Module 7 Homework - Magic Sort

General-purpose sorting algorithms, like python's `list.sort()`, use a combination of the algorithms we've learned so far. Generally speaking, we want an algorithm that:

- Runs in $\mathcal{O}(n)$ if lists are sorted, reverse sorted, or have at most $\mathcal{O}(1)$ unsorted pairs
- Has a worst-case running time of $\mathcal{O}(nlogn)$, like meregesort
- Is as fast as quicksort for large random lists

We cannot achieve all 3 behaviors with any single algorithm we have seen so far, so we will switch between them to create a general purpose sorting algorithm of our own - `magic_sort()`.

1) `linear_scan(L)` We'll begin with a linear scan to identify some common special cases:

   - **Case 0** - None of the cases below apply.
   - **Case 1** - List is already sorted.
   - **Case 2** - $\mathcal{O}(1)$ unsorted pairs (i.e. `L[j] > L[j+1]`). Use 10 pairs as the upper limit for this case.
   - **Case 3** - List is reverse sorted.

   Write and test a function `linear_scan(L)` that does a single linear scan of a list `L` and returns an integer denoting which case that lists fits into (return value will be one of `{0, 1, 2, 3}`).

2) `insertion_sort(L, left, right)` - write a version of insertion sort that only sorts the sublist `L[left:right]`. We're using Python's index convention, so we will go up to but not including `right` - if you call this function with `left=0` and `right=len(L)`, it should sort the entire list.

   - $\mathcal{O}(n)$ running time when at most $\mathcal{O}(1)$ items are unsorted (case 2 above)
   - $\mathcal{O}(n^2)$ worst case running time
   - This algorithm should be **stable** and **in-place**:
     - **stable** - never swaps the ordering of equal items
     - **in-place**
       - $\mathcal{O}(1)$ memory overhead
       - operates on the original list.

3) `reverse_list(L)` - write a function that efficiently reverses a list by:

   - swapping the first and last elements
   - swapping the second and penultimate elements
   - ... and so on, until the list is sorted

   This should run in $\mathcal{O}(n)$ time and in-place ($\mathcal{O}(1)$ memory overhead and operate on the original list). It does not need to be stable.

4) `merge_sort(L, left, right)` - uses mergesort to sort the sublist `L[left:right]`.

   - Make sure you **only** sort the specified sublist
   - Switch over to insertion sort for sublists of 20 or fewer items - quadratic algorithms actually outperform meregesort and quicksort on small lists.
   - This algorithm should have a $\mathcal{O}(nlogn)$ running time and sort the passed in list object, but it is not truly in-place (you will end up using $\mathcal{O}(n)$ memory overhead) and does not need to be stable.

5) `quick_sort(L)` - uses quicksort to sort `L`. We'll make a few modifications to the standard implementation of `quick_sort`:

- **Use the last item in a sublist as the pivot element**. This won't give optimal results, but it will make it easy for us to demonstrate how magic sort handles edge cases.
- **Keep track of the recursive depth**. Pass along a parameter `depth` with each recursive call that tracks how deep the recursive stack is. You can do this by incrementing depth by 1 at the top of every call.

  When `depth` gets too large, it indicates that we are consistently choosing poor pivots. **Sort this sublist with mergesort if the depth is greater than $3 * (log2(n) + 1)$, where `n` is the number of items in the original list.** The best-case maximum-depth should be `log2(n)+1`, and we give ourselves a 3x overhead to account for the fact that not every pivot will be a median. You can use `math.log2()` when calculating the maximum depth for a list.
- This modified version of quicksort should have a worst *and* average running time of $\mathcal{O}(nlogn)$, because we transition to mergesort if pivots are bad. It should also require $\mathcal{O}(1)$ memory overhead, unless we have to call mergesort, and it does not need to be stable.

6) With the functions above implemented, we're ready to implement `magic_sort()`:

   - Call `linear_scan()` to detect any special cases
   - If `linear_scan()` returns 1, 2, or 3, handle the edge case appropriately (immediately return or call the appropriate linear solution).
   - If `linear_scan()` returns 0, call `quicksort(L, left=0, right=len(L))`

   `magic_sort(L)` should sort `L` and should not return anything:

```
>>> import random
>>> is_sorted = lambda L: not any(L[i] > L[i+1] for i in range(len(L)-1))
>>> n = int(1E5)
>>> L = [(n-i) for i in range(n)]
>>> magic_sort(L) # reverse sorted - O(n)
>>> assert is_sorted(L)
>>> L = [(n-i) for i in range(n)]
>>> L[:20] = [-i for i in range(20)] # 20 pairs out of order - O(nlogn)
>>> magic_sort(L)
>>> assert is_sorted(L)
>>> L = [random.randint(0, n) for i in range(n)] # random liset
>>> magic_sort(L) # O(nlogn) expected, though L will occasionally be a special case
>>> assert is_sorted(L)
```

Tests

There's *a lot* of potential for undebuggable spaghetti code in this assignment - it is crucial that you approach things incrementally using TDD. Write a suite of test cases, then implement functionality, in the following order:

1) `linear_scan`
2) `reverse_list`
3) `insertion_sort`
4) `merge_sort`

5) `quick_sort`
6) `magic_sort`

For algorithms that should sort a sublist, be sure to test that:

- they *only* sort that sublist
- they do not accidentally overwrite any elements

```
class test_insertion(unittest.TestCase):
    def test_middle(self):
      # Setup
      n = 1000
      L = [random.randint(0, n) for i in range(n)]
      i_left = len(L)//4            # beginning of sublist to-sort
      i_right = 3*len(L)//4         # end of sublist to-sort
      L_pre_sort = L[:]             # full copy of unsorted L
      L_middle = L[i_left:i_right]  # copy of to-be-sorted portion

      # Sort
      insertion_sort(L, i_left, i_right)
      L_middle.sort()

      # Test
      self.assertEqual(L[:i_left], L_pre_sort[:i_left])   # left of sublist unchanged
      self.assertEqual(L[i_right:], L_pre_sort[i_right:])  # right of sublist unchanged
      self.assertEqual(L[i_left:i_right], L_middle)  # middle is sorted
```

Imports

No imports allowed on this assignment, with the following exceptions:

- Any modules you have written yourself
- `typing` - this is not required, but some students have requested it
- `math.log2`
- For testing only (do not use these for functionality in any other classes/algorithms):
  - unittest
  - random

Submission

At minimum, submit `MagicSort.py` and `TestMagicSort.py`.

Students must submit **individually** by the due date (typically Tuesday at 11:59 PM EST).