Algorithm Design & Analysis

Wei-Che Chien(簡暐哲)
Department of Computer Science and Information Engineering
National Dong Hwa University

Examples of Problems

- **Problem** 1.1: Sort a list S of n numbers in nondecreasing order. (sorting problem)
  - The solution is a sorted list of all elements in S.
  - The parameters are S and n.
  - S=[10,7,11,5,13,8], n=6 is an instance. The solution is [5, 7, 8, 10, 11, 13].
- **Problem** 1.2: Determine whether the number x is in the list S of n numbers. (search problem)
  - The solution is YES(TRUE) or NO(FALSE).
  - The parameters are x, S and n.
  - x=5, S=[10,7,11,5,13,8], n=6 is an instance. The solution is TRUE.

Algorithm and Representation

- An algorithm for Problem 1.2 can be as follows:
  1. Starting with the first item in S.
  2. Compare x with each item in S in sequence.
  3. If x is found then the answer is YES.
  4. If S is exhausted w/o finding x, the answer is NO.
- English representation of an algorithm is verbose and not precise.
- We use C++-like pseudocode to describe an algorithm. (next slide)
- Do remember that algorithms are language independent !!

Sequential Search

```
void seqsearch (int n, const keytype S[],
               keytype x, index& location)
{
  location = 1;
  while (location <=n && S[location] != x)
     location ++;
  if (location > n)
     location =0;
}
```

x=5, S=[10,7,11,5,13,8], n=6

Binary Search

```
void binsearch(int n, const keytype S[], keytype x, index& location)
{
    index low, high, mid;

    low = 1;  high = n;  location = 0;
    while (low <= high &&  location == 0) {
        mid = ⌊(low + high)/2⌋;
        if (x == S[mid])
            location = mid;
        else if (x < S[mid])
            high = mid - 1;
        else
            low = mid + 1;
    }
}
```

x=5, S=[10,7,11,5,13,8], n=6

x=5, S=[5, 7, 8, 10,11,13] n=6

Sequential vs Binary Search

- For a search problem with parameters x, S, n, the worst case occurs when $x \notin S$.
- Sequential Search: n operations
- Binary Search: lg n + 1 ($\log_2 n$+1) operations

Array Size	Number of Comparisons by Sequential Search	Number of Comparisons by Binary Search
128	128	8
1,024	1,024	11
1,048,576	1,048,576	21
4,294,967,296	4,294,967,296	33

Sequential vs Binary Search

- Even with a computer that can complete one pass through the while loop in a nanosecond, Sequential Search would take 4 seconds while Binary Search would be instantaneous.
- Different algorithms in solving the same problem may differ significantly on performance.

- 4 seconds execution time seems tolerable.
- Let’s take the computation of Fibonacci sequence as another example.

Fibonacci Sequence

- Consider another problem of computing the nth term of the Fibonacci sequence:

  $f_0 = 0$

  $f_1 = 1$

  $f_n = f_{n-1} + f_{n-2}$ for $n >= 2$

- It is only nature to use a recursive solution.
- We also use an iterative version as comparison.

nth Fibonacci Term (Recursive)

- The definition of Fibonacci sequence naturally leads to a recursive algorithm.

```
int fib(int n)
{
    if (n <=  1)
        return n;
    else
        return fib(n - 1) + fib(n-2);
}
```

n	$f(n)$	Number of Terms computed
0	0	1
1	1	1
2	1	3
3	2	5
4	3	9
5	5	15
6	8	25

nth Fibonacci Term (Iterative)

- It is also common to have an iterative solution.

```
int fib2 (int n)
{
  index i;
  int f[0 .. n];
  f[0] = 0;
  if (n > 0) {
    f[1] = 1;
    for (i=2; i<=n; i++)
      f[i] = f[i-1] + f[i-2];
  }
  return f[n];
}
```

Fibonacci: Recursive vs Iterative

- In Fibonacci, the most important factor of performance is the number of Fibonacci terms we need to calculate.
- It is easy to see that to calculate the nth Fibonacci term:
  - Recursive algorithm calculates $2^{n/2}$ terms
  - Iterative algorithm calculates n+1 terms
- It is a good exercise to figure out why it is the case.
- But how significant is the difference between them?

Recursive vs Iterative Fibonacci

- This time it is indeed significantly different !!

n	$n+1$	$2^{n/2}$	Execution Time Using Iterative	Lower Bound on Execution Time Using Recursive
40	41	1,048,576	41 ns*	1048 μs†
60	61	1.1×10^9	61 ns	1 s
80	81	1.1×10^{12}	81 ns	18 min
100	101	1.1×10^{15}	101 ns	13 days
120	121	1.2×10^{18}	121 ns	36 years
160	161	1.2×10^{24}	161 ns	3.8×10^7 years
200	201	1.3×10^{30}	201 ns	4×10^{13} years

*1 ns = 10^{-9} second.

†1 μs = 10^{-6} second.

Algorithms are NOT created equal

- From examples above, we can see that the performance of different algorithms for solving the same problem can vary significantly.
- It is essential to choose the best algorithm for the problem at hand.
- After designing an algorithm, we need to be able to analyze its performance, especially in terms of problem size.
- We also need a standard way to compare the efficiency of algorithms.

Proper Algorithm Analysis

- The computer run time of an algorithm(program) depends on CPUs, programing languages, systems, etc. which is hard to compare.
- We want a measure of algorithm efficiency independent of computers, languages, programmers, and tiny mingy details such as index increments, pointer setting, etc.
- The measure must also be general enough to be used to compare the relative efficiency between algorithms.

Complexity Analysis Approach

- In general, the run time increases with the input size.
- The total running time is proportional to how many time some basic operation is done.
- We analyze algorithm efficiency by determining the number of time some basic operation is done as a function of the input size.
- This is independent of the CPU, language, … etc. and can be easily compared.

How to Analyze?

- Determine the most important operation(or group of operations) as the Basic Operation.
- Count the number of times the basic operation executes for each value of the input size.
- For an algorithm, if the basic operation is always done the same number of times for a given input size n, it is a function of n.
- Define T(n) to be such a function.
- T(n) is called the every-case time complexity of the algorithm.

Every-Case Example: Array Sum

- **Problem**: Add all the numbers in the array *S* of *n* numbers.

```
number sum (int n, const number S[])
{
   index i;                         0
   number result;                   0
   result = 0;                      1
   for (i = 1; i <= n; i++)
      result = result + S[i];       n
   return result;                   1
}
```

Every-Case Analysis of `sum`

- **Baisc operation**: addition of an item in S
- **Input size**: n, the number of items in S
- **Analysis**:
  - For an instance of size n, the for loop is executed n times. Therefore the basic operation is done n times.
  - We have $T(n) = n$.

- Note that this is true for any instance of the problem.
- This is the every-case time complexity of `sum`.

Every-Case Example: Exchange Sort

- **Problem**: Sort *S* of *n* keys in nondecreasing order

```
void exchangesort (int n, keytype S[])
{
  index i, j;
  for (i=1;i<= n; i++)
    for (j=i+1; j<= n; j++)
      if (S[j] < S[i])
        exchange S[i] and S[j];
}
```

Every-Case Analysis of `exchangesort`

- **Baisc operation**: comparison of *S[i]* and *S[j]*.
- **Input size**: *n*, the number of items in *S*.
- **Analysis**:
  - For i=1, the inner loop executes n-1 times. For i=2, the inner loop execute n-2 times.
  - Therefore the total number of times is:

    $T(n) = (n-1)+(n-2)+\ldots+1 = n(n-1)/2$.
- This is also the every-case time complexity of `exchangesort`.

Complexity Analysis – Large n

- Every Case
  - T(n): every-case time complexity
- *Worst Case
  - W(n): worst-case time complexity
- Average Case
  - A(n): average-case time complexity
- Best Case
  - B(n): best-case time complexity

Worst-Case Analysis

- In `seqsearch` discussed earlier, the basic operation (what is it?) is not done the same number of times for all instances of size n.
- The algorithm does not have a every-case time complexity.
- We can still measure it by considering the maximum number of times the basic op is done.
- Define W(n) to be the maximum number of times the algorithm will ever do its basic operation for instances of size n.
- W(n) is called the worst-case time complexity.
- If T(n) exists, then W(n)=T(n). (Why?)

Worst-Case Analysis of `seqsearch`

- **Baisc operation**: comparison of x with an *S[i]*.
- **Input size**: *n*, the number of items in *S*.
- **Analysis**:
  - The basic operation is done at most n times when x is not in S.
  - Therefore the total number of times is:

    W(n) = n.
- This is the worst-case time complexity of `seqsearch`.

x=8, S=[10,7,11,5,13,8], n=6

Best-Case Complexity

- For a given algorithm, B(n) is defined as the minimum number of times the basic operation is done.
- B(n) is called the best-case time complexity.
- If T(n) exists, B(n)=T(n). (Why?)
- **Example**: the best-case time complexity of `seqsearch` is B(n) = 1. (Why?)
- B(n) is not as useful as W(n) and A(n). (Why?)

Best-Case Complexity

- **Baisc operation**: comparison of x with an *S[i]*.
- **Input size**: *n*, the number of items in *S*.
- **Analysis**:
  - The basic operation is done at most n times when x is not in S.
  - Therefore the total number of times is:

    B(n) = 1.
- This is the Borst-case time complexity of `seqsearch`.

x=10, S=[10,7,11,5,13,8], n=6

Average-Case Complexity

- Worst-case analysis seems too conservative and cannot reflect the performance on the average.
- A(n) is defined as the average (expected value) number of times the basic operation is done.
- A(n) is called the average-case time complexity of the algorithm.
- If T(n) exists, then A(n)=T(n). (Why?)
- To computer A(n), we need to assign probabilities to all possible inputs of size n.
- Average-case is usually harder to analyze.

Average-Case Analysis of `seqsearch`

- First assume that x is in S and items in S are all distinct.
- The probability of x in any slot of S is 1/n.
- If x == S[k], the basic operation is done k times.
- The average time complexity is:

$$A(n) = \sum_{k=1}^{n}\left(k \times \frac{1}{n}\right) = \frac{1}{n} \times \sum_{k=1}^{n} k = \frac{1}{n} \times \frac{n(n+1)}{2} = \frac{n+1}{2}$$

- Then we consider the case in which x may not be in S.

Average-Case Analysis of `seqsearch`

- We assume the probability *p* for x $\in$ S. Then, the probability of x in any slot k is *p/n*.
- The probability of x $\notin$ S is *1 − p*.
- If x == S[k], the basic operation is done k times.
- If x $\notin$ S, the basic operation is done n times.
- The average time complexity is:

$$A(n) = \sum_{k=1}^{n} \left(k \times \frac{p}{n} \right) + n(1-p)$$

$$= \frac{p}{n} \times \frac{n(n+1)}{2} + n(1-p) = n\left(1 - \frac{p}{2}\right) + \frac{p}{2}$$

Memory Complexity

- Similar to time complexity, we can also analyze how efficient an algorithm is in terms of memory.
- This is known as memory complexity.
- Memory complexity can be analyzed in similar ways as time complexity analysis.
- We will focus more on time complexity analysis.
- For embedded systems, memory complexity is of equal importance as time complexity.

Order(量級): Motivation

```
For i =1 to n do
    a=(b+c)/d+e;
```

T(n)?

```
For i =1 to n do
    x=(b+c)
    y=x/d;
    a=y+e;
```

T(n)?

T(n)= n

T(n)= 3n

n

1 (constant) $< \log n < n < n \log n < n^2 < n^3 < 2^n < 3^n \leq$ **n!**

Order: Motivation

- Once we have the complexity analysis of algorithms, we can compare.
- For example, algorithm A1 with $T1(n)=n$ (linear-time) is more efficient than algorithm A2 with $T2(n)=n^2$ (quadratic-time).
- How about $T1(n)=0.01n^2$ and $T2(n)=100n$?
- A1 will be more efficient if $0.01n^2 > 100n$ which can be simplified as $n > 10{,}000$.
- Any linear-time algorithm is eventually more efficient than any quadratic-time algorithm.
- Order help us characterize the eventual behavior.

Intuitive Introduction of Order

- Back to $0.01n^2$ and$100n$, the constants are much less important than the n^2 and n terms.
- When *n* gets larger, the quadratic term eventually dominates.

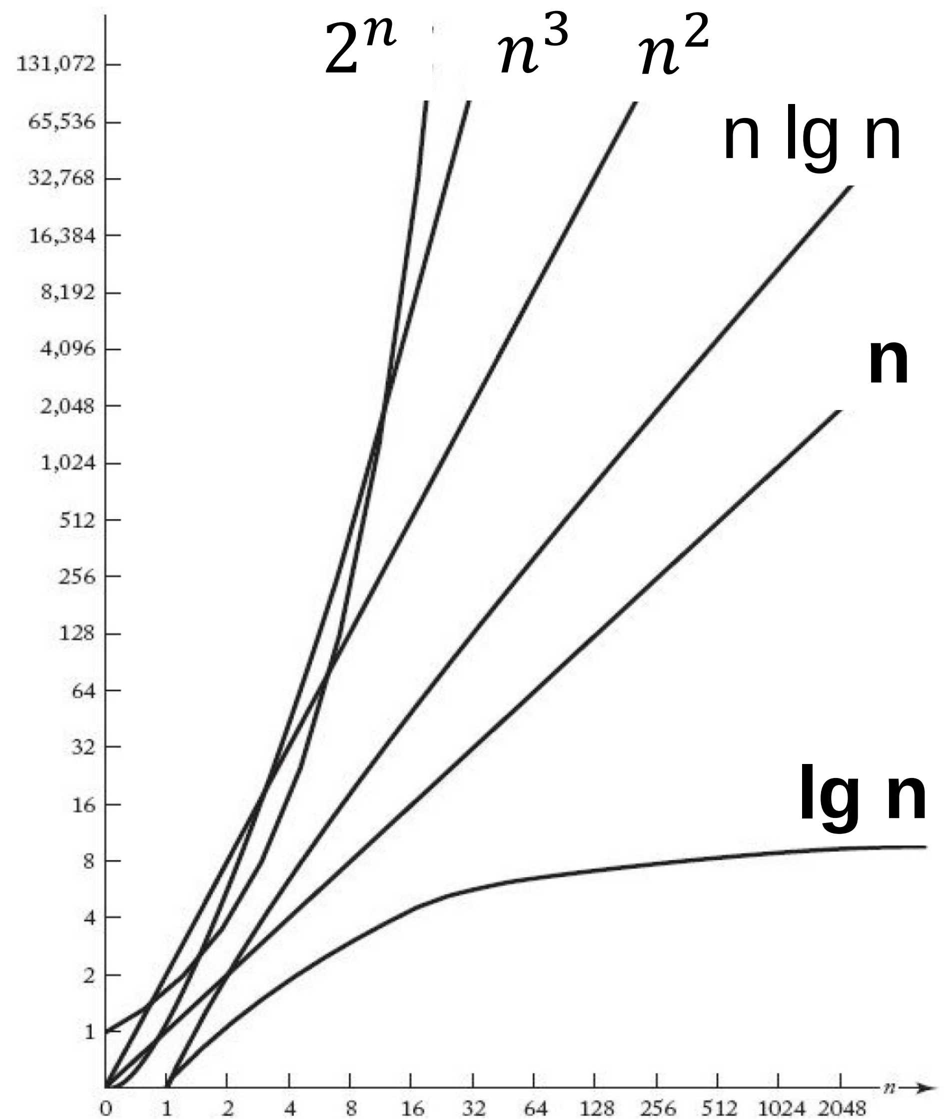
n	$0.1n^2$	$0.1n^2 + n + 100$
10	10	120
20	40	160
50	250	400
100	1,000	1,200
1,000	100,000	101,100

Intuitive Introduction of Order

- In general, the highest order term eventually dominates.
- The set of complexity functions with n^2 as the highest term is called $\theta(n^2)$ or $\Theta(n^2)$
- If a function is a member of $\theta(n^2)$, the order of the function is n^2.
- Example: $g(n) = 5n^2 + 100n + 20 \in \theta(n^2)$.
- When an algorithm's complexity is in $\theta(n^2)$, it is called a quadratic-time algorithm or $\theta(n^2)$ algorithm.

Complexity Categories(Classes)

- Similarly, we can have $\theta(n^3)$ or cubic-time algorithms, and so on.
- These are known as complexity categories.
- Common categories: $\theta(\lg n)$, $\theta(n)$, $\theta(n \lg n)$, $\theta(n^2)$, $\theta(n^3)$, $\theta(2^n)$
- The growth rates determine the superiority.



Execution Time of Different Classes

- Growth rates characterize the eventual behavior.

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	0.10 μs	1.0 μs	1 μs
20	0.004 μs	0.02 μs	0.086 μs	0.40 μs	8.0 μs	1 ms†
30	0.005 μs	0.03 μs	0.147 μs	0.90 μs	27.0 μs	1 s
40	0.005 μs	0.04 μs	0.213 μs	1.60 μs	64.0 μs	18.3 min
50	0.006 μs	0.05 μs	0.282 μs	2.50 μs	125.0 μs	13 days
10^2	0.007 μs	0.10 μs	0.664 μs	10.00 μs	1.0 ms	4×10^{13} years
10^3	0.010 μs	1.00 μs	9.966 μs	1.00 ms	1.0 s	
10^4	0.013 μs	10.00 μs	130.000 μs	100.00 ms	16.7 min	
10^5	0.017 μs	0.10 ms	1.670 ms	10.00 s	11.6 days	
10^6	0.020 μs	1.00 ms	19.930 ms	16.70 min	31.7 years	
10^7	0.023 μs	0.01 s	2.660 s	1.16 days	31,709 years	
10^8	0.027 μs	0.10 s	2.660 s	115.70 days	3.17×10^7 years	
10^9	0.030 μs	$\boxed{1.00 \text{ s}}$	29.900 s	31.70 years		

*1 $\mu s = 10^{-6}$ second.

†1 ms $= 10^{-3}$ second.

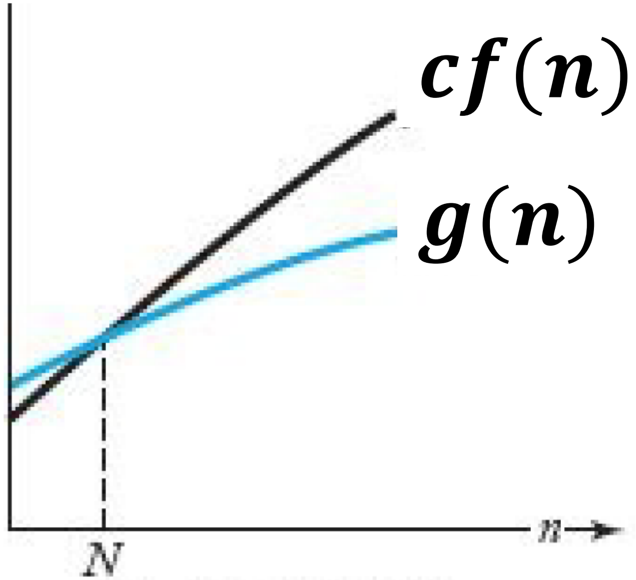
*: With the assumption of 1 ns (10^{-9}) per basic function.

Big O

- We need a formal concept to characterize complexity classes.
- Given a complexity function $f(n)$, $O(f(n))$ is the set of complexity function $g(n)$ s.t. there exists some **positive** real constant **c** and some nonnegative integer **N** such that for all $\mathbf{n \geq N}$, $\mathbf{g(n) \leq c \times f(n)}$.
- Note that c and N may not be unique.
- If $g(n) \in O(f(n))$, we say that g(n) is big O of f(n).
- The condition "for all $n \geq N$" is exactly to focus on eventual behavior.
- The complexity class of g(n) is f(n).

Big O Illustrated

- Illustrating Big O.



(a) $g(n) \in O(f(n))$

g(n) = O(f(n)) ↔ ∃ c, N> 0, ∋ g(n) ≤ cf(n), ∀ n≥ N

g(n) = O(f(n)) if and only if there are two positive numbers c and N, and g(n)< cf(n), for all n≥ N

Big O Illustrated

- **Example**: Given $g(n)=n^2+10n$,

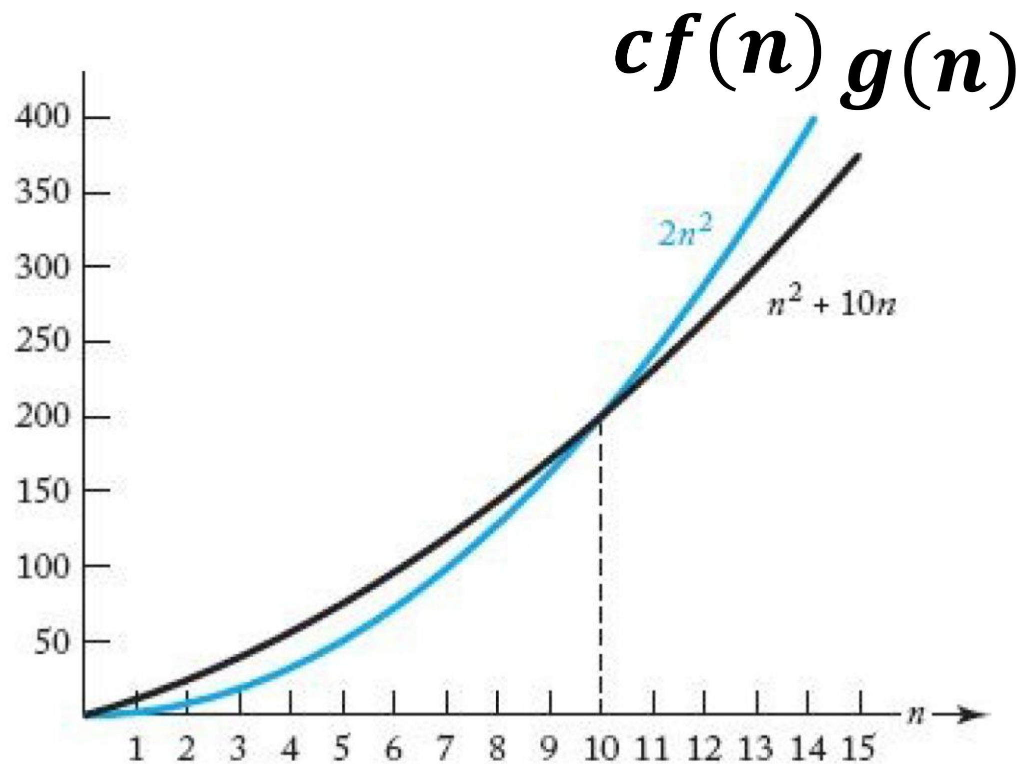
for n >= 10,

$n^2+10n <= 2n^2$.

Therefore

$g(n) \in O(n^2)$

with c=2 and N=10.



$g(n) = O(f(n)) \leftrightarrow \exists$ c, N> 0, $\ni$ $g(n) \leq cf(n)$, $\forall$ $n \geq N$

Big O Illustrated

- **Example**:

  g(n)= $5n^2$+3 n+2, g(n) = O(f(n)) = O(n^2)

  c=6

  $5n^2$+3 n+2 $\leq 6\ n^2$ => 3 n + 2 < n^2 => N=4

g(n) = O(f(n)) ↔ ∃ c, N> 0, ∋ g(n) ≤ cf(n), ∀ n≥ N

Asymptotic Upper Bound

- If $g(n) \in O(f(n))$, then eventually g(n) lies beneath f(n) and stay there.
- If g(n) is the time complexity of an algorithm A, this means that eventually the runtime of A will be at least as fast as f(n).
- For comparison, the time complexity of A is eventually at least as good as f(n).
- "Big O" describes the asymptotic behavior of a function. It puts an asymptotic upper bound on a function.

More Examples of Big O

- $5n^2 \in O(n^2)$ since for $n \geq 0$,

  $5n^2 \leq 5n^2$, we can take c=5 and N=0.

- $T(n)= \frac{n(n-1)}{2}$. Since for $n \geq 0$,

  $\frac{n(n-1)}{2} \leq \frac{n^2}{2}$, we can take c=1/2 and N=0.

- $T(n)= n^2 + 10n$. Since for $n \geq 1$,

  $n^2 + 10n \leq 11n^2$, we can take c=11 and N=1.

  $n^2 + 10n \leq 2n^2 \Rightarrow 10 \leq n$,

  we can take c=2 and N=10.

More Examples of Big O

- We can show that $n \in O(n^2)$. (how?)
- Since Big O is to denote upper bound, we prefer lowest upper bound which is the smallest possible function.

"=" or " ∈"

g(n) = O(f(n)) V.S. g(n) ∈ O(f(n))

Note

- "=" is not "equality", it is like "ϵ (belong to)"
  The equality is {g(n)} ⊆ O(f(n))
- O(f(n)) = g(n) X
- Ex: g(n) = O(n^2) and g(n) = O(n^3) ,
  so O(n^2) = O(n^3) ?
- In order to compare using asymptotic notation O, both have to be non-negative for sufficiently large n

Big O

The following statements hold for any real-valued functions $f(n)$ and $g(n)$, where there is a constant n_0 such that $f(n)$ and $g(n)$ are nonnegative for any integer $n \geq n_0$.

- Rule 1: $f(n) = O(f(n))$.
- Rule 2: If c is a positive constant, then $c \cdot O(f(n)) = O(f(n))$.
- Rule 3: If $f(n) = O(g(n))$, then $O(f(n)) = O(g(n))$.
- Rule 4: $O(f(n)) \cdot O(g(n)) = O(f(n) \cdot g(n))$.
- Rule 5: $O(f(n) \cdot g(n)) = f(n) \cdot O(g(n))$.

Review

```
int i, j                          O(1)
j =1                              O(1)
for (i =2; i<=n ; i++)            O(n)
      if (A[i]>A[j])              O(1)
            j=i;                  O(1)
return j                          O(1)
```

The worst-case time complexity is

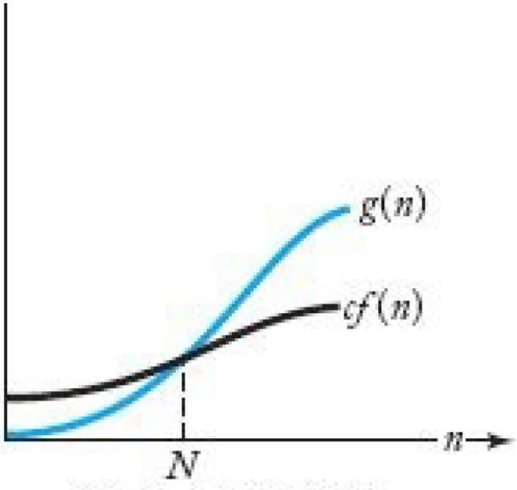
$$O(1) + O(1) + O(n)\cdot (O(1) + O(1)) + O(1)$$

$$= 3 \cdot O(1) + O(n)\cdot (2O(1))$$

$$= O(1) + O(n) = O(1) + O(n) = O(n)$$

Asymptotic Lower Bound

- Given a complexity function f(n), Ω(f(n)) is the set of complexity function g(n) s.t. there exists some positive real constant c and some nonnegative integer N s.t., for all $n \geq N$, $g(n) \geq c \times f(n)$.
- If $g(n) \in \Omega(f(n))$, we say that g(n) is omega of f(n).
- Eventually g(n) will be above cf(n) and stay there.
- Omega puts an asymptotic lower bound on g(n).
- Prefer highest lower bound.



(b) $g(n) \in \Omega(f(n))$

Omega (Ω) Illustrated

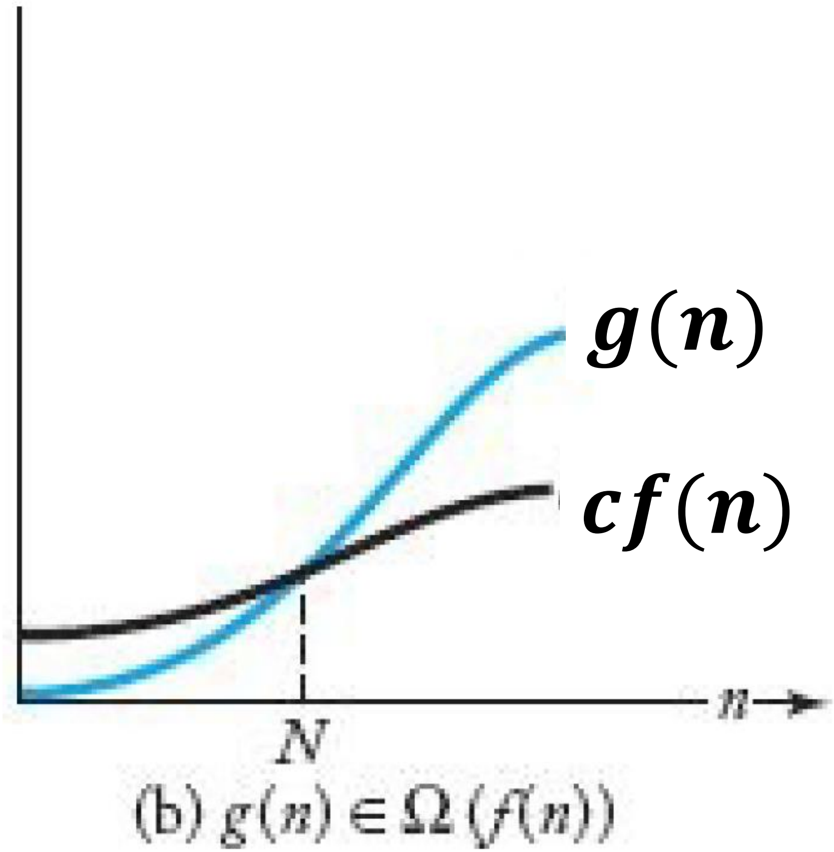
- **Example**:
- g(n)= $5n^2$+3 n+2,
- g(n) = Ω(f(n)) = Ω(n^2)

c=5

$5n^2+3n+2 \geq 5n^2$

=> 3n – 2 $\geq$0

=> N= 1



g(n) = Ω(f(n)) ↔ ∃ c, N> 0, ∋ g(n) ≥ cf(n), ∀ n≥ N

Big O Illustrated

- **Example**:

  g(n)= $5n^2$+3 n+2, g(n) = O(f(n)) = O(n^2)

  c=6

  $5n^2+3n+2 \le 6n^2 \Rightarrow 3n+2 < n^2 \Rightarrow N=4$

g(n) = O(f(n)) ↔ ∃ c, N> 0, ∋ g(n) ≤ cf(n), ∀ n≥ N

- g(n)= $5n^2$+3 n+2, g(n) = Ω(f(n)) = Ω(n^2)

  c=5

  $5n^2+3n+2 \ge 5n^2 \Rightarrow 3n-2 \ge 0 \Rightarrow N=1$

g(n) = Ω(f(n)) ↔ ∃ c, N> 0, ∋ g(n) ≥ cf(n), ∀ n≥ N

Review

```
1.  int i, j                      Ω(1)
2.  j =1                          Ω(1)
3.  for (i =2; i<=n ; i++)        Ω(n)
4.      if (A[i]>A[j])            Ω(1)
5.          j=i;                  Ω(1)
6.  return j                      Ω(1)
```

The worst-case time complexity is

$$\Omega(1) + \Omega(1) + \Omega(n)\cdot(\Omega(1) + \Omega(1)) + \Omega(1)$$

$$= 3 \cdot \Omega(1) + \Omega(n)\cdot(2\Omega(1))$$

$$= \Omega(1) + \Omega(n) = \Omega(1) + \Omega(n) = \Omega(n)$$?

Review

`int i, j`	$\Omega(1)$
`int m = A[1]`	$\Omega(1)$
`for (i =2; i<=n ; i++){`	$\Omega(n)$
`    if (A[i]>m)`	$\Omega(1)$
`        m=A[i];`	$\Omega(1)$
`    if (i ==n)`	$\Omega(1)$
`        do i++ n times`	$\Omega(n)$
`}`	
`return m;`	$\Omega(1)$

The worst-case time complexity is

$$3 \cdot \Omega(1) + \Omega(n) \cdot (3\Omega(1) + \Omega(n))$$
$$= \Omega(1) + \Omega(n) \cdot \Omega(n) = \Omega(1) + \Omega(n^2) = \Omega(n^2)$$

Order

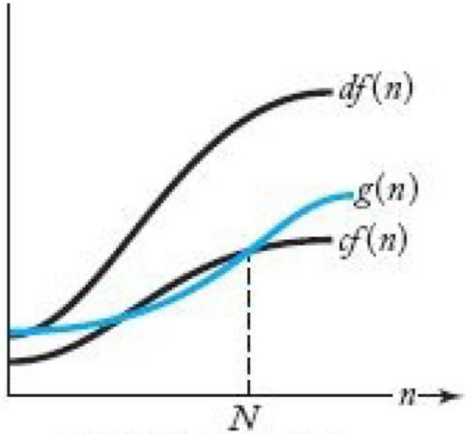
- For a complexity function f(n),

  $\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$.

- $\Phi(f(n))$ is the set of complexity function g(n) for which there exists some positive real constants c and d and some nonnegative integer N s.t., for all all $n \geq N$, $c \times f(n) \leq g(n) \leq d \times f(n)$.

Intersection

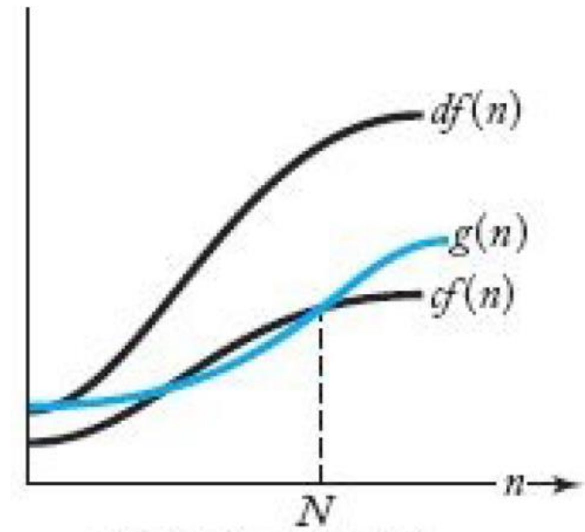
- If $g(n) \in \Phi(f(n))$, we say that g(n) is order of f(n).
- Eventually the growth of g(n) is similar to that of f(n) which can be used to characterize all such g(n)



(c) $g(n) \in \Theta(f(n))$

Order

- **Example**:
- g(n)= $5n^2$+3 n+2, g(n) = Θ(f(n)) = $\Theta(n^2)$



(c) $g(n) \in \Theta(f(n))$

d=6, c=5

$5n^2$+3 n+2 ≤ 6 n^2

$5n^2$+3 n+2 ≤ 6 n^2 =>

3 n + 2 < n^2 => N=4

g(n) = Θ(f(n)) ↔ ∃ c, d, N> 0, ∋ $cf(n) \leq g(n) \leq df(n)$, ∀ n≥ N

Small O

- Given a complexity function $f(n)$, $o(f(n))$ is the set of complexity function $g(n)$ s.t. for **every** positive real constant c there exists a nonnegative integer N such that for all $n \geq N$, $g(n) \leq c \times f(n)$.
- If $g(n) \in o(f(n))$, we say that $g(n)$ is small o of $f(n)$.
- This means that $g(n)$ becomes much smaller than $f(n)$ as n becomes large, independent of the constant c.
- For complexity comparison, $g(n)$ is much better than $f(n)$.

$\boxed{g(n) = O(f(n)) \leftrightarrow \exists\ c, N > 0,\ \ni g(n) < cf(n),\ \forall\ n \geq N}$

- **Example**: $n \in o(n^2)$. (why?)

Review

$g(n) = O(f(n))$

$g(n) \le f(n)$ in rate of growth

$\leftrightarrow \exists$ c, N> 0, $\ni$ g(n) $\le$ cf(n), $\forall$ n$\ge$ N

$g(n) = \Omega(f(n))$

$g(n) \ge f(n)$ in rate of growth

$\leftrightarrow \exists$ c, N> 0, $\ni$ g(n) $\ge$ cf(n), $\forall$ n$\ge$ N

$g(n) = o(f(n))$

$g(n) < f(n)$ in rate of growth

$\leftrightarrow \exists$ c, N> 0, $\ni$ g(n) $<$ cf(n), $\forall$ n$\ge$ N

$g(n) = \omega(f(n))$

$g(n) > f(n)$ in rate of growth

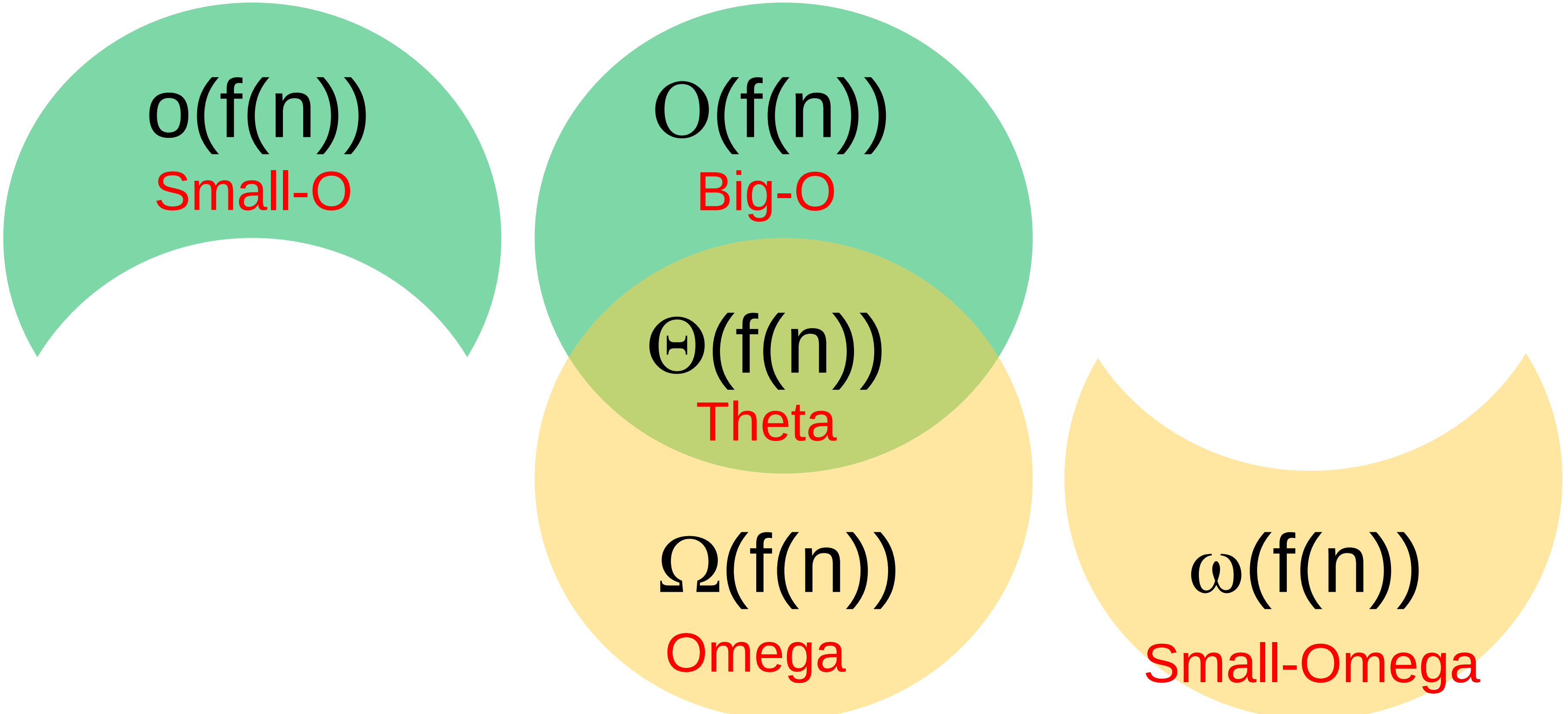
$\leftrightarrow \exists$ c, N> 0, $\ni$ g(n) $>$ cf(n), $\forall$ n$\ge$ N

$g(n) = \Theta(f(n))$ or $\theta(f(n))$

$g(n) = f(n)$ in rate of growth

$\leftrightarrow \exists$ c, d, N> 0, $\ni$ $cf(n) \le g(n) \le df(n)$, $\forall$ n$\ge$ N

Review



Review (Order)

how to analyze / measure the effort an algorithm needs

- Time complexity
- Space complexity

- Every Case
  - T(n): every-case time complexity
- *Worst Case
  - W(n): worst-case time complexity
- Average Case
  - A(n): average-case time complexity
- Best Case
  - B(n): best-case time complexity

Properties of Order 1/2

1. $g(n) \in O(f(n))$ if and only if $f(n) \in \Omega(g(n))$.
2. $g(n) \in \Theta(f(n))$ if and only if $f(n) \in \Theta(g(n))$.
3. If $b > 1$ and $a > 1$, then $\log_a n \in \Theta(\log_b n)$. (All logarithmic complexity functions are in the same class which is represented by $\Theta(\lg n)$.)
4. If $b > a > 0$, then $a^n \in o(b^n)$. (All exponential complexity functions are NOT in the same class.)
5. For all $a > 0$, $a^n \in o(n!)$. ($n!$ is worse than any exponential complexity.)

Properties of Order 2/2

6. With k > j > 2, b > a > 1, and the class ordering: $\Theta(\lg n)$ $\Theta(n)$ $\Theta(n\ lg\ n)$ $\Theta(n^2)$ $\Theta(n^j)$ $\Theta(n^k)$ $\Theta(a^n)$ $\Theta(b^n)$ $\Theta(n!)$, if g(n) is in class to the left of class f(n), then $g(n) \in o(f(n))$. (Classes differ in order-of-magnitude scale.)
7. If $c \geq 0$, $d > 0$, $g(n) \in O(f(n))$, and $h(n) \in \Theta(f(n))$, then $c \times g(n) + d \times h(n) \in \Theta(f(n))$. (What does this property means?)

Properties of Order 3/3

Big-Oh 函數	名稱
$O(1)$	常數時間(constant)
$O(\log_2 n)$	次線性時間(sub-linear) 或對數時間(logarithm)
$O(n)$	線性時間(linear)
$O(n\log_2 n)$	次平方時間(sub- quadratic)
$O(n^2)$	平方時間(quadratic)
$O(n^3)$	立方時間(cubic)
$O(2^n)$	指數時間(exponential)
$O(n!)$	階乘時間(factorial)

Assignment 1 Algorithms & Complexity

1. Design an algorithm to find all palindromes(迴文) of length ≥ 2. It does not need to be an optimal algorithm, as long as it can solve the problem.
2. Analyze the every-case (if exists), worst-case, average-case, and best-case time complexities of your algorithm.
3. Textbook exercises 1-15~18, 1-22.

Due date: two weeks.

Assignment 1 Algorithms & Complexity

1-15 Show directly that $f(n) = n^2 + 3n^3 \in (n^3)$. That is, use the definitions of O and Ω to show that $f(n)$ is in both $O(n^3)$ and $\Omega(n^3)$.

1-16 Using the definitions of O and Ω, show that

$$6n^2 + 20n \in O(n^3), \text{ but } 6n^2 + 20n \notin \Omega(n^3).$$

1-17 Using the Properties of Order in Section 1.4.2, show that

$$5n^5 + 4n^4 + 6n^3 + 2n^2 + n + 7 \in \Theta(n^5).$$

1-18 Let $p(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0$, where $a_k > 0$. Using the Properties of Order in Section 1.4.2, show that $p(n) \in \Theta(n^k)$.

1-22 Group the following functions by complexity category.

$$n \ln n \qquad (\lg n)^2 \qquad 5n^2 + 7n \qquad n^{5/2}$$

$$n! \qquad 2^{n!} \qquad 4^n \qquad n^n \qquad n^n + \ln n$$

$$5^{\lg n} \qquad \lg(n!) \qquad (\lg n)! \qquad \sqrt{n} \qquad e^n \qquad 8n + 12 \qquad 10^n + n^{20}$$

1 (constant) $<$ **log** $n < n < n$ **log** $n < n^2 < n^3 < 2^n < 3^n \le$ **n!**

Assignment 1 Algorithms & Complexity

名稱	公式	證明
和差	$\log_\alpha MN = \log_\alpha M + \log_\alpha N$	設$M=\beta^m$，$N=\beta^n$ $\begin{aligned}\log_\alpha MN &= \log_\alpha \beta^m\beta^n \\ &= \log_\alpha \beta^{m+n} \\ &= (m+n)\log_\alpha\beta \\ &= m\log_\alpha\beta + n\log_\alpha\beta \\ &= \log_\alpha \beta^m + \log_\alpha \beta^n \\ &= \log_\alpha M + \log_\alpha N \\ \log_\alpha \frac{M}{N} &= \log_\alpha M + \log_\alpha \frac{1}{N} \\ &= \log_\alpha M - \log_\alpha N\end{aligned}$
基變換（換底公式）	$\log_\alpha x = \frac{\log_\beta x}{\log_\beta \alpha}$	設 $\log_\alpha x = t$ $\therefore x = \alpha^t$ 兩邊取對數，則有$\log_\beta x = \log_\beta \alpha^t$ 即 $\log_\beta x = t\log_\beta \alpha$ 又$\because \log_\alpha x = t$ $\therefore \log_\alpha x = \frac{\log_\beta x}{\log_\beta \alpha}$
指係(次方公式)	$\log_{\alpha^n} x^m = \frac{m}{n}\log_\alpha x$	$\begin{aligned}\log_{\alpha^n} x^m &= \frac{\ln x^m}{\ln \alpha^n} \\ &= \frac{m\ln x}{n\ln\alpha} \\ &= \frac{m}{n}\log_\alpha x\end{aligned}$

Source:wiki

Assignment 1 Algorithms & Complexity

還原	$\alpha^{\log_\alpha x} = x$ $= \log_\alpha \alpha^x$	
互換	$M^{\log_\alpha N} = N^{\log_\alpha M}$	設 $b = \log_\alpha N, c = \log_\alpha M$ 則有 $\alpha^c = M$ and $\alpha^b = N$. 公式左側是 $(\alpha^c)^b$ 公式右側是$(\alpha^b)^c$
倒數	$\log_\alpha \theta = \frac{\ln\theta}{\ln\alpha} = \frac{1}{\frac{\ln\alpha}{\ln\theta}} = \frac{1}{\log_\theta \alpha}$	
鏈式	$\log_\gamma \beta \log_\beta \alpha = \frac{\ln\alpha}{\ln\beta}\frac{\ln\beta}{\ln\gamma}$ $= \frac{\ln\alpha}{\ln\gamma}$ $= \log_\gamma \alpha$	

log log n = log (log n)

$\log^k n = (\log n)^k$

Source:wiki