Design and Analysis of Algorithms

UNIT-1

Introduction

Topics

- **Algorithm Specification**
- **Pseudocode Convention**
- **Types of Algorithms**
- **Performance Analysis**
- **Asymptotic Notations**

What is an Algorithm

An algorithm is a set of rules.

An algorithm is a step-by-step procedure

An algorithm is a sequence of computational steps

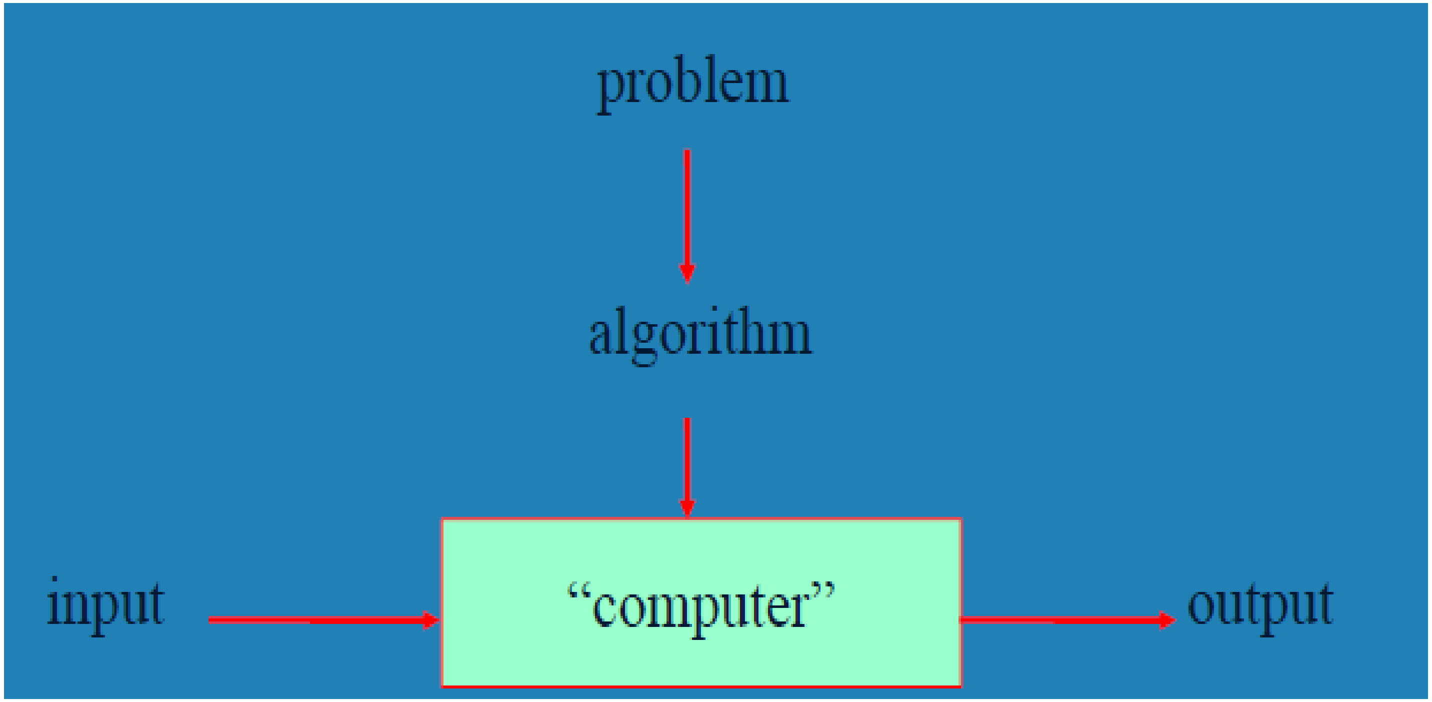
An algorithm is a sequence of operations performed data.

An algorithm is an abstraction of a program

Algorithm refers to a method that can be used by a computer for the solution of a problem

What is an Algorithm

- Definition
  - An ***Algorithm*** is a finite set of instructions that, if followed, accomplishes a particular task.



Characteristics of an Algorithm

All algorithms must satisfy the following criteria:

(1)***Input***. There are zero or more quantities that are externally supplied.

(2)***Output***. At least one quantity is produced.

(3)***Definiteness***. Each instruction is clear and unambiguous.

(4)***Finiteness***. If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.

(5)***Effectiveness***. Every instruction must be basic enough to be carried out, in principle, by a person using only pencil and paper. It is not enough that each operation be definite and also must be feasible.

Study of Algorithms

- The study of algorithms includes many important and active areas of research.

- **Four distinct areas of study:**

  - 1. How to devise algorithm

  - 2. How to validate algorithm

  - 3. How to analyze algorithms

  - 4. How to test a program

Devise and Validate an Algorithm

- **Devise an Algorithm:**
  - Study various design techniques that have proven to be useful in that they have often yielded good algorithms.
  - Various design Strategies:
    - Divide and Conquer
    - Greedy Method
    - Dynamic Programming
    - Back Tracking
    - Branch & Bound

- **Validate an Algorithm:**
  - Once an algorithm is devised, it is necessary to show that it computes the correct answer for all possible legal inputs.
  - Once the validity of the method has been shown, a program can be written and a second phase begins.

Analyze and Test the Algorithms

- **3. Analyze the Algorithm**
  – Analysis of algorithms or performance analysis refers to the task of determining how much computing time and storage an algorithm requires.

- **4. Test a Program**
  – Testing a program consists of two phases: **debugging and profiling** (or performance measurement).
  – Debugging is the process of executing programs on sample data sets to determine whether faulty results occur and, if so, to correct them.
  – A proof of correctness is much more valuable than a thousand tests(if that proof is correct),since it guarantees that the program will work correctly for all possible inputs.
  – Profiling or performance measurement is the process of executing a correct program on datasets and measuring the time and space it takes to compute the result.

Topics

- **Algorithm Specification**
- **Pseudocode Convention**
- **Types of Algorithms**
- **Performance Analysis**
- **Asymptotic Notations**

Describing Algorithms

- **Natural language**
  - English
    - Instructions must be definite and effectiveness
- **Graphic representation**
  - Flowchart
    - work well only if the algorithm is small and simple
- **Pseudocode**
  - Readable
  - Instructions must be definite and effectiveness

Pseudocode

- **Pseudocode:**
  - Implementation of an algorithm in the form of annotations and informative text written in plain English.
  - It has no syntax like any of the programming language and thus can't be compiled or interpreted by the computer.
- **Advantages of Pseudocode**
  - Improves the readability of any approach.
  - Acts as a bridge between the program and the process. Also works as a rough documentation, so the logic of one developer can be understood easily when a pseudo code is written out.
  - Explains what exactly each line of a program should do, hence making the code construction phase easier for the programmer.
- **Disadvantages of Pseudocode**
  - Pseudocode does not provide a visual representation of the logic of programming.
  - There are no proper format for writing the for pseudocode.

Pseudocode Conventions

- 1. Comments begin with // and continue until the end of line.
- 2. Block of statements are indicated with matching braces:{ and }.
- 3. Statements are delimited by “;”.
- 4. The data types of variables are not explicitly declared.
Compound data types can be formed with records.

```
node = record
        {
            data type_1 data;
              .
              .
              .
            data type_n data;
            node *link;
        }
```

Pseudocode Conventions

- **5.** Assignment of values to variables is done using the assignment statement
  variable:=expression or value;

- **6.** There are two Boolean values true and false. In order to produce these values, the logical operators and, or, and not and the relational operators <, and > are provided.

- **7.** 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].

- **8.** The while, repeat-until and for loops takes the following form:

```
while (condition)do
 {
statement 1    ...
statement n
}
```

```
Repeat
 {
Statement 1 ....
Statement n
} until(condition)
```

```
for variable:= value l to value 2 step value do
{
     (statement 1) ....
     (statement n)
}
```

Pseudocode Conventions

- **9.** A conditional statement has the following forms:
  - if (condition) then statement;
  - if (condition) then statement 1 else statement 2;
  - case

```
{
: condition 1: statement 1 .....
: condition n: statement n
: else: statement n + 1
}
```

- **10.** Input and output are done using the instructions read and write.

- **11.** 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)

Tasks on Algorithms

- Write an algorithm to merge the given two sorted arrays as one sorted array.
- Write an algorithm for printing nth Fibonacci number.
- Write an algorithm to find the maximum product of two integers in the given array.

Topics

- **Algorithm Specification**
- **Pseudocode Convention**
- **Types of Algorithms**
- **Performance Analysis**
- **Asymptotic Notations**

Types of Algorithms

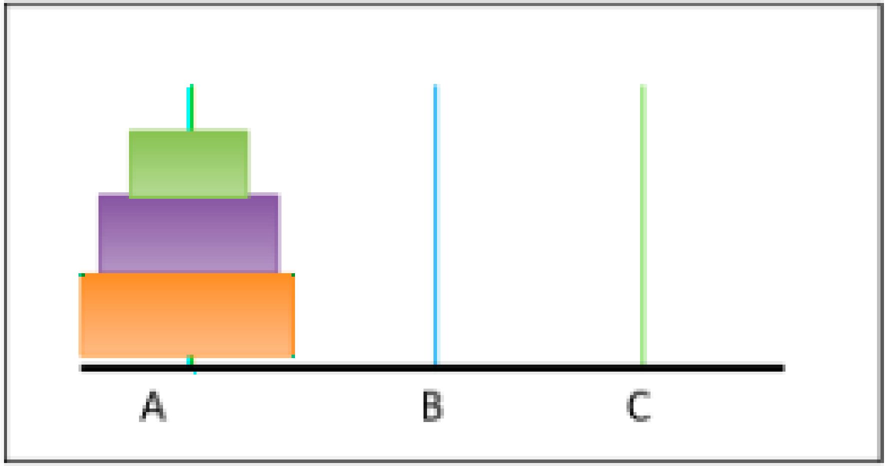
- **Two Types of Algorithms**:
  - **Iterative Algorithms**
    - The iteration is when a loop repeatedly executes until the controlling condition becomes false.
  - **Recursive Algorithms**
    - An algorithm is said to be recursive if the same algorithm is invoked in the body.
    - An algorithm that calls itself is direct recursive.
    - Algorithm A is said to be indirect recursive if it calls another algorithm which in turn calls A.

- The primary difference between recursion and iteration is that recursion is a process, always applied as a function and iteration is applied to the set of instructions which we want to get repeatedly executed.

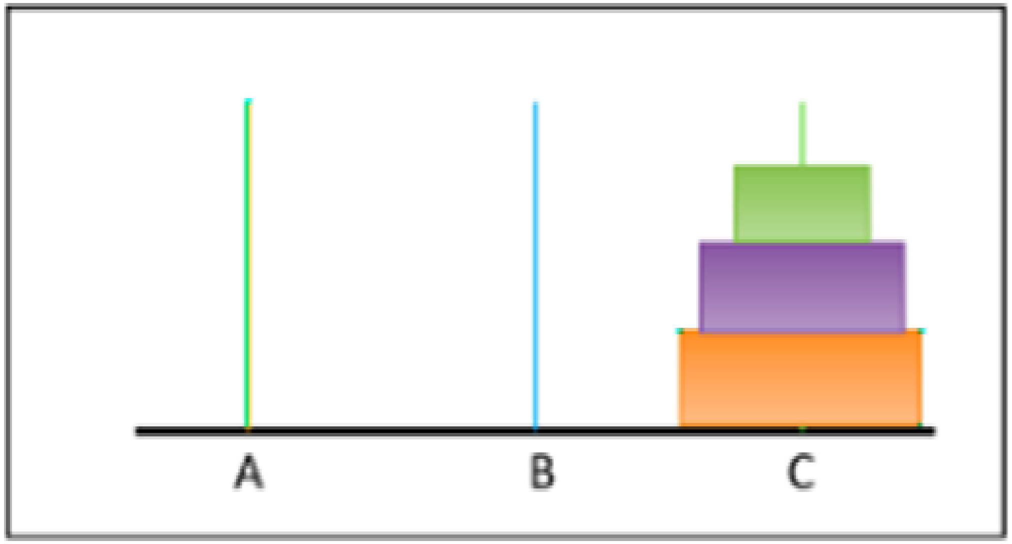
Example for Recursive Algorithms

- Towers of Hanoi Problem

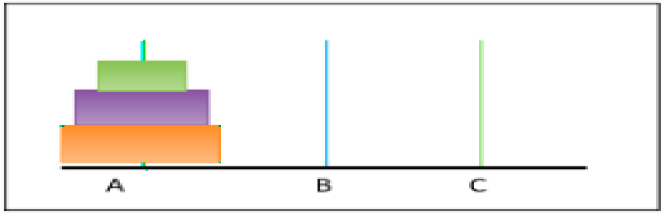
Example for Recursive Algorithms



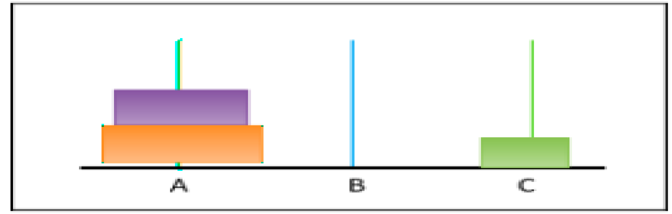
Initial State of the problem



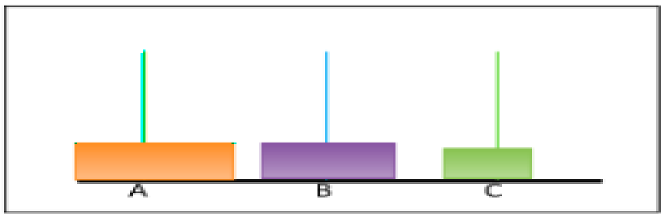
Example for Recursive Algorithms



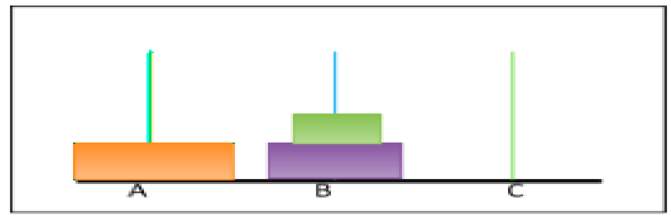
Initial State of the problem



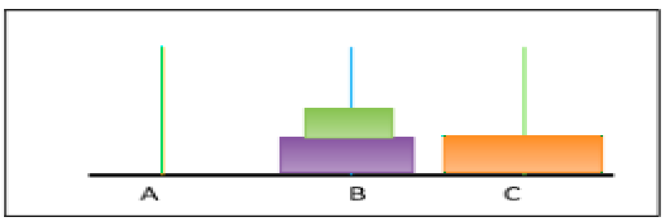
Step 1: Move The topmost disk to pole C



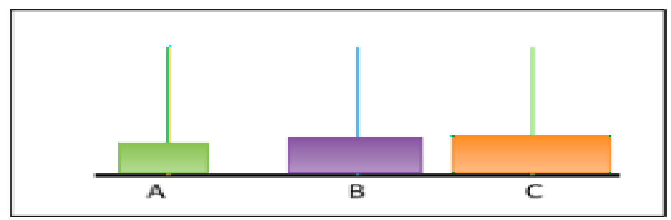
Step 2: Move the second disk to Pole B



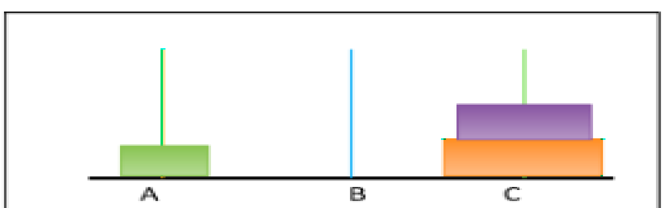
Step 3: Move the disk from pole C to pole B

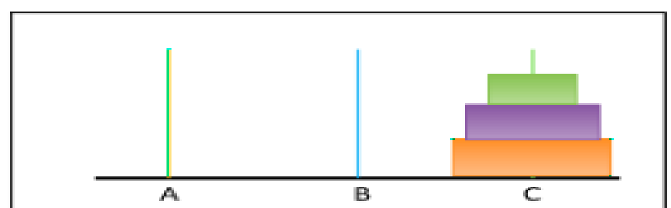


Step 4: Move the largest disk from pole A to pole C

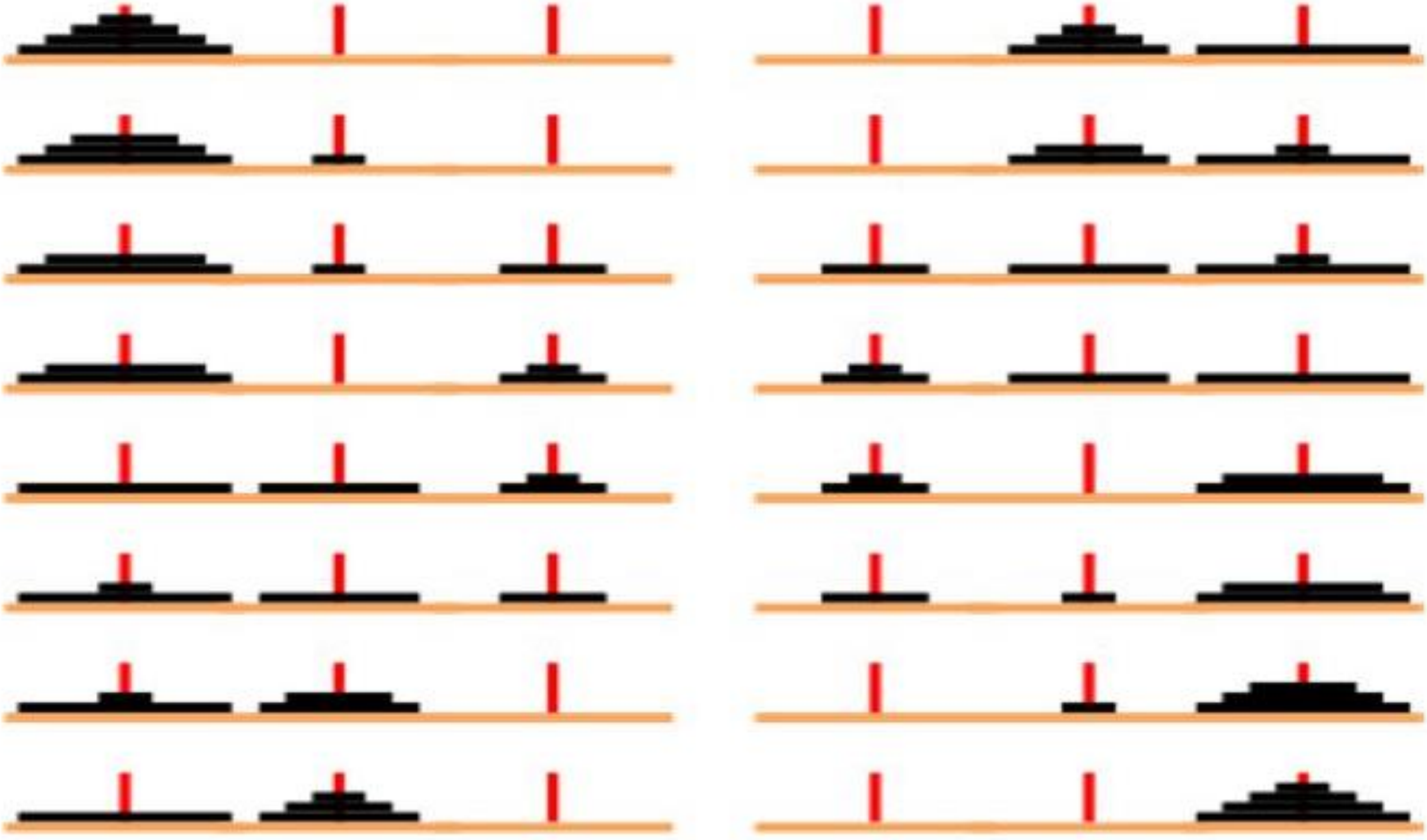


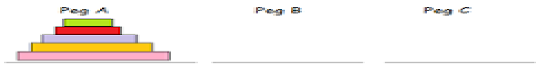
Step 5: Move the top disk from pole C to A





Example for Recursive Algorithms





Initially

Initially, all the disks are on *peg A*. *Peg B* and *peg C*.


Step 1

Step 1-> Transfer Smallest disk to *peg C*.


Step 2

Step 2-> Transfer second smallest disk to *peg B*.


Step 3

Step 3-> Transfer smallest disk on *peg B*, on top of second smallest disk.


Step 4

Step 4-> Transfer third largest disk on *peg C*.


Step 5

Step 5-> Transfer smallest disk from *peg B* to *peg A*.


Step 6

Step 6->Transfer third largest disk from *peg B* to *peg C*.


Step 7

Step 7-> Transfer smallest disk from *peg A* to *peg C*.


Step 8

Step 8<- Transfer second largest disk from *peg A* to *peg C*.


Step 9

Step 9-> Transfer smallest disk from *peg C* to *peg B*.


Step 10

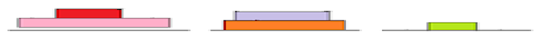
Step 10-> Transfer third largest disk from *peg C* to *peg A*.


Step 11

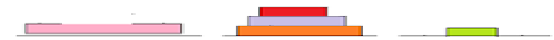
Step 11-> Transfer smallest disk from *peg B* to *peg A*.


Step 12

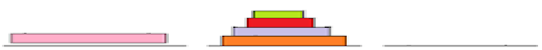
Step 12-> Transfer third largest disk from *peg C* to *peg B*.


Step 13

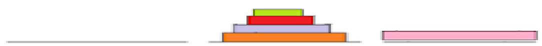
Step 13-> Transfer smallest disk from *peg A* to *peg C*.


Step 14

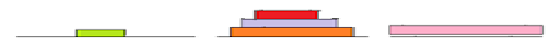
Step 14-> Transfer second smallest disk from *peg A* to *peg B*.


Step 15

Step 15-> Transfer smallest disk from *peg C* to *peg B*.


Step 16

Step 16-> Transfer largest disk from *peg A* to *peg C*.


Step 17

Step 17-> Transfer smallest disk from *peg B* to *peg A*.


Step 18

Step 18-> Transfer second smallest disk from *peg B* to *peg C*.


Step 19

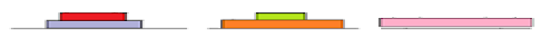
Step 19-> Transfer smallest disk from *peg A* to *peg C*.


Step 20

Step 20-> Transfer third largest disk from *peg B* to *peg A*.


Step 21

Step 21-> Transfer smallest disk from *peg C* to *peg B*.


Step 22

Step 22-> Transfer third largest disk from *peg C* to *peg A*.


Step 23

Step 23-> Transfer smallest disk from *peg B* to *peg A*.


Step 24

Step 24-> Transfer second largest disk from *peg B* to *peg C*.


Step 25

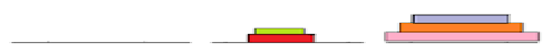
Step 25-> Transfer smallest disk from *peg A* to *peg C*.


Step 26

Step 26-> Transfer second smallest disk from *peg A* to *peg B*.


Step 27

Step 27-> Transfer smallest disk from *peg C* to *peg B*.


Step 28

Step 28-> Transfer third largest disk from *peg A* to *peg C*.


Step 29

Step 29-> Transfer smallest disk from *peg B* to *peg A*.

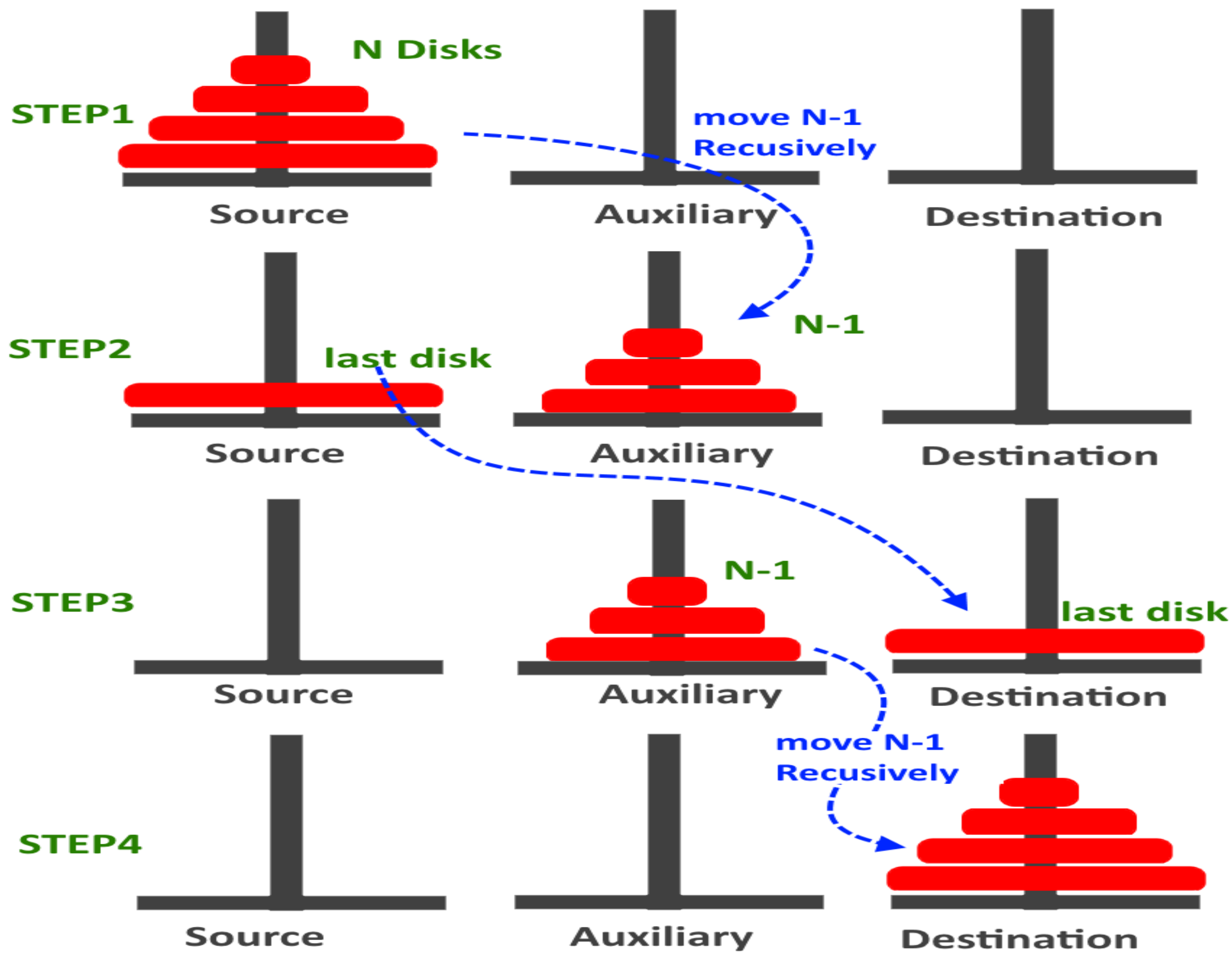

Step 30

Step 30-> Transfer second smallest disk from *peg B* to *peg C*.


Step 31

Step 31-> Final step. All the disks are on the *peg C*.

Example for Recursive Algorithms



Example for Recursive Algorithms

```
Algorithm Hanoi(disk, source, dest, aux)
{
        IF (disk == 1), THEN
                move disk from source to dest;
        ELSE
        {
                Hanoi(disk - 1, source, aux, dest);         // Step 1
                Write 'move disk n from source to dest'; // Step 2
                Hanoi(disk - 1, aux, dest, source);         // Step 3
        }
}
```

Topics

- **Algorithm Specification**
- **Pseudocode Convention**
- **Types of Algorithms**
- **Performance Analysis**
- **Asymptotic Notations**

Performance Evaluation

- Evaluate a program
  - *MWGWRERE*

    Meet specifications, **W**ork correctly,

    Good user-interface, **W**ell-documentation,

    Readable, **E**ffectively use functions,

    Running time acceptable,

    Efficiently use space

- How to achieve them?
  - Good programming style, experience, and practice

Performance Evaluation

- **Performance Evaluation**
  - Performance **Analysis**
  - Performance **Measurement**

- **Performance Analysis** - prior
  - an important branch of CS, ***complexity theory***
  - estimate ***time*** – ***Time Complexity***
  - estimate ***space*** – ***Space Complexity***
  - machine independent

- **Performance Measurement** -posterior
  - The actual ***time*** and ***space*** requirements
  - machine dependent

Space Complexity

- Definition
  - The ***space complexity*** of a program is the amount of memory that it needs to run to completion
- The space needed is the sum of
  - ***Fixed*** space and ***Variable*** space

- Fixed space
  - Includes the instructions, variables, and constants
  - Independent of the number and size of Input and Output
- Variable space
  - Depends on an instance '*I*' of the problem
  - Includes dynamic allocation, functions' recursion
- Total space of any program
  - $S(P) = c + \mathbf{S_p(Instance)}$

Examples of Evaluating Space Complexity

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

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

$S_{sum(n)} = 0$

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

$S_{rsum}(n) = 3*n$

parameter: float(list[])	1
parameter: integer(n)	1
return address	1

Time Complexity

- **Definition**

  The ***time complexity, T(p),*** taken by a program P is the sum of the compile time and the run time

  T(P)= compile time + **run** (or **execution**) **time**

  $= \boldsymbol{c} + \boldsymbol{t}_p(n)$

  *Compile time does not depend on the instance characteristics

- **How to evaluate?**
  – Use the system clock (machine dependent)
  – Number of ***steps*** performed (machine-independent)

- **Definition of a program step**
  – A ***program step*** is a syntactically or semantically meaningful instruction whose execution time is independent of the instance characteristics.

Examples of Determining Steps

- ***the first method: count increment by a step***
- ***EX: Algorithm for calculating sum of numbers in an array***

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

2n+ 3

Examples of Determining Steps(Cont.)

```
void add(int a[][], int b[][], int c[][], int R, int C)
{
    int i, j;
    for i=0 to R
    {
        count++;                    /* for the for i loop */
        for j=0 to C
        {
            count++;                /* for the for j loop */
            count++;                /* for the addition */
            c[i,j]= a[i,j] + b[i,j];
        }
        count++;                    /* last execution of for j */
    }
    count++;                        /* last execution of for i */
}
```

y

x

= (R* x)+1

= (R*(2+y))+1

= (R*(2+(2*C)))+1

= (2*R*C+2*R)+1

Examples of Determining Steps(Cont.)

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

$$
\begin{aligned}
trsum(1) &= 2 \\
trsum(n) &= 2 + trsum(n-1) \\
&= 2 + (2 + trsum(n-2)) \\
&= 2*2 + trsum(n-2) \\
&= 2*2+(2+ trsum(n-3)) \\
&= 2*3+ trsum(n-3) \\
&= \ldots\ldots\ldots\ldots\ldots \\
&= 2*(n-1) + trsum(n-(n-1)) \\
&= 2*n-2+2 \\
&= 2*n
\end{aligned}
$$

Examples of Determining Steps(Cont.)

- The second method: build a table to count the number of steps

s/e: steps per execution

frequency: total numbers of times each statements is executed

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

Examples of Determining Steps(Cont.)

- The second method: build a table to count

  s/e: steps per execution

  frequency: total numbers of times each statements is executed

Statement	*s/e*	*Frequency*	*Total Steps*
void add(int a[][], . . .	*0*	*0*	*0*
{	*0*	*0*	*0*
int i, j;	*0*	*0*	*0*
for (i=0; i< R; i++)	*1*	*R+ 1*	*R+ 1*
for (j=0; j< C; j++)	*1*	*R*(C+1)*	*R*C+ R*
c[i][j]= a[i][j] + b[i][j];	*1*	*R*C*	*R*C*
}	*0*	*0*	*0*
Total			***2*R*C+2*R+1***

Examples of Determining Steps(Cont.)

- The second method: build a table to count

s/e: steps per execution

frequency: total numbers of times each statements is executed

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			***2*n + 2***

Tasks on Performance Analysis

- Write an algorithm for matrix multiplication and calculate the its time complexity.
- Write an algorithm to print ‘n’ numbers in the Fibonacci series and estimate its time complexity.
- Write an algorithm to find the largest element in an array and estimate the time complexity.
- Write an algorithm to evaluate a polynomial using Horner’s rule and estimate the time complexity.
- Write an algorithm to check whether the given number is Armstrong Number or not and estimate the time complexity.
- Estimate the time complexity of factorial of a number using recursion.

Topics

- **Algorithm Specification**
- **Pseudocode Convention**
- **Types of Algorithms**
- **Performance Analysis**
- **Asymptotic Notations**

Algorithm Analysis

- To analyze the given algorithm, we need to know with which inputs the algorithm takes less time and with which inputs the algorithm takes a long time.
- There are three types of analysis:
- **Worst case Analysis**
  - Defines the input for which the algorithm takes a long time (slowest time to complete).
- **Best case Analysis**
  - Defines the input for which the algorithm takes the least time (fastest time to complete).
- **Average case Analysis**
  - Assumes that the input is random.
  - Run the algorithm many times, using many different inputs.
  - compute the total running time (by adding the individual times), and divide by the number of times the algorithm has excuted.

Asymptotic Notations

- Asymptotic notations are the mathematical notations used to describe the running time of an algorithm when the input tends towards a particular value or a limiting value (say 'n').

- The simplest example is a function f (n) = n^2+3n,
  - the term 3n becomes insignificant compared to n^2 when n is very large.
  - The function "f (n) is said to be asymptotically equivalent to n^2 as n $\rightarrow \infty$", and here is written symbolically as f (n) ~ n^2.

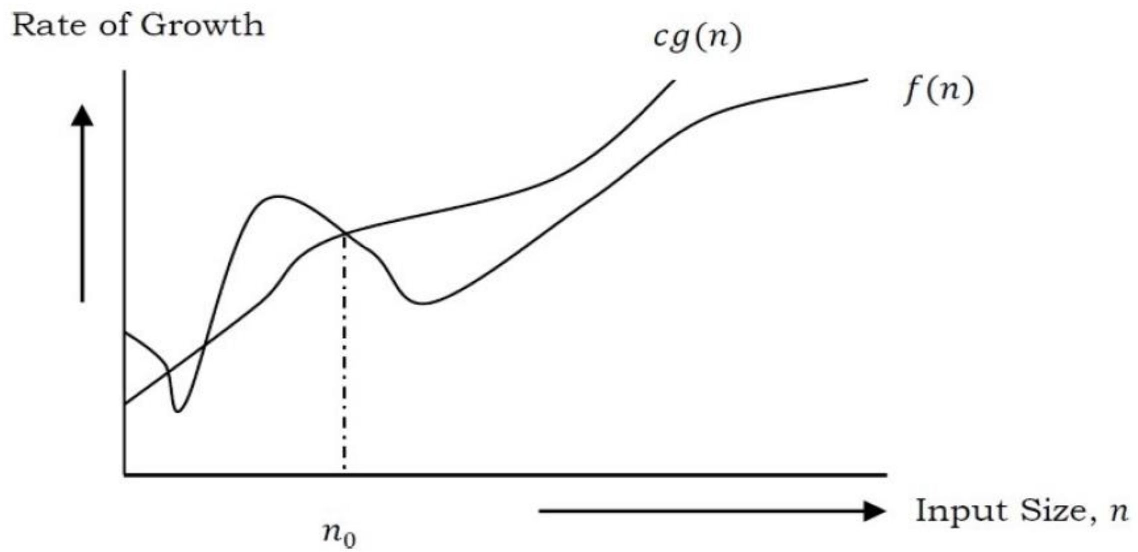
Asymptotic Notations

- The commonly used asymptotic notations to represent the time complexity of an algorithm:
  - **O (Big-Oh) Notation**
  - **Ω (Omega) Notation**
  - **θ (Theta) Notation**
  - o (Little-oh) Notation
  - ω (Little-omega) Notation

O (Big-Oh) Notation-Upper Bounding function

- It represents the upper bound running time complexity of an algorithm.
- It is the measure of the longest amount of time.
- **Definition:** The function **f (n) = O (g (n))** [read as "f of n is big-oh of g of n"] if and only if there exists positive constant **c** and **n0** such that

$$f(n) \leqslant c*(g(n)) \text{ for } \forall n >= n0$$



Examples on O (Big-Oh) Notation

- **Example-1:** Find upper bound for $f(n) = 3n + 8$

- **Example-2:** Find upper bound for $f(n) = n^2 + 1$

- **Example-3:** Find upper bound for $f(n) = n^4 + 100n^2 + 50$
- **Example-4:** Find upper bound for $f(n) = 2n^3 - 2n^2$
- **Example-5:** Find upper bound for $f(n) = n$
- **Example-6:** Find upper bound for $f(n) = 410$

O (Big-Oh) Notation

Theorem 1.2 If $f(n) = a_m n^m + \cdots + a_1 n + a_0$, then $f(n) = O(n^m)$.

Proof:

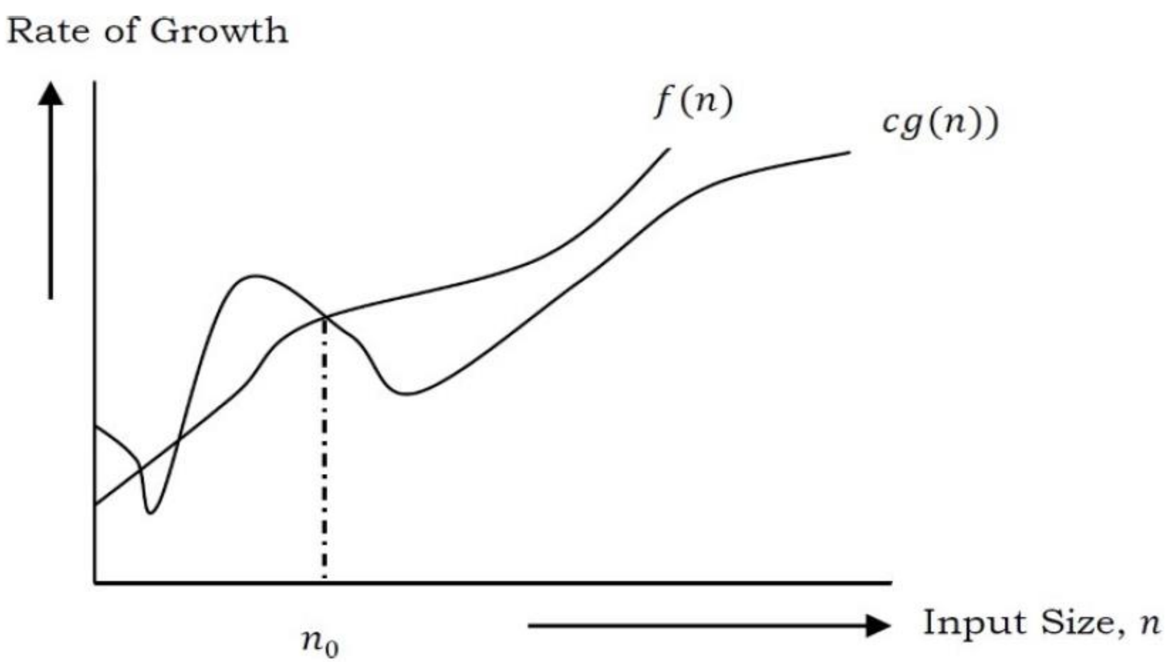
$$
\begin{aligned}
f(n) &\le \textstyle\sum_{i=0}^{m} |a_i| n^i \\
&\le n^m \textstyle\sum_{i=0}^{m} |a_i| n^{i-m} \\
&\le n^m \textstyle\sum_{i=0}^{m} |a_i| \qquad \text{for } n \ge 1
\end{aligned}
$$

So, $f(n) = O(n^m)$ (assuming that m is fixed).

Ω (Omega) Notation-Lower Bounding function

- The notation Ω(n) is the formal way to express the lower bound of an algorithm's running time.
- **Definition:** The function f (n) = Ω (g (n)) [read as "f of n is omega of g of n"] if and only if there exists positive constant c and n_0 such that

$$F(n) \geq c^* g(n) \text{ for all } n, n \geq n_0$$



Examples on Ω (Omega) Notation

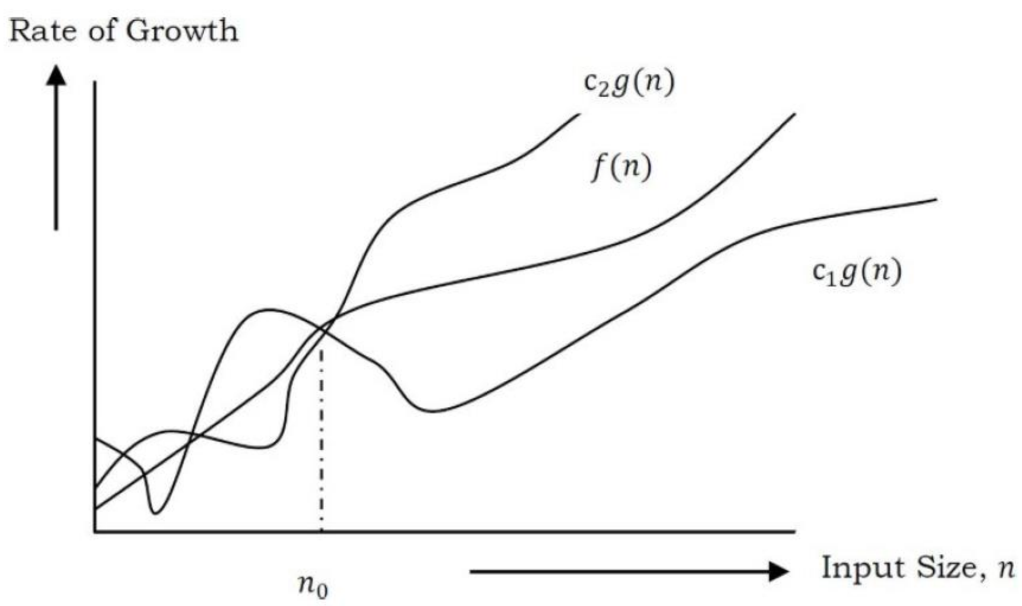
- **Example-1:** Find lower bound for $f(n) = 5n^2$.

- **Example-2:** Find lower bound for $f(n) = 10n^2 + 4n + 2$
- **Example-3:** Find lower bound for $f(n) = 6 * 2^n + n^2$
- **Example-4:** Prove $f(n) = 100n + 5 \neq \Omega(n^2)$.
- **Example-5:** Prove that $2n = \Omega(n)$, $n^3 = \Omega(n^3)$, $n^3 = O(logn)$.

θ (Theta) Notation

- The notation θ(n) is the formal way to express both the lower bound and the upper bound of an algorithm's running time.
- **Definition:** The function f (n) = θ (g (n)) [read as "f is the theta of g of n"] if and only if there exists positive constant c_1, c_2 and n_0 such that

$$c_1 * g(n) \le f(n) \le c_2 * g(n) \text{ for all } n,\ n \ge n_0$$



Examples on θ (Theta) Notation

- Prove that $3n+2 = \theta(n)$

- Prove that $10n^2+4n+2 = \theta(n^2)$
- Prove that $n^3 + 106n^2 = \theta(n^3)$
- Prove that the following is incorrect:

$$n^2/\log n = \theta(n^2)$$

Little oh and Little omega Notation

Definition 1.7 [Little "oh"] The function $f(n) = o(g(n))$ (read as "f of n is little oh of g of n") iff

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

Example: The function 3n +2 = o(n^2)

Definition 1.8 [Little omega] The function $f(n) = \omega(g(n))$ (read as "f of n is little omega of g of n") iff

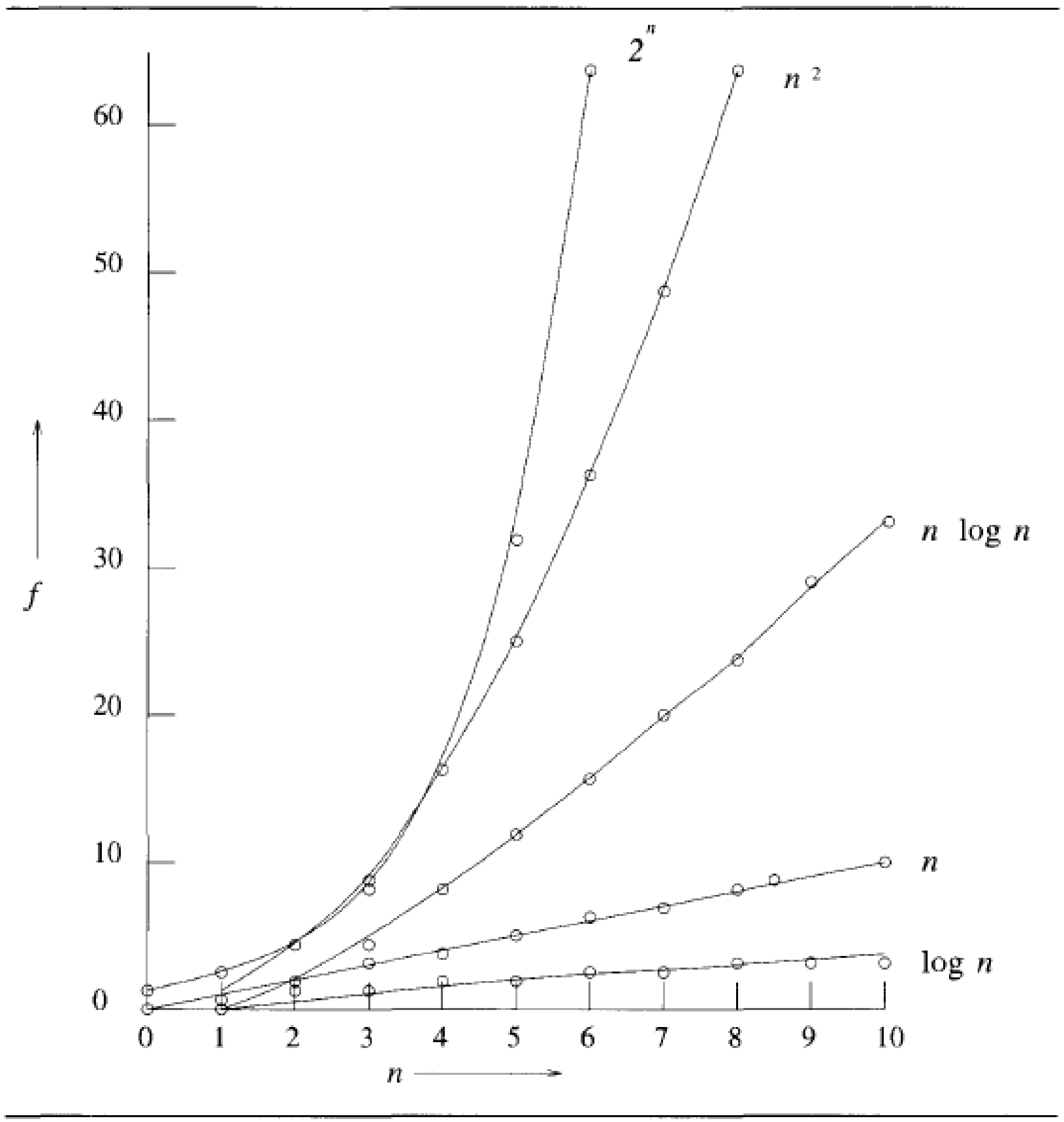
$$\lim_{n \to \infty} \frac{g(n)}{f(n)} = 0$$

Why is it called Asymptotic Analysis?

- From the discussion above, we can easily understand that, in every case for a given function $f(n)$ we are trying to find another function $g(n)$ which approximates $f(n)$ at higher values of n.

- In mathematics we call such a curve an *asymptotic curve*. In other terms, $g(n)$ is the asymptotic curve for $f(n)$. For this reason, we call algorithm analysis *asymptotic analysis.*

Different Asymptotic functions



Guidelines for Asymptotic Analysis

- 1) **Loops:** The running time of a loop is, at most, the running time of the statements inside the loop (including tests) multiplied by the number of iterations.

```
// executes n times
for (i=1; i<=n; i++)
    m = m + 2; // constant time, c
```

- 2) **Nested loops:** Analyze from the inside out. Total running time is the product of the sizes of all the loops.

```
//outer loop executed n times
for (i=1; i<=n; i++) {
    // inner loop executes n times
    for (j=1; j<=n; j++)
        k = k+1; //constant time
}
```

Guidelines for Asymptotic Analysis

- **3. Consecutive statements:** Add the time complexities of each statement.

```
x = x +1; //constant time
// executes n times
for (i=1; i<=n; i++)
    m = m + 2; //constant time
//outer loop executes n times
for (i=1; i<=n; i++) {
    //inner loop executed n times
    for (j=1; j<=n; j++)
        k = k+1; //constant time
}
```

Guidelines for Asymptotic Analysis

- 4) **If-then-else statements:** Worst-case running time: the test, plus *either* the *then* part *or* the *else* part (whichever is the larger).

```
//test: constant
if(length( ) == 0 ) {
    return false; //then part: constant
}
else {// else part: (constant + constant) * n
    for (int n = 0; n < length( ); n++) {
      // another if : constant + constant (no else part)
      if(!list[n].equals(otherList.list[n]))
          //constant
        return false;
    }
}
```

Guidelines for Asymptotic Analysis

- 5) **Logarithmic complexity:** An algorithm is O(*logn*) if it takes a constant time to cut the problem size by a fraction (usually by ½).

```
for (i=1; i<=n;)
    i = i*2;
```

```
for (i=n; i>=1;)
    i = i/2;
```

Tasks on Performance Analysis

- Problem 1: What is the running time of the following function?

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

Tasks on Performance Analysis

- **Problem 2:** Find the complexity of the program below.

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

Tasks on Performance Analysis

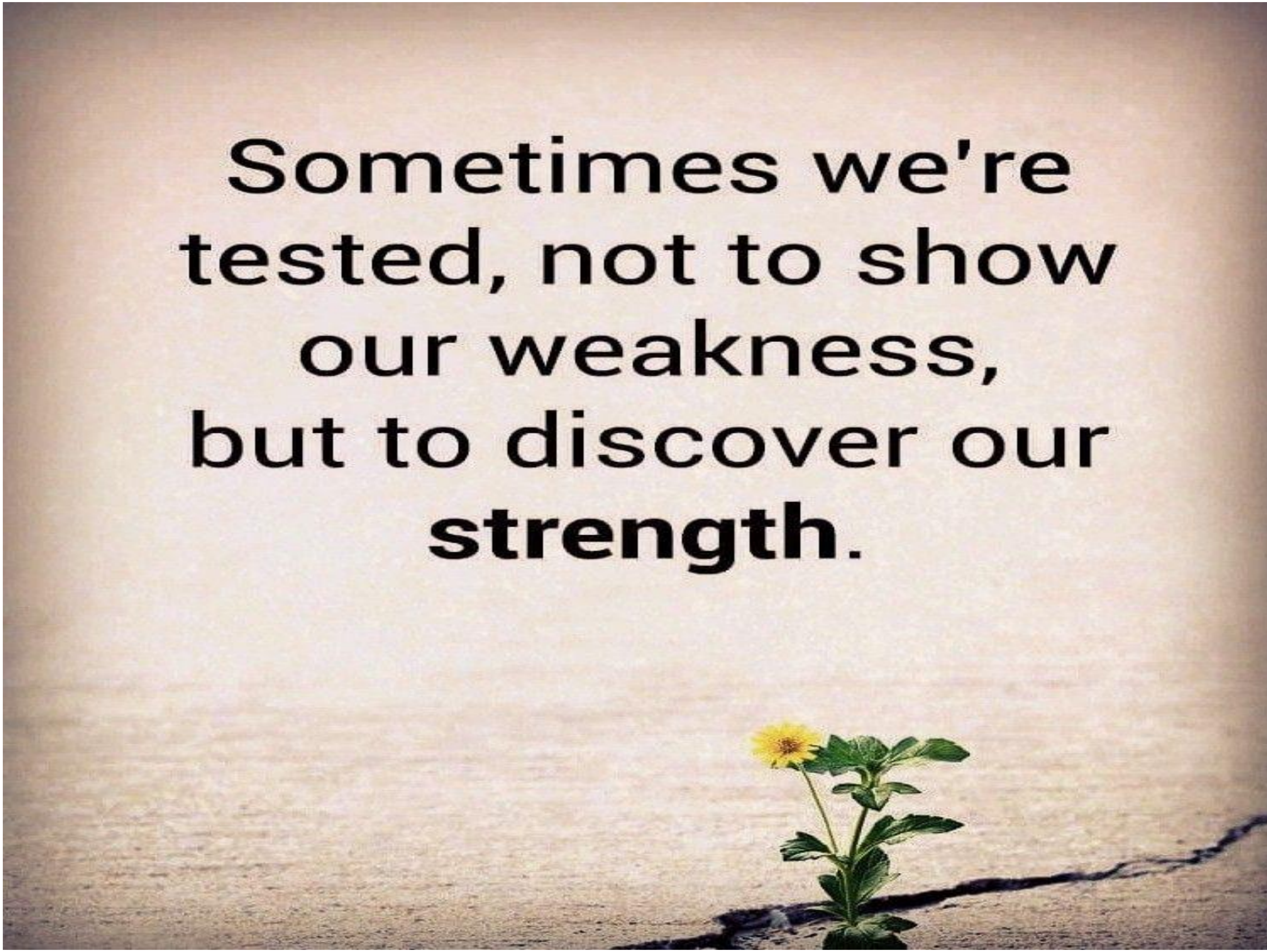
- **Problem 3:** Find the complexity of the program below.

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

Tasks on Performance Analysis

- **Problem 4:** Write a recursive function for the running time T(*n*) of the function given below.

```
function( int n ) {
        if( n == 1 ) return;
        for(int i = 1 ; i <= n ; i + + )
                for(int j = 1 ; j <= n ; j + + )
                        printf("*");
        function( n-3 );
}
```


Sometimes we're
tested, not to show
our weakness,
but to discover our
strength.