**ANÁLISIS**

**FASE 1.**

**Identificación del problema**

**Descripción del contexto:** Luego de estudiar los costos de implementación del ordenamiento como una operación nativa del coprocesador, una empresa de fabricación de microprocesadores desea implementar varios algoritmos de ordenamiento como instrucciones básicas de su próximo coprocesador matemático.

**Identificación y definición concreta del problema:**

* Una empresa quiere implementar varios algoritmos de ordenamiento como instrucciones básicas de su próximo coprocesador matemático.
* La solución al problema, deben ser 3 algoritmos de ordenamiento que permitan ordenar rápidamente, números enteros de tamaño arbitrariamente grande y números en formato de coma flotante de cualquier tamaño.

**Requerimientos Funcionales:**

|  |  |
| --- | --- |
| **NOMBRE** | **REQ 001 – Generación aleatoria** |
| **RESUMEN** | Permite generar aleatoriamente los valores (tanto **enteros** como de **coma flotante**), permitiendo configurar la cantidad total de números a generar y el intervalo en el cual se generan los números.  Permite indicar si los números a generar deben ser todos **diferentes** o pueden haber **repetidos.** |
| **ENTRADA** | * Intervalo en el que se generan los números. * Cantidad de números a generar * Elección de números diferentes o repetidos. |
| **SALIDA** | Los valores indicados se han generado aleatoriamente. |

|  |  |
| --- | --- |
| **NOMBRE** | **REQ 002 – Ordenar valores** |
| **RESUMEN** | Permite ordenar y mostrar la secuencia de valores ya ordenados, de menor a mayor. |
| **ENTRADA** | <No requiere> |
| **SALIDA** | Se han ordenado los valores. |

|  |  |
| --- | --- |
| **NOMBRE** | **REQ 003 – Ordenar aleatoriamente** |
| **RESUMEN** | Permite poner la secuencia de valores, en un orden aleatorio. |
| **ENTRADA** | <No requiere> |
| **SALIDA** | La secuencia de valores ha sido ordenada aleatoriamente. |

|  |  |
| --- | --- |
| **NOMBRE** | **REQ 004 – Ordenar inversamente** |
| **RESUMEN** | Permite ordenar la secuencia de valores inversamente (al revés). |
| **ENTRADA** | <No requiere> |
| **SALIDA** | Los valores han sido ordenados inversamente. |

|  |  |
| --- | --- |
| **NOMBRE** | **REQ 005 – Desordenar porcentualmente** |
| **RESUMEN** | Esta funcionalidad debe permitir que secuencia se genere en orden. Posteriormente, con base en el tamaño de la secuencia y el porcentaje (%) de desorden se obtiene un número (k) de cuantas posiciones deben estar desordenadas. Finalmente se generarán k/2 pares de posiciones diferentes y se intercambian los valores entre cada par de ellas. |
| **ENTRADA** | * Porcentaje (%) de desorden |
| **SALIDA** | Se ha desordenado porcentualmente la secuencia. |

|  |  |
| --- | --- |
| **NOMBRE** | **REQ 006 – Ingresar valores en la secuencia** |
| **RESUMEN** | Esta funcionalidad le debe permitir al usuario ingresar los valores de la secuencia manualmente. |
| **ENTRADA** | * Valores de la secuencia |
| **SALIDA** | Los valores han sido ingresados. |

**FASE 2.**

**Recopilación de la información necesaria:**

* **Coprocesador:** es un microprocesador de un ordenador utilizado como suplemento de las funciones del procesador principal (la CPU). Las operaciones ejecutadas por uno de estos coprocesadores pueden ser operaciones de aritmética en coma flotante, procesamiento gráfico, procesamiento de señales, procesador de texto, criptografía, etc.

Fuente:<https://docs.google.com/document/d/1-RqyZf9DvkJeZIo43N5RgZB8XZzhy7EyPH4XByz6Zck/edit>

* **Función de coprocesador:** Evitar que el procesador principal tenga que realizar tareas de cómputo intensivo. Puede acelerar el rendimiento del sistema por el hecho de esta descarga de trabajo en el procesador principal y porque suelen ser procesadores especializados que realizan las tareas para las que están diseñado más eficientemente. Fuente:<https://docs.google.com/document/d/1-RqyZf9DvkJeZIo43N5RgZB8XZzhy7EyPH4XByz6Zck/edit>
* **Algoritmo de ordenamiento:** En [computación](https://es.wikipedia.org/wiki/Computaci%C3%B3n) y [matemáticas](https://es.wikipedia.org/wiki/Matem%C3%A1ticas) es un [algoritmo](https://es.wikipedia.org/wiki/Algoritmo) que pone elementos de una [lista](https://es.wikipedia.org/wiki/Lista_(estructura_de_datos)) o un [vector](https://es.wikipedia.org/wiki/Vector_(programaci%C3%B3n)) en una secuencia dada por una [relación de orden](https://es.wikipedia.org/wiki/Orden_total), es decir, el resultado de salida ha de ser una [permutación](https://es.wikipedia.org/wiki/Permutaci%C3%B3n) —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](https://es.wikipedia.org/wiki/Orden_lexicogr%C3%A1fico). Ordenamientos eficientes son importantes para optimizar el uso de otros algoritmos (como los de [búsqueda](https://es.wikipedia.org/wiki/Algoritmo_de_b%C3%BAsqueda) y fusión) que requieren listas ordenadas para una ejecución rápida. También es útil para poner datos en forma canónica y para generar resultados legibles por humanos

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

* **Eficiencia algorítmica:** En [Ciencias de la Computación](https://es.wikipedia.org/wiki/Ciencias_de_la_Computaci%C3%B3n), el término es usado para describir aquellas propiedades de los [algoritmos](https://es.wikipedia.org/wiki/Algoritmos) que están relacionadas con la cantidad de [recursos](https://es.wikipedia.org/wiki/Recursos_computacionales) utilizados por el algoritmo. Un algoritmo debe ser analizado para determinar el uso de los recursos que realiza. La eficiencia algorítmica puede ser vista como análogo a la ingeniería de [productividad](https://es.wikipedia.org/wiki/Productividad) de un proceso repetitivo o continuo. <https://es.wikipedia.org/wiki/Eficiencia_algor%C3%ADtmica>
* **Complejidad temporal:** Para [analizar un algoritmo](https://es.wikipedia.org/wiki/An%C3%A1lisis_de_algoritmos) generalmente se usa la [complejidad temporal](https://es.wikipedia.org/wiki/Teor%C3%ADa_de_la_complejidad_computacional) 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. Estimaciones más detalladas se requieren para comparar algoritmos que procesan pequeñas cantidades de datos (de todas formas, en estos casos el tiempo no debería ser un problema). Algoritmos implementados para usar procesamiento paralelo de los datos son mucho más difíciles de analizar. https://es.wikipedia.org/wiki/Eficiencia\_algor%C3%ADtmica
* **Complejidad espacial:** Cuando queremos realizar un análisis de un algoritmo basado en su complejidad de espacio, consideramos solo el espacio de datos e ignore el espacio de instrucción y la pila ambiental. Eso significa que calculamos solo la memoria requerida para almacenar Variables, Constantes, Estructuras, etc. https://drive.google.com/file/d/1rbFCIzxceXdEichEG4gbKQMyV12-wjlF/view
* **Algoritmos 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.

Fuente: https://es.wikipedia.org/wiki/Algoritmo\_de\_ordenamiento

**FASE 3.**

**Búsqueda de soluciones creativas:**

Para este laboratorio usamos lluvia de ideas para llegar a las siguientes alternativas:

***Alternativa 1****:* Usar los algoritmos de ordenamientos conocidos previamente (Selección, Burbuja e Inserción).

***Alternativa 2:***Identificar los peores casos en los algoritmos de ordenamientos conocidos, y generar nuevas líneas de código para volverlos lo más eficientes posibles (Generar condiciones de parada que identifiquen cuando ya el arreglo esté ordenado, validar el tamaño del arreglo, pues si es 1, el arreglo ya está ordenado).

***Alternativa* 3**: Generar un algoritmo que identifique dependiendo el caso, cuál de los tres algoritmos de ordenamiento conocidos sirve para cada caso, esto se logra identificando el peor de los casos de cada uno de los algoritmos de ordenamiento y su complejidad en ellos.

***Alternativa 4:***Gracias a la investigación de algoritmos de ordenamiento, podemos saber que hay otros algoritmos más eficientes que podemos usar (Mergesort, Quicksort, Heapsort, Shellsort).

***Alternativa 5:***Investigar más a fondo cada uno de los algoritmos eficientes para saber cuál es el peor de los casos para cada algoritmo y así poderlos usar eficientemente.

***Alternativa 6:***Crear un nuevo algoritmo de ordenamiento que sea una mezcla entre los mejores algoritmos

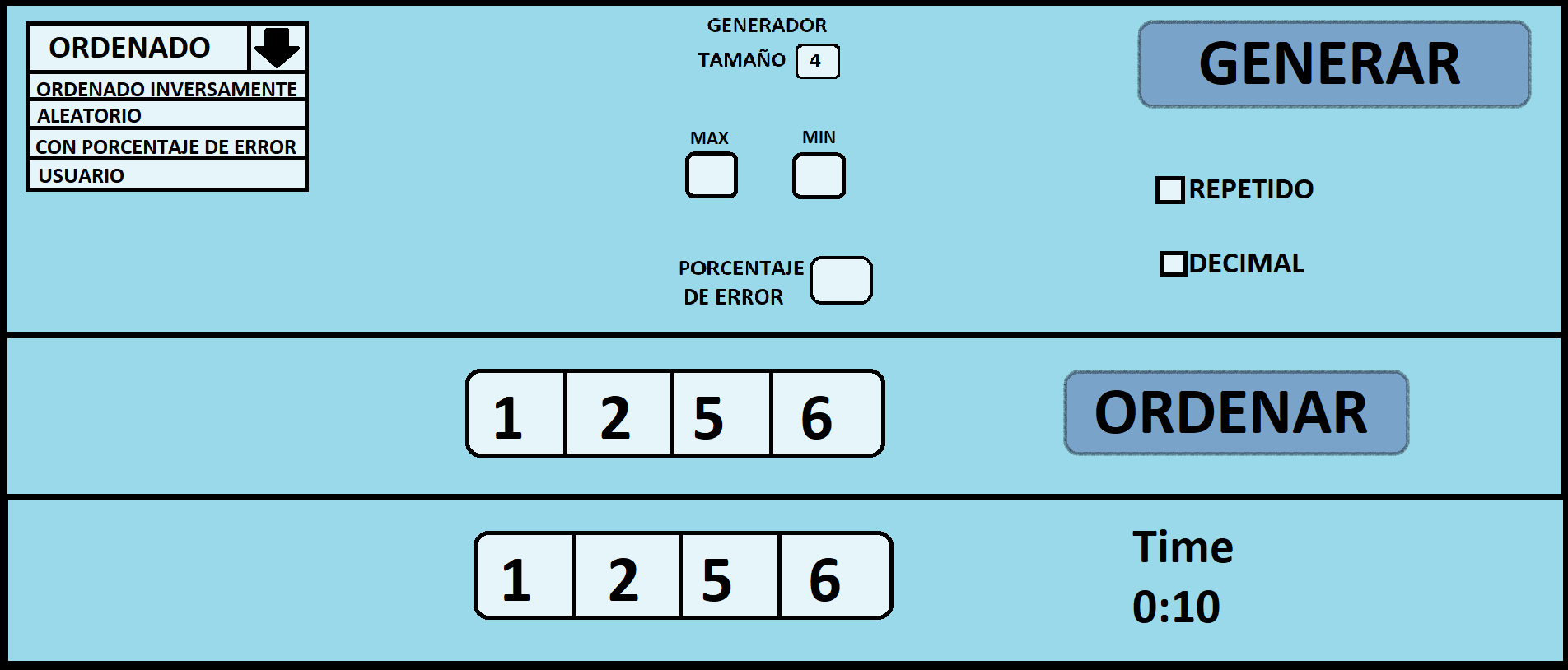
***Alternativa 7:***Generar un algoritmo que identifique dependiendo el caso, cuál de todos los algoritmos es más eficiente, pues dependiendo el tamaño del arreglo, un algoritmo eficiente puede ser superado por uno ineficiente.

**Alternativa 8**: Investigar e implementar los algoritmos radix sort, bucket sort, counting sort.

**Alternativa 9:** Pedir al usuario que genere números ordenados.

**Alternativa 10**: Sobrescribir métodos de java para efectuar las comparaciones (CompareTo y Comparable).

**DISEÑO:**



**FASE 4.**

**Transición de la formulación de ideas a los diseños preliminares:**

**Alternativa 9** no la usaremos porque la idea es que el usuario cree los numeros como le plazcan y no que los tenga que ordenar.

**Alternativa 10 y 6** no son necesarias porque usamos datos primitivos, entonces, no se necesita una comparación exhaustiva, un simple < o > sirve.

**Alternativa 2** no es recomendada implementar debido a que no poseemos la experiencia, para intentar mejorar un algoritmo que lleva tiempo bajo constante mejora, ni la necesidad de estar en ese plan de mejora para tratar solo tipos de datos primitivos.

**FASE 5.**

**Alternativa 1:** Los algoritmos de ordenamiento que hemos utilizado más frecuentemente (selección, burbuja, inserción), pueden ser útiles para implementar las utilidades que se necesitan en la solución a este problema. La experiencia en la utilización de estos algoritmos, puede ser útil para una mejor codificación.

**Alternativa 3 y 7:** Generar un algoritmo que identifique cuál de los tres algoritmos de ordenamiento implementados es más útil en cada caso, hará la aplicación más eficiente.

**Alternativa 4:** Hay otros algoritmos más eficientes que los tres que hemos utilizado con más frecuencia (mencionados en la alternativa 1), que podemos usar, estos son: Mergesort, Quicksort, Heapsort y Shellsort.

**Alternativa 5:** Investigar más a fondo cada uno de los algoritmos eficientes para saber cuál es el peor de los casos para cada algoritmo y así poderlos usar eficientemente.

**Alternativa 8: I**mplementar los algoritmos radix sort, bucket sort, counting sort. Los cuales tienen una manera rápida de ordenamiento.

Define criterios para evaluar las ideas. Explica en qué consiste cada criterio y todas las escalas que puede tener una alternativa evaluada con ese criterio. Evalua cada idea con base en dicho criterio y asigna un resultado de esa evaluación. Totaliza la evaluación para conocer, con base en los criterios elegidos, cuál o cuáles son las ideas que serán implementadas

**FASE 6.**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| PSEUDOCÓGIDO DE LOS TRES(3) ALGORITMOS MÁS RELEVANTES | | | | | | |
| 1. recibimos como parámetro una matriz de enteros, y dos enteros. 2. tomamos el primer elemento como pivote 3. < i > realiza la búsqueda de izquierda a derecha 4. < j > realiza la búsqueda de derecha a izquierda 5. Instanciamos una variable auxiliar, de tipo entero. 6. mientras no se crucen las búsquedas 7. busca elemento mayor que pivote 8. busca elemento menor que pivote 9. si no se han cruzado 10. intercambia auxiliar con el elemento en la posición <i> 11. Intercambia elemento <i> con elemento <j> 12. Intercambia elemento <j> con auxiliar 13. Salimos del “mientras” 14. se coloca el pivote en su lugar de forma que tendremos 15. los menores a su izquierda y los mayores a su derecha 16. ordenamos subarray izquierdo 17. ordenamos subarray derecho | | | | | | |
| 1. Se crea un algoritmo encargado de obtener el máximo valor en un arreglo. 2. Se crea un algoritmo encargado de hacer el conteo de tipo arreglo, según un dígito que ingresa por parámetro.    1. Se almacena el recuento de apariciones en un arreglo.    2. Cambiar el dígito “i” del arreglo de recuento para que la posición “i” ahora contenga la posición real de “i” en el arreglo de salida.    3. Se construye el arreglo de salida.    4. Se copia el arreglo de salida al arreglo ingresado por parámetro, para que ahora contenga números ordenados de acuerdo con el dígito actual. 3. Se crea un algoritmo encargado de ordenar un arreglo de tamaño n, usando Radix Sort.    1. Encuentra el número máximo para encontrar la cantidad de dígitos.(se hace llamado al segundo algoritmo)    2. Se hace un conteo ordenado para cada dígito.(se hace llamado al primer algoritmo) | | | | | | |
| SelectionSort() | | | | | | |
| DISEÑO DE CASOS PARA PRUEBAS UNITARIAS | | | | | | |
| Entrada | **Salida** | **Descripción** | **Nombre del método** | | **Clase** | **Escenario** |
| estandar |  |  |  | |  |  |
| Caso límite |  |  |  | |  |  |
| interesante |  |  |  | |  |  |
| estandar |  |  |  | |  |  |
| Caso limite |  |  |  | |  |  |
| interesante |  |  |  | |  |  |
| estándar |  |  |  | |  |  |
| Caso limite |  |  |  | |  |  |
| interesante |  |  |  | |  |  |
| ANÁLISIS DE COMPLEJIDAD TEMPORAL – ALGORITMO 1 | | | | | | |
| public static void quicksort(int A[], int izq, int der) { | | | Worst case. | | | |
| int pivote=A[izq]; | | | 1 | | | |
| int i=izq; | | | 1 | | | |
| int j=der; | | | 1 | | | |
| int aux; | | | 1 | | | |
| while(i<j){ | | | N-1 | | | |
| while(A[i]<=pivote && i<j) i++; | | | N-1 | | | |
| while(A[j]>pivote) j--; | | |  | | | |
| if (i<j) { | | |  | | | |
| aux= A[i]; | | |  | | | |
| A[i]=A[j]; | | |  | | | |
| A[j]=aux;  }  } | | |  | | | |
| A[izq]=A[j]; | | |  | | | |
| A[j]=pivote; | | |  | | | |
| if(izq<j-1) | | |  | | | |
| quicksort(A,izq,j-1); | | |  | | | |
| if(j+1 <der) | | |  | | | |
| quicksort(A,j+1,der);  } | | |  | | | |
| Función de tiempo del algoritmo:  Notación asintótica: | | | | | | |
| INVENTARIO DEL ALGORITMO 1/ ANÁLISIS DE COMPLEJIDAD ESPACIAL | | | | | | |
| Int pivote | | | | 1 | | |
| Int izq | | | | 1 | | |
| Int i | | | | 1 | | |
| Int j | | | | 1 | | |
| Int aux | | | | 1 | | |
| Int der | | | | 1 | | |
| Otros:  Int A[]  Int izq  Int der | | | | N  1  1 | | |
| Expresión en notación asintótica del espacio adicional utilizado: 8+N | | | | | | |
| ANÁLISIS DE COMPLEJIDAD TEMPORAL – ALGORITMO 2 | | | | | | |
| static int getMax(int arr[], int n)  {  int mx = arr[0];  for (int i = 1; i < n; i++)  if (arr[i] > mx)  mx = arr[i];  return mx;  }  static void countSort(int arr[], int n, int exp)  {  int output[] = new int[n];  int i;  int count[] = new int[10];  Arrays.fill(count,0);  for (i = 0; i < n; i++)  count[ (arr[i]/exp)%10 ]++;  for (i = 1; i < 10; i++)  count[i] += count[i - 1];  for (i = n - 1; i >= 0; i--)  {  output[count[ (arr[i]/exp)%10 ] - 1] = arr[i];  count[ (arr[i]/exp)%10 ]--;  }  for (i = 0; i < n; i++)  arr[i] = output[i];  }  static void radixsort(int arr[], int n)  {  int m = getMax(arr, n);  for (int exp = 1; m/exp > 0; exp \*= 10)  countSort(arr, n, exp);  } | | | 1  N  N  1  1  1  1  1  1  N  N  10  10  N  10  10  N  N  1  1 + n  n | | | |
| Función de tiempo del algoritmo: | | | | 1+N+N+1+1+1+1+1+1+N+N+10+10+N+10+10+N+N+1+1+N+N  9N + 49 | | |
| Notación asintótica: | | | | O(N) | | |
| INVENTARIO DEL ALGORITMO 2/ ANALISIS DE COMPLEJIDAD ESPACIAL | | | | | | |
| int output[] | | | N | | | |
| Int i | | | 1 | | | |
| Int mx | | | 1 | | | |
| Int count[] | | | N | | | |
| Int m | | | 1 | | | |
| Int exp | | | 1 | | | |
| Int i | | | 1 | | | |
| Otros:  Int exp int n  Int arr[] | | | 1  3  3N | | | |
| Expresión en notación asintótica del espacio adicional utilizado: 3N + 3 + 1 + 1 +1+1+N+1+1+N = 5N + 7 | | | | | | |
| ANÁLISIS DE COMPLEJIDAD TEMPORAL – ALGORITMO 3 | | | | | | |
| void sort(int arr[])  {  int n = arr.length;  for (int i = 0; i < n-1; i++)  {  int min\_idx = i;  for (int j = i+1; j < n; j++)  if (arr[j] < arr[min\_idx])  min\_idx = j;  int temp = arr[min\_idx];  arr[min\_idx] = arr[i];  arr[i] = temp;  } | | | 1  n-1  n-1  n-1  n-1  1  1  1  1 | | | |
| Función de tiempo del algoritmo: | | | 1+n-1+n-1+n-1+n-1+1+1+1+1  **1+4n** | | | |
| Notación asintótica: | | | | O(n) | | |
| INVENTARIO DEL ALGORITMO 2/ ANALISIS DE COMPLEJIDAD ESPACIAL | | | | | | |
| int min\_idx | | | 1 | | | |
| Int n | | | 1 | | | |
| Int j | | | 1 | | | |
| Int i | | | 1 | | | |
| Int temp | | | 1 | | | |
| Otros:  int arr[] | | | N | | | |
| Expresión en notación asintótica del espacio adicional utilizado: N+ 5 | | | | | | |

**Nota:** Usted debe entregar un archivo comprimido en formato zip de un directorio con únicamente 2 archivos: 1 archivo de informe en formato pdf con toda la documentación (de cada una de las fases del método y el análisis) y otro archivo comprimido de un directorio con los archivos de codificación en sus respectivos paquetes.

El nombre del archivo comprimido debe tener el formato: PRIMERAPELLIDOEST1\_PRIMERAPELLIDOEST2\_PRIMERAPELLIDOEST3.zip (tenga en cuenta que el separador entre cada apellido es un guion al piso).