#93

SERIE
COMPUTACIÓN

María de la Luz Gasca Soto

Algoritmos sobre busqueda y ordenamiento (análisis y diseño)

AÑO
2004

VÍNCULOS
MATEMÁTICOS

Facultad de Ciencias

Vínculos matemáticos

Algoritmos sobre búsqueda y ordenamiento

(Análisis y diseño)

Dra. María de Luz Gasca Soto*

Nº 93. 2011

* Departamento de Matemáticas. Facultad de Ciencias, UNAM

Impreso en la Coordinación de Servicios Editoriales de la Facultad de Ciencias, UNAM

Algoritmos sobre Búsqueda y Ordenamiento (Análisis y Diseño)

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

18 de septiembre de 2011

Índice general

Prefacio

Estas notas recopilan material de algoritmos sobre búsquedas y ordenamientos, temas que se estudian en cursos como Análisis de Algoritmos y Estructuras de datos; asignaturas asociadas a la Licenciatura en Ciencias de la Computación, de la Facultad de Ciencias, UNAM, donde se revisan estos temas con gran profundidad; las carreras de Actuaría y Matemáticas también cuentan con materias optativas que incluyen estos temas.

El objetivo principal es que estas notas formen parte parte del material didáctico para cursos básicos de Algoritmos y Estructuras de Datos.

Se revisa una gran variedad de algoritmos. Para cada algoritmo se presenta su descripción general (estrategia), ejemplos, pseudocódigo, análisis detallado en el peor de los casos, y para la mayoría de ellos, sobretodo para los de ordenamiento, se realiza el análisis del tiempo esperado.

Parte del material, de las presentes notas, fue elaborado en el marco de un proyecto de servicio social, donde el estudiante de la carrera de ciencias de la computación Alejandro Hernández Aguilar tuvo como tarea principal recopilar, y transcribir, material sobre Algoritmos de búsquedas y ordenamientos. He complementado este material con mis propias notas de clase, he ampliado algunos temas y, además, al hacer correcciones incluí interesantes modificaciones que completan el material.

El material está dividido en tres partes:
(I) Búsquedas; **(II)** Hashing; y **(III)** Ordenamientos.

En la primera parte se define el Problema de Búsqueda y se presentan cuatro algoritmos de que resuelven el problema. Para cada uno de ellos se describe la estrategia general y se ejemplifica; se calcula el desempeño computacional, en el peor de los casos y se da, al menos, un pseudo-código. También se presentan algunas variantes de una de las estrategias más populares, la Búsqueda Binaria.

En la segunda parte presentamos las Funciones Hash, usadas para realizar búsquedas eficientes sobre tablas. Se describen y diseñan varias funciones de dispersión; se presentan los conflictos de colisión y se analizan diversas estrategias para evitar y manipular las colisiones. Finalmente, se presenta el análisis de complejidad de las Tablas Hash.

Este material se complementa con el Apéndice B donde se describe brevemente el tipo de datos abstracto Tablas y se presentan estructuras de datos para las Tablas Hash.

En la tercera parte, se revisa una amplia gama de algoritmos que resuelven el problema de Ordenamiento, desde los más simples (no por ello mejores) a los más complicados pero con el mejor desempeño computacional. Finalmente, , para completar el tema. presentamos ordenamientos cuyo tiempo de ejecución es lineal. Estos aprovechan características especiales de los datos de entrada (ejemplares) para solucionar el problema.

Complementamos este material con varios apéndices, donde se revisa, brevemente, notación sobre el Análisis de Algoritmos, algunas propiedades de Árboles y algunas estructuras de datos como Colas de Prioridades y Heaps Binarios.

Dra. María de Luz Gasca Soto
luzg at ciencias.unam.mx
Profesor Asociado, C, T.C.
Departamento de Matemáticas

Capítulo 1

El Problema de Búsqueda

En este capítulo presentamos el problema de búsqueda, el cual consiste, de manera general, en encontrar un elemento x en una colección de objetos, parcialmente ordenada[1]. Considerando que siempre se realizan búsquedas sobre conjuntos, los cuales pueden estar o no ordenados, es necesario tener métodos que tomen en cuenta las características del problema que se desea resolver para así obtener procesos eficientes, ya que no es lo mismo efectuar búsquedas sobre conjuntos ordenados y desordenados. Además, también hay que considerar el tamaño del conjunto. Para mostrar esto, pensemos en los siguientes ejemplos:

1. Al pagar con tarjeta, de crédito o débito, la terminal tiene acceso a una base de datos y realiza una búsqueda para autentificar la tarjeta, es decir verifica que exista una cuenta asociada a tal tarjeta.

2. Buscar en el disco duro de una computadora un directorio o archivo.

3. La búsqueda que se realiza en algunas bibliotecas pequeñas para localizar los datos de un libro sobre ficheros ordenados alfabéticamente.

4. Al buscar un libro por tema en una biblioteca automatizada el sistema tendrá que filtrar la información de la base de datos que forma el catálogo para mostrar los libros que contienen el tema dado.

5. Al buscar, el ayudante, la tarea de un alumno en *montón* de tareas un curso, no es necesario que el conjunto de tareas esté organizado ni por número de tarea ni por nombre del alumno.

En general, la búsqueda localiza un objeto en un conjunto dado, la búsqueda se realiza considerando un atributo clave, es decir un atributo que lo identifica de los demás de forma única denominado llave.

Los métodos que describiremos en las siguientes secciones, están enfocados principalmente en búsquedas sobre secuencias ordenadas y se presenta un método para secuencias no necesariamente ordenadas.

[1]Es necesario que los elementos en la colección puedan ser comparados entre ellos.

Por simplicidad, supondremos que los conjuntos y secuencias usadas para describir e ilustrar los métodos son de tipo entero. Si los elementos no son enteros, siempre es posible construir una función biyectiva para asociar cada objeto del conjunto con un entero.

El resultado de la búsqueda dependerá de los requerimiento específicos del problema; por ejemplo, podría:

1. indicar si el dato está o no en el conjunto;
2. indicar en qué posición se encuentra el dato, si está, o en qué posición se encontraría, si no está en el conjunto;
3. regresar el primer dato, o al primero que se encuentre, que satisfaga las condiciones de la búsqueda;
4. regresar a todos los datos que satisfagan una propiedad dada.

Antes de iniciar con el tema daremos algúnos conceptos que usaremos, durante el desarrollo de este trabajo.

Objeto: Un objeto es la representación detallada, concreta y particular de un algo, que le permite ser identificado de otros. Para nuestro propósito a esta representación la llamaremos de forma indistinta "elemento u objeto".

Conjunto: Es una colección de objetos del mismo tipo, que no admite elementos repetidos. Consideraremos conjuntos para los cuales existe una relación de orden.

Secuencia: Es una sucesión de objetos del mismo tipo donde los objetos pueden o no estar repetidos, para los cuales existe una relación de orden.

Espacio de búsqueda: conjunto o secuencia de datos en la cual se realizará la búsqueda.

Un planteamiento más formal del problema es:

Problema de Búsqueda. Sea $S = \{s_1, s_2, s_3, \ldots, s_n\}$ una secuencia de números enteros, no necesariamente ordenada, no vacía, finita y de tamaño n.
Determinar se existe z en S, tal que $z = s_i$, para algún $i = 1, 2, 3 \ldots, n$.

A continuación presentamos cuatro técnicas empleadas para resolver este problema: La búsqueda secuencial, la binaria, la exponencial y la búsqueda por interpolación. Para cada una de ellas se presenta la estrategia general, el análisis de complejidad, en el peor de los casos (para algunos el análisis del caso esperado) y, al menos, un pseudocódigo, que podría ser recursivo o iterativo o ambos.

1.1. Búsqueda Secuencial

La Búsqueda secuencial es también llamada búsqueda lineal, es el método de búsqueda más intuitivo y simple, por lo que no se requiere ningún orden sobre los datos de entrada.

Estrategia

Esta técnica revisa uno a uno los elementos de la secuencia S hasta encontrar el dato deseado o llegar al último elemento de S, lo cual indicaría que el dato no está en S.

Para ilustrar el método consideremos los ejemplos de la Figura 1.1, ambos incisos se busca al número 7 y el índice i señala la posición que está siendo revisada.

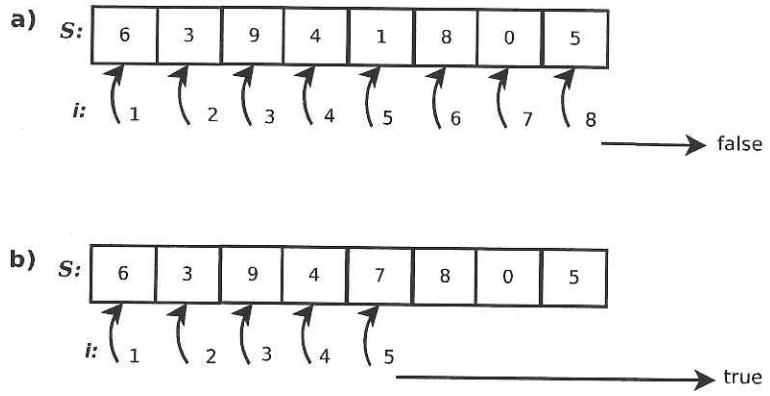


Figura 1.1: Ejemplos de búsqueda secuencial.

En la Figura 1.1(**a**) la secuencia es S={6,3,9,4,1,8,0,5}; aquí se tiene una búsqueda no exitosa, el proceso regresa false. La búsqueda termina cuando se llega al último elemento. Para el inciso (**b**) la secuencia es S={6,3,9,4,7,8,0,5}; ahora búsqueda es exitosa, el proceso regresa true, y termina al localizar el elemento en la posición $i = 5$ de S.

Análisis de Complejidad

En este apartado estableceremos el desempeño computacional tanto en el peor caso como en el caso esperado. Para tal propósito supondremos que la búsqueda se realiza sobre una secuencia S no necesariamente ordenada de tamaño n, donde $S = \{s_1, s_2, \ldots . s_n\}$.

Análisis del Peor Caso

Para establecer el desempeño computacional en el peor de los casos, es necesario que observemos que hay dos posibles escenarios; en el primero se tiene la situación de que el elemento buscado se encuentre al final de la secuencia, el segundo se da cuando el elemento no se encuentra en ella.

Para ambos casos se realizarán, necesariamente, n comparaciones por lo que el desempeño es exactamente el tamaño del ejemplar. Por tanto, en el peor caso la búsqueda secuencial requiere tiempo $O(n)$.

Análisis del Caso Esperado

En este caso también tenemos dos escenarios, en el primero la búsqueda siempre es exitosa, mientras que en el segundo consideramos que no lo es.

Para comenzar el análisis suponemos que la secuencia S ha sido obtenida bajo una distribución uniforme, entonces si el elemento sí está en S hay n localidades posibles para él, todas tienen la misma probabilidad de contenerlo, de esta forma la probabilidad de que el elemento buscado esté en la posición i, con $1 \leq i \leq n$, es de $1/n$. De manera más formal, sea x el elemento a buscar, entonces la probabilidad de que x esté en la posición i, denotado por $P_i[x]$, es: $P_i[x] = 1/n$.

Retomando la forma como el método resuelve el problema podemos observar que el número de comparaciones realizadas para ubicar al elemento, depende de su localización, por ello este número es simplemente su posición en la secuencia, de tal forma que si el elemento está en el primer lugar se realizará una comparación, si está en el segundo se efectuarán dos y así sucesivamente.

Entonces, podemos calcular el promedio de las comparaciones que esperamos realizar sumando el número de comparaciones efectuadas para cada localidad y multiplicando éstas por la probabilidad de que el elemento se encuentre en ella, esto es denotado por: $E\left[P_i\left[x\right]\right] = E\left[x\right]$, en específico:

$$E[x] = \sum_{i=1}^{n} i \cdot (P_i[x]) = \sum_{i=1}^{n} i \cdot \left(\frac{1}{n}\right) = \frac{1}{n} \cdot \sum_{i=1}^{n} i = \frac{1}{n} \cdot \frac{n(n+1)}{2} \in O(n).$$

Si incluimos la posibilidad de que el elemento no esté en la secuencia, tendremos ahora que considerar $(n+1)$ localidades por revisar y los $(n+1)$ posibles lugares son igualmente probables, entonces el número esperado de comparaciones ahora es:

$$E[x] = \frac{n+2}{2} = \frac{n+1}{2} + \frac{1}{2}.$$

Nótese que incluir la posibilidad de que el elemento pueda no estar en la secuencia sólo incrementa el caso esperado en $1/2$, cuando comparamos esta cantidad con el tamaño de la secuencia, esta cantidad es no significativa por lo que podemos concretarnos al valor $(n+1)/2$. Así, el número de comparaciones realizadas, en el caso esperado, por la búsqueda secuencial es lineal, es decir $O(n)$.

Este método de búsqueda no requiere ningún orden en los datos de entrada. Se puede usar en secuencias ordenadas, en este caso se recomienda modificar la condición de salida para terminar antes de recorrer toda la secuencia, aunque esto no influye en su desempeño global.

Algoritmo

A continuación se da el pseudocódigo para el algoritmo de búsqueda secuencial, presentado en el Listado 1. Suponemos que la secuencia S está contenida en un arreglo X, el cual no tiene datos repetidos.

Listado 1 Búsqueda Secuencial

```
// PreC: X es un arreglo no necesariamente ordenado, no vacio y finito de enteros
// PostC: X no sufre cambios;
//        regresa la posicion i, si el elemento z es encontrado;
//        regresa -1 si z no se encuentra en el conjunto.

int BSecuencial(int z; data_array  X) {

    int i;                          // indice de acceso al arreglo.
    i = 1;                          // inicializacion del indice.
    while (i < X.lenght) do {       // recorre el arreglo con el indice
         if (X[i]== z)
            return i;               // busqueda exitosa, regresa el indice
         else i=i+1
    }
    return -1;                      // busqueda no exitosa,
                                    // regresa una posicion no existente.
    }
```

Como se puede observar en el pseudocódigo se hace uso del índice i para recorrer el arreglo en el ciclo while, cuando la llave es encontrada (en caso de estar en el arreglo) se regresa i, la posición actual, lo que se interpreta como una búsqueda exitosa, ya que se obtiene la localidad en la que se encuentra el elemento buscado.

En caso de que se llegue al final del arreglo (condición en while) y no se encuentre el elemento, el algoritmo regresa -1, este valor fue escogido en caso de fracaso, pues estamos suponiendo que el índice inferior del arreglo es positivo.

1.2. Búsqueda Binaria

Este método pertenece al grupo de técnicas basadas en la estrategia de divide y vencerás. La Búsqueda Binaria requiere que el espacio de búsqueda esté ordenado. Así el nuevo planteamiento del problema es el siguiente:

Problema de Búsqueda. Sea $S = \{s_1, s_2, \ldots, s_n\}$ una secuencia ordenada y finita de números enteros, de tamaño n. Determinar si existe z en S tal que $z = s_i$, para algún i, $i = 1, 2, \ldots, n$.

Sin pérdida de generalidad. supondremos que la secuencia S está ordenada de forma ascendente y tiene longitud n_S. Denotaremos con m al elemento que divide a la secuencia en dos subsecuencias de tamaño similar y con z al elemento a buscar.

Estrategia

La estrategia empleada por esta técnica es más fácil de comprender si la vemos de manera recursiva, ésta consiste en dividir a la secuencia original en dos secuencias más pequeñas, de tamaño similar, donde el elemento que las divide m es comparado con z; en caso de ser igual, se considera una búsqueda exitosa y se da por concluida. En caso contrario, dependiendo de cómo sea la relación de orden entre el elemento m y el elemento buscado z, sin dejar de considerar, el tamaño de la secuencia S se tienen las siguientes situaciones:

Si $(m < z)$ **y** $(n_S > 1)$ se considera a la subsecuencia a la derecha del elemento m, como el nuevo espacio de búsqueda y se emplea nuevamente búsqueda binaria sobre esta subsecuencia.

Si $(m > z)$ **y** $(n_S > 1)$ se considera a la secuencia a la izquierda de m como el nuevo espacio de búsqueda y se aplica el método de la búsqueda binaria sobre ella.

Si ($(m < z)$ **y** $(n_S = 1)$) **o si** ($(m > z)$ **y** $(n_S = 1)$) se considera una búsqueda no exitosa y se da por terminada.

Para ilustrar el método, consideremos el ejemplo mostrado en la Figura 1.2(a). Se tiene la secuencia $S = \{2, 3, 5, 7, 9, 10, 13, 15, 16, 18\}$ y se busca al número $z = 13$.

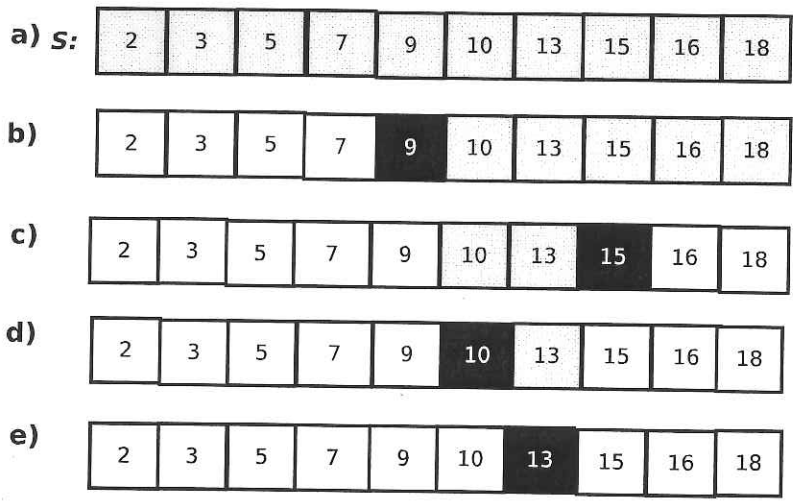


Figura 1.2: Ejemplo de búsqueda binaria

Como podemos ver en el inciso **(b)**, de la Figura 1.2, se tiene que el número de datos es $n_S = 10$; el elemento en la mitad, es $m = 9$ y está enmarcado con color negro; $S_1 = \{2, 3, 5, 7\}$ es la subsecuencia izquierda de m y $S_2 = \{10, 13, 15, 16, 18\}$ es la subsecuencia derecha. Ya que $z = 13 > 9 = m$ se toma como nuevo espacio de búsqueda a S_2, marcada en gris. Aplicamos la búsqueda binaria sobre ella, obteniendo $m = 15$, $n_S = 5$. Las nuevas

secuencias izquierda y derecha son $S_1 = \{10, 13\}$ y $S_2 = \{16, 18\}$, respectivamente, inciso **(c)**. Se tiene que $m = 15 \neq 13 = z$ y $z = 13 < 15 = m$, entonces se toma a la secuencia izquierda (marcada en gris) y se le aplica búsqueda binaria. Ahora, $m = 10$ y $n_S = 2$, dado que el espacio de búsqueda tiene solo dos elementos, se considera $S_1 = \{10\}$ y $S_2 = \{13\}$, como $m = 10 < 13 = z$, se aplica la búsqueda binaria sobre S_2, inciso **(d)**. Tenemos que $m = 13$, $n_S = 1$, ya que $m = 13$, es el número buscado, entonces se termina la búsqueda binaria con éxito.

Algo que es importante observar, con la misma secuencia S, es que si el número buscado hubiera sido el 12, se realizarían los mismos pasos pero al final como $m = 13 \neq 12 = z$ y $n_S = 1$ se terminaría con una búsqueda no exitosa, lo cual nos indica que al menos para este caso se realizán el mismo número de comparaciones para la búsqueda exitosa y la no exitosa, esto nos proporciona una pista del desempeño computacional de este algoritmo.

Complejidad Computacional

Calcularemos el desempeño computacional de este método de búsqueda tanto en el peor caso como en el caso promedio. Sin pérdida de generalidad y para facilitar el análisis consideraremos que S es de tamaño 2^k para $k \in \mathbb{Z}$ y $k \geq 1$.

Análisis del peor caso

Dado que el método siempre reduce la secuencia a la mitad y que el tamaño de la secuencia es: $n = 2^k$, entonces para cada llamada recursiva el tamaño de la secuencia revisada se puede expresar como $n' = 2^{k-1}$, de aquí podemos decir que el tamaño mínimo se da cuando $k = 1$. Por tanto, el método es usado k veces, entonces al resolver la ecuación $n = 2^k$ tenemos que $k = \log_2 n$, por lo que se realizan $\log_2 n$ comparaciones. Por lo tanto, la búsqueda binaria toma tiempo $O(\log_2 n)$, en el peor caso.

Análisis del Caso Esperado

Para realizar el análisis del caso esperado se requiere suponer que la secuencia fue obtenida bajo una distribución uniforme. Se revisan dos posibles escenarios: cuando el elemento a localizar está en la secuencia y cuando el elemento no está.

En el primer escenario se tienen n posibles posiciones para z, número a buscar, cada posición tiene la misma probabilidad de contener a z y ésta es de $1/n$, lo que denotaremos por: $P_i[z] = 1/n$. Si consideramos el árbol binario que representa a este proceso de búsqueda vemos que se requiere una comparación para encontrar el elemento raíz, que en el caso de la búsqueda es el elemento en la mitad, m, dos comparaciones para cualquier nodo del primer nivel y así sucesivamente; en general, se requieren i comparaciones para cualquier nodo en el nivel i. Sabemos además que un árbol binario tiene 2^{i-1} nodos en el nivel i, entonces cuando $n = 2^k$ se tiene un árbol con k niveles.

Entonces, podemos determinar el número de comparaciones hechas para cada posible localidad, esto lo conseguimos considerando el nivel en el que se encuentra el árbol binario que representa la búsqueda. Una vez que conocemos este dato, es posible calcular el

número promedio de comparaciones que se espera realizar para cada posición y al multiplicar esta cantidad por la probabilidad de que el elemento esté en ella y sumarlas todas obtenemos el análisis del caso esperado. Esto lo podemos escribir como:

$$E[P_i[z]] = E[z] = \frac{1}{n} \cdot \sum_{i=1}^{k} i \cdot (2^{i-1}) = \left(\frac{1}{n}\right) \cdot \frac{1}{2} \sum_{i=1}^{k} i \cdot (2^i)$$

$$= \frac{1}{n} \cdot \left(\frac{1}{2}\right) \cdot \left[(k-1)\, 2^{k+1} + 2\right] = \frac{1}{n} \cdot \left[(k-1)\, 2^k + 1\right]$$

$$= \frac{1}{n} \cdot \left[k\left(2^k\right) - 2^k + 1\right] = \frac{\left[k\left(2^k\right) - \left(2^k - 1\right)\right]}{n}$$

$$= \frac{k\left(2^k\right) - n}{n} = \frac{k\left(2^k\right)}{n} - 1 = \frac{k \cdot n}{n} - 1 = k - 1.$$

Ya que $n = 2^k$ entonces $k - 1 \approx \log(n) - 1$, por lo tanto $E[z]$ es $O(\log n)$.

Ahora bien, si consideramos el segundo escenario en el cual el número buscado no está en la secuencia, seguimos teniendo las n posibilidades del escenario anterior, pero además hay que agregar la $(n+1)$ generada a partir del hecho de que el elemento no está, las cuales son:

$$\begin{array}{l} z < s_1. \\ z > s_1, \text{ pero } z < s_2. \\ \vdots \\ z > s_{n-1}, \text{ pero } z < s_n. \\ z > s_n. \end{array}$$

Así, para este caso se tiene en total $(2n+1)$ posibilidades, las cuales son equiprobables, por lo tanto, la probabilidad de que el elemento esté en una posición es $P_i[z] = 1/(2n+1)$. Para calcular el número de comparaciones esperadas hay que tomar en cuenta que para cada una de las $(n+1)$ nuevas localidades que agregamos se realizan k comparaciones, por tanto tenemos $(n+1)k$ comparaciones adicionales. Al poner estos datos juntos obtenemos que el número esperado de comparaciones a realizar, formalmente:

$$E[P_i] = E[z] = \frac{1}{2n+1}\left[\left(\sum_{i=1}^{k} i\left(2^{i-1}\right)\right) + (n+1)k\right]$$

$$= \left(\frac{1}{2n+1}\left(\frac{1}{2}\right)\sum_{i=1}^{k} i\left(2^i\right)\right) + \frac{(n+1)\,k}{2n+1}$$

$$= \left(\frac{1}{2n+1}\left(\frac{1}{2}\right)\left[(k-1)\, 2^{k+1} + 2\right]\right) + \frac{(n+1)\,k}{2n+1}.$$

Distribuyendo la suma y sustituyendo el valor de $n = 2^k$,

$$E[z] = \frac{[(k-1)2^k+1] + (n+1)k}{2n+1} = \frac{[(k-1)2^k+1] + (2^k+1)k}{2(2^k)+1}$$

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

$$\approx \frac{k(2^{k+1}) - 2^k + k}{2^{k+1}} \approx \frac{k(2^{k+1}) - 2^k}{2^{k+1}} \approx k - \frac{1}{2}$$

Ya que $n = 2^k$ entonces $k - \frac{1}{2} \approx \log n - \frac{1}{2}$. Por lo tanto, la búsqueda binaria tiene desempeño computacional de orden $O(\log_2 n)$, en el caso esperado.

Algoritmo de Búsqueda Binaria

En esta subsección daremos el pseudocódigo para el algoritmo de búsqueda binaria, por cuestiones de simplicidad supondremos que los datos están contenidos en un arreglo y que no hay datos repetidos. Presentaremos dos versiones, una iterativa, Listado 2, y otra recursiva, Listado 3.

Listado 2 Búsqueda Binaria Iterativa

```
//PreC: X[a,b] es un arreglo ordenado, no vacio y finito de enteros
//PostC: X no sufre cambios;
//       regresa la posicion i, si el elemento z es encontrado;
//       regresa -1 si z no se encuentra en el conjunto.

int BBinaria(int z; data_array X){
    int max, mitad, min                          // variables temporales
       min = a;    max = b;
       while ( (max - min) > 1 ) do {
            mitad = (max + min) div 2;           // calcula mitad
            if (z < X[mitad])                    // busca a la izquierda
              then max = mitad;
            else
              min = mitad;                       // busca a la derecha
       }
       if (X[mitad] == z)
         return mitad;                           // busqueda exitosa
       if (X[min] == z)
         return min;                             // exito
       else
         return -1;                              // sin exito
}//end BBinaria.
```

A pesar de que la explicación de la estrategia se realizó recursivamente, este pseudocódigo se implementa de manera iterativa para mostrar cómo se ve la técnica con otro enfoque, en este caso el ciclo while es el equivalente a determinar el tamaño de la secuencia a emplear, con el garantizamos que la última secuencia a usar es de un elemento.

Como también se puede apreciar, la selección de la subsecuencia a emplear se determina al actualizar los valores ya sea para max, índice superior del subarreglo, o para min, índice inferior del arreglo actual. El valor de -1 se escoge como valor a regresar en caso de búsqueda sea no exitosa, debido a que consideramos a los índices del arreglo son estrictamente positivos y el método regresa la posición donde se encuentra el elemento si se encuentra en el arreglo.

Listado 3 Búsqueda Binaria Recursiva

```
//PreC: X[a,b]  es un arreglo  ordenado, no vacio y finito de enteros
//       a <= b+1  el arreglo tiene al menos un elemento
//PostC: X no sufre cambios;
//       regresa true, si el elemento z es encontrado;
//       regresa false, si z no esta en el conjunto.
//       Encontro = (z esta_en X)

// Proceso principal
 BusquedaB(array X; int a,b; int z){
     Encontro = BB (X, a, b, z);
 }

// Proceso auxiliar
boolean BB(array X; int a,b; int z) {
    int  mid;                                  // posicion mitad

     if ( a > b )                              // arreglo vacio
       return false;                           // sin exito
       else
         mid := (a+b) div 2;                   // calcula la mitad
           if ( z = X[mid])
              return true;                     // elemento encontrado
           else if (z < X[mid])                // busca a la izquierda
              return BB(X, a, mid-1, z);
            else                               // busca a la derecha
               return BB(X, mid+1, b, z);
}
// end BB
```

1.3. Aplicaciones de Búsqueda Binaria

En esta sección revisaremos tres problemas que se resuelven aplicando la estrategia de la Búsqueda Binaria.

Secuencia cíclica

Una secuencia $S = x_1, x_2, ..., x_n$ se dice que está cíclicamente ordenada si el número más pequeño en la secuencia es el elemento x_i para algún i desconocido y la secuencia S' está ordenada en orden creciente, con $S' = x_i, x_{i+1}, ...x_n, x_1, x_2, ..., x_{i-1}$.

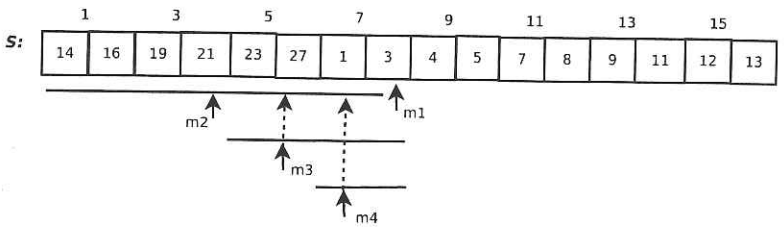
Por ejemplo, $S = \{14, 16, 19, 21, 23, 27, 1, 3, 4, 5, 7, 8, 9, 11, 12, 13\}$ es una secuencia cíclicamente ordenada, con $i = 7$.

Problema : Dada una lista cíclicamente ordenada, encontrar la posición del mínimo elemento en la lista. Supondremos, por simplicidad, que tal posición es única.

Estrategia

Para encontrar el mínimo elemento x_i en la secuencia, usamos la estrategia de la búsqueda binaria para eliminar la mitad de la secuencia en una comparación. Tomamos cualesquiera dos números x_k y x_m tales que $k < m$. Si $x_k < x_m$ entonces el índice i no puede estar en el rango $k < i \leq m$, ya que x_i es el mínimo en la secuencia. Nótese que no podemos excluir x_k. Por otro lado, si $x_k > x_m$ entonces el índice i debe estar en el rango $k < i \leq m$ ya que el orden fue cambiado en algún lugar de ese rango. De esta forma, con una comparación podemos eliminar varios elementos. Eligiendo k y m apropiadamente es posible encontrar al índice i en tiempo $O(\log n)$ comparaciones. El algoritmo está dado en el Listado 4.

Ejemplo



Consideremos la secuencia $S = \{14, 16, 19, 21, 23, 27, 1, 3, 4, 5, 7, 8, 9, 11, 12, 13\}$.
Aplicamos Encuentra_Indice(1,16). Tenemos que $mitad = 8$ y $X[8] = 3 < 13 = X[16]$.
Ejecutamos Encuentra_Indice(1,8). Ahora, $mitad = 4$ y $X[4] = 21 > 3 = X[8]$.
Llamamos a Encuentra_Indice(5,8). Tenemos $mitad = 6$ y $X[6] = 27 > 3 = X[8]$.
Aplicamos Encuentra_Indice(7,8). Tenemos que $mitad = 7$ y $X[7] = 1 < 3 = X[8]$.
Ejecutamos Encuentra_Indice(7,7). Ahora, i_izq= i_der y el algoritmo regresa al 7.

Listado 4 Búsqueda Binaria Cíclica

```
//PreC: X[1,n]  es una secuencia ciclica, no vacia y finita de enteros
//PostC: X no sufre cambios;
//       regresa la posicion p, indice del menor elemento

BBC (array X; int n){
   p = EncuentraIndice(1, n);
   return p
}

int EncuentraIndice(int i_izq, i_der ) {
  if  ( i_izq == i_der  )        return   i_izq;
     else{
        mitad = ( i_izq + i_der ) div 2;
        if    ( X[mitad] < X[i_der] )
           Encuentra_Indice ( i_izq, mitad );
        else
           Encuentra_Indice ( mitad+1, i_der )
     }
}//end Encuentra...
```

Índice Especial

En el siguiente problema de búsqueda la llave no está dada; en vez de ello buscamos un índice que satisfaga una propiedad especial.

Problema : Dada una secuencia ordenada de enteros distintos $a_1, a_2, ..., a_n$, determinar si existe un índice i tal que $a_i = i$.

Por ejemplo, para $S = \{-5, -4, -2, 0, 1, 3, 5, 7, 8, 9, 11, 13, 14, 15, 16\}$, $i = 11$.

Estrategia

La búsqueda binaria tradicional no es aplicable aquí, ya que el valor del elemento buscado no está dado. Sin embargo, la propiedad que buscamos es adaptable al principio de la búsqueda binaria. Para facilitar los cálculos, asumiremos que n, el tamaño del arreglo, es un número par.

Considere el valor de elemento en la posición $n/2, a_{n/2}$. Si este valor es exactamente $n/2$, hemos encontrado el índice deseado. En otro caso, si el valor es menor que $n/2$ entonces todos los números son distintos, el valor de $a_{n/2-1}$ es menor que $n/2 - 1$ y así. Ningún número en la mitad de la secuencia puede satisfacer la propiedad, pero podemos seguir buscando en la segunda mitad. Para la respuesta a "más grande que"se satisface un argumento similar. El algoritmo es dado en el Listado 5.

Listado 5 Búsqueda Binaria Índice Especial

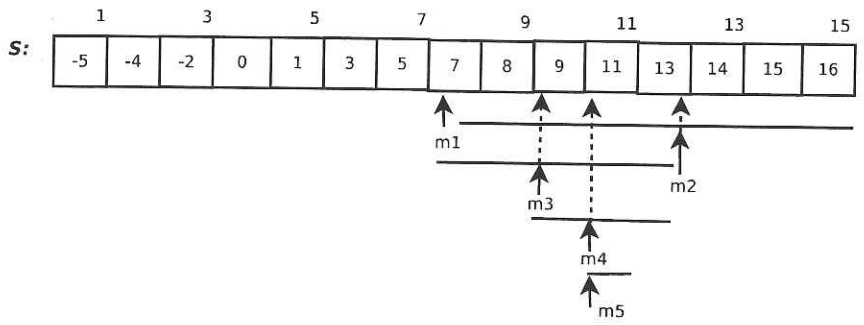
```
//PreC: A[1,n]  es un arreglo  ordenado, no vacio y finito de enteros.
//PostC: A no sufre cambios;
//       regresa la posicion p tal que A[p]=p


BB_IE(array X; int n){
   p = BusquedaEspecial(1, n);
   return p;
}

int Busqueda_Especial (int i_izq, i_der) {
  if   ( i_izq == i_der  )
     if   ( A[i_izq] = i_izq )  return  i_izq;
       else   return 0                              // busqueda  no exitosa
  else{                                             // continua la busqueda
        mitad = (i_izq + i_der) div 2;
        if  ( A [mitad] < mitad  )
           Busqueda_Especial ( mitad + 1, i_der )
        else
            Busqueda_Especial ( i_izq, mitad )
   } // end else
}// end Busqueda_E
```

Ejemplo

Sea secuencia $S = \{-5, -4, -2, 0, 1, 3, 5, 7, 8, 9, 11, 13, 14, 15, 16\}$. Aplicamos el proceso IndiceEspecial(1,15); es decir, nos quedamos con la parte izquierda, tenemos mitad=8, comparamos $A[8] = 7 < 8$ =mitad, entonces llamamos a IndiceEspecial(9,15); nótese que nos quedamos con la parte derecha. Ahora, mitad=12 y ($A[12] = 13 \not< 12$ =mitad), entonces ejecutamos IndiceEspecial(9,12). Ahora, mitad=10 y ($A[10] = 9 < 10$ =mitad), entonces aplicamos IndiceEspecial(11,12). Tenemos, mitad=11, comparamos $A[11] = 1 \not< 11$ =mitad, entonces llamamos a IndiceEspecial(11,11), como (i_izq = i_der) y (A[i_izq]=i_izq), el proceso regresa al índice i_izq= 11 y termina. La siguiente figura muestra esquemáticamente este ejemplo.



Secuencias de tamaño desconocido

Algunas veces usamos un procedimiento como la búsqueda binaria para duplicar el espacio de búsqueda en vez de partirlo a la mitad.

Considere un problema normal de búsqueda, pero suponga que el tamaño de la secuencia no es conocido. No podemos dividir a la mitad la secuencia, pues no conocemos sus cotas ni su tamaño. Buscaremos entonces un elemento x_i mayor o igual que z, el elemento buscado. Si encontramos tal elemento, podemos aplicar búsqueda binaria en el rango de 1 a i. Primero comparamos z con x_i. Si $z \leq x_i$ entonces z puede solo ser igual a x_i. Asumimos por inducción que sabemos que $z > x_j$ para alguna $j \geq 1$. Si comparamos a z con x_{2j}, entonces estamos duplicando el espacio de búsqueda con una sola comparación. Si $z \leq x_{2j}$, entonces sabemos que $x_j < z \leq x_{2j}$ y podemos encontrar z en $O(\log j)$ comparaciones adicionales. Además, si i es el menor índice tal que $z \leq x_i$ entonces toma $O(\log i)$ comparaciones encontrar un x_j tal que $z \leq x_j$ y realiza otras $O(\log i)$ comparaciones encontrar al índice i.

El mismo algoritmo puede ser usado cuando el tamaño de la secuencia es conocida, pero quizá i no sea suficientemente grande. Tendíamos, en tal caso, una versión mejorada de la búsqueda binaria con un desempeño computacional de $O(\log i)$ en vez de $O(\log n)$. Sin embargo, hay un factor de 2 en el desempeño computacional del algoritmo, pues ejecutamos dos búsquedas binarias. Este algoritmo es mejor sólo cuando i es $O(\sqrt{n})$.

1.4. Búsqueda Exponencial

Esta técnica es ideal para secuencias ordenadas de gran tamaño y es considerada como una variante de búsqueda binaria, por lo cual también emplea la estrategia divide y vencerás, lo que se hace evidente al observar cómo la técnica resuelve el problema.

Al igual que en la sección anterior se pide como condición inicial que la secuencia esté ordenada. Supondremos, sin pérdida de generalidad, que la secuencia está ordenada de forma ascendente. El problema se replantea como sigue:

Problema de Búsqueda.- Sea $S = \{s_1, s_2, \ldots, s_n\}$ una secuencia finita y no vacía, de tamaño n, ordenada ascendentemente, de números reales.
Determinar si existe z en el conjunto S con $z = s_i$, para $i, i = 1, 2, \ldots, n$.

Estrategia

Este método de búsqueda se basa en la idea de acotar y comparar, esto lo consigue dividiendo a S en intervalos de la forma $[2^i, 2^{i+1}]$. Una vez que se tiene el primer intervalo se compara a $s_{2^{i+1}}$ con z, y dependiendo de cómo sea la relación de orden se tienen tres posibles casos, los cuales surgen porque la secuencia está ordenada ascendentemente.

1. Si $s_{2^{i+1}} = z$, se considera una búsqueda exitosa y se da por finalizada.
2. Si $s_{2^{i+1}} > z$, se procede a realizar búsqueda binaria en el intervalo $[2^i, 2^{i+1}]$ para determinar si el elemento está en la secuencia.

3. Si $s_{2^{i+1}} < z$, se actualiza el valor de i como $i = i + 1$, se construye el nuevo intervalo con la forma $[2^i, 2^{i+1}]$, para realizar nuevamente la comparación. Cuando el tamaño de S sea menor a 2^{i+2} el último intervalo sería $[2^i, n]$.

Para el primer intervalo se fija la potencia de 2^k que se utilizará como límite superior, está potencia debe de ser relativamente grande; generalmente se usa $k = 8$ o $k = 10$. Sin embargo, por razones didácticas, para nuestro ejemplo, emplearemos como primera potencia a $k = 5$, es decir, la primera cota será 2^5.

Como ejemplo ilustrativo consideremos la Figura 1.4, en la que tenemos los primeros 500 números pares, es decir: $S = \{2, 4, 6, 8, \ldots, 998, 1000\}$ y buscamos el número $z = 608$.

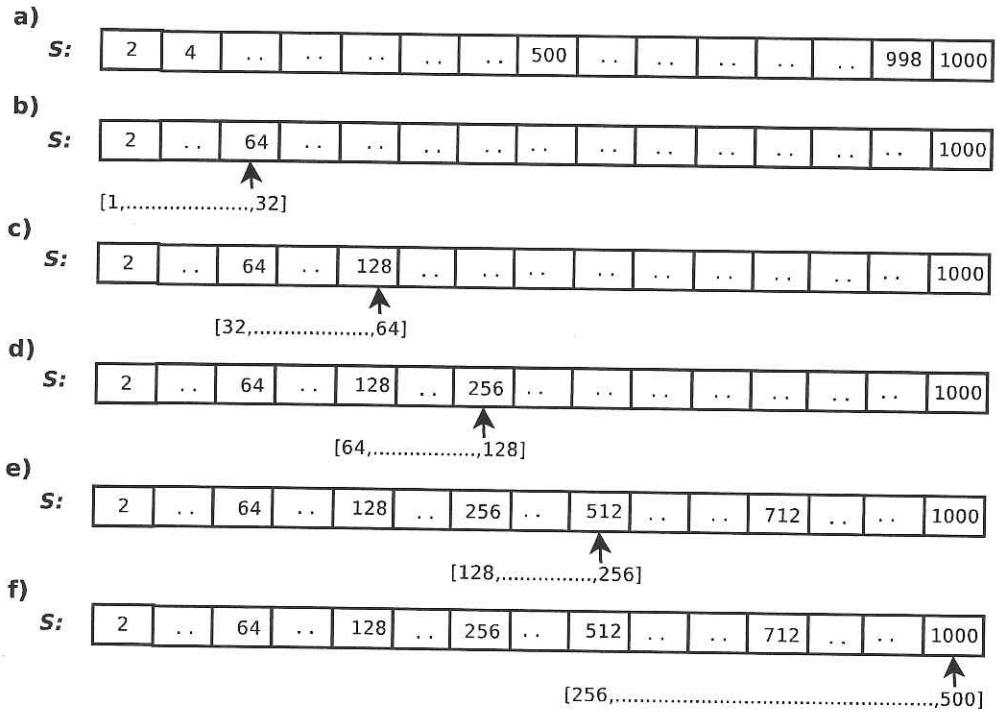


Figura 1.3: Ejemplo de búsqueda exponencial, I.

Se puede ver en los incisos del **(b)** al **(f)** que se realiza la partición en intervalos que son potencia de 2, así el primer intervalo va de la localidad 1 a la 32, es decir de 2^0 a 2^5, inciso **(b)**. Ahora, como $S[32] = 64$ se cumple con $64 < 608$, entonces se construye el siguiente intervalo $[2^5, 2^6] = [32, 64]$, inciso **(c)**. Nuevamente, el elemento en el límite superior del intervalo es menor al buscado por lo cual se construye el siguiente intervalo.

Dado que tenemos sólo 500 elementos en la secuencia el último intervalo construido contiene menos elementos que la potencia de 2 correspondiente, esté intervalo sería $[2^8, 2^9]$. Por tanto, el último intervalo debe ser $[2^8, n] = [256, 500]$.

Para nuestro ejemplo $z = 608$ es menor que 1000, entonces se inicia con búsqueda binaria, en el último intervalo, $[256, 500]$. La Figura 1.4, incisos del **(g)** al **(m)**, muestra el proceso.

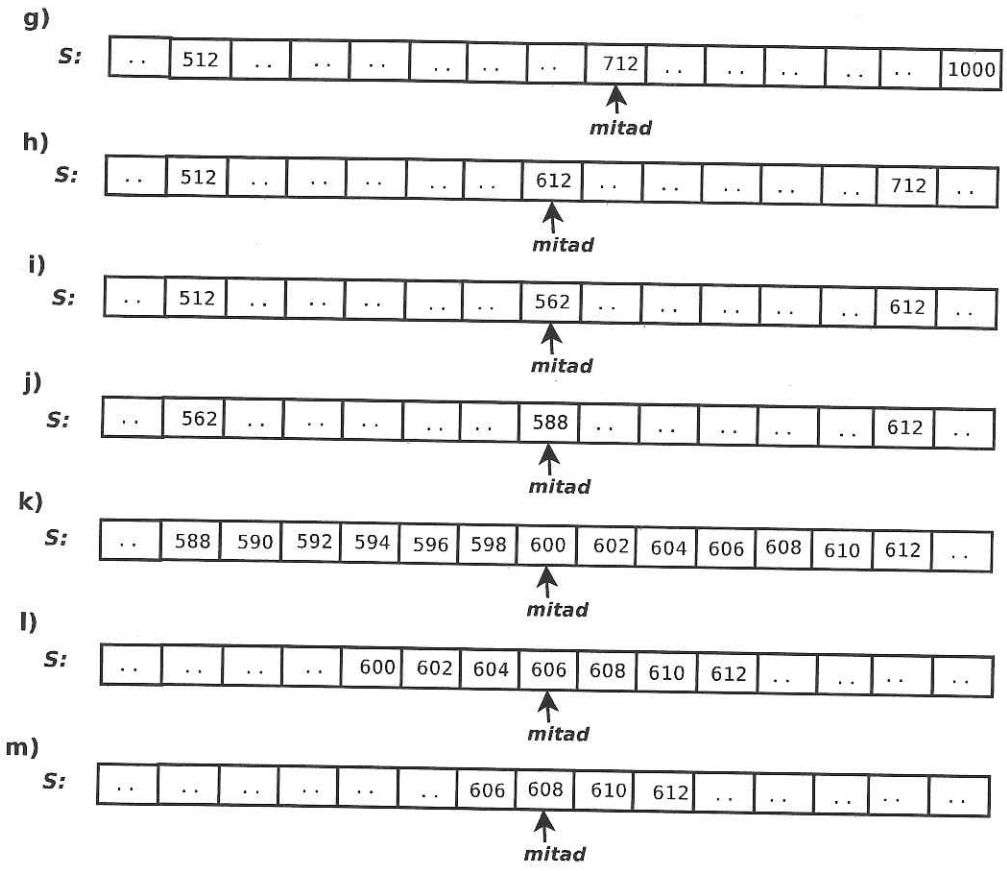


Figura 1.4: Ejemplo de búsqueda exponencial, II.

La búsqueda binaria en el intervalo ya no se ilustra detenidamente, sólo se hace énfasis en el elemento a la mitad debido a que esta técnica fue estudiada anteriormente.

Complejidad Computacional

Realizaremos el análisis del peor caso. Para facilitar los cálculos supondremos, sin pérdida de generalidad, que el tamaño de la secuencia es una potencia de dos, es decir $n = 2^k$, con $k \geq 8$ y la primera potencia es 2^8. El peor caso, para esta búsqueda, se da cuando el elemento buscado se encuentra en la último intervalo formado.

Nótese que para determinar que elemento buscado z se realizaron tantas comparaciones como intervalos, entonces es necesario determinar el número de intervalos creados. Para realizar esto, consideramos que cada intervalo es de la forma $[2^i, 2^{i+1}]$, como $n = 2^k$, entonces se forman $\log_2 n = k$ intervalos. Ahora bien, sobre el último intervalo $[2^{k-1}, 2^k]$, de tamaño 2^{k-1}, se realizaría una Búsqueda Binaria sobre un intervalo de de tamaño 2^{k-1}; por lo tanto, su tiempo de ejecución sería: $\log_2 2^{k-1} = (k-1) \approx \log n$. Por tanto, el desempeño computacional de la Búsqueda Exponencial, en el peor de los casos es de $O(\log_2 n)$.

Cuando el elemento z a buscar no está en la secuencia, este algoritmo acota el espacio de búsqueda y se queda con el intervalo donde estaría z. Para este escenario, en el peor caso, se llegaría al último intervalo y, por lo tanto, el desempeño computacional queda en $O(\log_2 n)$. Se puede demostrar que en caso promedio, la búsqueda exponencial requiere tiempo $O(\log \log n)$[7].

Algoritmo

Listado 6 Búsqueda Exponencial

```
//PreC: X[a,b]  es un arreglo  ordenado, no vacio y finito de enteros
//PostC: X no sufre cambios;
//       regresa la posicion i, si el elemento z es encontrado;
//       regresa -1 si z no se encuentra en el conjunto.

int BExponencial(int z; array X) {

 int lim_inf, lim_sup, n;
   lim_inf = 1;                                  // limite inferior inicial
   lim_sup = pow(2,8);                           // limite superior inicial
   n = S.length;                                 // num. de elementos en S
   while (lim_sup <= n) do {
      if ((lim_sup = n) && ( z > X[lim_sup] ))
            return -1;                           // busqueda fallida
        else if ( X[lim_sup] = key )
            return lim_sup;                      // busqueda exitosa
         else if ( z < X[lim_sup] )
            return binary(z,lim_inf,lim_sup);    // realiza busqueda binaria
          else {                                 // define el nuevo
            lim_inf := lim_sup;                  //   espacio de busqueda
            lim_sup := lim_sup*2;
          }
   }// end while

   if ( lim_sup > n )                            // aplica busqueda binaria
     return    binary(key,lim_inf,n);
}// end BExponencial
```

Para escribir el pseudocódigo de esta técnica se supone que la secuencia está contenida en un arreglo X y que no hay elementos repetidos.

Como se puede observar en el pseudocódigo del Listado 6, lo primero es determinar el probable espacio de búsqueda: [lim_inf , lim_sup]; hecho esto, se revisa la condición del ciclo while, el cual establecerá los siguientes intervalos.

En caso de entrar al ciclo, lo primero que se hace es revisar si lim_sup es igual a n, el tamaño de X, si es igual, se compara al elemento en la localidad lim_sup con el buscado, z, en caso de no ser el mismo se considera la búsqueda fallida y termina.

Listado 7 Proceso Binary

```
int procedure binary(int key; int min, max) {

   int mitad;
   while ( min < max ) do {
     mitad := (max+min)/2;
     if ( key=X[mitad] )  return mitad;
       else if ( X[mitad] > key ) min := mitad+1;
              else  max := mitad - 1;
   }//end while

   if ( min = max )
     if ( X[min] = key )   return min;
       else return -1;
}// end binary
```

En caso de que lim_sup $< n$, se tienen tres posibles casos:

1. Si X[lim_sup]$= z$, la búsqueda exitosa y se termina.
2. Si X[lim_sup]$< z$, entonces se inicia la búsqueda binaria en el intervalo [lim_inf , lim_sup].
3. Si X[lim_sup]$> z$, se establecen los nuevos valores para lim_inf y lim_sup.

Cuando se sale del ciclo y no se ha terminado la búsqueda o bien no se entró al ciclo, entonces se ejecuta la búsqueda binaria en el intervalo [lim_inf, n].

El código del procedimiento binary, presentado en el Listado 7, es esencialmente la búsqueda binaria. La única diferencia es que en binary se hace referencia al arreglo considerando los límites del arreglo.

Al observar de manera completa al pseudocódigo de búsqueda exponencial, se hace evidente que cuando se tiene una secuencia pequeña, éste se puede considerar como una búsqueda binaria, ya que solamente se emplearía el procedimiento binary. Para terminar, cabe mencionar que el valor de retorno -1, en caso de búsqueda fallida es válido para indicar que el elemento no está, pues estamos suponiendo que los índices del arreglo son positivos y el pseudocódigo presentado regresa la localidad donde se sí se encuentra el elemento z.

1.5. Búsqueda por Interpolación

Hasta ahora hemos estudiado técnicas que accesan linealmente a la secuencia, la dividen a la mitad o en intervalos. Sin embargo, si en el primer intento se quisiera iniciar la búsqueda a partir de una localidad cercana a la que podría contener al elemento deseado, se requiere de otro tipo de estrategia.

La idea de iniciar la búsqueda a partir de una posición que pensamos está cercana al elemento buscado, es la clave del método de búsqueda por interpolación y crea una posición candidata utilizando una función de interpolación lineal.

Esta técnica requiere que la secuencia de trabajo esté ordenada. El planteamiento del problema de búsqueda queda, ahora, como:

Problema de Búsqueda Sea $S = \{s_1, s_2, \ldots, s_n\}$ una secuencia de números, ordenada, finita y de tamaño n. Determinar, utilizando una función de interpolación, si existe el elemento z en S tal que $z = s_i$ con $i = 1, 2, \ldots, n$.

Estrategia

Para ver de forma intuitiva el funcionamiento de esta técnica consideremos el siguiente ejemplo: Cuando se quiere abrir un libro en cierta página, si conocemos el número total de páginas, abriremos éste dependiendo de qué número de página se quiera, así, si el libro tiene 100 páginas y se quiere la 50, se abrirá más o menos a la mitad; si se quiere la 25 se abrirá aproximadamente a un cuarto; y si se desea la 75 a tres cuartos. Es poco probable que al abrir el libro por primera vez se obtenga la página deseada, sin embargo es muy probable que se obtenga una cercana a ella y a partir de la cual se puede iniciar la búsqueda.

En el ejemplo se observa que dependiendo de qué página se quiera, es dónde se abre el libro; es decir, efectuamos una aproximación; en términos más formales diremos que se realiza una interpolación. La estrategia que emplea este método de búsqueda consiste en realizar interpolaciones sucesivas hasta que encontramos al elemento z o determinamos que éste no se encuentra. Nótese que al realizar cada interpolación también reducimos el espacio de búsqueda, ya que la siguiente búsqueda se realiza considerando el valor obtenido. De manera general, la búsqueda por interpolación consiste en:

1. Utilizar una función de interpolación lineal para determinar una posición p.
2. Comparar al elemento en la posición p con el dato buscado, z.
   a) Si son iguales, terminamos la búsqueda y la consideramos exitosa.
   b) En caso contrario, realizamos una nueva interpolación, en un espacio menor de búsqueda, y comparamos nuevamente.
3. Hacer el Paso 2, mientras se pueda seguir efectuando.
4. Si no es posible realizar el Paso 2 y el elemento buscado no está contenido en la posición p se termina la búsqueda y se considera fallida.

La función de interpolación lineal que se emplea es la siguiente:

$$pos = izq + \left\lceil \frac{(z - S[izq])\ (der - izq)}{S[der] - S[izq]} \right\rceil$$

Para que la interpolación sea efectiva los valores de las variables *der* e *izq* varían, dependiendo de la posición candidata obtenida, *pos*. Esto es, dependiendo de z y *pos*, se decide por el nuevo espacio de búsqueda, que puede ser el intervalo $[izq, pos]$ o $[pos, der]$. Consideremos un par de ejemplos para ilustrar el método.

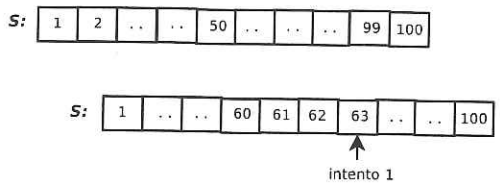


Figura 1.5: Ejemplo 1 de búsqueda por interpolación.

En la Figuras 1.5, la secuencia S está dada por los primeros cien enteros positivos: S={1,2,...,100} y se busca al 63. Aplicamos la búsqueda con los siguientes valores: $izq = 1, der = 100, z = 63, S[1] = 1$ y $S[100] = 100$, entonces,

$$pos = 1 + \left\lceil \frac{(63 - S[1])(100 - 1)}{S[100] - S[1]} \right\rceil = 1 + \left\lceil \frac{(63 - 1)(100 - 1)}{(100 - 1)} \right\rceil = 1 + 62 = 63.$$

Como se puede observar para este caso en la primera aplicación de la fórmula de interpolación da $p = 63$ que es donde se encuentra el elemento buscado y, por lo tanto, el proceso termina exitosamente.

En caso de buscarse un elemento que no se encuentre en la secuencia como sería el 101, la localidad por interpolación que se obtiene supera al número de elementos en la secuencia y, por lo tanto, se concluye que no está y se termina con la búsqueda.

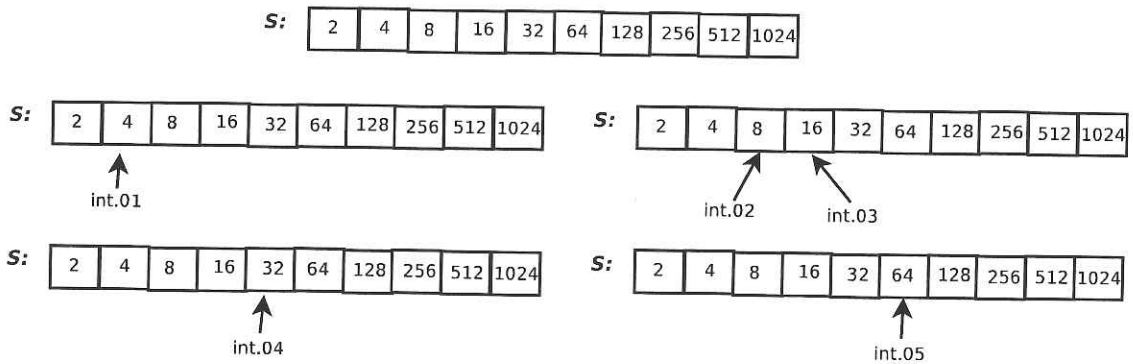


Figura 1.6: Ejemplo 2 de búsqueda por interpolación.

En la Figuras 1.6, ejemplo 2, la secuencia S está formada por las primeras diez potencias de 2: S={2, 4, 8, 16, 32, 64, 128, 256, 512, 1024} y se busca a $z = 64$. Aplicamos la búsqueda con los siguientes valores: $izq = 1, der = 10, z = 64$ y $S[1] = 2, S[10] = 1024$, entonces,

$$p_1 = 1 + \left\lceil \frac{(64 - 2)(10 - 1)}{(1024 - 2)} \right\rceil = 1 + \left\lceil \frac{(62)(9)}{1022} \right\rceil = 1 + \lceil 0{,}545 \rceil = 2.$$

Tenemos $S[2] = 4 \neq 64 = z$, entonces ahora $izq = 2, der = 10, z = 64, S[2] = 4$, así

$$p_2 = 2 + \left\lceil \frac{(64-4)(10-2)}{(1024-4)} \right\rceil = 2 + \left\lceil \frac{(60)(8)}{1020} \right\rceil = 2 + \lceil 0{,}470 \rceil = 3.$$

Revisamos: $S[3] = 8 \neq 64 = z$, el espacio de búsqueda queda $S[3.,10]$. Entonces los nuevos datos son: $izq = 3, der = 10, z = 64, S[3] = 8$, luego:

$$p_3 = 3 + \left\lceil \frac{(64-8)(10-3)}{(1024-8)} \right\rceil = 3 + \left\lceil \frac{(56)(7)}{1016} \right\rceil = 3 + \lceil 0{,}385 \rceil = 4.$$

Tenemos $S[4] = 16 \neq 64 = z$, entonces ahora $izq = 4, der = 10, z = 64, S[4] = 16$, así

$$p_4 = 4 + \left\lceil \frac{(64-16)(10-4)}{(1024-16)} \right\rceil = 4 + \left\lceil \frac{(46)(6)}{1008} \right\rceil = 4 + \lceil 0{,}285 \rceil = 5.$$

Revisamos: $S[5] = 32 \neq 64 = z$, el espacio de búsqueda queda $S[5.,10]$. Entonces los nuevos datos son: $izq = 5, der = 10, z = 64, S[5] = 32$, luego:

$$p_5 = 5 + \left\lceil \frac{(64-32)(10-5)}{(1024-32)} \right\rceil = 5 + \left\lceil \frac{(32)(5)}{992} \right\rceil = 5 + \lceil 0{,}161 \rceil = 6.$$

Finalmente, tenemos: $S[6] = 64 = z$ y la búsqueda ha terminado.

Para esta secuencia las posiciones obtenidas por interpolación avanzan de uno en uno, lo que se debe al hecho de que la distancia entre los datos no es constante y, por ello, la técnica supone que se está cerca del elemento buscado.

Los ejemplos mostrados dan una pista sobre el desempeño de este método y cómo influye la forma en la que están organizados los datos de entrada en él.

Análisis de Complejidad

El desempeño de este método depende tanto del tamaño de la secuencia como de la entrada, como se observó en los ejemplos. Por lo que, el método es muy eficiente cuando la secuencia de entrada consistente de elementos distribuidos en forma equidistante, como sería el caso de la numeración en las páginas de los libros.

En contraste cuando la secuencia está compuesta de elementos no distribuidos de manera equidistante, en el peor caso búsqueda por interpolación puede llegar a ser $O(n)$, lo cual indicaría que se comporta como una búsqueda lineal.

Manber [10] menciona lo siguiente en lo referente al caso promedio:

> "... se puede demostrar que el número promedio de comparaciones realizadas durante la búsqueda por interpolación, sobre todas las posibles secuencias, es $O(\log \log\ n)$; sin embargo, no se acostumbra hacer debido a que en la práctica no es mucho mejor que búsqueda binaria, por dos razones: la primera es, para secuencias de gran tamaño, es decir secuencias con n grande, $\log n$ es pequeño, pero $\log \log n$ no es mucho más pequeño y la segunda se refiere al hecho de que búsqueda por interpolación utiliza más operaciones ariméticas."

De lo anterior podemos concluir que en lo referente a complejidad en el peor caso búsqueda por interpolación tiene $O(n)$, mientras que para el caso promedio se tiene un $O(\log \log n)$, [7].

Algoritmo de Búsqueda por Interpolación

El pseudocódigo para búsqueda por interpolación es dado en el Listado 8, para implementarlo se supuso que la secuencia está contenida en un arreglo de n elementos, $X[1, n]$.

Listado 8 pseudocódigo Búsqueda por Interpolación

```
//  PreC: X[1,n] arreglo ordendo, finto y no vacio de enteros
// PostC: Regresa la posicion del elemento si esta en la secuencia,
//        Regresa -1 si no esta
//        X no sufre ningun cambio.
int BInterpolacion(int z; array X) {

   int izq, der, i;        // se definen las variables auxiliares.
   izq = 1;                // valor inicial del extremo izquierdo
   der = n;                // valor inicial del extremo derecho

   while (( X[der] >= z ) && ( z > X[izq] )) do {
                           // calcula la posicion a revisar
     i = izq+({(z-X[izq])(der-izq)} div {X[der]-X[izq]});

                         // actualiza los valores para izq y der
     if ( z > A[i] )     izq = i;
       else if ( z < X[i] )  der = i-1;
              else  izq = i;
   }// end while

      if ( X[izq] = z ) return  izq      // exito
       else return -1                    // sin exito
}//end BInterpolacion
```

En el pseudocódigo, podemos observar que al inicio se asignan los valores iniciales para los índices extremos *izq* y *der* los cuales corresponden a la primera y última localidad en la secuencia respectivamente; la condición del ciclo while se cumple en el momento en que se llega a una localidad que cruce el valor del elemento buscado, con lo cual se asegura que éste termina, dentro del ciclo se calcula el valor de la posición a revisar i por medio de la función de interpolación, así mismo se actualizan los valores para los extremos *izq* o *der* según sea el caso considerando la relación de orden entre el elemento contenido en la localidad escogida por interpolación y el elemento buscado.

Finalmente, el valor que se regresa es la posición que contiene al elemento buscado, z, en caso de encontrarse en la secuencia, y en caso de no estar regresa -1, este valor fué escogido debido a que se considera que todos los índices del arreglo son positivos.

Capítulo 2

Hashing

En este capítulo revisaremos las Funciones de Dispersión, *hashing*[1], usadas para construir búsquedas eficientes sobre tablas. Hemos revisado algoritmos de búsqueda con desempeños de $O(\log n)$, sin embargo, es posible, usando las Tablas Hash[2] hacer búsquedas en tiempo constante. En una Tabla Hash bien diseñada, encontrar una entrada puede tomar una o dos comparaciones, independientemente del número de elementos en la tabla.

Hashing es el proceso de transformar una llave en una dirección o localidad. Tal transformación involucra un cálculo, aritmético, rápido sobre la llave, junto a una manipulación de colisiones. Una colisión ocurre cuando la transformación manda diferentes llaves a una misma dirección. El término hashing sugiere que podamos *destrozar* la llave da alguna manera. Pero no importa cómo *destrocemos* las llaves siempre ocurrirá una colisión, por lo cual ésta es parte del proceso hashing.

La concesión hecha con *hashing* es que las entradas no están ordenadas en la tabla. Es posible recuperar el dato asociado para cualquier entrada, pero no es posible tener acceso ni al siguiente dato ni al anterior.

La funcionalidad, sobre la rapidez, de estas búsquedas tiene sentido en muchas aplicaciones. La tabla de símbolos de un compilador, es un buen ejemplo. Durante una compilación, el compilador debe revisar las palabras reservadas, y los nombres de la variables para obtener rápidamente sus atributos asociados. Una búsqueda veloz es de suma importancia, durante la compilación pueden realizarse cientos de búsquedas. Sin embargo, los símbolos no necesariamente están ordenados, excepto, posiblemente al final de la compilación cuando se genera una lista de referencias cruzadas. Aquí, un rápido ordenamiento debe ser usado para poner los símbolos en orden.

Ahora bien, resolver el problema por medio de tablas hash implica buscar o construir una función $h(k)$, que indique la posición exacta del elemento buscado. Una pregunta interesante ahora es ¿cuán difícil resulta encontrar o crear tal función? En un primer acercamiento, podemos decir que es realmente difícil, dado que bajo condiciones ideales la función $h(k)$ debe dar localidades diferentes para cada elemento, lamentablemente pocas funciones cumplen con esta característica.

---

[1] Utilizaremos frecuentemente el término *hashing*

[2] El Apéndice B, presenta brevemente el tipo de datos abstracto Tablas

Así, el método se concreta a resolver las siguientes situaciónes:

a) Buscar funciones que reduzcan el número de colisiones; es decir, funciones las cuales generen la mayor cantidad de direcciones posibles.

b) Buscar estrategias para resolver las colisiones cuando éstas ocurran.

c) Construir funciones rápidas, es decir de fácil cálculo.

2.1. Diseño de Funciones

Antes de iniciar el estudio de las funciones de dispersión o funciones hash[3] es necesario introducir algunos conceptos básicos. Iniciamos con clave o llave[4]. Le llamaremos llave, al valor asignado a un elemento que lo identifica de forma única de los demás.

Tradicionalmente, se asume que el universo de llaves es un conjunto finito de enteros positivos, digamos $\mathcal{N} = \{0, 1, 2, 3, ...\}$. Cuando las llaves no son enteros, se usa alguna forma de transformarlos, siempre es posible. En estas notas, supondremos, siempre, que las llaves son enteros positivos.

Dada una Tabla con m posiciones y una función de dispersión, $h(k)$, la Hipótesis de Hashing Uniforme, HHU, establece que la probabilidad de que a cada llave, generada por $h(k)$, se le asigne cualquiera de las m posiciones debe ser $1/m$. Toda función hash debe satisfacer la hipótesis de hashing uniforme. En la práctica, técnicas heurísticas son usadas para crear las funciones de dispersión y en el proceso se usa la información que se tiene sobre los datos.

Generalmente, una función de dispersión se deriva de tal manera que sea independiente de cualquier patrón que pueda existir entre los datos. Los tres métodos más usados, para el diseño de estas funciones, son: división, multiplicación y hashing universal.

Método por división

Dada una Tabla con m posiciones, este método toma el valor de la llave k módulo m. La función hash queda definida, formalmente, como: $h(k) = k \bmod m$.

Por ejemplo, si $m = 100$ y $k_1 = 123$ obtenemos $h(123) = 123 \bmod 100 = 23$.

Esta función es muy popular ya que automáticamente genera una posición dentro de la tabla. Sin embargo, tiene varios inconvenientes:

1. Diferentes llaves tendrán el mismo valor;
   para el ejemplo, si $k_2 = 523$ obtenemos: $h(123) = 123 \bmod 100 = 23$.
2. Si m es par las llaves pares irían a posiciones pares

Ante esto, hay que pensar seriamente el valor de m, una regla que en general funciona bien es que m sea un número primo no muy cercano a una potencia de dos. Por ejemplo: $m = 2^p - 1$. Cabe mencionar que hay formas más sofisticadas de elegir m.

[3]Usaremos indistintamente funciones hash o de dispersión

[4]Se usará indistintamente clave o llave

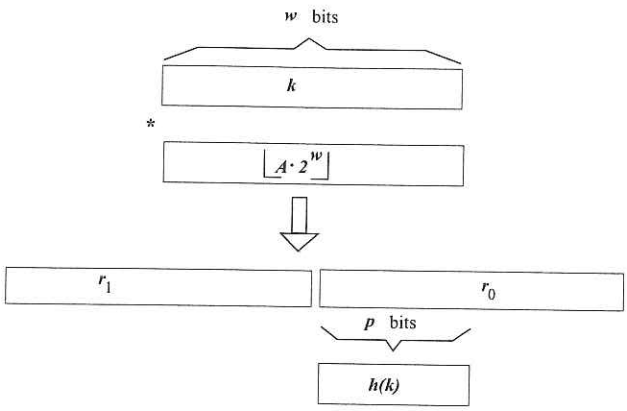


Figura 2.1: Método por Multiplicación

Método por multiplicación

Este método opera en dos pasos: en el primero, multiplica a la llave k por una constante A, donde $0 < A < 1$, obteniendo la parte fraccionaria de kA; en el segundo paso, multiplica ese valor por m, tomando el valor entero más cercano por abajo. De esta forma, la función queda: $h(k) = \lfloor m(k \cdot A \mod 1) \rfloor$, donde $(k \cdot A \mod 1)$ es la parte fraccionaria de kA; esto es: $k \cdot A - \lfloor k \cdot A \rfloor$.

La gran ventaja de este método es que el tamaño de la tabla, m, no es crítico. Una manera fácil de implementar esta función de dispersión, sobre cualquier computadora, inicia seleccionando m como una potencia de 2, digamos $m = 2^p$; después supone que el tamaño de una palabra en la computadora es de w bits y que la llave k cabe, también, en una palabra simple de w bits. Entonces, consideramos a A como una fracción de la forma $s/2^w$, con $0 < s < 2^w$. Ahora bien, la función, ilustrada en la Figura 2.1, queda determinada por las siguientes operaciones:

1. Multiplicamos: $k \cdot s$, $s = A \cdot 2^w$;
   k y s son enteros de w bits, su producto es un entero de $2w$ bits.
2. Consideramos el producto obtenido anteriormente como: $r_1 \cdot 2^w + r_0$;
   donde, r_1 es la palabra con los bits más significativos del producto y
   r_0 es la palabra formada con los bits menos significativos.
3. Tomamos a los p bits más significativos de r_0, como el valor final
   para la función hash.

La elección óptima de A depende de las características de los datos; sin embargo, se considera una buena elección el tomar $A \approx \left(\sqrt{5} - 1\right)/2 \approx 0.6180339887....$

Tomemos, por ejemplo, $k = 234527$, $m = 1000$ y A es el valor definido. Entonces:

$$\begin{aligned} h(k) &= \lfloor 1000 \cdot (234527 \cdot 0.6180339887 \bmod 1) \rfloor \\ &= \lfloor 1000 \cdot (149945.657267845 \bmod 1) \rfloor \\ &= \lfloor 1000 \cdot (0.657267845) \rfloor \\ &= \lfloor 657.267845 \rfloor \\ &= 657. \end{aligned}$$

Hashing universal

La idea principal del hashing universal consiste en seleccionar aleatoriamente, durante la ejecución, una función hash de una colección $\mathcal{H}$ de funciones cuidadosamente diseñadas. De manera formal, sea $\mathcal{H}$ una colección finita de funciones hash que proyecta un universo U de llaves en el rango $\{0, 1, 2, ..., m-1\}$.

A la colección $\mathcal{H}$ se le denomina universal si para cada par de llaves distintas $x, y \in U$, el número de funciones hash $h \in \mathcal{H}$ para el cual $h(x) = h(y)$ es justamente $|\mathcal{H}|/m$. Es decir, con una función hash aleatoriamente seleccionada de $\mathcal{H}$, la probabilidad de que estas dos llaves distintas colisionen es exactamente $1/m$.

Otras Funciones Hash

Para complementar esta sección, damos algunas funciones de dispersión que ejemplifican la creatividad que puede existir al construirlas.

a) **Selección de Dígitos.-** Dada una llave k, la función hash consiste en elegir cualesquiera dos o tres dígitos de la llave. Tal selección puede ser al azar o estar predeterminada por el diseñador. Por ejemplo: $h_1(k)$ regresa el número formado por los dígitos en las posiciones 3, 5 y 7. $h_1(20156439) = 163$.
$h_2(k)$ regresa el número formado por los dígitos en las posiciones 7, 5 y 3.
$h_2(20156439) = 361$.

b) **Desglosamiento.-** Dada una llave k, la función hash toma todos los dígitos de k y realiza algunas operaciones aritméticas básicas con ellos. Por ejemplo, $h_3(k)$ regresa el número formado por la suma de los dígitos. $h_3(20156439) = 30$.
$h_4(k)$ regresa el número formado por la suma de los dígitos, multiplicada por el cuarto dígito. $h_4(20156439) = 30 \cdot (5) = 150$.
Considerando que la suma de los dígitos de manera individual, generará un número pequeño (en el ejemplo la llave tiene ocho dígitos, el mayor valor que obtendríamos es 72), incluso si multiplicamos por otro (obtendríamos 72(9)=711 como máximo valor). Podríamos, entonces, *dividir* la llave en números de varios dígitos y operar con ellos. Por ejemplo: $h_5(20156439) = 201 + 56 + 439 = 696$;
$h_6(20156439) = 20 + 15 + 64 + 39 = 138$; $h_7(20156439) = 20(15) + 64 - 39 = 325$.

c) **Combinación** Cualquier combinación de las anteriores. Por ejemplo:
$h_8(k)$ regresa el número formado por los dígitos en las posiciones 3, 5 y 7; después tal resultado se multiplica por el dígito en la posición 4. $h_8(20156439) = 163 \cdot (5) = 815$.
$h_9(k)$ regresa el número formado por los dígitos en las posiciones 3, 5 y 7; después tal resultado se suma con el número formado por los tres primeros dígitos:
$h_9(20156439) = 163 + 201 = 364$

Dado que el resultado es un valor numérico, siempre es posible considerar, la función hash seleccionada módulo el tamaño de la tabla.

2.2. Manipulación de colisiones

En esta sección veremos cómo resolver las colisiones cuando éstas ocurren, para ello estudiaremos las dos estrategias básicas que son:

1) Direccionamiento abierto (*Open addressing*)
2) Encadenamiento (*Chaining*)

Direccionamiento abierto, *Open Addressing*

En el direccionamiento abierto, cada localidad de la tabla posee una llave o un valor nulo. Este comportamiento ocasiona problemas tanto de asignación como de búsqueda.

Cuando una colisión ocurre, se revisa la tabla con alguna estrategia hasta encontrar una localidad vacía, entonces la llave se almacena en tal lugar. Existen, al menos, tres técnicas básicas para el direccionamiento abierto:

1. Direccionamiento simple, *Open Addressing* ;
2. Aglomeración, *Clustering* ;
3. Compensación Cociente, *Quotient Offsets.*

Direccionamiento Simple

La estrategia más simple consiste en añadir 1 a la dirección, obtenida al inicio, hasta encontrar un lugar disponible. A este incremento se le denomina compensación, *off-set.* Variantes de esta estrategia podrían utilizar compensaciones mayores que 1.

Un intento o prueba es un acceso a la tabla. Se recomienda guardar el número de intentos para evitar caer en un ciclo infinito, cuando la tabla ya esté llena. Así, el ciclo eventualmente termina, pero regresa una dirección ocupada. El Listado 10 muestra un pseudo-código para la estrategia descrita con compensación igual a 1.

Se puede presentar un serio problema con esta estrategia. Suponga que se tienen varias llaves, $k_1, k_2, ..., k_j$ a las que se les asigna la misma dirección, d_o, con la estrategia dada, a cada llave k_i, se le asignará una localidad diferente en la tabla, la cual inicialmente está vacía. Se generará un problema al borrar una de estas llaves, digamos k_1, y buscar otra de ellas, digamos $k_i, 2 \leq i \leq j$. Al aplicar la función hash a k_i obtendremos d_o, pero esta dirección estará vacía porque ya se eliminó a k_1 y podríamos concluir erróneamente que la llave k_i no está en la tabla. Para evitar este lío, podemos agregar otro campo lógico a la Tabla Hash, por ejemplo, uno denominado *FueBorrado.* Se le asignará el valor de Falso durante la inserción y el del True durante el borrado. De esta manera, podríamos seguir buscando la llave k_i, en nuestro ejemplo, sin cometer error alguno.

Aglomeración

Un buen algoritmo hashing debe minimizar la ocurrencia de colisiones, aunque no pueda evitarlas. Si el espacio de las llaves es tan grande como el de direcciones, se podrían acumular los datos en una posición, porque las llaves dadas fueron creadas con un propósito específico.

Listado 10 Pseudocódigo HashOpenAddressing1

```
// PreC: A es Tabla Hash con Tsize maximo num. de elementos;
// PostC: obtiene una direccion para la llave (key) dada

HashOA1(int key ){
var int       dir = key mod Tsize;      // Aplica funcion hash
    int         i = 0;                  // numero de intentos
    boolean  libre = False;             // direccion libre

  while ( i < Tsize ) && (not libre)  do {
     if (not A[dir].EsOcupado) then     // direccion desocupada
        libre = True;
     else{                              // direccion ocupada
       if (key = A[dir].Item)           // las llaves son iguales
         then  libre = True;            // el dato ya estaba en la tabla
        else{                           // las llaves son diferentes
          dir = (dir+1) mod Tsize;      // incrementa en 1 la direccion
          i++ ;                         // incrementa el num de intentos
        } // end else
     } // end else
  } // end while
  return  dir                          // regresa la direccion obtenida
} // end HashOA1
```

Por ejemplo, el número de seguridad social podría estar compuesto de la siguiente manera: los primeros tres dígitos representan una región del país, los siguientes dos un estado y los últimos cuatro el número asignado al trabajador. De esta manera, al aplicar la función $h(key) = key \bmod 1000$, estaremos distribuyendo los datos de manera uniforme en el rango $[0, 999]$, provocando más acumulación en alguna localidad.

Este fenómeno es conocido como Aglomeración Primaria, *primary clustering*, y es la condición de tener un número excesivo de llaves-hash para la misma dirección inicial.

Sin embargo, la función hash que estamos usando crea una Aglomeración Secundaria, *secondary clustering*, que es la construcción de entradas para la tabla a lo largo de una trayectoria trazada por sinónimos de diferentes colisiones. Por ejemplo, suponga que se tiene una tabla inicialmente vacía y que generamos las direcciones $523, 524, 525, 526$, almacenamos las llaves en sus respectivas localidades, al volver a obtener cualquiera de estas direcciones, digamos la 523, usando open addressing, la respectiva llave tendría que viajar hasta la posición 527, al menos, para encontrar una localidad vacía. Cualquier método hashing que vincula la aglomeración secundaria es ineficiente ya que conlleva a una búsqueda secuencial.

En el ejemplo anterior, ¿Podríamos evitar la aglomeración secundaria si hubiesemos usado una compensación de 2? La respuesta es no, ya que la ruta de colisiones para 523 sería $523, 525, 527...$ y la ruta para el 525 sería $525, 527, ...$ Es evidente que cualquier salto generará conflictos.

Compensación Cociente

La aglomeración secundaria puede ser resuelta usando compensación cociente en el direccionamiento abierto. Para una llave dada, la dirección inicial se mantiene como $h(key) = key \bmod Tsize$, pero ahora la compensación (*off-set*) es el cociente entero de la llave *key* y el tamaño de la tabla, *Tsize*. Dado que el cociente es independiente del residuo dos llaves con el mismo residuo tendrán, seguramente, diferentes cocientes y diferentes compensaciones. En la siguiente tabla se presenta un ejemplo de esta estrategia con $h(key) = key \bmod 1000$:

key	Dir Inicial	*off-set*	Dir Final
304 27 1523	523	304271	523
513 01 7523	523	513017	540
287 32 6523	523	287326	849

El primer elemento ocupará la posición 523. Para los siguientes elementos, se realiza el siguiente cálculo: $(dir + offset) \bmod Tsize$. Para nuestro ejemplo:

$(523 + 513017) \bmod 1000 = 540$ y $(523 + 287326) \bmod 1000 = 849$.

De esta forma, los tres elementos tienen posiciones diferentes.

Hemos visto que con el ajuste de la compensación cociente, la técnica de acumulación secundaria es bien manejada, ya que los sinónimos para diversas colisiones (y aún en la misma colisión) no seguirán la misma trayectoria.

Si la acumulación primaria es minimizada, depende del cálculo de la dirección inicial; específicamente, sobre cómo la dirección inicial es dispersada en el espacio de direcciones.

Secuencia de Intentos

En esta subsección presentamos cómo generar diversas secuencia de intentos o pruebas con las siguientes propiedades:

(1) Debe ser capaz de revisar todas las localidades, de ser necesario;

(2) Debe generar intentos *dispersos*, para minimizar el número de colisiones.

Sea m el tamaño máximo de la tabla hash, sea $S_p = \langle p_o, p_1, ..., p_{m-1} \rangle$ la secuencia de intentos y sea $Hash(key)$ es la función de dispersión original. El primer requerimiento implica que la secuencia de intentos, S_p, sea realmente una permutación del conjunto de enteros $\{0, 1, 2, ..., m-1\}$, donde p_o es la primera dirección obtenida por la función de dispersión original. A continuación revisaremos diferentes formas de generarlas.

1. Lineal, *Linear Probing.*– La secuencia más simple que se puede generar es:

$$p_o, p_o + 1, ..., p_o + (m-1),$$

tomando la suma módulo m para regresar a la primera posición cuando la última es alcanzada. Este tipo de pruebas o intentos es conocida como intento lineal y el

i–ésimo intento está dado por las ecuaciones:

$$p_o = Hash(key); \qquad p_i = (p_o + i) \bmod m.$$

Claramente, esta es la función usada en el direccionamiento simple con compensación igual a 1.

2. **Cuadrática,** *Quadratic Probing.–* Una secuencia de intentos que elimina la aglomeración primaria es conocida como intento cuadrático, el i–ésimo intento está dado por las ecuaciones:

$$p_o = Hash(key); \qquad p_i = (p_o + i^2) \bmod m.$$

Con el intento cuadrático, el incremento entre cada prueba inicia en 1, luego 4, 9,16 y así sucesivamente; como en el caso anterior, usando el módulo regresamos al inicio de la Tabla cuando se alcanza la última posición. Ya que calcular directamente el cuadrado de i en cada llamada es costoso, la siguiente función de recurrencia nos proporciona los cálculos equivalentes:

$$c_o = 1; \qquad c_i = c_{i-1} + 2;$$
$$p_o = Hash(key); \qquad p_i = (p_{i-1} + c_i) \bmod m.$$

Estas ecuaciones están basadas en el hecho de que $1 + 3 + 5 + ... + (2i - 1) = i^2$.

El intento cuadrático *salta* mejor que el lineal, sin embargo, puede provocar una aglomeración secundaria, para evitarla se tienen las siguientes ecuaciones:

$$p_o = Hash(key); \qquad p_i = (p_o \pm i^2) \bmod m.$$

Esto quiere decir, que el incremento de i^2 puede ser hacia adelante o hacia atrás.

3. **Uniforme,** *Uniform Probing.–* Una secuencia de intentos que no provoca ninguna aglomeración, ni primaria ni secuendaria, es la Secuencia Uniforme de intentos, *uniform probe sequence*, que no hay que confundir con la función hash uniforme.

En esta secuencia, las localidades elegidas son aleatorias, donde cada dirección tiene la misma probabilidad de ser seleccionada. Se puede demostrar [6] que para este tipo de secuencias, el número promedio de pruebas durante la búsqueda está dado por las siguientes ecuaciones:

$$S(\alpha) = \frac{1}{\alpha} \cdot \ln\left(\frac{1}{1-\alpha}\right); \quad U(\alpha) = \frac{1}{1-\alpha}$$

Por ejemplo, cuando la tabla se encuentra medio llena (α =0.5), toma un promedio de 1.38 intentos buscar una entrada existente y cuesta 2 intentos, una no existente. Para α =0.75, el número de intentos es 1.8 y 4, respectivamente.

4. **Doble hash,** *Double Hash Probing.–* La secuencia uniforme de intentos, parece ser la mejor opción, pero también resulta un poco mítica la forma en que se obtienen los intentos. A continuación presentamos una secuencia de intentos denominada pruebas de Hash doble, la cual es cerrada.

La clave de esta secuencia de intentos es que, si ocurre una colisión, a la llave se le aplica otra función hash y el valor es usado como una compensación, *off-set*. De esta forma, se tienen las siguientes ecuaciones:

$p_o = Hash(key);$ $\qquad c = Hash2(key);$

$p_i = (p_{i-1} + c) \bmod m.$

Para que todas las localidades sean consideradas, se recomienda que c sea un primo relativo de m.

Un conjunto de funciones, sugeridas por Knuth [8], que virtualmente no provocan aglomeración secundaria son:

$Hash(key) = key \bmod m;$ $\qquad Hash2(key) = key \bmod (m-2).$

Nótese que esto significa aplicar el método de la división dos veces.

Para las secuencias de intentos, tanto lineal como doble, realmente se tiene una permutación del conjunto de enteros $\{0, 1, 2, ..., m-1\}$.

Tamaño de la tabla

Con la compensación cociente hemos visto que se crea un problema al usar la compensación, *off-set*, como el tamaño de la tabla, m. También habrá problemas cuando la compensación y el tamaño de la tabla tienen factores comunes.

Por ejemplo, suponga que el tamaño de la tabla es $m = 10$ y que la llave es un entero de dos dígitos. Suponga que los primeros tres elementos tienen llaves 11, 27 y 64, estos serán colocados en las posiciones 1, 7 y 4, respectivamente. Si el siguiente dato tiene llave 77: $dir = key \bmod m = 77 \bmod 10 = 7$ y $offset = key \text{ div } m = 77 \text{ div } 10 = 7$. Como la dirección obtenida es $dir = 7$ y está ocupada, debemos calcular una nueva dirección: $dir = (dir + offset) \bmod m = (7+7) \bmod 10 = 4$.

Sin embargo, la posición 4 ya está ocupada por el 64. Entonces volvemos a calcular: $dir = (dir + offset) \bmod m = (4+7) \bmod 10 = 1$. Nos encontramos con que la localidad 1, también ya está ocupada.

Este problema se puede evitar si el tamaño de la tabla m y el *offset* no tiene factores comunes, salvo el 1, por supuesto. Una manera fácil de resolver esto es seleccionando como tamaño de la tabla a un número primo. Así m y *off-set* no tiene factores comunes.

Encadenamiento

Aunque el direccionamiento abierto, con sus variantes, resuelve la mayoría de las colisiones, no las elimina. Otra manera de resolver colisiones consiste en almacenar cada localidad una lista de todos los elementos cuyas llaves tienen igual dirección.

Aquí también se pueden aplicar las estrategias anteriores y poner la restricción de que el tamaño de la tabla sea un número primo. Presentaremos dos técnicas:

1. Encadenamiento separado, *Separate Chaining* ;
2. Encadenamiento por fusión, *Coalesced Chaining.*

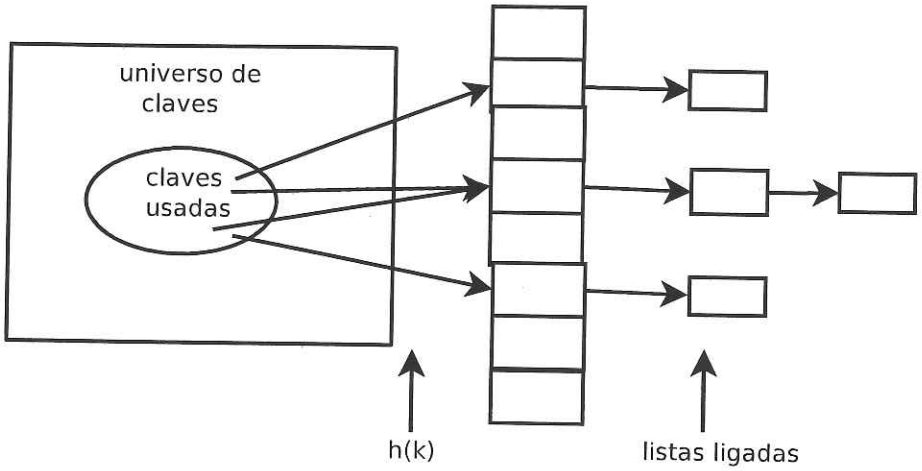


Figura 2.2: Vista gráfica de Encadenamiento Separado

Encadenamiento separado, *Separate Chaining*

El encadenamiento separado o hashing abierto consiste en asignar a cada posición de la tabla una lista ligada, donde almacenaremos los elementos que colisionen en la localidad. La lista debe ser implementada de tal forma que se mantenga ordenada después de cada inserción y la búsqueda dentro de una lista dependerá del número de elementos en ella.

Esta estrategia se muestra de forma gráfica en la Figura 2.2, se observa que para un conjunto del universo de llaves al aplicarles $h(k)$ obtenemos la localidad que les corresponde en la tabla, que es en realidad una lista.

Para ilustrar mejor la estrategia consideremos el ejemplo de la Figura 2.3 en donde se tiene la secuencia $S = \{119, 85, 43, 141, 72, 91, 109, 147, 38, 137, 148, 101\}$ y una tabla con $m = 13$ localidades; la función de dispersión está definida como $h(k) = k \bmod 13$.

En el inciso **(a)** se muestran los primeros 4 elementos que son insertados y no colisionan; en **(b)** se muestra la colisión de 72 con 85 y 43 con 147, así mismo se observa que estos elementos son insertados en la lista de tal forma que ésta se mantiene en orden; finalmente, en **(c)** observamos las dos últimas colisiones al insertar 137 y 148, de tal forma que el tamaño de las listas en las localidades 7 y 4 crecen.

Encadenamiento por fusión, *Coalesced Chaining*

Éste es una mezcla entre encadenamiento separado y direccionamiento abierto, ya que resuelve las colisiones manteniendo una lista ligada en las posiciones de la tabla pero en lugar de guardar los elementos en ella, guarda las posiciones en las cuales son puestos los elementos como si se tratara de un direccionamiento abierto.

Para ilustrar esta estrategia consideremos el ejemplo en la Figura 2.4. Se tiene una tabla de tamaño once, la función de dispersión está definida como: $h(k) = k \bmod 11$; y los datos a insertar son: $S = \{88, 212, 711, 288, 117, 211, 412, 517, 109, 105, 213\}$.

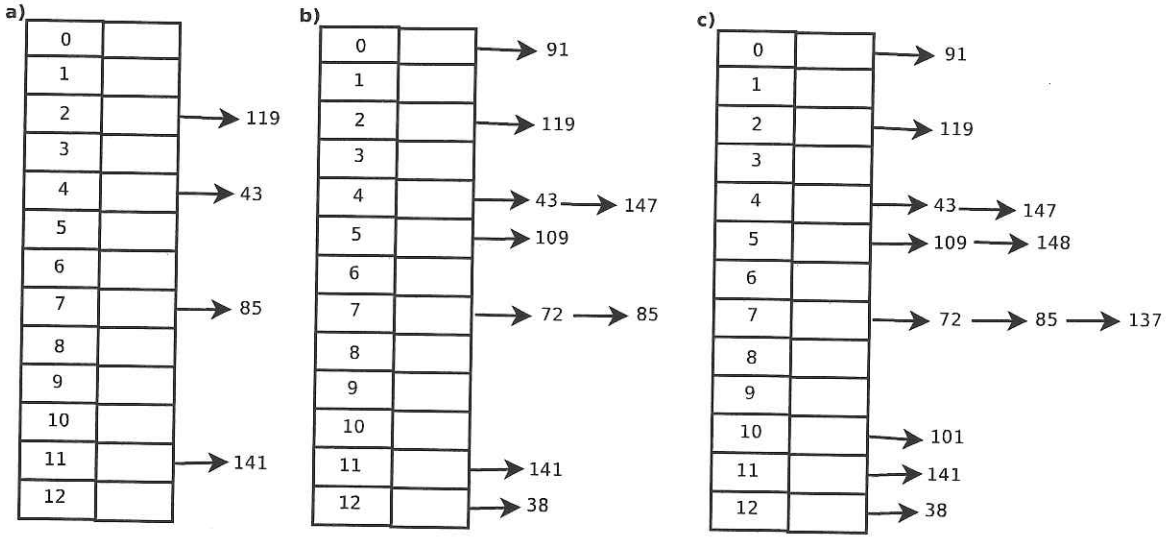


Figura 2.3: Ejemplo de Encadenamiento separado

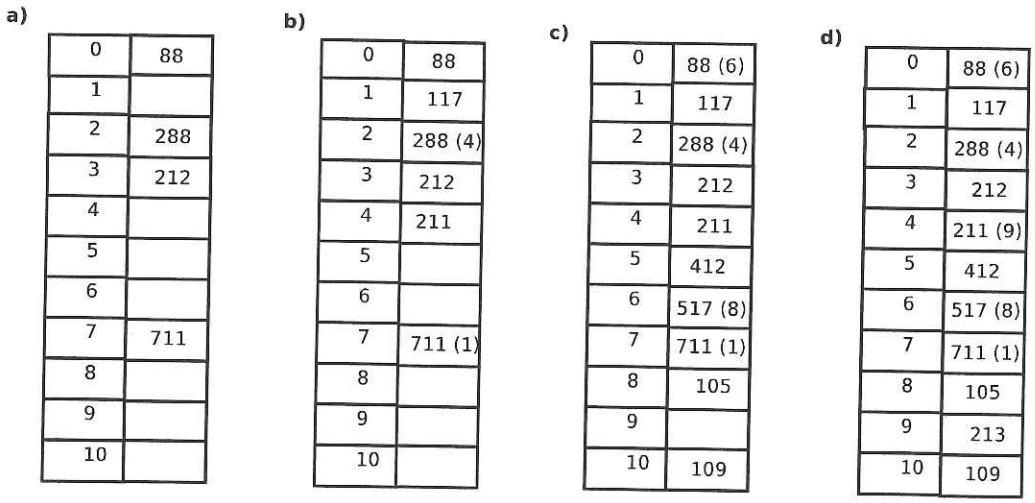


Figura 2.4: Ejemplo de Encadenamiento por fusión

En las primeras cuatro iteraciones, el proceso se sigue como en direccionamiento abierto, pero al insertar al 117 se observa que hay una colisión, entonces se busca la primera localidad vacía y en ella se introduce el dato que colisionó, tal localidad es la 1, para nuestro ejemplo. La Figura 2.4(a) muestra la tabla sin insertar al 117 y el (b) al insertarlo, entre paréntesis se pusó la dirección donde fue insertado. También se muestra que el 211 colisiona, y tiene que ser puesto en la localidad 4. Los incisos (c) y (d) muestran las dos últimas tablas.

Con esta estrategia, se inicia la búsqueda en una posición donde se generó una colisión; entonces realiza una búsqueda sobre una lista.

2.3. Análisis de Complejidad

Una vez revisados los conceptos necesarios para trabajar con tablas hash analizaremos la complejidad de éstas, para los dos métodos de solución de colisiones más usados.

Análisis de Tablas Hash con Encadenamiento Separado

Revisaremos el peor caso y el caso esperado del comportamiento de las tablas hash con Encadenamiento separado, *separate chaining*.

Peor Caso

En el peor caso tendríamos que todas las llaves fueron asignadas a una misma localidad y por ello están en una sola lista, como mencionamos en su momento la búsqueda dentro de una lista depende del tamaño de ésta y para este caso la lista tiene longitud n, por tal razón el problema se reduce a realizar una búsqueda secuencial lo que nos da una complejidad de $O(n)$.

Caso Promedio

Para facilitar el análisis supondremos la hipótesis de hashing uniforme y que $h(k)$ se realizá en tiempo $O(1)$. Para calcular el tamaño de las m listas utilizamos la hipótesis de hashing uniforme (HHU), entonces si queremos direccionar n llaves en m posiciones por HHU podemos decir que el tamaño promedio de cada lista es de (n/m), a esta cantidad se le conoce como factor de carga y se denota como $\alpha, \alpha = n/m$.

Cuando la búsqueda es no exitosa al aplicar la función h a la llave buscada ésta nos da una localidad de la tabla donde se encuentra la m-ésima lista, como la búsqueda no es exitosa entonces se recorre toda la lista; es decir, se realizan (n/m) comparaciones que es el factor de carga, ahora si sumamos el tiempo de cálculo de h tenemos que el tiempo total es de $O(1+\alpha)$.

Para revisar el caso de búsqueda exitosa asumiremos que cada uno de los n elementos almacenados en la tabla, tiene la misma probabilidad de ser buscados, también supondremos que cuando se inserta un nuevo elemento en la tabla, éste se inserta al final de

la m-ésima lista. De esta forma, el número esperado de comparaciones realizadas durante una búsqueda exitosa es 1 más el número de elementos existentes en la m-ésima lista cuando fue insertado el elemento buscado. Para obtener este número consideremos que tal elemento es el i-ésimo en ser insertado, entonces la longitud esperada en la lista es de $(i-1)/m$. Por lo tanto, el número esperado de comparaciones realizadas en una búsqueda exitosa lo podemos escribir como:

$$\frac{1}{n}\sum_{i=1}^{n}\left(1+\frac{i-1}{m}\right) = 1+\frac{1}{nm}\sum_{i=1}^{n}(i-1)$$

$$= 1+\left(\frac{1}{nm}\right)\left(\frac{(n-1)\,n}{2}\right) = 1+\frac{\alpha}{2}-\frac{1}{2m}$$

Por lo que el tiempo esperado para las tablas Hash utilizando encadenamiento separado, *separate chaining*, es de

$$O\left(2+\frac{\alpha}{2}-\frac{1}{2m}\right) = O\left(1+\alpha\right).$$

Análisis de Tablas Hash con Direccionamiento Abierto

Para realizar el análisis del peor caso para el direccionamiento abierto supondremos nuevamente la hipótesis de hashing uniforme (HHU), así como un factor de carga $\alpha < 1$. Para el análisis se considerarán los casos de búsqueda exitosa y no exitosa.

Búsqueda no Exitosa

Consideremos, p_i la probabilidad de que exactamente i pruebas llegaron a localidades ocupadas, cuando $i > n$ esta probabilidad es cero, esto indica que podemos encontrar a lo más n localidades ocupadas, por lo que el número esperado de comparaciones lo podemos escribir como:

$$1+\sum_{i=0}^{\infty} i\,(p_i)\,;$$

sea q_i la probabilidad de que al menos i pruebas accedieron a localidades ocupadas, entonces obtenemos:

$$\sum_{i=1}^{\infty} q_i = \sum_{i=0}^{\infty} i\,(p_i)\,.$$

Ahora, dado que la probabilidad de que en el primer intento se llegue a una localidad ocupada es α entonces $q_1 = \frac{n}{m}$, para el segundo intento $q_2 = \left(\frac{n}{m}\right)\left(\frac{n-1}{m-1}\right)$ y así sucesivamente. En general, podemos decir que para el i-ésimo intento:

$$q_i = \left(\frac{n}{m}\right)\left(\frac{n-1}{m-1}\right)\cdots\left(\frac{n-i+1}{m-i+1}\right) \quad \text{con} \quad q_i \leq \left(\frac{n}{m}\right)^i = \alpha^i;$$

entonces, el número esperado de comparaciones cuando la búsqueda es no exitosa queda:

$$1+\sum_{i=0}^{\infty} i\,(p_i) \;=\; 1+\sum_{i=1}^{\infty} q_i \leq 1+\alpha+\alpha^2+\alpha^3+\cdot s = \frac{1}{1-\alpha}$$

por lo cual, se tiene un desempeño computacional de $O\left(\frac{1}{1-\alpha}\right)$.

Búsqueda exitosa

Para determinar el número promedio de comparaciones realizadas cuando la búsqueda sí es exitosa hay que considerar las que se realizaron para poder insertar la llave buscada; si consideramos que cuando insertamos una llave, ésta no está en la tabla y, por lo tanto, si se buscara tal llave el resultado sería no exitoso, entonces para insertar una llave se harían $1/(1-\alpha)$ comparaciones y para insertar el i-ésimo elemento se realizarían $\left(\frac{1}{1-\frac{i}{m}}\right) = \left(\frac{m}{m-i}\right)$ comparaciones. Finalmente, el número promedio de comparaciones para el caso de búsqueda exitosa lo podemos escribir como:

$$\frac{1}{n}\sum_{i=0}^{n-1}\frac{m}{m-i} \;=\; \frac{m}{n}\sum_{i=0}^{n-1}\frac{1}{m-i} \;=\; \frac{1}{\alpha}\left(H_m - H_{m-n}\right)$$

donde $H_i = \sum_{j=1}^{i}\frac{1}{j}$ es el i-ésimo número harmónico.

Usando $\ln i \leq H_i \leq (\ln i + 1)$ obtenemos

$$\frac{1}{\alpha}\left(H_m - H_{m-n}\right) \leq \frac{1}{\alpha}\left(\ln m + 1 - \ln\left(m-n\right)\right) = \frac{1}{\alpha}\ln\frac{m}{m-n} + \frac{1}{\alpha} \;=\; \frac{1}{\alpha}\ln\frac{1}{1-\alpha} + \frac{1}{\alpha}$$

por lo que, para este caso, se tiene un tiempo de

$$O\left(\frac{1}{\alpha}\ln\frac{1}{1-\alpha} + \frac{1}{\alpha}\right)$$

Capítulo 3

El Problema de Ordenamiento

En computación el ordenamiento de datos cumple un rol muy importante, ya sea como un fin en sí o como parte de un proceso más complejo, una prueba de esto, son los métodos de búsqueda que requieren, como entrada, secuencias ordenada.

Un ordenamiento es la operación mediante la cual se organizan los elementos de una secuencia, basándonos en un criterio de orden. Entenderemos por criterio de orden al utilizado para establecer una relación binaria sobre la secuencia, y que es antisimétrica, transitiva y total; de tal forma que una vez definida la relación " $\leq$", para cualquier s_1, s_2 y s_3 en la secuencia, se tiene que:

Si $s_1 \leq s_2$ y $s_2 \leq s_1$ entonces $s_1 = s_2$.

Si $s_1 \leq s_2$ y $s_2 \leq s_3$ entonces $s_1 \leq s_3$; además, $s_1 \leq s_2$ o $s_2 \leq s_1$.

Una vez establecido el criterio de orden, podemos aplicar el proceso de ordenamiento sobre secuencias de cadenas de carácteres (listas de alumnos, cartera de clientes, por ejemplo) bajo un orden lexicográfico; también podemos organizar listas de números (reales o enteros) en orden ascendente o descendente.

En virtud de que siempre es posible construir una relación que asocie una cadena de carácteres a un entero, de manera única, nos limitaremos a trabajar con listas de enteros positivos.

Considere por ejemplo la secuencia de términos:

$T = \langle$Diseño, Algoritmo, Estructuras, Datos, Listas, Análisis, Árbol, Complejidad, Tiempo, Recursión , Proceso, Bases $\rangle$.

La siguiente asignación asocia cada elemento del conjunto T con un único elemento de los doce primeros enteros positivos:

1 – Algoritmo;	4 – Bases;	7 – Diseño;	10 – Proceso;
2 – Análisis	5 – Complejidad;	8 – Estructuras;	11 – Recursión;
3 – Árbol;	6 – Datos;	9 – Listas;	12 – Tiempo.

De esta manera, la secuencia de términos T queda directamente relacionada con la secuencia de enteros $T_E = \langle$ 7, 1, 8, 6, 9, 2, 3, 5, 12, 11, 10, 4 $\rangle$.

Clasificación de los algoritmos de ordenamiento

Existe una gran variedad de técnicas de ordenamiento, cada una de ellas con características particulares, así como ventajas y desventajas con respecto a las demás. A continuación presentamos algunas formas de clasificar los métodos de ordenamiento:

1. **Algoritmos de ordenamiento internos y externos.**
   Internos: Todos los elementos están contenidos en memoria.
   Si todos los datos a ordenar se encuentran almacenados en la memoria principal, el acceso a memoria puede ser usado sin ningún costo adicional. Es posible calcular llaves, a partir de los valores de los campos, o bien, usar estructuras de datos dinámicas para organizar los datos.
   Externos: Los elementos provienen de un dispositivo externo de almacenamiento.
   Si los datos a ordenar no caben todos a la vez en la memoria, entonces los patrones de acceso están más restringidos; una vez que un bloque de datos es recuperado, resulta importante hacer el *mayor uso* posible de él antes de dejarlo para manipular otro bloque.
   Es importante enfatizar que, muchas técnicas usadas para ordenamiento interno resultan completamente impropias para el ordenamiento externo y viceversa.

2. **Algoritmos de ordenamiento según el requerimiento de memoria.**
   Sin estructuras auxiliares, *in place* : Algunos métodos de ordenamiento interno no utilizán memoria adicional a la que ocupa la secuencia, el ordenamiento se genera a través de intercambios; es decir, los datos son simplemente re-organizados en sus posiciones existentes, usando una cantidad constante de memoria, independientemente del tamaño de la secuencia a ordenar.
   Con estructuras auxiliares: La técnica ocupa memoria adicional al construir estructuras auxiliares de datos, para así realizar el ordenamiento. Tal memoria adicional depende del tamaño de la secuencia.

3. **Algoritmos de ordenamiento de acuerdo a la complejidad computacional:**
   Comportamiento en el peor caso: Algunos métodos garantizan un desempeño computacional en el peor caso que puede considerarse aceptable.
   Comportamiento del caso promedio: La técnica sólo puede garantiza un buen comportamiento en el caso promedio. El caso esperado es muy común en la práctica. Las secuencias que están *cerca* del orden provocan que el desempeño del algoritmo sea más eficiente (rápido) que las secuencias que están *lejos* de tener orden.
   Algunos algoritmos tienen muy buen desempeño computacional para el peor de los casos garantizado y otro algoritmos, más eficientes, en la práctica, poseen un buen tiempo esperado para su desempeño computacional.

4. **Algoritmos de ordenamiento estables e inestables.**
   Estables: Si durante el proceso de ordenamiento se preserva el orden relativo entre los elementos de la entrada original, se dice que el método es estable.

Inestables: No se preserva el orden relativo de la entrada original durante la ejecución del proceso.
Depende de la aplicación la importancia de usar un algoritmo estable o no.

5. **Algoritmos de ordenamiento de acuerdo al modelo de cómputo:**
   Comparaciones: La técnica se basa en el Modelo de comparaciones.
   Sólo se hacen comparaciones entre los datos completos.
   Digital (Radix): El método utiliza el Modelo Radix.
   Se usa, y explota, la representación binaria de los datos para organizarlos.

6. **Algoritmos de ordenamiento según dificultad para recordarlos:**
   Fácil de recordar pero lentos: La estrategia empleada es muy simple pero el desempeño computacional resulta ser malo.
   Eficientes pero difícil de recordar: La estrategia, generalmente, es compleja por lo que no es fácil recordarlos pero son muy eficientes.

Dada la gran variedad de técnicas de ordenamiento existentes y las aplicaciones que las ocupan, en general no se puede decir que una de ellas sea la mejor. La elección del método más adecuado dependerá de la situación específica en la que se encuentre; es decir para escoger el más apropiado, debemos considerar qué sabemos sobre la secuencia a ordenar, por ejemplo:

1.– Información básica sobre los elementos: Número de elementos en la secuencia, tipo de datos, frecuencia de los elementos, tamaño del dato, etcétera.

2.– Información relevante sobre la secuencia: Cuán ordenada está la secuencia, cómo se generan los datos, distribución de los datos, cómo est'an almacenados los datos, entre otros.

3.– Frecuencia del ordenamiento: Qué tan frecuentemente se va a ordenar la secuencia, cada cuándo se actualiza la secuencia, etcétera.

Por ejemplo, si se requiere ordenar frecuentemente una secuencia, porque ésta cambia constantemente, deberemos usar un método cuyo desempeño computacional sea eficiente, el mejor. Pero si queremos ordenar por una única vez una secuencia, sin importar el tamaño, podríamos utilizar una técnica simple, cuya complejidad computacional sea pobre.

Antes de iniciar con el estudio de los métodos de ordenamiento recordemos que por simplicidad y, sin pérdida de generalidad, se considera a S como una secuencia de números enteros. De esta manera el problema queda definido como:

El Problema de Ordenamiento.– Sea $S = \{s_1, s_2, \ldots, s_n\}$ una secuencia no vacía y finita de n números enteros. Ordenar la secuencia S de forma ascendente.

Revisaremos los métodos de ordenamiento iniciando por los más sencillos, pero cuyo desempeño computacional es pobre, después estudiaremos los algoritmos cuya complejidad es la óptima, sobretodo en el caso esperado. Todos estos procesos están basados en el Modelo de Cómputo de comparaciones. Finalmente, revisaremos algunos algoritmos basados en el Modelo Radix.

En resumen, revisaremos los métodos en el siguiente orden:

Algoritmos con tiempo de ejecución $O(n^2)$, en el peor caso y en el caso esperado:

- Burbuja, *Bubble Sort*;
- Por Selección, *Selection Sort*;
- Por Inserción, *Insertion Sort*;
- Por Inserción Local, *Local Insertion Sort*;
- Shell Sort.

Algoritmos con tiempo de ejecución $O(n \log n)$, en el caso esperado:

- Merge Sort;
- Quick Sort;
- Heap Sort;
- Tree Sort.

Algoritmos con tiempo de ejecución $O(n)$:

- Counting Sort;
- Radix Sort;
- Bucket Sort.

Para cada uno de los algoritmos se describe su estrategia general; se proporciona al menos un ejemplo; se determina el desempeño computacional, tanto en el peor caso como en el mejor caso; y se proporciona un pseudo-código. Para la mayoría de los algoritmos se determina el desempeño computacional en el caso esperado.

Capítulo 4

Algoritmos Cuadráticos de Ordenamiento

En este capítulo presentamos cinco algoritmos de ordenamiento cuyo desempeño computacional, en el peor de los casos, es $O(n^2)$: Método de la Burbuja, Ordenamiento por selección, por inserción y por Inserción Local, así como el Algoritmo de Shell.

4.1. Método de la Burbuja

En esta sección estudiaremos uno de los algoritmos de ordenamiento más sencillos: el método de la burbuja, *bubble sort*. Este método es el más sencillo entre los algoritmos de ordenamiento. Es fácil de recordar y de programar, pero su desempeño computacional es bastante pobre, por lo cual es conveniente evitarlo.

El algoritmo de la burbuja ordena un arreglo intercambiando los elementos adyacentes que están en *desorden*, esto se repite hasta que todas las parejas adyacentes en el arreglo quedan en total orden. Cada paso completo pone al menor elemento en su posición correcta; es decir, después de completar el paso i, el $i-$ésimo elemento queda ordenado.

Estrategia

Para ordenar una lista, el método inicia al principio de la lista, compara cada elemento con el de la siguiente posición inmediata, y los intercambia de ser necesario. Después regresa al inicio, comparando e intercambiando. Continúa así hasta que el arreglo completo queda ordenado. Claramente, este proceso requiere diferentes pasos sobre los datos. Durante el primer paso, se comparan los primeros dos elementos en el arreglo, si están fuera de orden se intercambian. Entonces se comparan los elementos en el siguiente par (posiciones 2 y 3), de ser necesario se intercambian. Se procede de manera similar, comparando e intercambiando dos elementos a la vez, hasta llegar al final del arreglo.

La Figura 4.1 presenta un ejemplo. Se quiere ordenar a $S = 27, 10, 15, 38, 12$. En el primer paso, se comparan los dos primeros elementos: $s_1 = 27$ con $s_2 = 10$, y se genera

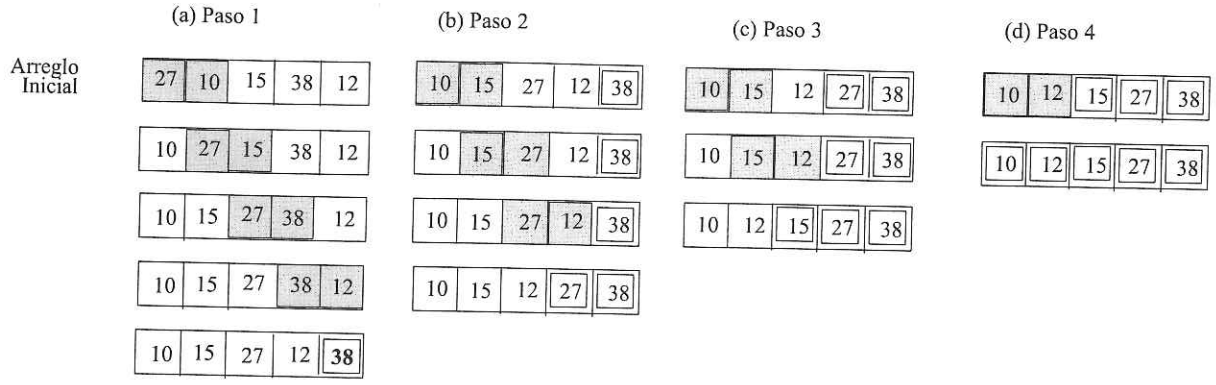


Figura 4.1: Ejemplo de Burbuja

un intercambio, pues están desordenados; después se comparan los siguientes elementos: $s_2 = 27$ con $s_3 = 15$, también se intercambian; luego se revisan los elementos $s_3 = 27$ y $s_4 = 38$, como están ordenados no hay intercambio; finalmente se comparan los elementos $s_4 = 38$ y $s_5 = 12$, ya que no están en orden, se intercambian. Al momento el arreglo no se encuentra ordenado, pero podemos observar que el elemento más grande, 38, quedo en la última posición del arreglo, la cual es la correcta para él.

Para el segundo paso, debemos regresar al inicio del arreglo y considerar las parejas exactamente como lo hicimos en el primer paso, sólo que ahora ya no consideraremos al elemento en la última posición; es decir, únicamente tomaremos en cuenta $n-1$ elementos, en esta ocasión. Al finalizar el segundo paso, los dos últimos elementos se encuentran en su posición correcta.

Durante el tercer paso, regresamos al inicio del arreglo y sólo revisaremos tres elementos, en general se consideran $n-3$; al finalizar el tercer paso, los tres últimos elementos se encuentran en su posición correcta.

Para el cuarto paso, en este ejemplo, sólo consideramos dos elementos, los dos primeros, como se encuentran en orden, no es necesario hacer ningún intercambio, por lo cual podemos concluir que el arreglo ha quedado ordenado.

La razón del nombre el método de la burbuja, es que los elementos más *pesados* van siendo *empujados* hacia *arriba*, como se puede observar en el ejemplo.

Análisis de Complejidad

El método requiere, a lo más, revisar el arreglo $(n-1)$ veces; es decir, se necesitan hacer $(n-1)$ pasos o iteraciones. El Paso 1 requiere $(n-1)$ comparaciones y, a lo más, $(n-1)$ intercambios. El Paso 2 requiere $(n-2)$ comparaciones y, a lo más, $(n-2)$ intercambios. En general, el Paso i requiere $(n-i)$ comparaciones y a lo más $(n-i)$ intercambios. Así, en el peor caso, el proceso requiere en total $(n-1)+(n-2)+\ldots+2+1 = n\cdot(n-1)/2$ comparaciones y, a lo más, la misma cantidad de intercambios.

Considerando que cada intercambio requiere tres movimientos, tenemos que el método de la burbuja requiere un total de $2(n)(n-1) = 2n^2 - 2n$ operaciones elementales, en el peor caso. Por lo tanto, podemos concluir que el algoritmo de la Burbuja requiere tiempo $O(n^2)$ en el peor de los casos.

El mejor caso ocurre cuando la secuencia ya se encuentra ordenada. El método ejecuta solo una iteración, durante la cual realiza $(n-1)$ comparaciones y cero intercambios. Por lo tanto, el Algoritmo de la Burbuja requiere tiempo $O(n)$ en el mejor de los casos.

Algoritmo

Para el pseudocódigo de esta técnica consideremos, por simplicidad, que la secuencia está contenida en un arreglo A. El código se presenta en el Listado 11.

Al revisar los dos ciclos for, se observa claramente que el desempeño computacional es cuadrático. El primer for se realiza a lo más $(n-1)$ veces y el segundo for, anidado en el primero, se ejecuta n veces.

Listado 11 Pseudocódigo Bubble Sort

```
// PreC: S una secuencia con n elementos, contenida en un arreglo A.
// PostC: A ha sido ordenado en forma ascendente.

burbuja(arrat A; int n){
 boolean sorted = false;          // falso cuando ocurre un intercambio
 int  temp;
  for (int pass=1; (pass < n) && !sorted; ++pass) {
      //invariante: A[n+1-pass.. n-1] esta ordenado

      sorted=true;                          //supone esta ordenado
       for (int i=0; i< n-pass; i++)
       {
          int si = i+1;                     // siguiente indice
           if (A[i]>A[si]) {                // hay desorden, intercambia
             temp = A[i];                   // intercambia el elemento en
             A[i] = A[si];                  // la posicion  i  con el de
             A[si] = temp;                  // la posicion  si
             sorted= false;                 // aun no esta ordenado
           } // end if
       } // end for i
    } // end for pass
} // end bubble sort
```

4.2. Ordenamiento por Selección

Este método de ordenamiento, *Selection Sort*, es de fácil implementación, por lo cual también es de los más recordados, pues su estrategía es bastante intuitiva.

Esta técnica no requiere memoria adicional, sólo el uso de una variable auxiliar para efectuar intercambios. Parte de su estrategia es comparar e intercambiar, razón por la cual realiza un gran número de comparaciones, lo que la hace lenta. Sin embargo, este método es bastante útil bajo ciertas; por ejemplo, si la secuencia a ordenar es pequeña.

Estrategia

La estrategia consiste en localizar al mínimo elemento en la secuencia y lo intercambia con el ubicado en la primera posición, luego busca el segundo más pequeño y lo intercambia con el de la segunda localidad, en general busca el i-ésimo elemento más pequeño y lo intercambia por el que se encuentra en la i-ésima posición, donde $i = 1, 2, \ldots, (n-1)$.

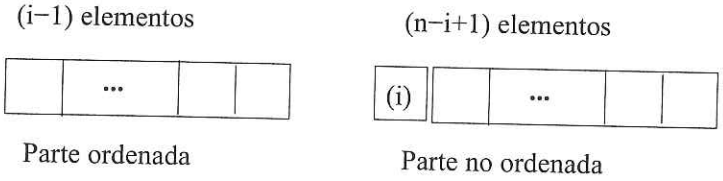


Figura 4.2: Estrategia general del *Selection Sort*

Otra forma de describir el proceso es la siguiente: en el paso general i se tiene la lista dividida en dos partes; la primera $\mathcal{L}_{\mathcal{O}}$ tiene $(i-1)$ elementos y está ordenada; la segunda, $\mathcal{L}_{\mathcal{R}}$, el resto de la lista, no está necesariamente ordenada y tiene $(n-i+1)$ elementos. En este paso, se busca al menor elemento del resto de la lista, $\mathcal{L}_{\mathcal{R}}$, y lo coloca en la primera posición de $\mathcal{L}_{\mathcal{R}}$. La Figura 4.2 ilustra gráficamente esta estrategia.

Para comprender mejor el proceso ilustramos en la Figura 4.3 un ejemplo de de esta técnica. Como podemos observar, para cada i, $1 \leq i \leq 9$, buscamos el correspondiente i-ésimo elemento más pequeño (flecha punteada), así cuando $i = 1$ intercambiamos al 9 por 0; algo que hay que notar es que conforme el índice i se incrementa el número de comparaciones realizadas para encontrar al elemento más pequeño en turno decrece, esto nos da una pista para poder realizar el análisis de esta técnica.

Análisis de Complejidad

Si consideramos la observación realizada, es facíl darse cuenta que para esta técnica el análisis del peor caso, en el caso promedio e incluso en el mejor caso pueden ser realizados simultáneamente ya que debido la estrategia empleada no hay diferencia entre uno y otro.

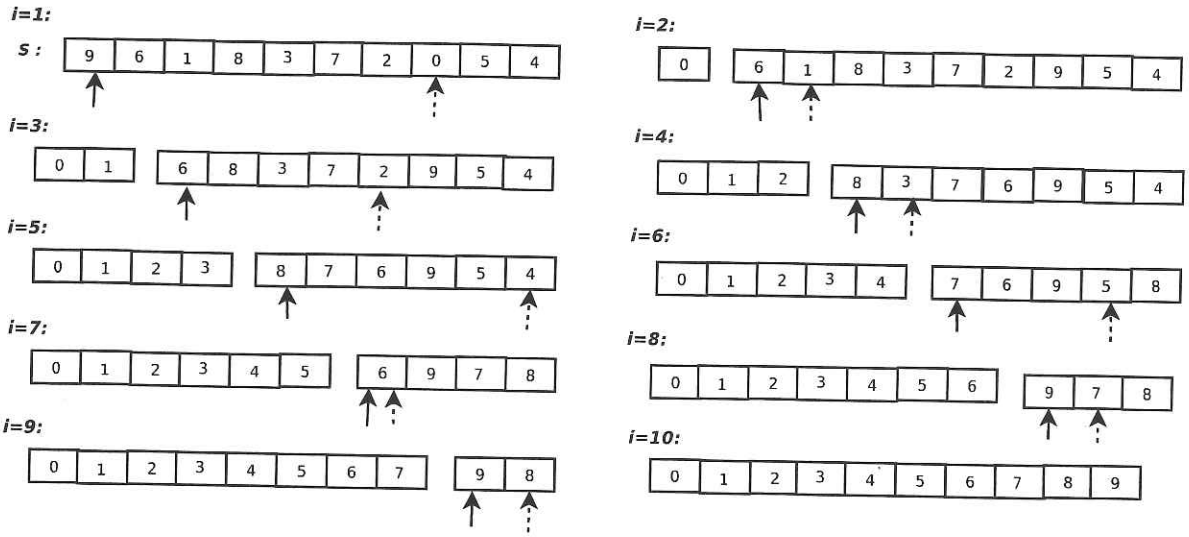


Figura 4.3: Ejemplo de *Selection Sort*

Siguiendo la pista obtenida, tenemos que al inicio, método realiza $(n-1)$ comparaciones para localizar al elemento más pequeño, cuando el índice es $i = 2$, hace $(n-2)$ comparaciones, con $i = 3$ realiza $(n-3)$. Ahora, podemos decir que, para el i-ésimo elemento hace $(n-i)$ comparaciones, entonces el total de comparaciones es:

$$(n-1) + (n-2) + \ldots + (n-i) + \ldots + 2 + 1,$$

lo que podemos reescribir como:

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

Por lo tanto, podemos concluir que el tiempo de ejecución para esta técnica tanto en el peor caso como el mejor caso y en el caso esperado es de $O(n^2)$. Por lo cual, podemos concluir que el desempeño computacional del Selection Sort es de $\Theta(n^2)$.

Algoritmo

Para el pseudocódigo de esta técnica, presentado en el Listado 12, consideremos, por simplicidad, que la secuencia está contenida en un arreglo A.

Como se aprecia en el pseudocódigo, el acceso a cada localidad del arreglo se realiza de manera secuencial, esto se hace al iniciar la búsqueda del i-esimo menor elemento, ya que a través del ciclo for se coloca un apuntador en la localidad i, mientras que con el segundo ciclo for y la variable j se encuentra al elemento que le corresponde, una vez que se encuentran estos dos elementos se realiza el cambio y se sigue con la localidad contigua.

Listado 12 Pseudocódigo Selection Sort

```
// PreC: S una secuencia con n elementos, contenida en un arreglo A.
// PostC: A contiene en orden ascendente a los elementos de S.

selection(array A; int n){

 int i, j, min, temp, aux, n;
 for (i=1; i=<n; i++) {              // se inicia la busqueda del i-esimo
       min=i;                        // menor elemento
       for (j=i+1; j=n; j++) {
           temp = j;
           if (A[temp]<A[min])  // se encuentra al i-esimo menor elemento
              min = temp;
           aux = A[i];             // intercambia al minimo con el de
           A[i] = A[min];         // la posicion  i
           A[min] = aux;
        } // end for j
    }// end for i
}// end selection
```

4.3. Ordenamiento por Inserción

El ordenamiento por inserción, *Insertion Sort*, es uno de los más conocidos, debido a que su estrategia es bastante natural y resulta fácil de implementar.

Los requerimientos de memoria para este método son mínimos, ya que sólo requiere una variable temporal para llevar a cabo el ordenamiento. Parte de la estrategia consiste en comparar, desplazar e insertar, razón por la cual es una técnica lenta debido al número de comparaciones que realiza. Sin embargo, esta técnica es muy útil cuando la secuencia que deseamos ordenar sabemos que está casi ordenada.

Estrategia

El método resulta muy natural, supongamos que se tiene una secuencia de números ordenados, a la cual agregaremos un nuevo elemento y una vez hecho esto, se ordenará nuevamente la secuencia. Es claro que la forma más eficiente de realizar estas dos operaciones, es insertar el nuevo elemento en su lugar correspondiente, en la secuencia, así ya no hay necesidad de un reordenamiento porque se mantiene ordenada.

Otra forma de describir el proceso es la siguiente: en el paso general i, se tiene la lista dividida en dos partes; la primera, una lista ordenada, $\mathcal{L}_{\mathcal{O}}$ con $(i-1)$ elementos; la segunda, $\mathcal{L}_{\mathcal{R}}$, el resto de la lista, no está necesariamente ordenada y tiene $(n-i+1)$ elementos. En este paso, se toma el primer elemento del resto de la lista, $\mathcal{L}_{\mathcal{R}}$, (que es el $i-$ésimo de la lista original, s_i) y se busca la posición correcta de tal elemento en $\mathcal{L}_{\mathcal{O}}$, abriendo el espacio para insertarlo. La Figura 4.4 ilustra gráficamente esta estrategia.

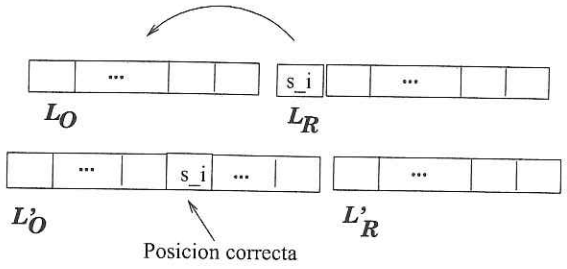


Figura 4.4: Estrategia General de *Insertion Sort*

Se desea ordenar la secuencia $S = \{9, 6, 1, 8, 3, 7, 2, 0, 5, 4\}$. La Figura 4.5 ilustra el ejemplo. En el inciso **(a)** el método inicia considerando una subsecuencia ordenada de un elemento que en nuestro ejemplo es $S_1 = 9$ y le agrega el 6, al cual coloca en la posición que le corresponde en la subsecuencia que contiene a los dos primeros elementos, así la nueva subsecuencia ordenada es $S_2 = \{6, 9\}$ como se observa en **(b)**, aquí además se indica que el siguiente elemento a insertar es el 1, que está en la posición 3. En los siguientes incisos se sigue con el proceso, finalmente en **(j)** obtenemos la secuencia ordenada.

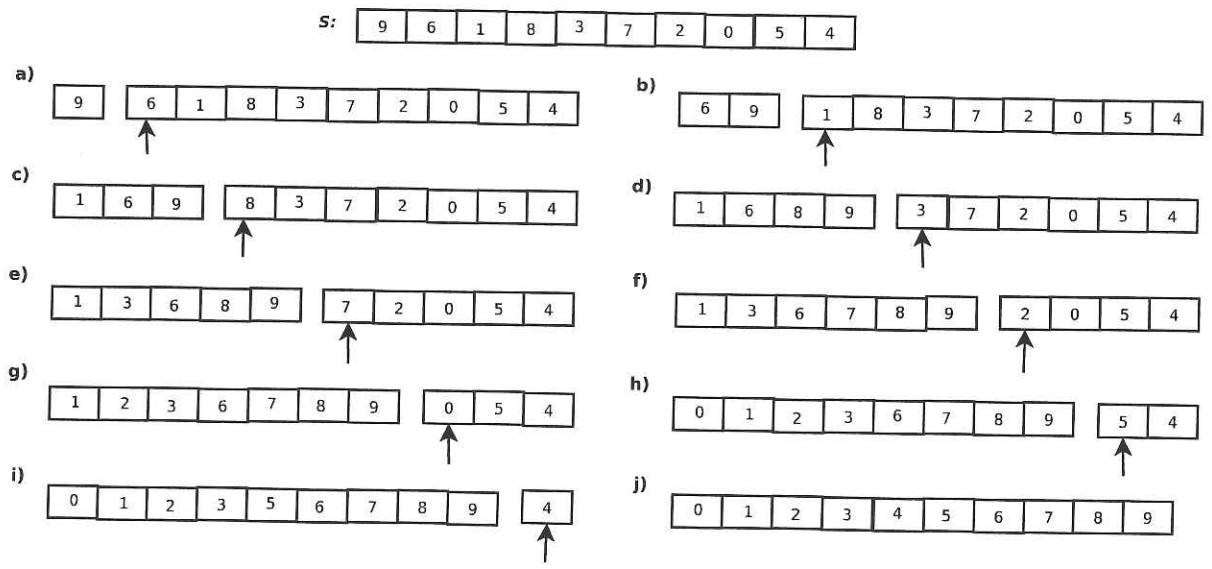


Figura 4.5: Ejemplo del Ordenamiento por Inserción

Análisis de Complejidad

Para este método se realizará el análisis de los tres casos: el peor, el mejor y el caso esperado; para todos los escenarios, supondremos que la secuencia ordenada se forma al agregar un nuevo elemento a una subsecuencia ya ordenada.

Peor caso

Este escenario se da cuando el elemento insertado es siempre mayor a los que se encuentran en la secuencia, teniendo en cuenta esto, si al inicio tenemos una secuencia de tamaño uno, al agregarle un elemento debemos realizar una comparación. Una vez insertado el elemento en su posición correcta, si agregamos otro, se realizaran dos comparaciones ya que secuencia tiene dos elementos, en general para este escenario se requieren i comparaciones para insertar el i-ésimo elemento; si suponemos que se ordenaran n datos entonces $i = 1, 2, \ldots, (n-1)$, como lo que nos interesa es el número total de comparaciones, éste es obtenido al hacer la suma sobre las comparaciones realizadas para los $(n-1)$ elementos insertados lo que podemos escribir como:

$$1 + 2 + \ldots + (n-1) = \sum_{i=1}^{n-1} i = \frac{n \cdot (n-1)}{2} = \frac{1}{2}(n^2 - n).$$

Por lo tanto para el peor caso se tiene una complejidad de $O(n^2)$.

Mejor caso

Este escenario ocurre cuando la secuencia a ordenar ya se encuentra ordenada. Suponga que estamos ordenando en forma ascendente. En el $i-$ésimo paso, el elemento en la posición i, $S[i]$, se compara con el $S[i-1]$, resulta ser mayor, así que no hay cambios; resulta que $S[i]$ ya está en su posición correcta. Entonces, en el mejor caso, se realizan $(n-1)$ comparaciones en total y no hay intercambios, en ninguna iteración. Por lo tanto, en este caso, el desempeño computacional del Insertion Sort es $O(n)$.

Caso Esperado

Para realizar el caso esperado, suponemos que la secuencia a ordenar ha sido obtenida bajo una distribución uniforme. Para análizar este caso se realizan dos cálculos. El primero consiste en estimar el número promedio de comparaciones que se requiere para colocar un elemento en su posición correcta. Hay que ver cuántas posibles localidades son candidatas para contener al i-ésimo elemento y cuántas comparaciones se realizan para cada una de ellas. Empezaremos con los casos simples y trataremos de encontrar algún patrón para generalizar.

Sea $S = \{s_1\}$, al agregarle s_2 se tienen dos posibles lugares, antes o después de s_1, para cualquiera de los dos casos hace una comparación. Supongamos, sin pérdida de generalidad, que la secuencia obtenida fue $S = \{s_1, s_2\}$, al añadir otro elemento, s_3, hay tres posibles localidades para él. Si le corresponde la primera, antes de s_1, entonces se hace una comparación; si queda segundo, entre s_1 y s_2, se realizan dos; si va en la tercera, después de s_2, se hacen dos comparaciones. Si ahora se agrega s_4 se tienen cuatro posibles localidades, los dos primeros casos son idénticos que cuando se agregó s_3, para la tercera localidad se requiere tres comparaciones y para la cuarta se requieren tres.

De aquí podemos ver que $(i+1)$ es el número de posiciónes posibles para el i-ésimo elemento y la cantidad de comparaciones realizadas, en total, para los primeros i lugares es: $(1+2+3+\ldots+i)$; además para el $(i+1)$ se hacen i comparaciones.

Por otro lado, la probabilidad de que s_i se coloque en cualquiera de las posiciones posibles es de: $(1/i+1)$. Esta probabilidad surge del hecho de que aún no se ha efectuado ninguna decisión sobre s_i y éste es tomado de manera uniformemente aleatoria con respecto a los elementos de la secuencia. Por lo tanto, el número promedio de comparaciones realizadas para insertar al $i-$ésimo elemento en la secuencia lo escribimos queda como:

$$A_i = \frac{1}{i+1}\left[\left(\sum_{p=1}^{i} p\right)+i\right] = \frac{1}{i+1}\left[\frac{i\,(i+1)}{2}+i\right] = \frac{i}{2}+\frac{i}{i+1} = \frac{i}{2}+1-\frac{1}{i+1}.$$

El segundo cálculo, consiste en determinar la cantidad de comparaciones realizadas para ordenar la secuencia, si deseamos ordenar una secuencia de tamaño n, entonces se requieren realizar $(n-1)$ inserciones. Así, considerando el resultado anterior, el número promedio de comparaciones realizadas para ordenar la secuencia es:

$$A(n) = \sum_{i=1}^{n-1} A_i = \sum_{i=1}^{n-1}\left(\frac{i}{2}+1-\frac{1}{i+1}\right) = \sum_{i=1}^{n-1}\left(\frac{i}{2}\right)+\sum_{i=1}^{n-1}(1)-\sum_{i=1}^{n-1}\left(\frac{1}{i+1}\right)$$

$$= \sum_{i=1}^{n-1}\left(\frac{i}{2}\right)+\sum_{i=1}^{n-1}(1)-\left(\sum_{i=1}^{n}\frac{1}{i}\right)-1.$$

Ahora bien, ya que,

$$\sum_{i=1}^{n}\frac{1}{i+1} = \sum_{i=2}^{n}\frac{1}{i} = \left(\sum_{i=2}^{n}\frac{1}{i}\right)-1 \approx \ln n - 1,$$

tenemos que,

$$A(n) = \left(\frac{1}{2}\right)\frac{n\,(n-1)}{2} + (n-1) - (\ln\, n\, -1)$$

$$= \frac{n^2-n}{4}+(n-1)-(\ln n-1) = \frac{n^2+3n-4}{4}-\ln n+1.$$

Por lo que para el caso promedio se tiene una complejidad de $O(n^2)$.

Listado 13 Pseudocódigo Insertion Sort

```
//PreC: A un arreglo que contiene a una secuencia S con n elementos.
//PostC: A esta en orden ascendente.

insertionSort(array A; int n){
 int i,j,k,temp,n;
  for (i=2; i= n; i++) {
     temp = A[i];
     for  (j=1; j=i-1; i++) {
          if (temp< A[j]) {
            k=A[j];    A[j]=temp;    temp=k; }
        } // end for j
     } // end for i
} // end InsertionSort
```

Algoritmo

El pseudocódigo para este método es presentado en el Listado 13 y para su eleboración se supuso que la secuencia está contenida en un arreglo A, que no es vacía y es finita.

Como se puede observar en el Listado 13, el primer ciclo for se encarga de ir recorriendo el arreglo; es decir, en este ciclo se indica que elemento se ordenará, mientras que el segundo ciclo for se encarga de encontrar su posición correcta en la parte del arreglo ya ordenado, una vez que se tienen ambos datos se procede a realizar el ordenamiento al insertar el elemento en su posición correspondiente y a recorrer los elementos necesarios dentro del arreglo para seguir teniendo una parte ordenada.

En el Listado 14, se presenta otra versión del código.

Listado 14 Pseudocódigo 2 de *Insertion Sort*

```
//PreC: A un arreglo que contiene a una secuencia S con n elementos.
//PostC: A esta en orden ascendente.

insertionSort(array A; int n){
  int j,i,temp;
  for (i=1; i < n; i++){
        temp=A[i];
        for (j=i; j>0 && A[j-1]>temp; j--){
           A[j]= A[j-1];
           A[j]=temp;
        } // end for j
    } // end for i
} // end InsertionSort
```

4.4. Ordenamiento de Inserción Local

Insertar elementos en una secuencia ordenada de tal manera que ésta siga ordenada, parece ser una gran idea para solucionar el problema de ordenamiento. Pero, ¿es posible realizar estas inserciones sin necesidad de comparar desde el inicio de la secuencia? La respuesta es sí y el Ordenamiento por Inserción Local (*Local Insertion Sort, LIS*) es el método que tiene como estrategia base tal idea. Esta técnica añade cada nuevo elemento a la lista considerando la posición donde se realizó la última inserción; además emplea una lista biligada, la cual permite tener acceso a los elementos en ambas direcciones.

Estrategia

La clave de este método consiste en mantener una lista biligada L siempre en orden, así como un apuntador u al último elemento insertado; a partir de u se inicia la búsqueda de la posición correcta para insertar el nuevo elemento.

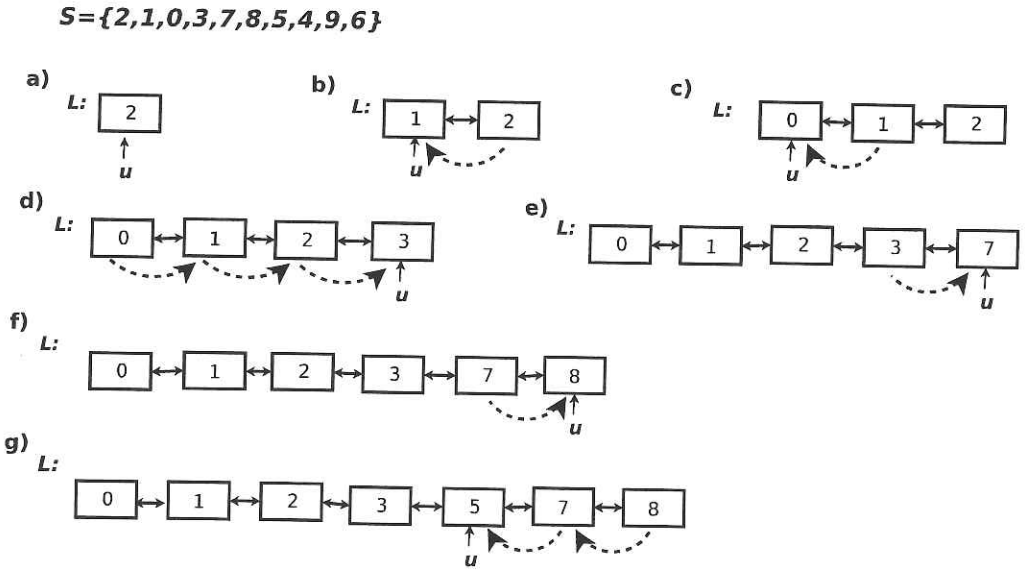


Figura 4.6: Ejemplo de Local Insertion Sort

Para entender mejor el proceso, consideremos un ejemplo. Sea la secuencia a ordenar $S = \{2, 1, 0, 3, 7, 8, 5, 4, 9, 6\}$. La Figura 4.6 ilustra el procedimiento, observamos que en el inciso **(a)** se inicia insertando en la lista el primer elemento de la secuencia, el 2, y apuntamos con u a su localidad; se sigue el procedimiento para los demás incisos hasta llegar al inciso **(g)**. Aquí se observa la ventaja de mantener a u ya que al insertar a 5 se realizan sólo 3 comparaciones en vez las 5 que se harían al realizar la lista desde el inicio.

Del ejemplo podemos observar que dada la posición de u en la lista se, tienen dos casos.

1. Si el elemento es mayor, se compara con el de la derecha hasta encontrar su posición correcta (como ocurre al insertar 6) o hasta llegar al final de la lista, en cuyo caso se inserta al final (como es el caso del 9).
2. Si el dato es menor se compara con el de la izquierda hasta encontrar la localidad que le corresponde o hasta llegar al inicio de la lista, en tal caso se insertara al inicio (como se observa al insertar 0).

Nótese que si la entrada se encuentra ordenada esta técnica tendrá un desempeño lineal ya que al insertar cada elemento hará sólamente una comparación, pero si la lista se encuentra muy desordenada, podría requerir recorrer toda la secuencia para insertar cada nuevo elemento, en este caso tendrá un desempeño de $O(n^2)$.

Análisis de Complejidad

Para esta técnica existen diferencias en el desempeño computacional dependiendo de cómo sea la secuencia de entrada, es decir qué tan desordenada esté. Para determinar el tiempo de ejecución, sea d_x la distancia entre la localidad del último elemento insertado y la que ocuparía el nuevo elemento a insertar x. La Figura 4.7 ilustra este desplazamiento.

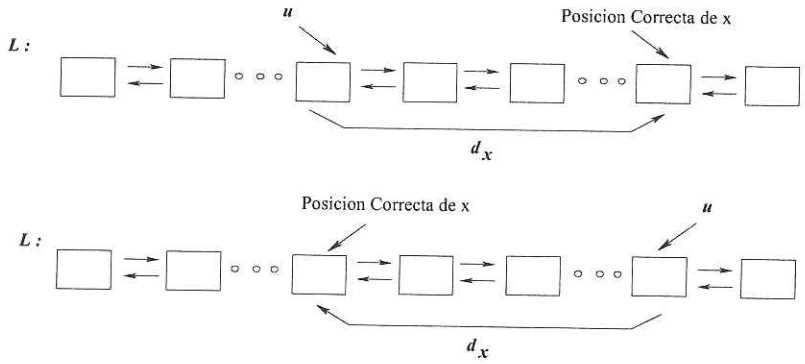


Figura 4.7: Desplazamiento d_x en Inserción local

Sea $t(\mathcal{L}.Insert(x); q)$ el tiempo que toma insertar x en una lista de tamaño q. Dado que agregar un elemento a la lista biligada toma tiempo constante, entonces el tiempo para insertar un elemento es: $t(\mathcal{L}.Insert(x); q) = d_x$; es decir, únicamente depende del número de comparaciones realizadas para encontrar la posición correcta de x en la lista.

Sea S la secuencia de tamaño n a ordenar, entonces el tiempo de ejecución del LIS, ordenamiento por inserción local, queda definido por:

$$T_{LIS}(S) = \sum_{q=1}^{n} t(\mathcal{L}.Insert(x); q)$$

Considerando lo anterior, podemos ahora analizar los escenarios correspondientes al mejor y peor caso así como en el caso esperado.

Mejor caso

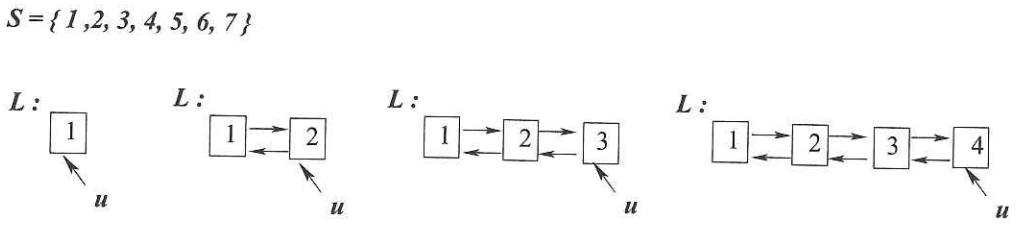


Figura 4.8: Mejor caso de Inserción local

Éste ocurre cuando la secuencia S está ordenada, ya sea ascendente o descendentemente. Sin pérdida de generalidad, supongamos que la lista está ordenada en forma ascendente. Al tomar un elemento x de S para insertarlo en la lista biligada, L, se tiene que la posición correcta de x es contigua a la posición a la que apunta u; es decir, x debe ser insertado justo a la derecha de u. La Figura 4.8 ilustra las primeras iteraciones de un ejemplo para el mejor caso.

Ahora bien, tenemos que: $d_x = 1$ y $t(\mathcal{L}.Insert(x); q)$ es $O(1), \forall x \in S$. Por lo tanto,

$$T_{LIS}(S) = \sum_{q=1}^{n} t(\mathcal{L}.Insert(x); q) \in O(n).$$

Podemos concluir que para el mejor caso el desempeño computacional del LIS es de $O(n)$; es decir, lineal.

Peor caso

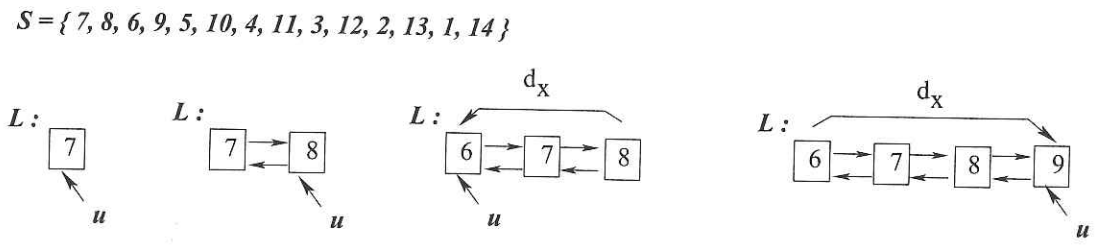


Figura 4.9: Peor caso de Inserción local

Este caso sucede cuando cada elemento a insertar requiere ser comparado con todos los elementos en la lista biligada; es decir, se debe recorrer toda la lista para insertar al nuevo dato. La secuencia $S = \{7, 8, 6, 9, 5, 10, 4, 11, 3, 12, 2, 13, 1, 14\}$ resulta ser un ejemplar para el peor caso. S tiene un patrón de *zigzag*; es decir, cada nuevo elemento insertado es mayor o menor a todos los que se encuentran en la lista. La Figura 4.9 ilustra las primeras iteraciones del proceso para este ejemplar S.

Entonces, para el peor caso, tenemos que $d_x = q, \forall x \in S$ y $t(\mathcal{L}.Insert(x); q)$ es $O(q), \forall x \in S$. Por lo tanto, el tiempo de ejecución para el algoritmo de inserción local, LIS, queda determinado por:

$$T_{LIS}(S) = \sum_{q=1}^{n} t(\mathcal{L}.Insert(x); q) = \sum_{q=1}^{n} q = \frac{n(n-1)}{2} \quad \text{es } O(n^2)$$

Podemos concluir que para el peor el desempeño computacional del LIS es de $O(n^2)$; es decir, cuadrático.

Caso Esperado

Para determinar el desempeño computacional del LIS en el caso esperado, antes que nada es necesario suponer que los elementos en la secuencia fueron obtenidos mediante una distribución uniforme, de esta manera podemos decir que la probabilidad de que el apuntador u esté situado en cualquiera de las posiciones, en una lista de tamaño q, es la misma para todos y, ésta es $(1/q)$. Además, la probabilidad de que el nuevo elemento a insertar le corresponda una determinada localidad en la lista es la misma para todas y es $(1/(q+1))$, no olvidemos que son $(q+1)$ posibles localidades para el nuevo elemento x.

Ahora bien, para determinar el caso esperado, requerimos calcular la Esperanza del tiempo de ejecución del algoritmo LIS, esto es:

$$E[T_{LIS}(S)] = E\left[\sum_{q=1}^{n} t(\mathcal{L}.Insert(x); q)\right] = \sum_{q=1}^{n} E\left[t(\mathcal{L}.Insert(x); q)\right]$$

El apuntador puede estar en cualquiera de las q posiciones en la lista y x puede ser insertado en cualquiera de las $(q+1)$ posibles localidades. Si para recorrer las posibles localidades donde se encuentre el apuntador usamos al índice j y para recorrer las posibles localidades donde insertar a x usamos a k, entonces a d_x se le puede expresar como $\mid j-k \mid$. Ahora definimos los siguientes eventos:

A_j = el apuntador u está en la posición j de la lista biligada;

B_k = el nuevo elemento x debe ser insertado en la posición k de la lista.

Así, tenemos que:

$Prob[A_j]$ es la probabilidad de que el apuntador u se encuentre en la posición j; y

$Prob[B_k]$ representa la probabilidad de que el elemento x sea insertado en la posición k.

Entonces, podemos reescribir el tiempo esperado para insertar x, como:

$$E\left[t(\mathcal{L}.Insert(x); q)\right] = \sum_{j=1}^{q} \sum_{k=0}^{q} Prob[A_j] \cdot Prob[B_k] \cdot \mid j-k \mid$$

$$= \sum_{j=1}^{q} \sum_{k=0}^{q} \left(\frac{1}{q}\right)\left(\frac{1}{q+1}\right) \cdot \mid j-k \mid \; = \; \left(\frac{1}{q\,(q+1)}\right) \sum_{j=1}^{q} \left(\sum_{k=0}^{q+1} \mid j-k \mid\right)$$

$$= \left(\frac{1}{q\,(q+1)}\right) \sum_{j=1}^{q} (\mid j \mid + \mid j-1 \mid + \mid j-2 \mid + \ldots + \mid j-q \mid)$$

La forma de estas sumas al variar j es:

$$\begin{array}{ccccccccc} 1 & + & 0 & + & 2 & + & \ldots & + & (q-1) \\ 2 & + & 1 & + & 0 & + & \ldots & + & (q-2) \\ \vdots & & \vdots & & \vdots & & \ddots & & \vdots \\ q & + & (q-1) & + & (q-2) & + & \ldots & + & 0 \end{array}$$

Para calcular el valor total se debe acumular el resultado de las q sumas. Una forma de hacerlo es considerando la suma por columna. Tenemos que la suma de la primer columna es $\sum_{i=1}^{q} i$; de la segunda en adelante se simplifican como: $\sum_{i=1}^{q-1} i$. Por lo tanto, tenemos:

$$\sum_{j=1}^{q} \mid j-q \mid \leq q\left(\sum_{i=1}^{q} i\right) = q \cdot \left(\frac{q \cdot (q+1)}{2}\right).$$

Así, considerando el resultado anterior obtenemos que:

$$\frac{1}{q \cdot (q+1)}\left(\sum_{j=1}^{q} \mid j-q \mid\right) \leq \frac{1}{q \cdot (q+1)}\left(q \cdot \frac{(q+1) \cdot q}{2}\right) \leq \frac{q}{2}.$$

Hasta el momento lo único que sabemos es que el tiempo esperado para insertar **un** elemento en una lista ordenada de tamaño q, de tal forma que ésta permanezca ordenada, es, al menos, $q/2$. Entonces el tiempo que se espera emplear para ordenar, una secuencia de tamaño n, usando esta técnica es la suma de las inserciones de los n elementos, lo que podemos escribir como:

$$E\left[\sum_{q=0}^{n} t(\mathcal{L}.Insert(x); q)\right] = \sum_{q=0}^{q=n} E\left[t(\mathcal{L}.Insert(x); q)\right] \leq \sum_{q=0}^{n} \frac{q}{2}$$

$$\leq \frac{1}{2}\sum_{q=0}^{n} q \leq \frac{1}{2}\frac{n\,(n+1)}{2} \leq \frac{n\,(n+1)}{4} \in O(n^2)$$

Lo cual nos indica que el desempeño computacional en el caso promedio, para el algoritmo de Inserción Local es cuadrático.

Algoritmo

Para generar el pseudocódigo de esta técnica suponemos que la secuencia está almacenada en un arreglo A, además se emplea como estructura auxiliar una lista biligada L, que es donde realmente se ordena a los elementos del arreglo y se usa un apuntador ui al último elemento insertado. El Listado 15 presenta el código.

Listado 15 pseudocódigo Local Insertion Sort

```
//PreC: La secuencia es finita, no vacia y esta contenida en un arreglo A.
//PostC: Regresa al arreglo A ordenado.

localinsertionsort(array A;int n){
   bi-list  L;                          // Lista bi-ligada
   pointer  ui;                         // apuntador a un elemento en L
   int       i;                         // contador

   L.create;                            // Crea la lista
   L.insert(A[1]);                      // agrega al primer elemento de A
   ui= L;                               // apunta al unico dato en L

    for ( i=1; i<=n; i++ ) {
       if (ui.dato < A[i]) then         // busca a partir del ultimo
          L.insert_right(ui, A[i]);     // insertalo a la derecha de ui
       else
          L.insert_left(ui, A[i]);      // insertalo a la izquierda de ui
    }// end for i

    i=1;  p= L.primerdato;
    while not ( L.empty ) do {          // regresa los datos ordenados a A
        A[i] = p.dato;
          p  = p.siguiente;
    }// end while
}//end lis
```

En el pseudocódigo se emplea el procedimiento insert_right que busca, moviéndose hacia la derecha de la lista, la posición correcta del nuevo elemento, cuando la encuentra, lo inserta en la lista. El procedimiento insert_left hace lo correspondiente, moviéndose hacia la izquierda en la lista biligada.

4.5. Ordenamiento de Shell

A esta técnica de ordenamiento se le considera como una mejora al método de ordenamiento por inserción, dado que emplea el hecho de que cuando la secuencia está casi ordenada insertion sort es eficiente. Fue creada por Donal L. Shell.

Estrategia

La técnica consiste en dividir a la secuencia en h subsecuencias y ordenar por inserción cada una de ellas, esto se hace sucesivamente disminuyendo en cada ocasión el valor de h hasta llegar a $h = 1$, lo que significaría que estamos empleando el ordenamiento por inserción.

Con el procedimiento descrito se logra es comparar elementos que no necesariamente son contiguos. Despúes de cada ordenamiento de las subsecuencias se tiene una secuencia $h-$ordenada; es decir, tenemos una secuencia compuesta por h subsecuencias ordenadas. Los puntos clave en este procedimiento son:

1. Toma menos comparaciones ordenar secuencias con pocos elementos, que con un gran número de elementos.
2. Requiere menos tiempo ordenar, usando el método por inserción, una secuencia casi ordenada que una muy desordenada.

Supongamos por simplicidad, y sin pérdida de generalidad, que tenemos una secuencia S, de tamaño $2k$, entonces si hacemos una partición a la secuencia en k subsecuencias, es decir $h = k$, cada subsecuencia tendrá 2 elementos, sean éstas:

$$S_1 = \{s_1, s_{h+1}\},\ S_2 = \{s_2, s_{h+2}\},\ S_3 = \{s_3, s_{h+3}\},\ \ldots,\ S_k = \{s_k, s_{2h}\}.$$

Es importante hacer notar que, estamos comparando elementos que se encuentran a distancia $h = k$, y las subsecuencias contienen pocos elementos por lo que el ordenamiento en cada una de ellas es rápido, como $(h > 1)$ repetimos el procedimiento pero incrementamos la cantidad de elementos en cada subconjunto, esto lleva, inevitablemente, a la reducción del número de subconjuntos. Ahora cada elemento comparado esta a distancia $h = k/2$ por lo que las subsecuencias[1] son: $S_1 = \{s_1, s_{h+1}, s_{k+1}, s_{(k+1)+h}\}$,

$$S_2 = \{s_2, s_{h+2}, s_{k+2}, s_{(k+2)+h}\}\ ,\ \ldots\ ,\ S_{k/2} = \{s_h, s_k, s_{k+h}, s_{2k}\}\ .$$

Como en el paso anterior se llevo acabo un ordenamiento, se espera que los elementos en las subsecuencias tengan un cierto orden para así realizar pocas comparaciones. Se sigue con el proceso hasta que $h = 1$; es decir, tenemos la secuencia $S = \{s_1,\ s_2, \ldots, s_{2k}\}$ y, en este caso, simplemente aplicamos ordenación por inserción, obteniendo finalmente la secuencia ordenada.

A continuación ilustraremos el método, con un ejemplo. Sea S la secuencia a ordenar:

$$S = \{503, 87, 512, 61, 908, 170, 897, 275, 653, 426, 154, 509, 612, 677, 765, 703\}.$$

La Figura 4.10 presenta esquemáticamente la ejecución del algoritmo sobre S.

En el inciso **(a)**, la secuencia original es dividida en 8 subsecuencias ya que $i = 1$ y $h = 16/2 = 8$; también se presentan estas subsecuencias después de haber sido ordenadas. En **(b)** se tiene la secuencia resultante del primer ordenamiento, las nuevas subsecuencias en las que se divide son 4, pues $i = 2$ y $h = 4$; además se muestran estas secuencias después de haberse ordenado. Análogamente se trabaja en **(c)**. Finalmente, en **(d)** donde se obtiene la secuencia totalmente ordenada. Nótese que los elementos que forman las sublistas están a distancia h y después al reunir la secuencia la distancia entre estos elementos no se altera, por lo cual cuando $h = 1$ la secuencia está casi ordenada, aquí se emplea, en realidad, el ordenamiento por inserción.

[1] Aquí, s_i representa al dato en la posición i de la secuencia S resultante de la iteración anterior.

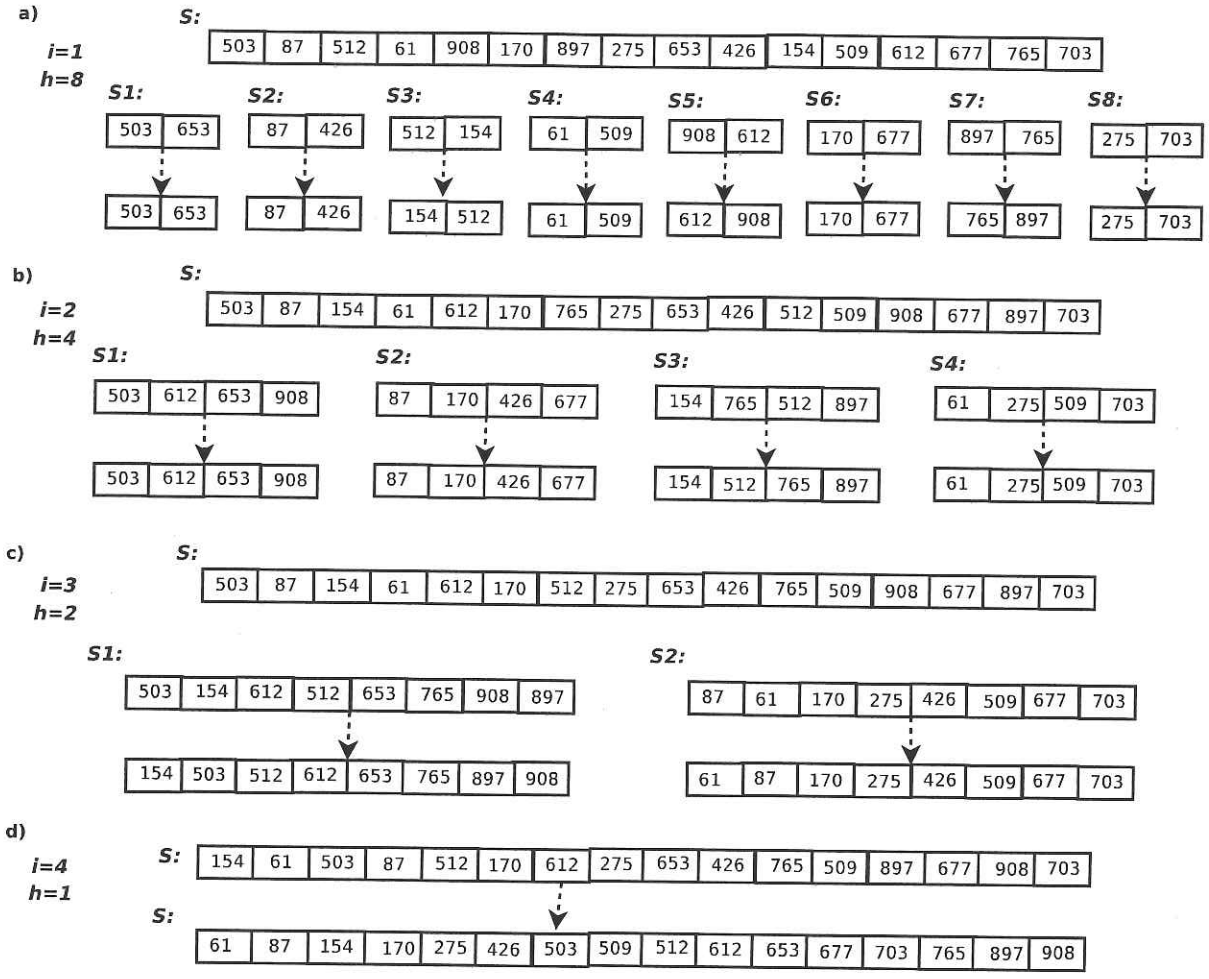


Figura 4.10: Ejemplo de Shell Sort

Análisis de Complejidad

Existen diversas secuencias para los incrementos h, por lo tanto, se pueden tener diferentes tiempos de ejecución para el algoritmo, por lo cual no es posible realizar un análisis puntual. Además, algunos factores que influyen sobre el comportamiento del algoritmo shell sort son los siguientes:

1. El tamaño de la secuencia.
2. El número de pasadas o el número de incrementos (secuencia escogida).
3. La suma de los incrementos.
4. El número de comparaciones.
5. La cantidad de movimientos realizados, es decir, la cantidad de inversiones en las subsecuencias.

Tomaremos dos formas clásicas de generar secuencias de incrementos para realizar el análisis para el algoritmo Shell Sort: los incrementos de Shell y los de Hibbard.

Incrementos de Shell

El algoritmo Shell Sort utiliza una secuencia de incrementos: $h_1, h_2, ..., h_t$, donde t toma al menos el valor de 1, el algoritmo resulta ser mejor para valores mayores. Después de una fase, usando un incremento h_k, para cada i se tiene que $A[i] \leq A[i + h_k]$, donde esto tenga sentido. Todos los elementos separados h_k localidades se encuentran ordenados. Una secuencia con esta propiedad es llamada secuencia h_k−ordenada. La siguiente figura muestra un arreglo después de aplicar varias fases del algoritmo Shell Sort.

Original	**81**	**94**	**11**	**96**	**12**	**35**	**17**	**95**	**28**	**58**	**41**	**75**	**15**
5−Orden	35	17	11	28	12	41	75	15	96	58	81	94	95
3−Orden	28	12	11	35	15	41	58	17	94	75	81	96	95
1−Orden	11	12	15	17	28	35	41	58	75	81	94	95	96

La estrategia general para un h_k−ordenamiento, para cada posición i, $h_{k+1}, h_{k+2}, ..., n$ consiste en colocan los elementos en las posiciones correctas entre i, $(i - h_k)$ y $(i - 2h_k)$. Un cuidadoso análisis muestra que la acción de un h_k−ordenamiento consiste en ejecutar un Insertion Sort sobre h_k subarreglos independientes. Esta observación será importante cuando se calcule el tiempo de ejecución del algoritmo. Shell propuso la siguiente secuencia de incrementos: $h_t = \lfloor n/2 \rfloor$ y $h_k = \lfloor h_{k+1}/2 \rfloor$. Por ejemplo, si $n = 16$, se tiene la secuencia: $h_4 = 8, h_3 = 4, h_2 = 2, h_1 = 1$. Lamentablemente, esta secuencia de incrementos induce un pobre desempeño computacional del algoritmo, como lo muestra el siguiente resultado,

Teorema 4.1 El tiempo de ejecución del algoritmo Shell Sort, en el peor de los casos, utilizando la secuencia de incrementos de Shell, es $\Theta(n^2)$.

Demostración. La prueba requiere mostrar no sólo una cota superior sobre el peor de los casos, sino también que existe un ejemplar que realmente toma tiempo $\Omega(n^2)$. Probaremos primero la cota inferior, construyendo un ejemplar malo. Sea n una potencia de 2, para facilitar los cálculos. Sea A el arreglo de entrada de longitud n, con los $n/2$ elementos más grandes en las posiciones pares y los $n/2$ elementos más pequeños en las posiciones impares. Como todos los incrementos, excepto el último son pares, cuando llegamos al último paso, los $n/2$ elementos más grandes aún están en las posiciones pares y los $n/2$ elementos más pequeños en las impares. El i−ésimo elemento más pequeño, $i \leq n/2$, está en la posición $(2i - 1)$ después del inicio del último paso. Colocar al i−ésimo elemento en su posición correcta requiere moverlo $(i - 1)$ localidades en el arreglo. Así que poner a los $n/2$ elementos más pequeños en su posición correcta requiere al menos $\sum_{i=1}^{n/2}(i - 1)$ lo cual es $\Omega(n^2)$.

Para finalizar la prueba, mostraremos que la cota superior es $O(n^2)$. Como se había observado anteriormente, un paso cuyo incremento es h_k consiste de h_k ejecuciones del algoritmo Insertion Sort con n/h_k elementos.

Como el algoritmo Insertion Sort es de orden cuadrático, el costo total de un paso resulta ser $O(h_k \cdot (n/h_k)^2) = O(n^2/h_k)$. Sumando todos los pasos obtenemos:

$$\sum_{i=1}^{k}(n^2/h_k) = n^2 \sum_{i=1}^{k}(1/h_k) < 2 \cdot n^2 \text{ ya que } \sum_{i=1}^{k}(1/h_k) < 2$$

Por lo tanto, la cota superior es: $O(n^2)$.
Finalmente, podemos concluir que el tiempo de ejecución del algoritmo Shell Sort, en el peor de los casos, y utilizando los incrementos de Shell es de $\Theta(n^2)$.

La Figura 4.11 muestra una lista que es un ejemplar malo para el algoritmo Shell Sort, aunque éste no resulta ser el peor caso, el último paso toma tiempo considerable.

Original	**1 9 2 10 3 11 4 12 5 13 6 14 7 15 8 16**
8–Orden	1 9 2 10 3 11 4 12 5 13 6 14 7 15 8 16
4–Orden	1 9 2 10 3 11 4 12 5 13 6 14 7 15 8 16
2–Orden	1 9 2 10 3 11 4 12 5 13 6 14 7 15 8 16
1–Orden	1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

Figura 4.11: Ejemplar malo para Shell Sort

Incrementos de Hibbard

El problema con la secuencia de incrementos de Shell es que los incrementos consecutivos no necesariamente son primos relativos, por lo cual los incrementos pequeños pueden tener poco efecto. Hibbard sugirió un leve, pero significativo, cambio para la secuencia de incrementos de Shell, el cual dá un mejor resultado tanto práctica como teóricamente. La secuencia de incrementos es de la forma: $1, 3, 7, ..., 2^{k-1}$. La diferencia clave es que los incrementos consecutivos no tiene factores comunes. La Figura 4.12 muestra la misma secuencia que la Figura 4.11, pero utilizando la secuencia de incrementos de Hibbard.

Original	**1 9 2 10 3 11 4 12 5 13 6 14 7 15 8 16**
15–Orden	1 9 2 10 3 11 4 12 5 13 6 14 7 15 8 16
7–Orden	1 5 2 6 7 4 8 9 13 10 14 11 15 12 16
3–Orden	1 3 2 4 5 7 6 8 9 11 10 12 13 15 14 16
1–Orden	1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

Figura 4.12: Shell Sort, usando incrementos de Hibbard

Teorema 4.2 El tiempo de ejecución del algoritmo Shell Sort, en el peor de los casos, utilizando la secuencia de incrementos de Hibbard, es $\Theta(n^{3/2})$.

Demostración. Sólo probaremos la cota superior. La prueba requiere de algunos resultados de Teoría de Números.

Para determinar la cota superior necesitamos acotar el tiempo de ejecución de cada paso y, entonces, sumar sobre todos los pasos. Para incrementos h_k con $h_k > n^{1/2}$, usaremos la cota $O(n^2/h_k)$ del Teorema 4.1. Aunque esta cota se satisface para los otros incrementos, es muy grande y no resulta muy útil. Intuitivamente, tomaremos ventaja del hecho de que estos incrementos son *especiales*. Necesitamos mostrar que para cada elemento A_p en la posición p, cuando se ejecute un h_k-ordenamiento, hay únicamente unos pocos elementos a la izquierda de p que son mayores a A_p.

Cuando llegamos a un h_k-ordenamiento, sabemos que el arreglo de entrada, ha sido $h_{k+1}-$ordenado y $h_{k+2}-$ordenado. A priori, el h_k-ordenamiento, considera elementos en las posiciones p y $(p-i)$, con $i \leq p$. Si i es múltiplo de h_{k+1} o de h_{k+2} entonces, claramente, $A[p-i] < A[p]$. De hecho, si i puede expresarse como una combinación lineal, de enteros no negativos, de h_{k+1} y h_{k+2} entonces $A[p-i] < A[p]$.

Ahora bien, $h_{k+2} = 2 \cdot h_{k+1} + 1$ por lo que h_{k+1} y h_{k+2} no tienen factores comunes. En este caso, es posible mostrar que todos los enteros que son al menos tan grandes como $(h_{k+1}-1)(h_{k+2}-1) = 8(h_k)^2 + 4 \cdot h_k$ puede ser expresado como una combinación lineal de h_{k+1} y h_{k+2}.

Así, la cantidad de incrementos que pueden ejecutarse, es a lo más $8 \cdot h_k + 4 = O(h_k)$ veces para cada una de las $n - h_k$ posiciones. Lo cual genera una cota de $O(n \cdot h_k)$ por paso. Usando el hecho de que casi la mitad de los incrementos satisface que $h_k < \sqrt{n}$ y suponiendo que t es par, entonces el tiempo total de ejecución es:

$$O\left(\sum_{k=1}^{t/2} n \cdot h_k + \sum_{k=t/2+1}^{t} n^2/h_k\right) = O\left(n \cdot \sum_{k=1}^{t/2} h_k + n^2 \cdot \sum_{k=t/2+1}^{t} 1/h_k\right)$$

$$= O(n \cdot h_{t/2}) + O\left(\frac{n^2}{h_{t/2}}\right) = O(n^{3/2})$$

Esto se simplificó ya que ambas sumas son series geométricas y $h_{t/2} = \Theta(\sqrt{n})$

En la demostración anterior, se afirmó que si i puede expresarse como una combinación lineal, de enteros no negativos, de h_{k+1} y h_{k+2} entonces $A[p-i] < A[p]$. Ilustramos un ejemplo: cuando aplicamos un $3-$ordenamiento, ya se han realizado un $7-$ordenamiento y un $15-$ordenamiento. Tenemos que 52 puede expresarse como una combinación lineal de 7 y 15: $52 = 1 \cdot 7 + 3 \cdot 15$. Así, $A[100]$ no puede ser mayor que $A[152]$, pues se tiene que: $A[100] \leq A[107] \leq A[122] \leq A[137] \leq A[152]$.

Análisis del Caso Promedio

Lo primero a establecer es la cantidad de comparaciones que se realizarán, y éstas sin importar el caso específico quedan determinadas por el número de inversiones eliminadas en un $h-$ordenamiento anterior. Antes de proseguir es necesario establecer que una **inversión** es la pareja (s_i, s_j) tal que $(s_i > s_j)$, con $i < j$. Entonces el número de inversiones existentes en una secuencia es la cantidad de parejas con esta forma que pueden existir en ella. Por ejemplo, si $S = \{3, 2, 4, 1\}$ se tienen cuatro inversiones que son: $(3,2)$, $(3,1)$, $(2,1)$ y $(4,1)$.

El primer caso a estudiar es cuando la secuencia de incrementos es $h_t = 2$ y $h_1 = 1$; es decir, se tienen dos incrementos. Para calcular la cantidad promedio de inversiones se hace uso del siguiente teorema.

Teorema 4.3 El número promedio de inversiones en una permutación $h-$ordenada de $\{1, 2, \ldots, n\}$ es:

$$f(n,h) = \frac{2^{2q-1} q! q!}{(2q+1)!} \left(\binom{h}{2} q(q+1) + \binom{r}{2}(q+1) - \frac{1}{2}\binom{h-r}{2} q \right)$$

donde $q = \lfloor n/h \rfloor$, $r = n \mod h$.

Demostración. Una permutación $h-$ordenada contiene r secuencias de longitud $(q+1)$ y $(h-r)$ de longitud q. Si cada inversión viene de un par distinto de subsecuencias, y dado que un par de distintas subsecuencia en una permutación $h-$ordenada define una permutación aleatoria $2-$ordenada, entonces el número de inversiones promedio es la suma del promedio de inversiones entre cada par de distintas subsecuencias, lo que se escribe como:

$$\binom{r}{2} \frac{A_{2q+2}}{\binom{2q+2}{q+1}} + r \cdot (h-r) \cdot \frac{A_{2q+1}}{\binom{2q+1}{q}} + \binom{h-r}{2} \cdot \frac{A_{2q}}{\binom{2q}{q}} = f(n,h),$$

donde A_n es el número total de inversiones sobre todas las permutaciones $2-$ordenadas del conjunto $\{1, 2 \ldots, n\}$ y $A_n = \lfloor n/2 \rfloor (2^n - 2)$.

Corolario 4.1 Si la secuencia de incrementos $h_t, \ldots, h_2, h_1$ satisface la condición $(h_{s+1} \mod h_s) = 0$ para $t > s \geq 1$, entonces el número promedio de movimientos es:

$$\sum_{t \geq s \geq 1} (r_s f(q_s + 1, h_{s+1}/h_s) + (h_s - r_s) f(q_s, h_{s+1}/h_s)),$$

donde $r_s = N \mod h_s$, $q_s = \lfloor N/h_s \rfloor$, $h_{t+1} = N t_h$ y f es la función definida en el Teorema 4.3.

Demostración. El proceso de $h-$ordenamiento consiste de un ordenamiento por inserción lineal en $r_s \cdot (h_{s+1}/h_s)-$ordenada secuencias de longitud $q_s + 1$ y en $(h_s - r_s)$ de longitud q_s. La condición de divisibilidad implica que cada una de estas subsecuencias es una permutación aleatoria $(h_{s+1}/h_s)-$ordenada, en este sentido cada una de las permutaciones es igualmente probable, entonces podemos asumir que la entrada original fue una permutación aleatoria de distintos elementos.

El corolario anterior siempre se satisface para Shell Sort cuando los incrementos son h y 1. Si $q = \lfloor N/h \rfloor$ y $r = N \mod h$ la cantidad promedio de inversiones es:

$$r \cdot f(q+1, N) + (h-r) \cdot f(q, N) + f(N, h)$$

$$= \frac{r}{2} \cdot \binom{q+1}{2} + \frac{q+1}{2} \cdot \binom{q}{2} + f(N, h)$$

Con las aproximaciones :

$$f(N, h) = \left(\sqrt{\pi}/8\right) n^{3/2} h^{1/2} \quad \text{y} \quad h = 1.72\sqrt[3]{N}$$

obtenemos que el desempeño computacional, en el caso promedio es: $O(N^{5/3})$.

Para finalizar daremos el desempeño de shell sort al usar otras secuencias de incremento, así de esta forma cuando se emplea la secuencia 8−ordenamiento, 4−ordenamiento, 2−ordenamiento, 1−ordenamiento; es decir el empleado en el ejemplo del funcionamiento de shell sort, obtenemos un desempeño de $O(N^{3/2})$. El desempeño anterior también es obtenido cuando la secuencia de incrementos tiene la forma $h_s = 2^s - 1$, $1 < s \leq t = \lfloor \log N \rfloor$. Mientras que si la secuencia tiene la forma $2^p 3^q$ tal que este valor es menor a N se tiene un desempeño de $O(N (\log N)^2)$.

Algoritmo

Para escribir el pseudocódigo se supone que los datos están contenidos en un arreglo y se genera la secuencia de incrementos de la forma $h_i = 2^i$. El Listado 16 presenta un pseudo-código para esta versión. Como se puede observar en este código, no dividimos la secuencia de entrada en $n/2^i$ subsecuencias, lo que se hace es indexar los elementos en la secuencia de tal manera que se puedan ordenar aquellos que se encuentran a distancia h, para lograr esto hacemos uso de los ciclos for y while, con el primer ciclo lo que establecemos es el equivalente a formar las subsecuencias con elementos a distancia h, mientras que con el segundo los ordenamos.

Listado 16 pseudocódigo Shell Sort

```
//PreC: la secuencia esta contenida en un arreglo no vacio y finito
//PostC: la secuencia esta ordenada

Shell_Sort(array A; int n){
 int  i, j, h, v;
  h=n/2;

  while (h>0) do {
    for (i=h;i<n;i++) {
       v = A[i];
       j = i;
       while (j>=h && A[j-h]>v) do {
          A[j] = A[j-h];
          j = j-h;
       }//end while

       A[j] = v;
    }// end for

    h = h/2;
  }//end while
}//end Shell
```

Capítulo 5

Algoritmos de Ordenamiento con desempeño $O(n \log n)$

En este capítulo presentamos cuatro algoritmos de ordenamiento cuyo desempeño computacional, en el peor de los casos, es $O(n \log n)$: Merge, Quick, Heap y Tree Sort.

5.1. Merge Sort

Este método es también conocido como ordenamiento por mezcla, en esta técnica se aprecia muy claramente el uso de la estrategia divide y vencerás. Este algoritmo es considerado estable y en ocasiones requiere de memoria adicional dependiendo de la manera en que sea implementado.

Estrategia

Usando la estrategia divide y vencerás es claro observar los pasos por medio de los cuales se resuelve el problema:

> Divide.- Particiona el ejemplar original en ejemplares más pequeños, de manera recursiva, hasta reducirlo a ejemplares de tamaño uno.
> Vence.- Resuelve para todos los ejemplares, iniciando con los de tamaño uno, continuando con los de tamaño 2, en cada paso va incrementando (duplicando) el tamaño del ejemplar hasta llegar al tamaño original.
> Mezcla.- Combina las soluciones de los problemas, aplicadas a ejemplares pequeños para resolver problemas con ejemplares más grandes, de manera recursiva, hasta obtener la solución del problema original.

La idea de esta estrategia se basa en los siguientes argumentos:

a) Toda secuencia de tamaño uno está ordenada.
b) Mezclar dos secuencias ordenadas para obtener una tercera secuencia ordenada no es complicado y no requiere muchas comparaciones.

Como ya mencionamos, en el paso divide partimos al ejemplar en otros más pequeños, pero no indicamos cómo hacer tales particiones. La estrategia consisten en ir dividiendo siempre a la mitad, de tal forma que cada vez que se realiza este paso se obtienen dos subsecuencias cuyo tamaños difieren a lo más en 1.

Para ilustrar el método consideremos el ejemplo de la Figura 5.1. Se desea ordenar la secuencia $S = \{1, 38, 27, 8, 43, 12, 3, 9, 82, 10\}$.

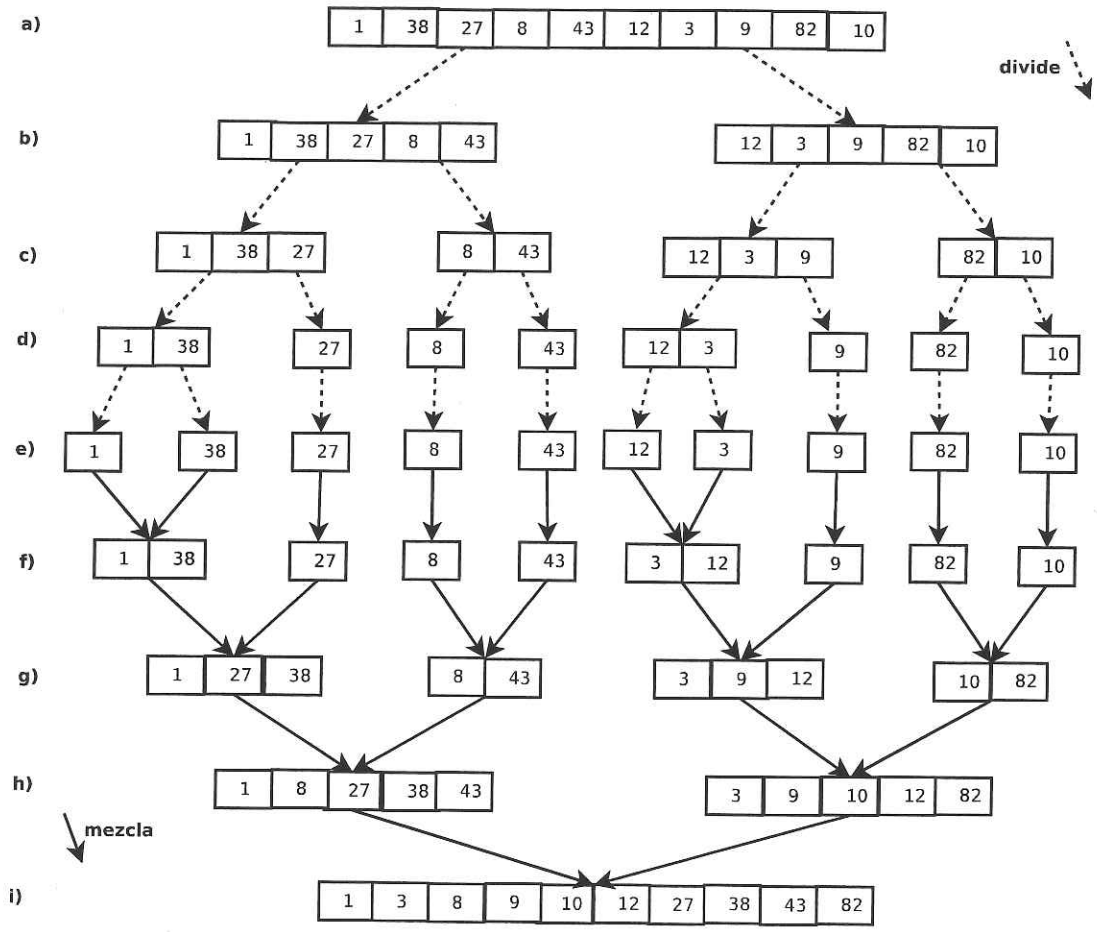


Figura 5.1: Ejemplo de Merge Sort

En el ejemplo podemos ver de forma clara como se lleva acabo el proceso divide, de manera recursiva, esto lo apreciamos de los incisos (**a**) al (**e**) en donde se obtiene subsecuencias de tamaño uno y a partir de inciso (**f**) se puede observar el proceso mezcla en el cual es donde se lleva acabo el ordenamiento.

Análisis de Complejidad

Para establecer el desempeño computacional es necesario primero conocer el desempeño del proceso mezcla, ya que en esta parte se efectúan las comparaciones. La cantidad de comparaciones realizadas aquí dependerá del número de elementos en las secuencias a mezclar. Revisaremos el mejor y peor caso de este proceso.

Una vez determinada la cantidad de comparaciones realizadas por mezcla se regresará al análisis de merge sort. Cabe aclarar que en el análisis se supone que la secuencia a ordenar es de tamaño n.

Sean S_A y S_B dos secuencias ordenadas con N_A y N_B elementos, respectivamente, en las cuales se aplicará el proceso mezcla.

En el mejor caso todos los elementos de una secuencia son menores al primero de la otra, supongamos, sin pérdida de generalidad, que tal secuencia es S_A, entonces el número de comparaciones realizadas es de N_A ya que sólo se compara al primer elemento de S_B con todos los elementos de S_A. La secuencia resultante, en este caso, se obtiene al agregar al final de S_A a S_B.

Para el peor caso, se tiene que $a_i < b_j, \forall a_i \in S_A$, $b_j \in S_B$, $1 \leq i \leq N_A$, $1 \leq j \leq N_B$ y $j = i$; es decir, los elementos de las secuencias S_A y S_B están intercalados, esto nos indica que para este caso se realizan $N_A + N_B - 1$ comparaciones. Como ambas secuencias están ordenadas, las comparaciones son directas. Por lo que podemos decir que el rango de comparaciones realizadas está entre N_A, el mejor caso, y $(N_A + N_B - 1)$, en el peor caso. Por lo tanto, el desempeño computacional, en el peor de los casos, para el proceso mezcla es lineal, ya que $(N_A + N_B - 1)$ es $O(n)$.

Peor caso

Para calcular el peor de los caso, tomaremos como referencia el desempeño computacional, para el peor caso del procedimiento mezcla y supondremos que en cada llamada recursiva siempre se tiene éste.

Como se está considerando a esta técnica de forma recursiva, el proceso divide se realiza hasta que las secuencias obtenidas son de tamaño uno y ya no es posible dividirlas. Ahora si a estas secuencias les aplicamos el proceso mezcla para obtener secuencias de tamaño uno es fácil ver que no hubo ninguna comparación, por ello en el peor caso de mezcla para obtener secuencias de tamaño uno se realizan cero comparaciones, lo que denotaremos como $W(1) = 0$.

Sea $w(n)$ el número de comparaciones realizadas en el peor caso para secuencias de tamaño n, entonces podemos establecer de forma recursiva el número de comparaciones realizadas por merge sort para su peor caso si consideramos lo siguiente: por un lado, sabemos que el proceso divide parte a la secuencia siempre en dos subsecuencias cuyo tamaño difiere en a lo más un elemento. Entonces, sin pérdida de generalidad y por simplicidad de cálculo, podemos decir que el tamaño de las subsecuencias al aplicar divide es de $n/2$. Por otro lado, en el peor caso, el proceso mezcla realiza $N_A + N_B - 1$ comparaciones, esto es: $(\frac{n}{2} + \frac{n}{2} - 1)$ comparaciones. Sin embargo, éstas representan sólo las últimas realizadas

debido a que el proceso mezcla se lleva acabo de manera inversa al proceso divide, por ello hay que agregar las comparaciones realizadas para obtener las secuencias de tamaño $n/2$ y dado que consideramos el peor caso esta cantidad es:

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

Dado que la técnica se está considerando desde un punto de vista recursivo no basta con considerar únicamente a las comparaciones realizadas para obtener las dos últimas secuencias ordenadas a mezclar sino que también hay que tomar en cuenta todas aquellas realizadas desde las secuencias de tamaño uno. Por ello el número de comparaciones total realizadas se puede escribir de manera recursiva teniendo en cuenta a divide de la siguiente forma:

$$\begin{aligned}
w(n) &= 2W(n/2) + (n-1) \\
&= 2(2W(n/4) + (n/2) - 1) + (n-1) \\
&= 4W(n/4) + (n-2) + (n-1) \\
&= 4(2W(n/8) + (n/4) - 1) + (n-2) + (n-1) \\
&= 8W(n/8) + (n-4) + (n-2) + (n-1) \\
&= 8(2W(n/16) + (n/8) - 1) + (n-4) + (n-2) + (n-1) \\
&= 16W(n/16) + (n-8) + (n-4) + (n-2) + (n-1) \\
&= 16(2W(n/32) + (n/16) - 1) + (n-8) + (n-4) + (n-2) + (n-1) \\
&= 32W(n/32) + (n-16) + (n-8) + (n-4) + (n-2) + (n-1) \\
&\vdots
\end{aligned}$$

Se continúa de forma análoga, hasta que llegar a $W(1)$ el cual tiene valor de cero, por lo que el primer término desaparece. Para obtener la cantidad de comparaciones realizadas de manera clara conviene reescribir el resultado anterior de la siguiente forma:

$$\begin{aligned}
w(n) &= 2^1 \cdot W(n/2^1) + n \cdot (\log_2 2^1) - 2^0 \\
&= 2^2 \cdot W(n/2^2) + n \cdot (\log_2 2^2) - \textstyle\sum_{i=0}^{1} 2^i \\
&= 2^3 \cdot W(n/2^3) + n \cdot (\log_2 2^3) - \textstyle\sum_{i=0}^{2} 2^i \\
&= 2^4 \cdot W(n/2^4) + n \cdot (\log_2 2^4) - \textstyle\sum_{i=0}^{3} 2^i \\
&= 2^5 \cdot W(n/2^5) + n \cdot (\log_2 2^5) - \textstyle\sum_{i=0}^{4} 2^i \\
&\vdots \\
&= n \cdot (W(1)) + n \cdot (\log_2 n) - \textstyle\sum_{i=0}^{(\log_2 n)-1} 2^i
\end{aligned}$$

Este último término se obtiene si consideramos que el proceso divide se detiene cuando las secuencias obtenidas tienen tamaño uno, por lo que tenemos n secuencias de tamaño uno y los dos términos siguientes al observar el patrón de los casos anteriores. Dado que el primer término se hace cero, pues $W(1) = 0$ tenemos que:

$$\begin{aligned}
w(n) &= n \cdot W(1) + n \cdot (\log_2 n) - \textstyle\sum_{i=0}^{(\log n)-1} 2^i \\
&= n \cdot (0) + n \cdot (\log_2 n) - \textstyle\sum_{i=0}^{(\log n)-1} 2^i \\
&= n \cdot (\log_2 n) - \textstyle\sum_{i=0}^{(\log n)-1} 2^i
\end{aligned}$$

Sabemos que $\sum_{i=0}^{k} 2^i = 2^{k+1} - 1$, entonces tenemos que:

$$n \cdot (\log_2 n) - \sum_{i=0}^{(\log_2 n)-1} 2^i = n \cdot (\log_2 n) - 2^{\log n} + 1 = n \cdot (\log_2 n) - n + 1.$$

Lo que implica un desempeño $w(n) = O(n \log n)$ para el peor caso de Merge sort.

Mejor caso

En este escenario nuevamente tomamos como punto de partida el número de comparaciones realizadas por mezcla para el mejor caso y supondremos que en las llamadas recursivas siempre obtenemos éste. Usaremos la notación $B(n)$ para denotar el mejor caso de mezcla y $b(n)$ para el mejor caso de merge sort.

Para que mezcla genere una lista de tamaño uno no se requiere ninguna comparación, por lo cual $B(1) = 0$. Realizando un procedimiento similar al efectuado para el peor caso tenemos:

$$\begin{aligned} b(n) &= 2B(n/2) + (n/2) \\ &= 2(2B(n/4) + (n/4)) + (n/2) \\ &= 4B(n/4) + (2n/2) \\ &= 4(2B(n/8) + (n/8)) + n \\ &= 8B(n/8) + (3n/2) \\ &= 8(2B(n/16) + (n/16)) + (3n/2) \\ &= 16B(n/16) + (4n/2) \\ &\vdots \end{aligned}$$

Para apreciar más claramente el patrón que sigue esta secuencia reescribimos como:

$$\begin{aligned} b(n) &= 2^1 \cdot B(n/2^1) + ([n(\log_2 2^1)]/2) \\ &= 2^2 \cdot B(n/2^2) + ([n(\log_2 2^2)]/2) \\ &= 2^3 \cdot B(n/2^3) + ([n(\log_2 2^3)]/2) \\ &= 2^4 \cdot B(n/2^4) + ([n(\log_2 2^4)]/2) \\ &\vdots \\ &= nB(1) + ([n(\log n)]/2) \end{aligned}$$

El primer término se hace cero: $B(1) = 0$, entonces tenemos que $b(n) = ([n(\log n)]/2)$, así que en el mejor caso merge sort tiene un $O(n \log n)$.

De esta manera podemos concluir que el Algoritmo Merge sort tiene un desempeño computacional de $O(n \log n)$ tanto para el peor como el mejor caso lo que lo convierte en un método de ordenamiento muy eficiente.

Listado 17 pseudocódigo Merge Sort

```
// PreC:  secuencia contenida en un arreglo S [1,N] y  S es no vacia.
//PostC:  regresa el arreglo que contiene a la secuencia ordenada

MergeSort {
   Divide_y_Mezcla (1, N)

  procedure Divide_y_Mezcla(int izq, der)
  // PreC:  izq, der  son los indices extremos del arreglo
  //PostC:  regresa el arreglo  ordenado
   int mitad;
   if ( izq < der )  {
     mitad = (izq + der) div 2;            // calcula la mitad
     Divide_y_Mezcla ( izq, mitad );       // aplica en el subarreglo izq
     Divide_y_Mezcla ( mitad, der );       // aplica en el derecho
     Mezcla ( izq, mitad, mitad+1, der ); // mezcla los subarreglos
   } //end if
  } // end Divide...

}//end MergeSort
```

Algoritmo

El Listado 17 presenta un pseudocódigo para la versión recursiva del Merge Sort, con un enfoque de arriba hacia abajo, *top-down*. El proceso Divide_y_Mezcla es el corazón del algoritmo, se encarga de aplicar la estrategia general: divide el arreglo en dos partes iguales hasta quedarse con subarreglos de tamaño 1 y luego, durante el regreso de la recursión, mezcla los subarreglos, hasta obtener el arreglo original ordenado. Cabe mencionar, que esta versión sólo trabaja con los índices, no es necesario tener arreglos auxiliares.

El sub-algoritmo Mezcla combina los dos sub-arreglos ordenados, *marcados* sólo por los índices, en un arreglo ordenado. El Listado 18 presenta un pseudo-código para este proceso. Aquí sí es necesario tener un arreglo temporal para mover los datos.

Una versión iterativa (presentada en el Listado 19) del algoritmo de Merge Sort, tomando un enfoque de abajo hacia arriba, *bottom-up*, trabaja de la siguiente manera: En el primer paso, mezclamos elementos consecutivos formando parejas ordenadas. En el segundo paso, mezclamos parejas ordenadas de elementos generando cuartetos ordenados y así sucesivamente, hasta tener todo el arreglo ordenado. La Figura 5.2 ilustra un ejemplo de esta versión.

Otro Análisis

Considerando el código recursivo del Listado 18, determinaremos de forma más directa el desempeño computacional del Merge Sort. Sin pérdida de generalidad, suponemos que n, el tamaño de la secuencia es una potencia de 2: $n = 2^k$, para k, entero positivo.

Listado 18 Proceso Mezcla

```
// PreC:  recibe dos secuencias ordenadas: S[Izq1,Izq2] y S[Der1,Der2];
//        Izq1 <= Izq2 = Der1-1;  Der1 <= Der2;
//PostC:  regresa el arreglo S[Izq1..Der2] ordenado

 Mezcla (int Izq1, Izq2, Der1, Der2){
  int i, j, k;                          // indices auxiliares
  array T[Izq1,Der2];                   // arreglo temporal

  i = Izq1;          // primer indice del sub-arreglo izquierdo
  j = Der1;          // primer indice del sub-arreglo derecho
  k = 1;             // indice auxiliar

  while (i <= Izq2) && (j <= Der2) do { // mientras no excedan los extremos
     if  ( S[i] <= S[j] ) {             //   mueve los k menores elementos
        T[k] = S[i];                    //   de S[Izq1..Der2] a T[1..k]
        i++ ;  k++ ;       }            // avanza los contadores
     else{
          T[k] = S[j];    j++ ;   k++ ;  }
  }// end while

  while  (i <= Izq2) do {               // mueve el resto de los elementos,
     T[k] = S[i];    i++ ;   k++ ; }    // si hay, de S[Izq1,Izq2] a T

  while  (i <= Der2) do {               // mueve el resto de los elementos,
     T[k] = S[j];    j++ ;   k++ ; }    // si quedan, de S[Der1,Der2] a T

  For (i= 1; i=k-1; i++);               // mueve los datos del temporal T
    S[i-1+ Izq1] = T[i];                // al original S

}// end
```

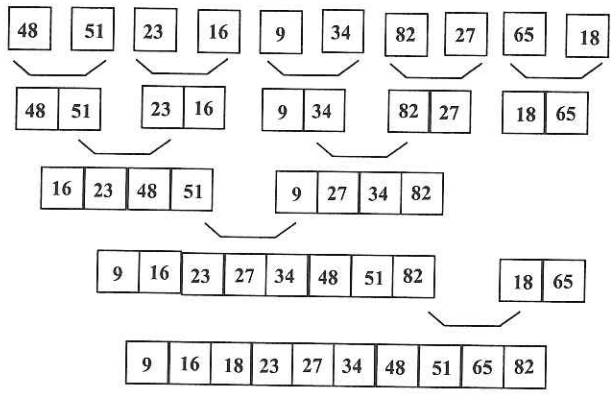


Figura 5.2: Ejemplo de Merge Sort Iterativo

Listado 19 pseudocódigo Merge Sort Iterativo

```
// PreC:  secuencia contenida en un arreglo S [1,N] y  S es no vacio.
//PostC:  el arreglo que contiene a la secuencia ordenada

MergeSort_It (array S; int n) {
  int s = 1;                              // tamanio del arreglo activo
  int Izq1, Izq2, Der1, Der2;             // indices auxiliares
   while ( s < N )  do {                  // calcula la cota de las sublistas
     Izq1 = 1;
     while ( Izq1 + s <= N ) do {
       Izq2 = Izq1 + s -1;   Der1 = Izq2 + 1;
       if ( ( Der1 + s - 1) > N ) then Der2 = N;
          else    Der2 = Der1 + s - 1;
       Mezcla ( Izq1, Izq2, Der1, Der2 );     // mezcla los subarreglos
       Izq1 = Der2 + 1;                  // actualiza el primer indice
       s = s * 2;                        // duplica el tamanio del arreglo
    } //end while Izq1 ...
   } //end while s ...
  } // end Divide...

}//end MergeSort
```

Sea $T_{MS}(n)$ el tiempo de ejecución del algoritmo Merge Sort aplicado a un ejemplar de tamaño n. Definimos $T_{MS}(1) = 1$. Ahora bien, $T_{MS}(n)$ es igual al tiempo requerido al aplicar el Merge Sort en dos sub-secuencias de tamaño $n/2$ más el tiempo para mezclarlas, el cual es lineal. Así, obtenemos la siguiente relación recurrente:

$T(n) = T_{MS}(1) = 1$ // tiempo sobre secuencias de tamaño 1;

$T(n) = T_{MS}(n) = 2 \cdot T_{MS}(n/2) + n.$ // tiempo sobre secuencias de tamaño n.

Para resolver esta relación de recurrencia, primero la dividiremos entre n:

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

Esta ecuación es válida para cualquier n que sea potencia de 2, entonces:

$$\frac{T(n/2)}{n/2} = \frac{T(n/4)}{n/4} + 1; \quad \frac{T(n/4)}{n/4} = \frac{T(n/8)}{n/8} + 1; \quad \cdots$$

$$\cdots \quad \frac{T(4)}{4} = \frac{T(2)}{2} + 1 \quad \frac{T(2)}{2} = \frac{T(1)}{1} + 1.$$

Si ahora sumamos todos los términos del lado izquierdo y los igualamos con la suma de los términos del lado derecho, tenemos que $T(n/2)/(n/2)$ aparece en ambos lados, por lo cual se cancela. De hecho, virtualmente todos los términos que aparecen en ambos lados serán cancelados, pero falta por sumar los 1's y éstos son $\log n$. Después de que todo es acumulado, tenemos que:

$$\frac{T(n)}{n} = \frac{T(1)}{1} + \log n \quad \Rightarrow \quad T(n) = n \log n + n \text{ es } O(n \log n).$$

5.2. Heap Sort

Esta técnica también es conocida como ordenamiento por montículos. La estrategia que emplea consiste en aprovechar el comportamiento de una estructura de datos denominda Heaps Binarios. El Apéndice D, presenta una breve descripción de los Heaps Binarios.

Estrategia

La estrategia de esta técnica requiere de las operaciones y acciones de los heaps, a continuación describiremos dos de ellas:

Reorganización del Heap: Dado un árbol binario completo, se reorganizan los elementos del árbol para que cumplan con la relación de orden existente en el heap, a tal proceso también se le conoce como heapify o reHeap.

Eliminación del Mayor: En un heap existe una relación de orden entre sus nodos, el nodo con la mayor prioridad se encuentra en la raíz y ésto sucede para cada subárbol. Usaremos las operaciones de una cola de prioridades para eliminar el elemento de mayor prioridad, a esta función la denominaremos BorraMayor y requiere de:

1. Extraer el elemento que está en la raíz del heap.
2. Almacenar tal elemento en una lista.
3. Eliminar la raíz del heap.
4. Reorganizar el heap usando el proceso reHeap.

El bloque de pasos anteriores se lleva acabo de manera recursiva hasta que el heap queda vacío; ahora se tiene ordenada la secuencia en la lista.

Para ilustrar la estrategia se realizará un ejemplo en donde se desea ordenar la secuencia $S = \{16, 4, 9, 14, 1, 3, 10, 2, 8, 7\}$ de manera ascendente. Primero construimos un heap con los elementos de S.

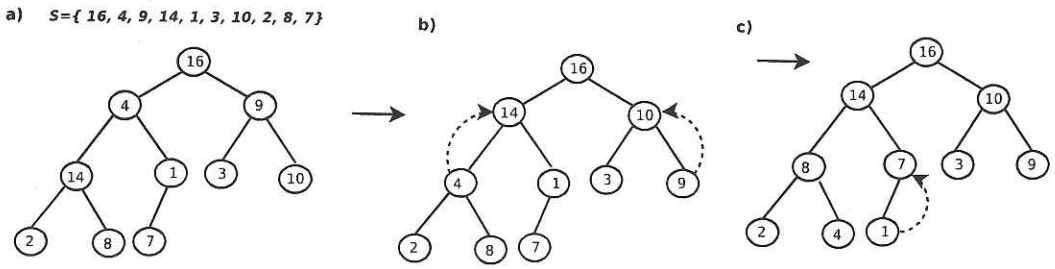


Figura 5.3: Ejemplo de construcción de un heap a partir de una secuencia

Como se observa en la Figura 5.3(a) el primer paso consiste en construir un árbol completo a la izquierda, se va insertando, cada dato, en la última posición del árbol, conforme se obtiene de la secuencia. En el inciso **(b)**, se observa el resultado de aplicar el procedimiento reHeap a los nodos intermedios del árbol, intercambiaron de posición el 14 y el 4, así como 9 y 10. En **(c)**, se tiene el resultado de aplicar reHeap a las hojas y sus padres, aquí cambio el 7 con el 1. Aquí se da por concluida la construcción de este heap maximal.

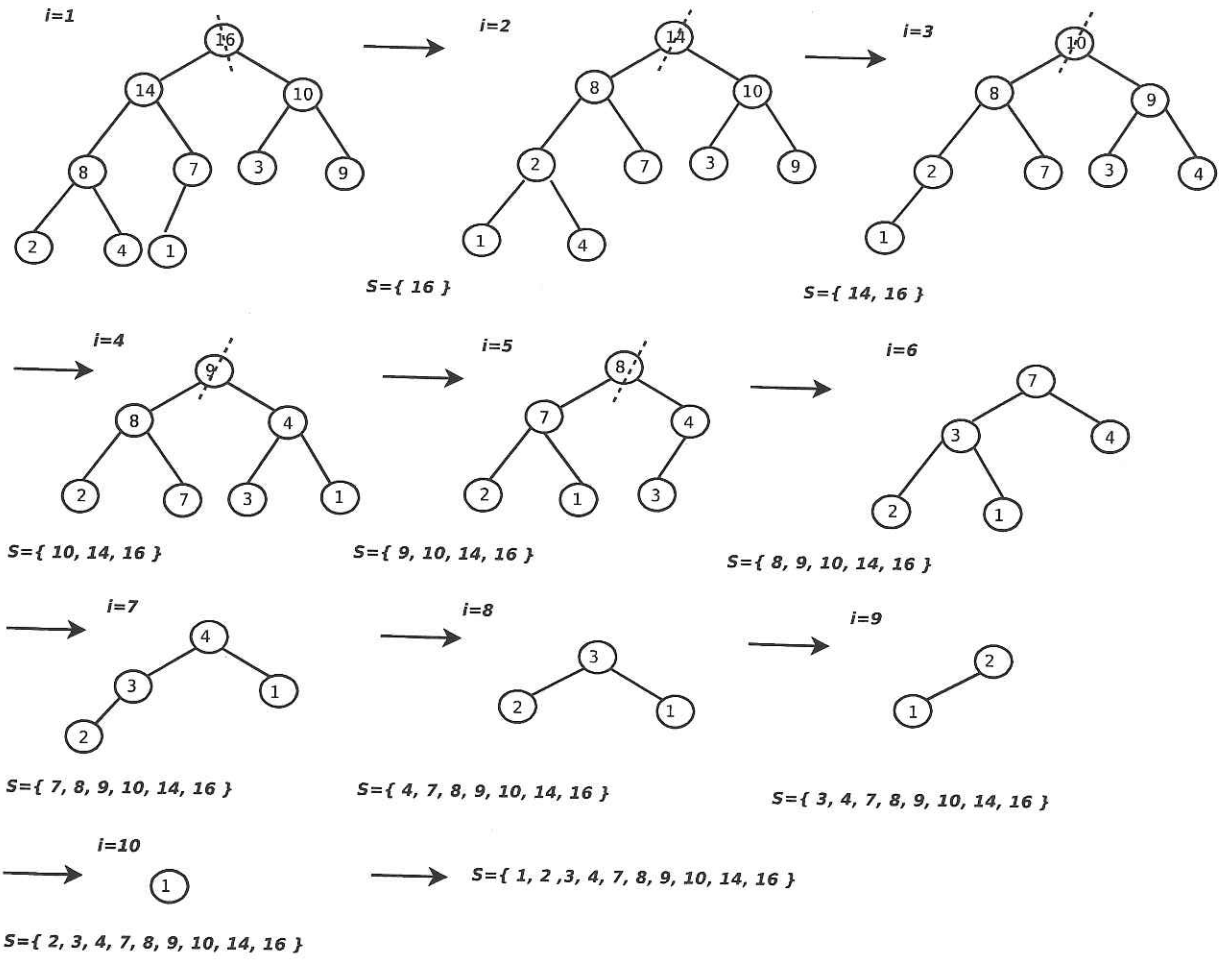


Figura 5.4: Ejemplo de implementación de una cola de prioridades

Una vez que se tiene el heap se procede a aplicar la función BorraMayor. En la Figura 5.4, se presenta en cada iteración cómo se toma la raíz del heap, la cual es almacenada en una lista, después se elimina la raíz del heap y se le aplica reHeap al árbol resultante para reorganizarlo. Se sigue con el proceso en las demás iteraciones.

Cabe mencionar, que no es necesario tener la lista, se puso aquí por motivos didácticos. En la práctica, se recomienda, ir dejando los elementos en el arreglo original.

Análisis de Complejidad

Para obtener el desempeño de este método hay que considerar la complejidad de los puntos clave de la estrategia, es decir construir el heap, reorganizarlo y eliminar al elemento de mayor prioridad.

El proceso reHeap, tiene un papel fundamental durante el desarrollo completo del método, desde convertir al árbol en un heap y mantener las propiedades de éste, así como borrar al elemento de mayor prioridad. Por lo tanto, el análisis global depende, en gran parte, del desempeño computacional del proceso reHeap.

Proceso reHeap

Sea T un árbol binario con n elementos, podemos descomponerlo T en: un nodo raíz r y sus dos subárboles: T_L y T_R. Sabemos que a un nodo se le puede considerar como un árbol con profundidad cero y también podemos considerarlo como un heap. Supongamos que los subárboles contenidos entre los niveles 1 y k son heaps. Es claro que al emplear la operación reHeap para bajar un elemento un nivel, se realizarón dos comparaciones, pues los subárboles inmediatos inferiores son heaps, entonces el número promedio de comparaciones que se espera realice reHeap en un árbol con profundidad k se puede escribir como:

$$\frac{1}{k} \cdot \sum_{i=1}^{k} 2i = \left(\frac{1}{k}\right)\left(\frac{2k\,(k+1)}{2}\right) = k+1.$$

Dado que $k = \log n$, entonces la operación reHeap tiene un desempeño de $O(\log n)$.

Construcción del Heap

Para calcular el desempeño de construir el heap emplearemos el proceso reHeap en forma ascendente, es decir empezaremos en el nivel más profundo e iremos subiendo en dirección a la raíz, de está forma podemos convertir al árbol en un heap. Si consideramos un árbol con n elementos entonces se ejecutara la misma cantidad de veces el proceso reHeap, entonces el convertir un árbol completo a la izquierda tiene una complejidad de $O(n \log n)$. Por otro lado, para calcular la complejidad de construcción del árbol completo a izquierda consideraremos que a éste lo representamos con un arreglo. Entonces el desempeño de insertar un elemento es constante y es 1, como se tienen n elementos entonces el desempeño al momento de crear el árbol a la izquierda es de $O(n)$.

Por lo tanto, la complejidad de crear un heap es de $O(n \log n + n)$, por lo que podemos concluir que construir el heap requiere tiempo $O(n \log n)$.

Análisis del BorraMayor

Acceder al elemento raíz, agregarlo a la lista y eliminarlo el árbol, son tareas que se realizan en tiempo constante. Después de estas tareas simples, es necesario reorganizar

el heap y para ello ejecutamos el proceso reHeap, cuyo desempeño es $O(\log n)$. Requerimos hacer esto para cada elemento, por lo tanto, el tiempo de ejecución de la función BorraMayor es:

$$\sum_{i=1}^{n} \log n = n \log n \quad \in \quad O(n \log n)$$

Por lo tanto, la función BorraMayor tiene desempeño computacional de $O(n \log n)$.

Desempeño Computacional de Heap Sort

En los apartados anteriores se calculó el desempeño para los dos procesos principales que componen método de ordenamiento de heap sort. Si denotamos al desempeño como T_{hs} lo podemos escribir como: $T_{hs} = O(n \log n) + O(n \log n) \in O(n \log n)$

Cabe mencionar que no importa si la secuencia esta ya ordenada o no, Heap Sort siempre realiza el mismo número de comparaciones. De hecho, si la secuencia de entrada S ya está ordenana, Heap Sort la desorganiza totalmente, para ordenarla!

Por lo tanto, podemos concluir que la complejidad de heap sort es de $\Theta(n \log n)$.

Algoritmo

Dada una Lista L de n elementos, algoritmo Heap Sort se puede resumir en los siguientes cuatro pasos:

1.- Meter en un árbol binario, los elementos de la lista L.
2.- Empezando con el *último* subárbol, re-establecer heaps binarios hasta llegar a la raíz.
3.- Reorganizar el Heap para que los elementos queden ordenados.
4.- Vaciar el Heap a la lista ordenada.

El Listado 20 presenta un pseudocódigo para heapsort, supone que S es una secuencia de números enteros, no vacía y de tamaño n, contenida en un arreglo A.

El Listado 21 presenta un pseudocódigo para el proceso reHeap. Se asume que el sub-árbol enraizado en la posición i es un heap, excepto (tal vez) en la posición i. Para hacer del sub-árbol un heap preguntamos si $A[i]$ es menor o igual a su hijo más grande, de ser así se realiza un intercambio.

Listado 20 pseudocódigo Heap Sort

```
// PreC: S una secuencia con n elementos, contenida en un arreglo A[1,n].
// PostC: Arreglo que contiene en orden ascendente a los elementos de S.

HeapSort(array A; int n){
  heap H;                          // arbol binario completo
  int  i;                          // contador

  H.create;                        // crea el arbol binario completo
  for  (i=1; i=n; i++)             // inserta los elementos en el arbol
     H.Insert_Last ( A[i] );

  for  (i=n/2; i=1; i--)           // convierte el arbol en un heap
     H.ReHeap (i, n);

  for  (i=n; i=2; i--) {           // pone los elementos, del mayor al menor
     H.swap( 1, i );               // en las posiciones n, n-1, ..., 2, 1.
     H.reHeap( 1, i-1 ); }

  for  (i=1; i=n; i++)             // copia los elementos del heap de
     H.retrieve ( A[i], i );       // regreso al arreglo

}// end HeapSort
```

Listado 21 pseudocódigo reHeap

```
// PreC: el subarbol enraizado en i es un heap, excepto (quiza) por i
// PostC:  el subarbol enraizado en i es un heap.

reHeap(int i,j){
  int pmg;                                    // Posicion del Mas Grande

  if  ( 2*i <= j )                            // A[i] tiene al menos un hijo
    if  (2*i = j)  pmg = j;                   // ... tiene solo un hijo
    else                                      // ... tiene dos hijo
      if  (A[2*i] >= A[2*i+1])  pmg = 2*i;
      else  pmg = 2*i+1;

  if (A[i] < A[pmg])  {
    swap (i, pmg);                            // intercambia los elementos
    if  (2*pmg <= j )
      reHeap(pmg, j) }                        // aplica el reHeap

}// end reHeap
```

5.3. Quick Sort

El algoritmo QuickSort es uno de los más eficientes métodos de ordenamiento; es uno de los más utilizados dada su fácil implementación, poco consumo de recursos (memoria) y por tener muy buen desempeño en la mayoría de los casos. De hecho, cuando se implementa cuidadosamente, su desempeño computacional en el caso promedio es menor que el de otros algoritmos. Desafortunadamente, en el peor de los casos su desempeño es pobre y en la práctica es necesario tomar precauciones para reducir la probabilidad de que el peor caso suceda. Realizaremos el análisis del algoritmo QuickSort, determinando su desempeño computacional en el mejor caso, peor caso y el caso promedio, primero de manera intuitiva; después, de manera rigurosamente formal, realizaremos el análisis del caso promedio.

La parte más interesante y, por lo tanto, más laboriosa es determinar el desempeño computacional del Algoritmo QuickSort en el caso promedio. Para hacerlo de manera formal, necesitamos utilizar propiedades de árboles binarios, en especial de árboles binarios de búsqueda, *binary search trees*. Por lo cual, presentamos en el Apéndice C las definiciones y resultados necesarios para árboles.

Estrategia

La estrategia del QuickSort está basada en la técnica *divide y vencerás*; de hecho, dada una secuencia de elementos, se puede describir como:

divide: La secuencia es dividida (particionada) en dos subsecuencias. Todos los elementos de la primera subsecuencia son menores o iguales a los elementos de la segunda; además, ambas subsecuencias no pueden ser vacías al mismo tiempo.

vence: Las dos subsecuencias son ordenadas mediante llamadas recursivas QuickSort.

Podemos describir el proceso de la siguiente manera: al aplicar el QuickSort a un arreglo S, primero particionamos el arreglo en dos: el subarreglo *izquierdo* y el *derecho* como se indicó anteriormente y así continuamos hasta que el arreglo quede ordenado. Ya que esta última operación es fácil de atender con dos llamadas recursivas al QuickSort, nos concentraremos en la fase de partición.

La partición inicia seleccionando un elemento del arreglo $S[1..n]$, el cual se denomina *pivote*. Las maneras más comunes de elegir el pivote son:

a) El primer elemento: $S[1]$;
b) El último elemento: $S[n]$;
c) El elemento medio: $S[n \text{ div } 2]$;
d) La media de: $\{S[1],\ S[n \text{ div } 2],\ S[n]\}$;

Para todos estos casos, especialmente para los tres primeros, es posible construir un arreglo S' que provoque el peor de los casos para QuickSort.

Para describir el algoritmo supondremos que el pivote es el primer elemento del arreglo dado $S[a..b]$, donde a y b son el menor y mayor índice del arreglo, respectivamente.

La idea de QuickSort es tomar el pivote, en este caso $S[a]$, y moverlo a la posición que ocupará cuando el arreglo esté ordenado. Al mismo tiempo, otras entradas del arreglo también son movidas, garantizando que todas las entradas a la izquierda del pivote no sean mayores a él pivote y que todas las entradas a la derecha no sean menores al pivote.

Por el momento, suponemos que existe la función Partition(a, b) que reorganiza las entradas de la manera descrita anteriormente y regresa el valor j, el índice final del pivote, es decir, la posición correcta de pivote. Después, los subarreglos $S[a..(j-1)]$ y $S[(j+1)..b]$ serán ordenados recursivamente y así el arreglo entero quedará ordenado.

El Listado 22 muestra el pseudo-código del método QuickSort descrito. El Listado 23 presenta una versión de función Partition, la cual considera al primer elemento como el pivote.

Listado 22 Quick Sort

```
QuickSort(array S; int a, b){
  int j;
  if (a <= b) {
        j = Partition(a,b);       // particiona el arreglo
        QuickSort(S,a,j-1);       // reorganiza los subarreglos
        QuickSort(S,j+1,b);
  }// end if
}// end QuickSort
```

Análisis de Complejidad

Primero determinaremos el desempeño computacional de la función Partition sobre una secuencia de n elementos. Después revisaremos, de manera intuitiva, el comportamiento del QuickSort en el mejor caso y el peor caso.

Complejidad de la función Partition

Para una secuencia S de n elementos, cada elemento en S es comparado con el pivote, para decidir si va a la izquierda o a la derecha de él. Esto significa que es necesario revisar todos los elementos del arreglo para determinar la partición. Por lo tanto, la función Partition requiere tiempo $O(n)$.

Revisando el código del Listado 23, podemos ver que los ciclos while $(S[i] < S[a])$ mueven, potencialmente, al índice i de la posición $(a+1)$ a la b. Esto significa que en el peor caso este índice se mueve $n-1$ posiciones. Tenemos algo análogo para los ciclos While $(S[j] > S[a])$. Por otro lado, el proceso termina cuando $(i < j)$, es decir cuando los índices se cruzan, lo que significa que ninguno de los índices llegó al extremo opuesto, pero juntos recorrieron los N elementos. Es decir, i se movió de $(a+1)$ a una posición mayor a j y el índice j se movió desde b a una posición menor a j. Por lo tanto, en este caso se tiene que la función Partition requiere tiempo $O(n)$.

Listado 23 Partition

```
int Partition(array S; int a, b) {
  int  i,j;                          // indices auxiliares

   i= a+1;                           // se mueve hacia el indice final b
   j= b;                             // se mueve hacia el indice inicial a

   while (S[i] < S[a]) do i=i+1; // avanza i mientras el elemento en i
                                     // sea menor que el pivote
   while (S[j] > S[a]) do j=j-1; // avanza j mientras el elemento en j
                                     // sea mayor al pivote

   while (i < j) do {                // mientras no se crucen los indices
      swap(S[i], S[j]);              //   intercambia los elementos
      i=i+1;  j=j-1;                 //   avanza los contadores
      While (S[i] < S[a])   i=i+1;
      While (S[j] > S[a])   j=j-1;
   }

   If (a<j)  swap(S[a], S[j]);       // ubica al pivote en su posicion
                                     // correcta

  Return  j                          // regresa la posicion de pivote,
                                     // el punto donde se hara el corte
}// end Partition
```

Peor caso del QuickSort

El peor caso del QuickSort sucede cuando la partición, sobre un arreglo de n elementos, nos regresa un subarreglo de tamaño $(n-1)$ cada vez. Para la implementación presentada en el Listado 23, el pivote es el primer elemento del arreglo. Entonces, una secuencia ya ordenada (ascendente o descendentemente) provoca el peor caso para esta versión del algoritmo QuickSort.

Supongamos que la secuencia S, de n elementos, ya está ordenada en forma ascendente. En cada iteración la función Partition(a, b) nos regresará $j = a$. Esto significa que el subarreglo izquierdo será vacío y el derecho tendrá cada vez un elemento menos. Es decir, en la primera iteración iniciamos con n elementos, para la segunda con $(n-1)$, en la tercera con $(n-2)$ y así sucesivamente.

La Figura 5.5 ilustra los subarreglos obtenidos en el peor caso, donde S es la secuencia ordenada $S = \{1, 2, ..., n\}$, a la derecha se pone el tamaño del subarreglo. Podemos observar que al final se tienen $(n-1)$ particiones. De esta manera, tenemos que para un arreglo de tamaño n, el número de particiones en el peor caso es $(n-1)$ y cada una de ellas cuesta $O(n)$, por lo que, en total se requieren del $O(n^2)$ operaciones para realizar el QuickSort. Podemos concluir, entonces, que el desempeño computacional del QuickSort, en el peor de los casos, es $O(n^2)$.

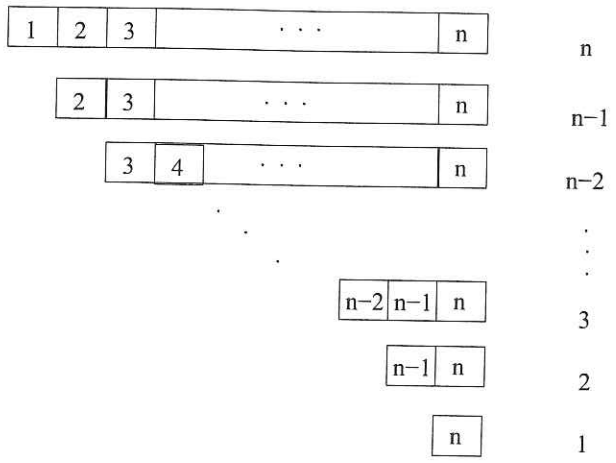


Figura 5.5: Ejemplo del peor caso para QuickSort

Mejor caso del QuickSort

Para la implementación presentada en el Listado 23, el mejor caso es generado al tener, en cada iteración, al elemento que representa la mediana de los elementos de la secuencia en la primera posición.

De esta manera, la función Partition(a, b) regresará $j = \lceil (a+b)/2 \rceil$. Esto significa que cada vez el arreglo es dividido en subarreglos de tamaño similar.

La Figura 5.6 ilustra los subarreglos obtenidos en el mejor caso; a la derecha se pone el tamaño de cada subarreglo.

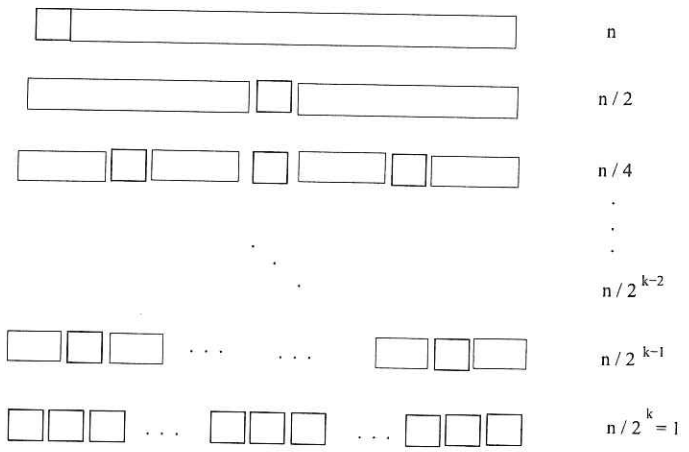


Figura 5.6: Ejemplo del mejor caso para QuickSort

Si para un arreglo de longitud n, obtenemos, para cada partición, subarreglos de tamaño similar, entonces el número de llamadas a la función Partition es de $O(\log n)$. Así que, el número total de operaciones a realizar es $O(n \log n)$.

Por lo tanto, en el mejor caso, el desempeño computacional de QuickSort requiere tiempo $O(n \log n)$.

Análisis del caso promedio

El análisis del QuickSort se basa en una correspondencia, quizá inesperada, entre él y los árboles binarios de búsquedas[1]. Es decir, podemos ver el efecto del QuickSort sobre una secuencia S de n datos similar a la de construir un árbol binario de búsqueda T cuyo elemento raíz es el pivote y donde los subárboles derecho e izquierdo corresponden a los subarreglos respectivos generados de la función Partition.

Por ejemplo, sea S la secuencia $S = \{22, 41, 11, 34, 5, 27\}$, el pivote es el número 22, el cual es comparado con cada uno de otros cinco elementos de S, para determinar los elementos que van a la izquierda y los de la derecha. La Figura 5.7 muestra a la izquierda el arreglo inicial y a la derecha el resultado de aplicar la función Partition.

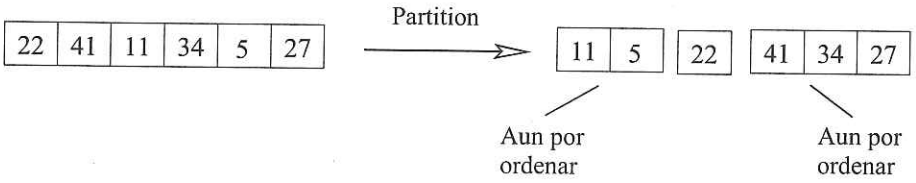
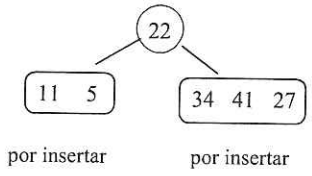


Figura 5.7: Ejemplo de la ejecución de la función Partition

Ahora, consideramos insertar los elemento de la secuencia S en un árbol binario de búsqueda T, inicialmente vacío. El primer elemento será la raíz, se realizan $(n-1)$ comparaciones entre la raíz y los otros elementos, los cuales pasan por la raíz en su camino al subárbol izquierdo o derecho. La siguiente figura muestra el resultado parcial de insertar los elementos en T. En este punto, el costo de ambos es $(n-1)$ y el mismo argumento se aplica recursivamente.



Si S_n es el conjunto de todas las permutaciones de n elementos, podemos concluir que para cada permutación α en S_n el costo de QuickSort sobre la secuencia α es el mismo costo que se tiene al insertar los elementos de α uno por uno en un árbol binario de búsqueda, inicialmente vacío.

Antes de iniciar el análisis formal del caso promedio, observemos el peor caso y el mejor caso del QuickSort desde este enfoque de árboles binarios. Tenemos que el peor caso sucede cuando la lista está ordenada. En este caso, la altura del árbol es n. La Figura 5.8 muestra, a la izquierda, el resultado de insertar los elementos de la secuencia S ordenada ascendentemente, $S_A = \{5, 11, 22, 27, 34, 41\}$, y, a la derecha, se presenta el árbol obtenido cuando S está ordenada descendentemente.

[1]Traducción literal de *Binary Search Tree*, presentados en el Apéndice C.

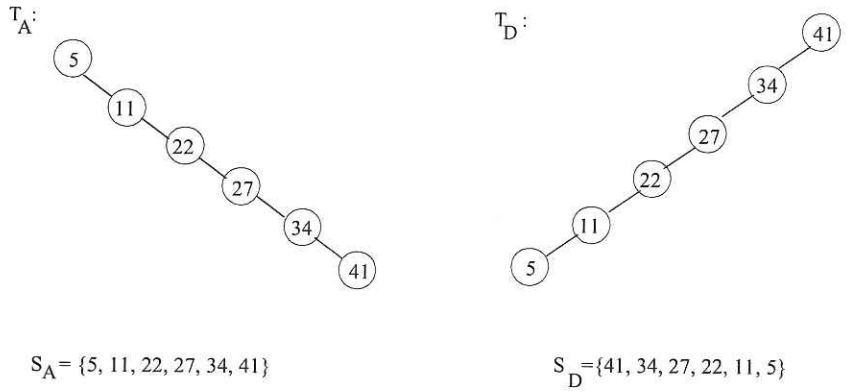
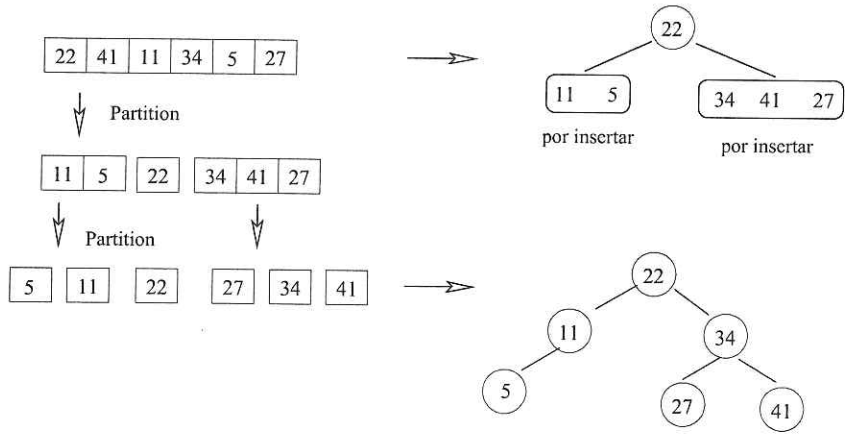


Figura 5.8: Peor caso: secuencia ordenada

En el mejor de los casos tenemos que cada vez los subarreglos son casi del mismo tamaño y la media de cada elemento está en la primera posición. La siguiente figura muestra, a la izquierda, el resultado de la función Partition en cada iteración y a la derecha el árbol parcial resultante.



Para determinar el orden del caso promedio (sobre $n!$ permutaciones de un arreglo de n elementos) el número de particiones es el promedio de la altura del árbol binario de búsqueda creado por la partición; se asegura que tal valor es $O(\log n)$, por lo tanto, en el caso promedio el tiempo del QuickSort es $O(n \log n)$.

Análisis detallado

Hemos visto que la altura del árbol binario de búsqueda depende de la forma cómo llegan los elementos al momento de insertarlos, en un árbol inicialmente vacío, y al final de todas las n inserciones, tal altura puede ser n, en el peor caso o $\log n$, en el mejor caso.

Intuimos que estos árboles deberían tener un mejor desempeño a $O(n)$ en la mayoría de los casos. Por lo cual realizaremos un análisis del caso promedio, suponiendo que las secuencias a insertar en el árbol binario de búsqueda se obtienen bajo una distribución uniforme.

Listado 24 Proceso β

```
beta(array S){
   BinarySearchTree T;          // arbol de busqueda binario T

    T.create;                   // construye un arbol vacio
    while  (i <= n ) do
      T.Inserta(S[i]);          // inserta en T al i-esimo dato

    return T;                   // regresa el arbol construido T
}
```

Hemos dicho que podemos ver el comportamiento del QuickSort sobre una secuencia S de n elementos similar al de construir un árbol binario de búsqueda T, inicialmente vacío, cuyo elemento raíz es el pivote y donde los sub-árboles derecho e izquierdo corresponden a los subarreglos respectivos generados por la función Partition. Así que realizar este análisis del caso promedio para los árboles binarios de búsquedas, es similar a revisar el comportamiento del algoritmo QuickSort en el caso promedio sobre secuencia de tamaño n distribuidas uniformemente.

Por lo cual, analizaremos el tiempo esperado de realizar una serie de n inserciones en un árbol binario de búsqueda inicialmente vacío. Utilizaremos las definiciones y resultados presentados en el Apéndice C.

Considere el Proceso *Building*, que denominamos β, presentado en el Listado 24, el cual recibe una secuencia S de n datos y construye un árbol binario de búsqueda, T, inicialmente vacío, insertando uno a uno los elementos de S en T.

El desempeño computacional de β en el caso promedio nos indicará el comportamiento en el caso promedio de los árboles binarios de búsquedas, que a su vez nos dirá el tiempo de ejecución, en el caso esperado, para el algoritmo QuickSort.

Supondremos que las secuencias que recibe β son listas de enteros del 1 a n ya que una secuencia como $S = \{carlos, daniel, beto, ana\}$ es similar a $S = \{3, 4, 2, 1\}$. Así que los ejemplares de β son las $n!$ permutaciones de los números $1, 2, 3, ..., n$.

Denotaremos por $T(a)$ al árbol construido por $\beta(a)$. Por ejemplo para $n = 3$, tenemos seis permutaciones. La Figura 5.9 presenta los árboles generado por cada una de las seis permutaciones. Al exterior de cada nodo x se indica el número de comparaciones usadas para insertar x en el árbol.

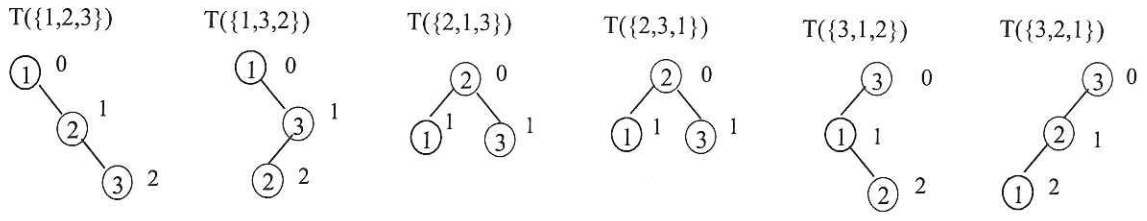


Figura 5.9: Construcción de los árboles $T(a)$ con la función $\beta(a)$

El costo de insertar al nodo x es igual a su profundidad en $T(a)$, ya que x se compara con cada uno de sus antecesores durante su inserción. Sumando sobre todos los nodos, el costo total de $\beta(a)$ resulta ser la longitud de la ruta interna $\iota(T(a))$. Por ejemplo, el costo de construir $T(\{2,1,3\})$ es 0+1+1=2 y el costo de construir $T(\{1,2,3\})$ es 0+1+2=3.

Así que, la máxima longitud interna se da sobre los árboles asimétricos y es: $n(n-1)/2$. Por lo tanto, la complejidad, en el peor de los casos para β es $O(n^2)$.

Para un análisis del caso esperado (promedio) se supone que la secuencia a se toma bajo una distribución uniforme sobre un conjunto de S_n permutaciones de n elementos. Entonces tomamos el promedio sobre esos $n!$ ejemplares para a:

$$\mathcal{A}(n) = \sum_{a \in S_n} \frac{1}{n!} \cdot \iota(T(a))$$

Por ejemplo, $\mathcal{A}(n) = \mathcal{A}(3) = (3+3+2+2+3+3)/6 = 8/3 = 2.6 < 3$. Para $n = 4$, hay nueve árboles con $\iota(T(a)) = 6$, tres con $\iota(T(a)) = 5$ y doce con $\iota(T(a)) = 4$ entonces $\mathcal{A}(4) = [9(6) + 3(5) + 12(4)]/24 = 117/24 = 4.8 < 5$. Esto nos indica que se espera que $\iota(T(a)) < 5$; es decir, se espera que se se realicen menos de cinco comparaciones para construir un árbol de cuatro elementos.

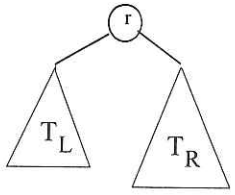


Figura 5.10: El árbol $T(a)$ y sus subárboles

Podemos convertir a $\mathcal{A}(n)$ en una ecuación recurrente, donde $\mathcal{A}(0) = 0$ y para $n > 0$ sea $T(a)$ un árbol binario de búsqueda con raíz r, subárbol izquierdo $L(a)$ y el derecho $R(a)$. La Figura 5.10 ejemplifica este árbol.

De la demostración del Teorema C.1 del Apéndice, tenemos que

$$\iota(T(a)) = n - 1 + \iota(L(a)) + \iota(R(a)).$$

Por lo tanto,

$$\mathcal{A}(n) = \sum_{a \in S_n} \frac{1}{n!} \cdot \iota(T(a)) = \frac{1}{n!} \sum_{a \in S_n} [n - 1 + \iota(L(a)) + \iota(R(a))]$$

$$= (n-1) + \frac{1}{n!} \cdot \sum_{a \in S_n} [\iota(L(a))] + \frac{1}{n!} \cdot \sum_{a \in S_n} [\iota(R(a))]$$

$$= (n-1) + \frac{2}{n!} \cdot \sum_{a \in S_n} [\iota(L(a))].$$

El último paso se cumple por simetría sobre el promedio: cuesta lo mismo construir subárboles izquierdos que derechos.

Sea S_n el conjunto de todas las permutaciones de 1,2, ..., n. Sea S_n^j el conjunto de permutaciones $a \in S_n$ tal que el primer elemento de a es el número j. Entonces, la raíz de $T(a)$ contiene a j.

La Figura 5.11 ejemplifica los árbol obtenidos por la función β aplicada a las permutaciones $a_1 = \{6, 2, 1, 5, 7, 3, 4, 8\}$ y $a_2 = \{6, 1, 8, 7, 3, 2, 5, 4\}$ de S_8^6.

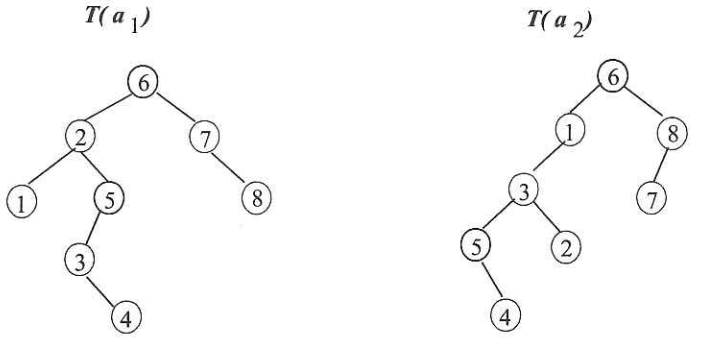


Figura 5.11: Construcción de dos árboles $T(a)$ con $a \in S_8^6$

Los ejemplos muestran que si $a \in S_n^j$, el subárbol $L(a)$ es determinado por el orden en que llegan los números $1, 2, ..., (j-1)$ a la permutación a. Hay $(j-1)!$ posibles ordenes de esos números y todos ellos ocurren en S_n^j. Pero S_n^j tiene $(n-1)!$ elementos, por lo que cada permutación de $1, 2, ..., (j-1)$ ocurre exactamente $(n-1)!/(j-1)!$ veces en S_n^j.

De esta manera, hemos determinado el efecto sobre los subárboles izquierdos.

Por lo tanto,

$$\sum_{a \in S_n^j} \iota(L(a)) = \frac{(n-1)!}{(j-1)!} \sum_{a \in S_{j-1}} \iota(T(a)),$$

luego,

$$\sum_{a \in S_n^j} \iota(L(a)) = (n-1)! \left[\frac{1}{(j-1)!} \sum_{a \in S_{j-1}} \iota(T(a)) \right] = (n-1)! \mathcal{A}(j-1).$$

La última igualdad se da por la definición de $\mathcal{A}(n)$. Entonces, al sustituir en $\mathcal{A}(n)$,

$$\mathcal{A}(n) = (n-1) + \frac{2}{n!} \sum_{a \in S_n} \iota(L(a)) = (n-1) + \frac{2}{n!} \sum_{j=1}^{n} \sum_{a \in S_n^j} \iota(L(a))$$

$$\mathcal{A}(n) = (n-1) + \frac{2}{n!} \sum_{j=1}^{n} (n-1)! \mathcal{A}(j-1) = (n-1) + \frac{2(n-1)!}{n!} \sum_{j=1}^{n} \mathcal{A}(j-1).$$

Por lo tanto,

$$\mathcal{A}(n) = (n-1) + \frac{2}{n} \cdot \sum_{j=1}^{n} \mathcal{A}(j-1).$$

Hemos obtenido una ecuación recurrente para $\mathcal{A}(n)$, con $\mathcal{A}(0) = 0$.

Así, tenemos que el costo esperado del proceso β es

$$\mathcal{A}(n) = (n-1) + \frac{2}{n} \cdot \sum_{j=1}^{n} \mathcal{A}(j-1).$$

Ahora, trataremos de resolver esta ecuación recurrente. El primer paso consiste en eliminar la suma sobre j: multiplicamos por n,

$$n\mathcal{A}(n) = n(n-1) + 2\sum_{j=1}^{n} \mathcal{A}(j-1);$$

sustituimos $(n-1)$ por n para obtener

$$(n-1)\mathcal{A}(n-1) = (n-1)(n-2) + 2\sum_{j=1}^{n-1} \mathcal{A}(j-1);$$

ahora restamos la segunda ecuación de la primera y la suma desaparece:

$$n\mathcal{A}(n) - (n-1)\mathcal{A}(n-1) = n(n-1) - (n-1)(n-2) + 2\mathcal{A}(n-1);$$

y así,

$$n\mathcal{A}(n) = 2(n-1) + (n+1)\mathcal{A}(n-1).$$

Dividimos por $2n(n+1)$ y obtenemos:

$$\frac{\mathcal{A}(n)}{2(n+1)} = \frac{(n-1)}{n(n+1)} + \frac{\mathcal{A}(n-1)}{2n}. \tag{5.1}$$

Pero,

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

por lo que,

$$\frac{(n-1)}{n(n+1)} = \frac{2}{(n+1)} - \frac{1}{n}. \tag{5.2}$$

Sustituyendo la ecuación 5.2 en la 5.1:

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

Aplicando recursivamente la fórmula:

$$\frac{\mathcal{A}(n)}{2(n+1)} = \frac{2}{(n+1)} - \frac{1}{n} + \left[\frac{2}{n} - \frac{1}{n-1} + \frac{\mathcal{A}(n-2)}{2(n-1)}\right].$$

$$\frac{\mathcal{A}(n)}{2(n+1)} = \frac{2}{(n+1)} - \frac{1}{n} + \left[\frac{2}{n} - \frac{1}{n-1} + \left[\frac{2}{n-1} - \frac{1}{n-2} + \frac{\mathcal{A}(n-3)}{2(n-2)}\right]\right];$$

reorganizando,

$$\frac{\mathcal{A}(n)}{2(n+1)} = \frac{2}{(n+1)} + \frac{2}{n} + \frac{2}{n-1} - \frac{1}{n} - \frac{1}{n-1} - \frac{1}{n-2} + \frac{\mathcal{A}(n-3)}{2(n-2)}.$$

Continuando así, en la $i-$ésima aplicación tenemos:

$$\frac{\mathcal{A}(n)}{2(n+1)} = \frac{2}{(n+1)} + \frac{2}{n} + \cdots + \frac{2}{n-(i-2)} - \frac{1}{n} - \frac{1}{n-1} - \cdots - \frac{1}{n-(i-1)} + \frac{\mathcal{A}(n-i)}{2(n-(i-1))}.$$

Si hacemos $i = n$, nos queda:

$$\begin{aligned}\frac{\mathcal{A}(n)}{2(n+1)} &= \frac{2}{(n+1)} + \frac{2}{n} + \cdots + \frac{2}{2} - \frac{1}{n} - \frac{1}{n-1} - \cdots - \frac{1}{1} + 0 \\ &= \frac{2}{(n+1)} + 2\left[\frac{1}{n} + \frac{1}{n-1} + \cdots + \frac{1}{2}\right] - \left[\frac{1}{n} + \frac{1}{n-1} + \cdots + \frac{1}{2}\right] - \frac{1}{1} \\ &= \frac{2}{(n+1)} + \left[\frac{1}{n} + \frac{1}{n-1} + \cdots + \frac{1}{2}\right] + \frac{1}{1} - 2 = \sum_{i=1}^{n}\left[\frac{1}{i}\right] - \frac{2n}{n+1}.\end{aligned}$$

Por lo tanto,

$$\frac{\mathcal{A}(n)}{2(n+1)} = \sum_{i=1}^{n}\left[\frac{1}{i}\right] - \frac{2n}{n+1}.$$

Ahora multiplicamos todo por $2(n+1)$, tenemos

$$\mathcal{A}(n) = 2(n+1)\sum_{i=1}^{n}\left[\frac{1}{i}\right] - 4n.$$

Por lo tanto, hemos demostrado el siguiente resultado:

Teorema 5.1 Sea T un árbol binario de búsqueda inicialmente vacío. Sea S_n el conjunto de todas las permutaciones de n elementos. Sea $\alpha \in S_n$ obtenida bajo una distribución uniforme. Entonces, el desempeño computacional, en el caso promedio, de insertar los n elementos de α en el árbol T es:

$$\mathcal{A}(n) = 2 \cdot (n+1) \cdot H_n - 4n$$

donde H_n es el $n-$ésimo número harmónico $H_n = \sum_{i=1}^{n} 1/i$.

Se tiene la aproximación: $H_n \simeq \ln n + \gamma$, donde $\gamma \simeq 0.5572$ es la constante de Euler.

Por lo tanto,

$$\mathcal{A} = 2(n+1)[\ \ln n + \gamma\ ] \quad \text{y} \quad \mathcal{A} \quad \text{es} \quad O(n \ln n).$$

Es decir, se requiere tiempo de $O(n \ln n)$ para realizar las n inserciones, por lo que cada inserción requiere, en promedio, tiempo $O(\ln n)$

Ante esto, podemos concluir que el desempeño computacional del algoritmo QuickSort, en el caso promedio es $O(n \ln n)$ al aplicarlo en una secuencia de tamaño n, la cual fue obtenida aleatoriamente.

5.4. Tree Sort

Este algoritmos usa a los árboles binarios de búsqueda, *binary search trees, bst*[2], como una estructura auxiliar para ordenar los datos de una secuencia dada.

Estrategia

Lo único que hace este proceso es guardar la secuencia dada S en un árbol de búsqueda binaria, con las reglas de éste, y después hace un recorrido en-orden, *in-order*, sobre el árbol, almacenando en una lista los datos recuperados del árbol.

Consideremos la secuencia $S = \{16, 4, 9, 14, 1, 3, 10, 2, 8, 7\}$. Para ilustrar la estrategia aplicaremos el proceso Tree Sort sobre S. La Figura 5.12 presenta el árbol de búsqueda binaria T, despúes de insertar la secuencia S. Finalmente, se muestra la secuencia obtenida al aplicar el recorrido en-orden sobre T.

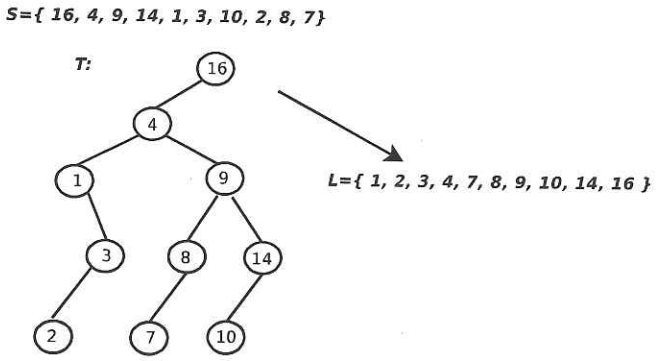


Figura 5.12: Ejemplo de Tree Sort

[2]Son descritos en el Apéndice C.

Análisis de Complejidad

Para determinar el desempeño computacional, suponemos que todos los elementos en la lista L a ordenar son distintos.

Peor Caso

Si los elementos en la lista S ya están ordenados, entonces el árbol de búsqueda binaria T construido será una trayectoria, cada nueva inserción necesitará recorrer toda la trayectoria. Para n elementos, el número total de inserciones es:

$$1 + 2 + 3 + \cdots + (n-1) = n \cdot (n-1)/2.$$

Podemos concluir que, en el peor caso, el tiempo de ejecución es $O(n^2)$.

La Figura 5.13 presenta ejemplos con el peor caso: cuando la secuencia está ordenada en forma descendente, $S_1 = \{16, 14, 10, 9, 7, 4, 1\}$, y en forma ascendente, S_2.

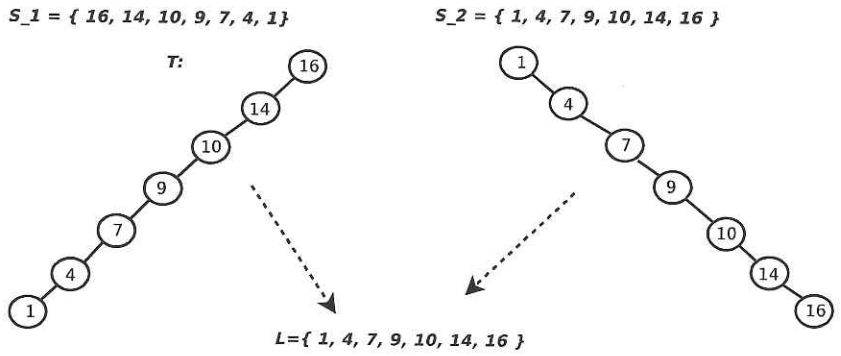


Figura 5.13: Peor caso de Tree Sort

Es posible reducir el tiempo, en el peor caso, a $O(n \cdot \log n)$ si usamos árboles AVL en vez de árboles de búsqueda binaria [3].

Mejor Caso

Este caso ocurre cuando el árbol resultante T está perfectamente balanceado, excepto quizá en el último nivel. En este caso, cada inserción requiere tiempo $O(\log n)$, por lo que las n inserciones requieren en total tiempo: $O(n \cdot \log n)$. Así, el desempeño computacional del Tree Sort, en el mejor caso es $O(n \cdot \log n)$.

La Figura 5.14 presenta un ejemplo con el mejor caso. La secuencia a ordenar es: $S = \{8, 12, 4, 14, 2, 10, 6, 13, 1, 5, 11, 3, 15, 7, 9\}$ y consta de los primeros quince números naturales.

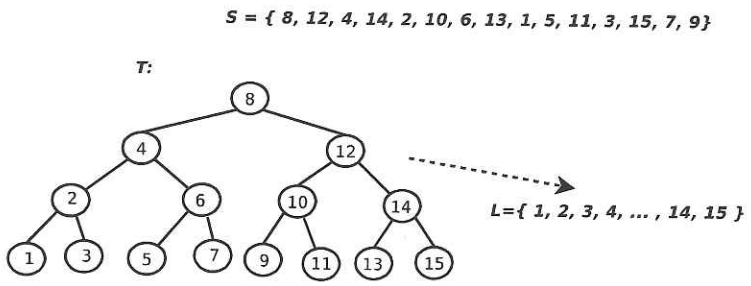


Figura 5.14: Mejor caso de Tree Sort

Caso Esperado

Consideramos el caso esperado sobre todas las $n!$ formas de organizar una secuencia de tamaño n. Así el número de iteraciones o llamadas recursivas es aproximadamente n veces la altura promedio (esperada) de un árbol de búsqueda binaria. Sabemos que tal altura promedio es $O(\log n)$. Por lo tanto, en el caso esperado, el desempeño computacional del Tree Sort es $O(n \cdot \log n)$.

La Figura 5.15 presenta un ejemplo con el caso esperado, la secuencia a ordenar es: $S = \{8, 12, 4, 14, 2, 9, 13, 1, 3, 15, 10\}$.

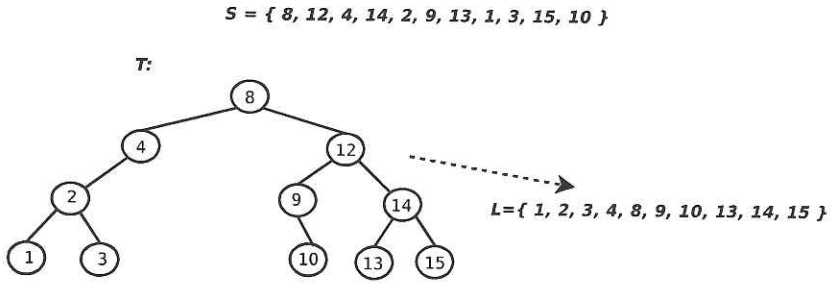


Figura 5.15: Caso Esperado de Tree Sort

Algoritmo

El Listado 25 presenta un pseudocódigo para Tree Sort, supone que S es una secuencia de números enteros diferentes, no vacía y de tamaño n, contenida en un arreglo A.

Listado 25 pseudocódigo Tree Sort

```
// PreC: S una secuencia con n elementos distintos,
//         contenida en un arreglo A[1,n].
// PostC: se regresa un arreglo que contiene en orden
//        ascendente a los elementos de S.

array TreeSort(array A){
  bst T;                        // arbol de busqueda binaria
  int i;                        // contador

  T.create;                     // crea el arbol
  For (i=1; i=n; i++)           // inserta los elementos en el arbol
     T.Insert ( A[i] );

  T.recorreInOrder(A[]);        // recorre el arbol en-orden
                                // dejandolo en el arreglo A

  return A;                     // regresa el arreglo ordenando.

}// end HeapSort
```

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.

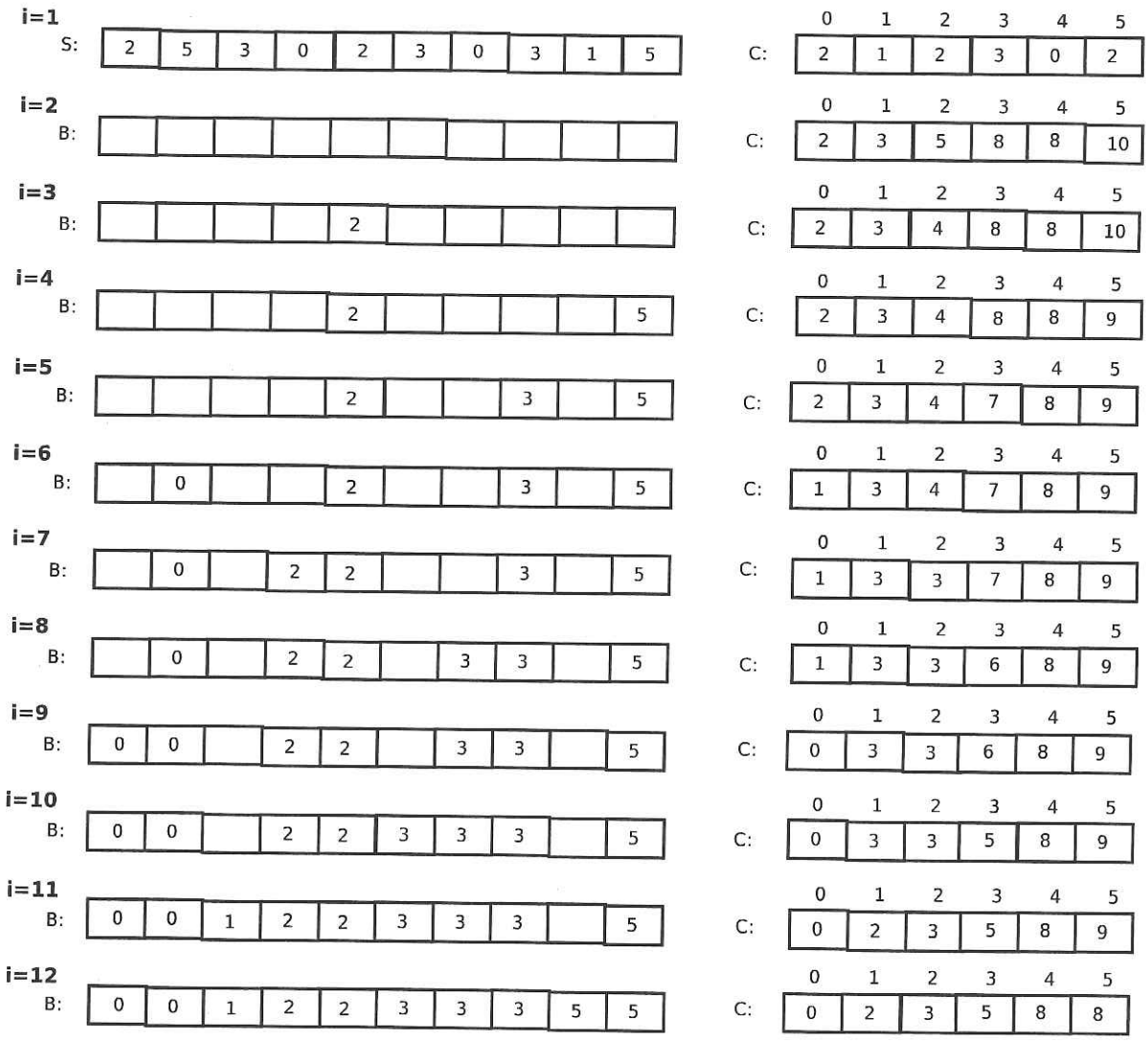


Figura 6.1: Ejemplo de Counting Sort

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:

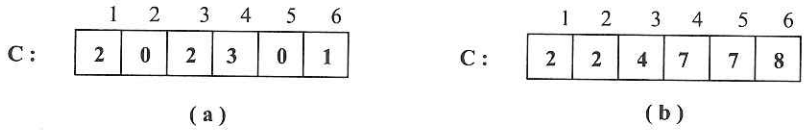


Figura 6.2: Arreglo C después del 2^o y 3^{er} for, respectivamente.

El tercer for se trabaja de la siguiente manera:

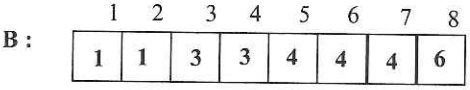
$C[2] \leftarrow C[2] + C[1] = 0 + 2 = 2; \quad C[3] \leftarrow C[3] + C[2] = 2 + 2 = 4;$
$C[4] \leftarrow C[4] + C[3] = 4 + 3 = 7; \quad C[5] \leftarrow C[5] + C[4] = 0 + 7 = 7;$
$C[6] \leftarrow C[6] + C[5] = 1 + 7 = 8.$

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

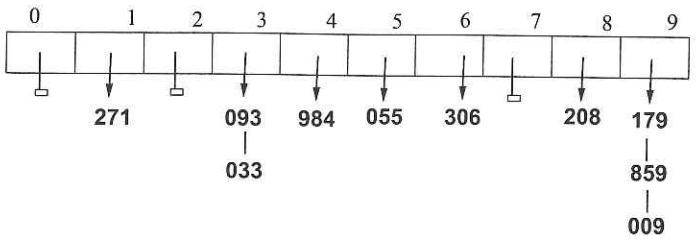


Figura 6.3: Radix Sorting, propagación sobre el último dígito.

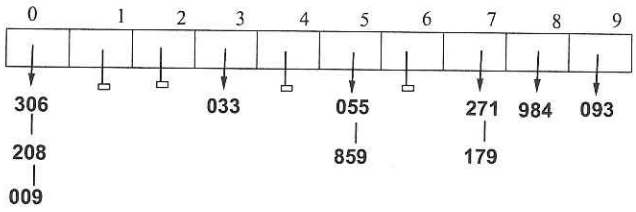


Figura 6.4: Radix Sorting, propagación sobre el segundo dígito.

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:

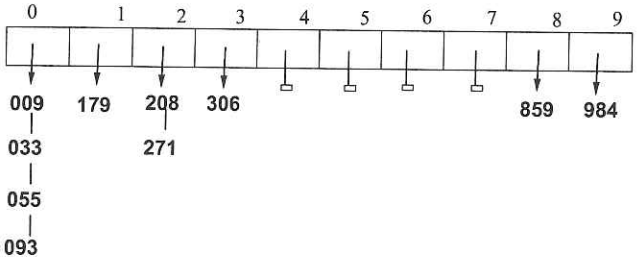


Figura 6.5: Radix Sorting, propagación sobre el primer dígito.

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}
271		306		009
093		208		033
033		009		055
984		033		093
055		055		179
306		859		208
208		271		271
179		176		306
859		984		859
009		093		984

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

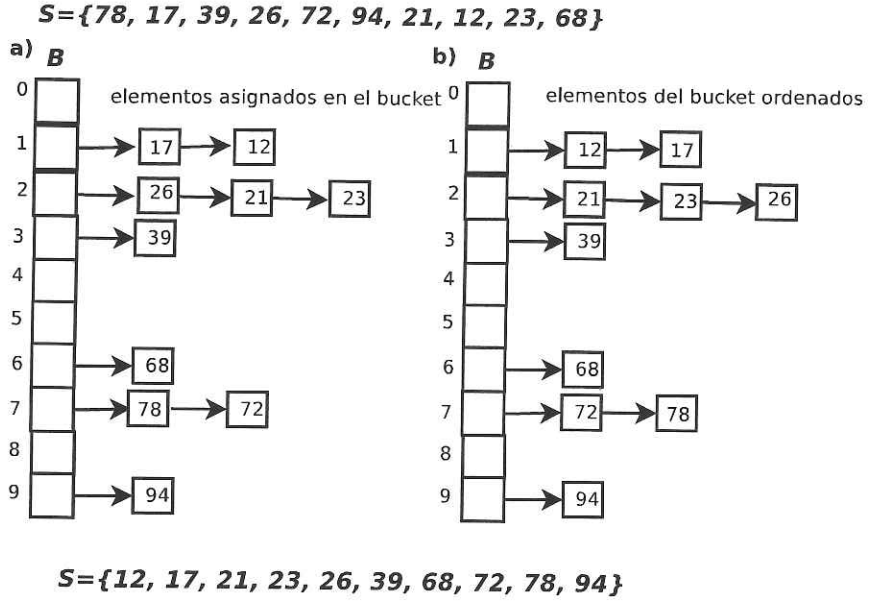


Figura 6.6: Ejemplo de Bucket Sort

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.

Capítulo 7

La Cota Mínima del Ordenamiento

Los algoritmos de ordenamiento se pueden clasificar en dos grupos: los que emplean comparaciones para realizar el ordenamiento y los que no. Los que no emplean comparaciones tienen un mejor desempeño comparado con los que sí las utilizan, aunque tienen como desventaja cumplir ciertas suposiciones sobre la secuencia para poder ser usados.

Los métodos de ordenamiento que no emplean comparaciones son llamados técnicas de ordenamiento lineal ya que tienen un desempeño de $O(n)$. Por otro lado, el desempeño de las técnicas basadas en comparaciones oscila entre $O(n^2)$ y $O(n \log n)$. El desempeño de los métodos basados en comparaciones mejoró conforme se empleaban estrategias más ingeniosas o se usaban estructuras de datos auxiliares. Considerando esto, surgen las siguientes preguntas: ¿Será posible mejorar ese tiempo? ¿Cuál será el mejor desempeño para una técnica de este tipo?

Una Cota mínima para un problema particular es una evidencia de que *ningún algoritmo* puede resolver mejor el problema. Resulta más díficil probar una cota mínima, ya que tenemos que revisar *todos* los posibles algoritmos y no sólo uno. Necesitamos definir un modelo que corresponda a un algoritmo arbitrario y probar que el desempeño computacional para cualquier algoritmo que se ajuste a este modelo debe ser mayor o igual a la cota mínima.

Trabajaremos con un modelo de cómputo llamado Árbol de Decisión, el cual consiste principalmente de comparaciones. Los árboles de decisión no son modelos generales de computación como la máquina de Turing. Existen diversas variantes de los árboles de decisión, las cuales han sido utilizadas para obtener la mayoría de las cotas mínimas conocidas. Definimos árbol de decisión como árboles binarios con dos tipos de nodos: *nodos internos* y *hojas*. Cada nodo interno está asociado con una *pregunta* cuya respuesta es una de dos posibilidades, cada una de las cuales, a su vez, está asociada a una de sus ramas. Cada hoja está asociada a una posible salida o *pregunta*.

Así que, para responder las dos preguntas formuladas es conveniente abordar el problema con otro enfoque: en lugar de centrarnos propiamente en la técnica nos centraremos en el árbol de decisión asociado a ella. Informalmente, un árbol de decisión es un árbol binario en el cual cada hoja representa una posibilidad de orden. En la Figura 7.1 se presenta un árbol de decisión para tres elementos.

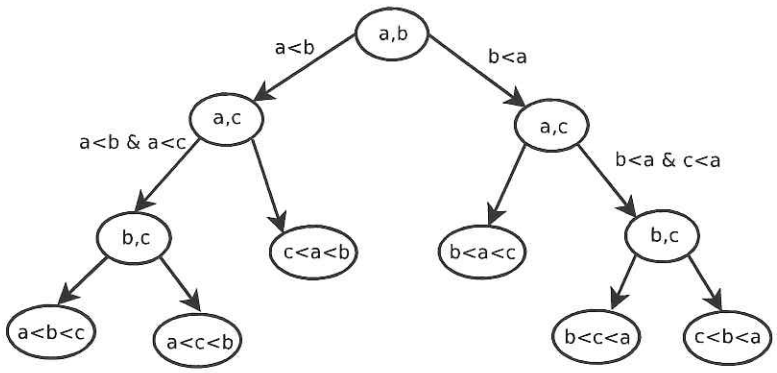


Figura 7.1: Árbol de decisión con tres elementos

Para este ejemplo suponemos que la entrada es una secuencia S de tres números: $S = \{a, b, c\}$. El cálculo empieza en la raíz del árbol. En cada nodo la pregunta se aplica a la entrada, de acuerdo a la respuesta se toma la rama izquierda o la derecha. En el ejemplo, se compara a a con b; si a resulta ser menor que b, toma la rama izquierda y ahora compara a a con c; si a es menor que c, entonces compara a b con c. Análogamente, se construye el sub-árbol derecho. De esta forma se va deduciendo el orden de la secuencia S. Cuando se llega a una hoja, la salida asociada a la hoja es justo el cálculo de comparaciones hechas para ordenar la secuencia S.

La acción de ordenar una entrada específica corresponde entonces a seguir un camino desde la raíz hasta una hoja, en donde los nodos internos por los que se pasa representan las comparaciones efectuadas por el algoritmo.

Es claro que, el tiempo de ejecución, en el peor de los casos asociado al árbol de decisión T, es la altura del árbol, la cual representa el máximo número de preguntas requeridas por la entrada. Un árbol de decisión construido de esta manera corresponde a un algoritmo. Aunque un árbol de decisión no puede modelar toda clase de algoritmos, son modelos razonables para algoritmos basados en comparaciones. Una cota mínima obtenida por árboles de decisión implica que *ningún* algoritmo basado en este modelo puede ejecutarse de mejor manera.

A continuación, usaremos los árboles de decisión para obtener la cota mínima de para el problema de ordenamiento. Revisaremos dos versiones que resultan ser equivalentes.

Primera Versión.

Sabemos que para una secuencia de n elementos, existen $n!$ maneras de acomodarlos, y como cada hoja dentro del árbol representa una posibilidad de orden, se infiere fácilmente que el árbol tendrá $n!$ hojas. Además, no necesariamente todas las hojas están al mismo nivel, como se aprecia en el árbol de la Figura 7.1.

Empleando un enfoque pesimista, tenemos que en el peor caso el número de comparaciones realizadas por una técnica de ordenamiento es igual a longitud de la trayectoria

más larga desde la raíz hasta una hoja. Así, para obtener una cota mínima, en el peor caso, usando una técnica basada en comparaciones debemos determinar una cota inferior para la altura de un árbol, del cual sólo conocemos el número de hojas que tiene.

Como el árbol es binario entonces tiene a lo más 2^h hojas por lo que $n! \leq 2^h$, aplicando logaritmo base dos a ambos lados obtenemos: $\log(n!) \leq h$; es decir, la altura es, por lo menos, $\log(n!)$.

La expresión obtenida para la altura a simple vista se ve compleja, por ello hay que determinar cual es el valor de $n!$, si se emplea la aproximación de Stirling obtenemos $n! \geq (n/e)^n$ donde e es la base del logaritmo natural, entonces tenemos:

$$\log(n!) \geq \log((n/e\ )^n) = n(\log n - \log e) \approx\ n(\log n - 1{,}44{,}3n)$$

Es decir, en el peor caso se realizan $n \log n\ -\ 1{,}443n$ comparaciones. Por lo tanto, hemos encontrado la cota mínima de ordenamiento que es el mejor desempeño que se puede alcanzar, en el peor de los casos, con las técnicas de ordenamiento basadas en comparaciones y que redondeando es de $n \log n$.

Segunda Versión

Teorema 6.1 (Manber) Cada algoritmo tipo árbol de decisión para el problema de ordenamiento tiene altura $\Omega(n \log_2 n)$.

Demostración. Sea S la secuencia de números $S = \{x_1, x_2, ..., x_n\}$ el ejemplar a ordenar. Suponemos que la salida es una permutación de la entrada, la salida indica cómo reorganizar los elementos de tal manera que se transformen en una secuencia ordenada. Cada permutación es una posible salida, ya que la entrada puede estar en cualquier orden inicial.

Un algoritmo de ordenamiento es correcto si es capaz de reorganizar todas las posibles entradas en una secuencia ordenada. Así, cada permutación de $\{1, 2, ..., n\}$ deberá representar una posible salida en el árbol de decisión para ordenamiento. Cada salida, en el árbol, está asociada a una hoja. Dos diferentes permutaciones corresponden a dos diferentes salidas. Por lo tanto, debe haber, al menos, una hoja para cada permutación, es decir, para cada salida.

El número total de permutaciones para n elementos es $n!$, como el árbol es binario, su altura es de, al menos, $\log_2(n!)$. Ahora bien, de acuerdo con la fórmula de Stirling,

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n (1 + O(1/n))$$

De aquí, $\log_2(n!)$ es $\Omega(n \log_2 n)$, lo cual, completa la prueba. ◇

Esta clase de cota es llamada una cota mínima teórica de información, ya que no depende los cálculos, sino de la cantidad de información contenida en la salida.

La cota mínima indica, en este caso, que cada algoritmo de ordenamiento requiere, en el peor de los casos, $\Omega(n \log_2 n)$ comparaciones, pues necesita distinguir entre los $n!$ diferentes casos y compara sólo dos posibilidades a la vez.

Si hubiésemos elegido un árbol de decisión donde cada nodo tuviese tres hijos, digamos, mayor, menor e igual, la altura del árbol sería, al menos, $\log_3(n!)$, la cual, de todas formas es $\Omega(n \log_2 n)$. Esto significa que la cota obtenida se aplica a cualquier árbol de decisión con un número constante de ramas por nodo.

La prueba del Teorema 6.1 implica que ningún algoritmo basado en comparaciones puede ser más rápido que $\Omega(n \log_2 n)$. Puede ser posible ordenar más rápido utilizando propiedades especiales sobre los datos o haciendo manipulaciones algebraicas sobre los datos.

Cabe señalar que, cuando discutimos sobre árboles de decisión, usualmente, ignoramos su tamaño y nos concentramos sólo en su altura. Un simple algoritmo de tiempo lineal puede corresponder a árboles de decisión con un número exponencial de nodos. El tamaño no es importante, ya que no intentamos construir el árbol, lo usamos únicamente como una herramienta para la demostración. Al ignorar el tamaño, hacemos más poderosa la prueba, ya que podemos aplicarla a programas de tamaño exponencial.

Por otro lado, esta técnica puede ser también poderosa. Los árboles de decisión son modelos de cómputo no uniformes. Dependen de n, el tamaño del ejemplar. Es potencialmente posible construir árboles para diferentes valores de n. También, es posible construir árboles de decisión de altura polinomial pero de tamaño exponencial, para problemas que probablemente requieran tiempo de ejecución exponencial.

Tenemos que los árboles de decisión son algunas veces optimistas. Esto es, un árbol de decisión para la cota mínima puede estar muy por debajo de la complejidad real del problema. Por otro lado, si la cota mínima es igual a la cota máxima para un algoritmo en particular, entonces la cota mínima implica que aún si usáramos más espacio, no es posible mejorar el desempeño del algoritmo.

Cabe mencionar que el caso promedio de cualquier algoritmo basado en comparaciones para el problema de ordenamiento también es $\Omega(n \log_2 n)$.

Apéndice A

Algoritmos

Los algoritmos son una parte central de todas las áreas en las ciencias de la computación y, de hecho, forma parte relevante de casi todas las ciencias, de los negocios y de la tecnología.

La naturaleza de un algoritmo se logra apreciar generalmente en todas aquellas disciplinas que se benefician del uso de computadoras. La computadora no solamente es una máquina que puede realizar procesos para darnos resultados, sin que tengamos la noción exacta de las operaciones que realiza para llegar a ellos.

Con la computadora, además de lo anterior, también podemos diseñar soluciones a la medida de problemas específicos. Sobre todo, si estos involucran operaciones matemáticas repetitivas, o que requieren del manejo de un volumen muy grande de datos.

El diseño de soluciones a la medida de nuestros problemas requiere, como en otras disciplinas, una metodología que nos enseñe de manera gradual la forma de llegar a las soluciones. Una forma de obtener soluciones, a través de las computadoras son los programas, que no son más que una serie de operaciones que realiza la computadora para llegar a un resultado, con un grupo de datos específicos.

Lo anterior nos lleva al razonamiento de que un programa nos sirve para solucionar un problema específico. Para poder realizar programas, además de conocer la metodología mencionada, también debemos de conocer, de manera específica las funciones que puede realizar la computadora y las formas en que se pueden manejar los elementos que hay en la misma.

Un **algoritmo** es una secuencia finita de pasos, que resuelve un problema en un tiempo finito (que termina en un número finito de operaciones). Un algoritmo tiene las siguientes características:

1. Trabaja a partir de datos, aunque ocasionalmente podría no recibir entradas.
2. Produce como salida un resultado que corresponde a la solución del problema; o bien termina indicando que no existe tal.
3. Secuencia Finita de pasos. El algoritmo define una secuencia de pasos, bien definidos, cuya ejecución transforma a la entrada en la salida.
4. Correctez. El algoritmo debe terminar y garantizar la solución para cualquier ejemplar del problema.

El propósito del Análisis de Algoritmos es predecir el comportamiento de los algoritmos sin implantarlos en máquina alguna [10]. Además es mucho más conveniente tener medidas sencillas para la eficiencia de un algoritmo que implantarlo y probar su eficiencia cada vez con parámetros específicos para sistemas diferentes. Tradicionalmente, el tiempo de ejecución de un algoritmo depende del tiempo de ejecución y tamaño de memoria utilizado.

Análisis de Algoritmos

El tiempo necesario para ejecutar un algoritmo resulta ser una función que depende generalmente de la cantidad de datos a procesar; mayor cantidad de datos significa más tiempo de ejecución. El valor exacto de esta función depende de varios factores, tales como la velocidad de la máquina. Así el resultado del análisis debe indicar cuánto tarda el algoritmo en cuestión, en tiempo de ejecución, para una entrada particular.

Dado que el número de entradas potenciales es enorme y que seguramente dos algoritmos tendrán comportamiento diferente para entradas distintas, debemos considerar una métrica para la entrada, como su tamaño, y entonces hacer un análisis referente a este tamaño.

Por otro lado, los algoritmos no se comportan de manera similar para todas las entradas del mismo tamaño, pero es imposible analizar todas las entradas del mismo tamaño para todos los algoritmos. En lugar de esto, podemos comparar el desempeño de algoritmos diferentes que resuelvan el mismo problema; pero , ¿cómo elegimos un indicador?

Elegir la mejor entrada usualmente no es muy representativo, porque representa una solución trivial. El caso esperado es una buena elección, pero algunas veces será muy difícil obtener una medida eficiente, pues la esperanza del tiempo de ejecución dependerá de parámetros diferentes o de la forma como están organizados los datos. Finalmente, el cálculo del tiempo de ejecución, usando la peor entrada como indicador es muy usual, aunque represente un análisis pesimista.

Para un programa fijo, ejecutándose en una computadora, podemos dibujar la gráfica que representa la función del tiempo de ejecución.

La Figura A.1 muestra una gráfica sobre cuatro programas, donde las curvas representan 4 funciones típicas en el análisis de algoritmos: *lineal,* $n \log n$*, cuadrática y cúbica.* El tamaño de la entrada n varía de 1 a 100 elementos y los tiempos de ejecución asociados varían de 0 a 10 milisegundos. La Figura A.2 muestra los tiempos de ejecución para los mismos programas, pero con tamaños de entrada mayores.

Para valores pequeños de n, por ejemplo aquellos menores a 35, en la Figura A.1 mostramos que hay puntos para los cuales una curva es inicialmente mejor que otra, aunque esto deja de ser cierto cuando la entrada n se hace "suficientemente" grande. Para tamaños de entradas pequeños, es difícil comparar las funciones. Por ejemplo, consideremos la función $f(n) = n + 2500$, es mayor que n^2 cuando n es menor que 50, pero a partir de este punto la función lineal será siempre menor que la función cuadrática y la constante pierde importancia.

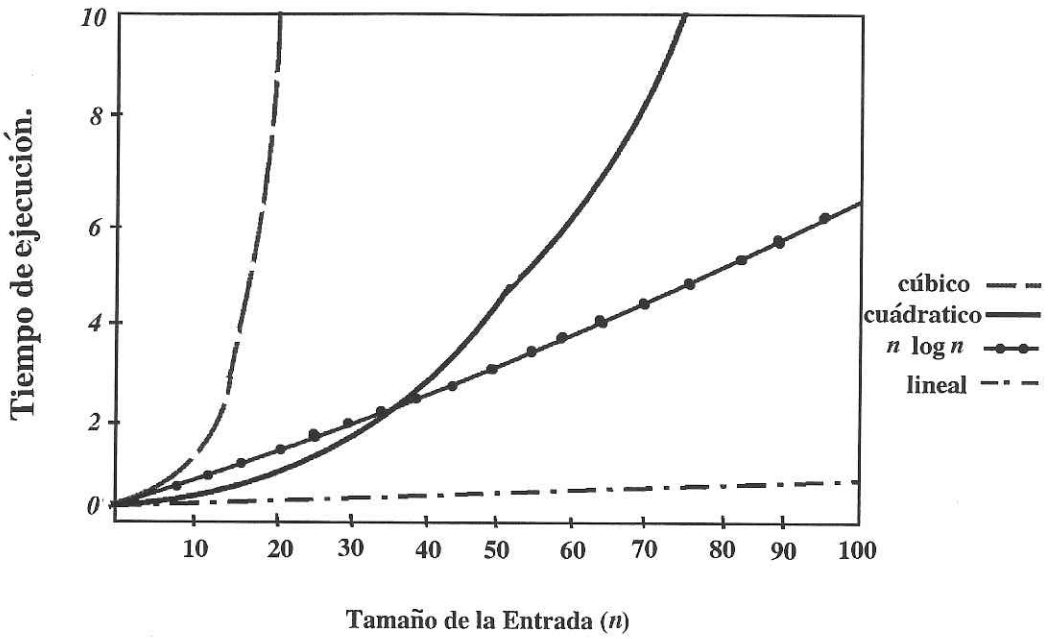


Figura A.1: Tiempos de ejecución para tamaños de entrada pequeños.

Otro punto importante es que para tamaños pequeños de entrada los tiempos de ejecución son insignificantes y comúnmente no tenemos que preocuparnos por ellos.

La Figura A.2 muestra mayores diferencias entre las curvas de la primera figura, cuando el tamaño de la entrada es grande. Un algoritmo resuelve un problema de tamaño 10,000 en una fracción de segundo mientras que el algoritmo $n \log n$ utiliza aproximadamente diez veces este tiempo.

La característica más notoria de estas curvas es que los algoritmos cuadráticos y cúbicos no pueden competir con los restantes para tamaños de entrada razonablemente grandes. El cuadro A.1 ordena de manera creciente distintas funciones que describen comúnmente el tiempo de ejecución de los algoritmos.

Función	Nombre
c	Constante
$\log\ n$	Logarítmica
n	Lineal
$n \log(n)$	$n \log n$
n^2	Cuadrática
n^3	Cúbica
2^n	Exponencial

Cuadro A.1: Funciones de índice de crecimiento.

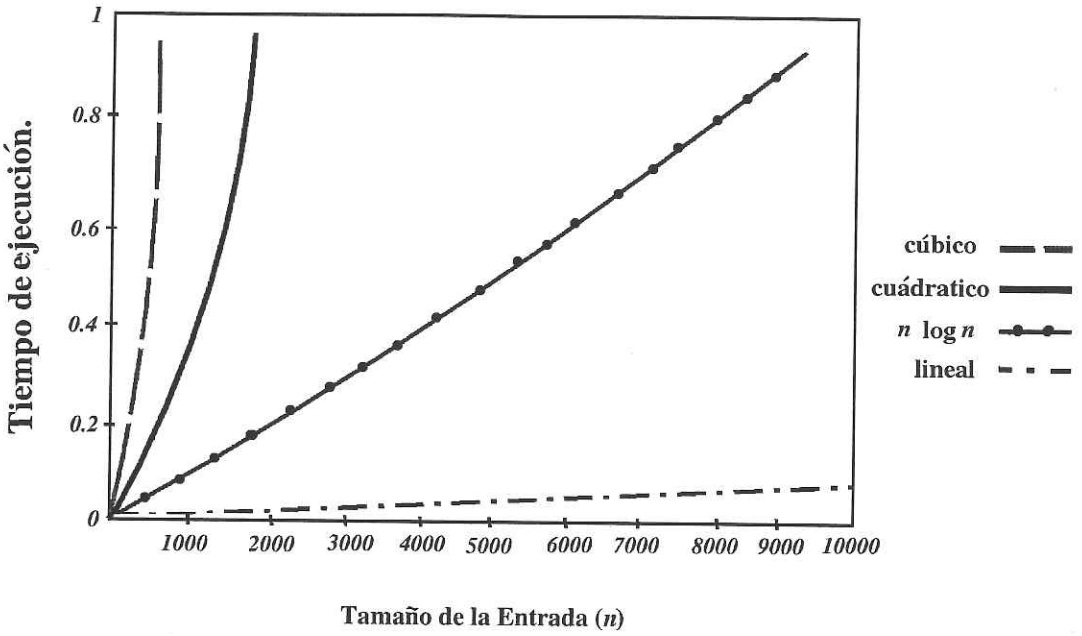


Figura A.2: Tiempos de ejecución para tamaños de entrada mayores.

Notación asintótica.

Una vez que revisamos las ideas básicas del análisis de algoritmos, podemos dar un enfoque más formal para la notación de las diferentes funciones de crecimiento.

Definición A.1 *Notación* **O**. Decimos que $f(n)$ es de orden de $g(n)$ si existen $c > 0$ y n_0 tales que: $f(n) \leq c \cdot g(n) \ \ \forall n \geq n_0$, se denota como $f(n) \in O(g(n))$,

La notación O representa una cota superior para el tiempo de ejecución del algoritmo, garantizando que no se va a tardar más, para n suficientemente grande.

Esta notación es similar a la relación menor o igual, con respecto al crecimiento de funciones, es decir, que f no crece más rápido que g.

Definición A.2 *Notación* Ω . Decimos que $f(n) \in \Omega(g(n))$ si existen $c > 0$ y n_0 tales que: $f(n) \geq c \cdot g(n) \ \ \forall n \geq n_0$.

Esta notación es similar a la relación mayor o igual, es decir, una función no crece más lento que otra.

Definición A.3 *Notación* Θ. Decimos que $f(n) \in \Theta(g(n))$ si y sólo si:
$f(n) \in O(g(n))$ y $f(n) \in \Omega(g(n))$.

Utilizamos esta notación para indicar que las dos funciones tienen un índice de crecimiento similar. Θ representa una categoría de orden.

Definición A.4 *Notación* **o**. Decimos que $f(n)$ es o pequeña de $g(n)$, denotado $f(n) \in o(g(n))$, si para toda $c > 0$ existe $n_0 > 0$ tales que: $0 \leq f(n) < c \cdot g(n) \ \ \forall n \geq n_0$.

La notación **o** es similar al menor estricto, es decir, una función crece estrictamente más lento que otra.

Definición A.5 *Notación* ω. Decimos que $f(n) \in \omega(g(n))$ si para toda $c > 0$ existe $n_0 > 0$ tales que: $0 \leq c \cdot g(n) < f(n) \quad \forall n \geq n_0$. Lo cual es equivalente a decir que: $f(n) \in \omega(g(n))$ si y sólo si $(g(n)) \in o(f(n))$.

La notación ω es similar al mayor estricto, es decir, una función crece estrictamente más rápido que otra.

Generalmente, utilizaremos la notación O para representar el índice de crecimiento de una función, así el tiempo de ejecución de un algoritmo cuadrático se describe como $O(n^2)$. La notación O nos permite establecer un orden relativo entre funciones, comparando los términos dominantes. También hay que notar que lo que se busca siempre es dar cotas lo más ajustadas posibles.

El Cuadro A.2 muestra algunos ejemplos Categorías de orden de complejidad y el nombre que se les asocia.

Categoría	Nombre
$\Theta(1)$	Constante
$\Theta(\log\ n)$	Logarítmico
$\Theta(n)$	Lineal
$\Theta(n \log n)$	$n \log(n)$
$\Theta(n^2)$	Cuadrática
$\Theta(n^3)$	Cúbica
$\Theta(2^n)$	Exponencial

Cuadro A.2: Categorías de orden de complejidad.

Los distintos órdenes de complejidad, en las diferentes notaciones, particionan a las funciones en clases de equivalencia, una por cada orden de complejidad. Listamos a continuación varias propiedades que preservan estas clases:

1. $g(n) \in O(f(n)) \iff f(n) \in \Omega(g(n))$
2. $k \in \mathbb{N},\ f(n) \in O(g(n)) \iff k \cdot f(n) \in O(g(n))$
3. $g(n) \in \Theta(f(n)) \iff f(n) \in \Theta(g(n))$
4. Si $b > 1,\ a > 1,\ \log_a n \in \Theta(\log_b n)$
5. Si $0 < a < b\ ,\ a^n \in o(b^n)$
6. $\forall\ 0 < a,\ a^n \in o(n!)$

Solución de Recurrencias

Generalmente, resolvemos los problemas usando inducción, así diseñamos el algoritmo recursivamente, usando el paradigma divide y vencerás.

Una recurrencia (o función recurrente) es una ecuación o desigualdad que describe una función en término de sus valores para ejemplares más pequeños. De esta forma, una recurrencia caracteriza el tiempo de ejecución de los algoritmos diseñado con la estrategia divide y vencerás.

Las recurrencias pueden tomar diversas formas. Por ejemplo, el desempeño computacional del MergeSort está determinado por:

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1; \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) & \text{si } n > 1. \end{cases}$$

Existen tres métodos para resolver las recurrencias:

1. **Método de Sustitución.**– Se propone una cota y se demuestra usando Inducción Matemática.
2. **M. del Árbol de recursión.**– Se construye un árbol cuyos nodos representan los costos incurridos en cada nivel de la recursión. Para resolver las recurrencias se usan técnicas para acotar las sumas.
3. **Método Maestro.**– Se determinan cotas para recurrencias de la forma: $T(n) = a \cdot T(n/b) + f(n)$, donde $a \geq 1$, $b > 1$ y $f(n)$ es una función dada. Tales recurrencias son muy comunes y caracterizan algoritmos del tipo divide y vencerás que genera a subproblemas, cada uno de los cuales es $(1/b)$ el tamaño del problema original y en el cual la división y mezcla de los pasos toma tiempo $f(n)$.

El Método Maestro depende del siguiente teorema:

El Teorema Maestro (*The Master Theorem* [4])

Sean $a \geq 1$ y $b > 1$ dos constantes, sea $f(n)$ un función y sea $T(n)$ una recurrencia sobre enteros no negativos dada por: $T(n) = a \cdot T(n/b) + f(n)$, donde n/b puede ser o $\lfloor n/b \rfloor$ o $\lceil n/b \rceil$. Entonces $T(n)$ tiene las siguientes cotas asintóticas:

1. Si $f(n) = O(n^{\log_b a - \epsilon})$ para alguna constante $\epsilon > 0$, entonces $T(n) = \Theta(n^{\log_b a})$.
2. Si $f(n) = \Theta(n^{\log_b a})$ entonces $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. Si $f(n) = \Omega(n^{\log_b a + \epsilon})$ para alguna constante $\epsilon > 0$, y si $a \cdot f(n/b) \leq c \cdot f(n)$ para alguna constante $c < 1$ y para toda n suficientemente grande, entonces $T(n) = \Theta(n^{\log_b a})$.

Aplicando el Teorema Maestro, tenemos que el desempeño computacional del MergeSort es $\Theta(n \log n)$.

Apéndice B

Tablas

En este apartado describiremos brevemente el Tipo de Datos Abstracto Tabla, después presentamos las Tablas Hash.

TDA: Tabla

Una tabla es apropiada para problemas que deben manipular datos por valor. Existen diversas implementaciones para las tablas (las cuales usan arreglos, listas ligadas o árboles binarios). En esta sección, solo definiremos el Tipo de datos Abstracto y las operaciones que lo definen. La siguiente figura representa una tabla.

#	llave	Dato
0		
	...	
999		

En el tipo de datos Tabla, el dominio lógico es la colección de todas las tablas, por lo que cada ejemplar es una Tabla. Una Tabla es una función, específicamente, un conjunto de pares ordenados de la forma $\langle$ *llave*, *dato* $\rangle$. Una tabla no posee explicítamente un orden. Las llaves podrían estar ordenadas, pero no es necesario que lo estén. Supondremos que:

1. El TDA item posee un método llamado KeyOf que regresa la llave de un elemento;
2. MaxSize representa al máximo número de elementos permitidos en la tabla;
3. Se tienen dos funciones relacionadas con la llave:
KeyNumber(Key):Integer Regresa un entero no negativo correspondiente a la llave. Con esta función, el diseñador del TDA Tabla no necesita preocuparse por si las llaves son números, cadenas o estructuras más sofisticadas.
EqualKeys(k1, k2): Boolean Indica si las llaves dadas son o no iguales.

Responsabilidades

A diferencia de otras estructuras clásicas, una tabla no posee un primer elemento, un siguiente o un último elemento, ni un elemento raíz o un elemento hijo. Un usuario no puede recuperar un elemento basándose en la posición directa del elemento en la tabla o iniciar una búsqueda dada una posición. Dejando a un lado estas prohibiciones, el TDA Tabla permite realizar operaciones usuales como:

+ Crear una tabla vacía;
+ Determinar si una tabla es o no vacía;
+ Indicar el tamaño de la tabla;
+ Recorrer la tabla;
+ Hacer copias de una tabla;
+ Insertar un dato, de acuerdo a la llave;
+ Borrar un dato, de acuerdo a la llave;
+ Consultar un dato, de acuerdo a la llave.

TDA: Tablas Hash

En una tabla hash los datos son almacenados de acuerdo a una función de dispersión o función hash que procesa la llave para generar una dirección en la tabla. Generalmente una tabla hash tiene un campo adicional que indica si la localidad está o no ocupada. A continuación se presenta una tabla hash, cuyos atributos son: ID: representa la llave; Nombre: caracteriza el dato y EnUso: indica si la localidad está en uso (TRUE) o vacía (FALSE)

#	ID:	Nombre:	EnUso:
1			
	...		
999			

Suponga que se tienen dos datos, E_1 y E_2 con llaves k_1 y k_2, respectivamente, al procesarlos con una función hash h, se tiene que $h(k_1) = h(k_2) = d_1$; el dato E_1 fue colocado en la dirección d_1 y el otro fue puesto en la localidad d_2. Suponga, además, que al manipular los datos en la tabla se borró al dato E_1. Después, al consultar si el dato E_2 está en la tabla, lo primero que se hace es aplicarle la función h, lo que indicaría que si está se encuentra en la dirección d_1, pero en ésta no hay dato. Por lo que concluiríamos que E_2 no es parte de la tabla, lo cual es un grave error!

Para evitar esta clase de confusiones, generalmente, se agrega un atributo más al a tabla: FueB: que indica que ahí hubo un elemento que fue borrado. De esta manera, al

hacer la búsqueda anterior para E_2, se haría un segundo intento, al menos, para localizar al dato E_2.

A continuación se presenta una tabla hash que incluye el atributo FueB:

#	ID:	Nombre:	EnUso:	FueB:
1				
	...			
999				

Otra variante consiste en agregar un atributo que indique a que dirección inmediata fue enviado el dato que colisiona, en vez de usar el atributo FueB, así es más fácil realizar las búsquedas. A continuación se presenta un ejemplo de este caso, el nuevo atributo es denominado ReD, de re-direccionado:

#	ID:	Nombre:	EnUso:	reD:
1				
	...			
999				

Apéndice C

Propiedades de Árboles

En este apartado daremos una breve revisión sobre árboles binarios y, en particular se presenta una muy breve introducción a los árboles binarios de búsquedas, *binary search trees*. Estos temas se revisan con detalle en el curso de Introducción a Ciencias de la Computación II o Estructuras de Datos.

Árboles binarios

Un árbol binario es vacío, o bien, consiste de un nodo raíz y un par de árboles binarios (posiblemente vacíos), denominados: subárbol izquierdo, T_L, y derecho, T_R.

En la Figura C.1(a) se muestra un árbol binario constituido únicamente por el nodo raíz; en (b) se ilustra la definición de árbol binario; y en (c) se presenta un árbol binario.

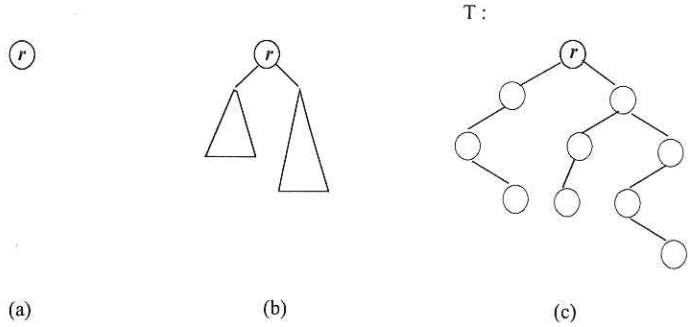


Figura C.1: Ejemplos de árbol binario

Cada elemento en un árbol binario está determinado de manera única por su posición en el árbol.

Considere la Figura C.2, aquí decimos que el nodo x es el padre de y y de z; y es el hijo izquierdo de x y z es el derecho.

En árbol binario la raíz es el único elemento que no tiene padre; todos los demás tienen exactamente un padre. Un elemento (nodo) en el árbol que no tiene hijos se llama hoja.

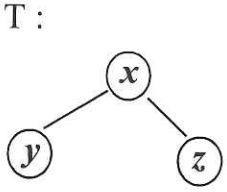


Figura C.2: Posiciones en un árbol binario

Rigurosamente, la **altura** de un árbol binario $h(T)$ es el número de aristas entre la raíz y la hoja más lejana. De manera formal se define $h(T)$ como sigue:

Si T es vacío, entonces $h(T) = -1$; y, en otro caso, $h(T) = 1 + \text{máx}\{h(T_L), h(T_R)\}$.

Sea T un árbol binario, no vacío, para cada elemento x en T, se define $\ell(x)$, el nivel del nodo x, como sigue: Si x es el elemento raíz, $\ell(x) = 0$; y, e.o.c. $\ell(x) = 1 + \ell(padre(x))$.

Se dice que un árbol binario está perfectamente balanceado si sus subárboles, derecho e izquierdo, tienen la misma altura y ambos, o son vacíos o están perfectamente balanceados.

Se dice que un árbol binario es completo si:

(1) T está perfectamente balanceado excepto, posiblemente, por el último nivel.

(2) Todas las hojas en el nivel más bajo están lo más a la izquierda posible.

Se tiene que todo árbol binario perfectamente balanceado es completo.

Propiedades de los árboles binarios

Al dibujar árboles binarios, algunas veces es conveniente representar explícitamente los subárboles vacíos, usando, por ejemplo, un cuadrado. A esta representación se le denomina *árbol binario extendido*. La Figura C.3 muestra un árbol binario y su representación extendida.

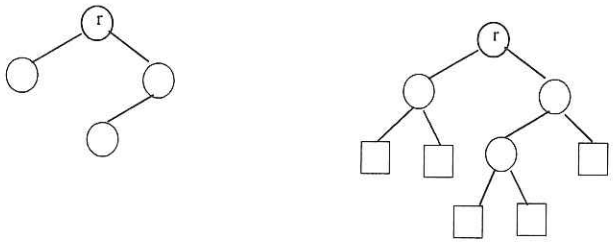


Figura C.3: Ejemplo de árbol binario extendido

Los nodos cuadrados se llaman *nodos externos* y $E(T)$ es el conjunto de todos los nodos externos de un árbol binario T. Los demás nodos en T se llaman *nodos internos* y el conjunto de todos los nodos internos de un árbol binario T es denotado por $I(T)$.

Afirmación C.1 Un árbol binario con n nodos internos tiene $(n + 1)$ nodos externos.

En la Figura C.3, se tiene un árbol con cuatro nodos internos y cinco nodos externos.

En un árbol binario cada nodo tiene un único *padre* (el nodo justo arriba de él), excepto el nodo raíz que no tiene padre. Los *antecesores* de un nodo x se definen, recursivamente, como el conjunto que contiene a x y a los antecesores de su padre, si x tiene padre. Estos nodos forman una *trayectoria* o *ruta* que va desde x hacia la raíz del árbol.

Cada nodo tiene a lo más dos hijos (justo debajo de él). Dos nodos son hermanos[1] si comparten el mismo padre. Un nodo interno sin hijos es una *hoja*.

El conjunto de *descendientes* de un nodo x es x junto con todos los descendientes de sus hijos. Estos nodos, junto con las aristas que los unen, forman el subárbol enraizado en x, que se denotará como T_x. El número de descendientes de un nodo x será denotado como $s(x)$ y, además, representa el tamaño del subárbol T_x.

La *altura* $h(x)$ de un nodo x en un árbol binario T, es el número de aristas sobre la longitud de la ruta más larga de x a un nodo externo. Alternativamente, es el número de nodos internos sobre la ruta que une a x con una hoja. La altura de un árbol binario T, $h(T)$, es la altura de su nodo raíz.

La *profundidad*, $d(x)$, de un nodo x es el número de aristas sobre la ruta que va de la raíz a x. Alternativamente, es el número de nodos internos sobre tal ruta excluyendo a x.

Considerando el árbol binario de la Figura C.4(a), en (b) y (c) se muestra su representación extendida; se ha escrito en el interior de sus nodos, en (b) la altura de cada nodo y en (c) la profundidad.

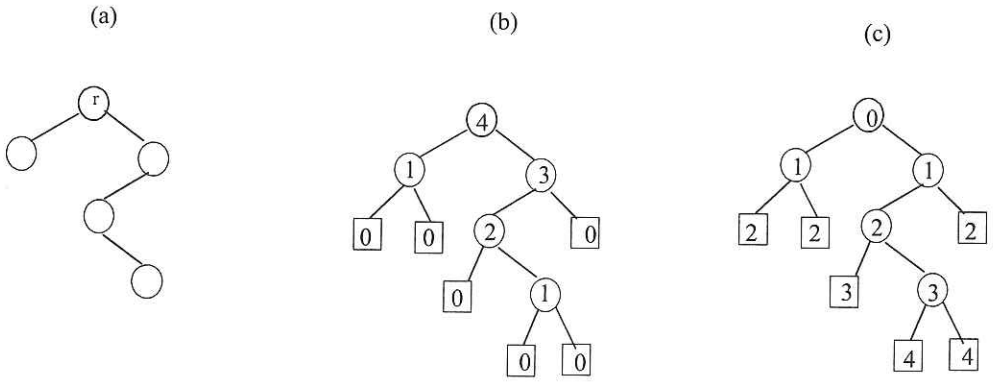


Figura C.4: La altura y profundidad de los nodos de árbol binario

Para un árbol binario T, se define la *longitud de la ruta interna*, $\iota(T)$ como la suma de las profundidades de los nodos internos de T; la longitud de la ruta externa $\epsilon(T)$ es la suma de las profundidades de los nodos externos de T. Esto es,

$$\iota(T) = \sum_{x \in I(T)} d(x); \qquad \epsilon(T) = \sum_{x \in E(T)} d(x).$$

Teorema C.1 Si T es un árbol binario con n nodos internos entonces $\epsilon(T) = \iota(T) + 2n$.

[1] usaremos hermano como traducción de *siblings*

Demostración. Inducción sobre n, número de nodos.

Caso Base. Si $n = 0$ entonces T es vacío y $\epsilon(T) = \iota(T) = 0$, por lo que, el teorema se satisface.

Hipótesis de Inducción. Suponemos que el teorema se cumple para todo árbol binario con j nodos internos, para toda j tal que $1 \leq j \leq k$.

Paso Inductivo. Sea T un árbol binario, no vacío, con $(k+1)$ nodos internos. Por demostrar que el teorema se cumple para T.

Podemos suponer que el subárbol derecho, T_L, tiene i nodos internos y el izquierdo, T_R, tiene $(k-i)$ nodos internos. La Figura C.5 ilustra esta idea.

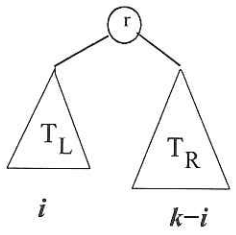


Figura C.5: Árbol binario con $(k+1)$ nodos internos

Ya que $0 \leq i \leq k$ y $0 \leq k - i \leq k$, entonces, para cada subárbol, se cumple el teorema. Es decir,

$$\epsilon(T_L) = \iota(T_L) + 2i \quad \text{(C.1)}$$
$$\epsilon(T_R) = \iota(T_R) + 2(k-i) \quad \text{(C.2)}$$

Ahora bien, nótese que cada nodo externo es un nivel más profundo en T que en los subárboles T_L o T_R. Como T tiene $(k+1)$ nodos internos entonces, por la Afirmación C.1, T tiene $(k+1)+1 = k+2$ nodos externos. Por lo tanto,

$$\epsilon(T) = \epsilon(T_L) + \epsilon(T_R) + (k+2). \quad \text{(C.3)}$$

Análogamente, los k nodos internos de los subárboles T_L y T_R están un nivel más profundo en T y, además, la raíz tiene profundidad cero, así que,

$$\iota(T) = \iota(T_L) + \iota(T_R) + k. \quad \text{(C.4)}$$

Sustituyendo las ecuaciones C.1 y C.2 en C.3:
$\epsilon(T) = \epsilon(T_L) + \epsilon(T_R) + (k+2)$
$\epsilon(T) = [\ \iota(T_L) + 2i\ ] + [\ \iota(T_R) + 2(k-i)\ ] + (k+2)$
$\epsilon(T) = [\ \iota(T_L) + \iota(T_R) + k\ ] + 2 + 2i + 2(k-i)$ reorganizando
$\epsilon(T) = \iota(T) + 2 \cdot [\ 1 + i + k - i\ ]$ por la Ecuación C.4
$\epsilon(T) = \iota(T) + 2 \cdot (k+1)$.

Por lo tanto, podemos concluir que el teorema se satisface para todo árbol binario con $(k+1)$ nodos internos.

Árboles binarios de búsqueda

En este apartado definimos a los árboles binarios de búsquedas, así como la función que inserta un nodo en el árbol y ejemplificamos estos conceptos.

Los árboles binarios de búsquedas satisfacen siempre una condición denominada el *Invariante del árbol binario de búsqueda:* la llave de cualquier nodo es mayor que las llaves de los nodos en su subárbol izquierdo y menor que todas las llaves en los nodos de su subárbol derecho. La Figura C.6 muestra un árbol binario de búsqueda.

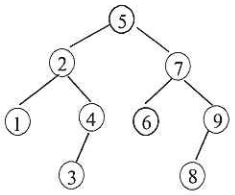


Figura C.6: Ejemplo de árbol binario de búsqueda

La función Insertar(x:nodo) de un árbol binario de búsqueda, se define fácilmente de manera recursiva: se compara la llave de x con la de la raíz; si es menor se debe insertar en el subárbol izquierdo, si es mayor se debe insertar en el subárbol derecho. Si el árbol es vacío, el nodo x se convertirá en el árbol y será el nodo raíz.

A continuación damos un ejemplo. Considere la lista $\mathcal{L} = \{5, 7, 2, 4, 9, 6, 3, 8, 1\}$.

Queremos construir un árbol binario de búsqueda T. Inicialmente T está vacío. Al llegar el nodo con el elemento 5, se convierte en la raíz de T; después llega el 7, que es mayor que el 5, por lo que es insertado a la derecha de éste; luego llega el 2, es comparado con el 5, es menor y es insertado a la izquierda de la raíz. La Figura C.7(a) ilustra la construcción parcial del árbol al insertar estos primeros tres datos.

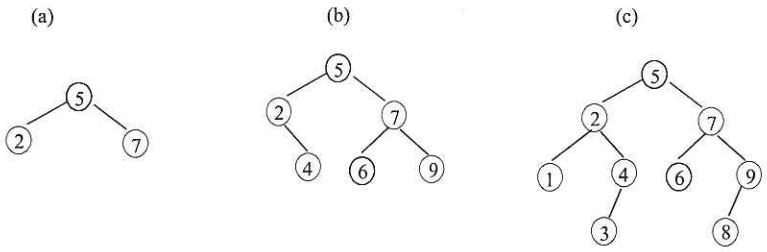


Figura C.7: Construcción de un árbol binario de búsqueda

Continuando con los siguientes elementos: el 4 es menor que el 5, pero mayor que el 2, por lo cual es insertado a la derecha del 2; el 9 es mayor que el 5 y el 7 y es insertado a la derecha del 7; y el 6 es mayor que el 5 y menor que el 7, va a la izquierda del 7. La Figura C.7(b) muestra la construcción parcial del árbol al insertar estos tres datos.

Finalmente, la Figura C.7(c) presenta la construcción total del árbol al insertar los elementos de la lista $\mathcal{L} = \{5, 7, 2, 4, 9, 6, 3, 8, 1\}$.

Apéndice D

Colas de Prioridades

En este apéndice describiremos brevemente las colas de prioridades y los heaps binarios, material que es requerido para los ordenamientos.

Colas de Prioridades

Una cola de prioridades de tipo T con prioridades reales es un Tipo de Datos Abstracto, que denotaremos como $ColaP$, cuyos elementos son conjuntos Q de objetos en T. Cada uno de los elementos en Q tiene asociado un número real, k, denominado llave del objeto.

En una cola de prioridades se desea mantener operaciones que manipulen al objeto de mayor prioridad, tales como:

1. Consultar al objeto con mayor prioridad;
2. Eliminar al objeto con mayor prioridad;
3. Insertar al objeto según su prioridad.

Para nuestra especificación, suponemos que la mayor prioridad la tiene el elemento cuya llave es mínima, y a éste le llamaremos elemento mínimo.

Las operaciones válidas sobre los elementos de una cola de prioridades Q son:

Inicia, *Create.*– Crea una nueva estructura de tipo Cola de prioridades.

Vacio, *Empty.*– Indica si la colección de objetos Q es o no vacía.

Inserta(x:objeto; k: llave), *Insert.*– Agrega un nuevo objeto x, con llave k, a Q.

EncuentraMin, *FindMin.*– Encuentra al objeto cuya llave es mínima en la colección.

BorraMin: objeto, *DeleteMin.*– Regresa al elemento mínimo de la colección y lo elimina de Q.

Elimina(x:objeto), *Delete.*– Elimina al objeto x de la colección Q.

DecrementaLlave (x:objeto; nk: llave), *DecreaseKey.*– Cambia el valor de la llave del objeto x, sólo si nk es menor que el valor de la llave anterior.

MinQ.– Apuntador al mínimo elemento en la cola de prioridades.

Funde($Q1, Q2$: **ColaP**), *Meld.*– Une dos Colas de prioridades en una.

Heaps

Una de las más conocidas estructura de datos para las Colas de Prioridades, son sin duda los Heaps Binarios. Iniciamos definiendo qué es un Heap Binario y después describiremos algunas de sus funciones básicas.

Un **Heap Binario** T es un árbol binario[1] completo en el cual, T es vacío o, bien,:

1.- Cada elemento en el subárbol izquierdo T_L es menor o igual que el elemento raíz de T.
2.- Cada elemento en el subárbol derecho T_R es menor o igual que el elemento raíz de T.
3.- Los subárboles T_L y T_R son heaps binarios.

Si un árbol binario, cumple las condiciones anteriores, se dice que satisface la propiedad de ser Heap Ordenado.

A continuación damos otra manera de definirlos: Un heap es un árbol binario T que satisface las siguientes condiciones:

1. El árbol T es completo a la izquierda, es decir, es completo hasta la profundidad $(h-1)$ y el nivel a profundidad h se llena de izquierda a derecha.
2. Todos los nodos de T satisfacen la propiedad de ser heap ordenado.

Los Heaps son estructura de datos interesantes ya que:

(1) admiten una representación sencilla, compacta y eficiente;
(2) se pueden implementar fácilmente con una cola de prioridades.

Hemos definido a los heaps binarios, considerando que el elemento de mayor prioridad tiene la menor llave, pero en realidad podemos hablar de dos tipos de heaps:

Heap maximal.– las llaves de los nodos padre son siempre mayores o iguales a las de los nodos hijos.
Heap minimal.– las llaves de los nodos padre son siempre menores o iguales a las de los nodos hijos.

[1]El Apéndice C presenta la definición y propiedades de los árboles binarios.

Operaciones

Las operaciones válidas sobre los elementos de un Heap Binario H:

Inicia, *Create* .– Crea un nuevo Heap Binario

Vacio, *Empty* .– Indica si el heap H es o no vacío.

Inserta_al_Final (x**:objeto;** k**: llave),** *Insert_Last.*– Incluye al nuevo objeto x, con llave k, al final del heap, en la última posición H.

BorraMin : objeto, *DeleteMin* .– Regresa al objeto cuya llave es mínima en la colección y lo elimina de H y reorganiza el heap.

Elimina_al_Ultimo(x**:objeto),** *Delete_Last* .– Elimina al elemento que está en la última posición del heap H.

DecrementaLlave (x**:objeto;** nk**: llave),** *DecreaseKey* .– Cambia el valor del objeto x su llave a nk, sólo si nk es menor al valor de la llave anterior.

Intercambio(i, j **: integer),** *Swap(i, j).*– Supone que: $1 \leq i$; $2 \cdot i \leq i \leq j$ n, n número de elementos en H; el subárbol enraizado en la posición i, incluyendo la posición j es un heap, excepto (quizá) en la posición i.
Los elementos en la posiciones i y j se intercambian; el subárbol enraizado en la posición j es un heap, excepto (quizá) en la posición j.

ReHeap(i, j **: integer).**– Supone que: $1 \leq i \leq j \leq n$, n número de elementos en H; H es un árbol binario completo; el subárbol enraizado en la posición i (cuyo último elemento está en una posición menor o igual a j) es un heap, excepto (posiblemente) en la posición i.
Esta operación, ReHeap, reorganiza el heap: el subárbol enraizado en la posición i es transformado en un heap.

Ejemplo

Para ilustrar los Heaps binarios, consideremos construir un heap minimal H, incialmente vacío, sobre el cual aplicaremos la siguiente serie de operaciones:

1. Insertar los elementos $L_1 = \{28, 37, 55, 7, 72\}$;
2. Borrar al elemento mínimo;
3. Insertar los elementos $L_1 = \{81, 64, 5, 13, 27\}$;

Los primeros tres elementos son insertados directamente y conservan el orden del heap, Figura D.1(a). Al llegar el 7 hay que re-organizar el heap: 7 es menor que su padre y se realiza el Intercambio$(2, 4)$, ahora el 7 está en la posición 2, es menor que su padre y se hace el Intercambio$(1, 2)$, la Figura D.1(c), muestra este proceso y en (d) se presenta el resultado. Después, el 72 se agrega a la última posición y no genera cambios, la Figura D.1(e), ilustra el heap H al momento.

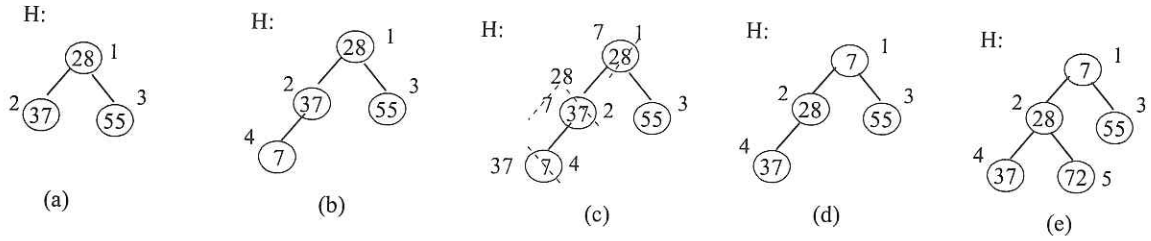


Figura D.1: Inserción de los elementos $L_1 = \{28, 37, 55, 7, 72\}$

Para borrar al mínimo elemento, hacemos un intercambio entre la raíz y el último elemento en H: Intercambio$(1, 5)$, la Figura D.2(a) ilustra esta acción; luego reorganizamos el heap, la Figura D.2(b), muestra este proceso y en (c) se presenta el heap H después de la operación.

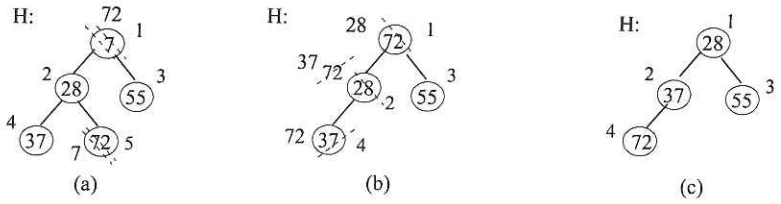


Figura D.2: Aplicación de la función BorraMin

Al insertar 81 y 64 no hay cambios, la Figura D.3(a), presenta el heap al momento; al insertar el 5, se reorganiza el Heap, la Figura D.3(b) muestra cómo queda el H al reorganizarse; después se inserta al 13 y hay que reorganizar el heap, el resultado se presenta en la Figura D.3(c). Al insertar el 27 se debe reorganizar el heap, la Figura D.3(d), presenta el heap final.

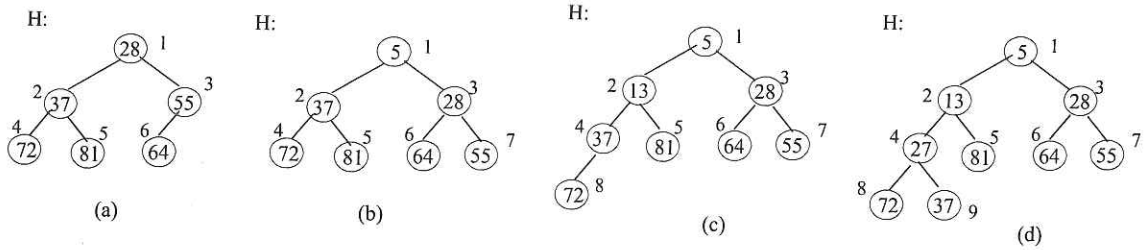


Figura D.3: Inserción de los elementos $L_1 = \{81, 64, 5, 13, 27\}$

Bibliografía

[1] S. Basee and A. Van Gelder. *Algoritmos Computacionales: Introducción al Analisis y diseño.* Pearson Education, 2002.

[2] F.M. Carrano and J.J. Prichard. *Data Abstraction and Problem Solving with Java. Walls and Mirrors.* Addison-Wesley publishing company, 2001.

[3] W. J. Collins. *Data Structures an Object Oriented Approach.* Addison-Wesley publishing company, 1992.

[4] T. H. Cormen, C. E. Leinserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* The MIT Press, McGraw Hill Book company, 2009.

[5] L. Denenberg and H. R. Lewis. *Data Structures and their Algorithms.* Harper Collins Publisher, 1991.

[6] B. Flaming. *Practical Algorithms in C++.* John Wiley and Sons inc, Coriolis Group Book, 1995.

[7] G.H. Gonnet and R. Baeza-Yates. *Handbook of Algorithms and Data Structures in Pascal and C.* Adison Wesley Publisher Company, 1991.

[8] D. E. Knuth. *The art of computer programming vol. 3.* Addison-Wesley Publishing company, 1973.

[9] T.G. Lewis and M.Z. Smith. *Applying Data Structures.* Houghton Miffling Company, 1976.

[10] U. Manber. *Introduction to Algoritms. A Creative Approach.* Addison-Wesley publishing company, 1989.

[11] J. J. McConell. *Analysis of Algorithms An Active Learning Approach.* Jones and Bartlett publishers, 2001.

[12] G. J.E. Rawlins. *Compared to what? an introduction to the analysis of algorithms.* Computer Science Press An Printing of W.H. Freeman and company, 1991.

Índice alfabético