



Carátula para entrega de prácticas

Facultad de Ingeniería

Laboratorio de docencia

Laboratorios de computación salas A y B

Profesor:

Jesús Cruz Navarro

Asignatura:

Estructuras de Datos y Algoritmos 2

Grupo:

1

No de Práctica(s):

3

Integrante(s):

Ugalde Velasco Armando

*No. de Equipo de
cómputo empleado:*

No. de Lista o Brigada:

32

Semestre:

2021-1

Fecha de entrega:

24 de octubre de 2020

Observaciones:

CALIFICACIÓN: _____

PRÁCTICA 3: ORDENAMIENTO 3

Objetivo: El estudiante conocerá e identificará la estructura de los algoritmos de ordenamiento *CountingSort* y *RadixSort*

1. Programar los algoritmos de ordenamiento vistos en clase. Cada algoritmo debe estar en una función y recibir solamente un arreglo de datos.

a. *CountingSort*

```
def countingSort(arr):  
    k = max(arr)  
    n = len(arr)  
  
    idxArr = [0] * (k + 1)  
    final = [0] * n  
  
    for num in arr:  
        idxArr[num] += 1  
  
    for i in range(1, k + 1):  
        idxArr[i] += idxArr[i - 1]  
  
    for j in range(n - 1, -1, -1):  
        toAdd = arr[j]  
        idxArr[toAdd] -= 1  
        final[idxArr[toAdd]] = toAdd  
  
    return final
```

Implementación de CountingSort

Se implementó el algoritmo analizado en clase. Primero, se determinó **k**, es decir, el valor del elemento mayor presente en la lista, y se definió la variable **n**, que contendría el tamaño de la lista. Posteriormente, se crearon dos arreglos: **idxArr**, que contendría los índices donde se colocarían los elementos ordenados, y **final**, que contendría los elementos ordenados.

Después, se colocó el número de ocurrencias de cada elemento en el arreglo **idxArr** y se calcularon los índices finales. Finalmente, se colocaron los elementos ordenados en sus respectivas posiciones en la lista **final**.

Como ya se mencionó en el análisis previo del algoritmo, la complejidad asintótica de este algoritmo es $O(n + k)$

b. RadixSort

```
def radixSort(arr):
    k = max(arr)
    d = int(math.log10(k) + 1)

    for i in range(d):
        arr = countingSortPosicion(arr, i)

    return arr

def countingSortPosicion(arr, posicion):
    n = len(arr)
    k = 9

    def obtenerDigito(numero):
        return (numero // 10 ** posicion) % 10

    idxArr = [0] * (k + 1)
    final = [0] * n

    for num in arr:
        idxArr[obtenerDigito(num)] += 1

    for i in range(1, k + 1):
        idxArr[i] += idxArr[i - 1]

    for j in range(n - 1, -1, -1):
        toAdd = arr[j]
        digito = obtenerDigito(toAdd)
        idxArr[digito] -= 1
        final[idxArr[digito]] = toAdd

    return final
```

Implementación de RadixSort

La función **countingSortPosicion** ordena el arreglo respecto a cierto dígito, utilizando el algoritmo **countingSort**. El funcionamiento del algoritmo **radixSort** consiste en ordenar el arreglo respecto a cada dígito de sus elementos, iniciando con el de menor valor. De esta forma, al finalizar el ordenamiento respecto a cada dígito, se logra ordenar el arreglo de forma exitosa, gracias a la estabilidad del algoritmo **countingSort**.

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 ($n = 500, 1000, 2000, 5000$) y registrar los tiempos de ejecución para cada uno de los siguientes casos:

- a. *Mejor Caso*
- b. *Peor Caso*
- c. *Caso promedio*

Para ambos algoritmos, las características de ordenamiento de las entradas no cambian su rendimiento, es decir, en el caso de **countingSort** su complejidad es $O(n + k)$ para cualquier entrada, y en el caso de **radixSort**, es $O(dn)$. Podríamos decir que, en el caso de **countingSort**, si el parámetro k aumenta considerablemente, el tiempo de ejecución aumentaría. Sin embargo, recordemos que el rendimiento del algoritmo ya se encuentra en función de esta variable, por lo tanto, este tipo de situaciones ya se considera en el análisis del rendimiento. Lo mismo ocurre con **radixSort**: el número de dígitos es una variable independiente; el rendimiento se encuentra en función de ésta y de la cantidad de números a ordenar.

3. Dado que los tiempos pueden variar de una ejecución a otra, realice las pruebas 3 veces y obtenga el promedio de los tiempos. Nota 1: Para generar lista ordenada ascendentemente, puede agregue el valor de i del ciclo `for` a la lista. Para generar lista ordenada descendentemente, puede usar `for i in range(max, min, - 1)`, lo hará que i tome los valores de max hasta min (exclusivo) en cada iteración. Nota 2: Si hay problemas con el límite de recursión que usa por default Python, modifíquelo para permitir los peores casos de QuickSort.

El programa realizado consta de distintas funciones auxiliares, las cuales se mostrarán a continuación.

Primero, se definieron algunas constantes importantes: los algoritmos a utilizar, sus nombres, y los valores mínimo y máximo de los elementos en los arreglos a ordenar.

Además, se definió una lista con los valores de **n** a analizar:

```
ALG1 = countingSort.countingSort
ALG2 = radixSort.radixSort
ALG1_NAME = "CountingSort"
ALG2_NAME = "RadixSort"

# Valores mínimo y máximo de los elementos
MIN_VALUE = 0
MAX_VALUE = 100000

# Valores de n a analizar
X_ARR = [1000, 2000, 5000, 10000]
```

Constantes a utilizar en el programa

sortAndComputeTime

Esta función ordena una lista con el algoritmo proporcionado (**función sortingAlg**), y calcula el tiempo utilizado por éste.

```
def sortAndComputeTime(sortingAlg, arr):
    # Mantener una copia ordenada
    sortedArr = arr.copy()
    sortedArr.sort()

    # Realizar una copia para evitar mutaciones en el arreglo original
    copy = arr.copy()

    # Ordenar la lista y calcular su tiempo de ejecución
    t1 = time.time()
    sortingAlg(copy)
    t2 = time.time()

    # Comprobar que el algoritmo ordenó correctamente la lista
    assert copy == sortedArr

    # Devolver el tiempo utilizado
    return t2 - t1
```

Función sortAndComputeTime

sortAndGetTimes

Esta función ordena la misma lista con los dos algoritmos de ordenamiento analizados y retorna una tupla con los tiempos utilizados por cada uno.

```
def sortAndGetTimes(arr):
    print("N =", len(arr))

    firstAlgTime = sortAndComputeTime(ALG1, arr)
    print(ALG1_NAME, ":", firstAlgTime)

    secondAlgTime = sortAndComputeTime(ALG2, arr)
    print(ALG2_NAME, ":", secondAlgTime, "\n")

    return firstAlgTime, secondAlgTime
```

Función sortAndGetTimes

computeAvgTimesArr

Esta función retorna una lista de tuplas con los tiempos promedio de ordenamiento para cada n , utilizando el generador de listas proporcionado. Éste último es el que determina el tipo de caso presentado (mejor, peor o promedio).

```
def computeAvgTimesArr(listGenerator):
    # Lista con las tuplas de tiempos promedios para cada n.
    AVG_TIMES = []
    # Calcular los tiempos promedio para cada n y agregar la tupla respectiva a la lista
    for n in X_ARR:
        # Lista que contendrá 3 tuplas con los tiempos obtenidos para n
        TIMES_FOR_N = []
        # Calcular 3 veces el tiempo de los algoritmos, ordenando
        # listas diferentes en cada iteración (utilizando el generador dado)
        for i in range(3):
            arr = listGenerator(n)
            TIMES_FOR_N.append(sortAndGetTimes(arr))

        # Calcular el promedio de tiempo para cada algoritmo
        firstMean = numpy.mean([t[0] for t in TIMES_FOR_N])
        print("Tiempo promedio", ALG1_NAME, "para", n, "elementos:", firstMean)
        secondMean = numpy.mean([t[1] for t in TIMES_FOR_N])
        print("Tiempo promedio", ALG2_NAME, "para", n, "elementos:", secondMean,
              "\n")
        # Agregar la tupla de tiempo promedio obtenida
        AVG_TIMES.append((firstMean, secondMean))
    return AVG_TIMES
```

Función computeAvgTimesArr

testAndPlot

Esta función obtiene los tiempos promedios para cada **n** utilizando el generador de listas proporcionado, e imprime la gráfica con la información necesaria utilizando la librería *matplotlib*. De nuevo, **listGenerator** determina el caso a analizar, y el parámetro **title** debe contener el título del caso a mostrar en la gráfica.

```
def testAndPlot(listGenerator, title):
    print(title, "\n")
    AVG_TIMES = computeAvgTimesArr(listGenerator)

    fig, ax = plt.subplots()
    ax.set_title(title)
    ax.set_xlabel("Tamaño del arreglo")
    ax.set_ylabel("Tiempo de ejecución")
    ax.plot(X_ARR, [y[0] for y in AVG_TIMES])
    ax.plot(X_ARR, [y[1] for y in AVG_TIMES])
    ax.legend([ALG1_NAME, ALG2_NAME])
```

Función testAndPlot

Función para ejecutar las pruebas del caso promedio

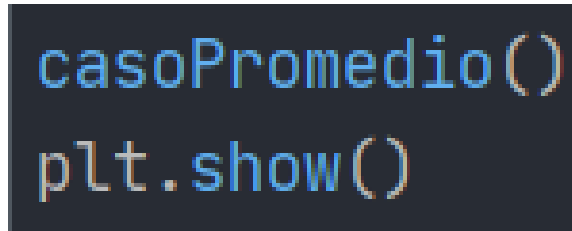
Esta función ejecuta los tests correspondientes de rendimiento para cada algoritmo, y grafican sus respectivos resultados. Nótese que, dentro de la función, se encuentra definida la función generadora de listas para el caso promedio, es decir, retorna una lista aleatoria.

```
def casoPromedio():
    # Devuelve una lista aleatoria de tamaño n
    def listGenerator(n):
        return [random.randint(MIN_VALUE, MAX_VALUE) for j in range(n)]

    testAndPlot(listGenerator, "CASO PROMEDIO")
```

Test para el caso promedio

Finalmente, se ejecuta las función anterior y se muestra la gráfica generada:



```
casoPromedio()  
plt.show()
```

Ejecución de los tests y graficación

A continuación, se muestran algunas capturas de pantalla de la salida producida por la ejecución del programa:

```
N = 1000  
CountingSort : 0.025907278060913086  
RadixSort : 0.00880575180053711  
  
N = 1000  
CountingSort : 0.019605636596679688  
RadixSort : 0.004986286163330078  
  
N = 1000  
CountingSort : 0.018901348114013672  
RadixSort : 0.005828857421875  
  
Tiempo promedio CountingSort para 1000 elementos: 0.021471420923868816  
Tiempo promedio RadixSort para 1000 elementos: 0.0065402984619140625  
  
N = 2000  
CountingSort : 0.019949674606323242  
RadixSort : 0.010967493057250977  
  
N = 2000  
CountingSort : 0.018947601318359375  
RadixSort : 0.010763883590698242  
  
N = 2000  
CountingSort : 0.019855976104736328  
RadixSort : 0.01002192497253418  
  
Tiempo promedio CountingSort para 2000 elementos: 0.01958441734313965  
Tiempo promedio RadixSort para 2000 elementos: 0.010584433873494467
```



```
N = 5000
CountingSort : 0.025203704833984375
RadixSort : 0.030272245407104492

N = 5000
CountingSort : 0.02837657928466797
RadixSort : 0.03291010856628418

N = 5000
CountingSort : 0.024361371994018555
RadixSort : 0.028783321380615234

Tiempo promedio CountingSort para 5000 elementos: 0.025980552037556965
Tiempo promedio RadixSort para 5000 elementos: 0.0306552251180013

N = 10000
CountingSort : 0.02796316146850586
RadixSort : 0.05636477470397949

N = 10000
CountingSort : 0.025053977966308594
RadixSort : 0.05099344253540039

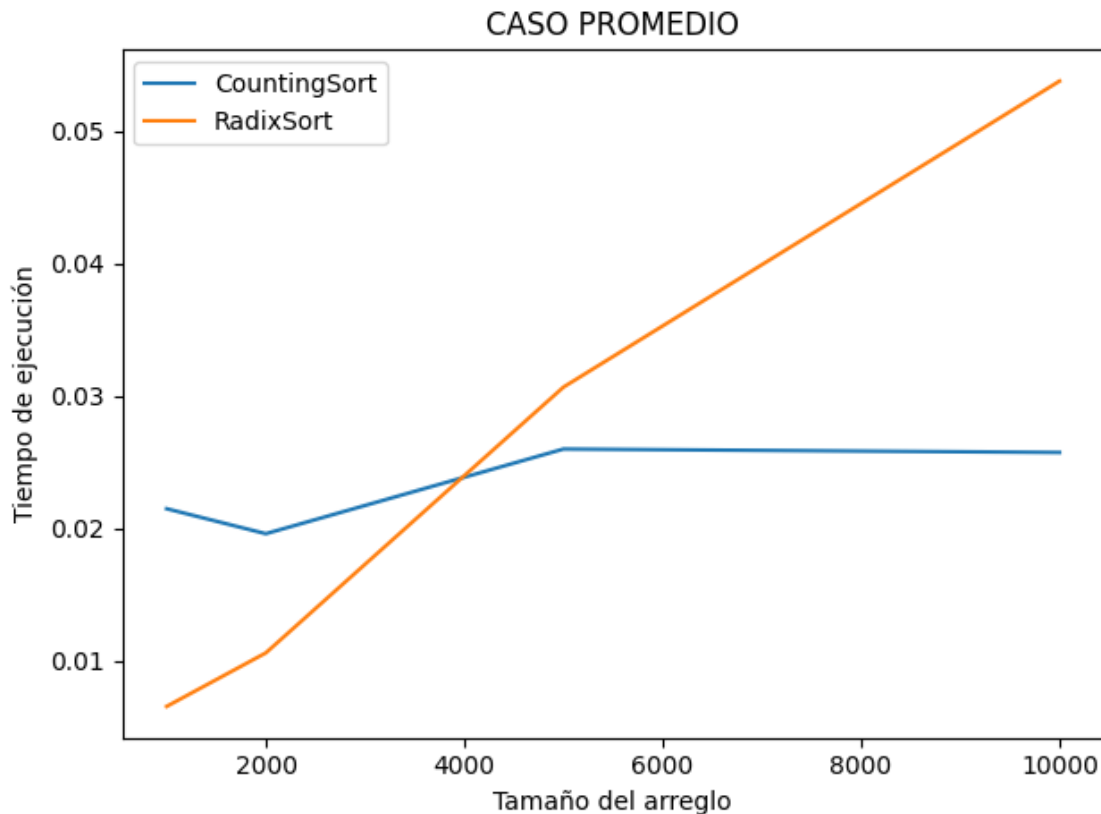
N = 10000
CountingSort : 0.024146318435668945
RadixSort : 0.05394625663757324

Tiempo promedio CountingSort para 10000 elementos: 0.025721152623494465
Tiempo promedio RadixSort para 10000 elementos: 0.053768157958984375
```

Salida del programa

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).

A continuación, se muestran las gráficas generadas por el programa anterior:



Gráfica de rendimiento

Como se logra apreciar, las curvas correspondientes al rendimiento de ambos algoritmos tienen una forma parecida a la de una función lineal. Es importante mencionar que, el rango de los valores a ordenar fue de **0** a **100000**. Dicho lo anterior, en la gráfica se observa que los tiempos de **countingSort** son, en un inicio, mayores a los de **radixSort**. Podemos inferir que la razón del comportamiento anterior es el hecho de que **k** es un número muy grande, por lo tanto, en un inicio supera por mucho a **n**, lo cual se ve reflejado en un tiempo elevado.

Sin embargo, se observa que, aproximadamente cuando **n = 4000**, las curvas se intersecan, significando que, a partir de ese punto, el rendimiento de **countingSort** es mejor respecto al de **radixSort**. Lo anterior probablemente se deba a distintos factores, como la obtención de los dígitos en **radixSort**, operaciones extras realizadas por utilización de funciones, entre otros.

En otras palabras, se realizó un análisis considerando a **k** constante.

Para analizar la variación de **k**, manteniendo **n** constante, se realizaron las siguientes funciones:

```

def testAndPlotK(listGenerator, title):
    print(title, "\n")
    AVG_TIMES = computeAvgTimesArr(listGenerator)

    fig, ax = plt.subplots()
    ax.set_title(title)
    ax.set_xlabel("Valor Máximo (k)")
    ax.set_ylabel("Tiempo de ejecución")
    ax.plot(X_ARR, [y[0] for y in AVG_TIMES])
    ax.plot(X_ARR, [y[1] for y in AVG_TIMES])
    ax.legend([ALG1_NAME, ALG2_NAME])

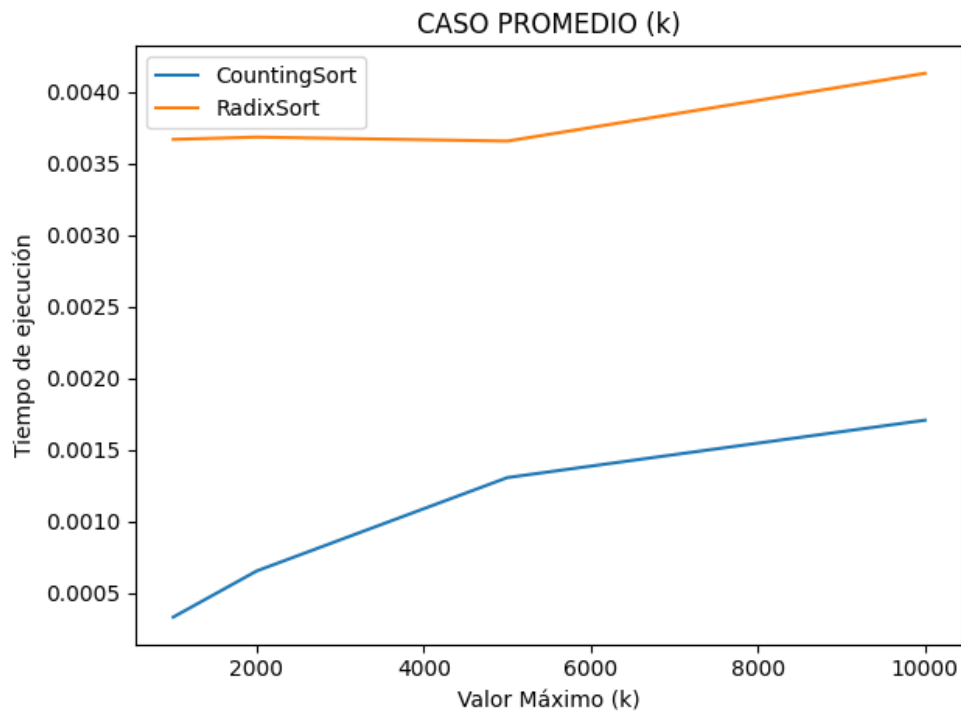
def casoPromedioK():
    # Devuelve una lista aleatoria de tamaño n
    def listGenerator(k):
        return [random.randint(0, k) for j in range(1000)]

    testAndPlotK(listGenerator, "CASO PROMEDIO (k)")

```

Funciones para analizar k

Como se logra observar, se cambió el generador para que retornara una lista de **1000** elementos de **0** a **k**, teniendo a ésta última cantidad como el parámetro. La gráfica obtenida es la siguiente:



Gráfica de rendimiento respecto a k

En este caso, la variable independiente es **k**: el valor máximo en la lista a ordenar. Como se logra observar, ambas funciones tuvieron un comportamiento similar al de una función lineal, concordando con la hipótesis realizada. Claramente, los tiempos de ejecución de **radixSort** superaron de nueva cuenta a los de **countingSort**.

CONCLUSIONES

CountingSort y **RadixSort** son dos algoritmos de ordenamiento que, a diferencia de los algoritmos ya vistos, no están basados en comparación, por lo tanto, su límite de complejidad asintótica posible no es el mismo. Como se analizó en clase, la complejidad de estos algoritmos es lineal, lo cual representa una significativa mejora de los tiempos, en teoría.

Sin embargo, en la práctica comúnmente se utilizan los algoritmos **quickSort** y **mergeSort**, a pesar de que su complejidad asintótica es mayor. Lo anterior se debe a distintos factores: uno de los más relevantes es el hecho de que los algoritmos analizados en la práctica tienen restricciones sobre los elementos en el arreglo: estos no pueden ser negativos. Además, también es posible que los factores constantes sean mayores y, por ende, provoquen que, para ciertas entradas, el rendimiento sea malo.

Otra posible ventaja de estos algoritmos es el hecho de que podrían considerarse más simples o intuitivos a comparación de los basados en comparación.