Práctica 7: Algoritmos de ordenamiento

Reyes Valenzuela Alejandro, Gutiérrez Povedano Pablo.

Escuela Superior de Cómputo
Instituto Politécnico Nacional, México
*areyesv*11@*gmail.com*, *gpovedanop*@*gmail.com*

Resumen: En este reporte se emplearán los conceptos vistos en clase para analizar tres algoritmos de ordenamiento no vistos en clase, siendo propuestos el Bucket-Sort, Radix Sort y Tree Sort, para calcular su complejidad tanto en el peor de los casos como en el mejor de los casos.

Palabras Clave: Bucket,Radix,Árbol,Binario.

1 Introducción

Hasta este momento, se han analizado algoritmos que únicamente se encargan de trabajar con los mismos arreglos que se ordenarán, sin auxiliarse de otras estructuras o de algún procedimiento donde cada uno de los valores del mismo arreglo son truncados o separados para ordenarlos posteriormente, por lo que en esta práctica se analizarán tres algoritmos que se auxilian de otras estructuras (Árboles binarios y Casilleros para Tree y Bucket respectivamente) y otro donde ordenamos enteros procesando sus dígitos de forma individual (Radix Sort).

Estos tipos de algoritmos tienden a tener complejidades logarítmicas y lineales, por lo que analizaremos los tres algoritmos para comprobar esto.

2 Conceptos Básicos

Radix Sort: Este ordenamiento se basa en los valores de los dígitos reales en las representaciones de posiciones de los nmeros que se ordenan. Por ejemplo el nmero 235 se escribe 2 en la posición de centenas, un 3 en la posición de decenas y un 5 en la posición de unidades.

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. Se auxilia de un algoritmo que se encarga de contar, cuya complejidad es $O(n + k)$.

```
RadixSort(A, x)
    for j = 1 to x do
        int count[10] = 0;
        for i = 0 to n do
            count[key of(A[i]) in pass j] ++
        for k = 1 to 10 do
            count[k] = count[k] + count[k − 1]
        for i = n-1 downto 0 do
            resultado[count[key of(A[i])]] = A[j]
            count[key of(A[i])] − −
        for i = 0 to n do
            A[i] = resultado[i]
```

Por ejemplo:

Dado un arreglo de enteros:

503, 12, 45, 3411, 304, 9

Nos fijaremos en la parte de las unidades, en éste caso, el mayor es el 9 y el menor es el uno, por lo que ordenamos con base a el dígito menos significativo:

3411, 12, 503, 304, 45, 9

Ahora ordenamos con base a centenas, en caso de los dígitos que no tengan cifra en esta parte se les asigna cero:

9, 12, 45, 304, 503, 3411

Podemos observar que el arreglo se encuentra ordenado

Bucket Sort: Distribuye todos los elementos a ordenar entre un número finito de casilleros. Cada casillero sólo puede contener los elementos que cumplan unas determinadas condiciones. Las condiciones deben ser excluyentes entre sí, para evitar que un elemento pueda ser clasificado en dos casilleros distintos. Después cada uno de esos casilleros se ordena individualmente con

otro algoritmo de ordenación (que podra ser distinto según el casillero), o se aplica recursivamente este algoritmo para obtener casilleros con menos elementos. Para el ordenamiento de los casilleros se hace uso de Insertion Sort (de complejidad $O(n^2)$).

BucketSort(A, n)
    casilleros = coleccion de n listas
    for i = 1 to A.length do
        c = buscar el casillero adecuado
        insertar elementos$[i]$ *en casillero*$[c]$
    para i = 1 hasta n hacer
        InsertionSort(*casilleros*$[i]$)
            return la concatenación de casilleros[1],..., casilleros[n]

Por ejemplo:

Dado un arreglo de flotantes:
0.897, 0.565, 0.656, 0.1234, 0.665, 0.3434
Como tenemos seis elementos creamos seis casilleros:

- c0 $->$ 0.897
- c1 $->$ 0.565
- c2 $->$ 0.656
- c3 $->$ 0.1234
- c4 $->$ 0.665
- c5 $->$ 0.3434

Ahora ordenaremos los datos en subcasilleros, para ello debemos de discriminar para evitar que un elemento pueda pertenecer a más de un subcasillero, por ejemplo, para este caso el casillero dos y el cuatro pueden ser organizados en uno solo, puesto que su ́digito más significativo es el mismo, para ello partimos desde el mayor y desde allí asignamos los subcasilleros de manera descendente:

- c0 $->$ 0.897
- c1 $->$ 0.565
- Sigue existiendo este casillero, pero por motivos ilustrativos se omite
- c3 $->$ 0.1234
- c4 $->$ 0.665 $->$ 0.656

- c5 $->$ 0.3434

Finalmente, reacomodamos los casilleros mediante InsertionSort, por lo que el arreglo ordenado queda de la siguiente manera:

0.565, 0.656, 0.665, 0.897, 0.1234, 0.3434

Tree Sort: Ordena sus elementos haciendo uso de un árbol binario de búsqueda. Se basa en ir construyendo poco a poco el árbol binario introduciendo cada uno de los elementos, los cuales quedarán ya ordenados. Despus, se obtiene la lista de los elementos ordenados recorriendo el árbol en inorden. Para ello nos auxiliamos de dos funciones externas, la de inserción y la de recorrimiento

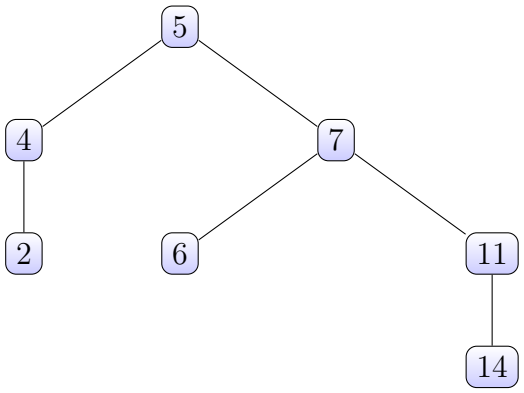
```
TreeSort(A)

Insertar(Arbol, elemento n)
   Si nodo está vacío
      Nodo -> n
   De otra forma
      Si n ≤ Nodo
         Insertar(SubÁrbolIzquierdo,n)
      De otra forma
         Insertar(Sub"ArbolDerecho,n)

Recorrer(Arbol)
   Si nodo está vacío
      break;
   De otra forma
      Recorrer(SubÁrbolIzquierdo)
      Recorrer(SubÁrbolDerecho)
```

Por ejemplo, dado el siguiente arreglo:

5, 4, 7, 2, 11, 14, 6

Creamos un árbol de búsqueda binaria empezando con el primer elemento, si el siguiente elemento es menor se agrega un nodo a la izquierda, si es mayor o igual a la derecha:

Árbol de búsqueda binaria ordenado.



Finalmente realizamos el recorrido en inorden transversal para obtener el arreglo ordenado:

2, 4, 5, 6, 7, 11, 14

3 Experimentación y Resultados

Radix Sort

```
[alejandro@alejandro-pc ~]$ python2 radix.py
Arreglo Original
[170, 45, 75, 90, 802, 24, 2, 66]
Arreglo Ordenado
2 24 45 66 75 90 170 802
[alejandro@alejandro-pc ~]$
```

Bucket Sort

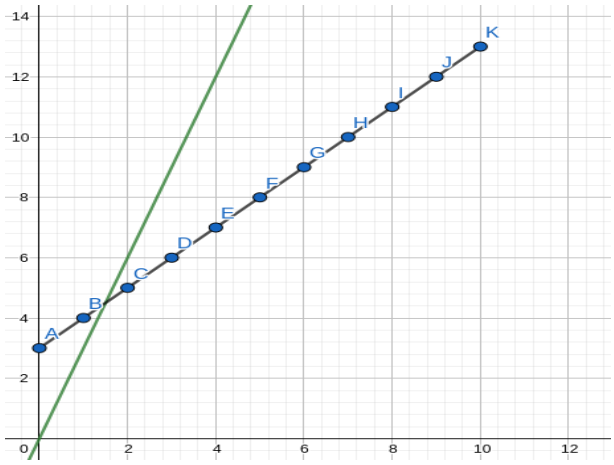
```
[alejandro@alejandro-pc ~]$ python3 bucket.py
Tamaño Arreglo: 16
Arreglo Original
[1, 3, 2, 4, 1, 1, 1, 2, 1, 1, 5, 2, 1, 5, 4, 1]
Arreglo Ordenado
[1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 3, 4, 4, 5, 5]
[alejandro@alejandro-pc ~]$
```

Tree Sort

```
run:
---ARREGLO ORIGINAL---
5 4 7 2 11 1 24 112 3
---ARREGLO ORDENADO---
1 2 3 4 5 7 11 24 112
BUILD SUCCESSFUL (total time: 0 seconds)
```

Para Radix Sort, su complejidad se basa en la cantidad máxima de dígitos que contenga el arreglo, (por ejemplo, si el mayor número del arreglo es 1000, su complejidad se basará en la potencia de este número, en este caso 10^3. Para esta gráfica tomamos en cuenta como n a la potencia de 10 máxima del número.

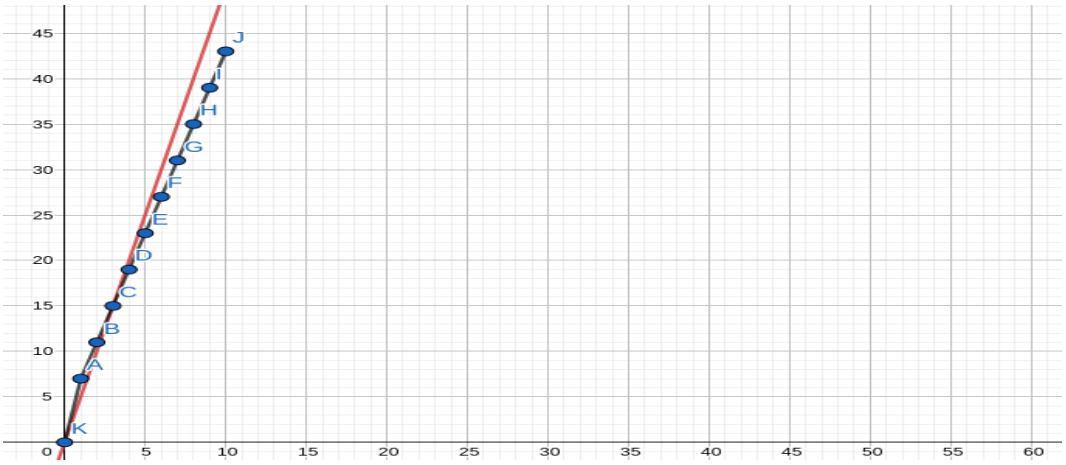
Gráfica de la función Radix Sort acotada con $f(n) = 3n$.



Podemos ver que su complejidad es $\Theta(n)$, ya que la potencia representa el número de veces que se realizará el ordenamiento.
Valores de función RadixSort

Potencia	**Valor**
0	3
1	4
2	5
3	6
4	7
5	8
6	9
7	10
8	11
9	12
10	13

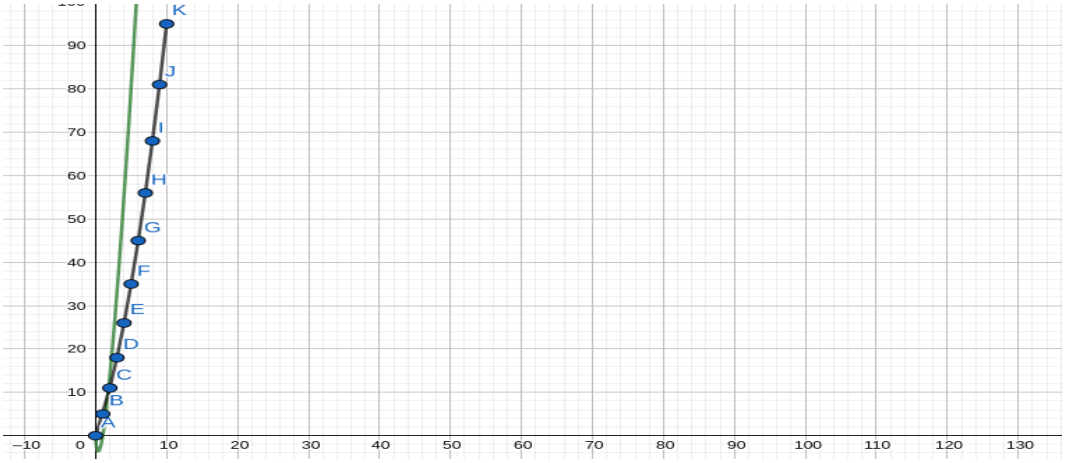
Para Bucket Sort sucede algo similar, la complejidad está dada en las veces que se tiene que ordenar por casilleros, por lo que su complejidad también es lineal tanto para el mejor como en el peor de los casos. Para ello se decidió acotar con la función $f(n) = 5n$. n representa el número de elementos del arreglo.



Tamano	**Valor**
0	0
1	7
2	11
3	15
4	19
5	23
6	27
7	31
8	35
9	39
10	42

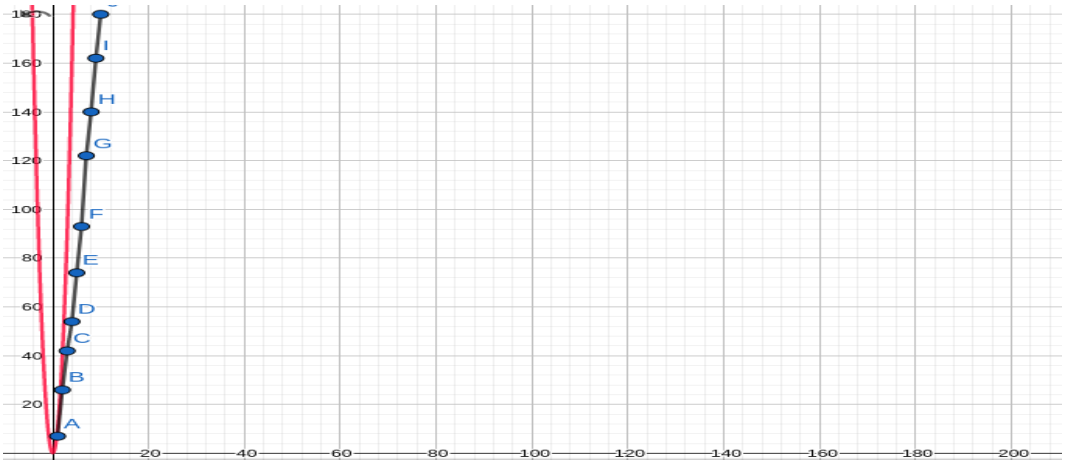
Finalmente, el recorrimiento del árbol tendrá complejidad logarítmica en el mejor de los casos, mientras que en peor será cuadrático, debido a que se tienen diversas ramificaciones en donde se realizará el recorrido.

Mejor Caso (acotando con 10n*logn)



Tamano	**Valor**
1	5
2	11
3	18
4	26
5	35
6	45
7	56
8	68
9	81
10	95

Peor Caso (acotando con $10n^2$)



Tamano	**Valor**
1	7
2	26
3	42
4	54
5	74
6	93
7	122
8	140
9	162
10	180

4 Conclusiones

Conclusión General: Pudimos observar que ciertos algoritmos no dependen enteramente del ordenamiento que tengan, si no de la metodología que usen para realizar el propio ordenamento, por lo que debimos de analizar otros factores para poder determinar su complejidad tanto teórica como práctica.

Conclusión Gutiérrez Povedano: Existen diversos tipos de algoritmos de ordenamiento, cada uno con sus ventajas, desventajas y complejidad, pero no se puede designar a un algoritmo de ordenamiento como el ms eficiente de todos o el de menor complejidad ya que cada uno est diseado para operar en ciertos problemas, como el algoritmo radix que a pesar de tener una complejidad (k*n) es diferente a los dems ya que permite también ordenar cadenas de caracteres utilizando los valores numricos que cada carácter representa en la cadena.

Conclusión Reyes Valenzuela: Esta práctica me permitió trabajar con diversas estrucuras que me facilitaron el manejo de números y que a su vez ayudan a reducir la complejidad de manera considerable, sin embargo, tuvimos que analizar ambos casos, y cuándo podrían suscitarse, por lo que podrían parecer eficaces en teoría, pero no en la práctica.

5 Bibliografía

Geeks for geeks (2018). [online] Available at: https://www.geeksforgeeks.org/tree-sort/ [Accessed 15 Oct. 2018].

Geeks for geeks (2018). [online] Available at: https://www.geeksforgeeks.org/bucket-sort-2/ [Accessed 15 Oct. 2018].

Geeks for geeks (2018). [online] Available at: https://www.geeksforgeeks.org/radix-sort/ [Accessed 15 Oct. 2018].