Procesamiento y Análisis de Datos Astronómicos

4.- Algoritmos de Ordenación

R. Gil-Hutton

Marzo 2020

Actividades:

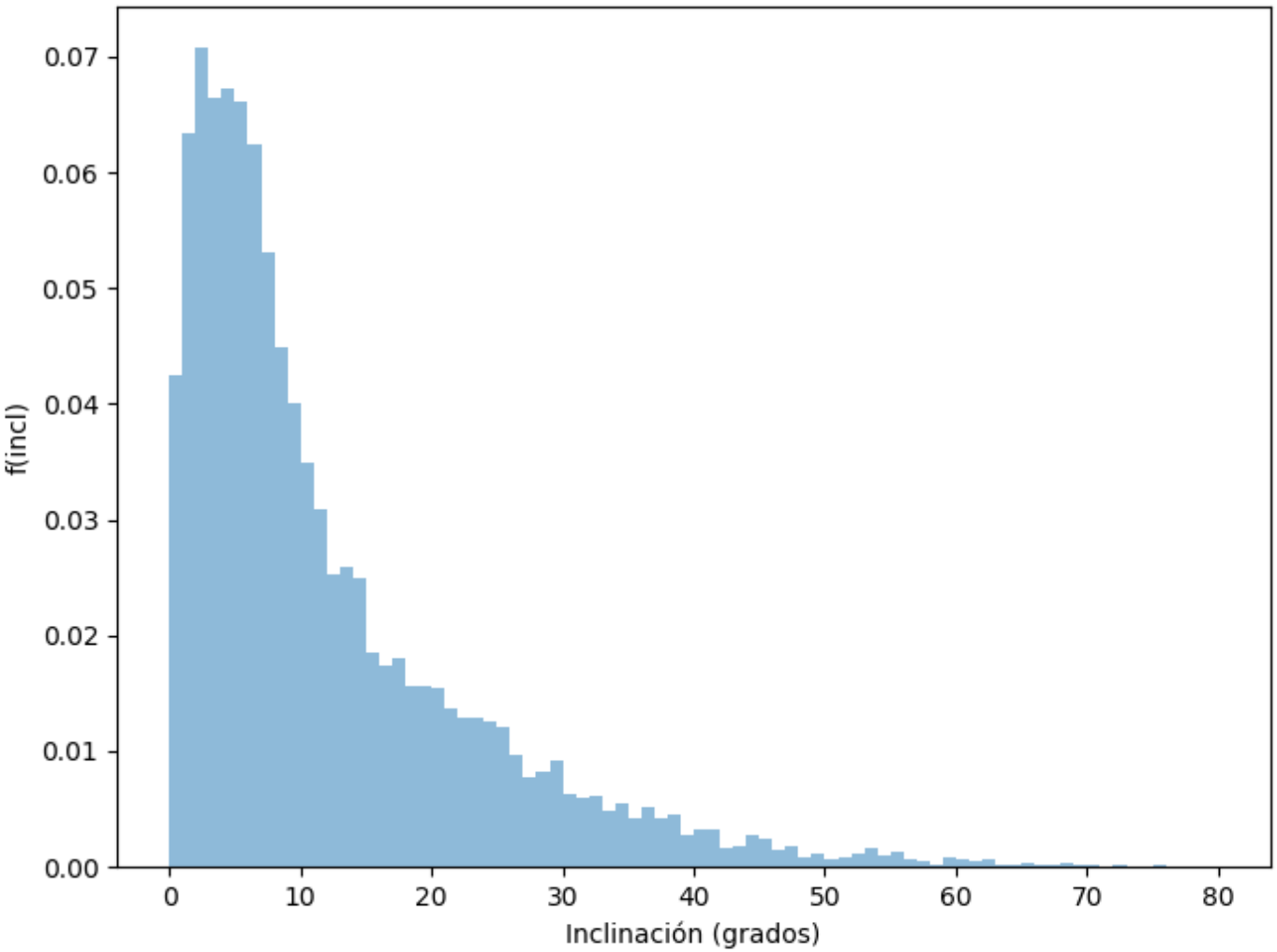
Práctica 3:

- Extraer del archivo de datos los valores para una de las variables y graficar su histograma, simular la distribución obtenida con una muestra de 300000 valores, y comparar los histogramas de frecuencias.
- Generar 10^8 pares de valores (X, Y) utilizando un generador de números aleatorios uniforme. Verificar que si el número de casos que cumplen con $X^2 + Y^2 \leq 1$ es M, se obtiene que $4 \times M/10^8 \approx \pi$.

Actividades:

```
In [54]: apo=np.loadtxt('apollo-aeih.dat')

In [55]: his=np.histogram(apo[:,2],bins=80,range=(0,80),density=True)

In [56]: plt.figure()
Out[56]: <Figure size 800x600 with 0 Axes>

In [57]: plt.bar(his[1][1:]-0.5,his[0],1.,alpha=0.5)
Out[57]: <BarContainer object of 80 artists>

In [58]:
```

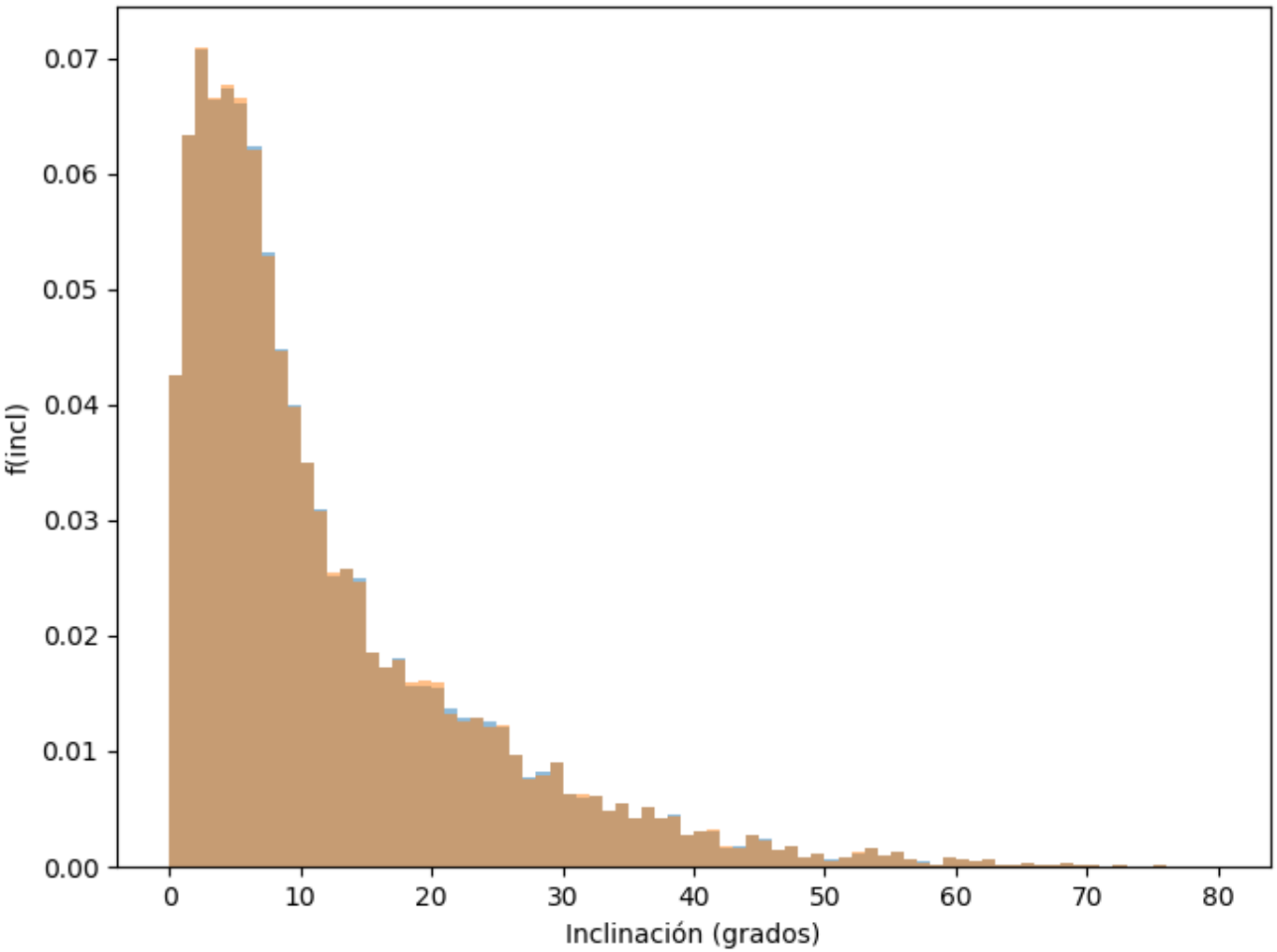
Actividades:



Actividades:

```
In [54]: apo=np.loadtxt('apollo-aeih.dat')

In [55]: his=np.histogram(apo[:,2],bins=80,range=(0,80),density=True)

In [56]: plt.figure()
Out[56]: <Figure size 800x600 with 0 Axes>

In [57]: plt.bar(his[1][1:]-0.5,his[0],1.,alpha=0.5)
Out[57]: <BarContainer object of 80 artists>

In [58]: acu=np.copy(his[0])

In [59]: acu=np.cumsum(acu,dtype=float)

In [60]: rr=np.random.random(300000)

In [61]: val=np.zeros(len(rr))

In [62]: for ii in range(len(rr)):
    ...:     jj=0
    ...:     while(rr[ii] > acu[jj]):
    ...:         jj+=1
    ...:     val[ii]=his[1][jj]+0.5+(np.random.random()-0.5)
    ...:

In [63]: sim=np.histogram(val,bins=80,range=(0,80),density=True)

In [64]: plt.bar(sim[1][1:]-0.5,sim[0],1.,alpha=0.5)
Out[64]: <BarContainer object of 80 artists>

In [65]:
```

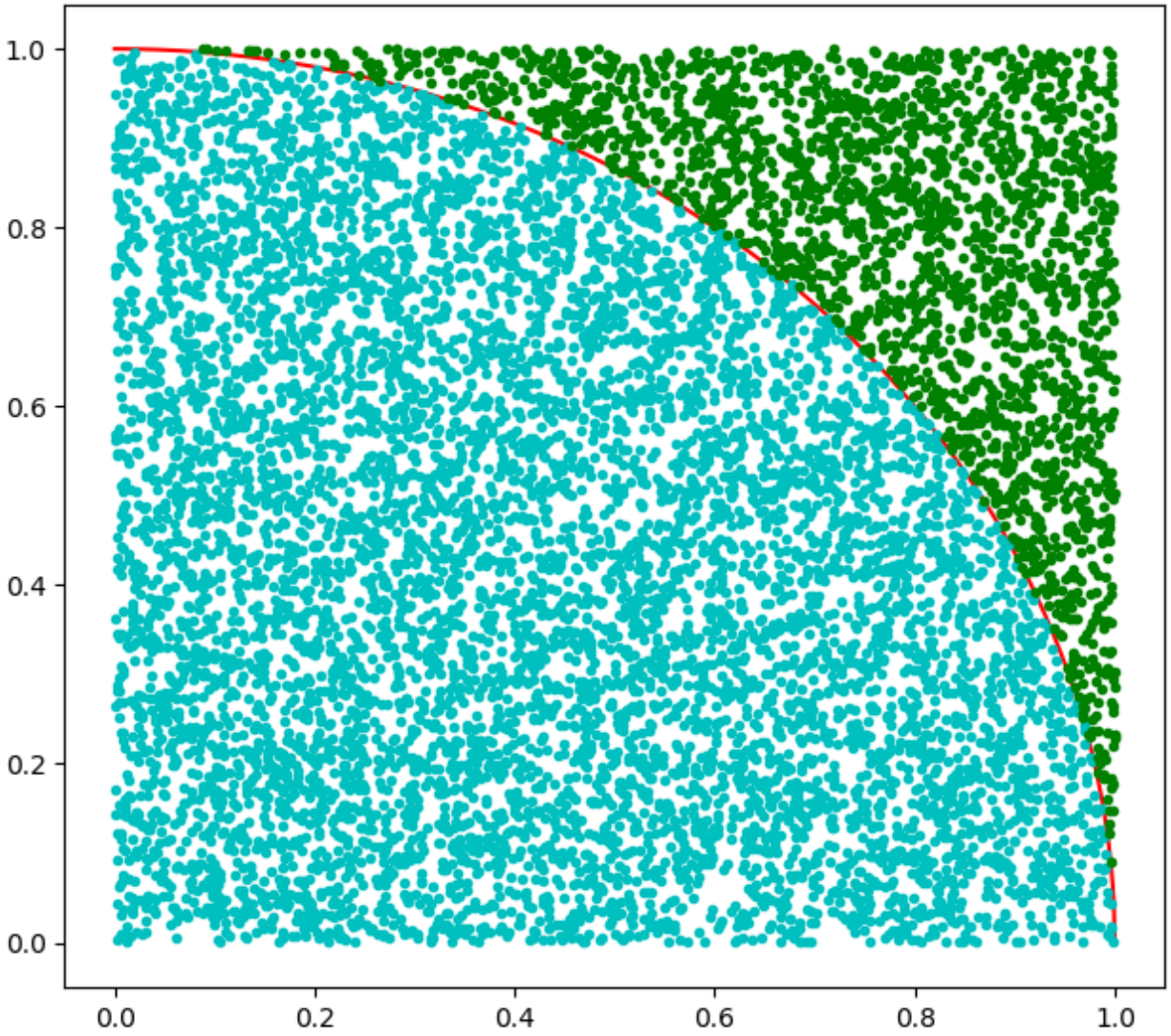
Actividades:



Actividades (Integración de Montecarlo):

```
In [32]: vx=np.random.random(100000000)

In [33]: vy=np.random.random(100000000)

In [34]: rr=vx**2+vy**2

In [35]: ind=np.where(rr <= 1.)

In [36]: 4.*len(rr[ind])/len(rr)
Out[36]: 3.14166056

In [37]:
```

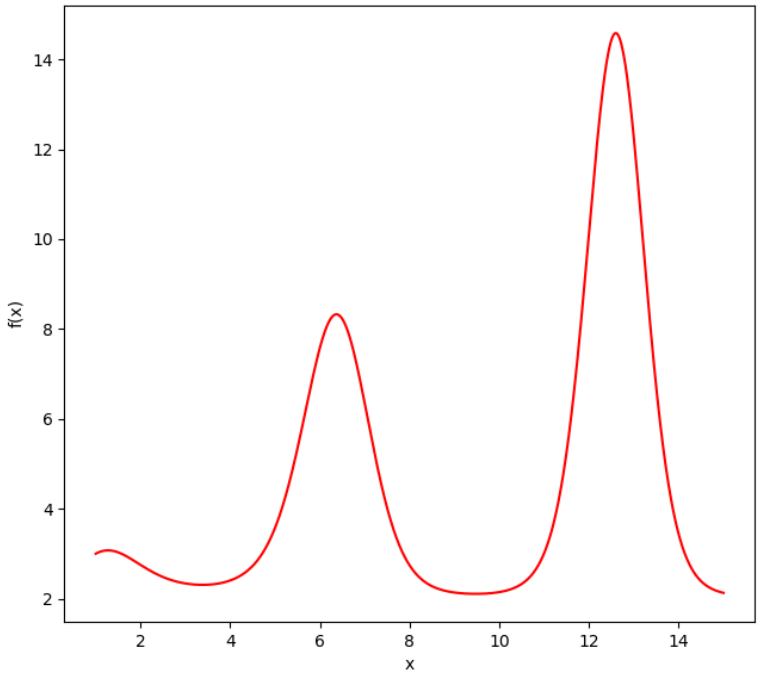
Integración de Montecarlo:



Integración de Montecarlo:

Supongamos que queremos calcular la integral:

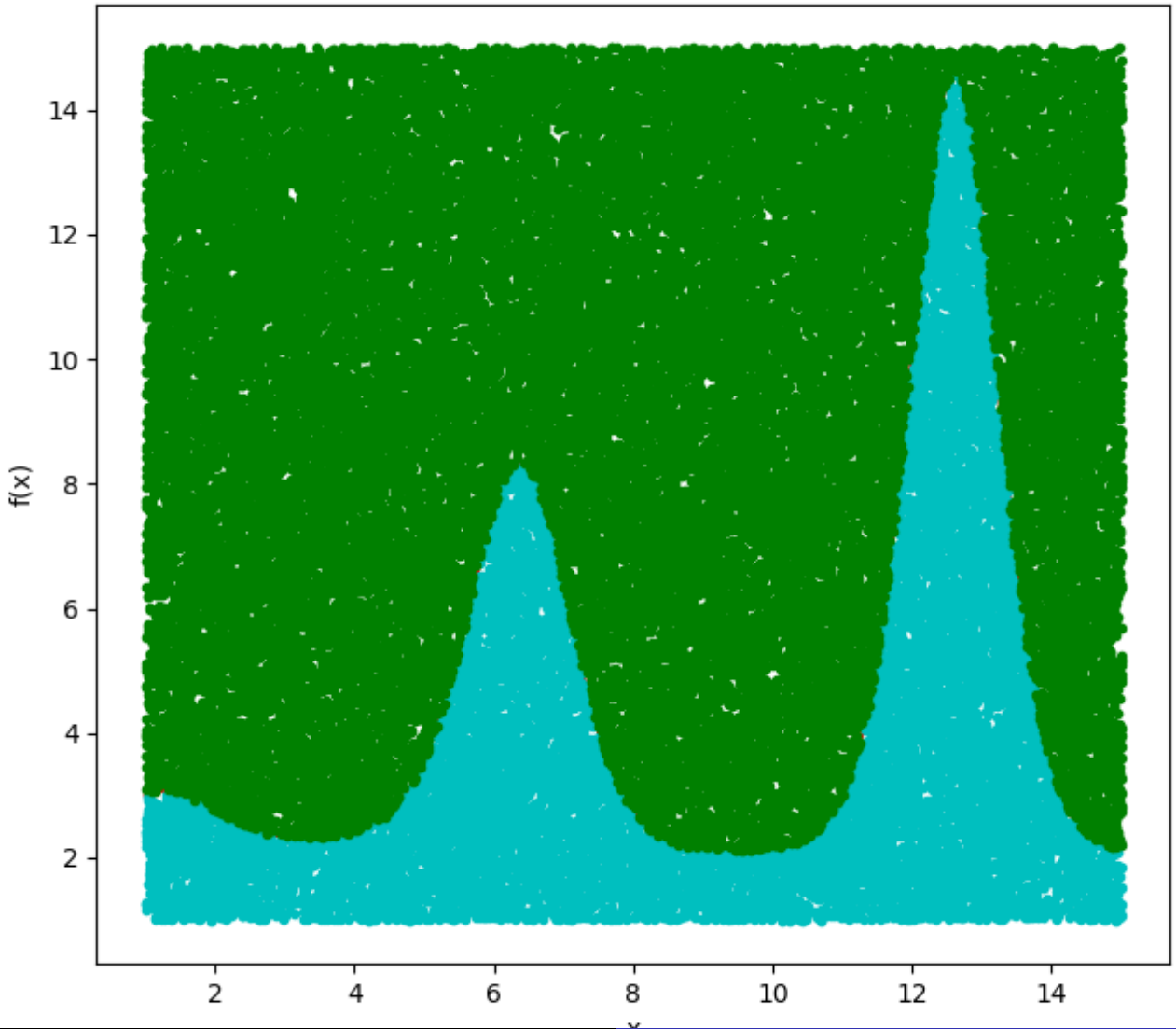
$$\int_1^{15} \left(x^{\cos(x)} + 2\right) dx$$



Integración de Montecarlo:

```
In [34]: def f(x):
    ...:     return x**(np.cos(x))+2
    ...:

In [35]: vx=np.random.random(50000)*14.+1

In [36]: vy=np.random.random(50000)*14.+1

In [37]: ind_si=np.where(vy <= f(vx))

In [38]: ind_no=np.where(vy > f(vx))

In [39]: 14.*14/50000.*len(vx[ind_si])
Out[39]: 49.67816

In [40]: plt.plot(vx[ind_si],vy[ind_si],'.c')
Out[40]: [<matplotlib.lines.Line2D at 0x7f94e5da5e10>]

In [41]: plt.plot(vx[ind_no],vy[ind_no],'.g')
Out[41]: [<matplotlib.lines.Line2D at 0x7f94e5db0f28>]
```

Integración de Montecarlo:



Algoritmos de ordenación:

Los algoritmos de ordenación se utilizan para resolver un gran número de problemas de cálculo. Por ejemplo:

- Búsqueda: buscar un elemento en una lista o array es mucho más rápido si está ordenado.
- Selección: seleccionar elementos en base a su relación con los restantes es mucho más fácil si los datos están ordenados.
- Duplicación: encontrar elementos duplicados puede ser muy rápido si están en orden.
- Distribución: analizar la distribución de frecuencia de un elemento es muy fácil si los datos están ordenados.

Algoritmos de ordenación:

- Es un algoritmo para poner los elementos de una lista, array, etc. en un cierto orden.
- Dependiendo del número de elementos, el algoritmo debe ser eficiente.
- La cantidad de memoria utilizada puede ser un condicionante.
- Es importante el grado de complejidad del algoritmo. Se pretende lograr $O(n \log n)$ en lugar de $O(n^2)$.
- Los algoritmos más rápidos requieren recursión.

Hay docenas de algoritmos de ordenación diferentes aplicables a diferentes problemas.

Algoritmos de ordenación:

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	θ(n log(n))	O(n log(n))	O(n)

Ordenación en Python:

Python tiene métodos y funciones de ordenación disponibles, al igual que Numpy y Pandas.

- Las listas disponen del método .sort() que ordena sus elementos en la misma lista. El método utiliza Timsort y sirve tanto para números como para strings.

```
In [12]: a=[3,6,1,9,4,8,7,2,5]

In [13]: b=a[:]

In [14]: a.sort()

In [15]: a
Out[15]: [1, 2, 3, 4, 5, 6, 7, 8, 9]

In [16]: b.sort(reverse=True)

In [17]: b
Out[17]: [9, 8, 7, 6, 5, 4, 3, 2, 1]

In [18]: c='Esta es una oración de prueba'.split()

In [19]: c.sort()

In [20]: c
Out[20]: ['Esta', 'de', 'es', 'oración', 'prueba', 'una']

In [21]:
```

Ordenación en Python:

- También está disponible la función intrínseca sorted() que ordena cualquier iterable (listas, tuples, diccionarios, etc.) pero lo copia siempre en una lista a la salida. La función utiliza Timsort y sirve tanto para números como para strings.

```
In [37]: a=[3,6,1,9,4,8,7,2,5]

In [38]: b=tuple(a)

In [39]: sorted(a)
Out[39]: [1, 2, 3, 4, 5, 6, 7, 8, 9]

In [40]: sorted(b,reverse=True)
Out[40]: [9, 8, 7, 6, 5, 4, 3, 2, 1]

In [41]: c='Esta es una oración de prueba'.split()

In [42]: sorted(c,reverse=True)
Out[42]: ['una', 'prueba', 'oración', 'es', 'de', 'Esta']

In [43]:
```

Ordenación en Python:

- Para ordenar arrays hay que recurrir a métodos y funciones de Numpy (o de Pandas): el método .sort() ordena los elementos en el mismo arrays mientras que la función np.sort() los ordena y copia el arrays en otro con las mismas dimensiones.
- .sort() y np.sort() permiten definir en un array multidimensional sobre qué eje se ordenará y cuál será el algoritmo de ordenación que se utilizará ('quicksort' es el default pero se puede elegir también 'mergesort', 'heapsort', o 'stable').
- El problema es que al elegir un eje de ordenación tanto .sort() como np.sort() ordenarán todos los elementos de ese eje sin respetar columnas o filas.

Ordenación en Python:

```
In [61]: a=np.array([[8,3,0],[5,2,1],[7,6,4]])

In [62]: a
Out[62]:
array([[8, 3, 0],
       [5, 2, 1],
       [7, 6, 4]])

In [63]: b=np.copy(a)

In [64]: a.sort(axis=0,kind='mergesort')

In [65]: a
Out[65]:
array([[5, 2, 0],
       [7, 3, 1],
       [8, 6, 4]])

In [66]: np.sort(b,axis=1,kind='stable')
Out[66]:
array([[0, 3, 8],
       [1, 2, 5],
       [4, 6, 7]])

In [67]:
```

Ordenación en Python:

Si se quiere ordenar un array por filas o columnas se debe utilizar la función np.argsort() que devuelve los índices que ordenan el array (o utilizar pandas.sort_values()). Entonces, si se desea ordenar por filas considerando el orden de los elementos de la primera columna el proceso usando np.argsort() será:

```
In [79]: a=np.array([[8,3,0],[5,2,1],[7,6,4]])

In [80]: a
Out[80]:
array([[8, 3, 0],
       [5, 2, 1],
       [7, 6, 4]])

In [81]: ind=np.argsort(a[:,0],kind='quicksort')

In [82]: ind
Out[82]: array([1, 2, 0])

In [83]: a[ind,:]
Out[83]:
array([[5, 2, 1],
       [7, 6, 4],
       [8, 3, 0]])

In [84]:
```

Ordenación en Python:

Numpy también tiene algunas funciones que permiten buscar ciertos elementos dentro de un array:

- np.argmax() y np.argmin() devuelven el índice del mayor y menor elemento en el eje solicitado.
- np.nonzero() devuelve los índices de los elementos que no son cero en un tuple por cada dimensión.
- np.where() devuelve los índices de los elementos que cumplen una cierta condición en un tuple por cada dimensión.
- np.extract() devuelve los elementos que cumplen una cierta condición.

Ordenación en Python:

```
In [106]: a
Out[106]:
array([[8, 3, 0],
       [5, 2, 1],
       [7, 6, 4]])

In [107]: np.argmax(a,axis=0)
Out[107]: array([0, 2, 2])

In [108]: np.argmin(a,axis=0)
Out[108]: array([1, 1, 0])

In [109]: np.nonzero(a)
Out[109]: (array([0, 0, 1, 1, 1, 2, 2, 2]), array([0, 1, 0, 1, 2, 0, 1, 2]))

In [110]: np.where(a<6)
Out[110]: (array([0, 0, 1, 1, 1, 2]), array([1, 2, 0, 1, 2, 2]))

In [111]: con=np.mod(a,2)==0

In [112]: np.extract(con,a)
Out[112]: array([8, 0, 2, 6, 4])

In [113]:
```

Algoritmos para pocos elementos:

- El uso de los algoritmos de ordenación en arrays de un cierto número de elementos depende de su eficiencia (tiempo) y de la cantidad de memoria utilizada.
- Los algoritmos más eficientes para grandes volúmenes de datos (Heapsort, Quicksort, Timsort) utilizan recursión y parcialmente emplean métodos más simples, por lo que su utilidad en conjuntos de pocos elementos es baja ya que pierden mucha eficiencia.
- Para arrays de 100-200 elementos o menos es mejor usar algoritmos más simples como selección o inserción.
- Si el arrays es de algunas decenas de elementos o menos incluso es posible utilizar un algoritmo poco eficiente como el de burbuja.

Algoritmos simples:

```
def burbuja(vector):
  """
  Ordena un vector por el metodo de burbuja
  """
  cambio=True
  while cambio:
    cambio=False
    for ii in range(len(vector)-1):
      if(vector[ii] > vector[ii+1]):
        vector[ii],vector[ii+1]=vector[ii+1],vector[ii]
        cambio=True
  return
```

Algoritmos simples:

```
def seleccion(vector):
  """
  Ordena un vector por el metodo de seleccion
  """
  for ii in range(len(vector)):
    menor=ii
    for jj in range(ii+1,len(vector)):
      if(vector[jj] < vector[menor]):
        menor=jj

    vector[ii],vector[menor]=vector[menor],vector[ii]

  return
```

Algoritmos simples:

```
def insercion(vector):
  """
  Ordena un vector por el metodo de insercion
  """
  for ii in range(1,len(vector)):
    ele=vector[ii]
    jj=ii-1

    while((jj >= 0) and (vector[jj] > ele)):
      vector[jj+1]=vector[jj]
      jj=jj-1

    vector[jj+1]=ele

  return
```

Algoritmos simples:

```
In [129]: %run ordena-simple.py

In [130]: a=[5,3,9,1,6,8,7,2,4]

In [131]: b=np.copy(a)

In [132]: c=np.copy(a)

In [133]: burbuja(a)

In [134]: a
Out[134]: [1, 2, 3, 4, 5, 6, 7, 8, 9]

In [135]: seleccion(b)

In [136]: b
Out[136]: array([1, 2, 3, 4, 5, 6, 7, 8, 9])

In [137]: insercion(c)

In [138]: c
Out[138]: array([1, 2, 3, 4, 5, 6, 7, 8, 9])

In [139]:
```

Actividades:

Práctica 4:

- Ordenar el archivo de datos con el que están trabajando fila por fila de menor a mayor según los elementos de una columna cualquiera a su elección y utilizando funciones de Numpy.
- Modificar la función de ordenamiento por selección que se mostró en esta clase para que pueda operar sobre un array con varias columnas ordenando fila por fila de menor a mayor según los elementos de una columna cualquiera.

Actividades:

Práctica 4 (cont.):

- Proponer un procedimiento para verificar que los ordenamientos realizados son correctos y aplicarlo a los dos casos anteriores.
- El archivo **landolt.dat** (va adjunto) contiene posiciones y valores fotométricos para las estrellas standard de Landolt. Ordenar este archivo según el valor de la magnitud R y guardarlo en un nuevo archivo.

Entrega

Para la próxima clase

Por consultas: ricardo.gil-hutton@conicet.gov.ar
Grupo de Ciencias Planetarias - CUIM 2