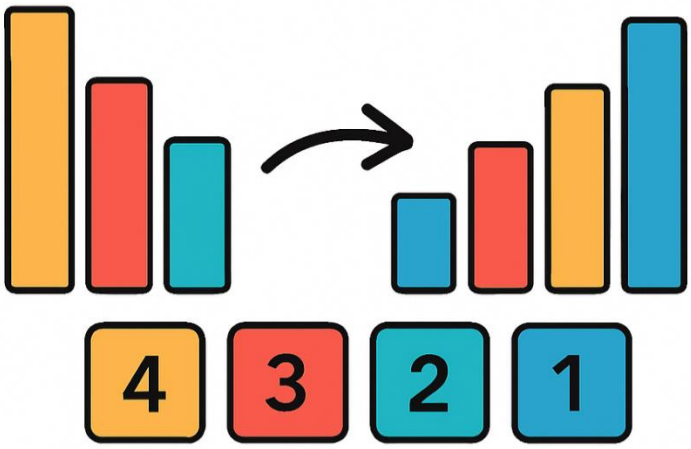
INFORME

Visualización de Algoritmos de Ordenamiento



Universidad: Universidad Nacional General Sarmiento

Materia: Introducción a la Programación (2C 2025)

Integrantes:

- Valentín Bodnar
- Juan Cruz Peralta
- Giuliana Varela

Fecha de entrega: 28 de noviembre de 2025

Indice

1. Introducción general

Se nos proporcionó una plantilla que incluye un archivo index.html con código HTML, CSS y JavaScript. El JavaScript se encarga de vincular los botones del código HTML con los algoritmos que implementamos en Python.

Cuando el usuario presiona el botón “Reproducir”, el JavaScript ejecuta una única vez la función init(vals) para inicializar los datos. Luego, comienza a llamar repetidamente al método step() hasta que retorne done= true, indicando que la lista ya está ordenada.

Es importante aclarar que las funciones init(vals) y step() que se ejecutan son las correspondientes al archivo Python asociado al algoritmo seleccionado en el menú desplegable.

En los archivos Python se implementaron distintos algoritmos de ordenamiento con el objetivo de analizar sus diferencias en funcionamiento y comportamiento paso a paso.

- Bubble (burbuja)
- Selection (selección)
- Insertion (inserción)
- Quick (rápido)
- Merge (mezcla)

Aunque todos los algoritmos de ordenamiento tienen el mismo objetivo: ordenar una lista, cada uno lo hace de manera distinta.

2. Código de los algoritmos

2.1 Bubble

Este algoritmo compara los elementos de la lista de a pares consecutivos y los intercambia si están en el orden incorrecto. En cada comparación, el valor mas grande se mueve hacia la derecha, quedando al final de la lista.
En cada vuelta, los últimos elementos ya comparados desplazados a al final de la lista no vuelven a evaluarse, porque se identifican como los mayores del ciclo anterior.

```python
items = []
n = 0
i = 0
j = 0

def init(vals):
    global items, n, i, j
    items = list(vals)
    n = len(items)
    i = 0
    j = 0

def step():
    global items, n, i, j

    if i >= n - 1:
        return {"done": True}

    a = j
    b = j + 1
    swap = False

    if items[a] > items[b]:
        items[a], items[b] = items[b], items[a]
        swap = True

    j += 1

    if j >= n - i - 1:
        j = 0
        i += 1

    return {"a": a, "b": b, "swap": swap, "done": False}
```

Decisiones y dificultades:

Definir el comportamiento del índice i no presento complicaciones, ya que por definición en el algoritmo de burbuja su valor se incrementa al finalizar cada recorrido y representa la cantidad de elementos que ya quedaron ordenados al final de la lista.

Si los pares comparados resultan ser el primero mayor al segundo se intercambian.

La principal dificultad fue manejar correctamente el índice j, que cuenta la cantidad de comparaciones dentro de cada ciclo. Como los elementos al final de la lista ya se identifican como los mayores en los ciclos anteriores, se resolvió que cuando j >= n – i – 1 se dé por finalizado el ciclo actual y se comience uno nuevo aumentando i en uno (i += 1).

2.2 Selection

Este algoritmo va por cada posición de la lista reemplazando por el valor más chico de las posiciones restantes, quedando así dos una parte ordenada y otra desordenada.
Este proceso lo hace hasta terminar de recorrer la parte desordenada.

```python
items = []
n = 0
i = 0          # inicio de la parte no ordenada
j = 0          # cursor que recorre buscando el mínimo
min_idx = 0    # índice del mínimo en la pasada actual

def init(vals):
    global items, n, i, j, min_idx
    items = list(vals)
    n = len(items)
    i = 0
    j = i + 1
    min_idx = i

def step():
    global items, n, i, j, min_idx

    if i >= n - 1:
        return {"done": True}

    a = min_idx
    b = j
    swap = False

    if j < n:
        if items[j] < items[min_idx]:
            min_idx = j
        j += 1
        return {"a": a, "b": b, "swap": swap, "done": False}
    else:
        if min_idx != i:
            items[i], items[min_idx] = items[min_idx], items[i]
            swap = True
        # Pasada completa: avanzamos al inicio de la parte no ordenada
        a, b = i, min_idx
        i += 1
        j = i + 1
        min_idx = i
        return {"a": a, "b": b, "swap": swap, "done": False}
```

Decisiones y dificultades:

Se comparan de a pares consecutivos buscando el mínimo sin modificar el índice i.

Cuando termina de recorrer y hallo el mínimo, verifica si el mínimo ya no se encuentra en esa misma posición lo intercambia.

La parte más compleja fue manejar correctamente el reinicio de los índices al finalizar cada pasada, es decir, cuando se halló el mínimo.

En particular, fue necesario recordar que, al terminar de buscar el mínimo, el índice j debe volver a iniciarse desde la posición siguiente a la parte ya ordenada j = i + 1 para comenzar la nueva búsqueda correctamente.

2.3 Insertion

Este algoritmo recorre la lista de izquierda a derecha, y en cada paso toma el elemento actual y lo inserta en la posición correcta dentro de la parte ya ordenada (los elementos a su izquierda), desplazando hacia la derecha los elementos que sean mayores. Si el elemento se encuentra en la posición correcta no hace ningún intercambio.

```
items = []
n = 0
i = 0      # elemento que queremos insertar
j = None   # cursor de desplazamiento hacia la izquierda (None = empezar)

def init(vals):
    global items, n, i, j
    items = list(vals)
    n = len(items)
    i = 1      # arrancar en el segundo elemento
    j = None

def step():
    global items, n, i, j

    if i >= n:
        return {"done": True}

    if j is None:
        j = i
        return {"a": j-1, "b": j, "swap": False, "done": False}

    if j > 0 and items[j-1] > items[j]:
        items[j-1], items[j] = items[j], items[j-1]
        j -= 1
        return {"a": j, "b": j+1, "swap": True, "done": False}

    i += 1
    j = None
    return {"a": i-1, "b": i, "swap": False, "done": False}
```

Decisiones y dificultades:

Se inicia desde del segundo elemento porque el primero ya se considera ordenado.

Una dificultad fue manejar correctamente los índices, especialmente el uso de j para desplazar el elemento actual hacia la izquierda solo mientras haya valores mayores antes de él (j-1). Se resolvió controlando los límites con la condición j > 0 y reiniciando j a None al finalizar cada inserción.

2.4 Quick

Este algoritmo elige un elemento llamado pivote y acomoda la lista de forma que todos los valores menores queden a su izquierda y los mayores a su derecha.
Luego se repite de manera recursiva el mismo proceso por separado en casa parte hasta que toda la lista queda ordenada.

```
items = []
n = 0
stack = []       # aca guardo los rangos que me falta ordenar

# estas variables las uso mientras particiono
inicio = 0       # desde donde arranca el rango que estoy ordenando
fin = 0          # hasta donde llega el rango
pivot_val = 0    # el valor del pivot (guardo el valor, no la posicion)
i = 0            # marca hasta donde llegaron los elementos menores que el pivot
j = 0            # voy recorriendo el rango con j

fase = 0         # controla en que parte del proceso estoy


def init(vals):
    global items, n, stack, inicio, fin, pivot_val, i, j, fase

    items = list(vals)
    n = len(items)
    stack = []

    # si hay mas de 1 elemento, meto todo en la pila para empezar
    if n > 1:
        stack.append((0, n - 1))

    # reseteo todo
    inicio = 0
    fin = 0
    pivot_val = 0
    i = 0
    j = 0
    fase = 0
```

```python
def step():
    global items, n, stack, inicio, fin, pivot_val, i, j, fase

    # si el array tiene 0 o 1 elemento, ya esta ordenado
    if n <= 1:
        return {"done": True}

    # FASE 0: sacar un rango nuevo de la pila para procesar
    if fase == 0:
        # si no quedan rangos, termine
        if not stack:
            return {"done": True}

        inicio, fin = stack.pop()

        # si el rango tiene 1 solo elemento, ya esta ordenado
        if inicio >= fin:
            return {"a": inicio, "b": fin, "swap": False, "done": False}

        # guardo el valor del pivot (es el ultimo del rango)
        pivot_val = items[fin]
        i = inicio  # aca va a empezar la frontera de elementos menores
        j = inicio  # arranco a recorrer desde el inicio

        fase = 1
        # muestro cual es el pivot
        return {"a": j, "b": fin, "swap": False, "done": False}

    # FASE 1: voy recorriendo y moviendo los menores a la izquierda
    if fase == 1:
        # mientras no llegue al pivot, sigo comparando
        if j <= fin - 1:
            # si el elemento actual es menor que el pivot
            if items[j] < pivot_val:
                # lo muevo a la zona de menores (intercambio con i)
                items[i], items[j] = items[j], items[i]
                a = i
                b = j
                i += 1  # avanzo la frontera de menores
                j += 1  # avanzo el cursor
                return {"a": a, "b": b, "swap": True, "done": False}
            else:
                # es mayor o igual, lo dejo donde esta
                a = j
                b = fin  # muestro que lo estoy comparando con el pivot
                j += 1   # solo avanzo el cursor
                return {"a": a, "b": b, "swap": False, "done": False}
        else:
            # llegue al final, ahora pongo el pivot en su lugar definitivo
            items[i], items[fin] = items[fin], items[i]
            pivot_pos = i

            # ahora apilo las dos mitades para ordenarlas despues
            # primero la mitad izquierda (menores que el pivot)
            if inicio < pivot_pos - 1:
                stack.append((inicio, pivot_pos - 1))
            # despues la mitad derecha (mayores que el pivot)
            if pivot_pos + 1 < fin:
                stack.append((pivot_pos + 1, fin))

            fase = 0  # vuelvo a la fase 0 para sacar otro rango
            return {"a": i, "b": fin, "swap": True, "done": False}

    return {"done": True}
```

Decisiones y dificultades:

Quick funciona eligiendo un elemento como "pivot" (yo siempre agarro el último) y reorganizando el array para que todos los menores queden a la izquierda del pivot y los mayores a la derecha. Una vez que hago eso, el pivot ya está en su posición final y no se mueve más. Después repito el proceso con las dos mitades (izquierda y derecha) hasta que todo el array queda ordenado. Como no puedo usar recursión en el visualizador, uso una pila para ir guardando las "tareas pendientes" (los rangos que me falta ordenar) y los voy procesando de a uno.

2.5 Merge

Este algoritmo divide la lista en mitades hasta que cada parte tiene un solo elemento. Luego va uniendo las partes de forma ordenada, comparando los elementos de cada mitad y colocando primero los más chicos.

```python
items = []
n = 0

# variables para controlar que bloques estoy procesando
tam_bloque = 1    # tamaño actual de los bloques que estoy mezclando
izq = 0           # inicio del primer bloque
med = 0           # fin del primer bloque (y comienzo del segundo)
der = 0           # fin del segundo bloque

# para mezclar uso una estrategia simple:
# calculo como deberia quedar el segmento ordenado (lo guardo en "objetivo")
# y voy haciendo swaps de a uno para alcanzar ese estado
fase = 0          # me dice en que etapa estoy: 0=preparar, 1=calcular, 2=mezclar
pos = 0           # posicion actual dentro del segmento que estoy ordenando
objetivo = []     # aca guardo como tiene que quedar el segmento


def init(vals):
    global items, n, tam_bloque, izq, med, der, fase, pos, objetivo

    items = list(vals)
    n = len(items)

    # si tiene 0 o 1 elemento, ya esta ordenado
    if n <= 1:
        return

    # empiezo con bloques de tamaño 1
    tam_bloque = 1
    izq = 0
    med = 0
    der = 0
    fase = 0
    pos = 0
    objetivo = []
```

```python
def step():
    global items, n, tam_bloque, izq, med, der, fase, pos, objetivo

    # array vacio o de 1 elemento ya esta listo
    if n <= 1:
        return {"done": True}

    # FASE 0: preparar el siguiente par de bloques para mezclar
    if fase == 0:
        # si el tamaño de bloque llego al tamaño del array, termine
        if tam_bloque >= n:
            return {"done": True}

        # si procese todos los bloques de este nivel, paso al siguiente
        if izq >= n:
            tam_bloque *= 2  # duplico el tamaño de bloque
            izq = 0          # arranco de nuevo desde el principio
            return step()    # llamo de vuelta para seguir

        # configuro los limites de los bloques a mezclar
        med = min(izq + tam_bloque, n)      # hasta donde llega el bloque izq
        der = min(izq + tam_bloque * 2, n)  # hasta donde llega el bloque der

        # si no hay bloque derecho, paso al siguiente par
        if med >= n or izq >= der:
            izq += tam_bloque * 2  # salto al siguiente par de bloques
            return step()

        # listo para calcular el objetivo
        fase = 1
        return {"a": izq, "b": med if med < n else n - 1, "swap": False, "done":
False}

    # FASE 1: calcular como deberia quedar el segmento ordenado
    if fase == 1:
        # simplemente ordeno el segmento completo y lo guardo
        objetivo = sorted(items[izq:der])
        pos = izq  # empiezo desde el inicio del segmento
        fase = 2   # paso a la fase de mezclar
        return {"a": pos, "b": der - 1, "swap": False, "done": False}

    # FASE 2: ir haciendo swaps hasta alcanzar el objetivo
    if fase == 2:
        # si llegue al final del segmento, paso al siguiente par de bloques
        if pos >= der:
            izq += tam_bloque * 2
            fase = 0
            return step()

        # si el elemento ya esta donde tiene que estar, avanzo
        if items[pos] == objetivo[pos - izq]:
            pos += 1
            return {"a": pos - 1, "b": pos - 1, "swap": False, "done": False}

        # busco donde esta el elemento que deberia ir en esta posicion
        idx_correcto = None
        for k in range(pos + 1, der):
            if items[k] == objetivo[pos - izq]:
                idx_correcto = k
                break

        # si no lo encuentro (raro, pero por las dudas), avanzo
        if idx_correcto is None:
            pos += 1
            return {"a": pos - 1, "b": pos - 1, "swap": False, "done": False}

        # hago un swap adyacente para acercar el elemento a su lugar
        # esto es como ir "burbujeando" el elemento hacia donde tiene que ir
        items[idx_correcto - 1], items[idx_correcto] = items[idx_correcto],
items[idx_correcto - 1]

        return {"a": idx_correcto - 1, "b": idx_correcto, "swap": True, "done": False}

    return {"done": True}
```

Decisiones y dificultades:

Merge funciona arrancando con bloques de tamaño 1 (que ya están ordenados por definición) y los voy mezclando de a pares para formar bloques más grandes: primero mezclo bloques de 1 para hacer bloques de 2, después mezclo los de 2 para hacer de 4, después de 4 para hacer de 8, y así sucesivamente hasta que el bloque es del tamaño del array completo. Para mezclar dos bloques, calculo cómo debería quedar el segmento ordenado usando sorted() y después voy haciendo swaps adyacentes paso a paso hasta alcanzar ese resultado, así se puede ver la animación en el visualizador.