CST 306	ALGORITHM ANALYSIS AND DESIGN	Category	L	T	P	Credit	Year of Introduction
		PCC	3	1	0	4	2019

No.	Statement	Level
CO1	Analyze any given algorithm and express its time and space complexities in asymptotic notations.	Apply (Level 3)
CO2	Compute time complexity of algorithms from recurrence equations using Iteration, Recurrence Tree, Substitution and Master’s Method.	Apply (Level 3)
CO3	Illustrate Graph traversal algorithms, it’s applications and Advanced Data structures like AVL trees and Disjoint set operations.	Apply (Level 3)
CO4	Demonstrate Divide-and-conquer, Greedy Strategy, Dynamic programming, Branch-and Bound and Backtracking algorithm design techniques	Apply (Level 3)
CO5	Classify a problem as computationally tractable or intractable, and discuss strategies to address intractability	Understand (Level 2)
CO6	Identify the suitable design strategy to solve a given problem.	Apply (Level 3)

Text Books

1. T.H.Cormen, C.E.Leiserson, R.L.Rivest, C. Stein, Introduction to Algorithms, 2nd Edition, Prentice-Hall India (2001)
2. Ellis Horowitz, Sartaj Sahni, Sanguthevar Rajasekaran, “Fundamentals of Computer Algorithms”, 2nd Edition, Orient Longman Universities Press (2008)
3. Sara Baase and Allen Van Gelder -Computer Algorithms, Introduction to Design and Analysis, 3rd Edition, Pearson Education (2009)

Reference Books

1. Jon Kleinberg, Eva Tardos, “Algorithm Design”, First Edition, Pearson (2005)
2. Robert Sedgewick, Kevin Wayne, “Algorithms”,4th Edition Pearson (2011)
3. GIlles Brassard, Paul Brately, “Fundamentals of Algorithmics”, Pearson (1996)
4. Steven S. Skiena, “The Algorithm Design Manual”, 2nd Edition, Springer(2008)

Module-1 (Introduction to Algorithm Analysis)

Characteristics of Algorithms, Criteria for Analysing Algorithms, Time and Space Complexity - Best, Worst and Average Case Complexities

Asymptotic Notations - Big-Oh (O), Big-Omega (Ω), Big-Theta (Θ), Little-oh (o) and Little-Omega (ω) and their properties. Classifying functions by their asymptotic growth rate, Time and Space Complexity Calculation of simple algorithms.

Analysis of Recursive Algorithms: Recurrence Equations, Solving Recurrence Equations - Iteration Method, Recursion Tree Method, Substitution Method and Master's Theorem (Proof not required).

Algorithm

- An algorithm is a finite set of instructions that accomplishes a particular task.

Properties of an Algorithm

- **Input:** Zero or more inputs are externally supplied.
- **Output:** At least one output is produced.
- **Definiteness:** Each instruction is clear and unambiguous.
    - "add 6 or 7 to x" , "compute 5/0" etc. are not permitted.
- **Finiteness:** The algorithm terminates after a finite number of steps.
- **Effectiveness:** Every instruction must be very basic so that it can be carried out by a person using only pencil and paper in a finite amount of time. It also must be feasible.

Algorithm Analysis

- Algorithm analysis is the systematic process of evaluating an algorithm to determine the **amount of computational resources** it requires to solve a problem.
- The primary goal is to measure an algorithm’s **efficiency** so that different algorithms can be compared objectively, independent of hardware or programming language.

Key Aspects of Algorithm Analysis

- **Time Complexity:** Measures the execution time of an algorithm as a function of input size (n). It helps determine how fast the algorithm grows with increasing inputs. Typically expressed using asymptotic notations such as O, Ω, and Θ.
- **Space Complexity:** Measures the memory used by the algorithm during execution, including input storage, auxiliary variables, recursion stacks, etc.
- **Best, Worst, and Average Case Analysis**
  - Best Case: Minimum resource usage.
  - Worst Case: Maximum resource usage (commonly used for guarantees).
  - Average Case: Expected resource usage for random inputs.

- **Asymptotic Analysis:** Focuses on the growth rate of time and space usage as n becomes very large. This helps compare the scalability of different algorithms.
- **Analysis of Recursive Algorithms:** Involves forming and solving recurrence relations, using methods such as:
  - Iteration
  - Recursion Tree
  - Substitution
  - Master Theorem

Space & Time Complexity
Best, Worst and Average Case Complexities

Space Complexity

- The space complexity of an algorithm is the amount of memory it needs to run to completion

- **Space Complexity = Fixed Part + Variable Part**

$S(P) \quad = c + S_P$, Where P is any algorithm

- **A fixed part:**
  - It is independent of the characteristics of the inputs and outputs.
  - Eg:
    - Instruction space(i.e., space for the code)
    - space for simple variables and fixed-size component variables
    - space for constants

- **A variable part:**
  - It is dependent on the characteristics of the inputs and outputs.
  - Eg:
    - Space needed by component variables whose size is dependent on the particular problem instance being solved
    - Space needed by referenced variables
    - Recursion stack space.

Space Complexity Calculation

```
Algorithm Sum(A,n)
{
        s=0
        for i=0 to n-1 do
                s = s + A[i]
        return s
}
```

```
Algorithm Sum(A,n)
{
        s=0
        for i=0 to n-1 do
                s = s + A[i]
        return s
}
```

Space Complexity= Space for parameters and Space for local variables

A[] → n　　n→1　　s→1　　i→1

Space Complexity = n+3

Space Complexity

```
Algorithm mAdd(m,n,a,b,c)
{
        for i=1 to m do
                for j=1 to n do
                        c[i,j] := a[i,j] + b[i,j];
}
```

Space Complexity

```
Algorithm mAdd(m,n,a,b,c)
{
        for i=1 to m do
                for j=1 to n do
                        c[i,j] := a[i,j] + b[i,j];
}
```

Space Complexity= Space for parameters and Space for local variables

Space Complexity

```
Algorithm mAdd(m,n,a,b,c)
{
        for i=1 to m do
                for j=1 to n do
                        c[i,j] := a[i,j] + b[i,j];
}
```

Space Complexity= Space for parameters and Space for local variables

m→1   n→1   a[]→mn   b[]→mn   c[]→mn   i→1   j→1

Space Complexity

```
Algorithm mAdd(m,n,a,b,c)
{
        for i=1 to m do
                for j=1 to n do
                        c[i,j] := a[i,j] + b[i,j];
}
```

Space Complexity= Space for parameters and Space for local variables

m→1 n→1 a[]→mn b[]→mn c[]→mn i→1 j→1

Space complexity = 3mn + 4

Space Complexity

```
Algorithm RSum(a,n)
{
        if(n<=0)
                return 0;
        else
                return a[n] + RSum(a,n-1)
}
```

Space Complexity

```
Algorithm RSum(a,n)
{
        if(n<=0)
                return 0;
        else
                return a[n] + RSum(a,n-1)
}
```

Space Complexity
= Space for Stack
= Space for parameters + Space for local variables + Space for return address

Space Complexity

```
Algorithm RSum(a,n)
{
        if(n<=0)
                return 0;
        else
                return a[n] + RSum(a,n-1)
}
```

Space Complexity

= Space for Stack

= Space for parameters + Space for local variables + Space for return address

For each recursive call the amount of stack required is 3

Space for parameters: a→1 n→1

Space for local variables: No local variables

Space for return address: 1

Space Complexity

```
Algorithm RSum(a,n)
{
        if(n<=0)
                return 0;
        else
                return a[n] + RSum(a,n-1)
}
```

Space Complexity
 = Space for Stack
 = Space for parameters + Space for local variables + Space for return address

For each recursive call the amount of stack required is 3

 Space for parameters: a→1 n→1
 Space for local variables: No local variables
 Space for return address: 1

Total number of recursive call = n+1

Space Complexity

```
Algorithm RSum(a,n)
{
        if(n<=0)
                return 0;
        else
                return a[n] + RSum(a,n-1)
}
```

Space Complexity

= Space for Stack

= Space for parameters + Space for local variables + Space for return address

For each recursive call the amount of stack required is 3

Space for parameters: a→1 n→1

Space for local variables: No local variables

Space for return address: 1

Total number of recursive call = n+1

Space complexity = 3(n+1)

Time Complexity

- The time complexity of an algorithm is the amount of computer time it needs to run to completion. Compilation time is excluded.
- **Time Complexity = Frequency Count * Time for Executing one Statement**
- **Frequency Count** → Number of times a particular statement will execute
- Time complexity α Frequency Count

Time Complexity Calculation

Frequency Count

```
Algorithm Sum(A,n)
{
        s=0
        for i=0 to n-1 do
                s = s + A[i]
        return s
}
```

```
                                                  Frequency Count
Algorithm Sum(A,n)
{
        s=0                            ------->   1
        for i=0 to n-1 do              ------->   n+1
                s = s + A[i]           ------->   n
        return s                       ------->   1
}
                                                  ---------------
```

Total Frequency Count = $2n + 3$

Time Complexity α Frequency Count

Time Complexity = $2n + 3$

Time Complexity

Frequency Count

```
Algorithm mAdd(m,n,a,b,c)
{
        for i=1 to m do
                for j=1 to n do
                        c[i,j] := a[i,j] + b[i,j];
}
```

Time Complexity

```
Algorithm mAdd(m,n,a,b,c)
{
        for i=1 to m do                    → m+1
                for j=1 to n do
                        c[i,j] := a[i,j] + b[i,j];
}
```

Frequency Count

m+1

Time Complexity

```
                                                        Frequency Count
Algorithm mAdd(m,n,a,b,c)
{
        for i=1 to m do                      ------>   m+1
                for j=1 to n do              ------>   m(n+1)
                        c[i,j] := a[i,j] + b[i,j];
}
```

Time Complexity

Frequency Count

```
Algorithm mAdd(m,n,a,b,c)
{
        for i=1 to m do                          →  m+1
            for j=1 to n do                      →  m(n+1)
                c[i,j] := a[i,j] + b[i,j];       →  mn
}
```

Time Complexity

```
Algorithm mAdd(m,n,a,b,c)
{
        for i=1 to m do                          → m+1
            for j=1 to n do                      → m(n+1)
                c[i,j] := a[i,j] + b[i,j];       → mn
}
```

Frequency Count

Statement	Frequency Count
for i=1 to m do	**m+1**
for j=1 to n do	**m(n+1)**
c[i,j] := a[i,j] + b[i,j];	**mn**

Time Complexity = 2mn + 2m + 1

Time Complexity

```
Algorithm RSum(a,n)
{
        if(n<=0)
                return 0;
        else
                return a[n] + RSum(a,n-1)
}
```

Time Complexity

```
Algorithm RSum(a,n)
{
        if(n<=0)
                return 0;
        else
                return a[n] + RSum(a,n-1)
}
```

Frequency Count n<=0

Frequency Count n>0

Time Complexity

```
Algorithm RSum(a,n)
{
        if(n<=0)
                return 0;
        else
                return a[n] + RSum(a,n-1)
}
```

	Frequency Count n<=0	**Frequency Count n>0**
if(n<=0) $\rightarrow$	1	

Time Complexity

```
Algorithm RSum(a,n)
{
        if(n<=0)
                return 0;
        else
                return a[n] + RSum(a,n-1)
}
```

Line	Frequency Count n<=0	Frequency Count n>0
if(n<=0)	**1**	
return 0;	**1**	

Time Complexity

```
Algorithm RSum(a,n)
{
        if(n<=0)
                return 0;
        else
                return a[n] + RSum(a,n-1)
}
```

	Frequency Count n<=0	Frequency Count n>0
if(n<=0)	1	
return 0;	1	
else	0	
return a[n] + RSum(a,n-1)	0	

Time Complexity

```
Algorithm RSum(a,n)
{
        if(n<=0)
                return 0;
        else
                return a[n] + RSum(a,n-1)
}
```

	Frequency Count $n<=0$	Frequency Count $n>0$
if(n<=0)	1	
return 0;	1	
else	0	
return a[n] + RSum(a,n-1)	0	
Time Complexity =	2	

Time Complexity

```
Algorithm RSum(a,n)
{
        if(n<=0)                    ⟶  1
                return 0;
        else
                return a[n] + RSum(a,n-1)
}
```

Frequency Count n<=0

Frequency Count n>0

1

Time Complexity

```
Algorithm RSum(a,n)
{
        if(n<=0)
                return 0;
        else
                return a[n] + RSum(a,n-1)
}
```

	Frequency Count $n<=0$	Frequency Count $n>0$
if(n<=0)		**1**
return 0;		**0**

Time Complexity

```
Algorithm RSum(a,n)
{
        if(n<=0)
                return 0;
        else
                return a[n] + RSum(a,n-1)
}
```

	Frequency Count n<=0	Frequency Count n>0
if(n<=0)		1
return 0;		0
else		0
return a[n] + RSum(a,n-1)		

Time Complexity

```
Algorithm RSum(a,n)
{
        if(n<=0)
                return 0;
        else
                return a[n] + RSum(a,n-1)
}
```

	Frequency Count n<=0	Frequency Count n>0
if(n<=0)		1
return 0;		0
else		0
return a[n] + RSum(a,n-1)		1+T(n-1)

Time Complexity

```
Algorithm RSum(a,n)
{
        if(n<=0)                        → 1
                return 0;               → 0
        else                            → 0
                return a[n] + RSum(a,n-1)   → 1+T(n-1)
}
```

	Frequency Count n<=0	Frequency Count n>0
if(n<=0)		1
return 0;		0
else		0
return a[n] + RSum(a,n-1)		1+T(n-1)
Time Complexity =		**2 + T(n-1)**

Time Complexity

	Frequency Count n<=0	Frequency Count n>0
Algorithm RSum(a,n)		
{		
if(n<=0)	1	1
return 0;	1	0
else	0	0
return a[n] + RSum(a,n-1)	0	1+T(n-1)
}		
	2	2 + T(n-1)

Time Complexity

```
Algorithm RSum(a,n)
{
        if(n<=0)
                return 0;
        else
                return a[n] + RSum(a,n-1)
}
```

Statement	Frequency Count n<=0	Frequency Count n>0
if(n<=0)	1	1
return 0;	1	0
else	0	0
return a[n] + RSum(a,n-1)	0	1+T(n-1)
	2	2 + T(n-1)

$$\text{Time Complexity} = T(n) = \begin{cases} 2 & \text{if } n<=0 \\ 2+T(n-1) & \text{Otherwise} \end{cases}$$

Time Complexity

```
int fun1(int n)
{       if(n ≤ 1)
                return n;
        return 2xfun1(n-1);
}
```

Frequency Count n<=1	Frequency Count n>1
1	1
1	0
0	1+T(n-1)
2	**2 + T(n-1)**

$$\text{Time Complexity} = T(n) = \begin{cases} 2 & \text{if } n<=1 \\ 2+T(n-1) & \text{Otherwise} \end{cases}$$

$$\textbf{Time Complexity} = T(n) = \begin{cases} \mathbf{2} & \textbf{if } n<=1 \\ \mathbf{2+T(n-1)} & \textbf{Otherwise} \end{cases}$$

$$\begin{aligned}
T(n) &= 2+ T(n-1) \\
&= 2+ [2+T(n-2)] \quad =2x2+T(n-2) \\
&= 2x2+ [2+T(n-3)] \quad =3x2+T(n-3) \\
&\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\
&= kx2+T(n-k)
\end{aligned}$$

Assume that $n-k=1 \rightarrow k=n-1$

$$\begin{aligned}
T(n) &= 2(n-1) + T(1) \\
&= 2(n-1) + 2 = 2n \\
&= \mathbf{O(n)}
\end{aligned}$$

Time Complexity

```
int fun2(int n)
{       if(n ≤ 1)
                return n;
        return fun2(n-1)xfun2(n-1);
}
```

	Frequency Count n<=1	Frequency Count n>1
if(n ≤ 1)	1	1
return n;	1	0
return fun2(n-1)xfun2(n-1);	0	1+2T(n-1)
	2	**2 + 2T(n-1)**

$$\text{Time Complexity} = T(n) = \begin{cases} 2 & \text{if } n<=1 \\ 2+2T(n-1) & \text{Otherwise} \end{cases}$$

$$\textbf{Time Complexity} = T(n) = \begin{cases} 2 & \text{if } n<=1 \\ 2+2T(n-1) & \text{Otherwise} \end{cases}$$

$$T(n) = 2+2T(n-1)$$

$$= 2+2[2+2T(n-2)] \quad =2+2^2+2^2T(n-2)$$

$$= 2+2^2+2^2[2+2T(n-3)] \qquad = 2+2^2+2^3+2^3T(n-3)$$

$$\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots$$

$$= 2+2^2+2^3+\ldots. +2^k+2^kT(n-k)$$

Assume that n-k=1 ➔ k=n-1

$$T(n) = 2[1+2+2^2+2^3+\ldots. +2^{k-1}]+2^kT(n-k)$$

$$= 2[(2^k-1)/(2-1)] + 2^kT(1)$$

$$= 2^{k+1}-2+ 2^kx2 \qquad = 2^{k+1}-2+ 2^{k+1}$$

$$= 2^n-2+ 2^n \quad = 2x2^n-2 \qquad = \mathbf{O(2^n)}$$

Time Complexity

Frequency Count

```
function(int n)
{       if(n==1)      return;
        for(i=1; i<=n; i++)
                for(int j=1; j<=n; j++)
                {       printf("*");
                        break;
                }
}
```

Time Complexity

```
function(int n)
{       if(n==1)      return;
        for(i=1; i<=n; i++)
                for(int j=1; j<=n; j++)
                {       printf("*");
                        break;
                }
}
```

Frequency Count

1	**=O(1)**
n+1	**=O(n)**
n	**=O(n)**
n	**=O(n)**
n	**=O(n)**

O(n)

Time Complexity = T(n) = O(n)

Time Complexity

```
void function(int n)
{       int i=1, s=1;          ——> O(1)
        while(s<=n)
        {       i++;
                s+=i;           } Most executing statements
                printf("*");
        }
}
```

Time Complexity

```
void function(int n)
{       int i=1, s=1;
        while(s<=n)
        {       i++;
                s+=i;
                printf("*");
        }
}
```

$\rightarrow$ **O(1)**

Most executing statements

For each iteration i will be incremented by one and added to S

$1+2+3+\ldots\ldots+k \le n$,   where k is the maximum number of iterations of while loop

$[k(k+1)]/2 \le n \quad \Rightarrow \quad (k^2+k)/2 \le n \quad \Rightarrow \quad k \approx \sqrt{n}$

Time Complexity = $O(\sqrt{n})$

Time Complexity

```
main()
{    for(i=1; i<=n; i=i*2)
          sum = sum + i + func(i);
}
void func(int m)
{    for(j=1; j<=m; j++)
          Statement with O(1) complexity
}
```

Most executing statements

Time Complexity

```
main()
{    for(i=1; i<=n; i=i*2)
          sum = sum + i + func(i);
}
int func(int m)
{    for(j=1; j<=m; j++)
          Statement with O(1) complexity
}
```

$\left.\right\}$ **Most executing statements**

for loop in main() will successfully execute in $\log_2(n)$ times

for loop in func() will execute $2^0, 2^1, 2^2, \ldots\ldots, 2^{\log(n)-1}$ times

$$T(n) = 2^0 + 2^1 + 2^2 + \ldots\ldots + 2^{\log(n)-1}$$
$$= (2^{\log(n)} - 1)/(2-1) \qquad = 2^{\log(n)} - 1$$
$$= n-1 = \mathbf{O(n)}$$

Time Complexity

```
void function(int n)
{    int count=0;
     for(int i=n/2; i<=n; i++)
          for(int j=1; j<=n; j=2*j)
               for(int k=1; k<=n; k=k*2)
                    count++;
}
```

Time Complexity

```
void function(int n)
{    int count=0;
     for(int i=n/2; i<=n; i++)
          for(int j=1; j<=n; j=2*j)
               for(int k=1; k<=n; k=k*2)
                    count++;
}
```

Frequency Count

1	**=O(1)**
(n/2)+1	**=O(n)**
(n/2)log(n) + 1	**=O(nlog(n))**
(n/2)log(n)log(n) +1	**=O(nlog(n)log(n))**
(n/2)log(n)log(n)	**=O(nlog(n)log(n))**

O(nlog(n)log(n))

Time Complexity = O(nlog(n)log(n))

Time Complexity

Express the return value of the function mystery in ϴ notation

```
int mystery(int n)
{        int j=0,total=0;
         for (int i=1;j<=n;i++)
         {
                  ++total;
                  j+=2*i;
         }
         return total;
}
```

Time Complexity

Express the return value of the function mystery in ɵ notation

```
int mystery(int n)
{         int j=0,total=0;
          for (int i=1;j<=n;i++)
          {
                    ++total;
                    j+=2*i;
          }
          return total;
}
```

The value of total is depends on the number of iterations of for loop

$0+2x1+2x2+2x3+ \ldots\ldots\ldots +2x(k-1) \le n$, where k is the number of iterations of for loop

$2(1+2+3+\ldots..+(k-1)) \le n$

$2\ [(k-1)k]/2\ \le n$ ➔ $[k^2-k)]\ \le n$ ➔ $k \approx \sqrt{n}$

Time Complexity = ɵ($\sqrt{n}$)

Best, Worst and Average Case Complexities

Linear Search

Best, Worst and Average Case Complexities

```
Algorithm Search(a,n,x)
{
      for i:=1 to n do
          if a[i] ==x then
               return i;
      return -1;
}
```

Best Case	Worst Case	Average Case

Best, Worst and Average Case Complexities

```
Algorithm Search(a,n,x)
{
    for i:=1 to n do
        if a[i] ==x then
            return i;
    return -1;
}
```

Best-case scenario: The search data will be located at the first position in the array.
Number of comparisons: Only 1 comparison

Best, Worst and Average Case Complexities

```
Algorithm Search(a,n,x)
{
      for i:=1 to n do
         if a[i] ==x then
             return i;
      return -1;
}
```

Best Case	Worst Case	Average Case
1		
1		
1		
0		
3		

Time Complexity = 3

Best-case scenario: The search data will be located at the first position in the array.

Number of comparisons: Only 1 comparison

Best, Worst and Average Case Complexities

```
Algorithm Search(a,n,x)
{
      for i:=1 to n do
          if a[i] ==x then
                return i;
      return -1;
}
```

Best Case **Worst Case** **Average Case**

Worst Case Situation: Search data does not exist in the array or
The key is present at the last position.
Number of comparisons: All n elements are compared.

Best, Worst and Average Case Complexities

```
Algorithm Search(a,n,x)
{
      for i:=1 to n do
         if a[i] ==x then
               return i;
      return -1;
}
```

	Best Case	Worst Case	Average Case
for i:=1 to n do		n+1	
if a[i] ==x then		n	
return i;		0	
return -1;		1	

Time Complexity = $2n + 2$

Worst Case Situation: Search data does not exist in the array or
The key is present at the last position.

Number of comparisons: All n elements are compared.

Best, Worst and Average Case Complexities

```
Algorithm Search(a,n,x)
{
      for i:=1 to n do
          if a[i] ==x then
                return i;
      return -1;
}
```

Best Case **Worst Case** **Average Case**

Average Case Situation: Search data is in the middle of the array

Best, Worst and Average Case Complexities

```
Algorithm Search(a,n,x)
{
    for i:=1 to n do
        if a[i] ==x then
            return i;
    return -1;
}
```

	Best Case	Worst Case	Average Case
for i:=1 to n do			n/2
if a[i] ==x then			n/2
return i;			1
return -1;			0
Time Complexity =			n+1

Average Case Situation: Search data is in the middle of the array.
(The key is assumed to be equally likely to be at any position in the array)

Number of comparisons: $\frac{1+2+3+\cdots+n}{n} = \frac{n+1}{2}$

Best, Worst and Average Case Complexities

```
Algorithm Search(a,n,x)
{
     for i:=1 to n do
        if a[i] ==x then
             return i;
     return -1;
}
```

	Best Case	Worst Case	Average Case
for i:=1 to n do	1	n+1	n/2
if a[i] ==x then	1	n	n/2
return i;	1	0	1
return -1;	0	1	0
Time Complexity =	3	2n + 2	n+1