

QuickSort, se basa en la estrategia divide y vencerás, el cual consiste en descomponer un problema en serie de subproblemas de menor tamaño, resolverlos recursivamente y combinar las soluciones para obtener una general (que es la solución del problema original). Así, el algoritmo, al dividir, realiza comparaciones para hacer correcta la división de subproblemas: elementos menores o iguales al objeto a comparar (pivote), a la derecha, y elementos mayores al pivote a la izquierda.

Este proceso estará guiado entre un Servidor y un Cliente. Así, el cliente inicialmente le otorgará la lista de elementos a ordenar al Servidor. Y éste último se encargará de realizar los procesos correspondientes de QuickSort sobre la lista con ayuda de sub-servidores; cada sub-server tendrá una parte de la lista otorgada por el Server, así cada uno se comprometerá en resolver/ordenar la sub-lista otorgada y entregársela correctamente a su Servidor. Finalmente, el Servidor tendrá la lista completa nuevamente, gracias a sus servidores ayudantes, y podrá darle entrega de la solución ordenada al Cliente.

Visual Paradigm Professional(Rafa(Universidad Icesi))



1. **Inicio del Cliente:** El proceso comienza con el método main, que inicializa el comunicador y establece la conexión con el Servidor. Luego, el Cliente entra en un bucle donde selecciona una opción del menú y, si se seleccionan datos para ordenar, se inicia el proceso de ordenamiento en el Servidor.

2. **Interacción con el Servidor:** El Cliente proporciona al Servidor una lista de elementos a ordenar. Esto se hace a través del método *startQuickSort(a: Info[], c: ReturnCallBack)* del Servidor.
3. **Creación de Sub-Servidores:** El Servidor crea varios Sub-Servidores utilizando el método *createSubServer()*. Cada Sub-Servidor se encarga de una porción específica de la lista.
4. **Ordenamiento de la Lista:** Cada Sub-Servidor ejecuta el algoritmo QuickSort en su porción de la lista utilizando el método *quickSortArray(array: int[], low: int, high: int)*. Este método también incluye la partición del array y la ordenación de los datos.
5. **Recepción de Resultados:** Una vez que cada Sub-Servidor ha ordenado su porción de la lista, los resultados se devuelven al Servidor a través del método *receiveResult(result: UnionResult)*.
6. **Entrega de la Lista Ordenada:** Finalmente, el Servidor entrega la lista completa y ordenada al Cliente.

## Diseño de pruebas.

### Entorno de pruebas:

(atributos del pc donde corren las pruebas)

### Datos de entrada:




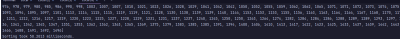

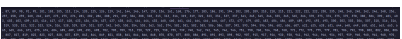



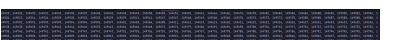
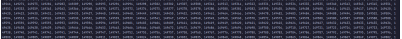

Los datos de entrada, son una serie de archivos de texto (.txt) que presentan una cantidad determinada de números enteros negativos y positivos, en este caso presentamos los siguientes tamaños:

- Lista de 100 números
- Lista de 1.000 números
- Lista de 10.000 números
- Lista de 100.000 números

- Lista de 1.000.000 números

Antes de ingresar los datos de entrada, se inicializan una serie de sub servidores o workers, donde cada uno recibe la tarea de ordenar una parte de los datos de entrada.

### Casos de prueba:

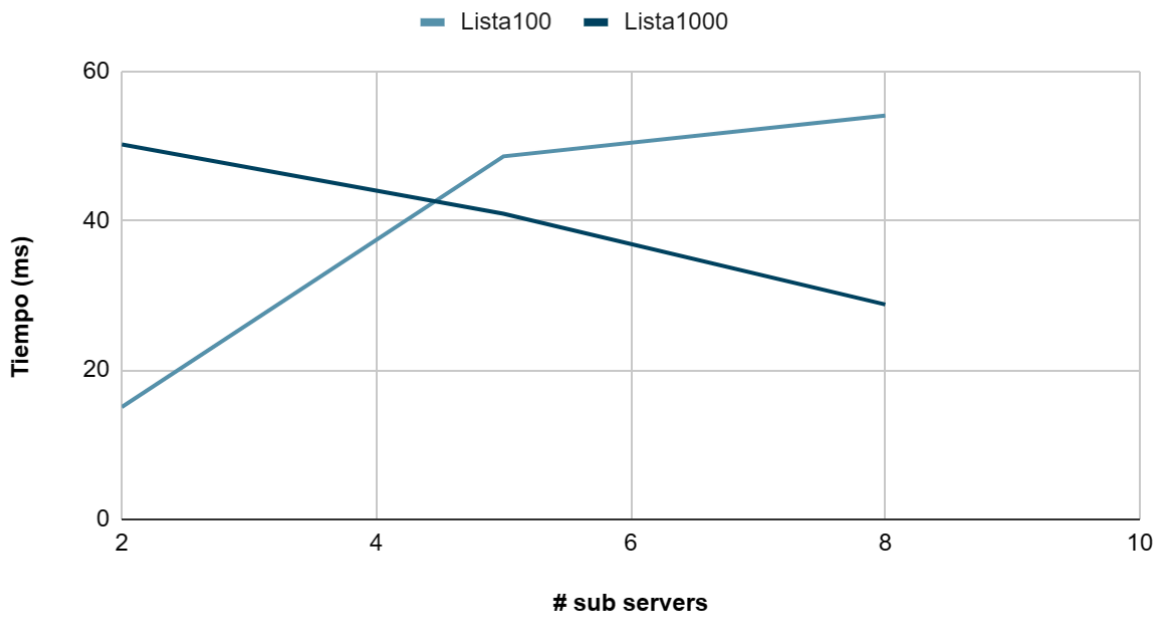
Número de prueba	Archivo de entrada	Número de sub servers	Tiempo de operación	Resultado (Captura de pantalla del resultado)
1	lista100.txt	2	15.0258ms	
2	lista100.txt	5	48.604ms	
3	lista100.txt	8	54.0508ms	
4	lista1000.txt	2	50.2013ms	
5	lista1000.txt	5	40.9228ms	
6	lista1000.txt	8	28.7734ms	
7	lista10000.txt	2	541.4562ms	
8	lista10000.txt	5	253.7128ms	
9	lista10000.txt	8	236.7279ms	
10	lista100000.txt	8	3589.2101ms	
11	lista100000.txt	9	3189.1831ms	
12	lista100000.txt	10	3075.277ms	

### Resultado de pruebas.

Graficamos los resultados para observar las tendencias dependiendo de las variables.

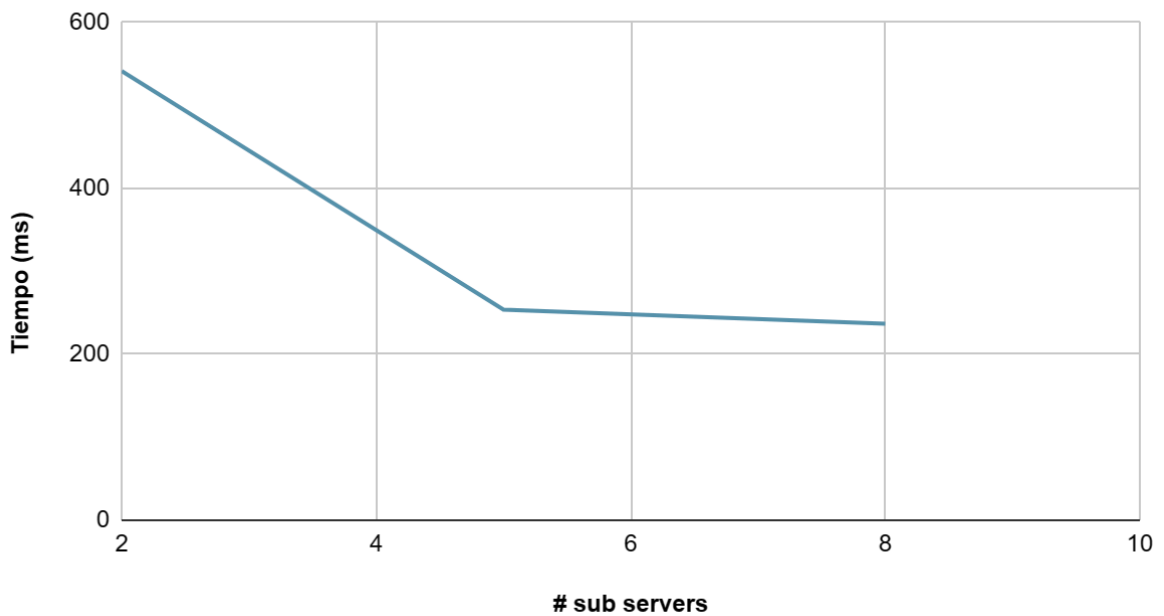
Gráfica de la lista de 100 números y la lista de 1,000 números.

## Tiempo de ejecución



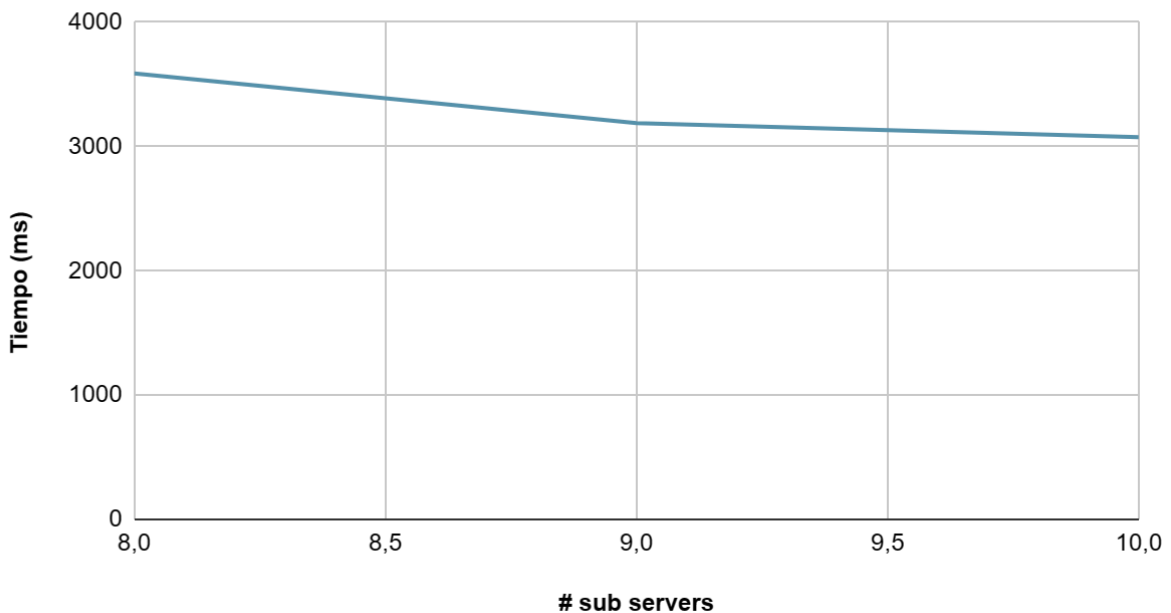
Gráfica de la lista de 10,000 números.

## Tiempo de ejecución



Gráfica de la lista de 100,000 números.

## Tiempo de ejecución



## Análisis de rendimiento.

Al analizar las gráficas de los resultados podemos llegar a unas conclusiones sobre el rendimiento del algoritmo de ordenamiento distribuido.

Primero observamos que en la lista de 100 números, a mayor cantidad de subservers el tiempo de ejecución aumenta, esto debido a tener una lista con pocos dígitos es más costoso el envío del segmento de datos que el ordenamiento así. Haciendo que no valga la pena tantos sub-servers para una tarea tan “pequeña”.

Por otro lado, observamos que en la lista de 1,000 y de 10,000, tienen un comportamiento similar, a mayor cantidad de subservers el tiempo disminuye, puesto que hablamos de tareas más complejas por su cantidad. Sin embargo y con base en el comportamiento de la lista de 100 números, podríamos afirmar que hay una cantidad de subservers en donde comienza a ser más costoso el envío del segmento de datos que el ordenamiento, adquiriendo un comportamiento similar al del primer análisis.

Para la lista de 100,000 observamos una diferencia. El mínimo de sub-servers en prueba son 8. Puesto que al ejecutar las pruebas en un solo terminal, los sub-servers se sobrecargan llegando a un “Stack overflow”, ya que estos llegan a su límite de segmento de pila, por eso se toma como base 8 sub-servers. Viendo el comportamiento, obtenemos el mismo que en el caso anterior, a mayor número de servers, menor el tiempo de ejecución.

## Conclusión

El uso de QuickSort de manera distribuida con subservidores es eficiente para ordenar grandes conjuntos de datos. Sin embargo, para conjuntos de datos pequeños, el costo de enviar segmentos de datos a los subservidores puede superar el beneficio del ordenamiento en paralelo, lo que resulta en un aumento del tiempo de ejecución.

Para conjuntos de datos medianos y grandes, el tiempo de ejecución disminuye a medida que aumenta el número de subservidores, lo que indica que el ordenamiento en paralelo puede acelerar significativamente el proceso. Sin embargo, hay un punto de inflexión en el que aumentar aún más el número de subservidores no mejora el rendimiento. Por lo tanto, es crucial equilibrar el número de subservidores para garantizar la eficiencia del algoritmo.