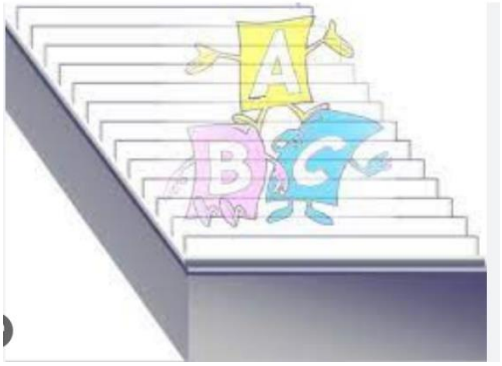
Tema V: Ordenación

Introducción a la ordenación. Clasificación de los algoritmos de ordenación. Métodos Directos. Métodos Logarítmicos. Ordenación por montículos. Ordenación de raíz. Intercalación.

5.1 Introducción a la ordenación

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.

Declaración

Sea A una lista de N elementos:

A1, A2, A3, ..., An

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

Ascendente: A1 <= A2 <= A3 <= A4 .....<= An

Descendente: A1 >= A2 >= A3 >= A4 ..... >=An

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

– Categoría de arreglos

– Categoría de archivos

5.2 Clasificación de los algoritmos de ordenación

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

– Directos de orden O (N2)

– 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.

A. *Ordenación por Intercambio Directo (Burbuja)*
B. *Ordenación por Selección (Obtención sucesivas de menores)*
C. *Ordenación por Inserción (Baraja)*

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

A. *Método de Shell (Inserción con incrementos decrecientes)*
B. *Método de QuickSort (Clasificación Rápida)*
C. *Método del Montículo*

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.

5.2.1 Métodos Directos.

5.2.1.A. Ordenación por Intercambio Directo (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.

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.

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

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 *n2*.

En el siguiente ejemplo, mostramos la forma, en el cual los elementos, se van ubicando en su posición correcta.

```
Inicial 21 35 17 08 14 42 02 26
Pasada1 21 17 08 14 35 02 26 42
Pasada2 17 08 14 21 02 26 35 42
Pasada3 08 14 17 02 21 26 35 42
Pasada4 08 14 02 17 21 26 35 42
Pasada5 08 02 14 17 21 26 35 42
Pasada6 02 08 14 17 21 26 35 42
```

Algoritmo de resolución

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
```

```
#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;
}
```

5.2.1.B Ordenación por Selección (Obtención sucesivas de menores)

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 más 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.**

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).

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.

En el segundo recorrido se averigua el menor entre los restantes elementos y se lo intercambia con el que está en la segunda posición

El resto de los recorridos utilizará la misma lógica.

{dibujo del algoritmo }

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:

$$C = \mathbf{n2 – n / 2}$$

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

Veamos la resolución algorítmica y en programación C

```
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
```

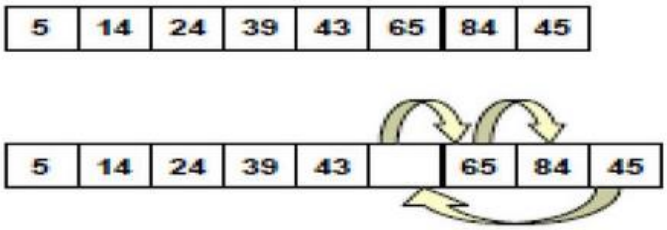
```
#include <stdio.h>
int main() {
      int array [100], n, i , d, pos, swap;
      printf("cuantos elementos deseas ordenar?\n");
      scanf("%d", &n);
printf("Introduce los %d numeros \n", n);
      for (i=0; i < n; i++)
```

```
      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;
}
```

5.2.1.C Ordenación por Inserción (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).



El método de inserción directa es el que generalmente utilizan los jugadores de cartas cuando ordenan éstas, de ahí que también 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.

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 más 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:

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

Veamos la resolución algorítmica y en programación C

```
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
```

```
#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();
}
```

Analisis de eficiencia

Ordenación por selección

N2/2 comparaciones y N intercambios.

Ordenación por inserción

N2/4 comparaciones y N2/8 intercambios en media

El doble en el peor caso.

Casi lineal para conjuntos casi ordenados.

Ordenación por burbuja

N2/2 comparaciones y N2/2 intercambios en media (y en el peor caso).

Lineal en su versión mejorada si el vector esta ordenado

5.2.2. Métodos logarítmicos

5.2.2.A.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 más 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 más rápidamente.

Consiste en:

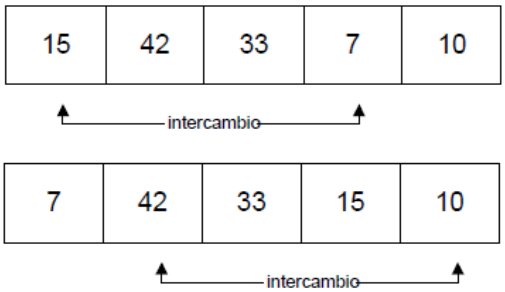
1. Se determina el salto (S) entre los elementos a comparar. Es la parte entera de sumar 1 a la longitud del vector, y dividirlo entre dos.
2. Se recorre comparando el primer elemento con el que está situado “S” posiciones más adelante. Si los elementos comparados están en orden incorrectos, se intercambian; caso contrario quedan donde están.
3. Luego se comparan los dos elementos siguientes y así sucesivamente.
4. Este proceso se repite hasta que uno de los elementos de la comparación sea situado en la última posición del vector.
5. Si durante el recorrido hubo algún intercambio, se vuelve a repetir el proceso con el mismo salto (hasta que no se registren intercambios).
6. Luego se modifica el salto: será la parte entera del resultado de sumar 1 al salto anterior, y dividirlo entre dos.
7. El proceso de ordenación finaliza cuando, siendo el salto (S) sea igual a 1, se haya realizado una pasada sin intercambios.

Un ejemplo practico

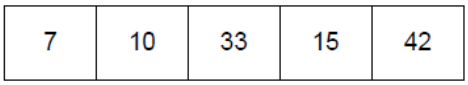
V =

15	42	33	7	10

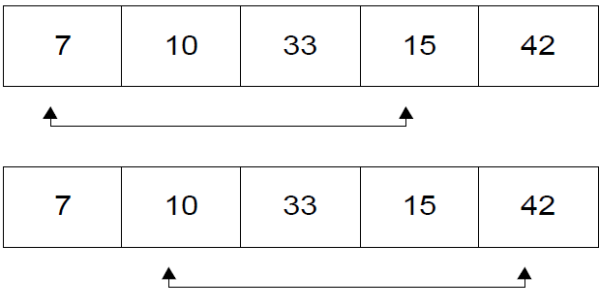
Se calcula el salto: *Ent(5+1)/2 = 3*



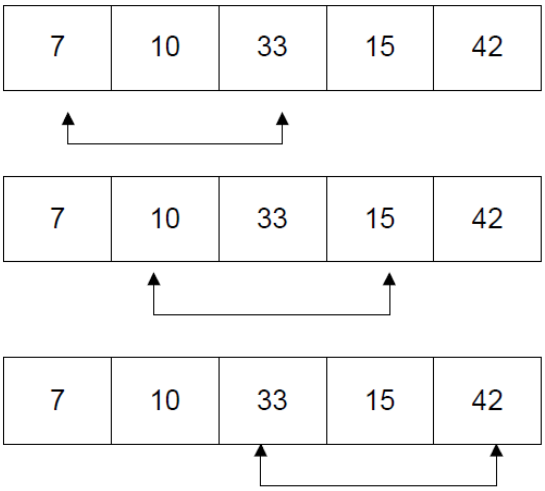
Al final del recorrido queda:



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

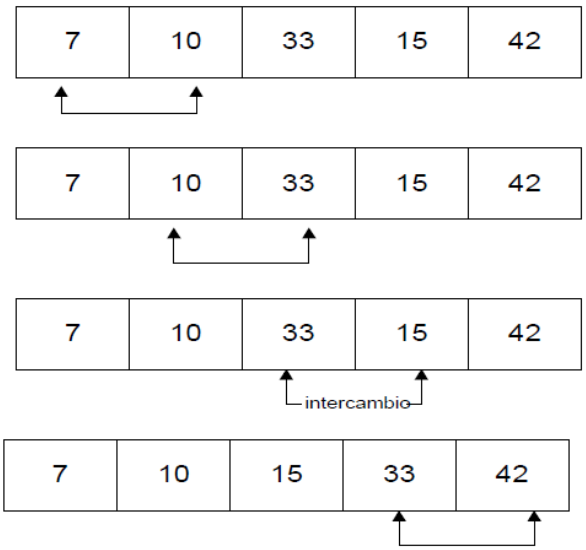


Se calcula el salto: *Ent(3+1)/2 = 2*

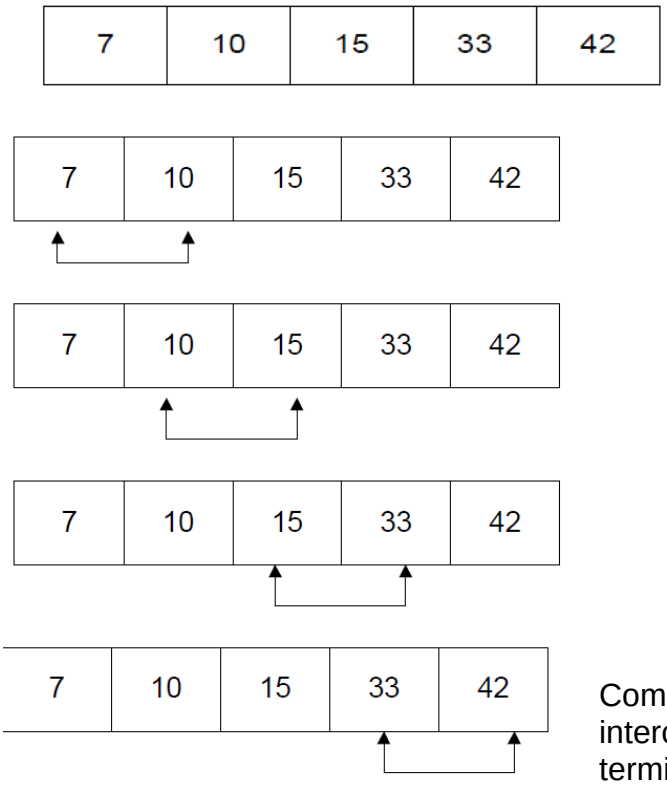


En el recorrido anterior no se realizó ningún intercambio

Se calcula el salto de nuevo: *Ent(2+1)/2 = 1*



Al final del recorrido quedaría:



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

```
#include<stdio.h>
int a[5];
int n=5;
voidmain() {
      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]);
      }
}
```

5.2.2.B. Método de QuickSort (Clasificación Rápida)

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.

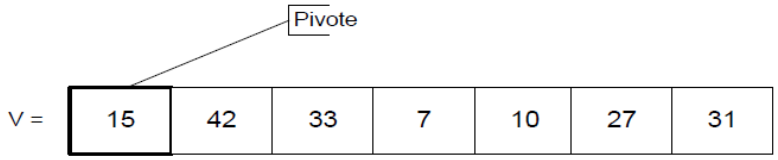
La idea central de este algoritmo consiste en lo siguiente:

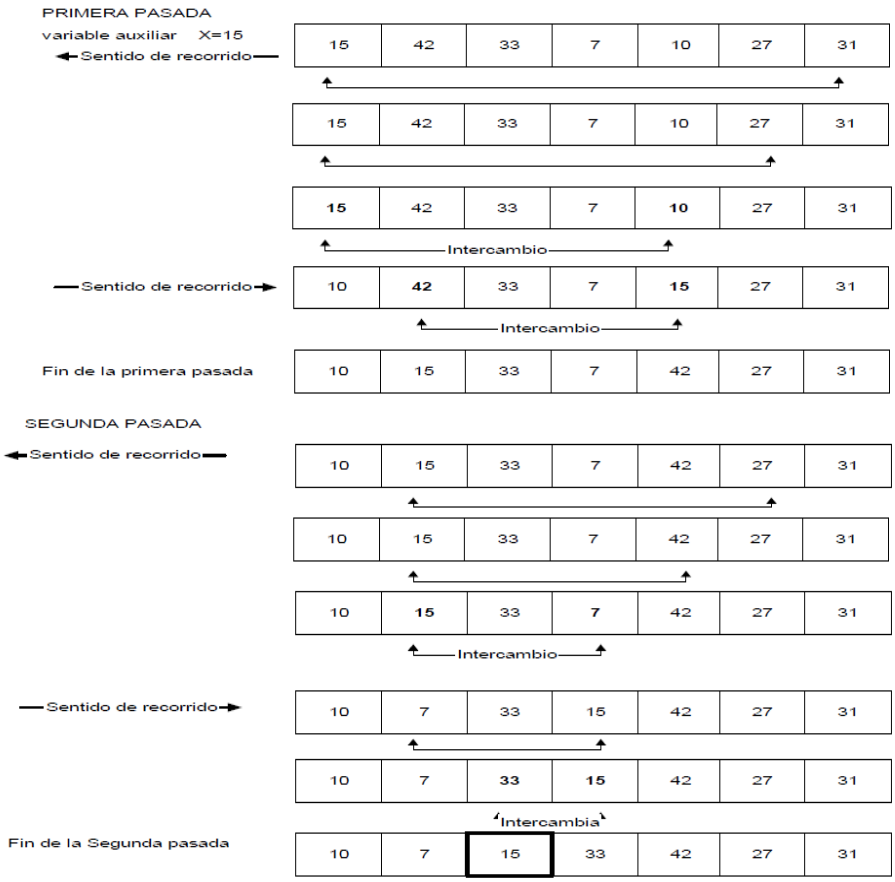
- Se toma un elemento X llamado pivote (valor elegido en forma arbitraria)
- Se trata de ubicar a X en la posición correcta del arreglo, de forma tal que todos los elementos que se encuentren a su izquierda sean menores o iguales a X y a su vez todos los elementos que se encuentren a la derecha sean mayores o iguales a X.

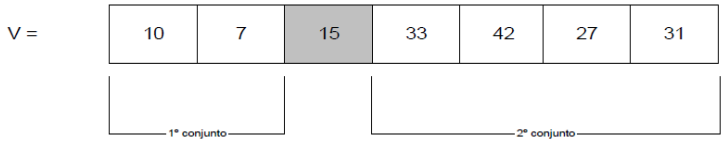
- Estos dos pasos se repiten pero ahora para los conjuntos de datos que se encuentran a la izquierda y a la derecha de la posición correcta X.
- El algoritmo termina cuando todos los elementos se encuentran en su posición correcta del arreglo.

Consiste en:

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.

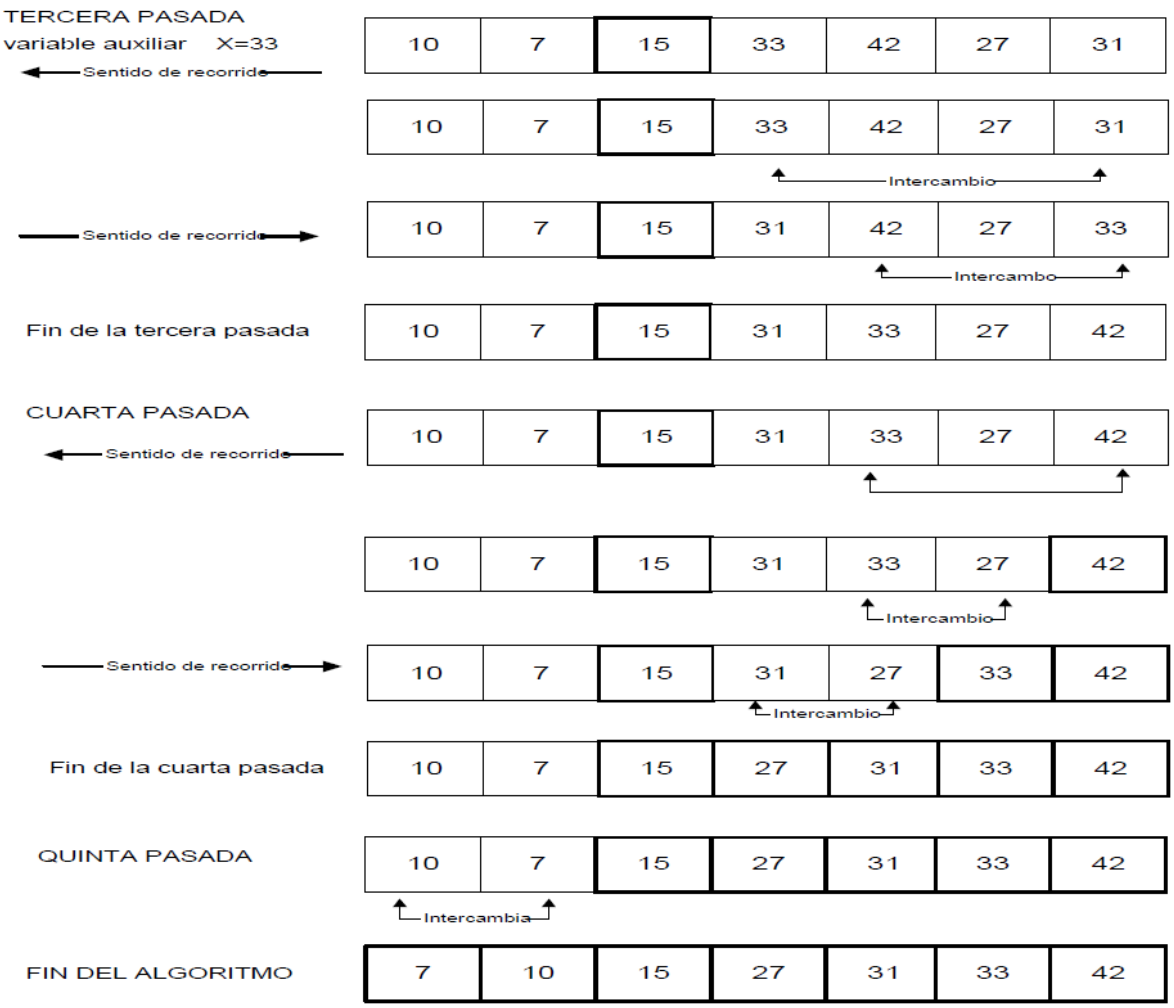






Como el recorrido de izquierda a derecha debería iniciarse en la misma posición donde se encuentra el elemento X, el proceso termina ya que dicho elemento se encuentra en su posición correcta.

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 más elementos. El proceso a repetir se realiza en forma iterativa o recursiva.



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*.

```
#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*/
```

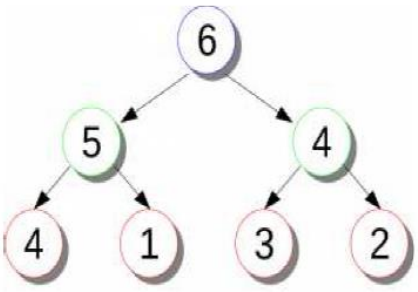
```
printf("Ingrese valores para el arreglo:\n");
for (i = 0; i < tam; i++)
     scanf("%d", &arreglo[i]);
     printf("\n");

/*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");
}
```

5.3 Método del Montículo

La ordenación por montículos **utiliza la propiedad de la raíz para ordenar el arreglo**. Una vez que el arreglo cumpla las propiedades del montículo, quitamos la raíz y la colocamos al final del arreglo. Con los datos que sobran, creamos otro montículo y repetimos hasta que todos los datos estén ordenados.

¿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*.
En computación, un montículo (o *heap* en inglés) es una **estructura de datos del tipo árbol con información perteneciente a un conjunto ordenado**.



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 fórmula 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.

El método de ***Heapsort*** es conocido con el nombre de montículo en el mundo de habla hispana. Es el método más eficiente de los métodos de ordenación que trabaja con árboles.

La idea central de este algoritmo consiste en:

1. Construir un montículo
2. Eliminar la raíz del montículo en forma repetida.

Para representar un montículo en un arreglo lineal:

1. El nodo *K* se almacena en la posición *K* correspondiente del arreglo.
2. El hijo izquierdo del nodo *k* se almacena en la posición *2*k.*
3. El hijo derecho del nodo *k* se almacena en la posición *2*k+1*.

La estructura de datos Montículo es un arreglo de objetos que puede ser visto como un árbol binario con raíz, cuyos nodos pertenecen a un conjunto totalmente ordenado, y tal que cumple las siguientes dos propiedades:

a) Propiedad de orden: La raíz de cada subárbol es mayor o igual que cualquiera de sus nodos restantes.

b) Propiedad de forma: La longitud de toda rama es h o h−1, donde “h” es la altura del árbol. Además, si una rama termina en una hoja a la derecha de otra hoja, ésta última de altura h−1, la primera debe ser de altura h−1.

Como trabaja el método

Se toman las mejores características de los dos algoritmos de ordenamiento basados en comparación (MergeSort e InsertionSort) para crear un nuevo algoritmo llamado HeapSort.

Este algoritmo está basado en la estructura de montículo.

La idea central de este algoritmo consiste en lo siguiente:

- Construir un montículo.
- Eliminar la raíz del montículo en forma repetida.

El método de ordenación se puede describir con los siguientes pasos:

1. Construir un montículo inicial con todos los elementos del vector A [1], A [2], ., A[n]
2. Intercambiar los valores de A [1] y A [n] (siempre queda el máximo en el extremo)
3. Reconstruir el montículo con los elementos A [1], A [2],……, A [n-1]
4. Intercambiar los valores de A [1] y A [n-1]
5. Reconstruir el montículo con los elementos A [1], A [2],……, A [n-2]

Este es un proceso iterativo que partiendo de un montículo inicial, repite intercambiar los extremos, decrementar en 1 la posición del extremo superior y reconstruir el montículo del nuevo vector.

Lo expresamos en forma algorítmica así:

Ordenación Heapsort (Vector, N)
Debe construirse un montículo inicial (Vector, N)
desde k = N hasta 2 hacer
    intercambio (Vector [1], Vector [k])
    construir montículo (Vector, 1, k-1)
fin desde

Entonces de acuerdo al algoritmo debemos considerar dos problemas:

- Construir el montículo inicial.
- Cómo restablecer los montículos intermedios

Para restablecer el montículo hay dos posibilidades:

- A [1] es menor o igual que los valores de sus hijos, entonces la propiedad del montículo no se ha roto.
- En otro caso, el elemento mínimo que necesitamos poner en A [1] es o su hijo izquierdo o su hijo derecho (A [2], A [3] respectivamente): por lo que se determina el menor de sus hijos y éste se intercambia con A [1]. El proceso continúa repitiendo las mismas comparaciones entre el nodo intercambiado y sus nodos hijos; así hasta llegar a un nodo en el que no se viole la propiedad de montículo, o estemos en un nodo hoja.

Procedimiento Heapify.
Heapify es una importante subrutina para manipular montículos.

```
{Pre: subárbol de raíz Izq (i) y subárbol de raíz Der (i) son
montículos}.
{Post: subárbol de raíz i es un montículo}.

Heapify (A, i)
izq = Izq ( i ) // La función Izq ( i ) = 2 * i
der = Der ( i ) // La función Der ( i ) = 2 * i + 1
if izq ≤ heap-size [A] and A [izq] > A [ i ] then
     pos-max = izq
else
     pos-max = i
if der ≤ heap-size [A] and A [der] > A [pos-max] then
     pos-max = der
if pos-max ≠ i then
     intercambiar (A [ i ], A [pos-max])
     Heapify (A, pos-max)
```

Construcción de un Montículo.

Utilizando Heapify:

```
Build-Heap (A)
Heap-size [A] = length [A]
      for I = [length [A]/2] down to 1
            do Heapify (A, I)
```

Algoritmo HeapSort.

HeapSort (A)

1. Build-Heap (A)
2. For j = length [A] down to 2
3. Do intercambio (A[1], A[j])
4. Heap-size [A] = heap-size [A]-1
5. Heapify (A,1)

Eficiencia

1. Construcción inicial del heap: n operaciones de inserción O(log n) sobre un árbol parcialmente ordenado ⇒ **O(n log n).**
2. Extracción del menor/mayor elemento del heap: n operaciones de borrado O(log n) sobre un árbol parcialmente ordenado ⇒**O(n log n).**

Podemos decir que es un algoritmo de ordenación no recursivo, no estable, con complejidad computacional **O(n log n).**

Es iterativo no recursivo.

Ventajas y Desventajas

Es usar este método de Ordenamiento trae consigo diversas ventajas y desventajas con respecto a los otros métodos, dichas características están en la tabla a continuación:

Ventajas	Desventajas
Funciona efectivamente con datos desordenados. Su desempeño es en promedio tan bueno como el Quicksort. No utiliza memoria adicional.	No es estable. Método complejo.

Código en C

```
#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;
}
```

5.4 Ordenación de raíz.

Este método puede considerarse como una generalización de la clasificación por urnas. Aprovecha la estrategia de la forma más antigua de clasificación manual, consistente en hacer diversos montones de fichas, cada uno caracterizado por tener sus componentes un mismo dígito (letra, si es alfabética) en la misma posición; estos montones se recogen en orden ascendente y se reparte de nuevo en montones según el siguiente dígito de la clave.

Como ejemplo, suponer que se han de ordenar estas fichas identificadas por tres dígitos:

345, 721, 425, 572, 836, 467, 672,194, 365, 236, 891, 746, 431, 834, 247, 529, 216, 389

Atendiendo al dígito de menor peso (unidades) las fichas se distribuyen en montones del 0 al 9;

	431 891 721	672 572		834 194	365 425 345	216 746 236 836	247 467		389 529
0	1	2	3	4	5	6	7	8	9

Recogiendo los *montones* en orden, la secuencia de fichas queda:

721, 891, 431, 572, 672, 194, 834, 345, 425, 365, 836, 236, 746, 216, 467, 247, 529, 389

De esta secuencia podemos decir que está ordenada respecto al dígito de menor peso, respecto a las unidades. Pues bien, ahora de nuevo se distribuye la secuencia de fichas en montones respecto al segundo dígito:

			236						
		529	836	247					194
		425	834	746		467	672		891
	216	721	431	345		365	572	389	
0	1	2	3	4	5	6	7	8	9

Recogiendo de nuevo los montones en orden, la secuencia de fichas queda:

216, 721, 425, 529, 431, 834, 836, 236, 345, 746, 247, 365, 467, 572, 672, 389, 891, 194

En este momento esta secuencia de fichas ya están ordenadas respecto a los dos últimos dígitos, es decir, respecto a las decenas. Por último, se distribuye las fichas en montones respecto al tercer dígito:

		247	389	467				891	
		236	365	431	572		746	836	
	194	216	345	425	529	672	721	834	
0	1	2	3	4	5	6	7	8	9

Recogiendo de nuevo los *montones* en orden, la secuencia de fichas queda ya ordenada:

194, 216, 236, 247, 345, 365, 389, 425, 431, 467, 529, 572, 672, 721, 746, 834, 836, 891

La idea clave de la ordenación Radixsort (también llamada por residuos) es clasificar por urnas primero respecto al dígito de menor peso (menos significativo) d_k, después concatenar las urnas, clasificar de nuevo respecto al siguiente dígito d_{k-1}, y así sucesivamente se sigue con el siguiente dígito hasta alcanzar el dígito más significativo d_l, en ese momento la secuencia estará ordenada. La concatenación de las urnas consiste en enlazar el final de una con el frente de la siguiente.

Las urnas se representan mediante un vector de listas. En el caso de que la clave respecto a la que se ordena sea un entero, se tendrán 10 urnas, numeradas de 0 a 9. Si la clave respecto a la que se ordena es alfabética, habrá tantas urnas como letras distintas, desde la urna que represente a la letra a hasta la z.

Para el caso de que clave sea entera, en primer lugar se determina el máximo número de dígitos que puede tener la clave. En un bucle de tantas iteraciones como máximo de dígitos se realizan las acciones de distribuir por urnas los registros, concatenar...

La distribución por urnas exige obtener el dígito del campo clave que se encuentra en la posición definida por el bucle externo, dicho dígito será el índice de la urna.

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.

Algoritmo de resolución

```
OrdenacionRadixsort(vector, n)
Inicio

< cálculo el número máximo de dígitos: ndig >
peso = 1 { permite obtener los dígitos de menor a mayor peso}
desde i = 1 hasta ndig hacer
      CrearUrnas(Urnas);
      desde j = 1 hasta n hacer
            d = (vector[j] / peso) modulo 10;
            AñadirEnUma(Urnas[d], vector[j]);
      fin_desde

< búsqueda de primera urna no vacía: j >
      desde r = j+1 hasta M hace { M: número de urnas }
            EnlazarUma(Urnas[r], Urnas[j]);
      fin_desde
{Se recorre la lista-urna resultante de la concatenación}
      r = 1;
      dir = frente(Urna[j]);
      mientras dir <> nulo hacer
            vector[r] = dir.registro;
            r = r+1;
            dir = siguiente(dir)
      end
      peso = peso * 10;
fin_desde
fin_ordenacion
```

5.5 Intercalación.

Desarrollaremos un método de intercalación de dos arreglos ordenados, los cuales se van a combinar para producir un único arreglo, también ordenado. Este proceso se realiza seleccionando sucesivamente los elementos con el mínimo valor de cada uno de los dos arreglos, situándolos en un nuevo arreglo. De esta forma el nuevo arreglo tiene todos sus elementos ordenados.

Algoritmo

Variable	Descripción
A	Arreglo lineal de dimensión n cuyos elementos están ordenados en forma ascendente.
B	Arreglo lineal de dimensión m cuyos elementos están ordenados en forma ascendente.
C	Arreglo lineal de dimensión n+m cuyos elementos son los de A y B Ordenados.
N	Variable de tipo entero cuyo valor es la dimensión del arreglo A.
M	Variable de tipo entero cuyo valor es la dimensión del arreglo B.
I,J,K,L	Variables de tipo entero que se usan como índices de los arreglos.

```
algoritmo "Intercalación"
comenzar
leer A, B
I ←1 /* Indice de A */
J ←1 /* Indice de B*/
K ←1 /* Indice de C*/

/* Se comparan los elementos correspondientes y se determina el más
pequeño*/
mientras I <= N y J <= M hacer
      si A(I) <= B(J) entonces
            C(K) ←  A(I)
            I ←I+1
            K ←K+1
      sino
            C(K) ←B(J)
            J ←J+1
            K ←K+1
      finsi
finmientras

/* Se copian los elementos restantes, que no han sido procesados en el
arreglo de salida */
si (I > N) entonces
      para L desde J hasta M hacer
            C(K) ←B(L)
            K ←K+1
      finpara
sino
      para L desde I hasta N hacer
            C(K) ←A(L)
            K ←K+1
      finpara
finsi
escribir C
```

```
fin
```