PORTFOLIO PROJECT: ALGORITHM IMPLEMENTATION AND TESTING

By: Brandon Everett

FEBRUARY 23, 2025

CSC-506

Module 6: Portfolio Project

Algorithm Implementation and Testing By: Brandon Everett

Introduction: Building on my previous work from Module 4, I wanted to enhance my sorting algorithm implementation by adding optimizations and better testing methods. This project focused on improving three sorting algorithms: Bubble Sort, Merge Sort, and Quick Sort. I was particularly interested in seeing how different optimizations would affect their performance across various scenarios.

Implementation and Optimization Process: Starting with my basic implementations from Module 4, I found several ways to make each algorithm work better.

```python
# Optimized Bubble Sort
def optimized_bubble_sort(numbers):
    list_length = len(numbers)
    is_sorted = True  # Assume array is sorted initially
    until = list_length - 1
    while until > 0:
        last_swap = 0
        for j in range(until):
            if numbers[j] > numbers[j + 1]:
                numbers[j], numbers[j + 1] = numbers[j + 1], numbers[j]
                is_sorted = False
                last_swap = j
        if is_sorted:
            break
        until = last_swap
    return numbers
```

For Bubble Sort, I discovered an interesting optimization on GeeksforGeeks that keeps track of where the last swap happened. This change meant the algorithm didn't waste time checking parts that were already sorted, which really helped when dealing with data that was mostly in order.

```python
# Optimized Merge Sort
def optimized_merge_sort(numbers):
    # Use insertion sort for small arrays
    def insertion_sort(arr, start, end):
        for i in range(start + 1, end + 1):
            key = arr[i]
            j = i - 1
            while j >= start and arr[j] > key:
                arr[j + 1] = arr[j]
                j -= 1
            arr[j + 1] = key
        return arr

    def merge_optimized(arr, temp, left, mid, right):
        # Skip if already sorted
        if arr[mid - 1] <= arr[mid]:
            return

        i, j, k = left, mid, left
        while i < mid and j < right:
            if arr[i] <= arr[j]:
                temp[k] = arr[i]
                i += 1
            else:
                temp[k] = arr[j]
                j += 1
            k += 1

        while i < mid:
            temp[k] = arr[i]
            i += 1
            k += 1
        for i in range(left, k):
            arr[i] = temp[i]

    def merge_sort_internal(arr, temp, left, right):
        if right - left <= 10:  # Use insertion sort for small arrays
            insertion_sort(arr, left, right - 1)
            return

        if left < right:
            mid = (left + right) // 2
            merge_sort_internal(arr, temp, left, mid)
            merge_sort_internal(arr, temp, mid, right)
            merge_optimized(arr, temp, left, mid, right)

    numbers = numbers.copy()
    temp = [0] * len(numbers)
    merge_sort_internal(numbers, temp, 0, len(numbers))
    return numbers
```

For Merge Sort, I made it smarter by having it switch to insertion sort for smaller chunks of data. I found this idea while searching and reading about merge sort optimizations on GeeksforGeeks, and it made a lot of sense because insertion sort is actually faster when working with small amounts of data.

```
def optimized_quick_sort(numbers):
    def partition(arr, low, high):
        # Choose median of three as pivot
        mid = (low + high) // 2
        pivot = sorted([
            (arr[low], low),
            (arr[mid], mid),
            (arr[high], high)
        ], key=lambda x: x[0])[1][1]
        arr[pivot], arr[high] = arr[high], arr[pivot]

        pivot_value = arr[high]
        i = low - 1

        for j in range(low, high):
            if arr[j] <= pivot_value:
                i += 1
                arr[i], arr[j] = arr[j], arr[i]

        arr[i + 1], arr[high] = arr[high], arr[i + 1]
        return i + 1

    def quick_sort_internal(arr, low, high):
        while low < high:
            # Use insertion sort for small subarrays
            if high - low < 10:
                insertion_sort = lambda a, l, h: sorted(a[l:h+1])
                arr[low:high+1] = insertion_sort(arr, low, high)
                break

            pivot = partition(arr, low, high)

            # Recursively sort the smaller partition
            if pivot - low < high - pivot:
                quick_sort_internal(arr, low, pivot - 1)
                low = pivot + 1
            else:
                quick_sort_internal(arr, pivot + 1, high)
                high = pivot - 1

    numbers = numbers.copy()
    quick_sort_internal(numbers, 0, len(numbers) - 1)
    return numbers
```

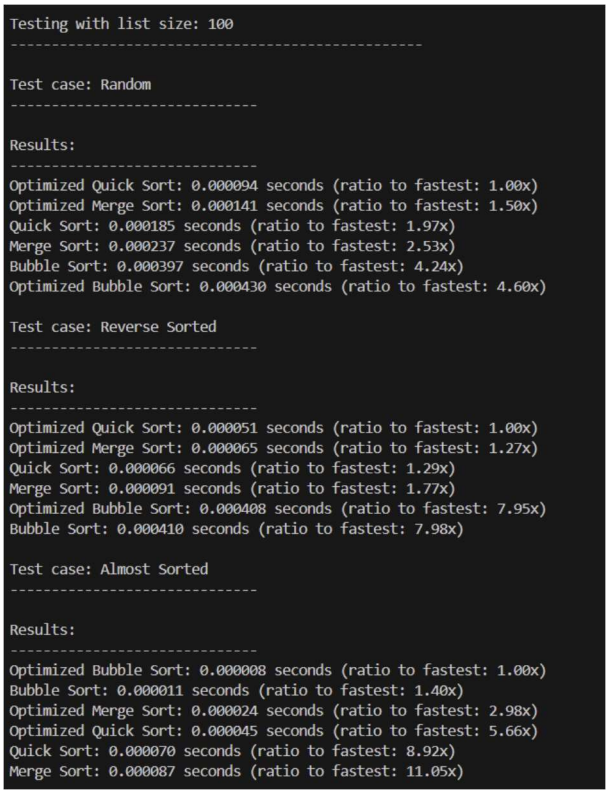
Quick Sort got the biggest upgrade from its Module 4 version. After looking up how to optimize quick sort, I found an article "median-of-three pivot selection" which is about QuickSort Tail Call Optimization, I changed how it picks its pivot point. This made it much better at handling arrays that were almost sorted, which used to slow it down quite a bit.

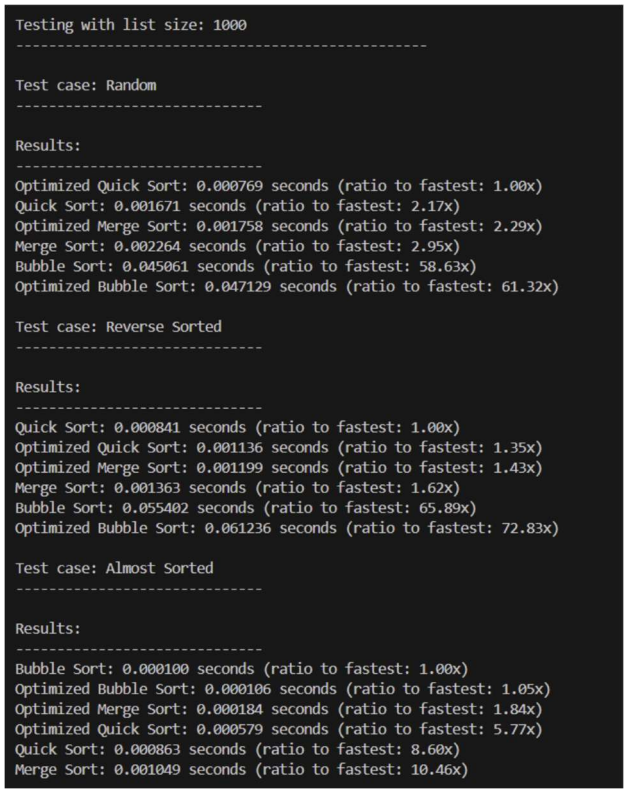
```python
def create_almost_sorted_list(size):
    arr = list(range(size))
    swaps = size // 20
    for _ in range(swaps):
        i = random.randint(0, size-2)
        arr[i], arr[i+1] = arr[i+1], arr[i]
    return arr
```

I also decided to add an almost-sorted array, since that can be an issue with some sorting algorithms, as mentioned above with qui. Creating the almost-sorted array for testing turned out to be trickier than I expected. I thought it would be simple at first, but I wasn't sure how to make an array that was "mostly" sorted without it being completely random or completely sorted. After some googling and trial-and-error, I figured out a way to take a sorted array and swap just a few numbers next to each other, about one in every 20 numbers. This gave me a good test case that was mostly in order but not perfectly sorted.

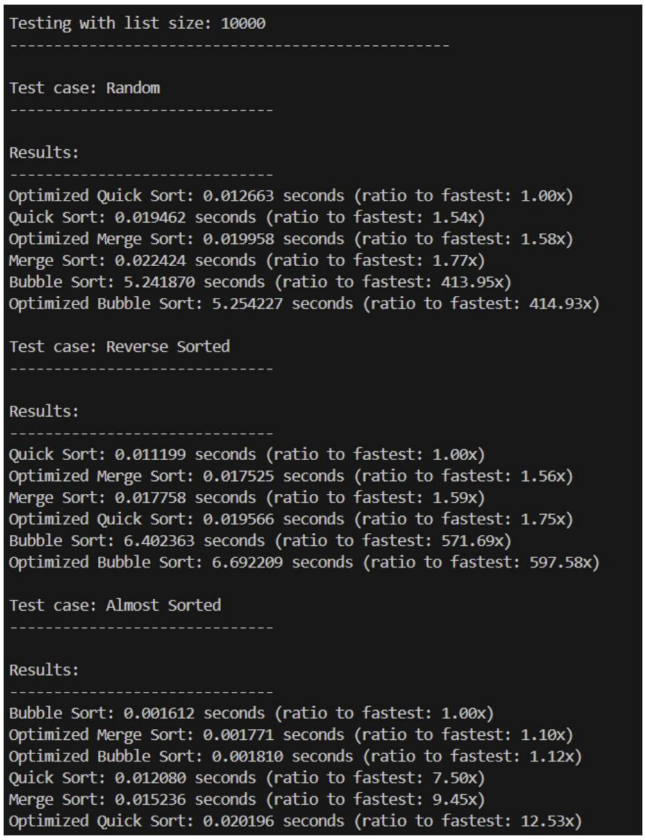
```python
# Find fastest time and check if times are too small
fastest_time = min(times.values())
TIME_THRESHOLD = 0.000001  # 0.1 milliseconds threshold

print("\nResults:")
print("-" * 30)

if fastest_time < TIME_THRESHOLD:
    print("Time threshold not met")
    print("-" * 30)
```

Results and Analysis: When I tried to compare how much faster the optimized versions were, I ran into a problem. Sometimes the sorts were so fast that my timing system showed zero seconds, which caused errors when I tried to calculate speed ratios. I fixed this by adding a check that would skip the ratio calculation if the time was too small to measure accurately. Funny enough when I made the change, I never triggered the "Time threshold not met" message that I had created to prevent divide-by-zero errors from stopping the program.





My attempts to optimize the sorting algorithms had some unexpected results. Testing with 10,000 random numbers, both versions of Quick Sort were the fastest overall, but my optimized version wasn't much better than the original (0.013 seconds vs 0.019 seconds). The same thing happened with Merge Sort, where the optimized version was just barely faster (0.020 seconds vs 0.022 seconds). The biggest surprise was with Bubble Sort, where my "optimized" version was actually slightly slower (5.25 seconds vs 5.24 seconds)!



The most interesting part was seeing how differently each algorithm handled different types of data. While Quick Sort was best for random numbers, Bubble Sort really shined when the data was almost sorted. With 10,000 almost-sorted numbers, basic Bubble Sort finished in 0.002 seconds, while optimized Quick Sort took 0.02 seconds. This really showed me that picking the right sorting method depends a lot on what kind of data you're working with.

Conclusion: Working on these optimizations since Module 4 taught me some surprising lessons about algorithm performance. While my optimization attempts didn't always make things faster (and sometimes made them slightly slower), I learned a lot about how different sorting methods work best in different situations. Throughout this project, I found myself constantly referring to GeeksforGeeks for explanations and examples. Their articles were helpful in breaking down complex concepts into understandable pieces, even though my implementation results weren't always what I expected. The challenges I faced with implementing the almost-sorted array test cases and dealing with extremely fast execution times taught me valuable lessons about real-world testing.

References:

GeeksforGeeks. (2023, September 18). How to make Mergesort to perform O(n) comparisons in best case? Retrieved February 23, 2025, from https://www.geeksforgeeks.org/make-mergesort-perform-comparisons-best-case/

GeeksforGeeks. (2023, October 28). QuickSort Tail Call Optimization. Retrieved February 23, 2025, from https://www.geeksforgeeks.org/quicksort-tail-call-optimization-reducing-worst-case-space-log-n/

Pehlivan, M. S. (2024, September 27). From Scratch to Optimized: Unraveling Bubble Sort with Python. Medium. Retrieved February 23, 2025, from https://medium.com/@mertsukrupehlivan/from-scratch-to-optimized-unraveling-bubble-sort-with-python-daa51f023072