Jessica Daniela Otero Fernandez
Cristian Andrés Gironza M
Santiago Figueroa Aguirre

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.

Jessica Daniela Otero Fernandez
Cristian Andrés Gironza M
Santiago Figueroa Aguirre

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 y matemáticas 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. Ordenamientos eficientes son importantes para optimizar el uso de otros algoritmos (como los de búsqueda 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, el término es usado para describir aquellas 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. La eficiencia algorítmica puede ser vista como análogo a la ingeniería de productividad de un proceso repetitivo o continuo. https://es.wikipedia.org/wiki/Eficiencia_algor%C3%ADtmica
- **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. 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.

Jessica Daniela Otero Fernandez
Cristian Andrés Gironza M
Santiago Figueroa Aguirre
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.

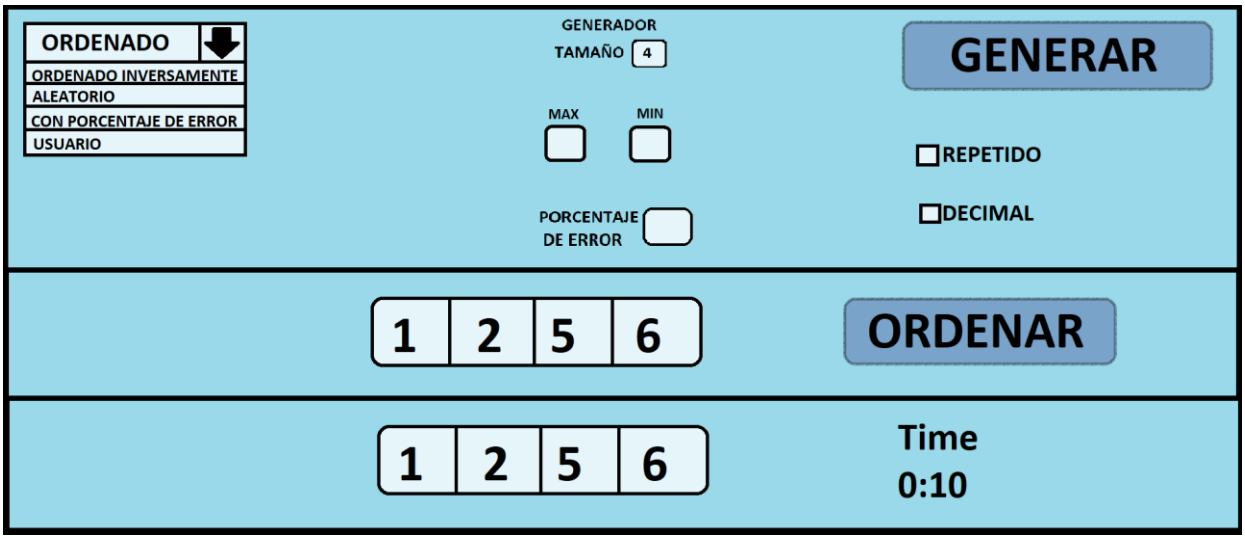
Alternativa 8: Investigar e implementar los algoritmos radix sort, bucket sort, counting sort.

Alternativa 9: Pedir al usuario que genere números ordenados.

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

Jessica Daniela Otero Fernandez
Cristian Andrés Gironza M
Santiago Figueroa Aguirre

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.

Jessica Daniela Otero Fernandez
Cristian Andrés Gironza M
Santiago Figueroa Aguirre

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: Implementar 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

```
Nombre del método, parámetros: matriz de enteros A, entero izq, entero der.
  Se crea entero pivote: A en la posición izq
  Se crea entero i: izq
  Se crea entero j: der
  Se crea entero aux.
Mientras que i sea menor que j
  Mientras que A en la posición i sea menor o igual a pivote y i sea menor que j
    "i" se incrementa de uno en uno.
  Mientras que A en la posición j sea mayor a pivote
    "j" se decrementa de uno en uno.
Si i es menor que j
    Aux: A en la posición i
    A en la posición i: A en la posición j
    A en la posición j: aux
A en la posición izq: A en la posición j
A en la posición J: pivote.
Si izq es menor que j-1
  Se hace llamado a quicksort, parámetros: A, izq,j-1
Si j+1 es menor que der
  Se hace llamado a quicksort, parámetros: A,j+1,der
```

```
1. nombre del método, ingresan como parámetros arreglo de enteros arr y
   un entero exp.
  crea mx: arreglo en la posición 0.
  Para i=1 hasta n
    Si arreglo en la posición i es mayor a mx
    Mx = arreglo en la posición i
    Retorna mx
```

Jessica Daniela Otero Fernandez
Cristian Andrés Gironza M
Santiago Figueroa Aguirre

2. nombre del método, ingresan como parámetros, un arreglo y 2 enteros.

Se crea el arreglo de enteros output de tamaño n
Se crea i
Se crea el arreglo de enteros count de tamaño 10.
Se llama a Arrays.fill(count,0)
Para i: 0 hasta n, incrementándose de uno en uno
  Count en la posición (arr i dividido exp) módulo 10, se incrementa de uno en uno
Para i: 1 hasta 10, incrementándose de uno en uno
  count en la posición i suma a count en la posición i-1
Para i:n-1 hasta que i: mayor o igual a 0, decrementándose de uno en uno
  Output en la posición (count en la posición((arr en la posición i dividido exp) módulo 10)-1) : arr en la posición i.
  Count en la posición((arr en la posición i dividido exp) módulo 10), decrementándose.
Para i:0 hasta n, incrementándose de uno en uno
  Arr en la posición i : output en la posición i.

3. nombre del método, parámetros arreglo entero arr y entero n

se crea el entero m: llamado al método getMax(), con parámetros, un arreglo de enteros, y un entero.
para exp: 1 hasta m dividido exp mayor a cero, exp incrementándose por 10.
  Se hace llamado al método countSort, con parámetros, un arreglo de enteros y dos enteros.

Nombre del método sort, parámetro: arreglo de enteros arr.

Se crea el entero n: arr.length
  Para i: 0 hasta n-1, i incrementándose de uno en uno.
    Se crea entero min_idx: i
   Para j:1+1 hasta n, j incrementándose de uno en uno.
      Si arr en la posición j es menor a arr en la posición min_idx
        Min_idx: j
  Se crea entero temp: arr en la posición min_idx
  Arr en la posición min_idx: arr en la posición i
  Arr en la posición i: temp

DISEÑO DE CASOS PARA PRUEBAS UNITARIAS

Entrada	Salida	Descripción	Nombre del método	Clase	Escenario
False,false,10,15,0	true	La prueba retorna true, cuando el caso es considerado	inverselyNormal()	SmartArray	InverselyCase()

Jessica Daniela Otero Fernandez
Cristian Andrés Gironza M
Santiago Figueroa Aguirre

		normal. Caso contrario, de no serlo.			
True,false,17,58,3	true	El método retorna true cuando el caso es considerado límite. Esto ocurre cuando el mínimo es diferente de 0.	InverselyLimit()	SmartArray	InverselyCase()
True, false, 17,58,1	false				
True, false, 20,15,0	true	El método retorna true cuando el caso es interesante. Esto ocurre cuando size es mayor al máximo	Inverselyinteresting()	SmartArray	InverselyCase()
False,20,30,0	true	La prueba retorna true, cuando el caso es considerado normal. Caso contrario, de no serlo.	Randomnormal()	SmartArray	Random()
True,30,40,0	true	El método retorna true cuando el caso es considerado límite. Esto ocurre cuando el mínimo es diferente de 0.	Randomlimit()	SmartArray	Random()
False,20,30,0	true	El método retorna true cuando el caso es interesante. Esto ocurre cuando size es mayor al máximo	Randominteresting()	SmartArray	Random()

ANÁLISIS DE COMPLEJIDAD TEMPORAL – ALGORITMO 1	
public static void quicksort(int A[], int izq, int der) {	Worst case.
int pivote=A[izq];	1

Jessica Daniela Otero Fernandez
Cristian Andrés Gironza M
Santiago Figueroa Aguirre

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--;	N-1
if (i<j) {	N
aux= A[i];	1
A[i]=A[j];	1
A[j]=aux; } }	1
A[izq]=A[j];	1
A[j]=pivote;	1
if(izq<j-1)	N
quicksort(A,izq,j-1);	1
if(j+1 <der)	N
quicksort(A,j+1,der); }	1
Función de tiempo del algoritmo: **Notación asintótica:**	9 + 6N O(N)
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;
}
	
	1
	N
	N
	1
	1

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);
	
	1
	1
	1
	1

Jessica Daniela Otero Fernandez
Cristian Andrés Gironza M
Santiago Figueroa Aguirre

`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);`
`}`
	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];`
	
	1
	
	n-1
	
	n-1
	n-1
	n-1
	1
	1
	1

Jessica Daniela Otero Fernandez
Cristian Andrés Gironza M
Santiago Figueroa Aguirre

arr[i] = temp; }	1
Función de tiempo del algoritmo:	1+n-1+n-1+n-1+n-1+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