



## Carátula para entrega de prácticas

Facultad de Ingeniería

Laboratorio de docencia

# Laboratorios de computación salas A y B

*Profesor:* Jesus Cruz Navarro

*Asignatura:* Programación Orientada a Objetos

*Grupo:* 01

*No de Práctica(s):* 03

*Integrante(s):* Hernandez Sarabia Jesus Ivan

*No. de Equipo de  
cómputo empleado:* No empleado

*No. de Lista o Brigada:*

*Semestre:* 2022-1

*Fecha de entrega:* 17 de septiembre de 2021

*Observaciones:*

**CALIFICACIÓN:** \_\_\_\_\_

**Objetivo:** El estudiante identificara la estructura de los algoritmos de ordenamiento Counting Sort y Radix Sort

## Código fuente:

1. Implementar las funciones que orden una lista de números utilizando los siguientes métodos:

A. CountingSort( arr ) .- Ordena lista de enteros positivos.

```
File Edit Selection View Go Run Terminal Help • CountingSort.py - Practica 3 - Visual Studio Code
CountingSort.py • RadixSort.py Main.py RadixSortLetritas.py
CountingSort.py > ...
1 import random # Importamos la biblioteca random para generar numero aleatorios
2 import math # Biblioteca para poder realizar operaciones aritmeticas
3
4 # Para poder calcular el elemento mas grande del arreglo
5 def countingSort(arr):
6
7     k = max(arr) # Sirve para encontrar el elemeto mas grande del arreglo
8
9     print("k:" , k) # Mostramos el valor de k
10
11     C = [0] * (k+1) # Creamos un arreglo con tantos elementos como el mayor numero del arreglo k
12
13
14     for i in range(len(arr)): # Se itera tantas veces como lementos tengamos en el arreglo
15         valor = arr[i]
16         C[valor] += 1
17
18     C[0] -= 1
19     for i in range(1, len(C)): # Se itera tantas veces como elementos tengamos en el arreglo C
20         C[i] = C[i] + C[i-1]
21     B = [0]*len(arr)
22
23     for j in range(len(arr)-1, -1, -1): # Se itera al reves tantas veces como elementos tengamos en el arreglo
24         valor = arr[j]
25         posicion = C[valor]
26         B[posicion] = valor
27         C[valor] -= 1
28     return B # Retornamos el arreglo B
29
30 a = [] # Arreglo
31 n = 10 # Numero de elemtnos del arreglo
32
33 for j in range(n): # Servira para poder llenar el arreglo
34     a.append(random.randint(0,2000)) # Guarmamos los numero generados aleatoriamente en el arrglo a
35
36
37 print("Arreglo desordenado: ", a) # Imprimimos el arreglo
38 print("Arreglo ordenado: ", countingSort(a)) # Llamamos a la funcion CountingSort para ordenar el arreglo
39
40
41
```

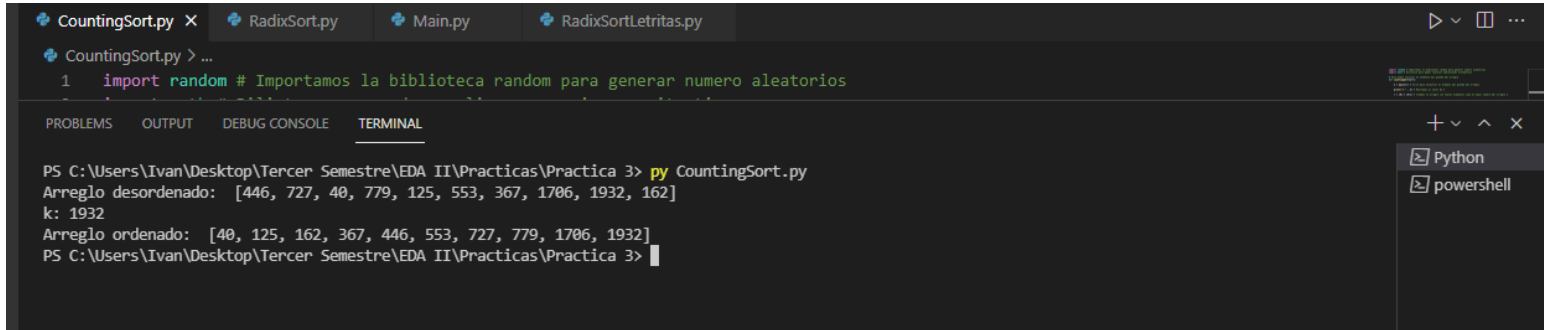
B. RadixSort\_numeros( arr ) .- Ordena listas de enteros positivos usando counting sort como algoritmo de ordenamiento auxiliar.

```
CountingSort.py  RadixSort.py  Main.py  RadixSortLetritas.py
RadixSort.py > ...
1  import random # Importamos la biblioteca random para generar numero aleatorios
2  import math # Bilioteca para poder realizar operaciones aritmetica
3
4  def radixSort(arr): # Funcion RadixSort para ordenar los elementos del arreglo
5      k = max(arr)
6      d = math.floor(math.log10(k)) + 1 # Opreacion aritmetica
7      print("k: ", k, "d: ", d)
8
9      for i in range(d):
10         arr = rSort(arr, 10, i)
11         print("arr en la itaracion ", i," es ", arr)
12     return arr
13
14 #Funcion para poder ordenar los elementos del arreglo, comparando las unidades, decenas, centenas y milesimas
15 def rSort(arr, b, i):
16     k = b - 1
17     C = [0] * (k + 1)
18
19     for j in range(len(arr)): # For que se ejetara segun el tamaño del arreglo
20         valor = arr[j]
21         digit = math.floor(valor/math.pow(b ,i)) % b # Opreacion aritmetica para calcular al numero mayor del arreglo
22         C[digit] += 1
23     C[0] -= 1
24
25     # Calcular la Frecuencia acumulada
26     for j in range(1, len(C)):
27         C[j] = C[j] + C[j-1]
28
29     B = [0] * (len(arr))
30
31     for j in range(len(arr)-1, -1, -1): # For para hacer la regresion al revers
32         valor = arr[j]
33         digit = math.floor(valor/math.pow(b ,i)) % b # Opreacion aritmetica
34         posicion = C[digit]
35         B[posicion] = valor
36
37         C[digit] -= 1
38
39     return B
```

```
40
41 a = [] # Arreglo
42 n = 10 # Numero de elemtnos del arreglo
43
44 for i in range (n): # Servira para poder llenar el arreglo
45     a.append(random.randint(0, 2000)) # Guarmamos los numero generados aletoriamente en el arrglo a
46
47 print("Arreglo desordenado: ", a) # Imprimimos el arreglo
48 print("arreglo ordenado", radixSort(a)) # Llamamos a la funcion RadixSort para ordenar el arreglo
49
50
```

## Ejecución de los programas.

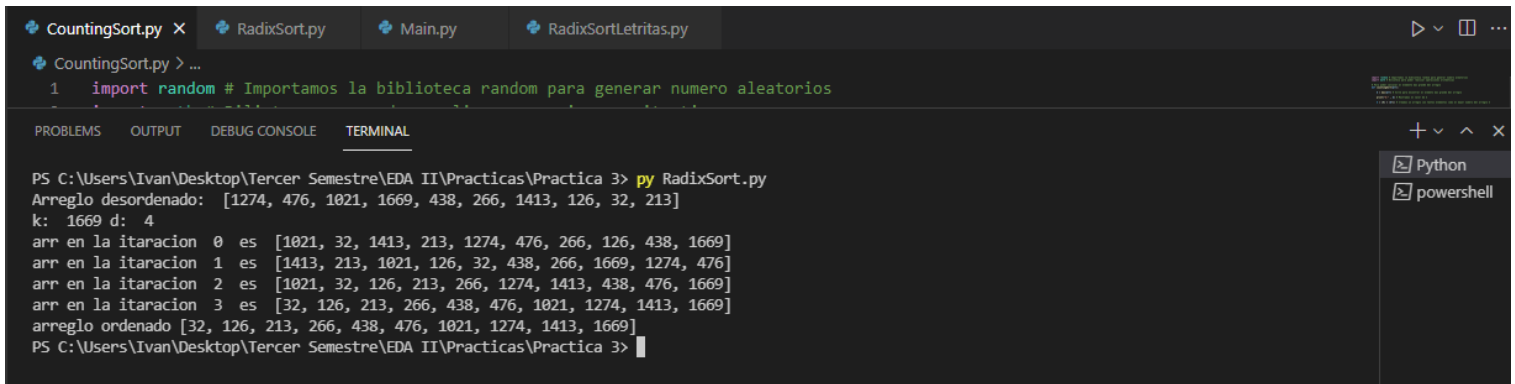
Para countingSort tenemos:



```
CountingSort.py X RadixSort.py Main.py RadixSortLetritas.py
CountingSort.py > ...
1 import random # Importamos la biblioteca random para generar numero aleatorios

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
PS C:\Users\Ivan\Desktop\Tercer Semestre\EDA II\Practicas\Practica 3> py CountingSort.py
Arreglo desordenado: [446, 727, 40, 779, 125, 553, 367, 1706, 1932, 162]
k: 1932
Arreglo ordenado: [40, 125, 162, 367, 446, 553, 727, 779, 1706, 1932]
PS C:\Users\Ivan\Desktop\Tercer Semestre\EDA II\Practicas\Practica 3>
```

Para radixSort tenemos:



```
CountingSort.py X RadixSort.py Main.py RadixSortLetritas.py
CountingSort.py > ...
1 import random # Importamos la biblioteca random para generar numero aleatorios

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
PS C:\Users\Ivan\Desktop\Tercer Semestre\EDA II\Practicas\Practica 3> py RadixSort.py
Arreglo desordenado: [1274, 476, 1021, 1669, 438, 266, 1413, 126, 32, 213]
k: 1669 d: 4
arr en la itaracion 0 es [1021, 32, 1413, 213, 1274, 476, 266, 126, 438, 1669]
arr en la itaracion 1 es [1413, 213, 1021, 126, 32, 438, 266, 1669, 1274, 476]
arr en la itaracion 2 es [1021, 32, 126, 213, 266, 1274, 1413, 438, 476, 1669]
arr en la itaracion 3 es [32, 126, 213, 266, 438, 476, 1021, 1274, 1413, 1669]
arreglo ordenado [32, 126, 213, 266, 438, 476, 1021, 1274, 1413, 1669]
PS C:\Users\Ivan\Desktop\Tercer Semestre\EDA II\Practicas\Practica 3>
```

2. Desarrolle un programa que ejecute de cada uno de los algoritmos implementados en el punto 1, y obtenga los tiempos de ejecución de cada una de las ejecuciones. Debe imprimir, el tiempo de ejecución para cada ejecución y el valor de k (para CountingSort). No es necesario imprimir en consola las listas.

```
CountingSort.py RadixSort.py Main.py mainP1.py RadixSortLetritas.py
Main.py > ...
1 import random # Importamos la biblioteca random para generar numero aleatorios
2 import math # Biblioteca para poder realizar operaciones aritmeticas
3 import time
4
5 # Para este programa tendremos las mismas
6 # funciones mostradas en el ejercicio anterior
7
8 # Para poder calcular el elemento mas grande del arreglo
9 > def countingSort(arr):...
33
34 def radixSort(arr): # Funcion RadixSort para ordenar los elementos del arreglo
35     k = max(arr)
36     d = math.floor(math.log10(k)) + 1 # Opreacion aritmetica
37     print("k: ", k, "d: ", d)
38
39 #Funcion para poder ordenar los elementos del arreglo, comparando las unidades, decenas, centenas y milesimas
40 > def rSort(arr, b, i):...
66
```

```
CountingSort.py RadixSort.py Main.py mainP1.py RadixSortLetritas.py
Main.py > ...
69
70 print("##### Ordenamiento para ", n, " elementos en el arreglo. #####\n\n")
71
72 print("///// Tiempo de ordenamiento para k:", n, " /////\n") # Mostramos cuantos elementos contendra la lista
73 print("----- Mejor Caso (Lista aleatoria) ----- \n\n")
74
75 print("Para countingSort tenemos: \n")
76
77 for i in range(n): # El ciclo se ejecurara tantas veces como elementos tengamos en la lista
78     aa.append(random.randint(0, n//2)) #Codigo para generar numeros aleatorios del 0 a 10
79
80 a1 = aa[:] # Copiamos la lista generada para asignarlo a cada algoritmo creado
81 a2 = aa[:]
82
83 inicio = time.time()
84 mejorCounting = countingSort(a1)
85 fin = time.time()
86 tiempoFinal = fin - inicio
87 print(tiempoFinal, " s")
88
89 print("Para RadixSort tenemos: \n")
90
91 inicio = time.time()
92 mejorRadix = radixSort(a2)
93 fin = time.time()
94 tiempoFinal = fin - inicio
95 print(tiempoFinal, " s")
96
97
98 print("\n\n///// Tiempo de ordenamiento para k:", n*10000, " /////\n") # Mostramos cuantos elementos contendra la lista
99 print("----- Peor Caso (Lista aleatoria) ----- \n\n")
100
101 ab = [] # Para guardar los elementos de la lista cuando k = n*10000
102
103 for i in range(n): # El ciclo se ejecurara tantas veces como elementos tengamos en la lista
104     ab.append(random.randint(0, n*10000))
105
106 a3 = ab[:] # Copiamos la lista generada para asignarlo a cada algoritmo creado
107 a4 = ab[:]
108
109 print("Para countingSort tenemos: \n")
110
111 inicio = time.time()
112 peorCounting = countingSort(a3)
113 fin = time.time()
114 tiempoFinal = fin - inicio
```

```
CountingSort.py RadixSort.py Main.py mainP1.py RadixSortLetritas.py
Main.py > ...
106 a3 = ab[:] # Copiamos la lista generada para asignarlo a cada algoritmo creado
107 a4 = ab[:]
108
109 print("Para countingSort tenemos: \n")
110
111 inicio = time.time()
112 peorCounting = countingSort(a3)
113 fin = time.time()
114 tiempoFinal = fin - inicio
115 print(tiempoFinal, " s")
116
117 print("Para RadixSort tenemos: \n")
118
119 inicio = time.time()
120 mejorRadix = radixSort(a4)
121 fin = time.time()
122 tiempoFinal = fin - inicio
123 print(tiempoFinal, " s")
124
125 print("\n\n///// Tiempo de ordenamiento para k:", n, " /////\n") # Mostramos cuantos elementos contendra la lista
126 print("----- Caso promedio (Lista aleatoria) ----- \n")
127
128 ac = [] # Para guardar los elementos de la lista cuando k = n
129
130 for i in range(n): # El ciclo se ejecutara tantas veces como elementos tengamos en la lista
131     ac.append(random.randint(0, n))
132
133 a5 = ac[:] # Copiamos la lista generada para asignarlo a cada algoritmo creado
134 a6 = ac[:]
135
136 print("Para countingSort tenemos: \n")
137
138 inicio = time.time()
139 promedCounting = countingSort(a5)
140 fin = time.time()
141 tiempoFinal = fin - inicio
142 print(tiempoFinal, " s")
143
144 print("Para RadixSort tenemos: \n")
145
146 inicio = time.time()
147 mejorRadix = radixSort(a6)
148 fin = time.time()
149 tiempoFinal = fin - inicio
150 print(tiempoFinal, " s")
```

## Ejecución del programa:

```
CountingSort.py RadixSort.py Main.py mainP1.py RadixSortLetritas.py
Main.py > ...
102
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
##### Ordenamiento para 5000 elementos en el arreglo. #####

///// Tiempo de ordenamiento para k: 5000 /////

----- Mejor Caso (Lista aleatoria) -----

Para countingSort tenemos:

k: 2500
0.0049991607666015625 s

Para RadixSort tenemos:

k: 2500 d: 4
0.11890721321105957 s

///// Tiempo de ordenamiento para k: 500000000 /////
```

```
///// Tiempo de ordenamiento para k: 500000000 /////

----- Peor Caso (Lista aleatoria) -----

Para countingSort tenemos:

k: 49977044
55.73724699020386 s
Para RadixSort tenemos:

k: 49977044 d: 8
0.025975465774536133 s

///// Tiempo de ordenamiento para k: 5000 /////

----- Caso promedio (Lista aleatoria) -----

Para countingSort tenemos:

k: 4999
0.027997493743896484 s
Para RadixSort tenemos:

k: 4999 d: 4
0.0009984970092773438 s
PS C:\Users\Ivan\Desktop\Tercer Semestre\EDA II\Practicas\Practica 3>
```

3. Pruebe el programa del punto 2 con listas de enteros para  $n = 5000$ ,  $10000$ , y  $20000$  datos, para el peor caso (e.j.  $k = n \cdot 10000$ ), mejor caso ( $k \ll n$ ) y caso promedio ( $k \approx n$ ). Dado que los tiempos pueden variar, ejecute la prueba 3 veces y obtenga el promedio de los tiempos.

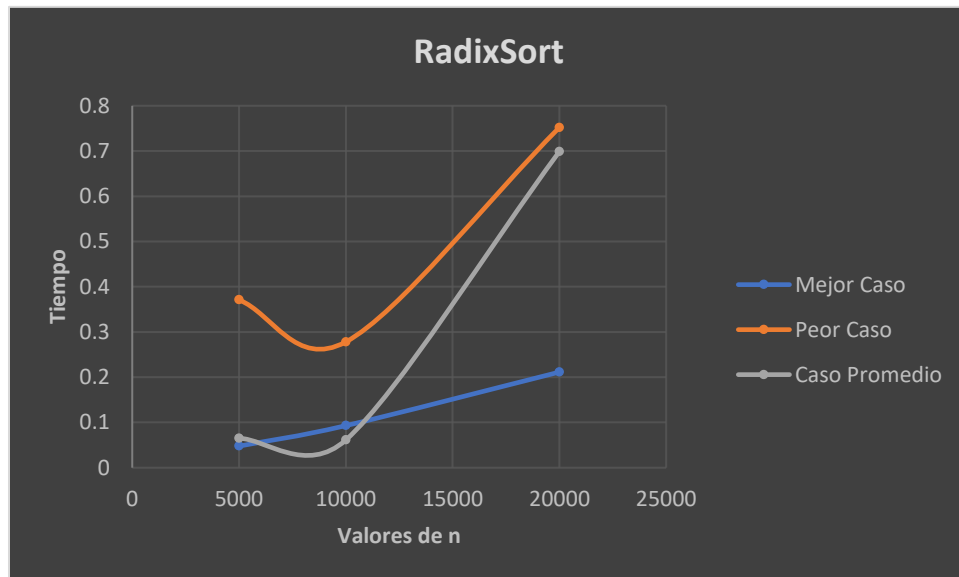
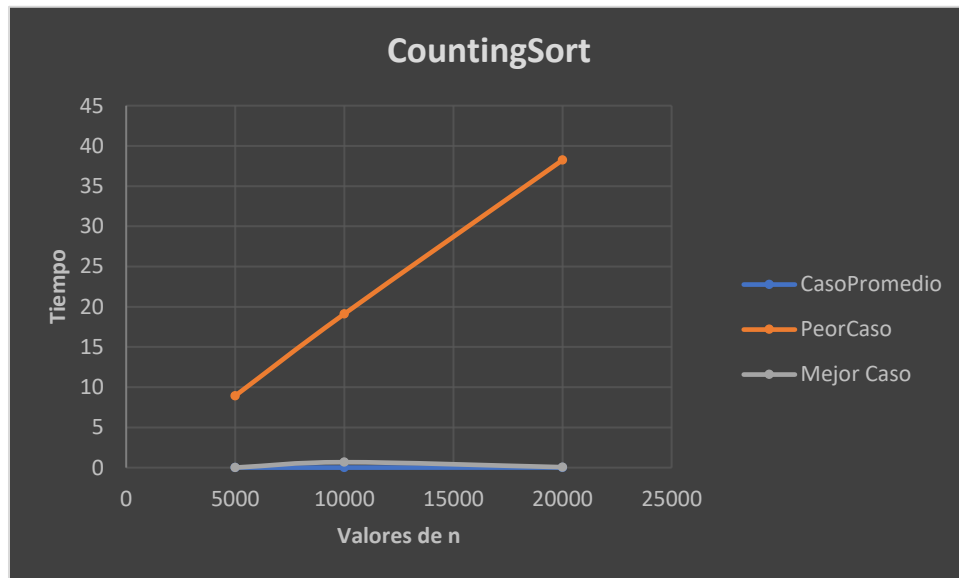
| :)       | Valores obtenidos |             |
|----------|-------------------|-------------|
|          | Mejor caso        |             |
|          | CountingSort      | RadixSort   |
| 5000     | 0.003000975       | 0.038002729 |
|          | 0.00563463        | 0.057373727 |
|          | 0.00756536        | 0.0476786   |
| Promedio | 0.005400322       | 0.047685019 |
| 10000    | 0.003001451       | 0.071847439 |
|          | 0.004865351       | 0.093647378 |
|          | 0.004758347       | 0.113547344 |
| Promedio | 0.004208383       | 0.093014053 |
| 20000    | 0.009000301       | 0.187015057 |
|          | 0.012535343       | 0.253453151 |
|          | 0.009234234       | 0.193535506 |
| Promedio | 0.010256626       | 0.211334571 |

| :)       | Valores obtenidos |             |
|----------|-------------------|-------------|
|          | Caso promedio     |             |
|          | CountingSort      | RadixSort   |
| 5000     | 0.002999783       | 0.040003538 |
|          | 0.004525252       | 0.056783538 |
|          | 0.009568378       | 0.098364235 |
| Promedio | 0.005697804       | 0.065050437 |
| 10000    | 0.005001068       | 0.075998068 |
|          | 0.007574681       | 0.01253942  |
|          | 2.009837459       | 0.0952526   |
| Promedio | 0.674137736       | 0.061263362 |
| 20000    | 0.01600194        | 0.455036402 |
|          | 0.093424235       | 0.986785504 |
|          | 0.05454356        | 0.654645504 |
| Promedio | 0.054656578       | 0.69882247  |

| :)       | Valores obtenidos |             |
|----------|-------------------|-------------|
|          | Peor caso         |             |
|          | CountingSort      | RadixSort   |
| 5000     | 8.797364391       | 0.081006765 |
|          | 9.47668314        | 0.105352101 |
|          | 8.539564831       | 0.926406765 |
| Promedio | 8.937870788       | 0.370921877 |
| 10000    | 18.93338108       | 0.210014343 |
|          | 19.82856491       | 0.349545454 |
|          | 18.57395739       | 0.274729798 |
| Promedio | 19.11196779       | 0.278096532 |
| 20000    | 36.39952445       | 0.364031076 |
|          | 37.83685763       | 0.757833108 |
|          | 40.46283645       | 1.134243108 |
| Promedio | 38.23307284       | 0.752035764 |



4. Grafique (graficas de líneas, no barras), para cada algoritmo los tiempos de ejecución promedio (n vs t) obtenidos en el punto 3 comparando cada uno de los casos. Es decir, deben ser 3 gráficas en total, una para cada algoritmo y en cada una deben venir las gráficas de los 3 casos para poder compararlos.



5. Conteste las siguientes preguntas:

a. ¿Por qué el último for de CountingSort se debe realizar de atrás para adelante?

R: El último for de CountingSort se realiza de atrás para adelante debido a que la forma en la que trabaja CountingSort que consiste en, que este encuentra el elemento más grande del arreglo, y por ende coloca este al final de dicho arreglo por la misma razón, entonces se itera de atrás hacia adelante debido a que como el último elemento del arreglo ya está ordenado debe ordenar los elementos de mayor a menor iniciando de atrás hacia adelante, por eso este ciclo se itera de dicha manera.

2. Desarrolle un programa que ejecute de cada uno de los algoritmos implementados en el punto 1, y obtenga los tiempos de ejecución de cada una de las ejecuciones. Debe imprimir, el tiempo de ejecución para cada ejecución y el valor de k (para CountingSort). No es necesario imprimir en consola las listas.

Programa 2:

1. Desarrolle un programa que ordene cadenas de longitud variable modificando el algoritmo de RadixSort visto en clase. Debe utilizar el orden lexicográfico, es decir, alfabéticamente donde el carácter más significativo es el primero, luego el segundo y así sucesivamente.

TIPS:

- Use el código ascii de los caracteres como índice para el arreglo auxiliar C. Para esto puede usar la función `ord()` de Python.
- Use un arreglo C de tamaño 255 para los 255 caracteres ASCII.
- Al obtener un carácter de la cadena, verifique si existe el índice comparando la longitud de la cadena. Si no existe carácter en la posición i, puede utilizar un carácter de espacio. A diferencia de la versión numérica, los ceros irían del lado izquierdo si no existen más caracteres. Ej:2 / 3

En la versión numérica => 001; En la versión con cadenas “Juan” (espacios del lado derecho)

2. Pruebe el programa con una lista de 10 palabras

```
RadixSortLetritas.py > ...
1
2 def RadixSortLetritas( arr ):
3
4     arrCadena = max(arr, key=len) # Obtenemos el String mas largo del arreglo y le asignamos k
5     d = len(arrCadena) # Obtenemos la longitud de la cadena k y la igualamos a d
6     for i in range( len(arr) ): # Iteramos tantas veces como elementos tenemos en el arreglo
7         while len(arr[i]) < d:
8             arr[i] = arr[i] + " "
9     for i in range( d-1,-1,-1 ): # Sirve para iterar el for de atras hacia adelante tantas veces como es el valor de d
10        arr = rSort( arr, len(arr), i )
11
12 # Codigo para poder eliminar los espacios que puedan dejar dentro de los elementos del arreglo
13 contadorIndice = 0 # Servira para contar los indices de la lista
14 for cadena in arr:
15     i=len(cadena) - 1
16     contEspacios = 0 # Servira para contar los espacios de cada string del arreglo
17
18     while i > -1: # Iteramos el while alreves para que cuente los elementos de la cadena
19         if cadena[i] == ' ': # Si encontramos espacios en la cadena
20             contEspacios += 1
21         else:
22             break
23         i -= 1 # Le restamos uno para que vaya iterando letra por letra de la cadena al reves
24
25     if contEspacios >= 1: # Sirve para comparar si el contador de espacios contiene elementos
26         aux = cadena[ : - 1 * contEspacios]
27         arr[contadorIndice] = aux
28
29     contadorIndice += 1 # Le sumamos uno al contador para que vaya iterando cada indice de la lista
30
31 return arr
32
```

```

32
33 def rSort( arr,b,i ):
34
35     C = [0]*256 # Arreglo para almacenar k elementos con tantos elementos como la tabla ASCII
36
37     for j in range( b ):
38         cadena = arr[j]
39         valor = ord(cadena[i])
40         C[valor] += 1
41
42     # Calcular la Frecuencia acumulada
43     for j in range( 1, len(C) ):
44         C[j] += C[j-1]
45     B = [0]*b
46
47     for j in range( b-1,-1,-1 ): # Arreglo que se iterara de atras hacia adelante
48         cadena = arr[j]
49         valor = ord(cadena[i])
50         posicion = C[valor]
51         B[posicion - 1] = cadena
52         C[valor] -= 1
53     return B
54

```


```


55
56 a = ["Scarlett Johansson", "Chris Hemsworth", "Chris Evans", "Robert Downey Jr", "Mark Ruffalo", "Jeremy Renner", "Brie Larson",
57 "Paul Rudd", "Karen Gillan", "Don Cheadle", "Tom Holland", "Elizabeth Olsen"]
58
59 print()
60 print("\t//////// RadixSort para ordenar Strings //////////")
61
62 print("\nArreglo Desordenado:\n", a) # Mostramos el arreglo desordenado
63 # Llamamos a la funcion radixSortLetritas para odernar el arreglo
64 print("\nArreglo Ordenado:\n", RadixSortLetritas(a)) # Mostramos la lista ordenada
65
66

```

## Ejecución del programa:

```
PS C:\Users\Ivan\Desktop\Tercer Semestre\EDA II\Practicas\Practica 3> py RadixSortLetritas.py
```

 Python

 powershell

```
//////// RadixSort para ordenar Strings //////////
```

Arreglo Desordenado:

```
['Scarlett Johansson', 'Chris Hemsworth', 'Chris Evans', 'Robert Downey Jr', 'Mark Ruffalo', 'Jeremy Renner', 'Brie Larson', 'Paul Rudd', 'Karen Gillan', 'Don Cheadle', 'Tom Holland', 'Elizabeth Olsen']
```

Arreglo Ordenado:

```
['Brie Larson', 'Chris Evans', 'Chris Hemsworth', 'Don Cheadle', 'Elizabeth Olsen', 'Jeremy Renner', 'Karen Gillan', 'Mark Ruffalo', 'Paul Rudd', 'Robert Downey Jr', 'Scarlett Johansson', 'Tom Holland']
```

```
PS C:\Users\Ivan\Desktop\Tercer Semestre\EDA II\Practicas\Practica 3> █
```

## Conclusiones:

Hernández Sarabia Jesús Ivan: Una vez finalizados los ejercicios de la practica de laboratorio podemos concluir que RadixSort es un algoritmo de ordenamiento más eficiente que CountingSort, esto lo podemos ver reflejado en las graficas o tablas donde recabamos los datos de cuanto se tarda en ejecutar para los tres tipos de casos que tenemos, es debido a que radixSort a comparación de countingSort realiza nada más las comparaciones con las unidades, decenas, centenas, milésimas, etc., para ordenar los elementos de los arreglos que generamos, a diferencia de CountingSort que genera más elementos en arreglos para ordenar los elementos del arreglo original por lo que esto le lleva más tiempo poder ordenar los números dados en el arreglo original.