[Seleccione la fecha]

****

***Métodos de Ordenamiento***

|  |  |
| --- | --- |
| **Integrantes:**  Aguirre Joana  Puyo Belén  Santillán Camila | **ALGORITMOS Y ESTRUCTURAS DE DATOS** |

**INDICE**

|  |  |
| --- | --- |
| **1. ¿Qué es el Análisis Algorítmico?** | Pág. 3 |
| **2.Definir Orden de un Algoritmo.** | Pág.3 |
| **3. Métodos de ordenamiento:**  **3. a.** Intercambio o Método Burbuja Mejorado  **3.b.** Inserción o método de la baraja  **3.c**. Selección o método sencillo  **3.d.** Rápido o QuickSort  **3.e.** Por Mezcla o MergeSort | Pág.4 |
| **4.Diferencia entre los diferentes métodos.** | Pág.16 |
| **5.Bibliografia** | Pág.17 |

1. **¿Qué es el Análisis Algorítmico?**

Para comenzar, es importante recordar la definición de algoritmo, la cual refiere a un grupo finito de operaciones organizadas de manera lógica y ordenada que permite solucionar un determinado [problema](https://definicion.de/problema/). Se trata de una serie de instrucciones o reglas establecidas que, por medio de una sucesión de pasos, permiten arribar a un resultado o solución.En tal sentido, el análisis algorítmico ocupa un lugar fundamental en el proceso, pues tiene como objetivo principal establecer propiedades sobre la eficiencia del algoritmo, permitiendo la comparación entre soluciones alternativas y prediciendo los recursos que usará un algoritmo en la resolución del problema, contribuyendo así a optar por la opción más conveniente.  
Algunos de los objetivos más importantes del Análisis de Algoritmos son:   
-Profundizar en la aplicación de técnicas para el análisis de la eficiencia en tiempo y espacio de un algoritmo.   
- Aplicar técnicas para el análisis de eficiencia de algoritmos recursivos.   
- Aplicar una técnica de diseño de algoritmos adecuada al problema a resolver.   
- Establecer las bases teóricas que permitan comprender una tipificación de problemas clásicos en las ciencias de la computación y la ingeniería informática.   
-Conocer acerca de la existencia de problemas intratables e insolubles.   
La técnica que se utilizaba en los primeros años de la programación para comparar la eficiencia de distintos algoritmos consistía en ejecutarlos para datos diferentes y medir el tiempo consumido.   
Dado que las máquinas y los lenguajes eran dispares, y que el tiempo de ejecución depende no solo del tamaño sino también del contenido de los datos, resultaba muy difícil comparar tales resultados.   
El primer estudio serio sobre la eficiencia de los algoritmos se lo debemos a Daniel Goldenberg del MIT. En 1952 realizó un análisis matemático del número de comparaciones necesarias, en el mejor y en el peor caso, de cinco algoritmos distintos de ordenación. Además, la tesis doctoral de Howard B. Demuth de la Universidad de Stanford estableció en 1956 las bases de lo que hoy llamamos análisis de la complejidad de los algoritmos.

1. **Definición de Orden de un Algoritmo**

El orden de un algoritmo, o método de ordenamiento empleado en su realización, es fundamental en determinar la efectividad de este. La ordenación (clasificación) es la operación de organizar un conjunto de datos en algún orden dado, tal como creciente o decreciente en datos numéricos, o bien en orden alfabético directo o inverso. Operaciones típicas de ordenación son: lista de números, archivos de clientes de banco, nombres de una agenda telefónica, etc.

La clasificación es una operación tan frecuente en programas de computadora, que una gran cantidad de algoritmos se han diseñado para clasificar listas de elementos con eficacia y rapidez. La elección de un determinado algoritmo depende del tamaño del vector o array (arreglo) a clasificar, el tipo de datos y la cantidad de memoria disponible. Los métodos de ordenación se dividen en dos categorías:

• Ordenación de vectores, tablas (arrays o arreglos).

• Ordenación de archivos.

La ordenación de arrays se denomina también ordenación interna, ya que se almacena en la memoria interna de la computadora de gran velocidad y acceso aleatorio. La ordenación de archivos se suele hacer casi siempre sobre soportes de almacenamiento externo, discos, cintas, etc., y, por ello, se denomina también ordenación externa. Estos dispositivos son más lentos en las operaciones de entrada/salida, pero, por el contrario, pueden contener mayor cantidad de información.

* Ordenación interna: clasificación de los valores de un vector según un orden en memoria central: rápida.
* Ordenación externa: clasificación de los registros de un archivo situado en un soporte externo: menos rápido.

EJEMPLO Clasificación en orden ascendente del vector.

7, 3, 2, 1, 9, 6, 7, 5, 4

se obtendrá el nuevo vector

1, 2, 3, 4, 5, 6, 7, 7, 9

Los métodos de clasificación directa más populares son:

* Intercambio.
* Selección.
* Inserción.

Otros:

* Quicksort
* Mergesort

1. **Métodos de ordenamiento:**

3.a. **Intercambio o Método Burbuja Mejorado**

Este método se basa en el algoritmo de clasificación de intercambio o de la burbuja. Es un sencillo algoritmo de ordenamiento que se basa en el principio de comparar pares de elementos adyacentes e intercambiarlos entre sí hasta que estén todos ordenados. Es necesario revisar varias veces toda la lista hasta que no se necesiten más intercambios. Este algoritmo obtiene su nombre de la forma con la que suben por la lista los elementos durante los intercambios, como si fueran pequeñas burbujas. Dado que solo usa comparaciones para operar elementos, se lo considera un algoritmo de comparación, siendo el más sencillo de implementar.



Como ya sabemos mediante el método burbuja, dado un arreglo de n números, se requiere de n-1 pasos para dejar el arreglo ordenado.

Se puede observar que en el primer paso el primer elemento mayor queda en la primera posición mayor (última si es que estamos ordenando de menor a mayor); en el segundo paso el segundo elemento mayor queda en la segunda posición mayor (penúltima); y así sucesivamente. Por esta razón el número de comparaciones debería irse reduciendo en uno, en cada paso.

Se puede observar, además, que en muchos casos se consigue tener ordenado el arreglo, en un número menor de pasos a n-1, por lo cual el resto de los pasos serían innecesarios.

Considerando estos dos hechos se podría mejorar el método burbuja eliminando los pasos innecesarios y reduciendo las comparaciones con cada paso.

Una manera sencilla de hacer esto sería detectando mediante algún registro si se han efectuado cambios o no, y reduciendo el número de comparaciones en cada paso.

A continuación, se muestra un fragmento de código que permite realizar estas mejoras:

float X[100], AUX;

int N, paso, j;

int bandera=1;

.

.

.

for(paso=0;paso<N-1&&bandera==1;paso++)

/\* si en el paso anterior no hubo cambios se detiene ciclo \*/

{

bandera=0;

for(j=0;j<N-paso-1;j++)

/\* las comparaciones van dismuyendo

a medida que se efectuan los pasos \*/

{

if(X[j]<X[j+1])

{

bandera=1; /\* indica si se han realizados cambios o no \*/

AUX=X[j];

X[j]=X[j+1];

X[j+1]=AUX;

}

}

}

3.b. **Inserción o Método de la Baraja**

El ordenamiento por inserción es una manera muy natural de ordenar para un ser humano, y puede usarse fácilmente para ordenar un mazo de cartas numeradas en forma arbitraria.

La idea de este algoritmo de ordenación consiste en ir insertando un elemento de la lista o un arreglo en la parte ordenada de la misma, asumiendo que el primer elemento es la parte ordenada, el algoritmo ira comparando un elemento de la parte desordenada de la lista con los elementos de la parte ordenada, insertando el elemento en la posición correcta dentro de la parte ordenada, y así sucesivamente hasta obtener la lista ordenada.

Así, por ejemplo, suponga que tiene la lista desordenada: 5 14 24 39 43 65 84 45

Para insertar el elemento 45, habrá que insertarlo entre 43 y 65, lo que supone desplazar a la derecha todos aquellos números de valor superior a 45, es decir, saltar sobre 65 y 84. Se obtiene: 5 14 24 39 43 65 84 45.

A continuación, la codificación de este método:

void Insercion(int numbers[], int array\_size) {

int i, a, index;

for (i=1; i < array\_size; i++) {

index = numbers[i];

a = i-1;

while (a >= 0 && numbers[a] > index) {

numbers [a + 1] = numbers[a];

a--;

}

numbers[a+1] = index;

}

}

**3.c. Selección o Método Sencillo**

Es un algoritmo sencillo que consiste en lo siguiente:

* Buscar el elemento más pequeño de la lista.
* Intercambiarlo con el elemento ubicado en la primera posición de la lista.
* Buscar el segundo elemento más pequeño de la lista.
* Lo intercambias con el elemento que ocupa la segunda posición en la lista.
* Se repite el proceso hasta que hayas ordenado toda la lista.

La codificación de este método es:

int minimo=0;

for(i=0 ; i<n-1 ; i++)

{

   minimo=i;

   for(j=i+1 ; j<n ; j++)

   {

      if (x[minimo] > x[j]) minimo=j;

   }

   temp=x[minimo];

   x[minimo]=x[i];

   x[i]=temp;

}

Por ejemplo, vamos a ordenar la siguiente lista: 4 - 3 - 5 - 2 - 1

Comenzamos buscando el elemento menor entre la primera y última posición. Es el 1. Lo intercambiamos con el 4 y la lista queda así: 1 - 3 - 5 - 2 - 4

Ahora buscamos el menor elemento entre la segunda y la última posición. Es el 2. Lo intercambiamos con el elemento en la segunda posición, es decir el 3. La lista queda así: 1 - 2 - 5 - 3 - 4

Buscamos el menor elemento entre la tercera posición y la última. Es el 3, que intercambiamos con el 5: 1 - 2 - 3 - 5 - 4

El menor elemento entre la cuarta y quinta posición es el 4, que intercambiamos con el 5. Concluyendo la lista ordenada: 1 - 2 - 3 - 4 - 5

**Análisis del algoritmo.**

[Estabilidad](http://c.conclase.net/orden/?cap=introduccion#estabilidad): no es estable.

[Requerimientos de Memoria](http://c.conclase.net/orden/?cap=introduccion#requerimientos): Al igual que el [ordenamiento burbuja](http://c.conclase.net/orden/?cap=burbuja#analisis), este algoritmo sólo necesita una variable adicional para realizar los intercambios.

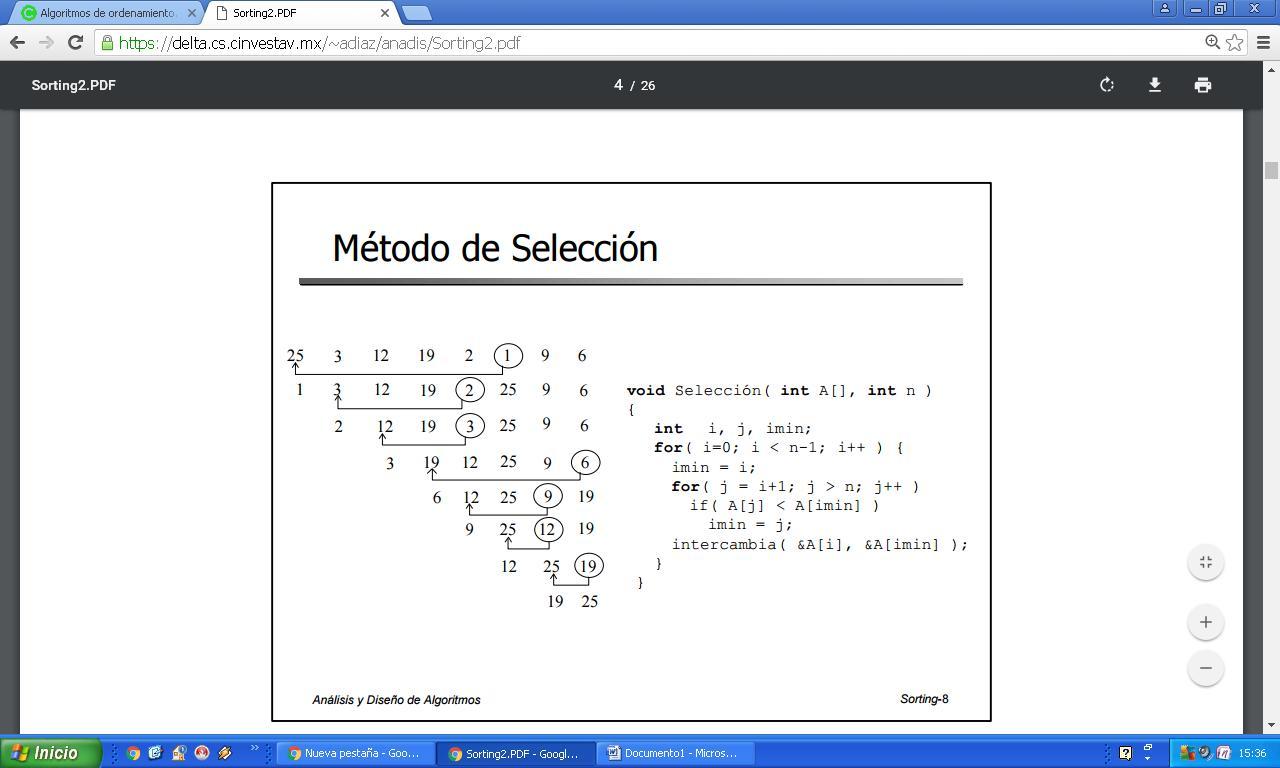
[Tiempo de Ejecución](http://c.conclase.net/orden/?cap=introduccion#tiempo_ejecucion): El ciclo externo se ejecuta n veces para una lista de n elementos. Cada búsqueda requiere comparar todos los elementos no clasificados. Luego la complejidad es O(n2). Este algoritmo presenta un comportamiento constante independiente del orden de los datos. Luego la complejidad promedio es también O(n2).

Ventajas:

* Fácil implementación.
* No requiere memoria adicional.
* Realiza pocos intercambios.
* Rendimiento constante: poca diferencia entre el peor y el mejor caso.

Desventajas:

* Lento.
* Realiza numerosas comparaciones.
* Este es un algoritmo lento. No obstante, ya que sólo realiza un intercambio en cada ejecución del ciclo externo, puede ser una buena opción para listas con [registros](http://c.conclase.net/orden/?cap=introduccion#registro) grandes y [claves](http://c.conclase.net/orden/?cap=introduccion#clave) pequeñas. Si miras el [programa de demostración](http://c.conclase.net/orden/?cap=introduccion#codigo) notarás que es el más rápido en la parte gráfica. La razón es que es mucho más lento dibujar las barras que comparar sus largos (el desplazamiento es más costoso que la comparación), por lo que en este caso especial puede vencer a algoritmos como Quicksort.



**3.d. Método Rápido o QuickSort**

El método de ordenamiento Quick Sort es actualmente el más eficiente y veloz de los métodos de ordenación interna. Es una mejora del método de ordenación por intercambio directo y es también conocido con el nombre del método rápido y de ordenamiento por partición, en el mundo de habla hispana.

Este método es una mejora sustancial del método de intercambio directo y recibe el nombre de Quick Sort por la velocidad con que ordena los elementos del arreglo. Su autor C.A. Hoare lo bautizó así.

El método se basa en la estrategia típica de “divide y vencerás” (divide and conquer):

1. Se elige un elemento x al azar (llamado *pivote* en ordenación rápida).
2. La lista a clasificar almacenada en un vector o array se divide (parte) en dos sublistas: una con todos los valores menores o iguales a un cierto valor específico y otra con todos los valores mayores que ese valor.
3. Se busca por la izquierda del arreglo un elemento mayor que x.
4. Se busca por la derecha un elemento menor o igual que x.
5. Si no se han cruzado los índices izquierda y derecha intercambiar el elemento mayor.
6. Se repite el proceso desde el paso 2 mientras no se crucen los índices izquierdos y derechos.
7. Se intercambia x con el elemento de la derecha.
8. En este momento el elemento x divide el arreglo en dos subarreglos tales que el subarreglo izquierdo contiene los elementos menores o iguales que x y el subarreglo derecho mayores que x, por lo tanto, x esta ordenado.
9. Se efectua quicksort con el arreglo izquierdo.
10. Se hace quicksort con el arreglo derecho.

La codificación del método es:

#include<stdio.h>

void quicksort(int number[25],int first,int last){

int i, j, pivot, temp;

if(first<last){

pivot=first;

i=first;

j=last;

while(i<j){

while(number[i]<=number[pivot]&&i<last)

i++;

while(number[j]>number[pivot])

j--;

if(i<j){

temp=number[i];

number[i]=number[j];

number[j]=temp;

}

}

temp=number[pivot];

number[pivot]=number[j];

number[j]=temp;

quicksort(number,first,j-1);

quicksort(number,j+1,last);

}

}

int main(){

int i, count, number[25];

printf("How many elements are u going to enter?: ");

scanf("%d",&count);

printf("Enter %d elements: ", count);

for(i=0;i<count;i++)

scanf("%d",&number[i]);

quicksort(number,0,count-1);

printf("Order of Sorted elements: ");

for(i=0;i<count;i++)

printf(" %d",number[i]);

return 0;

}

**Ejemplo de implementación:**

Ingresamos un listado de números: 5 - 3 - 7 - 6 - 2 - 1 - 4

Comenzamos con la lista completa. El elemento divisor será el 4:

5 - 3 - 7 - 6 - 2 - 1 - 4

Comparamos con el 5 por la izquierda y el 1 por la derecha.

5 - 3 - 7 - 6 - 2 - 1 - 4

5 es mayor que cuatro y 1 es menor. Intercambiamos:

1 - 3 - 7 - 6 - 2 - 5 - 4

Avanzamos por la izquierda y la derecha:

1 - 3 - 7 - 6 - 2 - 5 - 4

3 es menor que 4: avanzamos por la izquierda. 2 es menor que 4: nos mantenemos ahí.

1 - 3 - 7 - 6 - 2 - 5 - 4

7 es mayor que 4 y 2 es menor: intercambiamos.

1 - 3 - 2 - 6 - 7 - 5 - 4

Avanzamos por ambos lados:

1 - 3 - 2 - 6 - 7 - 5 - 4

En este momento termina el ciclo principal, porque los índices se cruzaron. Ahora intercambiamos lista[i] con lista[sup] (pasos 16-18):

1 - 3 - 2 - 4 - 7 - 5 - 6

Aplicamos recursivamente a la sublista de la izquierda (índices 0 - 2). Tenemos lo siguiente:

1 - 3 - 2

1 es menor que 2: avanzamos por la izquierda. 3 es mayor: avanzamos por la derecha. Como se intercambiaron los índices termina el ciclo. Se intercambia lista[i] con lista[sup]:

1 - 2 - 3

Al llamar recursivamente para cada nueva sublista (lista[0]-lista[0] y lista[2]-lista[2]) se retorna sin hacer cambios (condición 5.).Para resumir te muestro cómo va quedando la lista:

Segunda sublista: lista[4]-lista[6]

7 - 5 - 6

5 - 7 - 6

5 - 6 - 7

Para cada nueva sublista se retorna sin hacer cambios (se cruzan los índices).

Finalmente, al retornar de la primera llamada se tiene el arreglo ordenado:

1. - 2 - 3 - 4 - 5 - 6 - 7

**Método Optimizando:**

Hacer una versión iterativa: Para ello se utiliza una pila en que se van guardando los límites superior e inferior de cada sublista.

No clasificar todas las sublistas: Cuando el largo de las sublistas va disminuyendo, el proceso se va encareciendo. Para solucionarlo sólo se clasifican las listas que tengan un largo menor que n. Al terminar la clasificación se llama a otro algoritmo de ordenamiento que termine la labor. El indicado es uno que se comporte bien con listas casi ordenadas, como el ordenamiento por inserción, por ejemplo. La elección de n depende de varios factores, pero un valor entre 10 y 25 es adecuado.

Elección del elemento de división: Se elige desde un conjunto de tres elementos: lista[inferior], lista[mitad] y lista[superior]. El elemento elegido es el que tenga el valor medio según el criterio de comparación. Esto evita el comportamiento degenerado cuando la lista está prácticamente ordenada.

Una mejora para quick-sort consiste en usar para las particiones más pequeñas otro algoritmo, por ejemplo, el método de la burbuja. Recordemos que si bien la tasa de crecimiento O(n2) nos indica que para valores grandes de n burbuja será siempre más eficientes que quick-sort, para valores pequeños puede ser más eﬁciente y de hecho lo es.

**Análisis del algoritmo:**

[Estabilidad](http://c.conclase.net/orden/?cap=introduccion#estabilidad): No es estable, pero puede serlo si el algoritmo de partición lo es.

[Requerimientos de Memoria](http://c.conclase.net/orden/?cap=introduccion#requerimientos): No requiere memoria adicional en su forma recursiva. En su forma iterativa la necesita para la pila.

[Tiempo de Ejecución](http://c.conclase.net/orden/?cap=introduccion#tiempo_ejecucion):

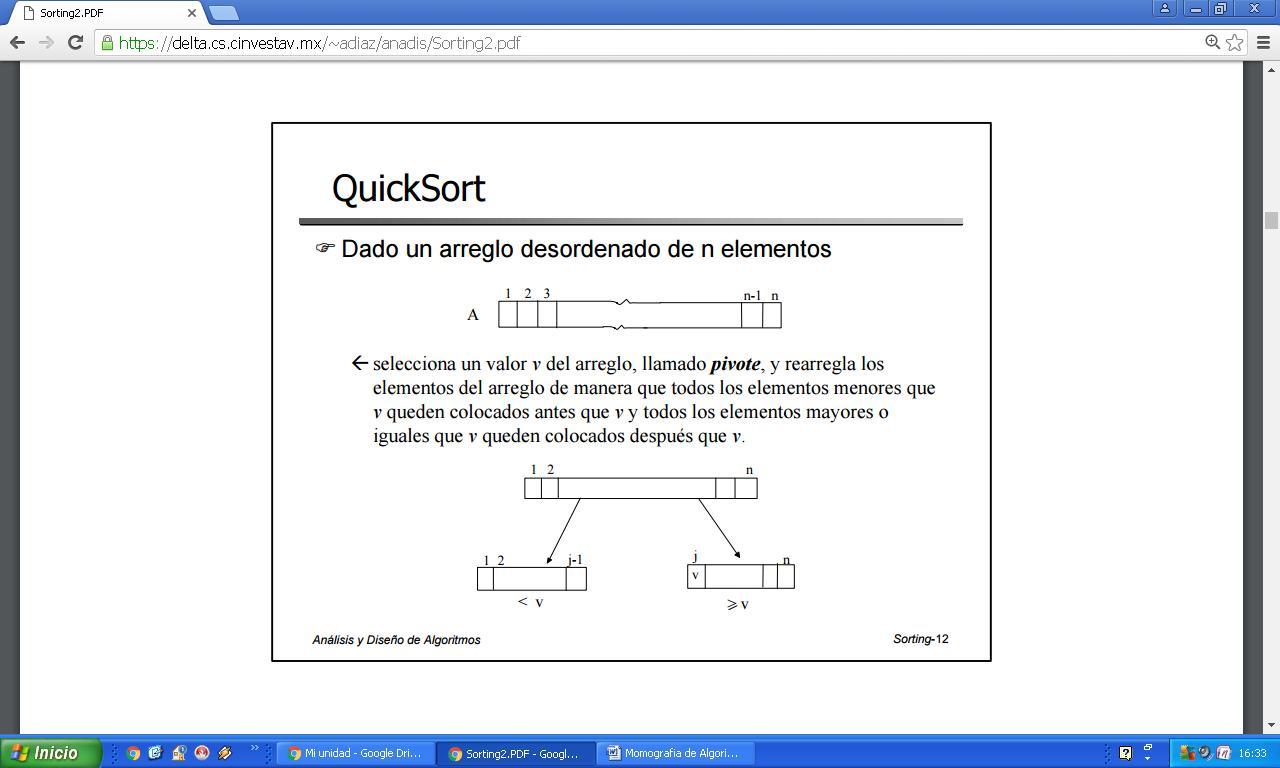
Caso promedio. La complejidad para dividir una lista de n es O(n). Cada sublista genera en promedio dos sublistas más de largo n/2.

**Ventajas:**

* Muy rápido
* No requiere memoria adicional.

**Desventajas:**

* Implementación un poco más complicada.
* Recursividad (utiliza muchos recursos).
* Mucha diferencia entre el peor y el mejor caso.

.

**3.e. Metodo por Mezcla o MergeSort**

Conceptualmente, uno de los algoritmos rápidos más simples de comprender es el algoritmo de “ordenamiento por fusión” o “intercalamiento” (“merge-sort”). Como quick-sort, la estrategia es también típica de “dividir para vencer” e intrínsecamente recursiva. Inicialmente la lista se divide (“split”) en dos sublistas del mismo tamaño y se aplica recursivamente merge\_sort() a cada una de las listas. Luego estas se concatenan (también “fusionan” o “merge”) en L manteniendo el orden.

A esta forma de separar la lista en dos de igual tamaño lo llamamos “splitting par/impar”. De manera que ciertamente es posible implementar merge-sort para listas en tiempo O(nlogn). El concepto de si el ordenamiento es in-place o no cambia un poco para listas. Si bien usamos contenedores auxiliares (las listas L1 y L2), la cantidad total de celdas en juego es siempre n, si tomamos la precaución de ir eliminando las celdas de L a medidas que insertamos los elementos en las listas auxiliares, y viceversa al hacer la fusión. De manera que podemos decir que merge-sort es in-place. Merge-sort es el algoritmo de elección para listas.

**En cuanto a estabilidad, este método es fácil de implementar en la etapa de fusión (“merge”) en forma estable, basta con tener cuidado al elegir el primer elemento de L1 o L2.**



|  |
| --- |
| /\* C program for Merge Sort \*/  #include<stdlib.h>  #include<stdio.h>    // Merges two subarrays of arr[].  // First subarray is arr[l..m]  // Second subarray is arr[m+1..r]  void merge(int arr[], int l, int m, int r)  {      int i, j, k;      int n1 = m - l + 1;      int n2 =  r - m;        /\* create temp arrays \*/      int L[n1], R[n2];        /\* Copy data to temp arrays L[] and R[] \*/      for (i = 0; i < n1; i++)          L[i] = arr[l + i];      for (j = 0; j < n2; j++)          R[j] = arr[m + 1+ j];        /\* Merge the temp arrays back into arr[l..r]\*/      i = 0; // Initial index of first subarray      j = 0; // Initial index of second subarray      k = l; // Initial index of merged subarray      while (i < n1 && j < n2)      {          if (L[i] <= R[j])          {              arr[k] = L[i];              i++;          }          else          {              arr[k] = R[j];              j++;          }          k++;      }        /\* Copy the remaining elements of L[], if there         are any \*/      while (i < n1)      {          arr[k] = L[i];          i++;          k++;      }        /\* Copy the remaining elements of R[], if there         are any \*/      while (j < n2)      {          arr[k] = R[j];          j++;          k++;      }  }    /\* l is for left index and r is right index of the     sub-array of arr to be sorted \*/  void mergeSort(int arr[], int l, int r)  {      if (l < r)      {          // Same as (l+r)/2, but avoids overflow for          // large l and h          int m = l+(r-l)/2;            // Sort first and second halves          mergeSort(arr, l, m);          mergeSort(arr, m+1, r);            merge(arr, l, m, r);      }  }    /\* UTILITY FUNCTIONS \*/  /\* Function to print an array \*/  void printArray(int A[], int size)  {      int i;      for (i=0; i < size; i++)          printf("%d ", A[i]);      printf("\n");  }    /\* Driver program to test above functions \*/  int main()  {      int arr[] = {12, 11, 13, 5, 6, 7};      int arr\_size = sizeof(arr)/sizeof(arr[0]);        printf("Given array is \n");      printArray(arr, arr\_size);        mergeSort(arr, 0, arr\_size - 1);        printf("\nSorted array is \n");      printArray(arr, arr\_size);      return 0;  } |

1. **Diferencia entre los diferentes métodos**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| BURBUJA MEJORADO | INSERCION | SELECCION | QUICKSORT | MERGE |
| Ordenamiento externo, directo | Ordenamiento externo, directo | Ordenamiento externo, directo | Ordenamiento interno, mejora del intercambio directo | Ordenamiento interno |
| Implementación sencilla | Implementación sencilla | Implementación sencilla | Implementación complicada | Implementación sencilla |
| Lento: de comparación. Revisa varias veces la misma lista. | Lento,  natural | Lento | El más rápido y EFICIENTE | Rápido |
| Empleo de banderas para indicar cambios o no.  bandera=1; /\* indica si se han realizados cambios o no | Inserta un elemento del vector en la parte ordenada de la misma, | Busca el elemento más pequeño de la lista  Realiza pocos intercambios | Elige un elemento x al azar (l*pivote*).  Se divide al vector en 2. | Se divide al vector en 2. |
| Las comparaciones van dismuyendo a medida que se efectúan los pasos | Compara un elemento de la parte desordenada de la lista con los elementos de la parte ordenada, | Lo intercambia con el elemento ubicado en la primera posición de la lista.  Realiza numerosas comparaciones. | Se busca por la izquierda del arreglo un elemento mayor que x.  Se busca por la derecha un elemento menor o igual que x. |  |
|  |  | No es estable. | No es estable, pero puede serlo si el algoritmo de partición lo es. | Estable |
| No requiere memoria adicional (añade solo una variable extra) |  | No requiere memoria adicional | No requiere memoria adicional en su forma recursiva. En su forma iterativa la necesita para la pila. |  |
| Eficiente con valores grandes |  | Rendimiento constante: poca diferencia entre el peor y el mejor caso.  Buena opción para listas con [registros](http://c.conclase.net/orden/?cap=introduccion#registro) grandes y [claves](http://c.conclase.net/orden/?cap=introduccion#clave) pequeñas. | Eficiente con valores pequeños |  |
|  |  | Es el más rápido en la parte gráfica | Recursividad (utiliza muchos recursos). |  |

**5. Bibliografía**

* <https://definicion.de/algoritmo/>
* <https://www.taringa.net/+ciencia_educacion/analisis-de-algoritmo_12rzdh>
* <https://runestone.academy/runestone/static/pythoned/AlgorithmAnalysis/QueEsAnalisisDeAlgoritmos.html>
* <https://www.ecured.cu/Ordenamiento_de_burbuja>
* <https://runestone.academy/runestone/static/pythoned/SortSearch/ElOrdenamientoBurbuja.html>
* <http://lwh.free.fr/pages/algo/tri/tri_insertion_es.html>
* <http://www.bufoland.cl/tc/burbuja2.php>
* https://www.ecured.cu/Ordenamiento\_por\_Inserci%C3%B3n
* **https://enrrike87.blogspot.com/2011/06/metodo-de-ordenamiento-por-seleccion-c.html**
* [Métodos de Ordenamiento Tipos de Ordenamiento -Pdf](https://delta.cs.cinvestav.mx/~adiaz/anadis/Sorting2.pdf)
* <https://delta.cs.cinvestav.mx/~adiaz/anadis/Sorting2.pdf>

* **[Algoritmos de ordenamiento. Capítulo seleccion - C++](http://c.conclase.net/orden/?cap=seleccion)** [**http://c.conclase.net/orden/?cap=seleccion**](http://c.conclase.net/orden/?cap=seleccion)

* [Métodos de Ordenamiento](http://mapaches.itz.edu.mx/~mbarajas/edinf/Ordenamiento.pdf): <http://mapaches.itz.edu.mx/~mbarajas/edinf/Ordenamiento>.
* https://ejerciciosresueltosdecsharp.blogspot.com/2014/04/metodo-de-ordenamiento-quick-sort-en-c-sharp.html
* <https://beginnersbook.com/2015/02/quicksort-program-in-c/>
* https://www.geeksforgeeks.org/merge-sort/
* Algoritmos y Estructuras de Datos- www: http://www.cimec.org.ar/aed Facultad de Ingeniería y Ciencias Hídricas Universidad Nacional del Litoral http://fich.unl.edu.ar Centro de Investigación de Métodos Computacionales http://www.cimec.org.ar(Document version: aed-3.1-12-gc28b6c4c) (Date: Thu Aug 17 16:54:16 2017 -0300)
* FUNDAMENTOS DE PROGRAMACIÓN Algoritmos, estructura de datos y objetos. Cuarta edición- Luis Joyanes Aguilar Catedrático de Lenguajes y Sistemas Informáticos Facultad de Informática, Escuela Universitaria de Informática Universidad Pontificia de Salamanca campus de Madrid.