| | |
|---|---|
| | **Carátula para entrega de prácticas** |
| Facultad de Ingeniería | Laboratorio de docencia |

# Laboratorios de computación salas A y B

*Professor:* Jesus Cruz Navarro

*Asignatura:* Estructura de Datos y Algoritmos 2

*Grupo:* 1

*No de Práctica(s):* 01

*Integrante(s):* Diego Santiago Gutiérrez

*No. de Equipo de cómputo empleado:*

*No. de Lista o Brigada:*

*Semestre:* Tercer Semestre

*Fecha de entrega:* 19/10/2020

*Observaciones:*

CALIFICACIÓN: __________

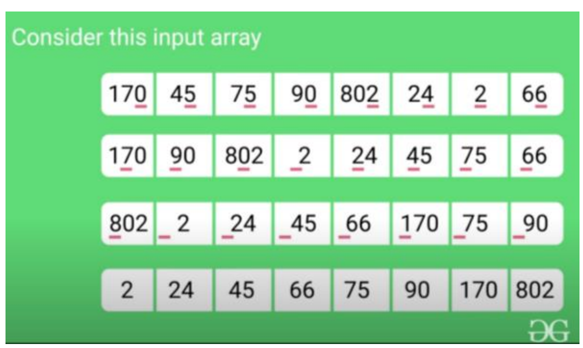
## PRACTICA 03: ALGORITMO DE ORDENAMIENTO

**OBJETIVO:** El estudiante conocerá e identifica la estructura de los algoritmos de ordenamiento Counting Sort y Radix Sort

## INTRODUCCIÓN:

Counting sort es un algoritmo de ordenamiento en el que se cuenta el número de elementos de cada clase para luego ordenarlos. Solo puede ser utilizado por tanto para ordenar elementos que sean contables (como los números enteros en un determinado intervalo, pero no los números reales, por ejemplo)


**Radix** es un algoritmo de ordenamiento que ordena enteros procesando sus dígitos de forma individual. Como los enteros pueden representar cadenas de caracteres (por ejemplo, nombres o fechas) y, especialmente, números en punto flotante especialmente formateados, *radix sort* no está limitado sólo a los enteros.
Existen dos clasificaciones de radix sort: el de dígito menos significativo (**LSD**) y el de dígito más significativo (**MSD**). *Radix sort LSD* procesa las representaciones de enteros empezando por el dígito menos significativo y moviéndose hacia el dígito más significativo. *Radix sort MSD* trabaja en sentido contrario.



## REQUERIMIENTOS:

:
1. Programar los algoritmos de ordenamiento vistos en clase. Cada algoritmo debe estar en una función. Estos son:
a. CountingSort
b. RadixSort (para enteros, usando CountingSort para ordenar internamente)

2. Desarrollar un programa para probar los algoritmos implementados anteriormente y registrar sus tiempos de ejecución con listas de tamaño n. Para probar, ambos algoritmos deben recibir el mismo conjunto de elementos (es decir, si es una lista aleatoria, los algoritmos deben recibir el mismo conjunto aleatorio). El programa debe imprimir al final n y el tiempo de ejecución de cada algoritmo.Se debe probar con listas de diferentes tamaños (n= 1000, 200, 5000, 10000) tiempos de ejecución para cada uno de los siguientes casos Mejor Caso, Pero Caso y Caso Promedio:
a. Debe explicar cuál es el peor, mejor y caso promedio para estos y diseñar las entradas para validar dichos casos.

3. Dado que los tiempos pueden variar de una ejecución a otra, realice las pruebas 3 veces y obtenga el promedio de los tiempos
.
4. Genere una gráfica para cada caso (3 gráficas), para comparar los tiempos de ambos algoritmos en función del tamaño de la lista (n vs t, es decir, las n en el eje X y los tiempos en segundos en el eje Y).

## DESARROLLO:

### 1.a) Counting Sort

```python
def CountingSort(A):
    n = len(A) #n es del tamaño del arreglo
    k = max(A) #k es el valor más alto del arreglo
    C = [0]*(k+1) #Una unidad más que el valor maximo sera el numero de elementos que serán rellenados con 0
    B = [0]*(n) #Una lista llena de 0 que tenga el tamaño del arreglo que estaremos trabajando

    for j in range(n): #For de j al tamaño del arreglo(Para ir llenando)
        C[A[j]] += 1 #El arreglo de C (que es el que tiene una unidad mas que k) en la posicion 0 del arreglo A se incrementará e6n uno si el elemento
                     #El numero en la posicion de j de A será el numero que guardará registro de su existencia en la posicion del arreglo
                     #Es decir, si A[0]=3, entonces C[3]= 1 y este for iterará hasta que se acabe el arreglo

    for j in range(1, k+1): #for de j a el valor maximo mas uno, es decir si el valor maximo es 15, el ciclo irá de 1 a 16
        C[j] += C[j-1] #C[j] su valor se sumara con la posición anteior y se asigará a la posicion de C[j]

    for j in range(n-1,-1,-1): #For ira del tamaño del arreglo hasta -1 e irra decrementando en uno
        B[C[A[j]]-1] = A[j] #Con en arrelgo B que es el que tiene el tamaño del arreglo A
                            #Buscaremos el valor que hay en el inciso a[j] y se le restará una unidad, ejemplo si en su valor es A[j] = 1 -> A[j] = 0
                            #Con el numero del arreglo anterior, se buscará en el indice del arreglo de C[0] siguiendo el ejemplo de arriba
                            #Para que el valor que este en C[0] sea el idice del arreglo B y posterior a eso se tendrá que asignarle el valor de A[j] que era uno

        C[A[j]] -= 1 #Para esto se le resta el valor del contador que llevamos para dar a entender que ese elemnto ha sido acomodado
    return B #Regresa el arreglo ordenado
```

### 1.b) Radix Sort

```python
def RadixSort(A): #Creamos nuestra funcion llamada RadixSort que recibe un arreglo
    k = max(A) #Sacamos el valor maximo del arreglo
    d = (int)(math.log10(k)+1) #Creamos d que por medio de una libreria sacamos el valor logaritmico en base diez del valor mas alto del arreglo
    for i in range (d):# Veces que tendremos que hacer este proceso del calculo del logaritmo del valor más alto del arreglo
        A = CountingSort2(A,i) #El arreglo sera recursivo y lo que tengamos en A lo iremos procesando con CountingSort2
    return A #Retornamos el valor del arreglo ya cuando acabe el arreglo

def CountingSort2(A,i): #Declaramos nuestra funcion que recibirá un arreglo y le pasas el indice del arreglo de Radix
    n = len(A) #Sacas el tamaño del arreglo en A
    k = 9
    C = [0]*(k+1) # llenamos el k+1 espacios de un arreglo con 0
    B = [0]*(len(A)) #Creamos y lleanamos el arreglo B con el tamaño del arreglo A con puros 0

    for j in range(n): #For del tamaño de i hasta el tamaño del arreglo a usar
        val = A[j] #Val se le asigna el valor del arreglo en la posicion j
        di = (val // 10**i) % 10 #di nos ayuda a encontrar el modulo del valor mas pequeño de todos los valores, obteniendo la unidad de los numeros que tengamos
        C[di] += 1 #C ira almacenando y recorriendo para poder hacer este proceso

    for j in range(1, k+1): #j en el rango de (1 a k+1)
        C[j] += C[j-1] #La posicion en C[j] se sumará al valor que este dentro de la posicion c(j-1) para asignarlo al mismo C[j]

    for j in range(n-1,-1,-1): #Ira del tamaño del arreglo hasta -1, es decir que no haya elemetnos y este ira recorriendo de menos uno en menos uno
        val = A[j] #val se le asignará el valor de lo que tenga la poscicion A[j]
        di = (val // 10**i) % 10 #Sacamos el valo de la unidad para ir comparando con los valores
        B[C[di]-1] = val #El valor del arrego C[di] menos uno, sera el indice para el arreglo de B para poder guardarlo en val
        C[di] -= 1 #Retiramos un
    return B # regresamos el arreglo ya ordenado de A
```

Nota: He decido explicar el programa dentro del mismo, creo que es mejor para el maestro y para mí y es continua la alimentación de la explicación.

En el main del programa podemos solo encontrar las creaciones de los arreglos mejores, normales y peores, su toma de tiempo para cada uno de los casos y sus impresiones de tiempos y de n.

# RESULTADOS:

N=1000

```
Tiempo RadixSort normal: 0.003002 segundos
n:  1000

Tiempo RadixSort MEJOR: 0.005062 segundos
n:  1000

Tiempo RadixSort PEOR: 0.007927 segundos
n:  1000

Tiempo CountingSort NORMAL: 0.000000 segundos

n:  1000

Tiempo CountingSort MEJOR: 0.001001 segundos

n:  1000

Tiempo CountingSort PEOR: 0.001000 segundos

n:  1000
```

```
Tiempo RadixSort normal: 0.002000 segundos
n:  1000

Tiempo RadixSort MEJOR: 0.005006 segundos
n:  1000

Tiempo RadixSort PEOR: 0.008005 segundos
n:  1000

Tiempo CountingSort NORMAL: 0.000000 segundos

n:  1000

Tiempo CountingSort MEJOR: 0.000987 segundos

n:  1000

Tiempo CountingSort PEOR: 0.001115 segundos

n:  1000
```

```
Tiempo RadixSort normal: 0.001997 segundos
n:  1000

Tiempo RadixSort MEJOR: 0.004128 segundos
n:  1000

Tiempo RadixSort PEOR: 0.005875 segundos
n:  1000

Tiempo CountingSort NORMAL: 0.001003 segundos

n:  1000

Tiempo CountingSort MEJOR: 0.000995 segundos

n:  1000

Tiempo CountingSort PEOR: 0.001017 segundos

n:  1000
```

N=2000

```
Tiempo RadixSort normal: 0.005001 segundos
n:  2000

Tiempo RadixSort MEJOR: 0.015129 segundos
n:  2000

Tiempo RadixSort PEOR: 0.015005 segundos
n:  2000

Tiempo CountingSort NORMAL: 0.000999 segundos

n:  2000

Tiempo CountingSort MEJOR: 0.002002 segundos

n:  2000

Tiempo CountingSort PEOR: 0.002993 segundos

n:  2000
```

```
Tiempo RadixSort normal: 0.005885 segundos
n:  2000

Tiempo RadixSort MEJOR: 0.015003 segundos
n:  2000

Tiempo RadixSort PEOR: 0.016062 segundos
n:  2000

Tiempo CountingSort NORMAL: 0.000946 segundos

n:  2000

Tiempo CountingSort MEJOR: 0.002099 segundos

n:  2000

Tiempo CountingSort PEOR: 0.002875 segundos

n:  2000
```

```
Tiempo RadixSort normal: 0.006003 segundos
n:  2000

Tiempo RadixSort MEJOR: 0.015006 segundos
n:  2000

Tiempo RadixSort PEOR: 0.016005 segundos
n:  2000

Tiempo CountingSort NORMAL: 0.002027 segundos

n:  2000

Tiempo CountingSort MEJOR: 0.001973 segundos

n:  2000

Tiempo CountingSort PEOR: 0.001998 segundos

n:  2000
```

## N=5000

```
Tiempo RadixSort normal: 0.013134 segundos
n:  5000

Tiempo RadixSort MEJOR: 0.092022 segundos
n:  5000

Tiempo RadixSort PEOR: 0.045007 segundos
n:  5000

Tiempo CountingSort NORMAL: 0.002997 segundos

n:  5000

Tiempo CountingSort MEJOR: 0.005003 segundos

n:  5000

Tiempo CountingSort PEOR: 0.007000 segundos

n:  5000
```

```
Tiempo RadixSort normal: 0.015007 segundos
n:  5000

Tiempo RadixSort MEJOR: 0.042010 segundos
n:  5000

Tiempo RadixSort PEOR: 0.047008 segundos
n:  5000

Tiempo CountingSort NORMAL: 0.002998 segundos

n:  5000

Tiempo CountingSort MEJOR: 0.007008 segundos

n:  5000

Tiempo CountingSort PEOR: 0.003998 segundos

n:  5000
```

```
Tiempo RadixSort normal: 0.016102 segundos
n:  5000

Tiempo RadixSort MEJOR: 0.041004 segundos
n:  5000

Tiempo RadixSort PEOR: 0.064010 segundos
n:  5000

Tiempo CountingSort NORMAL: 0.003004 segundos

n:  5000

Tiempo CountingSort MEJOR: 0.008001 segundos

n:  5000

Tiempo CountingSort PEOR: 0.004128 segundos

n:  5000
```

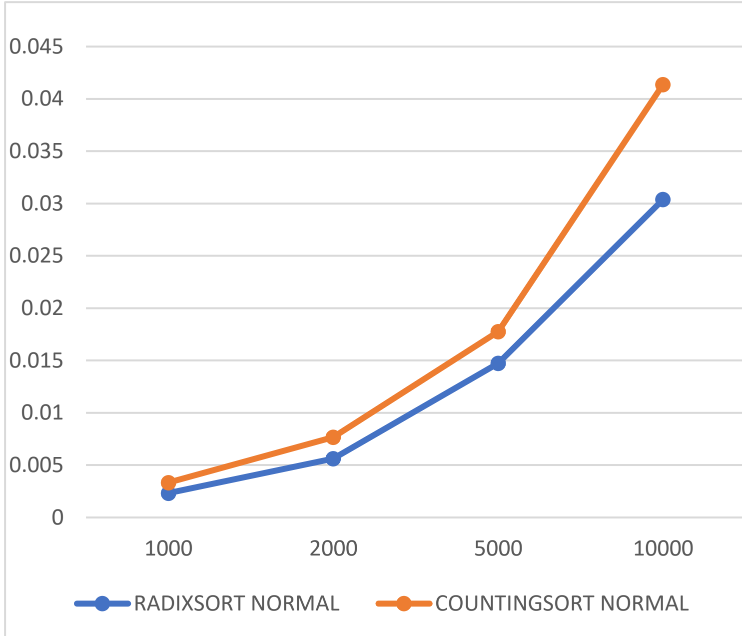
## N=10000

```
Tiempo RadixSort normal: 0.026008 segundos
n:  10000

Tiempo RadixSort MEJOR: 0.100026 segundos
n:  10000

Tiempo RadixSort PEOR: 0.102023 segundos
n:  10000

Tiempo CountingSort NORMAL: 0.010004 segundos

n:  10000

Tiempo CountingSort MEJOR: 0.012990 segundos

n:  10000

Tiempo CountingSort PEOR: 0.012978 segundos

n:  10000
```

```
Tiempo RadixSort normal: 0.028005 segundos
n:  10000

Tiempo RadixSort MEJOR: 0.094022 segundos
n:  10000

Tiempo RadixSort PEOR: 0.118028 segundos
n:  10000

Tiempo CountingSort NORMAL: 0.037019 segundos

n:  10000

Tiempo CountingSort MEJOR: 0.021003 segundos

n:  10000

Tiempo CountingSort PEOR: 0.049012 segundos

n:  10000
```

```
Tiempo RadixSort normal: 0.039011 segundos
n:  10000

Tiempo RadixSort MEJOR: 0.180043 segundos
n:  10000

Tiempo RadixSort PEOR: 0.170042 segundos
n:  10000

Tiempo CountingSort NORMAL: 0.011001 segundos

n:  10000

Tiempo CountingSort MEJOR: 0.014005 segundos

n:  10000

Tiempo CountingSort PEOR: 0.013001 segundos

n:  10000
```

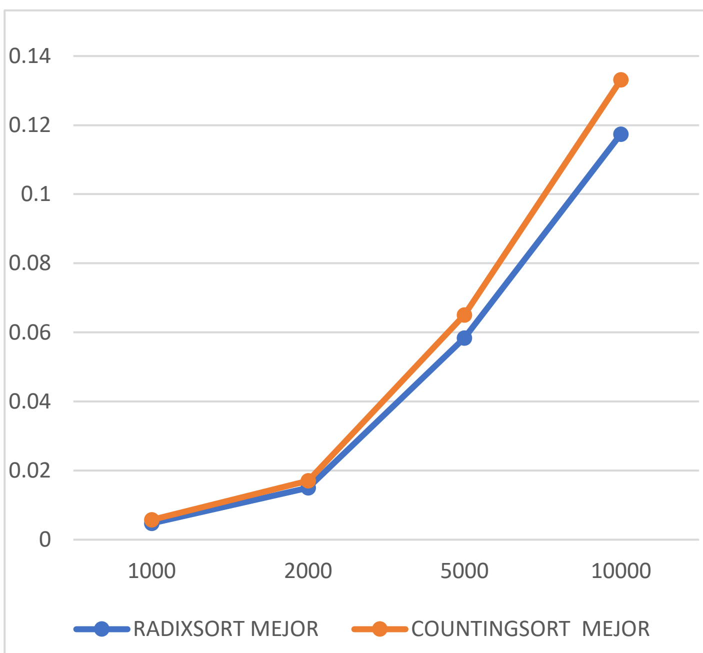
## CASO NORMAL




| n | P1 | P2 | P3 | PROMED |
|---|---|---|---|---|
| 1000 | 0.003002 | 0.002 | 0.001997 | 0.002333 |
| 2000 | 0.005001 | 0.005885 | 0.006003 | 0.0056297 |
| 5000 | 0.013134 | 0.015007 | 0.016102 | 0.0147477 |
| 10000 | 0.024134 | 0.028005 | 0.039011 | 0.0303833 |
| IXSORT NORMAL | | | | |

| n | P1 | P2 | P3 | PROMED |
|---|---|---|---|---|
| 1000 | 0 | 0 | 0.001003 | 0.0003343 |
| 2000 | 0.000999 | 0.000946 | 0.002027 | 0.001324 |
| 5000 | 0.002997 | 0.002998 | 0.003004 | 0.0029997 |
| 10000 | 0.007003 | 0.037019 | 0.011001 | 0.018341 |
| COUNTINGSORT NORMAL | | | | |

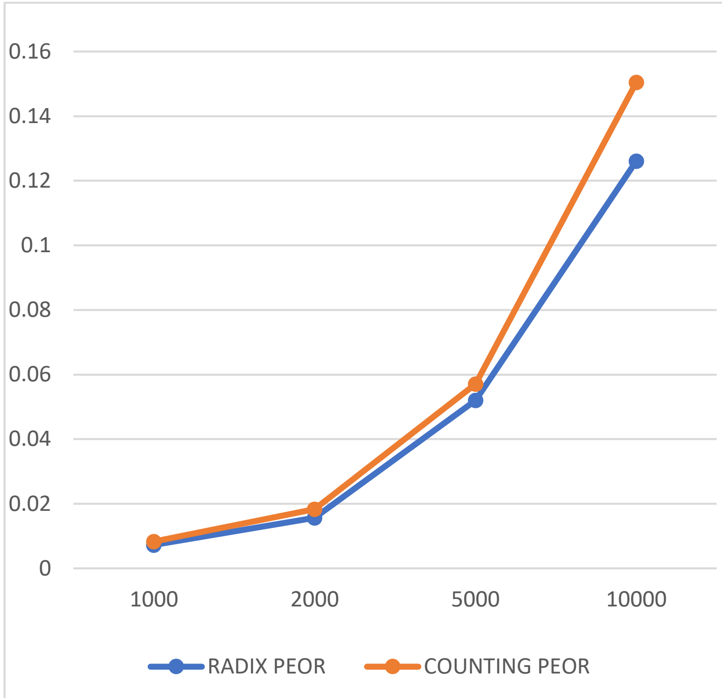
## MEJOR CASO




| n | P1 | P2 | P3 | PROMED |
|---|---|---|---|---|
| 1000 | 0.005062 | 0.005006 | 0.004128 | 0.004732 |
| 2000 | 0.015129 | 0.015003 | 0.015006 | 0.015046 |
| 5000 | 0.092022 | 0.04201 | 0.041004 | 0.0583453 |
| 10000 | 0.078143 | 0.094022 | 0.180043 | 0.1174027 |
| RADIXSORT MEJOR | | | | |

| n | P1 | P2 | P3 | PROMED |
|---|---|---|---|---|
| 1000 | 0.001001 | 0.000987 | 0.000995 | 0.0009943 |
| 2000 | 0.002002 | 0.002099 | 0.001973 | 0.0020247 |
| 5000 | 0.005003 | 0.007008 | 0.008001 | 0.0066707 |
| 10000 | 0.012002 | 0.021003 | 0.014005 | 0.01567 |
| COUNTINGSORT MEJOR | | | | |

PEOR CASO




| n | P1 | P2 | P3 | PROMED |
|---|---|---|---|---|
| 1000 | 0.007927 | 0.008005 | 0.005875 | 0.007269 |
| 2000 | 0.015005 | 0.016062 | 0.016005 | 0.0156907 |
| 5000 | 0.045007 | 0.047008 | 0.06401 | 0.0520083 |
| 10000 | 0.090149 | 0.118028 | 0.170042 | 0.126073 |
| RADIX PEOR | | | | |

| n | P1 | P2 | P3 | PROMED |
|---|---|---|---|---|
| 1000 | 0.001 | 0.001115 | 0.001017 | 0.001044 |
| 2000 | 0.002993 | 0.002875 | 0.001998 | 0.002622 |
| 5000 | 0.007 | 0.003998 | 0.004128 | 0.005042 |
| 10000 | 0.011031 | 0.049012 | 0.013001 | 0.024348 |
| COUNTING PEOR | | | | |

## CONCLUSIONES:

El algoritmo posee una serie de limitaciones que obliga a que solo pueda ser utilizado en determinadas circunstancias, la ventaja que podemos encontrar de Randix Sort es que estos datos que tiene cualquier arreglo serán medidos, comparados y ordenados por procedimientos que se establecen dentro del mismo, que además usa de apoyo CountingSort2 para generar este ordenamiento que se requiere para determinar si un numero es mayor que otro.

Para Radix sort que usa counting sort como una ordenación estable intermedia, la complejidad de tiempo es

O (d (n + k)).

Aquí, d es el ciclo numérico y O (n + k) es la complejidad temporal de la clasificación de conteo.

El algoritmo Counting Sort funciona mejor con una lista larga, de un solo elemento simple. La desventaja de este algoritmo es la necesidad de almacenar muchos datos en memoria.

Complejidad general = O (máx.) + O (tamaño) + O (máx.) + O (tamaño) = O (máx. + Tamaño)
Peor complejidad del caso: O (n + k)
Mejor complejidad de caso: O (n + k)
Complejidad de casos promedio: O (n + k)

Podemos encontrar como en mi práctica, se maneja en que, para ambos algoritmos de acomodamiento, el mejor caso siempre fue el que tenia ya su lista ordenada en el mejor de los casos. Esto es debido a que al tener un arreglo ya ordenado, no requerirá de ningún proceso de ordenamiento, este ya esta previo y por eso su ejecución es muy rápida. También es posible apreciar como en un manejo de pocos datos, RadixSort funciona mejor que countingsort, mientras que los datos aumentaban era el caso contrario, counting sort era mejor y podía acomodar una gran cantidad de números.