International Journal for Multidisciplinary Research (IJFMR)
E-ISSN: 2582-2160 ● Website: www.ijfmr.com ● Email: editor@ijfmr.com

Design and Implementation of SortLab: A Computer Program for Comparative Analysis of Classical Sorting Algorithms

Bade Jyothi

Asst. Professor, Department of Computer Science, DRK College of Science & Technology, Bowrampet, Hyderabad, Telangana, India

Abstract

This paper presents SortLab, a single command-line computer program written in Python for the study of classical sorting algorithms in a computer science laboratory. The program implements Bubble Sort, Insertion Sort, Merge Sort, Quick Sort and Heap Sort on the same input array, checks that each output is correctly ordered, and records wall-clock time, comparison count and data movement count. Arrays of size 200, 500, 1000 and 2000 were generated from a fixed random seed so that the trial can be repeated. Measured times confirm the expected gap between quadratic methods and O(n log n) methods. At n = 1000, Bubble Sort required 43.910 ms and 495759 comparisons, while Quick Sort required 1.149 ms and 10693 comparisons on the same data. The paper describes the problem, the program structure, the algorithms, the test method, the results and the classroom use of the program. SortLab is offered as a compact teaching artefact rather than as a replacement for library sort routines.

Keywords: Sorting Algorithms, Computer Program, Algorithm Analysis, Python, Quick Sort, Merge Sort, Computer Science Education

1. Introduction

Sorting is the arrangement of a sequence into a defined order. It is one of the first non-trivial problems taught in an undergraduate computer science course because it combines a clear specification, several contrasting algorithms, and an immediate link to time complexity. Textbooks present the algorithms as separate listings. Students often run only one listing on one small array and therefore do not see how cost grows with input size.

The present work reports one computer program, named SortLab, that places five standard algorithms under a common driver. The driver builds an integer array, copies it for each algorithm, runs the algorithm, verifies the result against the language library sort, and prints a compact report. The scientific claim of the paper is modest and testable: when the five methods are given identical random integer arrays, the measured comparison counts and running times follow the order predicted by standard complexity analysis.

The program is intentionally small. It uses only the Python standard library. No graphical interface is required. The same file can be shown on a projector, modified by a student, and executed in a laboratory period. The contribution is therefore not a new sorting method. The contribution is a complete,

documented program together with repeatable measurements that a computer science class can reproduce.

2. Related Research Work

The algorithms implemented here are classical. Knuth gives a detailed treatment of insertion methods, merging, and heap construction. Hoare introduced Quick Sort and analysed its average behaviour. Williams introduced Heap Sort. Cormen, Leiserson, Rivest and Stein present asymptotic bounds and proofs that are now standard in undergraduate courses.

Empirical comparison of sorting methods is also an old laboratory exercise. Sedgewick discussed practical improvements to Quick Sort. More recent teaching papers use visualisers and automated judges. Those tools are valuable, yet they often hide the counting of comparisons. SortLab keeps the counters inside the algorithm functions so that a student can read the exact line that increments the count.

Python is used because it is widely taught and because its list type makes the data movement visible. Library routines such as list.sort are faster than the teaching implementations. They are used in SortLab only as an oracle for correctness, not as a competitor in the timing table.

3. Problem Definition

Input. A list A of n integers and a choice of algorithm.

Output. A list B that contains the same integers as A and satisfies B[i] <= B[i+1] for every valid index i. In addition the program must report the number of key comparisons, the number of recorded data movements, and the elapsed time in milliseconds.

Constraints adopted for the reported experiments. n belongs to the set {200, 500, 1000, 2000}. Each element is drawn uniformly from 0 to 1000000 inclusive. The random generator is seeded with the integer 26 so that another machine can rebuild the same arrays. Bubble Sort is not run at n = 2000 in the reported table because its quadratic cost is already clear at n = 1000 and would dominate laboratory time.

4. Program Design

4.1 Architecture

SortLab is one source file organised as six functions and a main block. Five functions implement the algorithms. Each algorithm function receives a list, works on a local copy, and returns a triple: the sorted list, the comparison count and the movement count. The sixth function, named run_trial, builds the input, calls every algorithm, checks equality with the built-in sorted function, and prints one comma-separated line per algorithm. This layout keeps side effects out of the algorithms and makes unit testing straightforward.

4.2 Data Representation

A Python list of integers is used. The representation is not the fastest possible, but it matches the mental model used in first-year courses. Merge Sort allocates one temporary list of length n. Quick Sort and Heap Sort work in place on the copied list. Bubble Sort and Insertion Sort also work in place on the copied list.

4.3 Correctness Check

After each algorithm returns, the driver compares the result with sorted(base). If the two lists differ, the

program stops with an assertion failure. This check does not prove the algorithm for all inputs. It does prevent a silent error on the arrays that are actually measured.

4.4 Counting Rules

A comparison is counted when two keys are examined to decide order. A movement is counted when an element is assigned to a new index as part of a swap or a shift. Different authors count swaps rather than assignments. The rule used here is stated so that another implementation can be compared fairly. Wall-clock time is measured with time.perf_counter around a single call. Times are therefore indicative of relative cost on one interpreter and one machine. They are not portable benchmarks.

5. Algorithms Implemented in the Program

5.1 Bubble Sort

Adjacent pairs are compared and swapped when they are out of order. One pass places the largest remaining key at the end. An early-exit flag stops the process when a pass makes no swap. Worst-case time is O(n squared). Best case after the flag is added is O(n). The inner loop of the program is the following.

```
for i in range(n):
swapped = False
for j in range(0, n - i - 1):
comparisons += 1
if a[j] > a[j + 1]:
a[j], a[j + 1] = a[j + 1], a[j]
moves += 1
swapped = True
if not swapped:
break
```

5.2 Insertion Sort

The prefix A[0..i-1] is kept sorted. The key at index i is inserted into that prefix by shifting larger elements one place to the right. Worst-case time is O(n squared). The method is efficient on nearly sorted data. In the random trials reported below it remains far slower than the logarithmic methods.

5.3 Merge Sort

The list is split into two halves. Each half is sorted recursively. The two sorted halves are merged in linear time using a temporary buffer. Recurrence T(n) = 2T(n/2) + O(n) gives O(n log n) in every case. Extra memory of order n is the price paid for the stable bound.

5.4 Quick Sort

A pivot is chosen. Keys not larger than the pivot are moved to its left. Keys larger than the pivot are moved to its right. The two sides are then sorted recursively. The implementation uses the last element as pivot (Lomuto partition). Average time is O(n log n). Worst-case time is O(n squared) on already sorted input with this pivot rule. The random arrays used in the trials avoid that degenerate case. A production program would use median-of-three or a random pivot. Those improvements were omitted so that the textbook algorithm remains visible.

5.5 Heap Sort

The array is first turned into a max-heap in linear time. The largest key is then swapped with the last unsorted position and the heap property is restored. Time is O(n log n) in the worst case and extra

memory is O(1) aside from the recursion used in heapify. An iterative heapify would remove that recursion; the recursive form is kept for readability.

Table 1: Algorithms Encoded in SortLab

Algorithm	Usual Time Bound	Extra Memory	Stable
Bubble Sort	O(n squared) worst	O(1)	Yes
Insertion Sort	O(n squared) worst	O(1)	Yes
Merge Sort	O(n log n) always	O(n)	Yes
Quick Sort	O(n log n) average	O(log n) stack	No
Heap Sort	O(n log n) always	O(1)	No

6. Research Method

The program was executed once for each pair (n, algorithm) listed in Table 2. The host interpreter was CPython 3. The same base array was given to every algorithm at a given n. Seed 26 was fixed before the first array was built. Correctness was required for every completed run. Bubble Sort at n = 2000 was not executed in the timed series.

Independent variables. Algorithm name and array length.

Dependent variables. Elapsed time in milliseconds, number of key comparisons, and number of recorded movements.

Controlled factors. Identical input at each n, single-threaded execution, and a frozen seed. Cache effects and interpreter start-up are not isolated. For that reason the paper treats time as an ordinal indicator and treats comparison count as the more portable measure.

7. Results

Table 2 reports the measurements obtained from SortLab. Two patterns are visible. First, comparison counts for Bubble Sort and Insertion Sort grow much faster than those for Merge Sort, Quick Sort and Heap Sort. Second, among the three O(n log n) methods on these random arrays, Quick Sort was the fastest and Heap Sort performed more comparisons than Merge Sort.

Table 2: SortLab Measurements on Random Integer Arrays (Seed 26)

n	Algorithm	Time (ms)	Comparisons	Moves
200	Bubble Sort	1.708	19885	10232
200	Insertion Sort	0.684	10426	10232
200	Merge Sort	0.215	1277	674
200	Quick Sort	0.167	1617	930
200	Heap Sort	0.277	2451	1356
500	Bubble Sort	9.699	124497	59681
500	Insertion Sort	4.459	60174	59681
500	Merge Sort	0.772	3846	1962

500	Quick Sort	0.767	5091	3390
500	Heap Sort	0.857	7451	4075
1000	Bubble Sort	43.910	495759	236524
1000	Insertion Sort	20.112	237520	236524
1000	Merge Sort	1.534	8720	4352
1000	Quick Sort	1.149	10693	6312
1000	Heap Sort	1.901	16862	9120
2000	Insertion Sort	90.915	1011845	1009851
2000	Merge Sort	3.590	19413	9772
2000	Quick Sort	2.624	25130	12773
2000	Heap Sort	4.431	37792	20220

At n = 1000 the comparison ratio of Bubble Sort to Merge Sort is 495759 divided by 8720, which is about 56.9. The time ratio on the same pair is 43.910 divided by 1.534, which is about 28.6. The two ratios differ because time includes Python loop overhead, not only comparisons. Both ratios still separate the quadratic family from the logarithmic family.

Insertion Sort used fewer comparisons than Bubble Sort at every n, as expected, because it does not keep scanning a suffix that is already finished. Heap Sort used more comparisons than Merge Sort. That result is consistent with the known extra work of maintaining the heap property. Quick Sort used more comparisons than Merge Sort on these arrays yet less time, which is consistent with better locality and in-place partition in CPython.

When n grows from 1000 to 2000, Insertion Sort time grows from 20.112 ms to 90.915 ms, a factor of about 4.5, close to the factor of four predicted by a quadratic model. Merge Sort time grows from 1.534 ms to 3.590 ms, a factor of about 2.3, close to the factor expected from an n log n model over a doubling of n.

8. Discussion

The program does what a first course needs. It makes complexity visible as a table instead of as a symbol on a slide. A student who changes the pivot rule, adds a nearly-sorted array, or prints the array after each Bubble Sort pass can see the effect without leaving the same file.

Limits must be stated. The implementations are teaching versions. They are not tuned. Recursion depth in Quick Sort can exceed the default Python limit on adversarial input. Integer keys only were tested. Stability was not measured experimentally, although Merge Sort and the two quadratic methods are written so as to preserve equal-key order. Times will change on another machine. Comparison counts should not.

The program is not a substitute for a proof. O(n log n) is not established by four data points. The measurements only illustrate the proofs that the student is expected to read in parallel.

9. How the Program Is Used in a Computer Science Class

A ninety-minute laboratory can follow four steps. First, students run SortLab unchanged and copy Table

2. Second, they add a sixth method, for example Selection Sort, and rebuild the table. Third, they feed an already sorted array into Quick Sort with last-element pivot and observe the collapse toward quadratic behaviour. Fourth, they write four sentences that relate the new numbers to the recurrences in the lecture. Assessment can mark the table, the extra function, and the written explanation. Copying of output is easy to detect if each student is given a personal seed.

10. Conclusion

SortLab is one computer program. It implements five classical sorting algorithms, checks their output, and records time and operation counts. Trials on random integer arrays of length 200 to 2000 produced counts and times that follow the familiar separation between O(n squared) methods and O(n log n) methods. Quick Sort was the fastest method on the random data. Merge Sort used the fewest comparisons among the logarithmic methods. Insertion Sort, though faster than Bubble Sort, remained impractical at n = 2000.

The program is short enough to read in one sitting and complete enough to support a laboratory class in computer science. Future work may add a random-pivot Quick Sort, an iterative heapify, and a second input family of nearly sorted arrays. Those extensions would still belong to the same single program.

References

1. Knuth D. E. The Art of Computer Programming, Volume 3: Sorting and Searching. 2nd ed. Addison-Wesley; 1998.
2. Hoare C. A. R. Quicksort. The Computer Journal. 1962;5(1):10-16.
3. Williams J. W. J. Algorithm 232: Heapsort. Communications of the ACM. 1964;7(6):347-348.
4. Cormen T. H., Leiserson C. E., Rivest R. L., Stein C. Introduction to Algorithms. 3rd ed. MIT Press; 2009.
5. Sedgewick R. Implementing Quicksort Programs. Communications of the ACM. 1978;21(10):847-857.
6. Sedgewick R., Wayne K. Algorithms. 4th ed. Addison-Wesley; 2011.
7. Goodrich M. T., Tamassia R., Goldwasser M. H. Data Structures and Algorithms in Python. Wiley; 2013.
8. Wirth N. Algorithms + Data Structures = Programs. Prentice Hall; 1976.
9. Bentley J. Programming Pearls. 2nd ed. Addison-Wesley; 2000.
10. Python Software Foundation. The Python Language Reference. https://docs.python.org/3/reference/
11. Robins A., Rountree J., Rountree N. Learning and Teaching Programming: A Review and Discussion. Computer Science Education. 2003;13(2):137-172.
12. Luxton-Reilly A., et al. Introductory Programming: A Systematic Literature Review. ITiCSE-WGR 2018. p. 55-106.
13. McConnell S. Code Complete. 2nd ed. Microsoft Press; 2004.
14. IEEE Computer Society. Guide to the Software Engineering Body of Knowledge (SWEBOK). IEEE; 2014.