El cliente requiere un programa que permita evaluar el desempeño de 3 algoritmos de ordenamiento buscando el más o los más eficientes ordenando grandes cantidades de números y estos últimos con valores arbitrarios.

Problema…

“Desarrollar un programa que permita al usuario ordenar números aleatorios con diferentes algoritmos de ordenamiento eficientes”

|  |  |  |  |
| --- | --- | --- | --- |
| Requerimiento | Definición | Entrada | salida |
| Ingreso de valores a ordenar | El usuario podrá definir cuáles son los valores que quiere ordenar | Valores digitados por el usuario | Valores ordenados |
| Generación aleatoria de valores | El programa deberá generar valores de tamaños arbitrarios a ordenar.  El usuario podrá decir si los valores están:   * Ordenados ascendentemente * Ordenados descendentemente * Totalmente aleatorio * Parcialmente desordenados | Instrucciones del usuario | Arreglo ordenado |
| Decisión del algoritmo | El programa deberá decidir cuál de los algoritmos de ordenamiento implementados es el mejor para los datos ingresados | Cantidad de datos a ordenar | Algoritmo elegido y su ejecución |
| Reporte del proceso | Al finalizar el ordenamiento el programa mostrara cuanto tardo en el proceso. |  | Tiempo de ejecución |

Algoritmo de ordenamiento

En computación y matemáticas un algoritmo de ordenamiento es un algoritmo que pone elementos de una lista o un vector en una secuencia dada por una relación de orden, es decir, el resultado de salida ha de ser una permutación o reordenamiento de la entrada que satisfaga la relación de orden dada.

Los algoritmos de ordenamiento están pueden clasificarse por su complejidad computacional, el tiempo que tardan en realizarse y por su estabilidad.

Algunos ejemplos son:

|  |  |  |  |
| --- | --- | --- | --- |
| Estables | | | |
| Nombre original | Complejidad | Memoria | Método |
| Bubble sort | O(*n*²) | O(1) | Intercambio |
| Bucket sort | O(*n*) | O(*n*) | No comparativo |
| Merge sort | O(*n* log *n*) | O(*n*) | Mezcla |
| Inestables | | | |
| Shell sort | O(n**1.25**) | O(1) | Inserción |
| Quick sort | Promedio: O(*n* log *n*), peor caso: O(*n*²) | O(log *n*) | Partición |

Fuente: <https://es.wikipedia.org/wiki/Algoritmo_de_ordenamiento>

Un dato siempre lleva asociado un tipo de dato, que determina el conjunto de valores que puede tomar.

En Java toda la información que maneja un programa está representada por dos tipos principales de datos:

* Datos de tipo básico o primitivo.
* Referencias a objetos.

Los datos de interés son los primitivos y son los siguientes.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **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 IEEE 754 | 4 |  3.4x10-38 a  3.4x1038 | 0.0 | Float |
| **double** | Numérico en Coma flotante de precisión doble Norma IEEE 754 | 8 |  1.8x10-308 a  1.8x10308 | 0.0 | Double |
|  |  |  |  |  |  |

Fuente: <http://puntocomnoesunlenguaje.blogspot.com/2012/04/tipos-de-datos-java.html>

Para resolver el problema…

Nos enfrentamos a tamaños de arreglos con dimensiones desconocidas y tamaños de datos desconocidos por ello tenemos que evaluar al menos unos cuantos casos posibles.

Un caso posible sería tener un arreglo pequeño de al menos 100 a 1000 elementos con un tamaño en los datos pequeño y que no sean reales si no enteros, la solución sería un algoritmo que no consumiera mucha memoria y que fuera rápido.

Otro caso podría ser un arreglo con tamaños medios de 1001 elementos a 10000 o más elementos hasta quizá 100000 elementos y números reales se requeriría un algoritmo más adaptado a la velocidad de ordenamiento.

El último caso serían arreglos de más de 100001 elementos hasta el millón de elementos requería dividir el arreglo para trabajarlo desde segmentos y ordenarlo más eficientemente.

* para el primer caso si el arreglo es menor que 1000 datos y no está muy desordenado podemos usar el smooth sort
* para el primer caso si el arreglo es menor que 1000 datos y está bastante desordenado podemos implementar el heap sort.
* si el arreglo esta entre 1001 y 100000 datos podríamos usar bucket sort para organizar datos de gran valor enteros
* si el arreglo esta entre 1001 y 100000 datos podríamos usar radix sort para organizar datos de gran valor incluyendo reales
* para el ultimo caso usariamos una combinación del merge sort y el quick sort para ordenar grandes volumenes de datos
* si el anterior no logra ser implementado usaremos el merge sort.
* [Merge sort](https://www.geeksforgeeks.org/merge-sort/)
* Quick sort
* Bucket sort
* Heap sort
* Smooth sort
* Radix

(comparar las complejidades y manera en la que opera. para escoger las tres)