Bubble Sort Algorithm

Bubble Sort is one of the simplest sorting algorithms. As shown in Listing 1, the algorithm repeatedly swaps adjacent elements until the list is sorted. While this is useful for teaching basic algorithmic principles, it performs extremely poorly on larger datasets (GeeksforGeeks, n.d.). In this report, Bubble Sort's performance will be compared with Tim Sort (a hybrid of Merge Sort and Insertion Sort). Timsort is highly efficient for large-scale sorting tasks.

Listing 1: Python implementation of Bubble Sort. Inline comments have been deliberately kept demonstrating the reasoning behind each step.

```
def bubble_sort(sorted_list):
    n = len(sorted_list)
    steps = 0  # Counts how many steps the algorithm takes
    for _ in range(n):
        swapped = False # no elements have been swapped yet
        for i in range(n - 1): #The second-to-last element, compare with (i+1)
            steps += 1 # increment the step counter
            if sorted_list[i] > sorted_list[i + 1]:
               # This is the traditional way to swap two values
                temp = sorted_list[i]
                sorted_list[i] = sorted_list[i + 1]
                sorted_list[i + 1] = temp
                swapped = True #at least one element has been swapped
        if not swapped:
            break
    return sorted_list, steps
```

Results:

List size	Steps	Time (s)
10	54	0.000103
50	2,107	0.002469
100	8,910	0.009685
1000	926,073	1.113802

Table 1: Bubble sort performance for different dataset sizes (Source: Authors’ own experiment). The raw console output is included in Appendix B (Figure B1).

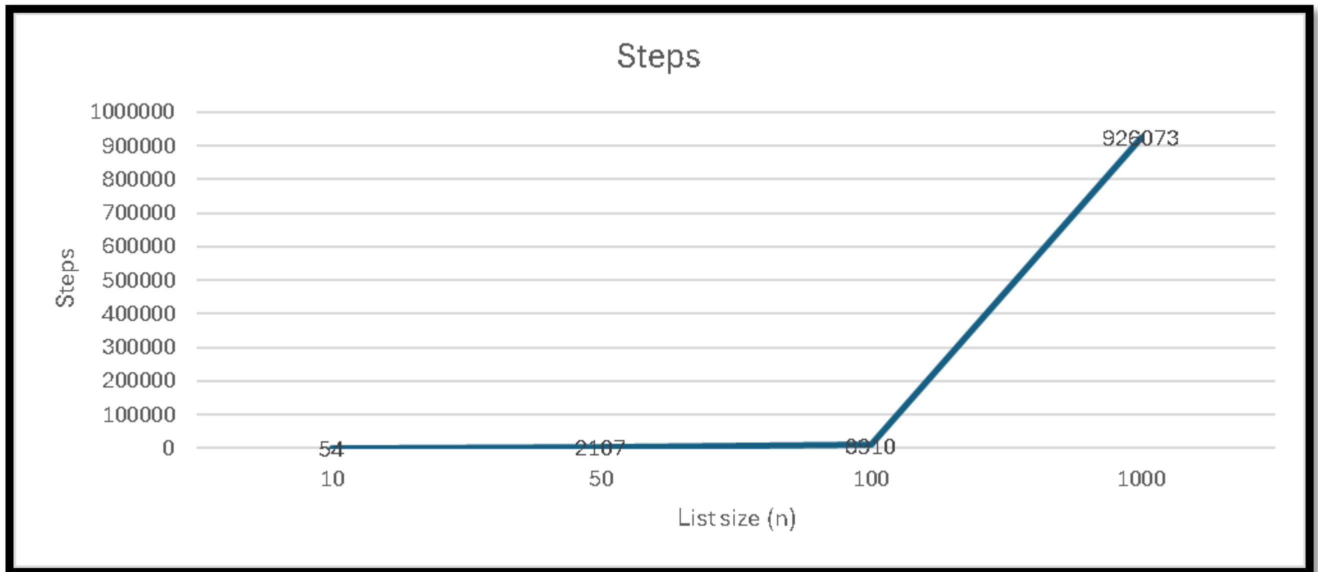

Figure 1: Bubble Sort performance by dataset size, number of steps for different inputs.

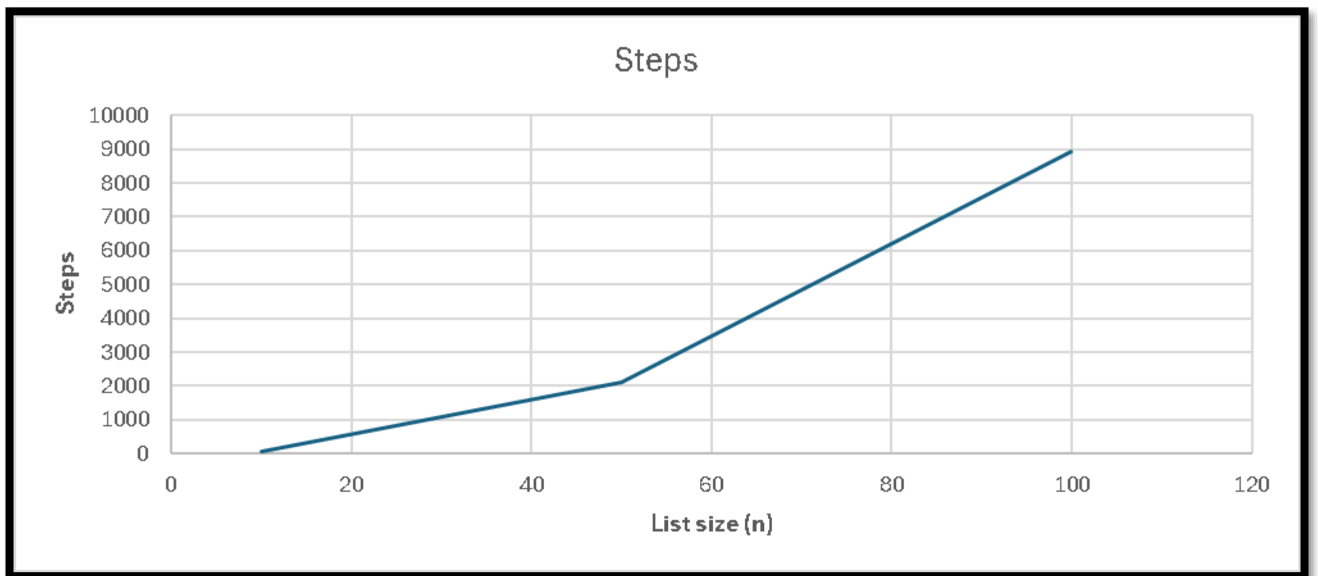

Figure 2: Bubble Sort performance zoomed in on smaller input sizes.

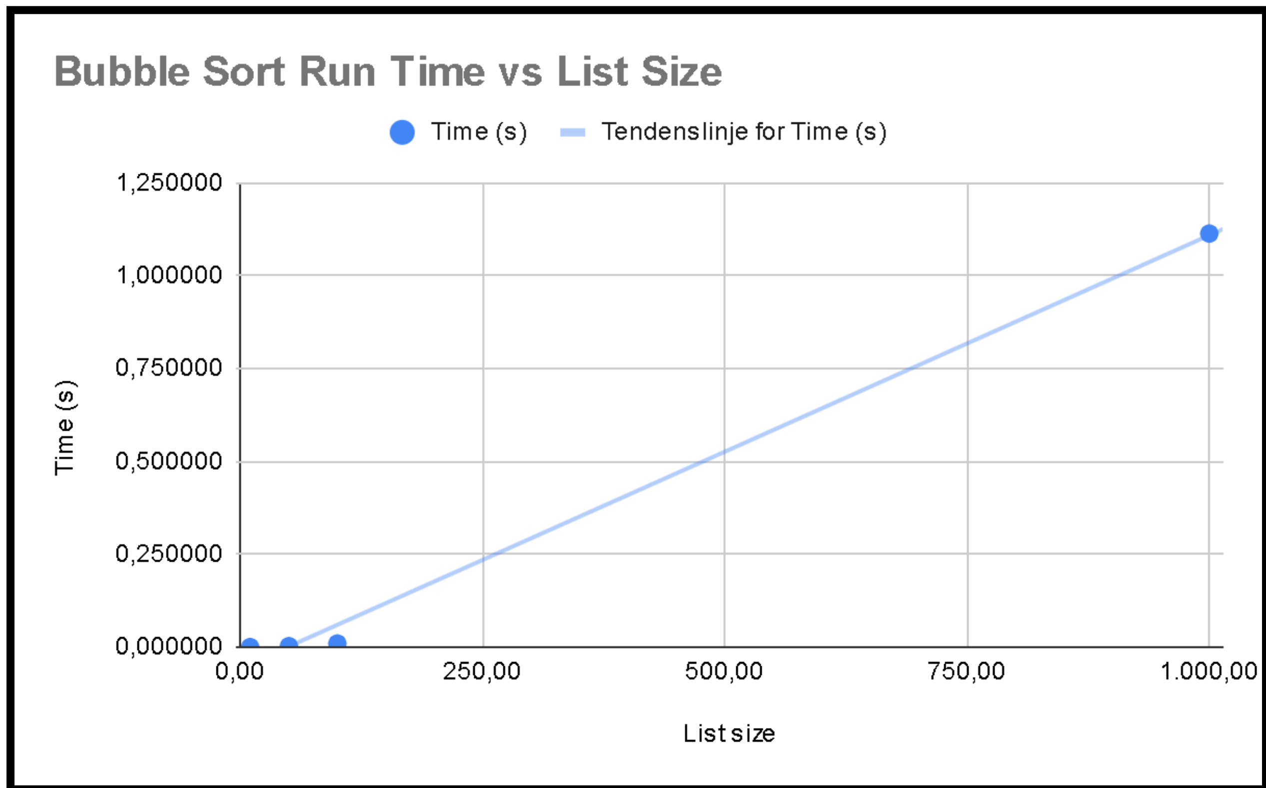

Figure 3: Bubble Sort runtime against input size, showing rapid growth as the dataset increases.

Big-O definition

According to Schwarz (n.d), once the Big-O complexity of an algorithm is known, its runtime can be estimated and compared to other implementations.

Figures 4 and 5 below illustrate the different complexity classes and their corresponding growth rates.

Notation	Name
$O(1)$	Constant
$O(\log(n))$	Logarithmic
$O(\log(\log(n))$	Double logarithmic (iterative logarithmic)
$o(n)$	Sublinear
$O(n)$	Linear
$O(n\log(n))$	Loglinear, Linearithmic, Quasilinear or Supralinear
$O(n^2)$	Quadratic
$O(n^3)$	Cubic
$O(n^c)$	Polynomial (different class for each $c > 1$)
$O(c^n)$	Exponential (different class for each $c > 1$)
$O(n!)$	Factorial
$O(n^n)$	- (Yuck!)

Figure 4: Common algorithmic complexity classes in Big-O notation (Source: Ryerson University, n.d.).

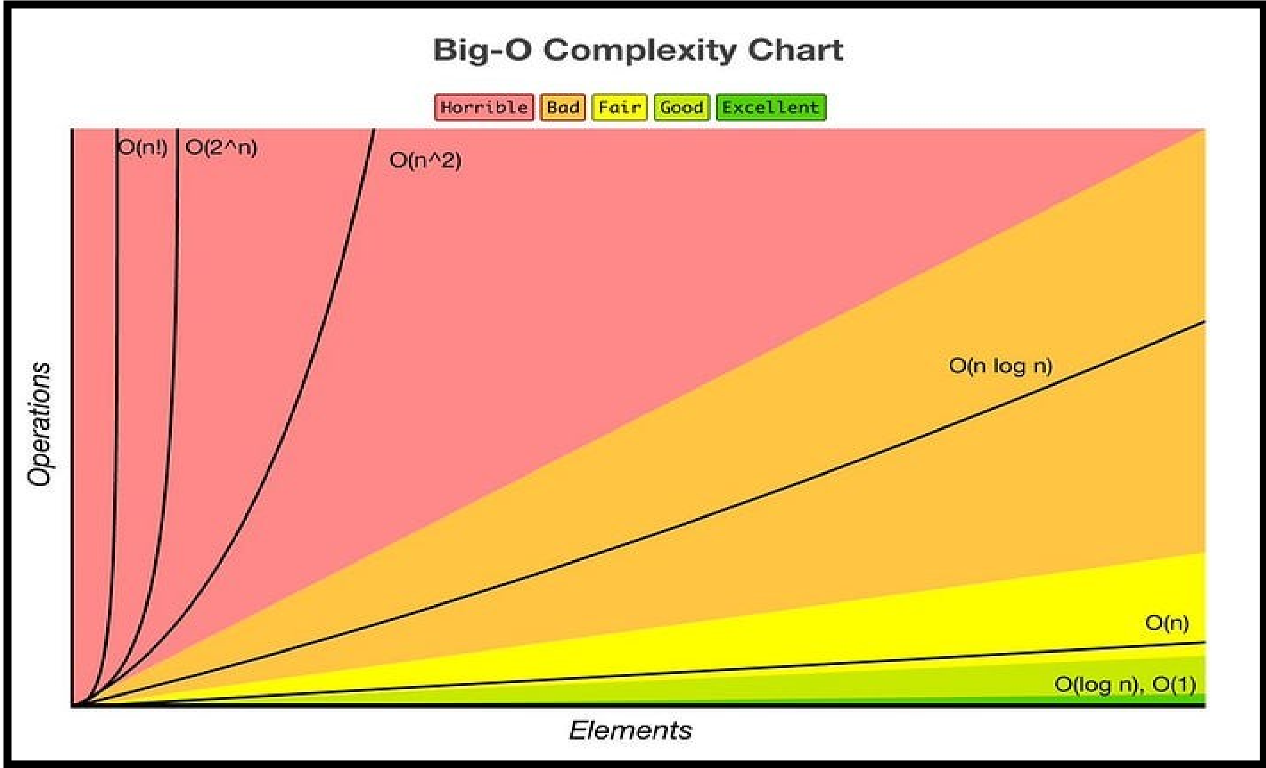

Figure 5: Big-O Complexity Chart showing typical growth rates O(1), O(log n), O(n), O(n log n), $O(n^2)$, $O(2^n)$ (Source: “The Big-Oh (O) – A Beginner’s Guide”, Medium, n.d.).

Manual Big-O Estimate for Bubble Sort

This implementation for Bubble Sort in Python uses two counters: Comparisons (neighbor checks) and Swaps (adjacent exchanges).

- **Comparisons**: The algorithm checks n neighboring pairs and repeats this n times until no operations occur -> n x n = n^2, in the worst case.
  In the best case, the list is already sorted, and it stops after the first iteration(n checks) => O(n).
- **Swaps:** A swap happens whenever a neighboring pair is out of order. In the worst case, there are n(n-1)/2 pairs out of order => =(n^2) swaps.
  Best case is the same as with comparisons, the list is already sorted => 0 operations.

Result: Bubble Sort has quadratic (Figure 4) time complexity, O(n^2).

Comparison: TimSort vs. Merge Sort — real-world use

Bubble Sort is very easy to learn, but inefficient, because it has O(n^2) complexity. As Hanafi et al. (2022) show, Timsort is different. It is designed to be much faster in practice. It guarantees O(n log n) in the worst case and near-linear performance on partially sorted lists. The study by Hanafi et al. (2022) clearly shows that Timsort is much faster than Bubble Sort for all dataset sizes.

Conclusion

Bubble Sort has relevance mainly for teaching purposes. In contrast, Timsort is efficient and the default sorting algorithm in Python and Java, proving its suitability for practical, large-scale applications.

Methods note: The experiments were implemented in Python, and the performance data was visualised using both Google Sheets and Microsoft Excel to demonstrate the use of different tools.

Parts of this report have been proofread and refined with the assistance of ChatGPT. All analysis, coding, and results are the author's work.

References

GeeksforGeeks (2024) Bubble Sort Algorithm. Available at: https://www.geeksforgeeks.org/bubble-sort/ (Accessed: 14 August 2025).

Hanafi, M.R., Faadhilah, M.A., Dwi Putra, M.T. and Pradeka, D. (2022) Comparison analysis of Bubble Sort algorithm with Tim Sort algorithm sorting against the amount of data. COELITE: Journal of Computer Engineering, Electronics and Information Technology, 1(1). Universitas Pendidikan Indonesia. Available at: https://doi.org/10.17509/coelite.v1i1.43794.

Medium (n.d.) The Big-O (O) – A Beginner's Guide. Available at: https://articles.wesionary.team/the-big-oh-o-a-beginners-guide-a4c48af39bfa (Accessed: 14 August 2025).

Python Software Foundation (n.d.) time — Time access and conversions. Available at: https://docs.python.org/3/library/time.html#time.perf_counter (Accessed: 14 August 2025).

Ryerson University (n.d.) Big-O notation. Available at: https://www.cs.ryerson.ca/~mth210/Handouts/PD/bigO.pdf (Accessed: 14 August 2025).

Schwarz, K. (n.d.) CS106B Guide to Big-O Notation. Stanford University. Available at: https://web.stanford.edu/class/cs106b/resources/bigo_guide.html (Accessed: 16 August 2025).

Appendix A:

Listing A1: Full Python source code for Bubble Sort performance analysis

The following code implements the Bubble Sort algorithm in Python and includes performance measurements using datasets of increasing size.

```
import random
import time


# Create lists of random integers : 10, 50, 100, 1000 and 10.000 for testing and
comparison
test_list10 = [random.randint (1, 10*10) for _ in range(10)]  # _ act as a
placeholder for the loop variable, we are not interested in indexing
test_list50 = [random.randint (1, 10*50) for _ in range(50)]
test_list100 = [random.randint (1, 10*100) for _ in range(100)]
test_list1000 = [random.randint (1, 10*1000) for _ in range(1000)]
#test_list10000 = [random.randint (1, 10*10000) for _ in range(10000)]



# Logic for Bubble Sort, swaps adjacent elements if they are in the wrong order
def bubble_sort(sorted_list):
    n = len(sorted_list)
    steps = 0  # Counts how many steps the algorithm takes
    for _ in range(n):
        swapped = False # no elements have been swapped yet
        for i in range(n - 1):# The second-to-last element, compare with (i+1)
            steps += 1 # increment the step counter
            if sorted_list[i] > sorted_list[i + 1]:
               # This is the traditional way to swap two values
                temp = sorted_list[i]
                sorted_list[i] = sorted_list[i + 1]
                sorted_list[i + 1] = temp
                swapped = True #at least one element has been swapped
        if not swapped:
            break
    return sorted_list, steps
```

```
print("Unsorted lists:", test_list10)
sorted_list, steps = bubble_sort(test_list10[:]) # [:] = creates a shallow copy of
the list, to avoid modifying the original - important!  got wrong step count without
                                                  # function returns (sorted_list,
steps)
print("Sorted lists10:", sorted_list, "| Steps:", steps)
print()
print("Unsorted lists:", test_list50)
sorted_list, steps = bubble_sort(test_list50[:])  # function returns (sorted_list,
steps)
print("Sorted lists50:", sorted_list, "| Steps:", steps)
print()
print("Unsorted lists:", test_list100)
sorted_list, steps = bubble_sort(test_list100[:])  # function returns (sorted_list,
steps)
print("Sorted lists100:", sorted_list, "| Steps:", steps)
print()
print("Unsorted lists:", test_list1000[:10]) # Small sample of the first 10 elements
to check that the unsorted list is being generated
sorted_list, steps = bubble_sort(test_list1000[:])  # function returns (sorted_list,
steps)
print("Sorted lists1000:", sorted_list[:10], "| Steps:", steps)
print()
#print("Unsorted lists:", test_list10000[:10]) #Small sample of the first 10
elements to check that the unsorted list is being generated
#sorted_list, steps = bubble_sort(test_list10000)
# function returns (sorted_list, steps)
#print("Sorted lists10000:", sorted_list[:10], "| Steps:", steps)

# The list with 10,000 elements has been removed while Bubble Sort has a Big(O) n^2
and is too slow for this demonstration...


# List10
#start = time.time()
#sorted_list, steps = bubble_sort(test_list10[:])
#t10 = time.time() - start
#print(f"List10  | Steps = {steps} | Time = {t10:.4f} s")

"""
Gave result: List10,50,100 = 0.0000 s, list1000 = 0.0020 s -> reason for using
time.perf_counter() instead of time.time()
Direct quote from Python documentation:
"a clock with the highest available resolution to measure a short duration."
Reference: https://docs.python.org/3/library/time.html#time.perf_counter
"""


# List10
start = time.perf_counter()
sorted_list, steps = bubble_sort(test_list10[:])
t10 = time.perf_counter() - start
print(f"List10  | Steps = {steps} | Time = {t10:.6f} s")  # increased {t10:.4f} s")
to 6 decimals


# List50
start = time.perf_counter()
```

```python
sorted_list, steps = bubble_sort(test_list50[:])
t50 = time.perf_counter() - start
print(f"List50  | Steps = {steps} | Time = {t50:.6f} s")

# List100
start = time.perf_counter()
sorted_list, steps = bubble_sort(test_list100[:])
t100 = time.perf_counter() - start
print(f"List100 | Steps = {steps} | Time = {t100:.6f} s")

# List1000
start = time.perf_counter()
sorted_list, steps = bubble_sort(test_list1000[:])
t1000 = time.perf_counter() - start
print(f"List1000| Steps = {steps} | Time = {t1000:.6f} s")
```

Appendix B:

Figure B1: Raw console output from Bubble Sort Performance Experiment.

The figure shows the output generated by the Python implementation. It provides the number of steps and run time for different dataset sizes. These values correspond to Table 1.

```
Unsorted lists: [98, 35, 2, 82, 52, 14, 89, 65, 38, 69]
Sorted lists10: [2, 14, 35, 38, 52, 65, 69, 82, 89, 98] | Steps: 54

Unsorted lists: [460, 213, 184, 145, 255, 80, 499, 75, 85, 424, 442, 125, 466, 305, 490, 238, 164, 384, 457, 362, 485, 3
91, 253, 195, 274, 227, 202, 411, 284, 442, 71, 17, 322, 197, 257, 449, 39, 380, 53, 340, 362, 110, 185, 298, 214, 263,
224, 251, 84, 370]
Sorted lists50: [17, 39, 53, 71, 75, 80, 84, 85, 110, 125, 145, 164, 184, 185, 195, 197, 202, 213, 214, 224, 227, 238, 2
51, 253, 255, 257, 263, 274, 284, 298, 305, 322, 340, 362, 362, 370, 380, 384, 391, 411, 424, 442, 442, 449, 457, 460, 4
66, 485, 490, 499] | Steps: 2107

Unsorted lists: [569, 259, 878, 63, 118, 550, 209, 529, 902, 231, 718, 235, 990, 638, 479, 609, 705, 76, 179, 257, 610,
587, 787, 562, 625, 38, 903, 556, 190, 787, 88, 755, 783, 651, 266, 465, 905, 829, 629, 707, 85, 606, 400, 202, 723, 756
, 162, 446, 332, 168, 774, 506, 813, 55, 421, 41, 440, 824, 415, 48, 39, 274, 423, 130, 427, 653, 791, 585, 506, 666, 78
4, 512, 895, 116, 850, 982, 708, 374, 233, 327, 560, 62, 540, 204, 800, 625, 1000, 207, 264, 882, 835, 220, 346, 49, 403
, 427, 932, 154, 171, 343]
Sorted lists100: [38, 39, 41, 48, 49, 55, 62, 63, 76, 85, 88, 116, 118, 130, 154, 162, 168, 171, 179, 190, 202, 204, 207
, 209, 220, 231, 233, 235, 257, 259, 264, 266, 274, 327, 332, 343, 346, 374, 400, 403, 415, 421, 423, 427, 427, 440, 446
, 465, 479, 506, 506, 512, 529, 540, 550, 556, 560, 562, 569, 585, 587, 606, 609, 610, 625, 625, 629, 638, 651, 653, 666
, 705, 707, 708, 718, 723, 755, 756, 774, 783, 784, 787, 787, 791, 800, 813, 824, 829, 835, 850, 878, 882, 895, 902, 903
, 905, 932, 982, 990, 1000] | Steps: 8910

Unsorted lists: [254, 107, 3789, 2293, 8759, 1059, 2560, 9885, 5075, 6235]
Sorted lists1000: [15, 26, 35, 36, 47, 63, 65, 73, 74, 77] | Steps: 926073

List10   | Steps = 54 | Time = 0.000103 s
List50   | Steps = 2107 | Time = 0.002469 s
List100  | Steps = 8910 | Time = 0.009684 s
List1000| Steps = 926073 | Time = 1.113802 s
```