CSCI B505 Spring 20: Programming assignment 1

Due date: January 31, 11:59 PM, File submission

Submit your work via Canvas, using "File upload". All work is strictly individual. If you have difficulties, please ask AI's for help during their office hours.

What to submit

For this assignment you will be submitting the following files:

1. Source code reproducing your results. Please, follow good coding practices:
   - Use proper formatting and indentation. Most IDE's have shortcuts for automatic formatting.
   - Use proper naming for functions/variables/etc.
   - Avoid code duplication.
   - Write comments. As a rule of thumb, every class and method should have a comment.
   - Etc. For Python, this guide is a good start (Google has similar guides for other languages): `http://google.github.io/styleguide/pyguide.html#3-python-style-rules`.
2. Write-up in PDF/Word format. **Handwriting is not allowed.** The file should contain:
   - Plots demonstrating performance of your code.
   - Justifications for any choices you've made.
   - Conclusions and analysis of the results.
3. If you write a code for drawing the plots in a separate file, please submit this file too.

Insertion Sort vs. Bubble Sort with check

Implement Insertion Sort (as discussed in lectures) and Bubble Sort with check (as described below) for integer arrays. **Note that this version of Bubble Sort is a bit different from the one described in a lab**. You can use one of the following programming languages: C/C++, Java or Python.

Bubble Sort with check

Bubble Sort is the following algorithm. Given an array $A[1 \dots n]$ we go through the array from left to right, and look at all pairs of adjacent elements: if $A[1] > A[2]$, then $A[1]$ and $A[2]$ are in the wrong order, and we swap them (exchange their positions). Then we look at $A[2]$ and $A[3]$, and, if $A[2] > A[3]$, we swap them. We continue doing this, stopping after we process $A[n-1]$ and $A[n]$.

As discussed in the lab, while 1 pass is not enough to sort an array, n passes suffice. However, here we will use a different approach: after each pass, we check whether any elements were swapped during this pass. If no elements were swapped, then the array is already sorted, and we stop the algorithm.

Input/Output:

Your functions should accept and array of n numbers $x_1, x_2, \ldots, x_n$ (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 $n = 5000, 10000, 15000, \ldots$ up to 30000. Plot both 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 reading, generating or writing 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 ($x_1 \leq x_2 \leq \ldots \leq x_n$). 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 ($x_1 \geq x_2 \geq \ldots \geq x_n$). 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 should generate 100,000 random arrays of size 50 each). Measure the overall runtime of sorting these inputs. There is no plot for this type of input, just report and compare the two resulting runtimes.

Plot generation

If you are using Python, the standard way would be to use matplotlib. You may find the following tutorials helpful:

- General tutorial: `https://matplotlib.org/tutorials/introductory/pyplot.html`.
- How to create a legend: `https://matplotlib.org/3.1.1/tutorials/introductory/usage.html#matplotlib-pyplot-and-pylab-how-are-they-related`
- For saving the resulting figure as a file, you may use plt.savefig(filename): `https://matplotlib.org/api/_as_gen/matplotlib.pyplot.savefig.html#matplotlib.pyplot.savefig` (check "Examples using matplotlib.pyplot.savefig" to clarify the usage).

While other languages certainly have plot-drawing capabilities, it probably would be simpler to save all generated data to a file, and draw the data using Python (or your favorite plot-drawing tool).

Write-up

Plots: Your write-up should contain all the required plots. It should be clear which plot corresponds to which input type, preferably by adding a caption (e.g. "Figure 1: Large random inputs"). For each plot, it should be clear which point corresponds to which algorithm, input size and running time.

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: Both of these algorithms have asymptotic quadratic running time ($O(n^2)$).

- Does the first plot reflect this? How do the two algorithms compare in terms of the running time?
- How about the second plot? Do you think this one is quadratic? Why do you think it looks the way it does?
- How does the third plot compare to the first and second?
- What about the fourth plot? Can you explain the behavior?
- What about the fifth input? Note that the total input size is greater than all other inputs combined; why the algorithms can handle it efficiently?
- Summarize in which situations these algorithms can be used in practice. Which one would you prefer?