**🧩 BACKTRACKING**

**📘 Descripción**

**Backtracking** (vuelta atrás) es una técnica algorítmica que **explora todas las posibles soluciones** de un problema de forma sistemática, **retrocediendo** cuando detecta que una ruta no puede conducir a una solución válida.

En esencia:

Se construye la solución paso a paso, y si en algún punto no cumple las condiciones → retrocedes (“backtrack”).

**💡 Idea general**

Podemos pensar el backtracking como una **búsqueda en árbol de decisiones**.

Cada nivel del árbol representa una elección, y cada rama una opción posible.  
Cuando una elección no conduce a una solución válida → se regresa (backtrack) y se prueba otra.

**Pseudocódigo general:**

void backtrack(estado actual) {

if (es\_solucion(actual)) {

procesar(actual);

return;

}

for (opcion in opciones\_posibles) {

if (es\_valido(actual, opcion)) {

aplicar(opcion);

backtrack(nuevo\_estado);

deshacer(opcion);

}

}

}

**⚙️ Lógica**

1. **Exploras decisiones** una a una (añadir elemento, colocar reina, asignar número…).
2. **Compruebas validez parcial** antes de continuar (para podar ramas inútiles).
3. **Si completas una solución**, la guardas.
4. **Si no puedes continuar**, retrocedes y pruebas otra opción.

Esto lo hace extremadamente útil para:

* Combinatoria (combinaciones, permutaciones, subconjuntos)
* Juegos (Sudoku, 8-reinas, laberintos)
* Caminos en grafos (Hamiltonianos, Knight’s Tour)
* Generación de estados (números binarios, palabras, etc.)

**🧮 Ejemplos clásicos**

**🎯 1. N-Queens**

**🧠 Idea**

Colocar n reinas en un tablero n×n de modo que **ninguna se ataque entre sí** (ni misma fila, columna o diagonal).

**🧩 Lógica**

* Colocar una reina por fila.
* En cada fila, probar todas las columnas posibles.
* Si una posición es segura (no hay conflicto), colocas la reina y pasas a la siguiente fila.
* Si no puedes colocar ninguna reina en una fila → retrocedes.

**✨ Código (C++)**

#include <bits/stdc++.h>

using namespace std;

int n;

vector<int> col; // col[i] = columna donde está la reina en la fila i

bool ok(int row, int c) {

for (int i = 0; i < row; i++)

if (col[i] == c || abs(col[i] - c) == abs(i - row))

return false;

return true;

}

void solve(int row) {

if (row == n) {

for (int i = 0; i < n; i++) {

for (int j = 0; j < n; j++)

cout << (col[i] == j ? "Q " : ". ");

cout << "\n";

}

cout << "\n";

return;

}

for (int c = 0; c < n; c++) {

if (ok(row, c)) {

col[row] = c;

solve(row + 1);

}

}

}

int main() {

cin >> n;

col.resize(n);

solve(0);

}

**🧩 2. Sudoku Solver**

**💡 Idea**

Llenar una cuadrícula 9×9 de modo que cada fila, columna y subcuadro 3×3 tenga los números del 1 al 9 sin repetir.

**🧠 Lógica**

* Buscar una celda vacía.
* Probar del 1 al 9.
* Si el número no rompe las reglas, continuar recursivamente.
* Si no se puede completar → retroceder.

*(En Codeforces verás problemas tipo Sudoku simplificado o “latin squares”)*

**🧮 3. Combinaciones y subconjuntos**

**🔹 Combinaciones (choose k of n)**

void combine(int start, vector<int>& path) {

if (path.size() == k) {

for (int x : path) cout << x << " ";

cout << "\n";

return;

}

for (int i = start; i <= n; i++) {

path.push\_back(i);

combine(i + 1, path);

path.pop\_back();

}

}

**🔹 Subconjuntos (powerset)**

void subset(int i, vector<int>& a, vector<int>& path) {

if (i == a.size()) {

for (int x : path) cout << x << " ";

cout << "\n";

return;

}

// sin incluir a[i]

subset(i + 1, a, path);

// incluyendo a[i]

path.push\_back(a[i]);

subset(i + 1, a, path);

path.pop\_back();

}