

INTELIGENCIA ARTIFICIAL

Métodos de ordenamiento

Instituto Tecnológico de Ensenada

Profesor: Ing. Alejandro Chávez Sánchez

Alumno: Morales Hernández M.

Álvarez Sandoval Igor

Ingeniería en Sistemas Computacionales

**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-esimo 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

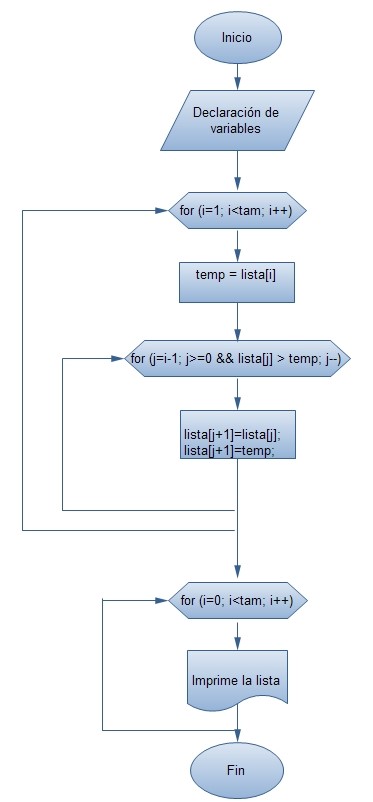
A[3] < A[2] 08 < 15 Si hay intercambio

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

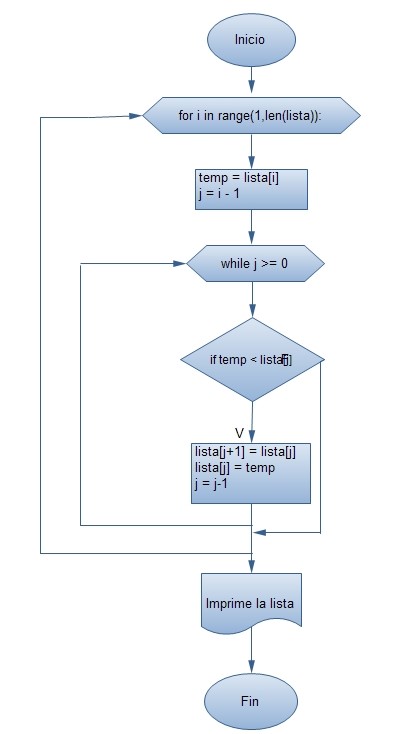
}

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.

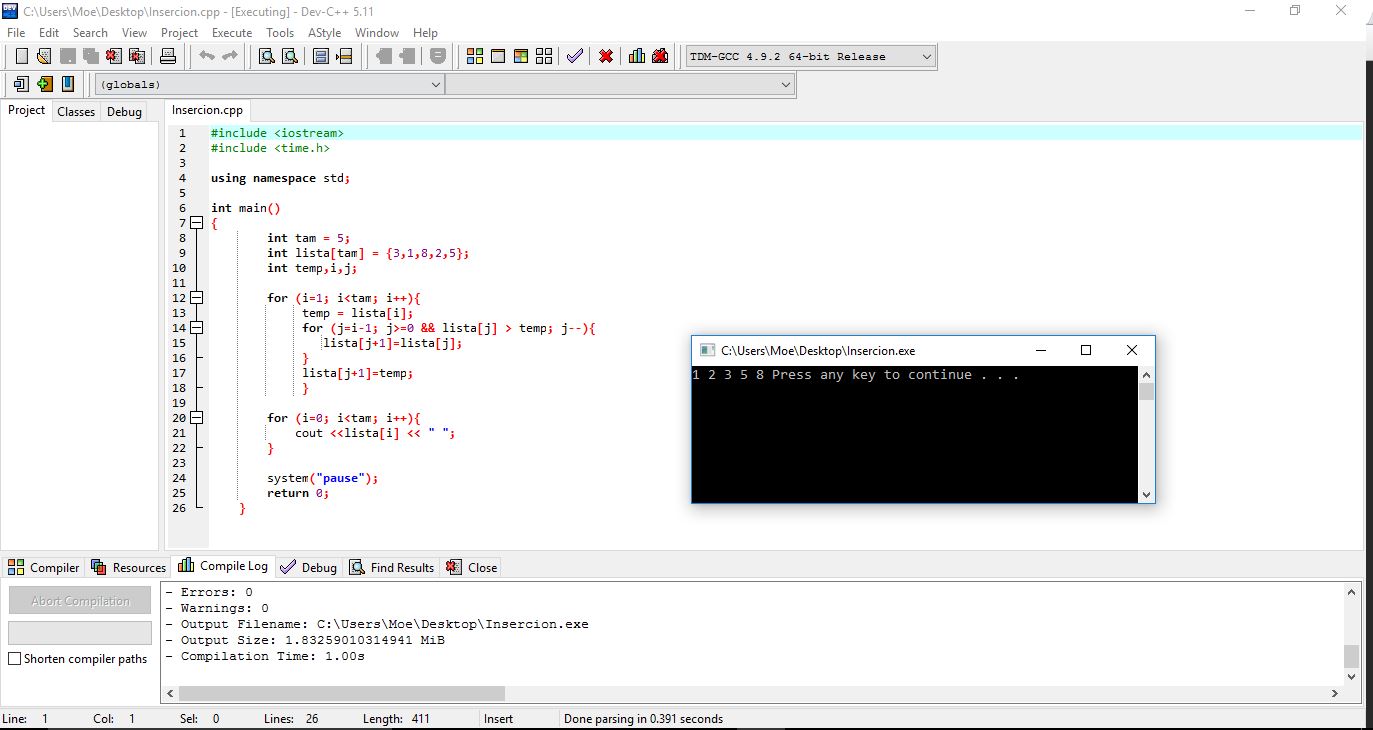
Int tam, lista, temp, i, j = 10 bytes

***En Python***

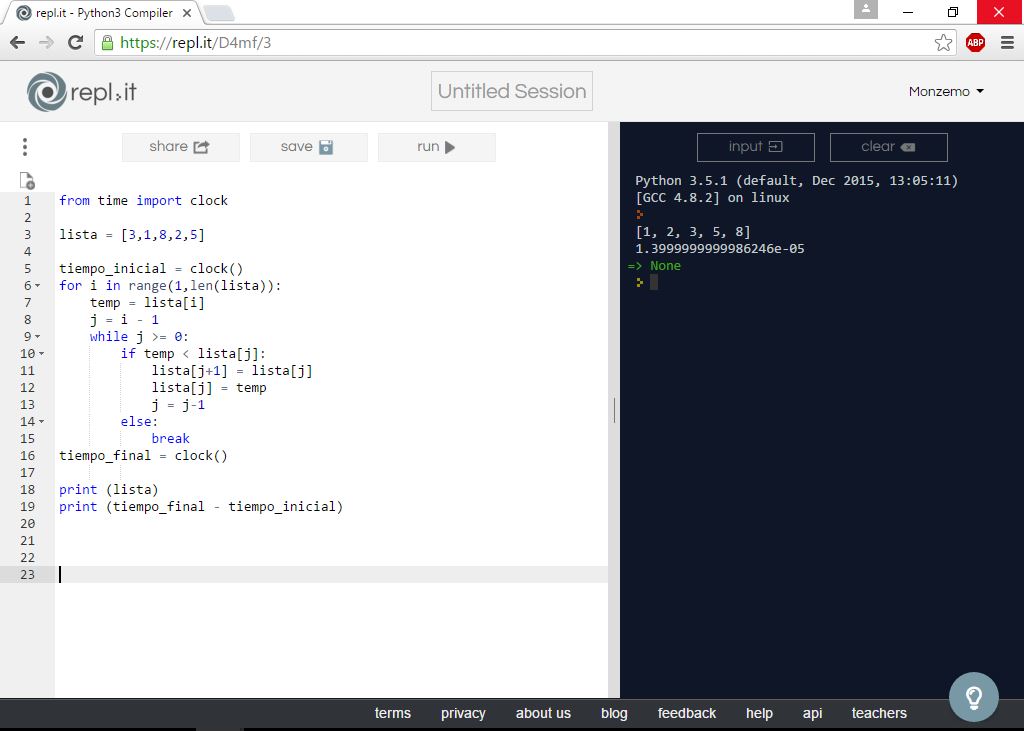
Memoria dinámica: depende de cada ejecución del algoritmo.

**Ejecución del algoritmo**

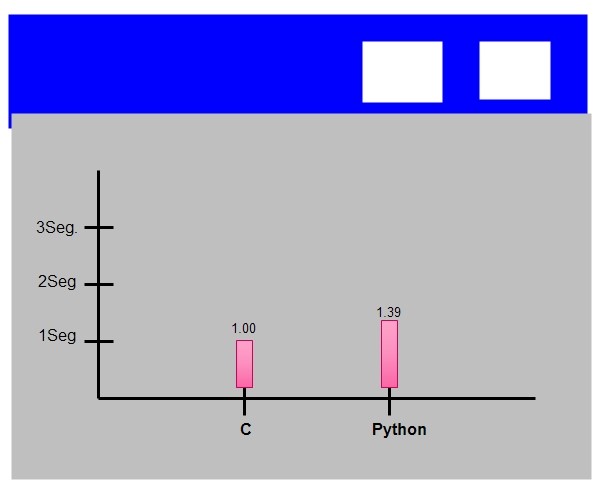
***En C++:***



***En Python:***



**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:

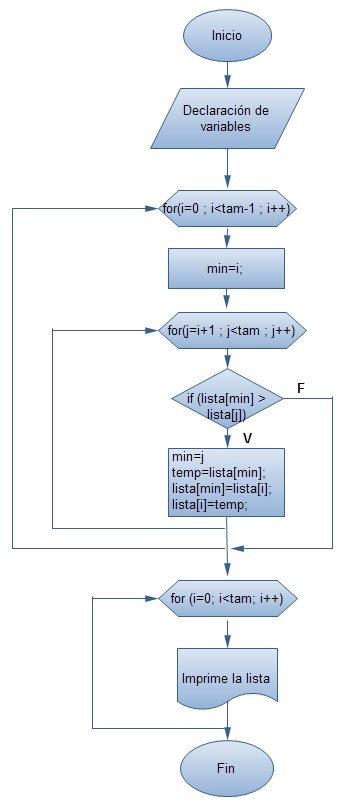
1 - 2 - 3 - 5 – 4

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];

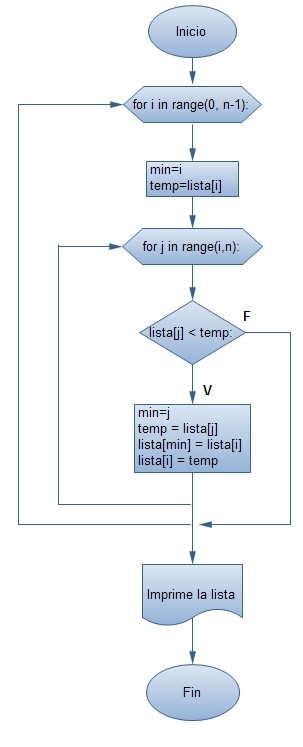
lista[min]=lista[i];

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.

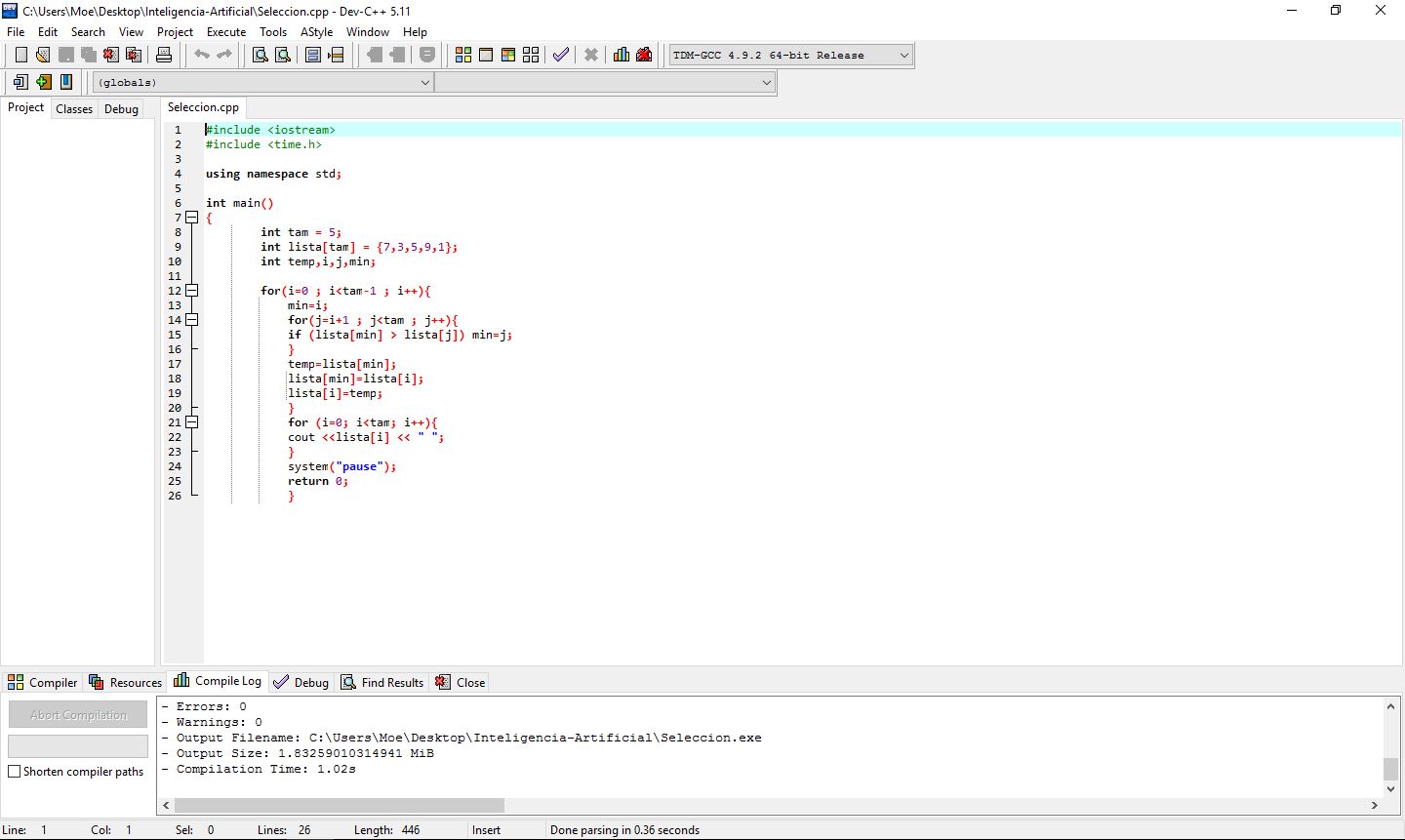
Int tam, lista, temp, min, i, j = 12 bytes

***En Python***

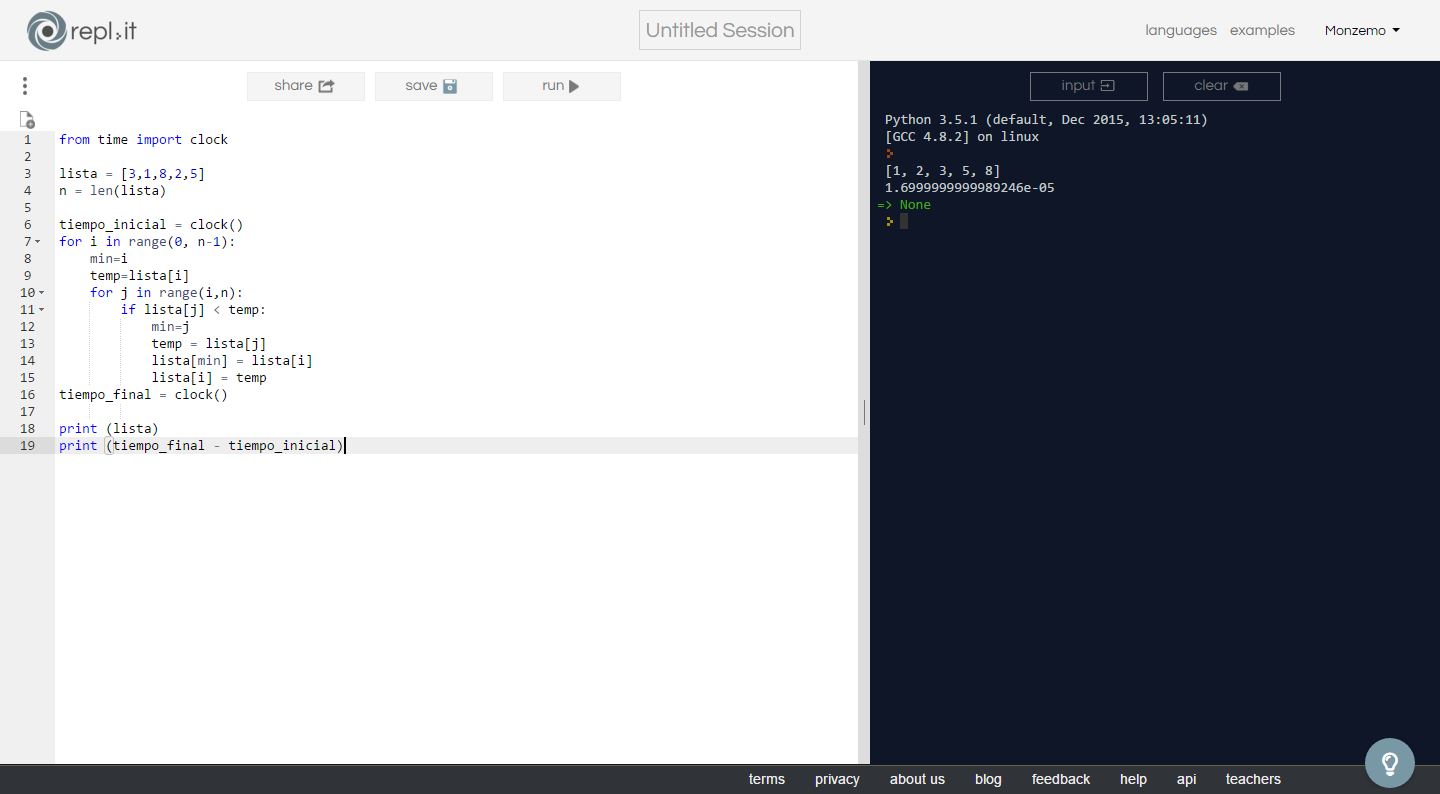
Memoria dinámica: depende de cada ejecución del algoritmo.

**Ejecución del algoritmo**

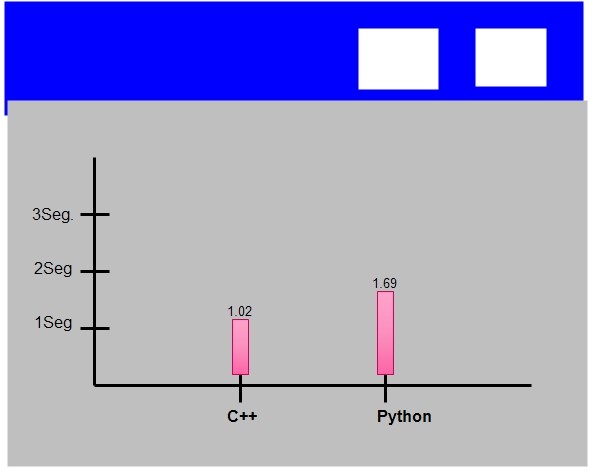
***En C++:***



***En Python:***



**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:

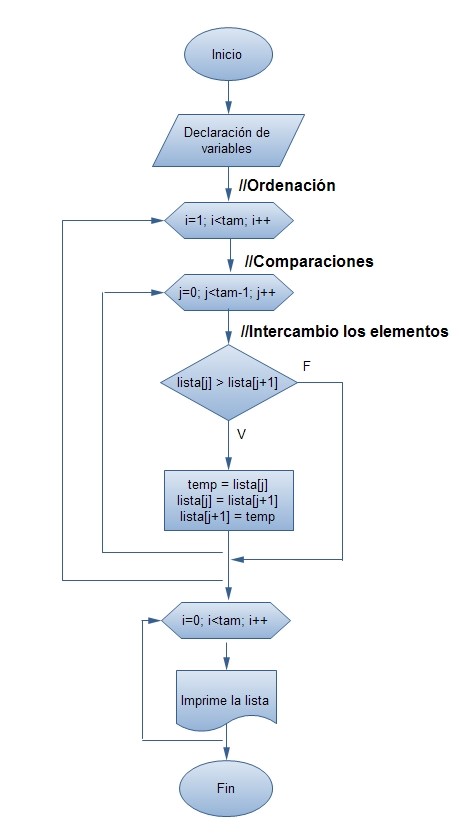
3 - 2 - 1 - 4 - 5

2 - 1 - 3 - 4 - 5

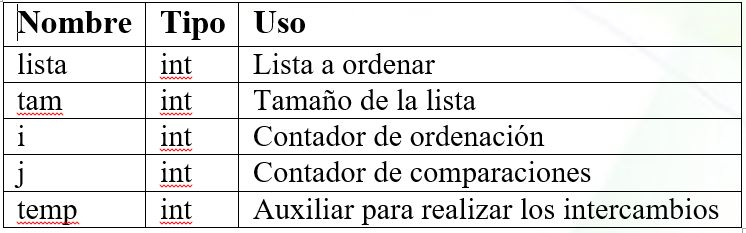
1 - 2 - 3 - 4 – 5

**Algoritmo en C++**

***Diagrama de flujo***

****

***Algoritmo Variables***

******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;

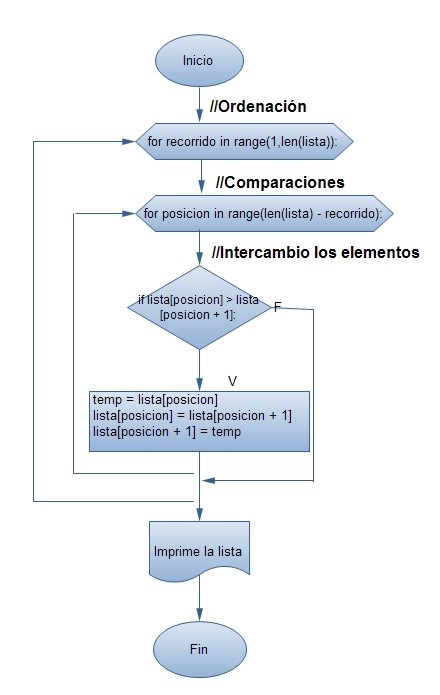
}

}

}

**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++***

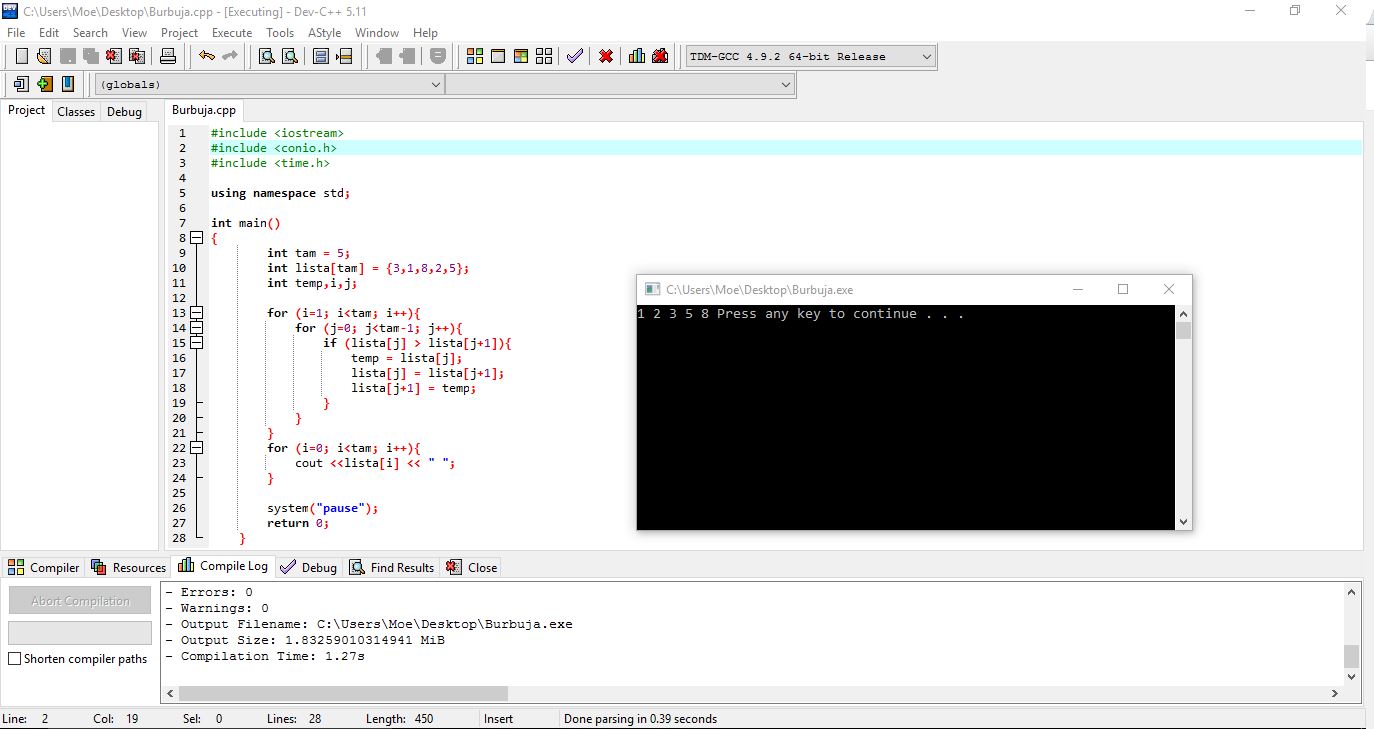
Memoria estática: variables declaradas en el algoritmo.

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

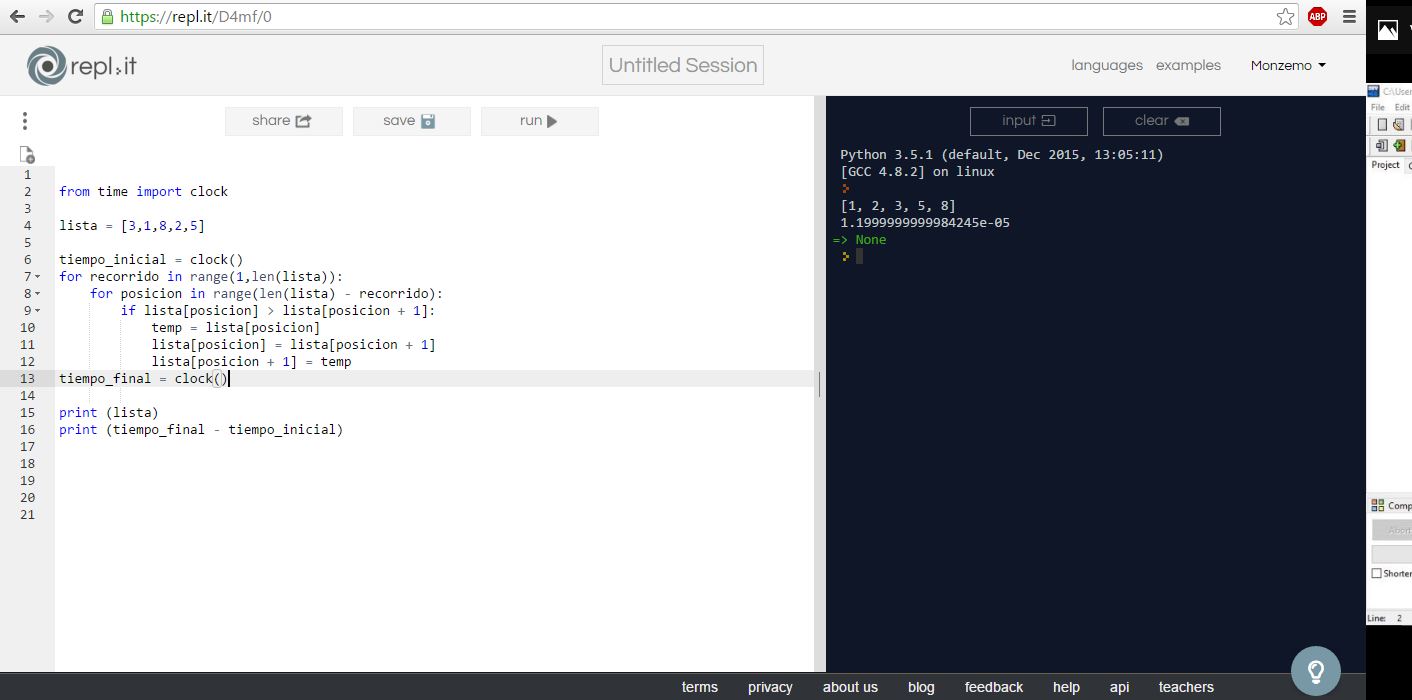
***En Python***

Memoria dinámica: depende de cada ejecución del algoritmo.

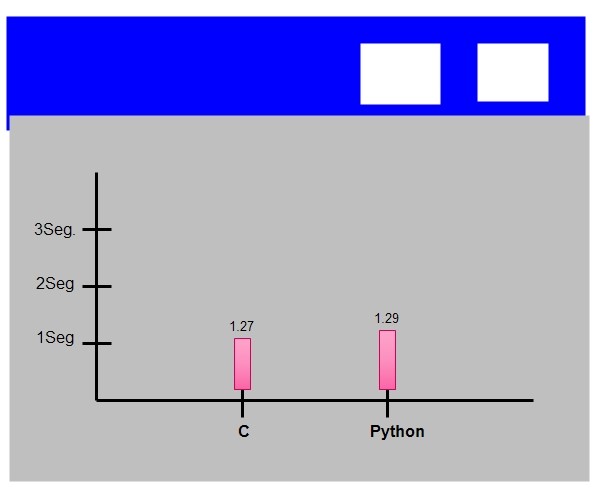
**Ejecución del algoritmo**

***En C++:***

***En Python:***

******

**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++**

***Untitled Diagram (3)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 merg () 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 [+ 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.

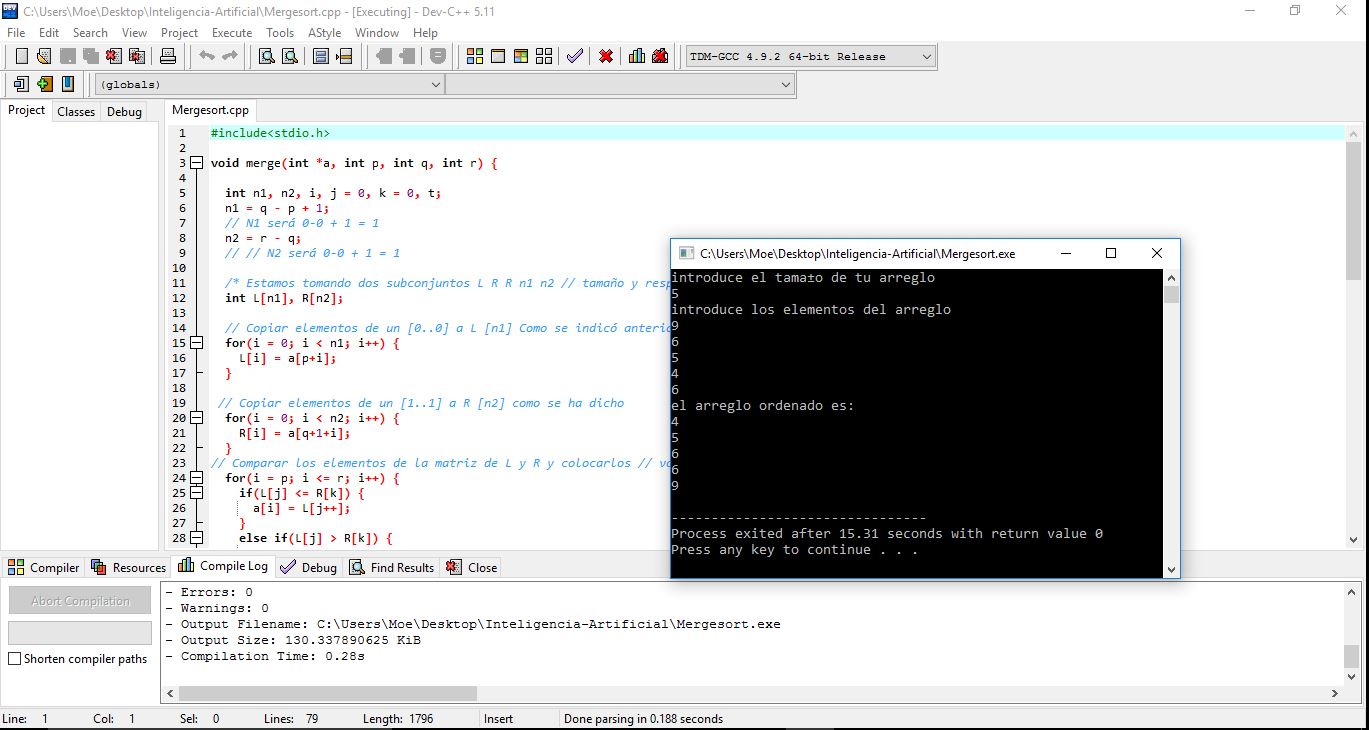
Int n1, n2, i, j, k, t, q = 14 bytes

***En Python***

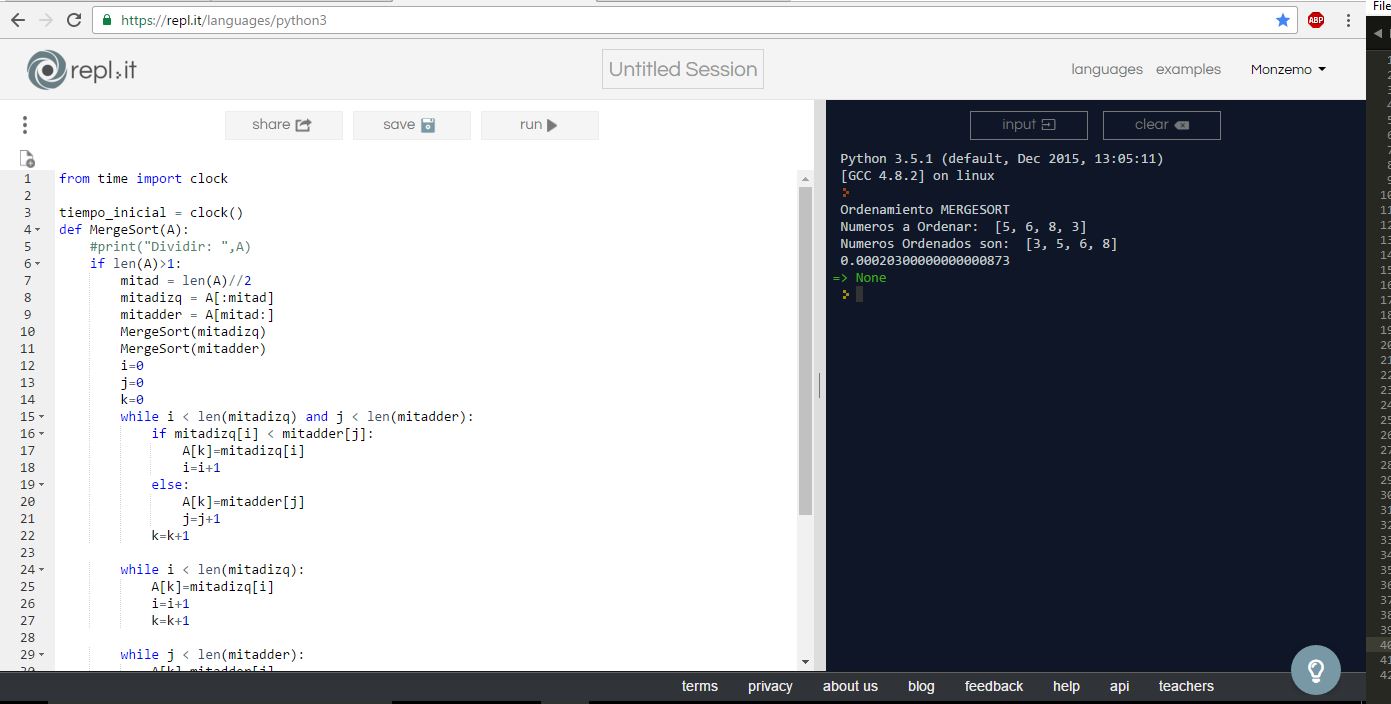
Memoria dinámica: depende de cada ejecución del algoritmo.

**Ejecución del algoritmo**

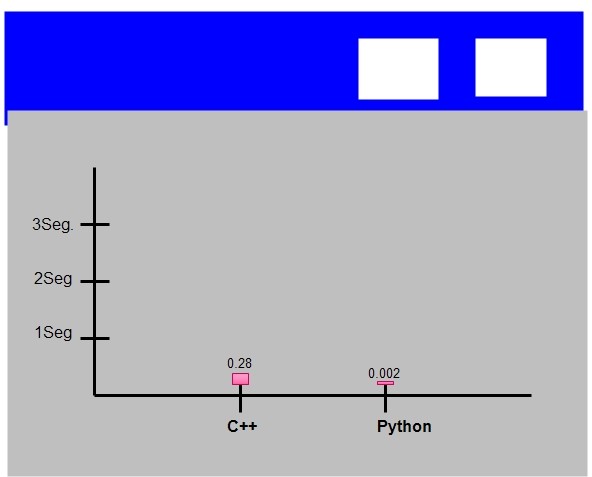
***En C++:***

******

***En Python:***



**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 dividendo 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 lasa 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++**

***Untitled Diagram (31)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.

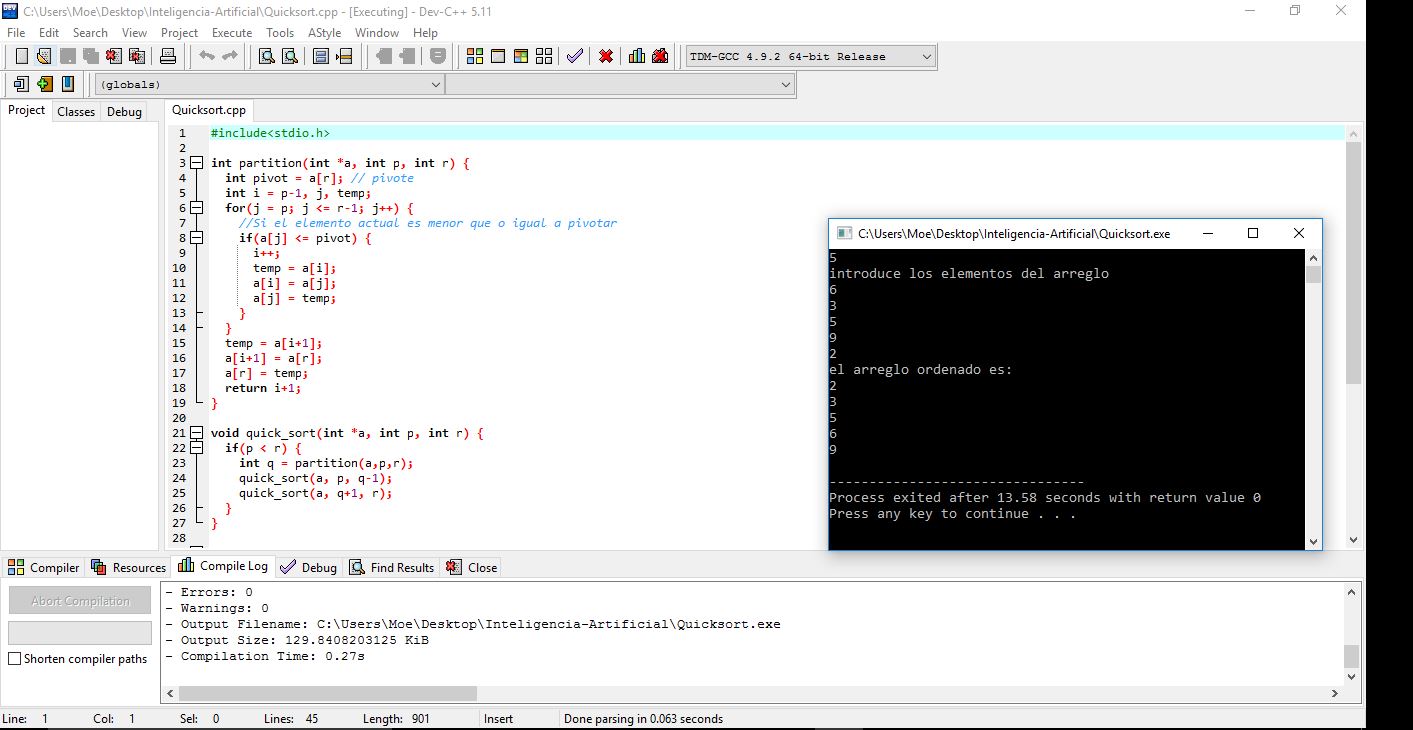
Int i, p, j, temp, pivot, n, \*a, r = 16 bytes

***En Python***

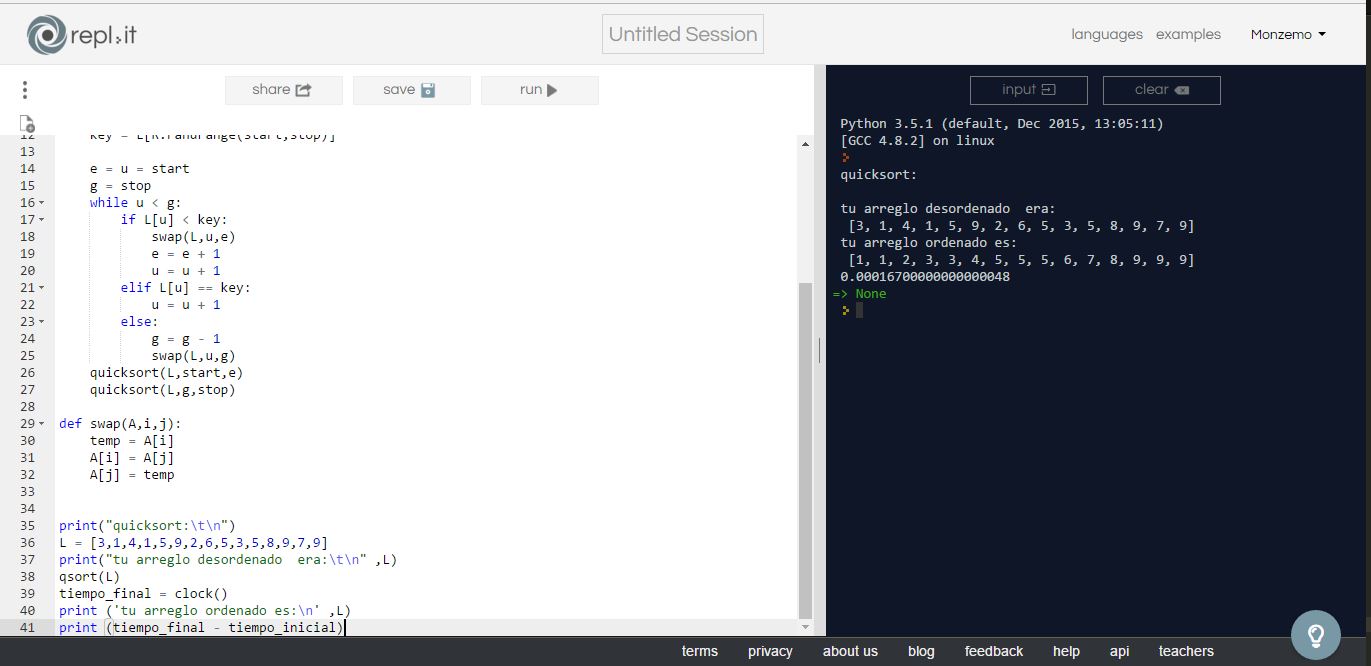
Memoria dinámica: depende de cada ejecución del algoritmo.

**Ejecución del algoritmo**

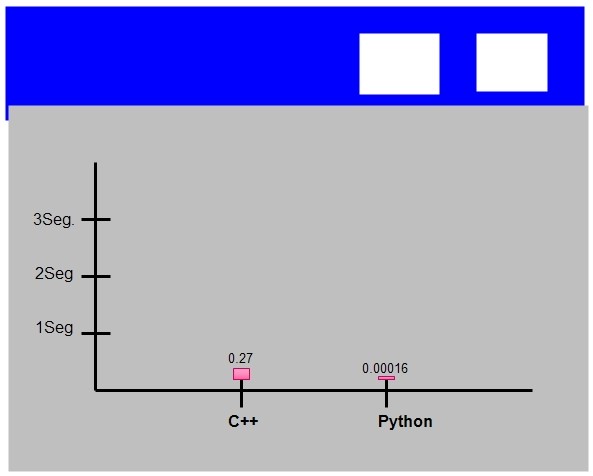
***En C++:***



***En Python:***



**Grafica de tiempos**

****

**Conclusión**

Sabemos que es unos 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 pregunta 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.(arboles)

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 lleno, 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++**

C:\Users\IGOR\AppData\Local\Microsoft\Windows\INetCache\Content.Word\Untitled Diagram (31) (1).png***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.

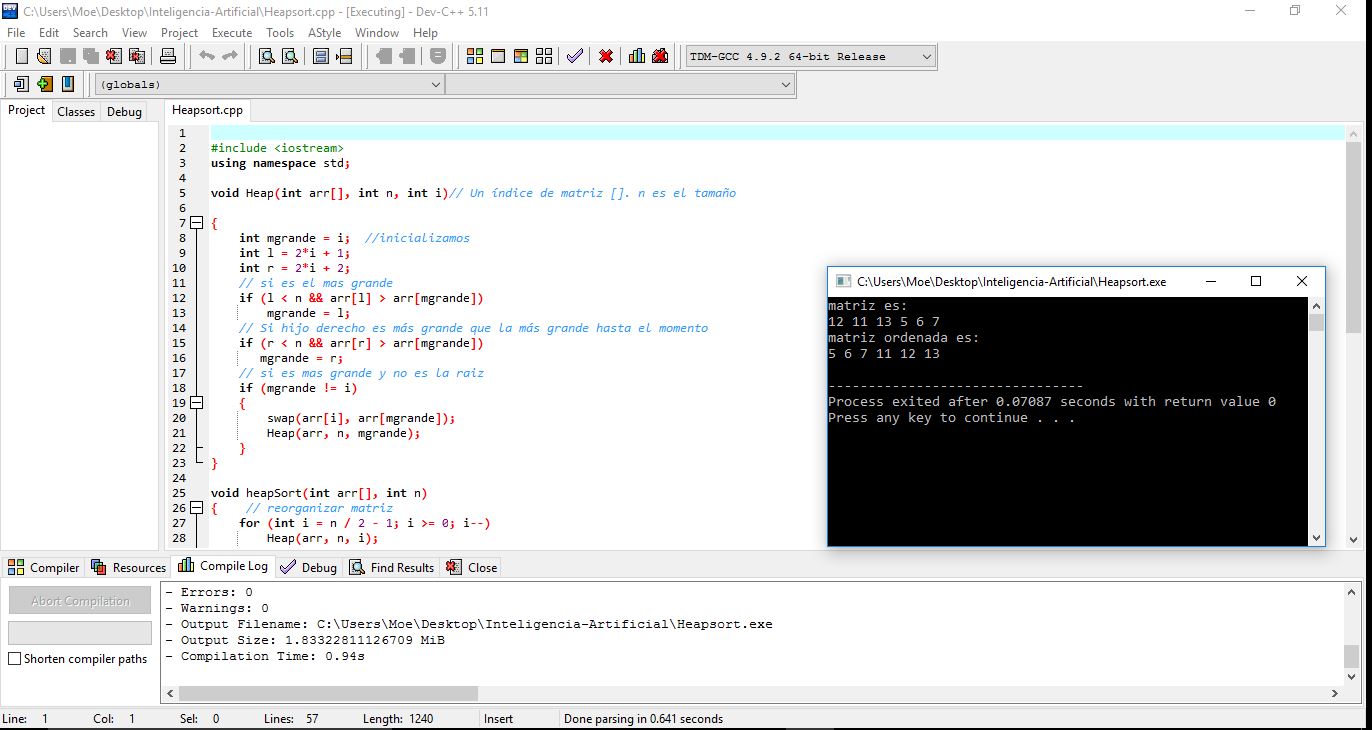
Int arr, n, i, mgrande, l, r = 12 bytes

***En Python***

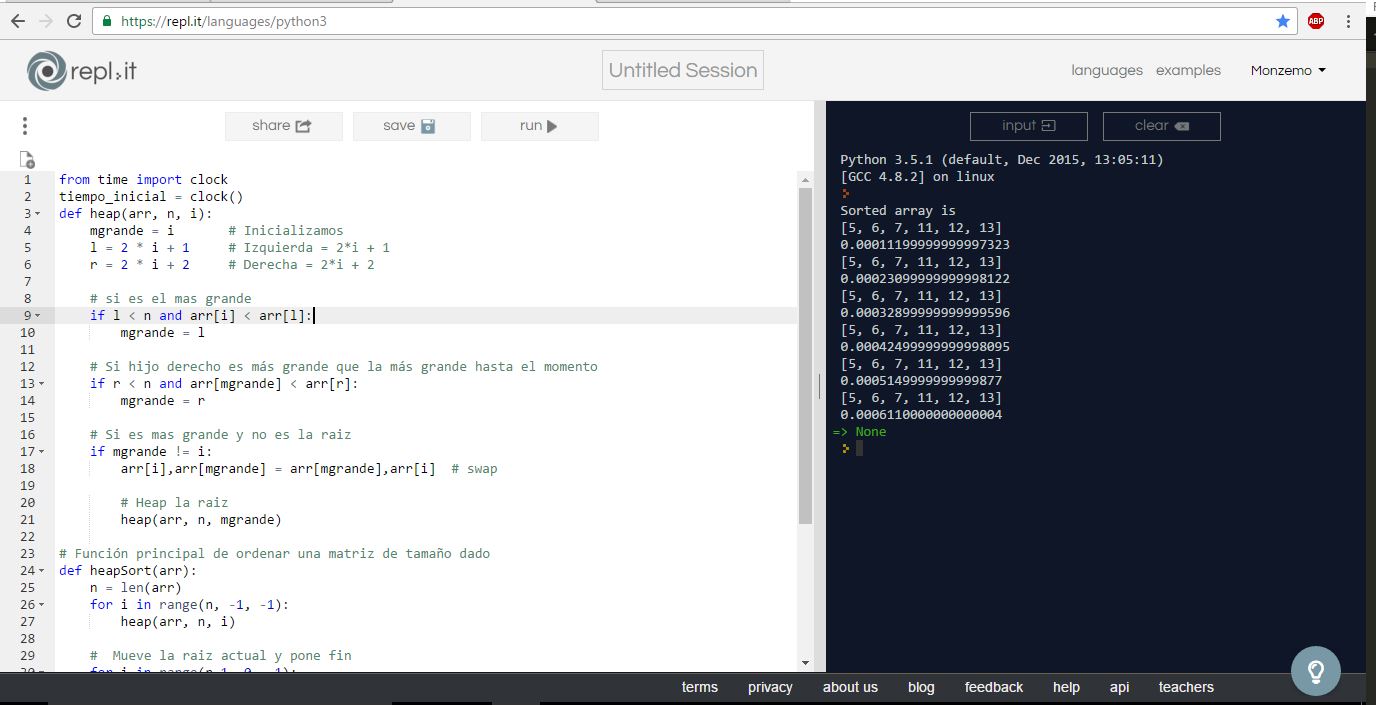
Memoria dinámica: depende de cada ejecución del algoritmo.

**Ejecución del algoritmo**

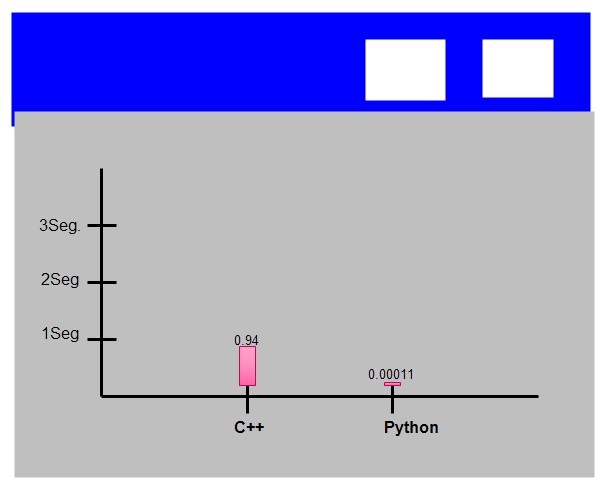
***En C++:***



***En Python:***



**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.