Nadar Saraswathi College of Engineering & Technology

NSCET E-LEARNING PRESENTATION

LISTEN … LEARN… LEAD…

COMPUTER SCIENCE AND ENGINEERING

II YEAR / IV SEMESTER

CS8451 – DESIGN AND ANALYSIS OF ALGORITHMS

S ARUL JOTHI M.E.,MISTE,

ASSISTANT PROFESSOR

Nadar Saraswathi College of Engineering & Technology,

Vadapudupatti, Annanji (PO), Theni – 625531.

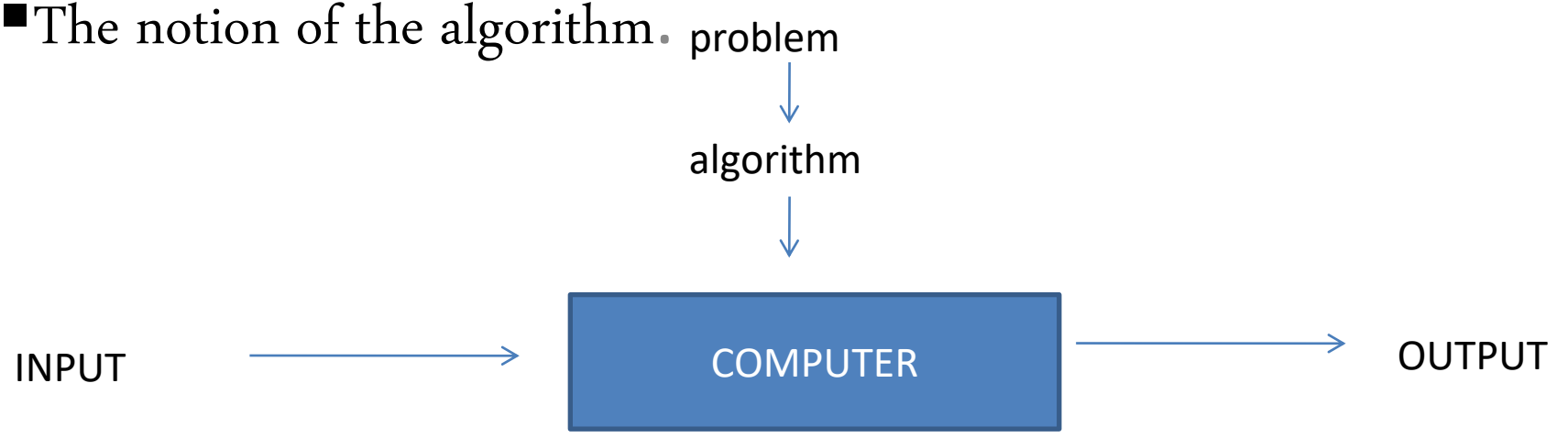
UNIT 1-INTRODUCTION

- ❑ Notion of an Algorithm
- ❑ Fundamentals of Algorithmic Problem Solving
- ❑ Important Problem Types
- ❑ The Analysis Framework
- ❑ Asymptotic Notations and Basic Efficiency Classes
- ❑ Mathematical Analysis of Non recursive & recursive Algorithms
- ❑ Empirical Analysis
- ❑ Algorithm Visualization

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.

- The notion of the algorithm.



Euclid's algorithm for computing gcd(m, n)

Step 1 If n = 0, return the value of m as the answer and stop; otherwise, proceed 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.

ALGORITHM Euclid(m, n)

//Computes gcd(m, n) by Euclid's algorithm //Input: Two nonnegative, not-both-zero integers m and n

//Output: Greatest common divisor of m and n while n = 0 do

r ⟵ m mod n

m ⟵ n

n ⟵ r return m

Consecutive integer checking algorithm for computing gcd(m, n)

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

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

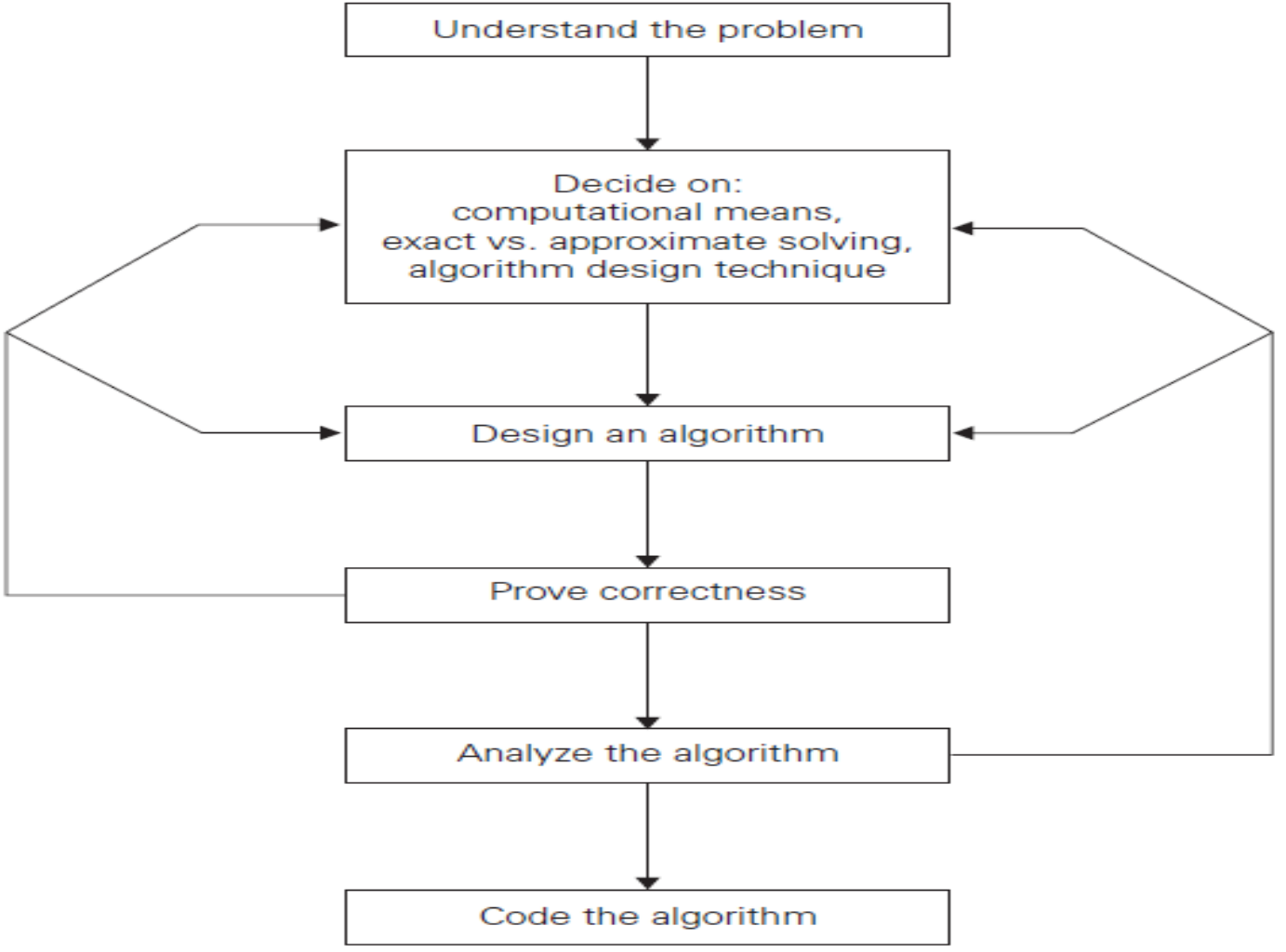
Step 3 Divide n by t. If the remainder of this division is 0, return the value of t as the answer and stop; otherwise, proceed to Step 4.

Step 4 Decrease the value of t by 1. Go to Step 2

Fundamentals of Algorithmic Problem Solving

- Understanding the Problem
- Ascertaining the Capabilities of the Computational Device
- Choosing between Exact and Approximate Problem Solving
- Algorithm Design Techniques
- Designing an Algorithm and Data Structures
- Methods of Specifying an Algorithm
- Proving an Algorithm's Correctness
- Analyzing an Algorithm
- Coding an Algorithm

ALGORTHMIC PROBLEM SOLVING STEPS



Important Problem Types

- Sorting
- Searching
- String processing
- Graph problems
- Combinatorial problems
- Geometric problems
- Numerical problems

Fundamentals of the Analysis of Algorithm Efficiency

The Analysis Framework:

Two kinds of efficiency

(i)Time efficiency, also called time complexity,

-indicates how fast an algorithm in question runs.

(ii)Space efficiency, also called space complexity,

-refers to the amount of memory units required by the algorithm in addition to the space needed for its input and output.

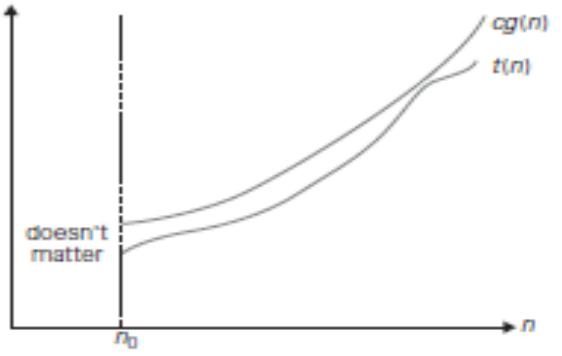
The Analysis Framework

- Measuring an Input's Size
- Units for Measuring Running Time
- Orders of Growth
- Worst-Case, Best-Case, and Average-Case Efficiencies

Asymptotic Notations and Basic Efficiency Classes

*(i)O-**notation***

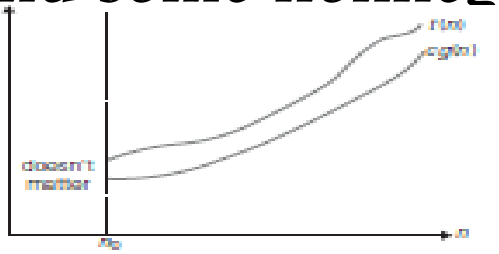
- **DEFINITION A function *t (n) is said to be in O(g(n)), denoted t (n) ∈ O(g(n)),*** if *t (n) is bounded above by some constant multiple of g(n) for all large n, i.e., if* there exist some positive constant *c and some nonnegative integer n0 such that t (n) ≤*



Asymptotic Notations

(ii) Ω –notation

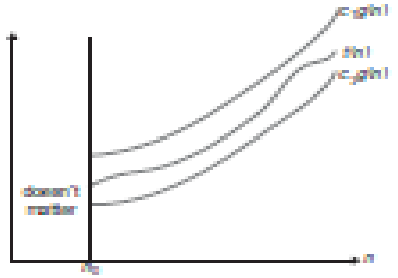
DEFINITION A function *t (n) is said to be in (g(n)), denoted t (n) ∈ (g(n)), if* *t (n) is bounded below by some positive constant multiple of g(n) for all large n* i.e., if there exist some positive constant *c and some nonnegative integer n0 such* that *t (n)* ≥ ‹



Asymptotic Notations

(iii)Θ-notation

DEFINITION A function *t (n) is said to be in (g(n)), denoted t (n) ∈ (g(n)),* if *t (n) is bounded both above and below by some positive constant multiples of g(n) for all large n, i.e., if there exist some positive constants c1 and c2 and some* nonnegative integer *n0 such that c2g(n) ≤ t (n) ≤ c1g(n) for a*



Mathematical Analysis of Non recursive Algorithms

Consider the problem of finding the value of the largest element in a list of *n numbers.*

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*

Mathematical Analysis of Non recursive Algorithms

- Let us denote *C(n) the number of times this comparison is executed and* to find a formula expressing it as a function of size *n.*

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

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

General Plan for Analyzing the Time Efficiency of Nonrecursive Algorithms

1. **Decide on a parameter (or parameters) indicating an input's size.**
2. **Identify the algorithm's basic operation.**
3. **Check whether the number of times the basic operation is executed depends** only on the size of an input. If it also depends on some additional property,the worst-case, average-case, and, if necessary, best-case efficiencies have to be investigated separately.
4. **Set up a sum expressing the number of times the algorithm's basic operation** is executed.
5. **Using standard formulas and rules of sum manipulation, either find a closed form** formula for the count or, at the very least, establish its order of growth.

Mathematical Analysis of Recursive Algorithms

Ex.1.Compute the factorial function *F(n) = n! for an arbitrary non negative* integer *n.*

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*

Mathematical Analysis of Recursive Algorithms

Since the function *F(n) is computed according to the formula F(n) = F(n – 1) . n for n > 0,* the number of multiplications *M(n) needed to compute it must satisfy the equality*

$M(n) = M(n-1)_{\text{to compute } F(n-1)} + \underbrace{1}_{\text{to multiply } F(n-1) \text{ by } n}$

for $n > 0$

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

Mathematical Analysis of Recursive Algorithms

it is defined by the recurrence

- $F(n) = F(n - 1) . n$ for every $n > 0, F(0) = 1.$
- $M(n) = M(n - 1) + 1$ substitute $M(n - 1) = M(n - 2) + 1$
- $= [M(n - 2) + 1] + 1 = M(n - 2) + 2$ substitute $M(n - 2) = M(n - 3) + 1$
- $= [M(n - 3) + 1] + 2 = M(n - 3) + 3.$
- $M(n) = M(n - 1) + 1 = \ldots = M(n - i) + i = \ldots = M(n - n) + n = n.$

General Plan for Analyzing the Time Efficiency of Recursive Algorithms

1. Decide on a parameter (or parameters) indicating an input's size.
2. Identify the algorithm's basic operation.
3. Check whether the number of times the basic operation is executed can vary on different inputs of the same size; if it can, the worst-case, average-case, and best-case efficiencies must be investigated separately.
4. Set up a recurrence relation, with an appropriate initial condition, for the number of times the basic operation is executed.
5. Solve the recurrence or, at least, ascertain the order of growth of its solution.

Ex.2 Recursive Algorithm to the Tower of Hanoi puzzle.

The recurrence equation is $M(n) = 2M(n - 1) + 1$ for $n > 1$, with

Initial condition $M(1) = 1$.

By the method of backward substitutions

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

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

$= 2^{n-1}$.

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

Ex 3.Algorithm for Conversion of Integer n into Binary digits

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(n/2) + 1***

Convert Integer n into Binary digits

The recurrence equation is

$A(n) = A(n/2) + 1$ *for* $n > 1$ *with initial condition* $A(1) = 0$.

By the method of backward substitutions

$A(2^k) = A(2^{k-1}) + 1$ *for* $k > 0$, $A(2^{0)} = 0$.

substitute $A(2^{k-1}) = A(2^{k-2}) + 1$

$= A(2^{k-i}) + i$

$. . . . . = A(2^{k-k}) + k.$

End up with $A(2^k) = A(1) + k = k$ after returning to the original variable

$n = 2^k$ *and hence* $k = \log_2 n$,

- $A(n) = \log_2 n \in (\log n)$.

Computing the *nth Fibonacci Number*

Fibonacci numbers, a famous sequence

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

that can be defined by the simple recurrence

$F(n) = F(n - 1) + F(n - 2)$ *for* $n > 1$ and two initial conditions

$F(0) = 0, F(1) = 1.$

Algorithm for Fibonacci sequence

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 $F(n-1) + F(n-2)$

Algorithm for Fibonacci sequence

The following recurrence for $A(n)$:

$A(n) = A(n-1) + A(n-2) + 1$ for $n > 1$, $A(0) = 0$, $A(1) = 0$.

$[A(n) + 1] - [A(n-1) + 1] - [A(n-2) + 1] = 0$

and substituting $B(n) = A(n) + 1$:

$B(n) - B(n-1) - B(n-2) = 0$,

$B(0) = 1$, $B(1) = 1$

- $A(n) = B(n) - 1 = F(n+1) - 1$

  $= 1/\sqrt{5}(\varphi^{n+1} - \varphi'^{n+1}) - 1 = A(n) \in \Theta(\varphi^n) = \Theta(2^b)$.

Empirical Analysis of Algorithms

General Plan for the Empirical Analysis of Algorithm Time Efficiency

1. Understand the experiment's purpose.
2. Decide on the efficiency metric *M to be measured and the measurement unit* (an operation count vs. a time unit).
3. Decide on characteristics of the input sample (its range, size, and so on).
4. Prepare a program implementing the algorithm (or algorithms) for the experimentation.

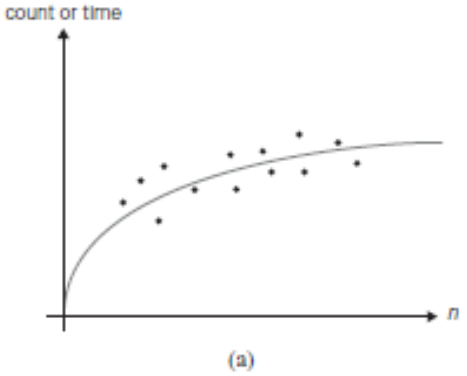
Empirical Analysis of Algorithms

5.Generate a sample of inputs.

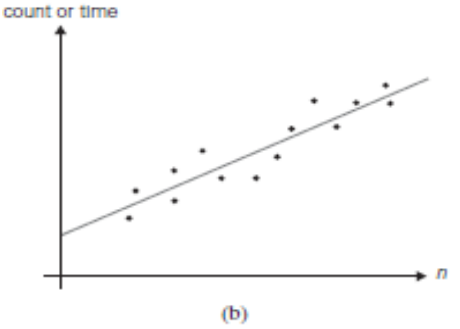
6. Run the algorithm (or algorithms) on the sample's inputs and record the data observed.

7. Analyze the data obtained

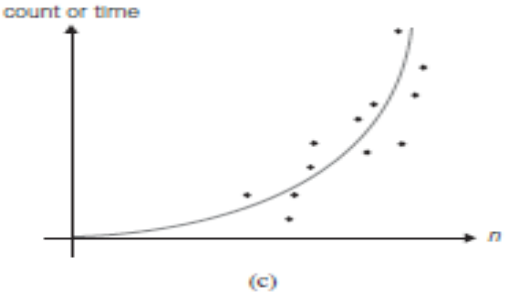
Typical Scatter plots



Logarithmic



Linear



One of the Convex functions

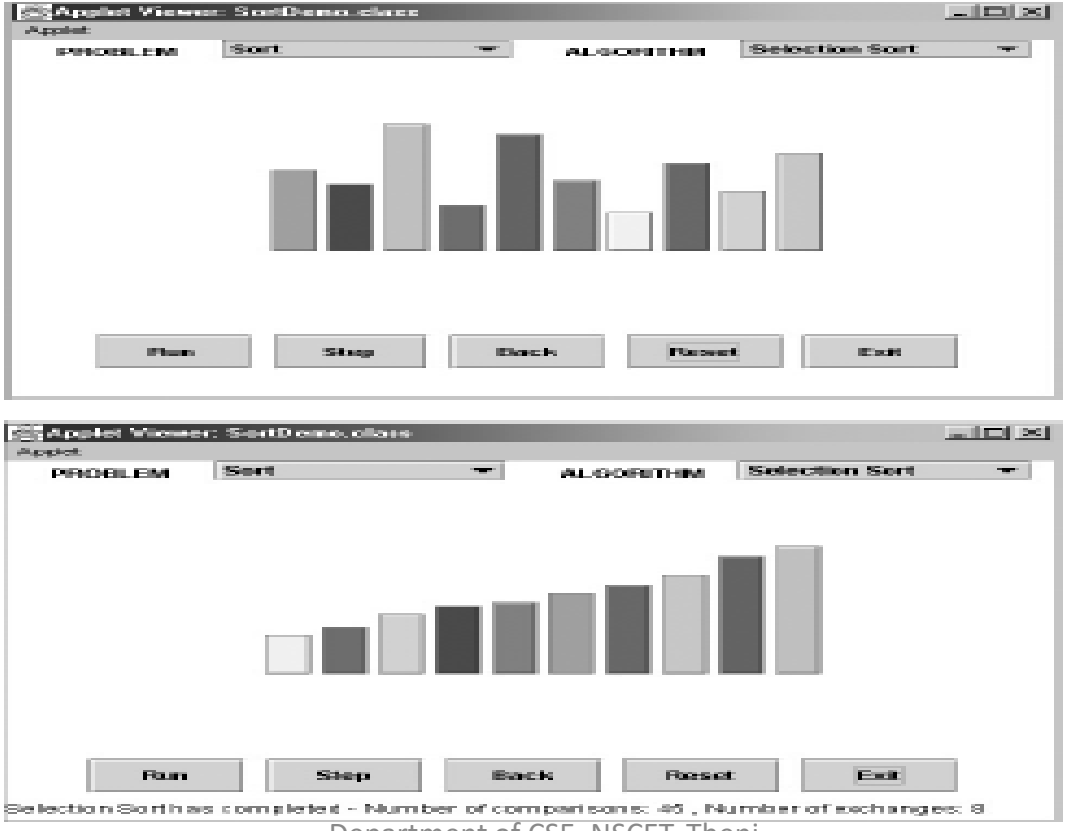
Algorithm Visualization

-can be defined as the use of images to convey some useful information about algorithms.

There are two principal variations of algorithm visualization:

- Static algorithm visualization
- Dynamic algorithm visualization, also called ***algorithm animation***

Visualization of sorting algorithm using the bar representation



Visualization of sorting algorithm using the scatterplot representation