Algorithms - Introduction

Nandini Mukherjee

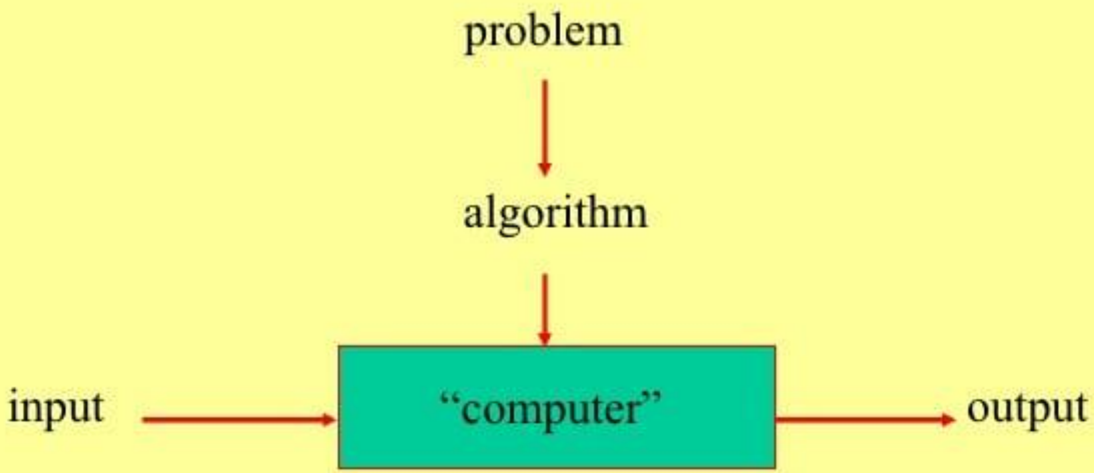
Department of Computer Science and Engineering

Jadavpur University

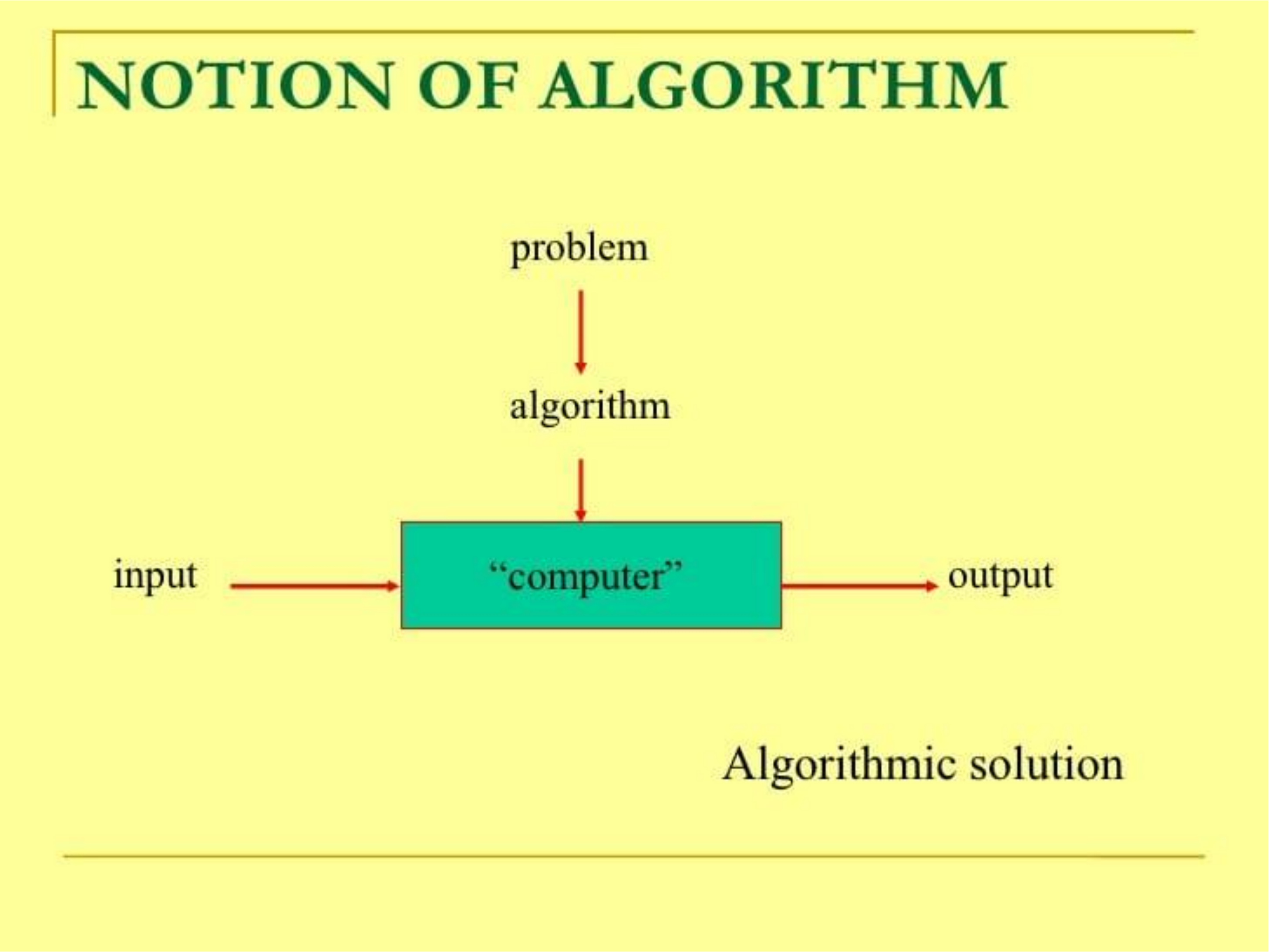
What is an algorithm?

- No agreed-to definition of "algorithm" exists.
- A simple definition: A finite set of unambiguous instructions for solving a problem.
- Knuth defined the five properties of an algorithm:
  - Finiteness: "An algorithm must always terminate after a finite number of steps ... a very finite number, a reasonable number"
  - Definiteness: "Each step of an algorithm must be precisely defined; the actions to be carried out must be rigorously and unambiguously specified for each case"
  - Input: "...quantities which are given to it initially before the algorithm begins. These inputs are taken from specified sets of objects"
  - Output: "...quantities which have a specified relation to the inputs"
  - Effectiveness: "... all of the operations to be performed in the algorithm must be sufficiently basic that they can in principle be done exactly and in a finite length of time by a man using paper and pencil"

NOTION OF ALGORITHM



Algorithmic solution



- Problem – Sorting
- Algorithm?
- Selection Sort
- Insertion Sort
- Merge Sort
- Bubble Sort
- Quick Sort
- ……

Performance of an algorithm matters

Basic Issues

- How to design an algorithm
- How to express an algorithm
- Proving correctness of algorithms
- Efficiency / Performance
  - Empirical Analysis
  - Theoretical Analysis
- Optimality

Empirical Analysis - Why not just run the program?

- Machine architecture
- Operating System and libraries
- Instructions used – compilers
- Programming languages
- Programmer’s style
- …..

So we focus on theoretical analysis.

Analysis of Algorithms

- The theoretical study of computer-program performance and resource usage.
- What other things are important?
  - modularity
  - correctness
  - maintainability
  - functionality
  - robustness
  - user-friendliness
  - programmer time
  - simplicity
  - extensibility
  - reliability

Analysis of Algorithms

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

How to measure the performance

- Length of the program (lines of code)
- Ease of programming (bugs, maintenance)
- Memory required
- Running time
  - Became a dominant standard
  - Quantifiable and easily comparable
  - Often the critical bottleneck

How do we measure running time theoretically?

How do we measure running time?

- Lines of code?
- Number of loops?
- Number of variables used?
- ….
- *The basic idea is to count the number of basic operations*
- And express the performance as a function of input size

How do we measure running time?

- However, there can be many input instances
- Which input instances should be the basis of our judgement?
- Generally, we seek upper bounds on the running time, because everybody likes a guarantee.
- We define
  - Best case – generally unrealistic
  - Worst case - $T(n)$ = maximum running time on any input of size n. Guarantees the performance
  - Average case - $T(n)$ = expected running time over all inputs of size n.
    - Needs assumption on statistical distribution of inputs.

How do we measure running time?

- What is the worst case running time of Algorithm A?
  - It depends on the speed of our computer: relative speed (on the same machine),
  - absolute speed (on different machines).
- How to make the running time machine independent?
- Look at ***growth*** of $T(n)$ as $n \to \infty$ - ***Asymptotic Analysis***

How do we measure running time?

Examples

- Vector addition **Z = A+B**

```
for (int i=0; i<n; i++)
    Z[i] = A[i] + B[i];
```

$T(n) = c\ n$

- Vector (inner) multiplication **z =**

```
z = 0;
for (int i=0; i<n; i++)
    z = z + A[i]*B[i];
```

$T(n) = c' + c_1 n$

27

Examples

- Vector (outer) multiplication **Z = A*B^T**

```
for (int i=0; i<n; i++)
    for (int j=0; j<n; j++)
        Z[i,j] = A[i] * B[j];
```

$T(n) = c_2 n^2;$

- A program does all the above

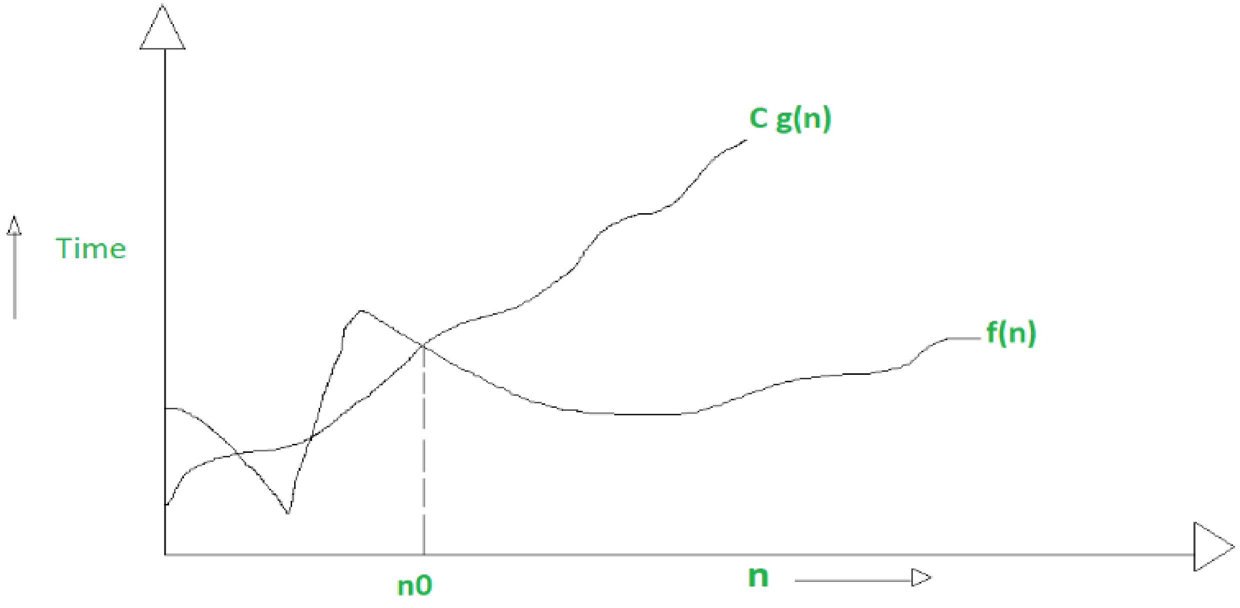
$T(n) = c_0 + c_1 n + c_2 n^2;$

28

How do we measure running time?

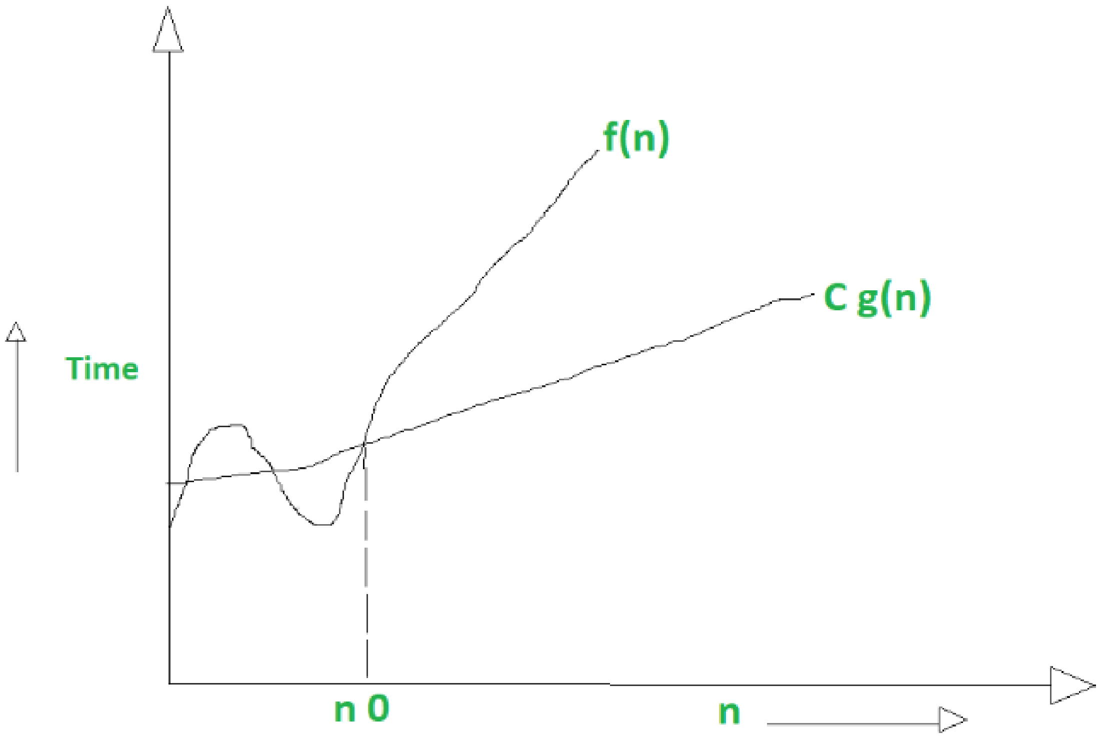
- The final expression may become too complicated
- Difficult to compare two such expressions
- Asymptotic behavior (as n increases) depends on the leading term

f(n) = O(g(n)) => g(n) is the upper bound



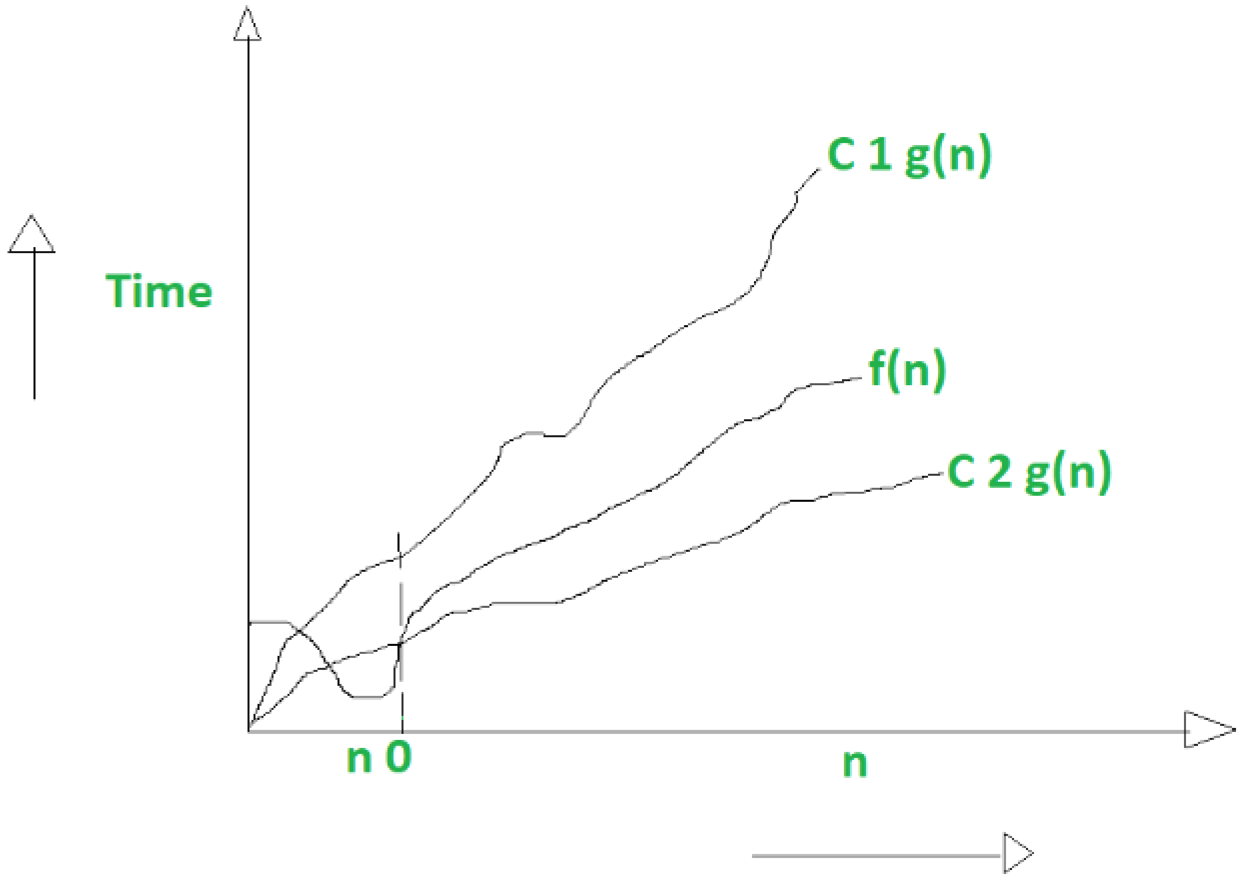
$0 <= f(n) <= c\ g(n)$ for all $n >= n0$

$f(n) = \Omega(g\ (n)) \Rightarrow g(n)$ is the lower bound



`0<= C g(n) <= f(n) for all n>=n 0`

$$f(n) = \Theta(g(n))$$



Algorithm design strategies

- Brute force
- Divide and Conquer
- Decrease and Conquer
- Transform and Conquer
- Greedy approach
- Dynamic programming
- Backtracking and Branch and Bound

Recursion

- Recursion is a powerful problem solving tool
- A function directly or indirectly makes a call to itself
- Many algorithms are easily expressed using recursion

Implementation of recursion

- Implemented using a stack and activation records
- Each time when a function is called, a new activation record is pushed onto the stack
- When a function returns, the stack is popped and the activation record of the calling method appears on top of the stack
- Each successive function call brings you closer to the solution – general case
- A case for which the answer is known (and can be expressed without recursion) is called a base case

Solving recurrences

- Recurrences are often applied for *Divide-and-Conquer* algorithms

Methods for solving recurrences

- Substitution method
  - *The most general method:*
    1. Guess the form of the solution
    2. Verify by induction
    3. Solve for constant
- Recursion-tree method
- The master method

Recursion-tree method

- A recursion tree models the costs (time) of a recursive execution of an algorithm
- The recursion-tree method can be unreliable
- The recursion-tree method promotes intuition
- The recursion tree method is good for generating guesses for the substitution method

The master method

- The master method applies to recurrences of the form

$$T(n) = a\,T(n/b) + f(n)\,,$$

where $a \geq 1$, $b > 1$, and f is asymptotically positive.

- Three cases –

CASE 1: $f(n) = O(n^{\log_b a - \varepsilon})$, constant $\varepsilon > 0$

$\Rightarrow T(n) = \Theta(n^{\log_b a})$.

CASE 2: $f(n) = \Theta(n^{\log_b a} \lg^k n)$, constant $k \geq 0$

$\Rightarrow T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.

CASE 3: $f(n) = \Omega(n^{\log_b a + \varepsilon})$, constant $\varepsilon > 0$,
and regularity condition

$\Rightarrow T(n) = \Theta(f(n))$.

Analysis of algorithms

- So far we considered only a single operation
- What happens if a sequence of operations are performed on a data structure?
- And if the operations have different costs?
- Next lecture – Amortized Analysis