UNIVERSIDAD ICESI

Facultad de Ingeniería

Laboratorio de Algoritmos y Estructuras de datos

II Periodo de 2018

Jose. G. Morales , J. Bellaiza, Alexis Bonilla

Link proyecto: https://github.com/JhonatanBellaiza/Laboratorio-1-AED.git

1.Identificación del problema.

Una empresa desarrolladora de microprocesadores desea implementar un software que sirva como prototipo para modelar un coprocesador que ordena secuencias de números.

Identificación de necesidades (Requerimientos del programa):

Nombre:	**R1. Generar aleatoriamente valores en un arreglo**
Descripción:	Este requerimiento permite generar un conjunto de valores numéricos en un arreglo con cotas señaladas por el usuario.
Entradas:	
Resultado:	Se mostrarán los personajes en la interfaz adecuadamente.

Nombre:	**R2. Interfaz Grafica**
Descripción:	Esta interfaz le permite al usuario ingresar los valores a ordenar, así como ver las demás funcionalidades del programa.
Entradas:	
Resultado:	Interfaz gráfica para la interacción con el usuario.

Nombre:	**R3. Generar números diferentes**
Descripción:	Este método permite al usuario elegir que todos los números generados sean diferentes.
Entradas:	Verificación por medio de check box.
Resultado:	Se generan los valores sin ningún número repetido.

Nombre:	**R4. Generar números diferentes**
Descripción:	Este método permite al usuario elegir que todos los números generados sean diferentes.
Entradas:	Verificación por medio de check box.
Resultado:	Se generan los valores aleatorios sin ningún número repetido.

Nombre:	**R5. Generar valores ordenados**
Descripción:	Este método permite al usuario elegir que todos los números generados estén ya ordenados.
Entradas:	Verificación por medio de check box.
Resultado:	Se generan los valores en orden.

Nombre:	**R6. Generar valores ordenados inversamente**
Descripción:	Este método permite al usuario elegir que todos los números generados estén ordenados a la inversa
Entradas:	Verificación por medio de check box.
Resultado:	Se generan los valores ordenados inversamente.

Nombre:	**R7. Generar valores con un porcentaje de ordenamiento**
Descripción:	Este método permite al usuario elegir en que porcentaje de ordenamiento quiere que se le generen los valores.
Entradas:	Porcentaje de ordenamiento
Resultado:	Se generan los valores con el porcentaje de ordenamiento ingresado por el usuario

Nombre:	**R8. Elegir el algoritmo más óptimo para ordenar los valores**
Descripción:	Este método permite generar los valores con el algoritmo más optimo, según los datos ingresados por el usuario
Entradas:	El arreglo y las especificaciones del usuario.
Resultado:	Se ordena el arreglo con el algoritmo más eficiente para el caso.

Nombre:	**R9. Elegir el algoritmo más óptimo para ordenar los valores**
Descripción:	Este método permite generar los valores con el algoritmo más optimo, según los datos ingresados por el usuario
Entradas:	El arreglo y las especificaciones del usuario.
Resultado:	Se ordena el arreglo con el algoritmo más eficiente para el caso.

Nombre:	**R10. Mostrar el tiempo que toma el ordenamiento**
Descripción:	Este método muestra en pantalla el tiempo que tarda el algoritmo en ordenar el arreglo.
Entradas:	
Resultado:	Se muestra en pantalla el tiempo que tardo el algoritmo en ordenar el arreglo.

2.Recopilación de la información necesaria:

- **Algoritmos de ordenamiento:** Este tipo de algoritmos será necesario implementarlos en el programa ya que su función es poner 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 un reordenamiento de la entrada, que satisfaga la relación de orden dado. Existen diferentes tipos de algoritmos de ordenamiento los cuales pueden ser muy útiles para ciertos casos definidos; a continuación, se listarán algunos de ellos:

  1) Ordenamiento de Burbuja(Bubblesort**):** Revisa cada elemento de la lista de izquierda a derecha, intercambiando de posición aquellos valores que están en el orden equivocado.

2) Ordenamiento de burbuja bidireccional(Cocktail sort**):** Es una mejora del algoritmo de ordenamiento por burbuja, ordena al mismo tiempo por los dos extremos del vector, de tal manera que en la primera iteración el menor y el mayor estarán en sus posiciones finales.

3)Ordenamiento por inserción(Insertion sort**):** Se usa para insertar a un arreglo ordenado un elemento, se itera sobre todas las posiciones en el arreglo comenzando por la primera posición hasta que encuentra la posición en la que debe estar posicionado.

4)Ordenamiento por casilleros(Bucket sort**):** Este es un programa java para ordenar los números usando la técnica de ordenamiento del cubo. El algoritmo asigna el número de ubicaciones de memoria igual al número máximo e inicializa todo en cero, luego cada ubicación se incrementa a medida que aparecen los números.

5)Ordenamiento por cuentas(Counting sort**):** solo se usa para elementos que sean contables, cuenta todo el número de los elementos de cada clase para después ordenarlos. Se toma un intervalo y se crea un vector con los elementos dentro del intervalo, luego, a cada elemento se le asigna el valor de cero, para luego recorrer todos los elementos y contar el número de apariciones de cada elemento. Por último, se vuelve a recorrer el vector para tener todos los elementos ordenados.

6)Ordenamiento por mezcla(Merge sort): Esta basado en la técnica divide y vencerás. Este método toma el vector y lo divide en dos, para después seguir dividiéndole forma recursiva tantas veces sea posible, luego, mezcla las sublistas en una sola lista ordenada.

7)Ordenamiento por selección(Selection sort): Busca el mínimo elemento de la lista para luego intercambiarlo por el de la primera posición, después, vuelve a buscar el mínimo elemento de la lista empezando por la segunda posición, y así sucesivamente hasta tener la lista completamente ordenada.

8) Ordenamiento rápido(Quick sort**):** Toma un elemento de la lista (Pivote) y empieza a almacenar todos los valores menores que el a un lado y los mayores al otro, el vector queda como dos sublistas y lo que se debe hacer es repetir este proceso hasta que se ordene completamente.

9)Ordenamiento Radix(Radixsort): Ordena enteros procesando cada digito de forma individual

- **Estabilidad de los algoritmos:**

  Aquellos algoritmos que se consideran como estables son los que mantienen un relativo preorden total, es decir, un algoritmo es estable solo si hay dos registros A y B con la misma clave, con A apareciendo antes que B en la lista original.

  Los algoritmos de ordenamiento inestable pueden cambiar el orden relativo de registros con claves iguales, pero los estables nunca lo hacen.

  Estables: Bubblesort, Cocktail sort, insertion sort, buscket sort, counting sort, merge sort, binary tree sort

  Inestables: Shell sort, selection sort, quicksort.

- **Tipos de datos numéricos (Java):**

Tipos primitivos: Sin métodos; no son objetos, no necesitan invocación para ser creados.

Nombre	Tipo	Espacio en memoria	Rango aproximado
byte	entero	1 byte	-128 a 127
short	entero	2 bytes	-32768 a 32767
int	entero	4 bytes	2*10^9
long	Entero	8 bytes	Muy grande
float	Decimal simple	4 bytes	Muy grande
double	Decimal doble	8 bytes	Muy grande

- **Generar números random en java:**

  En java existen dos clases principales para generar números aleatorios, estas son: Ramdom y SecureRandom., se definen los intervalos y se generan números aleatorios en ese rango.

3. Búsqueda de soluciones creativas

Alternativa 1: En lo que respecta a la interfaz gráfica con la cual el usuario va a interactuar, esta puede ser realizada en JavaFX, la cual es una biblioteca de java para el manejo de las interfaces graficas actualizada y con mayor versatilidad que las versiones anteriores.

Alternativa 2: Para la interfaz gráfica, también podemos utilizar sin ningún problema el swing de java, ya que somo un poco más expertos en este campo, y podemos implementar de una mejor manera la interfaz con la que el usuario va a interactuar.

Alternativa 3: Para el caso de que el usuario ingrese los valores que el desea ordenar, tomando en cuenta de que probablemente será una lista pequeña, podemos implementar el método de ordenamiento countig sort, el cual funciona de forma más eficiente en listas pequeñas a ordenar.

Alternativa 4: Para el caso en el que el usuario elija la opción de generar aleatoriamente los valores, debido a que la cantidad de valores puede ser muy grande y se le ofrece al usuario varias opciones de cómo quiere que se le ordenen los valores ingresados, es necesario utilizar diferentes algoritmos según el caso, por lo tanto, utilizaríamos los check box para luego, según los criterios escogidos, elegir el algoritmo que ordene de forma más eficiente los valores.

Alternativa 5: Podemos llegar a combinar diferentes tipos de ordenamiento en un solo algoritmo para que, según los criterios ingresados, el ordenamiento de los valores sea más eficiente.

Alternativa 6: Si resulta que, en la lista de valores a ordenar, el primer elemento del arreglo es mayor que el resto, podemos utilizar el countig sort, ya que, para este algoritmo, el mejor caso es que el valor mayor esté ubicado en la primera posición.

Alternativa 7: En la etapa de desarrollo e implementación de código podemos implementar en el mundo 4 clases, una principal para tener la conexión con la interfaz, y las otras tres para implementar cada uno de los algoritmos que escogeremos.

Alternativa 8: Si todos los valores del arreglo a ordenar son enteros, podríamos utilizar el método radixSort ya que para estos casos la complejidad temporal es mucho más eficiente.

4. Transición de las Ideas a los Diseños Preliminares

- **CountingSort**

CountingSort(A[ ])			
Pasos	Costo	Cantidad (temporal)	
n = A.length	C1	1	
max = A[0]	C2	1	
for i = 1 < n	C3	n	
If(A[i] > max)	C4	Mejor (n-1)	Peor (n-1)
max = A[i]	C5	0	n-1
c = new int[max +1]	C6	1	1
for i = 0 < n	C7	n+1	n+1
c[A[i]]++	C8	n	n
for i=1 <= k	C9	k+1	k+1
c[i] = c[i] + c[i-1];	C10	k	k
b = new int[n]	C11	1	1
for i = (n-1) >= 0	C12	n+1	n+1
c[A[i]]--	C13	n	n
b[c[A[i]]]= A[i];	C14	n	n
return b	C15	1	1

***Complejidad temporal del algoritmo CountingSort para ordenar [(3)]**

Se tiene que:

n: El tamaño del arreglo de entrada, es decir, A.length

k: El entero más grande que se encuentra en la lista, es decir la k = max

Este algoritmo ordena la lista de números enteros de menor a mayor.

Mejor caso: El mejor caso para este algoritmo es cuando el primer entero de la lista es el mayor número de toda la lista, es decir, A[0]>A[1], A[0]>A[2]… A[0]>A[n-1].

Peor caso: El peor caso para este algoritmo es cuando la lista se encuentra ordenada y además no hay números repetidos, es decir, A[0]<A[1], A[1]<A[2], A[2]<A[3]… A[n-2]<A[n-1].

Calculo de complejidad temporal:

Mejor caso:

T(n,k)=

1(C1) +1(C2)+ n(C3)+ (n-1)(C4)+ 1(C6)+ (n+1)(C7)+ n(C8)+ (k+1)(C9)+ k(C10)+ 1(C11)+ (n+1)(C12)+ n(C13)+ n(C14)+ 1(C15)

= n(C3+C4+C7+C8+C12+C13+C14)+ k(C10+C9)+(C1+C2- C4+ C6+ C7+ C9+ C11+ C12+ C15)

Si

A = (C3+C4+C7+C8+C12+C13+C14)

B = (C10+C9)

C = (C1+C2-C4+C6+C7+C9+C11+C12+C15)

Entonces

T(n,k) = 7n+2k+7= n(A)+k(B)+(C) $\in O(n+k)$

Peor caso:

T(n+k)=

1(C1) +1(C2)+ n(C3)+ (n-1)(C4)+ (n-1)(C5)+ 1(C6)+ (n+1)(C7)+ n(C8)+ (k+1)(C9)+ k(C10)+ 1(C11)+ (n+1)(C12)+ n(C13)+ n(C14)+ 1(C15)

=n (C3+C4+C5+C7+C8+C12+C13+C14)+ k(C10+C9)+ (C1+ C2- C4- C5+ C6+ C7+ C9+ C11+ C12+ C15)

Si

A = (C3+C4+C5+C7+C8+C12+C13+C14)

B = (C10+C9)

C = (C1+C2-C4-C5+C6+C7+C9+C11+C12+C15)

Entonces

T(n,k) = 8n+2k+6 = n(A)+k(B)+(C) $\in O(n+k)$

Tipo	Nombre	Costo	Cantidad
Entrada	A	C1	n
Auxiliar	n	C1	1
	máx.	C1	1
	i	C1	4
	c	C1	k
Salida	b	C1	n

***Complejidad espacial del algoritmo CountingSort para ordenar**

El costo se mantiene en C1, porque todas las variables utilizadas son de tipo entero, por lo cual tendrán un costo igual.

Calculo de complejidad espacial:

(n+1+1+4+k+n) (C1) = (2n+k+6)(C1) $\in O(n+k)$

- **Bucketsort**

BucketSort(A[ ])			
Pasos	Costo	Cantidad (temporal)	
n = A.length	C1	1	
max = A[0]	C2	1	
for i = 1 < n	C3	n	
If(A[i] > max)	C4	Mejor (n-1)	Peor (n-1)
max = A[i]	C5	0	n-1
bucket = new int[max+1]	C6	1	
aux = new[n]	C7	1	
for i = 0 < n	C8	n+1	
bucket[A[i]]++	C9	n	
op = 0;	C10	1	
for i = 0 < k	C11	k+1	
for j=0 to bucket[i]	C12	k+n	
aux[outPoss++] = i	C13	n	
return aux	C14	1	

***Complejidad temporal del algoritmo BucketSort para ordenar [(4)]**

En este algoritmo se puede encontrar un ciclo adentro de otro ciclo, la razón por la cual da k+n es que el for se ejecuta por lo menos 1 vez por cada ciclo, y entra en tal caso que el bucket[i] > 0, y como en un anterior ciclo se observa, este solo tiene un numero diferente a cero en las posiciones las cuales son equivalentes a los números del arreglo de entrada, por lo tanto, la suma del arreglo bucket, de bucket[0] hasta bucket[k] es igual a n, por esto la parte interna del for se ejecuta n veces y el for se ejecuta n veces más las k vees anterior mencionadas.

Se tiene que:

n: El tamaño del arreglo de entrada, es decir, A.length

k: Es el tamaño el arreglo bucket, es decir, bucket.length que es igual a max+1

Este algoritmo ordena la lista de números enteros de menor a mayor.

Mejor caso: El mejor caso para este algoritmo es cuando el primer entero de la lista es el mayor número de toda la lista, es decir, A[0]>A[1], A[0]>A[2]… A[0]>A[n-1].

Peor caso: El peor caso para este algoritmo es cuando la lista se encuentra ordenada inversamente y además no hay números repetidos, es decir, A[0]>A[1], A[1]>A[2], A[2]>A[3]… A[n-2]>A[n-1].

Se tiene que:

n: El tamaño del arreglo de entrada, es decir, A.length

k: El entero más grande que se encuentra en la lista, es decir la k = max

Este algoritmo ordena la lista de números enteros de menor a mayor.

Mejor caso: El mejor caso para este algoritmo es cuando el primer entero de la lista es el mayor número de toda la lista, es decir, A[0]>A[1], A[0]>A[2]… A[0]>A[n-1].

Peor caso: El peor caso para este algoritmo es cuando la lista se encuentra ordenada y además no hay números repetidos, es decir, A[0]<[1], A[1]<A[2], A[2]<A[3]… A[n-2]<A[n-1].

Calculo de complejidad temporal:

Mejor caso:

T(n,k)=

1(C1) +1(C2)+ n(C3)+ (n-1)(C4)+ 1(C6)+ 1(C7)+ (n+1)(C8)+ n(C9)+ 1(C10)+ (k+1)(C11)+ (k+n)(C12)+ n(C13)+ 1(C14)

= n(C3+C4+C8+C9+C12+C13)+ k(C11+C12)+ (C1+C2-C4+C6+C7+C8+C10+C11+C14)

Si

A = (C3+C4+C8+C9+C12+C13)

B = (C11+C12)

C = (C1+C2-C4+C6+C7+C8+C10+C11+C14)

Entonces

T(n,k) = 6n+2k+7= n(A)+k(B)+(C) $\in O(n+k)$

Peor caso:

T(n,k)=

1(C1)+1(C2)+n(C3)+(n-1)(C4)+(n-1)(C5)+1(C6)+1(C7)+(n+1)(C8)+n(C9)+1(C10)+ (k+1)(C11)+ (k+n)(C12)+ n(C13)+ 1(C14)

= n(C3+C4+ C5+C8+C9+C12+C13)+ k(C11+C12)+ (C1+C2-C4- C5+ C6+ C7+ C8+ C10+ C11+ C14)

Si

A = (C3+C4+C5+C8+C9+C12+C13)

B = (C11+C12)

C = (C1+C2-C4-C5+C6+C7+C8+C10+C11+C14)

Entonces

T(n,k) = 7n+2k+6= n(A)+k(B)+(C) $\in O(n+k)$

Tipo	Nombre	Costo	Cantidad
Entrada	A	C1	n
Auxiliar	n	C1	1
	max	C1	1
	i	C1	3
	j	C1	1
	bucket	C1	k
	op	C1	1
Salida	aux	C1	n

***Complejidad espacial del algoritmo BucketSort para ordenar**

El costo se mantiene en C1, porque todas las variables utilizadas son de tipo entero, por lo cual tendrán un costo igual.

Calculo de complejidad espacial:

(n+1+1+3+1+k+1+n) (C1) = (2n+k+7)(C1) $\in O(n+k)$

- **RadixSort (inestable)**

RadixSort(A[ ])			
Pasos	Costo	Cantidad (temporal)	
n = A.length	C1	1	
max = A[0]	C2	1	
for i = 1 < n	C3	n	
If(A[i] > max)	C4	Mejor (n-1)	Peor (n-1)
max = A[i]	C5	0	n-1
cantdigi= Integer.toString(max).length()	C6	1	
miles = (int) Math.pow(10, (cantdigi-1))	C7	1	
for pl = 1 <= miles (pl aumenta, tal que, pl*10 cada ciclo)	C8	k+1	
c = new int[10]	C9	k	
for i=0 to n	C10	k(n+1)	
digit = ((A[i] / pl) % 10)	C11	kn	
c[digit] ++	C12	kn	
for i=1 to c.length=10	C13	10k	
c[i] = c[i] + c [i-1]	C14	9k	
aux = new int[n]	C15	k	
for i=(n-1) >= 0	C16	k(n+1)	
digit = ((A[i] / pl) % 10)	C17	kn	
aux[c[digit]-1] = A[i]	C18	kn	
c[digit]--	C19	kn	
A = aux	C20	k	
return A	C21	1	

***Complejidad temporal del algoritmo RadixSort para ordenar [(5)]**

En este algoritmo se encuentra el número mayor, a este se mira cuantos dígitos tiene, y luego se va ordenando el arreglo por cada uno de los dígitos, de derecha a izquierda, por ejemplo, si se obtiene como mayor el número 102, se analiza en el siguiente orden, 10**2**, 1**0**2, y por ultimo **1**02, después de analizar el arreglo por el orden de sus dígitos tantas veces como dígitos tenga el número mayor del arreglo, el arreglo se encontrará ordenado, en este caso, para utilizar menos memoria, se analizó de manera inestable.

Se tiene que:

n: El tamaño del arreglo de entrada, es decir, A.length

k: Es la cantidad de dígitos que tiene el mayor número de la lista, por ejemplo, el numero 102 tiene 3 dígitos.

Este algoritmo ordena la lista de números enteros de menor a mayor.

Se sabe que el programa nos pide ordenar arreglos de números de tipo entero, entonces se sabe que las variables de tipo Integer contienen enteros de 32 bits con signo (4 bytes) comprendidos en el intervalo entre -2.147.483.648 y 2.147.483.647. Cómo este algoritmo no puede ordenar arreglos con números negativos, miro solo el limite positivo y tiene un total de 10 dígitos, por lo tanto, el valor máximo que puede tomar k es 10

Mejor caso: El mejor caso para este algoritmo es cuando el primer entero de la lista es el mayor número de toda la lista, es decir, A[0]>A[1], A[0]>A[2]… A[0]>A[n-1].

Peor caso: El peor caso para este algoritmo es cuando la lista se encuentra ordenada y además no hay números repetidos, es decir, A[0]<A[1], A[1]<A[2], A[2]<A[3]… A[n-2]<A[n-1].

Calculo de complejidad temporal:

Mejor caso:

T(n,k)=

1(C1) +1(C2)+ n(C3)+ (n-1)(C4)+ 1(C6)+ 1(C7)+ (k+1)(C8)+ k(C9)+ k(n+1)(C10)+ kn(C11)+ kn(C12)+ 10k(C13)+ 9k(C14)+ k(C15)+ k(n+1)(C16)+ kn(C17)+ kn(C18)+ kn(C19)+ k(C20)+ 1(C21)

=kn(C10+C11+C12+C16+C17+C18+C19)+ n(C3+C4)+
k(C8+C9+C10+10C13+9C14+C15+C16+C20)+ (C1+C2-C4+C6+C7+C8+C21)

Si

A = (C10+C11+C12+C16+C17+C18+C19)

B = (C3+C4)

C = (C8+C9+C10+10C13+9C14+C15+C16+C20)

D = (C1+C2-C4+C6+C7+C8+C21)

Entonces

T(n,k) = 7nk+ 2n+ 25k+ 5= nk(A)+n(B)+k(C)+D $\in O(nk)$

Peor caso:

T(n,k)=

1(C1) +1(C2)+ n(C3)+ (n-1)(C4)+ (n-1)(C5)+ 1(C6)+ 1(C7)+ (k+1)(C8)+ k(C9)+ k(n+1)(C10)+ kn(C11)+ kn(C12)+ 10k(C13)+ 9k(C14)+ k(C15)+ k(n+1)(C16)+ kn(C17)+ kn(C18)+ kn(C19)+ k(C20)+ 1(C21)

=kn(C10+C11+C12+C16+C17+C18+C19)+ n(C3+C4+C5)+
k(C8+C9+C10+10C13+9C14+C15+C16+C20)+ (C1+C2-C4-C5+C6+C7+C8+C21)

Si

A = (C10+C11+C12+C16+C17+C18+C19)

B = (C3+C4+C5)

C = (C8+C9+C10+10C13+9C14+C15+C16+C20)

D = (C1+C2-C4-C5+C6+C7+C8+C21)

Entonces

T(n,k) = 7nk+ 3n+ 25k+ 4= nk(A)+n(B)+k(C)+D $\in O(nk)$

Tipo	Nombre	Costo	Cantidad
Entrada	A	C1	n
Auxiliar	n	C1	1
	max	C1	1
	i	C1	4
	cantdigi	C1	1
	miles	C1	1
	pl	C1	1
	c	C1	10
	digit	C1	2
	aux	C1	n
Salida	A	C1	0(ya registrada)

***Complejidad espacial del algoritmo RadixSort para ordenar**

El costo se mantiene en C1, porque todas las variables utilizadas son de tipo entero, por lo cual tendrán un costo igual.

Calculo de complejidad espacial:

(n+1+1+4+1+1+1+10+2+n) (C1) = (2n+21) (C1) $\in O(n)$

- **MergeSort (inestable)**

  Este método cuenta con un método auxiliar el cual es la mezcla del algoritmo mergesort, por lo tanto se analizaran por separado, primero se analizara la mezcla (este será analizado para el momento del mezclado total de los datos, es decir que (izq+der+1)=n):

Merge Void(A[ ] , izq, m, der)			
Pasos	Costo	Cantidad (temporal)	
i = 0	C1	1	
j = 0	C2	1	
k = 0	C3	1	
B = new int(A.lenth())	C4	1	
for i=izq <= der	C5	n+1	
B[i] = A[i]	C6	n	
i = izq	C7	1	
j = m+1	C8	1	
k = izq	C9	1	
Mientras (i<=m && j<=der)	C10	Mejor (n/2) +1	Peor (n/2) + 1
if (B[i] <= B[j])	C11	n/2	n/2
A[k] = B[i]	C12	n/2	0
i++	C13	n/2	0
k++	C14	n/2	0
else			
A[k]=B[j]	C15	0	n/2

j++	C16	0	n/2
k++	C17	0	n/2
Mientras (i<=m)	C18	1	(n/2) +1
A[k] = B[i]	C19	0	n/2
i++	C20	0	n/2
k++	C21	0	n/2

***Complejidad temporal del algoritmo Merge para mezclar (6)**

Hasta este momento, el algoritmo de mezclar, fue analizado desde el momento en el cual de desea mezclar el total de números, ya que este tendrá el comportamiento temporal mayor, ya a este momento se encuentran dos mitades ordenas para mezclar

Se tiene que:

n: El tamaño del arreglo de entrada, es decir, A.length

Este algoritmo ordena la lista de números enteros de menor a mayor.

Este programa, ya que es comparativo, nos permite ordenar enteros negativos también.

Mejor caso: El mejor caso para este algoritmo es cuando la lista se encuentra ordenada, es decir, A[0]<=A[1], A[1]<=A[2], A[2]<=A[3]… A[n-2]<=A[n-1].

Peor caso: El peor caso para este algoritmo es cuando la lista se encuentra ordenada inversamente y Además no hay números repetidos, es decir, A[0]>A[1], A[1]>A[2], A[2]>A[3]… A[n-2]>A[n-1].

Calculo de complejidad temporal:

Mejor caso:

T(n)=

1(C1) +1(C2)+ n(C3)+ 1(C4)+ (n+1)(C5)+ n(C6)+ 1(C7)+ 1(C8)+ 1(C9)+ ((n/2)+1)(C10)+ (n/2)(C11)+ (n/2)(C12)+ (n/2)(C13)+ (n/2)(C14)+ 1(C18)

=n(C5+C6+(C10/2) +(C11/2) +(C12/2) +(C13/2) +(C14/2))+
(C1+C2+C4+C5+C7+C8+C9+C10+C18)

Si

A = (C5+C6+(C10/2) +(C11/2) +(C12/2) +(C13/2) +(C14/2))

B = (C1+C2+C4+C5+C7+C8+C9+C10+C18)

Entonces

T(n) = (9/2)n+ 9= n(A)+ B $\in O(n)$

Peor caso:

T(n)=

1(C1) +1(C2)+ n(C3)+ 1(C4)+ (n+1)(C5)+ n(C6)+ 1(C7)+ 1(C8)+ 1(C9)+ ((n/2)+1)(C10)+ (n/2)(C11)+ (n/2)(C15)+ (n/2)(C16)+ (n/2)(C17)+ ((n/2)+1)(C18)+ (n/2)(C19)+ (n/2)(C20)+ (n/2)(C21)

=n(C5+C6+(C15/2) +(C16/2) +(C17/2) +(C102) +(C18/2) +(C19/2) +(C20/2) +(C21/2))+ (C1+C2+C4+C5+C7+C8+C9+C10+C18)

Si

A = C5+C6+(C15/2) +(C16/2) +(C17/2) +(C102) +(C18/2) +(C19/2) +(C20/2) +(C21/2)

B = (C1+C2+C4+C5+C7+C8+C9+C10+C18)

Entonces

T(n) = (6)n+ 9= n(A)+ B $\in O(n)$

Por lo tanto, se dice que la complejidad del método auxiliar para mezclar es de O(n).

Ahora, se analiza el método recursivo mergesort, el cual utiliza el método auxiliar merge, y segú lo calculado se le pondrá en la casilla de cantidad como n.

MergeSort Void(A[ ] , izq, der)		
Pasos	Costo	Cantidad (temporal)
If (izq<der)	C1	
int m=(izq+der)/2	C2	1
mergesort(A,izq, m);	C3	T(n/2)
mergesort(A,m+1 , der);	C4	T(n/2)
merge(A,izq, m, der);	C5	n

***Complejidad temporal del algoritmo MergeSort para ordenar**

Analizando este método tenemos que:

$$T(n) = 2\,T(n/2) + (n)$$

Se lee de la forma que, el tiempo que se demora en dividir el arreglo, tanto por izquierda como por derecha, por eso 2T(n), más el tiempo que se demora mezclándolos, el cual ya se calculó y es n. Esto va hacia un tope, el cual es cuando no se puede dividir más el arreglo, donde Co es una contante.

Entonces, por razones de facilidad en los cálculos usamos n como potencias de 2

$$T(2^k) = 2\,T(2^{k-1}) + (2^k)$$

Siguiendo con la sucesión

$$T(2^k) = 2\left[2\,T(2^{k-2}) + (2^{k-1})\right] + (2^k) = 2^2\,T(2^{k-2}) + 2(2^k)$$

$$\mathrm{T}\left(2^k\right) = 2^2\left[2\,\mathrm{T}\left(2^{k-3}\right) + \left(2^{k-2}\right)\right] + 2(2^k) = 2^3\,\mathrm{T}(2^{k-3}) + 3(2^k)$$

Y así sucesivamente hasta llegar a una formula general la cual es:

$$\mathrm{T}\left(2^j\right) = 2^j\,\mathrm{T}(2^{k-j}) + \mathrm{j}(2^k)$$

Entonces esto va a terminar cuando j = k

$$\mathrm{T}\left(2^k\right) = 2^k\,\mathrm{T}(2^0) + \mathbf{k}(\mathbf{2^k})$$

La parte en negrita muestra cual es el que tiene el mayor crecimiento, pero como lo necesitamos en términos de n, y sabemos que n=2^k, entonces

$$\log_2 n = k$$

Esto lo remplazamos:

$$T(n) = \mathrm{n}\log_2 n + \mathrm{C}_0\,\mathrm{n}$$

Entonces se dice que el algoritmo tiene complejidad temporal: $O(n \log n)$

Tipo	Nombre	Costo	Cantidad
Entrada	A	C1	n
	izq	C1	1
	m	C1	1
	der	C1	1
Auxiliar	i	C1	1
	j	C1	1
	k	C1	1
	B	C1	n

***Complejidad espacial del algoritmo MergeSort más el Merger**

El costo se mantiene en C1, porque todas las variables utilizadas son de tipo entero, por lo cual tendrán un costo igual.

Calculo de complejidad espacial:

(n+1+1+1+1+1+1+n) (C1) = (2n+6) (C1) ∈ $O(\mathrm{n})$

- **QuickSort (inestable)**

  Similar al mergesort este método también utiliza la recursividad y el método divide y vencerás para ordenar las listas de números

QuickSort(A[ ] , izq, m, der)			
Pasos	Costo	Cantidad (temporal)	
i = izq	C1	1	
j = der	C2	1	
aux = 0	C3	1	
pivot = A[izq]	C4	1	
Mientras (i<j) (que no se choquen)	C5	Mejor 2	Peor 2
Mientras (A[i]<=pivote && i<j)	C6	(n/2) +1	2
i++	C7	n/2	1
Mientras (A[j]>pivote)	C8	(n/2) +1	n
j++	C9	n/2	n-1
if (i<j)	C10	1	1
aux= A[i];	C11	0	0
A[i]=A[j];	C12	0	0
A[j]=aux;	C13	0	0
A[izq]=A[j]	C14	1	1
A[j]=pivote	C15	1	1
if(izq<j-1)	C16	1	1
quicksort(A, izq,j-1);	C17	T(n/2)	0
if(j+1 <der)	C18	1	1
quicksort(A,j+1,der);	C19	T(n/2)	T(n-1)

***Complejidad temporal del algoritmo QuickSort para ordenar [(6)]**

Se tiene que:

n: El tamaño del arreglo de entrada, es decir, A.length

Este algoritmo ordena la lista de números enteros de menor a mayor.

Este método, al igual que el mergesort, nos permite arreglar arreglos de números reales.

Mejor caso: El mejor caso es cuando el pivote, termina dividiendo la lista en 2 mitades, por ejemplo, tenemos la lista (**5**,1,2,3,4,6,7,8,9), después de dividir menores a la izquierda y mayores a la derecha, y colocando el pivote en su lugar quedaría (1,2,3,4,**5**,6,7,8,9), dividiendo la lista en 2 mitades, es decir (1,2,3,4) y (6,7,8,9). Para que sea el mejor caso, lo anteriormente dicho se debe cumplir en todas las siguientes separaciones de la lista.

Peor caso: El peor caso es cuando el pivote, termina en un extremo, haciendo analizar una lista de tamaño n-1, ejemplo, tenemos la lista (**1**,2,3,4,5,6), después de dividir menores a la izquierda y mayores a la derecha, y colocando el pivote en su lugar quedaría (**1**,2,3,4,5,6), haciendo que el próximo análisis sea de la lista (2,3,4,5,6). (Cuando la lista está ordenada)

Calculo de complejidad temporal:

En estos cálculos omitiremos los costos, y nos concentraremos en las cantidades.

Analizando el mejor caso

$$\mathrm{T}(n) = \ 2\ \mathrm{T}\left(\frac{\mathrm{n}}{2}\right) + (2\mathrm{n}) + 13$$

Entonces, por razones de facilidad en los cálculos usamos n como potencias de 2

$$\mathrm{T}\left(2^k\right) = \ 2\ \mathrm{T}\left(2^{k-1}\right) + \left(2(2^k)\right) + 13$$

Siguiendo con la sucesión

$$\mathrm{T}\left(2^k\right) = \ 2\left[2\ \mathrm{T}\left(2^{k-2}\right) + \left(2(2^{k-1})\right) + 13\ \right] + \left(2(2^k)\right) + 13 \ = \ 2^2\ \mathrm{T}\left(2^{k-2}\right) + 4\left(2^k\right) + 2(13)$$

$$\mathrm{T}\left(2^k\right) = 2^2\left[2\ \mathrm{T}\left(2^{k-3}\right) + 2\left(2^{k-2}\right) + 13\right]\ + 4\left(2^k\right) + 2(13) = \ 2^3\ \mathrm{T}\left(2^{k-3}\right) + 6\left(2^k\right) + 3(13)$$

Y así sucesivamente hasta llegar a una formula general la cual es:

$$\mathrm{T}\left(2^j\right) = 2^j\ \mathrm{T}\left(2^{k-j}\right) + j2\left(2^k\right) + \mathrm{j}(13)$$

Esta se dividirá, k veces, talque que (techo) $\frac{n}{2^k} = 1$, entonces $T\ (1)\ =\ C_0$ (tiempo constante)

Entonces esto va a terminar cuando j = k

$$\mathrm{T}\left(2^k\right) = 2^k\ \mathrm{T}(2^0) + \boldsymbol{2k}\left(2^k\right) + \mathrm{K}13$$

La parte en negrita muestra cual es el que tiene el mayor crecimiento, pero como lo necesitamos en términos de n, y sabemos que n=2^k, entonces

$$\log_2 n = k$$

Esto lo remplazamos:

$$T(n) = \ 2\,\mathrm{n}\log_2 n + \ \mathrm{C}_0\ \mathrm{n} + \log_2 n\ \ 13$$

Entonces se dice que el algoritmo tiene complejidad temporal: $O(n \log n)$

Analizando el peor caso

$$\mathrm{T}(n) = \ \mathrm{T}(n-1) + (2\mathrm{n}) + 13$$

Entonces, por razones de facilidad en los cálculos usamos n

$$\mathrm{T}(n) = \ \ \mathrm{T}(\mathrm{n}-1) + (2(\mathrm{n})) + 13$$

Siguiendo con la sucesión

$$\mathrm{T}(n) = \ \ [\ \mathrm{T}(n-2) + (2(\mathrm{n}-1)) + 13\ ] + (2(\mathrm{n})) + 13 \ = \ \ \mathrm{T}(\mathrm{n}-2) + 4(\mathrm{n}) + 2(13) - 2$$

$$\mathrm{T}(n) = \ [\mathrm{T}(n-3) + 2(\mathrm{n}-2) + 13]\ + 4(\mathrm{n}) + 2(13) - 2 = \ \ \mathrm{T}(\mathrm{n}-3) + 6(\mathrm{n}) + 3(13) - 6$$

Y así sucesivamente hasta llegar a una formula general la cual es:

$$\mathrm{T}(j) = \ \mathrm{T}(j-(j-1)) + j2(j) + \mathrm{j}(13) - (j(j+1)$$

Remplazando j por n, seria:

$$\mathrm{T}(n) = \mathrm{T}(1) + 2\mathrm{n}^2 + n13 - n^2 - n$$

Si T(1) = C_0 (constante), entonces:

$$\mathrm{T}(n) = \mathbf{n^2} + 12\mathrm{n} + \mathrm{C}_0$$

Entonces se dice que el algoritmo tiene complejidad temporal: $O(n^2)$

Tipo	Nombre	Costo	Cantidad
Entrada	A	C1	n
	izq	C1	1
	m	C1	1
	der	C1	1
Auxiliar	i	C1	1
	j	C1	1
	aux	C1	1
	pivot	C1	1

***Complejidad espacial del algoritmo QuickSort**

El costo se mantiene en C1, porque todas las variables utilizadas son de tipo entero, por lo cual tendrán un costo igual.

Calculo de complejidad espacial:

(n+1+1+1+1+1+1+1) (C1) = (n+7) (C1) ∈ $O(\mathrm{n})$

5. Evaluación y Selección de la Mejor Solución

Criterios

Deben definirse los criterios que permitirán evaluar las alternativas de solución y con base en este resultado elegir la solución que mejor satisface las necesidades del problema planteado. Los criterios que escogimos en este caso son los que enumeramos a continuación. Al lado de cada uno se ha establecido un valor numérico con el objetivo de establecer un peso que indique cuáles de los valores posibles de cada criterio tienen más peso (i.e., son más deseables).

- Criterio A. Rapidez de la solución. La alternativa entrega una solución:

- [1] O ($\log n$)

- [2] O(n)

- [3] O(n+k)

- [4] O ($n \log n$)

- [6] O(nk)

- [7] O(n^2) o peores

- Criterio B. Eficiencia en memoria. Se recomienda utilizar algoritmos con bajo uso en la memoria. La eficiencia puede ser:

- [1] O ($\log n$)

- [2] O(n)

- [3] O(n+k)

- [4] O ($n \log n$)

- [6] O(nk)

- [7] O(n^2) o peores

- Criterio C. Variedad: Se necesitan soluciones las cuales puedan solucionar todo tipo de entrasas

- [1] Todos los reales
- [2] Solo enteros
- [3] Ninguno

Algoritmo	C1	C2	C3
CountingSort	[3]	[3]	[2]
BucketSort	[3]	[3]	[2]
RadixSort	[6]	[2]	[2]
MergeSort	[4]	[2]	[1]
QuickSort	[4]	[2]	[1]

Primero haremos la suma de los puntos obtenidos por cada método, y el que tenga menos será el mejor para ser implementado

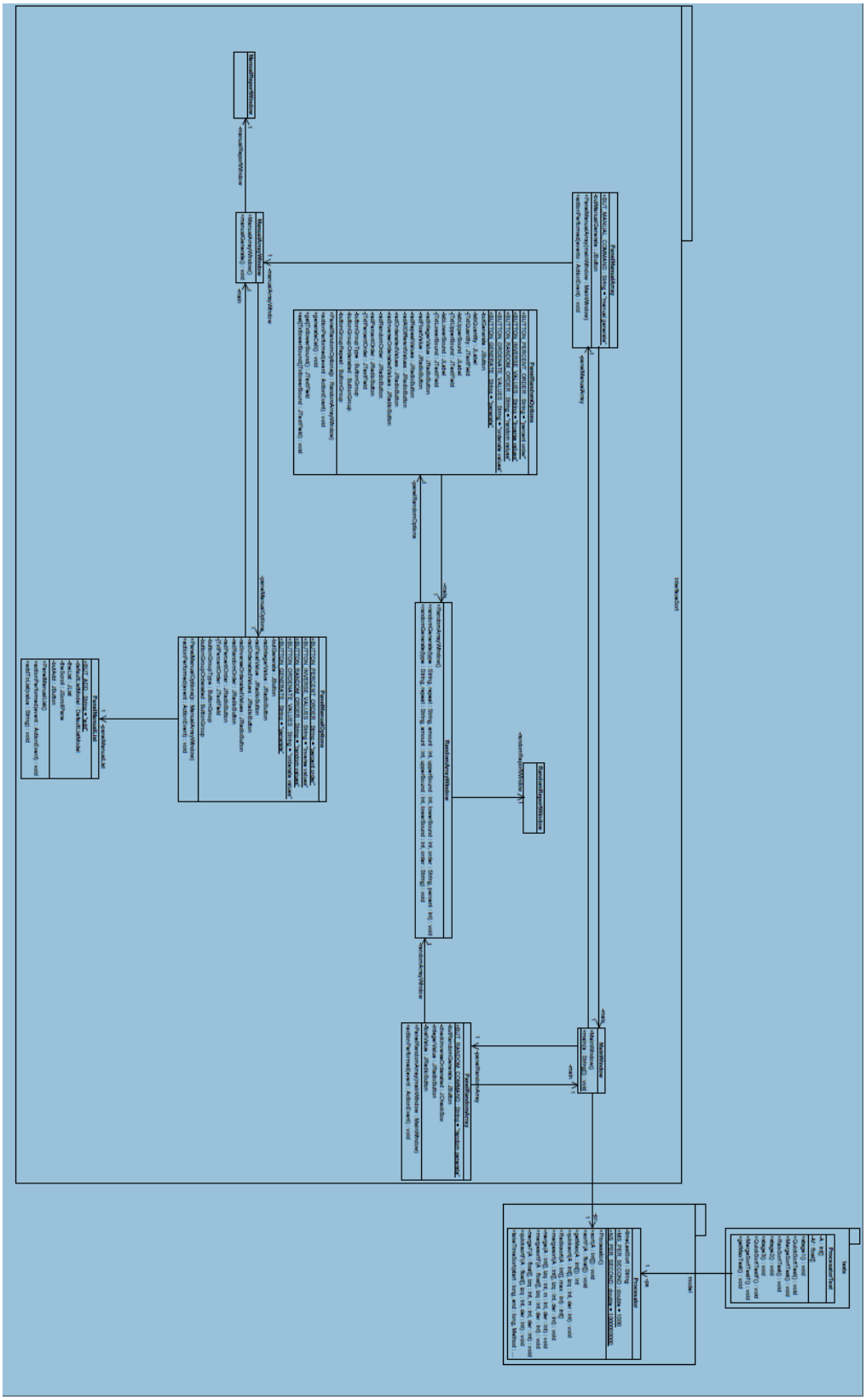
Algoritmo	Puntos
CountingSort	8
BucketSort	8
RadixSort	10
MergeSort	7
QuickSort	7

Como se observa los más convenientes son los algoritmos MergeSort y QuickSort.

Después de analizar los métodos, se considera utilizar como tercer método el RadixSort, que, aunque la tabla no lo favorece, más que todo por el puntaje en tiempo, pero esto depende, ya que el valor máximo que puede tomar k es igual a 10 en el RadixSort, en cambio, tanto

en el CountingSort como en el BucketSort el valor máximo que puede tomar k es de 2147483647, como se ve la gran diferencia, este proceso puede tardar mucho. Además el RadixSort es mucho más eficiente entre más grande sea n, y entonces por la ventaja de memoria que tiene ante el CountingSort y el BucketSort, se puede aprovechar para los tamaños de lista más grandes.

6. Preparación de Informes y Especificaciones(Diagrama de clases): (En el proyecto está el diagrama)



Bibliografía

[1] Wikipedia (2018). *Algoritmo de ordenamiento.* Recuperado de: https://es.wikipedia.org/wiki/Algoritmo_de_ordenamiento

[2] apr. (s.f). *Tipos de datos Java.* Recuperado de: https://www.aprenderaprogramar.com/index.php?option=com_content&view=article&id=419:tipos-de-datos-java-tipos-primitivos-int-boolean-y-objeto-string-array-o-arreglo-variables-cu00621b&catid=68&Itemid=188

[3] YoAndroide. (2017). *COUNTING SORT - IMPLEMENTACION EN JAVA, ALGORITMO DE ORDENAMIENTO.* Recuperado de: https://www.youtube.com/watch?v=eSv57sBfmXE&t=66s

[4] Sandfoundry. (s.f). *Java Program to Implement Bucket Sort.* Recuperado de: https://www.sanfoundry.com/java-program-implement-bucket-sort/

[5] Geeksforgeeks. (s.f). *Radix Sor*t . Recuperado de: https://www.geeksforgeeks.org/radix-sort/

[6] Java2novice. (s.f). *Program: Implement merge sort in java.* Recuperado de: http://www.java2novice.com/java-sorting-algorithms/merge-sort/