Tutorial 8: Sorting Algorithms

The Art of Computer Programming

18 November 2026

Objective. This tutorial puts into practice the ninth lecture: running quicksort and heapsort by hand; writing insertion sort, merge sort, and quicksort in Python, testing them, and counting their comparisons on random, sorted, and reversed inputs; watching quicksort degrade on sorted input and repairing it with a random pivot; timing quicksort and heapsort in C against the library; and using a heap as a priority queue. Do Parts A and B in the first half of the session and Parts C and D in the second half; Part E is for those who finish early.

Reminders

- Create a folder `tutorial08` with two files, `sorts.py` (Parts B, C, and E) and `sorts.c` (Part D).
- The card deck shown in the lecture is at `https://pierre.senellart.com/other/deck/`.
- `random.shuffle(t)` shuffles the list `t`, `random.randint(a, b)` draws an integer between a and b included, and `time.perf_counter()` gives a time in seconds. A function that assigns a global variable declares it `global` first (lecture 4).
- Python raises `RecursionError` beyond about 1 000 nested calls. This is expected in one question; do not raise the limit.
- In C, `rand()` (from `<stdlib.h>`) draws a random integer, and `clock()` (from `<time.h>`) measures time (tutorial 3). Compile with `$ gcc -std=c23 -O2 -Wall -Wextra sorts.c -o sorts`.

Part A: By hand, with cards

Q1. You are the algorithm. In the tab Run it yourself of the deck, every step is yours: select two cards, then compare or swap them. When the cards are sorted, the page reports how many comparisons and swaps you made.

(a) Run *Quicksort, pivot = last element* on the example from the slides, then on 8 cards *Already sorted*. How many comparisons does the second run take? Give a formula for n sorted cards, and say why every partition is as bad as it can be.

(b) Run *Heapsort* on the example from the slides. Which card is at the root of the tree once the heap is built, and why must it be there?

Part B: Three sorts in Python, tested

Every sort function of this part takes a list and returns the sorted list; those that sort in place (insertion sort, quicksort) sort the list they are given, then return it.

Q2. Tests first.

(a) Write `is_sorted(t)`, which returns `True` if every element of `t` is $\leq$ the next one, and `False` otherwise. Check: `[]`, `[5]`, and `[1, 2, 2, 3]` are sorted, `[2, 1]` and `[1, 3, 2]` are not.

(b) A sort that works on one example may still fail on an empty list, on duplicates, or on an input that is already in order: bugs hide in such special cases. To test the sorts of Q3 on all of them at once, write `test_lists()`, which returns a list of lists to sort: `[]`, `[5]`, `[2, 1]`, a sorted list, a reversed list, `[3, 1, 3, 2, 1]`, a list of six equal values, and 200 random lists of random lengths between 0 and 50, with values between 0 and 9 (hence many duplicates).

Q3. Three sorts.

(a) Write three functions: `insertion_sort(t)`; `merge_sort(t)`, with a function `merge(left, right)`; and `quicksort(t)`. The last one calls a recursive function `quicksort_rec(t, lo, hi)`, which takes `t[hi]` as its pivot and partitions as in the lecture. Try to write them from the pseudocode and the description of the lecture before looking at its code. Test each one: for every list of `test_lists()`, sort a copy (`list(t)`) and check the result with `is_sorted`.

(b) Introduce a mistake in each one in turn: start the loop of insertion sort at 2, forget the last line of `merge`, write `quicksort_rec(t, lo, p)` instead of `quicksort_rec(t, lo, p - 1)`. Which mistakes do your tests detect? For the one they miss, what else should they check? And one of them gives no result at all: what does it do instead, and why?

Q4. Counting comparisons. Add a global variable `comparisons` and a function `less_eq(a, b)` that adds 1 to it and returns `a <= b`. Replace every comparison between two *elements* of the list in the three sorts by a call to `less_eq` (comparisons between indices do not count; in insertion sort, `t[j - 1] > t[j]` becomes `not less_eq(t[j - 1], t[j])`). To count the comparisons of a sort on a list `t`, set `comparisons` to 0, sort a copy of `t`, then read `comparisons`. Check against Part A: quicksort on `list(range(8))` makes 28 comparisons, and so does insertion sort on 8 values in decreasing order.

Part C: Measuring

Q5. Three inputs.

(a) For $n = 500$, 1 000, 2 000, and 4 000, print a table of the number of comparisons made by the three sorts on a random list (`list(range(n))` shuffled), on a sorted one (`list(range(n))`), and on a reversed one. Quicksort fails on some of them: catch the exception (lecture 8) so that the table is printed anyway, with a dash in place of the number. Why does it fail?

(b) When n doubles, by what factor does each count grow? Compare with the table of complexities that ends the lecture: which algorithm is quadratic on which input, and which stays close to $n \log_2 n$? How many comparisons does insertion sort make on sorted input, and why is this the best possible for any algorithm?

(c) In the tab Compare them of the deck, choose the *Comparisons* measure, open the *Table of values*, and compare it with your table for the sorted and the reversed inputs (choose sizes up to 500 for quicksort). The numbers should be *exactly* the same: why not for the random input?

Q6. A random pivot. Write `quicksort_random(t)`, which, before each partition, swaps `t[hi]` with `t[r]` for `r` drawn at random between `lo` and `hi`. Add it to the table of Q5. Does it still fail on sorted input? How do its counts compare with $1.39\, n \log_2 n$, on each of the three inputs?

Q7. Time. Time `merge_sort`, `quicksort_random`, and Python's `sorted` on random lists of 100 000, 200 000, and 400 000 elements. Do the times grow as $n \log n$? By what factor is `sorted` faster than your merge sort, and why could that be, given that it is a merge sort too (lecture 1: how is Python executed)?

Part D: In C

Q8. Quicksort, heapsort, and the library. In `sorts.c`, write `quicksort` and `heapsort` as in the lecture, a function `is_sorted(t, n)`, and a function that allocates an array of n integers with `malloc` (reporting the failure if there is not enough memory), fills it with `rand()`, sorts it with one of the algorithms, prints the time taken and whether the result is sorted, and frees the array. Also sort with `qsort`, with the comparison function of the lecture.

(a) Time the three on random arrays of 10^6 and 10^7 elements. By what factor does each time grow? Which one is the fastest, and is heapsort as fast as its $\Theta(n \log n)$ promises (see the lecture on why it is not)?

(b) Now time quicksort on *sorted* arrays (`t[i] = i`) of 10 000, 20 000, and 40 000 elements. By what factor does the time grow? *Without running it*, estimate how long a sorted array of 10^6 elements would take. It would not get there: each recursive call on $n - 1$ elements adds a frame to a call stack of a few megabytes (lecture 7). What happens to a C program that overflows its stack?

Part E: If you have time: a heap as a priority queue

Python's module `heapq` keeps a heap in an ordinary list, with the array layout of the lecture, but it is a *min*-heap: the smallest element is at the root, `h[0]`. Starting from `h = []`, `heapq.heappush(h, x)` inserts `x`, `heapq.heappop(h)` removes and returns the smallest element, and `heapq.heapreplace(h, x)` does both at once (pop, then push x); each costs $O(\log n)$.

Q9. The k largest.

(a) Write `heap_sort(t)`, which pushes all the elements of `t` into a heap, then pops them all; it must pass your tests. What is its complexity?

(b) Write `largest(t, k)`, which returns the k largest elements of `t` by going through `t` once and keeping a heap of the k largest elements seen so far: when a new element is larger than the smallest of them, it replaces it. What is its complexity, compared with sorting? Compare it with `sorted(t)[-k:]` on 10^6 random values for $k = 10$.

Check

By the end of the session you should be comfortable with: writing insertion sort, merge sort, quicksort, and heapsort, and testing a sort on edge cases and random inputs; counting the operations of an algorithm to check its complexity independently of the machine; recognizing the inputs that make quicksort quadratic, and why a random pivot fixes it; and calling the library sort, which is what you should do in practice. If one of them is still unclear, ask us, or finish the corresponding exercises at home.