

# Universidad de General Sarmiento



## Introducción a la Programación

Comisión N° 9

Profesores:  
Miguel Rodríguez  
Nancy Mores

Alumnos: Gomez Silvia –  
Gomez Luis – Borda Luz

Trabajo Práctico

Año 2025

# Índice

Índice.....	Pag.1
Introducción .....	Pag. 2
Desarrollo .....	Pag. 3
Método de Selección (Selection sort) .....	Pag. 3
Ejemplo básico de Insertion sort) .....	Pag. 4
Código del TP .....	Pag. 4
Declaración de variables globales .....	Pag. 4
Función <i>init(vals)</i> .....	Pag. 4
La función <i>step()</i> .....	Pag. 5
La función <i>init(vals)</i> .....	Pag. 5
Método Burbuja (Bubble sort) .....	Pag. 7
Ejemplo básico de Bubble sort .....	Pag. 7
Método de Selección (Selection Sort) .....	Pag. 8
Análisis del código específico .....	Pag. 9
Método Quick (Quick Sort) .....	Pag. 10
Ejemplo básico de Quick Sort .....	Pag. 11
Eficacia VS Eficiencia .....	Pag. 13
Eficacia .....	Pag. 13
Eficiencia .....	Pag. 14
Conclusión .....	Pag. 15

## Introducción

En el presente trabajo vamos a desarrollar distintos algoritmos de ordenamiento para el funcionamiento de una aplicación gráfica. Los algoritmos de ordenamiento son procesos que ordenan a los elementos de una serie agrupados, en una lista o arreglo (array), en forma creciente o decreciente

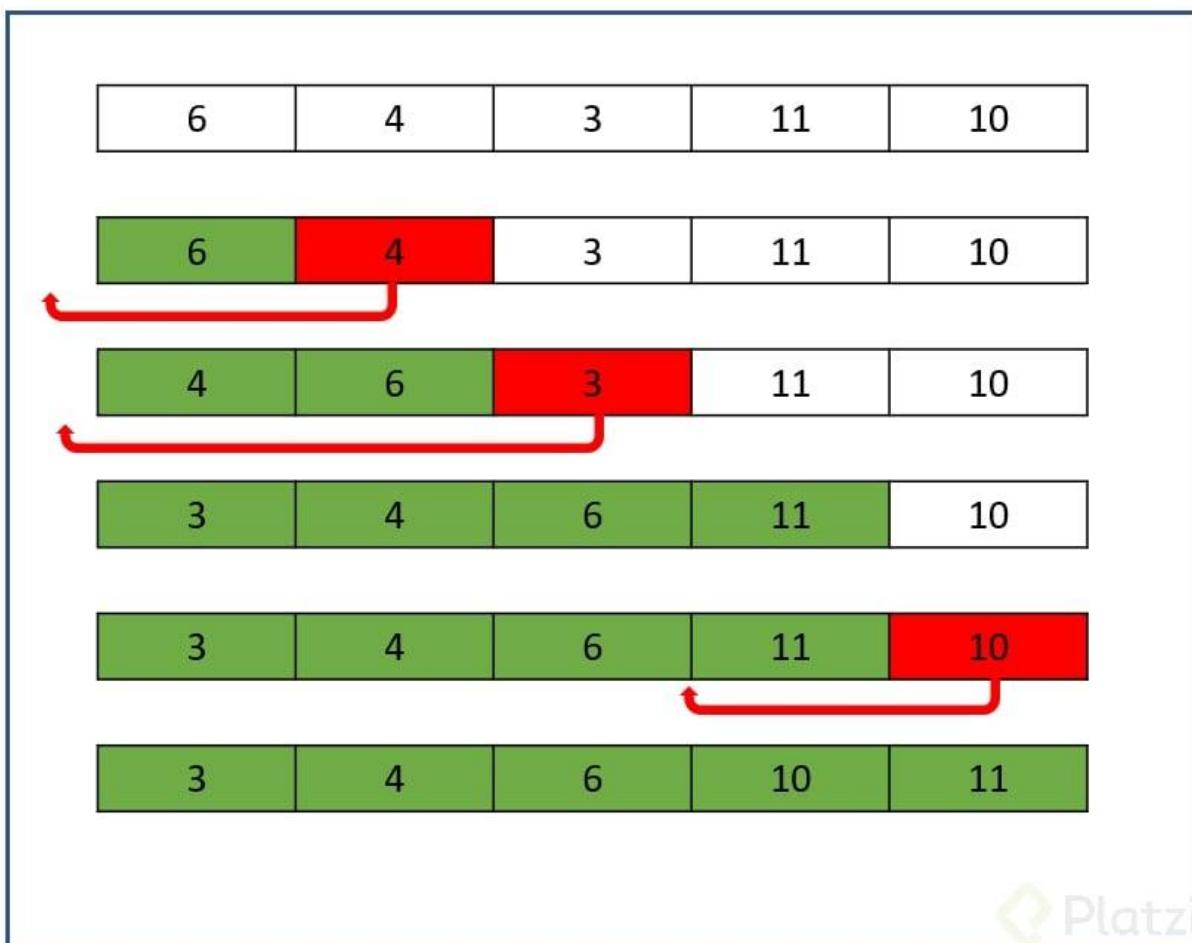
Los métodos de ordenamiento son varios como: Burbuja (Bubble Sort), Selección (Selection Sort), Inserción (Selection Sort), Ordenamiento Rápido (Quick Sort), por Mezcla (Merge Sort), etc. Cada uno de ellos posee ventajas y desventajas, como ser el tiempo de procesamiento, cantidad de elementos, efectividad, etc. Por lo tanto, la elección del método o algoritmo se realizará teniendo en cuenta las variables nombradas y las características específicas del problema a resolver.

# Desarrollo

## Método de Inserción (Insertion Sort):

En general cuando se busca información sobre este método, todas las webs consultadas remiten al ejemplo del ordenamiento manual de naipes que se utiliza al recibir las cartas en un juego.

Se toma el primer elemento como el de menor “valor” (numérico, por ejemplo) y se lo compara con el siguiente, por medio de una iteración sobre la lista, si el elemento inicial es el menor se continúa con el siguiente, si el siguiente es menor (3er elemento), se inserta en la primera posición previo al desplazamiento hacia la izquierda de él o los elementos mayores, siendo el nuevo menor. Como se observa en la figura:



Este método es muy eficiente para el ordenamiento de una lista o arreglo (Array), de poca cantidad de elementos, porque el procesamiento, pese a ser muy efectivo llevaría más tiempo.



## Ejemplo básico de Insertion sort:

```

def ordenamientoPorInsercion(unaLista):
    for i in range(1, len(unaLista)):

        valorActual = unaLista[i]
        posicion = i

        while posicion>0 and unaLista[posicion-1]>valorActual:
            unaLista[posicion]=unaLista[posicion-1]
            posicion = posicion-1

        unaLista[posicion]=valorActual

unaLista = [54, 26, 93, 17, 77, 31, 44, 55, 20]
ordenamientoPorInsercion(unaLista)
print(unaLista)

```

No nos vamos a explayar en la explicación de este código básico, sino que vamos a desarrollar el algoritmo utilizado en el trabajo.

## Código del TP

### Declaración de variables globales

```

3     items = []
4     n = 0
5     i = 0      # elemento que queremos insertar
6     j = None   # cursor de desplazamiento hacia la izquierda (None = empezar)
7

```

### Función *init(vals)*

```

8     def init(vals):
9         global items, n, i, j
10        items = list(vals)
11        n = len(items)
12        i = 1      # común: arrancar en el segundo elemento
13        j = None

```

Se llama por única vez al principio y prepara todo para iniciar el ordenamiento.

```
10     items = list(vals)
```

Se crea una lista a partir de los valores “vals”. Esta es la que se recorre para ordenar



## La función *step()*

```

15 def step():
16     # TODO:
17     # - Si i >= n: devolver {"done": True}.
18     # - Si j es None: empezar desplazamiento para el items[i] (p.ej., j = i) y devolver un highlight sin swap.
19     # - Mientras j > 0 y items[j-1] > items[j]: hacer UN swap adyacente (j-1, j) y devolverlo con swap=True.
20     # - Si ya no hay que desplazar: avanzar i y setear j=None.
21     return {"done": True}
22
23
24 ## ajustes desde aqui
25
26 items = []
27 n = 0
28 i = 0      # elemento que queremos insertar
29 j = None   # cursor de desplazamiento hacia la izquierda (None = empezar)
30

```

Esta función ejecuta un solo paso del algoritmo.

- El algoritmo va tomando cada elemento desde *i = 1* en adelante.
- Para cada elemento, lo compara con los anteriores (*j* retrocede).
- Si encuentra que está “fuera de lugar”, hace swaps adyacentes hasta insertarlo en la posición correcta.
- Devuelve en cada paso un diccionario con:
  - *a, b*: índices comparados o intercambiados.
  - *swap*: si hubo intercambio.
  - *done*: si el algoritmo terminó.

## La función *init(vals)*

```

31  def init(vals):
32      global items, n, i, j
33
34      items = list(vals)
35      n = len(items)
36      i = 1
37      j = None
38
39  def step():
40      global items, n, i, j
41
42      if i >= n:
43          return {"done": True}
44
45
46      if j is None:
47
48          j = i
49          return {"a": j - 1, "b": j, "swap": False, "done": False}
50
51
52      if j > 0 and items[j - 1] > items[j]:
53
54
55          items[j - 1], items[j] = items[j], items[j - 1]
56
57          puntero_a = j - 1
58          puntero_b = j
59
60          j -= 1
61
62
63          return {"a": puntero_a, "b": puntero_b, "swap": True, "done": False}
64
65
66      i += 1
67      j = None
68
69
70      return {"a": i - 1, "b": i, "swap": False, "done": False}

```

- *i*: recorre los elementos de la lista, uno por uno ( en este caso no es necesario realizar un for o while).
- *j*: se desplaza hacia la izquierda para insertar el elemento actual en su posición correcta.

## Método de Burbuja (Bubble Sort):

Este método recibe este nombre porque su comportamiento guarda similitud con el comportamiento de las burbujas en una bebida, ya que los elementos de menor peso son los que suben primero, (elementos más bajos), y finalmente los elementos de mayor peso.

Este método se caracteriza por ir acomodando el elemento mayor en la última posición al ir comparando desde la posición 0 (cero), número tras número hasta encontrar el mayor, luego sigue por el siguiente buscando el mayor de lo que queda y lo acomoda en la posición siguiente al ya ubicado y continúa así sucesivamente.

### Ejemplo básico de Bubble sort:

```

1 def bubble_sort(arr):
2     n = len(arr)
3     # Recorremos todos los elementos
4     for i in range(n):
5         # Últimos i elementos ya están en su lugar
6         for j in range(0, n - i - 1):
7             # Si el elemento actual es mayor que el siguiente, los intercambiamos
8             if arr[j] > arr[j + 1]:
9                 arr[j], arr[j + 1] = arr[j + 1], arr[j]
10
11     return arr
12
13 lista = [5, 3, 8, 4, 2]
14 print("Original:", lista)
15
16 ordenada = bubble_sort(lista)
17 print("Ordenada:", ordenada)

```

Este es un ejemplo básico de bubble sort, donde desde un programa principal se pasa por parámetro una lista (*arr*) a la función “*bubble\_sort(arr)*” y esta realiza el ordenamiento retornando la lista ordenada que, en este caso, se imprime en el programa principal.

Ahora vamos a proceder a explicar el algoritmo implementado:

Las primeras funciones y declaración de variables globales es similar al método Insertion Sort, por lo tanto, explicaremos desde la función *step()*.

```

15 def step():
16     # TODO:
17     # 1) Elegir índices a y b a comparar en este micro-paso (según tu Bubble).
18     # 2) Si corresponde, hacer el intercambio real en items[a], items[b] y marcar s
19     # 3) Avanzar punteros (preparar el próximo paso).
20     # 4) Devolver {"a": a, "b": b, "swap": swap, "done": False}.
21     #
22     # Cuando no queden pasos, devolvé {"done": True}.
23     for i in range(n):
24         for j in range(0,n-i-1):
25             a=j
26             b=j+1
27             swap=False
28             if items[a]>items[b]:
29                 items[a],items[b]=items[b],items[a]
30                 swap=True
31             return {"a": a, "b": b, "swap": swap, "done": False}
32
33     return {"done": True}
34

```

En principio al llamar a la función *step()*, se realiza un doble recorrido *i, j* y se comparan los valores adyacentes (*a* y *b* o *j, j+1*) si el primero es mayor se realiza un *swap* y se intercambian los valores, se verifica que no haya más elementos para “sweapear” y se finaliza el recorrido con la lista ya ordenada y se retorna

### Método de Selección (Selection sort):

```

def selection_sort(arr):
    n = len(arr)
    # Recorremos todos los elementos
    for i in range(n):
        # Suponemos que el mínimo está en la posición i
        min_idx = i
        # Buscamos el mínimo en el resto de la lista
        for j in range(i + 1, n):
            if arr[j] < arr[min_idx]:
                min_idx = j
        # Intercambiamos el mínimo encontrado con el primer elemento del
        # subarray
        arr[i], arr[min_idx] = arr[min_idx], arr[i]
    return arr

lista = [64, 25, 12, 22, 11]
print("Original:", lista)

ordenada = selection_sort(lista)
print("Ordenada:", ordenada)

```

Este es un método de ordenamiento que funciona de una manera simple con dos listas; la primera una sub-lista ordenada a la izquierda y una sub-lista no ordenada a la derecha.



El proceso se repite en cada pasada; primero se busca el elemento más pequeño en la parte no ordenada y se intercambia “swap” con el primer elemento de la parte no ordenada.

Las variables de estado son las siguientes:

- i= Es la cabeza de la parte no ordenada. Marca la posición donde debe colocarse el próximo elemento más pequeño.
- J=Es el cursor que recorre la sub-lista no ordenada para encontrar el mínimo.
- min\_idx= El índice que guarda la posición del valor mínimo encontrado hasta el momento en la pasada actual.
- fase= La variable crucial que alterna entre los dos estados principales: "buscar" y "swap".

### Análisis del código específico:

```

35  def init(vals):
36
37      global items, n, i, j, min_idx, fase
38
39      items = list(vals)
40      n = len(items)
41
42      i = 0
43      j = 1
44      min_idx = 0
45      fase = "buscar"
46
47  def step():
48
49      global items, n, i, j, min_idx, fase
50
51
52      if i >= n - 1:
53          return {"done": True}
54
55      if fase == "buscar":
56
57
58          if j < n:
59              puntero_a = j
60              puntero_b = min_idx
61
62              if items[j] < items[min_idx]:
63                  min_idx = j
64
65              j += 1
66
67          return {"a": puntero_a, "b": puntero_b, "swap": False, "done": False}
68

```

```

68
69     else:
70         fase = "swap"
71         return step()
72
73     elif fase == "swap":
74
75
76         puntero_a = i
77         puntero_b = min_idx
78         swap_hecho = False
79
80         if i != min_idx:
81             items[i], items[min_idx] = items[min_idx], items[i]
82             swap_hecho = True
83
84         # Preparar para la siguiente pasada
85         i += 1
86         j = i + 1
87         min_idx = i
88         fase = "buscar"
89
90     return {"a": puntero_a, "b": puntero_b, "swap": swap_hecho, "done": False}

```

Mientras  $j$  recorre la lista desde  $i + 1$  hasta el final:

- Compara el elemento actual ( $items[j]$ ) con el mínimo encontrado ( $items[min\_idx]$ ).
- Si encuentra un valor más pequeño, actualiza  $min\_idx$ .
- Devuelve lo más importante de la comparación ( $a, b$ ) sin swap todavía.
- Cuando  $j$  llega al final, cambia la fase a “ $swap$ ” para colocar el mínimo en su lugar.
- La fase “ $swap$ ” ubica al mínimo en la posición adecuada
- Al finalizar, si no hay más elementos para ordenar, se retorna

### Método Qwick (Qwick Sort):

Es un algoritmo de ordenamiento eficiente que funciona seleccionando un elemento como "pivot" y dividiendo el resto de la lista en dos sub-listas: una con elementos menores al pivot y otra con elementos mayores. La comparación se realiza entre los elementos de las sub-listas y el pivot

## Ejemplo básico de Quick Sort:

```

1 def quick_sort(arr):
2     if len(arr) ≤ 1:
3         return arr
4     else:
5         # Elegimos el pivote (aquí el último elemento)
6         pivot = arr[-1]
7         # Particionamos en menores y mayores al pivote
8         left = [x for x in arr[:-1] if x ≤ pivot]
9         right = [x for x in arr[:-1] if x > pivot]
10        # Ordenamos recursivamente y unimos
11        return quick_sort(left) + [pivot] + quick_sort(right)
12
13 lista = [10, 7, 8, 9, 1, 5]
14 print("Original:", lista)
15
16 ordenada = quick_sort(lista)
17 print("Ordenada:", ordenada)

```

En la imagen se ve la acción del pivote, que contiene el elemento a comparar y ubicar en la posición correcta.

Ahora vamos a analizar el código específico:

```

1 com09-Gomez-Borda-Gomez\visualizador\algorithms\sort_quick.py
2 stack = []      # pila de subrangos pendientes
3 i = j = 0
4 pivote = None
5 fase = "buscar"
6
7 def init(vals):
8     global items, stack, i, j, pivote, fase
9     items = list(vals)
10    stack = [(0, len(items)-1)]  # rango inicial completo
11    i = j = 0
12    pivote = None
13    fase = "buscar"
14
15 def step():
16     global items, stack, i, j, pivote, fase
17
18     # Caso base: si no quedan subrangos
19     if not stack:
20         return {"done": True}
21
22     # Tomar el subrango actual
23     left, right = stack[-1]
24
25     if left >= right:
26         stack.pop()
27         return {"done": False}
28
29     # Fase buscar: elegir pivote y preparar índices
30     if fase == "buscar":
31         pivote = items[right]  # pivote = último elemento
32         i = left - 1
33         j = left
34         fase = "particionar"

```

```

35     # Fase particionar: recorrer y hacer swaps
36     if fase == "particionar":
37         if j < right:
38             if items[j] <= pivot:
39                 i += 1
40                 items[i], items[j] = items[j], items[i]
41                 j += 1
42                 return {"a": i, "b": j-1, "swap": True, "done": False}
43             else:
44                 j += 1
45                 return {"a": j-1, "b": j-1, "swap": False, "done": False}
46         else:
47             # colocar pivot en su lugar
48             items[i+1], items[right] = items[right], items[i+1]
49             pos = i+1
50             stack.pop()
51             # agregar subrangos izquierdo y derecho
52             stack.append((left, pos-1))
53             stack.append((pos+1, right))
54             fase = "buscar"
55             return {"a": pos, "b": right, "swap": True, "done": False}
56

```

Este método ordena la lista realizando particiones iterativas a partir de un pivote, avanzando con cada llamada a `step()`. Cada paso devuelve un diccionario con información del swap y del estado del proceso.

- *items*: la lista que se está ordenando. Se modifica en tiempo real.
- *stack*: pila de subrangos pendientes por ordenar, de la forma (*left*, *rigth*).
- *left*, *right*: límites del subrango activo en la cima de la pila.
- *pivot*: valor elegido para particionar; aquí es *items[rigth]*.
- *i*, *j*: índices internos para la partición tipo colocando el pivote al final.
- fase: controla en qué etapa estamos: “*buscar*” o “*particionar*”.

Fase buscar: seleccionar pivote y preparar recorrido

- Elegir pivote: el último elemento del subrango actual *pivot = items[rigth]*.
- Inicializar índices: *i = left - 1* y *j = left*.
- Transición: cambia a “*particionar*” para comenzar a recorrer.

Fase particionar: recorrer y colocar menores a la izquierda

- Avance con *j*: mientras *j < rigth*, se compara *items[j]* con el *pivot*.
- Si es menor o igual: incrementar *i* y hacer swap *items[i] ↔ items[j]*.
- Retorno del paso: `{"a": j - 1, "b": j - 1, "swap": False, "done": False}` después de aplicar el intercambio.
- Si es mayor: no se intercambia, solo avanza *j*.
- Retorno del paso: `{"a": j-1, "b": j-1, "swap": False, "done": False}` para señalar avance sin swap.
- Cuando *j == right*: se cierra la partición colocando el pivote en su posición final:
- Swap final: *items[i+1]*, *items[right]* y *pos = i + 1*.

- Subrangos nuevos: se quita el actual de la pila y se agregan:
- Izquierdo: (*left, pos - 1*)
- Derecho: (*pos + 1 , rigth*)
- Retorno del paso: {"*a*": *pos*, "*b*": *right*, "swap": *True*, "done": *False*} y vuelve a "buscar" para el siguiente subrango.

Realizamos también cambios en el index.html, cambios mínimos de color de algunos botones o fondos. Cambiamos el estilo en la etiqueta del título.

```
title>Visualizador de Ordenamientos – Python externo</title>
```

```
<style>|
```

En <style> modificamos el color del panel a un celeste más claro, y el borde del mismo en verde.

En la parte de acciones a realizar cambiamos el color de los botones. Le agregamos style, cambiando los botones de mezclar, reproducir, pausa y reset a color rojo. Y el de recargar python a azul.

```
<div class="card">

    <label>Acciones</label>

    <div class="btns">

        <button style="background-color: #2e3436; color: white;" id="shuffle">Mezclar</button>

        <button style="background-color: #2e3436; color: white;" id="play" class="primary">Reproducir</button>

        <button style="background-color: #2e3436; color: white;" id="step">Paso</button>

        <button style="background-color: #2e3436; color: white;" id="pause">Pausa</button>

        <button style="background-color: #2e3436; color: white;" id="reset">Reset</button>

        <button style="background-color: #2e3436; color: white;" id="reloadPy">Recargar Python</button>

    </div>

    <div style="margin-top:8px">Ejecución: <span class="pill" id="modeExec">Python</span></div>

</div>
```

## Eficacia vs Eficiencia

Antes del análisis final y a modo de conclusión nos interesa cerrar con 2 conceptos como lo son eficacia y eficiencia. Podemos llegar a cuestionar: ¡qué se busca eficacia o eficiencia?

### **Eficacia:**

Se define como la capacidad de lograr un objetivo, es decir que, si lo llevamos a nuestro trabajo, todos los algoritmos desarrollados son eficaces.

### **Eficiencia:**

A la eficiencia se la define la capacidad de lograr el objetivo, pero teniendo en cuenta los recursos utilizados.

Es decir que si vamos a nuestro trabajo, algunos algoritmos van a eficientes y eficaces.

# Conclusión

En base al análisis final del trabajo notamos grandes diferencias en lo que respecta a eficacia y eficiencia, si bien todos los algoritmos, como ya lo habíamos anticipado, son eficaces. Pero, ¿Todos son eficientes?

Se van a fijar estas variables: Cantidad: 40 Tiempo: 20 ms

Bubble sort: 8,19 seg.

Insertion Sort: 10,20 seg.

Selection Sort: 17,36 seg.

Quick Sort: 8,09 seg

Según se puede observar del análisis de los tiempos de procesos, para la aplicación el algoritmo más eficiente es Qwick Sort, seguido de Bubble Sort.

De esto podemos concluir que todos los algoritmos van a ser eficaces y eficientes según los casos en los que los utilicemos. En este caso vemos que el quick sort realiza el ordenamiento en menos tiempo.

Los 3 algoritmos base están implementados y finalizan correctamente. En todos los casos las funciones init() resetea el estado y step() realiza el micro-paso. Presentan otros swaps antes de retornar a swap= True.

Probamos qué pasaría si la lista estuviera vacía. Podemos cambiar el valor mínimo de la lista modificando en el documento html. También desde localhost:8000 presionando F12 podemos inspeccionar el html y cambiar los valores del rango, ya que el mínimo es 5. Colocamos el mínimo como 0, para que la lista esté vacía. El programa no tiene errores, pero las barras no están y por ende el programa no hace nada.

```
draw() {  
  
    var wait = Math.max(5, parseInt($('speed').value, 10) -  
        (performance.now() - t0)); // ⚡ mínimo 5ms  
  
    setTimeout(loop, wait);  
}
```

En el caso de las listas cortas, las barras disminuyen y el tiempo de ejecución es menor. Si la lista ingresada ya estuviera ordenada el programa no haría nada.

También nos gustaría mencionar la dificultad que sufrimos con el Merge sort por su complejidad ya que se necesita un array auxiliar para realizar la mezcla. Los cuales

necesitan tres punteros: “i, j, k” pero la copia de datos con el array principal y el temporal dentro del step() demostró ser frágil para que se vea en el visualizador.

La falla de lógica para gestionar los límites se vio influenciada por igual. Es decir, arrancaba y paraba sin ordenar, o directamente, se congelaba.