1 Introducción

El objetivo de este capítulo es ver los conceptos básicos de programación que luego utilizaremos a lo largo de todo el curso. Esto incluye ver tanto los datos que vamos a manejar, como las instrucciones que operan sobre ellos, así como las técnicas básicas de programación. Para la implementación de los algoritmos que vamos a ver utilizaremos el lenguaje Python, sin perjuicio de que los conceptos que veremos son independientes del lenguaje específico utilizado.

En primer lugar, veremos algunos de los tipos de datos básicos de Python. Las variables cambian de tipo dinámicamente según el tipo del valor que tengan en un instante dado.

Datos numéricos

Python distingue entre números enteros (int) y números de punto flotante (float).

In [1]:

```
n=5 # int
print(n, type(n))
x=3.14 # float
print(x, type(x))
```

```
5 <class 'int'>
3.14 <class 'float'>
```

Sobre estos datos se pueden ejecutar las operaciones aritméticas habituales.

In [2]:

```
print(n+1)
print(n/2) # noten la diferencia con Python 2.7
print(n//2) # división entera
print(x**2)
```

```
6
2.5
2
9.8596
```

Datos lógicos

Los valores de verdad son `True` y `False`. Las operaciones lógicas se indican con palabras en lugar de símbolos.

In [3]:

```
t=True
f=False
print(t and f)
print(t and not f)
print(n>0)
```

```
False
True
True
```

Strings

Los strings se escriben entre comillas simples o dobles, y existen muchas operaciones definidas para ellos.

In [4]:

```
h="Hola"
print(h)
print(len(h))
m='mundo'
print(h + " " + m)
print(h.upper())
```

```
Hola
4
Hola mundo
HOLA
```

Listas

Una lista es una secuencia de datos, posiblemente de distintos tipos, de largo variable.

In [5]:

```
L=[3,2,1]
print(L)
L.append(0)
print(L)
x=L.pop()
print(L)
print(x)
```

```
[3, 2, 1]
[3, 2, 1, 0]
[3, 2, 1]
0
```

Los elementos de la lista se indexan partiendo desde cero.

In [6]:

```
print(L[0])
print(L[2])
print(L[-1]) # contando desde el extremo derecho
```

```
3
1
1
```

Una lista se puede construir en base a iterar una fórmula:

In [7]:

```
C = [n**2 for n in range(1,7)]
print(C)
```

```
[1, 4, 9, 16, 25, 36]
```

Una lista se puede recorrer a través de sus subíndices:

In [8]:

```
for i in range(0,len(C)):
    print(C[i])
```

```
1
4
9
16
25
36
```

O bien iterando sobre los elementos de la lista:

In [9]:

```
for v in C:
    print(v)
```

```
1
4
9
16
25
36
```

Funciones predefinidas

Hay funciones que están disponibles para ser utilizadas:

In [10]:

```
import math
a=3
b=4
c=math.sqrt(a**2+b**2)
print(c)
```

```
5.0
```

Funciones

Uno puede definir sus propias funciones:

In [11]:

```
def hipotenusa(a,b):
    import math
    c=math.sqrt(a**2+b**2)
    return c
print(hipotenusa(3,4))
```

```
5.0
```

Expresiones condicionales

Una expresión de la forma

```
valor_si_verdadero if condicion else valor_si_falso
```

permite por ejemplo definir funciones por tramos:

In [12]:

```
def valor_absoluto(x):
    return -x if x<0 else x
print(valor_absoluto(-5), valor_absoluto(5))
```

```
5 5
```

Diccionarios

Un diccionario es similar a una lista, pero los elementos se indexan con datos no necesariamente numéricos. Por ejemplo, supongamos que queremos convertir "piedra", "papel" y "tijera" a una representación numérica 0, 1, 2, respectivamente:

In [13]:

```
p_a_n = {"piedra":0, "papel":1, "tijera":2} # para convertir de palabra a número
print(p_a_n["papel"])
```

```
1
```

La conversión inversa se puede hacer simplemente con una lista:

In [14]:

```
n_a_p = ["piedra", "papel", "tijera"] # para convertir de número a palabra
print(n_a_p[1])
```

```
papel
```

Con esto podemos hacer una función en que el programa juegue contra el usuario:

In [15]:

```
def juega():
    from random import randint
    programa=randint(0,2)
    usuario=int(p_a_n[input("¿piedra, papel o tijera? ")])
    resultado="Empate" if programa==usuario else\
        "Gana Usuario" if (programa+1)%3==usuario else "Gana Programa"
    print("Usuario juega", n_a_p[usuario])
    print("Programa juega", n_a_p[programa])
    print("Resultado:", resultado)
juega()
```

```
¿piedra, papel o tijera? tijera
Usuario juega tijera
Programa juega tijera
Resultado: Empate
```

Instrucción condicional: if

La instrucción `if` permite elegir entre distintas alternativas de instrucciones a ejecutar.

In [16]:

```
# Encontrar el máximo entre dos valores
def max2(a, b):
    if a>b:
        m=a
    else:
        m=b
    return m
print(max2(3,7))
```

```
7
```

Esto se puede generalizar a 3 valores, pero el resultado no es muy elegante:

In [17]:

```
# Encontrar el máximo entre tres valores
def max3(a, b, c):
    if a>b:
        if a>c:
            m=a
        else:
            m=c
    else:
        if b>c:
            m=b
        else:
            m=c
    return m
print(max3(4,3,7))
```

```
7
```

Una mejor alternativa la podemos obtener si vamos obteniendo aproximaciones sucesivas al máximo:

In [18]:

```
# Encontrar el máximo entre dos valores
def max2(a, b):
    m=a
    if b>m:
        m=b
    return m
print(max2(3,7))
```

```
7
```

Esto se generaliza de manera mucho más simple a tres (o más) valores:

In [19]:

```
# Encontrar el máximo entre tres valores
def max3(a, b, c):
    m=a
    if b>m:
        m=b
    if c>m:
        m=c
    return m
print(max3(4,3,7))
```

```
7
```

Cuando hay preguntas encadenadas, se puede usar la cláusula `elif` (abreviatura de `else if`, pero que no abre un nuevo nivel de indentación):

In [20]:

```
# Dice si un año dado es bisiesto
def es_bisiesto(a):
    if a%400==0:   # Los múltiplos de 400 siempre son bisiestos
        b=True
    elif a%100==0: # Los demás múltiplos de 100 no son bisiestos
        b=False
    elif a%4==0:   # De los restantes, los múltiplos de 4 son bisiestos
        b=True
    else:          # Cualquier otro año no es bisiesto
        b=False
    return b
print(es_bisiesto(1900), es_bisiesto(2000), es_bisiesto(2020))
```

```
False True True
```

Instrucciones iterativas: for

La instrucción `for` itera con una variable que toma valores de una lista dada. A menudo, esa lista se especifica mediante `range`:

In [21]:

```
# Encuentra el máximo de una lista a
def maximo(a):
    m=a[0]
    # Al comenzar cada iteración, se cumple que m==max(a[0],...,a[k-1])
    for k in range(1,len(a)):
        if a[k]>m:
            m=a[k]
    return(m)
print(maximo([25, 42, 93, 17, 54, 28]))
```

```
93
```

El comentario que aparece junto al `for` describe lo que se llama el **invariante** del ciclo, y es muy útil para poder argumentar la correctitud del programa, así como para poder ayudar a su diseño.

El invariante una afirmación lógica que debe ser verdadera cada vez que se intenta iniciar una nueva iteración, lo cual incluye tanto la primera vez como el último intento, en que se detecta que el rango ya se ha agotado y el ciclo termina.

- La validez del invariante la primera vez la deben asegurar las instrucciones previas al ciclo, que se llaman instrucciones de *inicialización*. En este ejemplo, al comenzar el ciclo se tiene que $k = 1$, y por lo tanto trivialmente se cumple que $m = \max(a[0])$.
- Las instrucciones al interior del ciclo (el "cuerpo de ciclo") parten de la premisa de que el invariante se cumple al inicio, y deben garantizar que se cumpla al final (para dejar todo listo para la próxima iteración). Esto se llama *preservar el invariante*. En este ejemplo, para asegurar que $m = \max(a[0], \ldots, a[k])$ sabiendo que ya era cierto que $m = \max(a[0], \ldots, a[k-1])$, se debe modificar m solo en el caso de que $a[k]$ sea mayor que m, o dejarlo igual si no.
- Cuando se detecta que el rango se ha agotado, el invariante igualmente se cumple, y ambas cosas juntas deben asegurar que haya logrado el objetivo final deseado. En este ejemplo, cuando k llega a ser igual a $len(a)$, el invariante implica que $m = \max(a[0], \ldots, a[len(a)-1])$, o sea, es el máximo de toda la lista.

Instrucciones iterativas: while

La instrucción `while` ejecuta instrucciones mientra la condición especificada sea verdadera:

In [22]:

```
# Dice si un número dado es primo o no
def es_primo(n):
    if n==2:
        return True # 2 es primo
    if n%2==0:
        return False # ningún otro par es primo
    k=3
    while k**2<=n: # no es necesario buscar posibles factores más allá de sqrt(n
)
        if n%k==0:
            return False
        k+=2 # probamos solo los impares
    # si no se encontró ningún factor menor que raíz de n, es primo
    return True
print(es_primo(2), es_primo(7), es_primo(9), es_primo(79823492843), es_primo(798
23492869))
```

```
True True False False True
```

Ejemplo de programación con invariantes: Calcular $y = x^n$

Supongamos que no tuviéramos una operación de elevación a potencia, y que necesitáramos calcular x^n para n entero no negativo. El algoritmo obvio es calcular $x * x * \cdots * x$ (n veces):

In [23]:

```
def potencia(x, n):
    y=1
    for k in range(0,n):
        y*=x
    return y
```

In [24]:

```
print(potencia(2,10))
```

```
1024
```

El invariante, esto es, lo que se cumple al comenzar cada nueva iteración es $y = x^k$. Así, al inicio, cuando $k = 0$, se tiene $y = 1$ (inicialización), y al término, cuando $y = n$, se tiene la condición final buscada. La preservación del invariante consiste en multiplicar y por x, porque así se sigue cumpliendo el invariante cuando k se incrementa en 1.

Este algoritmo demora un tiempo proporcional a n, lo cual escribiremos $O(n)$ y lo leeremos "del orden de n". (Más adelante definiremos precisamente esta notación, y veremos que podríamos ser más precisos todavía al describir el tiempo que demora un algoritmo)

¿Será posible calcular una potencia de manera más eficiente?

Para ver cómo podríamos mejorar el algoritmo, comenzaremos por reescribirlo de modo que la variable k vaya disminuyendo en lugar de ir aumentando, usando para ello la instrucción `while` :

In [25]:

```
def potencia(x, n):
    y=1
    k=n
    while k>0:
        y*=x
        k-=1
    return y
```

In [26]:

```
print(potencia(2,10))
```

```
1024
```

El invariante en este caso sería $y = x^{n-k}$ o, lo que es lo mismo, $y * x^k = x^n$.

El reescribirlo de esta manera nos permite hacer el siguiente truco: vamos a introducir una variable z, cuyo valor inicial es x, y reformular el invariante como $y * z^k = x^n$:

In [27]:

```
def potencia(x, n):
    y=1
    k=n
    z=x
    while k>0:
        y*=z
        k-=1
    return y
```

In [28]:

```
print(potencia(2,10))
```

```
1024
```

Este cambio podría parecer ocioso, pero gracias a él ahora tenemos un grado adicional de libertad: en efecto, podemos modificar la variable z en la medida que eso no haga que el invariante deje de cumplirse.

En particular, una oportunidad de hacer esto aparece cuando k es par. En ese caso, si elevamos z al cuadrado y al mismo tiempo dividimos k a la mitad, ambos cambios se complementan para hacer que el invariante se preserve:

In [29]:

```
def potencia(x, n):
    y=1
    k=n
    z=x
    while k>0:
        if k%2==0: # caso k par
            z=z*z
            k//=2
        else:      # caso k impar
            y*=z
            k-=1
    return y
```

In [30]:

```
print(potencia(2,10))
```

```
1024
```

Este algoritmo admite todavía una pequeña optimización. Cuando k se divide por 2, no solo se preserva el invariante, sino que además k sigue siendo > 0, y por lo tanto no es necesario en ese caso volver a preguntar por la condición del `while`. El algoritmo queda como sigue:

In [31]:

```
def potencia(x, n):
    y=1
    k=n
    z=x
    while k>0:
        while k%2==0: # caso k par
            z=z*z
            k//=2
        y*=z # aquí estamos seguros que k es impar
        k-=1
    return y
```

In [32]:

```
print(potencia(2,10))
```

```
1024
```

Este algoritmo (en cualquiera de las dos últimas versiones) se llama el *algoritmo binario*, y es mucho más eficiente que el algoritmo inicial. Cada vez que se da el caso par, k disminuye a a mitad, y eso ocurre al menos la mitad de las veces. Pero si k comienza con el valor n, la operación de dividir por 2 se puede ejecutar a lo más $\log_2 n$ veces, por lo tanto el tiempo total de ejecución es $O(\log_2 n)$, en lugar de $O(n)$.

Decimos que el algoritmo original era de tiempo lineal, y que el algoritmo binario es de tiempo logarítmico. Para n grande, la diferencia de eficiencia es muy grande en favor del algoritmo binario.

Una observación importante es que para que el algoritmo binario funcione, solo es necesario que x y, z pertenezcan a un conjunto para el cual hay definida una operación multiplicativa que sea asociativa y que tenga un elemento neutro. Por lo tanto, este algoritmo no solo sirve para elevar a potencia números enteros o reales, sino que además, por ejemplo, matrices.

Numpy y Arreglos

Numpy es la principal biblioteca para computación científica en Python.

Una de las características de Numpy es que provee arreglos multidimensionales de alta eficiencia. Mientras la gran flexibilidad de las listas de Numpy puede hacer que no sea muy eficiente el acceso a elementos específicos, los arreglos de Numpy aseguran el acceso a cada elemento en tiempo constante. Por esa razón, utilizaremos estos arreglos cuando necesitemos asegurar la eficiencia de la implementación de los algoritmos.

In [33]:

```
import numpy as np

a = np.array([6.5, 5.2, 4.6, 7.0, 4.3])
print(a[2])
```

```
4.6
```

In [34]:

```
print(len(a))
```

```
5
```

Hay varias formas de crear arreglos inicializados con ceros, unos, valores constantes o valores aleatorios.

In [35]:

```
b = np.ones(10)
print(b)
```

```
[1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
```

In [36]:

```
c = np.zeros(7)
print(c)
```

```
[0. 0. 0. 0. 0. 0. 0.]
```

In [37]:

```
c = np.full(5, 2)
print(c)
```

```
[2 2 2 2 2]
```

In [38]:

```
d = np.random.random(6)
print(d)
```

```
[0.12499846 0.05138973 0.89319876 0.50248684 0.51511381 0.31853868]
```

También es posible crear y manejar arreglos de varias dimensiones.

In [39]:

```
a = np.array([[1,2,3],[4,5,6]])
print(a)
```

```
[[1 2 3]
 [4 5 6]]
```

In [40]:

```
print(a[0,2])
```

```
3
```

In [41]:

```
(m,n)=np.shape(a)
print(m,n)
```

```
2 3
```

In [42]:

```
b = np.zeros((3,3))
print(b)
```

```
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
```

In [43]:

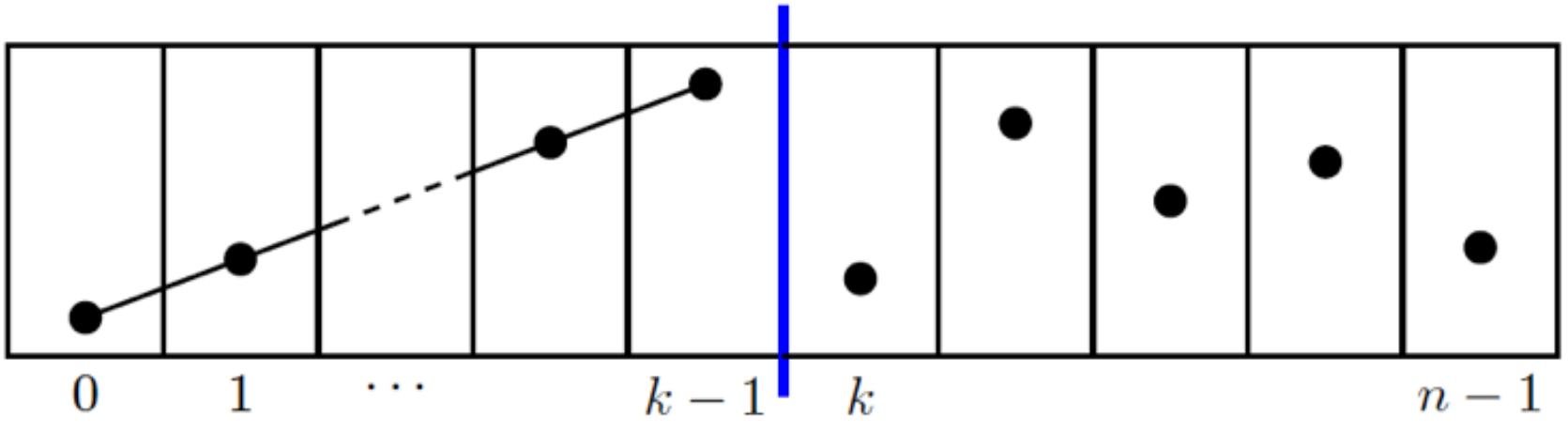
```
c = np.eye(3)
print(c)
```

```
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

Ejemplo: Ordenación por inserción

Supongamos que se tiene un arreglo a, de tamaño n, y queremos reordenar los datos almacenados en su interior de modo que queden en orden ascendente.

El método de **Ordenación por Inserción** se basa en formar en el sector izquierdo del arreglo un subconjunto ordenado, en el cual se van insertando uno por uno los elementos restantes. Para la inicialización, comenzamos con un subconjunto ordenado de tamaño 0, y el proceso termina cuando el sunconjunto ordenado llega a tener tamaño n. El invariante se puede visualizar como:



La variable k indica el tamaño del subconjunto ordenado. Equivalentemente, k es el subíndice del primer elemento todavía no ordenado, y que será el que se insertará en esta oportunidad.

In [44]:

```
# Ordenación Por Inserción
def ordena_insercion(a):

    n=len(a)
    for k in range(0,n):
        insertar(a,k)
```

Este algoritmo todavía no es ejecutable, porque falta definir la función `insertar`, que se encarga de tomar $a[k]$ e insertarlo entre los anteriores. La forma más simple de hacer esto es a través de intercambios sucesivos:

In [45]:

```
# Insertar a[k] entre los elementos anteriores preservando el orden ascendente (
versión 1)
def insertar(a, k):
    j=k # señala la posición del elemento que está siendo insertado
    while j>0 and a[j]<a[j-1]:
        (a[j], a[j-1]) = (a[j-1], a[j])
        j-=1
```

Para poder probar este algoritmo, generemos un arreglo con números aleatorios y ordenémoslo:

In [46]:

```
a = np.random.random(6)
print(a)
ordena_insercion(a)
print(a)
```

```
[0.70772925 0.40782006 0.03698861 0.76607866 0.02780469 0.26157451]
[0.02780469 0.03698861 0.26157451 0.40782006 0.70772925 0.76607866]
```

Si observamos el algoritmo de inserción, podemos ver que en los intercambios siempre uno de los dos elementos involucrados es el que se está insertando, el cual pasa por muchos lugares provisorios hasta llegar finalmente a su ubicación definitiva. Esto sugiere que podemos ahorrar trabajo si en lugar de hacer todos esos intercambios, sacamos primero el elemento a insertar hacia una variable auxiliar, luego vamos moviendo los restantes elementos hacia la derecha, y al final movemos directamente el nuevo elemento desde la variable auxiliar hasta su posición definitiva:

In [47]:

```
# Insertar a[k] entre los elementos anteriores preservando el orden ascendente (
versión 2)
def insertar(a, k):
    b=a[k] # b almacena transitoriamente al elemento a[k]
    j=k # señala la posición del lugar "vacío"
    while j>0 and b<a[j-1]:
        a[j]=a[j-1]
        j-=1
    a[j]=b
```

In [48]:

```
a = np.random.random(6)
print(a)
ordena_insercion(a)
print(a)
```

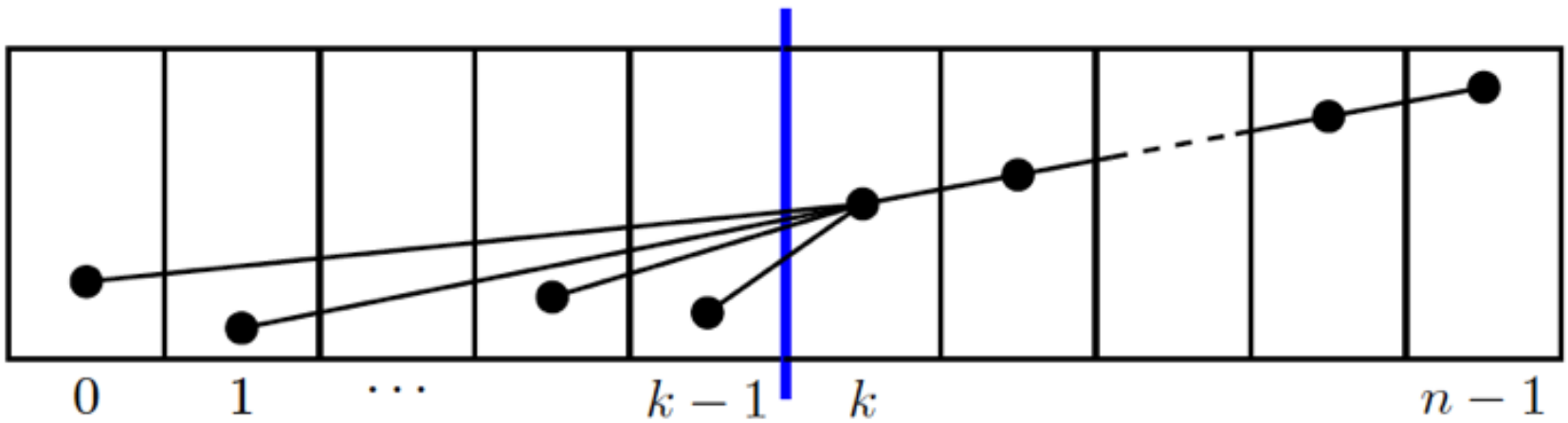
```
[0.93246556 0.69421783 0.12826226 0.81910785 0.20283646 0.99828171]
[0.12826226 0.20283646 0.69421783 0.81910785 0.93246556 0.99828171]
```

Para analizar la eficiencia de este algoritmo, podemos considerar varios casos:

- Mejor caso: Si el arreglo ya viene ordenado, el ciclo de la función `insertar` termina de inmediato, así que esa función demora tiempo constante, y el proceso completo demora tiempo $O(n)$, lineal en n.
- Peor caso: Si el arreglo viene originalmente en orden decreciente, el ciclo de la función `insertar` hace el máximo de iteraciones (k), y la suma de todos esos costos da un total de $O(n^2)$, cuadrático en n.
- Caso promedio: Si el arreglo viene en orden aleatorio, el número promedio de iteraciones que hace el ciclo de la función `insertar` es aproximadamente $k/2$, y la suma de todos esos costos igual da un total de $O(n^2)$. Esto es, el costo promedio también es cuadrático.

Ejemplo: Ordenación por Selección

El método de **Ordenación por Selección** se basa en extraer el máximo elemento y moverlo hacia el extremo derecho del arreglo, y repetir este proceso entre los elementos restantes hasta que todos hayan sido extraídos. El invariante se puede visualizar como:



La variable k indica el tamaño del subconjunto que todavía falta por procesar. Equivalentemente, es el subíndice del primer elemento que ya pertenece al subconjunto ordenado.

In [49]:

```
# Ordenación por Selección
def ordena_seleccion(a):
    n=len(a)
    for k in range(n,1,-1): # Paramos cuando todavía queda 1 elemento "desordena
do" (¿por qué está bien eso?)
        j=pos_maximo(a,k) # Encuentra posición del máximo entre a[0],...,a[k-1]
        (a[j],a[k-1])=(a[k-1],a[j])
```

In [50]:

```
# Encuentra posición del máximo entre a[0],...,a[k-1]
def pos_maximo(a, k):
    j=0 # j señala la posición del máximo
    for i in range(1,k):
        if a[i]>a[j]: # Encontramos un nuevo máximo
            j=i
    return j
```

Nuevamente, probamos nuestro algoritmo con un arreglo aleatorio:

In [51]:

```
a = np.random.random(6)
print(a)
ordena_seleccion(a)
print(a)
```

```
[0.17432399 0.92835432 0.93659177 0.18106299 0.97949602 0.66237785]
[0.17432399 0.18106299 0.66237785 0.92835432 0.93659177 0.97949602]
```

En este algoritmo, siempre se recorre todo el conjunto de tamaño k para encontrar el máximo, de modo que la suma de todos estos costos de un total de $O(n^2)$, en todos los casos.

Más adelante veremos que hay maneras mucho más eficientes de calcular el máximo de un conjunto, una vez que se ha encontrado y extraído el máximo la primera vez.

Piensen por ejemplo en un típico torneo de tenis, en donde los jugadores se van eliminando por rondas, hasta que en la final queda solo un jugador invicto: el campeón. Si hay n jugadores, ese proceso requiere exactamente $n-1$ partidos. **Pero** una vez que se ha jugado todo ese torneo, hagamos un experimento mental y pensemos que habría sucedido si el primer día el campeón no hubiera podido jugar por alguna causa. Para determinar quién habría resultado campeón en esas circunstancias (o sea,, para encontrar al subcampeón), **no sería necesario repetir todo el torneo, sino solo volver a jugar los partidos en los que habría participado el campeón**. Ese número de partidos es mucho menor a n, y en realidad no es difícil ver que es logarítmico. Y eso puede repetirse para encontrar al tercero, al cuarto, etc., siempre con el mismo costo logarítmico.

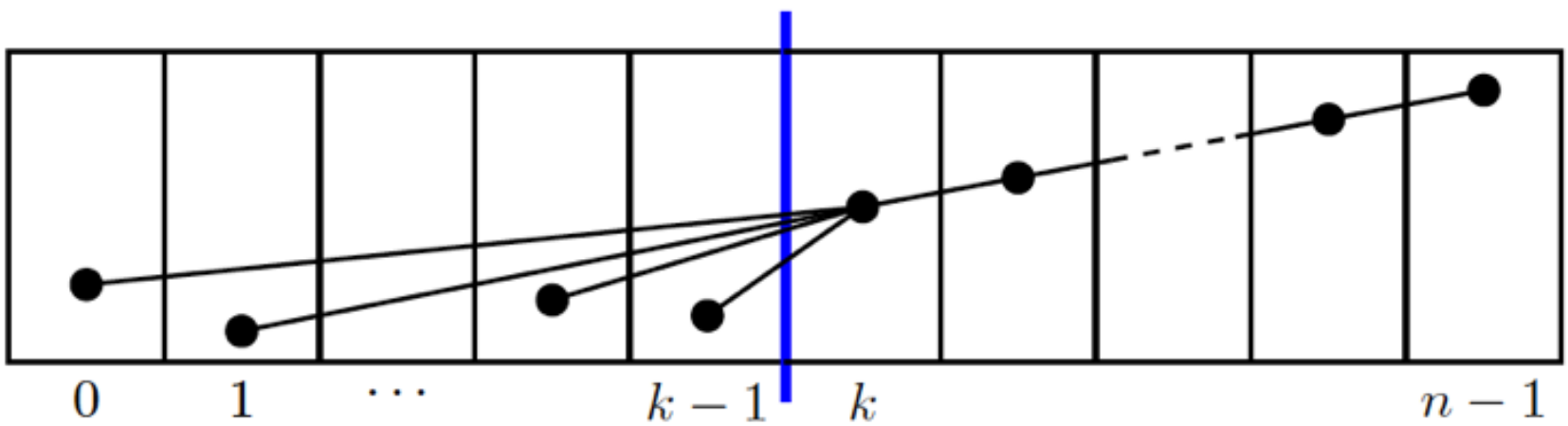
Si sumamos todos esos costos, da un total de $O(n \log n)$, en el peor caso.

Lo anterior es una "demostración de factibilidad" de que existen algoritmos de ordenación de costo $O(n \log n)$, más eficientes que $O(n^2)$. Más adelante en el curso veremos algoritmos prácticos que alcanzan esta eficiencia.

Ejemplo: Ordenación de la Burbuja

Este algoritmo se basa en ir haciendo pasadas sucesivas de izquierda a derecha sobre el arreglo, y cada vez que encuentra dos elementos adyacentes fuera de orden, los intercambia. Así, el arreglo va quedando cada vez "más ordenado", hasta que finalmente esté totalmente ordenado.

Analizando el efecto de una pasada de izquierda a derecha, vemos que, aparte de los pequeños desórdenes que pueda ir arreglando por el camino, una vez que el algoritmo se encuentra con el máximo, los intercambios lo empiezan a trasladar paso a paso hacia la derecha, hasta que finalmente queda en el extremo derecho del arreglo. Eso significa que ya ha llegado a su posición definitiva, y no necesitamos volver a tocarlo. Por lo tanto, el algoritmo puede ignorar esos elementos al extremo derecho, los que por construcción están ordenados y son mayores que todos los de la izquierda. Esto lo podemos visualizar como:



¡El mismo invariante que la Ordenación por Selección! Sin embargo, el algoritmo resultante es distinto.

In [52]:

```
# Ordenación de la Burbuja (versión 1)
def ordena_burbuja(a):
    n=len(a)
    k=n # número de elementos todavía desordenados
    while k>1:
        # Hacer una pasada sobre a[0],...,a[k-1]
        # intercambiando elementos adyacentes desordenados
        for j in range(0,k-1):
            if a[j]>a[j+1]:
                (a[j],a[j+1])=(a[j+1],a[j])
        # Disminuir k
        k-=1
```

In [53]:

```
a = np.random.random(6)
print(a)
ordena_burbuja(a)
print(a)
```

```
[0.80703406 0.08834158 0.71887355 0.26094751 0.68826211 0.12840384]
[0.08834158 0.12840384 0.26094751 0.68826211 0.71887355 0.80703406]
```

Este algoritmo demora siempre tiempo $O(n^2)$, ¡incluso si se le da para ordenar un arregla que ya viene ordenado!

No cuesta mucho introducir una variable booleana que señale si en una pasada no se ha hecho ningún intercambio, y usar esa variable para terminar el proceso cuando eso ocurre. Pero hay una manera mejor de modificar el algoritmo para aumentar su eficiencia.

Para esto, introducimos una variable i que recuerda el punto donde se hizo el último intercambio (el cual habría sido entre $a[i-1]$ y $a[i]$. Si a partir de ese punto ya no se encontraron elementos fuera de orden, eso quiere decir que $a[i-1] < a[i]$ y luego a partir de ahí todos los elementos están ordenados, **hasta el final del arreglo**. Por lo tanto, el invariante se preserva si hacemos $k = i$.

¿Qué pasa si no hubo ningún intercambio? Para este caso, si le damos a la variable i el valor inicial cero, al hacer $k = i$ tendríamos $k = 0$ y el proceso terminaría automáticamente. El algoritmo resultante es el siguiente:

In [54]:

```
# Ordenación de la Burbuja (versión 2)
def ordena_burbuja(a):
    n=len(a)
    k=n # número de elementos todavía desordenados
    while k>1:
        # Hacer una pasada sobre a[0],...,a[k-1]
        # intercambiando elementos adyacentes desordenados
        i=0
        for j in range(0,k-1):
            if a[j]>a[j+1]:
                (a[j],a[j+1])=(a[j+1],a[j])
                i=j+1 # recordamos el lugar del último intercambio
        # Disminuir k
        k=i
```

In [55]:

```
a = np.random.random(6)
print(a)
ordena_burbuja(a)
print(a)
```

```
[0.09630368 0.19007068 0.87057112 0.58364452 0.48520362 0.98393194]
[0.09630368 0.19007068 0.48520362 0.58364452 0.87057112 0.98393194]
```

Este algoritmo aprovecha mejor el orden previo que puede trar el arreglo, y en particular ordena arreglos ordenados en tiempo lineal. Pero tanto su peor caso como su caso promedio siguen siendo cuadráticos.

Recursividad

El poder escribir funciones que se llamen a sí mismas es una herramienta muy poderosa de programación. Veremos algunos ejemplos de aplicaciones de este concepto, y más adelamte veremos cómo puede conducir al diseño de algoritmos muy eficientes.

Ejemplo: Calcular $y = x^n$

Revisemos nuevamente este problema, pero ahora desde un punto de vista recursivo. Una potencia se puede definir recursivamente de la siguiente manera:

$$x^n = \begin{cases} x * x^{n-1} & \text{si } n > 0 \\ 1 & \text{si } n = 0 \end{cases}$$

lo cual se puede implementar directamente como una función recursiva:

In [56]:

```
def potencia(x,n):
    if n==0:
        return 1
    else:
        return x * potencia(x,n-1)
```

In [57]:

```
print(potencia(2,10))
```

```
1024
```

El algoritmo resultante demora tiempo $O(n)$, pero puede mejorarse si el caso n par lo tratamos aparte:

$$x^n = \begin{cases} (x^2)^{n/2} & \text{si } n > 0, \text{par} \\ x * x^{n-1} & \text{si } n > 0, \text{impar} \\ 1 & \text{si } n = 0 \end{cases}$$

y la función que lo implementa es:

In [58]:

```
def potencia(x,n):
    if n==0:
        return 1
    elif n%2==0:
        return potencia(x*x,n//2)
    else:
        return x * potencia(x,n-1)
```

In [59]:

```
print(potencia(2,10))
```

```
1024
```

El resultado es el algoritmo binario, que demora tiempo $O(\log n)$, en versión recursiva.

Recursividad vs. Iteración

Todo algoritmo iterativo puede escribirse recursivamente. En particular, cualquier ciclo de la forma

```
while C:
    A
```

puede implementarse como

```
def f():
    if C:
        A
        f()
f()
```

Por cierto, en la llamada recursiva se le debe entregar a la función el contexto en que habría operado en la siguiente iteración del ciclo.

Por ejemplo, si queremos imprimir uno por uno los elementos de un arreglo a, una forma iterativa de hacerlo sería:

In [60]:

```
def imprimir(a):
    k=0
    while k<len(a):
        print(a[k])
        k+=1
```

In [61]:

```
a=np.random.random(6)
imprimir(a)
```

```
0.3726666475753404
0.7880803913670976
0.039002697603143455
0.31354667739367115
0.2035818587054551
0.23127503030321217
```

En forma recursiva, esto queda como:

In [62]:

```
def imprimir(a):
    imprimir_recursivo(a,0)

def imprimir_recursivo(a,k): # imprimir desde a[k] en adelante
    if k<len(a):
        print(a[k])
        imprimir_recursivo(a,k+1)
```

In [63]:

```
a=np.random.random(6)
imprimir(a)
```

```
0.46102285532290554
0.9588005677031732
0.30558622618653475
0.1802946264459846
0.5144522544729416
0.6520998555996418
```

Este proceso es reversible: cuando una función recursiva lo último que hace es llamarse a sí misma, lo que se llama "recursividad a la cola" ("*tail recursion*"), eso se puede reemplazar por un `while` :

In [64]:

```
def imprimir(a):
    imprimir_recursivo(a,0)

def imprimir_recursivo(a,k): # imprimir desde a[k] en adelante, ahora NO recursi
vo
    while k<len(a): # reemplazó a "if k<len(a):"
        print(a[k])
        k+=1 # reemplazó a "imprimir_recursivo(a,k+1)"
```

In [65]:

```
a=np.random.random(6)
imprimir(a)
```

```
0.12966259005949132
0.20645674763761945
0.6122772354638825
0.1860604200374133
0.6836824042880958
0.05757989702623789
```

Pero ahora la función `imprimir_recursivo` es llamada desde un único lugar, con k=0, y por lo tento, en ese lugar podemos sustituir la llamada por el código de la función, con lo que el resultado es:

In [66]:

```
def imprimir(a):
    # Esto reemplaza a "imprimir_recursivo(a,0)"
    k=0
    while k<len(a): # reemplazó a "if k<len(a):"
        print(a[k])
        k+=1 # reemplazó a "imprimir_recursivo(a,k+1)""
```

In [67]:

```
a=np.random.random(6)
imprimir(a)
```

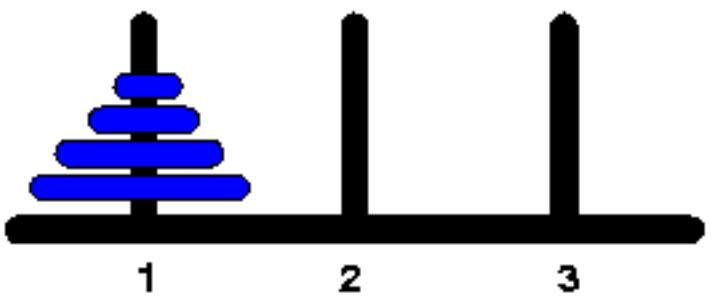
```
0.8088272759548449
0.008143187984316125
0.3029138679261496
0.6161144636878843
0.6148646241483327
0.4358448664697172
```

¡Con lo cual hemos vuelto al punto de partida!

Sin embargo, esto solo funciona para eliminar la "recursividad a la cola". Si hay llamadas recursivas que **no** son lo último que ejecuta la función, no pueden eliminarse con esta receta, y como veremos más adelante, requerirá el uso de una estructura llamada una "pila" (*stack*).

El siguiente ejemplo ilustra un caso en que esto ocurre.

Ejemplo: Torres de Hanoi



Este puzzle consiste en trasladar n discos desde la estaca 1 a la estaca 3, respetando siempre las dos reglas siguientes:

- Solo se puede mover de a un disco a la vez, y
- Nunca puede haber un disco más grande sobre uno más chico Esto se puede resolver recursivamente de la siguiente manera:

Para mover n discos desde a hasta b (usando la estaba c como auxiliar):

- Primero movemos (recursivamente) $n-1$ discos desde la estaca a a la estaca b
- Una vez despejado el camino, movemos 1 disco desde a hasta c
- Finalmente, movemos de nuevo (recursivamente) los $n-1$ discos, ahora desde b hasta c (usando a como auxiliar)

El caso base es $n=0$, en cuyo caso no se hace nada.

In [68]:

```
def Hanoi(n, a, b, c): # Mover n discos desde "a" a "c", usando "b" como auxilia
r
    if n>0:
        Hanoi(n-1, a, c, b)
        print(a, "-->", c) # Mueve 1 disco desde "a" hasta "c"
        Hanoi(n-1, b, a, c)
```

In [69]:

```
Hanoi(3, 1, 2, 3)
```

```
1 --> 3
1 --> 2
3 --> 2
1 --> 3
2 --> 1
2 --> 3
1 --> 3
```

Este algoritmo es muy claro y bastante intuitivo. Si aplicamos la regla de eliminación de "recursividad a la cola", lo que resulta es un algoritmo equivalente, pero mucho menos trivial de entender:

In [70]:

```
def Hanoi(n, a, b, c): # Mover n discos desde "a" a "c", usando "b" como auxilia
r
    while n>0: # reemplaza a "if n>0:"
        Hanoi(n-1, a, c, b)
        print(a, "-->", c) # Mueve 1 disco desde "a" hasta "c"
        n-=1
        (a,b)=(b,a) # reemplaza a "Hanoi(n-1, b, a, c)"
```

In [71]:

```
Hanoi(3, 1, 2, 3)
```

```
1 --> 3
1 --> 2
3 --> 2
1 --> 3
2 --> 1
2 --> 3
1 --> 3
```

Ejercicio: Dar una explicación intuitiva de por qué funciona este algoritmo.

Diagramas de Estados

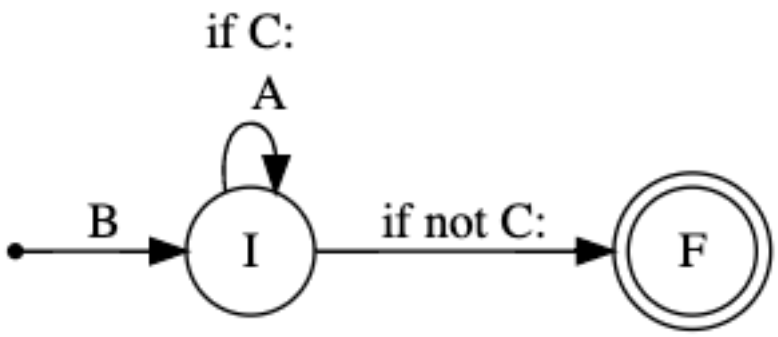
Consideremos una instrucción iterativa con un invariante "I", que se inicializa con instrucciones "B", con condición de continuación "C", con un cuerpo del ciclo consistente de instrucciones "A" y con el objetivo de lograr que se cumpla una afirmación lógica "F". Esto tendría la estructura siguiente:

```
B
while C: # invariante I
    A

# al terminar se debería cumplir F
```

Anotando con un poco más de precisión, podemos identificar lo que se cumple en cada punto:

```
B
# aquí se cumple el invariante "I" por primera vez
while C:
    # aquí se cumple "I and C"
    A
    # acá se debe cumplir nuevamente "I"

# al terminar el ciclo se cumple "I and not C"
# y eso debe implicar que se cumple F
```

Esto se puede visualizar de manera más sencilla en un **diagrama de estados** como el siguiente:



En este diagrama:

- Los estados (círculos) representan el estado del proceso en ese momento, descrito por la afirmación lógica correspondiente. En un ciclo, esa afirmación lógica es lo que hemos llamado invariante.
- Un doble círculo representa a un estado final.
- Las flechas pueden llevar como rótulo un "if" con una condición, que debe cumplirse para que se siga esa flecha, y también pueden estar rotuladas con una instrucción (o un bloque de instrucciones), que se ejecuta al hacer esa transición.

Una ventaja de modelar un algoritmo en base a un diagrama de estados es que no estamos restringidos solo a los diagramas simples que corresponden a ciclos `while`, sino que podemos construir diagramas mucho más complejos, con múltiples estados y transiciones.

Piensen por ejemplo en cómo modelar el funcionamiento de un **cajero automático**:

- El cajero está originalmente en un estado inicial, que es además el estado al cual regrasa cada vez que concluye la atención de un cliente.
- Cuando llega un cliente e inserta su tarjeta, el cajero debe verificar que la tarjeta es legible y si no es, debe devolverla con un mensaje de error y volver al estado inicial.
- Si la tarjeta es legible, debe pedir que ingrese el PIN y pasar a otro estado en que espera que el cliente ingrese dicha clave.
- Si la clave es incorrecta, debe pedir que la ingrese de nuevo y seguir en ese estado. Pero eso tiene un límite, porque si el usuario comete muchos errores, el cajero debe dar un mensaje de error, retener la tarjeta y volver al estado inicial.
- Si la clave es correcta, debe mostrar un menú de posibles operaciones y esperar que el cliente seleccione una.
- Si el cliente selecciona "Retirar dinero", debe pasar a otro estado en que le pregunta el monto que quiere sacar.
- Etc., etc., etc.,

En realidad, el diagrama de estados de un cajero automático es enorme, porque en cada estado hay múltiples opciones que conducen a realizar acciones y trasladarse a otros estados. Además, hay "timeouts" que interrumpen el proceso si una operación se demora demasiado. El diagrama de estados es la herramienta que permite modelar este tipo de procesos complejos.

Ejemplo: Contar palabras

Supongamos que tenemos un string que contiene una frase y queremos contar cuántas palabras contiene. Para simplificar, supondremos que una palabra es cualquier secuencia de caracteres distintos de un espacio en blanco. Por ejemplo, el string

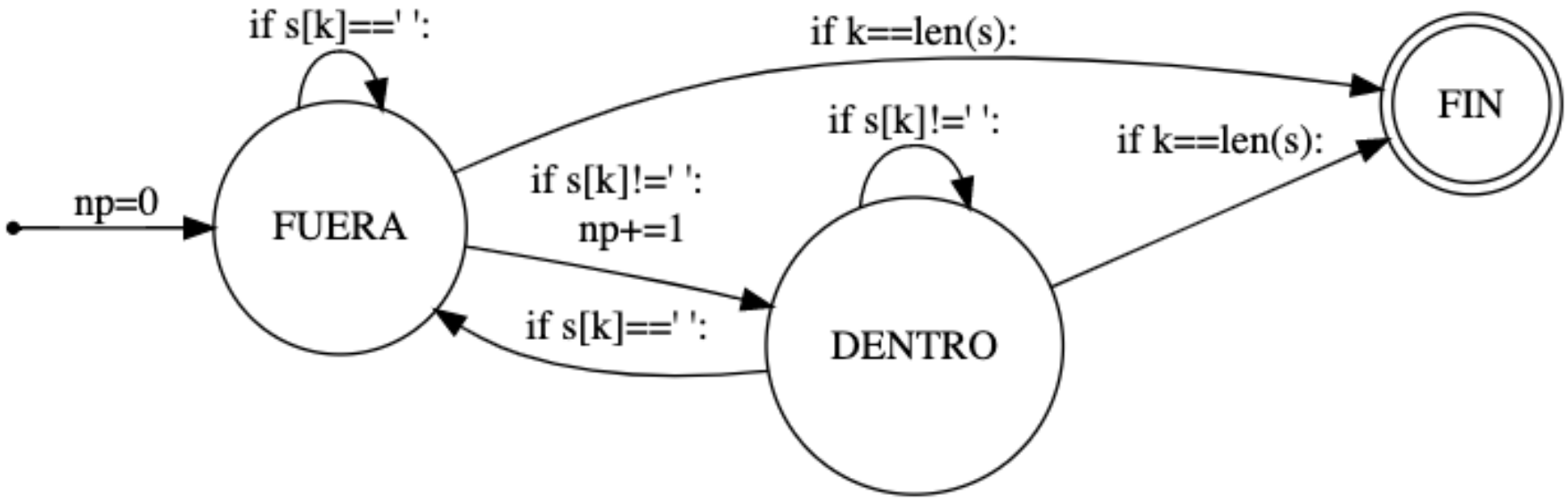
```
"    Algoritmos y    Estructuras  de   Datos   "
```

contiene 5 palabras.

Para resolver este problema, iremos examinando uno a uno los caracteres del string, y para cada uno deberemos decidir si ese caracter es el comienzo de una nueva palabra o no. Esto depende de si estábamos FUERA de una palabra (en cuyo caso sí es el inicio y hay que incrementar el contador de palabars) o si estábamos DENTRO de una palabra (en cuyo caso no se incrementa).

Esto sugiere tener un diagrama de estados con dos estados (FUERA y DENTRO), más un tercer estado final FIN, al que se llega cuando se agota el string.

El siguiente diagrama modela este proceso, donde el caracter que se está examinando en cada momento es $s[k]$. Para simplificar, suponemos que las transiciones hacia FIN tienen prioridad. Además, como en cada transición se examina un nuevo caracter, dejaremos implícita la inicialización $k = 0$ y el incremento $k+ = 1$ que hay después de cada transición.



Una vez modelado el proceso mediante un diagrama de estados, debemos escribirlo en forma de un programa. Veremos a continuación que **hay más de una manera de hacerlo**:

In [72]:

```
# Contar palabras, versión 1 con variables de estado
def contar_palabras(s):
    np=0
    estado="FUERA"
    for k in range(0,len(s)):
        if estado=="FUERA":
            if s[k]!=' ':
                np+=1
                estado="DENTRO"
        else: # estado=="DENTRO"
            if s[k]==' ':
                estado="FUERA"
    return np
```

In [73]:

```
s=input("Escriba frase: ")
print("Hay", contar_palabras(s), "palabras")
```

```
Escriba frase: Buenos días
Hay 2 palabras
```

In [74]:

```
# Contar palabras, versión 2 con variables de estado
def contar_palabras(s):
    np=0
    estado="FUERA"
    for k in range(0,len(s)):
        if s[k]==' ':
            estado="FUERA"
        else: # s[k]!=' '
            if estado=="FUERA":
                np+=1
            estado="DENTRO"
    return np
```

In [75]:

```
s=input("Escriba frase: ")
print("Hay", contar_palabras(s), "palabras")
```

```
Escriba frase:    Hola, ¿cómo estás?
Hay 3 palabras
```

In [76]:

```
# Contar palabras, versión sin variables de estado
def contar_palabras(s):
    np=0
    k=0
    while k<len(s):
        # Estamos en el estado FUERA
        if s[k]!=' ':
            np+=1
            k+=1
            # Ahora estamos en el estado DENTRO
            while k<len(s) and s[k]!=' ':
                k+=1
            if k==len(s):
                break
            # Por descarte, s[k]==' '
        k+=1
    return np
```

In [77]:

```
s=input("Escriba frase: ")
print("Hay", contar_palabras(s), "palabras")
```

```
Escriba frase: Hasta la vista,    baby
Hay 4 palabras
```