Universidad Nacional de Ingeniería
Ciencias de la Computación
Sorting in Linear Time

Yuri Nuñez Medrano *
ynunezm@gmail.com

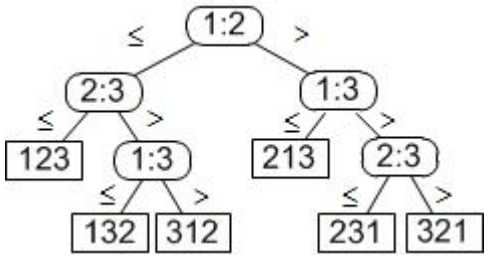

Figura 1: Arbol de decisión

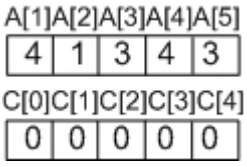

Figura 2: Datos de ingreso

Resumen

Se analizará algoritmos que tiene un tiempo de ejecución menor igual a $\Omega(nlgn)$.

1. Ordenamiento en Tiempo Lineal

Analizaremos algoritmos que pueden ordenar n elementos con $\Omega(nlgn)$ comparaciones en el peor de los casos. Tambien se examinará tres algorítmos de ordenamiento que ejecutan en tiempo lineal

1.1. Modelo de Arbol de Decision

En general $\langle a_1, a_2, a_3, ..., a_n\rangle$ cada nodo interno esta etiquetado i, j donde $i \wedge j \in \{1, 2, 3, ...n\}$ en el que compara.
- se compara $a_i vs a_j$
- el subarbol izquierdo compara $a_i \leq a_j$.
- el subarbol derecho compara $a_i > a_j$.
- cada leaf(hoja) tiene una permutación de $\langle a_{\pi(1)}, a_{\pi(2)}, .., a_{\pi(n)}\rangle$

Comparando
- un arbol para cada n.
- un algoritmo que divide y realiza una comparación.
- El arbol lista comparaciones a lo largo de todas las instrucciones trazadas.
- El tiempo de ejecución (número de comparaciones).
- El caso peor tiempo de ejecución es la altura h del arbol.
- Limite inferior en un ordenamiento de n elementos tiene una altura $\Omega(nlgn)$.

Prueba: nos enfocamos en el numero mínimo y máximo de hojas que pudiera tener, entonces quedaria así.
$n! \leq l \leq 2^h$
$n! \geq 2^h$
$h \geq lgn!$
teniendo en cuenta la formula de Stirling.
$n! = \sqrt{2\pi n}(\frac{n}{e})^n(1 + \Theta(\frac{1}{n}))$
reeemplazamos por conveniencia.
$h \geq lg(\frac{n}{e})^n$
$h \geq nlg(\frac{n}{e})$
$h \geq nlgn - nlge$
$h = \Omega(nlgn)$

1.2. Counting Sort

Se asume que cada uno de los n elementos de input(ingreso) es un entero de en el rango de 0 a k, para algun entero de k. Cuando $k = O(n)$, el ordenamiento ejecuta en un tiempo $\Theta(n)$.
Ejm: con los siguientes elementos donde $A = 4, 1, 3, 4, 3$ evaluamos de la linea 1 a la 3 del algoritmo 1 en el gráfico 2.
Posteriormente, de la linea 4 al 5 del algoritmo 1 empezamos a contar los elementos y lo guardamoe en el array C,como se ve en la siguiente figura 3

*Escuela de Ciencias de la Computación, 27-08-15

C[0]	C[1]	C[2]	C[3]	C[4]
0	1	0	2	2

Figura 3: El conteo

C[0]	C[1]	C[2]	C[3]	C[4]
0	1	1	3	5

Figura 4: El conteo

Luego en la linea 7 al 8 del algoritmo 1 en el array sumamos el iesimo elemeno mas el (i-1)esimo en i, como se ve en la siguiente figura 4
Finalmente obtenemos el valor de B en la siguiente figura 5

1.3. Radix Sort

Este algoritmo fue empleado en las maquinas de tarjetas perforadas, la que sólo se encuentran en los museos.
En el algoritmo 2 evaluamos el ejemplo en la figura 6 Asumimos mejor el ordenamiento empezando por el digito menos significativo.
El dígito en la posición d.
-Asumimos po rinducción el ordenamiento.
-Orden en el digito d, si dos elementos tienen el mismo d-esimo elemento se quedara con el mismo orden.
- Si son diferentes el d-esimo dígito se ordena.

Análisis
- Usamos counting sort por digito $O(n+k)$.
- Evaluamos a d digitos entonces quedara $O(d(n+k))$.

1.4. Bucket sort

Se asume que los datos de input (ingreso) estan distribuido uniformemente y tendran un tiempo de ejecución con un tiempo promedio de $O(n)$.
Entonces te asegura que los n_i son tan pequeños que puedes ordenarlo con un insertion sort.

Los datos input son valores de 0 al 1 . El algoritmo

C[0]	C[1]	C[2]	C[3]	C[4]
0	0	1	1	3

B[1]	B[2]	B[3]	B[4]	B[5]
1	3	3	4	4

Figura 5: El conteo

Algorithm 1: COUNTING_SORT(A,B,k)

Input: A[1...n], cada A[i] $i \in 1...k$
Output: B[1...n] ordenado de A

```
1  let C[0...k] be new array
2  for i = 0 to k do
3      C[i]=0
4  for j = 1 to A.length do
5      C[A[j]]=C[A[j]]+1
6  //C[i] now contain the numbers of elements equal to i
7  for i = 1 to k do
8      C[i]=C[i]+C[i-1]
9  //C[i] now contain the numbers of elements less than or
   equal to i
10 for j = A.length to 1 do
11     B[C[A[j]]]=A[j]
12     C[A[j]]=C[A[j]]-1
```

Algorithm 2: RADIX_SORT(A,d)

```
1  for j = 1 to d do
2      use a stable sort to sort array A on digit i
```

329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

Figura 6: Radix sort

Algorithm 3: BUCKET_SORT(A)

```
1  let B[0...n-1] be new array
2  n=A.length
3  for i = 0 to n − 1 do
4      make B[i] an empty list
5  for i = 1 to n do
6      insert A[i] into list B[⌊nA[i]⌋]
7  for j = 0 to n − 1 do
8      sort list B[i] with insertion sort
9  concadenate the list B[0],B[1],...,B[n-1] together ib order
```

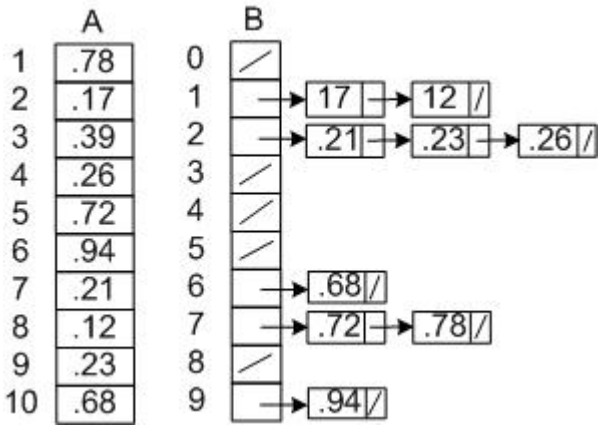

Figura 7: Radix sort

3 describe que se categorizan los valores en rangos de dependiendo del numero de elementos, de igual tamaño, en donde se adicionan como una lista enlazada dependiendo su rango, despues de haber ingresado los rangos se ordenan dentro de cada iesima lista, y por último se extrae todos los datos ya ordenados de cada lista, como en la figura 7.

Referencias

[H.Cormen et al., 2009] H.Cormen, T., Leiserson, C., and Riverson, R. L. (2009). *Algorithms*. The MIT Press.