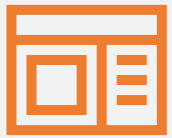
Lecture 1

Design and Analysis of Algorithms (CSET244)

Course Structure



3 Hours Lectures

1 Hour Tutorial

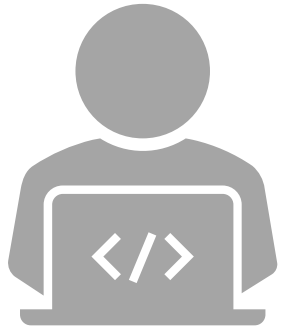
4 Hours Practical

Evaluation Policy

Components of Course Evaluation	Marks
Mid Semester Examination	20
End Semester Examination	40
Lab (Continuous Evaluation) *	20
Hackathon	05
Competitive programming (Leetcode)**	10
Certification***	5
Total	100

Foundation of the course (Prerequisite)

Data Structures



Programming

References

Books:

S.no	Book Title	Authors	Press
1.	Introduction to Algorithms	Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein	MIT Press
2.	Algorithm Design	Jon Kleinberg & Eva Tardos	Pearson
3.	Computer Algorithms	Ellis Horowitz, Sartaj Sahni, Sanguthevar Rajasekaran	Computer Science Press

Lab Continuous Evaluation

References

Other Resources:

Google

Geekforgeeks for practising questions and assignments.

Class Lectures

Lab Assignments

Programming language to follow: C++

Broad list of topics to be covered in the course

Basic of algorithms.

Sorting algorithms.

Divide and Conquer paradigm.

Greedy algorithms.

Dynamic programming.

Graph algorithms.

Computational classes.

Introduction to algorithm

Definition :

- It is a combination of sequence of finite steps to solve a problem.

Properties :

- It should take zero or more input.
- It should produce at least one output.
- It should terminate after finite time.
- Every state in the algorithm should be deterministic.
- It should be programming language independent.

Pseudocode

Using Iteration

```
Require: list a, size of list n, desired item x
  for i=0 to n-1 do
    if a[i]=x then
      return i
    end if
    i=i+1
  end for
  if i=n then
    return false
  end if
```

Steps Required to Construct Algorithm

- Problem Definition
- Design the Algorithm
- Flow Chart
- Verification (Testing)
- Implementation (Coding)
- Analysis

Analysis

Using time complexity and space Complexity we can analyze the algorithm.

Time Complexity:

$$T(A) = C(A) + R(A)$$

C(A)- Compile time (Depend on Compiler)
R(A) - Run Time (Depend on Processor)

Types of Analysis

Posteriori Analysis (Relative Analysis)	Priori Analysis (Absolute Analysis)
1. It is programming language of compiler and type of processor dependent analysis	1. It is programming language of compiler and type of processor independent analysis
2. Answer will change system to system	2. Answer will not change system to system
3. Exact answer	3. Approximate Analysis

Priori Analysis (Absolute Analysis)

"It is determination of order of magnitude of a statement."

Example 1:

```
main()
{
   x=y+z;
}
```

O(1)

If any program requires a fixed amount of time for all input values then its time complexity is said to be Constant Time Complexity.

Example 2:

```
main()
{
    x=y+z;      // Step 1: Constant time operation, O(1)
    for (i=1; i<=n; i++)   // Loop runs n times
      {
          x=y+z;   // Step 2: Constant time operation inside loop,
                   O(1), But executed n times. i.e. O(n).
      }
}     //overall time complexity of the given program
      is O(n).
```

Example 3:

```
main()
{
    x=y+z;
    for (i=1; i<=n; i++)
      x=y+z;
    for (i=1; i<=n; i++)
     {
         for (j=1; j<=n; j++)
             {
                 x=y+z;
              }
      }
 }
```

```
main()
{
   x = y + z;  // Line 1: O(1)

   for (i = 1; i <= n; i++)  // Outer Loop 1
      x = y + z;  // Line 2: O(n)

   for (i = 1; i <= n; i++)  // Outer Loop 2
   {
      for (j = 1; j <= n; j++)  // Inner Loop
      {
         x = y + z;  // Line 3: O(n^2)
      }
   }
}
```

Overall Time Complexity

Summing up the individual complexities:

- **$O(1) + O(n) + O(n^2)$**.
- The dominant term is **$O(n^2)$**, so the overall time complexity is **$O(n^2)$**.

Example 4:

```
main()
{
   for (i=1; i<=n; i++)        //This loop runs n times
     {
        for (j=1; j<=n/2; j++)    //For each iteration of the outer loop, the inner loop runs n / 2
            {                      times.
                x=y+z;            // It takes O(1) time.
             }
      }

}
```

Total number of iterations is **$n \times (n / 2) = n^2 / 2$**.
Since constant factors are ignored in Big-O notation, this gives **$O(n^2)$**.

Example 5:

```
main()
{
   for (i =1; i<=n; i++)
     {
        for (j=1; j<=i; j++)
            {
               for (k=1; k<=133; k++)
                {
                   x=y+z;
                }
             }

        }
 }
```

//The outer loop runs n times, with each iteration containing the middle loop.

// The middle loop runs from 1 to i, so for each iteration of i, it runs i times.

//This loop runs exactly 133 times, regardless of i or j. i.e. O(1)

1. Total Number of Iterations:

- For i = 1, the middle loop runs once, with the innermost loop running 133 times.
- For i = 2, the middle loop runs twice, and each iteration contains the innermost loop running 133 times.
- Therefore, the total iterations follow this pattern:

$$\sum_{i=1}^{n} \sum_{j=1}^{i} O(1) = \sum_{i=1}^{n} i \times 133$$

- Simplifying, we have:

$$133 \times \left(\sum_{i=1}^{n} i \right) = 133 \times \frac{n(n+1)}{2}$$

- Ignoring constants (133 and the factor of 1/2), the time complexity simplifies to **O(n²).**

Example 6:

```
main()
{
 while(n>=1)        // The loop continues as long as n >= 1
   {
       n=n/2;       // In each iteration, n is divided by 2. This reduces the size of n exponentially.

}
```

Number of Iterations:

Let's calculate how many times the loop runs before n becomes less than 1.
The sequence of values for n is :

n, n/2, n/4, ….

This is a geometric progression. After **k** iterations, $n=n/2^k$

The loop stops when $n/2^k<1$

Solving for **k,** $2^k = n \Rightarrow k=\log_2(n)$, Hence, total time complexity **O(log n)**.

Example 7:

```
main()
{
 i=1;
while(i<=n)
  {
      i=2*i;
   }
}
```

Number of Iterations:

- Let's determine how many iterations are needed before i > n.
- In the k-th iteration, i = 2^k.
- The loop stops when 2^k > n.
- Solving for k,
- $k=\log_2(n)$

Time complexity is **O(log n)**.

The Complexity of a recursive function

```
int factorial(int n)
{
   // Base case
   if (n == 0) {
      return 1;
   }

   // Recursive case
   return n * factorial(n - 1);
}
```

Time Complexity:

- The function calls itself recursively until it reaches the base case.
- For factorial(n), the number of recursive calls is `n` (e.g., factorial(n), factorial(n-1), ..., factorial(1), and finally factorial(0)).
- Therefore, the time complexity is **O(n).**

Space Complexity:

- Each function call consumes stack space, leading to a maximum depth of recursion equal to n.
- Thus, the space complexity due to the call stack is also **O(n)**.

```
unsigned factTR(unsigned int n, unsigned int a)
{
        if (n <= 1)
                return a;

        return factTR(n - 1, n * a);
}

unsigned int fact(unsigned int n) { return factTR(n, 1); }

int main()
{
        cout << fact(5);
        return 0;
}
```

Time Complexity: O(n)
Auxiliary Space: O(1)

Thank You