

Level 1 Training

Session 3

Bit Manipulation and Bitmasking

Binary representation, XOR,
and compact sets of choices

Give every bit a meaning.
Choose the operation. Check the width.

How to use this reference

Level 1 is about choosing an algorithm and explaining why it works. This book is a reference to study **after the session**. Read a little, trace a small example, and then write the idea yourself. A short expression is useful only when you can explain what information it preserves.

A study routine for compact representations

1. **Name the state.** Decide whether a bit represents a numeric weight, a feature, an input position, or a reachable total. These meanings are not interchangeable.
2. **Draw a small example.** Write the bit positions above the digits. Work out one operation by hand before combining several operations.
3. **State the contract.** Record valid indices, the type's width, and any promise about duplicates, signs, or value bounds.
4. **Test and rebuild.** Include zero, the highest permitted bit, repeated choices, an empty subset, and a case where an assumption fails. Try the practice before opening its hint.

How examples are labelled

A **complete example** or **complete solution** has headers and its own `main`. Its listing is loaded directly from an included `.cpp` file. Compile one program at a time. A **fragment** illustrates one smaller idea; its intended location is stated. Deliberately wrong examples are not templates to copy.

Pause and predict questions have explained answers nearby. Cover the answer first. **Optional extensions** and Practice 18 to 20 are for a second reading. Main topics and coding solutions begin on fresh pages.

Representation does not replace reasoning

A bitmask can store many yes/no choices compactly. It does not make all 2^n subsets cheap to enumerate, and it does not preserve frequencies unless frequencies are explicitly represented elsewhere.

C++23 and prerequisites

All complete programs use **standard C++23 headers**, without GNU extension headers. You should know loops, vectors, strings, functions, references, prefix summaries, and basic complexity analysis. The final optional backtracking example reuses Session 2. Binary notation is explained from the beginning; formal combinatorial formulas and dynamic programming are not prerequisites for the core material.

The Level 1 learning path

Session	Topics
1	Two Pointers and Binary Search. Part A: two pointers and sliding windows. Part B: sorted lookup, boundaries, and search on the answer.
2	Recursion and Backtracking. Calls, decision trees, enumeration, pruning, and state restoration.
3	Bit Manipulation and Bitmasking. Binary values, bitwise operations, XOR, compact sets, and subsets.
4	Counting Principles. Addition, multiplication, nCr , nPr , and inclusion-exclusion.
5	Number Theory, Part 1. Divisors, prime factorization, and the sieve.
6	Number Theory, Part 2. Modular arithmetic, binary power (exponentiation), modular inverse, GCD, and LCM.

What this edition contains

This handout contains **Session 3: Bit Manipulation and Bitmasking**. Its section numbers match the collected book, which contains Sessions 1 to 4. Sessions 5 and 6 are planned Number Theory sessions and are not included yet.

What Session 3 will help you do

Read binary digits, change selected bits safely, derive XOR cancellation, represent small sets, enumerate subsets without losing empty or repeated-position choices, and decide when a mask is an appropriate representation. Explain the type, the universe of valid bits, and the actual cost of every loop.

Conventions

Bit positions start at 0 from the **least significant** end. The alias U means **unsigned long long**; W means its number of value bits. The tested GNU environment has $W = 64$. Programs use the bounds stated beside them and assume valid input; no prompts are printed.

A bitmask's decimal value is not a selected item's index. **EMPTY** denotes a valid empty selection; **NONE** or a specified numeric sentinel denotes no answer. Signed weights and sums use **long long**, not unsigned masks. Complexity uses a fixed-width word model unless a section explicitly discusses a variable-width bitset.

Contents

How to use this reference	i
The Level 1 learning path	ii
3 Bit Manipulation and Bitmasking	1
3.1 Binary digits and place values	2
3.2 Converting representations without changing values	3
3.3 Types, widths, and unsigned masks	5
3.4 AND, OR, and XOR: one bit at a time	6
3.5 Shifts: move bits and respect the bounds	8
3.6 Logical operators and precedence	10
3.7 Check, set, clear, and toggle one bit	11
3.8 Low-bit masks and bounded complements	13
3.9 Clear the lowest set bit and count ones	15
3.10 Power tests and C++ bit utilities	17
3.11 Isolate one bit and visit selected positions	19
3.12 XOR cancellation and one odd occurrence count	21
3.13 Two odd values: separate before cancelling	23
3.14 Prefix XOR and interval queries	25
3.15 XOR of a consecutive integer range	27
3.16 Masks as small sets	29
3.17 Letter masks: presence is not frequency	31
3.18 Enumerating every subset exactly once	33
3.19 Signed subset sums, zeros, and duplicates	35
3.20 Partitioning and optimizing over subsets	37
3.21 Two universes: selected people and covered skills	39
3.22 Choosing recursion or mask enumeration	41
3.23 Fixed-size bitsets beyond a machine word	42
3.24 Enumerating submasks, including zero	44
3.25 Count the real work and reset each test case	46
3.26 Debug the type, the meaning, and the boundary	47
3.27 Optional extension: subset totals from smaller masks	48
3.28 Optional extension: count per-bit pair contributions	50
3.29 Optional extension: reachable sums with a bitset	52
3.30 Optional extension: backtracking with copied masks	54
3.31 Optional extension: powers, rotations, and compiler builtins	56
3.32 Practice problems: state the meaning of each bit	57
3.33 Trace, correctness, and complexity drills	67
3.34 Hints before solutions	73
3.35 Explained practice solutions	77
3.36 Explained trace and complexity drills	97
3.37 Quick reference, glossary, and readiness checklist	103
3.38 Technical notes, verification, and references	107

SESSION 3

Bit Manipulation and Bitmasking

Give every bit a meaning. Choose the operation. Check the width.

In Session 2, a subset came from a sequence of take-or-skip decisions. This session stores those decisions as the digits of an integer. The same representation can describe enabled features, selected positions, or a compact state used by another algorithm.

We will first learn what binary digits mean, then manipulate one bit at a time. Only after that will we combine bits into sets and enumerate choices. You do not need to memorize a page of tricks before you understand the first example.

Three different roles for bits

Role	Meaning of bit i	Typical use
Numeric value	The number includes the weight 2^i .	Read or change a binary digit.
Small set	Feature or item i is present.	Union, intersection, permissions.
Search choice	Input position i is selected.	Enumerate and evaluate subsets.

Your route through the chapter

Stage A: understand individual bits. Read binary values, choose an unsigned type, apply operators, respect shift bounds, and build masks.

Stage B: use algebra and representation. Derive XOR cancellation, answer interval queries, model sets, and distinguish presence from frequency.

Stage C: explore choices. Enumerate subsets and submasks. Solve counting, partition, and coverage tasks while preserving the exact meaning of a selection.

Stage D: practise and extend. Use the problems, hints, and drills. Optional sections connect masks to precomputed subset totals, per-bit counting, reachable sums, and backtracking.

The sentence to write before using a mask

“Bit i is 1 exactly when ...; the valid positions are ...; bits outside that range must”
Complete all three parts before choosing an operation.

3.1 Binary digits and place values

The same idea as decimal notation

In decimal, moving one position to the left multiplies the position's weight by 10. In binary, it multiplies the weight by 2. A binary digit is a **bit**; its value is either 0 or 1. Bit i contributes 2^i when it is set to 1, and contributes nothing when it is 0.^[3-1]

For example, $1101_2 = 1 \cdot 8 + 1 \cdot 4 + 0 \cdot 2 + 1 \cdot 1 = 13$. The subscript 2 says that the written digits are binary; it is not part of C++ syntax.

Position i	7	6	5	4	3	2	1	0
Weight 2^i	128	64	32	16	8	4	2	1
Bits of 13	0	0	0	0	1	1	0	1

The **least significant bit** is bit 0, at the right of this written representation. The highest set bit of 13 is bit 3. Its index is 3, its value is 8, and the whole number is 13. These are three different answers.

Leading zeros do not change a number

The strings 1101, 00001101, and a 64-character version with additional leading zeros describe the same nonnegative value. The chosen display width helps us align columns; it does not change the integer.

For n available bits, every bit independently chooses 0 or 1. There are 2^n patterns, with values from 0 through $2^n - 1$. The largest pattern has every bit set:

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

The number of patterns is one more than the largest pattern's value. This distinction will matter when we write a loop over masks.

A bit diagram is not a memory-byte diagram

We write significant digits on the left, as in ordinary numbers. This says nothing about the order in which an implementation stores the bytes of a multi-byte object. You do not need byte order to solve the mask problems in this chapter.

Pause and predict: 1

Which positions are set in 10110_2 ? What is its decimal value, and how many bits are set?

Check your prediction. The positions are 1, 2, and 4. Their weights add to $2 + 4 + 16 = 22$. There are three set bits, not 22 bits and not five set bits.

3.2 Converting representations without changing values

Extract the rightmost digit, then divide

For a nonnegative integer, the remainder after division by 2 is its last binary digit. Integer division by 2 removes that digit. Repeating this process discovers the digits from right to left, so reverse them before printing.

Current value	Remainder modulo 2	Next value	Digits collected
13	1	6	1
6	0	3	10
3	1	1	101
1	1	0	1011

Reversing 1011 gives 1101. Zero needs one digit too: use a **do-while** so that zero produces 0, not an empty string.

Worked program and checks

Input contract. One nonnegative integer representable in U . Print its binary representation without leading zeros, except for zero itself.

Complete example | C++23

```
#include <algorithm>
#include <iostream>
#include <string>
using U = unsigned long long;
int main() {
    U x;
    std::cin >> x;
    std::string digits;
    do {
        digits.push_back(char('0' + x % 2));
        x /= 2;
    } while (x != 0);
    std::reverse(digits.begin(), digits.end());
    std::cout << digits << '\n';
}
```

Sample input

13

Sample output

1101

The expression $x \% 2$ is 0 or 1. Adding it to '0' selects the corresponding digit character. The program changes its local copy of x ; save another copy first if later work needs the original.

Cost. If the output has B digits, time and temporary string storage are $O(B)$. For nonzero x , $B = \lfloor \log_2 x \rfloor + 1$; zero still uses one digit.

The reverse conversion and C++ literals

To read a binary string yourself, start with `value = 0`. For each digit from left to right, use `value = value * 2 + (digit - '0')`. This shifts the accumulated place values before adding the next digit. The input must contain only 0 and 1, and the result must fit the chosen type.

In C++ source, `0b1101ULL`, `13ULL`, and `0xDULL` express the same value in binary, decimal, and hexadecimal. An integer literal beginning with a plain 0 can be octal: do not write `0101` expecting a binary literal. Ordinary `std::cout << x` prints an integer in decimal unless you change its formatting.^[3-14]

Pause and predict: 2

Does `std::cout << 0b1010ULL`; print the four characters 1010?

Check your prediction. No. The literal's value is ten, and ordinary integer output prints 10. Source notation and output formatting are separate choices.

3.3 Types, widths, and unsigned masks

Choose the representation before the operation

Most masks in this chapter use the alias below. An alias is another name for a type; it does not create a different kind of integer.

```
using U = unsigned long long;
U mask = 0;
```

An unsigned integer with W value bits represents 0 through $2^W - 1$. Its arithmetic is modulo 2^W : a result is reduced to that range. `unsigned long long` has at least 64 value bits; the environment used to verify these programs has exactly 64. Query the type instead of assuming every C++ implementation uses the same widths.^[3-1]

Fragment inside a function, with `<limits>` included:

```
constexpr int W = std::numeric_limits<U>::digits;
U largest = std::numeric_limits<U>::max();
U one = 1ULL;
```

The type of the expression matters now, not later

`1ULL` makes the literal `unsigned long long`. A plain `1` has type `int`. Assigning a result to `U` does not retroactively make the earlier computation wider.^[3-14]

Wrong example on the tested 32-bit-int environment

`U bit = 1 << 40;` shifts an `int`, not a `U`. The shift count is outside that type's width. Use `1ULL << 40` instead. A wide destination cannot repair an invalid expression.

Keep signed mathematical quantities separate. A subset can have a nonnegative mask and a negative total when selected input values are negative. Use `U` for the selection and `long long` for its sum. Casting a negative sum to unsigned does not fix it; it changes its numeric interpretation.

Small types can be promoted

Operations on `char`, `unsigned char`, and `short` can first promote them to `int`. A tiny variable therefore does not automatically make an operation an 8-bit operation. In the core examples, `U` avoids this extra layer; the technical notes explain the narrow-type trap.^[3-9]

Pause and predict: 3

On a machine with 64-bit `U`, what are the largest valid bit index and the number of different mask values?

Check your prediction. The indices are 0 through 63, so the largest index is 63. There are 2^{64} different values, but the largest value is $2^{64} - 1$. The count 2^{64} itself cannot be stored in that 64-bit type.

3.4 AND, OR, and XOR: one bit at a time

Start with the truth table

These operators compare corresponding binary positions. There is no carry into a neighbouring position.^{[3-4][3-5][3-6]}

a	b	$a \text{ AND } b$	$a \text{ OR } b$	$a \text{ XOR } b$
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

AND, written `&`, keeps a bit only when both inputs have it. **OR**, written `|`, keeps a bit when either input has it, including when both do. **XOR**, written `^`, keeps a bit when the inputs differ.

For $12 = 1100_2$ and $10 = 1010_2$, AND gives $1000_2 = 8$, OR gives $1110_2 = 14$, and XOR gives $0110_2 = 6$.

NOT, written `~`, is unary: it flips every bit in the promoted operand's type. It is not limited to the short binary string you happened to draw. To flip only eight positions, keep the low eight result bits with `& 255ULL`.^[3-3]

Worked program and checks

Input contract. $0 \leq a, b \leq 255$. The first two output lines are eight-bit displays. The third line prints AND, OR, and XOR in decimal; the last prints the eight-bit complement of a . `bitset<8>` is used only as a display aid here and is explained later.

Complete example | C++23

```
#include <bitset>
#include <iostream>
using U = unsigned long long;
int main() {
    U a, b;
    std::cin >> a >> b;
    std::cout << std::bitset<8>(a) << '\n';
    std::cout << std::bitset<8>(b) << '\n';
    std::cout << (a & b) << ' ' << (a | b) << ' '
               << (a ^ b) << '\n';
    std::cout << ((~a) & 255ULL) << '\n';
}
```

Sample input

12 10

Sample output

00001100

00001010

8 14 6

243

XOR is not exponentiation

C++ has no built-in $\wedge$ operator for powers. $2 \wedge 3$ is XOR and has value 1, not 8. For a safe power of two, we will use an appropriately typed left shift.

OR is also not ordinary addition. For disjoint set bits it happens to match addition, but $3 \mid 1$ is 3, whereas $3 + 1$ is 4. Adding an already-present feature can carry into another feature's position.

Pause and predict: 4

For $a = 0101_2$ and $b = 0011_2$, find AND, OR, and XOR.

Check your prediction. AND is 0001 (1), OR is 0111 (7), and XOR is 0110 (6). The shared final 1 survives AND and OR, but cancels under XOR.

3.5 Shifts: move bits and respect the bounds

Left shift and right shift

For unsigned x , $x \ll k$ moves its bits k positions towards larger weights, introducing zeros at the low end. If all significant bits still fit, its value is $x \cdot 2^k$. Bits that leave the type's width are discarded. $x \gg k$ moves towards smaller weights and equals $\lfloor x/2^k \rfloor$.^[3-2]

Expression	Eight-bit illustration	Value
13ULL	00001101	13
13ULL $\ll$ 2	00110100	52
13ULL $\gg$ 2	00000011	3

The illustration uses eight columns for readability. The actual type in these expressions is **unsigned long long**.

A valid shift count is strictly below the width

For built-in shifts, the right operand must be nonnegative and smaller than the width of the promoted left operand. Shifting by exactly W is invalid, even when the left value is zero. A CPU's apparent behaviour on one run does not make an out-of-range shift valid C++.^[3-2]

Worked program and checks

Input contract. $0 \leq x \leq 10^9$ and $0 \leq k \leq 30$. These bounds make both the count and the left-shifted value safe on any U with at least 64 value bits.

Complete example | C++23

```
#include <iostream>
using U = unsigned long long;
int main() {
    U x;
    int k;
    std::cin >> x >> k;
    std::cout << (x << k) << '\n';
    std::cout << (x >> k) << '\n';
    std::cout << (1ULL << k) << '\n';
}
```

Sample input

13 2

Sample output

52

3

4

Do not lose information accidentally

On a 64-bit U , $(1ULL \ll 63) \ll 1$ is a valid unsigned shift with result zero. It is not the mathematical integer 2^{64} . Defined wraparound is still wrong when a task requires an exact larger value.

$x \ll k$ produces a value without changing x . Use $x \ll= k$ to store the result back. With streams, `std::cout << x` means output: the operand types determine which operator operation is used.

For masks, stay unsigned. In C++23, signed right shift of a negative integer rounds division by 2^k down: $-3 \gg 1$ is -2 , while signed integer division $-3 / 2$ is -1 . A shift is not a universal replacement for division on negative inputs.^[3-2]

Pause and predict: 5

On a 64-bit U , which is valid: $1ULL \ll 63$, $1ULL \ll 64$, or $0ULL \gg 64$?

Check your prediction. Only the first. Validity depends on the shift count, not on whether the value is small or zero.

3.6 Logical operators and precedence

A yes/no condition is not a mask operation

`&&`, `||`, and `!` operate on truth values. `&`, `|`, and `~` operate on bits of integers. For example, 2 and 4 are both nonzero, so `2 && 4` is true. Their set bits do not overlap, so `2 & 4` is zero.^{[3-4][3-11]}

Logical AND short-circuits: when its left side is false, its right side is not evaluated. Bitwise AND provides no such guard. Do not replace `i < n && a[i] == x` with `i < n & a[i] == x`; the latter cannot protect the array access.^[3-11]

Make the intended grouping explicit

Equality comparisons bind more tightly than bitwise AND. Consequently, `x & bit == 0` groups as `x & (bit == 0)`, not as the intended `(x & bit) == 0`. With nonzero `bit`, the wrong expression tests against zero regardless of `x`.^[3-4]

Fragments inside a function:

```
bool present = (mask & (1ULL << k)) != 0;
bool absent = (mask & (1ULL << k)) == 0;
bool hasAll = (mask & required) == required;
```

Use parentheses around bitwise expressions that you compare or print. For example, write `std::cout << (a ^ b);`. Do not ask a reader to reconstruct a precedence table while checking your algorithm.

A membership test need not return 1

If bit 5 is present, `mask & (1ULL << 5)` returns 32, not 1. Testing whether this result is nonzero is correct. Comparing it with 1 would incorrectly reject every present bit except bit 0.

Do not rely on clever-looking shorthand

The extra parentheses in `(mask & bit) != 0` communicate the algorithm. They are not wasted characters. Shortening a condition is not an optimization when it changes its meaning.

Pause and predict: 6

A mask equals 8 and `bit` equals 8. Why is `(mask & bit) == 1` false even though the bit is present?

Check your prediction. The intersection is 8, the weight of bit 3. Presence means the intersection is nonzero, not necessarily equal to 1.

3.7 Check, set, clear, and toggle one bit

Build one selected position

Let `bit = 1ULL << k`, with $0 \leq k < W$. This mask has exactly one set bit. Combining it with `mask` changes or checks one position while treating every other position predictably.

Job	Expression	Reason at position k
Check	<code>(mask & bit) != 0</code>	AND keeps it only if present.
Set	<code>mask = bit;</code>	OR with 1 forces a 1.
Clear	<code>mask &= ~bit;</code>	AND with 0 forces a 0.
Toggle	<code>mask ^= bit;</code>	XOR with 1 flips the bit.

At every other position, `bit` is 0. OR and XOR with zero preserve the old bit; AND with the corresponding 1 in `~bit` also preserves it.

Worked program and checks

Input contract. Read width q mask, where $1 \leq \text{width} \leq 60$, $0 \leq q \leq 200000$, and $0 \leq \text{mask} < 2^{\text{width}}$. Each command is SET k , CLEAR k , TOGGLE k , or CHECK k , with $0 \leq k < \text{width}$. Print each check as 0 or 1, then the final mask in decimal. The given width is an input guarantee, not a second bitset stored by the program.

Complete example | C++23

```
#include <iostream>
#include <string>
using U = unsigned long long;
int main() {
    int width, q;
    U mask;
    std::cin >> width >> q >> mask;
    for (int i = 0; i < q; ++i) {
        std::string command;
        int k;
        std::cin >> command >> k;
        U bit = 1ULL << k;
        if (command == "SET") mask |= bit;
        else if (command == "CLEAR") mask &= ~bit;
        else if (command == "TOGGLE") mask ^= bit;
        else std::cout << ((mask & bit) != 0) << '\n';
    }
    std::cout << mask << '\n';
}
```

Sample input

```
5 6 5
CHECK 2
SET 1
SET 1
CLEAR 0
TOGGLE 4
CHECK 0
```

Sample output

```
1
0
22
```

Assignment, removal, and inversion are different

Starting at 5, setting bit 1 gives 7. Setting it again still gives 7: setting is **idempotent**. Clearing a missing bit also leaves the mask unchanged. Toggling twice restores the original state, so toggling is not a replacement for setting.

The sample then clears bit 0, leaving 6, and toggles bit 4, giving $6 + 16 = 22$. Time is $O(q)$ and working storage is $O(1)$ under the fixed-word model.

Pause and predict: 7

Starting with zero, compare setting bit 2 twice with toggling bit 2 twice.

Check your prediction. Two sets produce 4. Two toggles produce 0. The first operation means “make present”; the second means “reverse the current answer.”

3.8 Low-bit masks and bounded complements

The universe is part of the representation

Suppose only positions 0 through $n - 1$ represent valid features. Their full mask is $2^n - 1$. For $0 \leq n < W$, compute it with $(1ULL \ll n) - 1ULL$. At $n = 0$, this correctly gives zero: the universe is empty.

If all W positions are allowed, do not evaluate $1ULL \ll W$. Use an all-ones value instead. The helper below handles $0 \leq n \leq W$ and has a **precondition**: callers must not pass a count outside that interval.

Trace the three different values

In the sample, 37 is 100101_2 . The five-bit full mask is $011111_2 = 31$. Restricting 37 gives $000101_2 = 5$. Flipping only the five valid positions gives $011010_2 = 26$. The sixth input bit is outside this universe and does not participate.

`~mask` alone flips the entire type width. `(~mask) & full` first flips and then discards every out-of-universe bit. If `mask` is already a subset of `full`, `full ^ mask` gives the same bounded complement. Without that subset promise, it need not.

Flip a group; do not overwrite the rest

To toggle only the lowest k positions while keeping all higher input bits, use `mask ^= lowMask(k)`. This is different from producing the complement restricted to those positions: the latter deliberately removes the high bits.

Worked program and checks

Input contract. Read n and a value `mask` representable in `U`. Print the low- n full mask, the input restricted to those positions, and the complement within those positions. In the checked 64-bit environment, $0 \leq n \leq 64$.

Complete example | C++23

```
#include <iostream>
#include <limits>
using U = unsigned long long;
U lowMask(int count) {
    constexpr int width = std::numeric_limits<U>::digits;
    if (count == width) return ~0ULL;
    return (1ULL << count) - 1ULL;
}
int main() {
    int n;
    U mask;
    std::cin >> n >> mask;
    U full = lowMask(n);
    std::cout << full << '\n';
    std::cout << (mask & full) << '\n';
    std::cout << ((~mask) & full) << '\n';
}
```

Sample input

5 37

Sample output

31

5

26

Pause and predict: 8

Within a three-item universe, what is the complement of mask 001_2 ? Why not use the unrestricted `~mask`?

Check your prediction. The complement is $110_2 = 6$. An unrestricted complement also sets bits for positions 3 and above, which do not represent any of the three items.

3.9 Clear the lowest set bit and count ones

Look carefully at subtracting one

For a nonzero unsigned value, subtracting 1 turns the lowest 1 into 0 and all less significant zeros into ones. Higher positions remain unchanged. Therefore AND with the original clears exactly that lowest set bit:

x	101100 (44)
x - 1	101011 (43)
x & (x - 1)	101000 (40)

The newly created low ones cannot survive AND because the original low positions were zero. The higher positions survive unchanged. This is a proof about every nonzero pattern, not just about 44.

Input contract. One value representable in U. Print its number of set bits.

Complete example | C++23

```
#include <iostream>
using U = unsigned long long;
int main() {
    U x;
    std::cin >> x;
    int count = 0;
    while (x != 0) {
        x &= x - 1ULL;
        ++count;
    }
    std::cout << count << '\n';
}
```

Sample input

Sample output

44

3

Trace progress, not the decimal magnitude

Count after removal	Remaining value	Remaining pattern
0	44	101100
1	40	101000
2	32	100000
3	0	000000

The loop runs once per set bit, not once per integer from 0 to x . Let p be the number of set bits. Its cost is $O(p + 1)$ including the final check, with $O(1)$ auxiliary space. For a fixed-width integer, $p \leq W$.

A simpler alternative shifts a copy right and adds $x \& 1ULL$ each time. That visits positions up to the highest set bit. Both methods are useful to understand before using a library count.

Pause and predict: 9

How many iterations does the clearing loop need for 2^{50} ? How many for $2^{50} - 1$?

Check your prediction. One for 2^{50} , which has one set bit, and 50 for $2^{50} - 1$, whose low 50 positions are all set.

3.10 Power tests and C++ bit utilities

A positive power of two has one set bit

The pattern of 2^k is one 1 followed by k zeros. Clearing its lowest set bit leaves zero. This gives the test `x != 0 && (x & (x - 1ULL)) == 0`. The explicit zero check matters: zero has no set bits, but it is not a power of two.

The header `<bit>` provides standard helpers for unsigned integer arguments. These are available in C++23; there is no need to use a compiler extension just to count set bits.^{[3-7][3-8]}

Call	Meaning
<code>std::popcount(x)</code>	Number of 1 bits.
<code>std::has_single_bit(x)</code>	Whether x is a positive power of two.
<code>std::bit_width(x)</code>	Positions needed for nonzero x ; returns 0 at zero.
<code>std::countr_zero(x)</code>	Zeros before the lowest 1, scanning from bit 0.
<code>std::countl_zero(x)</code>	Zeros before the highest 1, scanning from the top of the type.

Worked program and checks

Input contract. Read one U value. Print its set-bit count, its power-of-two test as 0 or 1, its bit width, then the lowest set-bit index. Print `NONE` for the final answer when $x = 0$.

Complete example | C++23

```
#include <bit>
#include <iostream>
using U = unsigned long long;
int main() {
    U x;
    std::cin >> x;
    std::cout << std::popcount(x) << '\n';
    std::cout << std::has_single_bit(x) << '\n';
    std::cout << std::bit_width(x) << '\n';
    if (x == 0) std::cout << "NONE\n";
    else std::cout << std::countr_zero(x) << '\n';
}
```

Sample input	Sample output
40	2
	0
	6
	3

For 40, the pattern is 101000_2 : two set bits, six significant positions, and the lowest 1 at index 3. It is not a power of two.

Defined zero results can still need interpretation

Both zero-counting helpers return W for zero. That is a count, not a valid bit index. Our program prints `NONE` because no lowest set bit exists. Similarly, `std::bit_width(0ULL)` is 0 even though a human-readable representation of zero uses the single character 0.^{[3-7][3-8]}

`std::popcount(13)` is not the appropriate call because 13 is signed. Use an unsigned value such as 13ULL. Under the fixed-word model these helpers have bounded word-sized work; do not promise that every compiler emits exactly one machine instruction.

Pause and predict: 10

What should a highest-set-bit helper return for zero? Is `std::bit_width(x) - 1` safe as an index without checking zero?

Check your prediction. There is no highest set bit in zero, so choose an explicit no-answer representation. At zero the expression produces -1 , not a valid index. Check the case before shifting or indexing.

3.11 Isolate one bit and visit selected positions

Clearing and isolating solve different jobs

$x \& (x - 1ULL)$ removes the lowest set bit. To keep only that bit, use $x \& (0ULL - x)$. All operands here are unsigned. The subtraction is defined modulo 2^W , including at zero.^{[3-1][3-3]}

Why does it work? If the lowest set bit of x is at position k , the low k bits are zero. Complementing x and adding one makes those low positions zero again, keeps position k set, and complements the higher positions. AND with x cancels all the higher positions and keeps only 2^k .

Input contract. Read one U. Print the isolated lowest set-bit value and the value with that bit removed. For zero, print 0 0.

Complete example | C++23

```
#include <iostream>
using U = unsigned long long;
int main() {
    U x;
    std::cin >> x;
    U lowest = x & (0ULL - x);
    U withoutLowest = x;
    if (x != 0) withoutLowest &= x - 1ULL;
    std::cout << lowest << ' ' << withoutLowest << '\n';
}
```

Sample input

40

Sample output

8 32

Visit only the members that exist

To list set-bit indices in increasing order, combine a trailing-zero count with clearing. The loop condition guarantees that the count refers to a real set bit.

Fragment inside a function, with `<bit>` and `<iostream>` included:

```
U remaining = mask;
while (remaining != 0) {
    int index = std::countr_zero(remaining);
    std::cout << index << '\n';
    remaining &= remaining - 1ULL;
}
```

This changes `remaining`, not the original mask. If there are p selected positions, the loop has p iterations. A full scan over the universe still has advantages when the order or an unselected position matters.

Do not silently switch this idiom to signed arithmetic

Negating the most negative signed integer is not representable in its signed type. The unsigned version avoids that signed-overflow case. A mask and a signed mathematical value need not use the same type.

Pause and predict: 11

For $x = 40$, what are the lowest set-bit index, its isolated value, and the value after clearing it?

Check your prediction. The index is 3, the isolated value is $2^3 = 8$, and the remaining value is 32. A program asking for an index must not print the isolated value instead.

3.12 XOR cancellation and one odd occurrence count

Derive the identities from one column

For a single bit, XOR with 0 preserves it and XOR with itself produces 0. Applying this independently to every position gives

$$x \oplus 0 = x, \quad x \oplus x = 0.$$

Here $\oplus$ is the mathematical symbol for the C++ operator `^`. XOR is commutative and associative: the order and grouping of its operands do not change the result. Each output bit records whether its input column contains an odd number of ones.^[3-6]

Consequently, a value repeated an even number of times cancels completely. A value repeated an odd number of times has the same net effect as one copy.

Task. Exactly one distinct value occurs an odd number of times, and every other distinct value occurs an even number of times. Find the odd-occurrence value. This is an input promise, not something the algorithm verifies.

Worked program and checks

Input contract. $1 \leq n \leq 200000$, followed by n values representable in U , satisfying that promise.

Complete example | C++23

```
#include <iostream>
using U = unsigned long long;
int main() {
    int n;
    std::cin >> n;
    U answer = 0;
    for (int i = 0; i < n; ++i) {
        U x;
        std::cin >> x;
        answer ^= x;
    }
    std::cout << answer << '\n';
}
```

Sample input

Sample output

7

7

9 4 9 2 4 7 2

The invariant and the proof

After reading i values, **answer** is their XOR. Initially it is zero, the XOR of no values. Each update appends exactly the new value to that expression. At the end, regroup the full XOR by value: every even group cancels, leaving the promised odd group.

Time is $O(n)$ and auxiliary space is $O(1)$. There is no need to store the sequence. A frequency map could verify a broader input contract, but uses more storage.

What XOR does not tell you

For 1 2 3, the XOR is zero even though zero does not occur. With several odd-occurrence values, the result is their combined XOR, not necessarily an input value. This technique is not a general replacement for a frequency table.

Do not confuse “occurs once” with “occurs oddly.” One copy, three copies, and five copies each leave one copy’s contribution in XOR. The common paired-elements problem is one special case.

Pause and predict: 12

What is the XOR of 4 4 7 7 7? Would XOR alone identify the unique value for 4 4 7 9?

Check your prediction. The first XOR is 7 because the two 4s cancel and three 7s leave one. The second leaves $7 \wedge 9$, not either value individually. It violates the one-odd-value promise.

3.13 Two odd values: separate before cancelling

One extra partition makes the information usable

Now exactly two different values have odd occurrence counts. Call them a and b . XOR of the whole input is $a \oplus b$, which is nonzero because $a \neq b$.

Choose any set bit in that combined XOR. At this position, a and b differ, so grouping inputs by that bit sends them to different groups. Equal values always enter the same group, so even-occurrence values still cancel within their groups. Each group contains exactly one odd-occurrence survivor.

Trace a separating bit

The sample's survivors are $9 = 1001_2$ and $12 = 1100_2$. Their XOR is 0101_2 . Its lowest set bit has value 1, so bit 0 separates the survivors: 9 has it, and 12 does not. The paired 5s stay together; so do the paired 2s.

The two passes cost $O(n)$ time. The implementation stores $O(n)$ input values because the separator is known only after the first pass; working state beyond that vector is $O(1)$.

Worked program and checks

Input contract. $2 \leq n \leq 200000$, followed by U values. Exactly two distinct values occur oddly and all others evenly. Print the two odd-occurrence values in increasing numeric order.

Complete example | C++23

```
#include <algorithm>
#include <iostream>
#include <vector>
using U = unsigned long long;
int main() {
    int n;
    std::cin >> n;
    std::vector<U> a(n);
    U combined = 0;
    for (U& x : a) {
        std::cin >> x;
        combined ^= x;
    }
    U separatingBit = combined & (0ULL - combined);
    U first = 0, second = 0;
    for (U x : a) {
        if ((x & separatingBit) != 0) first ^= x;
        else second ^= x;
    }
    if (first > second) std::swap(first, second);
    std::cout << first << ' ' << second << '\n';
}
```

Sample input

6
5 2 5 9 2 12

Sample output

9 12

The combined XOR is nonzero because of the contract

Do not remove the two-odd-values promise and hope the same code discovers arbitrary anomalies. Without it, the separating bit or the interpretation of the two results may be meaningless.

Pause and predict: 13

Could we choose a set bit other than the lowest one in $a \oplus b$? Why do equal duplicate values never split across the two groups?

Check your prediction. Any set bit works because it marks a position where the two survivors differ. Equal values have identical bits, so every copy follows the same grouping rule.

3.14 Prefix XOR and interval queries

Reuse cancellation, not subtraction

Let $\text{pref}[i]$ be the XOR of the first i values, with $\text{pref}[0] = 0$. This definition uses a padded prefix vector and one-based input positions:

$$\text{pref}[i] = \text{pref}[i - 1] \oplus a[i].$$

For an inclusive query $[l, r]$, both $\text{pref}[r]$ and $\text{pref}[l-1]$ contain the values before l . XORing them includes each of those values twice, so they cancel:

$$\text{xor}(l, r) = \text{pref}[r] \oplus \text{pref}[l - 1].$$

Position	0	1	2	3	4	5
Input		3	6	1	6	4
Prefix XOR	0	3	5	4	2	6

For $[2, 4]$, use $\text{pref}[4] \oplus \text{pref}[1] = 2 \oplus 3 = 1$. Directly, $6 \oplus 1 \oplus 6 = 1$.

Worked program and checks

Input contract. $0 \leq n, q \leq 200000$, then n U values, then q valid queries with $1 \leq l \leq r \leq n$. If $n = 0$, then $q = 0$ too. Print one XOR per query.

Complete example | C++23

```
#include <iostream>
#include <vector>
using U = unsigned long long;
int main() {
    int n, q;
    std::cin >> n >> q;
    std::vector<U> pref(n + 1, 0);
    for (int i = 1; i <= n; ++i) {
        U x;
        std::cin >> x;
        pref[i] = pref[i - 1] ^ x;
    }
    while (q-- > 0) {
        int l, r;
        std::cin >> l >> r;
        std::cout << (pref[r] ^ pref[l - 1]) << '\n';
    }
}
```

Sample input

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

Sample output

```
1
6
1
```

Complexity and limitations

Preprocessing is $O(n)$, each query is $O(1)$, and the whole program is $O(n + q)$ time with $O(n)$ prefix storage. The input array need not be sorted. Duplicate values and zeros are valid.

Changing an input value makes affected prefix entries stale. This is a static-query technique unless you rebuild the summaries. Also, an XOR prefix is not a sum prefix: $\text{pref}[r] - \text{pref}[l-1]$ is the wrong cancellation operation.

OR and AND do not have this same cancellation property. Knowing a prefix OR generally cannot tell you which earlier value supplied a particular 1. Do not copy a prefix formula to a different operation without proving an inverse relationship.

Pause and predict: 14

Why does the single-element query $[i, i]$ return exactly $a[i]$? What happens at $l = 1$?

Check your prediction. The expression $\text{pref}[i] \oplus \text{pref}[i-1]$ cancels every earlier value, leaving only $a[i]$. At $l = 1$, the second prefix is $\text{pref}[0]$ and is zero, so no special query branch is needed.

3.15 XOR of a consecutive integer range

A prefix function can have a short formula

Suppose the input is not an explicit array, but the consecutive integers from L through R . Define $F(x) = 0 \oplus 1 \oplus \dots \oplus x$.

Every complete block $4t, 4t + 1, 4t + 2, 4t + 3$ has XOR zero. The first adjacent pair differs only in bit 0 and XORs to 1; the second pair does too. After cancelling complete blocks, only the final incomplete block remains.

$x \bmod 4$	$F(x)$	Example
0	x	$F(8) = 8$
1	1	$F(9) = 1$
2	$x + 1$	$F(10) = 11$
3	0	$F(11) = 0$

The range answer is $F(R) \oplus F(L - 1)$ when $L > 0$, and just $F(R)$ when $L = 0$.

Worked program and checks

Input contract. $0 \leq L \leq R \leq 10^{18}$. Print the XOR of all integers in that inclusive range.

Complete example | C++23

```
#include <iostream>
using U = unsigned long long;
U xorThrough(U x) {
    if (x % 4 == 0) return x;
    if (x % 4 == 1) return 1;
    if (x % 4 == 2) return x + 1;
    return 0;
}
int main() {
    U l, r;
    std::cin >> l >> r;
    U before = (l == 0 ? 0 : xorThrough(l - 1));
    std::cout << (xorThrough(r) ^ before) << '\n';
}
```

Sample input

3 9

Sample output

2

A boundary is not a negative unsigned number

When $L = 0$, computing unsigned $L - 1$ wraps to a large value. Instead, interpret the prefix before zero as empty and give it XOR zero. The conditional operator evaluates only its selected branch, so the subtraction is not evaluated in this case.

The $x + 1$ branch stays safely within U under the stated bound. Time and auxiliary space are $O(1)$ in the fixed-word model. This formula applies to consecutive integers, not to an arbitrary array of the same length.

Pause and predict: 15

Find the XOR of all integers from 0 through 7 and of the single-element range $[6, 6]$.

Check your prediction. The first contains two complete four-number blocks and gives zero. The second must give 6; the formula gives $F(6) \oplus F(5) = 7 \oplus 1 = 6$.

3.16 Masks as small sets

Give each valid item one position

For a universe of n items, assign indices $0, 1, \dots, n-1$. A mask represents the subset whose positions are set. In a five-item universe, mask $01101_2 = 13$ represents items 0, 2, and 3. A vector might list those indices explicitly; the mask stores their membership decisions together.

Once the representation is fixed, set operations follow directly from the bit truth tables:

Set operation	Mask expression	Meaning
Union	$a \mid b$	Present in either set.
Intersection	$a \& b$	Present in both.
Difference $A \setminus B$	$a \& \sim b$	Present in A , absent in B .
Symmetric difference	$a \wedge b$	Present in exactly one.
Complement	$(\sim a) \& \text{full}$	Valid items absent in A .
B is a subset of A	$(a \& b) == b$	Every required bit survives.
Disjointness	$(a \& b) == 0$	No shared members.

Notice why $a \& \sim b$ needs no additional full mask when a is valid: AND cannot create a position that was absent in a . An unrestricted $\sim a$ can create such positions and must be bounded.

Worked program and checks

Input contract. Read n, a, b , with $0 \leq n \leq 60$ and $0 \leq a, b < 2^n$. Print union, intersection, $A \setminus B$, symmetric difference, complement of A , and whether $B \subseteq A$ as 0 or 1.

Complete example | C++23

```
#include <iostream>
using U = unsigned long long;
int main() {
    int n;
    U a, b;
    std::cin >> n >> a >> b;
    U full = (1ULL << n) - 1ULL;
    std::cout << (a | b) << '\n';
    std::cout << (a & b) << '\n';
    std::cout << (a & ~b) << '\n';
    std::cout << (a ^ b) << '\n';
    std::cout << ((~a) & full) << '\n';
    std::cout << ((a & b) == b) << '\n';
}
```

Sample input

5 13 6

Sample output

15

4

9

11

18

0

Presence, count, and order are separate information

Adding the same item twice using OR does not remember two copies. A mask has no ordering of inserted items either. Use a frequency array, multiset, or sequence when those details are required.

For a one-word mask, each set expression costs $O(1)$ under our model. Creating a mask from m listed members still costs $O(m)$: compact storage does not eliminate the input work.

Pause and predict: 16

Does `(have & required) != 0` prove that every requirement is satisfied? What if `required` is zero?

Check your prediction. No. A nonzero intersection proves only that at least one requested feature is present. Use `(have & required) == required`. An empty requirement is satisfied by every set, including the empty set.

3.17 Letter masks: presence is not frequency

A small alphabet gives a natural mapping

For lowercase ASCII letters, assign 'a' to bit 0, 'b' to bit 1, and so on through bit 25. For each character, OR its bit into the word's mask. All copies of the same letter produce the same presence bit.

The word **banana** has six characters but only three distinct letters: **a**, **b**, and **n**. A letter mask can answer whether two words share any letters or how many distinct letters they have in common. It cannot count how often each letter occurs.

Trace presence while reading a word

For **banana**, keep a set of letters on paper before calculating its integer mask:

Character just read	Distinct letters remembered	New presence?
b	{b}	Yes
a	{a, b}	Yes
n	{a, b, n}	Yes
a	{a, b, n}	No
n	{a, b, n}	No
a	{a, b, n}	No

The corresponding set positions are 0, 1, and 13. Their mask is $1 + 2 + 8192 = 8195$, but the number of distinct letters is only 3. The numeric value and the population count answer different questions.

Worked program and checks

Input contract. Two lines, each containing only lowercase ASCII letters, possibly empty. Each line has at most 200000 characters. Print their distinct-letter counts, their number of shared distinct letters, and whether they are disjoint as 0 or 1. The ASCII restriction justifies the mapping `ch - 'a'`.

Complete example | C++23

```
#include <bit>
#include <iostream>
#include <string>
using U = unsigned long long;
U letters(const std::string& text) {
    U mask = 0;
    for (char ch : text) mask |= 1ULL << (ch - 'a');
    return mask;
}
int main() {
    std::string a, b;
    std::getline(std::cin, a);
    std::getline(std::cin, b);
    U first = letters(a), second = letters(b);
    std::cout << std::popcount(first) << ' '
              << std::popcount(second) << '\n';
    std::cout << std::popcount(first & second) << '\n';
    std::cout << ((first & second) == 0) << '\n';
}
```

Sample input

banana

band

Sample output

3 4

3

0

Correctness. After processing a prefix of a word, a bit is set exactly when that letter has appeared in that prefix. ORing the next letter preserves every previous presence and adds the new one. Intersecting the two final masks therefore represents exactly their shared letters.

Time is $O(|a| + |b|)$, including reading the lines. The stored strings use $O(|a| + |b|)$ space; the two letter masks and loop state use $O(1)$ additional space. A streaming variant could avoid storing each entire word.

Pause and predict: 17

Can equality of letter masks prove that `abb` and `aab` are anagrams?

Check your prediction. No. Both contain the same distinct letters, but their frequencies differ. Anagram checking needs counts, not only presence.

3.18 Enumerating every subset exactly once

One mask is one sequence of take-or-skip choices

Use bit i to decide whether input position i is selected. Each position has exactly two choices, so the masks from 0 through $2^n - 1$ describe all subsets. Different masks disagree on at least one selected position; no positional subset is repeated.

Mask value	Bits ($b_2b_1b_0$)	Selected zero-based positions
0	000	None
1	001	0
2	010	1
3	011	0, 1
4	100	2
5	101	0, 2
6	110	1, 2
7	111	0, 1, 2

The binary digits are written with high positions on the left. The loop may inspect positions in increasing order, which is the opposite reading direction on this diagram. Neither convention changes which subset a mask represents.

Worked program and checks

Input contract. $0 \leq n \leq 12$. Print masks in increasing numeric order. Each line starts with the decimal mask and a colon, followed by selected *one-based labels* in increasing order. For mask zero, print EMPTY after the colon.

Complete example | C++23

```
#include <iostream>
using U = unsigned long long;
int main() {
    int n;
    std::cin >> n;
    U limit = 1ULL << n;
    for (U mask = 0; mask < limit; ++mask) {
        std::cout << mask << ':';
        if (mask == 0) std::cout << " EMPTY";
        for (int i = 0; i < n; ++i) {
            if ((mask & (1ULL << i)) != 0)
                std::cout << ' ' << i + 1;
        }
        std::cout << '\n';
    }
}
```

Sample input

2

Sample output

0: EMPTY
1: 1
2: 2
3: 1 2

Loop endpoints and the empty subset

The limit is $1ULL \ll n$, but it is not itself a valid n -bit mask. Use `mask < limit`, not `mask <= limit`. At $n = 0$, the limit is 1, so the loop still visits mask zero once. There is one empty subset, not zero subsets.

Time is $O((n+1)2^n)$, including output. Extra storage is $O(1)$ because each line is printed immediately. Output can be huge, which is why this example deliberately limits n to 12.

Pause and predict: 18

For values 5 5, what do masks 1 and 2 select? Should a positional-subset enumerator merge those masks?

Check your prediction. Mask 1 selects the first 5 and mask 2 the second. They are different position choices, even though their selected value lists look equal. Do not merge them unless the problem asks for distinct value selections instead.

3.19 Signed subset sums, zeros, and duplicates

A nonnegative mask can select negative values

To evaluate a subset sum, scan its selected positions and add the corresponding array values. The mask encodes membership; it does not contain the sum. Use a signed `long long` total because input values may be negative.

Task. Count positional subsets whose selected values sum to a given target. The empty subset is allowed, and input positions remain distinct even when values repeat.

Why the count is exact

The loop considers every positional subset once. For a fixed mask, the inner loop adds exactly its selected values, so the equality test accepts precisely the required subsets. Incrementing once per accepted mask therefore counts all and only valid choices.

The sample has four valid selections: position 1 alone; positions 1 and 4; positions 1, 2, and 3; and all four positions. Adding position 4 changes the selected subset but not its sum, because its value is zero.

Do not stop evaluating a subset just because a partial sum exceeds the target: a later negative value can bring it back down. The representation does not justify a pruning rule that was invalid in Session 2 either.

Worked program and checks

Input contract. $0 \leq n \leq 20$, $|a[i]| \leq 10^9$, and $|\text{target}| \leq 2 \cdot 10^{10}$. Print the number of matching subsets.

Complete example | C++23

```
#include <iostream>
#include <vector>
using U = unsigned long long;
int main() {
    int n;
    long long target;
    std::cin >> n >> target;
    std::vector<long long> a(n);
    for (long long& x : a) std::cin >> x;
    long long count = 0;
    for (U mask = 0; mask < (1ULL << n); ++mask) {
        long long sum = 0;
        for (int i = 0; i < n; ++i)
            if ((mask & (1ULL << i)) != 0) sum += a[i];
        if (sum == target) ++count;
    }
    std::cout << count << '\n';
}
```

Sample input

4 3
3 -1 1 0

Sample output

4

Bounds and cost. Every total has absolute value at most $2 \cdot 10^{10}$, so `long long` is sufficient. The answer is at most 2^{20} . Time is $O((n+1)2^n)$ and input storage is $O(n)$; additional evaluation state is constant-sized.

Pause and predict: 19

For input values 0 0 and target zero, how many subsets match? What should happen for no input values and target zero?

Check your prediction. All four masks match for two zero values. With no values, the single empty subset matches, so the count is one. Treating zeros as irrelevant would lose distinct positional choices.

3.20 Partitioning and optimizing over subsets

A selected side determines the other side

Suppose we divide every input item between two groups. A mask chooses the first group; the complement chooses the second. If the total of all values is T and the selected sum is S , the other sum is $T - S$, so their absolute difference is

$$|S - (T - S)| = |T - 2S|.$$

There is no need for two nested subset loops. Once one group is selected, the other is fixed.

Feasibility and initialization

Selecting nothing gives difference T , so `best = total` starts from a valid candidate. Every later mask defines another complete partition. Replacing `best` only with smaller differences preserves the minimum among all processed candidates.

The implementation visits both a subset and its complement. They describe the same unordered partition and have the same difference, but repeating a candidate does not change a minimum. It would matter in a task asking how many *unordered* partitions exist. Do not divide an arbitrary count by two without examining symmetry and empty cases.

Worked program and checks

Input contract. $0 \leq n \leq 20$, and $0 \leq a[i] \leq 10^9$. Empty groups are allowed. Print the minimum absolute difference between the two group sums.

Complete example | C++23

```
#include <cstdlib>
#include <iostream>
#include <vector>
using U = unsigned long long;
int main() {
    int n;
    std::cin >> n;
    std::vector<long long> a(n);
    long long total = 0;
    for (long long& x : a) { std::cin >> x; total += x; }
    long long best = total;
    for (U mask = 0; mask < (1ULL << n); ++mask) {
        long long chosen = 0;
        for (int i = 0; i < n; ++i)
            if ((mask & (1ULL << i)) != 0) chosen += a[i];
        long long difference = std::llabs(total - 2 * chosen);
        if (difference < best) best = difference;
    }
    std::cout << best << '\n';
}
```

Sample input

4
8 3 5 2

Sample output

2

Cost and numeric safety. Time is $O((n+1)2^n)$ and stored input uses $O(n)$ space. Since $T \leq 2 \cdot 10^{10}$, both $2S$ and $T - 2S$ fit in `long long`. The absolute-value call is not applied to the most negative representable integer under this contract.

The same search can optimize a different score

For a small-item budget problem, compute both selected cost and selected benefit. Reject masks whose cost exceeds the budget, then maximize benefit among the remaining masks. Bitmasking describes the candidates; the problem determines the feasibility test and score.

Pause and predict: 20

Why would choosing two independent masks for the two groups be unnecessary and dangerous?

Check your prediction. It wastes work and can omit items or put an item into both groups. Choosing one group and defining the second as its complement automatically assigns every item exactly once.

3.21 Two universes: selected people and covered skills

Do not combine masks with different meanings

Imagine n people and f required skills. Each person's skill mask uses skill positions 0 through $f - 1$. A selection mask uses *person* positions 0 through $n - 1$. These are different universes even when both fit in $\mathbb{U}$.

For each selection, OR together the skill masks of its selected people. The resulting coverage mask has a skill bit exactly when at least one selected person supplies it. Count set bits in the *selection* to measure team size, not in the coverage.

Follow a small selection

In the sample, person 1 covers 011_2 , person 2 covers 100_2 , and person 3 covers 101_2 . Selecting people 1 and 2 uses selection mask 011_2 but gives coverage mask 111_2 . The equal-looking bit positions in those two masks refer to different things.

After scanning some selected people, **covered** is exactly the union of their skills. Extending that union preserves the invariant. Testing against **full** accepts exactly complete coverage because the input contains no out-of-universe skill bits. Taking the minimum selected count finds the optimum.

Worked program and checks

Input contract. Read n, f with $0 \leq n, f \leq 20$, then n skill masks in $[0, 2^f)$. Find the smallest number of people whose combined skills contain every required skill. Print -1 if no team exists. An empty skill universe requires zero people.

Complete example | C++23

```
#include <bit>
#include <iostream>
#include <vector>
using U = unsigned long long;
int main() {
    int n, features;
    std::cin >> n >> features;
    std::vector<U> have(n);
    for (U& mask : have) std::cin >> mask;
    U full = (1ULL << features) - 1ULL;
    int best = n + 1;
    for (U chosen = 0; chosen < (1ULL << n); ++chosen) {
        U covered = 0;
        for (int i = 0; i < n; ++i)
            if ((chosen & (1ULL << i)) != 0) covered |= have[i];
        int used = std::popcount(chosen);
        if (covered == full && used < best) best = used;
    }
    std::cout << (best == n + 1 ? -1 : best) << '\n';
}
```

Sample input

3 3
3 4 5

Sample output

2

Time is $O((n+1)2^n)$, with $O(n)$ input storage. This is appropriate because the *number of people* is small. A small number of skill bits alone would not justify enumerating 2^n people selections for large n .

Pause and predict: 21

A team covers five skills using two people. Which mask's popcount measures the team size?
Can `chosen` | `covered` have a useful set meaning here?

Check your prediction. The selection mask's popcount is two; the coverage mask's is five. ORing them mixes person IDs with skill IDs and does not represent the intended set of either.

3.22 Choosing recursion or mask enumeration

The same candidates can be visited differently

Session 2 generated subsets by making take-or-skip decisions recursively. A loop over masks generates the same positional choices without a recursive call stack. Neither representation removes the underlying exponential number of candidates.

For uncomplicated evaluation of every subset, the mask loop is compact and easy to test. For a problem where partial choices can be rejected early, backtracking may avoid entering many branches. For structured objects such as permutations, a subset mask alone does not encode their order.

Question	Mask enumeration	Backtracking
Which subset is selected?	Stored directly in an integer.	Stored along the current decision path.
Can invalid partial states stop early?	A plain full-mask loop still visits every mask.	Can prune a whole unfinished branch.
Where is temporary state kept?	Loop variables and optional tables.	Parameters, path state, and active calls.
Does order matter?	Not encoded by membership alone.	Can build ordered sequences explicitly.

Representation capacity is not a runtime guarantee

A 64-bit mask can represent a subset of 64 items. Enumerating all 2^{64} possibilities is a completely different requirement. Even 2^{30} is over one billion candidates before evaluation work.

Do not exchange one hidden exponential for another

A loop from zero to `1ULL << n` looks short in source code. Its cost is controlled by the value of the limit, not by the number of written lines. Always substitute the largest allowed n before choosing exhaustive enumeration.

Pause and predict: 22

Does changing a subset generator from recursion to a mask loop automatically reduce its complexity from exponential to linear?

Check your prediction. No. The candidates are still 2^n subsets. It may change constants and temporary state, but an algorithmic speedup needs less work, fewer candidates, or reusable summaries.

3.23 Fixed-size bitsets beyond a machine word

One object, a compile-time number of bits

`std::bitset<N>` from `<bitset>` stores N yes/no positions, where N is known at compile time. It is useful for a fixed feature universe larger than one integer and for readable binary displays. Its size does not grow at runtime like a vector's.^[3-10]

`std::bitset<16> flags;` starts with every bit zero. Constructing it from a 16-character binary string maps the *rightmost* character to bit 0. Thus `flags[0]` and `text[0]` refer to opposite ends of a conventional printed binary representation.^[3-15]

Operations worth recognizing

Call or expression	Job
<code>bits.test(i)</code>	Read a valid bit, with a checked position.
<code>bits.set(i) / bits.reset(i)</code>	Set or clear one position.
<code>bits.flip(i)</code>	Toggle one position.
<code>bits.set() / bits.reset()</code>	Set or clear every position.
<code>bits.count()</code>	Count set bits.
<code>bits.any()</code> , <code>bits.all()</code> , <code>bits.none()</code>	Test whether some, all, or no positions are set.
<code>bits.to_string()</code>	Produce N digits, highest position first.

Two same-sized bitsets can use `&`, `|`, and `^`; `~bits` flips only their N positions. Index access requires a valid index. Checked members such as `test(i)` throw for an invalid position; `operator[]` is not a substitute for bounds checking.^[3-16]

Bitset shifts create zeros in vacated positions. Unlike built-in integer shifts, shifting a bitset by N or more is defined and produces all zeros. This is a library operation with a different contract.^[3-16]

Worked program and checks

Input contract. Exactly 16 binary digits, followed by $0 \leq k < 16$. Flip bit k , then print the resulting 16 digits, the set-bit count, and **any**, **all**, and **none** as 0/1 results.

Complete example | C++23

```
#include <bitset>
#include <iostream>
#include <string>
int main() {
    std::string text;
    int k;
    std::cin >> text >> k;
    std::bitset<16> flags(text);
    flags.flip(k);
    std::cout << flags.to_string() << '\n';
    std::cout << flags.count() << '\n';
    std::cout << flags.any() << ' ' << flags.all()
              << ' ' << flags.none() << '\n';
}
```

Sample input

00000000000000101 1

Sample output

00000000000000111

3

1 0 0

Pause and predict: 23

For `std::bitset<8> b("10000000");`, which bit is set? Does `b[0]` read the first printed character?

Check your prediction. Bit 7 is set. Bit 0 is the rightmost printed character and is zero. Character indices and bit indices must not be confused.

3.24 Enumerating submasks, including zero

Choose only from items already allowed

A **submask** of **mask** is a subset of its set positions. Equivalently, $(\text{sub} \ \& \ \text{mask}) == \text{sub}$. Numeric comparison alone is not enough: 7 is smaller than 8, but 0111_2 is not a submask of 1000_2 .

Instead of visiting every mask in a large universe, visit only the allowed subsets in descending order. Start with the full allowed mask. For a nonzero submask, the update $\text{sub} = (\text{sub} - 1\text{ULL}) \ \& \ \text{mask}$ finds the next smaller allowed pattern.

Why subtracting and masking visits every choice

Imagine deleting the unavailable bit columns and reading only the p allowed positions. Those columns form an ordinary p -bit counter. Subtracting one lowers the next available 1 and fills lower available positions with ones; AND removes any unavailable positions introduced by the borrow. This is exactly a countdown in those compressed columns.

For $\text{mask} = 1010_2$, the available positions are 3 and 1. Their patterns count down as 11, 10, 01, 00; expanded back into four columns, these are 1010, 1000, 0010, 0000.

Worked program and checks

Input contract. $0 \leq \text{mask} < 2^{20}$ and its popcount is at most 12. Print every submask in decreasing numeric order, including zero.

Complete example | C++23

```
#include <iostream>
using U = unsigned long long;
int main() {
    U mask;
    std::cin >> mask;
    U sub = mask;
    while (true) {
        std::cout << sub << '\n';
        if (sub == 0) break;
        sub = (sub - 1ULL) & mask;
    }
}
```

Sample input	Sample output
10	10
	8
	2
	0

Break at zero before updating again

With unsigned arithmetic, `0ULL - 1ULL` is an all-ones value. ANDing it with `mask` returns the original mask, which would restart the loop. Process zero and then break. When the allowed mask is itself zero, print it once and stop.

With p allowed bits, the loop emits exactly 2^p submasks. Time is $O(2^p)$ and extra storage is constant, excluding output buffering.

Pause and predict: 24

List the submasks of 0101_2 in descending order. Why does `for (sub = mask; sub != 0; ...)` omit a legitimate case?

Check your prediction. They are 5, 4, 1, and 0. A loop conditioned on nonzero never processes the empty submask, which may be a valid answer or needed base case.

3.25 Count the real work and reset each test case

Word operations, bit scans, and subset scans

The fixed-word model treats an operation on one U as constant work. A loop over n positions still performs n such operations, and a loop over masks may perform 2^n iterations.

Work being performed	Bound under the stated model
One bit update or one-word set expression	$O(1)$.
Scan all n item positions	$O(n)$.
Clear every set bit	$O(p + 1)$ for p initially set bits.
Evaluate all subsets by scanning positions	$O((n + 1)2^n)$.
Enumerate submasks of one p -bit mask	$O(2^p)$.
Enumerate every mask and all its submasks	$O(3^n)$ state pairs.

For the final row, each universe bit has three possibilities: absent from the outer mask, present only in the outer mask, or present in both outer mask and submask. Multiplying the three choices over n bits gives 3^n . This counts iterations, before any additional per-pair work.

At $n = 20$, scanning all positions for every subset performs about 21 million membership tests. At $n = 30$, it exceeds 32 billion. These are operation counts, not guarantees about a particular time limit or machine.

Reset both the state and the answer

For multiple test cases, define masks, counts, and best answers inside a per-case helper or reset them at the beginning of each case. An old set bit can silently survive into a smaller next universe if you only OR in new values.

Fragment showing fresh per-case state:

```
void solve() {
    int n;
    std::cin >> n;
    U mask = 0;           // New state for this case.
    long long answer = 0;
    // Read the case, build its state, and print its answer.
}
```

The fragment is organizational, not a complete program. For cases with sizes $n_1, \dots, n_t$, total subset work is a sum such as $\sum_i O((n_i + 1)2^{n_i})$, not automatically $O(2^{\sum_i n_i})$.

Pause and predict: 25

A program runs all masks and all their submasks for n items. Is the total $2^n \cdot n$ merely because there are two written loops?

Check your prediction. No. The inner loop's length depends on the outer mask's popcount. Counting the three states of each bit gives 3^n mask/submask pairs.

3.26 Debug the type, the meaning, and the boundary

A short checklist before submitting

Type. Is the shifted 1 already wide and unsigned? Is a mathematical sum signed when negative values are allowed? Could a product or an answer count exceed the selected type?

Meaning. Does the bit index represent an item, a feature, a letter, or a sum? Are you comparing a mask with an index by mistake? Does the problem require frequencies or order that a membership mask has discarded?

Boundary. Can zero occur? Is the shift count strictly below the type width? Does a full-width mask avoid `1ULL << W`? Does subset enumeration include the empty selection and exclude the limit itself?

Algebra. Are you setting with OR rather than adding? Clearing with AND and a complement rather than toggling? Using XOR, not subtraction, to cancel XOR prefixes?

Cost. Does the largest allowed n justify exponential enumeration? Does a dynamic bitset-sized operation process more than one word? Have you counted output and every test case?

Counterexamples that expose common wrong code

Wrong idea	Small test to try
Set a bit by adding its weight.	Set bit 1 twice, starting from 0.
Check presence by comparing with 1.	Mask 8, bit index 3.
Use unrestricted NOT for a three-item set.	Mask 1; only positions 0 to 2 exist.
A zero XOR proves every value is paired.	Values 1, 2, 3.
Ignore empty subsets.	No values, target sum zero.
Prune a sum when it exceeds the target.	Values 8 and -5 , target 3.
Update a submask again after zero.	Allowed mask 10; the sequence cycles.

Test the implementation without losing the proof

Compare tiny cases with a slow solution using vectors, sets, or explicit take-or-skip choices. Test one high bit as well as many low ones. Enable warnings and undefined-behavior checks while developing. These checks can expose executed errors, but do not prove that the algorithm matches every input contract.

Pause and predict: 26

A subset loop starts at mask 1. It passes every positive-target sample. What minimal case should you add before trusting it?

Check your prediction. Use $n = 0$ and target zero. The correct count is one because the empty subset is valid. Also try two zero values with target zero, which require all four masks.

3.27 Optional extension: subset totals from smaller masks

Reuse one smaller answer instead of scanning all positions

In the ordinary subset-sum loop, each mask rescans up to n positions. When enough memory is available, a table can remember the sum for every mask. Let `sum[mask]` be the total of exactly the positions set in that mask.

For a nonzero mask, clear its lowest set bit to obtain a smaller mask. If the removed position is i , then

$$\text{sum}[\text{mask}] = \text{sum}[\text{mask without its lowest bit}] + a[i].$$

The smaller mask has a smaller numeric value, so increasing-mask order has already computed its entry. Start with `sum[0] = 0`.

Trace the recurrence

For values `2 5 1` and mask `1012`, the lowest set bit is position 0. The smaller mask is `1002`, whose sum is 1. Adding $a[0] = 2$ gives `sum[5] = 3`. Each selected value belongs either to the smaller mask or to the one removed position, never to both.

The count is initialized for mask zero before the nonzero loop. Calling `count_zero` in that loop is safe as a position computation because the loop starts at 1.

Worked program and checks

Input contract. The same as the signed subset-sum example: $0 \leq n \leq 20$, $|a[i]| \leq 10^9$, and $|\text{target}| \leq 2 \cdot 10^{10}$. Count all matching positional subsets, including the empty one.

Complete example | C++23

```
#include <bit>
#include <iostream>
#include <vector>
using U = unsigned long long;
int main() {
    int n;
    long long target;
    std::cin >> n >> target;
    std::vector<long long> a(n);
    for (long long& x : a) std::cin >> x;
    U limit = 1ULL << n;
    std::vector<long long> sum(limit, 0);
    long long count = (target == 0 ? 1 : 0);
    for (U mask = 1; mask < limit; ++mask) {
        U smaller = mask & (mask - 1ULL);
        int index = std::count_zero(mask);
        sum[mask] = sum[smaller] + a[index];
        if (sum[mask] == target) ++count;
    }
    std::cout << count << '\n';
}
```

Sample input

3 2
2 2 0

Sample output

4

Tradeoff. Under the fixed-word model, time becomes $O(n + 2^n)$ and table storage becomes $O(2^n)$. On the tested environment, 2^{20} eight-byte `long long` entries occupy 8 MiB for the element storage alone. At $n = 25$, that element storage becomes 256 MiB. This is not a free optimization.

Pause and predict: 27

Why can the sum table use increasing numeric mask order? Would taking an arbitrary random order preserve that guarantee?

Check your prediction. Removing a set bit strictly decreases the mask value, so its dependency is already computed. A random order may access a dependency before it has been filled.

3.28 Optional extension: count per-bit pair contributions

Change what the outer loop represents

Suppose we need the sum of $a[i] \oplus a[j]$ over all unordered pairs $i < j$. Checking every pair costs $O(n^2)$. Instead, determine how many pairs contribute each bit's weight.

At bit b , let c values have a 1 and $n - c$ have a 0. XOR sets this bit exactly for a pair with one value from each group. There are $c(n - c)$ such unordered pairs, each contributing 2^b . Therefore

$$\sum_{i < j} (a[i] \oplus a[j]) = \sum_b c_b(n - c_b) 2^b.$$

There is no factor of two: selecting one item from the 1 group and one from the 0 group already identifies one unordered pair.

One pair can contribute in several positions

For 1 2 3, the pairwise XOR values are 3, 2, and 1, summing to 6. At bit 0, two values have a 1 and one has a 0, contributing $2 \cdot 1 \cdot 1 = 2$. At bit 1, the same split contributes $2 \cdot 1 \cdot 2 = 4$.

This is a change in counting order: add each pair's set-bit weights, or add each bit's contributions from all pairs. Both count exactly the same contributions.

Worked program and checks

Input contract. $0 \leq n \leq 100000$ and $0 \leq a[i] < 2^{30}$. Print the sum of pairwise XOR values, counting each $i < j$ pair once.

Complete example | C++23

```
#include <iostream>
#include <vector>
using U = unsigned long long;
int main() {
    int n;
    std::cin >> n;
    std::vector<U> a(n);
    for (U& x : a) std::cin >> x;
    long long answer = 0;
    for (int bit = 0; bit < 30; ++bit) {
        long long ones = 0;
        for (U x : a)
            if ((x & (1ULL << bit)) != 0) ++ones;
        long long zeros = n - ones;
        answer += ones * zeros * (1LL << bit);
    }
    std::cout << answer << '\n';
}
```

Sample input

3
1 2 3

Sample output

6

Cost and bounds. The code makes 30 scans, so time is $O(30n + 30)$ and input storage is $O(n)$. Since $c(n - c) \leq n^2/4$, the result is at most $(100000^2/4)(2^{30} - 1) < 2.685 \cdot 10^{18}$, which fits a signed 64-bit `long long`. The variables are widened *before* multiplication.

To count total differing *positions* rather than total XOR *values*, omit the factor 2^b . Practice 19 asks for that variation.

3.29 Optional extension: reachable sums with a bitset

A bit can represent a sum instead of an item

This extension changes the meaning of a position. Let `reachable[s]` be true exactly when some subset of the processed input has total s . It is a set of achievable sums, not one chosen subset and not a count of subsets.

Initially only sum zero is possible. When a new nonnegative value v arrives, the old sums remain possible by skipping it. Taking it adds v to every old sum. Shifting the old bitset left by v represents those new totals, so combine both choices with

```
reachable |= reachable << value;
```

The shifted expression is a separate result computed from the state before the OR update. The current item is not repeatedly reused within this one update.

Trace sets of sums rather than thousands of bits

For values 2, 5, and 8, the reachable sets evolve as

$$\{0\} \longrightarrow \{0, 2\} \longrightarrow \{0, 2, 5, 7\} \longrightarrow \{0, 2, 5, 7, 8, 10, 13, 15\}.$$

After each update, a subset either skipped the new item, producing an old sum, or took it, producing an old sum plus its value. These are exactly the two sets combined by OR. This inductive argument proves the state meaning.

What the summary keeps and loses

Equal totals merge into the same 1 bit. This can answer existence but cannot count how many subsets reach the target or recover a witness without more information. A zero value creates no new reachable total, though it would create additional positional subsets in a counting task.

This is a small introduction to reusing boolean states. Negative values are not supported by this left-shift model. With nonnegative values, totals beyond an allowed upper bound cannot later decrease back into range, so truncating them is safe for bounded queries.

Worked program and checks

Input contract. $0 \leq n \leq 100$, every value is in $[0, 100]$, and the queried target is in $[0, 10000]$. Print YES if some subset reaches it, otherwise NO. The sum of all inputs is at most 10000.

Complete example | C++23

```
#include <bitset>
#include <iostream>
int main() {
    int n, target;
    std::cin >> n >> target;
    std::bitset<10001> reachable;
    reachable.set(0);
    for (int i = 0; i < n; ++i) {
        int value;
        std::cin >> value;
        reachable |= reachable << value;
    }
    std::cout << (reachable.test(target) ? "YES" : "NO") << '\n';
}
```

Sample input

3 7
2 5 8

Sample output

YES

Cost model. With $S + 1$ tracked totals, a scalar-bit implementation can be analysed as $O((n + 1)(S + 1))$ work, including initialization, and $O(S + 1)$ bits of state. A typical word-packed bitset processes about $\lceil (S + 1)/w \rceil$ words per shift or OR, giving the corresponding word-operation estimate. This is not a standard-library promise that every operation is one CPU instruction. C++23 does not specify that packed complexity for `bitset`.^[3-16]

Pause and predict: 28

With the single input value 2, should sum 4 become reachable? Can the same bitset tell you whether sum 2 is reachable in one way or several ways?

Check your prediction. Sum 4 must stay unreachable because the one item may be used only once. The bitset records only yes/no reachability, so it cannot distinguish one construction from several constructions.

3.30 Optional extension: backtracking with copied masks

Compress the conflict state, not the search itself

Return to the N-queens problem from Session 2. Place one queen per row, never reusing a column or diagonal. A mask can store occupied columns. Two more masks describe columns attacked by diagonals in the *current* row.

For this implementation, the parameter named **left** moves its attacks towards larger column indices on the next row; **right** moves them towards smaller indices. These names describe shift directions, not the orientation of a printed board. Before choosing a square,

```
U available = full & ~(columns | left | right);
```

keeps only valid, unattacked columns. Isolate one available bit, explore it, and remove it from the loop's remaining choices.

Worked program and checks

Input contract. $0 \leq n \leq 12$. Print the number of arrangements of n nonattacking queens on an n by n board. The empty board has one arrangement.

Complete example | C++23

```
#include <iostream>
using U = unsigned long long;
long long countQueens(U full, U columns, U left, U right) {
    if (columns == full) return 1;
    U available = full & ~(columns | left | right);
    long long answer = 0;
    while (available != 0) {
        U bit = available & (0ULL - available);
        available &= available - 1ULL;
        answer += countQueens(full, columns | bit,
                               ((left | bit) << 1) & full,
                               (right | bit) >> 1);
    }
    return answer;
}
int main() {
    int n;
    std::cin >> n;
    U full = (1ULL << n) - 1ULL;
    std::cout << countQueens(full, 0, 0, 0) << '\n';
}
```

Sample input

Sample output

4

2

Why the diagonal shifts are needed

An attack moving along a diagonal changes its column by one between consecutive rows. After adding the chosen queen's column bit, one direction shifts left and the other shifts right. AND with `full` discards positions outside the board.

The `columns` mask gains exactly one new bit per recursive level. When it equals `full`, n queens have been placed, one in every row and in distinct columns. Every recursive choice is unattacked, and every possible valid next column is considered once, so the result counts exactly the legal arrangements.

No shared mask needs an explicit undo

The parameters are passed by value. A child receives new masks; when it returns, the caller still has its own masks. Only the caller's local `available` changes as it iterates. This is different from mutating a shared vector or board by reference.

Under the fixed-word model, the explored partial permutations give an $O(n!)$ upper bound for $n \geq 1$; the diagonal restrictions often prune many branches. Maximum active-call depth is $n + 1$, giving $O(n + 1)$ stack space. Bit operations reduce state and conflict-check costs, not the combinatorial nature of the search.

3.31 Optional extension: powers, rotations, and compiler builtins

Useful power helpers need valid results

For unsigned x , `std::bit_floor(x)` gives the largest power of two no greater than x , or zero for zero. `std::bit_ceil(x)` gives the smallest power of two at least x , and gives 1 for zero. Its result must be representable in the argument's type.^[3-8]

On a W -bit unsigned type, the largest representable power of two is 2^{W-1} . Larger inputs have no representable ceiling power. Guard the call rather than waiting for wraparound.

Fragment inside a function, with `<bit>`, `<limits>`, and `<iostream>`:

```
constexpr int W = std::numeric_limits<U>::digits;
U highestPower = 1ULL << (W - 1);
if (x <= highestPower) {
    std::cout << std::bit_ceil(x) << '\n';
} else {
    std::cout << "NONE\n";
}
```

A rotation wraps bits instead of discarding them

`std::rotl(x,k)` and `std::rotr(x,k)` rotate an unsigned value across the *whole type width*. Counts are interpreted modulo that width; a negative count rotates in the other direction.^[3-17]

A handwritten expression such as `(x << k) | (x >> (W - k))` has an invalid built-in shift when $k = 0$. The standard rotation helpers handle those boundary cases. Rotating a three-feature logical universe is a different task from rotating an entire 64-bit type: do not confuse the widths.

Recognize older GNU spellings without depending on them

You may encounter `__builtin_popcount` for unsigned int and `__builtin_popcountll` for unsigned long long. Using the narrower form on a wide value can lose its high bits during conversion. GNU trailing/leading-zero builtins such as `__builtin_ctzll` and `__builtin_clzll` do not define a result for zero. Standard `std::countr_zero` and `std::countl_zero` do define the zero result as the width.^{[3-12][3-7]}

All complete programs in this chapter use standard facilities. The GNU names are recognition notes, not additional compiler requirements.

Bit tricks should not obscure ordinary operations

Use `std::swap` rather than an XOR-swap trick. Use ordinary multiplication or division when that expresses the mathematical operation clearly. Use a bit operation when its bit-level meaning helps solve the problem, not merely to make the code look advanced.

3.32 Practice problems: state the meaning of each bit

Problems 1 to 7 build confidence with single-word operations. Problems 8 to 12 use set masks and XOR. Problems 13 to 17 enumerate and evaluate selections. Problems 18 to 20 are optional extensions.

For each problem, write the input contract, the meaning of a bit, and the reason each operation is correct. Explain the complexity using the largest stated bounds. The core tasks do not require the optional extensions.

The 64-bit training environment represents U inputs from 0 through $2^{64} - 1$. When a task uses W , query the type width with `std::numeric_limits<U>::digits`. Every complete solution appears later; use its hint first.

1. Parity from the last bit

Classify each nonnegative integer as even or odd using its least significant bit.

Input. t , followed by t values representable in U . $0 \leq t \leq 200000$.

Output. One line per value: EVEN or ODD. For $t = 0$, print nothing.

Sample input	Sample output
4	EVEN
0 5 8 13	ODD
	EVEN
	ODD

2. Inspect requested positions

Given one integer, answer queries asking for the value of a particular bit. Print the digit 0 or 1, not the bit's weight.

Input. A U value x and q , followed by q indices k . $0 \leq q \leq 200000$ and $0 \leq k < W$, where W is the width of U .

Output. The requested binary digit on each line.

Sample input	Sample output
13 4	1
0	0
1	1
2	1
3	

3. Assign one bit, do not toggle it

Change bit k of x to the supplied digit v , preserving every other bit. Applying the same assignment twice must have the same effect as applying it once.

Input. A U value x , an index $0 \leq k < W$, and $v \in \{0, 1\}$.

Output. The resulting integer in decimal.

Sample input	Sample output
13 2 0	9

4. Flip exactly the lowest k positions

Toggle the lowest k bits of x . Preserve every position at index k or greater. A request with $k = 0$ changes nothing; $k = W$ flips the whole type.

Input. A U value x and $0 \leq k \leq W$. The verification environment has $W = 64$.

Output. The resulting unsigned value in decimal.

Sample input	Sample output
42 4	37

5. Minimum bit flips

Find the minimum number of individual bit flips needed to change the complete unsigned bit pattern of a into that of b .

Input. Two values a, b representable in $\mathbb{U}$. Leading zero positions belong to the same fixed-width representation.

Output. The minimum number of flips.

Sample input	Sample output
12 10	2

6. Recognize powers of two

For each input, determine whether it is 2^k for an integer $k \geq 0$. Derive a test using the lowest-set-bit clearing identity.

Input. t , followed by t $\mathbb{U}$ values. $0 \leq t \leq 200000$.

Output. One line per value: YES or NO.

Sample input	Sample output
5	NO
0 1 2 3 16	YES
	YES
	NO
	YES

7. Locate the first set bit

For each value, report the index of its least significant set bit. This task asks for the index, not the isolated bit's decimal value.

Input. q , followed by q $\mathbb{U}$ values. $0 \leq q \leq 200000$.

Output. One zero-based index per value, or -1 if the value is zero.

Sample input	Sample output
4	-1
0 1 40 64	0
	3
	6

8. Compatible feature sets

Each record stores a feature-presence mask. A pair of records is compatible exactly when their masks have no common set bit. Answer the requested pairs without rebuilding sets.

Input. n, q , then n masks in $[0, 2^{60})$, then q pairs of one-based record labels. $0 \leq n, q \leq 200000$; all labels are valid. If $n = 0$, then $q = 0$. Labels in a query may be equal.

Output. YES for each compatible pair, otherwise NO.

Sample input	Sample output
3 3	YES
3 4 5	NO
1 2	NO
1 3	
2 3	

9. Find the missing identifier

The identifiers are every integer from 1 through n . Exactly one is omitted from a list, and every other identifier occurs once. Find the missing identifier using XOR and constant working memory.

Input. $1 \leq n \leq 200000$, followed by the $n - 1$ listed identifiers in any order.

Output. The missing identifier.

Sample input	Sample output
5	3
4 1 5 2	

10. Odd frequency across several test cases

In each test case, exactly one distinct value has odd frequency and all others have even frequency. Print the odd-frequency value for every case.

Input. $0 \leq t \leq 10000$. Each case contains n followed by n U values, with $1 \leq n \leq 200000$. The total of n over all cases is at most 200000.

Output. One odd-frequency value per case.

Sample input	Sample output
2	2
5	0
4 2 4 2 2	
1	
0	

11. Does an interval match the target XOR?

For each inclusive range and target, decide whether the XOR of that subarray equals the target. The array does not change between queries.

Input. n, q , then n U values, then queries $l \ r \ target$. $0 \leq n, q \leq 200000$, $1 \leq l \leq r \leq n$, and targets fit U . If $n = 0$, $q = 0$.

Output. One YES or NO per query.

Sample input	Sample output
3 3	YES
5 7 2	YES
1 2 2	NO
1 3 0	
2 3 4	

12. Reconstruct adjacent-XOR data

An unknown length- n sequence has first value $a[0]$. You are also given $e[i] = a[i-1] \oplus a[i]$ for $i = 1, \dots, n-1$. Reconstruct the original sequence.

Input. $1 \leq n \leq 200000$ and the first U value, followed by $n-1$ encoded U values.

Output. The n original values on one line, separated by spaces.

Sample input	Sample output
4 5	5 7 2 2
2 5 0	

13. Exactly k selected positions

Count positional subsets containing exactly k selected elements whose sum equals the target. Equal values at different positions remain distinct choices.

Input. n, k, target , then n integers. $0 \leq k \leq n \leq 20$, $|a[i]| \leq 10^9$, and $|\text{target}| \leq 2 \cdot 10^{10}$.

Output. The number of valid subsets.

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

14. Best benefit within a budget

Choose any subset of items, each at most once. Total cost must not exceed the budget. Maximize total benefit. Selecting nothing is permitted.

Input. n, budget , followed by n pairs **cost value**. $0 \leq n \leq 20$, $0 \leq \text{budget} \leq 2 \cdot 10^{10}$, $0 \leq \text{cost}[i] \leq 10^9$, and $|\text{value}[i]| \leq 10^9$.

Output. The maximum achievable total benefit.

Sample input	Sample output
3 5	11
3 7	
2 4	
0 -5	

15. Recover a minimum team

Find a smallest team covering every skill and print its member labels. If several minimum teams exist, choose the one with the smallest numeric selection mask, where bit i represents person $i + 1$.

Input. n, f , then n skill masks in $[0, 2^f)$. $0 \leq n, f \leq 20$. All f skills are required.

Output. If impossible, print -1 only. Otherwise print the minimum size on the first line and the selected one-based person labels in increasing order on the second. For a size-zero team, print `EMPTY` on the second line.

Sample input	Sample output
3 3	2
3 4 5	1 2

16. Count selections without forbidden pairs

There are n items and a list of incompatible pairs. A valid selection may not contain both items of any listed pair. Count every valid positional subset, including the empty subset.

Input. n, m , then m distinct unordered pairs of different one-based item labels. $0 \leq n \leq 20$ and $0 \leq m \leq \min(60, n(n-1)/2)$.

Output. The number of valid selections.

Sample input	Sample output
3 2	5
1 2	
2 3	

17. Count subsets with a requested XOR

Count positional subsets whose selected values XOR to the target. The XOR of the empty subset is zero.

Input. n , target, then n integers, with $0 \leq n \leq 20$ and $0 \leq a[i]$, target $< 2^{30}$.

Output. The number of matching subsets.

Sample input	Sample output
3 0 1 1 0	4

18. Allowed, required, and forbidden bits

List submasks of an allowed mask M that contain every bit in R and no bit in F . Bits required outside M , or both required and forbidden, can make the answer impossible.

Input. Three masks M, R, F in $[0, 2^{20})$, with at most 12 set bits in M .

Output. All valid submasks in decreasing decimal order, one per line. Print NONE if there are none. A valid empty submask is printed as the number 0.

Sample input	Sample output
13 1 4	9 1

19. Total differing positions over all pairs

For every unordered pair $i < j$, count the positions where $a[i]$ and $a[j]$ differ. Sum those counts over all pairs. This is not the sum of the numeric XOR values.

Input. n , then n integers with $0 \leq n \leq 200000$ and $0 \leq a[i] < 2^{60}$.

Output. The total number of differing-bit contributions.

Sample input	Sample output
3 1 2 3	4

20. Many subset-sum existence queries

Given one fixed list of nonnegative values, answer many questions asking whether some subset sums to a requested target. Each item may be used at most once per subset. The empty subset is allowed.

Input. n, q , then n values, then q targets. $0 \leq n \leq 100$, $0 \leq q \leq 200000$, values lie in $[0, 100]$, and targets in $[0, 10000]$.

Output. One YES or NO per target.

Sample input	Sample output
3 4	YES
2 5 8	YES
0	NO
7	YES
6	
15	

3.33 Trace, correctness, and complexity drills

Write your answers before reading the hints. A correct answer should include the reason, not only a final number. These fragments are study questions, not independent complete programs. Assume valid stated types and bounds unless a question deliberately asks you to find a violation.

Drill 1: Read positions before calculating

Write 45 and 26 as eight-bit patterns. List the set-bit positions of 45, then calculate their AND, OR, and XOR by aligning columns. For 45, distinguish the highest set-bit index from its weight and from the number of set bits.

Now flip only the lowest six positions of 45. Explain why an unrestricted complement of its entire type is a different result.

Drill 2: A wide destination does not repair a narrow operation

Assume `int` has 32 bits and `U` has 64 bits. For each fragment, say whether it is valid C++ for the intended operation, and repair it where needed.

```
U a = 1 << 40;
U b = 1ULL << 63;
U c = 1ULL << 64;
bool absent = mask & (1ULL << k) == 0;
```

In the final line, assume $0 \leq k < 64$. Do not run the invalid shifts to guess their meaning.

Drill 3: Trace assignment and toggling separately

Start with `mask = 3` in a five-position universe. Trace the full mask after each update and give the printed check result.

```
mask |= 1ULL << 1;
mask |= 1ULL << 1;
mask &= ~(1ULL << 0);
mask ^= 1ULL << 4;
mask ^= 1ULL << 4;
std::cout << ((mask & (1ULL << 1)) != 0);
```

Would replacing each of the first two updates with `mask += 1ULL << 1` preserve the intended operation?

Drill 4: What does the input promise buy you?

Compute the overall XOR for each input and decide whether it can be interpreted as the single odd-occurrence value:

A	8	3	8	5	3
B	1	2	3		
C	6	6	9	12	

Refute the statement “a zero XOR proves that every value occurs an even number of times.” Explain which additional method fits input C.

Drill 5: Prefix cancellation and stale summaries

For the array 5 1 5 7, build a padded prefix-XOR table. Use it to answer [2,4] and [3,3] with one-based inclusive positions.

Next, change the second input value from 1 to 3. Which prefix entries are now stale? What is the correct new XOR for [2,4]? Explain why leaving the old table unchanged is not a valid dynamic-query algorithm.

Drill 6: Any requirement versus every requirement

A six-feature mask `have` is 101001_2 . Compare these requirements:

$$R_1 = 001001_2, \quad R_2 = 001100_2.$$

For each, determine whether every required feature is present. Explain why the test `(have & required) != 0` gives a wrong interpretation for one of them.

Find the complement of `have` within the six-feature universe. Does numeric comparison `have >= required` characterize set containment?

Drill 7: Duplicate values still have separate positions

For the array `2 2 0`, list every mask whose selected sum is 2. Give the selected positions using one-based labels. Then count only those valid selections with exactly two positions.

Would storing the valid *sums* in a `set` preserve the requested subset count? Explain the answer for $n = 0$ and target zero too.

Drill 8: Two wrong loop endpoints can hide one another

A student writes:

```
for (U mask = 1; mask <= (1ULL << n); ++mask) {
    for (int i = 0; i < n; ++i) {
        if ((mask & (1ULL << i)) != 0) {
            // Use input position i.
        }
    }
}
```

With $n = 3$, identify the missing valid mask and the included invalid mask. Explain why a test that checks only a total sum can accidentally fail to expose the representation error. Repair the loop and state the shift-width precondition.

Drill 9: Submasks and the zero transition

List all submasks of 1101_2 in descending order. Starting at the allowed mask, trace `(sub - 1ULL) & mask` until zero.

What happens if you apply the update once more at zero? How many outputs should appear, and which quantity controls that number: the decimal mask value, the bit width, or the popcount?

Drill 10: Three loops, three different growth rates

Assume $0 \leq n \leq 20$ and `U limit = 1ULL << n`. Compare:

- A.** Scan the n bit positions of one mask once.
- B.** Visit every mask below `limit`, scanning all n positions inside each iteration.
- C.** Visit every mask below `limit`, enumerating all its submasks, including zero, inside each iteration.

Give each complexity and justify C by classifying one universe bit's possible roles. Explain why storing a state in one word does not make B or C constant-time overall.

Drill 11: The same bit storage can mean different things

For `std::bitset<8> b("10010000");`, state the set-bit positions and the value of `b[0]`. What does `b.flip(0)` print afterwards, and what is its count?

In a reachable-sum bitset, only bit 0 is initially set. After processing the single value 2 once, which totals must be set? Why must applying the update twice not be used to represent that one item? What does bit 0 mean in this new representation?

Drill 12: Count pair weights or count differing positions?

For 1 2 3, calculate both the sum of pairwise XOR *values* and the sum of pairwise counts of differing *positions*, using unordered pairs $i < j$.

Derive the per-bit contribution for each task. If `ones` and `zeros` are both 32-bit `int`, is the following safe at 50000 ones and 50000 zeros?

```
long long pairs = ones * zeros;
```

Repair the expression so its product is evaluated in the wider type.

3.34 Hints before solutions

Use only the hint for the problem you are attempting. Translate it into a small drawing or invariant, then close this page and write your own solution. Full code is in the next section.

1. Parity from the last bit

Which binary place value is not divisible by two? *After trying: read the explained solution.*

2. Inspect requested positions

First move the requested position to bit 0. Then discard the other positions. *After trying: read the explained solution.*

3. Assign one bit, do not toggle it

The requested result is a known digit, not the opposite of the old digit. *After trying: read the explained solution.*

4. Flip exactly the lowest k positions

This is a toggle operation with a group mask. The all-ones full-width mask cannot be built by shifting one by the type width. *After trying: read the explained solution.*

5. Minimum bit flips

Which operator marks positions where two bit patterns disagree? *After trying: read the explained solution.*

6. Recognize powers of two

Clear one set bit, but first distinguish zero from a value with exactly one set bit. *After trying: read the explained solution.*

7. Locate the first set bit

The zero-counting library helper has a defined answer for zero, but that answer is not a valid set-bit index. *After trying: read the explained solution.*

8. Compatible feature sets

First distinguish a record's array index from the feature positions inside its stored mask. *After trying: read the explained solution.*

9. Find the missing identifier

Create one XOR of what should exist and cancel the items that do exist. *After trying: read the explained solution.*

10. Odd frequency across several test cases

The one-case algorithm is unchanged. The important additional contract is where the accumulator is initialized. *After trying: read the explained solution.*

11. Does an interval match the target XOR?

The prefix definition should say what the first i elements contribute, with an empty-prefix identity. *After trying: read the explained solution.*

12. Reconstruct adjacent-XOR data

XORing the same value twice cancels it. Isolate the unknown in the encoded equation. *After trying: read the explained solution.*

13. Exactly k selected positions

The mask already contains its selection size as a bit count. The numeric sum belongs in a separate signed variable. *After trying: read the explained solution.*

14. Best benefit within a budget

A mask describes the candidate. Compute feasibility and quality as two different quantities. *After trying: read the explained solution.*

15. Recover a minimum team

Keep the best selection itself, not only its size. What comparison preserves the first minimum-size mask encountered? *After trying: read the explained solution.*

16. Count selections without forbidden pairs

Checking for any overlap is too strong: selecting just one member of a forbidden pair is allowed. *After trying: read the explained solution.*

17. Count subsets with a requested XOR

The selection mechanics match subset sums; change the combining operation and its identity, not the definition of a subset. *After trying: read the explained solution.*

18. Allowed, required, and forbidden bits

Use separate tests for *all required* and *no forbidden*. An empty valid result is different from no valid results. *After trying: read the explained solution.*

19. Total differing positions over all pairs

Count by bit position rather than by pair. Does this task ask for bit weights, or one unit per differing position? *After trying: read the explained solution.*

20. Many subset-sum existence queries

The queries concern the same fixed input, so do not rebuild the reachability summary for every target. *After trying: read the explained solution.*

Hints for the twelve drills

1. Label weights 128, 64, ..., 1. Restrict the complement with the six-position full mask.
2. Inspect the type of the shifted left operand before looking at the assignment destination. Parenthesize the bitwise result before comparing.
3. Setting an already-present bit leaves it present. Toggling twice undoes the first toggle.
4. Count odd-frequency groups before interpreting the accumulator. Different values can combine to a value absent from the input.
5. Entry i includes every input position through i , so an edit affects a suffix of the prefix table.
6. Full containment means that every required bit survives the intersection, not merely one of them.
7. Masks distinguish the two copies of 2. Adding the zero-valued position creates a different selection with the same sum.
8. The out-of-range mask has its only bit at position n , which the inner loop never inspects. A sum-only observation may therefore look like an empty choice.
9. At zero, unsigned subtraction creates an all-ones value. There are two choices for every allowed set bit.
10. In C, a position can be absent from both, present only in the outer mask, or present in both.
11. The rightmost printed character is bit 0. Reachable-sum positions identify totals rather than input items.
12. A differing position contributes one unit to a Hamming count, but contributes 2^b to a numeric XOR value. Widen before multiplying.

3.35 Explained practice solutions

These are separate programs, each with its own `main`. The problem statements and sample cases are in the practice section. Try reconstructing the representation and the proof before reading the code.

A subset mask, a feature mask, and a reachable-sum bitset may have identical storage syntax but different meanings. Read the explanation of the state in each solution rather than copying an operation into a new problem without its contract.

Solution 1: Parity from the last bit

Idea. Test bit 0 with `x & 1ULL`. All higher binary weights are multiples of two, so only the final bit determines parity.

Complete solution | C++23

```
#include <iostream>
using U = unsigned long long;
int main() {
    int t;
    std::cin >> t;
    while (t-- > 0) {
        U x;
        std::cin >> x;
        std::cout << ((x & 1ULL) == 0 ? "EVEN" : "ODD") << '\n';
    }
}
```

Why it works. Write $x = 2q + b_0$ with $b_0 \in \{0, 1\}$. AND with one returns exactly b_0 . It is zero precisely when x is even.

Complexity. $O(t)$ time and $O(1)$ auxiliary space.

Boundary checks. Zero is even. Also test one, a high power of two, and the largest representable unsigned value. The input contract excludes negative decimal values.

Solution 2: Inspect requested positions

Idea. Shift position k down to position 0, then keep only that lowest bit using $\& 1ULL$.

Complete solution | C++23

```
#include <iostream>
using U = unsigned long long;
int main() {
    U x;
    int q;
    std::cin >> x >> q;
    while (q-- > 0) {
        int k;
        std::cin >> k;
        std::cout << ((x >> k) & 1ULL) << '\n';
    }
}
```

Why it works. Right shift removes positions below k . The original bit k becomes the least significant bit, which AND with one isolates as the numeric digit zero or one.

Complexity. $O(q)$ time and $O(1)$ auxiliary space in the fixed-word model.

Boundary checks. Test zero, $k = 0$, and $k = W - 1$. A bit above the highest set bit but below W is a valid query whose answer is zero.

Solution 3: Assign one bit, do not toggle it

Idea. Build the one-bit mask. OR it in when $v = 1$; AND with its complement when $v = 0$.

Complete solution | C++23

```
#include <iostream>
using U = unsigned long long;
int main() {
    U x;
    int k, value;
    std::cin >> x >> k >> value;
    U bit = 1ULL << k;
    if (value == 1) x |= bit;
    else x &= ~bit;
    std::cout << x << '\n';
}
```

Why it works. At the requested position, OR with one forces one and AND with zero forces zero. At all other positions, these operations preserve the old bit. No carry changes an adjacent position.

Complexity. $O(1)$ time and auxiliary space.

Boundary checks. Assign one to an already-set bit and zero to an already-clear bit. Check both the lowest and highest permitted positions.

Solution 4: Flip exactly the lowest k positions

Idea. Create a mask with exactly the lowest k positions set, handling the full-width case separately. XOR it with x .

Complete solution | C++23

```
#include <iostream>
#include <limits>
using U = unsigned long long;
int main() {
    U x;
    int k;
    std::cin >> x >> k;
    constexpr int width = std::numeric_limits<U>::digits;
    U low = (k == width ? ~0ULL : (1ULL << k) - 1ULL);
    std::cout << (x ^ low) << '\n';
}
```

Why it works. XOR flips a position exactly when the corresponding mask bit is one. The constructed mask therefore flips the requested positions and preserves all others. The conditional branch avoids ever evaluating a shift by W .

Complexity. $O(1)$ time and auxiliary space.

Boundary checks. Test $k = 0$, $k = 1$, and $k = W$, plus inputs with high bits already set. Restricting the result with AND would wrongly delete high bits that must be preserved.

Solution 5: Minimum bit flips

Idea. XOR identifies exactly the differing positions. Count its set bits with `std::popcount`.

Complete solution | C++23

```
#include <bit>
#include <iostream>
using U = unsigned long long;
int main() {
    U a, b;
    std::cin >> a >> b;
    std::cout << std::popcount(a ^ b) << '\n';
}
```

Why it works. Every differing position must be flipped at least once, and a single flip at each differing position is sufficient. Equal positions need no changes. The count of XOR ones is therefore both a lower bound and an achievable answer.

Complexity. Bounded fixed-word work and $O(1)$ space. Equivalently, a manual implementation can inspect at most W positions.

Boundary checks. Equal values require zero flips. Compare zero with all ones to obtain W flips. The numeric difference $|a - b|$ is not the answer.

Solution 6: Recognize powers of two

Idea. A positive power of two has exactly one set bit. Check that x is nonzero and clearing its lowest set bit leaves zero.

Complete solution | C++23

```
#include <iostream>
using U = unsigned long long;
int main() {
    int t;
    std::cin >> t;
    while (t-- > 0) {
        U x;
        std::cin >> x;
        bool power = x != 0 && (x & (x - 1ULL)) == 0;
        std::cout << (power ? "YES" : "NO") << '\n';
    }
}
```

Why it works. Every positive integer has at least one set bit. If the clearing operation leaves zero, the removed bit was the only one, so the value equals its power-of-two weight. Conversely, clearing the only set bit of a power of two leaves zero.

Complexity. $O(t)$ time and $O(1)$ auxiliary space.

Boundary checks. Zero must produce NO, while one produces YES. Test one below and one above a power of two, and the highest representable power.

Solution 7: Locate the first set bit

Idea. For nonzero input, the number of trailing zeros is exactly the index of the first one. Handle zero with the requested sentinel.

Complete solution | C++23

```
#include <bit>
#include <iostream>
using U = unsigned long long;
int main() {
    int q;
    std::cin >> q;
    while (q-- > 0) {
        U x;
        std::cin >> x;
        std::cout << (x == 0 ? -1 : std::countr_zero(x)) << '\n';
    }
}
```

Why it works. If the first one is at index k , all positions $0, \dots, k - 1$ are zero, so the trailing-zero count is k . Zero has no such position and must not be interpreted as having a bit at index W .

Complexity. $O(q)$ fixed-word time and $O(1)$ auxiliary space.

Boundary checks. Test zero, one, 40, and the highest single-bit mask. For 40, print 3 rather than 8.

Solution 8: Compatible feature sets

Idea. Intersect the two masks with AND. An empty intersection means no shared features.

Complete solution | C++23

```
#include <iostream>
#include <vector>
using U = unsigned long long;
int main() {
    int n, q;
    std::cin >> n >> q;
    std::vector<U> mask(n);
    for (U& x : mask) std::cin >> x;
    while (q-- > 0) {
        int a, b;
        std::cin >> a >> b;
        bool disjoint = (mask[a - 1] & mask[b - 1]) == 0;
        std::cout << (disjoint ? "YES" : "NO") << '\n';
    }
}
```

Why it works. A position survives AND precisely when both records contain that feature. The result is zero exactly when no feature is shared, which is the stated definition of compatibility.

Complexity. $O(n + q)$ time and $O(n)$ stored masks, with constant additional query state.

Boundary checks. Two empty masks are compatible. Comparing a nonempty mask with itself is not compatible. Convert record labels to zero-based vector indices; those labels are not feature-bit indices.

Solution 9: Find the missing identifier

Idea. XOR every expected identifier, then XOR every received identifier into the same accumulator.

Complete solution | C++23

```
#include <iostream>
using U = unsigned long long;
int main() {
    int n;
    std::cin >> n;
    U missing = 0;
    for (int i = 1; i <= n; ++i) missing ^= U(i);
    for (int i = 0; i < n - 1; ++i) {
        U x;
        std::cin >> x;
        missing ^= x;
    }
    std::cout << missing << '\n';
}
```

Why it works. Each present identifier appears twice in the combined expression and cancels. The missing identifier appears only in the expected list and remains. Order does not matter because XOR is associative and commutative.

Complexity. $O(n)$ time and $O(1)$ auxiliary space. No array is needed.

Boundary checks. At $n = 1$, no identifiers follow and the answer is 1. Also test missing first, missing last, and shuffled input. The uniqueness promise is essential.

Solution 10: Odd frequency across several test cases

Idea. Use a new XOR accumulator initialized to zero inside `solve()` for every test case.

Complete solution | C++23

```
#include <iostream>
using U = unsigned long long;
void solve() {
    int n;
    std::cin >> n;
    U answer = 0;
    for (int i = 0; i < n; ++i) {
        U value;
        std::cin >> value;
        answer ^= value;
    }
    std::cout << answer << '\n';
}
int main() {
    int t;
    std::cin >> t;
    while (t-- > 0) solve();
}
```

Why it works. Within a case, even multiplicities cancel and the single odd group leaves its value. Because the accumulator is local and initialized on every call, no earlier case contributes to a later answer.

Complexity. $O(t + \sum n)$ time and $O(1)$ auxiliary space.

Boundary checks. The survivor may be zero or may occur three times rather than once. Put a zero-only case after a case with a nonzero answer to detect missing resets.

Solution 11: Does an interval match the target XOR?

Idea. Build a padded prefix-XOR array once. Cancel the prefix before l from the prefix through r , then compare the result with the supplied target.

Complete solution | C++23

```
#include <iostream>
#include <vector>
using U = unsigned long long;
int main() {
    int n, q;
    std::cin >> n >> q;
    std::vector<U> pref(n + 1, 0);
    for (int i = 1; i <= n; ++i) {
        U x;
        std::cin >> x;
        pref[i] = pref[i - 1] ^ x;
    }
    while (q-- > 0) {
        int l, r;
        U target;
        std::cin >> l >> r >> target;
        bool match = (pref[r] ^ pref[l - 1]) == target;
        std::cout << (match ? "YES" : "NO") << '\n';
    }
}
```

Why it works. Every value before l occurs in both prefixes and cancels. Every value inside the query occurs exactly once. The computed XOR therefore matches the requested interval, and the equality check answers the decision query.

Complexity. $O(n + q)$ time and $O(n)$ prefix storage.

Boundary checks. Test a single element, a range starting at 1, the whole array, repeated values that cancel, and target zero. Ordinary subtraction is not an XOR inverse.

Solution 12: Reconstruct adjacent-XOR data

Idea. If the previous value is known, XOR it with the next encoded value to recover the next original value. Print and reuse it as the new previous value.

Complete solution | C++23

```
#include <iostream>
using U = unsigned long long;
int main() {
    int n;
    U previous;
    std::cin >> n >> previous;
    std::cout << previous;
    for (int i = 1; i < n; ++i) {
        U encoded;
        std::cin >> encoded;
        previous ^= encoded;
        std::cout << ' ' << previous;
    }
    std::cout << '\n';
}
```

Why it works. $a[i-1] \oplus e[i] = a[i-1] \oplus a[i-1] \oplus a[i] = a[i]$. Starting from the given first value, induction recovers every subsequent value uniquely.

Complexity. $O(n)$ time, including output, and $O(1)$ auxiliary space by streaming.

Boundary checks. At $n = 1$, print only the given value. An encoded zero means consecutive original values are equal; it does not mean the next original value is zero.

Solution 13: Exactly k selected positions

Idea. Enumerate every mask, skip those with a different popcount, then sum the selected values of the remaining masks.

Complete solution | C++23

```
#include <bit>
#include <iostream>
#include <vector>
using U = unsigned long long;
int main() {
    int n, k;
    long long target;
    std::cin >> n >> k >> target;
    std::vector<long long> a(n);
    for (long long& x : a) std::cin >> x;
    long long answer = 0;
    for (U mask = 0; mask < (1ULL << n); ++mask) {
        if (std::popcount(mask) != k) continue;
        long long sum = 0;
        for (int i = 0; i < n; ++i)
            if ((mask & (1ULL << i)) != 0) sum += a[i];
        if (sum == target) ++answer;
    }
    std::cout << answer << '\n';
}
```

Why it works. Each positional subset appears once. Popcount is exactly its selected-position count, and the sum loop evaluates exactly those selected positions. A mask is counted precisely when it satisfies both independent conditions.

Complexity. $O((n+1)2^n)$ worst-case time and $O(n)$ stored input, with constant additional state.

Boundary checks. At $k = 0$, only the empty subset is eligible, and it matches only target zero. Negative values and zero values must remain valid. Do not count distinct sums instead of subsets.

Solution 14: Best benefit within a budget

Idea. Evaluate the total cost and benefit of each mask. Among masks within budget, keep the largest benefit. Initialize the answer to zero because the empty selection is feasible.

Complete solution | C++23

```
#include <iostream>
#include <vector>
using U = unsigned long long;
int main() {
    int n;
    long long budget;
    std::cin >> n >> budget;
    std::vector<long long> cost(n), value(n);
    for (int i = 0; i < n; ++i) std::cin >> cost[i] >> value[i];
    long long best = 0;
    for (U mask = 0; mask < (1ULL << n); ++mask) {
        long long spent = 0, gained = 0;
        for (int i = 0; i < n; ++i) {
            if ((mask & (1ULL << i)) != 0) {
                spent += cost[i];
                gained += value[i];
            }
        }
        if (spent <= budget && gained > best) best = gained;
    }
    std::cout << best << '\n';
}
```

Why it works. Every permitted selection is one enumerated mask. The cost test accepts exactly feasible masks, and the running maximum contains the best benefit among all accepted masks seen so far. After enumeration it is the global optimum.

Complexity. $O((n+1)2^n)$ time and $O(n)$ input storage.

Boundary checks. An item can have zero cost or negative benefit. All-negative benefits should leave the empty solution best. Use wide signed totals before any comparison with the budget.

Solution 15: Recover a minimum team

Idea. Scan selection masks in increasing numeric order, OR the selected skill masks, and save the mask only when a strictly smaller complete team is found.

Complete solution | C++23

```
#include <bit>
#include <iostream>
#include <vector>
using U = unsigned long long;
int main() {
    int n, features;
    std::cin >> n >> features;
    std::vector<U> have(n);
    for (U& x : have) std::cin >> x;
    U full = (1ULL << features) - 1ULL, bestMask = 0;
    int best = n + 1;
    for (U mask = 0; mask < (1ULL << n); ++mask) {
        U covered = 0;
        for (int i = 0; i < n; ++i)
            if ((mask & (1ULL << i)) != 0) covered |= have[i];
        int size = std::popcount(mask);
        if (covered == full && size < best) {
            best = size;
            bestMask = mask;
        }
    }
    if (best == n + 1) { std::cout << "-1\n"; return 0; }
    std::cout << best << '\n';
    if (best == 0) std::cout << "EMPTY";
    bool first = true;
    for (int i = 0; i < n; ++i) {
        if ((bestMask & (1ULL << i)) != 0) {
            if (!first) std::cout << ' ';
            std::cout << i + 1;
            first = false;
        }
    }
    std::cout << '\n';
}
```

Why it works. The coverage invariant accepts exactly complete teams. At each size that improves the answer, increasing mask order finds the smallest numeric mask of that size first. Refusing to replace it on equal size preserves the tie rule. The final saved mask is optimal and can be decoded to member labels.

Complexity. $O((n+1)2^n)$ time and $O(n)$ input storage, plus constant mask state and streamed output.

Boundary checks. No required skills means the empty team is optimal. No people with positive requirements means impossible. The tie rule is numeric-mask order, not lexicographic order of printed member lists.

Solution 16: Count selections without forbidden pairs

Idea. Encode each forbidden pair as a mask with two set positions. A selection violates it exactly when its intersection with that pair equals the complete pair mask.

Complete solution | C++23

```
#include <iostream>
#include <vector>
using U = unsigned long long;
int main() {
    int n, m;
    std::cin >> n >> m;
    std::vector<U> conflict(m);
    for (int i = 0; i < m; ++i) {
        int a, b;
        std::cin >> a >> b;
        conflict[i] = (1ULL << (a - 1)) | (1ULL << (b - 1));
    }
    long long count = 0;
    for (U mask = 0; mask < (1ULL << n); ++mask) {
        bool valid = true;
        for (U pair : conflict) {
            if ((mask & pair) == pair) { valid = false; break; }
        }
        if (valid) ++count;
    }
    std::cout << count << '\n';
}
```

Why it works. The two-bit pair survives the intersection precisely when both forbidden items are selected. Rejecting any such case removes all invalid selections and none of the valid ones. Enumeration counts each remaining positional subset once.

Complexity. $O((m + 1)2^n)$ time and $O(m)$ stored pair masks. A rejection can end that mask's checking early.

Boundary checks. With no forbidden pairs, every one of the 2^n subsets is valid. With $n = 0$, the answer is one. The pair's mask uses item positions, not the input line number.

Solution 17: Count subsets with a requested XOR

Idea. Enumerate all masks. Start each candidate's combined XOR at zero, fold in its selected values, and test the target.

Complete solution | C++23

```
#include <iostream>
#include <vector>
using U = unsigned long long;
int main() {
    int n;
    U target;
    std::cin >> n >> target;
    std::vector<U> a(n);
    for (U& x : a) std::cin >> x;
    long long count = 0;
    for (U mask = 0; mask < (1ULL << n); ++mask) {
        U combined = 0;
        for (int i = 0; i < n; ++i)
            if ((mask & (1ULL << i)) != 0) combined ^= a[i];
        if (combined == target) ++count;
    }
    std::cout << count << '\n';
}
```

Why it works. Each mask identifies one positional subset. Its fold includes each selected value once and no others, so the equality test accepts exactly the requested subsets. Duplicate or zero values do not merge distinct masks.

Complexity. $O((n+1)2^n)$ time and $O(n)$ input storage.

Boundary checks. For 1 1 0 and target zero, four subsets match. At $n = 0$, only target zero has one solution. A combined XOR of zero is a value, not proof that the subset is empty.

Solution 18: Allowed, required, and forbidden bits

Idea. Enumerate all submasks of M using the zero-safe countdown. For each, test full containment of R and disjointness from F .

Complete solution | C++23

```
#include <iostream>
using U = unsigned long long;
int main() {
    U mask, required, forbidden;
    std::cin >> mask >> required >> forbidden;
    bool found = false;
    U sub = mask;
    while (true) {
        if ((sub & required) == required && (sub & forbidden) == 0) {
            std::cout << sub << '\n';
            found = true;
        }
        if (sub == 0) break;
        sub = (sub - 1ULL) & mask;
    }
    if (!found) std::cout << "NONE\n";
}
```

Why it works. The submask loop covers exactly the allowed selections, once each in descending order. The two tests are exactly the remaining requirements. Printing only passing masks therefore gives all valid outputs in the required order.

Complexity. $O(2^p)$ time for $p = \text{popcount}(M)$, plus output, and $O(1)$ auxiliary space.

Boundary checks. Check $M = R = F = 0$, a required bit outside M , overlap between R and F , and a nonempty allowed mask with only zero permitted. Break immediately after processing zero.

Solution 19: Total differing positions over all pairs

Idea. For each of the 60 relevant positions, count ones and zeros. Add `ones * zeros`, without multiplying by the position's power-of-two weight.

Complete solution | C++23

```
#include <iostream>
#include <vector>
using U = unsigned long long;
int main() {
    int n;
    std::cin >> n;
    std::vector<U> a(n);
    for (U& x : a) std::cin >> x;
    long long answer = 0;
    for (int bit = 0; bit < 60; ++bit) {
        long long ones = 0;
        for (U x : a)
            if ((x & (1ULL << bit)) != 0) ++ones;
        answer += ones * (n - ones);
    }
    std::cout << answer << '\n';
}
```

Why it works. A pair contributes one at a given position exactly when one value has a one and the other a zero there. The product counts every such unordered pair once. Summing over positions counts exactly the requested differing-bit contributions.

Complexity. $O(60n + 60)$ time and $O(n)$ stored input. The answer is at most $60n^2/4 \leq 6 \cdot 10^{11}$, which fits `long long`.

Boundary checks. No pairs exist for fewer than two elements. Equal values contribute zero. Include bit 59 in testing and widen pair-count products before multiplication.

Solution 20: Many subset-sum existence queries

Idea. Build a bitset of reachable totals once. Start with only total zero, and for every item combine the old totals with a shifted copy that includes the item. Answer queries by testing their total's bit.

Complete solution | C++23

```
#include <bitset>
#include <iostream>
int main() {
    int n, q;
    std::cin >> n >> q;
    std::bitset<10001> reachable;
    reachable.set(0);
    for (int i = 0; i < n; ++i) {
        int value;
        std::cin >> value;
        reachable |= reachable << value;
    }
    while (q-- > 0) {
        int target;
        std::cin >> target;
        std::cout << (reachable.test(target) ? "YES" : "NO") << '\n';
    }
}
```

Why it works. After each item, a reachable subset either skips it and keeps an old total or includes it and adds its value to an old total. The update represents exactly this union. Later queries inspect the completed state without changing it, so each answer is independent.

Complexity. For $S = 10000$, $O((n + 1)(S + 1) + q)$ scalar-bit work and $O(S)$ bits of state form a conservative model. A typical packed implementation uses $O(n \lceil (S + 1)/w \rceil + q)$ word operations; that packing bound is not a C++23 library complexity guarantee.

Boundary checks. Target zero is always reachable. One value 2 cannot produce total 4. Repeated and zero values are allowed; the bitset answers existence, not the number of constructions.

3.36 Explained trace and complexity drills

Compare your reasoning, not just your numbers. A result can be accidentally correct even when a representation or input assumption is wrong.

Drill 1: Read positions before calculating

$45 = 00101101_2$ and $26 = 00011010_2$. The set positions of 45 are 0, 2, 3, and 5. Its highest set-bit index is 5, that bit's weight is 32, and the popcount is 4.

Operation	Pattern	Value
AND	00001000	8
OR	00111111	63
XOR	00110111	55

The six-position full mask is 63. Flipping the low six bits gives $45 \oplus 63 = 18$, or 010010_2 . An unrestricted complement also flips every higher position in the type, which is outside this six-position request.

Drill 2: A wide destination does not repair a narrow operation

The first shift is invalid on the stated 32-bit-`int` environment: use `1ULL << 40`. The second is valid and sets bit 63. The third is invalid because the count equals 64; no one-bit value 2^{64} fits in this U.

The final condition is valid syntax but groups incorrectly as `mask & ((1ULL << k) == 0)`. The shifted one is nonzero, so the equality is false and the AND result is zero. Repair it to `(mask & (1ULL << k)) == 0`. The destination type and a successful compilation do not establish the intended meaning.

Drill 3: Trace assignment and toggling separately

After	Value	Five-position pattern
Start	3	00011
First set bit 1	3	00011
Second set bit 1	3	00011
Clear bit 0	2	00010
First toggle bit 4	18	10010
Second toggle bit 4	2	00010

The printed check is 1 because bit 1 remains present. Addition is not a repair: adding 2 to the original 3 produces 5 and changes multiple bit positions. The requested operation was assignment of a membership flag, not numeric increment.

Drill 4: What does the input promise buy you?

A XORs to 5, and exactly one value has odd multiplicity, so the single-survivor interpretation is valid. B XORs to zero because $1 \oplus 2 = 3$, but all three values have odd multiplicity. This is the requested counterexample to the claimed pairing test.

C XORs to $9 \oplus 12 = 5$. That result is not either odd-occurrence value. The two-survivor algorithm fits C: a set bit of the combined XOR separates 9 from 12, while both copies of 6 stay in one group and cancel. Correct interpretation depends on the input promise, not merely on running the same reduction.

Drill 5: Prefix cancellation and stale summaries

The original prefix table, including the empty prefix, is 0 5 4 1 6. Query $[2, 4]$ gives $6 \oplus 5 = 3$, while $[3, 3]$ gives $1 \oplus 4 = 5$.

Changing the second input to 3 makes entries 2, 3, and 4 stale. The new table is 0 5 6 3 4, and the new answer for $[2, 4]$ is $4 \oplus 5 = 1$. Every affected prefix contains the modified value. A stored static summary does not update itself just because the input vector changed.

Drill 6: Any requirement versus every requirement

The present features are 0, 3, and 5. R_1 requires 0 and 3, so it is fully contained. R_2 requires 2 and 3; only bit 3 is present. The nonzero-intersection test accepts R_2 incorrectly because it tests “some” rather than “every.”

Use `(have & required) == required`. The six-bit complement is $010110_2 = 22$. Numeric order is unrelated to containment: the original value 41 is greater than $R_2 = 12$, yet it lacks required bit 2. Set inclusion is a condition on positions, not on decimal magnitude.

Drill 7: Duplicate values still have separate positions

Mask	Selected one-based positions	Sum
1	1	2
2	2	2
5	1, 3	2
6	2, 3	2

There are four matching subsets, of which two contain exactly two positions. A set of their sums contains only the single value 2 and loses multiplicity of constructions. With $n = 0$, mask zero represents the one empty selection, whose sum is zero. It therefore contributes one match for target zero.

Drill 8: Two wrong loop endpoints can hide one another

Mask zero is omitted. Mask 8 is included even though a three-item universe has valid masks only from 0 through 7. The included mask has bit 3 set, outside the inner loop's positions 0 through 2. A sum computed only from inspected positions therefore treats it like an empty selection, which can accidentally compensate for omitting mask zero.

That coincidence does not repair the representation. A popcount, printed selection mask, or universe-validity check exposes it. The correct loop is `for (U mask = 0; mask < (1ULL << n); ++mask)` with $0 \leq n < W$ and a small enough n for enumeration. A full-width limit cannot be created by a built-in shift by W .

Drill 9: Submasks and the zero transition

For $1101_2 = 13$, the sequence is

13, 12, 9, 8, 5, 4, 1, 0.

There are three allowed set positions, hence $2^3 = 8$ outputs. The decimal magnitude or the largest allowed position does not by itself give the count; the popcount does.

Applying the update after zero computes an all-ones unsigned value and ANDs it with 13, returning 13. The loop would cycle through the same sequence again. Processing zero followed immediately by a break both includes the empty subset and guarantees termination.

Drill 10: Three loops, three different growth rates

A costs $O(n)$ for one position scan. B costs $O((n+1)2^n)$, including loop overhead and the n position checks per mask. C visits exactly 3^n mask/submask pairs when both loops include zero.

For each universe bit in C there are three states: not in the outer mask, in the outer mask but not the submask, or in both. The forbidden fourth state, in the submask but not the outer mask, is excluded by definition. These independent choices give 3^n pairs. One-word storage makes an individual operation small; it does not reduce the number of iterations.

Drill 11: The same bit storage can mean different things

The bitset has positions 7 and 4 set, and `b[0]` is false. Flipping bit 0 produces `10010001`, with count 3. The string's first character represents the highest displayed position, not bit 0.

A reachable-sum state after one value 2 must contain only totals 0 and 2. Applying the same update twice also makes 4 reachable, as though there were two usable copies of the item. That is wrong for a one-item input. Here bit 0 means total zero is achievable, rather than input item 0 being selected. The same storage type serves a different state definition.

Drill 12: Count pair weights or count differing positions?

For 1 2 3, the XOR values are 3, 2, and 1, giving a sum of 6. Their set-bit counts are 2, 1, and 1, giving a sum of differing positions equal to 4.

At each bit, c ones and $n - c$ zeros produce $c(n - c)$ differing unordered pairs. Multiply by 2^b only for the numeric-XOR sum. For the Hamming-count sum, each differing position contributes one unit.

The product $50000 \cdot 50000 = 2500000000$ exceeds the maximum signed 32-bit `int`. Assignment to `long long` happens too late. Use `1LL * ones * zeros`, or make at least one multiplicand a suitable wide type before multiplication. Check the final weighted total's bound separately too.

3.37 Quick reference, glossary, and readiness checklist

One bit: use a typed weight, not an untyped guess

In these fragments, `U` means unsigned long long; `x` and `mask` have type `U`. Let `W` be `std::numeric_limits<U>::digits`. A bit index must satisfy $0 \leq k < W$. The fragments belong inside a function unless they define a helper.

Job	Expression or update
Build the weight of bit k	<code>U bit = 1ULL << k;</code>
Test whether bit k is set	<code>(mask & bit) != 0</code>
Set bit k	<code>mask = bit;</code>
Clear bit k	<code>mask &= ~bit;</code>
Toggle bit k	<code>mask ^= bit;</code>
Clear the lowest set bit	<code>x &= x - 1;</code> when <code>x != 0</code>
Isolate the lowest set bit	<code>x & (0ULL - x);</code> returns zero for zero.
Test a power of two	<code>x != 0 && (x & (x - 1)) == 0</code>
Count the set bits	<code>std::popcount(x)</code> from <code><bit></code> .
Find the lowest set index	<code>std::countr_zero(x);</code> interpret zero separately.

Build all valid bits, including a full-width universe

Helper above main. Include `<limits>`. The input contract is $0 \leq n \leq W$; validation belongs at the boundary of a general-purpose application.

```
using U = unsigned long long;
U lowMask(int n) {
    const int width = std::numeric_limits<U>::digits;
    return n == width ? ~0ULL : (1ULL << n) - 1;
}
```

For a mask already restricted to the universe, `full ^ mask` is its complement. For an arbitrary word, use `(~mask) & full`. To flip just the low n bits while retaining every higher bit, use `mask ^= full`. These are three related but different operations.

Do not turn this table into unchecked macros

The type, the width, and the zero case are part of each operation's meaning. An expression that worked with a small positive sample can fail when the highest bit is used or when the set is empty.

Sets, XOR, and enumeration: choose the model first

Need	Representation or rule	Essential condition
Any common feature	$(a \ \& \ b) \neq 0$	Same meaning for each bit.
All required features	$(\text{have} \ \& \ \text{need}) == \text{need}$	Requirement may be empty.
Features in a , not b	$a \ \& \ \sim b$	a already bounds the universe.
One odd-frequency value	XOR every input value.	Exactly one distinct value has odd frequency.
Two odd-frequency values	Separate by one set bit of the total XOR.	Exactly two distinct survivors.
Static interval XOR	$\text{pref}[r] \ \wedge \ \text{pref}[l - 1]$	One-based inclusive query; unchanged data.
All position subsets	$\text{mask} = 0$ through $(1\text{ULL} \ll n) - 1$.	$n < W$ and enumeration is affordable.
All submasks of M	Start at M ; use $(s - 1) \ \& \ M$.	Break at zero before updating again.
A fixed larger bit array	<code>std::bitset<N></code> .	N is a compile-time size.
Reachable nonnegative sums	$\text{reach} \ = \ \text{reach} \ll \text{value}$.	Boolean existence, not a count of ways.

A safe submask fragment, including the empty subset

Inside a function. Here `mask` has type `U`; replace the comment with the work required for one submask.

```

U sub = mask;
while (true) {
    // Process sub, including the value zero.
    if (sub == 0) break;
    sub = (sub - 1) & mask;
}

```

The descending order is numeric mask order, not an order by subset size. If p bits of `mask` are set, exactly 2^p submasks are processed. A body that scans n positions adds its own factor of n to the cost.

A glossary to read with the examples

Bit / binary digit

One digit with value 0 or 1. A numeric bit at position i has weight 2^i .

Least significant bit

The rightmost position, index 0. It is not necessarily the lowest *set* bit.

Set bit

A position whose digit is 1. “Set the bit” also means changing that digit to 1.

Mask

A bit pattern with a declared interpretation, such as selected positions or required features.

Universe / full mask

The set of positions that are meaningful for the problem. A full mask has exactly those positions set.

Population count

The number of set bits. It counts selected positions, not the magnitude of the integer.

Bit width

For a positive value, one more than its highest set-bit index. The C++ library returns zero for the value zero.

Hamming distance

The number of positions at which two equal-width bit patterns differ: the population count of their XOR.

Submask

A mask whose set bits are all also set in a specified parent mask.

Unsigned wraparound

Arithmetic on an unsigned type is reduced modulo 2^W . Defined wraparound can still violate the intended mathematical calculation.

Integer promotion

A small integer type may be converted to `int` or `unsigned int` before an operator is evaluated. The written variable type alone may not be the operator’s working type.

Bitwise versus logical

Bitwise operators work on individual digits. Logical operators interpret truth and falsehood; `&&` and `||` also short-circuit.

Packed state

Several yes/no facts stored in one word or in a collection of words. Compact storage is useful, but it does not automatically reduce the number of states an algorithm explores.

Readiness checklist

You are ready to use the core patterns when you can do the following without treating a memorized expression as an explanation:

1. Convert a small value between decimal and binary, then distinguish its highest set-bit *index*, that bit's *weight*, and its *population count*.
2. Explain the truth tables for AND, OR, and XOR; explain why $\wedge$ is not exponentiation in C++.
3. Choose an unsigned operand before shifting, state the valid shift range, and build a low-bit mask without shifting by the width.
4. Check, set, clear, and toggle a bit. Explain why OR sets a flag safely even when the flag was already enabled.
5. Derive XOR cancellation and state the exact frequency promise required by a survivor problem.
6. Interpret a mask as a set. Distinguish any overlap, full containment, and complement inside a bounded universe.
7. Enumerate all position subsets, including the empty one. Keep equal-valued positions distinct when the task treats them as distinct choices.
8. State the difference between an item's selection bit and a feature bit stored in that item's record.
9. Enumerate submasks without cycling after zero. Derive the number of iterations from the parent's set bits.
10. Estimate both evaluation cost and output size. Explain why fitting a selection in 64 bits does not make 2^{64} choices feasible.

A small test pack for almost every mask solution

Use zero, a single set bit, two separated bits, all bits in the declared universe, and the highest allowed position. Add an empty universe where permitted. For selection problems, include duplicate values, zero values, an impossible target, and a solution that uses the empty subset. For count problems, check whether a large count needs `long long`.

For state updates, repeat the same operation twice. Setting twice should not unset a flag; toggling twice should restore it. For XOR ranges, test a single element and a range beginning at the first element.

The explanation to give another trainee

“This bit means This update changes exactly These cases are excluded by the input contract There are this many states ..., and each costs” If one blank is unclear, revisit that idea before attempting a harder problem.

3.38 Technical notes, verification, and references

Edition note. Test counts below record the original content verification. This timetable revision preserves every complete C++ source file byte-for-byte; it changes organization, numbering, and source paths. Historical reports and a new preservation report are included in the source ZIP. The full C++ suites were not rerun for this layout-only revision.

Language version, types, and teaching scope

All 44 complete programs in this chapter use standard C++23 headers. The language references below link to the fixed N4950 C++23 working-draft wording, presented in a public HTML rendering. They are a technical reference, not additional beginner reading. GCC documentation records the behavior of the checking tools and the optional GNU builtin comparison.

The tested environment is GCC 14.2.0 with 32-bit `int`, 64-bit `long long`, and 64-bit `unsigned long long`. C++ guarantees minimum ranges, not these exact sizes on every implementation. Code that needs the unsigned width obtains it from `std::numeric_limits<U>::digits`; individual input contracts still state the intended contest bounds. The alias `U` is `unsigned long long`, not a different numeric representation.^[3-1]

The examples deliberately use unsigned masks so that a set bit is not mistaken for a negative signed value. Ordinary mathematical subset totals and objective values remain signed `long long` when negative input values are permitted. Avoid mixing the two roles just because both are integers.

Promotions, shifts, and the C++23 distinction

A shift first applies integer promotions; its result type is the promoted type of its left operand. The right operand must be nonnegative and strictly smaller than that type's width. Assigning the result into a wider variable does not repair an invalid shift already evaluated with a narrow operand.^{[3-2][3-9]}

In C++23, right shift on a negative signed value is arithmetic and rounds towards negative infinity. Signed integer division instead truncates towards zero. Do not apply old, pre-C++20 explanations of signed right shift to this chapter. The core mask algorithms avoid needing negative signed shifts.^[3-2]

C++23 specifies left-shift results by congruence modulo 2^N , where N is the result type's width, provided the shift count is valid. That rule is not a general permission for signed arithmetic overflow in addition or multiplication. For nonnegative mathematical scaling, separately establish that the answer fits the type; otherwise even a defined unsigned result can lose the high bits you needed.^{[3-1][3-2]}

A displayed 8-bit row is an illustration, not a change to an integer's type. In particular, a narrow character-sized integer often promotes before `~` is applied. The bounded complement examples explicitly mask away bits outside the requested universe.^{[3-3][3-9]}

Complexity conventions and limits

Ordinary word-sized bit operations are treated as constant-cost operations in the algorithmic model used here. An n -position subset loop still has 2^n states. Scanning each position costs $O(n2^n)$, and printing every selected label cannot be made cheaper than the resulting output size.

The standard specifies the effects of `std::bitset` operations but does not give the particular packed-word performance guarantee sometimes assumed in contest discussions. The optional reachable-sum section states a conservative bit-level cost and separately describes the commonly used packed implementation model. A compact expression is not itself a proof of a performance bound.^{[3-10][3-16]}

Input is assumed to follow the printed task contract. These programs are contest solutions, not interactive input-validation interfaces. Output contains only what the statement asks for. Optional examples have deliberately small bounds when either enumeration or answer output grows exponentially.

Original program verification

The source package contains **24 complete examples and 20 complete practice solutions**. All **44 programs** compiled successfully and passed **16,044 automated input/output checks**. The report records the compiler, flags, seed, per-program totals, and the tested unsigned width.

A check means one complete program execution with its full input compared against an independently computed expected result. A program execution containing many queries is counted once, not once per printed answer. Fixed printed samples are part of the checks. Generated cases use arithmetic, sets, binary strings, parity counts, combinations, or small exhaustive searches as appropriate.

The tests include zero and maximum words, the highest bit, full-width low masks, repeated flag changes, empty universes, duplicate-valued positions, signed subset totals, empty selections, missing covers, maximum-capacity bitsets, XOR boundary ranges, and large pair totals. Eight additional grouped assertions check selected standard-library and language boundary behaviors. These utility assertions are reported separately, not included in the 16,044 total.

The checking build used:

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

Sanitizers and library assertions can expose faults on executed paths; passing tests is not a proof over every allowed input. Deliberately incorrect fragments are labelled and are not compiled as working solutions.^[3-13]

Reproduce the checks and build the PDFs

Run these commands from the extracted project root with Python 3 and a C++23-capable GNU compiler installed:

```
python3 tests/verify_bit_manipulation.py
python3 build.py --target session3
python3 build.py --target book
```

The PDF commands require pdfLaTeX and the packages listed in the README. A first compilation creates references and a table of contents; the build script runs the necessary repeated passes. Complete programs each have their own `main`; compile one at a time, not as a single combined executable.

The standalone entry point is `session_03.tex`. The collected entry point is `book.tex`. They include the same chapter files and the same external C++ listings. The fixed sample tables are separate files under `samples/bit_manipulation`; their input and output are also recorded in the test manifest.

The verification suites for Sessions 1 and 2 passed in the original content edition. Together with Session 3, these three sessions contain **176 complete programs** that passed **48,057 input/output checks**. The earlier session-specific reports distinguish their own additional probes from that total. The earlier C++ files were not rewritten as part of this chapter.

Reading the reference markers

A superscript such as [3-2] links to one entry below. The leading 3 identifies this session, so references remain unambiguous in the collected book. References support language and library details; the teaching sequence, worked traces, exercises, and explanations were written for this training.

The central ideas to carry into the next session

One bit can encode one yes/no decision. Independent take-or-skip decisions explain why an n -position set has 2^n subsets. Counting all those objects by generating them may be unnecessary when only the total is wanted. Session 4 will develop counting principles and distinguish which choices can be added, multiplied, or corrected for overlap.

Do not skip the representation step just because an expression is short. A bit mask is useful when it preserves precisely the information the problem needs.

Language and standard-library sources

[3-1] C++23 N4950: fundamental types and unsigned arithmetic

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

[3-2] C++23 N4950: shift operators

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

[3-3] C++23 N4950: unary complement and negation

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

[3-4] C++23 N4950: bitwise AND

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

[3-5] C++23 N4950: bitwise inclusive OR

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

[3-6] C++23 N4950: bitwise exclusive OR

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

[3-7] C++23 N4950: bit counting functions

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

[3-8] C++23 N4950: integral powers of two and bit width

<https://timsong-cpp.github.io/cppwp/n4950/bit.pow.two>

[3-9] C++23 N4950: integral promotions

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

Library behavior and compiler tools

[3-10] C++23 N4950: the bitset class template

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

[3-11] C++23 N4950: logical AND and short-circuit evaluation

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

[3-12] GCC: bit-operation builtins and their argument restrictions

<https://gcc.gnu.org/onlinedocs/gcc/Bit-Operation-Builtins.html>

[3-13] GCC: instrumentation and sanitizer options

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

[3-14] C++23 N4950: integer literals and suffixes

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

[3-15] C++23 N4950: bitset construction and string order

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

[3-16] C++23 N4950: bitset members and boundary behavior

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

[3-17] C++23 N4950: rotating bits

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

Your practice plan

Sheet link: _____

Start date: _____

End date: _____

Use the timeline agreed with your mentors. The blank fields above are for the actual sheet details; they are not an announced schedule.

Three passes through the material

First pass: individual bits. Work through the small operator tables and Practices 1 to 7. For every expression, write the type and the valid bit range. Make zero and the highest allowed bit part of your tests.

Second pass: preserve the right information. Solve the XOR and feature-set problems. State the exact input promise before applying cancellation. Explain whether one bit represents a feature, an input position, or a reachable total.

Third pass: enumeration and analysis. Attempt the subset problems and the submask trace. Count states before running a large case. Return to the optional material only after you can explain the core algorithms without copying their code.

Keep an error log, not only a solved count

Mistake or failed case	What assumption was wrong?	One test to keep

After a contest, revisit unsolved problems without the time pressure. Rebuild a studied solution from its invariant and input contract rather than from remembered lines. A small counterexample you understand is more useful than a long template you cannot justify.