Estructura de Datos y Algoritmos
Departamento de electrónica
Universidad Técnico Federico Santa María

Ayudantía 10: Heap Sort, Radix Sort y Counting Sort

Objetivo Principal: Conocer cómo analizar e implementar los algoritmos de ordenamiento Heap Sort, Radix Sort y Counting Sort, comparándolos y analizando su complejidad algorítmica.

Autor: Kevin Pizarro Aguirre

1. Heap Sort

Uno de los tres grandes algoritmos de ordenamiento[1] basados en comparación y sin conocimiento de los datos. Usa como base la estructura de datos heap, ya sea max-heap para ordenar ascendentemente o min-heap para ordenar descendentemente. El max-heap consiste en que el nodo padre siempre será de valor mayor que los hijos, para min-heap funciona con valor menor que los hijos.

```
HEAPSORT(A)
BUILD-MAX-HEAP(A)
for i = A.length downto 2
    exchange A[1] with A[i]
    A.heap-size = A.heap-size − 1
    MAX-HEAPIFY(A, 1)

BUILD-MAX-HEAP(A)
A.heap-size = A.length
for i = ⌊A.length/2⌋ downto 1
    MAX-HEAPIFY(A, i)

MAX-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] > A[i]
    largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]
    largest = r
if largest ≠ i
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A, largest)
```

Figura 1: Pseudo código para algoritmo *Heap Sort*, con sub-rutinas *Build-Max-Heap y Max-Heapify*.

Al analizar el algoritmo nos daremos cuenta que su complejidad algorítmica viene dada por $T(n) = \Theta(n \log n)$. La ventaja es que es ordenamiento en el lugar, no pide más memoria como es el caso de Merge Sort.

[1] Los otros son Merge Sort y Quick Sort.

2. Counting Sort

Uno de los algoritmos de ordenamiento en **tiempo lineal**[2], pero que tiene restricciones en la entrada. Cada elemento de la entrada son enteros en el rango de 0 a k para algún k entero positivo. Se basa en contar cuantos elementos son menores al cual “estoy viendo” y luego posicionarlo en el lugar correspondiente. Notar que se utiliza un arreglo adicional para el algoritmo, es decir, más memoria.

```
COUNTING-SORT(A, B, k)
let C[0..k] be a new array
for i = 0 to k
    C[i] = 0
for j = 1 to A.length
    C[A[j]] = C[A[j]] + 1
// C[i] now contains the number of elements equal to i.
for i = 1 to k
    C[i] = C[i] + C[i − 1]
// C[i] now contains the number of elements less than or equal to i.
for j = A.length downto 1
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]] − 1
```

Figura 2: Pseudo código para algoritmo *Counting Sort*.

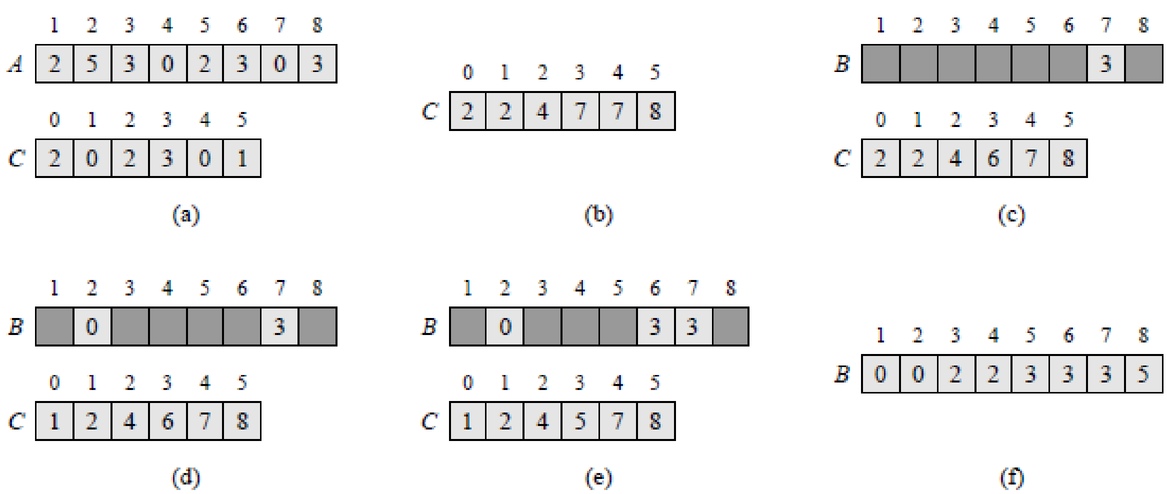

Figura 3: Ruteo para algoritmo *Counting Sort*.

Al analizar el algoritmo nos daremos cuenta que su complejidad algorítmica viene dada por $T(n) = \Theta(n + k)$, dependemos de la entrada y del rango de la entrada linealmente.

[2]Existen más que pueden ser revisados en ell libro *Introduction to Algorithms.*

3. Radix Sort

Otro de los algoritmos de ordenamiento en **tiempo lineal**, pero que tiene restricciones en la entrada. Cada elemento de la entrada tiene la misma cantidad de dígitos (d), son números enteros. Ordena desde el dígito menos significativo hasta el más significativo, desde el primer elemento hasta el el último para orden ascendente.

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

Figura 4: Pseudo código para algoritmo *Radix Sort*.

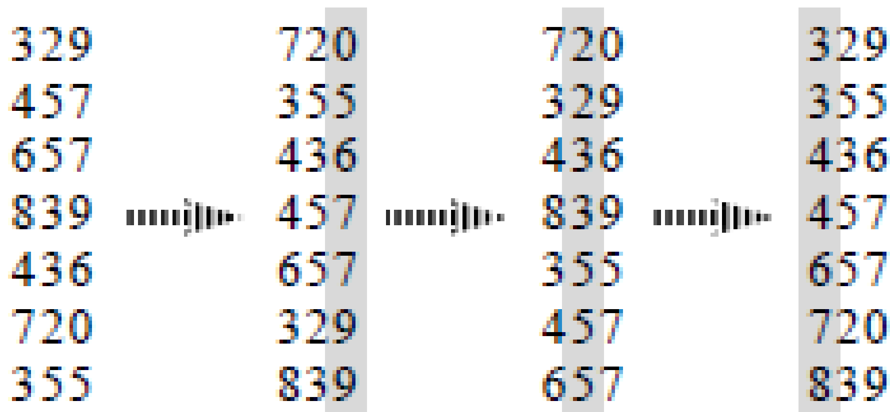

Figura 5: Ruteo para algoritmo *Radix Sort*.

Al analizar el algoritmo nos daremos cuenta que su complejidad algorítmica viene dada por $T(n) = \Theta(n * d)$, dependemos de la entrada y del rango de la entrada linealmente.

4. Ejercicios

a) Buscar información sobre dónde puede ser implementado y la importancia de las estructuras del tipo heap y el algoritmo heapsort.