Order of Growth of Algorithms

Efficiency of Algorithm

- Suppose X is an algorithm and n is the size of input data,
- The time and space used by the Algorithm X are the two main factors which decide the efficiency of X.
- Measuring Time Efficiency is also called Time Complexity Likewise we have space complexity
- Time complexity depends on
  - Input size
  - Basic operation

Concept of Basic Operation

- Time efficiency is analyzed by determining the number of repetitions of the *basic operation* as a function of *input size.*
- *Basic operation:* the operation that contributes most towards the running time of the algorithm.
  - As a rule, the basic operation is located in its inner-most loop
- Basic Operation in searching ?

Time Complexity

- Time Complexity of an algorithm represents the amount of time required by the algorithm to run to completion.
- Time requirements can be defined as a numerical function T(n), where T(n) can be measured as the number of steps, provided each step consumes constant time.

Space Complexity

- Space complexity of an algorithm represents the amount of memory space required by the algorithm in its life cycle.

Every case Time Complexity

- For a given algorithm, T(n) is every case time complexity if algorithm have to repeat its basic operation every time for given input size n. determination of T(n) is called every case time complexity analysis.

Every case time complexity(examples)

- Sum of elements of array

Algorithm sum_array(A,n)

sum=0

For i=1 to n

sum+=A[i]

Return n

Every case time complexity(examples)

- Basic operation addition of array elements
- Repeated how many number of times??
- Complexity n

Every case time complexity(examples)

- Exchange Sort

```
Algorithm exchange_sort(A,n)
For i=1 to n-1
    for j= i+1 to n
        if A[i]>A[j]
        exchange A[i] &A[j]
```

Every case time complexity(examples)

- Basic operation comparison of array elements
- Repeated how many number of times??
- Complexity n(n-1)/2

Best Case Time Complexity

- For a given algorithm, B(n) is every case time complexity if algorithm have to repeat its basic operation for minimum time for given input size n. determination of B(n) is called Best case time complexity analysis.

Best Case Time Complexity (Example)

```
Algorithm sequential_search(A,n,key)
i=0
While i<n &&A[i]!= key
   i=i+1
If  i<n
   return I
Else return-1
```

Best Case Time Complexity (Example)

- Input size: number of elements in the array ie n
- Basic operation :comparison of key with array elements
- Best case: first element is the required key

Worst Case Time Complexity

- For a given algorithm, W(n) is every case time complexity if algorithm have to repeat its basic operation for maximum number of times for given input size n. determination of W(n) is called worst case time complexity analysis.

Sequential Search

- Input size: number of elements in the array ie n
- Basic operation :comparison of key with array elements
- worst case: last element is the required key or key is not present in array at all
- Complexity :w(n)=n

Average Case Time Complexity

- For a given algorithm, A(n) is every case time complexity if algorithm have to repeat its basic operation for average number of times for given input size n. determination of A(n) is called average case time complexity analysis.

Sequential Search

- Input size: number of elements in the array i.e. n
- Basic operation :comparison of key with array elements

- Average Case time complexity is determined by considering probability of basic operation.

Order Of Growth

What is Order of Growth?

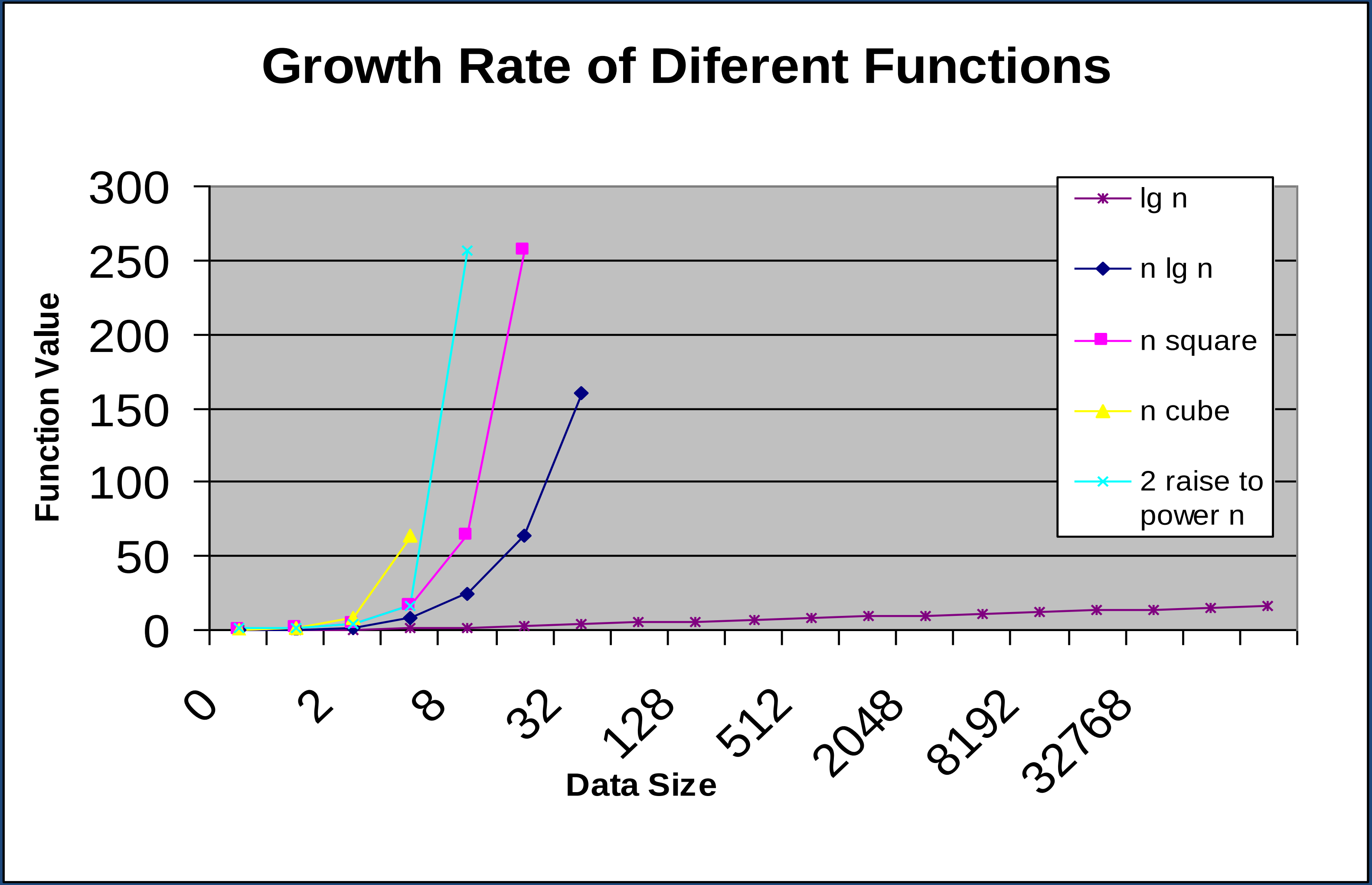
- When we want to compare two algorithm with respect to their behavior for large input size, a useful measure is so-called Order of growth.
- The order of growth can be estimated by taking into account the dominant term of the running time expression.

Dominant Term

- In the running time expression, when n becomes large a term will become significantly larger than the other ones: this is the so-called dominant term.

$T1(n)=an+b$	Dominant term: $a\ n$
$T2(n)=a \log n+b$	Dominant term: $a \log n$
$T3(n)=a\ n^2+bn+c$	Dominant term: $a\ n^2$
$T4(n)=a^n+b\ n+c$ $(a>1)$	Dominant term: a^n

Growth Rate


Growth Rate of Diferent Functions
300
250
200
150
100
50
0
Function Value
0
2
8
32
128
512
2048
8192
32768
Data Size
lg n
n lg n
n square
n cube
2 raise to power n

Growth Rates

n	lgn	nlgn	n^2	n^3	2^n
0	#NUM!	#NUM!	0	0	1
1	0	0	1	1	2
2	1	2	4	8	4
4	2	8	16	64	16
8	3	24	64	512	256
16	4	64	256	4096	65536
32	5	160	1024	32768	4294967296
64	6	384	4096	262144	1.84467E+19
128	7	896	16384	2097152	3.40282E+38
256	8	2048	65536	16777216	1.15792E+77
512	9	4608	262144	134217728	1.3408E+154
1024	10	10240	1048576	1073741824	
2048	11	22528	4194304	8589934592	

How can be interpreted the order of growth?

- Between two algorithms it is considered that the one having a smaller order of growth is more efficient
- However, this is true only for large enough input sizes

Example. Let us consider

$T1(n)=10n+10$ (linear order of growth)

$T2(n)=n^2$ (quadratic order of growth)

If $n<=10$ then $T1(n)>T2(n)$

Thus the order of growth is relevant only for $n>10$

Constant Factors

- The growth rate is not affected by
  - constant factors or
  - lower-order terms
- Examples
  - $10^2\boldsymbol{n} + 10^5$ is a linear function
  - $10^5\boldsymbol{n}^2 + 10^8\boldsymbol{n}$ is a quadratic function

Asymptotic Notations

Asymptotic Notations

- Asymptotic analysis refers to computing the running time of any operation in mathematical units of computation.
- For example, running time of one operation is computed as $f(n)$ and may be for another operation it is computed as $g(n^2)$.
- Which means first operation running time will increase linearly with the increase in n and running time of second operation will increase exponentially when n increases.

Asymptotic Notations

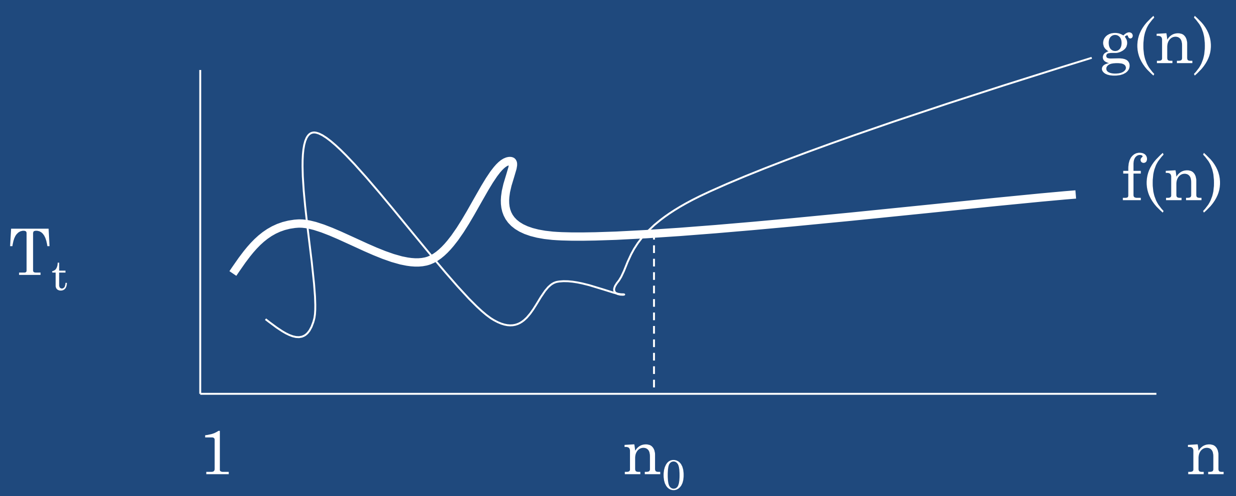
- Following are commonly used asymptotic notations used in calculating running time complexity of an algorithm.
- O Notation
- Ω Notation
- θ Notation

Big Oh Notation, O

- The O(n) is the formal way to express the upper bound of an algorithm's running time. It measures the worst case time complexity or longest amount of time an algorithm can possibly take to complete.

Big Oh Notation, O

There may be a situation, e.g.



f(n) <= g(n) for all n >= n_0 Or

f(n) <= cg(n) for all n >= n_0 and c = 1

g(n) is an **asymptotic upper bound** on f(n).

f(n) = O(g(n)) if there exist two positive constants c and n_0 such that

f(n) <= cg(n) for all n >= n_0

Omega Notation, Ω

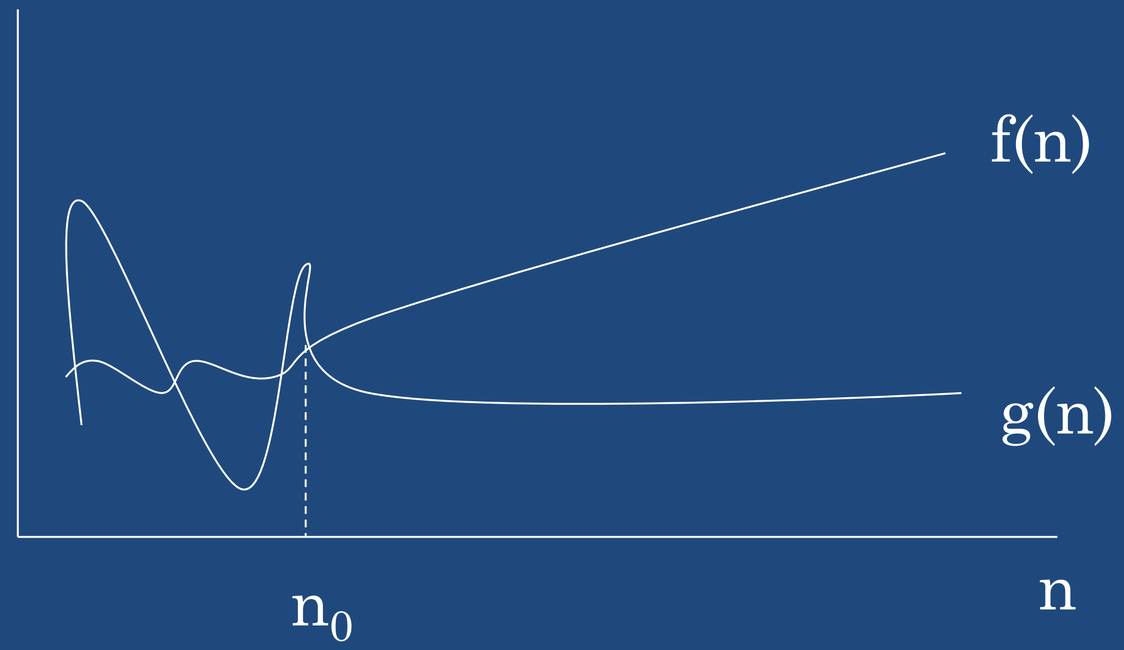
- The Ω(n) is the formal way to express the lower bound of an algorithm's running time. It measures the best case time complexity or best amount of time an algorithm can possibly take to complete.

Omega Notation, Ω

Asymptotic Lower Bound: $f(n) = \Omega(g(n))$,

if there exit positive constants c and n_0 such that

$$f(n) >= cg(n) \qquad \text{for all } n >= n_0$$



Theta Notation, θ

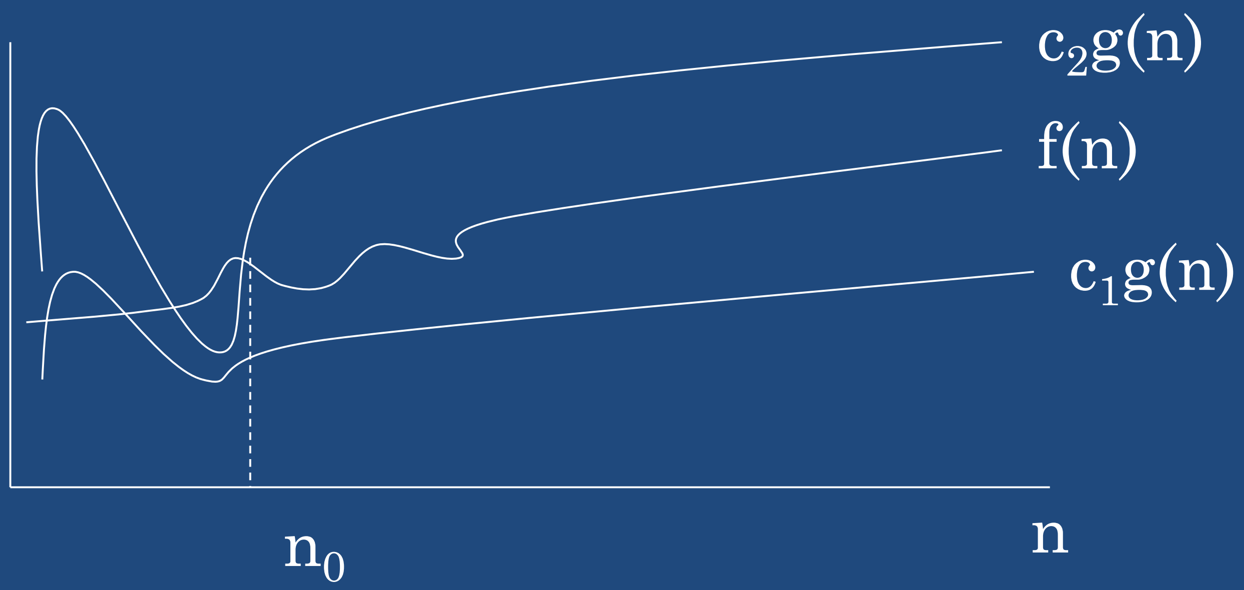
- The θ(n) is the formal way to express both the lower bound and upper bound of an algorithm's running time.
- This means that the best and worst case requires the same amount of time to within a constant factor.

Theta Notation, θ

Asymptotically Tight Bound: $f(n) = \theta(g(n))$,

iff there exit positive constants c_1 and c_2 and n_0 such that

$c_1\, g(n) <= f(n) <= c_2 g(n)$ for all $n >= n_0$



Big-Oh and Growth Rate

- The big-Oh notation gives an upper bound on the growth rate of a function
- The statement "$f(n)$ is $O(g(n))$" means that the growth rate of $f(n)$ is no more than the growth rate of $g(n)$
- We can use the big-Oh notation to rank functions according to their growth rate

	$f(n)$ is $O(g(n))$	$g(n)$ is $O(f(n))$
$g(n)$ grows more	Yes	No
$f(n)$ grows more	No	Yes
Same growth	Yes	Yes

Intuition for Asymptotic Notation

- **Big-Oh**
  f(n) is O(g(n)) if f(n) is asymptotically **less than or equal** to g(n)

- **Big-Omega**
  f(n) is Ω(g(n)) if f(n) is asymptotically **greater than or equal** to g(n)

- **Big-Theta**
  f(n) is Θ(g(n)) if f(n) is asymptotically **equal** to g(n)

Common Asymptotic Notations

constant	–	$O(1)$
logarithmic	–	$O(\log n)$
linear	–	$O(n)$
n log n	–	$O(n \log n)$
quadratic	–	$O(n^2)$
cubic	–	$O(n^3)$
polynomial	–	$n^{O(1)}$
exponential	–	$2^{O(n)}$