Algoritmos de Ordenamiento

Abstract: Clasificación tradicional de métodos de ordenamiento. Análisis de funcionamiento de métodos de ordenamiento simples o directos (intercambio directo, selección directa e inserción directa). Anáisis de funcionamiento de métodos de ordenamiento compuestos o mejorados (quick sort, shell sort y heap sort).

1.) Métodos de ordenamiento. Clasificación tradicional.

Es claro que pueden existir muchos algoritmos diferentes para resolver el mismo problema, y el problema de la *ordenación de un arreglo* es quizás el caso emblemático de esa situación. Podemos afirmar que existen varias docenas de algoritmos diferentes para lograr que el contenido de un arreglo se modifique para dejarlo ordenado de menor a mayor, o de mayor a menor. Muchos de esos algoritmos están basados en ideas intuitivas muy simples, y otros se fundamentan en estrategias lógicas muy sutiles y no tan obvias a primera vista. En general, se suele llamar *métodos simples* o *métodos directos* a los del primer grupo, y *métodos compuestos* o *métodos mejorados* a los del segundo, aunque la diferencia real entre ambos no es sólo conceptual, sino que efectivamente existe una diferencia de rendimiento muy marcada en cuanto al tiempo que los métodos de cada grupo demoran en terminar de ordenar el arreglo. Más adelante nos introduciremos en la cuestión del *análisis comparativo* del rendimiento de algoritmos, y justificaremos formalmente la diferencia entre los dos grupos, pero por ahora nos concentraremos sólo en la clasificación de los algoritmos y el funcionamiento de cada uno de ellos.

Tradicionalmente, como dijimos, los métodos de ordenamiento se suelen clasificar en los dos grupos ya citados, y en cada grupo se encuentran los siguientes:

Métodos Simples o Directos	Métodos Compuestos o Mejorados
Intercambio Directo (Burbuja)	Método de Ordenamiento Rápido (*Quicksort*) [C. Hoare - 1960]
Selección Directa	Ordenamiento de Montículo (*Heapsort*) [J. Williams – 1964]
Inserción Directa	Ordenamiento por Incrementos Decrecientes (*Shellsort*) [D. Shell – 1959]

En principio, los métodos clasificados como Simples o Directos son algoritmos sencillos e intuitivos, pero de mal rendimiento si la cantidad *n* de elementos del arreglo es grande o muy grande, o incluso si lo que se desea es ordenar un archivo en disco. Aquí, *mal rendimiento* por ahora significa "demasiada demora esperada". En cambio, los métodos presentados como Compuestos o Mejorados tienen un muy buen rendimiento en comparación con los simples, y se aconsejan toda vez que el arrreglo sea muy grande o se quiera ordenar un conjunto en memoria externa. Si el tamaño *n* del arreglo es pequeño (no más de 100 o 150 elementos), los métodos simples siguen siendo una buena elección por cuanto su escasa eficiencia no es notable con conjuntos pequeños. Note que la idea final, es que cada algoritmo compuesto mostrado en esta tabla representa en realidad una mejora en el planteo del algoritmo simple de la misma fila, y por ello se los llama "métodos mejorados". Así, el algoritmo *Quicksort* es un replanteo del algoritmo de *intercambio directo* (más conocido como *ordenamiento de burbuja*) para eliminar los elementos que lo hace ineficiente. Paradójicamente, el *ordenamiento de burbuja* o *bubblesort* es en general el de *peor rendimiento* para ordenar un arreglo, pero ha dado lugar al *Quicksort*, que en general es el de *mejor rendimiento* conocido.

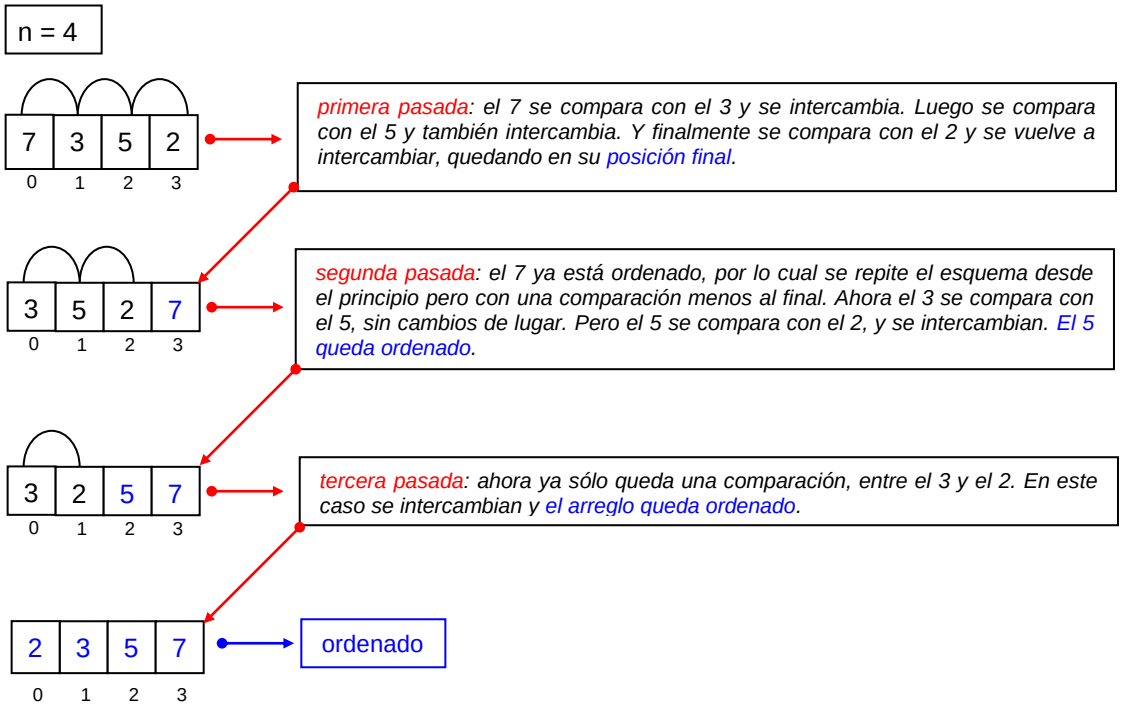
2.) Funcionamiento de los métodos de ordenamiento simples o directos.

Haremos aquí una breve descripción de las estrategias de funcionamiento de los tres métodos directos. Más adelante, mostraremos un análisis de rendimiento basado en calcular la cantidad de

comparaciones que estos métodos realizan. En todos los casos, suponemos un arreglo *v* de *n* elementos, y también suponemos que se pretende ordenar de menor a mayor. Hemos incluido un programa llamado *Ordenamiento*, que contiene la implementación de todos los métodos que aquí se dicutirán, sobre un arreglo de valores *int*.

---

a.) *Ordenamiento por Intercambio Directo (bubblesort):* la idea esencial del método de es que cada elemento en cada casilla *v[i]* se compara con el elemento en *v[i+1]*. Si este último es mayor, se intercambian los contenidos. Se usan dos ciclos anidados para conseguir que incluso los elementos pequeños ubicados muy atrás, puedan en algún momento llegar a sus posiciones al frente del arreglo. Gráficamente, supongamos que el tamaño del arreglo es *n = 4*. El método procede así:

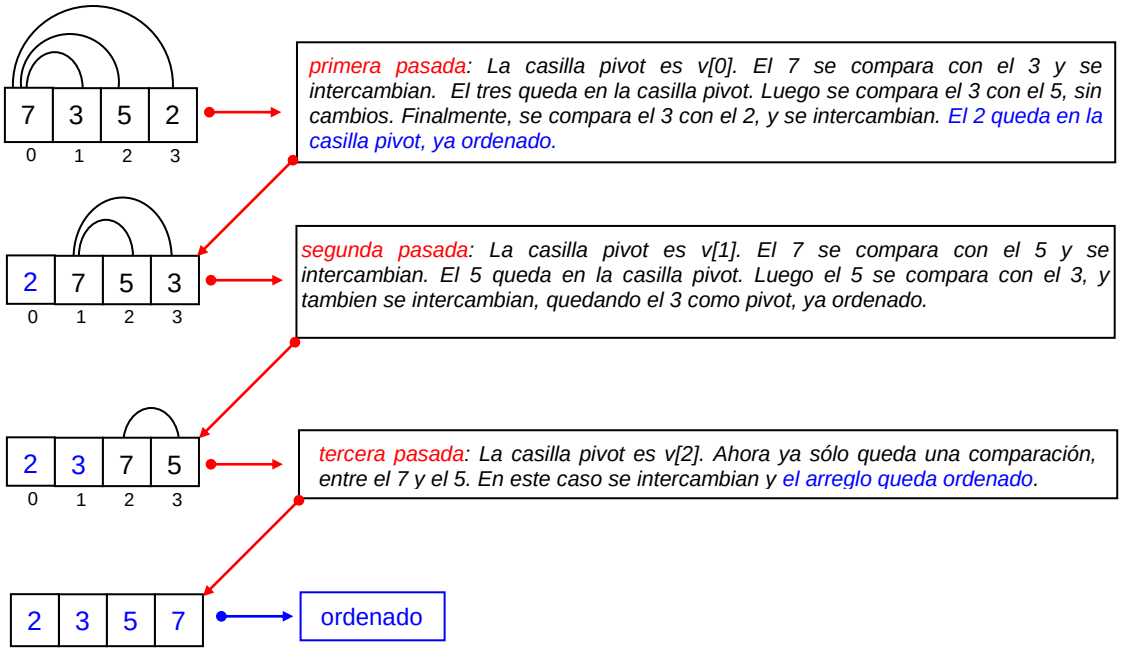


Puede verse que si el arreglo tiene *n* elementos, serán necesarias a lo sumo *n-1* pasadas para terminar de ordenarlo en el peor caso, y que en cada pasada se hace cada vez una comparación menos que en la anterior. Otro hecho notable es que el arreglo podría quedar ordenado antes de la última pasada. Por ejemplo, si el arreglo original hubiese sido [2 – 3 – 7 – 5], entonces en la primera pasada el 7 cambiaría con el 5 y dejaría al arreglo ordenado. Para detectar ese tipo de situaciones, el algoritmo conocido como *Burbuja Mejorado* agrega una variable a modo de bandera de corte: si se detecta que una pasada no hubo ningún intercambio, el ciclo que controla la cantidad de pasadas se interrumpe antes de llegar a la pasada *n-1* y el ordenamiento se da por concluido. Se ha denominado ordenamiento de burbuja porque metafóricamente los elementos parecen burbujear en el arreglo a medida que se ordena... (☹) La siguiente función (programa *Ordenamiento*) lo implementa:

```
void intercambio (int *v, long n)
{
  bool ordenado = false;
  for(long i=0; i < n-1 && !ordenado; i++)
```

```
{
   ordenado = true;
   for( long j=0; j < n-i-1; j++)
   {
            if (v[j] > v[j+1])
            {
              ordenado = false;
              int aux = v [j];
              v [j] = v [j+1];
              v [j+1] = aux;
            }
   }
 }
}
```

b.) *Ordenamiento por Selección Directa:* Igual que antes, se requieren *n-1* pasadas para terminar de ordenar el arreglo, pero ahora se toma la casilla *v[i]* y se compara su contenido contra todo el resto del vector. La idea es que cada vez que aparezca un valor menor que el contenido en *v[i]*, se reemplace *v[i]* por ese menor. La casilla *v[i]* se designa como *casilla pivot*. Cuando una pasada termina, la *casilla pivot* contiene al menor valor encontrado, que ya queda ordenado. En la pasada siguiente, se cambia la *casilla pivot* a la que sigue; y se procede igual, buscando el menor de los que quedan. Note que ahora no hay forma de usar una bandera de corte. El método procede así:



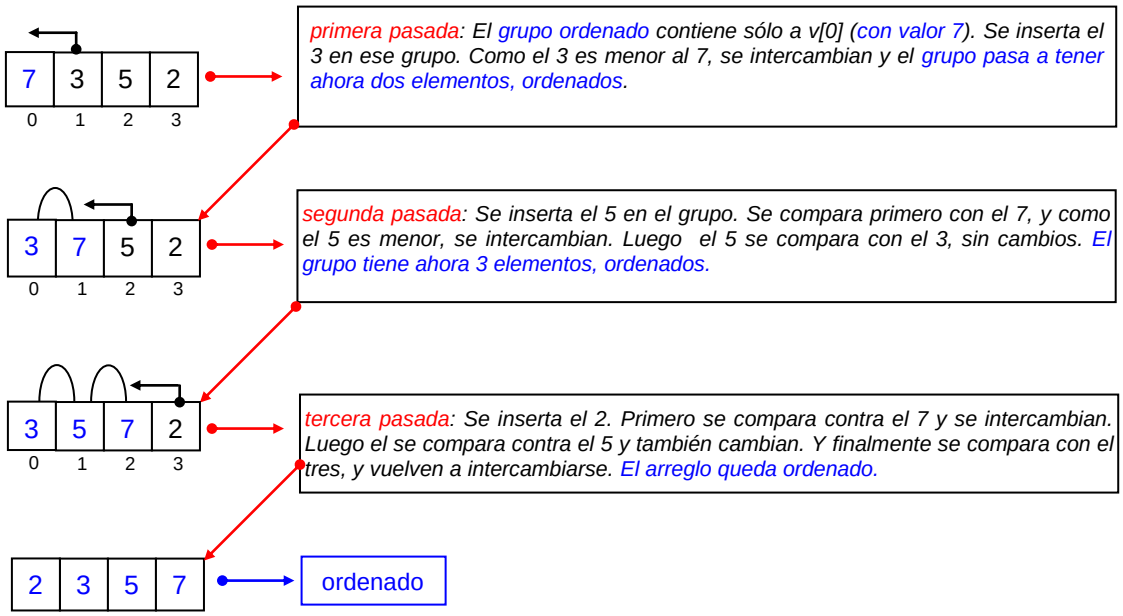
Y el siguiente método lo implementa:

```
void seleccion (int *v, long n)
{
  for (long i = 0; i < n - 1; i++)
  {
```

```
        for (long j = i + 1; j < n; j++)
        {
                if (v[i] > v[j])
                {
                  int aux = v [j];
                  v [j] = v [i];
                  v [i] = aux;
                }
        }
    }
}
```

---

c.) *Ordenamiento por Inserción Directa:* La idea es ahora distinta y más original. Se comienza suponiendo que el valor en la casilla *v[0]* conforma un subconjunto. Y como tiene un solo elemento, está ordenado. A ese subconjunto lo llamamos un *grupo ordenado* dentro del arreglo. Se toma *v[1]* y trata de insertar su valor en el grupo ordenado. Si es mayor a *v[0]* se intercambian, y si no, se dejan como estaban. En ambos casos, el grupo tiene ahora dos elementos y sigue ordenado. Se toma *v[2]* y se procede igual, comenzando la comparación contra *v[1]* (que es el mayor del grupo). Así, también hacen falta *n-1* pasadas, de forma que en cada pasada se inserta un nuevo valor al grupo. El método procede así:



Y el siguiente método lo implementa:

```
void insercion (int *v, long n)
{
  for (long j = 1; j < n; j++)
  {
       int y = v[j];
       long k;
       for (k = j-1; k >= 0 && y < v[k]; k--)
       {
```

```
                v[k+1]= v[k];
            }
            v[k+1]= y;
        }
    }
```

3.) Funcionamiento de los métodos de ordenamiento compuestos o mejorados.

Los métodos de este grupo están basados en la idea de mejorar algunos aspectos que hacen que los métodos directos no tengan buen rendimiento cuando el tamaño del arreglo es grande o muy grande. De nuevo, suponemos un arreglo v de n elementos, y ordenamiento de menor a mayor. En el programa *Ordenamiento* se implementan los métodos que mostrarán aquí.

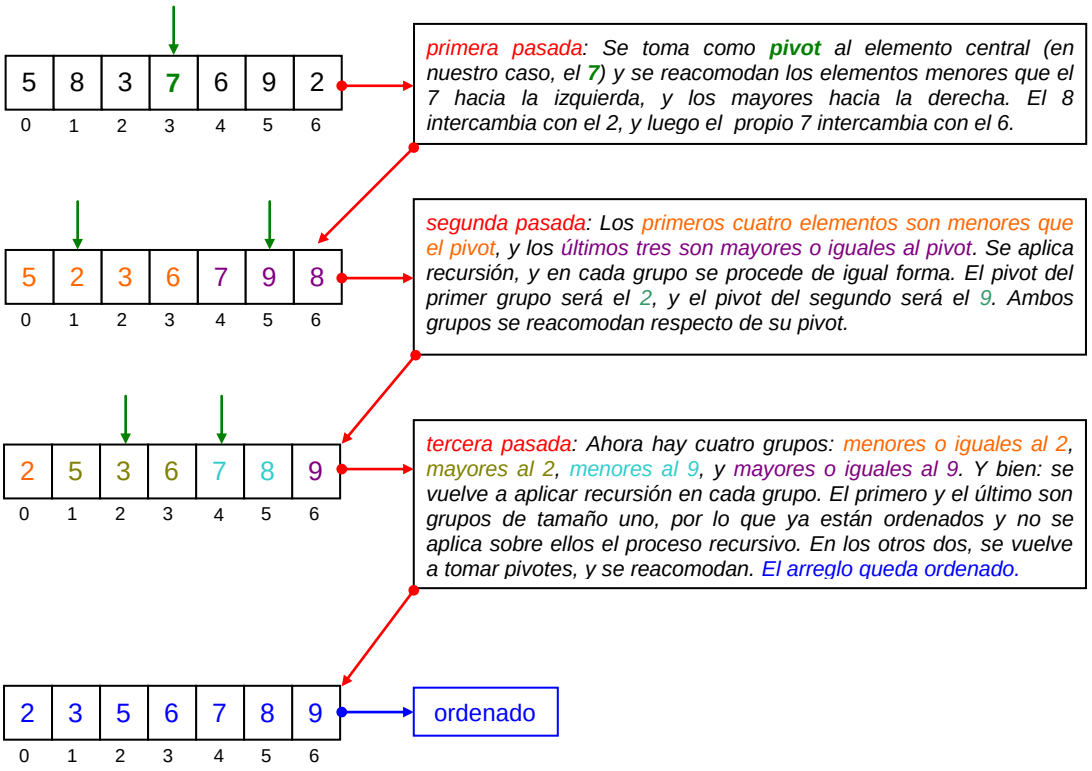
a.) *Ordenamiento Rápido: Quicksort.* Si se observa el método de *Intercambio Directo* o *Bubblesort*, se verá que algunos elementos (los mayores o "más pesados") tienden a viajar más rápidamente que otros (los menores o "más livianos") hacia su posición final en el arreglo. El proceso de "burbujeo" es evidentemente asimétrico, pues los elementos más grandes se comparan más veces que los más chicos en la misma pasada. También se llama *liebres* y *tortugas* a estos elementos... en obvia alusión a su velocidad de traslado por el vector. El resultado es que el método necesita muchas más pasadas hasta que se acomode el último de los menores que viaja desde la derecha...

En 1960, un estudiante de ciencias de la computación llamado *C. Hoare* se haría famoso al presentar en *Communicatons of the ACM* una versión mejorada del *bubble sort*, al cual llamó simplemente *Ordenamiento Rápido* o *Quicksort*... y desde entonces ese método se ha convertido en el más estudiado de la historia de la programación, entre otras cosas por ser hasta ahora en general el método más rápido (aunque hay situaciones en que se comporta bastante mal, esas situaciones son raras y poco probables)

La idea es recorrer el arreglo desde los dos extremos. Se toma un elemento *pivot* (que suele ser el valor ubicado al medio del arreglo pero puede ser cualquier otro). Luego, se recorre el vector desde la izquierda buscando algún valor que sea mayor que el pivot. Al encontrarlo, se comienza una búsqueda similar desde la derecha, pero ahora buscando un valor menor al pivot. Cuando ambos hayan sido encontrados, se intercambian entre ellos, y se sigue buscando otro par de valores en forma similar, hasta que ambas secuencias de búsquda se crucen entre ellas. De esta forma, se favorece que tanto los valores mayores ubicados muy a la izquierda como los menores ubicados muy a la derecha, viajen rápido hacia el otro extremo... y todos se vuelven liebres.

Al terminar esta pasada, se puede ver que el arreglo no queda necesariamente ordenado, pero queda claramente dividido en dos subarreglos: el de la izquierda contiene elementos que son todos menores o iguales al pivot, y el de la derecha contiene elementos mayores o iguales al pivot. Pero ahora se puede aplicar exactamente el mismo proceso a cada subarreglo, usando recursividad. El mismo método que particionó en dos el arreglo original, se invoca a sí mismo dos veces más, para partir en otros subarreglos a los dos que se obtuvieron recién. Con esto se generan cuatro subarreglos, y con más recursión se procede igual con ellos, hasta que

sólo se obtengan particiones de tamaño uno... Y en ese momento, el arreglo quedará ordenado. El método procede así:



Y el siguiente par de funciones lo implementan. Note que la segunda función es que la que realmente ordena el arreglo, aplicando recursión:

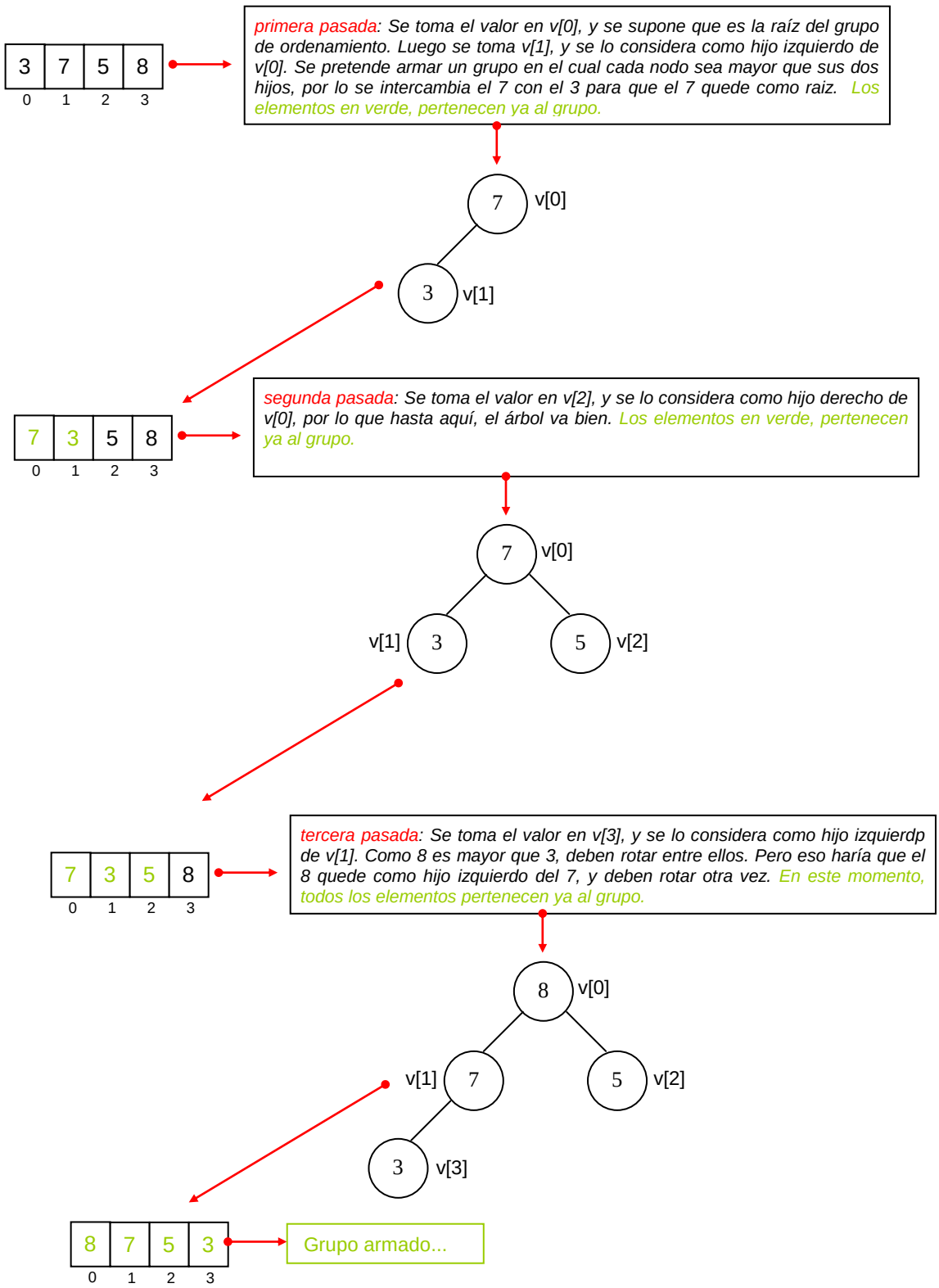
```
void quickSort (int *v, long n)
{
  ordenar (v, 0, n - 1);
}


void ordenar (int *v, long izq, long der)
{
  long i = izq, j = der;
  int  x = v[(izq + der) / 2];
  do
  {
          while (v[i] < x && i < der) i++;
          while (x < v[j] && j > izq) j--;
          if (i <= j)
          {
            int y = v[i];
            v[i] = v[j];
```

```
                v[j] = y;
                i++;
                j--;
              }
    }
    while (i <= j);
    if ( izq < j ) ordenar(v, izq, j);
    if ( i < der ) ordenar(v, i, der);
}
```

b.) *Ordenamiento de Montículo: Heapsort.* El método de ordenamiento por Selección Directa realiza n-1 pasadas, y el objetivo de cada una es seleccionar el menor de los elementos que sigan sin ordenar en el arreglo, y trasladarlo a la casilla pivot. El problema es que la búsqueda del menor en cada pasada se basa en un proceso secuencial, y exige demasiadas comparaciones.

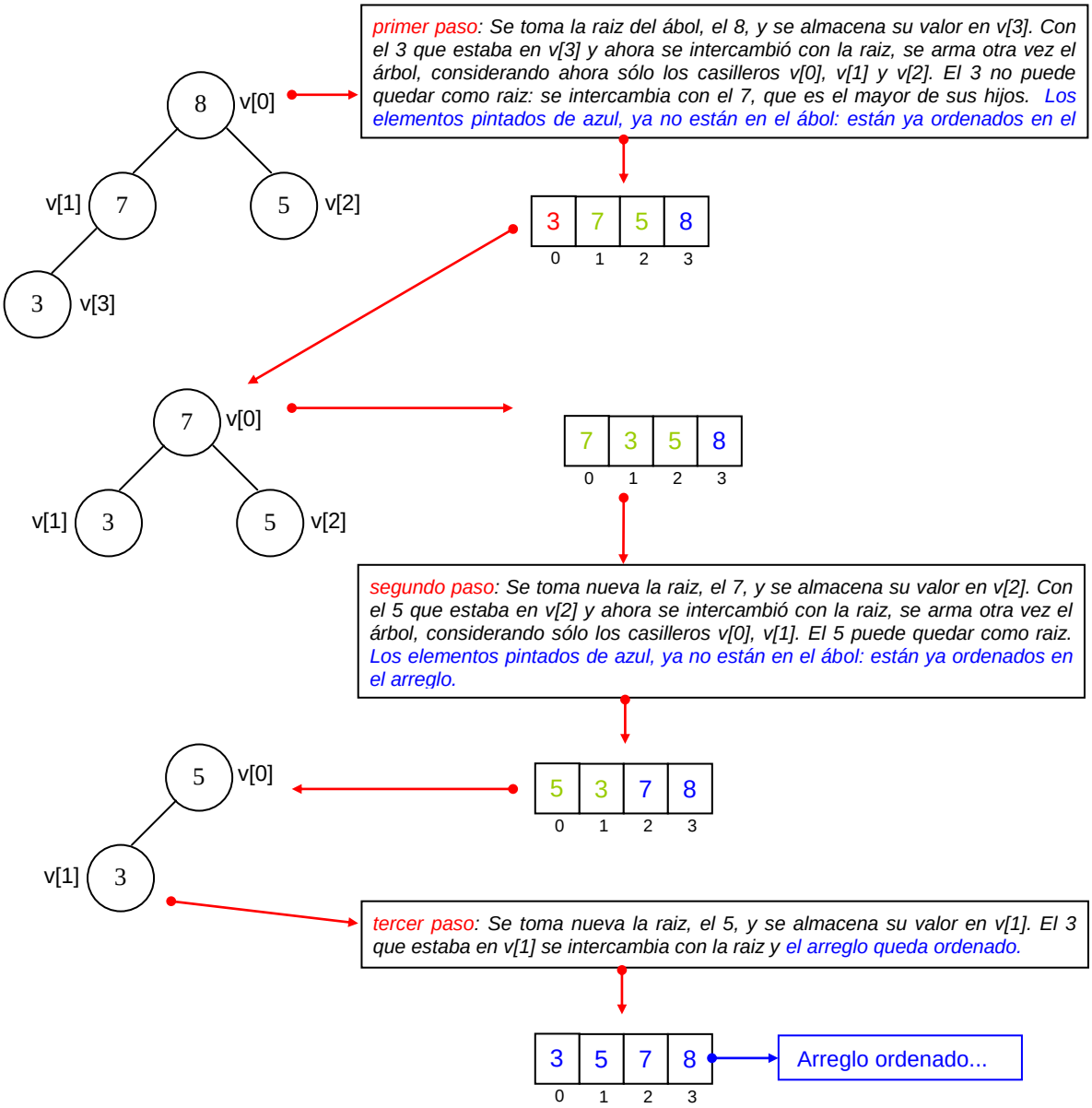
En 1964, otro estudiante de ciencias de la computación, llamado *J. Williams*, publicó un algoritmo mejorado en *Communicatons of de ACM*, y llamó al mismo *Ordenamiento de Montículos* o *Heapsort* (en realidad, debería traducirse como Ordenamiento de Grupos de Ordenamiento, pero queda redundante...) Ese algoritmo reduce de manera drástica el tiempo de búsqueda o selección del menor en cada pasada, usando una estructura de datos conocida como *grupo de ordenamiento* (que no es otra cosa que una *cola de prioridades*, pero optimizada en velocidad: los elementos se insertan rápidamente, ordenados de alguna forma, pero cuando se extrae alguno, se extrae el menor [o el mayor, según las necesidades]) Igual que antes, al seleccionar el menor (o el mayor) se lo lleva a su lugar final del arreglo. Luego se extrae el siguiente menor (o mayor), se lo reubica, y así se prosigue hasta terminar de ordenar.

En esencia, un *grupo de ordenamiento* es un *árbol binario casi completo* (todos los nodos hasta el anteúltimo nivel tienen dos hijos, a excepción de los nodos más a la derecha en ese nivel que podrían no tener los dos hijos) en el cual el valor de cada nodo es mayor que el valor de sus dos hijos. La idea es comenzar con todos los elementos del arreglo, y formar un árbol de este tipo, *pero dentro del mismo arreglo* (de otro modo, se usaría demasiado espacio adicional para ordenar...). Es simple implementar un árbol completo o casi completo en un arreglo: se ubica la raiz en el casillero *v[0]*, y luego se hace que para nodo en el casillero *v[i]*, su hijo izquierdo vaya en *v[2*i + 1]* y su hijo derecho *v[2*i + 2]*. Así, el algoritmo *Heapsort* primero lanza una fase de armado del grupo, en la cual los elementos del arreglo se reubican para simular el árbol. Gráficamente, el siguiente modelo simple muestra el proceso de armado del grupo para un pequeño arreglo de cuatro elementos:



Una vez armado el grupo, comienza la segunda fase del algoritmo: Se toma la raiz del árbol (que es el mayor del grupo), y se lleva al último casillero del arreglo (donde quedará ya ordenado). El valor que estaba en el último casillero se reinserta en el

árbol (que ahora tiene un elemento menos), y se repite este mecanismo, llevando el nuevo mayor al anteúltimo casillero. Así se continúa, hasta que el árbol quede con sólo un elemento, que ya estará ordenado en su posición correcta del arreglo. Gráficamente, la segunda fase sería así:



Y el siguiente método lo implementa:

```
void heapSort(int *v, long n)
{
  long i, s, f;
  int e,  valori;

  // crear el grupo inicial...
  for (i = 1; i < n; i++)
  {
          e = v [i];
```

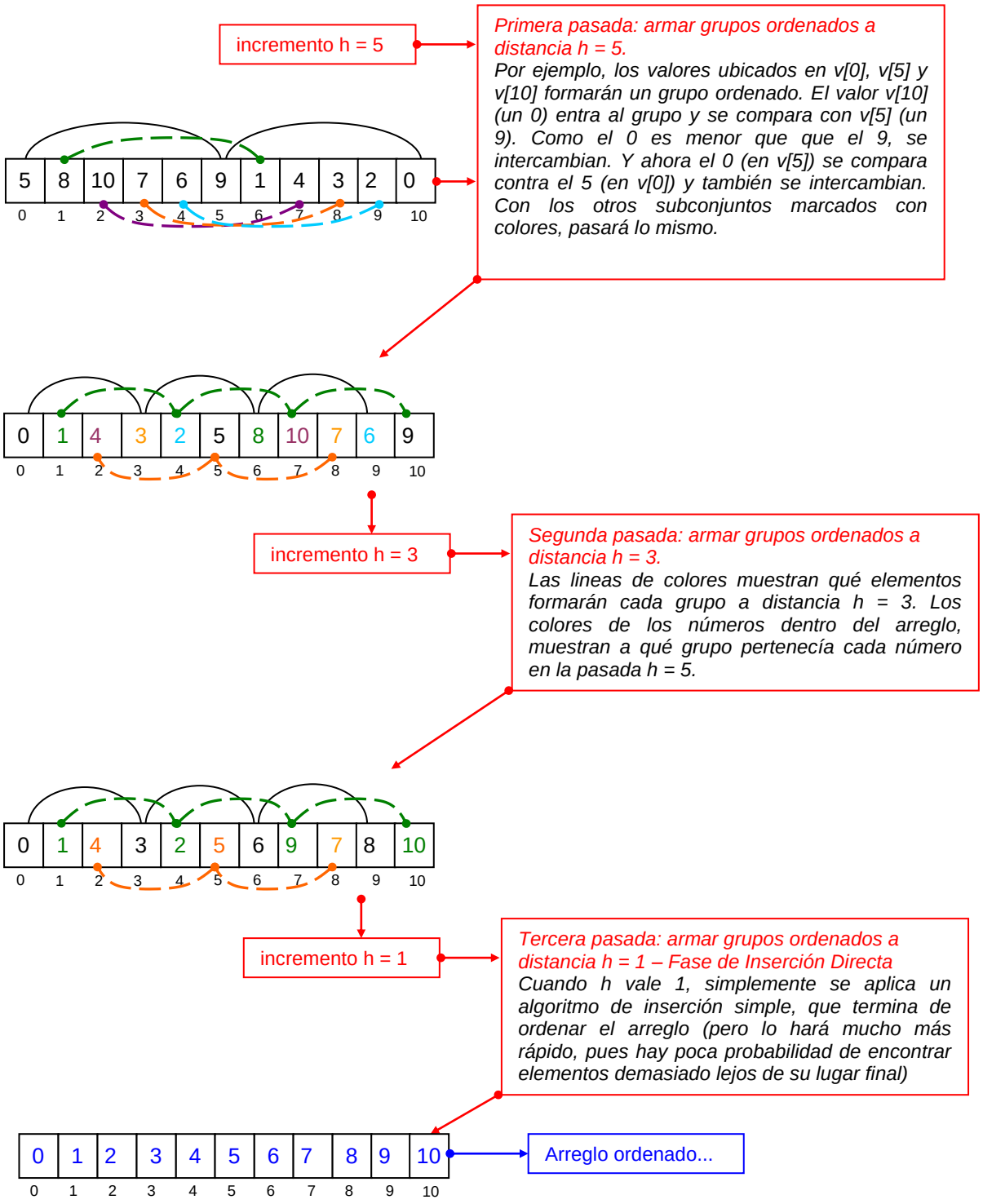
```
            s = i;
            f = (s-1)/2;
            while (s>0 && v[f] < e)
            {
              v[s] = v[f];
              s = f;
              f = (s-1)/2;
            }
            v[s] = e;
    }

    // se extrae la raiz, y se reordena el v y el grupo...
    for (i = n-1; i>0; i--)
    {
            valori = v[i];
            v[i] = v[0];
            f = 0;
            if (i == 1) s = -1; else s = 1;
            if (i > 2 && v[2] > v[1])  s = 2;
            while ( s >= 0 && valori < v[s])
            {
              v[f] = v[s];
              f = s;
              s = 2*f + 1;
              if (s + 1 <= i - 1 && v[s] < v[s+1]) s++;
              if (s > i - 1) s = -1;
            }
            v[f] = valori;
    }
}
```

c.) *Ordenamiento por Incrementos Decrecientes: Shellsort.* El método de ordenamiento por Inserción Directa es el más rápido de los métodos simples, pues aprovecha que un subconjunto del arreglo está ya ordenado y simplemente inserta nuevos valores en ese conjunto de forma que el grupo siga ordenado. El problema es que si llega a aparecer un elemento muy pequeño en el extremo derecho del arreglo, cuando el grupo ordenado de la izquierda ya contiene a casi todo el vector, la inserción de ese valor pequeño demorará demasiado, pues tendrá que compararse con casi todo el arreglo para llegar a su lugar final.

En 1959, un técnico de la *General Electric Company* llamado *Donald Shell*, publicó un algoritmo que mejoraba al algoritmo de inserción directa y lo llamó *High-Speed Sorting Procedure*, aunque en honor a su creador se lo terminó llamando *Shellsort*.

La idea es lanzar un proceso de ordenamiento por inserción, pero en lugar de hacer que cada valor que entra al grupo se compare con su vecino inmediato a la izquierda, se comience haciendo primero un reacomodamiento de forma que cada elemento del arreglo se compare contra elementos ubicados más lejos, a distancias mayores que uno, y se intercambien elementos a esas distancias. Luego, en pasadas sucesivas, las distancias de comparación se van acortando y repitiendo el proceso con elementos cada vez más cercanos unos a otros. De esta forma, se van armando grupos ordenados pero no a distancia uno, sino a distancia *h* tal que $h > 1$. Finalmente, se termina tomando una distancia de comparación igual a uno, y en ese momento el algoritmo se

convierte lisa y llanamente en una Inserción Directa para terminar de ordenar el arreglo. Las distancias de comparación se denominan en general *incrementos decrecientes*, y de allí el nombre con que también se conoce al método. Gráficamente, mostramos la idea con un pequeño arreglo, tomando incrementos de la forma [5 – 3 – 1]:



No es simple elegir los valores de los incrementos decrecientes, y de esa elección depende muy fuertemente el rendimiento del método. En general, digamos que no es necesario que sean demasiados: suele bastar con una cantidad de distancias igual o menor al 10% del tamaño del arreglo, pero debe asegurarse siempre que la última sea igual uno, pues de lo contrario no hay garantía que el arreglo quede ordenado. Por otra parte, es de esperar que los valores elegidos como distancias de comparación, no sean todos múliplos entre ellos, pues si así fuera se estarían comparando siempre las mismas subsecuencias de elementos, sin mezclar nunca esas subsecuencias. Sin embargo, no es necesario que los valores de los incrementos decrecientes sean todos necesariamente primos. Es suficiente con garantizar valores que no sean todos múltiplos entre sí.

Y el siguiente método lo implementa:

```
void shellSort(int *v, long n)
{
  long k, h;
  int y;
  for (h = 1; h <= n / 9; h = 3*h + 1);
  for ( ; h > 0; h /= 3)
  {
        for(long j = h; j < n; j++)
        {
              int y = v[j];
              for (k = j - h; k >= 0 && y < v[k]; k-=h)
              {
                    v[k+h] = v[k];
              }
              v[k+h] = y;
        }
  }
}
```