

# Análisis y Diseño de Algoritmos

## Tema 1: Algoritmos y Complejidad



Algoritmos y Complejidad está bajo licencia  
Creative Commons Reconocimiento-  
NoComercial-CompartirIgual 4.0



# Contenido

- Coste computacional de un algoritmo
- Análisis de algoritmos no recursivos
- Análisis de algoritmos recursivos
  - Resolución de ecuaciones de recurrencia
- Órdenes de crecimiento y notación asintótica
  - El teorema maestro
- Introducción a la Teoría de la Complejidad

# Algoritmo

- Un **algoritmo** es un procedimiento sistemático, descrito paso a paso y de manera no ambigua, susceptible de ser realizado de manera mecánica, y que permite resolver un problema computacional.
- Un **problema** computacional es una clase de tareas resolubles mediante computadores.
  - Ejemplo: Compra (P, L) es el problema de realizar una lista de compra que contenga una selección de los objetos únicos de la lista L, de los cuales conocemos su precio, sabiendo que sólo disponemos de un presupuesto de P euros, que hay que gastar totalmente.
- **Instancia** de un problema: posibles realizaciones del problema.
  - Ejemplos:
    - Compra(1000, {50,200,2000,800, 300,450})
    - Compra (100, {1,2,3,4,5})

# Tipos de problemas

- Problema de **satisfacción**: dada una instancia del problema, intenta encontrar una solución factible (válida).
- Problema de **decisión**: dada una instancia del problema, determina si existe una solución factible o no.
- Problema de **conteo**: dada una instancia del problema, determina cuántas soluciones factibles existen para ella.
- Problema de **enumeración**: dada una instancia del problema, encuentra todas las soluciones factibles para ella.
- Problema de **optimización**: dada una instancia del problema, encuentra la mejor solución factible para resolverla.
  - Es necesario una función objetivo que proporciona un valor de calidad de las soluciones para poder buscar el máximo o mínimo requerido.

# Análisis de Algoritmos

- Parte de la Teoría de la Complejidad que se encarga del estudio del coste en recursos que necesita un algoritmo en el proceso de ejecución.
- Tipos de recursos:
  - **Tiempo**: en general, el coste suele hacer referencia al tiempo empleado en ejecutar el algoritmo.
  - **Espacio**: memoria necesaria para almacenar los datos utilizados por el algoritmo que no sean de entrada.
- El factor principal del que depende el coste de un algoritmo es el tamaño de la entrada que tiene que procesar.
  - Normalmente, cuanto mayor es la entrada, más recursos necesita el algoritmo para ejecutarse.
- A veces la naturaleza de los datos de entrada también hay que tenerla en cuenta. Estudiar los casos mejor, peor y medio para una entrada de tamaño  $n$ .



# Complejidad de un algoritmo

- La **complejidad** de un algoritmo A se define como **una función** que asocia a cada entrada del algoritmo su coste en tiempo de ejecución.

$$T_A : \mathbb{N} \rightarrow \mathbb{N}$$

Tamaño de la entrada

Tiempo de ejecución

- Una cuestión importante es determinar el **tamaño** de la **entrada** de un algoritmo
  - Ordenación o búsqueda en una lista: longitud de la lista
  - Evaluación de un polinomio en un punto: grado del polinomio.
  - Multiplicación de matrices cuadradas de tamaño  $n \times n$ :
    - La dimensión  $n$  de la matriz
    - El número de elementos de la matriz  $n \times n$
- De forma general, puede utilizarse el número de bits de la representación binaria de la entrada.
- En otras ocasiones puede ser que la función de la complejidad dependa de más de un parámetro de entrada.



# Coste Temporal de un Algoritmo

- Se estudiará el coste temporal de las diferentes instrucciones de un lenguaje
  - Instrucciones simples.
  - Composición de instrucciones.
  - Sentencias de selección.
  - Bucles
  - Subprogramas
- El tiempo de ejecución en segundos depende de factores que no tienen que ver con el algoritmo
  - Tipo de ordenador
  - Lenguaje de programación
- Toda instrucción simple cuyo tiempo de ejecución sea constante y acotado (k segundos) se considera una **operación elemental** y tendrá **coste 1**.
- El coste de una sentencia compleja indica el número de operaciones elementales que se han de ejecutar durante la ejecución de dicha sentencia.



# Coste de Instrucciones Simples

- Se consideran operaciones elementales de coste 1:
  - Las expresiones aritméticas, siempre que los datos sean de tamaño constante.
  - Las comparaciones de datos simples.
  - Las operaciones de asignación.
  - La lectura de datos simples desde la entrada estándar.
  - La escritura de datos simples a la salida estándar.
  - Las llamadas a procedimientos o funciones
  - Los saltos en el código (break, return,...)
  - Las operaciones de acceso a:
    - Una componente de un array.
    - Un campo de un registro (struct) o un atributo de un objeto.
    - La siguiente posición de un registro de un archivo en disco duro.





# Composición de Instrucciones

Si suponemos que las instrucciones  $I_1$  y  $I_2$  poseen coste computacional (complejidad temporal), en el peor de los casos, de  $T_1(n)$  y  $T_2(n)$  respectivamente, entonces el coste de la composición de ambas instrucciones será:

$$T_{I_1;I_2}(n) = T_1(n) + T_2(n)$$

```
void swap(int [] a,  
          int i, int j) {  
    int aux = a[i];  
    a[i] = a[j];  
    a[j] = aux;  
}
```

$T_{\text{swap}}(n) =$

```
void swap2(int[] a,  
           int i, int j) {  
    a[i] = a[i] + a[j];  
    a[j] = a[i] - a[j];  
    a[i] = a[i] - a[j];  
}
```

$T_{\text{swap2}}(n) =$



# Composición de Instrucciones

Si suponemos que las instrucciones  $I_1$  y  $I_2$  poseen coste computacional (complejidad temporal), en el peor de los casos, de  $T_1(n)$  y  $T_2(n)$  respectivamente, entonces el coste de la composición de ambas instrucciones será:

$$T_{I_1;I_2}(n) = T_1(n) + T_2(n)$$

```
void swap(int [] a,  
          int i, int j) {  
    int aux = a[i]; --2  
    a[i] = a[j]; --3  
    a[j] = aux; --2  
}
```

$$T_{\text{swap}}(n) = 2 + 3 + 2 = 7$$

```
void swap2(int[] a,  
           int i, int j) {  
    a[i] = a[i] + a[j]; --5  
    a[j] = a[i] - a[j]; --5  
    a[i] = a[i] - a[j]; --5  
}
```

$$T_{\text{swap2}}(n) = 5 + 5 + 5 = 15$$

¿Complejidad espacial?



# Coste de las instrucciones de selección

if <condición> then  $I_1$  else  $I_2$

$$T_{if\_else}(n) = T_{condición}(n) + \max\{T_1(n), T_2(n)\}$$

Case <expresión> of

caso<sub>1</sub>:  $I_1$ ;

caso<sub>2</sub>:  $I_2$ ;

.....

.....

caso<sub>n</sub>:  $I_n$ ;

end; {case}

$$T_{case}(n) = T_{expresión}(n) + \max\{T_1(n), \dots, T_n(n)\}$$



# Ejercicio

- Estudia la complejidad de los siguientes algoritmos en el peor caso.

```
int max(int a, int b) {  
    int res = b;  
  
    if (a > b) {  
        res = a;  
    }  
    return res;  
}
```

```
int num_dias(int mes,  
             boolean bisiestro) {  
    int res = 31;  
    switch(mes) {  
        case 2: if (bisiestro) {  
                    res = 29;  
                } else {  
                    res = 28;  
                }; break;  
        case 4:  
        case 6:  
        case 9:  
        case 11: res = 30;  
    }  
    return res;  
}
```

# Bucle con Iteraciones Fijas (I)

Al estudiar la complejidad de un bucle hay que analizar el **número de iteraciones** del mismo.

Si el bucle se realiza un número fijo de veces,  **$K$ , independiente de  $n$** , entonces la repetición sólo introduce una constante multiplicativa  $K$ .

```
for (int i= 0; i < K; i++)
{ bloque con coste C }
```

```
int i = 0; while (i < K){
{bloque con coste C; i++}
```

El coste computacional  $T(n)$  es

$$\begin{aligned} & \text{Coste("int i=0")} + \\ & \left( \sum_{i=0}^{K-1} \text{Coste("i<K")} + \text{Coste("i + "+"")} + C \right) + \\ & \text{Coste("i<K")} = \end{aligned}$$

Inicialización variable control

Coste iteraciones

Última evaluación de  $i < K$

$$1 + \left( \sum_{i=0}^{K-1} 1 + 2 + C \right) + 1 = 1 + (3 + C) \cdot K + 1 = \text{Constante}$$

$$\sum_{i=\min}^{\max} c = c \cdot (\max - \min + 1)$$

$$\sum_{i=1}^n \frac{1}{i^2} = \frac{\pi^2}{6}$$


# Bucle con Iteraciones Fijas (II)

Si el tamaño  $n$  aparece como **límite** de **iteraciones**, el coste dependerá del tamaño de la entrada.

```
for (int i= 0; i < n; i++)  
{ bloque con coste C }
```

```
int i = 0; while (i < n)  
{bloque con coste C; i++}
```

$$T(n) = 1 + \left( \sum_{i=0}^{n-1} 1 + 2 + C \right) + 1 = 1 + (3 + C) \cdot n + 1 \approx C \cdot n$$


$$\sum_{i=\min}^{\max} c = c \cdot (\max - \min + 1)$$



# Calcular la complejidad del Algoritmo

```
double media(int a []) {  
    double suma = 0.0;  
    int n = a.length;  
    int elems = 0;  
  
    for (int i = 0; i < n; i++) {  
        suma += a[i];  
        elems++;  
    }  
    if (elems > 0) {  
        suma = suma / elems;  
    }  
    return suma;  
}
```

# Bucles Anidados Con Iteraciones Fijas

La variable de control del **bucle interno** es **independiente**.

```
for (int i= 0; i < n; i++) {  
    for (int j= 0; j < n; j++) {  
        bloque con coste C  
    }  
}
```

$$\begin{aligned} \text{Coste}(n) &= 1 + \left[ \sum_{i=0}^{n-1} 1 + 2 + \left( 1 + \left( \sum_{j=0}^{n-1} 1 + 2 + C \right) + 1 \right) \right] + 1 = \\ &= 2 + \left[ \sum_{i=0}^{n-1} 1 + 2 + (1 + (1 + 2 + C)n + 1) \right] = 2 + \left[ \sum_{i=0}^{n-1} 5 + (3 + C)n \right] = \\ &= 2 + [5 + (3 + C)n] \cdot n = 2 + 5n + (3 + C)n^2 \approx Cn^2 \end{aligned}$$





# Bucle Con Iteraciones Variables(I)

El número de iteraciones del bucle interno **depende** de la iteración del bucle externo en la que estamos.

```
for (int i= 0; i < n; i++) {  
    for (int j= 0; j < i; j++) {  
        bloque de coste C  
    }  
}
```

el bucle exterior se realiza  $n$  veces, mientras que el interior se realiza 0, 1, 2 ...  $n-1$  veces respectivamente.

$$T(n) = 1 + \left[ \sum_{i=0}^{n-1} 1 + 2 + \left( 1 + \left( \sum_{j=0}^{i-1} 1 + 2 + C \right) + 1 \right) \right] + 1$$
$$= 2 + \left[ \sum_{i=0}^{n-1} 5 + (3 + C) \cdot i \right] = 2 + \sum_{i=0}^{n-1} 5 + \sum_{i=0}^{n-1} (3 + C) \cdot i$$

$$\sum_{i=a}^b s_i = \frac{(s_a + s_b)}{2} (b - a + 1)$$

$$= 2 + \sum_{i=0}^{n-1} 5 + (3 + C) \sum_{i=0}^{n-1} i = 2 + 5n + (3 + C) \frac{(n-1)(n)}{2} \approx \frac{C}{2} n^2$$



# Bucle con Iteraciones Variables (II)

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) {  
    bloque de coste A  
    c= 2*c;  
}
```

Tras Iteración i	c
0 (antes de entrar)	$1 = 2^0$
1	$2 = 1 * 2 = 2^1$
2	$4 = 2 * 2 = 2^2$
3	$8 = 2 * 2 * 2 = 2^3$
...	...
k	$2^k$

El valor inicial de "c" es 1, siendo  $2^k$  al cabo de k iteraciones.

Si el bucle acaba tras la iteración k, se debe a que la condición de control fue evaluada a falso y se cumple que  $c \geq n$ . Por ello,

$$2^k \geq n \Rightarrow k \geq \log_2(n) \Rightarrow k = \lceil \log_2(n) \rceil$$

$$T(n) = 1 + (1 + A + 2) \cdot \lceil \log_2(n) \rceil + 1 \approx A \cdot \log_2(n)$$



# Ejercicio

Dado el siguiente segmento de código:

```
c= n;  
while (c > 1) {  
    bloque de coste A  
    c= c / 2;  
}
```

1. Explicar de forma justificada cómo evoluciona la variable  $c$  y cuántas veces se ejecuta el bucle.
2. Calcular la complejidad de ese conjunto de instrucciones.

# Coste de los subprogramas

- El coste de una llamada a un subprograma cuya complejidad es  $T_{\text{sub}}(n)$  será  $1 + T_{\text{sub}}(n)$ .
- Hay que tener especial cuidado si en la expresión final aparece una variable que se corresponde con un parámetro de entrada.

```
boolean valido(int [] a, int pos) {
    boolean valido = true;
```

```
    int i = 0;
    while (valido && i < pos) {
        valido = a[i] < a[pos] ;
        i++;
    }
    return valido;
}
```

$$T_{\text{valido}}(n) = 1 + 1 + \sum_{i=0}^{pos-1} (2 + 6) + 2 + 1 = 5 + 8pos \rightarrow T_{\text{valido}}(n, pos)$$

```
boolean hayValido (int [] a) {
    int i = 0;
```

```
    int n = a.length;
    while (i < n && !valido(a,i)) {
        i++;
    }

    return (i < n);
}
```

$$T_{\text{hayvalido}}(n) = 1 + 2 + \sum_{i=0}^{n-1} (4 + T_{\text{valido}}(n, i) + 2) + 2 + 2 = 7 + \sum_{i=0}^{n-1} (6 + 5 + 8i) =$$

$$7 + 11n + 8 \frac{(n-1)n}{2} = 4n^2 + 7n + 7$$

Evaluación en cortocircuito. (i < n) es falso.

# Análisis de algoritmos iterativos

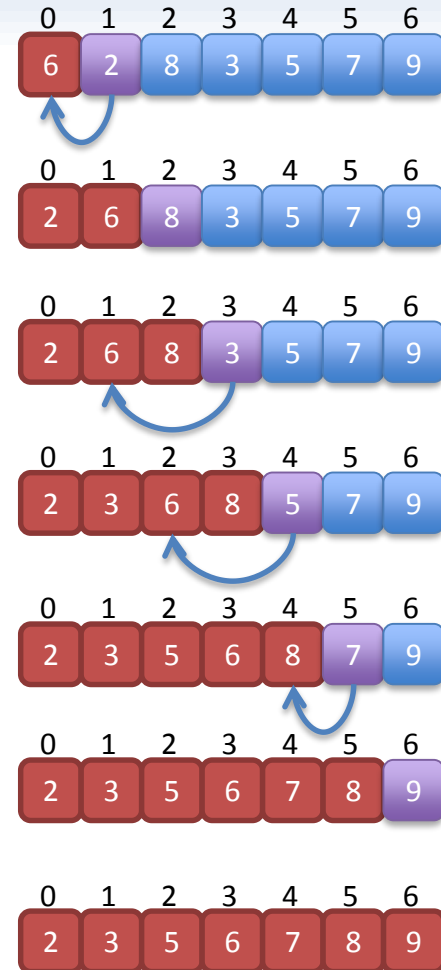
- Estrategia para analizar la eficiencia en tiempo de algoritmos no recursivos:
  - Decidir cómo medir el tamaño de la entrada
  - Identificar las operaciones básicas que se van a contabilizar (normalmente, las que están en el bucle más interno).
    - Simplifican las constantes de la expresión resultante.
  - Comprobar si el número de veces que se ejecutan las instrucciones básicas depende sólo del tamaño de la entrada.
    - Si hay otras condiciones a tener en cuenta habrá que estudiar las complejidades en el caso peor, mejor, y medio de manera separada.
  - Establecer la función de complejidad, una suma que defina el número de veces que se ejecutan las operaciones básicas.
  - Aplicar fórmulas para reducir la suma, de manera que pueda establecerse el orden de crecimiento (lo veremos más adelante)



# Análisis de algoritmos no recursivos:

## Ordenación por inserción

```
/**
 *
 * @param a array de enteros a
 * ordenar por inserción
 */
public static void insercion(int[] a){
    for (int i = 1; i < a.length; i++){
        // a[0..i-1] está ordenado
        int j = i-1;
        int x = a[i];
        // insertamos a[i] en a[0..i-1]
        // para que a[0..i] quede ordenado
        while (j >= 0 && a[j] > x) {
            a[j+1] = a[j];
            j--;
        }
        a[j+1] = x;
    }
}
```



# Análisis de algoritmos no recursivos:

## Ordenación por inserción (I)

```
/**
 *
 * @param a array de enteros a
 * ordenar por inserción
 */
public static void insercion(int[] a){
    for (int i = 1; i<a.length; i++){
        // a[0..i-1] está ordenado
        int j = i-1;
        int x = a[i];
        // insertamos a[i] en a[0..i-1]
        // para que a[0..i] quede ordenado
        while (j>=0 && a[j]>x) {
            a[j+1]= a[j];
            j--;
        }
        a[j+1]=x;
    }
}
```

- **Tamaño de la entrada:** longitud del array a ordenar,  $n = a.length$ .
- **Operaciones básicas:** acceso a componentes del array.
- **Caso mejor:** el array está ordenado
  - El bucle anidado no se ejecuta nunca

0	1	2	3	4	5	6
2	3	5	6	7	8	9

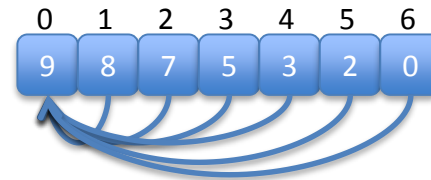
$$T_p(n) = \sum_{i=1}^{n-1} 3 = 3(n-1) = 3n-3$$

# Análisis de algoritmos no recursivos:

## Ordenación por inserción(II)

```
/**
 *
 * @param a array de enteros a
 * ordenar por inserción
 */
public static void insercion(int[] a){
    for (int i = 1; i<a.length; i++){
        // a[0..i-1] está ordenado
        int j = i-1;
        int x = a[i];
        // insertamos a[i] en a[0..i-1]
        // para que a[0..i] quede ordenado
        while (j>=0 && a[j]>x) {
            a[j+1]= a[j];
            j--;
        }
        a[j+1]=x;
    }
}
```

- **Caso peor:** el array está ordenado de forma decreciente
  - El bucle anidado se ejecuta siempre



$$T_p(n) = \sum_{i=1}^{n-1} 1 + \left( \sum_{j=0}^{i-1} (1 + 2) \right) + 1 =$$

$$\sum_{i=1}^{n-1} (2 + 3 \cdot i) = \sum_{i=1}^{n-1} 2 + \sum_{i=1}^{n-1} 3i$$

$$2(n-1) + 3 \frac{(1+n-1)}{2} (n-1) =$$

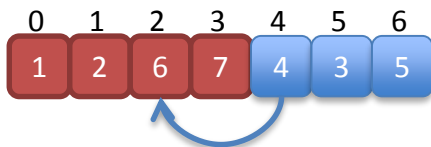
$$2(n-1) + \frac{3}{2}n(n-1) = 1.5n^2 + 0.5n - 2$$



# Análisis de algoritmos no recursivos:

## Ordenación por inserción (III)

```
/**
 *
 * @param a array de enteros a
 * ordenar por inserción
 */
public static void insercion(int[] a){
    for (int i = 1; i<a.length; i++){
        // a[0..i-1] está ordenado
        int j = i-1;
        int x = a[i];
        // insertamos a[i] en a[0..i-1]
        // para que a[0..i] quede ordenado
        while (j>=0 && a[j]>x) {
            a[j+1]= a[j];
            j--;
        }
        a[j+1]=x;
    }
}
```



- Caso medio:

- Supongamos que escogemos de forma aleatoria  $n$  números y aplicamos el algoritmo “insercion”.
- ¿Cuántas iteraciones se realizan para insertar el elemento  $a[i]$  en  $a[0..i-1]$ ? De media, la mitad de los números serán menores que  $a[i]$ , y la otra mitad mayores o iguales que  $a[i]$
- El bucle anidado se ejecuta  $i/2$  veces

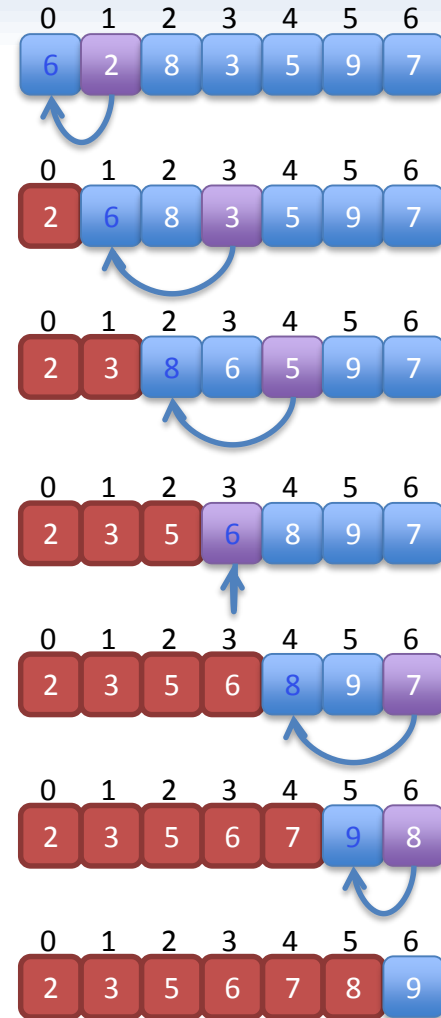
$$\begin{aligned} T_{1/2}(n) &= \sum_{i=1}^{n-1} \left( 2 + 3 \frac{i}{2} + 1 \right) = 3(n-1) + \frac{3}{2} \sum_{i=1}^{n-1} i = \\ &= 3(n-1) + \frac{3}{2} \frac{n(n-1)}{2} = \\ &= 3n - 3 + 0.75n^2 - 0.75n = \\ &= 0.75n^2 + 2.25n - 3 \end{aligned}$$

El caso medio es casi tan malo como el caso peor

# Análisis de algoritmos no recursivos:

## Ordenación por selección (I)

```
/**
 * @param a array de enteros
 * a ordenar por selección
 */
public static void seleccion(int[] a){
    for (int i = 0; i < a.length-1; i++){
        // a[0..i-1] está ordenado
        int min = i;
        for (int j=i+1; j < a.length; j++) {
            // min es el menor elemento
            // del array a[i.. n-1]
            if (a[j]< a[min]) min = j;
        }
        //intercambiamos a[i] y a[min]
        int temp = a[i];
        a[i] = a[min];
        a[min] = temp;
    }
}
```



# Análisis de algoritmos no recursivos:

## Ordenación por selección(II)

```
/**
 * @param a array de enteros
 * a ordenar por selección
 */
public static void seleccion(int[] a){
    for (int i = 0; i < a.length-1; i++){
        // a[0..i-1] está ordenado
        int min = i;
        for (int j=i+1; j < a.length; j++) {
            // min es el menor elemento
            // del array a[i..n-1]
            if (a[j] < a[min]) min = j;
        }
        //intercambiamos a[i] y a[min]
        int temp = a[i];
        a[i] = a[min];
        a[min] = temp;
    }
}
```

- **Tamaño** de la **entrada**: longitud del array a ordenar,  $n = a.length$ .
- **Operaciones básicas**: acceso a componentes del array.
- En este algoritmo, la complejidad **sólo** depende del tamaño de la entrada.

# Análisis de algoritmos no recursivos:

## Ordenación por selección(II)

```
/**
 * @param a array de enteros
 * a ordenar por selección
 */
public static void seleccion(int[] a){
    for (int i = 0; i < a.length-1; i++){
        // a[0..i-1] está ordenado
        int min = i;
        for (int j=i+1; j < a.length; j++) {
            // min es el menor elemento
            // del array a[i..n-1]
            if (a[j] < a[min]) min = j;
        }
        //intercambiamos a[i] y a[min]
        int temp = a[i];
        a[i] = a[min];
        a[min] = temp;
    }
}
```

- **Tamaño** de la **entrada**: longitud del array a ordenar,  $n = a.length$ .
- **Operaciones básicas**: acceso a componentes del array.
- En este algoritmo, la complejidad **sólo** depende del tamaño de la entrada.

$$\begin{aligned} T(n) &= \sum_{i=0}^{n-2} \left( 4 + \sum_{j=i+1}^{n-1} 2 \right) = \sum_{i=0}^{n-2} (4 + 2(n-i-1)) = \\ &= \sum_{i=0}^{n-2} 4 + 2n - 2 - 2 \sum_{i=0}^{n-2} i = \\ &= (4 + 2n - 2)(n-1) - (0 + n - 2)(n-1) = \\ &= (4 + 2n - 2 - n + 2)(n-1) = (n+4)(n-1) = \\ &= n^2 + 3n - 4 \end{aligned}$$

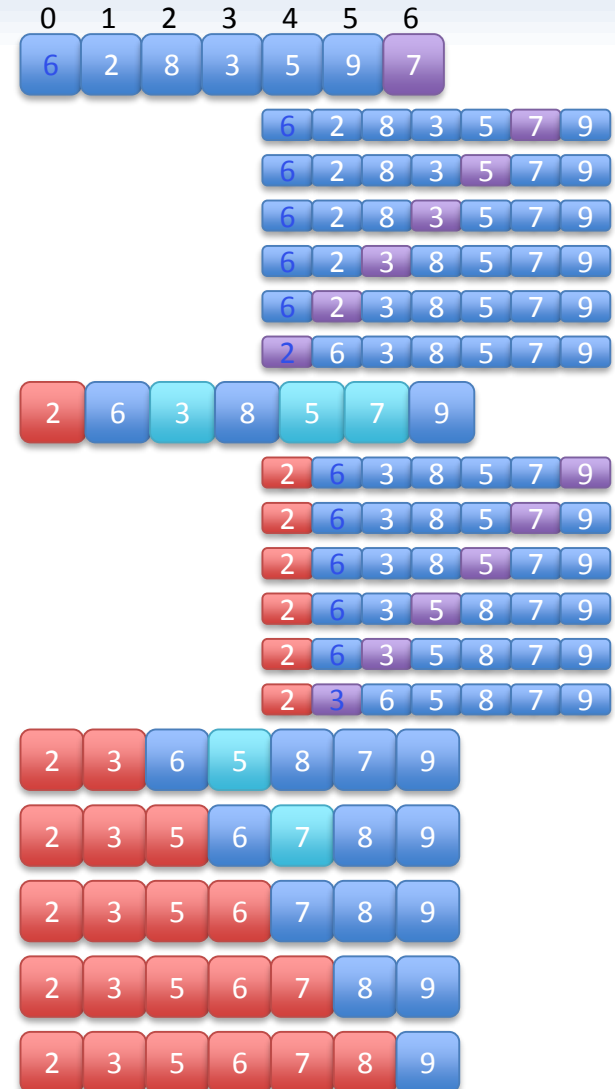
# Análisis de algoritmos no recursivos:

## Ordenación por el método de la Burbuja

```
/**
 * @param a array de enteros
 * Variante del algoritmo de selección
 * en el que en cada iteración del bucle
 * interno se mueven todos los datos
 * adyacentes que están desordenados
 */

public static void burbuja(int[] a){
    for (int i = 0; i<a.length-1; i++){
        // a[0..i-1] está ordenado
        for (int j=a.length-1;j>i;j--) {
            // a[j] es el menor elemento
            // del array a[j..a.length-1]
            if (a[j]< a[j-1]){
                //intercambiamos a[j] y a[j-1]
                int temp = a[j];
                a[j] = a[j-1];
                a[j-1] = temp;
            }
        }
    }
}
```

- En cada iteración del bucle externo, el i-ésimo elemento más pequeño flota hasta su posición.



# Análisis de algoritmos no recursivos:

## Ordenación por el método de la Burbuja

```
/**
 * @param a array de enteros
 * Variante del algoritmo de selección
 * en el que en cada iteración del bucle
 * interno se mueven todos los datos
 * adyacentes que están desordenados
 */
public static void burbuja(int[] a){
    for (int i = 0; i<a.length-1; i++){
        // a[0..i-1] está ordenado
        for (int j=a.length-1;j>i;j--) {
            // a[j] es el menor elemento
            // del array a[j..a.length-1]
            if (a[j]< a[j-1]){
                //intercambiamos a[j] y a[j-1]
                int temp = a[j];
                a[j] = a[j-1];
                a[j-1] = temp;
            }
        }
    }
}
```

- **Tamaño** de la **entrada**: longitud del array a ordenar,  $n = a.length$ .
- **Operaciones básicas**: acceso a componentes del array.
- El número de comparaciones es el mismo para todos los arrays de tamaño  $n$
- *La eficiencia del algoritmo va a depender del número de veces que se realicen intercambios.*

# Análisis de algoritmos no recursivos:

## Ordenación por el método de la Burbuja

```
/**
 * @param a array de enteros
 * Variante del algoritmo de selección
 * en el que en cada iteración del bucle
 * interno se mueven todos los datos
 * adyacentes que están desordenados
 */
public static void burbuja(int[] a){
    for (int i = 0; i<a.length-1; i++){
        // a[0..i-1] está ordenado
        for (int j=a.length-1;j>i;j--) {
            // a[j] es el menor elemento
            // del array a[j..a.length-1]
            if (a[j]<a[j-1]){
                //intercambiamos a[j] y a[j-1]
                int temp = a[j];
                a[j] = a[j-1];
                a[j-1] = temp;
            }
        }
    }
}
```

- El número de asignaciones depende de la distribución de los datos de entrada.
- El **caso peor** corresponde a un array ordenado descendentemente
  - Las asignaciones se ejecutan siempre.

0	1	2	3	4	5	6
9	8	7	5	3	2	0

$$T_p(n) = \sum_{i=0}^{n-2} \left( \sum_{j=i+1}^{n-1} 2 + 4 \right) =$$
$$\sum_{i=0}^{n-2} (6n - 6) - 6 \sum_{i=0}^{n-2} i =$$

$$(6n - 6)(n - 1) - 6 \frac{(n - 2)(n - 1)}{2}$$
$$= (6n - 6 - 3n + 6)(n - 1) = 3n(n - 1)$$
$$= 3n^2 - 3n$$

# Análisis de algoritmos no recursivos:

## Ordenación por el método de la Burbuja

```
/**
 * @param a array de enteros
 * Variante del algoritmo de selección
 * en el que en cada iteración del bucle
 * interno se mueven todos los datos
 * adyacentes que están desordenados
 */
public static void burbuja(int[] a){
    for (int i = 0; i<a.length-1; i++){
        // a[0..i-1] está ordenado
        for (int j=a.length-1;j>i;j--) {
            // a[j] es el menor elemento
            // del array a[j..a.length-1]
            if (a[j]<a[j-1]){
                //intercambiamos a[j] y a[j-1]
                int temp = a[j];
                a[j] = a[j-1];
                a[j-1] = temp;
            }
        }
    }
}
```

- El **caso mejor** corresponde a un array que ya está ordenado
  - Las asignaciones no se ejecutan nunca

0	1	2	3	4	5	6
2	3	5	6	7	8	9

$$T_m(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 2 = n(n-1) = n^2 - n$$

- Como en el algoritmo de ordenación por inserción, el **caso medio** ocurre cuando las asignaciones se realizan la mitad de las iteraciones del bucle interno,  $(n-i-1)/2$ .



# Algunas fórmulas útiles

$$\sum_{i=\min}^{\max} c \cdot a_i = c \sum_{i=\min}^{\max} a_i$$

$$\sum_{i=\min}^{\max} a_i \pm b_i = \sum_{i=\min}^{\max} a_i \pm \sum_{i=\min}^{\max} b_i$$

$$\sum_{i=\min}^{\max} c = c \cdot (\max - \min + 1)$$

$$\sum_{i=0}^n i = \sum_{i=1}^n i = \frac{n(n+1)}{2}$$

$$\sum_{i=\min}^{\max} a_i = \frac{(a_{\min} + a_{\max})}{2} (\max - \min + 1)$$

$$\log_a(x) = y \Leftrightarrow a^y = x$$

$$\log_a(x * y) = \log_a(x) + \log_a(y)$$

$$\log_a(x/y) = \log_a(x) - \log_a(y)$$

$$\log_a(x^y) = y * \log_a(x)$$

$$\log_a(x) = \frac{\log_b(x)}{\log_b(a)}$$

$$a^{\log_b(c)} = c^{\log_b(a)}$$

$$\forall i \geq 0 \ a_{i+1} - a_i = d, \quad d \in \mathbb{N}$$