1. Insertion Sort

1.1. Formalmente:

Entradas: Una secuencia S con N números:

$$S = \langle a_1, a_2, a_3, \cdots, a_N \rangle \, ; N > 0 \tag{1}$$

Salidas: Una secuencia S con N números tal que:

$$S = \langle a_1 \leq a_2 \leq, \cdots, \leq a_N \rangle \, ; \tag{2}$$

1.2. Pseudocódigo:

Algorithm 1 Insertion Sort.

```
1: function InsertionSort( S )
2: for i ← 2 to |S| do
3:     k ← S[j]
4:     i ← j − 1
5:     while i > 0 ∧ S[i] > k do
6:         S[i + 1] ← S[i]
7:         i ← i − 1
8:     end while
9:     S[i + 1] ← k
10: end for
11: return S
12: end function
```

1.3. Invariantes:

Se puede encontrar una invariante en este algoritmo.

1.3.1. En cada iteración for la secuencia entre 1 y $j-1$ siempre está ordenada:

1. Formalmente: $S \, \langle a_1, a_2, a_3, \cdots, a_{N-1} \rangle \equiv S' \, \langle a_1, a_2, a_3, \cdots, a_{N-1} \rangle$ Donde $S' | a_1 \leq a_2 \leq, \cdots, \leq a_N$

2. Prueba por inducción:

   a) Inicialización ($i \leftarrow 1$):
   En la primera iteración del ciclo se cumple que: $S \equiv S'$ ya que es un único elemento y un elemento por definición siempre está ordenado.

   b) Mantenimiento/Iteración/Actualización/Procesamiento ($2 \leq i \leq N$):
   Debido a la condición del ciclo **while** se controla que los números mayores a $k = S[j]$ queden en las posiciones $j, j+1, j+2, \cdots, N$ y los menores o iguales a k en las posiciones $j-1, j-2, \cdots, 1$.

   c) Terminación ($i \leftarrow N$):
   Al terminar el algoritmo S está ordenado.

1.4. Análisis de Complejidad Temporal:

1.4.1. Peor caso:

Para el algoritmo Insertion Sort el peor de los casos es que tenga que organizar cada uno de los elementos que están en el vector repitiéndose n veces en el ciclo **for** y n veces en el ciclo **while**. Hay que tener en cuenta que entre la secuencia este más ordenada cuando llega el algoritmo a de ser más demorado por consiguiente la complejidad queda:

$$T(n) = \begin{cases} c & n \leq 0 \\ \\ nT(1) + cn & n \geq 1 \end{cases}$$

$$T(n) = O(n^2)$$

1.4.2. Mejor caso:

Para el algoritmo Insertion Sort el mejor de los casos es que no entre al ciclo **while**, es decir, que llegue la secuencia ya ordenada por lo cual solo haría n veces el recorrido en la secuencia organizándola. La complejidad quedaría:

$$T(n) = \begin{cases} c & n \leq 0 \\ \\ cn & n \geq 1 \end{cases}$$

$$T(n) = \Omega(n)$$

1.5. Análisis de Complejidad Espacial:

Para todos los casos, Insertion Sort tiene una coplejidad espacial $O(1)$, ya que al no hacer ningún cambio, no debe crear espacio en memoria. Asumiendo que se le reserva memoria una sola vez a k.

1.6. Análisis Práctico:

Para analizar la teoría anteriormente establecida en casos prácticos se tomaron arreglos de tamaño 1 hasta 1000. Y se puso a prueba el algoritmo Insertion Sort para ordenar estos arreglos en tres escenarios distintos: Cuando todos los arreglos estaban ya ordenados ascendentemente (ordenado), ordenados descendentemente (desordenado) y mezclados, es decir, números aleatorios (aleatorio). Posteriormente se graficó, tamaño (n) vs tiempo (seg), el comportamiento de este algoritmo en los tres escenarios obteniendo la siguiente gráfica:

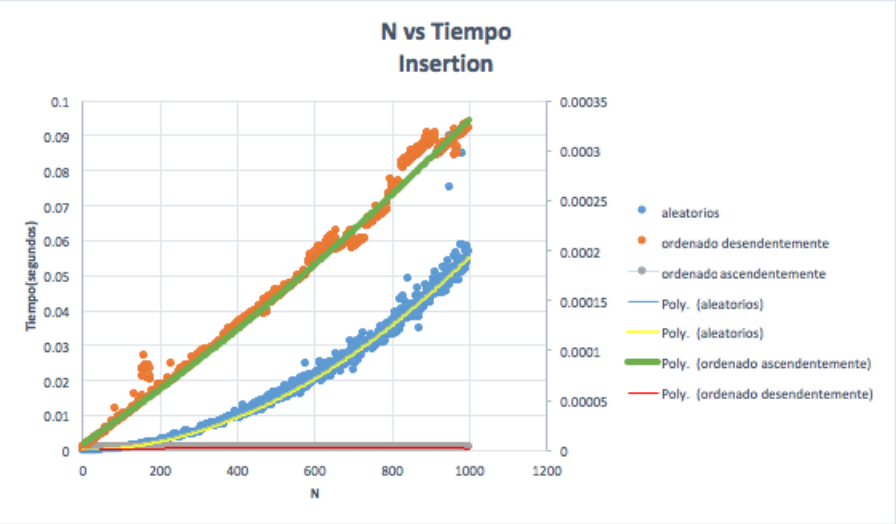

Figura 1: Comportamiento algoritmo Insertion Sort.

Insertion Sort es un algoritmo que depende de los datos que vengan en el vector que se va a organizar. Es decir si el vector que recibo ya esta todo organizado este será mucho mas eficiente como se puede ver en la línea gris de la grafica de Insertion donde el tiempo es constante a diferencia si el vector que se recibe tiene parcialmente números ordenados se puede notar que el algoritmo demorara más tiempo como se observa en el conjunto de datos color azul de la misma grafica. Finalmente si los datos están totalmente desordenados este algoritmo demorara muchos mas tiempo que los dos casos anteriores como se ve en la grafica naranja.

2. Merge Sort

2.1. Formalmente:

Entradas: Una secuencia S con N números:

$$S = \langle a_1, a_2, a_3, \cdots, a_N \rangle \, ; N > 0 \tag{3}$$

Salidas: Una secuencia S con N números tal que:

$$S = \langle a_1 \leq a_2 \leq, \cdots, \leq a_N \rangle \, ; \tag{4}$$

2.2. Pseudocódigo:

Algorithm 2 Merge Sort.

```
1: procedure MergeSort( S, f, l )
2: if f < l then
3:     h ← ⌊(f + l)/2⌋
4:     MergeSort( S, f, h )
5:     MergeSort( S, h + 1, l )
6:     Merge( S, f, h, l )
7: end if
8: end procedure
```

Algorithm 3 Merge.

```
1: procedure Merge( S, p, q, r )
2: n_1 ← q − p + 1
3: n_2 ← r − q
4: Let L[1, n_1 + 1] and R[1, n_2 + 1]
5: for i ← 0 to n_1 do
6:    L[i] ← S[p + i − 1]
7: end for
8: for i ← 0 to n_2 do
9:    R[i] ← S[q + j]
10: end for
11: L[n_1 + 1] ← ∞ ∧ R[n_2 + 1] ← ∞
12: i ← 1 ∧ j ← 1
13: for k ← p to r do
14:    if L[i] ≤ R[j] then
15:       S[k] ← L[i]
16:       i ← i + 1
17:    else
18:       S[k] ← R[j]
19:       j ← j + 1
20:    end if
21: end for
22: end procedure
```

2.3. Invariantes:

Se puede encontrar una invariante en este algoritmo.

2.3.1. En cada iteración for, la secuencia $S[p, k–1]$ contiene los k–elementos más pequeños entre L y R, ordenados. Además, $L[i]$ y $R[j]$ son los valores más pequeños de cada secuencia que no han sido copiados en S:

1. Prueba por inducción:

   a) Inicialización:
   En la primera iteración del ciclo la secuencia $S[p, k-1]$ con tiene un único elemento que corresponde al más pequeño debido a la condición de la sentencia **if** que solo deja agegar a esta sentencia el elemneto más pequeño de L o de R.

   b) Mantenimiento/Iteración/Actualización/Procesamiento:
   A medida que se va ordenando cada mitad recursiva de la secuencia por separado, se tienen las subsecuencias L y R que representan las dos mitades de la secuencia S ordenada por ende el elemento que se copie a S desde L o R se asegura, debido la condición **if**, que es el menor entre estas dos subsecuencias.

   c) Terminación ($i \leftarrow N$):
   Al terminar el algoritmo la secuencia $S[p, k-q] \equiv S[N]$ contiene todos los elementos ordenados y las subsecuencias L y R ya no contienen elementos esto aseguro que para llegar a la secuencia S totalmente ordenada se debieron copiar de manera ascendente (primero los más pequeños) los elementos de cada subsecuencia L y R.

2.4. Análisis de Complejidad Temporal:

Para este algoritmo de ordenamiento todos los casos (promedio, peor y mejor) son el mismo ya que no existen validaciones de orden para dividir la secuencia y unirla, es decir, así la secuencia esté ordenada Merge Sort la divide y une. Por lo tanto la complejidad de este algoritmo se define como:

$2T(\frac{n}{2})$ ya que la secuencia se divide en dos y el algoritmo se aplica a cada mitad.

cn es la complejidad ($O(n)$ peor de los casos) para la unión de las dos sub-secuencias.

$$T(n) = \begin{cases} c & n \leq 0 \\ \\ 2T(\frac{n}{2}) + cn & n \geq 1 \end{cases}$$

$$T(n) = \theta \equiv O \equiv \Omega(nlog_2(n))$$

2.5. Análisis de Complejidad Espacial:

En cada una de las iteraciones de merge sort para dividir la secuencia S se crea una nueva secuencia de tamaño n. Y teniendo en cuenta lo encotrado en el inciso anterior la cantidad de veces que se repite, en el peor de los casos, es $log_2(n)$ (altura del árbol), por lo que el total de tamaño extra tomado es, para las tres complejidades, de $O(nlog_2(n))$.

2.6. Análisis Práctico:

Para analizar la teoría anteriormente establecida en casos prácticos se tomaron arreglos de tamaño 1 hasta 1000. Y se puso a prueba el algoritmo Merge Sort para ordenar estos arreglos en tres escenarios distintos: Cuando todos los arreglos estaban ya ordenados ascendentemente (ordenado), ordenados descendentemente (desordenado) y mezclados, es decir, números aleatorios (aleatorio). Posteriormente se graficó, tamaño (n) vs tiempo (seg), el comportamiento de este algoritmo en los tres escenarios obteniendo la siguiente gráfica:

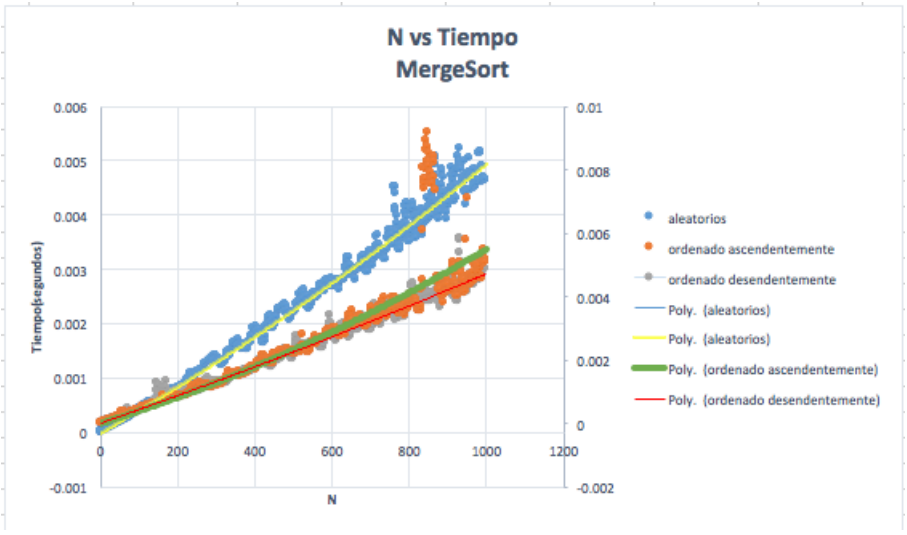

Figura 2: Comportamiento algoritmo Merge Sort.

En el caso de Merge Sort todos lo casos tienen la misma complejidad, esto se puede

ver reflejado en la gráfica ya que estás coinciden con la forma de la función *nlogn*. Sin embargo, es un poco extraño que el aleatorio haya tomado un poco más de tiempo. Esto pudo haber sido causado por cambios en el uso del computador el algoritmo hace exactamente las mismas operaciones para todos los casos.

3. Quick Sort

3.1. Formalmente:

Entradas: Una secuencia S con N números:

$$S = \langle a_1, a_2, a_3, \cdots, a_N \rangle \; ; N > 0 \tag{5}$$

Salidas: Una secuencia S con N números tal que:

$$S = \langle a_1 \leq a_2 \leq, \cdots, \leq a_N \rangle \; ; \tag{6}$$

3.2. Pseudocódigo:

Algorithm 4 Pseudcódigo de la función que reordena los números alrededor del pivote.

```
1: function Partittion( S, L, R )
2: pivot ← S[R]
3: i ← L
4: for j ← L to R do
5:    if S[j] ≤ pivot then
6:       tmp ← S[i]
7:       S[i] ← S[j]
8:       S[j] ← tmp
9:       i ← i + 1
10:   end if
11:   tmp ← S[i]
12:   S[i] ← S[R]
13:   S[R] ← tmp
14: end for
15: return i
16: end function
```

Algorithm 5 Quick Sort.

```
1: procedure QuickSort( S, L, R )
2: if L < R then
3:    index ← Partittion(S, L, R)
4:    QuickSort(S, L, index − q)
5:    QuickSort(S, index + 1, R)
6: end if
7: end procedure
```

3.3. Invariantes:

Se puede encontrar una invariante en este algoritmo.

3.3.1. Todos los elementos al lado derecho del pivote son mayores que el

1. Formalmente: $\forall i > R | ai, a_{i+1}, \cdots, a_R \geq a_R$
2. Prueba por inducción:
   a) Inicialización ($pivot \leftarrow R$):
   El pivote es el ultimo elemento por lo tanto es igual a el y no existen otros elementos a lado derecho de el.
   b) Mantenimiento/Iteración/Actualización/Procesamiento:
   Debido a la condición**if** se abrirá espacio a lado izquierdo de la secuencia reubicando solo los elementos menores al pivote y dejando el espacio donde el pivote debería ir (por delante de los más pequeños que el y detrás de los más grandes). En la siguiente iteración habrán dos pivotes, uno mayor y uno menor para los cuales se cumple la misma condición de arriba.
   c) Terminación:
   Al terminar la secuencia S está ordenada.

3.4. Análisis de Complejidad Temporal:

3.4.1. Peor caso:

Que el pivote seleccionado sea el último elemento, dejando una de las particiones vacías. Así el algoritmo se repetiría $n(n+1)/2$ solo en la parte derecha recorriendo así todo el arreglo menos la última posición. Por lo que la complejidad del peor de los caso sería:

$$T(n) = \begin{cases} c & n \leq 0 \\ \\ 2T(n) + cn & n \geq 1 \end{cases}$$
$$T(n) = O(n^2)$$

3.4.2. Mejor caso:

El pivote seleccionado queda justo en la mitad de los elementos de todas las iteraciones dando lugar a particiones de tamaño $\frac{n}{2}$. Generando un árbol de altura máxima $log_2(n)$. Por lo que la complejidad del mejor de los caso sería:

$$T(n) = \begin{cases} cn \leq 0 \\ \\ 2T(\frac{n}{2}) + cn & n \geq 1 \end{cases}$$
$$T(n) = \theta(nlog_2(n))$$

3.5. Análisis de Complejidad Espacial:

Para Quick Sort la complejidad espacial vendría siendo las veces que se crea la variable *tmp*. Esto es $O(log_2 n)$ veces, ya que se hace una vez por cada llamada recursiva.

3.6. Análisis Práctico:

Para analizar la teoría anteriormente establecida en casos prácticos se tomaron arreglos de tamaño 1 hasta 1000. Y se puso a prueba el algoritmo Quick Sort para ordenar estos arreglos en tres escenarios distintos: Cuando todos los arreglos estaban ya ordenados ascendentemente (ordenado), ordenados descendentemente (desordenado) y mezclados, es decir, números aleatorios (aleatorio). Posteriormente se graficó, tamaño (n) vs tiempo (seg), el comportamiento de este algoritmo en los tres escenarios obteniendo la siguiente gráfica:

Este algoritmo es dependiente del pivote dado que el dividir del algoritmo es tan efectivo como la buena elección del pivote. Se escogió el pivote como el de más a la derecha para poder forzar el peor caso con un conjunto de datos específicos. Esto se nota en la gráfica ya que en el peor de los casos toma un tiempo mayor que en los otros dos en los cuales las particiones son más grandes lo cual reduce el tiempo de procesamiento del algoritmo.

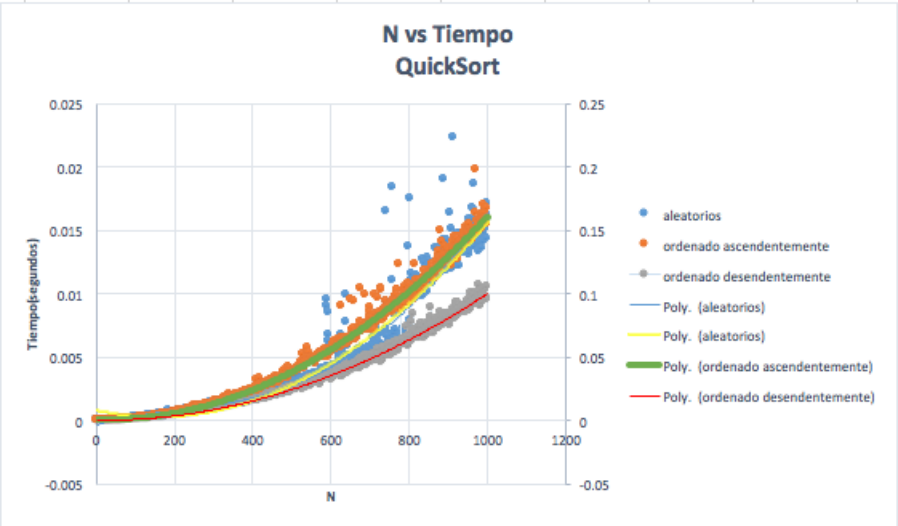

Figura 3: Comportamiento algoritmo Quick Sort.