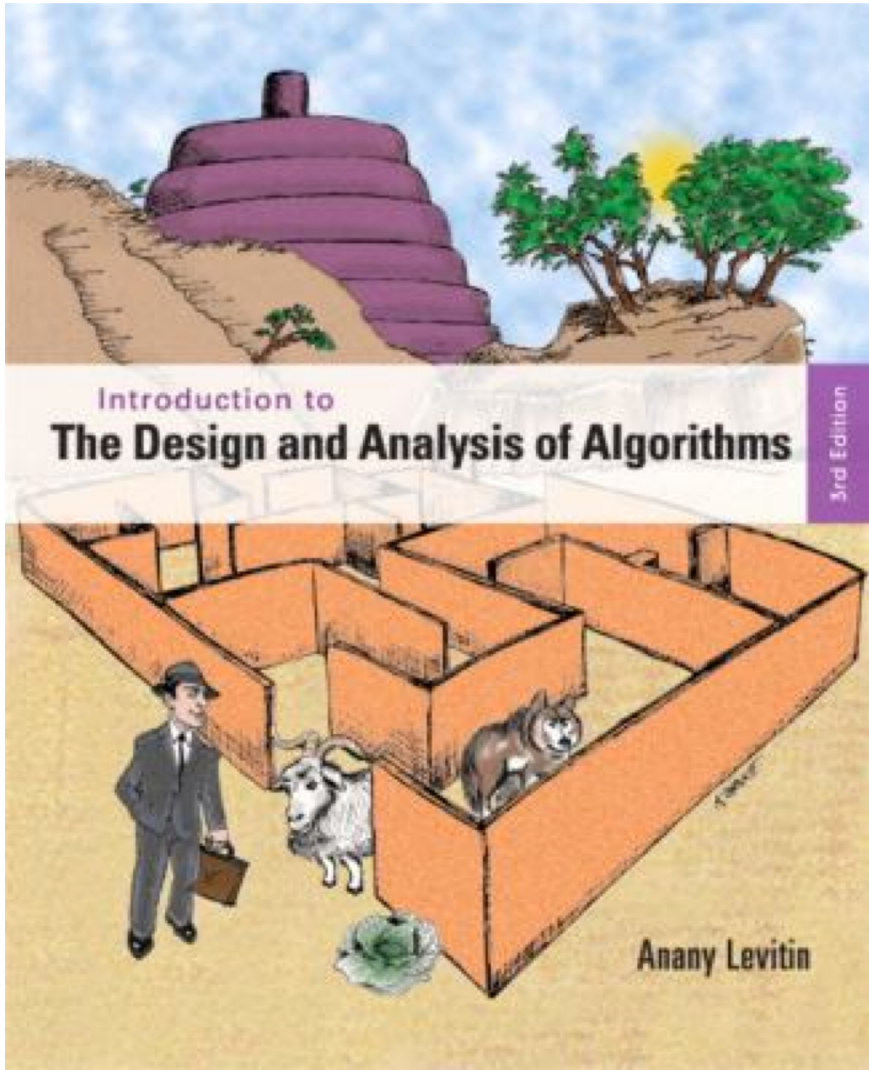
Chapter 2

Fundamentals of the Analysis of Algorithm Efficiency



Fundamentals of the Analysis of Algorithms

- Analysis Framework
- Asymptotic Analysis
- Basic Efficiency Classes
- Time Efficiency of Nonrecursive Algorithms
- Time Efficiency of Recursive Algorithms
- Calculating Fibonacci Numbers

Analysis Framework

- An *algorithm* is a finite set of precise instructions for performing a computation or for solving a problem.

- What is the goal of analysis of algorithms?
  - To compare algorithms mainly in terms of running time but also in terms of other factors (e.g., memory requirements, programmer's effort etc.)

Analysis Framework: Types of Analysis

- Worst case
  - Provides an ***upper bound*** on running time
  - An absolute guarantee that the algorithm would not run longer, no matter what the inputs are
- Best case
  - Provides a ***lower bound*** on running time
  - Input is the one for which the algorithm runs the fastest

$$Lower\ Bound \leq Running\ Time \leq Upper\ Bound$$

- Average case
  - Provides a ***prediction*** about the running time
  - Assumes that the input is random

Empirical analysis of time efficiency

- Select a specific (typical) sample of inputs
- Use physical unit of time (e.g., milliseconds)

  or

  Count actual number of basic operation's executions
- Analyze the empirical data

How to analyze algorithms?

- We need to define a number of objective measures.

(1) Compare execution times?

Not good: times are specific to a particular computer !!

(2) Count the number of statements executed?

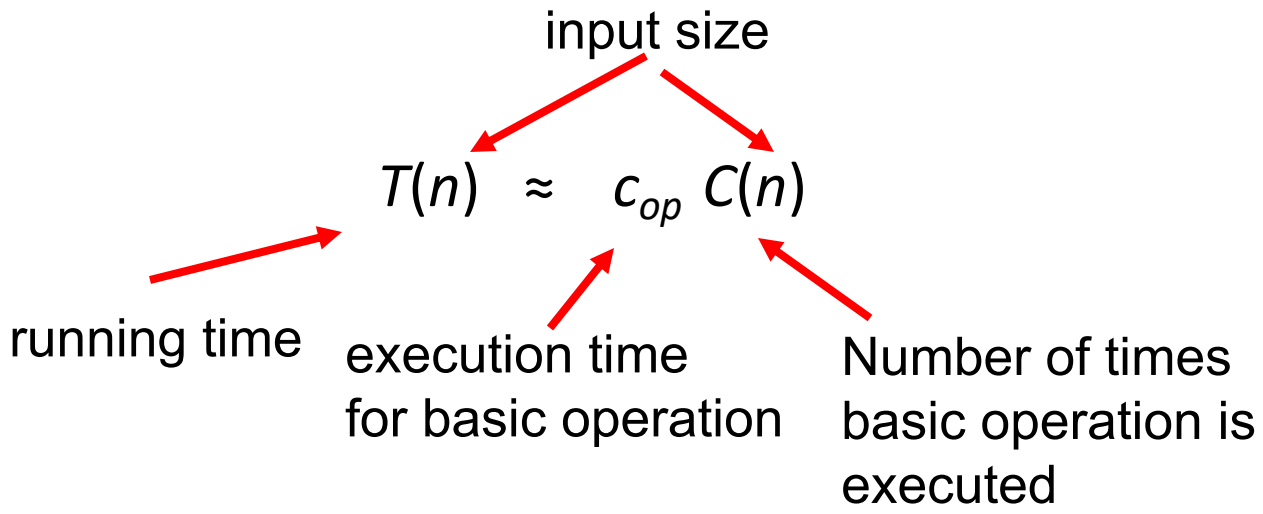
Not good: number of statements vary with the programming language as well as the style of the individual programmer.

Ideal Solution

- Express running time as a function of the input size n (i.e., $f(n)$).
- Compare different functions corresponding to running times.
- Such an analysis is independent of machine time, programming style, etc.

Theoretical analysis of time efficiency

- Time efficiency is analyzed by determining the number of repetitions of the *basic operation* as a function of *input size*
- *Basic operation*: the operation that contributes most towards the running time of the algorithm



Input Size & Basic operation

- What are an input's size and basic operation?
  - Searching for key in a list of n items
  - Multiplication of two matrices
  - Checking primality of a given integer n
  - Typical graph problem

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

Example: Derive a Function for an Input of Size *n*

- Associate a ***cost*** with each statement.

- Find the ***total cost*** by finding the total number of times each statement is executed.

Algorithm 1

	Cost
arr[0] = 0;	c_1
arr[1] = 0;	c_1
arr[2] = 0;	c_1
...	...
arr[N-1] = 0;	c_1

	$c_1+c_1+...+c_1 = c_1 \times N$

Algorithm 2

	Cost
for(i=0; i<N; i++)	c_2
arr[i] = 0;	c_1

	$(N+1) \times c_2 + N \times c_1 =$ $(c_2 + c_1) \times N + c_2$

Another Example

- ***Algorithm 3*** *Cost*

```
sum = 0;                          c1
for(i=0; i<N; i++)                c2
   for(j=0; j<N; j++)             c2
      sum += arr[i][j];           c3
```

T(N) = ?

Another Example (cont.)

- ***Algorithm 3*** *Cost*

```
sum = 0;                          c1
for(i=0; i<N; i++)                c2
   for(j=0; j<N; j++)             c2
      sum += arr[i][j];           c3
```

$c_1 + c_2 \times (N+1) + c_2 \times N \times (N+1) + c_3 \times N^2$

Asymptotic Analysis

- To compare two algorithms with running times *f(n)* and *g(n),* we need a **rough measure** that characterizes **how fast each function grows.**

- *Use rate of growth*

- Compare functions in the limit, that is, **asymptotically!**
  (i.e., for large values of *n*)

Rate of Growth

- The low order terms in a function are relatively insignificant for **large** n

$$n^4 + 100n^2 + 10n + 50 \quad \sim \quad n^4$$

i.e., we say that $n^4 + 100n^2 + 10n + 50$ and n^4 have the same **rate of growth**

Asymptotic Notation

- O notation: asymptotic "less than":
  - $f(n) \in O(g(n))$ implies: $f(n)$ "$\leq$" $g(n)$
- Ω notation: asymptotic "greater than":
  - $f(n) \in \Omega(g(n))$ implies: $f(n)$ "$\geq$" $g(n)$
- Θ notation: asymptotic "equality":
  - $f(n) \in \Theta(g(n))$ implies: $f(n)$ "$=$" $g(n)$

Informal Introduction to Big-Oh

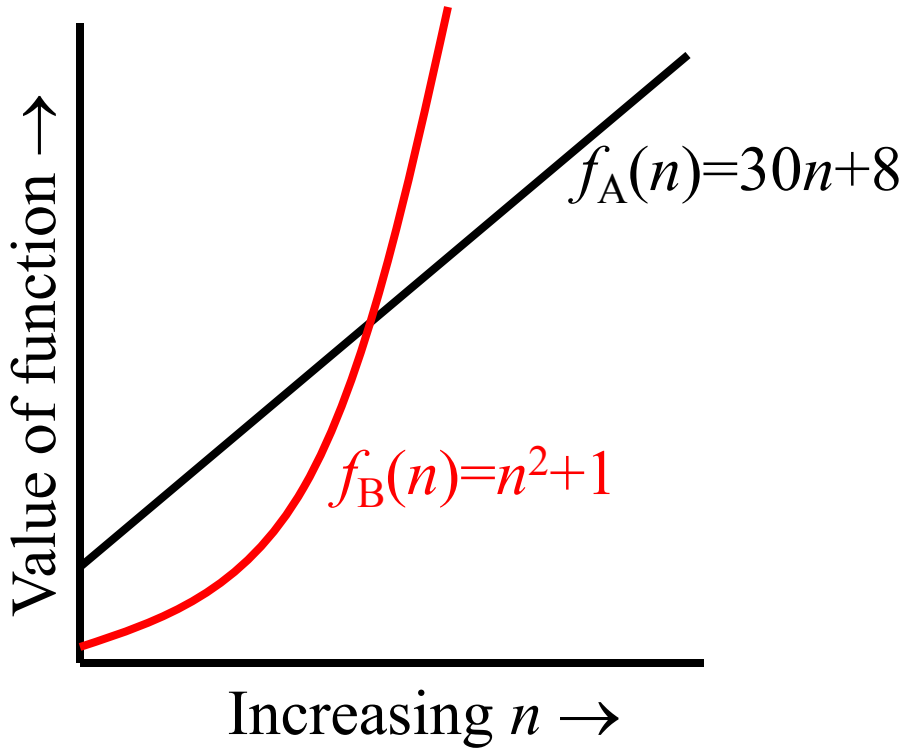
- We say $\boldsymbol{f_A(n)=30n+8}$ is *order n*, or O (*n*)
  It is, at most, roughly *proportional* to *n*.

- $\boldsymbol{f_B(n)=n^2+1}$ is *order* n^2, or O(n^2). It is, at most, roughly proportional to n^2.

- In general, any O(n^2) function is faster-growing than any O(*n*) function.

Visualizing Orders of Growth

- On a graph, as you go to the right, a faster growing function eventually becomes larger...



More Examples …

- $n^4 + 100n^2 + 10n + 50$ is $O(n^4)$
- $10n^3 + 2n^2$ is $O(n^3)$
- $n^3 - n^2$ is $O(n^3)$
- constants
  - 10 is $O(1)$
  - 1273 is $O(1)$

Back to Our Example

Algorithm 1

	Cost
arr[0] = 0;	c_1
arr[1] = 0;	c_1
arr[2] = 0;	c_1
...	
arr[N-1] = 0;	c_1

$c_1+c_1+...+c_1 = c_1 \times N$

Algorithm 2

	Cost
for(i=0; i<N; i++)	c_2
arr[i] = 0;	c_1

$(N+1) \times c_2 + N \times c_1 =$
$(c_2 + c_1) \times N + c_2$

- Both algorithms are of the same order: *O(?)*

Back to Our Example

Algorithm 1

	Cost
arr[0] = 0;	c_1
arr[1] = 0;	c_1
arr[2] = 0;	c_1
...	
arr[N-1] = 0;	c_1

$c_1+c_1+...+c_1 = c_1 \times N$

Algorithm 2

	Cost
for(i=0; i<N; i++)	c_2
arr[i] = 0;	c_1

$(N+1) \times c_2 + N \times c_1 =$
$(c_2 + c_1) \times N + c_2$

- Both algorithms are of the same order: *O(N)*

Example (cont'd)

Algorithm 3	***Cost***
sum = 0;	c_1
for(i=0; i<N; i++)	c_2
for(j=0; j<N; j++)	c_2
sum += arr[i][j];	c_3

$$c_1 + c_2 \text{ x } (N+1) + c_2 \text{ x } N \text{ x } (N+1) + c_3 \text{ x } N^2 = O(?)$$

Example (cont'd)

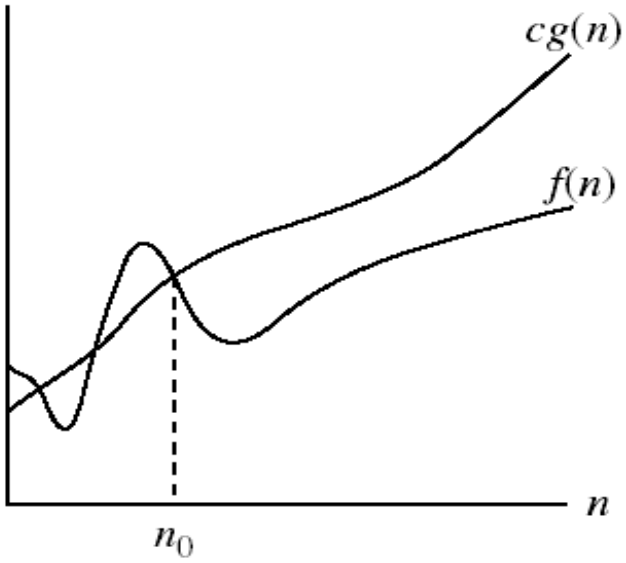
Algorithm 3	***Cost***
sum = 0;	c_1
for(i=0; i<N; i++)	c_2
for(j=0; j<N; j++)	c_2
sum += arr[i][j];	c_3

$$c_1 + c_2 \text{ x } (N+1) + c_2 \text{ x } N \text{ x } (N+1) + c_3 \text{ x } N^2 = O(N^2)$$

Asymptotic notations

- *O-notation*

$$O(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0\}.$$



O(g(*n*)) is the set of functions with smaller or same order of growth as g(*n*)

$g(n)$ is an ***asymptotic upper bound*** for $f(n)$.

Examples

- Show that $30n + 8 \in O(n)$.

  - Show $\exists\, c, n_0$: $30n + 8 \leq cn$, $\forall n > n_0$.

    - Let $c = 31$, $n_0 = 8$. Assume $n > n_0 = 8$.

    - Then $cn = 31n = 30n + n > 30n + 8$

      So, $30n + 8 < cn$.

No Uniqueness

- There is no unique set of values for n_0 and c in proving the asymptotic bounds
- Prove that $100n + 5 \in O(n^2)$

  $n_0 = 5$ and $c = 101$ is a solution

  $n_0 = 1$ and $c = 105$ is also a solution

Must find **SOME** constants c and n_0 that satisfy the asymptotic notation relation

Examples

- $2n^2 \in O(n^3)$:
- $n^2 \in O(n^2)$:
- $1000n^2+1000n \in O(n^2)$:
- $n \in O(n^2)$:

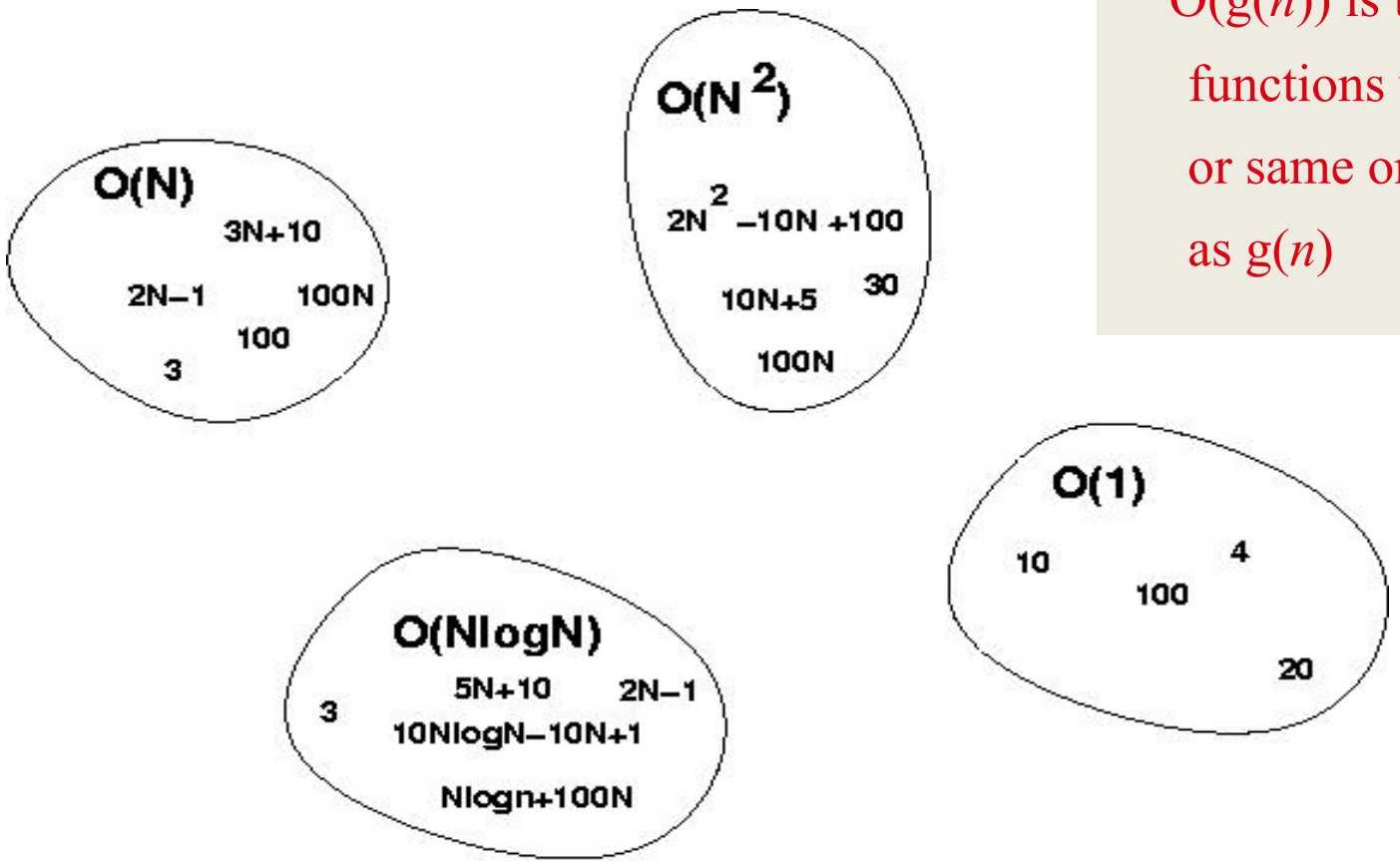
Examples

- $2n^2 \in O(n^3)$: $2n^2 \leq cn^3 \Rightarrow 2 \leq cn \Rightarrow c = 1$ and $n_0 = 2$
- $n^2 \in O(n^2)$: $n^2 \leq cn^2 \Rightarrow c \geq 1 \Rightarrow c = 1$ and $n_0 = 1$
- $1000n^2+1000n \in O(n^2)$:

  $1000n^2+1000n \leq 1000n^2+ n^2 =1001n^2 \Rightarrow c=1001$ and $n_0 = 1000$

- $n \in O(n^2)$: $n \leq cn^2 \Rightarrow cn \geq 1 \Rightarrow c = 1$ and $n_0 = 1$

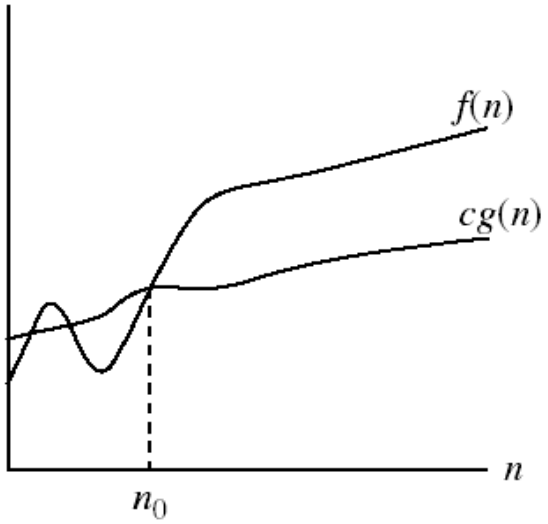
Big-O Visualization



O(g(*n*)) is the set of functions with smaller or same order of growth as g(*n*)

Asymptotic notations (cont.)

$\Omega(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that $0 \le cg(n) \le f(n)$ for all $n \ge n_0\}$.



$\Omega(g(n))$ is the set of functions with larger or same order of growth as $g(n)$

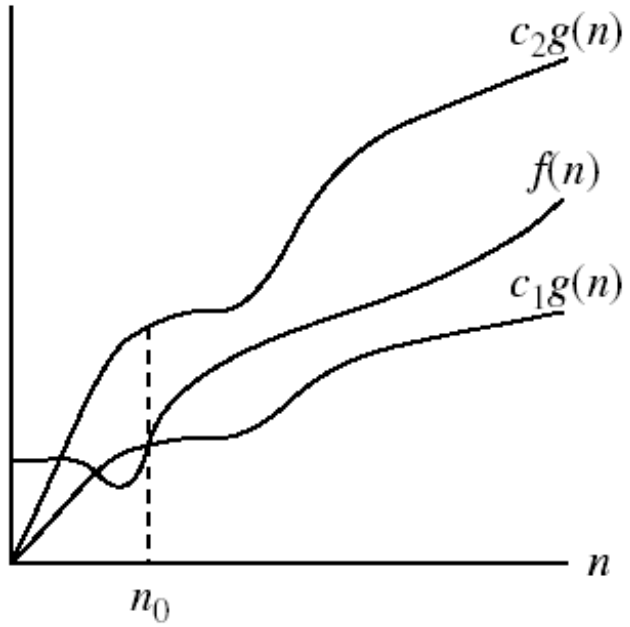
$g(n)$ is an ***asymptotic lower bound*** for $f(n)$.

Example: $5n^2 \in \Omega(n)$

$\exists\ c, n_0$ such that: $0 \le cn \le 5n^2 \Rightarrow cn \le 5n^2 \Rightarrow$ c = 1 and $n_0 = 1$

Asymptotic notations (cont.)

- *Θ-not* $\Theta(g(n)) = \{f(n)$: there exist positive constants c_1, c_2, and n_0 such that $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0\}$.



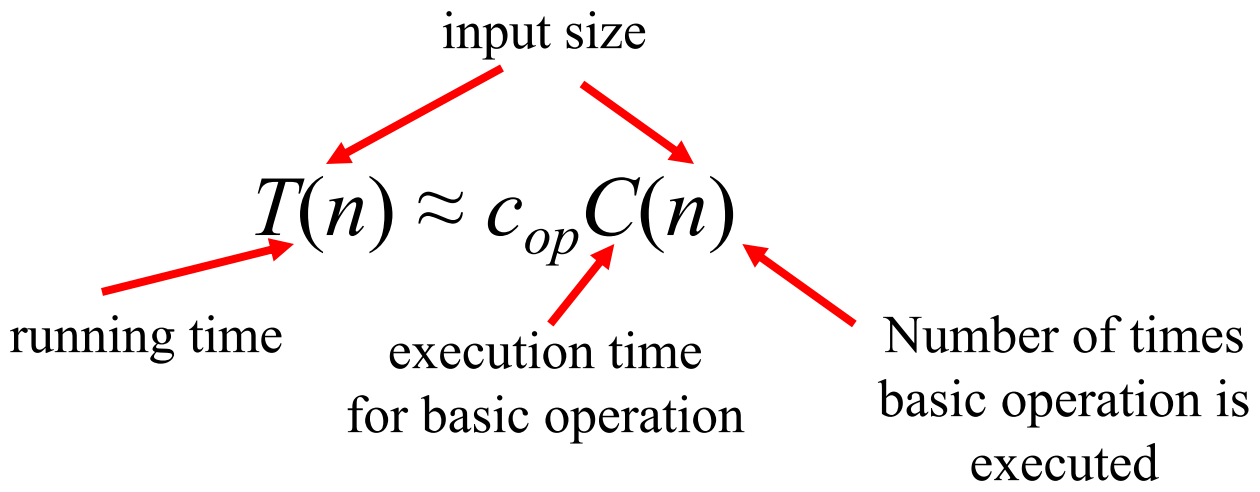
Θ(g(*n*)) is the set of functions with the same order of growth as g(*n*)

$g(n)$ is an ***asymptotically tight bound*** for $f(n)$.

Review of theoretical analysis of time efficiency

Time efficiency is analyzed by determining the number of repetitions of the *basic operation* as a function of *input size*

- *Basic operation*: the operation that contributes most towards the running time of the algorithm



Establishing order of growth using limits

$$\lim_{n\to\infty} f(n)/g(n) = \begin{cases} \mathbf{0} & \text{order of growth of } f(n) \ \mathbf{<} \ \text{order of growth of } g(n) \\ \mathbf{c > 0} & \text{order of growth of } f(n) \ \mathbf{=} \ \text{order of growth of } g(n) \\ \infty & \text{order of growth of } f(n) \ \mathbf{>} \ \text{order of growth of } g(n) \end{cases}$$

Examples:

- $10n$ vs. n^2
- $n(n+1)/2$ vs. n^2

L'Hôpital's rule

L'Hôpital's rule: If $\lim_{n\to\infty} f(n) = \lim_{n\to\infty} g(n) = \infty$ and the derivatives f′, g′ exist, then

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

Example: log n vs. n

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

Orders of growth of some important functions

- All logarithmic functions $\log_a n$ belong to the same class $\Theta(\log n)$ no matter what the logarithm's base $a > 1$ is
- All polynomials of the same degree k belong to the same class: $a_k n^k + a_{k-1} n^{k-1} + \ldots + a_0 \in \Theta(n^k)$
- Exponential functions a^n have different orders of growth for different a's
- order $\log n$ < order n^α ($\alpha>0$) < order a^n < order $n!$ < order n^n

Overview of Math Concepts

Math Concepts

- Exponents
- Logarithms
- Arithmetic Series
- Recurrence Relations

Exponents

- Exponents
  - X^Y = "X to the Y^{th} power";
    X multiplied by itself Y times

- Some useful identities
  - $X^A X^B = X^{A+B}$
  - $X^A / X^B = X^{A-B}$
  - $(X^A)^B = X^{AB}$
  - $X^N+X^N = 2X^N$
  - $2^N+2^N = 2^{N+1}$

Logarithms

- Logarithms
  - Definition: $X^A = B$ if and only if $\log_X B = A$
  - Intuition: $\log_X B$ means "the power X must be raised to, to get B"

  - A logarithm with no base implies base 2
    log B means $\log_2 B$

- Examples
  - $\log_2 16 = 4$ (because $2^4 = 16$)
  - $\log_{10} 1000 = 3$ (because $10^3 = 1000$)

Logarithms, continued

- log AB = log A + log B
- log A/B = log A – log B
- log (A^B) = B log A
- $\log_A B = \frac{\log_C B}{\log_C A} \quad A, B, C > 0, A \neq 1$
  - Example:
    $\log_4 32 = (\log_2 32) / (\log_2 4) = 5 / 2$

Arithmetic series

- Series $\sum_{i=j}^{k} Expr$
  - For some expression *Expr* (possibly containing *i*), means the sum of all values of *Expr* with each value of *i* between *j* and *k* inclusive
- Example: $\sum_{i=0}^{4} 2i+1$

= (2(0) + 1) + (2(1) + 1) + (2(2) + 1) + (2(3) + 1) + (2(4) + 1)
= 1 + 3 + 5 + 7 + 9
= 25

Series identities

- Sum from 1 through N inclusive

$$\sum_{i=1}^{N} i = \frac{N(N+1)}{2}$$

- Is there an intuition for this identity?
  - Sum of all numbers from 1 to N

    1 + 2 + 3 + ... + (N-2) + (N-1) + N

  - How many terms are in this sum? Can we rearrange them?

Series

- Sum of powers of 2

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

= 1 + 2 + 4 + 8 + 16 + 32 = 64 – 1

- Sum of powers of any number *a*
  - ("Geometric progression" for $a>0$, $a \neq 1$)

$$\sum_{i=0}^{N} a^i = \frac{1 - a^{N+1}}{1 - a}$$

Recurrence Relations

Sequences and Explicit Formulas

Consider the following two sequences:

$$S_1 : 3, 5, 7, 9, \ldots$$

$$S_2 : 3, 9, 27, 81, \ldots$$

We can find a formula for the nth term of sequences S_1 and S_2 by observing the pattern of the sequences.

Sequences and Explicit Formulas

Consider the following two sequences:

$$S_1 : 3, 5, 7, 9, \ldots$$
$$S_2 : 3, 9, 27, 81, \ldots$$

We can find a formula for the nth term of sequences S_1 and S_2 by observing the pattern of the sequences.

$$S_1 : 2 \cdot 1 + 1, 2 \cdot 2 + 1, 2 \cdot 3 + 1, 2 \cdot 4 + 1, \ldots$$
$$S_2 : 3^1, 3^2, 3^3, 3^4, \ldots$$

For S_1, $a_n = 2n + 1$ for $n \geq 1$, and for S_2, $a_n = 3^n$ for $n \geq 1$. This type of formula is called an **explicit formula** for the sequence, because using this formula we can directly find any term of the sequence without using other terms of the sequence. For example, $a_3 = 2 \cdot 3 + 1 = 7$.

Sequences and Recurrence Equations

- Let *S* denote the Fibonacci sequence 1, 1, 2, 3, 5, 8, 13, 21, …
- Explicit formula is not so obvious

Sequences and Recurrence Equations

- Let *S* denote the Fibonacci sequence 1, 1, 2, 3, 5, 8, 13, 21, …
- Explicit formula is not so obvious
- Pattern of the sequence:

  3^{rd} term = 2 = 1 + 1 = 1^{st} term + 2^{nd} term

  4^{th} term = 3 = 1 + 2 = 2^{nd} term + 3^{rd} term

  5^{th} term = 5 = 2 + 3 = 3^{rd} term + 4^{th} term

  6^{th} term = 8 = 3 + 5 = 4^{th} term + 5^{th} term

  7^{th} term = 13 = 5 + 8 = 5^{th} term + 6^{th} term

Sequences and Recurrence Equations

- Let *S* denote the Fibonacci sequence 1, 1, 2, 3, 5, 8, 13, 21, …
- Explicit formula is not so obvious
- Pattern of the sequence:

  3^{rd} term = 2 = 1 + 1 = 1^{st} term + 2^{nd} term

  4^{th} term = 3 = 1 + 2 = 2^{nd} term + 3^{rd} term

  5^{th} term = 5 = 2 + 3 = 3^{rd} term + 4^{th} term

  6^{th} term = 8 = 3 + 5 = 4^{th} term + 5^{th} term

  7^{th} term = 13 = 5 + 8 = 5^{th} term + 6^{th} term

- Hence, the sequence *S* can be defined by the equation

$$f_n = f_{n-1} + f_{n-2} \qquad \textbf{(I)}$$

for all $n \geq 3$ and

$$f_1 = 1, f_2 = 1. \qquad \textbf{(II)}$$

- Because f_n is defined in terms of previous terms of the sequence, equations of the form **(I)** are called ***recurrence equations*** of ***recurrence relations***.

Recursive Definition and Recurrence Relation

- Let $f : \mathbb{N} \to \mathbb{Z}^+$,

$$f(n) = 2f(n\text{-}1) + f(n\text{-}2), \text{ for all } n \geq 2 \qquad \text{(III)}$$

and

$$f(0) = 5, \quad f(1) = 7 \qquad \text{(IV)}$$

Recursive Definition and Recurrence Relation

- Let $f : \mathbb{N} \to \mathbb{Z}^+$,

$$f(n) = 2f(n\text{-}1) + f(n\text{-}2), \text{ for all } n \geq 2 \qquad \text{(III)}$$

and

$$f(0) = 5, \quad f(1) = 7 \qquad \text{(IV)}$$

- We can compute f_4 as follows:

$f(4) = 2f(3)+f(2)$

$f(2) = 2f(1)+f(0) = 2 \cdot 7 + 5 = 19$

$f(3) = 2f(2)+f(1) = 2 \cdot 19 + 7 = 45$

$f(4) = 2f(3)+f(2) = 2 \cdot 45 + 19 = 109$

- The definition of the function f as given in (III) is called a ***recursive definition***
- In this case, the equation $f(n) = 2f(n\text{-}1) + f(n\text{-}2)$, for all $n \geq 2$ is called a ***recurrence relation***, and $f(0) = 5$ and $f(1) = 7$ are called ***initial conditions*** for the function f.

Formal Definition of Recurrence Relation

A **recurrence relation** for a sequence $a_0, a_1, a_2, \ldots, a_n, \ldots$ is an equation that relates a_n to some of the terms $a_0, a_1, a_2, \ldots, a_{n-2}, a_{n-1}$ for all integers n with $n \geq k$, where k is a nonnegative integer. The **initial conditions** for the recurrence relation are a set of values that explicitly define some of the members of $a_0, a_1, a_2, \ldots, a_{k-1}$.

The equation

$$a_n = 2a_{n-1} + a_{n-2} \quad \text{for all } n \geq 2,$$

as defined above, relates a_n to a_{n-1} and a_{n-2}. Here $k = 2$. So this is a recurrence relation with initial conditions $a_0 = 5$ and $a_1 = 7$.

Sequences and Recurrence Relations

Consider the function $f : \mathbb{N}^0 \to \mathbb{Z}^+$ defined by

$$f(0) = 1,$$
$$f(n) = nf(n-1) \quad \text{for all } n \geq 1.$$

Sequences and Recurrence Relations

Consider the function $f : \mathbb{N}^0 \to \mathbb{Z}^+$ defined by

$$f(0) = 1,$$
$$f(n) = nf(n-1) \quad \text{for all } n \geq 1.$$

Then

$$f(0) = 1 = 0!,$$
$$f(1) = 1 \cdot f(0) = 1 = 1!,$$
$$f(2) = 2 \cdot f(1) = 2 \cdot 1 = 2 = 2!,$$
$$f(3) = 3 \cdot f(2) = 3 \cdot 2 \cdot 1 = 6 = 3!,$$

and so on. Here $f(n) = nf(n-1)$ for all $n \geq 1$ is the recurrence relation, and $f(0) = 1$ is the initial condition for the function f. Notice that the function f is nothing but the factorial function, i.e., $f(n) = n!$ for all $n \geq 0$.

Solution of a Recurrence Relations

Suppose a recurrence relation for a sequence $a_0, a_1, a_2, \ldots, a_n, \ldots$, is given. By a *solution of the recurrence relation* we mean to obtain an explicit formula for a_n, i.e., to find an expression for a_n that does not involve any other a_i.

- **Example:** Consider again sequence *S*: $a_1, a_2, \ldots, a_n, \ldots$ defined by

$$a_n = a_{n-1} + 7, \qquad \text{if } n > 1 \qquad \text{(V)}$$

with initial condition

$$a_1 = 4, \qquad \text{(VI)}$$

  - What is the explicit formula for a_n (i.e., expression for a_n that does not involve any other a_i)????

Determining Explicit Formulas

- In (V), replace n by n-1 to obtain

$$a_{n-1} = a_{n-2} + 7$$

- Substitute a_{n-1} into a_n to obtain

$$a_n = a_{n-1} + 7 = (a_{n-2} + 7) + 7 = a_{n-2} + 2 \bullet 7$$

- Next in (V), replace n by n-2 to obtain

$$a_{n-2} = a_{n-3} + 7$$

- Substitute a_{n-2} into a_n to obtain

$$a_n = a_{n-2} + 2 \bullet 7 = (a_{n-3} + 7) + 7 = a_{n-3} + 3 \bullet 7$$

- In general, we have

$$a_n = a_{n-k} + k \bullet 7,\ k = 1, 2, 3, \ldots, n\text{-}1 \qquad \text{(VII)}$$

Determining Explicit Formulas (continued)

- In (VII), substitute $k = n\text{-}1$ to obtain

$$a_n = a_{n-(n-1)} + (n\text{-}1) \bullet 7 = a_1 + (n\text{-}1) \bullet 7$$

- Next, substitute $a_1 = 4$ to obtain

$$a_n = a_1 + (n\text{-}1) \bullet 7 = 4 + (n\text{-}1) \bullet 7 = 7(n\text{-}1) + 4$$

$$= 7n - 3, \forall_{n \geq 1}, \qquad \text{(VIII)}$$

which gives the explicit formula for a_n.

- The explicit formula given by (VIII) seems to be correct.

- But how do we verify its correctness?????

Using Induction to verify Correctness of Explicit Formula

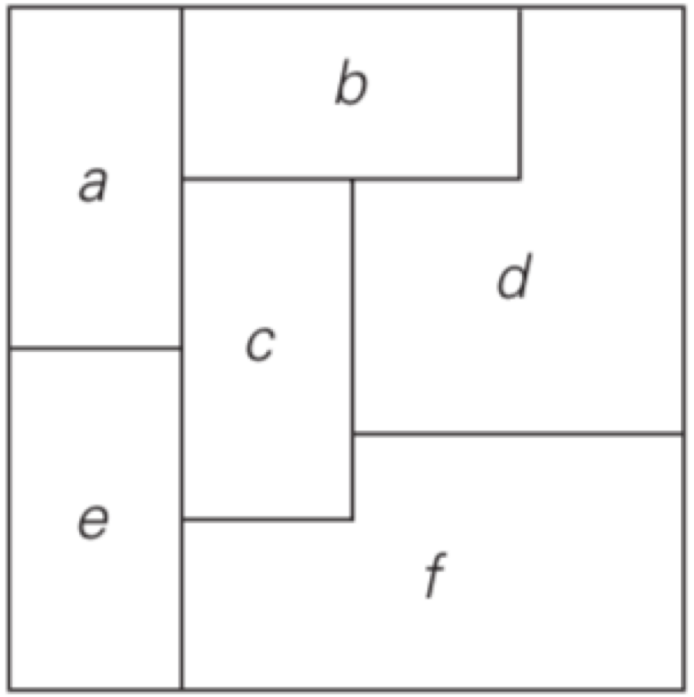
- Our explicit formula: $a_n = 7n - 3, \forall_{n \geq 1}$. (IX)
- *Basis step*: $n = 1 \rightarrow a_1 = 7 \bullet 1 - 3 = 4$, true.
- *Inductive hypothesis*: assume (IX) is true for $n = k \geq 1$, i.e., $a_k = 7k - 3$
- *Inductive step*: Let $n = k+1$. We have

  $a_{k+1} = a_k + 7$ (by V – i.e., $a_n = a_{n-1} + 7$)
  $= (7k - 3) + 7$ (by the inductive hypothesis, $a_k = 7k - 3$)
  $= 7k + 7 - 3$
  $= 7(k+1) - 3$

- Thus, (IX) is true for $n = k+1$. Hence, (IX) is true for all $n \geq 1$

Exercise

- Consider the following map:
  - Explain how we can use the graph-coloring problem to color the map so that no two neighboring regions are colored the same.
  - Use your answer to part (a) to color the map with the smallest number of colors.



Some properties of asymptotic order of growth

- $f(n) \in O(f(n))$
- $f(n) \in O(g(n))$ iff $g(n) \in \Omega(f(n))$
- If $f(n) \in O(g(n))$ and $g(n) \in O(h(n))$, then $f(n) \in O(h(n))$

  Note similarity with $a \leq b$
- If $f_1(n) \in O(g_1(n))$ and $f_2(n) \in O(g_2(n))$, then

  $$f_1(n) + f_2(n) \in O(\max\{g_1(n), g_2(n)\})$$

Time efficiency of nonrecursive algorithms

General Plan for Analysis

- Decide on parameter *n* indicating *input size*
- Identify algorithm's *basic operation*
- Determine *worst*, *average*, and *best* cases for input of size *n*
- Set up a sum for the number of times the basic operation is executed
- Simplify the sum using standard formulas and rules (see Appendix A)

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
$maxval \leftarrow A[0]$
for $i \leftarrow 1$ **to** $n-1$ **do**
    if $A[i] > maxval$
        $maxval \leftarrow A[i]$
return $maxval$

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 ← 0 to n − 2 do
    for j ← i + 1 to n − 1 do
        if A[i] = A[j] return false
return true
```

Example 3: Matrix multiplication

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

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

Example 4: Binary

ALGORITHM *Binary*(n)

```
//Input: A positive decimal integer n
//Output: The number of binary digits in n's binary representation
count ← 1
while n > 1 do
    count ← count + 1
    n ← ⌊n/2⌋
return count
```

- The comparison, $n > 1$, decides whether the while loop will be executed
- The value of n is about halved on each repetition of the loop
- The complexity is $\log_2 n + 1$
  - Number of times the comparison is executed = number of repetitions of the loop's body + 1

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 ← 0
while i < n and A[i] ≠ K do
    i ← i + 1
if i < n return i
else return −1
```

Assumptions:

(a) the probability of a successful search is equal to p.

(b) the probability of the first match occurring in the ith position of the list is the same for every i.

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 ← 0
while i < n and A[i] ≠ K do
    i ← i + 1
if i < n return i
else return −1
```

$$C_{avg}(n) = [1 \cdot \frac{p}{n} + 2 \cdot \frac{p}{n} + \cdots + i \cdot \frac{p}{n} + \cdots + n \cdot \frac{p}{n}] + n \cdot (1 - p)$$

$$= \frac{p}{n}[1 + 2 + \cdots + i + \cdots + n] + n(1 - p)$$

$$= \frac{p}{n}\frac{n(n+1)}{2} + n(1 - p) = \frac{p(n+1)}{2} + n(1 - p).$$

Plan for Analysis of Recursive Algorithms

- Decide on a parameter indicating an input's size.
- Identify the algorithm's basic operation.
- Check whether the number of times the basic op. is executed may vary on different inputs of the same size. (If it may, the worst, average, and best cases must be investigated separately.)
- Set up a recurrence relation with an appropriate initial condition expressing the number of times the basic op. is executed.
- Solve the recurrence (or, at the very least, establish its solution's order of growth) by backward substitutions or another method.

Example 1: Recursive evaluation of *n*!

Definition: $n! = 1 \cdot 2 \cdot \ldots \cdot (n\text{-}1) \cdot n$ for $n \geq 1$ and $0! = 1$

Recursive definition of *n*!: $F(n) = F(n\text{-}1) \cdot n$ for $n \geq 1$ and $F(0) = 1$

```
Algorithm F( n ){
    if (n == 0) {    return 1;   }
    else {   return F( n-1 ) * n; }
}
```

Size:
Basic operation:
Recurrence relation:

Solving the recurrence for M(*n*)

M(*n*) = M(*n*-1) + 1 for n>0,

M(0) = 0

M(n) *= M(n − 1) + 1* *sub. M(n−1) = M(n−2) + 1*

=[M(n−2)+1]+1=M(n−2)+2 *sub. M(n−2) = M(n−3) + 1*

=[M(n−3)+1]+2=M(n−3)+3

……

=M(n − i) + i

……

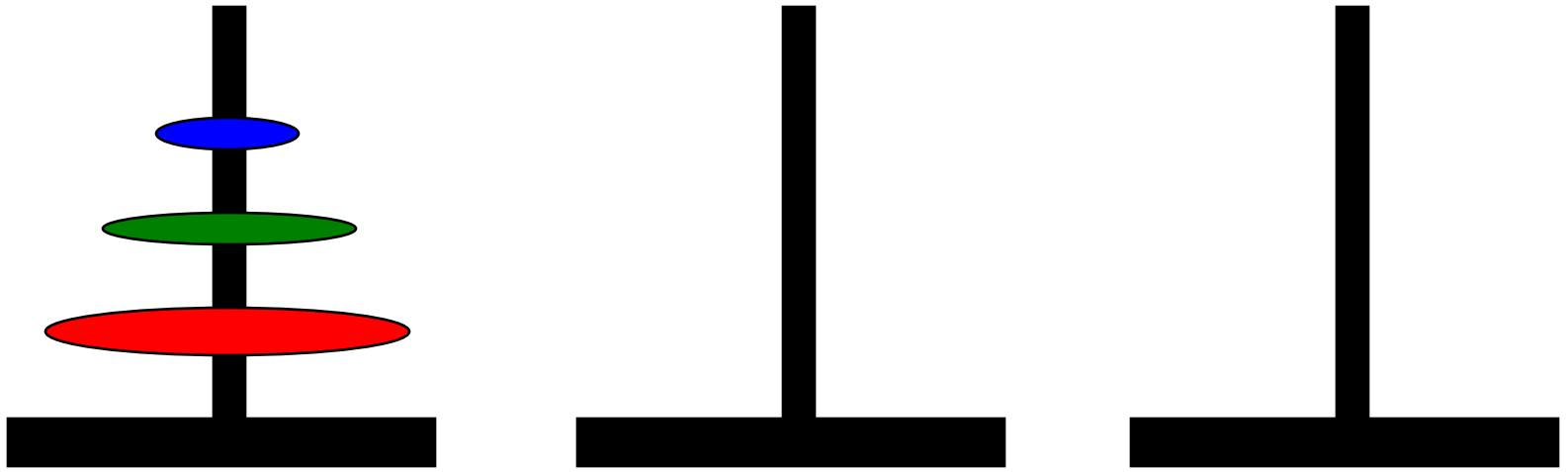
= M(n − n) + n

= n

∈ O(n)

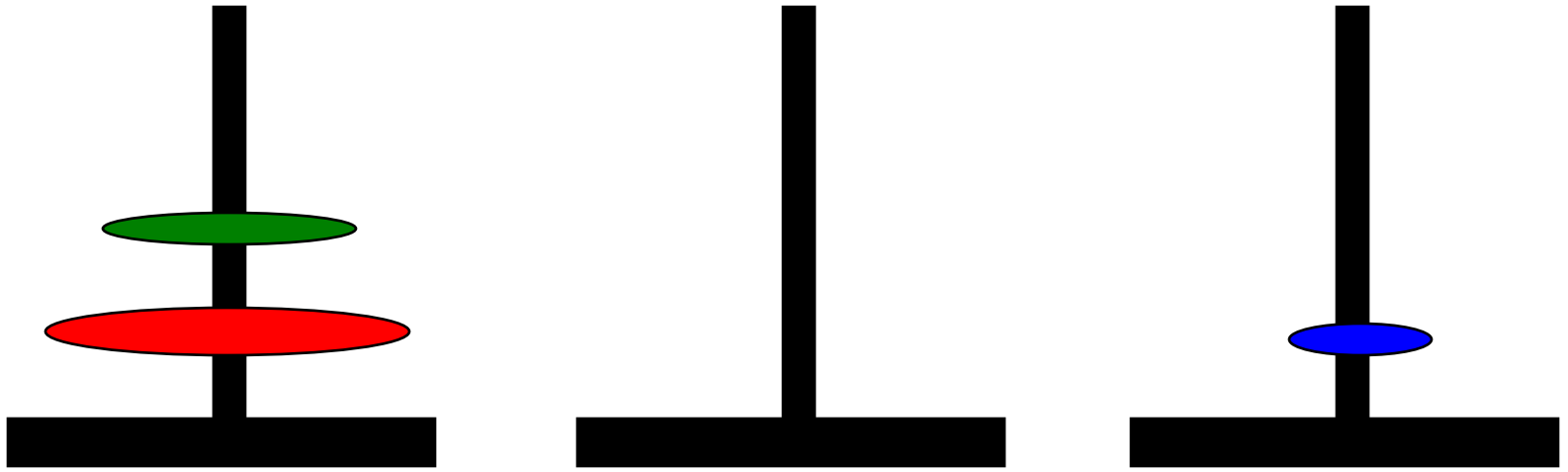
Example 2: Tower of Hanoi Puzzle

- Three poles, some disks of decreasing sizes on the first pole, each one smaller than the one below.
- Task: move all disks to the third pole
- Restrictions: move one disk at a time; a disk cannot be put on top of a smaller disk; disks can only be moved from pole to pole.



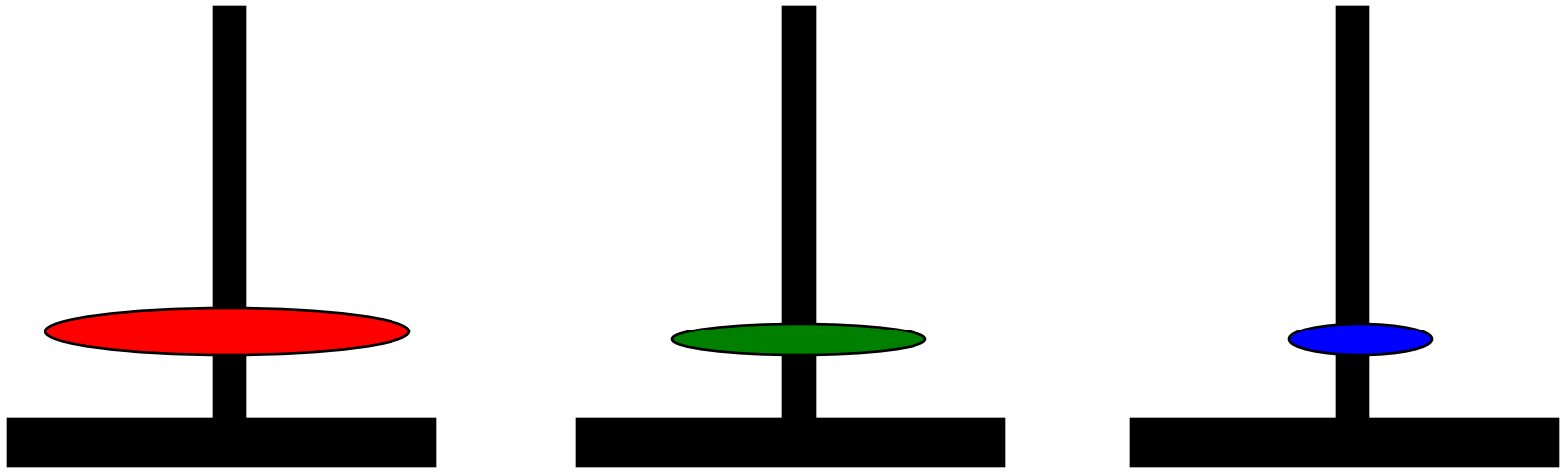
Tower of Hanoi

- Three poles, some disks of decreasing sizes on the first pole, each one smaller than the one below.
- Task: move all disks to the third pole
- Restrictions: move one disk at a time; a disk cannot be put on top of a smaller disk; disks can only be moved from pole to pole.



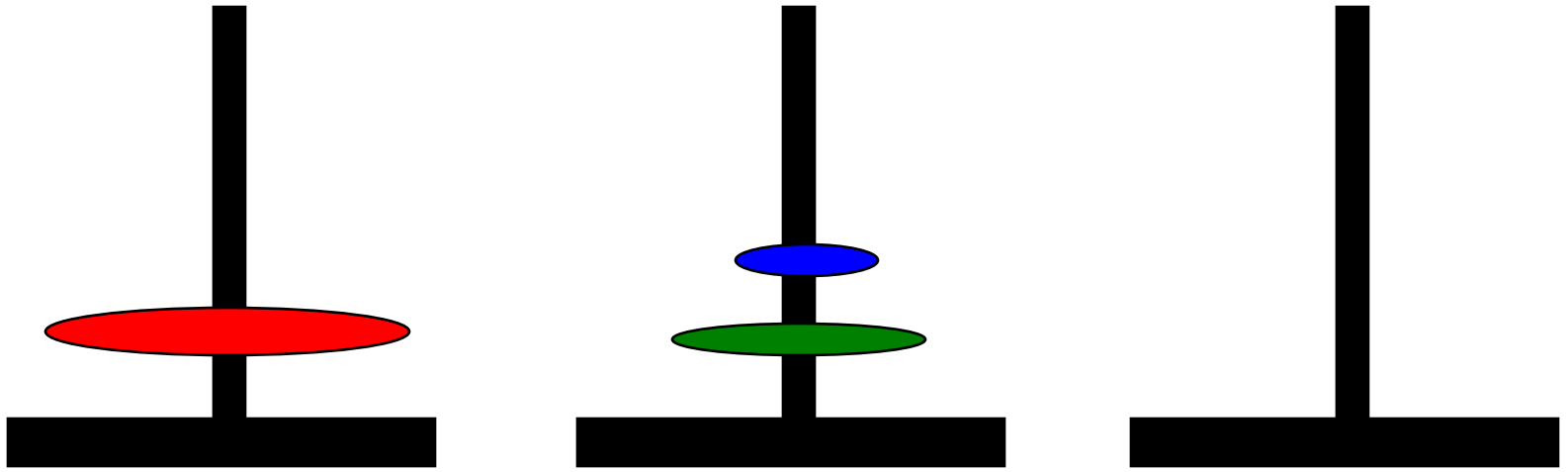
Tower of Hanoi

- Three poles, some disks of decreasing sizes on the first pole, each one smaller than the one below.
- Task: move all disks to the third pole
- Restrictions: move one disk at a time; a disk cannot be put on top of a bigger disk; disks can only be moved from pole to pole.



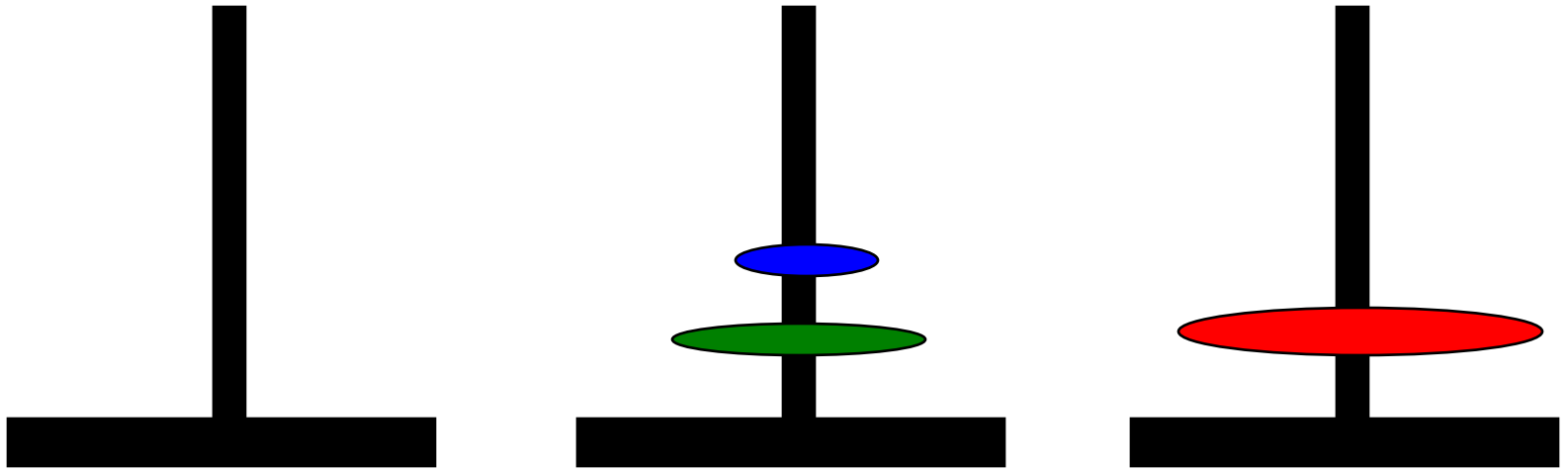
Tower of Hanoi

- Three poles, some disks of decreasing sizes on the first pole, each one smaller than the one below.
- Task: move all disks to the third pole
- Restrictions: move one disk at a time; a disk cannot be put on top of a bigger disk; disks can only be moved from pole to pole.



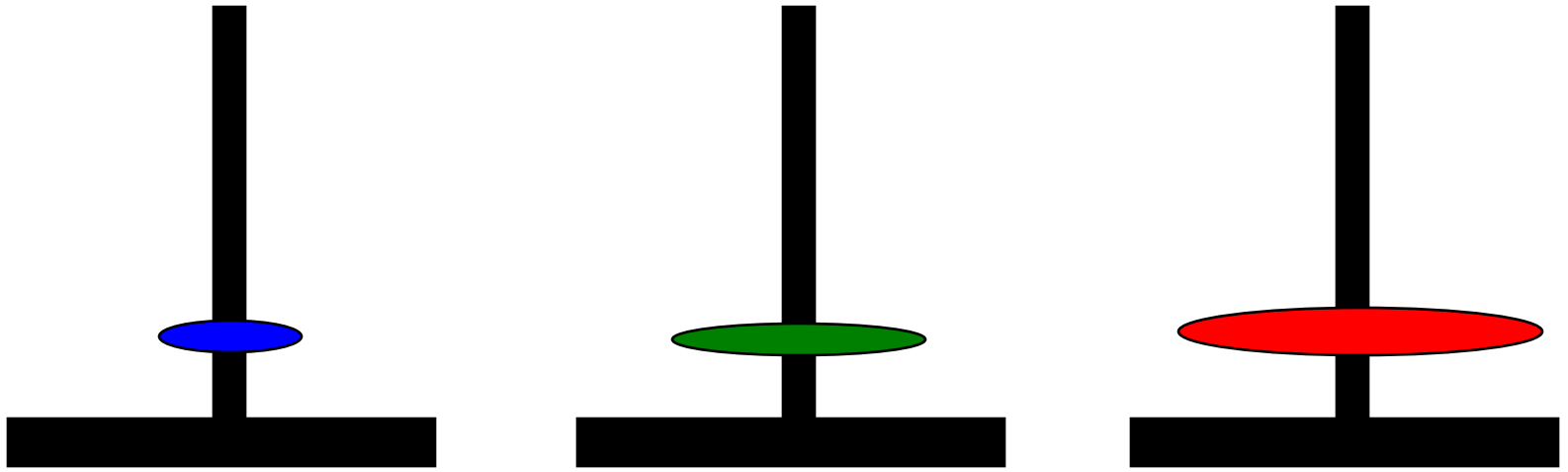
Tower of Hanoi

- Three poles, some disks of decreasing sizes on the first pole, each one smaller than the one below.
- Task: move all disks to the third pole
- Restrictions: move one disk at a time; a disk cannot be put on top of a bigger disk; disks can only be moved from pole to pole.



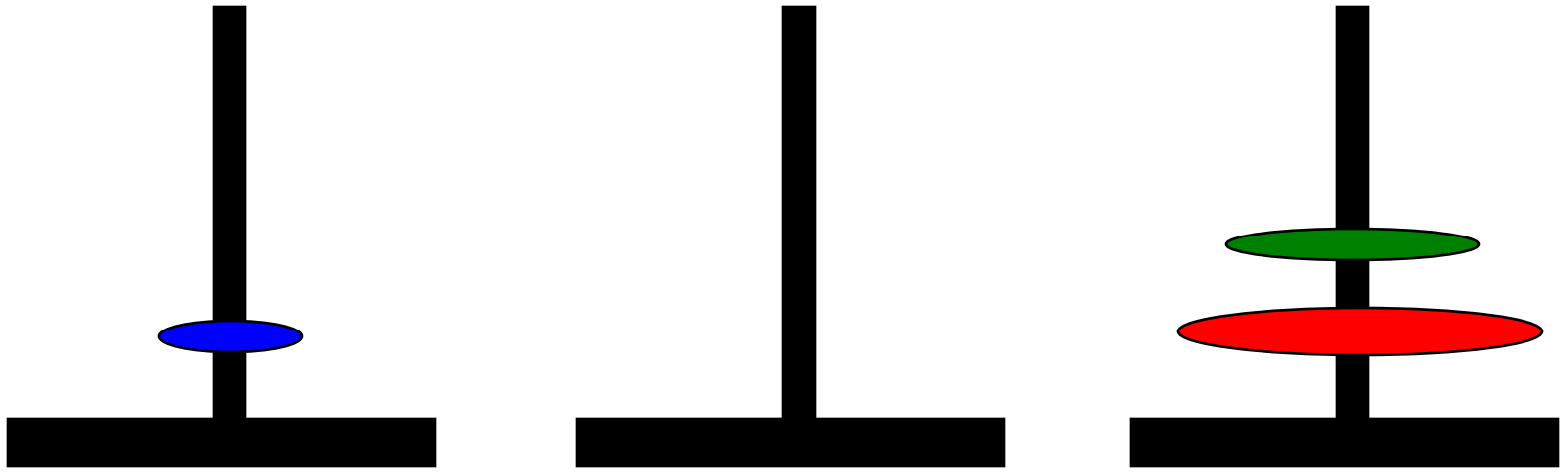
Tower of Hanoi

- Three poles, some disks of decreasing sizes on the first pole, each one smaller than the one below.
- Task: move all disks to the third pole
- Restrictions: move one disk at a time; a disk cannot be put on top of a bigger disk; disks can only be moved from pole to pole.



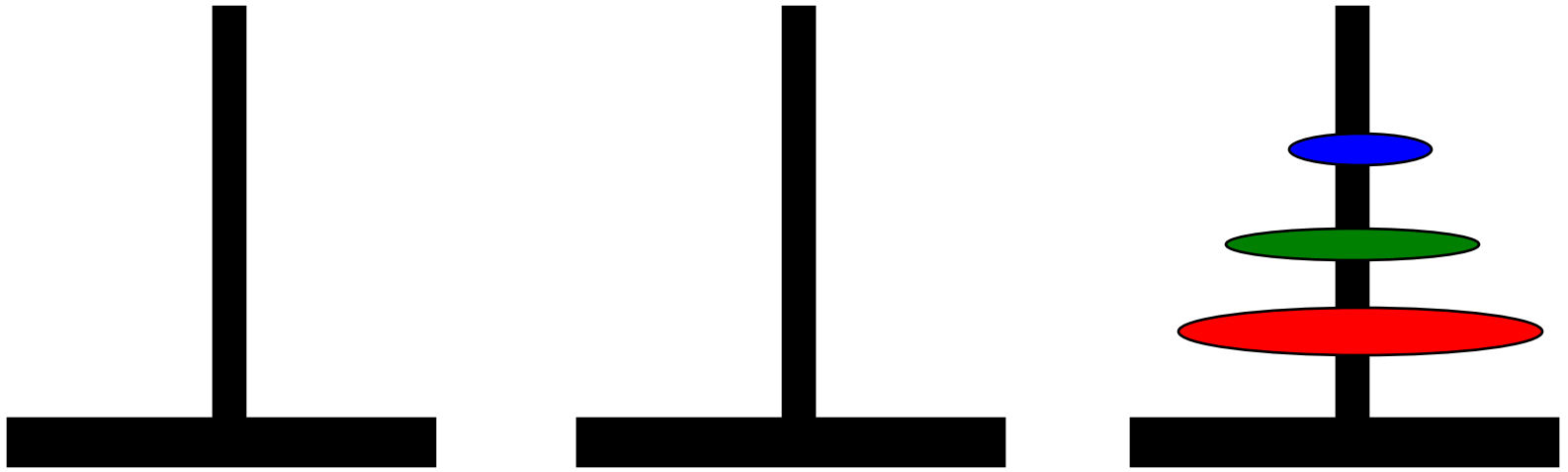
Tower of Hanoi

- Three poles, some disks of decreasing sizes on the first pole, each one smaller than the one below.
- Task: move all disks to the third pole
- Restrictions: move one disk at a time; a disk cannot be put on top of a bigger disk; disks can only be moved from pole to pole.



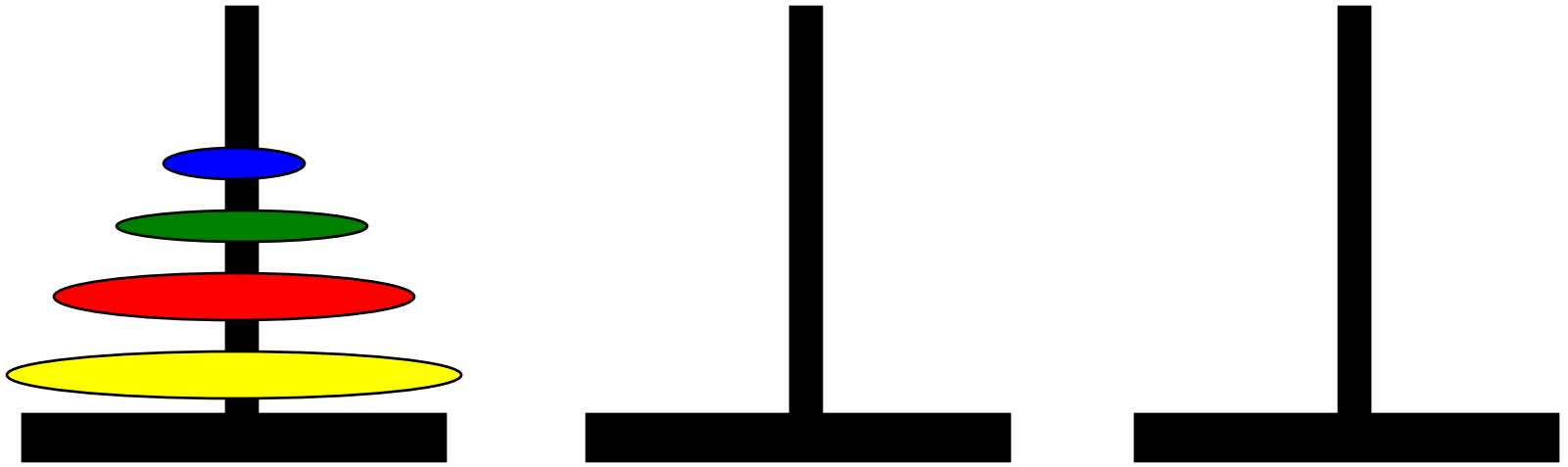
Tower of Hanoi

- Three poles, some disks of decreasing sizes on the first pole, each one smaller than the one below.
- Task: move all disks to the third pole
- Restrictions: move one disk at a time; a disk cannot be put on top of a bigger disk; disks can only be moved from pole to pole.



Solution with recursion:

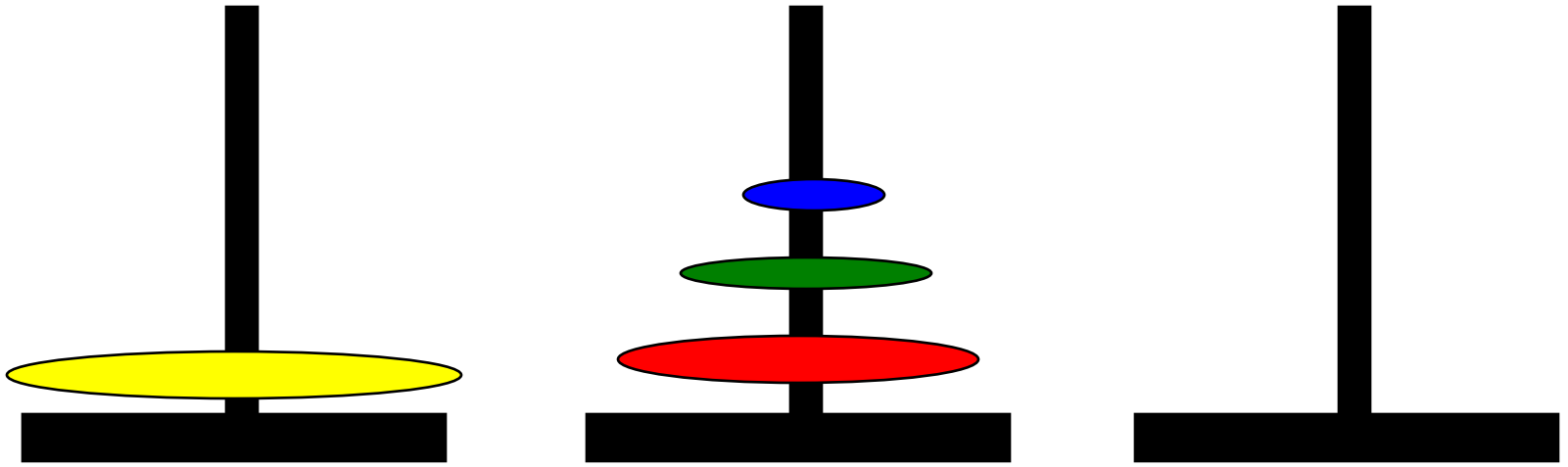
Consider 4 disks – can we solve it using the solution for 3 disks?



Solution with recursion:

Consider 4 disks – can we solve it using the solution for 3 disks?

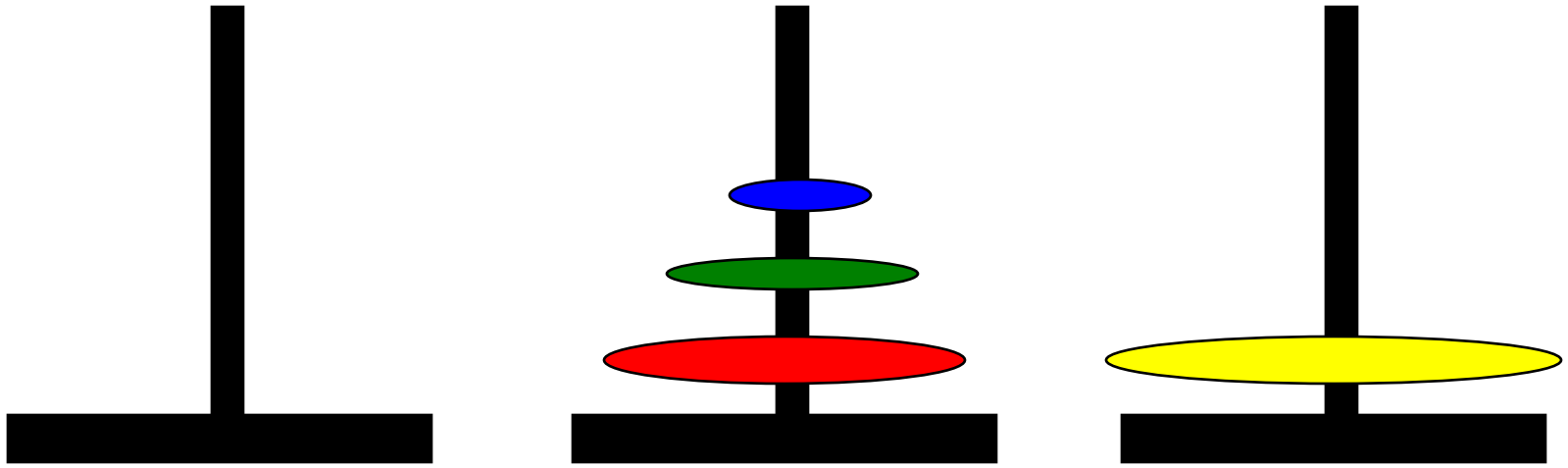
Yes ! Move top 3 disks to the middle pole;



Solution with recursion:

Consider 4 disks – can we solve it using the solution for 3 disks?

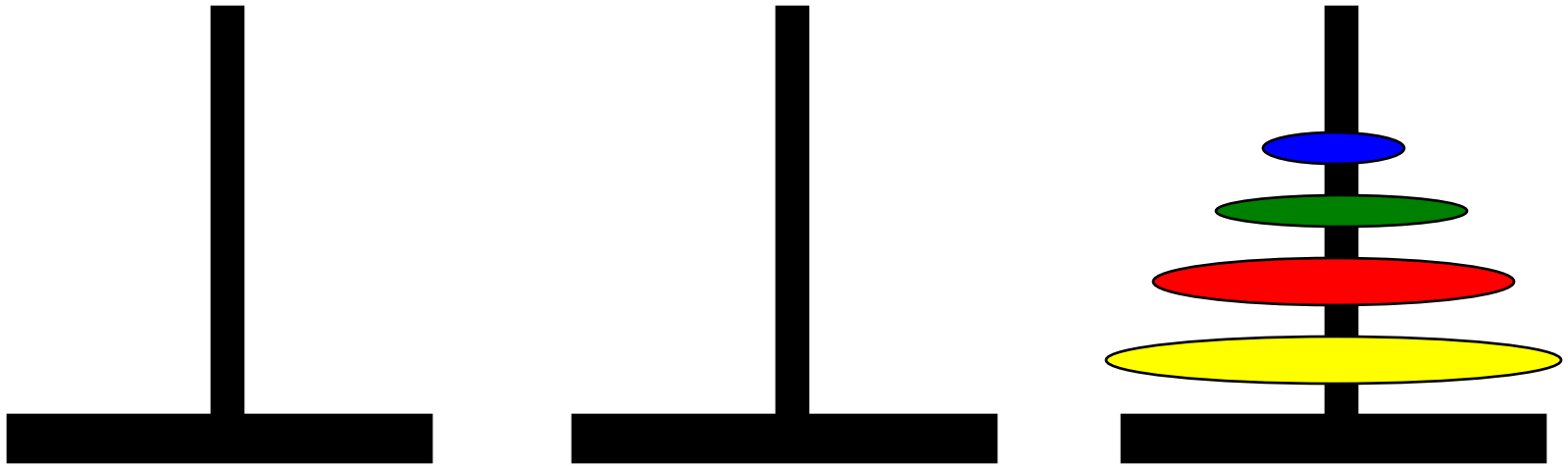
Yes ! Move top 3 disks to the middle pole; move the bottom disk to the target pole;



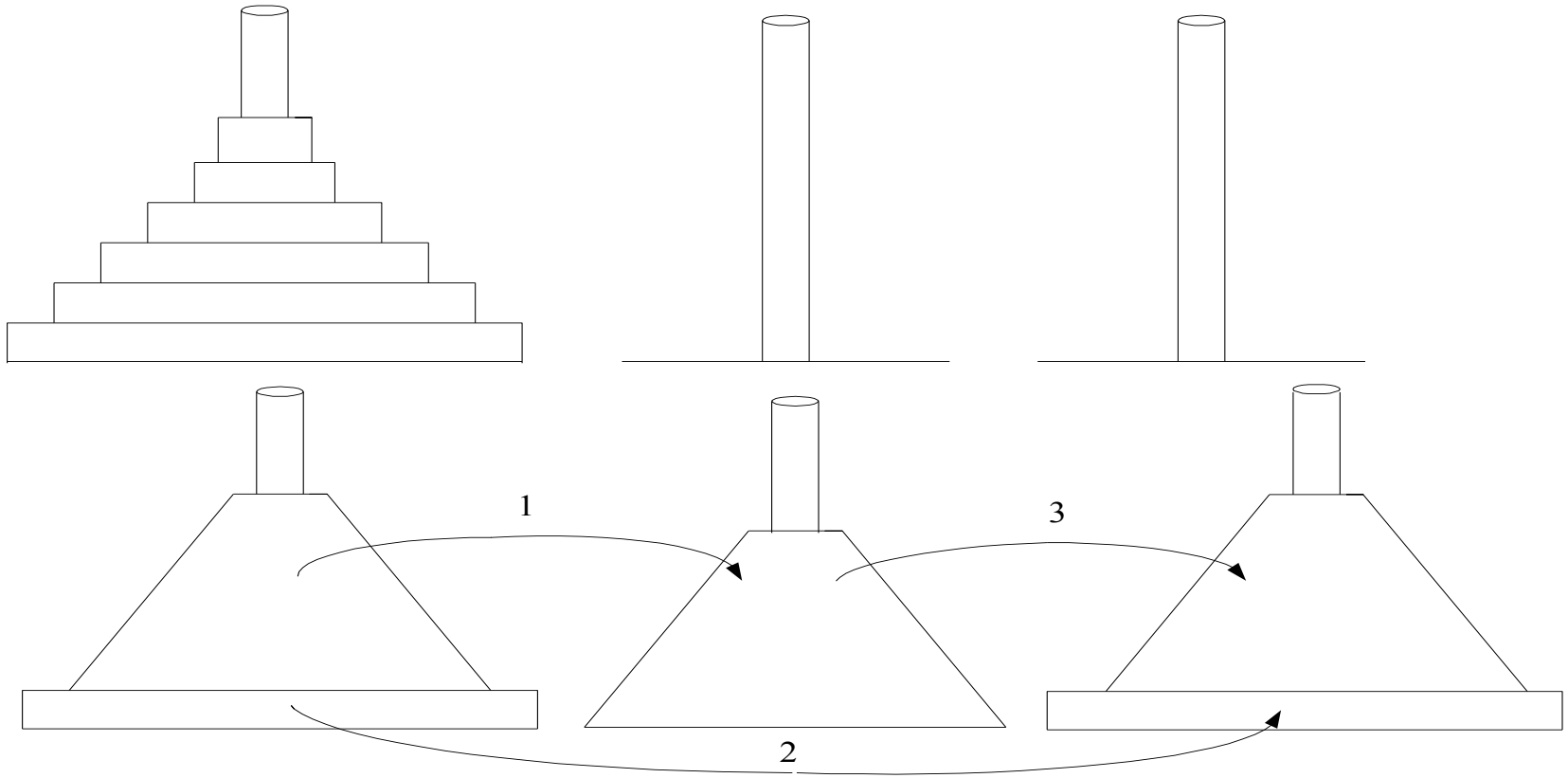
Solution with recursion:

Consider 4 disks – can we solve it using the solution for 3 disks?

Yes ! Move top 3 disks to the middle pole; move the bottom disk to the target pole; move the 3 disks to the target pole.



Generalizing Tower of Hanoi to Case of n Disks



Recurrence for number of moves:

Tower of Hanoi: Recurrence Relation

- Let $M(n)$ be the number of moves required to move n disks, $n \geq 0$, from pole 1 to pole 3
- Step 1 requires us to move the top n-1 disks from pole 1 to pole 2, which requires $M(n\text{-}1)$ moves
- Step 2 requires us to move the n^{th} disk from pole 1 to pole 3, which requires 1 move
- Step 3 requires us to move n-1 disks from pole 2 to pole 3, which requires $M(n\text{-}1)$ moves
- Thus, it follows that:

Tower of Hanoi: Recurrence Relation

- Let $M(n)$ be the number of moves required to move n disks, $n \geq 0$, from pole 1 to pole 3
- Step 1 requires us to move the top n-1 disks from pole 1 to pole 2, which requires $M(n\text{-}1)$ moves
- Step 2 requires us to move the n^{th} disk from pole 1 to pole 3, which requires 1 move
- Step 3 requires us to move n-1 disks from pole 2 to pole 3, which requires $M(n\text{-}1)$ moves
- Thus, it follows that:

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

$$M(1) = 1 \qquad \text{initial condition}$$

Solving the recurrence (cont.)

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

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

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

……

$=2^iM(n-i) + 2^i - 1$

……

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

$= 2^n - 1$

$\in O(2^n)$

Tree of calls for the Tower of Hanoi Puzzle

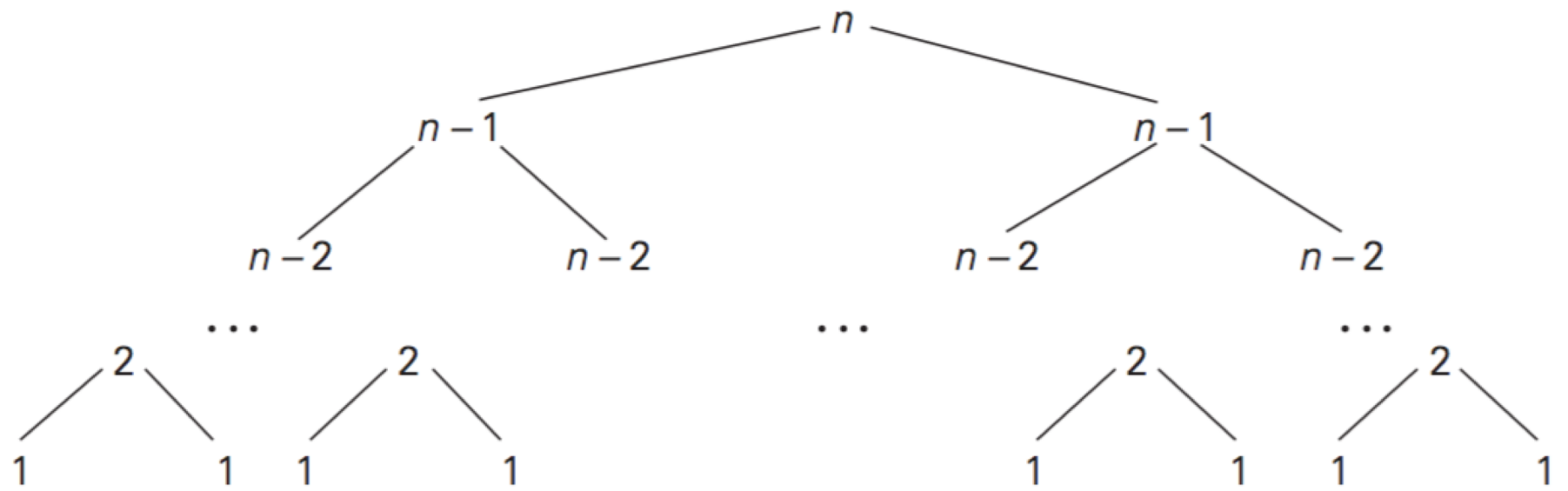


FIGURE 2.5 Tree of recursive calls made by the recursive algorithm for the Tower of Hanoi puzzle.

Example 3: Counting #bits

ALGORITHM *BinRec*(n)

//Input: A positive decimal integer n

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

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

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

Fibonacci numbers

```
0, 1, 1, 2, 3, 5, 8, 13, 21, 34,
  ...
```

where each number ($n \geq 3$) is the sum of the preceding two.

- Recursive definition:
  - `F(0) = 0;`
  - `F(1) = 1;`
  - `F(n) = F(n-1)+ F(n-2);`

Recursive Algorithm to Calculate Fibonacci Number

ALGORITHM $F(n)$

//Computes the nth Fibonacci number recursively by using its definition
//Input: A nonnegative integer n
//Output: The nth Fibonacci number
if $n \leq 1$ **return** n
else return $F(n-1) + F(n-2)$

Characterization of the Running Time

- **`A(n)`** : number of addition
- **`A(0) = A(1) = 0`**;
- For **`n≥2`**, we call **`F(n)`** plus all the calls needed to evaluate **`F(n-1)`** and **`F(n-2)`** recursively and independently;
  - So **`A(n) = A(n-1) + A(n-2) + 1, n≥2`**

Redundant Calculations I

- To compute **`F(n)`**, we recursively compute **`F(n-1)`**.
- When the recursive call return, we compute **`F(n-2)`** using another recursive call
  - We have already computed **`F(n-2)`** in the process of computing **`F(n-1)`**
  - We make two calls to **`F(n-2)`**

Redundant Calculations II

- Making two method calls would double the running time
- Compounding effect: each recursive call does more and more redundant work
  - Each call to **`F(n-1)`** and each call to **`F(n-2)`** makes a call to **`F(n-3)`**
    - there are 3 calls to **`F(n-3)`**
  - Each call to **`F(n-2)`** or **`F(n-3)`** results in a call to **`F(n-4)`**
    - so 5 calls to **`F(n-4)`**

Characterization of the Poor Efficiency

- Anticipated by the nature of its recurrence
- It contains two recursive calls with sizes of smaller instances only slightly smaller than *n*
- Look at the recursive calls in Figure 2.6
- Same values of the function are being evaluated again and again – ***extremely inefficient!!!***

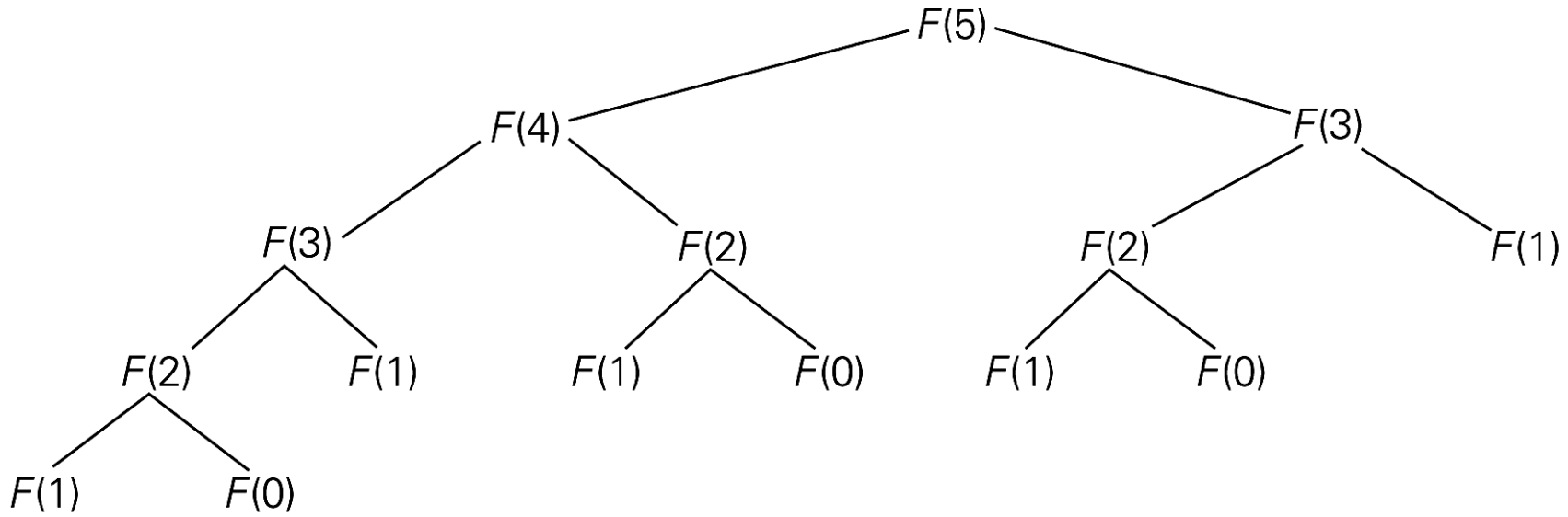


FIGURE 2.6 Tree of recursive calls for computing the 5th Fibonacci number by the definition-based algorithm

Iterative Algorithms for Computing Fibonacci Numbers

ALGORITHM *Fib*(n)

//Computes the nth Fibonacci number iteratively by using its definition
//Input: A nonnegative integer n
//Output: The nth Fibonacci number
$F[0] \leftarrow 0;\ F[1] \leftarrow 1$
for $i \leftarrow 2$ **to** n **do**
    $F[i] \leftarrow F[i-1] + F[i-2]$
return $F[n]$

- Basic operation: ***addition***
- Number of additions: ***n*-1**
- Hence, ***linear*** as a function of n.

Other Algorithms for Computing Fibonacci Numbers

- There exists an $\Theta(logn)$ algorithm to calculate n-th Fibonacci numbers. It is based on the equality below

$$\begin{bmatrix} F(n-1) & F(n) \\ F(n) & F(n+1) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^n \quad \text{for } n \geq 1$$