

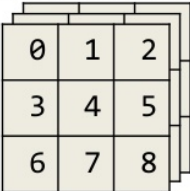
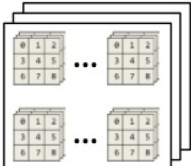


# Ejercicios de la UD04

<u>Dimensions</u>	<u>Example</u>	<u>Terminology</u>
<b>1</b>		Vector
<b>2</b>		Matrix
<b>3</b>		3D Array (3 <sup>rd</sup> order Tensor)
<b>N</b>		ND Array

# 1. Ejercicios

---

## 1.1. Arrays. Ejercicios de recorrido

---

1. (Estaturas) Escribir un programa que lea de teclado la estatura de 10 personas y las almacene en un array. Al finalizar la introducción de datos, se mostrarán al usuario los datos introducidos con el siguiente formato:

```
1 Persona 1: 1.85 m.  
2 Persona 2: 1.53 m.  
3 ...  
4 Persona 10: 1.23 m.
```

2. (Lluvias) Se dispone de un fichero, de nombre *lluviasEnero.txt*, que contiene 31 datos correspondientes a las lluvias caídas en el mes de enero del pasado año. Se desea analizar los datos del fichero para averiguar:

- La lluvia total caída en el mes.
- La cantidad media de lluvias del mes.
- La cantidad más grande de lluvia caída en un solo día.
- Cual fue el día que más llovió.
- La cantidad más pequeña de lluvia caída en un solo día.
- Cual fue el día que menos llovió.
- En cuantos días no llovió nada.
- En cuantos días la lluvia superó la media.
- Si en la primera quincena del mes llovió más o menos que en la segunda.
- En cuantos días la lluvia fue menor que la del día siguiente.

Para resolver el problema se desarrollarán los siguientes métodos:

3. 1. `public static void leerArray (double v[], String nombreFichero)`, que rellena el array `v` con datos que se encuentran en el fichero especificado. El número de datos a leer vendrá determinado por el tamaño del array y no por la cantidad de datos que hay en el fichero.
2. `public static double suma(double v[])`, que devuelve la suma de los elementos del array `v`
3. `public static double media(double v[])`, que devuelve la media de los elementos del array `v`. Se puede hacer uso del método del apartado anterior.
4. `public static double maximo(double v[])`, que devuelve el valor más grande almacenado en el array `v`.
5. `public static double minimo(double v[])`, que devuelve el valor más pequeño almacenado en el array `v`.

6. `public static int posMaximo(double v[])`, que devuelve la posición del elemento más grande de `v`. Si éste se repite en el array es suficiente devolver la posición en que aparece por primera vez.
  7. `public static int posMinimo(double v[])`, que devuelve la posición del elemento más pequeño de `v`. Si éste se repite en el array es suficiente devolver la posición en que aparece por primera vez.
  8. `public static int contarApariciones(double v[], double x)`, que devuelve el número de veces que el valor `x` aparece en el array `v`.
  9. `public static double sumaParcial(double v[], int izq, der)`, que devuelve la suma de los elementos del array `v` que están entre las posiciones `izq` y `der`.
  10. `public static int menoresQueElSiguiente(double v[])`, que devuelve el número de elementos de `v` que son menores que el elemento que tienen a continuación.
4. (Dados) El lanzamiento de un dado es un experimento aleatorio en el que cada número tiene las mismas probabilidades de salir. Según esto, cuantas más veces lancemos el dado, más se igualarán las veces que aparece cada uno de los 6 números. Vamos a hacer un programa para comprobarlo.

- Generaremos un número aleatorio entre 1 y 6 un número determinado de veces (por ejemplo 100.000). Para ello puedes usar la clase `Random`.
- Tras cada lanzamiento incrementaremos un contador correspondiente a la cifra que ha salido. Para ello crearemos un array `veces` de 7 componentes, en el que el `veces[1]` servirá para contar las veces que sale un 1, `veces[2]` para contar las veces que sale un 2, etc. `veces[0]` no se usará.
- Cada, por ejemplo, 1.000 lanzamientos mostraremos por pantalla las estadísticas que indican que porcentaje de veces ha aparecido cada número en los lanzamientos hechos hasta ese momento. Por ejemplo:

```

1  Número de lanzamientos: 1000
2  1: 18 %
3  2: 14 %
4  3: 21 %
5  4: 10 %
6  5: 18 %
7  6: 19 %
8
9  Número de lanzamientos: 2000
10 ...

```

- Para el número de lanzamientos (100.000 en el ejemplo) y para la frecuencia con que se muestran las estadísticas (1.000 en el ejemplo) utilizaremos dos **constantes** enteras, de nombre `LANZAMIENTOS` y `FRECUENCIA`, de esta forma podremos variar de forma cómoda el modo en que probamos el programa.
5. (Invertir) Diseñar un método `public static int[] invertirArray(int v[])`, que dado un array `v` devuelva otro con los elementos en orden inverso. Es decir, el último en primera posición, el penúltimo en segunda, etc.

Desde el método `main` crearemos e inicializaremos un array, llamaremos a `invertirArray` y mostraremos el array invertido.

6. (SumasParciales) Se quiere diseñar un método `public static int[] sumaParcial(int v[])`, que dado un array de enteros `v`, devuelva otro array de enteros `t` de forma que `t[i] = v[0] + v[1] + ... + v[i]`. Es decir:

```

1  t[0] = v[0]
2  t[1] = v[0] + v[1]
3  t[2] = v[0] + v[1] + v[2]
4  ...
5  t[10] = v[0] + v[1] + v[2] + ... + v[10]

```

Desde el método `main` crearemos e inicializaremos un array, llamaremos a `sumaParcial` y mostraremos el array resultante.

7. (Rotaciones) Rotar una posición a la derecha los elementos de un array consiste en mover cada elemento del array una posición a la derecha. El último elemento pasa a la posición 0 del array. Por ejemplo si rotamos a la derecha el array `{1,2,3,4}` obtendríamos `{4,1,2,3}`.

- Diseñar un método `public static void rotarDerecha(int v[])`, que dado un array de enteros rote sus elementos un posición a la derecha.
- En el método `main` crearemos e inicializaremos un array y rotaremos sus elementos tantas veces como elementos tenga el array (mostrando cada vez su contenido), de forma que al final el array quedará en su estado original. Por ejemplo, si inicialmente el array contiene `{7,3,4,2}`, el programa mostrará

```

1  Rotación 1: 2 7 3 4
2  Rotación 2: 4 2 7 3
3  Rotación 3: 3 4 2 7
4  Rotación 4: 7 3 4 2

```

- Diseña también un método para rotar a la izquierda y pruébalo de la misma forma.
8. (DosArrays) Desarrolla los siguientes métodos en los que intervienen dos arrays y pruébalos desde el método `main`

- `public static double[] sumaArraysIguales (double a[], double b[])` que dados dos arrays de `double` `a` y `b`, del mismo tamaño devuelva un array con la suma de los elementos de `a` y `b`, es decir, devolverá el array `{a[0]+b[0], a[1]+b[1], ....}`
- `public static double[] sumaArrays(double a[], double b[])`. Repite el ejercicio anterior pero teniendo en cuenta que `a` y `b` podrían tener longitudes distintas. En tal caso el número de elementos del array resultante coincidirá con la longitud del array de mayor tamaño.

## 1.2. Arrays. Ejercicios de búsqueda

1. (Lluvias – continuación). Queremos incorporar al programa la siguiente información:

- Cual fue el **primer** día del mes en que llovió exactamente 19 litros (si no hubo ninguno mostrar un mensaje por pantalla indicándolo)
- Cual fue el **último** día del mes en que llovió exactamente 8 litros (si no hubo ninguno mostrar un mensaje por pantalla indicándolo)

Para ello desarrollarán los siguientes métodos:

- `public static int posPrimero(double v[], double x)`, que devuelve la posición de la primera aparición de `x` en el array `v`. Si `x` no está en `v` el método devolverá -1. El método realizará una búsqueda ascendente para proporcionar el resultado.
- `public static int posUltimo(double v[], double x)`, que devuelve la posición de la última aparición de `x` en el array `v`. Si `x` no está en `v` el método devolverá -1. El método realizará una búsqueda descendente para proporcionar el resultado.

2. (Tocayos) Disponemos de los nombres de dos grupos de personas (dos arrays de `String`). Dentro de cada grupo todas las personas tienen nombres distintos, pero queremos saber cuántas personas del primer grupo tienen algún tocayo en el segundo grupo, es decir, el mismo nombre que alguna persona del segundo grupo. Escribir un programa que resuelva el problema (inicializa los dos arrays con los valores que quieras y diseña los métodos que consideres necesarios).

Por ejemplo, si los nombres son {"miguel","**josé**","ana","maría"} y {"ana", "**josé**", "luján", "juan", "**josé**", "pepa", "ángela", "sofía", "andrés", "bartolo"}, el programa mostraría:

```
1 | josé tiene tocayo en el segundo grupo.
2 | ana tiene tocayo en el segundo grupo.
3 | TOTAL: 2 personas del primer grupo tienen tocayo.
```

Optimiza el algoritmo para que no tenga en cuenta si se escribe el nombre en mayúsculas, minúsculas o cualquier combinación.

3. (PrimerImpar) Escribir un método que, dado un array de enteros, devuelva la suma de los elementos que aparecen tras el primer valor impar. Usar `main` para probar el método.

Para determinar si existe algún valor par en un array se proponen varias soluciones. Indica cual/cuales son válidas para resolver el problema.

```
1 | public static boolean haypares1(int v[]) {
2 |
3 |     int i = 0;
4 |
5 |     while (i < v.length && v[i] % 2 != 0) {
6 |         i++;
7 |     }
8 |
9 |     if (v[i] % 2 == 0) {
10 |         return true;
11 |     } else {
12 |         return false;
13 |     }
14 |
15 | }
16 |
17 | public static boolean haypares2(int v[]) {
18 |
19 |     int i = 0;
20 |
21 |     while (i < v.length && v[i] % 2 != 0) {
22 |         i++;
23 |     }
24 |
25 |     if (i < v.length) {
26 |         return true;
27 |     } else {
28 |         return false;
29 |     }
30 |
31 | }
32 |
33 | public static boolean haypares3(int v[]) {
34 |
```

```
35     int i = 0;
36
37     while (v[i] % 2 != 0 && i < v.length) {
38         i++;
39     }
40
41     if (i < v.length) {
42         return true;
43     } else {
44         return false;
45     }
46
47 }
48
49 public static boolean haypares4(int v[]) {
50
51     int i = 0;
52
53     boolean encontrado = false;
54
55     while (i <= v.length && !encontrado) {
56
57         if (v[i] % 2 == 0) {
58             encontrado = true;
59         } else {
60             i++;
61         }
62
63     }
64
65     return encontrado;
66
67 }
68
69 public static boolean haypares5(int v[]) {
70
71     int i = 0;
72
73     boolean encontrado = false;
74
75     while (i < v.length && !encontrado) {
76
77         if (v[i] % 2 == 0) {
78             encontrado = true;
79         }
80
81         i++;
82
83     }
84
85     return encontrado;
86
87 }
88
89 public static boolean haypares6(int v[]) {
90
91     int i = 0;
```

```

93     while (i < v.length) {
94
95         if (v[i] % 2 == 0) {
96             return true;
97         } else {
98             return false;
99         }
100
101     }
102
103 }
104
105 public static boolean haypares7(int v[]) {
106
107     int i = 0;
108
109     while (i < v.length) {
110
111         if (v[i] % 2 == 0) {
112             return true;
113         }
114
115         i++;
116
117     }
118
119     return false;
120
121 }

```

4. (Capicúa) Escribir un método para determinar si un array de palabras (`String`) es capicúa, esto es, si la primera y última palabra del array son la misma, la segunda y la penúltima palabras también lo son, y así sucesivamente. Escribir el método main para probar el método anterior.
5. (Subsecuencia) Escribir un método que, dado un array, determine la posición de la primera subsecuencia del array que comprenda al menos tres números enteros consecutivos en posiciones consecutivas del array. De no existir dicha secuencia devolverá -1.  
 Por ejemplo: en el array {23, 8, 12, 6, 7, **9, 10, 11**, 2} hay 3 números consecutivos en tres posiciones consecutivas, a partir de la posición 5: {9,10,11}
6. Se desea comprobar si dos arrays de `double` contienen los mismos valores, aunque sea en orden distinto. Para ello se ha escrito el siguiente método, que aparece incompleto:

```

1  public static boolean mismosValores(double v1[], double v2[]) {
2      boolean encontrado = false;
3      int i = 0;
4      while (i < v1.length && !encontrado) {
5          boolean encontrado2 = false;
6          int j = 0;
7          while (j < v2.length && !encontrado2) {
8              if (v1[?] == v2[?]) {
9                  encontrado2 = true;
10             } else {
11                 ?
12             }
13         }

```

```

14         if (encontrado2 == ?) {
15             encontrado = true;
16         }
17     }
18     return !encontrado;
19 }

```

Completa el programa en los lugares donde aparece el símbolo **?**

## 1.3. Matrices

1. (Notas). Se dispone de una matriz que contiene las notas de una serie de alumnos en una serie de asignaturas. Cada fila corresponde a un alumno, mientras que cada columna corresponde a una asignatura. Desarrollar métodos para:

1. Imprimir las notas alumno por alumno.
2. Imprimir las notas asignatura por asignatura.
3. Imprimir la media de cada alumno.
4. Imprimir la media de cada asignatura.
5. Indicar cual es la asignatura más fácil, es decir la de mayor nota media.
6. ¿Hay algún alumno que suspenda todas las asignaturas?
7. ¿Hay alguna asignatura en la que suspendan todos los alumnos?

2. (Ventas). Una empresa comercializa 10 productos para lo cual tiene 5 distribuidores.

Los datos de ventas los tenemos almacenados en una matriz de 5 filas x 10 columnas, `ventas`, con el número de unidades de cada producto que ha vendido cada distribuidor. Cada fila corresponde a las ventas de un distribuidor (la primera fila, del primer distribuidor, etc.), mientras que cada columna corresponde a un producto :

100	25	33	89	23	90	87	6	5	233
28	765	65	77	987	55	4	66	4	8
...									

El array `precio`, de 10 elementos, contiene el precio en € de cada uno de los 10 productos.

125.2	234.4	453.9	...						
-------	-------	-------	-----	--	--	--	--	--	--

Escribe el programa y los métodos necesarios para averiguar:

1. Distribuidor que más artículos ha vendido.
  2. El artículo que más se vende.
  3. Sabiendo que los distribuidores que realizan ventas superiores a 30.000€ cobran una comisión del 5% de las ventas y los que superan los 70.000€ una comisión del 8%, emite un informe de los distribuidores que cobran comisión, indicando nº de distribuidor, importe de las ventas, porcentaje de comisión e importe de la comisión en €.
3. (Útiles) Dada una matriz con el mismo número de filas y de columnas, diseñar los siguientes métodos:
- `public static void mostrarDiagonal(int m[][])` que muestre por pantalla los elementos de la diagonal principal.

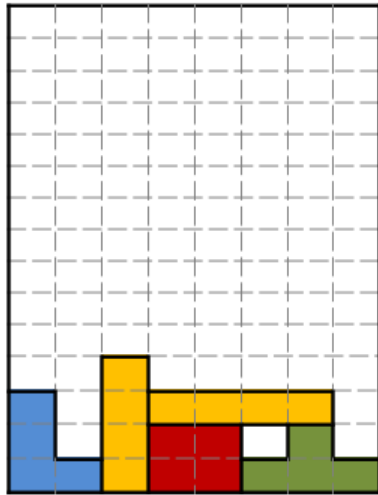


- `public static int filaDelMayor (int m[][])`, que devuelva la fila en que se encuentra el mayor elemento de la matriz.
- `public static void intercambiarFilas(int f1, int f2)`, que intercambie los elementos de las filas indicadas.
- Escribir un método `public static boolean esSimetrica (int m[][])` que devuelva true si la matriz m es simétrica. Una matriz es simétrica si tiene el mismo número de filas que de columnas y además  $m[i][j] = m[j][i]$  para todo par de índices  $i, j$ .

Por ejemplo, es simétrica:

```
1 | 1 5 3
2 | 5 4 7
3 | 3 7 5
```

4. (Tetris) Supongamos que estamos desarrollando un Tetris en Java y para representar la partida utilizamos una matriz bidimensional de enteros 15 filas por 8 columnas. Se utiliza el valor 0 para indicar que la celda está vacía y un valor distinto de cero para las celdas que contienen parte de una pieza (distintos valores para distintos colores):



0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	2	0	0	0	0	0
1	0	2	2	2	2	2	0
1	0	2	4	4	0	3	0
1	1	2	4	4	3	3	3

Escribir un método que reciba la matriz y elimine las filas completas, haciendo caer las piezas que hay por encima de las celdas eliminadas tal y como se hace en el juego.

## 2. Recursividad

1. Implemente, tanto de forma recursiva como de forma iterativa, una función que nos diga si una cadena de caracteres es simétrica (un palíndromo). Por ejemplo, "DABALEARROZALAZORRAELABAD" es un palíndromo.

2. Implemente, tanto de forma recursiva como de forma iterativa, una función que le dé la vuelta a una cadena de caracteres.

NOTA: Obviamente, si la cadena es un palíndromo, la cadena y su inversa coincidirán.

3. Implemente, tanto de forma recursiva como de forma iterativa, una función que permitan calcular el número de combinaciones de  $n$  elementos tomados de  $m$  en  $m$ .

Realice dos versiones de la implementación iterativa, una aplicando la fórmula y otra utilizando una matriz auxiliar (en la que se vaya construyendo el triángulo de Pascal).

4. Implemente, tanto de forma recursiva como de forma iterativa, una función que nos devuelva el máximo común divisor de dos números enteros utilizando el algoritmo de Euclides.

```
1  ALGORITMO DE EUCLIDES
2  Dados dos números enteros positivos m y n, tal que m > n, para encontrar su
   máximo común divisor (es decir, el mayor entero positivo que divide a ambos):
3  - Dividir m por n para obtener el resto r ( $0 \leq r < n$ )
4  - Si r = 0, el MCD es n.
5  - Si no, el máximo común divisor es MCD(n,r).
```

5. La ordenación por mezcla (mergesort) es un método de ordenación que se basa en un principio muy simple: se ordenan las dos mitades de un vector y, una vez ordenadas, se mezclan. Escriba un programa que implemente este método de ordenación.

6. Diseñe e implemente un algoritmo que imprima todas las posibles descomposiciones de un número natural como suma de números menores que él (sumas con más de un sumando).

7. Diseñe e implemente un método recursivo que nos permita obtener el determinante de una matriz cuadrada de dimensión  $n$ .

8. Diseñe e implemente un programa que juegue al juego de cifras de "Cifras y Letras". El juego consiste en obtener, a partir de 6 números, un número lo más cercano posible a un número de tres cifras realizando operaciones aritméticas con los 6 números.

9. Problema de las 8 reinas: Se trata de buscar la forma de colocar 8 reinas en un tablero de ajedrez de forma que ninguna de ellas amenace ni se vea amenazada por otra reina.

```
1  Algoritmo:
2  - Colocar la reina i en la primera casilla válida de la fila i
3  - Si una reina no puede llegar a colocarse en ninguna casilla, se vuelve atrás y
   se cambia la posición de la reina i-1
4  - Intentar colocar las reinas restantes en las filas que quedan
```

10. Salida de un laberinto: Se trata de encontrar un camino que nos permita salir de un laberinto definido en una matriz  $N \times N$ . Para movernos por el laberinto, sólo podemos pasar de una casilla a otra que sea adyacente a la primera y no esté marcada como una casilla prohibida (esto es, las casillas prohibidas determinan las paredes que forman el laberinto).

- 1 Algoritmo:
- 2 - Se comienza en la casilla (0,0) y se termina en la casilla (N-1,N-1)
- 3 - Nos movemos a una celda adyacente si esto es posible.
- 4 - Cuando llegamos a una situación en la que no podemos realizar ningún movimiento que nos lleve a una celda que no hayamos visitado ya, retrocedemos sobre nuestros pasos y buscamos un camino alternativo.

### 3. Fuentes de información

---

- [Wikipedia](#)
- [Programación \(Grado Superior\) - Juan Carlos Moreno Pérez \(Ed. Ra-ma\)](#)
- Apuntes IES Henri Matisse (Javi García Jimenez?)
- Apuntes AulaCampus
- [Apuntes José Luis Comesaña](#)
- [Apuntes IOC Programació bàsica \(Joan Arnedo Moreno\)](#)
- [Apuntes IOC Programació Orientada a Objectes \(Joan Arnedo Moreno\)](#)