Homework 07: Magic Sort

We've analyzed five different sorting algorithms (bubble, selection, insertion, merge, and quick), each with its own strengths and weaknesses. Which algorithm should we use if we don't know much about the input list?

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, adding a bit of logistical overhead to select a sorting algorithm with the best set of strengths and weaknesses for a given list. Python uses the hybrid algorithm Timsort for sorting.

In this homework, we will be creating our own hybrid sorting algorithm, `magicsort`, which will:

- Scan the list to see if we have a special case, like:
  - List is already sorted -> return.
  - List is reverse-sorted -> use `sort_reversed`.
  - List is almost sorted (only has a few items out-of-order) -> use `insertion_sort`.

If none of the above cases are true, we try to sort the list with a modified `quicksort`. This algorithm behaves as normal, but:

- Always uses the last element in a sublist as the pivot.
- If we get too many bad pivots, transitions to `mergesort`.
- For both `quicksort` and `mergesort`, we will transition to `insertion_sort` once our sublists get below 20 items.

This assignment can be deceptively complex. If you've never tried to piece together four or five algorithms to achieve recursive functionality, you will quickly find out why we push incremental test-driven development so hard. It's nigh-impossible to debug unless you have a thorough suite of unittests that you implement one-by-one for each algorithm.

We have provided you a suite of unittests for each function. Gradescope will also run these unittests, and **you will be graded on passing them**, but the test cases are hidden in gradescope. This is to incentivize you to debug locally rather than with the autograder. The testcases used in gradescope are functionally identical to the ones we have provided you - the only differences are some decorators we use to give unittests point-values and have them time-out after a few seconds.

Part 1: `linear_scan`

We will start by scanning our list from left to right, counting how many times an element is larger than the next (`L[i] > L[i+1]`). We'll refer to these out-of-order pairs as "inversions." Based on the total number of inversions, we will return a value denoting which kind of sorting we want to do.

$n_{inversions}$	Return Value	Description
0	`MagicCase.SORTED`	List is already sorted
`n-1`	`MagicCase.REVERSE_SORTED`	List is reverse-sorted
`<=INVERSION_BOUND`	`MagicCase.CONSTANT_INVERSIONS`	At most some constant number of inversions
Other	`MagicCase.GENERAL`	None of the above special cases apply

We have already defined a constant `INVERSION_BOUND` and an **enumeration** called `MagicCase` to help make this function's return value readable. An enumeration is a set of symbolic names that are bound to unique values (docs). We could just use integers instead, but enumerations make our code more readable:

```
def magicsort(L):
    """2050's hybrid sorting algorithm"""
    inversion_case = linear_scan(L)
    if inversion_case == MagicCase.GENERAL: # Readable. Demure. Mindful.
        quicksort(L)

def magicsort(L):
    """2050's hybrid sorting algorithm"""
    inversion_case = linear_scan(L)
    if inversion_case == 3: # Unreadable. Brazen. Boorish.
        quicksort(L)
```

Examples

```
>>> 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_INVERSIONS
```

Sorting function notes

Next we will implement 4 sorting functions. For each of these, we use the following nomenclature:

- `sorting_alg(L: list, left: int, right: int, algs_used: set[str])`
  - `L` - the list to be sorted.
  - `left, right` the leftmost and rightmost indices of the sublist in `L` to be sorted. We will follow Pythonic standards by including the item at index `left` and excluding the item at index `right`.

    ```
    >>> L = [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
    >>> #idx:0  1  2  3  4  5  6  7  8  9
    >>> sorting_alg(L, left=2, right=5)
    >>> print(L) # Note that list is only sorted from [2:5]
    [9, 8, 5, 6, 7, 4, 3, 2, 1, 0]
    ```

  - `algs_used` - a set of the names of the sorting algorithms used. The easiest way to get the name of a function is by calling `func.__name__`:

    ```
    >>> def foo():
    ...     pass
    ...
    >>> foo.__name__
    'foo'
    ```

Part 2 - `reverse_list`

Write a function that reverses a list. Your running time should be `O(n)` and your memory overhead should be `O(1)`. Our intention here is for you to use a for loop, but if you'd prefer to do something clever with slicing, go ahead.

Part 3: `magic_insertionsort`

Sorts the sublist `left:right` using insertion sort. This function should have:

- $O(n)$ running time when $O(1)$ items are out of place.
- $O(n^2)$ worse-case running time.
- $O(1)$ memory overhead.

A common bug here is to sort too much of the input list - ensure you only sort the sublist `[left:right]`, or the running times of `magic_mergesort` and `magic_quicksort` will be quadratic.

Part 4: `magic_mergesort`

Sorts the sublist `left:right` using merge sort. This function should:

- Call `magic_insertionsort` to sort sublists with 20 items or fewer. Quadratic sorting algorithms outperform `nlogn` algorithms on these small lists.
- Have $O(n \log n)$ worse-case running time.
- Have $O(n)$ memory overhead.

Part 5: `magic_quicksort`

Implement a function named `magic_quicksort(L, left, right, depth=0)` that sorts the items in `L` from index `left` up to but not including the item at index `right`. This function should:

- Use the last item in the sublist `[left:right]` as the pivot element.
- Keep track of the recursive depth using the parameter `depth`:
  - The best-case maximum depth for quicksort is `log2(len(L)) + 1`.
  - We expect to get a bit deeper, since we are not always getting the median value as our pivot.
  - Therefore, we will transition to mergesort for the sublist `[left:right]` if depth exceeds **three times** the optimal depth to avoid the dreaded `O(n^2)` worst-case running time of quick sort.
  - You can use `math.log2` to calculate the best-case maximum depth.
- Transition to `magic_insertionsort` to sort sublists with 20 items or fewer.
- Have $O(n \log n)$ average *and* worst-case running times (we avoid the `O(n^2)` worst case by transitioning to `mabic_mergesort` when pivots are bad)
- have $O(\log n)$ memory overhead if we get good pivots. This will become `O(n)` if we have to transition to `magic_mergesort`.

Part 6: `magicsort`

We're finally ready to fully implement our hybrid sorting algorithm! 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`, `REVERSE_SORTED`, lr `CONSTANT_INVERSIONS` 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.

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 you can use it if you'd like

Grading

This assignment is partially manually and partially automatically graded. There are hidden auto-grader tests that will count towards your grade. You have access to these tests locally - make sure you're passing all the provided unittests.

Submission

Submit the following files:

- `magicsort.py`

Students must submit individually to Gradescope by the posted due date (Tuesday 10/22 at 11:59pm) to receive credit.