Assignment 2

Implementing Sorting Algorithms in C and Creating a Shared Library for Python with Case Study

Pedram Pasandide

Due Date: 2024, 13 Oct

1 Introduction to Sorting Algorithms

Sorting algorithms are fundamental algorithms in computer science that rearrange the elements of a list or array into a specified order, typically ascending or descending. Various sorting algorithms have been developed, each with its unique methodology and characteristics. In this assignment, we will focus on five specific sorting algorithms:

Bubble Sort: A simple comparison-based algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The process is repeated until the list is sorted.

Insertion Sort: This algorithm builds a sorted array one element at a time. It takes each element from the unsorted part and places it in the correct position in the sorted part, making it efficient for small datasets.

Merge Sort: A divide-and-conquer algorithm that divides the input array into two halves, recursively sorts each half, and then merges the sorted halves back together. It is known for its efficiency and stability.

Heap Sort: This algorithm uses a binary heap data structure to create a sorted array. It involves building a max heap from the input data and then repeatedly extracting the maximum element from the heap and rebuilding it until the array is sorted.

Counting Sort: A non-comparison-based algorithm that is efficient for sorting integers within a specific range. It counts the occurrences of each distinct value in the input array, then calculates the position of each value in the sorted output array.

You can find more sorting algorithms in Section Appendix: Additional Sorting Algorithms.

2 Problem Statement (6 points)

In this assignment, you are required to implement the following sorting algorithms in C from scratch:

- Bubble Sort
- Insertion Sort
- Merge Sort
- Heap Sort
- Counting Sort

You will be provided with a structure for your implementation, including files for your main program, header file, and implementation file. Your goal is to complete the implementations of these sorting algorithms, ensuring they work correctly before proceeding to the next part of the assignment.

2.1 File Structure

- `main.c`: This file will contain your test cases for each sorting algorithm. You will include necessary libraries, utility functions, and the `main()` function to execute the sorting tests.
- `mySort.h`: This header file contains function prototypes for each sorting algorithm. **IMPORTANT: DO NOT CHANGE ANYTHING HERE.** You will NOT submit this file, as the instructor will use the same version.
- `mySort.c`: You are required to implement all the sorting algorithms in this file. Ensure you write the functions and algorithms from scratch. You will submit this file, which holds the actual implementations of the sorting functions.

The code for Bubble Sort is already provided to help you get started. Make sure to run the test cases in C to verify that your implementations are working correctly before moving on to the Python integration. Please, do not modify anything in `mySort.h`; the actual implementations of the functions will be in `mySort.c`.

2.2 How to Create a Shared Library

Once you have successfully implemented the sorting algorithms, you can create a shared library to be used in Python. To do this, you will use the GNU Compiler Collection (GCC) with specific compiler flags. Here's a command to compile your C code and create a shared library:

```
gcc -O3 -shared -o libmysort.so -fPIC mySort.c
```

Compiler Flags:

- `-shared`: This flag tells GCC to create a shared library.
- `-o libmysort.so`: This specifies the output file name of the shared library.
- `-fPIC`: This stands for "Position Independent Code," which is necessary for creating shared libraries.
- `-O3`: This flag is not necessary to create a shared library, it just makes your code work faster. We will talk about in our lecture notes in a section Effective Code Development Practices.

If you do not have Python working on your operating system, you can use Google Colab. You can upload your shared library to Google Drive and mount the library to use it in your Colab environment. For more information on how to do this, refer to the example codes and comments provided in the `mySort_test.ipynb` file, which demonstrates how to test the Bubble Sort implementation.

3 Space and Time Complexity with Case Study (4 points)

In this section, you will evaluate the performance of your sorting functions by testing them on two different arrays: a small array and a large random array. Follow the instructions below and comment in the `mySort_test.ipynb` to complete the comparison.

Step 1: First, test your sorting functions on a small array to ensure they work correctly before moving on to larger test cases. Use the following array:

```
arr0 = np.array([64, -134, -5, 0, 25, 12, 22, 11, 90], dtype=np.int32)
```

This will allow you to verify that the sorting algorithms are correctly implemented. Run your sorting functions (e.g., `bubbleSort`, `insertionSort`, etc.) on this array and print the results.

Step 2: Next, create a large random array using the following command:

```
arr = np.random.choice(np.arange(-1000000, 1000000, dtype=np.int32),
    size=500000, replace=False)
```

Explanation of `np.random.choice()`:

- The first argument `np.arange(-1000000, 1000000, dtype=np.int32)` creates a 1D array of integers from -1,000,000 to 1,000,000.
- The `size=500000` argument specifies that the array should have 500,000 elements.
- The `replace=False` argument ensures that no value is repeated in the randomly generated array.

Step 3: Run your sorting functions (e.g., `bubbleSort`, `insertionSort`, `mergeSort`, `heapSort`, `countingSort`) to sort the large array `arr`. Record the time it takes to sort the array and observe the performance. Use the same code structure for `bubbleSort` in `mySort_test.ipynb` to run and time your functions.

Step 4: Python provides two built-in functions to sort arrays:

- `sorted()`: A built-in Python function that works on any iterable.
- `numpy.sort()`: A NumPy function optimized for sorting arrays.

To compare your C implementations with these built-in functions, use the following code to time them:

```
start = time.time()
sorted_arr = sorted(arr)
end = time.time()
print(f"Built-in sorted() Time: {end - start} seconds")

start = time.time()
np_sorted_arr = np.sort(arr)
end = time.time()
print(f"NumPy sort() Time: {end - start} seconds")
```

Step 5: After running all sorting functions (both your C implementations and Python's built-in functions), fill out the table below by recording the time taken for each function to sort the large array `arr`.

Table 1: Sorting Method Time and Space Complexity with CPU time in Python

Sort Method	Complexity		CPU time (Sec)
	Time	Space	
Bubble	$O(n^2)$	$O(1)$	645.14
Insertion			
Merge			
Heap			
Counting			
Built-in `sorted()`			
`numpy.sort()`			

Make sure to compare the results and analyze the performance of each algorithm, especially in terms of the time complexity for large inputs. **Which function is faster?**

4 Report and Makefile (3 points)

In your `report.pdf`, explain the algorithms you have implemented, how to compile and run your program, fill out the given table after running the test cases on Python. Create a section named **Appendix** in your report and include all your codes in this section with a text based format (not screenshots). The `Makefile` must create **ONLY** the shared library named `libmysort.so`.

5 Submission On Avenue to Learn

Submit on Avenue:

1. `main.c`
2. `mySort.c`
3. `mySort_test.ipynb`
4. `report.pdf`
5. `Makefile`

Do NOT submit `mySort.h` as I will the original version. Please do **NOT** submit any zip files.

the source code `sqrtUser.c` and your report named `README.pdf`.

6 Appendix: Additional Sorting Algorithms

This section is optional to study. In addition to the primary sorting algorithms covered in this assignment, there are several other important sorting algorithms that are worth exploring. Each algorithm has its unique approach and characteristics, making them suitable for various applications. Below are a few notable algorithms:

Shell Sort: An optimization of insertion sort that allows the exchange of items that are far apart. The algorithm starts by sorting pairs of elements far apart and progressively reducing the gap between the elements to be compared. This results in a more efficient sorting process than simple insertion sort.

Quick Sort: A highly efficient sorting algorithm that employs a divide-and-conquer strategy. It selects a 'pivot' element from the array and partitions the other elements into two sub-arrays according to whether they are less than or greater than the pivot. The sub-arrays are then sorted recursively.

Radix Sort: A non-comparison-based sorting algorithm that sorts integers by processing individual digits. It groups numbers by their individual digits starting from the least significant to the most significant digit. Radix sort is particularly effective when sorting large sets of integers.

Bucket Sort: An algorithm that distributes elements into a number of buckets. Each bucket is then sorted individually, either using a different sorting algorithm or recursively applying the bucket sort. This algorithm works well when the input is uniformly distributed over a range.

Tim Sort: A hybrid sorting algorithm derived from merge sort and insertion sort. It is the default sorting algorithm in Python's built-in sort() method and is optimized for real-world data, making it efficient for practical use cases.

Comb Sort: A generalization of bubble sort that improves the efficiency by eliminating small values near the end of the list. It uses a gap sequence to compare and swap elements, which helps reduce the number of comparisons and swaps needed.

Gnome Sort: A simple sorting algorithm that is similar to insertion sort. It iterates through the array, swapping elements that are out of order, and moving left when it has to swap. If it reaches the beginning of the array, it starts again from the next position.

Pigeonhole Sort: This sorting algorithm is suitable for sorting integers within a small range. It creates an array of "pigeonholes" to hold the elements, then places each element in its corresponding hole. Finally, it collects the elements from the pigeonholes in order.

Sleep Sort: A humorous sorting algorithm that leverages multithreading. Each element is placed in a separate thread that sleeps for a duration proportional to its value, waking up when the time expires. It produces a sorted output based on the order of completion of the threads.

These algorithms can serve as a foundation for further study in sorting techniques and performance optimization. Understanding the principles behind these algorithms will enhance your problem-solving skills and prepare you for more advanced topics in computer science.