CSE3023-Design and Analysis of Algorithms

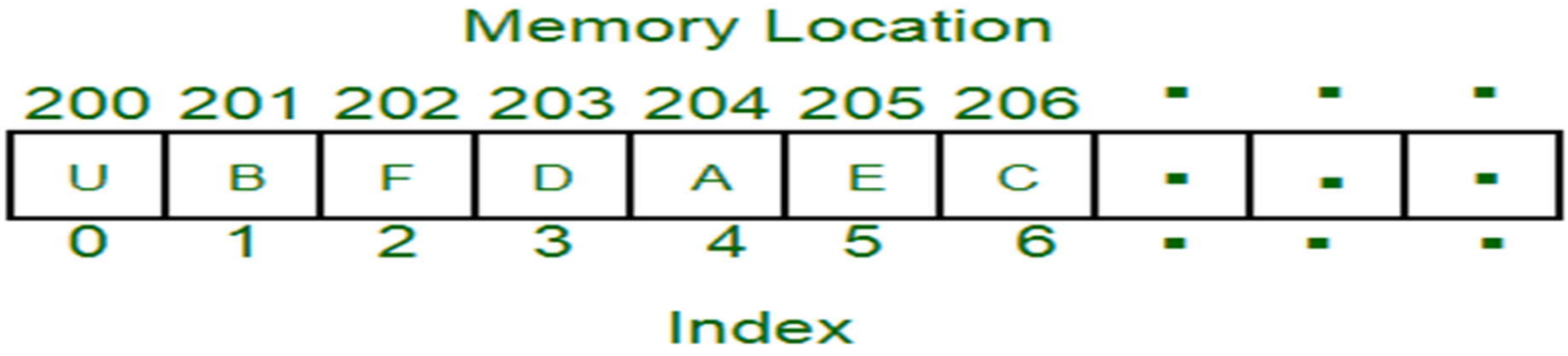
ZERO HOUR

Instructor :Dr. B. Sai Chandana

Course Code: CSE3023	**Course Title:** Design and Analysis of Algorithms	**TPC**	3	2	4
Version No.	1.0				
Course Pre-requisites/ Co-requisites	CSE2014 - Data Structures and Algorithms				
Anti-requisites (if any).	None				
Objectives:	• To understand the basics of computational complexity analysis and various algorithm design paradigms • The objective is to supply students with solid foundations to deal with a wide assortment of computational issues, and to provide a thorough information of the foremost common algorithms and data structures.				

Data Structures and Algorithm

A **data structure** is a particular way of organizing data in a computer so that it can be used effectively.
For example, we can store a list of items having the same data-type using the *array* data structure.



Types of Data Structure

Basically, data structures are divided into two categories:

- Linear data structure
- Non-linear data structure

Linear data structures

In linear data structures, the elements are arranged in sequence one after the other. Since elements are arranged in particular order, they are easy to implement.

List of data structure in a linear type of data structure

1. Array
2. Linked list
3. Stack
4. Queue

Non-linear data structure

•A non-linear data structure is another important type in which data elements are not arranged sequentially; mainly, data elements are arranged in random order without forming a linear structure.

- Map
- Graph
- Tree

Course Outcomes

1. To apply knowledge of computing and mathematics to algorithm design.

2. Describe the brute force and divide-and-conquer paradigm and explain when an algorithmic design situation calls for it. Derive and solve recurrences describing the performance of divide-and-conquer algorithms.

3. To examine the concepts of Graph & Tree Algorithms and Tractable and Intractable Problems.

4. Familiarize on the advanced concepts of Randomized algorithms and Quantum Algorithms

Syllabus

Module No. 1	Introduction	6 Hours
Characteristics of Algorithm. Analysis of Algorithm: Asymptotic analysis of Complexity Bounds – Best, Average and Worst-Case behavior; Performance Measurements of Algorithm, Time and Space Trade-Offs.		
Module No. 2	**Recursive Algorithms**	**8 Hours**
Analysis of Recursive Algorithms through Recurrence Relations: Substitution Method, Recursion Tree Method and Masters' Theorem, Fundamental Algorithmic Strategies: Brute-Force, Heuristics, Greedy, Dynamic Programming.		
Module No. 3	**Algorithmic Strategies**	**8 Hours**
Branch and Bound and Backtracking methodologies; Illustrations of these techniques for Problem-Solving, Bin Packing, Knapsack, Travelling Salesman Problem.		

Module No. 4	Graph and Tree Algorithms	7 hours
Traversal algorithms: Depth First Search (DFS) and Breadth First Search (BFS); Shortest path algorithms, Transitive closure, Minimum Spanning Tree, Topological sorting, Network Flow Algorithm.		
Module No. 5	**Tractable and Intractable Problems**	**8 hours**
Computability of Algorithms, Computability classes – P, NP, NP-complete and NP-hard. Cook's theorem, Standard NP-complete problems and Reduction techniques.		
Module No. 6	**Advanced Topics**	**8 hours**
Approximation algorithms, Randomized algorithms, Class of problems beyond NP – P SPACE, Introduction to Quantum Algorithms.		

Text Books

1. Fundamental of Computer Algorithms, E. Horowitz and S. Sahni. 2nd Edition,2008.
2. The Design and Analysis of Computer Algorithms, A. Aho, J. Hopcroft and J. Ullman,2014

Mode of Evaluation

Cumulative Lab Assessment	25%
Continuous Assessment Test-1	20%
Continuous Assessment Test-2	20%
Final Assessment Test	20%
Internal assessment	15%

Module No.1: Introduction

Algorithm

- Definition
  An *algorithm* is a finite set of instructions that accomplishes a particular task.
- Criteria
  - input
  - output
  - definiteness: clear and unambiguous
  - finiteness: terminate after a finite number of steps
  - effectiveness: instruction is basic enough to be carried out

Notion of an Algorithm

An ***algorithm*** is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time.

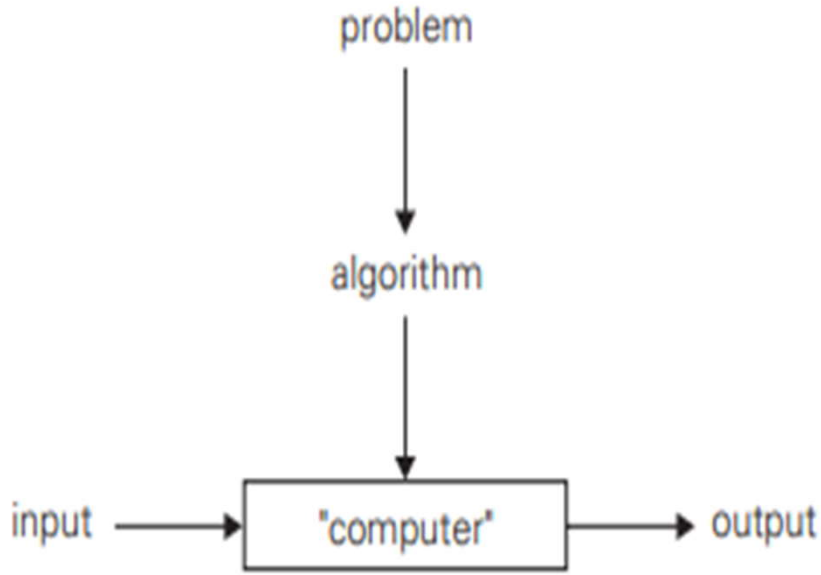


FIGURE 1.1 The notion of the algorithm.

As examples illustrating the notion of the algorithm, we consider in this section three methods for solving the same problem: computing the greatest common divisor of two integers. These examples will help us to illustrate several important points:

- The nonambiguity requirement for each step of an algorithm cannot be compromised.
- The range of inputs for which an algorithm works has to be specified carefully.
- The same algorithm can be represented in several different ways.
- There may exist several algorithms for solving the same problem.
- Algorithms for the same problem can be based on very different ideas and can solve the problem with dramatically different speeds.

What is Greatest Common Divisor?

For a set of positive integers (a, b), the greatest common divisor is defined as the greatest positive number which is a common factor of both the positive integers (a, b). GCD of any two numbers is never negative or 0 as the least positive integer common to any two numbers is always 1. There are two ways to determine the greatest common divisor of two numbers:

- By finding the common divisors
- By Euclid's algorithm

How to Find the Greatest Common Divisor?

For a set of two positive integers (a, b) we use the below-given steps to find the greatest common divisor:

- **Step 1:** Write the divisors of positive integer "a".
- **Step 2:** Write the divisors of positive integer "b".
- **Step 3:** Enlist the common divisors of "a" and "b".
- **Step 4:** Now find the divisor which is the highest of both "a" and "b".

Example: Find the greatest common divisor of 13 and 48.

Solution: We will use the below steps to determine the greatest common divisor of (13, 48).

Divisors of 13 are 1, and 13.

Divisors of 48 are 1, 2, 3, 4, 6, 8, 12, 16, 24 and 48.

The common divisor of 13 and 48 is 1.

The greatest common divisor of 13 and 48 is 1.

Thus, GCD(13, 48) = 1.

Finding Greatest Common Divisor by LCM Method

As per the LCM Method for the greatest common divisor, the GCD of two positive integers (a, b) can be calculated by using the following formula:

$$GCD\ (a, b) = \frac{(a \times b)}{LCM\ (a, b)}$$

The steps to calculate the GCD of (a, b) using the LCM method is:

- **Step 1:** Find the product of a and b.
- **Step 2:** Find the least common multiple (LCM) of a and b.
- **Step 3:** Divide the values obtained in Step 1 and Step 2.
- **Step 4:** The obtained value after division is the greatest common divisor of (a, b).

Example: Find the greatest common divisor of 15 and 70 using the LCM method.

Solution: The greatest common divisor of 15 and 70 can be calculated as:

- The product of 15 and 70 is given as, 15 × 70
- The LCM of (15, 70) is 210.
- GCD (15, 20) = (15 × 70)/ 210 = 5.

∴ The greatest common divisor of (15, 70) is 5.

Euclid's Algorithm for Greatest Common Divisor

As per Euclid's algorithm for the greatest common divisor, the GCD of two positive integers (a, b) can be calculated as:

- If a = 0, then GCD (a, b) = b as GCD (0, b) = b.
- If b = 0, then GCD (a, b) = a as GCD (a, 0) = a.
- If both a≠0 and b≠0, we write 'a' in quotient remainder form (a = b×q + r) where q is the quotient and r is the remainder, and a>b.
- Find the GCD (b, r) as GCD (b, r) = GCD (a, b)
- We repeat this process until we get the remainder as 0.

Example: Find the GCD of 12 and 10 using Euclid's Algorithm.

Solution: The GCD of 12 and 10 can be found using the below steps:

$a = 12$ and $b = 10$

$a \neq 0$ and $b \neq 0$

In quotient remainder form we can write $12 = 10 \times 1 + 2$

Thus, GCD (10, 2) is to be found, as GCD(12, 10) = GCD (10, 2)

Now, $a = 10$ and $b = 2$

$a \neq 0$ and $b \neq 0$

In quotient remainder form we can write $10 = 2 \times 5 + 0$

Thus, GCD (2,0) is to be found, as GCD(10, 2) = GCD (2, 0)

Now, $a = 2$ and $b = 0$

$a \neq 0$ and $b = 0$

Thus, GCD (2,0) = 2

GCD (12, 10) = GCD (10, 2) = GCD (2, 0) = 2

Thus, GCD of 12 and 10 is 2.

Euclid's algorithm is very useful to find GCD of larger numbers, as in this we can easily break down numbers into smaller numbers to find the greatest common divisor.

Fundamentals of Algorithmic Problem Solving

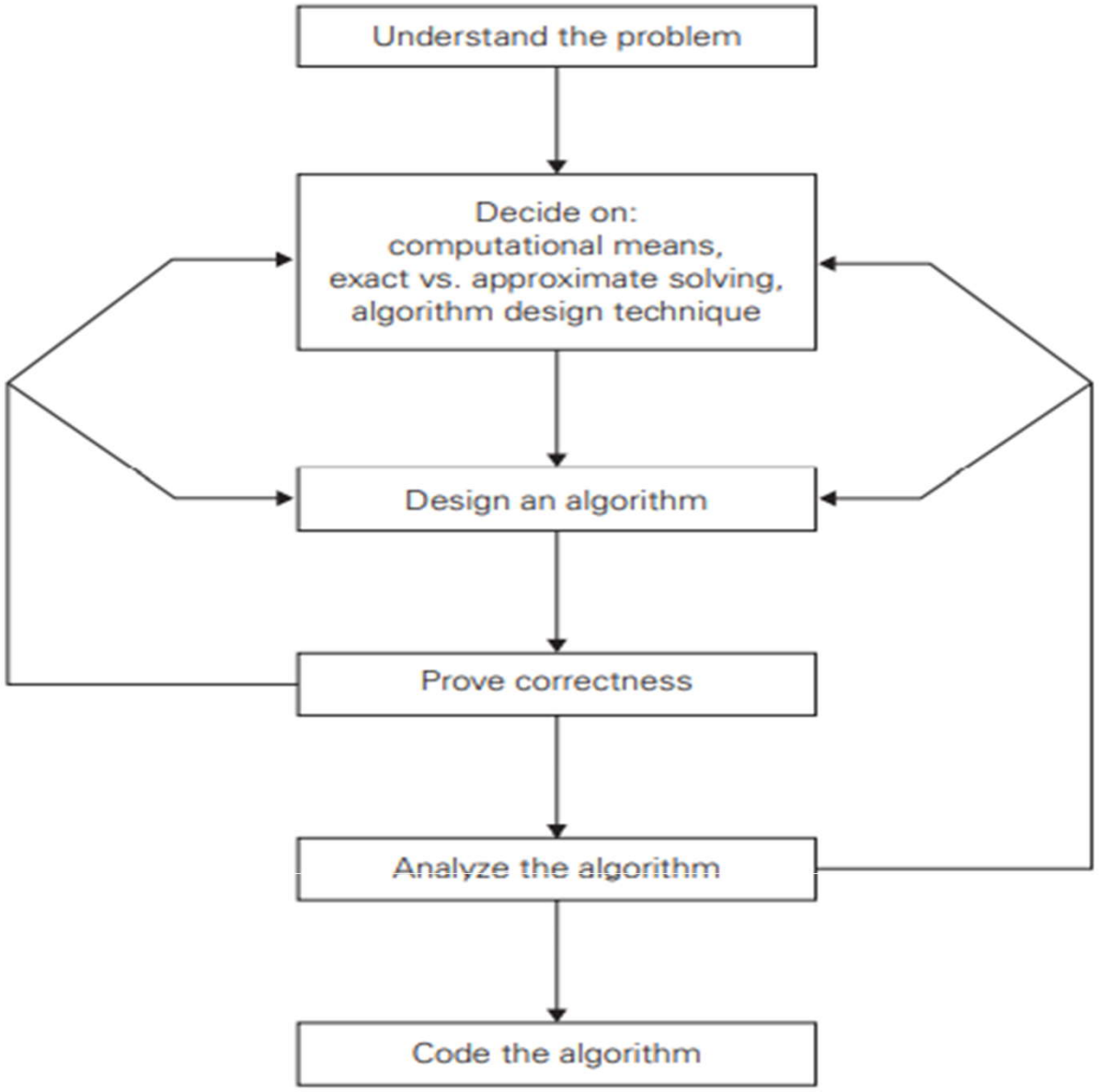


FIGURE 1.2 Algorithm design and analysis process.

An Algorithm Development Process

- How to design Algorithm
  1. **Divide and Conquer Approach**
  2. **Greedy Technique**
  3. **Dynamic Programming**
  4. **Branch and Bound**
  5. **Backtracking Algorithm etc**

- How to validate Algorithm

- How to analyse Algorithm

Space Complexity: The space complexity can be understood as the amount of space required by an algorithm to run to completion.

Time Complexity: Time complexity is a function of input size **n** that refers to the amount of time needed by an algorithm to run to completion.

Generally, we make three types of analysis, which is as follows:

Worst-case time complexity: For '**n**' input size, the worst-case time complexity can be defined as the maximum amount of time needed by an algorithm to complete its execution. Thus, it is nothing but a function defined by the maximum number of steps performed on an instance having an input size of n.

Average case time complexity: For '**n**' input size, the average-case time complexity can be defined as the average amount of time needed by an algorithm to complete its execution. Thus, it is nothing but a function defined by the average number of steps performed on an instance having an input size of n.

Best case time complexity: For '**n**' input size, the best-case time complexity can be defined as the minimum amount of time needed by an algorithm to complete its execution. Thus, it is nothing but a function defined by the minimum number of steps performed on an instance having an input size of n.

- How to test
  - ❖ Debugging
  - ❖ Profiling

ALGORITHM SPECIFICATION

1.Comments begin with // and continue until the end of line.
2.Blocks are indicated with matching braces:{ and }
3. Each Statements delimited by ;
4. An identifier begins with a letter. The data types of variables are not explicitly declare.
5. .Compound data types can be formed with records. Here is an example:

```
 node= record
  {
datatypeA data A;
datatypen   data n;
node *link;
}
```

6. Assignment of values to variables is done using the assignment statement (variable):=(expression)
7. There are two Boolean values true and false.
8. Elements of multi dimensional arrays are accessed using [and].
For example, if A is a two dimensional array, the (i,j) th element of the array is denoted as -A[i,j].Array indices start at zero.
9. The following looping statements are employed :for, while, and repeat- until.

A **repeat-until** statement is constructed as follows:

repeat
⟨*statement* 1⟩
⋮
⟨*statement n*⟩
until ⟨*condition*⟩

10.

A conditional statement has the following forms:

if ⟨*condition*⟩ **then** ⟨*statement*⟩
if ⟨*condition*⟩ **then** ⟨*statement* 1⟩ **else** ⟨*statement* 2⟩

9. Input and output are done using the instructions **read** and **write**. No format is used to specify the size of input or output quantities.

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

    Algorithm *Name* (⟨*parameter list*⟩)

As an example, the following algorithm finds and returns the maximum of n given 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 Result := A[i];
7      return Result;
8  }
```

What is Recursion?

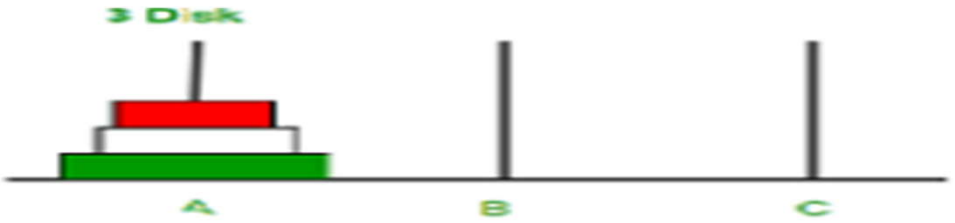
The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called a recursive function.

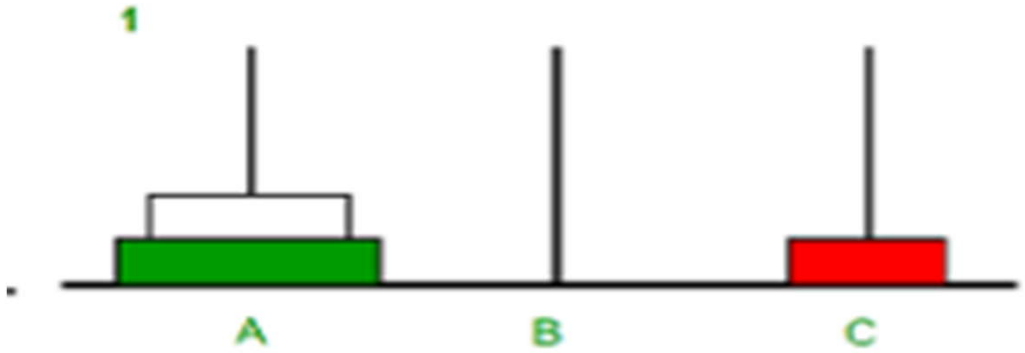
Using a recursive algorithm, certain problems can be solved quite easily.
Examples of such problems are
Towers of Hanoi (TOH)
Inorder/Preorder/Postorder Tree Traversals
DFS of Graph etc

Tower of Hanoi

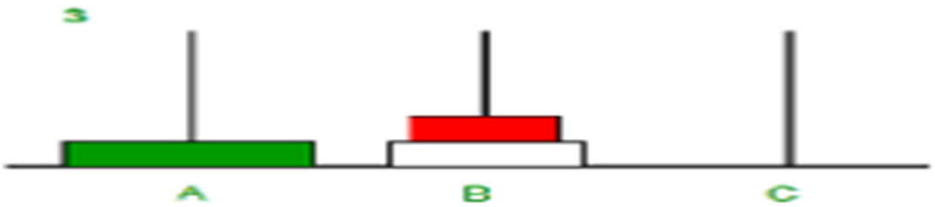
Tower of Hanoi is a mathematical puzzle where we have three rods and n disks. The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:

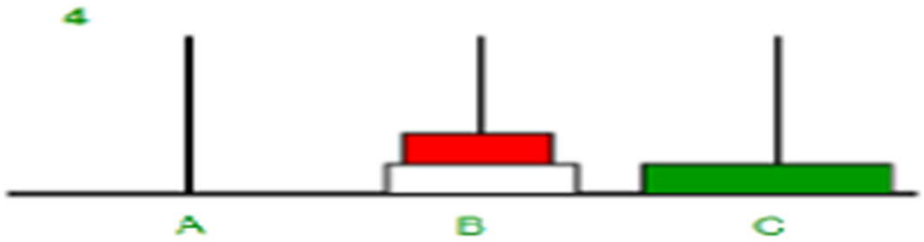
1. Only one disk can be moved at a time.
2. Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.
3. No disk may be placed on top of a smaller disk.

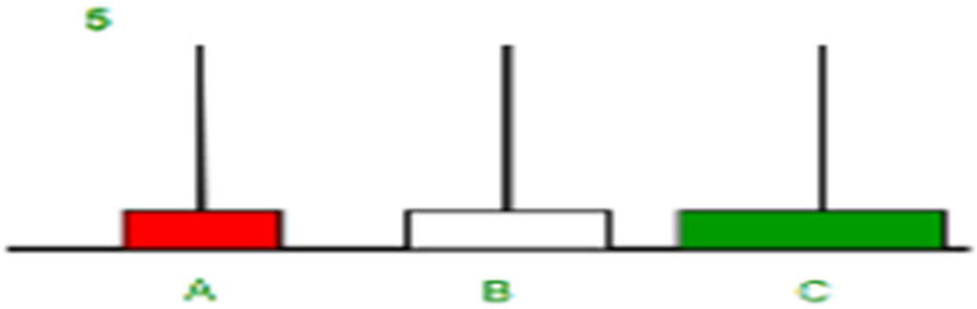

3 Disk
A
B
C


1
A
B
C


2
A
B
C


3
A
B
C


4
A
B
C


5
A
B
C


6
A
B
C


7
A
B
C

```
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,
8           "to top of tower", y);
9           TowersOfHanoi(n − 1, z, y, x);
10      }
11  }
```

Algorithm 1.3 Towers of Hanoi

Advantages of an Algorithm

- **Effective Communication:** Since it is written in a natural language like English, it becomes easy to understand the step-by-step delineation of a solution to any particular problem.
- **Easy Debugging:** A well-designed algorithm facilitates easy debugging to detect the logical errors that occurred inside the program.
- **Easy and Efficient Coding:** An algorithm is nothing but a blueprint of a program that helps develop a program.
- **Independent of Programming Language:** Since it is a language-independent, it can be easily coded by incorporating any high-level language.

Disadvantages of an Algorithm

- Developing algorithms for complex problems would be time-consuming and difficult to understand.
- It is a challenging task to understand complex logic through algorithms.

Definition [Space/Time complexity] The *space complexity* of an algorithm is the amount of memory it needs to run to completion. The *time complexity* of an algorithm is the amount of computer time it needs to run to completion. □

Performance evaluation can be loosely divided into two major phases: (1) a priori estimates and (2) a posteriori testing. We refer to these as *performance analysis* and *performance measurement* respectively.

Measurements

- Criteria
  - Is it correct?
  - Is it readable?
  - …
- Performance Analysis (machine independent)
  - space complexity: storage requirement
  - time complexity: computing time
- Performance Measurement (machine dependent)

Space Complexity
$S(P)=C+S_P(I)$

- Fixed Space Requirements (C)
  Independent of the characteristics of the inputs and outputs
  - instruction space
  - space for simple variables, fixed-size structured variable, constants
- Variable Space Requirements ($S_P(I)$)
  depend on the instance characteristic I
  - number, size, values of inputs and outputs associated with I
  - recursive stack space, formal parameters, local variables, return address

Example 1:

```
1    Algorithm abc(a, b, c)
2    {
3        return  a + b + b * c + (a + b − c)/(a + b) + 4.0;
4    }
```

Example 2:

```
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    }
```

Example 3:

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

Each call to RSum requires at least three words (including space for the values of n, the return address, and a pointer to $a[\,]$). Since the depth of recursion is $n + 1$, the recursion stack space needed is $\geq 3(n + 1)$. □

```
1    Algorithm abc(a, b, c)
2    {
3        return  a + b + b * c + (a + b − c)/(a + b) + 4.0;
4    }
```

```
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    }
```

$S_{\mathsf{Sum}}(n) \geq (\breve{n} + 3)$ ($\breve{n}$ for $a[\,]$, one each for n, i, and s).

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

Each call to RSum requires at least three words (including space for the values of n, the return address, and a pointer to $a[\,]$). Since the depth of recursion is $n + 1$, the recursion stack space needed is $\geq 3(n + 1)$. □

Time Complexity

$$T(P)=C+T_P(I)$$

- Compile time (C)
  independent of instance characteristics
- run (execution) time T_P
- Definition
  A *program step* is a syntactically or semantically meaningful program segment whose execution time is independent of the instance characteristics.
- Example

  S = a + b + b * c + (a + b - c) / (a + b) + 4.0

  S1 = a + b + c

Regard as the same unit
machine independent

Methods to compute the step count

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

Method 1: Count Method

Example 1 : Iterative summing of a list of numbers

```
float sum(float list[ ], int n)
{
   float tempsum : = 0;
   int i;
   for (i: = 0; i < n; i++)
{
      tempsum + = list[i];
}
       return tempsum;

}
```

2n + 3 steps

Method 1: Count Method

Example : Iterative summing of a list of numbers

```
float sum(float list[ ], int n)
{
    float tempsum = 0;                  count++; /* for assignment */
    int i;
    for (i = 0; i < n; i++) {
                                        count++;  /*for the for loop */
        tempsum += list[i];             count++;  /* for assignment */
    }
    count++;  /* last execution of for */
    return tempsum;
    count++;          /* for return */
}
```

2n + 3 steps

Example 2: Recursive summing of a list of numbers

```
float rsum(float list[ ], int n)
{
        if (n) {
            return rsum(list, n-1) + list[n-1];
        }
        return list[0];
}
```

2n+2 Steps

Example 2: Recursive summing of a list of numbers

```
float rsum(float list[ ], int n)
{
        count++; /*for if conditional */
    if (n) {
        count++;  /* for return and rsum invocation */
        return rsum(list, n-1) + list[n-1];
    }
    count++;
    return list[0];
}
```

2n+2 Steps

Matrix addition

```
void add( int a[ ] [MAX_SIZE], int b[ ] [MAX_SIZE],
                    int c [ ] [MAX_SIZE], int rows, int cols)
{
   int i, j;
   for (i = 0; i < rows; i++)
      for (j= 0; j < cols; j++)
        c[i][j] = a[i][j] +b[i][j];
 }
```

2rows * cols + 2 rows + 1

```
void add(int a[ ][MAX_SIZE], int b[ ][MAX_SIZE],
                    int c[ ][MAX_SIZE], int row, int cols )
{
  int i, j;
  for (i = 0; i < rows; i++){
      count++; /* for i for loop */
      for (j = 0; j < cols; j++) {
         count++; /* for j for loop */
         c[i][j] = a[i][j] + b[i][j];
         count++; /* for assignment statement */
      }
      count++;    /* last time of j for loop */
  }
  count++;        /* last time of i for loop */
}
```

2rows * cols + 2 rows + 1

Method 2:Tabular Method

Iterative function to sum a list of numbers

steps/execution

Statement	s/e	Frequency	Total steps
float sum(float list[ ], int n)	0	0	0
{	0	0	0
float tempsum = 0;	1	1	1
int i;	0	0	0
for(i=0; i <n; i++)	1	n+1	n+1
tempsum += list[i];	1	n	n
return tempsum;	1	1	1
}	0	0	0
Total			2n+3

Recursive Function to sum of a list of numbers

Statement	s/e	Frequency	Total steps
float rsum(float list[ ], int n)	0	0	0
{	0	0	0
if (n)	1	n+1	n+1
return rsum(list, n-1)+list[n-1];	1	n	n
return list[0];	1	1	1
}	0	0	0
Total			2n+2

Matrix Addition

Statement	s/e	Frequency	Total steps

Void add (int a[ ][MAX_SIZE], int b[ ][MAX_SIZE], int c[ ][MAX_SIZE], int rows, int cols)
{
int i, j;
for (i = 0; i < rows; i++)
for (j=0; j< cols; j++)
c[i][j] = a[i][j] + b[i][j];
}
	0
	0
	0
	1
	1
	1
	0
		0
		0
		0
		rows+1
		rows (cols+1)
		rows cols
		0
			0
			0
			0
			rows+1
			rows cols+rows
			rows cols
			0

Total			2rows cols+2rows+1

Mathematical Analysis of Non Recursive Algorithms

General Plan for Analyzing the Time Efficiency of Nonrecursive Algorithms

1. Decide on a parameter (or parameters) indicating an input's size.

2. Identify the algorithm's basic operation.

3. Check whether the number of times the basic operation is executed depends only on the size of an input. If it also depends on some additional property, the worst-case, average-case, and, if necessary, best-case efficiencies have to be investigated separately.

4. Set up a sum expressing the number of times the algorithm's basic operation is executed.

5. Using standard formulas and rules of sum manipulation, either find a closed form formula for the count or, at the very least, establish its order of growth.

Problem 1

Consider the problem of finding the value of the largest element

in a list of n numbers.

ALGORITHM MaxElement(A[0..n − 1])
//Determines the value of the largest element in a given array
//Input: An array A[0..n − 1] of real numbers
//Output: The value of the largest element in A
{
maxval ← A[0]
for i ← 1 to n − 1 do
if A[i] > maxval
maxval ← A[i]
return maxval
}

The obvious measure of an input's size here is the number of elements in the array, i.e., n.

The operations that are going to be executed most often are in the algorithm's for loop.

There are two operations in the loop's body: the comparison A[i] > maxval and the assignment maxval ← A[i].

Which of these two operations should we consider basic?

Since the comparison is executed on each repetition of the loop and the assignment is not, we should consider the comparison to be the algorithm's basic operation.

Let us denote C(n) the number of times this comparison is executed and try to find a formula expressing it as a function of size n.

The algorithm makes one comparison on each execution of the loop, which is repeated for each value of the loop's variable i within the bounds 1 and n − 1, inclusive.

Therefore, we get the following sum for C(n):

$$C(n) = \sum_{i=1}^{n-1} 1.$$

Problem 2

Consider the element uniqueness problem: check whether all the elements in a given array of n elements are distinct

ALGORITHM *UniqueElements*$(A[0..n-1])$

```
//Determines whether all the elements in a given array are distinct
//Input: An array A[0..n − 1]
//Output: Returns "true" if all the elements in A are distinct
//                and "false" otherwise
for i ← 0 to n − 2 do
    for j ← i + 1 to n − 1 do
        if A[i] = A[j] return false
return true
```

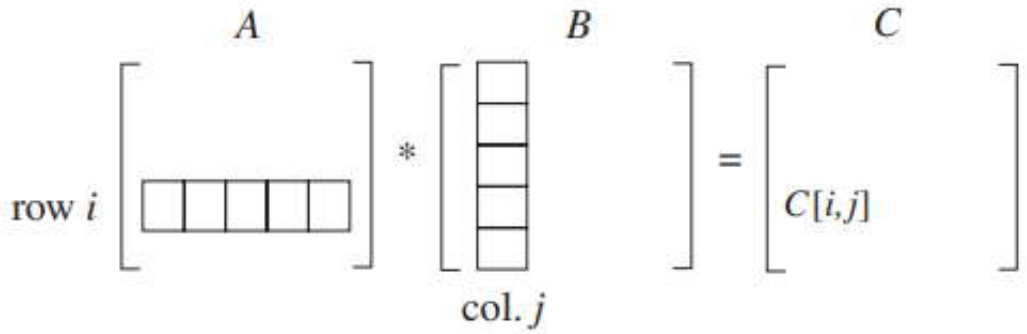
$$C_{worst}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i)$$

$$= \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i = (n-1) \sum_{i=0}^{n-2} 1 - \frac{(n-2)(n-1)}{2}$$

$$= (n-1)^2 - \frac{(n-2)(n-1)}{2} = \frac{(n-1)n}{2} \approx \frac{1}{2} n^2 \in \Theta(n^2).$$

Problem 3

- Given two n × n matrices A and B, find the time efficiency of the definition-based algorithm for computing their product C = AB. By definition, C is an n × n matrix whose elements are computed as the scalar (dot) products of the rows of matrix A and the columns of matrix B



where $C[i, j] = A[i, 0]B[0, j] + \cdots + A[i, k]B[k, j] + \cdots + A[i, n-1]B[n-1, j]$
for every pair of indices $0 \le i, j \le n-1$.

ALGORITHM *MatrixMultiplication*$(A[0..n-1, 0..n-1], B[0..n-1, 0..n-1])$

//Multiplies two square matrices of order n by the definition-based algorithm
//Input: Two $n \times n$ matrices A and B
//Output: Matrix $C = AB$
for $i \leftarrow 0$ **to** $n-1$ **do**
    for $j \leftarrow 0$ **to** $n-1$ **do**
        $C[i, j] \leftarrow 0.0$
        for $k \leftarrow 0$ **to** $n-1$ **do**
            $C[i, j] \leftarrow C[i, j] + A[i, k] * B[k, j]$
return C

Mathematical Analysis of Recursive Algorithms

General Plan for Analyzing the Time Efficiency of recursive Algorithms

1. Decide on a parameter (or parameters) indicating an input's size.

2. Identify the algorithm's basic operation.

3. Check whether the number of times the basic operation is executed can vary on different inputs of the same size; if it can, the worst-case, average-case, and best-case efficiencies must be investigated separately.

4. Set up a recurrence relation, with an appropriate initial condition, for the number of times the basic operation is executed.

5. Solve the recurrence or, at least, ascertain the order of growth of its solution.

Problem 1

Compute the factorial function F (n) = n! for an arbitrary nonnegative integer n.

ALGORITHM $F(n)$

//Computes $n!$ recursively
//Input: A nonnegative integer n
//Output: The value of $n!$
if $n = 0$ **return** 1
else return $F(n-1) * n$

$$M(n) = \underbrace{M(n-1)}_{\substack{\text{to compute} \\ F(n-1)}} + \underbrace{1}_{\substack{\text{to multiply} \\ F(n-1) \text{ by } n}} \quad \text{for } n > 0.$$

$$F(n) = F(n-1) \cdot n \quad \text{for every } n > 0,$$
$$F(0) = 1.$$

$$M(n) = M(n-1) + 1 \quad \text{for } n > 0,$$
$$M(0) = 0.$$

$$F(n) = F(n-1) \cdot n \quad \text{for } n > 0,$$

ALGORITHM *BinRec*(n)

//Input: A positive decimal integer n

//Output: The number of binary digits in n's binary representation

if $n = 1$ **return** 1

else return *BinRec*($\lfloor n/2 \rfloor$) + 1

$$A(n) = A(\lfloor n/2 \rfloor) + 1 \quad \text{for } n > 1.$$

$$A(1) = 0.$$

Asymptotic Notation

Asymptotic Notation is used to describe the running time of an algorithm - how much time an algorithm takes with a given input, n. There are three different notations: big O, big Theta (Θ), and big Omega (Ω). big-Θ is used when the running time is the same for all cases, big-O for the worst case running time, and big-Ω for the best case running time.

Types of Asymptotic Notations in Complexity Analysis of Algorithms

- The main idea of asymptotic analysis is to have a measure of the efficiency of algorithms that don't depend on machine-specific constants and don't require algorithms to be implemented and time taken by programs to be compared.

- Asymptotic notations are mathematical tools to represent the time complexity of algorithms for asymptotic analysis.

There are mainly three asymptotic notations:

1. *Big-O Notation (O-notation)*
2. *Omega Notation (Ω-notation)*
3. *Theta Notation (Θ-notation)*

Big-O Notation (O-notation):

Big-O notation represents the upper bound of the running time of an algorithm. Therefore, it gives the worst-case complexity of an algorithm.

.It is the most widely used notation for Asymptotic analysis.

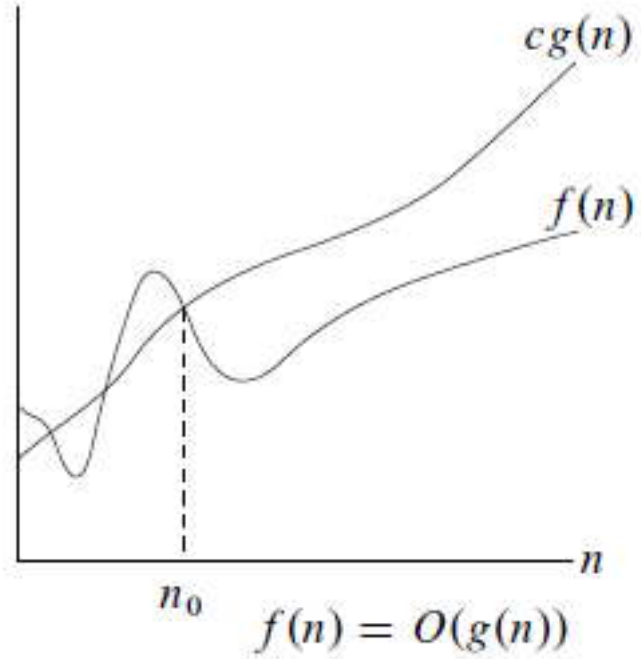
.It specifies the upper bound of a function.

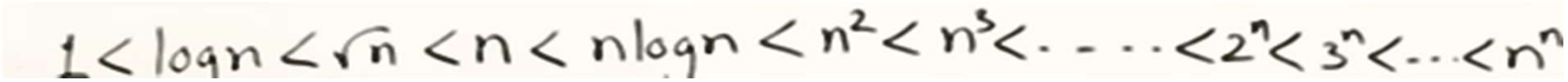
.The maximum time required by an algorithm or the worst-case time complexity.

.It returns the highest possible output value(big-O) for a given input.

.Big-Oh(Worst Case) It is defined as the condition that allows an algorithm to complete statement execution in the longest amount of time possible.

Definition [Big "oh"] 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 n_0 such that $f(n) \leq c * g(n)$ for all n, $n \geq n_0$. □





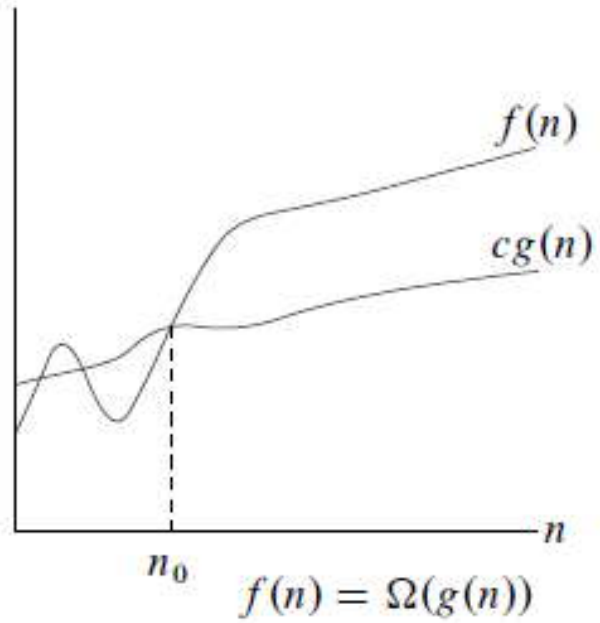
Asymptotic Notation (O)

- Examples
  1. $3n+2=O(n)$ /* $3n+2 \leq 4n$ for $n \geq 2$ */
  2. $3n+3=O(n)$ /* $3n+3 \leq 4n$ for $n \geq 3$ */
  3. $100n+6=O(n)$ /* $100n+6 \leq 101n$ for $n \geq 10$ */
  4. $10n^2+4n+2=O(n^2)$ /* $10n^2+4n+2 \leq 11n^2$ for $n \geq 5$ */
  5. $6*2^n+n^2=O(2^n)$ /* $6*2^n+n^2 \leq 7*2^n$ for $n \geq 4$ */

Omega Notation (Ω-Notation):

Omega notation represents the lower bound of the running time of an algorithm. Thus, it provides the best case complexity of an algorithm.

Definition [Omega] The function $f(n) = \Omega(g(n))$ (read as "f of n is omega of g of n") iff there exist positive constants c and n_0 such that $f(n) \geq c * g(n)$ for all n, $n \geq n_0$. □



1. $3n+2=\Omega(n)$ /* $3n+2>=3n$ for $n\geq 1$ */

2. $100n+6=\Omega(n)$ /* $100n+6>=100n$ for $n\geq 1$ */

3. $10n^2+4n+2=\Omega(n^2)$ /* $10n^2+4n+2>=n^2$ for $n\geq 1$ */

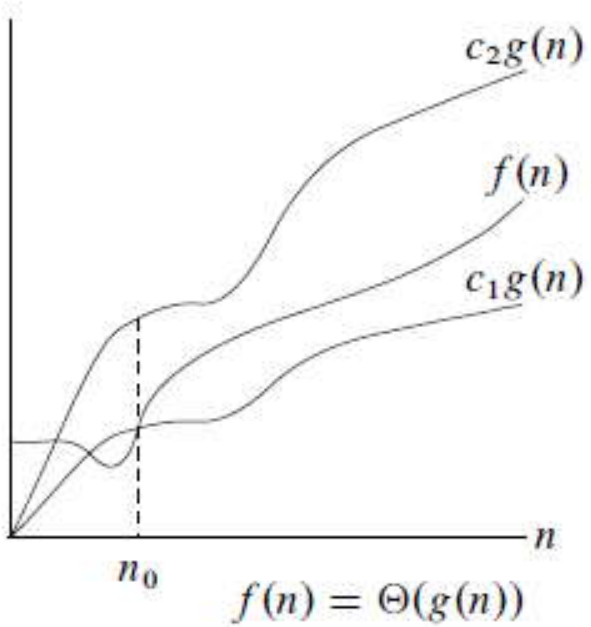
4. $6*2^n+n^2=\Omega(2^n)$ /* $6*2^n+n^2>=2^n$ for $n\geq 1$ */

Theta Notation (Θ-Notation):

Theta notation encloses the function from above and below. Since it represents the upper and the lower bound of the running time of an algorithm, it is used for analyzing the ***average-case*** *complexity of an algorithm.*

.Theta (Average Case) You add the running times for each possible input combination and take the average in the average case.

Definition [Theta] The function $f(n) = \Theta(g(n))$ (read as "f of n is theta of g of n") iff 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 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 $n_0 = 2$. $3n + 3 = \Theta(n)$,

Definition [Little "oh"] The function $f(n) = o(g(n))$ (read as "f of n is little oh of g of n") iff

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

Definition [Little omega] The function $f(n) = \omega(g(n))$ (read as "f of n is little omega of g of n") iff

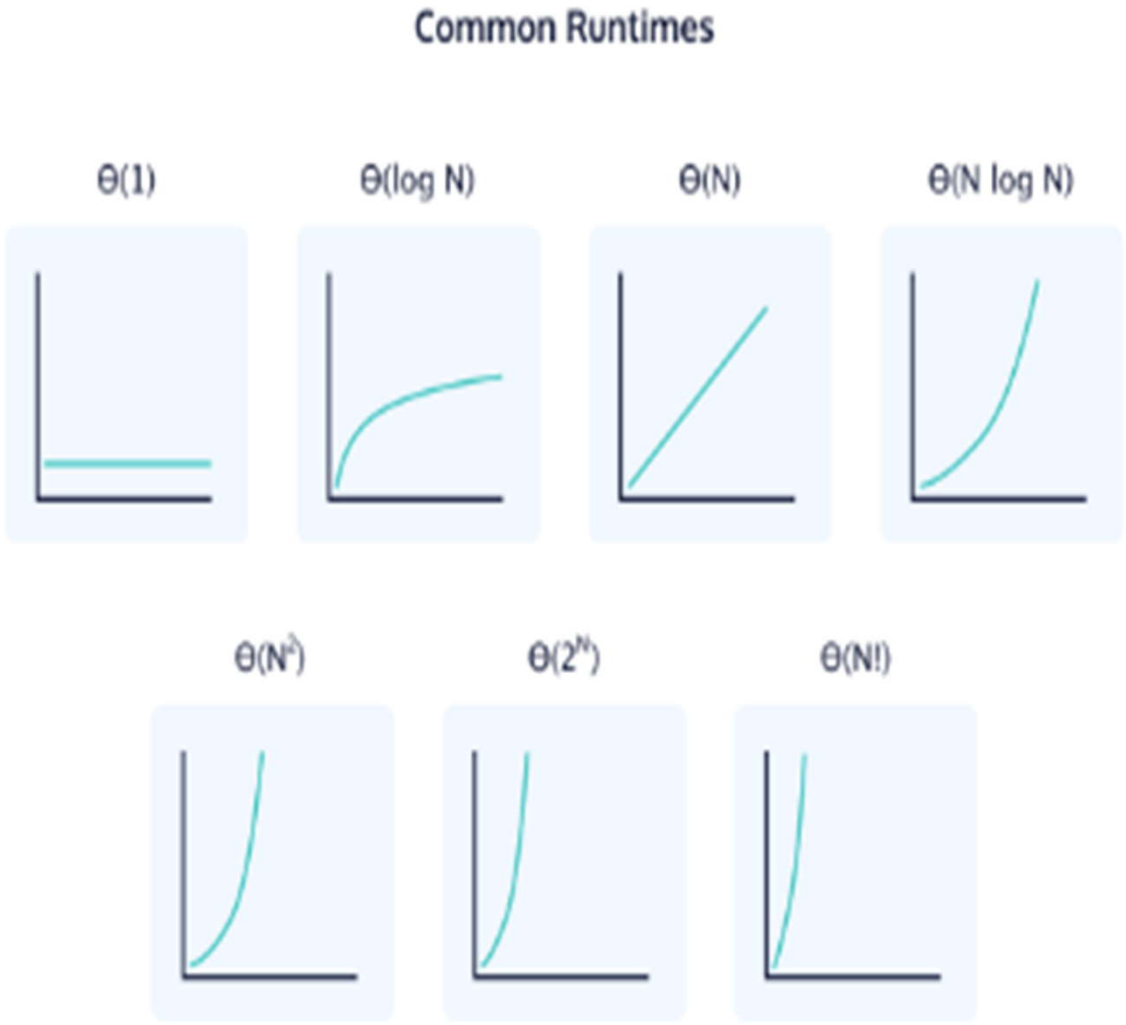
$$\lim_{n \to \infty} \frac{g(n)}{f(n)} = 0$$

Algorithmic Common Runtimes

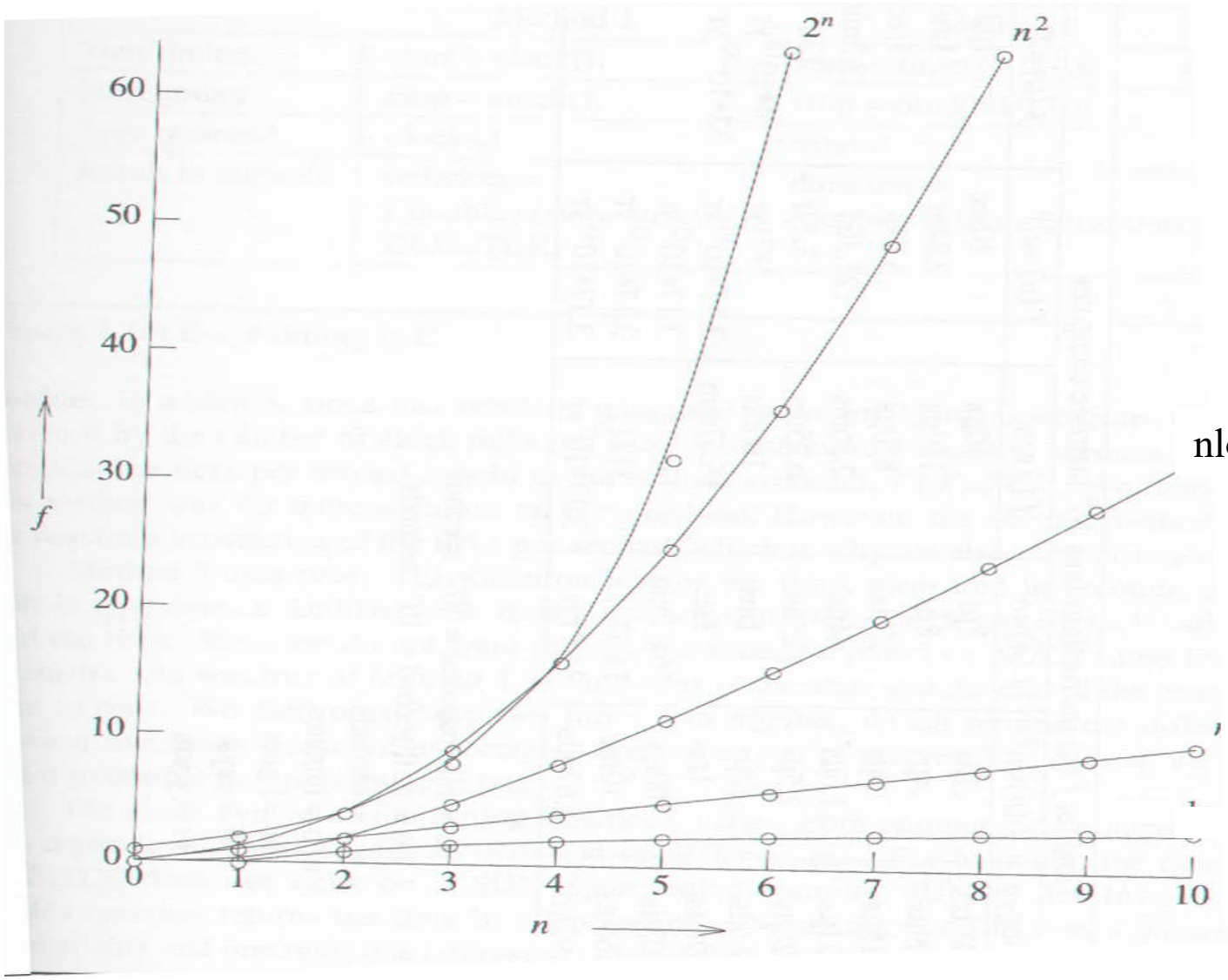
The common algorithmic runtimes from fastest to slowest are:

- constant: Θ(1)
- logarithmic: Θ(log N)
- linear: Θ(N)
- polynomial: Θ(N^2)
- exponential: Θ(2^N)
- factorial: Θ(N!)

$1 < \log n < \sqrt{n} < n < n\log n < n^2 < n^3 < \cdots < 2^n < 3^n < \cdots < n^n$


Common Runtimes
Θ(1)
Θ(log N)
Θ(N)
Θ(N log N)
Θ(N²)
Θ(2^N)
Θ(N!)

- O(1): constant
- O(n): linear
- $O(n^2)$: quadratic
- $O(n^3)$: cubic
- $O(2^n)$: exponential
- O(logn)
- O(nlogn)


2^n
n^2
60
50
40
30
20
10
0
f
0
1
2
3
4
5
6
7
8
9
10
n
nlogn
n
logn