Homework 07: Magic Sort

So far in CSE 2050, we've analyzed five different sorting algorithms: bubble, selection, insertion, merge, and quick. Each of these sorting algorithms has their own set of strengths and weaknesses. Namely, each algorithm works best for certain categories of inputs (e.g., `insertionsort` has a $O(n)$ running time on lists that have a constant number of items out of place.)

But, which algorithm should you use for general purpose sorting (where you may not know of lot of information about the input sequences)? In many programming languages, the general purpose sorting algorithm used is actually a *hybrid* sorting algorithm that utilizes several different sorting algorithms under the hood. For instance, Python uses Timsort as its general purpose sorting algorithm.

In this homework, we will be creating our own hybrid sorting algorithm: **magicsort**.

None of the individual pieces of `magicsort` are too complicated on their own, but we need to proceed through this assignment methodically with test-driven development to ease the debugging burdens that may arise. As such, we'll break down each piece of the overall `magicsort` algorithm into its own function and develop tests for that function alone. This ensures that it is working on its own before we move on and start calling it in subsequent functions.

Part 1: `linear_scan`

Step One

In `magicsort.py`, implement a function named `linear_scan(L)` that takes a list as input and returns the particular `MagicCase` that applies to it. In the starter code, we've already defined `MagicCase` as an enumeration, which is just a set of symbolic names that are bound to unique values. Using enumerations, rather than plain integers, allows us to have much more readable and maintable code. If you are curious, the Python documentation for enumerations can be found here.

The particular `MagicCase` that `linear_scan` should return is given by the following rules:

- If `L` is already sorted, return `MagicCase.SORTED`.
- If `L` is has fewer than e.g. 10 adjacent elements that are inverted (`L[i-1] > L[i]`), return `MagicCase.CONSTANT_NUM_INVERSIONS`. There is a global constant `INVERSION_BOUND` already defined in the starter code that you should utilize during this check.
- If `L` is reverse sorted, return `MagicCase.REVERSE_SORTED`.
- If none of the above cases apply, return `MagicCase.GENERAL`.

Step Two

In `test_magicsort.py`, create a test class for `linear_scan` and use the tests to ensure it exhibits the correct behavior on a variety of input lists before moving on.

Examples Several examples of behavior shown below.

```
>>> L = [1, 2, 3, 4, 5]
>>> linear_scan(L)
MagicCase.SORTED

>>> L = [5, 4, 3, 2, 1]
>>> linear_scan(L)
```

```
MagicCase.REVERSE_SORTED

>>> L = [1, 2, 4, 3, 5]
>>> linear_scan(L)
MagicCase.CONSTANT_NUM_INVERSIONS
```

Part 2: `reverse_list`

Step One

In `magicsort.py`, implement a function named `reverse_list(L)` that takes a list as input and efficiently reverse it in-place. You function should proceed as follows:

- swaps the first and last elements, then
- swaps the second and penultimate elements, then
- … repeats this pattern until the list is sorted.

This function should run in $O(n)$ time with $O(1)$ memory overhead.

Step Two

In `test_magicsort.py`, create a test class for `reverse_list` and use the tests to ensure it exhibits the correct behavior on a variety of input lists before moving on.

Part 3: `magic_insertionsort`

Step One

In `magicsort.py`, implement a function named `magic_insertionsort(L, left, right)` that takes an input list along with indices left/right and uses insertion sort to sort (in-place) ONLY the sublist given by `L[left:right]`. This function should:

- have $O(n)$ running time when $O(1)$ items are out of place (i.e., when `MagicCase.CONSTANT_NUM_INVERSIONS` applies)
- have $O(n^2)$ worse-case running time
- have $O(1)$ memory overhead – since its in-place and writes directly to the input list

Step Two

In `test_magicsort.py`, create a test class for `magic_insertionsort` and use the tests to ensure it exhibits the correct behavior on a variety of input lists before moving on.

Examples Example behavior shown below.

```
>>> L = [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
>>> magic_insertionsort(L, left=2, right=5)
>>> print(L)
[9, 8, 5, 6, 7, 4, 3, 2, 1, 0]
```

Part 4: `magic_mergesort`

Step One

In `magicsort.py`, implement a function named `magic_mergesort(L, left, right)` that takes an input list along with indices left/right and uses merge sort to sort ONLY the sublist given by `L[left:right]`. This function should:

- call `magic_insertionsort` to sort sublists with 20 items or fewer, as quadratic sorting algorithms actually outperform merge sort on very small lists
- have $O(n \log n)$ worse-case running time
- have $O(n)$ memory overhead

Step Two

In `test_magicsort.py`, create a test class for `magic_mergesort` and use the tests to ensure it exhibits the correct behavior on a variety of input lists before moving on.

Examples Example behavior shown below.

```
>>> L = [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
>>> magic_mergesort(L, left=2, right=5)
>>> print(L)
[9, 8, 5, 6, 7, 4, 3, 2, 1, 0]
```

Part 5: `magic_quicksort`

Step One

In `magicsort.py`, implement a function named `magic_quicksort(L, left, right, depth=0)` that takes an input list along with indices left/right and uses quick sort to sort ONLY the sublist given by `L[left:right]`. This function should:

- **Use the last item in a sublist as the pivot element**. This does not give optimal results, but it allows us to analyze how `magicsort` handles edge cases.
- **Keep track of the recursion depth**. If `depth` ever gets too large, indicating that we are choosing poor pivots, this function should call `magic_mergesort` to sort that particular sublist (this prevents us from experiencing the dreaded $O(n^2)$ worst-case running time of quick sort).
  - The best-case maximum depth for quick sort should be $\log_2(n) + 1$. Since the pivot will not always be the median element, we expect the actual depth to be a bit higher. However, if the depth ever exceeds **twice the best-case maximum depth**, sort the sublist using `magic_mergesort`. You can use `math.log2()` for these calculations.
- call `magic_insertionsort` to sort sublists with 20 items or fewer, as quadratic sorting algorithms actually outperform quick sort on very small lists.
- have $O(n \log n)$ average *and* worse-case running times because we transition to `magic_mergesort` if the pivot selection is bad.
- have $O(\log n)$ memory overhead, due to the function call stack (unless `magic_mergesort` is invoked).

Step Two

In `test_magicsort.py`, create a test class for `magic_quicksort` and use the tests to ensure it exhibits the correct behavior on a variety of input lists before moving on.

Examples Example behavior shown below.

```
>>> L = [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
>>> magic_quicksort(L, left=2, right=5)
>>> print(L)
[9, 8, 5, 6, 7, 4, 3, 2, 1, 0]
```

Part 6: `magicsort`

Step One

We're finally ready to fully implement our hybrid sorting algorithm! In `magicsort.py`, implement a function named `magicsort(L)` that takes an input list and does the following:

- Calls `linear_scan` on the list to determine which `MagicCase` it falls into.
- If the input falls into the `SORTED`, `CONSTANT_NUM_INVERSIONS`, or `REVERSE_SORTED` cases, immediately return or call the approriate linear time sorting method.
- If the input falls into the `GENERAL` case, call `magic_quicksort` on `L` with `left` and `right` set to 0 and `len(L)`, respectively.

Your `magicsort` algorithm should:

- mutate the input list so that it is in a sorted state upon return. In other words, to sort a list `L` we would use `magicsort(L)` rather than `L = magicsort(L)`.
- keep track of which of the sub-algorithms are invoked during the overall sorting process and return them as a set. I recommend adding an additional `alg_set=None` default formal parameter to each of the sorting functions from parts 2-5 to help handle this.

Step Two

In `test_magicsort.py`, create a test class for `magicsort` and use the tests to ensure it exhibits the correct behavior on a variety of input lists.

Examples Several examples of behavior shown below.

```
>>> L = [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
>>> magicsort(L)
{'reverse_list'}
>>> print(L)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> L = [1, 2, 4, 3, 5]
>>> magicsort(L)
{'magic_insertionsort'}
>>> print(L)
[1, 2, 3, 4, 5]

>>> # the following list is structured to give very poor pivots in quicksort
>>> # when choosing the last element as the pivot.
>>> # this should result in an invocation of magic_mergesort.
>>> # once the sublists get small enough, magic_insertionsort will also be invoked.
```

```
>>> L = list(range(10)) + list(reversed(range(10,100)))
>>> magicsort(L)
{'magic_quicksort', 'magic_mergesort', 'magic_insertionsort'}
>>> print(L)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21,
22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41,
42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61,
62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81,
82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99]
```

Imports

No imports are allowed on this assignment, with the exception of:

- `enum`: for the `MagicCase` enumeration (this is already defined in starter code).
- `math`: for calculating the best-case recursive depth for quick sort.
- `unittest` and `random`: for testing purposes.
- `typing`: not required, but students have requested it in the past.

Grading

This assignment is 100% manually graded. Your code will be graded on structure, efficiency, readability, and correctness.

Submission

Submit the following files:

- `magicsort.py`
- `test_magicsort.py`

Students must submit individually to Gradescope by the posted due date (Tuesday, March 19th at 11:59pm) to receive credit.