

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 first element and compare it with the next element in the array.
2. Compare the first 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:

```
1 void bubbleSort(int arr[], int n) {  
2     for (int i = 0; i < n - 1; i++) {  
3         for (int j = 0; j < n - i - 1; j++) {  
4             if (arr[j] > arr[j + 1])  
5                 swap(&arr[j], &arr[j + 1]);  
6         }  
7     }  
8 }
```

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 then 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 be seen below:

```
1 void insertionSort(int arr[], int n) {
2
3     //Assume the elements in the array prior to the current
4     //element are already sorted
5     //Start at the first element of the array and loop through all
6     //of the elements after that
7     for (int i = 1; i < n; i++) {
8
9         //This will grab the value of the element we want to place
10        //in the correct position
11        int shiftValue = arr[i];
12
13        //We want to use j later so we define it outside of the
14        //loop
15        int j;
16
17        //We loop through all of the elements in the list
18        for (j = i - 1; j >= 0; j--) {
19
20            //If the current element is greater than the value we
21            //are looking at we shift the values that are
22            //greater than it in the array one to the right
23            if(arr[j] > shiftValue) {
24                arr[j+1] = arr[j];
25            }
26            //Once we have found a place where the value is not
27            //greater than it the correct location is found
28            else {
29                break;
30            }
31        }
32    }
33 }
```

```

26     }
27
28     //Ran into a segmentation here many times because I tried
29     arr[j] instead of arr[j+1] which would result in the
30     indexing
31     //at a negative element
32     //This places the shiftValue at the correct position, one
33     position ahead of the last position we checked.
34     arr[j + 1] = shiftValue;
35 }
36 }

```

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 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:

```

1 void mergeElements(int arr[], int leftPosition, int rightPosition)
2 {
3     //It is important to do this instead of n/2 because the
4     position in the original array dynamically
5     //changes throughout the recursion
6     int middlePosition = leftPosition + (rightPosition-leftPosition
7     )/2;
8
9     int leftSize = middlePosition - leftPosition + 1;
10    int rightSize = rightPosition - middlePosition;
11
12    //Now we need two different arrays which we will use to
13    temporarily store the data
14
15    int *leftArray = (int *)malloc(leftSize * sizeof(int));
16    int *rightArray = (int *)malloc(rightSize * sizeof(int));
17
18    //Copy the left half of the array into the leftArray
19    for (int i = leftPosition; i < middlePosition; i++)
20        leftArray[i - leftPosition] = arr[i];
21
22    //Copy the right half of the array into the rightArray
23    for (int i = middlePosition; i < rightPosition; i++)
24        rightArray[i - middlePosition] = arr[i];
25
26    //Merge the two arrays back together
27    int i = 0;
28    int j = 0;
29    int k = leftPosition;
30
31    while (i < leftSize & j < rightSize)
32    {
33        if (leftArray[i] < rightArray[j])
34            arr[k] = leftArray[i];
35            i++;
36        else
37            arr[k] = rightArray[j];
38            j++;
39        k++;
40    }
41
42    //Copy the remaining elements of the leftArray
43    while (i < leftSize)
44        arr[k] = leftArray[i];
45        i++;
46        k++;
47
48    //Copy the remaining elements of the rightArray
49    while (j < rightSize)
50        arr[k] = rightArray[j];
51        j++;
52        k++;
53
54    //Free the memory of the temporary arrays
55    free(leftArray);
56    free(rightArray);
57 }

```

```

15
16 //Append the values of the array from the left side to the
    temporary array
17 for (int i = 0; i < leftSize; i++) {
18     leftArray[i] = arr[leftPosition + i ];
19 }
20
21 //Append the values of the array from the right side to the
    temporary array
22 for (int i = 0; i < rightSize; i++) {
23     rightArray[i] = arr[1 + middlePosition + i];
24 }
25
26 //Had to ask ChatGPT how to get an extra two indexing variables
    .
27 //It ended up being only these two extra lines that I needed.
28 //The reason that they are both set to 0 is because i was
    already used
29 int i = 0; //This is the indexing variable for the left array
30 int j = 0; //This is the indexing variable for the right array
31 int k; //This is the whole array index
32 int iterationNum = (rightSize+leftSize)/2 + 1;
33
34 //Now we need to add the values into the original array, which
    remember is still in the recursive loop
35 //so it isnt actually the original array we were sorting
36 // Total number of elements in the range to merge is from
    leftPosition to rightPosition
37 for (k = leftPosition; k <= rightPosition; k++) {
38     if (i < leftSize && (j >= rightSize || leftArray[i] <=
        rightArray[j])) {
39         arr[k] = leftArray[i];
40         i++; // Move index in leftArray
41     } else {
42         arr[k] = rightArray[j];
43         j++; // Move index in rightArray
44     }
45 }
46
47
48 //Now we just need to copy over any remaining elements that
    were not included in the loop.
49 //the reason that this happens is because of the bound we set
    with the division
50
51 //This copy strategy was taken from this article: https://www.
    geeksforgeeks.org/c-program-for-merge-sort/
52
53 while (i < leftSize) {
54     arr[k] = leftArray[i];
55     i++;
56     k++;
57 }
58
59 while (j < rightSize) {
60     arr[k] = rightArray[j];
61     j++;

```

```

62     k++;
63 }
64
65 //Free the allocated memory for the left array and right array
66 free(leftArray);
67 free(rightArray);
68
69 }
70
71 //These are the sorting algorithms
72 void mergeSort(int arr[], int l, int r) {
73
74     //This ensures that the size of the elements that are being
75     //merged are greater than 0
76     if (l < r) {
77
78         //This defines the middle value for the array
79         //The reason that single letter variable names are being
80         //used is because that is what is used in the header for
81         //this function
82         int m = l + (r-l)/2;
83
84         //We run the mergeSort recursively for the two different
85         //arrays, the left one and the right one
86         //Remember they will not run if the dimensions that it is
87         //expecting is less than one
88         mergeSort(arr, l, m);
89
90         mergeSort(arr, m+1,r);
91
92         //This function contains the main logic for the algorithm,
93         //it swaps the elements in the array depending on what
94         //order they are in
95         mergeElements(arr, l , r);
96     }
97 }

```

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 thought of as a tree where the top element is the parent 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 array is sorted.

To accomplish the heapSort algorithm the following code is used:

```

1 //Define the function for building all subsequent heaps after
  the initial maximum heap
2 //In this array index is the index in the array of the parent node
  that we want to heapify
3 //I find it helpful to visualize it as a bunch of little lists that
  we reverse sort and then append
4 //one by one
5
6 //When we recursively heapify we follow the value along until it is
  at the bottom of its respective branch
7 //of the heap
8 void heapify(int arr[], int index, int n) {
9
10 //Because each branch of the tree grows at  $2^n$  where n is the
   depth we just need two elements for every layer
11 int leftIndex = 2*index + 1;
12 int rightIndex = 2*index + 2;
13
14 int maxIndex = 0;
15
16 //The first condition ensures the the index is within the
   bounds
17 //The second checks to see if the value is in the wrong place
18
19 //The first condition could have been an if conditional
   statement surrounding
20 //both but I found that that looked messy, so I used &&
21 if((leftIndex < n) && (arr[leftIndex] > arr[index])) {
22     maxIndex = leftIndex;
23 }
24 else {
25     maxIndex = index;
26 }
27
28 //Same this as before but now with the right index
29 //We are fine to use maxIndex here because it will have always
   been defined at this point in the code
30 if((rightIndex < n) && (arr[rightIndex] > arr[maxIndex])) {
31     maxIndex = rightIndex;
32 }
33
34 if (maxIndex != index) {
35     swap(&arr[index], &arr[maxIndex]);
36     heapify(arr, maxIndex, n);
37 }

```

```

38 }
39
40
41 //Define the function for building the maximum heap, this only ever
    is called once for any given array
42 void buildMaxHeap(int arr[], int n) {
43     for (int i = n/2 - 1; i >= 0; i--) {
44         heapify(arr, i, n);
45     }
46 }
47
48 void heapSort(int arr[], int n) {
49     //Doing heap sort requires three different functions
50     //The heapify function, the build maximum heap function, and
        the sorting function itself
51
52     buildMaxHeap(arr, n);
53
54     for (int i = n - 1; i > 0; i--) {
55         swap(&arr[0], &arr[i]);
56         heapify(arr, 0, i);
57     }
58 }
59 }

```

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:


```

1 void countingSort(int arr[], int n) {
2
3     //Start by constructing a new array of variable length that
4     //will keep track of all the occurrences of numbers that we
5     //see
6     //So that we dont run into stack issues we can dynamically
7     //resize the array that holds the occurrences based on the
8     //amount of number we see
9
10    //Originally my implementation did not account for negative
11    //numbers so this for loops fixes that by determining the
12    //maximum and minimum values in the array and creating a range
13
14    //Create some original values so that the min and max aren't
15    //unassigned cause that would not be good :)
16    int min = arr[0];
17    int max = arr[0];
18
19    for (int i = 1; i < n; i++) {
20        if (arr[i] < min) {
21            min = arr[i];
22        }
23        else if (arr[i] > max) {
24            max = arr[i];
25        }
26    }
27
28    //Now we are going to define a variable which represents the
29    //range of the array (difference between the lowest and
30    //highest number)
31
32    int range = max - min + 1; //+1 for 0
33
34    //This is another change I made from previous iterations of
35    //this code and prevents having to dynamically
36    //resize the array occurrences, even if it does take another for
37    //loop. (We had to make the for loop anyway)
38
39    int *occurrences = (int *)malloc(range * sizeof(int));
40
41    //Here is the annoying part where we have to initialize the
42    //array with all zeroes because C doesn't default to doing
43    //that
44    for (int i = 0; i < range; i++) {
45        occurrences[i] = 0;
46    }
47
48    //Now we count the total occurrences of each number we see using
49    //a loop, account for negative numbers using our new range
50    //we defined
51    for (int i = 0; i < n; i++) {
52        occurrences[arr[i] - min]++; //Now we increment the
53        //occurrences array at our desired position by one!
54    }
55
56    //Now we need to fill the original array based on our
57    //occurrences array

```

```

42 //We need to remember to account for the min value that we
    created earlier
43
44 //Using knowledge gained from the merge sort code I remembered
    that I can place an index outside the loop
45 //Here I called it index to make it a little clearer
46 int index = 0;
47
48 //Loop through every value in the code
49 for (int i = 0; i < range; i++) {
50
51     //We get the "number of each number" using our little
        occurrences array
52     int numberOfValues = occurrences[i];
53
54     //We now add that many numbers back to our original array!
        (making sure to account for the minimum value of course
        ...)
55     for (int j = 0; j < numberOfValues; j++) {
56         arr[index] = i + min;
57
58         //Now we see why we needed this index value because it
            needs to be independent of both loops.
59         //The first loop goes through all the values we need to
            add and the second one adds that value the correct
            number of times
60         //The index keeps track of where we are in the original
            list.
61         index++;
62     }
63 }
64
65 }

```

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

```
1 // CODE: include necessary library(s)
2
3 #include <stdlib.h>
4 #include <stdio.h>
5 #include <strings.h>
6
7 // you have to write all the functions and algorithms from scratch,
8 // You will submit this file, mySort.c holds the actual
   implementation of sorting functions
9
10
11 void swap(int *x, int *y) {
12     int temp = *x;
13     *x = *y;
14     *y = temp;
15 }
16
17 // Bubble Sort
18 void bubbleSort(int arr[], int n) {
19     for (int i = 0; i < n - 1; i++) {
20         for (int j = 0; j < n - i - 1; j++) {
21             if (arr[j] > arr[j + 1])
22                 swap(&arr[j], &arr[j + 1]);
23         }
24     }
25 }
26
27 // CODE: implement the algorithms for Insertion Sort, Merge Sort,
   Heap Sort, Counting Sort
28
29 //This is the function to merge the elements in merge sort, it
   contains most of the logic
30 void mergeElements(int arr[], int leftPosition, int rightPosition)
   {
31
32     //It is important to do this instead of n/2 because the
       position in the original array dynamically
33     //changes throughout the recursion
34     int middlePosition = leftPosition + (rightPosition-leftPosition
       )/2;
35
36     int leftSize = middlePosition - leftPosition + 1;
37     int rightSize = rightPosition - middlePosition;
38
39     //Now we need two different arrays which we will use to
       temporarily store the data
40
41     int *leftArray = (int *)malloc(leftSize * sizeof(int));
42     int *rightArray = (int *)malloc(rightSize * sizeof(int));
43
44
45     //Append the values of the array from the left side to the
       temporary array
46     for (int i = 0; i < leftSize; i++) {
47         leftArray[i] = arr[leftPosition + i];
```

```

48     }
49
50     //Append the values of the array from the right side to the
    temporary array
51     for (int i = 0; i < rightSize; i++) {
52         rightArray[i] = arr[1 + middlePosition + i];
53     }
54
55     //Had to ask ChatGPT how to get an extra two indexing variables
    .
56     //It ended up being only these two extra lines that I needed.
57     //The reason that they are both set to 0 is because i was
    already used
58     int i = 0; //This is the indexing variable for the left array
59     int j = 0; //This is the indexing variable for the right array
60     int k; //This is the whole array index
61     int iterationNum = (rightSize+leftSize)/2 + 1;
62
63     //Now we need to add the values into the original array, which
    remember is still in the recursive loop
64     //so it isnt actually the original array we were sorting
65     // Total number of elements in the range to merge is from
    leftPosition to rightPosition
66     for (k = leftPosition; k <= rightPosition; k++) {
67         if (i < leftSize && (j >= rightSize || leftArray[i] <=
            rightArray[j])) {
68             arr[k] = leftArray[i];
69             i++; // Move index in leftArray
70         } else {
71             arr[k] = rightArray[j];
72             j++; // Move index in rightArray
73         }
74     }
75
76
77     //Now we just need to copy over any remaining elements that
    were not included in the loop.
78     //the reason that this happens is because of the bound we set
    with the division
79
80     //This copy strategy was taken from this article: https://www.
        geeksforgeeks.org/c-program-for-merge-sort/
81
82     while (i < leftSize) {
83         arr[k] = leftArray[i];
84         i++;
85         k++;
86     }
87
88     while (j < rightSize) {
89         arr[k] = rightArray[j];
90         j++;
91         k++;
92     }
93
94     //Free the allocated memory for the left array and right array
    free(leftArray);
95

```

```

96     free(rightArray);
97
98 }
99
100 //These are the sorting algorithms
101 void mergeSort(int arr[], int l, int r) {
102
103     //This ensures that the size of the elements that are being
104     //merged are greater than 0
105     if (l < r) {
106
107         //This defines the middle value for the array
108         //The reason that single letter variable names are being
109         //used is because that is what is used in the header for
110         //this function
111         int m = l + (r-l)/2;
112
113         //We run the mergeSort recursively for the two different
114         //arrays, the left one and the right one
115         //Remember they will not run if the dimensions that it is
116         //expecting is less than one
117         mergeSort(arr, l, m);
118
119         mergeSort(arr, m+1,r);
120
121         //This function contains the main logic for the algorithm,
122         //it swaps the elements in the array depending on what
123         //order they are in
124         mergeElements(arr, l , r);
125     }
126 }
127
128 void insertionSort(int arr[], int n) {
129
130     //Assume the elements in the array prior to the current
131     //element are already sorted
132     //Start at the first element of the array and loop through all
133     //of the elements after that
134     for (int i = 1; i < n; i++) {
135
136         //This will grab the value of the element we want to place
137         //in the correct position
138         int shiftValue = arr[i];
139
140         //We want to use j later so we define it outside of the
141         //loop
142         int j;
143
144         //We loop through all of the elements in the list
145         for (j = i - 1; j >= 0; j--) {
146
147             //If the current element is greater than the value we
148             //are looking at we shift the values that are
149             //greater than it in the array one to the right
150             if(arr[j] > shiftValue) {
151                 arr[j+1] = arr[j];

```

```

141     }
142     //Once we have found a place where the value is not
        greater than it the correct location is found
143     else {
144         break;
145     }
146
147 }
148
149 //Ran into a segmentation here many times because I tried
        arr[j] instead of arr[j+1] which would result in the
        indexing
150 //at a negative element
151 //This places the shiftValue at the correct position, one
        position ahead of the last position we checked.
152 arr[j + 1] = shiftValue;
153
154 }
155 }
156
157 //Define the function for building all subsequent heaps after the
        initial maximum heap
158 //In this array index is the index in the array of the parent node
        that we want to heapify
159 //I find it helpful to visualize it as a bunch of little lists that
        we reverse sort and then append
160 //one by one
161
162 //When we recursively heapify we follow the value along until it is
        at the bottom of its respective branch
163 //of the heap
164 void heapify(int arr[], int index, int n) {
165
166     //Because each branch of the tree grows at  $2^n$  where n is the
        depth we just need two elements for every layer
167     int leftIndex = 2*index + 1;
168     int rightIndex = 2*index + 2;
169
170     int maxIndex = 0;
171
172     //The first condition ensures the the index is within the
        bounds
173     //The second checks to see if the value is in the wrong place
174
175     //The first condition could have been an if conditional
        statement surrounding
176     //both but I found that that looked messy, so I used &&
177     if((leftIndex < n) && (arr[leftIndex] > arr[index])) {
178         maxIndex = leftIndex;
179     }
180     else {
181         maxIndex = index;
182     }
183
184     //Same this as before but now with the right index
185     //We are fine to use maxIndex here because it will have always
        been defined at this point in the code

```

```

186     if((rightIndex < n) && (arr[rightIndex] > arr[maxIndex])) {
187         maxIndex = rightIndex;
188     }
189
190     if (maxIndex != index) {
191         swap(&arr[index], &arr[maxIndex]);
192         heapify(arr, maxIndex, n);
193     }
194 }
195
196 //Define the function for building the maximum heap, this only ever
197 //is called once for any given array
198 void buildMaxHeap(int arr[], int n) {
199     for (int i = n/2 - 1; i >= 0; i--) {
200         heapify(arr, i, n);
201     }
202 }
203
204 void heapSort(int arr[], int n) {
205     //Doing heap sort requires three different functions
206     //The heapify function, the build maximum heap function, and
207     //the sorting function itself
208
209     buildMaxHeap(arr, n);
210
211     for (int i = n - 1; i > 0; i--) {
212         swap(&arr[0], &arr[i]);
213         heapify(arr, 0, i);
214     }
215 }
216
217 void countingSort(int arr[], int n) {
218
219     //Start by constructing a new array of variable length that
220     //will keep track of all the occurrences of numbers that we
221     //see
222     //So that we dont run into stack issues we can dynamically
223     //resize the array that holds the occurrences based on the
224     //amount of number we see
225
226     //Originally my implementation did not account for negative
227     //numbers so this for loops fixes that by determining the
228     //maximum and minimum values in the array and creating a range
229     //Create some original values so that the min and max aren't
230     //unassigned cause that would not be good :)
231     int min = arr[0];
232     int max = arr[0];
233
234     for (int i = 1; i < n; i++) {
235         if (arr[i] < min) {
236             min = arr[i];
237         }
238         else if (arr[i] > max) {
239             max = arr[i];
240         }
241     }

```



```

235     }
236 }
237
238 //Now we are going to define a variable which represents the
    range of the array (difference between the lowest and
    highest number)
239
240 int range = max - min + 1; //+1 for 0
241
242 //This is another change I made from previous iterations of
    this code and prevents having to dynamically
243 //resize the array occurrences, even if it does take another for
    loop. (We had to make the for loop anyway)
244
245 int *occurences = (int *)malloc(range * sizeof(int));
246
247 //Here is the annoying part where we have to initialize the
    array with all zeroes because C doesn't default to doing
    that
248 for (int i = 0; i < range; i++) {
249     occurences[i] = 0;
250 }
251
252 //Now we count the total occurrences of each number we see using
    a loop, account for negative numbers using our new range
    we defined
253 for (int i = 0; i < n; i++) {
254     occurences[arr[i] - min]++; //Now we increment the
        occurences array at our desired position by one!
255 }
256
257 //Now we need to fill the original array based on our
    occurrences array
258 //We need to remember to account for the min value that we
    created earlier
259
260 //Using knowledge gained from the merge sort code I remembered
    that I can place an index outside the loop
261 //Here I called it index to make it a little clearer
262 int index = 0;
263
264 //Loop through every value in the code
265 for (int i = 0; i < range; i++) {
266
267     //We get the "number of each number" using our little
        occurrences array
268     int numberOfValues = occurences[i];
269
270     //We now add that many numbers back to our original array!
        (making sure to account for the minimum value of course
        ...)
271     for (int j = 0; j < numberOfValues; j++) {
272         arr[index] = i + min;
273
274         //Now we see why we needed this index value because it
            needs to be independent of both loops.

```

```

275         //The first loop goes through all the values we need to
           add and the second one adds that value the correct
           number of times
276         //The index keeps track of where we are in the original
           list.
           index++;
277     }
278 }
279 }
280 }
281 }

```

Listing 1: C code for sorting algorithms

```

1  import time
2  import ctypes
3  import numpy as np
4  from numpy.ctypeslib import ndpointer
5  import platform
6
7  # Path to the shared library
8  if platform.system() == 'Windows':
9      lib_path = './libmysort.dll'
10 else:
11     lib_path = './libmysort.so'
12
13 # Load the shared library
14 mySortLib = ctypes.CDLL(lib_path)
15
16 # input argument types and return types for each sorting function
17 mySortLib.bubbleSort.argtypes = [ndpointer(ctypes.c_int, flags="
    C_CONTIGUOUS"), ctypes.c_int]
18 mySortLib.bubbleSort.restype = None
19
20 mySortLib.insertionSort.argtypes = [ndpointer(ctypes.c_int, flags="
    C_CONTIGUOUS"), ctypes.c_int]
21 mySortLib.insertionSort.restype = None
22
23 mySortLib.mergeSort.argtypes = [ndpointer(ctypes.c_int, flags="
    C_CONTIGUOUS"), ctypes.c_int, ctypes.c_int]
24 mySortLib.mergeSort.restype = None
25
26 mySortLib.heapSort.argtypes = [ndpointer(ctypes.c_int, flags="
    C_CONTIGUOUS"), ctypes.c_int]
27 mySortLib.heapSort.restype = None
28
29 mySortLib.countingSort.argtypes = [ndpointer(ctypes.c_int, flags="
    C_CONTIGUOUS"), ctypes.c_int]
30 mySortLib.countingSort.restype = None
31
32 # small array test
33 arr0 = np.array([64, -134, -5, 0, 25, 12, 22, 11, 90], dtype=np.
    int32)
34 n = len(arr0)
35
36 print("Original array:", arr0)
37
38 # Test Bubble Sort

```

```

39 arr_copy = np.copy(arr0)
40 mySortLib.bubbleSort(arr_copy, n)
41 print("Sorted array using Bubble Sort:", arr_copy)
42
43 # Test Insertion Sort
44 arr_copy = np.copy(arr0)
45 mySortLib.insertionSort(arr_copy, n)
46 print("Sorted array using Insertion Sort:", arr_copy)
47
48 # Test Merge Sort (passing 0 as the start index and n-1 as the end
    index)
49 arr_copy = np.copy(arr0)
50 mySortLib.mergeSort(arr_copy, 0, n-1)
51 print("Sorted array using Merge Sort:", arr_copy)
52
53 # Test Heap Sort
54 arr_copy = np.copy(arr0)
55 mySortLib.heapSort(arr_copy, n)
56 print("Sorted array using Heap Sort:", arr_copy)
57
58 # Test Counting Sort
59 arr_copy = np.copy(arr0)
60 mySortLib.countingSort(arr_copy, n)
61 print("Sorted array using Counting Sort:", arr_copy)
62
63 # Creating a large test case
64 arr = np.random.choice(np.arange(-1000000, 1000000, dtype=np.int32)
    , size=500000, replace=False)
65 n = len(arr)
66 print("Original array for large test (first 10 elements):", arr
    [:10], "...")
67
68 # Function to time and print sorting results
69 def time_sorting(sort_func, name, arr, n, *extra_args):
70     arr_copy = np.copy(arr)
71     start = time.time()
72
73     # Need to pass extra arguments for merge sort
74     if extra_args:
75         sort_func(arr_copy, *extra_args)
76     else:
77         sort_func(arr_copy, n)
78
79     end = time.time()
80     print(f"Time to sort using {name}: {end-start} seconds")
81     print(f"First 10 elements of the sorted array using {name}: {
        arr_copy[:10]}") # Print first 10 elements
82
83 # Timing all the algorithms with the large array
84 time_sorting(mySortLib.bubbleSort, "Bubble Sort", arr, n)
85 time_sorting(mySortLib.insertionSort, "Insertion Sort", arr, n)
86 time_sorting(mySortLib.mergeSort, "Merge Sort", arr, n, 0, n-1) #
    Passing start and end indices for merge sort
87 time_sorting(mySortLib.heapSort, "Heap Sort", arr, n)
88 time_sorting(mySortLib.countingSort, "Counting Sort", arr, n)
89
90 # Compare with python's sorting algorithm

```

```

91 arr_copy = np.copy(arr)
92 start = time.time()
93 sorted_arr = sorted(arr_copy)
94 end = time.time()
95 print("Time taken by Python's built-in sort:", end - start, "
      seconds")
96
97 # Compare with numpy's sorting algorithm
98 arr_copy = np.copy(arr)
99 start = time.time()
100 np_sorted_arr = np.sort(arr_copy)
101 end = time.time()
102 print("Time taken by NumPy's sort:", end - start, "seconds")

```

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

```

1 #include the flags mentioned in the assignment description
2 CC = gcc
3 CFLAGS = -O3 -fPIC -shared
4
5 #make a shared library
6 TARGET = libmysort.so
7
8 #the source file for the code
9 SRCS = mySort.c
10
11 #build the shared library
12 all:
13     $(CC) $(CFLAGS) -o $(TARGET) $(SRCS)
14
15 #include a clean if they want to remove the shared library
16 clean:
17     rm -f $(TARGET)

```

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