

# Universidad Nacional Autónoma de México

## Estructuras de Datos y Análisis de Algoritmos 2.

### Examen de Unidad 1.

Prof. Gerardo Tovar Tapia.

Alumna: Monroy Velázquez Alejandra Sarahí

Los archivos de los programas están contenidos en el ZIP:  
[grupo6\\_EDA\\_ProgramasExamen1](#)

#### Sección 1: Investigar y argumentar las siguientes cuestiones.

1) *Explicar que es una Función Recursiva.*

Las funciones recursivas son aquellas que se invocan a sí mismas en algún momento de su ejecución. La recursión permite programar algoritmos aparentemente complicados con un código simple y claro, ahorrando trabajo al programador. A simple vista parece la solución perfecta para muchos problemas, pero hay que tener en cuenta que en ocasiones ralentizará el programa en exceso. La recursión es una potente herramienta en programación, pero hay que saber diferenciar cuando es útil y cuando no.

2) *Explicar que es una función Iterativa.*

Los algoritmos iterativos son algoritmos que se caracterizan por ejecutarse mediante ciclos. Estos algoritmos son muy útiles al momento de realizar tareas repetitivas (como recorrer un arreglo de datos). Casi todos los lenguajes de programación modernos tienen palabras reservadas para la realización de iteraciones. La opción al uso de algoritmos iterativos es el uso de la recursividad en funciones. Estas implican una escritura más sencilla (corta), tanto para su implementación como para su entendimiento, pero en contraparte, utilizan mucho más recursos de sistema que una iteración debido a que necesitan, además del uso del procesador, la pila del sistema para "apilar" los diversos ámbitos de cada función.

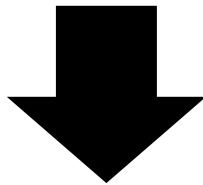
3) *Explique que es complejidad de un algoritmo.*

La complejidad algorítmica es una métrica teórica que se aplica a los algoritmos en este sentido. Entender la complejidad es importante porque a la hora de resolver muchos problemas, utilizamos algoritmos ya diseñados. Saber valorar su valor de complejidad puede ayudarnos mucho a conocer cómo se va a comportar el algoritmo e incluso a escoger uno u otro. A la idea del tiempo que consume un algoritmo para resolver un problema le llamamos complejidad temporal y a la idea de la memoria que necesita el algoritmo le llamamos complejidad espacial. La complejidad espacial, en general, tiene mucho menos

interés. El tiempo es un recurso mucho más valioso que el espacio. Así que cuando hablamos de complejidad a secas, nos estamos refiriendo prácticamente siempre a complejidad temporal. Es preciso mencionar que la complejidad no es un número, sino una función.

*4) Hacer un cuadro sinóptico dando las principales generalidades de los siguientes métodos:*

- Burbuja.
- Merge sort
- Quick Sort.
- Heap Sort.
- Radix Sort.
- Counting Sort



# ORDENACIÓN

## BURBUJA

El método de ordenación por burbuja (BubbleSort en inglés) es uno de los más básicos, simples y fáciles de comprender, pero también es poco eficiente.

La técnica utilizada se denomina ordenación por burbuja u ordenación por hundimiento y es tan fácil como comparar todos los elementos de una lista contra todos, si se cumple que uno es mayor a otro entonces se intercambia de posición. Los valores más pequeños suben o burbujan hacia la cima o parte superior de la lista, mientras que los valores mayores se hunden en la parte inferior.

## MERGE SORT

Merge Sort es un algoritmo de ordenamiento basado en el paradigma divide y vencerás o también conocido como divide y conquista

Esta es una técnica de diseño de algoritmos que consiste en resolver un problema a partir de la solución de sub-problemas del mismo tipo, pero de menor tamaño. Consta de tres pasos: Dividir el problema en dos utilizando la fórmula  $(p+r)/2$ ; resolver cada sub-problema y combinar las soluciones obtenidas para la solución al problema original

## QUICK SORT

Este algoritmo igual que el Merge Sort sigue el paradigma de Divide y Vencerás.

Divide: Se divide un algoritmo A en dos subarreglos utilizando un elemento pivote x de manera que de un lado queden todos los elementos menores o iguales a el y del otro los mayores.  
Conquista: Ordenar los subarreglos de la izquierda y derecha con llamadas recursivas a la función QuickSort().

## HEAP SORT

También conocido con el nombre de "montículo". Un montículo es una estructura de datos que se puede manejar como un arreglo de objetos o también puede ser visto como un árbol binario cuyos nodos contienen información de un conjunto ordenado.

Cada nodo del árbol corresponde a un elemento del arreglo.  
El algoritmo consiste en construir un montículo y después ir extrayendo el nodo que queda como raíz del árbol en sucesivas iteraciones hasta obtener el conjunto ordenado.

## COUNTING SORT

Es un algoritmo que no se basa en comparaciones y lo que hace es contar el número de elementos de cada clase en un rango de 0-k; para después ordenarlos determinando para cada elemento de entrada el número de elementos menores a él.

Se asumen 3 arreglos lineales: el arreglo A a ordenar con n elementos; el arreglo B de n elementos para guardar la salida ya ordenada; y el arreglo C para almacenar temporalmente k elementos.

## RADIX SORT

También llamado ordenamiento por residuos puede utilizarse cuando los valores a ordenar están compuestos por secuencias que admiten un orden lexicográfico.

Utiliza un algoritmo de ordenación estable, las letras o dígitos, partiendo desde el que está más a la derecha(menos significativo) hasta el que se encuentra mas a la izquierda(más significativo)

## Sección 2: Programar los siguientes ejercicios, documentar y entregar evidencia de cada programa.

5) Escribir una función que reciba 2 enteros  $n$  y  $b$  y devuelva *True* si  $n$  es potencia de  $b$ . Ejemplos:

```
>>> es_potencia(8,2)
```

True

```
>>> es_potencia(64,4)
```

True

```
>>> es_potencia(70,10)
```

False

-CÓDIGO-

```
1 def es_potencia(n,b):
2     comp=1
3     while (comp<b):
4         comp*=n
5     if comp==b:
6         resultado=True
7     else:
8         resultado=False
9     return resultado
10
11 print("-->Examen1 - Ejercicio5<--\n")
12 n=int(input("Introduce un número: "))
13 b=int(input("Introduce el número que se desea saber si es potencia de {0}:".format(n)))
14 resultado=es_potencia(n,b)
15 if(resultado):
16     print("\n>> {0} << \nEl {1} es potencia de {2}.".format(resultado,b,n))
17 else:
18     print("\n>> {0} << \nEl {1} no es potencia de {2}.".format(resultado,b,n))
19
```

El programa pide ingresar un número, y posteriormente pide ingresar el número del cual queremos saber si es potencia del primero, después llama a la función *es\_potencia* y le pasa los dos argumentos, dentro de esta función asigna un uno a la variable *comp*, entra en un ciclo *while* donde valida la condición de que mientras *comp* sea menor a *b* va a multiplicar a *comp* por *n*, y se le asigna a *comp*; después valida una condición de que si *comp* es igual a *b*, el resultado es *True* y si no el resultado es *False*, y finalmente retorna el resultado.

-CORRIDA-

```
[GCC 4.8.2] on linux
❖
-->Examen1 - Ejercicio5<--

Introduce un número: 2
Introduce el número que se desea saber si es potencia de 2: 8

>> True <<
El 8 es potencia de 2.
❖ █

Introduce un número: 4
Introduce el número que se desea saber si es potencia de 4: 64

>> True <<
El 64 es potencia de 4.
❖ █

Introduce un número: 10
Introduce el número que se desea saber si es potencia de 10: 70

>> False <<
El 70 no es potencia de 10.
❖ █
```

6) *Programa de astrología: el usuario debe ingresar el día y mes de su cumpleaños y el programa le debe decir a que signo corresponde. **Nota:***

Aries: 21 de marzo al 20 de abril.

Tauro: 21 de abril al 20 de mayo.

Géminis: 21 de mayo al 21 de junio.

Cáncer: 22 de junio al 23 de julio.

Leo: 24 de julio al 23 de agosto.

Virgo: 24 de agosto al 23 de septiembre.

Libra: 24 de septiembre al 22 de octubre.

Escorpio: 23 de octubre al 22 de noviembre.

Sagitario: 23 de noviembre al 21 de diciembre.

Capricornio: 22 de diciembre al 20 de enero.

Acuario: 21 de enero al 19 de febrero.

Piscis: 20 de febrero al 20 de marzo.

## -CÓDIGO-

```
1 def Horoscopo(m,d):
2     if m==1:
3         if(d<=20):
4             s="Capricornio."
5         else:
6             s="Acuario."
7
8     elif m==2:
9         if(d<=19):
10            s="Acuario."
11        else:
12            s="Piscis."
13    elif m==3:
14        if(d<=20):
15            s="Piscis."
16        else:
17            s="Aries."
18    elif m==4:
19        if(d<=20):
20            s="Aries."
21        else:
22            s="Tauro."
23    elif m==5:
24        if(d<=20):
25            s="Tauro."
26        else:
27            s="Geminis."
28    elif m==6:
29        if(d<=21):
30            s="Geminis."
31        else:
32            s="Cancer."
33    elif m==7:
34        if(d<=23):
35            s="Cancer."
36        else:
37            s="Leo."
38    elif m==8:
39        if(d<=23):
40            s="Leo."
41        else:
42            s="Virgo."
43    elif m==9:
44        if(d<=23):
45            s="Virgo."
46        else:
47            s="Libra."
48    elif m==10:
49        if(d<=22):
50            s="Libra."
51        else:
52            s="Escorpio."
53    elif m==11:
54        if(d<=22):
55            s="Escorpio."
56        else:
57            s="Sagitario."
58    elif m==12:
59        if(d<=21):
60            s="Sagitario"
61        else:
62            s="Capricornio."
63    else:
64        s="No Definido."
65        print("Ocurrió un error...")
66    return s
67
68 M=[0,31,29,31,30,31,30,31,31,30,31,30,31]
69 print("-->Examen1 - Ejercicio6<--\n")
70 m=int(input("Introduce tu mes de nacimiento: "))
71 while (m<1 or m>12):
72     m=int(input("Mes invalido, introduce uno valido entre 1 y 12: "))
73 ds=M[m]
74 d=int(input("Introduce tu dia de nacimiento: "))
75 while (d<1 or d>ds):
76     d=int(input("Dia invalido, introduce uno valido entre 1 y {0}: ".format(ds)))
77 signo=Horoscopo(m,d)
78
79 print("\nTu signo zodiacal es: ",signo)
```

El programa cuenta con una lista M, que almacena el ultimo día de cada mes, 31,29,31.... Etc. pide introducir el mes de nacimiento y con ayuda de un *while* valida que este dentro de uno y doce, después asigna a la variable *ds* la posición en la lista M del mes de nacimiento, luego pide ingresar el día y nuevamente con ayuda de un *while* valida que el día este dentro de uno y el último mes del mes de nacimiento. Posteriormente llama a la función *Horoscopo* pasándole como argumento el día y mes y dentro de esta función implementa la estructura *if-elif-else* para validar en que caso entra para devolver que signo zodiacal corresponde.

-CORRIDA-

```
Python 3.6.1 (default, Dec 12, 2015, 13:05:11)
[GCC 4.8.2] on linux
-->Examen1 - Ejercicio6<--

Introduce tu mes de nacimiento: 0
Mes invalido, introduce uno valido entre 1 y 12: 4
Introduce tu dia de nacimiento: 90
Dia invalido, introduce uno valido entre 1 y 30: 3

Tu signo zodiacal es: Aries.
>
```



- 7) Función que calcule el máximo o mínimo de un polinomio de segundo grado (dados los coeficientes  $a$ ,  $b$  y  $c$ ), indicando si es un máximo o un mínimo.

-CÓDIGO-

```
1 def maxymin(a,b,c):
2     ad = 2*a
3     x = -b/ad
4     y = (a*(x**2)) + (b*x) + c
5     if a>0:
6         print("\n----->Valor minimo<-----")
7         print("x = ",x, "\ny = ",y)
8     else:
9         print("\n----->Valor maximo<-----")
10        print("x = ",x, "\ny = ",y)
11
12    print("-->Examen1 - Ejercicio7<--\n")
13    print("Encontrando maximos o minimos de un polinomio (a(x)^2+b(x)+c)")
14    a=int(input("Ingrese el coeficiente a: "))
15    while a == 0:
16        a = int(input("Ingrese de nuevo a: "))
17    b=int(input("Ingrese el coeficiente b: "))
18    c=int(input("Ingrese el coeficiente c: "))
19    maxymin(a,b,c)
```

El programa pide ingresar los coeficientes  $a$ ,  $b$ , y  $c$  validando que  $a$  no sea un cero, llama a la función *maxymin* y dentro de esta función utiliza el criterio de la primera derivada para encontrar su máximo o mínimo, asigna a  $ad$ , la operación de dos por  $a$  que sería la derivada del coeficiente  $a$ , luego para despejar  $x$  asigna a esta variable la operación  $-b$  entre  $ad$ , y para la coordenada en  $y$  solo sustituye la variable  $x$  en la función original, si  $a$  es mayor a cero se tratara de un mínimo y si no de un máximo.

CORRIDA-

```
[GCC 4.8.2] on linux
-->Examen1 - Ejercicio7<--

Encontrando maximos o minimos de un polinomio (a(x)^2+b(x)+c)
Ingrese el coeficiente a: 0
Ingrese de nuevo a: 1
Ingrese el coeficiente b: 2
Ingrese el coeficiente c: 3

----->Valor minimo<-----
x = -1.0
y = 2.0

[GCC 4.8.2] on linux
-->Examen1 - Ejercicio7<--

Encontrando maximos o minimos de un polinomio (a(x)^2+b(x)+c)
Ingrese el coeficiente a: -1
Ingrese el coeficiente b: -2
Ingrese el coeficiente c: -3

----->Valor maximo<-----
x = -1.0
y = -2.0
```

- 8) Función que calcule la intersección de dos rectas (dadas las pendientes y ordenada al origen de cada recta). **Nota:** validar que no sean dos rectas con la misma pendiente, antes de efectuar la operación.

-CÓDIGO-

```
1 def interseccion(m1,b1,m2,b2):
2     temp1 = m1 - m2
3     temp2 = b2 - b1
4     x = temp2/temp1
5     y = (m1*x) + b1
6     print("\n\n-->Coordenadas de intersección<--")
7     print("x = {} \ny = {}".format(x,y))
8
9     print("-->Examen1 - Ejercicio 8<--")
10    print("Punto de intersección de una recta (y = mx+b)\n")
11    m1 = int(input("Ingrese la pendiente de la primer recta: "))
12    b1 = int(input("Ingrese la ordenada al origen de la primer recta: "))
13    m2 = int(input("\nIngrese la pendiente de la segunda recta: "))
14    while(m1 == m2):
15        print("-->Las rectas son paralelas, por lo tanto nunca se intersectan\n")
16        m2 = int(input("Ingrese de nuevo la segunda pendiente: "))
17        b2 = int(input("Ingrese la ordenada al origen de la segunda recta: "))
18    interseccion(m1,b1,m2,b2)
```

El programa pide el valor de las pendientes y las ordenadas al origen, validando que las pendientes no sean iguales, mediante un *while*. Después llama a la función intersección pasándole como argumentos dichos valores, almacena en una variable temporal la operación de la pendiente uno menos la dos y en una segunda variable temporal la operación de la ordenada dos menos la uno. Para la coordenada en x asigna la operación de la temporal uno entre la dos, y para la coordenada en y la multiplicación de la pendiente uno por la variable x más la ordenada al origen 1.

-CORRIDA-

```
[GCC 4.8.2] on linux
-->Examen1 - Ejercicio 8<--
Punto de intersección de una recta (y = mx+b)

Ingrese la pendiente de la primer recta: 2
Ingrese la ordenada al origen de la primer recta: 3

Ingrese la pendiente de la segunda recta: 2
-->Las rectas son paralelas, por lo tanto nunca se intersectan.
Ingrese de nuevo la segunda pendiente: 4
Ingrese la ordenada al origen de la segunda recta: 5

-->Coordenadas de intersección<--
x = -1.0
y = 1.0
```

9) Escribir una función recursiva que encuentre el mayor elemento de una lista.

-CÓDIGO-

```
1 def maxNum(L,j):
2     if j>0:
3         return max(L[j], maxNum(L, j-1))
4     else:
5         return L[0]
6
7 print("-->Examen1 - Ejercicio9<--\n")
8 lista=[1,2,-7,6,-3,9,4,5,8,0]
9 num =maxNum(lista,9)
10 print(lista)
11 print("\nEl numero mayor de la lista es: ",num)
12
```

El programa define una lista, y llama a la función *maxNum* pasándole como argumento la lista, y el tamaño de la lista más uno, dentro de esta función valida la condición de que si el ultimo índice es mayor a cero retorna el *max* pasándole como argumento el valor de *j* en la posición de la lista y una llamada a la misma función pasándole como argumento la lista y el valor de *j* menos uno. Si no es cierta la condición te retorna el valor contenido en el índice 0 de la lista.

-CORRIDA-

```
[GCC 4.8.2] on linux
-->Examen1 - Ejercicio9<--
[1, 2, -7, 6, -3, 9, 4, 5, 8, 0]
El numero mayor de la lista es: 9
```

- 10) Escribir una función recursiva para replicar los elementos de una lista una cantidad  $n$  de veces. Por ejemplo, replicar  $([1, 3, 3, 7], 2) = ([1, 1, 3, 3, 3, 3, 7, 7])$

-CÓDIGO-

```
1 def replica(li, r):
2     if not li:
3         return li
4     inicio = li[0:1]
5     fin = li[1:]
6     return (inicio * r) + replica(fin, r)
7
8 print("-->Examen1 - Ejercicio10<--\n")
9 lista = ([1, 3, 3, 7])
10 print("Lista original: ", lista)
11 newlist = replica(lista, 2)
12 print("Nueva lista: ", newlist)
```

El programa define una lista y la imprime, después llama a la función *replica* pasándole como argumento a la lista y el número de veces que se replicara, dentro de esta función valida la condición *if not* en la lista entonces retorna a la lista, después asigna a la variable *inicio*, la partición de la lista de cero a uno, y a la variable *fin* se le asigna la partición de la lista de uno hasta que termine la lista, y retorna la operación resultado de la multiplicación de la variable *inicio* por el número de veces que se va a replicar, más una llamada a la función *replica* pasándole como argumento la variable *fin* y el número de veces que se tiene que replicar.

-CORRIDA-

```
[GCC 4.8.2] on linux
-->Examen1 - Ejercicio10<--

Lista original: [1, 3, 3, 7]
Nueva lista: [1, 1, 3, 3, 3, 3, 7, 7]
```

11) Supongamos que tenemos una lista de listas, dada por:

*Lista2*=[[1, 200,50000],[1,2,3], [1],[100],[2,2,3,4,5,6,3,21],[1,2,5,2,5,7],[1,2,0,8,7,6],  
[5,6,4,8,3,2],[2,3],[1,2,99]]

Escribir una función *merge\_sort\_3* que funcione igual que el merge sort pero en lugar de dividir los valores en dos partes iguales, los divida en tres (asumir que se cuenta con la función *merge(lista\_1, lista\_2, lista\_3)*). ¿Cómo te parece que se va a comportar este método con respecto al merge sort original?

Modificando al programa de Merge Sort original, este programa divide a la lista en tres en vez de dos, este algoritmo no trabaja satisfactoriamente, ya que al dividirlo de esta forma lo hace menos eficiente, por lo tanto también hay deficiencias e inconsistencias dentro de este programa, ya que a la hora de dividir los últimos sub arreglos, el del centro queda vacío lo cual deja problemas a la hora de ordenar, y por este mismo motivo se agregó más código. También debido al casteo de la operación los sub arreglos tienen distintas longitudes.

-CÓDIGO-

```
1 def MergeSort(A,p,r):
2     if r-p>0:
3         q=int((r-p)/3)+p
4         q2=int(((r-p)/3)*2)+p
5         MergeSort(A,p,q)
6         MergeSort(A,q+1,q2)
7         MergeSort(A,q2+1,r)
8         Merge(A,p,q,q2,r)
9
10 def CrearSubArreglo(A, indIzq,indDer):
11     return A[indIzq:indDer+1]
12
13 def Merge(A,p,q,q2,r):
14     Izq=CrearSubArreglo(A,p,q)
15     Cen=CrearSubArreglo(A,q+1,q2)
16     Der=CrearSubArreglo(A,q2+1,r)
17     i=0
18     j=0
19     B=Izq+Cen+Der
20     print(B)
21     if len(Cen)==0:
22         for k in range (p,q+2):
23             if(j>=len(Der)) or (i< len(Izq) and Izq[i]< Der[j]):
24                 A[k]=Izq[i]
25                 i= i+1
26             else:
27                 A[k]= Der[j]
28                 j=j+1
29     else:
30         for l in range(0,len(Izq)):
```

```

31
32     for m in range(0, len(Cen)):
33         if (Cen[m] < Izq[1]):
34             temp = Cen[m]
35             Cen[m] = Izq[1]
36             Izq[1] = temp
37     for m in range(0, len(Der)):
38         if (Der[m] < Izq[1]):
39             temp = Der[m]
40             Der[m] = Izq[1]
41             Izq[1] = temp
42     for l in range(0, len(Cen)):
43         for m in range(0, len(Der)):
44             if (Der[m] < Cen[1]):
45                 temp = Der[m]
46                 Der[m] = Cen[1]
47                 Cen[1] = temp
48     for l in range(0, len(Der)-1):
49         for m in range(1, len(Der)):
50             if (Der[m] < Der[1]):
51                 temp = Der[m]
52                 Der[m] = Der[1]
53                 Der[1] = temp
54
55     B = Izq + Cen + Der
56     j = 0
57     for i in range(p, r+1):
58
59         A[i] = B[j]
60         j += 1
61     print("{0}      {1}      {2}\n".format(Izq, Cen, Der))
62
63     A = [10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
64     print("Lista original", A)
65     MergeSort(A, 0, len(A)-1)
66
67     print("Lista ordenada: ", A)
68

```

-CORRIDA-

```

[GCC 4.8.2] on linux
>
Lista original [10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
[10, 9]
[10]      []      [9]

[9, 10, 8, 7]
[7, 8]      [9]      [10]

[6, 5, 4]
[4]      [5]      [6]

[3, 2]
[3]      []      [2]

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

[7, 8, 9, 10, 4, 5, 6, 0, 1, 2, 3]
[0, 1, 2, 3]      [4, 5, 6]      [7, 8, 9, 10]

Lista ordenada: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>

```

### Sección 3: Experimental

12) *Mostrar los pasos del ordenamiento de la lista: 6, 0, 3, 2, 5, 7, 4, 1 con los métodos de Burbuja, Merge Sort, Quick Sort, Heap Sort, Radix sort, Counting Sort.*

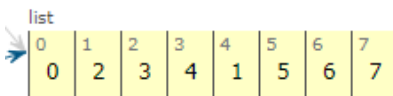
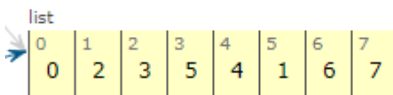
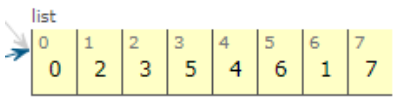
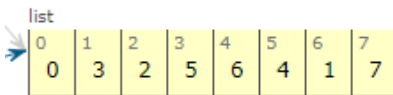
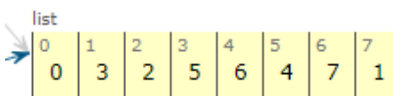
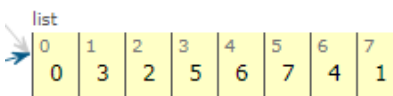
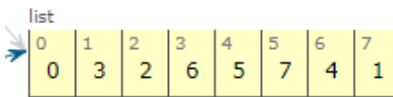
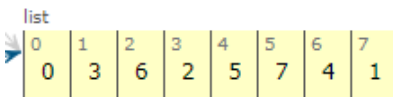
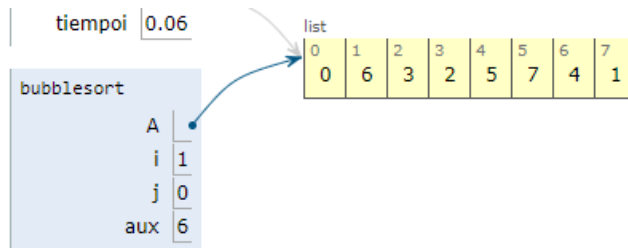
**Nota: Se deberá hacer prueba de escritorio, y cada prueba deberá ser documentada.**

*¿Cuáles son las principales diferencias entre los métodos? ¿Cuál usarías en qué casos?*

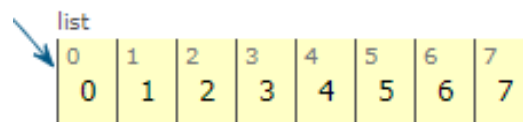
Las diferencias marcadas entre los métodos son: el método de la burbuja recorre todo el arreglo y hace comparaciones del elemento presente con el siguiente. Mientras que los métodos merge, quick y heap hacen particiones y después las unen para ordenar la lista, esto los hace más eficientes; el único algoritmo que no se basa en comparaciones es counting, ya que crea listas alternas y las llena contando cuantos elementos de cada uno hay para después ordenar la lista. El método de la burbuja se podría usar en una lista pequeña ya que es más eficiente, y los métodos restantes aplicados a listas de mayor longitud.

#### -BURBUJA-

```
1  from time import clock
2
3  def bubblesort(A):
4      for i in range(1,len(A)):
5          for j in range (len(A)-1):
6              if A[j]>A[j+1]:
7                  aux= A[j]
8                  A[j]=A[j+1]
9                  A[j+1]=aux
10
11
12  lista = [6, 0, 3, 2, 5, 7, 4, 1]
13  print("Lista desordenada: ",lista)
14  tiempoi=clock()
15  bubblesort(lista)
16  tiempof=clock()
17
18  print("\n\nLista ordenada:",lista)
19  t=tiempof-tiempoi
20  print("\n\nEl tiempo de la Burbuja es: {0} seg".format(t))
21
22
```



... Se recorre el uno 3 posiciones a la izquierda...



Se imprime la lista desordenada. Y se le pasa a la función *BubbleSort*. Entra en un ciclo for desde 1 hasta el tamaño de la lista, y dentro de este ciclo en otro ciclo for desde 0 hasta el tamaño de lista menos 1. Valida una condición si la posición del valor de j en la lista es mayor que la posición del valor j más uno de la lista, si la cumple, guarda A[ j ] en una variable auxiliar y después asigna al valor de la posición j más uno de la lista al valor de la posición j de la lista, después asigna el auxiliar al valor de la posición de j más uno.

Y de esta forma va recorriendo la lista hasta que se terminen los ciclos y devuelve la lista ya ordenada.

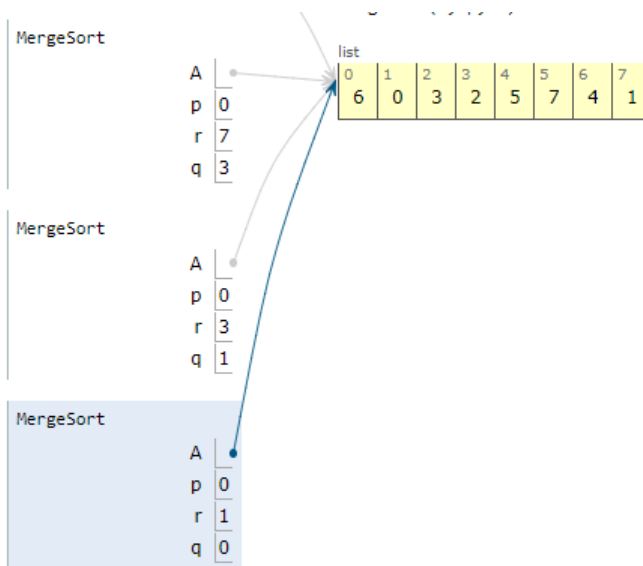
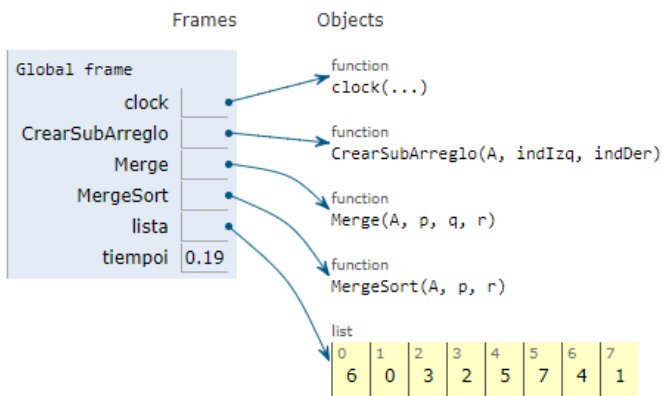


## -MERGE SORT-

```
1  from time import clock
2
3  def CrearSubArreglo(A, indIzq,indDer):
4      return A[indIzq:indDer+1]
5  def Merge(A,p,q,r):
6      Izq=CrearSubArreglo(A,p,q)
7      Der=CrearSubArreglo(A,q+1,r)
8      i=0
9      j=0
10     for k in range (p,r+1):
11         if(j>=len(Der)) or (i< len(Izq) and Izq[i]< Der[j]):
12             A[k]=Izq[i]
13             i= i+1
14         else:
15             A[k]= Der[j]
16             j=j+1
17     def MergeSort(A,p,r):
18         if r-p>0:
19             q=int((p+r)/2)
20             MergeSort(A,p,q)
21             MergeSort(A,q+1,r)
22             Merge(A,p,q,r)
23
24     lista = [6, 0, 3, 2, 5, 7, 4, 1 ]
25     print("Lista desordenada: {0} ".format(lista))
26     tiempoi=clock()
27     MergeSort(lista,0,len(lista)-1)
28     tiempof=clock()
29     print("\n\nLista ordenada: ",lista)
30     t=tiempof-tiempoi
31     print("\n\nEl tiempo de MergeSort es: {0} seg ".format(t))
```

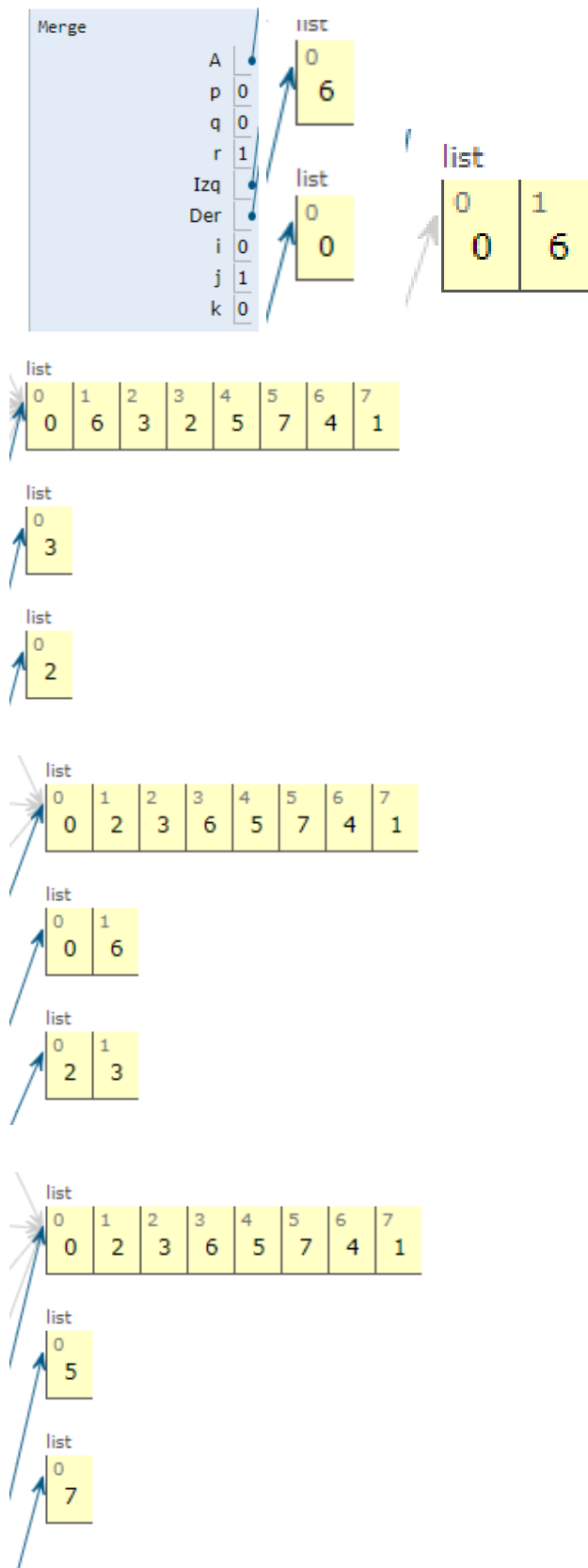
Print output (drag lower right corner to resize)

Lista desordenada: [6, 0, 3, 2, 5, 7, 4, 1]



Se imprime la lista desordenada, y se le pasa a la función *MergeSort* junto con otros dos argumentos, *p* que es el índice donde comienza la lista, siendo 0 y el índice donde termina, que es el tamaño de la lista menos uno.

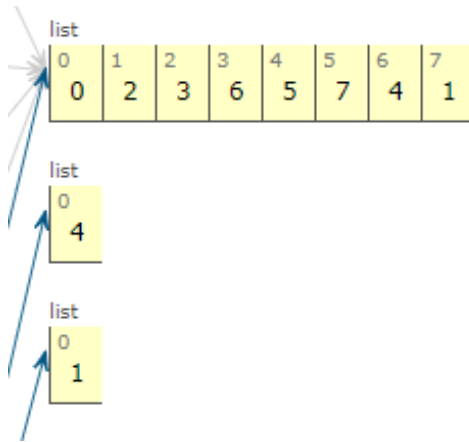
Dentro de esta función se hacen las particiones de la lista, se valida primeramente la condición si la operación de  $(r-p)$  es mayor a cero se hace lo siguiente: introducimos una nueva variable llamada *q* que es el resultado del casteo de la operación  $((p+r)/2)$ . Se va a estar llamando recursivamente a la función *MergeSort* pasándole como argumentos la lista, la variable *p* y la variable *q*, esto para ir creando las particiones del lado izquierdo, hasta que ya no se cumpla la condición ahora llamaremos recursivamente de nuevo a la misma función pero ahora pasándole como argumentos la lista, la variable  $q+1$  y *r*, para crear las particiones de lado derecho.



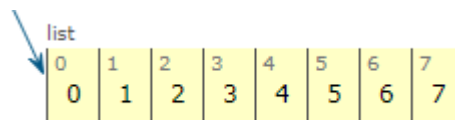
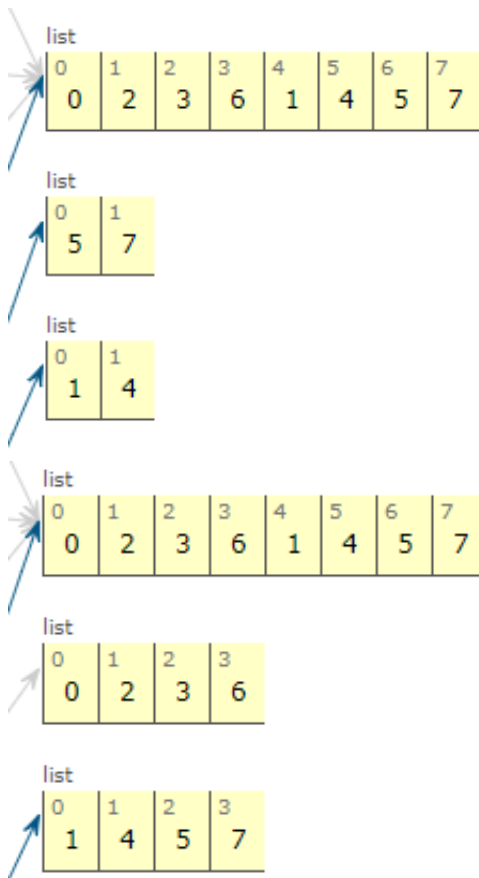
Posteriormente se llama a la función *Merge* pasándole como argumento a la lista, el valor de p, q y r. Dentro de esta función asigna a la variable Izq, una llamada a la función *CrearSubArreglo* pasándole como parámetro a la lista p como índIzq y q como indDer, esta función retorna una partición de la lista de p a q+1. Despues asigna a la variable Der de nuevo una llamada a la función *CrearSubArreglo* pasándole como parámetro la lista, el valor de q más uno y r. Y recibe una partición de la lista de q+1 a r.

Las variables i y j se inicializan en 0.

Luego de hacer dichas asignaciones, se entra a un ciclo for iterando con la variable k desde el valor de p hasta el valor de r+1. Se valida la condición de si j es mayor o igual a el tamaño de la partición derecha o si i es menor que la partición de la izquierda, y el valor en la posición i de la partición izquierda es menor que el valor de la posición de j en la partición derecha, se hace lo siguiente: Se asigna el valor de la partición izquierda en la posición i a la la posición k de la lista. La variable i aumenta en uno. Si la condición no se cumple se hace lo siguiente: se asigna el valor de la partición derecha en la posición j a la posición en k de la lista original, y j aumenta en uno.

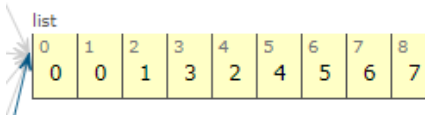
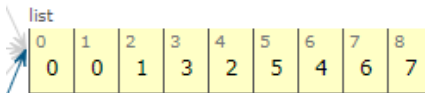
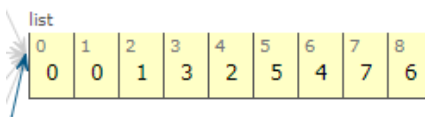
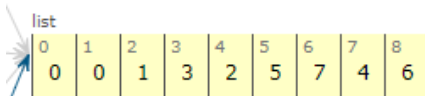
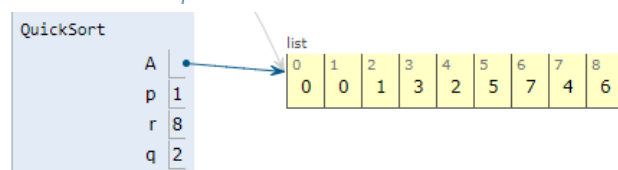
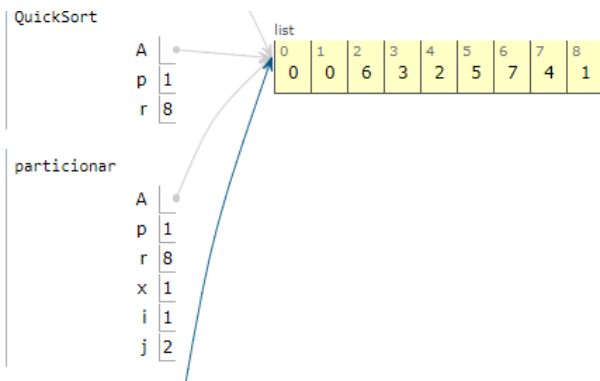
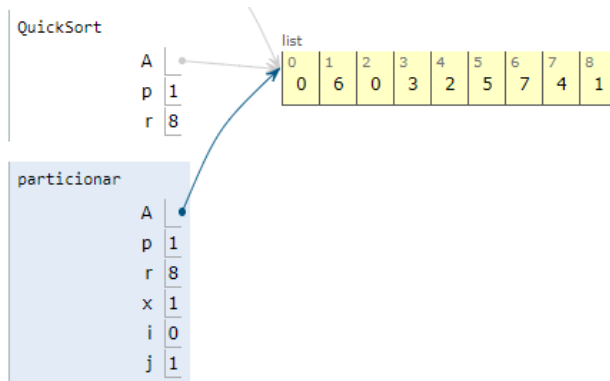


Cuando termina, devuelve la lista ya ordenada.

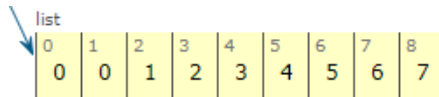


## -QUICK SORT-

```
1  from time import clock
2  def intercambia (A,x,y):
3      temp=A[x]
4      A[x]=A[y]
5      A[y]=temp
6  def particionar(A,p,r):
7      x=A[r]
8      i=p-1
9      for j in range (p,r):
10         if (A[j]<=x):
11             i=i+1
12             intercambia(A,i,j)
13         intercambia(A,i+1,r)
14         return i+1
15  def QuickSort(A,p,r):
16      if(p<r):
17         q=particionar(A,p,r)
18         QuickSort(A,p,q-1)
19         QuickSort(A,q+1,r)
20
21  lista = [0, 6, 0, 3, 2, 5, 7, 4, 1 ]
22  print ("Lista desordenada: {0} ".format(lista[1:]))
23  tiempoi = clock()
24  QuickSort(lista,1,len(lista)-1)
25  tiempof = clock()
26  print("\n\nLa lista arreglada: {0} ".format(lista[1:]))
27  print("\n\nEl tiempo de QuickSort es: {0} seg ".format((tiempof -
    tiempoi)))
```



Al inicio del programa se imprime la lista desordenada, se llama a la función *QuickSort* pasándole como parámetros a la lista, la variable p que es el índice desde donde empieza la lista, en este caso es 1 porque hay un cero antes, y la variable r que es el índice donde termina la lista. Dentro de la función se valida la condición de que si p es menor que r, entonces se le asigna a la variable q la llamada a la función *particionar* y se le pasa como parámetro la lista, y las variable p y r. Dentro de esta función se le asigna el valor de la lista en la posición r a la variable x, después a la operación p menos uno se le asigna a la variable i. Se entra en un ciclo for en el rango de "p" a "r", aquí se valida otra condición de que si el valor de la posición j en la lista original es menor o igual al valor de la variable x se realiza lo siguiente: la variable i aumenta en uno su valor y se llama a la función *intercambia*, pasándole como parámetro a la lista, la variable i y j. Dentro de esta función renombra a las variables y y j como "x" y "y" respectivamente y hace una transposición básica, para cambiar de lugar los valores almacenados en las variables. Al terminar el ciclo for vuelve a llamar a la función *intercambia* y le pasa como argumentos la lista, el valor de i+1 y r; nuevamente se hace el respectivo cambio por último la función *particionar* retorna el valor de i mas uno.



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

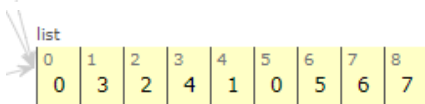
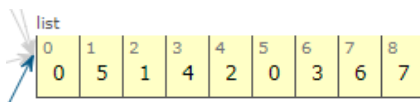
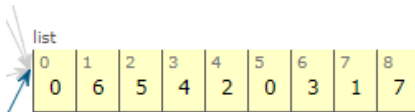
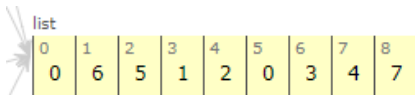
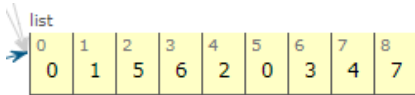
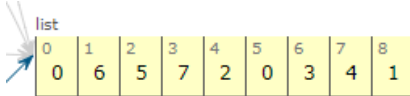
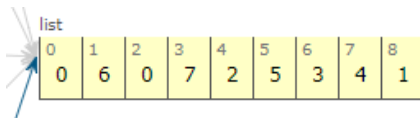
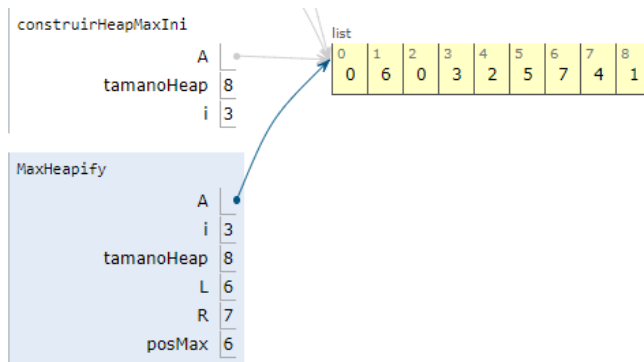
Posteriormente dentro de la función *QuickSort* y después de la llamada a la función *particionar*, se llama recursivamente a la misma función *QuickSort* pero ahora pasándole como argumentos, la lista, y las variables,  $p$  y  $q-1$ . Se repite el proceso descrito y cuando termina se llama de nuevo a la misma función pero ahora con los argumentos: la lista, y las variables  $q+1$  y  $r$ . Termina el proceso y la lista ya esta ordenada.

## -HEAP SORT-

```
4 ▾ def hIzq(i):
5     return 2*i
6 ▾ def hDer(i):
7     return 2*i+1
8 ▾ def intercambia(A,x,y):
9     tmp = A[x]
10    A[x] = A[y]
11    A[y] = tmp
12 ▾ def MaxHeapify(A,i,tamanoHeap):
13    L = hIzq(i)
14    R = hDer(i)
15    if (L <= tamanoHeap and A[L] > A[i]):
16        posMax = L
17    else:
18        posMax = i
19    if (R <= tamanoHeap and A[R] > A[posMax] ):
20        posMax = R
21    if (posMax != i):
22        intercambia(A,i,posMax)
23        MaxHeapify(A,posMax,tamanoHeap)
24 ▾ def construirHeapMaxIni(A,tamanoHeap):
25    for i in range(int(math.ceil(tamanoHeap/2)) -1,0,-1):
26        MaxHeapify(A,i,tamanoHeap)
27 ▾ def OrdemacioHeapSort(A,tamanoHeap):
28    construirHeapMaxIni(A,tamanoHeap)
29    for i in range(len(A[1:]),1,-1):
30        intercambia(A,1,i)
31        tamanoHeap=tamanoHeap-1
32    MaxHeapify(A,1,tamanoHeap)

34 lista = [0, 6, 0, 3, 2, 5, 7, 4, 1 ]
35 print("Arreglo desordenado: {0} ".format(lista[1:]))
36 tiempoi = clock()
37 OrdemacioHeapSort(lista,len(lista)-1)
38 tiempof = clock()
39 print("Arreglo ordenado: {0} ".format(lista[1:]))
40 print("Tiempo de HeapSort es: {0} seg ".format((tiempof - tiempoi
    )))
```





Para ordenar la lista, se llama a la función *OrdemacioHeapSort*, pasándole como argumento a la lista y al tamaño de la lista menos uno, esto será el tamaño del montículo o heap. Dentro de esta función llama a otra *construirHeapMaxIni*, dentro de esta hay un ciclo for que itera en un rango tomando como valor inicial el casteo del tamaño del heap entre dos y esa operación le resta un uno, se para en 0, y sigue iterando hasta -1. Dentro del ciclo llama a la función *MaxHeapFy* tomando como argumento a la lista, el valor de i de cada iteración y el tamaño del heap, dentro de esta función asigna a la variable L la multiplicación de 2 por i, y asigna a la variable R el valor de i multiplicado por dos y sumado un uno. Después valida la condición de que si el valor de L es menor o igual al tamaño del heap y el valor de L en la posición de la lista es mayor a el valor de i en la posición de la lista. Si se cumple asigna el valor de L a la variable posMax, si no la cumple le asigna el valor de i. Luego vuelve a validar una condición de que si R es menor o igual que el tamaño del heap y el valor de la posición en R de la lista es mayor al valor de la posición en posMax en la lista, si la cumple asigna el valor de R a la variable posMax .

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

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

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

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

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

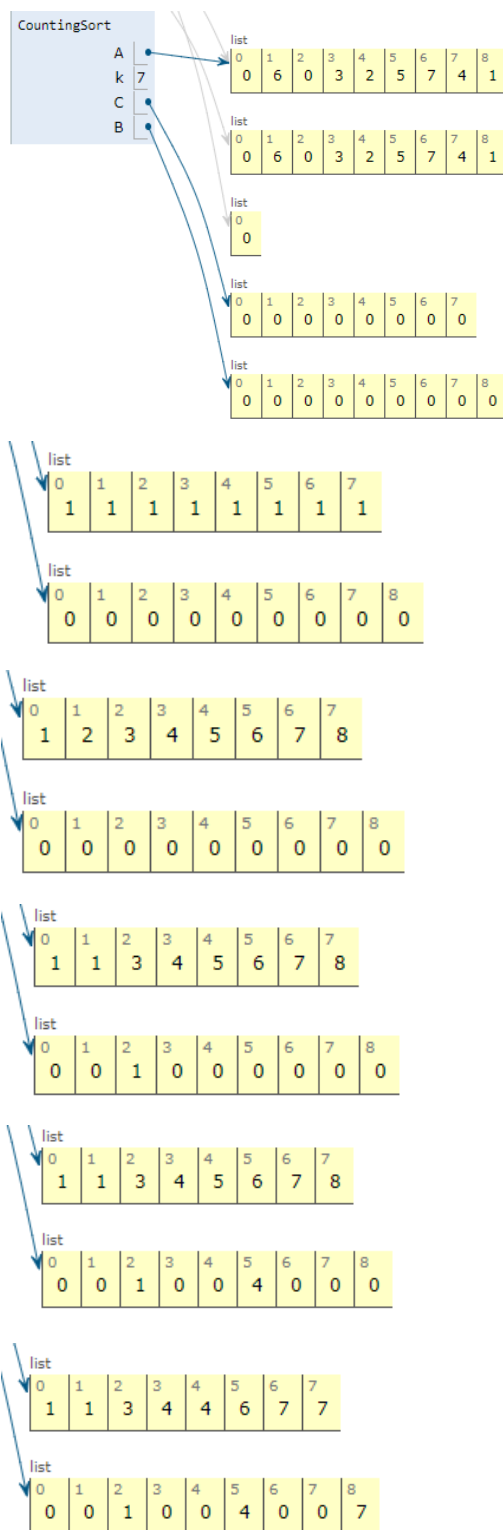
list	0	1	2	3	4	5	6	7	8
	0	0	1	2	3	4	5	6	7

Nuevamente hay una condición de que si posMax es diferente del valor de i, entonces llama a la función intercambia, pasándole como parámetro a la lista, el valor de i y posMax, realizando un intercambio de los valores contenidos en esas posiciones de la lista original, posteriormente llama recursivamente a la función MaxHeapfy pasándole como parámetro a la lista, y las variables, i el tamaño del heap.

Cuando termina el proceso de la construcción del heap, dentro de la función *OrdemacioHeapSort*, hay un ciclo for que itera desde el tamaño de la lista menos el primer índice, se para en uno y sigue iterando hasta menos uno, llama a la función intercambia pasando como parámetro a la lista, el valor de uno y el valor de i que es el contador. Después resta un uno al tamaño del heap y llama a la función *MaxHeapfy* pasando como parámetro a la lista, el valor de uno y el tamaño del heap. Finalmente la lista ha sido ordenada.

## -COUNTING SORT-

```
1 def CreaLista(k):
2     L = []
3     for i in range (k+1):
4         L.append(0)
5     return L
6 def CountingSort(A,k):
7     C = CreaLista(k)
8     B = CreaLista(len(A)-1)
9     for j in range(1,len(A)):
10        C[A[j]] = C[A[j]]+1
11    for i in range(1,k+1):
12        C[i] = C[i]+C[i-1]
13    for j in range(len(A)-1,0,-1):
14        B[C[A[j]]] = A[j]
15        C[A[j]] = C[A[j]]-1
16    return B
17 def Cambia(B):
18     for i in range(len(B)):
19         B[i] = B[i]*(-1)
20     return B
21 def CountingSortN(A,k):
22     C=CreaLista(k)
23     B=CreaLista(len(A)-1)
24     for j in range(1,len(A)):
25         C[A[j]]=C[A[j]]+1
26     for i in range(k-1,0,-1):
27         C[i]=C[i]+C[i+1]
28     for j in range (len(A)-1,0,-1):
29         B[C[A[j]]]=A[j]
30         C[A[j]]=C[A[j]]-1
31     return B
32 def Negativos(A):
33     for i in range(1,len(A)):
34         if A[i] < 0:
35             B.append(A[i])
36         else:
37             C.append(A[i])
38     Cambia(B)
39     return B
40
41 B=[]
42 C=[]
43 A = [0, 6, 0, 3, 2, 5, 7, 4, 1 ]
44 print("Lista inicial: ",A[1:])
45 Negativos(A)
46 B = CountingSortN(B,21)
47 Cambia(B)
48 C = CountingSort(C,7)
49 print("Lista ordenada: ",B[1:]+C[1:])
```



Para ordenar la lista, primeramente se validan si hay números negativos en ella, llamando a la función *Negativos*, en esta función separa a los números negativos y a los positivos en diferentes listas, a los negativos los cambia para tratarlos como positivos y poder usarlos en las funciones siguientes. Como esta lista solo contiene positivos, saltaremos la parte de ordenación de negativos. Se llama a la función *CountingSort* pasándole como argumentos a la lista y al número más grande contenido en ella, en este caso es el 7.

Esta función crea nuevamente dos listas, una la llena con tantos ceros sea el numero más grande y otra donde se almacenará la nueva lista ya ordenada, dentro de un ciclo itera desde uno hasta el tamaño de la lista y va usando la siguiente formula:  $C[A[j]] = C[A[j]] + 1$ , donde C es la lista de ceros y A la lista original, despues en otro ciclo for desde 1 hasta el valor del número más grande más uno, va asignando el valor resultado de la operación de i en la posición de la lista de ceros, más i menos uno en la posición de la lista de ceros a i en la posición de la misma lista. Luego en otro ciclo for itera desde el valor de el tamaño de la lista menos uno hasta menos uno, va asignando lo siguiente:  $B[C[A[j]]] = A[j]$   
 $C[A[j]] = C[A[j]] - 1$

Donde b es la lista ya ordenada y c la lista de ceros.

list

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

list

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

list

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

list

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

list

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

list

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

list

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

list

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

list

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

list

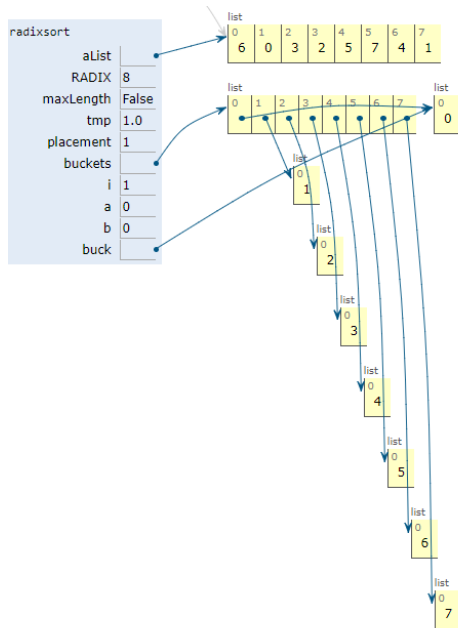
0	1	2	3	4	5	6	7	8
0	0	1	2	3	4	5	6	7

list

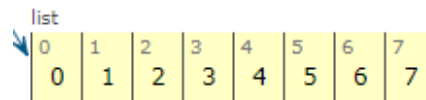
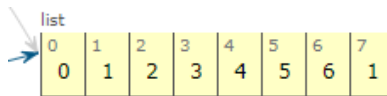
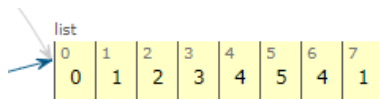
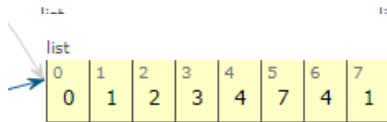
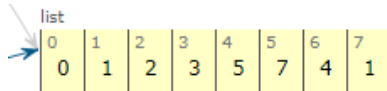
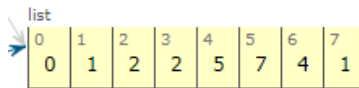
0	1	2	3	4	5	6	7	8
0	0	1	2	3	4	5	6	7

## -RADIX SORT-

```
1  from time import clock
2
3  def radixsort(aList):
4      RADIX=len(aList)
5      maxLength= False
6      tmp, placement = -1,1
7      while not maxLength:
8          maxLength=True
9          buckets=[list() for _ in range(RADIX)]
10         for i in aList:
11             tmp=i/placement
12             buckets[int(tmp%RADIX)].append(i)
13         if maxLength and tmp > 0:
14             maxLength=False
15         a=0
16         for b in range(RADIX):
17             buck=buckets[b]
18             for i in buck:
19                 aList[a]=i
20                 a+=1
21         placement*=RADIX
22
23  def radixsortInverso(aList):
24      RADIX=len(aList)
25      maxLength= False
26      tmp, placement = -1,1
27      while not maxLength:
28          maxLength=True
29          buckets=[list() for _ in range(RADIX)]
30         for i in aList:
31             tmp=i/placement
32             buckets[int(tmp%RADIX)].append(i)
33         if maxLength and tmp > 0:
34             maxLength=False
35         a=0
36         for b in range(RADIX):
37             buck=buckets[b]
38             for i in buck:
39                 aList[a]=i
40                 a-=1
41         placement*=RADIX
42
43
44  lista = [6, 0, 3, 2, 5, 7, 4, 1 ]
45
46  print("Lista desordenada: {0} ".format(lista))
47  tiempoi=clock()
48  radixsort(lista)
49  tiempof=clock()
50  print("\nLista ordenada: {0} ".format(lista))
51  print("Tiempo de Radixsort es: {0} seg".format((tiempof-tiempoi)))
52  ↓
```



Cabe destacar que para este programa no se inserta un cero al principio como en los pasados. Se llama a la función `radixsort` pasándole como argumento la lista. Se le asigna el tamaño de la lista a la variable `RADIX` y a la variable `maxLength`, que es la longitud máxima un falso; a las variables `tmp` y `placement` se les asigna un menos uno y uno respectivamente. Se entra en un ciclo `while`, mientras el valor de la longitud máxima sea lo contrario, cambia el valor a `true` y hace pequeñas listas, 8 en este caso y las rellena con el índice del tamaño de radix.



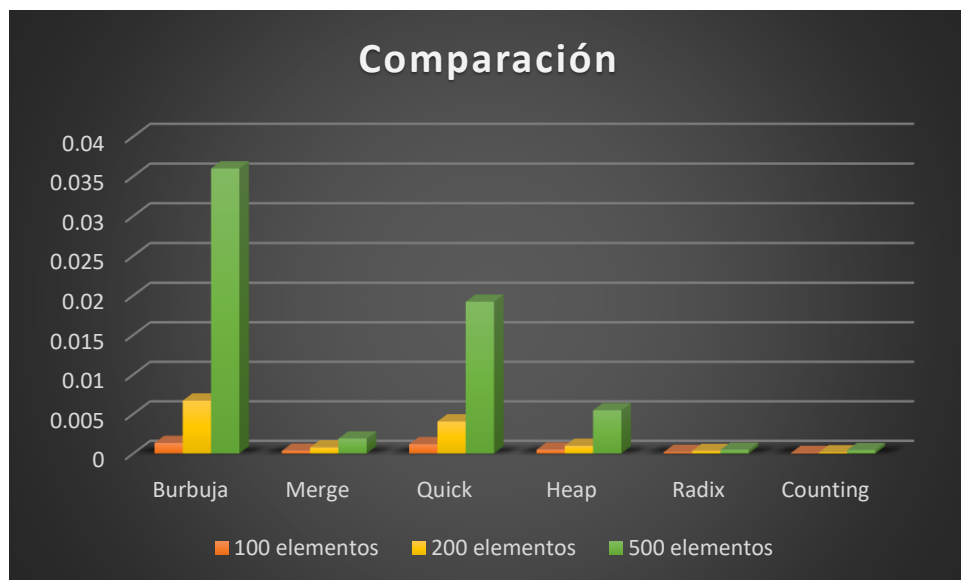
Después va recorriendo la lista original y asignando el nuevo valor donde corresponde para una vez terminado el proceso, la lista este arreglada.

13) Para un arreglo de 100 elementos, 200 elementos, 500 elementos, verificar tiempo de ordenamiento usando los métodos:

- Burbuja.
- Merge sort.
- Quick Sort.
- Heap Sort.
- Radix Sort.
- Counting Sort

Mostrar gráficamente los resultados y documentar lo obtenido.

	Burbuja	Merge	Quick	Heap	Radix	Counting
<b>100 elementos</b>	0.00133949	0.00035241	0.00118113	0.00048533	0.00020103	7.75E-05
<b>200 elementos</b>	0.00670686	0.00077826	0.00406317	0.00096984	0.00032738	0.00015672
<b>500 elementos</b>	0.03599708	0.00189743	0.01921434	0.00547487	0.00047097	0.00043077



De acuerdo con los datos obtenidos en la gráfica, podemos observar que el método de la burbuja es el menos eficiente de todos, hay una clara desventaja comparado con los demás ya que al ordenar los arreglos tardo mucha más que los otros, le sigue quick siendo el segundo meno eficiente, y heap que también tardo mucho tiempo en arreglar el arreglo de 500, aunque cabe destacar que al arreglar los arreglos de 200 y 100 va casi al mismo tiempo que merge, radix y counting fueron los más eficientes, siendo counting sort el más eficiente de todos los métodos.