Design and Analysis of Algorithms

Vandana M L
Department of Computer Science & Engineering

DESIGN AND ANALYSIS OF ALGORITHMS

Introduction to Algorithms, Design Techniques and Analysis

Slides courtesy of **Anany Levitin**

Vandana M L

Department of Computer Science & Engineering

Syllabus

- **UNIT I (12 Hours)**
  - Introduction
  - Analysis of Algorithm Efficiency,
  - Algebric structures
- **UNIT II (12 Hours)**
  - Brute Force,
  - Divide-and-Conquer
- **UNIT III (10 Hours)**
  - Decrease-and-Conquer
  - Transform-and-Conquer
- **UNIT IV (10 Hours)**
  - Space and Time Tradeoffs
  - Greedy Technique
- **UNIT V (12 Hours)**
  - Limitations of Algorithm Power
  - Coping with the Limitations of Algorithm Power
  - Dynamic Programming

Design and Analysis of Algorithms

Text Books

Book Type	Code	Title & Author	Publication Information		
			Edition	Publisher	Year
Text Book	T1	Introduction to The Design and Analysis of Algorithms Anany Levitin	2	Pearson	2012
Reference Book	R1	Introduction to Algorithms Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein	3	Prentice-Hall India	2009
Reference Book	R2	Fundamentals of Computer Algorithms Horowitz, Sahni, Rajasekaran,	2	Universities Press	2007
Reference Book	R3	Algorithm Design Jon Kleinberg, Eva Tardos,	1	Pearson Education	2006

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

Important Points about Algorithms

- 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 implemented in several different ways
- There may exist several algorithms for solving the same problem.

Characteristics of Algorithm

Input: Zero or more quantities are externally supplied

Definiteness: Each instruction is clear and unambiguous

Finiteness: The algorithm terminates in a finite number of steps.

Effectiveness: Each instruction must be primitive and feasible

Output: At least one quantity is produced

Why do we need Algorithms?

- It is a tool for solving well-specified Computational Problem.
- Problem statement specifies in general terms relation between input and output
- Algorithm describes computational procedure for achieving input/output relationship
  This Procedure is irrespective of implementation details

Why do we need to study algorithms?

Exposure to different algorithms for solving various problems helps develop skills to design algorithms for the problems for which there are no published algorithms to solve it

Basic Issues related to Algorithms

- How to design algorithms
- How to express algorithms
- Proving correctness of designed algorithm
- Efficiency
  - Theoretical analysis
  - Empirical analysis

Basic Issues related to Algorithms

- How to design algorithms
- How to express algorithms
- Proving correctness of designed algorithm
- Efficiency
    - Theoretical analysis
    - Empirical analysis

What do you mean by Algorithm Design Techniques?

General Approach to solving problems algorithmically .
Applicable to a variety of problems from different areas of computing

Various Algorithm Design Techniques

- Brute Force
- Divide and Conquer
- Decrease and Conquer
- Transform and Conquer
- Dynamic Programming
- Greedy Technique
- Branch and Bound
- Backtracking

Importance Framework for designing and analyzing algorithms
for new problems

Basic Issues related to Algorithms

- How to design algorithms
- How to express algorithms
- Proving correctness of designed algorithm
- Efficiency
  - Theoretical analysis
  - Empirical analysis

Methods of Specifying an Algorithm

- **Natural language**
  - Ambiguous
- **Pseudocode**
  - A mixture of a natural language and programming language-like structures
  - Precise and succinct.
  - Pseudocode in this course
    - omits declarations of variables
    - use indentation to show the scope of such statements as for, if, and while.
    - use ← for assignment
- **Flowchart**
  - Method of expressing algorithm by collection of connected geometric shapes

➢ Euclid’s Algorithm

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

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

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

Two descriptions of Euclid's algorithm

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

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.

```
ALGORITHM Euclid(m,n)
//computes gcd(m,n) by Euclid's method
//Input: Two nonnegative,not both zero integers
//Output:Greatest common divisor of m and n
while n ≠ 0 do
    r ← m mod n
    m ← n
    n ← r
return m
```

Algorithm for Sequential Search

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

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

Basic Issues related to Algorithms

- How to design algorithms
- How to express algorithms
- Proving correctness of designed algorithm
- Efficiency
  - Theoretical analysis
  - Empirical analysis

Basic Issues related to Algorithms

- How to design algorithms
- How to express algorithms
- Proving correctness of designed algorithm
- Efficiency
    - Theoretical analysis
    - Empirical analysis

Need of Analysis

- To determine resource consumption
  - CPU time
  - Memory space

- Compare different methods for solving the same problem before actually implementing them and running the programs.

- To find an efficient algorithm for solving the problem

Complexity of an algorithm

- A measure of the performance of an algorithm
- An algorithm's performance is characterized by
  - Time complexity

    How fast an algorithm maps input to output as a function of input
  - Space complexity

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

How to determine complexity of an algorithm?

- Experimental study(Performance Measurement)
- Theoretical Analysis (Performance Analysis)

Limitations of Performance Measurement

- It is necessary to implement the algorithm, which may be difficult
- Results may not be indicative of the running time on other inputs not included in the experiment.
- In order to compare two algorithms, the same hardware and software environments must be used
- Experimental data though important is not sufficient

Performance Analysis

- Uses a high-level description of the algorithm instead of an implementation

- Characterizes running time as a function of the input size, n.

- Takes into account all possible inputs

- Allows us to evaluate the speed of an algorithm independent of the hardware/software environment

THANK YOU

Vandana M L
Department of Computer Science & Engineering

vandanamd@pes.edu

Design and Analysis of Algorithms

Vandana M L
Department of Computer Science & Engineering

DESIGN AND ANALYSIS OF ALGORITHMS

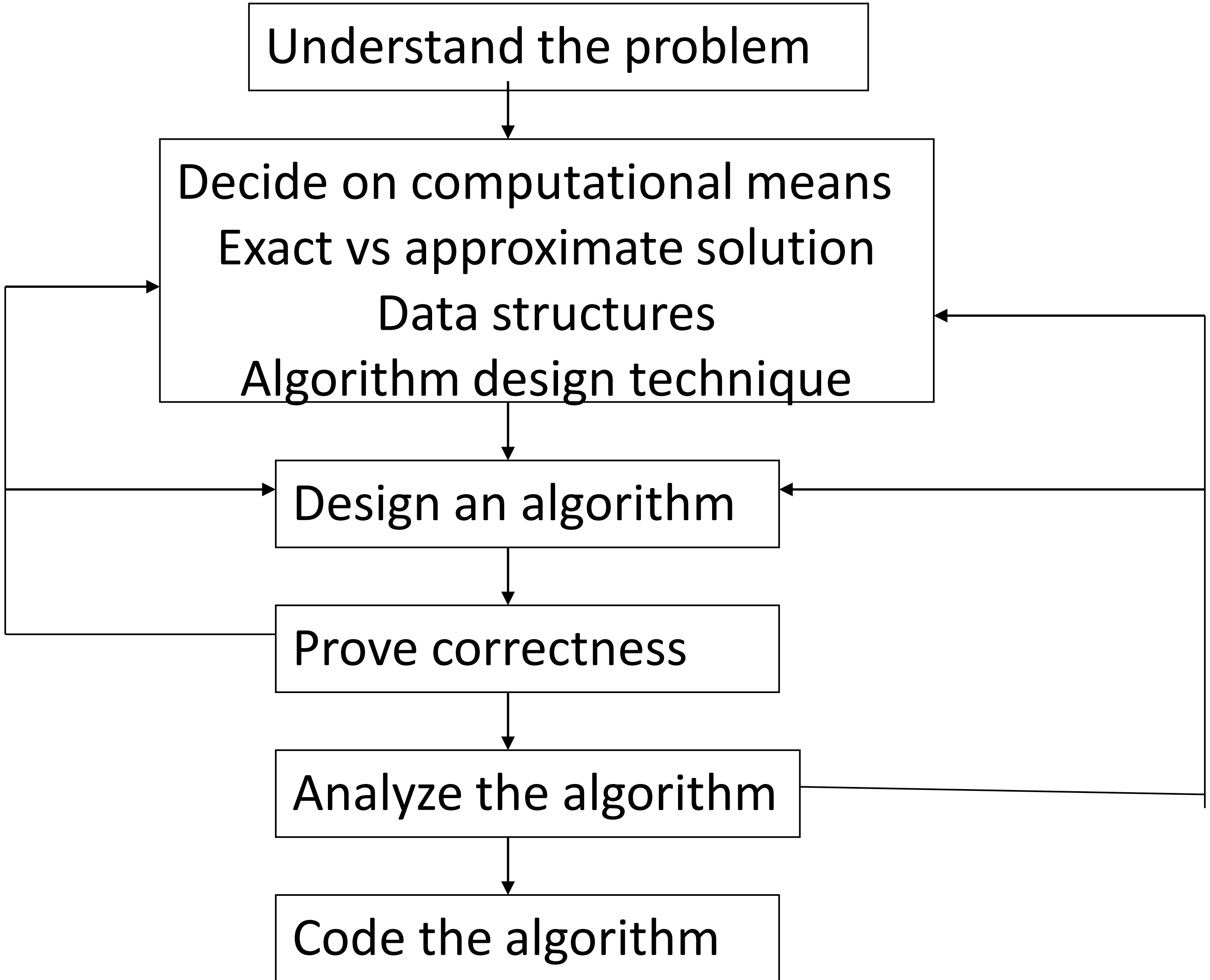
Fundamentals of Algorithmic Problem Solving

Slides courtesy of **Anany Levitin**

Vandana M L

Department of Computer Science & Engineering

Algorithm Design and Analysis Process



Computational Means

Computational Device the algorithm is intended for

RAM Sequential Algorithms

PRAM Parallel Algorithms

Exact vs approximate solution

Travelling Salesman Problem

NP complete!!!

Approximate algorithm can be used to solve it

Deciding on Data Structure

- Linear

  Linear list, Stack, Queues

- Non Linear

  Trees, Graphs

Choice of Data structure for solving a problem using an algorithm may dramatically impact its time complexity

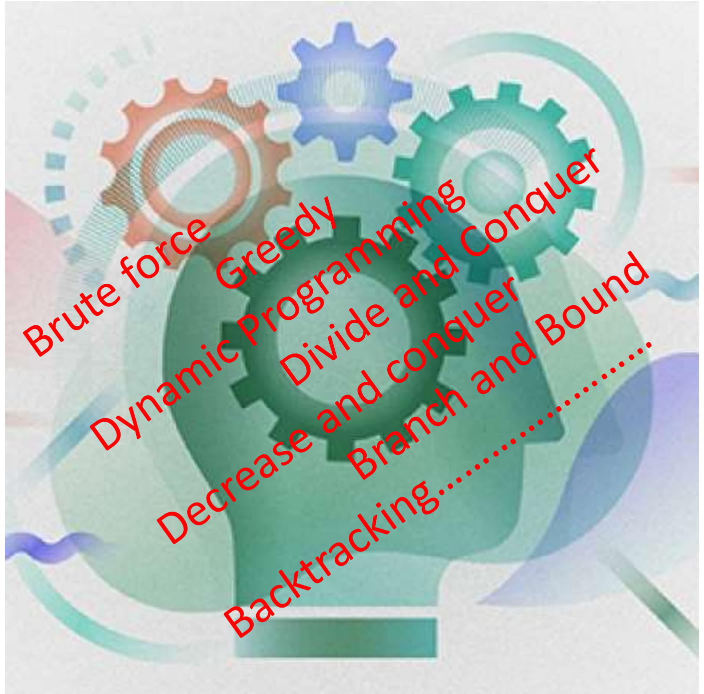
Dijkstra Algorithm

O(VlogV+E) with Fibonacci heap

General approach to solving problems algorithmically that is applicable to variety of problems from different areas of computing

ADT serves as heuristic for designing algorithms for new problems for which no satisfactory algorithm exists!!!

Algorithm Designer's Toolkit



Specifying an algorithm

- Natural Language
- Pseudo Code
- Flowchart

Specifying an algorithm

- Natural Language
- Pseudo Code
- Flowchart

Exact algorithms

Proving that algorithm yields a correct result for legitimate input in finite amount of time

Approximation algorithms

Error produced by algorithm does not exceed a predefined limit

Analyzing an algorithm

- Efficiency
  - Time efficiency
  - Space efficiency

- Simplicity

- Generality
  - Design an algorithm for the problem posed in more general terms
  - Design an algorithm that can handle a range of inputs that is natural for the problem at hand

Coding an algorithm

- Efficient implementation

- Correctness of program
  - Mathematical Approach: Formal verification for small programs
  - Practical Methods: Testing and Debugging

- Code optimization

THANK YOU

Vandana M L

Department of Computer Science & Engineering

vandanamd@pes.edu

Design and Analysis of Algorithms

Vandana M L
Department of Computer Science & Engineering

DESIGN AND ANALYSIS OF ALGORITHMS

Important Problem Types

Slides courtesy of **Anany Levitin**

Vandana M L

Department of Computer Science & Engineering

Important Product Types

- sorting
- searching
- string processing
- graph problems
- combinatorial problems
- geometric problems
- numerical problems

Important Problem Types: Sorting

- Rearrange the items of a given list in ascending order.
  - Input: A sequence of n numbers <a1, a2, …, an>
  - Output: A reordering $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.
- Why sorting?
  - Help searching
  - Algorithms often use sorting as a key subroutine.
- Sorting key

  A specially chosen piece of information used to guide sorting.

  Example: sort student records by SRN.

Important Problem Types: Sorting

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

Important Problem Types: Searching

Find a given value, called a search key, in a given set.

Examples of searching algorithms

- Sequential searching
- Binary searching…

Important Problem Types: String Processing

A string is a sequence of characters from an alphabet.

Text strings: letters, numbers, and special characters.

String matching: searching for a given word/pattern in a text.

Text: I am a computer science graduate

Pattern: computer

Important Problem Types: Graph Problems

Definition

Graph G is represented as a pair G= (V, E),
where V is a finite set of vertices and E is a finite set of edges

Modeling real-life problems

- Modeling WWW
- communication networks
- Project scheduling …

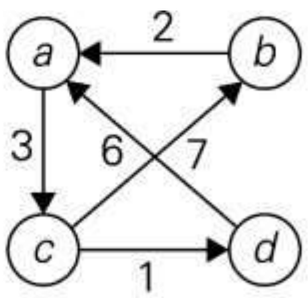
Examples of graph algorithms

- Graph traversal algorithms
- Shortest-path algorithms
- Topological sorting

Important Problem Types: Combinatorial Problems

Shortest paths in a graph

To find the distances from each vertex to all other vertices.



(a)

$$W = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \infty & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \infty & 0 \end{bmatrix} \end{array}$$

(b)

$$D = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 10 & 3 & 4 \\ 2 & 0 & 5 & 6 \\ 7 & 7 & 0 & 1 \\ 6 & 16 & 9 & 0 \end{bmatrix} \end{array}$$

(c)

FIGURE 8.5 (a) Digraph. (b) Its weight matrix. (c) Its distance matrix.

Minimum cost spanning tree

- A spanning tree of a connected graph is its connected acyclic sub graph (i.e. a tree).

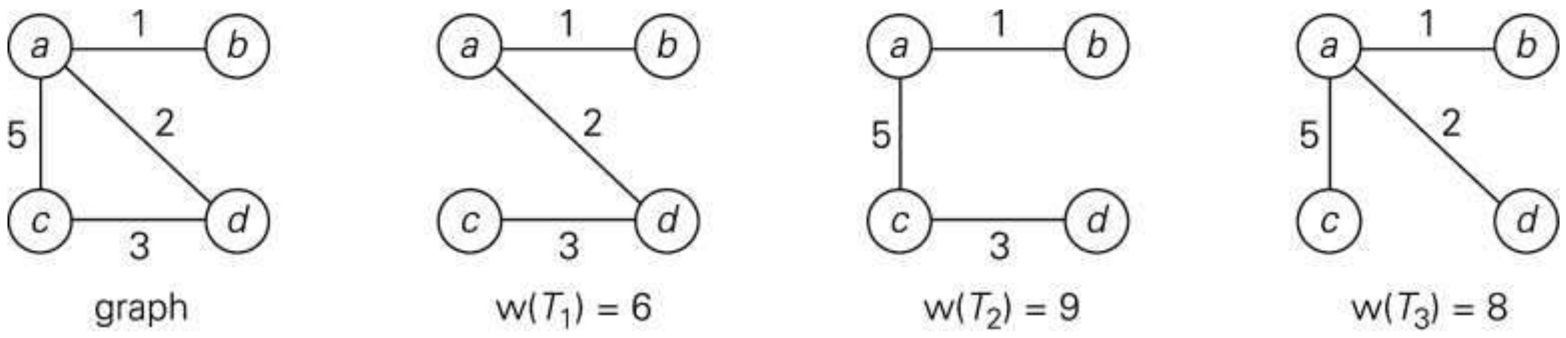
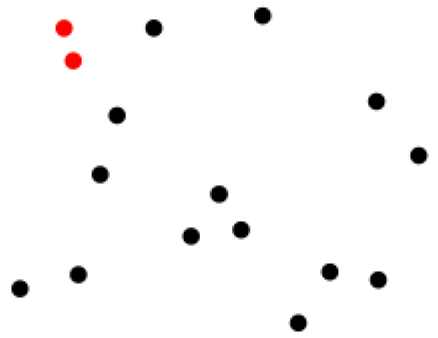


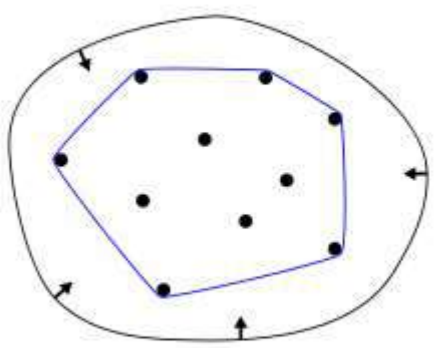
FIGURE 9.1 Graph and its spanning trees; T_1 is the minimum spanning tree

Important Problem Types: Geometric Problems

Closest Pair problem



Convex Hull Problem



Important Problem Types: Numerical Problems

- Solving Equations
- Computing definite integrals
- Evaluating functions

THANK YOU

Vandana M L

Department of Computer Science & Engineering

vandanamd@pes.edu

Design and Analysis of Algorithms

Vandana M L
Department of Computer Science & Engineering

DESIGN AND ANALYSIS OF ALGORITHMS

Analysis Framework

Slides courtesy of **Anany Levitin**

Vandana M L

Department of Computer Science & Engineering

What do you mean by analysing an algorithm?

Investigation of Algorithm’s efficiency with respect to two resources

- Time
- Space

What is the need for Analysing an algorithm?

- To determine resource consumption
  - CPU time
  - Memory space
- Compare different methods for solving the same problem before actually implementing them and running the programs.
- To find an efficient algorithm

Complexity of an Algorithm

- A measure of the performance of an algorithm
- An algorithm's performance depends on
  - *internal* factors
    - Time required to run
    - Space (memory storage)required to run
  - *external* factors
    - Speed of the computer on which it is run
    - Quality of the compiler
    - Size of the input to the algorithm

Performance of Algorithm

important Criteria for performance:

- Space efficiency - the memory required, also called, space complexity
- Time efficiency - the time required, also called time complexity

Space Complexity

S(P)=C+SP(I)

- Fixed Space Requirements (C)
  Independent of the characteristics of the inputs and outputs
  - instruction space
  - space for simple variables, fixed-size structured variable, constants
- Variable Space Requirements (SP(I))
  dependent on the instance characteristic I
  - number, size, values of inputs and outputs associated with I
  - recursive stack space, formal parameters, local variables, return address

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

```
float rsum(float list[ ], int n)
{
  if (n)
    return rsum(list, n-1) + list[n-1]
  return 0
}
```

$S_{sum}(I)=S_{sum}(n)=6n$

Type	Name	Number of bytes
parameter: float parameter: integer return address:(used internally)	list [ ] n	2 2 2
TOTAL per recursive call		6

Time Complexity

$T(P)=C+T_P(I)$

- Compile time (C)
  independent of instance characteristics

- run (execution) time TP

How to measure time complexity?

- Theoretical Analysis
- Experimental study

Experimental study

- Write a program implementing the algorithm
- Run the program with inputs of varying size and composition
- Get an accurate measure of the actual running time
- Use a method like System.currentTimeMillis()
- Plot the results

Limitations of Experimental study

- It is necessary to implement the algorithm, which may be difficult
- Results may not be indicative of the running time on other inputs not included in the experiment.
- In order to compare two algorithms, the same hardware and software environments must be used
- Experimental data though important is not sufficient

Theoretical Analysis

- Uses a high-level description of the algorithm instead of an implementation
- Characterizes running time as a function of the input size, n.
- Takes into account all possible inputs
- Allows us to evaluate the speed of an algorithm independent of the hardware/software environment

Two approaches:

1.Order of magnitude/asymptotic categorization –

This uses coarse categories and gives a general idea of performance.
If algorithms fall into the same category, if data size is small,
or if performance is critical, use method 2

2.Estimation of running time -

1. *operation counts* - select operation(s) that are executed most frequently and determine how many times each is executed.
2. *step counts* - determine the total number of steps, possibly lines of code, executed by the program.

Analysis Framework

- Measuring an input’s size
- Measuring running time
- Orders of growth (of the algorithm’s efficiency function)
- Worst-base, best-case and average efficiency

Measuring an input's size

Efficiency is defined as a function of input size.

Input size depends on the problem.

Example 1, what is the input size of the problem of sorting n numbers?

Example 2, what is the input size of adding two n by n matrices?

Units for Measuring Running Time

- Measure the running time using standard unit of time measurements, such as seconds, minutes?

  Depends on the speed of the computer.

- count the number of times each of an algorithm's operations is executed.

  (step count method)

  Difficult and unnecessary

- count the number of times an algorithm's basic operation is executed.

  Basic operation: the most important operation of the algorithm, the operation contributing the most to the total running time.

  For example, the basic operation is usually the most time-consuming operation in the algorithm's innermost loop.

Analysis in the RAM Model

SmartFibonacci(n)	*cost*	*times* ($n > 1$)
if $n = 0$	c_1	1
then return 0	c_2	0
elseif $n = 1$	c_3	1
then return 1	c_4	0
else $pprev \leftarrow 0$	c_5	1
$prev \leftarrow 1$	c_6	1
for $i \leftarrow 2$ **to** n	c_7	n
do $f \leftarrow prev + pprev$	c_8	$n - 1$
$pprev \leftarrow prev$	c_9	$n - 1$
$prev \leftarrow f$	c_{10}	$n - 1$
return f	c_{11}	1

$$T(n) = c_1 + c_3 + c_5 + c_6 + c_{11} + nc_7 + (n-1)(c_8 + c_9 + c_{10})$$

$T(n) = nC_1 + C_2 \quad \Rightarrow T(n)$ is a *linear function* of n

Input Size and Basic Operation Examples

Problem	***Input size measure***	***Basic operation***
Search for a key in a list of n items	Number of items in list, n	Key comparison
Add two n by n matrices	Dimensions of matrices, n	addition
multiply two matrices	Dimensions of matrices, n	multiplication

Theoretical Analysis of Time Efficiency: Basic operation count

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

input size

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

running time

execution time for the basic operation

Number of times the basic operation is executed

C(n) Basic Operation Count

- The efficiency analysis framework ignores the multiplicative constants of C(n) and focuses on the orders of growth of the C(n).

- Simple characterization of the algorithm's efficiency by identifying relatively significant term in the C(n).

Why do we care about the order of growth of an algorithm's efficiency function, i.e., the total number of basic operations?

- Because, for smaller inputs, it is difficult to distinguish inefficient algorithms vs. efficient ones.
- For example, if the number of basic operations of two algorithms to solve a particular problem are n and n^2 respectively, then
  - if $n = 2$, Basic operation will be executed 2 and 4 times respectively for algorithm1 and 2.

    Not much difference!!!
  - On the other hand, if $n = 10000$, then it does makes a difference whether the number of times the basic operation is executed is n or n^2.

Order of Growth

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

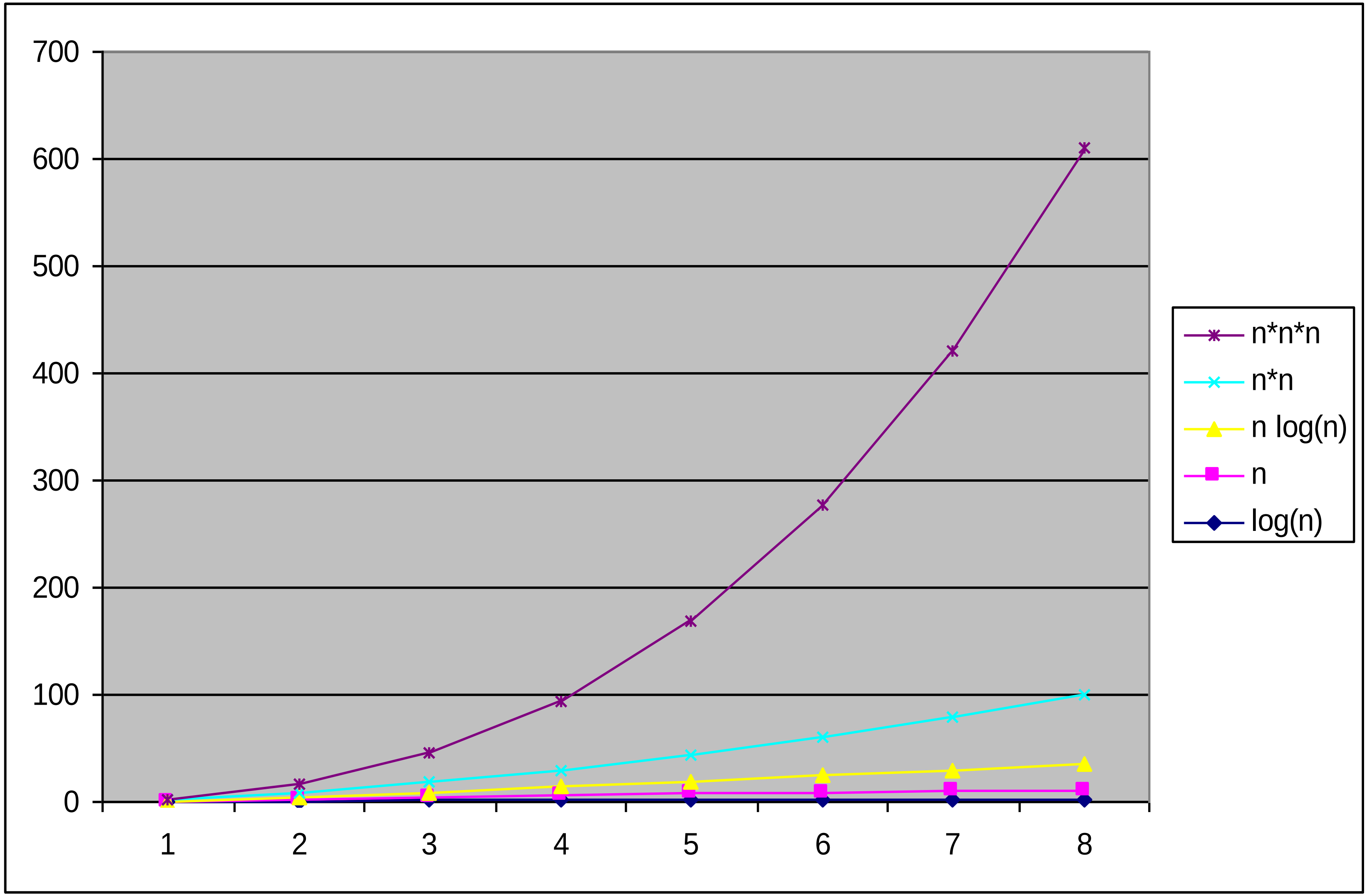
Exponential-growth functions

Table 2.1 Values (some approximate) of several functions important for analysis of algorithms

Orders of growth:

- consider only the leading term of a formula
- ignore the constant coefficient.

Order of Growth



Basic 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

Best, Worst and Average case Analysis

- Algorithm efficiency depends on the input size n
- For some algorithms efficiency depends on type of input.

  Example: Sequential Search

  Problem: Given a list of n elements and a search key K, find an element equal to K, if any.

  Algorithm: Scan the list and compare its successive elements with K until either a matching element is found (successful search) or the list is exhausted (unsuccessful search)

Given a sequential search problem of an input size of n, what kind of input would make the running time the longest? How many key comparisons?

Best, Worst and Average case Analysis

- **Worst case Efficiency**
  - Efficiency (# of times the basic operation will be executed) for the worst case input of size n.
  - The algorithm runs the longest among all possible inputs of size n.
- **Best case**
  - Efficiency (# of times the basic operation will be executed) for the best case input of size n.
  - The algorithm runs the fastest among all possible inputs of size n.
- **Average case:**
  - Efficiency (#of times the basic operation will be executed) for a typical/random input of size n.
  - NOT the average of worst and best case
- **How to find the average case efficiency?**

```
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: Returns 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          //A[I] = K
         return i
   else
         return -1
```

Best, Worst and Average case Analysis: Sequential Search

- Worst-Case: Cworst(n) = n

- Best-Case: Cbest(n) = 1

- Average-Case

  from **(n+1)/2** to **(n+1)**

Average case Analysis: Sequential Search

Let **'p'** be the probability that key is found in the list

Assumption: All positions are equally probable

Case1: key is found in the list

$C_{avg,case1}(n) = p*(1 + 2 + \ldots + n) / n = p*(n + 1) / 2$

Case2: key is not found in the list

$C_{avg,\ case2}(n) = (1-p)*(n)$

$\mathbf{C_{avg}(n) = p}(n + 1) / 2 \mathbf{\ + (1 - p)}(n)$

THANK YOU

Vandana M L
Department of Computer Science & Engineering
vandanamd@pes.edu

Design and Analysis of Algorithms

Vandana M L
Department of Computer Science & Engineering

DESIGN AND ANALYSIS OF ALGORITHMS

Asymptotic Notations

Slides courtesy of **Anany Levitin**

Vandana M L

Department of Computer Science & Engineering

Asymptotic Notations

Order of growth of an algorithm's basic operation count is important

How do we compare order of growth??

Using Asymptotic Notations

A way of comparing functions that ignores constant factors and small input sizes

- $O(g(n))$: class of functions $f(n)$ that grow *no faster* than $g(n)$
- $\Omega(g(n))$: class of functions $f(n)$ that grow *at least as fast* as $g(n)$
- $\Theta(g(n))$: class of functions $f(n)$ that grow *at same rate* as $g(n)$
- $o(g(n))$: class of functions $f(n)$ that grow *at slower rate* than $g(n)$
- $\omega(g(n))$: class of functions $f(n)$ that grow *at faster rate* than $g(n)$

Design and Analysis of Algorithms

O-notation

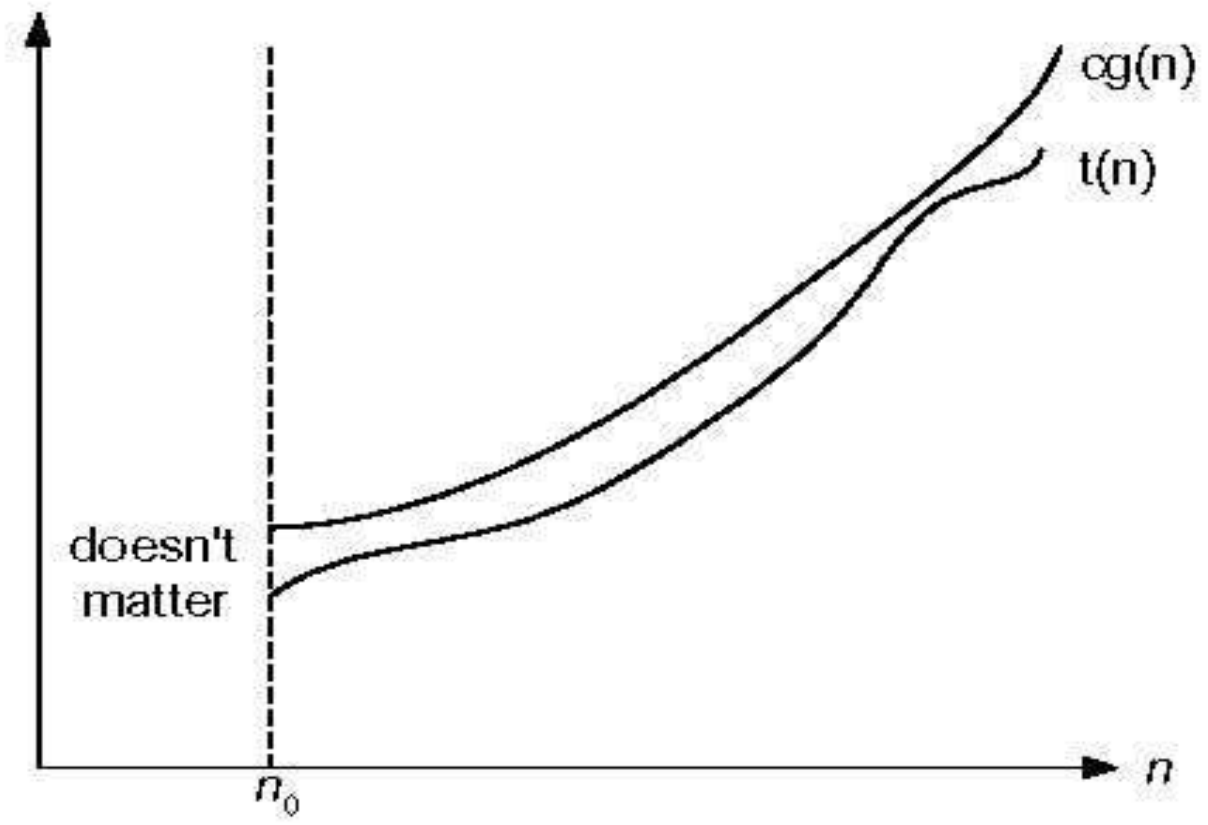


Figure 2.1 Big-oh notation: $t(n) \in O(g(n))$

O-notation

Formal definition

A function $t(n)$ is said to be in $O(g(n))$, denoted $t(n) \in O(g(n))$, if $t(n)$ is bounded above by some constant multiple of $g(n)$ for all large n,

i.e., <u>if there exist some positive constant c and some nonnegative integer n_0 such that</u>

$$t(n) \leq cg(n) \text{ for all } n \geq n_0$$

Example: $100n+5 \in O(n)$

Ω-notation

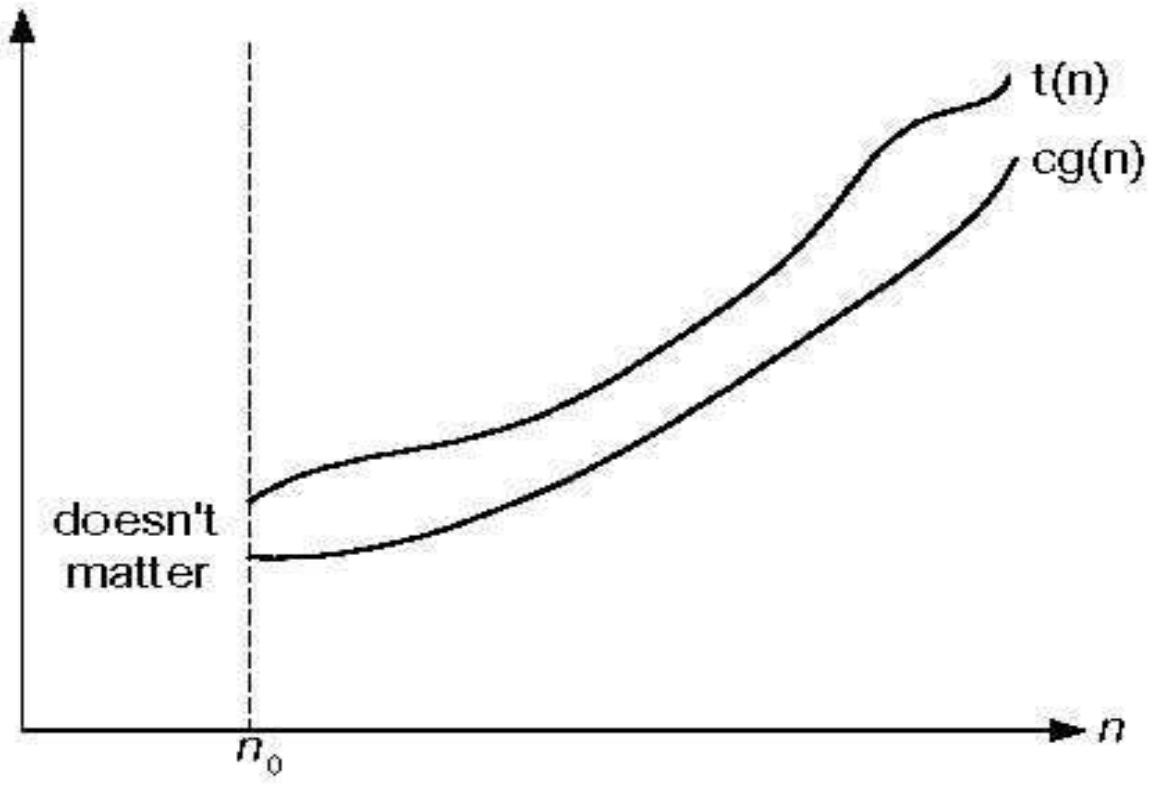


Fig. 2.2 Big-omega notation: $t(n) \in \Omega(g(n))$

Ω-notation

Formal definition

A function $t(n)$ is said to be in $\Omega(g(n))$, denoted $t(n) \in \Omega(g(n))$, if $t(n)$ is bounded below by some constant multiple of $g(n)$ for all large n,

i.e., if there exist some positive constant c and some nonnegative integer n_0 such that

$$t(n) \geq cg(n) \text{ for all } n \geq n_0$$

Example: $10n^2 \in \Omega(n^2)$

Θ-notation

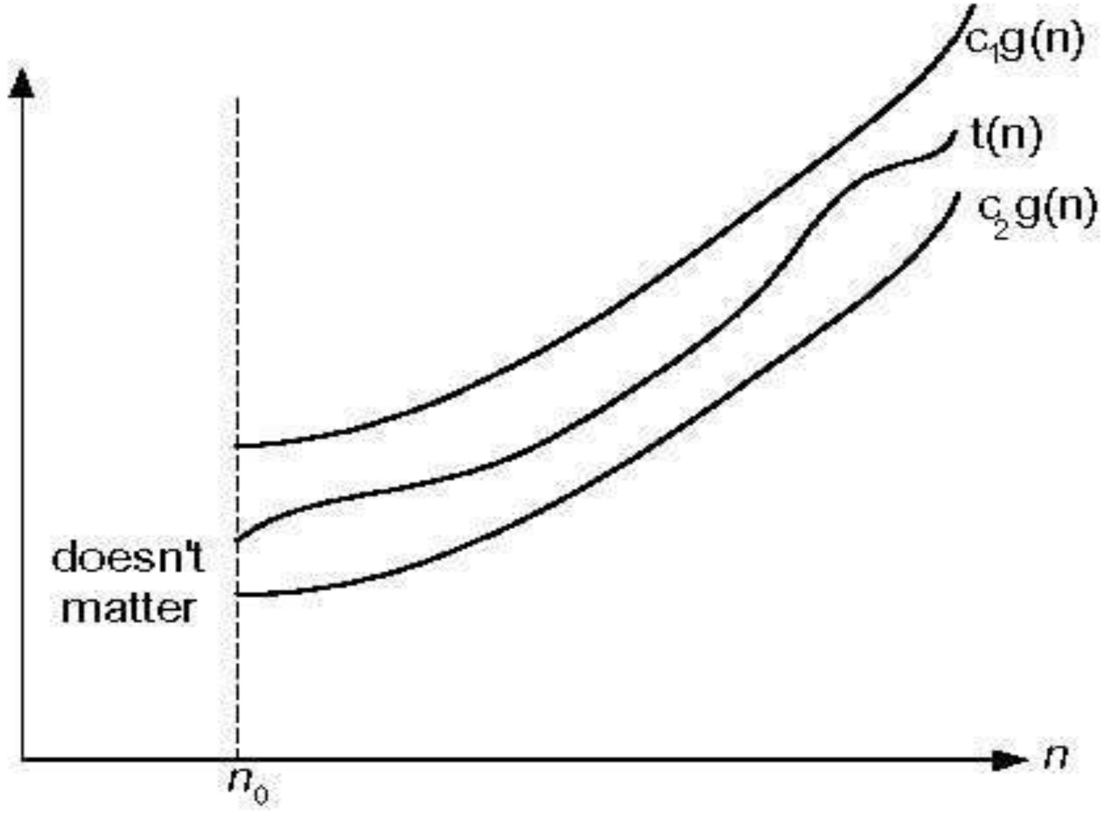


Figure 2.3 Big-theta notation: $t(n) \in \Theta(g(n))$

Θ-notation

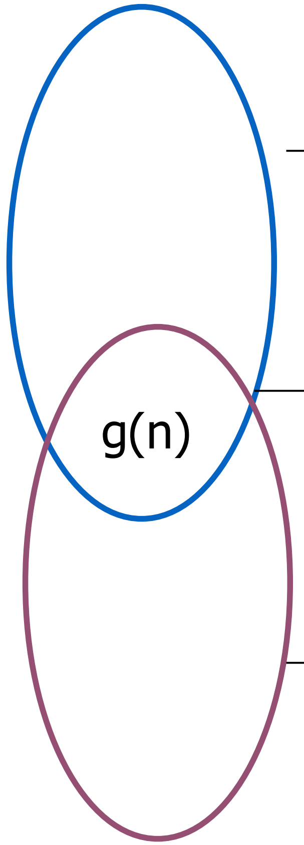
Formal definition

A function $t(n)$ is said to be in $\Theta(g(n))$, denoted $t(n) \in \Theta(g(n))$, if $t(n)$ is bounded both above and below by some positive constant multiples of $g(n)$ for all large n,

i.e., <u>if there exist some positive constants c_1 and c_2 and some nonnegative integer n_0 such that</u>

$$c_2\, g(n) \le t(n) \le c_1\, g(n) \text{ for all } n \ge n_0$$

Example: $(1/2)n(n-1) \in \Theta(n^2)$



$>=$

$\Omega(g(n))$, functions that grow at least as fast as $g(n)$

$=$

$\Theta(g(n))$, functions that grow at the same rate as $g(n)$

$<=$

$O(g(n))$, functions that grow no faster than $g(n)$

Little-o Notation

Formal Definition:

A function *t(n)* is said to be in Little-o*(g(n)),* denoted $t(n) \in o(g(n))$, if for any positive constant c and some nonnegative integer n_0

$$0 \le t(n) < cg(n) \text{ for all } n \ge n_0$$

Example: $n \in o(n^2)$

Little Omega Notation

Formal Definition:

A function $t(n)$ is said to be in Little- $\omega(g(n))$, denoted $t(n) \in \omega(g(n))$, if for any positive constant c and some nonnegative integer n_0

$t(n) > cg(n) \geq 0$ for all $n \geq n_0$

Example: $3n^2 + 2 \in \omega(n)$

Theorems

- If $t_1(n) \in O(g_1(n))$ and $t_2(n) \in O(g_2(n))$, then $t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})$.
  For example,
  $5n^2 + 3n\log n \in O(n^2)$

- If $t1(n) \in \Theta(g1(n))$ and $t2(n) \in \Theta(g2(n))$, then $t1(n) + t2(n) \in \Theta(\max\{g1(n), g2(n)\})$

- $t1(n) \in \Omega(g1(n))$ and $t2(n) \in \Omega(g2(n))$, then $t1(n) + t2(n) \in \Omega(\max\{g1(n), g2(n)\})$

Implication: The algorithm’s overall efficiency will be determined by the part with a larger order of growth, I.e., its least efficient part.

THANK YOU

Vandana M L
Department of Computer Science & Engineering

vandanamd@pes.edu

Design and Analysis of Algorithms

Vandana M L
Department of Computer Science & Engineering

PES UNIVERSITY ONLINE

DESIGN AND ANALYSIS OF ALGORITHMS

Basic Efficiency Classes
Problems based on Asymptotic notations

Slides courtesy of **Anany Levitin**

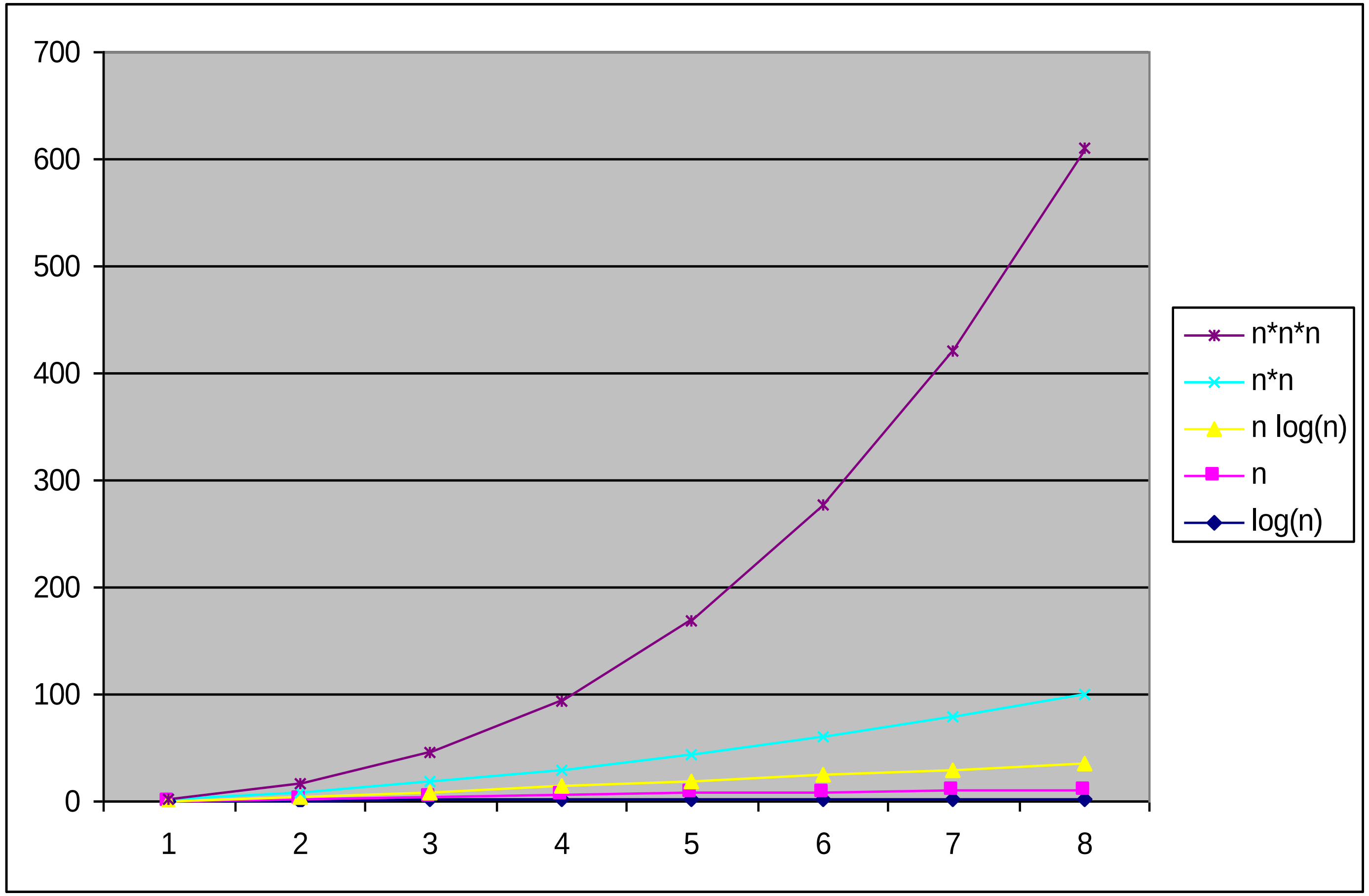
Vandana M L

Department of Computer Science & Engineering

Basic Efficiency Classes

Class	Name	Example
1	constant	Best case for sequential search
log n	logarithmic	Binary Search
n	linear	Worst case for sequential search
n log n	n-log-n	Mergesort
n^2	quadratic	Bubble Sort
n^3	cubic	Matrix Multiplication
2^n	exponential	Subset generation
$n!$	factorial	TSP using exhaustive search

Basic Efficiency Classes



Basic Efficiency Classes

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3{\cdot}10^1$	10^2	10^3	10^3	$3.6{\cdot}10^6$
10^2	6.6	10^2	$6.6{\cdot}10^2$	10^4	10^6	$1.3{\cdot}10^{30}$	$9.3{\cdot}10^{157}$
10^3	10	10^3	$1.0{\cdot}10^4$	10^6	10^9		
10^4	13	10^4	$1.3{\cdot}10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7{\cdot}10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0{\cdot}10^7$	10^{12}	10^{18}		

Table 2.1 Values (some approximate) of several functions important for analysis of algorithms

Design and Analysis of Algorithms

Asymptotic notations

$\underline{O}$

$f(n) = 3n+2 \quad g(n) = n$

$3n+2 \in O(n)$

$f(n) \le c\,g(n)$ for some +ve c $\forall n \ge n_0$

$\Rightarrow f(n) \in O g(n)$

$3n+2 \le cn$

Let $c = 4$

$3n+2 \le 4n$

$n=1 \quad n=2 \quad n=3$

$5 \not\le 4 \quad 8=8 \quad 11 \le 12$

n_0

$3n+2 \le 4n \quad \forall n \ge 2$

$\Rightarrow 3n+2 \in O(n)$

$\underline{\Omega}$

$3n+2 \in \Omega(n)$

$f(n) \ge c\,g(n)$ for some +ve c $\forall n \ge n_0$

$f(n) \in \Omega(n)$

$3n+2 \ge cn$

Let $c = 1$

$3n+2 \ge n$

$n=1 \quad n=2 \quad n=3$

$5 \ge 1 \quad 8 > 2 \quad 11 > 3$

$3n+2 \ge n \quad n \ge 1$

$3n+2 \in \Omega(n)$

$\underline{\Theta}$

$3n+2 \in \Theta(n)$

$c_1 g(n) \le f(n) \le c_2 g(n) \quad \forall n \ge n_0$

$c_1 = 1 \quad c_2 = 4$

$n_{01} = 1 \quad n_{02} = 2$

$n_0 = \max(n_{01}, n_{02}) = 2$

$c_1 = 1 \quad c_2 = 4 \quad n_0 = 2$

$3n+2 \in \Theta(n)$

$f(n) \in O g(n)$

$\in \Omega(n)$

$\Rightarrow f(n) \in \Theta g(n)$

THANK YOU

Vandana M L
Department of Computer Science & Engineering
vandanamd@pes.edu

Design and Analysis of Algorithms

Vandana M L
Department of Computer Science & Engineering

DESIGN AND ANALYSIS OF ALGORITHMS

Method of Limits for comparing order of Growth

Slides courtesy of **Anany Levitin**

Vandana M L

Department of Computer Science & Engineering

Using Limits to Compare Order of Growth

$$\lim_{n\to\infty}\frac{t(n)}{g(n)} = \begin{cases} 0 & \text{implies that } t(n) \text{ has a smaller order of growth than } g(n), \\ c & \text{implies that } t(n) \text{ has the same order of growth as } g(n), \\ \infty & \text{implies that } t(n) \text{ has a larger order of growth than } g(n). \end{cases}$$

Case1: $t(n) \in O(g(n)$

Case2: $t(n) \in \Theta(g(n))$

Case3: $g(n) \in O(t(n))$

t'(n) and g'(n) are first-order derivatives of t(n) and g(n)

L'Hopital's Rule $\lim_{n\to\infty}\frac{t(n)}{g(n)} = \lim_{n\to\infty}\frac{t'(n)}{g'(n)}$

Stirling's Formula $n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$ for large values of n

Using Limits to Compare Order of Growth: Example 1

Compare the order of growth of f(n) and g(n) using method of limits

$t(n) = 5n3 + 6n + 2$, $g(n) = n4$

$$\lim_{n\to\infty}\frac{t(n)}{g(n)} = \lim_{n\to\infty}\frac{5n^3+6n+2}{n^4} = \lim_{n\to\infty}\left(\frac{5}{n}+\frac{6}{n^3}+\frac{2}{n^4}\right) = 0$$

As per case1

t(n) = O(g(n))

$5n^3 + 6n + 2 = O(n^4)$

Using Limits to Compare Order of Growth: Example 2

$$t(n) = \sqrt{5n^2 + 4n + 2}$$

using the Limits approach determine g(n) such that f(n) = Θ(g(n))

Leading term in square root n2

$$g(n) = \sqrt{n^2} = n$$

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \lim_{n\to\infty} \frac{\sqrt{5n^2 + 4n + 2}}{\sqrt{n^2}}$$

$$= \lim_{n\to\infty} \sqrt{\frac{5n^2 + 4n + 2}{n^2}} = \lim_{n\to\infty} \sqrt{5 + \frac{4}{n} + \frac{2}{n^2}} = \sqrt{5}$$

non-zero constant

Hence, t(n) = Θ(g(n)) = Θ(n)

Using Limits to Compare Order of Growth

$$lim_{n \to \infty} t(n)/g(n) \neq 0, \infty \Rightarrow t(n) \in \Theta(g(n))$$

$$lim_{n \to \infty} t(n)/g(n) \neq \infty \quad \Rightarrow t(n) \in O(g(n))$$

$$lim_{n \to \infty} t(n)/g(n) \neq 0 \quad \Rightarrow t(n) \in \Omega(g(n))$$

$$lim_{n \to \infty} t(n)/g(n) = 0 \quad \Rightarrow t(n) \in o(g(n))$$

$$lim_{n \to \infty} t(n)/g(n) = \infty \quad \Rightarrow t(n) \in \omega(g(n))$$

Using Limits to Compare Order of Growth: Example 3

Compare the order of growth of t(n) and g(n) using method of limits

$$t(n) = \log_2 n,\ g(n) = \sqrt{n}$$

$$\lim_{n\to\infty} \frac{\log_2 n}{\sqrt{n}} = \lim_{n\to\infty} \frac{(\log_2 n)'}{(\sqrt{n})'} = \lim_{n\to\infty} \frac{(\log_2 e)\frac{1}{n}}{\frac{1}{2\sqrt{n}}} = 2\log_2 e \lim_{n\to\infty} \frac{1}{\sqrt{n}} = 0$$

$$\log_2 n \in o(\sqrt{n})$$

Design and Analysis of Algorithms

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

$\mathbf{\log_{10} n \in \Theta(\log_2 n)}$

- 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 an have different orders of growth for different a's

$\mathbf{3^n \notin \Theta(2^n)}$

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

How to Establish Orders of Growth of an Algorithm’s Basic Operation Count

Summary

- Method 1: Using limits.
  - L’ Hôpital’s rule
- Method 2: Using the theorem.
- Method 3: Using the definitions of O-, Ω-, and Θ-notation.

THANK YOU

Vandana M L
Department of Computer Science & Engineering

vandanamd@pes.edu

Design and Analysis of Algorithms

Vandana M L
Department of Computer Science & Engineering

DESIGN AND ANALYSIS OF ALGORITHMS

Mathematical Analysis of Non-recursive Algorithms

Slides courtesy of **Anany Levitin**

Vandana M L

Department of Computer Science & Engineering

Steps in mathematical analysis of non-recursive algorithms:

- Decide on parameter n indicating input size
- Identify algorithm's basic operation
- Check whether the number of times the basic operation is executed depends only on the input size n. If it also depends on the type of input, investigate worst, average, and best case efficiency separately.
- Set up summation for C(n) reflecting the number of times the algorithm's basic operation is executed.
- Simplify summation using standard formulas

Useful Summation Formulas and Rules

$\Sigma_{l \le i \le u} 1 = 1+1+\ldots+1 = u - l + 1$

$\Sigma_{1 \le i \le n}\, i = 1+2+\ldots+n = n(n+1)/2$

$\Sigma_{1 \le i \le n}\, i^2 = 1^2+2^2+\ldots+n^2 = n(n+1)(2n+1)/6$

$\Sigma_{0 \le i \le n}\, a^i = 1 + a + \ldots + a^n = (a^{n+1} - 1)/(a - 1)$ for any $a \neq 1$

$\Sigma(a_i \pm b_i) = \Sigma a_i \pm \Sigma b_i$ $\quad \Sigma c a_i = c \Sigma a_i$

$\Sigma_{l \le i \le u} a_i = \Sigma_{l \le i \le m} a_i + \Sigma_{m+1 \le i \le u} a_i$

$$\sum_{i=l}^{u} 1 = (u - l + 1)$$

Example 1: Finding Max Element in a list

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

- The basic operation- comparison
- Number of comparisons is the same for all arrays of size n.
- Number of comparisons

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

Example 2: Element Uniqueness Problem

Algorithm *UniqueElements (A[0..n-1])*
//Checks 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
```

<u>Best-case:</u>
If the two first elements of the array are the same
No of comparisons in Best case = 1 comparison

<u>Worst-case:</u>

- Arrays with no equal elements
- Arrays in which only the last two elements are the pair of equal elements

Example 2: Element Uniqueness Problem

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

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

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

Best-case: 1 comparison

Worst-case: $n^2/2$ comparisons

$T(n)_{\text{worst case}} = O(n^2)$

Example 3:Matrix Multiplication

```
Algorithm MatrixMultiplication(A[0..n-1, 0..n-1], B[0..n-1, 0..n-1] )
//Multiplies two square matrices of order n 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
```

M(n) € Θ (n^3)

THANK YOU

Vandana M L
Department of Computer Science & Engineering

vandanamd@pes.edu

Design and Analysis of Algorithms

Vandana M L
Department of Computer Science & Engineering

DESIGN AND ANALYSIS OF ALGORITHMS

Mathematical Analysis of Recursive Algorithms

Slides courtesy of **Anany Levitin**

Vandana M L

Department of Computer Science & Engineering

Steps in Mathematical Analysis of Recursive Algorithms

- Decide on parameter n indicating input size
- Identify algorithm's basic operation
- If the number of times the basic operation is executed varies with different inputs of same sizes , investigate worst, average, and best case efficiency separately
- Set up a recurrence relation and initial condition(s) for C(n)-the number of times the basic operation will be executed for an input of size n
- Solve the recurrence or estimate the order of magnitude of the solution

- Decrease-by-one recurrences

A decrease-by-one algorithm solves a problem by exploiting a relationship between a given instance of size *n* and a smaller size *n – 1*.

Example: n!

The recurrence equation has the form

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

- Decrease-by-a-constant-factor recurrences

A decrease-by-a-constant algorithm solves a problem by dividing its given instance of size *n* into several smaller instances of size *n/b*, solving each of them recursively, and then, if necessary, combining the solutions to the smaller instances into a solution to the given instance.

Example: binary search.

The recurrence has the form

$$T(n) = aT(n/b) + f(n)$$

- One (constant) operation reduces problem size by one.

  $T(n) = T(n\text{-}1) + c$ $\qquad$ $T(1) = d$

  Solution: $T(n) = (n\text{-}1)c + d$ $\qquad$ ***linear***

- A pass through input reduces problem size by one.

  $T(n) = T(n\text{-}1) + c\,n$ $\qquad$ $T(1) = d$

  Solution: $T(n) = [n(n+1)/2 - 1]\,c + d$ $\qquad$ ***quadratic***

Methods to solve recurrences

- Substitution Method
    - Mathematical Induction
    - Backward substitution
- Recursion Tree Method
- Master Method (Decrease by constant factor recurrences)

Recursive Evaluation of n!

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

Recursive definition of n!:

$F(n) = F(n\text{-}1) * n$ for $n \geq 1$

$F(0) = 1$

input size?

basic operation?

Best/Worst/Average Case?

ALGORITHM $F(n)$

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

$$M(n) = M(n-1) + 1 \quad \text{for } n > 0,$$

$$M(0) = 0.$$

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

M(n) = n

Overall time Complexity: Θ(n)

Counting number of binary digits in binary representation of a number

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

input size?

basic operation?

Best/Worst/Average Case?

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

$$\begin{aligned}
A(2^k) &= A(2^{k-1}) + 1 && \text{substitute } A(2^{k-1}) = A(2^{k-2}) + 1 \\
&= [A(2^{k-2}) + 1] + 1 = A(2^{k-2}) + 2 && \text{substitute } A(2^{k-2}) = A(2^{k-3}) + 1 \\
&= [A(2^{k-3}) + 1] + 2 = A(2^{k-3}) + 3 && \ldots \\
&\ldots \\
&= A(2^{k-i}) + i \\
&\ldots \\
&= A(2^{k-k}) + k.
\end{aligned}$$

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

Tower of Hanoi

Algorithm TowerOfHanoi(n, Src, Aux, Dst)
if (n = 0)
return
TowerOfHanoi(n-1, Src, Dst, Aux)
Move disk n from Src to Dst
TowerOfHanoi(n-1, Aux, Src, Dst)

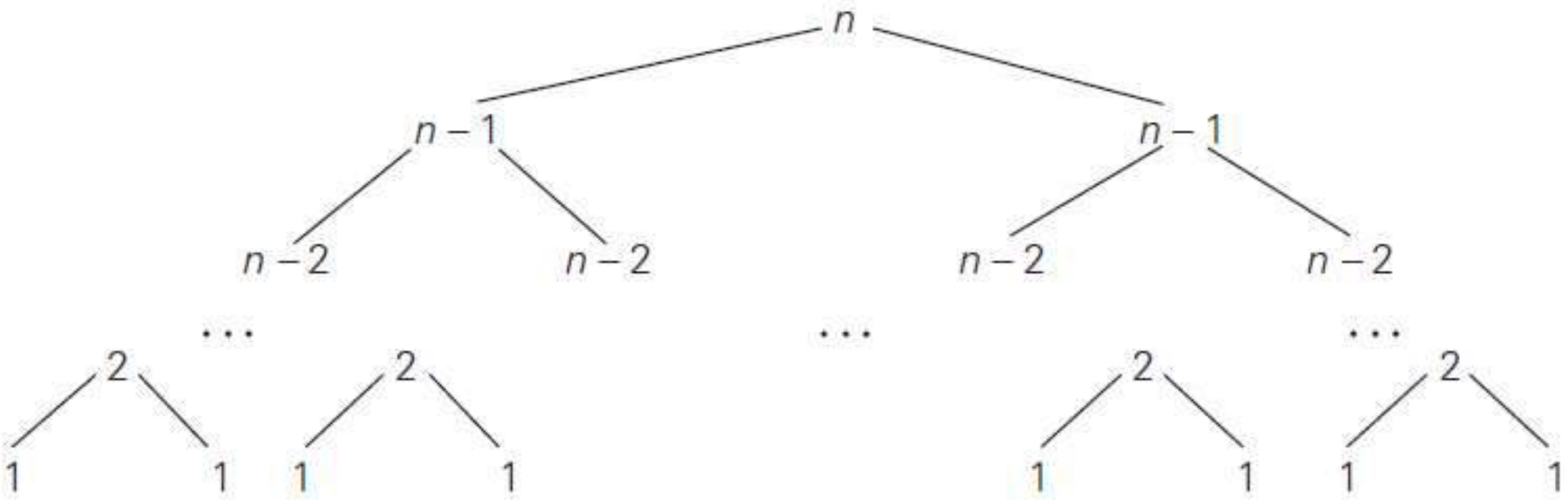
Input Size: **`n`**

Basic Operation : **`Move disk n from Src to Dst`**

$C(n) = \mathbf{2C(n-1) + 1}$ for $n > 0$ and $C(0)=0$

$= \mathbf{2^n - 1 \in \Theta(2^n)}$

Tower of Hanoi : Tree of Recursive calls



$$C(n) = \sum_{l=0}^{n-1} 2^l = 2^n - 1$$

THANK YOU

Vandana M L
Department of Computer Science & Engineering

vandanamd@pes.edu

Design and Analysis of Algorithms

Vandana M L
Department of Computer Science & Engineering

DESIGN AND ANALYSIS OF ALGORITHMS

Solving Recurrences

Slides courtesy of **Anany Levitin**

Vandana M L

Department of Computer Science & Engineering

Solving Recurrences: Example1

$T(n) = T(n-1) + 1 \qquad n>0 \qquad\qquad T(0) = 1$

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

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

$\quad = T(n-3) + 1 + 2 = T(n-3) + 3$

$\quad \ldots$

$\quad = T(n-i) + i$

$\quad \ldots$

$\quad = T(n-n) + n = n = O(n)$

Solving Recurrences: Example2

$T(n) = T(n-1) + 2n – 1 \qquad T(0)=0$

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

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

$= [T(n-3) + 2(n-2) -1] + 2(n-1) + 2n - 2$

$= T(n-3) + 2(n-2) + 2(n-1) + 2n - 3$

...

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

...

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

$= 0 + 2 + 4 +...+ 2n - n$

$= 2 + 4 +...+ 2n - n$

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

// arithmetic progression formula $1+...+n = n(n+1)/2$ //

$= O(n^2)$

Solving Recurrences: Example3

$$T(n) = T(n/2) + 1 \quad n > 1$$
$$T(1) = 1$$

$$T(n) = T(n/2) + 1$$
$$= T(n/2^2) + 1 + 1$$
$$= T(n/2^3) + 1 + 1 + 1$$
$$\ldots\ldots$$
$$= T(n/2^i) + i$$
$$\ldots\ldots$$
$$= T(n/2^k) + k \ \ (k = \log n)$$
$$= 1 + \log n$$
$$= O(\log \ n)$$

Solving Recurrences: Example4

$$T(n) \ = 2T(n/2) + cn \qquad n > 1 \qquad T(1) = c$$

$$T(n) = 2T(n/2) + cn$$

$$= 2(2T(n/2^2) + c(n/2)) + cn = 2^2T(n/2^2) + cn + cn$$

$$= 2^2(2T(n/2^3) + c(n/2^2)) + cn + cn = 2^3T(n/2^3) + 3cn$$

$$\ldots\ldots$$

$$= 2^iT(n/2^i) + icn$$

$$\ldots\ldots$$

$$= 2^kT(n/2^k) + kcn \quad (k = \log\ n)$$

$$= nT\ (1) + cn\log n = cn + cn\log n$$

$$= \mathrm{O}(n\log\ n)$$

THANK YOU

Vandana M L
Department of Computer Science & Engineering

vandanamd@pes.edu

Design and Analysis of Algorithms

Vandana M L
Department of Computer Science & Engineering

DESIGN AND ANALYSIS OF ALGORITHMS

Performance Analysis Vs Performance Measurement

Slides courtesy of **Anany Levitin**

Vandana M L

Department of Computer Science & Engineering

Performance Evaluation of Algorithm

- Performance Analysis
  - Machine Independent
  - Prior Evaluation
- Performance Measurement
  - Machine Dependent
  - Posterior Evaluation

Performance Analysis of Sequential search :Worst Case

```
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: Returns 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          //A[i] = K
         return i
   else
         return -1
```

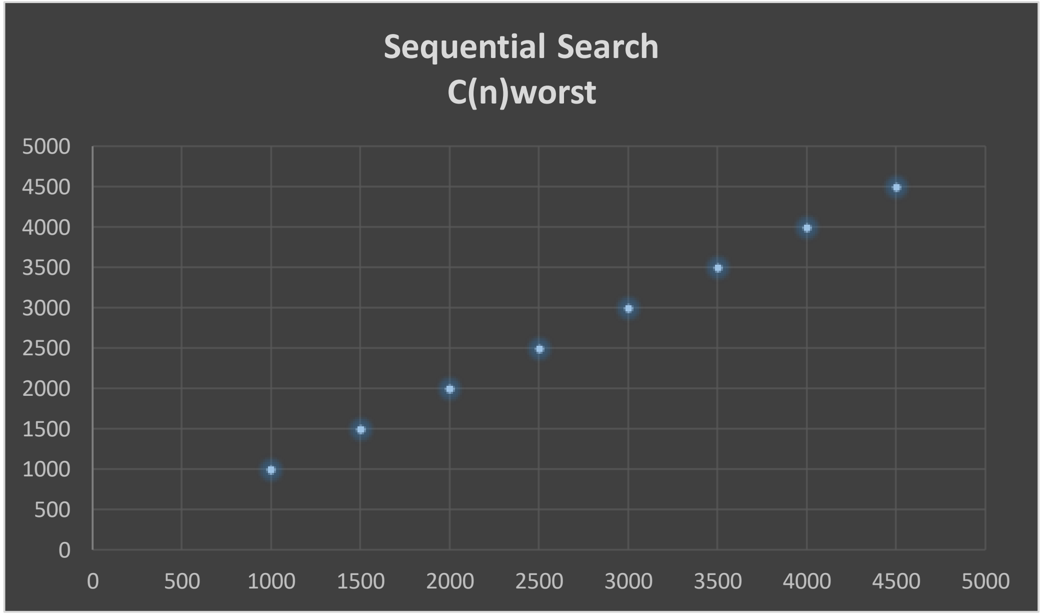
Basic operation: A[i] ‡ K

Basic operation count: n

Time Complexity: $T(n) \in O(n)$

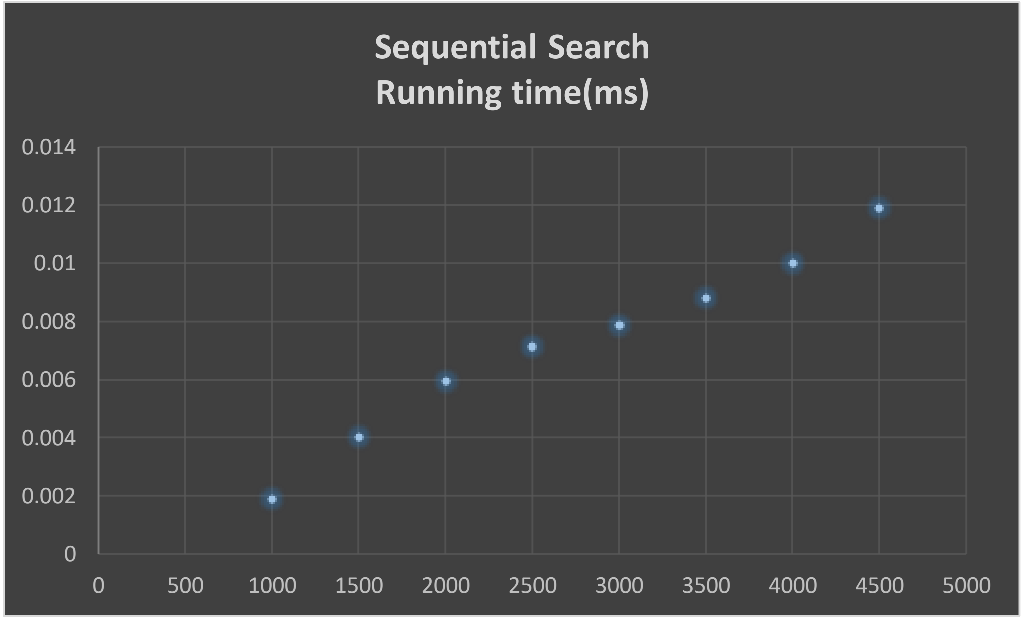
Performance Analysis of Sequential Search

Input Size	Sequential Search C(n)worst
1000	1000
1500	1500
2000	2000
2500	2500
3000	3000
3500	3500
4000	4000
4500	4500



Performance Measurement of Sequential Search

Input Size	Sequential Search Actual Running Time(ms)
1000	0.001907
1500	0.004053
2000	0.00596
2500	0.007153
3000	0.007868
3500	0.008821
4000	0.010014
4500	0.011921



THANK YOU

Vandana M L
Department of Computer Science & Engineering

vandanamd@pes.edu