Experiment **#1**		Student ID	
Date		Student Name	[@KLWKS_BOT THANOS]

Experiment Title: Analysis of Algorithm based on Arrays

Aim/Objective: To understand the concept and implementation of Basic programs on arrays.

Description: The students will understand and able to implement programs on Arrays.

Pre-Requisites:

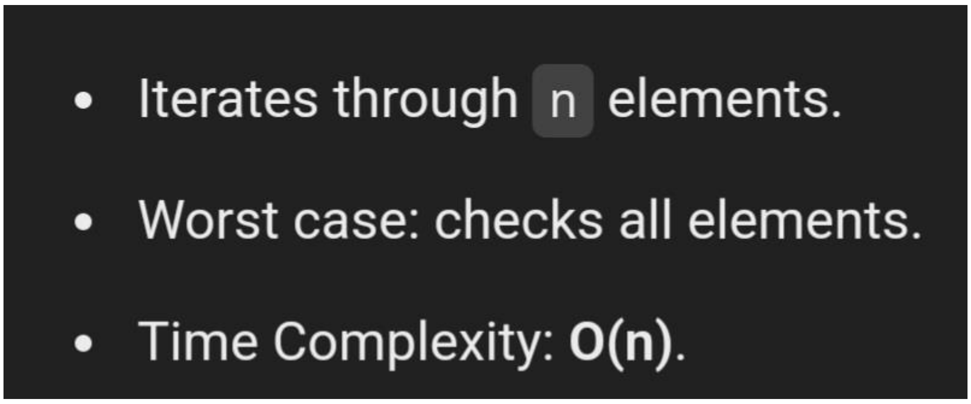
Knowledge: Arrays **Tools**: Code Blocks/Eclipse IDE

Pre-Lab:

1. Calculate the Time Complexity of the following code snippet:

```
Algorithm linear_search(arr, target):
for i in range(len(arr)):
    if arr[i] == target:
        return i;
return -1;
```

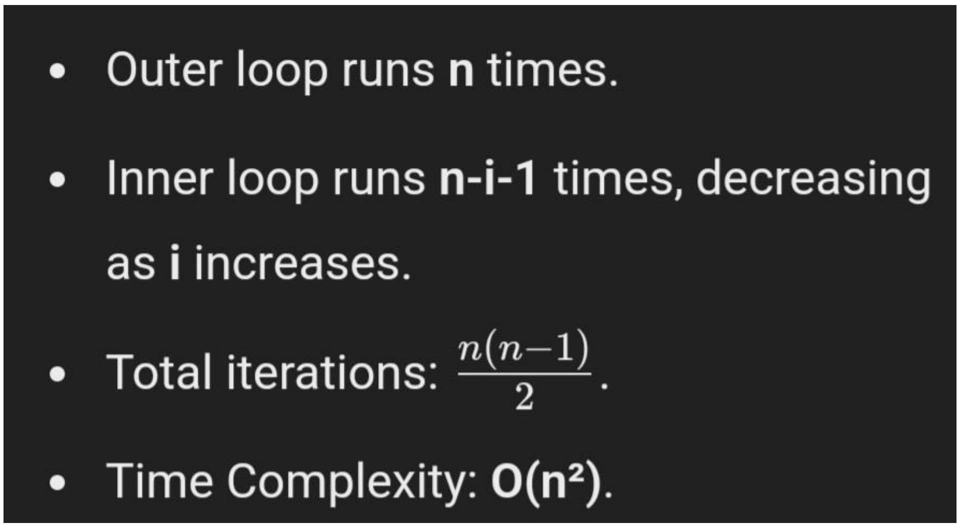
- **Procedure**:



- Iterates through n elements.
- Worst case: checks all elements.
- Time Complexity: **O(n)**.

```
2. Algorithm function(arr):
  n = len(arr)
  for i in range(n):
      for j in range(0, n-i-1):
          if arr[j] > arr[j+1]:
              arr[j], arr[j+1] = arr[j+1], arr[j]
  return arr;
```

- **Procedure**:



- Outer loop runs **n** times.
- Inner loop runs **n-i-1** times, decreasing as **i** increases.
- Total iterations: $\frac{n(n-1)}{2}$.
- Time Complexity: **O(n^2)**.

3. The median of a list of numbers is essentially its middle element after sorting. The same number ofelements occur after it as before. Given a list of numbers with an odd number of elements, find the median? Also find its time complexity.

Example arr = [5, 3, 1, 2, 4]

The sorted array a’ = [1, 2, 3, 4, 5]. The middle element and the median is 3.

Description: Write an algorithm *findMedian* **with the following parameter(s):**

arr[n]: an unsorted array of integers

Returns: median of the array

Sample Input

7

0 1 2 4 6 5 3

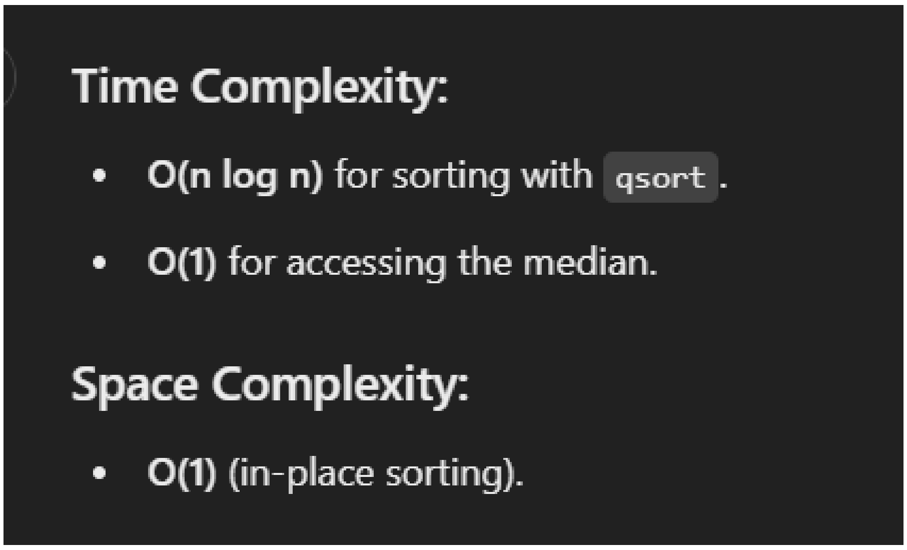
Sample Output

3

- **Procedure/Program**:

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

int compare(const void *a, const void *b) {
    return (*(int *)a - *(int *)b);
}



int findMedian(int arr[], int n) {
    qsort(arr, n, sizeof(int), compare);
    return arr[n / 2];
}
```

```
int main() {
   int arr[] = {0, 1, 2, 4, 6, 5, 3};
   int n = sizeof(arr) / sizeof(arr[0]);
   printf("Median: %d\n", findMedian(arr, n));
   return 0;
}
```



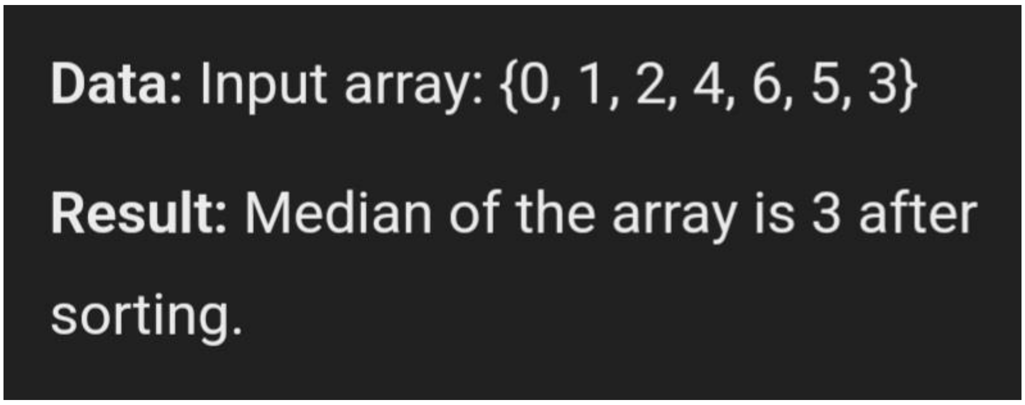
Time Complexity:

- **O(n log n)** for sorting with `qsort`.
- **O(1)** for accessing the median.

Space Complexity:

- **O(1)** (in-place sorting).

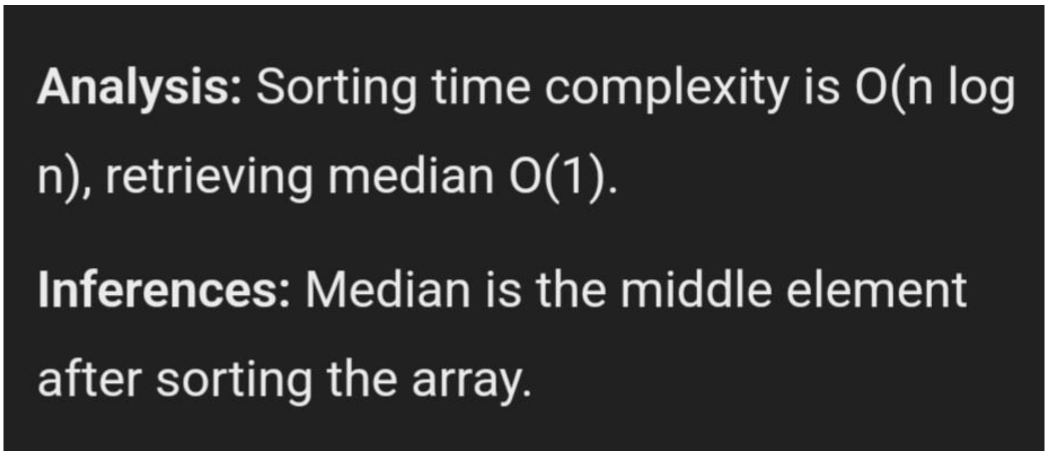
- **Data and Results**:



Data: Input array: {0, 1, 2, 4, 6, 5, 3}

Result: Median of the array is 3 after sorting.

- **Analysis and Inferences**:



Analysis: Sorting time complexity is O(n log n), retrieving median O(1).

Inferences: Median is the middle element after sorting the array.

Experiment **#1**		Student ID	
Date		Student Name	[@KLWKS_BOT THANOS]

In-Lab:

Given an array of strings **arr[]**, the task is to sort the array of strings according to frequency of each string, in ascending order. If two elements have same frequency, then they are to be sorted in alphabetical order.

Input: arr[] = {“Ramesh”, “Mahesh”, “Mahesh”, “Ramesh”}

Output: {“Mahesh”, ”Ramesh”}

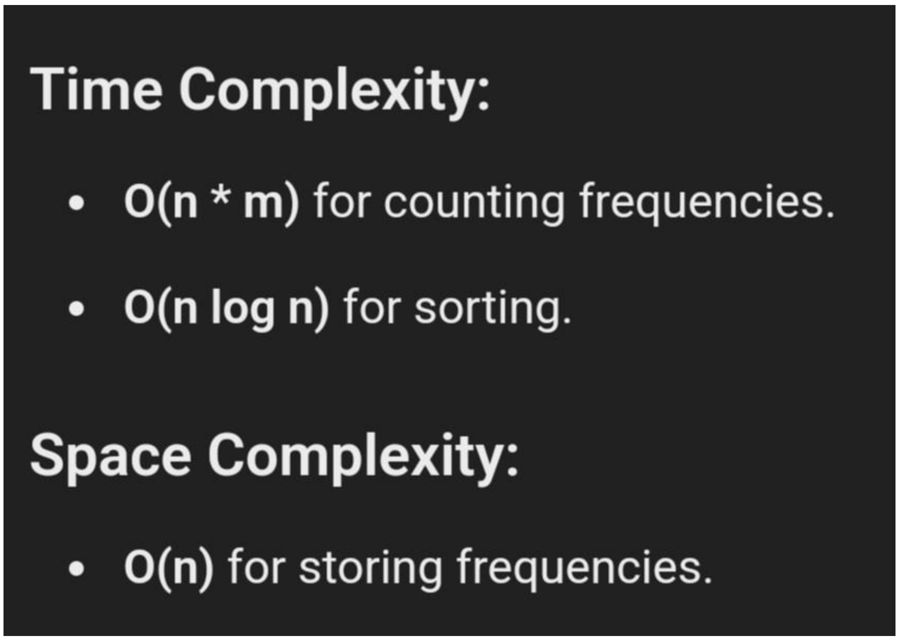
Explanation: As both the strings have the same frequency, the array is sorted in alphabetical order.

Find the time and space complexity for the procedure/Program.

- **Procedure/Program:**

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

typedef struct {
   char str[100];
   int freq;
} StringFreq;

int compare(const void *a, const void *b) {
   StringFreq *x = (StringFreq *)a, *y = (StringFreq *)b;
   return x->freq != y->freq ? x->freq - y->freq : strcmp(x->str, y->str);
}

int main() {
   char arr[][100] = {"Ramesh", "Mahesh", "Mahesh", "Ramesh"};
   int n = 4, count = 0;
   StringFreq freqArr[n];

   for (int i = 0; i < n; i++) {
      int found = 0;
      for (int j = 0; j < count; j++) {
         if (strcmp(freqArr[j].str, arr[i]) == 0) {
```

```
 freqArr[j].freq++;
          found = 1;

  break;
       }
     }
     if (!found) {
        strcpy(freqArr[count].str, arr[i]);
        freqArr[count++].freq = 1;
     }
   }

   qsort(freqArr, count, sizeof(StringFreq), compare);

   for (int i = 0; i < count; i++)
      printf("%s ", freqArr[i].str);
   printf("\n");
   return 0;
}
```



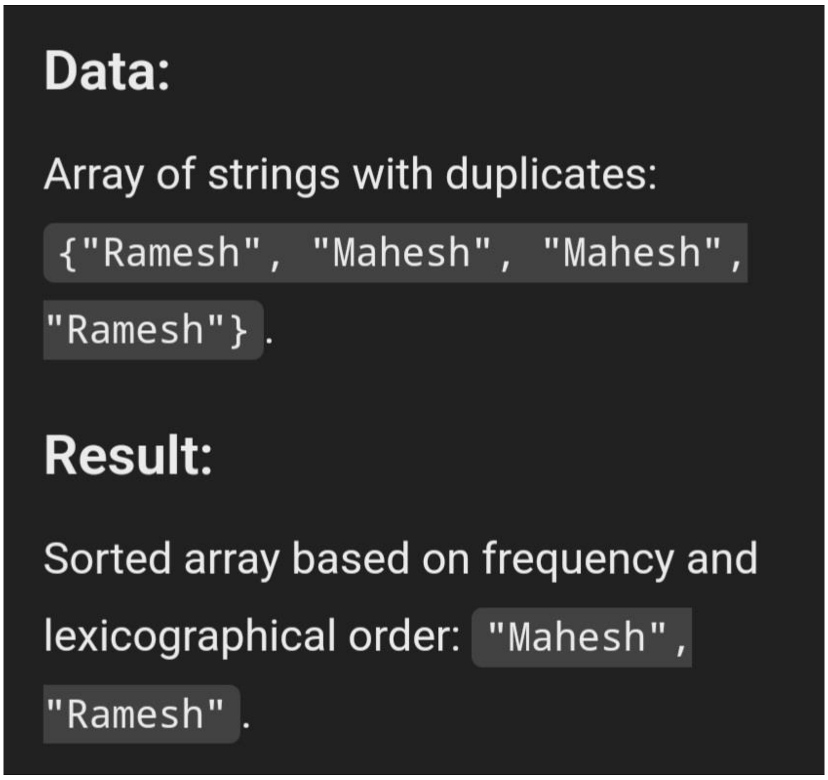
Time Complexity:

- **O(n * m)** for counting frequencies.
- **O(n log n)** for sorting.

Space Complexity:

- **O(n)** for storing frequencies.

- **Data and Results:**



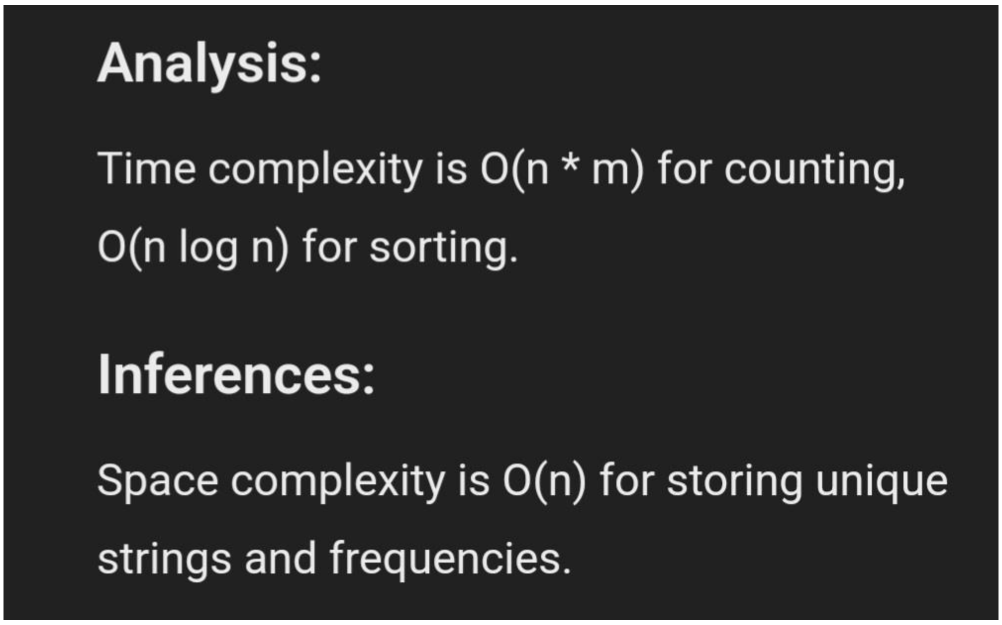
Data:

Array of strings with duplicates: `{"Ramesh", "Mahesh", "Mahesh", "Ramesh"}`.

Result:

Sorted array based on frequency and lexicographical order: `"Mahesh"`, `"Ramesh"`.

- **Analysis and Inferences:**



Analysis:

Time complexity is O(n * m) for counting, O(n log n) for sorting.

Inferences:

Space complexity is O(n) for storing unique strings and frequencies.

Post-Lab:

Given an array of strings **words []** and the **sequential order** of alphabets, our task is to sort the array according to the order given. Assume that the dictionary and the words only contain lowercase alphabets.

Input: words = {“word”, “world”, “row”},

Order= “worldabcefghijkmnpqstuvxyz” Output: “world”, “word”, “row”

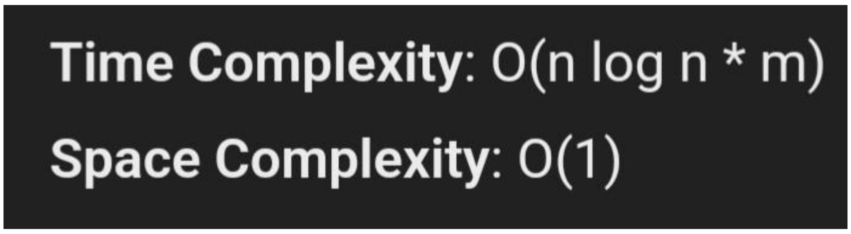
Explanation: According to the given order ‘l’ occurs before ‘d’ hence the words “world” will be kept first.

Find the time and space complexity for the procedure/Program.

- **Procedure/Program**:

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

int charToIndex(char c, char *order) {
   for (int i = 0; i < strlen(order); i++) {
      if (order[i] == c) {
         return i;
      }
   }
   return -1;
}

int compare(const void *a, const void *b) {
   char *word1 = *(char **)a;
   char *word2 = *(char **)b;

   int len1 = strlen(word1);
   int len2 = strlen(word2);
   int minLen = len1 < len2 ? len1 : len2;

   for (int i = 0; i < minLen; i++) {
      int index1 = charToIndex(word1[i], "worldabcefghijkmnpqstuvxyz");
```

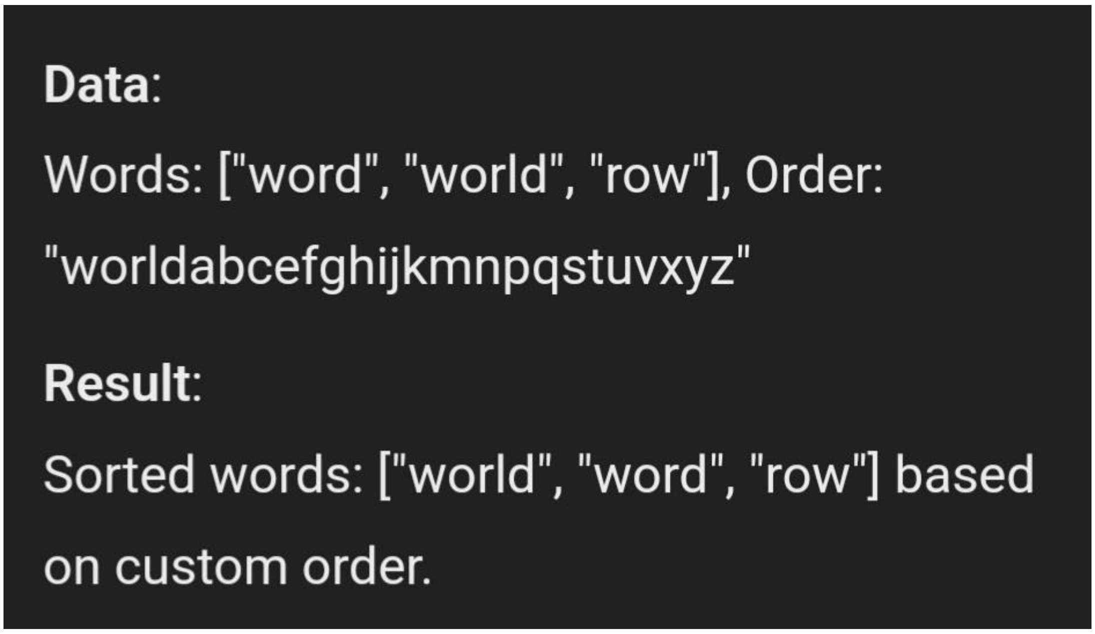
```
        int index2 = charToIndex(word2[i], "worldabcefghijkmnpqstuvxyz");

        if (index1 < index2) {
            return -1;
        } else if (index1 > index2) {
            return 1;
        }
    }

    if (len1 < len2) {
        return -1;
    } else if (len1 > len2) {
        return 1;
    }
    return 0;
}

int main() {
    char *words[] = {"word", "world", "row"};
    int n = sizeof(words) / sizeof(words[0]);

    qsort(words, n, sizeof(char *), compare);

    for (int i = 0; i < n; i++) {
        printf("%s ", words[i]);
    }
    return 0;
}
```



Time Complexity: O(n log n * m)

Space Complexity: O(1)

- **Data and Result:**



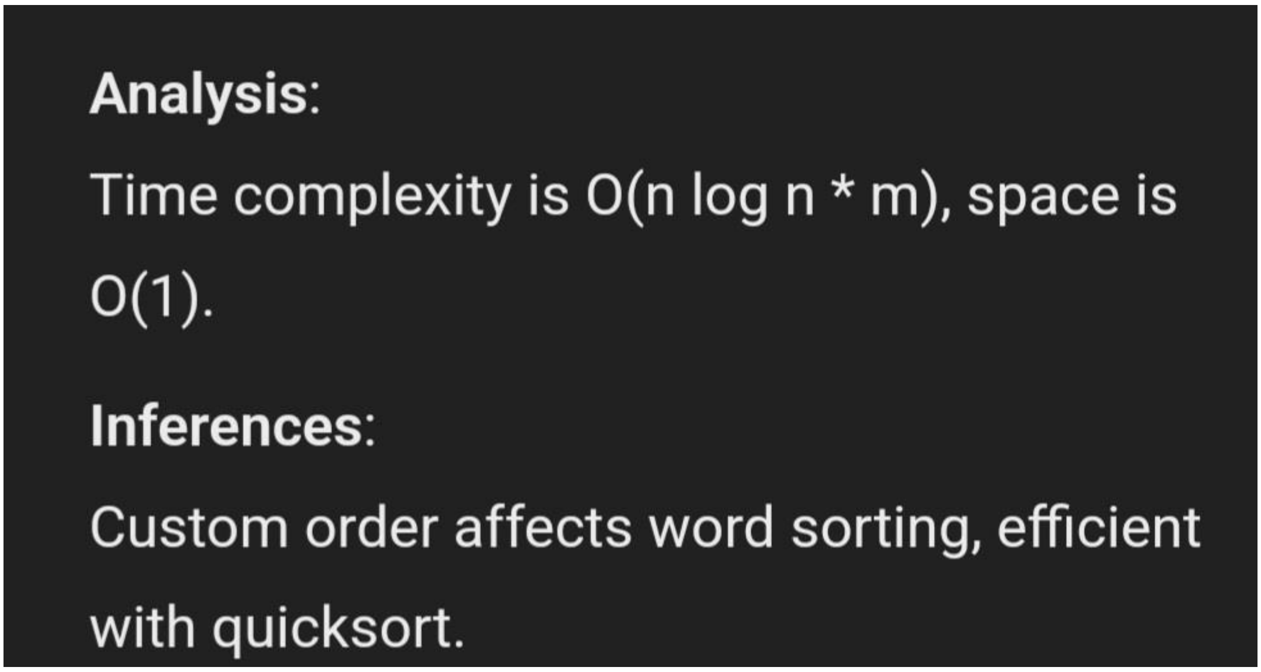
Data:

Words: ["word", "world", "row"], Order: "worldabcefghijkmnpqstuvxyz"

Result:

Sorted words: ["world", "word", "row"] based on custom order.

- **Analysis and Inferences:**



Analysis:

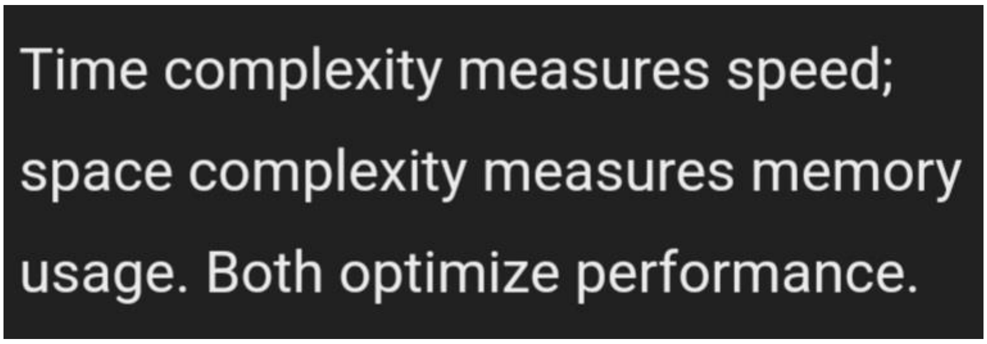
Time complexity is O(n log n * m), space is O(1).

Inferences:

Custom order affects word sorting, efficient with quicksort.

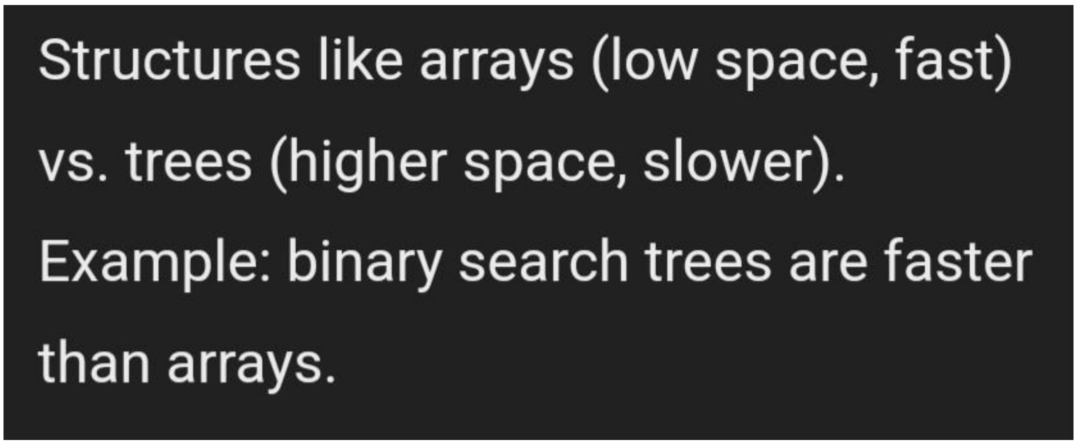
- **Sample VIVA-VOCE Questions (In-Lab):**

1. What is the significance of analyzing both time complexity and space complexity when evaluating algorithms?



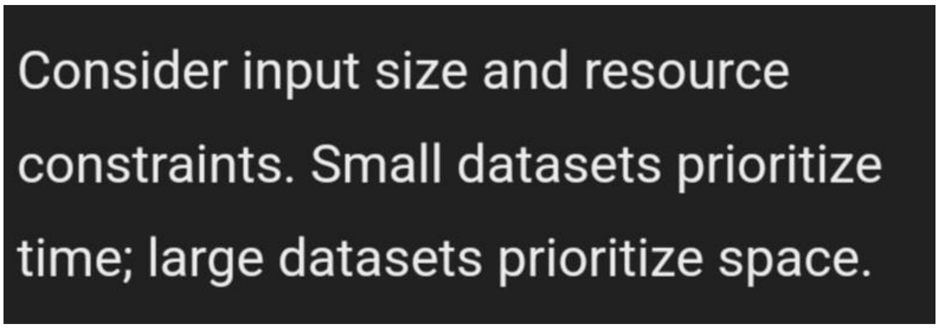
Time complexity measures speed; space complexity measures memory usage. Both optimize performance.

2. How does the choice of data structures impact both time and space complexity in algorithm implementations? Give examples.



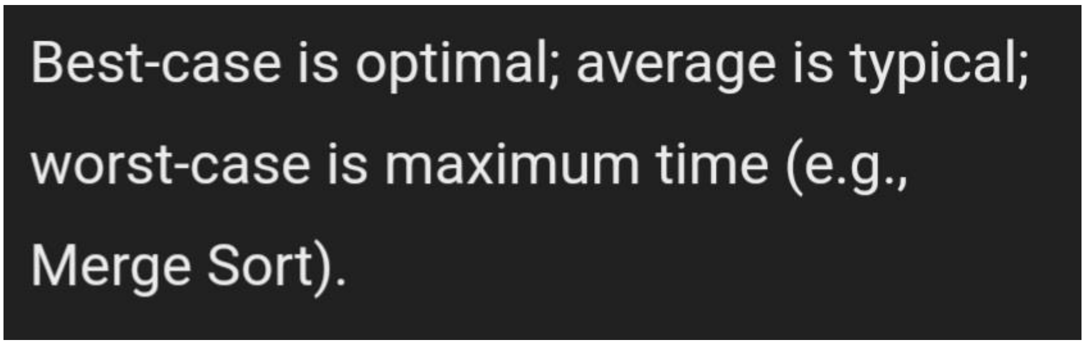
Structures like arrays (low space, fast) vs. trees (higher space, slower). Example: binary search trees are faster than arrays.

3. When comparing two algorithms with different time and space complexities, how do you decide which one is more suitable for a particular problem or application?



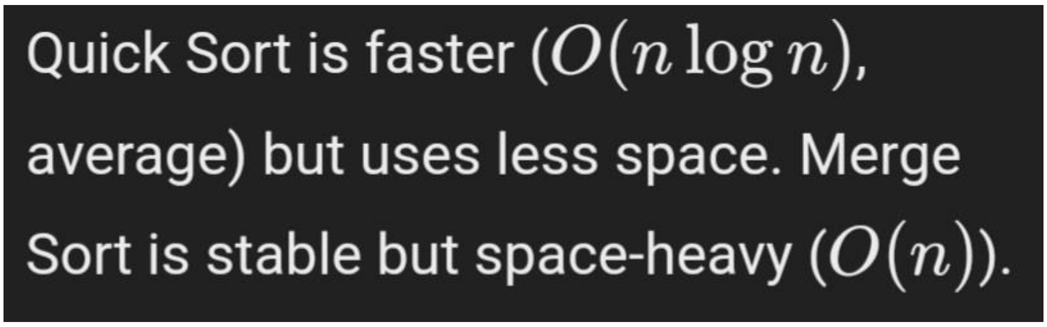
Consider input size and resource constraints. Small datasets prioritize time; large datasets prioritize space.

4. What is the difference between best-case, average-case, and worst-case time complexity? Provide examples.



Best-case is optimal; average is typical; worst-case is maximum time (e.g., Merge Sort).

5. In the context of sorting algorithms, compare the time and space complexities of algorithms like Quick Sort and Merge Sort. Which one is more time-efficient, and which one is more space-efficient?



Quick Sort is faster ($O(n \log n)$, average) but uses less space. Merge Sort is stable but space-heavy ($O(n)$).

Evaluator Remark (if Any):	**Marks Secured___ out of 50**
	Signature of the Evaluator with Date

Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment.