Ternary Mergesort and Quicksort

CSCI 112, Lab 6

File names: Names of files, functions, and variables, when specified, must be EXACTLY as specified. This includes simple mistakes such as capitalization.

Individual work: All work must be your own. Do not share code with anyone other than the instructor and teaching assistants. This includes looking over shoulders at screens with the code open. You may discuss ideas, algorithms, approaches, *etc.* with other students but NEVER actual code. Do not use code written by anyone else, in the class or from the internet.

Documentation: Each file should begin with a docstring that includes your name, the class number and name, the lab number, and a short description of the lab, as well as documentation pertinent to that particular file.

The project: Each of the sorting algorithms mergesort and quicksort divides the list into two portions, sorts the portions, and then puts them back together.

Rewrite each of these algorithms so that they divide the list into three portions, sort the portions, and then put them back together.

Use the implementation that comes with the textbook as a start. Try to make as few changes as possible. Mergesort should be fairly easy.

Quicksort's partition function will be more difficult. You will have to select two pivots, $p1$ and $p2$, and then arrange the list into three sections:

$$n < p1$$
$$p1 \leq n < p2$$
$$p2 <= n$$

There is a fairly easy way to do it if you let yourself traverse the array twice, instead of once, to make the partition. (Think about using the original partition algorithme, twice.) This, of course, will make the partition algorithm $O(2n)$, but of course, $2n = O(n)$. Implement this first so that you can create a unit test module and test your sorting implementations.

Improved quicksort partition: Develop a quicksort ternary partition that traverses the array only once, and partitions in place (*i.e.* using constant extra storage, that doesn't grow with the array size). To partition an array between indices `low` and `hi`, you will need two pivots, which will be elements at `low` and `hi`. If the first element is larger than the last, swap them first.

Then, the idea is to create three ranges of elements:

$$range1 < pivot1 \leq range2 < pivot2 \leq range3$$

You can do this with three indices into the array: `left`, `right`, which start at `low+1` and `hi-1`, and `i` which starts out at the same place as `left`, and moves gradually to the right.

When `i >= right` the list will be partitioned. At each step, you need to compare `a[i]` with both `a[left]` and `a[right]`. Depending on the three possible outcomes of these comparisons, you need to move one or more of `right`, `left`, and `i`.

Think carefully about how this has to be done to accomplish the partitioning required. Implement several variations and test them until you're sure you've got it right.

A trace of the algorithm as I implemented it is shown below:

low	left,i									right	hi
3	9	4	5	1	5	1	9	10	8	2	9

low	left,i								right		hi
3	2	4	5	1	5	1	9	10	8	9	9

low		left,i							right		hi
3	2	4	5	1	5	1	9	10	8	9	9

low		left	i						right		hi
3	2	4	5	1	5	1	9	10	8	9	9

low		left		i					right		hi
3	2	4	5	1	5	1	9	10	8	9	9

low			left		i				right		hi
3	2	1	5	4	5	1	9	10	8	9	9

low			left			i			right		hi
3	2	1	5	4	5	1	9	10	8	9	9

low				left			i		right		hi
3	2	1	1	4	5	5	9	10	8	9	9

low				left			i	right			hi
3	2	1	1	4	5	5	8	10	9	9	9

low				left				i,right			hi
3	2	1	1	4	5	5	8	10	9	9	9

low				left			right	i			hi
3	2	1	1	4	5	5	8	10	9	9	9

low			left					i,right			hi
1	2	1	3	4	5	5	8	9	9	9	10

Testing runtime: The runtime of binary and tertiary quicksort and mergesort should be very close to each other, all of them being $O(\log n)$. The ternary algorithms should be slightly faster since $\log_3(n) < \log_2(n)$, but since they have more overhead they may be slightly slower, at least for small lists. It is possible that they will show some improvement over the binary versions as the array gets largs. Run some tests to see if you can get consistent results as to whether tertiary is faster or slower than binary, or about the same. Write up your findings in a short document called either `sorts_runtime.txt` or `sorts_runtime.tex`. Please do not use any other word processing: either plain ascii text or LaTeX.

Fair tests: Make sure you instrument your tests so that each algorithm is used on the same starting array. If you use random arrays then some algorithms may "get lucky" and have easier starting arrays than others, skewing the results.

File names: Put all four sorting algorithms (mergesort and quicksort, binary and tertiary) in a single file called `sorts.py`. Put your unit tests in the file `sorts_test.py`. Put your runtime tests in a file called `sorts_runtime.py`. Zip all three together with your writeup into a folder called `lab06yourname` (where `yourname` is your name) and submit to canvas.