ICT1008 Data Structures and Algorithms

Lecture 2: Analysis of Algorithms

Agenda

- Mathematics for Algorithms
- Why analyse algorithms?
- Empirical analysis
- Mathematical models
- Asymptotic rules
- Theory of algorithms
- General plan for Algorithm analysis

Recommended Reading

1. Algorithms by Robert Sedgewick and Kevin Wayne. Addison-Wesley Professional. 4th edition, 2011
   – Chapter 1.4

2. Runestone Interactive book: "Problem Solving with Algorithms and Data Structures Using Python"
   – Section: "Analysis"

Mathematics for Algorithms

- Polynomials
- Combinations
- Logarithms
- Summation of Series

Polynomials

A polynomial of degree n is a function of the form

$$p(x) = c_n x^n + c_{n-1} x^{n-1} + \ldots + c_1 x + c_0$$

with $c_n \neq 0$. The numbers c_i are called coefficients.

Example, a polynomial of degree 5.

$$p(x) = 3x^5 - 12x^3 + 9x + 4$$

Combinations

For $n \geq k \geq 0$, the number of k-element subsets of an n element set is given by

$$\binom{n}{k} = \frac{n!}{(n-k)!\,k!}$$

Example

$$\binom{5}{2} = \frac{5!}{3!2!} = \frac{5 \times 4 \times 3 \times 2 \times 1}{(3 \times 2 \times 1) \times (2 \times 1)} = 10$$

The subsets are: { (1,2), (1,3), (1,4), (1,5), (2,3), (2,4), (2,5), (3,4), (3,5), (4,5) }

Logarithm - Definition

if $b^x = n$ then , $\log_b n = x$

Note: $\mathbf{log_2}$ is usually written as $\mathbf{lg}$

e.g. $10^3 = 1000$, then $\log_{10} 1000 = 3$

e.g. $2^6 = 64$, then $\log_2 64 = 6$

Law of Logarithm

Suppose that $b > 0$, and $b \neq 1$, then

$$\log_b(xy) = \log_b x + \log_b y$$

$$\log_b\left(\frac{x}{y}\right) = \log_b x - \log_b y$$

e.g. $\log_{10}(10*100) = \log_{10} 10 + \log_{10} 100$
$= 1 + 2 = 3$

e.g. $\log_2(64/8) = \log_2 64 - \log_2 8$
$= 6 - 3 = 3$

Law of Logarithm

Suppose that $b > 0$, and $b \neq 1$, then

$$if\ a > 0\ and\ a \neq 1, \log_a x = \frac{\log_b x}{\log_b a}$$

e.g. $\log_{10}32 = 1.505, \quad \log_{10}2 = 0.301$

$$\log_2 32 = \frac{\log_{10} 32}{\log_{10} 2} = \frac{1.505}{0.301} = 5$$

Summation of Series

- Arithmetic Series:

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

e.g. $1 + 2 + \cdots + 100$

$$= \frac{100(100+1)}{2}$$

$$= 5050$$

Summation of Series

- Geometric Series:

$$\sum_{k=0}^{n} ar^k = a + ar + \ldots + ar^n = \frac{a(r^{n+1}-1)}{r-1}$$

e.g. $1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \ldots$

$= 1 + \frac{1}{2} + (\frac{1}{2})^2 + (\frac{1}{2})^3 + (\frac{1}{2})^4 + \ldots$

$= \dfrac{1((\frac{1}{2})^{\infty} - 1)}{\frac{1}{2} - 1} = \dfrac{(0-1)}{-\frac{1}{2}} = 2$

α Mini Quiz

- Go to xSite -> assessments -> quizzes – “INF1008 Lecture 2 Quiz”
- 5 mins
- NOT graded
- Password is inf1008lec2

Analysis of Algorithms

To analyze an algorithm is to determine the amount of resources (such as time and storage) necessary to execute it.

$$(Time/Space)\ Complexity\ =\ f(n)$$

The running time of an algorithm typically grows with the input size n.

Analysis of Algorithms

- There are two ways to analyze an algorithm
    - Empirical/Experimental studies
    - Theoretical analyses

Empirical Study method

1. Write a program to implement the algorithm
2. Run the program with inputs of varying sizes and compositions
3. Get an `accurate measure` of the actual running time
4. Plot the results

Problems with Empirical Data Analysis

- System dependent effects.
  - `Hardware`: CPU, memory, cache, …
  - `Software`: compiler, interpreter, garbage collector, …
  - `System`: operating system, network, other applications, …

Theoretical Analysis

- To analyse the running time of algorithms, use a simple model of the underlying computer
- Aim: make simplifications to estimate resources to execute an algorithm
- Exact time, machine instructions for different machines are not relevant here
- We use a model of a Random Access Machine (RAM)
- Primary challenge: determine the frequency of execution statements

RAM Model

- Consider sequential 1-Processor architecture, no parallelism.
- All data are directly accessible in memory.
- All memory accesses take the same length.
- All elementary operations require constant time.
- Elementary operations are:
  - Value assignment;
  - Arithmetic operations such as addition, subtraction, multiplication;
  - Logical operations such as “and”, “or” ;
  - Comparison operations such as “<”, “>” ;
  - Commands to control the flow of instructions, such as “if then else”.
- For simplicity, assume each elementary operation takes one time unit.

Example: 1-Sum

```
def 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

```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
```

How many instructions as a function of input size N?

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 to $\frac{1}{2}N(N-1)$
Total	$N(2N+3)+2$ to $\frac{1}{2}N(5N+6)+\frac{3}{2}$

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
```

[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 to $\frac{1}{2}N(N-1)$
Total	$N(2N+3)+2$ to $\frac{1}{2}N(5N+6)+\frac{3}{2}$

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

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.

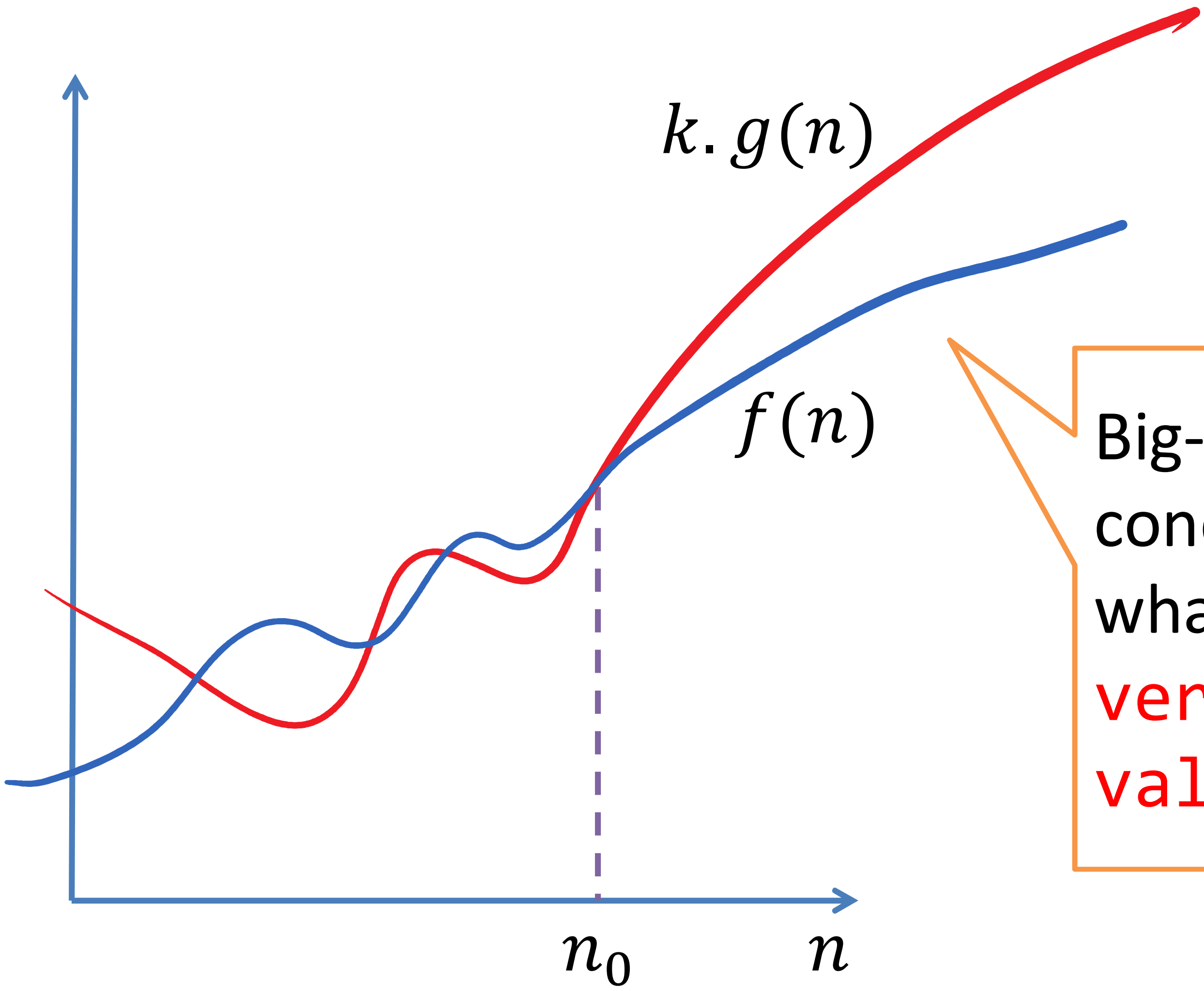
Asymptotic notations

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.

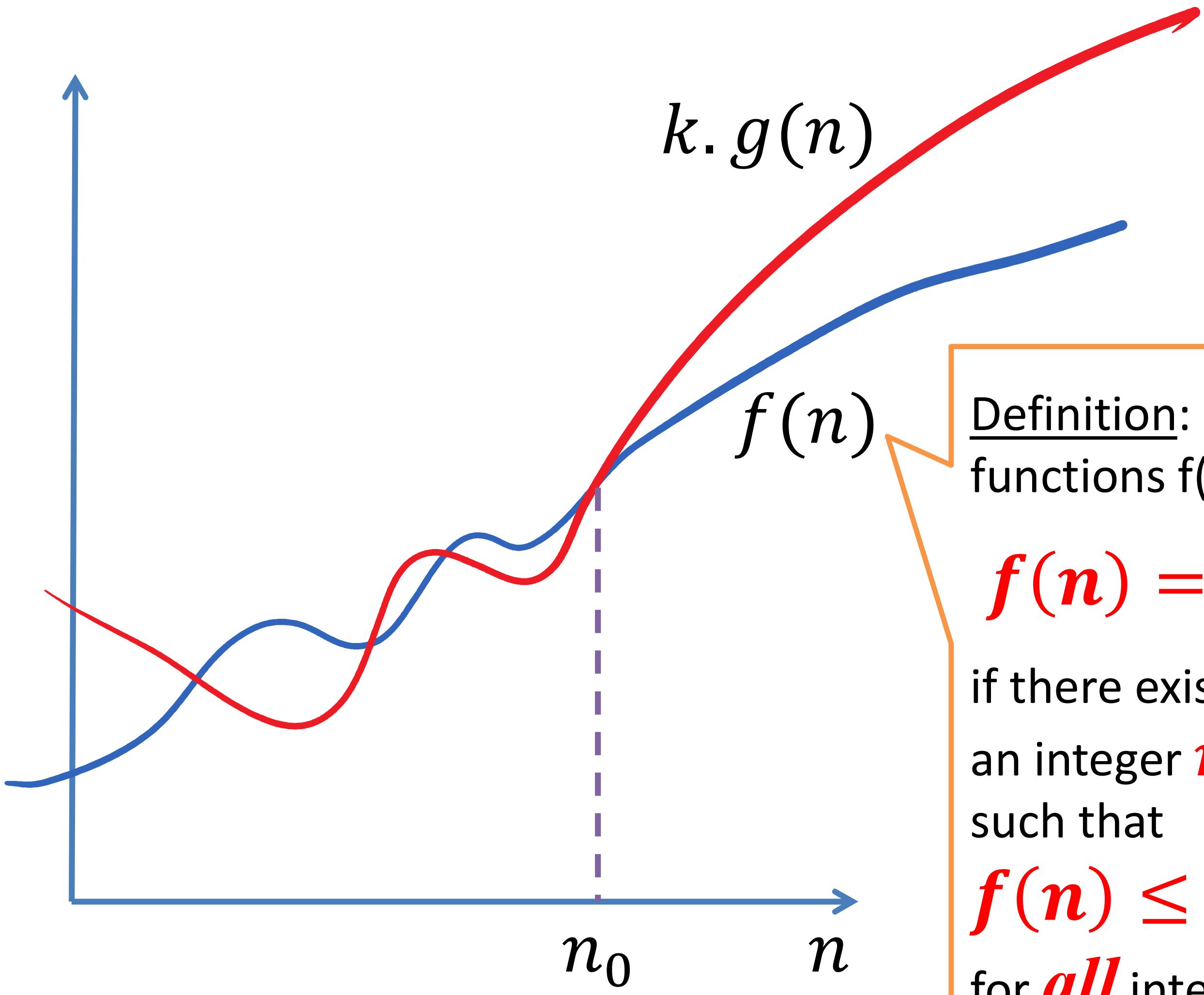
Big-Oh Notation

(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

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

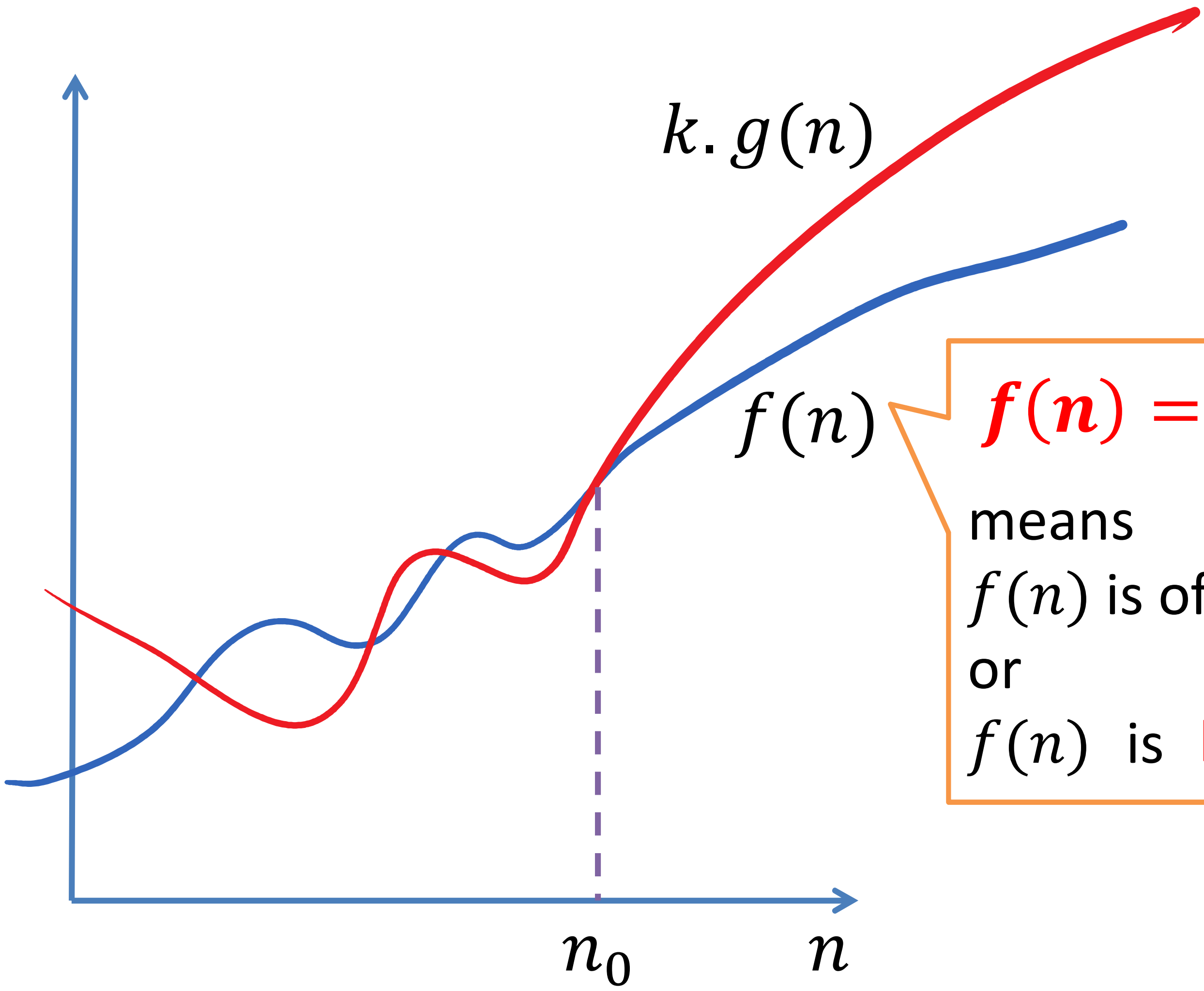
if there exists

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



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

means

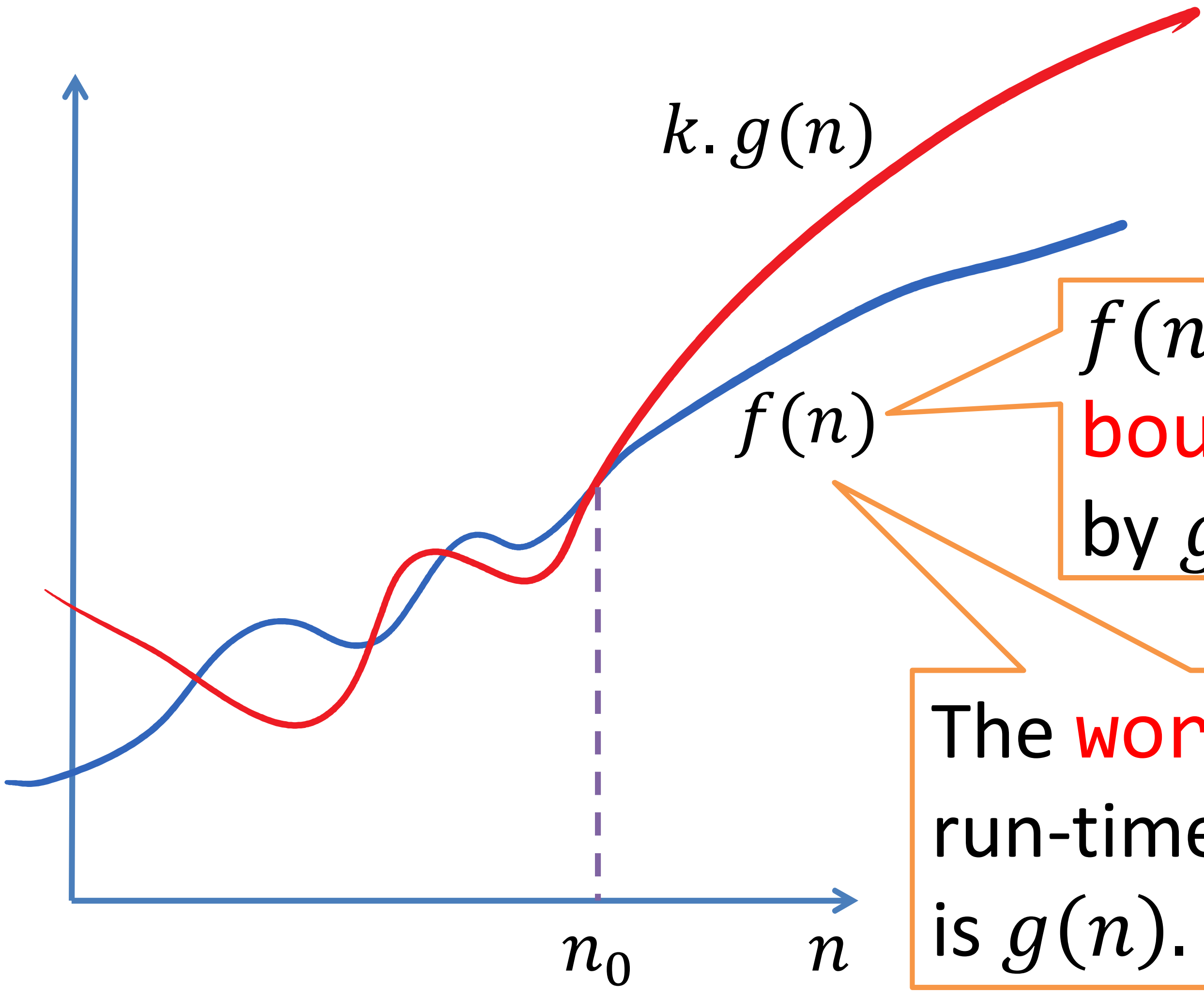
$f(n)$ is of order at most $g(n)$;

or

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

SIT Internal

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:

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

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

Can we find k (>0) and n_0?

We have:

$For\ all\ n \geq 1,$

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

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

Compare $5n^2+6n+3 \leq 2kn^2$

$\Rightarrow$ 2k=14

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

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

$\therefore\ Proven$.

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.

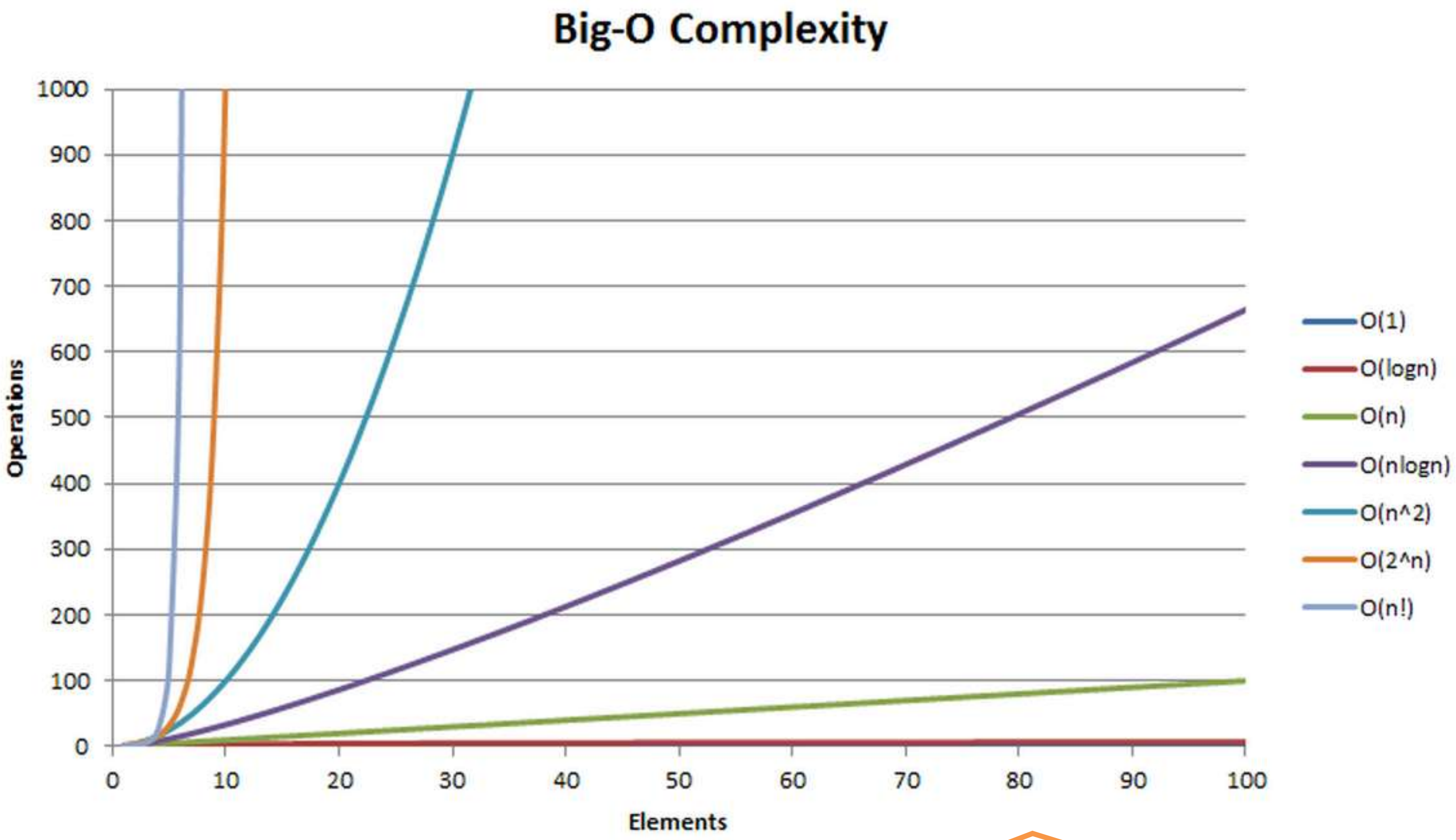
The statement

"$\boldsymbol{f}(\boldsymbol{n})$ *is* $\boldsymbol{O}(\boldsymbol{g}(\boldsymbol{n}))$"

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

Common order-of-growth

Graphical illustration



The set of functions $1,\ logN,\ N,\ NlogN,\ N^2,\ N^3\ and\ 2^N$

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

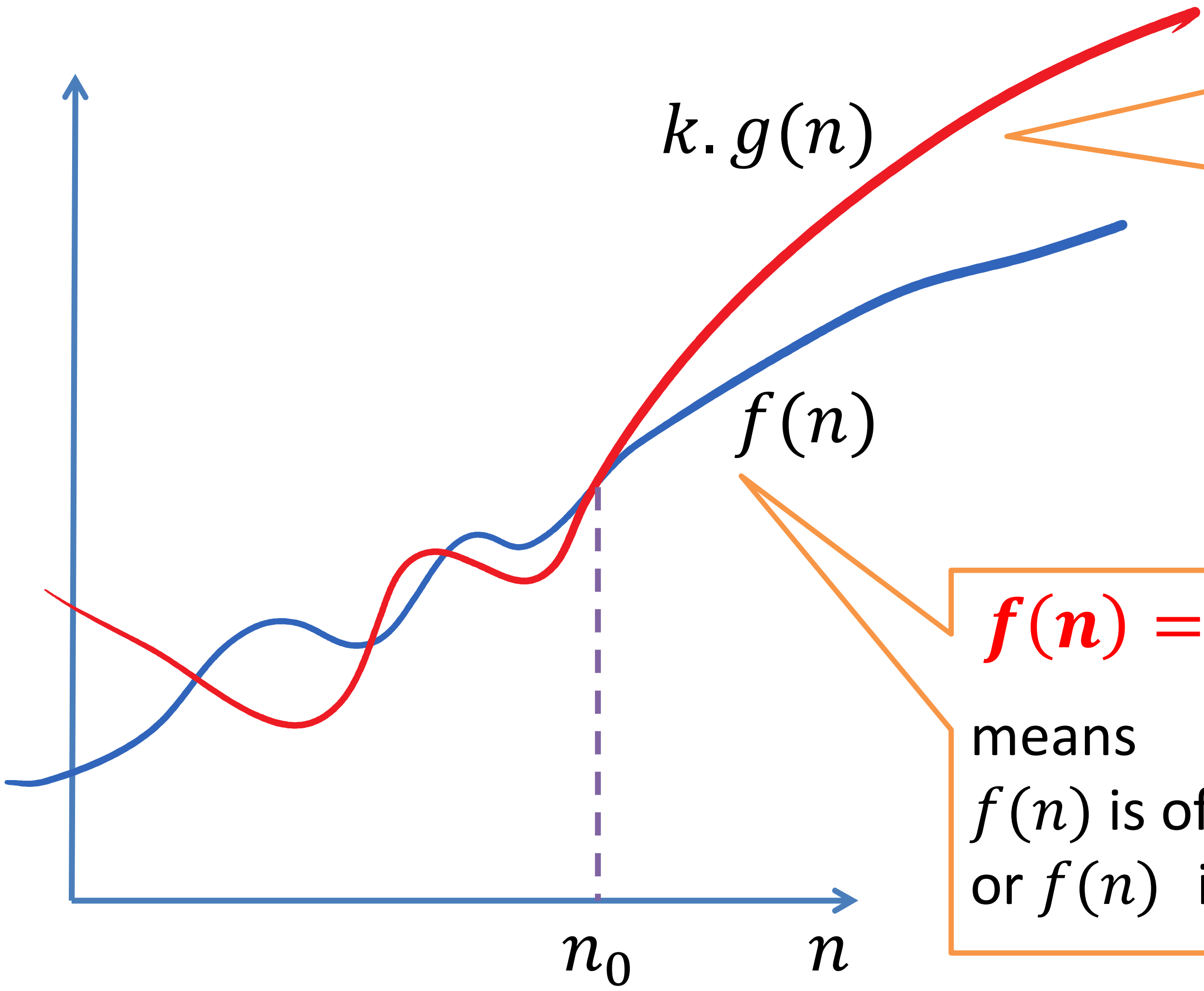
Types of Analyses

- 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

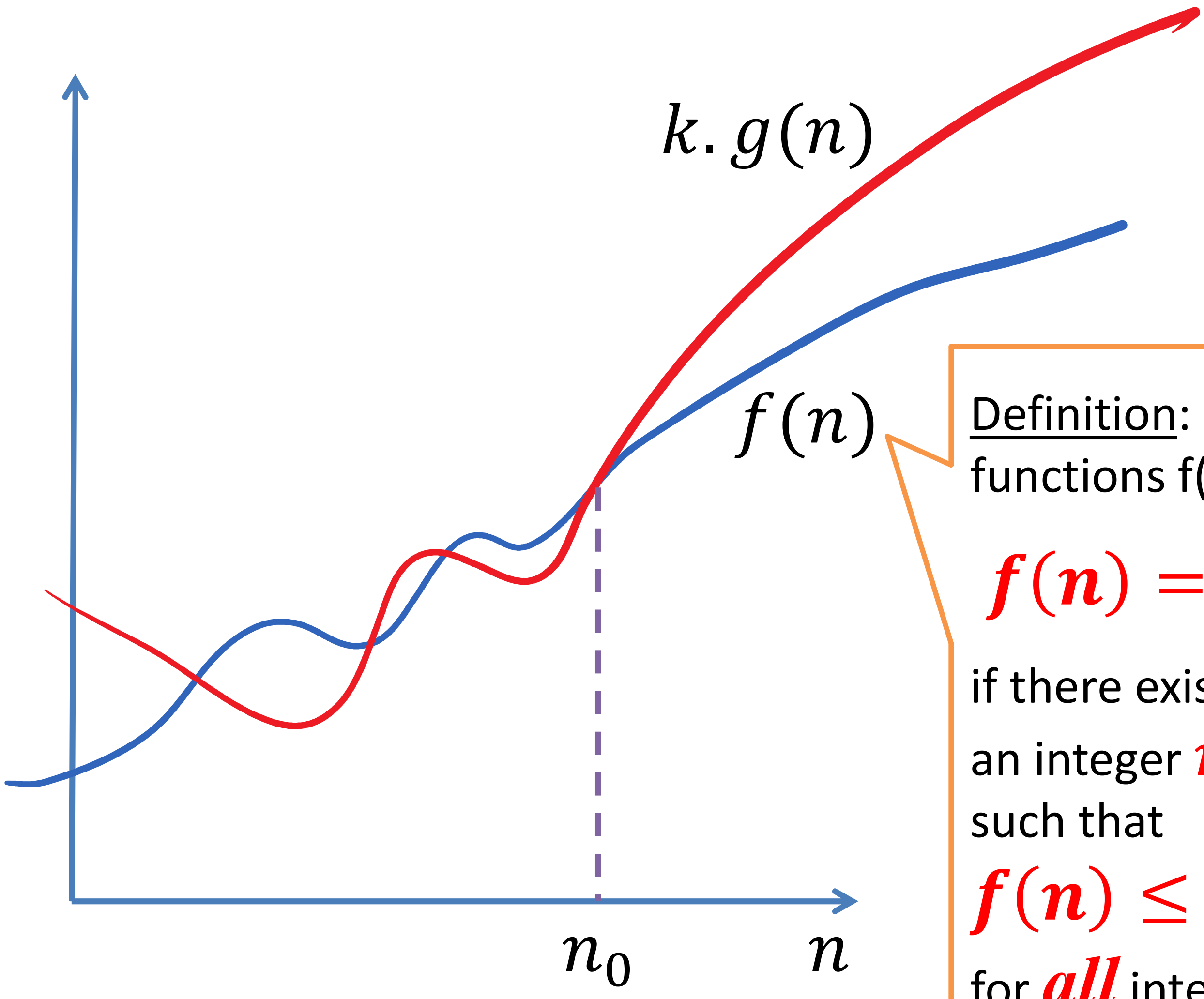


$g(n)$ is an asymptotic upper bound for $f(n)$.

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

means
$f(n)$ is of order at most $g(n)$;
or $f(n)$ is big-oh of $g(n)$.

Big-Oh Notation - upper bound



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

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

if there exists

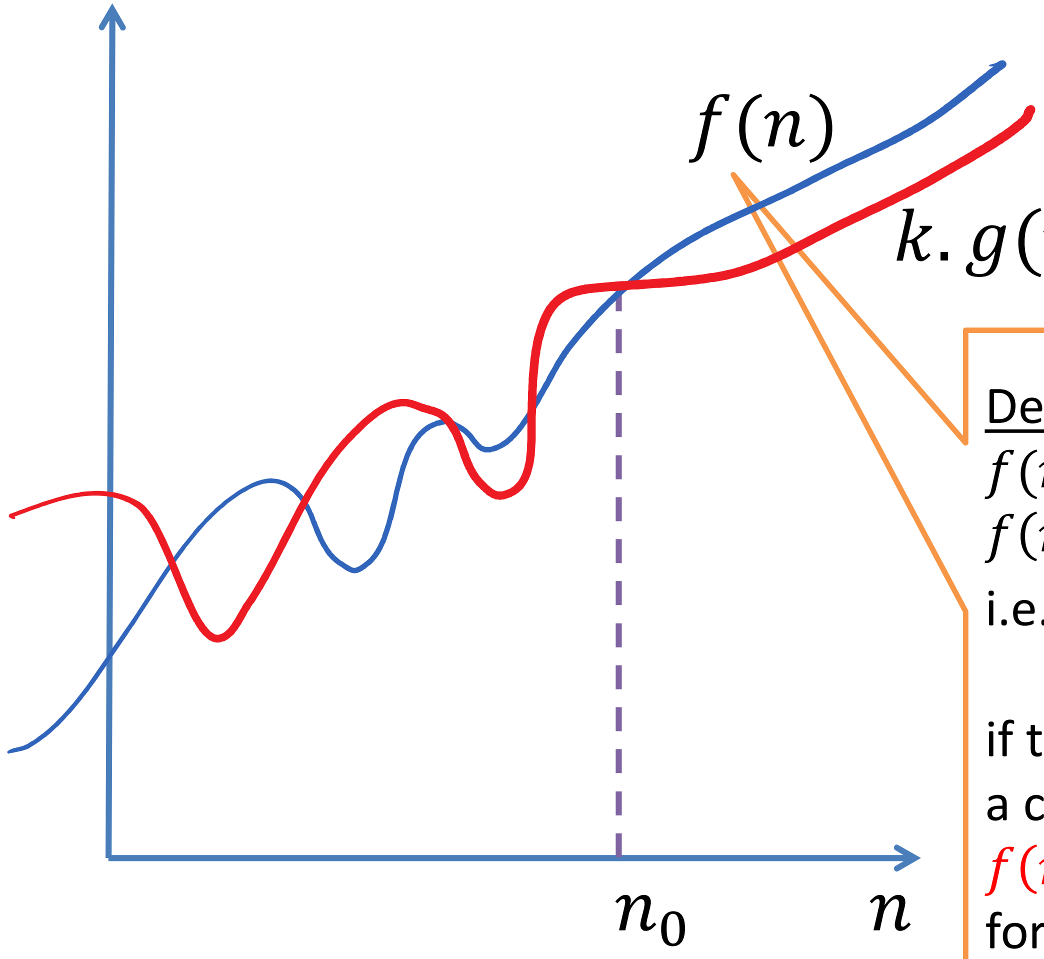
an integer n_0 & a constant $k > 0$

such that

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

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

Big-Omega Notation - lower bound

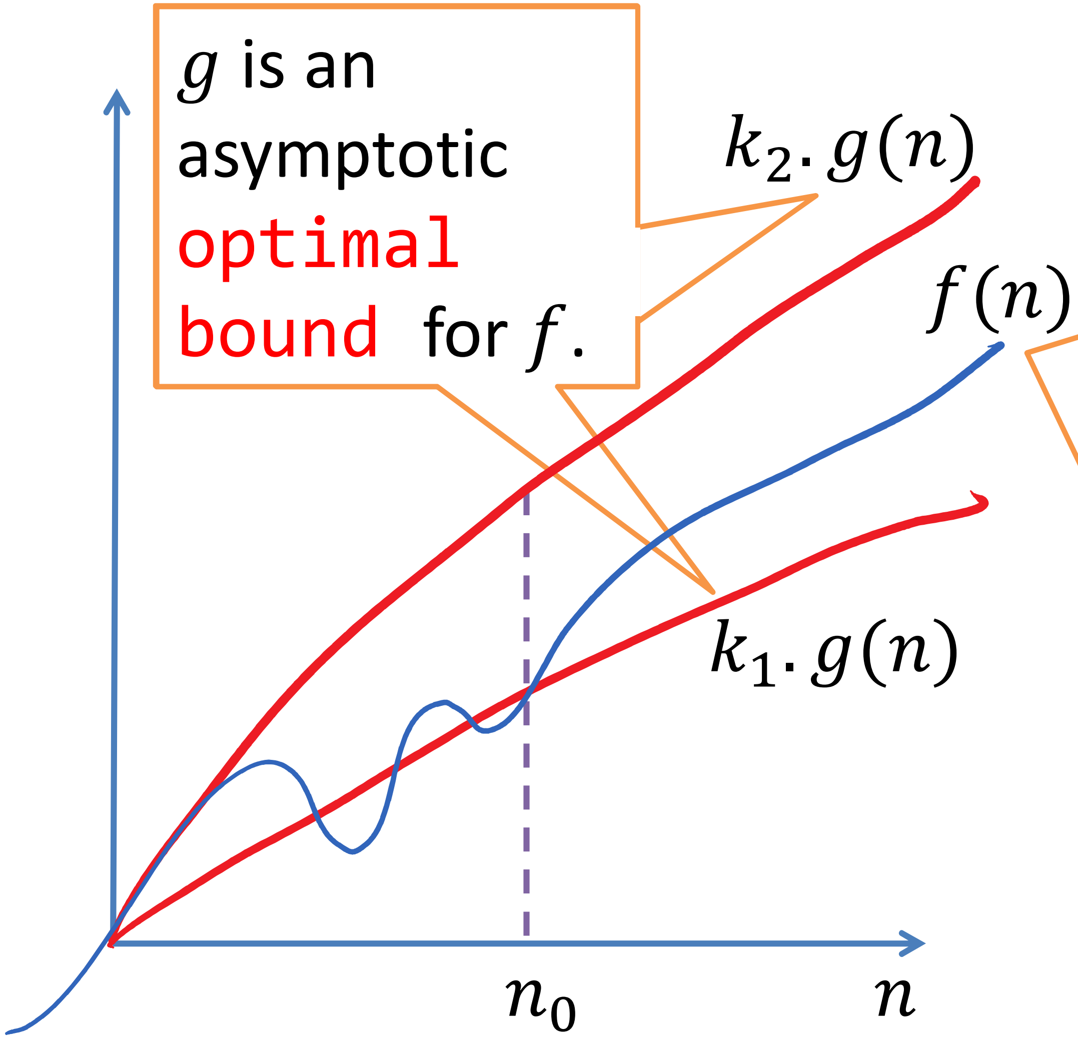


g is an asymptotic **lower bound** for f.

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



<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

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

Example 1: O(n)

```python
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))

```
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: $O(n^2)$

```python
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)

```
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 $\text{triple nested } for$ 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: $O(n^2)$

```
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:

   $= 1+2+3+\ldots+(n-2)+(n-1)+n$

   $= \frac{1}{2}\, n(n+1)$

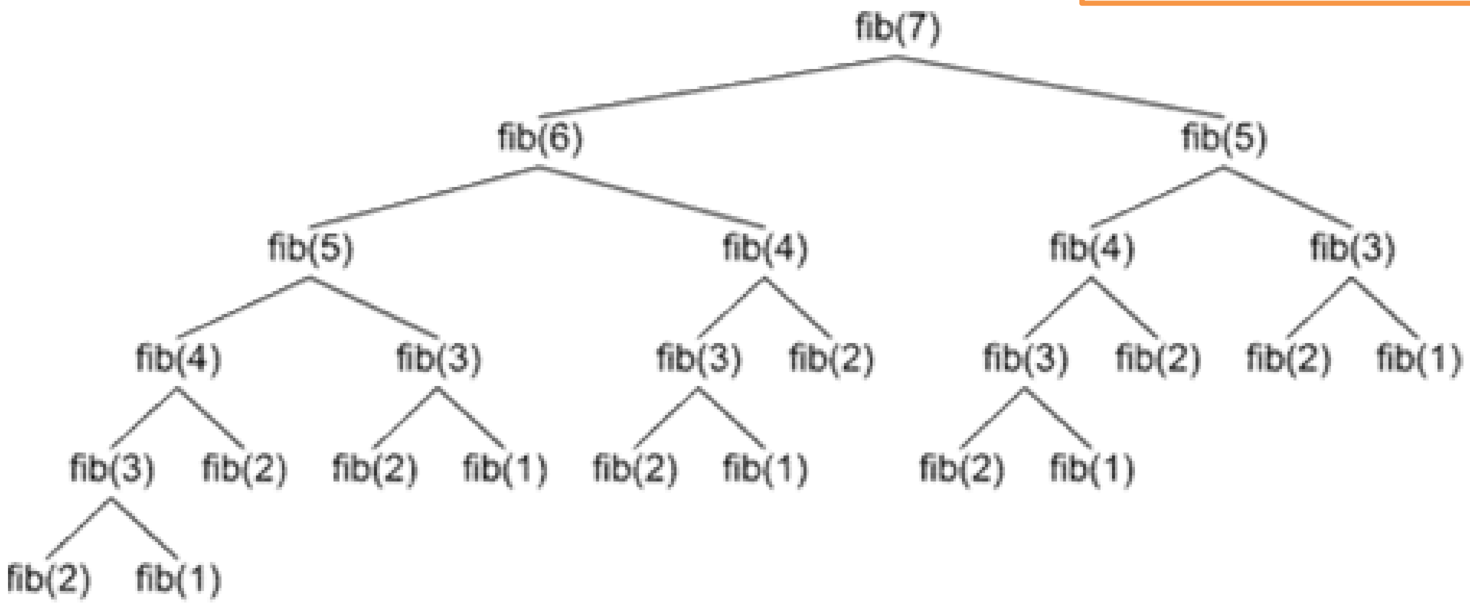
   $= \frac{1}{2}\,(n^2+n)$

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.



Others

- Linearithmic $O(n \lg(n))$ – Merge Sort