CS2121 - Assignment 3 - 2020

Sorting A Random List

Published: February 26^{th}, 2020

Due date: March 11^{th}, 2020 at 11:00 pm via OWL

Note Well:

You must take note of the following:

- You are given one (1) .py file that you are to complete. For your work with this file take note of the following:
  - Your function names, along with the parameters they take **must** remain the same; **you cannot alter these**. If you alter these, you will lose significant marks and risk getting a mark of zero.
  - A file called `test.py` will be used by the TA to mark your assignment — I am providing the file we will actually use to test your code. You are free to use this on your own code.
  - Please **do not** prompt the user for input; all functions will be called by just calling the functions.
- Some of the descriptions are deliberately left somewhat vague. This gives you practice interpreting design instructions, but things will become clear as you get further on with your implementation.
- If there is something you are not sure how to do, you should first Google it or look it up in the Python documentation online: `https://docs.python.org/3/index.html`

Purpose

The purpose of this assignment is to familiarize yourself some algorithms for sorting collections and with basic performance analysis and visualization using `numpy`, and `pyplot` from the matplotlib package.

Summary

You will be writing an algorithm comparison class to compare the performance of several sorting algorithms on randomly generated data. This class will be called `SortAnalysis`.

Provided:

You are provided with an incomplete python script (`SortAnalysis.py`) that you are to finish. A testing script (`test.py`) is also included.

`SortAnalysis` class

This class will provide an implementation of four plus one (4+1) different sorting algorithms on randomly generated numpy arrays of floating-point numbers. It will also provide the ability to run and plot comparisons of different sort algorithms for arrays of different sizes. You are using numpy arrays throughout the class because they are effective for working with numerical data and they work nicely with the `pyplot` module you will be using. Your class will have no attributes (so no need for an `__init__` function) and nine (9) different methods:

1. `bubbleSort(self, array)`
   - This function performs an in-place sort of an input `array` slightly more efficiently than the example in class. Your function should perform a bubble sort such that once a complete pass through the array is made without any new comparisons needing to be made, the algorithm exits.
2. `insertionSort(self, array)`
   - This function performs an in-place sort of an input `array` just as shown in class.
3. `mergeSort(self, array)`
   - This function performs a destructive, meaning the input is changed, but not in-place, meaning the algorithm uses space outside of the input array, merge sort of an input `array`. This algorithm uses a recursive divide-and-conquer strategy to perform and efficient sort, but is still straightforward to implement.
   - The steps in this algorithm are as follows:
     (a) set `n` to be the length of the input `array`.
     (b) check if `n` is less than or equal to `1`, and if so, return the input array (this is called the *base case*).
     (c) otherwise we perform a merge sort on the input array in the following way.
     (d) first create two temporary numpy arrays of size about half the size of the input array (exactly half if `n` is an even number), such that the number of elements in the new arrays adds up to `n` (you may find Python's floor division useful for this).
     (e) then fill the temporary arrays with the contents of the input array (so that the input is now divided into two temporary arrays).
     (f) now make two *recursive calls*, i.e., calls to the very `mergeSort` function you are defining, on the temporary arrays. After each of these recursive calls has been executed, the temporary array will be sorted in order.
     (g) finally, merge the two temporary arrays into a single sorted array by calling another function called `_merge`, described below, which takes three arrays as arguments. Make sure that the array in the first argument is the input array, and that the latter two arrays are the two temporary arrays you created.
   - That's it! Once you have done this, you will have both implemented (probably) your first recursive algorithm and (probably) your first divide-and-conquer algorithm.

4. `_merge(self, array, array1, array2)`

   - This function performs a merge of two sorted arrays `array1` and `array2` and stores the result of sorting both arrays in a third input array `array`.
   - This function has a leading underscore in its name because it is not intended to be called from outside of the class.
   - The basic idea here is that you take advantage of the fact that `array1` and `array2` are already sorted, and combine them into a larger sorted array.
   - More specifically, you will iterate through both `array1` and `array2` and compare elements.
   - Starting with the first element of each array, you compare elements to see which is smaller.
   - Whichever of the two elements being compared is smaller gets added to the larger output `array` and you then compare the next smaller element (from the array that had the smaller of the two) to the current element from the other array.
   - You continue this process until you reach the end of one of the arrays, and then you copy over the remaining of the elements from the other array to the output.
   - The result is then the combined sorted list stored in `array`.

5. `mergeSort2(self, array)`

   - This function is a variation of the previous merge sort that uses a different base case.
   - The implementation of this function should be identical to that of `mergeSort`, except that instead of checking whether `n` is less than or equal to `1`, you will check whether it is less than or equal to some threshold size (that you will determine later).
   - If `n` is less than or equal to the threshold, then you will return the result of calling `insertionSort` on the input array instead, since insertion sort is known to be very efficient on small arrays.
   - You will investigate what improvement this makes later on in the assignment.

6. `bucketSort(self, array)`

   - The steps of this algorithm are as follows:
     (a) set `n` to be the length of the input array.
     (b) create an empty list `B` of size `n` ('B' for "buckets").
     (c) then initialize each element of this list to the empty list (creating a list of `n` empty lists).
     (d) now iterate through the input array and append each `element` of the array to the list of `B` at index $\lfloor$`n*element`$\rfloor$, where the floor $\lfloor\cdot\rfloor$ indicates the largest integer less than or equal to its argument (this places all of the elements of the array into one of the buckets)—numpy may be helpful here.
     (e) then sort each element of `B`, i.e., each bucket, using `insertionSort`.

(f) finally, iterate through each of the elements of `B`, and copy the elements in order back into the input array, to give the sorted result.

7. `fibonacciArray(self, n)`

   - This function returns a numpy array of `n` Fibonacci numbers starting from 5, 8, ...
   - I recommend that you use the function `np.zeros(n,dtype=int)` to create the array.
   - This function should not return an array smaller than length 2, so should raise a `ValueError` if `n` is less than 2.
   - This will be used for the purposes of running and plotting the results of tests of the sorting algorithms.

8. `runSort(self, sortName, sizes, seed)`

   - This function will run the sorting algorithm with the name `sortName` for randomly generated numpy arrays of different `sizes` and then return an array containing the running time for that sorting algorithm for each of the `n` runs.
   - This function takes three parameters: a reference name `sortName`, which will be one of the names of the functions implementing the sort algorithms you implemented (note this is a reference name and **not** a string); an integer numpy array `sizes` containing the sizes of the arrays for the sequence of runs of the sorting algorithm; and an integer `seed` that will be used to ensure that different calls to the function use the same sequence of random numbers.
   - The steps for this function are as follows:
     (a) seed the random number generator using `np.random.seed` and the input seed value.
     (b) create a numpy array `times` that is the same size as `sizes` to hold the running time of the runs of the sorting algorithm.
     (c) then for each `sizes[i]` do the following:
        i. generate an array `A` of size `sizes[i]` containing randomly generated floating-point numbers between 0 and 1.
        ii. store the (current) start time in a variable `start` using `time.time_ns()`.
        iii. call the `sortName` function on the array `A` (keep in mind that Python functions work by using `def` to bind a function name to the code following the name, so that the name becomes a reference to the function).
        iv. store the elapsed time in a variable `elapsed` by subtracting the now current time from the time stored in `start`.
        v. store the elapsed time in the `i`the element of the `times` array (make sure to store the elapsed time in *micro*seconds, noting that the function `time.time_ns()` produces times in *nano*seconds)—your plots will not display correctly if you don't do this.
     (d) finally return the array of running times.

9. `runAnalysis(self, sortList, n, filename)`

    - This function will perform a runtime analysis of an input list of sort algorithms, plot the results and then print the plot to a file.
    - This function takes three arguments and should do the following things.
    - The input `sortList` is a python list of sort function names for which the runtime analysis is to be run.
    - The input `n` is the number of runs to perform, where the sizes of the input arrays will be Fibonacci numbers starting from 5.
    - The input `filename` is a string containing the name that will be given to the printout of the plot.
    - So, the routine needs to generate a numpy array of Fibonacci numbers of size `n`, choose a seed value, and then run each of the sort functions in the `sortList` with this array of sizes with a sequence of calls to `runSort`.
    - Before you start, make sure to call `plt.clf()` from `pyplot` to clear the current plot.
    - Then for each sort function in the list, you should plot the array of sizes versus the array of times from the runs of that sort function on a log-log scale by calling `plt.loglog(sizes,times)`.
    - Doing this for each sort function in sequence will plot the results of all the runs on the same graph.
    - Then add to the plot a legend containing the names of the sort functions using `plt.legend(sortNameList)`, where `sortNameList` is a list of strings you create from the function names in `sortList` (note that for a function called `function` you access the name of the function as a string with `function.__name__`)— make sure that the order of this list is the same as the order of your calls to `runSort`.
    - Then add labels to the axes using `plt.xlabel("array size")` and `plt.ylabel("runtime (microseconds)")`.
    - Finally, print the plot using the `plt.savefig` function by adding the suffix `'.pdf'` to the input `filename`.

This completes the contents of the `SortAnalysis` class. You will use this class to perform a basic runtime analysis of the sort algorithms you implemented.

Your Runtime Analysis

You need to perform three different runtime analyses by constructing a `SortAnalysis` object and making calls to `runAnalysis`. For each of these analyses you will have to consider a question, for which you will provide an answer as a comment in your `SortAnalysis.py` file. There are also two additional questions concerning the operation of `mergeSort` and `bucketSort`.

Note well: You may include the code to run these analyses in your `SortAnalysis.py` file, but make sure that it is either encapsulated into methods of the class or commented out when you submit, because this code should *not* run by default. You stand to lose marks if importing your class causes runtime analyses to run.

1. The first runtime analysis should just compare the results of running `bubbleSort`, `insertionSort`, `mergeSort` and `bucketSort`. You should run each algorithm up to array sizes of at least 1,000 to 10,000, depending on the capabilities of your computer, but go as high as you reasonably can and expect to wait a few minutes or more to compute all of the runs. You will need to find the value of `n` that produces arrays (so Fibonacci numbers) of the required size. Save the plot as `comparison1.pdf`.

2. You should observe that two of the algorithms outperform the other two. The second runtime analysis will focus on just these two. For the best two algorithms, run each algorithm up to array sizes of at least 1,000,000 to 10,000,000, depending on the capabilities of your computer, but go as high as you can reasonably within a few minutes or more. If it was not clear already, the best algorithm should be clear at this point. Save the plot as `comparison2.pdf`.

3. From `comparison1.pdf` you should see that `insertionSort` performs very well for small array sizes. This gives you the idea that you might be able to improve merge sort by taking advantage of this. To figure out how to do this, perform a third runtime analysis. First take a look at `comparison1.pdf` to determine at approximately which array size `insertionSort` and `mergeSort` take the same amount of time. Use this information to implement the threshold for the new base case in `mergeSort2`. Run a few comparisons of `mergeSort` and `mergeSort2` to try to pick an optimal value of the threshold, and make sure to take account of the behaviour of both algorithms for larger array sizes.

   Once you have picked your threshold you will perform your last runtime comparison output, this time comparing `mergeSort`, `mergeSort2` *and* the best sorting algorithm to see whether your optimization of `mergeSort` makes a significant difference. Save the plot as `comparison3.pdf`.

Questions

You have to answer several questions as part of this assignment. The first two questions concern the functioning of `mergeSort` and `bucketSort`. The last three questions concern your runtime analyses. Once you have completed the three analyses, or as you go, make sure to answer the following three questions as **numbered** comments in your `SortAnalysis.py` file (A space is provided for these in the supplied file).

1. Explain briefly how `mergeSort` is able to sort the input array by using recursion. Why is this approach called a "divide-and-conquer" approach?

2. In `bucketSort`, explain why the strategy of using the value of an element to determine which bucket it goes in ensures that once the buckets are sorted the whole array is sorted by combining the buckets in order.

3. Taking a look at the slopes of the curves for each of the four sorting algorithms in `comparison1.pdf`. What can you say about the relative time complexity of the four algorithms? Can you estimate the time complexity of each algorithm from the plot? If so, what are your complexity estimates for the four algorithms?

4. Taking a look now at the larger size runs of the best two algorithms in `comparison2.pdf`, can you detect any difference in their runtime complexity? And can you explain why the best algorithm is so much better?

5. Examining `comparison3.pdf`, does your modification to produce `mergeSort2` produce an *improved* algorithm? Even if it does not beat the best algorithm in terms of runtime here, is there a reason why `mergeSort2` might nevertheless be a better algorithm overall?

Submitting the Assignment

You will submit the files via OWL, under the assignments tab. **Only submit your complete python file** (`SortAnalysis.py`) **and the three runtime analysis plots** (`comparison1.pdf`, `comparison2.pdf`, `comparison3.pdf`) via OWL. Please follow the instructions carefully and do what is asked, but not more, to ensure that you do not unnecessarily lose marks. Make sure not to modify the function names.