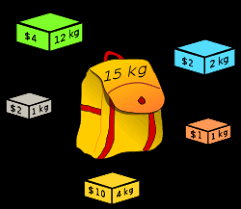
**Búsqueda Exhaustiva**

1. Definición de la Búsqueda Exhaustiva

La búsqueda exhaustiva, también conocida como búsqueda completa o fuerza bruta, es una técnica algorítmica utilizada para encontrar soluciones óptimas o satisfactorias a un problema examinando sistemáticamente todas las posibles soluciones en un espacio de búsqueda. Aunque puede ser ineficiente en términos de tiempo de ejecución, la búsqueda exhaustiva es útil cuando el espacio de búsqueda es pequeño o cuando no hay una estructura lógica clara que permita optimizar la búsqueda.

En C++, la búsqueda exhaustiva se implementa típicamente utilizando bucles anidados o recursión, dependiendo de la naturaleza del problema.



*Imagen 1: Problema del Viajero resuelto con búsqueda exhaustiva*

**2. Consideraciones:**

Las colas de prioridad tienen una amplia variedad de aplicaciones en el desarrollo de software. Algunos ejemplos destacados incluyen:

1. Eficiencia: La eficiencia de la búsqueda exhaustiva puede verse afectada significativamente por el tamaño del espacio de búsqueda y la complejidad del problema. Para problemas con un espacio de búsqueda grande, la búsqueda exhaustiva puede volverse prohibitivamente lenta y poco práctica.
2. Optimización: En algunos casos, puede ser necesario explorar otras técnicas de optimización, como la programación dinámica o el uso de algoritmos heurísticos, para mejorar el rendimiento y encontrar soluciones más rápidas y eficientes.
3. Limitaciones: La búsqueda exhaustiva puede no ser la mejor opción para todos los problemas, especialmente aquellos con un espacio de búsqueda muy grande o donde existen estructuras lógicas que permiten optimizaciones significativas.

**2.1 Ejemplo:**

A continuación, presentamos un ejemplo sencillo de implementación de búsqueda exhaustiva en c++ realizando sumas.

#include <iostream>

// Función para calcular la suma de los números en un rango dado

int sumaEnRango(int inicio, int fin) {

int suma = 0;

for (int i = inicio; i <= fin; ++i) {

suma += i;

}

return suma;

}

int main() {

int inicio, fin;

std::cout << "Ingrese el inicio del rango: ";

std::cin >> inicio;

std::cout << "Ingrese el final del rango: ";

std::cin >> fin;

int resultado = sumaEnRango(inicio, fin);

std::cout << "La suma de los números en el rango dado es: " << resultado << std::endl;

return 0;

}  
  
Tenemos otro ejemplo más detallado de Búsqueda Exhaustiva en la carpeta de Código Búsqueda Exhaustiva en el Git Hub.

**3. Conclusiones**

La búsqueda exhaustiva es una técnica simple pero poderosa para abordar problemas algorítmicos en C++. Aunque puede ser ineficiente en algunos casos, sigue siendo útil para resolver problemas con un espacio de búsqueda pequeño o cuando otras técnicas de optimización no son aplicables. Es importante considerar las limitaciones y explorar otras estrategias cuando sea necesario para lograr soluciones más eficientes.

**4. Recomendaciones**

* Evalúa la complejidad del problema: Antes de usar búsqueda exhaustiva, considera el tamaño del espacio de búsqueda y la complejidad del problema. Si el espacio de búsqueda es grande, otras técnicas pueden ser más eficientes.
* Explora técnicas de optimización: Si el problema es complejo, considera otras técnicas como programación dinámica o algoritmos heurísticos para mejorar el rendimiento.
* Usa búsqueda exhaustiva con moderación: Evalúa si hay alternativas más eficientes antes de recurrir a búsqueda exhaustiva.
* Prueba exhaustivamente tu implementación: Asegúrate de probar tu código en diversos casos para verificar su corrección y rendimiento.
* Considera la paralelización: Para mejorar la eficiencia, piensa en paralelizar la búsqueda exhaustiva usando bibliotecas como OpenMP o MPI si es posible.
* Documenta tu código: Documenta tu implementación para facilitar la comprensión y el mantenimiento del código.

**Algoritmos Voraces**

1. **Introducción**

Un algoritmo voraz, también conocido como algoritmo greedy, es una estrategia para resolver problemas complejos mediante la toma de decisiones localmente óptimas en cada paso. Aunque no siempre garantizan una solución globalmente óptima, estos algoritmos son populares por su simplicidad, eficiencia y capacidad para encontrar soluciones satisfactorias en muchos casos.

1. **Funcionamiento**

Los algoritmos voraces se basan en un ciclo iterativo que comprende tres etapas:

* **Evaluación:** En cada iteración, se evalúan las opciones disponibles en ese momento.
* **Selección:** Se elige la opción que parece ser la mejor según la estrategia voraz definida.
* **Actualización:** Se actualiza el estado del problema para reflejar la decisión tomada.

El ciclo se repite hasta que se alcanza un estado final que satisface las condiciones de terminación.

1. **Estrategias comunes**

Existen diferentes estrategias voraces que se pueden utilizar en la selección de la mejor opción en cada iteración. Algunas de las más comunes son:

* **El mejor ajuste**: Se elige la opción que mejor se ajusta a un criterio específico, como el menor costo, la mayor ganancia o la menor distancia.
* **El primero en llegar, el primero en ser atendido:** Se elige la opción que ha estado disponible durante más tiempo.
* **El más cercano:** Se elige la opción que está más cerca de un punto de referencia específico.

1. **Análisis de Algoritmos Voraces**

El análisis de un algoritmo voraz implica evaluar su eficiencia y su capacidad para encontrar soluciones óptimas.

* **Eficiencia:** La eficiencia de un algoritmo voraz se mide por su tiempo de ejecución y su complejidad espacial. En general, los algoritmos voraces son bastante eficientes y pueden ejecutarse en tiempo polinomial.
* **Optimalidad:** No todos los algoritmos voraces encuentran soluciones globalmente óptimas. La optimalidad de un algoritmo voraz depende de la naturaleza del problema y de la estrategia voraz utilizada.

1. **Aplicaciones de Algoritmos Voraces**

Los algoritmos voraces se utilizan en una amplia variedad de aplicaciones, incluyendo:

* **Ruteo:** Encontrar la ruta más corta entre dos puntos en un mapa.
* **Asignación:** Asignar tareas a recursos de manera eficiente.
* **Programación:** Seleccionar el mejor conjunto de tareas para ejecutar en un procesador.
* **Optimización:** Encontrar la mejor solución a un problema de optimización, como el problema del vendedor ambulante.

1. **Ejemplos de Algoritmos Voraces**

**Algoritmo de Dijkstra:** Es la ruta más corta entre un vértice y todos los otros vértices de un grafo.

**Divide y Vencerás.**

1. Definición de Divide y Vencerás.

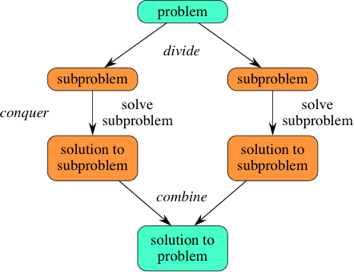
Divide y vencerás es un paradigma algorítmico (a veces llamado por error Divide y Concurrir - una adaptación en broma), similar a los paradigmas de programación Dinámica y Algoritmos ávidos o glotones. Un algoritmo Divide y Vencerás típico resuelve un problema siguiendo estos 3 pasos.

1. **Dividir:** Descomponer el problema en sub-problemas del mismo tipo. Este paso involucra descomponer el problema original en pequeños sub-problemas. Cada sub-problema debe representar una parte del problema original. Por lo general, este paso emplea un enfoque recursivo para dividir el problema hasta que no es posible crear un sub-problema más.
2. **Vencer:** Resolver los sub-problemas recursivamente. Este paso recibe un gran conjunto de sub-problemas a ser resueltos. Generalmente a este nivel, los problemas se resuelven por sí solos.
3. **Combinar:** Combinar las respuestas apropiadamente. Cuando los sub-problemas son resueltos, esta fase los combina recursivamente hasta que estos formulan la solución al problema original. Este enfoque algorítmico trabaja recursivamente y los pasos de conquista y fusión trabajan tan a la par que parece un sólo paso.

Usualmente este método nos permite hacer una reducción bastante significativa en la complejidad tiempo del algoritmo a emplear.

1. Técnica de Divide y Vencerás.

* Se divide el problema en subproblemas. Por eficiencia debemos intentar que los subproblemas tengan un tamaño similar. Si es posible resolver el problema, porque este es indivisible, se resuelve directamente.



*Imagen 1: Técnica Divide y vencerás.*

* Se resuelve el problema para cada una de las instancias más pequeñas.
* Se combinan las soluciones obtenidas para obtener una solución global.



*Imagen 2: Combinan las soluciones obtenidas para obtener una solución global*

1. Ejemplo:

En este ejemplo, la función encontrarMaximo divide la matriz en mitades más pequeñas, encuentra el máximo en cada mitad y luego combina los resultados para obtener el máximo global.

Divide Y Venceras.cpp

#include <iostream>

#include <vector>

// Función para encontrar el máximo elemento en una matriz

**int** encontrarMaximo(**const** std::vector<**int**>& arr, **int** izquierda, **int** derecha) {

**if** (izquierda == derecha) {

**return** arr.at(izquierda);

}

**else** {

**int** mitad = (izquierda + derecha) / 2;

**int** maxIzquierda = encontrarMaximo(arr, izquierda, mitad);

**int** maxDerecha = encontrarMaximo(arr, mitad + 1, derecha);

**return** std::max(maxIzquierda, maxDerecha);

}

}

**int** main() {

std::vector<**int**> numeros = { 10, 5, 8, 15, 3, 20, 12 };

**int** maximoElemento = encontrarMaximo(numeros, 0, numeros.size() - 1);

std::cout << "El maximo elemento es: " << maximoElemento << std::endl;

**return** 0;

}

1. Conclusiones.

* La técnica de "Divide y Vencerás" es efectiva para abordar problemas complejos al descomponerlos en subproblemas más simples.
* La fase de combinación es crucial para obtener la solución global de manera correcta y eficiente.
* El manejo adecuado de la recursividad es fundamental para evitar problemas como el desbordamiento de la pila de llamadas.

1. Recomendaciones.

* La técnica de "Divide y Vencerás" es efectiva para abordar problemas complejos al descomponerlos en subproblemas más simples.
* La fase de combinación es crucial para obtener la solución global de manera correcta y eficiente.
* El manejo adecuado de la recursividad es fundamental para evitar problemas como el desbordamiento de la pila de llamadas.

**Programación Dinámica**

1. **Que es la programación dinámica**

La programación dinámica es una técnica esencial de optimización utilizada para abordar problemas combinatorios y de toma de decisiones. Su importancia radica en su capacidad para descomponer problemas complejos en subproblemas más manejables y resolverlos de forma independiente. Esto no solo hace posible resolver problemas que de otro modo serían difíciles o incluso imposibles de abordar, sino que también permite una solución eficiente para problemas complejos como el recorrido del viajante, el problema de la mochila y la búsqueda de la subsecuencia común más larga. entre otros.

1. **Características**

* Almacenamiento de resultados previos: La programación dinámica almacena los resultados de los subproblemas previamente resueltos para evitar cálculos redundantes.
* Divide y vencerás: La técnica se basa en el análisis del problema en subproblemas más pequeños y manejables.
* Soluciones óptimas: La programación dinámica proporciona soluciones óptimas a los problemas, garantizando la mejor solución posible dentro del conjunto de todas las posibles soluciones.
* Problemas solapados: Los subproblemas en la programación dinámica se solapan, lo que significa que los mismos subproblemas se resuelven varias veces. Esto permite una optimización al evitar recalcular soluciones ya conocidas.

1. **Ventajas de la programación dinámica:**

* Eficiencia: La programación dinámica puede ofrecer soluciones eficientes para problemas complejos, reduciendo significativamente el tiempo de ejecución en comparación con otros enfoques.
* Optimalidad: Proporciona soluciones óptimas garantizadas para problemas de optimización, lo que significa que la solución encontrada es la mejor posible dentro del conjunto de todas las soluciones posibles.
* Reutilización de resultados: Almacenando y reutilizando los resultados de subproblemas previamente resueltos, se evita el cálculo redundante, lo que mejora aún más la eficiencia del algoritmo.
* Aplicabilidad: La técnica de programación dinámica es aplicable a una amplia gama de problemas combinatorios y de toma de decisiones, lo que la hace versátil y útil en diversas situaciones.

1. **Desventajas de la programación dinámica:**

Complejidad de implementación: La implementación de algoritmos de programación dinámica puede ser complicada y requerir un entendimiento profundo del problema y de la técnica en sí misma. Esto puede hacer que sea difícil para los programadores novatos.

Requisitos de memoria: Al almacenar los resultados de subproblemas previamente resueltos, la programación dinámica puede requerir una cantidad significativa de memoria, especialmente para problemas con muchos subproblemas.

Dificultad para identificar subproblemas: Identificar correctamente los subproblemas y definir la recurrencia puede ser desafiante en algunos casos, lo que puede dificultar la aplicación efectiva de la programación dinámica.

No siempre es la mejor opción: Aunque la programación dinámica puede proporcionar soluciones eficientes y óptimas para muchos problemas, no es la mejor opción para todos los problemas. En algunos casos, otros enfoques pueden ser más adecuados o más fáciles de implementar.**Principio del formulario**

1. **Problema de la mochila**

El problema de la mochila es un desafío clásico en optimización combinatoria que implica seleccionar la combinación óptima de objetos para maximizar el valor total, sin exceder la capacidad de una mochila dada. La técnica de programación dinámica es una forma eficaz de resolver este problema al descomponerlo en subproblemas más pequeños y reutilizar las soluciones parciales. La relación de recurrencia, que define cómo calcular el valor máximo posible para diferentes capacidades de mochila y diferentes conjuntos de objetos, se utiliza para construir una tabla de programación dinámica. Esta tabla almacena las soluciones parciales y se actualiza gradualmente durante la resolución del problema, lo que permite encontrar la solución óptima de manera eficiente.

* 1. **Código**

***Mochila.cpp***

#include <iostream>

#include <vector>

#include "validaciones.h"

**using** **namespace** std;

// Estructura de un objeto

**struct** Objeto {

**int** peso;

**int** valor;

};

// Función para calcular el máximo entre dos números

**int** max(**int** a, **int** b) {

**return** (a > b) ? a : b;

}

// Función que resuelve el problema de la mochila utilizando programación dinámica

**int** problemaMochila(**int** capacidad, vector<**int**>& pesos, vector<**int**>& valores, **int** n) {

vector<vector<**int**>> V(n + 1, vector<**int**>(capacidad + 1, 0));

**for** (**int** i = 1; i <= n; i++) {

**for** (**int** j = 1; j <= capacidad; j++) {

**if** (pesos[i - 1] <= j) {

V[i][j] = max(V[i - 1][j], valores[i - 1] + V[i - 1][j - pesos[i - 1]]);

} **else** {

V[i][j] = V[i - 1][j];

}

}

}

**return** V[n][capacidad];

}

**int** main() {

system("cls");

validaciones v;

**int** n;

// Validación para el número de objetos

**do** {

n = v.ingresar\_enteros("\nIngrese el numero de objetos (menos de 25): ");

**if** (n <= 0 || n >25) {

cout << "\nPor favor, ingrese un numero valido de objetos." << endl;

}

} **while** (n <= 0 || n > 25);

vector<**int**> pesos(n);

vector<**int**> valores(n);

// Ingreso de los pesos y valores de los objetos

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

cout << "\nObjeto " << i + 1 << endl;

// Validación para el peso del objeto

**do** {

pesos[i] = v.ingresar\_enteros("\nIngrese el peso del objeto (mayor que 0 y menor que 100): ");

**if** (pesos[i] <= 0 || pesos[i] >= 100) {

cout << "\nPor favor, ingrese un peso mayor que 0 y menor que 100" << endl;

}

} **while** (pesos[i] <= 0);

// Validación para el valor del objeto

**do** {

valores[i] = v.ingresar\_enteros("\nIngrese el valor del objeto (mayor que 0): ");

**if** (valores[i] <= 0) {

cout << "\nPor favor, ingrese un valor mayor que 0." << endl;

}

} **while** (valores[i] <= 0);

}

**int** capacidadMaxima;

capacidadMaxima = v.ingresar\_enteros("\nIngrese la capacidad maxima de la mochila: ");

// Resolución del problema de la mochila

**int** maxValor = problemaMochila(capacidadMaxima, pesos, valores, n);

cout << "\nEl valor maximo que se puede llevar en la mochila es: " << maxValor << endl;

system("pause");

**return** 0;

}

* 1. **Ejecución del código:**

A screenshot of a computer screen

Description automatically generated

*Figura 1. Ejecucicón de código de Mochila*

**Backtracking**

1. **Introducción**

El backtracking es una técnica algorítmica utilizada para encontrar soluciones a problemas computacionales, especialmente en situaciones donde se requiere buscar todas las posibles soluciones dentro de un espacio de búsqueda. Se basa en la idea de construir sistemáticamente todas las posibles configuraciones, descartando aquellas que no cumplen ciertas condiciones o restricciones, y retrocediendo (backtracking) cuando se llega a un punto donde ya no es posible seguir adelante sin violar alguna restricción.

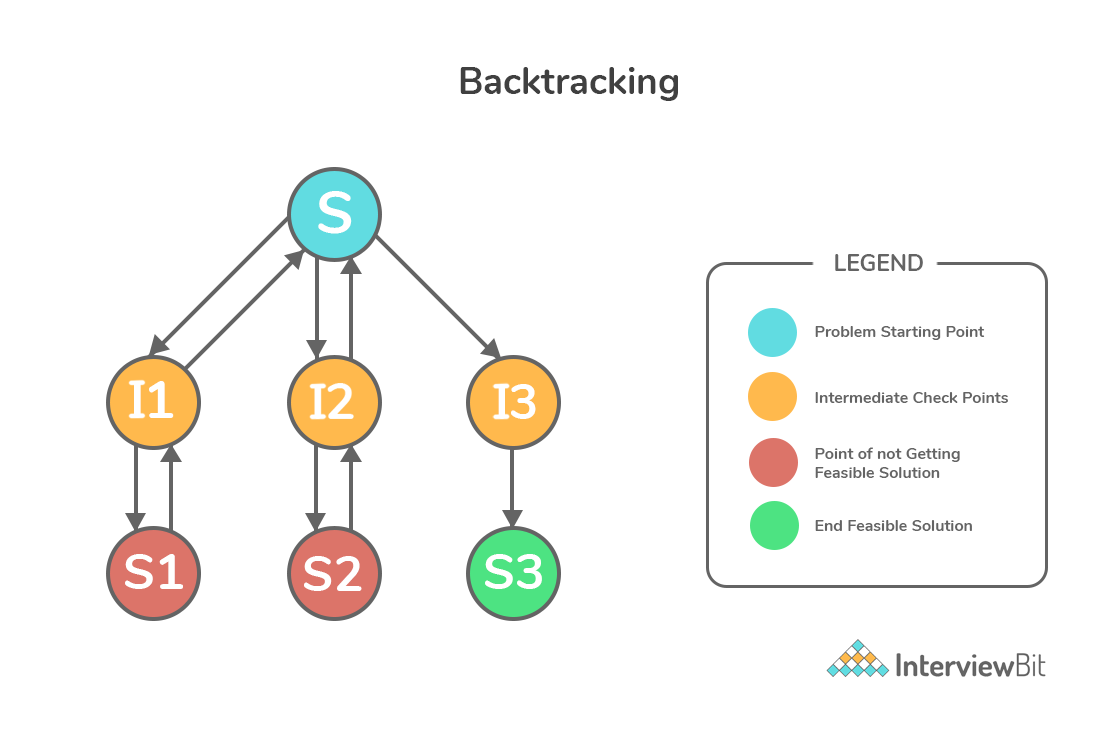
1. **Funcionamiento**

El algoritmo de retroceso normalmente funciona de la siguiente

manera:

* Elija una solución inicial y colóquela en la raíz del árbol de búsqueda.
* Examine la siguiente opción, seleccione una y agréguela a la solución actual.
* Si la solución actual es válida, pase a la siguiente opción y repita el segundo paso
* Si la solución actual no es válida, retroceda hasta el último punto de decisión y pruebe una opción diferente.

Continúe el proceso hasta que se encuentre una solución o se hayan examinado todas las soluciones posibles.



***Figura 1:*** *Funcionamiento del Backtracking*

1. **Ventajas**

* Permite encontrar todas las soluciones posibles a un problema.
* Es una técnica de resolución de problemas general que se puede aplicar a muchos tipos de problemas diferentes.
* El backtracking es una técnica fácil de entender y de implementar.
* El backtracking puede ser muy efectivo cuando se aplica correctamente.

1. **Desventajas**

* Puede ser ineficiente para problemas con un gran número de soluciones posibles.
* La complejidad temporal del backtracking puede ser muy alta en algunos casos, lo que hace que el algoritmo sea impráctico o incluso imposible de aplicar.
* Puede requerir una gran cantidad de memoria para almacenar el árbol de decisiones completo.
* No garantiza encontrar la solución óptima a un problema, a menos que se utilicen técnicas de poda y optimización adicionales.

1. **Análisis**

Complejidad temporal:

* La complejidad temporal del algoritmo de backtracking depende en gran medida del tamaño del espacio de búsqueda y de la cantidad de restricciones y condiciones que se aplican a cada paso.
* En el peor de los casos, donde se exploran todas las posibles combinaciones, la complejidad temporal puede ser exponencial, O(b^d), donde 'b' es el factor de ramificación (número de opciones disponibles en cada paso) y 'd' es la profundidad del árbol de decisiones (número de pasos).
* Sin embargo, con técnicas de poda y optimización adecuadas, la complejidad temporal puede reducirse significativamente.

Complejidad espacial:

* La complejidad espacial del algoritmo de backtracking también depende del tamaño del espacio de búsqueda y de la cantidad de memoria necesaria para almacenar las soluciones parciales y las restricciones.
* En el peor de los casos, la complejidad espacial puede ser lineal, O(d), donde 'd' es la profundidad del árbol de decisiones y representa la cantidad máxima de memoria necesaria para almacenar las soluciones parciales en un momento dado.

1. **Aplicaciones**

* Resolución de problemas de búsqueda exhaustiva: El backtracking se utiliza para encontrar todas las posibles soluciones a problemas donde se necesita explorar todo el espacio de búsqueda, como la generación de permutaciones, combinaciones, subconjuntos, etc.
* Problemas de optimización: En muchos problemas de optimización, el backtracking se puede utilizar para encontrar la mejor solución posible explorando exhaustivamente todas las posibles configuraciones y eligiendo la que optimice ciertas métricas o criterios.
* Problemas de satisfacción de restricciones (CSP): El backtracking se utiliza ampliamente en problemas de satisfacción de restricciones, donde se busca encontrar una solución que cumpla con un conjunto de restricciones o condiciones, como el problema de asignación de horarios, el problema de coloración de grafos, etc.
* Juegos y rompecabezas: El backtracking se utiliza en la resolución de juegos y rompecabezas, como el sudoku, el juego del 8, el ajedrez, etc., donde se busca encontrar una secuencia de movimientos que conduzca a una solución o victoria.
* Compiladores y análisis de código: En la compilación y el análisis de código, el backtracking se utiliza en la resolución de problemas de análisis sintáctico y semántico, como la generación de árboles de sintaxis abstracta, la resolución de conflictos de reducción y desplazamiento en el análisis LR, etc.

1. **Ejemplo – N Reinas**

Tablero.h

#pragma once

**class** **Tablero** {

**public:**

Tablero();

Tablero(**int**);

**void** **set\_TAM**(**int**);

**int** **get\_TAM**();

**void** **set\_piezas\_strings**(std::string\*\*\*);

std::string\*\*\* get\_piezas\_strings();

**void** **mostrar\_tablero**();

**bool** **esta\_ocupado**(**int**, **int**);

~Tablero();

**private:**

**int** TAM;

std::string\*\*\* piezas\_strings;

};

Tablero.cpp

#include <iostream>

#include "Tablero.h"

#include <iomanip>

Tablero::Tablero() {

TAM = **0**;

piezas\_strings = nullptr;

}

Tablero::Tablero(**int** \_TAM) {

//ENCERO MI MATRIZ DE PUNTEROS A strings CON DIRECCIONES NULAS

TAM = \_TAM;

piezas\_strings = (std::string\*\*\*)malloc(TAM \* **sizeof**(std::string\*\*));

**if** (piezas\_strings != nullptr) {

**for** (**int** i = **0**; i < TAM; i++) {

\*(piezas\_strings + i) = (std::string\*\*)malloc(TAM \* **sizeof**(std::string\*));

}

**for** (**int** i = **0**; i < TAM; i++) {

**if** (\*(piezas\_strings + i) != nullptr) {

**for** (**int** j = **0**; j < TAM; j++) {

\*(\*(piezas\_strings + i) + j) = **new** std::string("-");

}

}

}

}

}

**void** Tablero::set\_TAM(**int** \_TAM) {

TAM = \_TAM;

}

**int** Tablero::get\_TAM() {

**return** TAM;

}

**void** Tablero::set\_piezas\_strings(std::string\*\*\* \_piezas\_strings) {

piezas\_strings = \_piezas\_strings;

}

std::string\*\*\* Tablero::get\_piezas\_strings() {

**return** piezas\_strings;

}

**bool** Tablero::esta\_ocupado(**int** \_x, **int** \_y) {

**if** (\*(\*(piezas\_strings +\_x)+\_y) != nullptr) {

**return** true;

}

**return** false;

}

**void** Tablero::mostrar\_tablero() {

**if** (piezas\_strings != nullptr) {

**for** (**int** i = **0**; i < TAM; i++) {

**if** (\*(piezas\_strings + i) != nullptr) {

**for** (**int** j = **0**; j < TAM; j++) {

std::cout << std::setw(**10**) << \*(\*(\*(piezas\_strings + i) + j)) << std::setw(**10**);

}

}

printf("**\n**");

}

}

}

Tablero::~Tablero() {

}

Problema\_Reinas.h

#pragma once

#include "Tablero.h"

**bool** **validar\_posiciones**(**int**, **int**\*, **int**, **int**\*, **int**);

**void** **fun**(Tablero& tablero, **int** x, **int**\* \_x, **int**, **int**\*, **int**&);

**bool** **validar\_posiciones**(**int** x, **int**\* \_x, **int** y, **int**\* \_y, **int** TAM) {

**for** (**int** i = **0**; i < TAM; i++) {

**if** (y == \*(\_y + i) || std::abs((x - \*(\_x + i)) == std::abs(y - \*(\_y + i)))) {

**return** false;

}

}

**return** true;

}

**void** **fun**(Tablero& tablero, **int** x, **int**\* \_x, **int** y, **int**\* \_y, **int**& contador) {

**if** (x != tablero.get\_TAM() && y != tablero.get\_TAM()) {

**if** (validar\_posiciones(x, \_x, y, \_y, tablero.get\_TAM())) {

\*(\*(\*(tablero.get\_piezas\_strings() + x) + y)) = "Reina";

\*(\_x + x) = x;

\*(\_y + x) = y;

fun(tablero, x + **1**, \_x, **0**, \_y, contador);

**if** ((x+**1** != tablero.get\_TAM() && \*(\_y + tablero.get\_TAM() - **1**) == **10**)){

\*(\*(\*(tablero.get\_piezas\_strings() + x) + y)) = "-";

}

**else** {

contador++;

}

}

fun(tablero, x, \_x, y + **1**, \_y, contador);

**if** (\*(\_y + tablero.get\_TAM()-**1**) != **10**) {

tablero.mostrar\_tablero();

printf("SOL: %i, ------------------------------------------------------------------**\n**", contador);

}

\*(\*(\*(tablero.get\_piezas\_strings() + x) + y)) = "-";

\*(\_y + x) = **10**;

\*(\_x + x) = **20**;

}

}

N\_Reinas.cpp

#include <iostream>

#include <conio.h>

#include "windows.h"

#include "Menu.h"

#include "Tablero.h"

#include "validaciones.h"

#include "Problema\_Reinas.h"

**int** **main**()

{

**int** opcion = **1**;

**char** tecla;

**int** contador = **0**;

**int** TAM;

Tablero\* ajedrez = nullptr;

**int**\* \_x = nullptr;

**int**\* \_y = nullptr;

HANDLE consoleHandle = GetStdHandle(STD\_OUTPUT\_HANDLE);

CONSOLE\_CURSOR\_INFO cursorInfo;

GetConsoleCursorInfo(consoleHandle, &cursorInfo);

cursorInfo.bVisible = false;

SetConsoleCursorInfo(consoleHandle, &cursorInfo);

**do** {

mostrar(opcion);

tecla = \_getch();

**switch** (tecla) {

**case** **72**:

**if** (opcion > **1**) opcion--;

**break**;

**case** **80**:

**if** (opcion < **2**) opcion++;

**break**;

**case** **13**:

**if** (opcion == **1**) {

cursorInfo.bVisible = true;

SetConsoleCursorInfo(consoleHandle, &cursorInfo);

system("cls");

**do** {

TAM = ingresar\_enteros("Ingrese el Tamano: (4-9)");

printf("**\n**");

**if** (TAM < **4** || TAM > **9**) {

printf("Ese tamano no es valido. Intente de nuevo.**\n**");

}

**else** {

**break**;

}

} **while** (true);

ajedrez = **new** Tablero(TAM);

**if**(ajedrez->get\_piezas\_strings() != nullptr){

\_y = (**int**\*)malloc(TAM \* **sizeof**(**int**));

**if** (\_y != nullptr) {

**for** (**int** i = **0**; i < TAM; i++) {

\*(\_y + i) = **10**;

}

\_x = (**int**\*)malloc(TAM \* **sizeof**(**int**));

**if** (\_x != nullptr) {

**for** (**int** j = **0**; j < TAM; j++) {

\*(\_x + j) = **20**;

}

fun(\*ajedrez, **0**, \_x, **0**, \_y, contador);

}

}

}

**delete** ajedrez;

**delete** \_x;

**delete** \_y;

contador = **0**;

cursorInfo.bVisible = false;

SetConsoleCursorInfo(consoleHandle, &cursorInfo);

system("pause");

}

**else** **if** (opcion == **2**) {

system("cls");

std::cout << "Saliendo del programa." << std::endl;

exit(**0**);

}

**break**;

**default:**

**break**;

}

} **while** (true);

}

1. **Bibliografía**

* GeeksforGeeks. (n.d.). Backtracking algorithms. Retrieved March 10, 2023, from https://www.geeksforgeeks.org/backtracking-algorithms/
* Estructura de datos en C++, Luis Joyanes e Ignacio Zahonero.
* *LUDA UAM-Azc.* (2024). Aniei.org.mx. <http://aniei.org.mx/paginas/uam/CursoAA/curso_aa_01.html>

‌