Universidad ICESI

Algoritmos y Programación II

Ordenamiento

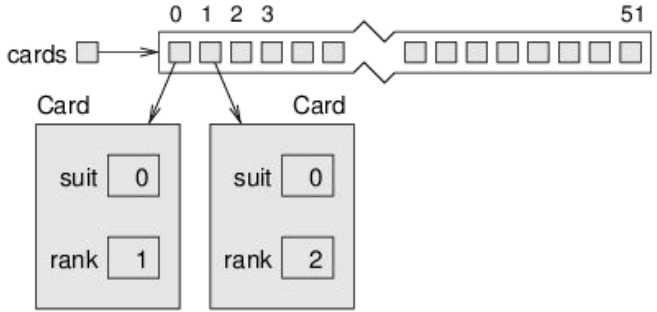
Esta presentación proporciona una exploración al concepto de ordenamiento en estructuras de datos. Cubre los conceptos básicos, algoritmos de ordenamiento y tipos de ordenamiento por primitivos y objetos.

¿Qué es un algoritmo de ordenamiento?

Un algoritmo de ordenamiento se utiliza para re organizar una estructura de datos determinada según un criterio de comparación de los elementos. El criterio de comparación se utiliza para decidir el nuevo orden de los elementos en la estructura de datos respectiva.

Primitivos

Objetos (no primitivos)



¿Para que los usamos?

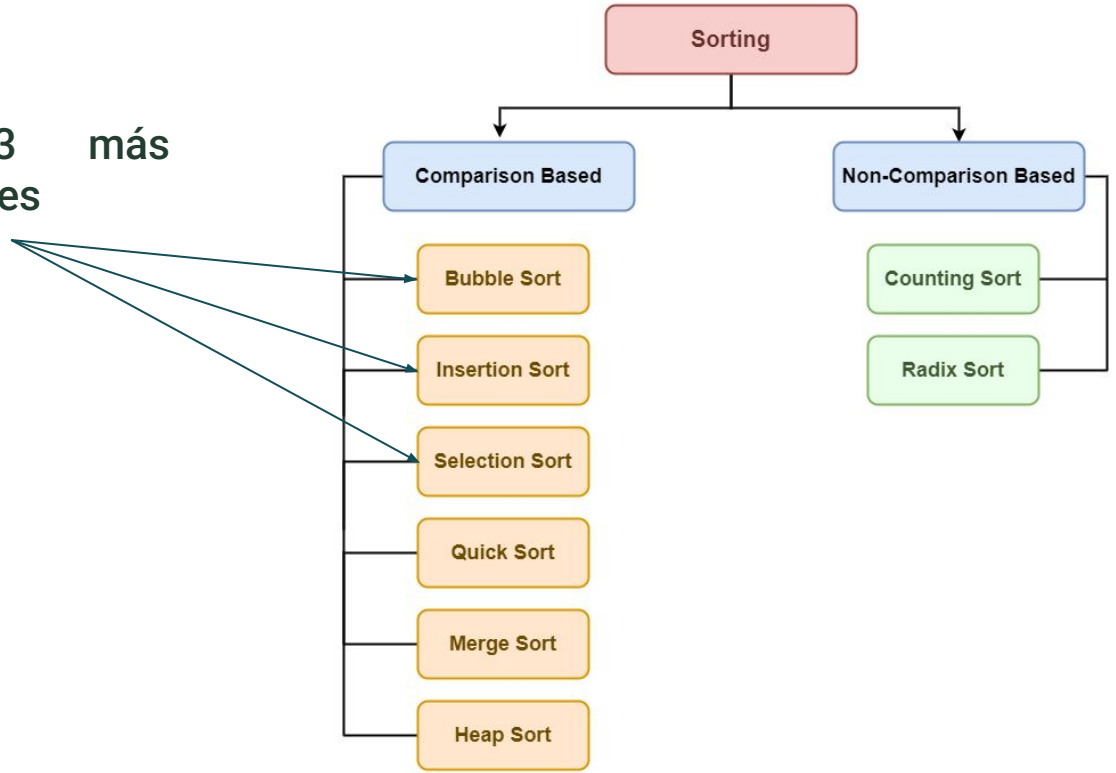
Cuando tenemos una gran cantidad de datos, puede resultar difícil manejarlos, especialmente cuando están ordenados de forma aleatoria. Cuando esto sucede, ordenar los datos se vuelve crucial. Es necesario ordenar los datos para facilitar su manipulación.

Hay varios algoritmos de ordenamiento que se utilizan en estructuras de datos, estos se pueden clasificar en dos grupos:

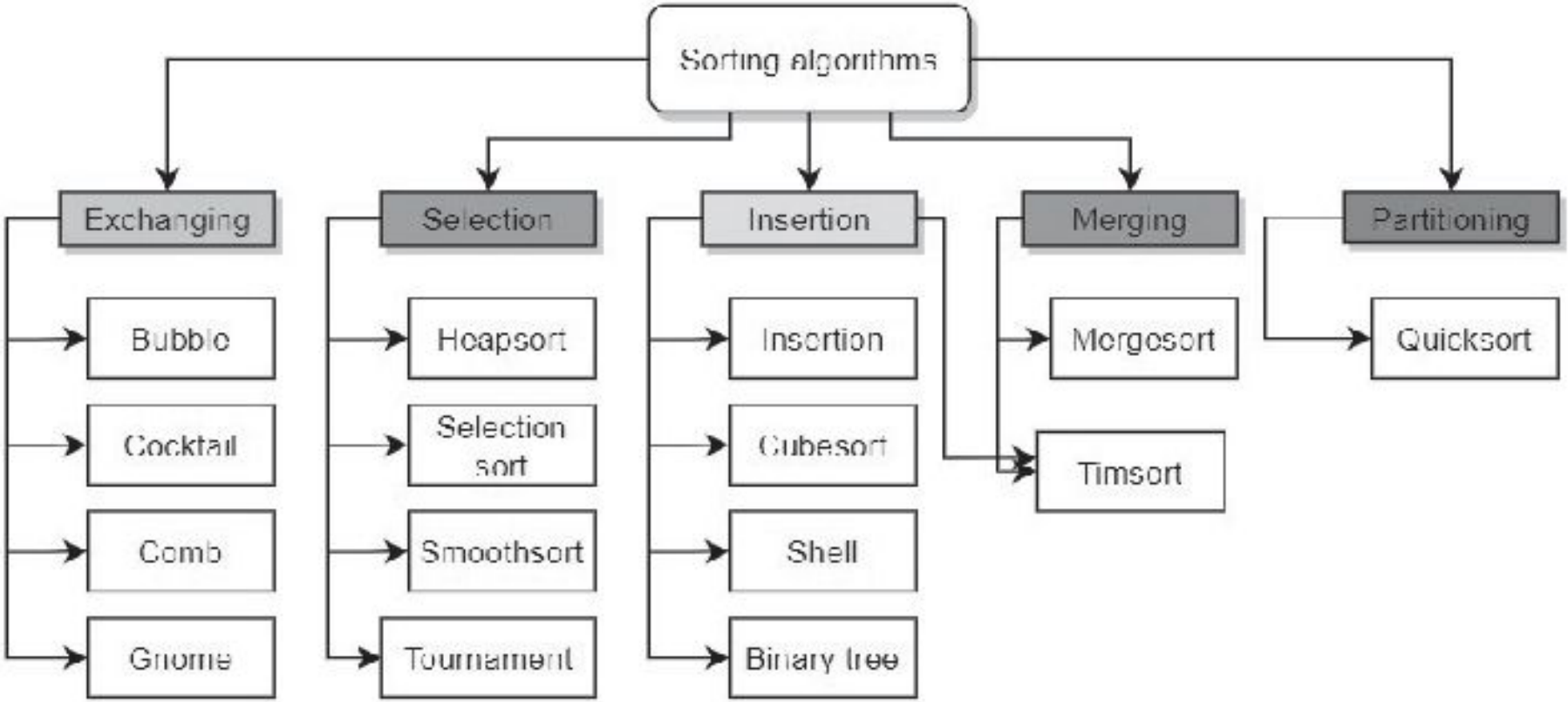
- Basado en comparación: comparamos los elementos
- No basado en comparación: no comparamos los elementos

Según su criterio de comparación

Los 3 más populares



Según su estrategia

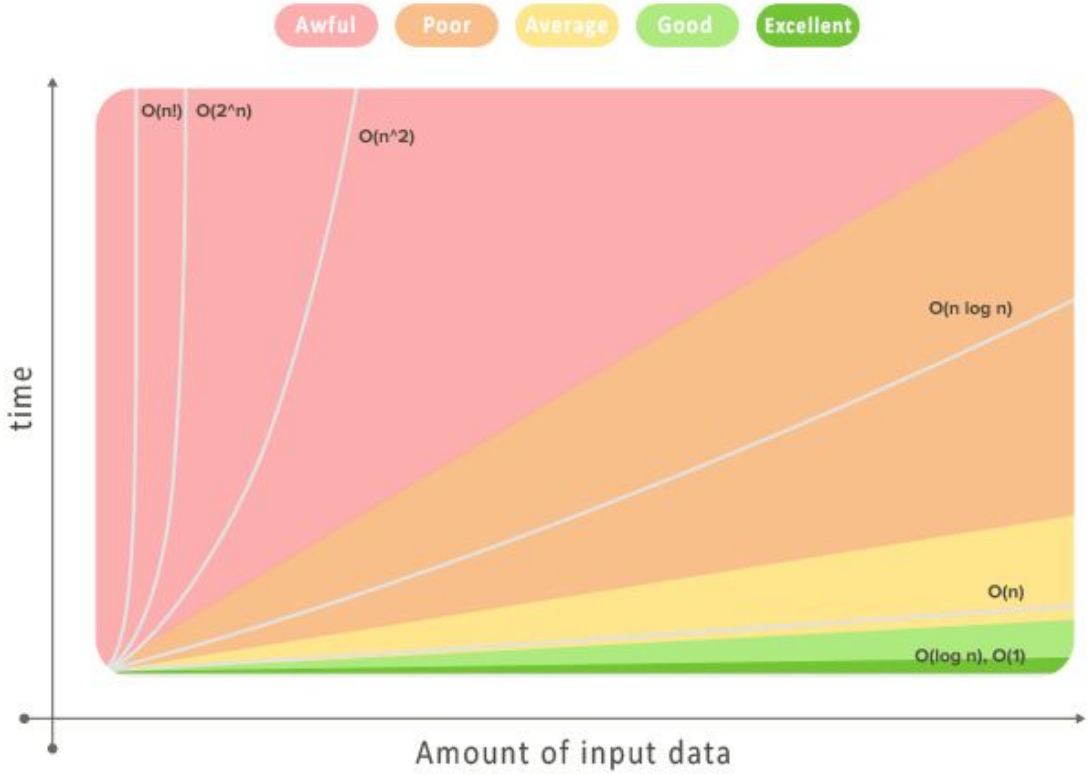


Comparación por complejidad temporal y espacial

Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	θ(n log(n))	O(n log(n))	O(n)

Graph of prevalence of algorithm complexity



Ordenamiento de Primitivos

Algoritmo de ordenamiento 1: Selección

En el ordenamiento por selección encontramos el elemento mínimo en cada iteración y lo colocamos en la estructura comenzando desde el primer índice.

Supongamos que A es una estructura de N valores. Queremos ordenar A en orden ascendente. Es decir, A[0] debería ser el más pequeño y A[N-1] debería ser el más grande.

La idea de ordenar por selección es que encontramos repetidamente el elemento más pequeño en la parte no ordenada de la matriz y lo intercambiamos con el primer elemento en la parte no ordenada de la matriz.

```
For i = 0 to n-1 do:
  minimum = i
  For j = i + 1 to n do:
    If A(j) < A(minimum)
      minimum = j
    End-If
  End-For
  temp = A(minimum)
  A(minimum) = A(i)
  A(i) = temp
End-For
```

Algoritmo de ordenamiento 2: Inserción

El ordenamiento por inserción funciona de manera similar a la forma en que clasificamos las cartas en las manos.

Supongamos que A es una estructura de N valores. Queremos ordenar A en orden ascendente.

La idea de la inserción es recorrer la estructura e insertar cada elemento en la parte ordenada de la lista a la que pertenece. Esto suele implicar empujar hacia al “fondo” los elementos más grandes de la estructura.

```
For i = 1 to n
 int temp = A(i)
 j = i - 1
 While (j >= 0) and (A(j) > temp)
  A(j+1) = A(j)
  j–
 End-While
A(j+1) = temp
End-For
```

6 5 3 1 8 7 2 4

Algoritmo de ordenamiento 3: Burbuja

El ordenamiento por burbuja es el más simple que funciona intercambiando repetidamente los elementos adyacentes si están en el orden incorrecto.

Supongamos que A es una estructura de N valores. Queremos ordenar A en orden ascendente.

La idea de burbuja es mirar la lista y dondequiera que encontremos dos elementos consecutivos desordenados, los intercambiamos. El nombre del algoritmo se refiere al hecho de que el elemento más grande "se hunde" hacia el fondo y los elementos más pequeños "flotan" hacia la parte superior.

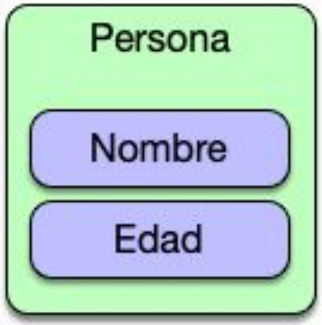
```
For i = 0 to n - 1
 For j = 0 to n - i - 1
  If (A(j) > A(j + 1)
    Temp = A(j)
    A(j) = A(j + 1)
    A(j + 1) = Temp
  End-If
 End-For
End-For
```

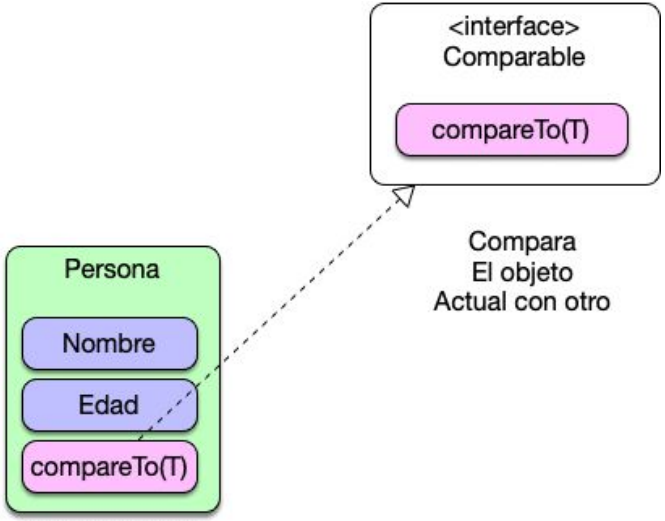
6 5 3 1 8 7 2 4

Ordenamiento de No Primitivos

La Interface Comparable

la interface Comparable permite comparar objetos entre sí para poderlos ordenar. Con esto podemos ordenar estructuras de datos que almacenan datos no primitivos!!!





https://www.arquitecturajava.com/java-comparable-interface-y-ordenaciones/

El método compareTo

public int compareTo (Object obj): se utiliza para comparar el objeto actual con el objeto especificado. devuelve:

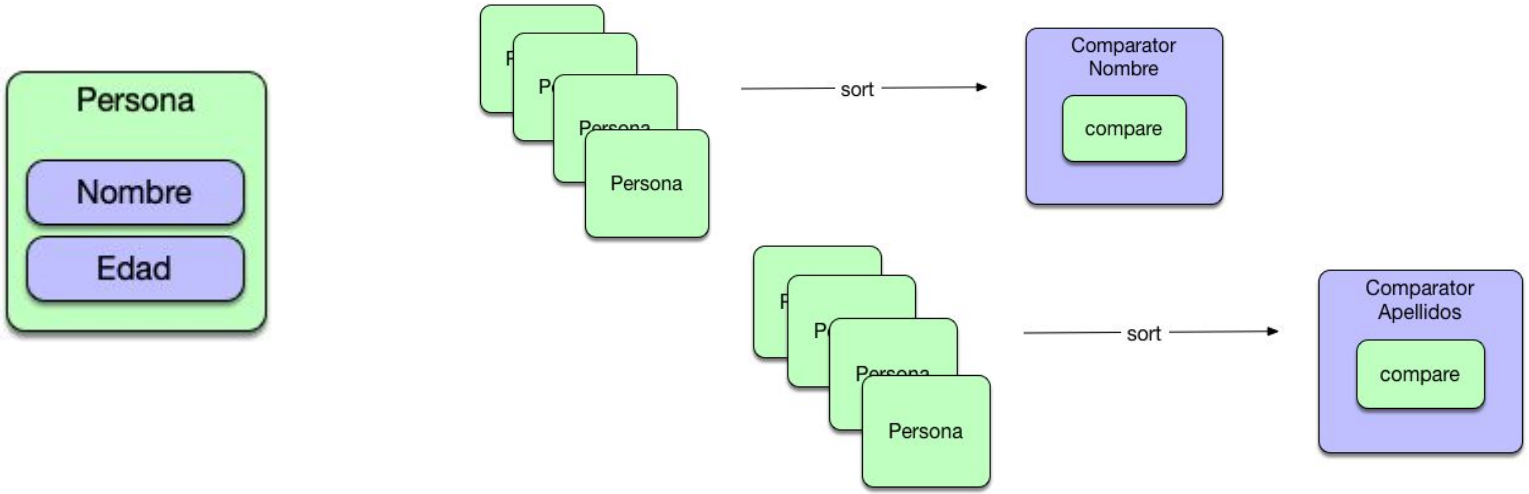
- Entero positivo (1), si el objeto actual es mayor que el objeto especificado.
- entero negativo (-1), si el objeto actual es menor que el objeto especificado.
- cero, si el objeto actual es igual al objeto especificado.

Un objeto comparable es capaz de compararse con otro objeto. La propia clase debe implementar la interface *java.lang.Comparable* para comparar sus instancias.

Se sobrescribe el método *compareTo()* para definir el criterio de ordenación natural, un ejemplo, orden alfabético por nombre.

La Interface Comparator

A diferencia de Comparable, Comparator es externo al tipo de elemento que estamos comparando. Es una clase separada. Creamos múltiples clases separadas (que implementan Comparador) para comparar entre diferentes miembros.



https://www.arquitecturajava.com/java-comparator-interface-y-lambdas/

El método compare

public int compare(Object obj1, Object obj2): se utiliza para comparar dos objetos. devuelve:

- Entero positivo (1), si el objeto obj1 es mayor que el obj2.
- entero negativo (-1), si el objeto obj1 es menor que el obj2.
- cero, si el objeto obj1 es igual que el obj2.

A diferencia de Comparable, Comparator es externo al tipo de elemento que estamos comparando. Es una clase separada. Creamos múltiples clases separadas (que implementan Comparator) para comparar miembros por diferentes criterios o atributos.

Clases anónimas (Java)

En Java, podemos crear un comparador anónimo.

```
/** Compare strings by length. */
Comparator<String> compareByLength =
          new Comparator<String>()
 { /* definition of anonymous class */
   public int Compare(String a, String b) {
        return a.length() - b.length();
   }
 };
Arrays.sort( strings, compareByLength );
```

Clases anónimas (Java)

```
13 public class StudentCompare{
14     public static void main(String args[]) {
15
16         ArrayList<Student> studList = new ArrayList<Student>();
17
18         studList.add(new Student("Gouthami", 90.61f));
19         studList.add(new Student("Raja", 83.55f));
20         studList.add(new Student("Honey", 85.55f));
21         studList.add(new Student("Teja", 77.56f));
22         studList.add(new Student("Varshith", 80.89f));
23
24         Comparator<Student> com = new Comparator<Student>() {
25             public int compare(Student stud1, Student stud2) {
26                 if(stud1.percentage < stud2.percentage)
27                     return 1;
28                 return -1;
29             }
30         };
31
32         Collections.sort(studList, com);
33
34         System.out.println("Avg % --> Name");
35         System.out.println("---------------------");
36         for(Student stud:studList) {
37             System.out.println(stud.percentage + " --> " + stud.name);
38         }
39     }
40 }
```

Clases anónimas (Java)

```
13 public class StudentCompare{
14     public static void main(String args[]) {
15 
16         ArrayList<Student> studList = new ArrayList<Student>();
17 
18         studList.add(new Student("Gouthami", 90.61f));
19         studList.add(new Student("Raja", 83.55f));
20         studList.add(new Student("Honey", 85.55f));
21         studList.add(new Student("Teja", 77.56f));
22         studList.add(new Student("Varshith", 80.89f));
23 
24         Comparator<Student> com = (stud1, stud2) -> {
25                 if(stud1.percentage < stud2.percentage)
26                     return 1;
27                 return -1;
28         };
29 
30         Collections.sort(studList, com);
31 
32         System.out.println("Avg % --> Name");
33         System.out.println("----------------------");
34         for(Student stud:studList) {
35             System.out.println(stud.percentage + " --> " + stud.name);
36         }
37     }
38 }
```

Diferencias

- Comparable está destinado a objetos con ordenamiento natural, lo que significa que el objeto mismo debe saber cómo debe ordenarse. Por ejemplo, números en una lista de estudiantes. Mientras que la clasificación de la interface Comparator se realiza a través de una clase separada.

- Lógicamente, la interfaz Comparable compara la referencia “this” con el objeto especificado y Comparator en Java compara dos objetos de clases diferentes proporcionados.

- Una característica diferenciadora básica es que al utilizar comparables solo podemos utilizar una comparación. Mientras que podemos escribir más de un comparador personalizado según lo desee para un atributo determinado, todos utilizando diferentes interpretaciones de lo que significa ordenar.

HOJA DE TRABAJO

https://docs.google.com/document/d/1ctSJOCZZzJ3zBUg81I7rFEnxSXdUYJJCWIg0EhDjP20/edit?usp=sharing

Gracias por su tiempo y atención 🙂