# Algoritmos sobre Búsqueda y Ordenamiento (Análisis y Diseño)

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

17 de marzo de 2020

Luz Gasca S

# Capítulo 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.

Luz Gasca S

# 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

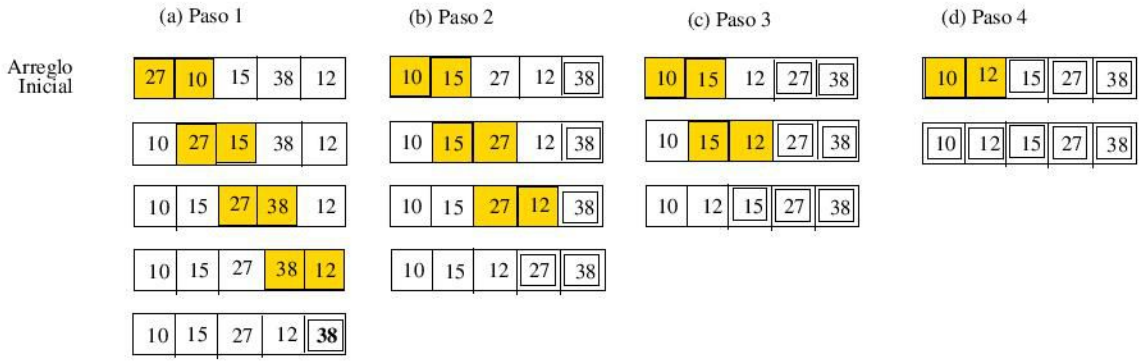


Figura 4.1: Ejemplo de Burbuja

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 $(n-1)+(n-2)+...+2+1 = n\cdot(n-1)/2$ 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 $i = 1, 2, \ldots, (n-1)$.

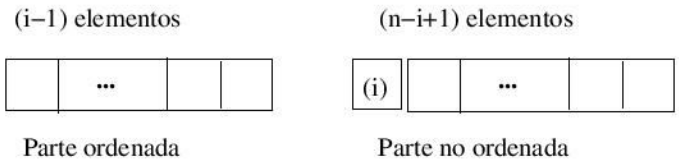


Figura 4.2: Estrategia general del *Selection Sort*

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.

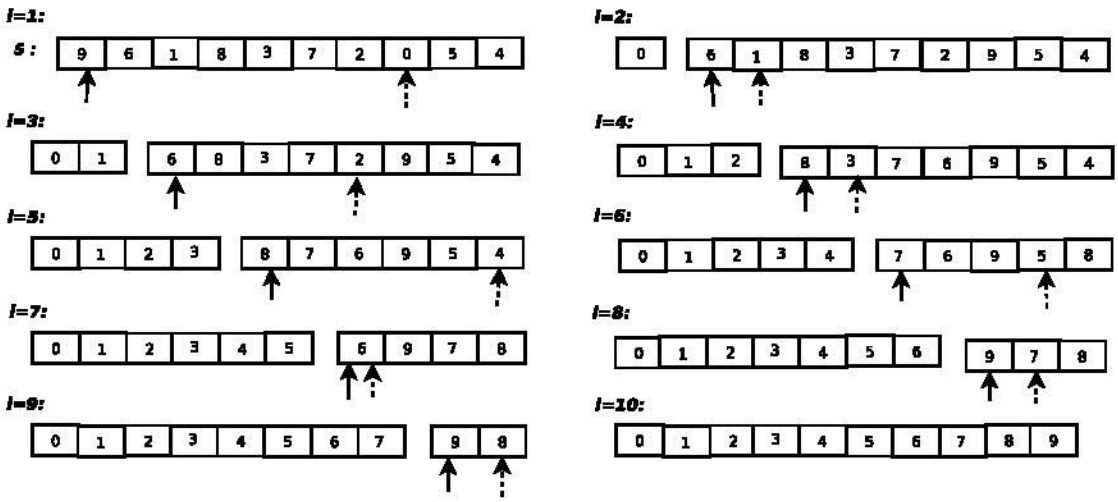


Figura 4.3: Ejemplo de *Selection Sort*

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:

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

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.

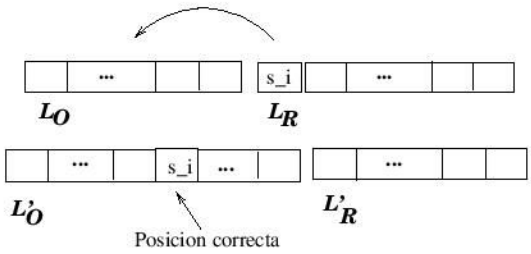


Figura 4.4: Estrategia General de *Insertion Sort*

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.

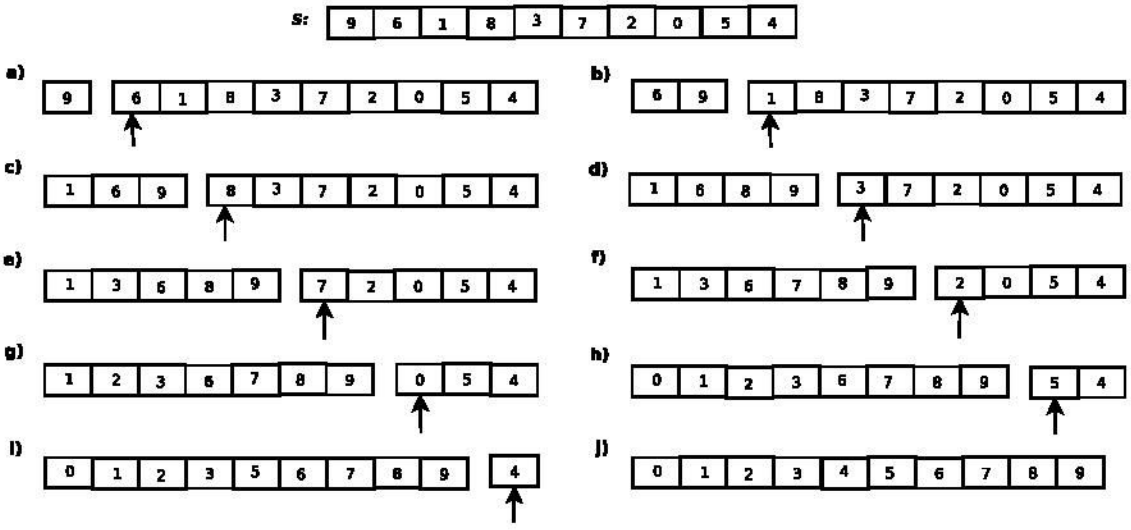


Figura 4.5: Ejemplo del Ordenamiento por Inserción

# 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:

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

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
```