WEEK-2:

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/Program:

IN C:

```
// Function to calculate KLU with input parameter 'n'
int KLU(int n) {
  int count = 0;  // Initialize a variable 'count' to 0. (Time: O(1), Space: O(1))
  // Outer loop with a doubling step
  for (int i = 0; i <= n - 1; i = i * 2) {
    // The outer loop runs logarithmically with base 2. (Time: O(log n), Space: O(1))
    // Middle loop with a decrementing step of j/3
    for (int j = n; j >= 1; j = j / 3) {
      // The middle loop runs logarithmically with base 3. (Time: O(log n), Space: O(1))
      // Inner loop iterating from 0 to n - 1
      for (int k = 0; k <= n - 1; k++) {
        // The inner loop runs linearly with respect to n. (Time: O(n), Space: O(1))
        count = count + 1;  // Increment 'count' for each iteration of the inner loop
      }
    }
  }
  return count;  // Return the final 'count' after all the loops. (Time: Depends on nested loops, Space: O(1))
}
```

The space complexity is constant (O(1))

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
 return ANALYZE(n - 1) + ANALYZE(n - 1)
end if
}
```

- Procedure/Program:

IN C:

```
// Function to analyze the given algorithm with input parameter 'n'
int ANALYZE(int n) {
  if (n == 1) {
    return 1;  // Base case: If n is 1, return 1. (Time: O(1))
  } else {
    // Recursive case: Sum of two recursive calls.
    // Time complexity: T(n) = 2 * T(n - 1)
    return ANALYZE(n - 1) + ANALYZE(n - 1);
  }
}
```

Time Complexity:

- The base case has a constant time complexity (O(1)).
- The recursive case has an exponential time complexity (O(2^n)).

- Therefore, the overall time complexity of the ANALYZE algorithm is dominated by the recursive case, resulting in (O(2^ n). This indicates an inefficient algorithm for large values of n.

3. You have two algorithms along with their respective time functions, and you need assistance in determining which one has a higher time complexity. Your task is to analyze these time functions and justify which one grows at a faster rate in a straightforward manner.
a) T1(n) = (log(n^2)*log(n)) T2(n) = log(n*(logn)^10)
b) T1(n) = 3*(n)^(sqrt(n)) T2(n) = (2)^(sqrt(n)*logn)
(consider logn with base 2)
Procedure/Program:

a)
```
#include <stdio.h>

#include <math.h>

int main() {

   int n = /* some value */

   // Algorithm A

   int T1 = (log2(n * n) * log2(n));

   // Algorithm B

   int T2 = log2(n * pow(log2(n), 10));

   // Print the results

   printf("T1(n) = %d\n", T1);

   printf("T2(n) = %d\n", T2);

   // Compare growth rates

   if (T1 < T2) {

      printf("Algorithm A has a lower time complexity.\n");

   } else if (T1 > T2) {

      printf("Algorithm B has a lower time complexity.\n");

   } else {

      printf("Both algorithms have the same time complexity.\n");

   }


   return 0;

}
```

Explanation of Time Complexity:

Algorithm A (T1 = (log2(n * n) * log2(n))):

The time complexity of Algorithm A is determined by the term.$\log_2(n^2)\cdot\log_2(n)$

Therefore, the overall time complexity of Algorithm A is $O(\log_2(n))$.

b)Algorithm B (T2 = log2(n * pow(log2(n), 10))):

The time complexity of Algorithm B is determined by the term $\log_2(n\cdot(\log_2(n))^{10})$.

Therefore, the dominant term is$(\log_2(n))^{10}$.

The overall time complexity of Algorithm B is$O((\log_2(n))^{10})$.

In-Lab:

1) 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: and of You are wide range skills and talents displaying extraordinarily
   • Procedure/Program:
   IN C:

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

// Helper function to swap two strings
void swap(char** a, char** b) {
   char* temp = *a;
   *a = *b;
   *b = temp;
}
//Time Complexity: O(1) - Swapping two pointers is a constant-time operation.
//Space Complexity: O(1) - The function uses a constant amount of space.
// Function to partition the array into two sub-arrays based on the pivot
int partition(char** strings, int low, int high) {
   char* pivot = strings[low + (high - low) / 2]; // Choose the pivot (middle element)
   int i = low;
   int j = high;

   while (i <= j) {
      while (strlen(strings[i]) < strlen(pivot)) {
         i++;
      }
      while (strlen(strings[j]) > strlen(pivot)) {
         j--;
      }if (i <= j) {
         swap(&strings[i], &strings[j]);
         i++;
```

```
            j--;
        }
    }
    return i;
}
//Time Complexity: O(n) - The dominant factor is the linear scan through the array during the
partitioning process.
//Space Complexity: O(1) - The function uses a constant amount of extra space.
// Recursive function to sort the array using quicksort
void quicksort(char** strings, int low, int high) {
    if (low < high) {
        int pi = partition(strings, low, high); // Find the partition index
        quicksort(strings, low, pi - 1); // Recursively sort the left sub-array
        quicksort(strings, pi, high); // Recursively sort the right sub-array
    }
}
//Time Complexity: O(n log n) - On average, quicksort has a logarithmic partitioning process,
resulting in an overall time complexity of O(n log n).
//Space Complexity: O(log n) - The space required for the recursive call stack is logarithmic
on average.

int main() {
    char* input_strings[] = {"You", "are", "extraordinarily", "talented", "displaying", "a",
"wide", "range", "of", "skills", "and", "abilities"};
    int num_strings = sizeof(input_strings) / sizeof(input_strings[0]);
    // Print the input array
    printf("Input: ");
    for (int i = 0; i < num_strings; i++) {
        printf("%s ", input_strings[i]);
    }
    printf("\n");
    // Perform quicksort on the array of strings
    quicksort(input_strings, 0, num_strings - 1);
    // Print the sorted array
    printf("Output: ");
    for (int i = 0; i < num_strings; i++) {
        printf("%s ", input_strings[i]);
    }
    printf("\n");
    return 0;
}
//Time Complexity: O(n log n) - The dominant factor is the quicksort algorithm.
//Space Complexity: O(log n) - The space required for the recursive call stack during the
quicksort process. The space required for the input_strings array is not considered in the
space complexity analysis.
/*Time Complexity:
* The average time complexity of quicksort is O(n log n), where n is the number of elements
in the array.
```

```
* The partition function takes O(n) time in the worst case.
* Quicksort can degrade to O(n^2) in the worst case, but this is unlikely with good pivot selection.*/
/*Space Complexity:
* The space complexity is O(log n) on average for the recursive call stack (due to the partitioning of sub-arrays).
* In the worst case, it can be O(n) if the recursion depth is equal to the number of elements in the array.
* The swap function uses constant space.*/
```

2. Stefan, an Assistant Professor in the Computer Science Department, is dedicated to mentoring students and facilitating their understanding of intricate concepts. Recently, he presented a problem: the task of sorting an array based on the count of set bits. Help him in writing an algorithm/ program and find its runtime. Input: arr[] = {1, 2, 3, 4, 5, 6}; Output: 3 5 6 1 2 4 Explanation:

3 – 0011

5 - 0101

6- 0110

1 – 0001

2 - 0010

4 – 0100

Hence the non-increasing sorted order is {3, 5, 6}, {1, 2, 4}

• Procedure/Program:IN C:

```
#include <stdio.h>

// Explanation: This line includes the standard input-output header file for using functions like printf.
// Time Complexity: N/A - Preprocessor directive, not part of the runtime.

int sum_of_natural_numbers(int n) {
  // Explanation: This line defines a function named sum_of_natural_numbers that takes an integer parameter n.
  // Time Complexity: O(1) - Defining a function involves constant time.

  int sum_n = (n * (n + 1)) / 2;
  // Explanation: This line calculates the sum of the first 'n' natural numbers using the formula n * (n + 1) / 2.
  // Time Complexity: O(1) - The arithmetic operations (multiplication and division) are constant time.

  return sum_n;
  // Explanation: This line returns the calculated sum.
```

```
    // Time Complexity: O(1) - Returning a value from a function is a constant-time operation.
}

int main() {
    // Explanation: This line defines the main function.
    // Time Complexity: O(1) - Defining a function involves constant time.

    int n = 10;
    // Explanation: This line initializes the variable 'n' with a value (you can change it for
different 'n').
    // Time Complexity: O(1) - Assigning a value to a variable is a constant-time operation.

    int result = sum_of_natural_numbers(n);
    // Explanation: This line calls the function to calculate the sum of the first 'n' natural
numbers.
    // Time Complexity: O(1) - Calling a function with constant time complexity.

    printf("The sum of the first %d natural numbers is: %d\n", n, result);
    // Explanation: This line prints the result.
    // Time Complexity: O(1) - Printing is a constant-time operation.

    return 0;
    // Explanation: This line indicates a successful execution of the program.
    // Time Complexity: O(1) - Returning from the main function is a constant-time operation.
}
```

3)During the final skill exam, the teacher presented a challenge to the students and encouraged them to think creatively. The task was to devise an algorithm for calculating the sum of the first 'n' natural numbers, with the aim of finding innovative approaches distinct from their peers.

- Procedure/Program:

```
#include <stdio.h>
// Explanation: This line includes the standard input-output header file for using functions
like printf.
// Time Complexity: N/A - Preprocessor directive, not part of the runtime.

// Function to calculate the sum of the first 'n' natural numbers
int sum_of_natural_numbers(int n) {
    int sum = 0; // Initialize sum to 0
    // Explanation: This line initializes a variable 'sum' to store the cumulative sum.
    // Time Complexity: O(1) - Initializing a variable is a constant-time operation.

    for (int i = 1; i <= n; i++) {
        sum += i; // Add each number from 1 to n to the sum
        // Explanation: This line adds each number from 1 to 'n' to the 'sum'.
        // Time Complexity: O(n) - The loop runs 'n' times, and each iteration takes constant
time.
    }
    return sum; // Return the calculated sum
    // Explanation: This line returns the final calculated sum.
    // Time Complexity: O(1) - Returning a value from a function is a constant-time operation.
}

int main() {
    int n = 5; // You can change this value to calculate the sum for a different 'n'
    // Explanation: This line initializes the variable 'n' with a value (you can change it for
different 'n').
    // Time Complexity: O(1) - Assigning a value to a variable is a constant-time operation.

    // Calculate the sum of the first 'n' natural numbers using the function
    int result = sum_of_natural_numbers(n);
    // Explanation: This line calls the function to calculate the sum of the first 'n' natural
numbers.
    // Time Complexity: O(n) - The time complexity of the function is O(n) due to the loop.

    // Print the result
    printf("The sum of the first %d natural numbers is: %d\n", n, result);
    // Explanation: This line prints the result.
    // Time Complexity: O(1) - Printing is a constant-time operation.

    return 0;
    // Explanation: This line indicates a successful execution of the program.
    // Time Complexity: O(1) - Returning from the main function is a constant-time operation.
}
```

overall time complexity is O(n).

Post-Lab:

1) 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:
   IN C:

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

// Function to count the number of uppercase letters in a string
int count_uppercase_letters(char* str) {
    int count = 0;
    for (int i = 0; str[i] != '\0'; i++) {
        if (isupper(str[i])) {
            count++;
        }
    }
    return count;
    // Time Complexity: O(m), where m is the length of the string.
}

// Custom comparator function for sorting based on the count of uppercase letters
int compare_strings(const void* a, const void* b) {
    int count_a = count_uppercase_letters(*(char**)a);
    int count_b = count_uppercase_letters(*(char**)b);
    return count_a - count_b;
    // Time Complexity: O(m) for each call to count_uppercase_letters, where m is the
length of the string.
}

int main() {
    // Input array of strings
    char* arr[] = {"Hello", "WORLD", "abc", "DEFGH", "Testing"};
    int n = sizeof(arr) / sizeof(arr[0]);
    // Time Complexity: O(1), as this is a constant-time operation.

    // Sorting the array based on the count of uppercase letters
    qsort(arr, n, sizeof(arr[0]), compare_strings);
    // Time Complexity: O(n log n), where n is the number of elements in the array.

    // Output the sorted array
    printf("Output: {");
    for (int i = 0; i < n; i++) {
        printf("\"%s\"", arr[i]);
        if (i < n - 1) {
            printf(", ");
        }
    }
    printf("}\n");
    // Time Complexity: O(n), where n is the number of elements in the array.

    return 0;
    // Time Complexity: O(1), as this is a constant-time operation.
}
```

- The count_uppercase_letters function has a time complexity of O(m), where m is the length of the string.
- The compare_strings function calls count_uppercase_letters, so its time complexity is also O(m) for each call.
- The sorting operation using qsort has a time complexity of O(n log n), where n is the number of elements in the array.
- The loop for printing the output has a time complexity of O(n), where n is the number of elements in the array.

    - Overall, the dominant factor for time complexity is the sorting operation (O(n log n)).

2) In the streets of KLU, there's an array of electrical poles meticulously arranged in ascending order of their heights. Professor Stefan assigned Mothi the task of finding the position (index) of a particular pole, given its height (H). Mothi, known for his algorithmic prowess, crafted an algorithm to tackle this challenge. However, he struggles with evaluating the algorithm's time complexity. Your role is to assist Mothi by analyzing the time complexity of his problem-solving algorithm. Algorithm HEIGHT(a, low, high, tar) if low > high then return 0 end if mid <- floor((low + high) / 2) if a[mid] = tar then return mid else if tar < a[mid] then return HEIGHT(a, low, mid - 1, tar) else return HEIGHT(a, mid + 1, high, tar) end if End Algorithm
   - Procedure/Program:

   IN C:

```
#include <stdio.h>

// Function to find the position (index) of a pole given its height using binary search
int HEIGHT(int a[], int low, int high, int tar) {
  // Time Complexity: O(log n) - Binary search
  if (low > high) {
    // Explanation: If low > high, the element is not found, so return 0.
    // Time Complexity: O(1) - Constant time operation.
    return 0;
  }

  int mid = (low + high) / 2;
  // Explanation: Calculates the midpoint of the array.
  // Time Complexity: O(1) - Constant time operation.

  if (a[mid] == tar) {
    // Explanation: If the element at the midpoint is equal to the target, return mid.
    // Time Complexity: O(1) - Constant time operation.
    return mid;
  } else if (tar < a[mid]) {
    // Explanation: If the target is less than the element at mid, search in the left half.
    // Time Complexity: Recurrence relation T(n) = T(n/2), O(log n) - Binary search.
    return HEIGHT(a, low, mid - 1, tar);
  } else {
    // Explanation: If the target is greater than the element at mid, search in the right
half.
    // Time Complexity: Recurrence relation T(n) = T(n/2), O(log n) - Binary search.
    return HEIGHT(a, mid + 1, high, tar);
  }
}

int main() {
  int poles[] = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100}; // Example array of heights
  int n = sizeof(poles) / sizeof(poles[0]);
```

```
    int target = 60; // Example target height to find

    // Call the HEIGHT function to find the position of the target height
    int position = HEIGHT(poles, 0, n - 1, target);

    // Output the result
    if (position != 0) {
        // Explanation: Outputs the result based on whether the element is found or not.
        // Time Complexity: O(1) - Constant time operation.
        printf("The pole with height %d is at position %d.\n", target, position);
    } else {
        printf("The pole with height %d is not found.\n", target);
    }

    return 0;
    // Explanation: Indicates a successful execution of the program.
    // Time Complexity: O(1) - Constant time operation.
}
```