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

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.

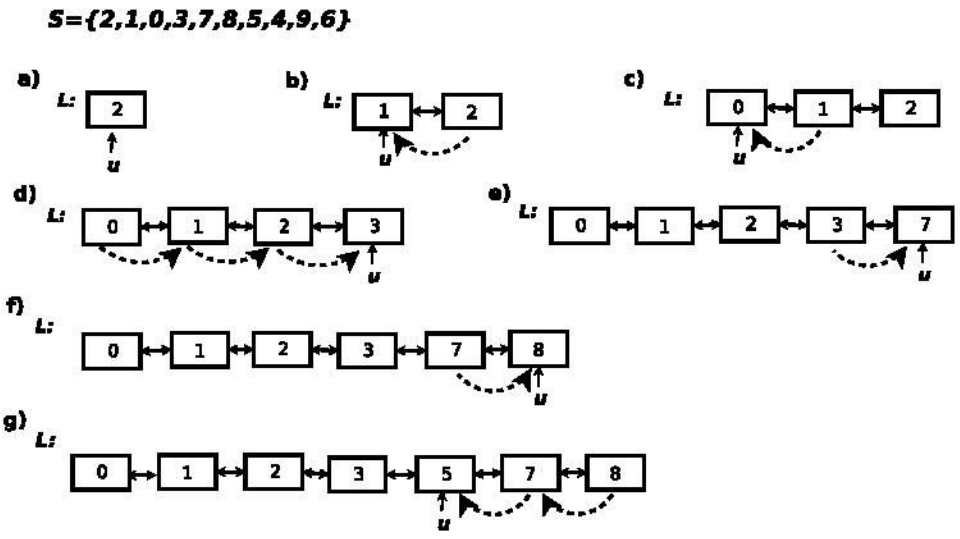

Figura 4.6: Ejemplo de Local Insertion Sort

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.

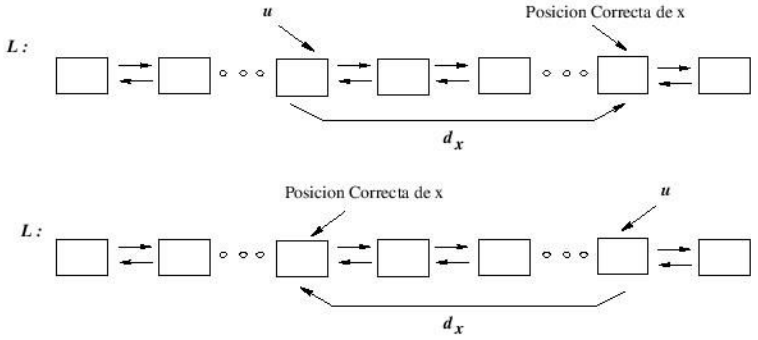

Figura 4.7: Desplazamiento d_x en Inserción local

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

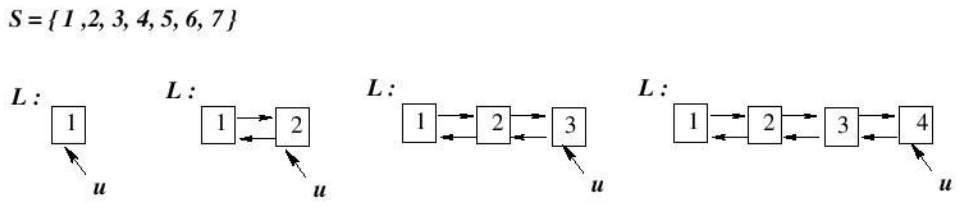

Figura 4.8: Mejor caso de Inserción local

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

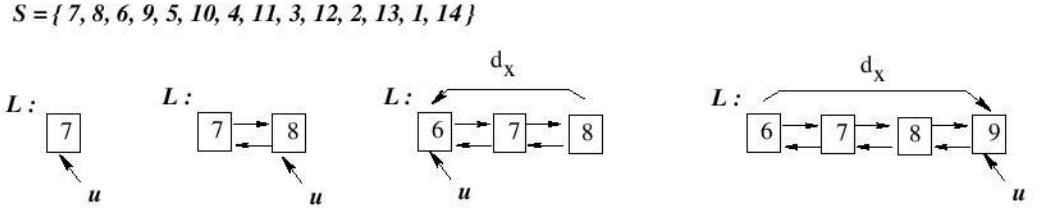

Figura 4.9: Peor caso de Inserción local

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.

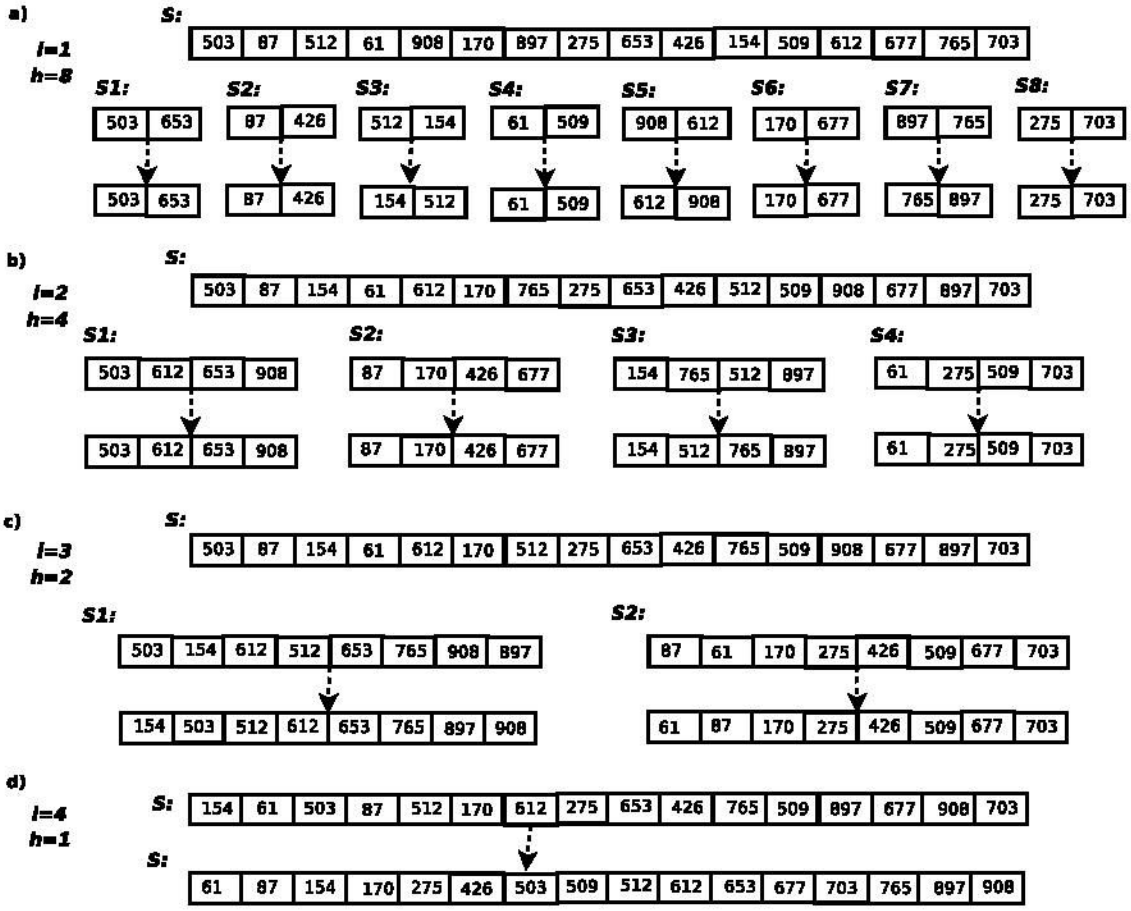

Figura 4.10: Ejemplo de Shell Sort

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:

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

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

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
```