Date of the Session:___/___/_____   **Time of the Session:______to_______**

EX – 1 Analysis of Time and Space Complexity of Algorithms

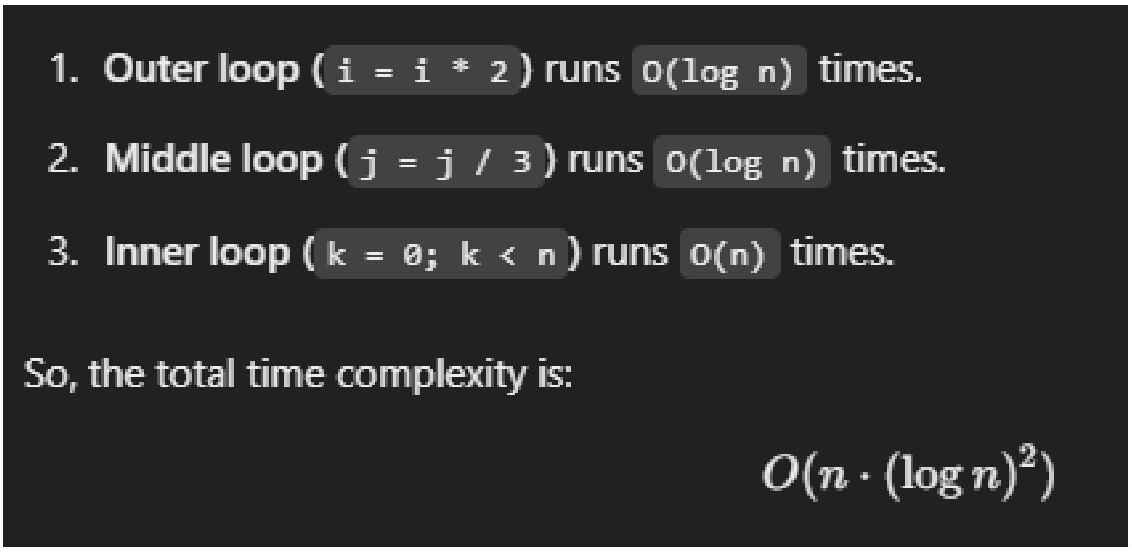
Prerequisites:

- Basics of Data Structures and C Programming.
- Basic knowledge about algorithms in C and Data Structures.

Pre-Lab:

1) During lockdown Mothi gets bored by his daily routine while scrolling youtube he found an algorithm that looks different Mothi is very crazy about algorithms, but he cannot solve algorithms of multiple loops so that he structed and need your help to find the time complexity of that algorithm

```
Algoritm KLU(int n)
{
        int count=0;
        for(int i=0;i<n;i=i*2)
        {
              for(int j=n;j>0;j=j/3)
              {
               for(int k=0;k<n;k++)
                    {
                    Count++;
                    }
              }
        }
}
```

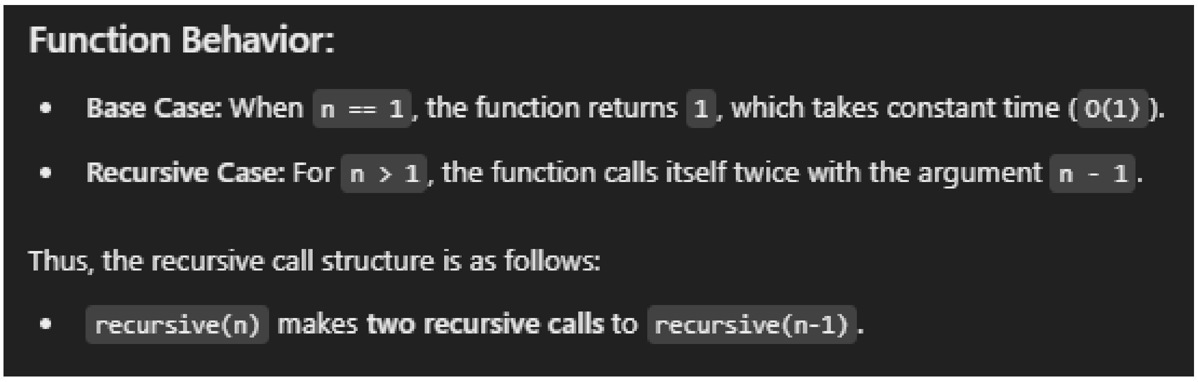

1. Outer loop (`i = i * 2`) runs `O(log n)` times.
2. Middle loop (`j = j / 3`) runs `O(log n)` times.
3. Inner loop (`k = 0; k < n`) runs `O(n)` times.

So, the total time complexity is:

$$O(n \cdot (\log n)^2)$$

2) Klaus Michaelson is interviewer, so he prepared a bunch of questions for the students. He focused on algorithms. Among all the questions the easiest question is what is the time complexity of the following C function is so can answer to this?

```
int recursive(int n)
{
        if(n==1)
    return (1);
    else
    return(recursive (n-1) + recursive (n-1));}

 }
```

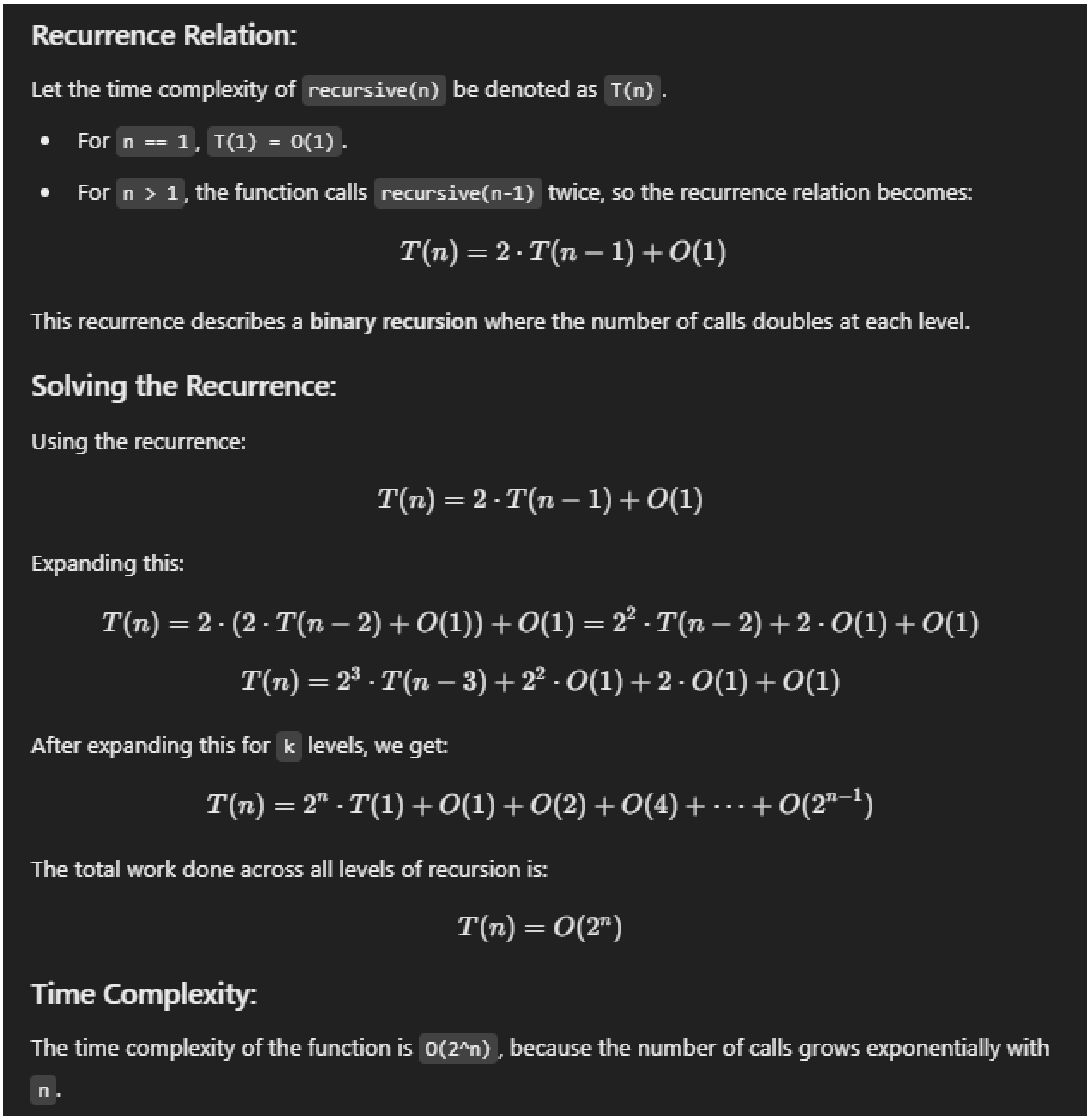


Function Behavior:

- **Base Case:** When `n == 1`, the function returns `1`, which takes constant time (`O(1)`).
- **Recursive Case:** For `n > 1`, the function calls itself twice with the argument `n - 1`.

Thus, the recursive call structure is as follows:

- `recursive(n)` makes **two recursive calls** to `recursive(n-1)`.



Recurrence Relation:

Let the time complexity of `recursive(n)` be denoted as `T(n)`.

- For `n == 1`, `T(1) = O(1)`.
- For `n > 1`, the function calls `recursive(n-1)` twice, so the recurrence relation becomes:

$$T(n) = 2 \cdot T(n-1) + O(1)$$

This recurrence describes a **binary recursion** where the number of calls doubles at each level.

Solving the Recurrence:

Using the recurrence:

$$T(n) = 2 \cdot T(n-1) + O(1)$$

Expanding this:

$$T(n) = 2 \cdot (2 \cdot T(n-2) + O(1)) + O(1) = 2^2 \cdot T(n-2) + 2 \cdot O(1) + O(1)$$

$$T(n) = 2^3 \cdot T(n-3) + 2^2 \cdot O(1) + 2 \cdot O(1) + O(1)$$

After expanding this for `k` levels, we get:

$$T(n) = 2^n \cdot T(1) + O(1) + O(2) + O(4) + \cdots + O(2^{n-1})$$

The total work done across all levels of recursion is:

$$T(n) = O(2^n)$$

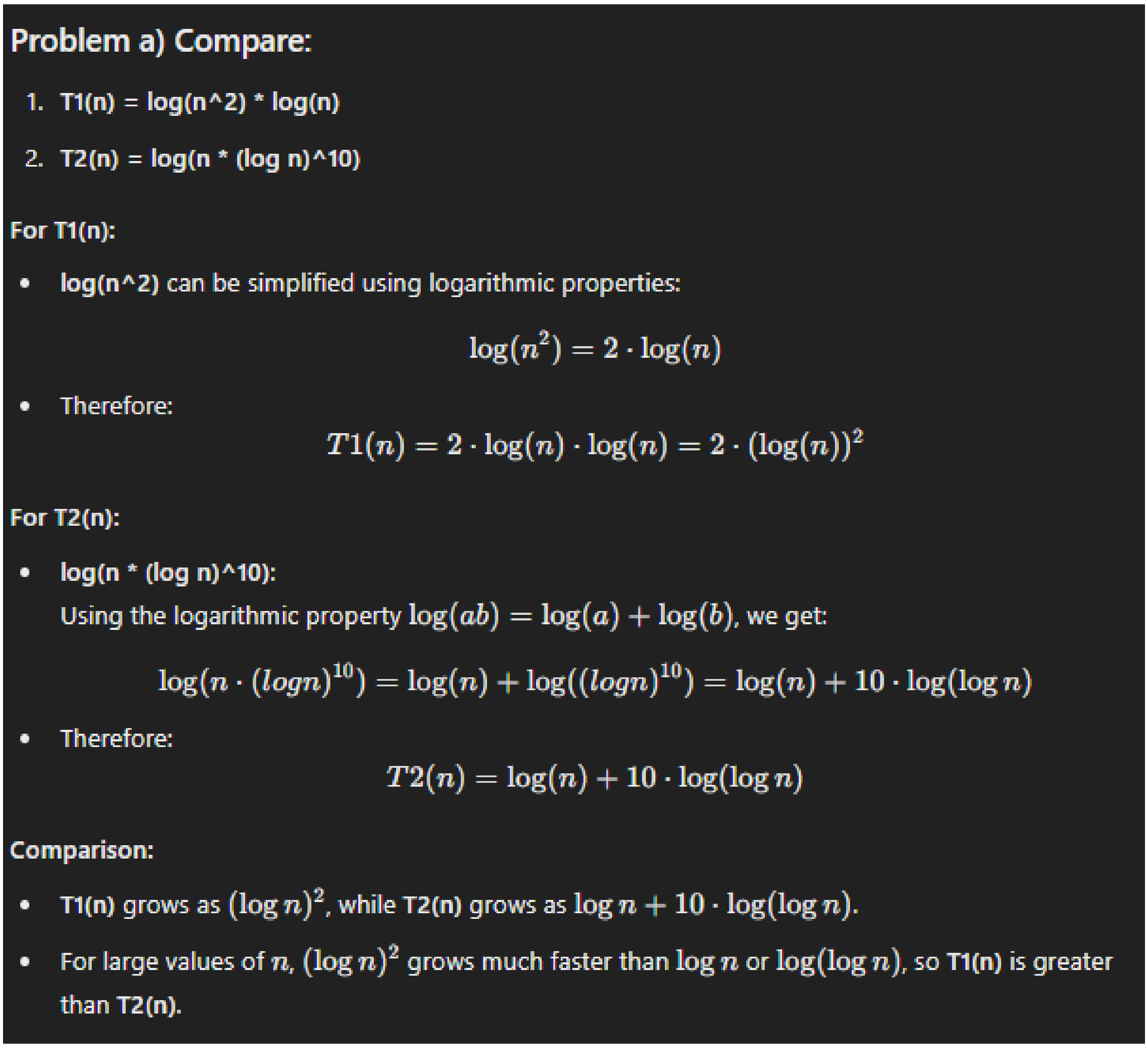
Time Complexity:

The time complexity of the function is `O(2^n)`, because the number of calls grows exponentially with `n`.

3) Mothi has 2 algorithms and he also know their time functions. Now he wants to analyze which one is greater function? But Mothi do not know how to compare two-time functions. Your task is to analyze the given two-time functions and judge which one is greater and justify with your solution

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)



Problem a) Compare:

1. **T1(n) = log(n^2) * log(n)**
2. **T2(n) = log(n * (log n)^10)**

For T1(n):

- **log(n^2)** can be simplified using logarithmic properties:

$$\log(n^2) = 2 \cdot \log(n)$$

- Therefore:

$$T1(n) = 2 \cdot \log(n) \cdot \log(n) = 2 \cdot (\log(n))^2$$

For T2(n):

- **log(n * (log n)^10):**
  Using the logarithmic property $\log(ab) = \log(a) + \log(b)$, we get:

$$\log(n \cdot (logn)^{10}) = \log(n) + \log((logn)^{10}) = \log(n) + 10 \cdot \log(\log n)$$

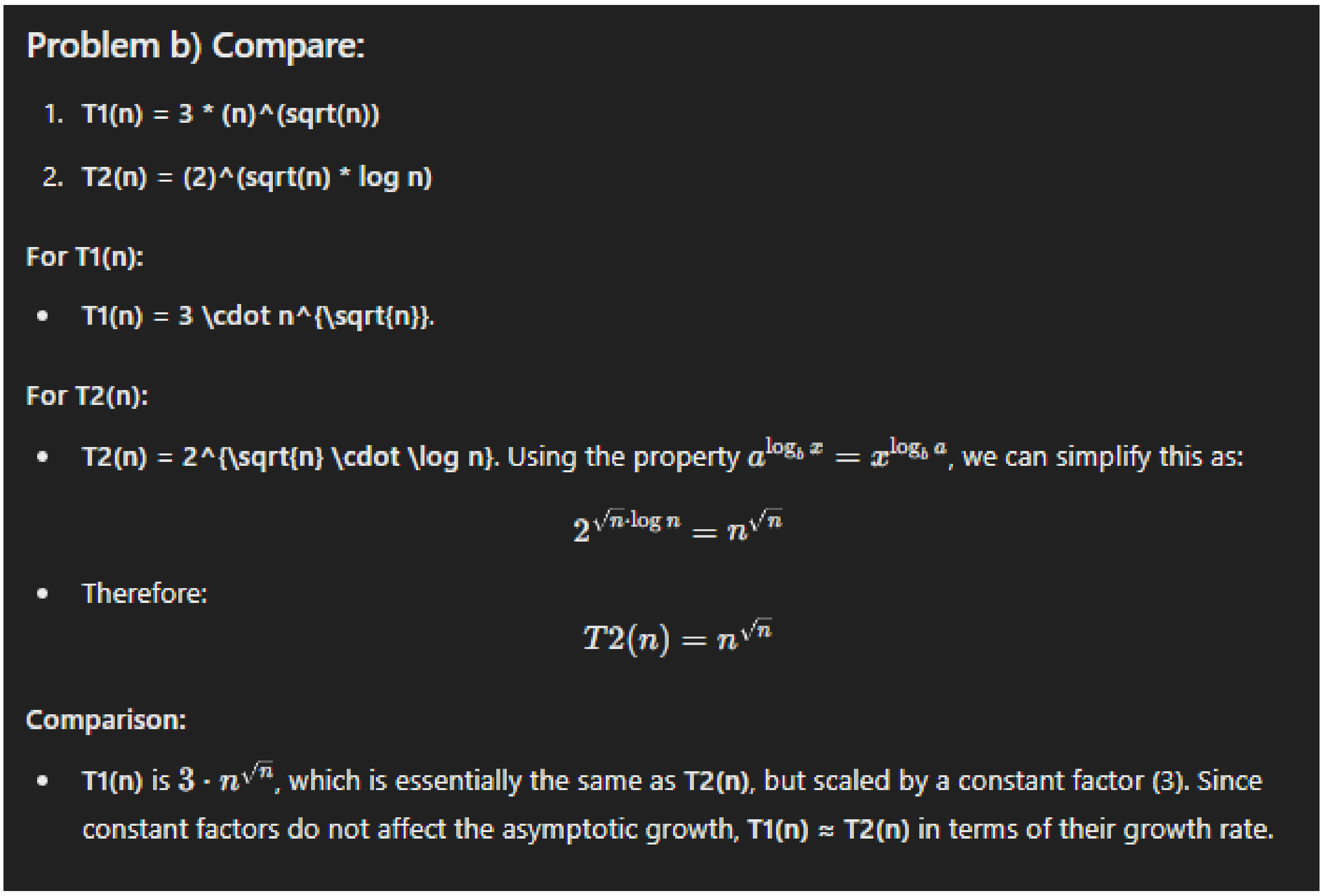
- Therefore:

$$T2(n) = \log(n) + 10 \cdot \log(\log n)$$

Comparison:

- **T1(n)** grows as $(\log n)^2$, while **T2(n)** grows as $\log n + 10 \cdot \log(\log n)$.
- For large values of n, $(\log n)^2$ grows much faster than $\log n$ or $\log(\log n)$, so **T1(n)** is greater than **T2(n)**.

[@KLWKS_BOT] THANOS



Problem b) Compare:

1. T1(n) = 3 * (n)^(sqrt(n))
2. T2(n) = (2)^(sqrt(n) * log n)

For T1(n):

- T1(n) = 3 \cdot n^{\sqrt{n}}.

For T2(n):

- T2(n) = 2^{\sqrt{n} \cdot \log n}. Using the property $a^{\log_b x} = x^{\log_b a}$, we can simplify this as:

$$2^{\sqrt{n} \cdot \log n} = n^{\sqrt{n}}$$

- Therefore:

$$T2(n) = n^{\sqrt{n}}$$

Comparison:

- T1(n) is $3 \cdot n^{\sqrt{n}}$, which is essentially the same as **T2(n)**, but scaled by a constant factor (3). Since constant factors do not affect the asymptotic growth, **T1(n) ≈ T2(n)** in terms of their growth rate.

In -Lab:

1) Caroline Forbes is an intelligent girl, every time she wins in any contest or programme, and solves complex problems so I want to give her a challenge problem that is. Sort an array of strings according to string lengths. If you are smarter than her, try to solve the problem faster than her?

Input

```
You       are    beautiful     looking
```

Output

```
You       are    looking       beautiful
```

Source code:

```
#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", "beautiful", "looking"};
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;
}
```

2) Stefan is the Assistant Professor of Computer science department so he just ask to interact with the students and helps the students to solve the complex problems in an easier way so, the problem given by the sir is sort an array according to count of set bits?
Example:
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}

Source code:

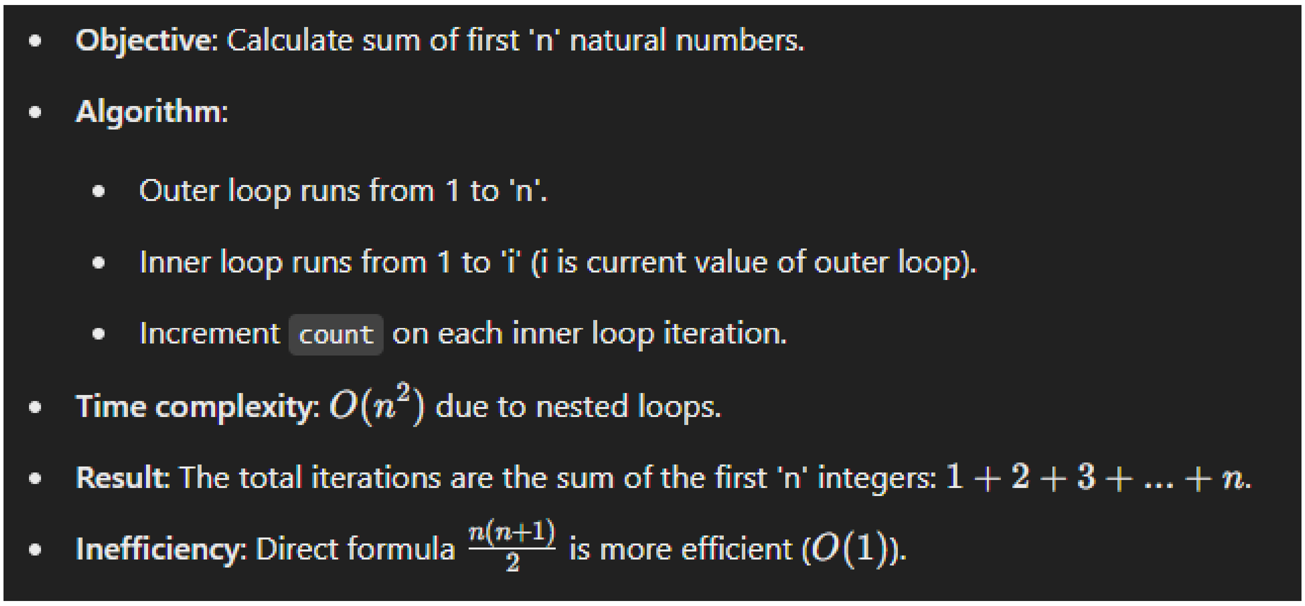
```
#include <stdio.h>
#include <stdlib.h>

int countSetBits(int n) {
    return __builtin_popcount(n);
}

int compare(const void *a, const void *b) {
    int bitsA = countSetBits(*(int *)a);
    int bitsB = countSetBits(*(int *)b);
    return bitsB - bitsA ? bitsB - bitsA : *(int *)a - *(int *)b;
}

int main() {
    int arr[] = {1, 2, 3, 4, 5, 6};
    int n = sizeof(arr) / sizeof(arr[0]);

    qsort(arr, n, sizeof(int), compare);

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

3) During the final skill exam teacher was given a problem and asked everyone to write the algorithm to it, and advised that try to implement a different approach from others.
Question: Write an algorithm to calculate sum of first 'n' natural numbers
Mothi, one of the student in the class thinking that his friends will write an efficient algorithm to that question. So, he wants to write a worst approach to make that algorithm as unique.

Algorithm:

```
Algorithm SumOfNNaturalNums(int n)
{
    int count=0;
    for( i=1;i<=n;i++)
    {
            for(j=1;j<=i;j++)
            {
            count++;
            }
    }
    return count;
}
```

Source code:



- **Objective**: Calculate sum of first 'n' natural numbers.
- **Algorithm**:
  - Outer loop runs from 1 to 'n'.
  - Inner loop runs from 1 to 'i' (i is current value of outer loop).
  - Increment `count` on each inner loop iteration.
- **Time complexity**: $O(n^2)$ due to nested loops.
- **Result**: The total iterations are the sum of the first 'n' integers: $1 + 2 + 3 + ... + n$.
- **Inefficiency**: Direct formula $\frac{n(n+1)}{2}$ is more efficient ($O(1)$).

Post- Lab :

1) Given an array arr[] of N strings, the task is to sort these strings according to the number of upper-case letters in them try to use zip function to get the format.

Input arr[] = {poiNtEr, aRRAy, cOde, foR}
Output: [('cOde', 1), ('foR', 1), ('poiNtEr', 2), ('aRRAy', 3)]

"aRRAy" R, R, A->3 Upper Case Letters
"poiNtEr" N, E->2 Upper Case Letters
"cOde" O->2 Upper Case Letters
"foR" R->3 Upper Case Letters

Source code:

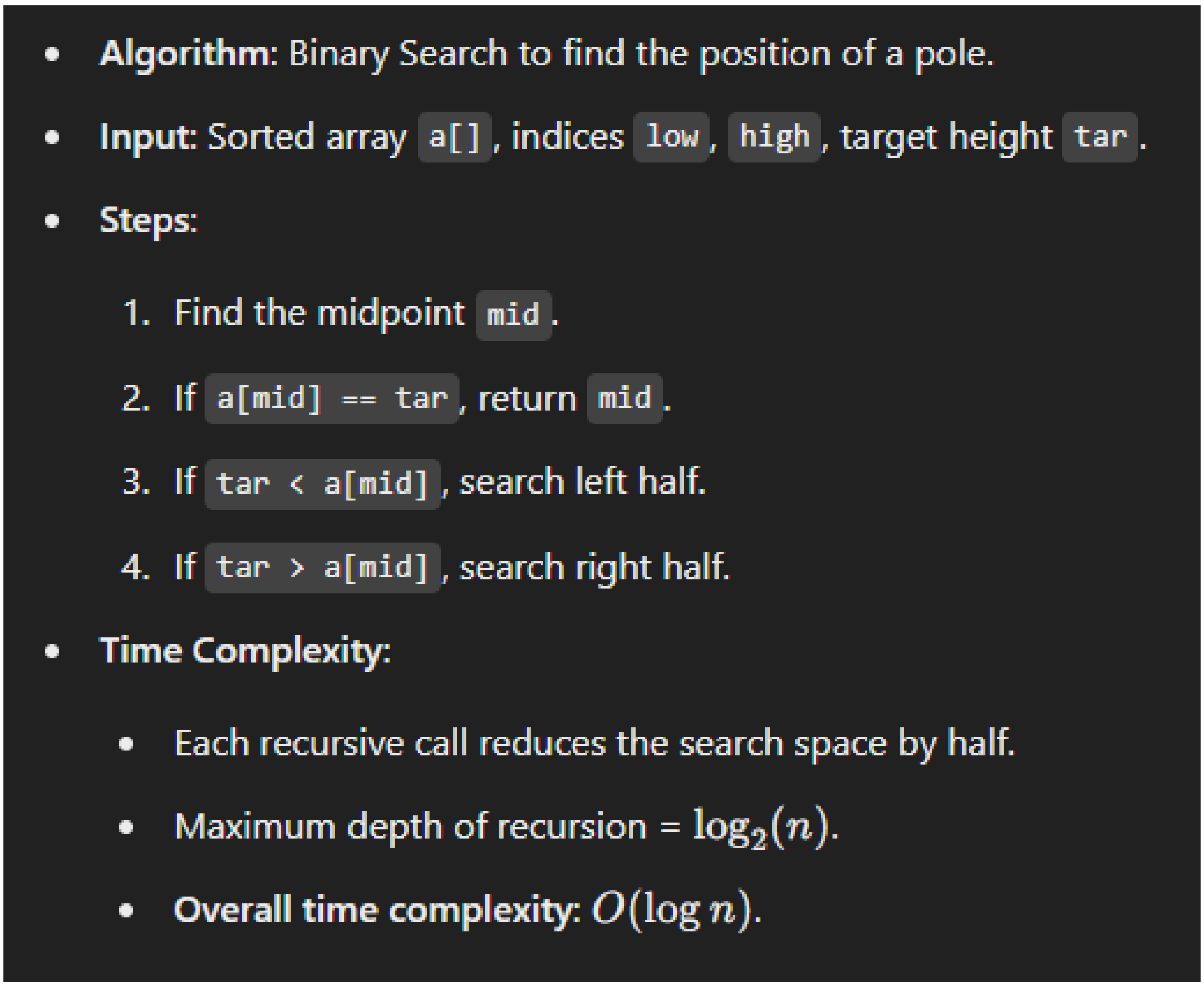
```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <stdlib.h>

int countUppercase(char *str) {
   int count = 0;
   while (*str) {
      if (isupper(*str)) count++;
      str++;
   }
   return count;
}

int compare(const void *a, const void *b) {
   return countUppercase(((char **)a)[0]) - countUppercase(((char **)b)[0]);
}

int main() {
   char *arr[] = {"poiNtEr", "aRRAy", "cOde", "foR"};
   int n = sizeof(arr) / sizeof(arr[0]);

   qsort(arr, n, sizeof(char *), compare);
   for (int i = 0; i < n; i++) {
      printf("('%s', %d)\n", arr[i], countUppercase(arr[i]));
   }

   return 0;
}
```

2) In KLU streets we have lots of electrical poles.
Note: all poles are sorted with respective to their heights.

Professor Stefan given the H = height of one pole to Mothi then asked him to print the position of that pole, here we consider index as a position. Mothi is particularly good at algorithms, so he written an algorithm to find that position. But he is extremely poor at finding time complexity. Your task is to help your friend Mothi to analyze the time complexity of the given problem.

```
Int BinarySearch (int a, int low, int high, int tar)
 {
int mid;
if (low > high) return 0;
mid = floor((low + high)/2)
if (a[mid] == tar)
        return mid;
else if (tar < a[mid])
        return BinarySearch (a, low, mid-1, tar)
else
        return BinarySearch (a, mid+1, high, tar)
 }
```

Source code:



- **Algorithm:** Binary Search to find the position of a pole.
- **Input:** Sorted array `a[]`, indices `low`, `high`, target height `tar`.
- **Steps:**
  1. Find the midpoint `mid`.
  2. If `a[mid] == tar`, return `mid`.
  3. If `tar < a[mid]`, search left half.
  4. If `tar > a[mid]`, search right half.
- **Time Complexity:**
  - Each recursive call reduces the search space by half.
  - Maximum depth of recursion = $\log_2(n)$.
  - **Overall time complexity:** $O(\log n)$.

Comments of the Evaluators (if Any)	Evaluator's Observation Marks Secured:________out of [50]. Signature of the Evaluator Date of Evaluation: