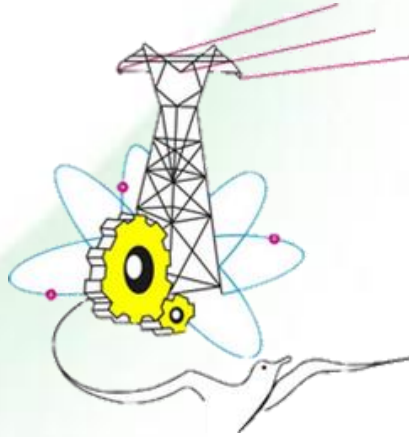


# Instituto Tecnológico de Ensenada



## INTELIGENCIA ARTIFICIAL

### Métodos de ordenamiento

Profesor: Ing. Alejandro Chávez Sánchez

Alumno: Morales Hernández M.

Álvarez Sandoval Igor

## Descripción del algoritmo

### *Ordenamiento por inserción*

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-ésimo elemento.

Ejemplo:

Se desea ordenar el siguiente arreglo A: 15, 67, 08, 16, 44, 27, 12, 35

Primera pasada

$A[2] < A[1]$   $67 < 15$  No hay intercambio

A: 15, 67, 08, 16, 44, 27, 12, 35

Segunda pasada

$A[3] < A[2]$   $08 < 67$  Si hay intercambio

$A[2] < A[1]$   $08 < 15$  Si hay

A: 15, 08, 67, 16, 44, 27, 12, 35

Tercera pasada

$A[4] < A[3]$   $08 < 15$  Si hay intercambio

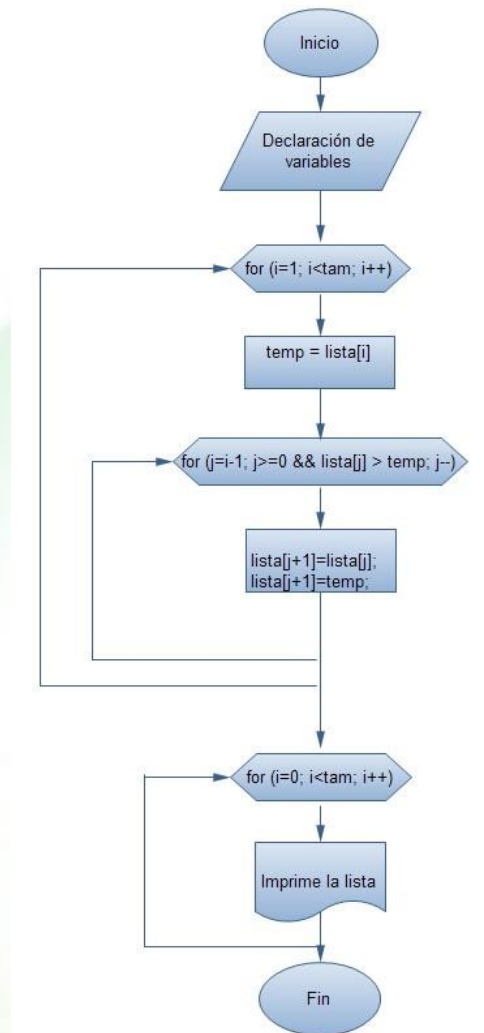
$A[3] < A[2]$   $08 < 15$  Si hay intercambio

A= 08, 15, 67, 16, 44, 27, 12, 35

Hasta la séptima pasada el arreglo queda ordenado: 08, 12, 15, 16, 27, 35, 44, 67

## Algoritmo en C++

### Diagrama de flujo

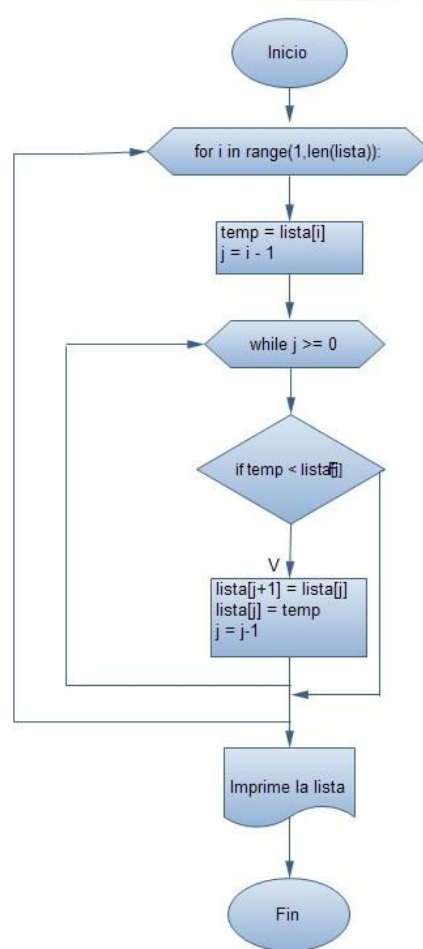


### Algoritmo

```
for (i=1; i<tam; i++){ //inicializamos el ciclo en la posición 1 para empezar la comparación
    temp = lista[i]; //inicializamos temp en la posición i del vector
    for (j=i-1; j>=0 && lista[j] > temp; j--){ //Realiza las comparaciones
        lista[j+1]=lista[j]; //Se intercambian los valores
    }
    lista[j+1]=temp; //Se iguala a variable temp
}
```

## Algoritmo en Python

### Diagrama de flujo



```
for i in range(1, len(lista)): //inicializamos el ciclo en la posición 1 para empezar la comparación
    temp = lista[i] //inicializamos temp en la posición i del vector
    j = i - 1 // Para comparar con posición 0
    while j >= 0: //Mientras sea mayor o igual a 0
        if temp < lista[j]: //Empezamos la comparaciones
            lista[j+1] = lista[j] // Se intercambian los valores
            lista[j] = temp // Se iguala a variable temp
            j = j - 1 // disminuimos el valor de j
```

## Complejidad en espacio

### En C++

Memoria estática: variables declaradas en el algoritmo.

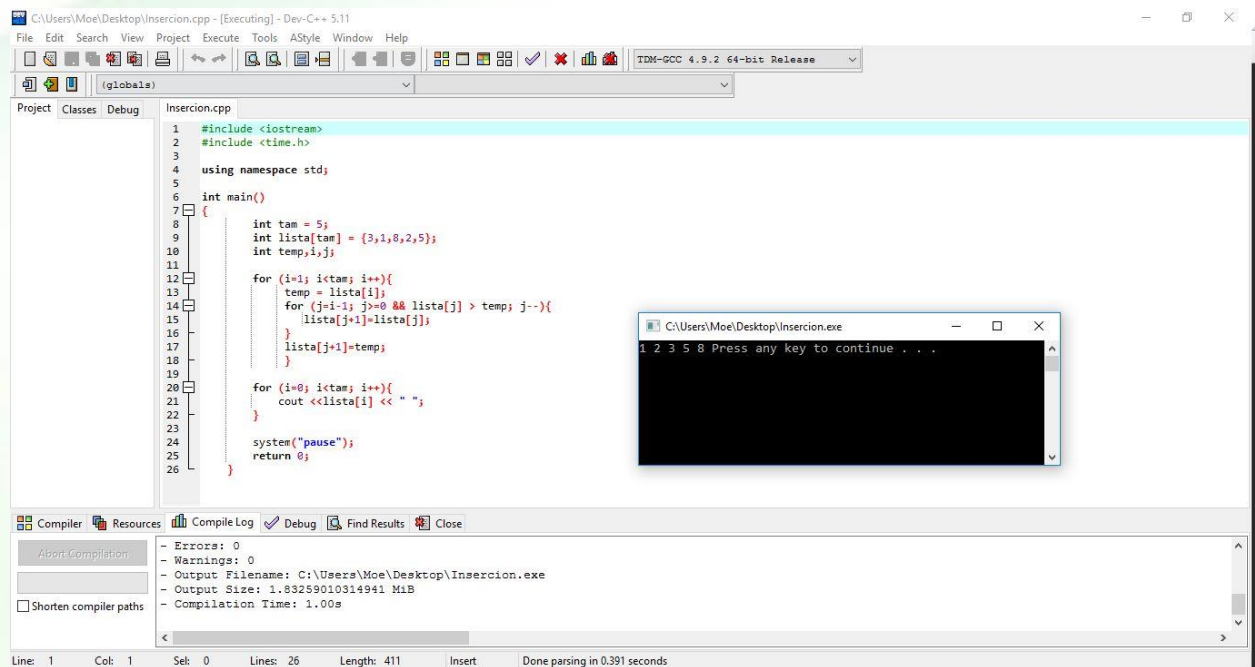
Int tam, lista, temp, i, j = 10 bytes

### En Python

Memoria dinámica: depende de cada ejecución del algoritmo.

## Ejecución del algoritmo

### En C++:

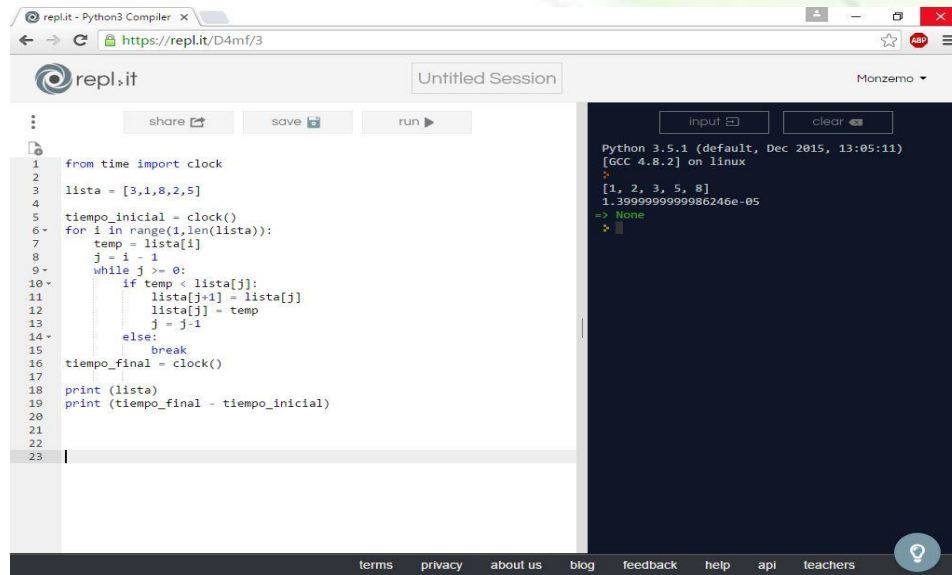


```
1 #include <iostream>
2 #include <time.h>
3
4 using namespace std;
5
6 int main()
7 {
8     int tam = 5;
9     int lista[tam] = {3,1,8,2,5};
10    int temp,i,j;
11
12    for (i=1; i<tam; i++){
13        temp = lista[i];
14        for (j=i-1; j>=0 && lista[j] > temp; j--){
15            lista[j+1]=lista[j];
16        }
17        lista[j+1]=temp;
18    }
19
20    for (i=0; i<tam; i++){
21        cout << lista[i] << " ";
22    }
23
24    system("pause");
25    return 0;
26 }
```

Output: 1 2 3 5 8 Press any key to continue . . .

Compiler Output: - Errors: 0 - Warnings: 0 - Output Filename: C:\Users\Moe\Desktop\Insercion.exe - Output Size: 1.83259010314941 MiB - Compilation Time: 1.00s

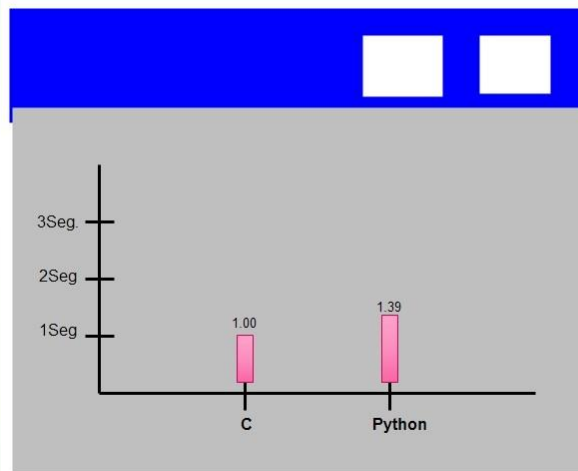
## En Python:



```
1 from time import clock
2
3 lista = [3,1,8,2,5]
4
5 tiempo_inicial = clock()
6 for i in range(1,len(lista)):
7     temp = lista[i]
8     j = i - 1
9     while j >= 0:
10         if temp < lista[j]:
11             lista[j+1] = lista[j]
12             lista[j] = temp
13             j = j - 1
14         else:
15             break
16 tiempo_final = clock()
17
18 print (lista)
19 print (tiempo_final - tiempo_inicial)
20
21
22
23
```

Python 3.5.1 (default, Dec 2015, 13:05:11)  
[GCC 4.8.2] on linux  
[1, 2, 3, 5, 8]  
1.3999999999986246e-05  
=> None

## Grafica de tiempos



## Conclusión

Podría decirse que es muy poca la diferencia de velocidad entre ambos programas midiendo los tiempos de ejecución, en conclusión ambos programas requieren más tiempo para vectores más grandes por lo cual se vuelven más lentos, son buenos y rápidos, pero depende mucho de a que se le implementaran los métodos. Es de fácil implementación, de requerimientos mínimos de memoria pero tiende a ser lento debido a sus numerosas comparaciones.



## Descripción del algoritmo

### *Ordenamiento por selección (Selection Sort)*

El algoritmo de ordenación por selección es una especie natural de la clasificación técnica. También es fácil de implementar, pero como el algoritmo de ordenamiento Burbuja, también no es eficiente, porque va perdiendo eficiencia el algoritmo cuando "n" se hace muy grande. El algoritmo de ordenación por selección toma el k-ésimo elemento más pequeño de una lista de números; Dependiendo de cómo se está organizando la matriz el elemento va siendo intercambiado con los elemento de la matriz para colocar el elemento menor en su posición correcta. El proceso se repite con el segundo valor más pequeño y así sucesivamente hasta que este ordenada la lista.

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:

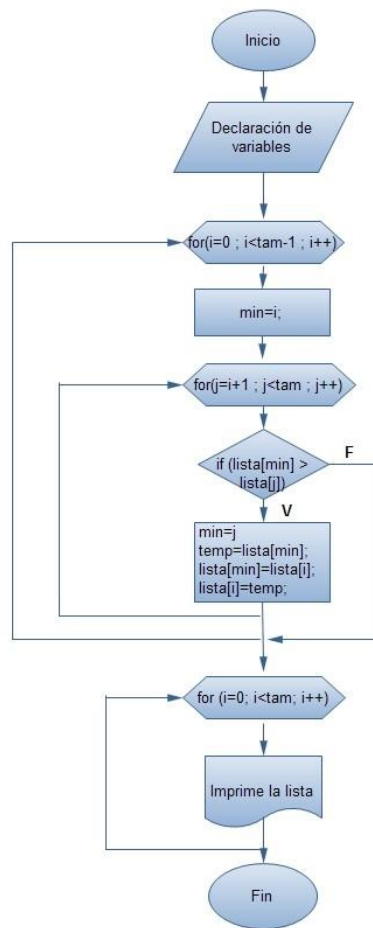
1 - 2 - 3 - 5 - 4

El menor elemento entre la cuarta y quinta posición es el 4, que intercambiamos con el 5:

1 - 2 - 3 - 4 - 5

## Algoritmo en C++

### Diagrama de flujo



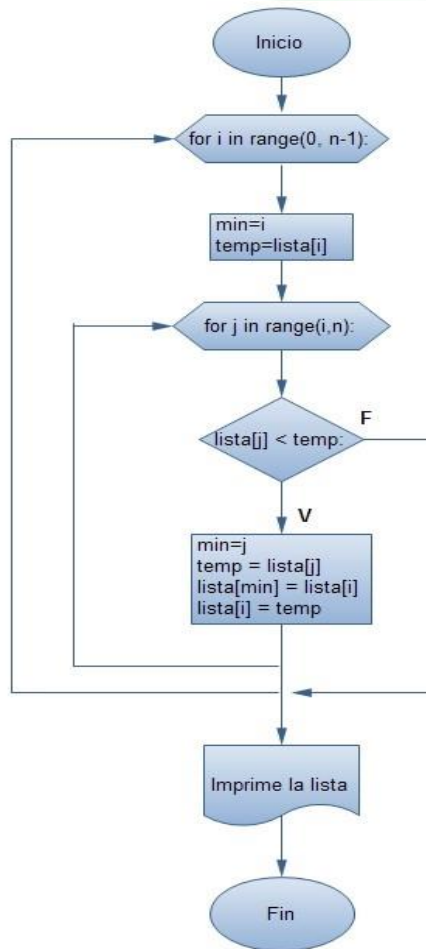
### Algoritmo

```
for(i=0 ; i<tam-1 ; i++){  
    min=i;  
    for(j=i+1 ; j<tam ; j++){  
        if (lista[min] > lista[j]) min=j;  
    }  
    temp=lista[min];  
    lista[min]=lista[i];  
    lista[i]=temp;  
}
```



## Algoritmo en Python

### Diagrama de flujo



### Algoritmo

```
for i in range(0, n-1):
    min=i
    temp=lista[i]
    for j in range(i,n):
        if lista[j] < temp:
            min=j
            temp = lista[j]
            lista[min] = lista[i]
            lista[i] = temp
```

## Complejidad en espacio

### En C++

Memoria estática: variables declaradas en el algoritmo.

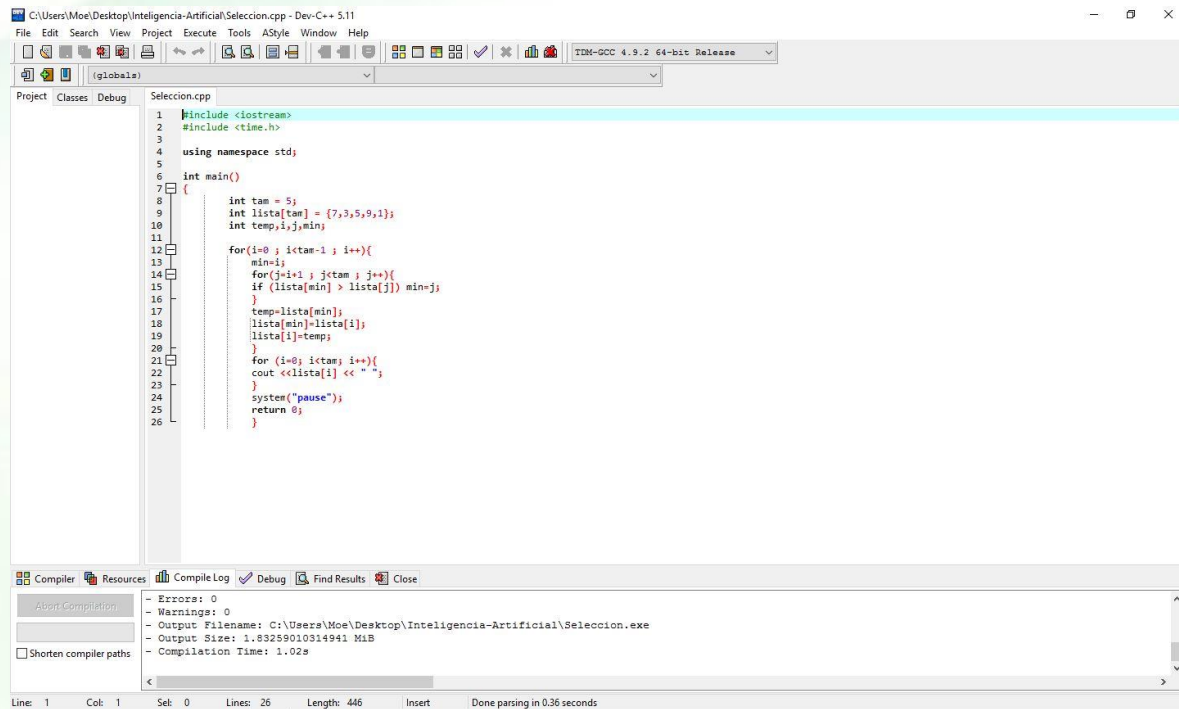
Int tam, lista, temp, min, i, j = 12 bytes

### En Python

Memoria dinámica: depende de cada ejecución del algoritmo.

## Ejecución del algoritmo

### En C++:



The screenshot shows a C++ IDE with the following code in `Seleccion.cpp`:

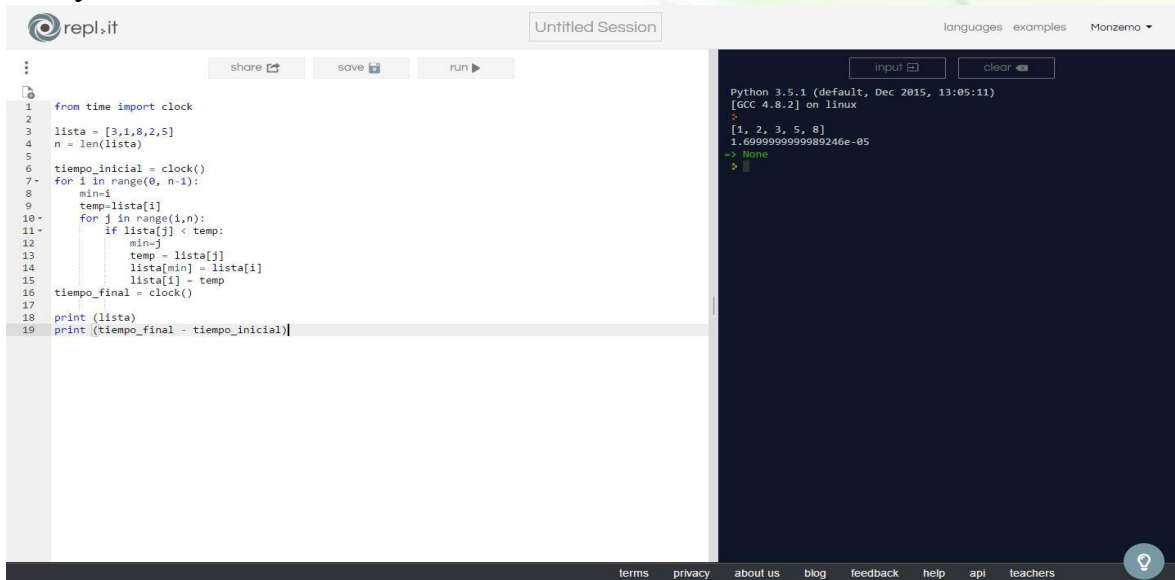
```
1 #include <iostream>
2 #include <time.h>
3
4 using namespace std;
5
6 int main()
7 {
8     int tam = 5;
9     int lista[tam] = {7,3,5,9,1};
10    int temp,i,j,min;
11
12    for(i=0 ; i<tam-1 ; i++){
13        min=i;
14        for(j=i+1 ; j<tam ; j++){
15            if (lista[min] > lista[j]) min=j;
16        }
17        temp=lista[min];
18        lista[min]=lista[i];
19        lista[i]=temp;
20    }
21    for (i=0; i<tam; i++){
22        cout << lista[i] << " ";
23    }
24    system("pause");
25    return 0;
26 }
```

The bottom panel shows the compilation output:

```
- Errors: 0
- Warnings: 0
- Output Filename: C:\Users\Moe\Desktop\Inteligencia-Artificial\Seleccion.exe
- Output Size: 1.83259010314941 MiB
- Compilation Time: 1.02s
```

At the bottom, it indicates: Line: 1 Col: 1 Sel: 0 Lines: 26 Length: 446 Insert Done parsing in 0.36 seconds

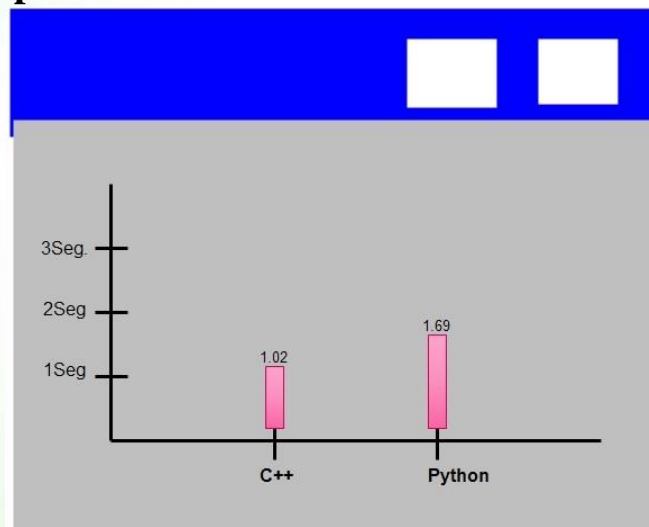
## En Python:



```
1 from time import clock
2 lista = [3,1,8,2,5]
3 n = len(lista)
4
5
6 tiempo_inicial = clock()
7 for i in range(0, n-1):
8     min=i
9     temp=lista[i]
10    for j in range(i,n):
11        if lista[j] < temp:
12            min=j
13            temp = lista[j]
14            lista[min] = lista[i]
15            lista[i] = temp
16    tiempo_final = clock()
17
18 print (lista)
19 print (tiempo_final - tiempo_inicial)
```

Python 3.5.1 (default, Dec 2015, 13:05:11)  
[GCC 4.8.2] on linux  
>  
[1, 2, 3, 5, 8]  
1.69999999999980246e-05  
=> None  
>

## Grafica de tiempos



## Conclusión

A diferencia del método de ordenamiento pasado (Ordenamiento por inserción) podría decirse que la velocidad entre ambos programas midiendo los tiempos de ejecución fue más amplia, una vez más siendo C++ quien lleva una pequeña ventaja con respecto a velocidad contra Python, en un vector pequeño podría decir que no se nota, pero conforme éste se hace más grande, ambos programas se hacen más lentos.

## Descripción del algoritmo

### *Ordenamiento burbuja (Bubblesort)*

Se basa en comparaciones sucesivas de dos elementos consecutivos y realizar un intercambio entre los elementos hasta que queden ordenados.

Para realizar la ordenación se han de seguir estos pasos:

- ✓ Se comparan los dos primeros elementos, si el segundo es superior al primero, se dejan tal como están, pero si el primero es el más grande, se intercambian los elementos.
- ✓ A continuación se compara el segundo elemento con el tercero aplicando los mismos criterios del paso anterior.
- ✓ De esta forma se repite la operación de comparación con todos los elementos que forman el vector. Cuando se alcance el último elemento se ha encontrado, el elemento que tiene el valor más elevado queda situado al final del vector.

Un ejemplo, ésta es nuestra lista a ordenar:

4 - 3 - 5 - 2 - 1

Tenemos 5 elementos. Es decir el tamaño toma el valor 5. Comenzamos comparando el primero con el segundo elemento. 4 es mayor que 3, así que intercambiamos. Ahora tenemos:

3 - 4 - 5 - 2 - 1

Ahora comparamos el segundo con el tercero: 4 es menor que 5, así que no hacemos nada. Continuamos con el tercero y el cuarto: 5 es mayor que 2. Intercambiamos y obtenemos:

3 - 4 - 2 - 5 - 1

Comparamos el cuarto y el quinto: 5 es mayor que 1. Intercambiamos nuevamente:

3 - 4 - 2 - 1 - 5

Repitiendo este proceso vamos obteniendo los siguientes resultados:

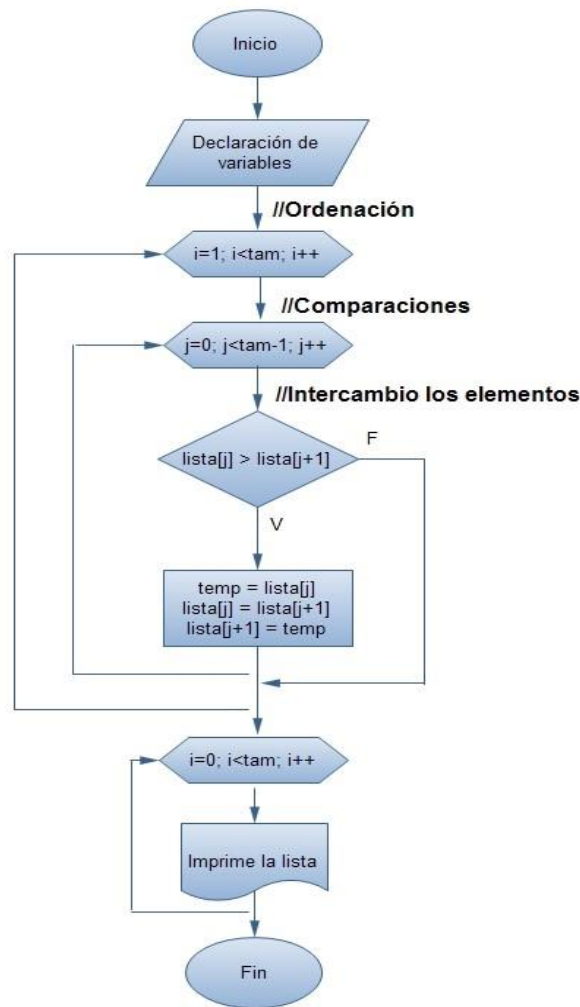
3 - 2 - 1 - 4 - 5

2 - 1 - 3 - 4 - 5

1 - 2 - 3 - 4 - 5

## Algoritmo en C++

### Diagrama de flujo



### Algoritmo

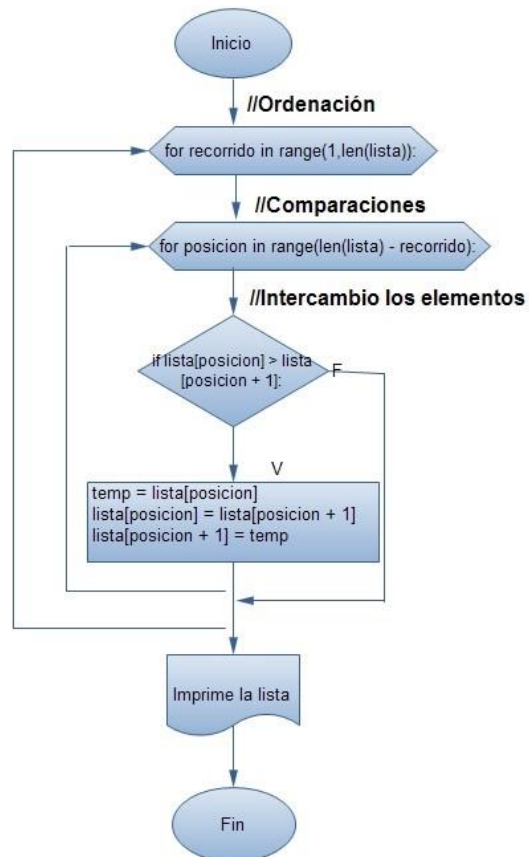
```
for (i=1; i<tam; i++){
    for (j=0; j<tam-1; j++){
        if (lista[j] > lista[j+1]){
            temp = lista[j];
            lista[j] = lista[j+1];
            lista[j+1] = temp;
        }
    }
}
```

### Variables

Nombre	Tipo	Uso
lista	int	Lista a ordenar
tam	int	Tamaño de la lista
i	int	Contador de ordenación
j	int	Contador de comparaciones
temp	int	Auxiliar para realizar los intercambios

## Algoritmo en Python

### Diagrama de flujo



### Algoritmo

```
for recorrido in range(1,len(lista)):
    for posicion in range(len(lista) - recorrido):
        if lista[posicion] > lista[posicion + 1]:
            temp = lista[posicion]
            lista[posicion] = lista[posicion + 1]
            lista[posicion + 1] = temp
```



## Complejidad en espacio

### En C++

Memoria estática: variables declaradas en el algoritmo.

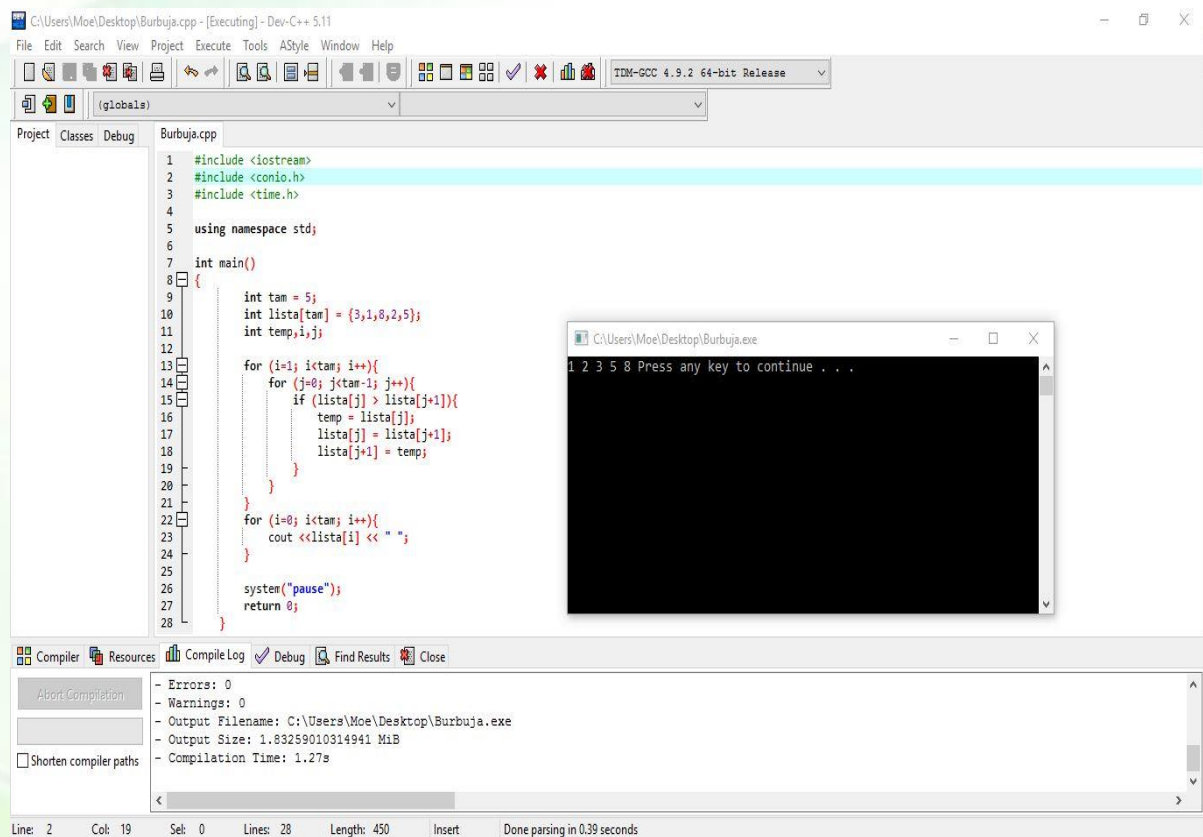
Int tam, lista, temp, i, j = 10 bytes

### En Python

Memoria dinámica: depende de cada ejecución del algoritmo.

## Ejecución del algoritmo

### En C++:



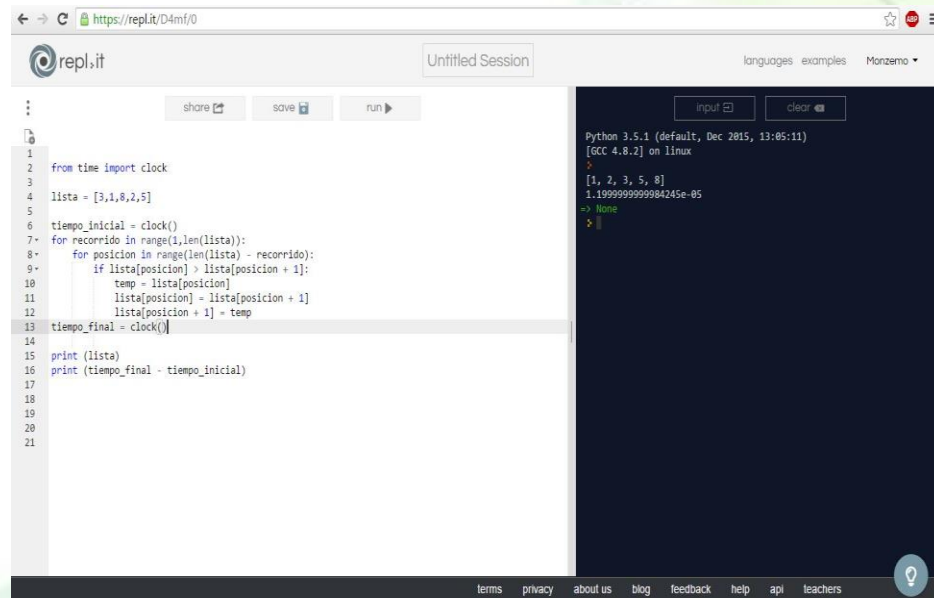
```
1 #include <iostream>
2 #include <conio.h>
3 #include <time.h>
4
5 using namespace std;
6
7 int main()
8 {
9     int tam = 5;
10    int lista[tam] = {3,1,8,2,5};
11    int temp,i,j;
12
13    for (i=1; i<tam; i++){
14        for (j=0; j<tam-i; j++){
15            if (lista[j] > lista[j+1]){
16                temp = lista[j];
17                lista[j] = lista[j+1];
18                lista[j+1] = temp;
19            }
20        }
21    }
22    for (i=0; i<tam; i++){
23        cout << lista[i] << " ";
24    }
25
26    system("pause");
27    return 0;
28 }
```

Compiler Output:

- Errors: 0
- Warnings: 0
- Output Filename: C:\Users\Moe\Desktop\Burbuja.exe
- Output Size: 1.83259010314941 MiB
- Compilation Time: 1.27s

Line: 2 Col: 19 Sel: 0 Lines: 28 Length: 450 Insert Done parsing in 0.39 seconds

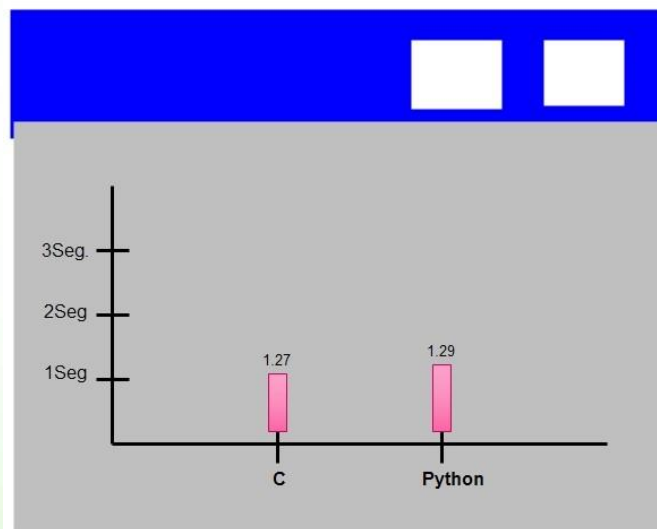
## En Python:



```
1 from time import clock
2
3
4 lista = [3,1,8,2,5]
5
6 tiempo_inicial = clock()
7 for recorrido in range(1,len(lista)):
8     for posicion in range(len(lista) - recorrido):
9         if lista[posicion] > lista[posicion + 1]:
10             temp = lista[posicion]
11             lista[posicion] = lista[posicion + 1]
12             lista[posicion + 1] = temp
13 tiempo_final = clock()
14
15 print(lista)
16 print(tiempo_final - tiempo_inicial)
17
18
19
20
21
```

Python 3.5.1 (default, Dec 2015, 13:05:11)  
[GCC 4.8.2] on linux  
>  
[1, 2, 3, 5, 8]  
1.1999999999984245e-05  
=> None  
>

## Grafica de tiempos



## Conclusión

Siendo éste método uno de los más sencillos y de fácil implementación, al momento de ejecutarlos podría decirse que la velocidad entre ambos programas midiendo los tiempos de ejecución fue más corta que los anteriores, pero al igual que los otros, con más datos se vuelven más lentos.

## Descripción del algoritmo

### *Ordenamiento por mezcla o Mergesort*

Sigue el principio de divide y vencerás algoritmo de clasificación. Divide matriz dada en dos mitades, llama a sí misma para las dos mitades y luego se fusionaron las dos mitades ordenadas.

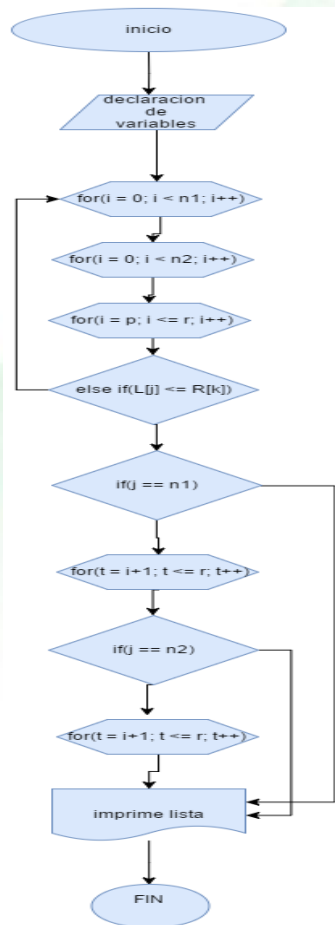
Este método se basa en la siguiente idea:

1. Si la lista es pequeña (vacía o de tamaño 1) ya está ordenada y no hay nada que hacer. De lo contrario hacer lo siguiente:
2. Dividir la lista al medio, formando dos sublistas de (aproximadamente) el mismo tamaño cada una.
3. Ordenar cada una de esas dos sublistas (usando este mismo método).
4. Una vez que se ordenaron ambas sublistas, intercalarlas de manera ordenada.

Por ejemplo, si la lista original es [6, 7, -1, 0, 5, 2, 3, 8] deberemos ordenar recursivamente [6, 7, -1, 0] y [5, 2, 3, 8] con lo cual obtendremos [-1, 0, 6, 7] y [2, 3, 5, 8]. Si intercalamos ordenadamente las dos listas ordenadas obtenemos la solución buscada: [-1, 0, 2, 3, 5, 6, 7, 8].

## Algoritmo en C++

### Diagrama de flujo



### Algoritmo

merge\_sort (arr [], p, r)

Si  $p < r$

1. Encontrar el medio (digamos q) de la matriz  
 $q = (p + r) / 2$
2. Llame merge\_sort para la primera mitad:  
merge\_sort (arr, p, q)
3. Llame merge\_sort para la segunda parte:  
merge\_sort (arr, q + 1, r)
4. Combinar las dos mitades ordenadas en el paso 2 y 3:  
fusionar (arr, p, q, r)

La función merge () se utiliza para la fusión de las dos mitades. La combinación de (a, p, q, r) es un proceso clave que asume que una a[p..q] y la matriz [a[q + 1..r] se clasifican y fusiona los dos sub-conjuntos ordenados en una sola.

## Complejidad en espacio

### En C++

Memoria estática: variables declaradas en el algoritmo.

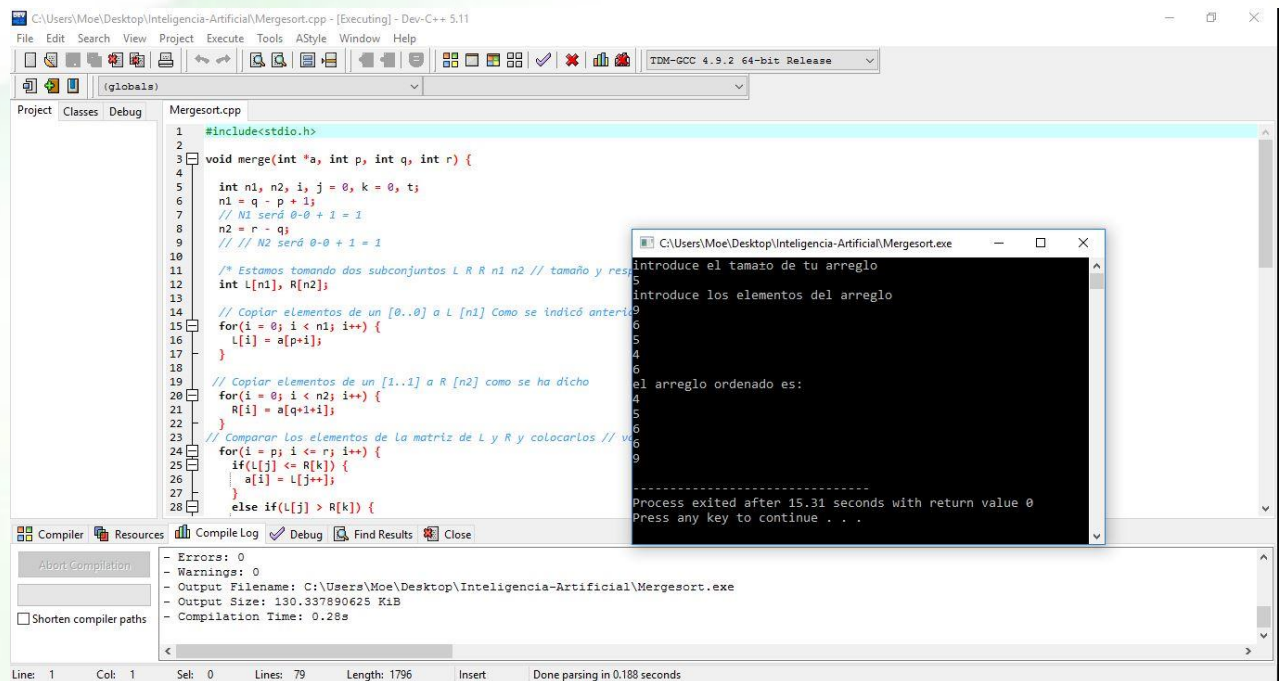
Int n1, n2, i, j, k, t, q = 14 bytes

### En Python

Memoria dinámica: depende de cada ejecución del algoritmo.

## Ejecución del algoritmo

### En C++:



The screenshot displays a C++ IDE with the MergeSort algorithm code in `Mergesort.cpp`. The code includes standard headers and defines a `merge` function that recursively sorts an array. Comments in Spanish describe the process of dividing the array into two halves and merging them back in sorted order. The IDE's output window shows the program's execution, where the user is prompted to enter the size of the array (5) and its elements (5, 4, 3, 2, 1). The output displays the sorted array (1, 2, 3, 4, 5) and confirms that the process exited successfully after 15.31 seconds.

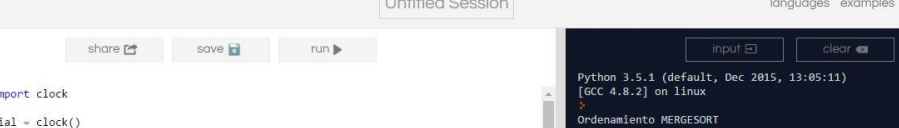
```
1 #include<stdio.h>
2
3 void merge(int *a, int p, int q, int r) {
4
5     int n1, n2, i, j = 0, k = 0, t;
6     n1 = q - p + 1;
7     // N1 será 0-0 + 1 = 1
8     n2 = r - q;
9     // N2 será 0-0 + 1 = 1
10
11     /* Estamos tomando dos subconjuntos L R R n1 n2 // tamaño y resp
12     int L[n1], R[n2];
13
14     // Copiar elementos de un [0..0] a L [n1] Como se indicó anterio
15     for(i = 0; i < n1; i++) {
16         L[i] = a[p+i];
17     }
18
19     // Copiar elementos de un [1..1] a R [n2] como se ha dicho
20     for(i = 0; i < n2; i++) {
21         R[i] = a[q+1+i];
22     }
23
24     // Comparar los elementos de la matriz de L y R y colocarlos // vo
25     for(i = p; i <= r; i++) {
26         if(L[j] <= R[k]) {
27             a[i] = L[j++];
28         }
29         else if(L[j] > R[k]) {
30             a[i] = R[k++];
31         }
32     }
33 }
```

introduce el tamaño de tu arreglo  
5  
Introduce los elementos del arreglo  
5  
4  
3  
2  
1  
el arreglo ordenado es:  
1  
2  
3  
4  
5  
-----  
Process exited after 15.31 seconds with return value 0  
Press any key to continue . . .

Compiler: TDM-GCC 4.9.2 64-bit Release  
Errors: 0  
Warnings: 0  
Output Filename: C:\Users\Moe\Desktop\Inteligencia-Artificial\Mergesort.exe  
Output Size: 130.337890625 KiB  
Compilation Time: 0.28s



*En Python:*



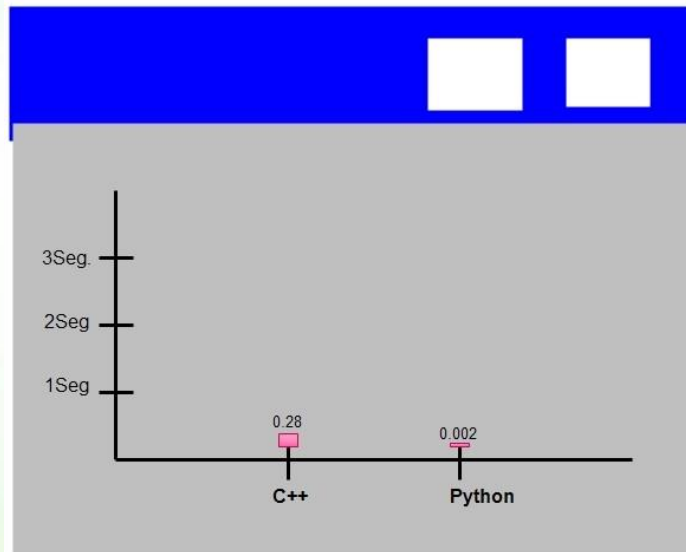
```
from time import clock
1
2
3 tiempo_inicial = clock()
4 def MergeSort(A):
5     #print("Dividir: ",A)
6     if len(A)>1:
7         mitad = len(A)//2
8         mitadizq = A[:mitad]
9         mitadder = A[mitad:]
10        MergeSort(mitadizq)
11        MergeSort(mitadder)
12        i=0
13        j=0
14        k=0
15        while i < len(mitadizq) and j < len(mitadder):
16            if mitadizq[i] < mitadder[j]:
17                A[k]=mitadizq[i]
18                i=i+1
19            else:
20                A[k]=mitadder[j]
21                j=j+1
22                k=k+1
23
24        while i < len(mitadizq):
25            A[k]=mitadizq[i]
26            i=i+1
27            k=k+1
28
29        while j < len(mitadder):
30            A[k]=mitadder[j]
31            j=j+1
32            k=k+1
33
34    tiempo_final = clock()
35    print("Tiempo de ejecucion: ", tiempo_final - tiempo_inicial)
```

Python 3.5.1 (default, Dec 2015, 13:05:11)  
[GCC 4.8.2] on linux

Ordenamiento MERGESORT  
Numeros a Ordenar: [5, 6, 8, 3]  
Numeros Ordenados son: [3, 5, 6, 8]  
0.00020300000000000873

>> None

## Grafica de tiempos



## Conclusión

Como se conoce este algoritmo pertenece a la clasificación de divide y vencerás, la peculiaridad que tiene este algoritmo es que va dividiendo el arreglo a la mitad y así sucesivamente, mitad tras mitad va ordenando, funciona, pero dado el caso que si el arreglo fuese mayor a 500 o más caracteres este algoritmo se torna un poco ineficaz (lento), en comparación de sus demás compañeros (algoritmos de ordenación), ya que funciona para una cierta cantidad de caracteres; entonces ya que dividido las partes, las junta las partes ordenadas.



## Descripción del algoritmo

### *Ordenamiento rápido o Quicksort*

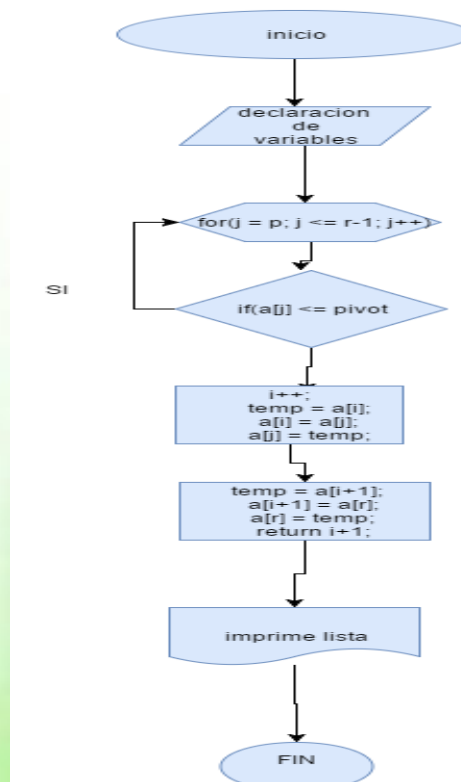
Quicksort es un algoritmo de divide y vencerás. Quicksort divide en primer lugar una gran variedad en dos más pequeños sub-series: los elementos bajos y los altos elementos. Ordenación rápida se puede ordenar de forma recursiva las sub-series.

Los pasos son los siguientes:

1. Elija un elemento, llamado un pivote, desde la matriz.
2. Reordenar la matriz de modo que todos los elementos con valores menores que el pivote vienen antes de que el pivote, mientras que todos los elementos con valores mayores que el pivote venir después de él (valores iguales pueden ir en cualquier dirección). Después de esta partición, el pivote está en su posición final. Esto se llama el partitionoperation.
3. Recursiva aplicar los pasos anteriores para la sub-serie de elementos con valores más pequeños y por separado a la sub-serie de elementos con valores mayores

## Algoritmo en C++

### *Diagrama de flujo*



### *Algoritmo*

```
for(j = p; j <= r-1; j++) {  
    // Si el elemento actual es menor que o igual a pivote  
    if(a[j] <= pivot) {  
        i++;  
        temp = a[i];  
        a[i] = a[j];  
        a[j] = temp;  
    }  
}  
temp = a[i+1];  
a[i+1] = a[r];  
a[r] = temp;  
return i+1;  
}
```

### **Algoritmo en Python**

#### *Algoritmo*

```
def quicksort(L,start,stop):  
    if stop - start < 2: return  
    key = L[R.randrange(start,stop)]  
  
    e = u = start  
    g = stop  
    while u < g:  
        if L[u] < key:  
            swap(L,u,e)  
            e = e + 1  
            u = u + 1  
        elif L[u] == key:  
            u = u + 1  
        else:  
            g = g - 1  
            swap(L,u,g)  
    quicksort(L,start,e)  
    quicksort(L,g,stop)
```

## Complejidad en espacio

### En C++

Memoria estática: variables declaradas en el algoritmo.

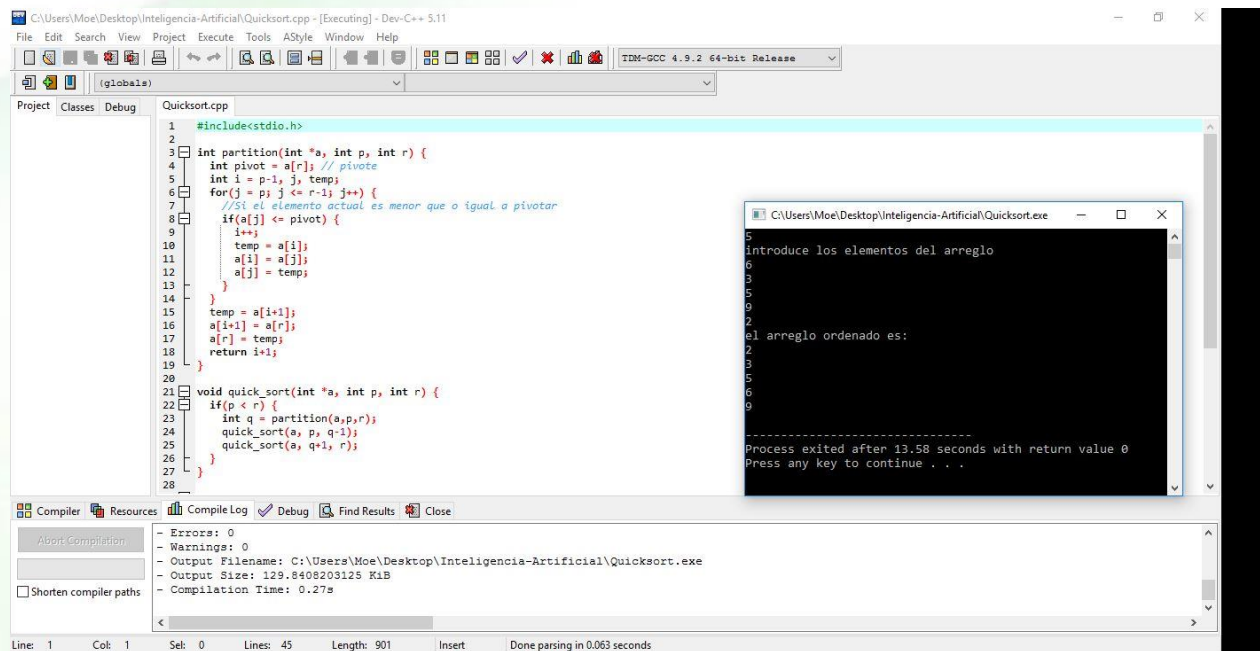
Int i, p, j, temp, pivot, n, \*a, r = 16 bytes

### En Python

Memoria dinámica: depende de cada ejecución del algoritmo.

## Ejecución del algoritmo

### En C++:



```
1 #include<stdio.h>
2
3 int partition(int *a, int p, int r) {
4     int pivot = a[r]; // pivote
5     int i = p-1, j, temp;
6     for(j = p; j <= r-1; j++) {
7         //Si el elemento actual es menor que o igual a pivotar
8         if(a[j] <= pivot) {
9             i++;
10            temp = a[i];
11            a[i] = a[j];
12            a[j] = temp;
13        }
14    }
15    temp = a[i+1];
16    a[i+1] = a[r];
17    a[r] = temp;
18    return i+1;
19 }
20
21 void quick_sort(int *a, int p, int r) {
22     if(p < r) {
23         int q = partition(a,p,r);
24         quick_sort(a, p, q-1);
25         quick_sort(a, q+1, r);
26     }
27 }
28
```

```
5
6 Introduce los elementos del arreglo
7 3
8 5
9 2
10
11 el arreglo ordenado es:
12 2
13 3
14 5
15
16 -----
17 Process exited after 13.58 seconds with return value 0
18 Press any key to continue . . .
```

## En Python:

```
repl.it
Untitled Session
languages examples Monzemo

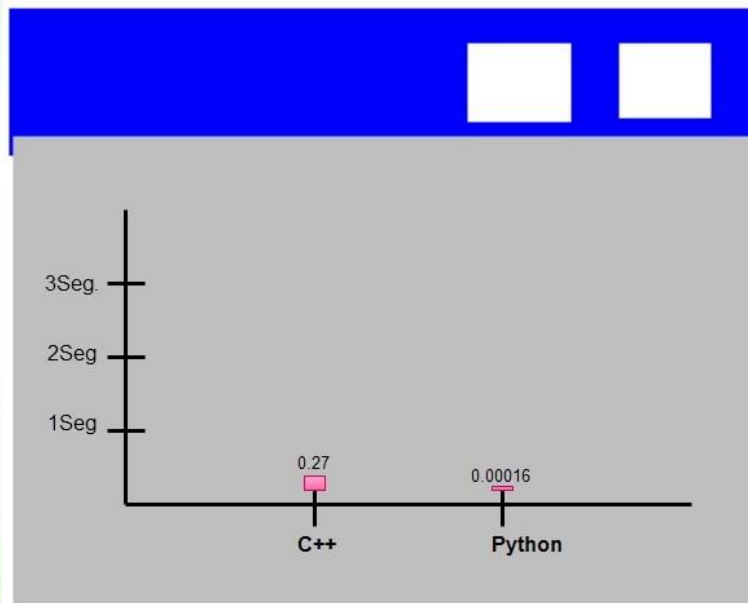
share save run

key = L[n+random.randint(0, len(L)-1)]
13
14 e = u = start
15 g = stop
16 while u < g:
17     if L[u] < key:
18         swap(L,u,e)
19         e = e + 1
20         u = u + 1
21     elif L[u] == key:
22         u = u + 1
23     else:
24         g = g - 1
25         swap(L,u,g)
26         quicksort(L,start,e)
27         quicksort(L,g,stop)
28
29 def swap(A,i,j):
30     temp = A[i]
31     A[i] = A[j]
32     A[j] = temp
33
34
35 print("quicksort:\n")
36 L = [3,1,4,1,5,9,2,6,5,3,5,8,9,7,9]
37 print("tu arreglo desordenado era:\n",L)
38 qsort(L)
39 tiempo_final = clock()
40 print('tu arreglo ordenado es:\n',L)
41 print(tiempo_final - tiempo_inicial)
```

```
Python 3.5.1 (default, Dec 2015, 13:05:11)
[GCC 4.8.2] on linux
>
quicksort:
tu arreglo desordenado era:
[3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9, 7, 9]
tu arreglo ordenado es:
[1, 1, 2, 3, 3, 4, 5, 5, 5, 6, 7, 8, 9, 9, 9]
0.00016700000000000048
-> None
>
```

terms privacy about us blog feedback help api teachers

## Grafica de tiempos



## Conclusión

Sabemos que es uno de los algoritmos de ordenación, lo que hace este algoritmo elegir un elemento de la lista al que se le llama pivote, de tal manera que va preguntando por la lista quienes son menores que y los mayores al pivote, en este caso los menores los manda a la izquierda y los mayores a la derecha, pero no lo hace solo con ayuda de punteros cada uno en los extremos de la lista que lo ayudan a preguntar y comparar. Del punto de vista es un poco más rápido que el Mergesort.

## Descripción del algoritmo

Esta clasificación es una técnica basada en la clasificación de comparación basado en la estructura de datos binarios. Es similar a una especie de selección en el que en primer lugar el elemento máximo y colocar el elemento máximo al final.(árboles)

Repetimos el mismo proceso para el elemento restante. Primero vamos a definir un árbol binario completo. Un árbol binario completo es un árbol binario en el que todos los niveles, con la posible excepción de la última, están completamente llenos, y todos los nodos están tan a la izquierda como sea posible.

Un árbol binario completo donde los artículos se almacenan en un orden especial de tal manera que el valor de un nodo padre es mayor (o menor) que los valores en sus dos nodos hijos. El primero se llama como máximo de almacenamiento dinámico y el segundo se llama min montón.

Puede ser representado por árbol binario o array. ¿Por qué la representación basada en matrices para binario del montón? Desde un Binario Montón es un árbol binario completo, que puede representarse fácilmente como matriz y representación basada matriz es eficiente en espacio.

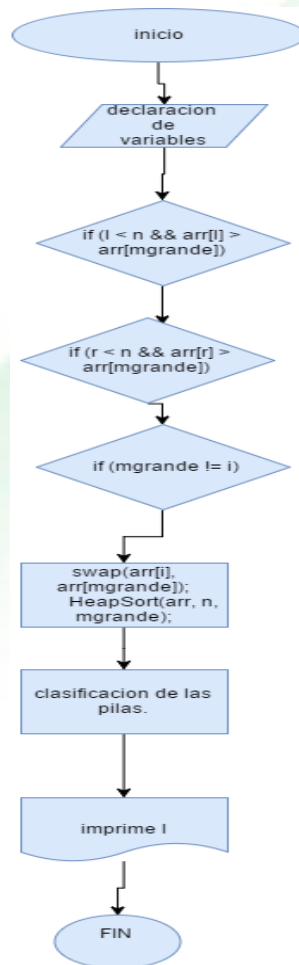
Si el nodo padre se almacena en el índice  $i$ , el hijo izquierdo se puede calcular por  $2 * i + 1$  y el hijo derecho por  $2 * i + 2$ .

Pila de clasificación Algoritmo de selección creciente:

1. Construir un máximo de almacenamiento dinámico a partir de los datos de entrada.
2. En este punto, la partida más importante se almacena en la base de la pila. Reemplazarlo con el último elemento de la pila seguido de la reducción del tamaño de almacenamiento dinámico a
1. Finalmente, heapify la raíz del árbol.
3. Repita los pasos anteriores hasta que el tamaño del montón es mayor que 1.

## Algoritmo en C++

### Diagrama de flujo



### Algoritmo

1. Se construye el montículo inicial a partir del arreglo original.
2. Se intercambia la raíz con el último elemento del montículo.
3. El último elemento queda ordenado.
4. El último elemento se saca del montículo, no del arreglo.
5. Se restaura el montículo haciendo que el primer elemento baje a la posición que le corresponde, si sus hijos son menores.
6. La raíz vuelve a ser el mayor del montículo.
7. Se repite el paso 2 hasta que quede un solo elemento en el montículo.



## Complejidad en espacio

### En C++

Memoria estática: variables declaradas en el algoritmo.

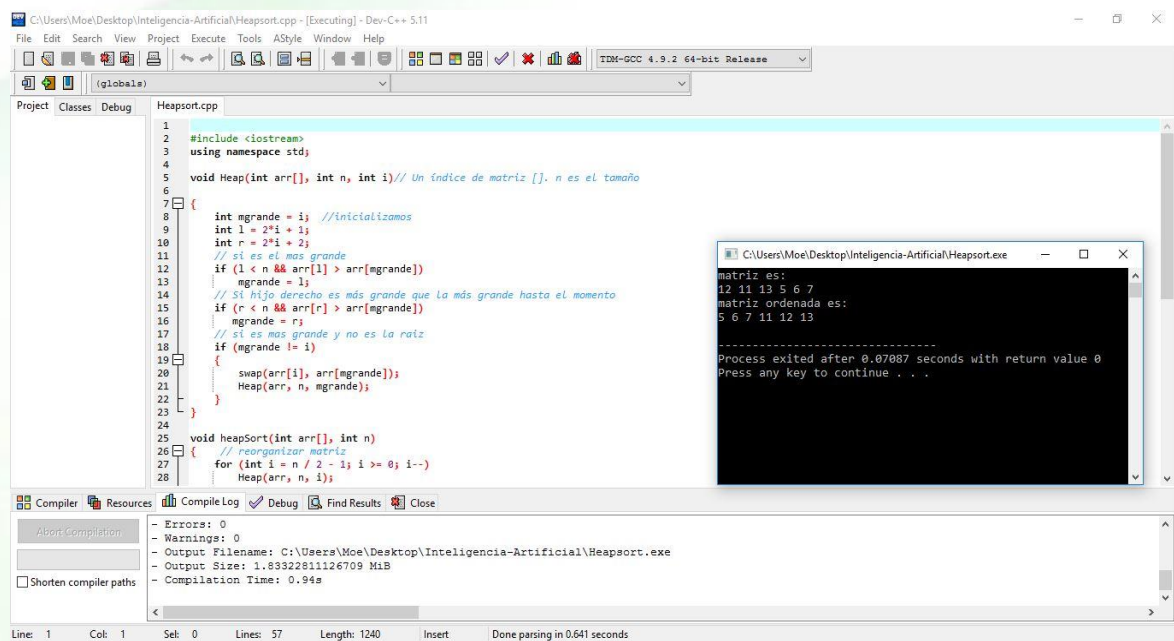
Int arr, n, i, mgrande, l, r = 12 bytes

### En Python

Memoria dinámica: depende de cada ejecución del algoritmo.

## Ejecución del algoritmo

### En C++:



The screenshot displays a C++ IDE with the following components:

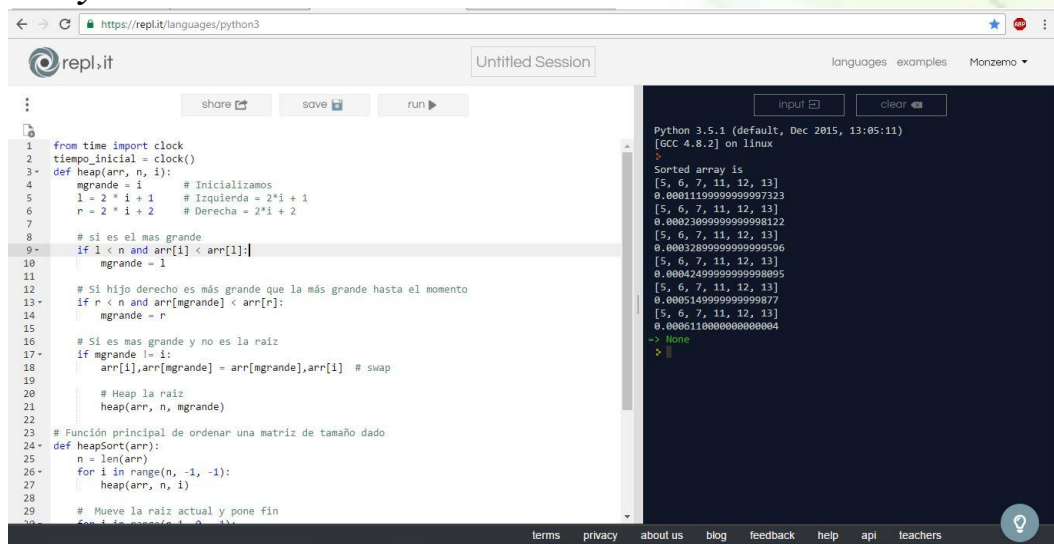
- Source Code (Heapsort.cpp):**

```
1
2 #include <iostream>
3 using namespace std;
4
5 void Heap(int arr[], int n, int i) // Un índice de matriz []. n es el tamaño
6
7 {
8     int mgrande = i; //inicializamos
9     int l = 2*i + 1;
10    int r = 2*i + 2;
11    // si es el mas grande
12    if (l < n && arr[l] > arr[mgrande])
13        mgrande = l;
14    // Si hijo derecho es más grande que la más grande hasta el momento
15    if (r < n && arr[r] > arr[mgrande])
16        mgrande = r;
17    // si es mas grande y no es la raíz
18    if (mgrande != i)
19    {
20        swap(arr[i], arr[mgrande]);
21        Heap(arr, n, mgrande);
22    }
23 }
24
25 void heapSort(int arr[], int n)
26 {
27     // reorganizar matriz
28     for (int i = n / 2 - 1; i >= 0; i--)
29         Heap(arr, n, i);
30 }
```
- Compiler Output:**
  - Errors: 0
  - Warnings: 0
  - Output Filename: C:\Users\Moe\Desktop\Inteligencia-Artificial\Heapsort.exe
  - Output Size: 1.8332281126709 MiB
  - Compilation Time: 0.94s
- Execution Output (Heapsort.exe):**

```
matriz es:
12 11 13 5 6 7
matriz ordenada es:
5 6 7 11 12 13

-----
Process exited after 0.07087 seconds with return value 0
Press any key to continue . . .
```

## En Python:

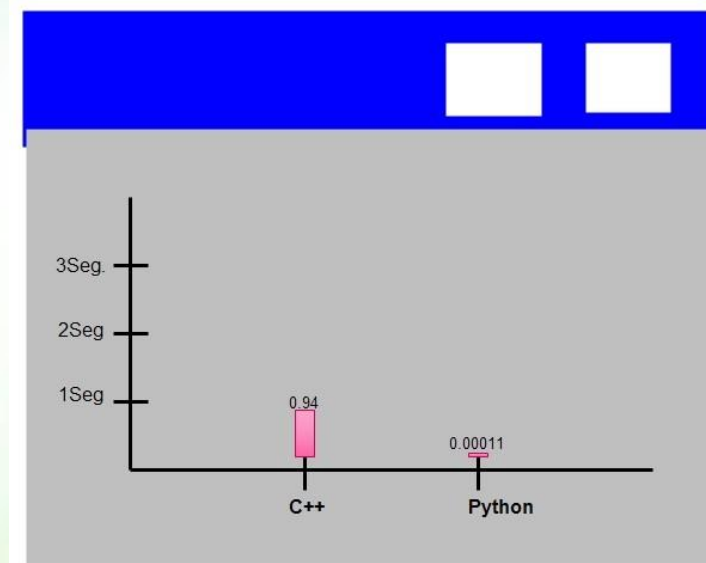


The screenshot shows a Repl.it Python environment. The left pane contains a Python script for a heap sort algorithm. The right pane shows the output of the script, which prints the sorted array and the time taken for each step.

```
1 from time import clock
2 tiempo_inicial = clock()
3 def heap(arr, n, i):
4     mgrande = i # Inicializamos
5     l = 2 * i + 1 # Izquierda = 2*i + 1
6     r = 2 * i + 2 # Derecha = 2*i + 2
7
8     # si es el mas grande
9     if l < n and arr[l] < arr[i]:
10         mgrande = l
11
12     # Si hijo derecho es más grande que la más grande hasta el momento
13     if r < n and arr[r] < arr[mgrande]:
14         mgrande = r
15
16     # Si es mas grande y no es la raíz
17     if mgrande != i:
18         arr[i], arr[mgrande] = arr[mgrande], arr[i] # swap
19
20     # Heap la raíz
21     heap(arr, n, mgrande)
22
23 # Función principal de ordenar una matriz de tamaño dado
24 def heapSort(arr):
25     n = len(arr)
26     for i in range(n, -1, -1):
27         heap(arr, n, i)
28
29 # Mueve la raíz actual y pone fin
30 for i in range(n, -1, -1):
```

Python 3.5.1 (default, Dec 2015, 13:05:11)  
[GCC 4.8.2] on linux  
Sorted array is  
[5, 6, 7, 11, 12, 13]  
0.000111999999999997323  
[5, 6, 7, 11, 12, 13]  
0.000230999999999998122  
[5, 6, 7, 11, 12, 13]  
0.00032899999999999596  
[5, 6, 7, 11, 12, 13]  
0.00042499999999999805  
[5, 6, 7, 11, 12, 13]  
0.00051499999999999877  
[5, 6, 7, 11, 12, 13]  
0.0006110000000000004  
-> None

## Grafica de tiempos



## Conclusión

El Heapsort está basado en el uso de un tipo especial de árbol binario. La estructura de ramificación del árbol conserva el número de comparaciones necesarias en  $n \log n$ .

Ventajas: Su desempeño es en promedio tan bueno como el Quicksort y se comporta mejor que este último en los peores casos.

Desventajas: Aunque el Heapsort tiene un mejor desempeño general que cualquier otro método presentado de clasificación interna, es bastante complejo de programar.

