**Contents**

[**Linear search** 2](#_Toc178921281)

[**Binary search** 4](#_Toc178921282)

[**Bubble Sort** 7](#_Toc178921283)

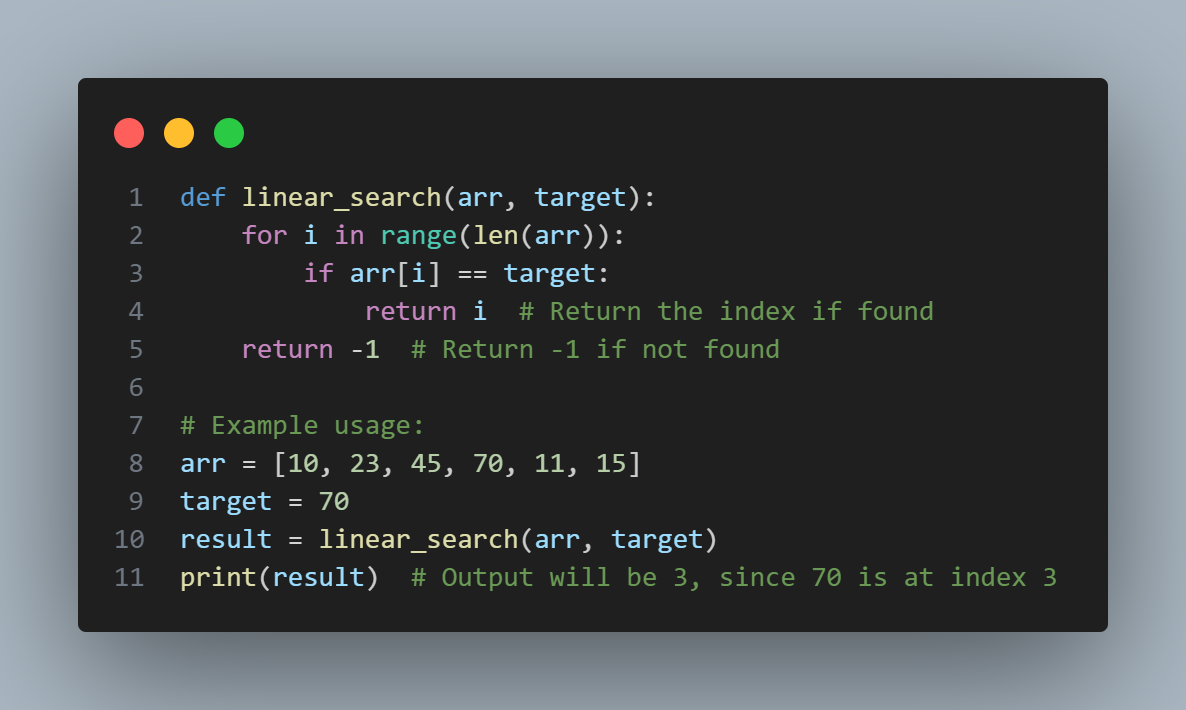
[**Insertion Sort** 11](#_Toc178921284)

# **Linear search**

***Linear search***, also known as ***sequential search***, is a straightforward algorithm used to search for an element in a list or array by checking each element one by one, from the beginning to the end, until the desired element is found or the list is exhausted.

**Steps in Linear Search:**

* Start from the first element of the list.
* Compare the current element with the target element (the one you're searching for).
* If the current element matches the target, return the index of the element.
* If the current element does not match the target, move to the next element.
* Repeat steps 2-4 until you either find the target or reach the end of the list.
* If the target is not found, return a message (or a specific value) indicating the target is not present in the list.



**Time Complexity of Linear Search**

**Best Case Time Complexity:**

* **O(1):** The best-case scenario occurs when the target element is at the first position of the list. In this case, the algorithm only needs to make one comparison, which is a constant-time operation.
* For example, if I want to search 10 from the above array then the 10 will be found at the first position, so the time complexity will be ***O(1)***.

**Worst Case Time Complexity:**

* **O(n):** The worst-case scenario happens when the target element is either at the last position of the list or not present at all. In such cases, the algorithm has to check all **`n`** elements, where **`n`** is the length of the list.

**Average Case Time Complexity:**

* **O(n):** On average, the target element might be somewhere in the middle of the list. On average, the algorithm will have to check about half of the elements, but in terms of Big-O notation, this is still considered ***O(n)***.

**Space Complexity of Linear Search**

* **O(1):** The space complexity of linear search is constant because the algorithm only uses a few additional variables to store the current index and the target element. No additional space is required that depends on the size of the input.

**Summary of Time Complexity:**

|  |  |
| --- | --- |
| **Scenario** | **Time complexity** |
| *Best case* | *O(1)* |
| *Worst case* | *O(n)* |
| *Average case* | *O(n)* |

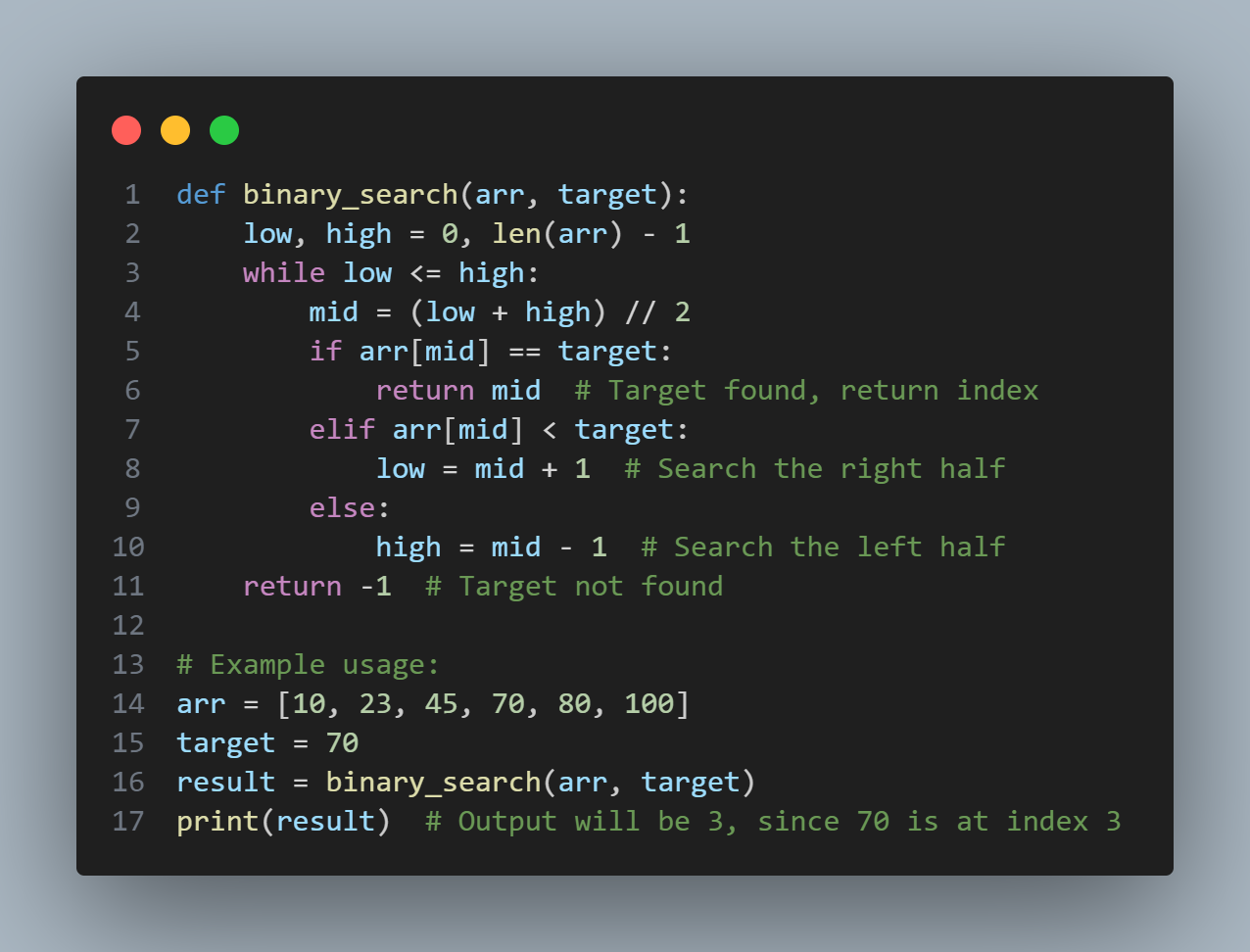
Linear search is simple but not the most efficient for large datasets. For sorted data, other algorithms like binary search ***(O(log n))*** are typically preferred. However, linear search is applicable to both sorted and unsorted data, unlike binary search, which requires sorted data.

# **Binary search**

***Binary search*** is an efficient algorithm used to find an element in a **sorted** list by repeatedly dividing the search interval in half. If the target value is less than the middle element of the interval, the search continues in the left half; otherwise, it continues in the right half. This process is repeated until the target value is found or the search interval is empty. Binary search follows the **`Divide and Conquer`** approach.

**Steps in Binary Search for normal way:**

* Start with two pointers, `low` and `high`, representing the start and end of the search range.
* Calculate the middle index `mid` of the current search range as:
* Compare the element at the middle index **`arr[mid]`** with the target:
* If **`arr[mid]`** equals the target, return the index `mid`.
* If **`arr[mid]`** is greater than the target, set **`high = mid - 1`** and search the left half.
* If **`arr[mid]`** is less than the target, set **`low = mid + 1`** and search the right half.
* Repeat steps 2 and 3 until `low` exceeds `high`, meaning the target is not present.



**Time Complexity of Binary Search**

Binary search is significantly more efficient than linear search for large datasets because it reduces the search space by half after each comparison.

**Best Case Time Complexity:**

* **O(1):** The best case occurs when the target is found at the middle index on the first attempt. In this case, the algorithm only needs to make one **`comparison`**, which is a constant-time operation.

**Worst Case and Average Case Time Complexity:**

* **O(log n):** In the worst case, binary search will repeatedly halve the search range until only one element remains. The number of steps required to reduce a list of size **`n`** to a single element is proportional to the base-2 logarithm of **`n`**.
* For a list of size `n`, the number of comparisons made is at most: ***log2 (n) + 1***

This is why the time complexity in both the average and worst cases is ***O(log n)***. In each iteration, the search space is reduced by half, which results in a logarithmic growth in the number of steps required.

**Explanation of Time Complexity:**

* **Best Case (O(1)):** The target is the middle element on the first comparison.
* **Worst Case (O(log n)):** In the worst case, the search space has to be halved repeatedly until only one element remains. The number of times you can halve a list of size **`n`** is the number of times you can divide **`n`** by 2 before you reach **1**. This is proportional to ***log base 2 of n***.
* **Average Case (O(log n)):** On average, binary search will locate the target after a few iterations of halving the list. Since the process is logarithmic, the average time complexity is also **O(log n)**.

**Example Time Complexity Breakdown:**

For a list of size `n = 16`:

* After 1 iteration, you check 8 elements (half the list).
* After 2 iterations, you check 4 elements.
* After 3 iterations, you check 2 elements.
* After 4 iterations, you check 1 element (or find the target).

In general, the number of iterations required is around ***log₂(n)***.

**Space Complexity of Binary Search**

**O(1)** for iterative implementations: The algorithm uses only a few extra variables ***(`low`, `high`, `mid`)***, so the space complexity is constant.

**O(log n)** for recursive implementations: In recursive implementations, each recursive call adds a new frame to the call stack, which requires ***(log n)*** space in the worst case.

**Summary of Time Complexity:**

|  |  |
| --- | --- |
| **Scenario** | **Time complexity** |
| *Best case* | *O(1)* |
| *Worst case* | *O(log n)* |
| *Average case* | *O(log n)* |

**Key Points:**

* Binary search is much faster than linear search for large, sorted datasets.
* It requires the array to be **sorted** for it to work correctly.

# **Bubble Sort**

Bubble Sort is one of the simplest sorting algorithms that works by repeatedly stepping through a list, comparing adjacent elements, and swapping them if they are in the wrong order. The algorithm gets its name because smaller elements "bubble" to the top of the list while larger elements "sink" to the bottom.

**Steps of the Bubble Sort Algorithm:**

* Start at the beginning of the list and compare the first two elements (element 1 and element 2).
  + If element 1 is greater than element 2, swap them.
  + If element 1 is less than or equal to element 2, leave them as they are.
* Move to the next pair of adjacent elements (element 2 and element 3) and repeat the comparison and swapping if necessary.
* Continue this process for all elements in the list. After the first pass (one complete traversal through the list), the largest element will have "bubbled" to its correct position at the end of the list.
* Repeat the process for the remaining unsorted portion of the list. Each subsequent pass places the next largest unsorted element in its correct position.
* The algorithm repeats this process until no swaps are needed during a full pass, meaning the list is sorted.

**Example Walkthrough:**

Let’s assume we have the following array: **[5, 1, 4, 2, 8]**

* **First Pass:**
  1. Compare 5 and 1. Since 5 > 1, swap them → [1, 5, 4, 2, 8].
  2. Compare 5 and 4. Since 5 > 4, swap them → [1, 4, 5, 2, 8].
  3. Compare 5 and 2. Since 5 > 2, swap them → [1, 4, 2, 5, 8].
  4. Compare 5 and 8. Since 5 < 8, leave them as is.

After the first pass, the largest element (8) is correctly placed at the end of the list.

* **Second Pass:**

1. Compare 1 and 4. Since 1 < 4, leave them as is.
2. Compare 4 and 2. Since 4 > 2, swap them → [1, 2, 4, 5, 8].
3. Compare 4 and 5. Since 4 < 5, leave them as is.

No swaps are needed for the last element since it’s already sorted.

* **Third Pass:**

1. Compare 1 and 2. Since 1 < 2, leave them as is.
2. Compare 2 and 4. Since 2 < 4, leave them as is.

No swaps were made in this pass, which means the list is now fully sorted.

**Final sorted list: [1, 2, 4, 5, 8]**

**For Clearly understanding: if the array size is ‘n’ then number of iterations will be (n-1) = (5-1) = 4 or 4 pass.**

**Pass-1:**

**[22, 14, 12, 18, 9]**

**[14, 22, 12, 18, 9]**

**[14, 12, 22, 18, 9]**

**[14, 12, 18, 22, 9]**

**[14, 12, 18, 9, 22]**

**Pass-2:**

**[14, 12, 18, 9, 22]**

**[12, 14, 18, 9, 22]**

**[12, 14, 18, 9, 22]**

**[12, 14, 9, 18, 22]**

**Pass-3:**

**[12, 14, 9, 18, 22]**

**[12, 14, 9, 18, 22]**

**[12, 9, 14, 18, 22]**

**Pass-4:**

**[12, 9, 14, 18, 22]**

**[9, 12, 14, 18, 22] 🡪 Final sorted list**

**Efficiency of Bubble Sort:**

**Number of Comparisons:**

* The **yellow** color represent the comparison.
* For each element in the list, Bubble Sort compares it with the next element, resulting in about ***n-1*** comparisons for each pass.
* For pass-1 number of comparisons 4, for pass-2 number of comparisons 3, and for pass-3, pass-4 number of comparisons 2, 1 respectively. So we can write:

**Swaps:**

* The **red text** color represent the swap.
* Bubble Sort swaps elements when necessary. In the **worst case**, the list may be **completely reversed**, meaning that each comparison would lead to a swap.

**Time Complexity:**

* **Best Case (Already Sorted):** O(n)  
  In the best case, when the list is already sorted, Bubble Sort only needs to make one pass through the list, with no swaps necessary. After the first pass, the algorithm can detect that the list is sorted, resulting in O(n) comparisons.
* **Average Case:** O(n²)  
  On average, Bubble Sort requires multiple passes through the list, where each pass performs O(n) comparisons and possibly some swaps. As a result, the average time complexity is quadratic.
* **Worst Case (Reversed List):** O(n²)  
  In the worst case (a list sorted in reverse order), Bubble Sort must make the maximum number of comparisons and swaps. It would take O(n) passes, with each pass performing O(n) comparisons, resulting in a time complexity of O(n²).

**Space Complexity:**

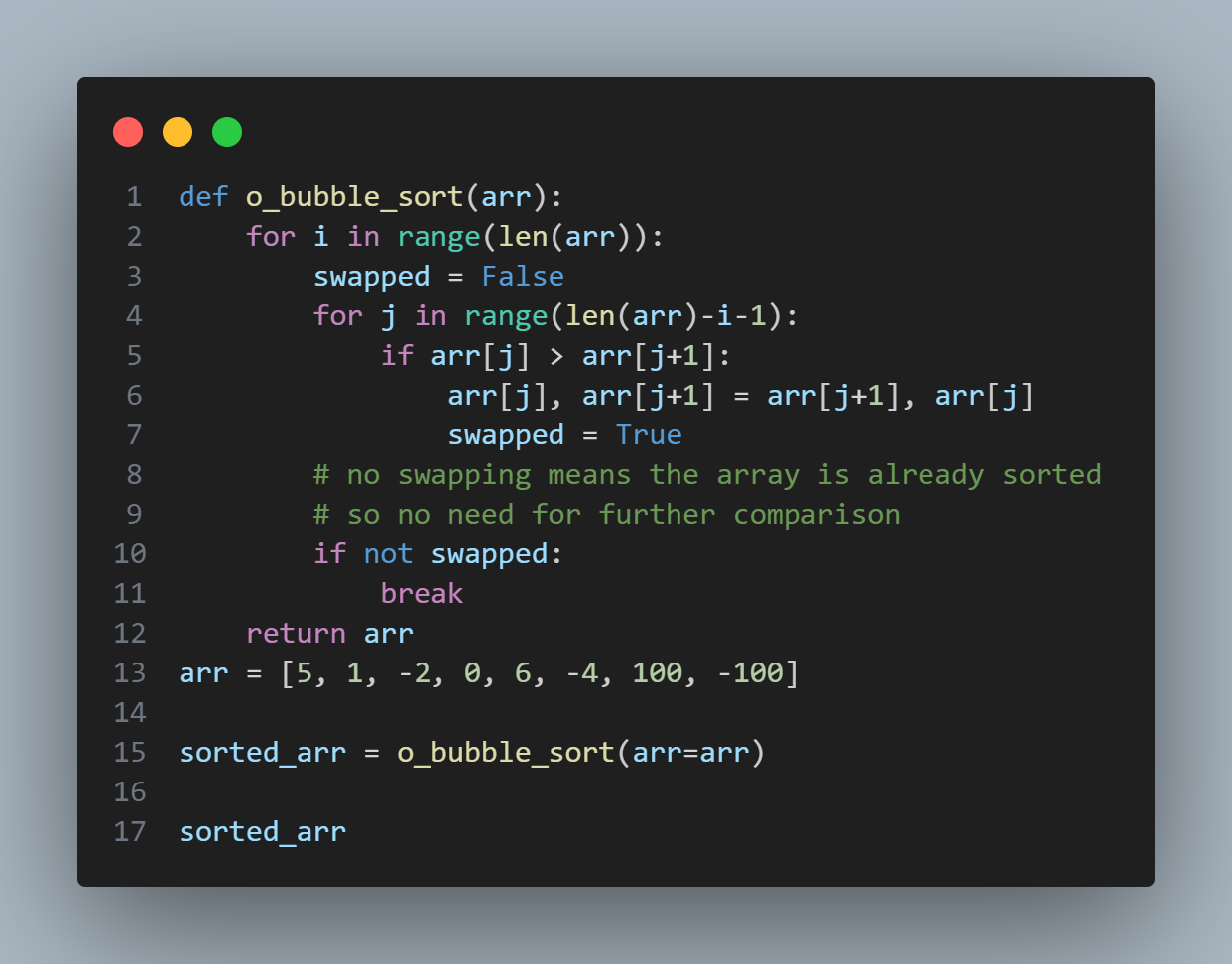
* **Space Complexity:** O(1)  
  Bubble Sort is an **in-place** sorting algorithm, meaning it does not require any additional storage space apart from a few variables for tracking comparisons and swaps. Therefore, its space complexity is constant (O(1)).

**Stability of Bubble Sort:**

* **Stable Sorting Algorithm:** Yes  
  Bubble Sort is a stable algorithm, meaning that two elements with equal values will remain in the same order relative to each other after sorting. This is because Bubble Sort only swaps adjacent elements if the first is greater than the second, so equal elements are not swapped.

**Summary of Time Complexity:**

|  |  |
| --- | --- |
| **Scenario** | **Time complexity** |
| *Best case* | *O(n)* |
| *Worst case* | *O(n2)* |
| *Average case* | *O(n2)* |



# **Insertion Sort**

Bubble Sort is one of the simplest sorting algorithms that works by repeatedly stepping through a list, comparing adjacent elements, and swapping them if they are in the wrong order. The algorithm gets its name because smaller elements "bubble" to the top of the list while larger elements "sink" to the bottom.

**Steps of the Bubble Sort Algorithm:**

* Start at the beginning of the list and compare the first two elements (element 1 and element 2).
  + If element 1 is greater than element 2, swap them.
  + If element 1 is less than or equal to element 2, leave them as they are.
* Move to the next pair of adjacent elements (element 2 and element 3) and repeat the comparison and swapping if necessary.
* Continue this process for all elements in the list. After the first pass (one complete traversal through the list), the largest element will have "bubbled" to its correct position at the end of the list.
* Repeat the process for the remaining unsorted portion of the list. Each subsequent pass places the next largest unsorted element in its correct position.
* The algorithm repeats this process until no swaps are needed during a full pass, meaning the list is sorted.

**Example Walkthrough:**

Let’s assume we have the following array: **[5, 1, 4, 2, 8]**

* **First Pass:**
  1. Compare 5 and 1. Since 5 > 1, swap them → [1, 5, 4, 2, 8].
  2. Compare 5 and 4. Since 5 > 4, swap them → [1, 4, 5, 2, 8].
  3. Compare 5 and 2. Since 5 > 2, swap them → [1, 4, 2, 5, 8].
  4. Compare 5 and 8. Since 5 < 8, leave them as is.

After the first pass, the largest element (8) is correctly placed at the end of the list.

* **Second Pass:**

1. Compare 1 and 4. Since 1 < 4, leave them as is.
2. Compare 4 and 2. Since 4 > 2, swap them → [1, 2, 4, 5, 8].
3. Compare 4 and 5. Since 4 < 5, leave them as is.

No swaps are needed for the last element since it’s already sorted.

* **Third Pass:**

1. Compare 1 and 2. Since 1 < 2, leave them as is.
2. Compare 2 and 4. Since 2 < 4, leave them as is.

No swaps were made in this pass, which means the list is now fully sorted.

**Final sorted list: [1, 2, 4, 5, 8]**

**For Clearly understanding: if the array size is ‘n’ then number of iterations will be (n-1) = (5-1) = 4 or 4 pass.**

**Pass-1:**

**[22, 14, 12, 18, 9]**

**[14, 22, 12, 18, 9]**

**[14, 12, 22, 18, 9]**

**[14, 12, 18, 22, 9]**

**[14, 12, 18, 9, 22]**

**Pass-2:**

**[14, 12, 18, 9, 22]**

**[12, 14, 18, 9, 22]**

**[12, 14, 18, 9, 22]**

**[12, 14, 9, 18, 22]**

**Pass-3:**

**[12, 14, 9, 18, 22]**

**[12, 14, 9, 18, 22]**

**[12, 9, 14, 18, 22]**

**Pass-4:**

**[12, 9, 14, 18, 22]**

**[9, 12, 14, 18, 22] 🡪 Final sorted list**

**Efficiency of Bubble Sort:**

**Number of Comparisons:**

* The **yellow** color represent the comparison.
* For each element in the list, Bubble Sort compares it with the next element, resulting in about ***n-1*** comparisons for each pass.
* For pass-1 number of comparisons 4, for pass-2 number of comparisons 3, and for pass-3, pass-4 number of comparisons 2, 1 respectively. So we can write:

**Swaps:**

* The **red text** color represent the swap.
* Bubble Sort swaps elements when necessary. In the **worst case**, the list may be **completely reversed**, meaning that each comparison would lead to a swap.

**Time Complexity:**

* **Best Case (Already Sorted):** O(n)  
  In the best case, when the list is already sorted, Bubble Sort only needs to make one pass through the list, with no swaps necessary. After the first pass, the algorithm can detect that the list is sorted, resulting in O(n) comparisons.
* **Average Case:** O(n²)  
  On average, Bubble Sort requires multiple passes through the list, where each pass performs O(n) comparisons and possibly some swaps. As a result, the average time complexity is quadratic.
* **Worst Case (Reversed List):** O(n²)  
  In the worst case (a list sorted in reverse order), Bubble Sort must make the maximum number of comparisons and swaps. It would take O(n) passes, with each pass performing O(n) comparisons, resulting in a time complexity of O(n²).

**Space Complexity:**

* **Space Complexity:** O(1)  
  Bubble Sort is an **in-place** sorting algorithm, meaning it does not require any additional storage space apart from a few variables for tracking comparisons and swaps. Therefore, its space complexity is constant (O(1)).

**Stability of Bubble Sort:**

* **Stable Sorting Algorithm:** Yes  
  Bubble Sort is a stable algorithm, meaning that two elements with equal values will remain in the same order relative to each other after sorting. This is because Bubble Sort only swaps adjacent elements if the first is greater than the second, so equal elements are not swapped.

**Summary of Time Complexity:**

|  |  |
| --- | --- |
| **Scenario** | **Time complexity** |
| *Best case* | *O(n)* |
| *Worst case* | *O(n2)* |
| *Average case* | *O(n2)* |