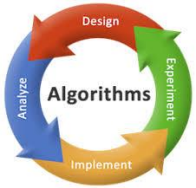
DESIGN AND ANALYSIS OF ALGORITHMS (DAA)

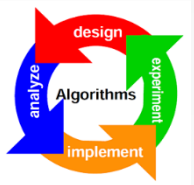
By :-

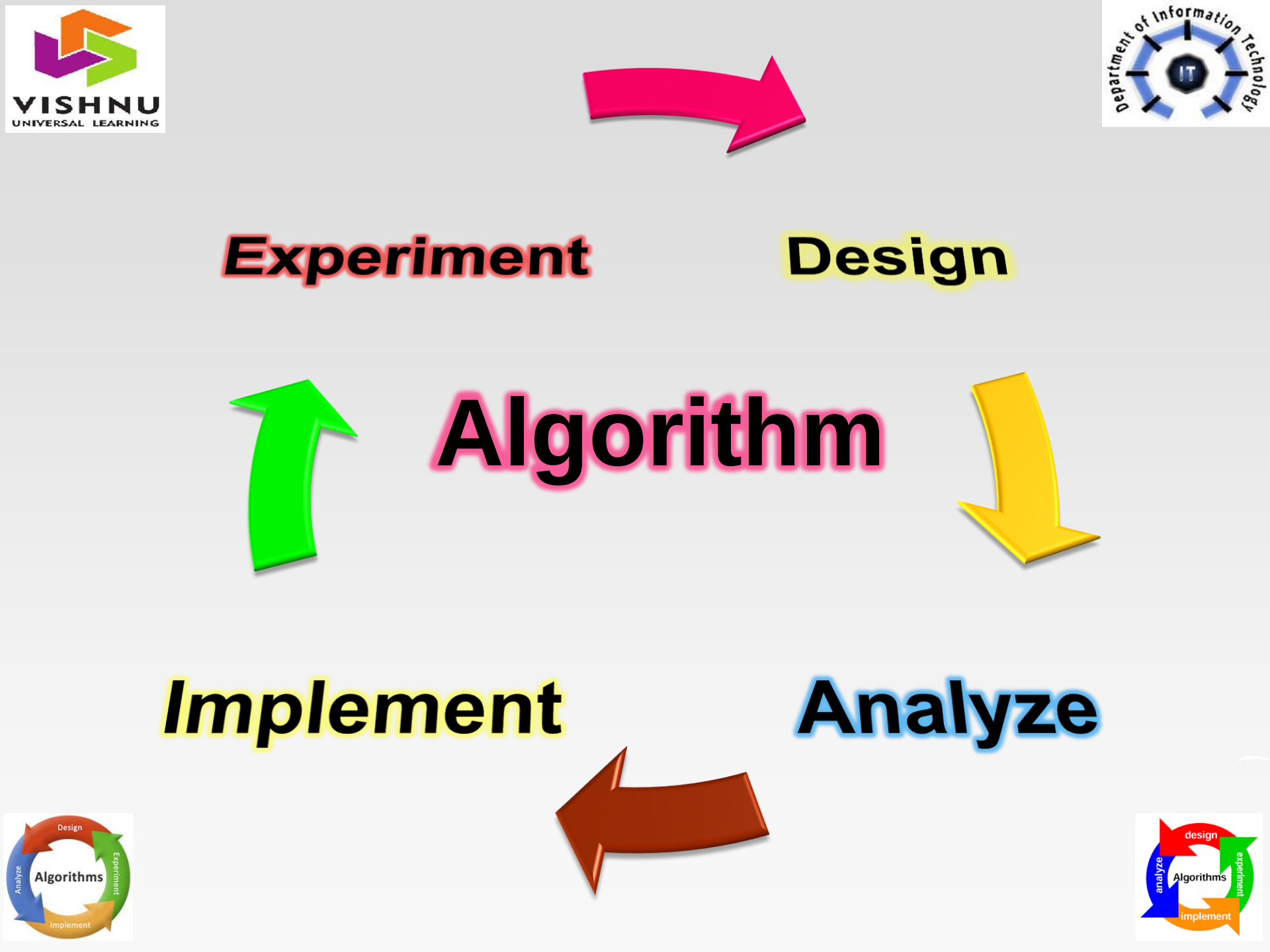
VIJAYKUMAR MANTRI,

ASSOCIATE PROFESSOR.

vijay_mantri.it@bvrit.ac.in






Experiment
Design
Algorithm
Implement
Analyze

Course Objective

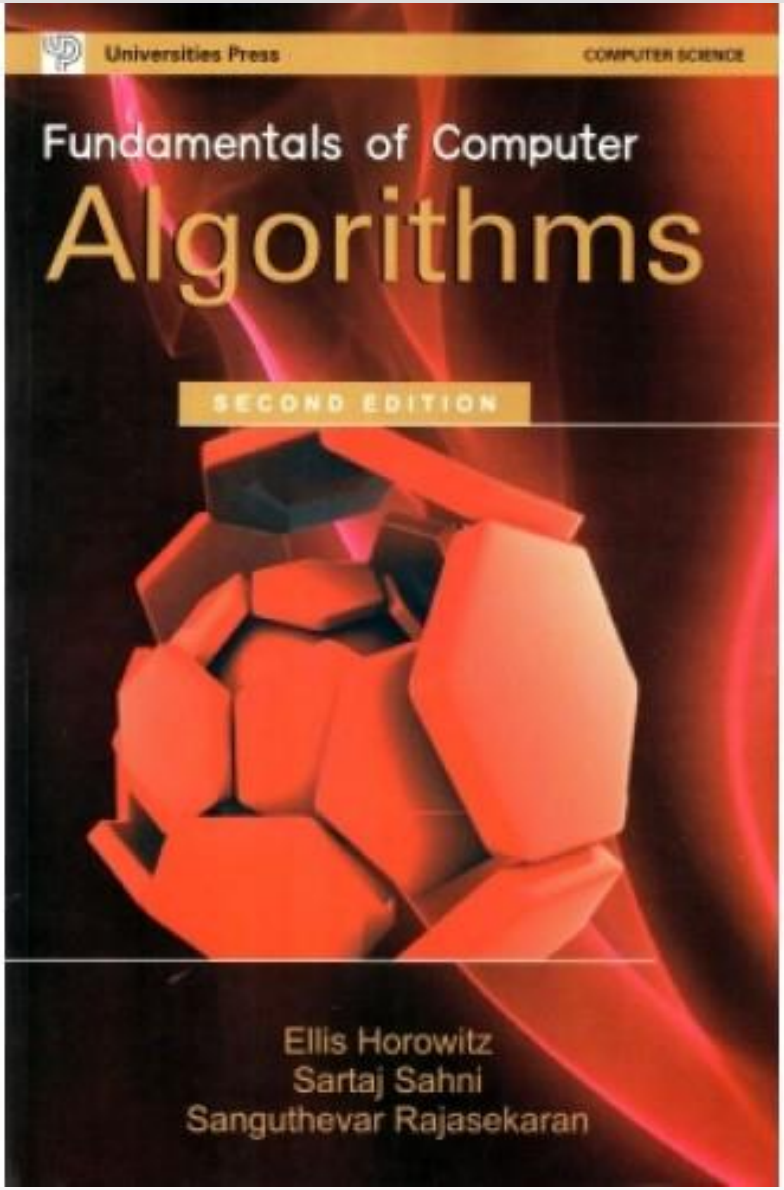
- Relate the algorithm properties with mathematical approaches to design and analyze real time problems.
- Make use of optimization techniques to solve complex problems in easy ways.
- Ability to perform dynamic actions for the particular problem based on specific constraints.
- Construction of state space tree in order to reduce the number of solutions and to find the optimal solution.
- Design elementary deterministic and randomized algorithms to solve computational problems.

Course Outcomes

Outcomes: Upon the successful completion of the course, the student will be able:

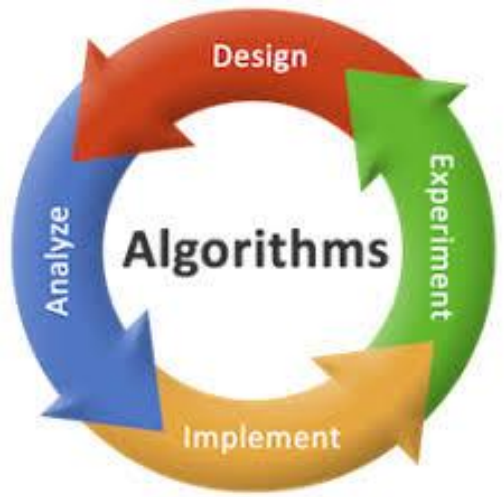
- Develop algorithms, as well as to analyze and measure the complexity in terms of space and time.
- Apply the knowledge of optimization techniques to develop and analyze algorithms for both theoretical and practical scenarios.
- Apply knowledge of mathematics, science, and engineering to real world problems.
- Build optimal solutions in personal and real time applications.
- Develop deterministic and randomized algorithms to solve computational problems.

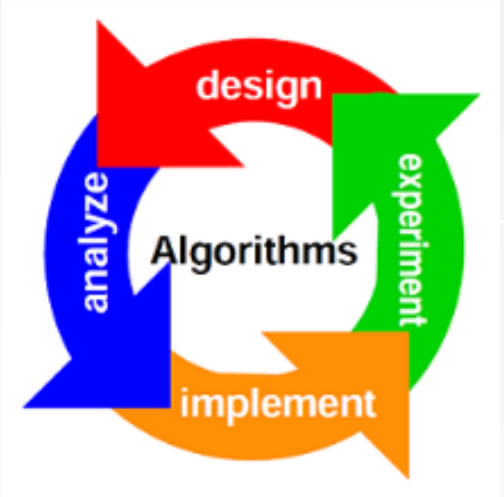
Textbook


Universities Press
COMPUTER SCIENCE
Fundamentals of Computer
Algorithms
SECOND EDITION
Ellis Horowitz
Sartaj Sahni
Sanguthevar Rajasekaran

DAA Unit I

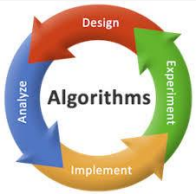
Introduction

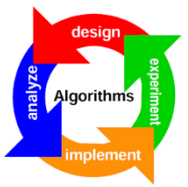




Unit I Syllabus Introduction

- Algorithm
- Pseudo Code for Expressing Algorithms
- Performance Analysis
- Space Complexity
- Time Complexity
- Asymptotic Notation –
  - Big Oh (O) Notation
  - Omega (Ω) Notation
  - Theta (Θ) Notation and
  - Little Oh (o) Notation



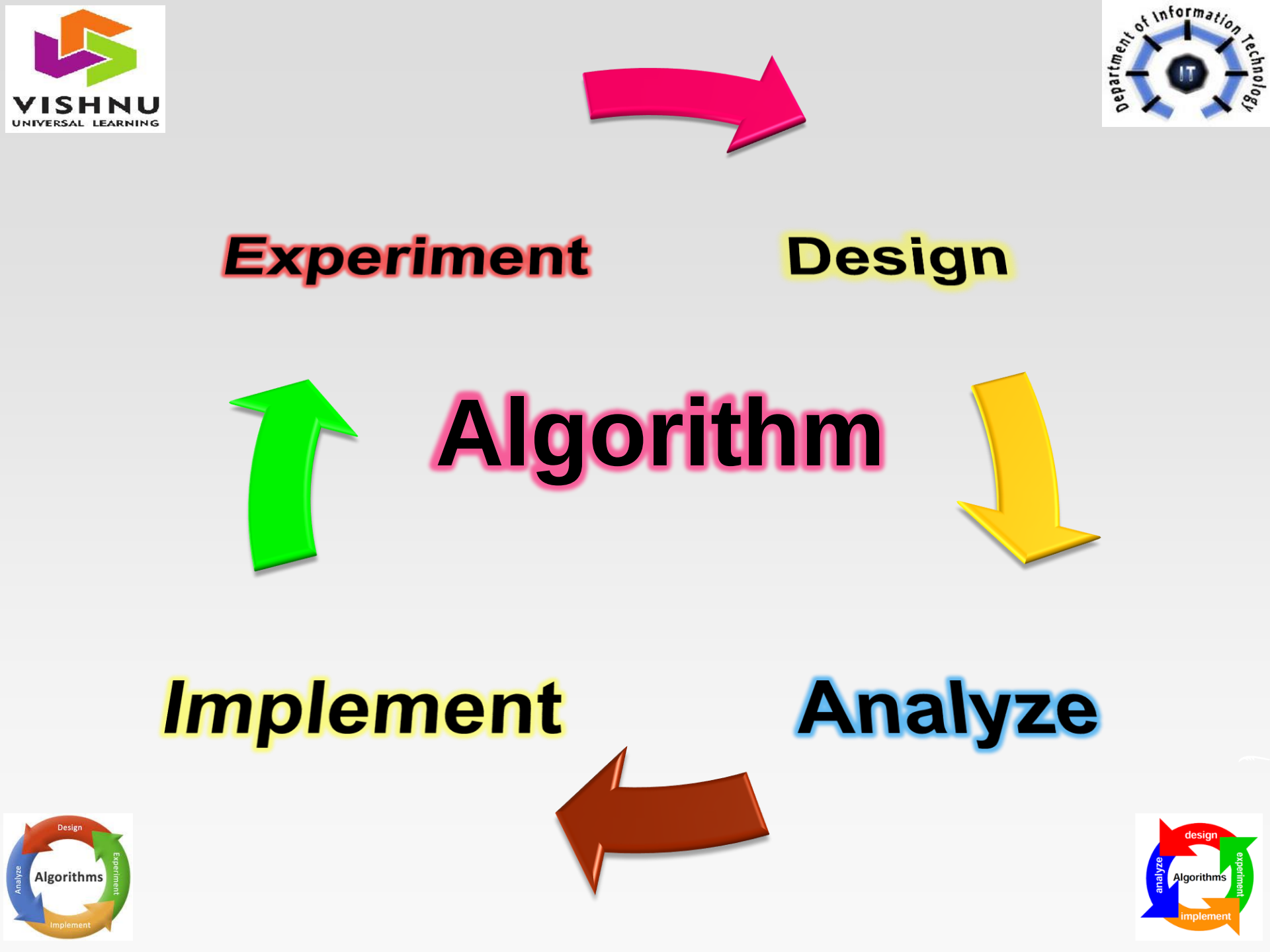


General Discussion about DAA

- In Computer Science, the Design and Analysis of Algorithms is the process of finding the computational complexity of algorithms – the amount of time, storage, or other resources needed to execute them.
- Designing algorithm is necessary before writing the program code as it (algorithm) explains the logic even before the code is developed.
- The code can be written in any programming language but the algorithm is written in a common language.
- As we know that for solution of any problem there may exist many versions of the same program written by same or different programmers. Just reading the source code the efficiency of the code can't be judged.
- So we design algorithms and analyze them for Time Complexity, Space Complexity, Efficiency etc.

General Discussion about DAA

- We can go about creating a program and in most of the cases we might be able to make a decent one as well but some else used algorithm and design to make it more optimized and interactive, everyone will definitely consider the optimized one.
- Knowing algorithm and design we can do some work prior to starting the coding giving us a proper and optimized techniques to code a program.
- If we are going to develop programs, we need to have knowledge of how to design and analyze algorithms to make an effective algorithm.
- Figure out how to apply technique for analysis algorithm that are solve many problems these technique are used to solve problem and find best case, average case, worst case.


Experiment
Design
Algorithm
Implement
Analyze

Algorithm

- **What is an Algorithm?**
- Webster's dictionary defines Algorithm as “Any special method of solving a certain kind of problem."
- **Informal Definition**: An Algorithm is any well-defined computational procedure that takes some value or set of values as Input and produces a set of values or some value as output. Thus algorithm is a sequence of computational steps that transforms the I/P into the O/P.
- **Formal Definition**: An Algorithm is a finite set of instructions that, if followed, accomplishes a particular task.

Algorithm

- In addition, all algorithms should satisfy the following criteria.

1. **INPUT** → Zero or more quantities are externally supplied.
2. **OUTPUT** → At least one quantity is produced.
3. **DEFINITENESS** → Each instruction is clear and unambiguous. For example, each operation must be definite, meaning that it must be perfectly clear what should be done.

   “Compute 5/0" or "add 6 or 7 to x" are not permitted because it is not clear what the result is or which of the two possibilities should be done.
4. **FINITENESS** → If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite no of steps.
5. **EFFECTIVENESS** → Every instruction must very basic so that it can be carried out, in principle, by a person using only pencil & paper.

Algorithm

- **Issues or study of Algorithms:**

- How to devise or design algorithms → creating and algorithm.
- How to express algorithms → definiteness.
- How to validate algorithms → fitness.
- How to analyze algorithms → time and space complexity.
- How to test a program - Testing the algorithm → checking for error (Debugging), Program Proving, Testing.

Algorithm Specification

- **Algorithm can be described in three ways.**

1. **Natural language like English** : When this way is chosen care should be taken, we should ensure that each & every statement is definite.
2. **Graphic representation called flowchart** : This method will work well when the algorithm is small & simple.
3. **Pseudo-code Method** : In this method, we should typically describe algorithms as program, which resembles language like PASCAL & ALGOL.

Pseudo Code for Expressing Algorithms

- **Pseudo-Code Conventions**

1. **Comments** begin with **//** and continue until the end of line.
2. **Blocks** are indicated with matching braces **{** and **}**.
3. An **identifier** begins with a letter. The data types of variables are not explicitly declared.
4. **Compound data types** can be formed with records. Here is an example,

```
Node = Record
{
   data type – 1   data-1;
          .
          .
          .
  data type – n  data – n;
  node * link;
}
```

Here link is a pointer to the record type node. Individual data items of a record can be accessed with → and period.

Pseudo Code for Expressing Algorithms

5. **Assignment** of values to variables is done using the assignment statement.

<Variable>:= <expression>;

6. There are two **Boolean** values TRUE and FALSE.

→ **Logical Operators** AND, OR, NOT

→ **Relational Operators** <, <=,>,>=, =, !=

7. The following **looping** statements are employed.

While, For and repeat-until

While Loop:

```
While < condition > do
{
        <statement-1>
              .
              .
        <statement-n>
}
```

Pseudo Code for Expressing Algorithms

For Loop:

```
For variable: = value-1 to value-2 step step do
 {
   <statement-1>
          .
          .
   <statement-n>
}
```

repeat-until:

```
repeat
   <statement-1>
          .
          .
   <statement-n>
until<condition>
```

Pseudo Code for Expressing Algorithms

8. A **conditional** statement has the following forms.

→ If <condition> then <Statement>

→ If <condition> then <Statement-1>
Else <Statement-2>

→ **Case statement:**

```
Case
{
        : <condition-1> : <statement-1>
                          .
        : <condition-n> : <statement-n>
        : else : <statement-n+1>
}
```

9. **Input and output** are done using the instructions **read & write**.

10. There is only one type of procedure: **Algorithm**. The algorithm consists of heading & body, the heading takes the form,

Algorithm Name (Parameter lists)

11. We use notation **a [i : j]** to indicate Array elements **a [i] to a [j]**

Example : Max of n numbers

```
1.    algorithm Max(A, n)
2.    // A is an array of size n
3.    {
4.     Result := A[1];
5.     for i := 2 to n do
6.       if A[ i ] > Result then
7.             Result := A[ i ];
8.      return Result;
9.   }
```

- In this algorithm (named **Max**), **A & n** are procedure parameters. **Result & i** are Local variables.

Example : Selection Sort

- Suppose we Must devise an algorithm that sorts a collection of n>=1 elements of arbitrary type.
- A Simple solution given by the following.
- From those elements that are currently unsorted, find the smallest & place it next in the sorted list.
- **Simple Algorithm :**

1. for i:= 1 to n do
2. {
3. Examine a[i] to a[n] and suppose the smallest element is at a[j];
4. Interchange a[i] and a[j];
5. }

Example : Selection Sort

- Finding the smallest element (say a[j]) and interchanging it with a[i]
- We can solve the latter problem using the code,

```
t    := a[ i ];
a[ i ] := a[ j ];
a[ j ] := t;
```

- The first subtask can be solved by assuming the minimum is a[i]; checking a[i] with a[i+1], a[i+2]……., and whenever a smaller element is found, regarding it as the new minimum. a[n] is compared with the current minimum.
- Putting all these observations together, we get the algorithm Selection sort.

Example : Selection Sort

Theorem: Algorithm selection sort(a, n) correctly sorts a set of n>=1 elements .The result remains is a a[1:n] such that a[1] <= a[2] ….<=a[n].

- **Selection Sort:** Selection Sort begins by finding the least element in the list. This element is moved to the front.
- Then the least element among the remaining element is found out and put into second position.
- This procedure is repeated till the entire list has been studied.
- Example**: LIST L = 3, 5, 4, 1, 2**
  - 1 is selected → **1,** 5, 4, 3, 2
  - 2 is selected → 1, **2,** 4, 3, 5
  - 3 is selected → 1, 2, **3,** 4, 5
  - 4 is selected → 1, 2, 3, **4,** 5

Example : Selection Sort

```
1. Algorithm selection sort (a, n)
2. // Sort the array a[1:n] into non-decreasing order.
3.  {
4.      for i:=1 to n do
5.      {
6.         j := i;
7.         for k := i+1 to n do
8.           if (a[ k ] < a[ j ]) then j := k;
9.             t := a[ i ];
10.          a[ i ] := a[ j ];
11.          a[ j ] := t;
12.    }
13. }
```

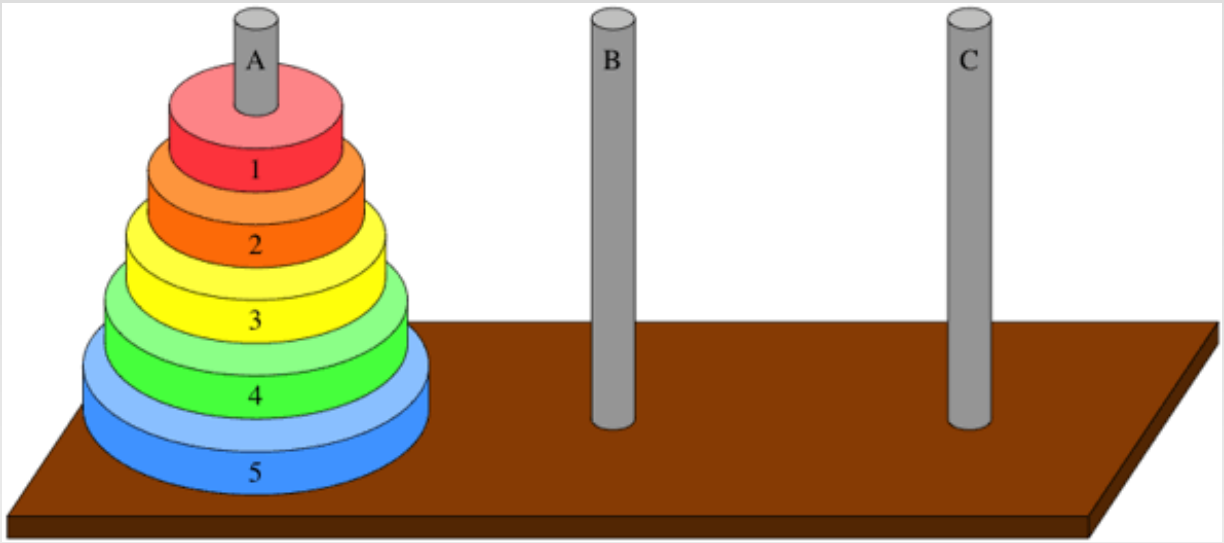
Recursive Algorithms

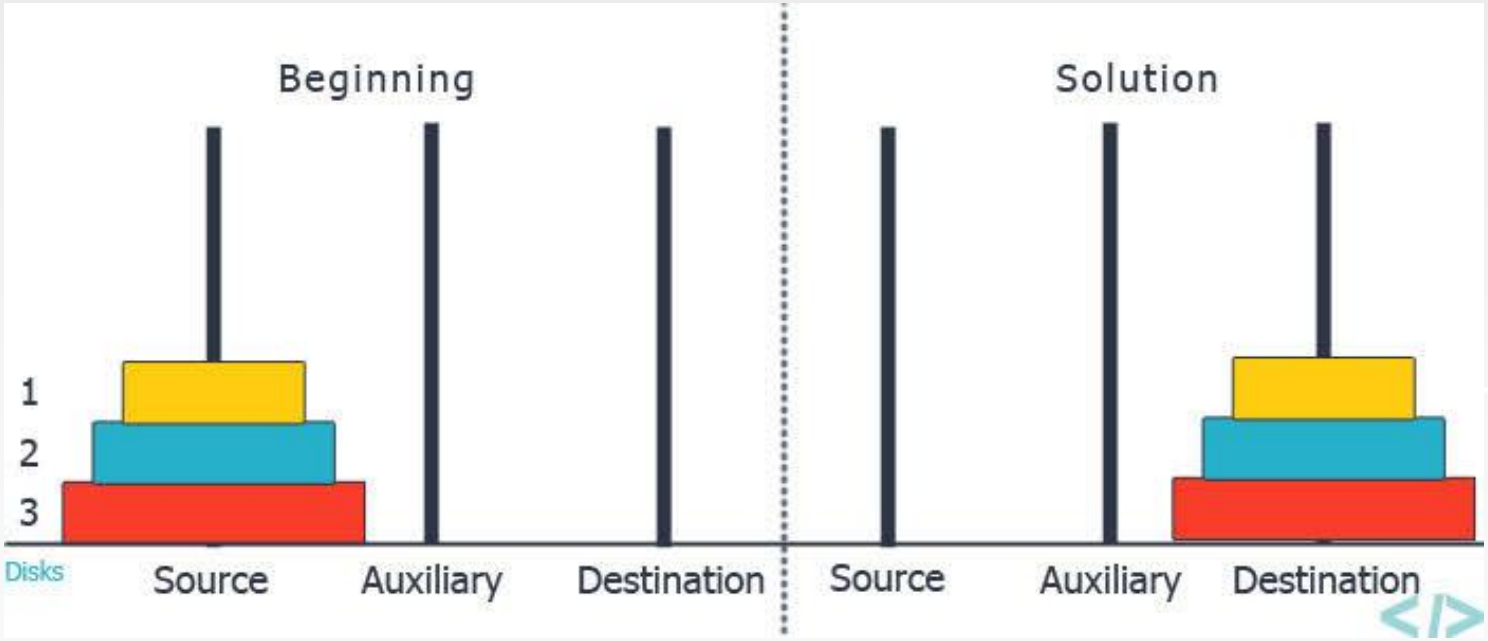
- A **Recursive** function is a function that is defined in terms of itself.
- Similarly, an algorithm is said to be **Recursive Algorithm** if the same algorithm is invoked in the body.
- An algorithm that calls itself is **Direct Recursive**.
- Algorithm ‘A’ is said to be **Indirect Recursive** if it calls another algorithm say ‘B’ which in turns calls ‘A’.
- The Recursive mechanism, are externally powerful, but even more importantly, many times they can express an otherwise complex process very clearly.
- The following 2 examples show how to develop a recursive algorithms.

Example : Towers of Hanoi

- It is Fashioned after the ancient tower of Brahma ritual.
- According to legend, at the time the world was created, there was a diamond tower (labeled A) with 64 golden disks.
- The disks were of decreasing size and were stacked on the tower in decreasing order of size bottom to top.
- Besides these tower there were two other diamond towers (labeled B & C)
- Since the time of creation, Brahman priests have been attempting to move the disks from tower A to tower B using tower C, for intermediate storage.
- As the disks are very heavy, they can be moved only one at a time.
- In addition, at no time can a disk be on top of a smaller disk.
- According to legend, the world will come to an end when the priest have completed this task.

Example : Towers of Hanoi





Example : Towers of Hanoi

- A very elegant solution results from the use of recursion.
- Assume that the number of disks is ‘n’.
- To get the largest disk to the bottom of tower B, we move the remaining ‘n-1’ disks to tower C and then move the largest to tower B.
- Now we are left with the tasks of moving the disks from tower C to B.
- To do this, we have tower A and B available.
- The fact, that towers B has a disk on it can be ignored as the disks larger than the disks being moved from tower C and so any disk can be placed on top of it.

Example : Towers of Hanoi

```
1. Algorithm TowersOfHanoi (n, x, y, z)
2. //Move the top ‘n’ disks from tower x to tower y.
3. {
4.   if(n>=1) then
5.   {
6.     TowersOfHanoi(n-1, x, z, y);
7.     Write(“Move Top Disk from Tower”, x , “to Top of
       Tower”, y);
8.     TowersOfHanoi(n-1, z, y, x);
9.    }
10. }
```

Example : Permutation Generator

- Given a set of n>=1 elements, the problem is to print all possible permutations of this set.
- For example, if the set is {a, b, c} ,then the set of permutation is, { (a,b,c),(a,c,b),(b,a,c),(b,c,a),(c,a,b),(c,b,a) }
- It is easy to see that given ‘n’ elements there are n! different permutations.
- A simple algorithm can be obtained by looking at the case of 4 statement (a, b, c, d)
- The Answer can be constructed by writing
    1. a followed by all the permutations of (b, c, d)
    2. b followed by all the permutations of (a, c, d)
    3. c followed by all the permutations of (a, b, d)
    4. d followed by all the permutations of (a, b, c)

Example : Permutation Generator

```
1.  Algorithm Perm(a, k, n)
2.  {
3.    if (k = n) then write (a[ 1:n ]);   // Output
      permutation
4.     else   //a[ k:n ] has more than one permutation
5.              // Generate this recursively.
6.       for i := k to n do
7.         {
8.           t := a[ k ];  a[ k ] := a[ i ]; a[ i ] := t;
9.           Perm(a, k+1, n);
10.         //All permutation of a[ k+1 : n ]
11.         t := a[ k ];  a[ k ] := a[ i ]; a[ i ] := t;
12.       }
13.  }
```

Performance Analysis

- There are many criteria upon which we can judge an algorithm

1. Does it do what we want it to do?
2. Does it work correctly according to the original specifications of the task?
3. Is there documentation that describes how to use it and how it works?
4. Are procedures created in such a way that they perform logical sub-functions?
5. Is the Code readable?

- These criteria are important for writing software.
- There are other Criteria for judging algorithms that have a more direct relationship to performance – **Space & Time**.

Performance Analysis

- **Space Complexity** : The space complexity of an algorithm is the amount of memory it needs to run to completion.

- **Time Complexity** : The time complexity of an algorithm is the amount of computer time it needs to run to compilation.

Space Complexity

- **Space Complexity :** The Space needed by each of these algorithms is seen to be the Sum of the following component.

1. A fixed part that is independent of the characteristics (eg: number, size) of the inputs and outputs. This part typically includes the **instruction space** (i.e. Space for the code), space for **simple variable and fixed-size component variables** (also called aggregate), **space for constants**, and so on.
2. A variable part that consists of the space needed by **component variables** whose size is dependent on the particular problem instance being solved, the **space needed** by **referenced variables** (to the extent that is depends on **instance characteristics**), and the **recursion stack space**.

- The space requirement **S(P)** of any algorithm **P** may therefore be written as,

$$S(P) = c + S_P \text{ (Instance characteristics)}$$

- Where **'c'** is a constant.

Example : Space Complexity

- **Algorithm :** Simple arithmetic function

```
Algorithm  abc (a, b, c)
{
    return a + b + b * c + (a + b - c) / (a + b) + 4.00;
}
```

- Here the problem instance is characterized by the specific values of a, b and c.
- If we assume one word is sufficient to store values of a, b, c and the result, the space needed by abc is independent of instance characteristics.
- So, S_P (Instance characteristics) i.e

  S_{abc}(Instance characteristics) = 0

Example : Space Complexity

- **Algorithm :** Iterative function for summing a list of numbers

```
1.  Algorithm Sum (a, n)
2.   {
3.        s := 0.0;
4.        for i := 1 to n do
5.        s := s + a[ i ];
6.        return s;
7.   }
```

- The problem instances for this algorithm are characterized by **n**, the number of elements to be summed.
- The space needed by **'n'** is one word, since it is of type integer.
- The space needed by **'a'** is the space needed by variables of type array of floating point numbers.
- This is at least **'n'** words, since **'a'** must be large enough to hold the 'n' elements to be summed.

So, we obtain $\mathbf{S_{Sum}(n) >= (n + 3)}$

(**n for a[ ], one each for n, i and s**)

Example : Space Complexity

- **Algorithm :** Recursive function for summing a list of numbers

```
1.  Algorithm RSum (a, n)
2.   {
3.       if (n <= 0) then return 0.0;
4.       else return RSum (a, n-1) + a[ n ];
5.   }
```

- Similar to Sum function, the problem instances for this algorithm are characterized by n.
- The recursion stack space includes space for **formal parameters,** the **local variables** and **return addresses**. Assume return address requires one word.
- So each call to RSum requires at least three words (including space for values of n, return address & pointer to a [ ]).
- Since depth of recursion is **n+1**, the **recursion stack space** needed is **> = 3 (n+1).**

Time Complexity

- **Time Complexity :** The time **T(P)** taken by a program P is the sum of the **Compile Time** and the **Run Time** (execution time).
- The compile time does not depend on the instance characteristics.
- Also we may assume that a compiled program will be run several times without recompilation .
- This run time is denoted by **tp** (instance characteristics).
- As many of factors **tp** depends on are not known at the time a program is conceived, we will attempt only to estimate **tp.**
- We will calculate count for total number of operations – **Steps**.
- A **program Step** is a syntactically or semantically meaningful program segment whose execution time is independent of the instance characteristics.

Time Complexity

- The number of steps any problem statement is assigned depends on the kind of statement. For example,
- **Comments → 0 steps** Non executable statements.
- **Assignment Statements → 1 steps.**

  [Which does not involve any calls to other algorithms]
- Interactive statements (loops) such as for, while & repeat-until → **Control part of the statement.**
- We can determine the number of steps needed by a program to solve a particular problem instance in two ways.

1. Introduce variable **count** into programs
2. **Tabular** method
   - Determine the total number of steps contributed by each statement **step per execution × frequency**
   - Add up the contribution of all statements

Time Complexity of Sum using Count

- We introduce a variable, **count** into the program.
- This is global variable with initial value **0**.
- Statement to increment **count** by the appropriate amount are introduced into the program.
- This is done so that each time a statement in the original program is executes count is incremented by the step count of that statement.
- Each invocation of Sum executes a total of **2n+3** steps

```
1.  Algorithm Sum (a, n)
2.  {
3.     s= 0.0;     count := count+1;
4.     for i := 1 to n do
5.     {
6.        count := count + 1; //For for loop
7.        s := s+ a [ i ];
8.        count := count + 1;
9.                 // For assignment
10.    }
11.    count := count + 1; // For last for
12.    count := count + 1; // For return
13.     return s;
14.  }
```

Time Complexity of Sum using Table

- The second method to determine the step count of an algorithm is to build a table in which we list the total number of steps contributes by each statement.
- First determine the number of **steps per execution (s/e)** of the statement and the total number of times (i.e., **frequency**) each statement is executed.
- By combining these two quantities, the total contribution of all statements, the step count for the entire algorithm is obtained.

Statement	s/e	Frequency	Total
1. Algorithm Sum (a, n)	0	-	0
2. {	0	-	0
3. s := 0.0;	1	1	1
4. for i := 1 to n do	1	n+1	n+1
5. s := s + a[i];	1	n	n
6. return s;	1	1	1
7. }	0	-	0
Total			2n + 3

Time Complexity of RSum using Count

```
1.  Algorithm RSum (a, n)
2.  {
3.    count := count + 1; // For if cond.
4.    if(n<=0) then
5.    {
6.      count := count + 1; //For return
7.       return 0.0;
8.    }
9.    else
10.   {
11.   count := count + 1; // For call
12.    return RSum(a, n-1) + a[n];
13.   }
14. }
```

Time Complexity of RSum using Count

- When the statements to increment count introduced into RSum algorithm we get algorithm as shown in previous slide.
- Let $\mathbf{t_{RSum}(n)}$ be the increase in the value of count at end of Algo.
- We see that $\mathbf{t_{RSum}(0) = 2}$.
- & when n>0, it's $\mathbf{2 + t_{RSum}(n-1)}$
- When analyzing a recursive program for its step count
- $$t_{RSum}(n) = \begin{cases} 2 & if\ n = 0 \\ 2 + t_{RSum}(n-1) & if\ n>0 \end{cases}$$
- These recursive formulas are referred as recurrence relations.

$$t_{RSum}(n) = 2 + t_{RSum}(n-1)$$

$$t_{RSum}(n) = 2 + 2 + t_{RSum}(n-2) = 2(2) + t_{RSum}(n-2)$$

$$\ldots\ldots\ldots$$

$$t_{RSum}(n) = n(2) + t_{RSum}(0)$$

$$t_{RSum}(n) = 2n + 2 \qquad n>=0$$

- So the step count for **RSum is 2n + 2**

Time Complexity of RSum using Table

- The below table gives the step count for RSum, note that the **else** clause has been given count of **1 + t_{RSum} (n-1)**.
- This is the total cost of this line each time it is executed.
- It includes all the steps that get executed as a result of invocation of RSum from the **else** clause.

Statement	s/e	Frequency		Total Steps	
		n =0	*n>0*	*n =0*	*n>0*
1. Algorithm RSum (a, n)	0	-	-	0	0
2. {	0	-	-	0	0
3. if (n <= 0) then	1	1	1	1	1
4. return 0.0;	1	1	0	1	0
5. else	0	-	-	0	0
6. return RSum(a, n-1)+a[n];	1+x	0	1	0	1+x
7. }	0	-	-	0	0
Total				2	2 + x
					x = t_{RSum} (n-1)

Time Complexity of Matrix Addition algorithm using Count

```
1.  Algorithm Add(a, b, c, m, n)
2.  {
3.   for i := 1 to m do
4.  {
5.    count := count + 1; // For 'for i'.
6.     for j := 1 to n do
7.     {
8.       count := count + 1; //For 'for j'
9.       c [i, j] := a [i, j] + b [i, j];
10.      count := count + 1; //For assignment
11.   }
12.    count := count + 1; // For last time of 'for j'
13.  }
14.  count := count + 1; // For last time of 'for i'
15. }
```

Time Complexity of Matrix Addition algorithm using Count

- Algorithm Add shown in previous slide will be used to add two
  m x n matrices **a & b**.
- We observe that **for j** is executed **n** times for each value of **i** or a total of **mn** times (so **for** loop & **addition** of elements executed for a total of **2mn** times).
- also **for i** is executed **m** times (so **for i** loop & last of **for j** loop executed for a total of **2m** times)
- and line 14 (last of **for i**) executed once.
- If count is 0 to begin, it will be **2mn + 2m + 1** when algorithm terminates.

Time Complexity of Matrix Addition algorithm using Tabular Method

- Note that the frequency of the first **for** loop is **m + 1** & not m.
- And similarly frequency of the second **for** loop is **m (n + 1)**.
- Total step count for Add algorithm is **2mn + 2m + 1**

Statement	s/e	Frequency	Total
1. Algorithm Add(a, b, c, m, n)	0	-	0
2. {	0	-	0
3. for i := 1 to m do	1	m + 1	m + 1
4. for j := 1 to n do	1	m(n+1)	m(n+1)
5. c [i, j] := a [i, j] + b [i, j];	1	mn	mn
6. }	0	-	0
Total			2mn + 2m + 1

Time Complexity - Exercise

- Write Algorithm and Find Time Complexity for
  - Fibonacci Series
  - Matrix Multiplication – Square Matrix
  - Matrix Multiplication – Non Square Matrix – A(M,N) & B(N,K)
  - Try to find time complexity for few more Programs/Algorithms you studied previously.

Summary of Time Complexity

- The time complexity of an algorithm is given by the number of steps taken by Algorithm to compute the function is was written for.
- The number of steps is itself function of instance characteristics.
- Any specific instance may have several characteristics like no of inputs/outputs, magnitude of input/output, etc.
- Sometimes we want to know how run time increases as number of inputs increase.
- The examples we have seen are simple that the time complexities are function of simple characteristics like no of inputs or no of rows & columns, etc.
- For many algorithm, the time complexity is not solely dependent on no of inputs or output. Ex : Searching.
- In such situation where chosen parameters are not sufficient to find step, count we can define three types of step counts –

  Best Case, Worst Case & Average Case.

Summary of Time Complexity

- **Best Case :** The Best case count is the minimum number of steps that can be executed for the given parameters.
- **Worst Case :** The Worst case count is the maximum number of steps that can be executed for the given parameters.
- **Average Case :** The average case count is the average number of steps that can be executed for the given parameters.
- Our motivation is to determine step count to compare time complexities of two algorithm that compute same function.
- Determining the exact step count (**Best Case, Worst Case & Average Case**) of algorithm is difficult task.
- **Complexity:** Complexity refers to the rate at which the storage time grows as a function of the problem size.
- **Asymptotic Analysis:** Expressing the complexity in term of its relationship to know function. This type analysis is called asymptotic analysis.

Asymptotic Notations (O, Ω, Θ)

- Let us see some terminology that enables us to make meaningful statements about the time & space complexities of algorithm.
- We will use functions ***f*** and ***g*** are nonnegative functions.

Asymptotic Notation - Big 'oh' (O)

- **Big 'oh' (O) :** The function ***f(n)* = O (*g(n)*)** [Read as "f of n is big oh of g of n"] iff (if and only if) there exist positive constants **c** and $\mathbf{n_0}$ such that **f(n) ≤ c * g(n) for all n, n ≥ $\mathbf{n_0}$.**

Example : 1) The function 3n + 2 = O(n) as 3n + 2 ≤ 4n for all n ≥ 2

2) 3n + 3 = O(n) as 3n + 3 ≤ 4n for all n ≥ 3.

3) 100n + 6 = O(n) as 100n + 6 ≤ 101n for all n ≥ 6.

4) $10n^2 + 4n + 2 = O(n^2)$ as $10n^2 + 4n + 2 \leq 11n^2$ for all n ≥ 5.

5) $1000n^2 + 100n - 6 = O(n^2)$ as $1000n^2 + 100n - 6 \leq 1001n^2$ for all n ≥ 100.

6) $6 * 2^n + n^2 = O(2^n)$ as $6 * 2^n + n^2 \leq 7 * 2^n$ for all n ≥ 4.

Asymptotic Notations (O, Ω, Θ)

- Following seven computing times are common & we will use these throughout our syllabus.

1. We use **O(1)** to mean a computing time that is a Constant.
2. **O(n)** is called Linear.
3. **O(n^2)** is called Quadratic.
4. **O(n^3)** is called Cubic.
5. **O(2^n)** is called Exponential.
6. Similarly we can use **O(log n)** to mean logarithmic and
7. **O(n log n)** to mean linear logarithmic.

- The statement ***f(n)* = O (*g(n)*)** states only that **g(n)** is an **upper** bound on the value of f(n) for all **n, n ≥ n_0.**

Asymptotic Notation - Omega (Ω)

- **Omega (Ω) :** The function ***f(n)* = Ω (*g(n)*)** [Read as "f of n is omega of g of n"] iff (if and only if) there exist positive constants **c** and $\mathbf{n_0}$ such that **f(n) ≥ c * g(n) for all n, n ≥ $\mathbf{n_0}$.**

Example : 1) The function 3n + 2 = Ω (n) as 3n + 2 ≥ 3n for all n ≥ 1.

2) 3n + 3 = Ω (n) as 3n + 3 ≥ 3n for all n ≥ 1.

3) 100n + 6 = Ω (n) as 100n + 6 ≥ 100n for all n ≥ 1.

4) $10n^2 + 4n + 2 = \Omega(n^2)$ as $10n^2 + 4n + 2 \geq n^2$ for all n ≥ 1.

5) $6 * 2^n + n^2 = \Omega(2^n)$ as $6 * 2^n + n^2 \geq 2^n$ for all n ≥ 1.

- As in the case of the big oh notation, there are several function **g(n)** for which **f(n) = Ω (g(n))**.
- The function **g(n)** is only a **lower** bound of **f(n)**.

Asymptotic Notation - Theta (Θ)

- **Theta (Θ) :** The function ***f(n)* = Θ (*g(n)*)** [Read as "f of n is theta of g of n"] iff (if and only if) there exist positive constants **c_1, c_2,** and **n_0** such that

  $c_1 * g(n) \leq f(n) \leq c_2 * g(n)$ for all n, $n \geq n_0$.

Example : 1) The function $3n + 2 = \Theta(n)$ as $3n + 2 \geq 3n$ for all $n \geq 2$ and $3n + 2 \leq 4n$ for all $n \geq 2$, so $c_1=3$, $c_2=4$ and $n_0=2$

2) $3n + 3 = \Theta(n)$.

3) $10n^2 + 4n + 2 = \Theta(n^2)$.

4) $6 * 2^n + n^2 = \Theta(2^n)$.

5) $10 * \log n + 4 = \Theta(\log n)$

- The Theta notation is more precise than both the Big oh & Omega notations.
- The function ***f(n)* = Θ (*g(n)*)** iff **g(n)** is both an **upper and lower** bound of **f(n)**.

Asymptotic Notations – Little 'oh' (o) & Little omega (ω)

- **Little 'oh' (o) :** The function ***f(n)* = o (*g(n)*)** [Read as "f of n is little oh of g of n"] iff (if and only if)

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

Example : 1) The function $3n + 2 = o(n^2)$ since $\lim_{n\to\infty} \frac{3n+2}{n^2} = 0$

Even $3n + 2 = o(n \log n)$

2) $6 * 2^n + n^2 = o(3^n)$ even $6 * 2^n + n^2 = o(2^n \log n)$

- **Little omega (ω) :** The function ***f(n)* = ω (*g(n)*)** [Read as "f of n is little omega of g of n"] iff (if and only if)

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$$

- Let us reexamine the time complexity analyses of Sum, RSum & Add (Matrix Addition) algorithms.
- For **Sum** algorithm we determined that

  $t_{Sum}(n) = 2n + 3$, so $t_{Sum}(n) = \Theta(n)$
- For **RSum** algorithm, $t_{Sum}(n) = 2n + 2 = \Theta(n)$

Asymptotic Complexity of Sum Algorithm

Statement	s/e	Frequency	Total
1. Algorithm Sum (a, n)	0	-	Θ (0)
2. {	0	-	Θ (0)
3. s := 0.0;	1	1	Θ (1)
4. for i := 1 to n do	1	n+1	Θ (n)
5. s := s + a[i];	1	n	Θ (n)
6. return s;	1	1	Θ (1)
7. }	0	-	Θ (0)
Total			Θ (n)

Asymptotic Complexity of RSum Algo.

Statement	s/e	Frequency		Total Steps	
		n =0	*n>0*	*n =0*	*n>0*
1. Algorithm RSum (a, n)	0	-	-	0	Θ (0)
2. {	0	-	-	0	Θ (0)
3. if (n <= 0) then	1	1	1	1	Θ (1)
4. return 0.0;	1	1	0	1	Θ (0)
5. else	0	-	-	0	Θ (0)
6. return RSum(a, n-1)+a[n];	1+x	0	1	0	Θ (1+x)
7. }	0	-	-	0	Θ (0)
Total				2	Θ (1+x)
					$x = t_{RSum}(n-1)$

- Although we might see that the O, Ω, Θ notations have been used correctly, do we first need to find exact step count?
- The answer to this question is that the Asymptotic Complexity can be determined easily without determining the exact step count.
- This is usually done by first determining the Asymptotic Complexity of each statement & adding these complexities.

- Although the analyses are carried out in terms of step counts, it is correct to interpret $t_P(n) = O(g(n))$, $t_P(n) = \Omega(g(n))$, or $t_P(n) = \Theta(g(n))$ as a statement about computing time of algorithm **P**.
- This is so because each step takes on $\Theta(1)$ time to execute.

Asymptotic Complexity of Matrix Addition Algo.

Statement	s/e	Frequency	Total
1. Algorithm Add(a, b, c, m, n)	0	-	$\Theta(0)$
2. {	0	-	$\Theta(0)$
3. for i := 1 to m do	1	$\Theta(m)$	$\Theta(m)$
4. for j := 1 to n do	1	$\Theta(mn)$	$\Theta(mn)$
5. c [i, j] := a [i, j] + b [i, j];	1	$\Theta(mn)$	$\Theta(mn)$
6. }	0	-	$\Theta(0)$
Total			$\Theta(mn)$

Next - Unit II

Disjoint Sets

Divide and Conquer