## **Definición del problema**

Evitar que el procesador principal tenga que realizar tareas de cómputo intensivo usando un coprocesador matemático, dicho coprocesador tendrá 3 algoritmos de ordenamiento como operaciones nativas.

Se deben encontrar 3 algoritmos de ordenamiento que puedan ordenar, números enteros de tamaño arbitrariamente grande y números en formato de coma flotante de cualquier tamaño, muy eficientemente.

Para eso se deben pasar los algoritmos por ciertas pruebas y especificar el tiempo que se demoran en ejecutar estos casos de prueba.

|  |  |
| --- | --- |
| Nombre | R1. Generar floats |
| Resumen | Generar de manera aleatoria números de tipo coma flotante |
| Entrada | · Cantidad de números  · Booleano para saber si se pueden repetir números con el mismo valor  · Identificador que determinara el orden en el que estarán los números generados |
| Salida | Números de tipo coma flotante |

|  |  |
| --- | --- |
| Nombre | R2. Generar enteros |
| Resumen | · Cantidad de números  · Booleano para saber si se pueden repetir números con el mismo valor  · Identificador que determinara el orden en el que estarán los números generados |
| Entrada | Cantidad de números |
| Salida | Números enteros |

|  |  |
| --- | --- |
| Nombre | R3. Ordenar valores ascendentemente |
| Resumen | Se debe ordenar un conjunto de valores de manera ascendente |
| Entrada | Valores a ordenar |
| Salida | Valores ordenados ascendentemente |

|  |  |
| --- | --- |
| Nombre | R4. Ordenar valores descendentemente |
| Resumen | Se debe ordenar un conjunto de valores de manera descendente |
| Entrada | Valores a ordenar |
| Salida | Valores ordenados descendentemente |

|  |  |
| --- | --- |
| Nombre | R5. Desordenar valores |
| Resumen | Desordenar valores según un % |
| Entrada | Coeficiente de desordenamiento del conjunto |
| Salida | Valores desordenados |

|  |  |
| --- | --- |
| Nombre | R6. Ingresar valores |
| Resumen | Ingresar valores manualmente a través de una interfaz gráfica |
| Entrada | Valores |
| Salida | Ninguna |

|  |  |
| --- | --- |
| Nombre | R7. Visualizar datos |
| Resumen | Interfaz gráfica que permita ver los valores ingresados o generados aleatoriamente, además de mostrar el tiempo de ejecución de los algoritmos de ordenamiento que se usan para organizar ese conjunto de valores |
| Entrada |  |
| Salida | Interfaz |

## **Recopilación de información**

Ya que se va a tratar con algoritmos de ordenamiento, se debe saber que son y que características pueden variar entre ellos.

Un algoritmo de ordenamiento lo que hace es acomodar elementos en una lista en una secuencia dada por un orden especificado anteriormente.

Los algoritmos de ordenamiento manejan ciertos conceptos clave:

* Complejidad temporal y espacial
* Estructuras de datos
* Algoritmos de ordenación natural y no natural

**Complejidad temporal**:

Para llevar a la mejor solución del problema se debe conocer qué tipo de complejidad temporal conviene más al momento de buscar los algoritmos de ordenamiento más eficientes.

|  |  |
| --- | --- |
| Complejidad | Terminología |
| O(1) | Complejidad constante |
| O(log n) | Complejidad logarítmica |
| O(n) | Complejidad lineal |
| O(n log n) | Complejidad n log n |
| O(nk) | Complejidad polinómica |
| O(kn) | Complejidad exponencial |
| O(n!) | Complejidad factorial |

Fig 1. Tabla de tipos de complejidad más comunes.

En la tabla, se inicia desde las complejidades computacionales más eficientes hasta las menos óptimas. En el caso de la solución del problema se desea encontrar cualquier algoritmo que esté entre complejidad constante hasta complejidad n log n.

**Estructuras de datos**:

Se debe conocer que es mucho más fácil organizar listas de elementos que manejan una estructura de datos distintiva, como lo hacen los árboles o los montículos, que aprovechan de la recursión para llegar a soluciones más rápidas.  
Así como también el uso del divide y vencerás en las listas de elementos.

**Algoritmos de orden natural**:

Son aquellos que tardan lo mínimo posible cuando la entrada esta ordenada.

**Algoritmos de ordenación no natural**:

Son aquellos que tardan lo mínimo posible cuando la entrada esta inversamente ordenada.

**Búsqueda de soluciones creativas:**

La técnica de generación de ideas fue lluvia de ideas. Se tomo en cuenta la mayor cantidad de algoritmos de ordenamiento sin importar sus atributos.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Algoritmo | Mejor caso | Peor caso | Caso más común | Memoria |
| Bubble sort | O(n2) | O(n2) | O(n2) | O(1) |
| Selection sort | O(n2) | O(n2) | O(n2) | O(1) |
| Insertion sort | O(1) | O(n2) | O(n2) | O(1) |
| Quick sort | O(n log n) | O(n2) | O(n log n) | O(1) |
| Merge sort | O(n log n) | O(n log n) | O(n log n) | O(n) |
| Heap sort | O(n log n) | O(n log n) | O(n log n) | O(1) |
| Counting sort | O(n+k) | O(n+k) | O(n+k) | O(n+2k) |
| Radix sort | O(n.k/s) | O(2s.n/k) | O(n.k/s) | O(n) |
| Bucket sort | O(n.k) | O(n2.k) | O(n.k) | O(n.k) |
| Bogo sort | O(n!) | Doesn´t finish | O(n.n!) |  |

**Bubble sort:** Los algoritmos de burbuja, selección e inserción (Bubble sort, selection sort, insertion sort) a pesar de tener complejidades temporales no tan deseables son los más fáciles y rápidos de implementar por eso mismo se suelen usar con frecuencia en casos donde no se manejan muchos datos. Para un grupo de elementos ordenado o casi ordenado estos algoritmos tienen a ser más eficientes.

**Counting sort:** El algoritmo por cuentas (Counting sort) tiene como limitante que solo se puede usar en números enteros.

**Radix sort:** El algoritmo radix (Radix sort) ordena enteros procesando sus dígitos de forma individual, aun así, no está solo limitado a trabajar con números enteros, también puede trabajar con números de coma flotante especialmente formateados.

**Quick sort:** El algoritmo rápido (Quick sort) es un algoritmo basado en la técnica divide y vencerás. donde divide el conjunto de elementos a ordenar en subconjuntos, escogiendo un pivote como base. Cada vez que organiza estos subconjuntos elige un nuevo pivote y vuelve a repetir el proceso. Por cuestiones prácticas es común escoger como pivote el elemento que esté más a la derecha en el subconjunto.  
Quicksort puede tener problemas de eficiencia al momento de tratar con conjuntos de elementos ordenados o casi ordenados donde el mayor elemento esté a la derecha del conjunto. Aun así es considerado el algoritmo de ordenamiento mas rapido hablando de propósitos generales

**Merge sort:** El algoritmo por mezcla (Merge sort) es un algoritmo basado en divide y vencerás, que divide continuamente una lista por la mitad. Si la lista está vacía o tiene un solo ítem, se ordena por definición (el caso base). Si la lista tiene más de un ítem, dividimos la lista e invocamos recursivamente un ordenamiento por mezcla para ambas mitades. Una vez que las dos mitades están ordenadas, se realiza la operación fundamental. La mezcla es el proceso de tomar dos listas ordenadas más pequeñas y combinarlas en una sola lista nueva y ordenada.

Este algoritmo suele ser más fácil de implementar y más eficiente que su rival (Quicksort) cuando ordenada listas enlazadas.

**Heap sort:** El algoritmo de ordenamiento por montículos (Heapsort) consiste en hacer uso de la estructura conocida como apilamiento, la cual es un tipo de árbol binario completo, en donde el elemento mayor o menor del arreglo (Dependiendo como se desea organizar) esta en el nodo raíz (cima). Este algoritmo va extrayendo el nodo raíz y pone el valor en el arreglo ya ordenado, después vuelve a construir el montículo pero sin el elemento que se acabo de extraer, y así va colocando todos los valores en el arreglo, el cual ya quedaría ordenado.

Heapsort a pesar de tener las mismas limitaciones en complejidad temporal que merge sort, es más eficiente en complejidad temporal que el mencionado anteriormente.

**Bucket sort:** El algoritmo por casilleros (Bucket sort) consiste en distribuir todos los elementos a ordenar

entre un número finito de casilleros. Cada casillero solo puede contener elementos que cumplan con determinadas condiciones. Estas condiciones deben ser excluyentes entre sí, para evitar que un elemento pueda estar en 2 casilleros a la vez. Se organizan los elementos dentro de cada casillero y luego se ordenan todos los casilleros.

**Bogo sort:** El algoritmo aleatorio (Bogosort) es un algoritmo inefectivo basado en el paradigma de ensayo y error, en este se mezclan las entradas del arreglo de forma aleatoria y después evalúa si el arreglo quedó ordenado, en caso de que no estar ordenado sigue tratando de organizarlo de forma aleatoria. En el peor de los casos, el algoritmo nunca termina de ordenar el arreglo.

**Criterios de selección**

* Complejidad temporal n log n o menor.
* Facilidad de implementación
* Que el algoritmo de ordenamiento pueda trabajar con números de coma flotante
* Que en el conjunto de algoritmos seleccionados los algoritmos se complementen entre sí, es decir, si uno no es eficiente manejando listas ordenadas inversamente debe haber otro que sea eficiente manejando ese tipo de listas y así.

**Algoritmos descartados**:

**Bogo sort:** Es inefectivo, puede haber casos en los que nunca llegue a una solución. Además, su complejidad temporal es demasiado ineficiente para nuestros requerimientos.  
**Bubble sort, selection sort y insertion sort:** Complejidad temporal ineficiente para la resolución del problema, aunque trabajan muy buenos con grupos pequeños de datos y son muy fáciles de implementar.

**Heap sort:** Es uno de los mas estables, muy eficiente, pero toma bastante tiempo de implementar y no cumple un propósito general.

**Bucket sort:** Es uno de los mas eficientes en complejidad temporal en cuanto a la ejecución del ordenamiento, pero se deben estructurar los datos de cierta manera antes de poder ejecutarlo dificultando su implementación.

**Radix sort:** Trabaja separando dígitos lo cual puede llegar a ser muy ineficiente al trabajar con números de coma flotante.

**Evaluación y selección de la mejor solución:**

Por lo anterior, los algoritmos que no fueron descartados son Quick sort, Merge sort y Counting sort.

Son los mas eficientes en complejidad temporal además de que se complementan entre sí, Counting sort es el mas eficiente pero solo trabaja con enteros, por lo que cuando se trabajen con números de coma flotante solo estarán disponibles Quick sort y Merge sort.

Quick sort se vuelve ineficiente con listas casi ordenadas mientras que Merge sort funciona muy bien con ese tipo de listas.

Por otro lado, Merge sort tiene una complejidad espacial de O(n) lo que puede generar problemas en problemas muy grandes, en esos casos Quick sort será más eficiente.

Los tres algoritmos elegidos para la implementación son:

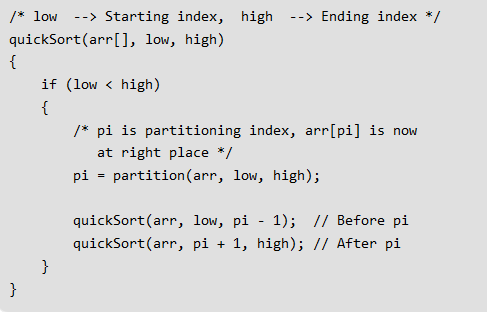
* Quick sort
* Merge sort
* Counting sort

**Pseudocódigo de los algoritmos de ordenamiento seleccionados:**

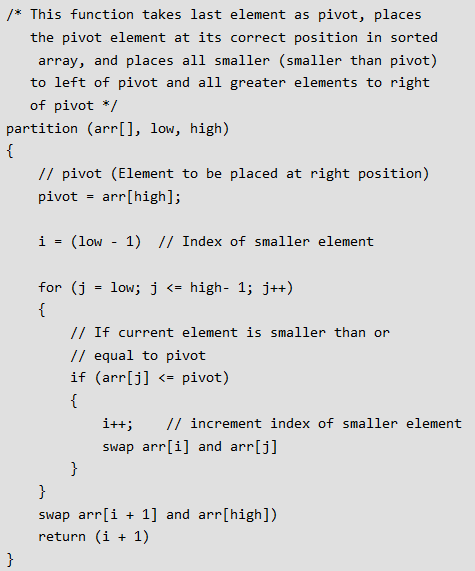
**Quicksort:**

Quicksort necesita un algoritmo de partición y otro de ordenamiento:

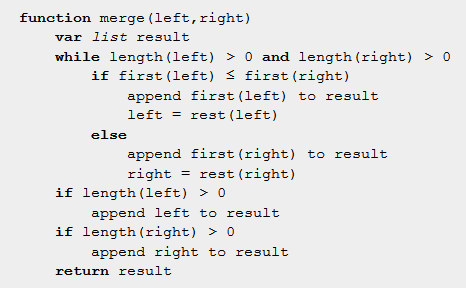
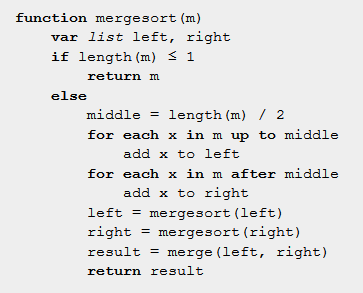
Ordenamiento:



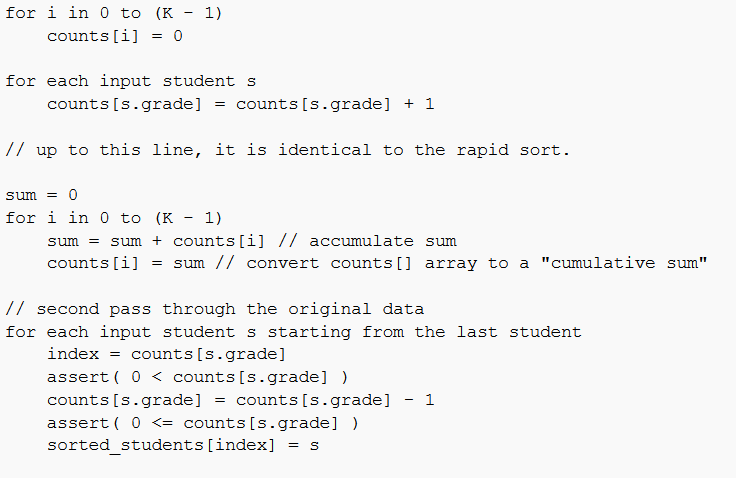
Partición:



**Merge sort**:

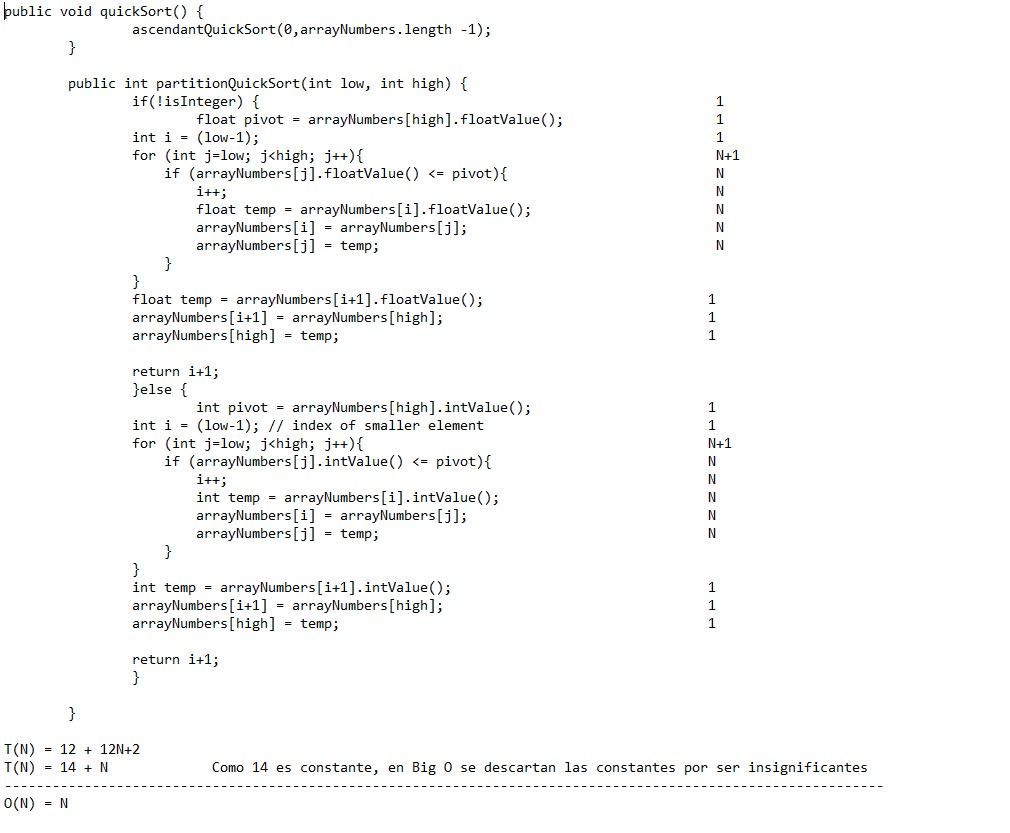


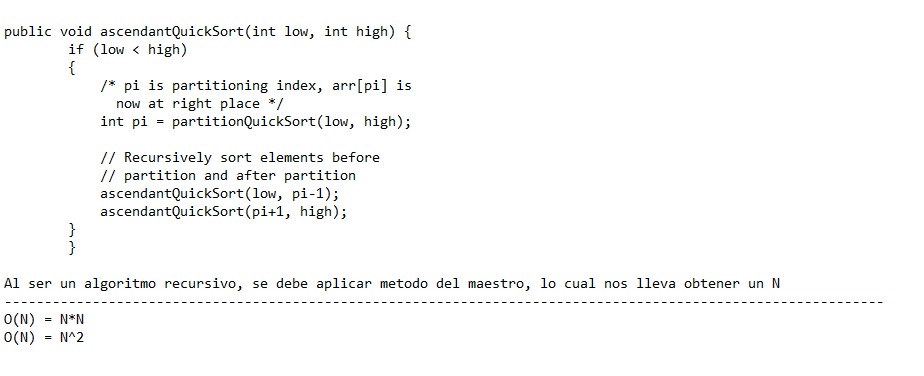
**Counting sort:**

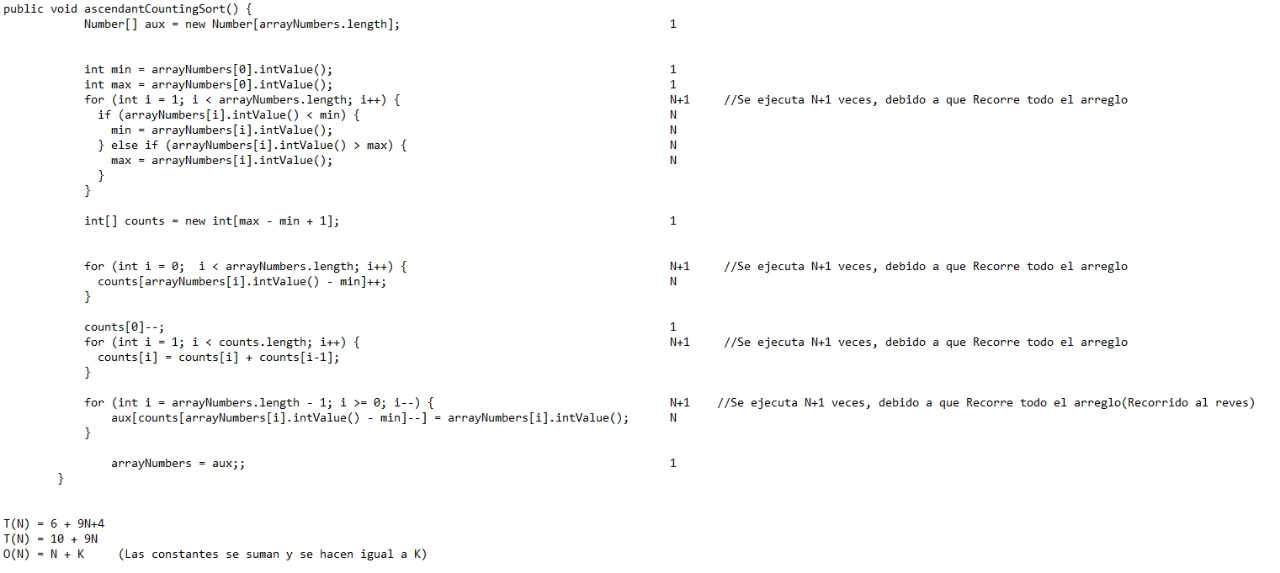


**Complejidad Temporal:**

Complejidad QuickSort:

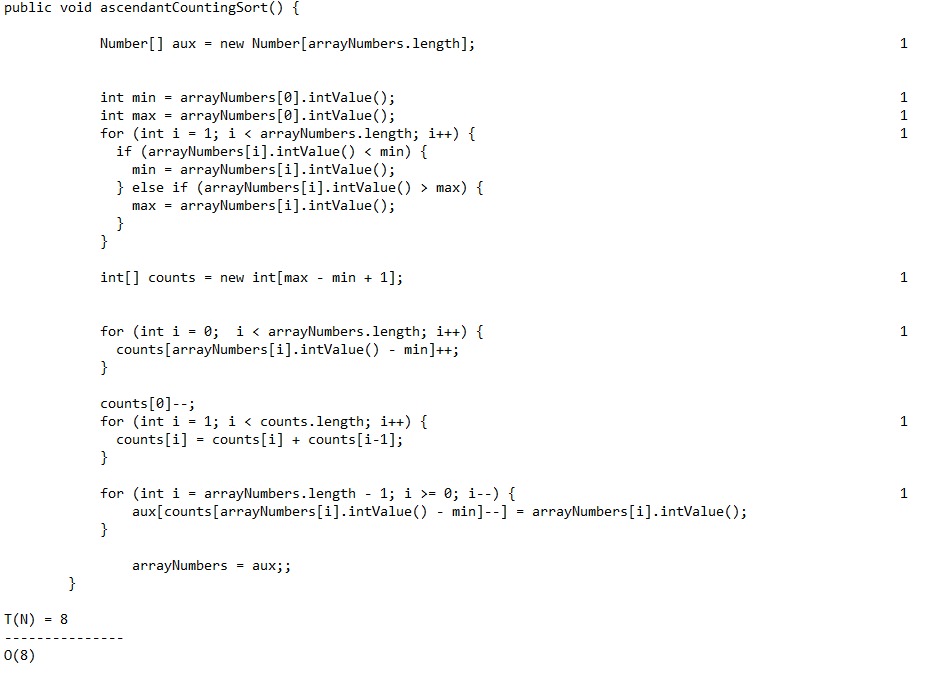


Complejidad temporal Counting sort:

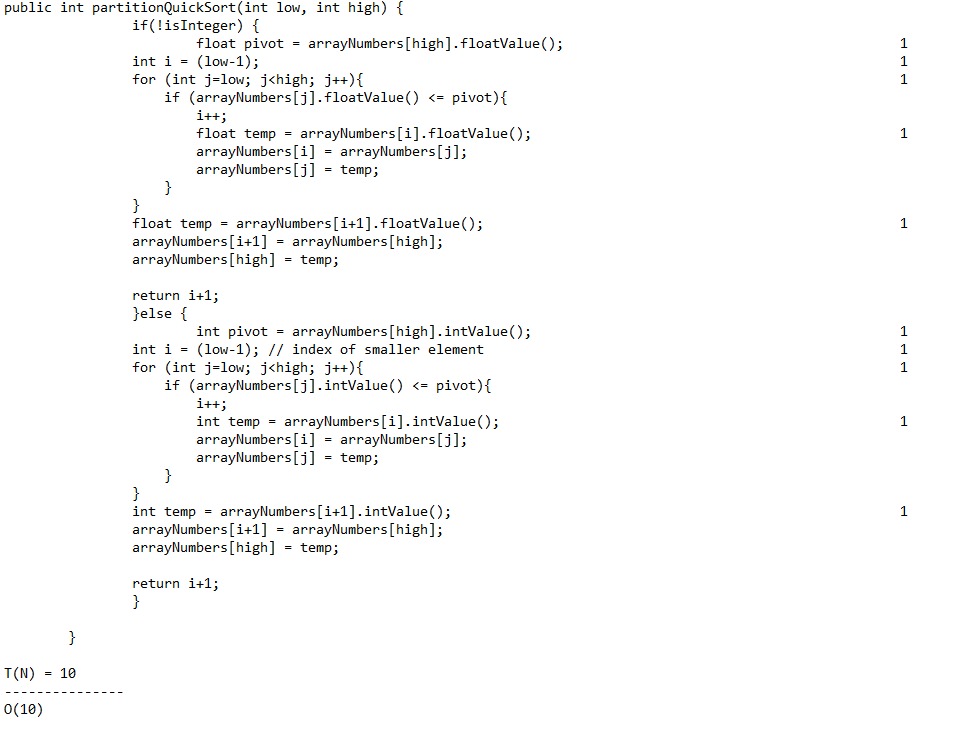


**Complejidad Espacial:**

Counting sort:



Quick sort:



Merge sort

La complejidad del espacio del Merge Sort siempre será O(n) incluida con las matrices. Si dibuja el árbol del espacio, parecerá que la complejidad del espacio es O(n log n). Sin embargo, como el código es un código de Profundidad Primero, siempre se expandirá a lo largo de una rama del árbol, por lo tanto, el uso de espacio total requerido siempre estará limitado por O(3n) = O(n). Mergesort es un algoritmo que emplea el método "Divide y vencerás" y es O (n log n) porque la entrada se reduce a la mitad repetidamente.