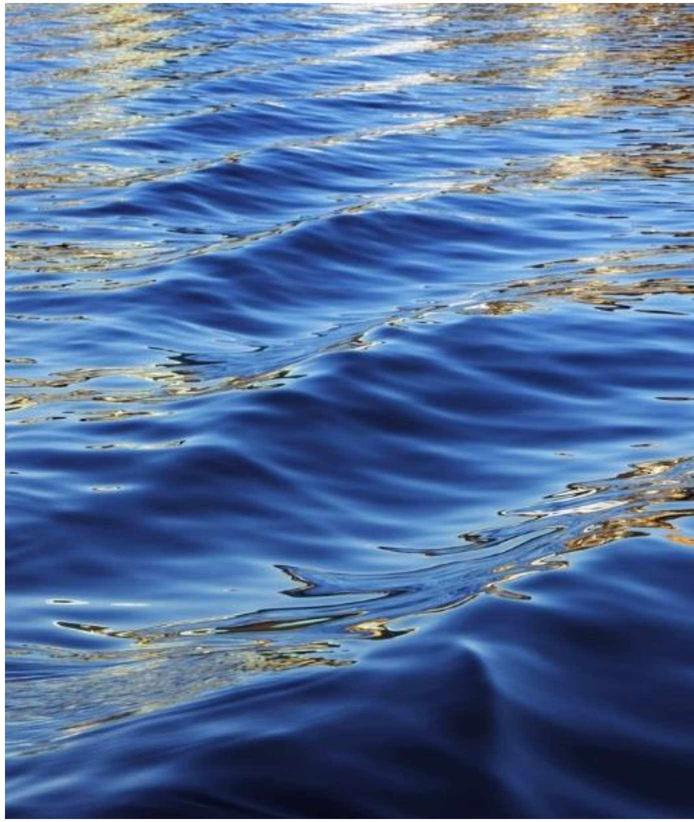
Analysis of Algorithms

Lecture 3



Outline

1.) Why performance analysis?

2.) Asymptotic Analysis

3.) Big O notation

3.) Guidelines for asymptotic analysis

4.) Example

5.) Exercises

Why we worry about performance?

There are many important things that should be taken care of, like user-friendliness, modularity, security, maintainability, Why we worry about performance.

We can have all those things only if we have performance.

Performance == Scale of system

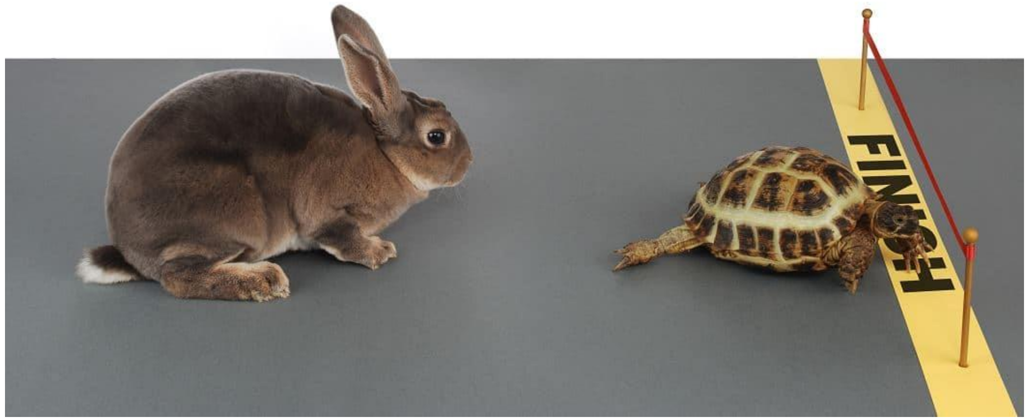
Example: You want to do a spell checking for your document.

If text editor can load 1000 pages but can spell check 1 page per minute.

What do you feel?

Given two algorithms for a task, how do we find out which one is better?

Input Size	Running time on A	Running time on B
10	2 sec	~ 1 h
100	20 sec	~ 1.8 h
10^6	~ 55.5 h	~ 5.5 h
10^9	~ 6.3 years	~ 8.3 h



Asymptotic Analysis

is a way to describe the running time or space complexity of an algorithm based on the input size.

It is commonly used in complexity analysis to describe how an algorithm performs as the size of the input grows

- Big O notation (O) : upper bound on the growth rate (worst-case scenario)
- Omega notation (Ω) : lower bound on the growth rate (best-case scenario)
- Theta notation (Θ) : both upper & lower bound (average-case scenario)

Asymptotic Analysis

- Big O notation (O) : upper bound on the growth rate (worst-case scenario)

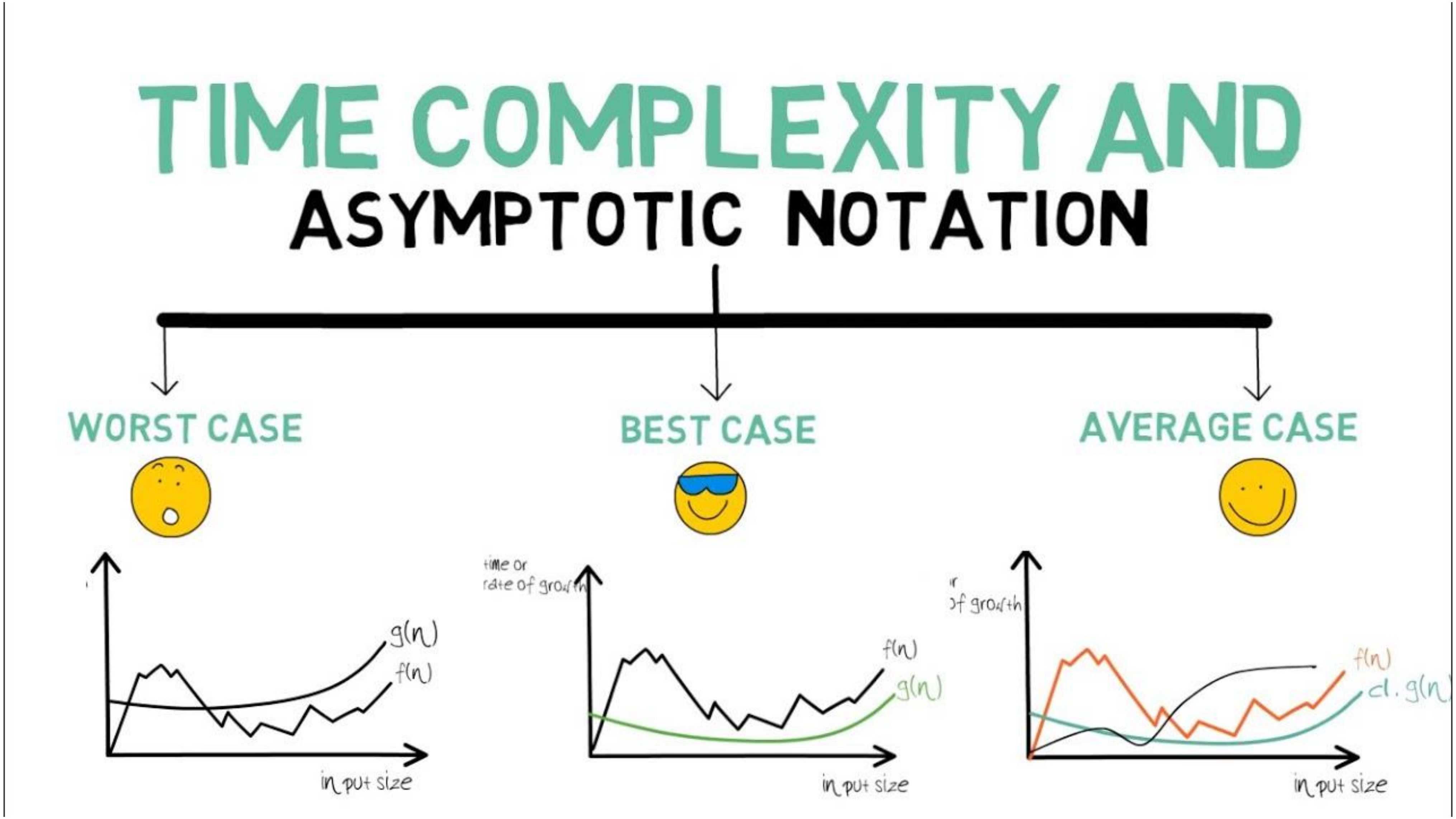
  maximum amount of time or space an algorithm may need to solve a problem.

- Omega notation (Ω) : lower bound on the growth rate (best-case scenario)

  minimum amount of time or space an algorithm may need to solve a problem.

- Theta notation (Θ) : both upper & lower bound (average-case scenario)

  typically amount of time or space an algorithm may need to solve a problem.



f(n) describes the running time of an algorithm

g(n) define as bound of the running time of an algorithm

Big O notation (O)

ค่า Big O น้อย = ประมวลผลได้ประสิทธิภาพดีกว่า (เร็วกว่า)

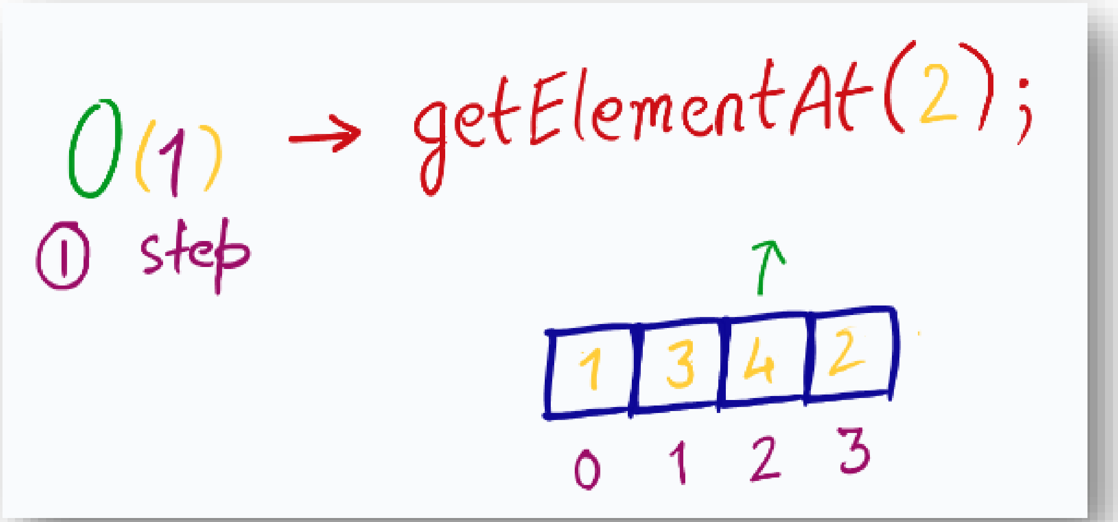
- ❑ O(1) Constant
- ❑ O(log n) Logarithmic
- ❑ O(n) Linear
- ❑ O(n log n) Linearithmic
- ❑ O(n^2) Quadratic
- ❑ O(n!) Factorial

Code สั้นๆ ประสิทธิภาพดีกว่ารึป่าว? ➔ **ไม่จริง**

Big O notation (O)

O(1) **Constant**

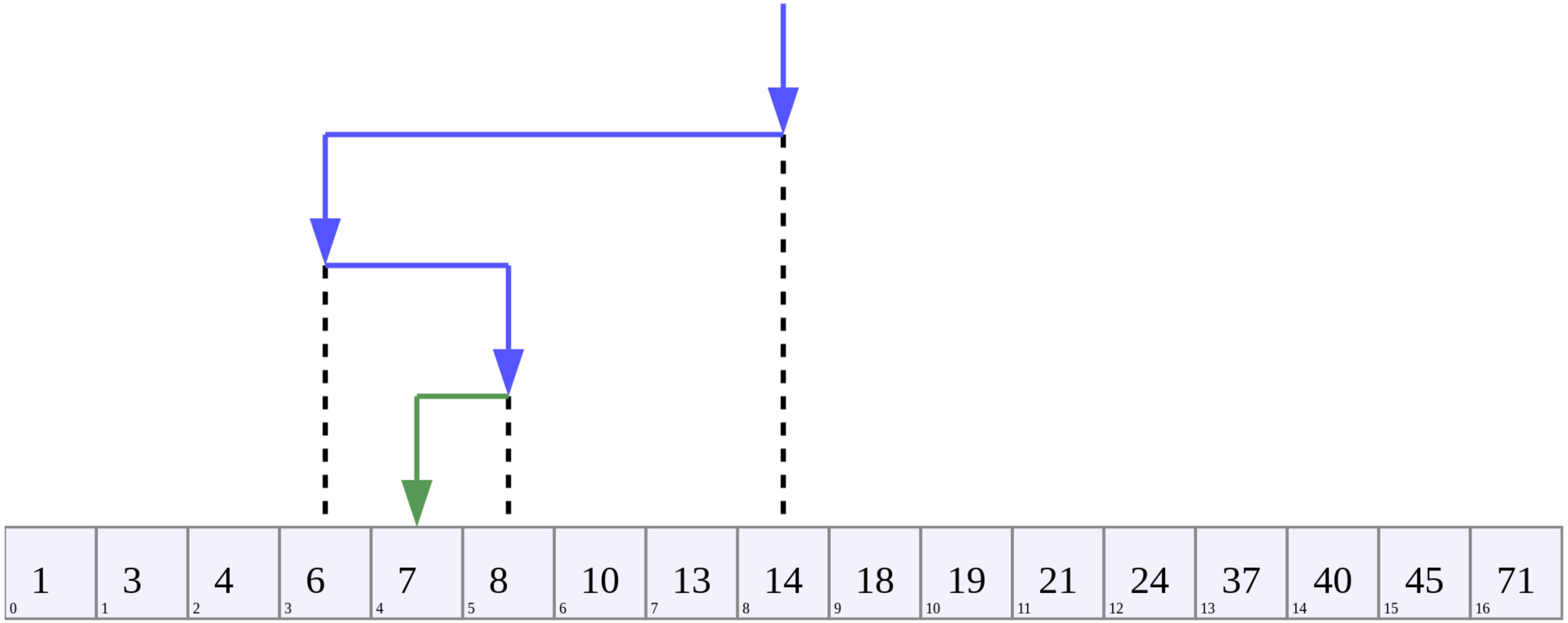
ระยะเวลาที่ใช้ในการประมวลผลคงที่ ไม่เปลี่ยนแปลงไปตามขนาดข้อมูล
ใส่ Input ไปยังไงก็ยังคงใช้ระยะเวลาในการประมวลผลเท่าเดิม



Big O notation (O)

O(log n) **Logarithmic**

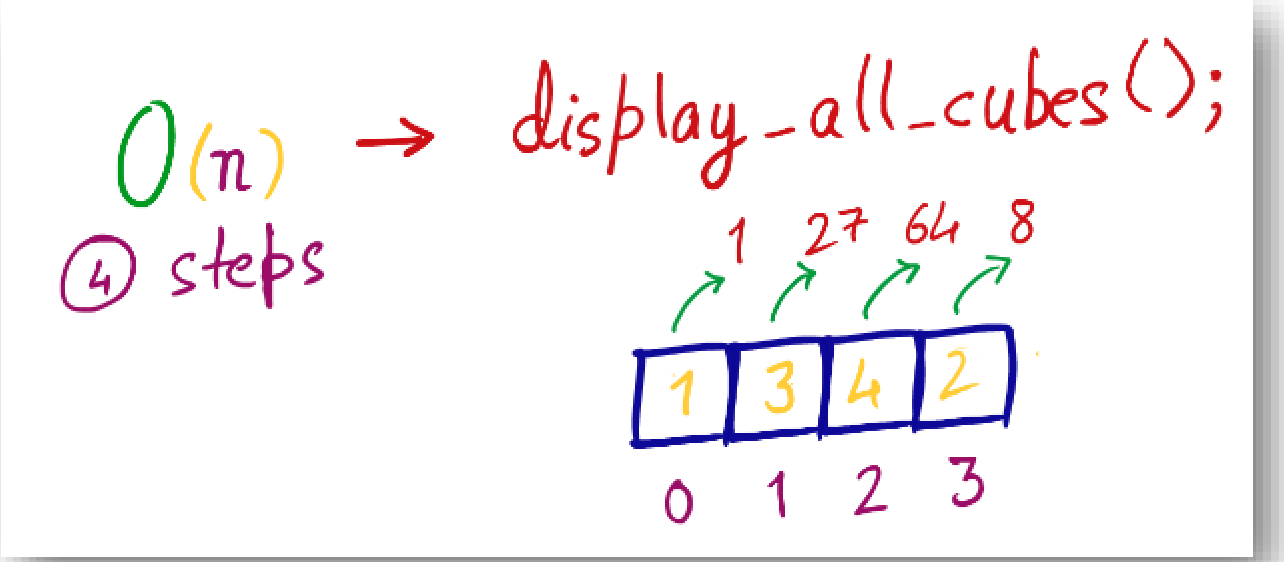
ลดจำนวนที่ไม่มีโอกาสเกิดขึ้นในแต่ละรอบ ทีละครึ่งหนึ่ง ในทุกรอบการประมวลผล



Big O notation (O)

O(n) **Linear**

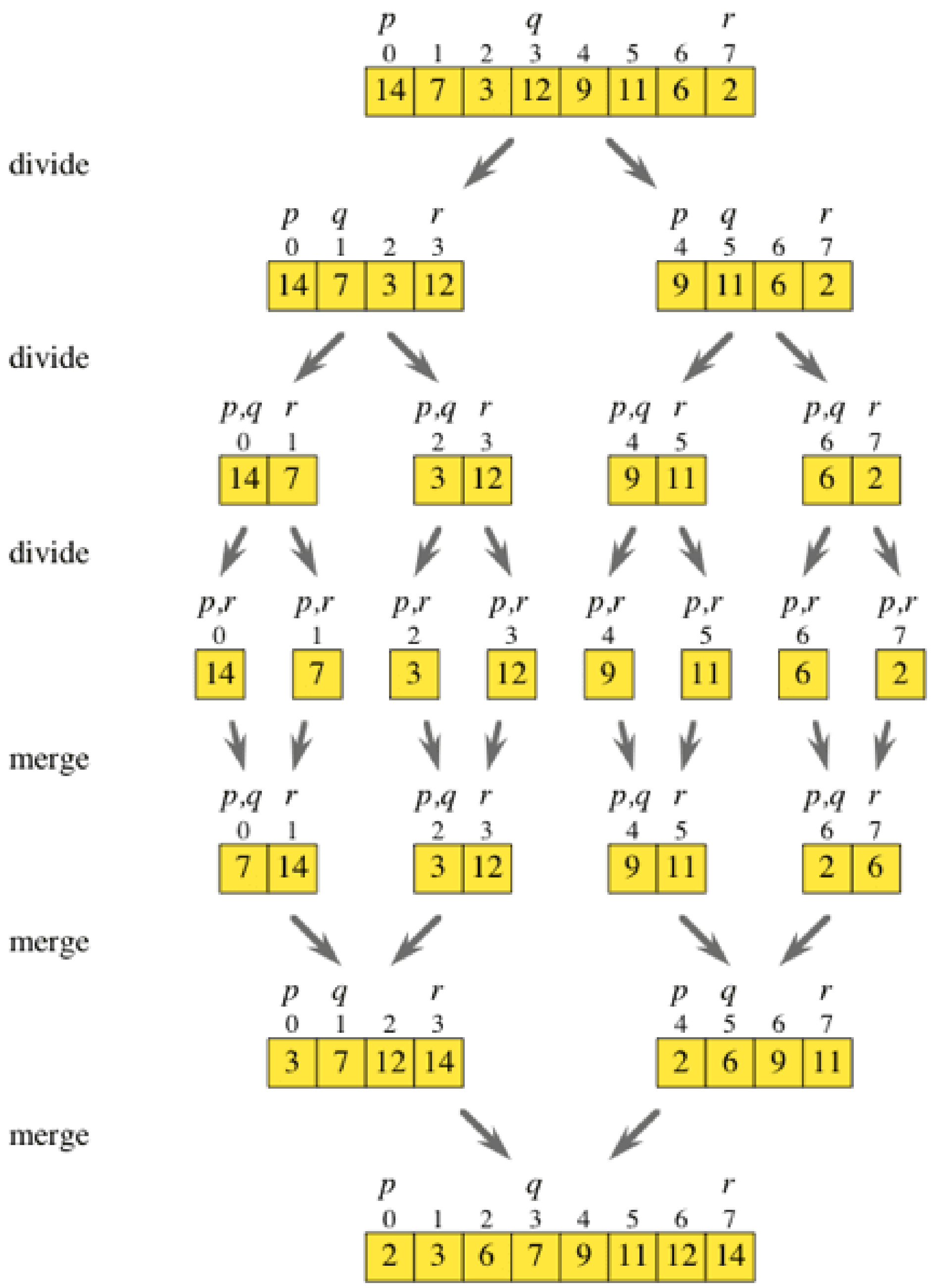
ระยะเวลาที่ใช้ในการประมวลผลขึ้นอยู่กับปริมาณข้อมูลที่ใส่เข้ามาในระบบ



Big O notation (O)

O(n log n)   Linearithmic

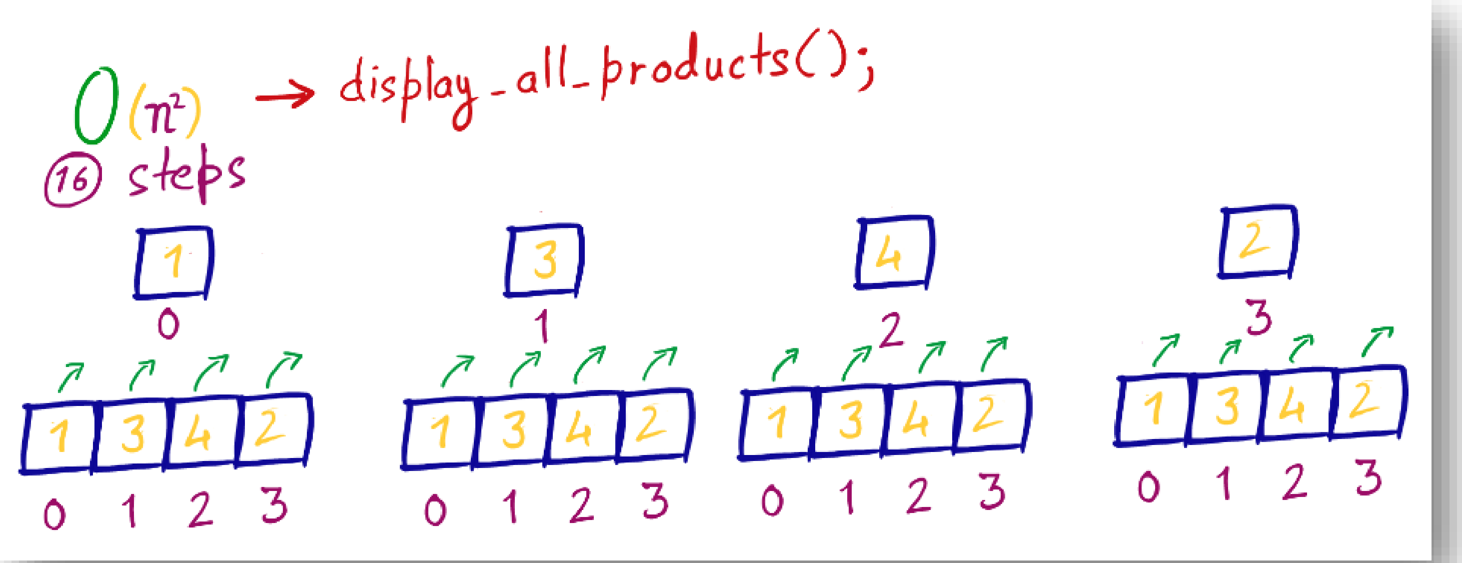
การประมวลผลเป็นไปในลักษณะของการซ้อน Loop ใน Loop ชั้นแรกเป็นไปตามจำนวนข้อมูล ส่วน Loop ด้านในจะลดลงทีละครึ่งหนึ่ง



Big O notation (O)

$O(n^2)$ Quadratic

เป็นการประมวลผลที่ใช้ระยะเวลาเป็นเท่าตัวเมื่อเทียบกับปริมาณของ Input
ความซับซ้อนในการประมวลผลในลักษณะนี้เริ่มมองได้ว่าเริ่มแย่แล้ว~~~

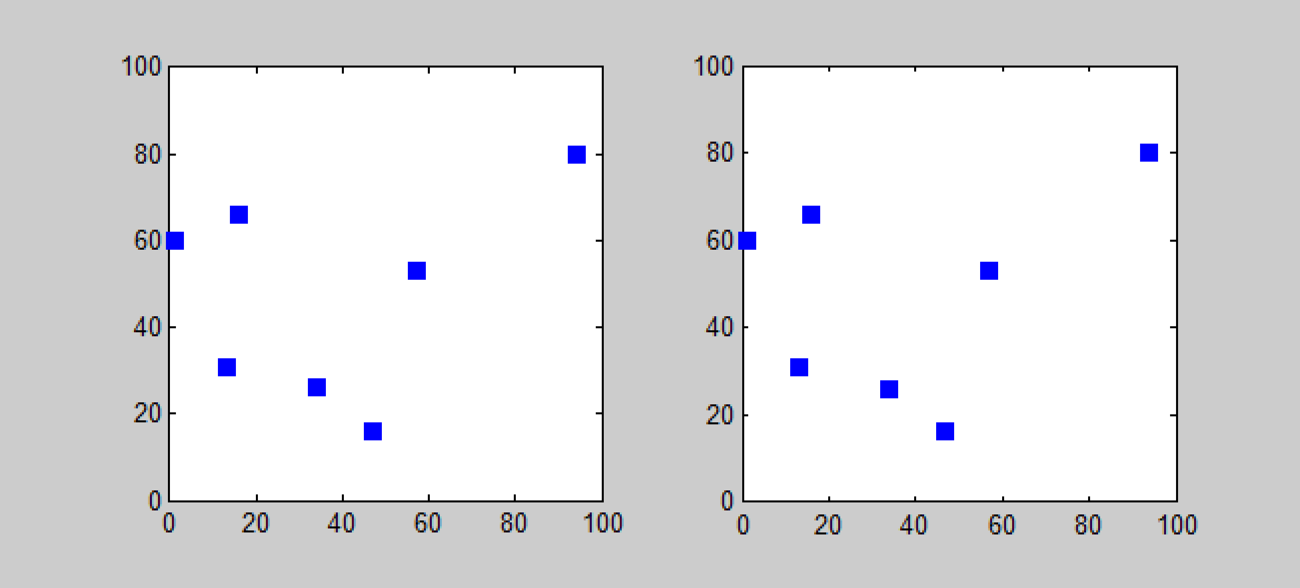


Big O notation (O)

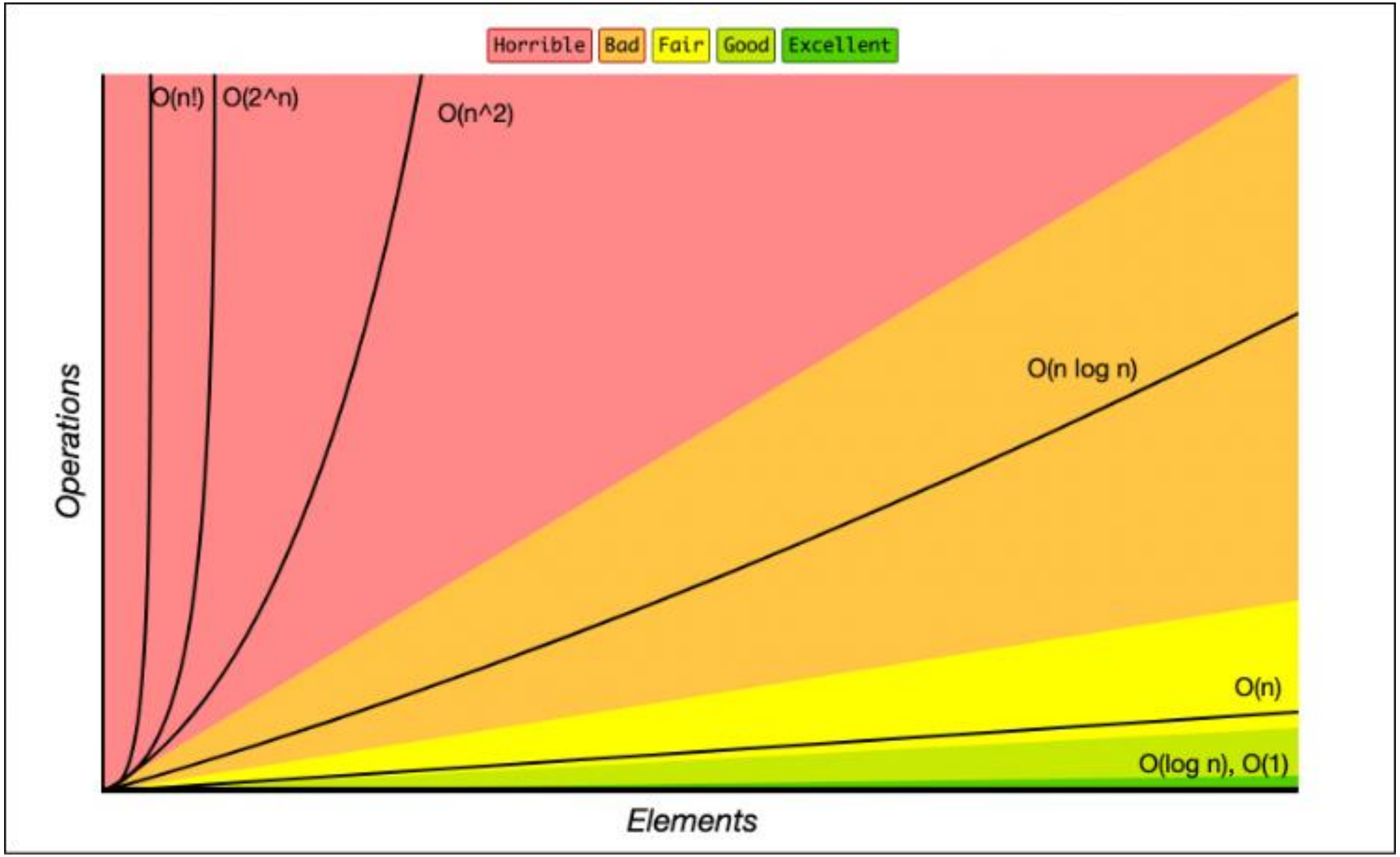
O(n!)   Factorial

เป็นการประมวลผลที่ใช้ระยะเวลาค่อนข้างอยู่ในระดับวิกฤต
ยิ่งประมาณข้อมูลเยอะ สัดส่วนของระยะเวลายิ่งเพิ่มเป็นทวีคูณแบบ Factorial

Travelling salesman problem



Big O notation (O)



Classes	n	*Complexity number of operations (10)*	Execution Time (1 instruction/μsec)
constant	*O(1)*	*1*	*1 μsec*
logarithmic	*O(logn)*	*3.32*	*3 μsec*
linear	*O(n)*	*10*	*10 μsec*
O(nlogn)	*O(nlogn)*	*33.2*	*33 μsec*
quadratic	$O(n^2)$	10^2	*100 μsec*
cubic	$O(n^3)$	10^3	*1msec*
exponential	$O(2^n)$	*1024*	*10 msec*
factorial	*O(n!)*	*10!*	*3.6288 sec*

If computer executing **1 million operation per second**

Guidelines for asymptotic analysis

การวิเคราะห์ Loop

การระบุปัจจัยที่ส่งผลกระทบกับ Run time

การนับจำนวนครั้งของ Operation

- **Arithmetic Operators (+, -, *, /, %)**
- **Relational Operators (==, !=, >, <, >=, <=)**
- **Assignment Operators (=, +=, -=, *=)**
- **Bitwise Operators (&, |, ^, ~, >>, <<)**

Guidelines for asymptotic analysis

Loops

The running time of a loop is, at most, the running time of the statements inside the loop, including tests) multiplied number of iterations

```
for i in range(0, n):
print ('Current Number:', i, sep = "")
```

Constant time (c) : $c \; x \; n \quad = \quad O(n)$

Guidelines for asymptotic analysis

Nested Loops

Analyze from the inside out. The total running time is the product of the sizes of all the loops.

```
for i in range(0, n):

    # inner loop executes n times
    for j in range(0, n):
      print("i value % d  and j value % d" % (i, j))
```

Constant time (c) : $c \ x \ n \ x \ n \quad = \quad O(n^2)$

Guidelines for asymptotic analysis

Consecutive statements

Add the time complexity of each statement

```
n = 100

# executes n times
for i in range (0, n):
    print (Current Number: i, sep = "")

    # outer loop executed n times
    for i in range (0, n):

      # inner loop executes n times
      for j in range(0, n):
        print(" i value % d and j value % d"%(i, j))
```

$$c_0 + c_1 n + c_2 n^2 \quad = \quad O(n^2)$$

Guidelines for asymptotic analysis

If-then-else statements

Worst-case running time: the test, plus either the then part of the else part whichever is the largest.

```
if n == I:
  print ("Incorrect Value")
  print (n)

else:
  for i in range(0, n):

    # constant time
    print (CurrNumber:, i, sep = "")
```

$$c_0 + c_1 n \quad = \quad O(n)$$

Guidelines for asymptotic analysis

Logarithmic Time Complexity

The time Complexity of a loop is considered as $O(\log n)$ **if the loop variables are divided/multiplied by a constant amount**. And also, for **recursive calls** in the **recursive function**, the Time Complexity is considered as $O(\log n)$.

```
i = 1
while(i <= n):
    # some O(1) expressions
    i = i*c

i = n
while(i > 0):
    # some O(1) expressions
    i = i//c
```

```
# Recursive function
def recurse(n):
    if(n <= 0):
        return
    else:
        # some O(1) expressions
    recurse(n/c)
```

Example

(ส่วนหนึ่งของโปรแกรม)

`int sum = 0;`	1	=
`int i = 0;`	1	=
`while (i < n)  {`	1	<
`sum = sum + i;`	2	= และ +
`i++;`	1	++
`}`		

ทำไม Loop ถึงเป็น n + 1
ทำไมถึงไม่ใช่ n เฉยๆ

loop	i
3	1
2	2
1	3
0	-

$$T(n) = 1 + 1 + (4n + 1)$$
$$= 4n + 3$$

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

$$\boldsymbol{O(n)}$$

Example

```
def funct(n):

    if (n==1):
        return
    for i in range(1, n+1):
        for j in range(1, n + 1):
            print("*", end = "")
            break
      print()
```

$O(n)$

Example

```
def function(n):
    count = 0

    # outer loop executes n/2 times
    for i in range(n//2, n+1):

        # middle loop executes  n/2 times
        for j in range((1, n//2 + 1):

            # inner loop executes logn times
            for k in range(1, n+1, 2):
                count++
```

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

Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Stack	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Queue	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Singly-Linked List	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Doubly-Linked List	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Skip List	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n \log(n))$
Hash Table	N/A	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	N/A	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Binary Search Tree	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Cartesian Tree	N/A	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	N/A	$O(n)$	$O(n)$	$O(n)$	$O(n)$
B-Tree	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$
Red-Black Tree	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$
Splay Tree	N/A	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	N/A	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$
AVL Tree	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$
KD Tree	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$

Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n^2)$	$O(\log(n))$
Mergesort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(n)$
Timsort	$\Omega(n)$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(n)$
Heapsort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(1)$
Bubble Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
Insertion Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
Selection Sort	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
Tree Sort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n^2)$	$O(n)$
Shell Sort	$\Omega(n \log(n))$	$\Theta(n(\log(n))^2)$	$O(n(\log(n))^2)$	$O(1)$
Bucket Sort	$\Omega(n+k)$	$\Theta(n+k)$	$O(n^2)$	$O(n)$
Radix Sort	$\Omega(nk)$	$\Theta(nk)$	$O(nk)$	$O(n+k)$
Counting Sort	$\Omega(n+k)$	$\Theta(n+k)$	$O(n+k)$	$O(k)$
Cubesort	$\Omega(n)$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(n)$

Classwork

Write a python program to read the **"Thai address"** data set

- Working as a **group**, use **GitHub** to do this work
- For now, you can should any data structure to collect the data
- However, you **must separate each data by " , "**
- Keep in mind that we need to **use** this data in the future

Reading resource

https://www.geeksforgeeks.org/python-string-methods/


Complexity is the enemy of execution!
Tony Robbins
quotefancy