2MP3 Assignment 2

Garret Langlois

October 2024

1 Introduction

Sorting algorithms are useful and fundamental systems which form the basis of many complex systems seen in computing. The general goal of sorting algorithms is to organize a list of elements from lowest to highest in the fastest time possible. Many different algorithms have been created to ensure sorting in the most efficient time complexities possible. However, time complexity is not the only consideration, ease of implementation and memory complexity are also large factors when considering which sorting algorithm is best to use for a given algorithm. This report will discuss five different sorting algorithms: **Bubble Sort**, **Insertion Sort**, **Merge Sort**, **Heap Sort**, **Counting Sort**. Bubble sort is already implemented as an example in the provided code.
The file structure of this assignment relies on three main files: **main.c**, **mySort.h**, **mySort.c**. The **main.c** file contains **two test arrays** and function calls for all of the different algorithms. This can be run directly to testing the sorting capabilities of all of the different algorithms. The **mySort.h** file contains function prototypes which are given to individual marking this assignment. The **mySort.c** file contains the actual implementations for all of the different algorithms.
The final aspect of the assignment to mention is the python code that is required to time the code to determine its efficiency versus other methods that are tested.

2 Problem Statement

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.

3 Solution

The implementation of all of the different sorting algorithms that are used in the **mySort.c** file are described in their own sections below along with the code that was required to create them

3.1 Bubble Sort

The Bubble Sort algorithm for sorting a list is one of the simplest methods for doing so. To describe how the algorithm sorts a list I will walk through the steps the algorithm uses:

1. Start at the firs element and compare it with the next element in the array.
2. Compare the firs element to the second and if it is greater, swap them.
3. Move on to the next element in the array compare it again.
4. Once the entire array has been iterated through the largest element is moved to the correct position at the end of the list. This is why the algorithm is called bubble sort, because it bubbles the element to the top.
5. The process is then repeated for the rest of the elements in the list. This requires nested for loops and therefore it is quite slow.

To execute this algorithm the following code was included with the files for the assignment, as Bubble Sort was given as an example algorithm:

```c
void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1])
                swap(&arr[j], &arr[j + 1]);
        }
    }
}
```

Here the nested for loops that were previously discussed can be seen.

3.2 Insertion Sort

The Insertion sort algorithm is a simple method for sorting an array. It works by iterating through the elements in the list one at a time and grabbing unsorted elements and putting them in the correct position in the list. Insertion Sort follows these steps to sort an array:

1. The second element is compared to the one before it.
2. If the second element is smaller than the first it shifts the larger element to the right.
3. If the current index is further along the array the we continue comparing to elements before it until we find the right position for the element.
4. We shift the rest of the elements to the right and insert the element at the correct location.
5. Continue this process for every element in the array.

To implement this method in C it takes some more code than what was required for the bubble sort algorithm. The code that I wrote to develop this algorithm can seen below:

```
void insertionSort(int arr[], int n) {

    //Assume the elements in the array prior to the current element are already sorted
    //Start at the first element of the array and loop through all of the elements after that
   for (int i = 1; i < n; i++) {

       //This will grab the value of the element we want to place in the correct position
       int shiftValue = arr[i];

       //We want to use j later so we define it outside of the loop
       int j;

       //We loop through all of the elements in the list
       for (j = i - 1; j >= 0; j--) {

           //If the current element is greater than the value we are looking at we shift the values that are
           //greater than it in the array one to the right
           if(arr[j] > shiftValue) {
               arr[j+1] = arr[j];
           }
           //Once we have found a place where the value is not greater than it the correct location is found
           else {
               break;
           }

```

```
        }

        //Ran into a segmentation here many times because I tried arr[j] instead of arr[j+1] which would result in the indexing
        //at a negative element
        //This places the shiftValue at the correct position, one position ahead of the last position we checked.
        arr[j + 1] = shiftValue;

    }
}
```

The bulk of the lines in the rest of the algorithms will be used for comments as I believe it is valuable to explain what every line does for the algorithm.

3.3 Merge Sort

From afar the Merge Sort algorithm could be confused with the Binary Search method as it is bears many similarities to it, being that the array is split down the middle into two halves recursively. The Merge Sort sorting algorithm is a highly efficient sorting algorithm and is great for large arrays. It works by using the following steps:

1. Recursively split the lists in half until each list is of size 2 or size 1.
2. Merge the arrays back together, one half at a time, making sure that the "subarray" is sorted
3. Eventually the entire array will be reconstructed and it will be sorted.

The code required to do this is recursive and as such may run into Stack Overflow issues depending on how deep the recursion needs to go. In saying this, even with the 500,000 length array I never had an issue with my implementation. The following code is my implementation of the Merge Sort algorithm:

```
void mergeElements(int arr[], int leftPosition, int rightPosition) {

    //It is important to do this instead of n/2 because the position in the original array dynamically
    //changes throughout the recursion
    int middlePosition = leftPosition + (rightPosition-leftPosition)/2;

    int leftSize = middlePosition - leftPosition + 1;
    int rightSize = rightPosition - middlePosition;

    //Now we need two different arrays which we will use to temporarly store the data

    int *leftArray = (int *)malloc(leftSize * sizeof(int));
    int *rightArray = (int *)malloc(rightSize * sizeof(int));

```

```
    //Append the values of the array from the left side to the temporary array
    for (int i = 0; i < leftSize; i++) {
        leftArray[i] = arr[leftPosition + i ];
    }

    //Append the values of the array from the right side to the temporary array
    for (int i = 0; i < rightSize; i++) {
        rightArray[i] = arr[1 + middlePosition + i];
    }

    //Had to ask ChatGPT how to get an extra two indexing variables.
    //It ended up being only these two extra lines that I needed.
    //The reason that they are both set to 0 is because i was already used
    int i = 0; //This is the indexing variable for the left array
    int j = 0; //This is the indexing variable for the right array
    int k; //This is the whole array index
    int iterationNum = (rightSize+leftSize)/2 + 1;

    //Now we need to add the values into the original array, which remember is still in the recursive loop
    //so it isnt actually the original array we were sorting
    // Total number of elements in the range to merge is from leftPosition to rightPosition
    for (k = leftPosition; k <= rightPosition; k++) {
        if (i < leftSize && (j >= rightSize || leftArray[i] <= rightArray[j])) {
            arr[k] = leftArray[i];
            i++; // Move index in leftArray
        } else {
            arr[k] = rightArray[j];
            j++; // Move index in rightArray
        }
    }


    //Now we just need to copy over any remaining elements that were not included in the loop.
    //the reason that this happens is because of the bound we set with the division

    //This copy strategy was taken from this article: https://www.geeksforgeeks.org/c-program-for-merge-sort/

    while (i < leftSize) {
        arr[k] = leftArray[i];
        i++;
        k++;
    }

    while (j < rightSize) {
        arr[k] = rightArray[j];
        j++;
```

```
        k++;
    }

    //Free the allocated memory for the left array and right array
    free(leftArray);
    free(rightArray);

}

//These are the sorting algorithms
void mergeSort(int arr[], int l, int r) {

    //This ensures that the size of the elements that are being merged are greater than 0
    if (l < r) {

        //This defines the middle value for the array
        //The reason that single letter variable names are being used is because that is what is used in the header for this function
        int m = l + (r-l)/2;

        //We run the mergeSort recursively for the two different arrays, the left one and the right one
        //Remember they will not run if the dimenions that it is expecting is less than one
        mergeSort(arr, l, m);

        mergeSort(arr, m+1,r);

        //This function contains the main logic for the algorithm, it swaps the elements in the array depending on what order they are in
        mergeElements(arr, l , r);

    }
}
```

3.4 Heap Sort

Heap sort is the most unique of all the sorting algorithms as it uses something called a heap. This can be though of as a tree where the top element is the parent element element and subsequent elements are the children of that parent element.

The key concept to know for this sorting algorithm is the max heap. We build the max heap at the start of the algorithm by arranging the tree, which in our case is represented as an array, has the largest element at the top and the smaller elements as branches of the tree below. Once we build the max heap we iteratively shrink the heap, moving elements one at a time into the sorted positions in the array. Here are the steps the algorithm follows to do this:

1. Start by building the aforementioned max heap and ensuring the largest element is at the top of the heap, or at the root of the array.

2. The minimum element in the heap is then swapped with the maximum element in the heap, placing the largest element at the end of the array. This puts it in the sorted position.

3. The array then needs to go through the heapify function to rebuild it so that the next element can be sent to the end.

4. The process is iteratively done until the entire rray is sorted.

   To accomplish the heapSort algorithm the following code is used:

```
    //Define the function for building all subsequent heaps after the initial maximum heap
//In this array index is the index in the array of the parent node that we want to heapify
//I find it helpful to visualize it as a bunch of little lists that we reverse sort and then append
//one by one

//When we recursively heapify we follow the value along until it is at the bottom of its respective branch
//of the heap
void heapify(int arr[], int index, int n) {

    //Because each branch of the tree grows at 2^n where n is the depth we just need two elements for every layer
    int leftIndex = 2*index + 1;
    int rightIndex = 2*index + 2;

    int maxIndex = 0;

    //The first condition ensures the the index is within the bounds
    //The second checks to see if the value is in the wrong place

    //The first condition could have been an if conditional statement surrounding
    //both but I found that that looked messy, so I used &&
    if((leftIndex < n) && (arr[leftIndex] > arr[index])) {
        maxIndex = leftIndex;
    }
    else {
        maxIndex = index;
    }

    //Same this as before but now with the right index
    //We are fine to use maxIndex here because it will have always been defined at this point in the code
    if((rightIndex < n) && (arr[rightIndex] > arr[maxIndex])) {
        maxIndex = rightIndex;
    }

    if (maxIndex != index) {
        swap(&arr[index], &arr[maxIndex]);
        heapify(arr, maxIndex, n);
    }
```

```
}

//Define the function for building the maximum heap, this only ever
    is called once for any given array
void buildMaxHeap(int arr[], int n) {
    for (int i = n/2 - 1; i >= 0; i--) {
        heapify(arr, i, n);
    }
}

void heapSort(int arr[], int n) {
    //Doing heap sort requires three different functions
    //The heapify function, the build maximum heap function, and
        the sorting function itself

    buildMaxHeap(arr, n);

    for (int i = n - 1; i > 0; i--) {
        swap(&arr[0], &arr[i]);
        heapify(arr, 0, i);
    }

}
```

This requires 3 different functions to accomplish, one for the **heapSort**, one for the **buildMaxHeap**, one for the **heapify**. For line by line descriptions see the comments in the code.

3.5 Counting Sort

The final algorithm implementation to be discussed is the counting sort algorithm. In my opinion this is the simplest of all the different algorithms that we were asked to implement and involved the following steps:

1. Find the range of the inputs that are in the list, this is the difference between the minimum and maximum elements. This is crucial because the array needs to be able to account for negative numbers. Further elaboration is provided in the comments.

2. An array called occurrences is created to keep track of all the different times a number is seen.

3. The array is looped through and all of the different occurrences of the numbers are added up. The array is then looped through one more time and iteratively sums together the current value and the previous values.

4. The indexes in the occurrences array are shifted to the right by one, these will be the starting indexes of each of the elements.

5. Finally, the array is rebuilt and now it is sorted.

To accomplish this the following code was used:

```c
void countingSort(int arr[], int n) {

//Start by constructing a new array of variable length that will keep track of all the occurences of numbers that we see
//So that we dont run into stack issues we can dynamically resize the array that holds the occurences based on the amount of number we see

//Originally my implementation did not account for negative numbers so this for loops fixs that by determining the
//maximum and minimum values in the array and creating a range

//Create some original values so that the min and max aren't unassigned cause that would not be good :)
int min = arr[0];
int max = arr[0];

for (int i = 1; i < n; i++) {
    if (arr[i] < min) {
        min = arr[i];
    }
    else if (arr[i] > max) {
        max = arr[i];
    }
}

//Now we are going to define a variable which represents the range of the array (difference between the lowest and highest number)

int range = max - min + 1; //+1 for 0

//This is another change I made from previous iterations of this code and prevents having to dynamically
//resize the array occurences, even if it does take another for loop. (We had to make the for loop anyway)

int *occurences = (int *)malloc(range * sizeof(int));

//Here is the annoying part where we have to initialize the array with all zeroes because C doesn't default to doing that
for (int i = 0; i < range; i++) {
    occurences[i] = 0;
}

//Now we count the total occurences of each number we see using a loop, account for negative numbers using our new range we defined
for (int i = 0; i  < n; i++) {
    occurences[arr[i] - min]++; //Now we incremement the occurences array at our desired position by one!
}

//Now we need to fill the original array based on our occurences array
```

```
    //We need to remember to account for the min value that we created earlier

    //Using knowledge gained from the merge sort code I remembered that I can place an index outside the loop
    //Here I called it index to make it a little clearer
    int index = 0;

    //Loop through every value in the code
    for (int i = 0; i < range; i++) {

        //We get the "number of each number" using our little occurences array
        int numberOfValues = occurences[i];

        //We now add that many numbers back to our original array! (making sure to account for the minimum value of course ...)
        for (int j = 0; j < numberOfValues; j++) {
            arr[index] = i + min;

            //Now we see why we needed this index value because it needs to be independent of both loops.
            //The first loop goes through all the values we need to add and the second one adds that value the correct number of times
            //The index keeps track of where we are in the original list.
            index++;
        }
    }

}
```

4 Performance Case Study

All of the sorting algorithms that have been discussed were subject to a space and time complexity case study to analyze their performance. This study was carried out with a python script which can be located in the Appendix. The python script is a slight modification of the ipynb to include the new sorting algorithms as well as make it cross platform. The program needed to be cross platform so that I could compile a dll on my Macbook and run the code on a friend's Windows machine, as it refused to run on my slow Macbook.

To import each sorting algorithm into python a shared library was created using a Makefile included with the code. The Makefile included is set to output a shared library for MacOS/Linux, but as I needed to run it on a Windows machine I compiled it using MinGW manually to get a 64-bit dll to execute using python, however the results should be the same. Below I have completed the table from the assignment with the determined time and space complexities from my code as well as the CPU Time it took to complete every method.

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

Sort Method	**Time Complexity**	**Space Complexity**	**CPU Time (Seconds)**
Bubble Sort	$O(n^2)$	O(1)	645.14
Insertion Sort	$O(n^2)$	O(1)	85.07
Merge Sort	O(n log n)	O(n)	0.11
Heap Sort	O(n log n)	O(1)	0.11
Counting Sort	O(n + k)	O(k)	0.014
Python sorted()	O(n log n)	O(n)	0.21
numpy.sort()	O(n log n)	O(n)	0.0029

From these results there is a pretty clear correlation between the observed time complexity and the CPU Time it took to complete every algorithm. It is very interesting how fast the Counting Sort method is! Below I have included the raw output from the code I modified:

```
Original array: [  64 -134   -5    0   25   12   22   11   90]
Sorted array using Bubble Sort: [-134   -5    0   11   12   22   25   64   90]
Sorted array using Insertion Sort: [-134   -5    0   11   12   22   25   64   90]
Sorted array using Merge Sort: [-134   -5    0   11   12   22   25   64   90]
Sorted array using Heap Sort: [-134   -5    0   11   12   22   25   64   90]
Sorted array using Counting Sort: [-134   -5    0   11   12   22   25   64   90]
Original array for large test (first 10 elements): [-806858  789186 -103909 -933328 -998348 -674742 -701594 -837460  524023
  797915] ...
Time to sort using Insertion Sort: 85.0696907043457 seconds
First 10 elements of the sorted array using Insertion Sort: [-999991 -999984 -999975 -999974 -999973 -999965 -999961 -999960 -999958
 -999955]
Time to sort using Merge Sort: 0.10778498649597168 seconds
First 10 elements of the sorted array using Merge Sort: [-999991 -999984 -999975 -999974 -999973 -999965 -999961 -999960 -999958
 -999955]
Time to sort using Heap Sort: 0.10764408111572266 seconds
First 10 elements of the sorted array using Heap Sort: [-999991 -999984 -999975 -999974 -999973 -999965 -999961 -999960 -999958
 -999955]
Time to sort using Counting Sort: 0.013864994049072266 seconds
First 10 elements of the sorted array using Counting Sort: [-999991 -999984 -999975 -999974 -999973 -999965 -999961 -999960 -999958
 -999955]
Time taken by Python's built-in sort: 0.20972967147827148 seconds
Time taken by NumPy's sort: 0.0029420852661132812 seconds
```

Figure 1: Output from python script

5 Conclusion and Instructions for Operation

Sorting algorithms are an integral part of computing and vary widely in their space and time complexity. When creating a sorting algorithm it is very important to take into the needs of the tasks you are using it for as every algorithm appears to have it's own trade off. For the task of sorting a large array I found that using Counting Sort proved to be the best algorithm for the job.
To run the code simply compile and execute the **main.c** file using a command such as **gcc -o a main.c**, and run it using **./a**. To create the shared library for windows type **make** into the terminal. I do want to give a heads up that sometimes the tabs need to be removed and re-entered in the Makefile for some reason, so if it throws a ***** missing separator.** error in the terminal this is the fix.

A C Code

```c
// CODE: include necessary library(s)

#include <stdlib.h>
#include <stdio.h>
#include <strings.h>

// you have to write all the functions and algorithms from scratch,
// You will submit this file, mySort.c holds the actual implementation of sorting functions


void swap(int *x, int *y) {
    int temp = *x;
    *x = *y;
    *y = temp;
}

// Bubble Sort
void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1])
                swap(&arr[j], &arr[j + 1]);
        }
    }
}

// CODE: implement the algorithms for Insertion Sort, Merge Sort, Heap Sort, Counting Sort

//This is the function to merge the elements in merge sort, it contains most of the logic
void mergeElements(int arr[], int leftPosition, int rightPosition) {

    //It is important to do this instead of n/2 because the position in the original array dynamically
    //changes throughout the recursion
    int middlePosition = leftPosition + (rightPosition-leftPosition)/2;

    int leftSize = middlePosition - leftPosition + 1;
    int rightSize = rightPosition - middlePosition;

    //Now we need two different arrays which we will use to temporarly store the data

    int *leftArray = (int *)malloc(leftSize * sizeof(int));
    int *rightArray = (int *)malloc(rightSize * sizeof(int));


    //Append the values of the array from the left side to the temporary array
    for (int i = 0; i < leftSize; i++) {
        leftArray[i] = arr[leftPosition + i ];
```

```c
    }

    //Append the values of the array from the right side to the temporary array
    for (int i = 0; i < rightSize; i++) {
        rightArray[i] = arr[1 + middlePosition + i];
    }

    //Had to ask ChatGPT how to get an extra two indexing variables.
    //It ended up being only these two extra lines that I needed.
    //The reason that they are both set to 0 is because i was already used
    int i = 0; //This is the indexing variable for the left array
    int j = 0; //This is the indexing variable for the right array
    int k; //This is the whole array index
    int iterationNum = (rightSize+leftSize)/2 + 1;

    //Now we need to add the values into the original array, which remember is still in the recursive loop
    //so it isnt actually the original array we were sorting
    // Total number of elements in the range to merge is from leftPosition to rightPosition
    for (k = leftPosition; k <= rightPosition; k++) {
        if (i < leftSize && (j >= rightSize || leftArray[i] <= rightArray[j])) {
            arr[k] = leftArray[i];
            i++; // Move index in leftArray
        } else {
            arr[k] = rightArray[j];
            j++; // Move index in rightArray
        }
    }


    //Now we just need to copy over any remaining elements that were not included in the loop.
    //the reason that this happens is because of the bound we set with the division

    //This copy strategy was taken from this article: https://www.geeksforgeeks.org/c-program-for-merge-sort/

    while (i < leftSize) {
        arr[k] = leftArray[i];
        i++;
        k++;
    }

    while (j < rightSize) {
        arr[k] = rightArray[j];
        j++;
        k++;
    }

    //Free the allocated memory for the left array and right array
    free(leftArray);
```

```
    free(rightArray);

}

//These are the sorting algorithms
void mergeSort(int arr[], int l, int r) {

    //This ensures that the size of the elements that are being merged are greater than 0
    if (l < r) {

        //This defines the middle value for the array
        //The reason that single letter variable names are being used is because that is what is used in the header for this function
        int m = l + (r-l)/2;

        //We run the mergeSort recursively for the two different arrays, the left one and the right one
        //Remember they will not run if the dimenions that it is expecting is less than one
        mergeSort(arr, l, m);

        mergeSort(arr, m+1,r);

        //This function contains the main logic for the algorithm, it swaps the elements in the array depending on what order they are in
        mergeElements(arr, l , r);

    }
}

void insertionSort(int arr[], int n) {

     //Assume the elements in the array prior to the current element are already sorted
     //Start at the first element of the array and loop through all of the elements after that
    for (int i = 1; i < n; i++) {

        //This will grab the value of the element we want to place in the correct position
        int shiftValue = arr[i];

        //We want to use j later so we define it outside of the loop
        int j;

        //We loop through all of the elements in the list
        for (j = i - 1; j >= 0; j--) {

            //If the current element is greater than the value we are looking at we shift the values that are
            //greater than it in the array one to the right
            if(arr[j] > shiftValue) {
                arr[j+1] = arr[j];
```

```
            }
            //Once we have found a place where the value is not greater than it the correct location is found
            else {
                break;
            }

        }

        //Ran into a segmentation here many times because I tried arr[j] instead of arr[j+1] which would result in the indexing
        //at a negative element
        //This places the shiftValue at the correct position, one position ahead of the last position we checked.
        arr[j + 1] = shiftValue;

    }
}

//Define the function for building all subsequent heaps after the initial maximum heap
//In this array index is the index in the array of the parent node that we want to heapify
//I find it helpful to visualize it as a bunch of little lists that we reverse sort and then append
//one by one

//When we recursively heapify we follow the value along until it is at the bottom of its respective branch
//of the heap
void heapify(int arr[], int index, int n) {

    //Because each branch of the tree grows at 2^n where n is the depth we just need two elements for every layer
    int leftIndex = 2*index + 1;
    int rightIndex = 2*index + 2;

    int maxIndex = 0;

    //The first condition ensures the the index is within the bounds
    //The second checks to see if the value is in the wrong place

    //The first condition could have been an if conditional statement surrounding
    //both but I found that that looked messy, so I used &&
    if((leftIndex < n) && (arr[leftIndex] > arr[index])) {
        maxIndex = leftIndex;
    }
    else {
        maxIndex = index;
    }

    //Same this as before but now with the right index
    //We are fine to use maxIndex here because it will have always been defined at this point in the code
```

```c
    if((rightIndex < n) && (arr[rightIndex] > arr[maxIndex])) {
        maxIndex = rightIndex;
    }

    if (maxIndex != index) {
        swap(&arr[index], &arr[maxIndex]);
        heapify(arr, maxIndex, n);
    }

}

//Define the function for building the maximum heap, this only ever is called once for any given array
void buildMaxHeap(int arr[], int n) {
    for (int i = n/2 - 1; i >= 0; i--) {
        heapify(arr, i, n);
    }
}

void heapSort(int arr[], int n) {
    //Doing heap sort requires three different functions
    //The heapify function, the build maximum heap function, and the sorting function itself

    buildMaxHeap(arr, n);

    for (int i = n - 1; i > 0; i--) {
        swap(&arr[0], &arr[i]);
        heapify(arr, 0, i);
    }

}

void countingSort(int arr[], int n) {

    //Start by constructing a new array of variable length that will keep track of all the occurences of numbers that we see
    //So that we dont run into stack issues we can dynamically resize the array that holds the occurences based on the amount of number we see

    //Originally my implementation did not account for negative numbers so this for loops fixs that by determining the
    //maximum and minimum values in the array and creating a range

    //Create some original values so that the min and max aren't unassigned cause that would not be good :)
    int min = arr[0];
    int max = arr[0];

    for (int i = 1; i < n; i++) {
        if (arr[i] < min) {
            min = arr[i];
        }
        else if (arr[i] > max) {
            max = arr[i];
```

```
        }
    }

    //Now we are going to define a variable which represents the range of the array (difference between the lowest and highest number)

    int range = max - min + 1; //+1 for 0

    //This is another change I made from previous iterations of this code and prevents having to dynamically
    //resize the array occurences, even if it does take another for loop. (We had to make the for loop anyway)

    int *occurences = (int *)malloc(range * sizeof(int));

    //Here is the annoying part where we have to initialize the array with all zeroes because C doesn't default to doing that
    for (int i = 0; i < range; i++) {
        occurences[i] = 0;
    }

    //Now we count the total occurences of each number we see using a loop, account for negative numbers using our new range we defined
    for (int i = 0; i  < n; i++) {
        occurences[arr[i] - min]++; //Now we incremement the occurences array at our desired position by one!
    }

    //Now we need to fill the original array based on our occurences array
    //We need to remember to account for the min value that we created earlier

    //Using knowledge gained from the merge sort code I remembered that I can place an index outside the loop
    //Here I called it index to make it a little clearer
    int index = 0;

    //Loop through every value in the code
    for (int i = 0; i < range; i++) {

        //We get the "number of each number" using our little occurences array
        int numberOfValues = occurences[i];

        //We now add that many numbers back to our original array! (making sure to account for the minimum value of course...)
        for (int j = 0; j < numberOfValues; j++) {
            arr[index] = i + min;

            //Now we see why we needed this index value because it needs to be independent of both loops.
```

```
            //The first loop goes through all the values we need to
                add and the second one adds that value the correct
                number of times
            //The index keeps track of where we are in the original
                list.
            index++;
        }
    }

}
```

Listing 1: C code for sorting algorithms

```
import time
import ctypes
import numpy as np
from numpy.ctypeslib import ndpointer
import platform

# Path to the shared library
if platform.system() == 'Windows':
    lib_path = './libmysort.dll'
else:
    lib_path = './libmysort.so'

# Load the shared library
mySortLib = ctypes.CDLL(lib_path)

# input argument types and return types for each sorting function
mySortLib.bubbleSort.argtypes = [ndpointer(ctypes.c_int, flags="
    C_CONTIGUOUS"), ctypes.c_int]
mySortLib.bubbleSort.restype = None

mySortLib.insertionSort.argtypes = [ndpointer(ctypes.c_int, flags="
    C_CONTIGUOUS"), ctypes.c_int]
mySortLib.insertionSort.restype = None

mySortLib.mergeSort.argtypes = [ndpointer(ctypes.c_int, flags="
    C_CONTIGUOUS"), ctypes.c_int, ctypes.c_int]
mySortLib.mergeSort.restype = None

mySortLib.heapSort.argtypes = [ndpointer(ctypes.c_int, flags="
    C_CONTIGUOUS"), ctypes.c_int]
mySortLib.heapSort.restype = None

mySortLib.countingSort.argtypes = [ndpointer(ctypes.c_int, flags="
    C_CONTIGUOUS"), ctypes.c_int]
mySortLib.countingSort.restype = None

# small array test
arr0 = np.array([64, -134, -5, 0, 25, 12, 22, 11, 90], dtype=np.
    int32)
n = len(arr0)

print("Original␣array:", arr0)

# Test Bubble Sort
```

```python
arr_copy = np.copy(arr0)
mySortLib.bubbleSort(arr_copy, n)
print("Sorted array using Bubble Sort:", arr_copy)

# Test Insertion Sort
arr_copy = np.copy(arr0)
mySortLib.insertionSort(arr_copy, n)
print("Sorted array using Insertion Sort:", arr_copy)

# Test Merge Sort (passing 0 as the start index and n-1 as the end
    index)
arr_copy = np.copy(arr0)
mySortLib.mergeSort(arr_copy, 0, n-1)
print("Sorted array using Merge Sort:", arr_copy)

# Test Heap Sort
arr_copy = np.copy(arr0)
mySortLib.heapSort(arr_copy, n)
print("Sorted array using Heap Sort:", arr_copy)

# Test Counting Sort
arr_copy = np.copy(arr0)
mySortLib.countingSort(arr_copy, n)
print("Sorted array using Counting Sort:", arr_copy)

# Creating a large test case
arr = np.random.choice(np.arange(-1000000, 1000000, dtype=np.int32)
    , size=500000, replace=False)
n = len(arr)
print("Original array for large test (first 10 elements):", arr
    [:10], "...")

# Function to time and print sorting results
def time_sorting(sort_func, name, arr, n, *extra_args):
    arr_copy = np.copy(arr)
    start = time.time()

    # Need to pass extra arguments for merge sort
    if extra_args:
        sort_func(arr_copy, *extra_args)
    else:
        sort_func(arr_copy, n)

    end = time.time()
    print(f"Time to sort using {name}: {end - start} seconds")
    print(f"First 10 elements of the sorted array using {name}: {
        arr_copy[:10]}")  # Print first 10 elements

# Timing all the algorithms with the large array
time_sorting(mySortLib.bubbleSort, "Bubble Sort", arr, n)
time_sorting(mySortLib.insertionSort, "Insertion Sort", arr, n)
time_sorting(mySortLib.mergeSort, "Merge Sort", arr, n, 0, n-1)  #
    Passing start and end indices for merge sort
time_sorting(mySortLib.heapSort, "Heap Sort", arr, n)
time_sorting(mySortLib.countingSort, "Counting Sort", arr, n)

# Compare with python's sorting algorithm
```

```python
arr_copy = np.copy(arr)
start = time.time()
sorted_arr = sorted(arr_copy)
end = time.time()
print("Time taken by Python's built-in sort:", end - start, "
    seconds")

# Compare with numpys sorting algorithm
arr_copy = np.copy(arr)
start = time.time()
np_sorted_arr = np.sort(arr_copy)
end = time.time()
print("Time taken by NumPy's sort:", end - start, "seconds")
```

Listing 2: Python code for executing the sorting algorithms and timing them

```makefile
#include the flags mentioned in the assignment description
CC = gcc
CFLAGS = -O3 -fPIC -shared

#make a shared library
TARGET = libmysort.so

#the source file for the code
SRCS = mySort.c

#build the shared library
all:
	$(CC) $(CFLAGS) -o $(TARGET) $(SRCS)

#include a clean if they want to remove the shared library
clean:
	rm -f $(TARGET)
```

Listing 3: Makefile for compiling the sorting algorithms and creating the shared library