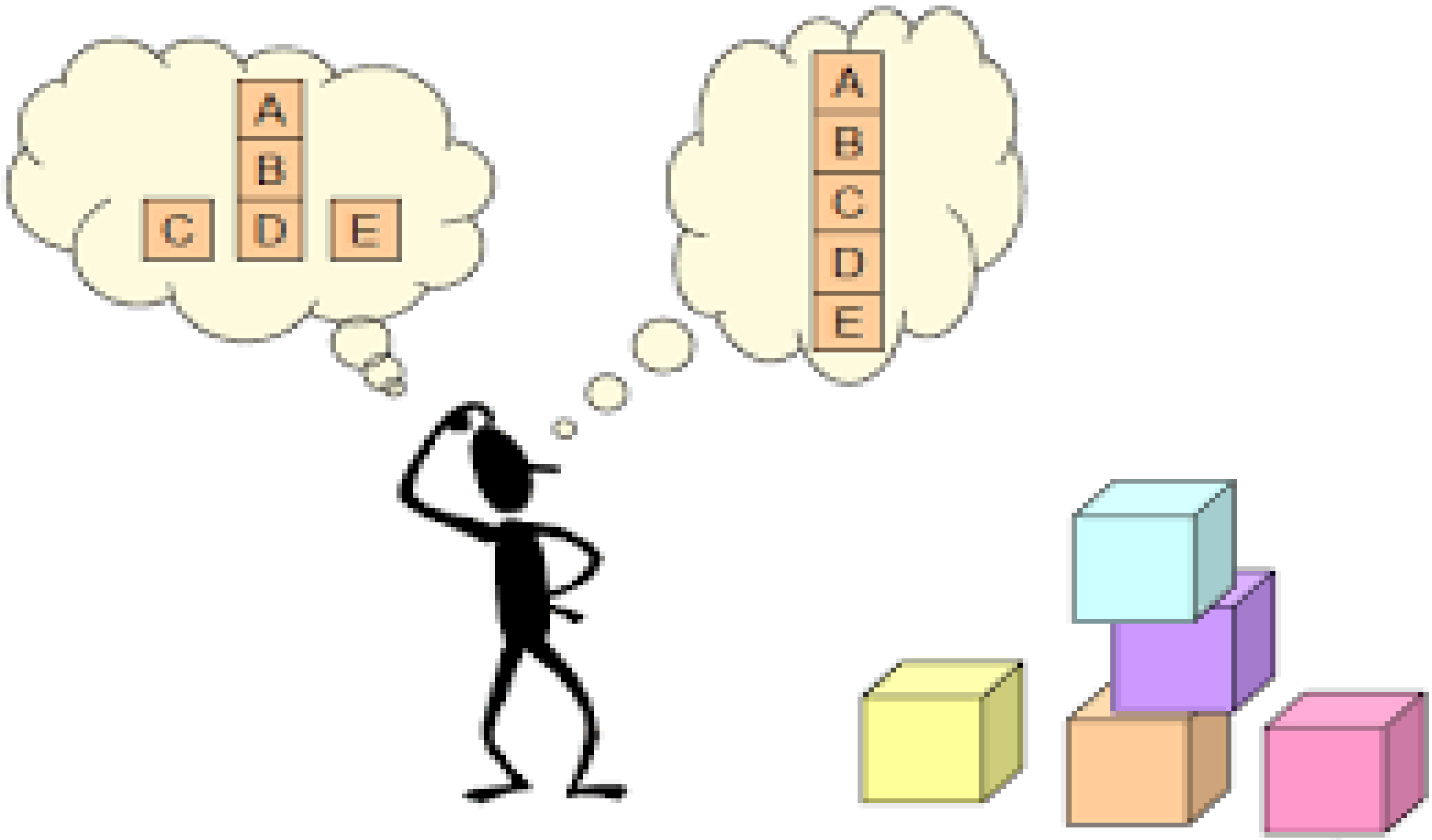
Unidad 3: Diseño y Análisis de Algoritmos

Tema VI: Ordenación y Búsqueda. Introducción a la ordenación. Clasificación de los algoritmos de ordenación. Métodos Directos. Métodos Logarítmicos. Intercalación. Ordenación por montículos. Problema de la Búsqueda Estática. Búsqueda Secuencial. Búsqueda Binaria. Búsqueda Interpolada. Cola de prioridades.

Mgter. Oscar Adolfo Vallejos
FaCENA - UNNE

PARTE SEGUNDA

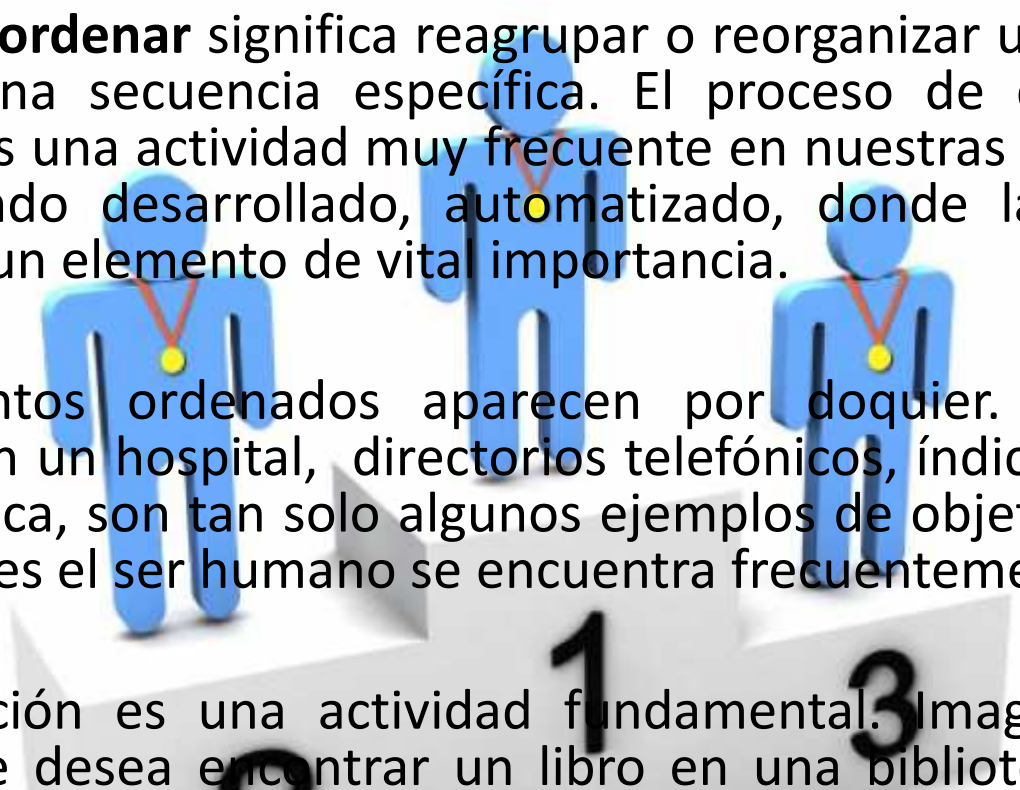


Clasificación

- **Clasificar u ordenar** significa reagrupar o reorganizar un conjunto de datos en una secuencia específica. El proceso de clasificación y búsqueda es una actividad muy frecuente en nuestras vidas. Vivimos en un mundo desarrollado, automatizado, donde la información representa un elemento de vital importancia.

- Los elementos ordenados aparecen por doquier. Registros de pacientes en un hospital, directorios telefónicos, índice de libros en una biblioteca, son tan solo algunos ejemplos de objetos ordenados con los cuales el ser humano se encuentra frecuentemente.

- La clasificación es una actividad fundamental. Imaginémonos un alumno que desea encontrar un libro en una biblioteca que tiene 100000 volúmenes y estos están desordenados o están registrados en los índices por orden de fecha de compra. También podemos pensar lo que ocurriría si deseamos encontrar el número de teléfono de una persona y la guía telefónica se encuentra ordenada por número.



Clasificación

Sea A una lista de N elementos:

$$A_1, A_2, A_3, ..., A_n$$

Clasificar significa permutar estos elementos de tal forma que los mismos queden de acuerdo con un orden preestablecido.

Ascendente: $A_1 <= A_2 <= A_3 <= A_4 .....<= A_n$

Descendente: $A_1 >= A_2 >= A_3 >= A_4 ..... >= A_n$

En el procesamiento de datos, a los métodos de ordenación se les clasifica en dos categorías:

- Categoría de arreglos
- Categoría de archivos

Clasificación

En cuanto a la cantidad de comparaciones que se realizan en un algoritmo se puede clasificar en:

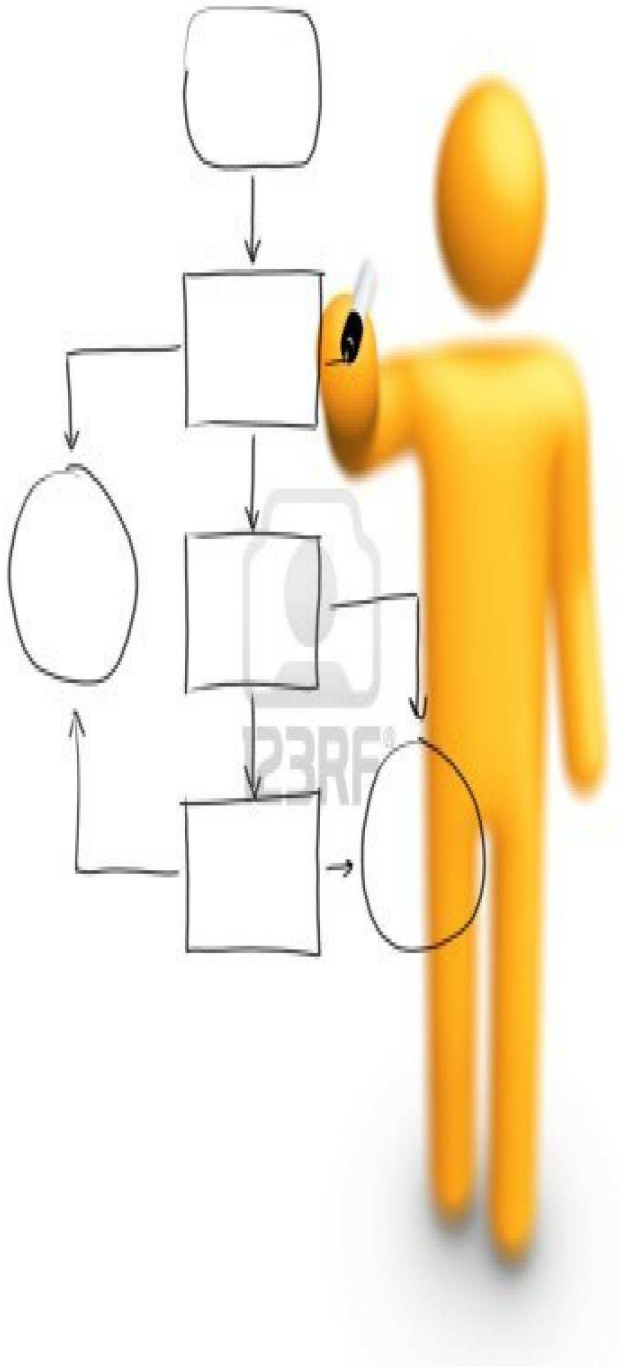
- Directos de orden O (N^2)
- Logarítmicos de orden O(N * log N)

Los **métodos directos** tienen la característica de que su resolución es mas corta, de fácil elaboración y comprensión, aunque son ineficientes cuando el número de elementos de un arreglo N, es mediano o considerablemente grande.

Los **métodos logarítmicos** son mas complejos con respecto a los directos, pero requieren menos comparaciones y movimientos para ordenar sus elementos, pero su elaboración y compresión resulta mas sofisticada y abstracta.

Se debe tener en cuenta que la eficiencia entre los distintos métodos se mide por el tiempo de ejecución del algoritmo y este depende fundamentalmente del número de comparaciones y movimientos que se realicen entre sus elementos.

Por lo tanto podemos decir que cuando N es pequeño debe utilizarse métodos directos y cuando N es mediana o grande deben emplearse métodos logarítmicos.



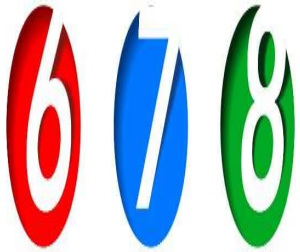
Clasificación

Métodos:

Directos:

- Ordenación por Intercambio Directo (Burbuja)
- Ordenación por Selección (Obtención sucesivas de menores)
- Ordenación por Inserción (Baraja)

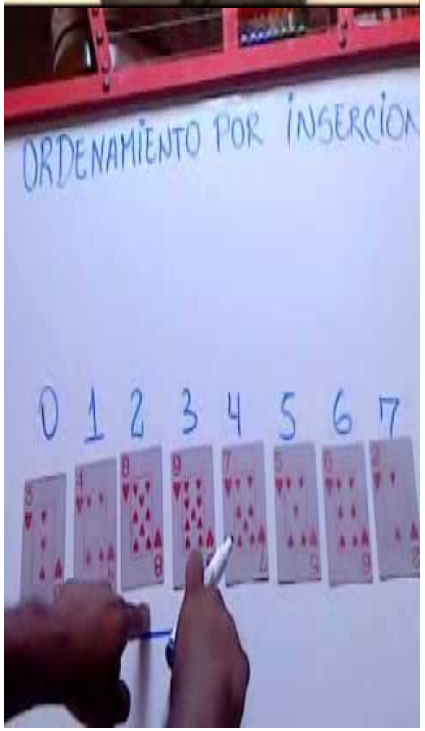
Logarítmicos:



- Método de Shell (Inserción con incrementos decrecientes)
- Método de QuickSort (Clasificación Rápida)
- Método del Montículo

Otros:

- Mezcla
- Método de raiz (radix)

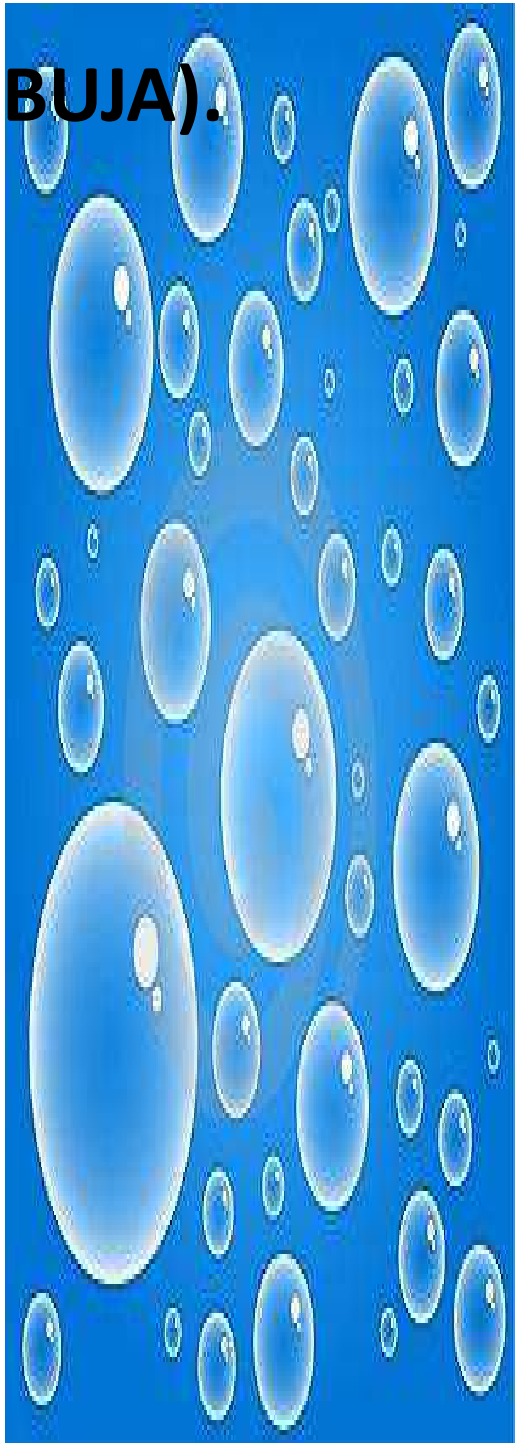


Ordenación por intercambio (BURBUJA).

Este método consiste en recorrer sucesivamente la lista o arreglo, comparando pares sucesivos de elementos adyacentes, e ir permutando los pares desordenados.

Se realizan (n-1) pasadas, transportando en cada pasada el menor o mayor elemento (según sea el caso) a su posición ideal. Al final de las (n-1) pasadas los elementos del arreglo estarán ordenados.

En cada pasada, el recorrido del vector se puede hacer de izquierda a derecha (desplazando los valores mayores hacia su derecha) o de derecha a izquierda (desplazando los valores menores hacia su izquierda), ambos para la clasificación en orden ascendente.



Ordenación por intercambio (BURBUJA) pasos a seguir …

1. • Comparar elemento (1) y elemento (2); si están ordenados, se deja como está; caso contrario se realiza el intercambio.

2. • Se comparan los dos elementos siguientes adyacentes elemento (2) y (3); y de nuevo se intercambia si es necesario.

3. • El proceso continúa hasta que cada elemento del arreglo haya sido comparado con sus elementos adyacentes y hayan sido intercambiados en los casos necesarios.

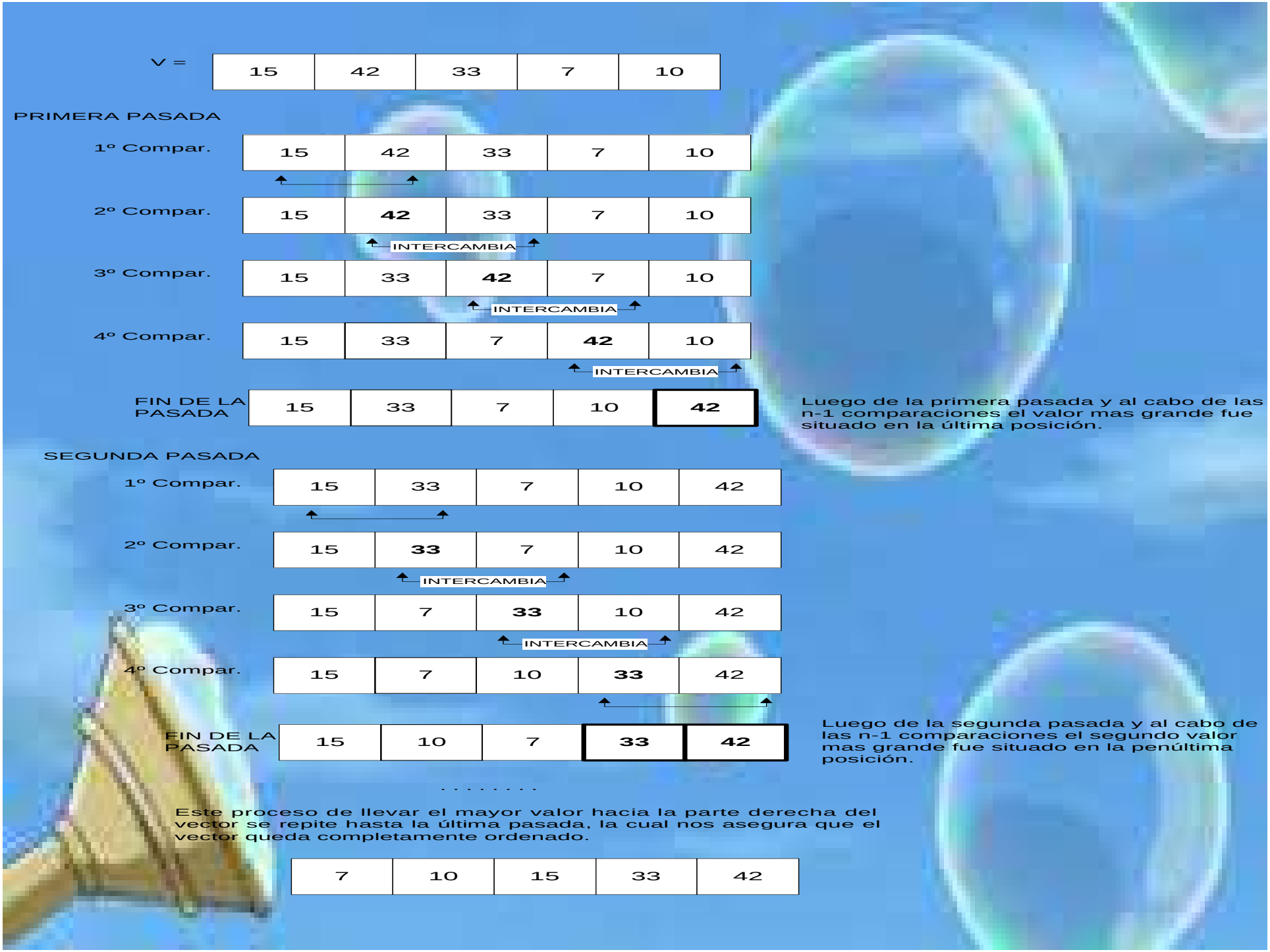
Ordenación por intercambio (BURBUJA).

La acción de intercambiar los elementos adyacentes requiere de una variable auxiliar. El proceso de esta triangulación será:

AUX = A(I)

A(I) = A(I+1)

A(I+1) = AUX


V =
15 42 33 7 10
PRIMERA PASADA
1º Compar.
15 42 33 7 10
2º Compar.
15 42 33 7 10
INTERCAMBIA
3º Compar.
15 33 42 7 10
INTERCAMBIA
4º Compar.
15 33 7 42 10
INTERCAMBIA
FIN DE LA PASADA
15 33 7 10 42
Luego de la primera pasada y al cabo de las n-1 comparaciones el valor mas grande fue situado en la última posición.
SEGUNDA PASADA
1º Compar.
15 33 7 10 42
2º Compar.
15 33 7 10 42
INTERCAMBIA
3º Compar.
15 7 33 10 42
INTERCAMBIA
4º Compar.
15 7 10 33 42
FIN DE LA PASADA
15 10 7 33 42
Luego de la segunda pasada y al cabo de las n-1 comparaciones el segundo valor mas grande fue situado en la penúltima posición.
. . . . . .
Este proceso de llevar el mayor valor hacia la parte derecha del vector se repite hasta la última pasada, la cual nos asegura que el vector queda completamente ordenado.
7 10 15 33 42

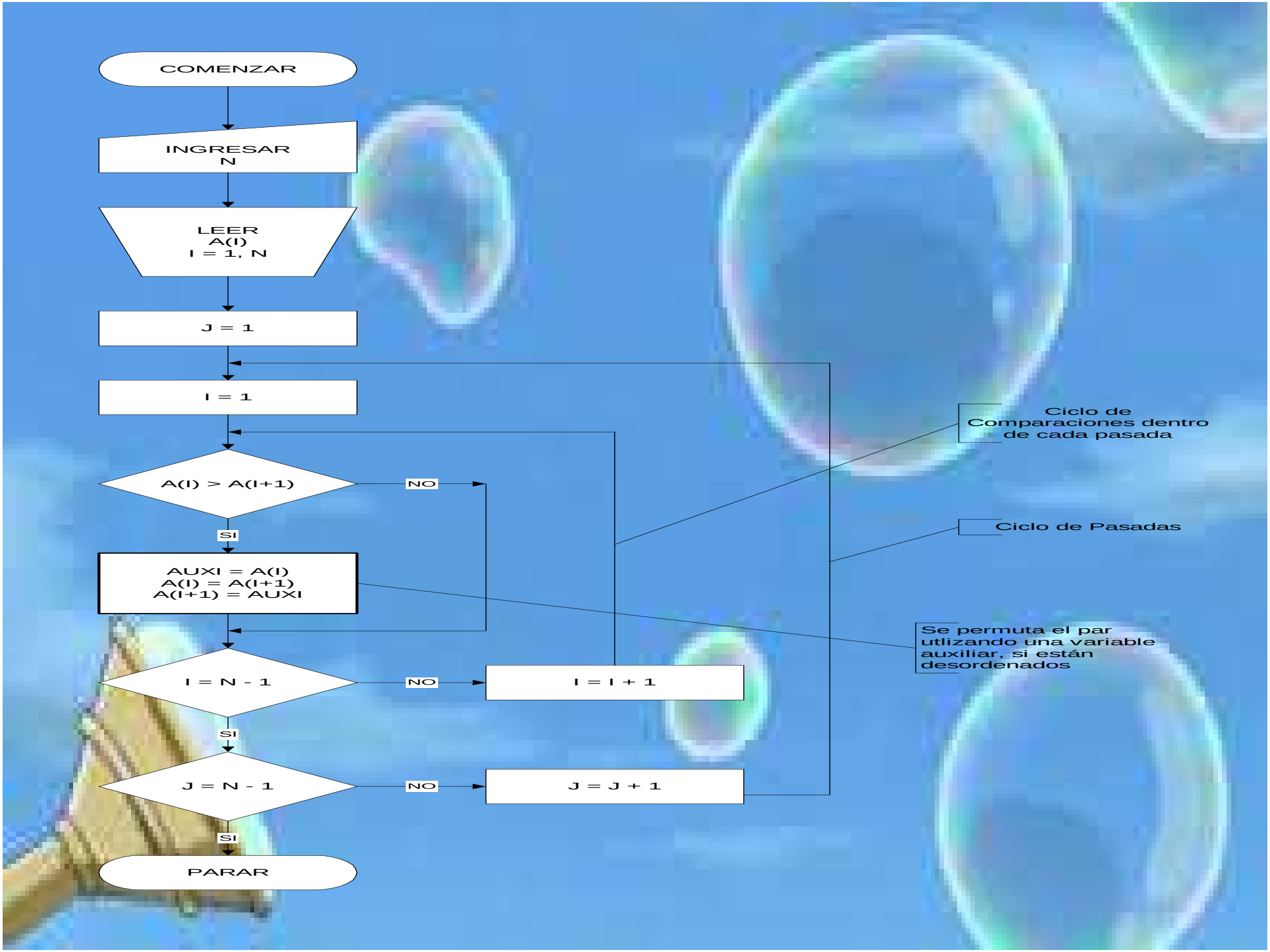
Ordenación por intercambio (BURBUJA)
Orden de complejidad …

Luego de este análisis y en forma genérica podemos observar que si se efectúan *n-1* pasadas y a su vez cada pasada requiere *n-1* comparaciones, la ordenación total de una tabla exigirá:

(n-1) * (n-1) = **$(n-1)^2$** comparaciones de elementos.

La cantidad de movimientos que se realicen en el arreglo dependerá del grado de desorden en que estén los datos.

Otro aspecto a considerar es el tiempo necesario para la ejecución del algoritmo, el mismo es proporcional a n^2.


COMENZAR
INGRESAR N
LEER A(I) I = 1, N
J = 1
I = 1
A(I) > A(I+1)
NO
SI
AUXI = A(I)
A(I) = A(I+1)
A(I+1) = AUXI
I = N - 1
NO
I = I + 1
SI
J = N - 1
NO
J = J + 1
SI
PARAR
Ciclo de Comparaciones dentro de cada pasada
Ciclo de Pasadas
Se permuta el par utlizando una variable auxiliar, si están desordenados

Ordenación por intercambio (BURBUJA)

```
Comenzar
Ingresar N
    Ingresar A(I) I = 1,N
    Desde J = 1 hasta N –1
        Desde I = 1 hasta N –1
            Si A(I) > A(I+1)
                Entonces
                    AUXI = A(I)
                    A(I) = A(I+1)
                    A(I+1) = AUXI
            Fin_si
        Fin_desde
    Fin_desde
Parar
```

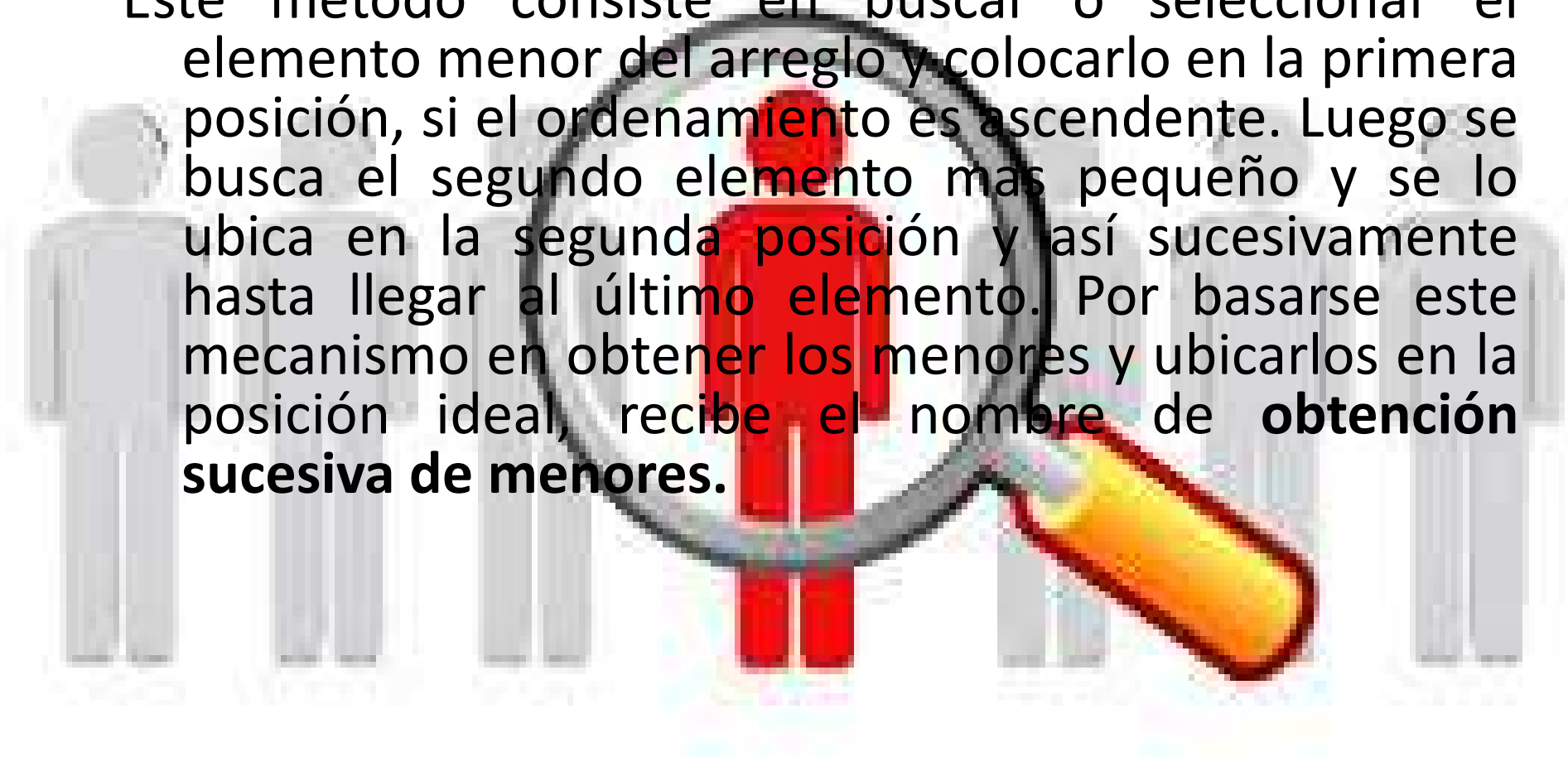
Método Burbuja

```
#include<stdio.h>
int main () {
	int aux,i,j,k;
	int n=10,A[n];
	for (i=0; i<n; i++) {
		printf("dame el dato %d ",i+1);
		scanf("%d",&A[i]);
		}
	for (i=1;i<n;i++) {
		for (j=0;j<n-i;j++)
		{
		if (A[j]>=A[j+1])
		{
				aux=A[j];
				A[j]=A[j+1];
				A[j+1]=aux;
		}
		}
		}
	for (i=0;i<n;i++) {
	printf(" %d", A[i]);
	}
return 0;
}
```

LENGUAJE

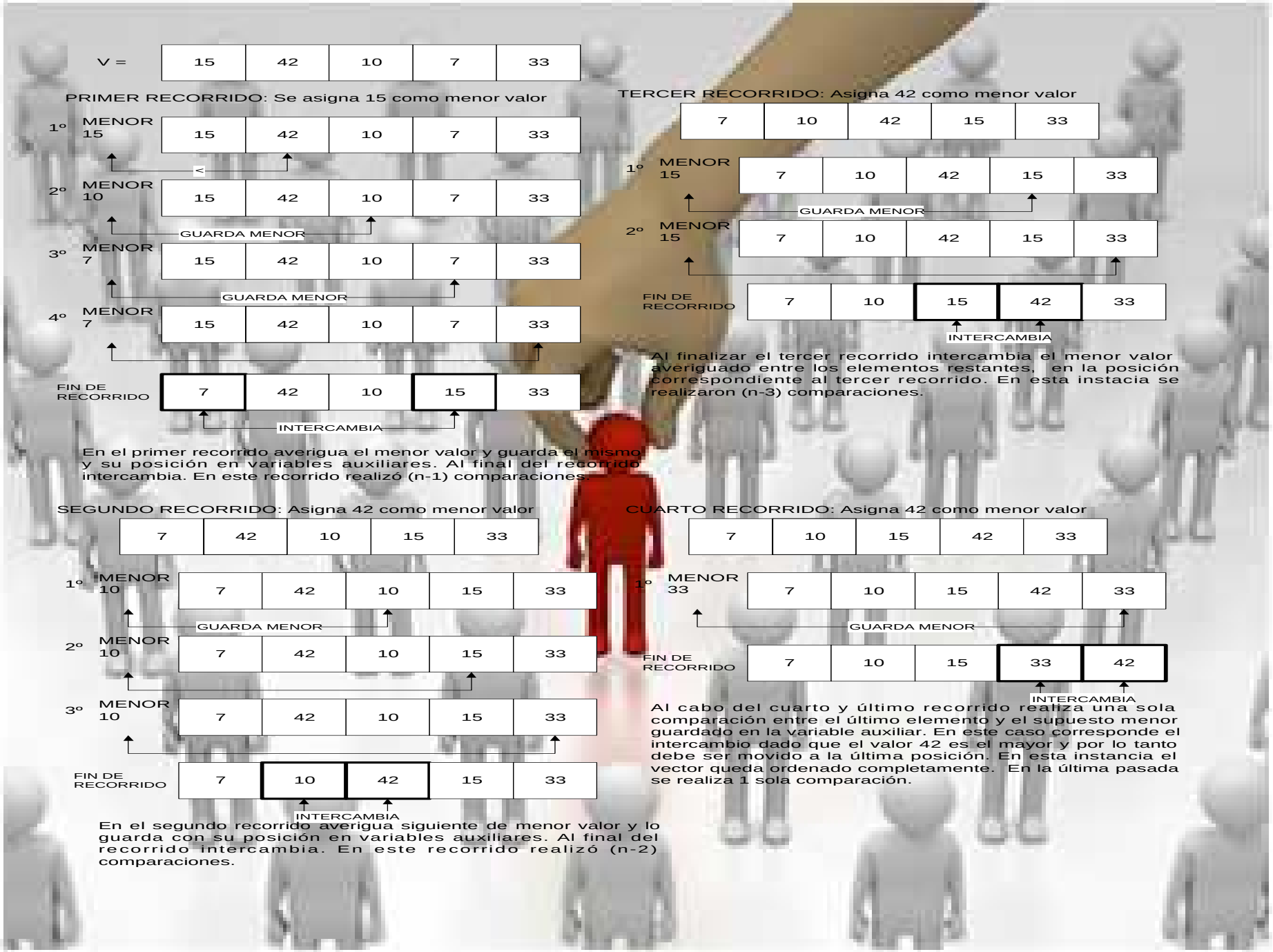
Ordenación por selección

Este método consiste en buscar o seleccionar el elemento menor del arreglo y colocarlo en la primera posición, si el ordenamiento es ascendente. Luego se busca el segundo elemento mas pequeño y se lo ubica en la segunda posición y así sucesivamente hasta llegar al último elemento. Por basarse este mecanismo en obtener los menores y ubicarlos en la posición ideal, recibe el nombre de **obtención sucesiva de menores.**



Ordenación por selección

1. Con este mecanismo, si pretendemos ordenar en forma creciente una tabla que posee 100 elementos, el método obliga a recorrer la tabla tantas veces como elementos tenga menos uno (n -1).
2. En el primer recorrido se averigua cual es el elemento menor y se intercambia con el que esté en la primera posición de la tabla.
3. En el segundo recorrido se averigua el menor entre los restantes elementos y se lo intercambia con el que está en la segunda posición
4. El resto de los recorridos utilizará la misma lógica.


V =
15 42 10 7 33
PRIMER RECORRIDO: Se asigna 15 como menor valor
1º MENOR 15
15 42 10 7 33
<
2º MENOR 10
15 42 10 7 33
GUARDA MENOR
3º MENOR 7
15 42 10 7 33
GUARDA MENOR
4º MENOR 7
15 42 10 7 33
FIN DE RECORRIDO
7 42 10 15 33
INTERCAMBIA
En el primer recorrido averigua el menor valor y guarda el mismo y su posición en variables auxiliares. Al final del recorrido intercambia. En este recorrido realizó (n-1) comparaciones.
SEGUNDO RECORRIDO: Asigna 42 como menor valor
7 42 10 15 33
1º MENOR 10
7 42 10 15 33
GUARDA MENOR
2º MENOR 10
7 42 10 15 33
3º MENOR 10
7 42 10 15 33
FIN DE RECORRIDO
7 10 42 15 33
INTERCAMBIA
En el segundo recorrido averigua siguiente de menor valor y lo guarda con su posición en variables auxiliares. Al final del recorrido intercambia. En este recorrido realizó (n-2) comparaciones.
TERCER RECORRIDO: Asigna 42 como menor valor
7 10 42 15 33
1º MENOR 15
7 10 42 15 33
GUARDA MENOR
2º MENOR 15
7 10 42 15 33
FIN DE RECORRIDO
7 10 15 42 33
INTERCAMBIA
Al finalizar el tercer recorrido intercambia el menor valor averiguado entre los elementos restantes, en la posición correspondiente al tercer recorrido. En esta instacia se realizaron (n-3) comparaciones.
CUARTO RECORRIDO: Asigna 42 como menor valor
7 10 15 42 33
1º MENOR 33
7 10 15 42 33
GUARDA MENOR
FIN DE RECORRIDO
7 10 15 33 42
INTERCAMBIA
Al cabo del cuarto y último recorrido realiza una sola comparación entre el último elemento y el supuesto menor guardado en la variable auxiliar. En este caso corresponde el intercambio dado que el valor 42 es el mayor y por lo tanto debe ser movido a la última posición. En esta instancia el vector queda ordenado completamente. En la última pasada se realiza 1 sola comparación.

Ordenación por selección

Si analizamos el desarrollo anterior podemos darnos cuenta que se realizan (n-1) recorridos, lo cual es un inconveniente del método dado que nos obliga a recorrer la lista un número fijo de veces, sin detectar si esta queda ordenada en alguno de los recorridos.

Al igual que en el método por intercambio, el número de comparaciones entre elementos es independiente de la disposición inicial de los mismos.

En el primer recorrido se realizan (n-1) comparaciones, en el segundo recorrido (n-2) comparaciones y así sucesivamente hasta llegar al último recorrido en la cual se realiza 1 comparación.

Por lo tanto la cantidad total de comparaciones la expresamos de la siguiente manera:

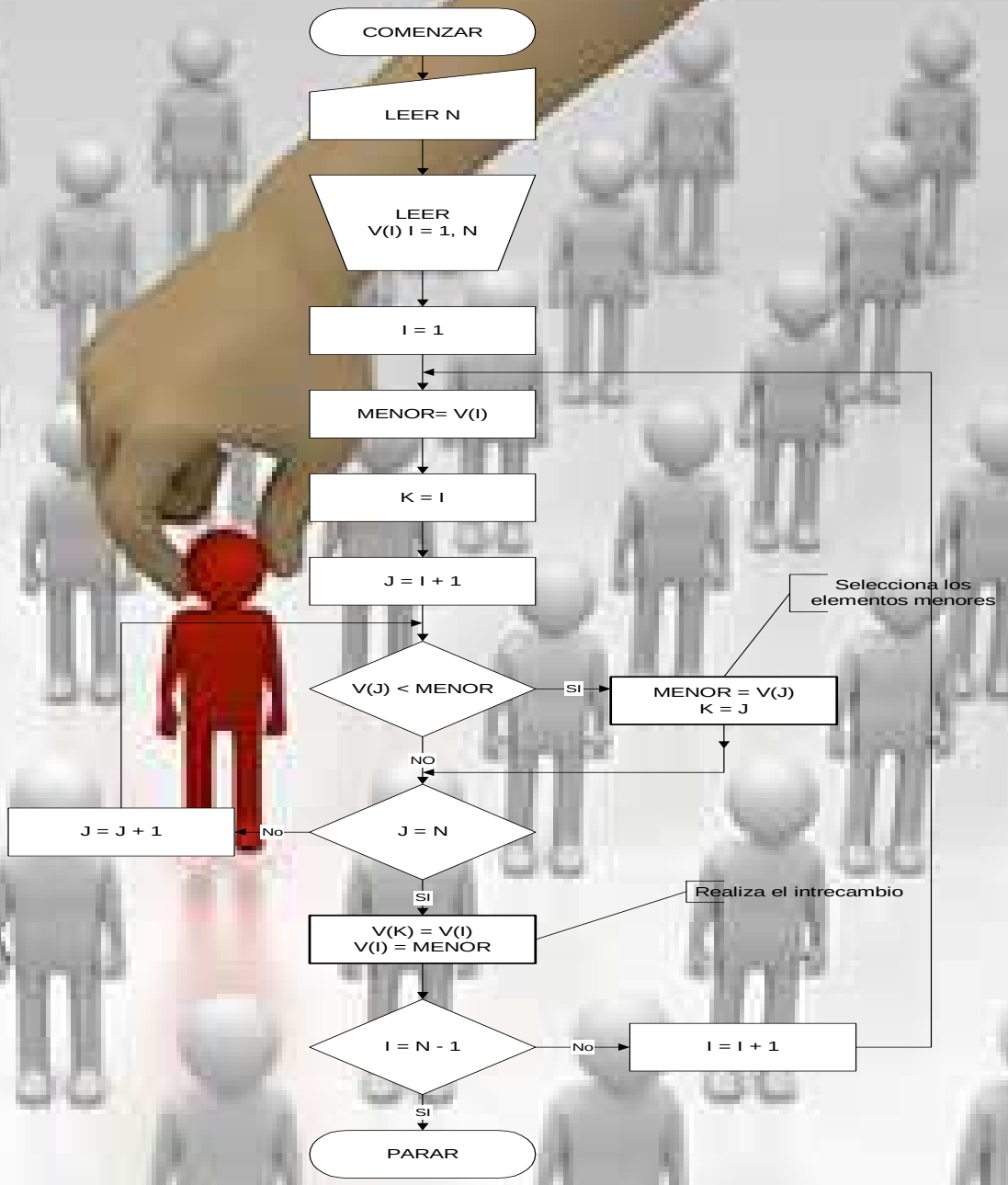
$C = (n-1) + (n-2) + ... + 2 + 1 = \mathbf{n * (n-1) / 2}$

Que es igual a:

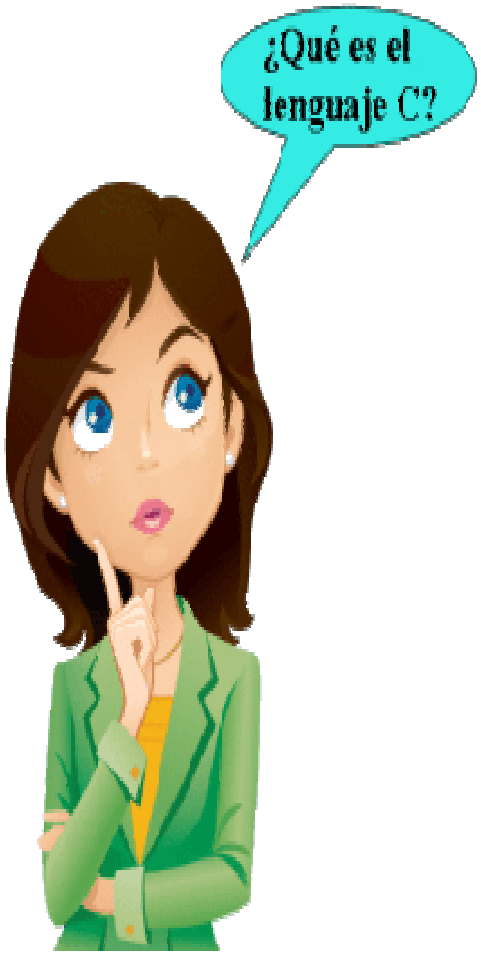
$$C = \mathbf{n^2 – n / 2}$$

En cuanto al número de movimientos será n-1 ya que el método, tal cual está desarrollado, realiza intercambio de un elemento consigo mismo.

Comenzar
Leer N
Ingresar V(I) I = 1,N
Desde I = 1 **hasta** N –1
MENOR = V(I)
K = I
Desde J = I + 1 **hasta** J = N
Si V(J) < MENOR
Entonces
MENOR = V(J)
K = J
Fin_si
Fin_desde
V(K) = V(I)
V(I) = MENOR
Fin_desde
Parar



Método Selección

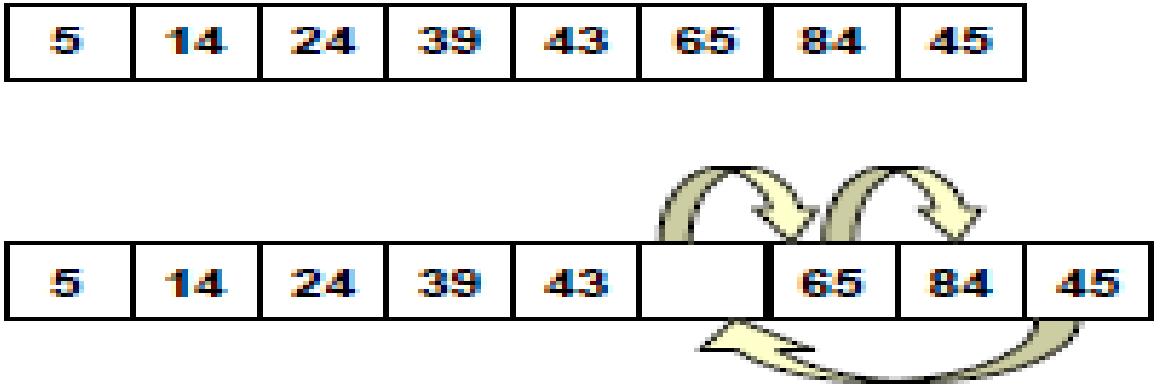



```
#include <stdio.h>
int main() {
        int array [100], n, i , d, pos, swap;
        printf("cuantos elementos deseas ordenar?\n");
        scanf("%d", &n);

    printf("Introduce los %d numeros \n", n);
        for (i=0; i < n; i++)
                scanf("%d", &array[i]);
                for (i=0; i < (n-1); i++){
                                pos=i;
                                for(d=i+1; d<n; d++){
                                 if (array[pos] > array[d])
                                    pos=d;
                                }
                                if (pos!=i){
                                    swap=array[i];
                                    array[i]=array[pos];
                                    array[pos]=swap;
                                }
                }
                printf("Lista ordenada:  \n");
                for (i=0; i<n; i++)
                      printf("%d \n", array[i]);
return 0;
        }
```

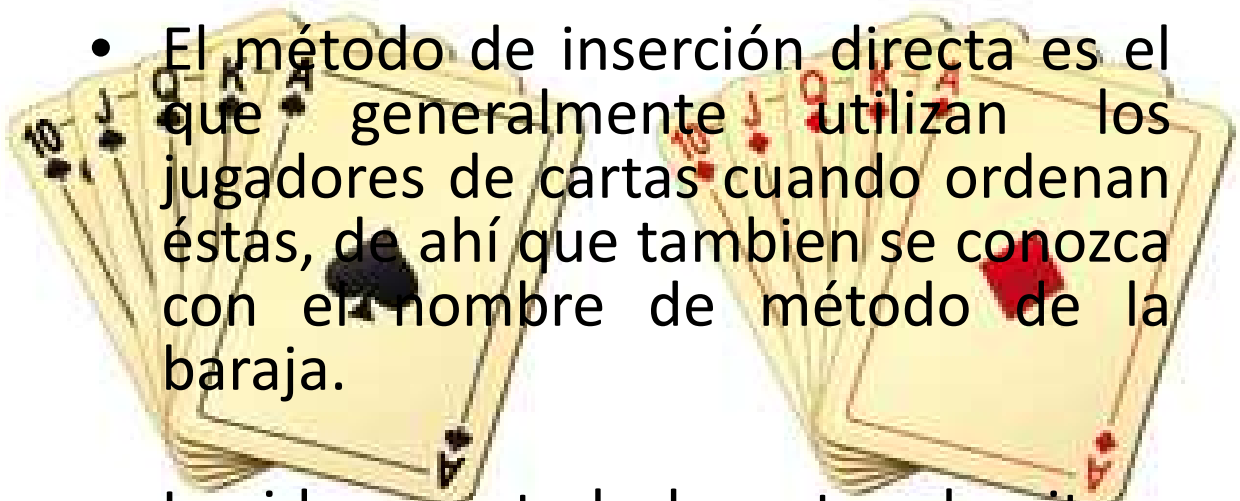
Ordenación por inserción directa (método de la baraja)

- Consiste en insertar un elemento en el vector en una parte ya ordenada de este vector y comenzar de nuevo con los elementos restantes.
- Esta inserción se realiza más fácilmente con una bandera (sw)

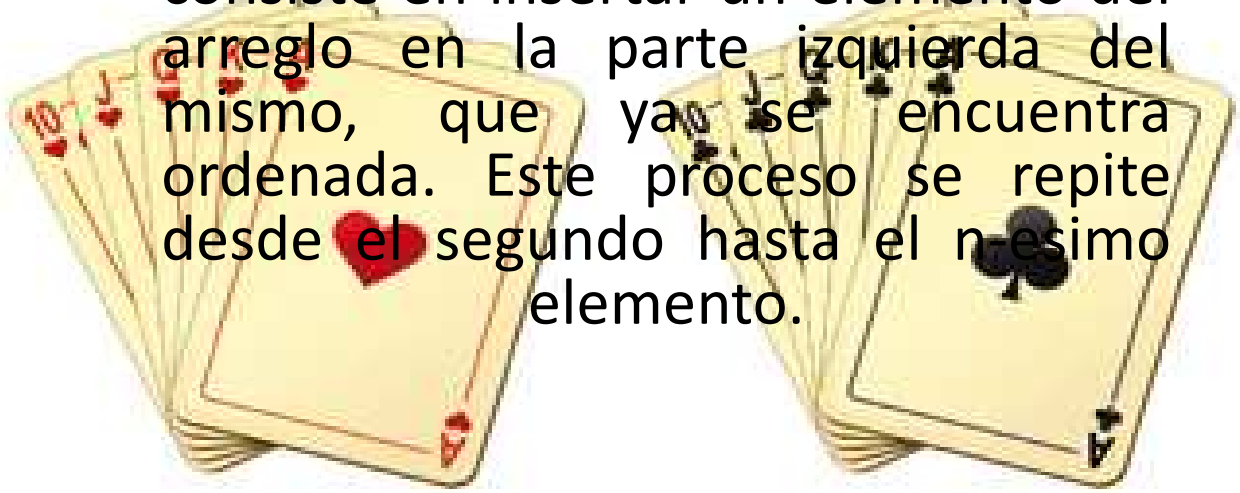


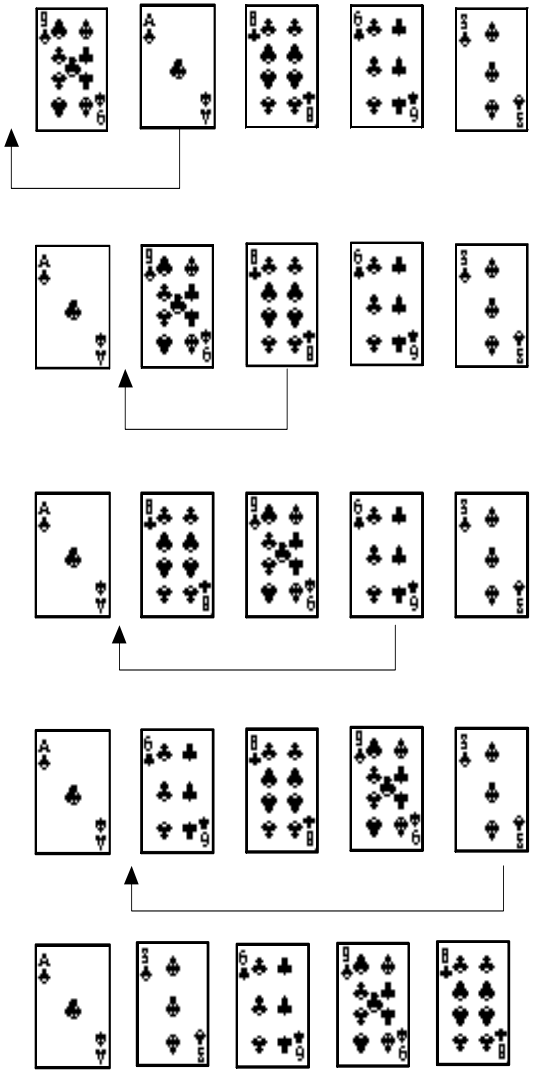
Ordenación por inserción directa

- El método de inserción directa es el que generalmente utilizan los jugadores de cartas cuando ordenan éstas, de ahí que tambien se conozca con el nombre de método de la baraja.

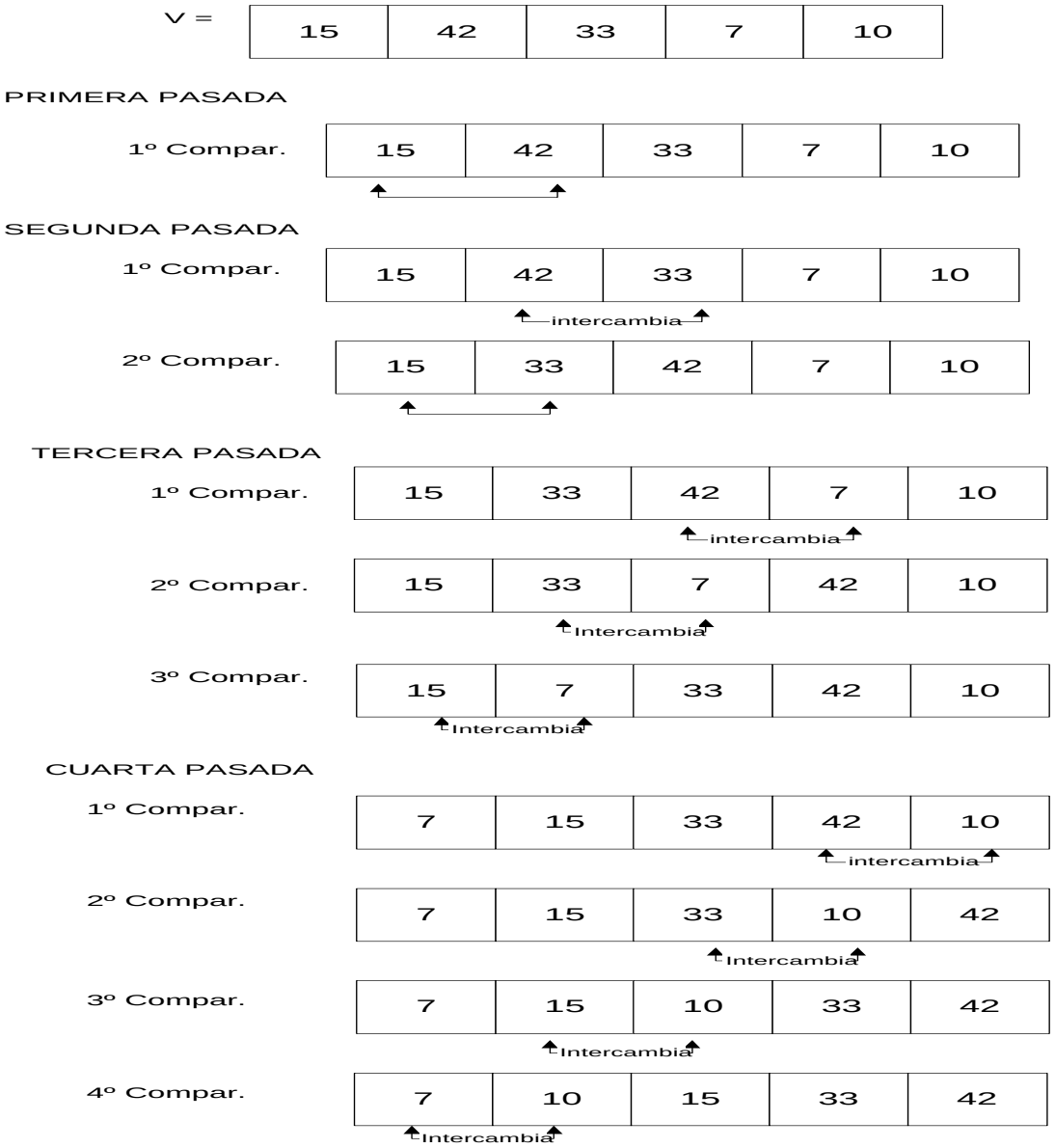


La idea central de este algoritmo consiste en insertar un elemento del arreglo en la parte izquierda del mismo, que ya se encuentra ordenada. Este proceso se repite desde el segundo hasta el n-esimo elemento.





Ordenación por inserción directa



Ordenación por inserción directa

El número de comparaciones que realiza este algoritmo se puede calcular fácilmente. Si el elemento X a insertar es mayor que los elementos restantes, el algoritmo realiza solo una comparación; si es menor a los elementos restantes, el algoritmo ejecuta *n-1* comparaciones. Por lo tanto el número de comparaciones en promedio será la mitad de dicho número.

Según el orden en que se encuentren los elementos dentro del vector podemos tener:

- Si los elementos están ordenados completamente realizamos *(n-1)* comparaciones como mínimo.
- Si los elementos están en orden inverso tenemos un máximo de

  $$n(n-1)/2 = (n2-n)/2$$

- Si los elementos aparecen en el arreglo en forma aleatoria, el número de comparaciones se calcula sobre la base del promedio, que no es mas que la suma de las comparaciones mínimas y máximas dividido 2.

  Comparaciones promedio = $[(n-1) + (n2-n)/2] / 2$

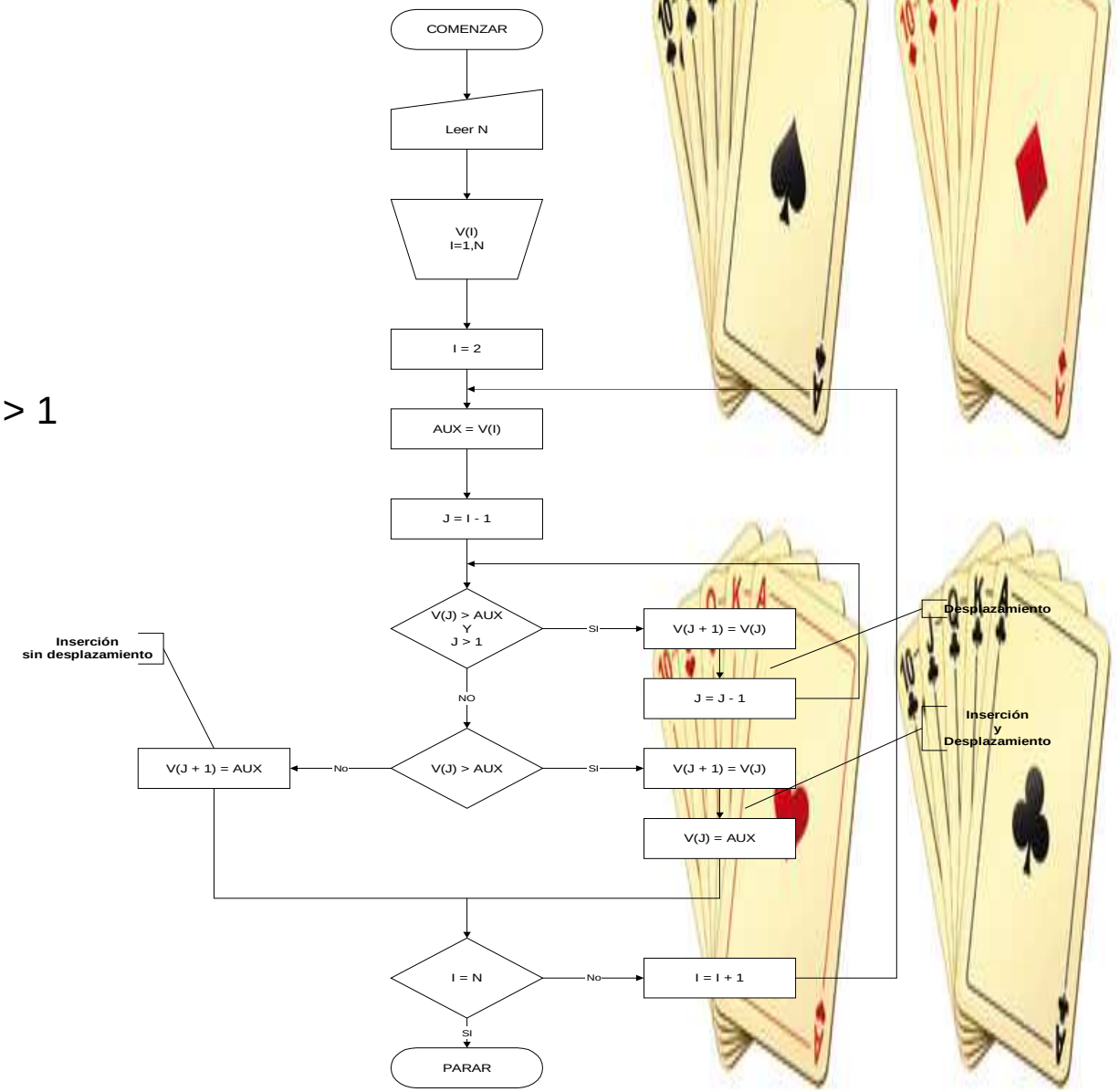
- *Luego de las distintas operaciones queda:*

  Comparaciones promedio = ***(n2+n-2) / 4***

Ordenación por inserción directa

Comenzar
Leer N
Ingresar V(I) I = 1,N
Desde I = 2 **hasta** N
    AUX = V(I)
    J = I –1
    Mientras V(J) > AUX **y** J > 1
      V(J+1) = V(J)
      J = J - 1
    Fin_mientras
    Si V(J) > AUX **Entonces**
          V(J+1) = V(J)
          V(J) = AUX
        Si_no
                V(J+1) = AUX
    Fin_si

Fin_desde
Parar

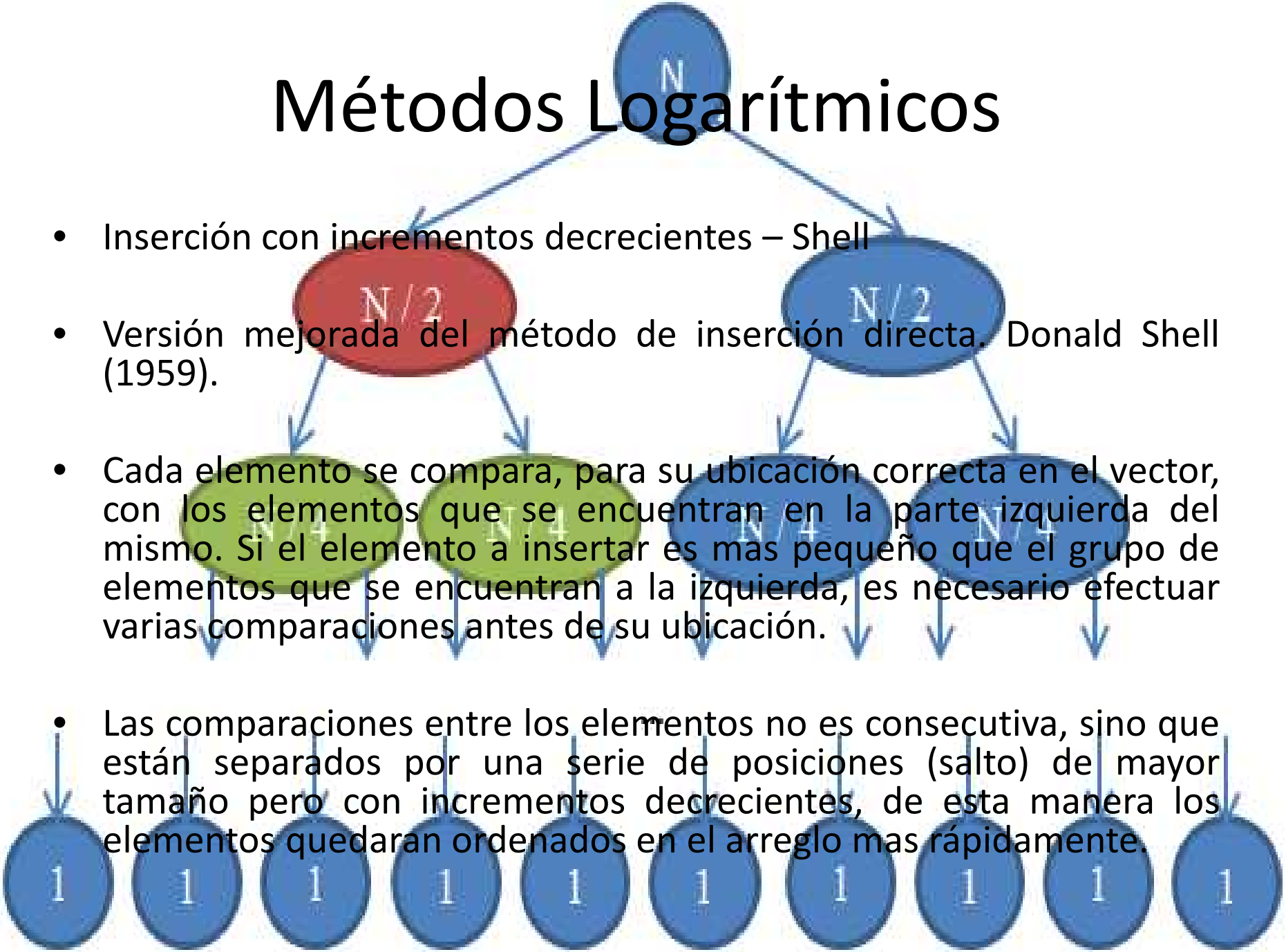


Método Inserción

```
#include<stdio.h>
int a[10]={56,41,78,11};
int n=4;
int i,j,aux;
void main(){
	for(i=1;i<n;i++) {
		j=i;
		aux=a[i];
		while(j>0 && aux<a[j-1]) {
			a[j]=a[j-1];
			j--;
		}
		a[j]=aux;
		}
	for(i=0;i<4;i++)
	{
	printf("%d \n",a[i]);
	}
	getch();
}
```

Métodos Logarítmicos

- Inserción con incrementos decrecientes – Shell
- Versión mejorada del método de inserción directa. Donald Shell (1959).
- Cada elemento se compara, para su ubicación correcta en el vector, con los elementos que se encuentran en la parte izquierda del mismo. Si el elemento a insertar es mas pequeño que el grupo de elementos que se encuentran a la izquierda, es necesario efectuar varias comparaciones antes de su ubicación.
- Las comparaciones entre los elementos no es consecutiva, sino que están separados por una serie de posiciones (salto) de mayor tamaño pero con incrementos decrecientes, de esta manera los elementos quedaran ordenados en el arreglo mas rápidamente.



Inserción con incrementos decrecientes – Shell

Consiste en:

Se determina el salto (S) entre los elementos a comparar. Es la parte entera de sumar 1 a la longitud del vector, y dividirlo entre dos.

Se recorre comparando el primer elemento con el que esta situado “S” posiciones mas adelante. Si los elementos comparados están en orden incorrectos, se intercambian; caso contrario quedan donde están.

Luego se comparan los dos elementos siguientes y así sucesivamente.

Este proceso se repite hasta que uno de los elementos de la comparación sea situado en la ultima pocisión del vector.

Si durante el recorrido hubo algún intercambio, se vuelve a repetir el proceso con el mismo salto (hasta que no se registren intercambios).

Luego se modifica el salto: será la parte entera del resultado de sumar 1 al salto anterior, y dividirlo entre dos.

El proceso de ordenación finaliza cuando, siendo el salto (S) sea igual a 1, se haya realizado una pasada sin intercambios.

Un ejemplo practico

V =

15	42	33	7	10

Se calcula el salto: Ent(5+1)/2 = 3

15	42	33	7	10

intercambio

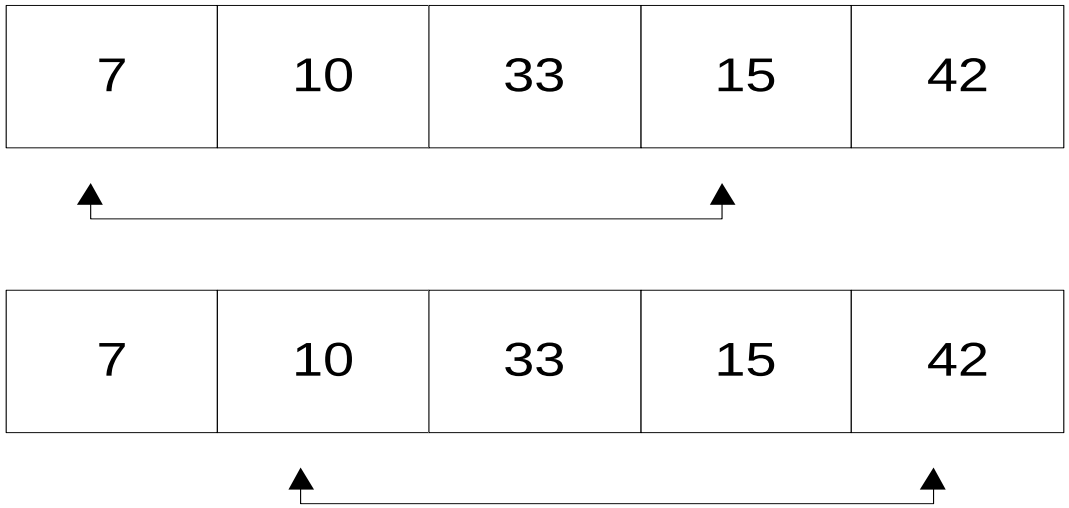
7	42	33	15	10

intercambio

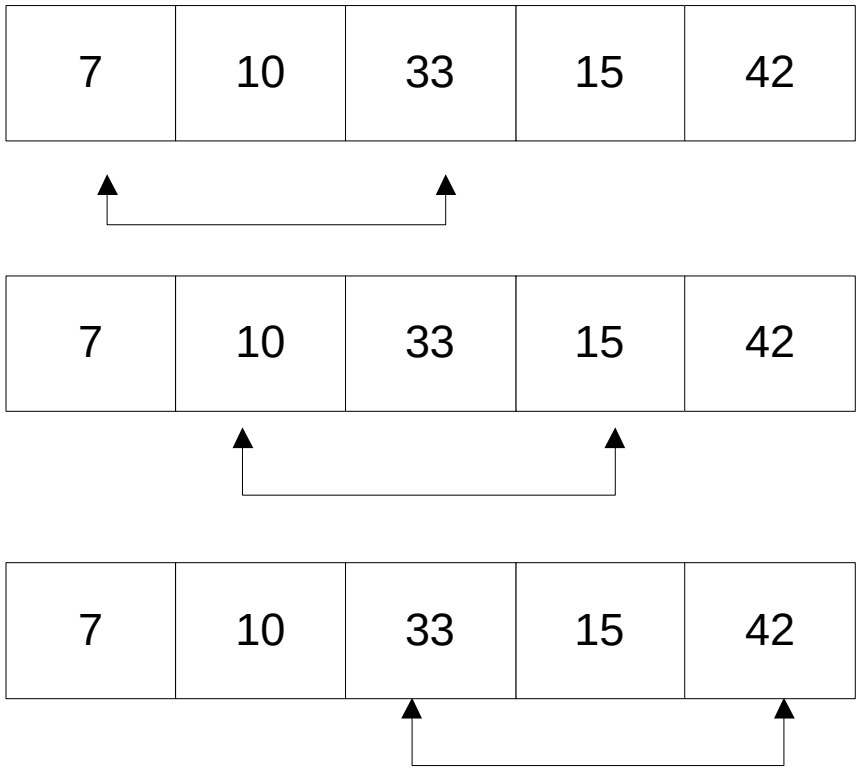
Al final del recorrido queda

7	10	33	15	42

Como en el recorrido se ha realizado al menos un intercambio, se vuelve a recorrer con el mismo salto



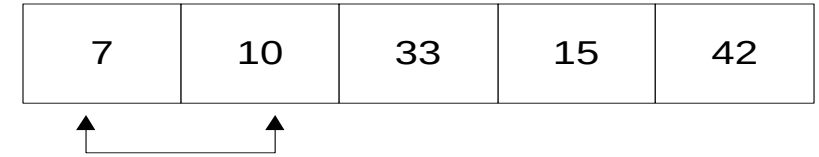
Se calcula el salto: Ent(3+1)/2 = 2

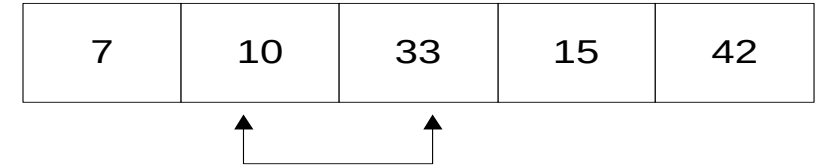


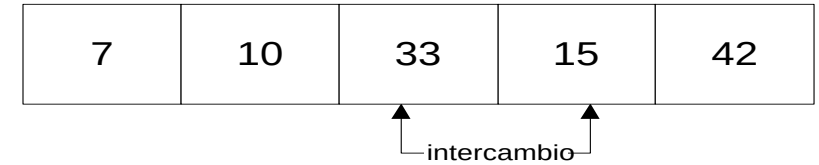
En el recorrido anterior no se realizo ningún intercambio

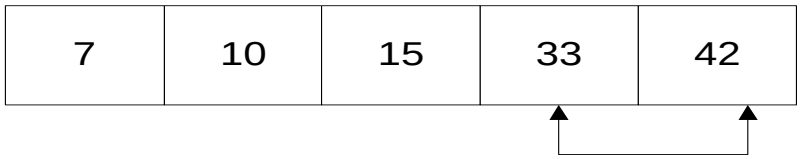
Se calcula el salto de nuevo:

Ent(2+1)/2 = 1

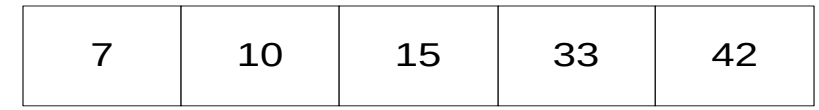


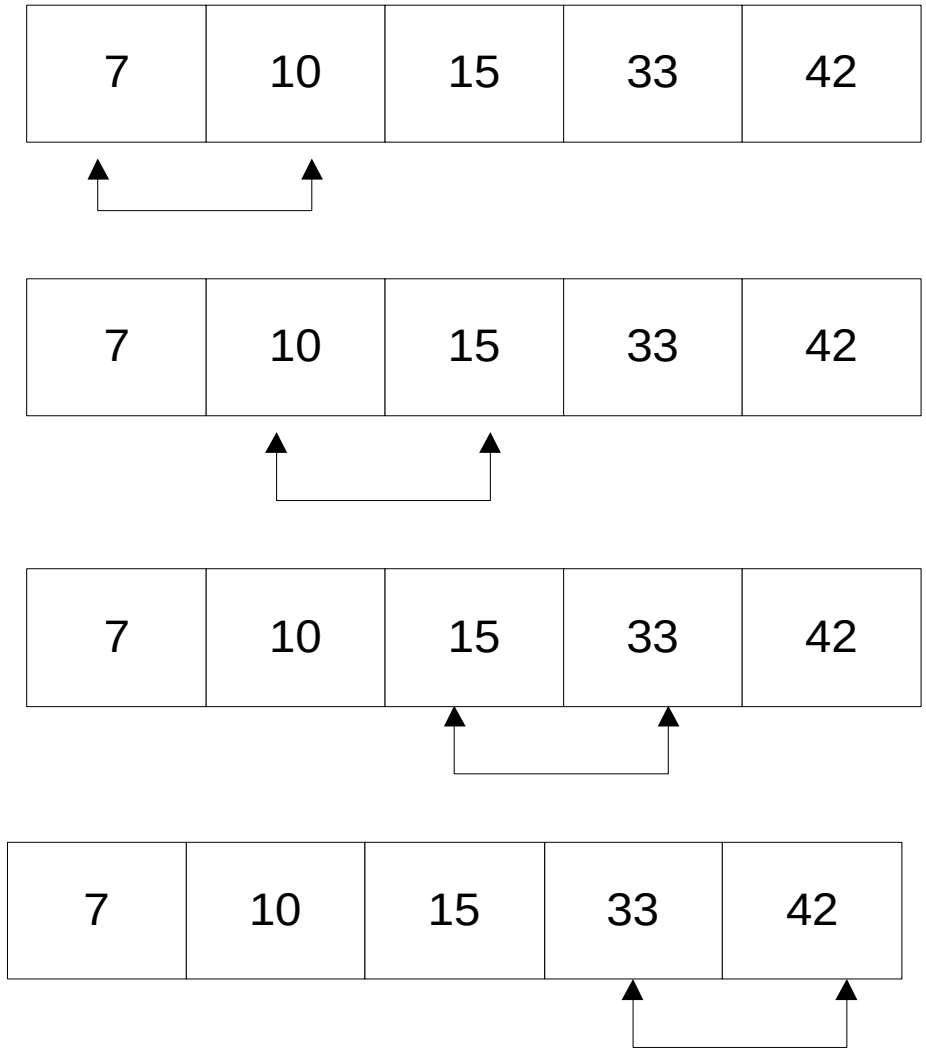




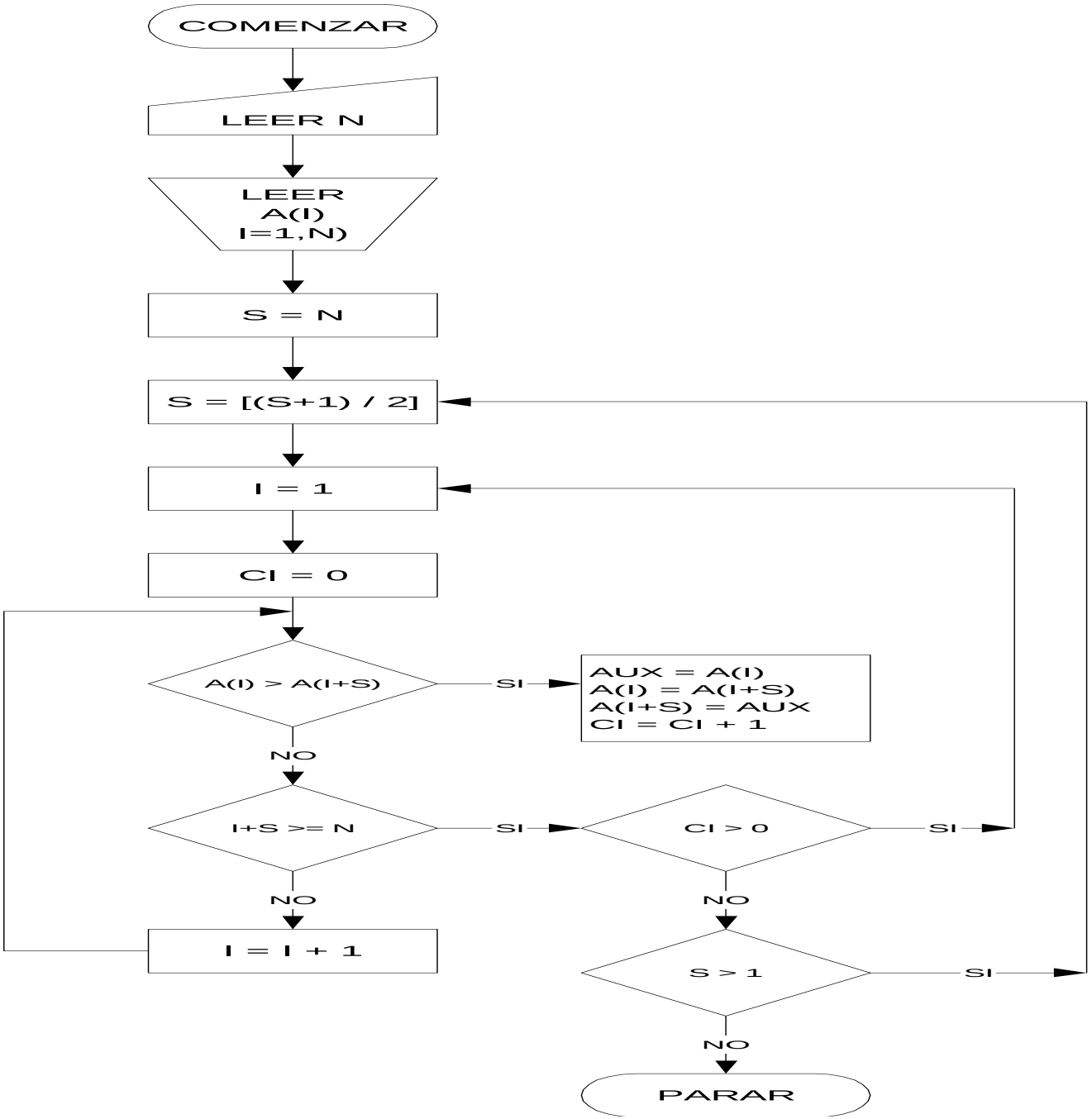


Al final del recorrido la tabla quedaría:





Como en esta pasada no se realizaron intercambio y el salto es igual a 1, se termina el proceso de ordenación.


COMENZAR
LEER N
LEER
A(I)
I=1,N)
S = N
S = [(S+1) / 2]
I = 1
CI = 0
A(I) > A(I+S)
SI
AUX = A(I)
A(I) = A(I+S)
A(I+S) = AUX
CI = CI + 1
NO
I+S >= N
SI
CI > 0
SI
NO
NO
I = I + 1
S > 1
SI
NO
PARAR

Método Shell

```
#include<stdio.h>
int a[5];
int n=5;
void main() {
        int inter=(n/2),i=0,j=0,k=0,aux;
        for (i=0; i<5; i++) {
                printf("INSERTA UN VALOR DEL INDICE: %d ", i);
                scanf("%d",&a[i]);
                }
        while(inter>0){
                for(i=inter;i<n;i++) {
                   j=i-inter;
                      while(j>=0) {
                         k=j+inter;
                         if(a[j]<=a[k]){
                            j--;
                           }
                           else{
                           aux=a[j];
                           a[j]=a[k];
                           a[k]=aux;
                           j=j-inter;
                           }
                 }
                }
          inter=inter/2;
      }
    for(i=0;i<5;i++) {
          printf("%d \n",a[i]);
           }
 }
```

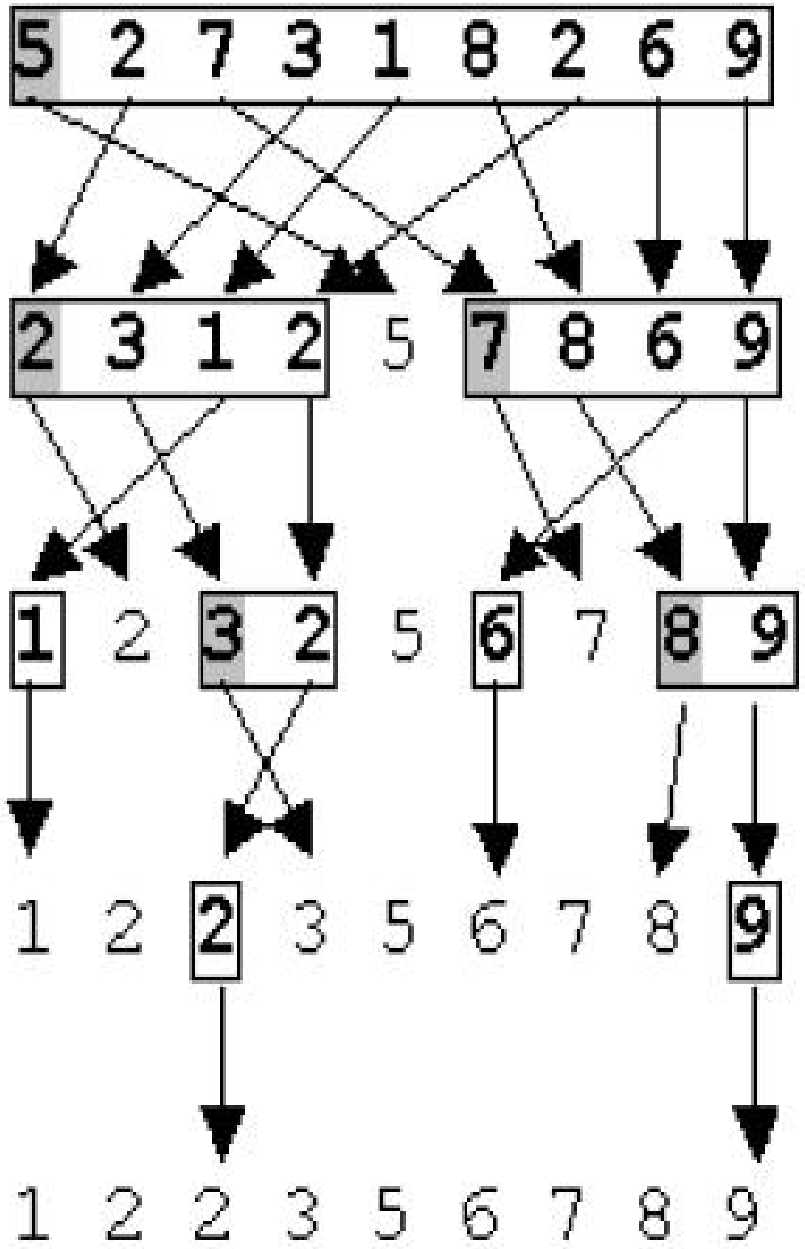
Este método es una mejora del método de intercambio directo y su autor *C.A. Hoare* lo llamó **Quicksort** (ordenación rápida) por la velocidad con que ordena los elementos de un arreglo.

Actualmente es uno de los más eficientes y más veloz de los métodos de ordenación interna, y se basa en el hecho de que es más rápido y fácil ordenar dos listas pequeñas que una lista grande.

METODO DE ORDENAMIENTO QUICK SORT

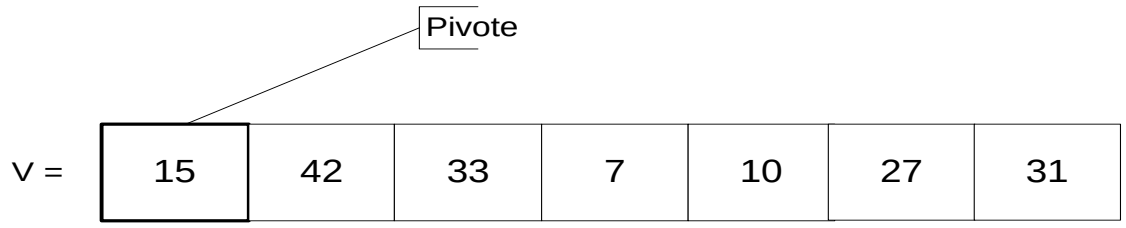
La idea central de este algoritmo consiste en lo siguiente:

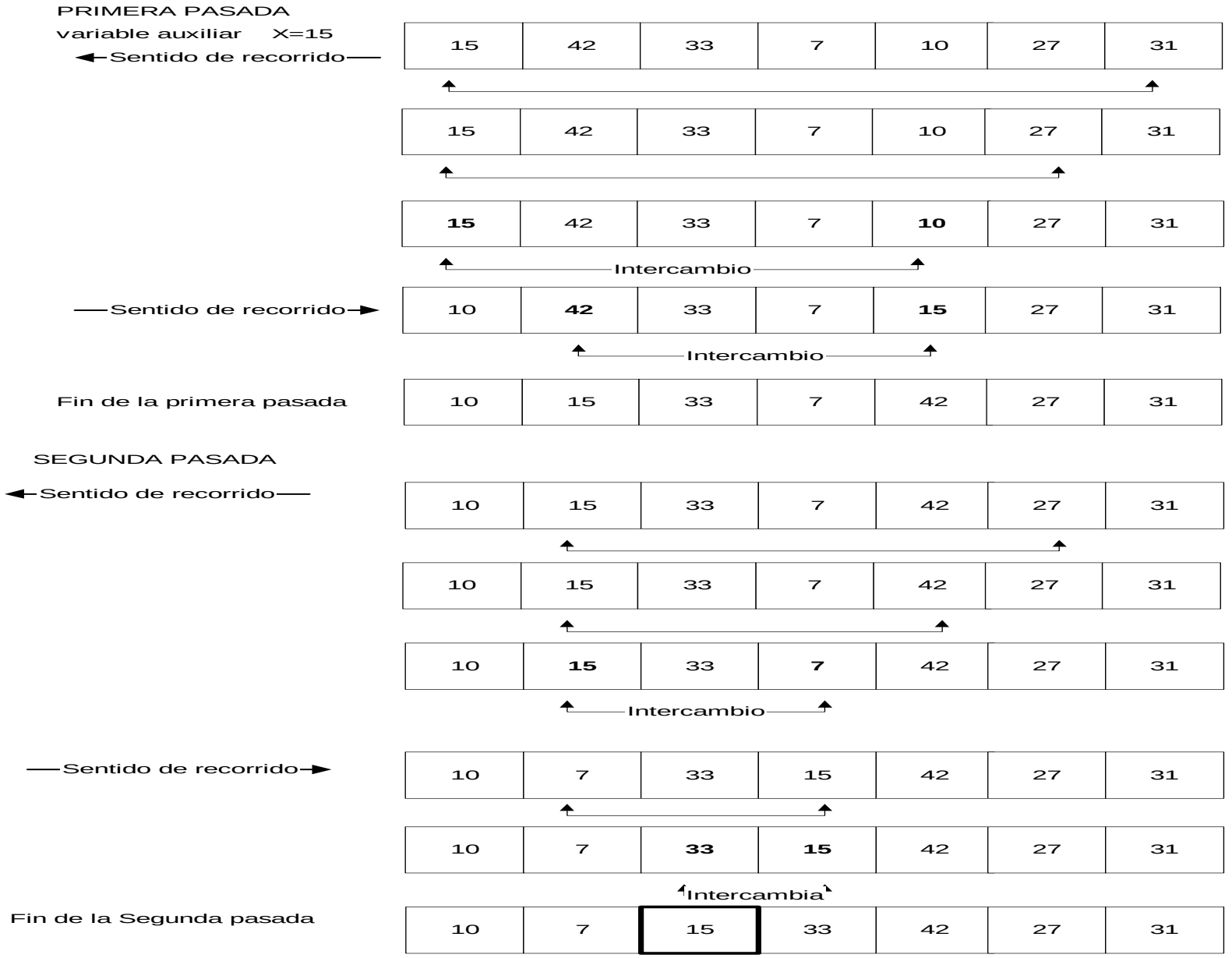
- Se toma un elemento X llamado *pivote* (valor elegido en forma arbitraria)
- Se trata de ubicar a X en la posición correcta del arreglo, de forma tal que todos los elementos que se encuentren a su izquierda sean menores o iguales a X y a su vez todos lo elementos que se encuentren a la derecha sean mayores o iguales a X.
- Estos dos pasos se repiten pero ahora para los conjuntos de datos que se encuentran a la izquierda y a la derecha de la posición correcta X.
- El algoritmo termina cuando todos los elementos se encuentran en su posición correcta del arreglo.



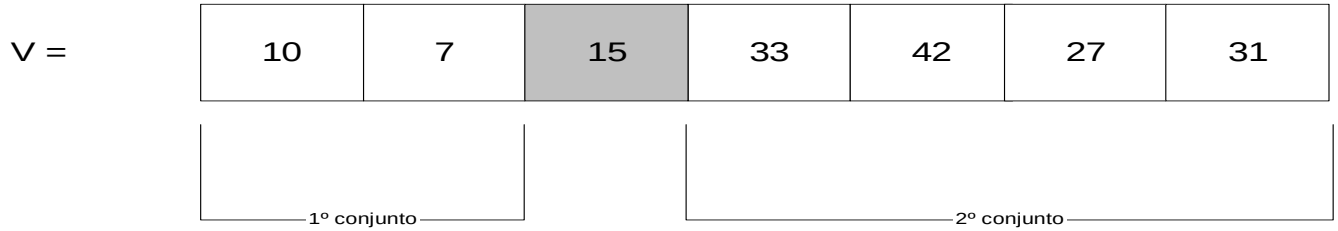
Método de QuickSort (Ordenación Rápida)

1. Se debe seleccionar un elemento pivote (X) cualquiera, por ejemplo V(1).
2. Se comienza a recorrer el arreglo de derecha a izquierda comparando si los elementos son mayores o iguales a X.
3. Si un elemento no cumple la condición se intercambian los mismos y se almacena en una variable auxiliar la posición del elemento intercambiado (con este mecanismo estamos acotando el arreglo por la derecha).
4. Se inicia nuevamente el recorrido del arreglo pero ahora de izquierda a derecha, comparando si los elementos son menores o iguales a X.
5. Y al igual que en proceso anterior, si un elemento no cumple la condición se intercambian los mismos y se almacena en otra variable auxiliar la posición del elemento intercambiado (con este mecanismo estamos acotando el arreglo por la izquierda).
6. Se repite los pasos descriptos anteriormente hasta que el elemento X encuentra su posición correcta en el arreglo.

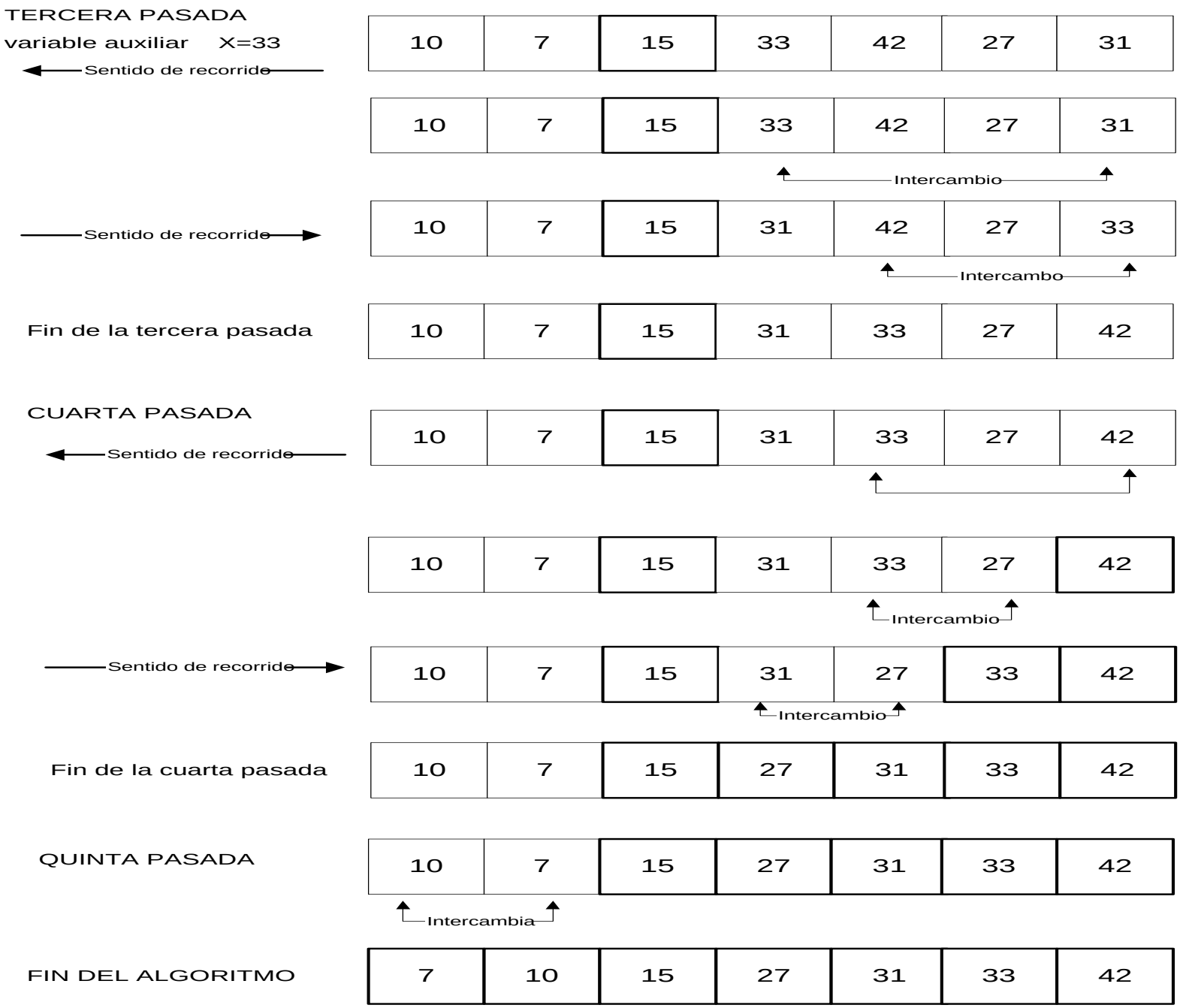



PRIMERA PASADA
variable auxiliar X=15
Sentido de recorrido
15 42 33 7 10 27 31
15 42 33 7 10 27 31
15 42 33 7 10 27 31
Intercambio
Sentido de recorrido
10 42 33 7 15 27 31
Intercambio
Fin de la primera pasada
10 15 33 7 42 27 31
SEGUNDA PASADA
Sentido de recorrido
10 15 33 7 42 27 31
10 15 33 7 42 27 31
10 15 33 7 42 27 31
Intercambio
Sentido de recorrido
10 7 33 15 42 27 31
10 7 33 15 42 27 31
Intercambia
Fin de la Segunda pasada
10 7 15 33 42 27 31

- Como el recorrido de izquierda a derecha debería iniciarse en la misma posición donde se encuentra el elemento X, el proceso termina ya que dicho elemento se encuentra en su posición correcta.



Se observa que los elementos que forman parte del 1º conjunto son menores o iguales a X elegido, y los que forman parte del segundo conjunto son mayores o iguales a X. Este proceso de particionamiento aplicado para localizar la posición correcta de un elemento X en el arreglo se repite cada vez que queden conjuntos formados por dos o mas elementos. El proceso a repetir se realiza en forma iterativa o recursiva.


TERCERA PASADA
variable auxiliar X=33
Sentido de recorrido
10 7 15 33 42 27 31
10 7 15 33 42 27 31
Intercambio
Sentido de recorrido
10 7 15 31 42 27 33
Intercambo
Fin de la tercera pasada
10 7 15 31 33 27 42
CUARTA PASADA
Sentido de recorrido
10 7 15 31 33 27 42
10 7 15 31 33 27 42
Intercambio
Sentido de recorrido
10 7 15 31 27 33 42
Intercambio
Fin de la cuarta pasada
10 7 15 27 31 33 42
QUINTA PASADA
10 7 15 27 31 33 42
Intercambia
FIN DEL ALGORITMO
7 10 15 27 31 33 42

- Los distintos estudios realizados sobre el comportamiento del método demuestran que si se escoge en cada pasada el elemento que ocupa la posición central del conjunto de datos a ordenar, la cantidad de pasadas necesarias para ordenar un arreglo es del orden *log n*. En cambio la cantidad de comparaciones, si el tamaño del arreglo es una potencia de 2, en la primera pasada se realiza (n-1) comparaciones, en la segunda pasada será (n-1) / 2 comparaciones, en la tercer pasada (n-1)/4 y así sucesivamente. Por lo tanto la fórmula resultaría:

$$C= (n-1) + 2* (n-1)/2 + 4 * (n-1)/4 + ...+ (n-1) * (n-1) / (n-1)$$

Que se puede expresar de la siguiente manera:

$$C = (n-1) + (n-1) + (n-1) + ...+ (n-1)$$

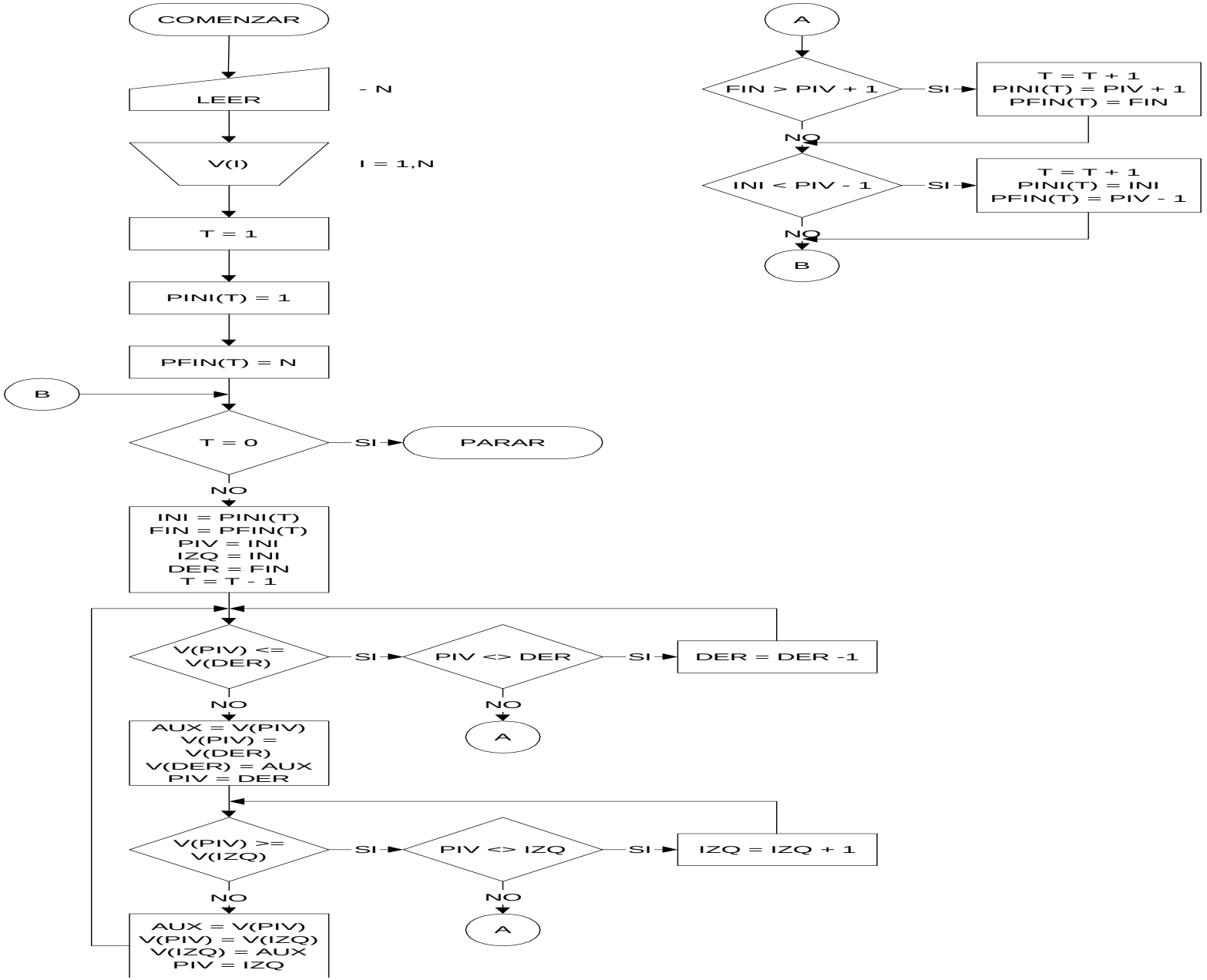
- Si se considera a cada uno de los componentes de la sumatoria como un término, y si el número de términos de la sumatoria es igual a *m*, entonces resulta:

$$C = (n-1) * m$$

- Pero si consideramos que *m* es el número de términos de la sumatoria e igual al número de pasadas, y esta a su vez es igual a *log n*, la expresión anterior quedaría:

$$C = (n-1) * \log n$$

- Si se analiza el tiempo de ejecución del método, se podría afirmar que el tiempo promedio del algoritmo es proporcional a (*n * log n)* y en el peor de los casos el tiempo de ejecución es proporcional a *n2*.


COMENZAR
LEER
- N
V(I)
I = 1,N
T = 1
PINI(T) = 1
PFIN(T) = N
B
T = 0
SI
PARAR
NO
INI = PINI(T)
FIN = PFIN(T)
PIV = INI
IZQ = INI
DER = FIN
T = T - 1
V(PIV) <= V(DER)
SI
PIV <> DER
SI
DER = DER -1
NO
NO
A
AUX = V(PIV)
V(PIV) = V(DER)
V(DER) = AUX
PIV = DER
V(PIV) >= V(IZQ)
SI
PIV <> IZQ
SI
IZQ = IZQ + 1
NO
NO
A
AUX = V(PIV)
V(PIV) = V(IZQ)
V(IZQ) = AUX
PIV = IZQ
A
FIN > PIV + 1
SI
T = T + 1
PINI(T) = PIV + 1
PFIN(T) = FIN
NO
INI < PIV - 1
SI
T = T + 1
PINI(T) = INI
PFIN(T) = PIV - 1
NO
B

Código “C” – Método QuitSort

```
#include <stdlib.h>
#include <stdio.h>

/*funcion para intercambiar los valores de dos elementos*/
void intercambio(int* a, int* b) {
   int temp = *a;
   *a = *b;
   *b = temp;
}

/*funcion recursiva quicksort para ordenar el arreglo*/
void quicksort(int* izq, int* der) {
   if (der < izq)
      return;
   int pivote = *izq;
   int* ult = der;
   int* pri = izq;
   while (izq < der) {
      while (*izq <= pivote && izq < der+1)
         izq++;
      while (*der > pivote && der > izq-1)
         der--;
      if (izq < der)
         intercambio(izq, der);
   }
   intercambio(pri, der);
   quicksort(pri, der-1);
   quicksort(der+1, ult);
}
```

```
int main(void) {
   int i;
   int tam;

   /*definimos el tamaño del arreglo*/
   printf("Ingrese el tamaño del arreglo:\n");
   scanf("%d", &tam);
   int arreglo[tam];

   /*llenamos el arreglo*/
   printf("Ingrese valores para el arreglo:\n");
   for (i = 0; i < tam; i++)
      scanf("%d", &arreglo[i]);
   printf("\n");

   /*mostramos el arreglo original*/
   printf("Arreglo Original \n");
   for (i = 0; i < tam; i++)
      printf("%d ", arreglo[i]);
   printf("\n");

   /*hacemos el llamado a la funcion quicksort
     para que ordene el arreglo*/
   quicksort(&arreglo[0], &arreglo[tam-1]);

   /*mostramos el arreglo ordenado*/
   printf("Arreglo Ordenado \n");
   for (i = 0; i < tam; i++)
      printf("%d ", arreglo[i]);
   printf("\n\n");
}
```

Intercalación

La **intercalación** es el proceso de mezclar (*merge)* dos vectores ordenados para obtener un nuevo vector ordenado.

Supongamos que A es un vector o lista ordenada de *m* elementos y B también es una lista ordenada de *n* elementos. La operación de mezclar producirá una nueva lista C ordenada de *m+n* elementos.

El método más sencillo, pero menos eficaz, consiste en colocar una lista detrás de la otra y luego ordenarla con algún método. Sin embargo esta metodología no aprovecha la propiedad que las listas ya estan ordenadas.

Veamos un ejemplo: *Mezclar las listas de números A y B, ambas ordenadas en forma ascendente.*

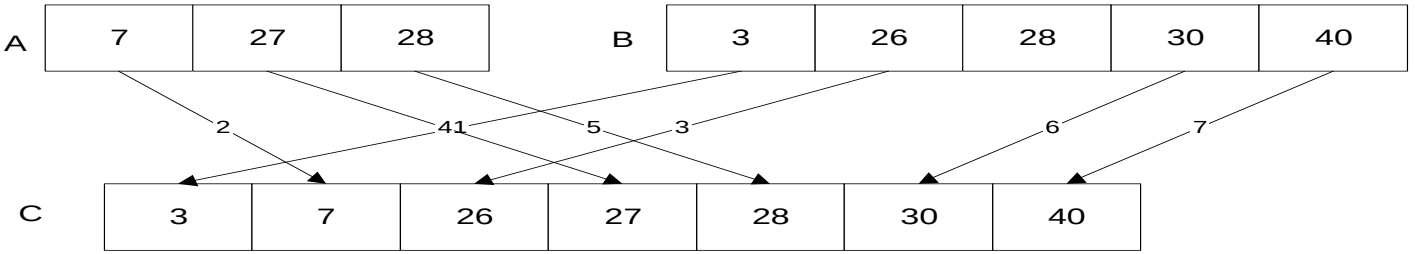
A: 7 27 20

B: 3 26 28 30 40

A	7	27	28

B	3	26	28	30	40

El proceso de intercalación de los vectores A y B se muestra en la gráfica siguiente:



Para comprender como se obtiene el vector C, realizaremos paso por paso la intercalación.

Se comparan los dos primeros elementos del los vectores y se envía el mas pequeño al primer lugar

C	3						

A continuación se comparan los dos primeros elementos que no se han enviado al vector C y nuevamente se envía el mas pequeño

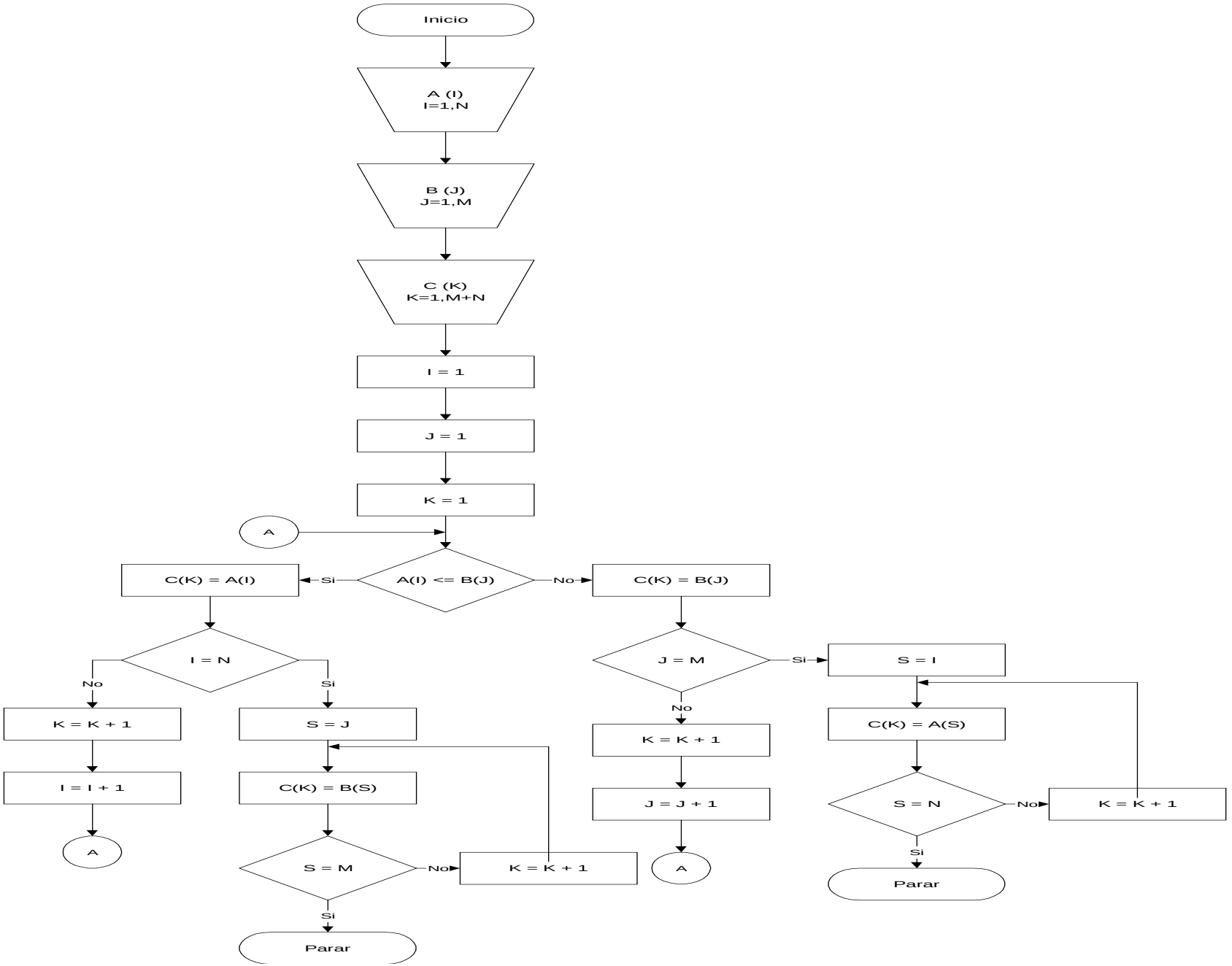
C	3	7					

Este proceso se repite hasta que a uno de los dos vectores termine (en nuestro caso vector A) y a continuación se copian los números directamente en el tercer vector.

C	3	7	26	27	28		

El vector B se copia integramente en el tercer vector C

C	3	7	26	27	28	30	40


Inicio
A (I)
I=1,N
B (J)
J=1,M
C (K)
K=1,M+N
I = 1
J = 1
K = 1
A
A(I) <= B(J)
Si
No
C(K) = A(I)
C(K) = B(J)
I = N
No
Si
K = K + 1
I = I + 1
A
S = J
C(K) = B(S)
S = M
No
K = K + 1
Si
Parar
J = M
Si
No
K = K + 1
J = J + 1
A
S = I
C(K) = A(S)
S = N
No
K = K + 1
Si
Parar

Pseudocódigo

```
function mergesort(array A[x..y])
begin
    if (x-y > 1)):
        array A1 := mergesort(A[x..(int( x+y / 2))])
        array A2 := mergesort(A[int(1+(x+y / 2))..y])
        return merge(A1, A2)
        else:
        return A
    end
function merge(array A1[0..n1], array A2[0..n2])
begin
    integer p1 := 0
    integer p2 := 0
array R[0..(n1 + n2 + 2)]//suponiendo que n1 y n2 son las posiciones
//del array y no el length de este mismo, de otro modo seria (n1 + n2)
while (p1 <= n1 or p2 <= n2):
If (p1 <= n1 and A1[p1] <= A2[p2]):
    R[p1 + p2] := A1[p1]
    p1 := p1 + 1
    else if (p2 <= n2 and A1[p1] > A2[p2]):
    R[p1 + p2] := A2[p2]
    p2 := p2 + 1
return R
end
```

Código en “C” - ´Método Mezcla

```
#include <stdio.h>
#include <stdlib.h>
void mezclar(int arreglo1[], int n1, int arreglo2[], int n2, int arreglo3[])
{
        int x1=0, x2=0, x3=0;

        while ( x1<n1 && x2<n2 ){
                if ( arreglo1[x1] < arreglo2[x2] ){
                   arreglo3[x3]=arreglo1[x1];
                   x1++;
                } else {
          arreglo3[x3]=arreglo2[x2];
                                x2++;
                }
                x3++;
        }
        while (x1<n1){
        arreglo3[x3]=arreglo1[x1];
        x1++;
        x3++;
        }
        while (x2<n2){
        arreglo3[x3]=arreglo2[x2];
        x2++;
        x3++;
        }
}
```

```
void mezcla(int vector[], int n){
        int *vector1, *vector2, n1, n2, x, y;
        if (n>1){
                if (n%2 ==0)
                n1=n2=(int) n / 2;
                n2=n1+1;
                }
        vector1=(int *) malloc (sizeof (int)*n1);
        vector2=(int *) malloc (sizeof (int)*n2);
                for (x=0;x<n1;x++)
                                vector1[x]=vector[x];
                for (y=0;y<n2;y++)
                                vector2[y]=vector[x];
     mezcla(vector1, n1);
     mezcla(vector2, n2);
     mezclar(vector1, n1, vector2, n2, vector);
     free(vector1);
     free(vector2);
  }

int main(){
        int i, n;
        int cad[100];
        printf("cuantos elementos ? \n");
        scanf("%d", &n);
        printf("Introduce los %d numeros \n", n);
        for (i=0;i<n;i++){
        scanf("%d", &cad[i]);
                }
        mezcla(cad, n);
        printf("lista ordenada: \n");
        for (i=0; i<n; i++)
        printf("%i \n", cad[i]);
return 0;
}
```

Método de ordenación por Montículo

- Es un algoritmo que se construye utilizando las propiedades de los montículos binarios.
- El orden de ejecución para el peor caso es O(N·log(N)), siendo N el tamaño de la entrada.
- Aunque teóricamente es más rápido que los algoritmos de ordenación vistos hasta aquí, en la práctica es más lento que el algoritmo de ordenación de Shell utilizando la secuencia de incrementos de Sedgewick.

- ¿Qué es un montículo?
- Un montículo es un árbol binario balanceado que cumple con la premisa de que: *ningún padre tiene un hijo mayor (montículo de máximos) o menor (montículo de mínimos) a él*. Una definición corta y concreta, ¿no?
- Esta estructura de datos fue propuesta por Robert W. Floyd (premio Turing en 1978) para resolver el problema de la ordenación de elementos dentro de un vector, el famoso *heapsort* (u ordenación por montículo).

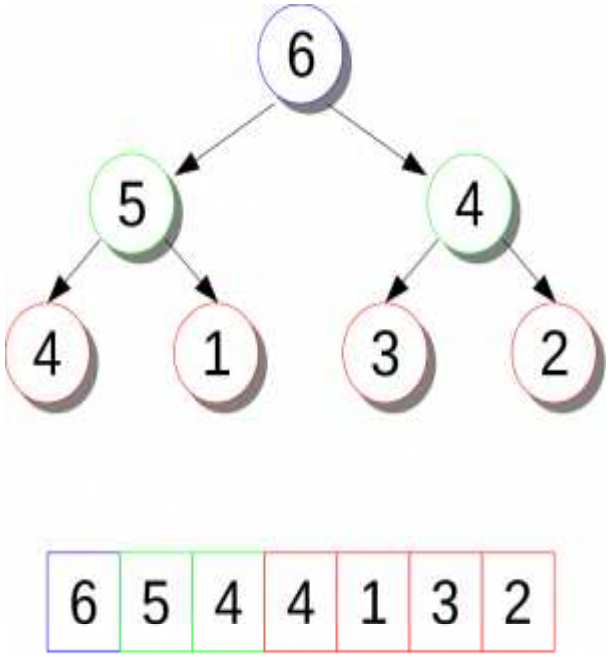
Breve repaso de las propiedades de los monticulos binarios (heaps)

Recordemos que un montículo Max es un árbol binario completo cuyos elementos están ordenados del siguiente modo: para cada subárbol se cumple que la raíz es mayor que ambos hijos. Si el montículo fuera Min, la raíz de cada subárbol tiene que cumplir con ser menor que sus hijos.

Recordamos que, si bien un montículo se define como un árbol, para representar éste se utiliza un array de datos, en el que se acceden a padres e hijos utilizando las siguientes transformaciones sobre sus índices. Si el montículo está almacenado en el array A, el padre de A[i] es A[i/2] (truncando hacia abajo), el hijo izquierdo de A[i] es A[2*i] y el hijo derecho de A[i] es A[2*i+1].

Al insertar o eliminar elementos de un montículo, hay que cuidar de no destruir la propiedad de orden del montículo. Lo que se hace generalmente es construir rutinas de filtrado (que pueden ser ascendentes o descendentes) que tomen un elemento del montículo (el elemento que viola la propiedad de orden) y lo muevan verticalmente por el árbol hasta encontrar una posición en la cual se respete el orden entre los elementos del montículo.

Tanto la inserción como la eliminación (`eliminar_min` o `eliminar_max` según sea un montículo Min o Max respectivamente), de un elemento en un montículo se realizan en un tiempo O(log(N)), peor caso (y esto se debe al orden entre sus elementos y a la característica de árbol binario completo).



El montículo, aunque conceptualmente se dibuja en forma de árbol, está implementado sobre un vector. Ya que es balanceado y binario, un nodo solo puede contener dos hijos. La formula para acceder a cada hijo, dado el índice del padre (i), sería:
hijo_izquierdo = 2i hijo_derecho = 2i+1
El acceso al padre se realizaría con una división entera: i / 2.

¿Para qué sirve un montículo?

El montículo en sí permite:

- Ordenar elementos de un vector de forma fácil y óptima, pudiendo implementar un comparador a medida.
- Búsquedas el máximos, mínimos, o cualquier otro factor a tener en cuenta en la selección de búsqueda de elementos.
- En problemas específicos de grafos como el recubrimiento mínimo de Prim o el camino más corto de Dijkstra.

Estrategia general del algoritmo. A grandes razgos el algoritmo de ordenación por montículos consiste en meter todos los elementos del array de datos en un montículo MAX, y luego realizar N veces `eliminar_max()`. De este modo, la secuencia de elementos eliminados nos será entregada en ordes decreciente.

Implementación práctica del algoritmo. Existen dos razones por las que la estrategia general debe ser refinada: el uso de un tad auxiliar (montículo binario) lo cual podría implicar demasiado código para un simple algoritmo de ordenación, y la necesidad de memoria adicional para el montículo y para una posible copia del array. Estas cosas también implican un gasto en velocidad.

Lo que se hace es reducir el código al máximo reduciéndose lo más posible la abstracción a un montículo. Primero que nada, dado el array de datos, este no es copiado a un montículo (insertando cada elemento). Lo que se hace es, a cada elemento de la mitad superior del array (posiciones 0,1,....N/2) se le aplica un filtrado descendente (se "baja" el elemento por el árbol binario hasta que tenga dos hijos que cumplan con el orden del montículo. Esto bastará para hacer que el array cumpla con ser un montículo binario.

Notamos también que al hacer un eliminar_max() se elimina el primer elemento del array, se libera un lugar a lo último de este. Podemos usar esto para no tener que hacer una copia del array para meter las eliminaciónes sucesivas. Lo que hacemos es meter la salida de eliminar_max() luego del último elemento del montículo. Esto hace que luego de N-1 eliminar_max(), el array quede ordenado de menor a mayor.

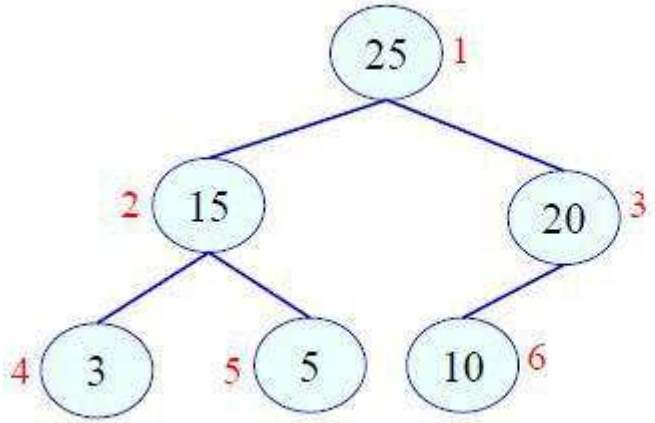
Para ahorrar código queremos lograr insertar y eliminar_max() con una sola rutina de filtrado descendente. Ya explicamos cómo hacer que el array de datos A preserve el orden de los elementos de un montículo. Lo que hacemos para ordenarlo usando sólo la rutina de filtrado descendente es un eliminar_max() intercambiando el primer elemento del array por el último del montículo y filtrar el nuevo primer elemento hasta que se respete el orden Max. Quizá esto quede más claro mirando el código.

```
Function Heapsort(array A[0..n]):
montículo M
integer i; // declaro variable i
for i = 0..n:
insertar_en_monticulo(M, A[i])
for i = 0..n:
A[i] = extraer_cima_del_monticulo(M)
return A
```

Heapsort - ejemplo

- Aplicar HeapSort( ) para ordenar el conjunto de datos,
  S = { 25, 15, 20, 3, 5, 10 }

construir el árbol, y numerar los nodos :

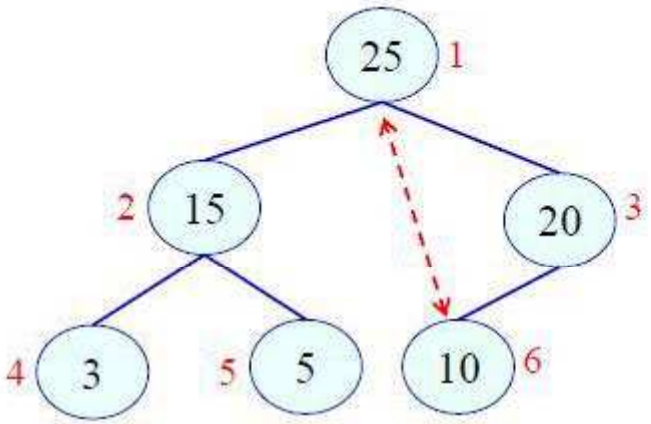


25	15	20	3	5	10
1	2	3	4	5	6

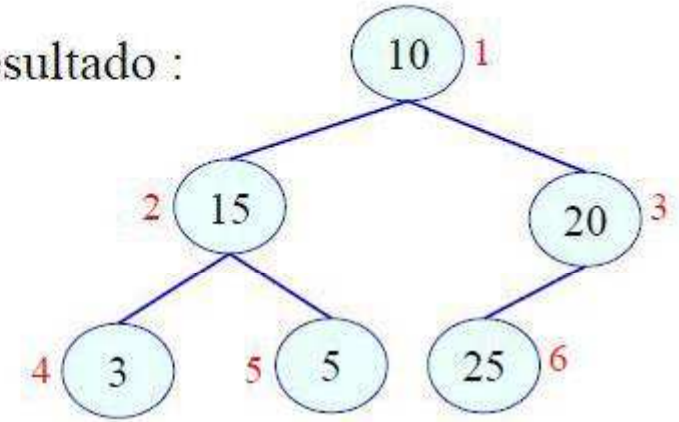
Verificar la condición de heap

k = 6 ; (k = nº de nodos)

Intercambiar nodo 1 con nodo 6 :

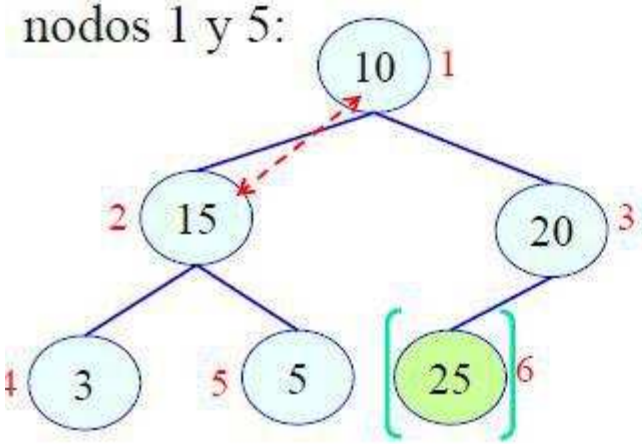


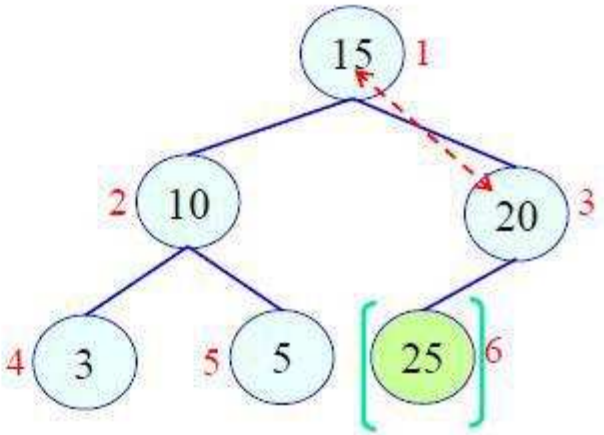
Resultado :



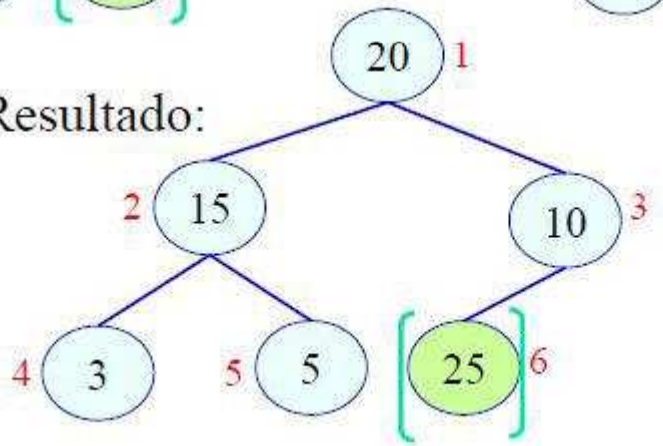
10	15	20	3	5	25
1	2	3	4	5	6

Revisar y recuperar condición de heap, si no se cumple, entre los nodos 1 y 5:



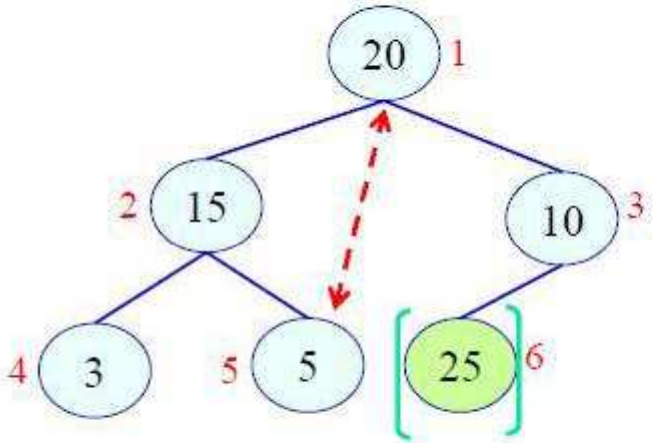


Resultado:

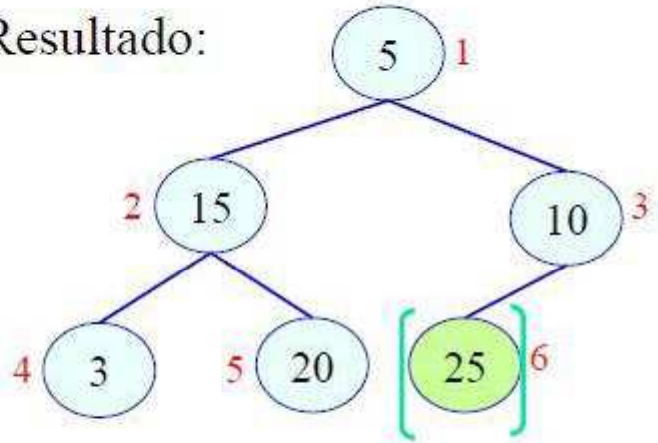


20	15	10	3	5	25
1	2	3	4	5	6

k = 5; intercambiar los nodos 1 y 5 :

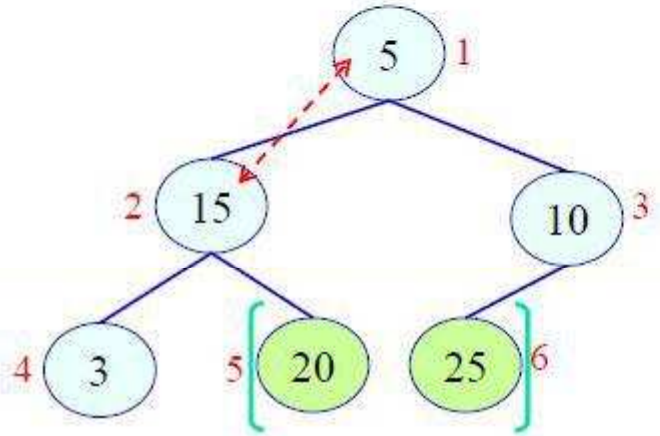


Resultado:

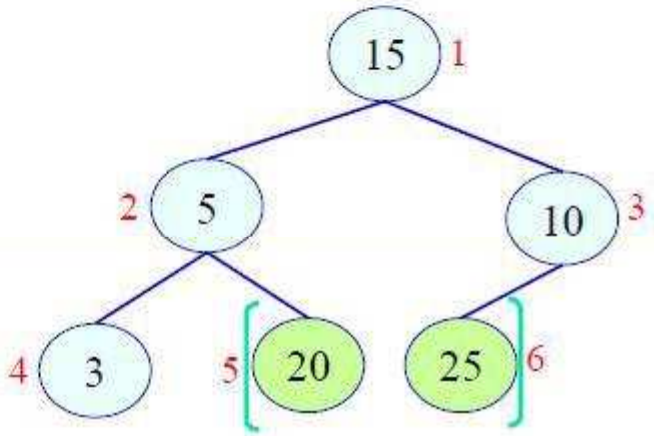


5	15	10	3	20	25
1	2	3	4	5	6

Revisar y recuperar condición de heap, si no se cumple, entre los nodos 1 y 4:

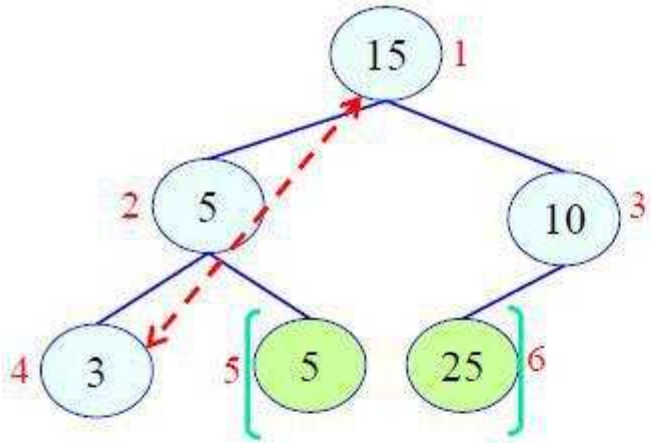


Resultado :

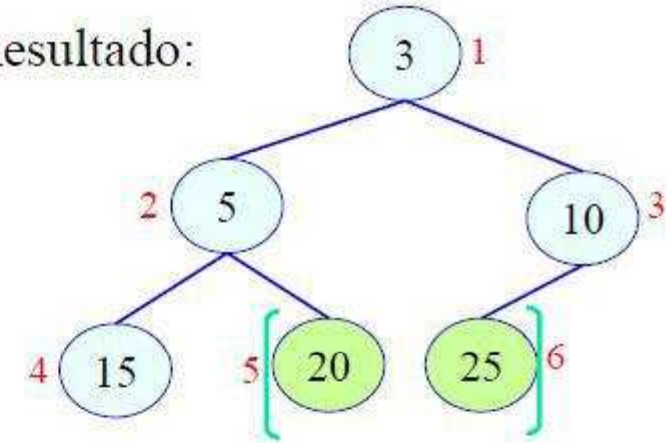


15	5	10	3	20	25
1	2	3	4	5	6

k = 4; intercambiar los nodos 1 y 4 :

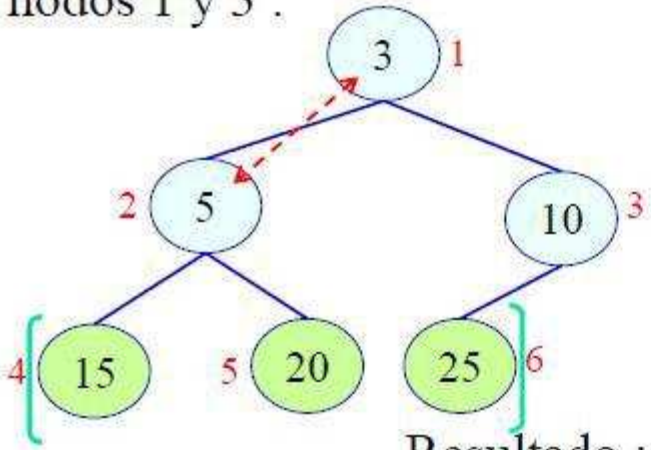


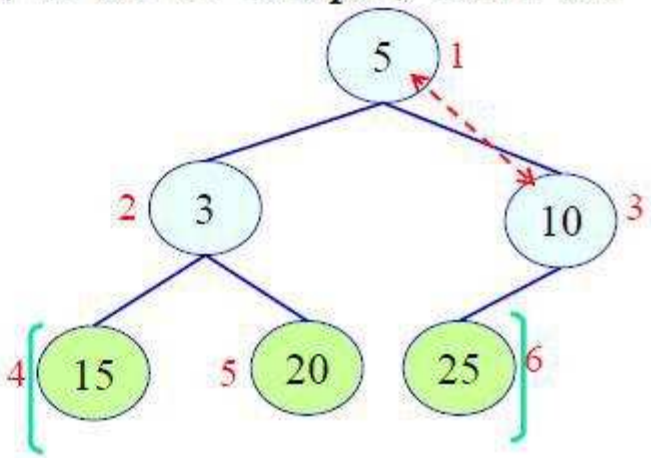
Resultado:



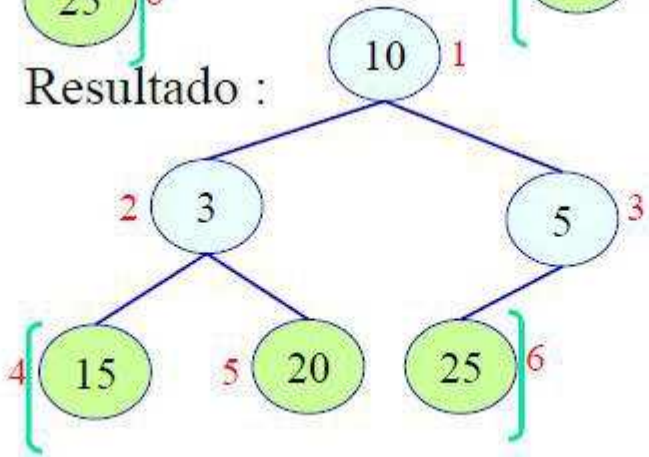
3	5	10	15	20	25
1	2	3	4	5	6

Revisar y recuperar condición de heap, si no se cumple, entre los nodos 1 y 3 :



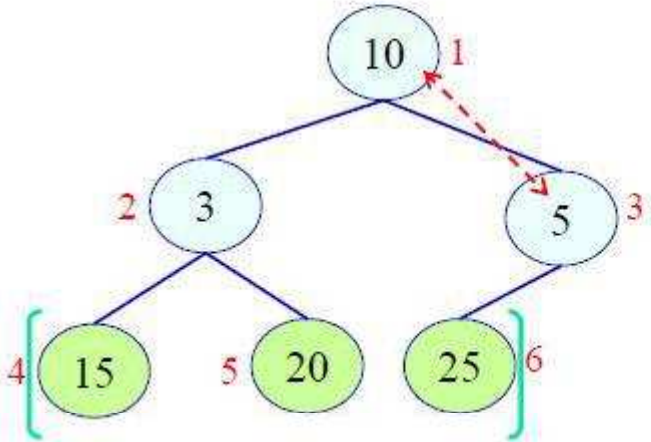


Resultado :

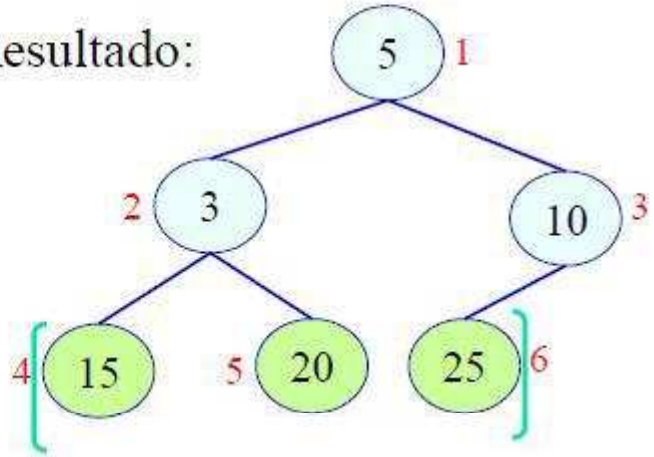


10	3	5	15	20	25
1	2	3	4	5	6

k = 3; intercambiar los nodos 1 y 3 :

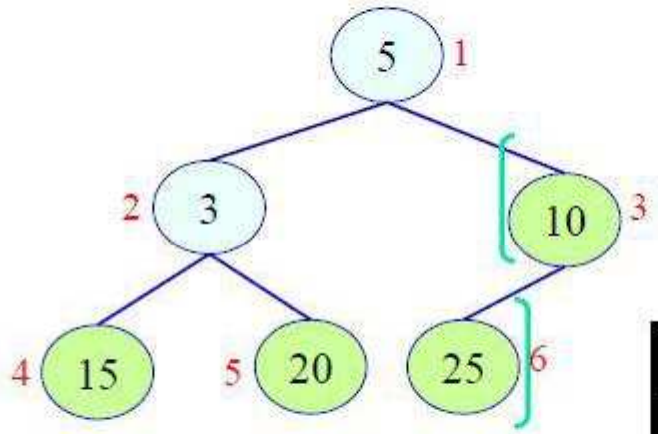


Resultado:



5	3	10	15	20	25
1	2	3	4	5	6

Revisar y recuperar condición de heap, si no se cumple, entre los nodos 1 y 2.



5	3	10	15	20	25
1	2	3	4	5	6

Código en C – Método Heap

```
#include <stdio.h>
long aiData[125001];
long i, iN, iHalf, iTemp;
int odd(long iX){
        return ((iX & 0x0001)!=0);
}
void swap(long *piVar1, long *piVar2){
        long iTemp;
        iTemp=*piVar1;
        *piVar1=*piVar2;
        *piVar2=iTemp;
}
void max_heapify(long iKey, long iSize, long aiArray[]){
        long iLeft, iRight, iMax, iTemp;

        iLeft= (iKey << 1) + 1;
        iRight= iLeft +1;
        iMax= ((iLeft<iSize) &&(aiArray[iLeft]> aiArray[iKey])) ? iLeft : iKey;
                if ((iRight<iSize) && (aiArray[iRight]>aiArray[iMax]))
                  iMax = iRight;
                if (iMax!= iKey) {
                                swap(&aiArray[iKey], &aiArray[iMax]);
                                max_heapify(iMax, iSize, aiArray);
                }
        }

        void build_amx_heap (long iSize, long aiArraY[]){
                 int i;

                                for (i=iSize/2 - 1; i>=0; i--)
                                max_heapify(i, iSize, aiArray);
        }

         void heapsort(long iSize, long aiArray[]){
                        long i;
                        build_max_heap(iSize, aiArray);
                        for (i=iSize-1; i>0; i--){
                swap(&aiArray[0], &aiArray[i]);
                max_heapify(0, i-1, aiArray);
                        }
         }

int main(void){
        scanf("%ld", &iN);
        for (i=0, iHalf = iN/2; i< iHalf +1; i++)
                        scanf("ld", &aiData[i]);
                        build_max_heap(iHalf+1, aiData);
                        for (i=iHalf+1; i< iN; i++){
                                        scanf("%ld", &iTemp);
                                        if (iTemp<aiData[0]){
                        aiData[0]= iTemp;
                        max_heapify(0, iHalf+1; aiData);
                                        }
                        }
heapsort(iHalf+1, aiData);
if (odd(iN))
        printf("%ld", aiData[iHalf]);
        else
        printf("%1.11f", (double) aiData[iHalf]+aiData[iHalf-1])/2;
return 0;
}
```

Conclusiones

- El montículo es un gran elemento que se sigue empleando por su simpleza y óptima respuesta, ya que en lugar de emplear estructura de árbol, grafos o listas, emplea un simple vector, por lo que cada acción a realizar sobre él resulta con un coste bastante reducido.

```cpp
//Heap Sort
//Codificado por sAfOrAs
#include <iostream>
#define max 100
using namespace std;


int main()
{
    int A[max],j,item,temp,i,k,n;
    cout<<"Ingresa la cantidad de elementos del arreglo: ";
    cin>>n;
    for(i=1;i<=n;i++)
    cin >> A[i];

    for(k=n;k>0;k--)
    {
        for(i=1;i<=k;i++)
        {
            item=A[i];
            j=i/2;
            while(j>0 && A[j]<item)
            {
                A[i]=A[j];
                i=j;
                j=j/2;
            }
            A[i]=item;
        }
        temp=A[1];
        A[1]=A[k];
        A[k]=temp;
    }
    cout<<"El orden es:"<<endl;
    for(i=1;i<=n;i++)
    cout<<A[i] << endl;
    return 0;
}
```

Ordenamiento de raíz (radix sort).

- Este ordenamiento se basa en los valores de los dígitos reales en las representaciones de posiciones de los números que se ordenan.

- Por ejemplo el número 235 se escribe 2 en la posición de centenas, un 3 en la posición de decenas y un 5 en la posición de unidades.

Reglas para ordenar

- Empezar en el dígito más significativo y avanzar por los dígitos menos significativos mientras coinciden los dígitos correspondientes en los dos números.

- El número con el dígito más grande en la primera posición en la cual los dígitos de los dos números no coinciden es el mayor de los dos (por supuesto sí coinciden todos los dígitos de ambos números, son iguales).

- Este mismo principio se toma para Radix Sort, para visualizar esto mejor tenemos el siguiente ejemplo. En el ejemplo anterior se ordeno de izquierda a derecha. Ahora vamos a ordenar de derecha a izquierda.

Primera Pasada

25 57 48 37 12 92 86 33

1	
2	12 92
3	33
4	
5	25
6	86
7	57 37
8	48
9	
10	

Después de la primera pasada:

12 92 33 25 86 57 37 48

Segunda Pasada

12 92 33 25 86 57 37 48

0
1 12
2 25
3 33 37
4 48
5 57
6
7
8 86
9 92

Archivo ordenado: 12 25 33 37 48 57 86 92

Características

- Debido a que el ciclo for (*k* = 1; *k* <= *m*; *k*++) externo se recorre *m* veces (una para cada dígito) y el ciclo interior *n* veces (una para cada elemento en el archivo) el ordenamiento es de aproximadamente (*m***n*).

- Si las llaves son complejas (es decir, si casi cada número que puede ser una llave lo es en realidad) *m* se aproxima a log *n*, por lo que (*m***n*) se aproxima a (*n*log *n*).

- Si la cantidad de dígitos es grande, en ocasiones es más eficiente ordenar el archivo aplicando primero el ordenamiento de raíz a los dígitos más significativos y después utilizando inserción directa sobre el archivo ordenado.

Ventajas

- El ordenamiento es razonablemente eficiente si el número de dígitos en las llaves no es demasiado grande.

- Si las máquinas tienen la ventaja de ordenar los dígitos (sobre todo si están en binario) lo ejecutarían con mucho mayor rapidez de lo que ejecutan una comparación de dos llaves completas.

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#define NUMELTS 20

void radixsort(int x[], int n)
{
  int front[10], rear[10];
  struct {
    int info;
    int next;
  } node[NUMELTS];
  int exp, first, i, j, k, p, q, y;

  /* Inicializar una lista vinculada */
  for (i = 0; i < n-1; i++) {
    node[i].info = x[i];
    node[i].next = i+1;
  } /* fin del for */
  node[n-1].info = x[n-1];
  node[n-1].next = -1;
  first = 0; /* first es la cabeza de la lista vinculada */
  for (k = 1; k < 5; k++) {
    /* Suponer que tenemos números de cuatro dígitos */
    for (i = 0; i < 10; i++) {
      /*Inicializar colas */
      rear[i] = -1;
      front[i] = -1;
    } /*fin del for */
    /* Procesar cada elemento en la lista */
    while (first != -1) {
      p = first;
      first = node[first].next;
      y = node[p].info;
      /* Extraer el kâsimo dÁgito */
      exp = pow(10, k-1);                    /* elevar 10 a la (k-1)ésima potencia */
      j = (y/exp) % 10;
      /* Insertar y en queue[j] */
      q = rear[j];
      if (q == -1)
                        front[j] = p;
      else
                        node[q].next = p;
      rear[j] = p;
    } /*fin del while */

/* En este punto, cada registro está en su cola basándose en el dígito k
       Ahora formar una lista única de todos los elementos de la cola.
       Encontrar el primer elemento. */
    for (j = 0; j < 10 && front[j] == -1; j++);
      ;
    first = front[j];

    /* Vincular las colas restantes */
    while (j <= 9) {      /* Verificar si se ha terminado */
      /*Encontrar el elemento siguiente */
      for (i = j+1; i < 10 && front[i] == -1; i++);
                        ;
      if (i <= 9) {
                        p = i;
                        node[rear[j]].next = front[i];
      } /* fin del if */
      j = i;
    } /* fin del while */
    node[rear[p]].next = -1;
  } /* fin del for */

  /* Copiar de regreso al archivo original */
  for (i = 0; i < n; i++) {
    x[i] = node[first].info;
    first = node[first].next;
  } /*fin del for */
} /* fin de radixsort*/


int main(void)
{
  int x[50] = {NULL}, i;
  static int n;

  printf("\nCadena de números enteros: \n");
  for (n = 0;; n++)
    if (!scanf("%d", &x[n])) break;
  if (n)
    radixsort (x, n);
  for (i = 0; i < n; i++)
    printf("%d ", x[i]);
  return 0;
}
```

Bibliografía

Libros

- Fundamentos de programación. Algoritmos, estructuras de datos y objetos; Luis Joyanes Aguilar; 2003; Editorial: MCGRAW-HILL. ISBN: 8448136642.

- ALGORITMOS, DATOS Y PROGRAMAS con aplicaciones en c, c++, Pascal, Delphi y Visual Da Vinci. De Guisti. Armando. 2001. editorial: Prentice Hall. ISBN: 987-9460-64-2

Internet

- https://blog.zerial.org/ficheros/Informe_Ordenamiento.pdf
- http://www.tutorialesprogramacionya.com/javaya/detalleconcepto.php?codigo=93&punto=&inicio=