Algorithms and Data Structures

Asymptotic Analysis

Analysis of Algorithms

- An *algorithm* is a finite set of precise instructions for performing a computation or for solving a problem.
- What is the goal of analysis of algorithms?
  - To compare algorithms mainly in terms of running time but also in terms of other factors (e.g., memory requirements, programmer's effort etc.)
- What do we mean by running time analysis?
  - **Determine how running time increases as the size of the problem increases**.

Types of Analysis

- Worst case
  - Provides an upper bound on running time
  - An absolute guarantee that the algorithm would not run longer, no matter what the inputs are
- Best case
  - Provides a lower bound on running time
  - Input is the one for which the algorithm runs the fastest

$$\boxed{Lower\ Bound \leq Running\ Time \leq Upper\ Bound}$$

- Average case
  - Provides a prediction about the running time
  - Assumes that the input is random

How do we compare algorithms?

- We need to define a number of objective measures.

- Compare execution times?
  - ***Not good***: times are specific to a particular computer !!

Ideal Solution

- Express running time as a function of the input size n (i.e., $f(n)$).
- 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		***Algorithm 2***	
	Cost		**Cost**
arr[0] = 0;	c_1	for(i=0; i<N; i++)	c_2
arr[1] = 0;	c_1	arr[i] = 0;	c_1
arr[2] = 0;	c_1		
...	...		
arr[N-1] = 0;	c_1		
	-----------		-------------
	$c_1+c_1+...+c_1 = c_1 \times N$		$(N+1) \times c_2 + N \times c_1 = (c_2 + c_1) \times N + c_2$

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
                                  ------------
```

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

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.**

  – *Hint:* use *rate of growth*

Complexity

Program 1

```
x := x + 1
```

Program 2

```
FOR i := 1 to n
DO
        x := x + 1
END
```

Program 3

```
FOR i := 1 to n
DO
        FOR j := 1 to n
DO
        x := x + 1
END
END
```

Complexity

Program 1:
–statement is not contained in a loop

–Frequency count is 1

Program 2
–statement is executed n times

Program 3
–statement is executed n^2 times

Complexity

- 1, n and n^2 are said to be different in increasing orders of magnitude

- We are primarily interested in determining the order of magnitude of an algorithm

Complexity

- Let's look at an algorithm to print the n^{th} term of the Fibonnaci sequence
- 0 1 1 2 3 5 8 13 21 34 …
- $t_n = t_{n-1} + t_{n-2}$
- $t_0 = 0$
- $t_1 = 1$

Complexity

```
1   procedure fibonacci
2      read(n)
3         if n<0
4            then print(error)
5            else if n=0
6               then print(0)
7               else if n=1
8                  then print(1)
9                  else
10                    fnm2 := 0;
11                     fnm1 := 1;
12                     FOR i := 2 to n DO
13                        fn := fnm1 + fnm2;
14                        fnm2 := fnm1;
15                        fnm1 := fn
16                     end
17                     print(fn);
```

step	n<0
1	1
2	1
3	1
4	1
5	0
6	0
7	0
8	0
9	0
10	0
11	0
12	0
13	0
14	0
15	0
16	0
17	0

Complexity

```
1    procedure fibonacci {print nth term}
2       read(n)
3          if n<0
4             then print(error)
5             else if n=0
6                then print(0)
7                else if n=1
8                   then print(1)
9                   else
10                     fnm2 := 0;
11                      fnm1 := 1;
12                      FOR i := 2 to n DO
13                         fn := fnm1 + fnm2;
14                         fnm2 := fnm1;
15                         fnm1 := fn
16                      end
17                      print(fn);
```

step	n=0
1	1
2	1
3	1
4	0
5	1
6	1
7	0
8	0
9	0
10	0
11	0
12	0
13	0
14	0
15	0
16	0
17	0

Complexity

```
1     procedure fibonacci
2        read(n)
3           if n<0
4              then print(error)
5              else if n=0
6                 then print(0)
7                 else if n=1
8                    then print(1)
9                    else
10                     fnm2 := 0;
11                      fnm1 := 1;
12                      FOR i := 2 to n DO
13                         fn := fnm1 + fnm2;
14                         fnm2 := fnm1;
15                         fnm1 := fn
16                      end
17                      print(fn);
```

step	n=1
1	1
2	1
3	1
4	0
5	1
6	0
7	1
8	1
9	0
10	0
11	0
12	0
13	0
14	0
15	0
16	0
17	0

Complexity

```
1     procedure fibonacci
2        read(n)
3           if n<0
4              then print(error)
5              else if n=0
6                 then print(0)
7                 else if n=1
8                    then print(1)
9                    else
10                     fnm2 := 0;
11                      fnm1 := 1;
12                      FOR i := 2 to n DO
13                         fn := fnm1 + fnm2;
14                         fnm2 := fnm1;
15                         fnm1 := fn
16                      end
17                      print(fn);
```

step	n>1
1	1
2	1
3	1
4	0
5	1
6	0
7	1
8	0
9	1
10	1
11	1
12	n
13	n-1
14	n-1
15	n-1
16	n-1
17	1

Complexity

step	n<0	n=0	n=1	n>1
1	1	1	1	1
2	1	1	1	1
3	1	1	1	1
4	1	0	0	0
5	0	1	1	1
6	0	1	0	0
7	0	0	1	1
8	0	0	1	0
9	0	0	0	1
10	0	0	0	1
11	0	0	0	1
12	0	0	0	n
13	0	0	0	n-1
14	0	0	0	n-1
15	0	0	0	n-1
16	0	0	0	n-1
17	0	0	0	1

Complexity

- In the case where $n>1$, we have the total statement frequency of:

  $$9 + n + 4(n-1) = 5n + 5$$

- We write this as $O(n)$, ignoring the constants

- It means that the order of magnitude is proportional to n $n + 4(n-1) = 5n + 5$

Complexity

- If an algorithm has a time complexity of $O(g(n))$ it means that its execution will take no longer than a constant times $g(n)$

- n is typically the size of the data set

Complexities

- *O*(1) Constant (computing time)
- *O*(*n*) Linear (computing time)
- *O*(*n*2) Quadratic (computing time)
- *O*(*n*3) Cubic (computing time)
- *O*(2*n*) Exponential (computing time)
- *O*(log *n*) is faster than *O*(*n*) for sufficiently large *n*
- *O*(*n* log *n*) is faster than *O*(*n*2) for sufficiently large *n*

Complexity

- What about computing the complexity of a recursive algorithm?
- In general, this is more difficult
- The basic technique
  - identify a recurrence relation implicit in the recursion $T(n) = f(T(k)), k \in \{1, 2, \ldots, n\text{-}1\}$
  - solve the recurrence relation by finding an expression for $T(n)$ in term which do not involve $T(k)$

Complexity

- Example: compute factorial *n* (*n*!)

```
int factorial(int n)
{
    int factorial_value;

    factorial_value = 0;

    /* compute factorial value recursively */

    if (n <= 1) {
        factorial_value = 1;
    }
    else {
        factorial_value = n * factorial(n-1);
    }
    return (factorial_value);
}
```

Complexity

- Let the time complexity of the function be $T(n)$
- which is what we want!
- Now, let's try to analyse the algorithm

Complexity

```
                                                        n>1

int factorial(int n)
{
   int factorial_value;                                 1

   factorial_value = 0;                                 1

   if (n <= 1) {                                        1
      factorial_value = 1;                              0
   }
   else {                                               1
      factorial_value = n * factorial(n-1);             T(n-1)
   }
   return (factorial_value);                            1
}
```

Complexity

- $T(n) = 5 + T(n\text{-}1)$
- $T(n) = c + T(n\text{-}1)$
- $T(n\text{-}1) = c + T(n\text{-}2)$
- $T(n) = c + c + T(n\text{-}2)$
  $= 2c + T(n\text{-}2)$
- $T(n\text{-}2) = c + T(n\text{-}3)$
- $T(n) = 2c + c + T(n\text{-}3)$
  $= 3c + T(n\text{-}3)$
- $T(n) = ic + T(n\text{-}i)$

Complexity

- $T(n) = ic + T(n\text{-}i)$
- Finally, when $i = n\text{-}1$
- $T(n) = (n\text{-}1)c + T(n\text{-}(n\text{-}1))$
  $= (n\text{-}1)c + T(1)$
  $= (n\text{-}1)c + d$
- *Hence*, $T(n) = O(n)$

Rate of Growth

- Consider the example of buying *elephants* and *fish:*

  Cost: cost_of_elephants + cost_of_fish

  Cost ~ cost_of_elephants (approximation)

- The low order terms in a function are relatively insignificant for **large** *n*

$$n^4 + 100n^2 + 10n + 50 \quad \sim \quad n^4$$

i.e., we say that $n^4 + 100n^2 + 10n + 50$ and n^4 have the same **rate of growth**

Asymptotic Notation

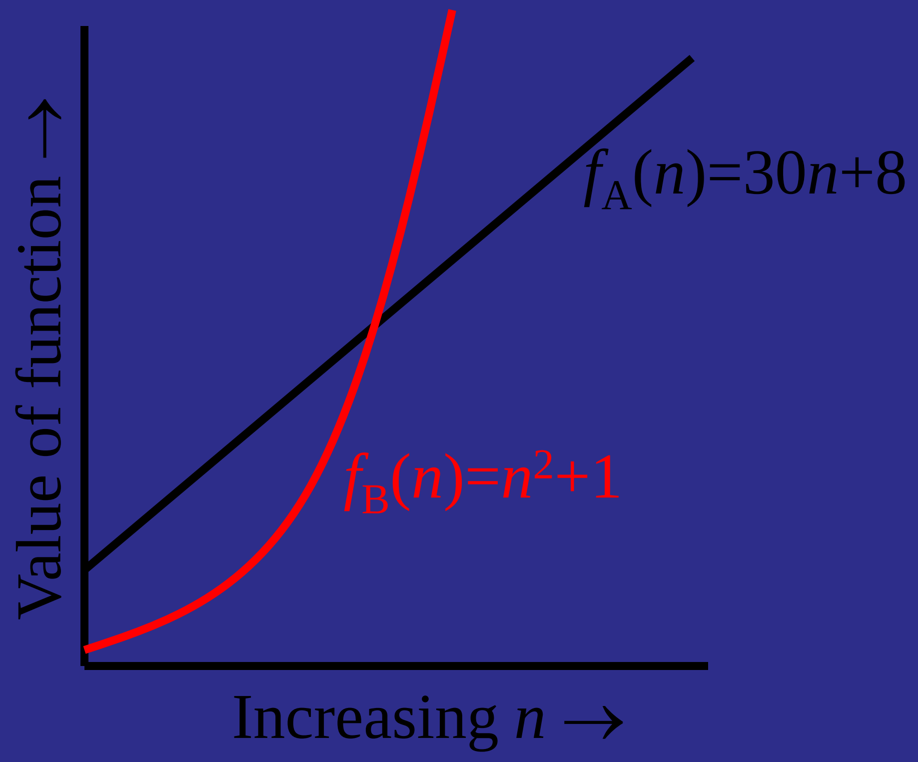
- O notation: asymptotic “less than”:
  - f(n)=O(g(n)) implies: f(n) “≤” g(n)
- Ω notation: asymptotic “greater than”:
  - f(n)= Ω (g(n)) implies: f(n) “≥” g(n)
- Θ notation: asymptotic “equality”:
  - f(n)= Θ (g(n)) implies: f(n) “=” g(n)

Big-O Notation

- We say $f_A(n)=30n+8$ is *order n*, or O(n)
  It is, at most, roughly *proportional* to n.
- $f_B(n)=n^2+1$ is *order* n^2, or $O(n^2)$. It is, at most, roughly proportional to n^2.
- In general, any $O(n^2)$ function is faster-growing than any $O(n)$ function.

Visualizing Orders of Growth

- On a graph, as you go to the right, a faster growing function eventually becomes larger...



Examples …

- $n^4 + 100n^2 + 10n + 50$ is $O(n^4)$
- $10n^3 + 2n^2$ is $O(n^3)$
- $n^3 - n^2$ is $O(n^3)$
- constants
  - 10 is $O(1)$
  - 1273 is $O(1)$

Back to Our Example

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

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

- Both algorithms are of the same order: *O(N)*

Example (cont'd)

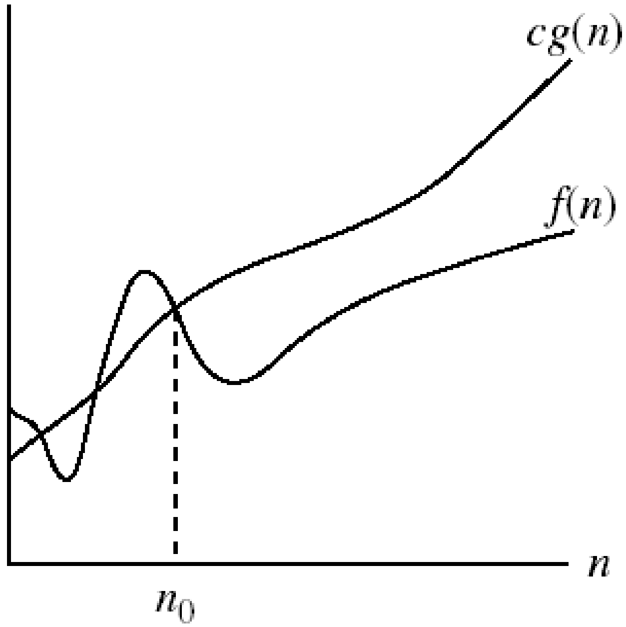
Algorithm 3	***Cost***
`sum = 0;`	c_1
`for(i=0; i<N; i++)`	c_2
`   for(j=0; j<N; j++)`	c_2
`   sum += arr[i][j];`	c_3

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

Asymptotic notations

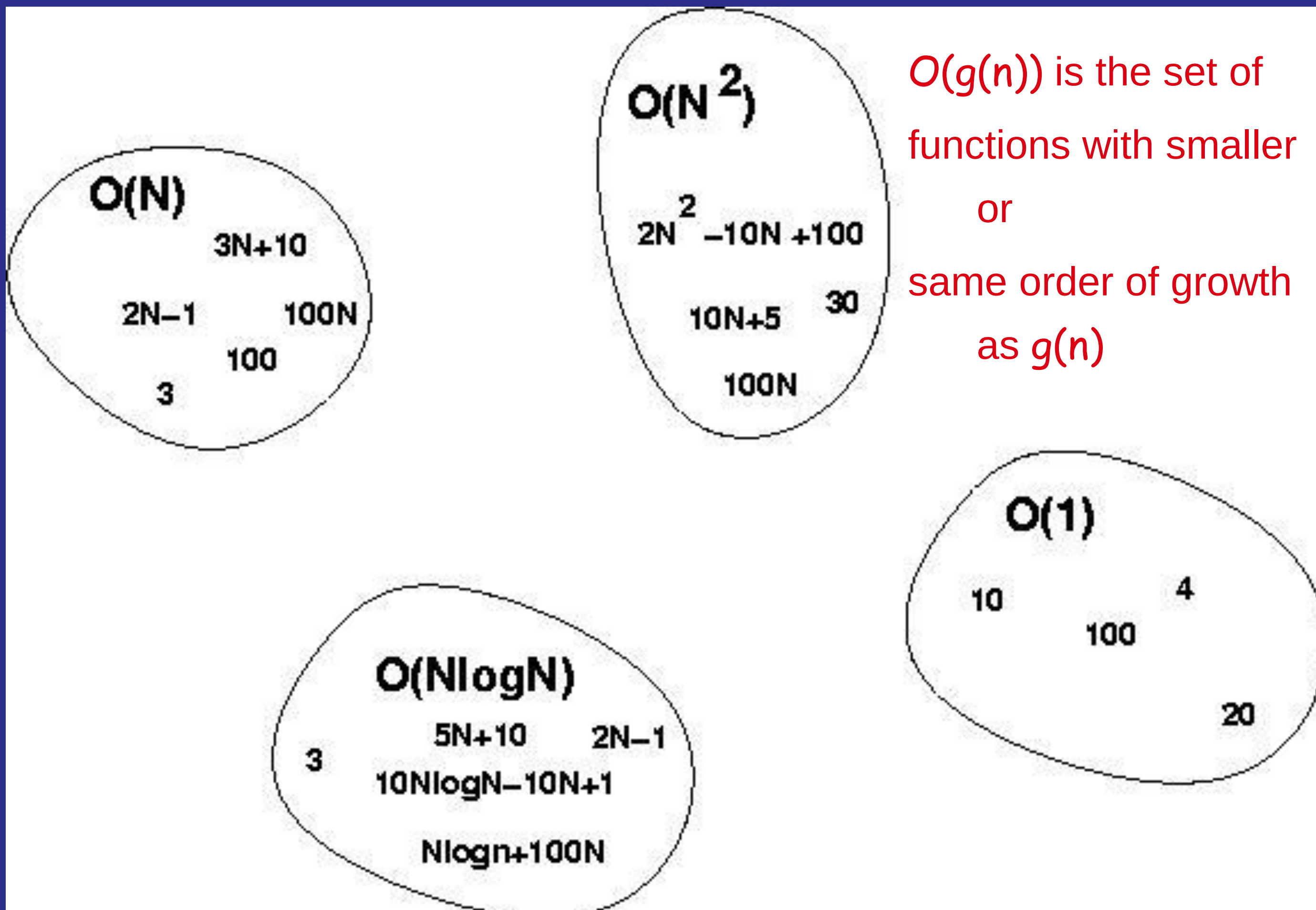
- *O-notation*

$O(g(n)) = \{f(n) :$ there exist positive constants c and n_0 such that $0 \le f(n) \le cg(n)$ for all $n \ge n_0\}$.



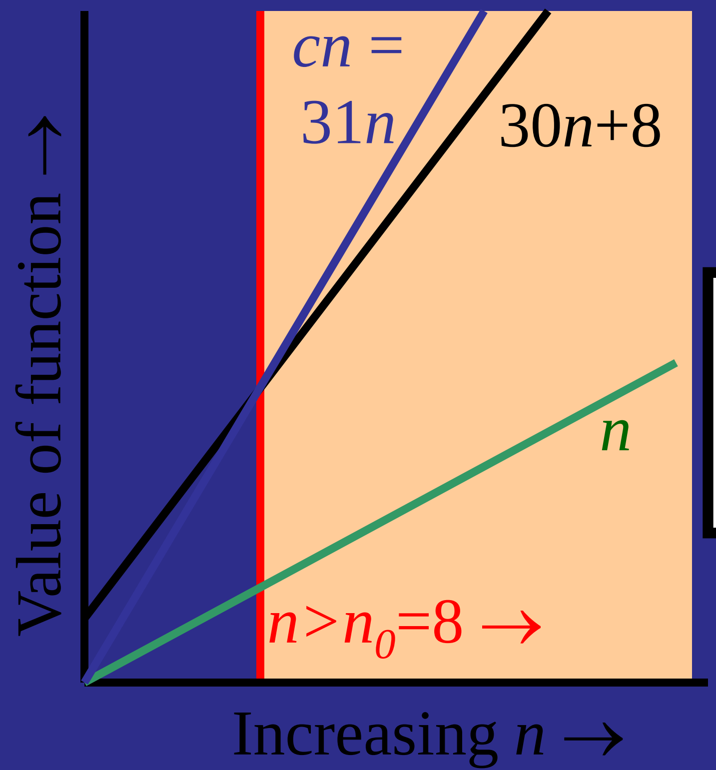
$g(n)$ is an ***asymptotic upper bound*** for $f(n)$.

Big-O Visualization



Big-O example, graphically

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

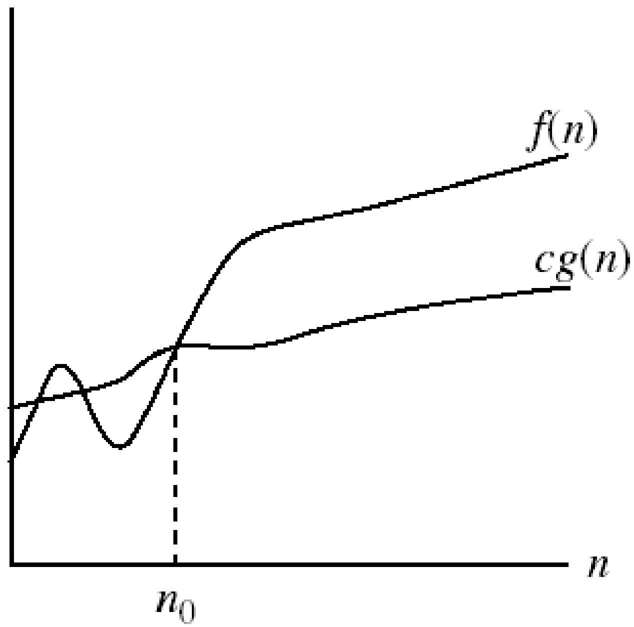


$30n+8$
$\in O(n)$

Asymptotic notations (cont.)

- Ω - *notation*

$$\Omega(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0\} .$$



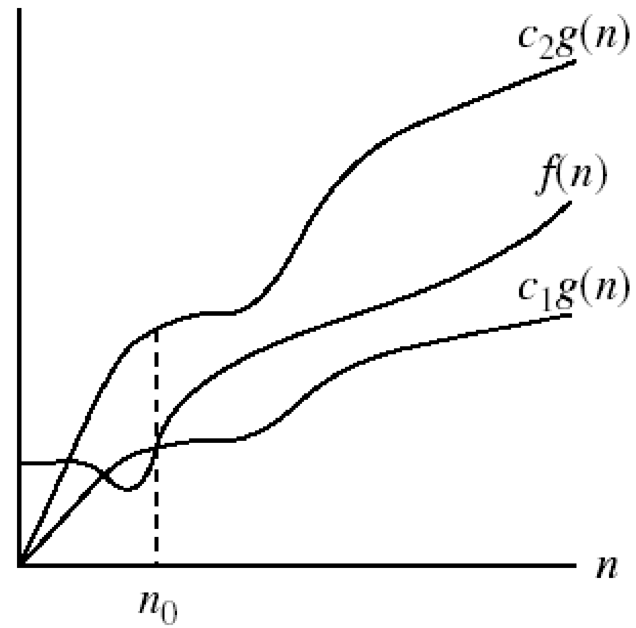
$\Omega(g(n))$ is the set of functions with larger or same order of growth as $g(n)$

$g(n)$ is an ***asymptotic lower bound*** for $f(n)$.

Asymptotic notations (cont.)

- Θ*-notation*

$$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0\} .$$

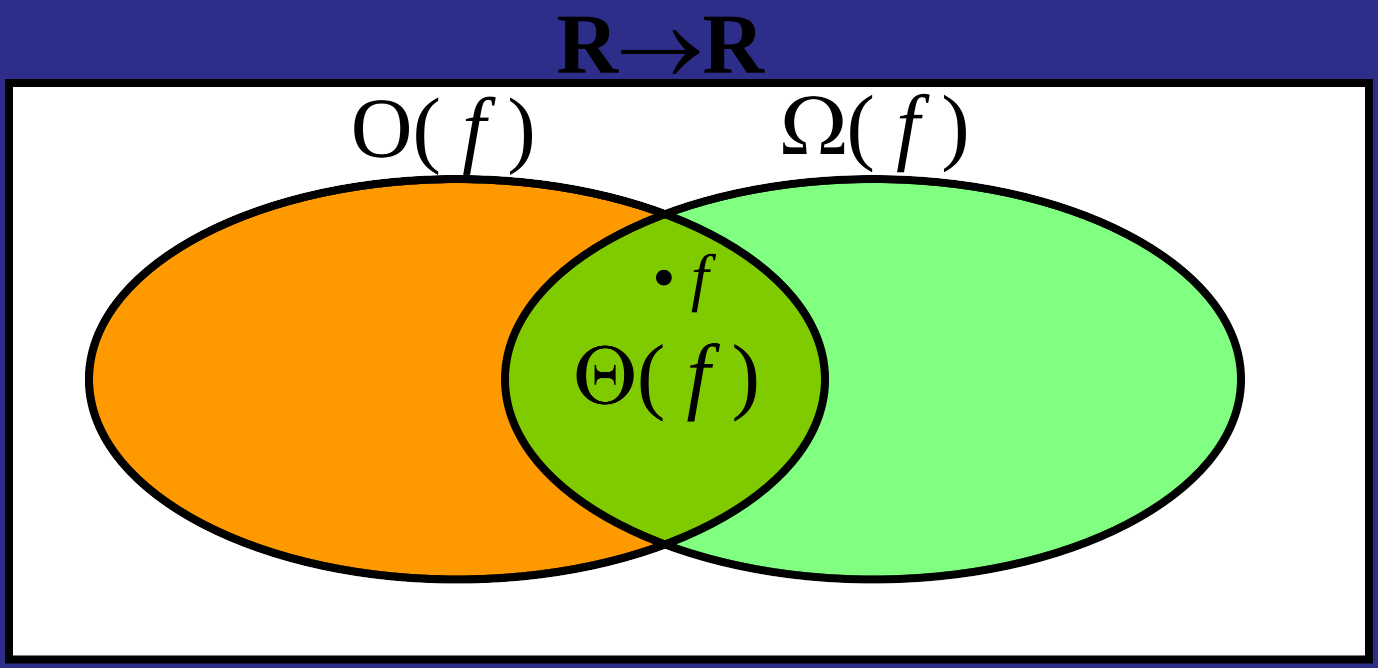


$\Theta(g(n))$ is the set of functions with the same order of growth as $g(n)$

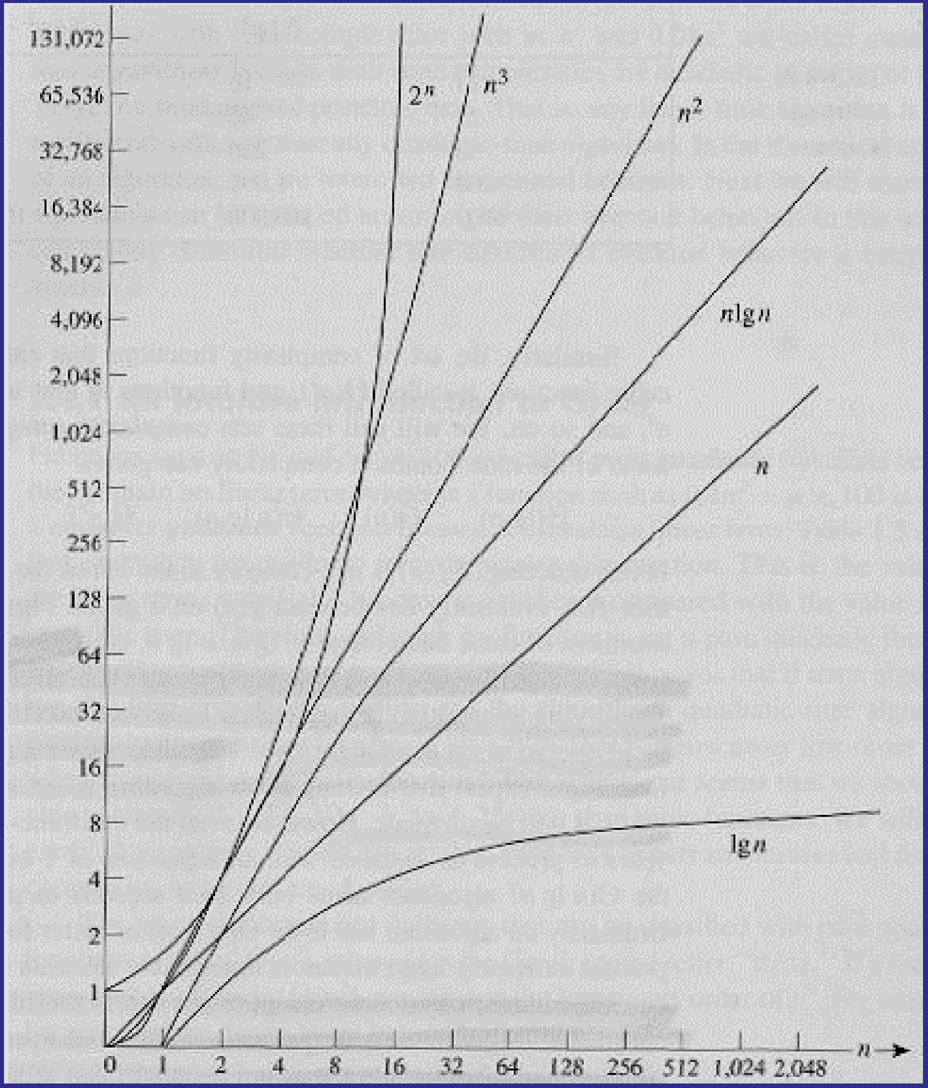
$g(n)$ is an ***asymptotically tight bound*** for $f(n)$.

Relations Between Different Sets

- Subset relations between order-of-growth sets.



Common orders of magnitude



Common orders of magnitude

Table 1.4 Execution times for algorithms with the given time complexities

n	$f(n) = \lg n$	$f(n) = n$	$f(n) = n \lg n$	$f(n) = n^2$	$f(n) = n^3$	$f(n) = 2^n$
10	0.003 µs*	0.01 µs	0.033 µs	0.1 µs	1 µs	1 µs
20	0.004 µs	0.02 µs	0.086 µs	0.4 µs	8 µs	1 ms†
30	0.005 µs	0.03 µs	0.147 µs	0.9 µs	27 µs	1 s
40	0.005 µs	0.04 µs	0.213 µs	1.6 µs	64 µs	18.3 min
50	0.006 µs	0.05 µs	0.282 µs	2.5 µs	125 µs	13 days
10^2	0.007 µs	0.10 µs	0.664 µs	10 µs	1 ms	4×10^{13} years
10^3	0.010 µs	1.00 µs	9.966 µs	1 ms	1 s	
10^4	0.013 µs	10 µs	130 µs	100 ms	16.7 min	
10^5	0.017 µs	0.10 ms	1.67 ms	10 s	11.6 days	
10^6	0.020 µs	1 ms	19.93 ms	16.7 min	31.7 years	
10^7	0.023 µs	0.01 s	0.23 s	1.16 days	31,709 years	
10^8	0.027 µs	0.10 s	2.66 s	115.7 days	3.17×10^7 years	
10^9	0.030 µs	1 s	29.90 s	31.7 years		

*1 µs = 10^{-6} second.
†1 ms = 10^{-3} second.