SINGAPORE INSTITUTE OF TECHNOLOGY

Big-O & Iterative Algorithms

Algorithm Analysis

Example: 1-Sum

```
count(a, N):
    sum = 0
    for i in range(N):
        if a[i] == 0:
            sum += 1
    return sum
```

How many instructions as a function of input size N?

Operation	Frequency
Assignment statement	1
For loop, "in range" comparison	N+1
"if equal" comparison	N
Array access []	N
Increment	N
Total	**(3N+2) to (4N+2)**

Example: 2-Sum

SINGAPORE INSTITUTE OF TECHNOLOGY

```python
def count(a, N):
    sum = 0
    for i in range(N):
        for j in range (i+1,N):
            if a[i] +a[j] == 0:
                sum += 1
    return sum
```

[A]
N+1 comparisons

[B]
i=0, j=1 … N => N comparisons
i=1, j=2 … N => N-1 comparisons
…
i=N-2, j=N-1 .. N => 2 comparisons
i=N-1, j=N … N => 1 comparisons

[A] + [B]

Operation	Frequency
Assignment statement	1
For loop "in range" comparison	$(N+1) + [N+(N-1)+..+1 + 0] = \frac{1}{2}N(N+3)+1$
Equal comparison	$(N-1)+(N-2)+..+1 + 0 = \frac{1}{2}N(N-1)$
Array access []	$N(N\text{-}1)$
Increment	$0 \text{ to } \frac{1}{2}N(N-1)$
Total	*2 + 2N^2 to 2 + 5/2N^2 - 1/2N*

Asymptotic notations: Comparing algorithms

- Consider two algorithms, A and B, for solving a given problem.
- Let the running times of the algorithms be $T_a(n)$ and $T_b(n)$ for problem size n.
- Suppose the problem size is n_0 and

$$T_a(n_0) < T_b(n_0)$$

Then algorithm A is better than algorithm B for **problem size** n_O .

Comparing algorithms

If

$$T_a(n) < T_b(n)$$

for all $n \geq n_0$

Then algorithm A is **better** than algorithm B **regardless** of the problem **size**

Comparing algorithms

SINGAPORE INSTITUTE OF TECHNOLOGY

For algorithm analysis,

we emphasize on the operation count's

`order of growth` `for` `large input` `sizes`

To compare and rank the order of growth (for comparing the efficiency of different algorithms), we use `Asymptotic notations.`

Note: the difference in running times on small inputs cannot really distinguish efficient algorithms from inefficient ones. `Interested in large values of input,` n.

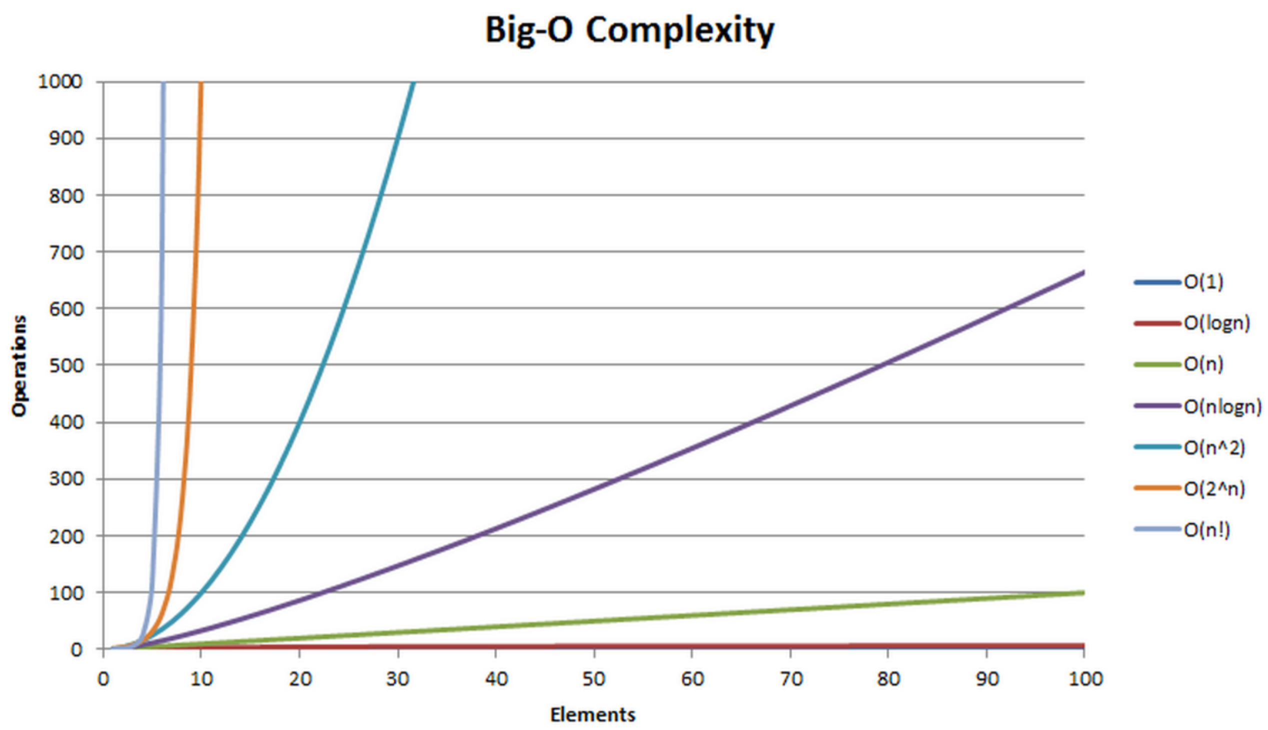
Asymptotic notations

In comparing algorithms, consider the `asymptotic` behaviour of the two algorithms for large problem sizes, under `worst-case`.

`Big-Oh notation`: used to characterize the asymptotic behavior of functions.

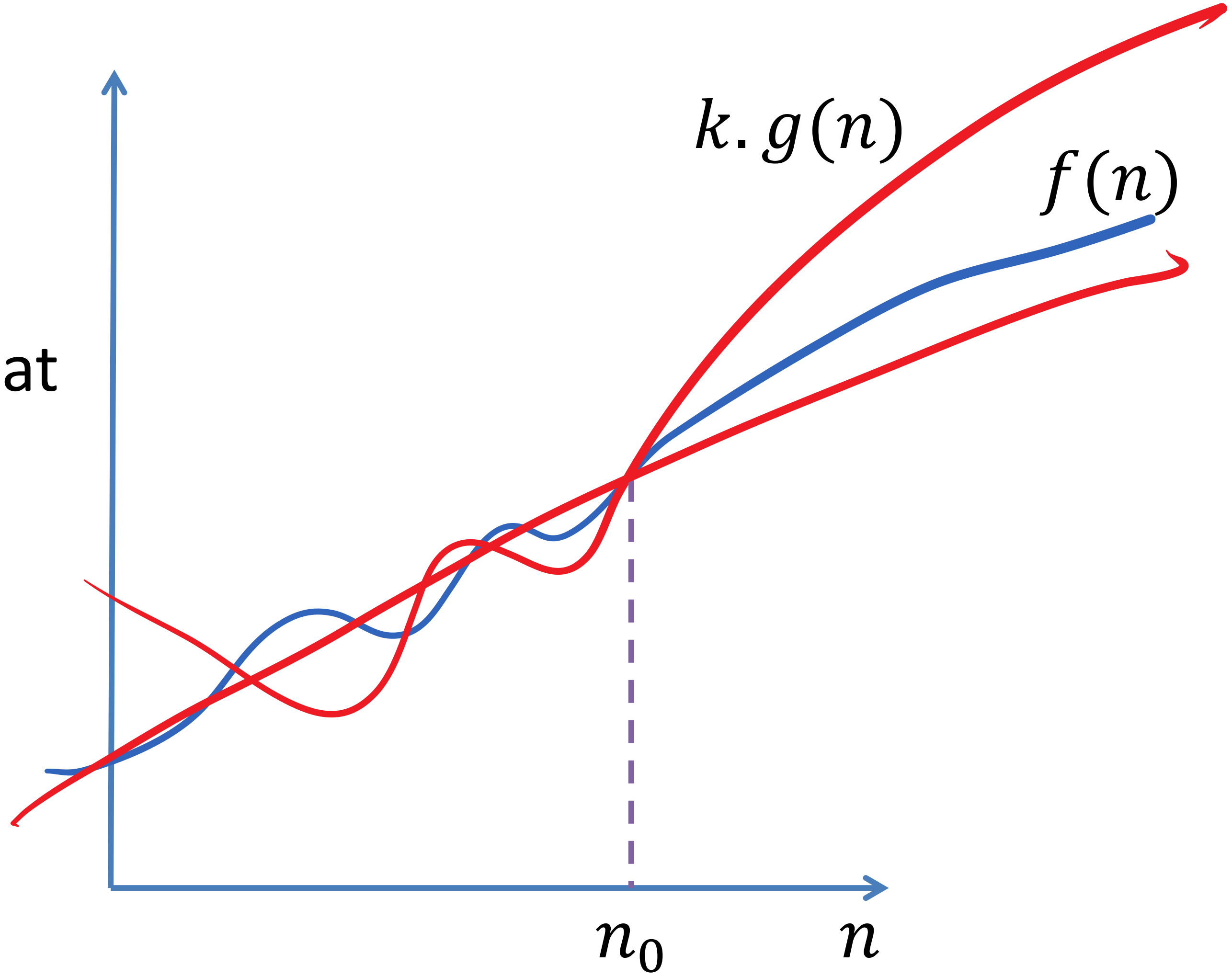
Revision on Big-O

- Informal way of thinking about Big-O Notation:
- O(1): Nightcrawler teleport
- O(lg(n)): train
- O(n): taxi
- $O(n^2)$: bus
- $O(n^3)$: cycling
- $O(n^4)$: walking



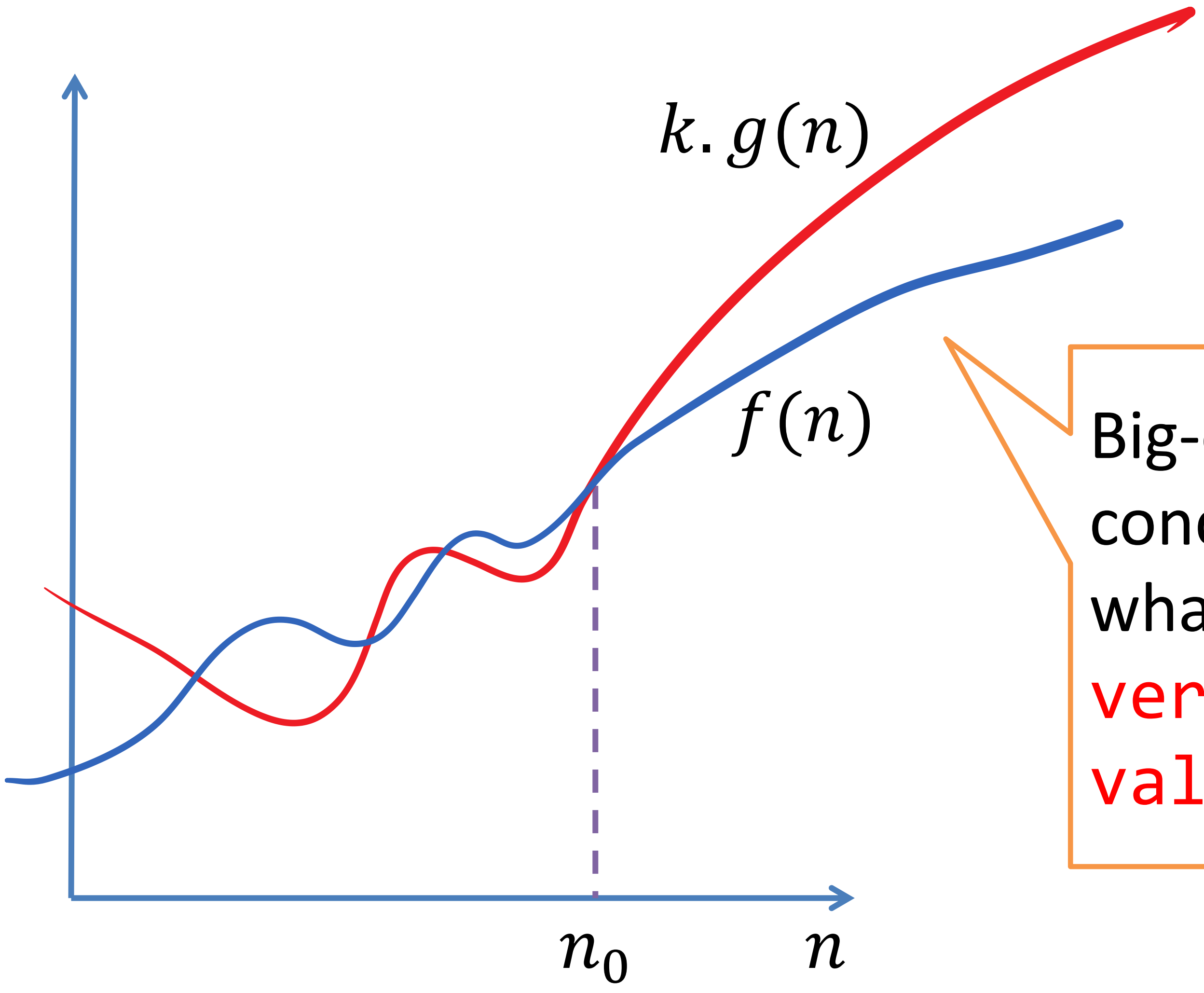
Revision on Big-O

- Informal way of thinking about Big-O Notation:
- Big-O is taking the taxi during peak hours.
- Big- Ω is taking a taxi that drives 120km/h



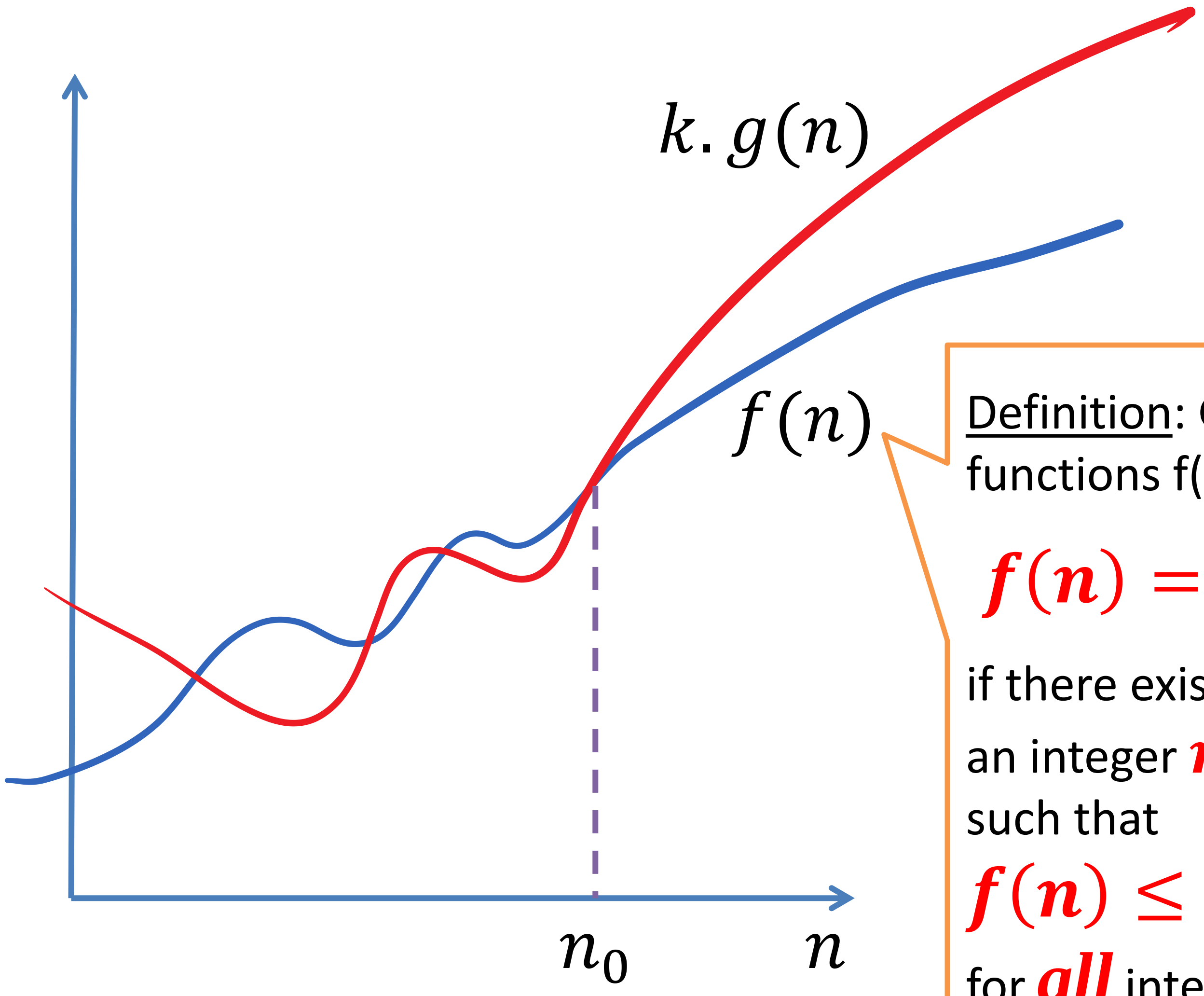
Big-Oh Notation

(the "O" stands for "order of")



Big-oh notation is concerned with what happens for **very large values of n**.

Big-Oh Notation



Definition: Given non-negative functions f(n) and g(n), we say that

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

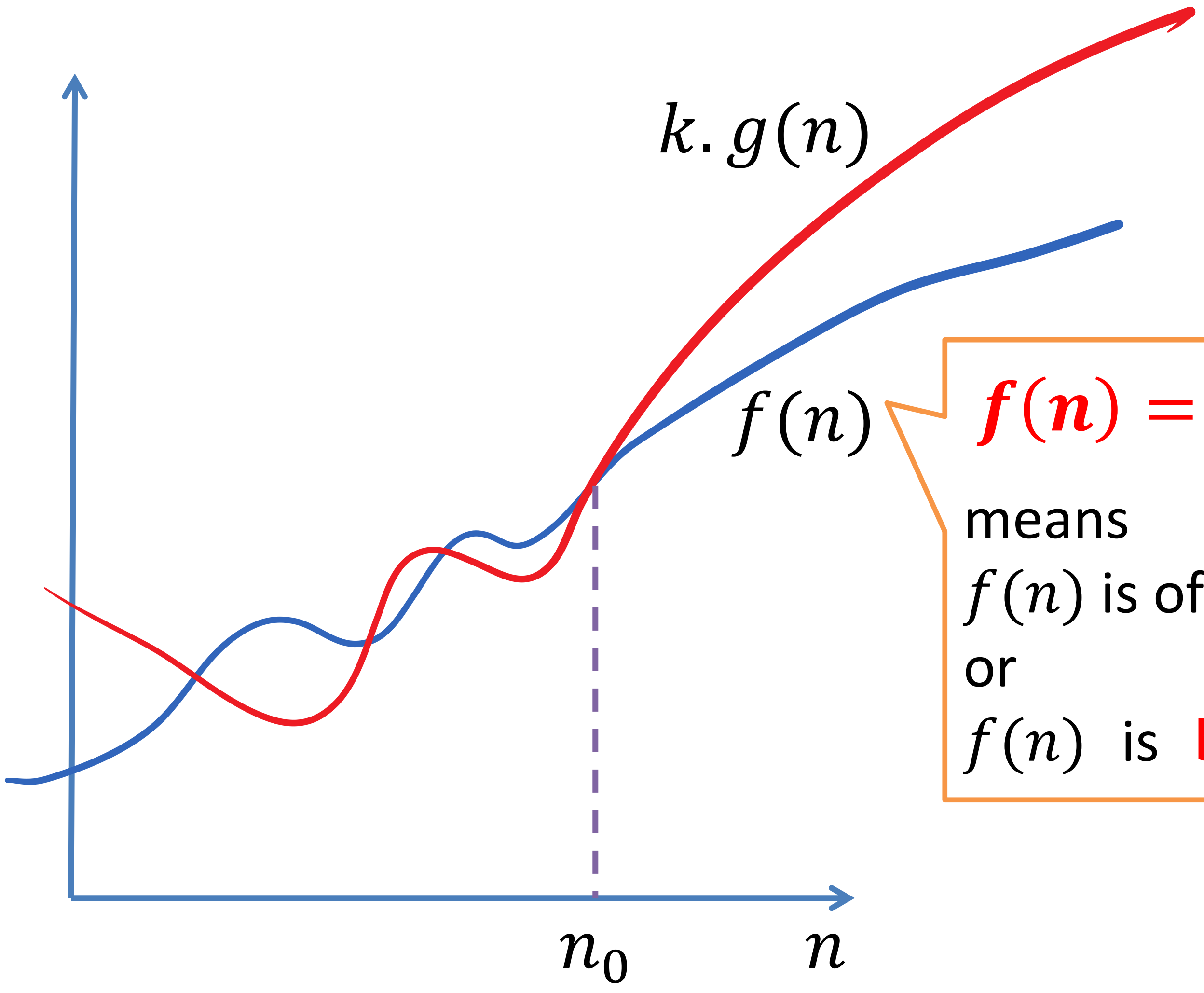
if there exists

an integer n_0 & a constant $k > 0$ such that

$$f(n) \leq k.g(n)$$

for **all** integers $n \geq n_0$.

Big-Oh Notation



$\boldsymbol{f(n) = O(\,g(n)\,)}$

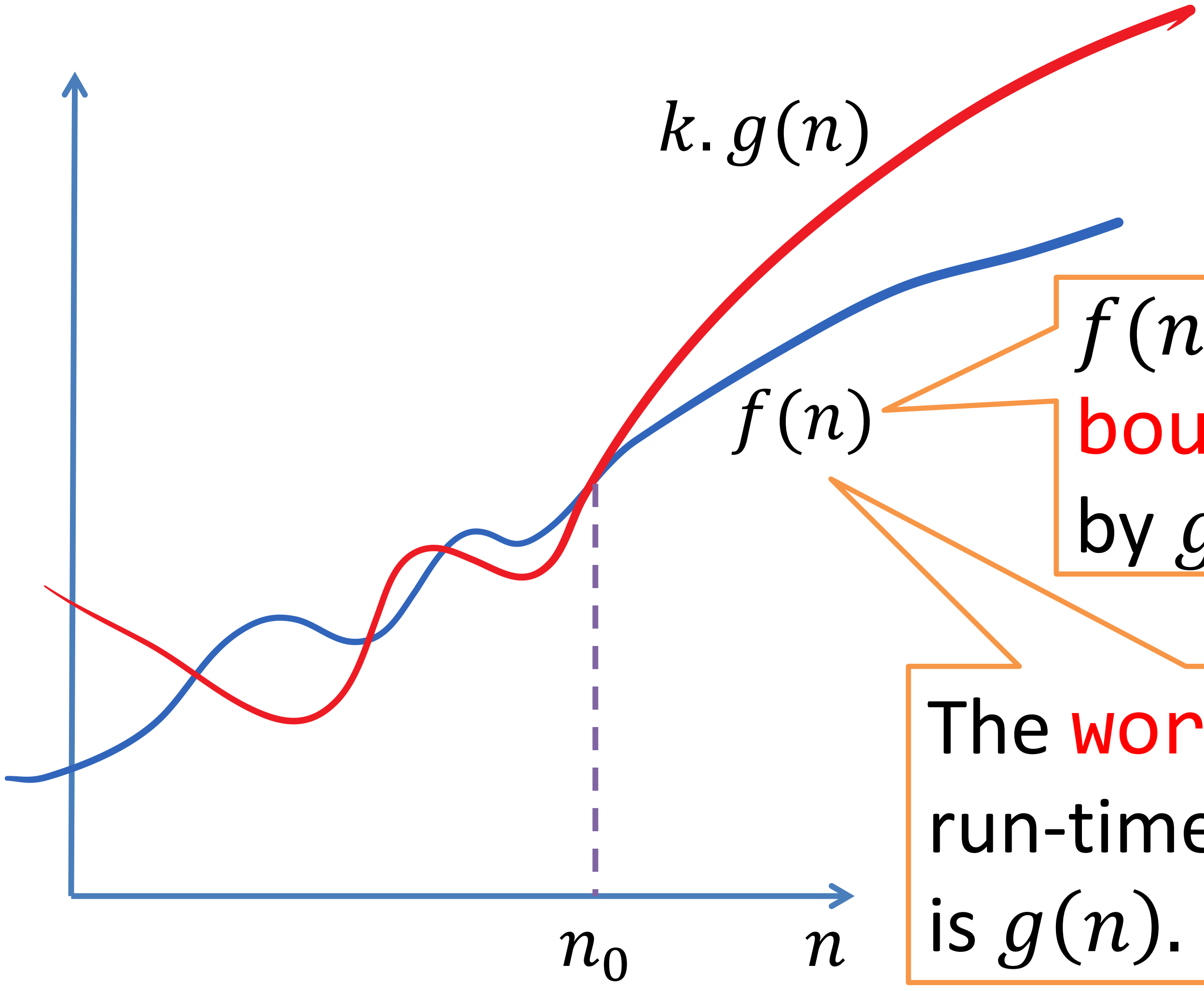
means

$f(n)$ is of order at most $g(n)$;

or

$f(n)$ is big-oh of $g(n)$.

Big-Oh Notation



$f(n)$ is **bounded above** by $g(n)$.

The **worst-case** run-time of $f(n)$ is $g(n)$.

Big-Oh Example: 1-Sum

```python
def count(a, N):
    sum = 0
    for i in range(N):
        if a[i] == 0:
            sum += 1
    return sum
```

Maximum
total operations:
$4n + 2$.

Prove that
$(4n + 2)$ is $O(n)$.

Proof

Need to prove this condition:

$4\text{n} + 2 \leq \text{kn}$ for all $n \geq n_0$.

Can we find k (> 0) and n_0?

$\Rightarrow 4\text{n} + 2 \leq \text{kn}$

$\Rightarrow (k - 4)n \geq 2$

$\Rightarrow n \geq \frac{2}{k-4}$

$\Rightarrow$ Pick $\text{k} = 5$ and $n_0 = 2$, gives:

$4n + 2 \leq 5n,$

$for\ all\ n \geq 2.$

$\therefore Proven.$

We say that the worst case run-time of 1-Sum is $O(n)$.

Big-Oh Example: 2-SUM

```python
def count(a, N):
    sum = 0
    for i in range(N):
        for j in range (i+1,N):
            if a[i] +a[j] == 0:
                sum += 1
    return sum
```

Maximum total operations:

$$\frac{1}{2}n(5n+6)+\frac{3}{2}$$

Prove that $\frac{1}{2}n(5n+6)+\frac{3}{2}$ is $O(n^2)$.

Proof

Need to prove this condition:

$\frac{1}{2}n(5n+6)+\frac{3}{2} \le \mathrm{k}n^2$ for all $n \ge n_0$.

i.e. $5n^2+6n+3 \le 2kn^2$.

Can we find k (> 0) and n_0?

We have:

For all $n \ge 1$,

$\Rightarrow\ 5n^2+6n+3 \le 5n^2+6n^2+3n^2$

$\Rightarrow\ 5n^2+6n+3 \le 14n^2$

Compare $(5n^2+6n+3)/2 \le kn^2$

$5n^2+6n+3 \le 2kn^2$

$\Rightarrow$ 2k=14

$\Rightarrow$ pick $\mathrm{k}=7$ & $n_0=1$, gives:

$\frac{1}{2}n(5n+6)+\frac{3}{2} \le 7\mathrm{n}^2$ for all $n \ge 1$.

$\therefore$ *Proven*.

Big-Oh Rules

If $f(n)$ is a polynomial of degree d, then $f(n)$ is $O\left(n^d\right)$.

- Drop lower-order terms
- Drop constant factors

Example:

$$f(n) = 8n^6 + 7n^4 + 5n^2 + 2n + 16$$

$$f(n) = O(n^6)$$

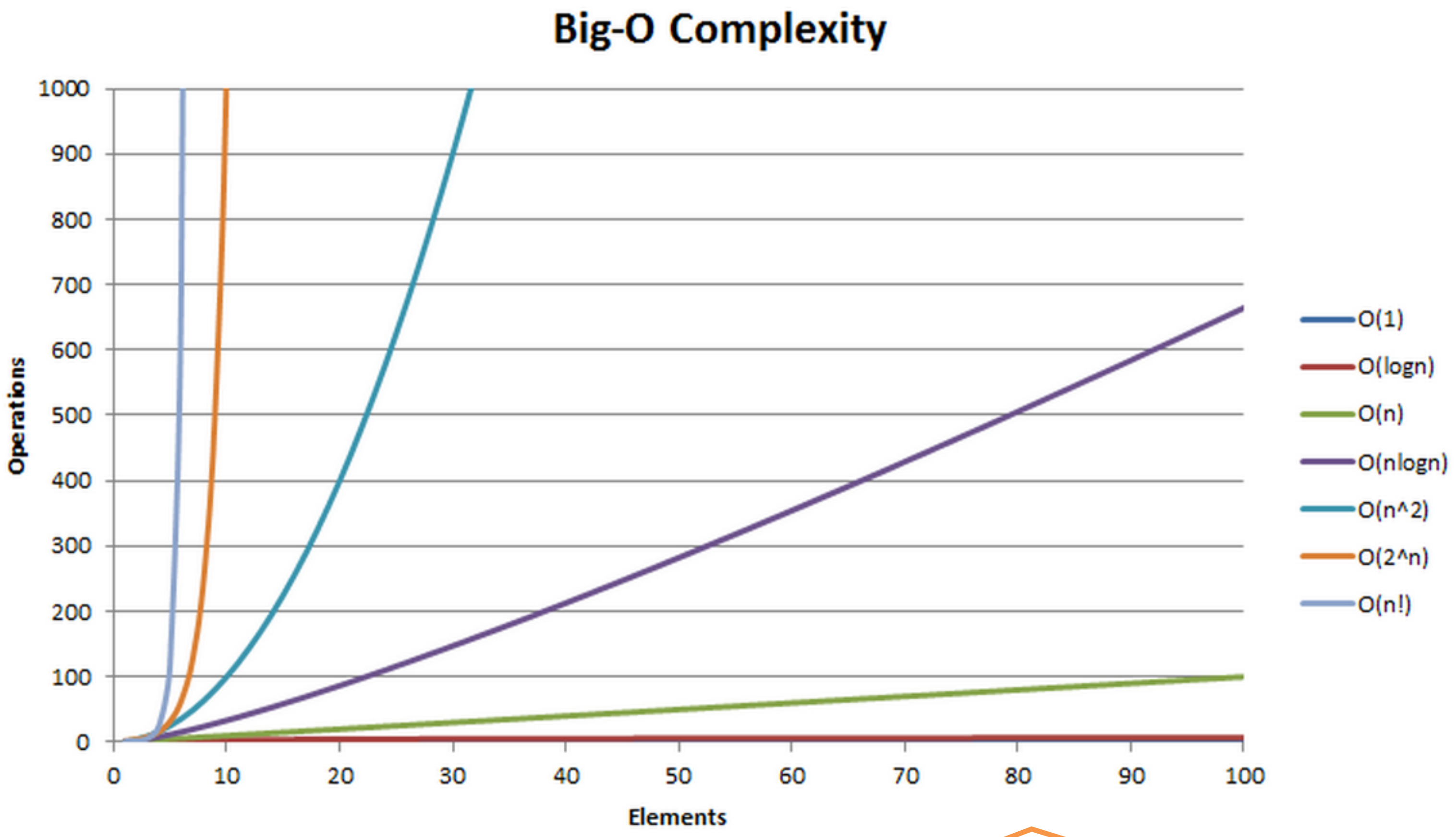
Big-Oh and Growth Rate

The big-Oh notation gives an upper bound on the growth rate of a function.

The statement
"$\boldsymbol{f(n)}$ *is* $\boldsymbol{O(g(n))}$"
means that the growth rate of $\boldsymbol{f(n)}$ is no more than the growth rate of $\boldsymbol{g(n)}$

Common order-of-growth

Graphical illustration



The set of functions $1,\ logN, \sqrt{N},\ N,\ NlogN,\ N^2,\ N^3\ and\ 2^N$ suffices to describe the order of growth of most common algorithms.

Common order-of-growth

Numeric illustration

lg N	$\lg^2 N$	$\sqrt{N}$	N	$N \lg N$	$N \lg^2 N$	$N^{3/2}$	N^2
3	9	3	10	30	90	30	100
6	36	10	100	600	3.600	1.000	10.000
9	81	31	1.000	9.000	81.000	31.000	1.000.000
13	169	100	10.000	130.000	1.690.000	1.000.000	100.000.000
16	256	316	100.000	1.600.000	25.600.000	31.600.000	10 Million
19	361	1.000	1.000.000	19.000.000	361.000.000	1 Million	1 Billion

In this table:
lg N means $\log_2 N$.
$\lg^2 N$ means $(\lg N)^2$ or $(\log_2 N)^2$.

Common order-of-growth

Algorithm or program code illustration

order of growth	name	typical code framework	description	example
1	constant	`a = b + c;`	statement	add two numbers
log N	logarithmic	`while (N > 1)` `{   N = N / 2;   ...   }`	divide in half	binary search
N	linear	`for (int i = 0; i < N; i++)` `{   ...         }`	loop	find the maximum
N log N	linearithmic	[see mergesort lecture]	divide and conquer	mergesort

N^2	quadratic	`for (int i = 0; i < N; i++)`
		` for (int j = 0; j < N; j++)`
		`  {   ...         }`
			double loop	check all pairs

N^3	cubic	`for (int i = 0; i < N; i++)`
		` for (int j = 0; j < N; j++)`
		`  for (int k = 0; k < N; k++)`
		`   {   ...         }`
			triple loop	check all triples

2^N	exponential	[see combinatorial search lecture]	exhaustive search	check all subsets

Growth Rate: Practical Implication

growth rate	name	description	effect on a program that runs for a few seconds	
			time for 100x more data	size for 100x faster computer
1	constant	independent of input size	-	-
log N	logarithmic	nearly independent of input size	-	-
N	linear	optimal for N inputs	a few minutes	100x
N log N	linearithmic	nearly optimal for N inputs	a few minutes	100x
N^2	quadratic	not practical for large problems	several hours	10x
N^3	cubic	not practical for medium problems	several weeks	4-5x
2^N	exponential	useful only for tiny problems	forever	1x

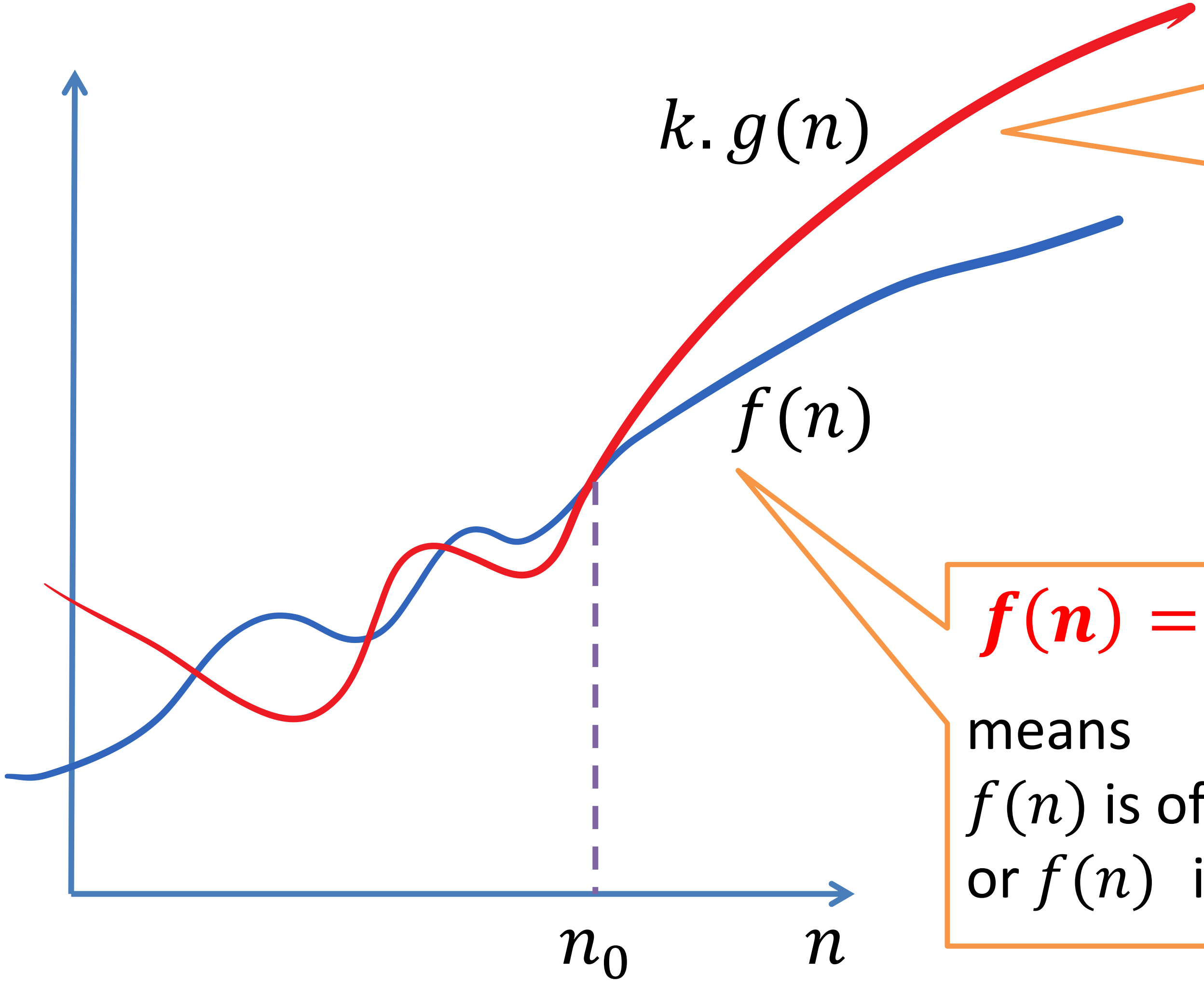
Types of Analyses

- Worst case. Upper bound on cost.
  - Determined by "most difficult" input.
  - Provides a guarantee for all inputs.
- Best case. Lower bound on cost.
  - Determined by "easiest" input.
  - Provides a goal for all inputs.
- Average case. Expected cost for random input.
  - Needs a model for "random" input.
  - Provides a way to predict performance.

Theory of Algorithms

- Upper bound.
  - Performance guarantee of algorithm for any input.
- Lower bound.
  - Proof that no algorithm can do better.
- Optimal algorithm.
  - Lower bound = upper bound (to within a constant factor).

Big-Oh Notation - upper bound



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

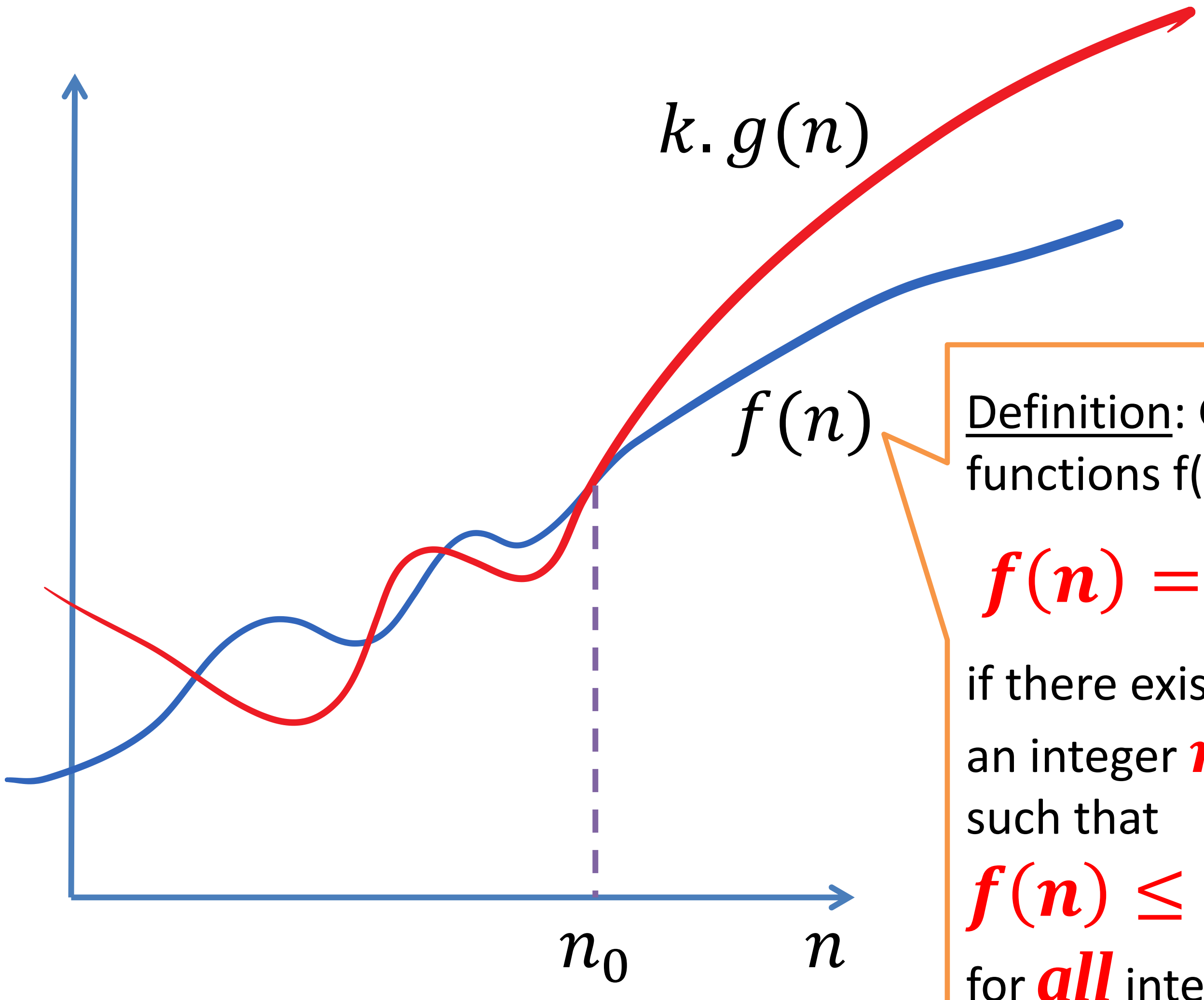
$\boldsymbol{f(n) = O(}\ g(n)\ \boldsymbol{)}$

means

$f(n)$ is of order at most $g(n)$;

or $f(n)$ is big-oh of $g(n)$.

Big-Oh Notation - upper bound



Definition: Given non-negative functions f(n) and g(n), we say that

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

if there exists

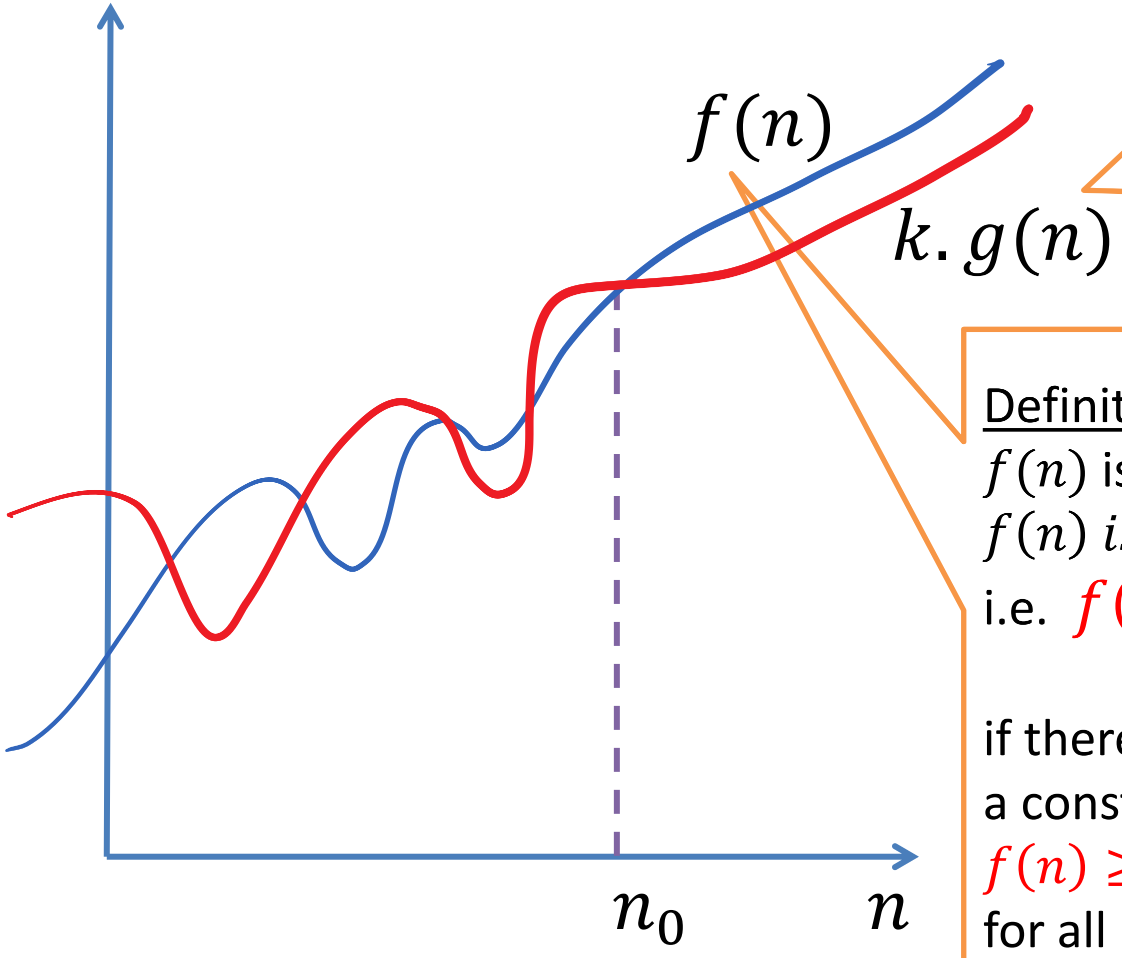
an integer n_0 & a constant $k > 0$

such that

$$f(n) \le k.g(n)$$

for ***all*** integers $n \ge n_0$.

Big-Omega Notation - lower bound



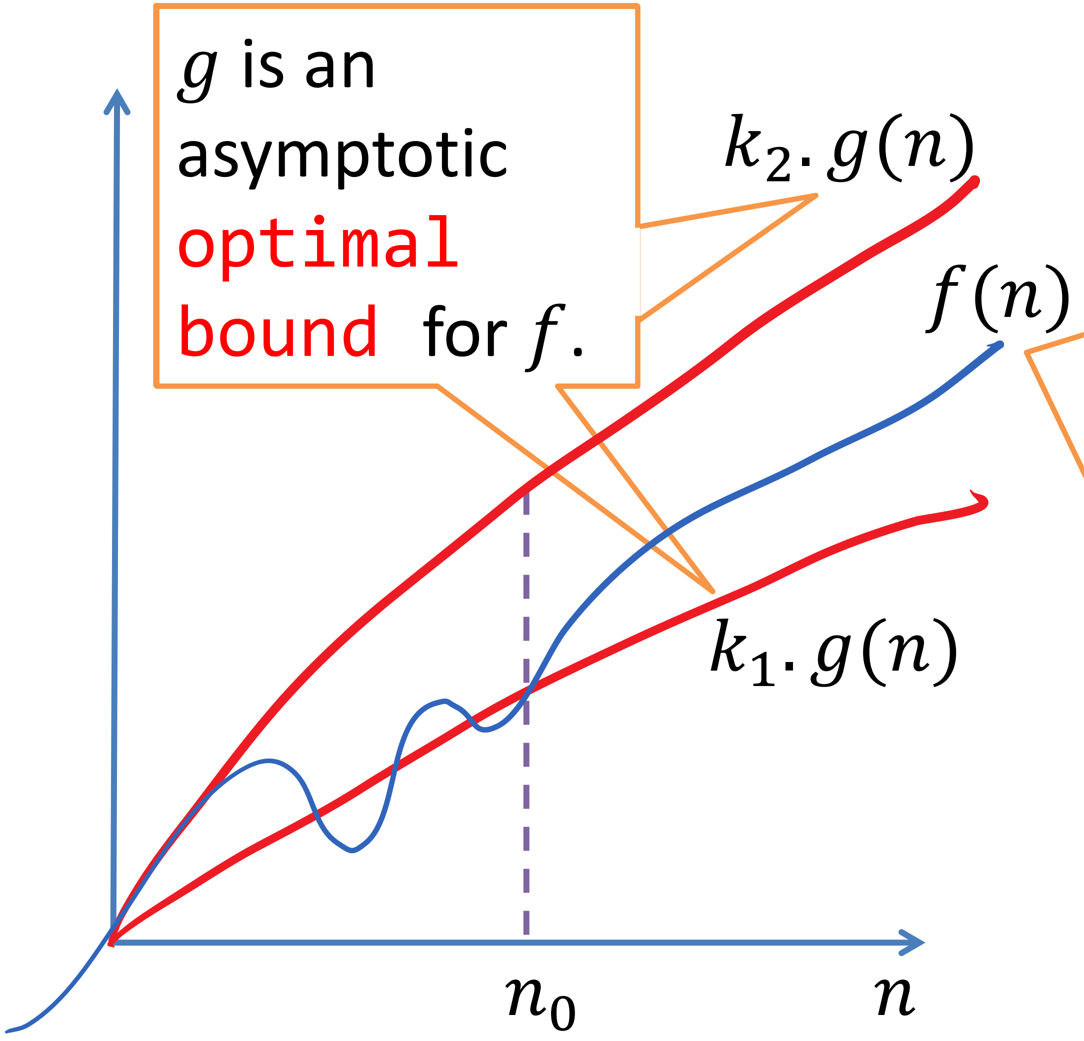
g is an asymptotic **lower bound** for f.

Definition:
$f(n)$ is of order at least $g(n)$ or
$f(n)$ *is Big–Omega of* $g(n)$
i.e. $f(n) = \boldsymbol{\Omega}(g(n))$

if there exists an integer n_0 and
a constant $k > 0$ such that
$f(n) \geq k.g(n)$
for all integers $n \geq n_0$.

Big-Theta Notation - optimal bound

SINGAPORE INSTITUTE OF TECHNOLOGY



<u>Definition</u>:

$f(n)$ is of <u>order</u> $g(n)$ or

$f(n)$ is big-theta of $g(n)$

i.e. $f(n) = \Theta(g(n))$

if

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

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

Properties of Asymptotic

Suppose we know that

$$f_1(n) = O(g_1(n))$$

$$f_2(n) = O(g_2(n))$$

What can we say about the asymptotic behavior of the sum and the product of $f_1(n)$ and $f_2(n)$?

Properties of Asymptotic

Suppose we know that

$$f_1(n) = O(g_1(n))$$
$$f_2(n) = O(g_2(n))$$

Theorem 1:
$$f_1(n) + f_2(n) = O(\max(g_1(n), g_2(n)))$$

Consider the functions
$f_1(n) = n^3 + n^2 + n + 1 = O(n^3)$ and
$f_2(n) = n^2 + n + 1 = O(n^2)$

By Theorem 1 , the asymptotic behavior of the sum
$f_1(n) + f_2(n)$ is $O\left(max(n^3, n^2)\right)$.
$\Rightarrow$ $f_1(n) + f_2(n)$ is $O(n^3)$.

Properties of Asymptotic

Suppose we know that

$$f_1(n) = O(g_1(n))$$
$$f_2(n) = O(g_2(n))$$

Theorem 2:
$f_1(n) \times f_2(n) = O(g_1(n) \times g_2(n))$

Consider the functions
$f_1(n) = n^3 + n^2 + n + 1 = O(n^3)$ and
$f_2(n) = n^2 + n + 1 = O(n^2)$

By Theorem 2 , the asymptotic behavior of the product
$f_1(n) \times f_2(n)$ is $O(n^3 \times n^2)$.
$\Rightarrow f_1(n) \times f_2(n)$ is $O(n^5)$.

General Plan for Algo Run-time Analysis

1. Decide on parameter n indicating $input\ size$.
2. Identify algorithm's $basic\ operation - cost\ model$.
3. Set up a sum expressing the $number\ of\ times$ the basic operation is executed.
4. Simplify the sum using standard formulas and rules to determine $big\text{–}Oh$ of the running time.

Example 1: O(n)

- Provide a Big-oh notation (means an upper bound or a worst case analysis) for the run-time of the following algorithm

```python
def funcA(n):
    sum = 0
    x = n*[100*random.random()]
    for i in range(n):
        sum += x[i]
    return sum
```

Example 1: O(n)

SINGAPORE INSTITUTE OF TECHNOLOGY

```
def funcA(n):
    sum = 0
    x = n*[100*random.random()]
    for i in range(n):
        sum += x[i]
    return sum
```

1. Input size: n
2. Basic operations: Statements in the for loop
3. Number of times the basic operations are executed: n
4. According to Big-Oh rules, the runtime of the algorithm is $O(n)$, i.e. $Linear$ run-time

Example 2: O(lg(n))

```python
def funcB(n):
    sum = 0
    x = n*[100*random.random()]
    count = 1
    while count<n:
        sum += x[count]
        count=count*2
    return sum
```

1. Input size: n
2. Basic operations: Statements in the *while* loop
3. Number of times the basic operations are executed: 2*lg(n)
4. According to Big-Oh rules, the runtime of the algorithm is $O(\lg(n))$, i.e. *Logarithmic* run-time

Example 3: $O(n^2)$

```
def funcC(n):
    sum = 0
    x = [ n*[100*random.random()] for i in range(n)]
    for i in range(n):
        for j in range(n):
            sum += x[i][j]
    return sum
```

1. Input size: n
2. Basic operations: Statements in the double nested *for* loop
3. Number of times the basic operations are executed: $n*n = n^2$
4. According to Big-Oh rules, the runtime of the algorithm is $O(n^2)$, i.e. *Quadratic* run-time

Example 4: $O(n^3)$

SINGAPORE INSTITUTE OF TECHNOLOGY

```python
def funcD(n):
    sum = 0
    x = [[ n*[100*random.random()] for i in range(n)] for i in range(n)]
    for i in range(n):
        for j in range(n):
            for k in range(n):
                sum += x[i][j][k]
    return sum
```

1. Input size: n
2. Basic operations: Statements in the triple nested for loop
3. Number of times the basic operations are executed: $n*n*n = n^3$
4. According to Big-Oh rules, the runtime of the algorithm is $O(n^3)$, i.e. *Cubic* run-time

Example 5: $O(n^2)$

SINGAPORE INSTITUTE OF TECHNOLOGY

```
def funcE(n):
    sum = 0
    x = [ n*[100*random.random()] for i in range(n)]
    for i in range(n):
        for j in range(i+1):
            sum += x[i][j]
    return sum
```

1. Input size: n
2. Basic operations: Statements in the doubly nested for loop
3. Number of times the basic operations are executed:

   = 1+2+3+…+(n-2)+(n-1)+n

   = ½ n(n+1)

   = ½ (n^2+n)

1. According to Big-Oh rules, the runtime of the algorithm is $O(n^2)$, i.e. *Quadratic* run-time

Example 6: $O(2^n)$

```python
def fib(n):
    if n==1 or n ==2:
        return 1
    else:
        return fib(n-1)+fib(n-2)
```

1. Input size: n.
2. Basic operations: Recursive call with two sub-branches.
3. The runtime of the algorithm is $O(2^n)$, i.e. *Exponential* run-time.