University of Science, VNU-HCM
Faculty of Information Technology

Data Structure and Algorithm

Analysis of Algorithms

Lecturer: Lê Ngọc Thành
Email: lnthanh@fit.hcmus.edu.vn

HCM City

Contents

- Introduction to algorithms and algorithm analysis
- Criteria and ways of evaluating algorithms
  - Basic criteria
  - $Big - O, Big - \Omega, Big - \Theta$
- Applying the assessment method

What is an algorithm?

- What is an algorithm?
  - *Algorithms are steps to take to solve a problem.*
  - For example, the problem of "fried chicken eggs“
    - *Step 1: take the saucepan.*
    - *Step 2: take the bottle of cooking oil*
      - *Is there a bottle of cooking oil?*
        - » *If so, pour it into the pan*
        - » *If not, go buy it?*
    - *Step 3: turn on the gas stove*
    - *…*

Necessary to analyze the algorithm?

- *How many ways to get from Ho Chi Minh City to Hanoi?*
  - Aircraft
  - Train
  - Boat
  - Buses
  - Bus
  - Ho Chi Minh Way...

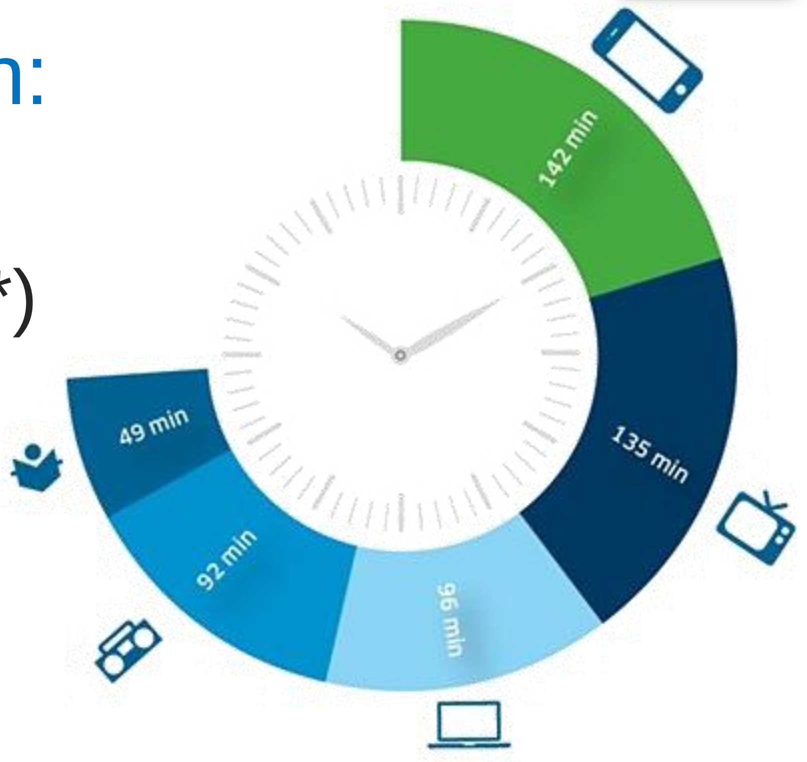
- *Why not optimize the only one way to go?*
  - Depends on circumstances, conditions, availability, convenience, comfort, …

Necessary to analyze the algorithm?

- In computer science, there are also many algorithms to solve the same problem.
  – Example?
    - Sorting problem: insertion sort, selection sort, quick sort, ...
    - Compression algorithm: Huffman, RLE compression, ...
- Algorithm analysis helps determine which algorithms are effective in terms of space and time, …
  – How long does the program run?
  – Why is the program out of memory?
  – …

Object of algorithm analysis

- Aspect of consideration:
  - – Running time (*)
  - – Memory consumption (*)
  - – Ease of understanding
  - – Stability
  - – Fault tolerance
  - – …
- Now, we focus on running time.



Time-out analysis

- Time-out analysis:
  - Consider how the processing time increases as the size of the problem increases.
  - The size of the problem usually depends on the size of the input:
    - Array size
    - Number of matrix elements
    - Number of bits expressed by input
    - Number of vertexes and edges of a graph
    - …

How to compare algorithms?

- Measurement:
  - Run time?
    - Not good because it depends on the configuration of each computer.
  - Number of statements are performed?
    - Not good because it depends on the programming language as well as the style of each programmer.
- So is there any way to compare independently of the machine, programming style, ...?
  - *Viewing the duration of a program's implementation is a function of n.*

$$f(n) = \cdots$$

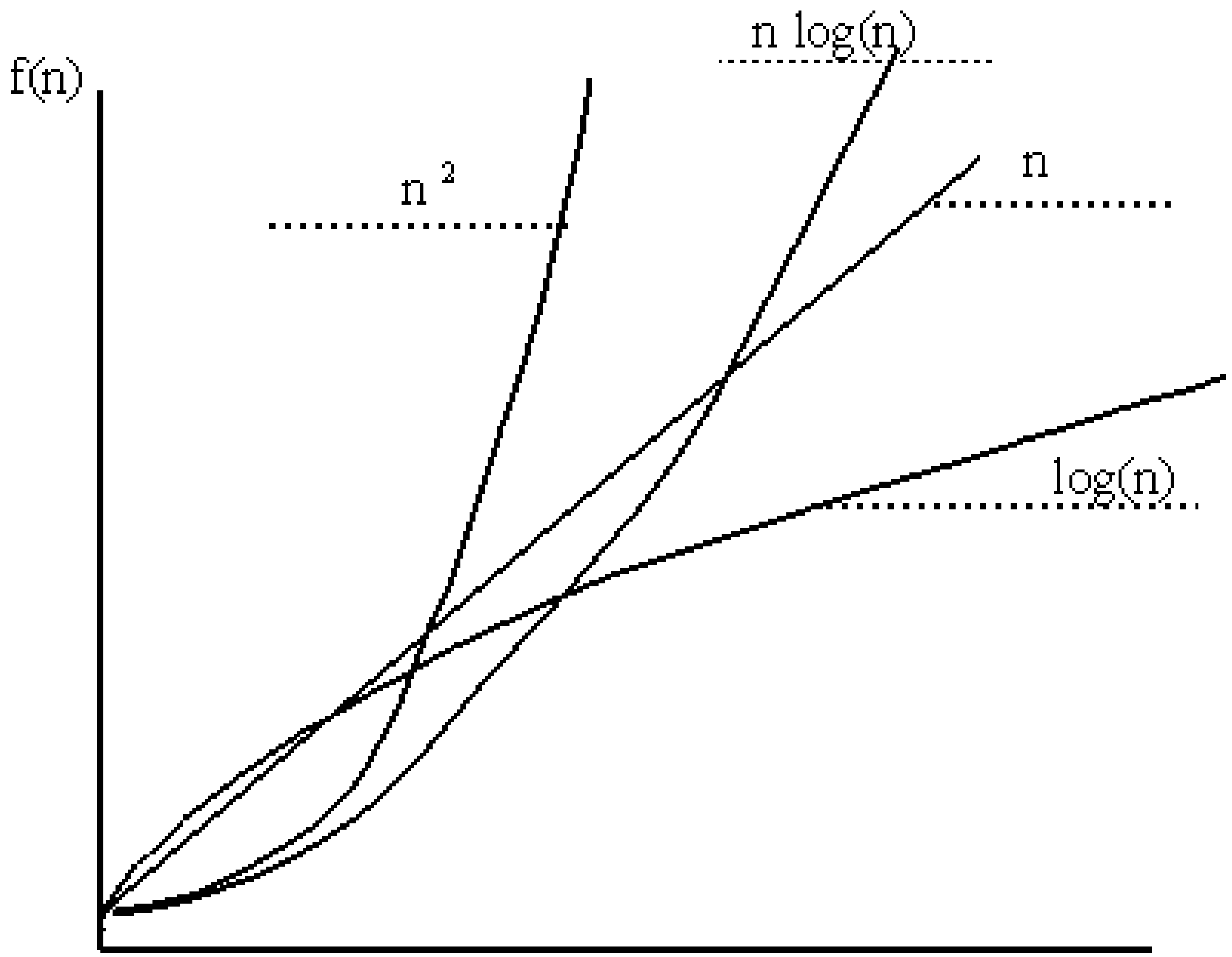
Growth rate

- **Growth rate** is the rate at which the cost of the algorithm increases as the size of input increases.
- The rate of increase usually *depends on the fastest element* as $n \longrightarrow \infty$:
  - Example 1: when the monthly income is several billion, a few thousand becomes odd, insignificant.
  - Example 2: an algorithm with an running time function

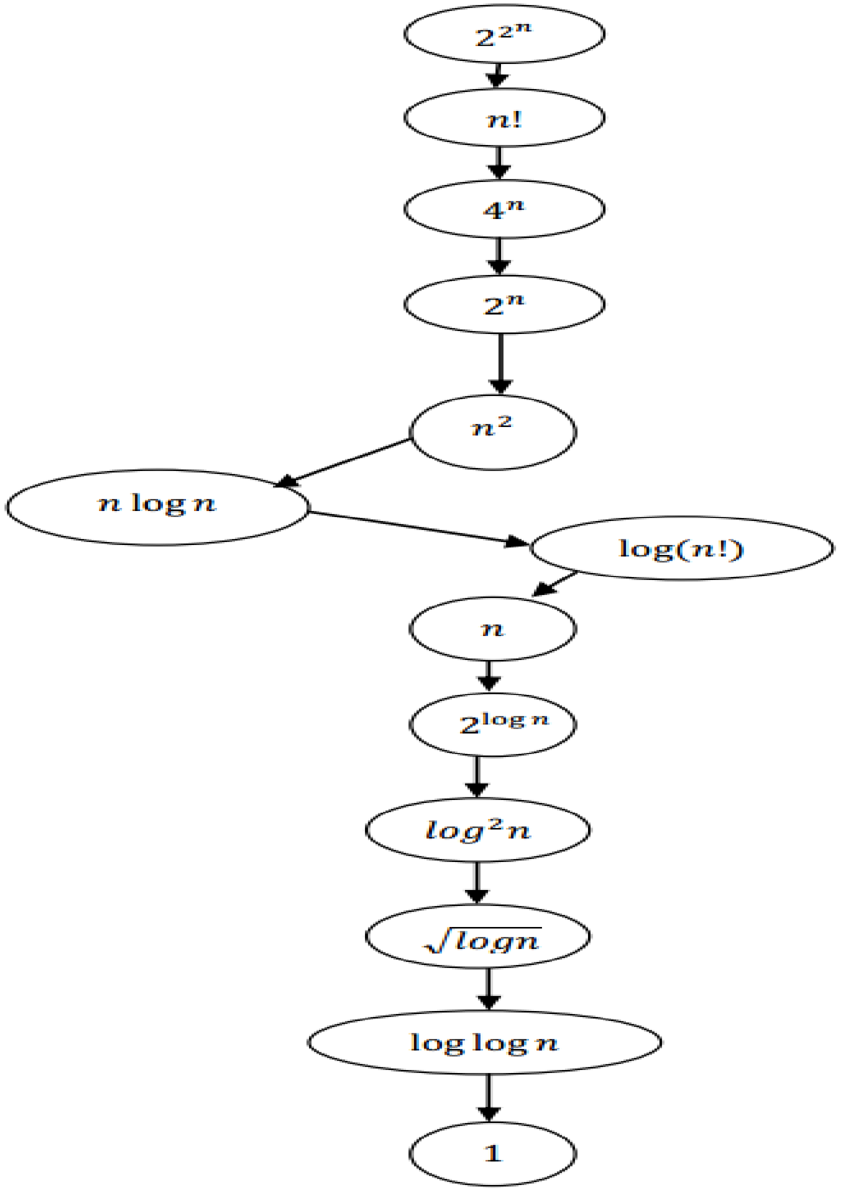
$$f(n) = n^{100} + n^2 + 10000000 \approx n^{100}$$

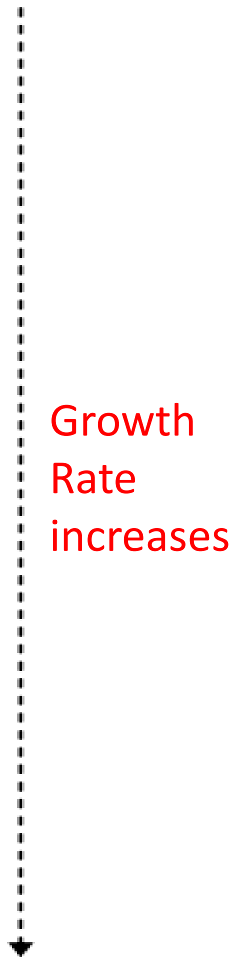
- Thus, when comparing algorithms, it is usually based on the growth rate.

Growth rate



Compare functions on growth rate





Type of analysis

- Depends not only on input size, *but the layout, structure, type* of input data
  → the growth rate is also different.
- *Example: number of comparisons to find value 1 in the following array?*

1	25	6	5	2	37	40

40	25	6	5	2	37	1

Type of analysis

- There are three types of analysis:
  - Best case (Big-Ω):
    - The input data layout makes the algorithm run the fastest.
    - Rarely happens in practice.
  - Average case:
    - Random data layout
    - Difficult to predict to distribution.
  - Worst case, Big-O:
    - Input data layout causes the algorithm to run at the slowest.
    - Make sure to upper bound.

$$\textit{Lower Bound} \le \textit{Average Time} \le \textit{Upper Bound}$$

Big-O

- In order to assess the worst case or upper bound, one gives the concept of Big-O.

O

$$\boldsymbol{Lower\ Bound \leq Average\ Time \leq Upper\ Bound}$$

History of Big-O

- The notation Big-O was introduced in 1894 by ***Paul Bachmann*** (Germany) in the book Analytische Zahlentheorie (“Analytic Number Theory") (2nd edition)
- This notation (later) was popularized by mathematics ***Edmund Landau***, so it's called the Landau notation, or Bachmann-Landau notation
- ***Donald Knuth*** was the one who introduced the notation into Computer Science in 1976 – “Big Omicron and big Omega and big Theta” - ACM SIGACT News, Volume 8, Issue 2

Big-O (tt)

- Given an algorithm's time function to be $f(n)$
  - – Example: $f(n) = n^4 + 100n^2 + 10n + 50$
- Calling the upper bound function of f(n) is $g(n)$ when n is large enough. That is, there is no large enough n value to make $f(n)$ beyond $cg(n)$ (c: constant)
  - – Ví dụ: $g(n) = n^4$
- $f(n)$ is called the Big-O of g(n).

Or $g(n) is\ the\ highest\ growth\ rate\ of\ f(n)$

Big-O (tt)

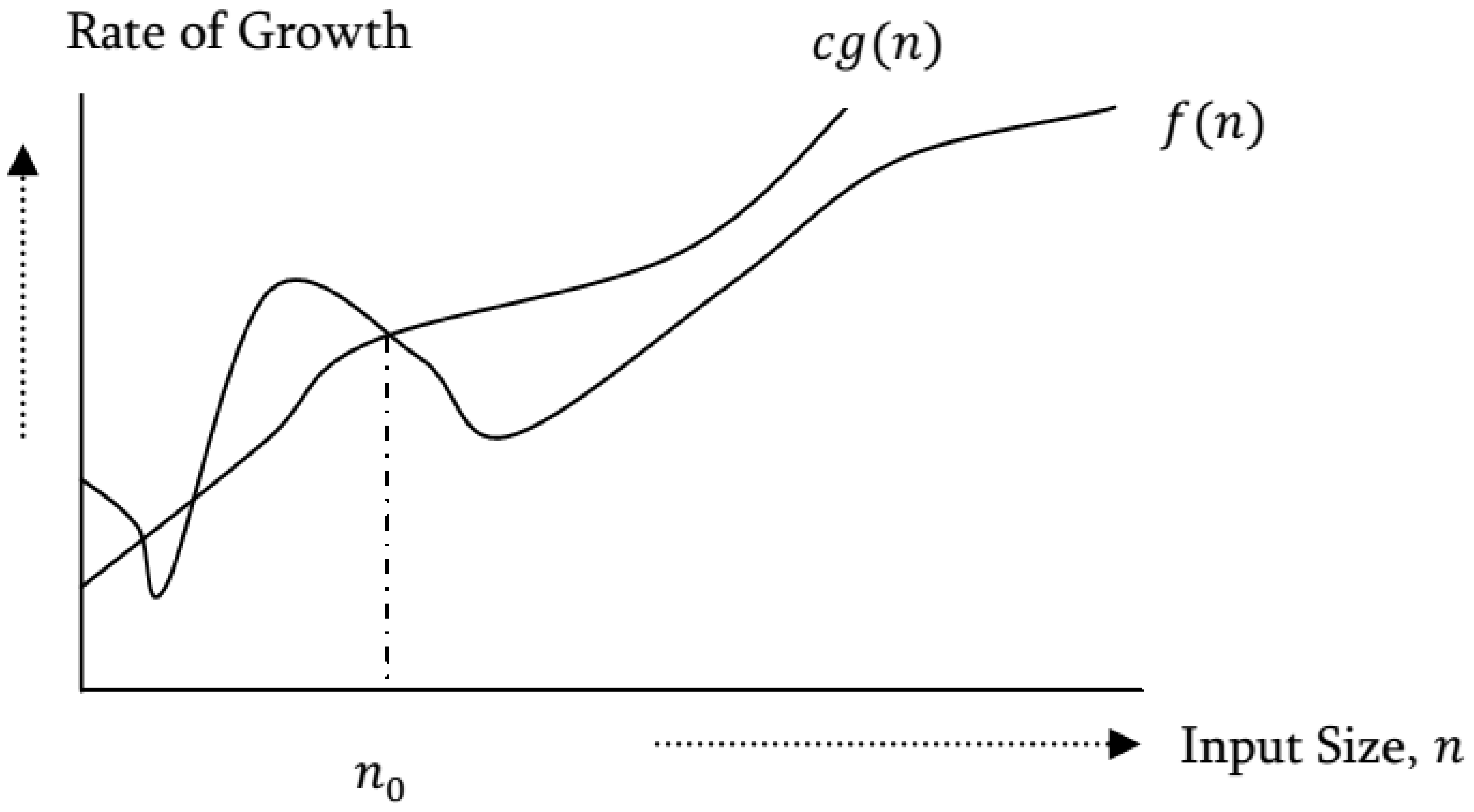
- Definition:
  - Given f(n) and g(n) are two functions
  - $f(n) = O(g(n))$, if there exists positive numbers c and n_0 so that:

    $$0 \leq f(n) \leq cg(n)$$

    where $n \geq n_0$
  - f is the Big-O of g if a positive number exists so that f cannot be greater than $c * g$ when n is large enough

$\rightarrow g(n)$ is called the upper bound of $f(n)$

Big-O



For small n ($n < n_0$) we often do not care because it is often in critical and does not show the complexity of the algorithm.

Algorithm analysis with Big-O

- Comments:
  - Each algorithm has many running time functions corresponding to each input's layout resulting in the best, average, and worst cases.
  - $g(n)$ is the upper bound of all those functions.
- So comparing algorithms, we just need to compare its $g(n)$ or Big-O.
  - For example: Instead of using the complexity of the algorithm is $2n^2 + 6n + 1$, we will use the upper bound of the complexity of the algorithm as n^2

Is there one or more functions g(n)?

- If exists $g'(n) > g(n)$, does $g'(n)$ satisfy Big-O conditions?
- Notes:
  - The best $g(n)$ is that the smallest function still satisfies Big-O.
  - As simple as possible (?)

Big-O exercises

- Find the upper bound function with the c and n_0 values of the following functions:
  - *a)* $f(n) = 3n + 8$
  - *b)* $f(n) = n^2 + 1$
  - *c)* $f(n) = n^4 + 100n^2 + 50$
  - *d)* $f(n) = 2n^3 - 2n^2$
  - *e)* $f(n) = n$
  - *f)* $f(n) = 410$
- Solutions: $(g(n), c, n_0)$

a. $(n, 4, 8)$	b. $(n^2, 2, 1)$	c. $(n^4, 2, 100)$
d. $(n^3, 2,1)$	e. $(n^2, 1,1)$	f. $(1, 1, 1)$

Is there another answer??

Big-O exercises

- Uniqueness?
  - $f(n) = 100n + 5$

Big-O exercises

- *Identify the O(g(n)) of the following functions:*
  - $f(n) = 10$
  - $f(n) = 5n + 3$
  - $f(n) = 10n^2 - 3n + 20$
  - $f(n) = \log n + 100$
  - $f(n) = n\log n + \log n + 5$
- *Which statement is correct?*
  - $2^{n+1} = O(2^n)$?
  - $2^{2n} = O(2^n)$?

Big-Ω

- *Lower bound* or *best case* expressed by Big-Ω.

Ω

$$\boldsymbol{Lower\ Bound} \le \boldsymbol{Average\ Time} \le \boldsymbol{Upper\ Bound}$$

Big-Ω

- Given an algorithm's time function to be $f(n)$
  - Example: $f(n) = 100n^2 + 10n + 50$
- Call the lower bound function of $f(n)$ as $g(n)$ when n is large enough. This means that no value of n is large enough to make $cg(n)$ exceed $f(n)$ (c: hằng số)
  - Example: $g(n) = n^2$
- $f(n)$ called Big-Ω of $g(n)$.

or $g(n)$ is the minimum growth rate of $f(n)$

Big-Ω

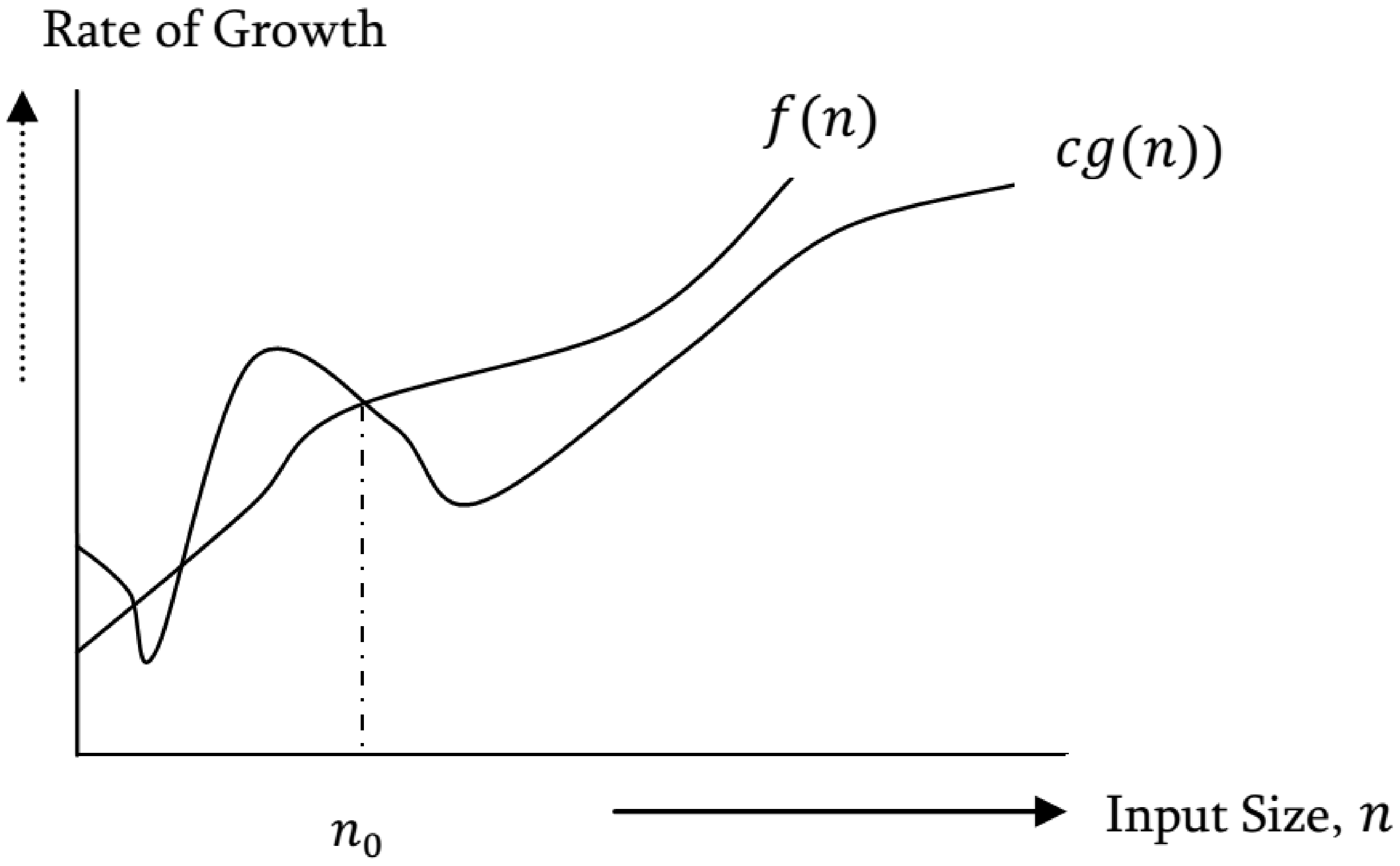
- Definition:
  - Given f(n) and g(n) are two functions
  - $f(n) = \Omega(g(n))$, if there exists positive numbers c and n_0 so that:

$$0 \le cg(n) \le f(n)$$

    where $n \ge n_0$
  - f is Big-Ω of g if a positive number exists so that f cannot be smaller *c*g* when n is large enough

$\rightarrow g(n)$ called the lower bound of $f(n)$

Big-Ω



Big-Ω exercises

- Find the lower bound function with the c and n_0 values of the following functions :
  $a)$ $f(n) = 5n^2$
  $b)$ $f(n) = n$
  $c)$ $f(n) = n^3$
- Solutions:

  a. $\Omega(n)$ b. $\Omega(logn)$ c. $\Omega(n^2)$

Big-Θ

- Big-Θ used to determine if the lower and upper bound are the same.

$$\textit{Lower Bound}\ (\Omega) \le \textit{Average Time} \le \textit{Upper Bound}(O)$$

$$\Theta$$

- If Ω and Θ are the same, the growth rate of the average case is similar.

Big-Θ

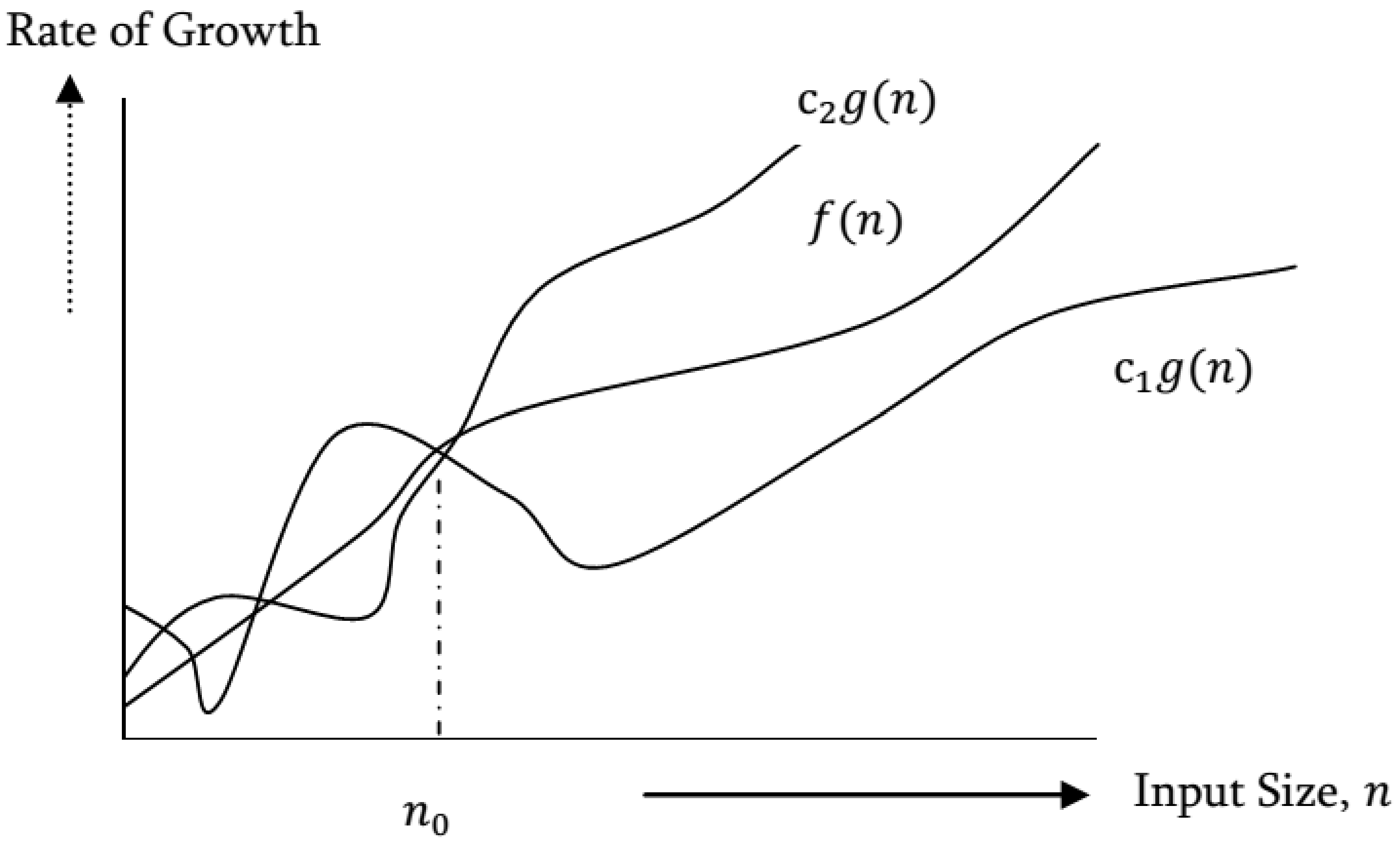
- Definition:
  - Given f(n) and g(n) are two functions
  - $f(n) = \Theta(g(n))$, if there exists positive numbers c and n_0 so that:

$$0 \le c_1 g(n) \le f(n) \le c_2 g(n)$$

where $n \ge n_0$

$\rightarrow g(n)$ is called the tight bound of $f(n)$

Big-Θ



Big- Θ Exercises

1) Find the tight bound function of the following function:

$$f(n) = \frac{n^2}{2} - \frac{n}{2}$$

2) Prove:

$a.\quad n \neq \theta(n^2)$

$b.\quad 6n^3 \neq \theta(n^2)$

$c.\quad n \neq \theta(\log n)$

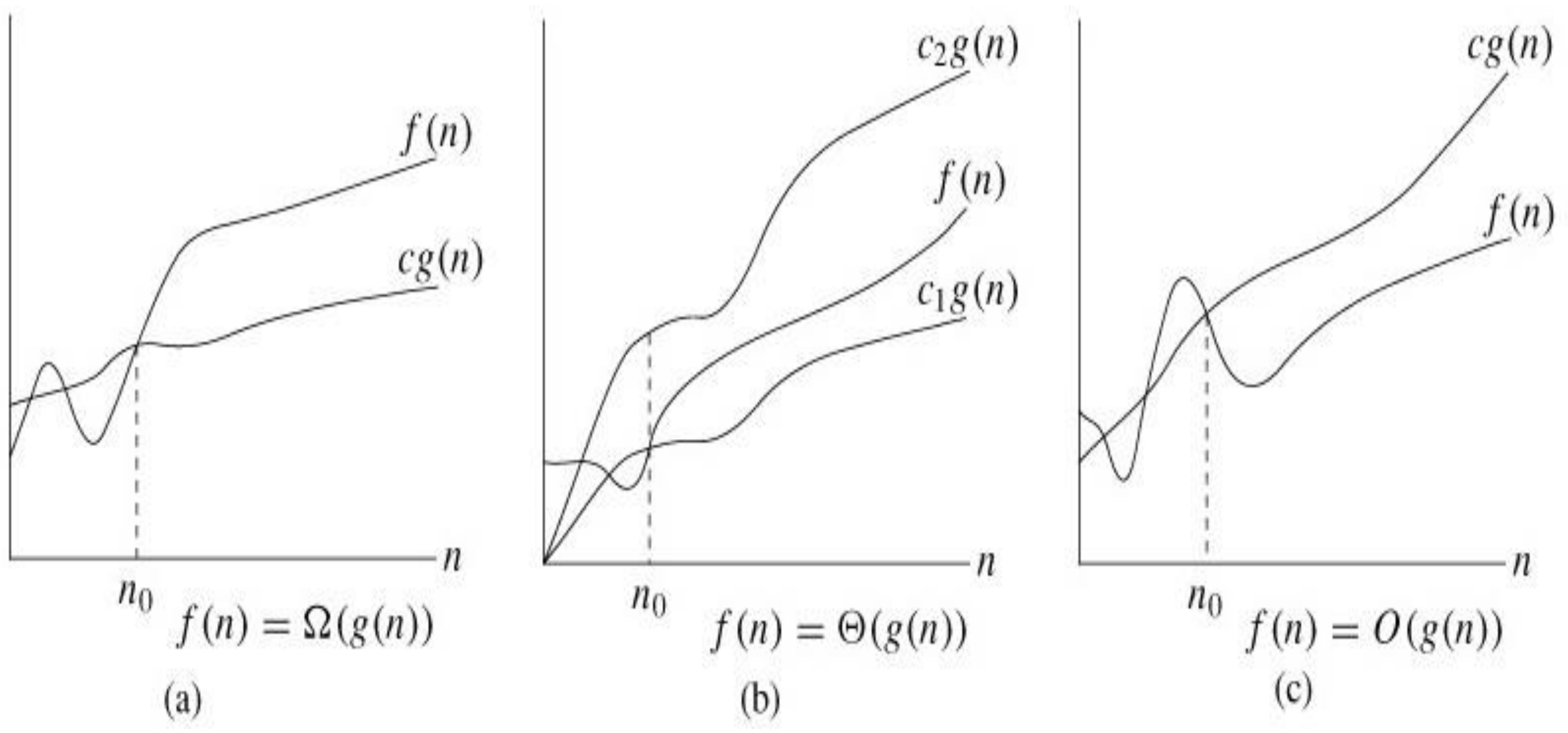
- Solution:

1) $\theta(n^2), c_1 = \frac{1}{5}, c_2 = 1, n_0 = 1$

2)

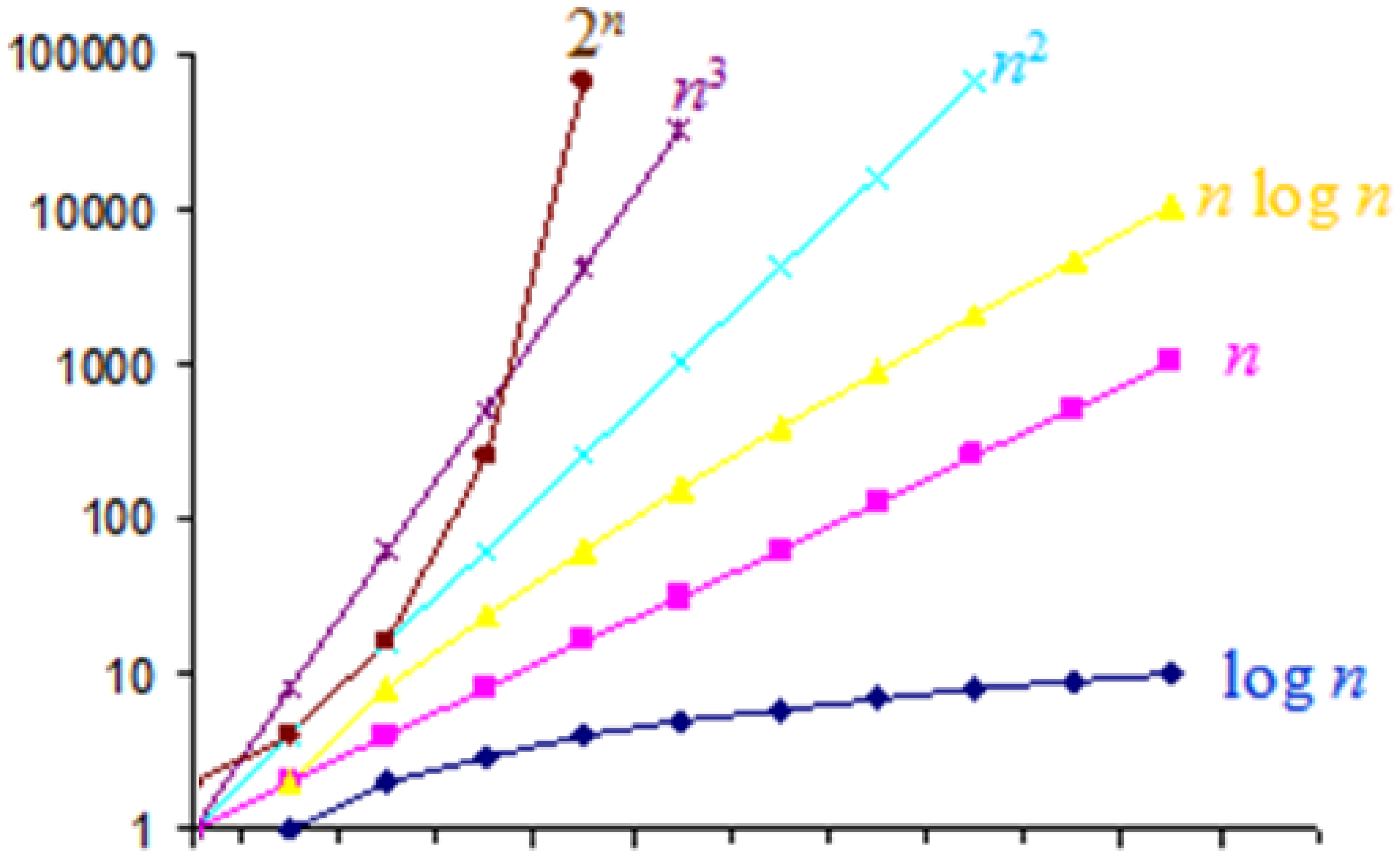
a. $\mathrm{n} \leq \frac{1}{c_1}$ (can't) …

Summary about Big-O, Big-Ω, Big-Θ

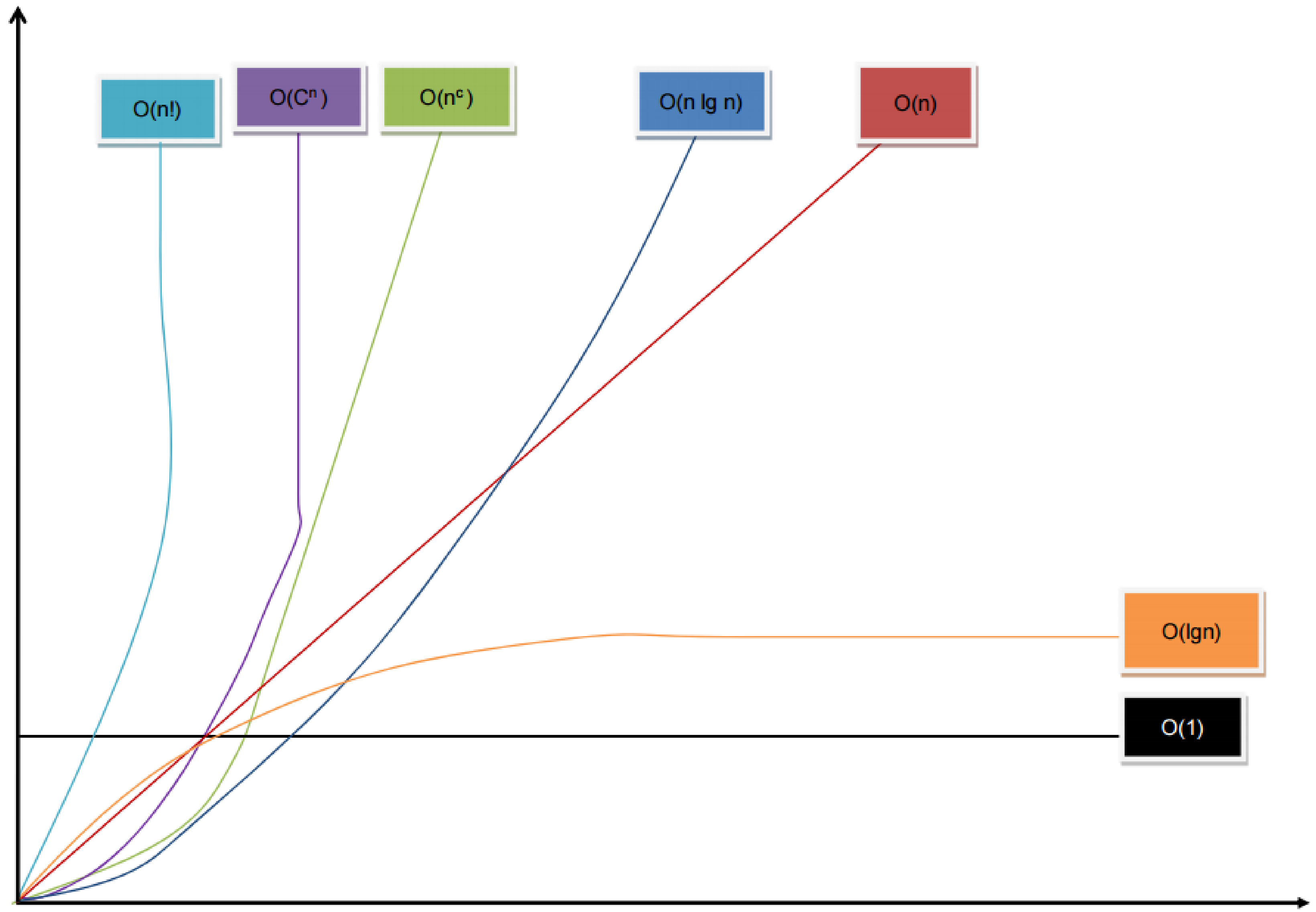


In general, the fact that people are only interested in the upper bound (Big-O), while the lower limit (Big-Ω) does not matter, the tight limit (Big- Θ) only considers when the upper and lower limits are the same.

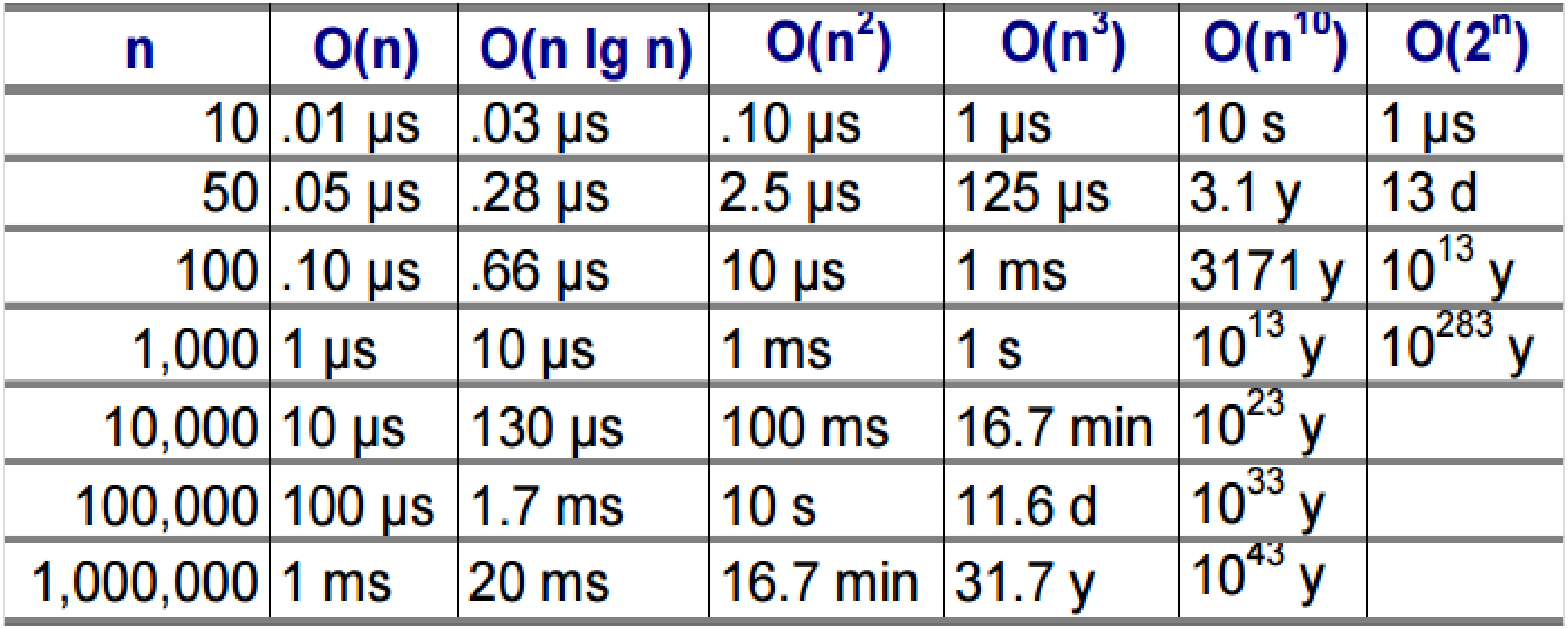
Compare some functions



Compare some functions



Run time

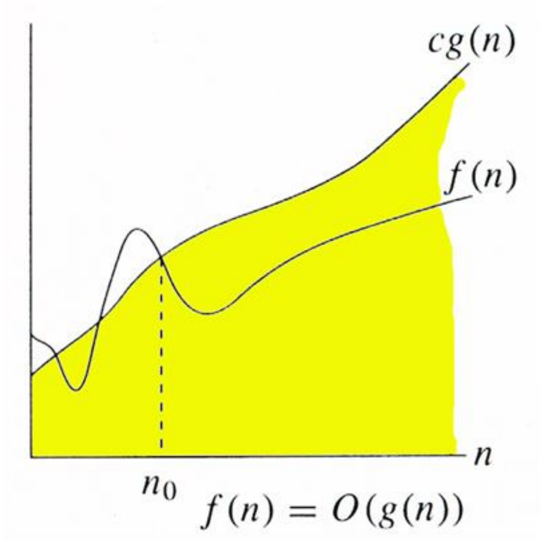


n	O(n)	O(n lg n)	O(n^2)	O(n^3)	O(n^{10})	O(2^n)
10	.01 µs	.03 µs	.10 µs	1 µs	10 s	1 µs
50	.05 µs	.28 µs	2.5 µs	125 µs	3.1 y	13 d
100	.10 µs	.66 µs	10 µs	1 ms	3171 y	10^{13} y
1,000	1 µs	10 µs	1 ms	1 s	10^{13} y	10^{283} y
10,000	10 µs	130 µs	100 ms	16.7 min	10^{23} y	
100,000	100 µs	1.7 ms	10 s	11.6 d	10^{33} y	
1,000,000	1 ms	20 ms	16.7 min	31.7 y	10^{43} y	

Complexity calculation method

Asymptotic analysis

- With an $f(n)$ algorithm, we always want to find another function $g(n)$ that approximately $f(n)$ has a higher value.
- The $g(n)$ curve is called the asymptotic curve.
- The problem of finding $g(n)$ is called asymptotic analysis



Asymptotic analysis – Loop

Loop runtime is usually the execution time of the instructions within the loop multiplied by the number of iterations.

For example:

```
for (i=1; i<=n; i++)
{
    m = m+2;        // constant time c
}
```

Run time = $c \times n = cn = O(n)$

Nested loops

For a nested loop, we analyze the inside out. The total time is the product of the size of all loops.

For example:

```
for (i=1; i<=n; i++)
{
    for (j=1; j<=n; j++)
    {
        k = k+1;        // constant time c
    }
}
```

Thời gian chạy = $c \times n \times n = cn^2 = O(n^2)$

Sequential instructions

For sequential instructions, to calculate run time, we add up the time complexity of each instruction

Example:

```
x = x + 1: //constant time c0
for (i=1; i<=n; i++)
{
        m = m + 2; //constant time c1
}
for (i=1; i<=n; i++)
{
        for (j=1; j<=n; j++)
        {
                k = k+1;         //constant time c2
        }
}
```

Run time = $c_0 + c_1 n + c_2 n^2$ = $O(n^2)$

Conditional statement

Usually we consider the worst case scenario, the run time is equal to the longest part of the conditional statement.

if (condition)	$\rightarrow$	$T_0(n)$
Statement 1	$\rightarrow$	$T_1(n)$
else		
Statement 2	$\rightarrow$	$T_2(n)$

Runtime : $T_0(n) + \max\,(T_1(n),\ T_2(n))$

Conditional statement

Example:

```
//test: constant c0
if (length() != otherStack. length())
{
      return false; //then part: constant c1
}
else
{
   // else part: (constant + constant) * n
   for (int n = 0; n < length(); n++)
   {
      if (!list[n].equals(otherStack.list[n]))
            //constant c2
            return false; // c3
      }
}
```

$Runtime = c_0 + c_1 n + (c_2 + c_3) n = O(n)$

Log complexity

- An algorithm has complexity $O\ (log n)$ if it takes a constant time to reduce the problem size by a fraction (usually ½).
- Example:

```
for (i=1; i<=n;)

{

      i=i*2;

}
```

The rule of addition and multiplication

- ***The rule of addition:***
  If $f_1 = O(g_1)$ and $f_2 = O(g_2)$ then $f_1+f_2 = O(\max\{g_1,g_2\})$
  If we perform several operations in order, the execution time is dominated by the operation that takes the most time.
- ***The rule of multiplication:***
  If $f_1 = O(g_1)$ and $f_2 = O(g_2)$ then $f_1*f_2 = O(g_1*g_2)$
  If we repeat an operation a number of times, the total execution time is the execution time of the operation multiplied by the number of iterations.

Some properties

- *Transitive property:*
  $f(n) = O\big(g(n)\big)$ and $g(n) = O\big(h(n)\big)$
  $\rightarrow f(n) = O(h(n))$
- *Reflectivity property:*
  $f(n) = O(f(n))$

Summary of basic complexity calculations

- Simple statement (read, write, assign)
  - O(1)
- Simple calculations (+ - * / == > >= < <=)
  - O(1)
- Sequence of simple statements / calculations
  - Rule of addition
- Loop for, do, while
  - Rule of multiplication

Calls a function at the same level?

Function A calls Function B

…

- The complexity of the sequence of operations calling the function?

$$O(n) = \max\{O_A(n), O_B(n), O_c(n) \ldots\}$$

Commonly used Logarithms and Summations

Arithmetic series

$$\sum_{K=1}^{n} k = 1 + 2 + \cdots + n = \frac{n(n+1)}{2}$$

Geometric series

$$\sum_{k=0}^{n} x^k = 1 + x + x^2 \ldots + x^n = \frac{x^{n+1} - 1}{x - 1} (x \neq 1)$$

Harmonic series

$$\sum_{k=1}^{n} \frac{1}{k} = 1 + \frac{1}{2} + \ \ldots + \frac{1}{n} \approx log\, n$$

Other important formulae

$$\sum_{k=1}^{n} log\, k \approx nlogn$$

$$\sum_{k=1}^{n} k^p = 1^p + 2^p + \cdots + n^p \approx \frac{1}{p+1} n^{p+1}$$

Big-O Exercises

- Calculate the upper bound (big-O) for the following algorithms:

```
Ex1: for  (i = 0; i < n; i++)
             for (j = 0; j < n; j++)
                    b[i][j] += c;

Ex2: for  (i = 0; i < n; i++)
             if (a[i] == k) return 1;
             return 0;

Ex3: for  (i = 0; i < n; i++)
             for (j = i+1; j < n; j++)
                    b[i][j] -= c;
```

Big-O Exercises

- Ex 4,5,6: matrix addition, multiplication and transposition

```
for(i = 0; i < n; i++)
    for(j = 0; j < n; j++)
        a[i][j] = b[i][j] + c[i][j];

for(i = 0; i < n; i++)
    for(j = 0; j < n; j++)
        for(k = a[i][j] = 0; k < n; k++)
            a[i][j] += b[i][k] * c[k][j];

for(i = 0; i < n - 1; i++)
    for(j = i+1; j < n; j++) {
        tmp = a[i][j];
        a[i][j] = a[j][i];
        a[j][i] = tmp;
    }
```

Some more exercises

```
s = 0;
for (i=0; i<=n;i++){
        p =1;
        for (j=1;j<=i;j++)
                p=p * x / j;
        s = s+p;
}
```

Some more exercises

```
s = 1; p = 1;
for (i=1;i<=n;i++) {
            p = p * x / i;
            s = s + p;
}
```

Some more exercises

- s=n*(n-1) /2;

Some more exercises

```
for (i= 1;i<=n;i++)
        for (j= 1;j<=m;j++)
                x += 2;
```

Some more exercises

```
(1)  for (i = 0 ; i < n ;  i++)
(2)       for (j = 0 ; j < n ; j++)
(3)                  A[i][j] = 0;

(4)  for (i = 0 ; i < n ; i++)
(5)       A[i][i] = 1;
```

Some more exercises

```
for (i = 0 ; i < n ;  i++)
    for (j = 0 ; j < n ; j++)
            if (i == j)
                  A[i][j] = 1;
            Else
                  A[i][j] = 0;
```

Some more exercises

```
sum  = 0;
for ( i = 0; i < n; i + +)
   for ( j = i + 1; j < = n; j + +)
      for ( k = 1; k < 10; k + +)
            sum = sum + i * j * k  ;
```

Some more exercises

```
sum  = 0;
for ( i = 0; i < n; i + +)
   for ( j = i + 1; j < = n; j + +)
      for ( k = 1; k < m; k + +) {
            x = 2*y;
            sum = sum + i * j * k  ;
      }
```

Some more exercises

```
sum  = 0;
for ( i = 0; i < n; i + +)
   for ( j = i + 1; j < = n; j + +)
      for ( k = 1; k < m; k + +) {
            x = 2*y;
            sum = sum + i * j * k  ;
     }
```

Some more exercises

```
for (i= 1;i<=n;i++)
        u = u + 2;
for (j= 1;j<=m;j++)
        u = u * 2;
```

Some more exercises

```
for (i= 1;i<=n;i++) {
        for (u= 1;u<=m;u++)
                for (v= 1;v<=n;v++)
                        h = x*100;
        for j:= 1 to x do
                for k:= 1 to z do
                        x = h/100;
}
```

Some more exercises

```
for (i= 1;i<=n;i++)
        for (j= 1;j<=m;j++) {
                for (k= 1;k<=x;k++)
                        //statement
                for (h= 1;h<=y;h++)
                        //statement
        }
```

Some more exercises

```
for (i= 1;i<=n;i++)
            for (u= 1;u<= m;u++)
                        for (v= 1;v<=n;v++)
                                    //statement
for (j= 1;j<=x;j++)
                        for (k= 1;k<=z;k++)
                                    // statement
```

Some more exercises

$P(x) = x^m x^m + a^{m-1} x^{m-1} + \ldots + a^1 x + a^0$

$Q(x) = b^n x^n + b^{n-1} x^{n-1} + \ldots + b^1 x + b^0$

```
if (m<n) p = m; else p =n;
for (i=0;i<=p;i++)
        c[i]=a[i] + b[i];
if (p<m)
        for (i=p+1;i<=m;i++) c[i] = a[i];
else
        for (i=p+1;i<=n;i++) c[i] = b[i];
while (p>0 && c[p] = 0) p = p-1;
```

Some more exercises

$P(x) = x^m x^m + a^{m-1} x^{m-1} + \ldots + a^1 x + a^0$

$Q(x) = b^n x^n + b^{n-1} x^{n-1} + \ldots + b^1 x + b^0$

```
p = m+n;
for (i=0;i<=p;i++) c[i] = 0;
for (i=0;i<=m;i++)
          for (j=0;j<=n;j++)
                    c[i+j] = c[i+j] + a[i] + b[j];
```

Conclusion

- Although analyzing algorithm complexity gives the most accurate comparison between algorithms, it can in fact make it difficult or impossible for programmers to find Big-O for some programs has high complexity.
- Therefore, people often return to the method of running an experiment (measuring the runtime) with the same hardware platform, language, data set. Also try changing the different data sets size and type.

The End.