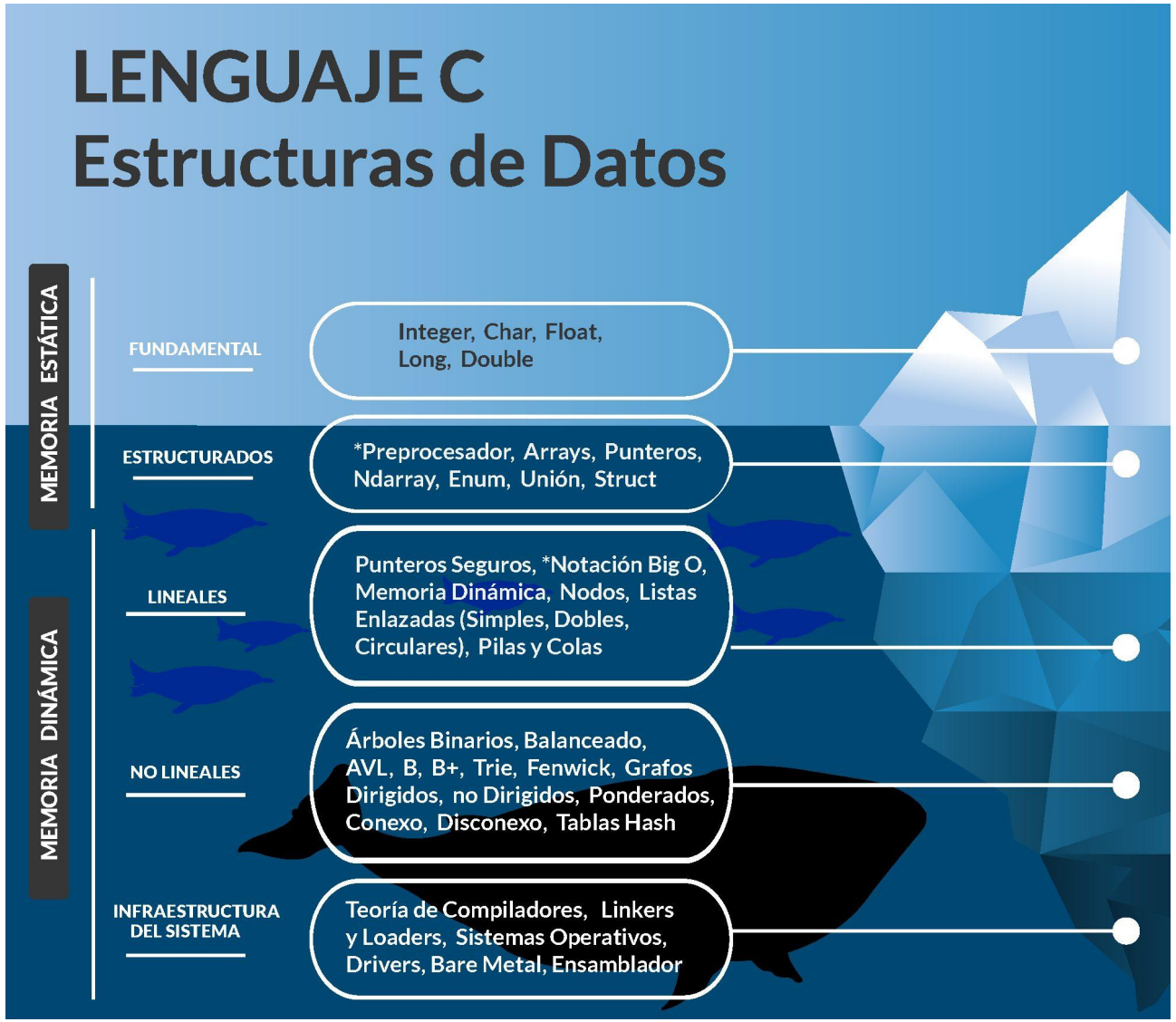
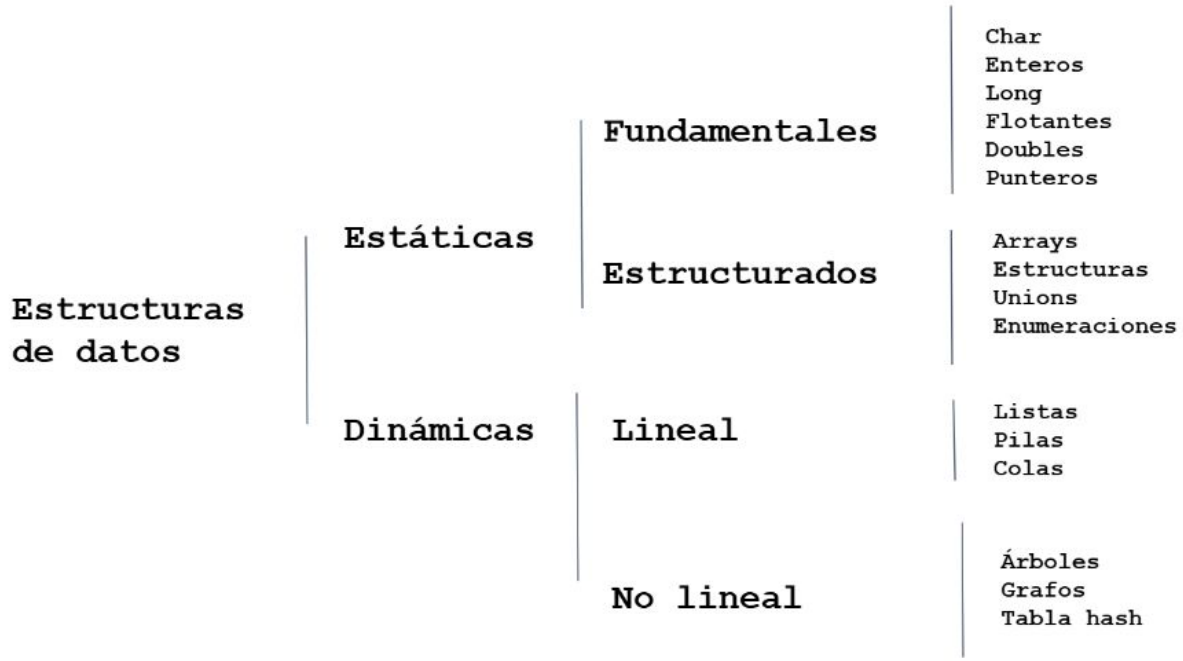
ESTRUCTURAS DE DATOS LINEALES

Las **estructuras de datos** son formas de organizar y almacenar datos en la memoria para que puedan ser utilizados y manipulados eficientemente.



ESTRUCTURAS DE DATOS LINEALES

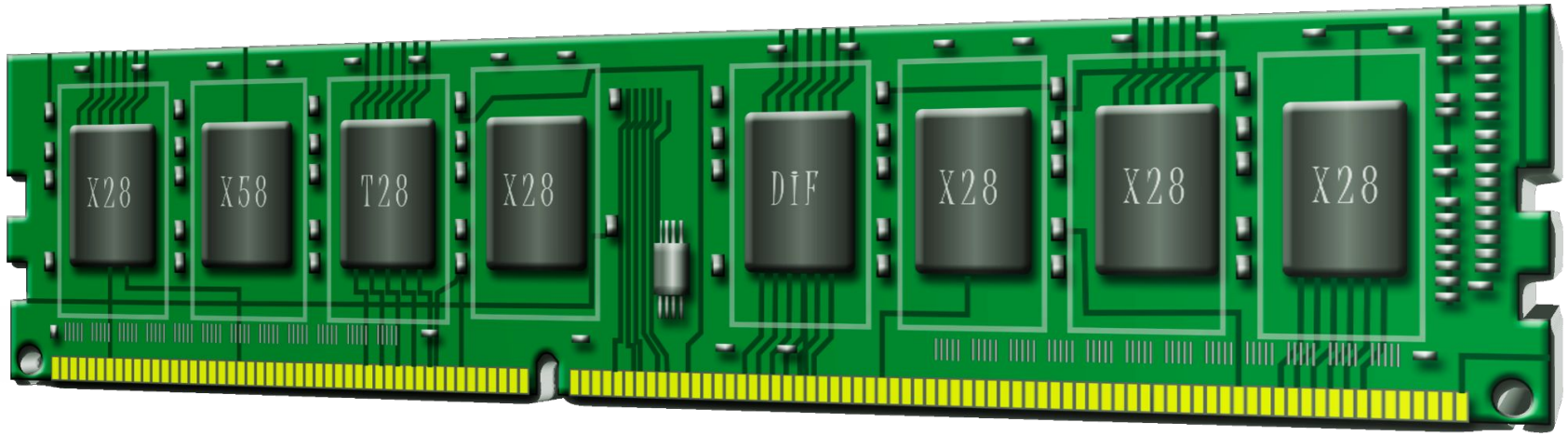
El lenguaje C proporciona herramientas tanto para estructuras de datos primitivas (como enteros, flotantes, caracteres, etc.) como para estructuras de datos más complejas, como **listas enlazadas**, **pilas**, **colas**, **árboles**, **grafos**, y muchas otras.

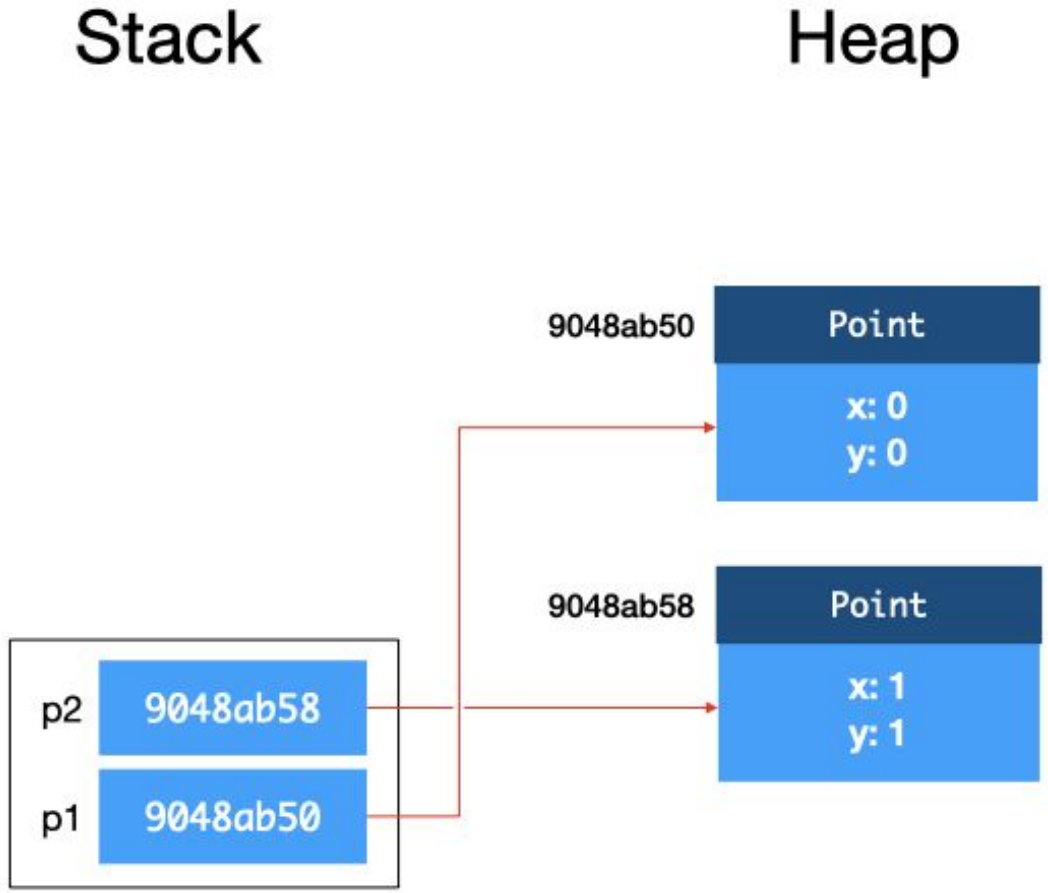


RAM

La memoria **RAM** (Memoria de acceso aleatorio) es donde un programa almacena datos e instrucciones para ser procesados. Los datos en la **RAM** son volátiles, lo que significa que se pierden cuando el sistema se apaga.

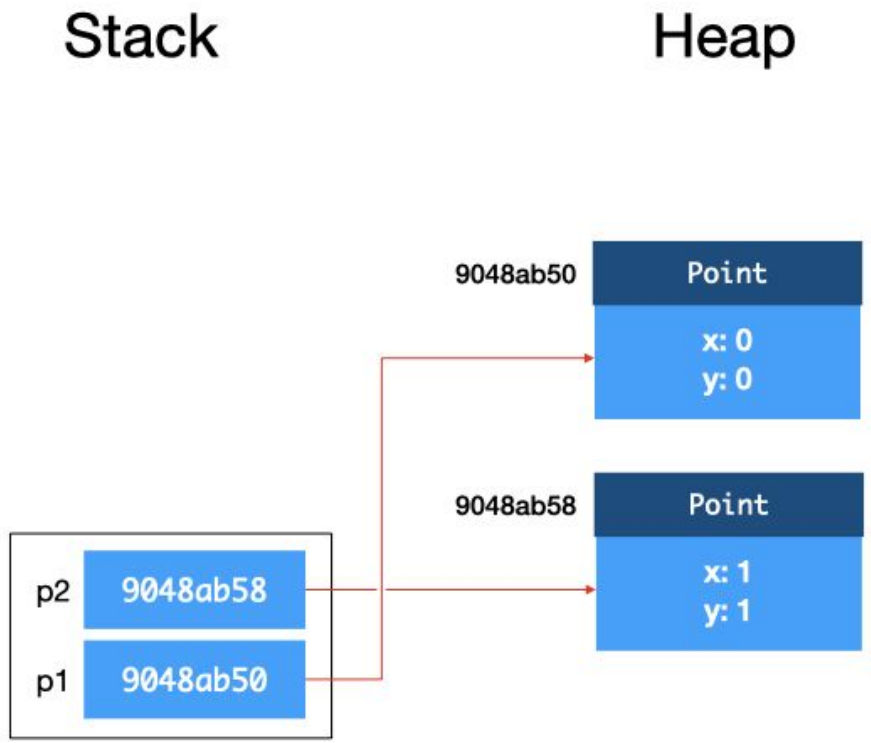
La **RAM** se organiza en diferentes segmentos, de los cuales dos de los más importantes son la **memoria stack** y la **memoria heap**.




Stack
Heap
9048ab50
Point
x: 0
y: 0
9048ab58
Point
x: 1
y: 1
p2
9048ab58
p1
9048ab50

STACK (Pila)

La **memoria stack** es una región de la memoria que almacena **datos temporales** relacionados con la ejecución de funciones. La **pila** sigue una estructura de tipo **LIFO** (Last In, First Out), lo que significa que el último elemento que se agrega es el primero en salir.



STACK (Pila)

La **memoria stack** es una región de la memoria que almacena **datos temporales** relacionados con la ejecución de funciones. La **pila** sigue una estructura de tipo **LIFO** (Last In, First Out), lo que significa que el último elemento que se agrega es el primero en salir.

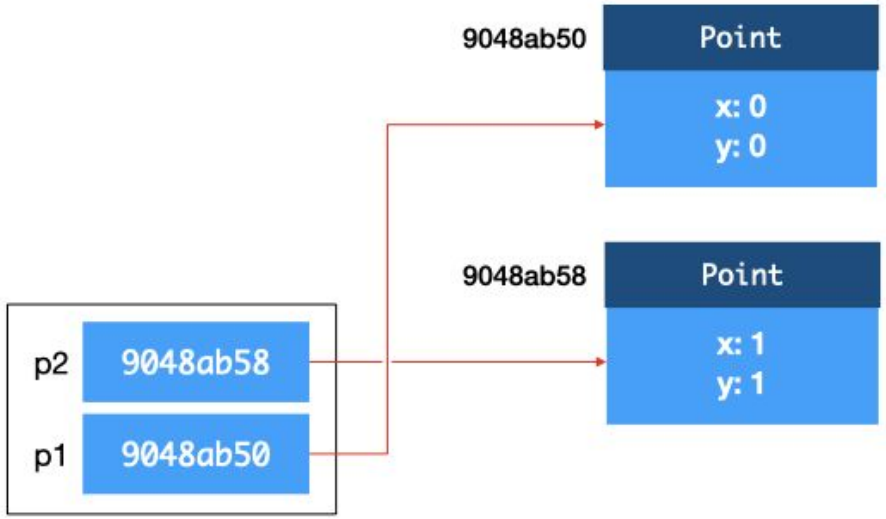
```c
void funcionEjemplo() {
    int x = 10;  // 'x' se almacena en la memoria stack
    printf("%d", x);
}

int main() {
    funcionEjemplo();  // Aquí ocurre una llamada a la función
    return 0;
}
```

STACK (Pila)

Características

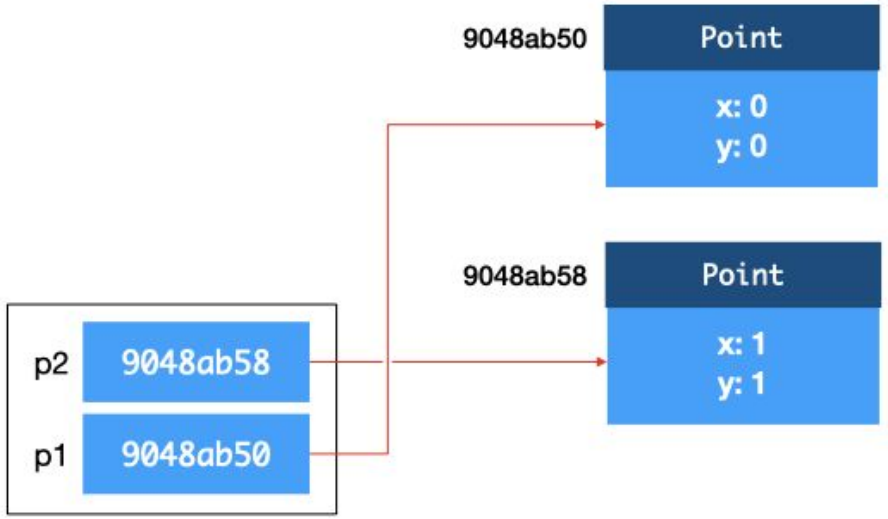
- Acceso rápido
- Tamaño fijo
- Limitación de espacio
- Duración y visibilidad
- Uso de punteros en la stack



STACK (Pila)

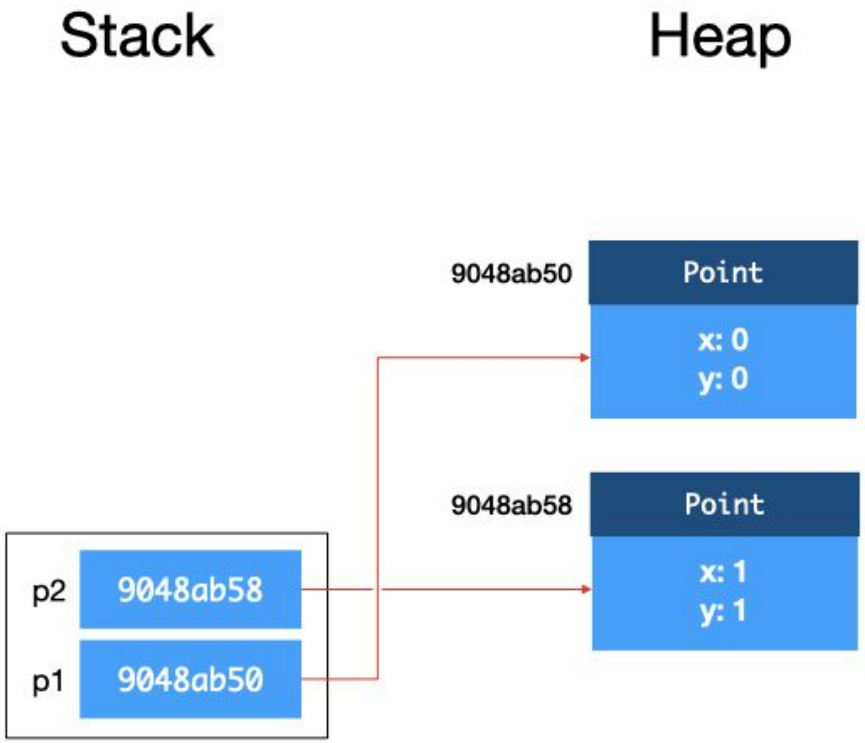
Limitaciones

- Tamaño fijo
- Recursión



HEAP

La **memoria heap** es una región de la memoria donde se almacenan objetos o bloques de memoria que tienen una duración más flexible, es decir, su vida no está limitada a la duración de una función, sino que pueden ser gestionados manualmente por el programador o el sistema.



HEAP

Características

- Almacenamiento dinámico
- Gestión manual de la memoria
- Más lenta que la stack
- No tiene límites fijos de tamaño
- Posible fragmentación

```c
#include <stdlib.h>

int main() {
    int* ptr = (int*) malloc(sizeof(int));  // Asignar memoria en el heap
    *ptr = 10;  // Usar la memoria asignada
    printf("%d", *ptr);

    free(ptr);  // Liberar la memoria del heap
    return 0;
}
```

DIFERENCIAS

Característica	Stack	Heap
Gestión de memoria	Automática(el compilador se encarga)	Manual(el programador o el recolector de basura se encarga)
Tamaño	Limitado y pequeño	Más grande limitado por la memoria física disponible
Acceso	Más rápido(LIFO)	Más lento debido a la asignación dinámica
Duración	Duración limitada(durante la ejecución de la función)	Duración controlada por el programador(hasta que se libera)
Fragmentación	No hay fragmentación	Puede haber fragmentación de memoria
Seguridad	Más seguro (por la gestión automática)	Más propenso a errores(por la gestión manual)

ESTRUCTURAS DE DATOS LINEALES

ESTRUCTURAS DE DATOS LINEALES

- Lista simplemente enlazada
- Lista doblemente enlazada
- Lista circular simplemente enlazada
- Lista circular doblemente enlazada
- Pilas y Colas simples
- Pilas y Colas circulares

Notación Big-O

La **notación Big-O** describe cómo el tiempo de ejecución o el uso de memoria de un algoritmo crece en función del tamaño de entrada n. Es una medida de la eficiencia en el peor de los casos, ignorando constantes y factores menores.

Principales categorías

O(1)- Tiempo constante

- El tiempo de ejecución no depende del tamaño de la entrada.
- Ejemplo: Acceder a un elemento específico en un arreglo.

```
int getFirstElement(int arr[], int size) {
    return arr[0]; // Siempre tarda lo mismo, sin importar el tamaño del arreglo.
}
```

Principales categorías

O(logn) - Tiempo logarítmico

- El tiempo de ejecución crece logarítmicamente con la entrada, típico en algoritmos de búsqueda binaria.
- Ejemplo: Búsqueda binaria en un arreglo ordenado.

```c
int binarySearch(int arr[], int size, int key) {
    int low = 0, high = size - 1;
    while (low <= high) {
        int mid = (low + high) / 2;
        if (arr[mid] == key)
            return mid;
        else if (arr[mid] < key)
            low = mid + 1;
        else
            high = mid - 1;
    }
    return -1; // Elemento no encontrado
}
```

Principales categorías

O(n) - Tiempo lineal

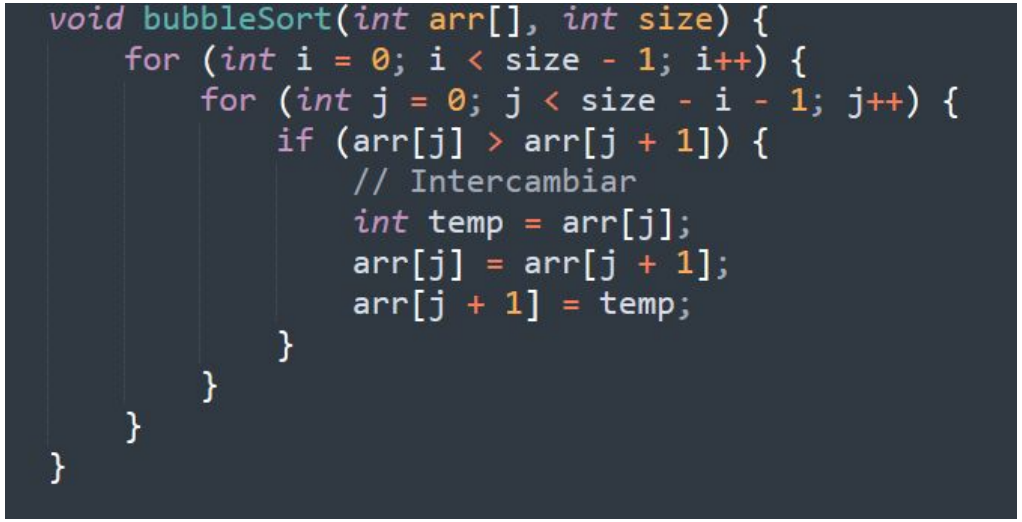
- El tiempo de ejecución crece proporcionalmente con el tamaño de la entrada.
- Ejemplo: Buscar un elemento en un arreglo no ordenado.

```
int linearSearch(int arr[], int size, int key) {
    for (int i = 0; i < size; i++) {
        if (arr[i] == key)
            return i;
    }
return -1; // Elemento no encontrado
}
```

Principales categorías

O(n2) Tiempo cuadrático

- Ocurre en algoritmos con bucles anidados, como el **Bubble Sort.**
- Ejemplo: Ordenamiento de burbuja.



```
void bubbleSort(int arr[], int size) {
    for (int i = 0; i < size - 1; i++) {
        for (int j = 0; j < size - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                // Intercambiar
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}
```

Principales categorías

O(2n) - Tiempo exponencial

- Común en problemas que requieren resolver todas las combinaciones posibles, como el problema de la **torre de Hanoi** o algunos algoritmos de backtracking.
- Ejemplo: Resolución de la Torre de Hanoi.

```c
void hanoi(int n, char from, char to, char aux) {
    if (n == 1) {
        printf("Mover disco 1 de %c a %c\n", from, to);
        return;
    }
    hanoi(n - 1, from, aux, to);
    printf("Mover disco %d de %c a %c\n", n, from, to);
    hanoi(n - 1, aux, to, from);
}
```