container.
Size	The number of elements that currently exist.
Capacity	The number of vector elements that fit before more storage is needed.
Index	A zero-based position used for indexed access.
Iterator	An object representing a position in a sequence.
Dereference	Access the element at an iterator, using <code>*it</code> .
Past-the-end	The position just after the last element; not an element to read.
Half-open range	A range including its first position but excluding its last.
Reference	Another way to refer to an existing object, not a separate copy.
Invalidation	An operation makes a stored iterator, pointer, or reference no longer usable in the old way.
Reallocation	A vector obtains another storage area and relocates its elements.
Amortized cost	A bound averaged over a sequence of operations, not an average over random inputs.
FIFO	First in, first out: queue order.
LIFO	Last in, first out: stack order.
Heap	A structure that efficiently maintains a priority extremum; not a fully sorted sequence.
Lexicographical	Compare the first differing field or element; shorter equal prefixes come first for sequences.
Tie-breaker	A secondary rule used when the primary values are equal.
Invariant	A property that stays true after each step of an algorithm.
Precondition	Something that must already be true before an operation is valid.
Auxiliary space	Extra storage used by an algorithm, under the stated input/output accounting convention.

Readiness checklist: explain before you answer yes

1. I can explain the difference between `vector<int> v(5);` and `vector<int> v{5};`.
2. I can read a vector with existing elements, or build an empty vector with `push_back`, without accidentally combining the two strategies.
3. I can check valid indices and explain why `reserve` does not make `v[0]` valid in an empty vector.
4. I can use a range loop to read values, modify values by reference, or inspect expensive elements by const reference.
5. I can explain what `begin()` and `end()` mean and why `end()` must not be dereferenced.
6. I can predict the order of sorted integer pairs, a max-priority queue of pairs, and a min-priority queue of pairs.
7. I can trace a queue, a stack, and a priority queue on the same inputs and explain their different next items.
8. I can copy a value before removal, handle an empty state, and avoid treating `pop()` as a returned value.
9. I can explain why a vector append is amortized constant time, and why repeatedly erasing its first element is not.
10. I can justify the invariant in a bracket checker, a recent-history deque, or a queue simulation.
11. I can reset state between test cases and account for copies rather than calling every short-looking function constant time.
12. I can choose a suitable tool for a new problem and defend the choice using required operations.

A practical revision route

First redo one vector problem without looking at the answer. Then trace a pair-ordering problem and a deque command sequence. Finally solve one FIFO problem, one priority problem, and one LIFO problem. For each, write one sentence describing the invariant and another describing the total cost.

The optional challenges are not a deadline for understanding the core chapter. Return to structured bindings, custom sorting, and bounded heaps after the ordinary operations are comfortable.

What comes next

Session 5 will introduce sets and maps: tools for maintaining ordered or searchable collections and associating keys with values. The collected book in this release contains Sessions 1–4; Sessions 5 and 6 remain on the roadmap. The habits to carry forward are unchanged: know the interface, state the ordering rule, check the preconditions, and count the actual operations.

4.34 Technical notes and references

These notes document the conventions behind the beginner-facing explanations. They are not extra prerequisites for the core practice.

Language version and source scope

The programs target **C++23**. Language and standard-library links point to the fixed **N4950 C++23 working draft**, in a public HTML rendering of committee wording, rather than to a moving draft that may describe later versions. These are technical references, not beginner tutorials. The GCC and libstdc++ links are official implementation documentation.

The chapter uses ordinary standard headers and default allocator/container choices. It intentionally avoids user-defined templates, custom heap comparator classes, ranges pipelines, and advanced move semantics. A named comparison function for `sort` and structured bindings appear only as optional extensions.

Bounds, element costs, and integer types

The verification environment has a 32-bit `int` and 64-bit `long long`. C++ does not require every implementation's `int` to be exactly 32 bits. The chapter's 200000-element and 10^9 -value limits assume an `int` wide enough for these bounds, as in the tested environment. Wide totals use `long long`.^[4-36]

Contest input follows each problem's stated format and constraints. Invalid *commands*, where the statement specifies an answer such as `EMPTY` or `INVALID`, are part of the algorithm; malformed input streams are not. A negative length is never passed to a container constructor. ASCII assumptions apply to text exercises.

The usual cost model treats arithmetic, comparisons, and copies of fixed-width integers or fixed-size integer pairs as constant work. Operations on long strings or other expensive element types can cost more. Allocation details and debug instrumentation are not included in the introductory operation model.^[4-4]

Vector is not one universal storage model for every T

The ordinary vectors used here have contiguous elements. `vector<bool>` is a specialized representation with proxy-reference behavior; it is not used in any example. Do not transfer every `int&` or raw-storage example to that specialization without studying its separate rules.^{[4-1][4-35]}

A `vector<vector<int>>` stores row objects in the outer vector; each row owns its own sequence. The integer cells of different rows need not form one contiguous rectangle. Rows may also have different lengths. The cost of copying the matrix includes copying its stored cells, not just its number of row objects.

What the complexity claims do and do not guarantee

For vector, the standard gives amortized constant-time append, not a fixed growth factor or a promise that every append is constant time. The doubling illustration explains how amortization works; it is not an assertion that all implementations double capacity. A request to `shrink_to_fit` is non-binding.^{[4-1][4-5]}

The default priority queue uses a vector. Its `push` performs vector insertion and then heap repair. Heap repair has logarithmic comparison cost, but an individual vector-growing call can also relocate existing elements. Thus this chapter distinguishes *amortized logarithmic total push work* from the possibly linear cost of one growing call. Pop repairs the heap and removes the back element.^{[4-17][4-18][4-37]}

The standard complexity clause for `sort` guarantees $O(n \log n)$ comparisons for the ordinary overload. It does not in that clause specify a universal auxiliary-space bound. Where a solution makes a vector copy, we explicitly count that $O(n)$ storage and separately mention the sorting implementation's workspace rather than promising that every conforming sort implementation uses a particular stack size.^[4-8]

The queue and stack examples use their default deque backing. Their endpoint costs follow the underlying container's operations. With a different backing container, revisit that container's guarantees instead of mechanically reusing the same per-call explanation.^{[4-14][4-16][4-19]}

Copies, returns, and lifetime

Passing an existing vector into a by-value parameter makes a separate vector value; passing it by reference does not make that element copy. Returning a locally constructed vector is not a dangling reference: the function returns a value, with move/copy-elision rules relevant to the transfer. This chapter does not promise a mandatory extra n -element copy at every return.^{[4-4][4-30][4-33][4-34]}

An invalidated iterator or a reference to an erased element must not be used merely because a previous run appeared to work. Vector reallocation and erasure rules differ from deque rules. A range-based loop caches its end position, so appending to the same vector while using that loop is not repaired simply by reserving spare capacity.^{[4-6][4-15][4-20]}

What a comparison actually orders

Ascending pair comparison is lexicographical: first field, then second. A min-heap of pairs follows that entire order, while a default max-heap reverses both priorities. Neither promises arrival order for equal-priority records. For a mixed rule such as score descending and ID ascending, express both parts deliberately. Sorting predicates must define a strict weak ordering; $\geq$ is not a substitute for $>$.^{[4-13][4-22][4-28]}

What was checked for this edition

The source package contains **34 complete example programs** and **22 complete practice solutions**. All 56 compiled with GCC 14.2.0 in C++23 mode and passed **5899 input/output checks**: 99 fixed cases, 5780 generated or exhaustive-small-case checks, and 20 large boundary checks.

The tests cover empty states, one-element inputs, duplicate values, negative values where allowed, absent targets, invalid requested indices, pair ties, both heap directions, fully removed strings, mismatched bracket types, repeated test cases, wide totals, and maximum-sized inputs. Generated expected results are computed independently by the Python runner. Its source and report record the fixed random seed, per-program counts, and source-file hashes.

Two verification profiles, both with undefined-behavior checks

The **fixed and generated/exhaustive checks** use:

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

The **20 large boundary checks** use the same flags **except** `-D_GLIBCXX_DEBUG`. Undefined-behavior checks and lightweight standard-library assertions remain enabled. Full libstdc++ debug mode can add linear heap-precondition scans; repeating those checks on large heaps measures the extra checking rather than the ordinary heap algorithm. The test report records both flag sets.^{[4-26][4-27][4-38]}

These are correctness checks, not timing benchmarks. Passing finite tests is not a mathematical proof for every allowed input. Instrumentation can reveal faults on executed paths, but it cannot establish that every possible error is absent. Deliberately wrong teaching fragments are labelled and excluded from the complete-program suite.

Normal learning build

From the source-package root, this compiles one complete example without the extra verification instrumentation:

```
g++ -std=c++23 -Wall -Wextra -O2 \  
    examples/session_04/18_queue_commands.cpp -o demo  
./demo
```

Compile one `.cpp` file at a time: each complete example and solution has its own `main`. The language mode is selected explicitly rather than relying on a compiler's default.^[4-25]

Reproduce the checks and rebuild the PDFs

From the project root, run:

```
python3 tests/verify_session_04.py
make session4
make book
```

The test command needs Python 3 and a C++23-capable GNU compiler. The Makefile uses `latexmk` and `pdfLaTeX` with the packages listed in the README. `make test` runs the suites for all four completed sessions; `make all` builds their separate PDFs and the collected book.

`session_04.tex` loads Session 4 alone with chapter number 4. `book.tex` includes the same chapter after Sessions 1–3. The shared chapter loads the printed listings directly from the tested `.cpp` files, so an edited program is not duplicated in an unrelated LaTeX listing. The existing reports describe the checked snapshots; rerun the suite after editing programs.

How to read the source markers

A superscript such as ^[4-6] links to the matching entry below. The leading 4 distinguishes this session's references from those in the earlier chapters. The technical sources support language and library rules. Teaching examples, problem statements, traces, solutions, and their explanations were written for this training.

Language and standard-library references

[4-1] C++23 N4950: vector overview and guarantees

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

[4-2] C++23 N4950: vector constructors

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

[4-3] C++23 N4950: sequence-container requirements

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

[4-4] C++23 N4950: general container requirements

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

[4-5] C++23 N4950: vector size, capacity, reserve, and resize

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

[4-6] C++23 N4950: vector modifiers and invalidation

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

[4-7] C++23 N4950: iterator requirements and ranges

<https://timsong-cpp.github.io/cppwp/n4950/iterator.requirements.general>

[4-8] C++23 N4950: sort

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

[4-9] C++23 N4950: reverse

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

[4-10] C++23 N4950: non-modifying operations, section 27.6.6 (find)

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

[4-11] C++23 N4950: accumulate

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

[4-12] C++23 N4950: pair utility

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

[4-13] C++23 N4950: pair comparison and specialized operations

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

[4-14] C++23 N4950: deque

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

[4-15] C++23 N4950: deque modifiers and invalidation

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

[4-16] C++23 N4950: queue definition

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

[4-17] C++23 N4950: priority queue overview

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

[4-18] C++23 N4950: priority queue push and pop

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

[4-19] C++23 N4950: stack definition

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

[4-20] C++23 N4950: range-based for statements

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

[4-21] C++23 N4950: structured bindings

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

[4-22] C++23 N4950: sorting and strict weak ordering

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

[4-23] C++23 N4950: vector erasure functions

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

[4-24] C++23 N4950: unique

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

[4-25] GCC: selecting a language standard

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

[4-26] GCC: instrumentation and undefined-behavior sanitizer

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

[4-27] libstdc++: using debug mode

https://gcc.gnu.org/onlinedocs/libstdc++/manual/debug_mode_using.html

[4-28] C++23 N4950: comparison function objects

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

[4-29] C++23 N4950: placeholder type specifiers (auto)

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

[4-30] C++23 N4950: return statements

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

[4-31] C++23 N4950: optional container comparison requirements

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

[4-32] C++23 N4950: type aliases

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

[4-33] C++23 N4950: function calls and argument passing

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

[4-34] C++23 N4950: copy elision

<https://timsong-cpp.github.io/cppwp/n4950/class.copy.elision>

[4-35] C++23 N4950: vector<bool> specialization

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

[4-36] C++23 N4950: fundamental types and minimum widths

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

[4-37] C++23 N4950: heap operations

<https://timsong-cpp.github.io/cppwp/n4950/alg.heap.operations>

[4-38] libstdc++: configuration macros and assertions

https://gcc.gnu.org/onlinedocs/libstdc++/manual/using_macros.html