# Prog. Básica- Laboratorio 6
# Vectores y matrices

**NOTAS:**

1. Cuando uno de los ejercicios que se proponen carece de programa de prueba o de plantilla, significa que lo tenéis que hacer vosotros desde cero. Además, el hecho de que se proporcionen algunos casos de prueba no significa que no pueda faltar alguno. Tendréis que añadir los casos de prueba que falten.
2. En este laboratorio no hay que completar ningún informe. Solamente tendréis que subir a eGela los ficheros fuente (*.adb* exportados de Eclipse y *.py* obtenidos de CodeSkulptor) comprimidos en un fichero (.zip preferiblemente) que deberá ajustarse a las reglas de nombrado exigidas hasta ahora (por ejemplo, JSaez_PEtxebarria_lab6.zip).
3. Se presupone que los ejercicios son correctos, es decir, que no tienen errores de compilación y que funcionan correctamente. Esto significa que las soluciones de los ejercicios no puntúan nada por ser correctas, pero penalizan si tienen errores o no se ajustan a lo que se pide. Una vez que la solución es correcta, lo que se evalúa (lo que puntúa) son:
    a. Los casos de prueba: ¿Se han contemplado todos los casos de prueba, desde los generales a los más críticos? Se dan algunos pero faltan muchos otros.
    b. La eficiencia: ¿Se utilizan los "chivatos" cuando hace falta? ¿No hay asignaciones innecesarias? ¿No hay condicionales que no deberían ejecutarse?, ¿Se definen solamente los parámetros o variables necesarios?, etcétera.
    c. La claridad: ¿El código está tabulado? ¿Los nombres de las variables ayudan a entender el código? ¿Hay un único *return* al final de la función?, etcétera.

## 1º ejercicio: Búsqueda en un vector

**Este ejercicio hay que resolverlo en ADA y en PYTHON.**
**Fichero (ADA):** *vectores.ads* (No hay que modificarlo).
**Plantillas (ADA):** *esta_en_vector.adb* y *prueba_esta_en_vector.adb*.
**Plantilla (PYTHON):** *esta_en_vector.py*.

Dado un valor entero y un vector de N enteros cuyos valores NO están ordenados, hacer un subprograma que diga si el entero está o no en el vector. VECTOR COMPLETO, DE N ELEMENTOS.

## 2º ejercicio: Búsqueda en un vector ordenado

**Este ejercicio hay que resolverlo en ADA y en PYTHON.**
**Fichero (ADA):** *vectores.ads* (No hay que modificarlo).
**Plantillas (ADA):** *esta_en_vector_ordenado.adb* y *prueba_esta_en_vector_ordenado.adb*.
**Plantilla (PYTHON):** *esta_en_vector_ordenado.py.*

Dado un valor entero y un vector de N enteros cuyos valores están ordenados de manera ascendente, hacer un subprograma que diga si el entero está o no en el vector. VECTOR COMPLETO, DE N ELEMENTOS ORDENADOS.

NOTA: Es imprescindible que la solución sea eficiente. En concreto, no se seguirá buscando el elemento cuando se sabe objetivamente que no está en el vector porque éste está ordenado.

## 3º ejercicio: Búsqueda de la posición en un vector

**Este ejercicio hay que resolverlo en ADA y en PYTHON.**
**Fichero (ADA):** *vectores.ads* (No hay que modificarlo).
**Plantillas (ADA):** *esta_en_posicion.adb* y *prueba_esta_en_posicion.adb.*
**Plantilla (PYTHON):** *esta_en_posicion.py*

Dado un valor entero y un vector de N enteros, hacer un subprograma que devuelva la posición en la que está el entero en el vector. Si dicho entero estuviera repetido, bastará con devolver la posición de una cualquiera de sus apariciones; y en caso de no encontrarse en el vector, se devolverá el valor -1. VECTOR COMPLETO, DE N ELEMENTOS.

## 4º ejercicio: Rotación de un vector hacia la derecha

**Este ejercicio hay que resolverlo en ADA y en PYTHON.**
**Fichero (ADA):** *vectores.ads* (No hay que modificarlo).
**Plantillas (ADA):** *escribir_vector.adb, rotar_derecha.adb y prueba_rotar_derecha.adb.*
**Plantilla (PYTHON):** *rotar_derecha.py.*

Dado un vector de N enteros, hacer un subprograma que rote sus elementos a la derecha, de modo que el último elemento pase a ser el primero, como se muestra en el ejemplo siguiente. VECTOR COMPLETO, DE N ELEMENTOS.

Ejemplo de entrada:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| 34 | 67 | 45 | 23 | 78 | 12 | 40 | 55 | 24 | 89 |

Salida:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| 89 | 34 | 67 | 45 | 23 | 78 | 12 | 40 | 55 | 24 |

## 5º ejercicio: Borrado del tercer elemento en lista no ordenada

**Este ejercicio únicamente hay que resolverlo en ADA.**
**Ficheros (ADA):** *vectores.ads* y *escribir_lista.adb* (No hay que modificarlos).
**Plantillas (ADA):** *eliminar_tercer_elemento.adb* y *prueba_eliminar_tercer_elemento.adb.*

Dado un vector de como máximo N elementos, hacer un subprograma que elimine su tercer elemento (si existe). El vector puede tener menos de N elementos, y para gestionar esta situación se encapsulará en un registro de tipo Lista_Enteros (véase *vectores.ads*). Si la lista tiene menos de 3 elementos, entonces no se hará nada.

Los elementos de la lista no tienen por qué estar ordenados (ni antes ni después del borrado), lo que se puede aprovechar para ganar en eficiencia al eliminar el tercer elemento. VECTOR NO NECESARIAMENTE COMPLETO, DE <=N ELEMENTOS NO NECESARIAMENTE ORDENADOS.

NOTA: Es imprescindible que la solución sea eficiente. En concreto, se aprovechará que el orden de los elementos de la lista es indiferente (tanto antes como después de la eliminación del primer elemento) para borrar el tercer elemento sin utilizar bucles.

## 6º ejercicio: Borrado del tercer elemento en una lista ordenada

**Este ejercicio únicamente hay que resolverlo en ADA.**
**Ficheros (ADA):** *vectores.ads* y *escribir_lista.adb* (No hay que modificarlos).
**Plantillas (ADA):** *eliminar_tercer_elemento_ordenada.adb* y *prueba_eliminar_tercer_elemento_ordenada.adb.*
Dada una lista de cómo máximo N elementos que están ordenados de manera ascendente, hacer un subprograma que elimine su tercer elemento (si existe). Este ejercicio es similar al anterior, con la diferencia de que en este caso los elementos de la lista están ordenados antes y deben seguir ordenados después de realizar el borrado del tercer elemento. VECTOR NO NECESARIAMENTE COMPLETO, DE <=N ELEMENTOS ORDENADOS.

## 7º ejercicio: Inserción de un elemento en una posición

**Este ejercicio únicamente hay que resolverlo en ADA.**
**Ficheros (ADA):** *vectores.ads* y *escribir_lista.adb* (No hay que modificarlos).
**Plantillas (ADA):** *insertar_elemento_en_pos.adb* y *prueba_insertar_elemento_en_pos.adb*.

Dado un vector de cómo máximo N-1 elementos (lo que significa que hay hueco para al menos un elemento más), hacer un subprograma que inserte el elemento *Num* en la posición *Pos*, desplazando los elementos que corresponda una posición a la derecha. Para resolver este problema conviene reutilizar el programa *rotar_derecha,* haciéndole una pequeña modificación y añadiéndole como parámetros de entrada la posición *Pos* y el número a insertar *Num*. Igual que en los ejercicios anteriores, también será necesario utilizar una Lista_Enteros en lugar de un Vector_de_Enteros. VECTOR NO COMPLETO, DE <=N-1 ELEMENTOS.

## 8º ejercicio: Búsqueda de la posición en una matriz

**Este ejercicio únicamente hay que resolverlo en PYTHON.**
**Plantilla (PYTHON):** *posicion_en_matriz.py*.

Dado un valor entero y una matriz de N x M enteros (N>0 y M>0), hacer un subprograma que devuelva la posición en la que se encuentra ese valor entero (es decir, sus coordenadas Fila y Columna). Si el entero está repetido, se devolverá la posición de la primera aparición del entero en la matriz, y si no está se devolverá como posición la fila -1 y la columna -1.

## 9º ejercicio: Máximo de una matriz

**Este ejercicio únicamente hay que resolverlo en ADA.**
**Fichero (ADA):** *matrices.ads* (No hay que modificarlo).
**Plantillas (ADA):** *calcular_maximo.adb* y p*rueba_calcular_maximo.adb.*

Dada una matriz de N x M enteros (N>0 y M>0), hacer un subprograma que calcule el valor máximo y la posición en que se encuentra. Si el valor máximo está repetido, habrá que devolver la posición de la primera de sus apariciones en la matriz.

## 10º ejercicio: Ordenación por inserción

**Este ejercicio únicamente hay que resolverlo en PYTHON.**
**Plantilla (PYTHON):** *ordenar_por_insercion.py*

Dado un vector de N enteros (N>0), hacer un subprograma que lo ordene mediante el método de inserción.

- Video ilustrativo: http://www.youtube.com/watch?v=gTxFxgvZmQs&feature=related

Los elementos se toman de uno en uno siguiendo el orden en el que están inicialmente y se insertan en el vector de forma ordenada de manera similar a como se hacía en el ejercicio de *inserción en un vector de un elemento en una posición*. Al comienzo de la *i*-ésima iteración, los primeros *i*-1 elementos del vector están ya ordenados; se toma el elemento actual (el de la posición *i*-1, recuérdese que en Python se comienza a contar desde 0) y se compara con los elementos ya ordenados para determinar cuál es el lugar que le corresponde. Como consecuencia, en la mayoría de las iteraciones habrá que hacer un hueco en el vector, desplazando algunos elementos hacia la derecha, para poder insertar el elemento actual en la posición correspondiente.

Para realizar este ejercicio se pide implementar los siguientes subprogramas:

- *buscar_posicion_de_insercion*. Recibe como parámetros el *vector V*, la *posición actual* (esto es, la que delimita los elementos que hay ya ordenados en las primeras posiciones del vector) y el *elemento actual* (es decir, el número que se quiere insertar); y devuelve la posición que le correspondería en el vector al elemento actual (entre 0 y *posición actual*-1).
- *desplazar_una_posicion_a_la_derecha.* Recibe como parámetros el *vector V* y dos enteros que representan sendas posiciones: la *posición actual* (que igual que antes se corresponde con la cantidad de elementos que hay ya ordenados en las primeras posiciones del vector) y la *posición de comienzo* (que será la posición en la que después se insertará el elemento actual). Devuelve el vector con todos los elementos que hay entre la *posición de comienzo* y la *posición actual*-1 desplazados una posición a la derecha. Esto implica que el valor de la *posición actual* (que es precisamente el elemento actual) se sobrescribe, por lo que antes de llamar a esta función conviene haberlo guardado en una variable auxiliar.

**Ejemplo**:

Lista inicial que se quiere ordenar (10 elementos):

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 9 | 5 | 3 | 4 | 10 | 8 | 13 | 24 | 15 | 11 |

Inserción 1: se inserta el primer elemento (9) en la posición 0; el vector no cambia (posActual=1):

| **0** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| **9** | 5 | 3 | 4 | 10 | 8 | 13 | 24 | 15 | 11 |

Inserción 2: se inserta el segundo elemento (5) en la posición 0, desplazando el 9 (posActual=2)

| **0** | **1** | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| **5** | **9** | 3 | 4 | 10 | 8 | 13 | 24 | 15 | 11 |

Inserción 3: se inserta el tercer elemento (3) en la posición 0, desplazando el 5 y el 9 (posActual=3):

| **0** | **1** | **2** | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| **3** | **5** | **9** | 4 | 10 | 8 | 13 | 24 | 15 | 11 |

Inserción 4: se inserta el cuarto elemento (4) en la posición 1, desplazando el 5 y el 9 (posActual=4):

| **0** | **1** | **2** | **3** | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| **3** | **4** | **5** | **9** | 10 | 8 | 13 | 24 | 15 | 11 |

Inserción 5: se inserta el quinto elemento (10) en la posición 4, sin desplazar nada (posActual= 5):

| **0** | **1** | **2** | **3** | **4** | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| **3** | **4** | **5** | **9** | **10** | 8 | 13 | 24 | 15 | 11 |

Etcétera...

NOTA: Es imprescindible que la solución funcione para cualquier número de elementos (N>0).

# 11º ejercicio: Ordenación por selección

**Este ejercicio únicamente hay que resolverlo en PYTHON.**
**Plantilla (PYTHON):** *ordenar_por_seleccion.py*

Dado un vector de N enteros (N>0), hacer un subprograma que lo ordene mediante el método de selección.

- Video ilustrativo: http://www.youtube.com/watch?v=boOwArDShLU

Se trata de buscar el menor elemento del vector e intercambiarlo con el elemento que ocupa la primera posición. Después se busca el menor del resto del vector y se intercambia con el que esté ocupando la segunda posición. En general, al comienzo de la *i*-ésima iteración, los primeros *i*-1 elementos del vector están ya ordenados en su posición definitiva; se busca el elemento menor entre la posición actual (es decir, la posición *i*-1, recuérdese que en Python se comienza a contar desde 0) y el final del vector, y se intercambia con el que actualmente ocupa la posición *i*-1.

Para realizar este ejercicio se pide implementar los siguientes subprogramas:

- *buscar_posicion_del_minimo*. Recibe como parámetros el *vector V* y la *posición de comienzo* (que corresponderá con la posición actual, esto es, la que delimita los elementos que hay ya ordenados en las primeras posiciones del vector). Devuelve la posición en la que se encuentra el valor mínimo comprendido entre la *posición de comienzo* y el final del vector V.
- *intercambiar*. Recibe como parámetros el *vector V* y dos enteros que representan sendas posiciones: *posA* y *posB*. Devuelve el vector V con los valores de las posiciones *posA* y *posB* intercambiados.

**Ejemplo**:

Lista inicial que se quiere ordenar (10 elementos):

| **0** | 1 | **2** | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 9 | 5 | **3** | 4 | 10 | 8 | 13 | 24 | 15 | 11 |

El menor entre la posición 0 y la última es 3 (posición 2); se intercambian las posiciones 0 y 2:

| **0** | 1 | 2 | **3** | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| **3** | 5 | 9 | **4** | 10 | 8 | 13 | 24 | 15 | 11 |

El menor entre la posición 1 y la última es 4 (posición 3); se intercambian las posiciones 1 y 3:

| **0** | **1** | 2 | **3** | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| **3** | **4** | 9 | **5** | 10 | 8 | 13 | 24 | 15 | 11 |

El menor entre la posición 2 y la última es 5 (posición 3); se intercambian las posiciones 2 y 3:

| **0** | **1** | **2** | 3 | 4 | **5** | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| **3** | **4** | **5** | 9 | 10 | **8** | 13 | 24 | 15 | 11 |

El menor entre la posición 3 y la última es 8 (posición 5); se intercambian las posiciones 3 y 5:

| **0** | **1** | **2** | **3** | 4 | **5** | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| **3** | **4** | **5** | **8** | 10 | **9** | 13 | 24 | 15 | 11 |

El menor entre la posición 4 y la última es 9 (posición 5); se intercambian las posiciones 4 y 5:

| **0** | **1** | **2** | **3** | **4** | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| **3** | **4** | **5** | **8** | **9** | 10 | 13 | 24 | 15 | 11 |

Etcétera...

NOTA: Es imprescindible que la solución funcione para cualquier número de elementos (N>0).

# 12º ejercicio: Ordenación por el método de la burbuja (eficiente)

**Este ejercicio únicamente hay que resolverlo en PYTHON.**
**Plantilla (PYTHON):** *ordenar_por_burbuja.py*

Dado un vector de N enteros (N>0), hacer un subprograma que lo ordene mediante el método de la burbuja de manera eficiente.

- Video ilustrativo: http://www.youtube.com/watch?v=1JvYAXT_064&feature=related

El algoritmo puede ser el siguiente:

```
num_recorridos:=1;
repetir salir si num_recorridos > num_elementos-1;
     i:=1;
     repetir salir si i> num_elementos-1;
          si V(i) > V(i+1) entonces ---intercambiarlos
               aux:=V(i);
               V(i):=V(i+1);
               V(i+1):=aux;
          fin_si;
          i:=i+1;
     fin_repetir;
     num_recorridos:= num_recorridos+1;
fin_repetir;
```

**Ejemplo**:

Lista inicial que se quiere ordenar (4 elementos):

| | | | |
|---|---|---|---|
| 7 | 5 | 1 | 2 |

Despues de la primera iteración, el último elemento ya está ordenado.

| | | | |
|---|---|---|---|
| 5 | 1 | 2 | 7 |

Después de dos iteraciones hay dos elementos ordenados.

| | | | |
|---|---|---|---|
| 1 | 2 | 5 | 7 |

Y después del tercer y último recorrido el vector quedaría igual.

| | | | |
|---|---|---|---|
| 1 | 2 | 5 | 7 |

En la última iteración no se ha hecho nada, porque la lista ya estaba ordenada. Sin embargo, en el peor de los casos (a saber, en el caso en el que la lista inicial hubiese estado TOTALMENTE desordenada) hubiera hecho falta esta vuelta. Como de antemano no se puede saber si la lista está o no totalmente desordenada, el algoritmo realiza el proceso completo, esto es, con todas las vueltas necesarias para el caso peor.

Se pide que vuestra solución mejore la eficiencia del algoritmo anterior, **añadiendo un booleano** tal que si no se ha hecho ningún intercambio en la iteración actual se salga del bucle, dado que esto significará que el vector ya está ordenado.

NOTA: Es imprescindible que la solución funcione para cualquier número de elementos (N>0).