Algorithm Analysis

- Prepare a table as shown below

statement	cost	times_executed
1		
2		
3		
4		
5		

- Problem: To determine the largest of n nos.
  - Running time if the input is strictly ascending/strictly descending ?

Algorithm Largest$_1$(x_i, n)

```
1 let max = x1.....   C1
2 for i = 2 to n....  C2
3 do if xi > max...   C3
4    then max = xi..  C4
5 Print max........   C5
```

Total steps = Total time = $C_1 + nC_2 + (n-1)C_3 + (n-1)C_4 + C_5$

Algorithm Largest$_2$(x_i, n)

```
1 for i = 1 to (n-1)....  C1
2 do if xi > xi+1.......  C2
3   then swap <xi, xi+1>..  C3
4 Print xn..............  C4
```

Total steps = Total time = $nC_1 + (n-1)\ C_2 + (n-1)C_3 + C_4$

- The execution time contributed for a statement that takes C_i steps
  - to execute and is executed *n* times ……is $C_i n$ ……to the total running time.

Algorithm Analysis

- Major Assumptions
  - the abstract operations are machine independent.
  - a constant amount of time is required to execute each line of pseudocode

Total steps = Total time = $C_1 + nC_2 + (n-1)C_3 + (n-1)C_4 + C_5$

Total steps = Total time = $nC_1 + (n-1)\ C_2 + (n-1)C_3 + C_4$

- How to count the program steps?
  - comments, declarations
  - assignment statement
  - iterative statement
- How to instantiate the values of c_i's ?

Insertion Sort

Algorithm Insertion-Sort(A[], n)

```
1 for j = 2 to n
2     do key = A[j]
3     i = j -1
4     while (i>0) and (A[i] > key)
5     do A[i+1] = A[i]
6           i = i-1
7     A[i+1]=key
```

j = 2	i = 1	key=12	13	12	10
j = 2	i = 1	key=12	13	13	10
j = 2	i = 0	key=12	12	13	10

j = 3	i = 2	key=10	12	13	10
j = 3	i = 2	key=10	12	13	13
j = 3	i = 1	key=10	12	12	13
j = 3	i=0	key=10	10	12	13

- How do we analyze the time complexity?

Insertion Sort

Algorithm Insertion-Sort(A[], n)

```
1 for j = 2 to n
2     do key = A[j]
3     i = j -1
4     while (i>0) and (A[i] > key)
5     do A[i+1] = A[i]
6           i = i-1
7     A[i+1]=key
```

j = 2	i = 1	key=12	13	12	10
j = 2	i = 1	key=12	13	13	10
j = 2	i = 0	key=12	12	13	10

j = 3	i = 2	key=10	12	13	10
j = 3	i = 2	key=10	12	13	13
j = 3	i = 1	key=10	12	12	13
j = 3	i=0	key=10	10	12	13

- How do we analyze the time complexity?
- We need to analyze how many times the while loop is executed ?
  - Assume while loop *test* is executed t_j times .....

Insertion sort Analysis

- Then, the running time is given by the expression

$$c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7(n-1)$$

```
1 for j = 2 to n
2     do key = A[j]
3     i = j -1
4     while (i>0) and (A[i] > key)
5      do A[i+1] = A[i]
6      i = i - 1
7      ……
```

Simulating Insertion Sort: Best case

j = 2	i = 1	key=12	10	12	13
while loop executed only once to test the condition……. i.e. t_j=1			A[i] !> key		

```
for j = 2 to n
      do key = A[j]
      i = j -1
      while (i>0) and (A[i] > key)
       do A[i+1] = A[i]
         … … … …
```

Simulating Insertion Sort: Best case

j = 3	i = 2	key=13	10	12	13
while loop executed only once to test the condition……. i.e. t_j=1			A[i] !> key		

```
for j = 2 to n
      do key = A[j]
      i = j -1
      while (i>0) and (A[i] > key)
       do A[i+1] = A[i]
         … … … …
```

Insertion sort :Best-case Analysis

- When does the best case occur?
  - t_j = 1in every case here
- Then, the best case running time is

$$c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} 1 + c_5 \sum_{j=2}^{n} 0 + c_6 \sum_{j=2}^{n} 0 + c_7(n-1)$$

```
1 for j = 2 to n
2     do key = A[j]
3     i = j -1
4     while (i>0) and (A[i] > key)
5      do A[i+1] = A[i]
6      i = i - 1
7      ……
```

- So, the best case running time is a linear function of inputs n.

Insertion sort : Worst-case analysis

- When does the worst case occur?
  - At least when the while loop is executed for all the *i* values…….
  - i.e. when the array is reverse sorted

```
1 for j = 2 to n
2     do key = A[j]
3     i = j -1
4     while (i>0) and (A[i] > key)
5      do A[i+1] = A[i]
6      i = i - 1
7      ……
```

Simulating Insertion Sort: Worst case

j = 2	i = 1	key=12	13	12	10
j = 2	i = 1	key=12	13	13	10
j = 2	i = 0	key=12	12	13	10

- j=2 and so two iterations…..

```
1 for j = 2 to n
2     do key = A[j]
3     i = j -1
4     while (i>0) and (A[i] > key)
5      do A[i+1] = A[i]
6      i = i - 1
7      ……
```

Simulating Insertion Sort: Worst case

j = 3	i = 2	key=10	12	13	10
j = 3	i = 2	key=10	12	13	13
j = 3	i = 1	key=10	12	12	13
j = 3	i = 0	key=10	10	12	13

- j=3 and so three iterations..

```
1 for j = 2 to n
2     do key = A[j]
3     i = j -1
4     while (i>0) and (A[i] > key)
5      do A[i+1] = A[i]
6      i = i - 1
7      ……
```

Insertion sort : Worst-case analysis`

- Thus, t_j = j in this case…..
- Therefore the expression is

$$c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} j + c_5 \sum_{j=2}^{n} (j-1) + c_6 \sum_{j=2}^{n} (j-1) + c_7(n-1)$$

Insertion sort : Worst-case analysis`

- Thus, t_j = j in this case…..
- Therefore the expression is

$$c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} j + c_5 \sum_{j=2}^{n} (j-1) + c_6 \sum_{j=2}^{n} (j-1) + c_7(n-1)$$

- Now,
  - Sigma of j=2 to n of j is (n(n+1)/2) -1
  - Sigma of j=2 to n of (j-1) is (n(n-1)/2)

- Then, the worst case running time is

```
T(n) =    c1n + c2(n-1) + c3(n-1)
        + c4((n(n+1)/2) -1) + c5(n(n-1))/2
        + c6(n(n-1))/2 + c7(n-1)
```

Growth of Functions

- Consider
  - an algorithm A which for a problem, does **2n** basic operations & **$2C_1n$** total operations; while some other algorithm does **4.5n** basic and **$4.5C_2n$** total operations.
    - Consider constant of proportionality representing overhead operations.
  - Which algorithm of the two do you think is better?

Growth of Functions (contd)

n	2n	4.5n	$n^3/2$	$5n^2$
5	10	22		
10	20	45		
100	200	450		
1000	2000	4500		
10000	20000	45000		
100000	$2.0 * 10^5$	$4.5 * 10^5$		
1000000	$2.0 * 10^6$	$4.5 * 10^6$		

Growth of Functions (contd)

- Consider
  - another such example with algo1 taking $\mathbf{n^3/2}$ multiplicative steps while algo2 taking $\mathbf{5n^2}$ steps.
    - Consider constant of proportionality representing overhead operations.
  - Which algorithm of the two do you think is better?

Growth of Functions (contd)

n	2n	4.5n	$n^3/2$	$5n^2$
5	10	22	45	125
10	20	45	500	500
100	200	450	$5*10^5$	$5*10^4$
1000	2000	4500	$5*10^8$	$5*10^6$
10000	20000	45000	$5*10^{11}$	$5*10^8$
100000	200000	450000	$5*10^{14}$	$5*10^{10}$
1000000	2000000	4500000	$5*10^{17}$	$5*10^{12}$

Order of growth and abstractions

- A relook at the costs of insertion sort with c_i's = 1
- Best case

`T(n)`

Order of growth and abstractions

- A relook at the costs of insertion sort with c_i's = 1
- Best case

```
T(n)
= c1n + c2(n-1) + c3(n-1) + c4(n-1) + c7(n-1)
= 5n - 4
```

Order of growth and abstractions

- A relook at the costs of insertion sort with c_i's = 1
- Best case

```
T(n)
= c1n + c2(n-1) + c3(n-1) + c4(n-1) + c7(n-1)
= 5n - 4
```

- Worst case

```
T(n)
```

Order of growth and abstractions

- A relook at the costs of insertion sort with c_i's = 1
- Best case

```
T(n)
= c1n + c2(n-1) + c3(n-1) + c4(n-1) + c7(n-1)
= 5n - 4
```

- Worst case

```
T(n)
= c1n + c2(n-1) + c3(n-1)+ c4((n(n+1)/2) -1)
      + c5(n(n-1))/2 + c6(n(n-1))/2 + c7(n-1)
= 3n² + 7n - 8
```

Order of growth and abstractions

- A relook at the costs of insertion sort with c_i's = 1
- Best case

```
T(n)
= c1n + c2(n-1) + c3(n-1) + c4(n-1) + c7(n-1)
= 5n - 4
```

- Worst case

```
T(n)
= c1n + c2(n-1) + c3(n-1)+ c4((n(n+1)/2) -1)
      + c5(n(n-1))/2 + c6(n(n-1))/2 + c7(n-1)
= 3n² + 7n - 8
```

- Which term dominates the overall result in the above expression, especially at large values of n ?

Order of growth and abstractions

- Which term dominates the overall result in the given expression, especially at large values of n ?

n	$T(n) = 3n^2 + 7n - 8$	$T(n) = 3n^2$
10	362	300
100	30692	30000
1000	$3.006992 * 10^6$	$3.00 * 10^6$
100000	$3.0000699992 * 10^{10}$	$3.00 * 10^{10}$

Growth of Functions (contd)

- Does constant of proportionality matter when n gets very large?
- Then, what is asymptotic growth rate, asymptotic order or order of functions?
- Is it reasonable to ignore smaller values and constants?

Growth of Functions (contd)

- Hence, we shall now also drop the all the terms
  - except the highest degree of the polynomial for the running time of the algorithm…….
  - i.e.
- Insertion sort Best case complexity……..

  $T(n) = 5n - 4$
  - So, we will say that complexity is of the order of n
- Insertion sort Worst case complexity……..

  $T(n) = 3n^2 + 7n - 8$
  - So, we will say that complexity is of the order of n^2

Order of growth and abstractions

- So, now we have assumed/abstracted at three different levels viz.
  - level 1 – ignored the actual cost of execution of each statement.
  - level 2 – ignored even the abstract cost (Ci) of each statement
  - level 3 – ignore all the terms except for the one with the highest degree in the expression of time complexity

- Such analysis is based on the asymptotic growth rate, asymptotic order or order of functions and called
  - asymptotic analysis

Worst-case analysis

- Again, we would normally focus only on the worst case analysis
- Why do we usually concentrate only on worst-case analysis?

Worst-case analysis

- Again, we would normally focus only on the worst case analysis
- Why do we usually concentrate only on worst-case analysis?
  - being upper bound, the worst case guarantees that the algorithm will not take any longer.
  - also, worst case occurs fairly often in many applications.
  - average case is often roughly as bad as the worst case.

Order of growth – various asymptotic orders

- What does the following comparison indicate?

lg n	$n^{1/2}$	n	n lg n	n $(\lg n)^2$	n^2
3	3	10	33	110	100
7	10	100	664	4414	10000
10	32	1000	9966	99317	10^6
13	100	10000	132877	1765633	10^8
17	316	100000	16660964	27588016	10^{10}
20	1000	1000000	19931569	397267426	10^{12}

An interesting "seconds" conversion

10^2	1.7 min
10^4	2.8 hours
10^5	1.1 days
10^6	1.6 weeks
10^7	3.8 months
10^8	3.1 years
10^9	3.1 decades
10^{10}	3.1 centuries

An interesting observation

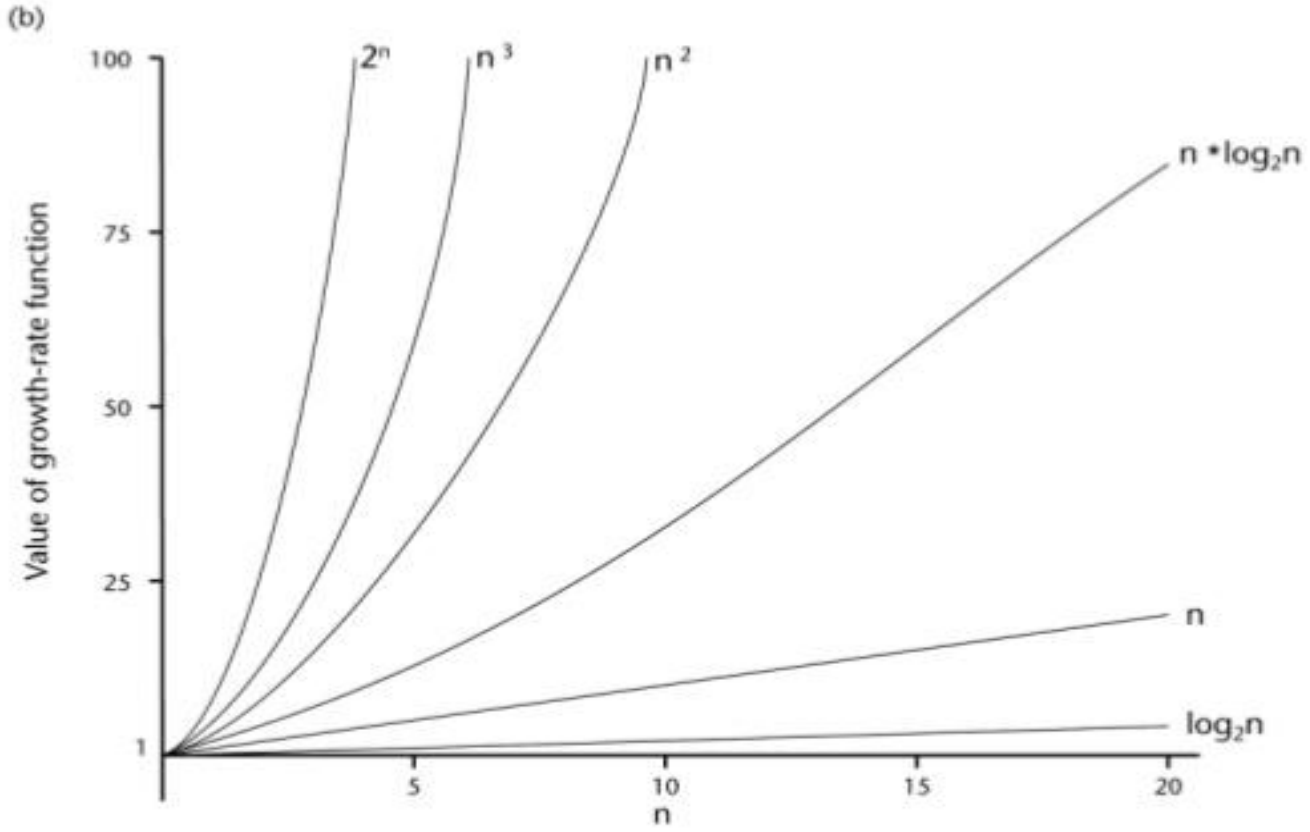
	n	$n \lg n$	N^2	N^3	1.5^n	2^n	$n!$
n=10	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 4 sec
n=30	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} yrs
n=50	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 yrs	very long
n=100	< 1 sec	< 1 sec	< 1 sec	1 sec	12.89 yrs	10^{17} yrs	very long
n=1000	< 1 sec	< 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

Basic Asymptotic Efficiency classes

1	constant
log n	logarithmic
n	linear
n log n	n log n
n^2	quadratic
n^3	cubic
2^n	exponential
$n!$	factorial

Growth of Functions (contd)

A Comparison of Growth-Rate Functions (cont.)



Asymptotic notations : The Big-O notation

- definition
  - for a given function $g(n)$, we say that

    $O(g(n)) = \{f(n) |$ if there exists positive constants c and n_o such that, $0 <= f(n) <= cg(n)$ for all $n > n_o\}$
- defines an upper bound for a function within a constant factor.
  - i.e. except for a constant factor and a finite number of exceptions, f is bounded above by g.
- $f(n) = O(g(n)) \Rightarrow$
  - $f(n)$ is dominated in the growth by $g(n)$ i.e.
  - $f(n)$ is of the order at most $g(n)$ i.e.
  - $g(n)$ grows at least as fast as $f(n)$
- Can **f(n)** grow faster than **g(n)**?
- Can **g(n)** grow faster than **f(n)**?

Big-oh notation

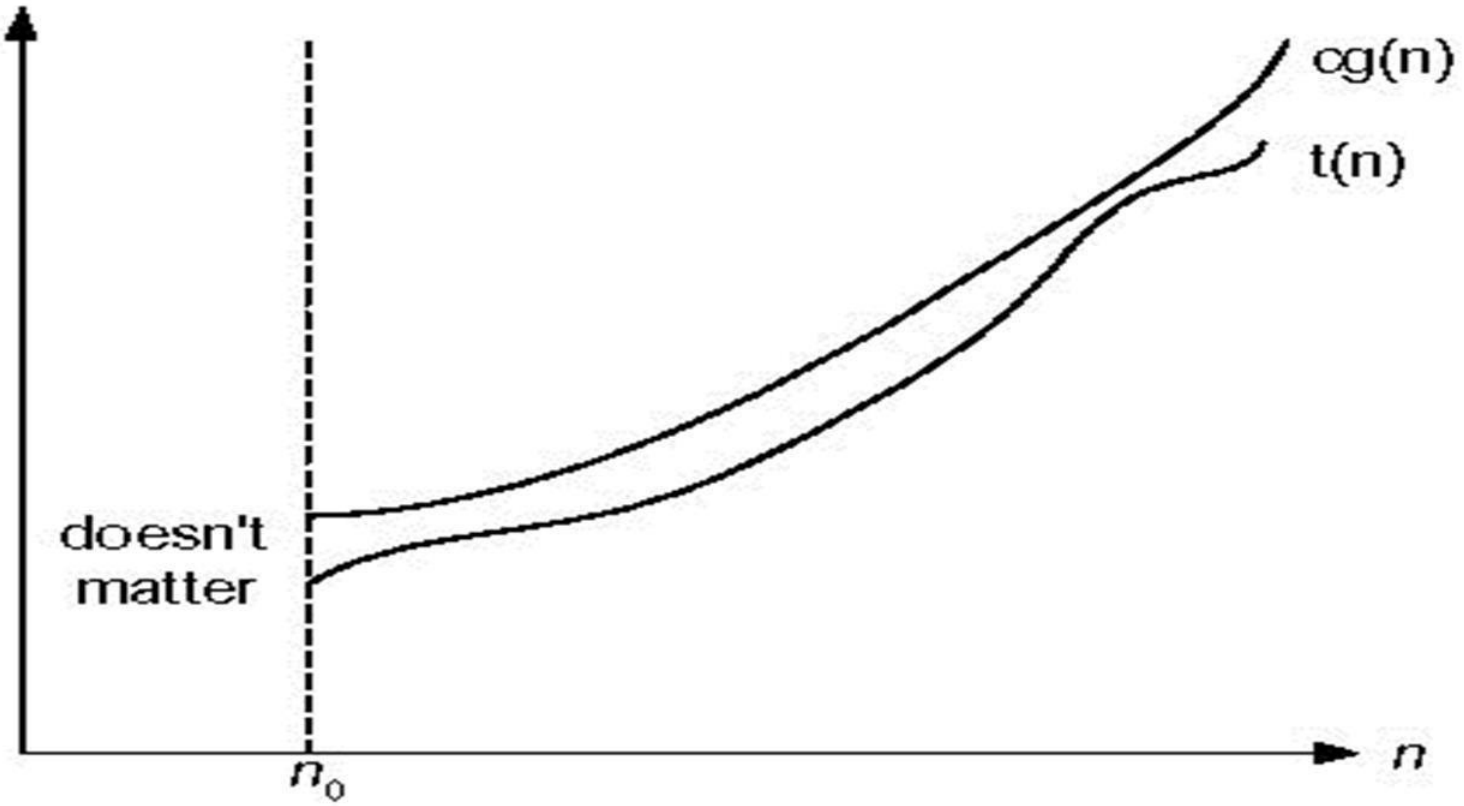


Figure 2.1 Big-oh notation: $t(n) \in O(g(n))$

The Big-O notation : Illustrations

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

The Big-Ω notation (contd)

- Many a times the big-O notation is misused
  - e.g. saying that the running time of insertion sort is $O(n^2)$
- the big-**Ω** notation is used to specify this aspect i.e.
  - to define a lower bound for a function within a constant factor..
- definition
  - the function $f(n) = \Omega(g(n))$ is true iff there exists positive constants c and n_0 such that $f(n) >= c * (g(n))$ for all $n, n >= n_0$ i.e. $0 <= c(g(n)) <= f(n)$.
  - i.e. except for a constant factor and a finite number of exceptions, f is bounded below by g.

Big-Ω notation

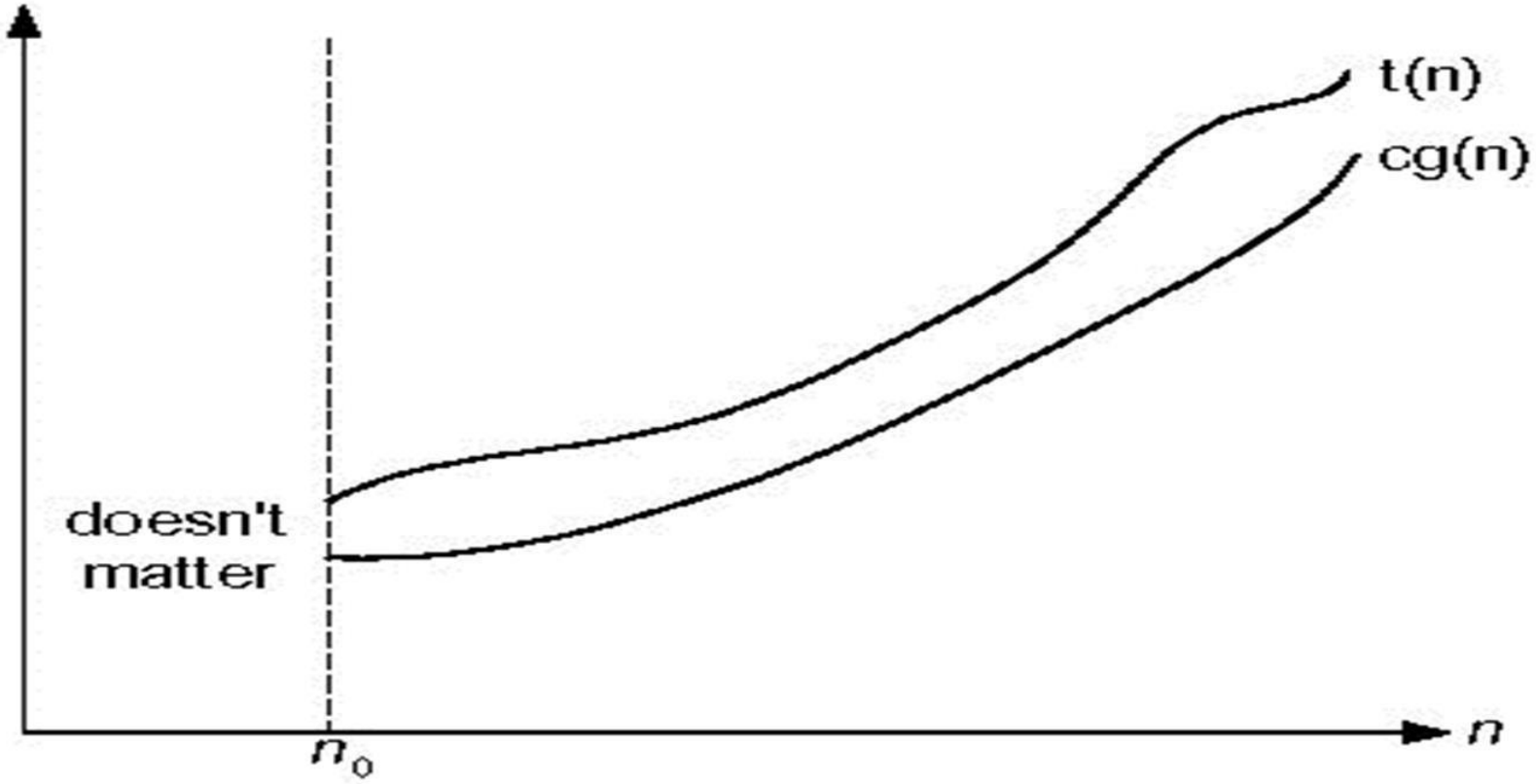


Fig. 2.2 Big-omega notation: $t(n) \in \Omega(g(n))$

The Big-Ω notation : Illustrations

Function	notation in Ω
$f(n) = 3n + 8$	$f(n) = \Omega(?)$
$f(n) = n^2 + 3n - 8$	$f(n) = \Omega(?)$
$F(n) = 12n^2 - 11$	$f(n) = \Omega(?)$
$F(n) = 6*2^n + n^2$	$f(n) = \Omega(?)$
$f(n) = 3n + 8$	$f(n) = \Omega(n^2)?$
$f(n) = 5n + 8$	$f(n) = \Omega(1)?$

The Big-Ω notation : Illustrations (contd)

- Is it correct to say that $3n + 8 = \Omega(1)$?
- there are several functions $g(n)$ for which $f(n) = \Omega(g(n))$.
- the function $g(n)$ is only lower bound on n.
- hence, for $f(n) = \Omega(g(n))$ to be meaningful,
  - $g(n)$ should be as large a function of n as possible.
- Which one shall we choose if $3n+3 = \Omega(n)$ and $3n+3 = \Omega(1)$?

The θ-notation

- Neither the big-O notation nor the big-Ω notation describe the asymptotically tight bounds.
- θ-notation to express tighter bounds
- used to specify the exact order of growth of functions.
- def
  - we say that $f(n)=\theta((g)n)$ iff there exists positive constants $c1$ and $c2$ and a number n_0 such that
  - $0<=c1g(n)<=f(n)<=c2g(n)$ for all $n >= n_0$
  - $f(n)=\theta((g)n)$ iff $f(n)=O(g(n))$ and $f(n)=\Omega(g(n))$
- Graphically,

Big-θ notation

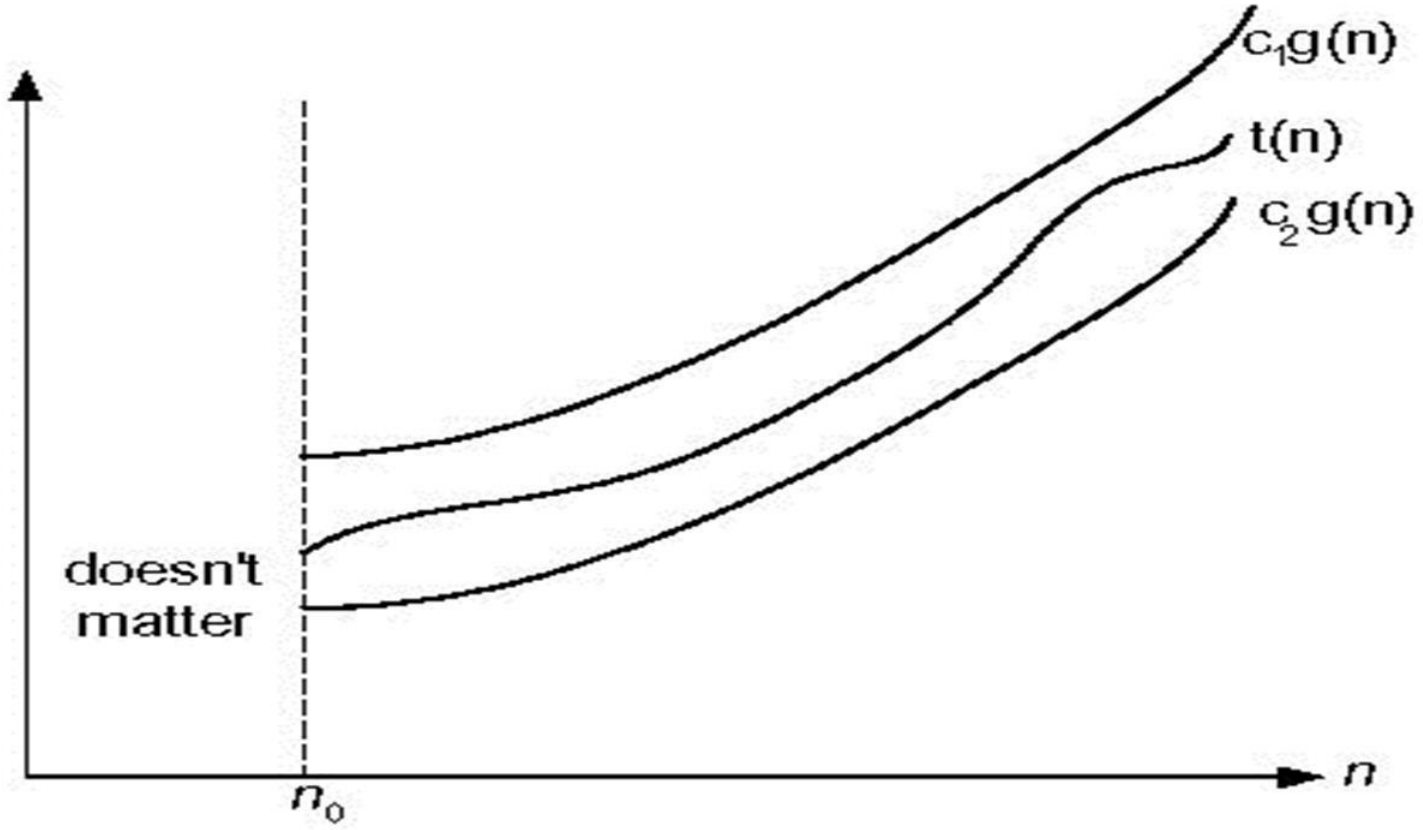


Figure 2.3 Big-theta notation: $t(n) \in \Theta(g(n))$

The θ-notation (contd)

- Let $f(n)=1/2n^2-3n$. Is $f(n)=\theta(n^2)$?
- Can we say $6n^3=\theta(n^2)$?

Function	notation in θ
$f(n) = 3n + 8$	$f(n)= \theta(?)$
$f(n) = 10n^2 + 3n - 8$	$f(n)= \theta(?)$
$F(n) = 12n^2 - 11$	$f(n)= \theta(?)$
$F(n) = 6*2^n + n^2$	$f(n)= \theta(2^n)$?
$F(n) = 6*2^n + n^2$	$f(n)= \theta(n^2)$?
$f(n) = 3n + 8$	$f(n)= \theta(n^2)$?
$f(n) = 5n + 8$	$f(n)= \theta(1)$?

Review of Asymptotic notations

- A way of comparing functions that ignores constant factors and small input sizes
- **O(g(n))**
  - class of functions f(n) that grow no faster than g(n)
- **Θ(g(n))**
  - class of functions f(n) that grow at same rate as g(n)
- **Ω(g(n))**
  - class of functions f(n) that grow at least as fast as g(n)

Complexities of various Data Structures

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Binary Search Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Complexities of various Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	O(n log(n))	O(n log(n))	O(n^2)	O(log(n))
Mergesort	O(n log(n))	O(n log(n))	O(n log(n))	O(n)
Bubble_Sort	O(n)	O(n^2)	O(n^2)	O(1)
Insertion_Sort	O(n)	O(n^2)	O(n^2)	O(1)
Selection_Sort	O(n^2)	O(n^2)	O(n^2)	O(1)