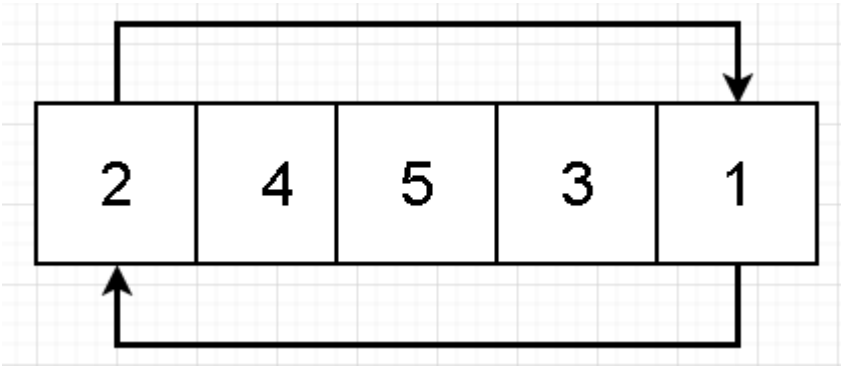
Cristian David Zuñiga Gutierrez 2259425-2724
Alejandro Marin Hoyos 2259353-3743
Juan Jose Marin Arias 2259337-2724

Ejercicio 1:

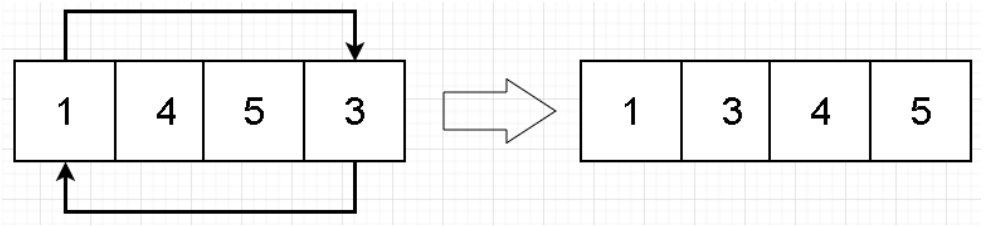
a) Supongamos que tenemos una lista desordenada [2,4,5,3,1]

i) Primero se compara el primer y último elemento de la lista (en este caso se va a comprar el 2 y 1), si el último número es mayor que el primero entonces se intercambian.



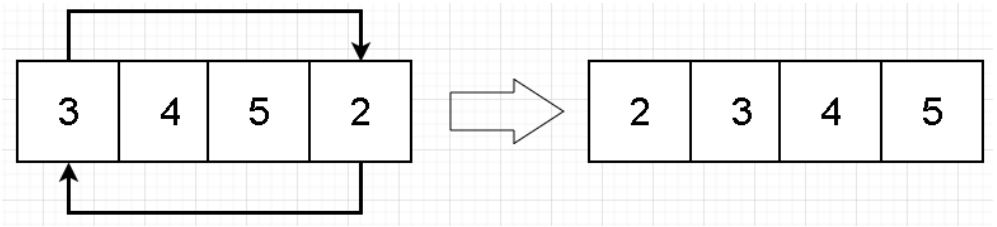
Como resultado obtenemos:

ii) En el siguiente paso si hay al menos 3 elementos, se ordena recursivamente los ⅔ iniciales de la lista de la siguiente manera:
stoogeSort(arr, i, j- k)



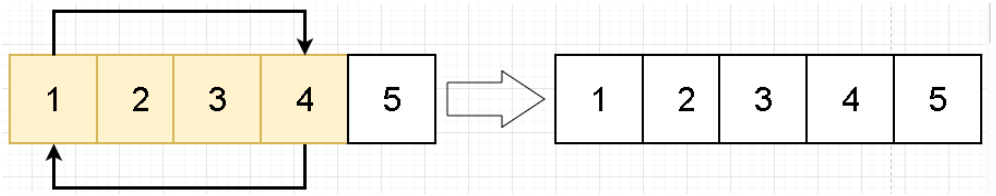
En este caso observamos que el primer elemento es menor al último entonces no se realizará el intercambio y se seguirán los pasos del algoritmo hasta ordenar los primeros ⅔ de la lista obteniendo [1,2,4,5]

iii) Luego se ordena de la misma manera los ultimos ⅔ de la lista:
stoogeSort(arr, i + k, j)



En este caso el primer elemento es mayor al último y se realiza el intercambio y a repite los pasos hasta tener los ultimos ⅔ de la lista ordenados.

iv) Como último paso se ordena nuevamente los ⅔ iniciales de la lista para confirmar que los datos finales estén ordenados:
stoogeSort(arr, i, j - k)



d) Complejidad teórica:

En este caso la ecuación de recurrencia es:

$$T(n) = 3T(\frac{n}{3/2}) + 1$$

y la solución mediante el método del maestro es:

$$T(n) = O(n^{log_{2/3} 3}) = O(n^{2.709})$$

Complejidad práctica:

En la prueba encontramos que:
para una entrada de n = 10, el tiempo de ejecución fue: 0.00010 segundos.
para una entrada de n = 100, el tiempo de ejecución fue: 0.06441 segundos.
para una entrada de n = 1000, el tiempo de ejecución fue: 16.13847 segundos.

En la práctica podemos observar el crecimiento exponencial del tiempo de ejecución lo cual es consistente con la complejidad teórica encontrada.

Donde podemos concluir comparando los tiempos y complejidad obtenida con los resultados de otros algoritmos que: El algoritmo de ordenamiento stooge sort destaca por su alta complejidad temporal, por lo cual el algoritmo no es práctico para un *n* de tamaño significativo, el enfoque del algoritmo de dividir la lista en tercios no garantiza que la división sea equitativa lo que resulta en un rendimiento deficiente a comparación con otros algoritmos.

Ejercicio 2:

a)

Este algoritmo divide el vector original en mitades iguales, luego se aplica recursión (cada llamada recursiva devuelve la moda de su respectiva mitad) en cada mitad mitad donde se continuará dividiendo cada mitad hasta obtener sub-vectores de tamaño 1.

Las modas encontradas se pasan a la función *combinarModas,* si la moda es igual en ambas mitades se retornara la moda izquierda, de lo contrario la función cuenta la frecuencia de cada elemento en la lista completa por medio de la función *Counter,* finalmente se encuentran los elementos que tienen la frecuencia máxima y se retornan.

d) Complejidad teórica:

Para este algoritmo encontramos que la relación de recurrencia puede ser expresada de la siguiente manera:

$$T(n) = 2T(\frac{n}{2}) + O(n)$$

Cuya solución por el método del maestro corresponde a:

$$T(n) = O(n \log n)$$

Complejidad Práctica:

En la prueba encontramos que:
para una entrada de n = 10, el tiempo de ejecución fue: 0.00015 segundos.
para una entrada de n = 100, el tiempo de ejecución fue: 0.00038 segundos.
para una entrada de n = 1000, el tiempo de ejecución fue: 0.00473 segundos.
para una entrada de n = 10000, el tiempo de ejecución fue: 0.03960 segundos.

En la práctica podemos observar que a medida que el tamaño de n aumenta el tiempo de ejecución no parece crecer linealmente con respecto al tamaño de la entrada, donde podríamos decir que tiene un crecimiento logarítmico lo cual concuerda con la complejidad teórica hallada anteriormente.

Para los tamaños n=100 y n=1000 el tiempo de ejecución sigue siendo bastante bajo, lo que indica que el algoritmo tiene una eficiencia razonable para un n de tamaño moderado.

Ejercicio 3:

Algoritmo QuickSort:

Entrada (n)	Tiempo Real (seg.)	Complejidad $(nlogn)$	Constantes
10	0.00002	10	0.000002
10	0.00002	10	0.000002
10	0.00001	10	0.000001
50	0.00009	84.94	0.00000106
50	0.00011	84.94	0.000001295
50	0.00009	84.94	0.00000106
100	0.00024	200	0.0000012
100	0.00025	200	0.00000125
100	0.00025	200	0.00000125
500	0.00219	1,349.48	0.000001623
500	0.00206	1,349.48	0.000001527
500	0.00207	1,349.48	0.000001534
1000	0.00486	3000	0.00000162
1000	0.00452	3000	0.000001507
1000	0.00443	3000	0.000001477
2000	0.01199	6,602.05	0.000001816
2000	0.01100	6,602.05	0.000001666
2000	0.01099	6,602.05	0.000001665
5000	0.03099	18,494.85	0.000001676
5000	0.03100	18,494.85	0.000001676
5000	0.03100	18,494.85	0.000001676
10000	0.08099	40,000	0.000002025
10000	0.07900	40,000	0.000001975
10000	0.08100	40,000	0.000002025
		Constante:	0.000001566

Algoritmo Insertion-Sort:

Entrada (n)	Tiempo Real (seg.)	Complejidad (n^2)	Constantes
10	0.00002	100	0.0000002
10	0.00002	100	0.0000002
10	0.00002	100	0.0000002
50	0.00009	2,500	0.00000004
50	0.00012	2,500	0.00000005
50	0.00009	2,500	0.00000004
100	0.00035	10,000	0.000000035
100	0.00032	10,000	0.000000032
100	0.00034	10,000	0.000000034
500	0.00914	250,000	0.0000000366
500	0.00725	250,000	0.000000029
500	0.00777	250,000	0.0000000311
1000	0.03201	1,000,000	0.000000032
1000	0.03359	1,000,000	0.0000000336
1000	0.03037	1,000,000	0.0000000304
2000	0.12735	4,000,000	0.0000000318
2000	0.12618	4,000,000	0.0000000315
2000	0.12613	4,000,000	0.0000000315
5000	0.80648	25,000,000	0.0000000323
5000	0.79223	25,000,000	0.0000000317
5000	0.78289	25,000,000	0.0000000313
10000	3.12125	100,000,000	0.0000000312
10000	3.14341	100,000,000	0.0000000314
10000	3.21802	100,000,000	0.0000000322
		Constante	0.000000054

Algoritmo Merge-Sort:

Entrada (n)	Tiempo Real (seg.)	Complejidad $(nlogn)$	Constantes
10	0.00003	10	0.000003
10	0.00003	10	0.000003
10	0.00003	10	0.000003
50	0.00013	84.94	0.00000153
50	0.00012	84.94	0.00000141
50	0.00013	84.94	0.00000153
100	0.00029	200	0.00000145
100	0.00025	200	0.00000125
100	0.00027	200	0.00000135
500	0.00229	1,349.48	0.00000169
500	0.00150	1,349.48	0.00000111
500	0.00165	1,349.48	0.00000122
1000	0.00355	3000	0.000001183
1000	0.00334	3000	0.000001113
1000	0.00478	3000	0.000001593
2000	0.0110	6,602.05	0.000001666
2000	0.01095	6,602.05	0.000001659
2000	0.01098	6,602.05	0.000001663
5000	0.02097	18,494.85	0.000001134
5000	0.02098	18,494.85	0.000001134
5000	0.02199	18,494.85	0.000001189
10000	0.04599	40,000	0.000001150
10000	0.04599	40,000	0.000001150
10000	0.04598	40,000	0.000001150
		Constante	0.000001556

Con las pruebas prácticas podemos corroborar lo encontrado al calcular la complejidad teórica de los algoritmos donde encontramos que QuickSort y MergeSort tienen una complejidad de $n\ log\ n$ mientras que InsertionSort tiene una complejidad de n^2 podemos confirmarlo al ver la similitud entre los tiempos de QuickSort y MergeSort, como también en el crecimiento exponencial en InsertionSort. Si comparamos estos algoritmos podemos decir que: Quicksort es eficiente para grandes conjuntos de datos, Mergesort garantiza un rendimiento consistente y es estable, mientras que Insertionsort es útil para conjuntos de datos pequeños o casi ordenados, la elección del algoritmo a utilizar siempre dependerá del contexto del problema.