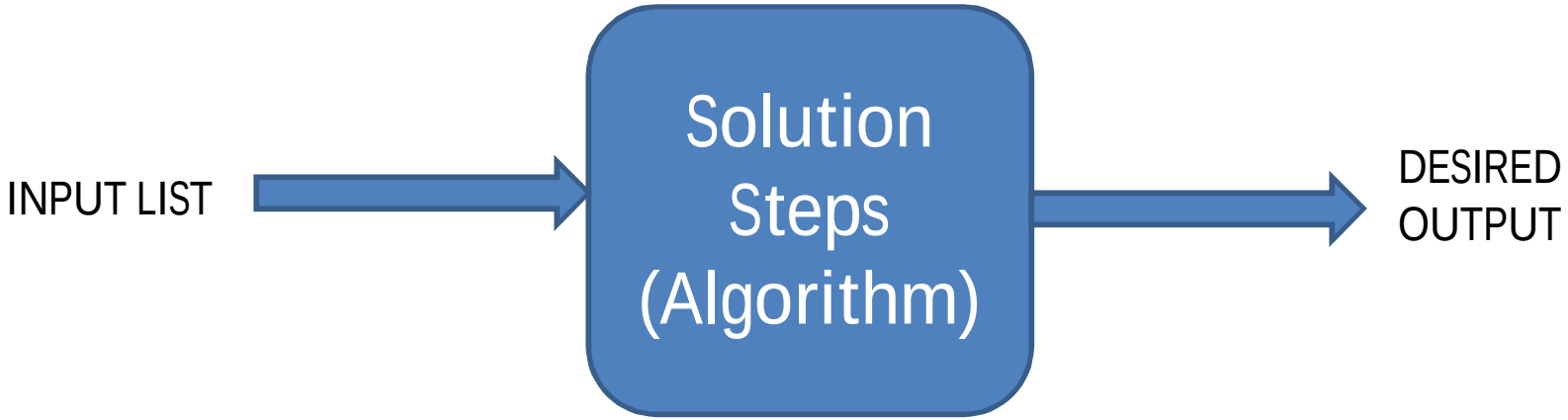
Algorithms Design and Complexity Analysis

Algorithm

- Well defined Sequence of statements which solve the problem logically
- Need not to belong one particular language
- Well defined Sequence of English statements can also be algorithm
- It is not a computer program
- A problem can have many algorithms
- **Algorithm always takes some input and produce some desired output.**



Properties of algorithm

- **Input**: Zero or more quantities are externally supplied.
- **Definiteness**: Each instruction of the algorithm should be clear and unambiguous.
- **Effectiveness:** Statements should be basic not complex
- **Finiteness**: The Execution must be finish after finite number of steps
- **Output**: Must provide desired output

An algorithm can be expressed in three ways:-

(i) in the form of pseudo code
(ii) in the form of a flowchart
(iii) Using program in any specific programming language

What is Good Algorithm?

- Good algorithm is an efficient algorithm.
- What is efficient?
- Efficient means which has small running time and also it takes less memory.
- **Hence, time and space are two measures of efficiency for an algorithm.**

Algorithm Vs. program?

- **a program** is an *implementation* of an algorithm to be run on a specific computer and operating system.
- **an algorithm** is more abstract – it does not deal with machine specific details – think of it as a *method* to solve a problem.

Program vs. Software

Computer Program ?

Well defined set of instructions written any programming language to perform some specific task.

Is Program itself a Software ?

NO, Program is small part of software.

Software generally comprises of Group of Programs (Source code), Documentation, Test cases, Input or Output Description etc.

Program design using algorithm

Two phase process –

1. **Problem analysis**:

   Analyze and understand the user defined problem.

   Write algorithm using basic **algorithmic construct**.

2. **Implementation:**

   Translate the algorithm into desired programming language.

Concept of Sub-algorithm

Sub-algorithm –

When the problem is very complex, it is divided into several independent sub-problems. Each sub-problem is called sub-algorithm and can be developed independently. These sub-problems can be combined together to form the solution for the entire problem. Thus by using sub-algorithms the complex problems can be solved easily.

➢The relationship between an algorithm and a sub-algorithm is similar to the relationship between a main program (function) and a sub-program (sub-function) in a programming language.

Example –

Binary search algorithm, we follow two steps

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

Why to Develop Sub-Algorithms?

- Smaller problems are less complicated and easier to solve
- It allows efficient task management
  - ✓John does problem 1 and Mary does problem 2
- Whole project is not completely stopped if one of the smaller problems cannot be solved immediately.
  - ✓Solve the other problems
  - ✓Then research and solve the difficult part of the problem.
- We may be able to reuse the solution to the smaller problem in another large problem

Approaches for designing algorithms

- **Greedy Approach –**At each step, it selects the best possible solution.

  The “greedy” sense of the “greedy algorithm” actually comes from the idea that we ignore what decisions we have made in previous stages, and simply focus on the sub-optimal solution in every single stage. In this way, we attempt to derive the overall optimal solution.

- ✓ Ex. **Shortest path algorithm**, **Kruskal's algorithm** and **Prim's algorithm** for finding **minimum spanning** trees.

- **Divide and Conquer –**The divide and conquer approach involves the following steps at each level.

  ✓**Divide** – The original problem is divided into sub-problems.
  ✓**Conquer** – The sub-problems are solved recursively.
  ✓**Combine** – The solutions of the sub-problems are combined together to get the solution of the original problem.

  ✓ Ex. Quick sort and merge sort etc.

Approaches for designing algorithms

➢**Dynamic Programming-** Dynamic programming is an optimization technique, which divides the problem into smaller sub-problems and after solving each sub-problem, dynamic programming combines all the solutions to get ultimate solution.
✓Unlike divide and conquer method, dynamic programming reuses the solution to the sub-problems many times.
✓Recursive algorithm for Fibonacci Series is an example of dynamic programming.

- **Randomized –** An algorithm that uses random numbers to decide what to do next anywhere in its logic is called Randomized Algorithm.
- ✓ Ex. In Quick Sort, using a random number to choose a pivot.

Algorithm analysis

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.

Can you suggest any approach to compare the algorithm?

Approach 1: Empirical analysis (Experimental Studies) approach–

1. Implement the algorithm into any programming language.

2. Run the program with inputs of varying size and composition.

3. Use a method like System.currentTimeMillis() to get an accurate measure of the actual running time.

Limitations of Empirical analysis approach

- **Three basic Limitations:**

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

Approach 2: Asymptotic analysis approach

- Method :
  - Develop a high level description of an algorithm.
  - From this high level description, count the number of primitive operations executed by the algorithm.

1. No need to implement algorithm on any system.

2. This approach considers all possible input instances

3. This approach also evaluate the efficiency of the algorithm in a way that it is independent of the platform we are using.

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 be 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.

Example: Largest Element in array

```
Algorithm Find_Largest(A, n)
1.      let max = x[0]
2.      for i = 1 to n-1
3.              if x[i] > max then,
4.                      max = x[i]
5.      Print max
```

How do we analyze algorithms using approach 2?

1. First identify primitive (basic) operations in the pseudo-code.
   primitive operation: It is a low level operation.
   Following operations are called as primitive operations:
   Data movement (assign)
   Ex. A=20, X=Y
   Control (branch, subroutine call, return)
   Ex. If-else, For, while, Find_Fact()
   Arithmetic and logical operations (e.g. addition, comparison)
   Ex. a=b/c, X>0

2. Let us assume that the cost of executing a primitive operation(statement) is some abstract cost *ci* *(a* constant amount of time is required to execute each line)

3. Count all the Ci's(C1,C2,C3 etc.) of the corresponding primitive operations that are executed by an algorithm.

4. Assume all Ci's as unity and find the time complexity function in term of n.

Example: Largest Element in array

```
Algorithm Find_Largest(A, n)
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

- Each of the operation takes a certain amount of time, depending upon the computer system you have.
- *C 1, C2, C 3, C4, C5, C 6 etc. just represent the amount of time taken for these* operations and they can be in any units.
- But, It is often difficult to get precise measurements of ci's
- The running time of an algorithm differs with respect to the nature of the input instance.
- The running time of an algorithm almost always expressed as a functions of the input size.

Note: **The instance is a set or a sequence of numbers that user will enter.**

Eg. 33,44,67,70,93 is one instance 978,45,2,66,67 is another instance.

Relook on complexity of above algorithms

- Largest Element in array

Descending Order –

Time Complexity or growth rate function f(n) = 1* C1 + n*C2 + (n-1)* C3 + 1* C5

= C1 + n * C2 + n * C3 – C3 + C5

= 2n + 1

Ascending Order –

Time Complexity or growth rate function f(n) = 1* C1 + n*C2 + (n-1)* C3 + (n-1) * C4 + 1* C5

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

= 3n

Find the Growth rate function

```
a=5;
b=6 ;
c=10 ;
for i =1 to n
    for j=1 to n
        x = i * I
        y = j * j
        z = i * j
for k=1 to n
        w = a*k + 45
        v = b*b
d = 33
```

f(n)= 1+1+1+(n+1)+n*(n+1)+n*n+n*n+n*n+(n+1)+n+n+1

$=4n^2 +5n+ 6$

Time Complexity

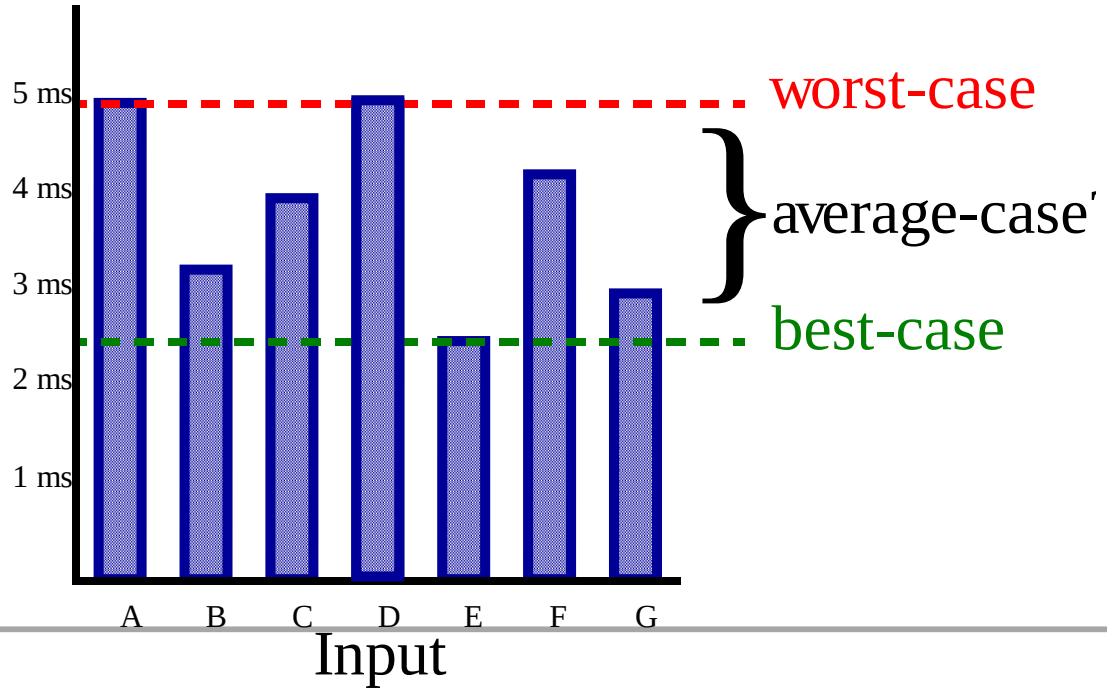
➢Time complexity :Time requirement of an algorithm, generally, a function in terms of size of input.

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. **WHY ????**
  - The computational/storage power on different machines vary

In general there are three cases for complexity analysis of an algorithm –

1. Best case
2. Average case
3. Worst case



Asymptotic Notations

- Asymptotic notation is a way of expressing the cost of an algorithm.
- Goal of Asymptotic notation is to simplify Analysis by getting rid of unneeded information

Big –O

Definition:

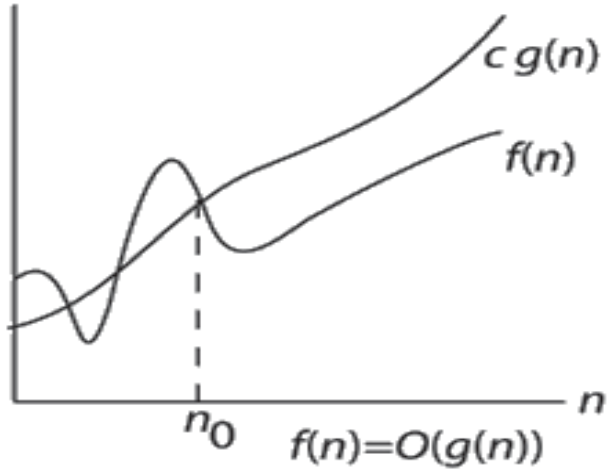
for a given function f(n), we say that

f(n)= O(g(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, **it tells you how long your algorithm is going to take in the worst case.**

- 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)



Asymptotic Notations (Cont…)

Big – Ω

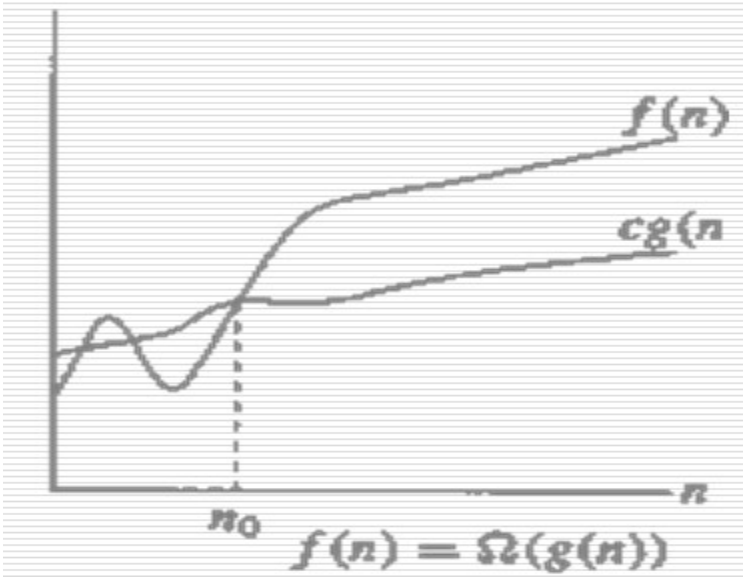
Definition:

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

f(n)= Ω (g(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
- 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)



Asymptotic Notations (Cont…)

Big – θ

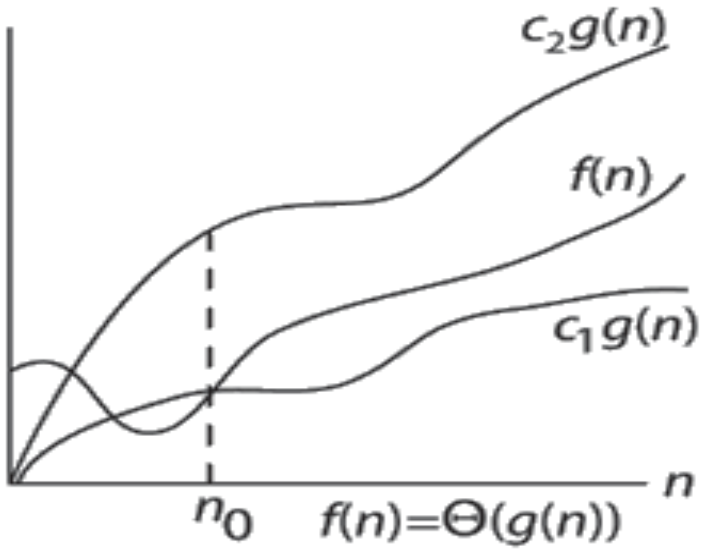
Definition:

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

f(n)=θ(g(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 –O Notation-Example 1

Given f(n) = 5n and g(n) = n then show that f (n)= O(g(n)).

Solution:

Just show that there exist two positive constants c and n0

Such that 5n<=cn for all n>=n0

Clearly for c=5 above condition is true for all n>=n0

Hence it is proved that f(n) = O*(g(n))

Note : We could choose a larger C such as 6, because the definition states that f(n) must be less than or equal to C * g(n), but we usually try and find the smallest value.

Big –O Notation-Example 2

Show that f(n) = 4n + 5= O(n)

Solution:To show that f(n) = 4n + 5 = O(n), we need to produce a constant C such that:

f(n) <= C * n for all n>=n0.

If we try C = 4, this doesn't work because 4n + 5 is not less than 4n. We need

C to be at least 9 to cover all n.

If n0 = 1, C has to be 9, but C can be smaller for greater values of n0 (if n0 = 100, C can be

5). Since the chosen C must work for all n>=n0, we must use 9:

4n + 5 <= 4n + 5n = 9n

Since we have produced a constant C that works for all n, we can conclude:

$$f(n) = O(n)$$

Big –O Notation-Example 3

If $f(n) = n^2$: prove that $f(n) \neq O(n)$

- **To do this, we must show that there cannot exist a constant C that satisfies the big-Oh definition.**

We will prove this by contradiction.

Suppose there is a constant C that works; then, by the definition of big-Oh: $n^2 \leq C * n$ for all value of n.

- **Suppose n is any positive real number greater than C, then: $n * n > C * n$, or $n^2 > C * n$.**

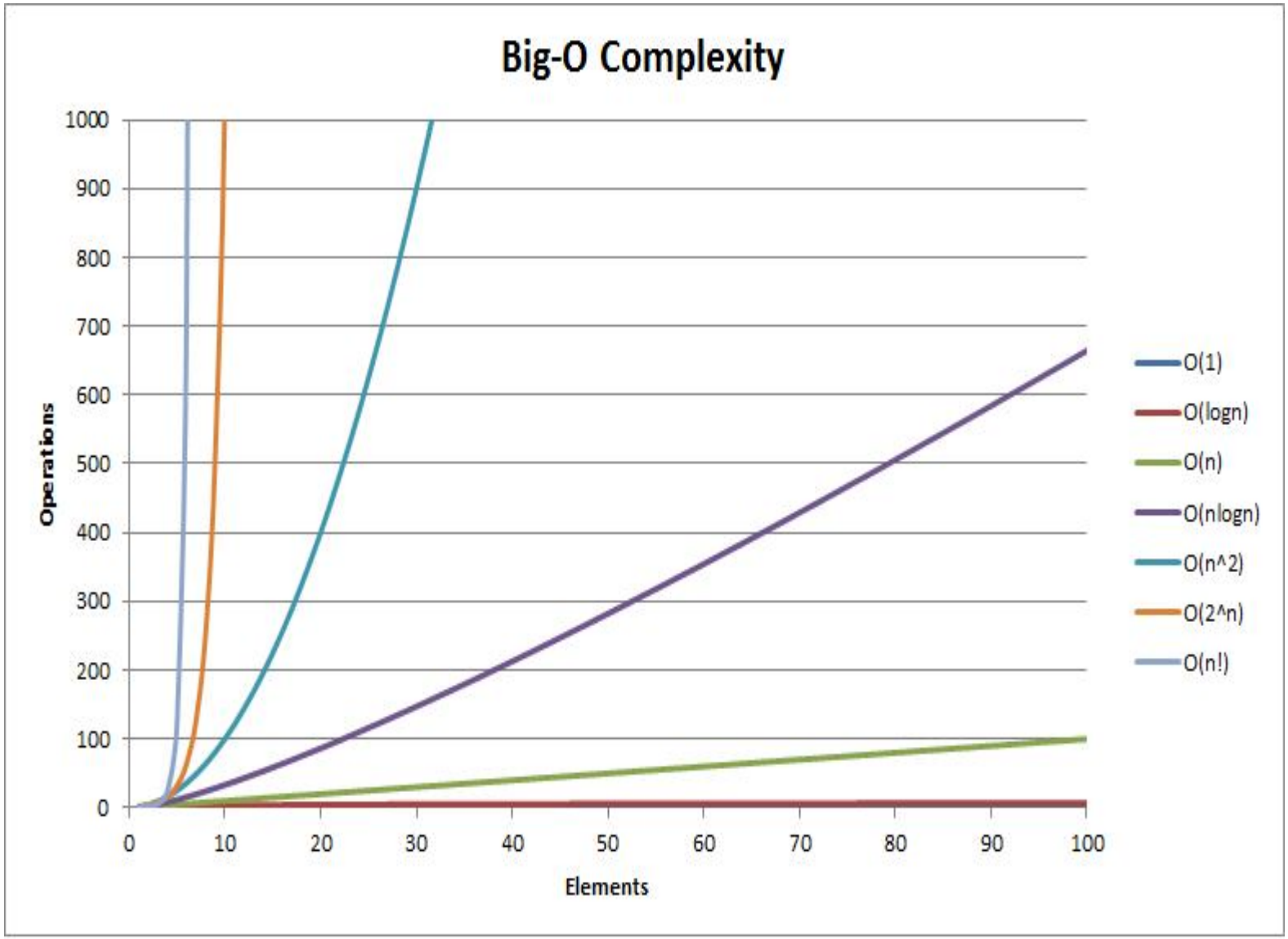
So there exists a real number n such that $n^2 > C * n$.

This contradicts the supposition, so the supposition is false

There is no C that can work for all n:

$f(n) \neq O(n)$ when $f(n) = n^2$

Big –O Notation



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 Example

>> Eg :

1. $3n + 2 = O(n)$

$3n + 2 \leq 4n$ for all $n \geq 2$.

So C=4 and n0=2

2. $3n + 3 = O(n)$

$3n + 3 \leq 4n$ for all $n \geq 3$.

So C=4 and n0=3

3. $100n + 6 = O(n)$

$100n + 6 \leq 101n$ for all $n \geq 6$.

So C=101 and n0=6

Worst Case Analysis (Usually Done)

- In the worst case analysis, we calculate upper bound on running time of an algorithm.
- We must know the case that causes maximum number of operations to be executed.
- For Linear Search, the worst case happens when the element to be searched (K in the our code) is not present in the array. When K is not present, the search() functions compares it with all the elements of arr[] one by one. Therefore, the worst case time complexity of linear search would be .

O(n)

Big –Ω Notation Example

1. $3n + 2 = \Omega(n)$

$3n + 2 \geq 3n$ for all $n \geq 1$.

So $c=3$ and $n0=1$

2. $3n + 3 = \Omega(n)$

$3n + 3 \geq 3n$ for all $n \geq 1$.

So $c=3$ and $n0=1$

3. $100n + 6 = \Omega(n)$

$100n + 6 \geq 100n$ for all $n \geq 1$.

So $c=100$ and $n0=1$

Best Case Analysis (Bogus)

- In the best case analysis, we calculate lower bound on running time of an algorithm.
- We must know the case that causes minimum number of operations to be executed.
- In the linear search problem, the best case occurs when K is present at the first location.
- The number of operations in the best case is constant (not dependent on n). So time complexity in the best case would be O(1)

Big –θ Notation Example

1. $3n + 2 = \Theta(n)$

$3n + 2 \geq 3n$ for all $n \geq 2$.

$3n + 2 \leq 4n$ for all $n \geq 2$.

So, $C1 = 3$, $C2 = 4$ & $n0 = 2$.

Average Case Analysis (Sometimes done)

- In average case analysis, we take all possible inputs and calculate computing time for all of the inputs.
- Sum all the calculated values and divide the sum by total number of inputs.

Problem

find such c and n_0 That proves **$n^2 + 2n + 1 = O(n^2)$.**

Solution:

We must find such c and n_0 that $n^2 + 2n + 1 \leq c*n^2$.

Let $n_0=1$, then for $n \geq 1$

$1 + 2n + n^2 \leq n + 2n + n^2 \leq n^2 + 2n^2 + n^2 = 4n^2$

Therefore, c = 4.

Conclusion

- We have already seen that an algorithm can be represented in the form of an expression.
- That means we represent the algorithm with multiple expressions: one for case where it is taking the less time(best case) and other for case where it is taking the more time(worst case).

Worst case

o Defines the input for which the algorithm takes huge time.
o Input is the one for which the algorithm runs the slower.

Best case

o Defines the input for which the algorithm takes lowest time.
o Input is the one for which the algorithm runs the fastest.

Average case

o Provides a prediction about the running time of the algorithm
o Assumes that the input is random

Conclusion

- Most of the times, we do worst case analysis to analyze algorithms.
- In the worst case analysis, we guarantee an upper bound on the running time of an algorithm which is **good information**.
- The average case analysis is not easy to do in most of the practical cases and it is rarely done.
- The Best Case analysis is bogus. Guaranteeing a lower bound on an algorithm doesn't provide any information as in the worst case, an algorithm may take years to run.

Note: We use theta(θ) notation if upper bound (O) and lower bound (Ω) are same.

Some Facts About Asymptotic Notations

No Uniqueness: There is no unique set of values for n0 and c in proving the asymptotic bounds. if we can show at least one pair of c and n_o such that **0<=f(n)<=cg(n)** for all n>=n_o

Then we can say f(n)= O(g(n)).

But it is always better to give smallest rate of growth g(n) which is greater than or equal to given algorithms rate of growth f(n) (tight upper bound).

Ex. **Prove that** 3n^3 +20n^2 + 5 is O(n^3)

Solution:

3n^3 + 20n^2 + 5 <= cn^3

⇒**20n^2 + 5 <= cn^3 - 3n^3**

⇒ **20n^2 + 5 <= n^3(c - 3)**

⇒ **20n^2/n^3 + 5/n^3 <= n^3(c - 3)/n^3**

⇒ **20/n + 5/n^3 <= c - 3**

⇒ **c >= 20/n + 5/n^3 + 3**

For n=1; c>=28

⇒ **Now we can take any c >=28 if we take** n_o =1

⇒**Hence proved 3n^3 +20n^2 + 5 is O(n^3)**

Note: It's almost always possible to find a lower c by raising n_0 e.g n_0 = 21 and c = 4 is also proves it.

Basic asymptotic efficiency classes

1	Constant
loglogn	double logarithmic
logn	Logarithmic
Log^2n	log-squared
N	Linear
nlogn	Log linear
n^2	Quadratic
n^3	Cubic
2^n	Exponential
n!	Factorial

Big-Oh Operations

Summation Rule:

$O(f1(n) + f2(n)) = O(\max(f1(n), f2(n)))$

Example:

1. **$O(2^n+n^3) = O(2^n)$**
2. **$O(n^4+n\log n)=O(n^4)$**

Product Rule

Suppose $T1(n) = O(f1(n))$ and $T2(n) = O(f2(n))$. Then, we can conclude that $T1(n) * T2(n) = O(f1(n) * f2(n))$.

Example:

1. **$O(n) * O(n^2) * O(n^3) = O(n^6)$**
2. **$O(n^2)*O(\log n)=O(n^2\log n)$**

Note: For any two functions *$f(n)$ and $g(n)$, we have $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.*

Big-Oh Rules

1. If $f(n)$ is a polynomial of degree d, then $f(n)$ is $O(n^d)$, i.e.,
    1. Drop lower-order terms
    2. Drop constant factors

    Eg. $f(n)=n^6+3n^2+77$

    Will be $O(n^6)$
2. Use the smallest possible class of functions

    Say "$2n$ is $O(n)$" instead of "$2n$ is $O(n^2)$"
3. Use the simplest expression of the class(discard coefficient)

    Say "$3n+5$ is $O(n)$" instead of "$3n+5$ is $O(3n)$"
4. Drop lower-order efficiency classes.

    Example: $f(n)=5000n^6+3n^2+77+34n!+n\log n$

    $f(n)=O(n!)$

True or False

N^2	=	$O(N^2)$	**true**
$2N$	=	$O(N^2)$	**true**
N	=	$O(N^2)$	**true**
N^2	=	$O(N)$	**false**
$2N$	=	$O(N)$	**true**
N	=	$O(N)$	**true**

True or False

$N^2 = \Omega(N^2)$ **true**

$2N = \Omega(N^2)$ **false**

$N = \Omega(N^2)$ **false**

$N^2 = \Omega(N)$ **true**

$2N = \Omega(N)$ **true**

$N = \Omega(N)$ **true**

True or False

$N^2 = \Theta(N^2)$ **true**

$2N = \Theta(N^2)$ **false**

$N = \Theta(N^2)$ **false**

$N^2 = \Theta(N)$ **false**

$2N = \Theta(N)$ **true**

$N = \Theta(N)$ **true**

Big –O Notation Illustrations

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

Problem

- Which of the following is not O(n^2)?
  (A) (15^10) * n + 12099
  (B) n^1.98
  (C) n^3 / (sqrt(n))
  (D) (2^20) * n
- **Ans:**
- **(C)**
- **The order of growth of option c is n^2.5 which is higher than n^2**

Some tips to find out time complexity of algorithms

1) O(1): Time complexity of a function (or set of statements) is considered as O(1) if it doesn't contain loop, recursion and call to any other non-constant time function.

// set of non-recursive and non-loop statements
For example swap() function has O(1) time complexity.
A loop or recursion that runs a constant number of times is also considered as O(1).
For example the following loop is O(1).

```
// Here c is a constant
for (int i = 1; i <= c; i++)
{
// some O(1) expressions
}
```

Like

```
for (int i = 1; i <= 1000; i++)
{
Print i;
}
```

Example: Adding an element to the front of a linked list

Some tips to find out time complexity of algorithms

2) O(n): Time Complexity of a loop is considered as O(n) if the loop variables is incremented / decremented by a constant amount.
For example following functions have O(n) time complexity.

```
// Here c is a positive integer constant
for (int i = 1; i <= n; i += c)
{
// some O(1) expressions
}

for (int i = n; i > 0; i -= c)
{
// some O(1) expressions
}
```

Like

```
for (int i = 1; i <= n; i += 1)
{
Print i;
}
```

Example: linear search

Some tips to find out time complexity of algorithms

3) $O(n^c)$: Time complexity of nested loops is equal to the number of times the innermost statement is executed.
For example the following sample loops have $O(n^2)$ time complexity

```
for (int i = 1; i <=n; i += c)
{
        for (int j = 1; j <=n; j += c)
            {
                // some O(1) expressions
             }
}

For (i = n; i > 0; i -= c) {
        for (int j = i+1; j <=n; j += c)
                { // some O(1) expressions
                }
```

For example **Selection sort** and **Insertion Sort** have $O(n^2)$ time complexity.

Some tips to find out time complexity of algorithms

4) O(Logn) Time Complexity of a loop is considered as O(Logn) if the loop variables is divided / multiplied by a constant amount.

```
for (int i = 1; i <=n; i *= c)
        {
                // some O(1) expressions
        }
for (int i = n; i > 0; i /= c)
        {
                // some O(1) expressions
        }
```

Like

```
for (int i = 1; i <=n; i *= 2)
        {
                // some O(1) expressions
        }
```

For example Binary Search has O(Logn) time complexity.

Some tips to find out time complexity of algorithms

5) O(LogLogn) Time Complexity of a loop is considered as O(LogLogn) if the loop variables is reduced / increased exponentially by a constant amount.

```
// Here c is a constant greater than 1
for (int i = 2; i <=n; i = pow(i, c))
        {
        // some O(1) expressions
        }
//Here fun is sqrt or cuberoot or any other constant root
for (int i = n; i > 0; i = fun(i))
        {
        // some O(1) expressions
        }
```

Like

```
for (int i = 2; i <=n; i = i^2)
        {
        // some O(1) expressions
        }
```

How to combine time complexities of consecutive loops?

When there are consecutive loops, we calculate time complexity as sum of time complexities of individual loops.

```
for (int i = 1; i <=m; i += c)
        {
        // some O(1) expressions
        }
for (int i = 1; i <=n; i += c)
        { // some O(1) expressions
        }
```

Time complexity of above code is O(m) + O(n) which is O(m+n)
If m == n, the time complexity becomes O(2n) which is O(n).

Find the growth rate function and Time Complexity

```
 x = x +1; //constant time
for (i=1; i<=n; i++)
{
m = m + 2; //constant time
}
for (i=1; i<=n; i++) //outer loop executed n times
{
        for (j=1; j<=n; j++) //inner loop executed n times
        {
                k = k+1; //constant time
        }
}
```

Homework

Q1. Rank the following function by their order of growth(increasing order).

$(n+1)!$, 1 , $n^{1/\lg n}$, $(3/2)^n$, $\log(\log n)$, $\text{sqrt}(2)^{\lg n}$, $(\lg n)^{\lg n}$, $\text{sqrt}(\lg n)$

Q2. Analyse the following fragments of code. Formulate exact expressions for the running time of each code fragment.

(a)

```
int sum = 0;
int num = 35;
for (int i=1; i<=n; i++) {
        for (int j=1; j<=n; j++) {
                num += j*3;
                sum += num;
        }
}
```

(b)

```
int count = 0;
for (int i = n; i > 0; i /= 2)
        for (int j = 0; j < i; j++)
                count += 1;
```

(b)

```
for ( i=0 ; i < n ; i++ )
        total = total + i;
for ( j=0 ; j < n ; j++ )
        total = total + j;
```

(d)

```
int count = 0;
for (int i = 0; i < n; i++)
        for (int j = i; j > 0; j--)
                count = count + 1;
```

Problem

- Consider the following functions: f(n) = 2^n , g(n) = n! , h(n) = n^logn
  Which of the following statements about the asymptotic behavior of f(n), g(n), and h(n) is true?
  (A) f(n) = O(g(n)); g(n) = O(h(n))
  (B) f(n) = **Ω**(g(n)); g(n) = O(h(n))
  (C) g(n) = O(f(n)); h(n) = O(f(n))
  (D) h(n) = O(f(n)); g(n) = **Ω** (f(n))
- Ans: (D)

Problem

- Let w(n) and A(n) denote respectively, the worst case and average case running time of an algorithm executed on an input of size n. which of the following is ALWAYS TRUE? (GATE CS 2012)
- (A) $A(n) = \Omega(W(n))$
  (B) $A(n) = \Theta(W(n))$
  (C) $A(n) = O(W(n))$
  (D) $A(n) = o(W(n))$
- Ans: (C)

Software development life cycle (SDLC)

1. Requirement gathering and analysis: Business requirements are gathered in this phase. This phase is the main focus of the project managers and stake holders. Meetings with managers, stake holders and users are held in order to determine the requirements like; Who is going to use the system? How will they use the system? What data should be input into the system? What data should be output by the system?
2. Requirement Specification: Requirement Specification document is created which serves the purpose of guideline for the next phase of the model.
3. Design: System Design helps in specifying hardware and system requirements and also helps in defining overall system architecture. The system design specifications serve as input for the next phase of the model.

Software development life cycle (SDLC)

4. Coding: On receiving system design documents, the work is divided in modules/units and actual coding is started. Since, in this phase the code is produced so it is the main focus for the developer. This is the longest phase of the software development life cycle.
5. Testing: After the code is developed it is tested against the requirements to make sure that the product is actually solving the needs addressed and gathered during the requirements phase. During this phase unit testing, integration testing, system testing, acceptance testing are done.
6. Deployment: After successful testing the product is delivered / deployed to the customer for their use.
7. Maintenance: Once when the customers starts using the developed system then the actual problems comes up and needs to be solved from time to time. This process where the care is taken for the developed product is known as maintenance.