sorting

May 10, 2025

1 Bubble Sorting

1.1 Why Is It Called "Bubble Sort"?

The name **"bubble sort"** comes from the way the **largest elements "bubble up" to the top (end) of the list** during each pass through the array.

1.1.1 Here's the logic:

- Imagine air bubbles in water — they naturally rise to the surface.
- Similarly, in bubble sort:
  - The largest unsorted value is compared and swapped forward until it reaches its correct position at the end of the array.
  - On the next pass, the second-largest value "bubbles up," and so on.
- Each pass ensures the largest remaining unsorted element is moved to the rightmost unsorted position, like bubbles rising to the top.

This repeated movement of the largest unsorted element toward the end on each pass gives the algorithm its name.

1.2 Time Complexity

Case	Time Complexity
Best (Sorted)	O(n)
Average	$O(n^2)$
Worst	$O(n^2)$

- **Space Complexity**: O(1) (in-place)
- **Stable**: Yes

Bubble Sort is a simple comparison-based sorting algorithm. It works by repeatedly swapping adjacent elements if they are in the wrong order. This process is repeated until the list is sorted.

1.3 How It Works (Step-by-Step):

1. Start at the beginning of the array.
2. Compare the first two elements.
3. If the first is greater than the second, swap them.

4. Move to the next pair and repeat.
5. After each pass, the largest element "bubbles up" to the end.
6. Repeat for the rest of the array, ignoring the last sorted elements each time.

1.4 Calling the `bubble_sort` Function

We will call the `bubble_sort` function and pass it a list that we want to sort.

"'python # Example list numbers = [5, 2, 9, 1, 5, 6]

2 Calling the bubble_sort function with the list

sorted_numbers = bubble_sort(numbers)

3 Output the sorted list

print(sorted_numbers) # Output: [1, 2, 5, 5, 6, 9]

Example with a List of 5 Elements

Outer loop runs 5 times (i = 0 to 4)

Inner loop runs:

4 times when i = 0 (indexes 0 to 3)

3 times when i = 1 (indexes 0 to 2)

2 times when i = 2 (indexes 0 to 1)

1 time when i = 3 (index 0)

0 times when i = 4 (no comparison needed)

3.0.1 Bubble Sort Loop Breakdown (n = 5)

Outer Loop Pass (i)	Inner Loop Range (`j in range(0, n - i - 1)`)	Number of Comparisons	Explanation
i = 0	j = 0 to 3	4 comparisons	Compare first 4 pairs
i = 1	j = 0 to 2	3 comparisons	Last element already sorted
i = 2	j = 0 to 1	2 comparisons	Two largest elements in place
i = 3	j = 0	1 comparison	Three largest elements in place
i = 4	— (loop ends)	0 comparisons	All elements are sorted

[2]:
```
# We will be calling a function named bubble_sort and we will pass it a list
```

```
def bubble_sort(my_list):
    length_of_list = len(my_list)
    length = length_of_list

    for i in range( length-1, 0, -1 ):
        for j in range(i):
            if my_list[j]>my_list[j+1]:
                temp = my_list[j+1]
                my_list[j+1] = my_list[j]
                my_list[j] = temp
    return my_list

bubble_sort([1,3,6,4,2])
```

[2]: `[1, 2, 3, 4, 6]`

length = 5 length - 1 = 4

for i in range(starting from - 4 „„ upto zero ;;; with one step decreament)

Outer Loop Pass (i)	range(i)	Index Pairs Compared	Number of Comparisons
i = 4	(0, 1, 2, 3)	(0,1), (1,2), (2,3), (3,4)	4 comparisons
i = 3	(0, 1, 2)	(0,1), (1,2), (2,3)	3 comparisons
i = 2	(0, 1)	(0,1), (1,2)	2 comparisons
i = 1	(0)	(0,1)	1 comparison

==========

Outer Loop Pass (i)	Inner Loop Range (j in range(i))	Index Pairs Compared	Number of Comparisons
i = 4	j = 0 to 3	(0,1), (1,2), (2,3), (3,4)	4 comparisons
i = 3	j = 0 to 2	(0,1), (1,2), (2,3)	3 comparisons
i = 2	j = 0 to 1	(0,1), (1,2)	2 comparisons
i = 1	j = 0	(0,1)	1 comparison

4 Selection Sort

“Selection Sort” refers to the act of selecting the smallest element in each pass and placing it in its correct position in the sorted part of the list.

I mean, selection in each Pass ;

Replace, swap with the “unsorted part”

Repeat the process (of selection in unsorted part)

4.0.1 Selection Sort: Logic Behind the Name

The name **Selection Sort** comes from the **selection** of the smallest (or largest) element in each pass of the algorithm, which happens in the **unsorted part** of the list.

The Process:

1. **In the first pass**: The algorithm selects the smallest element from the entire list and swaps it with the first element. Now, that element is in the sorted portion of the list, and the remaining unsorted portion has one less element.
2. **In each subsequent pass**: The algorithm selects the smallest element from the **remaining unsorted portion** of the list and places it at the start of the unsorted part by swapping.
3. This **selection** process continues, shrinking the unsorted part and growing the sorted part until the entire list is sorted.

4.0.2 Example (Revised with Focus on "Unsorted Part"):

For a list `[64, 25, 12, 22, 11]`:

- **1st Pass (Unsorted part: [64, 25, 12, 22, 11])**: Select the smallest element (`11`), and swap it with the first element (`64`). The list now looks like: `[11, 25, 12, 22, 64]`.
- **2nd Pass (Unsorted part: [25, 12, 22, 64])**: Select the smallest element (`12`), and swap it with the first element of the unsorted part (`25`). The list now looks like: `[11, 12, 25, 22, 64]`.
- **3rd Pass (Unsorted part: [25, 22, 64])**: Select the smallest element (`22`), and swap it with the first element of the unsorted part (`25`). The list now looks like: `[11, 12, 22, 25, 64]`.
- **4th Pass (Unsorted part: [25, 64])**: No need to swap because `25` is already the smallest element. The list is now sorted: `[11, 12, 22, 25, 64]`.

4.0.3 Key Point:

- **Selection** in each pass refers to the algorithm picking the smallest element from the **unsorted part** and placing it into the **sorted part** by swapping. This continues until the entire list is sorted.

4.0.4 Summary:

- The name **Selection Sort** reflects the idea that, in each pass, we **select** the smallest element from the **unsorted portion** and place it in the correct sorted position.
- The unsorted part keeps shrinking with each pass as we grow the sorted part by one element.

[ ]:
```python
def selection_sort(my_list):
    length = len(my_list)
    for i in range( length - 1, 0, -1):
        for j in range(i):
            if my_list[j]< my_list[j+1]:
                min = my_list[j]
```

```
            else:
                min = my_list[j+1]

# Fuck, how to replace  ? this is the error ; we need to store the index, not␣
 ↪the variable
```

[6]:
```
def selection_sort(my_list):
    length = len(my_list)
    for i in range(length-1):
        min_index = i
        for j in range(i+1, len(my_list) ): # range of (1 to 5) = 1,2,3,4
            if my_list[j]< my_list[min_index]:
                min_index = j
            # else:
            #     min_index = min_index
        if i!= min_index:
            my_list[i], my_list[min_index] = my_list[min_index], my_list[i]
    return my_list

my_list = [64, 25, 12, 22, 11]
selection_sort(my_list)
```

[6]:
```
[11, 12, 22, 25, 64]
```

5 Insertion Sort

5.0.1 Insertion Sort: Logic Behind the Name

The name **Insertion Sort** comes from the way the algorithm **inserts** elements into their correct position in the sorted portion of the list, one element at a time.

Why "Insertion"?

- **Insertion** refers to the process of **inserting** an element from the unsorted portion into the correct position in the sorted portion.
- The algorithm works by taking one element at a time from the unsorted part of the list and **inserting it into its correct position** in the sorted part of the list.

How the Name Works:

1. **In the first step**: The algorithm assumes the first element is already sorted (since a single element is trivially sorted).
2. **In each subsequent step**: The algorithm takes the next element from the unsorted part of the list and **inserts it** into the correct position in the sorted part by shifting the larger elements to the right.
3. This process of insertion continues until all the elements are sorted.

5.0.2 Example (for List [5, 2, 9, 1, 5, 6]):

- **Step 1**: The first element `5` is assumed to be in the sorted part.
- **Step 2**: The next element `2` is taken and **inserted** into the sorted part by shifting `5` to the right. The list becomes: `[2, 5, 9, 1, 5, 6]`.
- **Step 3**: The next element `9` is already in the correct position, so no changes are made. The list remains: `[2, 5, 9, 1, 5, 6]`.
- **Step 4**: The next element `1` is inserted into its correct position by shifting `9`, `5`, and `2` to the right. The list becomes: `[1, 2, 5, 9, 5, 6]`.
- **Step 5**: The next element `5` is inserted into its correct position, shifting `9` to the right. The list becomes: `[1, 2, 5, 5, 9, 6]`.
- **Step 6**: The final element `6` is inserted into its correct position by shifting `9` to the right. The list becomes: `[1, 2, 5, 5, 6, 9]`.

5.0.3 Key Point:

- The name **Insertion Sort** is derived from the fact that elements are **inserted** into their correct position in the sorted part of the list during each pass.

5.0.4 Summary:

- **Insertion Sort** is named after the **insertion** of each element from the unsorted portion into its correct position in the sorted part.
- The process continues with one element being inserted at a time until the entire list is sorted.

[ ]:
```python
def insertion_sort(my_list):
    length = len(my_list)
    for i in range(0, length-1, 1): # (i = 0, 1, 2, 3, 4, 5)
        for j in range(i+1, 0 , -1):
            min_index = i
            if my_list[j]<my_list[j-1]: # Since there is j-1; so we not need to␣
 ↪run upto 0
                temp = my_list[j]
                my_list[j] = my_list[j-1]
                my_list[j-1] = temp
            print("i :",i," | j :",j,"| list :",my_list)
        print(f'--------------------')
    # return my_list

my_list = [5, 2, 9, 1, 5, 6]
insertion_sort(my_list)
```

```
i : 0  | j : 1 | list : [2, 5, 9, 1, 5, 6]
--------------------
i : 1  | j : 2 | list : [2, 5, 9, 1, 5, 6]
i : 1  | j : 1 | list : [2, 5, 9, 1, 5, 6]
--------------------
```

```
i : 2  | j : 3 | list : [2, 5, 1, 9, 5, 6]
i : 2  | j : 2 | list : [2, 1, 5, 9, 5, 6]
i : 2  | j : 1 | list : [1, 2, 5, 9, 5, 6]
--------------------
i : 3  | j : 4 | list : [1, 2, 5, 5, 9, 6]
i : 3  | j : 3 | list : [1, 2, 5, 5, 9, 6]
i : 3  | j : 2 | list : [1, 2, 5, 5, 9, 6]
i : 3  | j : 1 | list : [1, 2, 5, 5, 9, 6]
--------------------
i : 4  | j : 5 | list : [1, 2, 5, 5, 6, 9]
i : 4  | j : 4 | list : [1, 2, 5, 5, 6, 9]
i : 4  | j : 3 | list : [1, 2, 5, 5, 6, 9]
i : 4  | j : 2 | list : [1, 2, 5, 5, 6, 9]
i : 4  | j : 1 | list : [1, 2, 5, 5, 6, 9]
--------------------
```

5.0.5 Insertion Sort – Detailed Table with i and j, Showing Element Comparisons

Pass	i	j	Action Taken	List State
1	0	1	(j=1)<(j=0) i.e. $2 < 5 \rightarrow$ swap	[2, 5, 9, 1, 5, 6]
2	1	2	(j=2)<(j=1) i.e. $9 < 5 \rightarrow$ no swap	[2, 5, 9, 1, 5, 6]
		1	(j=1)<(j=0) i.e. $5 < 2 \rightarrow$ no swap	[2, 5, 9, 1, 5, 6]
3	2	3	(j=3)<(j=2) i.e. $1 < 9 \rightarrow$ swap	[2, 5, 1, 9, 5, 6]
		2	(j=2)<(j=1) i.e. $1 < 5 \rightarrow$ swap	[2, 1, 5, 9, 5, 6]
		1	(j=1)<(j=0) i.e. $1 < 2 \rightarrow$ swap	[1, 2, 5, 9, 5, 6]
4	3	4	(j=4)<(j=3) i.e. $5 < 9 \rightarrow$ swap	[1, 2, 5, 5, 9, 6]
		3	(j=3)<(j=2) i.e. $5 < 5 \rightarrow$ no swap	[1, 2, 5, 5, 9, 6]
		2	(j=2)<(j=1) i.e. $5 < 2 \rightarrow$ no swap	[1, 2, 5, 5, 9, 6]
		1	(j=1)<(j=0) i.e. $2 < 1 \rightarrow$ no swap	[1, 2, 5, 5, 9, 6]
5	4	5	(j=5)<(j=4) i.e. $6 < 9 \rightarrow$ swap	[1, 2, 5, 5, 6, 9]
		4	(j=4)<(j=3) i.e. $6 < 5 \rightarrow$ no swap	[1, 2, 5, 5, 6, 9]
		3	(j=3)<(j=2) i.e. $5 < 5 \rightarrow$ no swap	[1, 2, 5, 5, 6, 9]
		2	(j=2)<(j=1) i.e. $5 < 2 \rightarrow$ no swap	[1, 2, 5, 5, 6, 9]
		1	(j=1)<(j=0) i.e. $2 < 1 \rightarrow$ no swap	[1, 2, 5, 5, 6, 9]

5.0.6 Alternative Version:

[24]:
```python
#  Alternative Version: (but optimal one)
def insertion_sort(my_list):
    for i in range(1, len(my_list)):
        temp = my_list[i]
        j = i - 1
        while temp < my_list[j] and j > -1:
```

```
            my_list[j + 1] = my_list[j]
            my_list[j] = temp
            j -= 1
    return my_list

insertion_sort(my_list)
```

[24]: `[1, 2, 5, 5, 6, 9]`

5.1 Detailed Comparison

Feature	Your Original Version	Alternative Version
Outer Loop (i)	`for i in range(0, length-1)`	`for i in range(1, len(my_list))`
Inner Loop (j)	`for j in range(i+1, 0, -1)`	`while temp < my_list[j] and j > -1`
Swap or Shift	Swaps two elements on each comparison	Shifts elements and inserts `temp` once
Efficiency	Slower due to multiple swaps	More efficient due to fewer assignments
Style	Simple nested loops	Cleaner logic with clear shifting
Write Operations	Multiple writes per insertion	Only necessary shifts and a single placement
Return Value	Missing `return` (unless added)	Properly returns the sorted list

---

5.2 Key Differences

1. **The second version is more optimal**:
   - It minimizes the number of assignments by **shifting** elements instead of swapping on every comparison.
   - This improves performance in practice, especially for larger lists.
2. **Cleaner logic**:
   - The `while` loop continues as long as `temp` is smaller and we haven't hit the start of the list.
   - Less indexing and fewer operations than nested loops.
3. **Functional return**:
   - The second version correctly returns the sorted list, whereas the first one does not unless explicitly added.

[25]:
```
#  Alternative Version: (but optimal one)
def insertion_sort(my_list):
    for i in range(1, len(my_list)):
        temp = my_list[i]
        j = i - 1
        while temp < my_list[j] and j > -1:
```

```
            my_list[j + 1] = my_list[j]
            my_list[j] = temp
            j -= 1
    return my_list

insertion_sort(my_list)
```

[25]: `[1, 2, 5, 5, 6, 9]`

5.3 Impact of Removing `j > -1` Condition in the `while` Loop

Step	i	j	Condition (`temp < my_list[j]`)	Action with `j > -1`	Action without `j > -1`	List After Action
Initial List				Initial list: `[2, 1, 3, 4, 5, 6]`	Initial list: `[2, 1, 3, 4, 5, 6]`	`[2, 1, 3, 4, 5, 6]`
1st Iteration (i=1)	1	0	`1 < 2` → **True**	2 is shifted right, `temp` becomes 1.	2 is shifted right, `temp` becomes 1.	`[1, 2, 3, 4, 5, 6]`
After Shift (i=1, j=0)	1	0	`1 < 2` → **True**	No further shifting, since `j` becomes `-1`, stops the loop.	**Error occurs**: Tries to access `my_list[-1]`, causing an `IndexError`.	Crashes with error.
Final List (with `j > -1`)				Safely stops shifting, final list: `[1, 2, 3, 4, 5, 6]`	**Not applicable** due to error.	**Final sorted list unavailable**

=====

Step	With `j > -1` Condition	Without `j > -1` Condition
Initial List	`[2, 1, 3, 4, 5, 6]`	`[2, 1, 3, 4, 5, 6]`
First Iteration (`i = 1`, temp = 1)	Compare `temp (1)` with `my_list[0] (2)` → `1 < 2` → **True**, shift 2	Compare `temp (1)` with `my_list[0] (2)` → `1 < 2` → **True**, shift 2
List After First Shift	`[1, 2, 3, 4, 5, 6]`	`[1, 2, 3, 4, 5, 6]`
Now `j = 0`	Compare `temp (1)` with `my_list[-1]` → **Valid**: Ends loop (no index error).	Compare `temp (1)` with `my_list[-1]` → **Invalid**: Causes `IndexError`.

Step	With `j > -1` Condition	Without `j > -1` Condition
Result	The `while` loop stops safely when `j > -1`, avoiding out-of-bounds error.	Without `j > -1`, `j = -1` leads to an **IndexError** when accessing `my_list[-1]`.
