

Level 1 Training

Session 4

Counting Principles

Addition, multiplication, permutations,
combinations, and inclusion-exclusion

Define one outcome. Count it exactly once.
Derive the formula. Protect the arithmetic.

How to use this reference

This book is a reference to study **after the session**. It connects an idea to a precise model, derives the algorithm, and explains how to implement and test it. Read the worked reasoning before using the final tables as a revision sheet.

A study routine for counting

1. **Describe one outcome.** Are you choosing a group, assigning roles, writing a word, distributing identical objects, or counting positions?
2. **List a tiny instance.** Decide when two constructions represent the same final object. A list of six outcomes is a useful test for a formula.
3. **Explain each operation.** Addition needs disjoint cases. Multiplication needs a fixed number of continuations per stage. Division needs a justified repetition count.
4. **Protect the calculation.** Check empty choices, impossible selections, exact divisions, intermediate overflow, and the difference between an exact answer and a remainder.

How examples are labelled

A **complete example** or **complete solution** contains headers and its own `main`. The listing comes from an included `.cpp` file. Compile one file at a time. A **fragment** illustrates a smaller idea; its placement is stated. Deliberately wrong fragments are not working templates.

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

A formula is the end of an explanation, not its beginning

You should be able to say what every term counts and why the parts combine correctly. A number that matches one sample is not a proof that every valid outcome is counted exactly once.

C++23 and prerequisites

The programs use **standard C++23 headers**, without GNU extension headers. Level 0 supplies loops, functions, vectors, maps, and complexity analysis. Session 2 supplies small exhaustive checks; Session 3 supplies masks for inclusion-exclusion. Factorials and binomial coefficients are explained here from the beginning. A small numeric helper is introduced where needed; full number theory follows in later sessions.

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 4: Counting Principles**. 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 4 will help you do

Count a family of outcomes without generating every object. Distinguish choosing a set from assigning ordered roles. Derive factorial, permutation, and combination formulas. Reuse Pascal's triangle, count complements, repair overlaps with inclusion-exclusion, and choose an implementation whose intermediate arithmetic is safe.

Conventions

$C(n, r)$, $\binom{n}{r}$, and nCr mean the same binomial coefficient. $P(n, r)$ and nPr count ordered selections without reuse. Unless stated otherwise, people and positions are distinct, selected groups have no internal order, and input obeys its printed bounds.

There is **one empty selection**, but **zero impossible selections**. Exact counts use `long long` only under an explicit bound. Modular examples state their modulus separately. Complexity counts fixed-width arithmetic operations, including preprocessing and work inside helpers; it does not assume arbitrary-size integers are constant-cost.

Contents

How to use this reference	i
The Level 1 learning path	ii
4 Counting Principles	1
4.1 Count outcomes, not the ways you happen to describe them	2
4.2 The addition rule: separate alternatives	3
4.3 The multiplication rule: complete every stage	4
4.4 Different branches: add products instead of guessing one	5
4.5 Leading zeros and reuse change the model	6
4.6 The complement rule: count what is missing	7
4.7 Factorials: arrange all distinct objects	8
4.8 Permutations: choose distinct objects for ordered roles	9
4.9 Combinations: forget the internal order, not the selected people	10
4.10 A factorial formula is not automatically a safe program	11
4.11 Pascal's identity: split by one distinguished object	12
4.12 Precompute a Pascal table for many queries	13
4.13 One row is enough when earlier rows are not needed	15
4.14 Useful identities: count the same family in two ways	16
4.15 Combine selections from separate groups	17
4.16 At least one member: use a complement, not a designated witness	19
4.17 Choose positions to count binary strings	21
4.18 Grid paths are arrangements of step types	23
4.19 A mandatory point splits a route into two stages	25
4.20 Count pairs by contributions instead of checking every pair	27
4.21 Two overlapping sets: remove one extra copy	28
4.22 Three sets: restore the triple intersection	29
4.23 General inclusion-exclusion: count memberships with alternating signs	31
4.24 Use masks to count a union of divisibility conditions	32
4.25 Require every symbol: exclude missing-symbol events	34
4.26 Exact answers and modular answers need different arithmetic	36
4.27 Debug the model, the boundaries, and the total cost	38
4.28 Optional extension: exact combinations with cancellation	39
4.29 Optional extension: repeated letters and circular order	41
4.30 Optional extension: stars and bars for identical objects	43
4.31 Optional extension: upper bounds become bad events	45
4.32 Optional extension: derangements by forbidden fixed positions	46
4.33 Practice problems: define the outcome before choosing a formula	47
4.34 Trace, modelling, and complexity drills	57
4.35 Hints before solutions	63
4.36 Explained practice solutions	69
4.37 Explained modelling and complexity drills	89
4.38 Quick reference, glossary, and readiness checklist	95
4.39 Technical notes, verification, and references	101

SESSION 4

Counting Principles

Define one outcome. Count it exactly once. Protect the arithmetic.

In Sessions 2 and 3, you generated choices using recursion or masks. Sometimes the task asks only *how many* choices exist. A counting argument can replace an exponential enumeration with a short calculation, but only after you explain what the objects are and how repetitions are handled.

This chapter develops the rules from small examples. The goal is not to recognize a familiar symbol and substitute numbers. It is to build a model whose outcomes match the problem exactly, then compute that model without losing information to overflow or truncation.

Five questions that organize the session

Question	Tool	Typical use
Choose one alternative?	Addition over disjoint cases.	Choose one activity format.
Complete several stages?	Multiplication of choice counts.	Choose every part of an outfit.
Assign distinct roles?	Permutations.	Choose ranked winners.
Select a group?	Combinations.	Choose an unranked committee.
Conditions overlap?	Inclusion-exclusion.	Count numbers satisfying at least one rule.

Your route through the chapter

Stage A: model choices. Define outcomes, split cases, multiply stages, and count the complement when it is simpler.

Stage B: choose and arrange. Derive factorials, permutations, combinations, and Pascal's identity. Use them for groups, bit strings, paths, and pairs.

Stage C: correct overlap. Build two-set, three-set, and general inclusion-exclusion arguments. Translate subsets of conditions into masks.

Stage D: implement and practise. Check numeric limits, distinguish exact from modular arithmetic, solve the problems, and return to optional extensions afterwards.

The sentence to write before any formula

"One outcome is These two descriptions represent the same outcome exactly when" If those blanks are unclear, the counting formula is not ready yet.

4.1 Count outcomes, not the ways you happen to describe them

Start with an object you can recognize

Suppose a group contains trainees A, B, C, and D. Choosing two trainees for an unranked team gives six outcomes:

$$\{A, B\}, \{A, C\}, \{A, D\}, \{B, C\}, \{B, D\}, \{C, D\}.$$

Choosing a captain and a deputy is a different task. The ordered record (A, B) means A is captain and B is deputy. The record (B, A) has different role assignments. There are twelve such records.

The same four people and the same number of selections can produce different answers because **the definition of an outcome changed**. Time order during construction is not necessarily part of the final outcome. Selecting A and then B can create the same team as selecting B and then A.^[4-3]

Distinct positions are not identical values

In an array $[5, 5, 8]$, there are three one-position subsets: choose position 0, 1, or 2. There are only two distinct one-value results: 5 or 8. A problem must tell you which is counted. Do not remove duplicates merely because a **set** is convenient.

Likewise, two different students with equal scores remain two different people. Repeated letters in a final word are different: swapping two identical letters does not create a new word. We will treat that situation separately.

Zero outcomes and one empty outcome

Choosing three different people from two is impossible, so the count is 0. Choosing nobody from two has one result: the empty group. This is why an empty product is 1, why $0! = 1$, and why $C(n, 0) = 1$ will be consistent.

A four-part counting contract

State the objects; decide whether order changes the outcome; state whether reuse is allowed; and identify any restrictions. Keep these statements beside the formula and the code.

Pause and predict: 1

From A, B, and C, how many unranked two-person teams exist? How many captain/deputy assignments exist when a person cannot hold both roles?

Check your prediction. There are three teams: AB, AC, BC. Each team has two role assignments, so there are six assignments. The team AB is one outcome, not two.

4.2 The addition rule: separate alternatives

Disjoint cases can be added

A trainee chooses *one* activity: one of four lectures or one of seven labs. Treat each listed lecture and each listed lab as a distinct activity. The two formats are separate, so the answer is $4 + 7 = 11$.

A collection of outcomes is a **set**. Its size is written $|A|$. Two sets are **disjoint** when no outcome belongs to both. For disjoint sets,

$$|A \cup B| = |A| + |B|.$$

The symbol $\cup$ means union: everything belonging to at least one set. The condition “disjoint” is essential, not decoration.^[4-1]

Turn the statement into a program

Contract. Read counts a, b with $0 \leq a, b \leq 10^9$. The categories describe disjoint activities. Print the number of possible single-activity choices.

Complete example | C++23

```
#include <iostream>
int main() {
    long long lectures, labs;
    std::cin >> lectures >> labs;
    std::cout << lectures + labs << '\n';
}
```

Sample input

4 7

Sample output

11

Each activity belongs to exactly one category, so it is counted once. The calculation uses a constant number of arithmetic operations and $O(1)$ extra storage. The sum is at most $2 \cdot 10^9$; `long long` makes the numeric choice explicit.

A tempting wrong addition

Suppose a club has 12 programmers and 9 designers, but some members do both. The number of people is not automatically 21. Adding the category sizes counts a dual-role member twice. You need the overlap, which we will correct with inclusion-exclusion.

For more than two alternatives, all cases must be **pairwise disjoint**: no outcome can lie in two cases. The cases must also cover every outcome you intend to count.

Pause and predict: 2

There are five morning-only workshops and zero evening-only workshops. You must choose one workshop. Is the answer 0 or 5?

Check your prediction. It is 5. An unavailable alternative contributes zero to a sum. It does not remove the valid outcomes in the other alternative.

4.3 The multiplication rule: complete every stage

A choice tree with equal branching counts

An outfit contains one shirt, one pair of trousers, and one hat. With 3 shirt choices, 5 trouser choices, and 2 hat choices, each shirt leads to 5 trouser choices, and each of those leads to 2 hats. The result is

$$3 \cdot 5 \cdot 2 = 30.$$

Each full outfit determines exactly one choice at each stage. Counting these paths through the choice tree counts every outfit once.^[4-2]

Stage	New choices per partial result	Complete partial results
Start	1	1
Choose shirt	3	3
Choose trousers	5	15
Choose hat	2	30

Contract. Read s, t, h , each between 0 and 10^6 . All combinations are allowed. Print the number of outfits.

Complete example | C++23

```
#include <iostream>
int main() {
    long long shirts, trousers, hats;
    std::cin >> shirts >> trousers >> hats;
    std::cout << shirts * trousers * hats << '\n';
}
```

Sample input

3 5 2

Sample output

30

The maximum product is 10^{18} , which fits the stated signed 64-bit environment. All operands are already `long long`, so widening does not happen only after an overflowing `int` product. Time and extra space are $O(1)$.

The identities may depend on earlier choices

Choosing a captain and then a different deputy from six people gives $6 \cdot 5$. Which five people are available depends on the captain, but *the number* of choices is always five. The product rule does not require probabilistic independence; it needs the asserted number of continuations for every partial choice at that stage.^[4-2]

Pause and predict: 3

The outfit task requires a hat, but there are no hats. Why does the answer become 0 even when shirts and trousers are available?

Check your prediction. No partial outfit can complete the mandatory hat stage. The product contains a zero factor. This differs from an unavailable alternative in the addition rule.

4.4 Different branches: add products instead of guessing one

The number of continuations can change

Suppose a badge is identified by its category, design, and size. Category 1 has 2 designs and 3 sizes; category 2 has 4 designs and 2 sizes; category 3 has 1 design and 5 sizes. Every design can use every size *within its own category*.

Category 1 contributes 6 badges, category 2 contributes 8, and category 3 contributes 5. Since the category is part of the badge's identity, these outcome groups are disjoint:

$$2 \cdot 3 + 4 \cdot 2 + 1 \cdot 5 = 19.$$

Multiplying the total number of designs by the total number of sizes would allow combinations that do not belong to the same category.

Contract. Read m categories, then m pairs (a_i, b_i) . Here $0 \leq m \leq 200000$ and $0 \leq a_i, b_i \leq 10^6$. Print the number of badges.

Complete example | C++23

```
#include <iostream>
int main() {
    int categories;
    std::cin >> categories;
    long long total = 0;
    for (int i = 0; i < categories; ++i) {
        long long designs, sizes;
        std::cin >> designs >> sizes;
        total += designs * sizes;
    }
    std::cout << total << '\n';
}
```

Sample input

3
2 3
4 2
1 5

Sample output

19

The loop invariant is: after processing i categories, **total** counts exactly the badges from those categories. A new category contributes a disjoint product. There are m constant-cost contributions, so time is $O(m+1)$ and extra space is $O(1)$. The total is at most $2 \cdot 10^{17}$.

This is the counting version of a Session 2 decision tree: add the completion counts of distinct branches; equal branch counts permit multiplication.

Pause and predict: 4

One first choice allows two completions, and another allows five. Is the total $2 \cdot 5$, $2 + 5$, or $2 \cdot 2$?

Check your prediction. It is $2+5 = 7$. The first choices define disjoint branches whose continuation counts differ. There is no uniform second-stage count to multiply by two.

4.5 Leading zeros and reuse change the model

A code is not automatically a number

A length-three decimal code can be 007. If every position independently chooses a digit, there are $10^3 = 1000$ codes. A positive three-digit decimal *number* cannot begin with zero, so there are $9 \cdot 10 \cdot 10 = 900$ such numbers.

Now prohibit repeated digits. The first position has 9 choices. After choosing a nonzero digit, the second position has 9 choices: zero plus the eight remaining nonzero digits. The third has 8. Thus there are $9 \cdot 9 \cdot 8 = 648$ positive three-digit numbers with distinct digits.

Contract. Read $1 \leq n \leq 18$. Count positive n -digit decimal numbers with no repeated digits. Leading zero is not allowed; the decimal digits are 0 through 9.

Complete example | C++23

```
#include <iostream>
int main() {
    int n;
    std::cin >> n;
    if (n > 10) {
        std::cout << 0 << '\n';
        return 0;
    }
    long long ways = 9; // The leading digit is not zero.
    for (int i = 1; i < n; ++i) ways *= 10 - i;
    std::cout << ways << '\n';
}
```

Sample input

3

Sample output

648

For $n \leq 10$, after i positions are filled, exactly $10 - i$ digits remain for the next position. The initial 9 handles the special leading position. For $n > 10$, ten available digits cannot fill more than ten positions without reuse, so the answer is 0.

The loop performs at most nine multiplications. In the general position-count model it is $O(n)$; under this ten-digit alphabet it is bounded constant work. The largest answer is $9 \cdot 9! = 3265920$.

Pause and predict: 5

Why is the second factor 9, not 8, for a distinct-digit decimal number?

Check your prediction. Only one digit has been used. Zero was forbidden at the first position, but it is allowed at the second. Of ten digits, nine remain available.

4.6 The complement rule: count what is missing

All outcomes split into good and bad

When a universe U is finite and bad outcomes form a subset B , every outcome is either good or bad, but not both. Therefore

$$\text{good} = |U| - |B|.$$

The universe must be the *same* in both terms. This rule is often useful for phrases such as “at least one,” “not all,” and “avoids.”^[4-4]

For a binary string of length n , each position has two choices, so there are 2^n strings. Exactly one contains no 1s: the all-zero string. Thus $2^n - 1$ strings contain at least one 1.

Contract. Read $0 \leq n \leq 60$. Print the number of length- n binary strings containing at least one 1.

Complete example | C++23

```
#include <iostream>
int main() {
    int n;
    std::cin >> n;
    long long all = 1;
    for (int i = 0; i < n; ++i) all *= 2;
    std::cout << all - 1 << '\n';
}
```

Sample input

5

Sample output

31

The loop maintains 2^i after i iterations. It computes an exact integer power without using floating-point `pow`. Time is $O(n + 1)$ and extra space is $O(1)$. At the maximum bound, both 2^{60} and $2^{60} - 1$ fit.

Why an early choice of the required 1 overcounts

“Choose a position for a 1, then fill every other position” gives $n2^{n-1}$. A string with three 1s is counted three times, because any of them could be the initially designated one. A string with one 1 is counted once. The multiplicities differ, so a single division does not repair this construction.

Pause and predict: 6

For $n = 0$, is the answer 1 because the empty string exists, or 0 because a 1 is required?

Check your prediction. It is 0. The universe has one empty string, but that string has no 1s. Subtracting the one bad outcome gives $2^0 - 1 = 0$.

4.7 Factorials: arrange all distinct objects

Fill the positions without reuse

To arrange n distinct objects in a line, choose the first object in n ways, the second in $n - 1$ ways, and continue. The product is called n **factorial**, written $n!$:

$$n! = n(n - 1) \cdots 2 \cdot 1.$$

For example, six labelled books have $6! = 720$ linear arrangements. Every arrangement determines one unique sequence of choices.^[4-3]

We define $0! = 1$: there is one way to arrange an empty collection, the empty arrangement. This also makes $1! = 1 \cdot 0!$ consistent with the recurrence $n! = n(n - 1)!$.

Contract. Read $0 \leq n \leq 20$. Print the exact factorial.

Complete example | C++23

```
#include <iostream>

long long factorial(int n) {
    long long result = 1;
    for (int i = 2; i <= n; ++i) result *= i;
    return result;
}

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

Sample input

6

Sample output

720

The function returns 1 without entering the loop when $n = 0$ or $n = 1$. After multiplying through i , **result** equals $i!$. It uses $O(n + 1)$ time and $O(1)$ extra storage, rather than listing $n!$ arrangements.

The bound is part of the algorithm

$20! = 2432902008176640000$ fits a signed 64-bit integer. $21! = 51090942171709440000$ does not. A loop with only 21 iterations is fast, but its arithmetic can still be invalid. Computational cost and representable result size are separate questions.

Pause and predict: 7

Why should `factorial(0)` return 1 rather than 0?

Check your prediction. It counts the one empty arrangement. Starting the running product at 1 also makes it the neutral value for multiplication. Starting at 0 would incorrectly make every factorial zero.

4.8 Permutations: choose distinct objects for ordered roles

Stop the factorial product after r factors

From n distinct candidates, assign r distinct roles without letting a candidate occupy two roles. The first role has n choices, the next has $n - 1$, and the last has $n - r + 1$. The count is

$$P(n, r) = nPr = \frac{n!}{(n-r)!} = n(n-1) \cdots (n-r+1), \quad 0 \leq r \leq n.$$

The quotient cancels the factors you never needed; you do not have to compute a large factorial to implement the product.^[4-3]

Contract. Read $0 \leq n \leq 20$ and $-1 \leq r \leq 25$. Print the number of ordered r -selections without reuse. An invalid selection size gives 0.

Complete example | C++23

```
#include <iostream>

long long permutations(int n, int r) {
    if (r < 0 || r > n) return 0;
    long long ways = 1;
    for (int i = 0; i < r; ++i) ways *= n - i;
    return ways;
}

int main() {
    int n, r;
    std::cin >> n >> r;
    std::cout << permutations(n, r) << '\n';
}
```

Sample input

7 3

Sample output

210

For seven candidates and three ranked positions, the factors are $7 \cdot 6 \cdot 5 = 210$. The loop stops after exactly three factors, not after reaching 1. If $r = 0$, the empty assignment contributes 1; if $r > n$ or $r < 0$, the function returns 0 before any multiplication.

Time is $O(r + 1)$ for a valid selection and extra space is $O(1)$. With $n \leq 20$, the product is at most $20!$. Outside that bound, checking only $r \leq n$ does not guarantee safety.

Pause and predict: 8

Six candidates fill two distinct roles. Why is the answer $6 \cdot 5$ rather than 6^2 ?

Check your prediction. Reuse is forbidden. After the first person is chosen, only five different people remain. With reuse allowed, 6^2 would describe a different task.

4.9 Combinations: forget the internal order, not the selected people

Why an ordered count must be divided

Choose three trainees from five for a team with no roles. If you choose the first, second, and third trainee in sequence, you count $P(5, 3) = 60$ ordered lists. But a team such as $\{A, C, E\}$ appears in six lists:

$$ACE, AEC, CAE, CEA, EAC, ECA.$$

Every three-person team has exactly $3! = 6$ internal orders. Grouping the 60 lists into groups of six leaves ten teams. This is the **division rule**: divide only when every desired outcome is represented the same number of times.^[4-3]

For $0 \leq r \leq n$, the number of r -element subsets of n distinct objects is

$$C(n, r) = nCr = \binom{n}{r} = \frac{P(n, r)}{r!} = \frac{n!}{r!(n-r)!}.$$

These are three notations for the same number. The comma in `choose(n, r)` separates arguments; it is not a mathematical operation.

Task on five candidates	Count	Why
Choose three ranked winners	60	$5 \cdot 4 \cdot 3$
Choose an unranked team of three	10	$60/3!$
Choose all five, unranked	1	There is one full group
Choose nobody	1	There is one empty group
Choose six different people	0	The selection is impossible

“Order matters” needs a precise object

A binary string is an ordered sequence, yet counting strings with exactly r ones uses a combination: choose the *set of positions* occupied by ones. The rule does not mean every task involving a sequence must use a permutation. Identify the actual choices and how they determine the final object.

Do not erase identities accidentally

$C(n, r)$ assumes the n selectable objects are distinct. Equal scores do not make different trainees identical. Repeated letters in a resulting word require a different repetition correction, covered in an optional extension.

Pause and predict: 9

A team has four members but one of them must be captain. Is $C(n, 4)$ enough?

Check your prediction. No. First choose the four-person group in $C(n, 4)$ ways, then choose its captain in four ways. The result is $4C(n, 4)$. The other three positions have no separate roles.

4.10 A factorial formula is not automatically a safe program

An exact implementation for a deliberately small bound

For $n \leq 20$, store factorials in `long long`. The formula then gives an exact integer because the denominator divides the numerator.

Contract. Read $0 \leq n \leq 20$ and $-1 \leq r \leq 25$. Print $C(n, r)$, taking invalid selection sizes to have count 0.

Complete example | C++23

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

int main() {
    int n, r;
    std::cin >> n >> r;
    if (r < 0 || r > n) {
        std::cout << 0 << '\n';
        return 0;
    }
    std::vector<long long> fact(n + 1, 1);
    for (int i = 2; i <= n; ++i) fact[i] = fact[i - 1] * i;
    std::cout << fact[n] / (fact[r] * fact[n - r]) << '\n';
}
```

Sample input

7 3

Sample output

35

`fact[0]` is initialized to 1. The invalid-size guard runs before either `fact[r]` or `fact[n-r]` is accessed. For valid r , the denominator product $r!(n-r)!$ is at most $n!$, so both numerator and denominator fit under this input contract.

Building the table costs $O(n+1)$ time and storage. A later query with an already-built factorial table would use $O(1)$ arithmetic operations, but that does not remove the table's construction cost.

A small answer can have enormous intermediate expressions

$C(60, 2) = 1770$, but $60!$ is far too large for a signed 64-bit integer. The small final quotient does not make evaluating the numerator safe. C++ signed overflow is undefined behavior; unsigned wraparound is defined but would not preserve this exact count.^[4-9]

Integer division discards a fractional part. For instance, $5 / 2 * 4$ is 8, not 10. You may reorder an exact mathematical fraction only after establishing that each integer division remains exact. Floating-point approximations are not a replacement when the problem requests an exact integer.^[4-10]

Pause and predict: 10

What should happen before evaluating factorial-array indices when $n = 3$ and $r = 5$?

Check your prediction. Return 0 for an impossible selection. Otherwise `n-r` is negative, so a factorial-table access is invalid even before considering the mathematical formula.

4.11 Pascal's identity: split by one distinguished object

A recurrence from two disjoint cases

To choose r people from n , single out one person, Z . Every team either includes Z or does not.

If Z is included, choose the remaining $r - 1$ members from the other $n - 1$ people: $C(n - 1, r - 1)$.

If Z is excluded, choose all r members from the other $n - 1$ people: $C(n - 1, r)$. These cases are disjoint and exhaustive, so

$$C(n, r) = C(n - 1, r - 1) + C(n - 1, r).$$

This is **Pascal's identity**. It uses addition rather than factorial division.^[4-5]

The boundary entries are $C(n, 0) = C(n, n) = 1$. Values with $r < 0$ or $r > n$ are treated as 0 in our helpers. Start with $C(0, 0) = 1$ and build rows in increasing n .

n	$r = 0$	1	2	3	4	5	6
0	1						
1	1	1					
2	1	2	1				
3	1	3	3	1			
4	1	4	6	4	1		
5	1	5	10	10	5	1	
6	1	6	15	20	15	6	1

For example, $C(6, 2) = C(5, 1) + C(5, 2) = 5 + 10 = 15$. In the two cases, Z belongs to five teams and is absent from ten teams. The recurrence is justified by the objects, not by noticing a pattern in a triangle.

A connection to take-or-skip recursion

Session 2 branched on taking or skipping the next object. Pascal's recurrence combines the counts of those branches while remembering only the remaining selection size. A table computes each entry once instead of repeatedly exploring identical subproblems.

Pause and predict: 11

Why is the row edge $C(n, n)$ equal to 1, rather than $n!$?

Check your prediction. It counts the unranked full group. There is one way to select every person. $n!$ would count their different linear arrangements, which is a different output.

4.12 Precompute a Pascal table for many queries

State the meaning of each cell

$c[n][r]$ stores the exact number of r -element subsets of an n -element set. Build row 0 first. Before building row n , every entry in row $n - 1$ is already available.

Why the table is correct and safe

The initialized edges count empty and full groups. Every interior entry follows the include/exclude proof from the previous topic. Induction on the row number proves that every stored entry has the stated meaning.

All entries through row 60 are at most $C(60, 30) = 118264581564861424$. Each computed sum is the next valid entry, so it also fits. Table allocation and preprocessing cost $O((N + 1)^2)$ time and storage. Each query costs $O(1)$, giving total time $O((N + 1)^2 + q)$.

Pause and predict: 12

With q queries, why not rebuild the entire table inside the query loop?

Check your prediction. The table depends on the maximum row, not on the individual query. Rebuilding repeats $O(N^2)$ work q times. Build once and reuse it, provided the arithmetic convention and modulus have not changed.

Implementation: Precomputed coefficient queries

Contract. Read $0 \leq N \leq 60$ and $0 \leq q \leq 200000$, followed by q queries (n, r) . Each query satisfies $0 \leq n \leq N$ and $-1 \leq r \leq N + 1$. Print $C(n, r)$, or 0 for an invalid selection size.

Complete example | C++23

```
#include <iostream>
#include <vector>
int main() {
    int limit, q;
    std::cin >> limit >> q;
    std::vector<std::vector<long long>>> c(
        limit + 1, std::vector<long long>(limit + 1, 0));
    c[0][0] = 1;
    for (int n = 1; n <= limit; ++n) {
        c[n][0] = c[n][n] = 1;
        for (int r = 1; r < n; ++r) {
            c[n][r] = c[n - 1][r - 1] + c[n - 1][r];
        }
    }
    while (q--) {
        int n, r;
        std::cin >> n >> r;
        std::cout << (r < 0 || r > n ? 0 : c[n][r]) << '\n';
    }
}
```

Sample input

6 4
6 2
4 0
0 0
3 5

Sample output

15
1
1
0

4.13 One row is enough when earlier rows are not needed

Update from right to left

To build only row n , store a single vector. Before the next row begins, `ways[r]` holds the old row's $C(i-1, r)$. The update needs the *old* `ways[r-1]` as well. Visit r in decreasing order so that this left neighbour has not yet changed.

Contract. Read $0 \leq n \leq 60$. Print every entry of Pascal row n , from $C(n, 0)$ through $C(n, n)$.

Complete example | C++23

```
#include <iostream>
#include <vector>
int main() {
    int n;
    std::cin >> n;
    std::vector<long long> ways(n + 1, 0);
    ways[0] = 1;
    for (int i = 1; i <= n; ++i) {
        for (int r = i; r >= 1; --r) ways[r] += ways[r - 1];
    }
    for (int r = 0; r <= n; ++r) {
        if (r > 0) std::cout << ' ';
        std::cout << ways[r];
    }
    std::cout << '\n';
}
```

Sample input

5

Sample output

1 5 10 10 5 1

Trace the change from row 3 to row 4

Start with `[1, 3, 3, 1, 0]`. The new last cell was previously zero.

Update	Vector after the update
$r = 4$: add old cell 3	<code>[1, 3, 3, 1, 1]</code>
$r = 3$: add old cell 2	<code>[1, 3, 3, 4, 1]</code>
$r = 2$: add old cell 1	<code>[1, 3, 6, 4, 1]</code>
$r = 1$: add old cell 0	<code>[1, 4, 6, 4, 1]</code>

The method still performs $1 + 2 + \dots + n$ additions, so time is $O((n+1)^2)$ including setup; extra storage is $O(n+1)$. Reducing memory does not automatically reduce the number of arithmetic operations.

Pause and predict: 13

What goes wrong if the inner loop visits $r = 1, 2, 3, 4$ in increasing order on that trace?

Check your prediction. The first update changes cell 1 from 3 to 4. Cell 2 then uses this new 4 instead of the old 3, becoming 7 instead of 6. The row mixes old and new states and no longer follows Pascal's identity.

4.14 Useful identities: count the same family in two ways

Symmetry comes from choosing who stays out

Selecting r people uniquely determines the $n - r$ people not selected. The correspondence works in both directions, so

$$C(n, r) = C(n, n - r).$$

A helper may therefore replace r by $\min(r, n - r)$ when it needs only one coefficient.

Group all subsets by their size

Every subset of n distinct positions chooses take or skip at each position, giving 2^n subsets. Alternatively, separate them by size:

$$\sum_{r=0}^n C(n, r) = 2^n.$$

The symbol $\sum$ means add the displayed expression for every integer index in the stated range. Different sizes are disjoint cases. The empty subset contributes the $r = 0$ term.^[4-6]

The binomial theorem is the same selection argument

Expand $(a + b)^n$ as a product of n copies of $(a + b)$. Each term chooses either a or b from every factor. To obtain $a^{n-r}b^r$, choose which r factors contribute b :

$$(a + b)^n = \sum_{r=0}^n C(n, r) a^{n-r} b^r.$$

For example, $(1 + x)^4 = 1 + 4x + 6x^2 + 4x^3 + x^4$. The coefficient of x^2 is 6 because there are six ways to choose its two contributing factors. With $a = 1$ and $b = 1$, the identity becomes the subset row sum. With $a = 1$, $b = -1$, and $n > 0$, it gives the alternating identity

$$\sum_{r=0}^n (-1)^r C(n, r) = 0.$$

We will use that cancellation to justify general inclusion-exclusion.

A practical use: check your implementation

For small safe rows, check that the endpoints are 1, the row is symmetric, and the row sum is 2^n . These properties catch many table and loop-direction errors. They supplement, rather than replace, the include/exclude proof.

Pause and predict: 14

The row for $n = 5$ is 1, 5, 10, 10, 5, 1. What does its sum count, and why is it not 5!?

Check your prediction. Its sum is 32, the number of all subsets of five distinct objects. The value $5! = 120$ counts arrangements of all five objects, not subsets of arbitrary sizes.

4.15 Combine selections from separate groups

Choose the required contribution from each group

There are a mentors and b trainees, with no person in both groups. An unranked team of size r must contain exactly k mentors. Choose k of the mentors and $r - k$ of the trainees:

$$\text{ways} = C(a, k)C(b, r - k).$$

These are two stages of one outcome, so multiply. The two selected subgroups uniquely determine the final team, and that team uniquely determines its two subgroups. No additional division by $r!$ is needed.

For $a = 5$, $b = 4$, $r = 3$, and $k = 2$, the count is $C(5, 2)C(4, 1) = 10 \cdot 4 = 40$.

A reusable coefficient helper

The helper below keeps only entries through r , using symmetry first. It initializes the empty selection to 1 and updates right to left. Impossible selection sizes return 0, which lets a formula handle an unavailable subgroup naturally.

All coefficients and the final product fit: the product counts a subfamily of all r -person groups from at most 60 people. When the selection is invalid, one factor is zero instead.

Pause and predict: 15

Why would $C(a + b, r)$ overcount teams that require exactly two mentors?

Check your prediction. It includes every possible number of mentors, not just two. Fix the mentor count using $C(a, 2)C(b, r - 2)$, or add several such products only when several disjoint counts are allowed.

Implementation: Select exact numbers from two groups

Contract. Read a, b, r, k with $a, b \geq 0$, $a + b \leq 60$, $0 \leq r \leq 60$, and $-1 \leq k \leq 61$. Count teams with exactly k members from the first group.

Complete example | C++23

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

long long choose(int n, int r) {
    if (n < 0 || r < 0 || r > n) return 0;
    r = std::min(r, n - r);
    if (r == 0) return 1;
    std::vector<long long> ways(r + 1, 0);
    ways[0] = 1;
    for (int i = 1; i <= n; ++i) {
        for (int j = std::min(i, r); j >= 1; --j) {
            ways[j] += ways[j - 1];
        }
    }
    return ways[r];
}

int main() {
    int a, b, r, k;
    std::cin >> a >> b >> r >> k;
    std::cout << choose(a, k) * choose(b, r - k) << '\n';
}
```

Sample input**Sample output**

5 4 3 2

40

Let $t = \min(r, n - r)$ for a valid call `choose(n, r)`. If $t = 0$, it returns in constant time. Otherwise it uses $O(nt)$ time and $O(t)$ extra storage. The complete two-group calculation has a conservative $O((a + b + 1)^2)$ time bound and $O(a + b + 1)$ extra-space bound.

4.16 At least one member: use a complement, not a designated witness

Subtract the all-trainee teams

To choose an r -person team containing at least one mentor, start with all teams from $a + b$ people and remove those containing only trainees:

$$\text{ways} = C(a + b, r) - C(b, r).$$

A team with one mentor and a team with three mentors each appear once in the total. Exactly the teams with zero mentors are removed.

Why “choose a mentor first” is a trap

The expression $aC(a + b - 1, r - 1)$ marks one mentor as special before filling the team. A team with two mentors is constructed twice; a team with three mentors is constructed three times. Unless a *designated mentor role* is actually part of the outcome, that expression counts the wrong objects.

A range of allowed group sizes

For between L and H mentors, inclusive, add the exact-size cases:

$$\sum_{k=L}^H C(a, k)C(b, r - k).$$

The cases are disjoint because a team’s mentor count is a single integer. Allowing every k gives $C(a + b, r)$, an identity often called Vandermonde’s identity. This provides another useful small-case check.^[4-6]

Pause and predict: 16

If $r = 0$, can the empty team satisfy “at least one mentor”? What does the complement formula return?

Check your prediction. No. The formula gives $C(a + b, 0) - C(b, 0) = 1 - 1 = 0$. Both terms count the same empty team before it is removed.

Implementation: Count teams with at least one mentor

Contract. Read a, b, r with $a, b \geq 0$, $a + b \leq 60$, and $0 \leq r \leq 60$. The first group contains mentors; the second contains trainees. Print the number of unranked teams with at least one mentor.

Complete example | C++23

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

long long choose(int n, int r) {
    if (n < 0 || r < 0 || r > n) return 0;
    r = std::min(r, n - r);
    if (r == 0) return 1;
    std::vector<long long> ways(r + 1, 0);
    ways[0] = 1;
    for (int i = 1; i <= n; ++i) {
        for (int j = std::min(i, r); j >= 1; --j) {
            ways[j] += ways[j - 1];
        }
    }
    return ways[r];
}

int main() {
    int mentors, trainees, r;
    std::cin >> mentors >> trainees >> r;
    std::cout << choose(mentors + trainees, r)
                - choose(trainees, r) << '\n';
}
```

Sample input

3 5 3

Sample output

46

For three mentors, five trainees, and team size three, there are $C(8, 3) - C(5, 3) = 56 - 10 = 46$ teams. The helper uses the same Pascal invariant as before. Total time is at most quadratic in $a + b + 1$, and extra storage is linear.

4.17 Choose positions to count binary strings

A bijection makes the formula precise

A length- n binary string with exactly k ones is uniquely determined by the k positions containing 1. Every k -position subset creates one such string, and reading its 1-positions recovers the same subset. This two-way correspondence is called a **bijection**.

$$\text{binary strings of length } n \text{ with } k \text{ ones} = C(n, k).$$

The bits form a sequence, but we choose an unranked *set of positions*. This is why a combination, rather than a permutation, is the right count.^[4-5]

Exactly, at most, and at least are different requests

Exactly k ones uses one coefficient. At most k ones uses

$$\sum_{j=0}^{\min(k,n)} C(n, j).$$

For $k < 0$, there are no valid nonnegative weights, so this sum is empty and equals 0. At least k ones may be counted by subtracting the weights below k from 2^n .

Pause and predict: 17

For a length-four string with two ones, are 1001 and 0110 different? Does choosing positions 0 then 3 differ from choosing 3 then 0?

Check your prediction. The two strings are different because their selected position sets differ. The two orders of choosing positions 0 and 3 produce the same string 1001, so those construction orders must not be counted separately.

Implementation: Count strings with a fixed number of ones

Contract. Read $0 \leq n \leq 60$ and $-1 \leq k \leq 61$. Print the number of such strings, using 0 when the requested weight is impossible. Leading zeros are allowed.

Complete example | C++23

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

long long choose(int n, int r) {
    if (n < 0 || r < 0 || r > n) return 0;
    r = std::min(r, n - r);
    if (r == 0) return 1;
    std::vector<long long> ways(r + 1, 0);
    ways[0] = 1;
    for (int i = 1; i <= n; ++i) {
        for (int j = std::min(i, r); j >= 1; --j) {
            ways[j] += ways[j - 1];
        }
    }
    return ways[r];
}

int main() {
    int n, k;
    std::cin >> n >> k;
    std::cout << choose(n, k) << '\n';
}
```

Sample input**Sample output**

6 2

15

The helper computes the coefficient in $O(nt)$ time and $O(t)$ extra space for $t = \min(k, n - k) > 0$; boundary or invalid cases return in $O(1)$. No strings are enumerated. Printing every valid string would require much more work when the count is large.

4.18 Grid paths are arrangements of step types

Count moves, not the number of grid points

Start at $(0, 0)$ and finish at (d, r) , where the first coordinate counts downward moves and the second counts rightward moves. Each move increases one coordinate by exactly one. No obstacles exist.

Every route contains $d + r$ moves: exactly d down and r right. Choose the positions of the down moves:

$$\text{routes} = C(d + r, d) = C(d + r, r).$$

Conversely, every such step sequence gives a valid route because it never needs to move backwards or exceed the target coordinate.^[4-5]

Translate a cell-based statement carefully

An R -row, C -column board from its top-left cell to its bottom-right cell requires $R - 1$ down moves and $C - 1$ right moves. Its formula is $C(R + C - 2, R - 1)$, not $C(R + C, R)$.

The helper's time is at most $O((d + r + 1)^2)$ and its extra storage is $O(d + r + 1)$. An obstacle or an additional move type changes the valid sequences; do not keep this formula without a new argument.

Pause and predict: 18

How many routes exist when $d = 0$ and $r = 5$? What about $d = r = 0$?

Check your prediction. Both counts are 1. The first route consists only of right moves. The second is the empty route, which already starts at its destination.

Implementation: Count unrestricted monotone paths

Contract. Read $d, r \geq 0$ with $d + r \leq 60$. Print the number of down/right routes from $(0, 0)$ to (d, r) .

Complete example | C++23

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

long long choose(int n, int r) {
    if (n < 0 || r < 0 || r > n) return 0;
    r = std::min(r, n - r);
    if (r == 0) return 1;
    std::vector<long long> ways(r + 1, 0);
    ways[0] = 1;
    for (int i = 1; i <= n; ++i) {
        for (int j = std::min(i, r); j >= 1; --j) {
            ways[j] += ways[j - 1];
        }
    }
    return ways[r];
}

int main() {
    int down, right;
    std::cin >> down >> right;
    std::cout << choose(down + right, down) << '\n';
}
```

Sample input**Sample output**

3 4

35

For three down moves and four right moves, choose the three down positions among seven: $C(7, 3) = 35$. A route such as RDRRDDR is one of those outcomes. Its construction uses no new combinatorial object beyond selected positions.

4.19 A mandatory point splits a route into two stages

Concatenate a prefix and a suffix

Require a route to pass through (x, y) , with $0 \leq x \leq d$ and $0 \leq y \leq r$. A monotone route visits that point at most once. Split there:

$$\text{through}(x, y) = C(x + y, x) C(d + r - x - y, d - x).$$

The first factor counts routes to the point. The second counts routes from the point to the destination. Every choice of a first segment can be paired with every choice of a second segment; concatenation and splitting are inverse operations.

The product cannot exceed the total route count

Every concatenated result is one valid full route, so the product is at most $C(d + r, d)$. This proves not only correctness but also the numeric bound used by the program. A constant number of coefficient calls keeps the total time quadratic and the extra space linear in $d + r + 1$.

To avoid one forbidden point, subtract the through-point count from all routes. For several forbidden points, simply subtracting each count can remove a route more than once. That is an overlap problem, not a reason to ignore the repeated subtraction.

Pause and predict: 19

If the mandatory point is the start, why does the formula not produce zero?

Check your prediction. There is one empty first segment, counted by $C(0, 0) = 1$. The second segment can be any full route, so multiplying by 1 leaves the total unchanged.

Implementation: Count routes through one required point

Contract. Read d, r, x, y with the inequalities above and $d + r \leq 60$. Print the number of routes through the point. The point may be the start or finish.

Complete example | C++23

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

long long choose(int n, int r) {
    if (n < 0 || r < 0 || r > n) return 0;
    r = std::min(r, n - r);
    if (r == 0) return 1;
    std::vector<long long> ways(r + 1, 0);
    ways[0] = 1;
    for (int i = 1; i <= n; ++i) {
        for (int j = std::min(i, r); j >= 1; --j) {
            ways[j] += ways[j - 1];
        }
    }
    return ways[r];
}

int main() {
    int down, right, x, y;
    std::cin >> down >> right >> x >> y;
    long long before = choose(x + y, x);
    long long after = choose(down + right - x - y, down - x);
    std::cout << before * after << '\n';
}
```

Sample input**Sample output**

3 4 1 2

18

For $(d, r) = (3, 4)$ and $(x, y) = (1, 2)$, the first factor is $C(3, 1) = 3$ and the second is $C(4, 2) = 6$, giving 18 routes. There are 35 routes in total, so 17 avoid the point.

4.20 Count pairs by contributions instead of checking every pair

A frequency is enough for one value group

An array contains f_v occurrences of value v . Choosing two different positions with that value gives $C(f_v, 2) = f_v(f_v - 1)/2$ pairs. Equal-value pairs from different value groups are disjoint, so their total is

$$\sum_v C(f_v, 2).$$

These are pairs of *positions*. Equal values do not make the positions identical.

A streaming version counts a pair when its second position arrives

Before reading a value, its frequency tells us how many earlier equal values exist. Pair the new position with each of them, then increase the frequency. Every pair is counted at its unique later position.

Contract. Read $0 \leq n \leq 200000$ followed by n values in $[-10^9, 10^9]$. Print the number of pairs $i < j$ with equal values.

Complete example | C++23

```
#include <iostream>
#include <map>
int main() {
    int n;
    std::cin >> n;
    std::map<long long, long long> frequency;
    long long pairs = 0;
    for (int i = 0; i < n; ++i) {
        long long value;
        std::cin >> value;
        pairs += frequency[value];
        ++frequency[value];
    }
    std::cout << pairs << '\n';
}
```

Sample input

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

Sample output

```
7
```

In the sample, the four copies of 4 contribute $C(4, 2) = 6$, and the two copies of 2 contribute 1. The total is 7. The streaming additions for value 4 are 0, 1, 2, 3, which sum to the same 6.

If D distinct values occur, ordered-map operations give $O(n \log(D + 1) + 1)$ time and $O(D + 1)$ extra storage. The answer can reach $C(200000, 2) = 19999900000$, so a 32-bit counter is insufficient.

Pause and predict: 20

Why must the frequency increase happen after adding the old frequency to the answer?

Check your prediction. The old frequency counts earlier positions only. Increasing first would also pair the new position with itself, adding one invalid pair per input element.

4.21 Two overlapping sets: remove one extra copy

Write the three disjoint regions

Let A and B be subsets of a finite universe U . The intersection $A \cap B$ contains outcomes in both. The union $A \cup B$ contains outcomes in at least one. Adding $|A| + |B|$ counts every intersection outcome twice, so

$$|A \cup B| = |A| + |B| - |A \cap B|.$$

This is the two-set inclusion-exclusion rule.^[4-4]

If $a = |A|$, $b = |B|$, and $c = |A \cap B|$, the disjoint regions are:

Region	Count
In A only	$a - c$
In both	c
In B only	$b - c$
In neither	$ U - (a + b - c)$

Thus “exactly one” is $a + b - 2c$, while “at least one” is $a + b - c$. The word *or* in a union includes the possibility of both unless the statement explicitly says otherwise.

Contract. Read N, a, b, c , consistent set sizes in a universe of size $0 \leq N \leq 10^9$. Print the union size, the exactly-one size, and the neither size, each on a separate line.

Complete example | C++23

```
#include <iostream>
int main() {
    long long universe, a, b, both;
    std::cin >> universe >> a >> b >> both;
    long long either = a + b - both;
    std::cout << either << '\n';
    std::cout << a + b - 2 * both << '\n';
    std::cout << universe - either << '\n';
}
```

Sample input

30 16 11 5

Sample output

22

17

8

The program uses $O(1)$ time and extra space. Input consistency means $0 \leq c \leq \min(a, b)$ and $a + b - c \leq N$, as well as nonnegative sizes. It is a contest solution under that promise, not a validator for inconsistent survey data.

Pause and predict: 21

If every member of A also belongs to B , so $A \subseteq B$, what should the union formula simplify to?

Check your prediction. The intersection is A itself. The formula becomes $|A| + |B| - |A| = |B|$, as it should.

4.22 Three sets: restore the triple intersection

Follow one outcome instead of memorizing signs

For A, B, C , first add the three set sizes. Then subtract all three pair intersections. An outcome in all three sets has been added three times and subtracted three times: its net contribution is zero. Add the triple intersection once to restore it.^[4-7]

$$\begin{aligned} |A \cup B \cup C| &= |A| + |B| + |C| \\ &\quad - |A \cap B| - |A \cap C| - |B \cap C| \\ &\quad + |A \cap B \cap C|. \end{aligned}$$

The pair intersections include members who also belong to the third set. They are not counts of people in *exactly* those two sets.

Derive exact-membership counts

Write $S_1 = |A| + |B| + |C|$, $S_2 = |A \cap B| + |A \cap C| + |B \cap C|$, and $S_3 = |A \cap B \cap C|$. Let E_j count outcomes belonging to exactly j sets. Then

$$S_1 = E_1 + 2E_2 + 3E_3, \quad S_2 = E_2 + 3E_3, \quad S_3 = E_3.$$

Therefore $E_2 = S_2 - 3S_3$ and $E_1 = S_1 - 2S_2 + 3S_3$.

Pause and predict: 22

A person belongs to all three sets. What is their contribution to $ab + ac + bc$, and why subtract $3abc$ for exactly two?

Check your prediction. They appear in all three pair intersections, contributing 3. They must contribute 0 to exactly two, so their three contributions are removed. Subtracting only one copy would leave two invalid contributions.

Implementation: Compute membership categories

Contract. Read $N, a, b, c, ab, ac, bc, abc$, consistent sizes for three subsets of a universe with $0 \leq N \leq 10^9$. Print counts for at least one set, exactly one, exactly two, and none, in that order.

Complete example | C++23

```
#include <iostream>
int main() {
    long long universe, a, b, c, ab, ac, bc, abc;
    std::cin >> universe >> a >> b >> c >> ab >> ac >> bc >> abc;
    long long singles = a + b + c;
    long long pairs = ab + ac + bc;
    long long either = singles - pairs + abc;
    std::cout << either << '\n';
    std::cout << singles - 2 * pairs + 3 * abc << '\n';
    std::cout << pairs - 3 * abc << '\n';
    std::cout << universe - either << '\n';
}
```

Sample input	Sample output
40 20 18 15 9 7 6 4	35
	21
	10
	5

All operations and storage are constant in number. `long long` protects sums such as $a + b + c$, which can exceed a 32-bit signed range even when the final union fits.

4.23 General inclusion-exclusion: count memberships with alternating signs

Choose which conditions are simultaneously required

Let $A_1, \dots, A_k$ be sets of outcomes satisfying individual conditions. For each nonempty index subset S , count the outcomes satisfying *every* condition whose index belongs to S . This is an intersection. Add odd-sized subsets and subtract even-sized subsets:

$$\left| \bigcup_{i=1}^k A_i \right| = \sum_{\emptyset \neq S \subseteq \{1, \dots, k\}} (-1)^{|S|+1} \left| \bigcap_{i \in S} A_i \right|.$$

The notation is compact but the instruction is simple: singles add, pairs subtract, triples add, and so on.^[4-8]

Why one outcome ends up counted once

Fix an outcome that belongs to exactly $t \geq 1$ of the sets. It occurs in $C(t, 1)$ singleton intersections, $C(t, 2)$ pair intersections, and so on. Its net contribution is

$$C(t, 1) - C(t, 2) + C(t, 3) - \dots + (-1)^{t+1} C(t, t) = 1.$$

The alternating binomial identity from the earlier topic proves this after moving the $C(t, 0) = 1$ term to the other side. For an outcome in no sets, every nonempty intersection omits it, so its contribution is zero. Thus the formula counts precisely the union, once per outcome.

Number of satisfied conditions	Net contribution to the union
0	0
1	1
2	$2 - 1 = 1$
3	$3 - 3 + 1 = 1$
4	$4 - 6 + 4 - 1 = 1$

Count avoidance with the same reasoning

If A_i means “violates rule i ,” valid outcomes belong to none of the bad sets. Count all outcomes in U , then subtract the bad union. Equivalently, include the empty index subset with intersection U and use sign $(-1)^{|S|}$.

Inclusion-exclusion still needs an intersection-counting method

The formula does not compute each intersection for free. Define what selected conditions mean together, derive their count, and include that cost for each of the 2^k index subsets.

Pause and predict: 23

Why do we add intersections of three conditions when counting a union, but subtract them when counting outcomes satisfying none?

Check your prediction. The none count is the universe minus the union. Subtracting the entire union formula reverses every nonempty-subset sign. The two tasks have different starting terms.

4.24 Use masks to count a union of divisibility conditions

The intersection is about a least common multiple

Count integers in $[1, N]$ divisible by at least one of $d_1, \dots, d_k$. For one divisor d , the count is $\lfloor N/d \rfloor$. For several selected divisors, the integer must be divisible by their **least common multiple**, abbreviated LCM: the smallest positive integer divisible by all of them.

For example, multiples of both 4 and 6 are multiples of 12, not necessarily 24. Multiplying divisors is correct only under an additional coprimality condition. Use a general LCM construction instead.

The **greatest common divisor**, GCD, is the largest positive integer dividing both positive arguments. C++ provides `std::gcd` in `<numeric>`. For positive a, b ,

$$\text{lcm}(a, b) = a \frac{b}{\text{gcd}(a, b)}.$$

The common part is already included in a , so only the remaining factor of b is needed. This small helper is enough here; later number theory sessions develop the underlying algorithms.^[4-11]

Cap the LCM before multiplying

If the LCM exceeds N , the selected intersection contains no integer in $[1, N]$. Let `factor = b / gcd(a, b)`. Rather than computing an overflowing `a * factor`, test `a > N / factor`. All quantities are positive, so this comparison exactly decides whether the product would exceed N .

Trace intersections before coding

For divisors 4, 6, and 9 with $N = 20$, singleton counts are 5, 3, 2. Pair LCMs are 12, 36, and 18, with counts 1, 0, 1. The triple LCM is 36, with count 0. The union is $5 + 3 + 2 - 1 - 0 - 1 + 0 = 8$.

There are $2^k - 1$ masks and at most k GCD calculations per mask. This is $O(k2^k)$ GCD calls; using the usual Euclidean $O(\log D)$ arithmetic-operation model gives $O(k2^k \log D)$ time for positive bounds $D \geq 2$. Extra storage is $O(k + 1)$. The absolute sum of all terms is at most $(2^{15} - 1)10^9$, safely below the signed 64-bit limit.

Pause and predict: 24

With divisors 2 and 2, does repeating the same condition invalidate inclusion-exclusion?

Check your prediction. No. Both singleton sets are the same and their intersection is also that same set. Adding twice and subtracting once counts each even integer once. Deduplication may save work, but is not required for this bounded implementation's correctness.

Implementation: Capped intersections of divisibility rules

Contract. Read $0 \leq N \leq 10^9$, $0 \leq k \leq 15$, then k divisors with $1 \leq d_i \leq 10^9$. Duplicate divisors are allowed. Print the union count. No divisor is zero.

Complete example | C++23

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

long long countUnion(long long n, const std::vector<long long>& d) {
    int k = static_cast<int>(d.size());
    long long answer = 0;
    for (unsigned mask = 1; mask < (1U << k); ++mask) {
        long long common = 1;
        int selected = 0;
        bool tooLarge = false;
        for (int i = 0; i < k; ++i) {
            if ((mask & (1U << i)) == 0) continue;
            ++selected;
            long long factor = d[i] / std::gcd(common, d[i]);
            if (common > n / factor) {
                tooLarge = true;
                break;
            }
            common *= factor;
        }
        if (tooLarge) continue; // No positive multiple lies in [1, n].
        long long count = n / common;
        if (selected % 2 == 1) answer += count;
        else answer -= count;
    }
    return answer;
}

int main() {
    long long n;
    int k;
    std::cin >> n >> k;
    std::vector<long long> d(k);
    for (long long& value : d) std::cin >> value;
    std::cout << countUnion(n, d) << '\n';
}
```

Sample input

20 3
4 6 9

Sample output

8

A set bit selects a *condition*, not an integer in $[1, N]$. Odd numbers of selected conditions add and even numbers subtract. If an intersection becomes empty because its LCM exceeds N , adding further divisors cannot make it nonempty again, so that mask may be skipped.

4.25 Require every symbol: exclude missing-symbol events

A different meaning for the selected conditions

Count length- n words over k labelled symbols in which every symbol occurs at least once. Reuse is allowed. There are k^n unrestricted words.

Let A_i mean “symbol i is missing.” If a selected set of j symbols is missing, every position chooses from the remaining $k - j$ symbols, giving $(k - j)^n$ words. There are $C(k, j)$ ways to choose which symbols are missing. Avoid every bad event:

$$\text{all used} = \sum_{j=0}^k (-1)^j C(k, j) (k - j)^n.$$

The empty subset $j = 0$ supplies all words. Unlike the divisor example, the intersection count depends only on the subset’s *size*, so we group equal-sized masks instead of enumerating them individually.^[4-8]

Why the grouped formula is safe

The integer-power helper returns 1 for exponent zero, including a zero base. Here this convention counts the one empty assignment. With positive length and no symbols, it returns 0. If $k > n$, the final count is also 0: there are not enough positions to use every symbol.

For $n = 4$, $k = 3$, the terms are $3^4 - 3 \cdot 2^4 + 3 \cdot 1^4 - 0^4 = 81 - 48 + 3 = 36$. Coefficients count choices of missing symbols; the powers count words over the remaining alphabet.

Using the shown Pascal helper separately in each term gives $O(k^3 + kn + 1)$ time and $O(k + 1)$ extra storage. Precomputing one coefficient row can reduce the coefficient work, but is unnecessary under these small bounds. The absolute sum of terms is at most $2^6 \cdot 6^{12} = 139314069504$, so the alternating accumulation is safe.

Pause and predict: 25

Why is choosing one occurrence position for each required symbol and then filling the rest not generally a correct count?

Check your prediction. A final word may contain the same symbol several times. Any of those positions could have been the designated occurrence, so the construction counts some words multiple times, with different multiplicities. Missing-symbol inclusion-exclusion avoids designating such witnesses.

Implementation: Count words using every symbol

Contract. Read $0 \leq n \leq 12$ and $0 \leq k \leq 6$. Count words using every symbol. With zero positions and zero symbols, the one empty word is valid.

Complete example | C++23

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

long long choose(int n, int r) {
    if (n < 0 || r < 0 || r > n) return 0;
    r = std::min(r, n - r);
    if (r == 0) return 1;
    std::vector<long long> ways(r + 1, 0);
    ways[0] = 1;
    for (int i = 1; i <= n; ++i) {
        for (int j = std::min(i, r); j >= 1; --j) {
            ways[j] += ways[j - 1];
        }
    }
    return ways[r];
}

long long power(long long base, int exponent) {
    long long result = 1;
    for (int i = 0; i < exponent; ++i) result *= base;
    return result;
}

int main() {
    int n, alphabet;
    std::cin >> n >> alphabet;
    long long answer = 0;
    for (int missing = 0; missing <= alphabet; ++missing) {
        long long term = choose(alphabet, missing)
            * power(alphabet - missing, n);
        if (missing % 2 == 0) answer += term;
        else answer -= term;
    }
    std::cout << answer << '\n';
}
```

Sample input

4 3

Sample output

36

4.26 Exact answers and modular answers need different arithmetic

A remainder is not the original integer

Some problems ask for a count modulo M : the remainder in $[0, M - 1]$ after division by a positive M . Addition can be reduced safely after each step because replacing an addend by its remainder changes the total only by a multiple of M .

Pascal's recurrence uses only addition, so it also computes binomial coefficients modulo *any* positive modulus. No primality condition is required. The empty-selection entry becomes $1 \bmod M$, which is 0 when $M = 1$.

Do not divide factorial remainders as ordinary integers

$C(4, 2) = 24/4 = 6$, so $C(4, 2) \bmod 8 = 6$. Reducing the numerator first gives $24 \bmod 8 = 0$, and ordinary $0/4$ gives the wrong answer 0. A modular inverse is a different operation with its own existence conditions; Session 6 will develop it. Do not improvise it by writing $/$.

Pause and predict: 26

Does the Pascal-modulo program still work for $M = 8$, which is not prime? What about $M = 1$?

Check your prediction. Yes to both. The recurrence uses only addition and remainders. With $M = 1$, every output is 0, including the remainder of the one empty selection.

Implementation: Pascal coefficients modulo any positive modulus

Contract. Read $0 \leq N \leq 1000$, $0 \leq q \leq 200000$, and $1 \leq M \leq 10^9$. Each query has $0 \leq n \leq N$, $-1 \leq r \leq N + 1$. Print $C(n, r) \bmod M$, with impossible selections giving 0.

Complete example | C++23

```
#include <iostream>
#include <vector>
int main() {
    int limit, q;
    long long modulus;
    std::cin >> limit >> q >> modulus;
    std::vector<std::vector<long long>>> c(
        limit + 1, std::vector<long long>(limit + 1, 0));
    c[0][0] = 1 % modulus;
    for (int n = 1; n <= limit; ++n) {
        c[n][0] = c[n][n] = 1 % modulus;
        for (int r = 1; r < n; ++r) {
            c[n][r] = (c[n - 1][r - 1] + c[n - 1][r]) % modulus;
        }
    }
    while (q--) {
        int n, r;
        std::cin >> n >> r;
        std::cout << (r < 0 || r > n ? 0 : c[n][r]) << '\n';
    }
}
```

Sample input

```
8 4 8
6 2
8 4
0 0
3 4
```

Sample output

```
7
6
1
0
0
```

Every stored value is between 0 and $M - 1$. Adding two stored entries is less than $2M$, so `long long` safely holds the sum before reduction. The algorithm uses $O((N + 1)^2 + q)$ time and $O((N + 1)^2)$ storage. A very large N still makes this table unsuitable even though its entries are small.

4.27 Debug the model, the boundaries, and the total cost

A correct arithmetic expression can still count the wrong thing

Before inspecting a loop, check what is being counted. A common failure is to count constructions rather than outcomes: choosing a special mentor, arranging an unranked group, or treating equal-valued positions as identical. List a tiny instance and match each counted construction to its final object.

Symptom	Question to investigate
The answer is too large.	Did one outcome appear in several cases or several internal orders?
The answer is too small.	Did you divide without equal multiplicities, or remove overlapping bad events more than once?
Empty input gives zero.	Is there actually one empty choice satisfying the task?
Large inputs become negative.	Did an exact count, product, or intermediate sum overflow?
A Pascal row looks asymmetric.	Were you mixing new and old entries in a one-row update?
An inclusion-exclusion result is wrong.	Do pair intersections include the triple intersection? Are signs for a union or its complement?

Choose a method from the constraints

A factorial formula with $n \leq 20$ is simple for exact counts. A Pascal table is useful for many small-row queries. One-row Pascal is useful when earlier rows are not needed. A few divisibility conditions can use 2^k inclusion-exclusion subsets, but $k = 100$ is not a small extension of $k = 15$.

Count *preprocessing*, every query, and every helper call. Calling a quadratic **choose** helper inside a loop does not make the loop linear. For string or arrangement enumeration, include output size even when the count itself comes from one formula.

Four numeric checks before submitting

Check the largest answer, the largest intermediate term, the largest sum of terms, and whether every integer division is exact. For positive products, divide a limit before multiplying to test a bound. Widen operands *before* an operation, not only its destination. For alternating inclusion-exclusion, a small final union does not imply small partial sums.

The bound $n \leq 60$ in the core coefficient helpers is deliberate. Increasing the limit in one place does not prove that all cells of a newly enlarged table fit. A single requested coefficient may fit even when unused coefficients in the same row do not.

Test independently. Compare formulas with small recursive or masked enumerations. Include $r = 0$, $r = n$, $r > n$, empty groups, identical or nested sets, and duplicate divisors. Reset each test case's accumulator; reuse tables only when their assumptions match.

4.28 Optional extension: exact combinations with cancellation

Avoid both factorials and a quadratic table

For one coefficient, let $r = \min(r, n - r)$ and build it through smaller coefficients:

$$C(n - r + i, i) = C(n - r + i - 1, i - 1) \frac{n - r + i}{i}.$$

At step i , **answer** should become the left-hand side. A direct **answer * numerator / denominator** can overflow during multiplication even when the quotient fits.

Cancel the GCD of the numerator and denominator first. The remaining denominator has no common factor with the remaining numerator. Since the final coefficient is an integer, that denominator must divide the old answer. Divide it out *before* multiplying by the remaining numerator.^[4-11]

A small trace: build $C(5, 2)$

After symmetry, $r = 2$. Each row below begins with the previous exact coefficient.

i	Old answer	Numerator	Denominator	New answer
1	1	4	1	$1 \cdot 4 = 4$
2	4	5	2	$(4/2) \cdot 5 = 10$

The second denominator divides the old answer after the numerator/denominator GCD is removed. In other steps it may cancel partly or entirely with the numerator first.

Pause and predict: 27

Can you always divide the old answer by the denominator before multiplying, without GCD cancellation? Consider the step $5 \cdot 6/2$.

Check your prediction. No. Integer $5/2$ truncates to 2, producing 12 instead of 15. Here the factor 2 should first cancel with 6. The algorithm determines the safe cancellation instead of assuming one operand is divisible.

Implementation: Exact cancellation without overflowing factorials

Contract. Read $0 \leq n \leq 66$ and $-1 \leq r \leq 67$. Print the exact $C(n, r)$, or 0 for an invalid selection size. Every coefficient through row 66 fits in signed 64-bit storage.

Complete example | C++23

```
#include <algorithm>
#include <iostream>
#include <numeric>
long long choose(int n, int r) {
    if (r < 0 || r > n) return 0;
    r = std::min(r, n - r);
    long long answer = 1;
    for (int i = 1; i <= r; ++i) {
        long long numerator = n - r + i;
        long long denominator = i;
        long long g = std::gcd(numerator, denominator);
        numerator /= g;
        denominator /= g;
        answer /= denominator; // The remaining divisor divides answer.
        answer *= numerator;
    }
    return answer;
}
int main() {
    int n, r;
    std::cin >> n >> r;
    std::cout << choose(n, r) << '\n';
}
```

Sample input

66 33

Sample output

7219428434016265740

After step i , the invariant is $\text{answer} = C(n - r + i, i)$. The guarded divisions are exact, and the multiplication produces precisely the next coefficient, rather than a larger un-divided product. With the stated bound, it is at most $C(66, 33) = 7219428434016265740$.

There are r GCD calls, so the Euclidean arithmetic-operation model gives $O(r \log(n + 1) + 1)$ time and $O(1)$ extra storage. The method is not arbitrary precision. For larger n , you must establish a new range guarantee or add checked arithmetic; do not silently rely on overflow.

4.29 Optional extension: repeated letters and circular order

Repeated symbols have uniform internal relabellings

Suppose a word has total length n with letter frequencies $f_1, f_2, \dots$. Temporarily label equal copies separately, such as a_1, a_2 . There are $n!$ labelled arrangements. Each final unlabelled word corresponds to exactly $f_1!f_2!\cdots$ labelled arrangements, because equal copies can trade their internal labels without changing the word. Therefore

$$\text{distinct words} = \frac{n!}{\prod_i f_i!}.$$

The symbol $\prod$ means multiply one factor for every index. Uniform multiplicity is again what makes division valid.^[4-3]

Circular arrangements need a different symmetry contract

For $n \geq 1$ distinct people around an unlabelled circular table, count rotations as the same and reflections as different. Fix one specified person as the reference position, then arrange the other $n - 1$ people around them: $(n - 1)!$.

If the seats are labelled, there are instead $n!$ assignments. If identical objects or reflection equivalence are introduced, symmetry classes can change. Do not divide by n or 2 merely because the drawing is circular; state which transformations preserve the outcome and justify their multiplicities. This section does not attempt arbitrary symmetry counting.

Implementation: Count distinct rearrangements of a word

Contract. Read one line of 0 to 20 lowercase ASCII letters. Print the number of distinct rearrangements of those letters. An empty line has one empty rearrangement.

Complete example | C++23

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

long long factorial(int n) {
    long long result = 1;
    for (int i = 2; i <= n; ++i) result *= i;
    return result;
}

int main() {
    std::string text;
    std::getline(std::cin, text);
    std::vector<int> frequency(26, 0);
    for (char ch : text) ++frequency[ch - 'a'];
    long long denominator = 1;
    for (int count : frequency) denominator *= factorial(count);
    int n = static_cast<int>(text.size());
    std::cout << factorial(n) / denominator << '\n';
}
```

Sample input

Sample output

aabbc

30

For aabbc, the result is $5!/(2!2!1!) = 30$. Each answer word has four labelled versions, no matter where its letters occur. The denominator product never exceeds $n!$, and $n \leq 20$ protects all factorial arithmetic.

Counting characters and building the factorial products costs $O(n + 26)$ time, since the frequencies sum to n . Storage for the input is $O(n)$; the 26-entry frequency table is constant extra storage under this fixed alphabet. These are counts of words, not subsets of the original letter positions.

4.30 Optional extension: stars and bars for identical objects

Choose separator positions instead of object identities

Distribute S identical objects among $K \geq 1$ labelled boxes, allowing empty boxes. An outcome is a count vector $(x_1, \dots, x_K)$ with nonnegative entries and total S .

Represent each object by a star and separate the boxes with $K - 1$ bars. For example,

$$\star \mid \star \star \star \mid \star \longleftrightarrow (1, 3, 1).$$

There are $S + K - 1$ positions, of which $K - 1$ hold bars. Every count vector has exactly one such star/bar representation, even when bars are adjacent. Hence

$$\text{distributions} = C(S + K - 1, K - 1) = C(S + K - 1, S).$$

The boxes are labelled and the objects are identical. Distinct objects would give a different model, usually K^S when every object independently chooses a box.^[4-12]

Minimums can be removed before counting

If box i requires at least b_i objects, give it those objects first. Let $y_i = x_i - b_i \geq 0$. The new total is $S' = S - \sum_i b_i$. If $S' < 0$, there are no distributions. Otherwise use stars and bars with total S' and the same number of boxes.

For at least one object per box, the result is $C(S - 1, K - 1)$ when $S \geq K$. Selecting r objects with repetition from $n \geq 1$ types is the same count-vector model, giving $C(r + n - 1, r)$, not n^r ; n^r counts ordered sequences of selections.

Pause and predict: 28

When distributing identical objects among labelled boxes, does swapping the counts in box 1 and box 2 always preserve the outcome?

Check your prediction. No. The boxes have identities. The vectors $(1, 3)$ and $(3, 1)$ differ. Swapping two individual identical objects inside a fixed distribution, however, changes nothing.

Implementation: Count nonnegative distributions

Contract. Read $S \geq 0$, $K \geq 1$, and $S + K - 1 \leq 60$. Print the number of nonnegative distributions into the K labelled boxes.

Complete example | C++23

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

long long choose(int n, int r) {
    if (n < 0 || r < 0 || r > n) return 0;
    r = std::min(r, n - r);
    if (r == 0) return 1;
    std::vector<long long> ways(r + 1, 0);
    ways[0] = 1;
    for (int i = 1; i <= n; ++i) {
        for (int j = std::min(i, r); j >= 1; --j) {
            ways[j] += ways[j - 1];
        }
    }
    return ways[r];
}

int main() {
    int objects, boxes;
    std::cin >> objects >> boxes;
    std::cout << choose(objects + boxes - 1, boxes - 1) << '\n';
}
```

Sample input**Sample output**

5 3

21

The coefficient helper gives a quadratic upper bound in $S + K$ for time and a linear extra-space bound. With zero objects, all boxes are empty and the answer is 1. With one box, every object goes there and the answer is also 1.

4.31 Optional extension: upper bounds become bad events

Why subtracting a minimum does not solve a maximum

For identical objects in K labelled boxes, suppose every count must satisfy $0 \leq x_i \leq B$. The unrestricted stars-and-bars count includes boxes with $B + 1$ or more objects.

Let A_i be the bad event $x_i \geq B + 1$. If a selected set of j boxes violates its bound, remove $B + 1$ objects from each selected box. The remaining nonnegative total is $S - j(B + 1)$; the other boxes are still present. Therefore the intersection count is

$$C(S - j(B + 1) + K - 1, K - 1)$$

when the remaining total is nonnegative, and 0 otherwise.

There are $C(K, j)$ ways to choose the selected boxes. Applying the avoidance version of inclusion-exclusion gives

$$\sum_{j=0}^K (-1)^j C(K, j) C(S - j(B + 1) + K - 1, K - 1),$$

where terms with negative remaining total are omitted. Other boxes may also violate the bound in an intersection count; inclusion-exclusion is precisely what corrects that overlap.^[4-8]

Trace a small distribution

Take $S = 5$, $K = 3$, $B = 2$. Without upper bounds, there are $C(7, 2) = 21$ distributions. For each selected violating box, remove 3 objects, leaving total 2 and $C(4, 2) = 6$ completions. The three one-box events contribute $3 \cdot 6 = 18$. Two boxes would require at least 6 objects, so all higher intersections are empty.

$$\text{valid} = 21 - 18 = 3.$$

The three distributions are $(1, 2, 2)$, $(2, 1, 2)$, and $(2, 2, 1)$.

Implementation boundaries

Practice 19 implements this model with $0 \leq S \leq 20$, $1 \leq K \leq 10$, and $0 \leq B \leq 20$. Test the remaining total before calling the coefficient helper. It decreases as j increases, so a negative value allows an early stop.

The largest coefficient row used is at most 29. The absolute sum of terms is bounded by $2^{10}2^{29} = 2^{39}$, safely within `long long`. This bounds intermediate sums as well as the final answer. With the shown repeated coefficient helper, a conservative time bound is $O(K(S + K)^2)$ and extra space is $O(S + K)$.

Do not change identical objects to distinct objects unnoticed

The subtraction trick works on count vectors. For distinct objects, fixing a set of overfull boxes does not reduce to the same simple stars-and-bars intersection. The definition of an outcome determines the formula.

4.32 Optional extension: derangements by forbidden fixed positions

A permutation in which nothing stays where it began

Arrange n distinct labels in n positions and forbid label i from position i . Such a permutation is a **derangement**. Let A_i be the bad event that label i is in its original position.

If a selected set of j labels is fixed, the remaining $n - j$ labels may be arranged freely in the remaining positions, giving $(n - j)!$ arrangements. There are $C(n, j)$ choices of the fixed labels. Thus

$$D_n = \sum_{j=0}^n (-1)^j C(n, j) (n - j)! = \sum_{j=0}^n (-1)^j \frac{n!}{j!}.$$

The objects being permuted are distinct and each bad event fixes a different position.^[4-8]

Trace the first few values

For $n = 3$, the terms are $6 - 6 + 3 - 1 = 2$. The two permutations are 231 and 312. Each differs from 123 at every position.

n	0	1	2	3	4	5
D_n	1	0	1	2	9	44

The empty permutation violates no restrictions, so $D_0 = 1$. One label cannot move to another position, so $D_1 = 0$.

Compute integer terms, not fractional approximations

Practice 20 uses the form $n!/j!$. Start with `term = factorial(n)`. After using term j , divide it by $j + 1$ to obtain the next term. Every division is exact because the current term contains that factor when $j < n$.

Do not write an integer expression for $1/j!$ and multiply it by $n!$ later: most of those integer fractions would already be zero. A floating-point alternating series is unnecessary for the stated $n \leq 20$ bound.

With $n \leq 20$, $n!$ fits. The sum of the absolute terms is less than $3n!$ for positive n , which is also below the signed 64-bit limit at $n = 20$. The factorial loop and the term loop together take $O(n + 1)$ time and $O(1)$ extra storage.

Use enumeration to verify the model

For $n \leq 8$, generate permutations from Session 2 and keep those with no fixed positions. Compare the resulting count with the formula. This checks the meaning of a fixed position, the alternating signs, and the empty-permutation case through an independent construction.

4.33 Practice problems: define the outcome before choosing a formula

Try each problem before opening its hint. State what one outcome is, derive the count, and check whether every intermediate value fits. Problems 18 to 20 use the optional extensions.

1. Choose one activity

A trainee must register for exactly one activity: a lecture or a lab. There are a distinct lectures and b distinct labs, with no activity belonging to both lists. Count the possible registrations.

Input. Two integers a, b , with $0 \leq a, b \leq 10^9$.

Output. One integer: the number of single-activity choices.

Sample input	Sample output
8 3	11

2. Two allowed code lengths

A code uses an alphabet of m distinct symbols. Reuse is allowed. Codes must have length exactly n or exactly $n + 1$; their lengths are part of their identity. Count all codes. The empty code is allowed when its length is permitted.

Input. Integers m, n , with $0 \leq m \leq 10$ and $0 \leq n \leq 17$.

Output. One exact integer count.

Sample input	Sample output
3 2	36

3. No repeated decimal digit

Count positive decimal integers with exactly n digits and no repeated digit. Leading zero is forbidden. Do not enumerate the integers.

Input. One integer n , with $1 \leq n \leq 18$.

Output. One integer count.

Sample input	Sample output
2	81

4. A required symbol appears

An alphabet has m distinct symbols, one of which is designated special. Count length- n codes in which the special symbol appears at least once. Reuse and leading symbols of any kind are allowed.

Input. Integers m, n , with $1 \leq m \leq 10$ and $0 \leq n \leq 18$.

Output. One exact integer count.

Sample input	Sample output
4 3	37

5. Assign ranked awards

For each query, assign r distinct ranked awards to different people among n candidates. A person may receive at most one award. Return 0 for an impossible selection size.

Input. $0 \leq q \leq 200000$, followed by q pairs (n, r) with $0 \leq n \leq 20$ and $-1 \leq r \leq 25$.

Output. One count per query. With zero awards, count the one empty assignment.

Sample input	Sample output
4	20
5 2	24
4 4	0
2 3	1
0 0	

6. Many unranked committees

Answer many queries asking how many unranked r -person committees can be selected from n distinct candidates. Prepare the useful counts once.

Input. $0 \leq N \leq 60$ and $0 \leq q \leq 200000$, then q pairs (n, r) with $0 \leq n \leq N$ and $-1 \leq r \leq N + 1$.

Output. One $C(n, r)$ per query; print 0 for invalid sizes.

Sample input	Sample output
8 4	56
8 3	1
6 6	1
3 0	0
2 4	

7. A team needs a mentor

There are a distinct mentors and b distinct trainees, with no overlap. Count unranked teams of size r that contain at least one mentor. There is no designated leader or distinguished mentor.

Input. Integers a, b, r , with $a, b \geq 0$, $a + b \leq 60$, and $0 \leq r \leq 60$.

Output. One exact integer count.

Sample input	Sample output
2 4 2	9

8. A permitted range from the first group

Two disjoint groups contain a and b distinct people. An unranked team has r members and must contain between L and H members of the first group, inclusive. Count the teams.

Input. Integers a, b, r, L, H , with $a, b \geq 0$, $a + b \leq 60$, and $0 \leq L \leq H \leq r \leq 60$.

Output. One exact integer count.

Sample input	Sample output
4 3 3 1 2	30

9. Exactly this many one-bits

Count binary strings of length n containing exactly k ones. Leading zeros are allowed. Strings differing at any position are different outcomes.

Input. Integers n, k , with $0 \leq n \leq 60$ and $-1 \leq k \leq 61$.

Output. One exact integer count; print 0 for an impossible weight.

Sample input	Sample output
7 3	35

10. At most this many one-bits

Count binary strings of length n containing at most k ones. Unlike the previous problem, every permitted weight contributes.

Input. Integers n, k , with $0 \leq n \leq 60$ and $-1 \leq k \leq 61$.

Output. One exact integer count. For $k < 0$, print 0.

Sample input	Sample output
5 2	16

11. Avoid one grid point

A route starts at $(0,0)$ and ends at (d,r) , using only unit down or right moves. It must never visit the forbidden point (x,y) , including at the start or finish. No other point is forbidden.

Input. Integers d, r, x, y , with $d, r \geq 0$, $d + r \leq 60$, $0 \leq x \leq d$, and $0 \leq y \leq r$.

Output. One exact route count.

Sample input	Sample output
3 4 1 2	17

12. Pairs with different values

Given an array, count pairs of distinct positions $i < j$ whose values differ. Equal-valued occurrences are still different positions; they are excluded only because their values are equal.

Input. $0 \leq n \leq 200000$, followed by n integers in $[-10^9, 10^9]$.

Output. One integer: the number of unequal-value position pairs.

Sample input	Sample output
5 2 2 7 9 7	8

13. Exactly one activity group

A universe contains N people. Group A has a members, group B has b , and c belong to both. Count people who belong to exactly one group.

Input. Integers N, a, b, c , representing consistent set sizes with $0 \leq N \leq 10^9$.

Output. One count for exactly one of A and B.

Sample input	Sample output
20 9 8 4	9

14. Exactly two of three groups

Three subsets A, B, and C have the supplied sizes and intersections. Count members belonging to exactly two of the three groups. Every pair intersection includes members who also belong to the third group.

Input. Integers $N, a, b, c, ab, ac, bc, abc$, representing consistent set sizes with $0 \leq N \leq 10^9$.

Output. One count for exactly two groups.

Sample input	Sample output
40 20 18 15 9 7 6 4	10

15. Avoid every listed divisor

Count integers in $[1, N]$ that are divisible by none of the given positive divisors. Divisors may repeat or share factors.

Input. $0 \leq N \leq 10^9$ and $0 \leq k \leq 15$, followed by k integers $1 \leq d_i \leq 10^9$.

Output. One integer: the number avoiding all divisibility conditions.

Sample input	Sample output
20 3 4 6 9	12

16. Every label must be used

Assign one of k distinct labels to each of n distinct positions. Labels may repeat, but every available label must appear somewhere. Count assignments. This is a count of labelled positions, not an unlabelled grouping.

Input. Integers n, k , with $0 \leq n \leq 12$ and $0 \leq k \leq 6$.

Output. One exact count. The assignment with zero positions and zero labels counts once.

Sample input	Sample output
3 2	6

17. Committee counts modulo a supplied number

Answer committee-count queries modulo a supplied positive modulus. The modulus may be composite; do not assume a prime or use ordinary division of factorial remainders.

Input. $0 \leq N \leq 1000$, $0 \leq q \leq 200000$, and $1 \leq M \leq 10^9$, then q pairs (n, r) with $0 \leq n \leq N$ and $-1 \leq r \leq N + 1$.

Output. One $C(n, r) \bmod M$ per query, or 0 for impossible sizes.

Sample input	Sample output
10 4 12	0
10 3	4
8 2	1
0 0	0
5 7	

18. Distinct rearrangements (optional)

Count distinct strings obtainable by rearranging every character of a given lowercase word. Equal copies of a letter are indistinguishable. Use the repeated-symbol extension.

Input. One line of 0 to 20 lowercase ASCII letters. An empty line is valid input.

Output. One exact count of distinct rearrangements.

Sample input	Sample output
banana	60

19. Capacity-limited boxes (optional)

Distribute S identical objects into K labelled boxes. Every box receives between 0 and B objects, inclusive. Count the resulting count vectors, using stars and bars with inclusion-exclusion.

Input. Integers S, K, B , with $0 \leq S \leq 20$, $1 \leq K \leq 10$, and $0 \leq B \leq 20$.

Output. One exact integer count.

Sample input	Sample output
5 3 2	3

20. Nothing stays in its original position (optional)

Permute labels $1, \dots, n$ so that label i never appears at position i . Count these derangements without generating every permutation.

Input. One integer n , with $0 \leq n \leq 20$.

Output. One exact derangement count. The empty permutation counts once.

Sample input	Sample output
5	44

4.34 Trace, modelling, and complexity drills

These questions test the argument behind a formula. Write an explanation before checking the hint. No new complete program is required. Unless a question deliberately asks you to diagnose a mistake, use the chapter's ordinary exact-count conventions.

Drill 1: One set of people, three different tasks

Six distinct students are available. Count each of these outcomes: a two-person committee; a president and a secretary; a two-position code whose symbols are student names and may repeat. Explain why the same two integers, 6 and 2, do not determine one universal formula.

For each task, describe the outcome when the number of selections or positions is zero. Does “choose zero” mean that no valid outcome exists? Worked answer: Drill 1.

Drill 2: When the branch determines the choices

A student chooses exactly one workshop format, then one session and one project within that format. Format A has 2 sessions with 3 projects available in each. Format B has 4 sessions with 2 projects in each. Format C has 1 session with 5 projects.

Count the outcomes. Explain why $(2 + 4 + 1)(3 + 2 + 5)$ is not a valid model. Now suppose 3 final outcomes are listed under both A and B, with no other overlap. What correction is necessary? Worked answer: Drill 2.

Drill 3: A number is not a padded code

Count three-character decimal codes with no repeated digit, allowing a leading zero. Then count three-digit positive integers with no repeated digit. In the second task, explain why the second position still has 9 choices rather than 8.

Would $10^3 - 9^3$ answer either task? State precisely what that expression counts instead. Worked answer: Drill 3.

Drill 4: Designating a witness creates uneven repetition

There are 2 mentors and 3 other students. An unranked three-person team must contain at least one mentor. A proposed answer is $2C(4, 2) = 12$: choose one mentor first, then two others.

Explain the error. Count the valid teams in two correct ways. Would dividing the proposed answer by 2 repair it? Worked answer: Drill 4.

Drill 5: Update a rolling Pascal row in the right direction

A vector currently holds the third row, with a spare zero: 1 3 3 1 0. To build row 4, trace the updates for $j = 4, 3, 2, 1$ using `row[j] += row[j - 1]`.

Next trace the same updates for $j = 1, 2, 3, 4$. Explain which values are no longer from the old row, and why the ascending version computes the wrong recurrence. Worked answer: Drill 5.

Drill 6: Correct mathematics can still have unsafe arithmetic

Assume a 64-bit signed `long long`. Explain why each claim is false:

- A. “If $C(n, r)$ fits, evaluating $n!/(r!(n - r)!)$ must fit.”
- B. “Whenever ab/c is an integer, $(a/c)b$ gives the same integer result.”
- C. “We can compute a combination modulo 8 by taking factorial remainders and dividing them as ordinary integers.”

Give a small counterexample for B and C, then name a safe method for $n \leq 60$. Worked answer: Drill 6.

Drill 7: Fixed weight, all weights, and indistinguishable bits

How many six-bit strings contain exactly two 1s? How many contain at most two 1s? How many six-bit strings exist in total?

Explain why $P(6, 2)$ is too large for the first question. Relate the three answers to row 6 of Pascal's triangle and to the subset masks from Session 3. Worked answer: Drill 7.

Drill 8: A path through a point, then a path avoiding it

A path starts at $(0, 0)$ and reaches $(3, 4)$ using only a step increasing the first coordinate or a step increasing the second coordinate. Count all paths, paths through $(1, 2)$, and paths avoiding $(1, 2)$.

Why can the two segments be multiplied? Would subtracting the through-point count once for each of two forbidden points always be correct? Worked answer: Drill 8.

Drill 9: Pair intersections already contain the triple

In a universe of 40 objects, let $|A| = 20$, $|B| = 18$, $|C| = 15$, $|A \cap B| = 9$, $|A \cap C| = 7$, $|B \cap C| = 6$, and $|A \cap B \cap C| = 4$. Pair intersections include objects in all three sets.

Find the union, exactly one set, exactly two sets, at least two sets, and none. Check that the disjoint categories add to 40. Worked answer: Drill 9.

Drill 10: Intersections of divisibility rules

Among the integers 1 through 20, count those divisible by at least one of 4, 6, and 9. List the selected numbers as an independent check. Explain why the 4-and-6 intersection uses 12 rather than 24.

Now give the union count for just two identical rules, “divisible by 4” and “divisible by 4”. What are the answers when the list of rules is empty or when $N = 0$? Worked answer: Drill 10.

Drill 11: Missing-symbol events and the empty word

Count length-three words over the two distinct symbols A and B that use both symbols. Do this by listing outcomes and by inclusion-exclusion. Explain what $C(k, j)(k - j)^n$ represents in the general all-symbols formula.

Give the all-symbols count when (n, k) is $(0, 0)$, $(0, 2)$, or $(3, 0)$. In this counting model, why must the power helper return 1 for exponent zero, including a zero base? Worked answer: Drill 11.

Drill 12: Choose the method, include the hidden work

Choose a method and give preprocessing, per-query, and storage costs where relevant:

- A. Up to 200000 exact $C(n, r)$ queries with $0 \leq n \leq 60$.
- B. Up to 200000 queries with $0 \leq n \leq 1000$, asking for coefficients modulo 12.
- C. Count a union of $k \leq 15$ divisibility events on $[1, N]$ without scanning N integers.
- D. Compute one exact coefficient $C(66, 33)$. Explain why using the $n \leq 20$ factorial helper is not a valid shortcut for D. Worked answer: Drill 12.

4.35 Hints before solutions

Use a hint to recover the idea, then close this section and try the implementation again. A hint is not a substitute for defining the outcomes.

Hint 1: Choose one activity

These are alternatives, not two required stages. Check that the activity lists are disjoint. Return to the problem.

Hint 2: Two allowed code lengths

Count each length with a product, then add the two disjoint length classes. Preserve one empty code even with an empty alphabet. Return to the problem.

Hint 3: No repeated decimal digit

Handle the first position separately. Zero becomes available at later positions. More than ten distinct decimal digits are impossible. Return to the problem.

Hint 4: A required symbol appears

The complement consists of codes over the other $m - 1$ symbols. Do not choose a designated occurrence of the special symbol. Return to the problem.

Hint 5: Assign ranked awards

The awards are distinct roles. Multiply r decreasing factors and reset the product for every query. Return to the problem.

Hint 6: Many unranked committees

A Pascal table covers all queries. Define $c[n][r]$ precisely and guard invalid indices before access. Return to the problem.

Hint 7: A team needs a mentor

Count all r -person teams, then subtract the teams formed only from trainees. Return to the problem.

Hint 8: A permitted range from the first group

Split by the exact number k selected from the first group. Cases with impossible subgroup sizes contribute zero. Return to the problem.

Hint 9: Exactly this many one-bits

Select the positions of the ones. The order in which you select those positions is not part of the final string. Return to the problem.

Hint 10: At most this many one-bits

Build row n once, then sum its entries for weights 0 through $\min(k, n)$. Return to the problem.

Hint 11: Avoid one grid point

Subtract routes that pass through the point from all routes. Split a through-point route uniquely into two segments. Return to the problem.

Hint 12: Pairs with different values

When a new value arrives after i earlier items, subtract its earlier equal-value frequency from i . Return to the problem.

Hint 13: Exactly one activity group

Add the A-only and B-only regions. The overlap is included in both reported group sizes. Return to the problem.

Hint 14: Exactly two of three groups

In the sum of pair intersections, an exactly-two member appears once, while an all-three member appears three times. Return to the problem.

Hint 15: Avoid every listed divisor

Find the union of bad divisibility sets using inclusion-exclusion, then subtract from N . Selected divisors combine by LCM, not always by product. Return to the problem.

Hint 16: Every label must be used

Define a bad event for each missing label. Every chosen set of j missing labels leaves $(k - j)^n$ assignments. Return to the problem.

Hint 17: Committee counts modulo a supplied number

Pascal's identity uses only addition. Reduce each entry after adding two previous-row entries. Return to the problem.

Hint 18: Distinct rearrangements (optional)

Temporarily label equal copies. Divide $n!$ by the product of the factorials of their frequencies. Return to the problem.

Hint 19: Capacity-limited boxes (optional)

A bad box has at least $B + 1$ objects. For j specified bad boxes, remove $j(B + 1)$ objects and count the remaining nonnegative distributions. Return to the problem.

Hint 20: Nothing stays in its original position (optional)

For j chosen fixed labels, the remaining $(n - j)$ labels can be arranged freely. Simplify the inclusion-exclusion term $C(n, j)(n - j)!$ before coding. Return to the problem.

Hints for the modelling and complexity drills

Drill 1

Ask whether order matters and whether the second selection may reuse the first. List the outcomes for two students before working with six.

Drill 2

Count continuations inside one format, then combine the formats. With duplicate listings, distinguish the construction path from the final outcome.

Drill 3

The first position of a positive three-digit integer cannot be zero. Once a nonzero digit is used, zero becomes an available later choice.

Drill 4

Compare a team containing one mentor with a team containing both mentors. How many first-stage choices generate each one?

Drill 5

For each assignment, label the right-hand entries as “old row” or “already changed”. An update in place must preserve the needed old values.

Drill 6

For A use $n = 30, r = 1$. For B use $a = 3, b = 2, c = 2$. For C compare the actual value of $C(4, 2)$ with factorial remainders.

Drill hints, continued**Drill 7**

Select the positions occupied by 1s. The two 1s have no individual names, but the six positions are distinct.

Drill 8

Count words of three moves of one type and four of the other. A route through a specified point splits uniquely into a prefix and a suffix.

Drill 9

An object in all three sets appears three times in the sum of pair intersections, not once. First obtain the count belonging to exactly two.

Drill 10

An intersection means divisibility by the least common multiple. Duplicate *events* must still cancel correctly in inclusion-exclusion.

Drill 11

Subtract words missing A and words missing B. Their intersection misses both. For zero positions, there is one empty assignment, not one symbol choice.

Drill 12

Queries with the same bound can share a table. A composite modulus rules out ordinary factorial division, but does not affect Pascal addition. For C, count event masks and their intersections.

4.36 Explained practice solutions

Solution 1: Choose one activity

Idea. Add the two category counts. Selecting an activity does not also require selecting an activity from the other category.

Complete solution | C++23

```
#include <iostream>
int main() {
    long long lectures, labs;
    std::cin >> lectures >> labs;
    std::cout << lectures + labs << '\n';
}
```

Why it works. Each valid registration belongs to exactly one of the two disjoint categories. The addition rule therefore counts every registration once and no invalid registration.

Cost and numeric bounds. $O(1)$ time and extra space. The sum is at most $2 \cdot 10^9$ and is evaluated using `long long`.

Boundary checks. One empty category leaves the other category's choices. Two empty categories give 0, since choosing one activity is mandatory.

Solution 2: Two allowed code lengths

Idea. There are m^n codes of length n and m^{n+1} of the other length. Use an integer loop for each power, then add.

Complete solution | C++23

```
#include <iostream>

long long power(long long base, int exponent) {
    long long result = 1;
    for (int i = 0; i < exponent; ++i) result *= base;
    return result;
}

int main() {
    int alphabet, n;
    std::cin >> alphabet >> n;
    std::cout << power(alphabet, n) + power(alphabet, n + 1) << '\n';
}
```

Why it works. Each position independently selects a symbol, so the product rule gives the power for a fixed length. A code cannot have both lengths, so addition joins the classes without overlap. The empty product counts the one empty code.

Cost and numeric bounds. $O(n + 1)$ time and $O(1)$ extra space. At the largest bounds, the sum is $10^{17} + 10^{18} = 1.1 \cdot 10^{18}$, which fits.

Boundary checks. For $m = 0, n = 0$, only the empty code exists and the answer is 1. For $m = 0, n > 0$, no allowed code exists.

Solution 3: No repeated decimal digit

Idea. Use 9 choices for the leading digit, then multiply by the numbers of unused digits. Return 0 for lengths above 10.

Complete solution | C++23

```
#include <iostream>
int main() {
    int n;
    std::cin >> n;
    if (n > 10) {
        std::cout << 0 << '\n';
        return 0;
    }
    long long ways = 9; // The leading digit is not zero.
    for (int i = 1; i < n; ++i) ways *= 10 - i;
    std::cout << ways << '\n';
}
```

Why it works. The leading position can contain any nonzero digit. At later position $i + 1$, exactly i digits have been used, leaving $10 - i$ choices, including zero when it has not been used. The product counts each digit sequence once.

Cost and numeric bounds. At most nine loop iterations, hence $O(1)$ work under the fixed decimal alphabet; $O(1)$ extra space. The maximum count is $9 \cdot 9! = 3265920$.

Boundary checks. For $n = 1$, all nine nonzero digits are valid. For $n = 11$, the count is 0, not a negative product.

Solution 4: A required symbol appears

Idea. Subtract the codes containing no special symbol from all codes: $m^n - (m - 1)^n$.

Complete solution | C++23

```
#include <iostream>

long long power(long long base, int exponent) {
    long long result = 1;
    for (int i = 0; i < exponent; ++i) result *= base;
    return result;
}

int main() {
    int alphabet, n;
    std::cin >> alphabet >> n;
    std::cout << power(alphabet, n) - power(alphabet - 1, n) << '\n';
}
```

Why it works. Every code either uses the special symbol or avoids it. The first term counts the full universe; the second counts exactly the bad class. Subtraction leaves each valid code once, regardless of how many special occurrences it has.

Cost and numeric bounds. $O(n + 1)$ time and $O(1)$ extra space. Each power is at most 10^{18} , so both the terms and their difference fit.

Boundary checks. With $n = 0$, the answer is 0. With $m = 1$ and positive n , exactly one code exists and it uses the special symbol.

Solution 5: Assign ranked awards

Idea. For each valid query, compute $P(n, r) = n(n-1) \cdots (n-r+1)$. Invalid sizes return 0 before the product loop.

Complete solution | C++23

```
#include <iostream>

long long permutations(int n, int r) {
    if (r < 0 || r > n) return 0;
    long long ways = 1;
    for (int i = 0; i < r; ++i) ways *= n - i;
    return ways;
}

int main() {
    int q;
    std::cin >> q;
    while (q--) {
        int n, r;
        std::cin >> n >> r;
        std::cout << permutations(n, r) << '\n';
    }
}
```

Why it works. The first award has n recipients, and every assigned award removes one person from later choices. Each complete sequence of recipients uniquely identifies one ranked assignment. A fresh helper call gives independent state to every query.

Cost and numeric bounds. Total time is $O(q + \sum r_i)$ over valid queries, at most $O(21q + 1)$ here. Extra space is $O(1)$. Every product is at most $20!$.

Boundary checks. Include $r = 0$, $r = n$, $r > n$, and an empty batch. An invalid query must not affect the next query's count.

Solution 6: Many unranked committees

Idea. Build all Pascal entries through row N , then use each query as a table lookup after checking its selection size.

Complete solution | C++23

```
#include <iostream>
#include <vector>
int main() {
    int limit, q;
    std::cin >> limit >> q;
    std::vector<std::vector<long long>> c(
        limit + 1, std::vector<long long>(limit + 1, 0));
    c[0][0] = 1;
    for (int n = 1; n <= limit; ++n) {
        c[n][0] = c[n][n] = 1;
        for (int r = 1; r < n; ++r) {
            c[n][r] = c[n - 1][r - 1] + c[n - 1][r];
        }
    }
    while (q--) {
        int n, r;
        std::cin >> n >> r;
        std::cout << (r < 0 || r > n ? 0 : c[n][r]) << '\n';
    }
}
```

Why it works. Row edges count empty and full committees. Every interior entry adds the disjoint cases that include or exclude one distinguished person. Induction proves the table values, and guarded lookup returns the requested one.

Cost and numeric bounds. $O((N + 1)^2 + q)$ time and $O((N + 1)^2)$ storage. Every entry through row 60 fits in long long.

Boundary checks. For $n = 0, r = 0$, print 1. For $r > n$ or $r < 0$, print 0 without indexing a nonexistent cell.

Solution 7: A team needs a mentor

Idea. Use $C(a+b, r) - C(b, r)$. The shared coefficient helper returns 0 for impossible sizes.

Complete solution | C++23

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

long long choose(int n, int r) {
    if (n < 0 || r < 0 || r > n) return 0;
    r = std::min(r, n - r);
    if (r == 0) return 1;
    std::vector<long long> ways(r + 1, 0);
    ways[0] = 1;
    for (int i = 1; i <= n; ++i) {
        for (int j = std::min(i, r); j >= 1; --j) {
            ways[j] += ways[j - 1];
        }
    }
    return ways[r];
}

int main() {
    int mentors, trainees, r;
    std::cin >> mentors >> trainees >> r;
    std::cout << choose(mentors + trainees, r)
                - choose(trainees, r) << '\n';
}
```

Why it works. All teams form a disjoint union of those with no mentor and those with at least one. The second coefficient counts precisely the first group. Unlike choosing a mentor first, the complement does not attach an unintended distinguished role.

Cost and numeric bounds. At most $O((a+b+1)^2)$ time and $O(a+b+1)$ extra storage with the shown helper. Both terms fit the row-60 bound, and the result is nonnegative.

Boundary checks. No mentors means answer 0. An empty team also fails the requirement. A size larger than the total population gives 0.

Solution 8: A permitted range from the first group

Idea. For every k from L through H , add $C(a, k)C(b, r - k)$. Each product counts one exact composition.

Complete solution | C++23

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

long long choose(int n, int r) {
    if (n < 0 || r < 0 || r > n) return 0;
    r = std::min(r, n - r);
    if (r == 0) return 1;
    std::vector<long long> ways(r + 1, 0);
    ways[0] = 1;
    for (int i = 1; i <= n; ++i) {
        for (int j = std::min(i, r); j >= 1; --j) {
            ways[j] += ways[j - 1];
        }
    }
    return ways[r];
}

int main() {
    int a, b, r, low, high;
    std::cin >> a >> b >> r >> low >> high;
    long long answer = 0;
    for (int k = low; k <= high; ++k) {
        answer += choose(a, k) * choose(b, r - k);
    }
    std::cout << answer << '\n';
}
```

Why it works. A team has exactly one first-group size, so the cases cannot overlap. Within a case, the two selected subgroups are independently chosen and uniquely determine the team. Their product counts every team of that composition once.

Cost and numeric bounds. Let $N = a + b$ and $h = H - L + 1$. The repeated helper gives a conservative $O(h(N + 1)^2)$ time bound and $O(N + 1)$ extra storage. Every partial sum counts a subset of all size- r teams, so it fits.

Boundary checks. With $L = 0, H = r$, the answer equals $C(a + b, r)$. Do not cap the allowed range by guessing; zero-returning coefficient calls already handle unavailable sizes.

Solution 9: Exactly this many one-bits

Idea. The count is $C(n, k)$. Compute it with the guarded one-row Pascal helper.

Complete solution | C++23

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

long long choose(int n, int r) {
    if (n < 0 || r < 0 || r > n) return 0;
    r = std::min(r, n - r);
    if (r == 0) return 1;
    std::vector<long long> ways(r + 1, 0);
    ways[0] = 1;
    for (int i = 1; i <= n; ++i) {
        for (int j = std::min(i, r); j >= 1; --j) {
            ways[j] += ways[j - 1];
        }
    }
    return ways[r];
}

int main() {
    int n, k;
    std::cin >> n >> k;
    std::cout << choose(n, k) << '\n';
}
```

Why it works. Each k -position subset gives one string by putting 1 there and 0 elsewhere. Reading the positions of the ones reverses the construction. This bijection proves the coefficient is the desired count.

Cost and numeric bounds. For $t = \min(k, n - k) > 0$, time is $O(nt)$ and extra space is $O(t)$. Invalid and boundary cases return in $O(1)$. The row-60 bound protects all additions.

Boundary checks. The empty string is valid for $n = 0, k = 0$. With $k < 0$ or $k > n$, no string exists.

Solution 10: At most this many one-bits

Idea. Use a right-to-left one-row Pascal construction, then add the coefficients of all allowed weights.

Complete solution | C++23

```
#include <algorithm>
#include <iostream>
#include <vector>
int main() {
    int n, k;
    std::cin >> n >> k;
    std::vector<long long> ways(n + 1, 0);
    ways[0] = 1;
    for (int i = 1; i <= n; ++i) {
        for (int r = i; r >= 1; --r) ways[r] += ways[r - 1];
    }
    long long answer = 0;
    for (int r = 0; r <= std::min(k, n); ++r) answer += ways[r];
    std::cout << answer << '\n';
}
```

Why it works. The row contains $C(n, r)$ for each weight r . Two different weights cannot describe the same binary string, so summing these disjoint cases counts exactly the strings with weight at most k .

Cost and numeric bounds. $O((n + 1)^2)$ time and $O(n + 1)$ extra storage. The sum is at most 2^{60} , including all possible weights, so it fits.

Boundary checks. When $k \geq n$, the answer must be 2^n . With $k = 0$, the only valid string is all zeros, including the empty string when $n = 0$.

Solution 11: Avoid one grid point

Idea. Compute $C(d+r, d) - C(x+y, x)C(d+r-x-y, d-x)$.

Complete solution | C++23

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

long long choose(int n, int r) {
    if (n < 0 || r < 0 || r > n) return 0;
    r = std::min(r, n - r);
    if (r == 0) return 1;
    std::vector<long long> ways(r + 1, 0);
    ways[0] = 1;
    for (int i = 1; i <= n; ++i) {
        for (int j = std::min(i, r); j >= 1; --j) {
            ways[j] += ways[j - 1];
        }
    }
    return ways[r];
}

int main() {
    int down, right, x, y;
    std::cin >> down >> right >> x >> y;
    long long all = choose(down + right, down);
    long long through = choose(x + y, x)
        * choose(down + right - x - y, down - x);
    std::cout << all - through << '\n';
}
```

Why it works. All monotone routes correspond to choices of down-step positions. Every route visiting the forbidden point has a unique prefix ending there and suffix starting there. The product counts precisely those bad routes, and the complement removes them once.

Cost and numeric bounds. A constant number of coefficient calls gives at most $O((d+r+1)^2)$ time and $O(d+r+1)$ extra space. The through-point product cannot exceed the total route count.

Boundary checks. Forbidding the start or destination gives 0. With no moves, the only point is both start and destination, so the forbidden-point answer is also 0.

Solution 12: Pairs with different values

Idea. Maintain counts of previously read values. For the new position, add the number of earlier positions with a different value, then update its frequency.

Complete solution | C++23

```
#include <iostream>
#include <map>
int main() {
    int n;
    std::cin >> n;
    std::map<long long, long long> frequency;
    long long answer = 0;
    for (int i = 0; i < n; ++i) {
        long long value;
        std::cin >> value;
        answer += i - frequency[value];
        ++frequency[value];
    }
    std::cout << answer << '\n';
}
```

Why it works. Every valid pair has one unique later endpoint. At that endpoint, the earlier positions split into equal and unequal values; $i - \text{frequency}[\text{value}]$ counts exactly the unequal ones. No pair is repeated or omitted.

Cost and numeric bounds. With D distinct values, time is $O(n \log(D + 1) + 1)$ and extra storage is $O(D + 1)$. The count can reach $C(200000, 2) = 19999900000$, so `long long` is necessary in the tested environment.

Boundary checks. All equal values give 0; all distinct values give $n(n - 1)/2$. Empty and one-element arrays contain no pairs.

Solution 13: Exactly one activity group

Idea. Return $(a - c) + (b - c) = a + b - 2c$.

Complete solution | C++23

```
#include <iostream>
int main() {
    long long universe, a, b, both;
    std::cin >> universe >> a >> b >> both;
    std::cout << a + b - 2 * both << '\n';
}
```

Why it works. A-only and B-only are disjoint and together are exactly the requested people. Each intersection member contributes once to each group total, so both contributions must be removed.

Cost and numeric bounds. $O(1)$ time and extra space. The arithmetic uses `long long`; all input counts are consistent with the stated universe.

Boundary checks. If the groups coincide, the answer is 0. If they are disjoint, it is $a + b$. The union size would use only one subtraction of c , not two.

Solution 14: Exactly two of three groups

Idea. Compute $ab + ac + bc - 3abc$. The singleton group sizes are not needed once the pair and triple intersections are known.

Complete solution | C++23

```
#include <iostream>
int main() {
    long long universe, a, b, c, ab, ac, bc, abc;
    std::cin >> universe >> a >> b >> c >> ab >> ac >> bc >> abc;
    std::cout << ab + ac + bc - 3 * abc << '\n';
}
```

Why it works. A member of exactly two groups lies in one pair intersection. A member of all three lies in all three intersections and must be removed entirely. Members of at most one group do not enter the sum. Thus the expression has exactly the desired per-person contribution.

Cost and numeric bounds. $O(1)$ time and extra space. Use `long long` for the sum and the triple correction, even if the final count is small.

Boundary checks. If every person belongs to all three groups, the answer is 0. Pair-intersection inputs must not be confused with already-disjoint exactly-two counts.

Solution 15: Avoid every listed divisor

Idea. Reuse the capped-LCM intersection calculation for every nonempty condition mask. Add odd intersections, subtract even ones, and return N minus that union.

Complete solution | C++23

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

long long countUnion(long long n, const std::vector<long long>& d) {
    int k = static_cast<int>(d.size());
    long long answer = 0;
    for (unsigned mask = 1; mask < (1U << k); ++mask) {
        long long common = 1;
        int selected = 0;
        bool tooLarge = false;
        for (int i = 0; i < k; ++i) {
            if ((mask & (1U << i)) == 0) continue;
            ++selected;
            long long factor = d[i] / std::gcd(common, d[i]);
            if (common > n / factor) {
                tooLarge = true;
                break;
            }
            common *= factor;
        }
        if (tooLarge) continue; // No positive multiple lies in [1, n].
        long long count = n / common;
        if (selected % 2 == 1) answer += count;
        else answer -= count;
    }
    return answer;
}

int main() {
    long long n;
    int k;
    std::cin >> n >> k;
    std::vector<long long> d(k);
    for (long long& value : d) std::cin >> value;
    std::cout << n - countUnion(n, d) << '\n';
}
```

Why it works. For a selected condition set, the common multiples are exactly the multiples of its LCM. If that LCM exceeds N , the intersection is empty. Inclusion-exclusion counts each integer satisfying at least one bad condition once, so its complement counts precisely the valid integers.

Cost and numeric bounds. $O(k2^k)$ GCD calls and $O(k+1)$ storage. Under the Euclidean model, time is $O(k2^k \log D + 1)$ for divisor bound $D \geq 2$. The cap avoids oversized products; the sum of absolute terms is at most $(2^{15} - 1)10^9$.

Boundary checks. An empty divisor list allows all N integers. A divisor of 1 allows none. With $N = 0$, the interval is empty and the answer is 0.

Solution 16: Every label must be used

Idea. Evaluate $\sum_{j=0}^k (-1)^j C(k, j) (k - j)^n$ with exact coefficient and power helpers.

Complete solution | C++23

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

long long choose(int n, int r) {
    if (n < 0 || r < 0 || r > n) return 0;
    r = std::min(r, n - r);
    if (r == 0) return 1;
    std::vector<long long> ways(r + 1, 0);
    ways[0] = 1;
    for (int i = 1; i <= n; ++i) {
        for (int j = std::min(i, r); j >= 1; --j) {
            ways[j] += ways[j - 1];
        }
    }
    return ways[r];
}

long long power(long long base, int exponent) {
    long long result = 1;
    for (int i = 0; i < exponent; ++i) result *= base;
    return result;
}

int main() {
    int n, alphabet;
    std::cin >> n >> alphabet;
    long long answer = 0;
    for (int missing = 0; missing <= alphabet; ++missing) {
        long long term = choose(alphabet, missing)
            * power(alphabet - missing, n);
        if (missing % 2 == 0) answer += term;
        else answer -= term;
    }
    std::cout << answer << '\n';
}
```

Why it works. The unrestricted universe has k^n assignments. For any selected missing-label set, all positions choose from the remaining labels. Grouping those intersections by size produces the formula, and the avoidance form of inclusion-exclusion leaves only assignments using every label.

Cost and numeric bounds. With the shown repeated Pascal helper, $O(k^3 + kn + 1)$ time and $O(k + 1)$ extra storage. The sum of absolute terms is below $2^6 6^{12}$ at the stated bounds.

Boundary checks. For $k > n$, the count is 0. The power helper treats an empty assignment as one possibility, including the 0^0 term when both quantities are zero.

Solution 17: Committee counts modulo a supplied number

Idea. Build the Pascal table modulo M and answer guarded lookups. Initialize edge counts to $1 \bmod M$, not unconditionally to 1.

Complete solution | C++23

```
#include <iostream>
#include <vector>
int main() {
    int limit, q;
    long long modulus;
    std::cin >> limit >> q >> modulus;
    std::vector<std::vector<long long>>> c(
        limit + 1, std::vector<long long>(limit + 1, 0));
    c[0][0] = 1 % modulus;
    for (int n = 1; n <= limit; ++n) {
        c[n][0] = c[n][n] = 1 % modulus;
        for (int r = 1; r < n; ++r) {
            c[n][r] = (c[n - 1][r - 1] + c[n - 1][r]) % modulus;
        }
    }
    while (q--) {
        int n, r;
        std::cin >> n >> r;
        std::cout << (r < 0 || r > n ? 0 : c[n][r]) << '\n';
    }
}
```

Why it works. The exact include/exclude recurrence remains valid after taking remainders, because addition preserves congruence modulo M . Induction therefore proves every stored residue. No modular division, and hence no prime-modulus assumption, is needed.

Cost and numeric bounds. $O((N + 1)^2 + q)$ time and $O((N + 1)^2)$ storage. Each unreduced addition is at most $2(M - 1)$ and safely fits in `long long`.

Boundary checks. With $M = 1$, every answer is 0. Include $n = r = 0$, an impossible size, and an empty batch of queries.

Solution 18: Distinct rearrangements (optional)

Idea. Count each letter, build the uniform relabelling factor, and return $n! / \prod f_i!$.

Complete solution | C++23

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

long long factorial(int n) {
    long long result = 1;
    for (int i = 2; i <= n; ++i) result *= i;
    return result;
}

int main() {
    std::string text;
    std::getline(std::cin, text);
    std::vector<int> frequency(26, 0);
    for (char ch : text) ++frequency[ch - 'a'];
    long long denominator = 1;
    for (int count : frequency) denominator *= factorial(count);
    int n = static_cast<int>(text.size());
    std::cout << factorial(n) / denominator << '\n';
}
```

Why it works. There are $n!$ arrangements when all copies are labelled. Each final word has exactly $\prod f_i!$ labelled versions, obtained by exchanging labels only among equal letters. Dividing by that common multiplicity counts each distinct word once.

Cost and numeric bounds. $O(n + 26)$ time. Input storage is $O(n)$ and the fixed alphabet table is $O(1)$ extra storage. The numerator and denominator are at most $20!$.

Boundary checks. An empty word has one rearrangement. An all-equal word also has one. An all-distinct word has $n!$ arrangements.

Solution 19: Capacity-limited boxes (optional)

Idea. Sum $(-1)^j C(K, j) C(S - j(B + 1) + K - 1, K - 1)$ over nonnegative remaining totals. Stop when the remaining total becomes negative.

Complete solution | C++23

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

long long choose(int n, int r) {
    if (n < 0 || r < 0 || r > n) return 0;
    r = std::min(r, n - r);
    if (r == 0) return 1;
    std::vector<long long> ways(r + 1, 0);
    ways[0] = 1;
    for (int i = 1; i <= n; ++i) {
        for (int j = std::min(i, r); j >= 1; --j) {
            ways[j] += ways[j - 1];
        }
    }
    return ways[r];
}

int main() {
    int total, boxes, capacity;
    std::cin >> total >> boxes >> capacity;
    long long answer = 0;
    for (int j = 0; j <= boxes; ++j) {
        int remaining = total - j * (capacity + 1);
        if (remaining < 0) break;
        long long term = choose(boxes, j)
            * choose(remaining + boxes - 1, boxes - 1);
        if (j % 2 == 0) answer += term;
        else answer -= term;
    }
    std::cout << answer << '\n';
}
```

Why it works. Choosing j violating boxes identifies an intersection of bad events. Removing $B + 1$ objects from each selected box is reversible and leaves a stars-and-bars problem with the same K boxes. Inclusion-exclusion cancels every distribution with any overfull box.

Cost and numeric bounds. A conservative bound with the helper is $O(K(S + K)^2)$ time and $O(S + K)$ extra space. Rows are at most 29, and the sum of absolute terms is at most 2^{39} under these bounds.

Boundary checks. For $S = 0$, every box is empty and the answer is 1. With $B = 0, S > 0$, no distribution exists. Also test $S > KB$ and $K = 1$.

Solution 20: Nothing stays in its original position (optional)

Idea. Compute the alternating sum of exact terms $n!/j!$. Build the first term once, and divide by $j + 1$ to obtain the next term after each iteration.

Complete solution | C++23

```
#include <iostream>

long long factorial(int n) {
    long long result = 1;
    for (int i = 2; i <= n; ++i) result *= i;
    return result;
}

int main() {
    int n;
    std::cin >> n;
    long long term = factorial(n); // n! / 0!
    long long answer = 0;
    for (int j = 0; j <= n; ++j) {
        if (j % 2 == 0) answer += term;
        else answer -= term;
        if (j < n) term /= j + 1;
    }
    std::cout << answer << '\n';
}
```

Why it works. An intersection fixing j specified labels leaves $(n - j)!$ permutations. Choosing the fixed labels gives coefficient $C(n, j)$. The avoidance form of inclusion-exclusion yields $\sum (-1)^j n! / j!$, and each successive division is exact for $j < n$.

Cost and numeric bounds. $O(n + 1)$ time and $O(1)$ extra space. For $n \leq 20$, the factorial and even the bound $3n!$ on the absolute term sum fit in signed 64-bit storage.

Boundary checks. Check $D_0 = 1$, $D_1 = 0$, and $D_2 = 1$. Do not evaluate $1/j!$ with integer division before multiplying by $n!$.

4.37 Explained modelling and complexity drills

Compare the counted objects as well as the numerical answers. A correct formula should also explain the empty, impossible, and overlapping cases.

Drill 1: One set of people, three different tasks

An unranked committee has $C(6, 2) = 15$ outcomes. Two distinct roles without reuse have $P(6, 2) = 6 \cdot 5 = 30$. A two-position code with reuse has $6^2 = 36$. The committee $\{A, B\}$ equals $\{B, A\}$, but the two role assignments differ. The code also permits AA .

With zero selections, the empty committee, empty role assignment, or empty word is one valid outcome. Thus $C(6, 0) = P(6, 0) = 6^0 = 1$. Zero ways is reserved for an impossible request, such as choosing seven different students from six.

Drill 2: When the branch determines the choices

The first model gives $2 \cdot 3 + 4 \cdot 2 + 1 \cdot 5 = 6 + 8 + 5 = 19$. Within a format the continuation count is fixed, but it differs across formats. The product $7 \cdot 10$ incorrectly pairs a session from one format with a project choice from another.

If three *final outcomes* belong to both the A list and the B list, and format labels are not part of their identity, subtract those three duplicate counts: $19 - 3 = 16$. If choosing a different format itself makes a different outcome, there is no overlap to correct. Outcome identity must be stated first.

Drill 3: A number is not a padded code

The code count is $10 \cdot 9 \cdot 8 = 720$. The integer count is $9 \cdot 9 \cdot 8 = 648$. After choosing the first nonzero digit, exactly one of ten digits is used; the nine remaining choices include zero.

The expression $10^3 - 9^3 = 271$ counts length-three decimal codes containing at least one occurrence of one designated digit, such as zero, with repetition allowed. It does not forbid repeated digits, and it does not enforce the leading-position rule for three-digit integers.

Drill 4: Designating a witness creates uneven repetition

A team with one mentor is generated once; a team with both is generated twice. Therefore there is no common overcount factor. Dividing 12 by 2 gives 6 and discards valid teams.

A complement count gives $C(5, 3) - C(3, 3) = 10 - 1 = 9$. Disjoint cases give $C(2, 1)C(3, 2) + C(2, 2)C(3, 1) = 6 + 3 = 9$. The invalid witness construction counts those same cases with multiplicities 1 and 2: $6 + 2 \cdot 3 = 12$.

Drill 5: Update a rolling Pascal row in the right direction

Descending updates produce $[1, 3, 3, 1, 1]$, $[1, 3, 3, 4, 1]$, $[1, 3, 6, 4, 1]$, then $[1, 4, 6, 4, 1]$. Each update reads two entries from row 3.

Ascending updates produce $[1, 4, 3, 1, 0]$, $[1, 4, 7, 1, 0]$, $[1, 4, 7, 8, 0]$, then $[1, 4, 7, 8, 8]$. After the first step, the left neighbour belongs to the partly constructed new row. Pascal's identity needs two values from the previous row, so this changes the meaning of the computation.

Drill 6: Correct mathematics can still have unsafe arithmetic

A fails because $C(30, 1) = 30$ fits, while $30!$ does not fit signed 64-bit storage. The final result does not bound all intermediate values.

B fails because $(3/2) \cdot 2 = 2$ under integer division, whereas $(3 \cdot 2)/2 = 3$. Dividing first is safe only after proving exact divisibility at that step.

C fails because $C(4, 2) = 6$, but $4! \bmod 8 = 0$ and $2! \bmod 8 = 2$; ordinary division of 0 by $2 \cdot 2$ gives 0, not 6. Pascal's addition recurrence computes exact coefficients safely for $n \leq 60$. Reducing each addition modulo any stated positive $M \leq 10^9$ also works, without division.

Drill 7: Fixed weight, all weights, and indistinguishable bits

Exactly two 1s gives $C(6, 2) = 15$. At most two gives $C(6, 0) + C(6, 1) + C(6, 2) = 1 + 6 + 15 = 22$. All strings give $2^6 = 64$, the sum of row 6: $1 + 6 + 15 + 20 + 15 + 6 + 1$.

$P(6, 2) = 30$ treats the first chosen 1-position and the second chosen 1-position as ordered roles. Choosing positions 1 then 4 produces the same string as choosing 4 then 1, so every string is counted twice. Each valid mask represents one subset of positions; popcount selects the row coefficient that counts its weight.

Drill 8: A path through a point, then a path avoiding it

All routes: $C(7, 3) = 35$. To reach $(1, 2)$, use one first-coordinate move and two second-coordinate moves: $C(3, 1) = 3$. The remaining route uses two moves of each type: $C(4, 2) = 6$. Thus 18 routes pass through the point, and $35 - 18 = 17$ avoid it.

Every through-point path has one unique pair of segments, and any legal first segment combines with any legal second segment. Multiplication is therefore justified. For two forbidden points, paths visiting both would be subtracted twice; their intersection must be handled, and it may be empty if their order is incompatible with monotone movement.

Drill 9: Pair intersections already contain the triple

Let $S_1 = 53$, $S_2 = 22$, and $S_3 = 4$. Exactly two is $S_2 - 3S_3 = 10$. Exactly one is $S_1 - 2S_2 + 3S_3 = 21$. The union is $S_1 - S_2 + S_3 = 35$. At least two is $10 + 4 = 14$, and none is $40 - 35 = 5$.

The disjoint categories give $21 + 10 + 4 + 5 = 40$. In particular, $S_2 - S_3 = 18$ is *not* the number in at least two sets: a triple member would still contribute twice.

Drill 10: Intersections of divisibility rules

The singles contribute $5 + 3 + 2 = 10$. Pair least common multiples are 12, 36, and 18, giving $1 + 0 + 1 = 2$. The triple least common multiple is 36, giving 0. The result is $10 - 2 + 0 = 8$; the numbers are 4, 6, 8, 9, 12, 16, 18, 20.

Both 4 and 6 divide 12, so using their product would miss a valid intersection member. Two identical rules give $5 + 5 - 5 = 5$. With no rules, the union is empty and contributes 0; its complement contains all N integers. With $N = 0$, both union and complement counts are 0.

Drill 11: Missing-symbol events and the empty word

The valid words are AAB, ABA, BAA, ABB, BAB, and BBA: 6. Inclusion-exclusion gives $2^3 - 2 \cdot 1^3 + 0^3 = 6$. Choosing j symbols that must be absent gives $C(k, j)$ event subsets; each leaves $(k - j)^n$ unrestricted words.

For $(0, 0)$ there is one empty word and no missing-symbol requirement, so the answer is 1. For $(0, 2)$ no word uses both required symbols, so it is 0; the formula is $1 - 2 + 1 = 0$. For $(3, 0)$ no symbol can fill a position, so it is 0. The convention $0^0 = 1$ here counts the unique assignment to zero positions; it is a combinatorial convention, not a claim about limits.

Drill 12: Choose the method, include the hidden work

For A, precompute exact Pascal entries through $N = 60$: $O(N^2)$ time and space, then $O(1)$ per query and $O(q)$ total query work. For B, use the same recurrence modulo 12 through $N = 1000$; costs are the same in terms of N and q . All entries stay below 12, so no inverse or ordinary division is required.

For C, examine $2^k - 1$ nonempty masks, scan up to k events, and build capped least common multiples. With Euclidean GCD cost included, a conventional bound is $O(2^k k \log(D + 1))$, where D bounds divisor inputs; input storage is $O(k)$ and working scalars need $O(1)$ extra space.

For D, the cancellation helper is $O(r \log(n + 1))$ time and $O(1)$ extra storage for $r = \min(33, 33)$. A Pascal row through 66 is another safe option: all coefficients through row 66 fit signed 64-bit storage. The factorial helper's stated precondition is violated, and $66!$ is vastly too large even though the requested coefficient fits.

4.38 Quick reference, glossary, and readiness checklist

Use these pages after studying the derivations. A formula should come with its outcome definition, its assumptions, and a safe way to calculate it.

Choose the counting model

Task	Count	Essential condition
Choose one case	$a_1 + \cdots + a_t$	Cases are disjoint and cover the required outcomes.
Complete all stages	$a_1 a_2 \cdots a_t$	Every partial choice at a stage has the stated number of continuations.
Ordered roles, no reuse	$P(n, r) = n! / (n - r)!$	Objects are distinct and $0 \leq r \leq n$.
Unranked group, no reuse	$C(n, r) = n! / [r!(n - r)!]$	Selected objects are distinct; their order is irrelevant.
Ordered positions, reuse	m^n	Each of n positions may independently choose any of m symbols.
All subsets	2^n	There are n distinct positions with take-or-skip choices.
Nonempty subsets	$2^n - 1$	Exclude the one empty selection.
Avoid a condition	all – bad	The universe is fixed and the bad outcomes are counted once.

Coefficients and their boundary values

$$\begin{aligned}
 C(n, 0) &= C(n, n) = 1, & C(n, r) &= 0 \text{ for } r < 0 \text{ or } r > n, \\
 C(n, r) &= C(n, n - r), & C(n, r) &= C(n - 1, r - 1) + C(n - 1, r), \\
 \sum_{r=0}^n C(n, r) &= 2^n.
 \end{aligned}$$

The zero convention is used for impossible selections with nonnegative n . A program should reject invalid indices *before* indexing a table. The same value convention does not authorize taking factorials of negative numbers.

Notation is not an implementation

The factorial formula proves what $C(n, r)$ means. Pascal addition, cancellation, or a bounded factorial calculation may compute it. Select the method from the bounds, not from the shortest written formula.

Common applications

Model	Formula and interpretation
Exactly k from a group of a , in a team of r from two disjoint groups	$C(a, k)C(b, r - k)$. Pick the two components of the team.
At least one member of a distinguished group	$C(a + b, r) - C(b, r)$. Remove teams chosen entirely from the other group.
Exactly k ones in an n -bit string	$C(n, k)$. Choose the positions carrying 1s.
A monotone path with d moves of one type and e of the other	$C(d + e, d)$. Choose which steps are the first type.
Equal-valued unordered index pairs	$\sum_v C(f_v, 2)$. Group pairs by their common value.
Rearrangements with repeated symbols	$n!/(f_1! \cdots f_t!)$. Symbol copies are indistinguishable.
Identical objects in labelled boxes	$C(S + K - 1, K - 1)$ for $K \geq 1$ and nonnegative contents.

Inclusion-exclusion formulas

For two finite sets, with intersections counted inclusively:

$$|A \cup B| = |A| + |B| - |A \cap B|.$$

For three sets, write $S_1 = |A| + |B| + |C|$, $S_2 = |A \cap B| + |A \cap C| + |B \cap C|$, and $S_3 = |A \cap B \cap C|$. Then:

$$\begin{aligned} \text{union} &= S_1 - S_2 + S_3, \\ \text{exactly one} &= S_1 - 2S_2 + 3S_3, \\ \text{exactly two} &= S_2 - 3S_3, \\ \text{at least two} &= S_2 - 2S_3. \end{aligned}$$

For k bad events $A_1, \dots, A_k$, the number avoiding every bad event is

$$\sum_{S \subseteq \{1, \dots, k\}} (-1)^{|S|} \left| \bigcap_{i \in S} A_i \right|.$$

The empty-subset intersection denotes the entire universe. A union instead omits the empty subset and gives positive signs to odd subset sizes.

Choose an implementation from its contract

Method	Bounds used here	Work and storage
Factorial multiplication	$0 \leq n \leq 20$	$O(n + 1)$ time; $O(1)$ extra storage.
Direct falling product for $P(n, r)$	$0 \leq n \leq 20$	$O(r + 1)$ time; $O(1)$ extra storage; guard impossible r .
Full Pascal table	Exact $N \leq 60$	$O(N^2 + 1)$ preprocessing and storage; $O(1)$ per query.
Truncated Pascal row for one coefficient	Exact $n \leq 60$	$O(nt)$ time and $O(t)$ storage for $t = \min(r, n - r) > 0$; edge cases return in $O(1)$.
Modular Pascal table	$N \leq 1000, 1 \leq M \leq 10^9$	$O(N^2 + 1)$ preprocessing and storage; $O(1)$ per query. No division.
Exact cancellation helper	$0 \leq n \leq 66$	$O((t + 1) \log(n + 1))$ is a loose GCD-based bound for $n \geq 1$; $O(1)$ extra storage.
Masked divisibility union	$k \leq 15, N, D \leq 10^9$	$O(2^k k \log(D + 1) + 1)$ conventional bound; $O(k)$ stored input, $O(1)$ extra state.

Small arithmetic checklist

1. **Promote before multiplying.** A wide destination does not widen an earlier narrow multiplication. Use an appropriately typed operand such as `1LL` or read the operands into `long long`.
2. **Bound intermediate values.** Factorials, products, partial sums, and signed inclusion-exclusion accumulators can exceed the final answer.
3. **Divide only when exact.** Integer division discards a fractional part. A cancellation helper proves divisibility before dividing; it does not simply change the order of arbitrary operations.
4. **Keep the problem's arithmetic.** Exact integers and residues modulo M are different representations. Ordinary division of residues does not implement division before taking a remainder.
5. **Guard the domain.** Handle $r < 0$, $r > n$, empty inputs, zero positions, and an empty event collection according to the stated model.

Glossary: the language of a counting argument

Outcome

One final object counted by the problem, such as an unranked team, an ordered word, or a path.

Distinct objects

Objects with different identities. Two people with the same score are still two people unless the problem identifies them.

Disjoint cases

Cases with no shared outcome. Their counts can be added without double counting.

Continuation count

The number of legal next choices after a partial construction. Uniform continuation counts support a product; varying counts may require a sum of products.

Uniform overcount

Every final outcome has the same number of constructions. Dividing by that number is justified only after proving the repetition is uniform.

Permutation

An ordered selection of distinct objects without reuse, or a complete arrangement when all objects are used.

Combination

A selection of distinct objects whose internal order is ignored. A binomial coefficient counts selections of a fixed size.

Complement

The outcomes in a specified universe that do not satisfy a named condition.

Intersection

Outcomes satisfying every condition in a chosen collection. An intersection of zero bad events is the entire universe in the complement version of inclusion-exclusion.

Inclusion-exclusion

An alternating sum of intersection sizes that corrects repeated counting caused by overlapping sets.

Bijection

A reversible correspondence: every object on one side matches exactly one on the other. It allows the two families to be counted in the same way.

Exact arithmetic versus modular arithmetic

Exact arithmetic stores the count itself; modular arithmetic stores its remainder. Both need valid intermediate operations for their chosen representation.

Readiness checklist

Before moving on, you should be able to explain these ideas without reading a template:

1. Define one outcome and decide whether order and reuse matter.
2. Explain why disjoint cases add and why fixed continuation counts multiply.
3. Diagnose a witness-based construction that counts some outcomes more often than others.
4. Derive $0! = 1$, $P(n, r)$, and $C(n, r)$ from choices rather than memorized syntax.
5. Prove Pascal's identity by deciding whether one distinguished object is selected.
6. Implement both a full coefficient table and a correctly updated rolling row.
7. Turn a team, bit-string, path, or pair task into a well-defined selection count.
8. Derive two-set and three-set inclusion-exclusion, including the triple correction.
9. State the sign and intersection meaning of one event mask in general inclusion-exclusion.
10. Check exact integer bounds and distinguish modular addition from invalid modular division.

A practical order for revision

First reading: outcome identity, addition, multiplication, complements, factorials, permutations, combinations, and Pascal's triangle. Attempt Practices 1 to 10.

Second stage: grid paths, pair contributions, two-set and three-set regions, and general inclusion-exclusion. Attempt Practices 11 to 17, then the modelling drills.

Optional return: exact cancellation, indistinguishable copies, stars and bars, upper bounds, and derangements. Attempt Practices 18 to 20 after their derivations feel comfortable. These are extensions, not prerequisites for the core questions.

The strongest small test

Generate all outcomes for a tiny input using the recursion or masks from earlier sessions. Compare their count with your formula. When the counts differ, inspect which individual outcome is missing or repeated; do not only adjust the arithmetic to match the sample.

Practice plan and personal revision notes

This page is for planning and recording your own progress. Session dates and an external practice-sheet link are intentionally left blank: they depend on the training group's schedule.

Practice-sheet link

Start date

End date

Review / contest date

For each problem, record the model before the formula

One outcome is:

Order and repetition rules:

My disjoint cases, stages, or bad events:

Why the formula counts each valid outcome once:

Largest intermediate value and chosen type:

Small counterexample or boundary to test:

A problem to retry without the solution:

4.39 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.

These notes document assumptions behind the programs. They support a second reading; they are not extra formulas to memorize before attempting the core practice.

Language version and integer representation

Every complete Session 4 program uses standard C++23 headers. The language references below use the fixed N4950 C++23 working draft rather than a moving draft that may describe later language changes. The tested GNU environment has a 32-bit `int` and a 64-bit `long long`; C++ does not require `int` to have exactly 32 bits. The standard guarantees at least the signed range needed for the `long long` bounds used here. Signed arithmetic that overflows is not a valid way to obtain a large count.^{[4-9][4-10]}

All tasks state valid input and numeric contracts. The programs are contest solutions, not interactive validators, and print no prompts. A generic helper name does not enlarge its contract: the factorial-based methods stop at 20; ordinary exact Pascal examples stop at 60; the explicitly optional cancellation example extends to 66.

Why those exact-count limits are safe

$$20! = 2432902008176640000,$$

$$C(60, 30) = 118264581564861424,$$

$$C(66, 33) = 7219428434016265740.$$

Each fits a 64-bit signed integer. The first value bounds factorial examples. Every coefficient through row 60 is bounded by that row's maximum, and all row sums through 60 are at most 2^{60} . Likewise all coefficients through row 66 fit, but *their sum* 2^{66} does not. A safe coefficient table is not permission to sum the whole row into the same type.

Products representing disjoint team components or path segments are also bounded by the number of corresponding unconstrained teams or paths. The signed inclusion-exclusion examples separately bound absolute term sums, rather than assuming that a small final answer guarantees safe intermediate arithmetic.

Intersections and capped least common multiples

For positive divisors a, b , their common-multiple intersection uses $a b / \gcd(a, b)$, with cancellation performed before the multiplication. The code first divides one factor by the GCD, then checks against the remaining cap. It never constructs a value above N just to discover that its contribution is zero.

The standard `std::gcd` helper is introduced locally; full Euclidean-algorithm study belongs to number theory. Calling `std::lcm` does not itself provide a checked-overflow operation: its contract requires the absolute inputs and resulting least common multiple to be representable.^{[4-11][4-13]}

Modular arithmetic is deliberately introduced without inverses

Pascal's recurrence uses only addition, so reducing each entry works for every positive modulus in the specified range, including composite moduli and $M = 1$. Values are held in `long long`, and each addition of two reduced entries is below $2M$. Modular inverse methods, factorial inverses, and their extra preconditions are left for the later number theory session. Ordinary division of factorial residues is not a substitute.

Repeated-squaring exponentiation is also not required here. Small exponents use a loop with a product initialized to 1. The convention $0^0 = 1$ is used only as the empty-assignment count needed by the stated word and inclusion-exclusion models.

Original program verification

The source package contains **24 complete examples and 20 complete practice solutions** for Session 4. All **44 programs** compiled with **GCC 14.2.0** in C++23 mode and passed **15,862 automated input/output checks**. One check means one program execution; a batch of many queries is not counted as many separate executions.

All printed program samples are checked from the same manifest used to generate the sample tables. Additional checks compare outputs with Python exact-integer calculations, small exhaustive selections, direct divisibility scans, independently constructed set regions, path-counting dynamic programs, and other problem-specific reference methods. Tests include empty outcomes, impossible selections, duplicate conditions, zero positions, maximum stated bounds, every coefficient through row 66, and composite-modulus cases.

The original content edition also reran the other four topic-level suites. Across Sessions 1 to 4, **220 complete programs passed 63,919 input/output checks**. The earlier Session 2 state-restoration probes and Session 3 utility assertions passed as separate checks; they are not included in that input/output total. The earlier 176 C++ program files were preserved unchanged.

Checking configuration

```
-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 detect some faults along executed paths; a passing suite is not a mathematical proof for every valid input. The correctness explanations and numeric contracts are still necessary. Deliberately incorrect teaching fragments are labelled and excluded from the complete-program verification.^[4-14]

Rebuild, verify, or edit

From the extracted project directory, build the standalone handout with `python3 build.py --target session4`, or the collected edition with `python3 build.py --target book`. The build script runs pdfLaTeX three times for a resolved table of contents and cross-references. A normal LaTeX installation with the listed packages is sufficient; shell escape is not required.

Run `python3 tests/verify_counting_principles.py` with a suitable GNU C++23 compiler available. Set `CXX=g++-14` where that compiler is installed under a versioned name. The README documents the four completed sessions and five topic-level verification suites. JSON reports record the compiler, flags, checks, and source hashes.

The individual handout and the collected book include the same chapter and C++ files. Change a complete program once, rerun its verification, and rebuild both editions. If a program's sample contract changes, update the manifest and sample table together. Every handout and the collected introduction now use the same six-session roadmap; this edition contains the first four sessions.

How references are used

A superscript such as ^[4-3] links to a source below. The first number identifies this session so its references do not collide with earlier chapters. Mathematical references are author-written textbooks or university course materials; C++ references are fixed committee wording and official compiler documentation. The chapter's teaching examples, exercises, traces, and test programs are written for this training.

[4-1] Oscar Levin: Combining Outcomes

https://discrete.openmathbooks.org/dmoi4/sec_counting-combine-outcomes.html

[4-2] MIT OpenCourseWare: Counting and Sets, 18.05 course notes

https://ocw.mit.edu/courses/18-05-introduction-to-probability-and-statistics-spring-2022/mit18_05_s22_class01-prep-b.pdf

[4-3] Oscar Levin: Combinations and Permutations

https://discrete.openmathbooks.org/dmoi4/sec_counting-combperm.html

[4-4] Oscar Levin: Non-Disjoint Outcomes

https://discrete.openmathbooks.org/dmoi4/sec_counting-non-disjoint.html

Mathematical and implementation references, continued**[4-5] Oscar Levin: Pascal's Arithmetical Triangle**

https://discrete.openmathbooks.org/dmoi4/sec_counting-pascal.html

[4-6] Oscar Levin: Combinatorial Proofs

https://discrete.openmathbooks.org/dmoi4/sec_comb-proofs.html

[4-7] Andrew Brandt, Western University: Discrete Structures, Counting

<https://www.csd.uwo.ca/~abrandt5/teaching/DiscreteStructures/Chapter6/counting.html>

[4-8] Oscar Levin: Advanced Counting Using PIE

https://discrete.openmathbooks.org/dmoi4/sec_advPIE.html

[4-9] C++23 N4950: Fundamental Types

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

[4-10] C++23 N4950: Multiplicative Operators and Integer Division

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

[4-11] C++23 N4950: Greatest Common Divisor

<https://timsong-cpp.github.io/cppwp/n4950/numeric.ops.gcd>

[4-12] Oscar Levin: Counting Multisets

https://discrete.openmathbooks.org/dmoi4/sec_counting-multisets.html

[4-13] C++23 N4950: Least Common Multiple Preconditions

<https://timsong-cpp.github.io/cppwp/n4950/numeric.ops.lcm>

[4-14] GCC: Instrumentation Options and Undefined-Behavior Sanitization

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