Assignment 2 Report: Sorting Algorithms in C

MECHTRON 2MP3: Programming for Mechatronics

Furqaan Khurram Qamar
400514719

Algorithms Implemented:

These are the algorithms implemented by me in this assignment. The space and time complexities will be analyzed in the ***Space and Time Complexity Comparison*** Section.

Bubble Sort:
Bubble Sort is a simple comparison sorting algorithm. It repeatedly steps through the list, compares the adjacent elements and swaps them if they are in the wrong order. This process repeats as many times as the size of the list.

Algorithm:
- Iterate through entire array
- Each element *i* is compared with the next element *i+1*
- If *i* > *i+1*, swap them
- Repeat the process for the entire list

Merge Sort:
Merge Sort is a “divide and conquer” algorithm that divides the unsorted list into sublists down to “lists” of individual elements. It repeatedly merges these sublists to produce new sorted sublists until there is only one “sublist” remaining.

Algorithm:
- Divide unsorted list into n sublists of one element each
- Repeatedly merge these sublists into a new sorted sublist
- Continue merging these until there is only one sublist remaining

Heap Sort:
Heap Sort is a comparison algorithm that uses a “binary heap structure”, like an upside down tree each parent root with two child branches that are in turn parents to two other child branches. It builds a max heap structure with highest on top lowest on bottom. It repeatedly extracts the max element from the heap and rebuilds the heap until it is empty.

Algorithm:

- Build a max heap from input array
- Largest element is at the root of the heap; swap it with the last “smallest” element of the heap
- Reduce the size of the heap and make a max heap again
- Repeat until the heap size is reduced to one

Insertion Sort:

Insertion Sort is a simpler sorting algorithm. It builds a sorted array one element at a time by taking the next element and inserting it into the proper position in the already sorted part of the array.

Algorithm:

- Start with the second element
- Compare the current element with the elements in the sorted portion
- Shift the sorted elements to the right to make room for current element

Counting Sort:

Counting Sort is an algorithm that is based on making an empty counting array the size of range then adding the instances each number occurs to the corresponding element in the counting array. Then use this counting array information to make a new array to place each element in its correct position.

Algorithm:

- Find range of input values (min to max)
- Make a count array the size of range to store frequency of each value
- Build output array based on the modified count array

These algorithms are significantly different in their approach to sorting. Some of these algorithms are significantly more efficient depending on the data input. Merge Sort and Heap Sort are more efficient for large datasets which will be shown by the case study below. Bubble Sort and Insertion Sort are easier to implement and understand since they don’t involve recursion. Counting sort is also effective for larger ranges as well.

Compiling and Running:

The C code provided below should compile and run on any operating system that has a C compiler, such as GCC or Clang. Firstly on an IDE or text editor, open up **main.c**, making sure that mySort.h and mySort.C are in the same folder as main.c. Replace arr[] in main.c with the array that needs to be sorted. To compile and run the code in the terminal using GCC we can use:

```
gcc -o sorting main.c
./sorting
```

This will then successfully run the code.

Space and Time Complexity Comparison:

***The space and time complexities of the below were found using ChatGPT

Sort Method	Complexity		CPU time (Sec)
	Time	***Space***	
Bubble	$O(n^2)$	O(1)	624.23
Insertion	$O(n^2)$	O(1)	30.125
Merge	O(n*log(n))	O(n)	0.08541
Heap	O(n*log(n))	O(1)	0.09359
Counting	O(n+k), k being the range of the input values	O(k)	0.01979
Built-in sorted()	O(n*log(n))	O(n)	0.31860
numpy.sort()	O(n*log(n))	O(n) / O(1)	0.00601

The fastest algorithm was **numpy.sort()** then countingSort(). If we check n*log(n) with n being 500000, we get O(2849485.00217). If we check the same for n+k with n being 500000 and k being 2 million, we get O(2500000). This shows why for large datasets some sorting algorithms like countingSort() would be faster than others like mergeSort() or heapSort() while for smaller datasets other algorithms would be faster.

Time Complexity:

Bubble Sort: Since Bubble Sort has to go through each element and compare to every other element so n elements compared n timed would be n^2.

Insertion Sort: Since Insertion Sort has to compare n times and shift over for each element, using similar logic to Bubble Sort we can see that n elements shifted and compared n times is n^2.

Merge Sort: Merge Sort divides arrays in half and takes a linear amount to merge them together therefore, we have a linear n multiplied by a logarithmic log(n) which is n*log(n).

Heap Sort: Building the heap takes n linear time and extracting each max element would take log(n). Therefore, the time complexity would be n*log(n).

Counting Sort: The algorithm’s time complexity depends on both the number of elements n and the range of the input k. It has a linear time complexity of both n and k making it n+k.

Sorted(): This function is a hybrid of merge and insertion sort. It has a time complexity of n*log(n).

numpy.sort(): This uses a quick sort algorithm which has a time complexity of n*log(n) as well.

Space Complexity:

Bubble Sort: It is an in-place sorting algorithm and only requires a constant amount of extra space. The space used does

not grow with the size of the input array. It has a space complexity of 1.

Insertion Sort: It is also an in-place sorting algorithm. It uses a constant amount of extra space needed to store the current element and index counters etc. The space required does not depend on the array size.

Merge Sort: Merge Sort requires additional space for temporary arrays used during the merging process. Specifically, it creates a temporary array that is proportional to the size of the input array n. Thus, the space complexity is linear relative to the input size.

Heap Sort: It is an in-place sorting algorithm that requires a constant amount of additional space. It constructs the heap in the original array and does not require any significant additional data structures. Therefore, the space complexity remains constant.

Counting Sort: The new count array the size of the range needs to store the count of each distinct value in the range, leading to space complexity that is proportional to the range of the input values.

Sorted(): The space required by sorted() is O(n) since the function requires additional storage for temporary arrays proportional to the input.

numpy.sort(): The space required by numpy.sort() is dependent on whether it is using quicksort or heapsort. Which means the space needed can either be linearly proportional to the input like quick sort or constant like heap sort.

Appendix:

References:

OpenAI. (2024). *ChatGPT* (Mar 14 version) [Large language model]. https://chat.openai.com/chat

Source Code:

main.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "mySort.h"
// Utility functions
void printArray(int arr[], int n);
#include "mySort.c"  // Include the implementation of sorting functions in mySort.c


// Test the sorting algorithms
int main() {
   int arr[] = {64, -134, -5, 0, 25, 12, 22, 11, 90};
   int n = sizeof(arr) / sizeof(arr[0]);

   int testArr[n];
   int testArr2[n];
   int testArr3[n];
   int testArr4[n];
   int testArr5[n];

   // Bubble Sort
   memcpy(testArr, arr, n * sizeof(int));
   printf("Original array: ");
   printArray(testArr, n);
   bubbleSort(testArr, n);
   printf("Bubble sorted array: ");
   printArray(testArr, n);
```

```
    // Merge Sort
    memcpy(testArr2, arr, n * sizeof(int));
    mergeSort(testArr2,0, n-1);
    printf("Merge sorted array: ");
    printArray(testArr2, n);

    //Heap Sort
    memcpy(testArr3, arr, n * sizeof(int));
    heapSort(testArr3,n);
    printf("Heap sorted array: ");
    printArray(testArr3, n);

    //Insertion Sort
    memcpy(testArr4, arr, n * sizeof(int));
    insertionSort(testArr4,n);
    printf("Insertion sorted array: ");
    printArray(testArr4, n);

    //Counting Sort
    memcpy(testArr5, arr, n * sizeof(int));
    countingSort(testArr5,n);
    printf("Counting sorted array: ");
    printArray(testArr5, n);

    return 0;
}

// Helper functions
void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
}
```

mySort.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```c
//-----------------------------------------------------------------
// Bubble Sort

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

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]);
           }
       }
   }
}

//-----------------------------------------------------------------
// Merge Sort - Divide and Conquer, recursively. Then merge,
recursively until you get the full sorted array.

void merge(int arr[], int l, int m, int r) {
   // Calculate the sizes of the left and right subarrays
   int leftSize = m - l + 1; // Size of left subarray
   int rightSize = r - m;    // Size of right subarray

   // Create temporary arrays for left and right subarrays
   int temp_left[leftSize], temp_right[rightSize];

   // Copy data to temporary arrays
   for (int i = 0; i < leftSize; i++) {
       temp_left[i] = arr[l + i]; // Copy left subarray
   }

   for (int i = 0; i < rightSize; i++) {
       temp_right[i] = arr[m + 1 + i]; // Copy right subarray
   }
```

```
        // Merge the temporary arrays back into arr[]
        int i = 0, j = 0, k = l; // Initialize indices for left, right,
and merged arrays
        while (k <= r) { // Iterate until all elements are merged
            // If there are remaining elements in left or right subarrays
            if (i < leftSize && (j >= rightSize || temp_left[i] <=
temp_right[j])) {
                arr[k] = temp_left[i]; // Take element from left subarray
                i++; // Move to next element in left subarray
            } else {
                arr[k] = temp_right[j]; // Take element from right
subarray
                j++; // Move to next element in right subarray
            }
            k++; // Move to the next position in the merged array
        }
    }

    // Function to perform merge sort using recursion
    void mergeSort_recursion(int arr[], int l, int r) {
        if (l < r) { // Base case: if the array has more than one element
            int m = l + (r - l) / 2; // Find the middle index

            // Recursively sort the left half
            mergeSort_recursion(arr, l, m);
            // Recursively sort the right half
            mergeSort_recursion(arr, m + 1, r);
            // Merge the sorted halves
            merge(arr, l, m, r);
        }
    }

    void mergeSort(int arr[], int l, int r) {
        mergeSort_recursion(arr, l, r); // Start the recursion
    }

    //-----------------------------------------------------------------
    // Heap Sort - works by taking "max heap" weird upside down tree
converting each little tree to max numbered tree.
```

```
     void heapify(int arr[], int n, int rootIndex) {
        int Largest = rootIndex; // Initialize largest as root
        int leftChildIndex = 2 * rootIndex + 1; // Left child index
        int rightChildIndex = 2 * rootIndex + 2; // Right child index

        if (leftChildIndex < n && arr[leftChildIndex] > arr[Largest]) {
// If left child index is bigger than root, make it the largest
            Largest = leftChildIndex;
        }

        if (rightChildIndex < n && arr[rightChildIndex] > arr[Largest]) {
// If right child index is bigger than root, make it the largest
            Largest = rightChildIndex;
        }

        if (Largest != rootIndex) { // If largest is not the root anymore
            int temp = arr[rootIndex]; // Swap root and largest
            arr[rootIndex] = arr[Largest];
            arr[Largest] = temp; // Swap root and largest in the original
array

            heapify(arr, n, Largest); // "Heapify" the branch that has
changed
        }
     }

     // Function to perform heap sort
     void heapSort(int arr[], int n) {
        for (int i = n / 2 - 1; i >= 0; i--) {
            // Build max heap
            heapify(arr, n, i);
        }

        // Take out the max element and place it at the end then add it
into arr[]
        for (int i = n - 1; i >= 0; i--) {
            int temp = arr[0];
            arr[0] = arr[i];
            arr[i] = temp;
```

```
            // Heapify yet again to maintain max heap after the swap
(from element 0)
            heapify(arr, i, 0);
        }
    }

    //---------------------------------------------------------------
    // Insertion Sort: simple sort that builds the sorted part one
element at a time

    void insertionSort(int arr[], int n) {
        for (int i = 1; i < n; i++) { // Iterate through each element in
the array starting from the second element
            int initial = arr[i];
            int j = i - 1;

            // Move elements of arr[0..i-1], that are greater than
initial,
            // to one position ahead of their current position
            while (j >= 0 && arr[j] > initial) {
                arr[j + 1] = arr[j]; // Shift element to the right
                j--; // Move to the next element on the left
            }
            // Place the initial element at its correct position
            arr[j + 1] = initial;
        }
    }

    //---------------------------------------------------------------
    // Counting Sort: simple sort that builds the sorted part one
element at a time

    void countingSort(int arr[], int n) {
        // Find the maximum element in the array
        int max = arr[0], min = arr[0];
        for (int i = 1; i < n; i++) {
            if (arr[i] > max) {
                max = arr[i];
            }
            if (arr[i] < min) {
```

```
                min = arr[i];
            }
        }

        // Create a count array to store the count of each element
        int range = max - min + 1; // Total range of elements
        int *count = (int *)calloc(range, sizeof(int));
        if (count == NULL) {
            fprintf(stderr, "Failed to allocate memory\n");
            return;
        }

        // Count occurrences of each element
        for (int i = 0; i < n; i++) {
            count[arr[i] - min]++;
        }

        // Reconstruct the sorted array
        int index = 0;
        for (int i = 0; i < range; i++) {
            while (count[i] > 0) {
                arr[index++] = i + min;
                count[i]--;
            }
        }

        // Free the dynamically allocated memory
        free(count);
    }
```

mySort_test.ipynb

```
import time
import ctypes
import sys
import numpy as np
from numpy.ctypeslib import ndpointer
lib_path = './libmysort.so'
new_limit = 10000000  # New Recursion Val
```

```
sys.setrecursionlimit(new_limit)
current_limit = sys.getrecursionlimit()
print(f"Current recursion limit is set to: {current_limit}")

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

# Define input argument types without conversion using ndpointer
mySortLib.bubbleSort.argtypes = [ndpointer(ctypes.c_int,
flags="C_CONTIGUOUS"), ctypes.c_int]
mySortLib.bubbleSort.restype = None

mySortLib.mergeSort.argtypes = [ndpointer(ctypes.c_int,
flags="C_CONTIGUOUS"), 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

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

arr0 = np.array([64, -134, -5, 0, 25, 12, 22, 11, 90], dtype=np.int32)
arr1 = np.copy(arr0)
arr2 = np.copy(arr0)
arr3 = np.copy(arr0)
arr4 = np.copy(arr0)
n = len(arr0)
print("Original array:", arr0)

mySortLib.bubbleSort(arr0, n)
mySortLib.mergeSort(arr1, 0, n-1)
mySortLib.heapSort(arr2, n)
mySortLib.insertionSort(arr3, n)
```

```
mySortLib.countingSort(arr4, n)

print("Sorted array using Bubble Sort:", arr0)
print("Sorted array using Merge Sort:", arr1)
print("Sorted array using Heap Sort:", arr2)
print("Sorted array using Insertion Sort:", arr3)
print("Sorted array using Counting Sort:", arr4)

# 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: ", arr)

arr_copy = np.copy(arr)
arr2_copy = np.copy(arr)
arr3_copy = np.copy(arr)
arr4_copy = np.copy(arr)
arr5_copy = np.copy(arr)

startMerge = time.time()
mySortLib.mergeSort(arr2_copy,0,n-1)
endMerge = time.time()
print("Sorted array using Merge Sort:", arr2_copy)
print(f"Time to convert: {endMerge - startMerge} seconds")

startHeap = time.time()
mySortLib.heapSort(arr3_copy, n)
endHeap = time.time()
print("Sorted array using Heap Sort:", arr3_copy)
print(f"Time to convert: {endHeap - startHeap} seconds")

startInsertion = time.time()
mySortLib.insertionSort(arr4_copy, n)
endInsertion = time.time()
print("Sorted array using Insertion Sort:", arr4_copy)
print(f"Time to convert: {endInsertion - startInsertion} seconds")

startCounting = time.time()
mySortLib.countingSort(arr5_copy, n)
```

```
endCounting = time.time()
print("Sorted array using Counting Sort:", arr5_copy)
print(f"Time to convert: {endCounting - startCounting} seconds")

startBubble = time.time()
mySortLib.bubbleSort(arr_copy, n)
endBubble = time.time()
print("Sorted array using Bubble Sort:", arr_copy)
print(f"Time to convert: {endBubble - startBubble} seconds")

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

arr_copy = np.copy(arr)
start = time.time()
np.sort(arr_copy)  # NumPy's optimized sort
end = time.time()
print("Time taken by NumPy sort:", end - start, "seconds")
```