# DAA Assignment 1

Let's address your questions step by step.

## 1a. Describe the importance of PIVOT element in quick sort. How the position of it affects the performance of quick sort? Explain with algorithm.

The **PIVOT** element is crucial in the Quick Sort algorithm as it determines how the array is partitioned. The process involves selecting a pivot element from the array, then rearranging the other elements into two partitions: those less than the pivot and those greater than the pivot. This partitioning is essential because it enables the algorithm to sort the array in a recursive manner.

### How the Position of the PIVOT Affects Performance

1. **Optimal Position**: When the pivot divides the array into two equal halves, it achieves the best performance, leading to a time complexity of $O(n \log n)$.
2. **Worst Position**: If the pivot is consistently the smallest or largest element (e.g., in a sorted array), it leads to unbalanced partitions, causing the worst-case scenario of $O(n^2)$.

## Quick Sort Algorithm

```
Algorithm quickSort(a, low, high)
    If (high > low) then
        m = partition(a, low, high)
        if (low < m) then quickSort(a, low, m)
        if (m + 1 < high) then quickSort(a, m + 1, high)

Algorithm partition(a, low, high)
    i = low
    j = high
    mid = (low + high) / 2
    pivot = a[mid]

    while (i <= j) do
        while (a[i] <= pivot) do
            i++
        while (a[j] > pivot) do
            j--

        if (i <= j) then
            temp = a[i]
            a[i] = a[j]
            a[j] = temp
            i++
            j--

    return j
```

## 1b. For the above quick sort algorithm compute the worst case, best case and average case time complexity.

1. **Best Case**: The best case occurs when the pivot divides the array into two equal halves at each level of recursion:
    - **Time Complexity**: $O(n \log n)$
2. **Worst Case**: The worst case occurs when the pivot is consistently the smallest or largest element, leading to $n$ recursive calls, each handling $n - 1$ elements:
    - **Time Complexity**: $O(n^2)$

3. **Average Case**: The average case, which considers random distribution of the input, tends to behave similarly to the best case:
    - **Time Complexity**: $O(n \log n)$

## 2. Explain about space and time complexity in detail.

The time complexity of the quicksort algorithm can vary depending on how the pivot element is chosen and the characteristics of the input data. Here's a breakdown of the worst-case, best-case, and average-case time complexities:

## 1. Best Case Time Complexity

- **Description**: The best case occurs when the pivot divides the array into two nearly equal halves at each recursive step.
- **Time Complexity**: (O(n \log n))
    - Each level of recursion processes (n) elements, and there are approximately (\log n) levels when the array is divided into halves.

## 2. Average Case Time Complexity

- **Description**: The average case occurs when the pivot divides the array into two sub-arrays of varying sizes, but on average, the divisions remain relatively balanced.
- **Time Complexity**: (O(n \log n))
    - Similar to the best case, this is derived from the analysis of random pivot selections over multiple inputs.

## 3. Worst Case Time Complexity

- **Description**: The worst case occurs when the pivot is consistently the smallest or largest element, leading to unbalanced partitions (e.g., when the array is already sorted or nearly sorted).
- **Time Complexity**: (O(n^2))
    - In this scenario, each level of recursion only reduces the size of the array by one, leading to (n) levels of recursion with (n) comparisons at each level.

| Case | Time Complexity |
|---|---|
| Best Case | (O(n \log n)) |
| Average Case | (O(n \log n)) |
| Worst Case | (O(n^2)) |

## Space Complexity

- **Space Complexity**: (O(\log n)) for the recursive stack in the best and average cases; (O(n)) in the worst case when using additional space for array copying (if

needed).

## 3. I=[2,12, 18,3,34,27,56,4,8,3,10] write and explain the merge sort algorithm that outputs the sorted list of elements.

**Merge Sort** is a divide-and-conquer algorithm that divides the input array into two halves, recursively sorts both halves, and then merges them back together.

Merge Sort algorithm:

```
Algorithm mergesort(low, high)
    if (low < high) then
        mid = (low + high) / 2
        mergesort(low, mid)
        mergesort(mid + 1, high)
        Merge(low, mid, high)

void Merge(low, mid, high)
    k = low
    i = low
    j = mid + 1

    while (i <= mid && j <= high) do
        if (a[i] <= a[j]) then
            temp[k] = a[i]
            i++
            k++
        else
            temp[k] = a[j]
            j++
            k++

    while (i <= mid) do
        temp[k] = a[i]
        i++
        k++

    while (j <= high) do
        temp[k] = a[j]
        j++
        k++

    for k = low to high do
        a[k] = temp[k]
```

1. **Divide**: The list is split in half recursively until the sub-lists contain a single element.
2. **Conquer**: Each half is sorted independently by the recursive calls.
3. **Combine**: The sorted halves are merged back together to form the sorted list.

## Time Complexity

- **Best Case**: $O(n \log n)$
- **Worst Case**: $O(n \log n)$
- **Average Case**: $O(n \log n)$

## Space Complexity

- **Space Complexity**: $O(n)$ due to the additional space required for the temporary arrays during the merge process.

Here's a step-by-step breakdown of the merge sort algorithm, showing the array's state at each step along with the conditions and explanations side by side.

## Merge Sort Steps with Array States

### 1. Divide the List

Given the list:
$l = (2, 12, 18, 3, 34, 27, 56, 4, 8, 3, 10)$

1. Find the midpoint:
    - Length of the list = 11
    - Midpoint index = $\frac{0+10}{2} = 5$
2. Divide into two halves:
    - Left half: $(2, 12, 18, 3, 34)$
    - Right half: $(27, 56, 4, 8, 3, 10)$

### 2. Recursively Sort Each Half

**Sorting the Left Half**:

- $(2, 12, 18, 3, 34)$
    - Midpoint = 2
    - Divide into $(2, 12)$ and $(18, 3, 34)$
    - Sort $(2, 12)$: already sorted.
    - Sort $(18, 3, 34)$:
        - Midpoint = 1
        - Divide into $(18)$ and $(3, 34)$
        - Sort $(3, 34)$: already sorted as $(3, 34)$
        - Merge $(18)$ and $(3, 34)$ to get $(3, 18, 34)$
    - Merge $(2, 12)$ and $(3, 18, 34)$:
        - Compare elements:
            - $2$ vs $3$: $2$ is smaller, so first.

        - Next is $3$.
        - Next is $12$.
        - Next is $18$.
        - Finally $34$.

    Resulting in: $(2, 3, 12, 18, 34)$

**Sorting the Right Half**:

- $(27, 56, 4, 8, 3, 10)$
    - Midpoint = 3
    - Divide into $(27, 56)$ and $(4, 8, 3, 10)$
    - Sort $(27, 56)$: already sorted.
    - Sort $(4, 8, 3, 10)$:
        - Midpoint = 1
        - Divide into $(4)$ and $(8, 3, 10)$
        - Sort $(8, 3, 10)$:
            - Midpoint = 1
            - Divide into $(8)$ and $(3, 10)$
            - Sort $(3, 10)$: already sorted.
            - Merge $(8)$ and $(3, 10)$ to get $(3, 8, 10)$
        - Merge $(4)$ and $(3, 8, 10)$:
            - Compare elements:
              - $3$ vs $4$: $3$ is smaller.
              - Next $4$.
              - Finally $8$ and $10$.
              Resulting in: $(3, 4, 8, 10)$

## 3. Merge the Sorted Halves

Finally, we merge the sorted left half $(2, 3, 12, 18, 34)$ and the sorted right half $(3, 4, 8, 10, 27, 56)$:

- Compare:
    - $2$ vs $3$: $2$ is smaller.
    - Next is $3$ (from either side).
    - Next is $3$ (right side).
    - Next is $4$.
    - Next is $8$.
    - Next is $10$.
    - Next is $12$.
    - Next is $18$.
    - Finally $27$ and $56$.

Resulting in the final sorted array:
$(2, 3, 3, 4, 8, 10, 12, 18, 27, 34, 56)$

Here's an updated table for the merge sort algorithm that includes the initial array state, conditions, actions, and resulting array states after each step, starting from the initial array [2, 12, 18, 3, 34, 27, 56, 4, 8, 3, 10]:

| Step | Condition / Action | Array State |
|---|---|---|
| 1 | Start with the input array | **[2, 12, 18, 3, 34, 27, 56, 4, 8, 3, 10]** |
| 2 | If $low < high$ then split into two halves | **[2, 12, 18, 3, 34] [27, 56, 4, 8, 3, 10]** |
| 3 | Sort the left half | **[2, 12] [18, 3, 34] [27, 56, 4, 8, 3, 10]** |
| 4 | If $low < high$ then split [2, 12] | **[2] [12]** |
| 5 | Merge [2] and [12]: If $a[i] \leq a[j]$ | **[2, 12]** |
| 6 | Sort the right half [18, 3, 34] | **[18] [3, 34]** |
| 7 | If $low < high$ then split [3, 34] | **[3] [34]** |
| 8 | Merge [3] and [34]: If $a[i] \leq a[j]$ | **[3, 34]** |
| 9 | Merge [2, 12] and [3, 34]: | **[2, 3, 12, 34]** |
| | - Compare [2] with [3] | - Place $2$ into the merged array |
| | - Compare [12] with [3] | - Place $3$ into the merged array |
| | - Place remaining [12, 34] | |
| 10 | Sort the right half [27, 56, 4, 8, 3, 10] | **[27, 56] [4, 8] [3, 10]** |
| 11 | If $low < high$ then split [27, 56] | **[27] [56]** |
| 12 | Merge [27] and [56]: If $a[i] \leq a[j]$ | **[27, 56]** |
| 13 | Sort [4, 8]: If $low < high$ | **[4] [8]** |
| 14 | Merge [4] and [8]: If $a[i] \leq a[j]$ | **[4, 8]** |
| 15 | Sort [3, 10]: If $low < high$ | **[3] [10]** |
| 16 | Merge [3] and [10]: If $a[i] \leq a[j]$ | **[3, 10]** |
| 17 | Merge [4, 8] and [3, 10]: | **[3, 4, 8, 10]** |
| | - Compare [3] with [4] | - Place $3$ into the merged array |
| | - Compare [4] with [10] | - Place $4$ into the merged array |
| | - Place remaining [8, 10] | |
| 18 | Merge [27, 56] and [3, 4, 8, 10]: | **[3, 4, 8, 10, 27, 56]** |
| | - Compare [3] with [27] | - Place $3$ into the merged array |
| | - Compare [4] with [27] | - Place $4$ into the merged array |
| | - Compare [8] with [27] | - Place $8$ into the merged array |

| Step | Condition / Action | Array State |
|---|---|---|
| | - Compare [10] with [27] | - Place $10$ into the merged array |
| | - Place remaining [27, 56] | |
| 19 | Merge [2, 12] and [3, 4, 8, 10, 27, 56]: | **[2, 3, 4, 8, 10, 12, 27, 56]** |
| 20 | Final Merge [2, 3, 12, 34] and [3, 4, 8, 10, 27, 56] | **[2, 3, 3, 4, 8, 10, 12, 18, 27, 34, 56]** |
| 21 | End of sorting | **[2, 3, 3, 4, 8, 10, 12, 18, 27, 34, 56]** |

The merge sort algorithm effectively sorts the given list $l$ by recursively dividing it into smaller sublists, sorting those, and merging them back together. The final sorted output is:
$(2, 3, 3, 4, 8, 10, 12, 18, 27, 34, 56)$

This approach is efficient with a time complexity of $O(n \log n)$ in all cases and has a space complexity of $O(n)$ due to the temporary storage used during merging.