

PDFZilla – Unregistered

PDFZilla - Unregistered

PDFZilla - Unregistered



Module-1

Design and Analysis of Algorithms

Introduction

Contents

- **Introduction:**

What is an Algorithm? (T2:1.1), Algorithm Specification (T2:1.2), Analysis Framework (T1:2.1), Performance Analysis: Space complexity, Time complexity (T2:1.3).

- **Asymptotic Notations:**

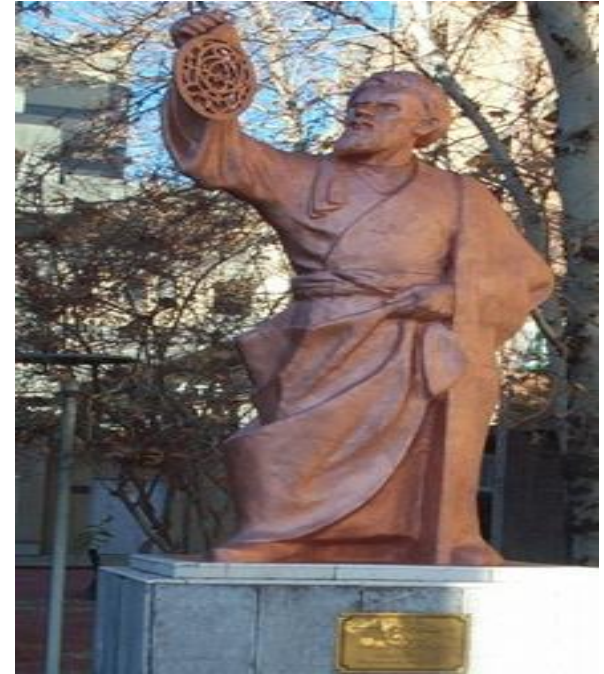
Big-Oh notation (O), Omega notation (Ω), Theta notation (Θ), and Little-oh notation (o), Mathematical analysis of Non-Recursive and recursive Algorithms with Examples (T1:2.2, 2.3, 2.4). Important Problem Types: Sorting, Searching, String processing, Graph Problems, Combinatorial Problems.

- **Fundamental Data Structures:**

Stacks, Queues, Graphs, Trees, Sets and Dictionaries. (T1:1.3,1.4).

Who Coined the word Algorithm?

- Mohammad ben Musā Khwārazmī(780 – 850), Arabized as al-Khwarizmi and formerly Latinized as Algorithmi, was a Persian polymath who produced vastly influential works in mathematics, astronomy, and geography.
- In Europe, the word "algorithm" was originally used to refer to the sets of rules and techniques used by Al-Khwarizmi to solve algebraic equations, before later being generalized to refer to any set of rules or techniques.



Why Algorithms?

Airways Route



Xerox Shop



Why Algorithms?

- Is the water safe & cleansified to drink?
- To Check documents similarity
- What are the affects of Climate Change?
- Design the self Driven cars.

1.1 What is an Algorithm? (T2:1.1)

- Abstract Computational procedure which accept valid input and produce valid output.
- **Program= Expression of an Algorithm.**
- **Algorithm are intended to be read by human beings.**

Program= Expression of an Algorithm

1.1 What is an Algorithm? (T2:1.1)

- Step by Step procedure to solve a given problem.
- Method used by a computer for the solution of the problem.
- An algorithm is a sequence of computational steps that transforms the input into the output.
- “An Algorithm is a finite set of instructions that if followed , accomplishes a particular task”.
- “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”

Program= Expression of an Algorithm

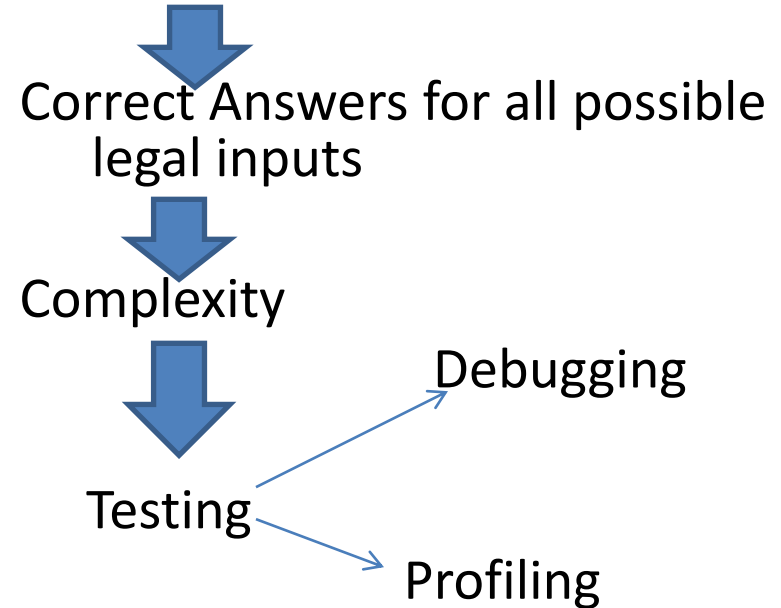
Criteria for Algorithm

- **Input-** Zero or more quantities externally supplied.
- **Output-** At least one quantity is produced.
- **Definiteness** – Each Instruction is clear and unambiguous.
- **Finiteness:** Algorithm terminates at finite number of steps.
- **Effectiveness:** Every instruction can be implemented.

The study of Algorithms include

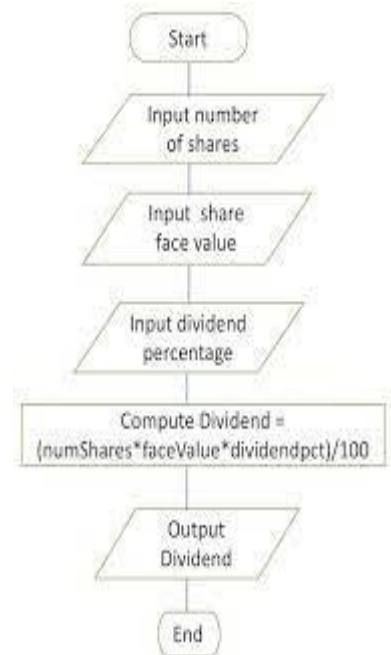
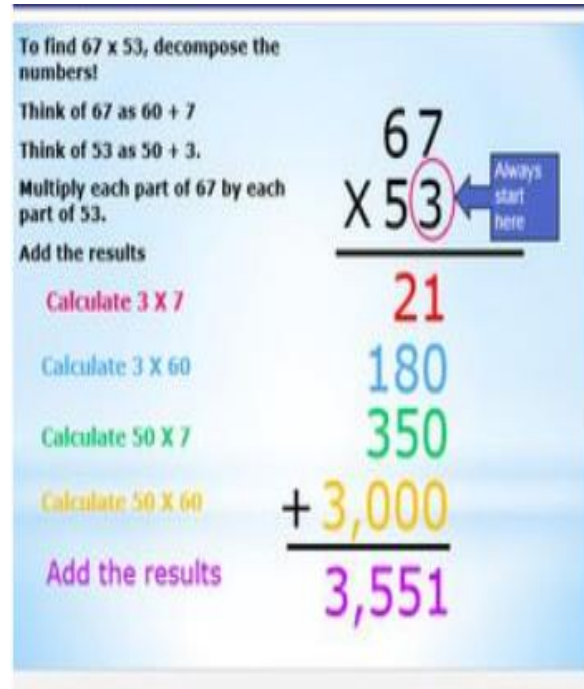
1. How to devise Algorithms?
2. How to validate Algorithms?
3. How to analyze Algorithms?
4. How to test a program?

Various Approaches



1.2 Algorithm Specification (T2:1.2)

- **Natural Representation-**
Instructions must be definite and clear.
- **Graphical Representation-**
They work only if algorithm is simple and small.



Pseudo Code Convention

- Comments begin with //
- Compound statements/Blocks are specified with a pair of matching braces i.e {
 }
 - Statements are delimited by ;
- An identifier/variable begins with letter and data types of variables are not explicitly specified.
- Assignment of values to variables is done using assignment statement
 - Variable:=expression; variable <- value/exp

- Logical Operators(‘and’, ‘or’ & ‘not’) and Relational Operators(<,<=,>,==,>=) are provided.
- Array elements are accessed using A[i,j].
- Looping
 - while<condition> do
 - {
 - }
 - for variable=value1 to value n step step do
 - {
 - }

```
for i= 1 to n do  
{  
  
}
```

repeat

<statement>

.

until<condition>

– break & return are used to exit from a loop and function.

8. Conditional Statements

if(condition) then <statements>

Case statements

9. Input/output – read/write

10. Procedure: Algorithm Name(<parameter list>)

Write?

- Driving directions from your home to college.
- Recipe for cooking your favorite dish.

```
Algorithm Max(a1,a2)
// finds Maximum of 2 number
{
    if(a1>a2)
        write a1 is greater;
    else
        write a2 is greater;
}
```

Puzzle

- A peasant find himself on a river bank with a wolf, a goat and a cabbage. He need to transport all the three to the other side of the river in the boat. However , the boat has room for only the peasant himself and one other item(either wolf/goat/cabbage). In his absence , the wolf would eat the goat, and the goat can eat cabbage. Solve the problem for peasant or show it has no solution.

“Not everything that can be counted counts, and not everything that counts can be counted”.

Albert Einstein(1879-1955)

Analysis Framework (T1:2.1)

Basic Idea:

Mathematical Model
for a computer

- Mentally executed.
- Will evaluate the time.

What is the time taken?

What is Input?

How does model relate
to real computer.

Analysis Framework (T1:2.1)

- **Analysis** is the process of breaking a complex topic or substance into smaller parts in order to gain a better understanding of it.
- The process of studying or examining something in an organized way to learn more about it, or a particular study .
- The separation of an intellectual or substantial whole into its constituent parts for individual study.

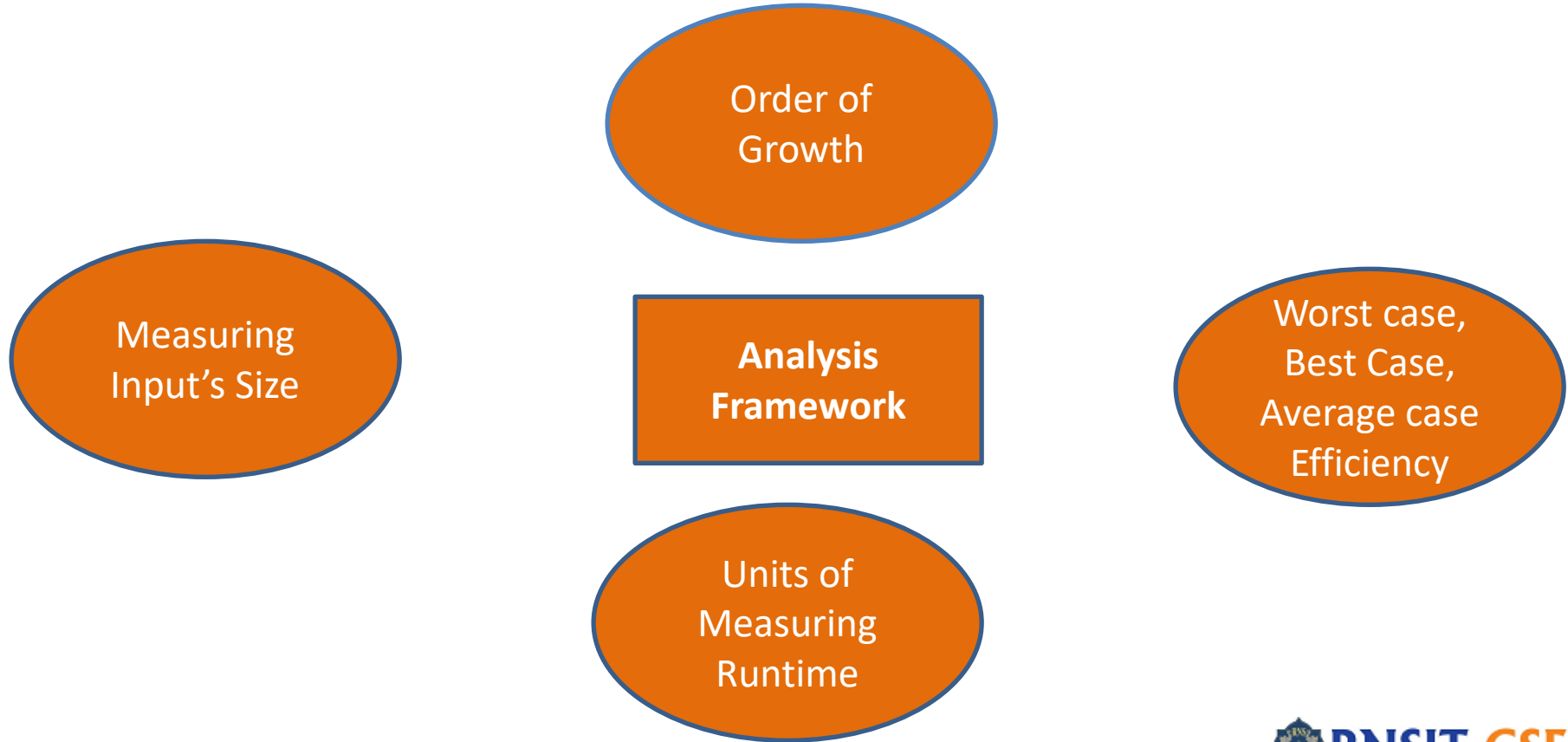
Analysis Framework

- Analysis Framework is a systematic approach that can be applied for analyzing any given algorithm.
- Framework for analyzing the efficiency of algorithms is through
 - *Time Efficiency*
 - *Space Efficiency*

Time efficiency indicated how fast an algorithm in question runs

Space efficiency deals with extra space/memory units the algorithm requires.

Factors for Analysis Framework



Measuring Input Size

- Almost all algorithms run longer on larger inputs
 - it takes longer to sort larger arrays, multiply larger matrices, and so on
- How about measuring algorithm's efficiency as a function of some parameter n (*in few cases more than one parameters*) indicating the algorithm's input size ?
- Is this selection of parameter n straightforward ?
- Guess the input size for the following problems
 - Sorting
 - Searching
 - Smallest element in a list
 - Matrix multiplications
 - Polynomial evaluation
 - Polynomial addition
 - Spell check algorithm

Measuring Input Size

- $b = \lceil \log_2 n \rceil + 1$

b = number of bits.

n = input parameter.

Units of Measuring Runtime

- **Basic Operation:** Most important operation of an algorithm.
- $T(n) = C_{op} C(n)$

Simple programs

Algorithm Sum(n)

// finds sum of given numbers

```
{  
    s = 0  
    for i = 1 to n do  
        s = s + i;  
    return s  
}
```

Algorithm Max(A[n])

// finds Largest of given Array

```
{  
    max=A[0];  
    for i=1 to n-1 do  
    {  
        if (max< A[i]) then  
        {  
            max= A[i];  
        }  
    }  
}
```

- Assume $C(n) = \frac{1}{2} n(n-1)$, how much longer will the algorithm run if we double its input size?

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

$$T(n) = O(C(n))$$

$$T_1(n) = \frac{1}{2} n^2$$

$$T_2(2n) = \frac{1}{2} 4n^2$$

Order of Growth

	n			
	10	50	100	1000
Log n	0.00003sec	0.00005 sec	0.00007 sec	0.00009 sec
$n^{1/2}$	0.00003sec	0.00007 sec	0.00010 sec	0.00032 sec
n	0.00010sec	0.00050 sec	0.00100 sec	0.01000sec
n log n	0.00033sec	0.00282 sec	0.00664 sec	0.09966sec
n^2	sec	sec	min	hrs
n^3	sec	min	hrs	day
n^4	min	day	yrs	cen
$2n$	sec	yrs	Cen	cen
$n!$	362.88 sec	$1 \cdot 10^{51}$ cen		

Efficiencies

- Worst Case
- Best Case
- Average Case

```
While i < n and A[i] ≠ k do  
    i = i + 1;  
If i < n return i  
else return -1
```

1.4 Performance Analysis (T2:1.3)

Performance Analysis of an algorithm depends upon two factors :

- **Time Complexity**
 - Amount of Computer time it needs to run to completion
- **Space Complexity**
 - Amount of memory it needs to run to completion.

Space Complexity

- Space needed by each of algorithms is the sum of the following components:
 - Fixed Part
 - Variable Part
- **Fixed Part** : Independent of characteristics of input and output. It includes Instruction space, space for simple variables, etc..
- **Variable Part**: Dependent on particular problem instance, referenced variables, recursion stack

Space Complexity

- Space requirement
- $S(P) = c + S_p$
- S_p : Instance Characteristics/variable Part
- c : Constant/Fixed part.
- When analyzing the space complexity of an algorithm, S_p is of more concern

Simple Examples

```
{
```

```
    int z = a + b + c;  
    return(z);
```

```
}
```

$$S(P) = c + S_p$$

$$S(P) = 12 + 0 \text{ bytes}$$


```
{  
    s = 0;  
    for i = 1 to n do  
    {  
        s = s + a[i];  
    }  
    return s;  
}
```

- $N+3$

Algorithm Rsum(a,n)

```
{  
    if(n<=0) then return 0;  
    else  
        return Rsum(a,n-1)+a[n]  
}
```

- $S(\text{Rsum}(a,n)) = 3(n+1)$

Puzzle

- Glove Selection: There are 22 gloves in a drawer. 5 pairs of red gloves, 4 pairs of yellow and 2 pairs of green. You select gloves in dark and can check them only after selection of all. What is the smallest number of gloves you need to select to have at least one matching pair in the best case? Worst case?

Time Complexity

- $T(P) = \text{Compile Time} + \text{Run Time}.$
- $\text{Runtime} = t_P$
- Identify program Step
- Program step is loosely defined as a syntactically or semantically a meaningful statement which is independent of instance characteristics.
- For iterative step counts must be considered.

```
{  
    S=0;  
    for i=1 to n do  
        s=s+a[i];  
    return s  
}
```

EXAMPLES

Algorithm Rsum(a,n)

```
{  
    if(n<=0) then return 0;  
    else  
        return Rsum(a,n-1)+a[n]  
}
```

- $2 + t \text{ Rsum}(n-1)$
- $2+2+ t \text{ Rsum}(n-2)$
-
-
- $2n+2 \quad \text{when } n>0$

Examples

```
{  
  for i=1 to m do  
    for j=1 to n do  
      c[i,j]=a[i,j]+ b[i,j];  
}
```

The challenge

Q. Will my program be able to solve a large practical input?

Why is my program so slow ?

Why does it run out of memory ?



Introduction- Order of Growth

- **Constant.** A program whose running time's order of growth is constant executes a fixed number of operations to finish its job; consequently its running time does not depend on N .
 - Accessing i^{th} element in an array
- **Logarithmic.** A program whose running time's order of growth is logarithmic is barely slower than a constant-time program
 - Binary search – $\log N$
- **Linear.** Programs that spend a constant amount of time processing each piece of input data, or that are based on a single for loop, are quite common.
 - The order of growth of such a program is said to be linear —its running time is proportional to N .
 - Searching for a key using sequential search
- **Linearithmic.** Algorithms whose running time for a problem of size N has order of growth $N \log N$.
 - Merge sort, Quick Sort

- **Quadratic.** A typical program whose running time has order of growth N^2 has two nested for loops, used for some calculation involving all pairs of N elements.
 - Selection sort, Bubble sort, Insertion sort
- **Cubic.** A typical program whose running time has order of growth N^3 has three nested for loops, used for some calculation involving all triples of N elements.
 - Matrix multiplication, Floyd-Warshall algorithm
- **Exponential.** Programs whose running times are proportional to 2^N or $N!$
 - Exponential algorithms are extremely slow—you will never run one of them to completion for a large problem.
 - But they do exist at large !!!!
 - it would take about $4 \cdot 10^{10}$ years for a computer making a trillion (10^{12}) operations per second to execute 2^{100} operations
 - $100!$ operations, it is still longer than 4.5 billion ($4.5 \cdot 10^9$) years—the estimated age of the planet Earth !!!!!

order of growth	name	typical code framework	description	example	$T(2N) / T(N)$
1	constant	<code>a = b + c;</code>	statement	add two numbers	1
$\log N$	logarithmic	<pre>while (N > 1) { N = N / 2; ... }</pre>	divide in half	binary search	~ 1
N	linear	<pre>for (int i = 0; i < N; i++) { ... }</pre>	loop	find the maximum	2
$N \log N$	linearithmic		divide and conquer	mergesort	~ 2
N^2	quadratic	<pre>for (int i = 0; i < N; i++) for (int j = 0; j < N; j++) { ... }</pre>	double loop	check all pairs	4
N^3	cubic	<pre>for (int i = 0; i < N; i++) for (int j = 0; j < N; j++) for (int k = 0; k < N; k++) { ... }</pre>	triple loop	check all triples	8
2^N	exponential	[see combinatorial search lecture]	exhaustive search	check all subsets	$T(N)$

Asymptotic Notations

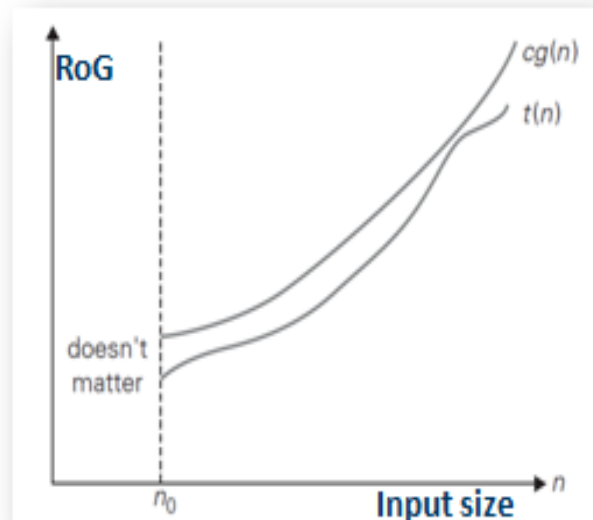
O (big oh)

- This notation gives the **tight upper** bound of the given function.
- A function $f(n)$ is said to be in $O(g(n))$, denoted $f(n) \in O(g(n))$, if $f(n)$ 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 n_0 such that $f(n) \leq cg(n)$ for all $n \geq n_0$.

Examples

- $n \in O(n^2)$,
- $100n + 5 \in O(n^2)$,
- $1/2n(n - 1) \in O(n^2)$.

Note: Analyze the algorithms at larger values of only, i.e. below which do not care for rate of growth.



Problems

Prove the assertion $100n + 5 = O(n^2)$

$$100n+5 = 100n + 5n$$

for all $n \geq 1$

$$= 105n$$

$$c=105$$

$$n_0=1$$

- $3n+2$
- $3n+3$
- $100n+6$
- $10n^2+4n+6$
- $1000n^2+100n-6$

Ω (big omega)

This notation gives the **tight lower bound** of the given function.

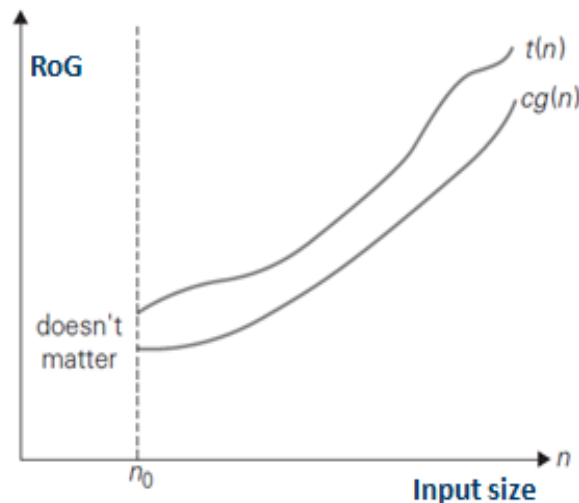
- A function $f(n)$ is said to be in $\Omega(g(n))$, denoted $f(n) \in \Omega(g(n))$, if $f(n)$ is bounded below by some positive constant multiple of $g(n)$ for all large n ,

- i.e., if there exist some positive constant c and some nonnegative integer n_0 such that $f(n) \geq cg(n)$ for all $n \geq n_0$.

Examples

- $n^3 \in \Omega(n^2)$, for $n \geq 0$ ($c=1, n_0=0$)
- $3n+2 = \Omega(n)$
- $3n+3 = \Omega(n)$

Note: Analyze the algorithms at larger values of only, i.e. below which do not care for rate of growth.



- $n^3 \in \Omega(n^2)$, for $n \geq 0$ ($c=1$, $n_0=1$)
- $3n+2 = \Omega(n)$
- $3n+3 = \Omega(n)$
- $100n+6 = \Omega(n)$
- $10n^2 + 4n + 2 = \Omega(n)$

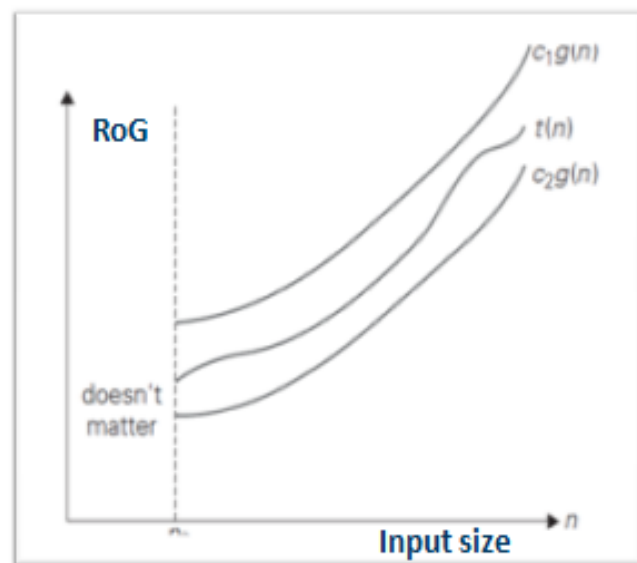
Θ (big theta).

- This notation decides whether the upper and lower bounds of a given function (algorithm) are same or not..
- A function $f(n)$ is said to be in $\Theta(g(n))$, denoted $f(n) \in \Theta(g(n))$, if $f(n)$ 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 constant c_1 and c_2 and some nonnegative integer n_0 such that $c_2 g(n) \leq f(n) \leq c_1 g(n)$ for all $n \geq n_0$.

Examples

- $\frac{1}{2} n(n-1) \in \Theta(n^2)$ ($c_1=1/2$, $c_2=1/4$, $n_0=2$)
- $3n+2 \in \Theta(n)$ ($c_1=4$, $c_2=3$, $n_0=2$)

Note: Analyze the algorithms at larger values of only, i.e. below which do not care for rate of growth.



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



Thank YOU

MODULE- 1 (CONTI)

Presented By

*Manjula L
Assistant Professor
Department of CSE
RNSIT*

Asymptotic Notations(Continued)

THEOREM If $t_1(n) \in O(g_1(n))$ and $t_2(n) \in O(g_2(n))$, then

$$t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\}).$$

(The analogous assertions are true for the Ω and Θ notations as well.)

PROOF (As you will see, the proof extends to orders of growth the following simple fact about four arbitrary real numbers a_1, b_1, a_2 , and b_2 : if $a_1 \leq b_1$ and $a_2 \leq b_2$, then $a_1 + a_2 \leq 2 \max\{b_1, b_2\}$.) Since $t_1(n) \in O(g_1(n))$, there exist some positive constant c_1 and some nonnegative integer n_1 such that

$$t_1(n) \leq c_1 g_1(n) \quad \text{for all } n \geq n_1.$$

Similarly, since $t_2(n) \in O(g_2(n))$,

$$t_2(n) \leq c_2 g_2(n) \quad \text{for all } n \geq n_2.$$

Let us denote $c_3 = \max\{c_1, c_2\}$ and consider $n \geq \max\{n_1, n_2\}$ so that we can use both inequalities. Adding the two inequalities above yields the following:

$$\begin{aligned} t_1(n) + t_2(n) &\leq c_1 g_1(n) + c_2 g_2(n) \\ &\leq c_3 g_1(n) + c_3 g_2(n) = c_3 [g_1(n) + g_2(n)] \\ &\leq c_3 2 \max\{g_1(n), g_2(n)\}. \end{aligned}$$

Hence, $t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})$, with the constants c and n_0 required by the O definition being $2c_3 = 2 \max\{c_1, c_2\}$ and $\max\{n_1, n_2\}$, respectively. ■

So what does this property imply for an algorithm that comprises two consecutively executed parts? It implies that the algorithm's overall efficiency is determined by the part with a larger order of growth, i.e., its least efficient part:

$$\left. \begin{array}{l} t_1(n) \in O(g_1(n)) \\ t_2(n) \in O(g_2(n)) \end{array} \right\} \quad t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\}).$$

USING LIMITS FOR COMPARING ORDER OF GROWTH

$$\lim_{n \rightarrow \infty} \frac{t(n)}{g(n)} = \begin{cases} 0 & \text{implies that } t(n) \text{ has a smaller order of growth than } g(n) \\ c > 0 & \text{implies that } t(n) \text{ has the same order of growth as } g(n) \\ \infty & \text{implies that } t(n) \text{ has a larger order of growth than } g(n).^3 \end{cases}$$

IMPORTANT FORMULAS

- LH Hospital's Rule:

$$\lim_{n \rightarrow \infty} \frac{t(n)}{g(n)} = \lim_{n \rightarrow \infty} \frac{t'(n)}{g'(n)}$$

- Stirling's Formula:

$$n! \approx \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \text{ for large values of } n.$$


Problems

1. Compare the orders of growth of $\frac{1}{2}n(n-1)$ and n^2 .
2. Compare the orders of growth of $\log_2 n$ and \sqrt{n} .
3. Compare the orders of growth of $n!$ and 2^n .

Solutions

1.

$$\lim_{n \rightarrow \infty} \frac{\frac{1}{2}n(n-1)}{n^2} = \frac{1}{2} \lim_{n \rightarrow \infty} \frac{n^2 - n}{n^2} = \frac{1}{2} \lim_{n \rightarrow \infty} \left(1 - \frac{1}{n}\right) = \frac{1}{2}.$$

Since the limit is equal to a positive constant, the functions have the same order of growth or, symbolically, $\frac{1}{2}n(n-1) \in \Theta(n^2)$. 

Solutions

2.

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

Solutions

3.

$$\lim_{n \rightarrow \infty} \frac{n!}{2^n} = \lim_{n \rightarrow \infty} \frac{\sqrt{2\pi n} \left(\frac{n}{e}\right)^n}{2^n} = \lim_{n \rightarrow \infty} \sqrt{2\pi n} \frac{n^n}{2^n e^n} = \lim_{n \rightarrow \infty} \sqrt{2\pi n} \left(\frac{n}{2e}\right)^n = \infty.$$

Mathematical analysis -Non- recursive algorithms

Problem statement

Find the value of the largest element in a list of n numbers. (assume that the list is implemented as an array)

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 $A[i] > \text{maxval}$

maxval $\leftarrow A[i]$

return maxval

Input size ?

n

Basic operation ?

Number of comparison
same of all arrays of
size **n** ?

YES !!!

- No need to distinguish among the worst, average, and best cases.

Mathematical analysis -Non- recursive algorithms

General Plan

1. Decide on a parameter (or parameters) indicating an **input's** size.
2. Identify the algorithm's **basic operation**. (As a rule, it is located in the innermost loop.)
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 -Non- recursive algorithms

Formulae that you may need !!!

$$1. \log_a 1 = 0$$

$$2. \log_a a = 1$$

$$3. \log_a x^y = y \log_a x$$

$$4. \log_a xy = \log_a x + \log_a y$$

$$5. \log_a \frac{x}{y} = \log_a x - \log_a y$$

$$6. a^{\log_b x} = x^{\log_b a}$$

$$7. \log_a x = \frac{\log_b x}{\log_b a} = \log_a b \log_b x$$

Sum Manipulation Formulae

$$1. \sum_{i=l}^u ca_i = c \sum_{i=l}^u a_i$$

$$2. \sum_{i=l}^u (a_i \pm b_i) = \sum_{i=l}^u a_i \pm \sum_{i=l}^u b_i$$

$$3. \sum_{i=l}^u a_i = \sum_{i=l}^m a_i + \sum_{i=m+1}^u a_i, \text{ where } l \leq m < u$$

$$4. \sum_{i=l}^u (a_i - a_{i-1}) = a_u - a_{l-1}$$

Remember Combinatorics ?

1. Number of permutations of an n -element set: $P(n) = n!$

2. Number of k -combinations of an n -element set: $C(n, k) = \frac{n!}{k!(n-k)!}$

3. Number of subsets of an n -element set: 2^n

Mathematical analysis -Non- recursive algorithms

■ Summation Formulae

$$1. \sum_{i=l}^u 1 = \underbrace{1 + 1 + \cdots + 1}_{u-l+1 \text{ times}} = u - l + 1 \text{ (} l, u \text{ are integer limits, } l \leq u \text{); } \sum_{i=1}^n 1 = n$$

$$2. \sum_{i=1}^n i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2} \approx \frac{1}{2}n^2$$

$$3. \sum_{i=1}^n i^2 = 1^2 + 2^2 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6} \approx \frac{1}{3}n^3$$

$$4. \sum_{i=1}^n i^k = 1^k + 2^k + \cdots + n^k \approx \frac{1}{k+1}n^{k+1}$$

$$5. \sum_{i=0}^n a^i = 1 + a + \cdots + a^n = \frac{a^{n+1} - 1}{a - 1} \text{ (} a \neq 1 \text{); } \sum_{i=0}^n 2^i = 2^{n+1} - 1$$

$$6. \sum_{i=1}^n i2^i = 1 \cdot 2 + 2 \cdot 2^2 + \cdots + n2^n = (n-1)2^{n+1} + 2$$

$$7. \sum_{i=1}^n \frac{1}{i} = 1 + \frac{1}{2} + \cdots + \frac{1}{n} \approx \ln n + \gamma, \text{ where } \gamma \approx 0.5772 \dots \text{ (Euler's constant)}$$

$$8. \sum_{i=1}^n \lg i \approx n \lg n$$

Mathematical analysis -Non- recursive algorithms

```

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
  
```

Let **C(n)** be number of times the basic operation is executed, hence

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

Mathematical analysis -Non- recursive algorithms

Element Uniqueness Problem

- Given an array, check whether all the elements in a given array of n elements are distinct

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 $i \leftarrow 0$ to $n - 2$ do

for $j \leftarrow i + 1$ to $n - 1$ do

if $A[i] = A[j]$ return false

return true

Input size ?

n

Basic operation ?



Mathematical analysis -Non- recursive algorithms

Element Uniqueness Problem

- Given an array, check whether all the elements in a given array of n elements are distinct

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 $i \leftarrow 0$ to $n - 2$ do

for $j \leftarrow i + 1$ to $n - 1$ do

if $A[i] = A[j]$ return false

return true

Input size ?

n

Basic operation ?

- Number of comparisons depends only on n ? **No !!!**
 - It depends also on whether there are equal elements in the array ?

Mathematical analysis -Non- recursive algorithms

- What are possible worst-case inputs ?
 - i.e. the inner loop doesn't exit prematurely
- Array with no equal elements (distinct)
- Last two elements are only pair of equals

```

ALGORITHM UniqueElements(A[0..n - 1])
  for i ← 0 to n - 2 do
    for j ← i + 1 to n - 1 do
      if A[i] = A[j ] return false
  return true

```

$$\begin{aligned}
 C_{\text{worst}}(n) &= \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i) \\
 &= \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i = (n-1) \sum_{i=0}^{n-2} 1 - \frac{(n-2)(n-1)}{2} \\
 &= (n-1)^2 - \frac{(n-2)(n-1)}{2} = \frac{(n-1)n}{2} \approx \frac{1}{2}n^2 \in \theta(n^2)
 \end{aligned}$$

Mathematical analysis -Non- recursive algorithms

Matrix Multiplication

- Given two $n \times n$ matrices A and B , compute their product $C=AB$.
- C is an $n \times n$ matrix whose elements are computed as the scalar (dot) products of the rows of matrix A and the columns of matrix B :

$$\begin{array}{c}
 \text{row } i \\
 \begin{array}{c} A \\ \left[\begin{array}{|c|c|c|c|c|} \hline \square & \square & \square & \square & \square \\ \hline \end{array} \right] \\
 \end{array}
 \end{array}
 *
 \begin{array}{c}
 \begin{array}{c} B \\ \left[\begin{array}{|c|} \hline \square \\ \square \\ \square \\ \square \\ \square \\ \square \\ \hline \end{array} \right] \\
 \text{col. } j
 \end{array}
 \end{array}
 =
 \begin{array}{c}
 \begin{array}{c} C \\ \left[\begin{array}{|c|} \hline C[i,j] \\ \hline \end{array} \right] \\
 \end{array}
 \end{array}$$

- where $C[i, j] = A[i, 0]B[0, j] + \dots + A[i, k]B[k, j] + \dots + A[i, n - 1]B[n - 1, j]$ for every pair of indices $0 \leq i, j \leq n - 1$.

Mathematical analysis -Non- recursive algorithms

ALGORITHM MatrixMultiplication(A[o..n – 1, o..n – 1], B[o..n – 1, o..n – 1])

//Multiplies two square matrices of order n by the definition-based algorithm

//Input: Two $n \times n$ 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

Input size ?

Matrix order **n**

Basic operation ?



- Do we need to investigate worst case, best case and average case separately?

No !!!

Mathematical analysis -Non- recursive algorithms

- Let $M(n)$ be total number of multiplications

$$M(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1 = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} n = \sum_{i=0}^{n-1} n^2 = n^3$$

Mathematical analysis -Non- recursive algorithms

Do It Yourself

Consider the following algorithm

ALGORITHM *Mystery(n)*

 //Input: A nonnegative integer n

$S \leftarrow 0$

for $i \leftarrow 1$ **to** n **do**

$S \leftarrow S + i * i$

return S

- What does this algorithm compute?
- What is its basic operation?
- How many times is the basic operation executed?
- What is the efficiency class of this algorithm?
- Suggest an improvement, or a better algorithm altogether, and indicate its efficiency class. If you cannot do it, try to prove that, in fact, it cannot be done.

Mathematical analysis -Non- recursive algorithms

Do It Yourself

Consider the following algorithm

ALGORITHM *Secret*($A[0..n - 1]$)
//Input: An array $A[0..n - 1]$ of n real numbers
 $minval \leftarrow A[0]; maxval \leftarrow A[0]$
for $i \leftarrow 1$ **to** $n - 1$ **do**
 if $A[i] < minval$
 $minval \leftarrow A[i]$
 if $A[i] > maxval$
 $maxval \leftarrow A[i]$
return $maxval - minval$

- What does this algorithm compute?
- What is its basic operation?
- How many times is the basic operation executed?
- What is the efficiency class of this algorithm?
- Suggest an improvement, or a better algorithm altogether, and indicate its efficiency class. If you cannot do it, try to prove that, in fact, it cannot be done.

Mathematical analysis -Non- recursive algorithms

Do It Yourself

Consider the following algorithm

ALGORITHM *Enigma*($A[0..n-1, 0..n-1]$)
//Input: A matrix $A[0..n-1, 0..n-1]$ of real numbers
for $i \leftarrow 0$ **to** $n-2$ **do**
 for $j \leftarrow i+1$ **to** $n-1$ **do**
 if $A[i, j] \neq A[j, i]$
 return false
return true

- What does this algorithm compute?
- What is its basic operation?
- How many times is the basic operation executed?
- What is the efficiency class of this algorithm?
- Suggest an improvement, or a better algorithm altogether, and indicate its efficiency class. If you cannot do it, try to prove that, in fact, it cannot be done.

Mathematical analysis - Recursive algorithms

- Compute the factorial function $F(n) = n!$ for an arbitrary nonnegative integer n .
- i.e. $n! = 1 \dots (n - 1) \cdot n = (n - 1)! \cdot n$ for $n \geq 1$ and $0! = 1$ by definition
- i.e. $F(n) = F(n - 1) \cdot n$
- Following is the recursive algorithm that computes the factorial (n)

ALGORITHM $F(n)$

//Computes $n!$ recursively

//Input: A nonnegative integer n

//Output: The value of $n!$

if $n = 0$ return 1

else return $F(n - 1) * n$

Input size ?

n

Basic operation ?

Multiplication

Mathematical analysis - Recursive algorithms

- Let $M(n)$ be the number of executions of the basic operation.
- $F(n)$ is computed according to the formula

$$F(n) = F(n - 1) \cdot n \text{ for } n > 0,$$

- The number of multiplications $M(n)$ is given by

$$M(n) = \underbrace{M(n-1)}_{\text{to compute } F(n-1)} + \underbrace{1}_{\text{to multiply } F(n-1) \text{ by } n} \text{ for } n > 0$$

ALGORITHM $F(n)$
 if $n = 0$ return 1
 else return $F(n - 1) * n$

- $M(n - 1)$ multiplications are spent to compute $F(n - 1)$, and one more multiplication is needed to multiply the result by n .
- The equation defines the sequence $M(n)$ that we need to find.
- This equation defines $M(n)$ not explicitly, i.e., as a function of n , but implicitly as a function of its value at another point, namely $n - 1$.
- Such equations are called recurrence relations or, for brevity, recurrences

Mathematical analysis - Recursive algorithms

- To solve a recurrence relation say $M(n) = M(n-1) + 1$ for $n > 0$, you need an initial condition that tells us the value with which the sequence starts.
- Observe that this value can be obtained by inspecting the condition that makes the algorithm stop its recursive calls:

If $n = 0$ return 1

ALGORITHM F(n)

if $n = 0$ return 1

else return $F(n - 1) * n$

- This tells us two things
 - First, since the calls stop when $n = 0$, the smallest value of n for which this algorithm is executed and hence $M(n)$ defined is 0
 - Second, by inspecting the pseudocode's exiting line, we can see that when $n = 0$, the algorithm performs no multiplications

$M(0) = 0$

The call stops when $n = 0$

no multiplications when $n = 0$

Mathematical analysis - Recursive algorithms

- The resulting recurrence relation with initial condition is given by

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

$$M(0) = 0.$$

- Observe two functions

- The first is the factorial function $F(n)$ itself; it is defined by the recurrence

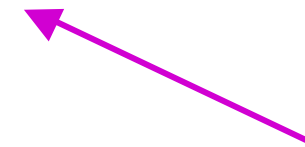
$$F(n) = F(n - 1) \cdot n \text{ for every } n > 0,$$

$$F(0) = 1.$$

- The second is the number of multiplications $M(n)$ needed to compute $F(n)$ by the recursive algorithm

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

$$M(0) = 0.$$



This is to be solved

Mathematical analysis - Recursive algorithms

Solve the following recurrence relation (Backward Substitution)

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

$$M(0) = 0.$$

Solution:

$$\begin{aligned}
 M(n) &= M(n - 1) + 1 && \text{Substitute } n-1 \text{ in place of } n \\
 &= [M(n - 2) + 1] + 1 = M(n - 2) + 2 && \text{Substitute } n-1 \text{ in place of } n \\
 &= [M(n - 3) + 1] + 2 = M(n - 3) + 3 \text{ and so on...}
 \end{aligned}$$

Can you observe an emerging pattern ? $M(n-i)+i$

$$M(n) = M(n - 1) + 1 = \dots = M(n - i) + i = \dots = M(n - n) + n = \mathbf{n}. \text{ As } (M(0)=0)$$

Mathematical analysis - Recursive algorithms

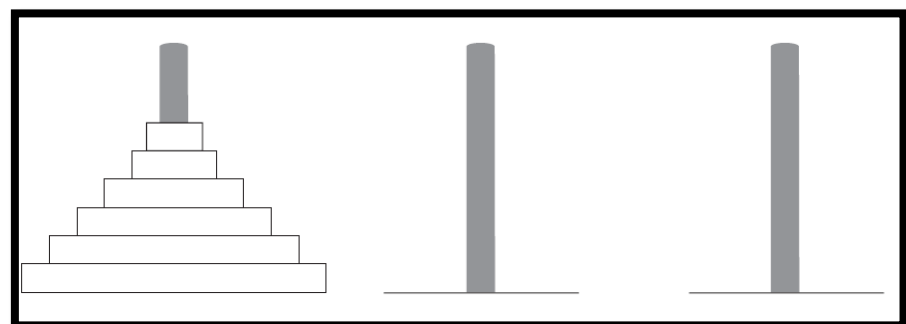
General plan

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;
 1. 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.

Mathematical analysis - Recursive algorithms

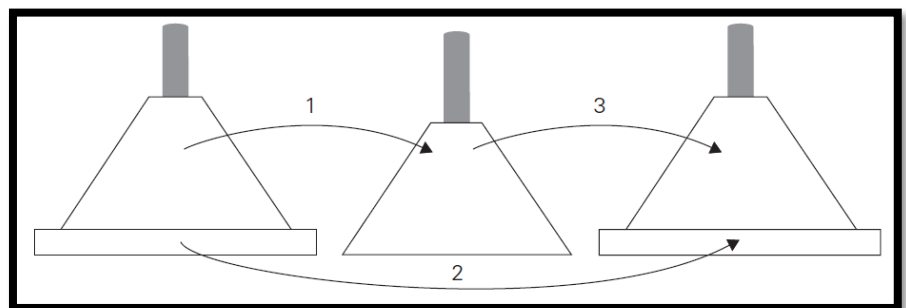
Tower of Hanoi Puzzle

- Given **n** disks of different sizes, goal is to move/slide all of them onto any of **three pegs** with the help of **second peg** as an auxiliary .
- The constraints are
 - Move **only one** disk at a time
 - Larger disk can not be placed on smaller one
- Initially, all the disks are on the first peg in order of size, the largest on the bottom and the smallest on top.



Tower of Hanoi Puzzle

- This problem has a elegant recursive solution
- To move $n > 1$ disks from peg 1 to peg 3 (with peg 2 as auxiliary),
 - we first move recursively $n - 1$ disks from peg 1 to peg 2 (with peg 3 as auxiliary),
- Then move the largest disk directly from peg 1 to peg 3, and,
 - finally, move recursively $n - 1$ disks from peg 2 to peg 3 (using peg 1 as auxiliary).
- Of course, if $n = 1$, we simply move the single disk directly from the source peg to the destination peg



Mathematical analysis - Recursive algorithms

Recurrence relation and the solution approach

- Input size ?
 - Number of discs i.e., **n**
- Basic operation ?
 - Moving one disc
- Hence the number of moves $M(n)$ depends only on **n**
- The recurrence relation is given by

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

- What is the initial condition ?
 - $M(1)=1$
- Hence the obvious recurrence relation is given by

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

$$M(1) = 1.$$

Mathematical analysis - Recursive algorithms

Solve the recurrence relation

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

$$M(1) = 1.$$

Solution

$$\begin{aligned}
 M(n) &= 2M(n - 1) + 1 && \text{substitute } M(n - 1) = 2M(n - 2) + 1 \\
 &= 2[2M(n - 2) + 1] + 1 = 2^2M(n - 2) + 2 + 1 && \text{substitute } M(n - 2) = 2M(n - 3) + 1 \\
 &= 2^2[2M(n - 3) + 1] + 2 + 1 = 2^3M(n - 3) + 2^2 + 2 + 1.
 \end{aligned}$$

Can you guess the next one ?

$$2^4M(n - 4) + 2^3 + 2^2 + 2 + 1,$$

After i substitutions we get

$$M(n) = 2^iM(n - i) + 2^{i-1} + 2^{i-2} + \dots + 2 + 1 = 2^iM(n - i) + 2^i - 1.$$

Since the initial condition is specified for $n = 1$, which is achieved for $i = n - 1$,

$$\begin{aligned}
 M(n) &= 2^{n-1}M(n - (n - 1)) + 2^{n-1} - 1 \\
 &= 2^{n-1}M(1) + 2^{n-1} - 1 = 2^{n-1} + 2^{n-1} - 1 = 2^n - 1.
 \end{aligned}$$

Important problem types

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

Important problem types

Sorting

- The **sorting problem** is to rearrange the items of a given list in nondecreasing order
- As a practical matter, we usually need to sort
 - lists of numbers,
 - characters from an alphabet,
 - character strings, and, most important,
 - records similar to those maintained by schools about their students,
 - libraries about their holdings, and
 - companies about their employees
- In the case of records, we need to choose a piece of information to guide sorting
 - Such a specially chosen piece of information is called a **key**.

Important problem types

Sorting

- Sorting helps searching !!!!
- Although some algorithms are indeed better than others, there is no algorithm that would be the best solution in all situations.
 - Some of the algorithms are simple but relatively slow,
 - while others are faster but more complex;
 - Some work better on randomly ordered inputs,
 - while others do better on almost-sorted lists;
 - some are suitable only for lists residing in the fast memory,
 - while others can be adapted for sorting large files stored on a disk; and so on.

Important problem types

Sorting

Properties of sorting algorithms

- A sorting algorithm is called **stable** if it preserves the relative order of any two equal elements in its input
- In other words, if an input list contains two equal elements in positions i and j where $i < j$, then in the sorted list they have to be in positions i^* and j^* respectively, such that $i^* < j^*$
- This property can be desirable if, for example, we have a list of students sorted alphabetically and we want to sort it according to student GPA:
 - a stable algorithm will yield a list in which students with the same GPA will still be sorted alphabetically.
- Generally speaking, algorithms that can exchange keys located far apart are not stable, but they usually work faster

Important problem types

Sorting

Properties of sorting algorithms

- A sorting algorithm is called **in-place** if it does not require extra memory, except, possibly, for a few memory units

Important problem types

Sorting

Better Example for stable sort

Consider the following example of student names and their respective class section

(Dave, A)
(Alice, B)
(Ken, A)
(Eric, B)
(Carol, A)

Sort the data according to names, the sorted list will not be grouped according to sections !!!!

(Alice, B)
(Carol, A)
(Dave, A)
(Eric, B)
(Ken, A)

Sort again to obtain list of students section wise too.

The dataset is now sorted according to sections, but not according to names.

(Carol, A)
(Dave, A)
(Ken, A)
(Eric, B)
(Alice, B)

In the name sorted list the tuple (Alice, B) was before (Eric, B)

A stable sorting algorithm would result in

(Carol, A)
(Dave, A)
(Ken, A)
(Alice, B)
(Eric, B)

Important problem types

- **Searching**
- The ***searching problem*** deals with finding a given value, called a ***search key***, in a given set (or a multiset, which permits several elements to have the same value).
 - Sequential / linear
 - Binary search (sorted input is a prerequisite but very efficient for large database)

Important problem types

String processing

- A ***string*** is a sequence of characters from an alphabet.
- Strings of particular interest are text strings which comprise letters, numbers, and special characters;
 - bit strings, which comprise zeros and ones; and
 - gene sequences, which can be modeled by strings of characters from the four-character alphabet {A, C, G, T}
- One particular problem—that of searching for a given word in a text—has attracted special attention from researchers

Important problem types

- **Graph problems**
- a ***graph*** can be thought of as a collection of points called **vertices**, some of which are connected by line segments called **edges**.
- Graphs can be used for modeling a wide variety of applications, including
 - transportation,
 - communication, social and economic networks,
 - project scheduling, and
 - games

Important problem types

Combinatorial Problems

- These are problems that ask, explicitly or implicitly, to find a combinatorial object—such as a permutation, a combination, or a subset—that satisfies certain constraints.

Numerical Problems

- ***Numerical problems***, another large special area of applications, are problems that involve mathematical objects of continuous nature:
 - solving equations and systems of equations,
 - computing definite integrals,
 - evaluating functions, and so on.

