

Chapter One

Algorithm Analysis Concepts

Algorithm Analysis

- ≡ **Algorithm is a step by step procedure to solve a problem. E.g. industrial activities, Student Registration, etc, all need an algorithm to follow.**
- ≡ **More than one algorithm is possible for the same task.**

Properties of algorithms

- ≡ **Finiteness:** any algorithm should have finite number of steps to be followed.
- ≡ **Language Independence:** it must not depend on any one programming language.
- ≡ **Sequential:** the first step (start step) and last step (halt step) must be clearly noted.
- ≡ **Correctness:** it must compute correct answer all possible legal inputs.
- ≡ **Completeness:** it must solve the problem completely.
- ≡ **Input/output:** zero or more input, one or more output.
- ≡ **And so on.**

Algorithm Analysis Concepts

- ≡ Complexity analysis is concerned with determining the efficiency of algorithms.
- ≡ What is Efficiency depends on?
 - Execution speed (most important)
 - Amount of memory used
- ≡ Efficiency differences may not be noticeable for small data, but become very important for large amounts of data

How do we estimate the complexity of algorithms?

- ≡ 1. Algorithm analysis - Find $f(n)$ - helps to determine the complexity of an algorithm
- ≡ 2. Order of Magnitude - $g(n)$ belongs to $O(f(n))$ - helps to determine the category of the complexity to which it belongs.

Analysis Rule:

- 1. Assume an arbitrary time unit
- 2. Execution of one of the following operations takes time unit 1
 - Assignment statement
 - Single I/O statement;
 - Single Boolean statement.
 - Single arithmetic.
 - Function return.
- 3. selection statement
 - Time for condition evaluation + the maximum time of its clauses
- 4. Loop statement
 - $\Sigma(\text{no of iteration of Body}) + 1 + n + 1 + n$ (initialization time + checking + update)
- 5. For function call
 - $1 + \text{time}(\text{parameters}) + \text{body time}$

Estimation of Complexity of Algorithm

Examples:

```
1. int count(){  
    int k=0;  
    cout<< "Enter an integer";  
    cin>>n;  
    for (i=0;i<n;i++)  
        k=k+2;  
    return 0;}
```

Time Units to Compute

- 1 for the assignment statement: `int k=0`
- 1 for the output statement.
- 1 for the input statement.

In the for loop:

- 1 assignment, $n+1$ tests, and n increments.
- n loops of 2 units for an assignment, and an addition.
- 1 for the return statement.

$$T(n) = 1 + 1 + 1 + (1 + n + 1 + n) + 2n + 1 = 4n + 6 = O(n)$$

...

- Calculate $T(n)$ for the following

- 1. $k=0;$

$\text{Cout} << \text{"enter an integer"};$

$\text{Cin} >> n;$

For ($i=0; i < n; i++$)

$K++$

- $T(n) = 1 + 1 + 1 + (1 + n + 1 + n + n)$
 $= 5 + 3n$

- **2. i=0;**
While (i<n)
{
 x++;
 i++;
}
J=1;
While(j<=10)
{
 x++;
 j++
}
- **$T(n) = 1 + n + 1 + n + n + 1 + 11 + 10 + 10$**
 $= 3n + 34$

...

3. **for(i=1;i<=n;i++)**
 for(j=1;j<=n; j++)
 k++;
T(n)=1+n+1+n+n(1+n+1+n+n)=3n²+4n+2

4. *Sum=0;*

if(test==1)

{

for (i=1;i<=n;i++)

sum=sum+i

}

else

{

cout<<sum;

}

• **$T(n) = 1 + 1 + \text{Max}(1 + n + 1 + n + 2n, 1) = 4n + 4$**

Exercise

- Calculate $T(n)$ for the following codes

A. *sum*=0;
for(*i*=1;*i*≤*n*;*i*++)
for(*j*=1;*j*≤*m*;*j*++)
Sum++;

B. *sum*=0;
for(*i*=1;*i*≤*n*;*i*++)
for(*j*=1;*j*≤*i*;*j*++)
sum++;

The Best, Average and Worst case

- In order to determine the running time of an algorithm it is possible to define three functions.
- ≡ Average Case (T_{avg}): The amount of time the algorithm takes on an "average" set of inputs.
- ≡ Worst Case (T_{worst}): The amount of time the algorithm takes on the worst possible set of inputs.
- ≡ Best Case (T_{best}): The amount of time the algorithm takes on the smallest possible set of inputs.

...cont

 We are interested in the worst-case time, since it provides a bound for all input - this is called the “Big-Oh” estimate.

Therefore need to distinguish a number of different cases:

- Best case (minimum number of operations)
- Worst case (maximum number of operations)
- Average case (average number of operations)

☰ Types of Asymptotic Notations

- 1. Big-O Notation
- 2. Big - Omega (Ω)
- 3. Big -Theta (Θ)

1. Big-O Notation

- Definition : The function $f(n)$ is $O(g(n))$ if there exist constants c and N such that $f(n) \leq c.g(n)$ for all $n \geq N$.
- As n increases, $f(n)$ grows no faster than $g(n)$ or in the long run (for large n) f grows at most as fast as g
- It computes the tight upper bound of $f(n)$
- Describes the worst case analysis.

O-notation

- Assume there is function $f(n)$ for rate of growth
- Where n is the number of inputs and the function growth is continues as inputs get larger.
- What is the *upper limit* or *worst case* of this function?
- We have to find another function which is greater than $f(n)$ and lets call this function $cg(n)$
- After some limits $n \geq n_0$, $c > 0$ and $n_0 \geq 1$

O-notation

For function $g(n)$, we define $O(g(n))$, big-O of n , as the set

$O(g(n)) = \{f(n) :$
 $\exists$ positive constants c
and n_0 , such that $\forall n \geq n_0$,
we have $0 \leq f(n) \leq$
 $cg(n)\}$

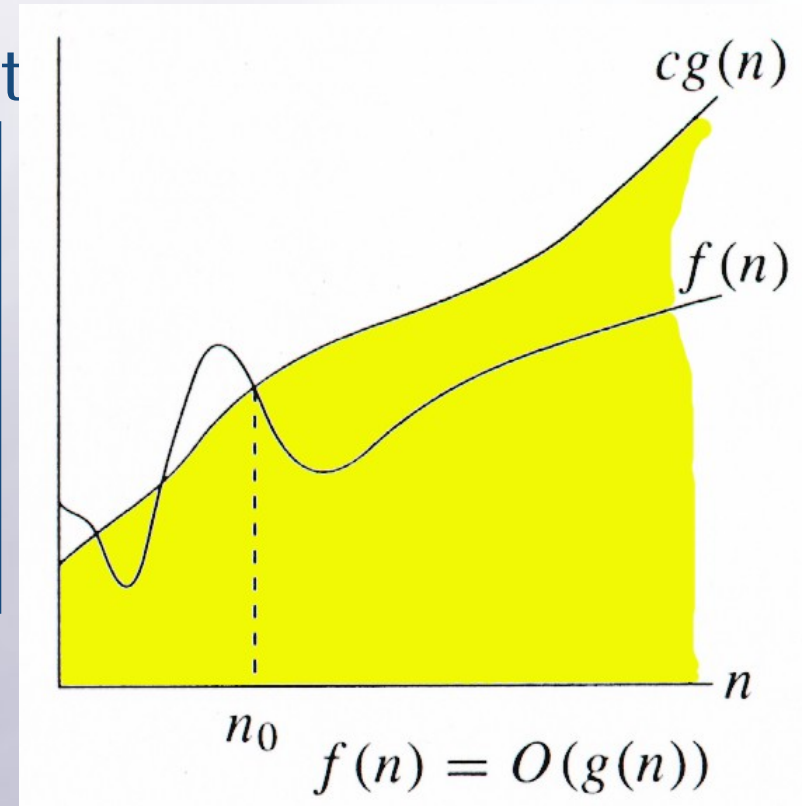
Intuitively: Set of all

functions whose *rate of growth* is the same as or lower than that of $g(n)$.

is an **asymptotic upper bound** for $f(n)$.

$f(n) = \Theta(g(n)) \Rightarrow f(n) = O(g(n)).$

$\Theta(g(n)) \subset O(g(n)).$



Examples

$O(g(n)) = \{f(n) : \exists \text{ positive constants } c \text{ and } n_0, \text{ such that } \forall n \geq n_0, \text{ we have } 0 \leq f(n) \leq cg(n) \}$

$f(n)=3n+2$ and $g(n)=n$

Finding Asymptotic Complexity: Examples

≡ Rules to find Big-O from a given $T(n)$

- Take highest order
- Ignore the coefficients

Finding Big O of given Algorithm

≡ 1. for (i=1;i<=n;i++)

● Cout<<i;

● $T(n) = 1+n+1+n+n$

$$= 3n+2$$

● $T(n) = O(n)$

≡ 2. For (i=1;i<=n;i++)

- For(j=1;j<=n;j++)

- Cout<<i;

- $T(n) = 1 + n + 1 + n + n(1 + n + 1 + n + n)$

$$O = 3n^2 + 4n + 2$$

- $T(n) = O(n^2)$

≡ 3. for (i=1; i<=n; i++)

● Cout<<i;

≡ T(n) = 1+n+1+n+n

● = 3n+2

≡ T(n) = O(n)

Exercise

Find Big O of the following algorithm

1. `for(i=1;i<=n;i++)`

- `Sum=sum+i;`

- `For (i=1;i<=n;i++)`

- `For(j=1;j<=m;j++)`

- `Sum++;`

Ω -notation

For function $g(n)$, we define $\Omega(g(n))$, big-Omega of n , as

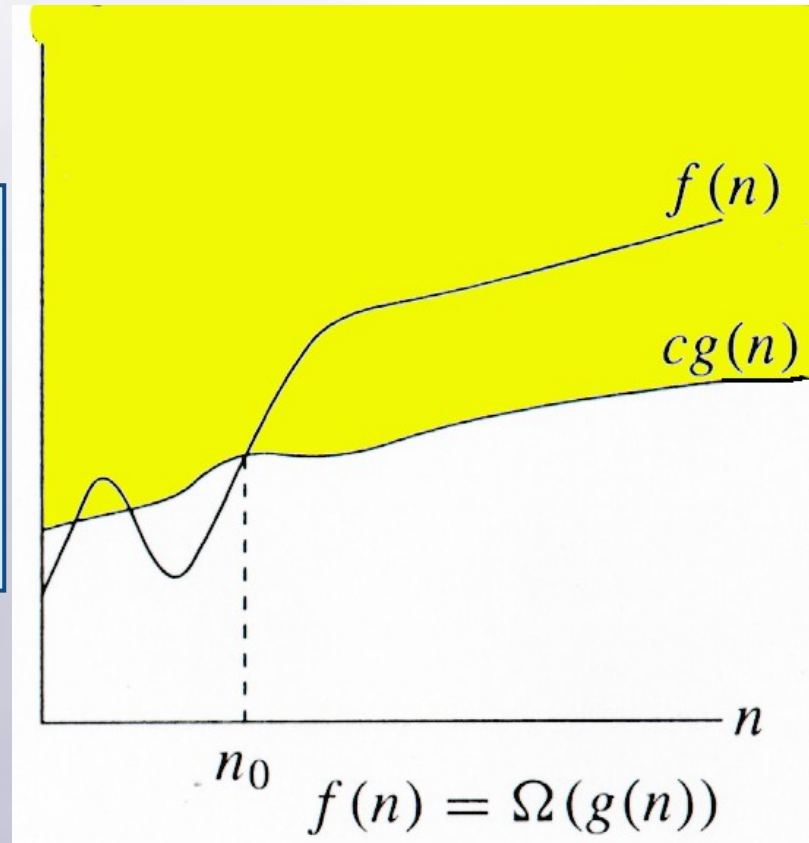
the set:
 $\Omega(g(n)) = \{f(n) :$
 $\exists$ positive constants c
and n_0 , such that $\forall n \geq n_0$,
we have $0 \leq cg(n) \leq f(n)\}$

Intuitively: Set of all functions whose *rate of growth* is the same as or higher than that of $g(n)$.

is an **asymptotic lower bound** for $f(n)$.

$$f(n) = \Theta(g(n)) \Rightarrow f(n) = \Omega(g(n)).$$

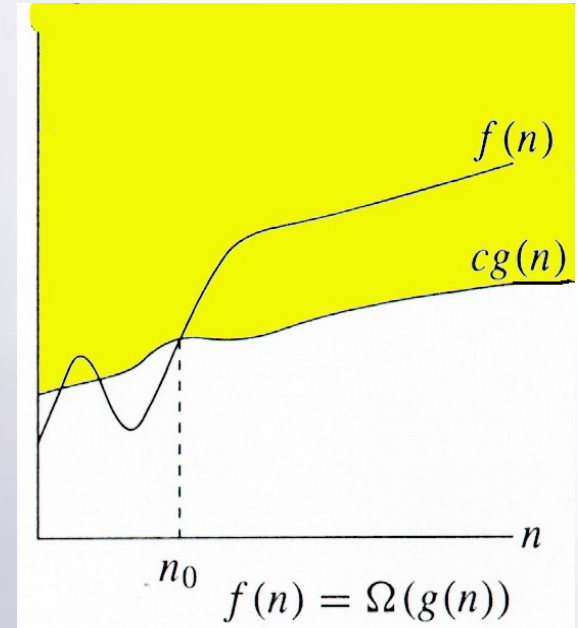
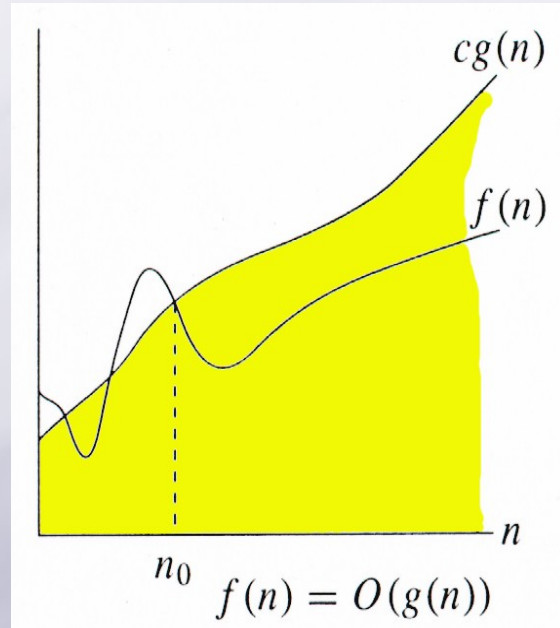
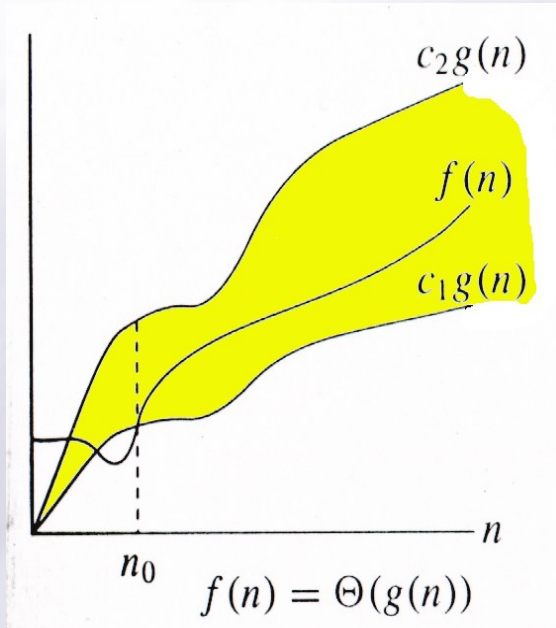
$$\Theta(g(n)) \subset \Omega(g(n)).$$



Example


$$\Omega(g(n)) = \{f(n) : \exists \text{ positive constants } c \text{ and } n_0, \text{ such that } \forall n \geq n_0, \text{ we have } 0 \leq cg(n) \leq f(n)\}$$

Relations Between Θ , O , Ω



Relations Between Θ , Ω , O

Theorem : For any two functions $g(n)$ and $f(n)$,

$$f(n) = \Theta(g(n)) \text{ iff}$$

$$f(n) = O(g(n)) \text{ and } f(n) = \Omega(g(n)).$$

≡ In practice, asymptotically tight bounds are obtained from asymptotic upper and lower bounds.

Running Times

≡ “Running time is $O(f(n))$ ” $\Rightarrow$ Worst case is $O(f(n))$

≡ $O(f(n))$ bound on the worst-case running time $\Rightarrow O(f(n))$ bound on the running time of every input.

≡ $\Theta(f(n))$ bound on the worst-case running time $\Rightarrow \Theta(f(n))$ bound on the running time of every input.

≡ “Running time is $\Omega(f(n))$ ” $\Rightarrow$ Best case is $\Omega(f(n))$

≡ Can still say “Worst-case running time is $\Omega(f(n))$ ”

- Means worst-case running time is given by some unspecified function $g(n) \in \Omega(f(n))$.

Example

- ≡ ***Insertion sort*** takes $\Theta(n^2)$ in the worst case, so sorting (as a *problem*) is $O(n^2)$. Why?
- ≡ Any sort algorithm must look at each item, so sorting is $\Omega(n)$.
- ≡ In fact, using (e.g.) merge sort, sorting is $\Theta(n \lg n)$ in the worst case.

Development of Notation

- ≡ Not concerned with small values of n
- ≡ Concerned with VERY LARGE values of n
- ≡ Asymptotic - refers to study of function f as n approaches infinity
- ≡ Example: $f(n) = n^2 + 4n + 20$
 - n^2 is the dominant term and the term $4n + 20$ becomes insignificant as n grows larger

Development of Notation

- ≡ Drop insignificant terms and constants
- ≡ Say function is of $O(n^2)$ called Big-O of n^2
- ≡ Common Big-O functions in algorithm analysis
 - $g(n) = 1$ (growth is constant)
 - $g(n) = \log_2 n$ (growth is logarithmic)
 - $g(n) = n$ (growth is linear)
 - $g(n) = n \log_2 n$ (growth is faster than linear)
 - $g(n) = n^2$ (growth is quadratic)
 - $g(n) = 2^n$ (growth is exponential)

<i>n</i>	<i>log2n</i>	<i>nlog2 n</i>	<i>n^2</i>	<i>2^n</i>
1	0	0	1	2
2	1	2	4	4
4	2	8	16	16
8	3	24	64	256
16	4	64	256	65536
32	5	160	1024	4294967296

Asymptotic Complexity

- ≡ Running time of an algorithm as a function of input size n **for large n .**
- ≡ Expressed using only the **highest-order term** in the expression for the exact running time.
 - Instead of exact running time, say $\Theta(n^2)$.
- ≡ Describes behavior of function in the limit.
- ≡ Written using ***Asymptotic Notation***.