

# Design and Analysis of Algorithms

## UNIT – I

Algorithms and Problem Solving

# Syllabus

CO1, CO2, CO3, CO4, CO5, CO6, CO7, CO8, CO9, CO10, CO11, CO12, CO13, CO14, CO15, CO16, CO17, CO18, CO19, CO20, CO21, CO22, CO23, CO24, CO25, CO26, CO27, CO28, CO29, CO30, CO31, CO32, CO33, CO34, CO35, CO36, CO37, CO38, CO39, CO40, CO41, CO42, CO43, CO44, CO45, CO46, CO47, CO48, CO49, CO50, CO51, CO52, CO53, CO54, CO55, CO56, CO57, CO58, CO59, CO60, CO61, CO62, CO63, CO64, CO65, CO66, CO67, CO68, CO69, CO70, CO71, CO72, CO73, CO74, CO75, CO76, CO77, CO78, CO79, CO80, CO81, CO82, CO83, CO84, CO85, CO86, CO87, CO88, CO89, CO90, CO91, CO92, CO93, CO94, CO95, CO96, CO97, CO98, CO99, CO100

## Course Contents

Unit I	Algorithms and Problem Solving	07 Hours
<p>Algorithm: The Role of Algorithms in Computing - What are algorithms, Algorithms as technology, Evolution of Algorithms, Design of Algorithm, Need of Correctness of Algorithm, Confirming correctness of Algorithm – sample examples, Iterative algorithm design issues.</p> <p>Problem solving Principles: Classification of problem, problem solving strategies, classification of timecomplexities (linear, logarithmic etc.)</p>		
#Exemplar/Case Studies	Towers of Hanoi	
*Mapping of Course Outcomes for Unit I	CO1,CO3	

# Algorithms

- A tool for solving a well-specified computational problem



- Algorithms must be:
  - ❑ Correct: For each input produce an appropriate output
  - ❑ Efficient: run as quickly as possible, and use as little memory as possible – more about this later

# Role of Algorithms in Computing

- The word Algorithm means “a process or set of rules to be followed in calculations or other problem-solving operations”.
- Therefore Algorithm refers to a set of rules/instructions that step-by-step define how a work is to be executed upon in order to get the expected results.

## Algorithm Design Technique

- Divide and conquer — problem is divided into smaller instances
- Dynamic Programming — Results of smaller , reoccurring instances are obtained to solve problem
- Greedy Technique — Solve the problem by making locally optimal decisions
- Backtracking-

# Role of Algorithms in Computing

## Analysis of Algorithm

- Time Efficiency — Indicates how fast algorithm runs
- Space Efficiency — How much extra memory the algorithm needs to complete its execution
- Simplicity — Generating sequence of instructions which are easy to understand
- Generality — Range of inputs it can accept .

# Algorithms Cont.

- A well-defined **computational procedure** that takes some value, or set of values, as **input** and produces some value, or set of values, as **output**.
- Written in a **pseudo code** which can be implemented in the language of programmer's choice.

# Correct and incorrect algorithms

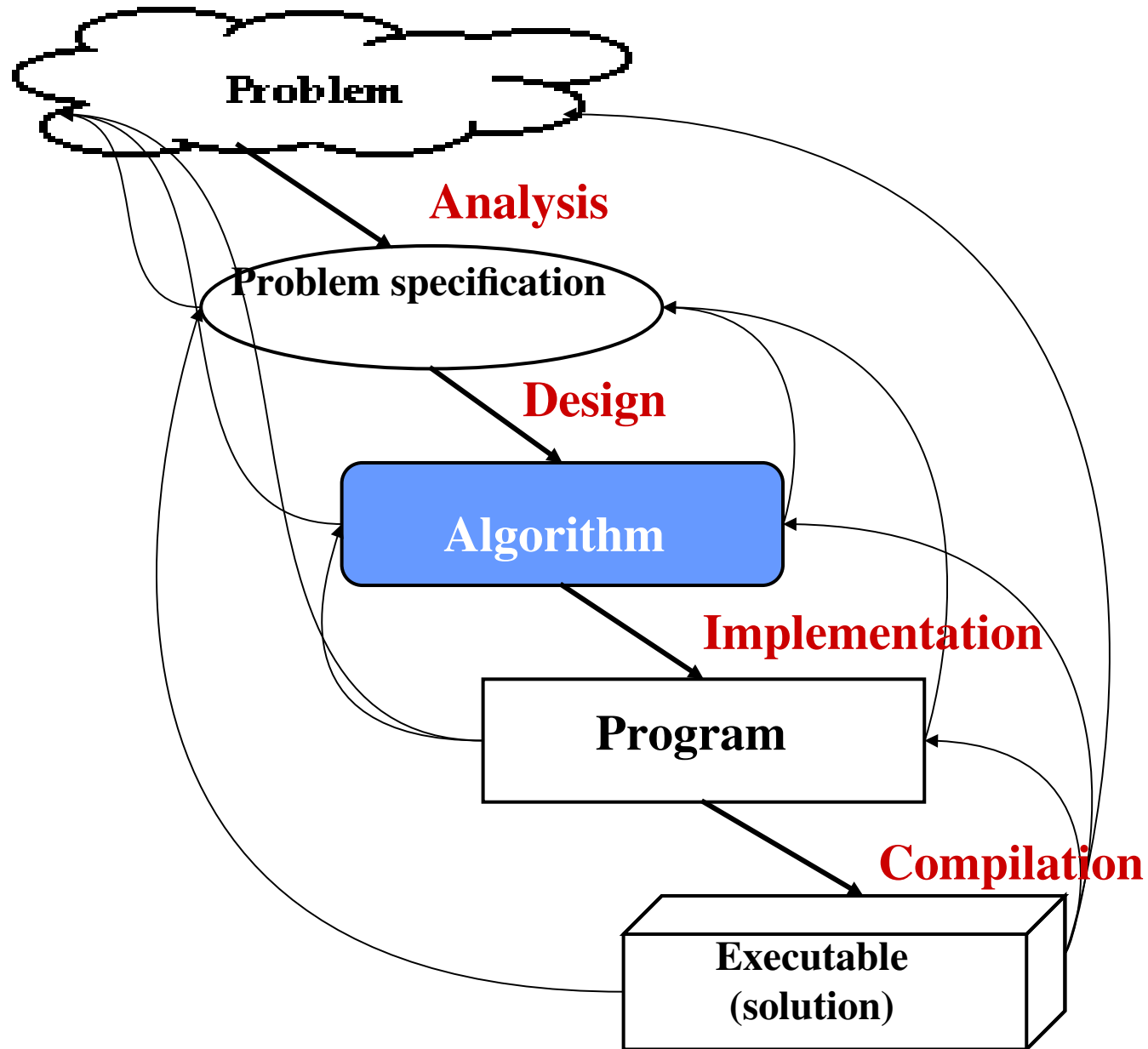
- Algorithm is correct if, for every input instance, it ends with the correct output. We say that a correct algorithm solves the given computational problem.
- An incorrect algorithm **might not end** at all on some input instances, or it might end with an answer other than the desired one.
- We shall be concerned only with correct algorithms.

# Problems and Algorithms

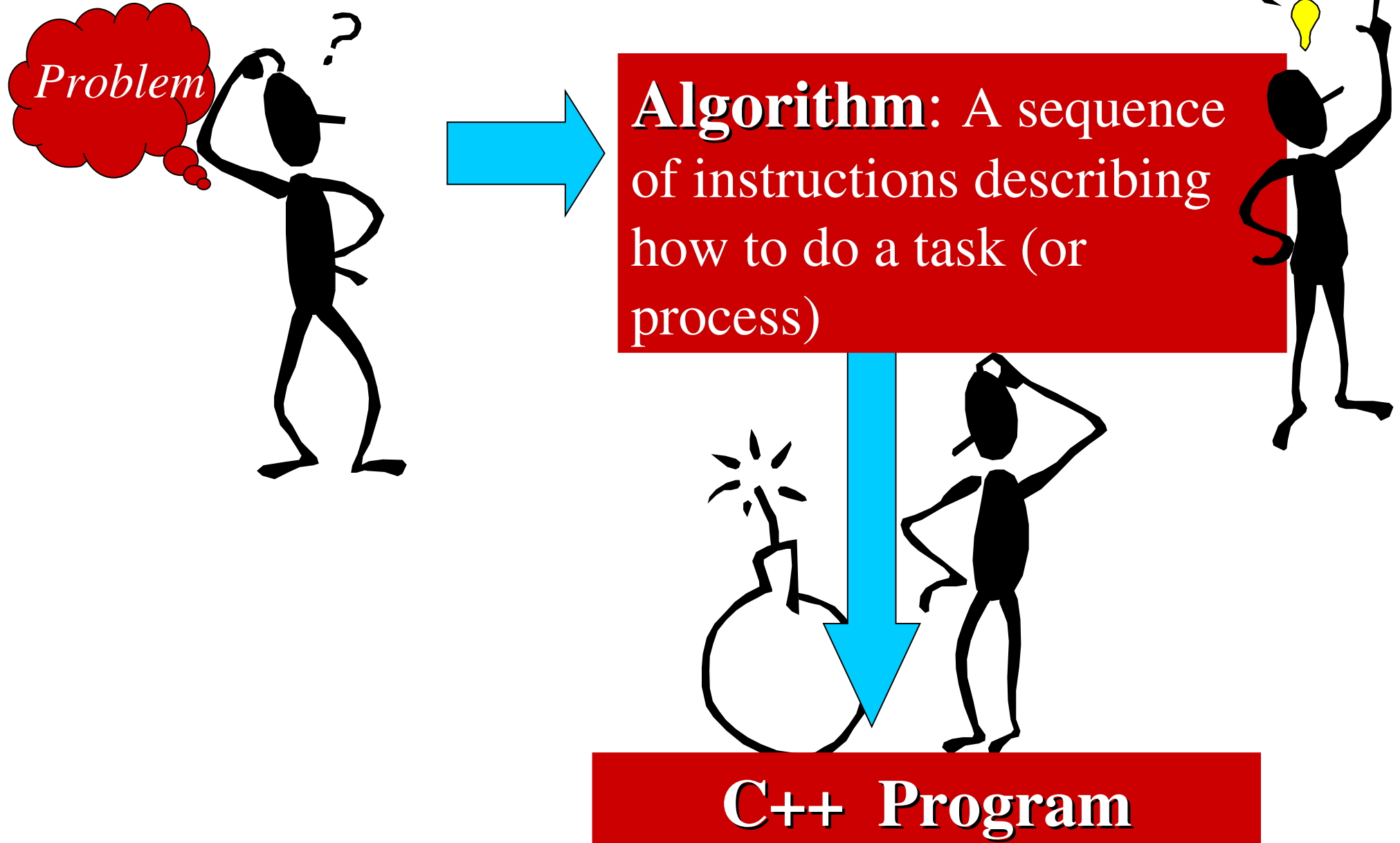
- We need to solve a computational problem
  - “Convert a weight in pounds to Kg”
  
- An algorithm specifies how to solve it, e.g.:
  - 1. Read weight-in-pounds
  - 2. Calculate  $\text{weight-in-Kg} = \text{weight-in-pounds} * 0.455$
  - 3. Print weight-in-Kg
  
- A computer program is a computer-executable description of an algorithm



# The Problem-solving Process



# From Algorithms to Programs



# Algorithms as technology

- Computers may be fast, but they are not infinitely fast.
  - Memory may be cheap, but it is not free.
  - Computing time is therefore a bounded resource, and so is space in memory.
  - These resources should be used wisely, and algorithms that are efficient in terms of time or space will help you do so.
- 
- Efficiency**
  - hardware with high clock rates, pipelining, and superscalar architectures,**
  - easy-to-use, intuitive graphical user interfaces (GUIs),**
  - object-oriented systems, and**
  - local-area and wide-area networking.**

# Evaluation of Algorithm

- Algorithm evaluation is the process of assessing a property or properties of an algorithm.

**Definition 1.1** [Algorithm]: An *algorithm* is a finite set of instructions that, if followed, accomplishes a particular task. In addition, all algorithms must 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.
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 very basic so that it can be carried out, in principle, by a person using only pencil and paper. It is not enough that each operation be definite as in criterion 3; it also must be feasible. □

An algorithm is composed of a finite set of steps, each of which may require one or more operations. The possibility of a computer carrying out these operations necessitates that certain constraints be placed on the type of operations an algorithm can include.

# Evaluation of Algorithm..

- **How to devise Algorithm:-**
- **How to validate Algorithm:-**
- **How to analyze algorithm:-**
- **How to test a program:-**

# Algorithms Design Techniques

The algorithms can be classified in various ways. They are:

- Implementation Method
- Design Method
- Design Approaches
- Other Classifications

# Algorithms Design Techniques

Classification by Implementation Method:

- Recursion or Iteration:

Example: The Tower of Hanoi

- Exact or Approximate: Algorithms that are capable of finding an optimal solution for any problem are known as the exact algorithm. For all those problems, where it is not possible to find the most optimized solution, an approximation algorithm is used. Approximate algorithms are the type of algorithms that find the result as an average outcome of sub outcomes to a problem.

Example: For NP-Hard Problems, approximation algorithms are used. Sorting algorithms are the exact algorithms.

- Serial or Parallel or Distributed Algorithms: In serial algorithms, one instruction is executed at a time while parallel algorithms are those in which we divide the problem into sub problems and execute them on different processors. If parallel algorithms are distributed on different machines, then they are known as distributed algorithms.

# Algorithms Design Techniques

Classification by Design Method:

- Greedy Method: In the greedy method, at each step, a decision is made to choose the local optimum, without thinking about the future consequences.

Example: Fractional Knapsack, Activity Selection.

- Divide and Conquer: The Divide and Conquer strategy involves dividing the problem into sub-problem, recursively solving them, and then recombining them for the final answer.

Example: Merge sort, Quicksort.

- Dynamic Programming: The approach of Dynamic programming is similar to divide and conquer. The difference is that whenever we have recursive function calls with the same result, instead of calling them again we try to store the result in a data structure in the form of a table and retrieve the results from the table. Thus, the overall time complexity is reduced. “Dynamic” means we dynamically decide, whether to call a function or retrieve values from the table.

Example: 0-1 Knapsack, subset-sum problem.



# Algorithms Design Techniques

- Backtracking:** This approach is based on exploring each available option at every stage one-by-one. While exploring an option if a point is reached that doesn't seem to lead to the solution, the program control backtracks one step, and starts exploring the next option. In this way, the program explores all possible course of actions and finds the route that leads to the solution.

Example: N-queen problem, maize problem.

- Branch and Bound:** This technique is very useful in solving combinatorial optimization problem that have multiple solutions and we are interested in find the most optimum solution. In this approach, the entire solution space is represented in the form of a state space tree.

Example: Job sequencing, Travelling salesman problem.

# Algorithms Design Techniques

## **Classification by Design Approaches :**

**Top-Down Approach :**

**Bottom-up approach**

- Top-Down Approach:** In the top-down approach, a large problem is divided into small sub-problem. and keep repeating the process of decomposing problems until the complex problem is solved.
- Bottom-up approach:** The bottom-up approach is also known as the reverse of top-down approaches.

# Algorithms Design Techniques

**Other Classifications:** Apart from classifying the algorithms into the above broad categories, the algorithm can be classified into other broad categories like:

- Randomized Algorithms:** Algorithms that make random choices for faster solutions are known as randomized algorithms.

**Example: Randomized Quicksort Algorithm.**

# Computational Problems

```
graph LR; CP((Computational Problems)) --> C((Concurrent)); CP --> S((Sequential)); CP --> D((Distributed)); CP --> EB((Event Based)); C --- C_desc[Operations overlap in time]; S --- S_desc[Operations are performed in a step-by-step manner]; D --- D_desc[Operations are performed at different locations]; EB --- EB_desc[Operations are performed based on the input];
```

**Concurrent**

Operations overlap in time

**Sequential**

Operations are performed in a step-by-step manner

**Distributed**

Operations are performed at different locations

**Event Based**

Operations are performed based on the input

# Problem Solving Strategies

**1) Divide and conquer:** A divide-and-conquer algorithm works by recursively breaking down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly.

•Eg: Binary search, Merge sort, Quick sort

**2) Greedy method:** The *greedy approach* is an *algorithm* strategy in which a set of resources are recursively divided based on the maximum, immediate availability of that resource at any given stage of execution. To solve a problem based on the *greedy approach*, there are two stages. scanning the list of items. optimization.

•Eg: Knapsack Problem, Job scheduling with deadlines.

# **Problem Solving Strategies**

**3) Dynamic programming:** Dynamic Programming is mainly an optimization over plain recursion. Wherever we see a recursive solution that has repeated calls for same inputs, we can optimize it using Dynamic Programming. The idea is to simply store the results of subproblems, so that we do not have to re-compute them when needed later.

**•Eg: 0/1 Knapsack problem, Optimum Binary Search Tree, Travelling salesman problem**

**4) Trial and error:** Trial and error is a problem solving method in which multiple attempts are made to reach a solution. It is a basic method of learning that essentially all organisms use to learn new behaviors. Trial and error is trying a method, observing if it works, and if it doesn't trying a new method.

**•Eg: Printer not working problem: check ink level or check paper tray jamming**

# Types of problems

- We can **identify the Efficiency** of an algorithm from its **speed** (how **long** does the algorithm take to **produce the result**).
- Trackable
- Intrackable
- Decision
- Optimization

Trackable : Problems that can be solvable in a reasonable (polynomial) time.

Intrackable : Some problems are intractable, as they grow large, we are unable to solve them in reasonable time.

Tractable means that the problems can be solved in theory as well as in practice.

But the problems that can be solved in theory but not in practice are known as intractable.

# Tractability

What constitutes reasonable time?

Standard working definition: polynomial time

an input of size  $n$  the worst-case running time is  $O(n^k)$  for some constant  $k$

$O(n^2)$ ,  $O(n^3)$ ,  $O(1)$ ,  $O(n \lg n)$ ,  $O(2^n)$ ,  $O(n^n)$ ,  $O(n!)$

■ Polynomial time:  $O(n^2)$ ,  $O(n^3)$ ,  $O(1)$ ,  $O(n \lg n)$

■ Not in polynomial time:  $O(2^n)$ ,  $O(n^n)$ ,  $O(n!)$

Are all problems solvable in polynomial time?

No: Turing's “Halting Problem” is not solvable by any computer, no matter how much time is given.



# Optimization/Decision Problems

## ■ Optimization Problems

An optimization problem is one which asks, “What is the optimal solution to problem X?”

Examples:

0-1 Knapsack

Fractional Knapsack

Minimum Spanning Tree

## ■ Decision Problems

An decision problem is one with yes/no answer

Examples: Does a graph  $G$  have a MST of weight  $W$ ?

# P problems

- The P in the P class stands for Polynomial Time. It is the collection of decision problems (problems with a “yes” or “no” answer) that can be solved by a deterministic machine in polynomial time.
- The solution to P problems is easy to find.
- P is often a class of computational problems that are solvable and tractable.
- Fractional Knapsack
- MST
- Sorting
- Others?

# NP problems

NP: the class of decision problems that are solvable in polynomial time on a nondeterministic machine (or with a nondeterministic algorithm)

A nondeterministic computer is one that can “guess” the right answer or solution

Think of a nondeterministic computer as a parallel machine that can freely spawn an infinite number of processes.

Thus NP can also be thought of as the class of problems “whose solutions can be verified in polynomial time”.

Note that NP stands for “Nondeterministic Polynomial-time”

# NP problems

- Traveling Salesman
- Graph Coloring
- Satisfiability (SAT):-the problem of deciding whether a given Boolean formula is satisfiable

# NP-hard

What does NP-hard mean?

A lot of times you can solve a problem by reducing it to a different problem.

I can reduce Problem B to Problem A if, given a solution to Problem A, I can easily construct a solution to Problem B. (In this case, "easily" means "in polynomial time.“).

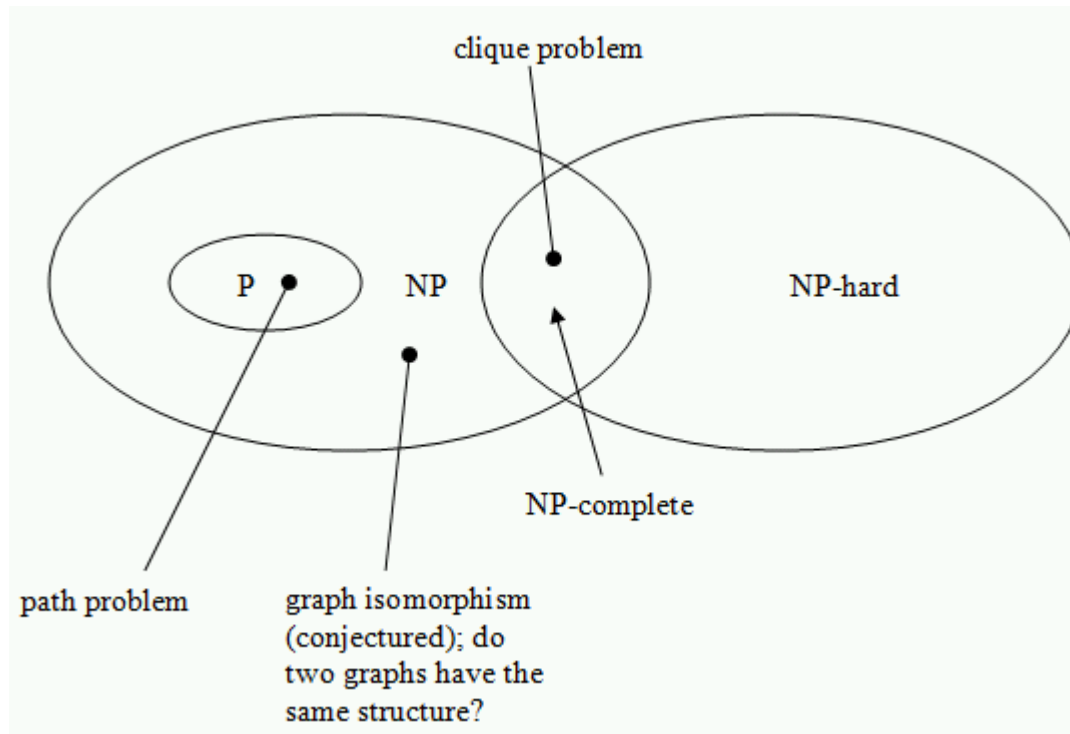
- A problem is NP-hard if all problems in NP are polynomial time reducible to it, ...

Ex:- Hamiltonian Cycle

# NP-Complete

A problem is NP-complete if the problem is both

- NP-hard, and
- NP.



# Components of an Algorithm

- Variables and values
- Instructions
- Sequences
  - A series of instructions
- Procedures
  - A named sequence of instructions
  - we also use the following words to refer to a “Procedure” :
    - Sub-routine
    - Module
    - Function

# Components of an Algorithm Cont.

## ■ Selections

- An instruction that decides which of two possible sequences is executed
- The decision is based on true/false condition

## ■ Repetitions

- Also known as iteration or loop

## ■ Documentation

- Records what the algorithm does



# Classification of Time Complexities

- Constant Time Complexity:  $O(1)$  ...
- Linear Time Complexity:  $O(n)$  ...
- Logarithmic Time Complexity:  $O(\log n)$  ...
- Quadratic Time Complexity:  $O(n^2)$  ...
- Exponential Time Complexity:  $O(2^n)$

# Asymptotic Notation ( $O$ $\Omega$ $\Theta$ )

[Big “oh” ]  $O$  : This notation is used to express an upper bound on computing time of an algorithm. When we say that time complexity of Selection sort algorithm is  $O(n^2)$  , means that for sufficiently large values of ‘n’ , computation time will not exceed some constant time \*  $n^2$  i.e proportional to  $n^2$  (Worst Case).

## [Omega] $\Omega$

This notation is used to express a lower bound on computing time of an algorithm. When we say that best case time complexity of insertion sort is  $\Omega(n)$ , means that for sufficiently large values of 'n', minimum computation time will be some constant time \* n i.e proportional to n.

# [Theta] $\Theta$

- This notation is used to express time complexity of an algorithm when it is same for worst & Best cases. For example best and worst case time complexities for selection sort is  $O(n^2)$  &  $\Omega(n^2)$  i.e. it can be expressed as  $\Theta(n^2)$ .
- Definition : 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  $n$ ,  $n \geq n_0$ .

# What does an algorithm ?

- An algorithm is described by:
  - Input data
  - Output data
  - *Preconditions*: specifies restrictions on input data
  - *Postconditions*: specifies what is the result
- Example: Binary Search
  - Input data: a:array of integer; x:integer;
  - Output data: found: boolean;
  - Precondition: a is sorted in ascending order
  - Postcondition: found is true if x is in a, and found is false otherwise

# Correct algorithms

**An algorithm is correct if:**

- for **any correct** input data:
  - it **stops** and
  - it produces **correct** output.
- Correct input data: satisfies precondition
- Correct output data: satisfies postcondition

# Proving correctness

An algorithm = a list of actions

• *Proving that an algorithm is totally correct:*

1. *Proving that it will terminate*

2. *Proving that the list of actions applied to the precondition imply the postcondition*

- This is easy to prove for simple sequential algorithms
- This can be complicated to prove for repetitive algorithms (containing loops or recursivity)
  - use techniques based on *loop invariants* and *induction*

# Example – a sequential algorithm

Swap1 (x, y) :

    aux := x

    x := y

    y := aux

**Precondition:**

    x = a and y = b

**Postcondition:**

    x = b and y = a

**Proof:** *the list of actions applied to the precondition imply the postcondition*

Precondition:      x = a and  
                    y = b

aux := x  $\Rightarrow$  aux = a

x := y  $\Rightarrow$  x = b

y := aux  $\Rightarrow$  y = a

x = b and y = a is the  
Postcondition



# Example – a repetitive algorithm

Algorithm

Sum\_of\_N\_numbers

Input: a, an array of N  
numbers

Output: s, the sum of the N  
numbers in a

s:=0;

k:=0;

While (k<N) do

    k:=k+1;

    s:=s+a[k];

end

*Proof: the list of actions  
applied to the  
precondition imply the  
postcondition*

*BUT: we cannot enumerate  
all the actions in case  
of a repetitive  
algorithm !*

*We use techniques based  
on loop invariants and  
induction*

# Loop invariants

- A loop invariant is a logical predicate such that: if it is satisfied before entering any single iteration of the loop then it is also satisfied after the iteration

# Example: Loop invariant for Sum of n numbers

Algorithm Sum\_of\_N\_numbers

Input:  $a$ , an array of  $N$  numbers

Output:  $s$ , the sum of the  $N$  numbers in  $a$

```
s:=0;  
k:=0;  
While (k<N) do  
    k:=k+1;  
    s:=s+a[k];  
end
```

Loop invariant = induction  
hypothesis: At step  $k$ ,  $S$  holds the  
sum of the first  $k$  numbers

# Using loop invariants in proofs

**We must show the following 3 things about a loop invariant:**

**1.Initialization:** It is true prior to the first iteration of the loop.

**2.Maintenance:** If it is true before an iteration of the loop, it remains true before the next iteration.

**3.Termination:** When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

## Example: Proving the correctness of the Sum algorithm (1)

- Induction hypothesis:  $S =$  sum of the first  $k$  numbers

1. *Initialization: The hypothesis is true at the beginning of the loop:*

Before the first iteration:  $k=0$ ,  $S=0$ . The first 0 numbers have sum zero (there are no numbers)  
 $\Rightarrow$  hypothesis true before entering the loop

## Example: Proving the correctness of the Sum algorithm (2)

- Induction hypothesis:  $S =$  sum of the first  $k$  numbers
- 2. *Maintenance: If hypothesis is true before step  $k$ , then it will be true before step  $k+1$  (immediately after step  $k$  is finished)*  
We assume that it is true at beginning of step  $k$ : “ $S$  is the sum of the first  $k$  numbers”  
We have to prove that after executing step  $k$ , at the beginning of step  $k+1$ : “ $S$  is the sum of the first  $k+1$  numbers”  
We calculate the value of  $S$  at the end of this step  
 $K := k+1, s := s + a[k+1] \Rightarrow s$  is the sum of the first  $k+1$  numbers

## Example: Proving the correctness of the Sum algorithm (3)

- Induction hypothesis:  $S = \text{sum of the first } k \text{ numbers}$
- 3. *Termination: When the loop terminates, the hypothesis implies the correctness of the algorithm*

The loop terminates when  $k=n \Rightarrow s = \text{sum of first } k=n \text{ numbers} \Rightarrow \text{postcondition of algorithm, DONE}$

# Loop invariants and induction

- **Proving loop invariants is similar to mathematical induction:**
  - showing that the invariant holds before the first iteration corresponds to the **base case**, and
  - showing that the invariant holds from iteration to iteration corresponds to the **inductive step**.



# Mathematical induction - Review

- Let  $T$  be a theorem that we want to prove.  $T$  includes a natural parameter  $n$ .
- Proving that  $T$  holds for all natural values of  $n$  is done by proving following two conditions:
  1.  $T$  holds for  $n=1$
  2. For every  $n > 1$  if  $T$  holds for  $n-1$ , then  $T$  holds for  $n$

## Terminology:

$T$  = *Induction Hypothesis*

1 = *Base case*

2 = *Inductive step*

# Mathematical induction - Review

- **Strong Induction:** a variant of induction where the inductive step builds up on all the smaller values
- Proving that  $T$  holds for all natural values of  $n$  is done by proving following two conditions:
  1.  $T$  holds for  $n=1$
  2. For every  $n > 1$  if  $T$  holds for all  $k \leq n-1$ , then  $T$  holds for  $n$

# Mathematical induction review – Example1

Theorem: *The sum of the first  $n$  natural numbers is  $n*(n+1)/2$*

Proof: by *induction* on  $n$

1. **Base case:** If  $n=1$ ,  $s(1)=1=1*(1+1)/2$
2. **Inductive step:** We assume that  $s(n)=n*(n+1)/2$ , and prove that this implies  $s(n+1)=(n+1)*(n+2)/2$ , for all  $n \geq 1$

$$s(n+1)=s(n)+(n+1)=n*(n+1)/2+(n+1)=(n+1)*(n+2)/2$$

# Correctness of algorithms

- Induction can be used for proving the correctness of repetitive algorithms:
  - Iterative algorithms:
    1. Loop invariants
      - Induction hypothesis = loop invariant = relationships between the variables during loop execution
  - Recursive algorithms
    1. Direct induction
      - Hypothesis = a recursive call itself ; often a case for applying *strong* induction

# Example: Correctness proof for Decimal to Binary Conversion

Algorithm Decimal\_to\_Binary

Input:  $n$ , a positive integer

Output:  $b$ , an array of bits, the bin repr. of  $n$ ,  
starting with the least significant bits

```
t:=n;  
k:=0;  
While (t>0) do  
    k:=k+1;  
    b[k]:=t mod 2;  
    t:=t div 2;  
end
```

It is a repetitive (iterative) algorithm, thus we use loop invariants and proof by induction

# Example: Loop invariant for Decimal to Binary Conversion

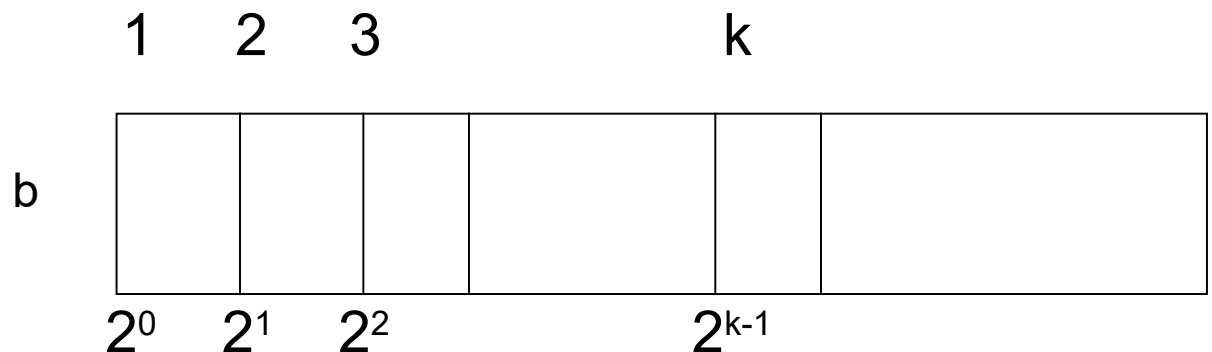
Algorithm `Decimal_to_Binary`

Input:  $n$ , a positive integer

Output:  $b$ , an array of bits, the bin repr. of  $n$

```
t:=n;  
k:=0;  
While (t>0) do  
    k:=k+1;  
    b[k]:=t mod 2;  
    t:=t div 2;  
end
```

At step  $k$ ,  $b$  holds the  $k$  least significant bits of  $n$ , and the value of  $t$ , when shifted by  $k$ , corresponds to the rest of the bits



# Example: Loop invariant for Decimal to Binary Conversion

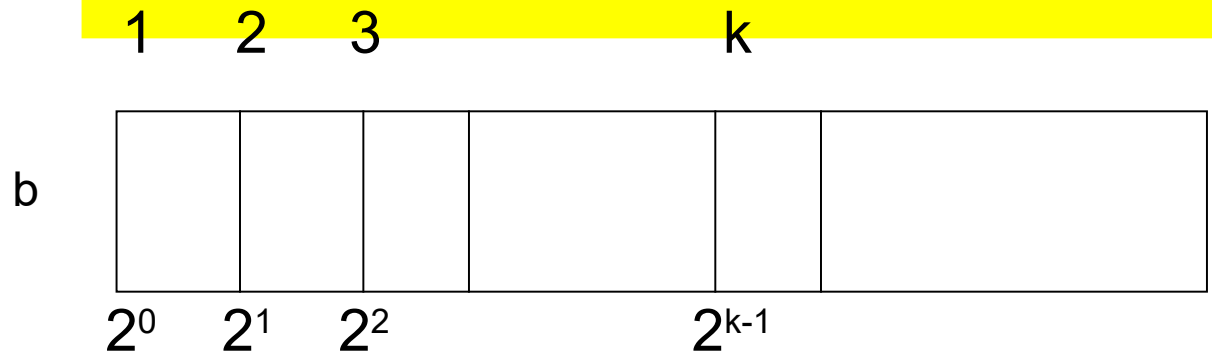
Algorithm Decimal\_to\_Binary

Input:  $n$ , a positive integer

Output:  $b$ , an array of bits, the bin repr. of  $n$

```
t:=n;  
k:=0;  
While (t>0) do  
    k:=k+1;  
    b[k]:=t mod 2;  
    t:=t div 2;  
end
```

**Loop invariant:** If  $m$  is the integer represented by array  $b[1..k]$ , then  $n = t \cdot 2^k + m$



# Example: Proving the correctness of the conversion algorithm

- **Induction hypothesis=Loop Invariant:** If  $m$  is the integer represented by array  $b[1..k]$ , then  $n = t \cdot 2^k + m$
- **To prove the correctness of the algorithm, we have to prove the 3 conditions:**
  1. *Initialization: The hypothesis is true at the beginning of the loop*
  2. *Maintenance: If hypothesis is true for step  $k$ , then it will be true for step  $k+1$*
  3. *Termination: When the loop terminates, the hypothesis implies the correctness of the algorithm*



## Example: Proving the correctness of the conversion algorithm (1)

- Induction hypothesis: If  $m$  is the integer represented by array  $b[1..k]$ , then  $n = t * 2^k + m$
- 1. *The hypothesis is true at the beginning of the loop:*  
 $k=0$ ,  $t=n$ ,  $m=0$  (array is empty)  
 $n = n * 2^0 + 0$

## Example: Proving the correctness of the conversion algorithm (2)

Induction hypothesis: If  $m$  is the integer represented by array  $b[1..k]$ , then  $n = t * 2^k + m$

*2. If hypothesis is true for step  $k$ , then it will be true for step  $k+1$*

At the start of step  $k$ : assume that  $n = t * 2^k + m$ , calculate the values at the end of this step

If  $t = \text{even}$  then:  $t \bmod 2 == 0$ ,  $m$  unchanged,  $t = t / 2$ ,  $k = k + 1 \Rightarrow (t / 2) * 2^{(k+1)} + m = t * 2^k + m = n$

If  $t = \text{odd}$  then:  $t \bmod 2 == 1$ ,  $b[k+1]$  is set to 1,  $m = m + 2^k$ ,  $t = (t - 1) / 2$ ,  $k = k + 1 \Rightarrow (t - 1) / 2 * 2^{(k+1)} + m + 2^k = t * 2^k + m = n$

## Example: Proving the correctness of the conversion algorithm (3)

- Induction hypothesis: If  $m$  is the integer represented by array  $b[1..k]$ , then  $n = t * 2^k + m$

3. *When the loop terminates, the hypothesis implies the correctness of the algorithm*

The loop terminates when  $t=0 \Rightarrow n = 0 * 2^k + m = m$   
 $n == m$ , proved

# Proof of Correctness for Recursive Algorithms

In order to prove recursive algorithms, we have to:

1. Prove the partial correctness (the fact that the program behaves correctly)
  - *we assume that all recursive calls with arguments that satisfy the preconditions behave as described by the specification, and use it to show that the algorithm behaves as specified*
2. Prove that the program terminates
  - any chain of recursive calls eventually ends and all loops, if any, terminate after some finite number of iterations.

# Example - Merge Sort

**MERGE-SORT** (**A**, **p**, **r**)

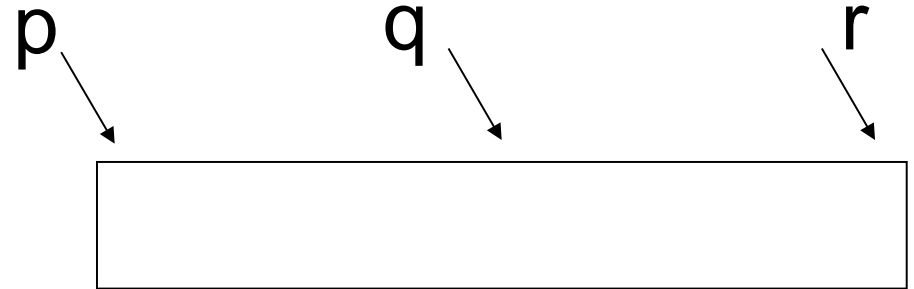
**if**  $p < r$

$q = (p+r) / 2$

**MERGE-SORT** (**A**, **p**, **q**)

**MERGE-SORT** (**A**, **q+1**, **r**)

**MERGE** (**A**, **p**, **q**, **r**)



## **Precondition:**

Array A has at least 1 element between indexes p and r ( $p \leq r$ )

## **Postcondition:**

The elements between indexes p and r are sorted

# Example - Merge Sort

- **MERGE-SORT** calls a function **MERGE**(A,p,q,r) to merge the sorted subarrays of A into a single sorted one
- The proof of **MERGE** (which is an iterative function) can be done separately, using loop invariants
- We assume here that **MERGE** has been proved to fulfill its postconditions (can do it as a distinct exercise)

**MERGE** (A,p,q,r)

**Precondition:** A is an array and p, q, and r are indices into the array such that  $p \leq q < r$ . The subarrays A[p.. q] and A[q +1.. r] are sorted

**Postcondition:** The subarray A[p..r] is sorted



# Correctness proof for Merge-Sort

- Number of elements to be sorted:  $n=r-p+1$
- **Base Case:**  $n = 1$ 
  - A contains a single element (which is trivially “sorted”)
- **Inductive Hypothesis:**
  - Assume that MergeSort correctly sorts  $n=1, 2, \dots, k$  elements
- **Inductive Step:**
  - Show that MergeSort correctly sorts  $n = k + 1$  elements.
  - First recursive call  $n_1=q-p+1=(k+1)/2 \leq k \Rightarrow$  subarray  $A[p \dots q]$  is sorted
  - Second recursive call  $n_2=r-q=(k+1)/2 \leq k \Rightarrow$  subarray  $A[q+1 \dots r]$  is sorted
  - $A, p, q, r$  fulfill now the precondition of Merge
  - The postcondition of Merge guarantees that the array  $A[p \dots r]$  is sorted  $\Rightarrow$  postcondition of MergeSort

# Correctness proofs for recursive algorithms

**RECURSIVE(n) is**

**if** (n=small\_value)

return ct

else

RECURSIVE(n<sub>1</sub>)

...

RECURSIVE(n<sub>r</sub>)

some\_code

*n<sub>1</sub>, n<sub>2</sub>, ... n<sub>r</sub> are some  
values smaller than n but  
bigger than small\_value*

- **Base Case:** Prove that RECURSIVE works for  $n = \text{small\_value}$
- **Inductive Hypothesis:**
  - Assume that RECURSIVE works correctly for  $n = \text{small\_value}, \dots, k$
- **Inductive Step:**
  - Show that RECURSIVE works correctly for  $n = k + 1$



# Summary

- Proving that an algorithm is totally correct means:
  1. Proving that it will *terminate*
  2. Proving that the list of *actions* applied to the *precondition* imply the *postcondition*
- How to prove *repetitive algorithms*:
  - *Iterative* algorithms: use *Loop invariants*, Induction
  - *Recursive* algorithms: use induction using as hypothesis the recursive call

# Tower of Hanoi

There are three towers

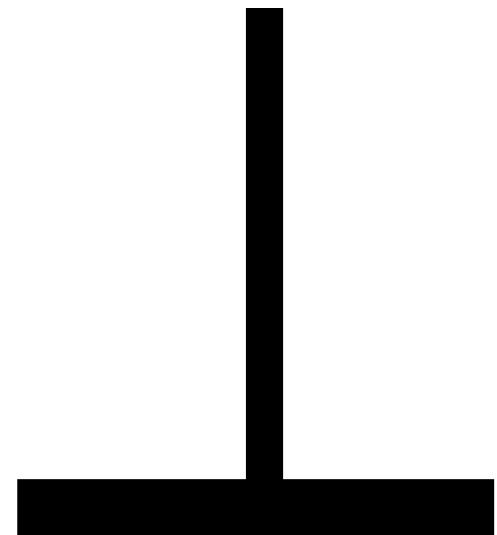
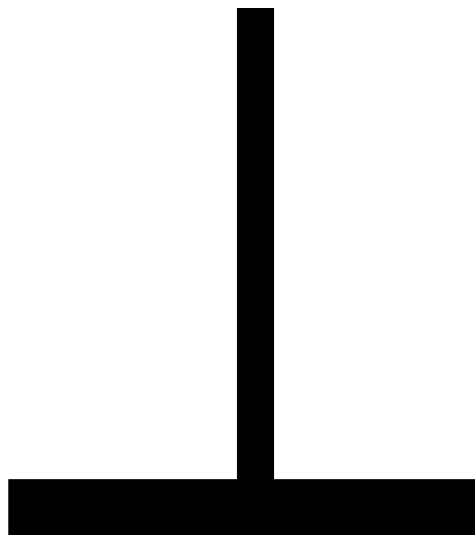
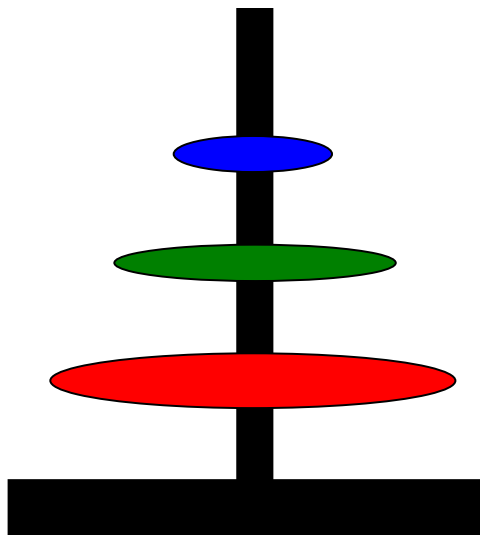
64 gold disks, with decreasing sizes, placed on the first tower

You need to move all of the disks from the first tower to the last tower

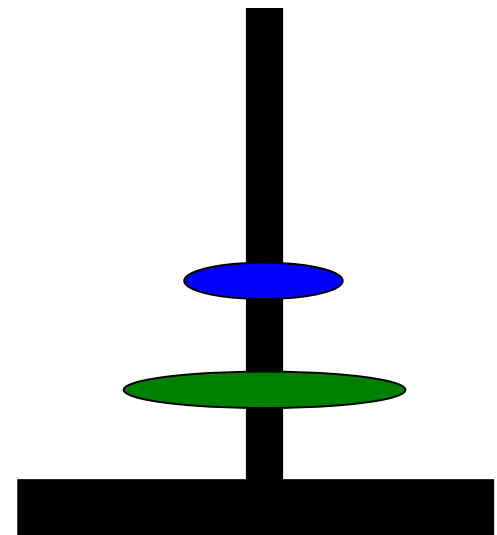
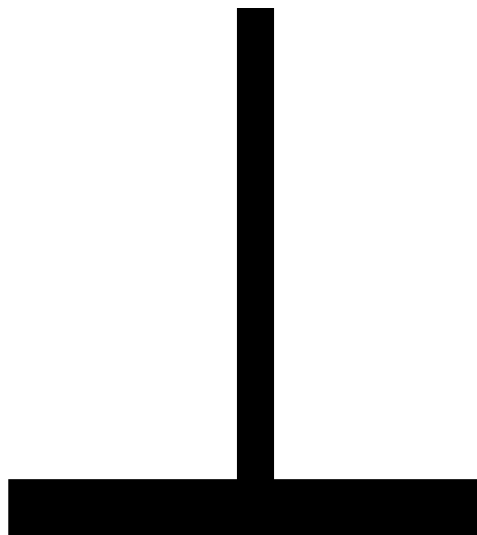
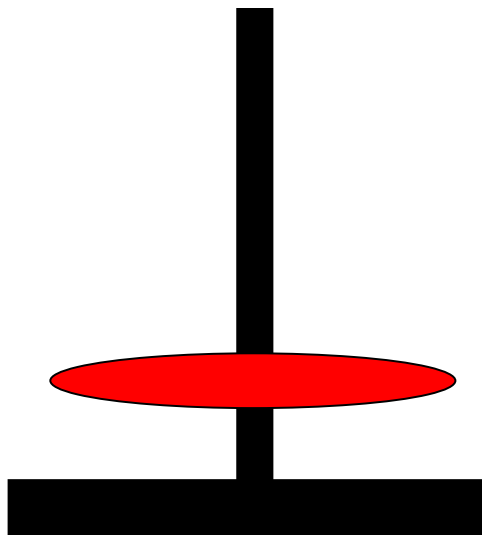
Larger disks can not be placed on top of smaller disks

The third tower can be used to temporarily hold disks

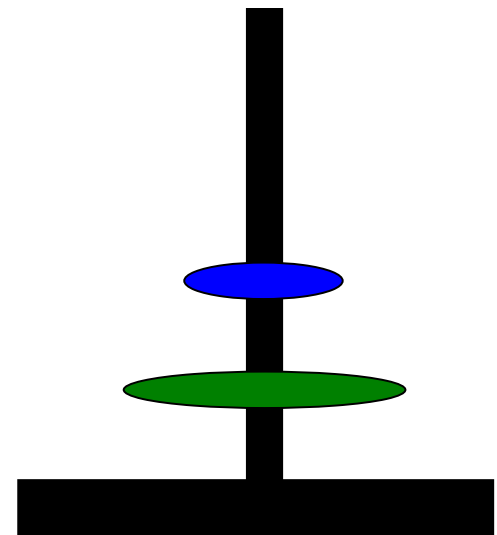
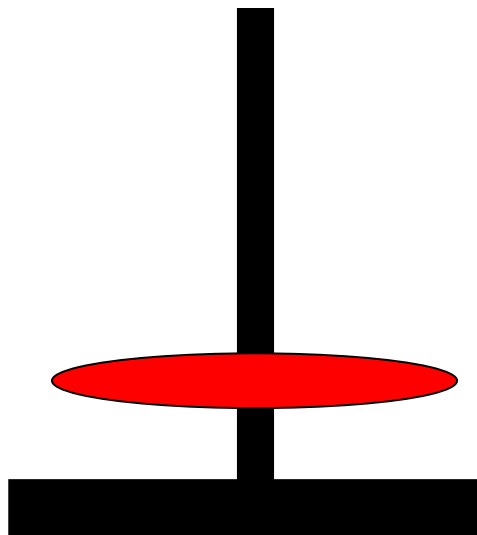
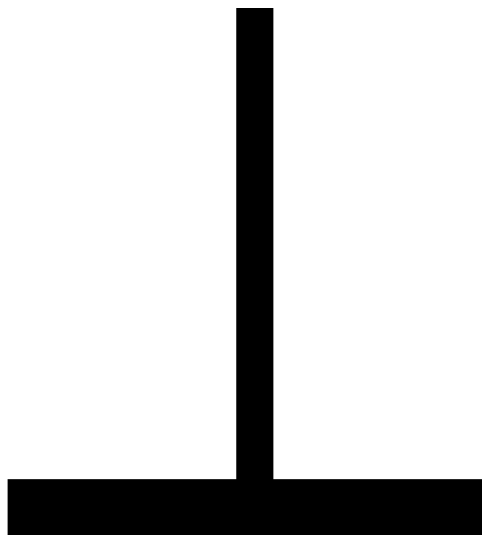
# Recursive Solution



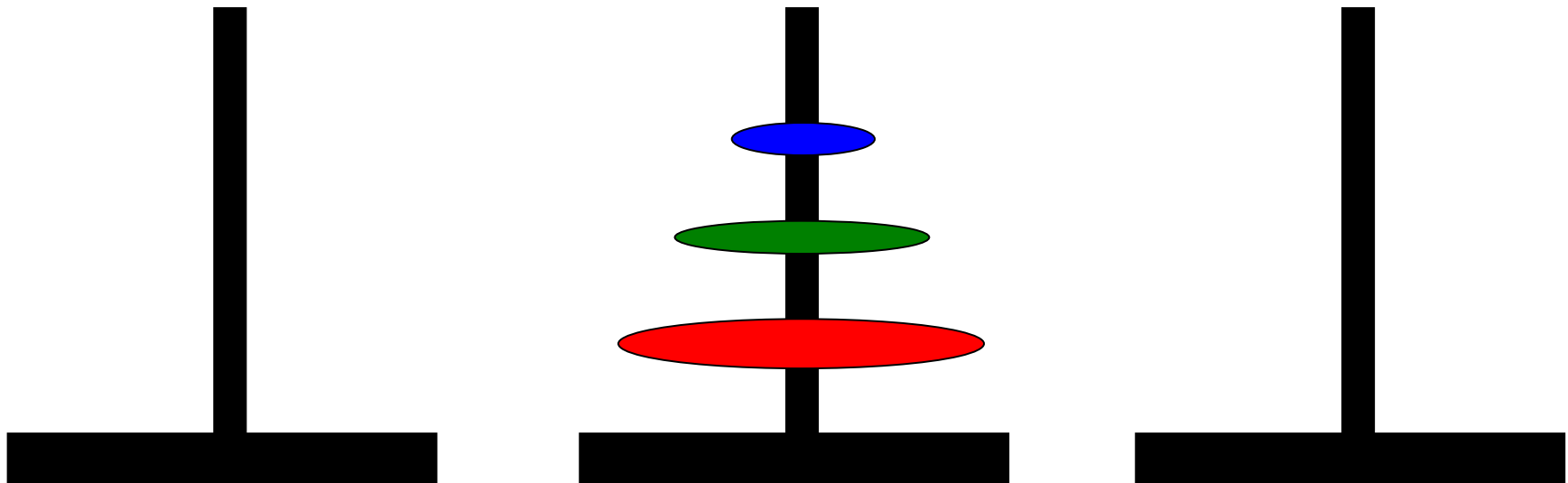
# Recursive Solution



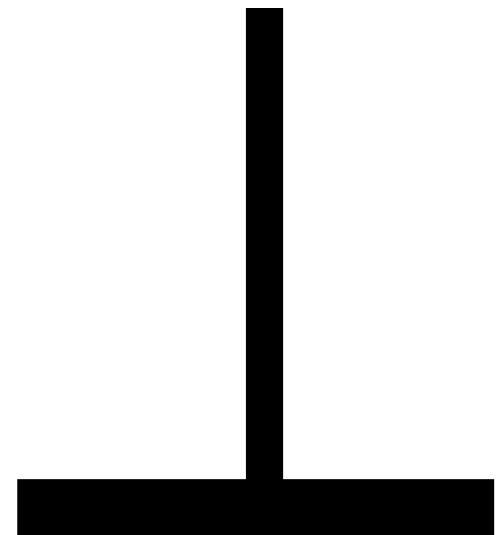
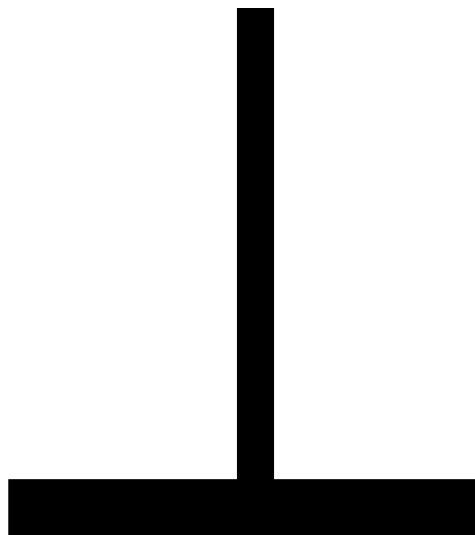
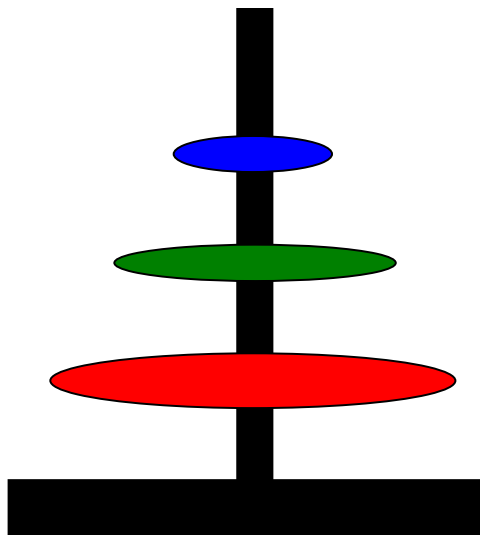
# Recursive Solution



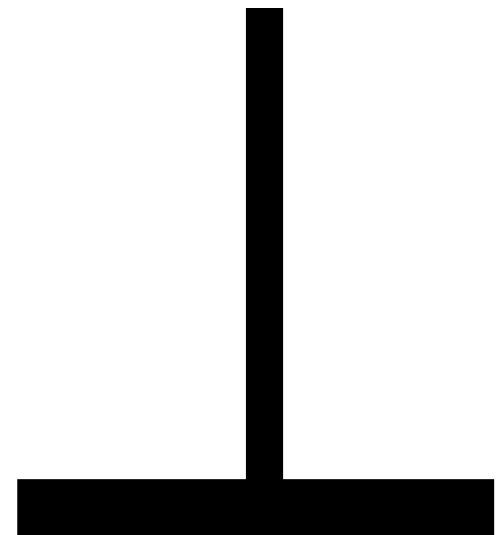
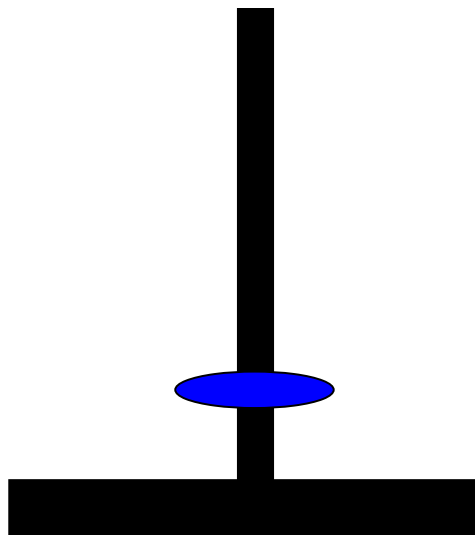
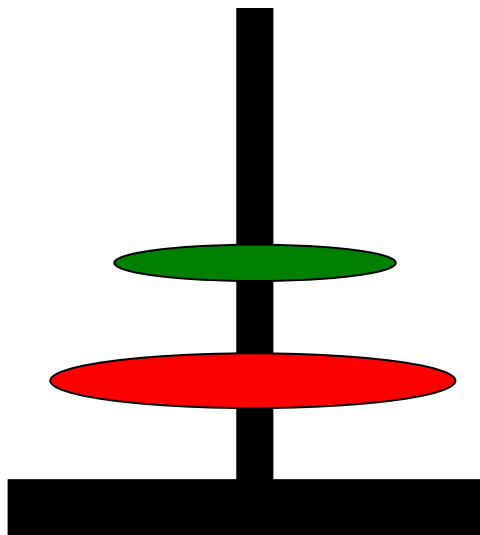
# Recursive Solution



# Tower of Hanoi

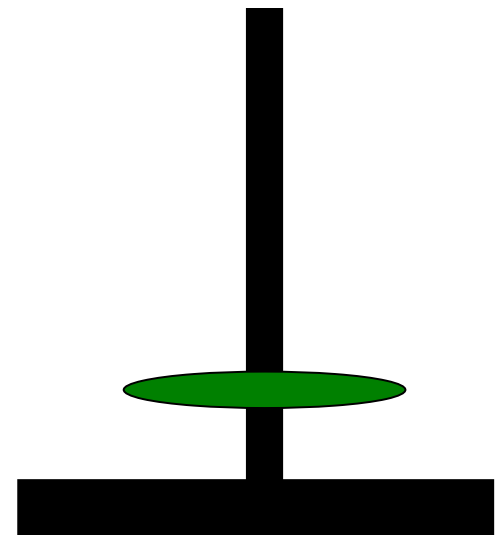
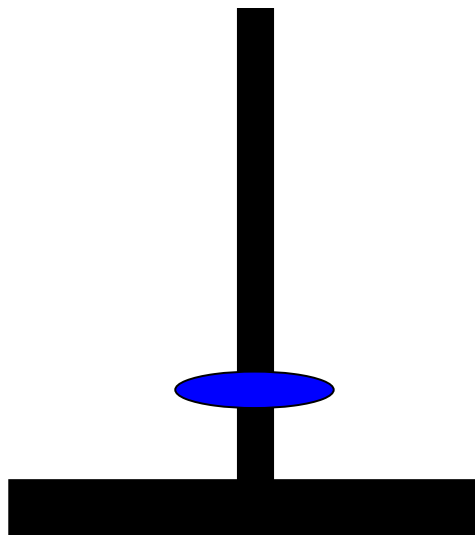
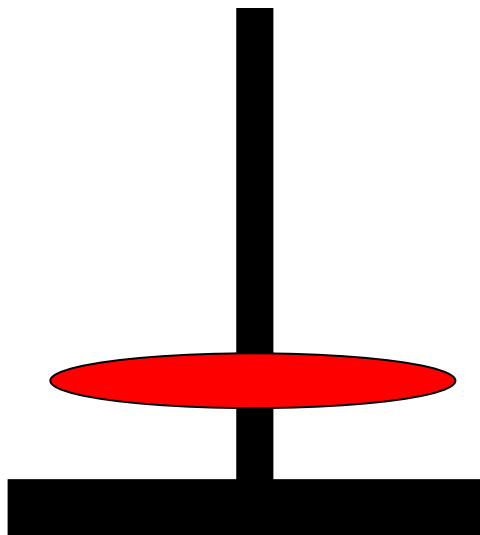


# Tower of Hanoi

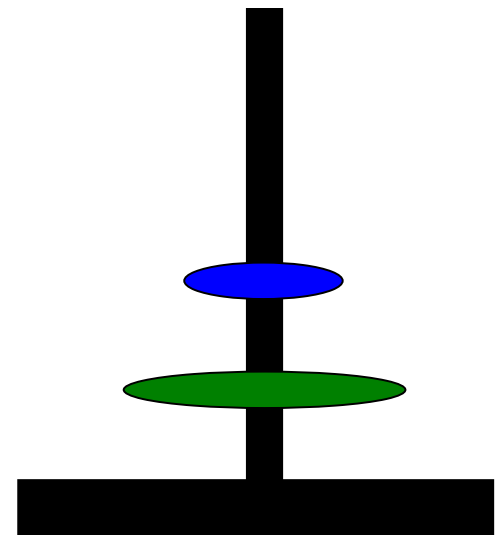
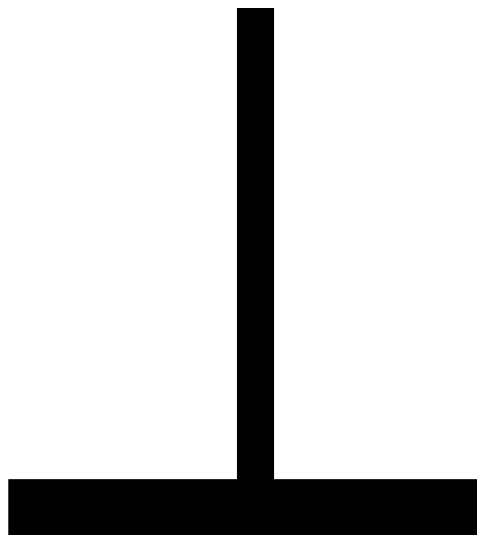
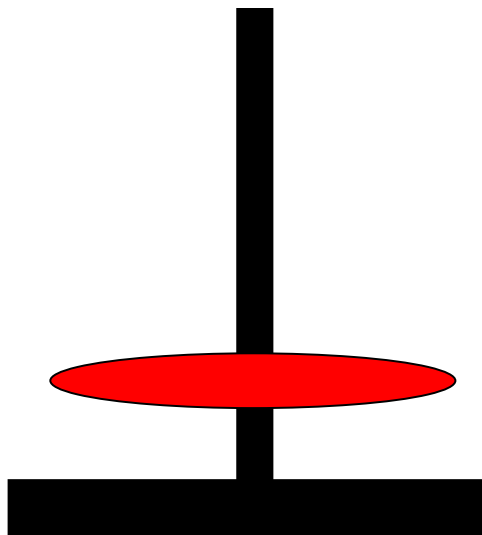




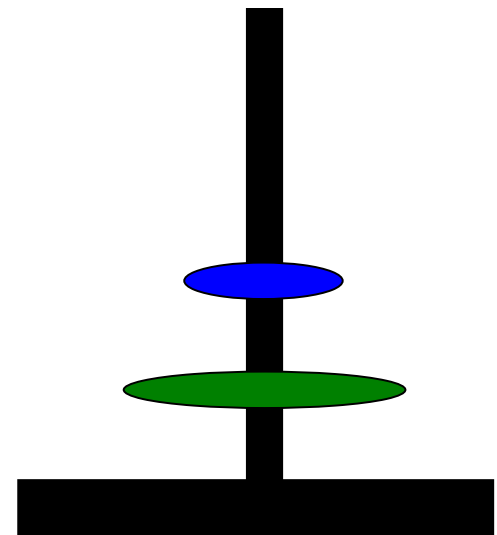
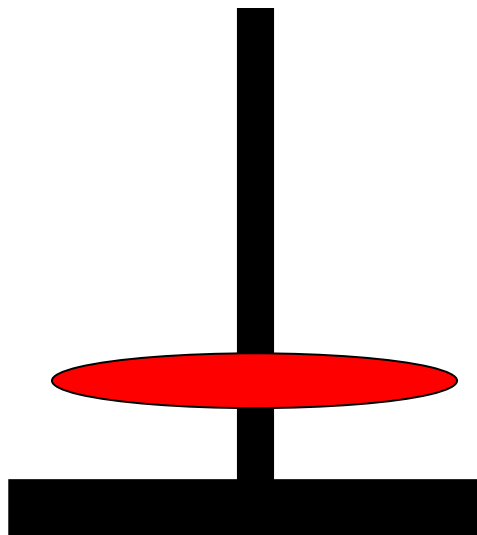
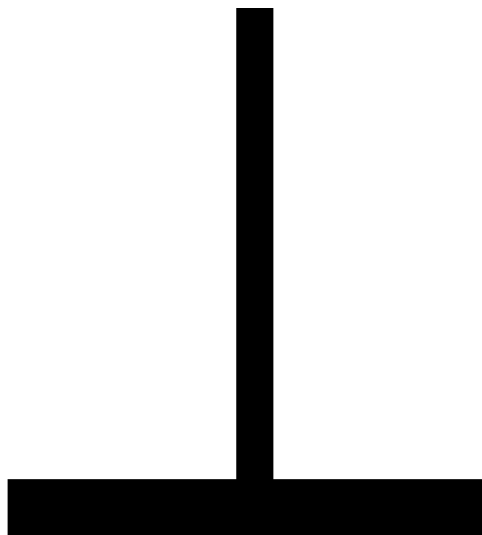
# Tower of Hanoi



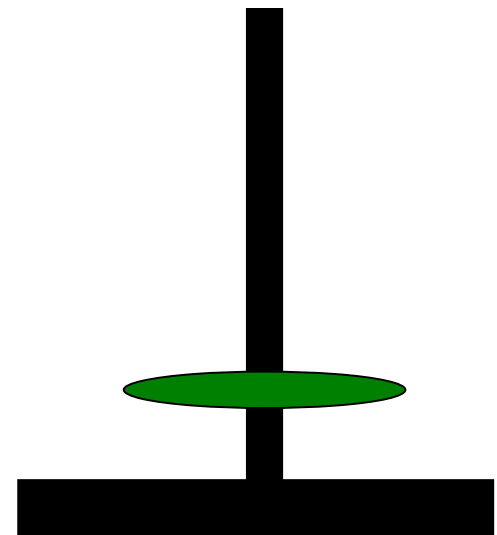
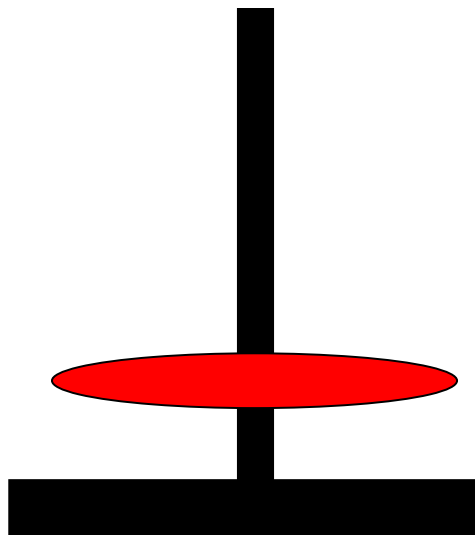
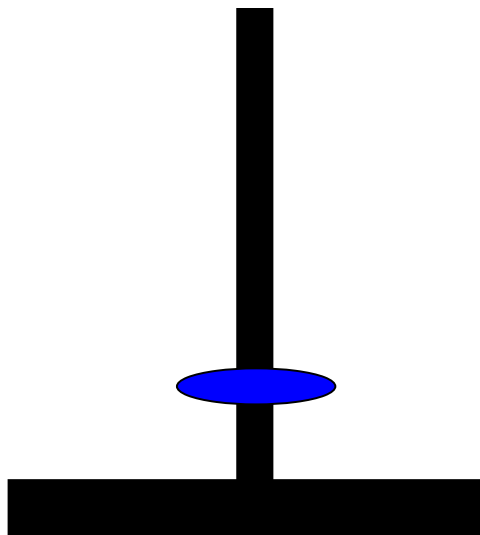
# Tower of Hanoi



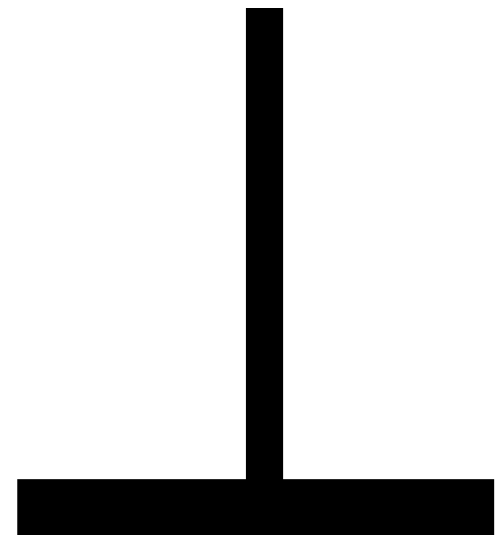
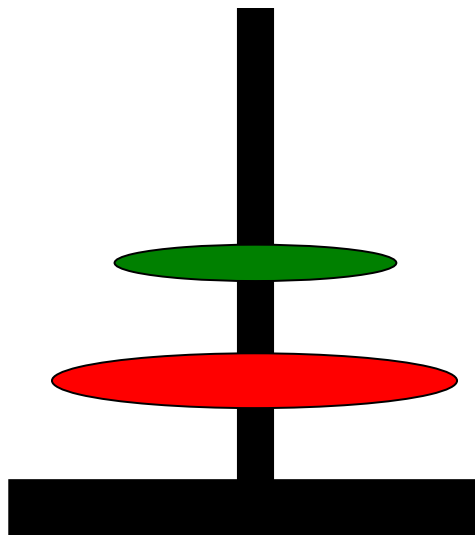
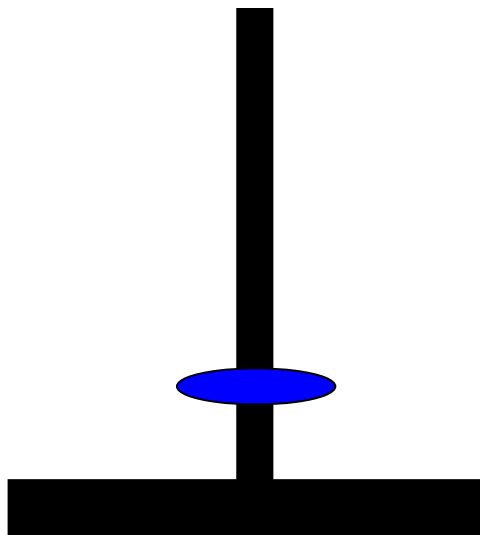
# Tower of Hanoi



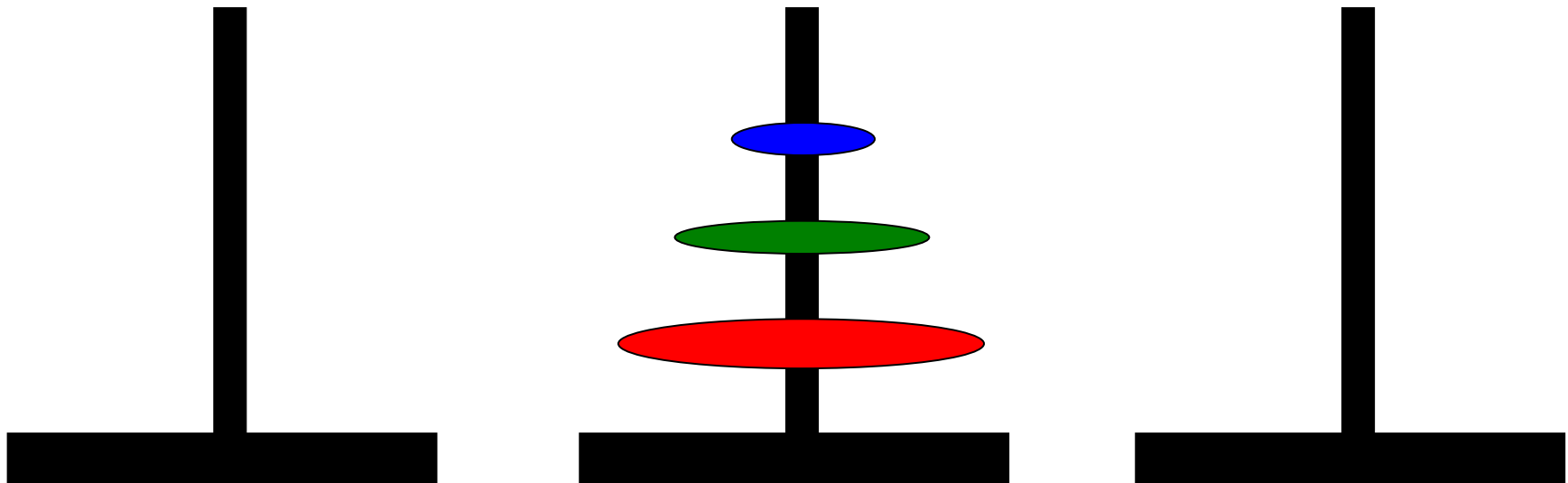
# Tower of Hanoi



# Tower of Hanoi



# Tower of Hanoi



# Recursive Algorithm

```
void Hanoi(int n, string a, string b, string c)
{
    if (n == 1) /* base case */
        Move(a,b);
    else { /* recursion */
        Hanoi(n-1,a,c,b);
        Move(a,b);
        Hanoi(n-1,c,b,a);
    }
}
```

## Complexity Analysis of Tower Of Hanoi

- Moving  $n-1$  disks from source to aux means the first peg to the second peg (in our case). This can be done in  $T(n-1)$  steps.
- Moving the  $n$ th disk from source to dest means a larger disk from the first peg to the third peg will require 1 step.
- Moving  $n-1$  disks from aux to dest means the second peg to the third peg (in our example) will require again  $T(n-1)$  step.

So, total time taken  $T(n) = T(n-1) + 1 + T(n-1)$

Our Equation will be

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



# Guess and Prove

Calculate  $M(n)$  for small  $n$   
and look for a pattern.

Guess the result and prove  
your guess correct using  
induction.

$n$	$M(n)$
1	1
2	3
3	7
4	15
5	31

# Substitution Method

Unwind recurrence, by repeatedly replacing  $M(n)$  by the r.h.s. of the recurrence until the base case is encountered.

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

$$= 2*[2*M(n-2)+1] + 1 = 2^2 * M(n-2) + 1+2$$

$$= 2^2 * [2*M(n-3)+1] + 1 + 2$$

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

# Geometric Series

After k steps

$$M(n) = 2^k * M(n-k) + 1+2 + 2^2 + \dots + 2^{n-k-1}$$

Base case encountered when  $k = n-1$

$$M(n) = 2^{n-1} * M(1) + 1+2 + 2^2 + \dots + 2^{n-2}$$

$$= 1 + 2 + \dots + 2^{n-1} = \sum_{i=0}^{n-1} 2^i$$

Use induction to reprove result for  $M(n)$  using this sum. Generalize by replacing 2 by x.

**THANK  
YOU**