GLOBAL EDGE

Intelligence Of Things™

Big O and Complexity

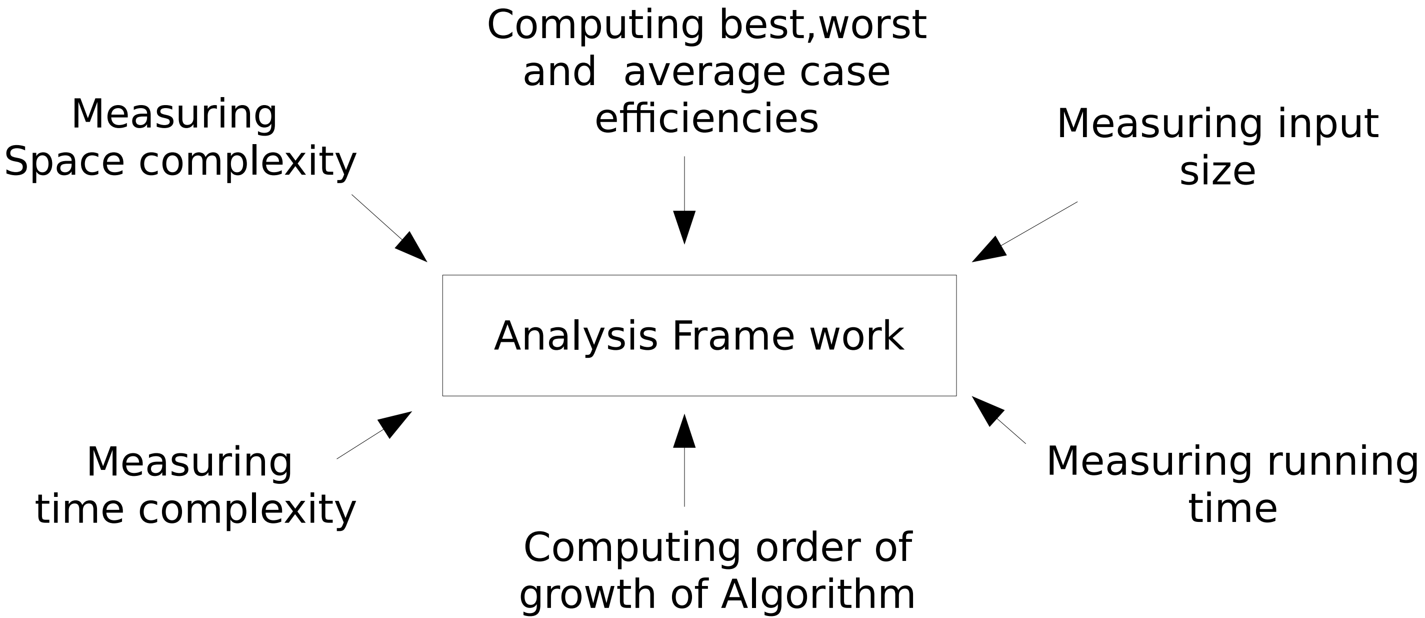
Dilipkumar S.

Copyright © 2017, Global Edge Software Ltd.

Content

- Asymptotic notation
- Order of Growth
- Calculation of time complexity
  - Non-recursive function
  - Recursive function
- Complexity for sorting and searching algorithms

Analysis framework



Order of Growth

Value	logn	n	nlogn	n^2	n^3	2^n	n!
1	0	1	0	1	1	2	1
2	1	2	2	4	8	4	2
4	2	4	8	16	64	16	24
8	3	8	24	64	512	256	40320
16	4	16	64	256	4096	65536	2.6*10^48
10	3.2	10	3.3*10	10^2	10^3	10^3	3.6*10^6
10^2	6.6	10^2	6.6*10^2	10^4	10^6	1.3*10^30	9.3*10^157
10^3	10	10^3	1.0*10^4	10^6	10^9	high	high

GLOBAL EDGE

Asymptotic Notation

What is Asymptotic Notation ?

- Asymptotic Notations of a function is the study of how the value of the function varies for large values of n, where n is the size of input.

- Using Asymptotic notation we could easily find the efficiency of the algorithm.

Types of Asymptotic notation

Three different type of asymptotic notation

- O (Big Oh)
- Ω (Big Omega)
- Θ (Big Theta)

O (Big Oh)

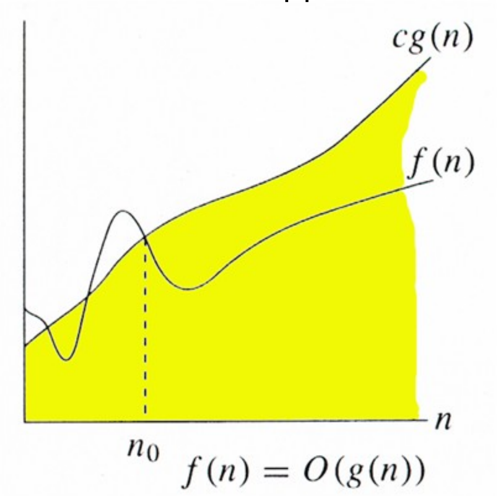
- Upper bound of algorithm's running time.
- Worst case running time for an algorithm
- Longest amount of time the algorithm takes to compute.

$$f(n) \leq c*g(n) \text{ for all } n \geq n0.$$

or

$$f(n) \in O(g(n))$$

Upper bound



Examples for Big- O

1. fn() = 100n + 5

   Let c * g(n) = 100n + n (replacing constant 5 with n)

   ie c * g(n) = 101n

   since f(n) <= c * g(n)

   100n + 5 <= 101n for all n >= 5

   so c = 101   g(n) = n   n0 = 5

   Hence $f(n) \leq c * g(n)$ for all $n \geq n0$

   $f(n) \in O(g(n))$

Cont....

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

   Ans: $3n + 2 <= 4n$

2. $f(n) = 10n^3 + 8$

   Ans: $10n^3 <= 11n^3$

3. $f(n) = 10n^2 + 4n + 2$

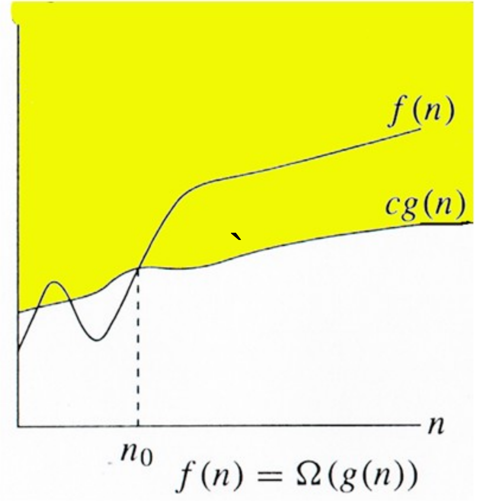
   Ans: $10n^2 + 4n + 2 <= 11n^2$

Ω (Big Omega)

- Lower bound of algorithm's running time.
- Best case running time for an algorithm.
- Gives minimum amount of time the algorithm takes to compute.

f(n) > = c * g(n) for all n >= n0

f(n) Ω (g(n))



Examples for Big- Ω

1. fn() = 100n + 5

Let c * g(n) = 100n (Discarding constant 5)

ie c * g(n) = 100n

since f(n) >= c * g(n)

100n + 5 >= 100n for all n >= 0

so c = 100   g(n) = n   n0 = 0

Hence f(n) >=   c * g(n) for all n ≥ n0

f(n) Є Ω(g(n))

Cont....

1. $f(n) = 50n + 6$

   Ans: $50n$

2. $f(n) = 17n^3 + 5$

   Ans: $17n^3$

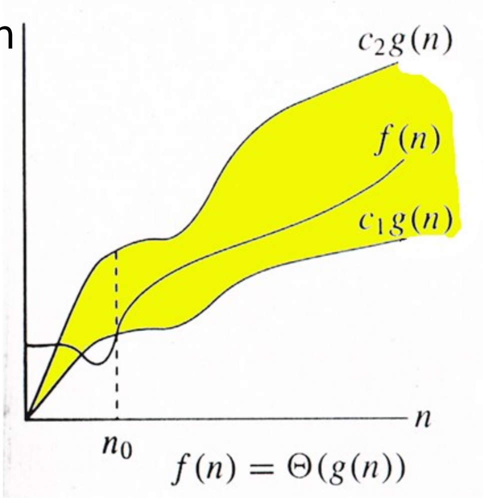
Θ (Big Theta)

- Gives average amount of time the algorithm takes to compute.

$$c1 * g(n) <= f(n) <= c2 * g(n)$$

$$\text{for all } n >= n0$$

f(n) Θ (g(n))



Examples for Big- Θ

1. fn() = 10n^3 + 5

   Let c2 * g(n) = 10n^3 + n (replacing constant 5 with n)

   ie c2 * g(n) = 11n^3

   Let c1 * g(n) = 10n^3

   since c1 * g(n) <= f(n) <= c2 * g(n)

   10n^3 <= 10n^3 +5 <= 11n^3

- so c1 = 10n^3, c2 = 11n^3, g(n) = n^3, n0 = 2

f(n) ∈ Θ(g(n))

General plan for Analyzing the Time Efficiency of Non-recursive Algorithm

1. Based on input determine the number of parameters to be considered.

2. Identify the algorithms **basic operation**.

3. Check the basic operation depends only on the size of the input.

4. Obtain the total number times to basic operation is executed.

5. An useful formula to compute is $$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Matrix multiplication

```
for i ← 0 to n - 1 do
    for j ← 0 to n - 1 do
        c[j] ← 0
        for k ← 0 to n - 1 do
            c[i,j] ← c[i,j] + A[i,k] * B[k,j]
```

Basic operation

$T(n) = \Theta(n^3)$

General plan for Analyzing the Time Efficiency of recursive Algorithm

1. Based on input determine the number of parameters to be considered.

2. Identify the algorithms ***basic operation***.

3. Check the basic operation depends only on the size of the input.

4. Set up a ***recurrence relation*** with an appropriate ***initial condition***, for number of times the basic operation is executed.

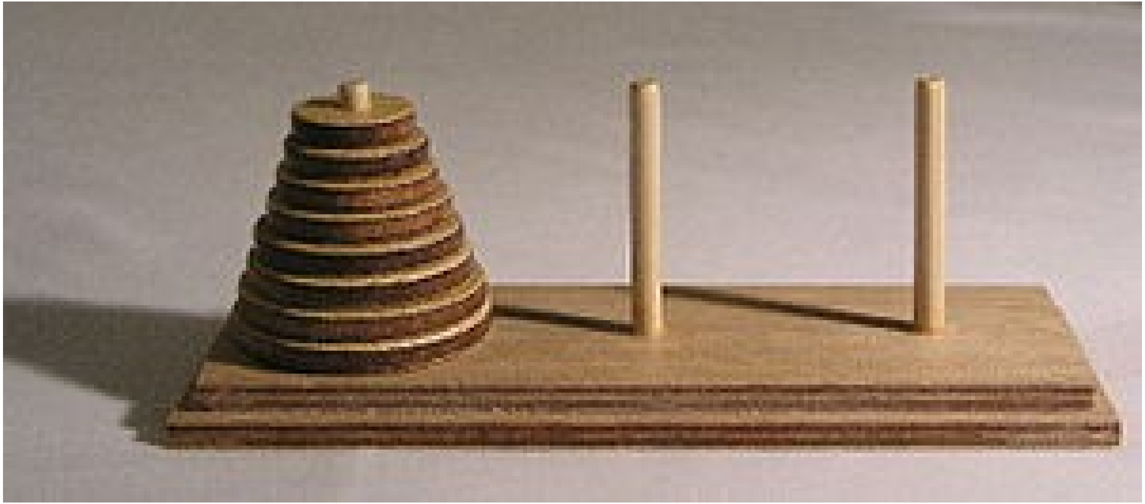
Example: 1

Recursive algorithm to find factorial of a number

```
Factorial(int n){
    if (n == 0)
            return 1;
    return (Factorial(n – 1) * n);
}
```

Example: 2

Tower of Hanoi



Algorithm

```
    Tower_honai(n, source, temp, dest)

Step 1:   If( n == 1)
          move(n'st, source, dest)
          return

Step 2:   Tower_honai(n - 1, source, dest, temp)

Step 3:   move(nth, source, dest)

Step 3:   Tower_honai(n - 1, temp, dest, source)
```

The Algorithm - details

```
BubbleSort(A[0..n-1])
        for i  <- 0  to n–2 do
            for j  <- 0 to n-2-i do
                if A[j+1] < A[j]
                swap A[j] and A[j+1]
```

The Algorithm - details

```
SelectionSort(A[0..n-1])

for i  <- 0  to n–2 do
      min <- i
     for j  <- i+1 to n-1 do
         if A[j] > A[min]  min <- j
         swap A[i] and A[min]
```

The Algorithm - details

```
InsertionSort(A[0..n-1])
for i  <- 0  to n–2 do {
    v <- a[i];
    j<- i – 1;
    while( j >= 0 and A[j] > v) do{
        A[j+1] <- A[j];
        j<- j – 1;
    }
    A[j+1] <- v;
}
```

Merge sort

```
mergesort (A , p , r )        - T(n)
     if( p > r)   return
    q = (p+r)/2
    mergesort(B , p , q)  – T(n/2)
    mergesort(C , q+1 , r)– T(n/2)
    merge(A, B, C)          – (n)
```

Efficiency

$$T(n) = T(n/2) + T(n/2) + (n)$$

$$T(n) = 2T(n/2) + (n)$$

$$T(n) = (n \text{ Log } (n))$$

Comparison between various sorting algorithms

Sorting Algorithm	Best	Average	worst
Bubble sort	Ω(n^2)	θ(n^2)	O(n^2)
Insertion sort	Ω(n)	θ(n^2)	O(n^2)
Selection sort	Ω(n^2)	θ(n^2)	O(n^2)
Heap sort	Ω(n*log(n))	θ(n*log(n))	O(n*log(n))
Merge sort	Ω(n*log(n))	θ(n*log(n))	O(n*log(n))
Quick sort	Ω(n*log(n))	θ(n*log(n))	O(n^2)

References

- The Design and Analysis of Algorithms by ***Anany Levitin***

Large enough to Deliver, Small enough to Care

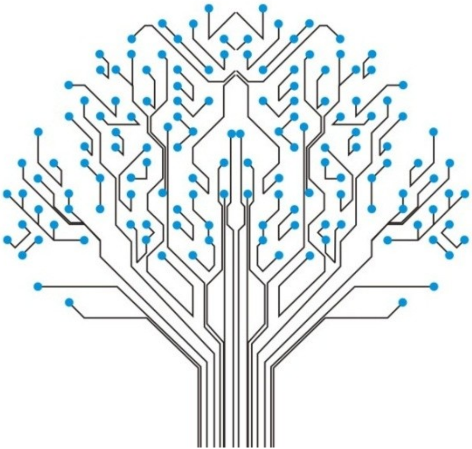
Global Village
IT SEZ
Bangalore

South Main Street
Milpitas
California

Raheja Mindspace
IT Park
Hyderabad

www.globaledgesoft.com

Thank you



Fairness

Learning

Responsibility

Innovation

Respect