Project 4: Sorting Algorithms

Due Thursday, February 24th @ 10:00 PM ET

This is not a team project, do not copy someone else's work.

Make sure to read the entire project description, especially the grading policies.

Background Information

A **sorting algorithm** is an algorithm that puts elements in a certain order. Such algorithms are often used to organize an array or list in numerical or lexicographical order. However, their use is not limited in scope to such simple orderings, a fact that will be demonstrated in this project.

Throughout the 20th century, as the domain of problems to which computers were applied grew, so to did the size of data sets that required sorting. This resulted in the rapid development of sorting algorithms. Simple *O(n^2)* algorithms, such as selection and bubble sort, were supplemented by faster *O(n log(n))* algorithms, such as quick or merge sort. Still, these *O(n^2)* algorithms have their place to this day because they are often faster for sorting small sets of data. Optimized modern sorting methods use hybrid techniques, which leverage the recursive nature of quicksort or merge sort by using these algorithms for large sets of data, but which use an algorithm such as insertion sort for the smaller fragments of data that the input ends up being separated into.

This project will expose you to insertion sort, selection sort, bubble sort, merge sort, and quicksort. Additionally, it will include a hybrid sort using merge and insertion sorts. Python's built in `list.sort` is actually based on a (somewhat more advanced) merge/insertion hybrid sort.

In addition to the overviews of each sort presented below, we encourage you to refer to the relevant sections in Zybooks.

Bubble Sort

Bubble sort is one of the simplest sorting algorithms, and it works by repeatedly traversing a list and swapping adjacent elements whenever it finds two that are out of order. This traversal is then repeated until a complete traversal is completed without having to do any swaps, which indicates that the list has been sorted.

Like selection and insertion sorts, it has *O(n^2)* worst/average case and *O(n)* best case (if the list is already sorted) time complexity, and can operate in-place for *O(1)* auxiliary space complexity. Bubble sort, however, tends to be the slowest of the sorting algorithms mentioned here in practice.

Insertion Sort

Insertion sort works by keeping track of sorted and unsorted portions of the list, and building up the sorted portion on the lefthand side of the list. To start, the first element is considered sorted (a single-element list is always sorted), and the remainder of the list is the unsorted portion. Next, the first element of the unsorted portion is compared to each element of the sorted portion in reverse order until its proper place in the sorted portion is found. Finally, the element from the unsorted portion is *inserted* into the list at the proper spot, which for arrays requires a series of swaps. Each of these insertion steps increases the size of the sorted section by one, allowing the algorithm to proceed with a new "first element of the unsorted section" until the entire list has been sorted.

Insertion sort has the same best/worst/average time complexity and the same space complexity as bubble sort, but it tends to be a bit faster in practice. Insertion sort is especially well suited to sorting small lists.

Selection Sort

Selection sort works quite similarly to insertion sort, keeping a sorted and unsorted portion of the list, and building up the sorted portion one element at a time. The difference with selection sort is that instead of taking the first element of the unsorted portion and inserting it at the proper spot, selection sort *selects* the smallest element of the unsorted portion on each pass, and puts it at the end of the sorted portion. This time, the entire list starts out as the unsorted portion instead of the first element being sorted–the starting element of the sorted portion has to be found from the list like every other element since elements don't move after being put in the sorted portion.

To highlight the difference: insertion sort picks a spot for the next element by searching through the sorted portion, selection sort picks an element for the next spot by searching through the unsorted portion.

Selection sort has identical time/space complexity to bubble and insertion sorts, and like insertion sort is faster than bubble sort. Still, insertion sort is usually preferred for small-data sorting.

Merge Sort

Merge sort is a more efficient algorithms than the three mentioned above. It works on the principle of Divide and Conquer, repeatedly breaking down a list into several sublists until each sublist consists of a single element, then repeatedly merging pairs of these sublists in a manner that results in a sorted list.

Unlike bubble, insertion, and selection sorts, merge sort is worst case *O(n log(n))*, so it scales much better to large lists. While there are ways to write an in-place merge sort, the typical space complexity is *O(n)*.

Quicksort

Quicksort is an advanced sorting algorithm which works differently from the others we've seen so far. Like merge sort, it is recursive, but for each step a "pivot" element from the list is selected, and elements to the left and right of the pivot are swapped as needed so that the list is partitioned into elements less than the pivot and elements greater than or equal to the pivot. Quicksort is then applied recursively to these partitions until the list is fully sorted.

Like merge sort, quicksort is average case *O(n log(n))*, but its worst case performance is *O(n^2)*. The performance of quicksort depends heavily on the method used for pivot selection, with the median-of-three pivot selection algorithm helping to avoid pitfalls common in more naive (e.g., random, first, last) selection techniques.

In practice, quicksort is still popular because it performs well on array-backed lists by exploiting optimizations for locality of reference. Merge sort may outperform it for very large data sets, and is usually preferred for linked lists. Both of these algorithms are significant improvements on the average case *O(n^2)* algorithms mentioned above.

Hybrid Sorting

While merge sort has a better runtime complexity than insertion sort, it has some overhead from not being an in-place sort, and insertion sort tends to be faster for sorting small amounts of data. This means that it is more efficient to combine the two algorithms into a hybrid sorting routine, where the recursive list partitions that merge sort creates are sorted by insertion sort instead of merge sort once they get small enough.

Project Details

"There's a term for people who don't read the project details : unemployed" -Dr. Owen

Overview

In this project, you will be implementing: the bubble, insertion, selection, and merge sort algorithms. We will provide the completed code for the quicksort algorithm for your reference. While you don't have any assignment relating to the quicksort code on this project, we recommend looking through it to familiarize yourself with that algorithm. The merge sort that you implement will be a hybrid merge sort which defers to insertion sort for handling small lists.

All of the sorting algorithms should accept a custom `comparator` argument which substitutes for `<` when comparing items to put in the list. If `comparator(a, b)` returns `True`, you should treat `a` as less than `b` for the sort and put it before `b` in the sorted list.

There is also an argument `descending` which defaults to `False`. If the `descending` argument is `True`, you should sort the list in descending order. Since you can sort the list in descending order by flipping the order of the inputs of the comparator and leaving the other logic the same, it might be helpful for you to write a *helper function*, perhaps called `do_comparison`, which takes elements `a` and `b`, the `comparator`, and `descending` as arguments, and tells you whether or not to put `a` before `b` in the sorted list. This helper function should only be a few lines!

It is important to note that ***the comparator means strictly `<` and not `<=`***, so for descending you should consider `comparator(b, a)` instead of `not comparator(a, b)`, since the second one would give you `a >= b` instead of `a > b`. If you did it the second way, your bubble sort might never stop swapping!

You can call the argument `comparator` the same as any other function, and the underlying function that gets called will be whatever function was passed in for this argument. This takes advantage of the fact that Python has what are called first-class functions, meaning functions can be stored and passed around the same way as any other type of value. The type of `comparator` is explained by this diagram:

Also note that some arguments will be specified after the pseudo-argument `*,`. The arguments following the asterisk `*` are "keyword-only" arguments. Keyword-only arguments are designed to prevent accidental miscalls that can occur with positional parameters.

```python
# Note the "argument" *, which some of the other arugments come after
def some_func(a, b, *, c, d):
    pass

# Ok
some_func(1, 2, c=3, d=4)

# will raise TypeError: some_func() takes 2 positional arguments but 4 were given
some_func(1, 2, 3, 4)
```

Assignment Specs

You will be given one file to edit, `solution.py`. You must complete and implement the following functions. Take note of the specified return values and input parameters.

Do not change the function signatures, including default values.

If you implement a function that passes the tests but does not use the specified sorting algorithm for that function, you will not get any points for that function.

Make sure to consult the lectures, Zybooks, and other resources available if you are not sure how a given sorting algorithm works. To earn manual points, you must also meet the required time and space complexity. Using the right algorithm will help!

solution.py:

- **selection_sort(data: List[T], *, comparator: Callable[[T,T], bool], descending: bool = False)**
    - Given a list of values, sort that list in-place using the selection sort algorithm and the provided comparator, and perform the sort in descending order if `descending` is `True`.
    - **param data**: List of items to be sorted
    - **param comparator**: A function which takes two arguments of type `T` and returns `True` when the first argument should be treated as less than the second argument.
    - **param descending**: Perform the sort in descending order when this is `True`. Defaults to `False`.
    - Time Complexity: *O(n^2)*
    - Space Complexity: *O(1)*
- **bubble_sort(data: List[T], *, comparator: Callable[[T,T], bool], descending: bool = False)**
    - Given a list of values, sort that list in-place using the bubble sort algorithm and the provided comparator, and perform the sort in descending order if `descending` is `True`.
    - **param data**: List of items to be sorted
    - **param comparator**: A function which takes two arguments of type `T` and returns `True` when the first argument should be treated as less than the second argument.
    - **param descending**: Perform the sort in descending order when this is `True`. Defaults to `False`.

- Time Complexity: *O(n^2)*
- Space Complexity: *O(1)*

- **insertion_sort(data: List[T], *, comparator: Callable[[T,T], bool], descending: bool = False)**
  - Given a list of values, sort that list in-place using the insertion sort algorithm and the provided comparator, and perform the sort in descending order if `descending` is `True`.
  - **param data**: List of items to be sorted
  - **param comparator**: A function which takes two arguments of type `T` and returns `True` when the first argument should be treated as less than the second argument.
  - **param descending**: Perform the sort in descending order when this is `True`. Defaults to `False`.
  - Time Complexity: *O(n^2)*
  - Space Complexity: *O(1)*

- **hybrid_merge_sort(data: List[T], *, threshold: int = 12, comparator: Callable[[T,T], bool], descending: bool = False)**
  - Given a list of values, sort that list in-place using a hybrid sort with the merge sort and insertion sort algorithms and the provided comparator, and perform the sort in descending order if `descending` is `True`. The function should use `insertion_sort` to sort lists once their size is less than or equal to `threshold`, and otherwise perform a merge sort.
  - **param data**: List of items to be sorted
  - **param threshold**: Maximum size at which insertion sort will be used instead of merge sort.
  - **param comparator**: A function which takes two arguments of type `T` and returns `True` when the first argument should be treated as less than the second argument.
  - **param descending**: Perform the sort in descending order when this is `True`. Defaults to `False`.
  - Time Complexity: *O(n log(n))*
  - Space Complexity: *O(n)*

Application

In the future, you have graduated this class and are working on a project. We definitely thought of a cool idea for what this project is so we could tell you in these assignment specs, but the project is top secret so we can't discuss that here! The important thing is that you have to sort a lot of data, and sometimes within that data you have to sort lists which are quite small, and sometimes lists which are really big. But you have to sort. A lot of sorting. Remember the project itself is top secret but bad things will happen if you cannot get huge amounts of data from an arbitrary order into a well-defined order. Very bad things.

You have no doubt solved many problems by sorting before. But what algorithm should you use to sort this data? Most of the time you should just use whatever sorting function your language provides, but where's the fun in that? After taking 331 you feel like you should know which algorithms are actually the best for this situation, and with the impending doom that will unfold if you do not get this data into a sorted order quickly you don't have time to play around. Unfortunately, you've also forgotten about the sorting unit.

You contact Onsay and your TAs from 331 asking what algorithms are the fastest, but it is Thursday night! The graph project is due tomorrow! Everyone is busy helping students finish their projects and keeping the course running smoothly, and unfortunately you can't inform them that 10 PM will never even happen if you can't get this data sorted soon (because the project is top secret).

So, you will have to figure it out for yourself...

For this application problem, you will compare the real-world, empirical performance of different sorting algorithms for different data sizes.

- **compare_times(algorithms: Dict[str, Callable[[List], None]], sizes: List[int], trials: int)**
  - Given a set of sorting algorithms and sizes, return the average times (in seconds) required to run each algorithm on a list of each size.
  - More specifically, for each input list size provided in `sizes`, your code should repeatedly call each algorithm in `algorithms` with a shuffled input list containing all numbers in 0 <= x < `size` exactly `trials` number of times, and the ***average*** time required to run the algorithm on this size should be saved. In other words, each algorithm should be run for the specified number of `trials` for each size in `sizes`, with the resulting times averaged together. For each trial, the input should be all of the numbers in 0 <= x < size, and the input should be put in a random order for each trial.
  - There are multiple timing functions in Python, time.perf_counter is good for performance benchmarks like this.
  - **param algorithms**: A dictionary whose values are callable sorting algorithm functions to consider, and whose keys are names for the algorithms.
  - **param sizes**: List of sizes of data to use in the benchmarks. Every size should be used for every algorithm.
  - **param trials**: Number of trials to run each size/algorithm combination for
  - **return**: A dictionary whose keys are the keys of `algorithms`, and whose values are lists of the average times it took to run those algorithms for the different sizes, in the same order as `sizes`.

Example

```
algorithms = {"bubble": bubble_sort, "insertion": insertion_sort}
sizes = [10, 20, 30]
result = compare_times(algorithms, sizes, 5)

# result should look like this, but the times are only for illustration
{
    "bubble": [
      1.013, # Time in seconds to run bubble sort for data of size 10, average of
5 trials
      2.102, # Time in seconds to run bubble sort for data of size 20, average of
5 trials,
      3.063, # Time in seconds to run bubble sort for data of size 30, average of
5 trials
    ],
    "insertion": [
      0.524, # Time in seconds to run insertion sort for data of size 10, average
of 5 trials
      1.067, # Time in seconds to run insertion sort for data of size 20, average
of 5 trials,
      1.623, # Time in seconds to run insertion sort for data of size 30, average
of 5 trials
    ],
}
```

Sorting Algorithm Comparison Graph

We have included a function `plot_time_comparison` in the starter code. Once you have implemented your sorts and finished the application problem, you can run this function to generate a graph based on the timing comparison output looking at all of your sorting functions. This will require matplotlib to be installed to generate the graph. The function can take 10+ seconds to run even on a fast computer, so don't be alarmed if it takes a while before you see the graph. You could always add a progress bar or print progress results in your `compare_times` function if you wanted to see the progress in realtime!

If you cannot install matplotlib or otherwise have issues with the `plot_time_comparison` function, feel free to comment it out by highlighting it in PyCharm and using the `CTRL + /` shortcut.

Submission

Deliverables

Be sure to upload the following deliverables in a .zip file to Mimir by 10:00p ET on Thursday, 2/24/22.

Your zipped folder can contain other files (for example, specs.md and tests.py), but must include at least the following:

```
|- Project04/
   |- __init__.py (to make sure Mimir can load test cases)
   |- solution.py (contains your solution source code)
   |- feedback.xml (project feedback, make sure to fill this out)
```

Please make sure to zip this folder before submitting.

Grading

The following 100-point rubric will be used to determine your grade on Project 4:

- Policies
  - ***Making all of these policies bold or italic would get too visually fatiguing but read them all because they're important!***
  - Using a different sorting algorithm than the one specified for some function will result in the loss of all automated and manual points for that function.
  - Not making the merge sort hybrid will result in the loss of half of all automated and manual points for that function.
  - You will not receive any points on this project if you use Python's built-in sorting functions or sorting functions imported from any library.
  - You will not receive any points on the project if you use any list-reversing function such as `reversed`, `list.reverse`, or a homemade alternative to these. You must sort the lists in ascending or descending order directly.
- Tests (70)

  - Test feedback.xml: __/3
  - Test coding standard: __/3
  - Sorts: __/50
    - Selection: __/11
      - test_selecton_sort_basic: __/2
      - test_selection_sort_comparator: __/3
      - test_selection_sort_descending: __/3
      - test_selection_sort_comprehensive: __/3
    - Bubble: __/11
      - test_bubble_sort_basic: __/2
      - test_bubble_sort_comparator: __/3
      - test_bubble_sort_descending: __/3
      - test_bubble_sort_comprehensive: __/3
    - Insertion: __/12
      - test_insertion_sort_basic: __/2
      - test_insertion_sort_comparator: __/3
      - test_insertion_sort_descending: __/3
      - test_insertion_sort_comprehensive: __/4
    - Hybrid Merge: __/16
      - test_hybrid_merge_sort_basic: __/2
      - test_hybrid_merge_sort_threshold: __/4
      - test_hybrid_merge_sort_comparator: __/3
      - test_hybrid_merge_sort_descending: __/3
      - test_hybrid_merge_sort_comprehensive: __/4
  - Application: __/14
    - test_compare_times_inputs: __/7
    - test_compare_times_timing: __/7

- Manual (30)

  - Time and space complexity points are all-or-nothing for each function. If you fail to meet time or space complexity in a given function, you do not receive manual points for that function.
  - Manual points for each function require passing all tests for that function, except for the comprehensive tests.
  - Time and space complexities:
    - selection_sort: __/7
    - bubble_sort: __/7
    - insertion_sort: __/7
    - hybrid_merge_sort: __/9

Tips, Tricks, and Notes

- There are different ways to implement merge sort, but make sure you are aiming for a solution that will fit the time complexity! If your recursive calls are some form of `hybrid_merge_sort(data[1:])`, this will not be *O(n log(n))*, as this does not divide the input list in half.

- A recursive implementation of merge sort will be the easiest to write. As you split the arrays, you should switch to insertion sort as soon as the split arrays get smaller than threshold. This means each of the recursive calls should be using the same threshold, such that the threshold is considered at each recursive call.
- Make sure to pass `comparator` and `descending` properly for all recursive calls as well.
- Using a helper function to do your comparisons that takes `descending` into account will make your code much easier to write. Look at the `do_comparison` stub that's provided in the starter code.
- Try these web applications to visualize sorting algorithms:
  - https://visualgo.net/bn/sorting
  - https://opendsa-server.cs.vt.edu/embed/mergesortAV (good merge sort visualization)
  - https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html
- You can see how long something took to run if you know how much the time changed while running it.
- random.shuffle is useful for putting numbers in a list in a random order.
- The application is not about coming up with an ingenious algorithm to minimize the complexity of solving the problem. Of course, don't do it less efficiently on purpose, but the biggest goal is to illustrate the efficiencies of the different algorithms and give a practical programming problem.
- We probably don't have to tell you this if you made it this far but make sure to read the specs including all grading requirements!

This project was created by Abhinay Devapatla and Zach Matson