Estudiante: Bryan Alexander Díaz Vásquez

País: El Salvador

1. Búsqueda Binaria (Binary Search)

La **búsqueda binaria** se utiliza para encontrar un elemento dentro de una lista ordenada. Este algoritmo funciona dividiendo repetidamente el espacio de búsqueda a la mitad y comparando el valor objetivo con el valor medio de la lista.

- **Mejor caso**: O(1)
  El mejor caso ocurre cuando el elemento que estamos buscando se encuentra en el primer intento (en el medio de la lista), lo que significa que no es necesario realizar ninguna búsqueda adicional.
- **Peor caso**: O(log n)
  En el peor caso, la búsqueda requiere dividir la lista repetidamente en mitades. El número de divisiones necesarias para reducir el espacio de búsqueda a un solo elemento es logarítmico en relación al tamaño de la lista. Así que la complejidad temporal en el peor caso es O(log n).
- **Espacio adicional**: O(1)
  La búsqueda binaria no requiere espacio adicional significativo, ya que trabaja directamente sobre la lista original y solo usa unos pocos punteros o índices.

2. Bubble Sort

El **Bubble Sort** es un algoritmo de ordenamiento sencillo que compara elementos adyacentes y los intercambia si están en el orden incorrecto. Este proceso se repite hasta que la lista está completamente ordenada.

- **Mejor caso**: O(n)
  Si la lista ya está ordenada, el algoritmo solo necesita pasar una vez por todos los elementos sin realizar ningún intercambio, lo que da como resultado una complejidad lineal de O(n).
- **Peor caso**: $O(n^2)$
  El peor caso ocurre cuando la lista está ordenada en orden inverso. En este caso, el algoritmo tiene que hacer tantas comparaciones e intercambios como el número total de elementos, lo que lleva a una complejidad cuadrática $O(n^2)$.
- **Espacio adicional**: O(1)
  El algoritmo **Bubble Sort** es un algoritmo in-place, lo que significa que no necesita espacio adicional para realizar las operaciones, aparte de algunas variables auxiliares para las comparaciones.

3. Selection Sort

Selection Sort es otro algoritmo de ordenamiento que funciona seleccionando el mínimo (o máximo) valor de una lista y colocándolo en la posición correcta, repitiendo este proceso hasta que todos los elementos están ordenados.

- **Mejor caso**: $O(n^2)$
  Aunque la lista ya esté ordenada, **Selection Sort** sigue haciendo el mismo número de comparaciones. Es un algoritmo que no mejora su desempeño incluso si la lista ya está en orden, por lo que la complejidad sigue siendo $O(n^2)$ en todos los casos.
- **Peor caso**: $O(n^2)$
  Al igual que en el mejor caso, el número de comparaciones y selecciones sigue siendo el mismo en el peor caso: el algoritmo realiza una búsqueda de mínimo en el resto de la lista en cada iteración, lo que da como resultado $O(n^2)$.
- **Espacio adicional**: $O(1)$
  Selection Sort es un algoritmo in-place, lo que significa que no necesita espacio adicional para funcionar más allá de las variables de control.

4. Insertion Sort

Insertion Sort ordena una lista construyéndola gradualmente. En cada iteración, el algoritmo toma un elemento y lo inserta en su posición correcta dentro de la sublista ordenada, desplazando los elementos a la derecha si es necesario.

- **Mejor caso**: $O(n)$
  En el mejor caso, cuando la lista ya está ordenada, el algoritmo simplemente compara cada elemento con el anterior y no realiza ningún intercambio. Por lo tanto, solo realiza una pasada por los elementos, lo que resulta en una complejidad $O(n)$.
- **Peor caso**: $O(n^2)$
  En el peor caso, cuando la lista está completamente desordenada, el algoritmo tendrá que mover cada elemento a través de toda la sublista ordenada. En este caso, la complejidad es $O(n^2)$, ya que se realizan muchas comparaciones e intercambios.
- **Espacio adicional**: $O(1)$
  Al igual que otros algoritmos in-place, **Insertion Sort** no requiere espacio adicional más allá de las variables auxiliares para realizar las operaciones.

5. Merge Sort

Merge Sort es un algoritmo de ordenamiento basado en el paradigma de "divide y vencerás". Divide la lista en dos mitades, las ordena recursivamente y luego las fusiona de manera eficiente.

- **Mejor caso**: $O(n \log n)$
  Incluso si la lista está ya parcialmente ordenada, **Merge Sort** siempre realiza las

mismas divisiones y fusiones. La complejidad es O(n log n) tanto en el mejor caso como en el peor caso, debido al proceso de dividir y combinar los elementos.

- **Peor caso**: O(n log n)
  La complejidad O(n log n) también se aplica en el peor caso. El algoritmo siempre divide la lista en dos y la fusiona, independientemente del orden inicial de los elementos.
- **Espacio adicional**: O(n)
  Merge Sort requiere espacio adicional para almacenar las sublistas mientras se realiza la fusión. Esto lo hace menos eficiente en términos de espacio, ya que necesita un array adicional del tamaño de la lista original.

6. Quick Sort

Quick Sort es otro algoritmo basado en "divide y vencerás". Selecciona un "pivote" y divide la lista en dos sublistas: una con elementos menores que el pivote y otra con elementos mayores. Luego aplica recursivamente el mismo proceso en ambas sublistas.

- **Mejor caso**: O(n log n)
  El mejor caso ocurre cuando el pivote divide la lista de manera equilibrada en cada paso, lo que hace que el número de divisiones y fusiones crezca logarítmicamente. Esto da como resultado una complejidad O(n log n).
- **Peor caso**: $O(n^2)$
  El peor caso ocurre cuando el pivote seleccionado no divide bien la lista, como cuando siempre se elige el elemento más grande o el más pequeño. Esto puede hacer que el algoritmo realice un número de comparaciones y particiones lineales en cada paso, lo que resulta en una complejidad cuadrática $O(n^2)$.
- **Espacio adicional**: O(log n)
  La complejidad espacial es O(log n) debido a la pila de recursión. Cada vez que el algoritmo hace una llamada recursiva, se agrega un nuevo marco a la pila. En el caso promedio, esto es logarítmico, pero en el peor caso puede llegar a O(n).