QUICKSORT, MERGE SORT Y HEAP SORT

Funcionamiento, lógica e implementación en lenguaje C

Material didáctico para estudiantes de Estructuras de Datos. El objetivo es comprender la lógica de cada algoritmo de ordenamiento antes de analizar su implementación.

1. ¿Qué es un algoritmo de ordenamiento?

Un algoritmo de ordenamiento reorganiza los elementos de una colección siguiendo un criterio. En este material se utiliza el orden ascendente. Por ejemplo, el vector **[45, 12, 78, 34, 23]** debe convertirse en **[12, 23, 34, 45, 78]**.

Los tres algoritmos estudiados utilizan estrategias diferentes: Quick Sort se basa en la partición alrededor de un pivote; Merge Sort utiliza división y combinación de subproblemas; Heap Sort utiliza una estructura de montículo o heap.

2. Quick Sort

Idea fundamental

Quick Sort utiliza la estrategia de **divide y vencerás**. Selecciona un elemento llamado **pivote** y reorganiza el vector de modo que los elementos menores queden a la izquierda y los mayores a la derecha. Después aplica el mismo procedimiento de forma recursiva a cada parte.

Lógica paso a paso

1. Seleccionar un pivote. En la implementación presentada se utiliza el último elemento.

2. Recorrer los elementos anteriores al pivote.

3. Colocar a la izquierda los elementos menores que el pivote.

4. Colocar el pivote en su posición definitiva.

5. Aplicar Quick Sort al subvector izquierdo.

6. Aplicar Quick Sort al subvector derecho.

7. Detener la recursión cuando el subvector tenga cero o un elemento.

Ejemplo conceptual

Vector: [45, 12, 78, 34, 23, 9, 56]. Si el pivote es 56, los elementos menores se llevan hacia la izquierda y los mayores hacia la derecha. Una vez ubicado 56, ninguna operación posterior necesita moverlo porque ya está en su posición relativa correcta.

Pseudocódigo

```
QUICKSORT(A, inicio, fin)
    si inicio < fin:
        p = PARTICION(A, inicio, fin)
        QUICKSORT(A, inicio, p-1)
        QUICKSORT(A, p+1, fin)
```

Implementación básica en C

```
int particion(int v[], int inicio, int fin) {
    int pivote = v[fin];
    int i = inicio - 1;

    for (int j = inicio; j < fin; j++) {
```

```
        if (v[j] < pivote) {
            i++;
            int temp = v[i];
            v[i] = v[j];
            v[j] = temp;
        }
    }

    int temp = v[i + 1];
    v[i + 1] = v[fin];
    v[fin] = temp;

    return i + 1;
}

void quickSort(int v[], int inicio, int fin) {
    if (inicio < fin) {
        int p = particion(v, inicio, fin);
        quickSort(v, inicio, p - 1);
        quickSort(v, p + 1, fin);
    }
}
```

Complejidad

Mejor caso: O(n log n). Caso promedio: O(n log n). Peor caso: O(n^2), por ejemplo cuando las particiones quedan muy desequilibradas. La elección del pivote influye directamente en el comportamiento.

3. Merge Sort

Idea fundamental

Merge Sort también utiliza **divide y vencerás**, pero su estrategia es distinta: divide el vector aproximadamente por la mitad hasta obtener subproblemas de un solo elemento. Después combina las partes ordenadas comparando sus primeros elementos.

Lógica paso a paso

1. Dividir el vector en dos mitades.

2. Dividir nuevamente cada mitad hasta llegar a vectores de un elemento.

3. Un vector de un elemento ya está ordenado.

4. Comparar los elementos iniciales de las dos partes.

5. Copiar primero el menor elemento a un vector temporal.

6. Continuar hasta consumir ambas partes.

7. Copiar el resultado ordenado nuevamente al vector original.

Ejemplo conceptual

[45, 12, 78, 34] → [45, 12] y [78, 34] → [45] [12] [78] [34]. Luego se combinan: [12,45] y [34,78], y finalmente [12,34,45,78]. La parte esencial de Merge Sort es la operación de **mezcla o merge**.

Pseudocódigo

```
MERGESORT(A, inicio, fin)
    si inicio < fin:
        medio = (inicio + fin) / 2
        MERGESORT(A, inicio, medio)
        MERGESORT(A, medio + 1, fin)
        COMBINAR(A, inicio, medio, fin)
```

Implementación básica en C

```
void mergeSort(int v[], int inicio, int fin) {
    if (inicio < fin) {
        int medio = (inicio + fin) / 2;

        mergeSort(v, inicio, medio);
        mergeSort(v, medio + 1, fin);

        combinar(v, inicio, medio, fin);
    }
}
```

La función **combinar()** realiza el trabajo principal de comparar las dos secuencias ya ordenadas y construir una sola secuencia ordenada.

Complejidad

Mejor, promedio y peor caso: O(n log n). Requiere memoria auxiliar O(n) para realizar la combinación.

4. Heap Sort

Idea fundamental

Heap Sort se basa en una estructura llamada **heap o montículo binario**. Para ordenar de menor a mayor se construye un **Max Heap**, donde cada padre es mayor o igual que sus hijos. Por ello, el elemento máximo queda en la raíz.

Representación del heap en un vector

No es necesario crear nodos explícitos. Un árbol binario completo puede representarse mediante un vector. Para un elemento ubicado en la posición **i**:

Relación	Fórmula
Padre	(i - 1) / 2
Hijo izquierdo	2*i + 1
Hijo derecho	2*i + 2

Lógica paso a paso

1. Construir un Max Heap a partir del vector.
2. El elemento mayor queda en la posición 0.
3. Intercambiar la raíz con el último elemento del segmento no ordenado.
4. Reducir el tamaño del heap en una posición.
5. Aplicar heapify para recuperar la propiedad del Max Heap.
6. Repetir hasta que solo quede un elemento.

¿Qué hace heapify?

Heapify verifica si un nodo es menor que alguno de sus hijos. Si existe un hijo mayor, intercambia el nodo con el mayor de sus hijos y continúa el proceso hacia abajo. Su objetivo es restaurar la propiedad del heap.

Implementación básica en C

```
void heapify(int v[], int n, int i) {
    int mayor = i;
    int izq = 2 * i + 1;
    int der = 2 * i + 2;

    if (izq < n && v[izq] > v[mayor])
        mayor = izq;

    if (der < n && v[der] > v[mayor])
        mayor = der;

    if (mayor != i) {
        int temp = v[i];
        v[i] = v[mayor];
        v[mayor] = temp;

        heapify(v, n, mayor);
    }
}

void heapSort(int v[], int n) {
    for (int i = n/2 - 1; i >= 0; i--)
        heapify(v, n, i);

    for (int i = n - 1; i > 0; i--) {
```

```
            int temp = v[0];
            v[0] = v[i];
            v[i] = temp;

            heapify(v, i, 0);
        }
    }
```

Complejidad

Mejor, promedio y peor caso: O(n log n). Una ventaja importante es que puede realizarse con memoria auxiliar O(1), sin contar la pila utilizada por la versión recursiva de heapify.

5. Comparación de los tres algoritmos

Característica	Quick Sort	Merge Sort	Heap Sort
Estrategia	Pivote y partición	Dividir y combinar	Heap / Max Heap
Mejor caso	O(n log n)	O(n log n)	O(n log n)
Promedio	O(n log n)	O(n log n)	O(n log n)
Peor caso	$O(n^2)$	O(n log n)	O(n log n)
Memoria	O(log n) aprox.	O(n)	O(1)*
Recursividad	Sí	Sí	heapify puede ser recursivo
Idea central	Ubicar pivote	Fusionar ordenadamente	Extraer máximos

* Heap Sort se considera O(1) en espacio auxiliar cuando se implementa de forma iterativa y sobre el mismo vector.

6. ¿En qué se basa la lógica de cada algoritmo?

Quick Sort: la pregunta central es: “¿Dónde debe quedar el pivote?”. Una vez que el pivote está correctamente ubicado, el problema se divide en dos problemas menores.

Merge Sort: la pregunta central es: “¿Cómo puedo ordenar partes pequeñas y después unirlas correctamente?”. La combinación de dos secuencias ordenadas es eficiente.

Heap Sort: la pregunta central es: “¿Cómo puedo mantener disponible el elemento mayor?”. El Max Heap coloca el máximo en la raíz para poder retirarlo y colocarlo al final.

7. Aspectos que el estudiante debe comprender

• Identificar qué estrategia utiliza cada algoritmo.

• Explicar qué representa cada parámetro de las funciones.

• Seguir manualmente el algoritmo con un vector pequeño.

• Explicar dónde aparece la recursividad.

• Distinguir la partición de Quick Sort, la combinación de Merge Sort y heapify de Heap Sort.

• Relacionar el número de operaciones con la complejidad temporal.

• Explicar qué estructuras auxiliares utiliza cada algoritmo.

• Modificar las implementaciones para ordenar de forma descendente.

8. Ejercicio sugerido

Dado el vector **[38, 12, 45, 7, 23, 19, 4]**, realice primero el proceso manual de Quick Sort, después represente las divisiones y fusiones de Merge Sort y finalmente construya el Max Heap utilizado por Heap Sort. En cada caso indique el estado del vector después de cada operación importante.

Posteriormente implemente los tres algoritmos en C y compruebe que producen el mismo vector ordenado.