Análisis y Complejidad de Algoritmos

Métodos de Ordenamiento

Arturo Díaz Pérez

- Tipos de ordenamiento y medidas de eficiencia
- Algoritmos básicos
- QuickSort
- HeapSort
- BinSort
- RadixSort
- Arboles de Decisión

Análisis y Diseño de Algoritmos *Sorting*-1

Tipos de Ordenamiento

- ***Ordenamiento interno***.
  - Se lleva a cabo completamente en memoria principal. Todos los objetos que se ordenan caben en la memoria principal de la computadora
- ***Ordenamiento externo***.
  - No cabe toda la información en memoria principal y es necesario ocupar memoria secundaria. El ordenamiento ocurre transfiriendo bloques de información a memoria principal en donde se ordena el bloque y este es regresado, ya ordenado, a memoria secundaria

Análisis y Diseño de Algoritmos *Sorting*-2

Formulación

- Registros: $r_1, r_2, ..., r_n$
- Llaves: $k_1, k_2, ..., k_n$
- Obtener la secuencia

$$r_{i_1}, r_{i_2}, \ldots, r_{i_n}$$

- tal que,

$$k_{i_1} \le k_{i_2} \le \ldots \le k_{i_n}$$

Análisis y Diseño de Algoritmos *Sorting*-3

Criterios de Eficiencia

- Criterios de eficiencia
  - El número de pasos
  - El número de comparaciones entre llaves para ordenar *n* registros.
    - De utilidad cuando la comparación entre llaves es costosa
  - El número de movimientos de registros que se requieren para ordenar *n* registros.
    - De utilidad cuando el movimiento de registros es costoso

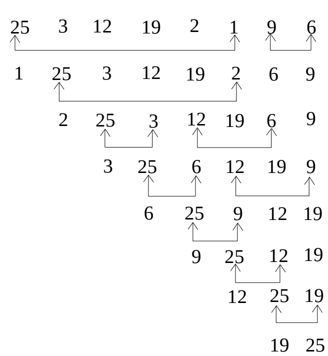
Análisis y Diseño de Algoritmos *Sorting*-4

Métodos Simples de Ordenamiento

- Métodos simples
  - Método de Burbujeo
  - Método de Inserción
  - Método de Selección

Análisis y Diseño de Algoritmos *Sorting*-5

Método de Burbujeo




```
void Burbujeo( int A[], int n)
{
  int i,j;
  for( i=0; i < n-1; i++ )
      for( j=n-1; j > i; j--)
          if( A[j] < A[j-1] )
              Intercambia( &A[j], &A[j-1] );
}
```

Análisis y Diseño de Algoritmos *Sorting*-6

Método de Burbujeo

```
void Burbujeo( int A[], int n)
{
  int i,j;
  for( i=0; i < n-1; i++ )
     for( j=n-1; j > i; j--)
        if( A[j] < A[j-1] )
           Intercambia( &A[j], &A[j-1] );
}
```

Comparaciones: $c = \sum_{i=1}^{n-1}(n-i) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2}$

Movimientos: $M_{min} = 0$

$$M_{max} = \sum_{i=1}^{n-1}(n-i) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2}$$

Análisis y Diseño de Algoritmos *Sorting-7*

Método de Selección

25 3 12 19 2 1 9 6

1 3 12 19 2 25 9 6

2 12 19 3 25 9 6

3 19 12 25 9 6

6 12 25 9 19

9 25 12 19

12 25 19

19 25

```
void Selección( int A[], int n )
{
   int   i, j, imin;
   for( i=0; i < n-1; i++ ) {
     imin = i;
     for( j = i+1; j > n; j++ )
       if( A[j] < A[imin] )
         imin = j;
     intercambia( &A[i], &A[imin] );
   }
 }
```

Análisis y Diseño de Algoritmos *Sorting-8*

Método de Selección

```
void Selección( int A[], int n )
{
   int   i, j, imin;
   for( i=0; i < n-1; i++ ) {
     imin = i;
     for( j = i+1; j > n; j++ )
       if( A[j] < A[imin] )
         imin = j;
     intercambia( &A[i], &A[imin] );
   }
 }
```

Comparaciones: $C = \sum_{i=1}^{n-1} n - i = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2}$

Movimientos: $M = n - 1$

Análisis y Diseño de Algoritmos *Sorting*-9

Método de Inserción

25 3 12 19 2 1 9 6
3 25 12
3 12 25 19
3 12 19 25 2
2 3 12 19 25 1
1 2 3 12 19 25 9
1 2 3 9 12 19 25 6
1 2 3 6 9 12 19 25

```
void Inserción( int A[], int n )
{
 int i,j;

 for( i=1; i < n; i++ ) {
   j:= i;
   while( j > 0  &&  A[j] < A[j-1] ) {
     Intercambia( &A[j], &A[j-1] );
     j--
   }
 }
}
```

Análisis y Diseño de Algoritmos *Sorting*-10

Método de Inserción

```
void Inserción( int A[], int n )
{
 int i,j;

 for( i=1; i < n; i++ ) {
   j:= i;
   while( j > 0  &&  A[j] < A[j-1] ) {
     Intercambia( &A[j], &A[j-1] );
     j--
   }
 }
}
```

Comparaciones:

$$C_{min} = n-1$$

$$C_{max} = \sum_{i=1}^{n-1}(n-i) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2}$$

Movimientos:

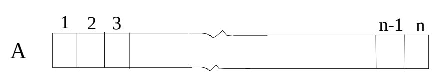
$$M_{min} = 0$$

$$M_{max} = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2}$$

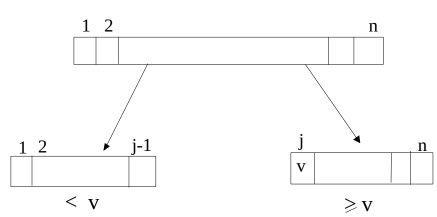
Análisis y Diseño de Algoritmos *Sorting*-11

QuickSort

- Dado un arreglo desordenado de n elementos

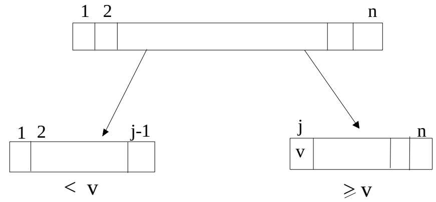


- selecciona un valor *v* del arreglo, llamado ***pivote***, y rearregla los elementos del arreglo de manera que todos los elementos menores que *v* queden colocados antes que *v* y todos los elementos mayores o iguales que *v* queden colocados después que *v*.



Análisis y Diseño de Algoritmos *Sorting*-12

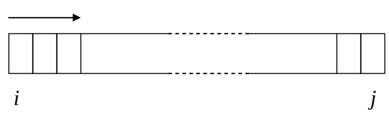
QuickSort




```
void quicksort( int A[], int i, int j )
{
    if( desde A[i] hasta A[j] existen al
        menos dos llaves distintas )  {
      Seleccionar un pivote, v
      Permutar A[i], ...,A[j], tal que,
      para algún k entre i+1 y j,
      A[i], ..., A[k-1] < v  y
      A[k], . . . , A[j] ≥ v
      quicksort( A, i, k-1 );
      quicksort( A, k, j );
    }
  }
```

Análisis y Diseño de Algoritmos *Sorting*-13

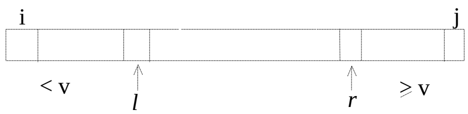
QuickSort: Pivote




```
int pivote( int A[],int i,int j )
{
  int k, r;

  for( k = i+1; k <= j; k++ ) {
    /* Compara dos elementos  */
    r = comp( A[k], A[i] );
    if( r < 0 )
      return i;
    else if( r > 0 )
      return k;
  }
  /* No hay llaves diferentes */
  return -1;
}
```

Análisis y Diseño de Algoritmos *Sorting*-14

QuickSort: Particionamiento



A[p] $< v$, $i \leq p < l$
A[p] $\geq v$, $r < p \leq j$

```
int particion( int A[],int i, int j,
               int v )
  .
  .
{
  int l,r;
  l=i; r=j;
  do {
    Intercambia( A[l], A[r] )
    while(  comp( A[l], v )  <  0  )
      l++;
    while(  comp( A[r], v ) >= 0 )
      r--;
  } while ( l<r );
  return l;
}
```

El tiempo de ejecución de partición es $O(j\text{-}i+1)$

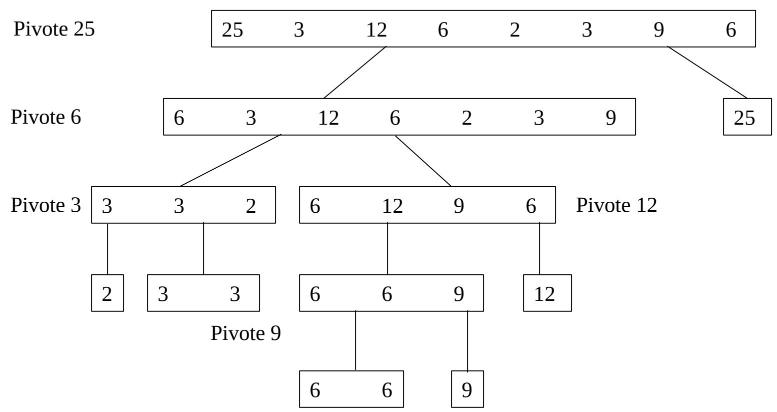
Análisis y Diseño de Algoritmos *Sorting*-15

QuickSort: Particionamiento

```
void quicksort( int A, int i, int j )
{
  int ind, k;
  ind = pivote( A, i, j );
  if( ind >= 0 ) {
    k = particion(A, i, j, A[ind] );
    quicksort( A, i, k-1 );
    quicksort( A, k, j );
  }
}
```

Análisis y Diseño de Algoritmos *Sorting*-16

QuickSort: Ejemplo

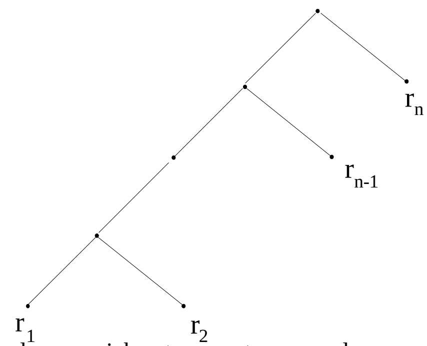


Caso Promedio, Peor y Mejor

- ¿Puede dar una idea de cuales son los casos
  - Promedio
  - Peor
  - Mejor?

QuickSort: Peor Caso

- ¿cuántos llamados recursivos se hacen a quicksort?
- Un ejemplo del peor caso se puede mostar en la figura siguiente:

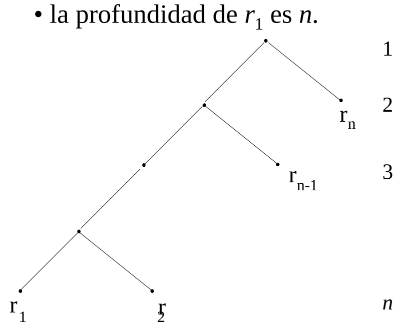


- El tiempo tomado por quicksort para este caso es la suma sobre todos los elementos del número de veces que el elemento forma parte de un subarreglo en el cual se hace un llamado a quicksort.

Análisis y Diseño de Algoritmos *Sorting*-19

QuickSort: Peor Caso

- En el árbol anterior, se representa por la profundidad de cada elemento del arreglo.
  - La profundidad de r_i es $n-i+2$, $2 \le i \le n$,
  - la profundidad de r_1 es n.



La suma de profundidad es.

$$n + \sum_{i=2}^{n} n - i + 2 = n + \sum_{i=2}^{n} i$$

$$= n + \frac{n(n+1)}{2} - 1$$

$$= \frac{n^2}{2} + \frac{3n}{2} - 1$$

- Por lo tanto, en el peor caso quicksort toma un tiempo $O(n^2)$

Análisis y Diseño de Algoritmos *Sorting*-20

QuickSort: Caso Promedio

- No existen elementos con llaves iguales; todos los elementos tienen llaves diferentes.
- Cuando se llama a quicksort(A, l, m) todos los órdenes (permutaciones) de A[l], ...,A[m] son igualmente probables.
- Si el ordenamiento inicial es $r_1, r_2, ..., r_n$ por el método de elección del pivote, éste puede ser ó r_1 ó r_2.
- Suponga que existen *i* elementos más pequeños que el pivote.
- Si el pivote, v, es r_1, uno de los i elementos más pequeños que el pivote está en la segunda posición.
- Similarmente, si el pivote es r_2, uno de los i elementos más pequeños que el pivote está en la primera posición

Análisis y Diseño de Algoritmos — *Sorting*-21

QuickSort: Caso Promedio

- El pivote debe ser el $(i+1)$-ésimo elemento más pequeño, del arreglo.
  - La probabilidad de que cualquier elemento, tal como el $(i+1)$-ésimo más pequeño, aparezca en la primera posición es $1/n$.
  - Dado que ya apareció en la primera posición, la probabilidad de que el segundo elemento sea uno de los i elementos más pequeños de los $n-1$ restantes es $i/(n-1)$.
  - La probabilidad de que el pivote aparezca en la primera posición y uno de los i elementos más pequeños en la segunda posición es $i/n(n-1)$.
- Similarmente, la probabilidad de que el pivote aparezca en la segunda posición y uno de los i elementos más pequeños en la primera posición es $i/n(n-1)$

Análisis y Diseño de Algoritmos — *Sorting*-22

QuickSort: Caso Promedio

- Por lo tanto, la probabilidad de que haya ***i*** elementos más pequeños que el pivote es **2*i*/*n*(*n*-1)**

```
void quicksort( int A, int i, int j )
{
  int ind, k;
  ind = pivote( A, i, j );
  if( ind >= 0 ) {
    k = particion(A, i, j, A[ind] );
    quicksort( A, i, k-1 );
    quicksort( A, k, j );
  }
}
```

$$T(n) \leq \text{tiempo de la partición} + P_i[T(i) + T(n-i)]$$

$$T(n) \leq c_2 n + \sum_{i=1}^{n-1} \frac{2i}{n(n-1)} [T(i) + T(n-i)]$$

Análisis y Diseño de Algoritmos *Sorting-23*

QuickSort: Caso Promedio

- Se puede demostar que

$$\sum_{i=1}^{n-1} f(i) = \sum_{i=1}^{n-1} f(n-i) \Rightarrow \sum_{i=1}^{n-1} f(i) = \frac{1}{2}\sum_{i=1}^{n-1} f(i) + f(n-i)$$

- Aplicándolo a la recurrencia con $T(n)$, se tiene

$$T(n) \leq c_2 n + \sum_{i=1}^{n-1} \frac{2i}{n(n-1)} [T(i) + T(n-i)] \quad \boxed{f(i)}$$

$$T(n) \leq \frac{1}{2}\sum_{i=1}^{n-1}\left(\frac{2i}{n(n-1)}[T(i) + T(n-i)] + \frac{2(n-i)}{n(n-1)}[T(n-i) + T(i)] \right) + c_2 n$$

$$= \frac{1}{n-1}\sum_{i=1}^{n-1}\left(\frac{i}{n}[T(i) + T(n-i)] + \frac{n-i}{n}[T(n-i) + T(i)] \right) + c_2 n$$

$$= \frac{1}{n-1}\sum_{i=1}^{n-1}(T(i) + T(n-i)) + c_2 n$$, por lo tanto

Análisis y Diseño de Algoritmos *Sorting-24*

QuickSort: Caso Promedio

$$T(n) \le \frac{1}{n-1}\sum_{i=1}^{n-1}(T(i)+T(n-i)) + c_2 n$$

- Por lo tanto,

$$T(n) \le \frac{2}{n-1}\sum_{i=1}^{n-1}T(i) + c_2 n$$

- Se puede probar probar por inducción que

  $T(n) \le c\ n\log_2 n$

  para alguna constante c y para $n \ge 2$.

Análisis y Diseño de Algoritmos — *Sorting*-25

QuickSort: Caso Promedio

$$T(n) \le \frac{2}{n-1}\sum_{i=1}^{n-1}T(i) + c_2 n$$

- Supongamos que $T(1) = c_1$
- Para $n = 2$
  - ← $T(2) \le 2c_1 + 2c_2 = 2(c_1+c_2)\log_2 2$,
  - ← Si $c \ge c_1+c_2$
  - ← $T(2) \le c2\log_2 2$

Análisis y Diseño de Algoritmos — *Sorting*-26

QuickSort: Caso Promedio

☞ Supongamos que es válida para todo $k < n$.

$$T(n) \le \frac{2}{n-1}\sum_{i=1}^{n-1} T(i) + c_2 n$$

$$\le \frac{2}{n-1}\sum_{i=1}^{n-1} ci\log i + c_2 n$$

$$T(n) \le \frac{2c}{n-1}\left[\sum_{i=1}^{\frac{n}{2}} i\log i + \sum_{i=\frac{n}{2}+1}^{n-1} i\log i\right] + c_2 n$$

$$\le \frac{2c}{n-1}\left[\sum_{i=1}^{\frac{n}{2}} i(\log n - 1) + \sum_{i=\frac{n}{2}+1}^{n-1} i\log n\right] + c_2 n$$

$$\le \frac{2c}{n-1}\left[\frac{n}{4}\left(\frac{n}{2}+1\right)\log n - \frac{n}{4}\left(\frac{n}{2}+1\right) + \frac{3}{4}n\left(\frac{n}{2}-1\right)\log n\right] + c_2 n$$

Análisis y Diseño de Algoritmos — *Sorting*-27

QuickSort: Caso Promedio

$$T(n) \le \frac{2c}{n-1}\left[\frac{n}{4}\left(\frac{n}{2}+1\right)\log n - \frac{n}{4}\left(\frac{n}{2}+1\right) + \frac{3}{4}n\left(\frac{n}{2}-1\right)\log n\right] + c_2 n$$

$$\le \frac{2c}{n-1}\left[\left(\frac{n^2}{2} - \frac{n}{2}\right)\log n - \left(\frac{n^2}{8} + \frac{n}{4}\right)\right] + c_2 n$$

$$\le \frac{2c}{n-1}\left[\frac{n(n-1)}{2}\log n - \frac{n^2+2n}{8}\right] + c_2 n$$

$$\le cn\log n - \frac{cn}{4} - \frac{cn}{2(n-1)} + c_2 n$$

☞ Si $c \ge 4c_2$, entonces, la suma del segundo y cuarto término no es mayor que 0,

☞ Ya que el tercer término es una contribución negativa, entonces **$T(n)$ £ $cn\log_2 n$**.

Análisis y Diseño de Algoritmos — *Sorting*-28

HeapSort

- Estructura de datos abstracta
  - AGREGA, BORRAMIN, VACIA e INICIA

```
INICIA( S);
for( cada elemento, x, a ordenar )
    AGREGA( x, S );
while( !VACIA(S) ) {
    y = BORRA_MIN(S);
    printf(" ... ", y );
}
```

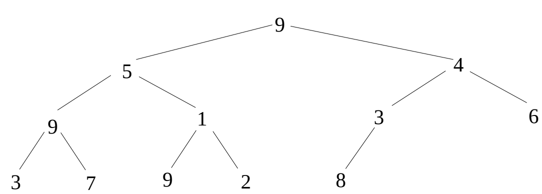
Si las operaciones VACIA e INICIA toman un tiempo $O(1)$ y las operaciones AGREGA y BORRA_MIN toman un tiempo $O(\log n)$, donde n es el número de elementos a ordenar, es claro que el método de ordenamiento anterior tomaría un tiempo $O(n\log n)$

Análisis y Diseño de Algoritmos *Sorting*-29

Arbol Parcialmente Ordenado

- Un árbol parcialmente ordenado cumple con las siguientes propiedades:
  - El valor de un nodo en el árbol no es mayor que el de sus hijos
- Un árbol parcialmente ordenado se puede representar mediante un arreglo unidimensional, A, en el cual
  - La raíz es A[1], y
  - Los hijos del nodo A[i] son A[2i] y A[2i+1]

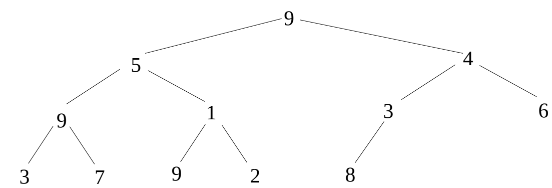
9	5	4	9	1	3	6	3	7	9	2	8
1	2	3	4	5	6	7	8	9	10	11	12



Análisis y Diseño de Algoritmos *Sorting*-30

HeapSort, cont.

9	5	4	9	1	3	6	3	7	9	2	8
1	2	3	4	5	6	7	8	9	10	11	12



- Nota:
  - ← Si n es el número de elementos del arreglo, $\lfloor n/2 \rfloor$ son nodos interiores del árbol binario.
- Sólo los nodos interiores se deben considerar para ordenar el árbol en forma parcial.

Análisis y Diseño de Algoritmos — *Sorting*-31

HeapSort: Descenso.

- Supongamos que los elementos

A[i], . . . ,A[j]

obedecen ya la propiedad de los árboles parcialmente ordenados, excepto posiblemente por A[i].

- La función siguiente desciende a A[i] hasta que se obtenga la propiedad de los árboles parcialmente ordenados.

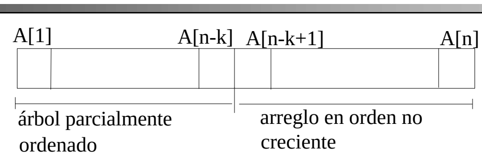
Análisis y Diseño de Algoritmos — *Sorting*-32

HeapSort: Descenso.

```
void desciende( int A, int i, int j )
{
  int r;

  r = i;
  while( r <= j/2) {
    if( 2*r+1 > j ) { /*  r tiene sólo un hijo  */
      if( comp( A[r], A[2*r]) > 0 )
            intercambia ( &A[r], &A[2*r] );
            r = j;
    } else { /*  r tiene dos hijos  */
      if( comp( A[r], A[2*r] ) > 0  &&
          comp( A[2*r], A[2*r+1] ) <= 0 ) {
        intercambia( &A[r], &A[2*r] );
        r = 2*r;
      } else if( comp( A[r], A[2*r+1] ) > 0  &&
          comp( A[2*r+1], A[2*r] ) <= 0 ) {
        intercambia( &A[r], &A[2*r+1] );
        r = 2*r+1;
      } else
        /* no se viola la propiedad de los árboles
        parcialmente ordenados */
        r = j;
    }
}
```

Análisis y Diseño de Algoritmos — *Sorting*-33

HeapSort, cont.



$A[n-k+1] \geqslant A[n-k+2] \geqslant \cdots \geqslant A[n]$

```
void HeapSort( A, n )
  .
  .
{
  int i;

  for( i = n/2; i >= 1; i-- )
  /* Inicialmente, establece la propiedad
    del árbol parcialmente ordenado */
    desciende ( A, i, n);
  for( i = n; i >= 1; i-- ) {
    /* Quita el menor elemento */
    intercambia( &A[1], &A[i] );
  /* reestablece el árbol parcialmente ordenado */
    desciende( A, 1, i-1 );
  }
}
```

Análisis y Diseño de Algoritmos — *Sorting*-34

Ordenamiento Lineal

- Supongamos que las llaves son enteros en el rango de 1 a *n*, y que **no existen llaves iguales**.
  - Si A y B son arreglos de tipo adecuado y los *n* elementos que van a ser ordenados están inicialmente en A, se pueden colocar en el arreglo B en el orden de sus llaves mediante

```
for( i = 1; i <= n; i++ )
  B[ A[i].llave ] = A[i];
```

- El ciclo completo toma un tiempo $O(n)$.

Ordenamiento Lineal

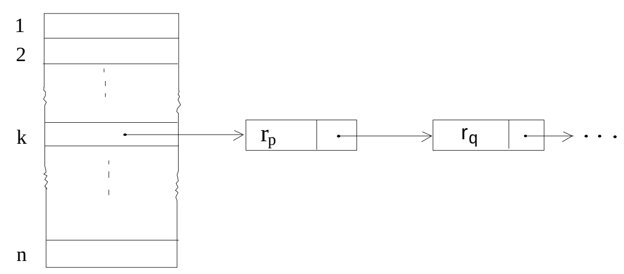
- Se puede usar otra forma para ordenar el arreglo A con llaves 1, ..., *n* usando el mismo arreglo en tiempo $O(n)$.
  - Se visitan A[1], . . . , A[*n*] en ese orden.
    - Si el registro A[*i*] tiene una llave $j \neq i$, se intercambia con A[*j*].
    - Si después del intercambio, el elemento que está ahora en A[*i*] tiene llave $k \neq i$, se intercambia A[*i*] con A[*k*], y se sigue así hasta que en A[*i*] quede el registro con llave *i*.

```
for( i=1; i <= n; i++ )
  while( A[i].llave != i )
       intercambia(&A[i], &A[ A[i].llave] );
```

  - Cada intercambio coloca algún registro en su lugar y una vez ahí nunca se mueva más.

Ordenamiento Lineal

Si se permiten llaves duplicadas, entonces, para cada valor llave, *k*, debe existir una lista, L[*k*], de todos los elementos con la llave *k*.



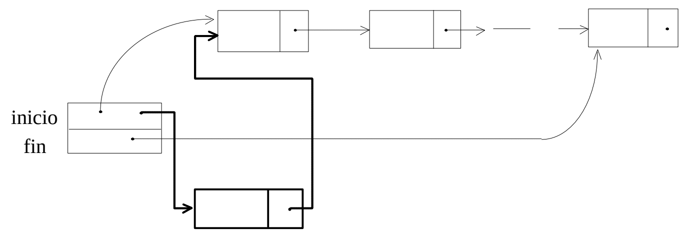
Ordenamiento Lineal

```
#define    MAX_LLAVE            . . .
typedef     . . .  LISTA

void BinSort( OBJETO A[], LISTA[], int n );
{
  LLAVE  K;
  int i;

  for( k = 1; k <= MAX_LLAVE; K++)
    /*Inicia las listas haciedolas nulas */
    INICIA( L[K] );

  for( i = 1; i <= n; i++ )
    /* Crea las listas de cada llave  */
    INSERTA( A[i], L[A[i].llave] );

  for( K = 2; K <= MAX_LLAVE; K++ )
    /* Crea una lista con todos los
      registros en orden*/
    CONCATENA( L[1], L[K] );
}
```

Si INICIA, INSERTA y CONCATENA toman un tiempo $O(1)$ y MAX_LLAVE < n, entonces, la función anterior toma un tiempo $O(n)$.

BinSort: Inserta

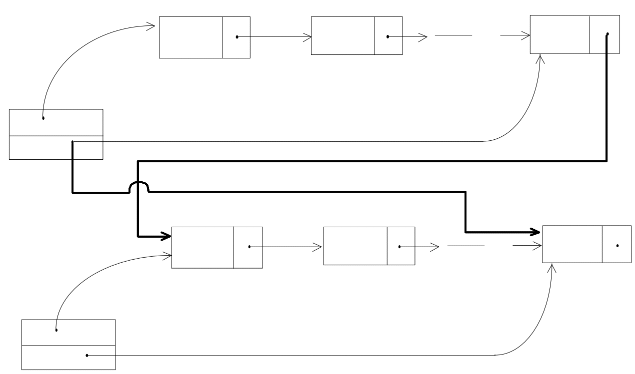
Operación de Inserción



Análisis y Diseño de Algoritmos *Sorting*-39

BinSort: Concatena

Operación de Concatenación



Análisis y Diseño de Algoritmos *Sorting*-40

BinSort

- Observaciones:
  - Suponga que se desea ordenar una lista de valores enteros en el rango 0 a n^2-1. Se utilizan n cubetas, una para cada uno de los enteros 0,1, . . . ,n-1
  - Dos fases:
    - Se coloca el entero i al final de la cubeta i **mod** n
    - Se seleccionan los valores en el orden en que están almacenados en las cubetas desde la 0 hasta la n-1. El entero i se coloca al final de la cubeta $\lfloor i/n \rfloor$

Análisis y Diseño de Algoritmos — *Sorting*-41

BinSort: Ejemplo

- **Ejemplo**
  - 0, 1, 81, 64, 4, 25, 36, 16, 9, 49

Primera Fase

Cubeta	Contenido
0	0
1	1,81
2	
3	
4	64,4
5	25
6	36,16
7	
8	
9	9,49

Segunda Fase

Cubeta	Contenido
0	0,1,4,9
1	16
2	25
3	36
4	49
5	
6	64
7	
8	81
9	

Análisis y Diseño de Algoritmos — *Sorting*-42

BinSort

- Observaciones:
  - Si los números i, j están en el rango 0 a n^2-1 se puede pensar que ellos se pueden expresar en base n como números de dos dígitos, esto es,
    - $i = an + b$ y $j = cn + d$, donde a, b, c, y d están dentro del rango 0,n-1
    - Si $i < j$, entonces, $a \le c$
      - Si $a < c$, i aparece en una cubeta menor a la de j en la segunda pasada, así que i precederá a j en el ordenamiento final.
      - Si $a = c$, entonces, $b < d$.
        - » En la primera pasada i debe preceder a j, i en b y j en d.
        - » En la segunda pasada i y j son colocados en la misma cubeta, i antes que j

Análisis y Diseño de Algoritmos *Sorting*-43

RadixSort

- Suponga que se desea ordenar una lista de elementos cuyos valores llave consisten de k componentes, f_1, f_2, . . . , f_k, cuyos tipos son t_1, t_2, ..., t_k.
- Orden lexicográfico
  - $(a_1, a_2, ..., a_k)$ es menor que $(b_1, b_2, ..., b_k)$, si sucede solo una de las condiciones siguientes:
    - $a_1 < b_1$
    - $a_1 = b_1$ y $a_2 < b_2$
    - . . .
    - $a_1 = b_1$, $a_2 = b_2$, ..., $a_{k-1} = b_{k-1}$ y $a_k < b_k$
    - Así, para algún j entre 0 y k-1, $a_1 = b_1$, ..., $a_j = b_j$ y $a_{j+1} < b_{j+1}$.

Análisis y Diseño de Algoritmos *Sorting*-44

RadixSort

- La idea clave detrás del método es ordenar todos los elementos de acuerdo al campo f_k (el dígito menos significativo) y luego concatenar las cubetas.
  - Después aplicar el método para f_{k-1} y así hasta f_1.
- Se debe asegurar que al agregar un elemento a una cubeta se haga siempre al final.
- En general, después de aplicar el método sobre los campos, f_k, f_{k-1}, ..., f_i los elementos aparecerán en el orden lexicográfico si sus llaves consistieran únicamente de los campos f_i, ..., f_k

RadixSort

```
void RadixSort()
/* Ordena la lista A de n elementos con llaves que
  consisten de los campos f1, f2, ..., fk de tipos
  t1, t2, . . . , tk, respectivamente. Utiliza arreglos
  Bi, 1 ≤ i ≤ k, para las cubetas de los valores en el
  campo fi */
{

(1) for( i = k; k >= 1; k-- ) {
(2)   for( cada valor v del tipo ti )
(3)     INICIA( Bi[v] );

(4)   for( cada elemento r en la lista A )
(5)     Colocar r al final de la cubeta Bi[v], donde v
        es el valor para el campo fi de la llave de r.
(6)   for( cada valor v del tipo ti, de menor a mayor)
(7)     Concatenar Bi[v] al final de A.
    }
  }
```

RadixSort

```
void RadixSort()
{
(1)  for( i = k; i >= 1; i-- ) {
(2)    for( cada valor v del tipo ti )
(3)      INICIA( Bi[v] );

(4)    for( cada elemento r en la lista A )
(5)      Colocar r al final de la cubeta Bi[v], donde v
         es el valor para el campo fi de la llave de r.
(6)    for( cada valor v del tipo ti, de menor a mayor)
(7)      Concatenar Bi[v] al final de A.
     }
   }
```

- Si s_i es el número de valores diferentes del tipo t_i.
  - ← Las líneas (2) y (3) toman un tiempo $O(s_i)$
  - ← Las líneas (4) y (5) toman un tiempo $O(n)$
  - ← Las líneas (6) y (7) toman un tiempo $O(s_i)$.
- El tiempo total del RadixSort es

$$\sum_{i=1}^{k} O(s_i + n) \qquad \Rightarrow \qquad O(kn + \sum_{i=1}^{k} s_i) \quad ú \quad O(n + \sum_{i=1}^{k} s_i)$$

Análisis y Diseño de Algoritmos *Sorting*-47

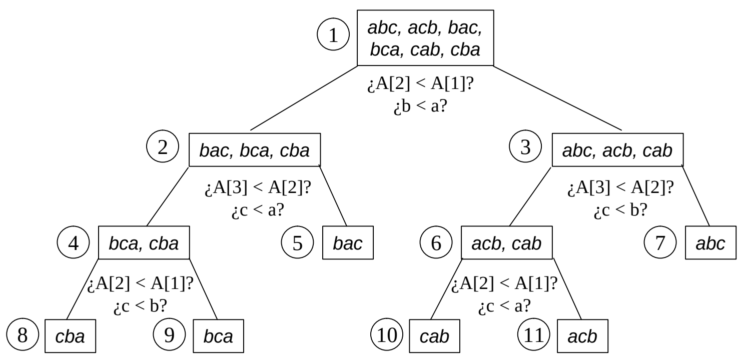
Arboles de Decisión

- Un árbol de decisión para un método de ordenamiento es un árbol binario en el cual sus nodos representan el estado del método después de hacer algún número de comparaciones.

- El estado de un programa de ordenamiento es esencialmente el conocimiento acerca de los ordenamientos iniciales que se han obtenido por el programa hasta ese momento.

Análisis y Diseño de Algoritmos *Sorting*-48

Arboles de Decisión



Arboles de Decisión

- si se ordena una lista de elementos de tamaño n, a_1, a_2, ..., a_n existen **n! = n(n-*1*) (n-2) (2) (1)** posibles ordenamientos correctos.

- La longitud del camino más largo de la raíz a una hoja es una cota inferior en el número de pasos ejecutados por el algoritmo en el peor caso.

- Ya que un árbol binario con k hojas debe tener un camino de longitud al menos log k, entonces, un algoritmo de ordenamiento que utiliza sólo comparaciones para ordenar una lista de n elementos debe tomar en el peor caso un tiempo Ω(log n!).

$$n! = n\cdot(n-1)\cdot\ldots\cdot(2)\cdot(1) \geq \underbrace{\frac{n}{2}\cdot\frac{n}{2}\cdot\ldots\cdot\frac{n}{2}}_{n/2 \text{ veces}} = \left(\frac{n}{2}\right)^{\frac{n}{2}}$$

$$\Longrightarrow \quad \log(n!) \geq \frac{n}{2}\log\frac{n}{2} = \frac{n}{2}\log n - \frac{n}{2}$$

Arboles de Decisión

- El ordenamiento por comparaciones requiere un tiempo W**(*n* log*n*)** en el peor caso

- Ejercicio: Uno puede preguntarse si existe un algoritmo de ordenamiento que utilice únicamente comparaciones y que toma un tiempo $\Omega(n \log n)$ en el peor caso, pero en el caso promedio toma un tiempo $O(n)$ o algo menor a $O(n \log n)$.

- ¿**Existe tal algoritmo**?

Análisis y Diseño de Algoritmos *Sorting*-51