# INSTITUTO POLITECNICO NACIONAL

*Escuela Superior de Cómputo*

# Análisis De Algoritmos

## Práctica 9

–>***Programación Dinámica:*** *Heap Sort, Counting Sort and Radix Sort.* <–

**Emiliano Gonzalez Hinojosa**

*metalica_el_01@hotmail.com*

*Semestre 2018*

**Grupo 3CV2**

Parte I

# Contenido

# Introducción

## Finalidad y Objetivo

Éste reporte tiene la finalidad de mostrar y dar a conocer la importancia de funcionamiento de los algoritmos que comúnmente el alumno desarrolla en clase.
Así mismo nos da algunos conceptos que nos ayudaran a comprender como es que la máquina (el compilador) cumple su funcionamiento y, nos muestra el costo computacional que se tiene al desarrollar ciertas prácticas en la vida cotidiana.

# Conceptos Básicos

## Counting Sort

El ordenamiento por cuentas (counting sort en inglés) es un algoritmo de ordenamiento en el que se cuenta el número de elementos de cada clase para luego ordenarlos. Sólo 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).

El primer paso consiste en averiguar cuál es el intervalo dentro del que están los datos a ordenar (valores mínimo y máximo). Después se crea un vector de números enteros con tantos elementos como valores haya en el intervalo [mínimo, máximo], y a cada elemento se le da el valor 0 (0 apariciones). Tras esto se recorren todos los elementos a ordenar y se cuenta el número de apariciones de cada elemento (usando el vector que hemos creado). Por último, basta con recorrer este vector para tener todos los elementos ordenados.

## Limitaciones

Sólo ordena números enteros, no vale para ordenar cadenas y es desaconsejable para ordenar números decimales. Teóricamente se puede, pero debería recrear en la matriz auxiliar tantas posiciones como decimales quepan entre 2 números consecutivos, si se restringe a 1 o 2 decimales podría ser asequible un número mayor de decimales puede llegar a suponer una memoria auxiliar impracticable.
Otra limitación (por ineficiencia) incluso con números enteros es cuando el rango entre el mayor y el menor es muy grande.

---

**Counting sort**

***Counting sort*** assumes that each of the $n$ input elements is an integer in the range 0 to $k$, for some integer $k$. When $k = O(n)$, the sort runs in $\Theta(n)$ time.

Counting sort determines, for each input element $x$, the number of elements less than $x$. It uses this information to place element $x$ directly into its position in the output array. For example, if 17 elements are less than $x$, then $x$ belongs in output position 18. We must modify this scheme slightly to handle the situation in which several elements have the same value, since we do not want to put them all in the same position.

In the code for counting sort, we assume that the input is an array $A[1..n]$, and thus $A.length = n$. We require two other arrays: the array $B[1..n]$ holds the sorted output, and the array $C[0..k]$ provides temporary working storage.

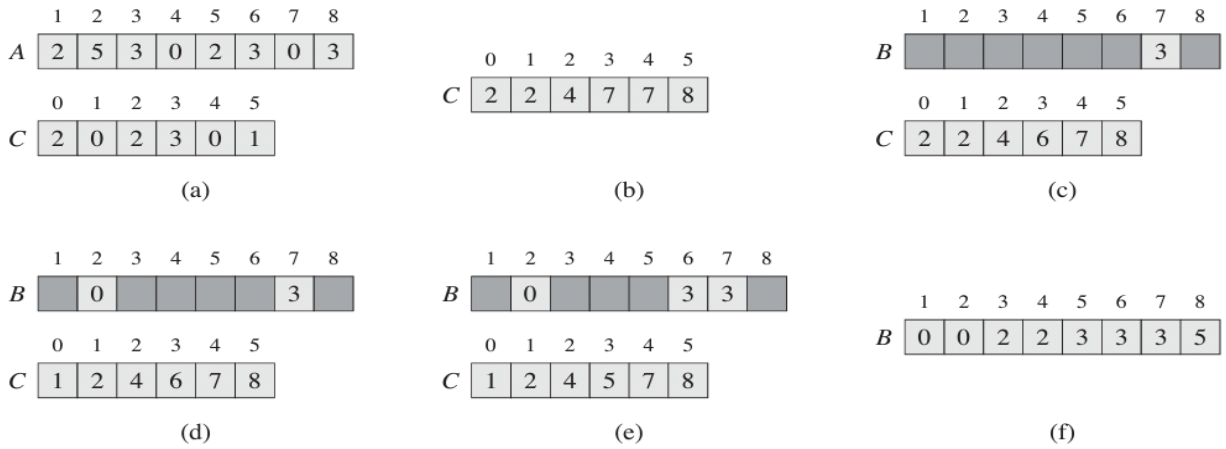


**Figure 8.2** The operation of COUNTING-SORT on an input array $A[1..8]$, where each element of $A$ is a nonnegative integer no larger than $k = 5$. **(a)** The array $A$ and the auxiliary array $C$ after line 5. **(b)** The array $C$ after line 8. **(c)–(e)** The output array $B$ and the auxiliary array $C$ after one, two, and three iterations of the loop in lines 10–12, respectively. Only the lightly shaded elements of array $B$ have been filled in. **(f)** The final sorted output array $B$.

```
COUNTING-SORT(A, B, k)
 1  let C[0..k] be a new array
 2  for i = 0 to k
 3      C[i] = 0
 4  for j = 1 to A.length
 5      C[A[j]] = C[A[j]] + 1
 6  // C[i] now contains the number of elements equal to i.
 7  for i = 1 to k
 8      C[i] = C[i] + C[i − 1]
 9  // C[i] now contains the number of elements less than or equal to i.
10  for j = A.length downto 1
11      B[C[A[j]]] = A[j]
12      C[A[j]] = C[A[j]] − 1
```

Figure 8.2 illustrates counting sort. After the **for** loop of lines 2–3 initializes the array $C$ to all zeros, the **for** loop of lines 4–5 inspects each input element. If the value of an input element is $i$, we increment $C[i]$. Thus, after line 5, $C[i]$ holds the number of input elements equal to $i$ for each integer $i = 0, 1, \ldots, k$. Lines 7–8 determine for each $i = 0, 1, \ldots, k$ how many input elements are less than or equal to $i$ by keeping a running sum of the array $C$.

## Heap Sort

Animación mostrando el funcionamiento del heapsort. El ordenamiento por montículos (heapsort en inglés) es un algoritmo de ordenamiento no recursivo, no estable.

Este algoritmo consiste en almacenar todos los elementos del vector a ordenar en un montículo (heap), y luego extraer el nodo que queda como nodo raíz del montículo (cima) en sucesivas iteraciones obteniendo el conjunto ordenado. Basa su funcionamiento en una propiedad de los montículos, por la cual, la cima contiene siempre el menor elemento (o el mayor, según se haya definido el montículo) de todos los almacenados en él. El algoritmo, después de cada extracción, recoloca en el nodo raíz o cima, la última hoja por la derecha del último nivel. Lo cual destruye la propiedad heap del árbol. Pero, a continuación realiza un proceso de "descenso"del número insertado de forma que se elige a cada movimiento el mayor de sus dos hijos, con el que se intercambia. Este intercambio, realizado sucesivamente "hunde.[el] nodo en el árbol restaurando la propiedad montículo del árbol y dejándo paso a la siguiente extracción del nodo raíz.

## Radix Sort

En informática, el ordenamiento Radix (radix sort en inglés) 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.
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.

---

## Radix sort

***Radix sort*** is the algorithm used by the card-sorting machines you now find only in computer museums. The cards have 80 columns, and in each column a machine can punch a hole in one of 12 places. The sorter can be mechanically "programmed" to examine a given column of each card in a deck and distribute the card into one of 12 bins depending on which place has been punched. An operator can then gather the cards bin by bin, so that cards with the first place punched are on top of cards with the second place punched, and so on.

For decimal digits, each column uses only 10 places. (The other two places are reserved for encoding nonnumeric characters.) A $d$-digit number would then occupy a field of $d$ columns. Since the card sorter can look at only one column at a time, the problem of sorting $n$ cards on a $d$-digit number requires a sorting algorithm.

Intuitively, you might sort numbers on their *most significant* digit, sort each of the resulting bins recursively, and then combine the decks in order. Unfortunately, since the cards in 9 of the 10 bins must be put aside to sort each of the bins, this procedure generates many intermediate piles of cards that you would have to keep track of. (See Exercise 8.3-5.)

Radix sort solves the problem of card sorting—counterintuitively—by sorting on the *least significant* digit first. The algorithm then combines the cards into a single deck, with the cards in the 0 bin preceding the cards in the 1 bin preceding the cards in the 2 bin, and so on. Then it sorts the entire deck again on the second-least significant digit and recombines the deck in a like manner. The process continues until the cards have been sorted on all $d$ digits. Remarkably, at that point the cards are fully sorted on the $d$-digit number. Thus, only $d$ passes through the deck are required to sort. Figure 8.3 shows how radix sort operates on a "deck" of seven 3-digit numbers.

In order for radix sort to work correctly, the digit sorts must be stable. The sort performed by a card sorter is stable, but the operator has to be wary about not changing the order of the cards as they come out of a bin, even though all the cards in a bin have the same digit in the chosen column.

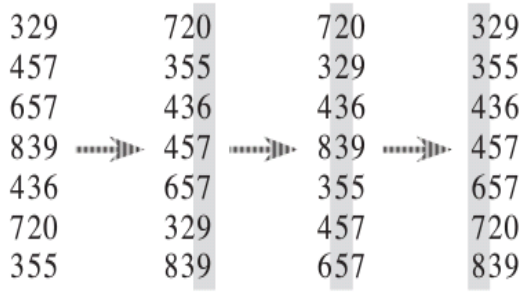


**Figure 8.3** The operation of radix sort on a list of seven 3-digit numbers. The leftmost column is the input. The remaining columns show the list after successive sorts on increasingly significant digit positions. Shading indicates the digit position sorted on to produce each list from the previous one.

In a typical computer, which is a sequential random-access machine, we sometimes use radix sort to sort records of information that are keyed by multiple fields. For example, we might wish to sort dates by three keys: year, month, and day. We could run a sorting algorithm with a comparison function that, given two dates, compares years, and if there is a tie, compares months, and if another tie occurs, compares days. Alternatively, we could sort the information three times with a stable sort: first on day, next on month, and finally on year.

The code for radix sort is straightforward. The following procedure assumes that each element in the $n$-element array $A$ has $d$ digits, where digit 1 is the lowest-order digit and digit $d$ is the highest-order digit.

```
RADIX-SORT(A, d)
1  for i = 1 to d
2      use a stable sort to sort array A on digit i
```

# Experimentación

**Para la parte de la experimentación los programas fueron desarrollados en JAVA y las clases fueron las siguientes: Practica8.java(La Principapal), CountingSort.java, HeapSort.java y RadixSort.java**

## Counting Sort

**A continuación se mostrará el código:**

CountingSort.java

```
1  package practica9;
2
3  import java.util.Random;
4  import java.util.Scanner;
5
6  public class CountingSort {
7      /*****************/
8      /* Counting Sort */
9      /*****************/
10
11     public static int BuscaMayorCountingSort(int A[]){
12         int mayor = 0;
13         for( int i=0; i<A.length ;i++){
14             if(A[i] > mayor){
15                 mayor = A[i];
16             }else{continue;}
17         }
18         return mayor;
19     }
20
21     public static void CountingSort(int A[], int B[], int k){
22         int C[] = new int[k+1];
23         for(int i =0; i<= k ; i++){
24             C[i] =0;
25         }
26         for(int j = 1; j< A.length; j++){
27             C[A[j]] = (C[A[j]]) +1;
28         }
29         for(int i = 1; i <= k ; i++){
30             C[i] = C[i] + C[i-1];
31         }
32         for(int j = (A.length)-1 ; j>=1 ; j--){
33             B[C[A[j]]] = A[j];
34             C[A[j]] = (C[A[j]])-1;
35         }
36
37         System.out.println("Arreglo Ordenado");
38         for(int l = 1; l<B.length; l++){
39             System.out.print(B[l]+" ");
40         }
41     }
42
43
44
45 }
```

## Heap Sort

**A continuación se mostrará el código:**

HeapSort.java

```java
package practica9;

public class HeapSort {

    public static void sort(int arr[]){
        int n = arr.length;

        for (int i = n / 2 - 1; i >= 0; i--)
            heapify(arr, n, i);

        for (int i=n-1; i>=0; i--){
            int temp = arr[0];
            arr[0] = arr[i];
            arr[i] = temp;

            heapify(arr, i, 0);
        }
    }

    public static void heapify(int arr[], int n, int i){
        int largest = i;  // Initialize largest as root
        int l = 2*i + 1;  // left = 2*i + 1
        int r = 2*i + 2;  // right = 2*i + 2

        if (l < n && arr[l] > arr[largest])
            largest = l;

        if (r < n && arr[r] > arr[largest])
            largest = r;

        if (largest != i){
            int swap = arr[i];
            arr[i] = arr[largest];
            arr[largest] = swap;

            heapify(arr, n, largest);
        }
    }
}
```

## Radix Sort

**A continuación se mostrará el código:**

RadixSort.java

```java
package practica9;

import java.util.Arrays;

public class RadixSort {
    public static int getMax(int arr[], int n){
        int mx = arr[0];
        for (int i = 1; i < n; i++)
            if (arr[i] > mx)
                mx = arr[i];
        return mx;
    }

    public static void countSort(int arr[], int n, int exp){
        int output[] = new int[n]; // output array
        int i;
        int count[] = new int[10];
        Arrays.fill(count,0);

        for (i = 0; i < n; i++)
            count[ (arr[i]/exp)%10 ]++;

        for (i = 1; i < 10; i++)
            count[i] += count[i - 1];

        for (i = n - 1; i >= 0; i--){
            output[count[ (arr[i]/exp)%10 ] - 1] = arr[i];
            count[ (arr[i]/exp)%10 ]--;
        }

        for (i = 0; i < n; i++)
            arr[i] = output[i];
    }

    static void radixsort(int arr[], int n){
        int m = getMax(arr, n);
        for (int exp = 1; m/exp > 0; exp *= 10)
            countSort(arr, n, exp);
    }
}
```

## Función Principal

### A continuación se mostrará el código:

Practica9.java

```
package practica9;

import java.util.Random;
import java.util.Scanner;
import static practica9.CountingSort.*;
import static practica9.RadixSort.*;
import static practica9.HeapSort.*;


public class Practica9{

    public static int[] GenerarArreglo(int tam){
        Random r = new Random();
        int A[] = new int[tam];
        for(int i=0; i<tam; i++){
            A[i] = r.nextInt(tam);
        }
        return A;
    }

    public static int[] GenerarArregloCS(int tam){
        Random r = new Random();
        int A[] = new int[tam+1];
        for(int i=0; i<=tam; i++){
            A[i] = r.nextInt(tam);
        }
        return A;
    }

    public static void printCS(int arr[], int n){
        for (int i=1; i<=n; i++)
            System.out.print(arr[i]+" ");
    }

    public static void print(int arr[], int n){
        for (int i=0; i<n; i++)
            System.out.print(arr[i]+" ");
    }

    public static void menu(){
        Scanner scan = new Scanner(System.in);
        int valor;
        int tam;
        do {
            System.out.println("<------------------------------>");
            System.out.println("<--- BIENVENIDO AL SISTEMA --->");
            System.out.println("<------------------------------>");
            System.out.println("<------Elija El Algoritmo----->");
            System.out.println("<------------------------------>");
            System.out.println("Counting sort----------------(1)");
            System.out.println("Radix sort-------------------(2)");
            System.out.println("Heap sort--------------------(3)");
            valor = scan.nextInt();
        } while (valor <1 || valor > 3);
        switch(valor){
            case 1:
                System.out.println("<---------------------->");
                System.out.println("<--- Counting Sort --->");
                System.out.println("<---------------------->");
                System.out.println("Tama o Del Arreglo: ");
                tam = scan.nextInt();
                int A[] = GenerarArregloCS(tam);
                int B[] = new int[tam+1];
                int mayor = BuscaMayorCountingSort(A);
                System.out.println("Arreglo Desordenado");
                printCS(A, tam);
                System.out.println("\nValor mas grande del arreglo es: "+mayor);
                CountingSort(A, B, mayor);
            break;
            case 2:
                System.out.println("<------------------>");
```

```
72                  System.out.println("<---␣Radix␣Sort␣--->");
73                  System.out.println("<------------------->");
74                  System.out.println("Tama o␣Del␣Arreglo:␣");
75                  tam = scan.nextInt();
76                  int C[] = GenerarArreglo(tam);
77                  System.out.println("Arreglo␣Desordenado");
78                  print(C, tam);
79                  radixsort(C, tam);
80                  System.out.println("\nArreglo␣Ordenado");
81                  print(C, tam);
82              break;
83              case 3:
84                  System.out.println("<------------------->");
85                  System.out.println("<---␣Heap␣Sort␣--->");
86                  System.out.println("<------------------->");
87                  System.out.println("Tama o␣Del␣Arreglo:␣");
88                  tam = scan.nextInt();
89                  int D[] = GenerarArreglo(tam);
90                  System.out.println("Arreglo␣Desordenado");
91                  print(D, tam);
92                  sort(D);
93                  System.out.println("Arreglo␣Ordenado");
94                  print(D, tam);
95              break;
96          }
97      }
98
99      public static void main(String[] args){
100         menu();
101     }
102
103 }
```

# Conclusiones

## Opinion Personal.

Sin duda el poder analizar algoritmos de forma recursiva no es algo trivial. A veces es un poco complicado poder definir la funció y lejos de eso, lo que se complica es la condicion de paro ya que sin esta no podemos tener una certeza de que lo que estamos planteando como solucion es correcto.
De igual forma, a veces algunos algoritmos recursivos se comportan de diferente forma para los diferentes valores de los argumentos.

# Bibliografía

## Libros Recomendados

[ 1 ] Baase and Van Gelder. ”Computer Algorithms: Introduction to Design and Analysis”. Addison- Wesley.
[ 2 ] Thomas H. Cormen. ”Introduction to Algorithms”. The MIT press.