

Trabajo Final Integrador

ALGORITMOS Y ESTRUCTURA DEDATOS

Prof. Rafael Montesinos - Prof. Javier Cantó

Métodos de Ordenamiento de Vectores

Francisco Miguel Perez, Ignacio Paez, Pastrana Angel
Daniel, Rosconi Ignacio Federico, Gerez Braian Esteban



Índice:

1. Definición de Algoritmos
2. Algoritmo Informático
3. Almacén de datos en algoritmos informáticos.
 - a. Los arreglos unidimensionales (vectores)
4. Métodos de Ordenamiento de Vectores.
 - a. Burbuja
 - b. Baraja
 - c. Selección
 - d. QuickSort
 - e. MergeSort
5. Complejidad Computacional
6. Comparaciones de los métodos.
 - a. Comparación de todos los métodos.
 - b. Comparación QuickSort vs MergeSort
7. Bibliografía

¿Qué es un Algoritmo?

*Un **algoritmo** es un conjunto de instrucciones o reglas definidas y no-ambiguas, ordenadas y finitas que permite, típicamente, solucionar un problema, realizar un cómputo, procesar datos y llevar a cabo otras tareas o actividades. Dados un estado inicial y una entrada, siguiendo los pasos sucesivos se llega a un estado final y se obtiene una solución. Los algoritmos son el objeto de estudio de la **algoritmia**.*

Además, podemos hablar que existen distintos tipos de algoritmos y que son empleados para diferentes cosas,

Según Donald Knuth (científico de computación), para que sea considerado un algoritmo informático debe cumplir las siguientes condiciones:

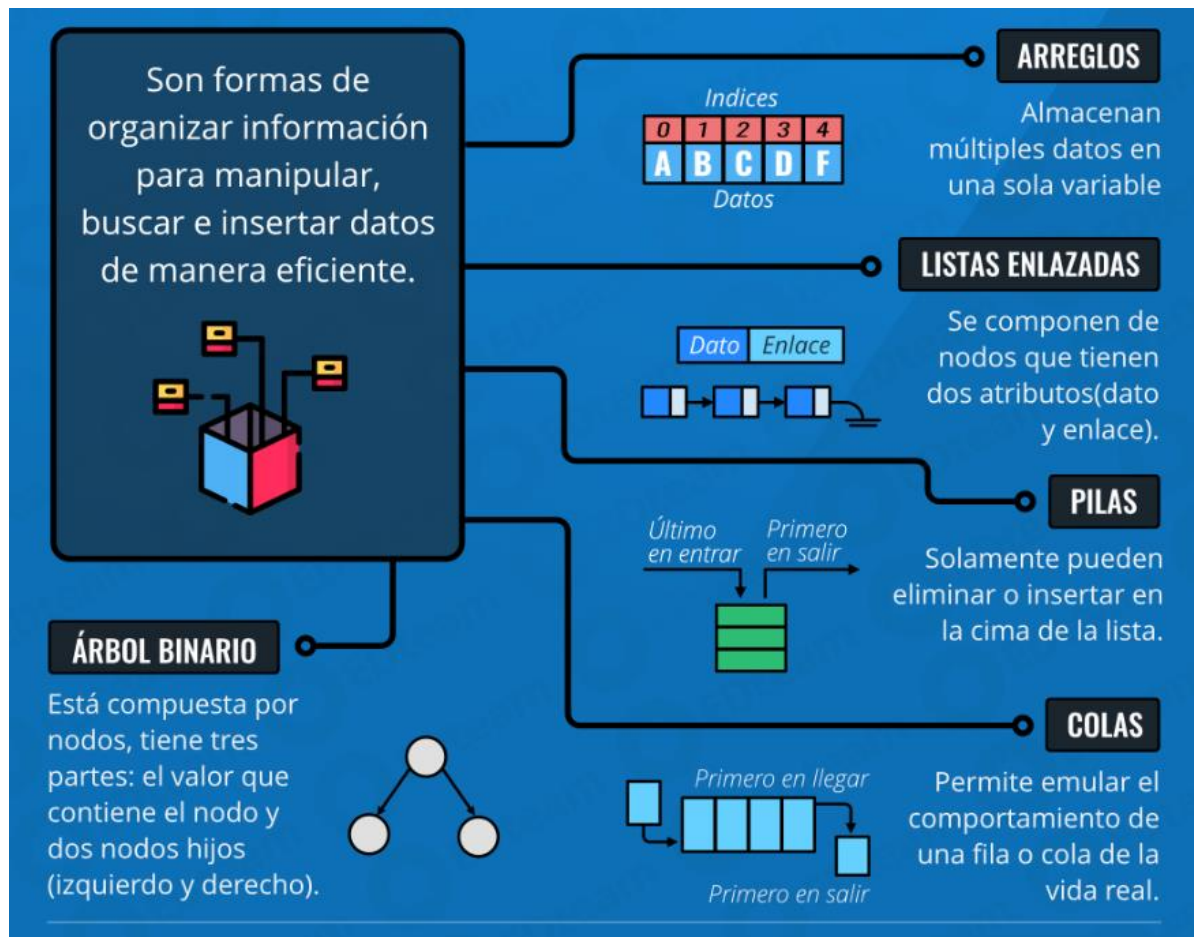
- 1) **SECUENCIAL**: "Los pasos deben tener un orden perfectamente definido".*
- 2) **DEFINIDO**: "Cada paso de un algoritmo debe estar precisamente definido; las operaciones a llevar a cabo deben ser especificadas de manera rigurosa y no ambigua para cada caso".*
- 3) **GENERAL**: "Al encararse la solución de un problema, ésta debe desarrollarse para la generalidad de los casos que pueden presentarse dentro del ámbito de validez de esa solución".*
- 4) **FINITO**: "Un algoritmo siempre debe terminar después de un número finito de pasos"*

Así es como podemos definir que en esta materia hablaremos siempre de algoritmos informáticos y además tendremos que ver el análisis de los algoritmos y la complejidad asociada a ellos.

Por lo tanto en el análisis teórico de algoritmos, es común estimar su complejidad en el sentido asintótico, es decir, estimar la función de complejidad para entradas

arbitrariamente grandes. El término «análisis de algoritmos» fue acuñado por [Donald Knuth](#).

El análisis de algoritmos es una parte importante de la teoría de la complejidad



computacional, que proporciona una estimación teórica de los recursos necesarios de un algoritmo para resolver un problema computacional específico. La mayoría de los algoritmos están diseñados para trabajar con entradas de longitud arbitraria. El análisis de algoritmos es la determinación de la cantidad de recursos de tiempo y espacio necesarios para ejecutarlo.

Por lo general, la eficiencia o el tiempo de ejecución de un algoritmo se establece como una función que relaciona la longitud de entrada con el número de pasos, conocido como complejidad de tiempo o volumen de memoria, conocido como complejidad de espacio.

En resumen:

El análisis de algoritmos se encarga del estudio del tiempo y espacio requerido por un algoritmo para su ejecución. Ambos parámetros pueden ser estudiados con respecto al peor caso (también conocido como caso general) o respecto al caso probabilístico (o caso esperado).

Además, como las estructuras de datos que necesitamos para representar la variedad de información que poseemos es bastante diversa, a ello es que le siguen las

estructuras de datos y como éstas se van clasificando según la necesidad de el programador en si, por lo tanto podemos ver las siguientes formas de organizar datos.

En este caso, veremos sobre los arreglos, del inglés arrays unidimensionales que en nuestro caso serán llamados vectores.

Definimos a un arreglo es una colección de posiciones de almacenamiento de datos, donde cada una tiene el mismo tipo de dato y el mismo nombre.

Viendo la composición de un vector en este caso podemos ver que este vector va desde 0 hasta n siendo n un número natural, cada uno de estos casilleros se denomina elemento del vector y puede ser llenado de un mismo tipo de dato.

Tipos de

Según el dato que

Int's

Float's

Vector "misNotasAyEDFinal" - Tipo de dato almacenado: float

	7	8	9	3	9	10
Índice	0	1	2	3	4	5

Vectores

En este caso tenemos una representación de un vector cualquiera

Vector "misNotas" -- Tipo de Dato almacenado: INT (Entero)							
Índice	7	9	3	r	z	m	d
	0	1	2	n-1	n

Siendo r, z, m y d notas cualesquiera almacenadas en nuestro vector.

Así como podemos generar un vector que pueda contener las notas de un alumno de la UTN-FRT también se puede almacenar cualquier otro tipo de dato necesario para que nosotros, como programadores podamos almacenar los datos necesarios para realizar nuestro trabajo. Además, hay distintos métodos de ordenamiento de vectores según nuestra necesidad; de ahí es donde surge el trabajo.

Mostrando el siguiente ejemplo:

Se posee el siguiente vector que representa las notas en la materia de Algoritmos y Estructura de Datos I de la Universidad Tecnológica Nacional, Facultad Regional de Tucumán.

Esto puede representar, tal vez el orden en que las notas fueron dadas, perteneciendo el índice 0 a la inicial y el 5 a la final o puede representar otras cosas. Sobre esto, surge la necesidad de organizar nuestros vectores de una forma tal que sea más fácil la interpretación o lo que estamos buscando en realidad. Este problema tenía los programadores del pasado, pero ahora con la tecnología y los nuevos lenguajes hay lenguajes que poseen posibles soluciones a el ordenamiento, por ejemplo C# se pueden utilizar métodos dentro de los objetos como `misNotasAyEDFinal.OrderBy` o `misNotasAyEDFinal.OrderByDescending` y así de cierta manera poder ordenar los vectores de forma fácil y rápida. Pero el objeto de investigación es buscar como se realizaban los ordenamientos de vectores y cual es la ventaja o desventaja de cada uno así como también su forma de aplicarlos en C.

Finalmente, de este modo se muestra como los vectores son ordenados de la forma más convencional posible.

Por lo tanto, ahora podemos plantear *¿Qué es ordenar un vector?*

Es la operación de arreglar los elementos en algún orden secuencial de acuerdo a un criterio de ordenamiento. El propósito principal de un ordenamiento es el de facilitar las búsquedas de los miembros del conjunto ordenado. Ordenar un grupo de datos significa mover los datos o sus referencias para que queden en una secuencia por categorías y en forma ascendente o descendente.

Ordenamientos eficientes son importantes para optimizar el uso de otros algoritmos (como los de búsqueda y fusión) que requieren listas ordenadas para una ejecución rápida. También es útil para poner datos en forma canónica y para generar resultados legibles por humanos.

Por último, cabe mencionar que el orden se suele aplicar al tiempo de ejecución, pero puede aplicarse a otros aspectos (p.ej. la memoria que consume un determinado algoritmo)

Distintos Métodos de ordenamiento:

Método de la burbuja (Proporcionado por los profesores).

a. Intercambio o burbuja mejorada.

b. Inserción o método de la baraja

c. Selección o método sencillo

d. Rápido o QuickSort

e. Por Mezcla o MergeSort

Vector "misNotasAyEDFinal" - Tipo de dato almacenado: float

	7	8	9	3	9	10
Índice	0	1	2	3	4	5

Vector "misNotasAyEDFinal" - Ascendente

	10	9	9	8	7	3
Índice	0	1	2	3	4	5

Vector "misNotasAyEDFinal" - Descendente

	3	7	8	9	9	10
Índice	0	1	2	3	4	5

Veremos una forma de ordenar los algoritmos por el denominado método de la burbuja.

Este método es uno de los más viejos y conocidos por todos los programadores, pero en cuanto a la eficiencia que posee dicho método es pésimo, realiza demasiadas comparaciones y para vectores muy largos es demasiado ineficiente por la cantidad de memoria que necesita utilizar al igual de iteraciones.

En este caso veremos el siguiente código que es el del método de la burbuja.

Img 1.

En este caso, creemos que es de alta importancia entender conceptos básicos para el ordenamiento de vectores para así poder seguir con los siguientes métodos.

Trabajo Final Integrador

Ordenamiento por Burbuja



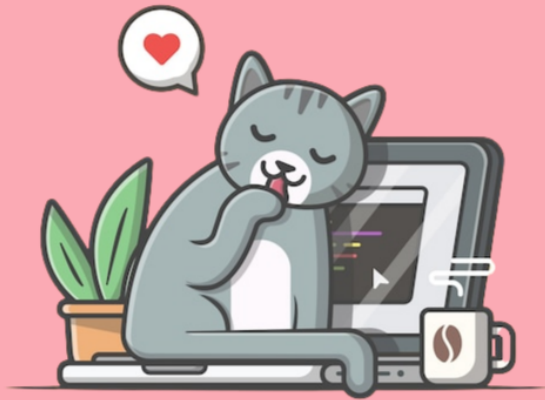
```
if (A [ i ] > A[i+1] ) {  
    AUX=A[i];  
    A[i] = A [i+1];  
    A[i+1]= AUX;  
}
```

En la condición anterior, se debe incluir el operador de relación:

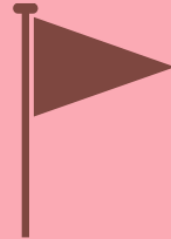
- > si quiere realizar un ordenamiento en forma ascendente.
- < si quiere realizar un ordenamiento en forma descendente

Trabajo Final Integrador

introducción al uso de "Variables Banderas"



Para poder hablar de estos métodos de burbuja, tendremos que tener en claro que es una **Bandera**

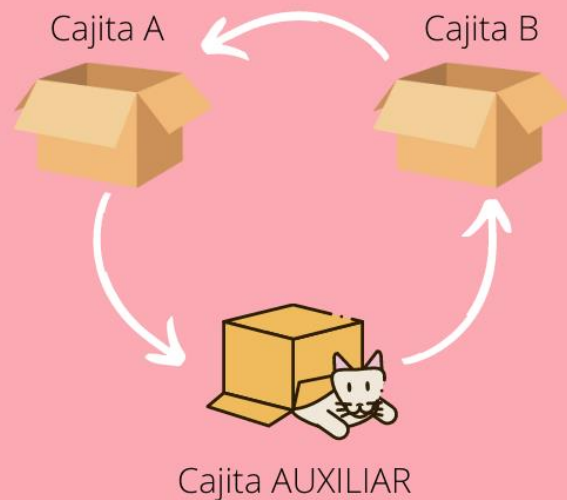


Ejemplo:
¿Está lloviendo?
tiempo = 0
1. Si
2. No
Llueve.
tiempo = 1

Llamaremos una Variable Bandera a aquella que se utiliza para conservar el estado (verdadero o falso) de una condición.

Trabajo Final Integrador

introducción al uso de "cajitas"



Para poder entender este concepto trataremos a las variables como “Cajitas” las cuales almacenan algún tipo de dato, para modo del ejemplo diremos que son datos del tipo ENTERO (int). Daremos 2 Cajitas (variables) que las denominaremos A y B, y además crearemos una cajita Auxiliar para la transferencia de datos.

Se guardarán datos en la Cajita A [Guardemos un 3], y a su vez en la Cajita B [Guardemos un 7], ahora queremos hacer la transferencia de datos de A hacia B y de B hacia A, pero si lo hacemos directamente perderemos un valor. Por lo tanto ahí es donde entra nuestra cajita auxiliar (La que tiene el gatito) y ayudará a almacenar de forma **TEMPORAL** el dato de alguna cajita principal así de este modo se pueda transferir sin miedo a perder los datos.

Así es como, nos quedará la siguiente secuencia:

```
/* *- PSEUDOCÓDIGO ASOCIADO - */
```

```
A = 3, B = 7, AUX = 0
```

```
AUX = A
```

```
A = B
```

```
B = AUX
```

Imprimir A y B

Aquí es como terminará en un ejemplo bastante fácil mostrado la idea de las cajitas, ahora aplicaremos esto en la realidad. Analizaremos por partes la Imagen 1, al principio vemos que tenemos un if (una operación lógica de comparación) si algún elemento de nuestro vector es mayor que el siguiente elemento al primero analizado, y así sucesivamente hasta terminar el ordenamiento del vector, de este modo en el caso de cumplirse la parte del “SI” de dicho ciclo, entonces tendremos que aplicar el concepto de las cajitas donde queda explícitamente detallado como se hace el cambio entre la posición donde se encuentra el **itinerador** (comúnmente llamado **i, j** depende del programador) y el siguiente de la posición, donde el encontrado en **i+1** pasa a **i** y el que antes de cumplirse dicho ciclo (el de **i** pasa a la posición **i+1**). De este modo se realiza un ordenamiento repetido aplicando dicho concepto, pero se ve nuevamente la ineficiencia del mismo.

Así como al ver la poca eficiencia que posee dicho algoritmo para ordenar vectores de gran cantidad de elementos nace **el método de intercambio o burbuja mejorada**.



Método de Ordenamiento por Burbuja Mejorada


Se basa en comparaciones seguidas de dos elementos consecutivos y realizar un intercambio entre los elementos hasta que queden ordenados. En cuanto los datos ya están ordenados se termina inmediatamente ya que detecta que están ordenados porque no se realizan más intercambios en la bandera (bandera en cero al terminar el ciclo interno).

Veamos como sería el código de este método.

Img 3.

Trabajo Final Integrador

Método de Ordenamiento por Burbuja Mejorada



Código:

```
1. for (i=1; i<TAM; i++){
2.     for (j=0 ; j<TAM - 1; j++){
3.         if (lista[j] > lista[j+1])
4.             aux= lista[j];
5.             lista[j] = lista[j+1];
6.             lista[j+1] = aux;
7.     }
8. }
```

El análisis de este por medio de su código es bastante abstracto, por eso lo veremos en su representación gráfica su funcionamiento.

Tomaremos como ejemplo el siguiente vector:

Vector "lista"

	4	3	5	2	1
Índice	0	1	2	3	4

Nos concentraremos en la iteración del for en las variables contadores "i,j", quedándonos:

Vector "lista" [i = 1][j = 0]

	4	3	5	2	1
Índice	0	1	2	3	4

Vector "lista" [i = 1][j = 1]

	3	4	5	2	1
Índice	0	1	2	3	4

Vector "lista" [i = 1][j = 2]

	3	4	5	2	1
Índice	0	1	2	3	4

Vector "lista" [i = 1][j = 3]

	3	4	2	5	1
Índice	0	1	2	3	4

Vector "lista" [i = 1][j = 4]

Índice

3	4	2	1	5
0	1	2	3	4

Vector "lista" [i = 2][j = 0]

Índice

3	4	2	1	5
0	1	2	3	4

Vector "lista" [i = 2][j = 1]

Índice

3	2	4	1	5
0	1	2	3	4

Vector "lista" [i = 2][j = 2]

Índice

3	2	1	4	5
0	1	2	3	4

Vector "lista" [i = 2][j = 3]

Índice

3	2	1	4	5
0	1	2	3	4

De modo tal que se observa que en la iteración del ciclo externo "2" ya podemos encontrar claramente cual es el segundo mayor, y de aquí es donde se deduce una posible "mejora" que podríamos realizar en nuestro algoritmo si somos lo suficientemente capaces de comprenderlo como tal.

Vamos a ver la tercera iteración en i...

Vector "lista" [i = 3][j = 0]

Índice

3	2	1	4	5
0	1	2	3	4

Vector "lista" [i = 3][j = 0]

	2	3	1	4	5
Índice	0	1	2	3	4

Vector "lista" [i = 3][j = 2]

	2	1	3	4	5
Índice	0	1	2	3	4

Finalmente, en la iteración número 4 en I (EL CICLO EXTERNO) Se termina de ordenar nuestro vector.

Vector "lista" [i = 4][j = 0]

	2	1	3	4	5
Índice	0	1	2	3	4

Vector "lista" [i = 4][j = 1]

	1	2	3	4	5
Índice	0	1	2	3	4

Justamente, si notamos algo interesante en nuestra programación es que hay una forma aún de mejorar este algoritmo para que sea mucho más eficiente es implementando el uso de variables banderas (véase en la imagen de variables banderas).

O podemos hacer una simple modificación en la forma de itinerar **j**, como se ve que este va disminuyendo 1 a 1, evitando comparaciones innecesarias a partir del primer ciclo terminado. Es decir que nuestro código quede de la siguiente manera.

```
for (j=0; j<tamaño - i; j++){  
    // el resto de nuestro código aquí  
}
```

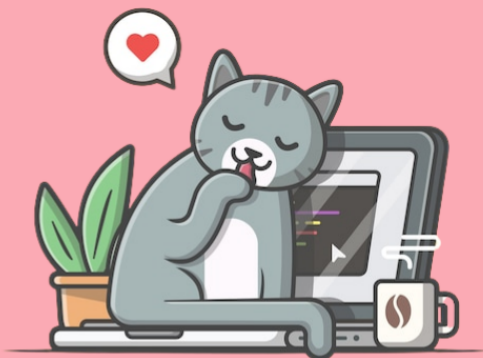
Así de este modo, tendremos que evitar muchísimas comparaciones que consumen memoria y no aportan en nada a nuestro ordenamiento.



Método de Ordenamiento por Inserción / Baraja

Trabajo Final Integrador

introducción al método por Baraja



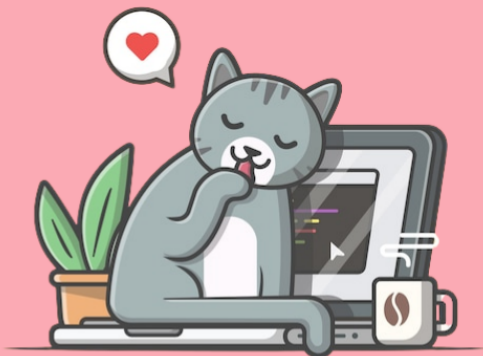
Si alguna vez jugaste a las cartas, te darás cuenta que cuando las recibes, en la mayoría de los casos las ordenas de menor a mayor.



Obviamente cada carta tiene su palo, pero para poder llegar a ordenar nuestros vectores tendremos que abstraernos un poco, de ahí sale la idea de esta forma de ordenar arrays.

Trabajo Final Integrador

Método de Ordenamiento por Baraja



Código:

```
1. for (i=1; i<TAM; i++)
2.   temp = lista[i];
3.   j = i - 1;
4.   while ( (lista[j] > temp) && (j >= 0) )
5.     lista[j+1] = lista[j];
6.     j--;
7.   lista[j+1] = temp;
```


En este caso, deberemos poseer un nivel de abstracción más alto para poder entender lo que intenta hacer este ordenamiento, lo principal es un `for` donde se usa la variable 'TAM' que se refiere al tamaño del vector que queremos ordenar. A su vez utilizamos una variable 'temp' que hace referencia a una variable temporal o auxiliar (nuevamente ponemos en vista la idea de las cajitas) donde almacenamos el elemento en la posición `i` de nuestro vector y creamos una posición `j` que estará dada por `i-1` (Se entiende que en la primera vez que se ejecute el ciclo se encontrará en 0).

Próximamente usaremos una estructura repetitiva, el `while` que nos ayudará a poner 2 condiciones para que se repita dicha estructura, la primera donde hace la comparación del elemento 0 con el elemento guardado en la variable auxiliar temporal **para usar números**, la comparación es entre `i = 1` y `j = 0` de este modo se busca otro elemento y se lo compara.

Nuevamente, entenderlo por medio del código es bastante difícil, por eso es mejor verlo en una aplicación en un vector. Dado el siguiente vector, desordenado:

Vector "lista" (INICIO)

4	3	5	2	1
---	---	---	---	---

Índice

0	1	2	3	4
---	---	---	---	---

Vector "lista"

4	4	5	2	1
---	---	---	---	---

Índice

0	1	2	3	4
---	---	---	---	---

Vector "lista"

3	4	5	2	1
---	---	---	---	---

Índice

0	1	2	3	4
---	---	---	---	---

Vector "lista"

3	4	5	5	1
---	---	---	---	---

Índice

0	1	2	3	4
---	---	---	---	---

Vector "lista"

3	4	4	5	1
---	---	---	---	---

Índice

0	1	2	3	4
---	---	---	---	---

Vector "lista"

	3	3	4	5	1
Índice	0	1	2	3	4

Vector "lista"

	2	3	4	5	1
Índice	0	1	2	3	4

Vector "lista"

	2	3	4	5	5
Índice	0	1	2	3	4

Vector "lista"

	2	3	4	4	5
Índice	0	1	2	3	4

Vector "lista"

	2	3	3	4	5
Índice	0	1	2	3	4

Vector "lista"

	2	2	3	4	5
Índice	0	1	2	3	4

Vector "lista" (FIN)

1	2	3	4	5
---	---	---	---	---

Índice

0	1	2	3	4
---	---	---	---	---

Como se ve, podemos ordenar nuestro vector de una forma mucho más fácil y al igual que el método de la burbuja necesitamos el uso de las "cajitas" para poder ordenarlo. Pero al ser tan parecido al método de la burbuja, tiene casi las mismas ventajas, es fácil de implementar y requiere muy poca memoria como tal, pero hace demasiadas comparaciones lo cual lo hace muy lento.

Nuevamente se recomienda solamente utilizarlo para vectores que sean de poca cantidad de elementos ya que, si no, podría tardarse mucho en su ejecución.

A modo de resumen podemos decir que, tendremos además un contador j , que iniciará en $i-1$, y recorrerá todo el arreglo a la izquierda del elemento i .

- Se guardará el elemento i en una variable temporal.
- Para posteriormente recorrer todo el arreglo de j a 0 . Si algún elemento es mayor que la variable temporal se intercambiará. De lo contrario se recorrerá el arreglo.
- Esto significa que se recorre el arreglo hasta que se encuentre el mejor lugar para insertar el elemento temporal.
- Este procedimiento se repite hasta que se haya concluido con todos los elementos del arreglo. En este momento también el arreglo también estará ordenado.



Método de Ordenamiento por Selección

El algoritmo utilizado es bastante sencillo, se emplea de la siguiente forma:

Buscas el elemento más pequeño de la lista.

Lo intercambias con el elemento ubicado en la primera posición de la lista.

Buscas el segundo elemento más pequeño de la lista.

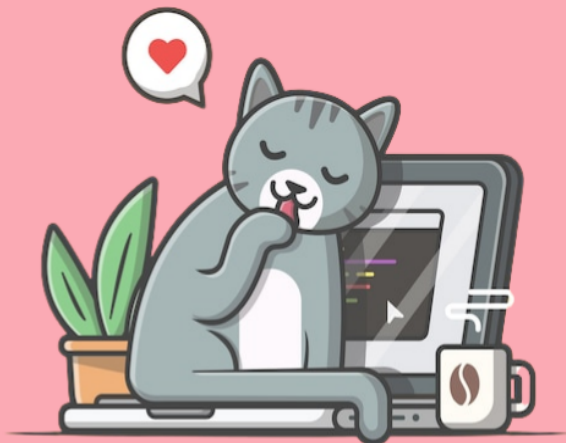
Lo intercambias con el elemento que ocupa la segunda posición en la lista.

Repites este proceso hasta que hayas ordenado toda la lista.

Cabe mencionar que en este algoritmo implementaremos una función, podría no realizarse con ella y hacer que el mismo código se encuentre en el ordenamiento, pero por motivos de que quede más prolijo y más limpio nuestro código implementaremos funciones, ya que recurriremos a ella la cantidad de iteraciones necesarias para ordenar nuestro vector.

Trabajo Final Integrador

introducción a las funciones en C



Para poder implementar este algoritmo recordaremos las funciones en C.

En C tendremos 2 tipos de funciones.

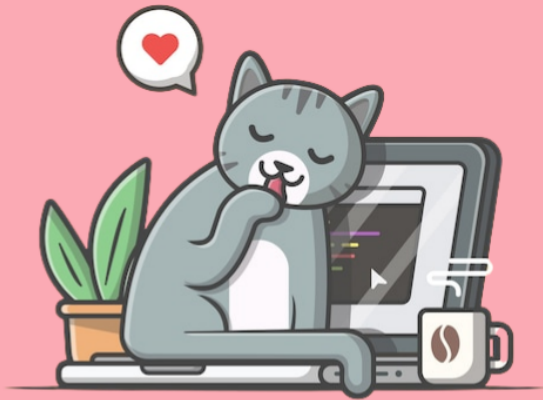
- Con Tipo (int, float, boolean)
- Sin Tipo (Void)

Las implementaremos en nuestro código para poder de cierta manera "reutilizar" el código que ya tenemos para que sea mucho más fácil acomodar nuestro vector.



Trabajo Final Integrador

Función Menor en C, (Con Tipo)



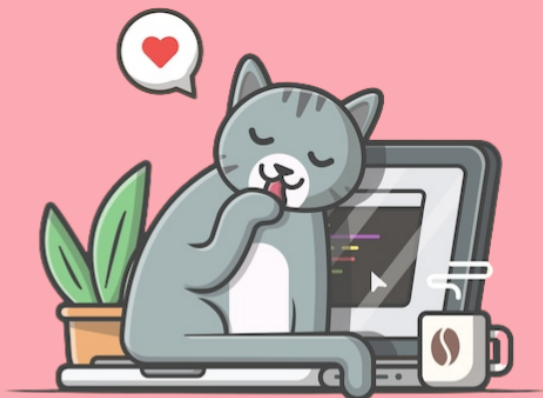
Veremos en este caso la función que nos proporcionará donde se encuentra el elemento menor de una lista.

```
int Menor(int lista[], int TAM, int m){  
    int pos;  
    for(int m = 0; m < TAM; m++){  
        if(lista[m+1] > lista[m]){  
            pos = m;  
        }  
    }  
    return pos;  
}
```

Es bastante fácil el funcionamiento de la función Menor creada, en este caso solo recibirá vectores del tipo **ENTERO** y sobre estos buscará el menor, simplemente recorre dicho vector haciendo comparaciones buscando cual es el menor y cuando finaliza entrega cual es el menor de todo nuestro arreglo analizado.

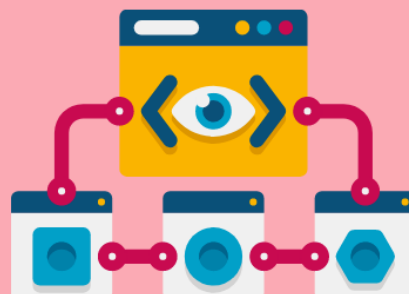
Trabajo Final Integrador

- Ordenamiento por Selección -



Código:

```
1. for (i=0; i<TAM - 1; i++){  
2.     pos_men = Menor(lista, TAM, i);  
3.     temp = lista[i];  
4.     lista[i] = lista [pos_men];  
5.     lista [pos_men] = temp;  
6. }
```



Lo veremos ahora en un vector como tal, el mismo que los ejemplos anteriores y como funciona este método para ordenarlo.

Vector "lista"

4	3	5	2	1
---	---	---	---	---

Índice

0	1	2	3	4
---	---	---	---	---

Vector "lista"

4	3	5	2	1
---	---	---	---	---

Índice

0	1	2	3	4
---	---	---	---	---

Vector "lista"

1	3	5	2	4
---	---	---	---	---

Índice

0	1	2	3	4
---	---	---	---	---

Vector "lista"

1	2	5	3	4
---	---	---	---	---

Índice

0	1	2	3	4
---	---	---	---	---

Vector "lista"

1	2	3	5	4
---	---	---	---	---

Índice

0	1	2	3	4
---	---	---	---	---

Vector "lista"

1	2	3	4	5
---	---	---	---	---

Índice

0	1	2	3	4
---	---	---	---	---

Finalmente obtendremos nuestro vector ordenado.

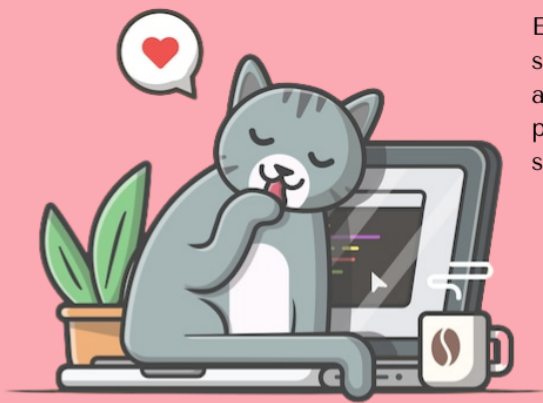


Método de Ordenamiento

Rápido / QuickSort

Trabajo Final Integrador

- Qué es la recursividad? -



Es un proceso mediante el que una función se llama a sí misma de forma repetida, hasta que se satisface alguna determinada condición. El proceso se utiliza para computaciones repetidas en las que cada acción se determina mediante un resultado anterior.



Podemos concluir sobre el algoritmo del método de la siguiente manera; buscaremos un elemento de la lista; puede ser cualquiera, lo denominaremos como elemento de división, buscaremos la posición que le corresponde en la lista ordenada

Acomodamos los elementos de la lista a cada lado del elemento de división, de manera que a un lado queden todos los menores que él y al otro los mayores (explicado más abajo también). En este momento el elemento de división separa la lista en dos sublistas.

Realizas esto de forma recursiva para cada sublista mientras éstas tengan un largo mayor que 1.

Una vez terminado este proceso todos los elementos estarán ordenados.

La idea de ir aplicando recursividad es bastante difícil de entender, pero para ello podemos repensar nuestro algoritmo mediante nuevos índices para recorrer y verificar el ordenamiento de nuestro vector por el método utilizaremos las variables *i*, *j* dándole el nombre a *i* como el contador por la izquierda y a *j* como el contador por la derecha, aplicando el siguiente algoritmo:

- Recorreremos la lista simultáneamente con *i* y *j*: por la izquierda con *i* (desde el primer elemento), y por la derecha con *j* (desde el último elemento).

• Cuando `lista[i]` sea mayor que el elemento de división y `lista[j]` sea menor los intercambiaremos.

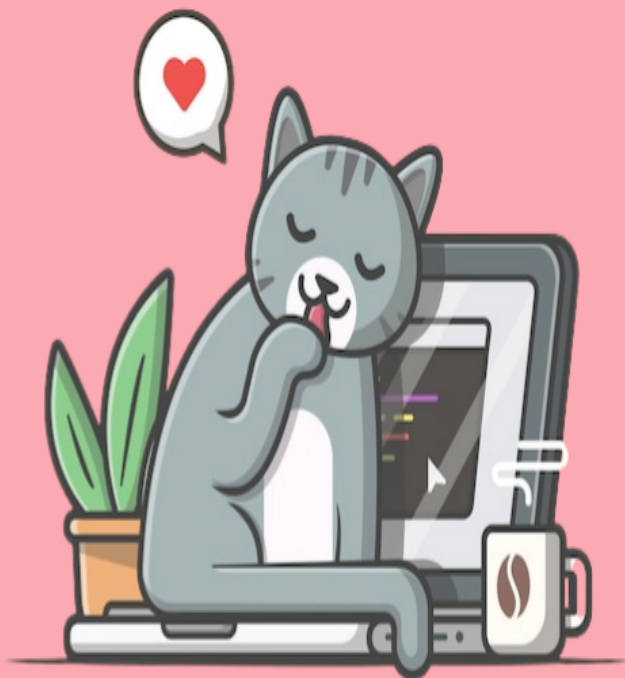
Repites esto hasta que se crucen los índices.

El punto en que se cruzan los índices es la posición adecuada para colocar el elemento de división, porque sabemos que a un lado los elementos son todos menores y al otro son todos mayores (o habrían sido intercambiados).

Al finalizar este procedimiento el elemento de división queda en una posición en que todos los elementos a su izquierda son menores que él, y los que están a su derecha son mayores.

Trabajo Final Integrador

- (cómo sería el código) -



Nombre Procedimiento: OrdRap

Parámetros:

lista a ordenar (lista)

índice inferior (inf)

índice superior (sup)

// Inicialización de variables

1. elem_div = lista[sup];

2. i = inf - 1;

3. j = sup;

4. cont = 1;

// Verificamos que no se crucen los límites

5. if (inf >= sup)

6. retornar;

// Clasificamos la sublista

7. while (cont)

8. while (lista[++i] < elem_div);

9. while (lista[--j] > elem_div);

10. if (i < j)

11. temp = lista[i];

12. lista[i] = lista[j];

13. lista[j] = temp;

14. else

15. cont = 0;

// Copiamos el elemento de división

// en su posición final

16. temp = lista[i];

17. lista[i] = lista[sup];

18. lista[sup] = temp;

// Aplicamos el procedimiento

// recursivamente a cada sublista

19. OrdRap (lista, inf, i - 1);

20. OrdRap (lista, i + 1, sup);

Véase que este método es mucho más rápido que los demás pero es bastante y mucho más confuso que los anteriores para entender con el código, por eso luego de la explicación inicial, lo mejor será verlo en vectores como es el funcionamiento.

Vector "lista"

	5	3	7	6	2	1	4
Índice	0	1	2	3	4	5	6

Vector "lista" [4 es el elemento divisor.]

	5	3	7	6	2	1	4
Índice	0	1	2	3	4	5	6

Vector "lista" (Gris para Izq [i], Azul para Derecha [j])

	5	3	7	6	2	1	4
Índice	0	1	2	3	4	5	6

Vector "lista" [4 es el elemento divisor.]

	1	3	7	6	2	5	4
Índice	0	1	2	3	4	5	6

Vector "lista" [4 es el elemento divisor.]

	1	3	7	6	2	5	4
Índice	0	1	2	3	4	5	6

Vector "lista" [4 es el elemento divisor.]

	1	3	7	6	2	5	4
Índice	0	1	2	3	4	5	6

Vector "lista" [4 es el elemento divisor.]

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

Índice 0 1 2 3 4 5 6

Vector "lista" [Se cruzaron los indices] (Es decir $i = j$)

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

Índice 0 1 2 3 4 5 6

Aquí es donde se empieza a aplicar la recursividad, cuando se finaliza el ciclo principal.

Vector "Lista" (Separado en 2)

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

Índice 0 1 2 3 4 5 6

Vector "Lista" (Separado en 2)

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

Índice 0 1 2 3 4 5 6

Vector "Lista" (Separado en 2)

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

Índice 0 1 2 3 4 5 6

En este lugar se realiza la subdivisión para poder ordenarlos de manera más rápida, dividimos la lista en partes iguales para poder aprovecharnos de esto, luego de esto en cada paso se van ordenando hasta que nos retorna el vector completo ordenado.

Vector "Lista" **ORDENADO.**

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

Índice 0 1 2 3 4 5 6



Método de Ordenamiento

Mezcla / MergeSort

Nuevamente haremos uso de conceptos aprendidos con anterioridad, en efecto la recursividad aparece en este algoritmo de ordenamiento.

Consiste en dos pasos específicos:

1. Dividir el arreglo por la mitad hasta encontrar un vector de una sola posición.
2. Mezclar los arreglos resultantes considerando los valores de los elementos.

La forma de trabajar de este método es mucho más abstracta que las demás, ya que es muy difícil entenderlo simplemente con la teoría, por eso lo mejor es proporcionar un ejemplo de su funcionamiento.

Trabajo Final Integrador

- MergeSort -

[12 0 6 2 9 34 1]

Se busca el elemento del medio y a partir usamos recursividad.



[12 0 6 2]

[9 34 1]

[12 0] [6 2]

[9 34] [1]

[12] > [0] [6] > [2] [9] > [34] [1]

[0 12] [2 6]

[9 34] [1]

[0 2 6 12]

[1 9 34]

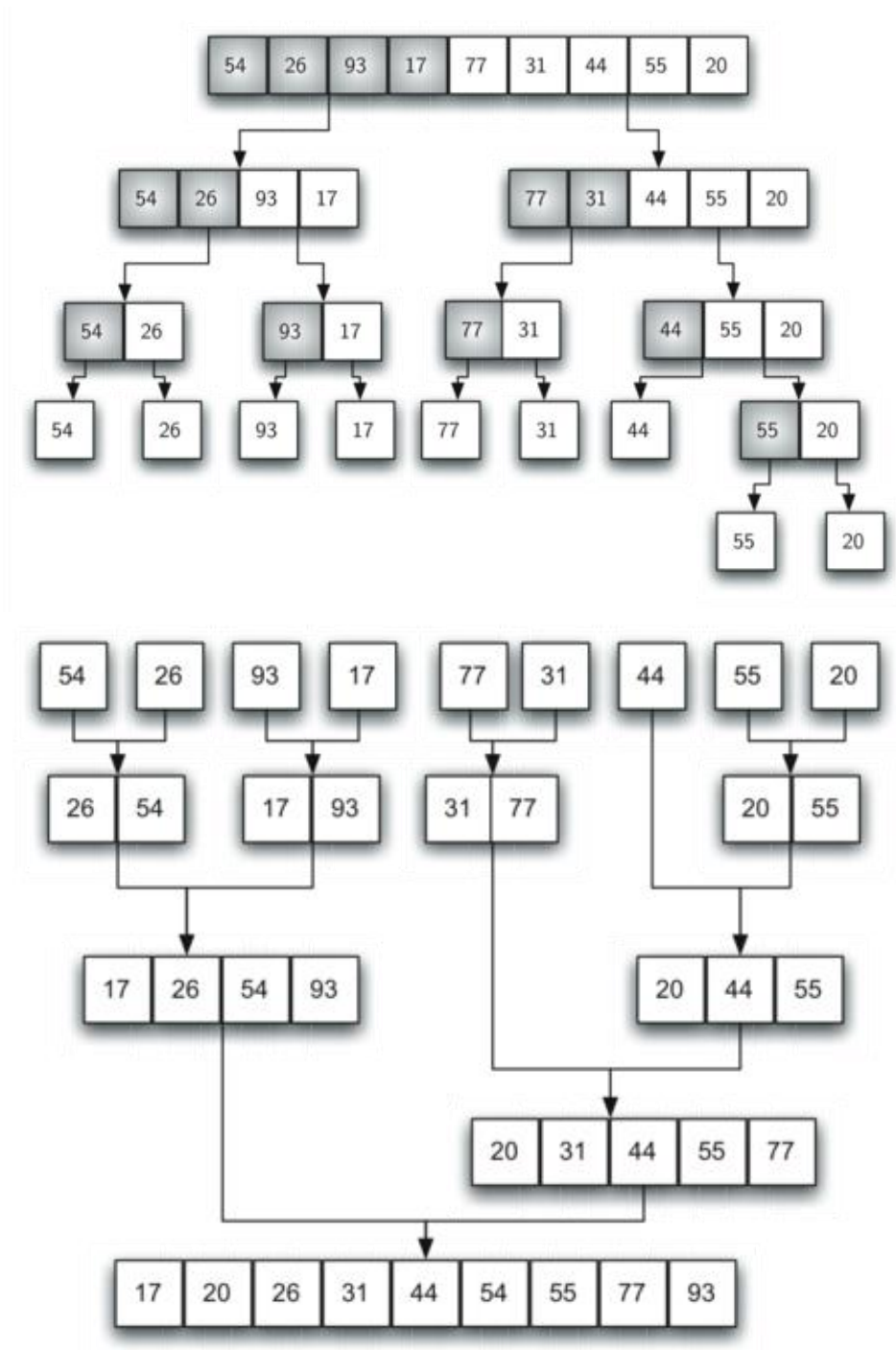
Vector Ordenado Ascendente: [0 1 2 6 9 12 34]

Como vemos poseemos el siguiente arreglo [12, 0, 6, 2, 9, 34, 1] y lo primero que haremos será encontrar el elemento medio que servirá como divisor inicial para hacerlo entre las listas de este modo, podremos obtener subarreglos para que sea más fácil la comparación.

Es importante destacar que la idea es llevarlo a la mínima cantidad de elementos posibles de un arreglo (uno) de este modo así compararlo con otro, y ahí es cuando entra en juego la recursividad. Una vez que se comparan y se ordenan los arreglos se mezcla a ambos en arreglos más grandes para de este modo poder seguir ordenandolos hasta llegar a la

ultima posición representada en el gráfico, (anterior al vector ordenado) donde se posee 2 vectores ordenados y se realiza el mismo proceso juntándolos y ordenándolos para así poder obtener el vector ordenado de forma ascendente.

Del mismo modo se da otro ejemplo con un vector de 9 elementos.





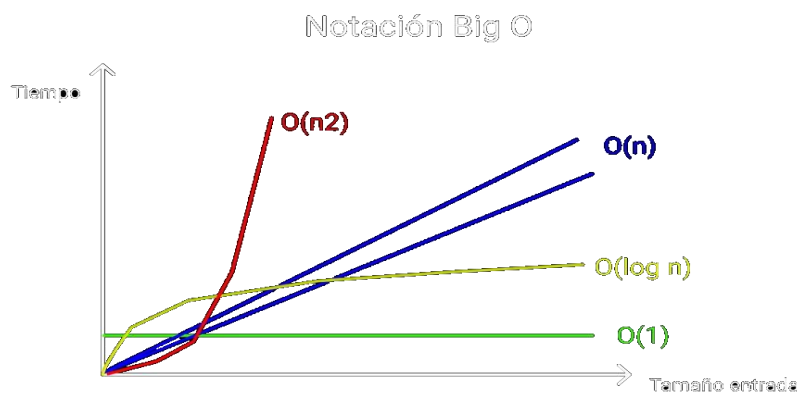
Complejidad Computacional

Notación Big O.

Este tipo de notación es específica para analizar el rendimiento de un algoritmo, es una forma matemática básica y sencilla de expresar cuanto tarda un algoritmo en ejecutarse y su eficiencia para poder ser comparados con otros.

En este caso, como parece casi imposible dar tiempos exactos se utiliza una variable para abstraerse pero que represente una magnitud **N**, básicamente será nuestra unidad de medida. En este tipo de notación hablamos de un análisis asintótico fijándonos que sucede con él en el límite infinito, porque así representamos los peores casos en que pueden ser medidos el rendimiento de nuestros algoritmos.

A su vez, podemos encontrar formas de clasificación, como:



Así mismo, se seguirán los siguientes patrones, a escalas generales:

Cualquier línea de código se considera de $O(1)$ siempre y cuando no sea un ciclo, no sea recursividad o que tenga una llamada a una función de tiempo constante.

En los ciclos consideramos $O(n)$ cuando la variable del ciclo incrementa o decrementa siempre y cuando el punto de control no varíe, en el caso de variar se considera $O(1)$.

En el caso de encontrar ciclos anidados se considera $O(n^2)$ para dos, $O(n^3)$ para tres, etcétera. Esto depende de obviamente que hacen los ciclos internos.

Cuando la variable del contador de un ciclo se multiplique o divida se considera $O(\log[n])$

Cuando se incrementa de forma exponencial sería $O(\log[\log\{n\}])$

En el caso de los condicionales, se toma el peor caso que puede existir (Donde más líneas de ejecución existan) y se analiza su notación.

Para definir la notación final, se analizará que tan complejo es el código y próximamente se sumarán agrupando semejantes y quedándose con el término más significativo.

$3n^2 + 4n + 3 \rightarrow$ El término más significativo es n^2



Trabajo Final Integrador

Complejidad Computacional

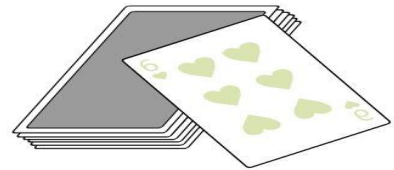
En notación Big O.



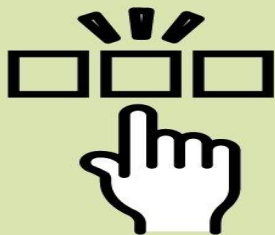
BURBUJA

Su complejidad computacional es de $O(n^2)$

Su complejidad computacional es de $O(n^2)$



BARAJA



SELECCIÓN

Su complejidad computacional es de $O(n^2)$

Su complejidad computacional es de $O(n \log[n])$



QUICKSORT



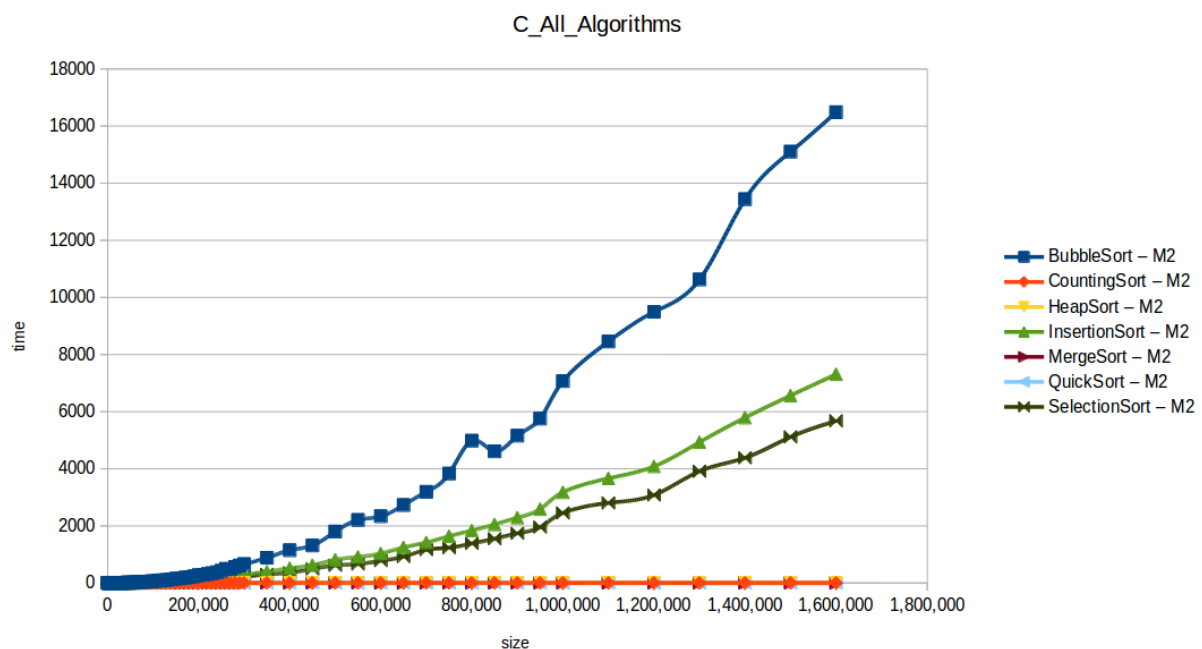
MERGESORT

Su complejidad computacional es de $O(n \log[n])$

Para poder seguir hablando de comparaciones y ventajas de ciertos algoritmos frente a otros hay que ponerlos en práctica a estos para ver que frutos dan al realizar extensas comparaciones de vectores, estos datos son recopilados a partir de la página pereiratechtalks.com de donde se sacaron y fueron probados en servidores alojados en la nube de **Digital Ocean** obteniendo resultados de comparaciones de 7 métodos de ordenamiento de algoritmos, de los cuales existen dos métodos que no trabajamos en esta investigación que son **Counting Sort** y **Heapsort**.

A los fines prácticos de la investigación si se desea más información se puede acceder al link relacionado en la bibliografía, aquí se recopilará la conclusión de la investigación.

Fueron probados en ordenamiento de 1.000.000.000 de datos en 2 máquinas virtuales dando como resultado los siguientes gráficos.



Esta es una gráfica de todos los algoritmos en C y su funcionamiento en la máquina número 2, la cual era mucho más potente que la 1.

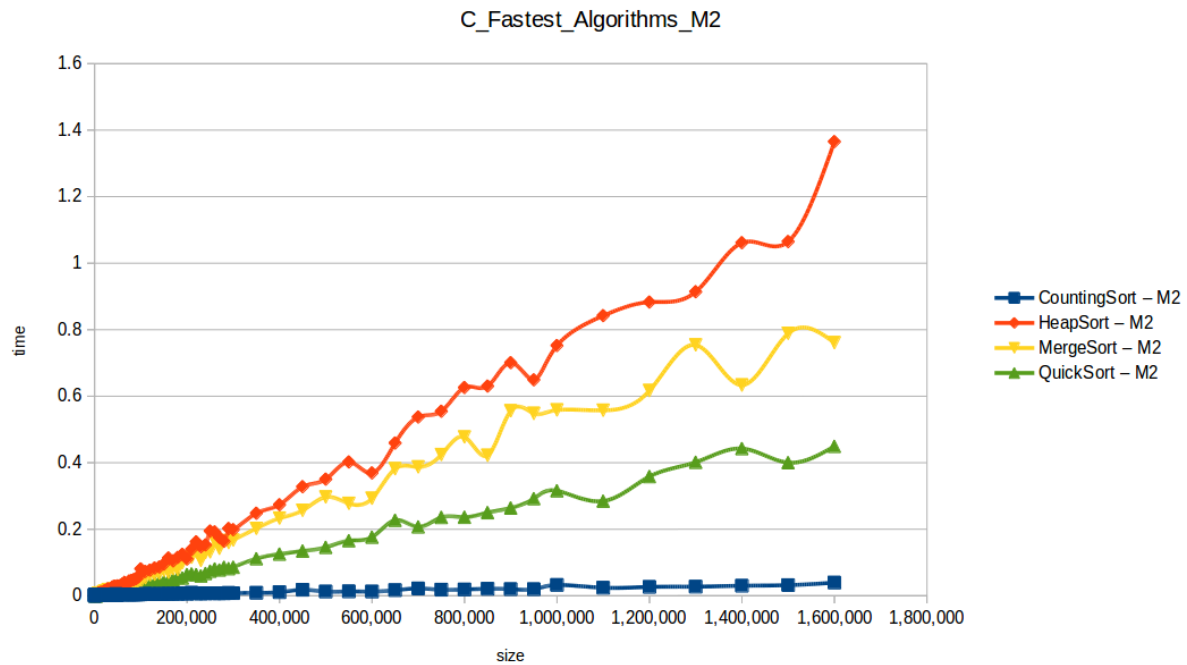
Pero como se ve en la gráfica tenemos un claro perdedor en cuanto a eficiencia que es el BubbleSort, y esto es más que claro pues su complejidad es de $O(n^2)$ próximamente le siguen los algoritmos de la Baraja y la Selección.

En nuestro análisis el caso del QuickSort y MergeSort se solapan entre ellos, pero si vamos a un análisis donde solo veamos a estos 2 algoritmos para ordenar 1,6 millones de datos.

Nos fijaremos únicamente en la gráfica amarilla y verde que son el MergeSort y QuickSort, respectivamente.

Podremos obtener que el QuickSort es mucho mejor en términos de eficiencia, pero hay que tener cuidado con su implementación ya que una mala elección del pivote podría hacer que su complejidad escale a $O(n^2)$ y eso haría que sea menos eficiente.

Se adjunta el respectivo gráfico para su visualización.



Por lo tanto, concluimos que siempre y cuando trabajemos con grandes cantidades de datos deberemos utilizar métodos como el QuickSort o MergeSort y todo dependerá del programador quien decide cual utilizar.

Trabajo Final Integrador

- QuickSort vs MergeSort -



Ambos son algoritmos de ordenamiento
Ambos comparten la idea de divide y vencerás.

QuickSort



MergeSort

No divide en partes iguales.
No necesitas un Array Auxiliar
No necesitas mezclar.
Complejidad en Casos:
Mejor: $O(n \log[n])$
Promedio: $O(n \log[n])$
Peor: $O(n^2)$

Divide en el medio exacto.
Un array auxiliar se necesita.
Se debe mezclar el array.
Complejidad en Casos:
Mejor: $O(n \log[n])$
Promedio: $O(n \log[n])$
Peor: $O(n \log[n])$

Bibliografía

<https://es.wikipedia.org/wiki/Algoritmo>

[https://es.wikipedia.org/wiki/An%C3%A1lisis de algoritmos](https://es.wikipedia.org/wiki/An%C3%A1lisis_de_algoritmos)

http://www.luchonet.com.ar/aed/?page_id=209

<http://mapaches.itz.edu.mx/~mbarajas/edinf/Ordenamiento.pdf>

[https://es.wikipedia.org/wiki/Algoritmo de ordenamiento#:~:text=En%20computaci%C3%B3n%20y%20matem%C3%A1ticas%20un,la%20relaci%C3%B3n%20de%20orden%20da%20da.](https://es.wikipedia.org/wiki/Algoritmo_de_ordenamiento#:~:text=En%20computaci%C3%B3n%20y%20matem%C3%A1ticas%20un,la%20relaci%C3%B3n%20de%20orden%20da%20da.)

<https://slideplayer.es/slide/3581520/https://slideplayer.es/slide/3581520/>

[http://www.alciro.org/alciro/Programacion-cpp-Builder 12/ordenacion-intercambio-burbuja 447.htm](http://www.alciro.org/alciro/Programacion-cpp-Builder_12/ordenacion-intercambio-burbuja_447.htm)

<http://eenube.com/index.php/ldp/algoritmos/135-algoritmo-de-ordenamiento-por-insercion-con-c>

[https://www.ecured.cu/Algoritmo de ordenamiento por selecci%C3%B3n](https://www.ecured.cu/Algoritmo_de_ordenamiento_por_selecci%C3%B3n)

<https://ronnyml.com/2009/07/19/quicksort-en-c/>

<https://runestone.academy/runestone/static/pythoned/SortSearch/ElOrdenamientoPorMezcla.htm>

<https://naps.com.mx/blog/ejemplos-explicados-de-arreglos-en-c/>

<https://tecpro-digital.com/ejercicios-con-metodos-de-ordenamiento-en-c/>

<https://conclase.net/c>

https://www.youtube.com/watch?v=MyAiCtuhigQ&ab_channel=ChioCode

<https://pabloclanes.com/notacion-big-o/>

<https://pereiratechtalks.com/analisis-de-algoritmos-de-ordenamiento/>

https://www.youtube.com/watch?v=Jjr4OqZ2VY&ab_channel=CSEGURUS