# Tema 5:

# Algoritmia básica: ordenación y búsqueda

***Objetivos**: en este tema se presentan algoritmos que permiten buscar un elemento dentro de una colección y ordenar una colección en base a algún criterio (el valor de un número, orden alfabético...). Ambas operaciones son empleadas profusamente en computación, de ahí su gran importancia, y se apoyan la una en la otra, por lo que las hemos agrupado bajo un mismo tema.*

# INDICE:

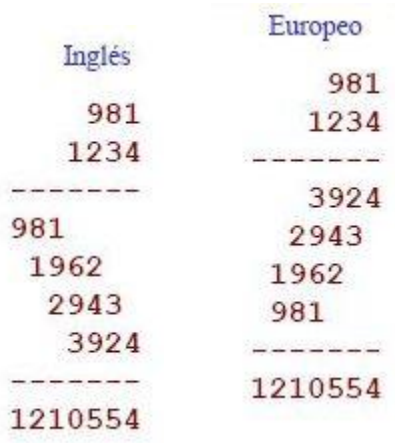
# 1 Introducción

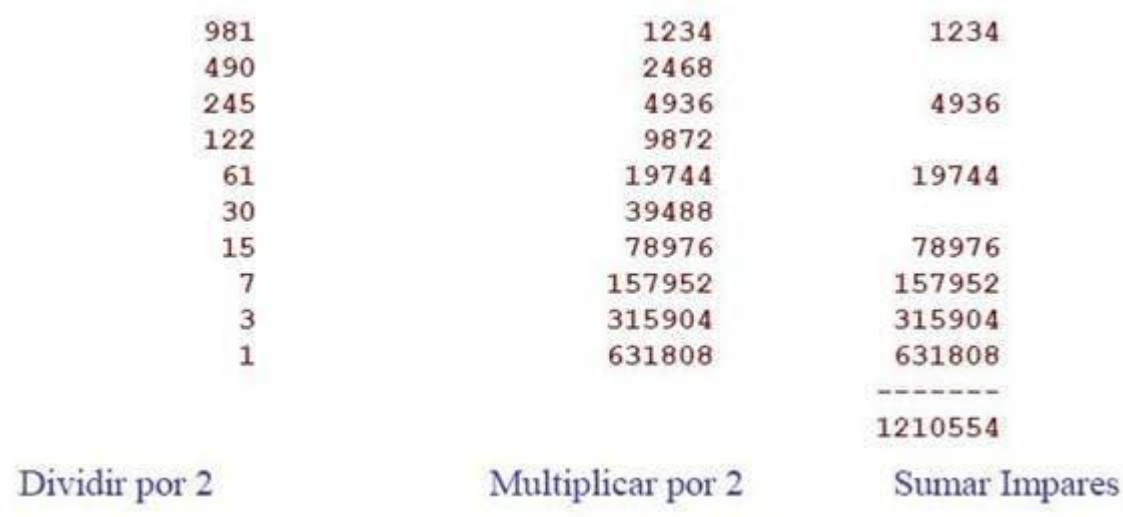
La ordenación es una aplicación fundamental en computación. La mayoría de los datos producidos por un programa están ordenados de alguna manera, y muchos de los cómputos que tiene que realizar un programa son más eficientes si los datos sobre los que operan están ordenados. Uno de los tipos de cómputo que más se benefician de operar sobre un conjunto de datos ordenados es la búsqueda de un dato: encontrar el número de teléfono de una persona en un listín telefónico es una tarea muy simple y rápida si conocemos su nombre, ya que los listines telefónicos se encuentran ordenados alfabéticamente. Encontrar a qué usuario corresponde un número de teléfono dado, sin embargo, es una tarea prácticamente inabordable. Valga esto a modo de ejemplo de cómo el disponer de una colección de datos ordenados simplifica la búsqueda de información entre ellos, de ahí la importancia de las tareas de ordenación y búsqueda y la relación que existe entre ellas.

Un factor clave en cualquier algoritmo de búsqueda u ordenación es su complejidad computacional, y el cómo esta depende del número de datos a procesar. Habitualmente, cuando nos enfrentamos a una tarea de ordenación o búsqueda con un ordenador el volumen de datos de entrada será enorme y es importante contar con algoritmos que no degraden considerablemente su rendimiento con el tamaño del conjunto de datos. Por ello, antes de abordar algoritmos de ordenación y búsqueda realizaremos una pequeña introducción al cálculo de la complejidad computacional de un algoritmo.

# 2 Complejidad computacional y algoritmos

Habitualmente, un mismo problema puede tener numerosas soluciones que tienen distinta eficiencia (rapidez de ejecución). Así, por ejemplo, un problema tan fácil como el multiplicar dos números enteros es resuelto mediante dos algoritmos diferentes en Inglaterra y el resto de Europa. Otra forma mucho más curiosa de realizar la multiplicación de los números enteros es el método ruso; un método más complejo que los anteriores y que requiere realizar un número de operaciones matemáticas bastante superior, aunque obtiene el mismo resultado:

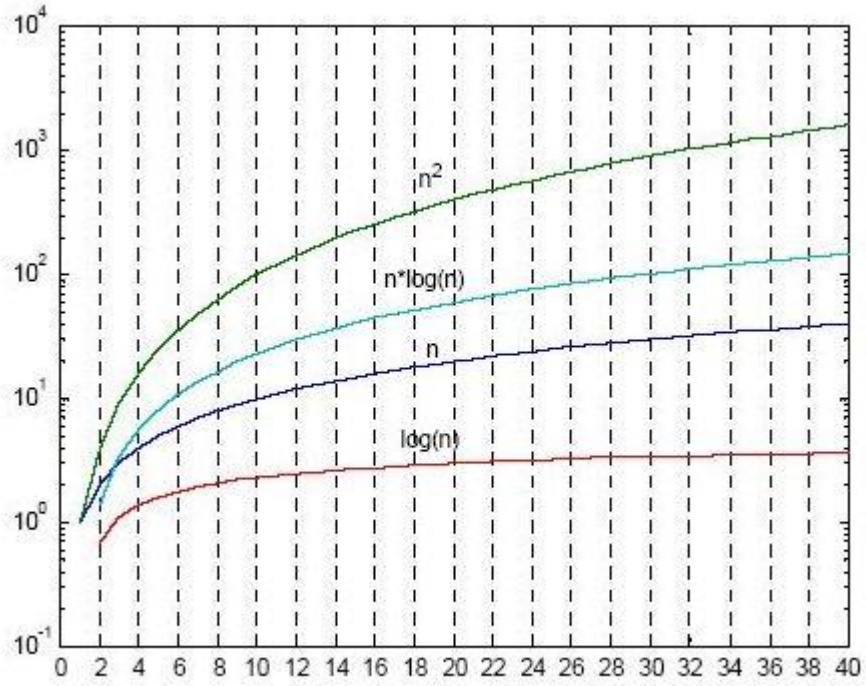


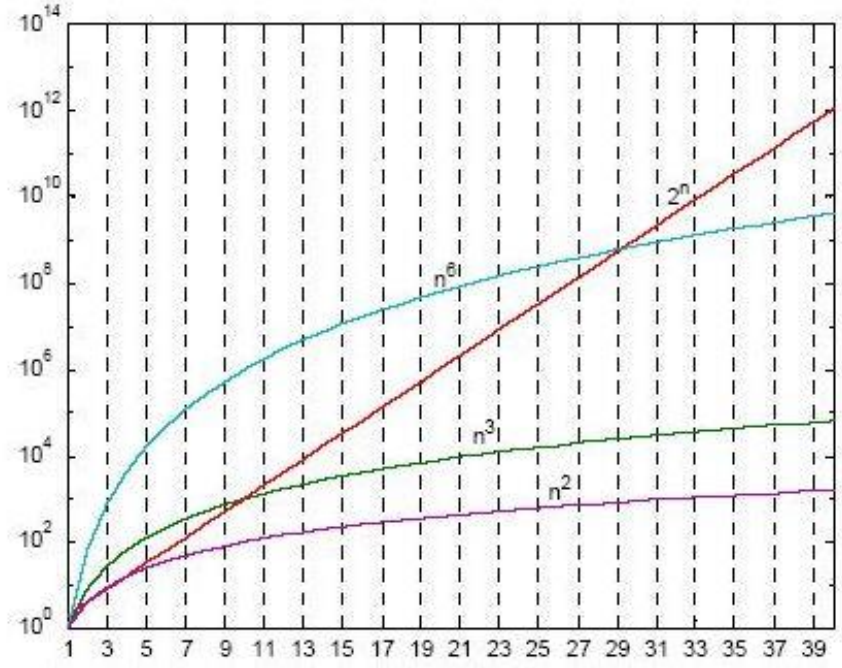



Cuando seleccionamos un algoritmo para resolver un determinado problema es importante determinar los recursos computacionales necesarios (tiempo de computación y espacio de almacenamiento en memoria) en función del volumen de datos a procesar. La eficiencia de un algoritmo no puede medirse en unidades de tiempo, ya que esto obligaría a realizar medidas empíricas sobre una implementación concreta, un compilador concreto y una máquina concreta.

Para estudiar la complejidad computacional de los algoritmos se toma por cierto el principio de invarianza: dos implementaciones distintas del mismo algoritmo no difieren en su eficiencia más que en una constante multiplicativa; esto es, si dos implementaciones del mismo algoritmo necesitan $t_1(n)$ y $t_2(n)$ unidades de tiempo, donde n es el tamaño del vector de entrada, entonces existe un c>0 tal que $t_1(n) < c*t_2(n)$. Este principio permite concluir que un cambio en la máquina donde se ejecuta un algoritmo proporciona una mejora de un factor constante, mientras que las mejoras dependientes del número de datos que procesa el algoritmo deberán venir dadas por cambios en el propio algoritmo.

Hay dependencias en el tamaño n del volumen de datos a procesar muy frecuentes: tiempo logarítmico (c*log (n)), tiempo lineal (c*n), tiempo cuadrático (c*n2), tiempo polinomial ($c*n^k$), y tiempo exponencial ($c^n$). Estas dependencias están ordenadas de menor a mayor, siempre que se consideren valores de n suficientemente grandes.




El cálculo de la eficiencia de un algoritmo se basa en contar el número de operaciones elementales que realiza. Por operación elemental se entiende operación cuyo tiempo de ejecución es constante y depende únicamente de la implementación como, por ejemplo, sumas, restas, productos, divisiones, módulo, operaciones lógicas, operaciones de comparación, etcétera. Estas operaciones no deben depender del volumen de datos manejados por el algoritmo, y sólo nos interesa el número de operaciones, no cuánto consume cada una de ellas. La diferencia en los tiempos de ejecución entre las distintas operaciones quedan incluidos en la constante multiplicativa.

Para representar la eficiencia de un algoritmo suele emplearse la notación "del orden de" (O (...)). Diremos que una función t(n) está en el orden de f(n) si existe una constante c y un umbral $n_0$ tales que:

$$\forall n \geq n_0 \quad t(n) \leq c \cdot f(n)$$

Textualmente esto se representa por "t es del orden de f" (t=O(f)). Esta frase indica que si t es el tiempo de la implementación de un algoritmo podemos considerar que dicho tiempo está en el orden de f. Además, por el principio de invarianza, cualquier implementación de dicho algoritmo también será de orden f. La notación t=O(f) suele usarse aunque f(n) sea menor que cero para algún n, o incluso cuando no esté definida para algún valor de n; lo importante es el comportamiento de f(n) para valores grandes de n.

## *2.1 Complejidad computacional de las estructuras básicas de programación*

En esta sección estudiaremos cuál es la complejidad computacional de las estructuras básicas que permiten construir algoritmos.

### Sentencias sencillas

Nos referimos a las sentencias de asignación, comparación, operaciones lógicas y matemáticas. Este tipo de sentencias tiene un tiempo de ejecución constante que no depende del tamaño del conjunto de datos de entrada, siendo su complejidad O(1).

### Secuencia de sentencias

La complejidad de una serie de elementos de un programa es del orden de la suma de las complejidades individuales; en el caso de una secuencia de sentencias sencillas la complejidad será: O(1)+ O(1)+...+ O(1)= k* O(1)= O(1).

### Selección

La evaluación de la condición suele ser de O(1), complejidad a sumar con la mayor complejidad computacional posible de las distintas ramas de ejecución, bien en la rama IF, o bien en la rama ELSE. En decisiones múltiples (ELSE IF, SWITCH CASE), se tomará la rama cuya complejidad computacional es superior.

### Bucles

Los bucles son la estructura de control de flujo que suele determinar la complejidad computacional del algoritmo, ya que en ellos se realiza un mayor número de operaciones.

En los bucles con contador podemos distinguir dos casos: que el tamaño del conjunto de datos n forme parte de los límites del bucle o que no. Si la condición de salida del bucle es independiente de n entonces la repetición sólo introduce una constante multiplicativa:

```
for (int i= 0; i < K; i++) { O(1)}
```

la complejidad será K*O(1) = O(1).

Cuando el número de iteraciones a realizar depende del tamaño de datos a procesar, la complejidad computacional del bucle incrementará con el tamaño de los datos de entrada:

```
for (int i= 0; i < n; i++) { O(1)}
```

la complejidad será n* O(1)= O(n).

```
for (int i= 0; i < n; i++) {
       for (int j= 0; j < n; j++) {
         O(1)
       }
     }
```

tendremos n * n * O(1) = ($n^2$).

```
for (int i= 0; i < n; i++) {
       for (int j= 0; j < i; j++) {
         O(1)
       }
     }
```

En este caso el bucle exterior se realiza n veces, mientras que el interior se realiza 1, 2, 3, ..., n veces respectivamente. En total 1 + 2 + 3 + ... + n = n*(1+n)/2 = O(n2).

A veces aparecen bucles multiplicativos, donde la evolución de la variable de control no es lineal (como en los casos anteriores)

```
c= 1;
     while (c < n) {
       O(1)
       c= 2*c;
     }
```

El valor inicial de "c" es 1, siendo $2^k$ al cabo de k iteraciones. El número de iteraciones es tal que $2^k >= n$ -> k= $\log_2$ (n) (el entero inmediato superior) y, por tanto, la complejidad del bucle es O (log n).

```
c= n;
     while (c > 1) {
       O(1)
       c= c / 2;
     }
```

Razonando de un modo similar al caso anterior, obtenemos un orden O(log n).

```
for (int i= 0; i < n; i++) {
       c= i;
       while (c > 0) {
         O(1)
         c= c/2;
       }
     }
```

En este caso tenemos un bucle interno de orden O(log n) que se ejecuta n veces, luego el conjunto es de orden O(n log n).

### Llamadas a funciones

La operación de llamada a una función en si tiene un costo O(1), suponiendo que el propio cuerpo de la función también tiene costo O (1), … ¡a no ser que sean funciones recursivas!. Las funciones recursivas, esto es, aquellas funciones que se invocan a sí mismas, pueden comportarse de un modo análogo a un bucle. En el caso de funciones recursivas, tendremos que contar el número de veces que la función se invoca recursiva mente a sí misma. En el siguiente apartado veremos qué es una función recursiva.

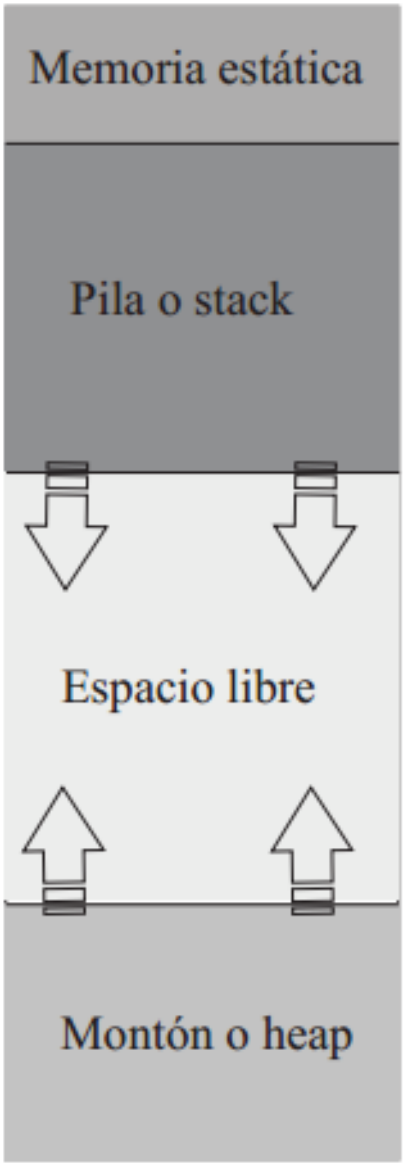
## *2.2 Limitaciones del análisis O*

El análisis O no es adecuado para pequeñas cantidades de datos, ya que las simplificaciones que se realizan en el cálculo de la complejidad del algoritmo se apoya en la suposición de que n es un número grande.

Las constantes grandes pueden entrar a juego en los algoritmos excesivamente complejos; en esto también influye el hecho de que el análisis no tiene en cuenta que la constante asociada a una operación simple, como un acceso memoria, es muy inferior a la constante asociada a un acceso a disco.

Por último, el análisis supone que contamos con una memoria infinita y no mide el impacto de este recurso en la eficiencia del algoritmo.

# 3 Recursividad




Una función recursiva es una función que se llama a sí misma. Esto es, dentro del cuerpo de la función se incluyen llamadas a la propia función. Esta estrategia es una alternativa al uso de bucles. Una solución recursiva es, normalmente, menos eficiente que una solución basada en bucles. Esto se debe a las operaciones auxiliares que llevan consigo las llamadas a las funciones. Cuando un programa llama a una función que llama a otra, la cual llama a otra y así sucesivamente, las variables y valores de los parámetros de cada llamada a cada función se guardan en la pila o stack, junto con la dirección de la siguiente línea de código a ejecutar una vez finalizada la ejecución de la función invocada. Esta pila va creciendo a medida que se llama a más funciones y decrece cuando cada función termina. Si una función se llama a si misma recursivamente un número muy grande de veces existe el riesgo de que se agote la memoria de la pila, causando la terminación brusca del programa.

A pesar de todos estos inconvenientes, en muchas circunstancias el uso de la recursividad permite a los programadores especificar soluciones naturales y sencillas que sin emplear esta técnica serían mucho más complejas de resolver. Esto convierte a la recursión en una potente herramienta de la programación. Sin embargo, por sus inconvenientes, debe emplearse con cautela.

Como regla básica, para que un problema pueda resolverse utilizando recursividad, **el problema debe poder definirse recursivamente** y, segundo, el problema debe incluir una **condición de terminación** porque, en otro caso, la ejecución continuaría indefinidamente. Cuando la condición de terminación es cierta la función no vuelve a llamarse a si misma.

Pueden distinguirse distintos tipos de llamadas recursivas dependiendo del número de funciones involucradas y de cómo se genera el valor final. A continuación veremos cuáles son.

## *3.1 Recursión lineal*

En la recursión lineal cada llamada recursiva genera, como mucho, otra llamada recursiva. Se pueden distinguir dos tipos de recursión lineal atendiendo a cómo se genera el resultado.

## Recursión lineal no final

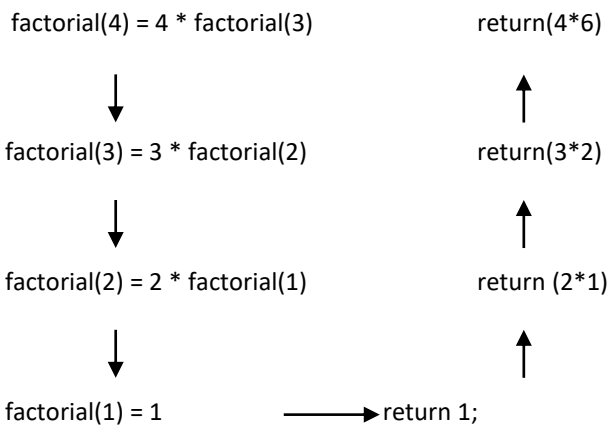
En la recursión lineal no final el resultado de la llamada recursiva se combina en una expresión para dar lugar al resultado de la función que llama. El ejemplo típico de recursión lineal no final es el cálculo del factorial de un número (n! = n * (n-1) * ...* 2 * 1 ). Dado que el factorial de un número n es igual al producto de n por el factorial de n-1, lo más natural es efectuar una implementación recursiva de la función factorial. Veamos una implementación de este cálculo:

```
package recursividad;
public class Ejemplo1 {

    public static int factorial(int numero) {
        if (numero > 1) {
            return (numero * factorial(numero - 1));
        } else {
            return (1);
        }
    }

    public static void main(String[] args) {
        int numero = 5;
  System.out.println("El factorial es " + factorial(numero));
    }
}
```

para calcular el factorial de 4 esta sería la secuencia de llamadas:




Cada fila del anterior gráfico supone una instancia distinta de ejecución de la función fact. Cada instancia tiene un conjunto diferente de variables locales.

## Recursión lineal final

En la recursión lineal final el resultado que es devuelto es el resultado de ejecución de la última llamada recursiva. Un ejemplo de este cálculo es el máximo común divisor, que puede hallarse a partir de la fórmula:

$$mcd(n,m)=\begin{cases} n & n=m \\ mcd(n-m,m) & n>m \\ mcd(n,m-n) & n<m \end{cases}$$

```
package recursividad;

public class Ejemplo2 {

    public static int mcd(int numero1, int numero2) {
        if (numero1 == numero2) {
            return numero1;
        } else if (numero1 < numero2) {
            return mcd(numero1, numero2 - numero1);
        } else {
            return mcd(numero1 - numero2, numero2);
        }
    }

    public static void main(String[] args) {
        int numero1 = 4454, numero2 = 143052;
        System.out.println("El m.c.d. es " + mcd(numero1,
numero2));
    }
}
```

## 3.2 Recursión múltiple

En la recurso múltiple alguna llamada recursiva puede generar más de una llamada a la función. Uno de los ejemplos más típicos son los números de Fibonacci, números que reciben el nombre del matemático italiano que los descubrió. Estos números se calculan mediante la fórmula:

$$F(n)=\begin{cases} 1, & \text{si } n=1; \\ 1, & \text{si } n=2; \\ F(n-1)+F(n-2) & \text{si } n>2; \end{cases}$$

Estos números poseen múltiples propiedades, algunas de las cuales todavía siguen descubriéndose hoy en día, entre las cuales están:

- La razón (el cociente) entre un término y el inmediatamente anterior varía continuamente, pero se estabiliza en un número irracional conocido como razón áurea o número áureo, que es la solución positiva de la ecuación $x^2$-x-1=0, y se puede aproximar por 1,618033989.
- Cualquier número natural se puede escribir mediante la suma de un número limitado de términos de la sucesión de Fibonacci, cada uno de ellos distinto a los demás. Por ejemplo, 17=13+3+1, 65=55+8+2.

- Tan sólo un término de cada tres es par, uno de cada cuatro es múltiplo de 3, uno de cada cinco es múltiplo de 5, etc. Esto se puede generalizar, de forma que la sucesión de Fibonacci es periódica en las congruencias módulo m, para cualquier m.
- Si F(p) es un número primo, p también es primo, con una única excepción. F(4)=3; 3 es primo, pero 4 no lo es.
- La suma infinita de los términos de la sucesión F(n)/10n es exactamente 10/89.

Veamos un programa que nos permite calcular el número n de la serie de Fibonacci:

```
package recursividad;

public class Ejemplo3 {

    public static int fibonacci(int numero) {
    if (0 == numero || 1 == numero) {
        return 1;
    } else {
        return (fibonacci(numero-1) + fibonacci(numero-2));
    }
}

    public static void main(String[] args) {
        int numero = 30;
        System.out.println("El m.c.d. es " +
fibonacci(numero));
    }
}
```

## 3.3 Recursión mutua

Implica más de una función que se llaman mutuamente. Un ejemplo es el determinar si un número es par o impar mediante dos funciones:

```
package recursividad;

public class Ejemplo4 {

    public static boolean par(int numero) {
        if (numero == 0) {
            return true;
        } else {
            return (impar(numero - 1));
        }
    }

    public static boolean impar(int numero) {
```

```
        if (numero == 0) {
            return false;
        } else {
            return (par(numero - 1));
        }
    }

    public static void main(String[] args) {
        int numero = 30;
        if (par(numero)) {
            System.out.println("El nmero es par");
        } else {
            System.out.println("El nmero es impar");
        }
    }
}
```

Veamos un ejemplo de ejecución de este programa:

```
par(5) -> return(impar(4))    return(0)  (no es par)
impar(4) -> return(par(3))    return(0)
par(3) -> return(impar(2))    return(0)
impar(2) -> return(par(1))    return(0)
par(1) -> return(impar(0))    return(0)
impar(0) -> return(0)
```

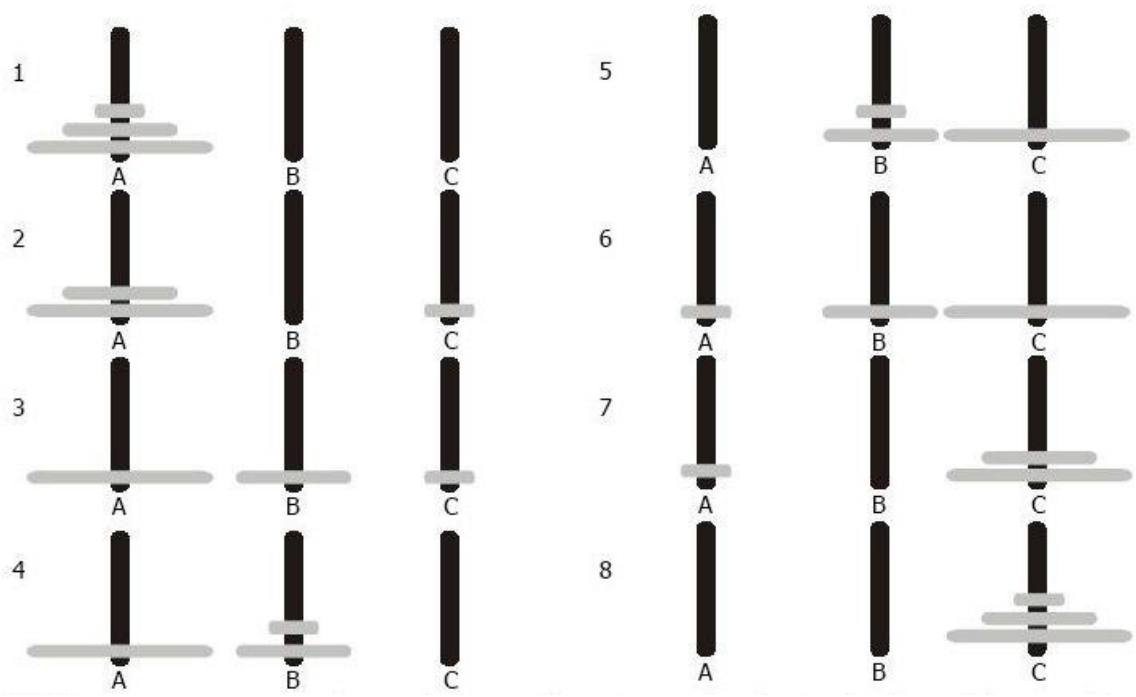
## *3.4 Las torres de Hanoi*

Las torres de Hanoi son un conocido juego de niños que se juega con tres pivotes y un cierto número de discos de diferentes tamaños. Los discos tienen un agujero en el centro y se pueden apilar en cualquiera de los pivotes. Inicialmente, los discos se amontonan en el pivote de la izquierda por tamaño decreciente como se muestra en la figura.

El objeto del juego es llevar los discos del pivote que está más a la izquierda al de más a la derecha. No se puede colocar nunca un disco sobre otro más pequeño y sólo se puede mover un disco a la vez. Cada disco debe encontrarse siempre en uno de los pivotes.

El problema de las torres de Hanoi puede resolverse fácilmente usando recursividad. La estrategia consiste en considerar uno de los pivotes como el origen y otro como destino. El problema de mover n discos al pivote derecho se puede formular recursivamente como sigue:

1. mover los n-1 discos superiores del pivote izquierdo al del centro empleando como ayuda el pivote de la derecha.
2. mover el n-ésimo disco (el más grande) al pivote de la derecha.
3. mover los n-1 discos del pivote del centro al de la derecha.




Este problema tiene una complejidad computacional exponencial: el número mínimo de movimientos para mover n discos es $2^n$-1.

### Leyenda sobre las torres de Hanoi

Cuenta la leyenda que Dios al crear el mundo, colocó tres varillas de diamante con 64 discos en la primera. También creó un monasterio con monjes, los cuales tienen la tarea de resolver esta Torre de Hanoi divina. El día que estos monjes consigan terminar el juego, el mundo acabará en un gran estruendo.

El mínimo número de movimientos que se necesita para resolver este problema es de 264-1. Si los monjes hicieran un movimiento por segundo, los 64 discos estarían en la tercera varilla en poco menos de 585 mil millones de años. Como comparación para ver la magnitud de esta cifra, la Tierra tiene unos 5 mil millones de años, y el Universo entre 15 y 20 mil millones de años de antigüedad, solo una pequeña fracción de esa cifra.

**Ejercicio voluntario**: Escribe un programa que resuelva el problema de las torres de Hanoi para n discos.

# 4 Algoritmos de búsqueda

Un problema de búsqueda puede enunciarse del siguiente modo: dado un conjunto de elementos CB (Conjunto Búsqueda) de un cierto tipo determinar si un elemento ("dato") se encuentra en el conjunto o no.

Existen diferentes algoritmos de búsqueda y la elección depende de la forma en que se encuentren organizados los datos: si se encuentran ordenados o si se ignora su disposición o se sabe que están al azar. También depende de si los datos a ordenar pueden ser accedidos de modo aleatorio o deben ser accedidos de modo secuencial.

## *4.1 Búsqueda secuencial*

Es el algoritmo de búsqueda más simple, menos eficiente y que menos precondiciones requiere: no requiere conocimientos sobre el conjunto de búsqueda ni acceso aleatorio. Consiste en comparar cada elemento del conjunto de búsqueda con el valor deseado hasta que éste sea encontrado o hasta que se termine de leer el conjunto.

Supondremos que los datos están almacenados en un array y se asumirá acceso secuencial. Se pueden considerar dos variantes del método: con y sin centinela.

### Búsqueda sin centinela

El algoritmo simplemente recorre el array comparando cada elemento con el dato que se está buscando. Los códigos de ejemplo que veremos a partir de ahora emplean la clase Utilidades para algunas tareas como inicializar de modo aleatorio los datos del conjunto de búsqueda o mostrarlos por pantalla. Veamos un código de ejemplo para la búsqueda secuencial sin centinela:

```
package ordenacionybusqueda;

public class Ejemplo5 {

    private static int TAM = 100;

    private static int secuencial(int[] conjuntoBusqueda, int
dato) {
        int i = 0;
        while ((conjuntoBusqueda[i] != dato) && (i < TAM)) {
            i++;
        }
        return i;
    }

    public static void main(String[] args) {
        int posicion, dato;

        int[] conjuntoBusqueda =
Utilidades.generaArayDeIntsAleatorio(TAM);
        Utilidades.imprimeCB(conjuntoBusqueda, TAM);
        dato = Utilidades.generaIntAleatorio();
        System.out.println("Dato a buscar " + dato);

        posicion = secuencial(conjuntoBusqueda, dato);
```

```
        if (conjuntoBusqueda[posicion] == dato) {
            System.out.println("Posicion " + posicion);
        } else {
            System.out.println("Elemento no esta en el
array");
        }
    }
}
```

La complejidad del algoritmo medida en número de iteraciones en el mejor caso será 1, y se corresponderá con aquella situación en la cual el elemento a buscar está en la primera posición del array. En el peor caso la complejidad será TAM y sucederá cuando el elemento buscar esté en la última posición del array. El promedio será (TAM+1)/2. El orden de complejidad es lineal (O (TAM)). Cada iteración necesita una suma, dos comparaciones y un AND lógico.

Consideremos un ejemplo: buscar 8 en el siguiente conjunto de datos:

```
0 9 5 5 8 4 6 0 4 9
- - - - -
Posicion 5
```

## Búsqueda con centinela

Si tuviésemos la seguridad de que el elemento buscado está en el conjunto, nos evitaría controlar si se supera el límite superior. Para tener esa certeza, se almacena un elemento adicional (centinela), que coincidirá con el elemento buscado y que se situará en la última posición del array de datos. De esta forma se asegura que encontraremos el elemento buscado.

```
package ordenacionybusqueda;

public class Ejemplo6 {

    private static int TAM = 100;

    private static int secuencialConCentinela(int[]
conjuntoBusqueda, int dato) {
        int posicion = 0;

        conjuntoBusqueda[TAM] = dato;
        while ((conjuntoBusqueda[posicion] != dato)) {
            posicion++;
        }
        return posicion;
    }
```

```
    public static void main(String[] args) {
        int posicion, dato;

        int[] conjuntoBusqueda =
Utilidades.generaArayDeIntsAleatorio(TAM + 1);
        Utilidades.imprimeCB(conjuntoBusqueda, TAM);
        dato = Utilidades.generaIntAleatorio();
        System.out.println("Dato a buscar " + dato);

        posicion = secuencialConCentinela(conjuntoBusqueda,
dato);

        if (posicion != TAM) {
            System.out.println("Posicion " + posicion);
        } else {
            System.out.println("Elemento no esta en el
array");
        }
    }
}
```

Ahora sólo se realiza una suma y una única comparación (se ahorra una comparación y un AND). El algoritmo es más eficiente.

## *4.2 Búsqueda binaria o dicotómica*

Es un método muy eficiente, pero tiene varios prerrequisitos:

1. El conjunto de búsqueda está ordenado.
2. Se dispone de acceso aleatorio.

Este algoritmo compara el dato buscado con el elemento central. Según sea menor o mayor se prosigue la búsqueda con el subconjunto anterior o posterior, respectivamente, al elemento central, y así sucesivamente.

```
package ordenacionybusqueda;

import java.util.Arrays;

public class Ejemplo7 {

    private static int TAM = 100;

    private static int busquedaDicotomica(int[]
conjuntoBusqueda, int dato) {
        int inicio = 0, fin = TAM - 1, mitad;

        mitad = (inicio + fin) / 2;
```

```
        while ((inicio <= fin) && (conjuntoBusqueda[mitad] !=
dato)) {
            if (dato < conjuntoBusqueda[mitad]) {
                fin = mitad - 1;
            } else {
                inicio = mitad + 1;
            }
            mitad = (inicio + fin) / 2;
        }

        if (dato == conjuntoBusqueda[mitad]) {
            return mitad;
        } else {
            return -1;
        }
    }

    public static void main(String[] args) {
        int posicion, dato;

        int[] conjuntoBusqueda =
Utilidades.generaArayDeIntsAleatorio(Ejemplo7.TAM);
        Utilidades.imprimeCB(conjuntoBusqueda, TAM);
        dato = Utilidades.generaIntAleatorio();
        System.out.println("Dato a buscar " + dato);

        Arrays.sort(conjuntoBusqueda);
        Utilidades.imprimeCB(conjuntoBusqueda, TAM);
        posicion = busquedaDicotomica(conjuntoBusqueda,
dato);

        if (posicion != -1) {
            System.out.println("Posicion " + posicion);
        } else {
            System.out.println("Elemento no esta en el
array");
        }
    }
}
```

En el caso más favorable (el dato es el elemento mitad) se realiza 1 iteración. En el caso más desfavorable, el número de iteraciones es el menor entero K que verifica $2^K >= TAM$. Esto es, el orden de complejidad es $O(\log^2(TAM))$.

Veamos un ejemplo; tenemos que buscar el número 8 en el siguiente conjunto de datos:

```
1 2 2 2 4 4 5 6 6 9  (ini,fin)      mitad
        -            (0  ,9)         4
1 2 2 2 4 4 5 6 6 9
              -      (5  ,9)         7
1 2 2 2 4 4 5 6 6 9
                -    (8  ,9)         8
```

```
1 2 2 2 4 4 5 6 6 9
                  -   (9 ,9)          9
No se encontro....    (9 ,8)
```

Busquemos el mismo valor en otro conjunto de datos:

```
0 1 1 2 2 6 7 7 8 9   (ini,fin)      mitad
        -             (0  ,9)          4
0 1 1 2 2 6 7 7 8 9
              -       (5  ,9)          7
0 1 1 2 2 6 7 7 8 9
                -     (8  ,9)          8
Posicion 8
```

A continuación mostramos una tabla que compara la eficiencia de la búsqueda binaria con la búsqueda secuencial. Se comprueba como para conjuntos de datos grandes la búsqueda binaria sigue siendo eficiente mientras que la secuencial se va degradando.

| **Número de elementos examinados (peor caso)** | | |
|---|---|---|
| **Tamaño del array** | **Búsqueda binaria** | **Búsqueda secuencial** |
| 1 | 1 | 1 |
| 10 | 4 | 10 |
| 1000 | 11 | 1000 |
| 5000 | 14 | 5000 |
| 100000 | 18 | 100000 |
| 1000000 | 21 | 1000000 |

# 5 Ordenación

La ordenación o clasificación es un proceso de organizar un conjunto de datos en algún orden o secuencia específica, tal como creciente o decreciente para datos numéricos o el orden alfabético para datos compuestos por caracteres. Los algoritmos de ordenación permutan los elementos del conjunto de datos hasta conseguir dicho orden. Para ello se basan en dos operaciones básicas: la comparación y el intercambio.

Existen muchos algoritmos de ordenación con diferentes ventajas e inconvenientes; en este tema veremos los más comunes.

## *5.1 Método de la búrbuja*

Se basa en recorrer el array ("realizar una pasada") un cierto número de veces, comparando pares de valores que ocupan posiciones adyacentes (0-1,1-2, ...). Si ambos datos no están ordenados, se intercambian. Esta operación se repite n-1 veces, siendo n el tamaño del conjunto de datos de entrada. Al final de la última pasada el elemento mayor estará en la última posición; en la segunda, el segundo elemento llegará a la penúltima, y así sucesivamente.

Su nombre se debe a que el elemento cuyo valor es mayor sube a la posición final del array, al igual que las burbujas de aire en un depósito suben a la parte superior. Para ello debe realizar un recorrido paso a paso desde su posición inicial hasta la posición final del array.

```
package ordenacionybusqueda;

public class Ejemplo8 {

    private static int TAM = 100;

    private static void burbuja(int[] datos) {
        int pasada = 0, auxiliar;
        for (pasada = 0; pasada < TAM; pasada++) {
            for (int i = 0; i < TAM - 1 - pasada; i++) {
                if (datos[i] > datos[i + 1]) {
                    auxiliar = datos[i + 1];
                    datos[i + 1] = datos[i];
                    datos[i] = auxiliar;
                }
            }
        }
    }

    public static void main(String[] args) {

        int[] datos =
Utilidades.generaArayDeIntsAleatorio(TAM);

        System.out.println("Antes de ordenar: ");
        Utilidades.imprimeCB(datos, TAM);

        burbuja(datos);

        System.out.println("Despues de ordenar: ");
        Utilidades.imprimeCB(datos, TAM);
    }
}
```

Veamos un ejemplo:

```
4 0 6 5 7 7 0 2 9 7
0 4 5 6 7 0 2 7 7 9
0 4 5 6 0 2 7 7 7 9
0 4 5 0 2 6 7 7 7 9
0 4 0 2 5 6 7 7 7 9
0 0 2 4 5 6 7 7 7 9
0 0 2 4 5 6 7 7 7 9
0 0 2 4 5 6 7 7 7 9
0 0 2 4 5 6 7 7 7 9
0 0 2 4 5 6 7 7 7 9
```

La complejidad computacional de este algoritmo es $O(TAM^2)$.

## 5.2 Burbuja mejorada

Existe una forma muy obvia para mejorar el algoritmo de la burbuja. Basta con tener en cuenta la posibilidad de que el conjunto esté ordenado en algún paso intermedio. Si el bucle interno no necesita realizar ningún intercambio en alguna pasada, el conjunto estará ya ordenado.

```
package ordenacionybusqueda;

public class Ejemplo9 {

    private static int TAM = 100;

    private static void burbujaMejorada(int[] datos) {
        int auxiliar, intercambio;
        for (int pasada = 0; pasada < TAM; pasada++) {
            intercambio = 0;
            for (int i = 0; i < TAM - 1 - pasada; i++) {
                if (datos[i] > datos[i + 1]) {
                    auxiliar = datos[i + 1];
                    datos[i + 1] = datos[i];
                    datos[i] = auxiliar;
                    intercambio = 1;
                }
            }
            if (intercambio == 0) {
                System.out.println("Para en la iteración " +
pasada);
                break;
            }
        }

    }

    public static void main(String[] args) {

        int[] datos =
Utilidades.generaArayDeIntsAleatorio(TAM);

        System.out.println("Antes de ordenar: ");
        Utilidades.imprimeCB(datos, TAM);

        burbujaMejorada(datos);

        System.out.println("Despues de ordenar: ");
        Utilidades.imprimeCB(datos, TAM);
    }
}
```

Veamos el mismo ejemplo que el caso anterior procesado mediante el algoritmo mejorado:

```
4 0 6 5 7 7 0 2 9 7
0 4 5 6 7 0 2 7 7 9
0 4 5 6 0 2 7 7 7 9
0 4 5 0 2 6 7 7 7 9
0 4 0 2 5 6 7 7 7 9
0 0 2 4 5 6 7 7 7 9
0 0 2 4 5 6 7 7 7 9
```

En el mejor caso (si ya está ordenado) realiza TAM-1 comparaciones. En el peor caso (el elemento menor estaba situado al fin del array) se necesitan las mismas pasadas que antes y el orden es TAM2. En el caso medio el orden es proporcional a TAM2/2. Obsérvese que éste algoritmo tiene una complejidad computacional en el peor caso igual al de la burbuja simple, aunque en término medio es aproximadamente dos veces más eficiente (requiere la mitad de tiempo para ejecutarse).

## *5.3 Método de inserción*

Este método considera que el array está formado por 2 partes: una parte ordenada (la izquierda) que sólo tendrá el primer elemento al principio y al final comprende todo el array; y una parte desordenada (la derecha) que al principio comprende todo el array menos el primer elemento y al final estará vacía. El algoritmo va tomando un elemento de la parte no ordenada para colocarlo en su lugar en la parte ordenada. El primer elemento del array (CB[0]) se considera ordenado (la lista ordenada inicial consta de un elemento). A continuación se inserta el segundo elemento (CB[1]) en la posición correcta (delante o detrás de CB[0]) dependiendo de que sea menor o mayor que CB[0]. Repetimos esta operación sucesivamente de tal modo que se va colocando cada elemento en la posición correcta. El proceso se repetirá TAM-1 veces.

Para colocar el dato en su lugar, se debe encontrar la posición que le corresponde en la parte ordenada y hacerle un hueco de forma que se pueda insertar. Para encontrar la posición se puede hacer una búsqueda secuencial desde el principio del conjunto hasta encontrar un elemento mayor que el dado. Para hacer el hueco hay que desplazar los elementos pertinentes una posición a la derecha.

```
package ordenacionybusqueda;

public class Ejemplo10 {

    private static int TAM = 100;

    private static void insercion(int[] datos) {
        int i, temp;
        for (int elementoAInsertar = 1; elementoAInsertar <
TAM; elementoAInsertar++) {
            temp = datos[elementoAInsertar];
            int dondeInsertar = 0;
```

```
            while (datos[dondeInsertar] < temp) {
                dondeInsertar++;
            }
            if (dondeInsertar < elementoAInsertar) {
                for (i = elementoAInsertar; i >
dondeInsertar; i--) {
                    datos[i] = datos[i - 1];
                }
                datos[dondeInsertar] = temp;
            }
        }
    }

    public static void main(String[] args) {

        int[] datos =
Utilidades.generaArayDeIntsAleatorio(TAM);

        System.out.println("Antes de ordenar: ");
        Utilidades.imprimeCB(datos, TAM);

        insercion(datos);

        System.out.println("Despues de ordenar: ");
        Utilidades.imprimeCB(datos, TAM);
    }
}
```

Veamos un ejemplo de la ejecución de este algoritmo:

```
4   1 6 9 1 0 2 9 8 4
1 4   6 9 1 0 2 9 8 4
1 4 6   9 1 0 2 9 8 4
1 4 6 9   1 0 2 9 8 4
1 1 4 6 9   0 2 9 8 4
0 1 1 4 6 9   2 9 8 4
0 1 1 2 4 6 9   9 8 4
0 1 1 2 4 6 9 9   8 4
0 1 1 2 4 6 8 9 9   4
0 1 1 2 4 4 6 8 9 9
```

El orden de complejidad de este algoritmo es cuadrático ($O(TAM^2)$).

## 5.4 Método de selección

Este método considera que el array está formado por 2 partes: una parte ordenada (la izquierda) que estará vacía al principio y al final comprende todo el array; y una parte desordenada (la derecha) que al principio comprende todo el array y al final estará vacía. El algoritmo toma elementos de la parte derecha y los coloca en la parte izquierda; empieza por el menor elemento de la parte desordenada y lo intercambia con el que ocupa su posición en la parte ordenada. Así, en la primera iteración se busca el

menor elemento y se intercambia con el que ocupa la posición 0; en la segunda, se busca el menor elemento entre la posición 1 y el final y se intercambia con el elemento en la posición 1. De esta manera las dos primeras posiciones del array están ordenadas y contienen los dos elementos menores dentro del array. Este proceso continúa hasta ordenar todos los elementos del array.

```
package ordenacionybusqueda;

public class Ejemplo11 {

    private static int TAM = 100;

    private static void selecion(int[] datos) {
        int posMenor, aux;
        for (int elemento = 0; elemento < (TAM - 1);
elemento++) {
            posMenor = elemento;
            for (int i = elemento + 1; i < TAM; i++) {
                if (datos[i] < datos[posMenor]) {
                    posMenor = i;
                }
            }
            aux = datos[elemento];
            datos[elemento] = datos[posMenor];
            datos[posMenor] = aux;
        }
    }

    public static void main(String[] args) {

        int[] datos =
Utilidades.generaArayDeIntsAleatorio(TAM);

        System.out.println("Antes de ordenar: ");
        Utilidades.imprimeCB(datos, TAM);

        selecion(datos);

        System.out.println("Despues de ordenar: ");
        Utilidades.imprimeCB(datos, TAM);
    }
}}
```

En cada pasada se coloca un elemento en su lugar, y la variable “elemento” marca donde empezar la búsqueda en la parte desordenada, que será secuencial si no tenemos más información. La búsqueda del siguiente elemento menor comienza suponiendo que dicho elemento es “elemento”. Se comprueba la hipótesis comparándolo con cada uno de los restantes. Si se encuentra uno menor, se intercambia.

Veamos un ejemplo de la ejecución de éste algoritmo:

```
    1 6 8 5 9 3 0 3 7
0     6 8 5 9 3 1 3 7
```

```
0 1     8 5 9 3 6 3 7
0 1 3     5 9 8 6 3 7
0 1 3 3     9 8 6 5 7
0 1 3 3 5     8 6 9 7
0 1 3 3 5 6     8 9 7
0 1 3 3 5 6 7     9 8
0 1 3 3 5 6 7 8     9
```

El número de comparaciones que realiza este algoritmo es independiente de la ordenación inicial. El bucle interno hace TAM-1 comparaciones la primera vez, TAM-2 la segunda,..., y 1 la última. El bucle externo hace TAM-1 búsquedas. El total de comparaciones es (TAM2+TAM)/2. Por tanto el orden de complejidad es cuadrático ($O(TAM^2)$).

## 5.5 Quicksort

El método de ordenación rápida (Quicksort) para ordenar los elementos del array se basa en el hecho de que es más rápido y fácil ordenar dos listas pequeñas que una lista grande. Su nombre se debe a que este método, en general, puede ordenar una lista de datos mucho más rápido que cualquier otro método de la bibliografía.

El método se basa en la estrategia típica de "divide y vencerás". El array a ordenar se divide en dos partes: una contendrá todos los valores menores o iguales a un cierto valor (que se suele denominar pivote) y la otra contendrá los valores mayores que dicho valor. El primer paso es dividir el array original en dos subarrays y un valor que sirve de separación, esto es, el pivote. Así, el array se dividirá en tres partes:

- La parte izquierda, que contendrá valores inferiores o iguales al pivote.
- El pivote.
- La parte derecha, que contiene valores superiores o iguales al pivote.

Inicialmente, las partes izquierda y derecha no estarán ordenadas, excepto en el caso de que estén compuestas por un único elemento. Consideremos, por ejemplo, la lista de valores:

```
28      21    37    23    19    14    26
```

elegimos como pivote el 23. Recorremos el array desde la izquierda y buscamos un elemento mayor que 23 (encontramos el 28). A continuación, recorremos el array en sentido descendente empezando por el extremo derecho y buscamos un valor menor que 23 (encontramos el 14). Se intercambian estos dos valores; los datos serán:

```
14      21    37    23    19    28    26
```

se sigue recorriendo el array por la izquierda y se encuentra otro número que es mayor que 23: el 37; continuamos el recorrido por la izquierda y encontramos otro valor menor que 23: 19. Volvemos a cambiar sus posiciones:

```
14      21    19    23    37    28    26
```

en este punto todo los valores que están a la izquierda del pivote son menores que él, y todos los que están a su derecha son mayores. Ninguno de los dos subconjuntos de valores está ordenado. En este momento procesamos cada uno de los dos subconjuntos de valores del mismo modo que el inicial: elegimos un valor de pivote y lo ordenamos de tal modo que todos los que sean mayores que el pivote estén a la derecha y todos los que sean menores a la izquierda. Así, si cogemos el primer conjunto de datos:

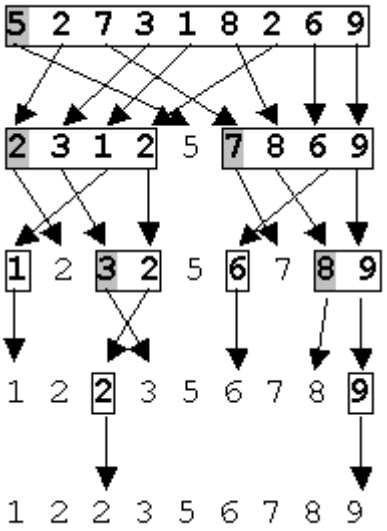
```
14      21    19
```

y tomamos como pivote el 14 obtenemos:

```
14      19    21
```

del mismo modo, procesamos el otro conjunto de datos. Cuando el tamaño de los conjuntos de datos que se sitúan a la izquierda y a la derecha del valor tomado como pivote es 0 o 1 habremos terminado el procedimiento de ordenación. Este algoritmo ordena correctamente los datos porque:

- El subconjunto de elementos menores que el pivote se ordenará correctamente, en virtud del proceso recursivo.
- El mayor elemento en el subconjunto de elementos menores no es mayor que el pivote, debido a cómo se realiza la partición.
- El menor elemento en el subconjunto de elementos mayores no es menor que el pivote, debido a cómo se realiza la partición.
- El subconjunto de elementos mayores se ordena correctamente, en virtud del proceso recursivo.




```
package ordenacionybusqueda;

public class Ejemplo12 {

    private static int TAM = 100;

    private static void quickSort(int datos[], int primerElemento,
int ultimoElemento) {
        int indicePivote;
```

```
        if (primerElemento < ultimoElemento) {
            indicePivote = hacerParticion(datos, primerElemento,
ultimoElemento);
            quickSort(datos, primerElemento, indicePivote - 1);
            quickSort(datos, indicePivote + 1, ultimoElemento);
        }
    }

    private static int hacerParticion(int datos[], int
primerElemento, int ultimoElemento) {
        int pivote, izquierda, derecha, tmp;
        pivote = datos[primerElemento];
        izquierda = primerElemento;
        derecha = ultimoElemento;

        while (true) {
            while (izquierda <= ultimoElemento && datos[izquierda]
<= pivote) {
                ++izquierda;
            }
            while (datos[derecha] > pivote) {
                --derecha;
            }
            if (izquierda >= derecha) {
                break;
            }
            tmp = datos[izquierda];
            datos[izquierda] = datos[derecha];
            datos[derecha] = tmp;
        }

        tmp = datos[primerElemento];
        datos[primerElemento] = datos[derecha];
        datos[derecha] = tmp;
        return derecha;
    }

    public static void main(String[] args) {

        int[] datos = Utilidades.generaArayDeIntsAleatorio(TAM);

        System.out.println("Antes de ordenar: ");
        Utilidades.imprimeCB(datos, TAM);

        quickSort(datos, 0, TAM-1);

        System.out.println("Despues de ordenar: ");
        Utilidades.imprimeCB(datos, TAM);
    }
}
```

## Análisis del rendimiento de Quicksort

El mejor caso para Quicksort se presenta cuando el pivote divide al conjunto en dos subconjuntos de igual tamaño. En este caso tendremos dos llamadas recursivas con un tamaño igual a la mitad del original y el tiempo de ejecución es O (TAM*log ($TAM^2$)).

Ya que los subconjuntos de igual tamaño son los mejores para Quicksort, es de esperar que los de muy distinto tamaño sean los peores y, efectivamente, así es. Suponiendo, por ejemplo, que el elemento que se toma como pivote es el menor del conjunto el subconjunto de la izquierda (elementos menores) estará vacío, y el de la derecha (elementos mayores) contendrá a todos los elementos menos al pivote. Poniéndonos en el peor de los casos, aquél en el que siempre obtenemos uno de los dos subconjuntos vacíos y el otro contiene n-1 elementos, la complejidad del algoritmo sería O ($TAM^2$).

En el caso medio, si el algoritmo es implementado cuidadosamente y los subconjuntos de elementos generados en cada partición contienen aproximadamente el mismo número de elementos, puede demostrarse que el tiempo de ejecución es O (TAM*log ($TAM^2$)). Para conseguir esta implementación cuidadosa es crucial determinar adecuadamente el elemento pivote.

Una elección muy popular para el elemento pivote, que puede degradar notablemente el rendimiento del algoritmo, es emplear el primer elemento de la izquierda. Este pivote es aceptable si la entrada es completamente aleatoria, pero si la entrada ya está ordenada, o está ordenada en orden inverso, este pivote proporciona la peor división posible. Una elección más razonable es elegir como elemento pivote el elemento central del array.

Lo ideal, sería elegir el valor mediana del array; esto es, aquel valor que ordenando en orden decreciente o creciente la lista de elementos quedaría justo en el medio (por tanto, este es el pivote ideal). Sin embargo, el cálculo de la mediana de una lista de elementos es excesivamente costoso para incorporarse al algoritmo; esto lleva a calcular aproximaciones tomando un subconjunto de los elementos del array. En muchas ocasiones se emplea la "partición con la mediana de tres", en esta partición se emplea como pivote la mediana de los elementos primero, último y punto medio del array.

Otro factor que afecta de modo crítico la eficiencia del algoritmo es qué hacer con aquellos datos que son iguales al pivote. Si, por ejemplo, sistemáticamente colocamos los datos que son mayores que el pivote en el subconjunto de la derecha esto puede llevarnos a realizar particiones desproporcionadas cuando un dato se repite un número excesivo de veces. El caso extremo será cuando todos los elementos de la lista a particionar sean iguales al pivote. En este hipotético caso la partición sería la peor de las posibles: una lista estaría vacía y la otra contendría todos los elementos.

A simple vista, podría parecer absurdo que un programa informático tuviera que ordenar una lista de valores iguales. Sin embargo, no lo es tanto: supongamos que tenemos que ordenar una lista con un millón de datos enteros comprendidos entre el 0 y 999. Si realizamos la ordenación mediante Quicksort y suponemos una distribución uniforme de los datos llegará un momento en el que, tras varias llamadas

recursivas, nuestro problema consista en ordenar 1000 listas de, aproximadamente, unos 1000 valores cada una de ellas. Estos valores serán en su mayor parte iguales: habrá una lista con todo 0s, una con todo 1s,.., y una con todo 999s. Llegado este momento, es obvia la importancia de que nuestra implementación de Quicksort gestione adecuadamente conjuntos de datos iguales.

La mejor estrategia en este caso es intercambiar los elementos iguales que estén a ambos lados del pivote: si al buscar un elemento menor que el pivote en el subconjunto de la izquierda encontramos un elemento igual al pivote lo intercambiaremos por un elemento mayor o igual que el pivote del subconjunto de la derecha. A pesar del aparente derroche de operaciones que, en caso de que ambos elementos intercambiados sean iguales, no hace nada en la práctica, es mejor gestionar esta situación de este modo y no añadir código adicional que chequee la ocurrencia de esta situación y que, inevitablemente, penalizaría la eficiencia del algoritmo.

# 6 Ejercicios

1. Construir una función recursiva que calcule la suma de los n primeros números naturales.
2. Construir una función recursiva que imprima la lista de números naturales comprendidos entre dos valores a y d dados por el usuario.
3. Escribir una función recursiva que devuelva la cantidad de dígitos de un número entero.
4. Escribir una función recursiva que calcule $x^y$ mediante multiplicaciones sucesivas, siendo x e y dos números enteros.
5. Calcula mediante un diseño recursivo el valor máximo de un vector de componentes numéricas.
6. Escribir la función recursiva que recibiendo como parámetros una cadena de dígitos hexadecimales y su longitud devuelva el valor decimal que representa dicha cadena.
7. Calcular C (n,k) siendo:

   $C(n,k) = C(n-1,k) + C(n-1,k-1)$ si $n>k>0$

   1 en otro caso

8. Cada tres bases de una cadena de ADN codifican un aminoácido de una proteína. Existe una secuencia especial de bases de la cadena, ATG, que es

una especie de “marca” del principio de un gen, esto es, una secuencia de bases que codifica una determinada proteína. También existen tres secuencias diferentes de fin de gen. Podemos a partir de una cadena de ADN contar cuántos genes tiene contando el número de veces que ocurre la tripleta ATG. Genera una cadena de ADN aleatorio, y a continuación cuenta cuántos genes contendría si se tratase de una cadena de ADN real.

9. Dado un fichero de texto ordena sus líneas alfabéticamente, generando un archivo nuevo que contenga el texto ordenado.
10. Dado un fichero de texto realizar un programa que permita buscar en él una palabra dada. Las palabras no tienen por qué estar ordenadas alfabéticamente y ninguna de ellas contiene la letra ñ.
11. Modificar el programa anterior para que emplee una búsqueda binaria.
12. Escribir un programa que permita introducir fechas al usuario. El usuario podrá introducir n fechas, y a continuación podrá listar todas las fechas ordenadas cronológicamente.
13. Optimizar la implementación del algoritmo quicksort dada en los apuntes según las indicaciones del último apartado de este tema (**ejercicio voluntario**).