

Level 1 Training

Session 2

Recursion and Backtracking

Smaller problems, decision trees,
and systematic search

Define one call. Trace its return.
Choose, explore, and restore the state.

How to use this reference

Level 1 is about choosing an algorithm and explaining why it works. This book is a reference to study **after the session**. Read one idea, trace a small example, and write your own version before attempting the practice.

A study routine for recursive thinking

1. **Describe one call.** Complete the sentence “This function solves ...” using its parameters. Do not begin by trying to hold the entire call tree in your head.
2. **Find the stopping point.** Identify the smallest valid case and the quantity that gets closer to it on every recursive step.
3. **Trace entry and return.** Write each call’s parameters, the line it will resume at, and its returned value. For backtracking, also record the shared state before and after a choice.
4. **Test and rebuild.** Try an empty choice, repeated values, impossible answers, and a case that forces more than one branch. Use hints before solutions, then rewrite the idea without looking.

How the material is labelled

A **complete example** or **complete solution** has its own headers and `main`. The printed listing comes from the supplied `.cpp` file. Compile one file at a time. A **fragment** illustrates one detail; its placement is stated. Deliberately wrong fragments are not templates to copy.

Pause and predict questions have 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.

Two different promises

A recursive function must make progress towards its base case. A backtracking helper that shares a path or board must also respect its *state-restoration contract*. Returning from a function does not automatically undo changes to an object passed by reference.

Language and prerequisites

The complete programs use **standard C++23**, with no GNU extension headers. You should know functions, references, vectors, strings, loops, sorting, and basic Big-O notation. The binary-search example connects to Session 1, Part B; the optional merge-sort example reuses merging from Session 1, Part A. Bitmasks, formal combinatorial formulas, graphs, and dynamic programming are not prerequisites.

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 2: Recursion and Backtracking**. 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 2 will help you do

Design a recursive helper with a precise meaning. Trace calls and returns without confusing local variables with shared data. Generate choices systematically, restore a path or board, avoid duplicate outputs, prune only impossible branches, and distinguish finding one answer from counting or printing all answers.

Conventions

Programs assume valid input within the stated bounds and print no prompts. Array indices are zero-based; printed selections use one-based labels when stated. Text examples use ASCII characters. Enumeration is deliberately limited to small inputs because output can itself be exponential.

EMPTY represents one valid empty sequence in enumeration tasks. NONE means no solution exists. They are different outcomes. Numerical tasks use their specified numeric answer instead. Complexity includes output work where relevant and counts the maximum *simultaneously active* calls for stack space.

Contents

How to use this reference	i
The Level 1 learning path	ii
2 Recursion and Backtracking	1
2.1 Recursion starts with the meaning of one call	2
2.2 Base cases and progress: two separate obligations	3
2.3 Tracing calls: descent, suspension, and return	4
2.4 Returning an answer: factorial and the recursive promise	6
2.5 Parameters, local variables, and shared state	8
2.6 Recursive arrays: an index describes the remaining work	10
2.7 Digit recursion: shrink the number by division	12
2.8 Recursive strings without copying substrings	13
2.9 Recursive binary search: reuse an earlier invariant	15
2.10 Branching calls: total work is not the same as depth	16
2.11 Split into halves: call count and work per level	18
2.12 Practical limits: recursion depth and numeric bounds	20
2.13 From a smaller problem to a tree of decisions	21
2.14 Generate binary strings: choose, explore, and undo	22
2.15 Subsets: make a take-or-skip decision for each position	24
2.16 Existence, a witness, a count, or every answer?	26
2.17 Subset-sum existence: keep negative values valid	27
2.18 Counting subsets: add disjoint branches and keep zero choices	29
2.19 Pruning: reject only branches that cannot succeed	31
2.20 Combinations: increasing choices prevent reordering duplicates	33
2.21 Permutations: order matters and a used table prevents reuse	35
2.22 Duplicate values: remove equivalent choices at the same depth	37
2.23 Balanced parentheses: reject invalid prefixes early	39
2.24 N-queens: encode conflicts so each choice is easy to check	41
2.25 Grid routes with only down and right moves	43
2.26 Four-direction simple paths: visited means on this path	45
2.27 Return one witness without leaving temporary state behind	47
2.28 Debugging a recursive or backtracking solution	49
2.29 Optional extension: Towers of Hanoi and parameter roles	50
2.30 Optional extension: memoization reuses repeated answers	52
2.31 Optional extension: merge sort combines recursion with two pointers	54
2.32 Practice problems: describe the call before writing it	56
2.33 Trace, correctness, and complexity drills	66
2.34 Hints before solutions	72
2.35 Explained practice solutions	78
2.36 Explained trace and complexity drills	98
2.37 Quick reference, glossary, and readiness checklist	104
2.38 Technical notes, verification, and references	109

SESSION 2

Recursion and Backtracking

Define one call. Trace its return. Choose, explore, and restore the state.

In the previous sessions, you eliminated unnecessary candidates using order. Sometimes a problem is better expressed as a smaller version of itself. Sometimes you must explore several alternatives because one local decision is not enough. Recursion helps express both kinds of work.

The goal is not to replace every loop with a function call. It is to describe a problem clearly, reduce it safely, and control how partial choices are explored. A short recursive program can still take enormous time, so correctness and cost must be considered together.

Three related ideas

Idea	Question	Example
Recursion	How does one smaller answer help solve this call?	Sum the remaining elements of an array.
Divide and conquer	Can separate smaller parts be solved and combined?	Sort two halves, then merge them.
Backtracking	Which next choices can lead to a complete valid answer?	Build a permutation or place nonattacking queens.

Your route through the chapter

Stage A: understand calls. Learn contracts, base cases, progress, call stacks, return values, local and shared state, and recursive time and space.

Stage B: explore choices. Generate binary strings, subsets, combinations, and permutations. Decide whether the task asks for existence, a witness, a count, or every answer.

Stage C: enforce constraints. Prune impossible prefixes, handle duplicates, generate balanced parentheses, place queens, and explore small grids without reusing cells.

Stage D: practise and explain. Trace state restoration, solve the coding problems and drills, and test cases that expose an incorrect assumption. Optional sections introduce Hanoi, memoization, and merge sort.

The sentence to write before the first recursive call

“Given these parameters, this call will ...; when it returns, the shared state will ...” A precise answer to those two blanks prevents more mistakes than memorizing a generic recursive template.

2.1 Recursion starts with the meaning of one call

A function can use a smaller instance of itself

A recursive function calls itself, directly or through other functions. C++23 permits recursive calls to ordinary functions, but not to `main`. We will put the recursive work in a named helper and call it from `main`.^[2-1]

Suppose we want to print the integers from n down to 1, one per line. Instead of thinking about every number at once, separate the job into two parts: print n , then print the integers from $n - 1$ down to 1. The second part is the same job with a smaller input.

The **contract** of `countdown(n)` is: for $n \geq 0$, print $n, n - 1, \dots, 1$ in that order; if $n = 0$, print nothing. The base case is not an error. It describes a job with no remaining work.

Complete example | C++23

```
#include <iostream>
void countdown(int n) {
    if (n == 0) return;
    std::cout << n << '\n';
    countdown(n - 1);
}
int main() {
    int n;
    std::cin >> n;
    countdown(n);
}
```

Sample input

3

Sample output

3
2
1

Input contract. One integer $0 \leq n \leq 1000$. At $n = 0$, this particular example produces no output. Later enumeration tasks use **EMPTY** for a different reason: to represent one valid empty answer.

Why it works. At $n = 0$ there is nothing to print. For $n > 0$, print the first required number and let the smaller call print the rest. If that smaller call fulfills its contract, the combined output fulfills this call's contract.

Cost. There are $n + 1$ calls including the base case, so $O(n + 1)$ time and $O(n + 1)$ active-call space in the ordinary call-stack model. This is a teaching example; a loop would print the same sequence with constant auxiliary space.

Pause and predict: 1

What does `countdown(0)` print, and how many helper calls does that execution contain?

Check your prediction. It prints nothing, but the initial call still exists and immediately returns. There is one call, not zero.

2.2 Base cases and progress: two separate obligations

A base case must be reached, not merely written

A **base case** is solved without another recursive call. A **recursive case** reduces the task and uses a smaller result. A correct design also identifies a **progress measure**: a nonnegative integer that strictly decreases along each continuing branch.

For `countdown(n)`, the measure is n : it decreases to zero. For an array helper at index i , use $n - i$. Increasing an index can still make the unfinished problem smaller.

State	Progress measure	Typical stopping condition
$f(n)$ for $n \geq 0$	n	$n == 0$
<code>visit(i)</code> over n elements	$n - i$	$i == n$
<code>search(lo, hi)</code> on $[lo, hi)$	$hi - lo$	$lo == hi$
<code>build(path)</code> of length n	$n - path $	Path has length n .

Two deliberately wrong helper fragments

Both fragments below are incorrect for arbitrary positive inputs. They illustrate different failures, not templates to copy.

```
void wrong(int n) {
    if (n == 0) return;
    wrong(n);          // Same state: no progress.
}

void wrongStep(int n) {
    if (n == 0) return;
    wrongStep(n - 2);  // Starting at 3 skips zero.
}
```

The first repeats the same positive value forever. The second starts $3, 1, -1, -3, \dots$: it moves but never reaches zero. A base case written in the code is not enough.

For the second helper, the real task may require even nonnegative input or a stopping rule that correctly handles an overshoot. Do not blindly replace every base with $n \leq 0$; its returned value must still mean the right thing.

A small correctness proof. Check three things: the base answer is correct; every child satisfies its input contract and makes progress; assuming the smaller calls are correct, their results complete this call correctly. This is the practical form of proof by induction. You need not expand every call.

Pause and predict: 2

A function starts at $i = 0$, calls itself with $i + 1$, and stops at $i == n$. What becomes smaller?

Check your prediction. The number of unprocessed elements, $n - i$. The argument need not decrease numerically; the unfinished work must decrease.

2.3 Tracing calls: descent, suspension, and return

The caller waits at its own call site

When a helper calls itself, the current invocation is not replaced. Think of it as paused, with a reminder of where to continue. The child invocation has its own parameters. When the child returns, execution resumes in the caller. A **return** transfers control to that caller, not automatically to `main`.^[2-2]

A **call stack** is a useful model for those unfinished invocations. A **frame** records one invocation's local state and continuation. We use the model for tracing and space analysis; compilers may optimize its physical implementation.

Complete example | C++23

```
#include <iostream>
void trace(int n) {
    if (n == 0) {
        std::cout << "base\n";
        return;
    }
    std::cout << "enter " << n << '\n';
    trace(n - 1);
    std::cout << "leave " << n << '\n';
}
int main() {
    int n;
    std::cin >> n;
    trace(n);
}
```

Sample input

2

Sample output

```
enter 2
enter 1
base
leave 1
leave 2
```

Input contract. One integer $0 \leq n \leq 20$. The words in this output are intentional: this program is a trace demonstration, not the output format for an unrelated contest task.

The first print occurs on the way *down*, before the recursive call. The second print occurs on the way *back*, after the child has finished. Those are different moments even though both prints use the variable named `n`.

Follow every event for trace(2)

Step	Active helper frames	Action
1	[2]	Print enter 2. The frame with $n = 2$ requests trace (1).
2	[2, 1]	Print enter 1. The frame with $n = 1$ requests trace (0).
3	[2, 1, 0]	Print base . Return from the $n = 0$ call.
4	[2, 1]	Resume the $n = 1$ frame after its call. Print leave 1, then finish it.
5	[2]	Resume the $n = 2$ frame. Print leave 2, then finish it.
6	[]	The helper has finished. Execution continues in main .

The frames are shown from oldest to newest. At the deepest point, three helper calls are alive together. Only the newest one is executing; its ancestors are waiting. The frame in which n is 2 still has 2 after its child returns.

Return order is not a second set of recursive calls

The **leave** lines do not create more invocations. They are work performed by the existing callers as execution unwinds. For input n , there are $n + 1$ invocations and $2n + 1$ output lines, not $2n + 1$ invocations.

A tracing technique you can use on paper

Give repeated calls an identity such as $f(3)$, $f(2)$, and $f(1)$. Beside each, write its local variables and the next line it must execute after a child finishes. Mark a return only when that invocation actually ends.

If two calls have the same argument, give them separate labels. The second $f(2)$ in a branching computation is another invocation; it does not automatically reuse the first result.

Pause and predict: 3

For **trace**(3), in what order are the **leave** lines printed?

Check your prediction. **leave** 1, then **leave** 2, then **leave** 3. The deepest unfinished positive call resumes first.

2.4 Returning an answer: factorial and the recursive promise

Printing is not returning

A function that returns a number gives that number to its caller, which can store, multiply, or print it. Printing a number only writes text to an output stream. A value-returning helper needs a return value on every reachable normal path; falling off the end of such a helper is not a substitute.^[2-2]

For a nonnegative integer n , the **factorial** $n!$ is the product $1 \times 2 \times \cdots \times n$. The empty product is defined as $0! = 1$. Thus, for $n > 0$:

$$n! = n \times (n - 1)!$$

The recursive promise is precise: `factorial(n)` returns $n!$ for $0 \leq n \leq 20$.

Complete example | C++23

```
#include <iostream>
long long factorial(int n) {
    if (n == 0) return 1;
    long long smaller = factorial(n - 1);
    return n * smaller;
}
int main() {
    int n;
    std::cin >> n;
    std::cout << factorial(n) << '\n';
}
```

Sample input

5

Sample output

120

Input contract. $0 \leq n \leq 20$. On the tested 64-bit long long environment, $20! = 2432902008176640000$ fits, while $21!$ does not. The bound is part of the algorithm's numeric contract. A base case prevents endless recursion; it does not prevent overflow.

Why it works. The base returns the empty product. For $n > 0$, the smaller call returns the product through $n - 1$, and multiplying by n adds the last factor. Progress is by $n - 1$.

Cost. $n + 1$ calls, $O(n + 1)$ time and active-call space. An iterative product uses the same number of multiplications but only constant additional space.

Values return upwards

For `factorial(4)`, the calls descend before any of the pending multiplications can finish:

```
factorial(4) waits for 4 * factorial(3)
factorial(3) waits for 3 * factorial(2)
factorial(2) waits for 2 * factorial(1)
factorial(1) waits for 1 * factorial(0)
factorial(0) returns 1
```

Now each waiting call receives a concrete number:

Resuming call	Value from its child	Value it returns
<code>factorial(1)</code>	1	$1 \times 1 = 1$
<code>factorial(2)</code>	1	$2 \times 1 = 2$
<code>factorial(3)</code>	2	$3 \times 2 = 6$
<code>factorial(4)</code>	6	$4 \times 6 = 24$

Do not repeat a child call when you only need its value

The variable `smaller` is a stored result. Using it twice does not run the recursive function twice. Writing `factorial(n - 1)` twice does request two computations unless an optimization happens to eliminate one; do not base algorithmic reasoning on that possibility.

Wrong example: discarding the returned value. This is an intentionally incomplete helper.

```
long long wrongFactorial(int n) {
    if (n == 0) return 1;
    n * wrongFactorial(n - 1); // Computed, then discarded.
} // Missing return on this path.
```

Pause and predict: 4

What happens to every positive factorial if its base case incorrectly returns 0 instead of 1?

Check your prediction. Each waiting multiplication receives a value derived from zero, so every positive result becomes zero. The base case must be the correct answer to the smaller problem, not just a way to stop.

2.5 Parameters, local variables, and shared state

Separate calls do not share ordinary scalar parameters

A call initializes its parameters from its arguments. With an ordinary value parameter such as `int n`, changing that parameter does not change the caller's separate integer. A non-const reference parameter can instead refer to the caller's object.^[2-1]

Automatic local variables belong to the execution of their scope; they are not one permanent shared variable for every recursive invocation.^[2-3] This is why the `n=2` frame in the trace still prints 2 after its `n=1` child finishes.

Classify each parameter by its role

Typical parameter	Reason for that form
<code>int i, long long remaining</code>	Small per-call values. Each child receives a separate value.
<code>const std::vector<int>& a</code>	Shared read-only input. Avoid copying the whole array at each call.
<code>std::string& path</code>	Shared mutable partial answer. Changes may need to be undone.
<code>std::vector<int>& answer</code>	An output object intentionally changed when a result is found.

Returning does not reverse reference mutations

Consider this fragment, which belongs inside a backtracking helper that receives `path` by reference:

```
path.push_back('A');
exploreNext(path);
path.pop_back();
```

The pop removes the choice made by this frame, assuming the child restores the state it received. Without the pop, the caller still sees the appended 'A'. The reference refers to the same string, not to a private child copy.

By contrast, passing `remaining` - value into a value parameter does not require adding value back after the child returns. You never changed this frame's `remaining` in the first place.

Pause and predict: 5

A child appends a character to a string passed by reference and returns without removing it. Does returning restore the old string?

Check your prediction. No. The string is shared. Either the child must restore it under its contract, or the caller must deliberately handle the changed state. Ordinary function return is not an undo operation.

Copies can simplify reasoning, but they have a cost

A helper receiving `std::string path` by value has a private string object. Changes to that copy do not require undoing the parent's string. However, copying a length- k string can take $O(k)$ work and storage.^[2-4]

If every level of a depth- n chain keeps a separate prefix of length $0, 1, 2, \dots, n$, the total characters alive together can be

$$0 + 1 + 2 + \dots + n = \frac{n(n+1)}{2} = O(n^2).$$

A shared path plus append/pop needs only one length- n path and $O(n)$ small frames. It requires more careful restoration, but avoids that collection of simultaneous copies.

Similarly, passing a complete length- n input vector by value at each of n levels can turn a simple linear scan into quadratic copying work and storage. Our read-only sequence helpers therefore use `const` references.

Do not mutate the caller accidentally while forming an argument

Wrong example: confusing an argument with a state update.

```
// Inside a recursive helper with a value parameter n:  
wrong(n--); // Passes the OLD n, then changes this frame's n.
```

The child receives the old value, so a positive countdown does not make progress. Prefer `helper(n - 1)`. It clearly passes the smaller value without modifying the caller's local variable.

Shared counters and repeated test cases

A global or `static` counter persists beyond a single ordinary invocation. That can be intentional, but it is easy to forget to reset it between independent tasks. Prefer a fresh counter in `main` or `solve`, then pass it by reference only when the helper really needs to update it.

Do not return a reference to a local path that will cease to exist. Return an independent value or copy the successful path into an output object whose lifetime belongs to the caller.

2.6 Recursive arrays: an index describes the remaining work

The suffix contract

For a vector of length n , define `suffixSum(a, i)` as the sum of elements at indices $i, i+1, \dots, n-1$. Its valid index contract is $0 \leq i \leq n$. At $i = n$, the suffix is empty and has sum zero.

$$S(i) = \begin{cases} 0, & i = n, \\ a[i] + S(i+1), & 0 \leq i < n. \end{cases}$$

Passing the index is enough to describe the suffix. We do not create another vector containing that suffix.

Complete example | C++23

```
#include <iostream>
#include <vector>
long long suffixSum(const std::vector<int>& a, int i) {
    if (i == static_cast<int>(a.size())) return 0;
    long long rest = suffixSum(a, i + 1);
    return a[i] + rest;
}
int main() {
    int n;
    std::cin >> n;
    std::vector<int> a(n);
    for (int& x : a) std::cin >> x;
    std::cout << suffixSum(a, 0) << '\n';
}
```

Sample input

4
8 -3 0 5

Sample output

10

Input contract. $0 \leq n \leq 1000$, followed by n integers with $|a[i]| \leq 10^9$. The total can exceed `int`, so the result and `rest` use `long long`.

Why it works. The empty suffix contributes zero. Every nonempty suffix consists of its first element plus the remaining suffix. Increasing i decreases the number of elements left.

Cost. $n+1$ calls, $O(n+1)$ time, $O(n)$ stored input, and $O(n+1)$ additional active-call space. There is no sequence copying. A loop is the more practical implementation for very long linear scans.

Pause and predict: 6

Why must the `i == a.size()` check occur before reading `a[i]`?

Check your prediction. At the empty-suffix base case, i is one past the final valid element. Returning zero is valid; reading that element is not.

Trace a suffix, not a changing copy of the array

For $a = [8, -3, 0, 5]$:

i	Meaning of this call	Returned result
4	Sum of an empty suffix.	0
3	Sum of [5].	$5 + 0 = 5$
2	Sum of [0, 5].	$0 + 5 = 5$
1	Sum of [-3, 0, 5].	$-3 + 5 = 2$
0	Sum of [8, -3, 0, 5].	$8 + 2 = 10$

The rows are shown in return order. All calls read the same vector. What changes is the boundary describing their task.

Changing the operation changes the base case

For a sum, an empty suffix has a natural answer of zero. For a maximum, returning zero would be wrong when all remaining values are negative. One simple maximum contract requires a nonempty suffix and stops at the final actual element, returning that value. Practice 4 uses this design.

A recursive search can return an index or a boolean. Be explicit about what an absent answer means. A return value of zero may be a valid index, so it is not a suitable absence marker unless the output convention makes that distinction elsewhere.

Do not lose linear time through hidden work

A helper that calls itself on a by `const` reference performs one constant-sized combination per element. Repeatedly constructing a new suffix vector would also copy data. Likewise, recomputing a suffix sum twice instead of storing the returned result can multiply work across levels.

The right question is not “How many lines are inside the helper?” It is “How many calls run, and how much work does each one perform?”

2.7 Digit recursion: shrink the number by division

For a nonnegative decimal integer, $x \% 10$ extracts the last digit and $x / 10$ removes it. If $x < 10$, there is only one digit left. That includes zero: the ordinary decimal representation of 0 contains one digit, '0'.

Complete example | C++23

```
#include <iostream>
long long digitSum(long long x) {
    if (x < 10) return x;
    return x % 10 + digitSum(x / 10);
}
int main() {
    long long x;
    std::cin >> x;
    std::cout << digitSum(x) << '\n';
}
```

Sample input

Sample output

5070

12

Input contract. $0 \leq x \leq 10^{18}$. We process the ordinary numeric representation, not a string with preserved leading zeros. Negative values are outside this helper's contract.

Trace. For 5070, the recursive values are 5070, 507, 50, 5. The pending digits are 0, 7, 0, and the base returns 5. The result is $0 + 7 + 0 + 5 = 12$.

Why it works. A multidigit number separates into its final digit and a shorter remaining decimal representation. Integer division by 10 strictly decreases any $x \geq 10$.

Cost. With d decimal digits, there are d calls, so $O(d)$ time and space. For positive x , $d = \lfloor \log_{10} x \rfloor + 1$; zero has $d = 1$. You can reason about the number of divisions without memorizing the formula.

A useful change of task

To count occurrences of a requested digit, return 1 or 0 for the last digit and add the count from the remaining number. The one-digit base case is important: treating $x == 0$ as “no digits” would mishandle the input number zero unless it is handled separately.

Pause and predict: 7

What should `digitSum(0)` return? How many zero digits does the number 0 contain for a digit-count task?

Check your prediction. Its digit sum is 0, and it contains one zero digit. A sum of zero is not evidence that the representation has no digits.

2.8 Recursive strings without copying substrings

A palindrome reduces from both ends

A palindrome reads the same forwards and backwards. If the outer characters differ, the answer is immediately false. If they match, only the interior remains to be checked. An empty interval or a single character is already a palindrome.

The contract of `palindrome(s, left, right)` refers to the closed interval $[left, right]$. The initial call uses $left = 0$ and $right = n - 1$, with a signed length so that an empty input produces $right = -1$ safely.

Complete example | C++23

```
#include <iostream>
#include <string>
bool palindrome(const std::string& s, int left, int right) {
    if (left >= right) return true;
    if (s[left] != s[right]) return false;
    return palindrome(s, left + 1, right - 1);
}
int main() {
    std::string s;
    std::getline(std::cin, s);
    int n = static_cast<int>(s.size());
    std::cout << (palindrome(s, 0, n - 1) ? "YES" : "NO")
               << '\n';
}
```

Sample input

level

Sample output

YES

Input contract. One ASCII line of length at most 1000, possibly empty or containing spaces. All characters count exactly as read; the check is case-sensitive. `getline` is used because a word-reading operator would stop at whitespace.

Why it works. Equal outside characters reduce the question to the interior. A mismatch disproves the whole interval. Moving both ends inward reduces the remaining interval length by two.

Cost. At most about $n/2$ comparisons, so $O(n + 1)$ time and additional active-call space in the worst case. The string is stored once. An early mismatch can finish much sooner.

Pause and predict: 8

For an empty line, does the helper read `s[0]` or `s[-1]`?

Check your prediction. Neither. The initial boundaries are 0 and -1 , so the base check succeeds before either indexed access is evaluated.

Keep boundaries instead of constructing the middle

A tempting alternative is to compare the ends and call the helper on `s.substr(1, s.size() - 2)`. That creates a new string representing the interior. Repeating that operation copies lengths roughly $n, n-2, n-4, \dots$, whose sum is quadratic.

The boundary-based helper keeps the same information using two integers. Each level does only its comparison and a constant-sized call. The original characters never move.

A tiny trace for `level`

Boundaries	Comparison	Next action
[0, 4]	<code>l == l</code>	Check [1, 3].
[1, 3]	<code>e == e</code>	Check [2, 2].
[2, 2]	No comparison needed.	Return true.

Each waiting call returns its child's boolean result. No new string is produced. Reversing a line is a different contract: it intentionally changes the string, swapping the ends before recursing into the interior. That mutation is the desired result, so it should *not* be undone afterwards.

Restoration is a contract, not a universal rule

Backtracking usually restores temporary choices. A recursive function whose purpose is to modify an array or string may intentionally keep its changes. Decide which behavior the helper promises before adding an “undo” line.

2.9 Recursive binary search: reuse an earlier invariant

The lower-bound search from Session 1, Part B finds the first index whose value is at least x , or returns n when every element is smaller. A recursive implementation can preserve the same half-open unknown interval $[lo, hi)$.

At each nonempty interval, compare the middle element. If it is smaller than x , discard it and everything before it. Otherwise, keep the boundary at or before the middle. Only *one* child call is necessary.

Complete example | C++23

```
#include <iostream>
#include <vector>
int firstAtLeast(const std::vector<int>& a, int lo, int hi,
                int x) {
    if (lo == hi) return lo;
    int mid = lo + (hi - lo) / 2;
    if (a[mid] < x) return firstAtLeast(a, mid + 1, hi, x);
    return firstAtLeast(a, lo, mid, x);
}
int main() {
    int n, x;
    std::cin >> n >> x;
    std::vector<int> a(n);
    for (int& value : a) std::cin >> value;
    std::cout << firstAtLeast(a, 0, n, x) << '\n';
}
```

Sample input

```
5 4
1 4 4 7 9
```

Sample output

```
1
```

Input contract. $0 \leq n \leq 200000$, target $|x| \leq 10^9$, and n nondecreasing integers with absolute values at most 10^9 . The result is a zero-based boundary from 0 through n , not necessarily an index containing an actual match.

Why it works. The sorted-order elimination argument is unchanged from the iterative version. The quantity $hi - lo$ strictly decreases, and when it reaches zero the two boundaries identify the required insertion position.

Cost. A search takes $O(\log(n+1) + 1)$ time and active-call space. Including reading the vector, the complete program takes $O(n+1)$ time. The iterative version performs the same search with $O(1)$ additional search space.

Pause and predict: 9

Does having two return statements in the helper mean that every call makes two recursive calls?

Check your prediction. No. Only one branch executes. A taken return exits that invocation. The recursion follows one narrowing chain, not a binary tree of both halves.

2.10 Branching calls: total work is not the same as depth

Fibonacci exposes repeated subproblems

Define $F(0) = 0$, $F(1) = 1$, and $F(n) = F(n-1) + F(n-2)$ for $n \geq 2$. Translating that definition directly creates two child computations. The following example also counts every helper invocation, including base cases.

Complete example | C++23

```
#include <iostream>
long long fibonacci(int n, long long& calls) {
    ++calls;
    if (n <= 1) return n;
    long long left = fibonacci(n - 1, calls);
    long long right = fibonacci(n - 2, calls);
    return left + right;
}
int main() {
    int n;
    std::cin >> n;
    long long calls = 0;
    long long answer = fibonacci(n, calls);
    std::cout << answer << ' ' << calls << '\n';
}
```

Sample input

5

Sample output

5 15

Input contract. $0 \leq n \leq 25$. Output the Fibonacci value followed by the number of helper calls. The deliberately small bound keeps this inefficient demonstration manageable.

The two child calls appear in separate statements so the teaching trace has an explicit left-then-right order. Do not assume that both operands of an arbitrary addition expression are evaluated left to right.

Why it works. The base values follow the definition; the nonbase result adds the two required smaller values. Both arguments decrease. Correctness does not make the repeated computation efficient.

A recursion tree is a record of calls, not shared storage

For $F(4)$, separate branches request $F(2)$ more than once:

```

F(4)
  F(3)
    F(2)
      F(1)
      F(0)
    F(1)
  F(2)
    F(1)
    F(0)

```

Those two $F(2)$ invocations perform their work independently. Unless we explicitly cache an answer, the earlier result is not automatically available to the later call.

n	0	1	2	3	4
$F(n)$	0	1	1	2	3
Calls	1	1	3	5	9

For $n \geq 2$, the call count satisfies $C(n) = 1 + C(n-1) + C(n-2)$. The extra 1 is the current invocation. This count grows exponentially; $O(2^n)$ is a valid, deliberately simple upper bound.

Why the stack is only linear

Along one active chain, arguments decrease by at least one, so the maximum active depth is $O(n+1)$. The left child finishes before the right child starts. The complete exponential tree is *not* stored on the call stack at once.

Cost. Exponential time, bounded above by $O(2^n)$; $O(n+1)$ additional active-call space. The optional memoization section changes the algorithm by storing repeated answers.

Pause and predict: 10

A depth- n binary recursion tree can contain exponentially many calls. Must it also use exponential call-stack space?

Check your prediction. No. Depth-first recursive execution keeps only one root-to-current-call chain active, plus its waiting ancestors. With constant-sized frames and no stored outputs or large copies, that space is $O(n)$.

2.11 Split into halves: call count and work per level

Two half-size calls can still have linear total work

The vector sum can also split a half-open range into two disjoint halves. Stop at an empty range or one element. Otherwise, add the sums of the two halves. This version reduces maximum depth without avoiding any input element.

Complete example | C++23

```
#include <iostream>
#include <vector>
long long rangeSum(const std::vector<int>& a, int lo, int hi) {
    if (lo == hi) return 0;
    if (hi - lo == 1) return a[lo];
    int mid = lo + (hi - lo) / 2;
    long long left = rangeSum(a, lo, mid);
    long long right = rangeSum(a, mid, hi);
    return left + right;
}
int main() {
    int n;
    std::cin >> n;
    std::vector<int> a(n);
    for (int& x : a) std::cin >> x;
    std::cout << rangeSum(a, 0, n) << '\n';
}
```

Sample input

5
3 -1 4 0 2

Sample output

8

Input contract. $0 \leq n \leq 200000$ and $|a[i]| \leq 10^9$. All sums fit `long long` under these bounds.

Why it works. The two half-open ranges partition $[lo, hi)$ without a gap or overlap. Every value contributes once. For a range longer than one element, both children are strictly smaller.

Cost. For nonempty input, there are n one-element leaves and $n - 1$ combining calls: $2n - 1$ calls in total. The helper runs in $O(n)$ time and has $O(\log n + 1)$ active depth. At $n = 0$, it makes one constant-time call. Input storage is $O(n)$.

Pause and predict: 11

Why is this sum not $O(\log n)$ time merely because each call splits the range in half?

Check your prediction. Both halves are processed. Halving controls the depth, but all n elements still become leaves. Binary search is different because it follows only one half.

A compact method for analyzing recursion

First identify the progress measure and maximum depth. Next count how many children actually execute. Then account for the work done by each call, including loops, copies, and printing.

Shape	Total time	Active depth
One call on $n - 1$, constant extra work.	$O(n)$	$O(n)$
One call on about $n/2$, constant work.	$O(\log n)$	$O(\log n)$
Two calls on disjoint halves, constant combining work.	$O(n)$	$O(\log n)$
Two calls on disjoint halves, linear combining work.	$O(n \log n)$	$O(\log n)$
Two calls on $n - 1$, constant work per internal call.	$O(2^n)$	$O(n)$

These are asymptotic bounds for growing positive n under the stated work assumptions. They do not include an unmentioned copied vector or stored collection of answers.

Derive the rows instead of memorizing their shapes

For two half-size children, the number of calls at depth d is roughly 2^d . If each call does constant combining work, summing $1 + 2 + 4 + \dots + n$ gives linear total work. If each level instead processes a total of n elements during combining, roughly $\log_2 n$ levels give $O(n \log n)$ work.

For two children on $n - 1$, there can be 2^d nodes at every depth d through n . The sum $1 + 2 + \dots + 2^n = 2^{n+1} - 1$ is exponential. This is the shape used by binary choices such as selecting or skipping each position.

2.12 Practical limits: recursion depth and numeric bounds

Termination does not guarantee enough stack space

A recursive algorithm may be logically correct yet create more active calls than its runtime environment can hold. Very deep chains can exhaust the call stack. The available depth depends on frame size, compiler, optimization, and environment; there is no single universal safe number to memorize.^[2-5]

This chapter keeps linear-recursion demonstrations at small bounds such as 1000 elements. That is a tested teaching bound, not a guarantee for every C++ implementation. When a production task requires hundreds of thousands of one-step calls, prefer an appropriate loop or an explicit stack rather than assuming it will fit.

Tail calls are not a portable space guarantee

A call is in a **tail position** when its result can be returned with no remaining work in that invocation. Compilers can optimize some such calls; GCC documents optimization of sibling and tail-recursive calls.^[2-6] We do not assume that optimization when stating the auxiliary space of a recursive C++ solution.

Separate three resource questions

Question	What to check
Will it stop?	Base case, valid child states, and a strictly decreasing progress measure.
Will it finish in time?	Number of calls, work inside each call, and output size.
Will its values and storage fit?	Maximum active depth, frame contents, stored results, and numeric bounds.

A subset enumeration can have depth only 20 but more than a million leaves. A linear sum of a million values can have only a million calls but a dangerously deep chain. These are different risks.

Test multiple invocations, not just one sample

When a helper shares a path, board, or accumulator, call it twice on small input during local testing. A changed second answer is a strong signal that some state survived the first call unintentionally. Reset independent test-case data deliberately.

Pause and predict: 12

Does `return helper(n - 1);` by itself justify claiming $O(1)$ auxiliary space in portable C++23?

Check your prediction. No. Without relying on a particular compiler optimization, the active calls can still form a depth- n chain. A loop makes the constant-space behavior explicit.

2.13 From a smaller problem to a tree of decisions

A partial answer is a state

Suppose you must generate every binary string of length 3. At the first position you may choose 0 or 1. After either choice, you still have a smaller job: fill the remaining two positions. Each additional choice creates another branch.

A **decision tree** records these possibilities. A node represents a partial state. An edge represents one next choice. A leaf represents a completed decision sequence or a branch that cannot continue. You need not store the whole tree: recursion can explore it a branch at a time.

```
empty prefix
0
  00 -> 000, 001
  01 -> 010, 011
1
  10 -> 100, 101
  11 -> 110, 111
```

Backtracking is systematic exploration, not random guessing

Backtracking builds a partial answer, explores choices, and returns to earlier decision points to try other choices. A common implementation pattern is **choose, explore, unchoose**: temporarily modify shared state, make the child call, then restore the state for the next alternative.^[2-7]

This traversal is **depth-first**: explore a chosen child and its descendants before returning to try the next sibling. The active calls represent one branch, not the whole tree.

Pruning can remove branches that are already known to be impossible. But the basic systematic generator is useful even without pruning: it gives a complete, understandable reference against which a faster method can be tested.

Write five things before implementing a search

Describe the state, list the legal next choices, state when an answer is complete, prove what becomes smaller, and specify what must be restored after a child returns. A helper should not rely on the reader guessing which entries of a path or board are meaningful.

For length- n binary strings: the state is the chosen prefix; choices are 0 and 1; completion means length n ; remaining length decreases; the chosen character is removed before exploring the sibling branch.

Pause and predict: 13

How many completed decision sequences are there when three positions each have two independent choices? What is the maximum number of choices along one path?

Check your prediction. There are $2 \times 2 \times 2 = 8$ completed sequences, but only three choices along a path. The number of leaves and the search depth measure different things.

2.14 Generate binary strings: choose, explore, and undo

One shared path, one clearly defined promise

The helper's contract is: print every length- n binary string extending the current `path`, then leave `path` exactly as it was on entry. The initial path is empty. Trying '0' before '1' gives lexicographic output order: compare at the first different character, just as in dictionary order.

Complete example | C++23

```
#include <iostream>
#include <initializer_list>
#include <string>
void generate(int n, std::string& path) {
    if (static_cast<int>(path.size()) == n) {
        std::cout << (path.empty() ? "EMPTY" : path) << '\n';
        return;
    }
    for (char bit : {'0', '1'}) {
        path.push_back(bit);
        generate(n, path);
        path.pop_back();
    }
}
int main() {
    int n;
    std::cin >> n;
    std::string path;
    path.reserve(n);
    generate(n, path);
}
```

Sample input

2

Sample output

00
01
10
11

Input contract. $0 \leq n \leq 12$. For $n = 0$, there is one valid empty string, printed as `EMPTY`. There are not zero answers.

Why it works. Every unfinished binary string has exactly one next character, either 0 or 1. The two branches cover all possibilities without overlap. Each child adds one character, and the matching pop restores the parent prefix for the next choice.

Cost. There are 2^n outputs, each of length n when $n > 0$. Including output, time is $O((n+1)2^n)$; path and active-call space are $O(n+1)$. The code streams results instead of storing them all.

Trace restoration for the prefix 0

Action	Path	Meaning
Enter this frame.	0	One more position remains for $n = 2$.
Append '0'.	00	Choose the first alternative.
Call the child.	00	It prints 00 and returns without changing the path.
Pop one character.	0	Restore this frame's entry state.
Append '1'.	01	Explore the second alternative.
Call and pop.	0	Print 01, then restore the entry state again.
Return.	0	The caller still owns this original prefix.

Every frame removes the character that *it* appended. A deeper frame does not remove its parent's earlier characters. The complete chain of contracts makes this safe: each child promises to restore the state it was given.

Why reserve capacity?

The maximum path length is known. Reserving that capacity avoids repeated growth allocations while the path expands and contracts. More importantly, the algorithm never copies the full prefix into every child. No pointer or reference to an individual path character is retained across a growth operation.

Output is part of the work

It is not accurate to claim $O(n)$ time because the recursion depth is n . Nor can an algorithm explicitly print 2^n different length- n strings in time independent of their total length. A problem that asks only for the *number* of strings is a different task and need not enumerate them.

Pause and predict: 14

After exploring both choices beneath the prefix 01, what should the shared path contain when that helper returns?

Check your prediction. Exactly 01. The helper may explore longer descendants, but its restoration contract preserves its entry prefix. Its caller will remove the earlier choice when appropriate.

2.15 Subsets: make a take-or-skip decision for each position

A position is considered exactly once

To select a subset of n labelled positions, decide whether to skip or take position 1, then position 2, and so on. Unlike a permutation, the order of selecting the same positions does not create a new subset. Store selected labels in increasing order.

The helper `subsets(i, n, path)` means: decisions before index i are fixed; enumerate all possibilities for positions i through $n - 1$. The stored labels are one-based, so taking index i appends $i + 1$.

Complete example | C++23

```
#include <iostream>
#include <vector>
void show(const std::vector<int>& path) {
    if (path.empty()) std::cout << "EMPTY";
    for (int i = 0; i < static_cast<int>(path.size()); ++i) {
        if (i > 0) std::cout << ' ';
        std::cout << path[i];
    }
    std::cout << '\n';
}
void subsets(int i, int n, std::vector<int>& path) {
    if (i == n) {
        show(path);
        return;
    }
    subsets(i + 1, n, path);
    path.push_back(i + 1);
    subsets(i + 1, n, path);
    path.pop_back();
}
int main() {
    int n;
    std::cin >> n;
    std::vector<int> path;
    path.reserve(n);
    subsets(0, n, path);
}
```

Sample input

2

Sample output

EMPTY

2

1

1 2

Input contract. $0 \leq n \leq 12$. Output one selected-label list per line, in skip-before-take traversal order. Use `EMPTY` for the empty subset.

Completeness and uniqueness

For any particular subset, its membership decision at each position is determined: take exactly the positions it contains. That gives one path from the root to a leaf. No different decision sequence can represent the same set of labels. Therefore every subset appears exactly once.

The two recursive branches are disjoint. One branch omits position $i + 1$; the other contains it. After the take branch returns, popping that label restores the parent's earlier decisions.

Do not confuse positions with values

For input values $[5, 5]$, there are four subsets of *positions*:

Selected labels	Selected values
EMPTY	Empty list.
2	$[5]$ using the second occurrence.
1	$[5]$ using the first occurrence.
1 2	$[5, 5]$.

If a task asks for distinct value-multisets instead, the two single-5 selections should be merged into one output. That requires a duplicate-handling rule, not merely the ordinary take-or-skip generator. Practice 11 makes that distinction explicit.

Count the generation cost

The full decision tree has $1 + 2 + \dots + 2^n = 2^{n+1} - 1$ calls. Printing each selected list adds output work, giving $O((n + 1)2^n)$ total time. The shared selected path and active recursion stack use $O(n + 1)$ space; the outputs are streamed.

The order in the sample is intentional. Skip-before-take traversal gives **EMPTY**, **2**, **1**, **1 2** for $n = 2$. It is not lexicographic order of the printed lists. A problem may accept any order or require a particular order; read that requirement before choosing branch order.

Pause and predict: 15

For values $[5, 5]$, how many position-subsets have sum 5? How many distinct value lists do those subsets produce?

Check your prediction. Two position-subsets have sum 5, one for each occurrence. Both produce the same value list $[5]$. Counting selections and deduplicating value outputs answer different questions.

2.16 Existence, a witness, a count, or every answer?

The same decision tree can support several tasks, but the return rule and stopping behavior differ. Before writing the helper, identify what the caller needs.

Task	What a successful leaf contributes	How branches combine
Existence	<code>true</code>	Logical OR; a proven success can stop the search.
One witness	Copy one successful path.	Stop after a success, while respecting cleanup.
Count	1	Add contributions from all relevant branches.
Enumerate	Print or save the completed path.	Explore every relevant branch.
Optimize	A score for this completed answer.	Take the minimum or maximum over valid branches.

A **witness** is an actual object demonstrating success, such as selected positions whose values sum to a target. A boolean says one exists; it does not tell the caller which positions to use.

Short-circuiting is useful only for the right question

For existence, once a child proves that a solution exists, unexplored children cannot change the answer from true to false. For counting, a success from one child says nothing about how many additional successes are in the other child.

With values $[0, 0, 1]$ and target 1, each zero can be taken or skipped independently. There are four different position-subsets with sum 1. Returning after the first one is correct for existence and incorrect for counting.

The empty selection has meaning

When the empty subset is allowed, an existence search with remaining target zero may immediately answer true: skip every remaining element. But a counting search usually cannot immediately return 1, because unprocessed zeros or cancelling positive and negative values may create further successful subsets.

Pause and predict: 16

At remaining target zero, may an existence search stop immediately when an empty completion is allowed? May a subset-counting search always return 1 there?

Check your prediction. The existence search may stop: choosing nothing else is a witness. The counting search may not assume it is the only completion. Its base case must preserve all choices that the task considers different.

2.17 Subset-sum existence: keep negative values valid

Define the remaining target

The helper `possible(a, i, remaining)` asks whether some subset of indices i through $n - 1$ sums to `remaining`. Skipping `a[i]` leaves the target unchanged; taking it subtracts that value from the target needed from later elements.

Complete example | C++23

```
#include <iostream>
#include <vector>
bool possible(const std::vector<int>& a, int i,
              long long remaining) {
    if (i == static_cast<int>(a.size())) return remaining == 0;
    if (possible(a, i + 1, remaining)) return true;
    return possible(a, i + 1, remaining - a[i]);
}
int main() {
    int n;
    long long target;
    std::cin >> n >> target;
    std::vector<int> a(n);
    for (int& x : a) std::cin >> x;
    std::cout << (possible(a, 0, target) ? "YES" : "NO")
               << '\n';
}
```

Sample input

3 3

8 -5 2

Sample output

YES

Input contract. $0 \leq n \leq 22$, $|a[i]| \leq 10^6$, and $|target| \leq 22 \cdot 10^6$. The empty subset is allowed, and positions are distinct. Output YES or NO.

Why it works. Every subset either omits the current position or includes it. The helper checks those two possibilities. When no positions remain, the only possible completion is empty, so it succeeds exactly when the remaining target is zero.

Cost. Worst-case $O(2^n)$ time, even though early success can save work. The stored input and the maximum active chain each use $O(n + 1)$ space. Do not promise a polynomial worst case because the function returns early on easy examples.

Why a negative remaining target is not automatically impossible

For values $[8, -5, 2]$ and target 3, taking 8 leaves target -5 . The later value -5 can satisfy that remaining target. A rule such as “return false whenever remaining is negative” would wrongly discard this valid selection.

Chosen positions	Chosen sum	Remaining target
None yet	0	3
Take position 1, value 8	8	-5
Take position 2, value -5	3	0
Skip position 3	3	0 at a complete leaf.

This trace isolates the successful branch; the sample program explores skip-before-take order and may visit other branches first. The proof must justify every pruning condition independently of the particular sample traversal.

Use values, not mutable shared totals, when possible

The remaining target is a value parameter. Passing `remaining - a[i]` does not modify the parent’s copy, so no “add it back” statement is required. Only objects you actually modify and share need explicit restoration.

One correct algorithm can still be unsuitable for a larger bound

At $n = 22$, exhaustive subset decisions can be used for a small-input teaching task. At $n = 200000$, the same decision tree is not reasonable, regardless of code formatting or fast input. First recognize the exponential search; later techniques may exploit additional bounds or structure.

2.18 Counting subsets: add disjoint branches and keep zero choices

The counting helper has the same state as the existence helper, but returns the number of successful completions. At the final index, it returns 1 for a matching remaining target and 0 otherwise. Both branches must be explored.

Complete example | C++23

```
#include <iostream>
#include <vector>
long long countWays(const std::vector<int>& a, int i,
                    long long remaining) {
    if (i == static_cast<int>(a.size())) {
        return remaining == 0 ? 1 : 0;
    }
    long long skip = countWays(a, i + 1, remaining);
    long long take = countWays(a, i + 1, remaining - a[i]);
    return skip + take;
}
int main() {
    int n;
    long long target;
    std::cin >> n >> target;
    std::vector<int> a(n);
    for (int& x : a) std::cin >> x;
    std::cout << countWays(a, 0, target) << '\n';
}
```

Sample input

Sample output

3 1

4

0 0 1

Input contract. $0 \leq n \leq 22$, $|a[i]| \leq 10^6$, and $|target| \leq 22 \cdot 10^6$. Count subsets of positions, including the empty subset when its sum matches. The count is at most 2^{22} under these bounds.

Why it works. The skip and take branches partition all candidate subsets according to whether they use the current position. Their counts add without overlap. Only complete decisions contribute a leaf count, so zero-valued and cancelling choices are not lost.

Cost. $O(2^n)$ time and $O(n + 1)$ active-call space, plus input storage. No list of all successful subsets is stored.

Pause and predict: 17

For values $[0, 0]$ and target 0, what count should the helper return?

Check your prediction. Four: the empty subset, each single position, and both positions. All four sum to zero. Returning 1 as soon as the remaining target is zero would miss three of them.

Why the sample has four answers

For values $[0, 0, 1]$ and target 1, the final position must be taken. Either of the first two positions may be taken or skipped:

Selected labels	Sum
3	1
2 3	$0 + 1 = 1$
1 3	$0 + 1 = 1$
1 2 3	$0 + 0 + 1 = 1$

Be explicit about the object being counted

Equal-valued positions are still different positions. A frequency map that groups values would change the state representation and would need to retain the number of ways to select occurrences. Do not silently replace the input with a set: a set discards multiplicity.

An optimization task needs another combination rule. For dividing values into two groups, each leaf can return the absolute difference of their sums, and each internal call can take the smaller of its children's results. Practice 10 uses that approach.

Counting avoids printing, not necessarily searching

Returning a count instead of every subset removes output work and storage, but the shown helper still visits the full decision tree. A closed formula or a dynamic-programming state may sometimes avoid that traversal. Neither benefit happens merely because the return type is numeric.

2.19 Pruning: reject only branches that cannot succeed

A pruning rule is a small impossibility proof

Pruning means returning before exploring all descendants because none can be valid. A faster search obtained by deleting a valid branch is simply wrong. State the assumption that makes the rejection safe.

For nonnegative remaining values in a subset-sum search, a remaining target below zero cannot be reached by adding more nonnegative values. If `suffix[i]` is the sum of all remaining nonnegative values, a target greater than that total is also impossible.

Fragment inside such a helper only:

```
// Requires every remaining value to be nonnegative.  
if (remaining < 0 || remaining > suffix[i]) return false;
```

This is not a valid drop-in change to the earlier signed-input helper. With negative values, the total suffix sum is not even a valid upper bound on all selectable sums: skipping a negative element can produce a larger sum.

A bound that also works with signed values

For a remaining suffix, let *low* be the sum of all negative values and *high* the sum of all positive values. Every subset sum lies in $[low, high]$: omitting any negative cannot make the sum smaller than *low*, and adding negatives cannot make it larger than *high*.

Therefore a remaining target outside that interval is impossible. A target *inside* it is not necessarily possible. For the suffix $[2, 4]$, the bound is $[0, 6]$, but target 3 cannot be formed. Such bounds are necessary conditions, not complete solutions.

Capacity pruning for an exact number of choices

If a partial selection still needs k elements but only r undecided elements remain, a branch with $k > r$ is impossible. If $k < 0$, too many elements have already been selected. These rules depend on counts, not signs, and remain valid for signed values.

Pause and predict: 18

A subset search has remaining values $[2, 4]$ and target 3. The target lies between 0 and the total 6. Does that prove a valid subset exists?

Check your prediction. No. The interval test can reject some impossible targets but cannot certify every target inside it. The available subset sums are 0, 2, 4, and 6.

Pruning and answer mode interact

For existence, a successful child can end the search. For minimum-score search, finding one feasible candidate only gives a current best value; unexplored branches might improve it. A **branch-and-bound** rule may prune a branch only when a valid bound proves it cannot improve the best known score. Deriving such bounds is problem-specific.

For counting or enumeration, zero remaining target is not generally a safe completion test while optional positions remain. For an exact- k subset counter, however, `need == 0` forces every remaining position to be skipped; there is exactly one completion if the target is also zero. The different state contract changes the correct stopping condition.

Choose good local checks

Check the cheapest clear restrictions before modifying state or making a child call. In parentheses generation, do not place a closing bracket before a matching opening bracket exists. In a board search, reject an attacked square before placing a queen. These checks avoid creating impossible prefixes in the first place.

State a worst-case bound honestly

Pruning often improves practical performance substantially, but it need not improve the worst-case order. An instance with many feasible prefixes may still force exploration of almost the whole search tree. Unless you have proved a tighter bound, retain the valid exponential upper bound.

A useful debugging pair

Write a tiny unpruned reference implementation and compare it with the pruned version on many small inputs. A difference reveals either an incorrect bound, a state-restoration bug, or a mismatch in what the two programs count.

2.20 Combinations: increasing choices prevent reordering duplicates

Choose exactly k different labels

To list k -element subsets of $\{1, 2, \dots, n\}$, always choose the next label larger than the previous one. This means $[1, 3]$ can appear but $[3, 1]$ cannot. They would represent the same combination.

The parameter `start` is the smallest label available next. If `need` labels still have to be chosen, the next label cannot exceed $n - \text{need} + 1$, or there would not be enough larger labels to finish.

Complete example | C++23

```
#include <iostream>
#include <vector>
void choose(int start, int n, int k, std::vector<int>& path) {
    int need = k - static_cast<int>(path.size());
    if (need == 0) {
        if (path.empty()) std::cout << "EMPTY";
        for (int i = 0; i < k; ++i) {
            if (i > 0) std::cout << ' ';
            std::cout << path[i];
        }
        std::cout << '\n';
        return;
    }
    for (int x = start; x <= n - need + 1; ++x) {
        path.push_back(x);
        choose(x + 1, n, k, path);
        path.pop_back();
    }
}
int main() {
    int n, k;
    std::cin >> n >> k;
    std::vector<int> path;
    path.reserve(k);
    choose(1, n, k, path);
}
```

Sample input

4 2

Sample output

1 2

1 3

1 4

2 3

2 4

3 4

Input contract. $0 \leq k \leq n \leq 15$. Print increasing selected-label lists in lexicographic order. For $k = 0$, print one EMPTY line.

Why the upper loop bound is correct

Suppose $n = 5$ and two more labels are needed. Choosing 5 next would leave no larger label for the final position. The greatest feasible next label is 4, which still permits 5 afterwards. This is exactly $n - need + 1 = 5 - 2 + 1$.

Completeness. Every k -element subset has one increasing order. The loop can choose its next smallest element, and enough larger labels remain because that subset itself provides a completion.

Uniqueness. Two different increasing choice paths differ at some chosen label, so they represent different subsets. There is no need to store a set of completed outputs to remove duplicates afterwards.

Restoration. Each frame appends its selected label, lets the child enumerate extensions, and pops that label before trying the next one.

Output-sensitive complexity

Let S be the number of output combinations. For $k > 0$, every visited prefix belongs to at least one completed combination, and the generator can be bounded by $O(kS)$ work including output. At $k = 0$, there is one constant-sized output. The combined form is $O(1 + kS)$ time and $O(k + 1)$ working space.

Later, Counting Principles will develop the formula for S . You do not need that formula to write or justify this generator.

Pause and predict: 19

With $n = 6$ and three labels still needed, what is the greatest next label worth trying?

Check your prediction. 4, because labels 5 and 6 can still complete the selection. Starting at 5 would leave only one larger label, so that branch cannot supply three labels in total.

2.21 Permutations: order matters and a used table prevents reuse

The choices are the unused values

A permutation arranges every label exactly once. Unlike a combination, 1 2 3 and 3 2 1 are different answers. At each depth, try each label that is not already in the path.

The invariant is: `used[x] == 1` exactly when label x occurs in the current path. Both pieces of state must be updated and restored together.

Complete example | C++23

```
#include <iostream>
#include <vector>
void arrange(int n, std::vector<int>& used,
            std::vector<int>& path) {
    if (static_cast<int>(path.size()) == n) {
        if (path.empty()) std::cout << "EMPTY";
        for (int i = 0; i < n; ++i) {
            if (i > 0) std::cout << ' ';
            std::cout << path[i];
        }
        std::cout << '\n';
        return;
    }
    for (int x = 1; x <= n; ++x) {
        if (used[x]) continue;
        used[x] = 1;
        path.push_back(x);
        arrange(n, used, path);
        path.pop_back();
        used[x] = 0;
    }
}
int main() {
    int n;
    std::cin >> n;
    std::vector<int> used(n + 1, 0), path;
    path.reserve(n);
    arrange(n, used, path);
}
```

Sample input

3

Sample output

1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1

Input contract. $0 \leq n \leq 8$. Print all permutations of labels 1 through n in lexicographic order. At $n = 0$, print one EMPTY line.

Why no label is repeated and no permutation is lost

A chosen label is marked before exploring the next position. Descendants therefore cannot choose it again. When the path reaches length n , it contains n distinct labels from an n -label domain, so it is a permutation.

For any desired permutation, its first label is offered at the root, its second label is offered after the first is marked, and so on. Exactly that order of choices reaches the desired leaf. Two different choice orders produce different permutations.

The undo order is part of the state contract

After a child finishes, pop the last path entry and clear its used mark. At the next iteration, the parent's shorter prefix and used table agree again. Clearing the mark before calling the child would allow a duplicate label inside the same path.

Factorial growth explains the small bound

There are n choices for the first position, $n - 1$ for the second, and so on. This gives $n!$ leaves. The shown implementation scans n candidate labels at each nonleaf call and prints n labels per leaf, for an $O(n \cdot n!)$ bound when $n \geq 1$. Path, flags, and active calls need $O(n + 1)$ space.

The code does not become suitable for $n = 20$ merely because the recursion depth is only 20. The number of outputs is the main obstacle.

Pause and predict: 20

After finishing the branch that begins with label 1, why must `used[1]` become zero before the root tries label 2?

Check your prediction. The new branch has not used label 1 yet and may need it in a later position. A mark left over from another branch would incorrectly forbid valid permutations.

2.22 Duplicate values: remove equivalent choices at the same depth

Repeated characters are not distinct labels

If the input word is `aba`, permuting its three positions generates six position orders but only three distinct strings: `aab`, `aba`, and `baa`. The task here asks for distinct output strings, so equivalent choices must be skipped.

Sort the word first. At a given depth, when two equal occurrences are both unused, choose the earlier occurrence as their representative. A later equal occurrence may still be used deeper in the same branch after the earlier one has been used.

Complete example | C++23

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

void generate(const std::string& s, std::vector<int>& used,
             std::string& path) {
    int n = static_cast<int>(s.size());
    if (static_cast<int>(path.size()) == n) {
        std::cout << path << '\n';
        return;
    }
    for (int i = 0; i < n; ++i) {
        if (used[i]) continue;
        if (i > 0 && s[i] == s[i - 1] && !used[i - 1]) continue;
        used[i] = 1;
        path.push_back(s[i]);
        generate(s, used, path);
        path.pop_back();
        used[i] = 0;
    }
}

int main() {
    std::string s, path;
    std::cin >> s;
    std::sort(s.begin(), s.end());
    std::vector<int> used(s.size(), 0);
    path.reserve(s.size());
    generate(s, used, path);
}
```

Sample input

aba

Sample output

aab

aba

baa

Input contract. A lowercase ASCII word of length 1 through 8. Print each distinct permutation once, in lexicographic order.

Read the skip condition in words

```
if (i > 0 && s[i] == s[i - 1] && !used[i - 1]) continue;
```

It says: if the previous occurrence has the same character and has *not* been used in this path, do not start an equivalent sibling branch from this later occurrence.

For sorted word **aab**, call the two positions a_0 and a_1 . At the root, choosing a_1 while a_0 is unused would create the same set of output strings as choosing a_0 . Skip that sibling. After a_0 has been chosen, however, a_1 is allowed in the next position, so strings with both copies are still generated.

Why this does not delete a distinct answer

Any word containing repeated equal characters has a representative position-order in which those equal occurrences are used from left to right. Relabelling equal occurrences does not change the output word. The condition retains that representative and rejects alternatives that differ only in which equal occurrence is chosen first.

Complexity. Sorting costs $O(n \log n)$. A simple worst-case upper bound for the generator is $O(n \cdot n!)$, reached in order of growth when all values are distinct. Repeated characters may reduce the search drastically. The used table, shared string path, and active calls use $O(n)$ working space, with outputs streamed.

Do not transfer the condition blindly to a different generator

For sorted unique-subset generation with a loop starting at **start**, the common rule is **i > start && a[i] == a[i - 1]**. It skips equal *sibling* choices but permits another equal occurrence at a deeper level. A global rule **i > 0** would incorrectly discard some repeated-value subsets. Practice 11 develops that model.

Pause and predict: 21

In the **aab** permutation generator, may the second **a** be chosen when the first **a** is already in the path?

Check your prediction. Yes. The earlier occurrence is then marked used, so the duplicate-sibling skip condition is false. Both copies can occur in the same completed word.

2.23 Balanced parentheses: reject invalid prefixes early

A valid prefix has enough opening brackets

To generate balanced strings with n pairs, keep the number of opening and closing brackets already used. An opening bracket is allowed while $\text{opened} < n$. A closing bracket is allowed only while $\text{closed} < \text{opened}$; otherwise that prefix would contain a closing bracket with no earlier unmatched opening bracket.

The state invariant is $0 \leq \text{closed} \leq \text{opened} \leq n$, and the current string contains exactly those numbers of brackets. Completion at $\text{closed} == n$ then also implies $\text{opened} == n$.

Complete example | C++23

```
#include <iostream>
#include <string>
void generate(int n, int opened, int closed,
             std::string& path) {
    if (closed == n) {
        std::cout << (path.empty() ? "EMPTY" : path) << '\n';
        return;
    }
    if (opened < n) {
        path.push_back('(');
        generate(n, opened + 1, closed, path);
        path.pop_back();
    }
    if (closed < opened) {
        path.push_back(')');
        generate(n, opened, closed + 1, path);
        path.pop_back();
    }
}
int main() {
    int n;
    std::cin >> n;
    std::string path;
    path.reserve(2 * n);
    generate(n, 0, 0, path);
}
```

Sample input

3

Sample output

((()))
 (()())
 (())()
 ()(())
 ()()()

Input contract. $0 \leq n \leq 9$. Print all balanced strings in lexicographic order, trying '(' before ')'. For $n = 0$, print one EMPTY line.

Why checking prefixes is enough

Every balanced bracket string has at least as many opens as closes in each prefix; otherwise the earliest excess closing bracket would be unmatched. It also uses exactly n opens and n closes. The generator enforces precisely those conditions while building the string.

Every generated leaf is therefore valid. Conversely, every next character of a valid balanced string obeys the relevant condition, so the generator has a branch following that string. Different character sequences are different outputs, giving uniqueness without a set.

A partial-state trace for $n = 2$

Prefix	Counts	Allowed next choices
Empty	$opened = 0, closed = 0$	Only '('.
(	1, 0	Either bracket.
((	2, 0	Only ')'; opening allowance is exhausted.
()	1, 1	Only '('; no unmatched opening remains.
(())	2, 2	Complete answer.

The condition prevents impossible prefixes such as $) ($ from ever being expanded. It is more efficient than generating all length- $2n$ bracket strings and checking only at the leaves.

Cost. Let B be the number of balanced outputs. Every explored valid prefix extends to a complete balanced string, so total generation and printing can be bounded by $O(1 + nB)$. The path and maximum active depth use $O(n + 1)$ space. A looser elementary bound is $O((n + 1)4^n)$, since there are at most 2^{2n} length- $2n$ candidate strings.

Pause and predict: 22

Why is `closed < n` alone not a sufficient condition for appending `)`?

Check your prediction. It limits the total number of closing brackets but does not ensure an earlier unmatched opening exists. The required condition is `closed < opened`.

2.24 N-queens: encode conflicts so each choice is easy to check

One row per recursive level

Place n queens on an $n \times n$ board so that no two share a row, column, or diagonal. Choose exactly one column for each row. Rows are distinct by construction, so only column and diagonal conflicts need flags.

For zero-based coordinates (row, col) , one diagonal family has constant $row - col$ and the other has constant $row + col$. Offset the first by $n - 1$ so both flag indices lie between 0 and $2n - 2$.

Complete example | C++23

```
#include <iostream>
#include <vector>
long long place(int row, int n, std::vector<int>& column,
                std::vector<int>& down, std::vector<int>& up) {
    if (row == n) return 1;
    long long ways = 0;
    for (int col = 0; col < n; ++col) {
        int d = row - col + n - 1, u = row + col;
        if (column[col] || down[d] || up[u]) continue;
        column[col] = down[d] = up[u] = 1;
        ways += place(row + 1, n, column, down, up);
        column[col] = down[d] = up[u] = 0;
    }
    return ways;
}
int main() {
    int n;
    std::cin >> n;
    std::vector<int> column(n, 0), down(2 * n - 1, 0);
    std::vector<int> up(2 * n - 1, 0);
    std::cout << place(0, n, column, down, up) << '\n';
}
```

Sample input

4

Sample output

2

Input contract. $1 \leq n \leq 10$. Output the number of valid placements. Reflections and rotations count as different placements when their occupied squares differ. We do not divide by symmetry factors.

What each flag means on entry

At `place(row, ...)`, queens have been placed in rows 0 through $row - 1$. A column or diagonal flag is set exactly when one of those earlier queens occupies it. The next row's candidate is legal precisely when all three relevant flags are clear.

After marking a legal square, the child handles the remaining rows. When it returns, all three marks are cleared. Since the choice was made only on clear flags, resetting them does not accidentally remove an earlier queen's occupancy.

Trace two placements for $n = 4$

One valid arrangement uses columns $[1, 3, 0, 2]$ in rows 0 through 3. Its $row - col$ values are $[-1, -2, 2, 1]$, all different; its $row + col$ values are $[1, 4, 2, 5]$, also all different. The other arrangement is its reflected pattern $[2, 0, 3, 1]$.

These examples explain the sample count; the algorithm does not hard-code them. It explores the allowed column choices and sums 1 for each complete placement.

Why it works. Every visited partial placement is nonattacking. Every valid board chooses a legal column in each row, so its choices remain available. Each row-by-row column sequence describes exactly one board.

Cost. Ignoring diagonal pruning, at most $n!$ complete column permutations remain. With up to n column checks per nonleaf call, $O(n \cdot n!)$ is a valid upper bound. Flags and active recursion use $O(n)$ space. Diagonal pruning reduces visited branches but does not justify a general polynomial bound.

Pause and predict: 23

Why does the `row - col` diagonal index need an offset, but `row + col` does not?

Check your prediction. `row - col` can be as small as $-(n - 1)$, which is not a valid vector index. Adding $n - 1$ makes it nonnegative. `row + col` is already between 0 and $2n - 2$.

2.25 Grid routes with only down and right moves

Progress can remove the need for a visited table

Start at the top-left cell of a rectangular grid and finish at the bottom-right. If only down and right moves are allowed, every move increases $row + col$ by one. A route cannot return to an earlier cell, so no visited table is needed.

Let `path` record the move letters taken so far. It is extended temporarily before the child call and restored afterwards. Down is tried before right, so routes are printed in lexicographic order using $D < R$.

Complete example | C++23

```
#include <iostream>
#include <string>
void routes(int row, int col, int rows, int cols,
            std::string& path) {
    if (row == rows - 1 && col == cols - 1) {
        std::cout << (path.empty() ? "EMPTY" : path) << '\n';
        return;
    }
    if (row + 1 < rows) {
        path.push_back('D');
        routes(row + 1, col, rows, cols, path);
        path.pop_back();
    }
    if (col + 1 < cols) {
        path.push_back('R');
        routes(row, col + 1, rows, cols, path);
        path.pop_back();
    }
}
int main() {
    int rows, cols;
    std::cin >> rows >> cols;
    std::string path;
    path.reserve(rows + cols - 2);
    routes(0, 0, rows, cols, path);
}
```

Sample input

2 3

Sample output

DRR

RDR

RRD

Input contract. $1 \leq rows, cols \leq 8$, with no obstacles in this example. Print one route per line. On a 1×1 grid, the empty route is already at the target and is printed as `EMPTY`.

The helper's contract

The current coordinates are inside the grid, and `path` describes a valid route from the start to those coordinates. The helper prints all completions to the destination and restores the path on return.

A down child is created only if the next row exists; a right child only if the next column exists. At the destination, the complete path is printed and the call stops. Without the stop, the destination could incorrectly become an intermediate point rather than the required end.

Completeness. Every route begins with one of the legal next directions. Recursively exploring each such direction covers every completion.

Uniqueness. A route is its move sequence. Different branches differ at a direction choice, so they print different sequences.

Count output length as well as the number of routes

Every completed route contains $L = \text{rows} + \text{cols} - 2$ moves. If P routes exist in the empty rectangle, generation and printing take $O(1 + LP)$ time, and path plus active calls use $O(L + 1)$ space. The 1×1 case performs constant work despite $L = 0$.

The no-obstacle assumption means every in-bounds prefix can be extended to the target. With obstacles, some legal prefixes may lead to dead ends, so a bound written only in terms of the successful output count needs additional care. Practice 16 uses a conservative decision-tree upper bound for that version.

A direction restriction can simplify both correctness and storage

Do not add a visited table merely because the task contains a grid. Ask whether revisiting is possible under the allowed moves. Conversely, removing the restriction to only down and right changes the problem substantially.

2.26 Four-direction simple paths: visited means on this path

The same cell can be used by different candidate routes

Now permit down, right, up, and left. A route may loop back unless it remembers which cells it has already used. We count **simple paths**: routes that do not visit a cell twice.

The shared grid contains '.' for open cells and '*' for blocked cells. While a cell is part of the current path, temporarily mark it '#'. Restore it to '.' when its exploration finishes. Different branches may legitimately visit it later.

Complete example | C++23

```
#include <iostream>
#include <string>
#include <vector>
long long paths(int row, int col, std::vector<std::string>& g) {
    int rows = static_cast<int>(g.size());
    int cols = static_cast<int>(g[0].size());
    if (row < 0 || row >= rows || col < 0 || col >= cols) return 0;
    if (g[row][col] != '.') return 0;
    if (row == rows - 1 && col == cols - 1) return 1;
    g[row][col] = '#';
    const int dr[4] = {1, 0, -1, 0};
    const int dc[4] = {0, 1, 0, -1};
    long long ways = 0;
    for (int d = 0; d < 4; ++d) {
        ways += paths(row + dr[d], col + dc[d], g);
    }
    g[row][col] = '.';
    return ways;
}
int main() {
    int rows, cols;
    std::cin >> rows >> cols;
    std::vector<std::string> g(rows);
    for (std::string& line : g) std::cin >> line;
    std::cout << paths(0, 0, g) << '\n';
}
```

Sample input

```
2 2
..
..
```

Sample output

```
2
```

Input contract. $1 \leq \text{rows}, \text{cols} \leq 4$. Each row contains exactly `cols` characters from . and *. Count simple paths from (0,0) to the opposite corner. A blocked endpoint gives zero; an open 1×1 grid gives one empty path.

Bounds before indexing, and rejection before marking

The helper first rejects coordinates outside the grid. Only then does it read `g[row][col]`. Any character other than `'.'` is unavailable: it is either a wall or already in this route. These checks happen before the target test so a blocked destination is not accepted.

At the destination, return 1 and stop. Otherwise mark the current cell, explore its four neighbours, restore the cell, and return the sum of the children's counts.

Why the mark must be undone for counting

A mark here means “belongs to this candidate path,” not “has ever been seen in the entire search.” If it remained set after returning from one branch, a later branch would lose valid routes passing through the same cell by a different prefix.

For a plain reachability question with no path-dependent constraints, a different algorithm can deliberately keep a global visited set to avoid repeating cells. That is a different state meaning. It is not a replacement for per-path marking when enumerating or counting simple routes.

Why it works. The marks forbid exactly the cells already in the path. Each child extends the path by one unused adjacent cell. Every simple route has a unique sequence of such choices, and the destination contributes once. Restoring the mark preserves choices for siblings.

Cost. With $V = rows \cdot cols$, active valid paths contain at most V cells. A conservative upper bound is $O(4^V)$ time and $O(V)$ additional active-call space, plus the stored grid. The exponential bound explains the small grid size; this is not an efficient large-grid shortest-path algorithm.

Pause and predict: 24

May a cell used in one completed route be used again in another candidate route? What must happen to its temporary mark?

Check your prediction. Yes. Routes are independent candidates. Remove that route's mark when unwinding so another branch may use the cell. Only reuse within the same route is forbidden.

2.27 Return one witness without leaving temporary state behind

A successful early exit still has a cleanup contract

To return one subset summing to a target, keep a shared path of selected one-based positions and copy it into `answer` at a successful leaf. The path is temporary search state; `answer` is the intentionally persistent result.

Complete example | C++23

```
#include <iostream>
#include <vector>
bool search(const std::vector<int>& a, int i, long long rem,
            std::vector<int>& path, std::vector<int>& answer) {
    if (i == static_cast<int>(a.size())) {
        if (rem != 0) return false;
        answer = path;
        return true;
    }
    if (search(a, i + 1, rem, path, answer)) return true;
    path.push_back(i + 1);
    bool found = search(a, i + 1, rem - a[i], path, answer);
    path.pop_back();
    return found;
}
int main() {
    int n;
    long long target;
    std::cin >> n >> target;
    std::vector<int> a(n), path, answer;
    for (int& x : a) std::cin >> x;
    if (!search(a, 0, target, path, answer)) {
        std::cout << "NONE\n";
    } else {
        if (answer.empty()) std::cout << "EMPTY";
        for (int j = 0; j < static_cast<int>(answer.size()); ++j) {
            if (j > 0) std::cout << ' ';
            std::cout << answer[j];
        }
        std::cout << '\n';
    }
}
```

Sample input

3 3
8 -5 2

Sample output

1 2

Input contract. $0 \leq n \leq 22$, $|a[i]| \leq 10^6$, and $|target| \leq 22 \cdot 10^6$. Print the first successful subset under skip-before-take traversal, `EMPTY` if that witness is empty, or `NONE` if no solution exists.

Separate the child result from returning

The important sequence is:

```
path.push_back(i + 1);
bool found = search(a, i + 1, rem - a[i], path, answer);
path.pop_back();
return found;
```

The boolean is saved so cleanup runs before the function exits. This preserves the promise that the temporary path is unchanged on return, regardless of success or failure.

Wrong fragment under this restoration contract:

```
path.push_back(i + 1);
if (search(a, i + 1, rem - a[i], path, answer)) return true;
path.pop_back(); // Skipped on the success path above.
```

Some one-shot programs deliberately leave the successful path in place and use that as their result. That can be a valid *different* contract. It is not valid to promise restoration while silently skipping it on success.

Why the successful answer survives

The assignment `answer = path` copies the selected labels into another vector. Later pops change `path`, not that saved `answer`. On failure, callers should rely on the false result and not read `answer` as a fresh witness.

Why it works. The search covers the same complete subset decisions as before. The first successful leaf copies a correct witness. Early returns preserve it while the temporary mutations are undone at each level. Skip-before-take ordering makes the requested first witness deterministic.

Cost. Worst-case $O(2^n + n)$ time, including at most one successful length- n copy. Input, `path`, `answer`, and maximum active stack use $O(n + 1)$ space.

Pause and predict: 25

Why does the correct code store the child's boolean in `found` instead of returning directly from the recursive call after pushing a label?

Check your prediction. It must pop that temporary label before returning. Saving the result separates “the child succeeded” from “this frame has finished its own cleanup.”

2.28 Debugging a recursive or backtracking solution

Debug the contract before the syntax

Write the meaning of each parameter and the expected state on return. A wrong result often comes from a mismatch between that meaning and a base case: an empty sum versus a nonempty maximum, a count versus existence, or position choices versus distinct value outputs.

Symptom	Likely question to investigate
Endless calls or a crash on larger input	Does every child make progress? Is the active depth practical?
Correct first answer, missing later answers	Did an early return stop counting or enumeration?
Results depend on branch order	Was shared state restored before exploring the next branch?
Duplicate output lines	Are positions or values the intended identity? Is skipping applied at the right depth?
All-negative arrays give zero	Did the base value match a sum contract rather than a maximum contract?
A second test case fails	Did a global counter, used flag, or path survive the previous task?
Tiny input works, large totals do not	Do every intermediate value and returned count fit their types?

Use small exhaustive checks

For subset sums, compare with all position-subsets on arrays of length at most 8. For permutations, validate length, multiplicity, uniqueness, and output count. For a grid witness, replay each move and confirm bounds, walls, and cell non-reuse. For a count, compare with an independent exhaustive generator on very small grids.

During local debugging, print an entry line and an exit line with indentation proportional to depth. Keep this diagnostic output out of the final judged solution. To check restoration, snapshot a small path or board before a helper call and compare it after the call.

A successful sample is evidence, not a proof

The later solutions explain why choices are complete, why outputs are valid, and why state remains consistent. Automated tests complement those arguments; they do not replace them.

2.29 Optional extension: Towers of Hanoi and parameter roles

The smaller problem changes which peg is the destination

There are three pegs and n differently sized disks, initially stacked from largest to smallest on peg A. Move the whole stack to C, moving one top disk at a time and never putting a larger disk on a smaller one.

To move n disks from `from` to `to`, first move the top $n - 1$ disks to `spare`, using `to` as the temporary peg. Move the largest disk directly to `to`. Then move the $n - 1$ disks from `spare` to `to`, using `from` temporarily.

Complete example | C++23

```
#include <iostream>
void moveDisks(int n, char from, char spare, char to) {
    if (n == 0) return;
    moveDisks(n - 1, from, to, spare);
    std::cout << from << ' ' << to << '\n';
    moveDisks(n - 1, spare, from, to);
}
int main() {
    int n;
    std::cin >> n;
    moveDisks(n, 'A', 'B', 'C');
}
```

Sample input

2

Sample output

A B
A C
B C

Input contract. $0 \leq n \leq 12$. Print one source-destination peg pair per move. At $n = 0$, print no moves. The output format here is a sequence of actions, not a list of possible solutions.

Why the helper is correct

The smaller calls preserve the disk-size rule by their contract. Before the largest disk moves, all smaller disks have been moved off it; its destination is clear of those smaller disks. The final smaller call puts them above it in the correct order.

The peg names are *roles in this invocation*, not permanently fixed positions in the algorithm. Swapping the argument roles for a smaller call describes a different source, spare, and destination.

An exact move count

Let $M(n)$ be the number of printed moves. Then $M(0) = 0$ and

$$M(n) = 2M(n-1) + 1.$$

Expanding a few values gives 0, 1, 3, 7, 15, ... Induction shows that $M(n) = 2^n - 1$: inserting $2^{n-1} - 1$ for each smaller count gives $2(2^{n-1} - 1) + 1 = 2^n - 1$.

The helper's total call count includes base cases as well, giving $2^{n+1} - 1$ invocations. Its time is $O(2^n)$, while the maximum active depth is only $n + 1$, so extra call space is $O(n + 1)$.

This is not a branching search for alternatives

Both smaller moves must be performed in order; the algorithm is not trying one peg choice and undoing it to try another. It is a recursively defined sequence of required actions. There are two recursive calls, but their meaning is composition, not candidate enumeration.

Pause and predict: 26

In the first recursive call, why is `to` passed as the spare peg rather than keeping every peg argument in the same order?

Check your prediction. That child must move the smaller stack from the current source to the current spare. The current destination is the only remaining temporary peg. Parameter roles change with the smaller task.

2.30 Optional extension: memoization reuses repeated answers

Repeated arguments can describe the same subproblem

The naive Fibonacci program solves the same `fibonacci(n)` subproblem repeatedly. Its answer depends only on n . A **memoization table** remembers each computed answer and returns it when the same state is requested again.

A table entry of -1 means “not yet computed.” That marker is safe here because Fibonacci values are nonnegative. It would not be a universally safe unknown marker for a problem whose valid answers may include -1 .

Complete example | C++23

```
#include <iostream>
#include <vector>
long long fibonacci(int n, std::vector<long long>& memo) {
    if (n <= 1) return n;
    if (memo[n] != -1) return memo[n];
    long long left = fibonacci(n - 1, memo);
    long long right = fibonacci(n - 2, memo);
    memo[n] = left + right;
    return memo[n];
}
int main() {
    int n;
    std::cin >> n;
    std::vector<long long> memo(n + 1, -1);
    std::cout << fibonacci(n, memo) << '\n';
}
```

Sample input

10

Sample output

55

Input contract. $0 \leq n \leq 90$. These results fit the tested signed 64-bit `long long` type. The memo table is created fresh for the task.

Why it works. The returned values follow the same recurrence. Reusing an already computed value is valid because the full answer is determined by its index n ; it does not depend on a temporary path or another caller’s choices.

Cost. Each nonbase state is expanded once, with constant work and two child requests, so time is $O(n + 1)$. The table and active recursion are each $O(n + 1)$ space. Memoization removes repeated work; it does not automatically shorten the initial deepest chain.

Do not memoize an incomplete description of the state

In subset-sum counting, the answer generally depends on both the current index and the remaining target. Caching only by index would combine different questions. If the input changes, the cache also belongs to that new input, not to an earlier test case.

For simple grid paths, the number of completions from a cell depends on which cells are already used by the current path. Two calls at the same coordinates can therefore have different legal futures. Caching only `(row, col)` would be incorrect for that task.

Memoization is not the same as a visited mark

A memo entry stores a completed answer for a full subproblem state. A temporary visited mark says a cell is unavailable on the current path. One reuses a result; the other constrains legal choices. Their lifetimes and meanings differ.

This small preview is enough to distinguish repeated work from exhaustive output generation. Printing every permutation still requires outputting every requested permutation; a cache cannot remove that output requirement.

Pause and predict: 27

Why is caching simple-path counts using only the current grid cell generally wrong?

Check your prediction. The set of cells already used changes which moves remain legal. The current cell alone does not identify the whole subproblem. Two paths arriving there can require different answers.

2.31 Optional extension: merge sort combines recursion with two pointers

Sort disjoint halves, then merge their sorted values

Merge sort sorts a half-open range $[lo, hi)$. A range of length zero or one is already sorted. Otherwise sort both halves recursively, then merge them using the coordinated scan from Session 1, Part A. One temporary vector is created in `main` and reused across all calls.

Complete example | C++23

```
#include <iostream>
#include <vector>
void mergeSort(std::vector<int>& a, std::vector<int>& temp,
               int lo, int hi) {
    if (hi - lo <= 1) return;
    int mid = lo + (hi - lo) / 2;
    mergeSort(a, temp, lo, mid);
    mergeSort(a, temp, mid, hi);
    int i = lo, j = mid, out = lo;
    while (i < mid && j < hi) {
        if (a[i] <= a[j]) temp[out++] = a[i++];
        else temp[out++] = a[j++];
    }
    while (i < mid) temp[out++] = a[i++];
    while (j < hi) temp[out++] = a[j++];
    for (int k = lo; k < hi; ++k) a[k] = temp[k];
}
int main() {
    int n;
    std::cin >> n;
    std::vector<int> a(n), temp(n);
    for (int& x : a) std::cin >> x;
    mergeSort(a, temp, 0, n);
    if (n == 0) std::cout << "EMPTY";
    for (int i = 0; i < n; ++i) {
        if (i > 0) std::cout << ' ';
        std::cout << a[i];
    }
    std::cout << '\n';
}
```

Sample input

6
4 -1 4 2 0 7

Sample output

-1 0 2 4 4 7

Input contract. $0 \leq n \leq 200000$, $|a[i]| \leq 10^9$. Print sorted values, or `EMPTY` if $n = 0$. This optional implementation teaches the combination step; ordinary contest solutions can usually use `std::sort` when only sorting is required.

Why this is divide and conquer

The children solve two disjoint subranges. After they return, each half is sorted. The merge repeatedly takes the smaller next value and appends the remaining suffix when one side is exhausted. Thus the merged range is sorted and contains exactly the original values, including duplicates.

Copying the merged range back to `a` is an intentional output mutation. It is not undone. The children and parent operate on different ranges of the same vector, but their calls are executed sequentially and the parent merges only after both children finish.

Derive its time from the combining work

Each recursion level processes a total of at most n values during merging and copying back. There are $O(\log n)$ levels for $n \geq 2$, giving $O(n \log n)$ time. Empty and one-element inputs take constant helper work.

Unlike the split-sum helper, this combination step is not constant-time: it scans the whole current range. The difference between $O(n)$ and $O(n \log n)$ comes from that per-level merge work, not from having a different number of children.

Storage is not just stack depth

The temporary vector uses $O(n)$ extra storage. The active stack has $O(\log n + 1)$ frames. Therefore total auxiliary storage is $O(n)$ for growing nonempty inputs, not merely logarithmic. The input vector itself is stored separately.

Creating a fresh length- n temporary vector in every recursive call would add unnecessary allocation and memory costs. The shared scratch vector makes the intended storage bound explicit.

Pause and predict: 28

Both recursive lower bound and merge sort split intervals in half. Why is merge sort not a logarithmic-time algorithm?

Check your prediction. Lower bound keeps only one half. Merge sort processes both halves and then merges all values in the current range. Across each level that is linear work, repeated for logarithmically many levels.

2.32 Practice problems: describe the call before writing it

Attempt the core problems first. For every recursive helper, write its input contract, base case, progress measure, and return meaning. For shared state, say whether it must be restored. Then estimate calls, active depth, and output size.

Problems 1 to 6 develop recursion itself. Problems 7 to 17 use decision trees, pruning, and backtracking. Problems 18 to 20 are optional extensions. Every task below has a complete solution later; use the hints first.

The requested recursive approach is part of these learning exercises. In a contest, prefer the simplest correct method that fits the constraints, which may sometimes be iterative.

1. Print on the way back

Write a recursive helper that prints the integers from 1 through n in increasing order. Use the return phase of the calls rather than a loop over the output values.

Input. One integer n . $0 \leq n \leq 1000$.

Output. One integer per line. At $n = 0$, print EMPTY.

Sample input	Sample output
3	1
	2
	3

2. Sum the squares recursively

Return $1^2 + 2^2 + \dots + n^2$ using a recursive helper. The sum of no terms is zero.

Input. One integer n . $0 \leq n \leq 1000$.

Output. The sum as an integer.

Sample input	Sample output
4	30

3. Count a requested decimal digit

Count occurrences of digit d in the ordinary decimal representation of the nonnegative integer x . Treat 0 as the one-digit representation 0; leading zeros in the input are not preserved as digits.

Input. Two integers x and d . $0 \leq x \leq 10^{18}$ and $0 \leq d \leq 9$.

Output. The number of occurrences.

Sample input	Sample output
10020 0	3

4. Maximum of a nonempty sequence

Return the maximum value in a sequence using recursion and an index. Do not make a new copy of the remaining suffix at each call.

Input. n , followed by n integers. $1 \leq n \leq 1000$ and $|a[i]| \leq 10^9$.

Output. The maximum value.

Sample input	Sample output
4 -8 -2 -10 -2	-2

5. Reverse a full line in place

Reverse every character of a complete input line using recursive swaps of its two ends. Spaces and punctuation are ordinary characters and must be preserved.

Input. One ASCII line, possibly empty. Line length is at most 1000.

Output. The reversed line. An empty line produces an empty output line, not the word **EMPTY**.

Sample input	Sample output
a bc	cb a

6. Recursive threshold counts

For each query value x , count how many elements of an already sorted sequence are at most x . Implement the boundary search recursively.

Input. n and q , then n nondecreasing integers, then q query values. $0 \leq n, q \leq 200000$; all values have absolute value at most 10^9 .

Output. One count per query. If $q = 0$, print nothing.

Sample input	Sample output
5 3	0
1 3 3 6 9	3
0	5
3	
10	

7. Binary strings with exactly k ones

Generate every length- n binary string containing exactly k ones, without storing all answers.

Input. Two integers n and k . $0 \leq k \leq n \leq 14$.

Output. One string per line in lexicographic order. At $n = k = 0$, print EMPTY.

Sample input	Sample output
3 2	011 101 110

8. Binary strings without adjacent ones

Generate all length- n binary strings that do not contain 11 as a substring.

Input. One integer n . $0 \leq n \leq 14$.

Output. One string per line in lexicographic order; EMPTY when $n = 0$.

Sample input	Sample output
3	000 001 010 100 101

9. Count exact-size target subsets

Count the subsets of exactly k different input positions whose values sum to a target. Equal values at different positions still represent different selections.

Input. n , k , and the target, followed by n integers. $0 \leq k \leq n \leq 22$, $|a[i]| \leq 10^6$, and $|target| \leq 22 \cdot 10^6$.

Output. The number of valid position-subsets.

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

10. Split into two groups as evenly as possible

Assign every value to one of two groups and minimize the absolute difference of their sums. Empty groups are allowed. Return the minimum difference, not the groups themselves.

Input. n , followed by n nonnegative integers. $0 \leq n \leq 22$ and $0 \leq a[i] \leq 10^9$.

Output. The smallest possible absolute difference.

Sample input	Sample output
4	0
3 1 4 2	

11. Distinct value-subsets from repeated inputs

Print each distinct nondecreasing value list selectable from the input occurrences, including the empty list. Using two equal copies is allowed when two copies exist; selecting either one alone must not produce duplicate output.

Input. n , followed by n integers. $0 \leq n \leq 15$ and $|a[i]| \leq 10^9$.

Output. Lists in lexicographic order, with each list space-separated and the empty list printed as EMPTY first.

Sample input	Sample output
3	EMPTY
2 1 1	1
	1 1
	1 1 2
	1 2
	2

12. Permutations without consecutive neighbours

Count permutations of labels 1 through n in which adjacent values never differ by exactly 1. For $n = 1$, the one-label permutation is valid.

Input. One integer n . $1 \leq n \leq 9$.

Output. The number of valid permutations.

Sample input	Sample output
4	2

13. Choose k labels when order matters

Print every ordered length- k sequence of different labels from 1 through n . Unlike a combination, 1 2 and 2 1 are different answers.

Input. Two integers n and k . $0 \leq k \leq n \leq 8$.

Output. One space-separated sequence per line in lexicographic order. At $k = 0$, print EMPTY.

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

14. Balanced brackets with a nesting limit

Generate balanced strings with n pairs whose nesting depth never exceeds h . Nesting depth at a prefix is the number of opens minus closes seen so far.

Input. n and h . $0 \leq h \leq n \leq 9$.

Output. All valid strings in lexicographic order, or NONE if none exist. At $n = 0$, print EMPTY.

Sample input	Sample output
3 1	() () ()

15. Queens with unavailable squares

Count nonattacking placements of n queens on an $n \times n$ board with some unavailable cells. Each row must contain exactly one queen. Unavailable cells do not block a queen's lines of attack.

Input. n , then n rows containing . for available cells and * for unavailable cells. $1 \leq n \leq 9$.

Output. The number of placements; rotated and reflected boards count separately when their occupied cells differ.

Sample input	Sample output
4 .*..	1

16. Down-and-right routes around obstacles

Print every route from the top-left to the bottom-right open cell using only down and right moves, without stepping on blocked cells.

Input. $rows, cols$, then the grid, using . for open and * for blocked. $1 \leq rows, cols \leq 8$.

Output. Move strings in lexicographic order with D tried before R. Print NONE if no route exists; an open 1×1 grid produces EMPTY.

Sample input	Sample output
3 3 ... *. ...	DDRR RRDD

17. Find one simple maze route

Find the first simple path from the top-left to bottom-right under depth-first exploration in direction order down, right, up, left. Do not reuse a cell. The result is not required to be shortest.

Input. $rows, cols$, then a $./^*$ grid. $1 \leq rows, cols \leq 4$.

Output. The first move string under the stated order, EMPTY for an open one-cell route, or NONE if impossible.

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

2 3	RRD
...	
.*.	

18. Optional: ordered step sequences

Print every ordered sequence of step sizes from $\{1, 2, 3\}$ whose sum is exactly t . Reusing a step size is allowed. Different orders are different answers.

Input. One integer t . $0 \leq t \leq 12$.

Output. Space-separated step sequences in lexicographic order. At $t = 0$, print EMPTY.

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

3	1 1 1
	1 2
	2 1
	3

19. Optional: reusable values without reordering duplicates

Given distinct positive allowed values, print all nondecreasing lists that sum to a target. Each value may be used any number of times; lists differing only by order are the same answer.

Input. n and target t , followed by n distinct allowed values. $1 \leq n \leq 8$, $1 \leq a[i] \leq 25$, and $0 \leq t \leq 25$.

Output. Lists in lexicographic order, **EMPTY** for target zero, or **NONE** if impossible.

Sample input	Sample output
3 7	2 2 3
2 3 5	2 5

20. Optional: a word on a small board

Decide whether a given word can be formed by starting at any cell and moving to orthogonally adjacent cells. A cell may be used at most once in that word path. Matching is exact and case-sensitive.

Input. $rows$, $cols$, the lowercase ASCII board, then the nonempty lowercase word. $1 \leq rows, cols \leq 4$; word length $1 \leq L \leq 10$.

Output. YES if a path exists, otherwise NO.

Sample input	Sample output
2 3 abc def abe	YES

2.33 Trace, correctness, and complexity drills

Work these out on paper before running code. Fragments are teaching illustrations, not separate complete programs. Deliberately wrong fragments are labelled and excluded from the program verification suite.

For a complexity answer, state what n measures, how many calls run, the work of each call, and how many frames are alive at once. For a correctness answer, identify a contract or give a concrete counterexample.

Drill 1: Trace entry and return

The following is a helper fragment, defined above `main`. Assume the program calls `f(2)`. Write the exact token output, the total number of calls, and the greatest number of helper frames alive together.

```
void f(int n) {
    if (n == 0) {
        std::cout << "X ";
        return;
    }
    std::cout << n << ' ';
    f(n - 1);
    std::cout << n << ' ';
}
```

Drill 2: Multiplication is not another recursive call

For this helper, compute `f(3)` and the total number of invocations it uses. Is the time linear or exponential in n ? Inputs in the executable thought experiment satisfy $0 \leq n \leq 20$ so these values fit.

```
long long f(int n) {
    if (n == 0) return 1;
    return 2 * f(n - 1) + 1;
}
```

Drill 3: Two children on the same smaller size

Compare this helper with Drill 2. For $g(3)$, find the returned value, total algorithm-level calls, and maximum active depth. Ignore compiler optimizations; the displayed calls describe the algorithm.

```
int g(int n) {
    if (n == 0) return 1;
    int left = g(n - 1);
    int right = g(n - 1);
    return left + right;
}
```

Use $0 \leq n \leq 20$ for bounded integer examples. Which resource is exponential, and which is linear?

Drill 4: A hidden vector copy in every call

Assume `a` initially has n integers and `scan(a, 0)` is called. Analyze time and maximum live storage, counting the copies explicitly. Then change one part of the declaration to avoid them.

```
void scan(std::vector<int> a, int i) {
    if (i == static_cast<int>(a.size())) return;
    scan(a, i + 1);
}
```

The helper is a teaching fragment; it intentionally does no useful per-element output.

Drill 5: A base case that the recursion skips

Deliberately wrong fragment. Trace only the arguments of the first five calls starting at `bad(3)`. Do not run it as a working solution.

```
void bad(int n) {  
    if (n == 0) return;  
    bad(n - 2);  
}
```

Explain why a base case is not enough. Give two possible ways a corrected contract and implementation could handle the issue.

Drill 6: Find the missing undo operations

This **deliberately wrong** binary-string helper promises to preserve its entry path. Locate the missing operations. Starting with $n = 2$ and an empty path, what happens after the first completed output 00?

```
void generate(int n, std::string& path) {  
    if (static_cast<int>(path.size()) == n) {  
        std::cout << path << '\n';  
        return;  
    }  
    path.push_back('0');  
    generate(n, path);  
    path.push_back('1');  
    generate(n, path);  
}
```

Drill 7: Zero targets: existence is not counting

A subset counter adds the early line below before considering all positions:

```
if (remaining == 0) return 1;
```

For values $[0, 0]$ and target 0, compare its returned count with the correct count. Explain why the line may be valid for an existence question but not this counting question.

Would the same return be valid when the state additionally says `need == 0` and the task requires selecting exactly that many further positions?

Drill 8: Pruning with signed values

A subset-sum solver rejects a branch when its remaining target is negative. Use values $[8, -5]$ with target 3 to disprove that rule for signed input.

For a remaining suffix $[-5, 8]$, compute a safe minimum and maximum possible-subset-sum bound. Does target 6 lying inside those bounds prove that 6 can be formed?

Drill 9: The wrong duplicate condition

A sorted-word permutation generator for `aab` uses the condition below after checking `used[i]`:

```
if (i > 0 && s[i] == s[i - 1]) continue;
```

Why does this remove valid strings? Give the corrected condition and list the distinct expected outputs.

Drill 10: Temporary visited marks on a small grid

Consider a fully open 2×3 grid. Start at the top-left and finish at the bottom-right. Moves may go down, right, up, or left, but a route may not reuse a cell.

List all simple move strings. Explain why a mark meaning “ever visited in any branch” is unsuitable for counting them. Why does the down-and-right-only version need no visited table at all?

Drill 11: Depth, total calls, and output storage

A binary-string generator emits all strings of length $n = 20$ using one shared path and no saved collection of outputs. Assume one character or newline costs one output unit.

Find the number of completed strings, the total helper calls in the full two-child tree, the maximum active helper frames, and the total output characters including one newline per string. How would storing all strings change the space bound?

Drill 12: Optional bridge: what belongs in a memo key?

For each helper, decide whether the proposed cache key fully describes the result. Assume immutable input stays fixed during the current task.

1. Fibonacci value, cached by n .
2. Subset-sum count, cached only by index i while the remaining target can differ.
3. Simple grid-path count, cached only by current cell while the current visited set can differ.

What must happen to a cache when an independent test case changes the input values or grid?

2.34 Hints before solutions

Try one small case and write the helper contract before reading a hint. These hints point to a state or a safe decision; they are not substitutes for tracing it yourself.

1. Print on the way back

Ask the smaller call to print through $n - 1$ before printing n . Handle the empty output convention in the driver.

2. Sum the squares recursively

For $n > 0$, the result is the square of n plus the answer for $n - 1$. Make the multiplication wide before adding it.

3. Count a requested decimal digit

Use a one-digit base case. Otherwise count the last digit and recurse on integer division by 10.

4. Maximum of a nonempty sequence

The final element can be the nonempty suffix base case. Compare the current element with the maximum of the remaining suffix.

5. Reverse a full line in place

Stop when the indices meet or cross. Swap once, then process the interior through a reference to the same string.

6. Recursive threshold counts

The first value strictly greater than x is at an index equal to the requested count. Keep a half-open interval.

7. Binary strings with exactly k ones

Track the number of ones already used. Prune if it is too large or if even filling all remaining positions with ones cannot reach k .

8. Binary strings without adjacent ones

Appending 0 is always safe. Appending 1 is safe only if the entry prefix is empty or ends in 0.

9. Count exact-size target subsets

Keep index, remaining number of choices, and remaining sum. When no more choices are needed, there is only one completion: skip every later position.

10. Split into two groups as evenly as possible

Choose which values enter one group. The other group contains the complement. If the chosen sum is s and total is T , the difference is $|T - 2s|$.

11. Distinct value-subsets from repeated inputs

Sort first. Print the current subset at every call, then choose its next value from a suffix. Skip equal candidates only when they are siblings: `i > start`.

12. Permutations without consecutive neighbours

Use a used table and inspect only the new last pair when appending a candidate. A forbidden adjacent pair cannot be repaired by adding later values.

13. Choose k labels when order matters

Use the permutation generator but stop when the path length reaches k , not n .

14. Balanced brackets with a nesting limit

The ordinary opening condition needs an extra restriction: the current depth must be below the limit before adding another open.

15. Queens with unavailable squares

Keep the row-by-row queen solver. Reject a candidate if the square is unavailable before setting its column and diagonal flags.

16. Down-and-right routes around obstacles

Reject out-of-bounds coordinates and blocked cells before checking whether you reached the destination. No visited table is needed for these monotone moves.

17. Find one simple maze route

Use temporary cell marking, but copy the path at the first successful destination. Store success in a boolean so the cell and path are restored before returning.

18. Optional: ordered step sequences

Try each step no larger than the remaining total. The remaining total decreases even though the same step value is available in later calls.

19. Optional: reusable values without reordering duplicates

Sort the allowed values. Recurse with the same candidate index to permit reuse, but never go back to a smaller index. Strictly positive values make the remaining target shrink.

20. Optional: a word on a small board

The state needs row, column, next word index, and the cells already used on the current path. Restore a marked cell even when a child succeeds.

Hints for the trace and correctness drills

Drill 1

The print after the recursive call belongs to the waiting parent frame. Include the base invocation when counting calls.

Drill 2

Count calls in the code, not coefficients in the arithmetic expression. Work upwards from the base value.

Drill 3

Each nonleaf node has two children, but they run one after the other. Count levels for total nodes and one branch for depth.

Drill 4

The vector is not shortened. Every call receives a fresh length- n copy, and the parent copies remain alive while the child runs.

Drill 5

Starting from an odd number and subtracting two preserves oddness. Decide whether odd inputs are valid for the intended problem.

Drill 6

Every push made by this frame needs a matching pop after its child has restored the state it received.

Drill 7

List all subsets of the two positions. Then distinguish optional future choices from a state in which no more choices are permitted.

Drill 8

Taking a positive value can leave a negative target that a later negative value satisfies. Lower and upper bounds are necessary conditions, not exact descriptions of every reachable sum.

Drill 9

The second equal occurrence must be allowed after the first is already used in the current path. Reject equivalent siblings, not all repeated occurrences.

Drill 10

The extra upward move permits a route that is not monotone. Different routes may pass through the same middle cell via different prefixes.

Drill 11

Use 2^n leaves, $2^{n+1} - 1$ total nodes, and one depth- n decision chain. Do not store the whole tree just because it exists conceptually.

Drill 12

Two calls may share a key only if they ask exactly the same question and have the same legal future choices.

2.35 Explained practice solutions

Each complete solution has its own `main`. The problem statements and sample cases appear in the practice section. Reconstruct the recursive state and stopping rule before reading the code.

The explanations distinguish the result returned by a call from temporary changes made to shared state. When comparing complexity, include the number of candidates, the cost of printing each output, and the maximum number of calls alive at the same time.

Solution 1: Print on the way back

Idea. Recurse before printing. The child handles every smaller positive integer, then the current frame prints its own final value.

Complete solution | C++23

```
#include <iostream>
void printAscending(int n) {
    if (n == 0) return;
    printAscending(n - 1);
    std::cout << n << '\n';
}
int main() {
    int n;
    std::cin >> n;
    if (n == 0) std::cout << "EMPTY\n";
    else printAscending(n);
}
```

Why it works. The base performs no helper output. Assuming the child prints $1, \dots, n-1$ correctly, appending n completes the required sequence. Each child decreases n . The driver represents the empty case explicitly.

Complexity. $O(n+1)$ time and $O(n+1)$ active-call space. A loop would use constant auxiliary space; recursion is required here as a learning exercise.

Boundary checks. Test 0, 1, and 3. Printing before the call would reverse the required order.

Solution 2: Sum the squares recursively

Idea. Remove the final term: $S(n) = n^2 + S(n - 1)$, with $S(0) = 0$. Use 1LL so the product is evaluated in long long.

Complete solution | C++23

```
#include <iostream>
long long squareSum(int n) {
    if (n == 0) return 0;
    return 1LL * n * n + squareSum(n - 1);
}
int main() {
    int n;
    std::cin >> n;
    std::cout << squareSum(n) << '\n';
}
```

Why it works. The smaller sum includes exactly the first $n - 1$ squares. Adding the missing n th square includes each required term once. The nonnegative argument decreases to zero.

Complexity. $O(n + 1)$ time and stack space. The maximum sum under the given bound is 333833500; the wide calculation also makes the intended arithmetic type explicit.

Boundary checks. Test $n = 0$, $n = 1$, and the maximum bound. This exercise studies a recurrence even though a nonrecursive formula also exists.

Solution 3: Count a requested decimal digit

Idea. Return whether the only remaining digit matches when $x < 10$. For a longer number, add the last-digit match to the count for the shorter number.

Complete solution | C++23

```
#include <iostream>
int occurrences(long long x, int digit) {
    if (x < 10) return x == digit ? 1 : 0;
    int last = x % 10 == digit ? 1 : 0;
    return last + occurrences(x / 10, digit);
}
int main() {
    long long x;
    int digit;
    std::cin >> x >> digit;
    std::cout << occurrences(x, digit) << '\n';
}
```

Why it works. Every decimal digit is inspected exactly once. The base correctly handles zero without pretending it has an empty representation. Dividing a number with at least two digits by 10 strictly shortens it.

Complexity. With D decimal digits, $O(D)$ time and $O(D)$ call space. Here $D \leq 19$.

Boundary checks. Test 0 0, 0 7, and a number containing several internal zeros.

Solution 4: Maximum of a nonempty sequence

Idea. Define the helper on a nonempty suffix. Return its only element at the final index, and otherwise combine the current value with the smaller suffix maximum.

Complete solution | C++23

```
#include <algorithm>
#include <iostream>
#include <vector>
int suffixMax(const std::vector<int>& a, int i) {
    if (i == static_cast<int>(a.size()) - 1) return a[i];
    int rest = suffixMax(a, i + 1);
    return std::max(a[i], rest);
}
int main() {
    int n;
    std::cin >> n;
    std::vector<int> a(n);
    for (int& x : a) std::cin >> x;
    std::cout << suffixMax(a, 0) << '\n';
}
```

Why it works. A nonempty suffix consists of its first value and a shorter suffix. The maximum of those two contributions is the suffix maximum. The one-element base uses an actual input value, so all-negative arrays remain valid.

Complexity. $O(n)$ time, $O(n)$ stored input, and $O(n)$ additional active-call space. The input vector is passed by `const` reference.

Boundary checks. Test one element, all negative values, and repeated maxima. Zero is not a valid universal maximum base value.

Solution 5: Reverse a full line in place

Idea. Move the endpoints inward after swapping them. The string is intentionally changed; this is not a backtracking path that must be restored.

Complete solution | C++23

```
#include <iostream>
#include <string>
#include <utility>
void reverseLine(std::string& s, int left, int right) {
    if (left >= right) return;
    std::swap(s[left], s[right]);
    reverseLine(s, left + 1, right - 1);
}
int main() {
    std::string s;
    std::getline(std::cin, s);
    reverseLine(s, 0, static_cast<int>(s.size()) - 1);
    std::cout << s << '\n';
}
```

Why it works. The two endpoint characters are placed at their final reversed positions. The recursive call reverses exactly the remaining interior. The base handles empty and single-character intervals before any access.

Complexity. $O(n + 1)$ time, $O(n)$ stored text, and $O(n + 1)$ additional stack space. No substring copies are created.

Boundary checks. Test an empty line, one character, and leading or trailing spaces. `getline` is necessary to read the full line.

Solution 6: Recursive threshold counts

Idea. Compute the upper-bound position: discard the middle element and everything before it when $a[\text{mid}] \leq x$. The returned boundary directly counts values at most x .

Complete solution | C++23

```
#include <iostream>
#include <vector>
int firstGreater(const std::vector<int>& a, int lo, int hi,
                int x) {
    if (lo == hi) return lo;
    int mid = lo + (hi - lo) / 2;
    if (a[mid] <= x) return firstGreater(a, mid + 1, hi, x);
    return firstGreater(a, lo, mid, x);
}
int main() {
    int n, q;
    std::cin >> n >> q;
    std::vector<int> a(n);
    for (int& x : a) std::cin >> x;
    while (q--) {
        int x;
        std::cin >> x;
        std::cout << firstGreater(a, 0, n, x) << '\n';
    }
}
```

Why it works. Sorted order makes each eliminated region uniform relative to the predicate. The half-open interval shrinks until both boundaries agree. All earlier elements are at most x and all later ones are greater, including duplicate and absent-target cases.

Complexity. $O(n + q(\log(n + 1) + 1) + 1)$ total time including input/output. Stored input is $O(n)$; one query uses $O(\log(n + 1) + 1)$ stack space, reused between queries.

Boundary checks. Test an empty sequence, a repeated query value present many times, a target above every element, and $q = 0$.

Solution 7: Binary strings with exactly k ones

Idea. Build a shared prefix, trying 0 before 1. With `left` positions unfilled, reject `ones > k` or `ones + left < k`.

Complete solution | C++23

```
#include <iostream>
#include <string>
void build(int n, int k, int ones, std::string& path) {
    int left = n - static_cast<int>(path.size());
    if (ones > k || ones + left < k) return;
    if (left == 0) {
        std::cout << (path.empty() ? "EMPTY" : path) << '\n';
        return;
    }
    path.push_back('0');
    build(n, k, ones, path);
    path.pop_back();
    path.push_back('1');
    build(n, k, ones + 1, path);
    path.pop_back();
}
int main() {
    int n, k;
    std::cin >> n >> k;
    std::string path;
    path.reserve(n);
    build(n, k, 0, path);
}
```

Why it works. The pruning checks reject only prefixes that cannot end with exactly k ones. Every valid string survives these tests and follows one character-choice path. Reaching length n after the checks guarantees the correct count, and each push is undone.

Complexity. A conservative bound is $O((n+1)2^n)$ time including output and $O(n+1)$ workspace. The count-based pruning reduces the actually visited tree.

Boundary checks. Test $k = 0$, $k = n$, and $n = 0$. Neither pruning condition may reject the valid empty string.

Solution 8: Binary strings without adjacent ones

Idea. Check whether the current last character allows a 1 before exploring either child. Always explore the 0 child and explore the 1 child only when allowed.

Complete solution | C++23

```
#include <iostream>
#include <string>
void build(int n, std::string& path) {
    if (static_cast<int>(path.size()) == n) {
        std::cout << (path.empty() ? "EMPTY" : path) << '\n';
        return;
    }
    bool canUseOne = path.empty() || path.back() != '1';
    path.push_back('0');
    build(n, path);
    path.pop_back();
    if (canUseOne) {
        path.push_back('1');
        build(n, path);
        path.pop_back();
    }
}
int main() {
    int n;
    std::cin >> n;
    std::string path;
    path.reserve(n);
    build(n, path);
}
```

Why it works. The prefix is valid on entry. The only new adjacent pair is the old last character and the new one, so the local check preserves validity. Every valid completed string obeys that check at each step and is generated once.

Complexity. $O((n+1)2^n)$ is a simple worst-case upper bound including output; the actual search is smaller. Extra path and active-call space are $O(n+1)$.

Boundary checks. Test $n = 0, 1$, and 3 . Reading `path.back()` without first checking emptiness would be invalid.

Solution 9: Count exact-size target subsets

Idea. Split by skipping or taking each position. Reject a negative required size or a size larger than the remaining suffix. At `need == 0`, return whether the remaining target is zero.

Complete solution | C++23

```
#include <iostream>
#include <vector>
long long count(const std::vector<int>& a, int i, int need,
               long long remaining) {
    int left = static_cast<int>(a.size()) - i;
    if (need < 0 || need > left) return 0;
    if (need == 0) return remaining == 0 ? 1 : 0;
    long long skip = count(a, i + 1, need, remaining);
    long long take = count(a, i + 1, need - 1, remaining - a[i]);
    return skip + take;
}
int main() {
    int n, k;
    long long target;
    std::cin >> n >> k >> target;
    std::vector<int> a(n);
    for (int& x : a) std::cin >> x;
    std::cout << count(a, 0, k, target) << '\n';
}
```

Why it works. The two branches partition subsets by current membership. A size bound removes only impossible selections. Unlike unrestricted subset counting, once zero more positions are allowed, no later zero may be selected; the unique completion is to skip them all.

Complexity. Worst-case $O(2^n)$ time and $O(n + 1)$ active-call space, plus input. Counts are at most 2^{22} under the contract.

Boundary checks. Test $k = 0$ with targets 0 and 1, repeated zeros, and negative values. Do not prune merely because the remaining target is negative.

Solution 10: Split into two groups as evenly as possible

Idea. Explore every subset as the first group. At each leaf evaluate `abs(total - 2 * chosen)` and return the smaller result from the two branches.

Complete solution | C++23

```
#include <algorithm>
#include <cstdlib>
#include <iostream>
#include <vector>
long long best(const std::vector<int>& a, int i,
               long long chosen, long long total) {
    if (i == static_cast<int>(a.size())) {
        return std::llabs(total - 2 * chosen);
    }
    long long skip = best(a, i + 1, chosen, total);
    long long take = best(a, i + 1, chosen + a[i], total);
    return std::min(skip, take);
}
int main() {
    int n;
    std::cin >> n;
    std::vector<int> a(n);
    long long total = 0;
    for (int& x : a) {
        std::cin >> x;
        total += x;
    }
    std::cout << best(a, 0, 0, total) << '\n';
}
```

Why it works. Every two-group assignment is represented by some chosen subset and its complement. Every leaf evaluates that assignment correctly. Taking the minimum over both children eventually covers every assignment; symmetric duplicates do not change the minimum.

Complexity. $O(2^n)$ time and $O(n + 1)$ recursion space, plus $O(n)$ input. Totals are at most $22 \cdot 10^9$ and intermediate differences fit `long long`.

Boundary checks. Test no values, one value, zeros, and an odd total. A perfect zero difference is not always possible.

Solution 11: Distinct value-subsets from repeated inputs

Idea. Generate increasing position selections from a sorted vector. At a given call, one branch represents each distinct next value. A deeper call may use another occurrence of the same value.

Complete solution | C++23

```
#include <algorithm>
#include <iostream>
#include <vector>
void generate(const std::vector<int>& a, int start,
             std::vector<int>& path) {
    if (path.empty()) std::cout << "EMPTY";
    for (int i = 0; i < static_cast<int>(path.size()); ++i) {
        if (i > 0) std::cout << ' ';
        std::cout << path[i];
    }
    std::cout << '\n';
    for (int i = start; i < static_cast<int>(a.size()); ++i) {
        if (i > start && a[i] == a[i - 1]) continue;
        path.push_back(a[i]);
        generate(a, i + 1, path);
        path.pop_back();
    }
}
int main() {
    int n;
    std::cin >> n;
    std::vector<int> a(n), path;
    for (int& x : a) std::cin >> x;
    std::sort(a.begin(), a.end());
    path.reserve(n);
    generate(a, 0, path);
}
```

Why it works. Sorted increasing indices preserve nondecreasing output and input multiplicities. The sibling skip removes only equivalent choices for the same next value. A later equal occurrence remains available after one has been chosen, preserving lists such as $[1, 1]$. Each valid value-multiset has one representative path.

Complexity. $O((n + 1)2^n)$ is a valid worst-case bound including output and candidate scanning; sorting is dominated for growing n . Path and active calls use $O(n + 1)$ space, plus stored input.

Boundary checks. Test an empty input, all equal values, and two repeated groups. Using $i > 0$ instead of $i > \text{start}$ wrongly removes repeated copies at deeper levels.

Solution 12: Permutations without consecutive neighbours

Idea. Extend a valid permutation prefix with an unused label whose difference from the previous label is not 1. Sum successful complete paths.

Complete solution | C++23

```
#include <cstdlib>
#include <iostream>
#include <vector>
long long count(int n, std::vector<int>& used,
                std::vector<int>& path) {
    if (static_cast<int>(path.size()) == n) return 1;
    long long ways = 0;
    for (int x = 1; x <= n; ++x) {
        if (used[x]) continue;
        if (!path.empty() && std::abs(path.back() - x) == 1) continue;
        used[x] = 1;
        path.push_back(x);
        ways += count(n, used, path);
        path.pop_back();
        used[x] = 0;
    }
    return ways;
}
int main() {
    int n;
    std::cin >> n;
    std::vector<int> used(n + 1, 0), path;
    std::cout << count(n, used, path) << '\n';
}
```

Why it works. Used marks enforce one occurrence per label. The adjacency check preserves validity because no other adjacent pair changes. Any valid permutation passes the check at every extension, so every valid order is counted once.

Complexity. $O(n \cdot n!)$ is a conservative time upper bound; pruning may reduce it. Used flags, path, and active calls use $O(n)$ space.

Boundary checks. Test $n = 1, 2, 3$, and 4 . Both the used flag and the path entry must be undone between sibling branches.

Solution 13: Choose k labels when order matters

Idea. Try all unused labels in increasing order. Stop after k choices while retaining the used-table invariant from full permutations.

Complete solution | C++23

```
#include <iostream>
#include <vector>
void generate(int n, int k, std::vector<int>& used,
             std::vector<int>& path) {
    if (static_cast<int>(path.size()) == k) {
        if (path.empty()) std::cout << "EMPTY";
        for (int i = 0; i < k; ++i) {
            if (i > 0) std::cout << ' ';
            std::cout << path[i];
        }
        std::cout << '\n';
        return;
    }
    for (int x = 1; x <= n; ++x) {
        if (used[x]) continue;
        used[x] = 1;
        path.push_back(x);
        generate(n, k, used, path);
        path.pop_back();
        used[x] = 0;
    }
}
int main() {
    int n, k;
    std::cin >> n >> k;
    std::vector<int> used(n + 1, 0), path;
    path.reserve(k);
    generate(n, k, used, path);
}
```

Why it works. Every output has length k and no repeated label. Every ordered selection can be followed through exactly its sequence of choices. The increasing candidate loop orders the output lexicographically without forcing the selected labels themselves to be increasing.

Complexity. With S output arrangements, $O((n + 1)S)$ bounds the candidate scans and output. Used flags need $O(n)$ and path plus recursion need $O(k + 1)$ space.

Boundary checks. Test $k = 0$, $k = 1$, and $k = n$. Passing an increasing **start** would generate combinations instead and lose valid orders.

Solution 14: Balanced brackets with a nesting limit

Idea. Use opening and closing counters. Allow an opening only if the quota is not exhausted and its new depth stays within the limit. Count printed results to detect no-answer cases.

Complete solution | C++23

```
#include <iostream>
#include <string>
long long generate(int n, int limit, int opened, int closed,
                  std::string& path) {
    if (closed == n) {
        std::cout << (path.empty() ? "EMPTY" : path) << '\n';
        return 1;
    }
    long long ways = 0;
    if (opened < n && opened - closed < limit) {
        path.push_back('(');
        ways += generate(n, limit, opened + 1, closed, path);
        path.pop_back();
    }
    if (closed < opened) {
        path.push_back(')');
        ways += generate(n, limit, opened, closed + 1, path);
        path.pop_back();
    }
    return ways;
}

int main() {
    int n, limit;
    std::cin >> n >> limit;
    std::string path;
    path.reserve(2 * n);
    long long ways = generate(n, limit, 0, 0, path);
    if (ways == 0) std::cout << "NONE\n";
}
```

Why it works. The original prefix-balance rules ensure proper matching. The extra opening check enforces the depth limit, and closing cannot increase depth. Every valid bounded-depth string follows allowed choices. Each complete path is printed once and contributes 1 to the generated-result count.

Complexity. With B printed strings, $O(1 + nB)$ time; valid prefixes can be completed when $h \geq 1$, and the no-solution $h = 0 < n$ case stops at the root. Workspace is $O(n + 1)$.

Boundary checks. Test $n = 0, h = 0$, positive n with $h = 0$, and $h = 1$. At depth 1, only alternating pairs can be completed without nesting.

Solution 15: Queens with unavailable squares

Idea. Combine the availability check with the same three conflict flags used by ordinary N-queens. The board itself is read-only; temporary occupancy lives in the flags.

Complete solution | C++23

```
#include <iostream>
#include <string>
#include <vector>
long long place(int row, const std::vector<std::string>& g,
                std::vector<int>& col, std::vector<int>& down,
                std::vector<int>& up) {
    int n = static_cast<int>(g.size());
    if (row == n) return 1;
    long long ways = 0;
    for (int c = 0; c < n; ++c) {
        int d = row - c + n - 1, u = row + c;
        if (g[row][c] == '*' || col[c] || down[d] || up[u]) continue;
        col[c] = down[d] = up[u] = 1;
        ways += place(row + 1, g, col, down, up);
        col[c] = down[d] = up[u] = 0;
    }
    return ways;
}
int main() {
    int n;
    std::cin >> n;
    std::vector<std::string> g(n);
    for (std::string& row : g) std::cin >> row;
    std::vector<int> col(n, 0), down(2 * n - 1, 0);
    std::vector<int> up(2 * n - 1, 0);
    std::cout << place(0, g, col, down, up) << '\n';
}
```

Why it works. Each partial placement uses only available squares and remains nonattacking. Every valid final board has one legal next column per row along its path. Clearing only the flags set for the current choice restores the earlier placement.

Complexity. $O(n \cdot n!)$ worst-case time upper bound. The board uses $O(n^2)$ storage; flags and recursive workspace use $O(n)$.

Boundary checks. Test one open cell, one blocked cell, a fully blocked row, and the sample. Walls are not screens that stop diagonal attacks in this task.

Solution 16: Down-and-right routes around obstacles

Idea. Explore down and then right from each usable cell. Append and remove the move letter around each child call. Return the number printed so the driver can detect zero results.

Complete solution | C++23

```
#include <iostream>
#include <string>
#include <vector>
long long routes(int r, int c, const std::vector<std::string>& g,
                std::string& path) {
    int rows = static_cast<int>(g.size());
    int cols = static_cast<int>(g[0].size());
    if (r >= rows || c >= cols || g[r][c] == '*') return 0;
    if (r == rows - 1 && c == cols - 1) {
        std::cout << (path.empty() ? "EMPTY" : path) << '\n';
        return 1;
    }
    path.push_back('D');
    long long down = routes(r + 1, c, g, path);
    path.pop_back();
    path.push_back('R');
    long long right = routes(r, c + 1, g, path);
    path.pop_back();
    return down + right;
}
int main() {
    int rows, cols;
    std::cin >> rows >> cols;
    std::vector<std::string> g(rows);
    for (std::string& row : g) std::cin >> row;
    std::string path;
    long long ways = routes(0, 0, g, path);
    if (ways == 0) std::cout << "NONE\n";
}
```

Why it works. Only legal open cells reach the target test. Each route has a unique down/right sequence, so the search prints every valid route once. Coordinates always progress, preventing cycles, and path restoration allows all sibling routes to be considered.

Complexity. Let $L = rows + cols - 2$. A safe bound is $O((L + 1)2^L)$ time including output; dead-end prefixes still cost work. Extra path and stack space are $O(L + 1)$, in addition to the $O(rows \cdot cols)$ grid.

Boundary checks. Test a blocked start, blocked destination, one row, one column, and an open 1×1 grid.

Solution 17: Find one simple maze route

Idea. Mark a current open cell, try neighbours in DRUL order, and stop after the first success. Save its path in `answer`, then undo the temporary path and grid changes.

Complete solution | C++23

```
#include <iostream>
#include <string>
#include <vector>
bool findPath(int r, int c, std::vector<std::string>& g,
              std::string& path, std::string& answer) {
    int rows = static_cast<int>(g.size());
    int cols = static_cast<int>(g[0].size());
    if (r < 0 || r >= rows || c < 0 || c >= cols) return false;
    if (g[r][c] != '.') return false;
    if (r == rows - 1 && c == cols - 1) {
        answer = path;
        return true;
    }
    const int dr[4] = {1, 0, -1, 0}, dc[4] = {0, 1, 0, -1};
    const std::string move = "DRUL";
    g[r][c] = '#';
    bool found = false;
    for (int d = 0; d < 4 && !found; ++d) {
        path.push_back(move[d]);
        found = findPath(r + dr[d], c + dc[d], g, path, answer);
        path.pop_back();
    }
    g[r][c] = '.';
    return found;
}
int main() {
    int rows, cols;
    std::cin >> rows >> cols;
    std::vector<std::string> g(rows);
    for (std::string& row : g) std::cin >> row;
    std::string path, answer;
    if (!findPath(0, 0, g, path, answer)) std::cout << "NONE\n";
    else std::cout << (answer.empty() ? "EMPTY" : answer) << '\n';
}
```

Why it works. Marks forbid exactly reuse within the current path. Ordered depth-first exploration gives the requested first witness. Saving the successful path makes it survive cleanup, while restoration on both outcomes preserves the helper's entry-state promise.

Complexity. With $V = rows \cdot cols$, $O(4^V)$ is a conservative time upper bound. Grid, current path, saved witness, and active calls together use $O(V)$ space.

Boundary checks. Test an initial dead end that requires backtracking, a blocked endpoint, and an open one-cell grid. Early return must not skip required cleanup.

Solution 18: Optional: ordered step sequences

Idea. Keep the remaining total and shared path. For each allowed step, subtract it in the child and undo its path entry on return.

Complete solution | C++23

```
#include <iostream>
#include <vector>
void generate(int remaining, std::vector<int>& path) {
    if (remaining == 0) {
        if (path.empty()) std::cout << "EMPTY";
        for (int i = 0; i < static_cast<int>(path.size()); ++i) {
            if (i > 0) std::cout << ' ';
            std::cout << path[i];
        }
        std::cout << '\n';
        return;
    }
    for (int step = 1; step <= 3 && step <= remaining; ++step) {
        path.push_back(step);
        generate(remaining - step, path);
        path.pop_back();
    }
}
int main() {
    int target;
    std::cin >> target;
    std::vector<int> path;
    path.reserve(target);
    generate(target, path);
}
```

Why it works. Positive steps ensure progress and prevent overshooting. Every valid ordered sequence has one corresponding choice path. At remaining total zero, that path is exactly a valid completion. Ascending step choices produce the requested order.

Complexity. With S output sequences, $O((t+1)S)$ time includes output and all extendible prefixes. A simpler loose bound is $O((t+1)3^t)$. Workspace is $O(t+1)$.

Boundary checks. Test $t = 0, 1$, and 3 . A zero-sized reusable step would destroy the progress argument.

Solution 19: Optional: reusable values without reordering duplicates

Idea. Use a nondecreasing candidate index and a decreasing remaining target. Trying candidate i passes i again, not $i + 1$, because that value may be reused.

Complete solution | C++23

```
#include <algorithm>
#include <iostream>
#include <vector>
long long generate(const std::vector<int>& a, int start,
                  int remaining, std::vector<int>& path) {
    if (remaining == 0) {
        if (path.empty()) std::cout << "EMPTY";
        for (int i = 0; i < static_cast<int>(path.size()); ++i) {
            if (i > 0) std::cout << ' ';
            std::cout << path[i];
        }
        std::cout << '\n';
        return 1;
    }
    long long ways = 0;
    for (int i = start; i < static_cast<int>(a.size()); ++i) {
        if (a[i] > remaining) break;
        path.push_back(a[i]);
        ways += generate(a, i, remaining - a[i], path);
        path.pop_back();
    }
    return ways;
}

int main() {
    int n, target;
    std::cin >> n >> target;
    std::vector<int> a(n), path;
    for (int& x : a) std::cin >> x;
    std::sort(a.begin(), a.end());
    long long ways = generate(a, 0, target, path);
    if (ways == 0) std::cout << "NONE\n";
}
```

Why it works. Every generated list is nondecreasing and has the required sum. Each valid list has one nondecreasing index sequence because input values are distinct. Positive choices guarantee finite depth. Sorting allows the loop to stop once a candidate exceeds the remaining target.

Complexity. With S visited calls and P outputs, time is $O(n \log n + nS + (t + 1)P)$. This remains an exhaustive search in general. The stored values use $O(n)$ space; path and recursion use $O(t + 1)$.

Boundary checks. Test target zero, a target smaller than all values, one reusable value, and a target with several decompositions. Zero or negative allowed values are explicitly excluded.

Solution 20: Optional: a word on a small board

Idea. Try every starting cell. Match the current character before marking the cell, then search four neighbours for the next character. Save and restore the character around the child search.

Complete solution | C++23

```
#include <iostream>
#include <string>
#include <vector>
bool match(int r, int c, int i, std::vector<std::string>& g,
           const std::string& word) {
    int rows = static_cast<int>(g.size());
    int cols = static_cast<int>(g[0].size());
    if (r < 0 || r >= rows || c < 0 || c >= cols) return false;
    if (g[r][c] != word[i]) return false;
    if (i + 1 == static_cast<int>(word.size())) return true;
    char saved = g[r][c];
    g[r][c] = '#';
    const int dr[4] = {1, 0, -1, 0}, dc[4] = {0, 1, 0, -1};
    bool found = false;
    for (int d = 0; d < 4 && !found; ++d) {
        found = match(r + dr[d], c + dc[d], i + 1, g, word);
    }
    g[r][c] = saved;
    return found;
}
int main() {
    int rows, cols;
    std::cin >> rows >> cols;
    std::vector<std::string> g(rows);
    for (std::string& row : g) std::cin >> row;
    std::string word;
    std::cin >> word;
    bool found = false;
    for (int r = 0; r < rows && !found; ++r) {
        for (int c = 0; c < cols && !found; ++c) {
            found = match(r, c, 0, g, word);
        }
    }
    std::cout << (found ? "YES" : "NO") << '\n';
}
```

Why it works. Each accepted cell matches exactly the required word position, and temporary marks forbid reuse. Advancing the word index reaches success only after its last character matches. Every valid word path has one start and a sequence of explored moves. Restoration keeps later starts and branches independent.

Complexity. With $V = rows \cdot cols$ cells and a word of length L , $O(V \cdot 4^L)$ is a conservative time bound. Stored input is $O(V + L)$; the active path needs $O(\min(V, L) + 1)$ additional stack space. No whole-board copy is made per call.

Boundary checks. Test a one-letter match, an absent letter, and board **ab** with word **aba**: the last one is impossible because it would reuse the first cell.

2.36 Explained trace and complexity drills

Compare the state meanings and reasoning, not only the final number. When your answer differs, identify the first call, branch, or storage assumption where the explanations diverge.

Drill 1: Trace entry and return

The tokens are 2 1 X 1 2. There are three invocations: $f(2)$, $f(1)$, and $f(0)$. All three are alive at the deepest point, but only $f(0)$ executes then.

The caller with $n = 1$ resumes first and prints 1; the caller with $n = 2$ resumes next and prints 2. Those two final prints do not create new calls. For general nonnegative n , the helper makes $n + 1$ calls and prints $2n + 1$ tokens.

Drill 2: Multiplication is not another recursive call

The returned values are $f(0) = 1$, $f(1) = 3$, $f(2) = 7$, and $f(3) = 15$. The execution has four invocations.

Each nonbase call requests one smaller value and does one constant-sized combination. Thus time and ordinary stack space are $O(n + 1)$. Multiplying the returned value by 2 is not the same as calling the helper twice. The numeric answers grow exponentially, but this implementation's number of calls grows linearly.

Drill 3: Two children on the same smaller size

The result is 8. There are $1 + 2 + 4 + 8 = 15$ calls and a maximum of four active helper frames.

More generally, the returned value is 2^n , while the call count is $2^{n+1} - 1$. Time is $O(2^n)$ and stack space is $O(n+1)$ with constant-sized frames. The second child begins after the first child has finished; sibling subtrees are not simultaneously retained on the call stack.

Storing both child results uses only two integers in a frame. That is different from storing the whole recursion tree.

Drill 4: A hidden vector copy in every call

There are $n + 1$ calls, each receiving a by-value copy of the same length- n vector. Under the direct copying model, total copying time is $O(n^2)$ and the simultaneously live element storage can also be $O(n^2)$, in addition to small frames.

Change the parameter to `const std::vector<int>& a`. Now every call reads the same vector, and passing it does not copy its elements. Time becomes $O(n + 1)$ and additional call space becomes $O(n + 1)$, with one $O(n)$ input vector.

Do not claim a constant-cost call without checking how its arguments are passed.

Drill 5: A base case that the recursion skips

The first five arguments are 3, 1, -1, -3, -5. None is zero. Merely decreasing a signed number does not prove that this specific stopping point will be reached.

One valid design requires even nonnegative input and keeps the zero base. Another, for a task where overshooting below zero means no work remains, can use $n \leq 0$ as its stopping condition. Which design is correct depends on the actual task and the required base answer.

Blindly changing the comparison is not a proof: confirm that every valid initial input and every child state obey the new contract.

Drill 6: Find the missing undo operations

Insert `path.pop_back()` after each recursive call. The first restores the parent prefix before choosing 1; the second restores it before returning to the parent's caller.

Without those pops, the leaf that prints 00 returns while the shared path still contains 00. Its parent then appends 1, producing 001. The equality base case no longer matches because length 3 is greater than target 2. Continued pushes can cause nonterminating growth rather than just a few duplicated answers.

Changing the base to `size() >= n` would avoid that specific endless growth but would not repair the state contract or make the generated outputs correct.

Drill 7: Zero targets: existence is not counting

The early line returns 1, but the correct count is 4: select neither position, only the first, only the second, or both. All have sum zero.

For existence with empty completion allowed, one empty continuation already proves success. For unrestricted counting, more choices may still produce successful completions, so they cannot be discarded.

When `need == 0` in an exact-size selection task, every remaining position must be skipped. There is exactly one allowed completion, and it succeeds precisely when the remaining target is zero. The extra state condition changes the meaning of the base case.

Drill 8: Pruning with signed values

Taking 8 leaves remaining target -5 , and taking the next value -5 then leaves zero. Rejecting the negative intermediate target discards the valid subset containing both positions.

For suffix $[-5, 8]$, the sum of all negatives is -5 and the sum of all positives is 8, giving safe bounds $[-5, 8]$. The actual subset sums are 0, -5 , 8, and 3. Target 6 lies inside the interval but is not reachable.

A branch outside the safe interval is impossible. A branch inside it merely deserves further investigation.

Drill 9: The wrong duplicate condition

The condition always skips the second **a**, even when the first **a** is already in the path. A length-three output can never use all required occurrences under that rule.

The corrected condition is:

```
if (i > 0 && s[i] == s[i - 1] && !used[i - 1]) continue;
```

It removes a duplicate sibling choice only when the earlier equal occurrence is still unused. The expected distinct outputs are **aab**, **aba**, and **baa**.

This rule belongs to the permutation state with used flags. A sorted-subset loop uses a different same-depth condition tied to its **start** index.

Drill 10: Temporary visited marks on a small grid

The four simple routes are **DRR**, **DRURD**, **RDR**, and **RRD**. The second one travels back upward without reusing a cell.

A cell that belonged to a previous branch must become available again for another route. A permanent global mark merges different path states and can delete valid candidates. Temporary marking means only “on the current path” and is undone while returning.

If moves are restricted to down and right, $row + col$ strictly increases, so returning to any earlier cell is impossible. The monotone version has the three routes **DRR**, **RDR**, and **RRD**; the longer fourth-direction route is no longer allowed.

Drill 11: Depth, total calls, and output storage

There are $2^{20} = 1048576$ strings and $2^{21} - 1 = 2097151$ helper calls. At most 21 helper frames are active together, including the completed-prefix base call.

Each output has 20 digits and one newline, giving $21 \times 1048576 = 22020096$ output characters. The shared path and active frames need $O(n)$ working space, and time including output is $O(n2^n)$ for positive n .

Saving all completed strings changes output storage to $\Theta(n2^n)$ characters, in addition to object/container overhead. Streaming avoids retaining that collection, but does not remove the cost of writing the outputs.

Drill 12: Optional bridge: what belongs in a memo key?

The Fibonacci key n is sufficient for the fixed recurrence. Its answer is determined by that integer alone.

The subset-count key is incomplete: both index and remaining target affect the answer. For example, with remaining suffix $[1]$, targets 0 and 1 each have one answer, but target 2 has none. An index alone cannot distinguish them.

The grid-path key is also incomplete because earlier visited cells constrain future routes. The full path-dependent state is not represented by coordinates alone.

A cache tied to a particular input must be cleared or otherwise separated when that input changes. Reusing a correct result for a different underlying array or grid answers the wrong subproblem.

2.37 Quick reference, glossary, and readiness checklist

Choose the meaning of one call first

A recursive helper is a promise about a smaller problem. Its parameters describe the state; its base cases and transitions must respect that description. The examples below are revision cues, not interchangeable templates.

Pattern	Meaning of a call	What must progress?
One smaller answer	Return the sum or maximum of the suffix beginning at i .	The suffix gets shorter.
Two boundaries	Check or transform the remaining interval.	Its length decreases, before accessing an empty interval.
One retained half	Return a search boundary inside $[lo, hi)$.	The half-open interval shrinks.
Skip or take	Consider all subsets of the remaining labelled positions.	The next undecided index increases in both branches.
Increasing choices	Extend a combination using only later positions.	The next allowed index increases.
Unused choices	Fill the next permutation position using an unused item.	The number of unused items decreases.
Grid paths	Complete a route from this cell without reusing the current path.	The number of available cells decreases.
Divide and combine	Solve disjoint smaller ranges, then combine their answers.	Every child range is strictly shorter.

Four different kinds of stopping rule

Direct base. The answer is already known: an empty sum is zero; a one-element maximum is that element.

Complete candidate. All required decisions have been made. Check any remaining condition, then count, save, or print the candidate.

Impossible prefix. A proof shows that no completion of the current choices can be valid. Reject this branch without rejecting unrelated siblings.

Early success. A single answer is enough for an existence or witness task. Return success, but still satisfy the shared-state contract on the way back.

A base case must match the question

“The remaining target is zero” can finish an existence search immediately when choosing nothing else is allowed. It cannot finish unrestricted subset counting when remaining zeros or canceling values could create additional answers.

Separate return values from temporary state

The value returned by a child is its result. The caller's local variables remain available when that child returns. Changes through references, however, are changes to the same shared object. A return does not automatically remove a pushed character or clear a used-cell mark.^{[2-1][2-2][2-3]}

State	Typical parameter	Expected behavior
Small independent value	<code>int index, long long sum</code>	Each call has its own parameter value.
Read-only input	<code>const vector<int>& a</code>	Reuse the same input without copying it.
Temporary path	<code>string& path</code>	Extend before the child and restore before the next sibling.
Availability flags	<code>vector<int>& used</code>	Mark a chosen item; unmark it after exploration.
Saved output	<code>vector<int>& answer</code>	Copy a successful path into this separate output object.
Intentional result mutation	<code>string& text</code> in reversal	Keep the required changes; do not undo the answer.

The reusable backtracking sequence

The following is **pseudocode**, not a complete C++ program:

```
search(state):
    if the state is complete:
        process this answer
        return
    for each legal next choice:
        remember what must be restored
        apply the choice
        search(the resulting smaller state)
        restore the exact entry state
```

If a child reports success, store its Boolean result, restore the temporary state, and only then return that result. A task may instead deliberately leave the successful path in place, but that is a different contract and must be documented consistently.

Match the aggregation to the requested answer

Use logical OR for existence, addition for counts over disjoint alternatives, and minimum or maximum for optimization. Enumeration visits every required branch. A witness search saves one valid candidate. Do not stop after the first success in a counting or enumeration problem.

Time and memory: the questions to ask

Count the number of visited calls, the work inside each call, and the output work. For memory, count only objects alive at the same time, including the call stack, shared working state, copies, and any answers intentionally stored.

Algorithm	Time	Extra workspace
One-element-at-a-time sum	$O(n + 1)$	$O(n + 1)$ call stack.
Recursive lower bound	$O(\log(n + 1) + 1)$ per query	Same bound for active calls.
Balanced split sum	$O(n + 1)$	$O(\log(n + 1) + 1)$.
Naive Fibonacci	$O(2^n)$ upper bound	$O(n + 1)$, not the total call count.
Binary-string generation	$O((n + 1)2^n)$ including output	$O(n + 1)$ when printing immediately.
Subset count by skip/take	$O(2^n)$	$O(n + 1)$ plus stored input.
Permutation generation	$O(n n!)$ for $n \geq 1$	$O(n)$ for flags, path, and calls.
Simple paths in a small grid	$O(4^V)$ conservative bound	$O(V)$ including the grid and path stack.
Memoized Fibonacci	$O(n + 1)$	$O(n + 1)$ table and stack.
Merge sort	$O(n \log n)$ for $n \geq 2$	$O(n)$ scratch plus $O(\log n)$ stack.

Here n is the size or index used by that algorithm, and $V = \text{rows} \cdot \text{cols}$ counts all grid cells, including blocked ones. Bounds for enumeration include writing each output item. A solution that *stores* all generated strings uses more space than one that prints and discards them.

Factorials and powers are counts, not advanced prerequisites

There are two choices at each of n unrestricted binary positions, giving 2^n leaves. For permutations, there are n choices for the first position, then $n - 1$, and so on. Their product is $n!$. Session 4 studies counting systematically; these elementary products are enough for this chapter.

Pruning is not a replacement for a complexity argument

Rejecting impossible prefixes may save substantial work, but it does not automatically produce a polynomial-time algorithm. State the worst-case bound you can justify, and use small input constraints for exhaustive search.

Glossary

Term	Meaning in this chapter
Recursive call	A request to solve a smaller instance using the same helper.
Contract	The meaning of the input, result, and allowed changes of one call.
Base case	A situation handled directly without another recursive call.
Progress measure	A quantity that moves toward a stopping condition.
Call frame	The information needed for one active call to continue later.
Unwinding	Returning through suspended callers after a child finishes.
Recursion tree	A drawing of the calls made during an execution.
Depth	The number of calls along an active ancestor-to-child chain.
Branch	One alternative choice or recursive child.
Backtracking	Exploring choices while restoring temporary state between alternatives.
Invariant	A property that remains true at a stated point in every call.
Pruning	Rejecting a branch after proving it has no required completion.
Witness	One concrete candidate demonstrating that a solution exists.
State restoration	Returning temporary shared objects to their entry values.
Memoization	Reusing a computed result for the same complete subproblem state.
Output-sensitive cost	A cost description that accounts for the answers that must be produced.

Four parts of a generation proof

Validity: every completed candidate satisfies the task. **Completeness:** every required candidate survives the pruning and is eventually reached. **Uniqueness:** no candidate is produced twice when distinct outputs are required. **Restoration:** each sibling begins with the correct shared state.

A claim about one of these does not establish the others. Generating only valid permutations does not prove that you generated all of them. Restoring the path does not prove that your duplicate-skipping rule is correct.

Readiness checklist

Before leaving this session, be able to do the following without copying a template:

- Describe one helper call, identify its base cases, and prove that every recursive branch makes progress.
- Trace a small call tree and distinguish what prints before the child, during the child, and after the child returns.
- Decide which information is passed by value, read by `const` reference, changed temporarily, or saved as output.
- Generate subsets, combinations, and permutations while explaining position identity, order, and duplicate handling.
- Prove a pruning condition and produce a counterexample to an invalid signed-sum pruning rule.
- Explain per-path grid marking, restore state on both success and failure, and distinguish a witness from a shortest route.
- Count total calls, maximum active depth, copying costs, and the unavoidable cost of printing many answers.

A practical route through the exercises

First pass: calls and returns. Solve Problems 1 to 6. Use very small inputs and write the expected call order before compiling. The recursive variants are exercises in reasoning, not recommendations to replace every simple loop.

Second pass: decision trees. Solve Problems 7 to 13. Draw at least one full tiny tree. Explain when a repeated input value represents a different labelled position and when it would create a duplicate output.

Third pass: constrained choices. Solve Problems 14 to 17. Put the shared state beside the code and match every temporary change with its restoration. Test a branch that succeeds early and a branch that fails.

Second-reading extensions. Return to Hanoi, memoization, and merge sort, then attempt Problems 18 to 20. These exercises combine the same contracts with different state descriptions; no bitmasking or advanced dynamic-programming background is required.

Use a small brute-force oracle

A slower independent solution is useful for testing a faster or pruned one. Compare complete output sets or counts on small inputs, then separately check output order, empty cases, and repeated calls. A disagreement is a concrete case to trace, not a reason to add an unexplained special case.

2.38 Technical notes, verification, and references

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

Language version and source conventions

The complete examples and solutions target **C++23** using standard headers. They do not require GNU PBDS, bitmasks, lambdas, or user-written templates. Ordinary function-call, return, and automatic-storage rules are supported by the fixed N4950 C++23 working-draft references below. These committee documents are technical references, not the suggested starting point for a beginner.^{[2-1][2-2][2-3]}

The printed complete listings are loaded from the actual `.cpp` files in the source package. Each file has its own `main`; compile one at a time. Fragments and deliberately wrong teaching examples are labelled separately and are not counted as working programs.

Numbers, characters, and valid inputs

The verification environment uses 32-bit `int` and 64-bit `long long`. C++ does not require every implementation to use exactly those widths. The stated input bounds and wide accumulators fit this tested environment. In particular, $20!$ and Fibonacci $F(90)$ fit signed 64-bit storage, while arbitrary larger values need not.

Products must be evaluated in a sufficiently wide type, not merely assigned to one afterwards. Signed integer overflow is not a valid way to represent a large mathematical result.^[2-8] The chapter also bounds output size and search depth, not only the magnitude of input values.

Unless a problem explicitly accepts a full line, text is given as a token. Character exercises use ordinary ASCII characters. The programs assume input follows the specified format and bounds; they do not print prompts or ask the user to correct invalid data.

Stack and memory claims

Call-stack analyses use the ordinary model in which active calls require frames. A compiler may optimize some calls, but the source code must not rely on tail-call optimization or a particular maximum supported recursion depth. Available stack space, frame sizes, compiler choices, and the runtime environment can differ.^{[2-5][2-6]}

Temporary strings and vectors are passed by reference when the intention is shared state. Copying a string or a whole vector is not a constant-time operation simply because it appears in one argument. String-copy rules and vector modification guarantees are documented in the library references.^{[2-4][2-9]}

For grid search, a mark belongs to the *current path*. A permanent visited set can be appropriate for a different task such as reachability, but is not a drop-in replacement when counting all simple paths. The explanation and code consistently use the path-specific contract.

Original program verification

The Session 2 source package contains **24 complete examples** and **20 complete practice solutions**. All **44 programs** compiled successfully with **GCC 14.2.0** in C++23 mode and passed **11,077 automated input/output checks**.

The suite checks all printed samples, small exhaustive cases, generated cases with a fixed seed, boundary values, duplicate handling, negative subset values, zeros, empty outputs, and large input sizes where the algorithm permits them. Independent expected results use iterative calculations, combinations and permutations, sorted outputs, and explicit-stack or exhaustive small-grid searches.

An additional **9 shared-state restoration probes** call selected helpers directly. They check that temporary paths, grids, and availability flags are restored after exploration, including successful witness searches and repeated helper calls. These probes are recorded separately from the input/output count.

Verification build

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

Undefined-behavior instrumentation and standard-library assertions can expose faults on executed paths. Passing these checks is not a mathematical proof for every valid input, nor a guarantee of safe recursion depth on another machine. Correctness explanations and explicit input contracts remain necessary.^[2-10]

Reproduce the checks

From the project root, run:

```
python3 tests/verify_recursion_backtracking.py
```

The fixed random seed is 62023103. The script records results and source hashes in **reports/verification_recursion_backtracking.json**. It compiles the program files referenced by the printed listings rather than a separately maintained copy.

The source package also retains separate verification scripts for the two parts of Session 1. Their individual reports give their own program and check counts; Session 2 does not silently include earlier tests in its 11,077-check figure.

Rebuild the handout or the collected book

The root files `session_02.tex` and `book.tex` include the same Session 2 chapter. The chapter includes the same complete program files, so changes stay synchronized between the two editions.

```
python3 build.py --target session2
python3 build.py --target book
```

The build script uses three LaTeX passes so page numbers, the contents, and links resolve. The README explains the required LaTeX packages and the locations of the output PDFs. The collected book's opening roadmap records which sessions have been added; later sessions are listed as planned rather than presented as finished chapters.

Reading the reference markers

A marker such as ^[2-1] links to a source below. The first number identifies this session, keeping its references separate from earlier chapters. Sources support language rules and the teaching concepts; the explanations, practice statements, examples, derivations, and test cases were prepared for this training.

[2-1] C++23 N4950: function calls and parameters

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

[2-2] C++23 N4950: returning from a function

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

[2-3] C++23 N4950: automatic storage duration

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

[2-4] C++23 N4950: string construction and copying

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

[2-5] Stanford CS106B: introduction to recursion and call-stack tracing

<https://web.stanford.edu/class/archive/cs/cs106b/cs106b.1244/lectures/07-recursion1/>

Further primary references

[2-6] GCC: optimization options, including sibling and tail calls

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

[2-7] Stanford CS106B: recursive backtracking

<https://web.stanford.edu/class/archive/cs/cs106b/cs106b.1244/lectures/11-backtracking1/>

[2-8] C++23 N4950: expression results outside the representable range

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

[2-9] C++23 N4950: vector modification rules

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

[2-10] GCC: instrumentation and undefined-behavior checks

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

A final check before the next session

Choose one example that returns a number and one that builds a path. Explain why the first does not need an undo operation and why the second does. Then explain why the recursion tree may contain exponentially many calls even when the deepest active chain is short.

Those distinctions prepare you to use new representations without changing the underlying algorithmic reasoning. The next session introduces bit manipulation and bitmasking; the subset and state ideas here will remain useful.

Practice planning

Use the practice sheet assigned by your training team alongside this chapter. Fill in the timeline when it is announced; no external sheet or deadline is assumed here.

Field	Your plan
Practice sheet	_____
Start date	_____
End date	_____
Problems to revisit	_____
One mistake to avoid	_____

After a contest, return to problems you could not complete and remove the time pressure. Rebuild a solution from its call contract and proof rather than trying to remember the exact code.