METHODOLOGY FOR TEACHING THE QUICKSORT ALGORITHM FOR ARRAYS IN THE PYTHON PROGRAMMING LANGUAGE

Fathiddinov Saidma'ruf Lazizxo'ja o'g'li
Teacher, Department of Exact Sciences and
Digital Technologies, Asia International University

Abstract. The article proposes a methodology for teaching the quicksort algorithm for arrays (lists) in the course "Fundamentals of the Python Programming Language". Quicksort combines three ideas that are difficult for novices – recursion, the divide-and-conquer strategy and in-place partitioning – so the methodology introduces them gradually. It is based on the PRIMM approach (Predict – Run – Investigate – Modify – Make) and consists of five stages: modelling recursion by hand with number cards, completing a partition trace table and a recursion tree, coding in two steps (first a short list-based version, then an in-place version with Lomuto partitioning), visualising the call stack and testing automatically, and analysing the average and worst-case complexity of the algorithm experimentally. The article presents reference Python implementations, an execution trace, the results of a computational experiment on the number of comparisons, typical student errors, a comparison with other sorting methods and a plan for a pedagogical experiment.

Keywords: Python, array, list, quicksort, recursion, divide and conquer, partitioning, algorithm complexity, PRIMM, programming teaching methodology.

INTRODUCTION

Sorting algorithms are one of the central topics in learning to program: through them, students master working with arrays, loops, indices and conditions, learn to evaluate algorithmic efficiency and develop computational thinking [3, 4, 7]. Among them, quicksort, proposed by C.A.R. Hoare [2], holds a special place: it is one of the fastest comparison-based sorting algorithms in practice, and its ideas underlie the sorting functions of many standard libraries [5, 6]. Given that the "Digital Uzbekistan – 2030" strategy [1] sets the development of young people's programming and digital skills as a priority, designing an effective methodology for teaching such fundamental algorithms is a pressing task.

However, quicksort is considerably harder for students than simple quadratic methods such as selection or insertion sort. It requires an understanding of recursion, which is known to be one of the most difficult topics for novices: Kahney showed that many beginners hold an incorrect "looping" model of recursion instead of the correct "copies" model, in which each call works with its own data [8]. In addition, students must understand the partitioning procedure, in which several indices move through the array at the same time, and the dependence of the running time on the choice of the pivot element. Lister and colleagues found that a considerable share of novice programmers struggle even to trace simple code and predict its output [14], and Sorva explains such difficulties by the absence of a correct mental model of program execution – a "notional machine" [13]. Finally, because Python provides the built-in sorted() function and the list.sort() method, students often consider studying sorting algorithms unnecessary.

Several approaches to these problems are known: computer-free (*unplugged*) activities help students grasp the idea of an algorithm before seeing code [10]; Parsons problems reduce syntactic load [11]; visualisation of program execution makes the call stack visible [12]; and the PRIMM approach suggests reading code and predicting its output before writing it [9]. *The aim of the study* is to combine these approaches into a coherent methodology for teaching quicksort and to justify a system for evaluating its effectiveness. *Working hypothesis:* if the physical modelling of recursion, the trace table,

© 2026 London International Monthly Conference on Multidisciplinary Research and Innovation. All Rights Reserved.

two-step coding and an experimental complexity analysis are consistently linked, students' understanding of recursion and of the algorithm, as well as their code-tracing skills, will be higher than under the traditional "explanation – sample code – exercises" sequence.

MAIN PART

The essence of the algorithm. Quicksort follows the divide-and-conquer strategy [2, 4]. One element of the array, called the *pivot*, is selected; the array is rearranged so that all elements less than or equal to the pivot are to its left and all greater elements are to its right (the *partition* step); after this, the pivot is in its final position, and the left and right parts are sorted recursively in the same way. An array of zero or one element is already sorted and forms the base case of the recursion. To separate the idea of the algorithm from the technical details of partitioning, it is first presented in a short list-based form (Listing 1).

Listing 1. List-based version of quicksort

```
def qsort(a):
    """Returns a new sorted list (the divide-and-conquer idea)."""
    if len(a) <= 1:                    # base case
        return a
    pivot = a[len(a) // 2]
    less = [x for x in a if x < pivot]
    equal = [x for x in a if x == pivot]
    greater = [x for x in a if x > pivot]
    return qsort(less) + equal + qsort(greater)
```

This version clearly shows the recursive structure of the algorithm but creates new lists at each level and therefore requires additional memory. In the second step, students move to the classic in-place version, in which the array is rearranged by exchanging elements. Listing 2 uses the Lomuto partitioning scheme, in which the last element of the subarray serves as the pivot [4].

Listing 2. In-place quicksort with Lomuto partitioning

```
def partition(a, lo, hi):
    """Lomuto partition: puts pivot a[hi] in its final place."""
    pivot = a[hi]
    i = lo                          # end of the "<= pivot" zone
    for j in range(lo, hi):
        if a[j] <= pivot:
            a[i], a[j] = a[j], a[i]
            i += 1
    a[i], a[hi] = a[hi], a[i]          # move the pivot into place
    return i

def quick_sort(a, lo=0, hi=None):
    if hi is None:
        hi = len(a) - 1
    if lo < hi:                     # base case: 0 or 1 element
        p = partition(a, lo, hi)
        quick_sort(a, lo, p - 1)       # left part
        quick_sort(a, p + 1, hi)       # right part
    return a

numbers = [7, 2, 9, 4, 1, 8, 3, 5]
print(quick_sort(numbers))             # [1, 2, 3, 4, 5, 7, 8, 9]
```


© 2026 London International Monthly Conference on Multidisciplinary Research and Innovation. All Rights Reserved.

Table 1 shows all calls of the partition function when the array [7, 2, 9, 4, 1, 8, 3, 5] is sorted (the pivot is shown in bold in its final position). This table, together with the recursion tree, is completed with students in class and serves as the main didactic tool of the methodology.

Table 1. Trace of partition calls for the array [7, 2, 9, 4, 1, 8, 3, 5]

Call (lo, hi)	Subarray before	Pivot	Subarray after	Pivot index	Comparisons
(0, 7)	[7, 2, 9, 4, 1, 8, 3, 5]	5	[2, 4, 1, 3, **5**, 8, 7, 9]	4	7
(0, 3)	[2, 4, 1, 3]	3	[2, 1, **3**, 4]	2	3
(0, 1)	[2, 1]	1	[**1**, 2]	0	1
(5, 7)	[8, 7, 9]	9	[8, 7, **9**]	7	2
(5, 6)	[8, 7]	7	[**7**, 8]	5	1
Total			[1, 2, 3, 4, 5, 7, 8, 9]		**14**

For comparison, selection sort would perform $n(n-1)/2 = 28$ comparisons on the same array of eight elements, whereas quicksort needed only 14. Students obtain this figure themselves, which naturally motivates the question of why quicksort is faster and in which cases this advantage disappears.

Stages of the methodology. The methodology is based on the PRIMM approach [9], is designed for two practical sessions plus homework, and consists of five stages: (1) *modelling recursion by hand* – one student (the "manager") receives a stack of 8–10 number cards, takes the last card as the pivot, splits the remaining cards into a "less than or equal" pile and a "greater" pile, passes each pile to another pair of students with the same instruction and waits for the sorted piles to be returned; a pair that receives zero or one card returns it immediately; in this way each "call" physically works with its own data, which forms the correct "copies" model of recursion [8, 10]; (2) *trace table, recursion tree and prediction* – without running the code, students complete a table similar to Table 1 for a new array, draw the recursion tree and predict the number of comparisons; (3) *two-step coding* – first the list-based version (Listing 1) is written, then the partition function is written and tested separately on small arrays, and only after that is it used in quick_sort (Listing 2); students with less programming experience assemble partition from shuffled lines of code (a Parsons problem) [11]; (4) *visualisation and testing* – the code is executed step by step in Python Tutor [12], where students observe how call frames are created and removed on the stack and compare this with the prediction made in stage (2); both versions are then checked automatically against sorted() on random arrays with many repeated values (Listing 3); (5) *analysis, modification and creation* – students count comparisons experimentally (Table 2), investigate the worst case, improve the choice of the pivot and compare quicksort with other methods (Table 3).

Listing 3. Automatic testing of both versions

```
import random

for _ in range(1000):
    n = random.randint(0, 30)
    m = [random.randint(-20, 20) for _ in range(n)]
    assert qsort(m) == sorted(m), m
    assert quick_sort(m.copy()) == sorted(m), m
print("All tests passed")
```


© 2026 London International Monthly Conference on Multidisciplinary Research and Innovation. All Rights Reserved.

Complexity analysis. In each call, partition performs hi − lo comparisons, i.e. one fewer than the length of the subarray. If the pivot divides the array roughly in half each time, the depth of recursion is about $\log_2 n$ and the total work is O(n log n); the average number of comparisons for random arrays is $2(n + 1)H(n) − 4n \approx 1.39\, n \log_2 n$, where $H(n) = 1 + 1/2 + \ldots + 1/n$ [3, 5]. If, however, the pivot is always the smallest or largest element – which happens with Lomuto partitioning on an already sorted array – each call removes only one element, the depth of recursion becomes n − 1 and the number of comparisons reaches $n(n − 1)/2$, i.e. $O(n^2)$. Students verify these conclusions by adding a comparison counter to partition. Table 2 shows the results of such a computational experiment (random arrays, mean of 100 runs).

Table 2. Number of comparisons performed by quicksort (Listing 2)

n	Random arrays (measured mean)	Theory: 2(n + 1)H(n) − 4n	Sorted array: n(n − 1)/2
100	637	648	4 950
1 000	11 012	10 986	499 500
10 000	155 979	155 772	49 995 000

The measured values practically coincide with the theoretical formula, while on a sorted array the number of comparisons grows quadratically – for n = 10 000 it is more than 300 times higher. Moreover, with Python's default recursion limit of 1000 calls, the program in Listing 2 terminates with a RecursionError on a sorted array already at $n \approx 1000$, so the values in the last column for large n are obtained from the formula. This "failure" becomes a valuable learning situation: students discover for themselves that the choice of the pivot matters and try two remedies – choosing a random pivot (swapping a randomly selected element with a[hi] before partitioning) and the median-of-three rule [5, 6] – and also learn that making the first recursive call on the smaller part limits the depth of recursion to O(log n) [5].

Stability. The in-place version of quicksort is not stable. For example, when the list [(1, "a"), (1, "b"), (0, "c")] is sorted by the first element only, the first partition swaps (1, "a") and (0, "c"), and the result is [(0, "c"), (1, "b"), (1, "a")]. Interestingly, the list-based version from Listing 1 is stable when comparing by key, since list comprehensions preserve the original order of elements; this contrast clearly shows students the cost of saving memory. The main properties of the sorting methods are summarised in Table 3.

Table 3. Comparative characteristics of sorting methods

Method	Best case	Average case	Worst case	Extra memory	Stability
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)	not stable
Insertion sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)	stable
Merge sort	O(n log n)	O(n log n)	O(n log n)	O(n)	stable
Quicksort (Listing 2)	O(n log n)	O(n log n)	$O(n^2)$	O(log n) on average, O(n) at worst (stack)	not stable
sorted() (Timsort)	O(n)	O(n log n)	O(n log n)	O(n)	stable

© 2026 London International Monthly Conference on Multidisciplinary Research and Innovation. All Rights Reserved.

Common errors. In teaching practice, the following errors frequently occur with this topic: (a) the base case is missing or incorrect, which leads to infinite recursion and a RecursionError; (b) the pivot is included in a recursive call, for example quick_sort(a, lo, p) instead of quick_sort(a, lo, p - 1) – when the pivot is the largest element, the same subarray is processed again and the recursion never ends; (c) in the list-based version, the condition x <= pivot is used for the less list – for an array such as [1, 2] all elements fall into less, and the recursion again becomes infinite; (d) the equal list is omitted, and all elements equal to the pivot disappear from the result; (e) the result of qsort(a) is not assigned, and the student is surprised that the list has not changed – this is a good occasion to discuss the difference between functions that return a new list and functions that modify a list in place; (f) hi = len(a) is written instead of len(a) - 1, which causes an IndexError. For each error a "find the bug" task is prepared: the student receives faulty code and, using the trace table and the recursion tree, determines at which call the algorithm goes wrong.

Assessment and pedagogical experiment plan. The experiment is planned within the course "Fundamentals of the Python Programming Language" with an experimental group and a control group. In the control group the topic is taught in the traditional way, while in the experimental group it is taught using the proposed five-stage methodology. Both groups take the same test before and after the sessions; it consists of tasks on tracing recursive code, predicting the result of partitioning, drawing a recursion tree, finding bugs and estimating the number of comparisons. The results are compared using the normalised gain g = (post − pre)/(100 − pre) [15] and Student's t-test. Practical assignments are assessed for correctness (automatic tests), code quality and the quality of the student's explanation during an oral defence.

Expected results. The methodology is expected to produce: (1) a more accurate "copies" model of recursion, since students physically act out each call with its own data [8]; (2) improved skills in tracing code and predicting its output [9, 14]; (3) an understanding of average-case and worst-case complexity through students' own measurements rather than through a ready-made formula; (4) an informed appreciation of why library functions such as sorted() use carefully engineered algorithms. These results are hypotheses, and confirming them with empirical data is the task of the pedagogical experiment.

Limitations and their mitigation. First, quicksort requires more time than simple sorting methods, so stages (1)–(2) are assigned to the first session and stages (3)–(5) to the second session and homework; it is also advisable to study the topic after students have completed an introductory recursion topic (factorial, binary search). Second, since artificial intelligence tools can easily produce ready-made code, assessment focuses not on the code itself but on the trace table, the recursion tree, "find the bug" tasks and the oral defence. Third, because students' programming levels differ, task difficulty is differentiated through Parsons problems and pair work.

CONCLUSION

The proposed methodology teaches quicksort not as code to be memorised but as an algorithm that students act out, trace, program in two steps, test and analyse experimentally. Physical modelling with cards forms a correct model of recursion, the trace table and the recursion tree link this model to the Python code, and the computational experiment – including the deliberately provoked RecursionError on a sorted array – shows the difference between the average and the worst case in a way that students remember. The effectiveness of the methodology will be determined through a pedagogical experiment comparing experimental and control groups, using tests and assessment criteria. In the next stage, similar sets of tasks and a teaching guide are planned for merge sort and other recursive algorithms.

© 2026 London International Monthly Conference on Multidisciplinary Research and Innovation. All Rights Reserved.

REFERENCES

1. Decree of the President of the Republic of Uzbekistan No. PF-6079 of 5 October 2020 “On approval of the Digital Uzbekistan – 2030 strategy and measures for its effective implementation”.
2. Hoare C.A.R. Quicksort // The Computer Journal. – 1962. – Vol. 5, No. 1. – P. 10–16.
3. Knuth D.E. The Art of Computer Programming. Vol. 3: Sorting and Searching. 2nd ed. – Reading, MA: Addison-Wesley, 1998. – 780 p.
4. Cormen T.H., Leiserson C.E., Rivest R.L., Stein C. Introduction to Algorithms. 4th ed. – Cambridge, MA: MIT Press, 2022. – 1312 p.
5. Sedgewick R. Implementing Quicksort programs // Communications of the ACM. – 1978. – Vol. 21, No. 10. – P. 847–857.
6. Bentley J.L., McIlroy M.D. Engineering a sort function // Software: Practice and Experience. – 1993. – Vol. 23, No. 11. – P. 1249–1265.
7. Wing J.M. Computational thinking // Communications of the ACM. – 2006. – Vol. 49, No. 3. – P. 33–35.
8. Kahney H. What do novice programmers know about recursion? // Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI ’83). – 1983. – P. 235–239.
9. Sentance S., Waite J., Kallia M. Teaching computer programming with PRIMM: a sociocultural perspective // Computer Science Education. – 2019. – Vol. 29, No. 2–3. – P. 136–176.
10. Bell T., Witten I.H., Fellows M. CS Unplugged: An enrichment and extension programme for primary-aged students. – Christchurch: University of Canterbury, 2015.
11. Parsons D., Haden P. Parson’s programming puzzles: a fun and effective learning tool for first programming courses // Proceedings of the 8th Australasian Conference on Computing Education. – 2006. – Vol. 52. – P. 157–163.
12. Guo P.J. Online Python Tutor: embeddable web-based program visualization for CS education // Proceedings of the 44th ACM Technical Symposium on Computer Science Education (SIGCSE ’13). – 2013. – P. 579–584.
13. Sorva J. Notional machines and introductory programming education // ACM Transactions on Computing Education. – 2013. – Vol. 13, No. 2. – Article 8.
14. Lister R., Adams E.S., Fitzgerald S., et al. A multi-national study of reading and tracing skills in novice programmers // ACM SIGCSE Bulletin. – 2004. – Vol. 36, No. 4. – P. 119–150.
15. Hake R.R. Interactive-engagement versus traditional methods: A six-thousand-student survey of mechanics test data for introductory physics courses // American Journal of Physics. – 1998. – Vol. 66, No. 1. – P. 64–74.

© 2026 London International Monthly Conference on Multidisciplinary Research and Innovation. All Rights Reserved.