

Mod 6 Homework - Quadratic Sorts

Modify the bubble and insertion sort algorithms for (situationally) better performance.

Part 0 - tests

Write unittests for the following deliverables in `TestHw6.py`. Use a separate unittest class for each function:

```
class TestCocktailSort(unittest.TestCase):
    # Thorough suite of unittests

class TestOptInsertionSort(unittest.TestCase):
    # Thorough suite of unittests

class TestBSSublist(unittest.TestCase):
    # Thorough suite of unittests (provided for you)
```

Write your tests first - it will make the rest of the assignment easier. “Go slow to go fast.”

Some scenarios to consider when testing sorting algorithms (use a separate unittest for each):

- initially sorted list
- reverse-sorted list
- randomly sorted list (feel free to use the `random` module for unittests)
- arbitrary list lengths (e.g. `n = 0, 1, 2, 3, ...`)

Part 1 - cocktail_sort

`bubble_sort` works well on rabbits (large items near the beginning) but poorly on turtles (small items near the end). Consider the following list:

```
[5, 2, 2, 2, 2, 2, 1] # initial list - 5 is a rabbit, 1 is a turtle
[2, 2, 2, 2, 2, 1, 5] # first pass - 5 is in final position
[2, 2, 2, 2, 1, 2, 5] # second pass
[2, 2, 2, 1, 2, 2, 5] # third pass
[2, 2, 1, 2, 2, 2, 5] # fourth pass
[2, 1, 2, 2, 2, 2, 5] # fifth pass
[1, 2, 2, 2, 2, 2, 5] # sixth pass - 1 is (finally) in final position
```

- 5 (rabbit) moves all the way to the end in one pass
- 1 (turtle) requires 6 passes to move to the beginning

We can reduce the impact of turtles by alternating directions with each pass: scan left-to-right, then right-to-left, then left-to-right, and so on. This is called `cocktail_sort`. Consider the same list as above, sorted with `cocktail_sort`:

```
[5, 2, 2, 2, 2, 2, 1] # initial list
[2, 2, 2, 2, 2, 1, 5] # first pass: left-to-right
[1, 2, 2, 2, 2, 2, 5] # second pass: right-to-left
```

Implement `cocktail_sort`. It should sort lists with a constant number of rabbits and turtles in $O(n)$.

Part 2 - opt_insertion_sort and biary_search

In a first pass at `insertion_sort`, you might come up with something like the following:

```
1 def insertion_sort(L):
2     n = len(L)
3     for j in range(n): # go through every item
4         for i in range(n - 1 - j, n - 1): # bubble it into a sorted sublist
5             if L[i] > L[i+1]: # 1 comparison
6                 L[i], L[i+1] = L[i+1], L[i] # 2 writes
7             else: break
```

The code above has some inefficiencies. Assuming the sorted sub-list contains m items, the worst case for bubbling an item into its correct position involves $\sim 3m$ operations:

- m comparisons (line 5)
- m writes as items in the sorted sublist are shifted (line 6)
- m writes as the item being added to the sorted sublist is written (line 6)

We can do better:

- 1) Use a binary search to reduce the number of comparisons to $O(\log m)$:

```
def opt_insertion_sort(L):
    # some logic
    # pos = bs_sublist(L, left, right, item)
    # more logic

def bs_sublist(L, left, right, item):
    # logic
    # returns position
```

- Return the minimum index where `item` can be written
 - You can use a recursive or an iterative binary search.
- 2) Reduce the number of times the new item (`L[i]`) is written to $O(1)$ by storing it in a temporary variable once instead of writing it every time you shift an item, as above.
 - **Do not use the built-in `list.insert()`.**

Your optimized insertion sort should be 2~5x faster than the insertion sort algorithm given above:

```
===Random Distribution===
      insert  opt_insert
      n  t(ms)   t(ms)
-----
10000  2.40     0.49
20000  4.83     1.01
30000  7.20     1.47
40000  9.68     1.93
50000 12.15     2.40
-----
```

Submitting

At a minimum, submit the following files:

- `TestHw6.py`
 - `TestCocktailSort()`
 - `TestOptInsertionSort()`
- `hw6.py`
 - `cocktail_sort()`
 - `bs_sublist()`
 - `opt_insertion_sort()`

Students must submit **individually** by the due date (typically Tuesday at 11:59 pm EST) to receive credit.

Grading

We have provided unittests for `bs_sublist()`, and will use those to auto-grade that function.

`cocktail_sort()` and `opt_insertion_sort()` will be manually graded for both functionality and unittests.

- 40 - `cocktail_sort`
 - 20 - tests
 - 20 - functionality
- 20 - `binary_search`
 - autograded
- 40 - `opt_insertion_sort`
 - 20 - tests
 - 20 - functionality

Our manual grading will consider the following:

- Did you structure your code according to best practices in object-oriented programming? (largely irrelevant in this assignment)
- Did you thoroughly test your code using the unittest module?
- Did you choose and correctly implement the best data structures and algorithms?
- Is the code well-organized and documented? (use docstrings, comments, whitespace, and reasonable naming conventions)

Feedback

If you have any feedback on this assignment, please leave it [here](#).