Design and Analysis of Algorithms

UNIT – I:

Introduction: Algorithm, Pseudo code for expressing algorithms, Performance Analysis: Space Complexity, Time Complexity, Asymptotic Notation – Big Notation, Omega Notation, Theta Notation and Little oh notation, Probabilistic Analysis,

Introduction to Algorithms:

"Simply, An Algorithm is a well-defined step-by-step procedure for performing some task in a finite amount of time"

The word 'Algorithm' comes from the name of Persian author, ***"Abu Ja'far Mohammed ibn Musa al Khowarizmi"***. He wrote lot of textbooks on mathematics. The latest addition of Webster's dictionary defines its meaning as "any special method of solving a certain kind of problem".

Informally, there are many definitions of algorithms. Some of them are,

- An algorithm is any well-defined computational procedure that takes some values or set of values as Input and produces some value or set of values as Output. That is an algorithm is a sequence of computational steps that transforms the input into output.
- An algorithm is a sequence of operations performed on data they have to be organized in data structure.
- An algorithm is an abstraction of a program to be executed on physical machine.
- An algorithm is a set of rules for carrying out calculations either by hand or on a machine.

In general, An algorithm is a finite set of well-defined instructions that, if followed, accomplish a particular task. In addition to this definition, all algorithms must follow five basic criteria or rules. They are

1. Input 2. Output 3. Definiteness 4. Finiteness 5. Effectiveness

1. INPUT: Zero or more quantities are externally supplied to the algorithm and the logic of that algorithm processes those quantities, are called inputs.

2. OUTPUT: At least one quantity is produced after applying the logic on input quantities. These results are called outputs. Every algorithm should have at least one output.

3. DEFINITENESS: Every instruction is clear and unambiguous. The steps or instructions, that are used to describe algorithm or logic of algorithm should clearly specifies, what it going to do with out any confusion.

In order to achieve this criterion, algorithms are written in any programming language. The languages are designed so that each legitimate statement has unique meaning. ***"A Program is the expression of an algorithm in programming language"***

4. FINITENESS: If we trace out the instructions of an algorithm, thus for all cases, the algorithm terminates after a finite amount of time. That means, any problem can be solved by algorithmically in a finite amount of time. To do so, the algorithm is composed of steps, each of which may carry one or more number of operations. By simply, the algorithm must terminate after finite number of operations.

5. EFFECTIVENESS: Every instruction must be very basic, so that it can be carried out, in principle, by a person using only pen and paper. That is, each step of algorithm is simple, easy to understand, and also perform the processing in easy ways by anyone. The operations of algorithm are not only definiteness, but also feasible.

An algorithm that satisfies the criteria of definiteness and effectiveness is also called "computational procedures". One important example related to computational procedure is operating system of digital computer. This procedure is designed to control the execution of jobs, in such a way that, when no jobs are available, it does not terminate but continuous in a waiting state until new job is entered.

DATA STRUCTURE*: "A data structure is a systematic way of organizing and accessing the data"*. The algorithms and data structures are central to computing. Almost all of the practical problems are solved by algorithms. So the applications of algorithms are ubiquitous.

LIFE CYCLE OF AN ALGORITHM: The life cycle of an algorithm has four stages. They are 1. How to devise an algorithm, 2. How to validate an algorithm, 3. How to analyze algorithm and 4. How to test a program.

How to devise an algorithm: After getting the knowledge on problem specifications and user requirements, we are ready to write algorithms for that problem solving. The creation of an algorithm is an art and there is no fully automated mechanism to write all algorithms. But there are some design strategies or methods; those have proven to be useful to write algorithms in easy way. We are selecting anyone of design technique to write algorithm, based on problem specification and user requirements. Some of design strategies are Greedy Method, Dynamic Programming, Brute – force Method, etc. some design techniques are also useful especially other than Computer Science such as Operation Research and Electrical Engineering.

How to validate an algorithm: Once the algorithm is devised, it is necessary to show that, it gives correct outputs for all legal inputs. We termed this process as "Algorithm Validation". The algorithm validation is done in two phases.

In the first phase, the algorithm is not expressed in any of the programming language. Here the purpose of validation is to assure that, this algorithm will work

correctly, independent of the programming language constraints. So we can use any precise method to show correctness of algorithm. The second phase will come after the validation method in first phase.

In the second phase, the algorithm is written in any one of the programming language. So this phase is also called "program proving" or "program verification". In this phase the proof of correctness is done based on the solution of two forms. One form is usually as a program, which is annotated by a set of assertions about the input and output variables of the program. These assertions are often expressed in predicate calculus. The second form is called, specification, and is also expressed in predicate calculus. The proof of consists of showing that these two forms are equivalent in that for every given legal input, they describe the same output. A complete proof of program correctness requires that, each statement of the program language be precisely defined and all basic operations be proved correct.

How to analyze algorithm: As the algorithm is executed, it use computer's Central Processing Unit (CPU) and its memory (both auxiliary and primary) to hold the program and data. So the analysis of algorithm or performance analysis refers to determining how much computing time it takes and how much storage is needed for that algorithm to execute on ideal system. This analysis is used to make quantitative judgment about algorithm over another. This is also used to compare the algorithms.

How to test a program: As we discussed in second phase, correctness is most important attribute of a program. Providing mathematical proof of correctness for even small algorithms is quite difficult task. So we are gone for one more process, called "program testing". In this process, the program is executed on target computer using input data set, called "test data", and compares the programs behavior with expected behavior. We can increase our confidence on correctness of program, by considering all legal possible input data. The subset of input data space that is actually used to testing is called "test set".

The programming testing consists of two phases. They are Debugging and Profiling or Performance Measurement.

Debugging is the process of executing programs on sample data sets to determine whether faulty result occurs and, if so correct them. If we are unable to verify the correctness of program, then develop the same program with different programmers and select one, which result is matched with expected result. **"*The proof of correctness is more valuable than thousand tests, since it is guarantee that the program will work on all possible legal inputs*".**

Profiling or performance measurement is the process of executing a correct program on data sets and measures the time and space, it takes to compute results.

Algorithm Specifications:

There is a distinct difference between algorithm and program. We know that, the algorithm is usually described in English language to ensure definiteness condition. There are some other ways to describe the algorithms. Some of them are Flow charts, Pseudo code etc. Flow chart is one that is used to represent the algorithm and algorithm flow control in graphical representation. This method is not efficient and makes more complex for large algorithms.

"***Pseudo – code is mixture of natural language and high – level programming constructs that describes the main ideas behind a generic implementation of a data structure or algorithm***". The Pseudo – code is easy to read and understand. Pseudo code is also used to implement structured design element concepts. The pseudo code should not resemble any particular programming language. The pseudo code is more compact than an equivalent actual software code fragment would be. The pseudo code is also uses high level language abstractions. So it follows some conventions to describe the algorithm. Some of the pseudo code conventions are

1. Comments are begin with // and continue until the end of the line.
2. Compound statement is represented by a block. Each block is indicated by matching braces {and}.
3. Every statement is delimited by semicolon (;).
4. An identifier begins with a latter. The data types and scope of the variables are depends upon context. The data types are not explicitly defined in pseudo code. Compound data types are formed with records.

   Example:

```
node = record
       {
           datatype_1    data_1;
           datatype_2    data_2;
             …             …
          datatype_n    data_n;
            Node    * link;
       }
```

   link is a pointer to the record type node. Individual data items of a record can be accessed with → and period. Example P pointes to record of the type node, P → data_1 stands for the values of the first field of the record.
5. Assigning a value to a variable done using assignment operator.

   <Variable>:= <expression> or <variable> ← <expression>
6. It uses Boolean values (TRUE and FLASE), Logical Operators (AND , OR and NOT) and Relational Operators(<,≤,≥,>and==).
7. Elements of arrays can be accessed using subscripts braces and subscripts / indices
8. It also uses for, while and repeat – until looping statements.

   The while loop form:

```
While <condition> do
   {
     <Statements - 1>;
```

:
:
<Statements - n>;
}

The for loop form:

```
for variable := value_1 to value_2 step STEP do
    {
        <Statement - 1>;
        <Statement - 2>
            :
            :
        <Statement - n>;
    }
```

The repeat – until loop form:

```
repeat
    <Statement - 1>;
        :
        :
    <Statement - n>;
until <condition>;
```

9. It also uses conditional statements like IF-THEN block, IF-THEN-ELSE block, CASE etc.

IF – THEN block form:

```
IF <condition> THEN
    <Statements>;
```

IF – THEN – ELSE block form:

```
IF <condition> THEN
    <Statements>;
ELSE
    <Statements>;
```

CASE statement form:

```
CASE
  {
    :< condition - 1>:  <statement - 1>;
    :< condition - 2>:  <statement - 2>;
        :                   :
        :                   :
    :< condition - n>: <statement - n>;
    : Else:     <statement - n>
  }
```

10. READ and WRITE phases are used to specify the input and output of algorithm.
11. Method Declaration: There is only one type of method of Procedure. That is Algorithm. An algorithm consists of heading and body. The form of algorithm is

    Algorithm name (<parameters list>)

Where name is the name of procedure and parameters list is the list of procedure parameters.'

12. It also uses break statement and return statement. The break statement is used for force exit from loops. The return statement with value is return from the specified method also exit from function it self.

EXAMPLE – 1

```
Algorithm Max (a,n)
  // a is an array of size – n
 {
    larger ← a[0];
      for i ← 1 to n do
             If (a[i] > larger) then
           larger ← a[i];
     Return larger
  }
```

EXAMPLE – 2

```
Algorithm Selection_sort (a,n)
  // assume array a[1:n]
 {
    for i ← 1 to n do
     {
          j ← I;
          for k ← i+1 to n do
                if(a[k]<a[j])
                    j ← k;
         t← a[i];
         a[i] ← a[j];
         a[j] ← t;
  }
```

RECURSIVE ALGORITHMS: Another useful technique to solve problems is Recursion. ***"An algorithm is said to be recursive if the same algorithm is invoked in the body of the algorithm"***. That is, in this technique, we define a procedure – p that allows to make a call to itself as a subroutine, provided those calls to procedure – p are for solving sub-problem of smaller size. The sub routine calls to 'p' in smaller instances are called "recursive calls". A recursive procedure should always define a "base case / base component", which is small enough that the algorithm can solve it directly with out using recursion.

There are two recursive methods available. They are

1. Direct Recursive
2. Indirect Recursive

"An algorithm that, directly calls it self is called direct recursive"

"An algorithm – A said to be indirect recursive, if it calls another algorithm which in turn calls Algorithm - A"

Example – 1 Towers of Hanoi

```
Algorithm TowersofHanoi(n,x,y,z)
// n is the number of disks on tower. x,y and z are towers.
// move the top  n – disks from tower – x  to tower – y
 {
    if(n≥1) then
       {
          TowersofHanoi(n-1, x, z, y);
           Write("Move to disk from tower ",x,"to top of tower",y);
           TowersofHanoi(n-1, z, y, x);
         }
  }
```

Example 2 Finding the largest element in given array

```
Algorithm RMax(a,n)
{
     if(n==1)then
          return a[0];
     else
          return max{RMax(a,n-1),a(n-1)};
 }
```

Performance Analysis:

In order to classify some good algorithms and data structures, we must have a precise way to analyzing them. There are mainly two factors for judging algorithms that have a more direct relationship to performance. They are

1. Running time of algorithm and data structure operations
2. Space utilization for each operation of an algorithm

Because time is a precious resource, analyzing algorithm based on running time is good measurement. By simple, The performance of a program means the amount of computer memory and time needed to run a program. There are two approaches to determine performance of a program. They are

1. Analytical Method
2. Experimental Method

The performance evaluation can be loosely divided into major phases. They are

1. Priori Estimate | Apriori Analysis | Perform Analysis
2. Posteriori Testing | Empirical Method | Performance Measurement

The priori estimates is used to describe the task of estimating the time and space utilization of an algorithm during execution time. This analysis is done at algorithmic level. That means this analysis is done before implementing that program on system. This priori analysis uses analytical approach to calculate time and space requirement of algorithm. Under this analytical method, we usually use RAM (Random Access Machine) Model.

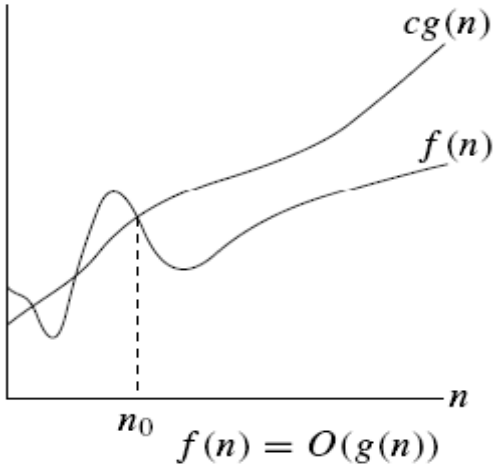
Asymptotic Notations: The program step count in step count is inexact, because the step count is one for either instruction x=y or x = y+(x*y). But the running time for latter instruction is high. So this step count is not very useful comparative purpose of algorithms.

But we can use step count to accurately predict the growth in run time for larger instance size [that is, the asymptote as n approaches infinity] and to predict the relative performance of two programs when the instance size becomes large. There is a notation that allows us to characterize the main factors affecting an algorithm's running time with out going to all the details of exactly how many primitive operations are performed for each constant time set of instructions [but not their coefficients] in the complexity functions. ***By simple Asymptotic Notations are used to formulized that "An algorithm has running time or storage requirement that are NEVER MORE THAN, ALWAYS GREATER THAN, or EXACTLY some amount".***

There are mainly five asymptotic notations available.
1. Big – Oh (O) Notation [NEVER MORE THAN]
2. Big – Omega (Ω) Notation [ALWAYS GRATER THAN]
3. Theta (Θ) Notation [EXACTLY]

Big – Oh Notation:

" f(n) and g(n) are two positive functions, the function f(n) = O(g(n)) [read as " f of n is Big-Oh of g of n" OR " f of n is the order of Big – Oh of g of n"] if and only if, there exist positive constants c and n_o such that f(n)≤ c g(n) for all n, where n ≥n_o".



The Big-Oh notation allows us to say that a function of n [f(n)] is less that or equal to another function, up to a constant factor [constant - c] and in the asymptotic sense as n grows towards infinity [$n \geq n_o$].
That is, the growth rate f(n) always less than or equal to the growth rate of g(n) for large n-values.

The Big – Oh notation is used widely to characterize running times and space bounds in terms of some parameters – n, which varies from problem to problem, but is usually defined as an intuitive notion of size of the program. The statement f(n)=O(g(n)) states only that g(n) is an **upper bound** on the value of f(n) for all n, where n ≥ no. it does not say about how good this bound is.

RULES OF BIG – OH NOTATIONS:

1. The constants are ignored. That is O(100n) is write as O(n). Here 100(constant) is ignored. That is units are not important.
2. Lower order terms are also ignored. That is $O(2n^2+26n+39)$ is written as $O(n^2)$. It mainly concentrates on comparative relative growth only.

EXAMPLES:

1. Find Big – Oh representation of **f(n)=3n+2**

Sol. According to the definition of Big – Oh, $f(n) \le c\ g(n)$ for $n \ge n_o$. so

$3n + 2 \le c.\ n$ here g(n) is n.

$2 \le c.n - 3n$

$2 \le n(c-3)$

$2/(c-3) \le n$ here we take n as n_o

$\rightarrow n_o \ge 2/(c-3)$

According to the definition of Big – Oh, c and n_o are positive constants.

So we should choose c-value that leads n_o have to positive.

So choose c has 4, that implies n_o has 2.

So $f(n) = 3n+2 = O(4n)$. here we can neglect c-value in Big – Oh representation.

So $f(n) = 3n+2 = O(n)$, where $n \ge 2$.

If you want to test the condition [$f(n) \le c.\ g(n)$], use n = 2,3,.. [Using Induction method]

for n= 2 $f(n=2)=3.(2)+2=8$. and $c.g(n) = 4.(2)=8$. That is $f(n)=c.g(n)$.

for n=3 $f(n=3)=11$ and $c.g(n)=12$. That is $f(n)<c.g(n)$

We can prove this by varying n-values.

2. Find Big – Oh representation of **$f(n)=10n^2+4n+2$**

Sol. According to the definition of Big – Oh, $f(n) \le c\ g(n)$ for $n \ge n_o$. so

$10n^2+4n + 2 \le 10n^2+4n+2n^2$

[Here we are using basic mathematics. Our intension is to find the c.g(n) and that is always bigger than f(n). For finding g(n), we are just altering the f(n). The newly function **(i.e c.g(n))** should grater than or equal to f(n). To do this, we are just adding highest power to the constant term in the equation. This ensures the condition. The alteration depends upon the f(n) nature.]

$10n^2+4n+2 \le 12n^2+4n$, where $n>0$.

By observing, above equation $c.g(n)=12n^2+4n$. From this function we have to get c & n_o

$12n^2+4n = c.n^2$

$4n = n^2(c-12)$

$n_o = 4/(c-12)$ since c and n_o are positive constants. C=13 and $n_o \ge 4$

that is $f(n)=10n^2+4n+2=O(13n^2)$, $n \ge 4$

$\rightarrow f(n)=10n^2+4n+2=O(n^2)$

3. Find Big – Oh representation of **$f(n)=5n^2-6n$**.

Sol. According to the definition of Big – Oh, $f(n) \leq c.g(n)$.

So $5n^2-6n \leq 5n^2$, where $n > 0$

[By observing f(n), we come to know that one term (6n) is subtracted from another term. So we are neglected the subtracted terms in f(n), that gives the new function c.g(n) and this function is always greater than the f(n). So we are simply getting the c.g(n) by such modifications on f(n)]

$\rightarrow 5n^2-6n \leq 5n^2$

By observing above equation, $c.g(n)= 5n^2$ where $c= 5$ and $n_o \geq 1$

That is $f(n) = 3n^2-6n = O(5n^2)$

$f(n) = 3n^2 = O(n^2)$

4. Find a upper bound on **$f(n) = n^8 + 7n^7 -10n^5 - 2n^4 + 3n^2 -17.$**

Sol. We can determine the upper bound of any function using Big – Oh notations. So if we represent the order of f(n) using Big –Oh notation, then it gives the upper bound.

By observing above function, we can easily determine the upper bound of f(n) as $O(n^8)$

Proof: We will prove that $f(n) = O(n^8)$. First, we will prove an upper bound for f(n).

It is clear that when $n > 0$,

$n^8 + 7n^7 – 10n^5 – 2n^4 + 3n^2 - 17 \leq n^8 + 7n^7 + 3n^2$

[We can upper bound any function by removing the lower order terms with negative coefficients, as long as $n > 0$.]

Next, it is not hard to see that when $x \geq 1$,

$n^8 + 7n^7 + 3n^2 \leq n^8 + 7n^8 + 3n^8 = 11n^8$

[**We can upper bound any function by replacing lower order terms with positive coefficients by the dominating term with the same coefficients. Here, we must make sure that the dominating term is larger than the given term for all values of x larger than some threshold x_0, and we must make note of the threshold value x_0.**]

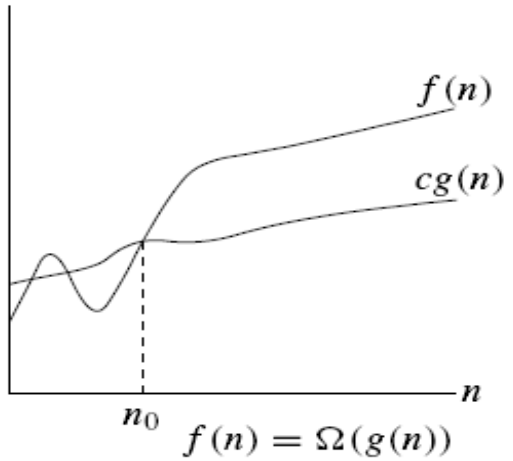
Thus, we have

$f(n) = n^8 + 7n^7 -10n^5 - 2n^4 + 3n^2 -17 \cdot \leq 11n^8$ for all $n \geq 1$;

Thus we have proved that $f(n) = O(n^8)$ where $n \geq 1$.

Big – Omega Notation:

" let f(n) and g(n) are two positive functions, the function f(n) = Ω(g(n)) [read as " f of n is Big-Omega of g of n" OR " f of n is the order of Big – Omega of g of n"] if and only if, there exist positive constants c and n_o such that f(n)≥ c g(n) for all n, where n ≥n_o".



The Big-Omega notation allows us to say that a function of n [f(n)] is greater that or equal to another function, up to a constant factor [constant - c] and in the asymptotic sense as n grows towards infinity [$n \geq n_o$].
That is, the growth rate f(n) always greater than or equal to the growth rate of g(n) for large n-values.

The Big – Omega notation is used widely to characterize running times and space bounds in terms of some parameters – n, which varies from problem to problem, but is usually defined as an intuitive notion of size of the program. The statement f(n)=O(g(n)) states only that g(n) is an **Lower bound** on the value of f(n) for all n, where n ≥ no. it does not say about how good this bound is.

EXAMPLES

1. Find the Big – Omega representation of **f(n) = 3n+2.**

sol. According to the definition of Big – Omega, $f(n) \geq c\ g(n)$ for $n \geq n_o$. so

$3n + 2 \geq c.\ n$ here g(n) is n.

By observing above equation carefully, we can conclude as follows

$3n+2 \geq 3n$

[This is done, by ignoring the low order positive constant on left hand function. Then it gives view of Big – Omega Definition.]

That is, we can lower bound any function by removing the lower order terms with positive coefficients, as long as n > 0.

→ c = 3

According to Big – Omega definition, n_0 is positive constant. So take $n_0 \geq 1$.

$3n + 2 \geq 3n$, where $n \geq n_0$ and $n_0 \geq 1$.

So the above in $f(n) = c.g(n)$ where $n \geq n_0$ and $n_0 \geq 1$.

So we can represent f(n) with g(n) order. That is $f(n) = \Omega(n)$.

We can also prove this by varying n-values using induction method talked in Big – Oh notation.

2. Find the lower bound of $f(n) = 45n^4 – 5n^2 + 4n – 456$

Sol: According to the definition, $f(n) \geq c.\ g(n)$ where $n \geq n_0$ and n_0 is positive constant.

$f(n) = 45n^4 – 5n^2 + 4n – 456$

$> 45n^4 – 5n^2 – 456$ [we thrown the positive terms]

$> 45n^4 (1 – (5n^2 - 456) / 45n^4)$

$> 45n^4 (1 – (1/9)\ n^2 – (456/45)\ n^4)$ for $n > 456$

$> 45\ n^4 / 2$

$= \Omega\ (n^4)$

3. Find the lower bound of $f(n) = 33n^3+4n^2$

Sol: According to the definition, $f(n) \geq c.\ g(n)$ where $n \geq n_0$ and n_0 is positive constant.

$f(n) = 33n^3 + 4n^2$

$> 33n^3$ where $n \geq 1$

[We ignored positive $4n^2$ term in order to get definition criteria]

It is like criteria in Definition, so here $c = 33$ and $g(n) = n^3$

$= \Omega\ (n^3)$

4. Find a Lower bound on **$f(n) = n^8 + 7n^7 -10n^5 - 2n^4 + 3n^2 -17.$**

Sol:

Now, we will get a lower bound for f(x). It is not hard to see that when $x \geq 0$,

$$x^8 + 7x^7 - 10x^5 - 2x^4 + 3x^2 - 17 \geq x^8 - 10x^5 - 2x^4 - 17:$$

- We can lower bound any function by removing the lower order terms with positive coefficients, as long as $x > 0$.

Next, we can see that when $x \geq 1$,

$$x^8 - 10x^5 - 2x^4 - 17 \geq x^8 - 10x^7 - 2x^7 – 17x^7 = x^8-29x^7$$

We can lower bound any function by replacing lower order terms with negative coefficients by a sub-dominating term with the same coefficients. (By sub-dominating, I mean one which dominates all but the dominating term.) Here, we must make sure that the sub-dominating term is larger than the given term for all values of x larger than some threshold x0, and we must make note of the threshold value x0. Making a wise choice for which sub-dominating term to use is crucial in finishing the proof.

Next, we need to find a value $c > 0$ such that $x^8 - 29x^7 \geq cx^8$, Doing a little arithmetic, we see that this is equivalent to $(1 - c)x^8 \geq 29x^7$. When $x \geq 1$, we can divide by x^7 and obtain $(1 - c)x \geq 29$. Solving for c, we obtain

$$c \leq 1 - \frac{29}{x}.$$

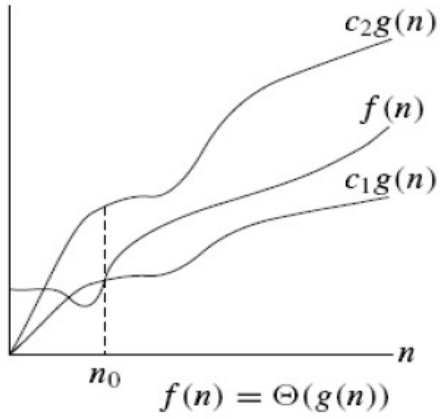
If $x \geq 58$, then $c = (1/2)$ suffices. We have just shown that if $x \geq 58$, then

$$f(x) = x^8 + 7x^7 - 10x^5 - 2x^4 + 3x^2 - 17 \geq (1/2)x^8:$$

Thus, $f(x) = \Omega(x^8)$.

Theta Notation:

"let f(n) and g(n) are two positive functions, the function f(n) = Θ(g(n)) [read as " f of n is Big-Theta of g of n" OR " f of n is the order of Big – Theta of g of n"] if and only if, there exist positive constants c1, c2 and n_o such that c1g(n)≤f(n)≤ c2 g(n) for all n, where n ≥n_o".



Intuitively Θ(g(n)) = the set of functions with the same order of growth as g(n)

Here, g(n) is an asymptotically tight bound for f(n)..

EXAMPLE

1. Find a tight bound on **$f(n) = n^8 + 7n^7 - 10n^5 - 2n^4 + 3n^2 - 17$.**

Sol: we will prove that $f(n) = \Theta(n^8)$.

First we will prove upper bound for f(n). We can determine the upper bound of any function using Big – Oh notations. So if we represent the order of f(n) using Big –Oh notation, then it gives the upper bound.

By observing above function, we can easily determine the upper bound of f(n) as $O(n^8)$

Proof: We will prove that $f(n) = O(n^8)$. First, we will prove an upper bound for f(n).

It is clear that when n > 0,

$$n^8 + 7n^7 - 10n^5 - 2n^4 + 3n^2 - 17 \le n^8 + 7n^7 + 3n^2$$

[We can upper bound any function by removing the lower order terms with negative coefficients, as long as n > 0.]

Next, it is not hard to see that when x ≥ 1,

$$n^8 + 7n^7 + 3n^2 \le n^8 + 7n^8 + 3n^8 = 11n^8$$

[**We can upper bound any function by replacing lower order terms with positive coefficients by the dominating term with the same coefficients. Here, we must make sure that the dominating term is larger than the given term for all values of x larger than some threshold x_0, and we must make note of the threshold value x_0.**]

Thus, we have

$f(n) = n^8 + 7n^7 - 10n^5 - 2n^4 + 3n^2 - 17 \cdot \le 11n^8$ for all n ≥ 1;

Thus we have proved that $f(n) = O(n^8)$ where n ≥ 1.

Now, we will get a lower bound for f(x). It is not hard to see that when x ≥ 0,

$x^8 + 7x^7 - 10x^5 - 2x^4 + 3x^2 - 17 \geq x^8 - 10x^5 - 2x^4 - 17$:

- We can lower bound any function by removing the lower order terms with positive coefficients, as long as $x > 0$.

Next, we can see that when $x \geq 1$,

$x^8 - 10x^5 - 2x^4 - 17 \geq x^8 - 10x^7 - 2x^7 - 17x^7 = x^8 - 29x^7$

We can lower bound any function by replacing lower order terms with negative coefficients by a sub-dominating term with the same coefficients. (By sub-dominating, I mean one which dominates all but the dominating term.) Here, we must make sure that the sub-dominating term is larger than the given term for all values of x larger than some threshold x0, and we must make note of the threshold value x0. Making a wise choice for which sub-dominating term to use is crucial in finishing the proof.

Next, we need to find a value $c > 0$ such that $x^8 - 29x^7 \geq cx^8$, Doing a little arithmetic, we see that this is equivalent to $(1 - c)x^8 \geq 29x^7$. When $x \geq 1$, we can divide by x^7 and obtain $(1 - c)x \geq 29$. Solving for c, we obtain

$$c \leq 1 - \frac{29}{x}.$$

If $x \geq 58$, then $c = (1/2)$ suffices. We have just shown that if $x \geq 58$, then

$f(x) = x^8 + 7x^7 - 10x^5 - 2x^4 + 3x^2 - 17 \geq (1/2)x^8$:

Thus, $f(x) = \Omega(x8)$.

Since we have shown that $f(n) = \Omega(n^8)$ and $f(n) = O(n^8)$, so $f(n) = \Theta(n^8)$

Theorems - 1:

Prove that if $f(x) = O(g(x))$, and $g(x) = O(f(x))$, then $f(x) = \Theta(g(x))$.

Proof:

If $f(x) = O(g(x))$, then there are positive constants c_2 and n_0' such that

$$0 \leq f(n) \leq c_2\, g(n) \text{ for all } n \geq n_0'$$

Similarly, if $g(x) = O(f(x))$, then there are positive constants c_1' and n_0'' such that

$$0 \leq g(n) \leq c_1'\, f(n) \text{ for all } n \geq n_0''.$$

We can divide this by c_1' to obtain

$$0 \leq \frac{1}{c_1'} g(n) \leq f(n) \text{ for all } n \geq n_0''.$$

Setting $c_1 = 1/c_1'$ and $n_0 = \max(n_0', n_0'')$, we have

$$0 \leq c_1 g(n) \leq f(n) \leq c_2\, g(n) \text{ for all } n \geq n_0.$$

Thus, $f(x) = \Theta(g(x))$.

Theorem - 2:

Let $f(x) = O(g(x))$ and $g(x) = O(h(x))$. Show that $f(x) = O(h(x))$.

Proof:

If $f(x) = O(g(x))$, then there are positive constants c_1 and n_0' such that

$$0 \leq f(n) \leq c_1\, g(n) \text{ for all } n \geq n_0',$$

and if $g(x) = O(h(x))$, then there are positive constants c_2 and n_0'' such that

$$0 \leq g(n) \leq c_2\, h(n) \text{ for all } n \geq n_0''.$$

Set $n_0 = \max(n_0', n_0'')$ and $c_3 = c_1\, c_2$. Then

$$0 \leq f(n) \leq c_1\, g(n) \leq c_1\, c_2\, h(n) = c_3\, h(n) \text{ for all } n \geq n_0.$$

Thus $f(x) = O(h(x))$.

Properties

Transitivity:

$$\begin{array}{lllll}
f(n) = \Theta(g(n)) & \text{and} & g(n) = \Theta(h(n)) & \text{imply} & f(n) = \Theta(h(n)) \\
f(n) = O(g(n)) & \text{and} & g(n) = O(h(n)) & \text{imply} & f(n) = O(h(n)) \\
f(n) = \Omega(g(n)) & \text{and} & g(n) = \Omega(h(n)) & \text{imply} & f(n) = \Omega(h(n)) \\
f(n) = o(g(n)) & \text{and} & g(n) = o(h(n)) & \text{imply} & f(n) = o(h(n)) \\
f(n) = \omega(g(n)) & \text{and} & g(n) = \omega(h(n)) & \text{imply} & f(n) = \omega(h(n))
\end{array}$$

Reflexivity:

$$\begin{aligned}
f(n) &= \Theta(f(n)), \\
f(n) &= O(f(n)), \\
f(n) &= \Omega(f(n)).
\end{aligned}$$

Symmetry:

$$f(n) = \Theta(g(n)) \text{ if and only if } g(n) = \Theta(f(n)).$$

Transpose symmetry:

$$\begin{array}{lll}
f(n) = O(g(n)) & \text{if and only if} & g(n) = \Omega(f(n)), \\
f(n) = o(g(n)) & \text{if and only if} & g(n) = \omega(f(n)).
\end{array}$$

Little – Oh Notation:

"let f(n) and g(n) are two positive functions, the function f(n) = o(g(n)) [read as " f of n is Little-Oh of g of n" OR " f of n is the order of Little – Oh of g of n"] if and only if, for all positive constants c and n_o such that f(n) < c g(n) for all n, where n ≥n_o". It specifies the upper bound, but not an asymptotically upper bound.

Little – Omega Notation:

"let f(n) and g(n) are two positive functions, the function f(n) = ω(g(n)) [read as " f of n is Little-Omega of g of n" OR " f of n is the order of Little – Omega of g of n"] if and only if, for all positive constants c and n_o such that f(n) > c g(n) for all n, where n ≥n_o". It specifies the Lower bound, but not an asymptotically lower bound.

EXAMPLE

1. Is $7n + 8 \in o(n^2)$?

Again, to claim this we need to be able to argue that for any c, we can find an n_0 that makes $7n+8 < c\ n2$.

Let's try examples again to make our point, keeping in mind that we need to show that we can find an n0 for any c.

If c = 100, the inequality is clearly true. If c = (1/100) , we'll have to use a little more imagination, but we'll be able to find an n_0. (Try $n_0 = 1000$.)

At this point, it should be clear that, regardless of the c we choose, we'll be able to get $c\ n^2$ to dominate 7n+8 eventually (that is, we can find a large enough n_0 to make the definition hold). Thus, $7n+8 \in o(n^2)$.

The following diagram specifies all asymptotic notations.