Module-I

Introduction
to
Mathematical Problem Solving

By

Ms. Sasmita Kumari Nayak

Unit 1- TOPICS

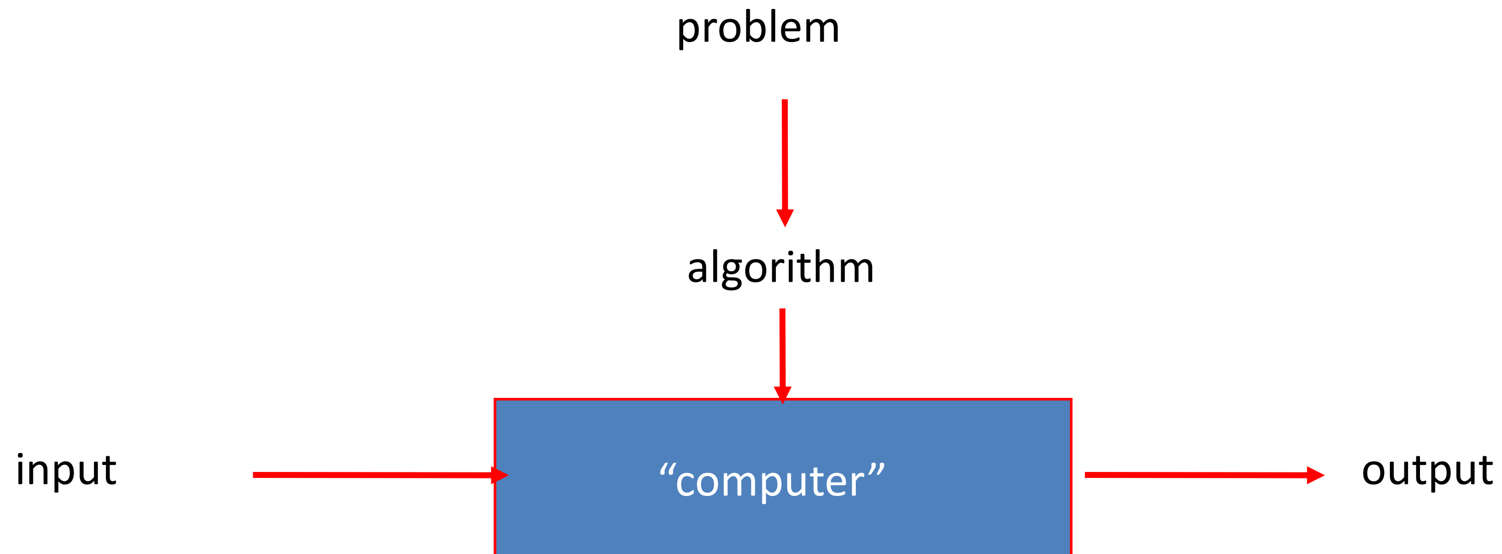
- Notion of Algorithm
- Review of Asymptotic Notations
- Mathematical Analysis of Non-Recursive and Recursive Algorithms
- Brute Force Approaches: Introduction
- Selection Sort and Bubble Sort
- Sequential Search and Brute Force String Matching

ALGORITHM

- An algorithm is an exact specification of how to solve a computational problem
- An algorithm must specify every step completely, so a computer can implement it without any further "understanding"
- An algorithm must work for all possible inputs of the 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
- There can be many different algorithms for each computational problem.

WHAT IS AN ALGORITHM?

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



NOTION OF ALGORITHM

- Euclid’s algorithm

  Step 1 If $n = 0$, return m and stop; otherwise go to Step 2

  Step 2 Divide m by n and assign the value of the remainder to r

  Step 3 Assign the value of n to m and the value of r to n. Go to Step 1.

```
while n ≠ 0 do
    r ← m mod n
    m← n
    n ← r
return m
```

Cont…

- Problem: Find gcd(m,n), the greatest common divisor of two non-negative, not both zero integers m and n

- Examples: gcd(60,24) = 12, gcd(60,0) = 60

- Euclid's algorithm is based on repeated application of equality gcd(m,n) = gcd(n, m mod n) until the second number becomes 0, which makes the problem trivial.

- Example: gcd(60,24) = gcd(24,12) = gcd(12,0) = 12

Other methods for computing gcd(m,n)

- Consecutive integer checking algorithm

  Step 1 Assign the value of min{m,n} to t

  Step 2 Divide m by t. If the remainder is 0, go to Step 3; otherwise, go to Step 4

  Step 3 Divide n by t. If the remainder is 0, return t and stop; otherwise, go to Step 4

  Step 4 Decrease t by 1 and go to Step 2

Cont…

- Middle-school procedure

Step 1 Find the prime factorization of m

Step 2 Find the prime factorization of n

Step 3 Find all the common prime factors

Step 4 Compute the product of all the common prime factors and return it as gcd(m,n)

Sieve of Eratosthenes

Input: Integer $n \geq 2$

Output: List of primes less than or equal to n

```
for p ← 2 to n do  A[p] ← p
for p ← 2 to n do
        if A[p] != 0  //p hasn't been previously eliminated from the list
    j ← p* p
      while j ≤ n  do
          A[j] ← 0  //mark element as eliminated
          j ← j + p
```

Example: 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

2 3 5 7 9 11 13 15 17 19

2 3 5 7 11 13 17 19

2 3 5 7 11 13 17 19

SOME OF THE IMPORTANT POINTS

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

HOW DO WE COMPARE 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.

Example

Associate a "cost" with each statement.

Find the "total cost“ by finding the total number of times each statement is executed.

Algorithm 1	Cost	Algorithm 2	Cost
arr[0] = 0;	c1	for(i=0; i<N; i++)	c2
arr[1] = 0;	c1	arr[i] = 0;	c1
arr[2] = 0;	c1		
...	...		
arr[N-1] = 0;	c1		
------------------		---------------------	
c1+c1+...+c1 = c1 x N		(N+1) x c2 + N x c1 = (c2 + c1) x N + c2	

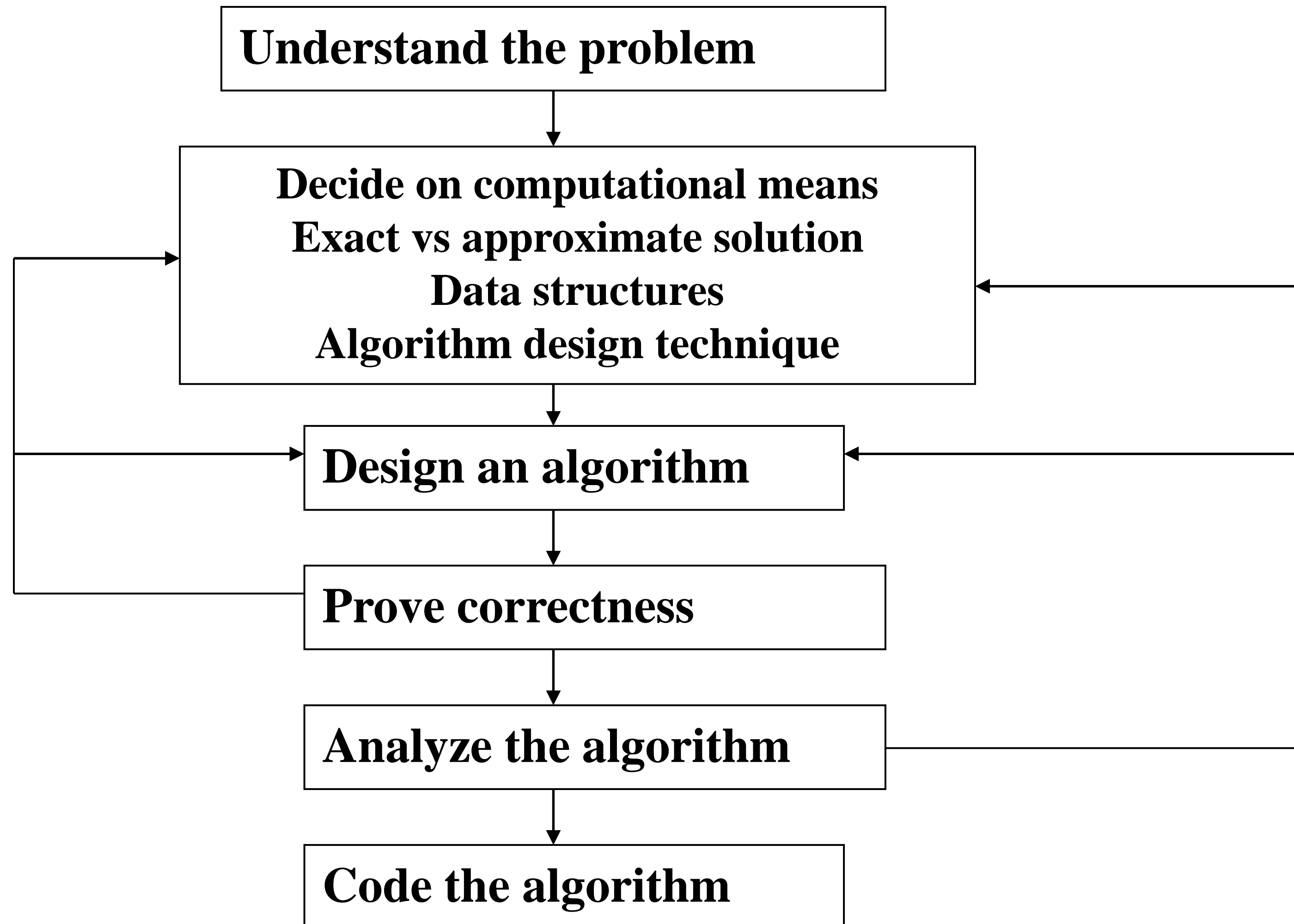
Another Example

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
                        ------------
```

c1 + c2 x (N+1) + c2 x N x (N+1) + c3 x N2

Fundamentals of Algorithmic Problem solving



ANALYSIS OF ALGORITHMS

- How good is the algorithm?
    - Correctness
    - Time efficiency
    - Space efficiency
- Does there exist a better algorithm?
    - Lower bounds
    - Optimality

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.
- Hint: use rate of growth
- Compare functions in the limit, that is, asymptotically!
- (i.e., for large values of n)

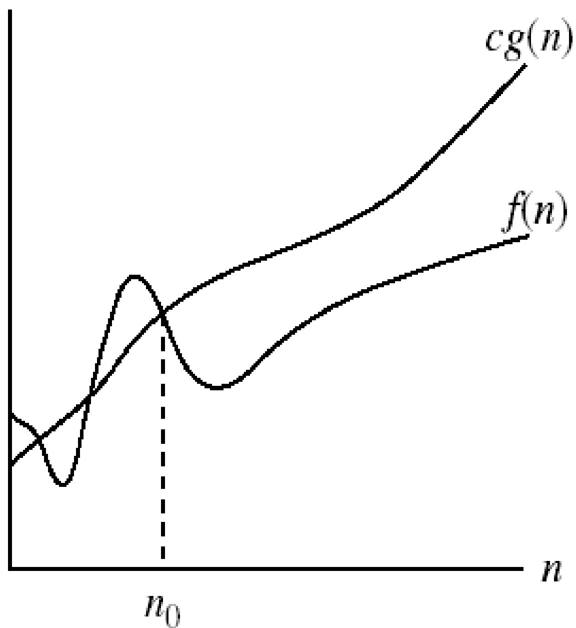
ASYMPTOTIC NOTATION

- **O** notation: asymptotic "less than":
  - f(n)=O(g(n)) implies: f(n) "≤" g(n)
- **Ω** notation: asymptotic "greater than":
  - f(n)= Ω (g(n)) implies: f(n) "≥" g(n)
- **θ** notation: asymptotic "equality":
  - f(n)= θ (g(n)) implies: f(n) "=" g(n)

ASYMPTOTIC NOTATIONS

O-notation

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



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

Examples

- f(n)=3n2+n

  $= 3n^2+n^2$

  $=4n^2$

n2 ≤ cn2 ; c ≥ 1 ; c = 1 and n0= 1

- f(n)<=c*g(n)

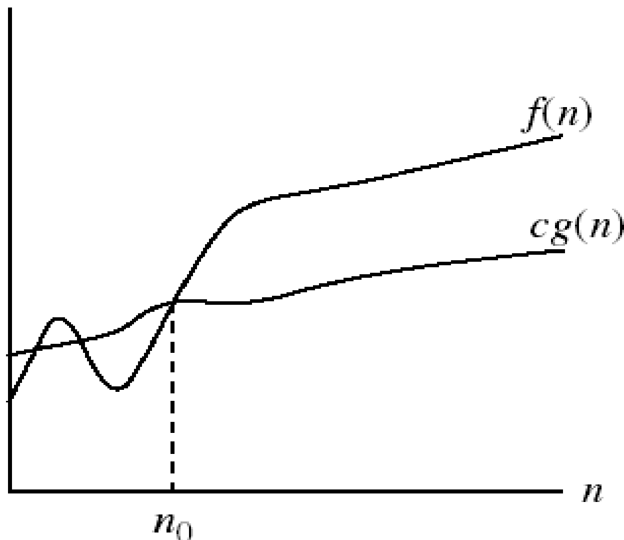
$3n^2+n<=4n^2=o(n^2)$

Where $n>=n_0$ and n=1

ASYMPTOTIC NOTATIONS (CONT.)

Ω -notation

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



Ω(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 $f(n)$.

Examples

- f(n)=3n2+n

  $= 3n^2+n$

  $=3n^2$

- f(n)>=c*g(n)

$3n^2+n>=3n^2= \mathbf{\Omega}(n^2)$

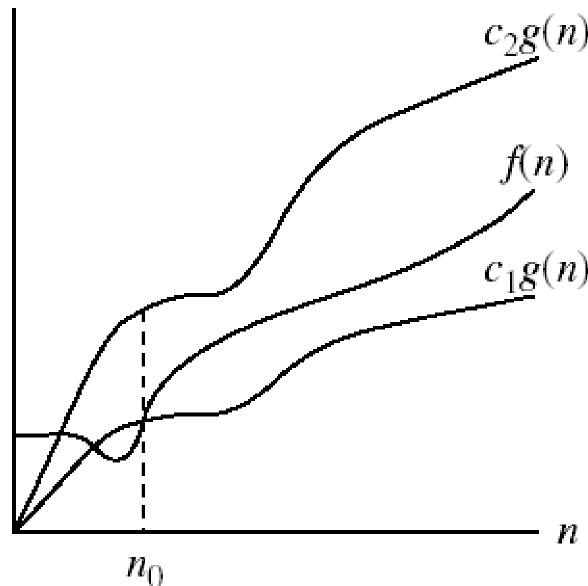
Where $n<=n_0$ and n=1

ASYMPTOTIC NOTATIONS (CONT.)

θ -notation

$$\Theta(g(n)) = \{ f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0 \} .$$

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 $f(n)$.

Examples

$C2g(n) <= f(n) <= c1g(n)$ for all $n >= n0$

$3n2 + n <= f(n) <= 3n2 + n2$

$3n2 <= f(n) <= 4n2$

Where $c2=3$, $c1=4$ and $n=1$

Therefore, $3n2+n \in \theta(n2)$

Establishing order of growth using limits

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

Examples:

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

L'Hôpital's rule and Stirling's formula

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

Stirling's formula: $n! \approx (2\pi n)^{1/2} (n/e)^n$

Example: 2^n vs. $n!$

Ms. Sasmita Kumari Nayak

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: $a_k n^k + a_{k-1} n^{k-1} + ... + a_0 \in \Theta(n^k)$

- Exponential functions a^n have different orders of growth for different a's

- order $\log n$ < order n^{α} (α>0) < order a^n < order $n!$ < order n^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**

MATHEMATICAL ANALYSIS OF NO RECURSIVE 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

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
```

$C(n) \in \Theta(n)$

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

$C(n) \in \Theta(n^2)$

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 ← 0 to n − 1 do
        for j ← 0 to n − 1 do
            C[i, j] ← 0.0
            for k ← 0 to n − 1 do
                C[i, j] ← C[i, j] + A[i, k] ∗ B[k, j]
    return C
```

$C(n) \in \Theta(n^3)$

MATHEMATICAL 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: $n\,! = 1 * 2 * \ldots *(n\text{-}1) * n$ for $n \geq 1$ and $0! = 1$

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

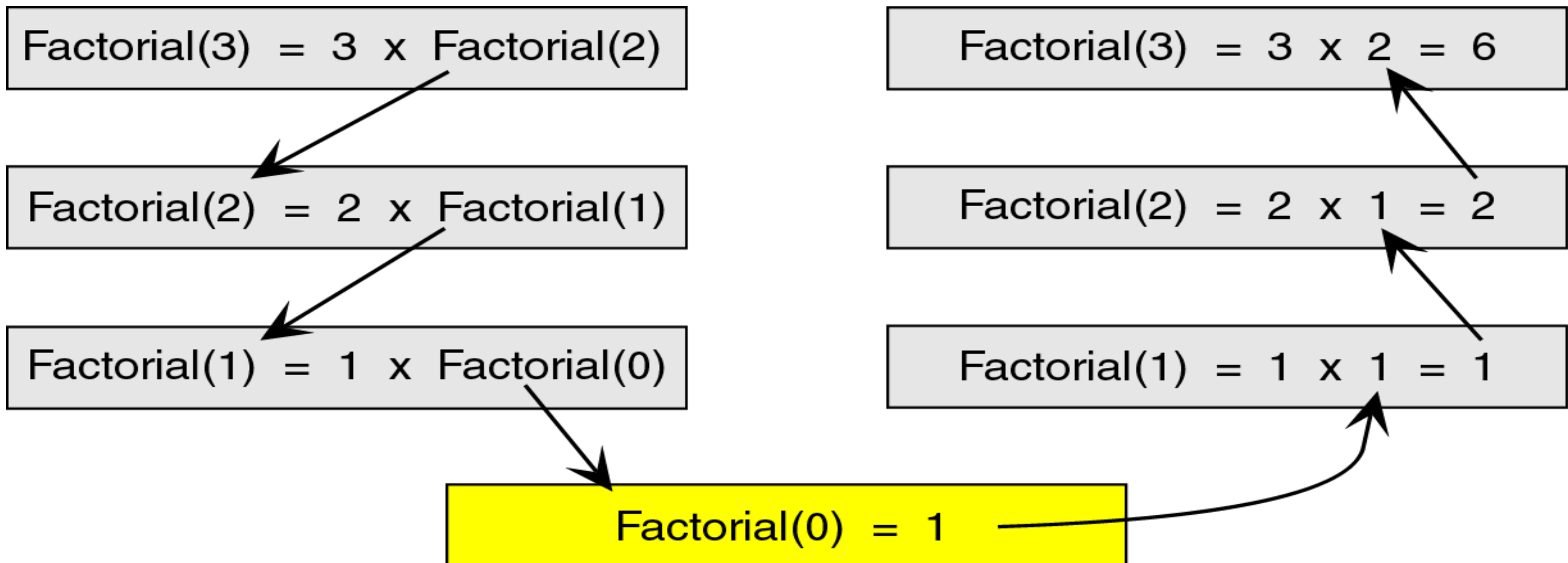
$$F(0) = 1$$

ALGORITHM $F(n)$

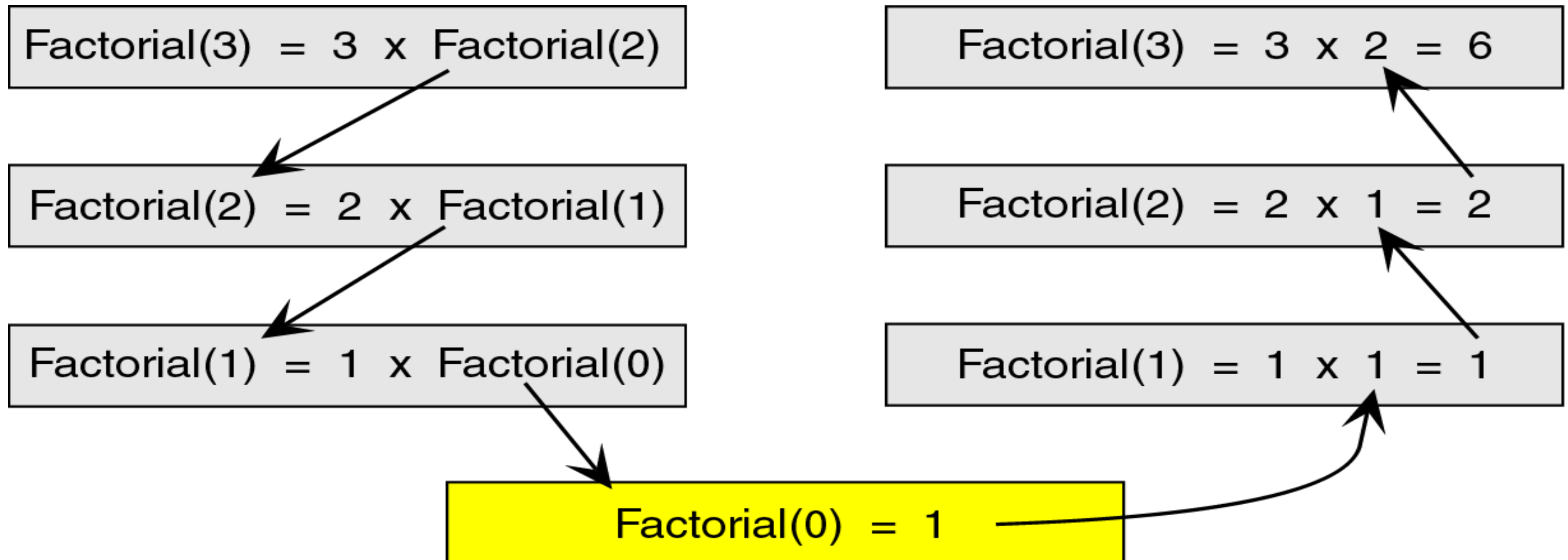
//Computes n! recursively
//Input: A nonnegative integer n
//Output: The value of n!
if $n = 0$ **return** 1
else return $F(n-1) * n$

Recursion

- To see how the recursion works, let's break down the factorial function to solve factorial(3)

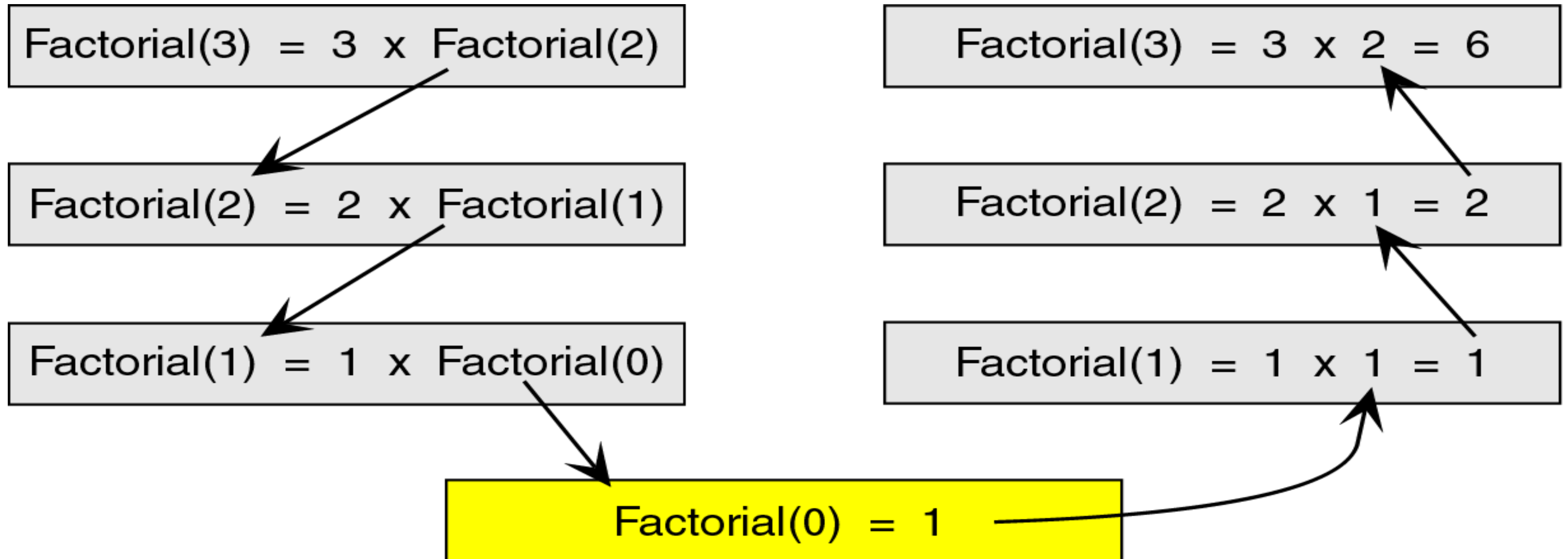


Breakdown



- Here, we see that we start at the top level, factorial(3), and simplify the problem into 3 x factorial(2).
- Now, we have a slightly less complicated problem in factorial(2), and we simplify this problem into 2 x factorial(1).

Breakdown



- We continue this process until we are able to reach a problem that has a known solution.
- In this case, that known solution is factorial(0) = 1.
- The functions then return in reverse order to complete the solution.

Analysis of Factorial

- Recurrence Relation

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

  $M(0) = 0$

- Solve by the method of *backward substitutions*

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

  $= [M(n\text{-}2) + 1] + 1 = M(n\text{-}2) + 2$ substituted $M(n\text{-}2)$ for $M(n\text{-}1)$

  $= [M(n\text{-}3) + 1] + 2 = M(n\text{-}3) + 3$ substituted $M(n\text{-}3)$ for $M(n\text{-}2)$

  .. a pattern evolves

  $= M(0) + n$

  $= n$

Therefore $M(n) \in \Theta(n)$

Example 2: 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 $BinRec(\lfloor n/2 \rfloor) + 1$

Analysis of Counting # of bits

Recursive relation including initial conditions

A(n) = A(floor(n/2)) + 1

IC A(1) = 0

substitute n = 2k (also k = lg(n))

A(2k) = A(2k-1) + 1 and IC A(20) = 0

A(2k) = [A(2k-2) + 1] + 1 = A(2k-2) + 2

= [A(2k-3) + 1] + 2 = A(2k-3) + 3

...

= A(2k-i) + i

...

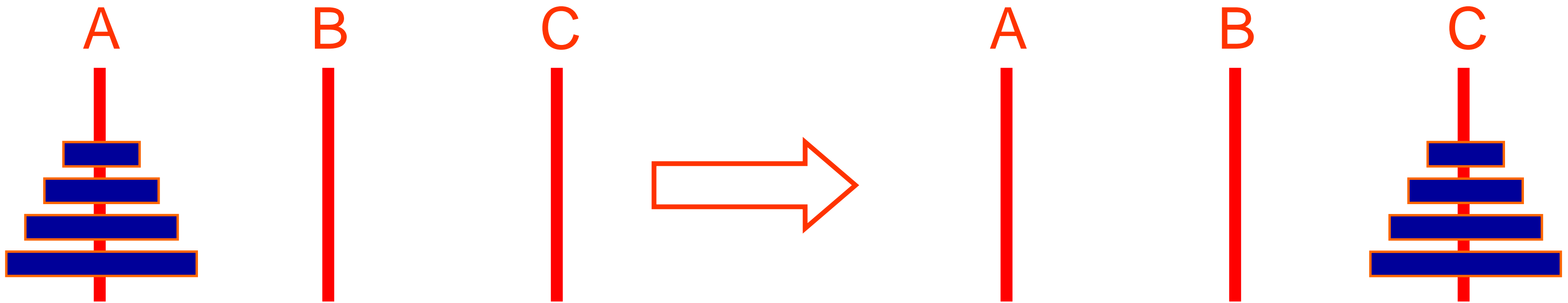
= A(2k-k) + k

A(2k) = k

Substitute back k = lg(n)

A(n) = lg(n) ε Θ(lg n)

Example 3: Tower of Hanoi



Given: Three Pegs A, B and C

Peg A initially has n disks, different size, stacked up, larger disks are below smaller disks

Problem: to move the n disks to Peg C, subject to

1. Can move only one disk at a time
2. Smaller disk should be above larger disk
3. Can use other peg as intermediate

Tower of Hanoi

- How to Solve: Strategy…
  - Generalize first: Consider n disks for all $n \geq 1$
  - Our example is <u>*only*</u> the case when n=4
- Look at small instances…
  - How about n=1
    - Of course, just "Move disk 1 from A to C"
  - How about n=2?
    1. "Move disk 1 from A to B"
    2. "Move disk 2 from A to C"
    3. "Move disk 1 from B to C"

Tower of Hanoi (Solution!)

- General Method:
  - First, move first (n-1) disks from A to B
  - Now, can move largest disk from A to C
  - Then, move first (n-1) disks from B to C
- Try this method for n=3

1. "Move disk 1 from A to C"
2. "Move disk 2 from A to B"
3. "Move disk 1 from C to B"

4. "Move disk 3 from A to C"

5. "Move disk 1 from B to A"
6. "Move disk 1 from B to C"
7. "Move disk 1 from A to C"

Algorithm for Towel of Hanoi (recursive)

- Recursive Algorithm
  - when (n=1), we have simple case
  - Else (decompose problem and make recursive-calls)

```
Hanoi(n, A, B, C);
(* Move n disks from A to C via B *)
begin
  if (n=1) then "Move top disk from A to C"
  else (* when n>1 *)
    Hanoi (n-1, A, C, B);
    "Move top disk from A to C"
    Hanoi (n-1, B, C, A);
  endif
end;
```

Analysis of Tower of Hanoi

Recursive relation for moving n discs

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

IC: $M(1) = 1$

Solve using backward substitution

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

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

$=2^2[2M(n\text{-}3) +1] + 2+1 = 2^3M(n\text{-}3) + 2^2 + 2 + 1$

..

$M(n) = 2^iM(n\text{-}i) + \sum_{j=0}{}^{-i}2^j = 2^iM(n\text{-}i) + 2^i\text{-}1$

...

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

$M(n)\ \varepsilon\ \Theta(2^n)$

Iteration vs. Recursion

- After looking at both iterative and recursive methods, it appears that the recursive method is much longer and more difficult.
- If that's the case, then why would we ever use recursion?
- It turns out that recursive techniques, although more complicated to solve by hand, are very simple and elegant to implement in a computer.

BRUTE FORCE

- A straightforward approach, usually based directly on the problem's statement and definitions of the concepts involved
- Usually can solve small sized instances of a problem
- A yardstick to compare with more efficient ones

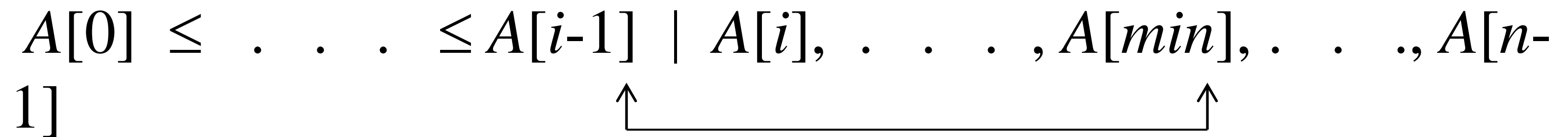
Examples:

- Computing an (a > 0, n a nonnegative integer)
- Computing n!
- Multiplying two matrices
- Searching for a key of a given value in a list

BRUTE-FORCE SORTING ALGORITHM

<u>*Selection Sort*</u> Scan the array to find its smallest element and swap it with the first element. Then, starting with the second element, scan the elements to the right of it to find the smallest among them and swap it with the second elements. Generally, on pass i ($0 \le i \le n$-2), find the smallest element in $A[i..n-1]$ and swap it with $A[i]$:

$A[0] \le \ . \ . \ . \ \le A[i-1] \mid A[i], \ . \ . \ . \ , A[min], \ . \ . \ ., A[n-1]$



in their final positions

Selection Sort Algorithm

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

```
//Sorts a given array by selection sort
//Input: An array A[0..n − 1] of orderable elements
//Output: Array A[0..n − 1] sorted in ascending order
for i ← 0 to n − 2 do
    min ← i
    for j ← i + 1 to n − 1 do
        if A[j] < A[min]   min ← j
    swap A[i] and A[min]
```

Analysis of Selection Sort

| 89 45 68 90 29 34 **17**

17 | 45 68 90 **29** 34 89

17 29 | 68 90 45 **34** 89

17 29 34 45 | 90 **68** 89

17 29 34 45 68 | 90 **89**

17 29 34 45 68 89 | 90

C(n) ϵ Θ(n²)
# of key swaps ϵ Θ(n)

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$$

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

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

BUBBLE SORT

- Compare adjacent elements and exchange them if out of order
- Essentially, it bubbles up the largest element to the last position

$$A_0, \ldots \ldots, A_j \overset{?}{<->} A_{j+1}, \ldots \ldots, A_{n-i-1} \mid A_{n-i} \leq \ldots \leq A_{n-1}$$

Cont…

```
ALGORITHM BubbleSort(A[0..n-1])
for i <- 0 to n-2 do
        for j <- 0 to n-2-i do
                if A[j+1] < A[j]
                        swap A[j] and A[j+1]
```

Example : 89, 45, 68, 90, 29, 34, 17

$C(n) \in \Theta(n^2)$ $S_{worst}(n) = C(n)$

SEQUENTIAL SEARCH

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

//Output: index of the first element in A, whose //value is equal to K or -1 if no such element is found

i <- 0

while i < n **and** A[i] ≠ K **do**

    i <- i+1

if i < n

    return i

else

    return -1

Input size: n

Basic op: <, ≠

$C_{worst}(n) = n$

BRUTE-FORCE STRING MATCHING

- *pattern*: a string of *m* characters to search for
- *text*: a (longer) string of *n* characters to search in
- problem: find a substring in the text that matches the pattern

Brute-force algorithm

Step 1 Align pattern at beginning of text

Step 2 Moving from left to right, compare each character of pattern to the corresponding character in text until

- all characters are found to match (successful search); or
- a mismatch is detected

Step 3 While pattern is not found and the text is not yet exhausted, realign pattern one position to the right and repeat Step 2

Pseudocode and Efficiency

ALGORITHM *BruteForceStringMatch*$(T[0..n-1], P[0..m-1])$

//Implements brute-force string matching
//Input: An array $T[0..n-1]$ of n characters representing a text and
//   an array $P[0..m-1]$ of m characters representing a pattern
//Output: The index of the first character in the text that starts a
//   matching substring or -1 if the search is unsuccessful
for $i \leftarrow 0$ **to** $n - m$ **do**
    $j \leftarrow 0$
    while $j < m$ **and** $P[j] = T[i + j]$ **do**
        $j \leftarrow j + 1$
    if $j = m$ **return** i
return -1

Time efficiency: **Θ(mn) comparisons (in the worst case)**