INFORME DE INGENIERÍA

MARIA CAMILA LENIS RESTREPO

JUAN SEBASTIAN PALMA GARCÍA

JAVIER ANDRÉS TORRES REYES

ALGORITMOS Y ESTRUCTURAS DE DATOS

2018-2

INFORME DE INGENIERÍA

Paso 1: Identificación del problema

Definición del problema

Encontrar los tres algoritmos de ordenamiento más eficientes para la implementación de un programa que tiene como base ordenar números enteros o reales.

Justificación

En busca de un programa más eficiente las empresas naturalmente usan coprocesadores cuya función es descargar trabajo del procesador principal ya que poseen una tarea especializada. Los coprocesadores pueden realizar operaciones nativas como lo es la función de ordenamiento. Por lo tanto, después de estudiar los costos de implementación de dicho algoritmo se ha decidido encontrar tres algoritmos de ordenamiento más eficientes dependiendo del caso de ordenamiento dado (real o entero, y el número de elementos a ordenar).

Requerimientos funcionales

R1	Ordenar datos ingresados
Descripción	Ordena datos (reales o enteros) ingresados por el usuario, y muestra el resultado.
Entradas	Datos por ordenar.
Salidas	Datos ordenados.

R2	Generar valores aleatorios ya ordenados
Descripción	Generar el número de valores que el usuario quiera y que estos estén ordenados
Entradas	Número de valores a generar
Salidas	Lista de valores ordenados

R3	Generar valores aleatorios de acuerdo a un % de desorden.
Descripción	Generar valores aleatorios con base en el tamaño de la secuencia y el % de desorden se obtiene un número k de cuantas posiciones deben estar desordenadas.
Entradas	Tamaño de la secuencia y % de desorden
Salidas	Lista de valores con k posiciones desordenadas

R4	Generar valores intercambiados
Descripción	Generar valores aleatorios ordenados y con base en el tamaño de la secuencia y el % de desorden se generan k/2 pares de posiciones diferentes y se intercambian los valores entre cada par de ellas.
Entradas	Tamaño de la secuencia y % de desorden
Salidas	Lista de valores con k/2 posiciones intercambiadas.

R5	Mostrar el tiempo de ordenamiento
Descripción	Mostrar el tiempo en que se ordenan los valores.
Entradas	Datos por ordenar
Salidas	Tiempo de ordenamiento.

R6	Restringir o permitir la ejecución de los algoritmos de ordenamiento.
Descripción	Dependiendo del número y tipo de datos a ordenar se debe restringir o permitir la ejecución de alguno de los tres algoritmos de manera que resulte ser el más eficiente.
Entradas	Datos por ordenar
Salidas	Opciones de algoritmo que pueda ser usado para ordenar esa secuencia.

Paso 2: Recopilación de la información

Algoritmo

Es una secuencia de pasos bien definidos que buscan resolver un problema computacional.

Eficiencia

Medida del uso de los recursos computacionales requeridos por la ejecución de un algoritmo en función del tamaño de las entradas. (Departamento de Ciencias de Computación e I.A. de la Universidad de Granada)

Complejidad temporal

Función que describe el comportamiento (en tiempo) de un algoritmo conforme se incrementa el tamaño de la entrada.

Complejidad espacial

Uso de recursos de espacio que requiere un algoritmo en cuestión.

Notación asintótica

Son aquellas notaciones utilizadas para describir el tiempo de ejecución asintótico de un algoritmo. Conocemos la notación O para el peor caso, Ω para el mejor caso y Θ para el caso promedio.

Según Kenneth H. Rosen en su libro *Matemática Discreta y sus aplicaciones*, los algoritmos más eficientes ordenados de manera descendente son aquellos que:

Complejidad	Terminología
O(l)	Complejidad constante
O(log n)	Complejidad logarítmica
O(n)	Complejidad lineal
O(n log n)	Complejidad n log n
O(n^k)	Complejidad polinómica
O(k^n)	Complejidad exponencial
O(n!)	Complejidad factorial

Por lo tanto, debemos tener como criterio de selección la complejidad temporal del algoritmo con base en esa tabla.

Un análisis ya existente:

Encontramos en un blog de *pereiratechtalks.com* perteneciente a Sergio Andrés Flórez un análisis hecho para encontrar el algoritmo de ordenamiento más eficiente:

Sergio toma siete algoritmos conocidos: Burbuja, inserción, selección, Heapsort (ordenamiento por montículos), conteo, merge sort y Quicksort, los implementa y los ejecuta en los maquinas con diferentes especificaciones para encontrar cuál es el algoritmo más eficiente para ordenar hasta 1.000.000.000 datos.

Después de un análisis de las gráficas de los siete algoritmos, llegó a la conclusión de que en definitiva los perdedores eran burbuja, inserción y selección. Por lo tanto, se decide analizar más a de cerca los cuatro restantes, y se llega a la conclusión de que el algoritmo más eficiente fue el de conteo con una complejidad de O(n+k).

Paso 3: Búsqueda de soluciones creativas

Decidimos hacer una lluvia de ideas de posibles soluciones:

- Crear un árbol binario e ir insertado los valores ingresados y luego imprimirlos de acuerdo al recorrido inorden.
- Ordenamiento burbuja, selección o inserción que hemos implementado antes.
- Bogosort: consiste en tomar el conjunto de valores ingresados y reorganizarlos de manera aleatoria en el conjunto, hasta que logre organizarlos de manera ascendente o descendente.
- Ordenamiento por panqueques: Ordena un arreglo de números enteros mediante rotaciones del arreglo en entre el valor más alto y el más bajo; este va reduciendo el tamaño del arreglo rotado mientras que se ordenan los valores en orden ascendente.
- Ordenamiento Radix: Es un método de ordenamiento que consiste en organizar valores de un arreglo en base a ciertas claves o categorías que el algoritmo aplica para separar los valores mediante la comparación de números en sus posiciones de cifra significativas.
- Listar algoritmos que no hemos implementados pero que son conocidos por su eficiencia: ordenamiento por mezcla, montículos, conteo y ordenamiento rápido.
- Mezclar algoritmos de ordenamiento que ya hemos usado (por ejemplo, un algoritmo que junte inserción y burbuja)

Paso 4: Transición de ideas a los diseños preliminares

A primera vista, la opción de crear un árbol y luego imprimir su recorrido inorden queda descartada debido a la complejidad de su implementación.

Los algoritmos inestables como el bogosort y el ordenamiento por panqueques también quedan descartadas porque no garantiza una respuesta segura y no tienen la capacidad de organizar arreglos grandes de manera eficiente.

Investigando más a fondo el algoritmo Radix queda descartado porque este algoritmo depende mucho de la eficiencia de las instrucciones es un interior y tiene dificultades con diferentes tipos de datos (enteros y decimales).

Por otro lado, ya hemos implementado los algoritmos de ordenamiento por inserción, selección y burbuja. Hemos concluido que el ordenamiento burbuja es el menos eficiente de ellos tres debido al número de comparaciones y recorridos que hace al arreglo. De esta manera queda descartado. Por este mismo motivo, se descarta el mezclar algoritmos que ya hemos usado.

Por lo tanto, las ideas más factibles son: algoritmo de selección, inserción, por montículos, conteo, mezcla, quick sort y rapid sort (una versión más sencilla del algoritmo conteo. Fuente: http://www.algorithmist.com/index.php/Counting_sort).

Ordenamiento	Explicación	Observaciones	Complejidad conocida
Selección	Se ordena un arreglo de valores mediante la búsqueda del valor más bajo dentro del arreglo, este valor se reemplaza con el primero; luego se busca el siguiente valor más bajo y se reemplaza con el segundo y se repite hasta que se ordene todo el arreglo.	Es sencillo de implementar, pero poco eficiente con una lista grande de valores	O(n^2)
Inserción	La principal característica de este algoritmo es que intercambia valores para "abrirle" espacio al valor que quedará en su posición correcta. Es sencillo de implementar, no necesita almacenamiento temporal adicional.	En sencillo de implementar, pero poco eficiente con una lista grande de valores.	O(n^2)
Montículos	Se ordena una estructura de datos específica (binaria) mediante la cual se transforma un arreglo en un árbol binario, en este proceso se toma el valor máximo en el árbol y se retira, este proceso se repite hasta que se acaben todos los valores dentro del árbol y queden ordenados de menor a mayor en el arreglo.	Tienen un buen desempeño con listas grandes, se comporta mejor que el Quicksort en los peores casos. Complejo de implementar.	O(n log n)
Mezcla	Se ordena un arreglo mediante la división del arreglo en mitades y luego esas mitades se dividen en la mitad hasta que quedan valores individuales y luego se juntan para tener un arreglo ordenado.	Está definido recursivamente. Puede ocupar el doble de espacio que ocupan los datos inicialmente. Es estable.	O(n log n)
Conteo	Mantiene la pista de la información adicional de sus elementos, por lo que es estable. Añade cada repetición de un valor "i" a la posición "i" de un arreglo auxiliar. Luego modifica el arreglo auxiliar de manera que cada elemento en cada índice guarda la suma de las cuentas previas. Finalmente, imprime cada elemento de la secuencia de entrada y disminuye su cuenta en 1. Las entradas deben tener enteros como llaves.	Requiere memoria adicional, pero resulta muy eficiente en tiempo dependiendo del rango de los datos. No realiza comparaciones, solo ordena enteros, bueno con los números repetidos. Es estable.	O(n+k)
Quick sort	Es un método recursivo que consiste en tomar un pivote dentro del arreglo de valores y se encarga de separar los valores más altos que el pivote en un lado y los más bajos en el otro. Por último, se repite este proceso hasta que queden ordenados los valores	Eficiente en tiempo y en memoria. Implementación requiere recursión.	O(n log n)

Rapid sort	Solo ordena llaves, por lo que es inestable. Añade cada repetición de un valor "i" a la posición "i" de un arreglo auxiliar. Luego recorre el arreglo auxiliar, añadiendo los valores a la lista por retornar. Las entradas deben ser enteros.	Requiere memoria adicional, pero resulta muy eficiente en tiempo dependiendo del rango de los datos. No realiza comparaciones, solo ordena enteros, bueno con los números repetidos. Es inestable.	O(n+k)

Paso 5: Evaluación o selección de la mejor solución (Criterios y selección)

Criterio A: Implementación

- [3] Implementación iterativa
- [2] Implementación recursiva
- [1] Implementación compleja

Criterio B: Almacenamiento temporal adicional

- [3] No usa almacenamiento temporal adicional
- [2] Usa almacenamiento temporal adicional, pero es estable.
- [1] Usa almacenamiento temporal adicional, y es inestable.

Criterio C: Complejidad temporal

- [5] Complejidad lineal
- [3] Complejidad n log n
- [1] Complejidad polinómica

Criterio D: Comparaciones

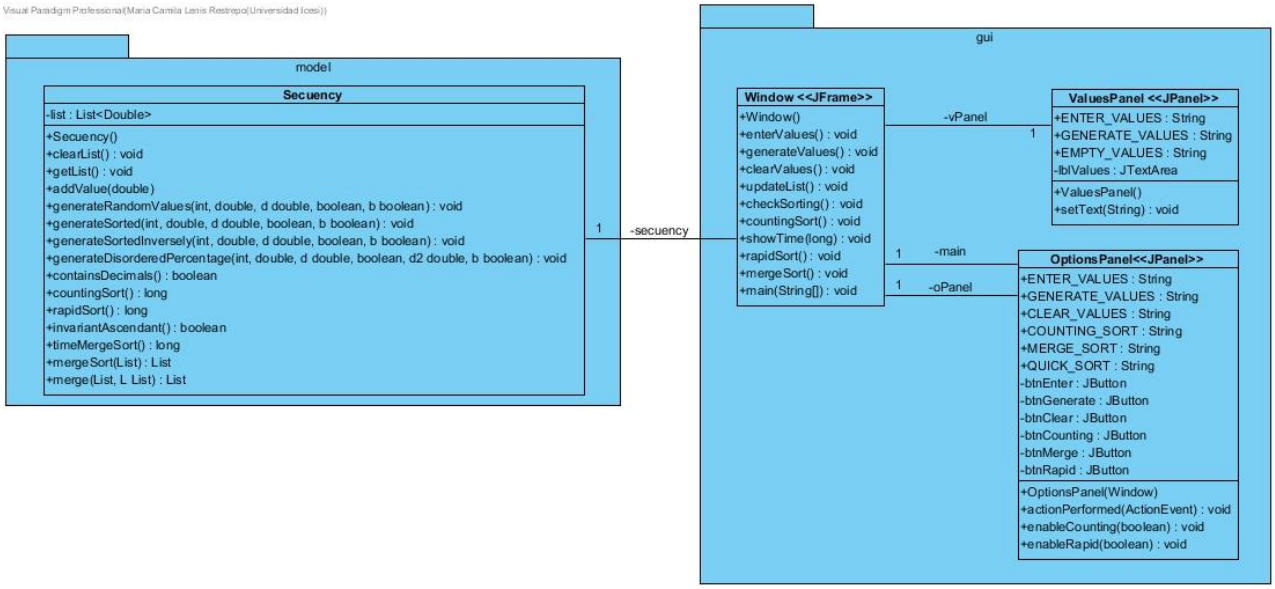
- [3] No realiza comparaciones
- [2] Realiza pocas comparaciones
- [1] Realiza numerosas comparaciones

	Criterio A	Criterio B	Criterio C	Criterio D	Total
Selección	3	1	1	2	7
Inserción	3	2	1	2	8
Montículos	1	2	3	1	7
Mezcla	2	2	3	2	9
Conteo	3	2	5	3	11
QuickSort	2	2	3	2	9
Rapid sort	3	1	5	3	10

Los tres algoritmos que elegir son: Mezcla (Merge sort), conteo (Counting sort) y Rapid sort.

Paso 6: Preparación de informes

Diseño del diagrama de clases de la solución



Pseudocódigo de los tres algoritmos. Sea list el arreglo de entrada

Rapid sort:

```
min <- list[0]
max <- min
i <- 0
while(i<list.length)
      if(list[i] < min)
            min = list[i]
      if(list[i] > max)
            max = list[i]
      i <- i+1
aux <- array(max-min+1)
i <- 0
while(i<list.length)
      value <- list[i]-min
      aux[value] <- aux[value]+1
      i <- i+1
output <- new list
i <- 0
while (i < aux.length)
      if(aux[i] > 0)
            for j in to aux[i]
                  output.add(i+min)
      i <- i+1
return output
```

Counting sort:

```
min <- list[0]
max <- min
```

```
i <- 0
while(i<list.length)
      if(list[i] < min)
            min = list[i]
      if(list[i] > max)
            max = list[i]
      i <- i+1
aux <- array(max-min+1)
i <- 0
while(i<list.length)
      value <- list[i]-min
      aux[value] <- aux[value]+1
      i <- i+1
sum <- 0
for i in 0 to aux.lenght-1
    sum <- sum + aux[i]
    aux[i] <- sum
output <- new array(list.length)
for i in n-1 downto 0
    index <- aux[list[i]-min]
    output[index-1] <- list[i]
    aux[list[i]+min] <- aux[list[i]-min]-1
return output
```

Merge sort:

```
func mergesort( var a as array )// n = a.length
     if ( n == 1 )
       return a

     var l1 as array
     var l2 as array
     for i = 0 to n-1
          if (i < n/2)
               l1.add(a[i])
          else
               l2.add(a[i])

     l1 = mergesort( l1 )
     l2 = mergesort( l2 )

     return merge( l1, l2 )
end func

func merge( var a as array, var b as array )
     var c as array
```

```
     while ( a and b have elements )
          if ( a[0] > b[0] )
               add b[0] to the end of c
               remove b[0] from b
          else
               add a[0] to the end of c
               remove a[0] from a
     while ( a has elements )
          add a[0] to the end of c
          remove a[0] from a
     while ( b has elements )
          add b[0] to the end of c
          remove b[0] from b
     return c
end func
```

Fuente: http://www.algorithmist.com/index.php/Merge_sort

Diseño de casos de pruebas unitarias

Prueba 1: Algoritmo de ordenamiento por conteo				
Clase	Método	Escenario	Entrada	Resultado
Secuency	countingSort(): long	Existe un objeto Secuency que tiene una lista donde: lista = {6, 5, 4, 3, 2, 1}	Ninguna	Lista = {1, 2, 3, 4, 5, 6}
Secuency	countingSort(): long	Existe un objeto Secuency que tiene una lista donde la lista contiene los números del 1 al 50 de manera ordenada	Ninguna	Verifica el invariante con el método invariantAscendant() de la clase Secuency
Secuency	countingSort(): long	Existe un objeto Secuency que tiene una lista donde la lista es de tamaño 50 y tiene un 3 en cada posición	Ninguna	Verifica el invariante con el método invariantAscendant() de la clase Secuency
Secuency	countingSort(): long	Existe un objeto Secuency que tiene una lista donde: Lista = {5, 5, 5, 5, 5, 1, 1, 1, 1}	Ninguna	Verifica el invariante con el método invariantAscendant() de la clase Secuency

Secuency	countingSort(): long	Existe un objeto Secuency que tiene una lista donde: Lista = {3}	Ninguna	Verifica el invariante con el método invariantAscendant() de la clase Secuency

Prueba 1: Algoritmo de ordenamiento por mezcla				
Clase	Método	Escenario	Entrada	Resultado
Secuency	timeMergeSort(): long	Existe un objeto Secuency que tiene una lista donde: lista = {6, 5, 4, 3, 2, 1}	Ninguna	Lista = {1, 2, 3, 4, 5, 6}
Secuency	timeMergeSort(): long	Existe un objeto Secuency que tiene una lista donde la lista contiene los números del 1 al 50 de manera ordenada	Ninguna	Verifica el invariante con el método invariantAscendant() de la clase Secuency
Secuency	timeMergeSort(): long	Existe un objeto Secuency que tiene una lista donde la lista es de tamaño 50 y tiene un 3 en cada posición	Ninguna	Verifica el invariante con el método invariantAscendant() de la clase Secuency
Secuency	timeMergeSort(): long	Existe un objeto Secuency que tiene una lista donde: Lista = {5.1, 5.35, 5.6, 5.2, 5.9, 1.1, 1.5, 1.6, 1.9}	Ninguna	Verifica el invariante con el método invariantAscendant() de la clase Secuency
Secuency	timeMergeSort(): long	Existe un objeto Secuency que tiene una lista donde: Lista = {3.6}	Ninguna	Verifica el invariante con el método invariantAscendant() de la clase Secuency
Secuency	timeMergeSort(): long	Existe un objeto Secuency que tiene una lista que ha sido generada aleatoriamente por el método generateRandomValues, donde: Number= 20 Start= -10 End= 10 Repeated= false OnlyInt= false	Ninguna	Verifica el invariante con el método invariantAscendant() de la clase Secuency

Prueba 1: Algoritmo de ordenamiento Rapid				
Clase	Método	Escenario	Entrada	Resultado

Secuency	rapidSort(): long	Existe un objeto Secuency que tiene una lista donde: lista = {6, 5, 4, 3, 2, 1}	Ninguna	Lista = {1, 2, 3, 4, 5, 6}
Secuency	rapidSort(): long	Existe un objeto Secuency que tiene una lista donde la lista contiene los números del 1 al 50 de manera ordenada	Ninguna	Verifica el invariante con el método invariantAscendant() de la clase Secuency
Secuency	rapidSort(): long	Existe un objeto Secuency que tiene una lista donde la lista es de tamaño 50 y tiene un 3 en cada posición	Ninguna	Verifica el invariante con el método invariantAscendant() de la clase Secuency
Secuency	rapidSort(): long	Existe un objeto Secuency que tiene una lista donde: Lista = {5.1, 5.35, 5.6, 5.2, 5.9, 1.1, 1.5, 1.6, 1.9}	Ninguna	Verifica el invariante con el método invariantAscendant() de la clase Secuency
Secuency	rapidSort(): long	Existe un objeto Secuency que tiene una lista donde: Lista = {3.6}	Ninguna	Verifica el invariante con el método invariantAscendant() de la clase Secuency
Secuency	rapidSort(): long	Existe un objeto Secuency que tiene una lista que ha sido generada aleatoriamente por el método generateRandomValues, donde: Number= 20 Start= -10 End= 10 Repeated= false OnlyInt= false	Ninguna	Verifica el invariante con el método invariantAscendant() de la clase Secuency

Análisis de complejidad temporal de los tres algoritmos

Rapid sort. Sea n el tamaño del arreglo de entrada y k su rango (valor máximo – valor mínimo +1)

1	`min <- list[0]`	c1	1
2	`max <- min`	c2	1
3	`i <- 0`	c3	1
4	`while(i<list.length)`	c4	n+1
5	`if(list[i] < min)`	c5	n
6	`min = list[i]`	c6	n
7	`if(list[i] > max)`	c7	n

8	`max = list[i]`	c8	n
9	`i <- i+1`	c9	n
10	`aux <- array(max-min+1)`	c10	1
11	`i <- 0`	c11	1
12	`while(i<list.length)`	c12	n+1
13	`value <- list[i]-min`	c13	n
14	`aux[value] <- aux[value]+1`	c14	n
15	`i <- i+1`	c15	n
16	`output <- new list`	c16	1
17	`i <- 0`	c17	1
18	`while (i < aux.length)`	c18	k+1
19	`if(aux[i] > 0)`	c19	k
20	`for j <- 1 to aux[i]`	c20	n+1
21	`output.add(i+min)`	c21	n
22	`i <- i+1`	c22	k
23	`return output`	c23	1

Sea T(n,k) el tiempo que tarda en ejecutarse el algoritmo en función de n y k.

(1) T(n,k) = c1+c2+c3+c4n+c4+c5n+c6n+c7n+c8n+c9n+c10+c11+c12n+c12+c13n+c14n +c15n+c16+c17+c18k+c18+c19k+c20n+c20+c21n+c22k+c23

Sea A una contante tal que A = c1+c2+c3+c4+c10+c11+c12+c16+c17+c18+c20+c23 , B una constante tal que B = c4+c5+c6+c7+c8+c9+c12+c13+c14+c15+c20+c21 y C una constante tal que C = c18+c19+c22

(2) T(n.k) = A+Bn+Ck

Asi, como A, B y C son coeficientes constantes, n+k>n y n+k>k, T(n,k) es O(n+k).

Counting sort. Sea n el tamaño del arreglo de entrada y k su rango (valor máximo – valor mínimo +1)

1	`min <- list[0]`	c1	1
2	`max <- min`	c2	1
3	`i <- 0`	c3	1
4	`while(i<list.length)`	c4	n+1
5	`if(list[i] < min)`	c5	n
6	`min = list[i]`	c6	n
7	`if(list[i] > max)`	c7	n
8	`max = list[i]`	c8	n
9	`i <- i+1`	c9	n
10	`aux <- array(max-min+1)`	c10	1
11	`i <- 0`	c11	1
12	`while(i<list.length)`	c12	n+1
13	`value <- list[i]-min`	c13	n
14	`aux[value] <- aux[value]+1`	c14	n
15	`i <- i+1`	c15	n

16	`sum <- 0`	c16	1
17	`for i in 0 to aux.lenght-1`	c17	k+1
18	`    sum <- sum + aux[i]`	c18	k
19	`    aux[i] <- sum`	c19	k
20	`output <- new array(list.length)`	c20	1
21	`for i in n-1 downto 0`	c21	n+1
22	`    index <- aux[list[i]-min]`	c22	n
23	`    output[index-1] <- list[i]`	c23	n
24	`    aux[list[i]+min] <- aux[list[i]-min]-1`	c24	n
25	`return output`	c25	1

Sea T(n,k) el tiempo que tarda en ejecutarse el algoritmo en función de n y k.

(1) T(n,k) = c1+c2+c3+c4n+c4+c5n+c6n+c7n+c8n+c9n+c10+c11+c12n+c12+c13n+c14n +c15n+c16+c17k+c17+c18k+c19k+c20+c21n+c21+c22n+c23n+c24n+c25

Sea A una contante tal que A = c1+c2+c3+c4+c10+c11+c12+c16+c17+c20+c21+c25 , B una constante tal que B = c4+c5+c6+c7+c8+c9+c12+c13+c14+c15+c21+c22+c23+c24 y C una constante tal que C = c17+c18+c19

(2) T(n.k) = A+Bn+Ck

Asi, como A, B y C son coeficientes constantes, n+k>n y n+k>k, T(n,k) es O(n+k).

Merge sort.

Antes de analizar la complejidad del algoritmo de ordenamiento, hay que analizar la complejidad de la función auxiliar merge(). Sean a y b los arreglos de entrada, para simplificar se asume que k=a.length = b.length = n/2. También se asume que $\forall x \in a \, \forall y \in b, x > y$

1	`var c as array`	c1	1
2	`while(a.size()>0 && b.size()>0)`	c2	k+1
3	`  if ( a[0] > b[0] )`	c3	k
4	`    c.add(b[0])`	c4	k
5	`    b.remove(b[0])`	c5	k
6	`  else`	c6	k
7	`    c.add(a[0])`	c7	0
8	`    a.remove(a[0])`	c8	0
9	`while ( a.size()>0 )`	c9	k+1
10	`  c.add(a[0])`	c10	k
11	`  a.remove(a[0])`	c11	k
12	`while ( b.size()>0 )`	c12	1
13	`  c.add(b[0])`	c13	0
14	`  b.remove(b[0])`	c14	0
15	`return c`	c15	1

Sea T(k) el tiempo que tarda en ejecutarse el algoritmo en función de k.

(1) T(k) = c1+c2k+c2+c3k+c4k+c5k+c6k+c9k+c9+c10k+c11k+c12+c15

Sea E = c1+c2+c9+c12+c15 y C = c2+c3+c4+c5+c6+c9+c10+c11

(2) T(k) = Ck+E

Reemplazando k por n/2. Sea D = C/2

(3) T(n) = Dn+E

Así, la función auxiliar es O(n).

Ahora se pasa a analizar el algoritmo de ordenamiento. Sea a el arreglo de entrada y n su tamaño. Para simplificar, se asume que $n = 2^k, k \in \mathbb{Z}^+$. Sea T(n) el tiempo que tarda en ejecutarse el algoritmo en función de n.

1	`if(n==1)`	c1	1
2	`  return a`	c2	0
3	`var l1 as array`	c3	1
4	`var l2 as array`	c4	1
5	`for i = 0 to n-1`	c5	n+1
6	`  if(i < n/2)`	c6	n
7	`    l1.add(a[i])`	c7	n/2
8	`  else`	c8	n
9	`    l2.add(a[i])`	c9	n/2
10	`l1 = mergesort(l1)`	c10	T(n/2)
11	`l2 = mergesort(l2)`	c11	T(n/2)
12	`return merge(l1,l2)`	c12	Dn+E

Así, cuando n ≠ 1 (si n = 1, sea A = c1+c2, T(n) = c1+c2 = A):

(1) T(n) = c1+c3+c4+c5n+c5+c6n+c7n/2+c8n+c9n/2+c10T(n/2)+c11T(n/2)+Dn+E

Para simplificar el análisis, se asume que c1=c2=…=c12=D=E=1. Así,

(2) T(n) =5+4n+2n/2+2T(n/2) = 2T(n/2)+5n+5

Así, tenemos que:

(3) T(n/2) = 2T(n/4)+5n/2+5

Al reemplazar (3) en (2) tenemos:

(4) T(n) = 2(2T(n/4)+5n/2+5)+5n+5 = 2^2T(n/2^2)+2*5n+2*5
(5) T(n/4) = 2T(n/8)+5n/4+5

Al reemplazar (5) en (4) tenemos:

(6) T(n) = 2^2(2T(n/8)+5n/4+5)+2*5n+2*5 = 2^3T(n/2^3)+3*5n+3*5

Al continuar de esta manera, tenemos el siguiente patrón:

(7) T(n) = 2^iT(n/2^i)+i*5n+i*5

Como sabemos que T(1) es constante (A), se debe encontrar el valor de i tal que n/2^i=1. Así, tenemos:

(8) $\frac{n}{2^i} = 1 \rightarrow n = 2^i \rightarrow \log_2 n = \log_2 2^i \rightarrow i = \log_2 n$

Al reemplazar i en (7):

(9) $T(n) = 2^{\log_2 n}T(1) + \log_2 n \cdot 5n + \log_2 n \cdot 5 = 5n\log_2 n + An + 5\log_2 n$

Por lo tanto, como el término de mayor orden en la expresión es $n\log_2 n$, la complejidad del algoritmo merge sort es O(nlogn).

Análisis de complejidad espacial de los tres algoritmos

Rapid sort Sea n el tamaño del arreglo de entrada, y k su rango (max-min+1)

Tipo	Variable	Cantidad de valores atómicos
Entrada	list	n
Auxiliar	min max i aux i value i j	1 1 1 k 1 1 1 1
Salida	output	n

Complejidad Espacial Total = Entrada + Auxiliar + Salida = 2n + k+7 = O(n+k)

Complejidad Espacial Auxiliar = k+7 = O(k)

Complejidad Espacial Auxiliar + Salida = n+k+7 = O(n+k)

Counting sort Sea n el tamaño del arreglo de entrada, y k su rango (max-min+1)

Tipo	Variable	Cantidad de valores atómicos
Entrada	list	n
Auxiliar	Min Max I Aux I Value Sum I I index	1 1 1 K 1 1 1 1 1 1
Salida	output	n

Complejidad Espacial Total = Entrada + Auxiliar + Salida = 2n + k+9 = O(n+k)

Complejidad Espacial Auxiliar = k+9 = O(k)

Complejidad Espacial Auxiliar + Salida = n+k+9 = O(n+k)

Merge sort

Primero realizamos el análisis de la función auxiliar merge. Sean a y b los arreglos de entrada, para simplificar se asume que a.length = b.length = n/2

Tipo	Variable	Cantidad de valores atómicos
Entrada	a b	n/2 n/2
Auxiliar		
Salida	c	n

Complejidad Espacial Total = Entrada + Auxiliar + Salida = 2n = O(n)

Complejidad Espacial Auxiliar = 0 = O(1)

Complejidad Espacial Auxiliar + Salida = n = O(n)

Ahora se pasa a analizar la complejidad espacial del algoritmo de ordenamiento

Tipo	Variable	Cantidad de valores atómicos
Entrada	a	n
Auxiliar	L1 L2 i	n/2 n/2 1
Salida	merge	n (pues la función merge no usa espacio auxiliar)

Complejidad Espacial Total = Entrada + Auxiliar + Salida = 3n+1 = O(n)

Complejidad Espacial Auxiliar = n+1 = O(n)

Complejidad Espacial Auxiliar + Salida = 2n+1 = O(n)

Sin embargo, hay que tener en cuenta que en cada llamada recursiva crea nuevas instancias de las variables auxiliares. Como cada llamada realiza 2 llamadas, en la llamada i ha creado $2^i(n/2^i+1)=n+2^i$ valores atómicos. Como vimos al realizar la complejidad temporal, el algoritmo realiza $\log_2 n$ llamadas, por lo que la complejidad espacial total es de $\log_2 n(n+n)=2n\log_2 n$, por lo que la complejidad espacial es de $O(n\log_2 n)$

Bibliografía

Cofre, Jonathan (Agosto 23,2018). *Algoritmos de Ordenamiento*. Tomado de: https://jona83.wordpress.com/unidad-1/algoritmos-de-ordenamiento/

Counting Sort [Presentación en Prezi]. Tomado de: https://prezi.com/rifcpibcg--b/counting-sort/?webgl=0

Florez, Sergio A.(Enero 12, 2018). *Análisis comparativo de algoritmos de ordenamiento.* Tomado de: https://pereiratechtalks.com/analisis-de-algoritmos-de-ordenamiento/

La eficiencia de los Algoritmos [Presentación PowerPoint]. Tomado del Departamento de Ciencias de Computación e I.A. de la Universidad de Granada: http://elvex.ugr.es/decsai/algorithms/slides/2%20Eficiencia.pdf

Matemática Discreta y sus aplicaciones. Kenneth H. Rosen. Quinta edición.