Assignment 1

In Ho Han | Arman Jeevani | Luca Novello

Step 1 - Implementing Sorting Algorithms

#Bubble Sort

```
def bubble_sort(my_list):
    if (len(my_list) == 0):
        return 0
    elif (len(my_list) == 1):
        return my_list
    else:
        n = len(my_list)
        for i in range(n - 1):
            for j in range(n - 1 - i):
                if my_list[j] > my_list[j + 1]:
                    my_list[j], my_list[j + 1] = my_list[j + 1], my_list[j]
    return my_list
```

#Modified Version

```
def bubble_sort(my_list, key):
    if len(my_list) == 0:
        return []
    elif len(my_list) == 1:
        return my_list
    else:
        n = len(my_list)
        for i in range(n - 1):
           for j in range(n - 1 - i):
                if getattr(my_list[j], key)  > getattr(my_list[j + 1], key):
                    my_list[j], my_list[j + 1] = my_list[j + 1], my_list[j]
    return my_list
```

#Modified version for counting operation

```
def bubble_sort(my_list):
    if len(my_list) <= 1:
        return my_list, 0
    n = len(my_list)
    Tn = 0
    for i in range(n - 1):
        swapped = False
        for j in range(n - 1 - i):
            Tn += 1
            if my_list[j] > my_list[j + 1]:
                my_list[j], my_list[j + 1] = my_list[j + 1], my_list[j]
                Tn += 1
```

```
                swapped = True
        if not swapped:
            break
    return my_list, Tn
```

Explanation

In bubble sort, two for loops go through the entire array. First array starts from index 0 and the other array starts from index 1. This is to compare two consecutive numbers in each loop iteration. Let's say the first index is A and second index is B. Bubble sort compares A and B in each iteration. If A is higher than B, A is moved to B index and B is moved to A index. (if B is higher, then both stays in same index)

#Selection Sort

```
def selection_sort(my_list):
    if (len(my_list) == 0):
        return 0
    elif (len(my_list) == 1):
        return my_list
    n = len(my_list)
    for i in range(n - 1):
        min_idx = i
        for j in range(i + 1, n):
            if my_list[j] < my_list[min_idx]:
                min_idx = j
        if min_idx != i:
            my_list[min_idx], my_list[i] = my_list[i], my_list[min_idx]
```

#Modified version

```
def selection_sort(my_list, key):
    if len(my_list) == 0:
        return []
    elif len(my_list) == 1:
        return my_list
    n = len(my_list)
    for i in range(n - 1):
        if getattr(my_list[j], key) < getattr(my_list[min_idx], key):
            min_idx = j
    if min_idx != i:
        my_list[min_idx], my_list[i] = my_list[i], my_list[min_idx]
    return my_list
```

#Modified version for counting operation

```
def selection_sort(my_list):
    if len(my_list) <= 1:
        return my_list, 0
    n = len(my_list)
```

```
    Tn = 0
    for i in range(n - 1):
        min_idx = i
        for j in range(i + 1, n):
            Tn += 1
            if my_list[j] < my_list[min_idx]:
                min_idx = j
        if min_idx != i:
            my_list[min_idx], my_list[i] = my_list[i], my_list[min_idx]
            Tn += 1
    return my_list, Tn
```

Explanation

Selection sort's outer loop goes through each index of the array. In each iteration, the inner for loop goes through all indexes (except outer loop's current index) and finds the smallest value. Each time a smaller value is found, the minimum index is updated. Once inner loop reaches the end of the array, the function swaps the current index of the outer loop with minimum value. This process continues until the outer loop goes through the entire array.

#Insertion Sort

```
def insertion_sort(my_list):
    for i in range(1, len(my_list)):
        curr = my_list[i]
        j = i
        while j > 0 and my_list[j - 1] > curr:
            my_list[j] = my_list[j - 1]
            j -= 1
        my_list[j] = curr
```

#Modified Version

```
def insertion_sort(my_list, key):
    for i in range(1, len(my_list)):
        curr = my_list[i]
        j = i
        while j > 0 and getattr(my_list[j - 1], key) > getattr(curr, key):
            my_list[j] = my_list[j - 1]
            j -= 1
        my_list[j] = curr
    return my_list
```

#Modified version for counting operation

```
def insertion_sort(my_list):
    Tn = 0
    for i in range(1, len(my_list)):
```

```
        curr = my_list[i]
        j = i
        while j > 0:
            Tn += 1
            if my_list[j - 1] > curr:
                my_list[j] = my_list[j - 1]
                j -= 1
                Tn += 1
            else:
                break
        my_list[j] = curr
        Tn += 1
    return my_list, Tn
```

Explanation

In the insertion array, the function loops through the entire array starting from the first index. During each iteration, the current index is stored into a temporary variable. This is to compare with other index's value, but also to make the current index empty (for swapping purposes). Also in each iteration, the temporary variable is compared with all previous indexes within the inner loop. Inner for loop will compare all previous indexes of current index (which is stored in temp variable). If the previous index's value is greater than temp value, the previous index is taking over the original current index. If temp value is lower, temp value will be inserted to the previous index location. Inner loop process applies to all previous indexes, and once done, outer loop iteration increments.

#Quick Sort

```
def quick_sort(my_list):
      length = len(my_list)
      if length <= 1:
          return my_list
      else:
          pivot = my_list.pop()
      greater = []
      lower = []
      for item in my_list:
          if item > pivot:
              greater.append(item)
          else:
              lower.append(item)
      return quick_sort(lower) + [pivot] + quick_sort(greater)
```

#Modification

```
def quick_sort(my_list, key):
    length = len(my_list)
    if length <= 1:
        return my_list
    else:
```

```
        pivot = my_list.pop()
    greater = []
    lower = []
    for item in my_list:
        if getattr(item, key) > getattr(pivot, key):
            greater.append(item)
        else:
            lower.append(item)
    return quick_sort(lower, key) + [pivot] + quick_sort(greater, key)
```

#Modification version for countering operation

```
def quick_sort(my_list):
    Tn = 0
    if len(my_list) <= 1:
        return my_list, Tn
    pivot = my_list[-1]
    lower, greater = [], []
    for item in my_list[:-1]:
        Tn += 1
        if item > pivot:
            greater.append(item)
        else:
            lower.append(item)
    sorted_lower, lower_Tn = quick_sort(lower)
    sorted_greater, greater_Tn = quick_sort(greater)
    Tn += lower_Tn + greater_Tn
    return sorted_lower + [pivot] + sorted_greater, Tn
```

Explanation

Quick sort's key step is select pivot point, which is used to separate a given list into multiple pieces based on the following condition. In my solution, I have selected the last index of the array as pivot.
Once pivot is set, the function iterates through the list and checks if each array index's value is higher or lower than pivot. If value is higher, the index is added to 'greater', if not the index is added to lower. As last step, greater and lower will be sent to function itself by using recursion. Then returned greater and lower will be concatenated with the pivot to return sorted list.

Questions

1. Explain how each sorting algorithm works in your own words. What are the key steps involved in each?

Explanations are provided below function.

2. Compare and contrast the five sorting algorithms implemented. What are their fundamental differences?

Explanations are provided below function.

3. Which sorting algorithm would you choose for small datasets? Why?

- If I am using declared functions on step 1, I would use either insertion sort.
- If data set is already sorted, the insertion sort would only go through the list once and it will make O(n). Since insertion sort basically splits the array into two pieces and inserts the next array into the end of the sorted side of the array. And if the array is nearly sorted, this takes a lot less complexity.
- Selection sort is also a viable option even though it might take O(n^2) time complexity. If the data set is small and if data is nearly sorted, selection sort will be taking less steps compared to insertion sort.

4. Modify your implementation to sort a list of custom objects instead of integers. What changes were needed?

- Modified functions are listed below the original function above.
- First change required was adding a new parameter to the function since it requires the function to understand what kind of attribute is being compared to since the function must sort the object.
- Second change required was using getattr() function, so function use passed on key to retrieve data from object's key value.

Step 2 - Counting Steps (T(n))

Task

- Modify your sorting functions so that they:
  - Sort the received list.
  - Count the number of operations (T(n)) required to perform the sorting.
  - Return T(n) along with the sorted list.

Modified version for counting operation is listed below originals

- Testing:
  - Send a best-case scenario, a worst-case scenario, and an average-case scenario to each sorting function.
  - Record the output values for T(n).

All function uses same best case, worst case, and average case

1. Bubble sort
   a. Bestcase ⇒ [1,2,3,4,5] ⇒ Tn is 0 since swapping never occurred
   b. Worstcase ⇒ [5,4,3,2,1] ⇒ Tn is 20
   c. Avg case ⇒ [3,4,5,2,1] ⇒ Tn is 17
2. Selection sort
   a. Best case ⇒ 10
   b. Worst case ⇒ 14
   c. Average case ⇒ 13
3. Insertion sort
   a. Best case ⇒ 8
   b. Worst case ⇒ 24
   c. Average case ⇒ 20
4. Quick Sort
   a. Best case ⇒ 10

b. Worst case ⇒ 10
c. Average case ⇒ 10

Questions

1. In your implementation, how do you count the exact number of operations? What assumptions did you make?

I have implemented the ways to count whenever actual swapping or comparison is occurring. If my function implements a recursion method, the steps of each recursive call will be calculated since all functions return Tn. I have assumed that basic operation takes constant time. And I didn't count the memory allocation of using a new variable or array.

2. How does T(n) behave for the best, worst, and average cases in each sorting algorithm? Provide theoretical and empirical results.

Theory

Algorithm	Best case	Worst case	Average case
Bubble sort	O(n)	O(n^2)	O(n^2)
Selection sort	O(n^2)	O(n^2)	O(n^2)
Insertion sort	O(n)	O(n^2)	O(n^2)
Quick sort	O(nlogn)	O(n^2)	O(nlogn)

Actual Result

Algorithm	Best Input	Best	Worst Input	Worst	Average Input	Avg
Bubble sort	[1,2,3,4,5]	0	[5,4,3,2,1]	20	[3,1,4,5,2]	17
Selection sort	[1,2,3,4,5]	10	[5,4,3,2,1]	14	[3,1,4,5,2]	13
Insertion sort	[1,2,3,4,5]	8	[5,4,3,2,1]	24	[3,1,4,5,2]	20
Quick sort	[1,2,3,4,5]	10	[5,4,3,2,1]	10	[3,1,4,5,2]	10

3. How do the step counts (T(n)) you measured compared to the theoretical Big-O complexity of each algorithm? Are they aligned?

For bubble, selection, and insertion sort, result aligned since best case had least, worst had most, and average had steps count in between best- and worst-case scenario.

4. If two sorting algorithms have the same worst-case complexity, does that mean they will always take the same number of steps? Why or why not?

Even if two sorting algorithms have the same worst-case complexity, it doesn't mean that those two sorting algorithms will have the same number of steps taken. For example, looking at the graph above, even both bubble and selection sort had the same worst-case big o notation, insertion sort had more step count. This is because insertion sort requires comparing temporary variables with previous indexes and moving those indexes based on the value.

Step 3 - Analyzing T(n) Graphs

Task

- Use the sorting functions with the T(n) calculation feature to plot T(n) vs. n for different list sizes:
  - List sizes: **10, 50, 100, 500, 1000, 5000, 10000, 50000, 100000, 1,000,000, 10,000,000**
  - Use a **worst-case scenario** for your input lists.

N	Bubble Sort	Selection Sort	Insertion Sort	Quick Sort
10	90	54	55	45
50	2450	1274	1275	1225
100	9900	5049	5050	4950
500	249500	125249	125250	124750
1000	999000	500499	500500	499500
5000	24995000	12502499	12502500	12497500
10000	99990000	50004999	50005000	49995000
50000	249995000	1250024999	1250025000	1249975000
100000	9999900000	5000049999	5000050000	4999950000
1000000	999999000000	500000499999	500000500000	499999500000
10000000	99999990000000	50000004999999	50000005000000	49999995000000

Formulas:

- Bubble Sort: $Tn = n(n-1)$
- Selection Sort: $Tn = n(n-1)/2 + (n-1)$
- Insertion Sort: $Tn = n(n-1) + n = n^2$
- Quick Sort: $Tn = n(n-1)/2$

Questions

9. Describe the shape of the T(n) vs. n plots for each sorting algorithm. What patterns do you observe?

- All four algorithms produce $T(n) = n^2$, reflecting their worst-case $O(n^2)$.
- Bubble Sort: Almost twice the operations of others, this is the steepest parabola since $Tn = n(n-1)$.
- Selection Sort: $Tn = n^2/2 + n$, somewhat flatter than Bubble Sort, and quite steep.
- Insertion Sort: Visually nearly overlapping with Selection, but a little higher ($Tn = n^2$).
- Quick Sort: Because there are fewer operations per partition, $Tn = n(n-1)/2$ is the flattest of the four.
- Pattern: Quadratic growth is primary, with Bubble Sort continuously scoring best and Quick Sort lowest.

10. Which sorting algorithm exhibits the steepest growth in T(n)? What does this indicate about its efficiency?

The growth of Bubble Sort is the sharpest ($Tn = n(n-1) = n^2$). As a result of repetitive comparisons and swaps across all passes, even when parts are ignored (unlike Selection Sort's focused swaps or Insertion Sort's early breaks), it is the least efficient.

11. How does the graph confirm or contradict your expectations based on Big-O notation?

- The quadratic shapes align with $O(n^2)$ expectations for all algorithms in their worst cases:
- Bubble Sort: $O(n^2)$ matches $Tn = n(n-1)$.
- Selection Sort: $O(n^2)$ matches $Tn = n^2/2$.
- Insertion Sort: $O(n^2)$ matches $Tn = n^2$.
- Quick Sort: $O(n^2)$ matches $Tn = n(n-1)/2$, confirming the worst-case choice overrides $O(n \log n)$.

12. What would you expect to happen if you added an optimized merge sort or heapsort to the comparison? Would the T(n) vs. n behavior be different?

- Merge Sort: $O(n \log n)$ in all cases. T(n) would grow log-linearly (e.g., n=10,000: Tn = 140,000 vs. 50M+ here), producing a much flatter curve.
- Heapsort: Also $O(n \log n)$, with $Tn = n \log n + n$ (heap build), similar log-linear growth but with a slightly higher constant than Merge Sort.
- Difference: Both would starkly contrast the quadratic growth here, remaining practical for large n (e.g., n=1M: Tn = 20M vs. 500B+), highlighting their superior scalability.

Step 4 - Timing Analysis

Task

- Use Python's time library to measure the execution time of each sorting algorithm.
- Run the sorting functions on the same list sizes as in Step 3.
- Plot **algorithm completion time vs. n** using the worst-case scenario input lists.
- Compare the actual execution time with the T(n) results.

```
import time
import random
import matplotlib.pyplot as plt
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]

def selection_sort(arr):
    n = len(arr)
    for i in range(n):
        min_idx = i
        for j in range(i+1, n):
            if arr[j] < arr[min_idx]:
                min_idx = j
```

```
        arr[i], arr[min_idx] = arr[min_idx], arr[i]

def insertion_sort(arr):
    n = len(arr)
    for i in range(1, n):
        key = arr[i]
        j = i-1
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key

def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quick_sort(left) + middle + quick_sort(right)

def measure_time(sort_function, arr):
    start_time = time.perf_counter()
    sort_function(arr)
    end_time = time.perf_counter()
    return end_time - start_time

sizes = [10, 50, 100, 500, 1000, 5000, 10000, 50000, 100000, 1000000, 10000000]
sorting_algorithms = [bubble_sort, selection_sort, insertion_sort, quick_sort]
algorithm_names = ["Bubble Sort", "Selection Sort", "Insertion Sort", "Quick Sort"]

execution_times = {name: [] for name in algorithm_names}

for size in sizes:
    worst_case_list = list(range(size, 0, -1))
    for name, algo in zip(algorithm_names, sorting_algorithms):
        test_list = worst_case_list.copy()
        execution_time = measure_time(algo, test_list)
        execution_times[name].append(execution_time)

plt.figure(figsize=(10, 6))
for name in algorithm_names:
    plt.plot(sizes, execution_times[name], label=name, marker='o')

plt.xlabel("Input Size (n)")
plt.ylabel("Execution Time (seconds)")
plt.title("Sorting Algorithm Execution Time vs. Input Size")
plt.legend()
plt.xscale("log")
plt.yscale("log")
plt.grid()
plt.show()
```

Questions

13. Compare the runtime graphs (completion time vs. n) with the T(n) graphs. Do they follow similar trends? Why or why not?

Yes, the runtime graphs closely follow the T(n) trends:

- Bubble Sort, Selection Sort, and Insertion Sort show quadratic growth ($O(n^2)$).
- QuickSort shows faster execution for most cases (O(n log n) on average).
- Large datasets make the differences more noticeable.

14. Are there any discrepancies between the T(n) and actual runtime measurements? What factors might contribute to these differences?

Yes, there are small differences due to:

1. System Overhead – CPU scheduling and background processes.
2. Python's Interpreter – Slower than compiled languages like C++.
3. Cache & Memory Access – Algorithms that access memory efficiently (like QuickSort) perform better.
4. Built-in Optimizations – Python handles list operations efficiently, affecting runtime.

15. For larger datasets (e.g., 1,000,000 elements), which algorithm performs the worst in actual execution time? Is this expected?

- Bubble Sort is the worst because of its $O(n^2)$ complexity.
- It takes exponentially more time compared to QuickSort (O(n log n)).
- Selection and Insertion Sort are slightly better but still slow.

16. Why does quicksort perform better on average, even though its worst-case complexity is $O(n^2)$?

QuickSort typically runs in O(n log n) because:

- The pivot divides the list efficiently.
- Recursive calls reduce problem size quickly.
- Memory locality improves performance.

Worst-case ($O(n^2)$) only happens with bad pivot choices, which can be avoided using randomized pivots.

17. What role does Python's built-in sorting algorithm (sorted()) play in real-world applications? How does it compare to the sorting algorithms implemented in this assignment?

Python's sorted() uses Timsort, a mix of Merge Sort (O(n log n)) and Insertion Sort (O(n)). It outperforms most manual sorting implementations for real-world use.

- Optimized for nearly sorted data.
- Uses memory efficiently.
- Handles large datasets better.

In real-world applications, Timsort is preferred over QuickSort, Merge Sort, or Bubble Sort.

General Reflection and Optimization

Questions

18. If you had to optimize one of your sorting algorithms, which one would it be and why?

I would optimize Bubble Sort because it is the least efficient of all the sorting algorithms I implemented. Bubble Sort has a time complexity of $O(n^2)$, making it extremely slow for large datasets.

19. How does memory usage compare for different sorting algorithms? Would memory constraints affect your choice of sorting algorithm?

Sorting algorithms differ in memory usage: Bubble, Selection, and Insertion Sort use O(1) space, QuickSort uses O(log n) space, and Merge Sort requires O(n) extra memory, making it less ideal for large datasets. If memory is a concern, I'd use an in-place algorithm like QuickSort or Insertion Sort instead of Merge Sort.

20. Research an advanced sorting algorithm (e.g., radix sort, merge sort). How does it compare to the ones you implemented in terms of T(n) and runtime?

Merge Sort runs in O(n log n) for all cases, making it much faster than Bubble, Selection, or Insertion Sort ($O(n^2)$), but it requires O(n) extra space. It's stable and works well for linked lists, but QuickSort is usually preferred since it's faster on average and works in-place with lower memory usage.