University of Science, VNU-HCM
Faculty of Information Technology

Data Structure and Algorithm

Analysis of Algorithms

Lecturer: Le Ngoc Thanh
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 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...

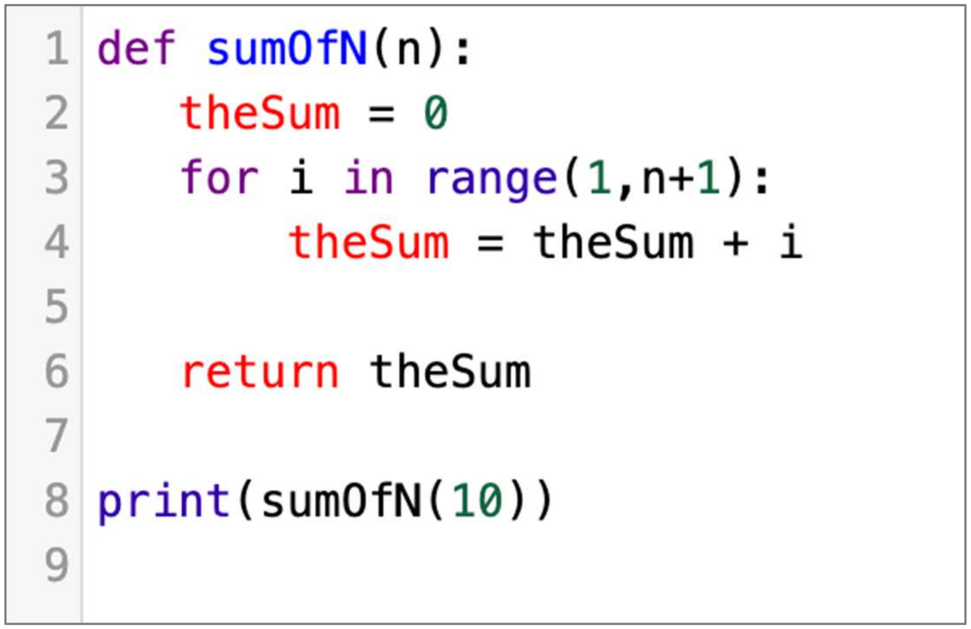
- *Why not optimize the only one way to go?*
  - Depends on circumstances, conditions, availability, convenience, comfort, …

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?
  - …

Algorithms for a problem

- Write a progam to calculate summation of the first n integers

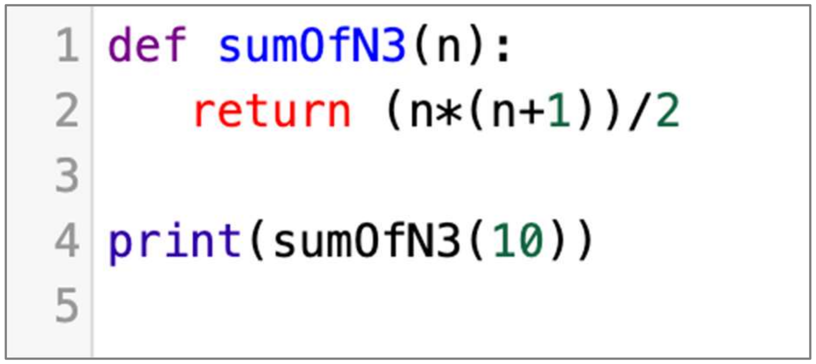


```python
def sumOfN(n):
    theSum = 0
    for i in range(1,n+1):
        theSum = theSum + i

    return theSum

print(sumOfN(10))
```

Algorithm 1



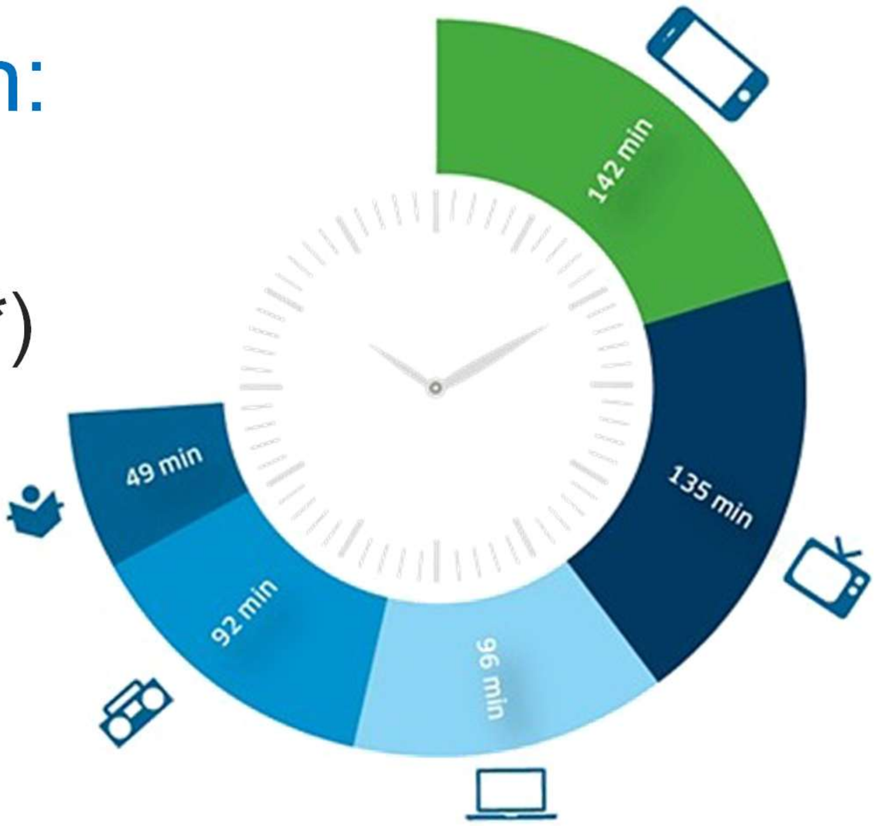
```python
def sumOfN3(n):
    return (n*(n+1))/2

print(sumOfN3(10))
```

Algorithm 2

Object of algorithm analysis

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



Questions

- Which of the following best describes the useful criterion for comparing the efficiency of algorithms?

A) Time

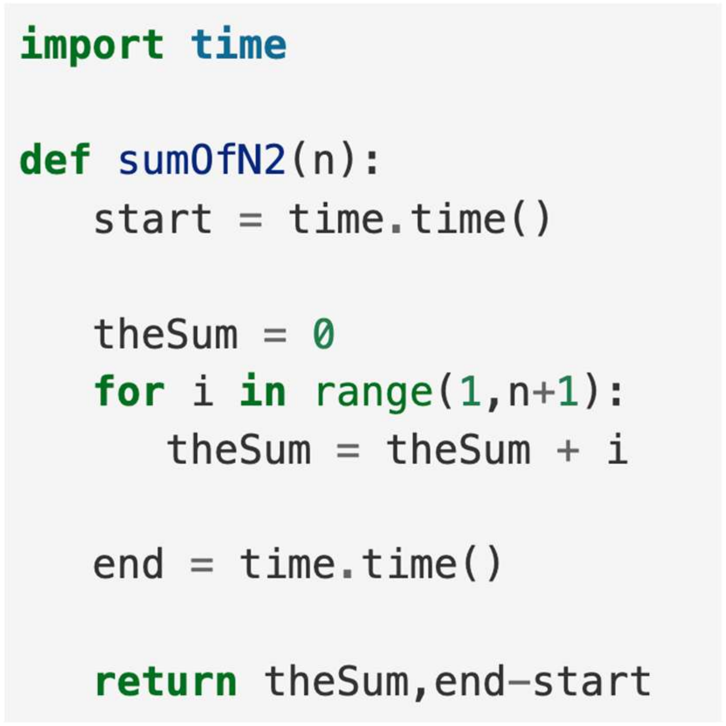
B) Memory

C) Both of above

D) None of abve

How to compare algorithms?

- Measurement:
  - Run time?
    - Not good because it depends on the configuration of each computer.



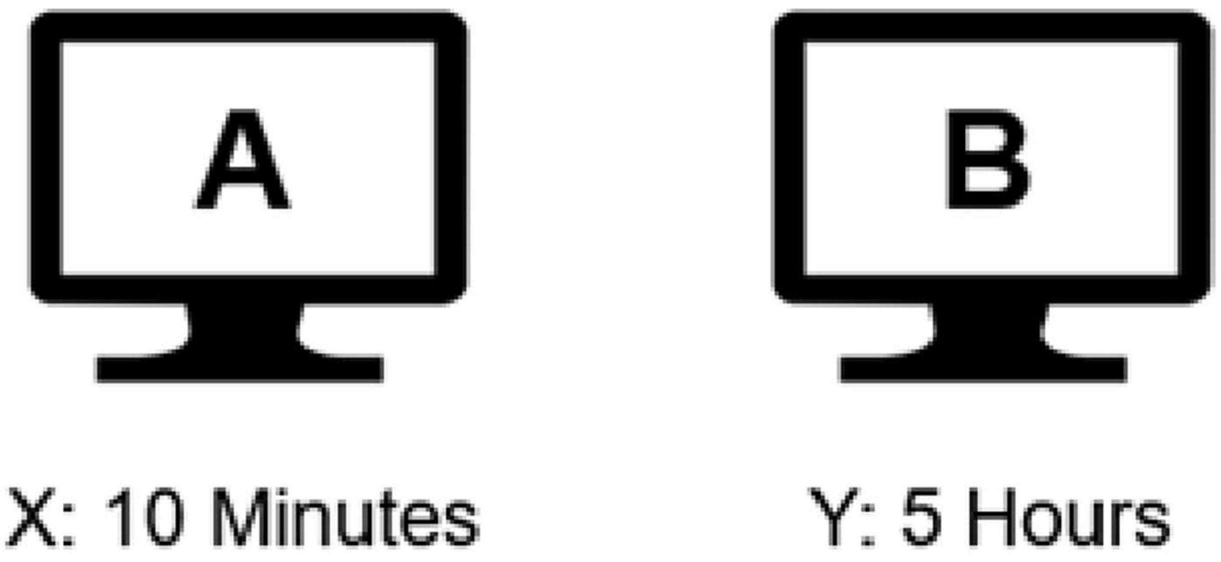
```
import time

def sumOfN2(n):
    start = time.time()

    theSum = 0
    for i in range(1,n+1):
        theSum = theSum + i

    end = time.time()

    return theSum,end-start
```



How to compare algorithms?

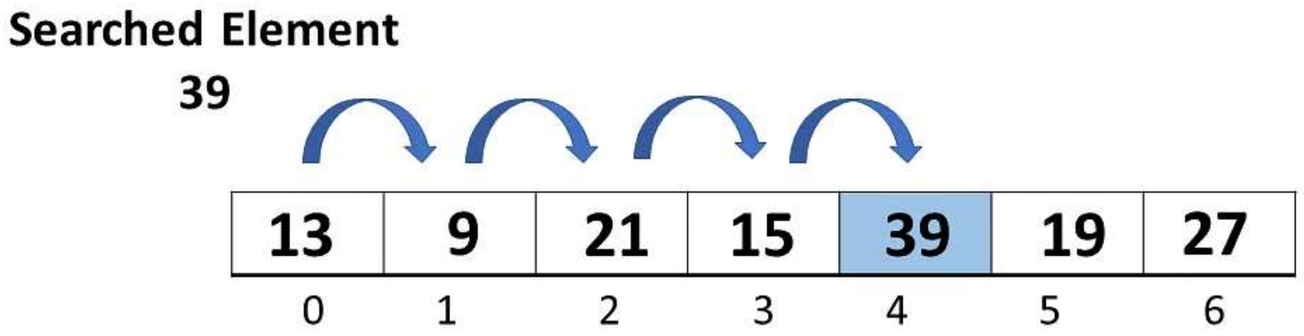
- Measurement:
  - Run time?
    - Not good because it depends on the configuration of each computer.
  - Number of statements are performed?
    - Not good because it depends on the programming language as well as the style of each programmer.
- So is there any way to compare independently of the machine,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

$$f(n) = \cdots$$

The basic question

If there are N data elements, how many steps will the algorithm take?

If the array contains 22 elements, linear search takes ? steps.

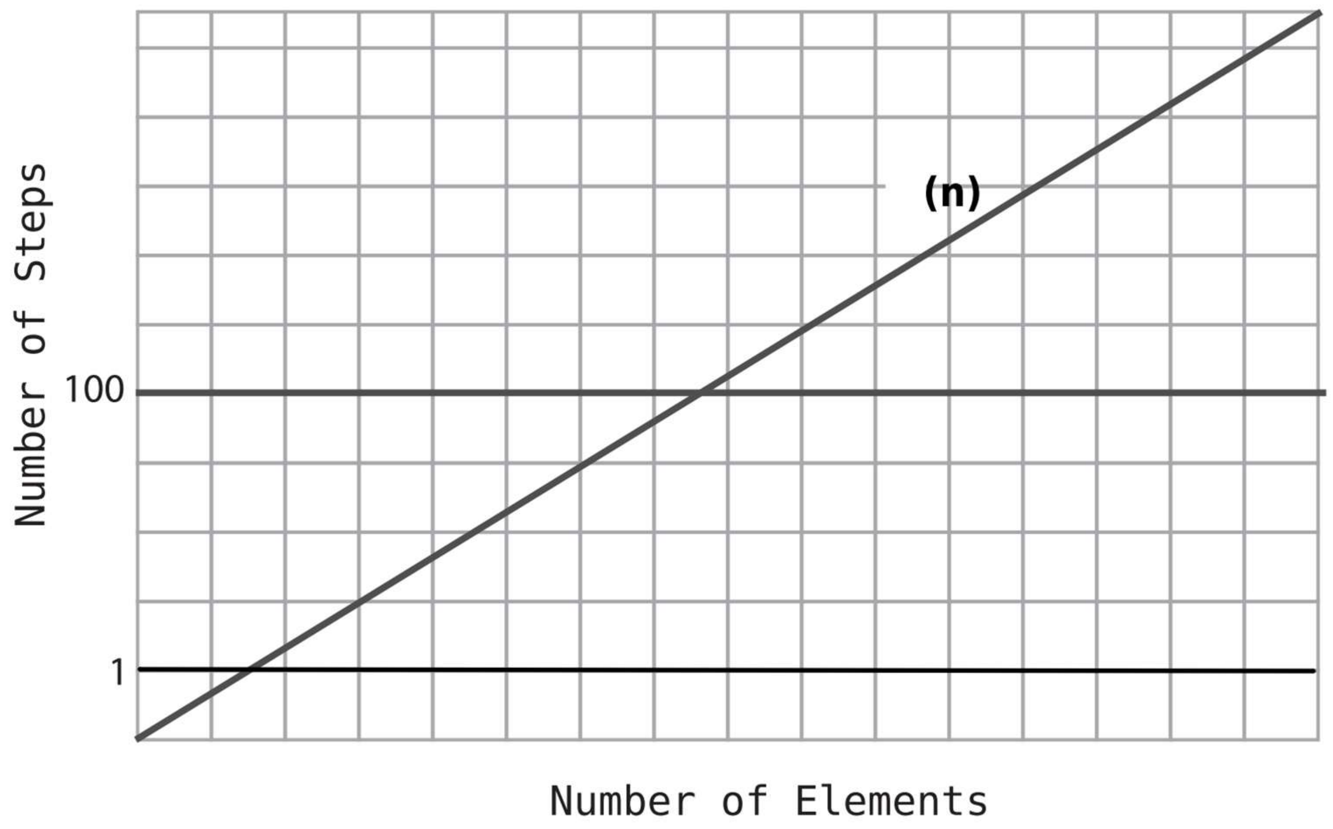


The basic question

- Assume that the final element is at the last position, if:
  - The array contains 22 elements, linear search takes ? steps.
  - The array contains 100 elements, linear search takes ? steps.
  - The array contains 1000 elements, linear search takes ? steps.
  - The array contains N elements, linear search takes ? steps.

The basic question

- Assume that we have two other algorithm (ever 1 and ever 100) to search an element in the array. Which one is the better?



=> We don’t care how many steps an algorithm takes, we care how the number of steps increase as the data changes.

Soul of analysis

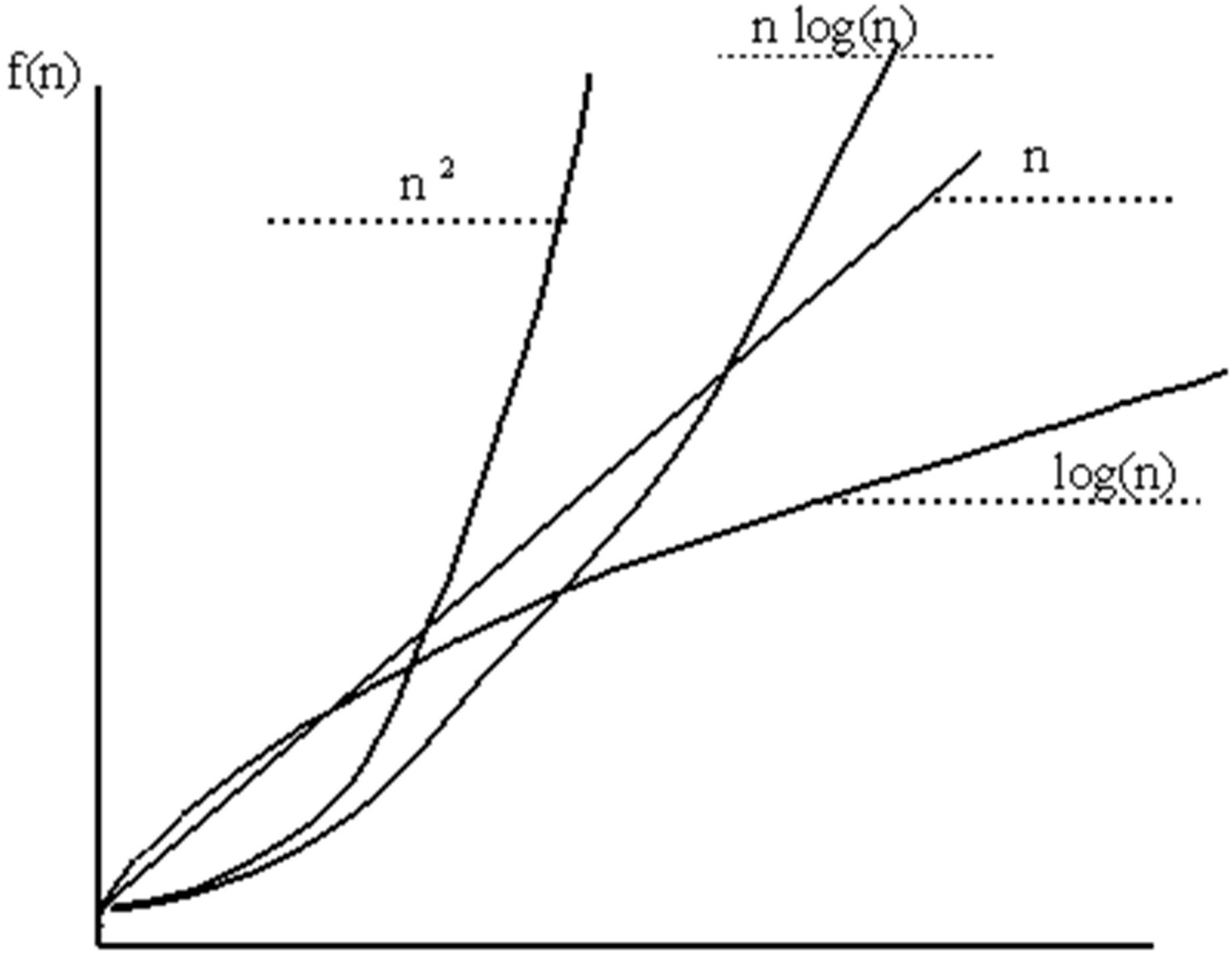
- Size of the problem:
  - – Consider how the processing time increases as the size of the problem increases.
  - – The size of the problem usually depends on the size of the input:
    - Array size
    - Number of matrix elements
    - Number of bits expressed by input
    - Number of vertexes and edges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

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



Growth rate

- The order of the functions from slowest to fastest are:
  - – Constant Functions (E.g. 3)
  - – Logarithmic Functions (E.g. logn)
  - – Linear Functions (E.g. n)
  - – Quadratic Functions (E.g. n^2)
  - – Cubic Functions (E.g. n^3)
  - – Exponential Functions (E.g. 2^n)
  - – n^n kind of functions (E.g. n^n)

Questions

- Arrange the following expressions by growth rate from slowest to fastest.

  4n^2, log3n, n!, 3n, 20n, 2, log2n, n^2/3

Questions

- Fill number of operations for each following functions:

n	$logn$	n	$nlogn$	n^2	n^3	2^n	$n!$
10							
100							
1000							
10^4							
10^6							

- If CPU can process 10^10 operation/sec, how long does finish programs with above functions, respectively?

Questions

Which of the following functions grows fastest?

(a) $n \log n$

(b) 2^n

(c) $\log n$

(d) n^2

(e) n^{20}

Questions

Which of the following functions grows fastest?

(a) $n + \log n$

(b) $n \log n$

(c) $n - \log n$

(d) n

Question

- A recursive algorithm works by solving two half-sized problems recursively, with an additional linear-time overhead. The total running time is most accurately given by

a) O(log n)

b) O(n)

c) O(n log n)

d) O(n^2)

e) none of the above

Same algorithm, different scenarios

- *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

- Depends not only on input size, *but the layout, structure, type* of input data
  → the growth rate is also different.

Scenarios

- There are three main types of scenarios:
  - 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.

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

Same algorithm, different scenarios

- *Number of steps to find value 1 in the following array?*

1	25	6	5	2	37	40

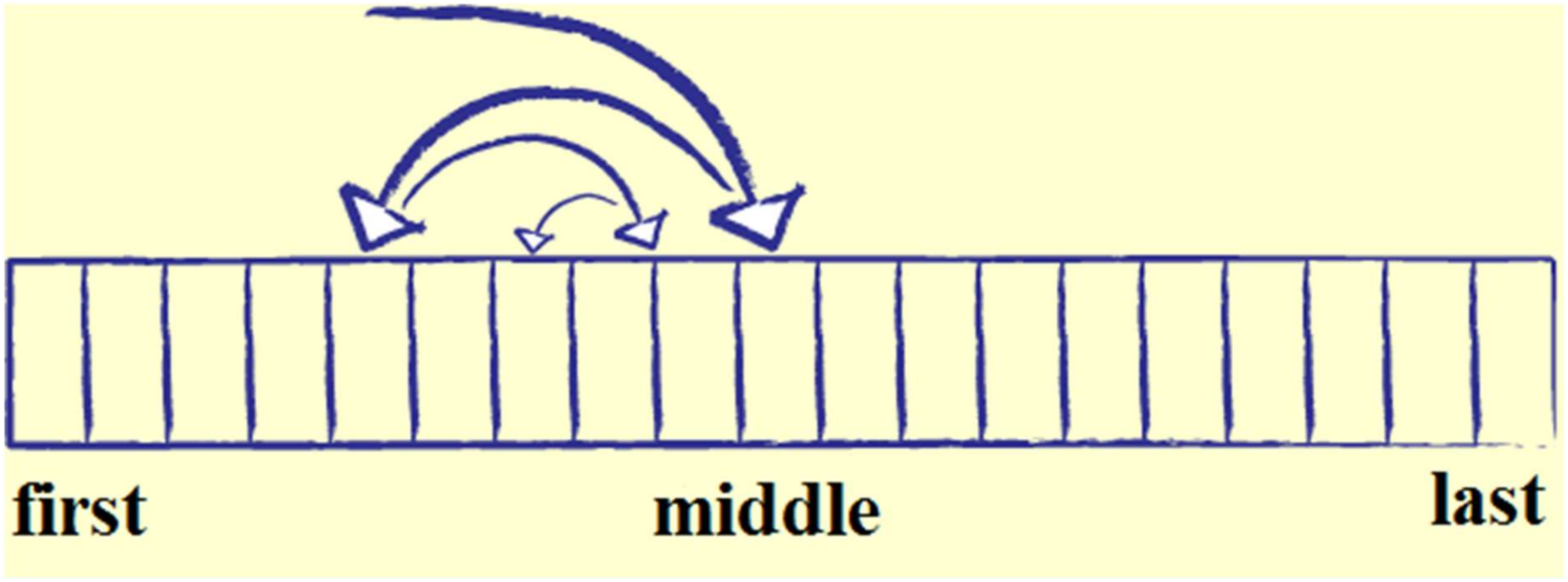
40	25	6	5	2	37	1

- Big-Ω, Big-O?

=> We can conlude that the algorithm’s time is between (1) and (N).

Case study: Binary search

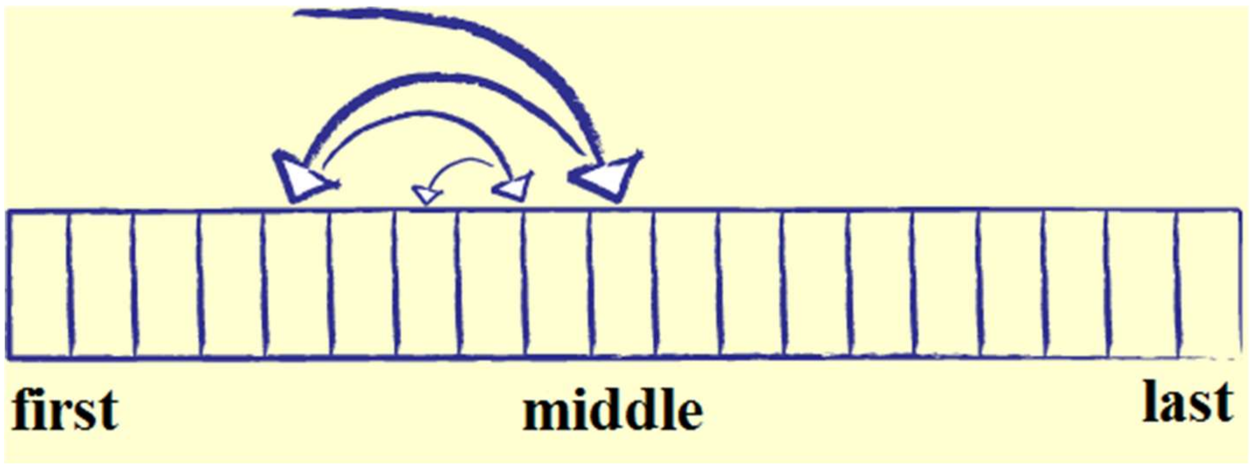
- Number of steps to find a value with binary search? Big-Ω, Big-O?



$(1) - \log_2(N)$

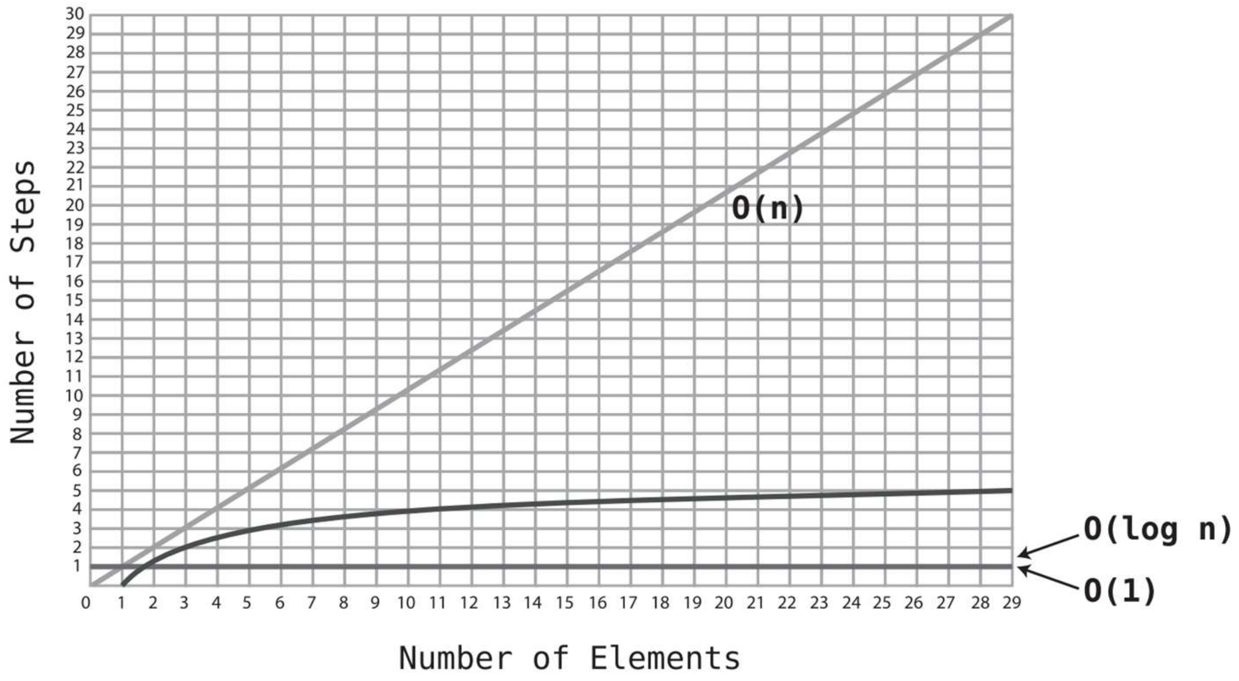
Case study: Binary search

- What mean of $\log_2(N)$?
  - Increases one step each time the data is doubled.
- Why is $\log_2(N)$?
  - Logarithms are the inverse of exponents.
  - It means: how many times do you have to multiply 2 by itself to get a result of N.
  - Another way to explain: how many times do we need to halve N until we end up with 1.



O(N) vs O(logN)

N Elements	O(N)	O(log N)
8	8	3
16	16	4
32	32	5
64	64	6
128	128	7
256	256	8
512	512	9
1024	1024	10



Big-O

- 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 $\mathrm{c}g(n)$ (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 $f(n)$

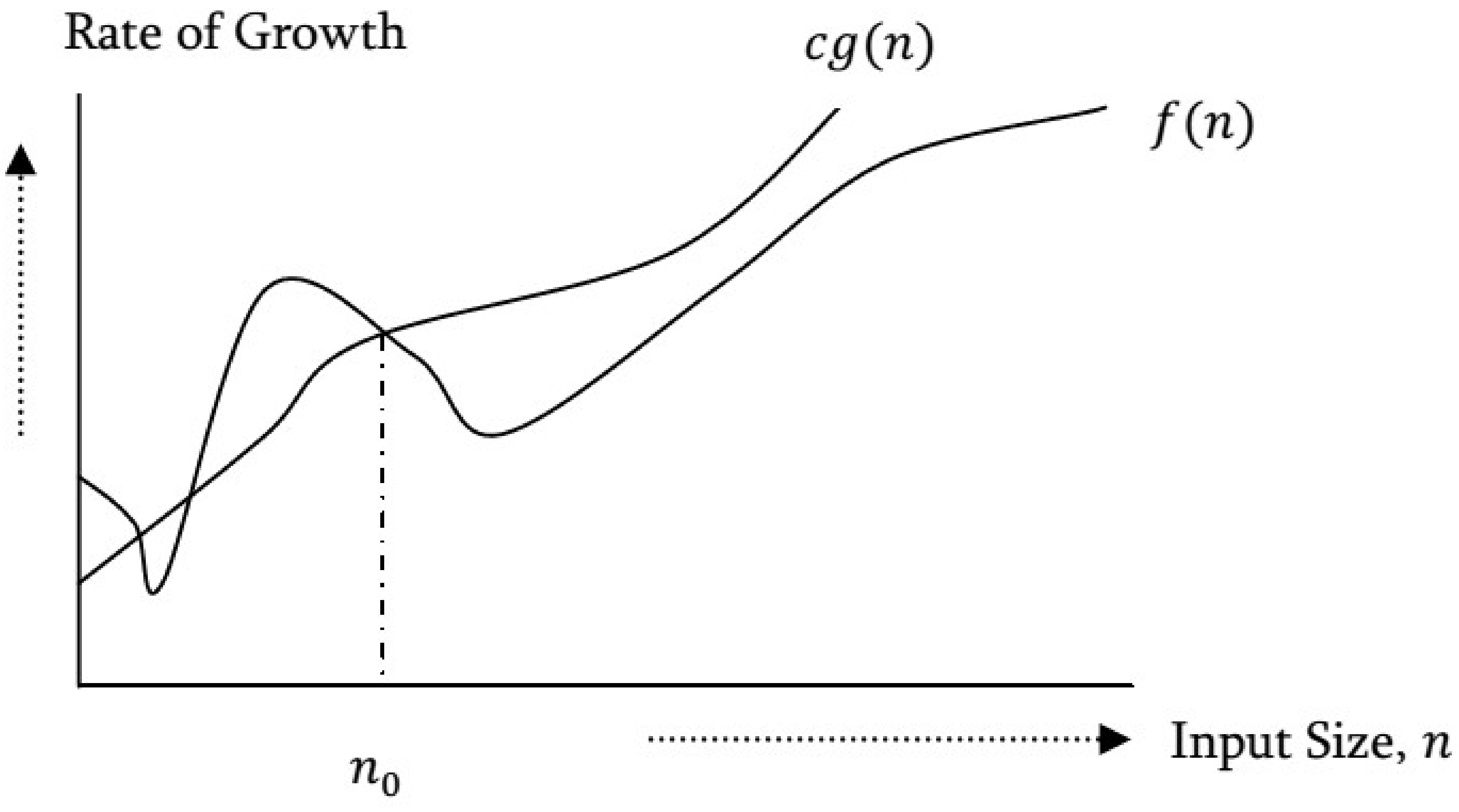
Questions

- Algorithm A and B have a worst-case running time of O(n) and O(logn), respectively. Therefore, algorithm B always runs faster than algorithm A?

A) True

B) False

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.

Questions

- What does it mean when we say that an algorithm X is asymptotically more efficient than Y?

A) X will always be a better choice for small inputs

B) X will always be a better choice for large inputs

C) Y will always be a better choice for small inputs

D) X will always be a better choice for all inputs

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?
- Answer:
  - 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 g(n) 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.

$$\Omega$$

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

Big-Ω

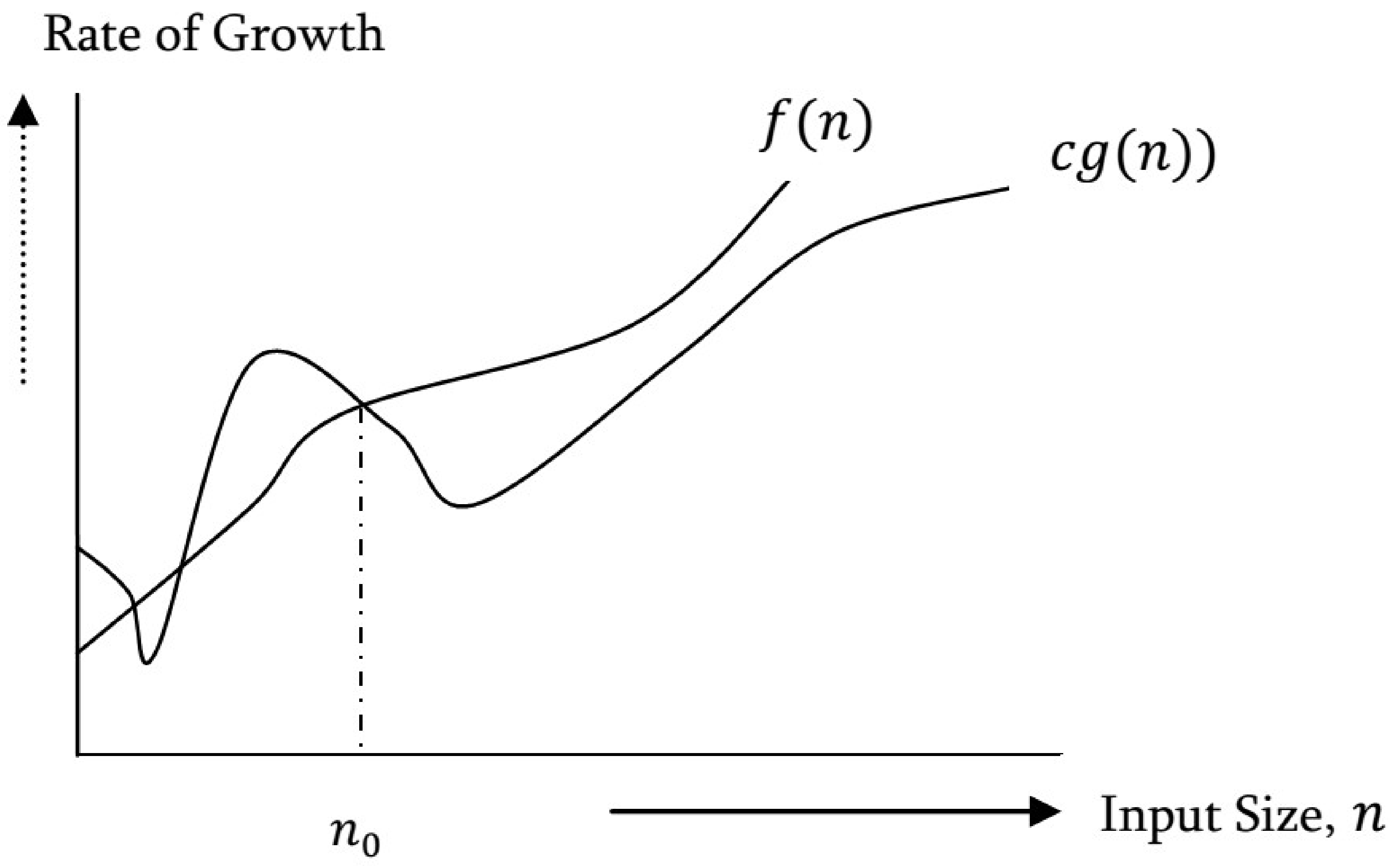
- Definition:
  - Given f(n) and g(n) are two functions
  - $f(n) = \Omega(g(n))$, if there exists positive numbers c and n_0 so that:

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

    where $n \geq n_0$
  - 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-Θ

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

$$\boldsymbol{Lower\ Bound\ (\Omega)\ \leq Average\ Time\ \leq Upper\ Bound(O)}$$

$$\boldsymbol{\Theta}$$

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

Big-Θ

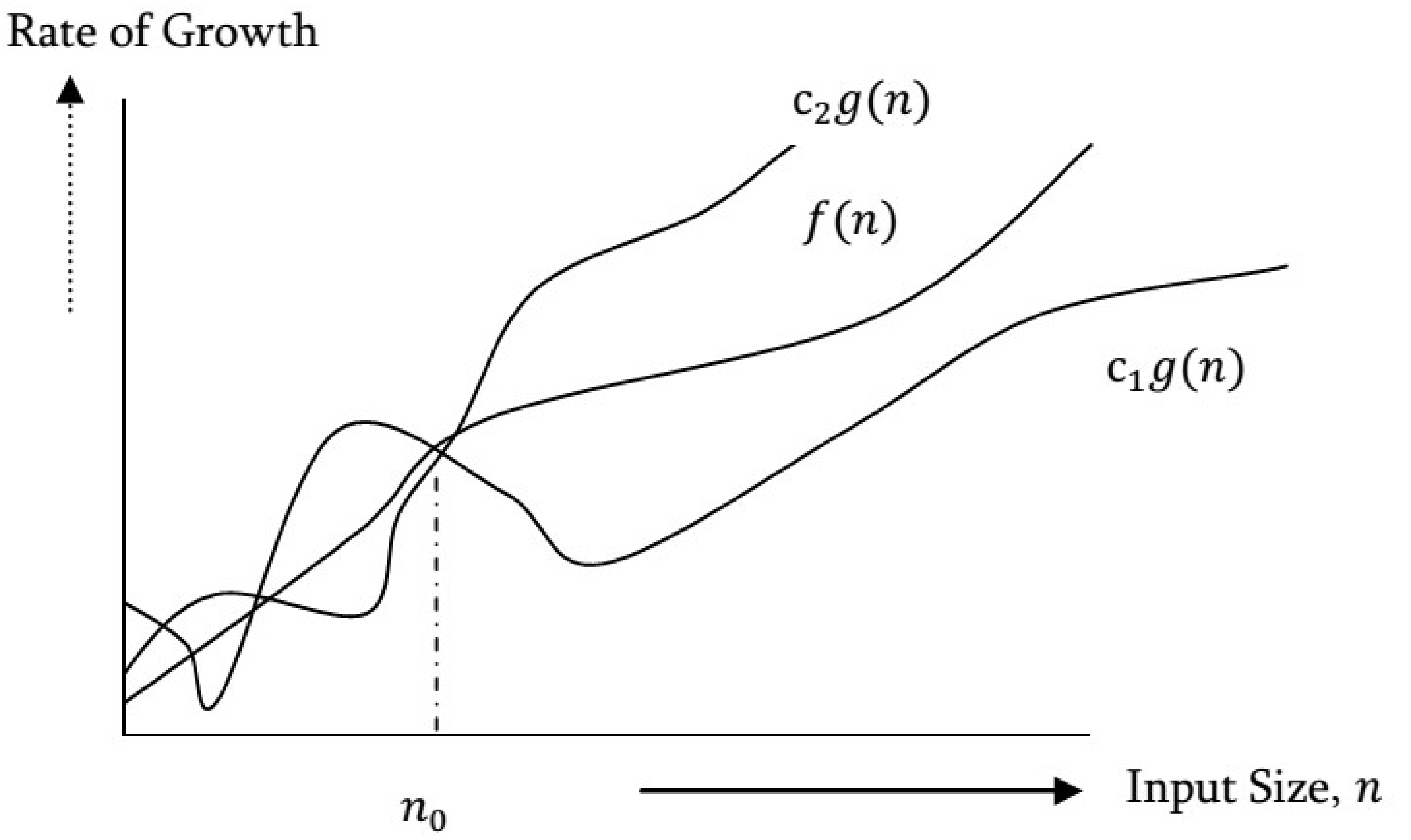
- Definition:
  - Given f(n) and g(n) are two functions
  - $f(n) = \Theta(g(n))$, if there exists positive numbers c and n_0 so that:

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

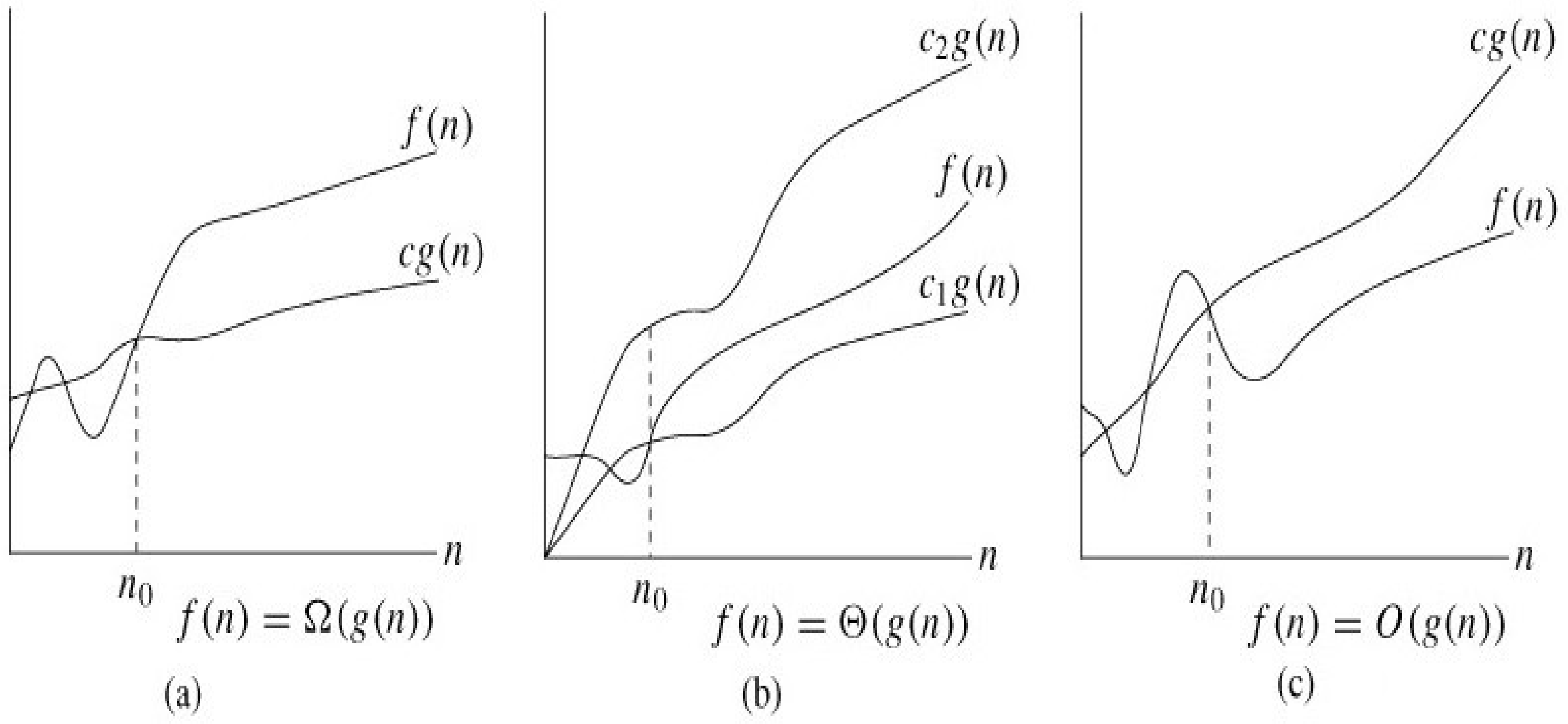
where $n \geq n_0$

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

Big-Θ



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.

Question

- An algorithm takes 10 seconds for an input size of 50.
- If the algorithm is quadratic, approximately how long does it take to solve a problem of size 100?

A) 10 seconds

B) 20 seconds

C) 40 seconds

D) 100 seconds

E) None of above

Question

- An algorithm takes thirty seconds for an input of size 1000.
- If the algorithm is quadratic, how large a problem can be solved in two minutes?

A) 2000

B) 4000

C) 6000

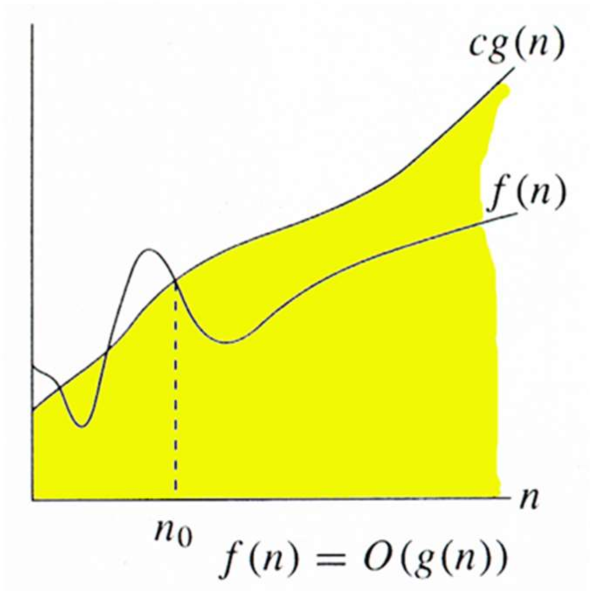
D) 60000

E) None of above

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



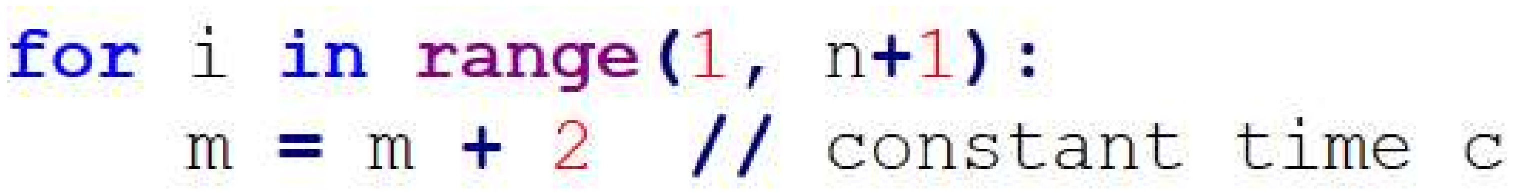
Primitive operation

- Primitive operations are the following:
  - Assigning a value to a variable* (=)
  - Arithmetic operations (+, -, *, …)
  - Logical operations (||, &&, !, …)
  - Comparisons (==, >, < …)
  - Some more…
- We consider that primitive instructions primitive require the same amount of time (even though they actually don't exactly).

Asymptotic analysis – Loop

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

For example:



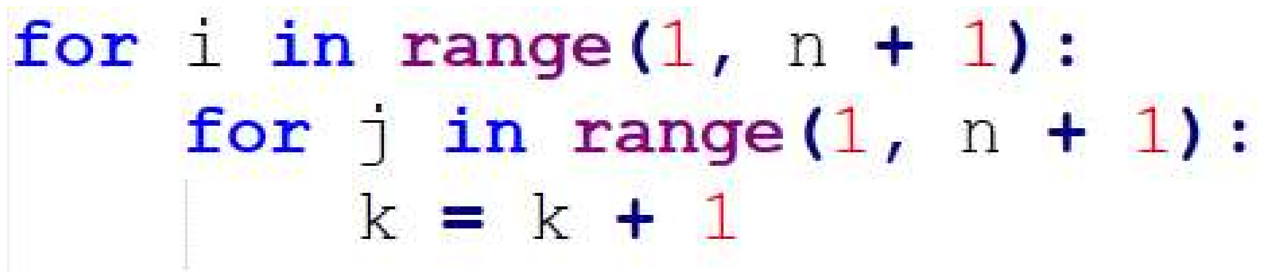
```
for i in range(1, n+1):
    m = m + 2  // constant time c
```

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:



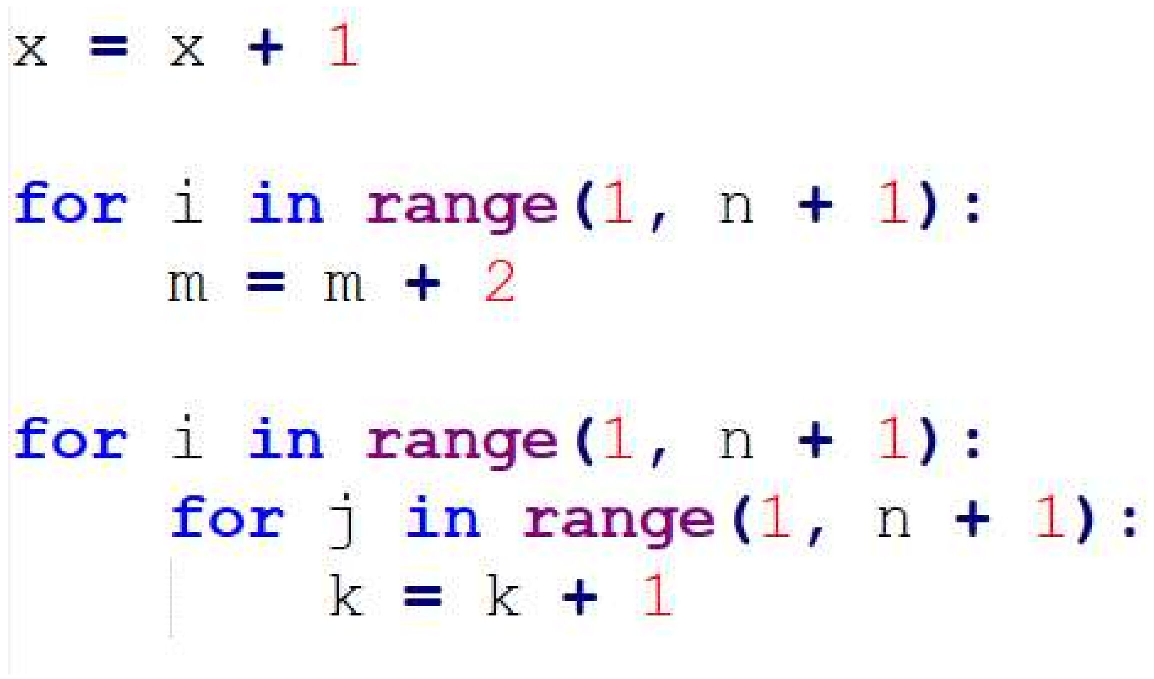
```
for i in range(1, n + 1):
    for j in range(1, n + 1):
        k = k + 1
```

Run time = $c \times n \times n = cn^2 = O(n^2)$

Sequential instructions

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

Example:



```
x = x + 1

for i in range(1, n + 1):
    m = m + 2

for i in range(1, n + 1):
    for j in range(1, n + 1):
        k = k + 1
```

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

Conditional statement

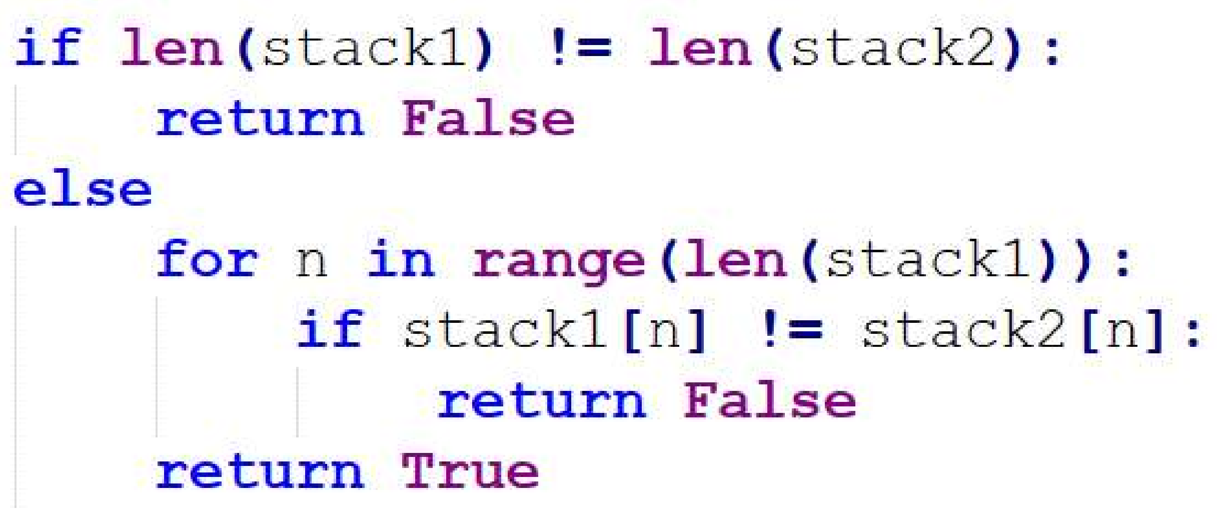
Usually we consider the worst case scenario, the run time is equal to the longest part of the conditional statement.

```
if ( condition )        →    $T_0(n)$
        Statement 1     →    $T_1(n)$
else
        Statement 2     →    $T_2(n)$
```

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

Conditional statement

Example:

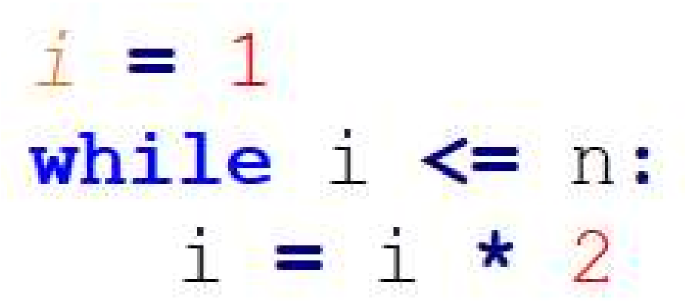


```
if len(stack1) != len(stack2):
    return False
else
    for n in range(len(stack1)):
        if stack1[n] != stack2[n]:
            return False
    return True
```

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

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:



```
i = 1
while 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?

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

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

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

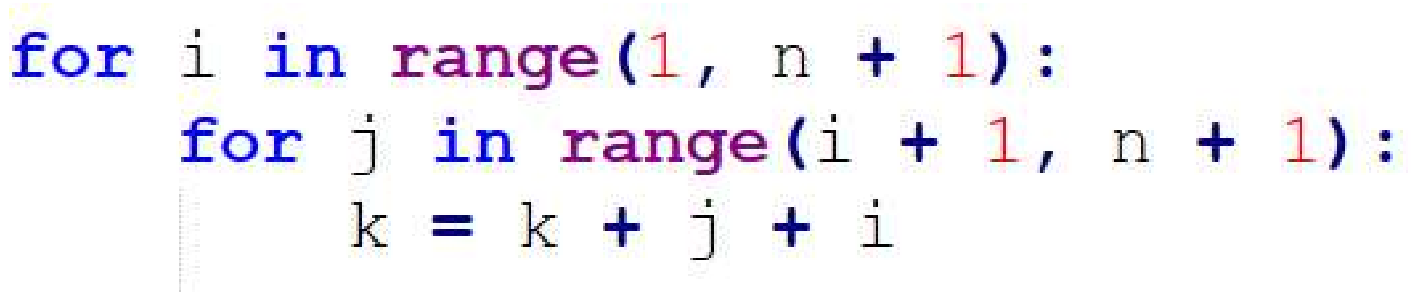
Other important formulae

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

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

Exercise

- Calculate Big-O for the following algorithm:

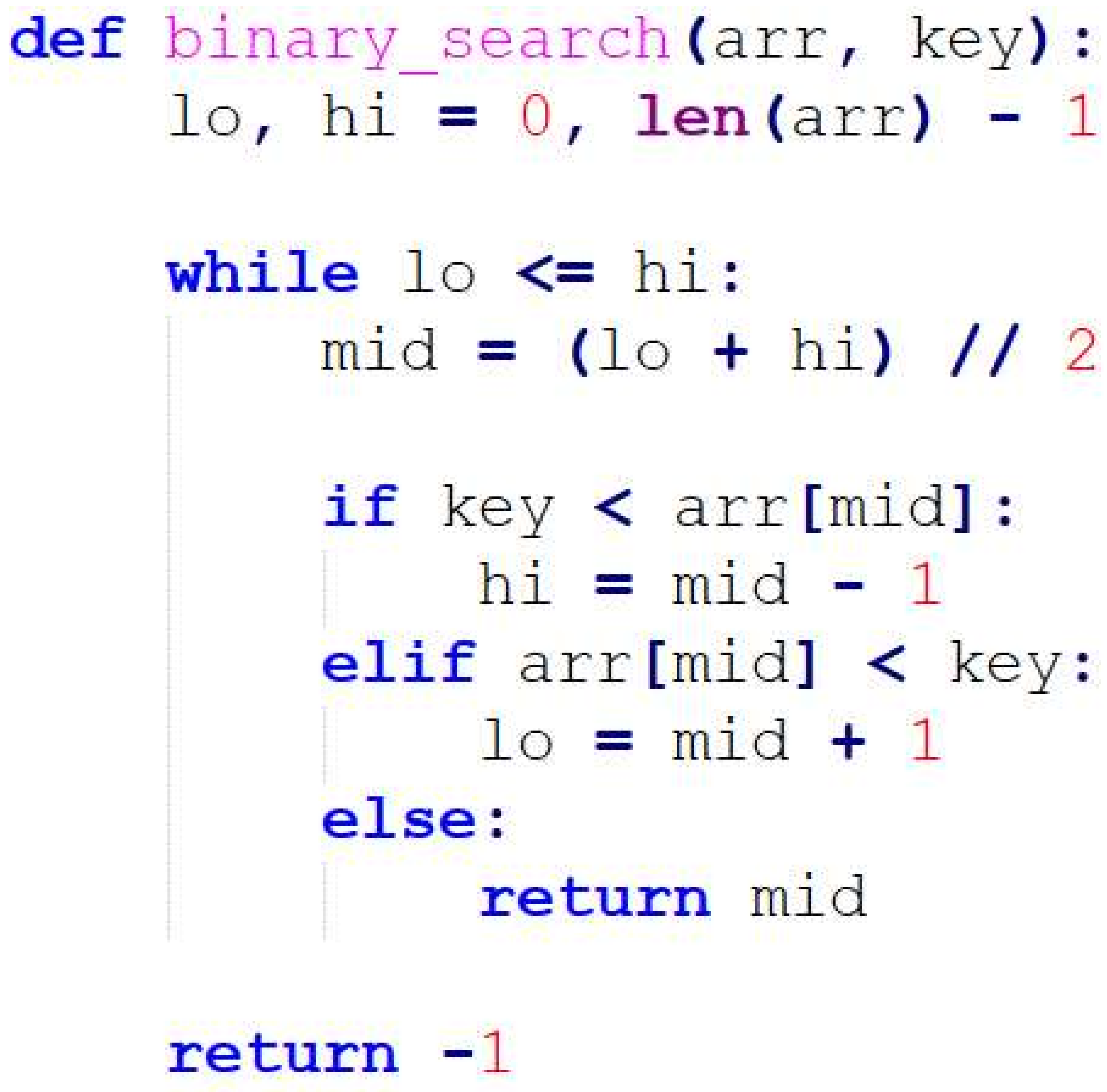


```
for i in range(1, n + 1):
    for j in range(i + 1, n + 1):
        k = k + j + i
```

Exercises

```
for i in range(4, n):
    sum_value = a[i - 4]
    for j in range(i - 3, i + 1):
        sum_value += a[j]
```

Exercises



```python
def binary_search(arr, key):
    lo, hi = 0, len(arr) - 1

    while lo <= hi:
        mid = (lo + hi) // 2

        if key < arr[mid]:
            hi = mid - 1
        elif arr[mid] < key:
            lo = mid + 1
        else:
            return mid

    return -1
```

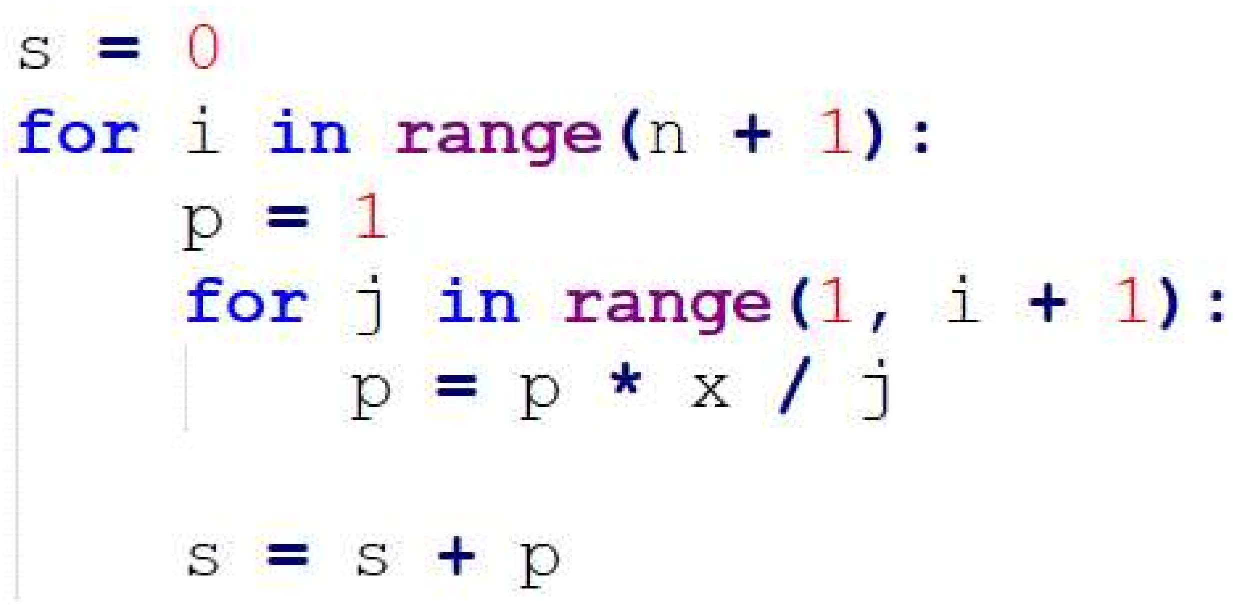
Exercises

- Find the k^{th} smallest integer in an unordered array of integers:
  - Write your code
  - Identify the Big(O) of your algorithm

Exercises

- Implementations of the algorithms for adding, multiplying, and transposing n x n matrices.
  - Write 3 functions to implement requirements
  - Identify the Big(O) of each function

Big-O Exercises



```
s = 0
for i in range(n + 1):
    p = 1
    for j in range(1, i + 1):
        p = p * x / j

    s = s + p
```

Big-O Exercises

```
for i in range(1, n + 1):
    for j in range(1, m + 1):
        x += 2
```

Exercise

```
def fibonacci(n):
    if n <= 1:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)
```

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.