Data Structures Assignment 3

Aditi Srivastava

April 2023

1 Question 2

FULL CODE SUBMITTED ON GITHUB

1) Implementing insertion sort:

```
void insertionSort (int arr[], int n){
    for (int k = 1; k < n; k++){
      int key = arr[k];
      int p;
      for (p = k - 1; p >= 0 && arr[p] > key; p--){
      arr[p + 1] = arr[p];
        }
      arr[p + 1] = key;
    }
}
```

Figure 1: Insertion Sort C-Code

2) Calculating time complexity of Insertion sort:

Worst Case Scenario:
When k=arr[1], p=arr[0]: in the worst case scenario, 1 comparison and shift is made.
In the next iteration, when k=2, in the worst case, p=1 and then p=0: 2 comparisons and 2 shifts are made
and so on.
In the end, when k=arr[n-1] in an n term array, n-1 comparisons and shifts will be made in the worst case scenario.
So,

$$(n-1) + (n-2) + ... + 2 + 1$$

$$= (n-1)(n)/2 = O(n^2)$$

Best Case Scenario:
No more than 1 comparison made for all n elements, hence, time complexity is O(n). 0 Shifts will be made.

3) Time Complexity of Bubble Sort:

Time Complexity: Bubble sort's time complexity is similar to that of insert

```
void swap(int *xp, int *yp){
    int temp;
    temp = *xp; *xp = *yp; *yp = temp;
}

void bubbleSort(int arr[], int n){
    int i, j, flag, comparisonCnt = 0, swapCnt = 0;
    for (i = 0; i < n - 1; i++) {
        flag = 0;
        for (j = 0; j < n - i - 1; j++) {
        comparisonCnt++;
            if (arr[j] > arr[j + 1]) {
                swap(&arr[j], &arr[j + 1]);
                flag = 1;
                swapCnt++;
                }
            }
        if (flag == 0) break;
        }
    printf("comp %d, swap %d\n", comparisonCnt, swapCnt);
}
```

Figure 2: Bubble sort, as seen in class slides

sort as adjacent elements are iteratively swapped till largest terms are bubbled near the end of the array.
Worst Case Scenario:
For the largest number, the worst case scenario would be that it is the first element of the array, hence, n-1 swaps.
Second largest number, n-2 swaps.
So on,

$$(n-1) + (n-2) + ... + 2 + 1.$$

$$= (n)(n-1)/2 = O(n^2)$$

Best Case Scenario:
No more than 1 comparison needs to be made for all n elements, hence, time complexity is O(n). 0 Swaps will be made.

4) Experimental Data:
Running bubble sort and insertion sort through same array, we note that in the case of a small array, comparison and swap count of insertion sort are lower than bubble sort. Instances of outputs are attached.

2 Question 3

FULL CODE SUBMITTED ON GITHUB
Time complexity of Merge Sort:
Worst Case Scenario:
To reduce arrays with n elements to arrays with 1 element by splitting, x number of splits need to take place, where 2powerx = n. This implies x = log(n). However, during the re-merging phase, n elements need to be sorted and merged again, resulting in a time complexity of **O(n log n)**.

Best Case Scenario:
Even if the array is completely sorted, merge sort will still split the entire array and then re-merge all the elements. The complexity will always be **O(n log n)**.

Time Complexity of Heap Sort:
For heap sort, both the best-case and worst-case scenarios have a time complexity of **O(nlogn)**. This is because regardless of whether the input is already sorted or in a random order, heap sort always requires O(nlogn) time to sort n elements.

In a heap, the height of the binary tree is log(n) for n elements. The process of moving elements downward in the heap takes logn time for each element: This results in time complexity of **O(nlogn)** since there are n elements that are making this journey sequentially.

Time Complexity of Quick Sort:
Worst Case Scenario:
Smallest or largest element could end up as the pivot, this would lead to n calls for the n'th element, n-1 for n-1'th and so on, leading to the previously noted **O(n-squared)** time complexity.

Best Case Scenario:
Similar to heap and merge, **O(nlogn)** will be time complexity as the tree has log-n levels and at all levels of recursion, n elements are involved, leading to O

(n log n).

Experimental Data: Instances of outputs are attached.