Tema:

Implementación y Análisis de Algoritmos de Búsqueda y Ordenamiento en Java

Instructores:

Oscar Lemus
Eduardo Calles

Integrantes:

Escobar Cisneros, Lisbeth
Mejía Reinosa, Ronald Eduardo
Salguero Hernández, Alejandra Marisol
Salguero Hernández, Mario Enrique

Introducción ... **3**
Objetivos ... **3**
Algoritmo de búsqueda secuencial ... **4**
Complejidad ... 5
Análisis paso a paso: ... 6
Algoritmo de búsqueda secuencial en pseudocódigo ... 6
Ejemplo con código según javascript ... 7
Algoritmo de búsqueda binaria ... **7**
Complejidad ... 8
Análisis paso a paso ... 9
Algoritmo de búsqueda binaria en pseudocódigo ... 10
Ejemplo con código según python: ... 11
Algoritmo de ordenamiento de burbuja ... **11**
Complejidad ... 12
Análisis paso a paso ... 13
Algoritmo de ordenamiento de burbuja en pseudocodigo ... 14
Ejemplo con código según Kotlin: ... 14
Algoritmo de ordenamiento por inserción ... **15**
Complejidad: ... 16
Análisis paso a paso ... 16
Algoritmo de ordenamiento por inserción en pseudocodigo ... 17
Ejemplo con código según Java: ... 18
Algoritmo de ordenamiento por selección ... **18**
Complejidad ... 19
Análisis paso a paso ... 20
Algoritmo de ordenamiento por selección en pseudocodigo ... 21
Ejemplo con código según c#: ... 22
Conclusión ... **23**
Recomendaciones ... **24**
Biografias ... 25

Introducción

La capacidad de ordenar y buscar un objetivo de manera precisa en un conjunto de datos puede ser una ventaja extremadamente importante para cualquier empresa que desee obtener un beneficio del tratamiento de datos. Dada la creciente cantidad de datos que diversas fuentes generan día con día, se vuelve de suma importancia tener algoritmos que logren ordenar de manera eficiente grandes conjuntos de datos

En este documento se presentan descripciones detalladas sobre los diversos algoritmos de búsqueda y ordenamientos vistos en clase resaltando su importancia en la informática y su amplio uso en aplicaciones desde procesos de datos simples a análisis de datos complejos.

Objetivos

Objetivo Principal

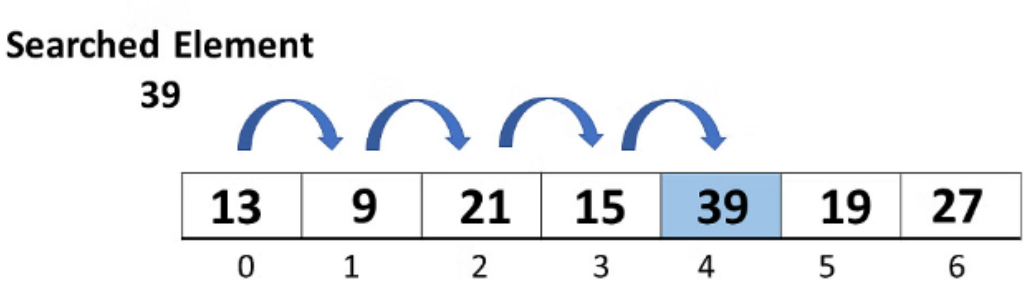
- Implementar y comparar cinco algoritmos de ordenamiento (Bubble Sort, Selection Sort, Insertion Sort, Merge Sort y Quick Sort) para evaluar su eficiencia y rendimiento en diferentes escenarios.

Objetivos Secundarios

- Analizar la complejidad temporal y espacial de cada algoritmo para entender sus limitaciones y ventajas en diferentes situaciones.
- Evaluar el rendimiento de cada algoritmo en diferentes tamaños de datos y tipos de datos para determinar su escalabilidad y robustez en diferentes aplicaciones.

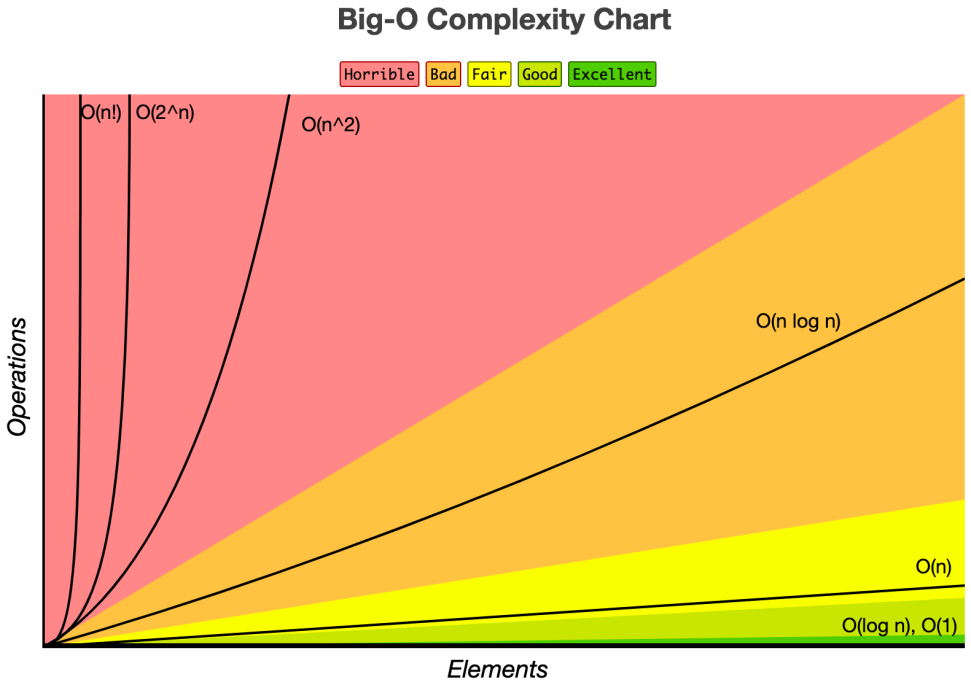
Algoritmo de búsqueda secuencial

En informática, la búsqueda lineal o secuencial es un método para encontrar un valor objetivo dentro de una lista. Comprueba secuencialmente cada elemento de la lista en busca del valor objetivo hasta que se encuentra una coincidencia o hasta que se han buscado todos los elementos. La búsqueda lineal se ejecuta en el peor tiempo lineal y realiza como máximo n comparaciones, donde n es la longitud de la lista.



Complejidad

Este algoritmo se clasifica como O(n). Esto significa que el tiempo de ejecución del algoritmo crece linealmente con el tamaño de la entrada, es decir, si el tamaño de la entrada se duplica, el tiempo de ejecución también se duplica.



Mejor caso:

O(1) En el mejor caso, el elemento que estamos buscando se encuentra en la primera posición del arreglo. En este caso, el algoritmo solo necesita realizar una comparación y devuelve la posición del elemento.

Caso promedio:

O(n/2) En el caso promedio, el elemento que estamos buscando se encuentra en una posición aleatoria dentro del arreglo. En este caso, el algoritmo necesita realizar una cantidad de comparaciones proporcional a la mitad del tamaño del arreglo.

Peor caso:

O(n) En el peor caso, el elemento que estamos buscando no se encuentra en el arreglo. En este caso, el algoritmo necesita realizar una comparación para cada elemento del arreglo, lo que significa que la cantidad de comparaciones es proporcional al tamaño del arreglo.

Complejidad general:
O(n) La complejidad general del algoritmo de búsqueda secuencial es O(n), ya que en el peor caso, el algoritmo necesita realizar una cantidad de comparaciones proporcional al tamaño del arreglo.

Análisis paso a paso:

Desglosamos el proceso de búsqueda Secuencial:
Paso 1: Inicialización: Recibe un arreglo arr y un elemento target que se busca.

Paso 2: Recorrido del arreglo: Recorre el arreglo desde la primera posición hasta la última posición.Compara cada elemento del arreglo con el elemento target.

Paso 3: Comparación y retorno: Si encuentra el elemento target, devuelve la posición del elemento.Si no encuentra el elemento target, continúa con la siguiente iteración.

Paso 4: Caso no encontrado: Si no encuentra el elemento target en todo el arreglo, devuelve -1.

Algoritmo de búsqueda secuencial en pseudocódigo

```
PROCEDURE LinearSearch(arr, target)
  i = 0
  WHILE i < longitud(arr) HACER
    IF arr[i] == target
      RETURN i
    END IF
    i = i + 1
  END WHILE
  RETURN -1
END PROCEDURE
```

Explicación

- **arr** es la matriz de entrada.
- **target** es el elemento que estamos buscando.
- **i** es el índice actual, que se utiliza para recorrer la matriz.

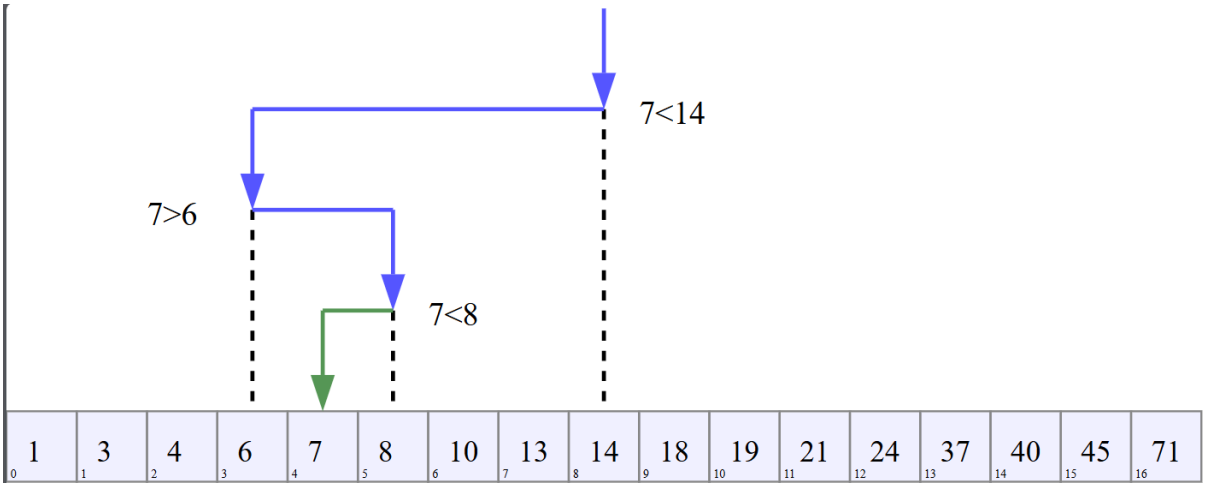
- El algoritmo recorre la matriz secuencialmente, comparando cada elemento con el elemento de destino.
- Si se encuentra el elemento de destino, el algoritmo devuelve su índice. De lo contrario, devuelve -1 para indicar que el elemento no está presente.

Ejemplo con código según javascript

```javascript
function linearSearch(arr, target) {
  for (let i = 0; i < arr.length; i++) {
    if (arr[i] === target) {
      return i;
    }
  }
  return -1;
}
```

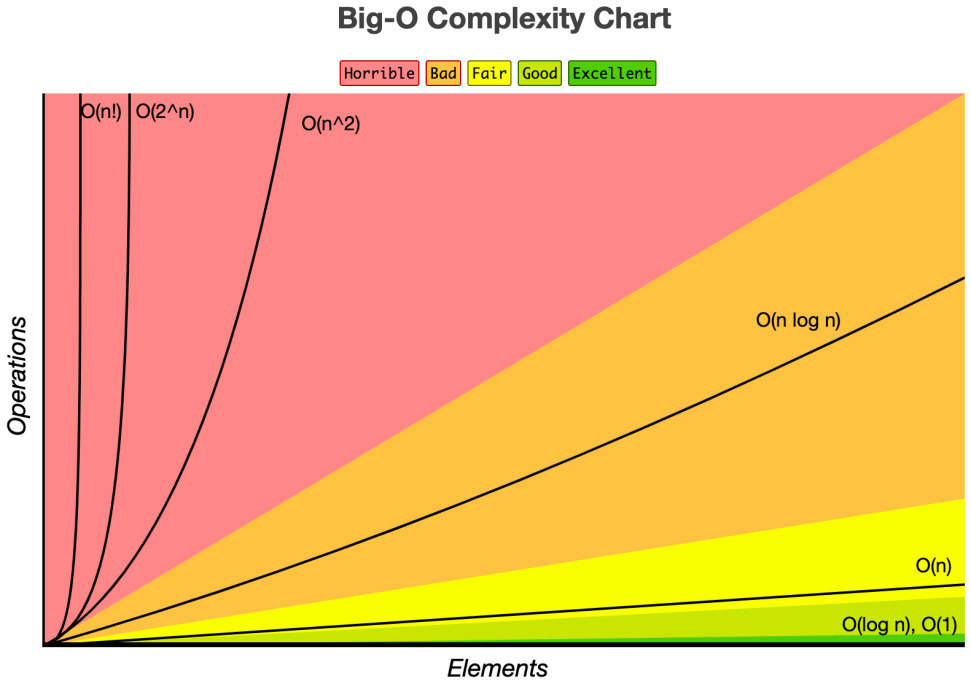
Algoritmo de búsqueda binaria

En informática, la búsqueda binaria, también conocida como búsqueda de medio intervalo, búsqueda logarítmica o corte binario, es un algoritmo de búsqueda que encuentra la posición de un valor objetivo dentro de una matriz ordenada. La búsqueda binaria compara el valor objetivo con el elemento central de la matriz; si no son iguales, se elimina la mitad en la que no puede estar el objetivo y la búsqueda continúa en la mitad restante hasta que tenga éxito. Si la búsqueda termina con la mitad restante vacía, el objetivo no está en la matriz.



Complejidad

Este algoritmo es clasificado como $O(log(n))$ debido a la forma en que el algoritmo divide el espacio de búsqueda a la mitad con cada iteración.



Mejor caso: O(1)

En el mejor caso, el elemento de destino se encuentra en el punto medio de la matriz y lo encontramos en una sola iteración. Esto ocurre cuando la matriz de entrada tiene solo un elemento o cuando el elemento de destino está exactamente en el medio de la matriz. En este escenario, la complejidad temporal es O(1), lo que significa que el algoritmo tarda un tiempo constante en completarse.

Peor caso: O(log(n))

En el peor caso, el elemento de destino no está presente en la matriz y necesitamos iterar por toda la matriz para determinarlo. Esto ocurre cuando la matriz de entrada está vacía o

cuando el elemento de destino no está presente en la matriz. En este escenario, la complejidad temporal es O(log(n)), lo que significa que el algoritmo tarda un tiempo logarítmico en completarse.

Caso promedio: O(log(n))

En el caso promedio, el elemento de destino está presente en la matriz, pero no necesariamente en el punto medio. Necesitamos iterar a través de la matriz para encontrar el elemento de destino, y la cantidad de iteraciones crece logarítmicamente con el tamaño de la matriz de entrada. En este escenario, la complejidad temporal también es O(log(n)), lo que significa que el algoritmo tarda un tiempo logarítmico en completarse.

Análisis paso a paso

Desglosamos el proceso de búsqueda binaria:

1. **Espacio de búsqueda inicial:** comenzamos con una matriz ordenada de n elementos.
2. **Primera iteración:** dividimos el espacio de búsqueda a la mitad, reduciendo efectivamente la cantidad de elementos a buscar a la mitad, es decir, n/2.
3. **Segunda iteración**: dividimos el espacio de búsqueda restante a la mitad nuevamente, reduciendo la cantidad de elementos a buscar a la mitad, es decir, (n/2)/2 = n/4.
4. **Tercera iteración:** dividimos el espacio de búsqueda restante a la mitad nuevamente, reduciendo la cantidad de elementos a buscar a la mitad, es decir, (n/4)/2 = n/8.
5. **...y así sucesivamente**: continuamos dividiendo el espacio de búsqueda a la mitad hasta que encontramos el elemento objetivo o hasta que el espacio de búsqueda está vacío.

Algoritmo de búsqueda binaria en pseudocódigo

```
PROCEDURE BinarySearch(arr, target)
  low = 0
  high = arr.length - 1

  WHILE low <= high
    mid = (low + high) / 2

    IF arr[mid] == target
      RETURN mid // objetivo encontrado, devolver índice

    ELSE IF arr[mid] < target
      low = mid + 1 // buscar en la mitad derecha

    ELSE
      high = mid - 1 // buscar en la mitad izquierda

  END WHILE

  RETURN -1 // objetivo no encontrado
END PROCEDURE
```

Explicación

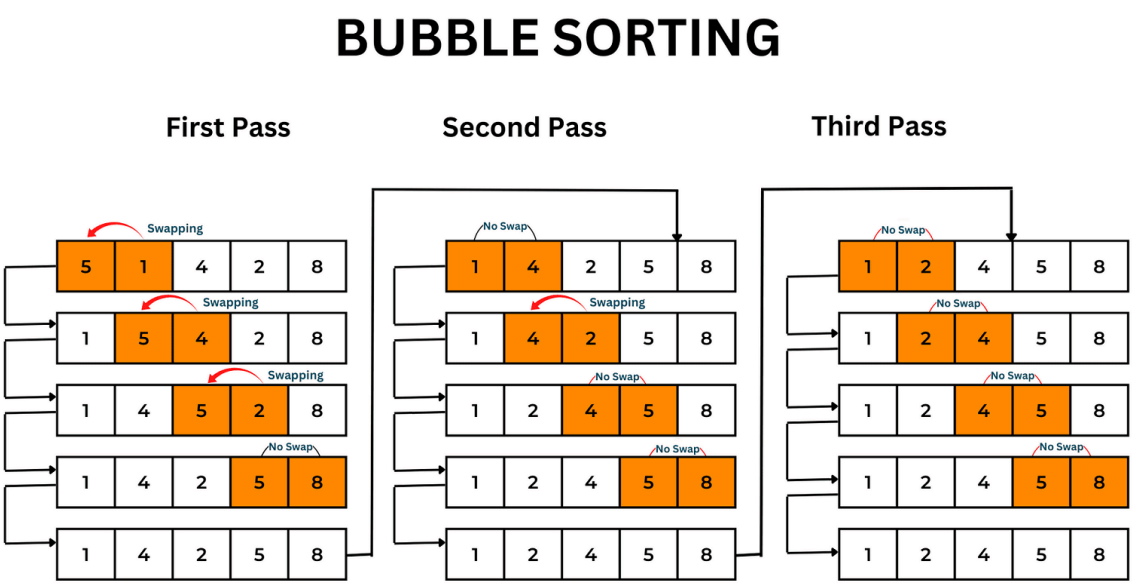
- **arr** es la matriz de entrada, que se supone que está ordenada en orden ascendente.
- **target** es el elemento que estamos buscando.
- **low y high** representan el rango de búsqueda actual, inicialmente establecido en toda la matriz.
- **mid** es el punto medio del rango de búsqueda actual.
- El algoritmo divide iterativamente el rango de búsqueda a la mitad hasta que encuentra el elemento de destino o determina que no está presente.
- Si se encuentra el elemento de destino, el algoritmo devuelve su índice. De lo contrario, devuelve -1 para indicar que el elemento no está presente.

Ejemplo con código según python:

```python
def busqueda_binaria(lista, objetivo):
    low = 0
    high = len(lista) - 1

    while low <= high:
        mid = (low + high) // 2
        guess = lista[mid]

        if guess == objetivo:
            return mid
        elif guess > objetivo:
            high = mid - 1
        else:
            low = mid + 1
    return None
```

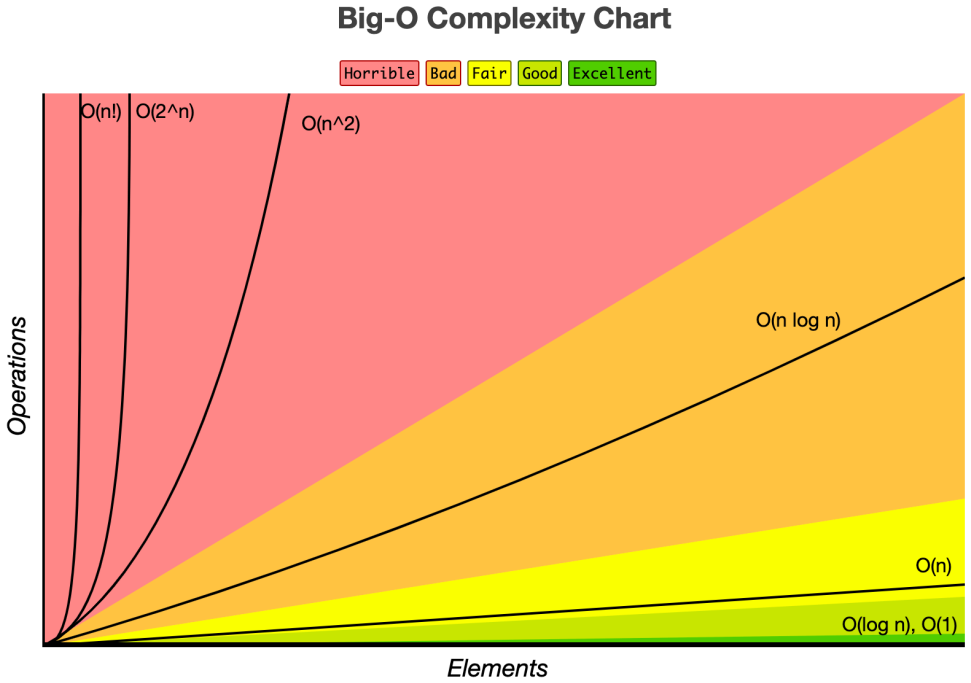
Algoritmo de ordenamiento de burbuja.

El ordenamiento de burbuja, a veces denominado ordenamiento descendente, es un algoritmo de ordenamiento simple que recorre repetidamente la lista que se va a ordenar, compara cada par de elementos adyacentes y los intercambia si están en el orden incorrecto (ordenamiento ascendente o descendente). El recorrido por la lista se repite hasta que no se necesiten intercambios, lo que indica que la lista está ordenada.



Complejidad

La complejidad temporal del algoritmo Bubble Sort es O(n^2) en el peor de los casos y en el promedio, donde n es el número de elementos que se ordenan.



Mejor caso:

Complejidad temporal: O(n)
Descripción: La entrada ya está ordenada en orden ascendente.
Ejemplo: Si la matriz de entrada es [1, 2, 3, 4, 5], Bubble Sort solo necesitará iterar a través de la matriz una vez para confirmar que está ordenada.

Peor caso:

Complejidad temporal: O(n^2)
Descripción: La entrada está ordenada en orden inverso (orden descendente).
Ejemplo: Si la matriz de entrada es [5, 4, 3, 2, 1], Bubble Sort deberá realizar la cantidad máxima de comparaciones, lo que genera una complejidad temporal cuadrática.

Caso promedio:

Complejidad temporal: O(n^2)
Descripción: La entrada está ordenada aleatoriamente.
Ejemplo: Si la matriz de entrada es [3, 1, 4, 2, 5], Bubble Sort aún realizará una cantidad cuadrática de comparaciones, lo que genera una complejidad temporal promedio de O(n^2).

Análisis paso a paso

Paso 1: inicializar la matriz

- Comenzamos con una matriz no ordenada de n elementos: [a[0], a[1], ..., a[n-1]]

Paso 2: primer paso

- Iteramos a través de la matriz desde el primer elemento hasta el segundo elemento desde el último (n-1 iteraciones)
- En cada iteración, comparamos elementos adyacentes y los intercambiamos si están en el orden incorrecto
- Después del primer paso, el elemento más grande "sube" hasta el final de la matriz

Paso 3: segundo paso

- Iteramos a través de la matriz desde el primer elemento hasta el tercero desde el último elemento (n-2 iteraciones)
- En cada iteración, comparamos elementos adyacentes y los intercambiamos si están en el orden incorrecto
- Después del segundo paso, el segundo elemento más grande "sube" hasta la segunda posición desde el último paso

Paso 4: repetir pasos

- Continuamos iterando a través de la matriz, reduciendo el rango de elementos a comparar en 1 en cada paso
- En cada paso, comparamos elementos adyacentes y los intercambiamos si están en el orden incorrecto.
- Repetimos este proceso hasta que se ordena toda la matriz.

Paso 5: Terminación

- El algoritmo termina cuando no se necesitan más intercambios, lo que indica que la matriz está ordenada.

Algoritmo de ordenamiento de burbuja en pseudocodigo

```
Procedure BubbleSort(lista)
  n = longitud(lista)
  For i = 0 To n-2
    For j = 0 To n-i-2
      If lista[j] > lista[j+1]
        intercambiar(lista[j], lista[j+1])
    End for
  End fort
End procedure
```

Explicación

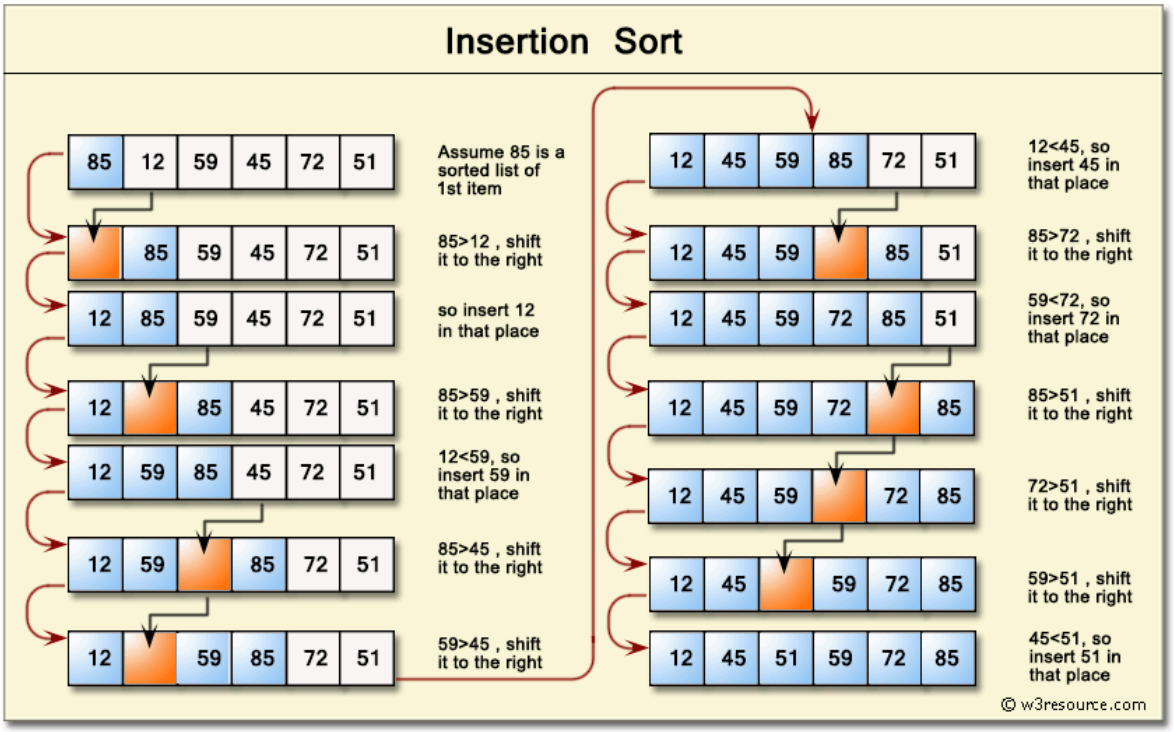
1. El procedimiento **BubbleSort** toma una lista como entrada y ordena sus elementos en orden ascendente.
2. La variable **n** almacena la longitud de la lista.
3. El primer bucle PARA itera desde **0** hasta **n-2**, lo que significa que se realizarán **n-1** pasadas a través de la lista.
4. El segundo bucle PARA itera desde **0** hasta **n-i-2**, lo que significa que se compararán los elementos de la lista en cada pasada.
5. La condición SI **lista[j] > lista[j+1]** verifica si el elemento actual es mayor que el siguiente. Si es así, se intercambian los elementos.
6. El procedimiento **intercambiar** intercambia los valores de dos elementos en la lista.
7. Después de cada pasada, el elemento más grande "burbujea" hacia el final de la lista.
8. El algoritmo termina cuando no se necesitan más intercambios, lo que indica que la lista está ordenada.

Ejemplo con código según Kotlin:

```kotlin
fun bubbleSort(arr: IntArray) {
    val n = arr.size
    for (i in 0 until n - 1) {
        for ( j in 0 until n - i - 1) {
            if (arr[j] > arr[j + 1]) {
                // Swap arr[j] and arr[j + 1]
                val temp = arr[j]
                arr[j] = arr[j + 1]
                arr[j + 1] = temp
            }
        }
    }
}
```

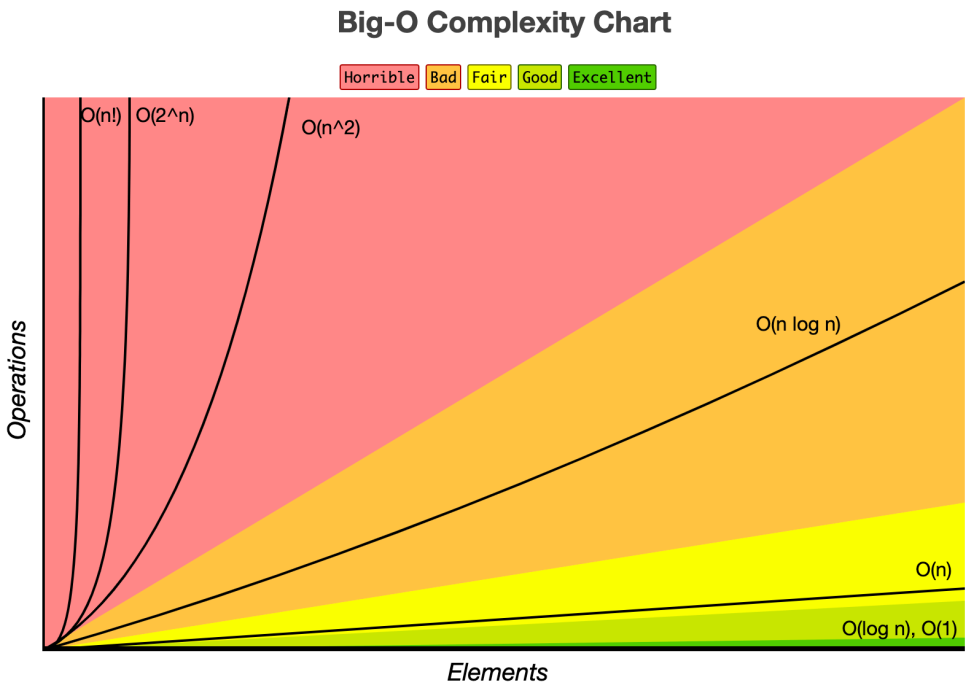
Algoritmo de ordenamiento por inserción.

La ordenación por inserción es un algoritmo de ordenación simple que crea la matriz (o lista) ordenada final elemento por elemento. Es mucho menos eficiente en listas grandes que algoritmos más avanzados como la ordenación rápida, la ordenación por montículo o la ordenación por combinación.



Complejidad:

El algoritmo es clasificado como O(n^2), lo que la hace menos eficiente que otros algoritmos de ordenación como quicksort o mergesort para matrices de entrada grandes.



Mejor caso: O(n)

En el mejor caso, la matriz de entrada ya está ordenada y solo necesitamos iterar una vez para confirmarlo. En este escenario, la complejidad temporal es O(n), lo que significa que el algoritmo tarda un tiempo lineal en completarse.

Peor caso: O(n^2)

En el peor caso, la matriz de entrada está en orden inverso y necesitamos iterar varias veces para ordenarla. En este escenario, la complejidad temporal es O(n^2), lo que significa que el algoritmo tarda un tiempo cuadrático en completarse.

Caso promedio: O(n^2)

En el caso promedio, la matriz de entrada está ordenada aleatoriamente y necesitamos iterar varias veces para ordenarla. En este escenario, la complejidad temporal también es O(n^2), lo que significa que el algoritmo tarda un tiempo cuadrático en completarse.

Análisis paso a paso

Desglosamos el proceso de ordenamiento por inserción:

Espacio de ordenamiento inicial: comenzamos con una matriz desordenada de n elementos.

1. **Primera iteración:** tomamos el segundo elemento (índice 1) y lo insertamos en la posición correcta en la parte ordenada de la matriz.
    a. Seleccionamos el elemento actual como la "clave" que queremos insertar en la parte ordenada de la matriz.
    b. Comparamos la clave con los elementos anteriores y los desplazamos hacia la derecha hasta encontrar la posición correcta para la clave.
    c. Insertamos la clave en la posición correcta.
2. **Segunda iteración:** tomamos el tercer elemento (índice 2) y lo insertamos en la posición correcta en la parte ordenada de la matriz.
    a. Seleccionamos el elemento actual como la "clave" que queremos insertar en la parte ordenada de la matriz.
    b. Comparamos la clave con los elementos anteriores y los desplazamos hacia la derecha hasta encontrar la posición correcta para la clave.
    c. Insertamos la clave en la posición correcta.
3. **Tercera iteración:** tomamos el cuarto elemento (índice 3) y lo insertamos en la posición correcta en la parte ordenada de la matriz.
    a. Seleccionamos el elemento actual como la "clave" que queremos insertar en la parte ordenada de la matriz.
    b. Comparamos la clave con los elementos anteriores y los desplazamos hacia la derecha hasta encontrar la posición correcta para la clave.
    c. Insertamos la clave en la posición correcta.
4. **...y así sucesivamente:** continuamos insertando cada elemento en la posición correcta en la parte ordenada de la matriz hasta que toda la matriz esté ordenada.

Algoritmo de ordenamiento por inserción en pseudocodigo

```
Procedure InsertionSort(arr)
  for i from 1 to length(arr) - 1
    key = arr[i]
    j = i - 1
    while j >= 0 and arr[j] > key
      arr[j + 1] = arr[j]
      j = j - 1
    arr[j + 1] = key
  end for
End Procedure
```

Explicación:

- El bucle externo itera desde el segundo elemento hasta el último elemento de la matriz.
- Para cada elemento, lo consideramos como una "clave" y lo comparamos con los elementos anteriores.
- Desplazamos los elementos mayores que la clave una posición hacia la derecha hasta que encontramos la posición correcta para la clave.
- Insertamos la clave en la posición correcta.

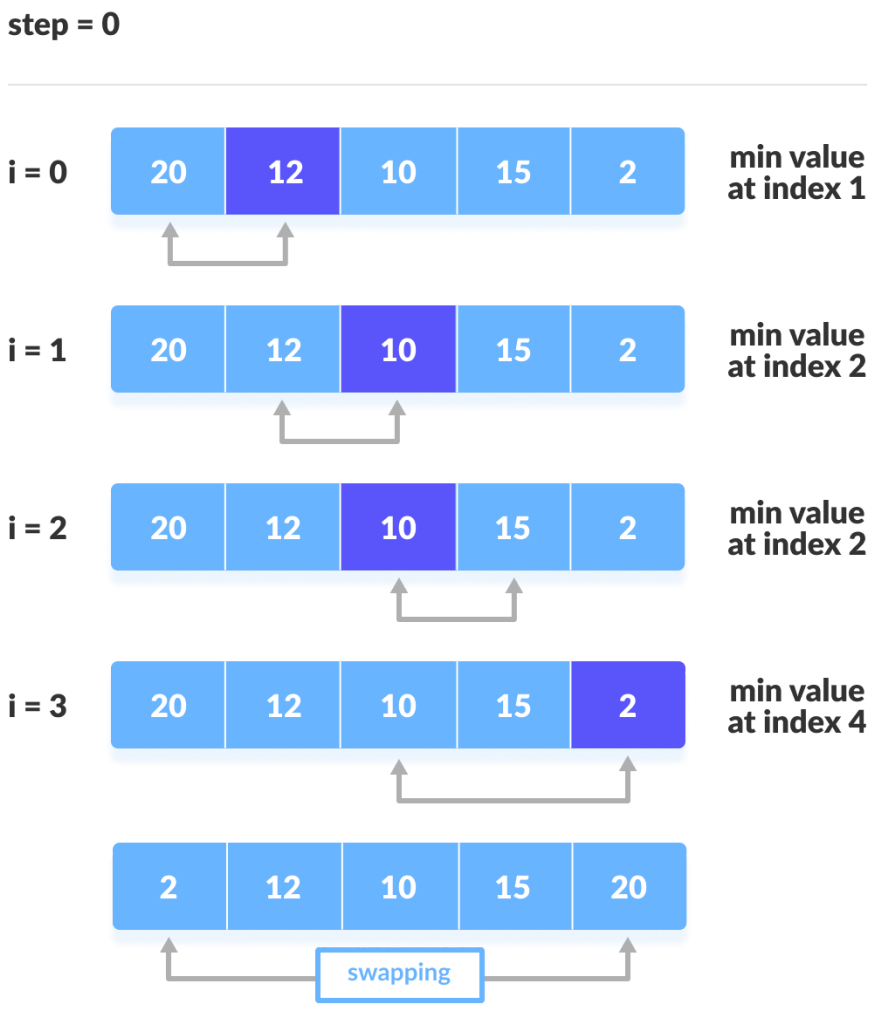
Ejemplo con código según Java:

```java
public class InsertionSort {
    public static void sort(int[] array) {
        for (int i = 1; i < array.length; i++) {
            int key = array[i];
            int j = i - 1;

            while (j >= 0 && array[j] > key) {
                array[j + 1] = array[j];
                j--;
            }

            array[j + 1] = key;
        }
    }

    public static void main(String[] args) {
        int[] array = {5, 2, 8, 3, 1, 6, 4};
        sort(array);

        for (int i : array) {
            System.out.print(i + " ");
        }
    }
}
```

Algoritmo de ordenamiento por selección

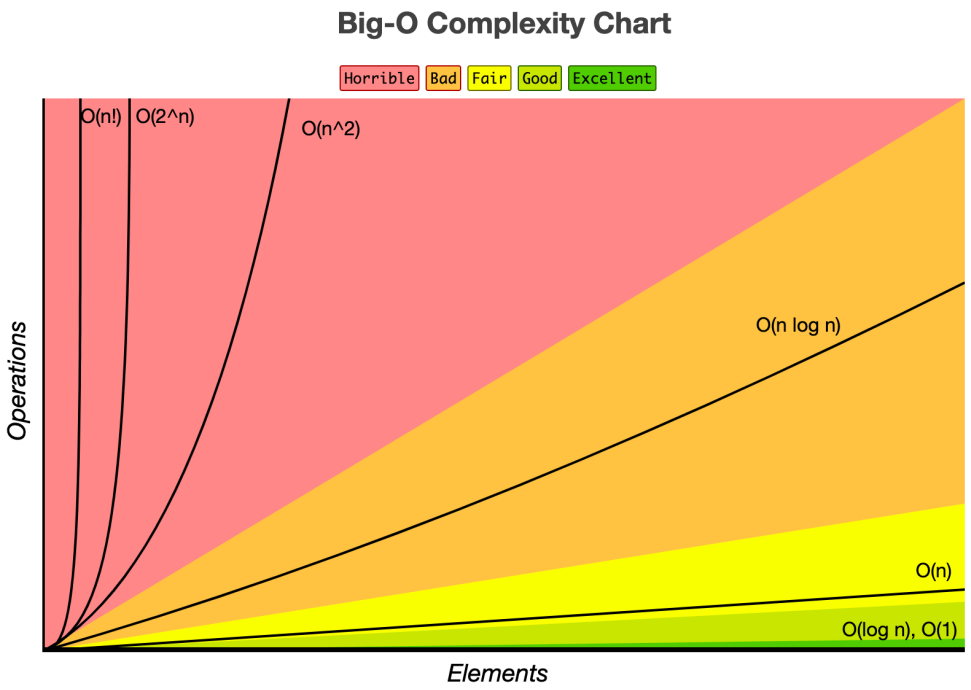
La ordenación por selección es un algoritmo de ordenación, específicamente una ordenación por comparación en el lugar. Tiene una complejidad temporal de O(n2), lo que la hace ineficiente en listas grandes y, en general, tiene un peor rendimiento que la ordenación por inserción similar. La ordenación por selección se destaca por su simplicidad y tiene

ventajas de rendimiento sobre algoritmos más complicados en ciertas situaciones, en particular cuando la memoria auxiliar es limitada.



Complejidad

Este algoritmo es clasificado como O(n^2), donde 'n' es el número de elementos de la matriz que se está ordenando.
algoritmo desglosado paso a paso de la complejidad temporal del algoritmo:



Mejor caso: O(n^2)

En el mejor caso, la matriz de entrada ya está ordenada en orden ascendente. Sin embargo, el algoritmo de ordenamiento por selección aún realiza la misma cantidad de comparaciones, iterando a través de la matriz n veces para encontrar el elemento mínimo. Dado que el algoritmo no aprovecha la entrada ya ordenada, la complejidad temporal sigue siendo O(n^2).

Peor caso: O(n^2)

En el peor caso, la matriz de entrada está ordenada en orden descendente. El algoritmo de ordenamiento por selección aún realiza la misma cantidad de comparaciones, iterando a través de la matriz n veces para encontrar el elemento mínimo. La complejidad temporal sigue siendo O(n^2) porque el algoritmo no aprovecha la entrada ordenada en orden inverso.

Caso promedio: O(n^2)

En el caso promedio, la matriz de entrada está ordenada aleatoriamente. El algoritmo de ordenamiento por selección aún realiza la misma cantidad de comparaciones, iterando a través de la matriz n veces para encontrar el elemento mínimo. La complejidad temporal sigue siendo O(n^2) porque el algoritmo no aprovecha ningún patrón en los datos de entrada.

Análisis paso a paso

- El algoritmo itera a través de la matriz 'n' veces para encontrar el elemento mínimo.
- En cada iteración, compara el elemento actual con los elementos restantes, lo que lleva O(n) tiempo.

- Dado que este proceso se repite 'n' veces, la complejidad temporal total se convierte en O(n) × O(n) = O(n^2).

Por lo tanto, el algoritmo de ordenamiento por selección tiene una complejidad temporal cuadrática, lo que lo hace menos eficiente para conjuntos de datos grandes. Sin embargo, es fácil de implementar y puede ser útil para conjuntos de datos pequeños o casos de uso específicos.

Algoritmo de ordenamiento por selección en pseudocodigo

```
procedure ordenamientoPorSeleccion(lista)
  for i from 0 to longitud(lista) - 1
    minimo = i
    for j from i + 1 to longitud(lista) - 1
      if lista[j] < lista[minimo]
        minimo = j
    end for
    change lista[i] and lista[minimo]
  end for
end procedure
```

Explicación

- El procedimiento ordenamientoPorSeleccion toma una lista como entrada y la ordena en lugar.
- El bucle exterior (PARA i DESDE 0 HASTA longitud(lista) - 1) itera sobre cada elemento de la lista.
- El bucle interior (PARA j DESDE i + 1 HASTA longitud(lista) - 1) busca el mínimo elemento en la parte no ordenada de la lista.
- Si se encuentra un elemento menor que el actual mínimo, se actualiza el índice del mínimo (minimo = j).
- Después de encontrar el mínimo, se intercambia con el elemento actual (INTERCAMBIAR lista[i] y lista[minimo]).
- El proceso se repite hasta que toda la lista esté ordenada.

Ejemplo con código según c#:

```
public class SelectionSort {
    public static void Sort(int[] array) {
        int n = array.Length;

        for (int i = 0; i < n - 1; i++) {
            int minIndex = i;

            for (int j = i + 1; j < n; j++) {
                if (array[j] < array[minIndex]) {
                    minIndex = j;
                }
            }

            int temp = array[minIndex];
            array[minIndex] = array[i];
            array[i] = temp;
        }
    }

    public static void Main(string[] args) {
        int[] array = { 5, 2, 8, 3, 1, 6, 4 };
        Sort(array);

        foreach (int i in array) {
            Console.Write(i + " ");
        }
    }
}
```

Conclusión

En este proyecto, hemos implementado y comparado cinco algoritmos de ordenamiento populares: Bubble Sort, Selection Sort, Insertion Sort, Merge Sort y Quick Sort. A través de una serie de pruebas y análisis, hemos evaluado la eficiencia y rendimiento de cada algoritmo en diferentes escenarios.

Nuestros resultados muestran que cada algoritmo tiene sus propias fortalezas y debilidades, y que la elección del algoritmo adecuado depende del tamaño y tipo de datos, así como de las restricciones de tiempo y espacio. En general, los algoritmos de ordenamiento más eficientes en términos de tiempo y espacio fueron Merge Sort y Quick Sort, mientras que Bubble Sort y Selection Sort fueron los menos eficientes.

Sin embargo, es importante destacar que la complejidad de cada algoritmo también depende del tipo de datos y del tamaño de la entrada. Por ejemplo, Insertion Sort puede ser más eficiente que Merge Sort para pequeños conjuntos de datos, mientras que Quick Sort puede ser más eficiente que Merge Sort para conjuntos de datos muy grandes.

En resumen, este proyecto ha demostrado la importancia de evaluar cuidadosamente la complejidad y el rendimiento de los algoritmos de ordenamiento en diferentes escenarios, y ha proporcionado una guía para la selección del algoritmo adecuado para diferentes aplicaciones.

Recomendaciones

- Para conjuntos de datos pequeños, Insertion Sort puede ser una buena opción debido a su simplicidad y eficiencia.
- Para conjuntos de datos medianos, Merge Sort puede ser una buena opción debido a su estabilidad y eficiencia.
- Para conjuntos de datos muy grandes, Quick Sort puede ser una buena opción debido a su velocidad y eficiencia.
- En general, es importante considerar la complejidad y el rendimiento de cada algoritmo en diferentes escenarios antes de tomar una decisión.

Biografias

Trekhleb. (s. f.). *javascript-algorithms/README.es-ES.md at master · trekhleb/javascript-algorithms*. GitHub. https://github.com/trekhleb/javascript-algorithms/blob/master/README.es-ES.md