8. Backtracking

El **Backtracking** es una técnica algorítmica general para encontrar todas (o algunas) soluciones a problemas computacionales que intentan construir una solución a un problema de forma incremental. Se basa en una búsqueda exhaustiva del espacio de estados del problema, explorando todas las posibles soluciones paso a paso. Si en algún momento la solución parcial actual no puede llevar a una solución completa válida, el algoritmo retrocede"(backtrack) y prueba una alternativa diferente.

—

0.1. ¿Qué es el Backtracking?

El Backtracking se puede visualizar como una exploración de un **árbol de espacio de estados**. Cada nodo en este árbol representa un estado parcial de la solución, y las ramas representan las elecciones que se pueden tomar.

El algoritmo procede de la siguiente manera:

1. Se comienza con un estado inicial vacío (la raíz del árbol).
2. En cada paso, se intenta extender la solución parcial actual eligiendo una de las posibles opciones disponibles.
3. Si la elección lleva a un estado que es una solución completa y válida, se registra la solución (o se cuenta).
4. Si la elección lleva a un estado que no puede ser parte de una solución válida (es decir, viola alguna restricción o es imposible completar la solución a partir de ahí), el algoritmo **retrocede**. Esto significa que se deshace la última elección y se prueba la siguiente opción disponible en el nivel anterior del árbol.
5. Si todas las opciones en un punto dado han sido probadas y ninguna lleva a una solución válida, se retrocede un nivel más.

La clave del Backtracking es la capacidad de **podar** (pruning) ramas del árbol de búsqueda que se sabe que no conducirán a una solución. Esto evita explorar combinaciones innecesarias, aunque la complejidad en el peor caso sigue siendo a menudo exponencial.

—

0.2. Ejemplos Clásicos de Backtracking en C++

Veamos un par de ejemplos ilustrativos de problemas que se resuelven eficientemente con Backtracking.

Generación de Permutaciones

Dado un conjunto de elementos distintos, encontrar todas las posibles permutaciones de esos elementos.

Listing 1: Generación de todas las permutaciones de un vector

```cpp
#include <iostream>
#include <vector>
#include <string>
#include <algorithm> // Para std::swap

using namespace std; // Incluyendo namespace std como se solicit

// Funci n auxiliar recursiva para generar permutaciones
void generatePermutations(vector<int>& nums, int start, vector<vector<int>>& result) {
    // Caso base: Si 'start' ha llegado al final del vector, hemos formado una permutaci n completa.
    if (start == nums.size()) {
        result.push_back(nums);
        return;
    }

    // Paso recursivo: Iterar a trav s de los elementos restantes para colocar en la posici n 'start'
    for (int i = start; i < nums.size(); ++i) {
        // 1. Elegir: Intercambiar el elemento actual con el elemento en 'start'
        swap(nums[start], nums[i]);

        // 2. Explorar: Llamada recursiva para generar permutaciones del resto del vector
        generatePermutations(nums, start + 1, result);

        // 3. Deshacer (Backtrack): Deshacer el intercambio para restaurar el estado original
        // Esto es crucial para que otras ramas de la recursi n puedan operar con el estado original
        swap(nums[start], nums[i]);
    }
}

int main() {
    vector<int> nums = {1, 2, 3};
    vector<vector<int>> allPermutations;

    cout << "Generando permutaciones para: ";
    for(int x : nums) cout << x << " ";
```

```
    cout << endl;

    generatePermutations(nums, 0, allPermutations);

    cout << "Todas las permutaciones:" << endl;
    for (const auto& perm : allPermutations) {
        cout << "[";
        for (int i = 0; i < perm.size(); ++i) {
            cout << perm[i] << (i == perm.size() - 1 ? "" : ", ");
        }
        cout << "]" << endl;
    }
    return 0;
}
```

Problema de las N-Reinas

Colocar N reinas en un tablero de ajedrez de $N \times N$ de manera que ninguna reina ataque a otra (es decir, no haya dos reinas en la misma fila, columna o diagonal).

Listing 2: Solución al Problema de las N-Reinas

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

using namespace std; // Incluyendo namespace std como se solicit

// Funci n para imprimir una soluci n del tablero
void printBoard(const vector<string>& board) {
    for (const string& row : board) {
        cout << row << endl;
    }
    cout << endl;
}

// Funci n para verificar si es seguro colocar una reina en (row, col)
bool isSafe(const vector<string>& board, int row, int col, int n) {
    // Comprobar la misma columna
    for (int i = 0; i < row; ++i) {
        if (board[i][col] == 'Q') {
            return false;
        }
    }

```

```cpp
    // Comprobar diagonal superior izquierda
    for (int i = row - 1, j = col - 1; i >= 0 && j >= 0; --i, --j) {
        if (board[i][j] == 'Q') {
            return false;
        }
    }

    // Comprobar diagonal superior derecha
    for (int i = row - 1, j = col + 1; i >= 0 && j < n; --i, ++j) {
        if (board[i][j] == 'Q') {
            return false;
        }
    }
    return true;
}

// Funci n recursiva de backtracking para resolver N-Reinas
void solveNQueens(vector<string>& board, int row, int n, vector<vector<string>>& solutions) {
    // Caso base: Si todas las reinas est n colocadas
    if (row == n) {
        solutions.push_back(board);
        return;
    }

    // Probar colocar una reina en cada columna de la fila actual
    for (int col = 0; col < n; ++col) {
        if (isSafe(board, row, col, n)) {
            // Elegir: Colocar la reina
            board[row][col] = 'Q';

            // Explorar: Llamada recursiva para la siguiente fila
            solveNQueens(board, row + 1, n, solutions);

            // Deshacer (Backtrack): Quitar la reina para probar otras configuraciones
            board[row][col] = '.';
        }
    }
}

int main() {
    int n = 4; // Para el problema de 4 Reinas
    vector<string> board(n, string(n, '.')); // Inicializar tablero vac o
```

```cpp
    vector<vector<string>> allSolutions;

    cout << "Resolviendo el problema de las " << n << "
        Reinas." << endl;
    solveNQueens(board, 0, n, allSolutions);

    cout << "Se encontraron " << allSolutions.size() << "
         soluciones:" << endl;
    for (const auto& sol : allSolutions) {
        printBoard(sol);
    }
    return 0;
}
```

—

0.3. Análisis de Eficiencia del Backtracking

La complejidad de los algoritmos de Backtracking es generalmente **exponencial**, debido a la naturaleza de la búsqueda exhaustiva del espacio de estados.

- **Complejidad Temporal:**
  - Varía enormemente según el problema y la eficacia de la poda.
  - En el peor caso, puede ser $\mathcal{O}(N!)$ (como en permutaciones sin poda), o $\mathcal{O}(B^D)$, donde B es el factor de ramificación (número de opciones en cada paso) y D es la profundidad máxima del árbol de búsqueda.
  - Para N-Reinas, aunque hay N^N posibles posiciones, las restricciones reducen significativamente el espacio, pero sigue siendo un problema de complejidad exponencial.
- **Complejidad Espacial:**
  - Generalmente $\mathcal{O}(N)$ o $\mathcal{O}(D)$, debido a la profundidad de la pila de llamadas recursivas y al almacenamiento de la solución parcial actual.

A pesar de su complejidad exponencial, el Backtracking es a menudo la única forma práctica de resolver problemas donde se deben encontrar todas las soluciones o una solución óptima en un espacio de búsqueda combinatorio. La clave es maximizar la eficiencia de la poda.

—

0.4. Aplicaciones Comunes del Backtracking

El Backtracking se utiliza para resolver una amplia gama de problemas de búsqueda combinatoria y optimización:

- **Generación de todas las Permutaciones, Combinaciones y Subconjuntos.**

- **Sudoku Solver:** Encontrar una solución para un tablero de Sudoku dado.
- **Problema del Salto del Caballo:** Encontrar una secuencia de movimientos de un caballo que visite cada casilla de un tablero una sola vez.
- **Problema de la Suma de Subconjuntos:** Encontrar un subconjunto de números que sume un valor objetivo.
- **Problema de la Mochila (0/1 Knapsack):** Encontrar la combinación de ítems que maximice el valor sin exceder una capacidad dada (aunque a menudo se resuelve con Programación Dinámica para eficiencia).
- **Coloración de Grafos.**

El Backtracking es una técnica fundamental para problemas donde la búsqueda exhaustiva es necesaria, pero puede ser guiada por un corte para mejorar la eficiencia práctica.