1. Contexto Problemático:

Una empresa de fabricación de microprocesadores requiere implementar un algoritmo de ordenamiento como una operación nativa de coprocesador, con el fin de que se pueda acelerar el rendimiento del sistema, evitando al procesador principal realizar tareas de cómputo intensiva.

Identificación del Problema.

- Es necesario que los métodos sean eficientes.
- Los métodos de ordenamiento deben permitir ordenar muy rápidamente, números enteros de tamaño arbitrariamente grande y números en formato de coma flotante de cualquier tamaño.
- La empresa requiere tres (3) tipos de métodos de ordenamiento.
- Al implementar estos métodos el coprocesador puede acelerar el rendimiento del sistema por el hecho de liberar trabajo en el procesador principal.

2. Recopilación de la información necesaria:

Tras el análisis del contexto problemático en él nos encontramos y de las necesidades que se tienen, se han definido que del programa a desarrollar se requiere las siguientes funcionalidades:

- El programa debe tener dos opciones para el usuario para ordenar los valores. Pueden ser ingresados por el usuario mediante una interfaz gráfica o el usuario puede indicar el número de valores a generar de manera aleatoria.
- El programa debe permitirle al usuario al escoger la generación aleatoria, el tipo de configuración que desea para los valores a generar. Debe ser posible generar los valores de 4 maneras posibles:
  - La generación aleatoria de valores ya ordenados.
  - La generación aleatoria de valores que estén ordenados de manera inversa.
  - La generación aleatoria de valores en un orden arbitrario.
  - La generación aleatoria de valores, combinando un porcentaje de ellos entregados en orden y otro en desorden. Como entrada, el usuario debe indicar qué porcentaje desea que esté ordenado.
- El programa debe ordenar los valores, usando uno de los tres algoritmos propuestos, buscando el más eficiente para el tipo de valores a ordenar y presentar el tiempo de ejecución de él.

Definiciones

Algoritmo de ordenamiento: Este tipo de algoritmos permiten ordenar un conjunto de valores que poseen una relación de orden, relación transitiva (Siempre que un elemento dentro de un conjunto se relaciona con otro y este último con un tercero, entonces el primero se relaciona con el tercero). Estos ordenamientos al ser eficientes, son importantes a la hora

de optimizar el uso de otros algoritmos, al facilitarse usar listas ordenadas para su rápida ejecución.

Estos algoritmos se clasifican según diferentes criterios. Uno de ellos es según su estabilidad. Los algoritmos estables, son aquellos que mantienen un relativo preorden total, por ejemplo, si dados dos elementos con claves iguales, después de ordenarlos, tienen el mismo orden que antes de la clasificación. Mientras que los algoritmos de ordenamiento inestables, pueden cambiar el orden relativo de registros con claves iguales.

Clave: Es la parte de un registro por la cual se ordena la lista o vector. (En una lista con diferentes atributos para cada elemento, se tiene en cuenta uno para hacer el ordenamiento).

Criterio de ordenamiento o comparación: Es el criterio que permite asignar valores a cada elemento con base a una o más claves que se hayan predeterminado, así se define si un elemento es mayor o menor que otro.

- Estabilidad
- Tiempo de Ejecución
- Requerimientos de Memoria

Los algoritmos de ordenamiento también los podemos identificar en cuatro grandes grupos según su método para ordenar la información, los cuales son los algoritmos de inserción, los de intercambio, los de selección y los de enumeración.

A continuación, dentro de estos grupos, presentamos diferentes alternativas de algoritmos de ordenamiento:

- **Insertion Sort:** En este algoritmo se recibe un arreglo con n elementos para ordenar y cambia las posiciones de sus elementos hasta dejarlo en el orden requerido. Este algoritmo es muy útil cuando se trabaja con pequeñas listas de datos".
- **Selection Sort:** Este método busca el menor de los elementos dentro del arreglo dado y lo intercambia con el que está en la primera posición, luego el segundo más pequeño, y así sucesivamente, hasta lograr el arreglo ordenado.

  Este método podría ser útil para arreglos de tamaño no muy grande, pues evalúa posición por posición.

- **Bubble Sort:** Este algoritmo, recorre todo el arreglo cambiando de orden los elementos "adyacentes" que estén en desorden, recorriendolo hasta que no encuentre ningún cambio para hacer.

  "La ventaja principal del ordenamiento de burbuja es que es muy popular y fácil de implementar. Además, en este tipo de ordenamiento, los elementos se intercambian sin utilizar almacenamiento temporal adicional, de modo que el espacio requerido es el mínimo. La principal desventaja del ordenamiento de burbuja es el hecho de que no se comporta adecuadamente con una lista que contenga un número grande de elementos. Esto se debe a que este ordenamiento requiere n al cuadrado de pasos de procesamiento para cada n número de elementos a ser ordenados. Como tal, este tipo

de ordenamiento es más apropiado para la enseñanza académica pero no para aplicaciones de la vida real."[1]

- **Bucket Sort:** Este algoritmo distribuye los elementos del arreglo a ordenar, en un número de casilleros finitos, en el caso de valores numéricos, cada casillero cumple con cierto rango, y en cada casillero, sus elementos se ordenan según un algoritmo de ordenamiento básico.
- **Quicksort:** Este algoritmo escoge un "pivote" o "elemento" de la lista a ordenar y reordenar los elementos para que a un lado quedan los menores que él y al otro lado los mayores. Al momento en el que el "pivote" termina de clasificar los otros "pivotes", él, queda en el lugar que le corresponderá en la lista ordenada. A partir de este momento queda divida la lista en dos sublistas y este proceso se repite para cada sublista mientras contengan al menos un elemento.
- **Heap Sort:** Este algoritmo organiza los elementos en un árbol, y comienza a extraer el nodo que queda como raíz (nodo sin nodos debajo de él), insertándose en el arreglo hasta obtener el conjunto ordenado. Su funcionamiento se basa en una propiedad de "montículos", los cuales el elemento de menor o mayor valor se encuentra en su cima, según se haya establecido.
- **Counting Sort:** Este algoritmo crea un vector de los números enteros ordenados que pertenecen al intervalo de los datos a organizar, y a cada elemento le da el valor de 0, simulando que no ha aparecido en la lista a ordenar. Seguidamente se recorren los elementos a ordenar y se cuenta el número de apariciones indicándose en el vector creado. Tras esto se eliminan los elementos que no aparezcan en la lista a ordenar, y se tiene el arreglo ordenado.
- **Merge Sort:** Este algoritmo se basa en la técnica "divide y vencerás", es un algoritmo recursivo que al insertarse una lista, la divide en dos sublistas del mitad del tamaño de la lista original, luego se ordenan las sublistas con el mismo algoritmo de manera recursiva y se mezclan las dos sublistas en una mezcla ordenada.

3. Búsqueda de soluciones creativas:

Los algoritmos, que se seleccionaron, que son más acordes al problema fueron los siguientes:

- Heap Sort
- Quicksort
- Merge Sort

Se seleccionaron los anteriores métodos de ordenamiento porque cumplian las espectativas de ordenamiento de forma masiva y eficiente.

[1] https://techlandia.com/ventajas-desventajas-algoritmos-ordenamiento-info_181515/

4. Transición de la formulación de ideas a los diseños preliminares

Las otras alternativas que no fueron seleccionadas no manejaban datos de gran cantidad de forma eficiente y no cumplian las espectativas del programa que se realizaría. Una forma más resumida de ver el asunto de selección de métodos de ordenamiento para el problema, es que todos realizaban el ordenamiento de los datos de forma eficiente pero para cantidades de datos muy pequeños, pero a la hora de someterlos a una cantidad de datos considerables, no cumplian el objetivo especifico que era el de ordenar los datos de una forma eficiente y rápida, en lugar de realizar la tarea eficientemente, le dan muchas vueltas a los datos y realizan muchas operaciones innecesarias.

La revisión cuidadosa de las otras alternativas nos conduce a lo siguiente:

- Heap Sort: Este método es más lento que otros métodos, pero es más eficaz en escenarios más rigurosos. Es un método no recursivo y no es estable, tiene una complejidad para el peor de los casos $O(n \cdot \log n)$
- Quicksort: Esta es la técnica de ordenamiento más rápida conocida y aplica el método de divide y vencerás. Es uno de los algoritmos más eficientes. Además, puede ordenar elementos de cualquier tipo para los cuales se define una relación "menor que".En el peor de los casos , hace comparaciones $O(n \cdot \log n)$.
- Merge Sort: Es un algoritmo de clasificación basado en comparación y eficiencia. Muchos de los casos cuando son implementados producen un tipo estable , lo que significa que la implementación conserva el orden de entrada de elementos iguales en la salida ordenada. Tiene una complejidad para el peor de los casos de $O(n \cdot \log n)$.

5. Evaluación y Selección de la Mejor Solución

Se escogieron los tres métodos de ordenamiento (Heap Sort, QuickSort y Mergesort) ya que, admiten ordenamiento para los dos tipos de datos que estamos manejando, int y double. Además los tres presentan el mismo tipo de complejidad para el peor caso que es el $O(n \cdot \log n)$ y esta complejidad ,después de la complejidad lineal ,es una de las mejores.

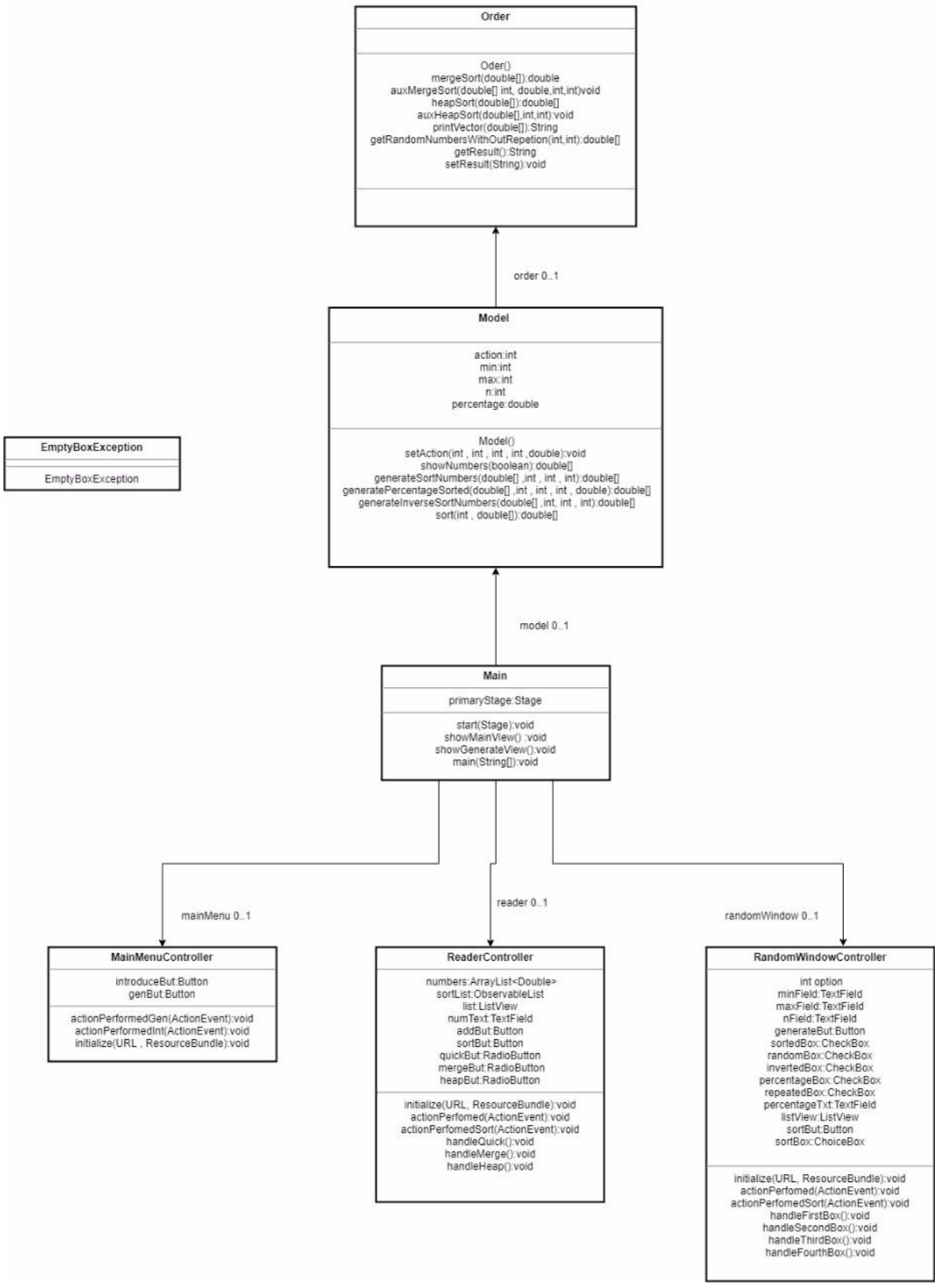
A grandes rasgos estos tres algoritmos los describiremos brevemente con sus respectivos pseudocodigo, con el fin de facilitar su implementación.

QuickSort(arreglo[], inicio, fin)

```
{       if ( inicio < fin)
        {
                pivote =arreglo[inicio]
                temp = inicio
                desde i= a+1 hasta fin
                {
```

```
                    mientras (arreglo[i] < pivote]
                    {
                        intercambiar ( arreglo[i] -- arreglo[temp+1)
                        temp = temp +1
                    }
                }
                intercambiar ( pivote -- arreglo[temp])
            }
            QuickSort( arreglo, inicio, temp)
            QuickSort( arreglo, temp +1, final)
}
```

6.Preparación de Informes y Especificaciones



7.Implementación del Diseño

implementación en el lenguaje de programación java.

Lista de tareas a implementar:

1. Generar una lista de números conforme a los datos ingresados
2. Ordenar la lista de números generada
3. Ordenar un porcentaje de números
4. Invertir la lista de números ordenada
5. Imprimir la lista de números
6. Leer datos ingresados por el usuario y ordenarlos con el algoritmo de ordenamiento especificado
7. Mostrar una interfaz gráfica con la lista de datos generados, ordenados o no ordenados
8. Mostrar una interfaz gráfica que le permita al usuario ingresar los valores a ordenar

Construcción:

1. Generar los números

Nombre:	generateNumbers
Descripción:	Genera aleatoriamente una matriz de números enteros y dobles en una matriz doble, dentro de un rango específico
Entradas:	size: Tamaño de la matriz; min: Rango min; max: Rango máximo
Salida:	Retorna un array de int y double

```java
public double[] generateNumbers(int size, int min, int max)
{
    array = new double[size];
    int type;
    this.min = min;
    this.max = max;

    for(int i = 0; i<size; i++)
    {
        type = (int)Math.floor(Math.random()*2);
        if(type == INT)
        {
            int b =  (int)Math.floor(Math.random()*(max-(min+1)) +min);
            array[i] = b;
        }
        else
        {
            array[i] = Math.random()*(max-(min+1)) +min;
        }
    }
    return array;
}
```

2. Ordenar la lista de números por los tres métodos escogidos

Nombre:	quickSort
Descripción:	Ordena la matriz de datos por el método de ordenamiento Quick Sort
Entradas:	array: Matriz de datos; min: Rango min; max: Rango máximo
Salida:	Retorna un array

```java
public double[] quickSort( double[] array, int min, int max){
    this.array = array;

    double pivot = array[min];
    double temporal;
    int a = min;
    int b = max;

    while(a<b)
    {
        while(array[a]<= pivot && a<b)
        {
            a++;
        }
        while(array[b]>pivot)
        {
            b--;
        }
        if(a<b)
        {
            temporal = array[a];
            array[a] = array[b];
            array[b] = temporal;
        }
    }
    array[min] = array[b];
    array[b] = pivot;
    if(min < b-1)
    {
        quickSort(array, min, b-1);
    }
    if(b+1 < max)
    {
        quickSort(array, b+1, max);
    }
    return array;
}
```

Nombre:	doubleMergeSort
Descripción:	Ordena los datos de tipo double por el método de ordenamiento Merge Sort
Entradas:	array, es el main statement del algoritmo recursivo
Salida:	array ordenado

```java
public static double[] doubleMergeSort(double[] n) {
    int i = 1;
    for (i = 1; i < n.length; i *= 2) {
        for (int j = 0; j < n.length; j += i) {
            int p = i >> 1;
            doubleMerge(n, j, j + p - 1, j + p, j + p + p - 1);
        }
    }
    doubleMerge(n, 0, i / 2 - 1, i / 2, n.length);
    return n;
}
```

Nombre:	doubleMerge
Descripción:	Ordena los datos de tipo double por el método de ordenamiento Merge Sort
Entradas:	array: double o int, a,b,c,d números enteros los cuales permiten dividir y combinar el problema en subprocesos más sencillos.
Salida:	sin retorno

```java
public static void doubleMerge(double[] n, int a, double b, int c, int d) {
    d = Math.min(d, n.length - 1);
    double mer[] = new double[d - a + 1];
    int idx = 0;
    int or = a;
    while (idx < mer.length)
        if ((a > b ? false : (c > d ? true : n[a] <= n[c])))
            mer[idx++] = n[a++];
        else
            mer[idx++] = n[c++];

    for (int i = 0; i < mer.length; i++)
        n[or + i] = mer[i];
}
```

Nombre:	doubleHeapSort

Descripción:	llamado principal del método recursivo
Entradas:	array: Recibe el arreglo que se espera ordenar
Salida:	retorna el array ordenado

```
static void doubleHeapSort(double arr[]) {
    int n = arr.length;

    // Build heap (rearrange array)
    for (int i = n / 2 - 1; i >= 0; i--)
        doubleHeap(arr, n, i);

    // One by one extract an element from heap
    for (int i = n - 1; i >= 0; i--) {
        // Move current root to end
        double temp = arr[0];
        arr[0] = arr[i];
        arr[i] = temp;

        // call max heapify on the reduced heap
        doubleHeap(arr, i, 0);

    }

}
```

Nombre:	doubleHeap
Descripción:	Llamado recursivo del método
Entradas:	array: vector a organizar, n, i valores enteros que representan los valores mayor y menor del arreglo
Salida:	sin retorno

```java
public static void doubleHeap(double arr[], int n, int i) {
    int largest = i; // Initialize largest as root
    int l = 2 * i + 1; // left = 2*i + 1
    int r = 2 * i + 2; // right = 2*i + 2

    // If left child is larger than root
    if (l < n && arr[l] > arr[largest])
        largest = l;

    // If right child is larger than largest so far
    if (r < n && arr[r] > arr[largest])
        largest = r;

    // If largest is not root
    if (largest != i) {
        double swap = arr[i];
        arr[i] = arr[largest];
        arr[largest] = swap;

        // Recursively heapify the affected sub-tree
        doubleHeap(arr, n, largest);
    }
}
```

3. Ordenar un porcentaje de números

Nombre:	sortPercentage
Descripción:	De acuerdo acuerdo a la lista que es ingresada, se intercambian k/2 veces, dos posiciones dentro del arreglo ordenado (K representa el número de elementos que deben estar en desorden segun el porcentaje)
Entradas:	Lista de datos ordenados Porcentaje de desorden
Salida:	Lista de datos desordenados

```
public double[] sortPercentage(double[] array, double percentage)
{
    this.array = array;
    int k = (int)(array.length*percentage);


    int index1;
    int index2;
    double temp;
    for(int i = 0; i<k/2; i++)
    {
        index1 = (int) Math.floor(Math.random() +(array.length-1));
        index2 = (int) Math.floor(Math.random() +(array.length-1));
        while(index1==index2)
        {
            index2 = (int) Math.floor(Math.random() +(array.length-1));
        }

        temp = array[index1];
        array[index1] = array[index2];
        array[index2] = temp;
    }

    return array;

}
```

4. Invertir la lista de números ordenada

Nombre:	invertArray
Descripción:	Se invierte el arreglo, y las posiciones de cada elemento pasan a ser n-i (donde i es la posición del arreglo ordenado)
Entradas:	Lista de números ordenada
Salida:	Lista de números ordenada invertida

```
public double[] invertArray(double[] array)
{
    this.array = array;
    int n = array.length;
    double[] inverted = new double[n];

    for(int i = 0; i<array.length;i++)
    {
        inverted[i] = array[n-i-1];
    }
    return inverted;
}
```

Fuente: *https://es.wikipedia.org/wiki/Relaci%C3%B3n_transitiva*

https://es.wikipedia.org/wiki/Algoritmo_de_ordenamiento
http://metododeordenacion.blogspot.com/p/algoritmo-de-ordenamiento.html
https://www.monografias.com/trabajos/algordenam/algordenam.shtml
http://c.conclase.net/orden/#registro
https://es.wikipedia.org/wiki/Heapsort
https://es.wikipedia.org/wiki/Mont%C3%ADculo_(inform%C3%A1tica)
https://es.wikipedia.org/wiki/Ordenamiento_por_cuentas

Tipo de dato	Representación	Tamaño (Bytes)	Rango de Valores	Valor por defecto	Clase Asociada
byte	Numérico Entero con signo	1	-128 a 127	0	Byte
short	Numérico Entero con signo	2	-32768 a 32767	0	Short
int	Numérico Entero con signo	4	-2147483648 a 2147483647	0	Integer
long	Numérico Entero con signo	8	-9223372036854775808 a 9223372036854775807	0	Long
float	Numérico en Coma flotante de precisión simple Norma	4	$\pm 3.4x10^{-38}$ a $\pm 3.4x10^{38}$	0.0	Float
double	Numérico en Coma flotante de precisión doble Norm	8	$\pm 1.8x10^{-308}$ a $\pm 1.8x10^{308}$	0.0	Double

5.

Análisis de complejidad

Método de ordenamiento: Ordenamiento por mezcla (MergeSort)

Función de recurrencia: *2T(n/2) + nc*

Función de complejidad temporal: *O (n log n)*

EXPLICACIÓN:

El ordenamiento por mezcla utiliza la técnica de ***Divide y Vencerás***. La técnica de ***Divide y Vencerás*** utiliza básicamente tres pasos:

1. ***Divide***: Dividir el problema en un cierto número de subproblemas.
2. ***Vence:*** Soluciona los problemas de manera recursiva. Si el tamaño de los subproblemas es suficientemente pequeño, simplemente resuélvelos de la manera más obvia.
3. ***Combina:*** Combina el resultado de los subproblemas para obtener la solución al problema original.

La ordenación por mezcla se apega estrictamente a la técnica. La idea del algoritmo es la siguiente:

1. ***Divide:*** Divide la secuencia de ***n*** elementos en dos subsecuencias de ***n/2*** elementos.
2. ***Vence:*** Ordena ambas subsecuencias de manera recursiva.
3. ***Combina:*** *Mezcla* las dos subsecuencias ordenadas para obtener la solución del problema.

Para la solución recursiva cada subsecuencia a su vez se divide en dos sub-subsecuencias, y así hasta obtener una subsecuencia de tamaño 1, en este momento se detienen la recursión, ya que una subsecuencia de tamaño uno, siempre está ordenada.

Veamos el siguiente pseudocódigo:

Si V es de tamaño 1 entonces

El vector v ya está ordenado sino

dividir V en dos subvectores A y B

fin {si}

Ordenar A y B usando MergeSort

Mezclar las ordenaciones de A y B para generar el vector ordenado.

Donde dividir V en dos subvectores se puede refinar:

Asignar al vector A el subvector [, ... ,]

Asignar a B el subvector [,.....]

Mientras que mezclar las ordenaciones de A y B consiste en ir entremezclando adecuadamente las componentes ya ordenadas de A y B.

Veamos una implementación a manera de lenguaje de procedimiento SQL:

En resumen, el procedimiento ***Mezcla*** funciona como sigue:

- Las líneas 1 y 2 calculan el largo del primer y segundo sub-arreglos.
- Las líneas 3 y 4 apartan memoria suficiente para los dos sub-arreglos.
- Los ciclos desde de las líneas 5, 6, 7 y 8 copian ambos sub-arreglos a los arreglos ***L*** y ***R***.
- En las líneas 9 y 10 se inicializa el último término de ***L*** y ***R*** a infinito, esto es muy importante, ya que aquí el infinito nos sirve para saber cuándo ya se terminó uno de los sub-arreglos. Aunque en cuestión práctica no existe tal cosa como "infinito" en las computadoras, se suele usar algún valor que previamente hayamos establecido como nuestro "infinito".
- El ciclo *desde* que inicia en la línea 13 va desde ***p*** hasta ***r***, es decir se ejecuta un número de veces igual a la suma de los elementos de ambos sub-arreglos. El funcionamiento de este ciclo es la idea básica del algoritmo. Compara el primer elemento no mezclado de cada sub-arreglo y retira el menor, avanzando hacia el siguiente elemento del sub-arreglo de donde se retiró el elemento.

Supongamos que la división de nuestro problema nos entrega ***a*** subproblemas, cada uno de tamaño ***1/b*** (para el caso de la ordenación por mezcla, tanto ***a*** como ***b*** son iguales a 2). Si tomamos que ***D(n)*** es el tiempo que tardamos en dividir el problema en subproblemas, y ***C(n)***, el tiempo que tardamos en combinar los subproblemas en una solución, entonces tenemos que:

Hay un "teorema maestro" que sirve para resolver recurrencias de esta forma, sin embargo, para varios casos, como el de la ordenación por mezcla, se puede ver de manera intuitiva cual va a ser el resultado. Revisemos paso a paso cada una de las partes.

- ***Dividir:*** Para dividir, basta con calcular cual es la mitad del arreglo, esto, se puede hacer sin ningún problema en tiempo constante, por lo que tenemos que para la ordenación por mezcla ***D(n) = O(1)***
- ***Vencer:*** Como dividimos el problema a la mitad y lo resolvemos recursivamente, entonces estamos resolviendo dos problemas de tamaño ***n/2*** lo que nos da un tiempo ***T(n) = 2T(n/2)***
- ***Combinar:*** Vimos anteriormente que nuestro ***Mezcla*** era de orden lineal, por lo que tenemos que ***C(n)=O(n)***

Para sustituir, sumamos ***D(n) + C(n) = O(n) + O(1) = O(n)***, recordemos que para el análisis de complejidad únicamente nos importa el término que crezca más rápido. Por lo que sustituyendo en la recurrencia tenemos que:

donde ***c*** representa una constante igual al tiempo que se requiera para resolver un problema de tamaño 1.

Sabemos que ***n*** sólo puede ser dividido a la mitad ***log n*** veces, por lo que la profundidad de nuestra recursión será de ***log n***, también del funcionamiento del algoritmo podemos ver que cada vez que dividamos a la mitad, vamos a tener que mezclar los ***n*** elementos, y sabemos que mezclar ***n*** elementos nos toma un tiempo proporcional a ***n***, por lo que resolver la recursión debe tomar un tiempo proporcional a ***n log n***, que de hecho es la complejidad del algoritmo. La ordenación por mezcla tiene una complejidad de ***O(n log n).***

Podemos comprobarlo, sustituyendo valores en la ecuación de recurrencia:

De la tabla anterior vemos que el tiempo real de corrida del ordenamiento por mezcla es proporcional a ***n log n + n***, de nuevo, como en el análisis de complejidad únicamente nos importa el término que crezca más rápido y en esta función ese término es ***n log n***, por lo tanto, la complejidad del sistema es ***O(n log n)***.

Método de ordenamiento: Ordenamiento por montículos (HeapSort)

Función de recurrencia: $T(k) \leq 2T(k-1) + t(k)$,

Función de complejidad temporal: $O(n \log n)$

EXPLICACIÓN:

La idea es construir, con los elementos a ordenar, un montículo sobre el propio vector. Una vez construido el montículo, su elemento mayor se encuentra en la primera posición del vector (a[i]). Se intercambia entonces con el último (a[f]) y se repite el proceso para el subvector a[i, ... , f–1]. Así sucesivamente hasta recorrer el vector completo. Esto nos lleva a un algoritmo de orden de complejidad O(n log n).

Veamos las funciones que se deberían implementar a manera de lenguaje de procedimiento SQL:

Para estudiar la complejidad del algoritmo hemos de considerar dos partes. La primera es la que construye inicialmente el montículo a partir de los elementos a ordenar y la segunda va recorriendo en cada iteración un subvector más pequeño, colocando el elemento raíz en su posición correcta dentro del montículo. En ambos casos nos basamos en la función que “empuja” elementos en el montículo. Observando el comportamiento del algoritmo, la diferencia básica entre el caso peor y el mejor está en la profundidad que hay que recorrer cada vez que necesitamos “empujar” un elemento. Si el elemento es menor que todos los demás, necesitaremos recorrer todo el árbol (profundidad: log n); si el elemento es mayor o igual que el resto, no será necesario. El procedimiento HacerMonticulo es de complejidad O(n) en el peor caso, puesto que si k es la altura del montículo (k = log n), el algoritmo transforma primero cada uno de los dos subárboles que cuelgan de la raíz en montículos de altura a lo más k–1 (el subárbol derecho puede tener altura k–2), y después empuja la raíz hacia abajo, por un camino que a lo más es de longitud k. Esto lleva a lo más un tiempo t(k) de orden de complejidad O(k) con lo cual $T(k) \leq 2T(k-1) + t(k)$, ecuación en recurrencia cuya solución verifica que $T(k) \in O(2^k)$. Como $k = \log n$, la complejidad de HacerMonticulo es lineal en el peor caso. Este caso ocurre cuando hay que recorrer siempre la máxima profundidad al empujar a cada elemento, lo que sucede si el vector está originalmente ordenado de forma creciente. Respecto al mejor caso de HacerMonticulo, éste se presenta cuando la profundidad a la que hay que empujar cada elemento es cero. Esto se da, por ejemplo, si todos los elementos del vector son iguales. En esta situación la complejidad del algoritmo es O(1). Estudiemos ahora los casos mejor y peor del resto del algoritmo Montículos. En esta parte hay un bucle que se ejecuta siempre n–1 veces, y la complejidad de la función que intercambia dos elementos es O(1). Todo va a depender del procedimiento Empujar, es decir, de la profundidad a la que haya que empujar la raíz del montículo en cada iteración, sabiendo que cada montículo tiene n–i elementos, y por tanto una altura de log(n–i), siendo i el número de la iteración. En el peor caso, la profundidad a la que hay que empujar las raíces respectivas es la máxima, y por tanto la complejidad de esta segunda parte del algoritmo **es O(n logn).** ¿Cuándo ocurre esto? Cuando el elemento es menor que todos los demás. Pero esto sucede siempre que los elementos a ordenar sean distintos, por la forma en la que se van escogiendo las nuevas raíces. En el caso mejor, aunque el bucle se sigue repitiendo n–1 veces, las raíces no descienden, por ser mayores o

iguales que el resto de los elementos del montículo. Así, la complejidad de esta parte del algoritmo es de orden O(n). Pero este caso sólo se dará si los elementos del vector son iguales, por la forma en la que originariamente se construyó el montículo y por cómo se escoge la nueva raíz en cada iteración (el último de los elementos, que en un montículo ha de ser de los menores).

Método de ordenamiento: Ordenamiento rápido (Quicksort)

Función de recurrencia: $T(n) = 8 + T(a) + T(b) + TPivote(n)$

Función de complejidad temporal:

- ***Mejor caso:*** $O(n \log n)$
- ***Peor caso:*** $O($

EXPLICACIÓN:

Veamos la implementación a manera de lenguaje de procedimiento SQL:

Este método es de orden de complejidad **Θ(n 2)** en el peor caso y **Θ(n log n)** en los casos mejor y medio. Para ver los tiempos de ejecución nos basamos en la siguiente ecuación en recurrencia:

T(n) = 8 + T(a) + T(b) + TPivote(n)

Donde a y b son los tamaños en los que la función Pivote divide al vector (por tanto, podemos tomar que a + b = n), y TPivote(n) es la función que define el tiempo de ejecución de la función Pivote. El procedimiento Quicksort "rompe" la filosofía de caso mejor, peor y medio de los algoritmos clásicos de ordenación, pues aquí tales casos no dependen de la ordenación inicial del vector, sino de la elección del pivote.

Así, el mejor caso ocurre cuando a = b = n/2 en todas las invocaciones recursivas del procedimiento, pues en este caso obtenemos TPivote(n) = 13 + 4n y por tanto:

T(n) = 21 + 4n + 2T(n/2).

Resolviendo esta ecuación en recurrencia y teniendo en cuenta las condiciones iniciales T(0) = 1 y T(1) = 27 se obtiene la expresión final de T(n), en este caso:

T(n) = 15nlogn + 26n +1.

Ahora bien, si a = 0 y b = n–1 (o viceversa) en todas las invocaciones recursivas del procedimiento, TPivote(n) = 11 + 39/8n, obteniendo:

T(n) = 19 + 39/8n + T(n–1).

Resolviendo la ecuación para las mismas condiciones iniciales, nos encontramos con una desagradable sorpresa:

En consecuencia, la elección idónea para el pivote es la mediana del vector en cada etapa, lo que ocurre es que encontrarla requiere un tiempo extra que hace que el algoritmo se vuelva más ineficiente en la mayoría de los casos. Por esa razón como pivote suele escogerse un elemento cualquiera, a menos que se conozca la naturaleza de los elementos a ordenar. En nuestro caso, como a priori suponemos equiprobable cualquier ordenación inicial del vector, hemos escogido el primer elemento del vector, que es el que se le pasa como segundo argumento a la función Pivote. Esta elección lleva a tres casos desfavorables para el algoritmo: cuando los elementos son todos iguales y cuando el vector está inicialmente ordenado en orden creciente o decreciente. En estos casos *la complejidad es cuadrática* puesto que la partición se realiza de forma totalmente descompensada. A pesar de ello suele ser el algoritmo más utilizado, y se

demuestra que su tiempo promedio es menor, en una cantidad constante, al de todos los algoritmos de ordenación de complejidad O(n logn). En todo esto es importante hacer notar, como hemos indicado antes, la relevancia que toma una buena elección del pivote, pues de su elección depende considerablemente el tiempo de ejecución del algoritmo.

COMPLEJIDAD ESPACIAL: N dado que se deben organizar n elementos siempre y tomarán espacio de memoria en los recursos de la máquina para llevar a cabo dichas tareas

Referencias:

http://www.olimpiadadeinformatica.org.mx/omi/omi/archivos/apuntes/AnalisisDeComplejidad.htm

https://www2.infor.uva.es/~jvalvarez/docencia/tema5.pdf

http://www.lcc.uma.es/~av/Libro/CAP2.pdf

- Algoritmos de ordenamiento internos y externos??
  https://www.monografias.com/trabajos/algordenam/algordenam.shtml