

CSCI B505 Fall 19: Programming assignment 2

Due online: September 27, 11:59pm EST

Submit your work online using Canvas (no assignments will be accepted on paper this time). You can use LaTeX, Word, or even pen and paper to write down your answers. But please try to submit a PDF file. Use the two lab sessions prior to the due date to get help from your AI/UI, if necessary.

What to submit

For this assignment you will be submitting two things:

1. Source code. Please, follow good coding practices: use indentation, write comments, etc.
2. Write-up. This should contain:
 - Plots demonstrating performance of your code.
 - Justifications for any choices you've made.
 - Conclusions and analysis of the results.

Insertion Sort vs. Merge Sort vs Deterministic Quicksort

Implement INSERTION SORT, MERGESORT and DETERMINISTIC QUICKSORT (as described in CLRS, Chapter 7.1). You can use one of the following programming languages: C/C++, Java or Python.

For Python users, if your code runs out of stack for some of the larger input sizes, just report it as such.

Input/Output:

Your input will be a sequence of n numbers x_1, x_2, \dots, x_n given in an input file in this order. Each number will be an integer between 1 and n . The output should be a file containing these integers sorted in non-decreasing order.

For each of the first 4 input types below you should plot the running time of each algorithm for inputs of size $n = 5000, 10000, 15000, \dots$ up to 30000. Plot all three algorithms on the same chart so that it is easy to compare. For the last input type (small random inputs) see instructions below. When measuring the running time you should only measure the time of sorting, not the time spent generating the data.

Input/Plot 1: Large random inputs. Generate each x_i to be a uniformly random integer between 1 and n . On random inputs that you generate: For each data point take the average of 3 runs (each time generating a new random input).

Input/Plot 2: Non-decreasing inputs. Generate each x_i to be a uniformly random integer between 1 and n and sort the resulting sequence in non-decreasing order ($x_1 \leq x_2 \leq \dots \leq x_n$). Then run each of the sorting algorithms again and measure its performance.

Input/Plot 3: Non-increasing inputs. Generate each x_i to be a uniformly random integer between 1 and n and sort the resulting sequence in non-increasing order ($x_1 \geq x_2 \geq \dots \geq x_n$). Then run each of the sorting algorithms again and measure its performance.

Input/Plot 4: Noisy non-decreasing inputs. Generate input in two steps:

1. Generate input as in Plot 2.
2. Repeat the following 50 times: Pick two random integers i and j and exchange x_i and x_j .

For each data point take the average of 3 runs (each time generating a new random input).

Input 5: Small random inputs. Generate 100,000 inputs as in Input/Plot 1 for $n = 50$ (i.e. you need to generate 100K sequences of random numbers between 1 and 50, of length 50 each). Measure the overall runtime of sorting these sequences. There is no plot for this type of input, just compare the two resulting runtimes.

Write-up

Explain your choices: Explain any platform/language choices that you made for your code and plots. How did you create and store the data you used to make the plots. Did you run into any difficulties or made any interesting observations?

Conclusions: The last two algorithms have asymptotic running time of $O(n \log n)$. Does the first plot reflect this? How do the three algorithms compare in terms of the running time? How about the second plot? Which algorithm do you think is the best one here? How does the third plot compare to the first and second? What about the fourth plot? What kind of functions do you think you're observing in the three plots (linear, logarithmic, quadratic, exponential, etc.)? Would you use these algorithms for real data and if so why?