Algoritmos sobre Búsqueda y Ordenamiento (Análisis y Diseño)

Dra. María de Luz Gasca Soto
Departamento de Matemáticas,
Facultad de Ciencias, UNAM.

17 de marzo de 2020

Luz Gasca S

Capítulo 6

Algoritmos Lineales de Ordenamiento

En este capítulo presentamos tres algoritmos de ordenamiento cuyo desempeño computacional es lineal. Dos de ellos (Radix y Bucket Sort) usan el Modelo de cómputo Radix.

Las estrategias de estos algoritmos aprovechan características especiales de los datos, Counting Sort, así como algunas versiones de Radix y Bucket Sort, requieren revisar la representación binaria de los datos.

Si las llaves son números, éstos pueden ser usados como direcciones o índices de tablas (archivos). Si las llaves son cadenas, éstas pueden ser partidas en sus componentes carácteres, los cuales pueden ser usados como índices. Sobre cualquier computadora digital es posible, al menos en teoría, tratar las llaves como números binarios cuyos componentes (bits) pueden ser usados en el proceso de ordenamiento.

6.1. Counting Sort

Esta técnica pertenece a los llamados métodos de ordenación lineal. Dada su naturaleza este método presupone condiciones especiales sobre la secuencia. Los elementos de la secuencia deben ser enteros no negativos, pueden estar repetidos y deben estar contenidos en el rango entre 0 y k, para algún entero k. Este método, como su nombre lo indica, realiza el ordenamiento contando los elementos (su frecuencia); por ello sólo puede ordenar elementos que sean contables o que a través de alguna función biyectiva se le pueda asociar un número natural.

Estrategia

La idea básica consiste en determinar para cada elemento s_i en la secuencia, el número de elementos menores que él. Esta información es usada para poner al elemento s_i directamente en su posición en el arreglo de salida. Por ejemplo si hay 14 elementos menores

que s_i entonces éste debe ser colocado en la posición 15.

Este esquema debe ser modificado ligeramente para manejar la situación en la cual varios elementos tengan el mismo valor, ya que no queremos ponerlos todos en la misma posición. Para calcular el número de elementos menores al elemento s_i, se realizan las siguientes acciones:

1. Se toma el rango, para determinar el mínimo y el máximo elemento.
2. Se crea un vector auxiliar para contar el número de apariciones de cada elemento en la secuencia.
3. Se estima la cantidad de elementos menores al número en revisión, esto se hace sumando los valores contiguos a la casilla en revisión.
4. Se crea un arreglo que contendrá a los elementos ordenados.
5. Se recorren la secuencia y el arreglo auxiliar colocando al elemento examinado en su posición correspondiente en el arreglo final.

Para ilustrar la estrategia consideremos el ejemplo de la Figura 6.1. Se desea ordenar de forma ascendente a la secuencia $S = \{2, 5, 3, 0, 2, 3, 0, 3, 1, 5\}$.

Para la primera iteración (i=1), iniciamos construyendo el vector auxiliar C el cual es del tamaño del rango de los elementos; en el ejemplo es de longitud 6 debido a que el rango va de 0 a 5. Este vector almacena en cada una de sus localidades el número de veces que aparece ese elemento en la secuencia, así pues a la localidad cero le asigna el valor de 2 ya que el cero aparece dos veces en la secuencia a ordenar. Una vez que se tiene este vector se crea otro, que llamamos B, el cual almacenará a los elementos de la secuencia en orden. Para iniciar con el ordenamiento es necesario saber cuántos datos son menores a un determinado elemento, esto se consigue sumando el valor de la localidad anterior, observemos (i=2).

Con estos datos ya es posible empezar el proceso de ordenamiento, para ello se recorren la secuencia S y el vector C, de la siguiente forma: el primer elemento en S es 2, en C vemos que esta localidad tiene almacenado el valor 5, en (i=2), entonces asignamos a la localidad $B[5]$ el valor 2 y restamos uno al valor de $C[2]$, como se observa en (i=3). Seguimos este procedimiento. Cuando (i=12) B contiene la secuencia ordenada.

Análisis de Complejidad

Determinar el tiempo que se tarda en calcular tanto el mínimo como el máximo elementos es constante, $O(1)$ ya que conocemos el rango de los datos.

Para contar las ocurrencias de cada elemento en la secuencia, tenemos que recorrer la secuencia completamente, por lo cual esta parte del proceso requiere tiempo $O(n)$. Calcular la cantidad de elementos menores en el vector C toma tiempo $O(k)$ ya que se emplean $(k-1)$ sumas y suponemos que cada suma consume tiempo constante.

Finalmente, recorrer tanto la secuencia S como el vector C e insertar un valor en el vector B tiene complejidad de $O(n+k+n) = O(2n+k) = O(n)$. Por lo tanto, podemos concluir que esta técnica tiene una complejidad total de $O(n)$, es decir, lineal.

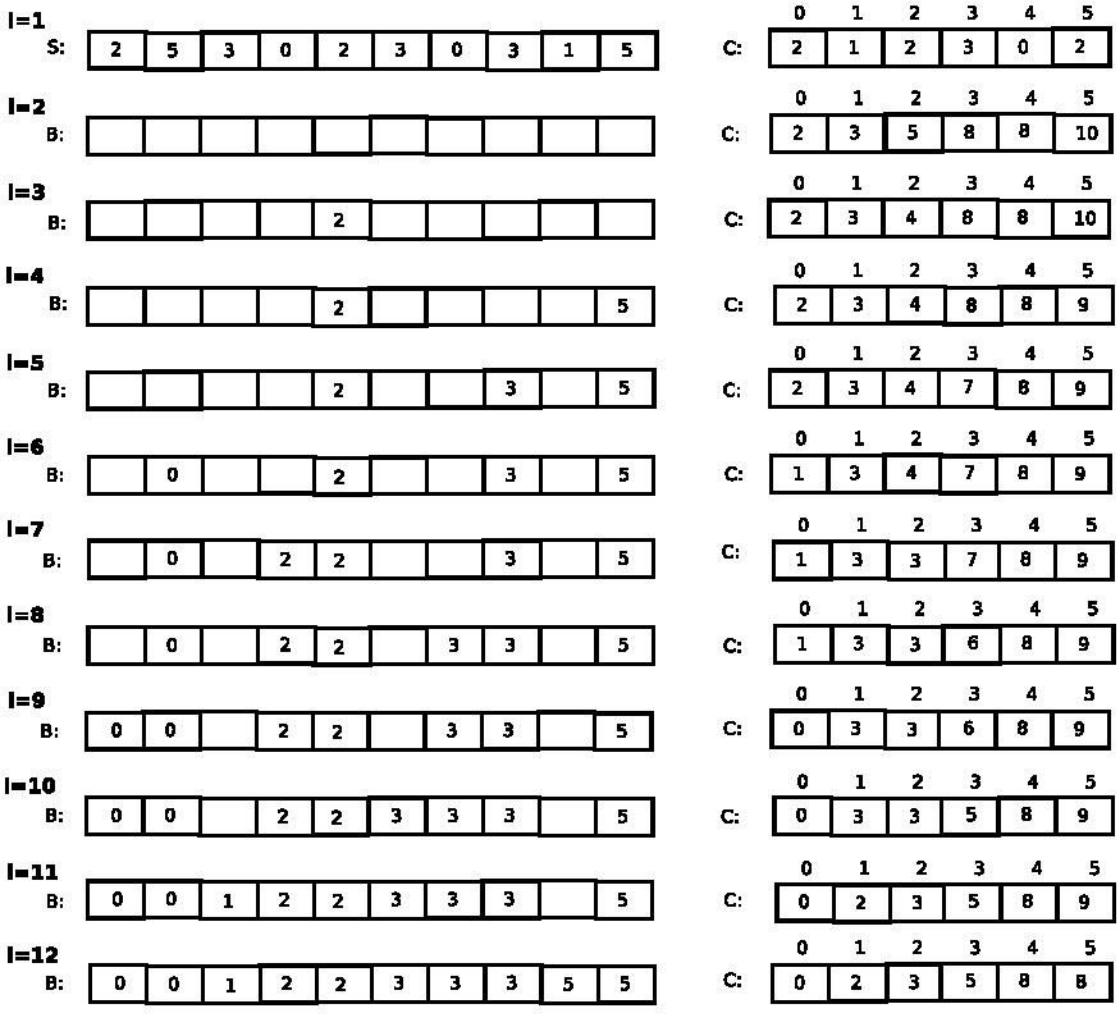


Figura 6.1: Ejemplo de Counting Sort

Listado 26 Pseudocódigo para Counting sort

```
//PreC: S secuencia no vacia contenida en el arreglo A de n enteros
//Post: El arreglo B contiene a la secuencia ordenada

 countingSort(array A, B; int n; int k){
   int i, j, n;
   array of integers  C[1..k];

   C = array.create();              // Arreglo auxiliar para contar

   for ( i=1; i<=k; i++)            // Limpia el arreglo auxiliar
      C[i] = 0;

   for (j=1; j<=n; j++)             // Calcula la frecuencia
      C[A[j]] = C[A[j]]+1;          // Al finalizar C[j] indica el numero
                                    //  de elementos iguales a j

   for (i=2 ;i<=k; i++)             // determina los elementos menores
      C[i] = C[i] + C[i-1];         //  o iguales a i

   for (j=n; j>=1; j--){            // Auxiliandose del arreglo temporal C
       B[C[A[j]]] = A[j];           // para tomar los datos del arreglo A,
          C[A[j]] = C[A[j]]-1;      // pasa los datos al arreglo de salida B
   }
} // end counting Sort
```

Algoritmo

El Listado 26 presenta un pseudicódigo para para Counting Sort, se supone que la entrada en un arreglo $A[1..n]$ con longitud n. Requerimos los otros dos arreglos: $B[1..n]$ que será la salida del arreglo ordenado y $C[1..k]$ que será temporal durante el proceso.

En este pseudocódigo se observa de forma clara el desempeño lineal. Se puede observar los ciclos for realizan una tarea específica y una vez que la termina no interfiere con el siguiente ciclo. En el primer ciclo se limpia el vector C, mientras que en el segundo se cuentan las ocurrencias de los datos correspondientes a las localidades en la secuencia. En el tercer ciclo se cuentan los elementos menores a un determinado dato y, finalmente, en el último ciclo se recorren la secuencia y el vector C para obtener la secuencia ordenada en el vector B.

Ejemplo

Suponga que se desea ordenar el arreglo $A = [3, 6, 4, 1, 3, 4, 1, 4]$ de enteros positivos cuyos elementos están en el intervalo $[1, 6]$. Tenemos $k = 6$. Siguiendo el código dado en el Listado 15, ilustraremos el ejemplo. El primer for únicamente pone cero a cada localidad del vector C.

El segundo for se desarrolla de la siguiente manera:

$A[1] = 3, \Rightarrow C[3] \leftarrow C[3] + 1 = 0 + 1 = 1;$
$A[2] = 6, \Rightarrow C[6] \leftarrow C[6] + 1 = 0 + 1 = 1;$
$A[3] = 4, \Rightarrow C[4] \leftarrow C[4] + 1 = 0 + 1 = 1;$
$A[4] = 1, \Rightarrow C[1] \leftarrow C[1] + 1 = 0 + 1 = 1;$
$A[5] = 3, \Rightarrow C[3] \leftarrow C[3] + 1 = 1 + 1 = 2;$
$A[6] = 4, \Rightarrow C[4] \leftarrow C[4] + 1 = 1 + 1 = 2;$
$A[7] = 1, \Rightarrow C[1] \leftarrow C[1] + 1 = 1 + 1 = 2;$
$A[8] = 4, \Rightarrow C[4] \leftarrow C[4] + 1 = 2 + 1 = 3.$

Quedando el arreglo C tal que $C[i]$ contiene el número de elementos iguales a i, la Figura 6.2(a) muestra el contenido del arreglo C:

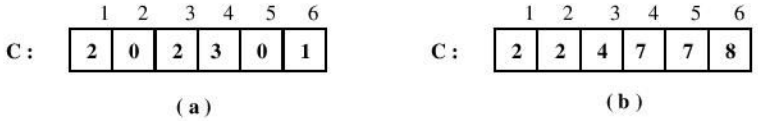


Figura 6.2: Arreglo C después del 2^o y 3^{er} for, respectivamente.

El tercer for se trabaja de la siguiente manera:

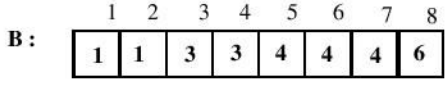
$C[2] \leftarrow C[2] + C[1] = 0 + 2 = 2; \quad C[3] \leftarrow C[3] + C[2] = 2 + 2 = 4;$
$C[4] \leftarrow C[4] + C[3] = 4 + 3 = 7; \quad C[5] \leftarrow C[5] + C[4] = 0 + 7 = 7;$
$C[6] \leftarrow C[6] + C[5] = 1 + 7 = 8.$

Quedando el arreglo C tal que $C[i]$ contiene el número de elementos menores o iguales a i, la Figura 6.2(b) presenta el contenido del arreglo C, al finalizar el for.

El cuarto for se desarrolla de la siguiente manera:

$B[C[A[8]]] = B[C[4]]] = B[7] \leftarrow 4 = A[8]; \quad C[A[8]] = C[4] \leftarrow 6 = C[4] - 1;$
$B[C[A[7]]] = B[C[1]]] = B[2] \leftarrow 1 = A[7]; \quad C[A[7]] = C[1] \leftarrow 1 = C[1] - 1;$
$B[C[A[6]]] = B[C[4]]] = B[6] \leftarrow 4 = A[6]; \quad C[A[6]] = C[4] \leftarrow 5 = C[4] - 1;$
$B[C[A[5]]] = B[C[3]]] = B[4] \leftarrow 3 = A[5]; \quad C[A[5]] = C[3] \leftarrow 3 = C[3] - 1;$
$B[C[A[4]]] = B[C[1]]] = B[1] \leftarrow 1 = A[4]; \quad C[A[4]] = C[1] \leftarrow 0 = C[1] - 1;$
$B[C[A[3]]] = B[C[4]]] = B[5] \leftarrow 4 = A[3]; \quad C[A[3]] = C[4] \leftarrow 4 = C[4] - 1;$
$B[C[A[2]]] = B[C[6]]] = B[8] \leftarrow 6 = A[2]; \quad C[A[2]] = C[6] \leftarrow 7 = C[6] - 1;$
$B[C[A[1]]] = B[C[3]]] = B[3] \leftarrow 3 = A[1]; \quad C[A[1]] = C[3] \leftarrow 2 = C[3] - 1.$

La siguiente figura muestra el arreglo B resultante,



6.2. Radix Sort

Esta técnica de ordenamiento no se basa en el modelo de cómputo de comparaciones. Resulta ideal para ordenar datos que pueden ser vistos como la composición de diferentes campos, como pueden ser las fechas, o bien para ordenar números de acuerdo a sus dígitos.

Estrategia

Este método es radicalmente diferente a los vistos anteriormente. Usa un arreglo para indexar en vez de comparar las llaves y así distinguirlas. Se asume que las llaves están formadas de d dígitos en base r, éste último es llamado el $r$adical (*radix*). Se considera que el primer dígito, de izquierda a derecha, es el más significativo.

Por ejemplo, si las llaves son cadenas, de letras mínusculas, de longitud seis, entonces $d = 6$ y $r = 26$. En otro ejemplo, se podrían tener llaves enteras en el intervalo $[0, 999]$, aquí tenemos que $d = 3$ y $r = 10$.

Cualquier dígito d_i de una llave satisface que $0 \leq d_i \leq r - 1$, por lo cual podemos usar un arreglo auxiliar de r listas para indexar las llaves.

La idea básica del Radix Sorting es ir iterando de acuerdo a los dígitos de la entrada, poniendo cada dato de entrada al final de la d_i-lista, donde d_i es el $i-$ésimo dígito de la llave. Esto es llamado propagación sobre el $i-$ésimo dígito.

La manera más natural de ordenar usando propagaciones es empezar con un ordenamiento sobre el primer dígito, el más significativo. Las $r-$sublistas resultantes son, entonces, ordenadas recursivamente (empezando con una propagación sobre el segundo dígito) y su concatenación será el resultado final. Este algoritmo es llamado MSD[1]-Radix Sort. Se puede notar que la entrada sufre una fragmentación en muchas sub-listas pequeñas.

Una estrategia menos obvia es la llamada LSD[2]-Radix Sort. Ésta inicia con una propagación sobre el último dígito, el menos significativo.

Para ilustrar este último algoritmo, considere la lista $\mathcal{L}$ a ordenar.

$$\mathcal{L} = \{179, 208, 306, 093, 859, 984, 055, 009, 271, 033\}.$$

El método inicia con una propagación sobre el tercer dígito, el proceso es mostrado en la Figura 6.3. El 179, es insertado en la posición 9, el 208 es colocado en la 8 y así se continua, hasta insertar el 033 que va a la posición 3.

Entonces, las listas son concatenadas, obteniéndose:

$$\mathcal{L}' = \{271, 093, 033, 984, 055, 306, 208, 179, 859, 009\}$$

Los datos están ahora en orden, si sólo consideramos el último dígito de cada llave.

Una segunda propagación/concatenación nos da el orden sobre el segundo dígito, esto es ilustrado en la Figura 6.4. Ahora el primer dato es el 271 y es colocado en la posición 7, luego el 093 es puesto en el lugar 9, al llegar el 033 se coloca en la posición 3.

[1] Most Significant Digit
[2] Least Significant Digit

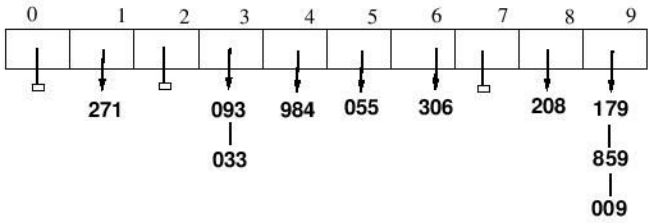


Figura 6.3: Radix Sorting, propagación sobre el último dígito.

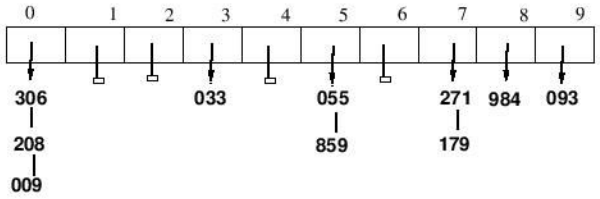


Figura 6.4: Radix Sorting, propagación sobre el segundo dígito.

Al concatenar las listas se obtiene:

$$\mathcal{L}'' = \{306, 208, 009, 033, 055, 859, 271, 179, 984, 093\}$$

Nótese como el orden correcto para los dígitos finales no es alterado; ellos están en orden en cada sublista, ya que las entradas son colocadas al final de cada sublista durante la propagación.

Ejecutando de esta manera, la propagación/concatenación final sobre el dígito más significativo completa el ordenamiento:

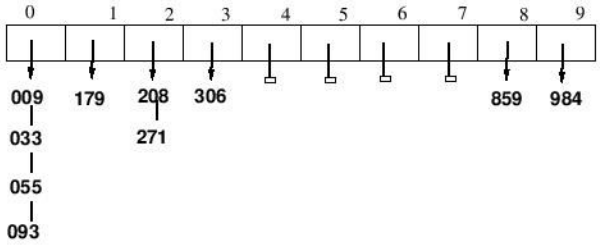


Figura 6.5: Radix Sorting, propagación sobre el primer dígito.

En la Tabla 6.1 se muestra esquemáticamente cómo se va modificando la lista de entrada durante la ejecución del algoritmo.

3^{er}		2^{o}		1^{er}
27**1**		3**0**6		**0**09
09**3**		2**0**8		**0**33
03**3**		0**0**9		**0**55
98**4**		0**3**3		**0**93
05**5**		0**5**5		**1**79
30**6**		8**5**9		**2**08
20**8**		2**7**1		**2**71
17**9**		1**7**6		**3**06
85**9**		9**8**4		**8**59
00**9**		0**9**3		**9**84

Cuadro 6.1: Comportamiento del Radix Sort sobre $\mathcal{L}$

Análisis de Complejidad

El cálculo del desempeño computacional se basa en el siguiente invariante:

"La secuencia $\mathcal{L}$ está ordenada si sólo consideramos los dígitos de i a d"

La complejidad de LSD-Radix Sort es una función que depende de n, d y r.

La operación característica para la fase de propagación es el movimiento de una entrada de la lista maestra a una sub-lista. Esto ocurre n veces por propagación.

La operación clave para la fase de concatenación es la apertura de una sublista a una nueva lista maestra. Esto sucede r veces por concatenación.

Por lo tanto, el costo total de una fase de propagación/concatenación es $O(n + r)$ ya que todo el proceso de ordenamiento realiza d de estas fases, la complejidad total es de $O(d(n + r))$. Para d y r fijas, LSD-Radix Sort es $O(n)$; es decir, lineal. Nótese sin embargo, que si las n llaves son diferentes, entonces $d \geq \log_r n$. Esto sucede pues dados d dígitos en base r, es posible representarlos en a lo más $n = r^d$ llaves distintas. Así que n se incrementa cuando r aumenta, si $O(n)$ se mantiene. Otro problema es que $O(d \cdot r)$ es independiente de n: si n es pequeña, se gastará tiempo en concatenar listas vacías.

Algoritmo

El Listado 27 presenta el pseudocódigo para el LSD-Radix Sort, supone que lista de n datos está contenida en una cola L y que los datos son cadenas de d dígitos en base r.

Se crea un arreglo auxiliar sL, de r localidades (primer for) para realizar las propagaciones sobre el $i-$ésimo dígito (segundo for); después se concatenan las sublistas (tercer for). Finalmente, regresa la lista ordenada.

Listado 27 Pseudocódigo para LSD-Radix Sorting

```
//PreC: S secuencia no vacia contenida en el arreglo A de n elementos
//Post: El arreglo B contiene a la secuencia ordenada

 Queue  RadixSort(Queue L; int n; int r,d) {
  int  i, j;                          // indices auxiliares
  array of Queue sL[0..r-1];          // sublistas

   for ( j=0; i<=r-1; i++)            // Crea el arreglo auxiliar sL
      sL[j].Create ;

   for (i=d; i<=d; j--){              // propagacion sobre el i-esimo
      while (not L.Empty) do {        //  digito
          p = L.Delete;               // toma un elemento en la cola
          v = p.getValue;             // recupera su valor
        dig = v.key[i];               // toma el i-esimo digito
        sL[dig].Insert(p);            // inserta el dato dig-esima sub-lista
      }// end while

     for (j=0 ;i<=r-1; i++){          // concatena las sublistas.
        L = L.Append (sL[j]);         // pega la j-esima sublista a L
        sL[j].Create                  // re-inicializa la j-esima sublista.
     } // end for j
   }// end for i

   return L                           // Lista ordenada
} // end Radix Sort
```

6.3. Bucket Sort

Este método resulta ser uno de los más sencillos métodos de ordenamiento dígital. Presentaremos dos versiones del algoritmo. Para obtener esta complejidad es necesario hacer algunas suposiciones, tales como:

1. la secuencia de datos es obtenida bajo una distribución uniforme sobre el intervalo $[0, 1)$;
2. el rango en que se encuentra la secuencia de n datos está bien determinado y es conocido;
3. el intervalo $[0, 1)$ es dividido en n sub-intervalos del mismo tamaño, llamados *buckets*[3];
4. los n datos son distribuidos en los buckets.

Estrategia

La estrategia del Bucket Sort se puede resumir en los siguientes pasos:

1. Crear tantos buckets como elementos hay en la secuencia a ordenar.
2. Asignar a cada elemento en la secuencia un bucket.
3. Ordenar cada uno de los buckets.
4. Concatenar los buckets.

Al describir la estrategia se uso el término *bucket* pero no establecimos qué entendemos por éste, así que a continuación estableceremos este concepto. Un bucket es una estructura de datos en la cual es posible guardar uno o varios elementos y, además, es posible distinguirlo de otros buckets.

Para asignar un bucket a un elemento, se procede a multiplicar al elemento por el número de buckets creados, al resultado de esta operación se le divide por el mayor elemento en la secuencia y de este último resultado se toma la parte entera, la cual indica el bucket asignado el elemento. Es decir, al elemento i le corresponde el bucket $\lfloor i\,(n/m) \rfloor$ donde n es el número de buckets formados y m es el mayor elemento en la secuencia.

En cuanto a la ordenación de cada bucket ésta puede ser realizada por cualquiera de los métodos que se han estudiado, incluso se puede efectuar recursivamente el bucket sort.

Finalmente, entendemos por concatenar buckets a la operación de unir todos los buckets de tal manera que estos puedan ser vistos como la secuencia ordenada.

Para ilustrar el proceso, presentamos el ejemplo de Figura 6.6. Se desea ordenar a la secuencia $S = \{78, 17, 39, 26, 72, 94, 21, 12, 23, 66\}$.

En el inciso **(a)**, del ejemplo, se observan a los diez elementos de la secuencia en los buckets que les fuerón asignados antes de realizar el ordenamiento en cada bucket; mientras que en **(b)** se muestran ya ordenados. Al concatenar los buckets obtenemos la secuencia ordenada $S = \{12, 17, 21, 23, 26, 39, 68, 72, 78, 94\}$.

[3]Por comodidad usaremos el término en inglés *bucket* en lugar del término en español *cubeta* y lo interpretaremos como un lugar donde se almacenan varios datos

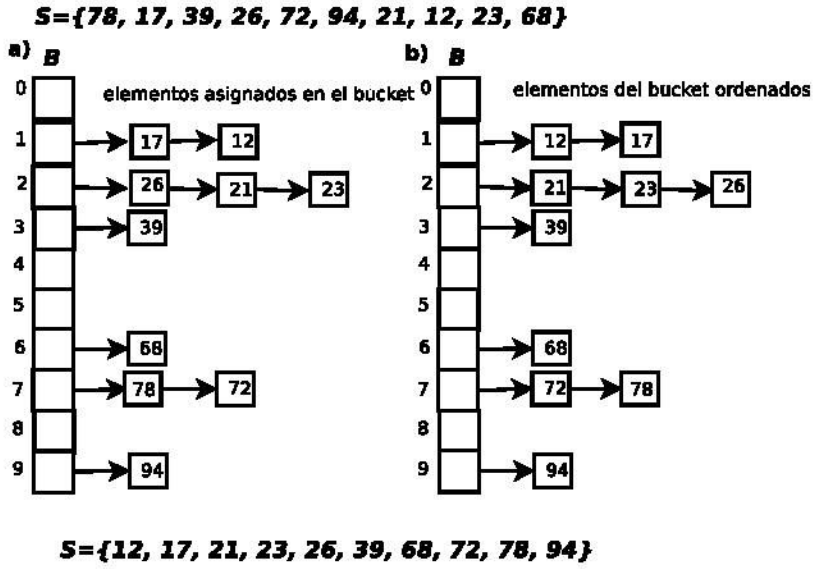


Figura 6.6: Ejemplo de Bucket Sort

Análisis de Complejidad

Para calcular la complejidad de este método consideramos los pasos que sigue la estrategia. Como se puede apreciar, crear buckets, asignar a cada elemento un bucket y concatenar los buckets son procedimientos que se llevan a cabo con un desempeño lineal. El proceso que determina la complejidad es el que ordena cada bucket y dado que, en general, esta operación es realizada por insertion sort, se llevará acabo el análisis considerando a este método de ordenamiento.

Sea n_i la variable aleatoria que denota al número de elementos en el bucket $B[i]$, sabemos que el tiempo que toma insertion sort para ordenar depende del número de comparaciones que realiza y que éste es n^2, entonces el tiempo que se espera consumir para ordenar $B[i]$ es $E[O(n_i^2)] = O(E[n_i^2])$. Por lo cual, el total de tiempo empleado para ordenar todos los buckets es de:

$$\sum_{i=0}^{n-1} O\left(E\left[n_i^2\right]\right) = O\left(\sum_{i=0}^{n-1} E\left[n_i^2\right]\right).$$

Para evaluar esta suma, debemos determinar la distribución de cada variable aleatoria n_i. Tenemos n elementos en n buckets. La probabilidad de que un elemento caiga en el bucket $B[i]$ es $1/n$, pues cada bucket es responsable de $1/n$ del intervalo $[0, 1)$. Esta situación es similar a realizar un experimento en el cual se tienen n urnas y se lanzan pelotas de manera independiente, se obtiene que la probabilidad de que una pelota caiga en una urna específica es de $1/n$. Ahora, si en lugar de urnas consideramos buckets y en vez de pelotas los elementos de la lista, podemos decir que la asignación de un elemento a un bucket sigue una distribución binomial con parámetros $binomial(n_i; n, p)$, $p = 1/n$.

Entonces la esperanza $E[n_i] = n \cdot p = 1$ y la varianza es $var[n_i] = n \cdot p \cdot (1-p)$. Por otro lado, sabemos que para toda variable aleatoria x se cumple:

$$var\,(x) = E\left[x^2\right] - E^2\,[x] \Rightarrow E\left[x^2\right] = var\,(x) + E^2\,[x].$$

Entonces, para este caso se cumple que: $E\left[n_i^2\right] = n \cdot p\,(1-p) + (np)^2$.

Como $p = 1/n$, $E\left[n_i^2\right] = 1 - \left(\frac{1}{n}\right) + 1^2 = 2 - \left(\frac{1}{n}\right)$.

Si n es grande entonces $(1/n)$ se hace cero, entonces $E[n_i^2]$ es constante y tiene un desempeño de $O(1)$, por ello podemos decir que:

$$\sum_{i=0}^{n-1} O\left(E\left[n_i^2\right]\right) = \sum_{i=0}^{i=n-1} 1 = n-1.$$

Con lo cual tenemos que el ordenamiento de todos los buckets tiene complejidad $O(n)$ y dado que los demás procesos tienen desempeño lineal, podemos concluir, que el desempeño computacional de bucket sort es de $O(n)$.

Algoritmo

El pseudocódigo para bucket sort supone que la secuencia no es vacía, consta de números enteros y está contenida en un arreglo.

Listado 28 pseudocódigo Bucket Sort

```
//PreC: A un arreglo que contiene a una secuencia S con n elementos no
//necesariamente ordenada.
//PostC: A esta en orden ascendente.

bucketSort(array A; int n; int max){
  int i ; array B;

  B=array.create(n);                    // crea los buckets

  for  (i=0;i< n;i++)                   // inserta a A[i] en el bucket
      B[n*A[i]].Insert(A[i]);           // correspondiente

  for  (i=0; i< n; i++);                // ordena cada bucket con
      B[i].I_Sort;                      //   Insertion Sort
                                        // concatena, en orden, las listas
  A = (B[0]+B[1]+  ... +B[n-1]);        // y las deja en A
  return A                              // regresa el arreglo A ordenado
end bucketSort
```

En el Listado 28 se observa con claridad lo descrito en la estrategia de la técnica, así mismo se facilita el distinguir el desempeño lineal que se mostró en el análisis del método.

Hay que señalar que en este código los buckets son representados con un arreglo de listas ya que esto permite la concatenación de una manera sencilla; además es muy fácil identificar los buckets.

Segunda Versión

Se aplica si los datos a ordenar son pequeños enteros no negativos, los cuales pueden ser usados como índices. En otras palabras, el tamaño del universo de las llaves debe ser fijado, para así representar al conjunto de llaves en un vector de bits.

Para ordenar una arreglo $A[0..n-1]$ de números diferentes describimos el universo $U = \{0, 1, 2, ..., N-1\}$ y creamos un vector de bits $B[0..N-1]$ que represente al conjunto de números en A. Iniciamos, asignando a cada localidad de B el valor de cero, después asignamos a $B[A[0]], B[A[1]], B[A[2]], ..., B[A[n-1]]$ el valor de 1. Los números en A han cumplido ahora su propósito; el resto del proceso reconstruye esos números de izquierda a derecha en A en ese orden. Esto se realiza recorriendo el vector de bits B de izquierda a derecha, cada vez que encontremos una localidad con un 1, insertamos su índice en la siguiente posición en A.

Si esta representación simple del vector de bits es usada, entonces Bucket Sort toma tiempo $O(N)$ en asignar valores iniciales a B; toma $O(n)$ insertar los elementos en A; y $O(N)$ en recorrer B. En total, Bucket Sort requiere tiempo $O(N)$.

Nótese que Bucket Sort es, en efecto, un algoritmo de tiempo lineal en la práctica, en el sentido teórico realmente no lo es. Es decir, si consideramos N y n arbitrariamente grande, entonces las llaves deben tener al menos $(\log N)$ bits, para que todas ellas sean distintas.

Si $(\log N)$ fuese suficientemente grande, entonces la tabla usará índices que no pueden ser considerados como una operación en tiempo constante, sino que debería costar $\Omega(\log N)$. Si las referencias de las tablas son consideradas con costo $\Omega(\log N)$ en vez de $O(1)$, entonces bucket sort se convierte en un algoritmo de orden $O(N \log N)$. Debemos considerar que Bucket sort toma tiempo lineal sólo porque la tabla indexada toma tiempo constante para arreglos de tamaño práctico.

Si los números en A no son necesariamente distintos, entonces podemos usar un método similar, pero debemos reemplazar el vector de bits por una tabla (arreglo) que indique el número de ocurrencias de la llave en A. Esto significa que en vez de tener el vector de bits $B[0..N-1]$, con valores 0 y 1; requerimos de un arreglo $C[0..N-1]$ cuyos elementos representan la frecuencia (ocurrencia) de la llave, los valores en C son enteros entre 0 y n. Para reconstruir la tabla A de estos contadores, necesitamos simplemente replicar A en cada índice i el número de veces dado en $C[i]$.

Ahora consideremos una generalización de Bucket Sort, para el caso en que A contiene registros o apuntadores a registros. Guardar los contadores del número de ocurrencias de una llave ya no es suficiente, pues A no podría ser reconstruida a partir de la enumeración de las llaves. En lugar del vector de bits B o del arreglo C, es necesario utilizar un arreglo de conjuntos $S[0..N-1]$, donde $S[i]$ contiene apuntadores a los registros con llave i. Por ejemplo S puede ser un arreglo de listas ligadas.

El conjunto $S[i]$ es llamado un bucket de datos con llave i. La primera fase del algoritmo consiste en recoger los miembros de A y echar a cada uno de ellos en el bucket apropiado. La segunda fase del algoritmo consiste en visitar los buckets en el orden de los índices para construir la versión ordenada de A.

Si A contiene apuntadores a registros, esta construcción puede ser hecha sobre el mismo arreglo en un paso; de otra manera, será necesario usar memoria auxiliar para construir el arreglo ordenado.