Informe de Algoritmos de Ordenamiento y Búsqueda en C

Henry B. Guerrero

17 de noviembre de 2025

Resumen

Este documento presenta una descripción detallada de varios algoritmos clásicos de ordenamiento y búsqueda implementados en lenguaje C, con fines didácticos para un curso de Programación Básica. Se analizan los algoritmos Bubble Sort, Selection Sort, Insertion Sort, Quick Sort y Merge Sort, así como los métodos de búsqueda lineal y búsqueda binaria (con manejo de elementos repetidos). Para cada algoritmo se discuten la idea principal, su implementación general en C y su complejidad computacional. Finalmente, se presentan comparaciones y sugerencias de actividades para los estudiantes.

1. Introducción

El estudio de algoritmos de ordenamiento y búsqueda constituye un pilar fundamental en los cursos de programación y estructuras de datos. Estos algoritmos permiten al estudiante comprender cómo organizar y localizar información de forma eficiente, además de introducir conceptos clave como *recorridos*, *intercambios*, *recursividad* y *complejidad temporal*.

En este informe se documentan y explican los códigos desarrollados en C para un conjunto de algoritmos de ordenamiento y búsqueda, organizados en dos carpetas:

- **Sort**/: contiene los algoritmos de ordenamiento Bubble Sort, Selection Sort, Insertion Sort, Quick Sort y Merge Sort.
- **Search**/: contiene los algoritmos de búsqueda lineal y búsqueda binaria, incluyendo variantes para manejar elementos repetidos.

El objetivo principal es ofrecer un material de referencia que acompañe las prácticas de laboratorio, de modo que el estudiante pueda relacionar la teoría del aula con la implementación concreta en código C.

2. Elementos comunes en los programas

Antes de analizar cada algoritmo es conveniente resaltar algunos elementos que aparecen de forma recurrente en los distintos archivos:

2.1. Inclusión de bibliotecas y función `main`

Todos los programas incluyen la biblioteca estándar de entrada/salida:

```
1 #include <stdio.h>
```

Esto permite utilizar las funciones `printf` y `scanf` para interactuar con el usuario. Además, cada archivo contiene una función `main` que declara un arreglo de ejemplo, invoca el algoritmo correspondiente y muestra los resultados en pantalla.

2.2. Cálculo del tamaño del arreglo

En C, el tamaño de un arreglo estático se calcula normalmente mediante:

```
int n = sizeof(arr) / sizeof(arr[0]);
```

donde `arr` es el arreglo y `arr[0]` es su primer elemento. Esta expresión divide el número total de bytes que ocupa el arreglo entre los bytes que ocupa un único elemento, proporcionando así el número de elementos.

2.3. Función auxiliar para impresión de arreglos

En varios archivos se define una función auxiliar para imprimir el contenido del arreglo:

```
void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++)
        printf("%d␣", arr[i]);
    printf("\n");
}
```

Esta función se utiliza antes y después de ejecutar los algoritmos de ordenamiento para visualizar el efecto de cada método.

3. Algoritmos de ordenamiento

En la carpeta `Sort` se implementan cinco algoritmos clásicos de ordenamiento: Bubble Sort, Selection Sort, Insertion Sort, Quick Sort y Merge Sort. A continuación se describe la idea principal y la implementación general de cada uno.

3.1. Bubble Sort (`bubble1.c`)

Idea del algoritmo

Bubble Sort recorre el arreglo repetidas veces, comparando elementos adyacentes e intercambiándolos cuando están en orden incorrecto. Tras cada pasada, el elemento más grande "sube" hacia el final del arreglo, de forma análoga a una burbuja en un líquido. Después de un número suficiente de pasadas, el arreglo queda ordenado.

Estructura del código

La función típica de Bubble Sort tiene la siguiente forma:

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

El bucle externo controla el número de pasadas y el bucle interno realiza las comparaciones e intercambios entre elementos adyacentes. La expresión `n - 1 - i` evita revisar la parte final del arreglo que ya ha quedado ordenada.

Complejidad

En el peor caso, Bubble Sort realiza un número de comparaciones proporcional a n^2, donde n es el número de elementos del arreglo. Su complejidad temporal es, por tanto, $O(n^2)$.

3.2. Selection Sort (`selection1.c`)

Idea del algoritmo

Selection Sort ordena el arreglo seleccionando, en cada iteración, el elemento mínimo de la parte no ordenada y colocándolo en la posición correcta al inicio. El algoritmo mantiene dos zonas: una porción inicial ya ordenada y otra porción restante no ordenada.

Estructura del código

```
void selectionSort(int arr[], int n) {
    int i, j, minIndex, temp;
    for (i = 0; i < n - 1; i++) {
        minIndex = i;
        for (j = i + 1; j < n; j++) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }
        temp          = arr[i];
        arr[i]        = arr[minIndex];
        arr[minIndex]= temp;
    }
}
```

El índice `minIndex` almacena la posición del menor elemento encontrado en la parte no ordenada. Al final de cada iteración, se intercambia con la posición `i`.

Complejidad

Selection Sort realiza, aproximadamente, $n(n-1)/2$ comparaciones independientemente del orden inicial de los datos. Su complejidad temporal es $O(n^2)$.

3.3. Insertion Sort (`insertion1.c`)

Idea del algoritmo

Insertion Sort construye la parte ordenada del arreglo de manera incremental: considera que el primer elemento está ordenado y luego inserta cada nuevo elemento en la posición correcta dentro de la porción ya ordenada. La analogía clásica es el ordenamiento de cartas en la mano.

Estructura del código

```
void insertionSort(int arr[], int n) {
    int i, key, j;
    for (i = 1; i < n; i++) {
        key = arr[i];
        j   = i - 1;
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
```

```
        arr[j + 1] = key;
    }
}
```

El elemento `key` representa el valor que se insertará en la porción ordenada. El bucle `while` desplaza a la derecha aquellos elementos mayores que `key`, creando el espacio para insertarlo.

Complejidad

En el peor caso (arreglo inicialmente ordenado en orden inverso), Insertion Sort tiene complejidad $O(n^2)$. Sin embargo, cuando el arreglo está casi ordenado, el número de desplazamientos es mucho menor y el algoritmo se comporta de forma cercana a $O(n)$.

3.4. Quick Sort (`quickSort.c`)

Idea del algoritmo

Quick Sort es un algoritmo de tipo *divide y vencerás*. El procedimiento general es:

1. Elegir un elemento del arreglo como *pivote*.
2. Reorganizar el arreglo de modo que los elementos menores que el pivote queden a su izquierda, y los mayores o iguales, a su derecha.
3. Aplicar recursivamente el mismo proceso a las subpartes izquierda y derecha.

Funciones principales

Se utiliza una función auxiliar `swap` para intercambiar elementos:

```
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

La función de partición suele tomar esta forma:

```
int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i     = low - 1;

    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return i + 1;
}
```

La función principal `quickSort` llama recursivamente a la partición:

```
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low,     pi - 1);
        quickSort(arr, pi + 1,  high);
```

```
    }
}
```

Complejidad

En promedio, Quick Sort tiene complejidad $O(n \log n)$, lo que lo hace muy eficiente para arreglos grandes. En el peor caso (por ejemplo, si el pivote siempre es el mínimo o el máximo) la complejidad se degrada a $O(n^2)$.

3.5. Merge Sort (`mergeSort.c`)

Idea del algoritmo

Merge Sort es otro algoritmo *divide y vencerás*, pero con una estrategia diferente:

1. Dividir el arreglo en dos mitades.
2. Ordenar recursivamente cada mitad.
3. Mezclar (*merge*) las dos mitades ordenadas en un único arreglo ordenado.

Funciones principales

La función `merge` mezcla dos subarreglos ordenados:

```
void merge(int arr[], int l, int m, int r) {
    int n1 = m - l + 1;
    int n2 = r - m;

    int L[n1], R[n2];

    for (int i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    int i = 0, j = 0, k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }
    while (i < n1) { arr[k++] = L[i++]; }
    while (j < n2) { arr[k++] = R[j++]; }
}
```

La función recursiva `mergeSort` es:

```
void mergeSort(int arr[], int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2;
        mergeSort(arr, l,     m);
        mergeSort(arr, m + 1, r);
```

```
        merge(arr, l, m, r);
    }
}
```

Complejidad

Merge Sort realiza siempre un número de operaciones proporcional a $O(n \log n)$, independientemente del orden inicial del arreglo. Requiere memoria adicional para los arreglos temporales `L` y `R`, pero ofrece un comportamiento muy estable.

4. Algoritmos de búsqueda

En la carpeta `Search` se implementan dos tipos de búsqueda: búsqueda lineal y búsqueda binaria. La primera no requiere que el arreglo esté ordenado, mientras que la segunda exige un arreglo ordenado para funcionar correctamente.

4.1. Búsqueda lineal (`lineal1.c`)

Idea del algoritmo

La búsqueda lineal recorre el arreglo desde la primera hasta la última posición, comparando uno a uno los elementos con la clave buscada. Es el método más sencillo de búsqueda y no requiere ningún tipo de orden previo en los datos.

Primera aparición

La función básica para hallar la primera aparición de la clave es:

```
int linearSearch(int arr[], int n, int key) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == key)
            return i;
    }
    return -1;
}
```

Todas las apariciones

La variante que encuentra todas las apariciones utiliza un arreglo adicional para almacenar los índices:

```
int linearSearchAll(int arr[], int n, int key, int indices[]) {
    int count = 0;
    for (int i = 0; i < n; i++) {
        if (arr[i] == key) {
            indices[count] = i;
            count++;
        }
    }
    return count;
}
```

Complejidad

La complejidad temporal de la búsqueda lineal en el peor caso es $O(n)$, ya que puede ser necesario revisar todos los elementos del arreglo.

4.2. Búsqueda binaria (`binaria1.c`)

Idea general

La búsqueda binaria se aplica sobre arreglos ordenados. En lugar de recorrer todos los elementos, compara la clave con el elemento central y decide si debe continuar la búsqueda en la mitad izquierda o en la mitad derecha del arreglo. De esta forma, reduce el espacio de búsqueda a la mitad en cada paso.

Primera y última aparición

En el archivo `binaria1.c` se implementan dos variantes:

- `firstOccurrence`: encuentra la primera posición donde aparece la clave.
- `lastOccurrence`: encuentra la última posición de la clave.

Ambas funciones utilizan el esquema de búsqueda binaria, pero al encontrar la clave no se detienen inmediatamente, sino que continúan buscando hacia la izquierda o hacia la derecha según el caso.

```
int firstOccurrence(int arr[], int n, int key) {
    int low = 0, high = n - 1, result = -1;
    while (low <= high) {
        int mid = (low + high) / 2;
        if (arr[mid] == key) {
            result = mid;
            high   = mid - 1; // seguir a la izquierda
        } else if (key < arr[mid]) {
            high = mid - 1;
        } else {
            low  = mid + 1;
        }
    }
    return result;
}
```

```
int lastOccurrence(int arr[], int n, int key) {
    int low = 0, high = n - 1, result = -1;
    while (low <= high) {
        int mid = (low + high) / 2;
        if (arr[mid] == key) {
            result = mid;
            low    = mid + 1; // seguir a la derecha
        } else if (key < arr[mid]) {
            high = mid - 1;
        } else {
            low  = mid + 1;
        }
    }
    return result;
}
```

El programa principal combina ambas para calcular el número total de apariciones de la clave en un arreglo ordenado con elementos repetidos.

Complejidad

Cada llamada a `firstOccurrence` o `lastOccurrence` realiza un número de pasos proporcional a $\log_2(n)$. Por tanto, la complejidad de la búsqueda binaria es $O(\log n)$, muy superior a la búsqueda lineal cuando el tamaño del arreglo es grande.

5. Comparación y discusión

La Tabla 1 resume las complejidades de los algoritmos de ordenamiento implementados, mientras que la Tabla 2 presenta un resumen para los algoritmos de búsqueda.

Algoritmo	**Archivo**	**Complejidad peor caso**
Bubble Sort	`bubble1.c`	$O(n^2)$
Selection Sort	`selection1.c`	$O(n^2)$
Insertion Sort	`insertion1.c`	$O(n^2)$
Quick Sort	`quickSort.c`	$O(n^2)$ (promedio $O(n \log n)$)
Merge Sort	`mergeSort.c`	$O(n \log n)$

Cuadro 1: Resumen de complejidades de los algoritmos de ordenamiento.

Algoritmo	**Archivo**	**Complejidad peor caso**
Búsqueda lineal	`lineal1.c`	$O(n)$
Búsqueda binaria (1ª y última)	`binaria1.c`	$O(\log n)$

Cuadro 2: Resumen de complejidades de los algoritmos de búsqueda.

Desde el punto de vista didáctico, los algoritmos cuadráticos (Bubble, Selection e Insertion Sort) son muy útiles para introducir la idea de ordenamiento y el uso de ciclos anidados, a pesar de no ser eficientes para grandes volúmenes de datos. Por su parte, Quick Sort y Merge Sort permiten discutir recursividad, estrategias de *divide y vencerás* y notación $O(n \log n)$.

En cuanto a la búsqueda, la comparación entre búsqueda lineal y binaria es especialmente ilustrativa: ambos algoritmos resuelven el mismo problema, pero la búsqueda binaria es órdenes de magnitud más rápida cuando el arreglo es grande y está ordenado.

6. Conclusiones

En este informe se ha presentado una descripción detallada de varios algoritmos fundamentales de ordenamiento y búsqueda implementados en C. El análisis ha cubierto tanto la idea conceptual de cada método como su traducción a código y su complejidad computacional.

Este conjunto de programas sirve como base para las prácticas de programación en cursos introductorios, permitiendo al estudiante:

- Relacionar la teoría de algoritmos con implementaciones concretas.
- Comprender el impacto del orden de los datos en el rendimiento.
- Experimentar con modificaciones, impresiones intermedias y mediciones de tiempo.

Como trabajo futuro, se pueden incorporar otros algoritmos (por ejemplo, Shell Sort o Heap Sort), así como ejemplos de búsqueda en estructuras dinámicas (listas enlazadas, árboles de búsqueda binaria o tablas hash), ampliando el panorama hacia cursos de estructuras de datos.