**INFORME**

**1.Contexto Problemático**

Una empresa de fabricación de microprocesadores requiere de la implementación de varios algoritmos de ordenamiento que permita un mayor rendimiento para realizar instrucciones básicas de su próximo coprocesador matemático.

**2.Desarrollo de la Solución**

Para resolver el caso anterior se eligió el Método de la Ingeniería del libro “Introduction to Engineering” de Paul Wright. Cada uno de los pasos del Método de la Ingeniería se presentan a continuación especificando la descripción y el procedimiento de cada uno de estos pasos.

**2.1 Identificación del problema:**

De acuerdo al enunciado se identificaron los siguientes problemas:

1. Permitir al usuario ingresar los números que desea ordenar
2. El usuario debe tener la posibilidad de generar números aleatorios (valores enteros y de coma flotante).
3. La configuración de la cantidad de números a ordenar y el intervalo en el cual se generarán estos dependerá del usuario
4. El usuario debe tener la posibilidad de indicar si los números a generar deben ser todos diferentes o pueden ser repetidos
5. La generación aleatoria del programa debe dar la libertad al usuario de elegir:
   1. Si se ordenan los números ingresados.
   2. Si se ordenan los números de manera inversa
   3. Los números se distribuyan aleatoriamente
   4. La cantidad de valores que se desean desordenar están dados por un porcentaje ingresado por el usuario

**Definición del problema:** El problema que se nos presenta en el enunciado es encontrar algoritmos de ordenamiento que nos permita organizar -en este caso- números enteros y de coma flotante. De acuerdo a las indicaciones que nos dé el usuario como lo son: cantidad de números ordenados, intervalo y porcentaje de desorden.

**2.2 Recopilación de la información.**

Para una debida solución de nuestro problema inicial, requerimos recopilar información que nos ayude a esclarecer la manera mas optima para poder resolverlo. de esta manera entenderemos el problema a fondo con todas las implicaciones que este contenga. Además de generar diferentes ideas que nos permitirá tomar el mejor camino para llevarlo a cabo.

***Algoritmo de ordenamiento***

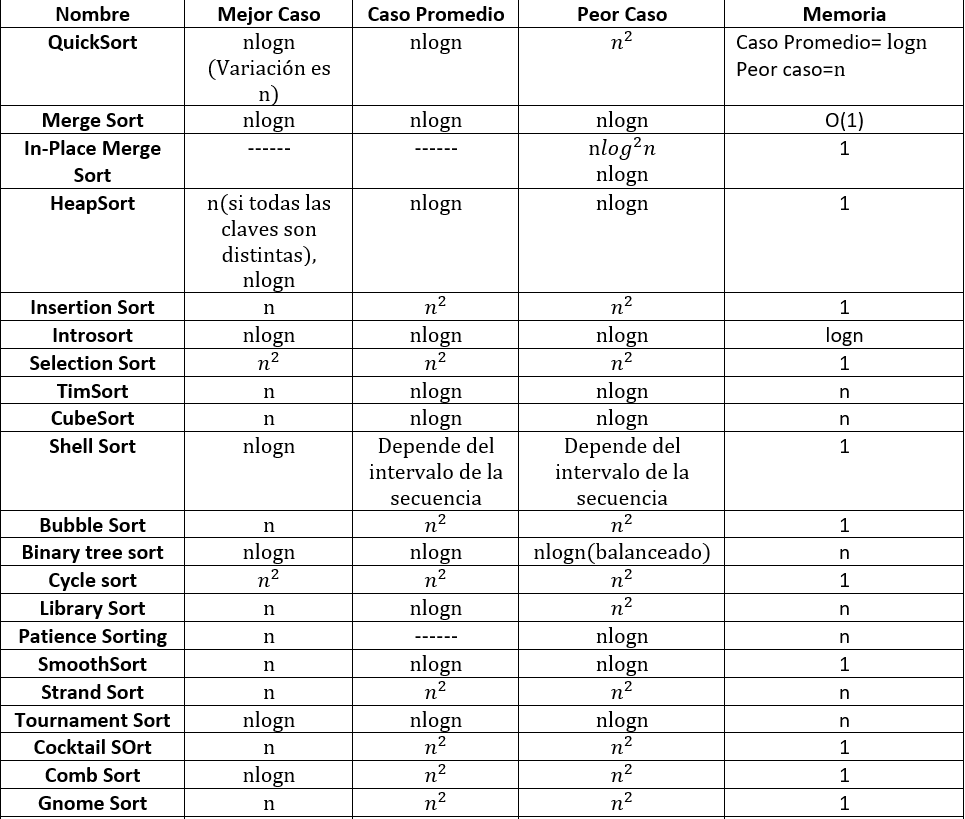
En computación y matemáticas un **algoritmo de ordenamiento** es un algoritmo que pone elementos de una lista o un vector en una secuencia dada por una relación de orden, es decir, el resultado de salida ha de ser una permutación —o reordenamiento— de la entrada que satisfaga la relación de orden dada. Las relaciones de orden más usadas son el orden numérico y el orden lexicográfico.

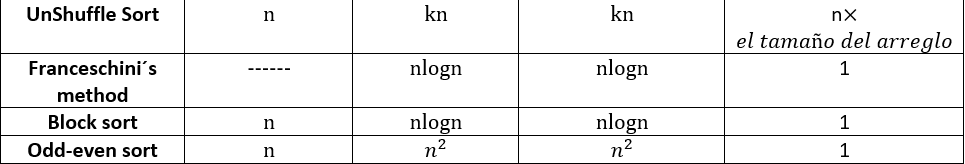
Los diferentes métodos de ordenamiento se dividen de acuerdo a su estabilidad, por un lado se encuentran los algoritmos de ordenamiento estables, que son aquellos que mantienen un preorden. Por otro lado se encuentran los inestables, los cuales pueden cambiar el orden relativo de registros con claves iguales. Un algoritmo inestable puede ser implementado de tal forma que sea estable.

Algoritmo In Place: Es un algoritmo que transforma la entrada sin estructura de datos auxiliares. Se caracteriza porque la entrada suele ser sobrescrita por la salida a medida que se ejecuta el algoritmo. El algoritmo In Place tiene la propiedad de que opera sobre el arreglo original, el cual va siendo modificado, en vez de crear un nuevo arreglo y regresarlo.

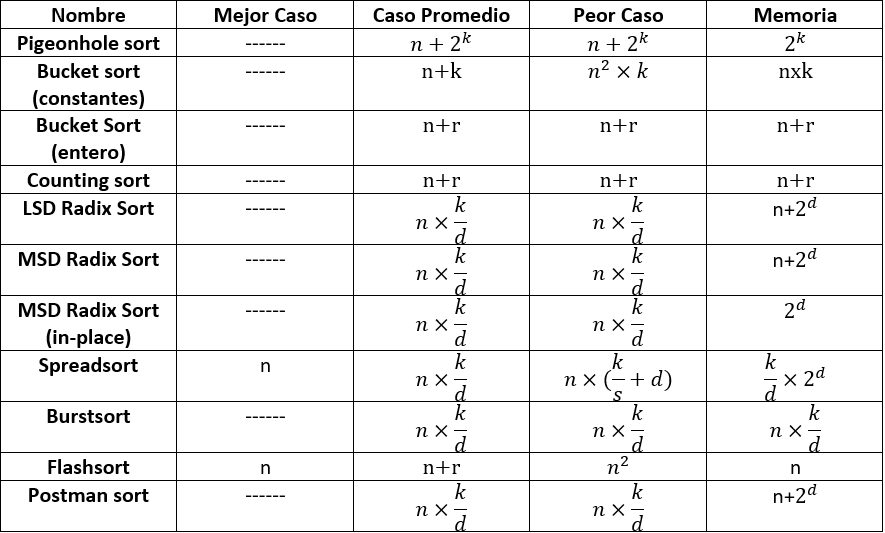
***Tipos de algoritmos de ordenamiento***

***Ordenamiento por Comparación***

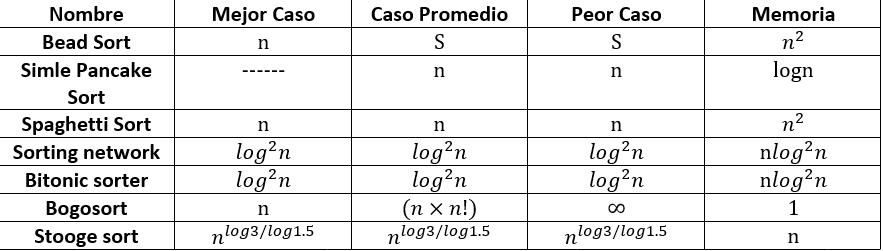
******



**Ordenamientos sin comparación**

****

**Ordenamientos que requieren condiciones especiales para su funcionamiento**

****

**Tim Sort**

Timsort es un algoritmo de clasificación es[t](https://en.wikipedia.org/wiki/Category:Stable_sorts)able híbrido , derivado del género de fusión y el tipo de inserción , diseñado para funcionar bien en muchos tipos de datos del mundo real.

para su ejecución el Tim Sort fue diseñado para aprovechar las corridas de elementos ordenados consecutivos que ya existen en la mayoría de los datos del mundo real, ejecuciones naturales . Se itera sobre los elementos de recopilación de datos en ejecuciones y, al mismo tiempo, combina esas ejecuciones. Cuando hay ejecuciones, hacer esto disminuye el número total de comparaciones necesarias para ordenar completamente la lista.

En el peor de los casos , Timsort toma O (n log n) comparaciones para ordenar una matriz de n elementos. En el mejor de los casos, que ocurre cuando la entrada ya está ordenada, se ejecuta en tiempo lineal, lo que significa que es un algoritmo de clasificación adaptativo.

**Insertion Sort**

Este es uno de los métodos más sencillos. Consta de tomar uno por uno los elementos de un arreglo y recorrerlo hacia su posición con respecto a los anteriormente ordenados. Así empieza con el segundo elemento y lo ordena con respecto al primero. Luego sigue con el tercero y lo coloca en su posición ordenada con respecto a los dos anteriores, así sucesivamente hasta recorrer todas las posiciones del arreglo.

**Selection Sort**

El método de ordenamiento por selección consiste en encontrar el menor de todos los elementos del arreglo e intercambiarlo con el que está en la primera posición. Luego el segundo más pequeño, y así sucesivamente hasta ordenar todo el arreglo.

**Shellsort**

Denominado así por su desarrollador Donald Shell (1959), ordena una estructura de una manera similar a la del Bubble Sort, sin embargo no ordena elementos adyacentes sino que utiliza una segmentación entre los datos. Esta segmentación puede ser de cualquier tamaño de acuerdo a una secuencia de valores que empiezan con un valor grande (pero menor al tamaño total de la estructura) y van disminuyendo hasta llegar al '1'. Una secuencia que se ha comprobado ser de las mejores es: ...1093, 364, 121, 40, 13, 4, 1. En contraste, una secuencia que es mala porque no produce un ordenamiento muy eficiente es ...64, 32, 16, 8, 4, 2, 1.

Su complejidad es de O(n1.2) en el mejor caso y de O(n1.25) en el caso promedio.

**Bubble Sort**

El bubble sort, también conocido como ordenamiento burbuja, funciona de la siguiente manera: Se recorre el arreglo intercambiando los elementos adyacentes que estén desordenados. Se recorre el arreglo tantas veces hasta que ya no haya cambios. Prácticamente lo que hace es tomar el elemento mayor y lo va recorriendo de posición en posición hasta ponerlo en su lugar.

**Cocktail Sort**

La manera de trabajar de este algoritmo es ir ordenando al mismo tiempo por los dos extremos del vector. De manera que tras la primera iteración, tanto el menor como el mayor elemento estarán en sus posiciones finales. De esta manera se reduce el número de comparaciones aunque la complejidad del algoritmo sigue siendo O(n²).

**Merge Sort**

El método Quicksort divide la estructura en dos y ordena cada mitad recursivamente. El caso del MergeSort es el opuesto, es decir, en éste método de unen dos estructuras ordenadas para formar una sola ordenada correctamente.

Tiene la ventaja de que utiliza un tiempo proporcional a: n log (n), su desventaja radica en que se requiere de un espacio extra para el procedimiento.

Este tipo de ordenamiento es útil cuando se tiene una estructura ordenada y los nuevos datos a añadir se almacenan en una estructura temporal para después agregarlos a la estructura original de manera que vuelva a quedar ordenada.

**Heap Sort**

Este método garantiza que el tiempo de ejecución **siempre** es de:

O(n log n)

El significado de *heap* en ciencia computacional es el de una cola de prioridades (priority queue). Tiene las siguientes características:

* Un heap es un arreglo de n posiciones ocupado por los elementos de la cola. (Nota: se utiliza un arreglo que inicia en la posición 1 y no en cero, de tal manera que al implementarla en C se tienen n+1 posiciones en el arreglo.)
* Se mapea un árbol binario de tal manera en el arreglo que el nodo en la posición i es el padre de los nodos en las posiciones (2\*i) y (2\*i+1).
* El valor en un nodo es mayor o igual a los valores de sus hijos. Por consiguiente, el nodo padre tiene el mayor valor de todo su subárbol.

Heap Sort consiste esencialmente en:

* convertir el arreglo en un heap
* construir un arreglo ordenado de atrás hacia adelante (mayor a menor) repitiendo los siguientes pasos:
  + sacar el valor máximo en el heap (el de la posición 1)
  + poner ese valor en el arreglo ordenado
  + reconstruir el heap con un elemento menos
* utilizar el mismo arreglo para el heap y el arreglo ordenado.

**Partition-Exchange Sort o Quicksort**

Es un método de ordenamiento recursivo y en lenguajes en donde no se permite la recursividad esto puede causar un retraso significativo en la ejecución del quicksort.

Su tiempo de ejecución es de n log2n en promedio.

**Stupid Sort (BogoSort)**

*Stupid Sort*, es un algoritmo de búsqueda particularmente inefectivo basado en el paradigma de ensayo y error. No es útil para ordenar, pero puede ser utilizado con propósitos educativos para contrastarlo con algoritmos más efectivos. También ha sido usado como ejemplo en programación lógica.

Si stupid sort fuera utilizado para ordenar un mazo de cartas, consistiría en verificar primero si el mazo está en orden, y si no lo está, entonces deberíamos mezclar las cartas de forma aleatoria, verificar de nuevo si están ordenadas y así sucesivamente hasta que por una mezcla al azar encontremos el mazo ordenado. El nombre bogosort proviene de la palabra *bogus*.

Este algoritmo de ordenamiento es probabilístico por naturaleza. Si se aplica a un arreglo donde todos los elementos son distintos, el número esperado de comparaciones es asintóticamente equivalente a (e-1)n!, y el número esperado de intercambios (swaps) en el caso promedio es igual a(n-1)n!.

En el peor caso el número de comparaciones e intercambios no está acotada. Es decir, no hay certeza de que el algoritmo termine. El mejor caso es cuando la lista original está ordenada, entonces se realizan {\displaystyle n-1} comparaciones y ningún intercambio.

**Radix Sort**

Es un algoritmo de ordenamiento conocido en el mundo de la programación que ordena enteros a partir de sus dígitos de forma individual.

El siguiente ejemplo funciona con una lista desordenada de números.

1. En primer lugar los va ordenando tomando en consideración el número menos significativo (la unidad) del más pequeño al más grande. Como se muestra en el punto 1.
2. Luego, a partir de la lista que obtuvimos en el paso anterior, ordenamos los números de menor a mayor considerando esta vez la decena de cada uno de ellos. Como se observa en el punto 2.
3. Finalmente comprobamos que la lista fue ordenada satisfactoriamente mediante este procedimiento.

**Ordenamiento con Árbol binario**

El ordenamiento con árbol binario es un algoritmo de ordenamiento, el cual ordena sus elementos haciendo uso de un árbol binario de búsqueda. Se basa en ir construyendo poco a poco el árbol binario introduciendo cada uno de los elementos, los cuales quedarán ya ordenados. Después, se obtiene la lista de los elementos ordenados recorriendo el árbol en inorden.

Insertar elementos en un árbol binario de búsqueda tiene una complejidad O(log n). Entonces, agregar n elementos a un árbol cualquiera da como resultado una complejidad O(n log n). Además, recorrer los elementos del árbol en inorden tiene complejidad O(n).

**Counting Sort**

El ordenamiento por cuentas (counting sort en inglés) es un algoritmo de ordenamiento en el que se cuenta el número de elementos de cada clase para luego ordenarlos. Sólo puede ser utilizado por tanto para ordenar elementos que sean contables (como los números enteros en un determinado intervalo, pero no los números reales, por ejemplo).

El primer paso consiste en averiguar cuál es el intervalo dentro del que están los datos a ordenar (valores mínimo y máximo). Después se crea un vector de números enteros con tantos elementos como valores haya en el intervalo [mínimo, máximo], y a cada elemento se le da el valor 0 (0 apariciones). Tras esto se recorren todos los elementos a ordenar y se cuenta el número de apariciones de cada elemento (usando el vector que hemos creado). Por último, basta con recorrer este vector para tener todos los elementos ordenados.

**Bucket Sort**

Es un algoritmo de ordenamiento que distribuye todos los elementos a ordenar entre un número finito de casilleros. Cada casillero sólo puede contener los elementos que cumplan unas determinadas condiciones. En el ejemplo esas condiciones son intervalos de números. Las condiciones deben ser excluyentes entre sí, para evitar que un elemento pueda ser clasificado en dos casilleros distintos. Después cada uno de esos casilleros se ordena individualmente con otro algoritmo de ordenación (que podría ser distinto según el casillero), o se aplica recursivamente este algoritmo para obtener casilleros con menos elementos.

**Eficiencia Algorítmica**

Es usado para describir aquella propiedades de los algoritmos que están relacionadas con la cantidad de recursos utilizados por el algoritmo. Un algoritmo debe ser analizado para determinar el uso de los recursos que realiza.

**Complejidad Temporal**

Para analizar un algoritmo generalmente se usa la complejidad temporal para obtener un estimado del tiempo de ejecución expresado en función del tamaño de la entrada. El resultado es típicamente expresado en notación O grande. Esto suele ser útil para comparar algoritmos, especialmente cuando se necesita procesar una gran cantidad de datos.

**Complejidad Espacial**

Se enfoca en el uso de memoria (usualmente RAM) por los algoritmos mientras son ejecutados. Es utilizada para obtener un estimado del uso de memoria principal expresado mediante una función según el tamaño de la entrada. El resultado es expresado usualmente en notación O grande.

**Fuentes:**

https/es.wikipedia.org

http://ict.udlap.mx/people/ingrid/Clases/IS211/Ordenar.html

http://c.conclase.net/orden/

https://arq232.wordpress.com/2012/08/30/algoritmo-de-ordenamiento-radix-sort/

https://www.geeksforgeeks.org/timsort/

https://www.infopulse.com/blog/timsort-sorting-algorithm/

https://pereiratechtalks.com/analisis-de-algoritmos-de-ordenamiento/

**2.3 Búsqueda de Soluciones Creativas**

Para este paso todas las ideas daban una solución al problema de una o cierta manera, en las que utilizamos algoritmos de ordenamiento pensados de cierta forma que ayuden a la solución del problema teniendo en cuenta en qué situaciones cada uno puede hacer la tarea más rápido que otro.

* **Alternativa 1:** Crear nuevos algoritmos de ordenamiento que tengan una excelente eficiencia en cuanto a la complejidad temporal y espacial.
* **Alternativa 2:** Utilizar los métodos que hemos trabajado desde semestres anteriores, que son los más conocidos por nosotros, los cuales son inserción, selección y burbuja.
* **Alternativa 3:** Ordenar con el algoritmo de ordenamiento Radix Sort, que ordena procesando cada unos de los dígitos que componen a cada número.
* **Alternativa 4:** Hacer uso del método Cocktail Sort dado que va ordenando en una misma interacción, el menor y el mayor. Por lo tanto se estaría ahorrando -en teoría- tiempo de ejecución.
* **Alternativa 5:** Para el caso de ordenar números que no necesitan ser comparados entre sí, es decir que deben ser organizados solo teniendo la cantidad de veces que se repite dicho número, se puede hacer uso del ordenamiento por casilleros.

* **Alternativa 6:** Dado que el método Quick Sort y el método de inserción tienen buena sinergia, una alternativa sería combinarlos para ordenar los números, cuando la cantidad de números restantes a ordenar sea entre 8-15 entraría en juego el método de inserción.

* **Alternativa 7:** Puesto que el uso de árboles binarios ya lo vimos en el curso anterior, podríamos tener como alternativa hacer uso del ordenamiento con árbol binario, agregando los datos para generar el árbol y luego leerlo en inorden para tener la solución.
* **Alternativa 8:** Si deseamos ordenar un grupo de números de acuerdo a un intervalo determinado, entonces se podría hacer uso del método de ordenamiento por cuentas. (Counting Sort)
* **Alternativa 9:** Un método el cual se puede utilizar en la solución de nuestro problema es el Shell sort, por el cual se pueden ordenar los números de manera ascendente, es decir desde el menor hasta el mayor, y que es considerado mucho más eficiente que el método por inserción.
* **Alternativa 10:** Utilizar el algoritmo de ordenamiento por mezcla (Merge Sort) ya que es un algoritmo muy práctico para cantidades grandes. Utiliza la técnica del divide y vencerás y de esta forma crea sub-arreglos que los ordena para luego juntarlos y dejar ordenado el arreglo principal.
* **Alternativa 11:** Hacer uso del algoritmo Heap Sort, que consiste en separar los datos en regiones ordenadas y no ordenadas, luego se encarga de encoger la región no ordenada sacando el mayor elemento y lo mueve a la región ordenada y así sucesivamente hasta que queden ordenados los datos.
* **Alternativa 12:** Utilizar un algoritmo híbrido, como el caso del TimSort que es un algoritmo bastante optimizado siendo la combinación del Merge Sort y el Insertion Sort.
* **Alternativa 13:** Como última instancia y si la cantidad de números a ordenar es poca, podríamos hacer uso del Stupid Sort (Bogo sort) dado que no tendría muchas combinaciones posibles por hacer.

**2.4 Transición de las Ideas a los Diseños Preliminares:**

**Ideas Descartadas:**

**Consenso de la alternativa 1: *Creación de nuevos métodos de ordenamiento***

Esta alternativa resulta interesante para un proceso investigativo que permita un aprendizaje acerca de los métodos de ordenamiento. pero dado a las limitaciones que tenemos en recursos -el tiempo como principal ítem- no resulta una solución viable en estos momentos.

**Consenso de la alternativa 2: *Uso de algoritmos de ordenamiento vistos anteriormente***

Gracias a los estudios que hemos hecho a lo largo del pasado curso de programación y el que estamos cursando, se ha decidido que no se hará uso de estos algoritmos, pues la teoría nos ha mostrado que existen mejores algoritmos con mayor rapidez que nos ayudará a solucionar de una mejor manera nuestro problema inicial.

***Consenso de la alternativa 3: Uso del Radix Sort***

Este es un excelente algoritmo de ordenamiento, primero porque no se hacen comparaciones entre elementos, funciona en tiempo lineal y es estable, pero para nuestro caso que se presentarán números enteros arbitrariamente grandes sería una gran desventaja ya que el total de tiempo que se tarda es proporcional a la longitud del número más grande y al número de elementos a ordenar.

**Consenso de la alternativa 4: *Uso del Cocktail Sort (Burbuja bidireccional)***

Este algoritmo es una mejora del burbuja, ya que va arrastrando al mayor de izquierda a derecha y cuando se devuelve de derecha a izquierda va arrastrando al menor, de tal forma que hace el burbuja en dos sentidos. Pero a pesar de ser una mejora no deja de ser un algoritmo con complejidad O(n²), por tanto decidimos descartar esta idea porque necesitamos algoritmos más rápidos.

**Consenso de la alternativa 7 : *Uso del Binary Tree Sort***

Siendo los árboles binarios una buena alternativa para los ordenamientos, decidimos descartar la idea ya que cuando los datos se agregan al árbol y este queda desbalanceado, su peor caso da como resultado una complejidad de O(n²). Como requerimos de algoritmos lo suficientemente rápido para todos los casos en lo posible, por tanto esta idea queda descartada.

**Consenso de la alternativa 9 : *Uso del Shell Sort***

Este es un buen algoritmo de ordenamiento, pero al tener en cuenta su caso promedio O(n log² n ) decidimos no tenerlo en cuenta ya que su complejidad es mejor que O(n²) pero no tanto como O(n log n). Además en su peor caso se encuentra el O(n²).

**Consenso de la alternativa 13: *Uso del Stupid Sort (Bogo Sort)***

Este método sería el menos indicado para nuestro trabajo. si los cantidad de números fuese poca seria tomado en cuenta. pero dado a los requerimientos de la empresa, la cantidad de números a ordenar es muy grande, lo cual hace que el uso de este algoritmo se convierta en una alternativa no viable, pues no sabemos exactamente en cuánto tiempo termine y para cantidad de números muy grandes, la cantidad de iteraciones sería un problema impensable.

Luego de una detallada revisión a las otras alternativas nos conduce a lo siguiente:

**Alternativa 5: Bucket Sort**

* Es un algoritmo que no realiza comparaciones.
* Su mejor caso y el promedio son de complejidad O(n).
* Su peor caso es de complejidad O(n²).

**Alternativa 6: Quick Sort**

* Es un algoritmo que realiza comparaciones.
* Su mejor caso y el promedio son de complejidad O(n log n), pero su mejor caso puede llegar a ser O(n).
* Su peor caso es de complejidad O(n²).
* Se pudo observar en un análisis que les presentaremos en los criterios de evaluación, que el Quicksort a pesar de tener el peor caso en una complejidad O(n²) es más rápido que el Heap Sort.

**Alternativa 8: Counting Sort**

* Es un algoritmo que no realiza comparaciones
* Todos sus casos son O(n) especificamente n+r donde r es el rango de números a ordenar.
* Es un algoritmo muy rápido.
* Sólo sirve con números enteros. (No es recomendable con números racionales)

**Alternativa 10: Merge Sort**

* Todos sus casos son O(n log n)
* Divide la entrada en varias parte que las organiza en sus mismas divisiones para luego juntarlas
* Cuando la entrada es muy grande no es muy eficiente en memoria.

**Alternativa 11: Heap Sort**

* El caso promedio y el peor son O(n log n).
* El mejor caso puede ser O(n), si todas las claves son distintas entonces O(n log n)
* La entrada la separa en montículos y luego va sacando los nodos mayores de cada montículo.
* Se pudo observar en un análisis que les presentamos en los criterios de evaluación, que el Heap Sort es superado en tiempo por el QuickSort a pesar de tener el peor caso en O(n²) y que sus casos mejor y promedio son el mismo O(n log n).

**Alternativa 12: Tim Sort**

* Es un algoritmo híbrido derivado del Merge Sort y el Insertion Sort.
* El peor y caso promedio son O(n log n).
* El mejor caso es O(n).

**2.5 Evaluación y Selección de la Mejor Solución:**

**Criterios:**

**-Criterio A:Completitud.** El algoritmo está en la capacidad de cumplir su función completamente, es decir que dé la solución esperada

[3]Ordenamiento completado

[2]Ordenamiento sin completar

[1]No hay un ordenamiento

**-Criterio B :Accesibilidad.** el algoritmo y su información correspondiente : complejidad temporal, pseudocódigo, funcionamiento, restricciones, beneficios, etc. se encuentra con facilidad y claridad.

[3] Acceso completo a la información, clara y concisa.

[2] Acceso medio a la información, requiere algo de trabajo hallar su claridad.

[1] la información es de difícil acceso y a su vez es complejo de entender.

**-Criterio C:Complejidad en el mejor caso.** Se debe tener presente que la complejidad del algoritmo sea mínima en el mejor caso posible buscando solucionar el problema de la mejor manera.

[3] su complejidad es de orden O(n)

[2]su complejidad es de orden O(n log n)

[1] su complejidad es de orden O(n²)

[0] su complejidad es de orden mayor a O(n²)

**-Criterio D:Complejidad en el caso promedio.** Se debe tener presente que la complejidad del algoritmo sea mínima en el caso promedio buscando solucionar el problema de la mejor manera.

[3] su complejidad es de orden O(n)

[2]su complejidad es de orden O(n log n)

[1] su complejidad es de orden O(n²)

[0] su complejidad es de orden mayor a O(n²)

**-Criterio E:Complejidad en el peor caso.** Se debe tener presente que la complejidad del algoritmo sea mínima en el peor caso posible (el que nos interesa) buscando solucionar el problema de la mejor manera.

[3] su complejidad es de orden O(n)

[2]su complejidad es de orden O(n log n)

[1] su complejidad es de orden O(n²)

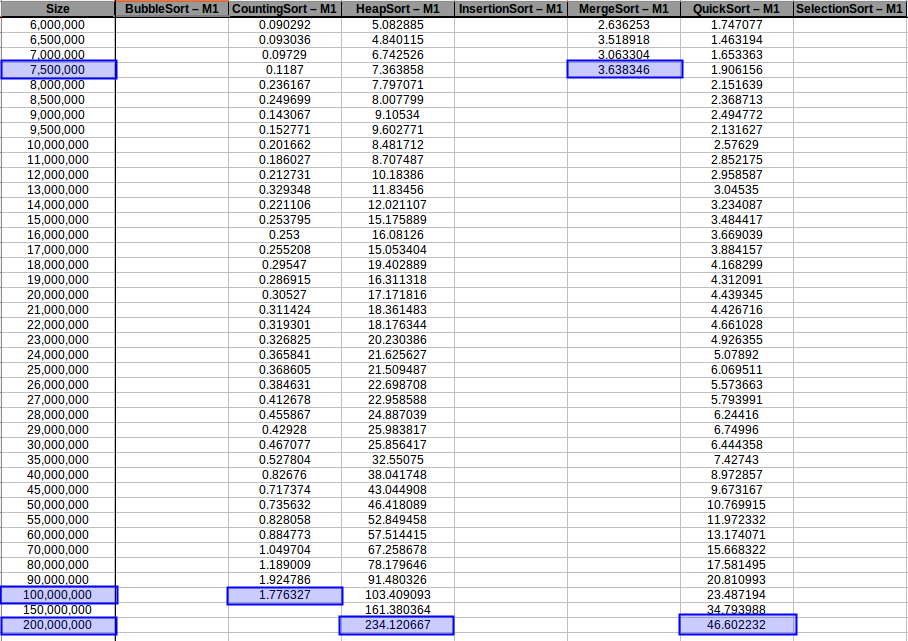
[0] su complejidad es de orden mayor a O(n²).

**-Criterio F: Tiempo que tarda el algoritmo en ordenar.** Se tiene en cuenta el tiempo que tarda en segundos el algoritmo de ordenamiento. **(Este criterio está basado en el análisis de Sergio Alexander Florez en la página Pereira Tech Talks.** [**https://pereiratechtalks.com/analisis-de-algoritmos-de-ordenamiento/**](https://pereiratechtalks.com/analisis-de-algoritmos-de-ordenamiento/)

[1] Es rápido.

[-1] No es rápido.

[0] No aplica.

****

*Imagen que muestra los tiempos que tarda un algoritmo ordenando cierta cantidad de datos.*

**Evaluación:** Evaluando los criterios anteriores en las alternativas que se mantienen se obtiene que:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | **CRITERIOS** | | | | | | |
| **Nombre del método** | **A** | **B** | **C** | **D** | **E** | **F** | **Total** |
| **Bucket Sort** | **3** | **2** | **3** | **3** | **1** | **0** | **12** |
| **Quick Sort** | **3** | **3** | **3** | **2** | **1** | **1** | **13** |
| **Counting Sort** | **3** | **3** | **3** | **3** | **3** | **1** | **16** |
| **Merge Sort** | **3** | **3** | **2** | **2** | **2** | **1** | **13** |
| **Heap Sort** | **3** | **3** | **3** | **2** | **2** | **-1** | **12** |
| **Tim Sort** | **2** | **3** | **3** | **2** | **2** | **0** | **12** |

**Selección:**

De acuerdo con la evaluación anterior se deben seleccionar los algoritmos Quicksort, Counting Sort y el Merge Sort, es decir las alternativas 6, 8 y 12 respectivamente, ya que obtuvieron los mejores puntajes.

**2.6 Preparación de Informes y Especificaciones:**

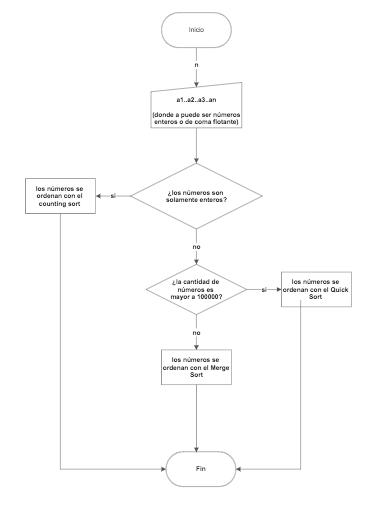
*Especificación del Problema* (en términos de entrada/salida)

* Problema: Ordenar números de manera eficiente
* Entradas: por una parte, los números que se desean ordenar. Y en otra instancia, la cantidad de números a ordenar y su intervalo.
* Salida: todos los números requeridos ordenados eficientemente.

*Consideraciones*

Se debe tener en cuenta las siguientes consideraciones para la solución del problema de la mejor manera

1. El usuario será quien decida si escribir los datos que se desean ordenar o si genera automáticamente los números de acuerdo a un tamaño e intervalo.
2. Se debe considerar el porcentaje de desorden ingresado por el usuario, donde él % indicará la cantidad de posiciones que se van a desordenar
3. El caso de números ordenados en el sentido contrario debe ser tenido en cuenta, pues es un servicio que requiere el usuario.
4. Se debe tener en cuenta las especificaciones de cada conjunto de números pues dependiendo de sus características el programa debe permitir/restringir el tipo de algoritmo.



**2.7 Implementación del Diseño:**

* **los requerimientos funcionales serán entregados en un archivo adicional**
* **el código del programa será entregado en lenguaje Java, usado mediante Java eclipse.**

**Anexos**

**Pseudocódigo de algunos métodos de ordenamiento**

* **Insertion Sort**

paso 1: [Para cada pos. del arreglo] For i <- 2 to N do  
paso 2: [Inicializa v y j] v <- a[i]  
 j <- i.  
paso 3: [Compara v con los anteriores] While a[j-1] > v AND j>1 do  
paso 4: [Recorre los datos mayores] Set a[j] <- a[j-1],  
paso 5: [Decrementa j] set j <- j-1.  
paso 5: [Inserta v en su posición] Set a[j] <- v.  
paso 6: [Fin]

* **Selection Sort**

paso 1: [Para cada pos. del arreglo] For i <- 1 to N do  
paso 2: [Inicializa la pos. del menor] menor <- i  
paso 3: [Recorre todo el arreglo] For j <- i+1 to N do  
paso 4: [Si a[j] es menor] If a[j] < a[menor] then  
paso 5: [Reasigna el apuntador al menor] min = j  
paso 6: [Intercambia los datos de la pos.  
 min y posición i] Swap(a, min, j).  
paso 7: [Fin] End.

* **Shell Sort**

void shellsort ( int a[], int n)  
{  
 int x,i,j,inc,s;  
   
 for(s=1; s < t; s++) /\* recorre el arreglo de incrementos \*/  
 {  
 inc = h[s];  
 for(i=inc+1; i < n; i++)   
 {  
 x = a[i];  
 j = i-inc;  
 while( j > 0 && a[j] > x)  
 {  
 a[j+h] = a[j];  
 j = j-h;  
 }  
 a[j+h] = x;  
 }  
 }  
}

* **Bubble Short**

paso 1: [Inicializa i al final de arreglo] For i <- N downto 1 do  
paso 2: [Inicia desde la segunda pos.] For j <- 2 to i do  
paso 4: [Si a[j-1] es mayor que el que le sigue] If a[j-1] < a[j] then  
paso 5: [Los intercambia] Swap(a, j-1, j).  
paso 7: [Fin] End.

* **Cocktail Sort**

Procedimiento Ordenacion\_Sacudida (v:vector, tam:entero)   
 Variables  
 i, j, izq, der, último: tipoposicion;  
 aux: tipoelemento;  
 Inicio  
 //Límites superior e inferior de elementos ordenados  
 izq <- 2  
 der <- tam  
 último <- tam  
   
 Repetir  
 //Burbuja hacia la izquierda}  
 //Los valores menores van a la izquierda  
 //der va disminuyendo en 1 hasta llegar a izq  
 Para i <- der hasta izq hacer  
 Si v(i-1) > v(i) entonces  
 aux <- v(i)  
 v(i) <- v(i-1)  
 v(i-1) <- aux  
 último <- i  
 Fin\_si  
 Fin\_para  
   
 izq <- último+1  
   
 //Burbuja hacia la derecha  
 //Los valores mayores van a la derecha  
 Para j <- izq hasta der hacer  
 Si v(j-1) > v(j) entonces  
 aux <- v(j)  
 v(j) <- v(j-1)  
 v(j-1) <- aux  
 último <- j  
 Fin\_si  
 Fin\_para  
   
 der <- último-1  
   
 Hasta (izq > der)  
 Fin

* **Merge Sort**

Procedimiento ***MergeSort***

void mergesort(int a[], int l, int r)  
{  
 int i,j,k,m,b[MAX];  
 if (r > l)   
 {  
 m = (r+l) /2;  
 mergesort(a, l, m);  
 mergesort(a, m+1, r);  
 for (i= m+1; i > l;i--)  
 b[i-1] = a[i-1];  
 for (j= m; j < r;j++)   
 b[r+m-j] = a[j+1];  
 for (k=l ; k <=r; k++)  
 if(b[i] < b[j])  
 a[k] = b[i++];  
 else  
 a[k] = b[j--];  
 }  
}

* ***Heapsort***

void heapsort(int a[], int N)  
{  
 int k;  
 for(k=N/2; k>=1; k--)  
 downheap(a,N,k);  
 while(N > 1)  
 {  
 swap(a,1,N);  
 downheap(a,--N,1);  
 }  
}

* **Quicksort**

void quicksort(int a[], int l, int r)  
{  
 int i,j,v;  
 if(r > l)  
 {  
 v = a[r];  
 i = l-1;  
 j = r;  
 for(;;)  
 {  
 while(a[++i] < v && i < r);  
 while(a[--j] > v && j > l);  
 if( i >= j)  
 break;  
 swap(a,i,j);   
 }  
 swap(a,i,r);  
 quicksort(a,l,i-1);  
 quicksort(a,i+1,r);  
 }  
}

* **Counting Sort**

comentario: listaValores es una matriz de valores enteros.  
 Entrada Función counting\_sort(listaValores)  
 ListaVariables  
 minValor comentario: el valor del elemento menor en la lista  
 maxValor comentario: el valor del elemento mayor en la lista  
 vAux comentario: una matriz de elementos auxiliar de tanto elementos como   
 define el rango minValor a maxValor   
 i, j, n comentario: contadores de bucle  
 valor comentario: valor del elemento actual en el bucle  
 tamaño comentario: cantidad de elementos en listaValores  
   
 Previos   
 comentario: Buscar valor mínimo y máximo  
 LlamadaaFuncion BuscarLimites(listaValores, minValor, maxValor)  
   
 comentario: Crea el vector auxiliar, con todos sus elementos a 0  
 vAux = NuevoVector(minValor,maxValor)  
   
 comentario: Obtiene la cantidad de elementos que contiene la matriz  
 tamaño = TotalElementosEn(listaValores)  
  
 comentario: Contar elementos. En el índice expresado por el valor, se van contando  
 las veces que aparece dicho valor en la matriz de entrada   
 comentario: Este bucle realiza una matriz de conteo, cada valor indica cuantas veces  
 aparece el valor representado por el índice en la lista de valores.  
 Inicio   
 i = 0   
 Hacer Mientras (i < tamaño)  
 valor = listaValores(i)  
 vAux(valor] = vAux(valor) + 1  
 i = i + 1  
 Repetir  
  
 comentario: Trasvasar la matriz de conteo a la lista, que queda así ya ordenada  
   
 i = minValor   
 j = 0  
 Hacer Mientras (i < maxValor)  
 comentario: Si para el índice 'i' se contó 1 o más elementos, transferir a la lista   
 Si vAux(i) > 0 Entonces   
 Para n Repetir Desde 1 Hasta vAux(i)   
 listaValores(j) = i   
 j = j + 1  
 Siguiente n  
 Fin si  
 Repetir   
 Fin  
 Salida Función  
  
comentario: Esta función auxiliar busca y devuelve el mayor y menor elementos de una matriz  
 Entrada Función BuscarLimites(listaValores, minValor, maxValor)  
 Variables  
 i comentario: contador del bucle   
   
 Inicio  
 minValor = listaValores(0)  
 maxValor = minValor  
   
 Para i Repetir Desde 1 Hasta TotalElementosEn(ListaValores)  
 Si listaValores(i) < minValor entonces   
 minValor = listaValores(i)  
 Ó Si listaValores(i) > maxValor entonces  
 maxValor = listaValores(i)  
 Fin Si  
 Siguiente i  
 Fin   
 Salida Función

* **Bucket Sort**

función bucket-sort(elementos, n)   
 casilleros ← colección de n listas  
 para i = 1 hasta longitud(elementos) hacer  
 c ← buscar el casillero adecuado  
 insertar elementos[i] en casillero[c]  
 fin para  
 para i = 1 hasta n hacer  
 ordenar(casilleros[i])  
 fin para  
 devolver la concatenación de casilleros[1],..., casilleros[n]