Universidad de La Serena

Diseño y Análisis de Algoritmos
Profesor: Erick Luciano Castillo Bastias
Fecha : Miércoles 29 de mayo
Equipo 2

Integrantes :
Fernando López
Rodrigo Mamani
Bryan Townsend

Índice

1. Introducción

En el siguiente informe se detallara la actividad realizada durante el desarrollo de este. Como sabemos los métodos de ordenamiento cumplen un factor relevante en la computación, ya que el propósito principal por lo general es ordenar una secuencia para luego facilitar las búsquedas de los miembros del conjunto ordenado. Es tanto así que existen diferentes tipos de algoritmos de ordenamiento en los cuales una buena practica sería escoger cuidadosamente que algortimo nos será mas util al momento de tener una secuencia de datos por ejemplo. Un concepto importante a mencionar es la estabilidad del algortimo que escojamos, ya que será fundamental en el proceso de ordenamiento.

Figura 1: Tipos de Algortimos de Ordenamiento(Estables e Inestables)

Estables				
Nombre traducido	Nombre original	Complejidad	Memoria	Método
Ordenamiento de burbuja	Bubblesort	$O(n^2)$	$O(1)$	Intercambio
Ordenamiento de burbuja bidireccional	Cocktail sort	$O(n^2)$	$O(1)$	Intercambio
Ordenamiento por inserción	Insertion sort	$O(n^2)$("(en el peor de los casos)")	$O(1)$	Inserción
Ordenamiento por casilleros	Bucket sort	$O(n)$	$O(n)$	No comparativo
Ordenamiento por cuentas	Counting sort	$O(n+k)$	$O(n+k)$	No comparativo
Ordenamiento por mezcla	Merge sort	$O(n \log n)$	$O(n)$	Mezcla
Ordenamiento con árbol binario	Binary tree sort	$O(n \log n)$	$O(n)$	Inserción
	Pigeonhole sort	$O(n+k)$	$O(k)$	
Ordenamiento Radix	Radix sort	$O(nk)$	$O(n)$	No comparativo
	Distribution sort	$O(n^3)$ versión recursiva	$O(n^2)$	
	Gnome sort	$O(n^2)$	$O(1)$	
Inestables				
Nombre traducido	Nombre original	Complejidad	Memoria	Método
Ordenamiento Shell	Shell sort	$O(n^{1.25})$	$O(1)$	Inserción
	Comb sort	$O(n \log n)$	$O(1)$	Intercambio
Ordenamiento por selección	Selection sort	$O(n^2)$	$O(1)$	Selección
Ordenamiento por montículos	Heapsort	$O(n \log n)$	$O(1)$	Selección
	Smoothsort	$O(n \log n)$	$O(1)$	Selección
Ordenamiento rápido	Quicksort	Promedio: $O(n \log n)$, peor caso: $O(n^2)$	$O(\log n)$	Partición
	Several Unique Sort	Promedio: $O(n \mathrm{u})$, peor caso: $O(n^2)$; u=n; u = número único de registros		

es por eso que a partir de la Figura 1,surgen diferentes preguntas considerando a cada uno de los tipos de ordenamientos,una de ellas es:
¿Cuándo conviene usar uno o otro método de ordenamiento?.

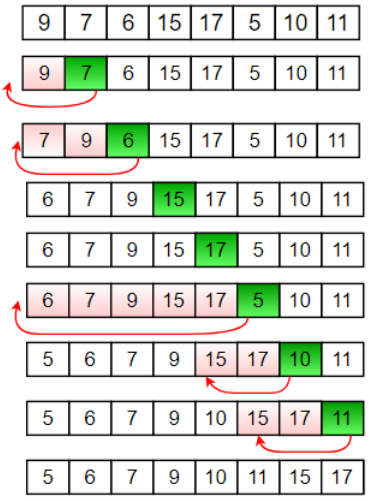
y la respuesta más rapida es considerar cuando se requiere hacer una cantidad considerable de búsquedas y es importante el factor tiempo.

Una vez contextualizado el tema que estaremos constantemente abordando y aterrrizando con la actividad planteada propongo dar una breve definicón muy por encima de los algoritmos que estaremos estudiando y poniendo a prueba durante el desarrollo de este informe:

1.1. Orden por Inserción (InsertionSort)

Dado un array A de n-elementos, la estrategia consiste en realizar n-recorridos por el array. En el recorrido i-ésimo, el elemento almacenado en A[i-1] es trasladado a su lugar apropiado en A[0..i - 1]. De este modo, después del recorrido i-ésimo, los elementos en A[0...i -1] están en orden, lo que significa que después de n-recorridos, el array A completo estará ordenado.

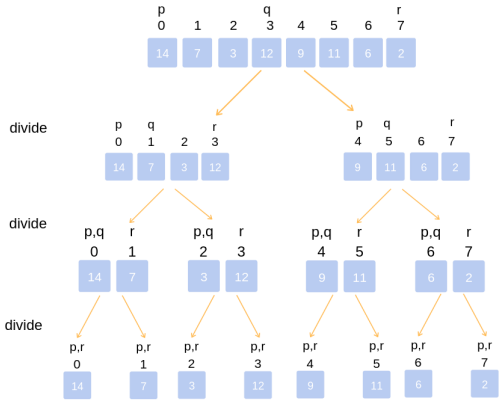
Figura 2: Ejemplo de Ordenamiento por InsertionSort.



1.2. Ordenación por Mezcla (MergeSort)

Dado un array A de n-elementos, la estrategia esta basada en la metodología Dividir para Conquistar, en donde el array es particionado en 2 subarrays, y así sucesivamente hasta Finalmente lograr los subarrays ya ordenados para luego combinarlos para producir el array ordenado final.

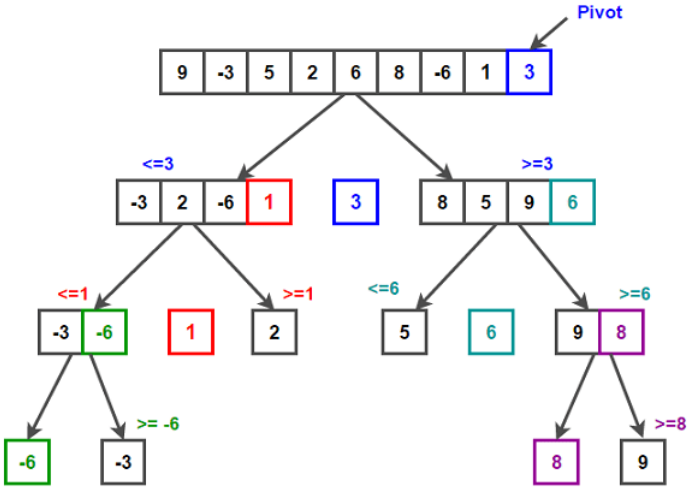
Figura 3: Ejemplo de Ordenamiento por MergeSort.



1.3. Orden Rápido (QuickSort)

Dado un array A de n-elementos, la estrategia también esta basada en la metodología Dividir para Conquistar. Sin embargo, a diferencia del algoritmo de ordenación por mezcla, la parte no recursiva del algoritmo de ordenación rápida con lleva la construcción de subejemplares por medio de una técnica de partición, Se selecciona un elemento especifico del array A[p] llamado pivote, seguidamente se recorre el array haciendo una serie de operaciones considerando en todo momento este elemento que escogeremos, hasta lograr así el ordenamiento.

Figura 4: Ejemplo de Ordenamiento por QuickSort.



2. Desarollo de Software Algoritmos desarrollados y métodos utilizados:

En la siguiente sección se describirá el progreso que realizamos en conjunto con los desarrolladores durante el progreso de la implementación de los metodos de ordenamiento, para luego poder análizar y diferenciar cada uno de los algoritmos como así tambien sus métodos utilizados.

3. QuickSort

El ordenamiento rápido(quicksort en ingles) es un algoritmo basado en comparaciones que utiliza la técnica divide y vencerás que permite ordenar N con costo $O(nlog(n))$. Es una de las técnicas más rápidas para ordenar elementos (basado en comparaciones). Fue desarollada por el científico brítanico en computacion Charles Antony Richard en 1960.

3.1. Descripción

El algortimo trabaja de la siguiente forma:

1. Elegir un elemento del arreglo a ordenar, que llamaremos pivote.
2. Resituar los demas elementos del arreglo a acada lado del pivote, de manera que al lado izquiero solo queden los menores a el pivote y al lado derecho solo queden lo mayores a el pivote.
3. El arreglo queda "dividido.[en] dos los menores al pivote y los mayores a el.
4. Repetir de forma recursiva para cada división hasta que el tamaño de las diviciones sea de 1 elemento.
5. luego de realixar todas las diviciones el arreglo queda ordenado.

3.2. Implementación

3.2.1. Pivote

El pivote sera el primer elemento del arreglo a ordenar

3.2.2. Sort

El método *Sort*() sera el encargado de dividir el arreglo de forma recursiva hasta que solo contengan un elemento.

```
private void Sort(int[] ordenar, int bot, int top) {
    int j = 0;
    if (bot < top) {
        j = particion(ordenar, bot, top);
        Sort(ordenar, bot, j - 1);
        Sort(ordenar, j + 1, top);
    }
}
```

Figura 5: Metodo Sort

3.2.3. Particion

EL método *paricion*() es el encargado de mover los elementos del arreglo a los lados del pivote y retorna un entero que sera la pocición final del pivote donde se dividira el arreglo.

```
private int particion(int[] ordenar, int bot, int top) {
    int piv = ordenar[bot];
    int i = bot;
    int j = top + 1;
    while (true) {
        // encuentra mayor
        while (ordenar[++i] <= piv) {
            if (i == top) {
                break;
            }
        }
        // encuantra menor
        while (ordenar[--j] >= piv) {
            if (j == bot) {
                break;
            }
        }
        if (i >= j) {
            break;
        }
        Swap(ordenar, i, j);
    }
    Swap(ordenar, bot, j);

    return j;
}
```

Figura 6: Metodo particion

4. MergeSort

El ordenamiento por mescla (mergesort en ingles) es un algortimo basado en comparaciones que al igual que en el Quicksort utiliza la tecnica de divir y vencerás, este algortimo permite ordenar N elementos de un arreglo con costo $O(nLog(n))$ ademas de ser estable.

4.1. Descripción

El algortimo trabaja de la siguiente forma:

1. Si el arreglo es de longitud 0 o 1 entonces ya se encuentra oredenado paso 3. En caulquier otro caso paso 2.
2. Divivir el arreglo en dos mitades de aproximadamente la mitad del tamaño, con cada arreglo repetir paso 1.
3. Reordenar cada sub-array de forma recursiva aplicandoe el ordenamiento por mescla.
4. Mesclar los dos sub-array resultantes para generar un rreglo ordenado.

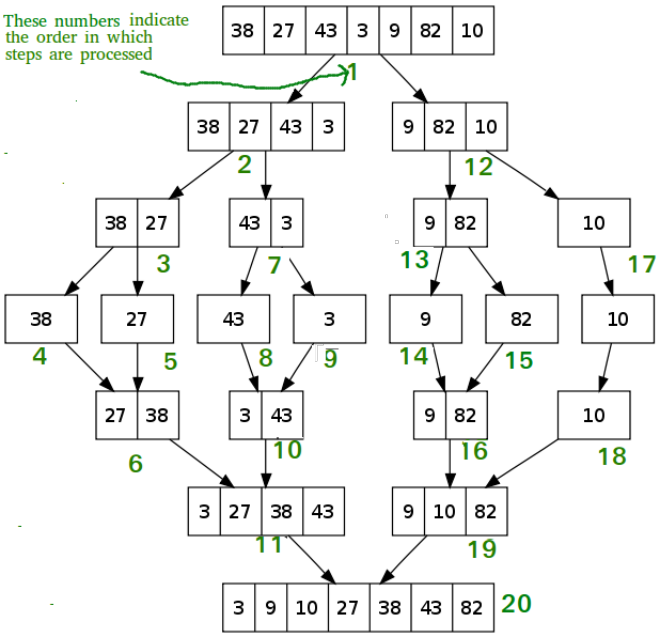


Figura 7: MergeSort

4.2. Implementación

4.2.1. Sort

Primero debemos crear un arreglo auxiliar que nos ayudara a ordenar el arreglo original.

```
public void Sort(int[] ordenar) {
    int[] aux = new int[ordenar.length];
    Sort(ordenar, aux, 0, ordenar.length);
}
```

Figura 8: Sort

4.2.2. Merge

Método que ordenara los elementos de los sub arreglos mediante comparaciones en el arreglo auxiliar, luego de tener el arreglo auxiliar ordenado, se pasan los elementos a el arreglo original.

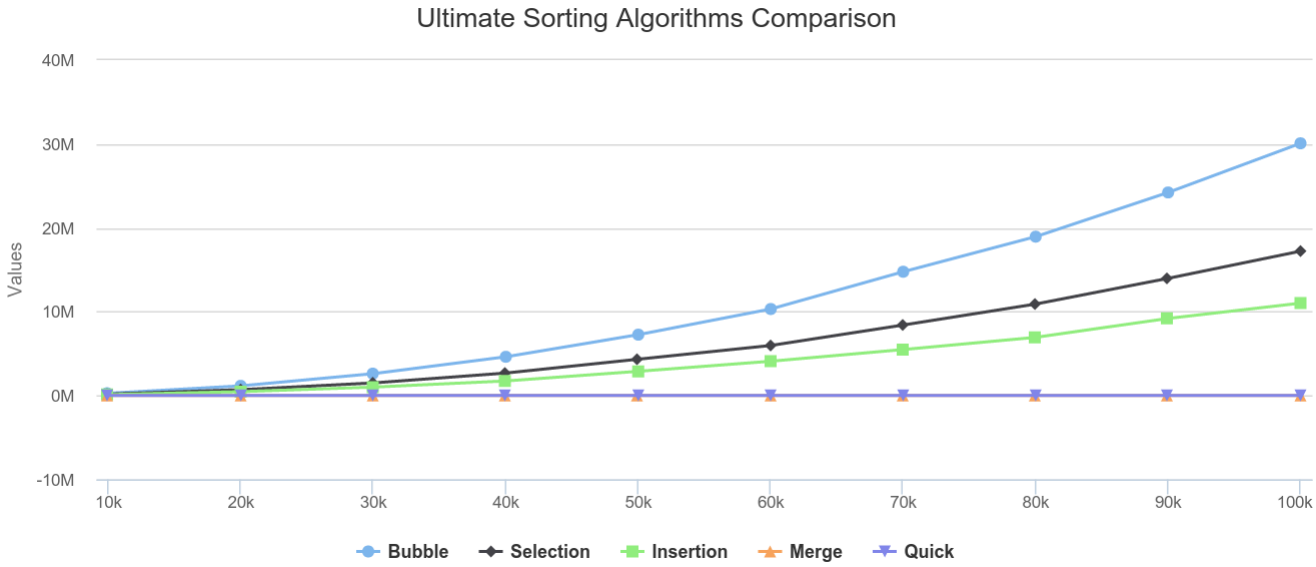
```
private void Merge(int[] ordenar, int[] aux, int bot, int mid, int top) {
    int i = bot;
    int j = mid;

    for (int k = bot; k < top; k++) {
        if (i == mid) {
            aux[k] = ordenar[j++];
        } else if (j == top) {
            aux[k] = ordenar[i++];
        } else if (ordenar[j] < ordenar[i]) {
            aux[k] = ordenar[j++];
        } else {
            aux[k] = ordenar[i++];
        }
    }

    for (int k = bot; k < top; k++) {
        ordenar[k] = aux[k];
    }
}
```

Figura 9: Merge

5. Diferencias entre algoritmos y métodos

Para encontrar diferencias entre los algoritmos estudiados es fundamental considerar los tiempos de ordenamiento para una secuencia dada, es decir, su complejidad, por ende es inverosimil escoger a priori y rapidamente que algoritmo es mejor que otro, por lo que debemos fijarnos cual de estos nos podra ser más util dependiendo de la problemática y la secuencia de la que queremos ordenar. Es por esto que la diferencia más considerable respecto a estos algortimos es el tiempo de ejecución.
Por lo que como sabemos en el peor de los casos QuickSort se tiene O(n elevado a 2) al igual que el InsertionSort, y donde el ordenamiento MergeSort será de O(n log n).

Figura 10: Gráfica y Análisis Algoritmos de Ordenamientos.



Como se puede observar en esta gráfica comparativa se muestran cinco algoritmos de ordenamiento, en los cuales nos enfocaremos en los tres que estudiamos durante el desarollo de esta actividad (InsertionSort, MergeSort,QuickSort). Podemos conocer a ciencia cierta quien fue el ganador, y también se puede notar con total seguridad quien fue el claro perdedor, el cual fue el algoritmo Insertion con una complejidad algoritmica de O(n elevado a 2), pero sus implementaciones son un poco más eficientes dado que se hacen menos comparaciones, pero no dejan de ser algoritmos poco competentes.

6. Conclusión

Para concluir la actividad realizada Como pudimos observar durante la actividad, existen una gran variedad de Algoritmos de Ordenamientos por lo cual la verdadera problemática existe y esta en escoger de buena forma cual de todos escogeremos y usaremos, para así cumplir con nuestro proposito de ordenar la secuencia que tengamos,

Las consideraciones anteriores son un claro ejemplo de los retos que enfrentamos día a día quienes trabajamos en ciencias de la computación, existe una gran cantidad de lenguajes de programación, herramientas y soluciones diversas a un mismo problema, pero cada una con un campo de acción enfocadas a condiciones específicas.

7. Referencias Web

Referencias

[1] Diseño y Análisis de Algoritmos. (DAA-2009)–Dr. Eric Jeltsch F. *PDF Diseño y Análisis de Algoritmos*, Segunda Clase, DAA 2019.

[2] E78. INGENIERÍA DEL SOFTWARE 5º CURSO DE INGENIERÍA INFORMÁTICA 2000-2001, *Especificación de Requisitos Software según el estándar de IEEE 830*, Departament de Informática Universitario Jaume I.

[3] Bazaraa, M.S., J.J. Jarvis y H.D. Sherali, *Programación lineal y flujo en redes*, segunda edición, Limusa, México, DF, 2004.

[4] Dantzig, G.B. y P. Wolfe, «Decomposition principle for linear programs», *Operations Research*, **8**, págs. 101–111, 1960.

Appendices

Anexo I: Diagramas UML (casos de uso y de clases)

Anexo II: Manual de usuario