Asymptotic Notations

- Asymptotic notations are used to represent the time complexity of algorithm.
- If we analyze the algorithm by studying and finding frequency of execution of statements then we get time complexity of any algorithm.
- The time complexity can be any one among these

Constant	1
Logarithmic	log n
Linear	n
Qudratic	n^2
Cubic	n^3
Exponetial	2^n
Factorial	n!

- Asymptotic notations are used to show the function belongs to which class

- Example 1
- Algorithm Add(a,b)
- {
- Return a+b;
- }

T(n)=O(1)

Time complexity of this algorithm is constant

Example 2

```
sum (  a, n)
{
int s=0;
for(i=0;i<n;i++)
{
 s=+a[i];
}
return (s);
}
```

$T(n)=2n+2$

Time complexity of this algorithm is linear

$O(n)$

Example 3: Matrix multiplication

ALGORITHM *MatrixMultiplication*$(A[0..n-1, 0..n-1], B[0..n-1, 0..n-1])$

```
//Multiplies two n-by-n matrices by the definition-based algorithm
//Input: Two n-by-n matrices A and B
//Output: Matrix C = AB
for i ← 0 to n − 1 do
    for j ← 0 to n − 1 do
        C[i, j] ← 0.0
        for k ← 0 to n − 1 do
            C[i, j] ← C[i, j] + A[i, k] ∗ B[k, j]
return C
```

T(n)=**O(n^3)** multiplications

- Asymptotic notations are
- O (Big Oh) upper bound of function
- Θ (Big theta) lower bound of function
- Ω (Big omega) Average bound(tight bound)
- $1<\log<\sqrt{n}<n<n\log n<n^2<n^3\ldots\ldots<2^n<3^n\ldots.<n^n$

O (Big Oh) upper bound of function

Definition: $f(n)$ is in $O(g(n))$, denoted $f(n) \in O(g(n))$,

if order of growth of $f(n) \leq$ order of growth of $g(n)$ (within constant multiple), i.e., there exist positive constant c and non-negative integer n_0 such that

$$f(n) \leq c * g(n) \text{ for every } n \geq n_0$$

Examples

- F(n)=2n+3

  $2n+3<=10n$

  $5n<=10n \quad n>=1$

  $2n+3<=2n+3n$

  $2n+3<=2n^2+3n^2 \quad n>=1$

  $F(n)=O(n^2)$

  $F(n)=O(2^n)$

  - F(n)=O(log n) this cannot be written because logn belongs to lower bound.
  - Hence n and above n all functions can be written as O(n)

Big-oh

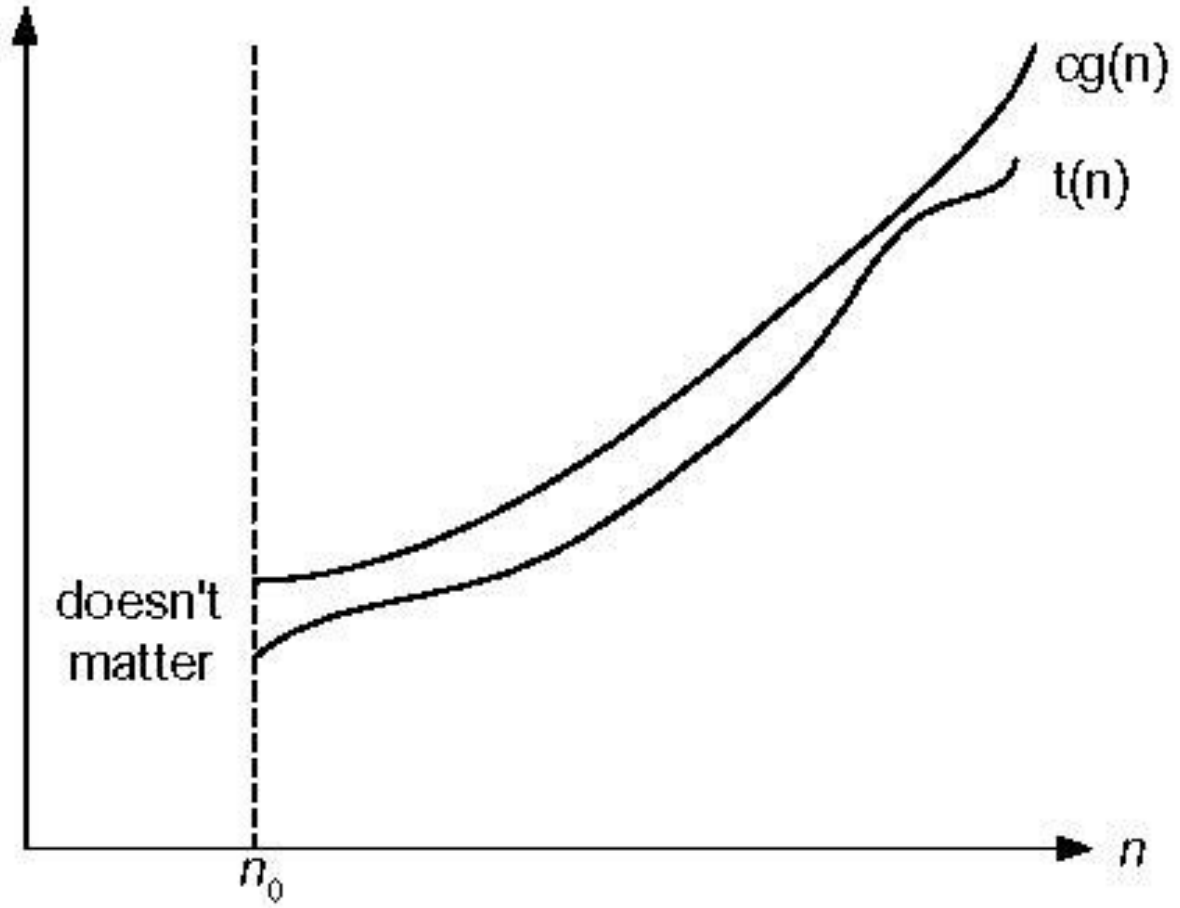


Figure 2.1 Big-oh notation: $t(n) \in O(g(n))$

Ω-notation

- Formal definition
  - A function *t(n)* is said to be in *Ω(g(n)),* denoted *t(n)* $\in$ *Ω(g(n)),* if *t(n)* is bounded below by some constant multiple of *g(n)* for all large *n*, i.e., if there exist some positive constant c and some nonnegative integer n_0 such that

    $t(n) \geq c*g(n)$ for all $n \geq n_0$

F(n)=2n+3

2n+3>=1*n n>=1

c*g(n)

F(n)= Ω(n)

F(n)= Ω(logn)

F(n)= $\Omega(n^2)$cannot e written because n^2 belongs to upperbound

Big-omega

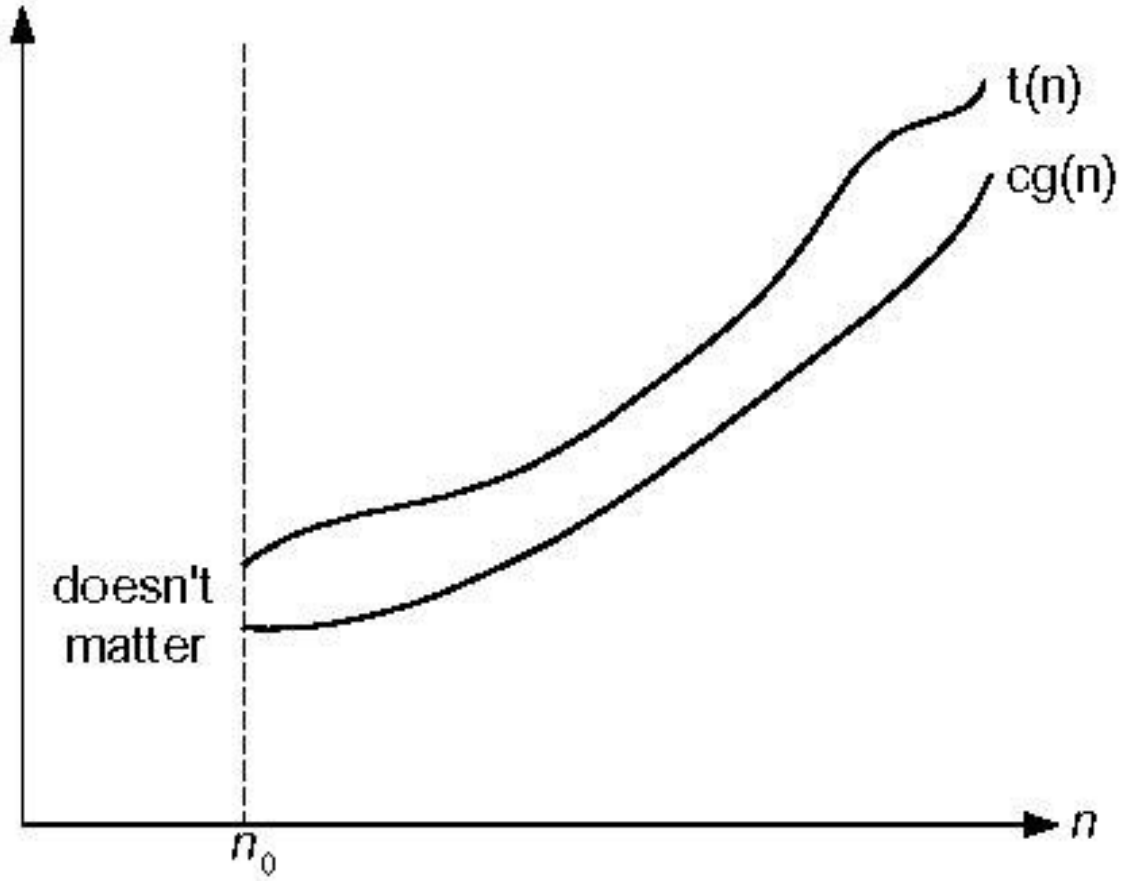


Fig. 2.2 Big-omega notation: $t(n) \in \Omega(g(n))$

Θ-notation

- Formal definition
- A function *t(n)* is said to be in Θ*(g(n))*, denoted *t(n)* $\in$ Θ*(g(n))*, if *t(n)* is bounded both above and below by some positive constant multiples of *g(n)* for all large *n*, i.e., if there exist some positive constant c_1 and c_2 and some nonnegative integer n_0 such that

  $c_2\, g(n) \leq t(n) \leq c_1\, g(n)$ for all $n \geq n_0$

Examples

- F(n)=2n+3
- 1*n<=2n+3<=5n
- C1 *g(n)<=f(n)<=C2*g(n)
- F(n)= $\Theta(n)$

Big-theta

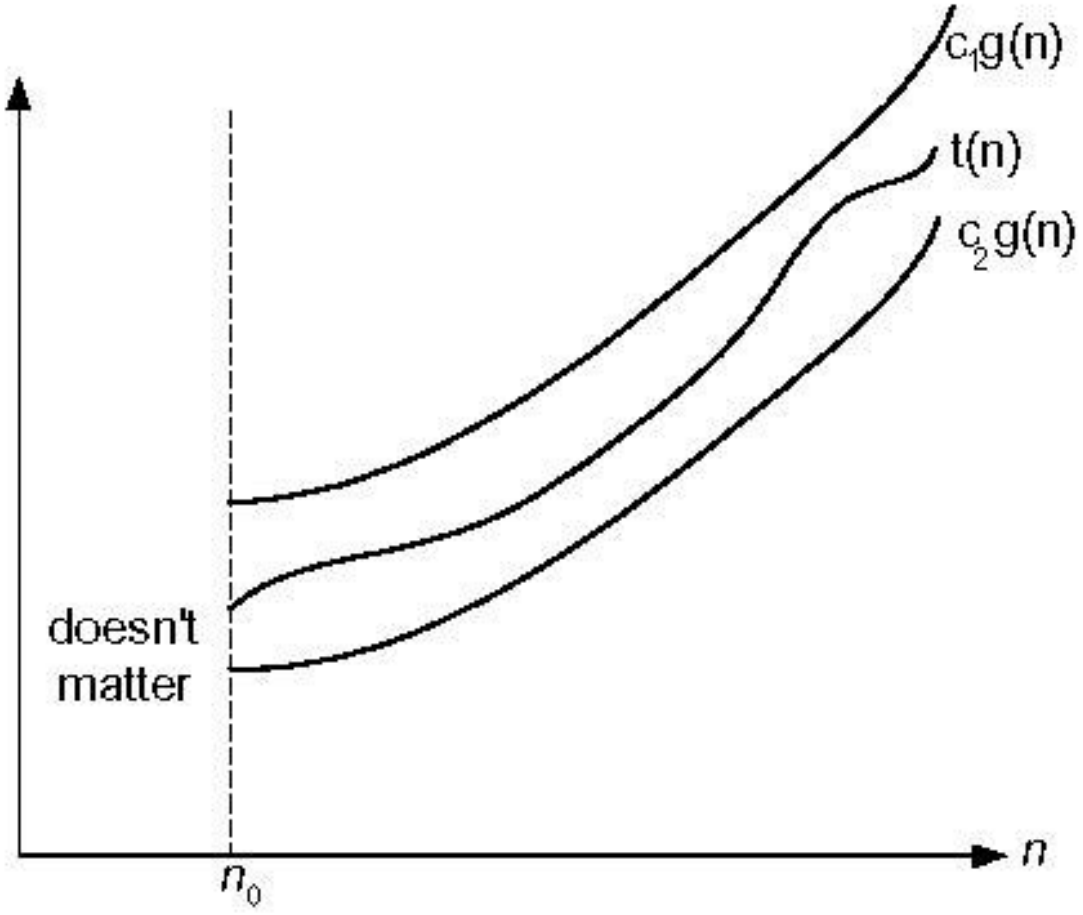


Figure 2.3 Big-theta notation: $t(n) \in \Theta(g(n))$

Time efficiency of nonrecursive algorithms

General Plan for Analysis

- Decide on parameter *n* indicating *input size*
- Identify algorithm's *basic operation*
- Determine *worst*, *average*, and *best* cases for input of size *n*
- Set up a sum for the number of times the basic operation is executed
- Simplify the sum using standard formulas and rules

Example 1: Maximum element

ALGORITHM *MaxElement*$(A[0..n-1])$

```
//Determines the value of the largest element in a given array
//Input: An array A[0..n − 1] of real numbers
//Output: The value of the largest element in A
maxval ← A[0]
for i ← 1 to n − 1 do
    if A[i] > maxval
        maxval ← A[i]
return maxval
```

$\Sigma_{1 \le i \le n-1} 1 = n-1 = \Theta(n)$

Example 2: Element uniqueness problem

ALGORITHM *UniqueElements*$(A[0..n-1])$

```
//Determines whether all the elements in a given array are distinct
//Input: An array A[0..n − 1]
//Output: Returns "true" if all the elements in A are distinct
//          and "false" otherwise
for i ← 0 to n − 2 do
    for j ← i + 1 to n − 1 do
        if A[i] = A[j] return false
return true
```

T(n) = $\Sigma_{0\le i\le n-2}\ (\Sigma_{i+1\le j\le n-1}\ 1)$

$= \Sigma_{0\le i\le n-2}\ n-i-1 = (n-1+1)(n-1)/2$

$= \Theta(n^2)$ comparisons

Example 3: Matrix multiplication

ALGORITHM *MatrixMultiplication*$(A[0..n-1, 0..n-1], B[0..n-1, 0..n-1])$

```
//Multiplies two n-by-n matrices by the definition-based algorithm
//Input: Two n-by-n matrices A and B
//Output: Matrix C = AB
for i ← 0 to n − 1 do
    for j ← 0 to n − 1 do
        C[i, j] ← 0.0
        for k ← 0 to n − 1 do
            C[i, j] ← C[i, j] + A[i, k] ∗ B[k, j]
return C
```

T(n) = $\Sigma_{0\le i\le n-1} \Sigma_{0\le i\le n-1}\ n$

= $\Sigma_{0\le i\le n-1} \Theta(n^2)$

= $\Theta(n^3)$ multiplications

Plan for Analysis of Recursive Algorithms

- Decide on a parameter indicating an input's size.
- Identify the algorithm's basic operation.
- Check whether the number of times the basic op. is executed may vary on different inputs of the same size. (If it may, the worst, average, and best cases must be investigated separately.)
- Set up a recurrence relation with an appropriate initial condition expressing the number of times the basic op. is executed.
- Solve the recurrence (or, at the very least, establish its solution's order of growth) by backward substitutions or another method.

Example 1: Recursive evaluation of $n!$

Definition: $n! = 1 * 2 * \ldots * (n-1) * n$ for $n \geq 1$ and $0! = 1$

Recursive definition of $n!$: $F(n) = F(n-1) * n$ for $n \geq 1$ and $F(0) = 1$

ALGORITHM $F(n)$

```
//Computes n! recursively
//Input: A nonnegative integer n
//Output: The value of n!
if n = 0 return 1
else return F(n − 1) * n
```

Size: n

Basic operation: multiplication

Recurrence relation: $M(n) = M(n-1) + 1$

$M(0) = 0$

Solving the recurrence for M(*n*)

M(*n*) = M(*n*-1) + 1, M(0) = 0

M(n) = M(n-1) + 1

= (M(n-2) + 1) + 1 = M(n-2) + 2

= (M(n-3) + 1) + 2 = M(n-3) + 3

…

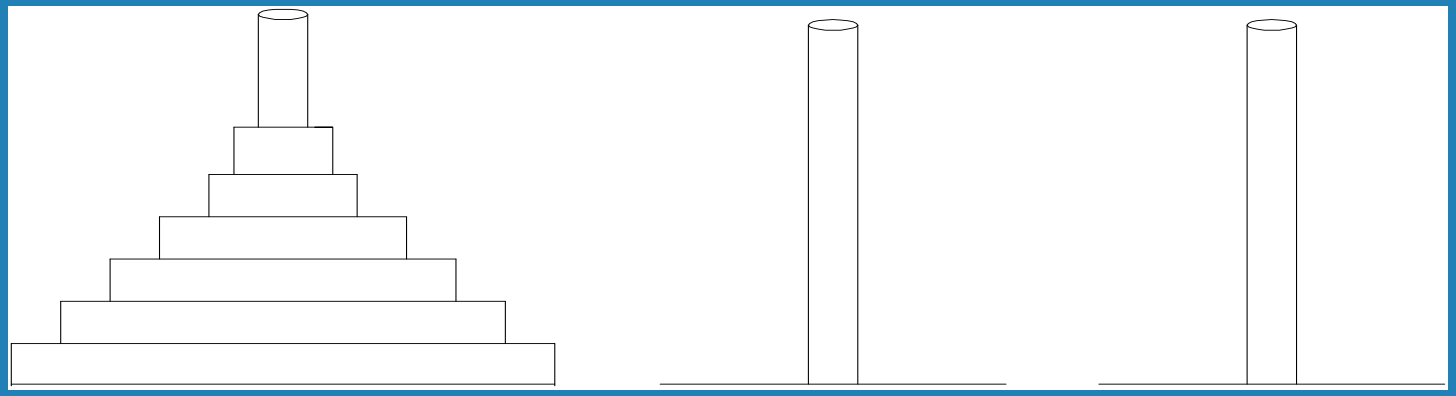
= M(n-i) + i

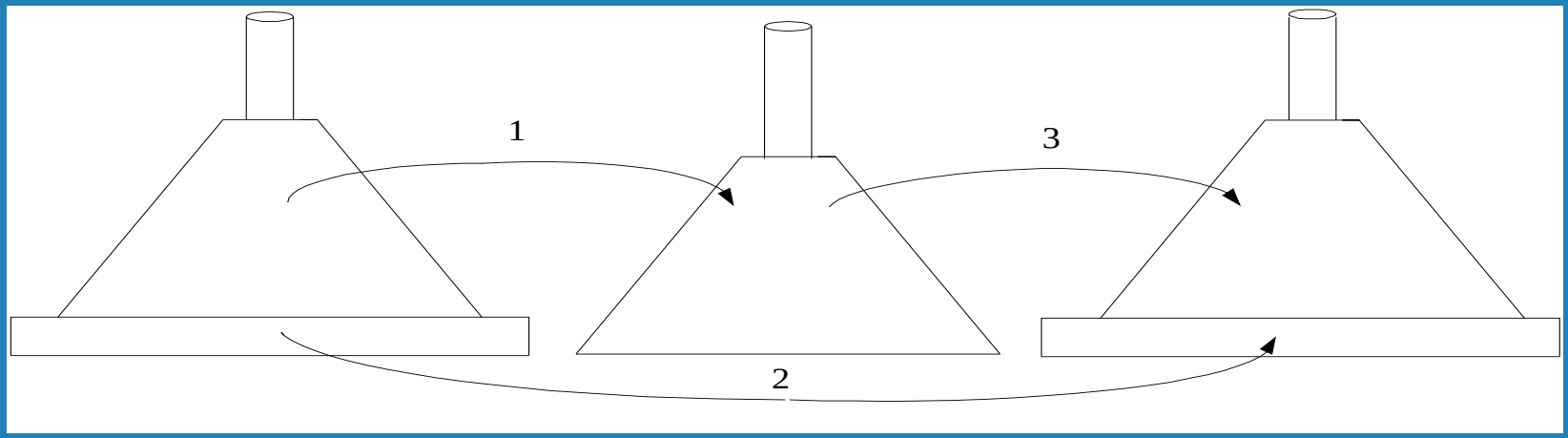
= M(0) + n

= n

The method is called backward substitution.

Example 2: The Tower of Hanoi Puzzle





Recurrence for number of moves:

$$M(n) = 2M(n-1) + 1$$

Solving recurrence for number of moves

$M(n) = 2M(n-1) + 1, \quad M(1) = 1$

$M(n) = 2M(n-1) + 1$

$= 2(2M(n-2) + 1) + 1 = 2^2 * M(n-2) + 2^1 + 2^0$

$= 2^2 * (2M(n-3) + 1) + 2^1 + 2^0$

$= 2^3 * M(n-3) + 2^2 + 2^1 + 2^0$

$= \ldots$

$= 2^{(n-1)} * M(1) + 2^{(n-2)} + \ldots + 2^1 + 2^0$

$= 2^{(n-1)} + 2^{(n-2)} + \ldots + 2^1 + 2^0$

$= 2^n - 1$