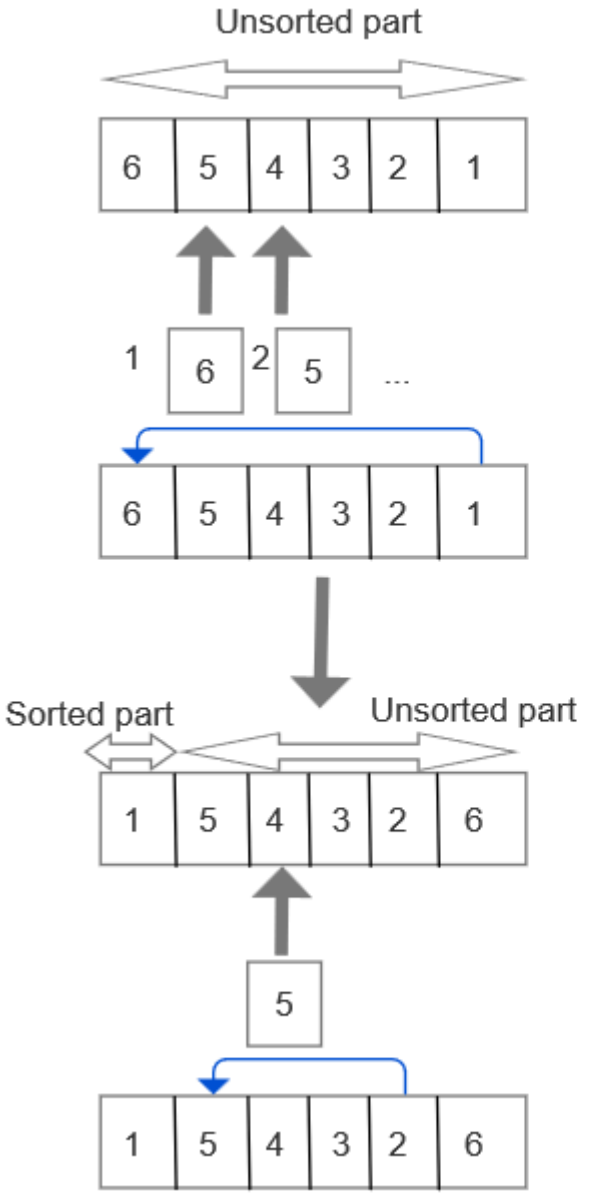
Question 3:

a) Given 3 different types of sorting algorithm (selection, insertion and merge sort), analyse the algorithms to determine the number of comparisons under the following 3 scenarios (a sorted array, array with all identical items, an unsorted array) for 6 items.

Selection Sort

In Selection Sort, the array is split into two parts, the sorted part and the unsorted part, firstly, the element is selected from the front of the unsorted part of the array, the selected element is compared with the following elements in the unsorted array where the selected element may change based on the result of each comparison. After going through the whole unsorted part of the array, a shift occurs where the number of elements in the sorted array increases by 1, and the number of elements in the unsorted array decreases by 1. The time complexity is O(n^2).



No.	Selection	Comparison
1	6	6 vs 5
2	5	5 vs 4
3	4	4 vs 3
4	3	3 vs 2
5	2	2 vs 1
6	1	

No.	Selection	Comparison
1	5	5 vs 4
2	4	4 vs 3
3	3	3 vs 2
4	2	2 vs 6
5	2	

Regardless of the content of the array, selection sort aims to go through the entire unsorted array to determine the smallest element each time, hence the number of comparisons for an unsorted array, a sorted array and an array with all identical items will be the same.

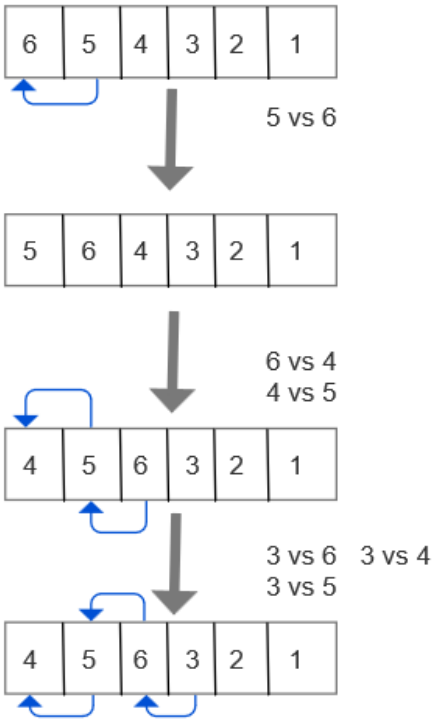
For an array of size n = 6, the total number of comparisons for all scenarios following the pattern as seen in the above chart is 5 + 4 + 3 + 2 + 1 = 15. The formula (n(n-1))/2 may also be used to determine the total number of comparisons.

Insertion Sort

The formula for insertion sort can be written as:

```
for (x = 1; x < length(array); x++)
{
   y = x;
   while( y>0 && array[y-1] > array[y])
     {
       swap array[y] and array[y-1];
       y = y - 1;
     }
}
```

Considering the worst case scenario, the unsorted array in reverse order, the time complexity would be O(n^2). Comparisons would be made between the current element and every preceding element as shown in the partial solution chart example below.



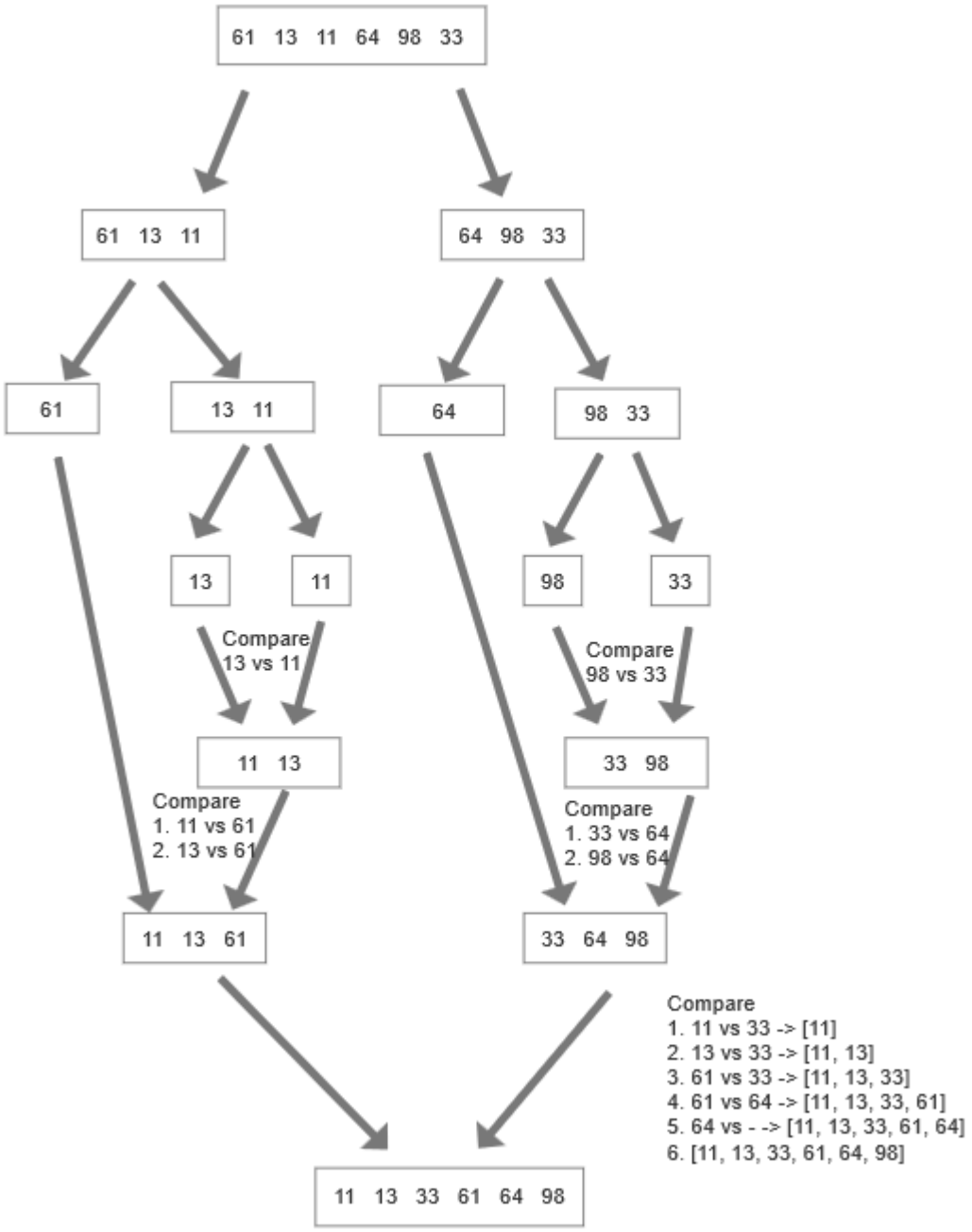
As seen by the pattern, for n = 6 items in an array in the worst case scenario, an unsorted array in the reverse order, the total number of comparisons would be 1 + 2 + 3 + 4 + 5 = 15.

The number of comparisons for an unsorted, reverse ordered array follows the formula (n(n-1))/2.
In the case that the unsorted array is not the worst case where the array is in reverse order, the total number of comparisons may be less than 15. The time complexity of a randomly ordered or reverse ordered array is O(n^2).

For a sorted array, each element in the array need only be compared to the element directly preceding it once before moving on to the next element. Hence, the number of comparisons for a sorted array is 1 + 1 + 1 + 1 + 1 = 5.

An array with all identical items would be similar to the case of a sorted array, where no swapping between elements is needed. The number of comparisons is 1 + 1 + 1 + 1 + 1 = 5. The time complexity for a sorted array and array with all identical items is O(n) and the number of comparisons for both also follow the formula n-1.

Merge Sort
Merge Sort utilises the Divide and Conquer method where the list is first divided into sub-lists and then the items are compared and merged. The time complexity of Merge sort is O(nlogn).
Case 1: Unsorted array
Considering an unsorted array of 6 randomly generated numbers [61 13 11 64 98 33]



There are a total of 11 comparisons for merge sort for an unsorted array when the comparisons are counted manually.
Note: To approximate the number of comparisons required, the formula nlog(base2)n can be used where n = 6. 6log(base2)(6) = 15.51 = 16 (rounded). For smaller arrays like n = 6, the formula nlog(base2)n - (n-1) may be used to more precisely approximate the number of comparisons required 6. 6log(base2)(6)- (6-1) = 10.51 = 11 (rounded).

For the cases of sorted array and array with all identical items, the method of divide and conquer does not take into account the content of the arrays, so the number of comparisons would be the same, 11 comparisons for the other 2 cases.

b) Explain what it means by stability in a sorting algorithm. Explain 2 situations why stability in sorting is desired. Implement and discuss 1 stable and 1 unstable sorting algorithm. Proof your selection with test cases and a detailed discussion of the algorithm. Analyse the efficiency of your algorithm using the Big-oh notation.

What is stability in a sorting algorithm?

The stability of a sorting algorithm refers to the ability of the algorithm to maintain the original relative order of elements of an array with the same value (key) after sorting. To note, certain sorting algorithms are naturally stable, such as Merge Sort, however some are not naturally stable like Quick Sort.

2 situations why stability in sorting is desired:

1st situation: Preserving the same relative order for data with meaningful order for students' names and exam scores.
Unsorted dataset: (Ben, 40), (Eric, 80), (Oscar, 80), (Rachel, 100).
Sorted by score: (Rachel, 100), (Eric, 80), (Oscar, 80), (Ben, 40).
In a stable sort, items with equal keys will maintain the same relative order as in the unsorted/original dataset. In this example, in the unsorted dataset, Eric and Oscar were in the order of (Eric, 80), (Oscar, 80). When sorted based on scores, even though Eric and Oscar have the same score, they appear in the same order as in the unsorted dataset: (Eric, 80), (Oscar, 80). If an unstable sorting algorithm were used, Eric and Oscar could swap positions in the sorted dataset.
Stability in sorting is important in this situation because it preserves the original relative order even after sorting, which is crucial for maintaining consistency in the data or preventing the disruption of a meaningful order. A meaningful order could imply that the students were already sorted by name, and if we sort by score afterward while keeping the name order intact, a stable sort is preferable.

2nd situation: Preserving the order in time sensitive data like flights.
Unsorted dataset: (Plane A, 360 minutes, 8:00 AM), (Plane B, 180 minutes, 9:00 AM), (Plane C, 180 minutes, 9:30 AM), (Plane D, 240 minutes, 10:00 AM)
Sorted by flight duration: (Plane B, 180 minutes, 9:00 AM), (Plane C, 180 minutes, 9:30 AM), (Plane D, 240 minutes, 10:00 AM), (Plane A, 360 minutes, 8:00 AM)
In this example, the dataset was originally ordered based on departure time, from earliest to latest. In a stable sort, flights with the same duration will preserve their original relative order from the unsorted dataset. This ensures that the planes, which were originally sorted by departure time, maintain their order even after sorting by flight duration.
This is shown by Plane B and Plane C, both of which have the same flight duration (180 minutes). Since Plane B has an earlier departure time (9:00 AM) than Plane C (9:30 AM), Plane B stays before Plane C in the sorted dataset, preserving the original time-based order.
Stability is crucial in this situation because it maintains the original departure time order when sorting by a secondary key. This is important in situations where tracking the sequence of events, such as plane departures, is crucial. If an unstable sorting algorithm were used, the departure order could be disrupted.

Stable algorithm: Merge Sort

Explanation of code:
First we have the merge function. This function merges 2 subarrays of the array of dataset. The first subarray is left to middle whereas the second subarray is middle+1 to right.
Next we have the mergeSort function. This will recursively divide the arrays into 2 halves by using the merge function until it cannot be divided any further.

```
#include <iostream>
void merge(int arr[], int left, int middle, int right)
{
    int i, j, k;
    int n1 = middle - left + 1;
    int n2 = right - middle;

    int L[n1], R[n2];

    for (i = 0; i < n1; i++)
    {
        L[i] = arr[left + i];
    }
    for (j = 0; j < n2; j++)
    {
        R[j] = arr[middle + 1 + j];
    }

    i = 0;
    j = 0;
    k = left;

    while (i < n1 && j < n2)
    {
        if (L[i] <= R[j])
        {
            arr[k] = L[i];
            i++;
        }
        else
        {
            arr[k] = R[j];
            j++;
        }
            k++;
    }
```

```
    while (i < n1)
    {
        arr[k] = L[i];
        i++;
        k++;
    }

    while (j < n2)
    {
        arr[k] = R[j];
        j++;
        k++;
    }
}

void mergeSort(int arr[], int left, int right)
{
    if (left < right)
    {
        int middle = left + (right - left) / 2;

        mergeSort(arr, left, middle);
        mergeSort(arr, middle + 1, right);
        merge(arr, left, middle, right);
    }
}
```

How does it show that it is stable:

```
while (i < n1 && j < n2)
{
    if (L[i] <= R[j])
    {
        arr[k] = L[i];
        i++;
    }
    else
    {
        arr[k] = R[j];
        j++;
    }
    k++;
}
```

This part of the merge sort code showcases the stability. The condition L[i] <= R[j] preserves the original order of the dataset. This checks if the values on the left and right are equal and if the element in the left array is placed in the array first. This is to guarantee that in the situation the values are the same, when sorting, the original order is preserved.

Unstable algorithm: Quick Sort

Explanation of code:
First we have the partition function, this function divides the array into halves based on the pivot element.
The quickSort function comes next, and it manages the recursive construction of a quick sort algorithm. This demonstrates the divide and conquer algorithm, which partitions the array recursively around the pivot element in order to sort it.

```
int partition(int arr[], int low, int high)
{
    int i = low;
    int j = high;
    int pivot = arr[low];
    while (i < j)
    {
        while (pivot >= arr[i]) i++;
        while (pivot < arr[j]) j--;
        if (i < j)
        {
            std::swap(arr[i], arr[j]);
        }
    }
    std::swap(arr[low], arr[j]);
    return j;
}

void quickSort(int arr[], int low, int high)
{
    if (low < high)
    {
        int pivot = partition(arr, low, high);
        quickSort(arr, low, pivot - 1);
        quickSort(arr, pivot + 1, high);
    }
}
```

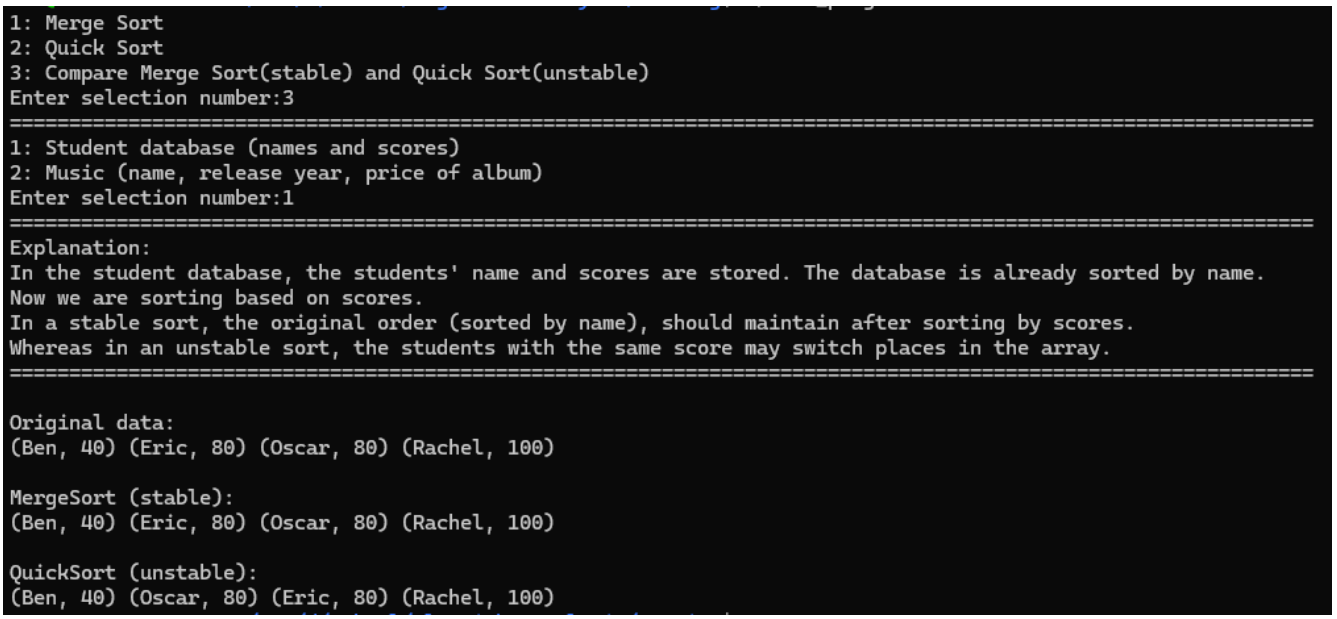
How does it show that it is unstable:

```
int i = low;
int j = high;
int pivot = arr[low];
while (i < j)
{
    while (pivot >= arr[i]) i++;
    while (pivot < arr[j]) j--;
    if (i < j)
    {
        std::swap(arr[i], arr[j]);
    }
}
std::swap(arr[low], arr[j]);
return j;
```

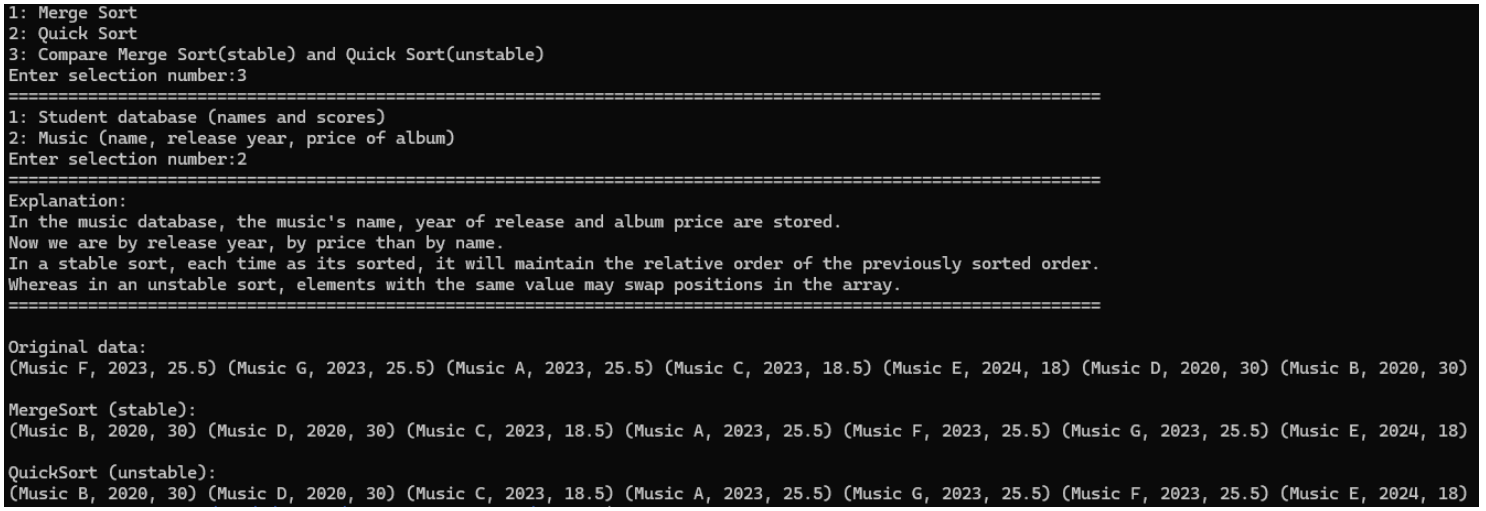
Unstable sorting can cause elements with equal values to change places in the sorted array, preserving neither the original order of the dataset nor its original arrangement.
Basic partitioning is used in the code, items with values greater than the pivot are shifted aside, and elements with values less than or equal to the pivot are shifted to the opposite side. During this, elements with equal values may be swapped and not preserve the original order. This is what causes the instability.

Test cases:
To showcase a merge sort having a stable algorithm and quicksort having an unstable algorithm, some test cases we could use are the dataset of students' names and exam scores, musics' names, release year and prices of albums (handling multi-key).

Students' names and exam scores



Musics' names, release year and prices of albums (multi-key)



Conclusion:
As demonstrated in the two test cases, Merge Sort, which is a stable algorithm, preserves the original order of elements with equal values after sorting. On the other hand, Quick Sort, which is an unstable algorithm, may swap the positions of elements with the same value, changing their original order.

In the student dataset test case, a stable sorting algorithm should preserve the original order (which is based on names) when elements with the same value are sorted. For instance, in

the original dataset, Eric and Oscar both have a score of 80, and Eric comes before Oscar when sorted by name. When using Merge Sort, which is stable, sorting by scores maintains the original order of names, so Eric remains before Oscar. However, with Quick Sort, which is unstable, Oscar and Eric swap positions, disrupting the original order by name. This showcases that Merge Sort is a stable sorting algorithm and Quick Sort is an unstable sorting algorithm.

In the music dataset test case, it will be sorted by release year, by price of album then lastly name. In a stable sorting algorithm, as the sorting progresses through different categories (release year, price, and name), the original order from the previous sort should be preserved for elements with equal values. For instance, in the original dataset, Music F, Music G and Music A have the same release year and price. After being sorted based on those categories, when Merge Sort is used, the relative order of the music is preserved. However, when using Quick Sort, the order was disrupted as seen with Music G and Music F which positions were swapped.

Through these two test cases, it is evident that Merge Sort is a stable sorting algorithm, while Quick Sort is an unstable one. When choosing a preferred sorting algorithm, it is advisable to choose a stable algorithm, especially in situations where maintaining the original or relative order is important for data consistency

Analysis of the efficiency of Merge Sort algorithm using the Big-oh notation

The time complexity of a Merge Sort is O(n lg n) in the worst, average and the best case. For best case it is when the array has already been sorted or close to being sorted. For the average case, it is when the array is randomly ordered. For the worst case, it is when the array is sorted in reverse order.
In the algorithm, it consists of functions merge and mergeSort. mergeSort function recursively divides the array into subarrays then merges them back together whereas the merge function merges the 2 subarrays into a single sorted array. For merge function the time complexity is O n and it is called O(log n) times recursively. mergeSort function has a time complexity of O(n log n).
With the same time complexity throughout the three cases. It shows that it operates consistently and reliably in a variety of situations. Additionally with both functions having a time complexity of O(n log n), it showcases that it scales competently with large datasets due to its logarithmic component.

Analysis of the efficiency of Quick Sort algorithm using the Big-oh notation

The time complexity of Quick Sort for any partitioning scenario where T(n) is the time complexity for sorting n elements is T(n) = T(k) + T(n-k-1) + θ(n), where T(k) is the time complexity to sort the left sub-array which is of size k and T(n-k-1) is the time complexity to sort the right sub-array which is of size n-k-1.
The best, average and worst case scenarios are dependent on the value of size k.
The best case scenario is when the array is divided into 2 equal halves by the pivot (partitioning is nearly equally balanced), so that k approximates to n/2, the average case

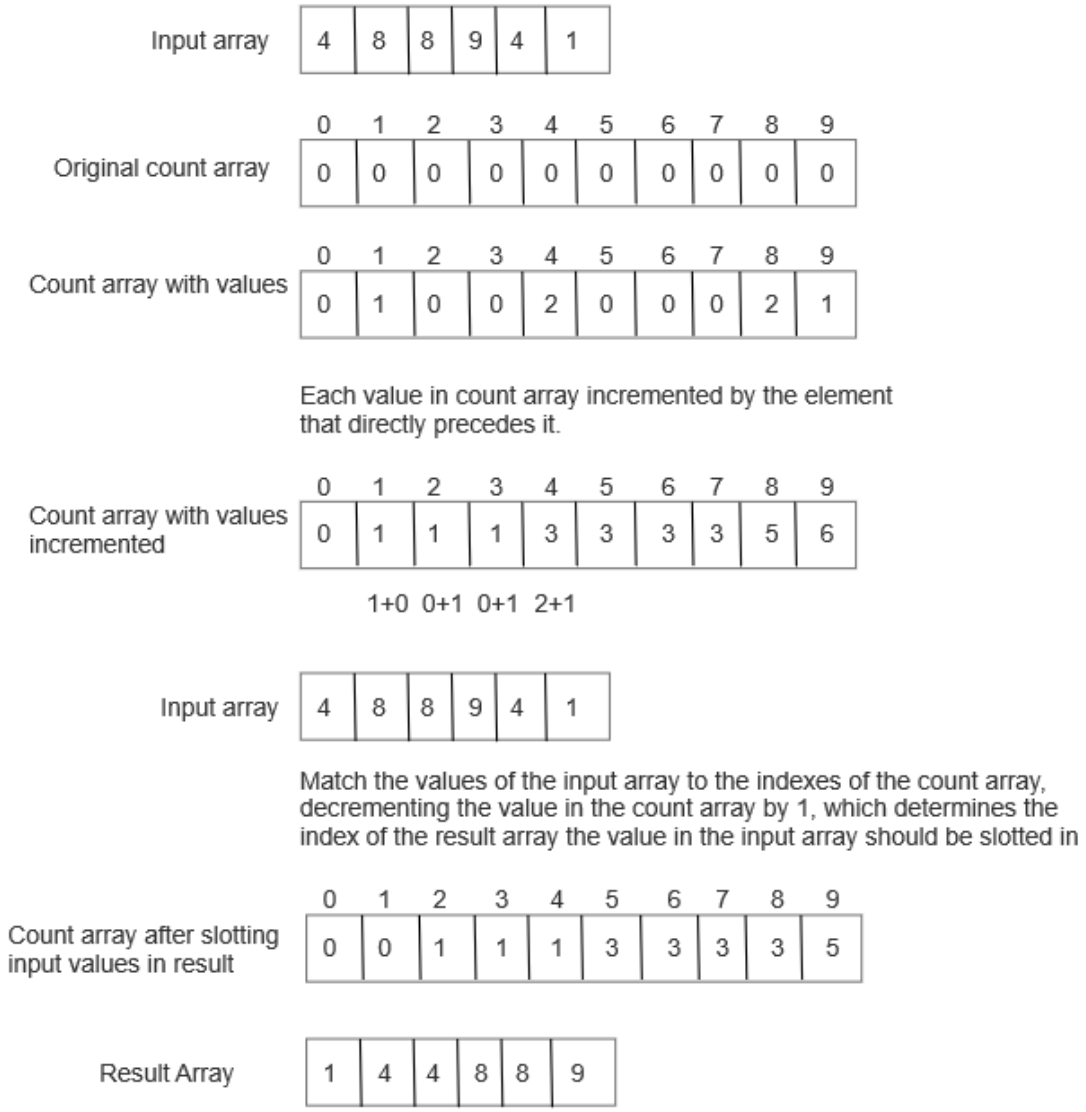
scenario is when the array is divided into approximately equal halves (partitioning may be slightly unbalanced) so that k also approximates to n/2. In both cases, $T(n) = 2T(n/2) + \theta(n)$, or O(n lg n). In the worst case scenario, k = 0 or k = n-1 where the pivot is either the smallest or largest element. In this case, the time complexity is $T(n)=T(0)+T(n-1)+\theta(n)$ or $O(n^2)$. For best and average case, it is considered efficient when the pivot is well selected, it also provides fast performances. However, unlike merge sort, in the worst case, quicksort is inefficient and performs much slower.

c) Most of the sorting algorithms have a $O(n^2)$ complexity. The best algorithm has achieved a O(n lg n) complexity. Research and discuss a sorting algorithm which achieve better than O(n lg n) complexity.

Research states that a sorting algorithm that directly compares the elements in the array have at best O(n lg n) complexity, however, there are existing sorting algorithms like the counting sort algorithm that have the potential of having a better time complexity than O(n lg n).

The counting sort algorithm involves having an additional array, a “count array”, the size of which is the maximum value element in the input array + 1. This “count array” stores the number of times each element appears in the input array.

For example, considering a data range of 0-9 and an input array [4 8 8 9 4 1]. The maximum value element is 9, hence the size of the count array is 9+1 = 10.



The time complexity of the counting sort algorithm is based on the size of the input array (n) + the range of values in the input array (k), O(n+k).

This is because counting the occurrences of each element takes O(n) time and the extra time to handle k is O(k). Hence, counting sort can achieve a better time complexity than O(n log n) when the range of values in the input array (k) is small relative to n, the size of the input array.

For example, consider the input array [2,5,3,0,2,8,3,5,6,8] with data range 0-9, n = 10 (the size of the input array) and k = 10 (data range 0-9). The time complexity would be O(10+10) = O(20). Using a merge sort for the same scenario would have the time complexity O(nlgn), or O(10lg10) = O(33) (approximately).

However, this means that counting sort is also only efficient when the range of values in the input array is small relative to n, the size of the input array. If k is large and the size of the count array is large, the space and time required to manage the count array would be less practical than other sorting algorithms with O(n lg n) time complexity.

For example, if the data range is 0-1000, with the input array being [2,5,3,0,2,8,3,5,6,900], n = 10, k = 1000. The time complexity for counting sort would be O(n+k) = O(10+1000) = O(1010). Merge sort time complexity would remain at O(nlgn) = O(33).