MA3105

Koel Das

Running time of a program

- Programs should be simple and easy to code and debug
- Efficiently use computer resources to reduce execution time
- Factors
  - Input to the program
  - Code quality
  - Nature and speed of instructions for execution
  - Time complexity of the algorithm

How do we compare algorithms?

Compare execution times?

times are specific to a particular computer !!

Count the number of statements executed?

vary with the programming language and programming style

Solution

- Express running time as a function of the input size *n* (i.e., *f(n)*).
- Compare different functions corresponding to running times.
- Such an analysis is independent of machine time, programming style, etc.

Example

- Associate a "cost" with each statement.
- Find the "total cost“ by finding the total number of times each statement is executed.

Algorithm 1

	Cost
arr[0] = 0;	c_1
arr[1] = 0;	c_1
arr[2] = 0;	c_1
...	...
arr[N-1] = 0;	c_1

$c_1+c_1+...+c_1 = c_1 \times N$

Algorithm 2

	Cost
for(i=0; i<N; i++)	c_2
arr[i] = 0;	c_1

$(N+1) \times c_2 + N \times c_1 =$
$(c_2 + c_1) \times N + c_2$

Another Example

- ***Algorithm 3*** *Cost*

```
sum = 0;                                  c1
for(i=0; i<N; i++)                        c2
  for(j=0; j<N; j++)                      c2
      sum += arr[i][j];                   c3
                                          ----------
```

$c_1 + c_2 \times (N+1) + c_2 \times N \times (N+1) + c_3 \times N^2$

Order of growth

- Most important: Order of growth within a constant multiple as $n \to \infty$

- Example:
  - How much faster will algorithm run on computer that is twice as fast?
  - How much longer does it take to solve problem of double input size?

Asymptotic Analysis

- To compare two algorithms with running times *f(n)* and *g(n),* we need a **rough measure** that characterizes **how fast each function grows.**
- *Need rate of growth*
- Compare functions in the limit, that is, **asymptotically!**

  (i.e., for large values of *n*)

Best-case, average-case, worst-case

- Worst case: $T_w(n)$- maximum over inputs of size n
- Best case: $T_w(n)$- minimum over inputs of size n
- Average case: $T_{avg}(n)$-"average" over inputs of size n
  - Hard to determine
  - NOT the average of worst and best case
  - Expected number of basic operations considered as a random variable under some assumption about the probability distribution of all possible inputs.

Example: Sequential search

ALGORITHM *SequentialSearch*$(A[0..n-1], K)$

```
//Searches for a given value in a given array by sequential search
//Input: An array A[0..n − 1] and a search key K
//Output: The index of the first element of A that matches K
//        or −1 if there are no matching elements
i ← 0
while i < n and A[i] ≠ K do
    i ← i + 1
if i < n return i
else return −1
```

- Worst case — n key comparisons
- Best case — 1 comparison
- Average case — ~n/2, assuming K is in A

Asymptotic order of growth

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 $g(n)$
  - Upper Bound

- Θ($g(n)$): class of functions $f(n)$ that grow at same rate as $g(n)$
  - Tight Bound

- Ω($g(n)$): class of functions $f(n)$ that grow at least as fast as $g(n)$
  - Lower Bound

O-notation

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:

- 10n is in $O(n^2)$

- $5n+20$ is in $O(n)$

Big-oh

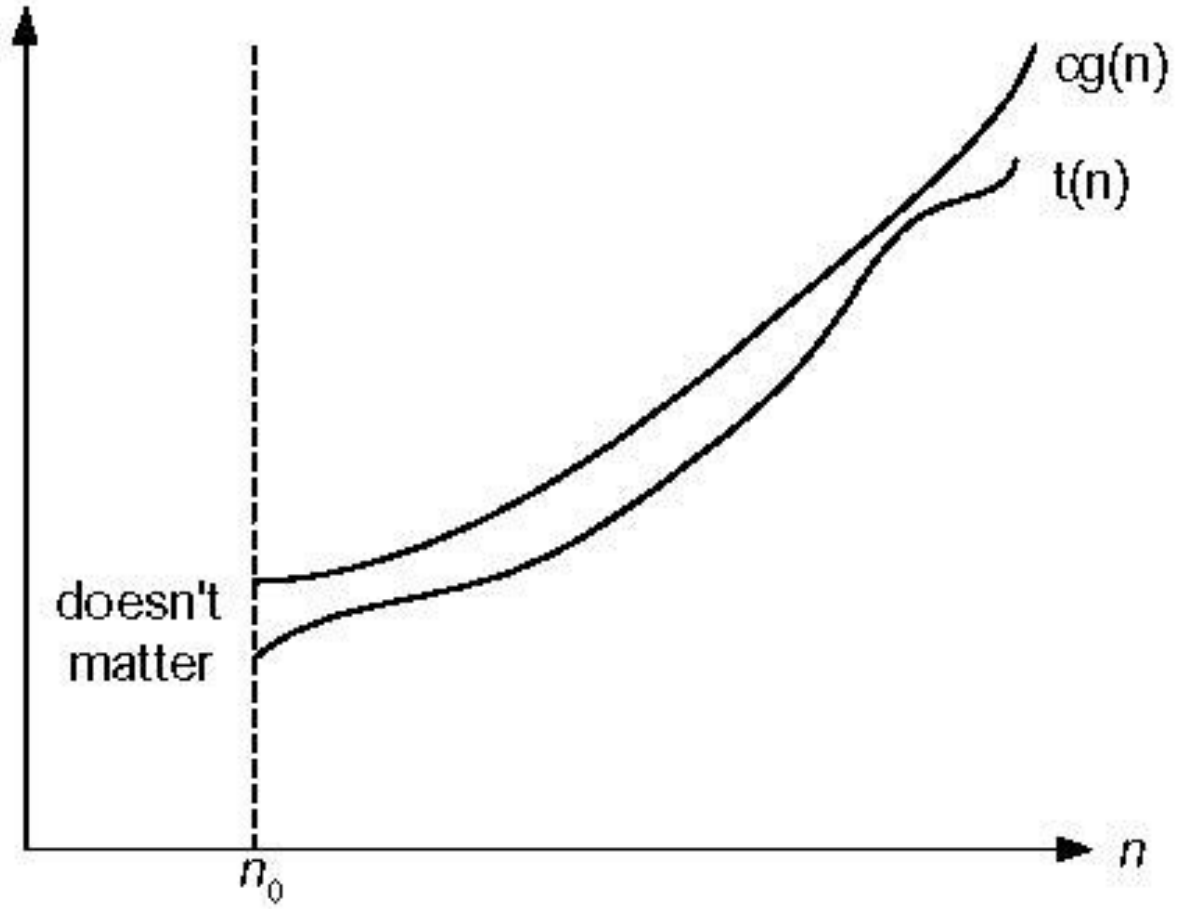
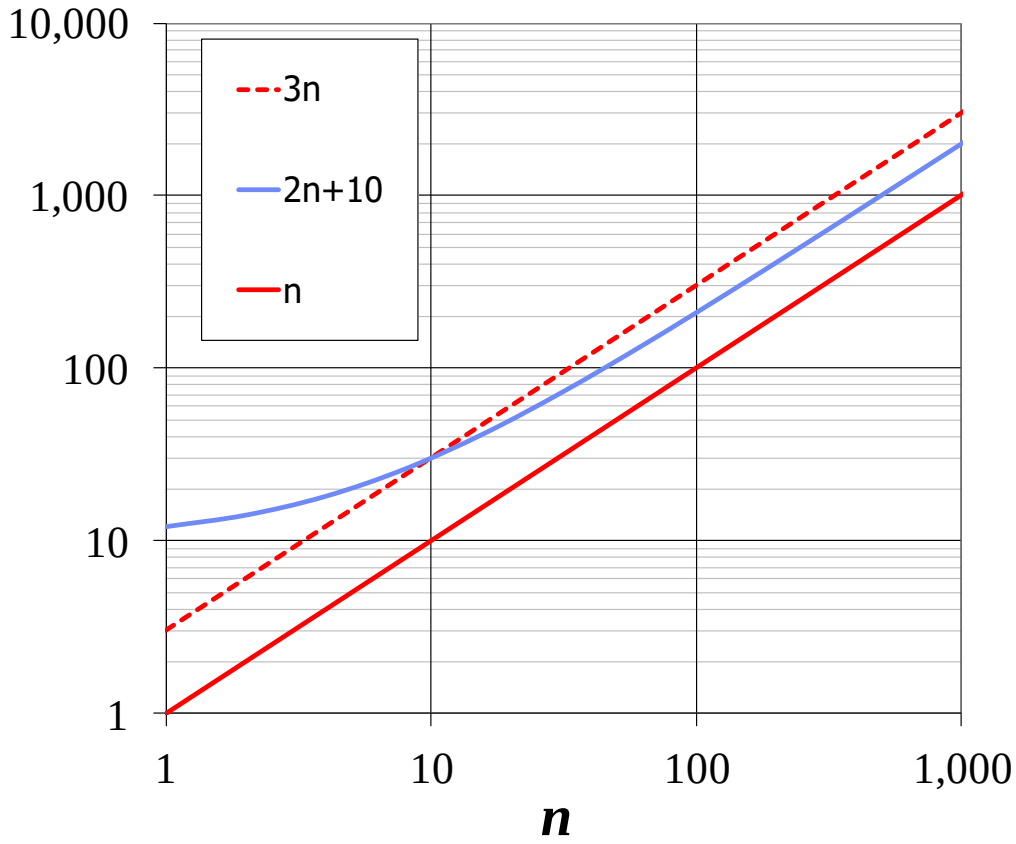


Figure 2.1 Big-oh notation: $t(n) \in O(g(n))$

Examples

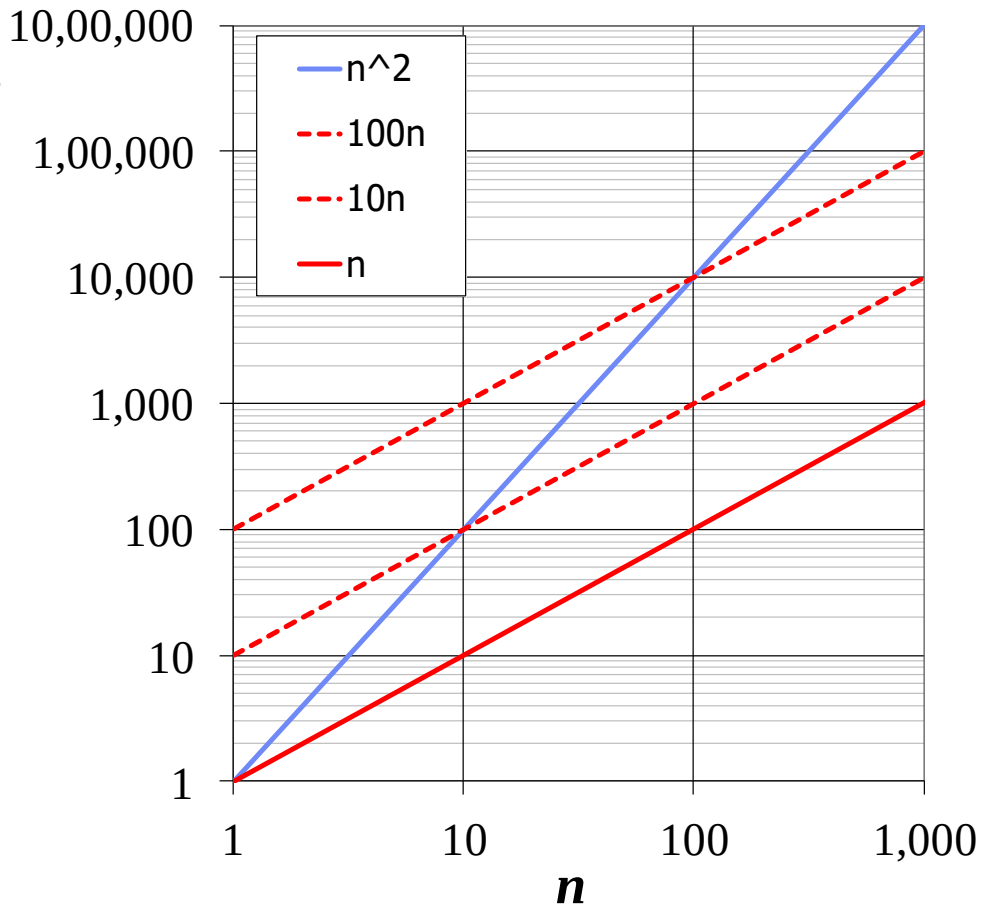
Example: $2n + 10$ is $O(n)$

- $2n + 10 \leq cn$
- $(c - 2)\, n \geq 10$
- $n \geq 10/(c - 2)$
- Pick $c = 3$ and $n_0 = 10$



Big-Oh Example

- Example: the function n^2 is not $O(n)$
  - $n^2 \leq cn$
  - $n \leq c$
  - The above inequality cannot be satisfied since c must be a constant
  - n^2 is $O(n^2)$.

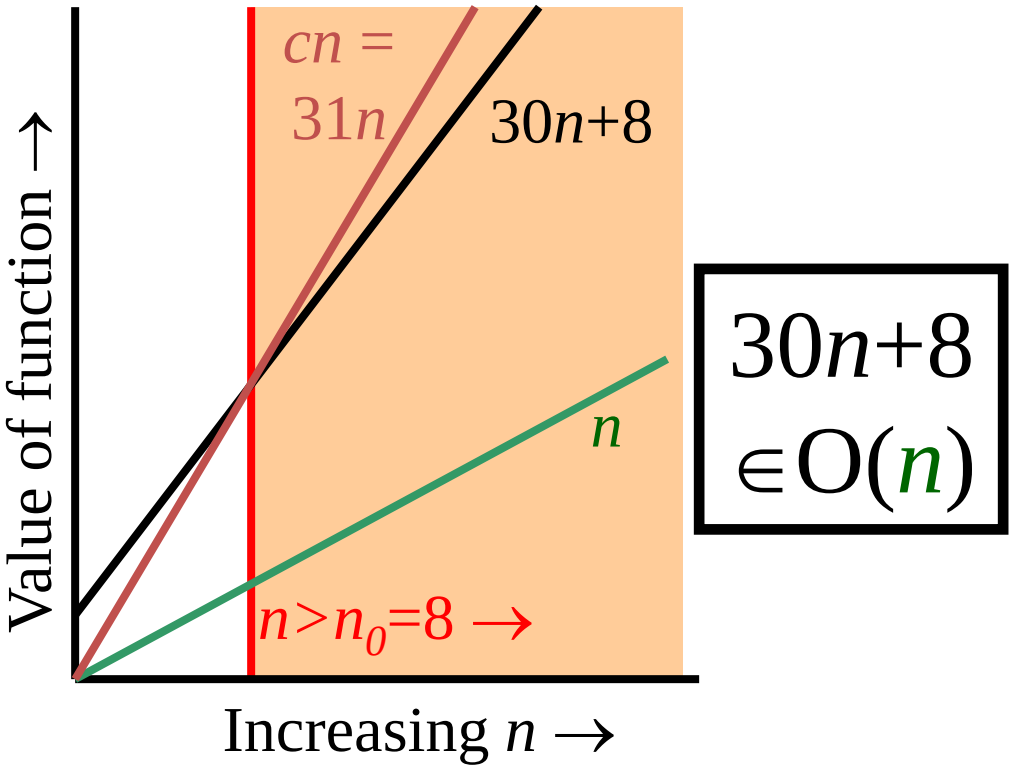


More Examples

- Show that $30n+8$ is $O(n)$.
  - Show $\exists c, n_0$: $30n+8 \leq cn$, $\forall n > n_0$.

Big-O example, graphically

- Note 30*n*+8 isn't less than *n* *anywhere* (*n*>0).
- It isn't even less than 31*n* *everywhere*.
- But it *is* less than 31*n* <u>everywhere to the right of *n*=8</u>.



Simplifying with Big-O

By definition, Big-O allows us to:

Eliminate low order terms

- $4n + 5 \Rightarrow 4n$
- $0.5\ n \log n - 2n + 7 \Rightarrow 0.5\ n \log n$

Eliminate constant coefficients

- $4n \Rightarrow n$
- $0.5\ n \log n \Rightarrow n \log n$
- $\log n^2 = 2 \log n \Rightarrow \log n$

But when might constants or low-order terms matter?

Ω-notation

Definition: f*(n)* is said to be in $\Omega(g(n))$, denoted $f(n) \in \Omega(g(n))$, if *f(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

$$f(n) \geq c\, g(n) \text{ for all } n \geq n_0$$

Examples

- $10n^2 \in \Omega(n^2)$
- $0.3n^2 - 2n \in \Omega(n^2)$

Big-omega

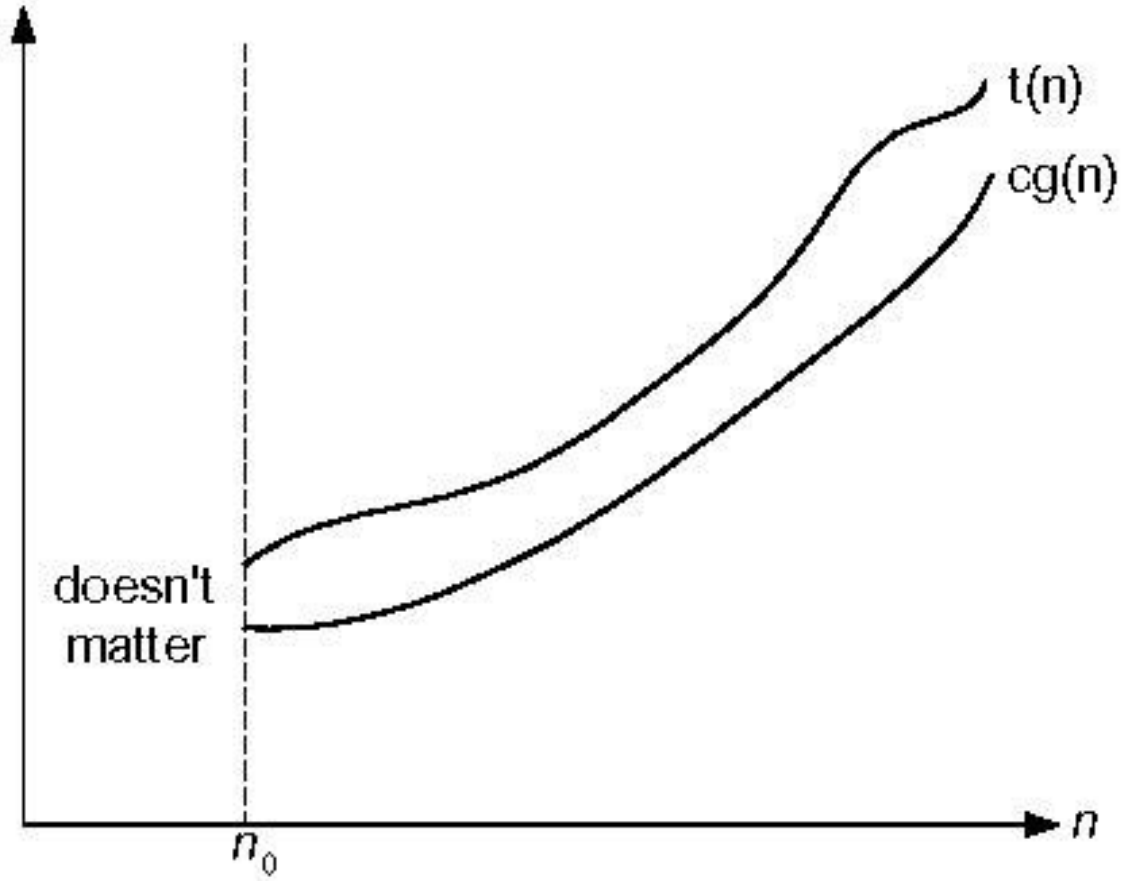


Fig. 2.2 Big-omega notation: $t(n) \in \Omega(g(n))$

Θ-notation

Definition: f*(n)* is said to be in $\Theta(g(n))$, denoted $f(n) \in \Theta(g(n))$, if *f(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

$$c_2\, g(n) \leq f(n) \leq c_1\, g(n) \text{ for all } n \geq n_0$$

Examples

$10n^2 \in \Theta(n^2)$
$0.3n^2 - 2n \in \Theta(n^2)$
$(1/2)n(n+1) \in \Theta(n^2)$

Big-theta

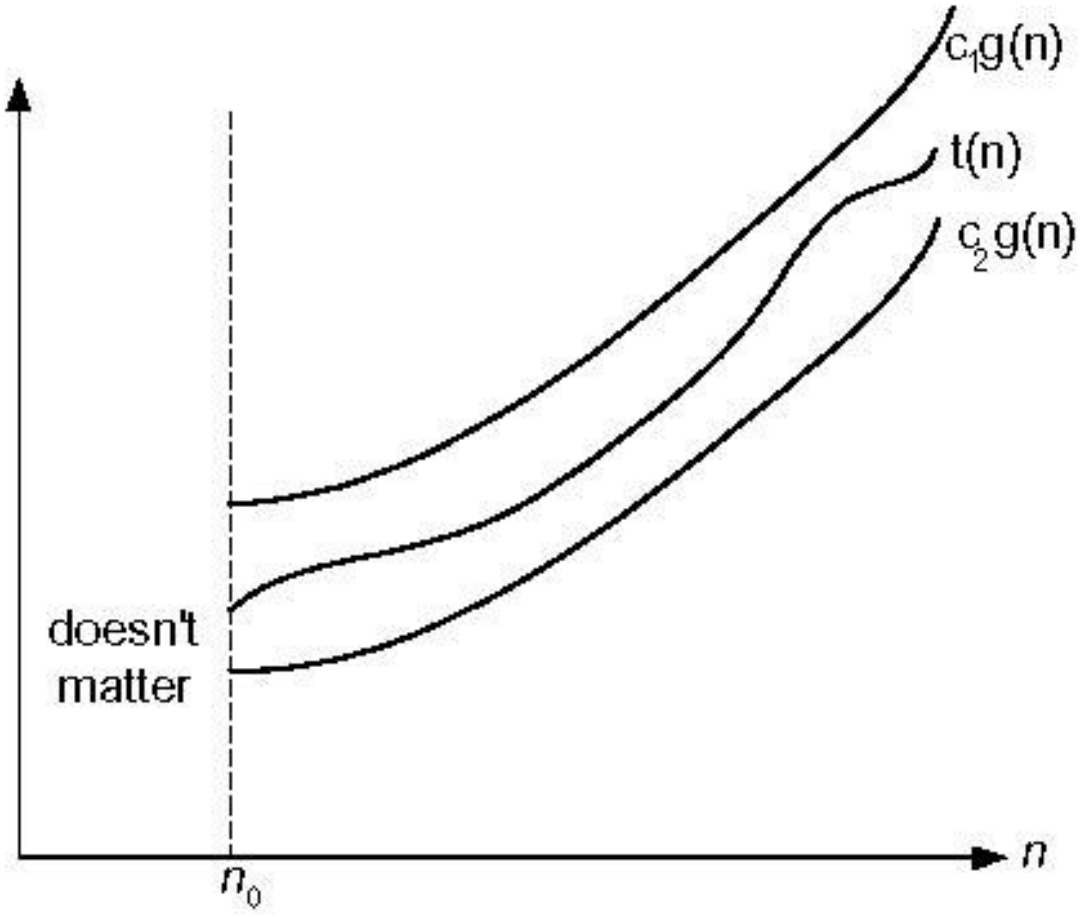
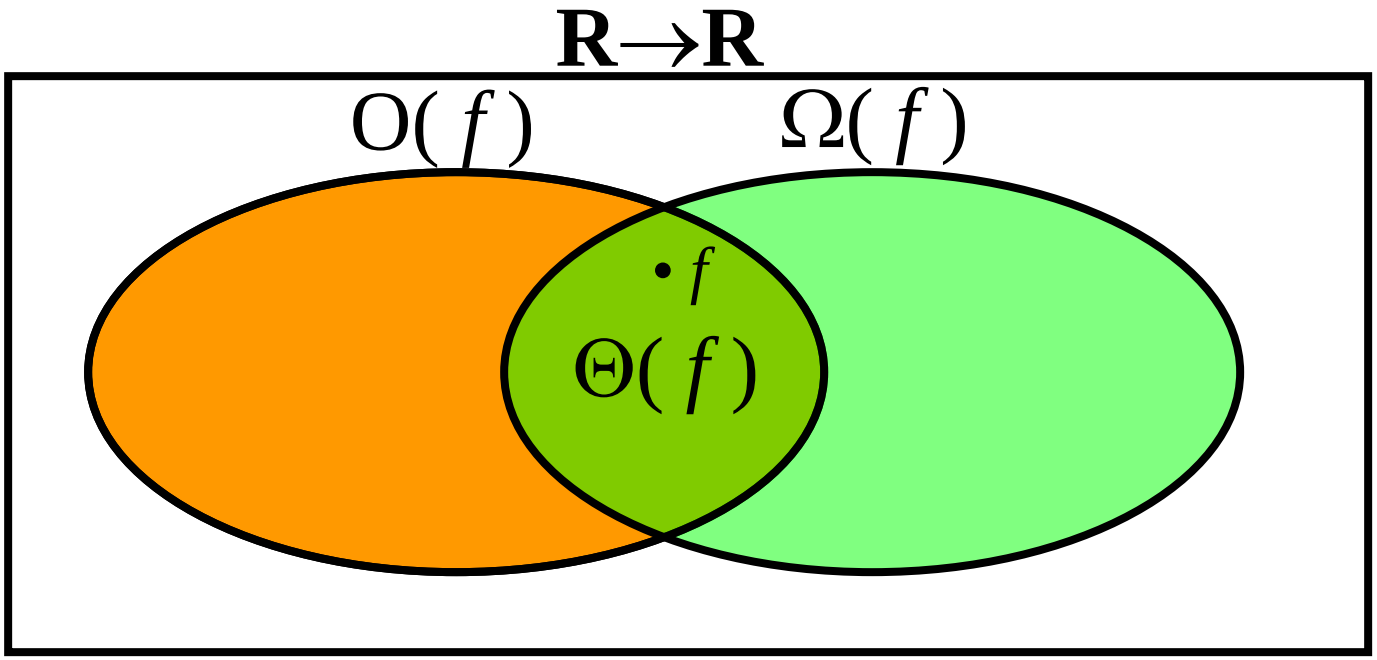


Figure 2.3 Big-theta notation: $t(n) \in \Theta(g(n))$

Relations Between Different Sets

- Subset relations between order-of-growth sets.



Examples

$n^2 + 100\,n = O(n^2)$

follows from … $(n^2 + 100\,n) \leq 2n^2$ for $n \geq 100$

$n^2 + 100\,n = \Omega(n^2)$

follows from … $(n^2 + 100\,n) \geq 1n^2$ for $n \geq 0$

$n^2 + 100\,n = \theta(n^2)$

by definition

$n \log n = O(n^2)$

$n \log n = \theta(n \log n)$

$n \log n = \Omega(n)$

Theorem

- If $t_1(n) \in O(g_1(n))$ and $t_2(n) \in O(g_2(n))$, then $t_1(n) + t_2(n) \in O(max\{g_1(n), g_2(n)\})$.
  - The analogous assertions are true for the Ω-notation and Θ-notation.

- Implication: The algorithm's overall efficiency will be determined by the part with a larger order of growth, i.e., its least efficient part.
  - For example, $5n^2 + 3nlogn \in O(n^2)$

Some properties of asymptotic order of growth

- Transitivity:
  - $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$
  - Same for O and Ω
- Reflexivity:
  - $f(n) = \Theta(f(n))$
  - Same for O and Ω
- Symmetry:
  - $f(n) = \Theta(g(n))$ if and only if $g(n) = \Theta(f(n))$
- Transpose symmetry:
  - $f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$

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

Usage

Order notation is not symmetric:

- *we can say* $2n^2 + 4n = O(n^2)$
- *… but never* $O(n^2) = 2n^2 + 4n$

Order expressions on left can produce unusual-looking, but *true*, statements:

$O(n^2) = O(n^3)$

$\Omega(n^3) = \Omega(n^2)$

Asymptotic analysis - terminology

- Special classes of algorithms:
  - ***logarithmic***: **O(log n)**
  - ***linear***: **O(n)**
  - ***quadratic***: **O(n^2)**
  - ***polynomial***: **O(n^k), k ≥ 1**
  - ***exponential***: **O(a^n), n > 1**

Polynomial vs. exponential ?

Logarithmic vs. polynomial ?

order log n < order n^{α} (α>0) < order a^n < order $n!$ < order n^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

Useful summation formulas and rules

$\Sigma_{1 \le i \le n} 1 = 1+1+\ldots+1 = n - 1 + 1$

In particular, $\Sigma_{1 \le i \le n} 1 = n - 1 + 1 = n \in \Theta(n)$

$\Sigma_{1 \le i \le n} i = 1+2+\ldots+n = n(n+1)/2 \approx n^2/2 \in \Theta(n^2)$

$\Sigma_{1 \le i \le n} i^2 = 1^2+2^2+\ldots+n^2 = n(n+1)(2n+1)/6 \approx n^3/3 \in \Theta(n^3)$

$\Sigma_{0 \le i \le n} a^i = 1 + a + \ldots + a^n = (a^{n+1} - 1)/(a - 1)$ for any $a \neq 1$

In particular, $\Sigma_{0 \le i \le n} 2^i = 2^0 + 2^1 + \ldots + 2^n = 2^{n+1} - 1 \in \Theta(2^n)$

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
```

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
```

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.