| Experiment **#2** | | Student ID | |
|---|---|---|---|
| Date | | Student Name | [@KLWKS_BOT THANOS] |

**Experiment Title:** Analyzing Algorithm Performance: Time and Space Complexity

**Aim/Objective:** To understand the concept of time and space complexity.

**Description:** The students can understand and analyzing the performance of algorithms in terms of time and space complexity.

**Pre-Requisites:**

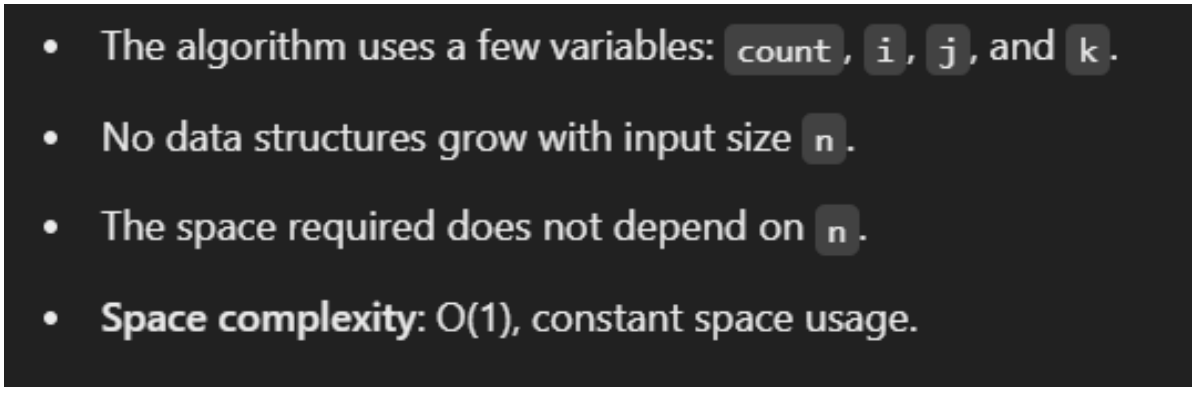
Knowledge: Strings in C/C++/Python Tools: Code Blocks/ IDE

**Pre-Lab:**

1. During the lockdown, Mothi stumbled upon a unique algorithm while browsing YouTube. Although he's enthusiastic about algorithms, he struggles with those involving multiple loops. Now, he's seeking your assistance to determine the space complexity of this algorithm. Can you help him?

```
Algorithm KLU(int n) {
count ← 0
for i ← 0 to n - 1 step i = i * 2 do
   for j ← n down to 1 step j = j / 3 do
         for k ← 0 to n - 1 do    count ← count + 1
         end for
    end for
 end for
return count   }
```

- **Procedure:**



2. Klaus Michaelson, an interviewer, has compiled a list of algorithm-related questions for students. Among them, one of the most basic questions involves determining the time complexity of a given algorithm. Can you determine the time complexity for this algorithm?

```
Algorithm ANALYZE( n)  {
   if n == 1 then    return 1
   else  eturn ANALYZE(n - 1) + ANALYZE(n - 1)
   end if
}
```

- **Procedure:**

- The algorithm makes two recursive calls for each `n`.
- Each call reduces `n` by 1, leading to binary recursion.
- The recurrence relation is T(n) = 2T(n - 1) + O(1).
- The number of calls grows exponentially, proportional to $2^n$.
- **Time complexity**: O(2^n).

**In-Lab:**

Caroline Forbes is known for her intelligence and problem-solving skills. To test her abilities, you've decided to present her with a challenge: sorting an array of strings based on their string lengths. If you're up for the challenge, see if you can solve this problem faster than her!

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

**Input**: You are extraordinarily talented, displaying a wide range of skills and abilities

**Output**: a of You are and wide range skills talented abilities displaying extraordinarily

- **Procedure/Program:**

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

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

int main() {
   char *arr[] = {
      "You", "are", "extraordinarily", "talented", "displaying",
      "a", "wide", "range", "of", "skills", "and", "abilities"
   };


   int n = sizeof(arr) / sizeof(arr[0]);

   qsort(arr, n, sizeof(char *), compare);
```

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

    return 0;
}
```

**Time Complexity:**

- **O(n log n * k)**, where `n` is the number of strings and `k` is the average length.

**Space Complexity:**

- **O(n)**, for storing the array of strings.

- **Data and Results:**

**Data:**

Array of strings with varying lengths to be sorted.

**Result:**

Sorted array of strings based on their lengths printed.

- **Analysis and Inferences:**

**Analysis:**

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

**Inferences:**

Sorting by length improves arrangement; space and time grow linearly.

**Post-Lab:**

Given an array **arr[]** containing N strings, the objective is to arrange these strings in ascending order based on the count of uppercase letters they contain. Find the time complexity for the algorithm/program.

Input arr[] = {"Hello", "WORLD", "abc", "DEFGH", "Testing"}

Output: {"abc", "Hello", "Testing", "WORLD", "DEFGH"}

**Procedure/Program:**

## Algorithm:

1. Input: An array `arr[]` of `N` strings.
2. Count uppercase letters: For each string, count uppercase letters using `countUppercase(str)`.
3. Sort strings: Sort strings based on the uppercase letter count using a comparison function.
4. Output: Print the sorted array.

Input:

arduino Copy code

```
arr[] = {"Hello", "WORLD", "abc", "DEFGH", "Testing"}
```

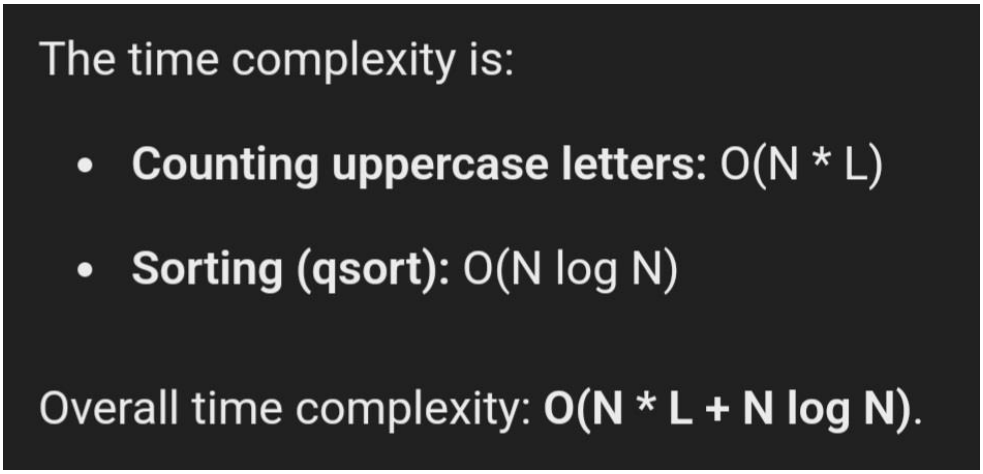
Output:

Copy code

```
abc Hello Testing WORLD DEFGH
```

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

int countUppercase(const char str[]) {
    int count = 0;
    for (int i = 0; str[i] != '\0'; i++) {
        if (isupper(str[i])) {
            count++;
        }
    }
    return count;
}

int compare(const void *a, const void *b) {
    int countA = countUppercase(*(const char **)a);
    int countB = countUppercase(*(const char **)b);
    return countA - countB;
}

int main() {
    char *arr[] = {"Hello", "WORLD", "abc", "DEFGH", "Testing"};
    int n = sizeof(arr) / sizeof(arr[0]);

    qsort(arr, n, sizeof(char *), compare);
```

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



- **Data and Results:**

**Data:**

Input: `arr[] = {"Hello", "WORLD", "abc", "DEFGH", "Testing"}`.

**Result:**

Output: `abc Hello Testing WORLD DEFGH`.

- **Analysis and Inferences:**

**Analysis:**

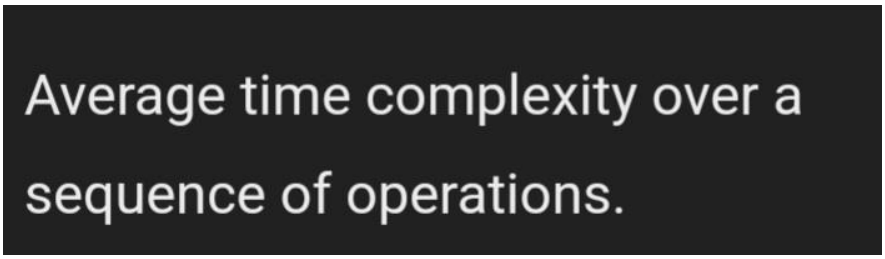
Time complexity: O(N * L + N log N).

**Inferences:**

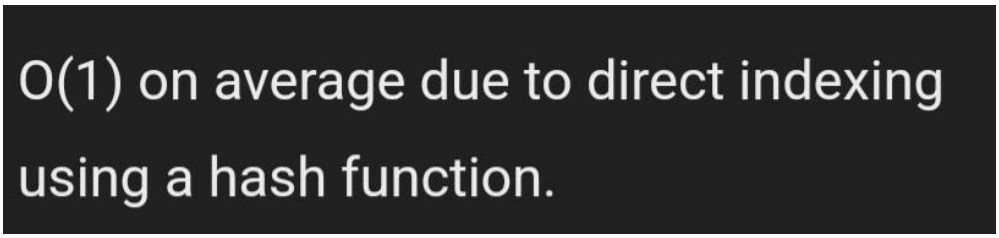
Uppercase count and sorting determine final string order.

- **Sample VIVA-VOCE Questions :**

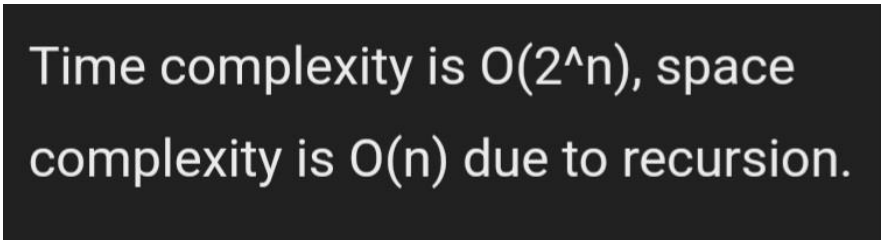
1. What do you understand by amortized analysis in the context of algorithm complexity?




2. Why is the complexity of searching for an element in a hash table O(1) on average?




3. What is the time complexity of the Tower of Hanoi problem, and how does its recursive nature affect its space complexity?



4. How can hashing improve the time complexity of search operations?

Improves search time by directly accessing data with a hash function.

| **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.**