Design & Analysis of Algorithms(CSE2007)

MODULE 1

Fundamentals of Algorithmic problem solving

PRESIDENCY UNIVERSITY

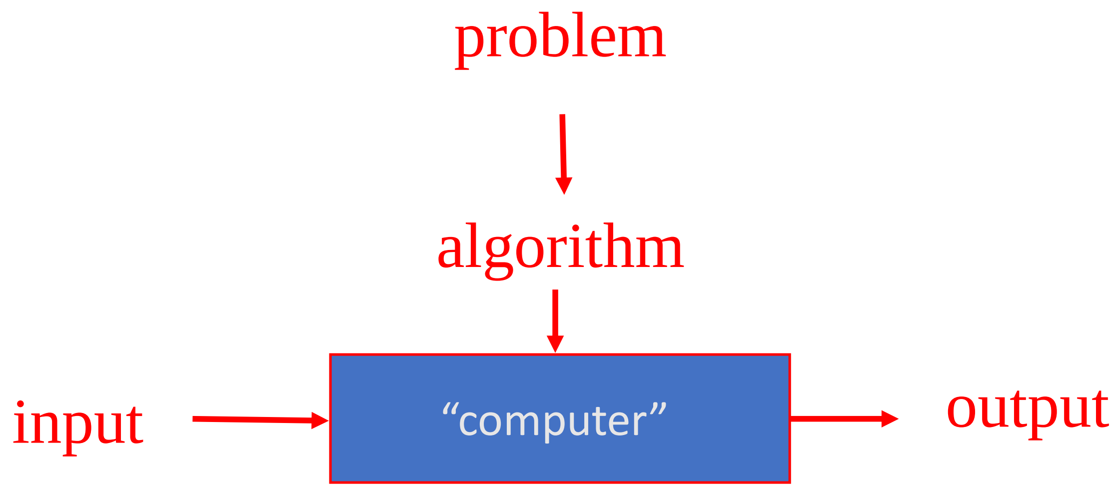
Private University Estd. in Karnataka State by Act No. 41 of 2013

What is an algorithm?

- word algorithm comes from Persian author, **Abu Ja'far Mohammed ibn Musa al Khowarizmi**
- Who wrote a textbook on mathematics
- An **algorithm** is an unambiguous step-by-step procedure to solve the given problem in finite number of steps by accepting set of legitimate inputs to produce the desired output.
- After producing the result, algorithm should terminate

PRESIDENCY UNIVERSITY
Private University Estd. in Karnataka State by Act No. 41 of 2013

Notion of Algorithm



Importance of writing algorithm?

- Writing an algorithm is just like preparing plan to solve the problem.
- Algorithm just gives the solution but not the answer.
- Importance of writing algorithms are:
  - ✓ We can save the resources like time, human effort & cost
  - ✓ Debugging will be easier.
  - ✓ Since they are written using pseudo code, any technical person can understand easily
  - ✓ Can be used as an design document
  - ✓ Once algorithm is written & understood neatly, easily can be converted into executable program by using any programming language

PRESIDENCY UNIVERSITY
Private University Estd. in Karnataka State by Act No. 41 of 2013
40 YEARS

Features/ Properties of Algorithm

- Every algorithm must satisfy the following criteria's

  Input: Should accept one or more external inputs

  Output: Should produce at least one output

  Effectiveness: Every instruction should transform the given I/P to desired output

  Finiteness: Algorithm should terminate after finite number of steps

  Definiteness: Each Instruction should be clear and unambiguous

Algorithm Design And Analysis Process

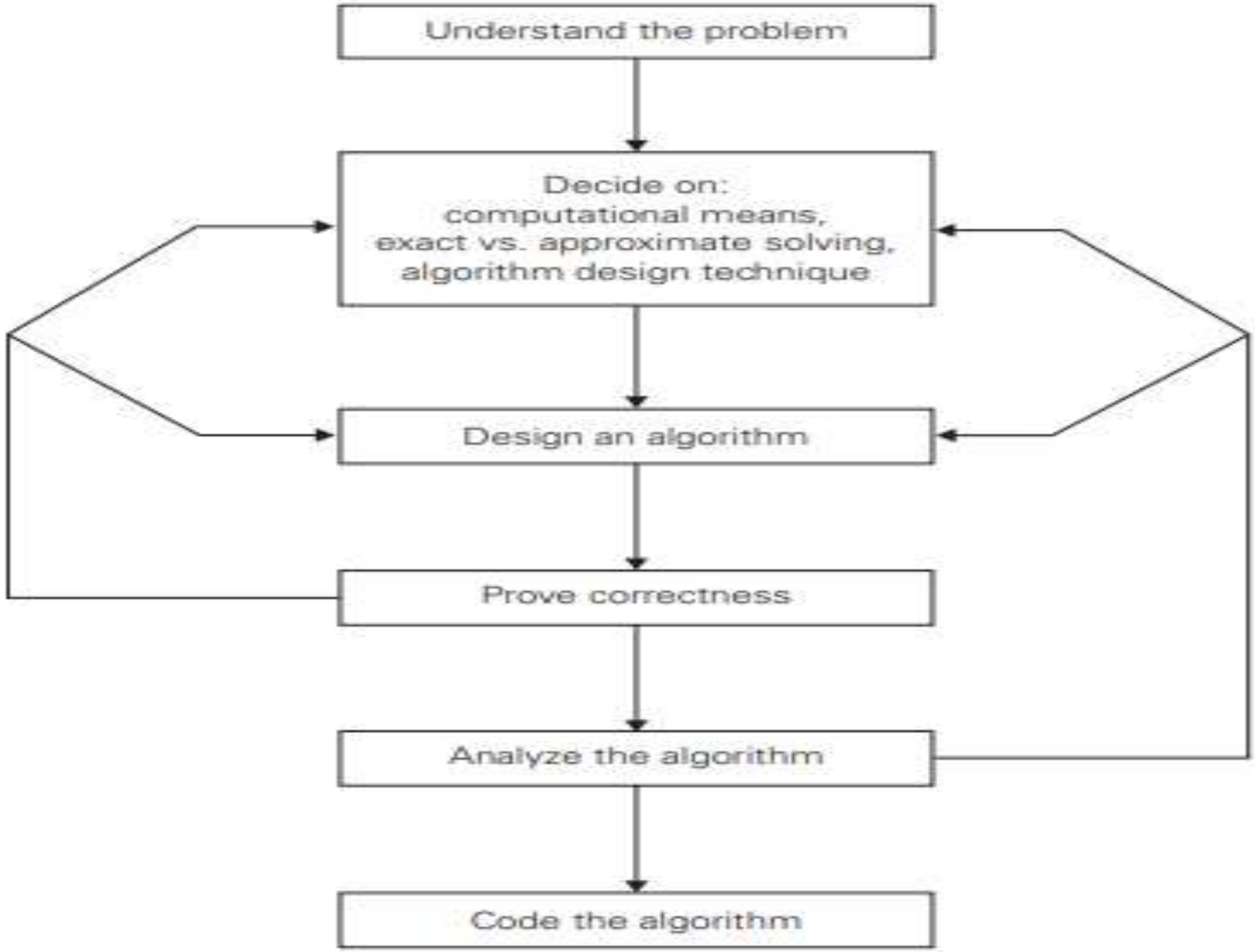


FIGURE 1.2 Algorithm design and analysis process.

Analysis of algorithms

- How good is the algorithm?
  - correctness
  - time efficiency
  - space efficiency

- Does there exist a better algorithm?
  - lower bounds
  - optimality

Important problem types

- sorting
- searching
- string processing
- graph problems

Sorting (I)

- Rearrange the items of a given list in ascending order.
  - Input: A sequence of n numbers $<a_1, a_2, \ldots, a_n>$
  - Output: A reordering $<a'_1, a'_2, \ldots, a'_n>$ of the input sequence such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.
- Why sorting?
  - Help searching
  - Algorithms often use sorting as a key subroutine.
- Sorting key
  - A specially chosen piece of information used to guide sorting. E.g., sort student records by names.

PRESIDENCY UNIVERSITY

Sorting (II)

- Examples of sorting algorithms
  - Selection sort
  - Bubble sort
  - Insertion sort
  - Merge sort
  - Heap sort …
- Evaluate sorting algorithm complexity: the number of key comparisons.
- Two properties
  - Stability: A sorting algorithm is called stable if it preserves the relative order of any two equal elements in its input.
  - In place : A sorting algorithm is in place if it does not require extra memory, except, possibly for a few memory units.

Searching

- Find a given value, called a search key, in a given set.
- Examples of searching algorithms
  - Sequential search
  - Binary search ...

```
Input: sorted array a[i] < … < a[j] and key x;
m ←(i+j)/2;
while i < j and x != a[m] do
    if x < a[m] then j ← m-1
                    else  i ← m+1;
 if x = a[m] then output a[m];
```

Time: O(log n)

PRESIDENCY UNIVERSITY
Private University Estd. in Karnataka State by Act No. 41 of 2013

String Processing

- A string is a sequence of characters from an alphabet.
- Text strings: letters, numbers, and special characters.
- String matching: searching for a given word/pattern in a text.

Examples:

(i) searching for a word or phrase on WWW or in a Word document

(ii) searching for a short read in the reference genomic sequence

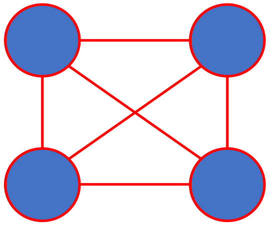
Graph Problems

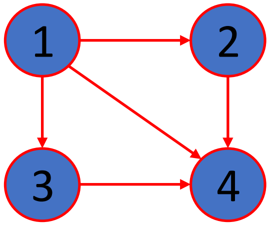
- Informal definition
  - A graph is a collection of points called vertices, some of which are connected by line segments called edges.

  - Types of graphs
    - Adjacency matrix
    - Cost adjacency matrix
    - Spanning tree

Graphs

- Formal definition
  - A graph $G = \langle V, E \rangle$ is defined by a pair of two sets: a finite set V of items called vertices and a set E of vertex pairs called edges.
- Undirected and directed graphs (digraphs).
- Complete, dense, and sparse graphs
  - A graph with every pair of its vertices connected by an edge is called complete, $K_{|V|}$



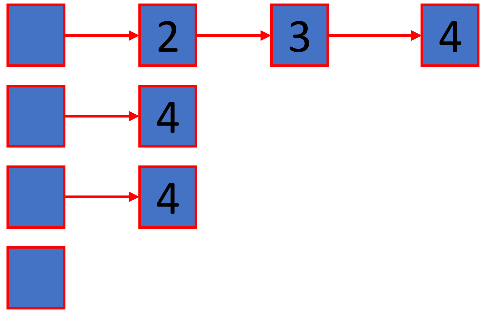


PRESIDENCY UNIVERSITY
Private University Estd. in Karnataka State by Act No. 41 of 2013

Graph Representation

- Adjacency matrix
  - n x n boolean matrix if |V| is n.
  - The element on the ith row and jth column is 1 if there's an edge from ith vertex to the jth vertex; otherwise 0.
  - The adjacency matrix of an undirected graph is symmetric.
- Adjacency linked lists
  - A collection of linked lists, one for each vertex, that contain all the vertices adjacent to the list's vertex.

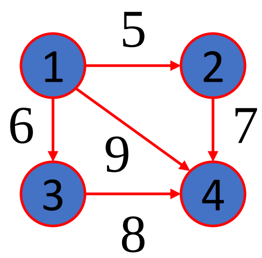
$$\begin{matrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{matrix}$$



PRESIDENCY UNIVERSITY
Private University Estd. in Karnataka State by Act No. 41 of 2013

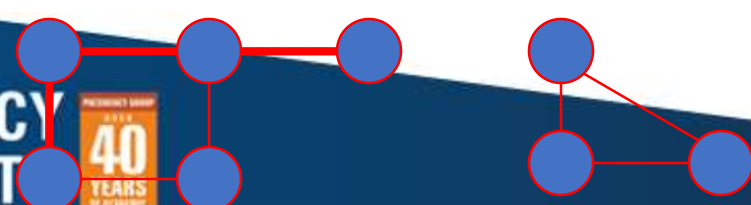
Weighted Graphs

- Weighted graphs
  - Graphs or digraphs with numbers assigned to the edges.



Graph Properties -- Paths and Connectivity

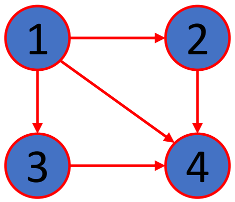
- Paths
  - A path from vertex u to v of a graph G is defined as a sequence of adjacent (connected by an edge) vertices that starts with u and ends with v.
  - Simple paths: All edges of a path are distinct.
  - Path lengths: the number of edges, or the number of vertices – 1.
- Connected graphs
  - A graph is said to be connected if for every pair of its vertices u and v there is a path from u to v.
- Connected component
  - The maximum connected subgraph of a given graph.



PRESIDENCY UNIVERSITY
Private University Estd. in Karnataka State by Act No. 41 of 2013

Graph Properties -- Acyclicity

- Cycle
  - A simple path of a positive length that starts and ends a the same vertex.
- Acyclic graph
  - A graph without cycles
  - DAG (Directed Acyclic Graph)



Algorithm design strategies

- **Brute force**
- **Divide and conquer**
- **Decrease and conquer**
- **Transform and conquer**
- **Greedy approach**
- **Dynamic programming**
- **Backtracking and branch-and-bound**
- **Space and time tradeoffs**

Analysis Framework

- The process of finding the efficiency of an algorithm is called as **analysis of algorithm.**
- We can find the efficiency of an algorithm in 2 ways
  1) Time efficiency/complexity
  2) Space efficiency/complexity
- We do analysis of algorithms because of following reasons
  1) To compare different algorithms for the same task
  2) To predict the performance in a new environment
  3) To specify the range of inputs on which algorithm works properly

Time complexity or time efficiency

- **Time complexity** of an algorithm is the amount of time taken by the program to run completely & efficiently
- Factors on which time complexity of an algorithm depends are
  1) Speed of computer
  2) Choice of programming language
  3) Compiler used
  4) Choice of algorithmic design technique
  5) Number of inputs/outputs
  6) Size of the inputs/outputs

PRESIDENCY UNIVERSITY

Private University Estd. in Karnataka State by Act No. 41 of 2013

Time complexity or time efficiency

- By considering the number of inputs given to the algorithm & size of the inputs given to the algorithm, time efficiency is normally computed by considering the **base operation** or **basic operation**
- The statement/instruction which is consuming more time is called as **basic operation**

Time complexity or time efficiency

- To find the time efficiency, we need to
  1) Identify the base operation
  2) Find the time taken by the basic operation to execute once
  3) Find, how many times, this basic operation is executing
- *Basic operation*: the operation that contributes the most towards the running time of the algorithm

n is the input size

$$T(n) \approx c_{op}C(n)$$

running time of the algorithm

Time taken by the basic operation to execute once

Total number of times, the basic operation is executing

Note: Different basic operations may cost differently!

Order of growth

- As the value of n(input size) increases, time required for execution also increases i.e., behavior of algorithm changes with the increase in the value of n. This change in the behavior is called **orders of growth**
- Most important: Order of growth within a constant multiple as $n \to \infty$
- Example:
  - How much faster will algorithm run on computer that is twice as fast?
  - How much longer does it take to solve problem of double input size?

Order of growth for few values of n

n	$\log_2 n$	n	n $\log_2 n$	n^2	n^3	2^n	n!
1	0	1	0	1	1	2	1
2	1	2	2	4	8	4	2
4	2	4	8	16	64	16	24
8	3	8	24	64	512	256	40320
16	4	16	64	256	4096	65536	HIGH
32	5	32	160	1024	32768	4294967296	VERY HIGH

The order of growth of basic efficiency classes is

$$1 < \log_2 n < n < n \log_2 n < n^2 < n^3 < 2^n < n!$$

Input size and basic operation examples

Problem	***Input size measure***	***Basic operation***
Searching for key in a list of *n* items	**Number of list's items, i.e. *n***	**Key comparison**
Multiplication of two matrices	**Matrix dimensions or total number of elements**	**Multiplication of two numbers**
Checking primality of a given integer *n*	***n'*size = number of digits (in binary representation)**	**Division**
Typical graph problem	**#vertices and/or edges**	**Visiting a vertex or traversing an edge**

Basic asymptotic efficiency classes

1	**constant**
log *n*	**logarithmic**
n	**linear**
n* log *n	***n*-log-*n***
n^2	**quadratic**
n^3	**cubic**
2^n	**exponential**
***n*!**	**factorial**

Space Complexity or Space Efficiency

- **Space complexity** of an algorithm is the amount of space consumed by the program to run completely & efficiently.
- Total amount of space consumed by the program is calculated by sum of the following components
  - Fixed space requirements
  - Variable space requirements
- Fixed space requirements are the requirements that do not depend on the number of inputs & outputs and also size of inputs & outputs of the program.
  - Program space (instruction space to store the code)
  - Data space (space for constants, variables, structures, etc)
  - Stack space (space for parameters, local variables, return values, etc)

Fixed space requirements = Program space + Data Space + Stack Space

Space Complexity or Space Efficiency

<u>Variable space requirement</u>

- Along with fixed space requirements, it also includes the extra space required
  1) When function uses recursion
  2) Dynamically allocated arrays, structures, etc
- So, space 'S' of a program 'P' on a particular instance 'I' is denoted by $S_P(I)$ = Fixed space requirements + space used during recursion + space used by run time variables
- Total space 'S' of a program 'P' is given by

$$S(P)=c + S_P(I)$$

Asymptotic Notations

- Mathematical notations used to express the time complexity of an algorithm is called **asymptotic notations**
- The 3 different asymptotic notations are
  1) O (Big-oh) for worst case
  2) Ω (Big-Omega) for best case
  3) Θ (Big-Theeta) for average case

O (Big-oh) notation

- Big-oh is the formal method of expressing the upper bound of an algorithm's running time.
- It is a measure of longest amount of time it could possibly take for the algorithm to complete.
- "The function, $t(n)$ is said to be in $O(g(n))$, which is denoted by $t(n) \in O(g(n))$ such that, there exist a positive constant c & positive integer n_0 satisfying the constraint $t(n) \le c\, g(n)$ for all $n \ge n_0$ "

PRESIDENCY UNIVERSITY

Private University Estd. in Karnataka State by Act No. 41 of 2013

Big-oh

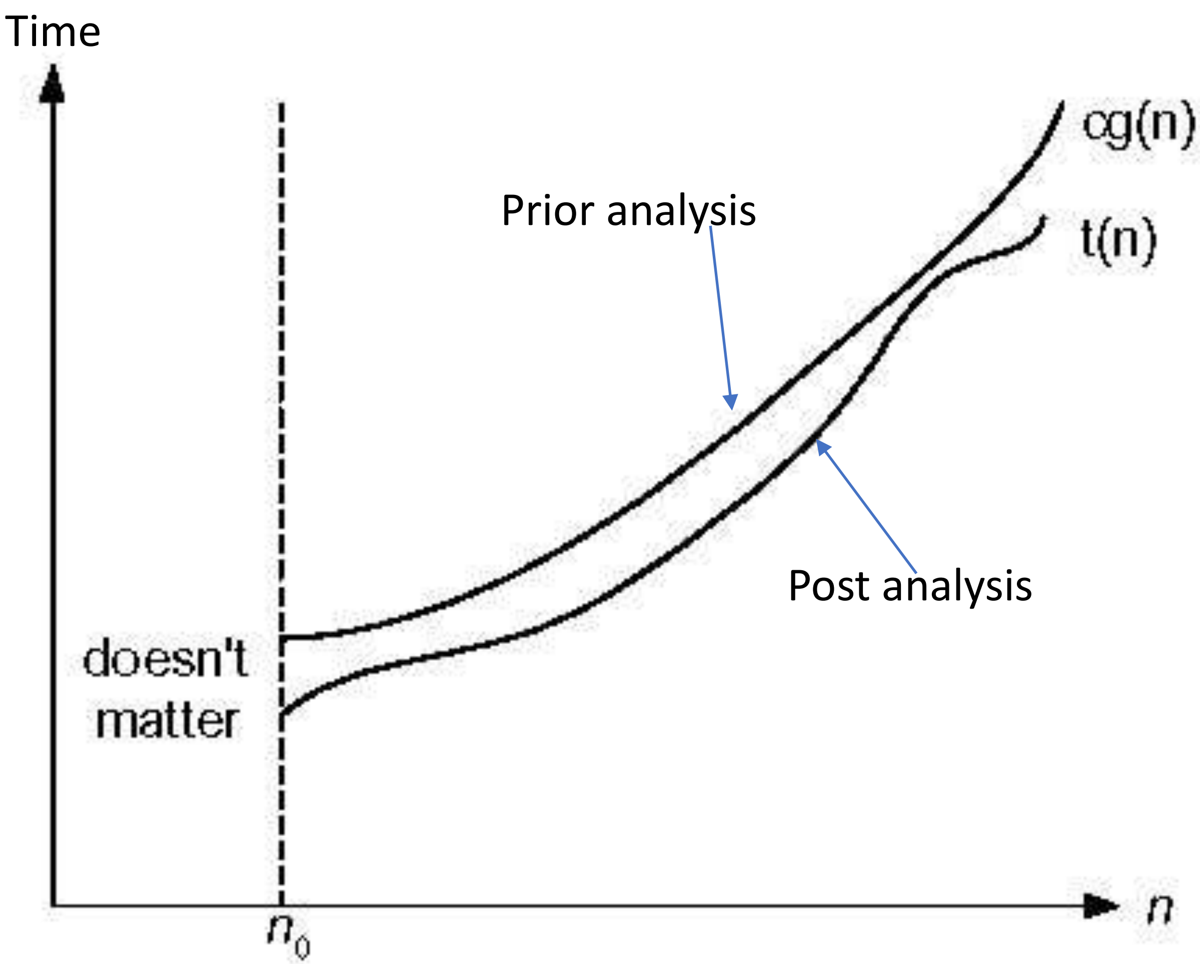


Figure 2.1 Big-oh notation: $t(n) \in O(g(n))$

Ω (Big-Omega) notation

- Big-omega (Ω) is the formal method of expressing the lower bound of an algorithm's running time.
- It is a measure of minimum amount of time it could possibly take for the algorithm to complete.
- "The function, t(n) is said to be in $\Omega(g(n))$, which is denoted by $t(n) \in \Omega(g(n))$ such that, there exist a positive constant c & positive integer n_0 satisfying the constraint $t(n) \geq c\, g(n)$ for all $n \geq n_0$ "

PRESIDENCY UNIVERSITY
Private University Estd. in Karnataka State by Act No. 41 of 2013

Big-omega

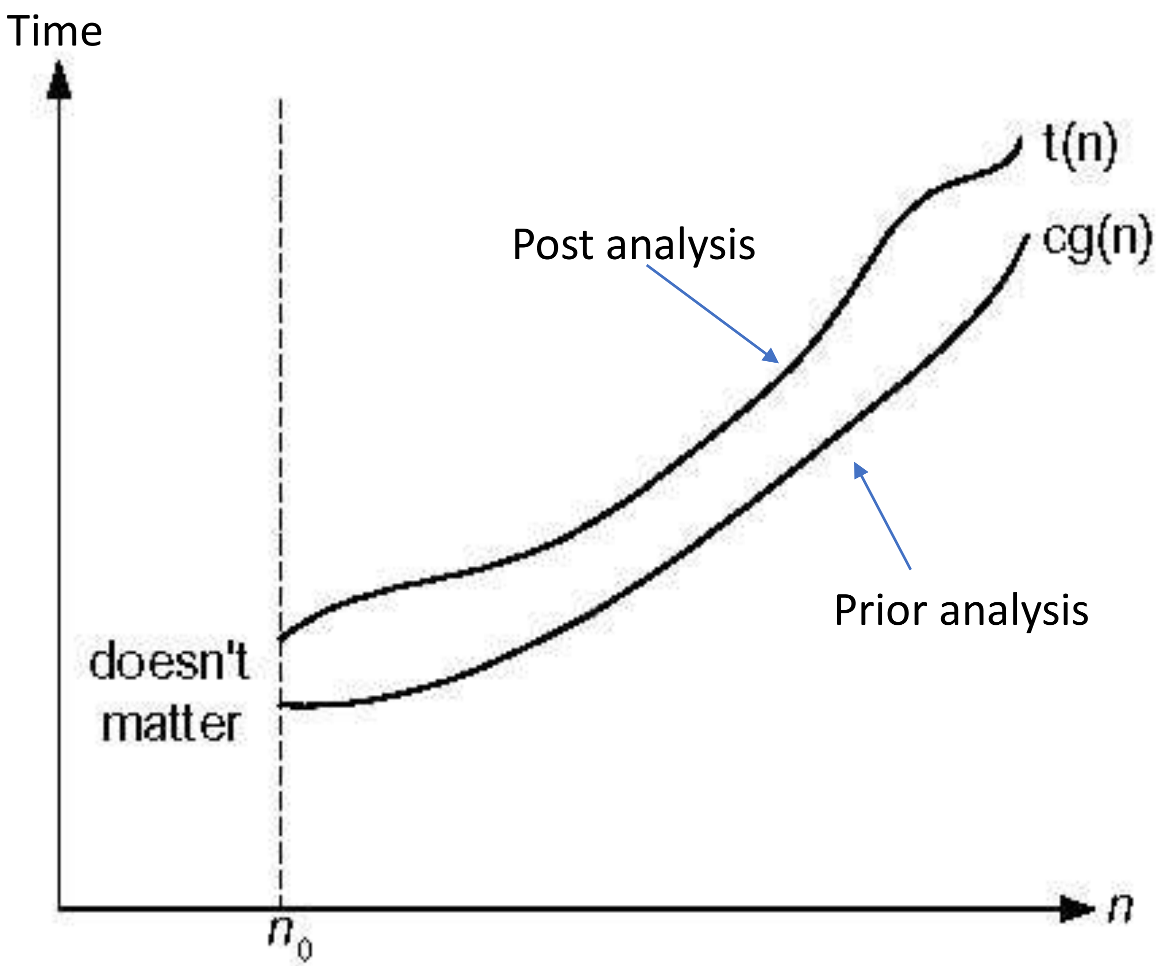


Fig. 2.2 Big-omega notation: $t(n) \in \Omega(g(n))$

Θ (Big-Theeta) notation

- Big-Theeta (Θ) is the formal method of expressing the average case efficiency of the algorithm.
- "The function, t(n) is said to be in Θ (g(n)), which is denoted by t(n)Є Θ (g(n)) such that, there exist a positive constants c1, c2 & positive integer n_0 satisfying the constraint $c1g(n) \leq t(n) \leq c2\, g(n)$ for all $n \geq n_0$ "

PRESIDENCY UNIVERSITY
Private University Estd. in Karnataka State by Act No. 41 of 2013

Big-theta

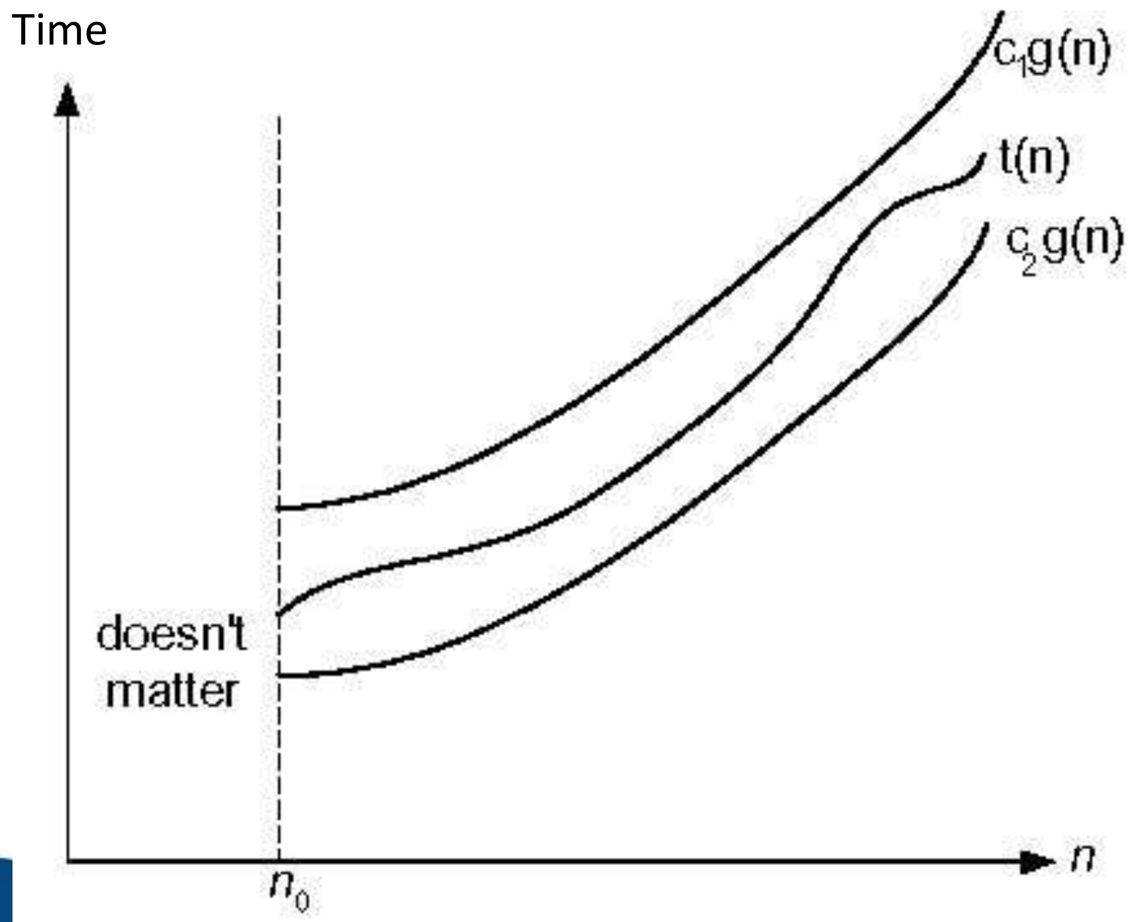


Figure 2.3 Big-theta notation: $t(n) \in \Theta(g(n))$

Example: Sequential search

ALGORITHM *SequentialSearch*$(A[0..n-1], K)$

//Searches for a given value in a given array by sequential search
//Input: An array $A[0..n-1]$ and a search key K
//Output: The index of the first element of A that matches K
// or -1 if there are no matching elements
$i \leftarrow 0$
while $i < n$ **and** $A[i] \neq K$ **do**
    $i \leftarrow i + 1$
if $i < n$ **return** i
else return -1

- Worst case — n key comparisons
- Best case — 1 comparisons
- Average case — (n+1)/2, assuming K is in A

- #include<time.h>
- Main()
- { clock_t start, end, total;
- Max=a[0];
- Start = clock();
- For(i=1;i<n; i++)
- {
- If(A[i]>max)
- Max=a[i];
- }
- End = clock();
- Total = double(end-start)/CLOCKS_PER_SEC;

Linear Search

- A[10], k, i=0;
- While(i<n)
- {
  - If(a[i]==k)
  - Return I;
  - Else i++;
  - }
  - If(i>=n)
  - Printf(“element not found”);

PRESIDENCY UNIVERSITY
Private University Estd. in Karnataka State by Act No. 41 of 2013

Mathematical Analysis of non-recursive algorithms

General Plan for Analysis of non-recursive algorithms

- Decide the number of input parameters given to the algorithm & decide the size of inputs. (identify the **problem size**)
- Identify algorithm's **basic operation**
- Decide whether we need to find only average case (Θ) or all the 3 cases separately.
  - *For this, check whether the number of times the basic operation is executed depends only on the problem size or not.*
  - *If it depends only on the problem size, then we need to find only average case* Θ
  - *If it also depends on some other additional properties, then we need to find all 3 cases separately.*
- Find the total number of times, the basic operation is executing & express it in sum expressions
- Simplify the expressions using standard formulas & rules of sum manipulations & obtain their order of growth.

Two summation rules:

$$\sum_{i=l}^{u} ca_i = c\sum_{i=l}^{u} a_i, \qquad \textbf{(R1)}$$

$$\sum_{i=l}^{u} (a_i \pm b_i) = \sum_{i=l}^{u} a_i \pm \sum_{i=l}^{u} b_i, \qquad \textbf{(R2)}$$

and two summation formulas

$$\sum_{i=l}^{u} 1 = u - l + 1 \quad \text{where } l \le u \text{ are some lower and upper integer limits,} \qquad \textbf{(S1)}$$

$$\sum_{i=0}^{n} i = \sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2} \approx \frac{1}{2}n^2 \in \Theta(n^2). \qquad \textbf{(S2)}$$

Example 1: Maximum element

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
```

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

T(n) = ***Σ1≤i≤n-1 1 = n-1 = Θ(n)*** comparisons

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

Example 2: Element uniqueness problem

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 $i \leftarrow 0$ **to** $n-2$ **do**
  for $j \leftarrow i+1$ **to** $n-1$ **do**
    if $A[i] = A[j]$ **return false**
return true

T(n) = *Σ0≤i≤n-2 (Σi+1≤j≤n-1 1)*

=(n-1)summation(-i)

= Σ0≤i≤n-2 n-i-1 = (n-2+1)(n-1)/2

= Θ(n)^2 comparisons

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

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

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

We also could have computed the sum $\sum_{i=0}^{n-2}(n-1-i)$ faster as follows:

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

PRESIDENCY UNIVERSITY
GAIN MORE KNOWLEDGE REACH GREATER HEIGHTS
Private University Estd. in Karnataka State by Act No. 41 of 2013
40 YEARS OF ACADEMIC WISDOM

Mathematical Analysis of Recursive Algorithms

General Plan for Analysis of recursive algorithms

- Decide the number of input parameters given to the algorithm & decide the size of inputs. (identify the **problem size**)
- Identify algorithm's **basic operation**
- Decide whether we need to find only average case (Θ) or all the 3 cases separately.
  - *For this, check whether the number of times the basic operation is executed depends only on the problem size or not.*
  - *If it depends only on the problem size, then we need to find only average case* Θ
  - *If it also depends on some other additional properties, then we need to find all 3 cases separately.*
- Obtain the recurrence relation with an appropriate initial condition for the number of times the basic operation is executing
- Solve the recurrence relation & obtain the order of growth & then express it using asymptotic notations

PRESIDENCY UNIVERSITY
Private University Estd. in Karnataka State by Act No. 41 of 2013

Example 3: factorial of a number

//Purpose: to compute n! recursively

//Input: A non-negative integer ‘n’

//Output: value of n!

```
ALGORITHM:- fact(n)
{ If n==0   then
        return 1
  else
        return n*fact(n-1)
}
```

PRESIDENCY UNIVERSITY

Analysis of example3

- Recursive relation is

fact(n)=$\begin{cases} 1 & if\ n = 0 \\ n\ * fact(n-1) & if\ n > 0 \end{cases}$

- Number of times multiplication operation is executing is

m(n)=$\begin{cases} 0 & if\ n = 0 \\ 1 + m(n-1) & if\ n > 0 \end{cases}$

- Time complexity to compute n! is given by the order of growth ‘n’.
- t(n)ЄΘ(n)

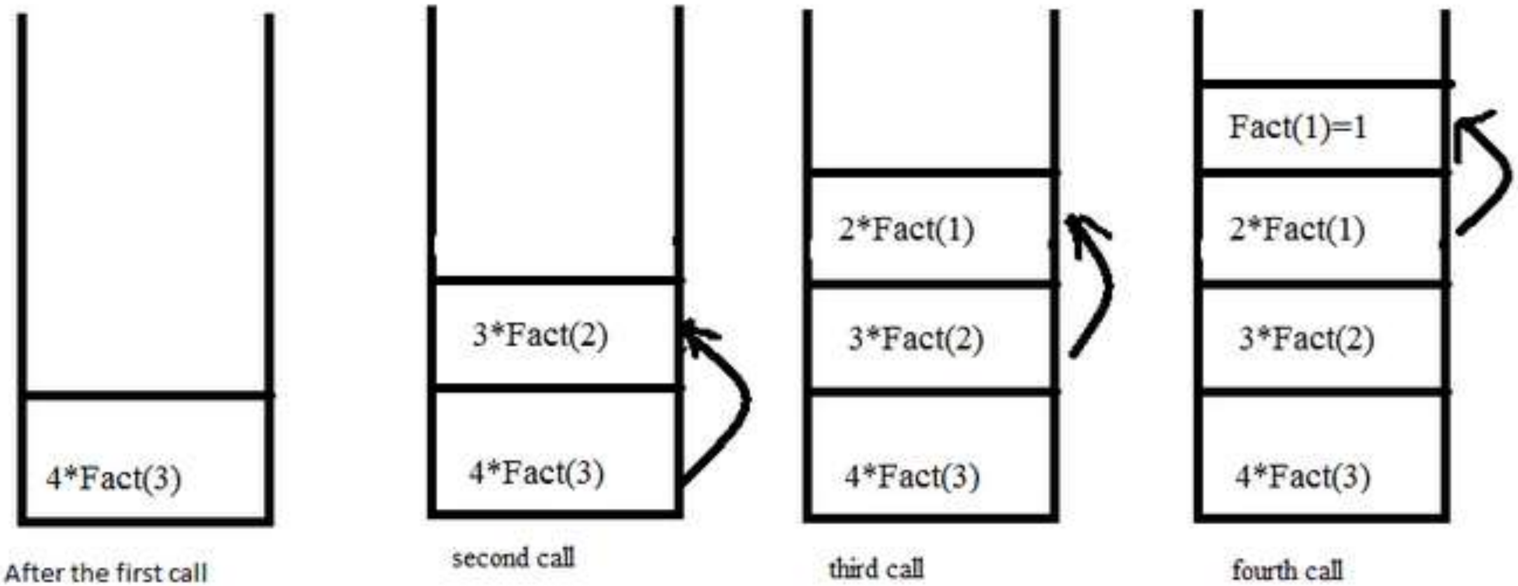
PRESIDENCY UNIVERSITY

GAIN MORE KNOWLEDGE REACH GREATER HEIGHTS

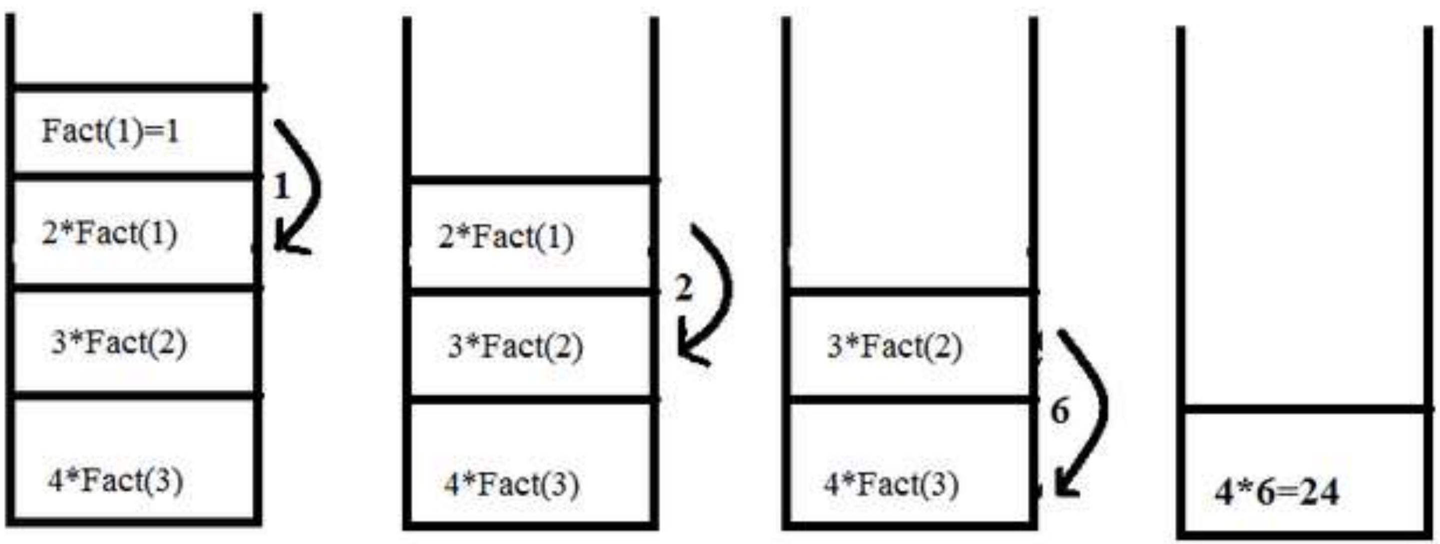
Private University Estd. in Karnataka State by Act No. 41 of 2013

40 YEARS OF ACADEMIC WISDOM

When function call happens previous variables gets stored in stack



Returning values from base case to caller function



Recursion in the stack

Solving the recurrence for M(n)

$M(n) = M(n\text{-}1) + 1, \; M(0) = 0$

M(n) = M(n-1) + 1

M(N-1) = M(n-1-1)+1= M(N-2)+1

= (M(n-2) + 1) + 1 = M(n-2) + 2

M(N-2) = M(N-2-1)+1= M(N-3)+1

= (M(n-3) + 1) + 2 = M(n-3) + 3

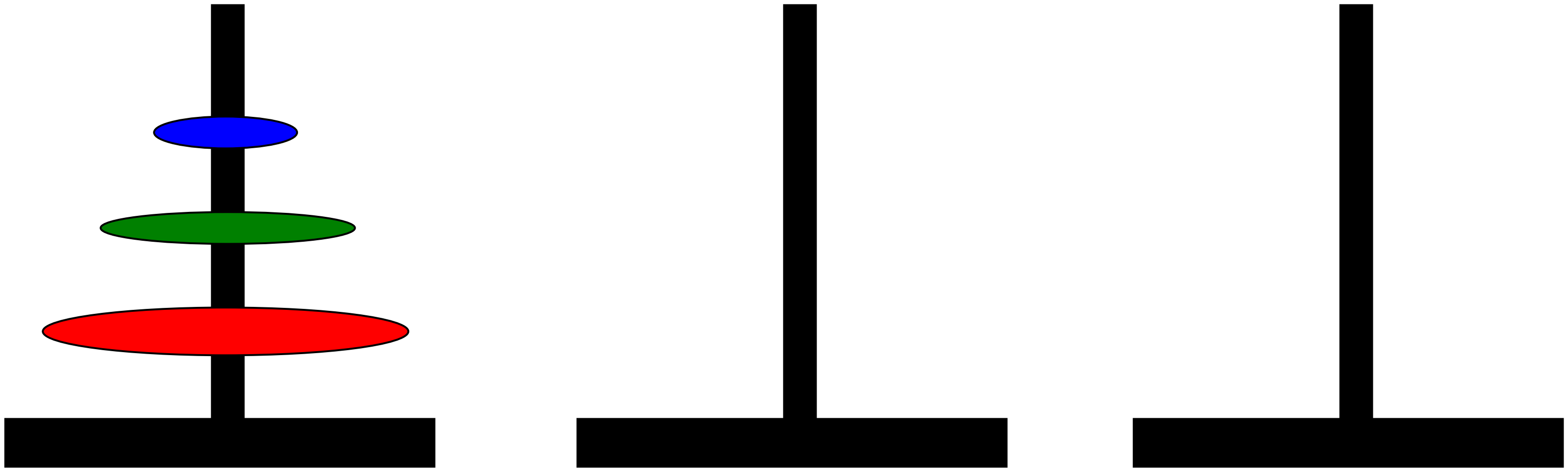
…

= M(n-i) + i = M(0) + n = n

The method is called backward substitution.

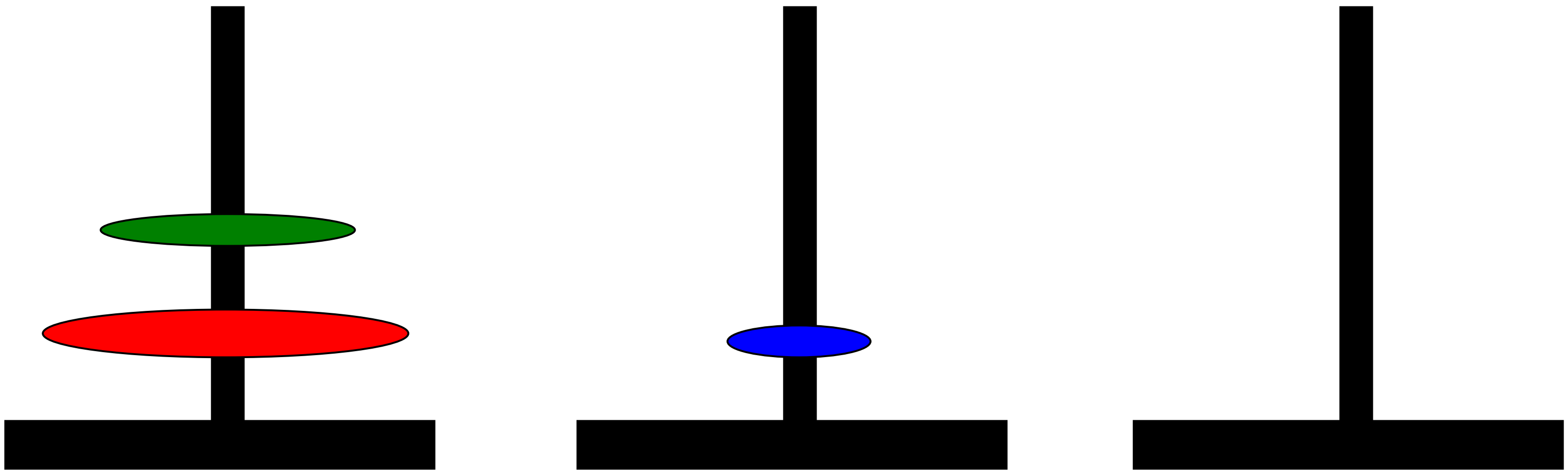
Tower of Hanoi

(0,0,0)



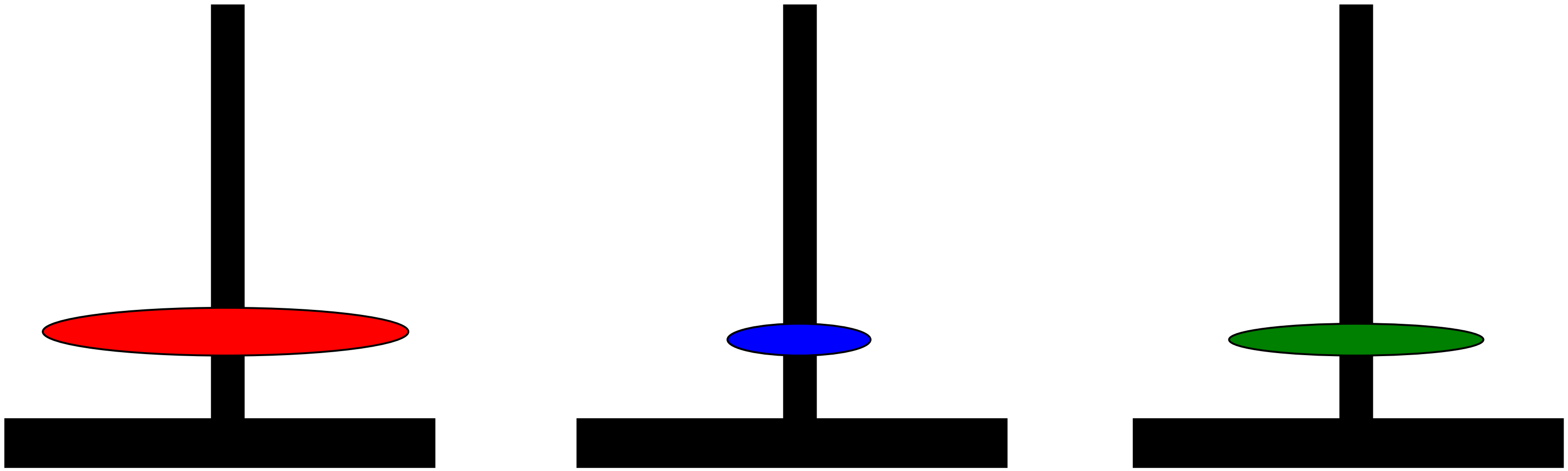
Tower of Hanoi

(0,0,1)



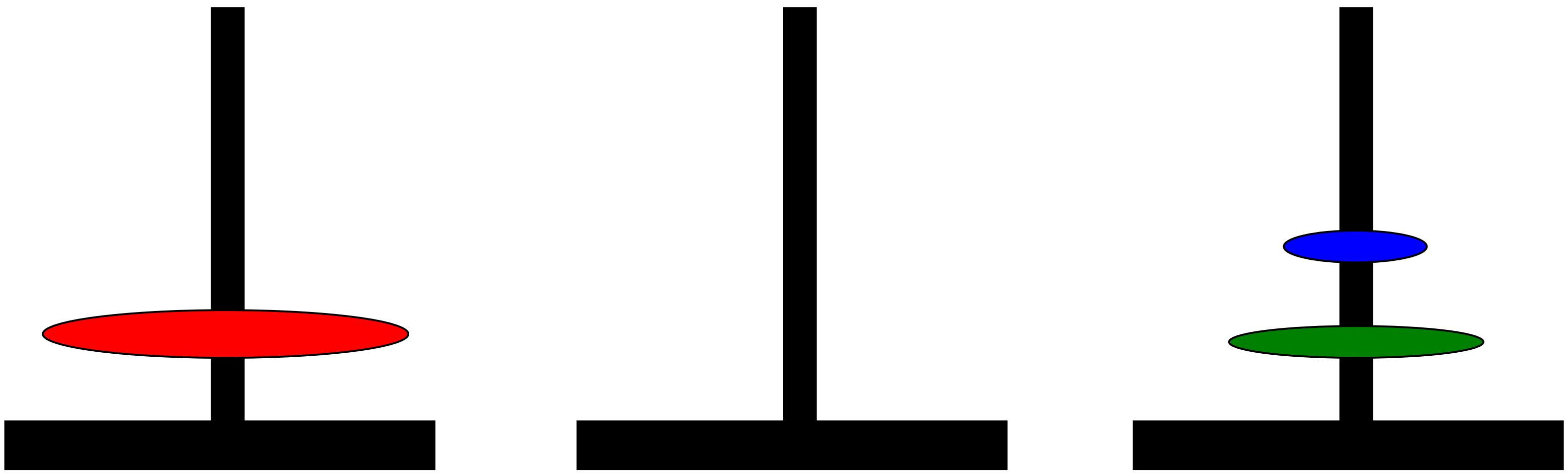
Tower of Hanoi

(0,1,1)



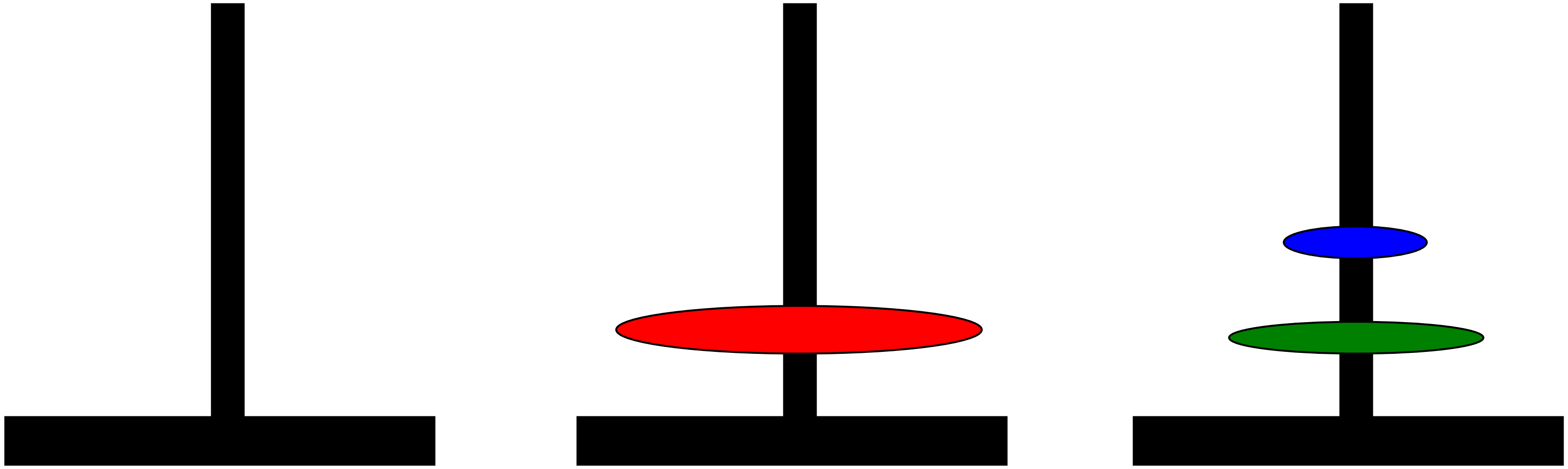
Tower of Hanoi

(0,1,0)



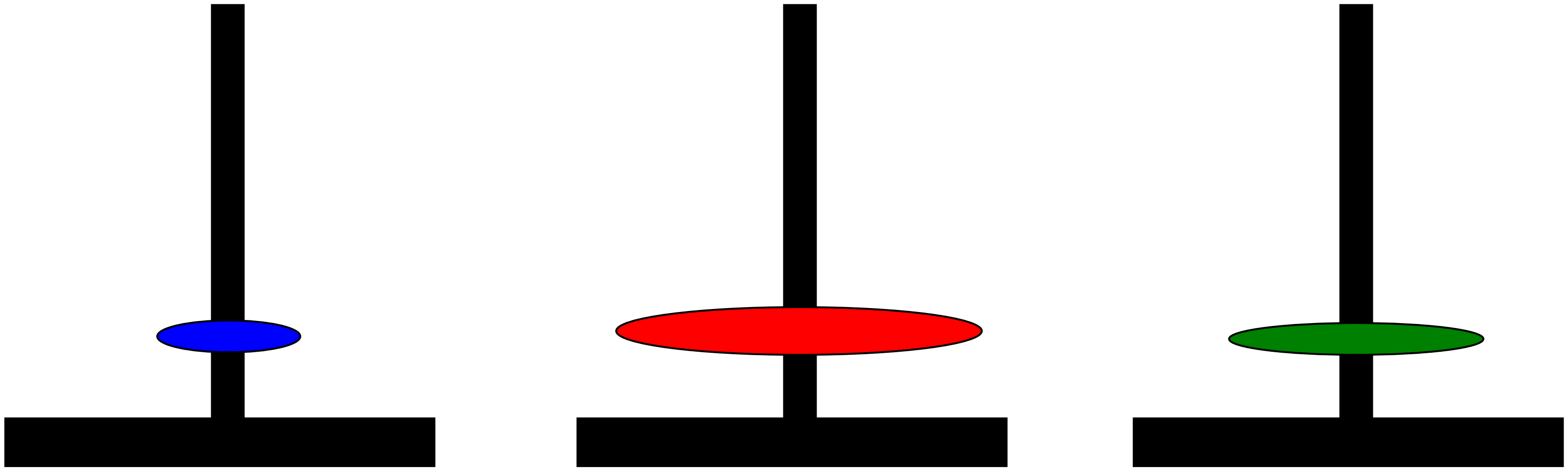
Tower of Hanoi

(1,1,0)



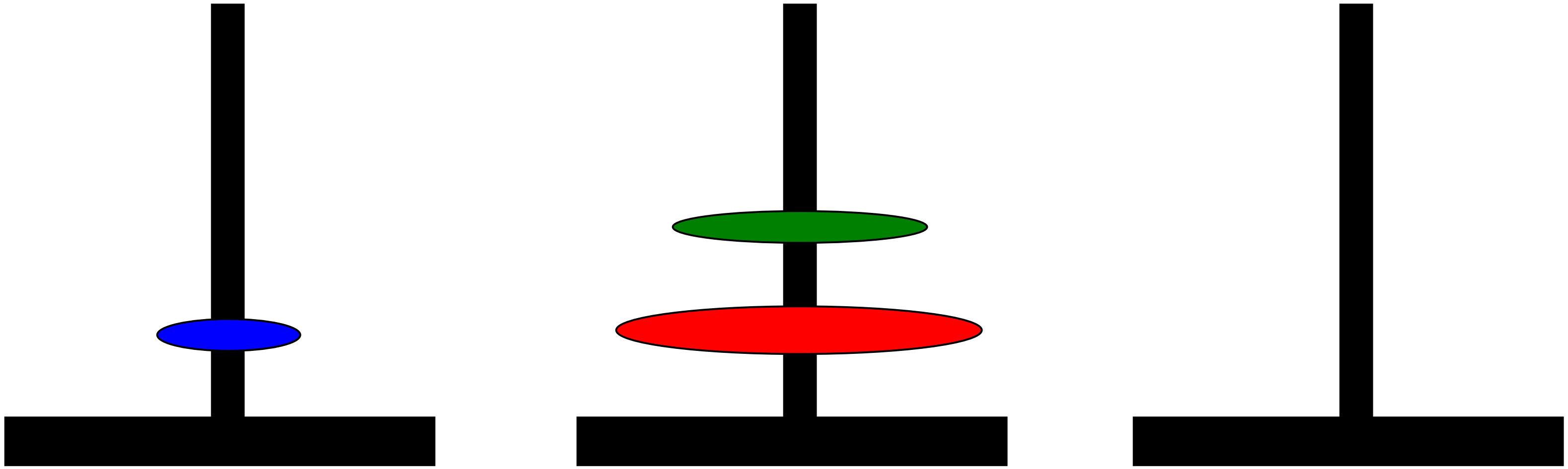
Tower of Hanoi

(1,1,1)



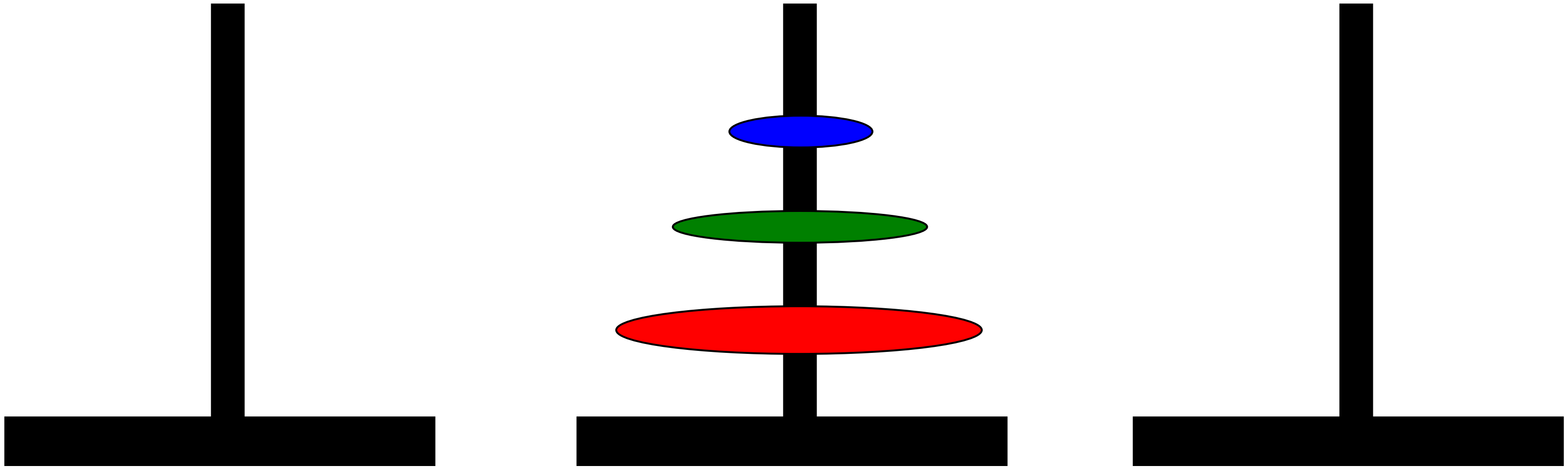
Tower of Hanoi

(1,0,1)



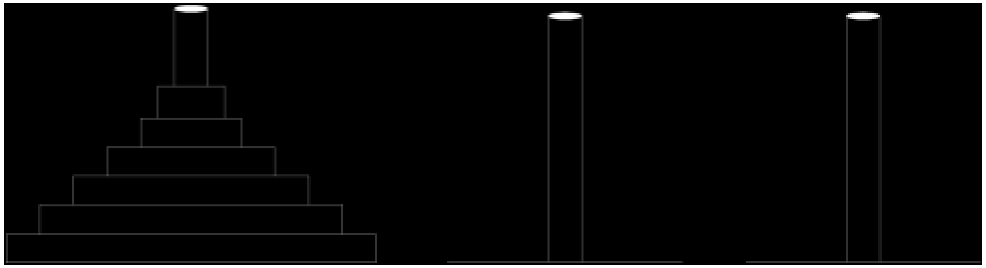
Tower of Hanoi

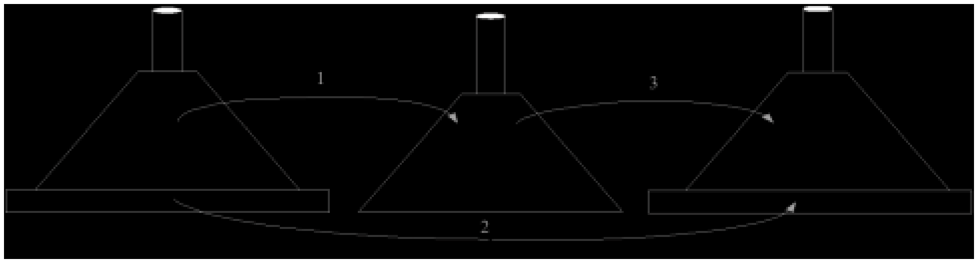
(1,0,0)



PRESIDENCY UNIVERSITY
Private University Estd. in Karnataka State by Act No. 41 of 2013

Example 2: The Tower of Hanoi Puzzle






```
Procedure Hanoi(disk, source, dest, aux)

   IF disk == 1, THEN
      move disk from source to dest
   ELSE
      Hanoi(disk - 1, source, aux, dest)      // Step 1
      move disk from source to dest           // Step 2
      Hanoi(disk - 1, aux, dest, source)      // Step 3
   END IF

END Procedure
```

Recurrence for number of moves:

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

PRESIDENCY UNIVERSITY
GAIN MORE KNOWLEDGE REACH GREATER HEIGHTS
Private University Estd. in Karnataka State by Act No. 41 of 2013
40 YEARS OF ACADEMIC WISDOM

Solving recurrence for number of moves

$M(n) = 2M(n\text{-}1) + 1, \;\; M(1) = 1$

$M(n) = 2M(n\text{-}1) + 1$

$M(N\text{-}1) = 2M(N\text{-}1\text{-}1)+1 = 2M(N\text{-}2)+1$

= 2(2M(n-2) + 1) + 1 = 2^2*M(n-2)+2+1

= 2^2*M(n-2) + 2^1 + 2^0

= 2^2*(2M(n-3) + 1) + 2^1 + 2^0

= 2^3*M(n-3) + 2^2 + 2^1 + 2^0

= 2^I M(N-I)+ 2^(n-2) + … + 2^1 + 2^0

= 2^(n-1)*M(1) + 2^(n-2) + … + 2^1 + 2^0

= 2^(n-1) + 2^(n-2) + … + 2^1 + 2^0

= 2^n - 1

PRESIDENCY UNIVERSITY

GAIN MORE KNOWLEDGE REACH GREATER HEIGHTS

Private University Estd. in Karnataka State by Act No. 41 of 2013

40 YEARS OF ACADEMIC WISDOM

- $M(n) = 2M(n-1) + 1$, $M(1) = 1$
- $= 2^2*M(n-2) + 2^1 + 2^0$
- $= 2^3*M(n-3) + 2^2 + 2^1 + 2^0$
- $= 2^I M(N-I)+2^{I-1} + \ldots\ldots 2^0$

$S = a(r^n - 1)/r-1 \dashrightarrow a=1, r=2, n=I$

$S= 1(2^i - 1)/(2-1) = 2^I - 1$

$= 2^I M(N-I)+ 2^I - 1$

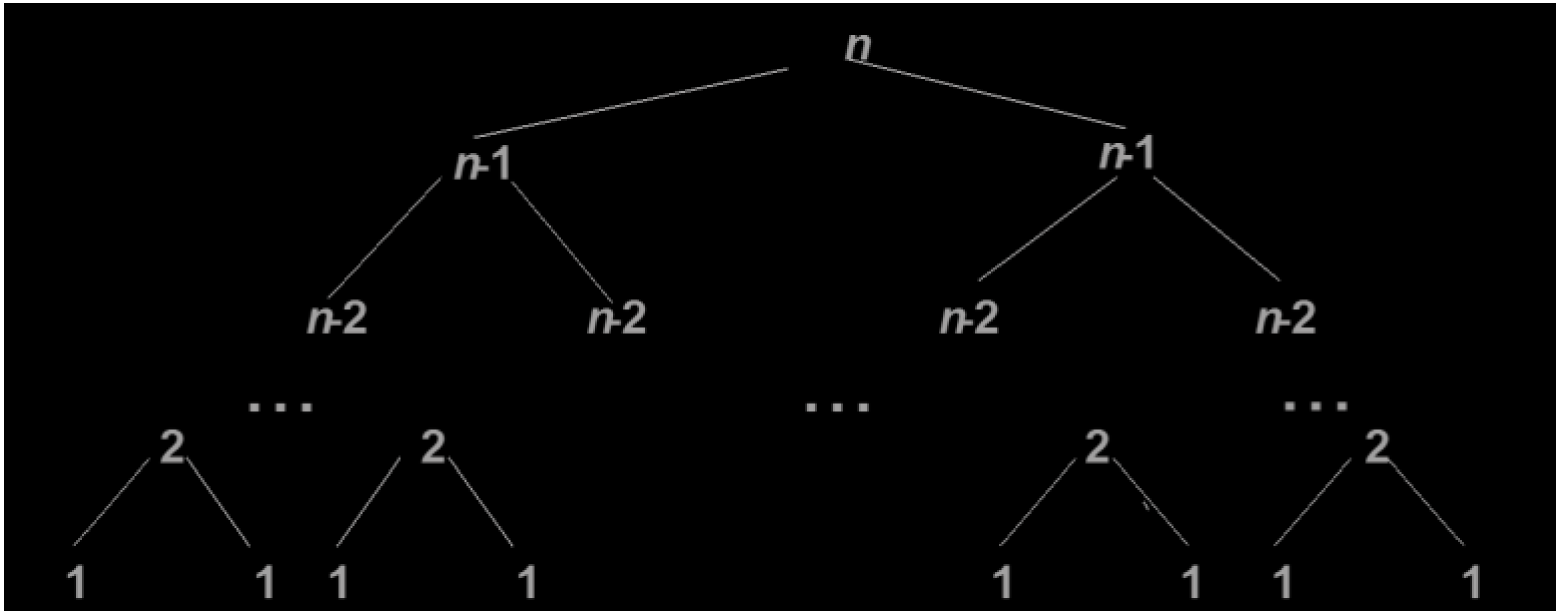
$I=N-1$

PRESIDENCY UNIVERSITY

- $= 2^I M(N-I) + 2^I - 1$
- $I = n-1$
- $2^{n-1} M(N-(N-1)) + 2^{N-1} - 1$
- $2^{n-1} M(1) + 2^{N-1} - 1$
- $2^{n-1} + 2^{n-1} - 1$
- $2 * 2^{n-1} – 1 = 2*(2^n / 2) – 1 = 2^n - 1$

PRESIDENCY UNIVERSITY
GAIN MORE KNOWLEDGE REACH GREATER HEIGHTS
Private University Estd. in Karnataka State by Act No. 41 of 2013
40 YEARS OF ACADEMIC WISDOM

Tree of calls for the Tower of Hanoi Puzzle



PRESIDENCY UNIVERSITY
Private University Estd. in Karnataka State by Act No. 41 of 2013