Data Structures Assignment 3

Soundarya Lahari Murari

October 2023

1 Merge Sort

Lets say we have n elements in the array. Merge sorts works by dividing the array into two halves recursively, sorting them, and then merging them back together into one array.
Lets say that the time taken to sort n elements is T(n) and the time taken to merge the elements back together is M(n). M(n) is cn where c is some constant, because merging is O(n).
Since we split each array into 2 parts, we can write

$$\begin{aligned} T(n) &= 2T(n/2) + M(n) \\ &= 2T(n/2) + nc \\ &= 2(2T(n/4) + \frac{cn}{2} + cn \\ &= 2^2T(n/4) + 2cn \\ &= \ldots\ldots\ldots\ldots\ldots \\ &= 2^pT(n/2^p) + pcn \end{aligned}$$

The program ends when $n/2^p = 1$, so $p = log_2 n$.
Hence, $T(n) = 2^pT(n/2^p) + pcn = n + nlog_2 n$.
Therefore, the time complexity is O(nlog$_2$n). The worst case, best case, and average case all have the same time complexity because even in the best case when the input array is already sorted, merge sort still divides the array and then merges it. This is evident from the code as well.

2 Quick Sort

Quick Sort works by selecting a pivot(usually the first, last, or middle element of the array) and moving all elements smaller than the pivot to the

left of the pivot and all elements greater than the pivot to the right of the pivot. the array of elements greater and the array of elements smaller are further sorted recursively.
T(n) = T(k) + T(n-k-1) + P(n) where k is the number of elements less than the pivot and P(n) is the time complexity needed to organise the elements. P(n) = cn. The best case time complexity of quick sort occurs when we take the pivot to be the middle element of the array, so k = n/2.
T(n) = T(n/2) + T(n/2 - 1) + P(n)
T(n) = 2*T(n/2) + P(n)
This is similar to merge sort, hence the best case time complexity is O(n$\log_2$n). The worst case time complexity occurs when the algorithm ends up picking the largest or the smallest element as the pivot. This usually happens when the first or last element are chosen to be the pivot and the input array is already sorted or is in descending order. The average time complexity is O(n$\log_2$n).

$$\begin{aligned} T(n) &= T(n-1) + cn \\ &= T(n-2) + c(n-1) + cn \\ &= T(n-3) + c(n-2) + c(n-1) + cn \\ &= T(0) + c(n(n+1)/2) \\ &= (n^2 + n)/2 \end{aligned}$$

Hence, the worst case time complexity = O(n^2). The average case time complexity is O(n$\log_2$n).

3 Heap Sort

The best case, worst case, and average case time complexity of heap sort are all O(n$\log_2$n), since regardless of the order of the elements, we call the heapify function n times and the time complexity of the heapify function is $\log_2$n.

4 Comparison of Heap Sort with Quick Sort and Merge Sort

Both heap sort and merge sort have the same time complexity, but merge sort is faster than heap sort because heap sort requires swapping, but merge

sort only requires the movement of data from one array to another and swapping takes more time than movement.
If quick sort is implemented properly, with the right pivot chosen, then it will never reach its worst case. If it is implemented properly, then the time complexity is the same as heap sort's, but it is usually faster because it only swaps what is out of order. When array is already sorted:

```
Enter length of array: 8
Enter integers in array: 1
2
3
4
5
6
7
8
Unsorted Array: 1 2 3 4 5 6 7 8
Merge Sorted Array: 1 2 3 4 5 6 7 8
Comparison Count: 12
Move Count: 24

Quick Sorted Array: 1 2 3 4 5 6 7 8
Comparison Count: 28
Swap Count: 35

Heap Sorted Array: 1 2 3 4 5 6 7 8
Comparison Count: 20
Swap Count: 23
```

This is worst case for quick sort, hence the time complexity = 63 which almost = 64 = n^2 as explained earlier. Time complexity of merge sort = 36 which is close enough to n + $n\log_2 n$ = 8 + 8*3 = 32. It is of $O(n\log_2 n)$ as explained earlier. For heap sort, the time complexity = 43 which is $O(n\log_2 n)$, as explained earlier. The time complexity of merge sort is less than heap sort as explained earlier.
When array is random:

```
Enter length of array: 8
Enter integers in array: 2
56
34
```

```
17
4
0
21
964
Unsorted Array: 2 56 34 17 4 0 21 964
Merge Sorted Array: 0 2 4 17 21 34 56 964
Comparison Count: 16
Move Count: 24

Quick Sorted Array: 0 2 4 17 21 34 56 964
Comparison Count: 8
Swap Count: 12

Heap Sorted Array: 0 2 4 17 21 34 56 964
Comparison Count: 13
Swap Count: 20
```

The time complexity of quick sort, heap sort, and merge sort are all O(nlog$_2$n).

5 Insertion Sort

Works by iterating through the array comparing the current element with all the elements before it and placing it right before all the elements that are greater than it. It does this by moving all the greater elements one position to the right to make space for the current element.
The best case complexity occurs when the array is already sorted because nothing is moved at all. Hence, the best case time complexity is O(n).
The worst case complexity occurs when the array elements are in descending order. This is because each element requires all the elements before it to move one position forward.
In the worst case, the number of swaps is 1 + 2 + 3 + ... + n-2 + n-1 + n = n(n+1)/2. Therefore, the worst case time complexity is O(n^2). The average case complexity is O(n^2).

6 Bubble Sort

The best case complexity is O(n) when the array is already sorted because nothing is moved at all.

The worst case complexity is when the array is in descending order. There will be n-1 + n-2 + ... + 1 = n(n-1)/2 swaps, which is O(n^2).
The average case complexity is O(n^2).

7 Comparison of Bubble and Insertion Sort

They both have the same best, worst, and average time complexity, but insertion sort will be faster because swapping has a higher time complexity than moving.
Best Case Output:

```
Enter length of array: 8
Enter integers in array: 1
2
3
4
5
6
7
8
Insertion Sort
Comparison Count: 0
Swap Count: 0
1 2 3 4 5 6 7 8

Bubble Sort
Comparison Count: 0
Swap Count: 0
1 2 3 4 5 6 7 8
```

As explained earlier, it is O(n).
Worst Case Output:

```
Enter length of array: 8
Enter integers in array: 8
7
6
5
4
3
2
```

```
1
Insertion Sort
Comparison Count: 28
Move Count: 28
1 2 3 4 5 6 7 8

Bubble Sort
Comparison Count: 28
Swap Count: 28
1 2 3 4 5 6 7 8
```

As explained earlier, it is O(n^2).
Average Case Output:

```
Enter length of array: 8
Enter integers in array: 12
6
3
9
47
83
0
2
Insertion Sort
Comparison Count: 16
Move Count: 16
0 2 3 6 9 12 47 83

Bubble Sort
Comparison Count: 16
Swap Count: 16
0 2 3 6 9 12 47 83
```

As explained earlier, it is O(n^2).