Algorithms Design and Complexity Analysis

Algorithm

- Sequence of statements which solve the problem logically
- Need not to belong one particular language
- Sequence of English statements can also be algorithm
- It is not a computer program
- A problem can have many algorithms

Properties of algorithm –

- Input: Set of initial conditions and required data
- Definiteness: No ambiguity
- Effectiveness: Statements should be basic not complex
- Finiteness: The Execution must be finish after finite number of steps
- Output: Must provide desired output

Program design using algorithm

Two phase process –

1. Problem analysis:

   Analyze and understand the user defined problem.

   Write algorithm using basic algorithm construct.

2. Implementation:

   Translate the algorithm into desired programming language.

Program vs. Software

Computer Program ?

Set of instructions to perform some specific task

Is Program itself a Software ?

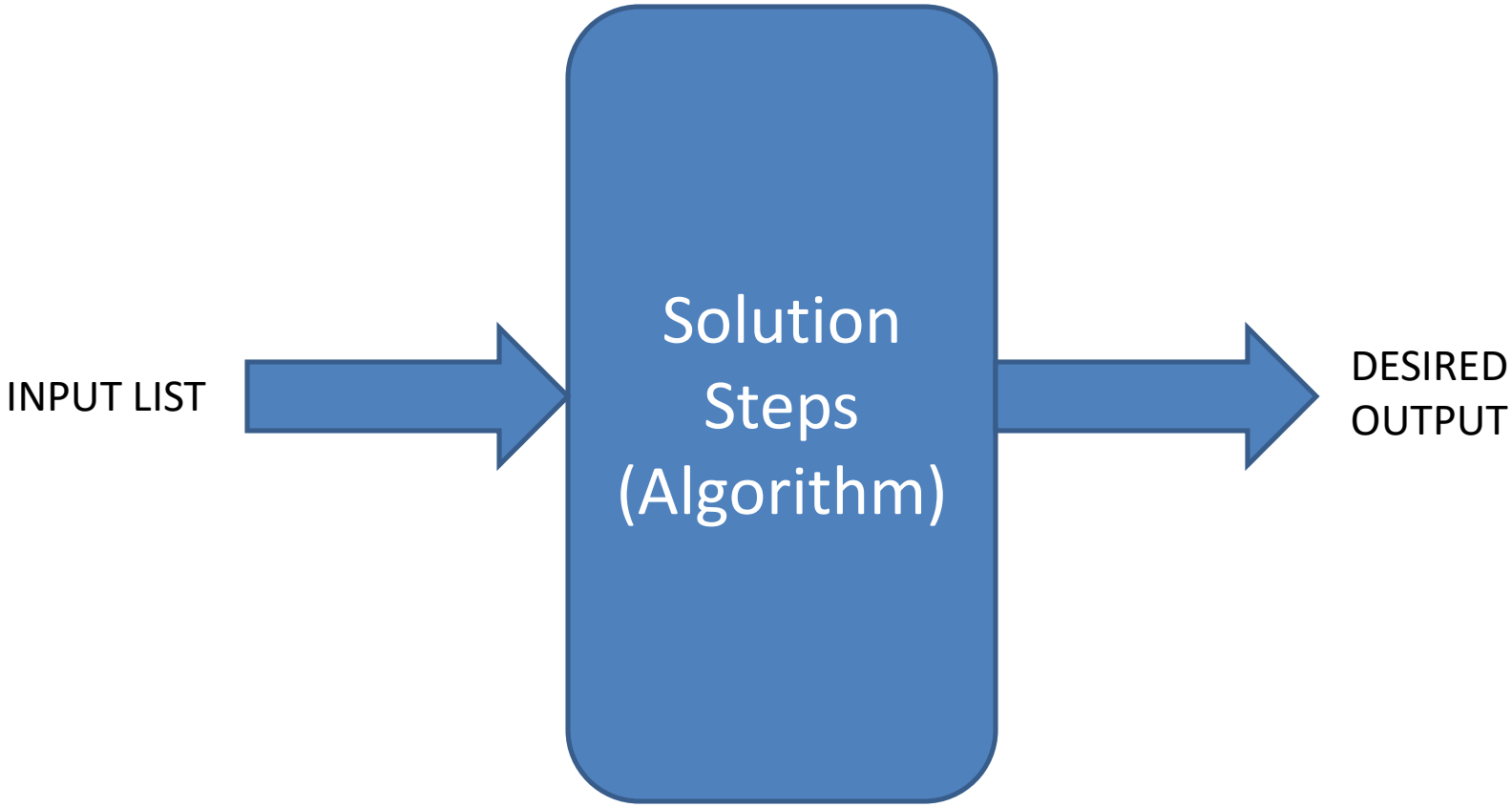
NO, Program is small part of software.

Software merely comprises of Group of Programs (Source code), Documentation, Test cases,

Input or Output Description etc.

Informally

- Algorithm is just a logical form of solution for any problem which can be implemented in any of the programming language.
- Algorithm always takes some input and produce some desired output.



Approaches for designing algorithms

Greedy Approach –

At each step, it selects the best possible solution. Ex. Shortest path algorithm

Divide and Conquer –

a. Divide problem into smaller problem

b. Solve each smaller problem

c. Combine the result of smaller of problem

Ex. Quick sort and merge sort etc.

Non recursive –

Recursion is very powerful but some compiler doesn't support the this feature.

Therefore, this approach uses iterative concept for designing algorithm.

Randomized –

On the base of random feature of problem like randomness in input data, is used to design the algorithm. Ex. Quick sort

Modular programming approach –

Divide big problem into smaller problem (module)

Solve each smaller problem using different algorithm design

Combine them to design algorithm for big problem

Constructs of algorithm

1. Identifying Numbers – Each statement should assigned a number.
2. Comment – Comment may be used to make algorithm easily readable and should not assigned any step number.
3. Variable – Generally, variable names will use capital letters.
4. Input/output – For taking input from user Read construct and for displaying any message or any variable value Print/Write construct can be used.
5. Control Structure –

   Controlling statements may of three types:

   Sequential logic – Write numbered steps continuously unless any contrary statement.

   Selection logic – To select statement from many statements, If – else or its variations can be used.

   Iteration logic – To repeat some statements, loops like while, for and do….while can be used.

Searching an element in array (Version 1)

Algorithm - An array ARR of N elements is given. LOC represents the index of element in array ARR, if element found. Element to be searched is K and I is a simple variable to be used in loop.

Step 1: Initialize LOC =-1,
Step 2: Repeat upto step 4 for I=0 to N-1
Step 3: If ARR[I] == K then,
Step 4: LOC = I
[End of step 3]
[End of step 2]
Step 5: If LOC >= 0 then,
Step 6: Print “Element is at LOC index in array ARR”
Step 7: Else
Step 8: Print “Element is not found”
[End of step 5]
Step 9: Exit

Searching an element in array (Version 2)

Algorithm - An array ARR of N elements is given. LOC represents the index of element in array ARR, if element found. Element to be searched is K and I is a simple variable to be used in loop.

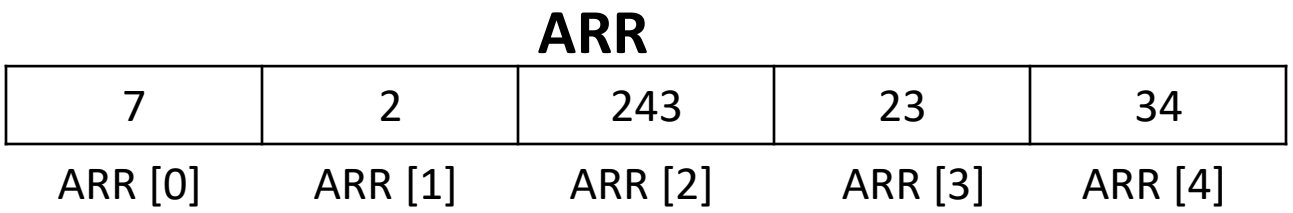
Step 1: Initialize LOC =-1, I =0
Step 2: Repeat upto step 5 while I <N
Step 3: If ARR[I] == K then,
Step 4: LOC = I
[End of step 3]
Step 5: I = I+1
[End of step 2]
Step 6: If LOC >= 0 then,
Step 7: Print “Element is at LOC index in array ARR”
Step 8: Else
Step 9: Print “Element is not found”
[End of step 5]
Step 10: Exit

Searching an element in array

Input:

Given

K = 6 , N = 5



Output: Element is not found

Concept of Sub-algorithm

Sub-algorithm –

Instead of writing an algorithm for large problem we can decompose the original problem into sub problems and algorithm for these sub problems are called as sub algorithms. We can call sub-algorithm in algorithm by passing some required arguments.

Example –

Binary search algorithm, we follow two steps

1. Sort the given list
2. Search the element or data

Algorithm analysis

- Concerned with demand for resources and performance

A-priori estimates –

Performance analysis

A-posteriori analysis –

Measurement & testing

Algorithm analysis (Cont···)

Aim –

To understand, how to select an algorithm for a given problem

Assumption –

1. The algorithms we wish to compare are correct.
2. The algorithm that probably takes lesser time is a better one.

Time –

- We do not bother about the Exact time taken on any machine. **WHY ????**
  - The computational/storage power on different machines vary

Therefore, let us assume that

- The cost of executing a statement is some abstract cost *ci*
- *a* constant amount of time is required to execute each line
- constant time is unity

Methods of analysis

Empirical analysis –

- Implement the solution
- No assumptions

Asymptotic analysis –

- Growth rate function of algorithm with respect to time on increasing the input size
- Asymptotic notations like Big O, Omega and Theta etc.

Efficiency

- Find out possible solutions or algorithms for given problem
- Select the feasible algorithm which can easily used in practice.
- Find space requirement for algorithm. Generally, it is equal to size of input.
- Find the time requirement. Generally, a function in terms of size of input.

Complexity

- Major criteria for complexity of any algorithm is comparison of keys and moving the data i.e. number of times the key is compared and data is moved.
- If space is fixed then only run time will be considered for obtaining the complexity of algorithm.
- We do not bother about the Exact time taken on any machine.

We analyze basically 3 cases for complexity of algorithms –

1. Best case - Ω (Omega)
2. Average case - θ (Theta)
3. Worst case - O (Big O)

Illustration 1: Searching an element in array

```
1:      Let LOC =-1, I =0
2:      while I <N
3:              If ARR[I] == K then,
4:                      LOC = I
5:              I = I+1
6:      If LOC >= 0 then,
7:              Print “Element is at LOC index in array ARR”
8:      Else    Print “Element is not found”
```

C1
C2
C3
C4
C5
C6
C7
C8

Total Steps = Total time =

1* C1 + (N+1)*C2 + N * C3 + N * C4 + N * C5 + 1 * C6 + 1 * C7 or 1 * C8

Illustration 1(Cont···), array with distinct elements

```
1:      Let LOC =-1, I =0                                              C1
2:      while I <N                                                     C2
3:              If ARR[I] == K then,                                   C3
4:                      LOC = I                                        C4
5:              I = I+1                                                C5
6:      If LOC >= 0 then,                                              C6
7:              Print “Element is at LOC index in array ARR”           C7
8:      Else    Print “Element is not found”                           C8
```

Total Steps = Total time =

1* C1 + (N+1)*C2 + N * C3 + 1 * C4 + N * C5 + 1 * C6 + 1 * C7 or 1 * C8

Illustration 2: Largest Element in array

```
1:      let max = x[0]                          C1
2:      for i = 1 to n-1                        C2
3:          if x[i] > max then,                 C3
4:              max = x[i]                      C4
5:      Print max                               C5
```

Descending Order –

Total Steps = Total time = 1* C1 + n*C2 + (n-1)* C3 + 1* C5

Ascending Order –

Total Steps = Total time = 1* C1 + n*C2 + (n-1)* C3 + (n-1) * C4 + 1* C5

Observations

- The running time of an algorithm differs with respect to the nature of the input instance
- But, It is often difficult to get precise measurements of ci's
- The running time of an algorithm almost always expressed as a functions of the input size.

Relook on complexity of above algorithms

A relook at the time complexity expressions we have obtained so far

- Searching an element in array

Time complexity

$= 1* C1 + (N+1)*C2 + N * C3 + N * C4 + N * C5 + 1 * C6 + 1 * C7$

$= C1 + N*C2 + C2 + N*C3 + N * C4 + N *C5 + C6 + C7$

$= N + N + N + N + 4$

$= 4N + 4$

- Searching an element in array (distinct element)

Time complexity

$= 1* C1 + (N+1)*C2 + N * C3 + 1 * C4 + N * C5 + 1 * C6 + 1 * C7$

$= C1 + N*C2 + C2 + N*C3 + C4 + N *C5 + C6 + C7$

$= N + N + N + 5$

$= 3N + 4$

➢ Largest Element in array

Descending Order –

Time Complexity = 1* C1 + n*C2 + (n-1)* C3 + 1* C5

= C1 + n * C2 + n * C3 – C3 + C5

= 2n + 1

Ascending Order –

Time Complexity = 1* C1 + n*C2 + (n-1)* C3 + (n-1) * C4 + 1* C5

= C1 + n * C2 + n * C3 – C3 + n * C4 – C4 +C5

= 3n

Asymptotic Notations

Big –O (Worst Case) –

Definition:

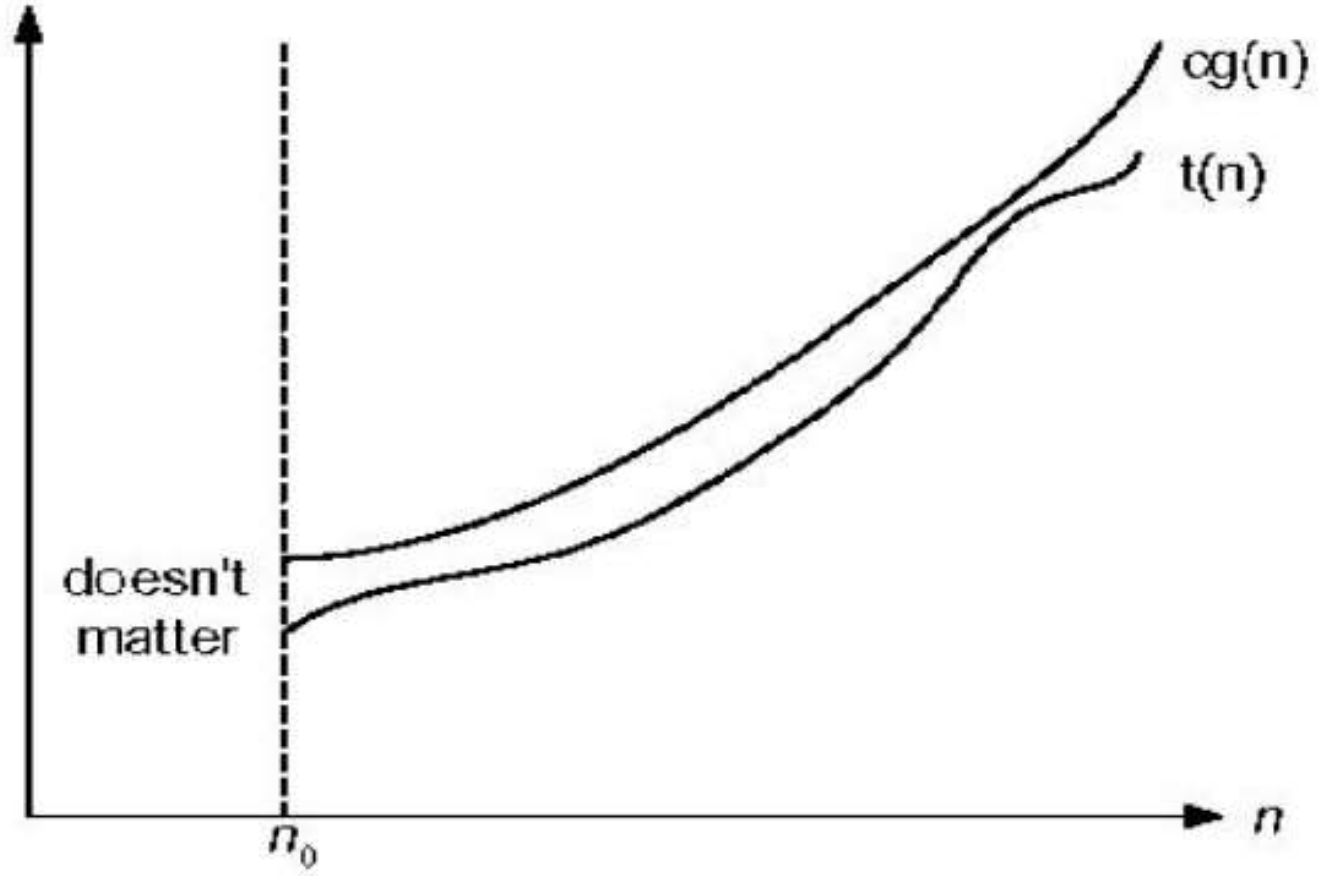
for a given function g(n), we say that

O(g(n)) = {f(n)| if there exists positive constants c and n_o such that,

0<=f(n)<=cg(n) for all n>n_o}

- Defines a tight upper bound for a function
- allows us to keep track of the leading term while ignoring smaller terms
- f(n)= O(g(n)) implies that
  - g(n)grows at least as fast as f(n) **or**
  - f(n)is of the order at most g(n)

Big –O Notation



Asymptotic Notations (Cont···)

Big – Ω (Best Case) –

Definition:

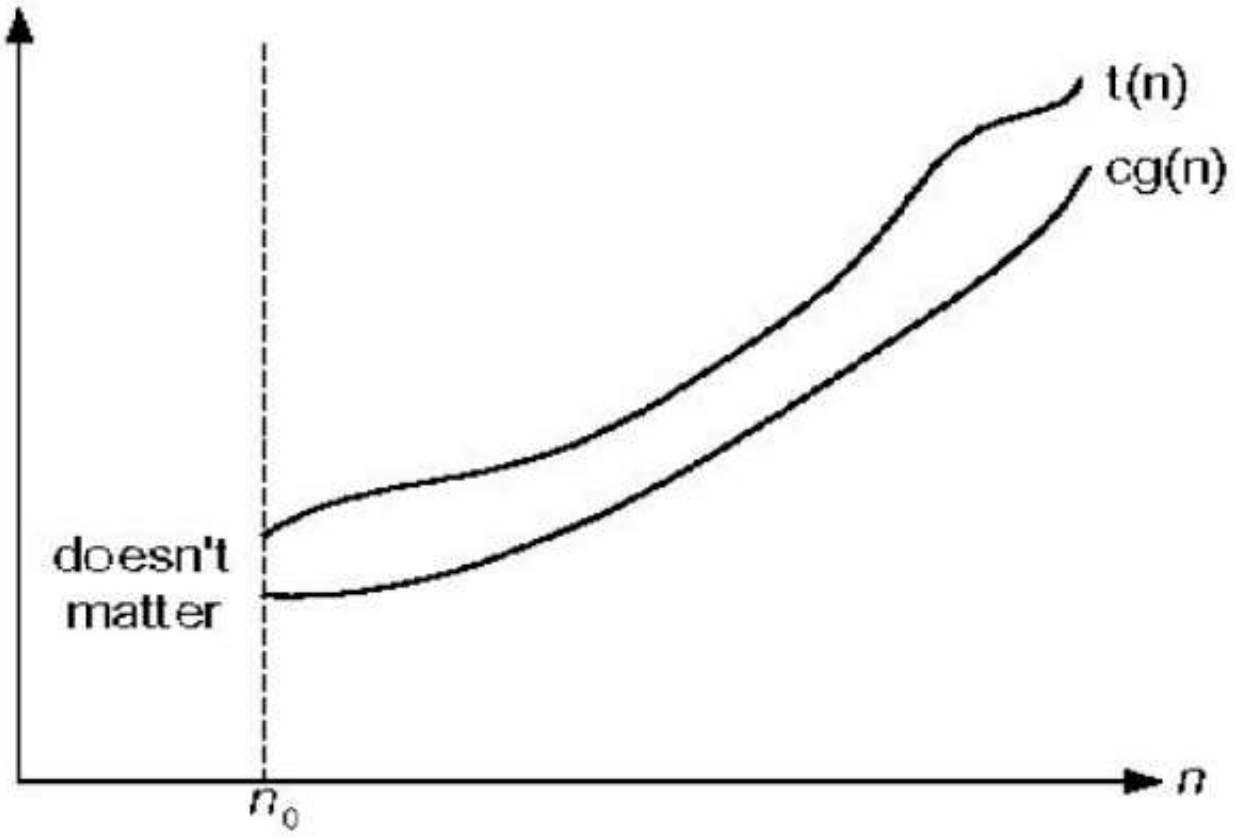
for a given function g(n), we say that

Ω(g(n)) = {f(n)| if there exists positive constants c and n_o such that,

0<=cg(n)<=f(n) for all n>n_o}

- Defines tight lower bound for a function
- allows us to keep track of the leading term while ignoring smaller terms
- f(n)= Ω (g(n)) implies that
  - g(n) grows at most as fast as f(n) **or**
  - f(n) is of the order at least g(n)

Big –Ω Notation



Asymptotic Notations (Cont···)

Big – θ (Average Case) –

Definition:

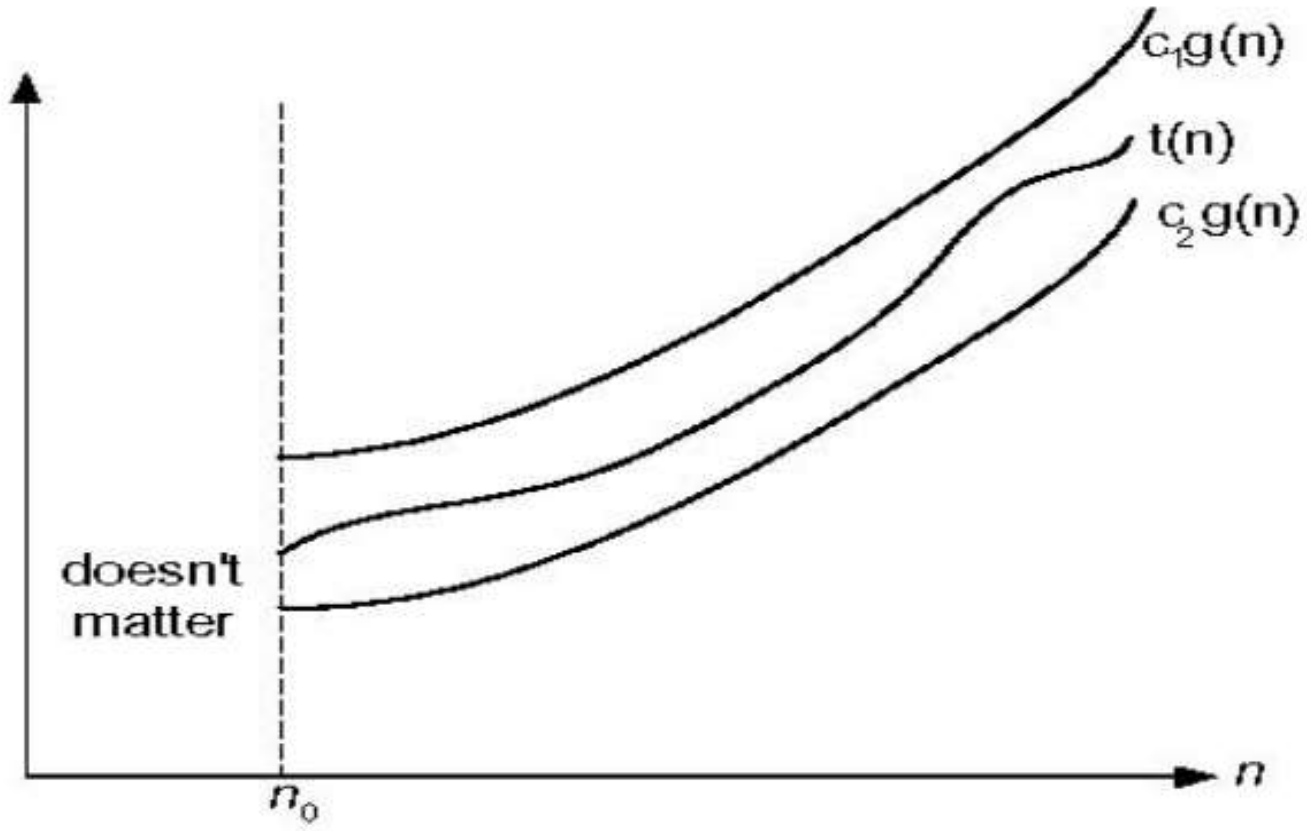
for a given function g(n), we say that

θ(g(n)) = {f(n)| if there exists positive constants c1 and c2 and n_o such that,

0<=c1g(n)<=f(n)<=c2g(n) for all $n>n_o$}

- f(n)=θ(g(n)) **iff** f(n)=O(g(n)) and f(n)= Ω(g(n))
- Defines tight lower bound and upper bound for a function, at the same time

Big – θ Notation



Basic asymptotic efficiency classes

1	Constant
logn	Logarithmic
n	Linear
nlogn	nlogn
n^2	Quadratic
n^3	Cubic
2^n	Exponential
n!	Factorial

An interesting observation

	n	$n \lg n$	N^2	N^3	1.5^n	2^n	$n!$
n=10	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 4 sec
n=30	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} yrs
n=50	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 yrs	very long
n=100	< 1 sec	< 1 sec	< 1 sec	1 sec	12.89 yrs	10^{17} yrs	Very long
n=1000	< 1 sec	< 1 sec	1 sec	18 min sec	very long	very long	very long
n=10K	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
n=100K	< 1 sec	2 sec	3 hrs	32 yrs	very long	very long	very long
n=1M	1 sec	20 sec	12 days	31.71 yrs	very long	very long	very long

Big –O Notation Illustrations

Function	notation in O
$f(n) = 5n + 8$	$f(n) = O(?)$
$f(n) = n^2 + 3n - 8$	$f(n) = O(?)$
$F(n) = 12n^2 - 11$	$f(n) = O(?)$
$F(n) = 5*2^n + n^2$	$f(n) = O(?)$
$f(n) = 3n + 8$	$F(n) = O(n^2)$?
$f(n) = 5n + 8$	$f(n) = O(1)$?

Increasing Order of time complexities

In general,

$n^n > n! > (\text{constant})^n > n^{(\text{constant})}$

And

$n > \sqrt{n} > \log n$

$$
\begin{array}{ccccccccccc}
2^{2^{n+1}} & > & 2^{2^n} & > & (n+1)! & > & n! & > & e^n & > \\
n \cdot 2^n & > & 2^n & > & (\frac{3}{2})^n & > & \begin{array}{c} n^{\lg \lg n} \\ (\lg n)^{\lg n} \end{array} & > & (\lg n)! & > \\
n^3 & > & \begin{array}{c} n^2 \\ 4^{\lg n} \end{array} & > & \begin{array}{c} n \lg n \\ \lg(n!) \end{array} & > & \begin{array}{c} n \\ 2^{\lg n} \end{array} & > & (\sqrt{2})^{\lg n} & > \\
2^{\sqrt{2 \lg n}} & > & \lg^2 n & > & \ln n & > & \sqrt{\lg n} & > & \ln \ln n & > \\
2^{\lg^* n} & > & \begin{array}{c} \lg^*(\lg n) \\ \lg^* n \end{array} & > & \lg(\lg^* n) & > & \begin{array}{c} 1 \\ n^{1/\lg n} \end{array} & & &
\end{array}
$$

Software development life cycle (SDLC)

1. Requirement analysis
2. Specification
3. Design
4. Coding
5. Testing
6. Deployment
7. Maintenance

Summary

- What is algorithm ?
- Constructs of algorithm to follow while writing algorithm
- Difference between algorithm, program and software
- Algorithm analysis and complexity
- Asymptotic notations to represent the time complexity
- SDLC phases