Math 560 Project 1: Sorting Fall 2020

Due: Friday, October 9

In this project, you will implement a number of sorting algorithms that we have discussed in class. You will be required to implement your solutions in their respective locations in the provided `project1.py` file.

1 Problem Statement

You will implement five algorithms for sorting an array. We define this sorting task as:

- The input array will have a positive integer length.
  - Will not be asked to sort an empty array.
  - Could be asked to sort a singleton array.
- The elements need not be distinct (i.e. repeats allowed).
- The elements can be any (positive or negative) real number.

The goal is to sort the elements from the input array into ascending order so that the smallest element is at index 0, i.e.,

$$A[0] \leq A[1] \leq \cdots \leq A[k-1] \leq A[k] \leq A[k+1] \leq \cdots \leq A[end]$$

You will implement the following five algorithms as discussed in class:

- Selection Sort
- Insertion Sort
- Bubble Sort
- Merge Sort
- Quick Sort

1.1 Runtime Comparisons

In the provided code is the function `measureTime(preSorted = False, numTrials = 30)`. This function can be used to time your implementations and obtain plots of the runtime versus input size. There are several ways this function can be called:

- The call `measureTime()` will use randomly generated test arrays, and for each input size n considered, will average the runtime over 30 separate trials.
- The call `measureTime(False,x)` will use randomly generated test arrays, and for each input size n considered, will average the runtime over x separate trials, where x is an integer.
- The call `measureTime(True)` will use already sorted test arrays, and for each input size n considered, will average the runtime over 30 separate trials.
- The call `measureTime(True,x)` will use already sorted test arrays, and for each input size n considered, will average the runtime over x separate trials, where x is an integer.

Once the runtimes have been determined, a number of plots are created so you can see directly the runtime versus the input size.

1.2 Log-Log Runtime

The function `measureTime` will also generate a log-log plot of runtime versus input size for Selection Sort, Insertion Sort, and Bubble Sort. It then attempts to fit a line to these log-log plots and will output the fitted slope. It first attempts this fit using all of the runtime data (including even very small values of n) and will report those slopes. It then attempts the fitting using only larger values of n. One of the questions you must answer is which attempted fit gives more accurate results, and why you think that is the case.

Looking at log-log plots of the runtime versus the input size is a very common method used to interpret the runtime of a polynomial time algorithm. To understand why we do this, consider the following hypothetical:

Let's assume that we have an algorithm that runs in $O(n^k)$ time, where k is some positive integer. Once we have implemented this algorithm, how can we determine if it really has obtained the promised runtime complexity?

The claim that the runtime is $O(n^k)$ means that

$$T(n) \sim n^k \quad \Longrightarrow \quad T(n) = a\, n^k, \quad a \in \mathbb{R}.$$

Now take the log of both sides of this equation. This gives us

$$\log T = \log(a\, n^k) = \log(n^k) + \log a = k \log n + \log a.$$

$$\implies \quad \log T = k \log n + \log a.$$

So if we plot $y = \log T$ versus $x = \log n$, we should see a straight line! Moreover the slope of that line should be the value of k.

This means that if you want to verify that an algorithm runs in $O(n^2)$ time, you should plot the runtime versus the input size on a log-log scale. If you fit the resulting log-log plot to a straight line, the slope should be 2.

1.3 Questions

Along with your code, you should submit a short report (1-3 pages **of text**) that addresses the following questions. Your report can include any of the generated plots that *you deem relevant*. You are not required to include any of the images. Note that you should not directly answer these questions one at a time, but rather your report should discuss their answers with details/evidence obtained from the results of running your code.

- Do your algorithms behave as expected for both unsorted and sorted input arrays?
- Which sorting algorithm was the best (in your opinion)? Which was the worst? Why do you think that is?
- Why do we report theoretical runtimes for asymptotically large values of n?
- What happens to the runtime for smaller values of n? Why do you think this is?
- Why do we average the runtime across multiple trials? What happens if you use only one trial?
- What happens if you time your code while performing a computationally expensive task in the background (i.e., opening an internet browser during execution)?
- Why do we analyze theoretical runtimes for algorithms instead of implementing them and reporting actual/experimental runtimes? Are there times when theoretical runtimes provide more useful comparisons? Are their times when experimental runtimes provide more useful comparisons?

2 Provided Code

The file `project1.py` has a number of pre-defined functions listed below:

- `SelectionSort(listToSort)`: to contain your implementation for Selection Sort.
- `InsertionSort(listToSort)`: to contain your implementation for Insertion Sort.
- `BubbleSort(listToSort)`: to contain your implementation for Bubble Sort.
- `MergeSort(listToSort)`: to contain your implementation for Merge Sort.
- `QuickSort(listToSort, i=0, j=len(listToSort))`: to contain your implementation for Quick Sort. Note that the values of the indices `i` and `j` are present to help you recursively call the function on the partitions you create (so it will be in-place). For example, if the pivot is at location `p` after partitioning, you would recurse on the two partitions using `(i=0, j=p+1)` and `(i=p+1, j=len(listToSort))`. Note that `j` must point *one past* the end of the array. All testing with this function will be done with the function call `QuickSort(listToSort)`, which uses the default values for `i` and `j`.

Meanwhile the following functions are found in `project1tests.py`:

- `isSorted(unsortedList, sortedList)`: this function returns True if the array `sortedList` is the sorted version of the array `unsortedList`, and False otherwise.
- `testingSuite(alg)`: this function runs a number of tests on the algorithm in question. This is not an exhaustive list of tests by any means, but covers the edge cases for your sorting algorithms. The valid inputs to this testing function are the strings:
  - `'SelectionSort'`
  - `'InsertionSort'`
  - `'BubbleSort'`
  - `'MergeSort'`
  - `'QuickSort'`
- `measureTime(preSorted = False, numTrials = 30)`: this function runs your algorithms on a number of randomized inputs of varying sizes while tracking the runtimes. It will plot the runtime versus n for each algorithm and save these as `.png` files to the current directory. It also creates a log-log plot of the runtime for several of the algorithms, and will print some info about those plots.