Escuela Politécnica del Ejército

RECURSIVIDAD

Cuaderno de Estructuras de Datos

En el siguiente documento podrán encontrar temas como:

1. Algoritmos recursivos.
2. Casos en los que debe evitarse el uso de la recursividad .
3. Recursividad directa e indirecta.
   1. Metodos para la resolucion de problemas que usan recursividad .
   2. Divide y venceras.
   3. Backtracking(retroceso).
4. Ejemplos de Recursividad.

2023

Tercero   
Curso :14675

**RECURSIVIDAD.**

La recursividad (recursión) es aquella propiedad que posee una función por lo cual dicha función puede llamarse así misma. La recursión es una herramienta poderosa e importante en resolución de problemas y en programación. Se puede utilizar la recursividad como una alternativa a la iteración. Una solución recursiva es normalmente menos eficiente en términos de tiempo de computadora que una solución iterativa debido a las operaciones auxiliares que llevan consigo las llamadas suplementarias a las funciones; sin embargo, en muchas circunstancias el uso de la recursión permite a los programadores especificar soluciones naturales, sencillas, que serían, en caso contrario, difíciles de resolver. (Luis Aguilar, 2005)

1. **Algoritmos recursivos.**

Una función recursiva es aquella que se llama a si misma o bien directamente o bien a través de otra función. Una función que tiene sentencias entre las que se encuentra al menos una que llama a la propia función se dice que es recursiva. La definición de factorial de un número natural n se define como para todo número mayor que 0, y Una programación recursiva de la función factorial puede consultarse en el Ejercicio 1. En el diseño de un algoritmo. (Luis Aguilar, 2005)

* El instrumento necesario para expresar los programas recursivamente es la función.
* Una función tiene recursividad directa cuando se llama a sí misma dentro de su propia definición.
* Una función tiene recursividad indirecta si llama a otra función que, a su vez, contiene una referencia directa o indirecta a la primera.
* El cuerpo de una función recursiva debe tener una o varias instrucciones selectivas donde establece para los casos generales las soluciones generales, y para los casos triviales las soluciones triviales.
* Cada llamada recursiva debe disminuir el tamaño del ejemplar de entrada de tal manera que se acerque a las soluciones triviales.
* Cada vez que se llama a una función recursiva todos los valores de los parámetros formales y variables locales son almacenados en una pila.
* Cuando termine de ejecutarse la función recursiva retorna al nivel inmediatamente anterior. En la pila se recuperan los valores tanto de los parámetros como de las variables locales, y se continúa con la ejecución de la siguiente instrucción a la llamada recursiva.

1. **Casos en los que debe evitarse el uso de la recursividad .**
2. **Recursividad Directa e Indirecta**

**Directa:**

La recursividad puede considerarse de cabeza, cola o intermedia según que las operaciones se hagan después, antes o antes y después de la llamada recursiva. Por otra parte, también existe la recursividad múltiple, con diversas llamadas recursivas y la anidada.

**Ejemplo**

**Recursividad de Cabeza:** En este tipo de recursividad, las operaciones se realizan antes de la llamada recursiva.

#include <iostream>

void recursionDeCabeza(int n) {

if (n > 0) {

// Operaciones antes de la llamada recursiva

std::cout << n << " ";

// Llamada recursiva

recursionDeCabeza(n - 1);

}

}

int main() {

std::cout << "Recursividad de Cabeza: ";

recursionDeCabeza(5);

return 0;

}

En este ejemplo, se imprimen los números en orden descendente antes de realizar la llamada recursiva.

**Indirecta:**

La recursividad indirecta se produce cuando una función llama a otra, que eventualmente termina llamando de nuevo a la primera función.

#include <iostream>

void funcionB(int n); // Declaración anticipada

void funcionA(int n) {

if (n > 0) {

std::cout << n << " ";

funcionB(n - 1);

}

}

void funcionB(int n) {

if (n > 1) {

std::cout << n << " ";

funcionA(n / 2);

}

}

int main() {

std::cout << "Recursividad Indirecta: ";

funcionA(10);

return 0;

}

En este ejemplo, funcionA llama a funcionB, y luego funcionB llama de nuevo a funcionA, creando así un ciclo de llamadas recursivas indirectas.

1. **Metodos para la resolucion de problemas que usan recursividad .**

**Recursividad: Métodos para la Resolución de Problemas**

La recursividad es un concepto en programación donde una función se llama a sí misma para resolver un problema. La idea básica es dividir un problema en subproblemas más pequeños y resolver cada subproblema utilizando la misma función. La recursividad se utiliza para simplificar problemas complejos dividiéndolos en partes más pequeñas y manejables.

**Características de la Recursividad:**

1. **Caso Base:** Cada función recursiva debe tener un caso base que detenga la recursión. Es el resultado para el caso más simple del problema.
2. **Llamada Recursiva:** La función se llama a sí misma con argumentos diferentes, generalmente reduciendo la complejidad del problema.
3. **Problemas Divididos:** La solución al problema principal se construye combinando las soluciones de los subproblemas.

**Ejemplo de Recursividad: Factorial**

El factorial de un número nn, denotado como n!n!, se define como el producto de todos los enteros positivos hasta nn. Matemáticamente, n!=n×(n−1)×(n−2)×…×1n!=n×(n−1)×(n−2)×…×1.

En este ejemplo, la función factorial se llama a sí misma con un argumento reducido en 1 hasta que se alcanza el caso base (cuando nn es 0). La llamada recursiva simplifica el problema original al dividirlo en subproblemas más pequeños, proporcionando una solución concisa y elegante.

#include <iostream>

int factorial(int n) {

// Caso base: factorial de 0 es 1

if (n == 0) {

return 1;

}

// Llamada recursiva: n! = n \* (n-1)!

else {

return n \* factorial(n - 1);

}

}

int main() {

// Ejemplo de uso de la función factorial

int numero = 5;

std::cout << "El factorial de " << numero << " es: " << factorial(numero) << std::endl;

return 0;

}

En este ejemplo, la función factorial se llama a sí misma con un argumento reducido en 1 hasta que se alcanza el caso base (cuando nn es 0). La llamada recursiva simplifica el problema original al dividirlo en subproblemas más pequeños, proporcionando una solución concisa y elegante.

1. **Divide y venceras.**

**Divide y Vencerás: Métodos para la Resolución de Problemas**

Divide y Vencerás es una técnica de diseño de algoritmos que resuelve un problema dividiéndolo en subproblemas más pequeños y resolviendo cada subproblema de manera independiente. La solución al problema original se construye combinando las soluciones de los subproblemas.

**Características de Divide y Vencerás:**

1. **División del Problema:** Divide el problema original en subproblemas más pequeños y manejables.
2. **Resolución Recursiva:** Resuelve cada subproblema de manera recursiva.
3. **Combinación de Soluciones:** Combina las soluciones de los subproblemas para obtener la solución del problema original.

**Ejemplo de Divide y Vencerás: Merge Sort**

Un ejemplo clásico de algoritmo basado en la técnica Divide y Vencerás es el algoritmo de ordenación Merge Sort. Este algoritmo ordena una lista dividiéndola en mitades, ordenando cada mitad de manera recursiva y luego combinando las dos mitades ordenadas en una sola lista ordenada.

#include <iostream>

#include <vector>

void merge(std::vector<int>& arr, int izquierda, int medio, int derecha) {

// Combina dos subarreglos ordenados en uno solo

int n1 = medio - izquierda + 1;

int n2 = derecha - medio;

// Crear subarreglos temporales

std::vector<int> izquierdaTemp(n1);

std::vector<int> derechaTemp(n2);

// Copiar datos a los subarreglos temporales

for (int i = 0; i < n1; i++) {

izquierdaTemp[i] = arr[izquierda + i];

}

for (int j = 0; j < n2; j++) {

derechaTemp[j] = arr[medio + 1 + j];

}

// Mezclar los subarreglos temporales en el arreglo original

int i = 0;

int j = 0;

int k = izquierda;

while (i < n1 && j < n2) {

if (izquierdaTemp[i] <= derechaTemp[j]) {

arr[k] = izquierdaTemp[i];

i++;

} else {

arr[k] = derechaTemp[j];

j++;

}

k++;

}

// Copiar los elementos restantes de izquierdaTemp, si hay alguno

while (i < n1) {

arr[k] = izquierdaTemp[i];

i++;

k++;

}

// Copiar los elementos restantes de derechaTemp, si hay alguno

while (j < n2) {

arr[k] = derechaTemp[j];

j++;

k++;

}

}

void mergeSort(std::vector<int>& arr, int izquierda, int derecha) {

// Aplicar Merge Sort a un subarreglo

if (izquierda < derecha) {

// Encuentra el punto medio del subarreglo

int medio = izquierda + (derecha - izquierda) / 2;

// Ordena las dos mitades recursivamente

mergeSort(arr, izquierda, medio);

mergeSort(arr, medio + 1, derecha);

// Combina las dos mitades ordenadas

merge(arr, izquierda, medio, derecha);

}

}

void imprimirArreglo(const std::vector<int>& arr) {

// Imprimir el arreglo

for (int elemento : arr) {

std::cout << elemento << " ";

}

std::cout << std::endl;

}

int main() {

// Ejemplo de uso del algoritmo Merge Sort

std::vector<int> arreglo = {12, 11, 13, 5, 6, 7};

std::cout << "Arreglo original: ";

imprimirArreglo(arreglo);

// Aplicar Merge Sort

mergeSort(arreglo, 0, arreglo.size() - 1);

std::cout << "Arreglo ordenado: ";

imprimirArreglo(arreglo);

return 0;

}

En este ejemplo, la función mergeSort utiliza la técnica Divide y Vencerás para ordenar un arreglo. Divide el arreglo en dos mitades, ordena cada mitad de manera recursiva y luego combina las dos mitades ordenadas en un arreglo ordenado mediante la función merge. Este proceso continúa hasta que se alcanzan subarreglos de un solo elemento (caso base). La función imprimirArreglo se utiliza para mostrar el arreglo antes y después de aplicar Merge Sort.

1. **Backtracking (Retroceso): Métodos para la Resolución de Problemas**

Backtracking, o retroceso en español, es una técnica utilizada para resolver problemas que involucran la búsqueda de soluciones a través de la exploración de todas las posibilidades. Se basa en el principio de prueba y error, donde el algoritmo intenta construir una solución de manera incremental, retrocediendo cuando se encuentra en un punto donde la solución parcial no puede completarse a una solución válida.

**Características del Backtracking:**

1. **Construcción Incremental:** El algoritmo construye la solución paso a paso, evaluando en cada etapa si la solución parcial es viable.
2. **Retroceso (Backtrack):** Cuando una solución parcial no puede ser extendida a una solución válida, el algoritmo retrocede a la última decisión tomada y explora otra posibilidad.
3. **Exploración Exhaustiva:** El algoritmo explora todas las posibilidades antes de llegar a una solución óptima o una solución que cumple con ciertos criterios.

**Ejemplo de Backtracking: Problema de las N-Reinas**

Un problema clásico que se puede resolver utilizando backtracking es el problema de las N-Reinas. En este problema, se coloca un número de reinas en un tablero de ajedrez de manera que ninguna reina amenace a las demás.

#include <iostream>

#include <vector>

bool esSeguro(int fila, int col, const std::vector<int>& tablero) {

// Verifica si es seguro colocar una reina en la posición (fila, col)

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

if (tablero[i] == col || abs(tablero[i] - col) == abs(i - fila)) {

return false; // Conflicto en la misma columna o diagonal

}

}

return true;

}

void imprimirTablero(const std::vector<int>& tablero) {

// Imprime el tablero con las posiciones de las reinas

for (int fila : tablero) {

for (int i = 0; i < tablero.size(); ++i) {

if (i == fila) {

std::cout << "Q ";

} else {

std::cout << ". ";

}

}

std::cout << std::endl;

}

std::cout << std::endl;

}

void colocarReinas(int fila, int n, std::vector<int>& tablero) {

// Coloca reinas en el tablero utilizando backtracking

if (fila == n) {

// Se alcanzó una solución válida, imprimir tablero

imprimirTablero(tablero);

return;

}

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

if (esSeguro(fila, col, tablero)) {

// Coloca una reina y procede a la siguiente fila

tablero[fila] = col;

colocarReinas(fila + 1, n, tablero);

// Retrocede (backtrack) para explorar otras posibilidades

tablero[fila] = -1;

}

}

}

void resolverNReinas(int n) {

// Inicializa un tablero vacío

std::vector<int> tablero(n, -1);

// Comienza la colocación de reinas desde la primera fila

colocarReinas(0, n, tablero);

}

int main() {

int n = 8; // Número de reinas y tamaño del tablero

resolverNReinas(n);

return 0;

}

void colocarReinas(int fila, int n, std::vector<int>& tablero) {

// Coloca reinas en el tablero utilizando backtracking

if (fila == n) {

// Se alcanzó una solución válida, imprimir tablero

imprimirTablero(tablero);

return;

}

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

if (esSeguro(fila, col, tablero)) {

// Coloca una reina y procede a la siguiente fila

tablero[fila] = col;

colocarReinas(fila + 1, n, tablero);

// Retrocede (backtrack) para explorar otras posibilidades

tablero[fila] = -1;

}

}

}

void resolverNReinas(int n) {

// Inicializa un tablero vacío

std::vector<int> tablero(n, -1);

// Comienza la colocación de reinas desde la primera fila

colocarReinas(0, n, tablero);

}

int main() {

int n = 8; // Número de reinas y tamaño del tablero

resolverNReinas(n);

return 0;

}

En este ejemplo, colocarReinas es la función recursiva que realiza el backtracking para colocar las reinas en el tablero. La función esSeguro verifica si es seguro colocar una reina en una posición dada. La función principal resolverNReinas inicializa el tablero y comienza la resolución del problema.

Este programa imprimirá todas las soluciones posibles para el problema de las N-Reinas en un tablero de ajedrez de tamaño N. Cada solución muestra la ubicación de las reinas en el tablero sin amenazarse entre sí.

**Ejercicio 1.** La función suma impares emplea la recursividad para hallar la suma de los n primeros números enteros impares**.**

#include <iostream>

using namespace std;

double sumaimpares(int);

int main () {

int n;

cout << "Escriba un numero entero positivo: \n";

cin >> n;

cout << "Suma de los " << n << " primeros impares: " << sumaimpares(n) << endl;

return 0;

}

double sumaimpares(int n) {

if (n <= 1)

return 1.0;

return ((2 \* n - 1) + sumaimpares(n - 1));

}

# **Bibliografía**

Luis Aguilar, M. F. (2005). *Estructuras de datos en C.* Madrid: McGraw-Hill.

*mergeSort. (2023). In Wikipedia. Retrieved December 6, 2023, from https://en.wikipedia.org/wiki/Merge\_sort.*