Algoritmos sobre Búsqueda y Ordenamiento (Análisis y Diseño)

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

17 de marzo de 2020

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 $O(n^2)$.

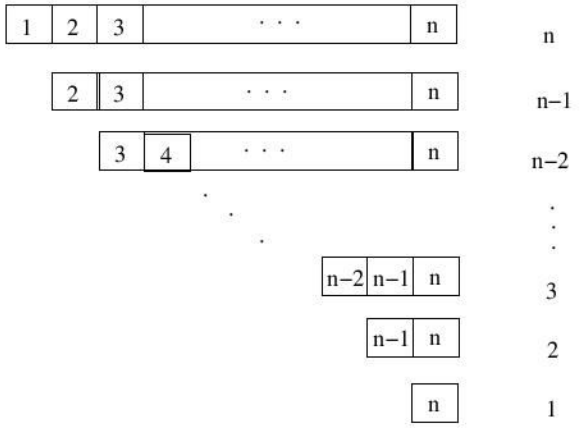


Figura 5.5: Ejemplo del peor caso para QuickSort

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.

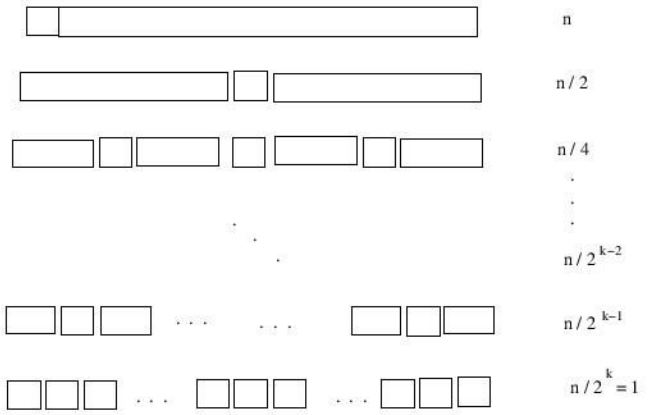


Figura 5.6: Ejemplo del mejor caso para QuickSort

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.

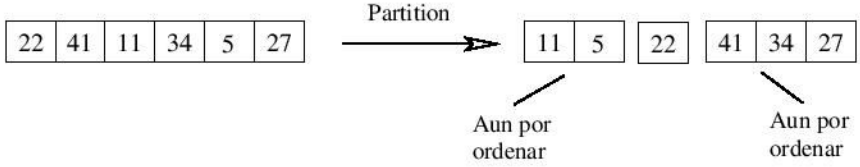
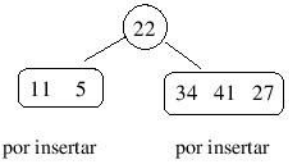


Figura 5.7: Ejemplo de la ejecución de la función Partition

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.

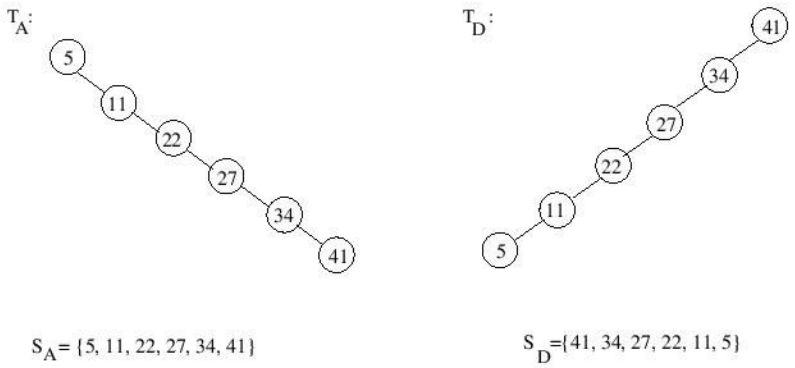
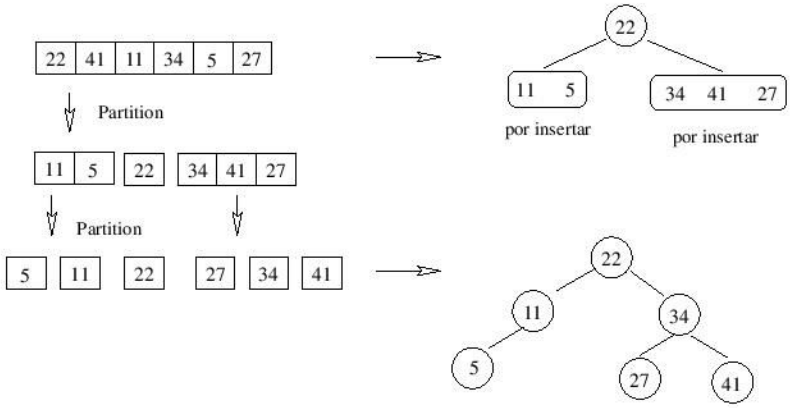


Figura 5.8: Peor caso: secuencia ordenada

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.

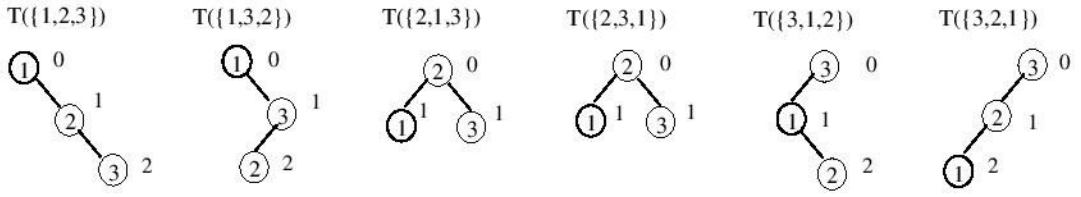


Figura 5.9: Construcción de los árboles $T(a)$ con la función $\beta(a)$

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.

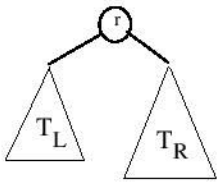


Figura 5.10: El árbol $T(a)$ y sus subárboles

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.

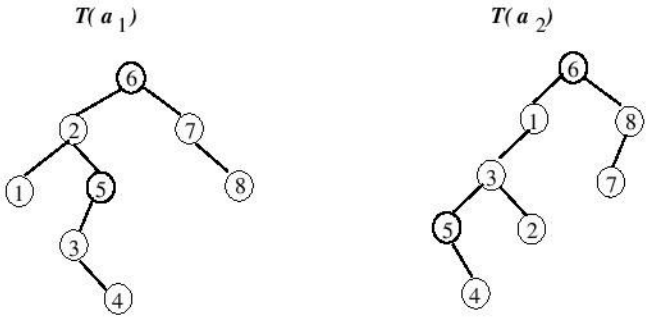


Figura 5.11: Construcción de dos árboles $T(a)$ con $a \in S_8^6$

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.

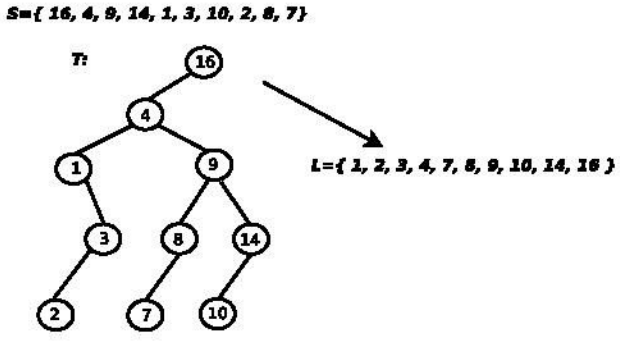


Figura 5.12: Ejemplo de Tree Sort

[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.

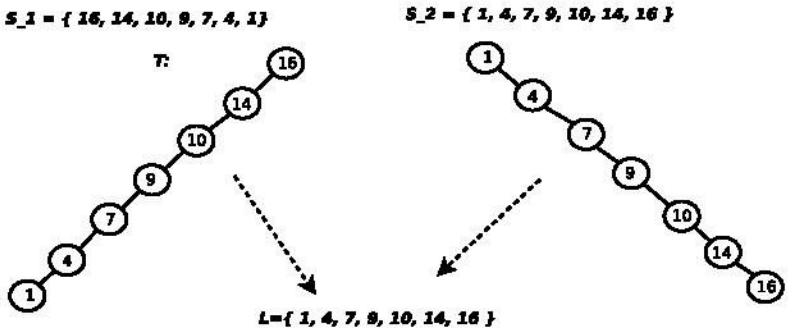


Figura 5.13: Peor caso de Tree Sort

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.

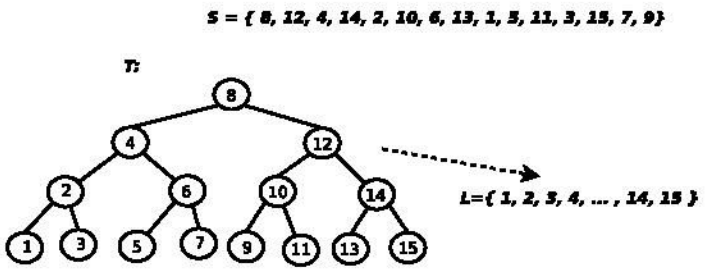


Figura 5.14: Mejor caso de Tree Sort

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: $S = \{8, 12, 4, 14, 2, 9, 13, 1, 3, 15, 10\}$.

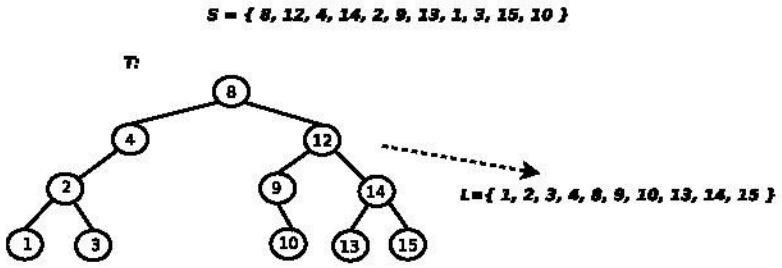


Figura 5.15: Caso Esperado de Tree Sort

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
```