```
        j = i;
        aux = a[i];
        /* se localiza el punto de inserción explorando hacia abajo */
        while (j > 0 && aux < a[j-1])
        {
            /* desplazar elementos hacia arriba para hacer espacio */
          a[j] = a[j-1];
          j--;
        }
        a[j] = aux;
      }
    }
```

El análisis del algoritmo de inserción se realizó como ejemplo en el Apartado 2.6.2, se determinó que la complejidad del algoritmo es $0(n^2)$, *complejidad cuadrática*.

6.6. ORDENACIÓN POR BURBUJA

El método de *ordenación por burbuja* es el más conocido y popular entre estudiantes y aprendices de programación, por su facilidad de comprensión y programación; por el contrario, es el menos eficiente y por ello, normalmente, se aprende su técnica pero no suele utilizarse.

La técnica utilizada se denomina ***ordenación por burbuja*** u ***ordenación por hundimiento*** debido a que los valores más pequeños «burbujean» gradualmente (suben) hacia la cima o parte superior del array de modo similar a como suben las burbujas en el agua, mientras que los valores mayores se hunden en la parte inferior del array. La técnica consiste en hacer varias pasadas a través del array. En cada pasada, se comparan parejas sucesivas de elementos. Si una pareja está en orden creciente (o los valores son idénticos), se dejan los valores como están. Si una pareja está en orden decreciente, sus valores se intercambian en el array.

6.6.1. Algoritmo de la burbuja

En el caso de un array (lista) con n elementos, la ordenación por burbuja requiere hasta $n - 1$ pasadas. Por cada pasada se comparan elementos adyacentes y se intercambian sus valores cuando el primer elemento es mayor que el segundo elemento. Al final de cada pasada, el elemento mayor ha «burbujeado» hasta la cima de la sublista actual. Por ejemplo, después que la pasada 0 está completa, la cola de la lista $A[n - 1]$ está ordenada y el frente de la lista permanece desordenado. Las etapas del algoritmo son:

- En la pasada 0 se comparan elementos adyacentes:

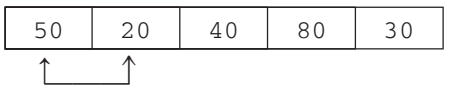
```
(A[0],A[1]),(A[1],A[2]),(A[2],A[3]),...(A[n-2],A[n-1])
```

  Se realizan $n - 1$ comparaciones, por cada pareja `(A[i],A[i+1])` se intercambian los valores si `A[i+1] < A[i]`. Al final de la pasada, el elemento mayor de la lista está situado en `A[n-1]`.
- En la pasada 1 se realizan las mismas comparaciones e intercambios, terminando con el elemento segundo mayor valor en `A[n-2]`.
- El proceso termina con la pasada $n - 1$, en la que el elemento más pequeño se almacena en `A[0]`.

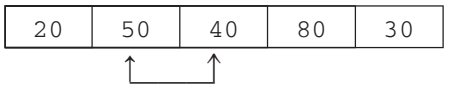
El algoritmo tiene una mejora inmediata, el proceso de ordenación puede terminar en la pasada $n - 1$, o bien antes, si en una pasada no se produce intercambio alguno entre elementos del vector es porque ya está ordenado, entonces no es necesario más pasadas.

El ejemplo siguiente ilustra el funcionamiento del algoritmo de la burbuja con un array de 5 elementos (`A = 50, 20, 40, 80, 30`), donde se introduce una variable `interruptor` para detectar si se ha producido intercambio en la pasada.

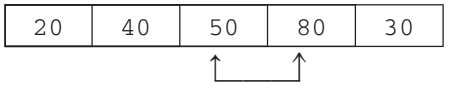
Pasada 0



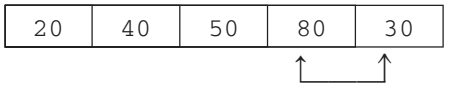
Intercambio `50` y `20`



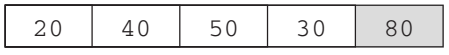
Intercambio `50` y `40`



`50` y `80` ordenados



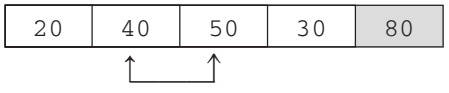
Intercambio `80` y `30`



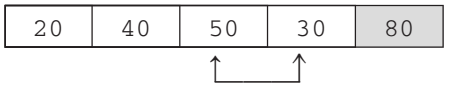
- Elemento mayor es `80`
- `interruptor = TRUE`

En la pasada 1:

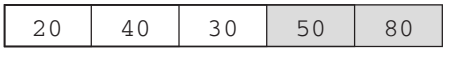
`20` y `40` ordenados



`40` y `50` ordenados

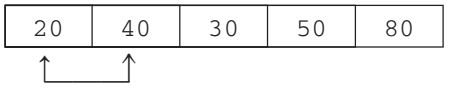


Se intercambian `50` y `30`

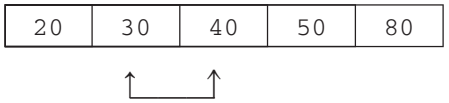


- `50` y `80` elementos mayores y ordenados
- `interruptor = TRUE`

En la pasada 2, sólo se hacen dos comparaciones:

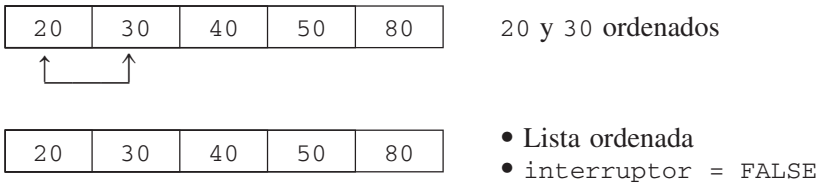


`20` y `40` ordenados



- Se intercambian `40` y `30`
- `interruptor = TRUE`

En la pasada 3, se hace una única comparación de `20` y `30`, y no se produce intercambio:



En consecuencia, el algoritmo de ordenación de burbuja mejorado contempla dos bucles anidados: el *bucle externo* controla la cantidad de pasadas (al principio de la primera pasada todavía no se ha producido ningún intercambio, por tanto la variable `interruptor` se pone a *valor falso* (0); el *bucle interno* controla cada pasada individualmente y cuando se produce un intercambio, cambia el valor de `interruptor` a *verdadero* (1).

El algoritmo terminará, bien cuando se termine la última pasada ($n - 1$) o bien cuando el valor del `interruptor` sea falso (0), es decir, no se haya hecho ningún intercambio. La condición para realizar una nueva pasada se define en la expresión lógica

```
(pasada < n-1) && interruptor
```

6.6.2. Codificación en C del algoritmo de la burbuja

La función `ordBurbuja()` implementa el algoritmo de ordenación de la burbuja; tiene dos argumentos, el array que se va a ordenar crecientemente, y el número de elementos `n`. En la codificación se supone que los elementos son de tipo entero largo.

```
void ordBurbuja (long a[], int n)
{
  int interruptor = 1;
  int pasada, j;

  for (pasada = 0; pasada < n-1 && interruptor; pasada++)
  {
      /* bucle externo controla la cantidad de pasadas */
    interruptor = 0;
    for (j = 0; j < n-pasada-1; j++)
      if (a[j] > a[j+1])
      {
            /* elementos desordenados, es necesario intercambio */
        long aux;
        interruptor = 1;
        aux = a[j];
        a[j] = a[j+1];
        a[j+1] = aux;
      }
  }
}
```

Una modificación al algoritmo anterior puede ser utilizar, en lugar de una variable bandera `interruptor`, una variable `indiceIntercambio` que se inicie a 0 (cero) al principio de cada

pasada y se establezca al índice del último intercambio, de modo que cuando al terminar la pasada el valor de `indiceIntercambio` siga siendo 0 implicará que no se ha producido ningún intercambio (o bien, que el intercambio ha sido con el primer elemento), y, por consiguiente, la lista estará ordenada. En caso de no ser 0, el valor de `indiceIntercambio` representa el índice del vector a partir del cual los elementos están ordenados. La codificación en C de esta alternativa es:

```
/*
  Ordenación por burbuja : array de n elementos
  Se realizan una serie de pasadas mientras indiceIntercambio > 0
*/

void ordBurbuja2 (long a[], int n)
{
  int i, j;
  int indiceIntercambio;

               /* i es el índice del último elemento de la sublista */
  i = n-1;

               /* el proceso continúa hasta que no haya intercambios */
  while (i > 0)
  {
    indiceIntercambio = 0;
               /* explorar la sublista a[0] a a[i] */
    for (j = 0; j < i; j++)
               /* intercambiar pareja y actualizar indiceIntercambio */
      if (a[j+1] < a[j])
      {
        long aux=a[j];
        a[j] = a[j+1];
        a[j+1] = aux;
        indiceIntercambio = j;
      }
               /* i se pone al valor del índice del último intercambio */
    i = indiceIntercambio;
  }
}
```

6.6.3. Análisis del algoritmo de la burbuja

¿Cuál es la eficiencia del algoritmo de ordenación de la burbuja? Dependerá de la versión utilizada. En la versión más simple se hacen $n - 1$ pasadas y $n - 1$ comparaciones en cada pasada. Por consiguiente, el número de comparaciones es $(n - 1) * (n - 1) = n^2 - 2n + 1$, es decir, la complejidad es $0(n^2)$.

Si se tienen en cuenta las versiones mejoradas haciendo uso de las variables `interruptor` o `indiceIntercambio`, entonces se tendrá una eficiencia diferente a cada algoritmo. En el mejor de los casos, la ordenación de burbuja hace una sola pasada en el caso de una lista que ya está ordenada en orden ascendente y por tanto su complejidad es $0(n)$. En el caso peor se requieren $(n - i - 1)$ comparaciones y $(n - i - 1)$ intercambios. La ordenación completa requiere $\frac{n(n-1)}{2}$ comparaciones

y un número similar de intercambios. La complejidad para el caso peor es $0(n^2)$ *comparaciones* y $0(n^2)$ *intercambios.* De cualquier forma, el análisis del caso general es complicado dado que alguna de las pasadas pueden no realizarse. Se podría señalar, entonces, que el número medio de pasadas k sea $0(n)$ y el número total de comparaciones es $0(n^2)$. En el mejor de los casos, la ordenación por burbuja puede terminar en menos de $n - 1$ pasadas pero requiere, normalmente, muchos más intercambios que la ordenación por selección y su prestación media es mucho más lenta, sobre todo cuando los arrays a ordenar son grandes.

6.7. ORDENACIÓN SHELL

La **ordenación Shell** debe el nombre a su inventor, D. L. Shell. Se suele denominar también *ordenación por inserción con incrementos decrecientes*. Se considera que el método Shell es una mejora de los métodos de inserción directa.

En el álgoritmo de inserción, cada elemento se compara con los elementos contiguos de su izquierda, uno tras otro. Si el elemento a insertar es el más pequeño hay que realizar muchas comparaciones antes de colocarlo en su lugar definitivo.

El algoritmo de Shell modifica los saltos contiguos resultantes de las comparaciones por saltos de mayor tamaño y con ello se consigue que la ordenación sea más rápida. Generalmente se toma como salto inicial $n/2$ (siendo n el número de elementos), luego se reduce el salto a la mitad en cada repetición hasta que el salto es de tamaño 1. El Ejemplo 6.1 ordena una lista de elementos siguiendo paso a paso el método de Shell.

Ejemplo 6.1

Obtener las secuencias parciales del vector al aplicar el método Shell para ordenar en orden creciente la lista:

```
6 1 5 2 3 4 0
```

El número de elementos que tiene la lista es 6, por lo que el salto inicial es 6/2 = 3. La siguiente tabla muestra el número de recorridos realizados en la lista con los saltos correspondiente.

Recorrido	*Salto*	*Intercambios*	*Lista*
1	3	(6,2),(5,4),(6,0)	2 1 4 0 3 5 6
2	3	(2, 0)	0 1 4 2 3 5 6
3	3	*ninguno*	0 1 4 2 3 5 6
salto 3/2 = 1			
4	1	(4,2),(4,3)	0 1 2 3 4 5 6
5	1	*ninguno*	0 1 2 3 4 5 6

6.7.1. Algoritmo de ordenación Shell

Los pasos a seguir por el algoritmo para una lista de `n` elementos son:

1. Dividir la lista original en `n/2` grupos de dos, considerando un incremento o salto entre los elementos de `n/2`.
2. Clarificar cada grupo por separado, comparando las parejas de elementos, y si no están ordenados, se intercambian.