VANDERBILT UNIVERSITY School of Engineering

Discrete Structures

CS 2212

(Fall 2020)

14 – Algorithms

Chapter - 6

Computation and Algorithms

Introduction to Algorithm Analysis

Algorithm Analysis:

- A step-by-step method for solving a problem.
- How do we know it’s a “good” or an “efficient” algorithm?
- What does “good” or “efficient” mean here?
- How should we analyze an algorithm?

Time Complexity	How much “time” does the algorithm take to return an output?	Computational complexity
Space Complexity	How much “space and memory” resources are required to run the algorithm?	
Performance	Does the algorithm return an “exact” solution or an “approximate” solution?	

Introduction to Algorithm Analysis

Algorithms may perform differently, that is more or less efficient in different situations (for instance, for some inputs more efficiently than the other).

Time Complexity

Space Complexity

Performance

- **Best Case** Scenario
- **Worst Case** Scenario
- **Average Case** Scenario

How can we formalize these ideas, and develop tools to quantify the above aspects for any algorithm?

Analyzing Algorithms – Time Complexity

We count the total number of **“atomic operations”** performed by the algorithm. Each atomic operation takes a **unit time**.

The number of operations depend on the **size of input**, and so the time complexity is a function of the input size typically.

Analyzing Algorithms

Example:

```
x = fixed number
For y = 1  till n
    Compare x and y
End
```

Total operations: n
(Time complexity)

Analyzing Algorithms

Example:

```
For x = 1  till n
    For y = 1  till n
        Some "operation" that depends on x and y.
    End
End
```

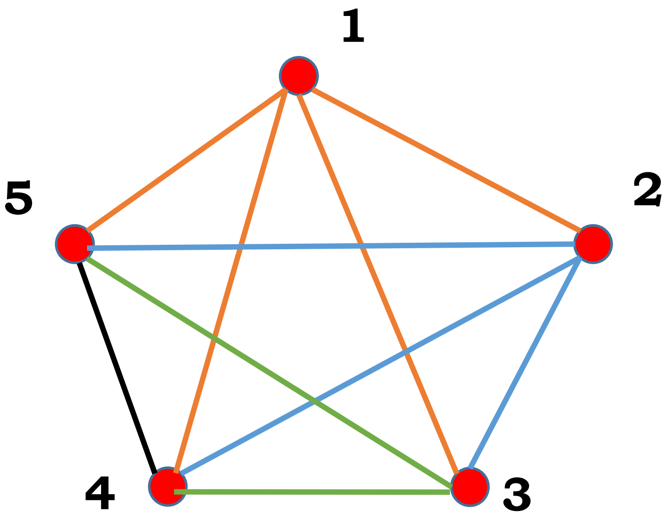
Total operations: n^2
(Time complexity)

Analyzing Algorithms

Problem: Given a set of n points, connect every pair of points by drawing a line between them.

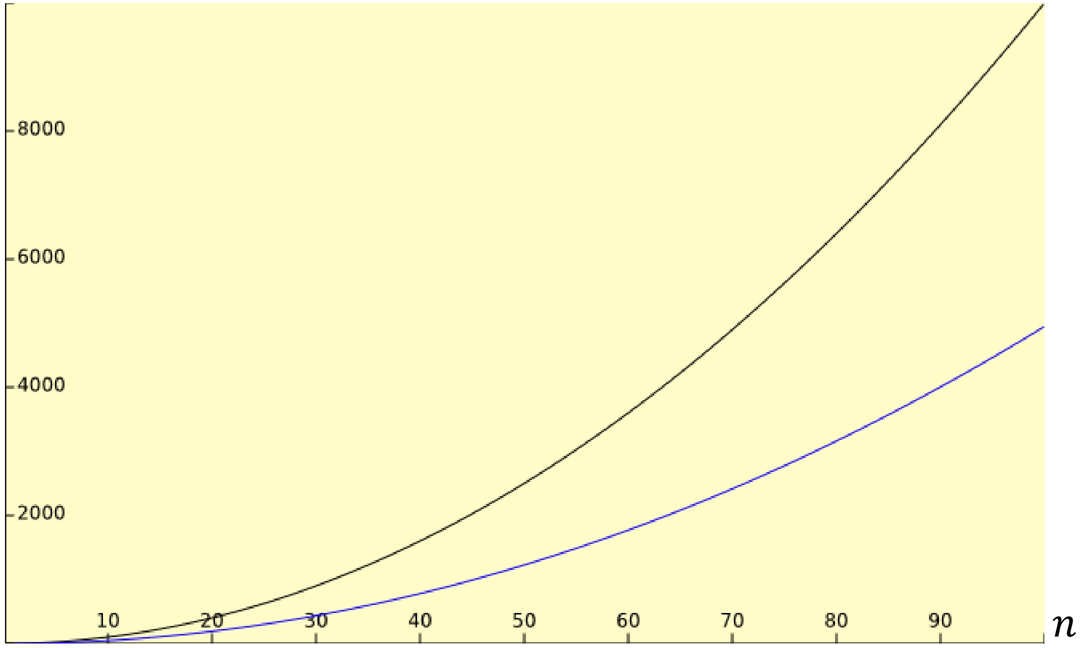
```
For i = 1  till n-1
    For j = i+1  till n
        Draw a line between
        point i and j.
    End
End
```

Total operations: $\frac{n\,(n-1)}{2}$



Is it ok to say that time complexity of the Algorithm is n^2 instead of $\frac{n\,(n-1)}{2}$?

Analyzing Algorithms



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

Black = n^2

Next, we will see a formal way to represent the complexity of an algorithm as a function of its input size.

Time Complexity Analysis

No. of operations in the algorithm

“Exact” expression

(in terms of input size n)

“Approximate” expression

(in terms of input size n)

Could be hard to find.

Simple but only give bounds.

Asymptotic Growth of Functions

Asymptotic Growth of the function f measures:

how fast the output $f(n)$ grows as the input n grows.

The actual expression (closed form) of f may be too complex.

Our goal is to see if we can represent the limiting behavior of $f(n)$ using some simpler functions that can give us a good idea of how fast the function grows with n.

Asymptotic Growth of Functions

Asymptotic Growth of the function f measures:

how fast the output $f(n)$ grows as the input n grows.

We can go for the upper and lower bounds of $f(n)$ under “certain conditions”.

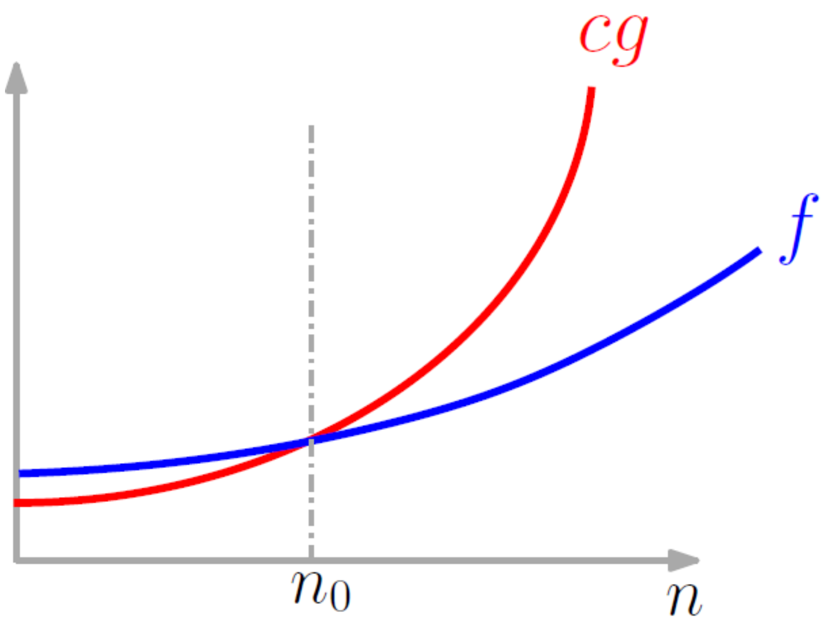
$f(n)$ using some simpler functions that can give us a good idea of how fast the function grows with n.

Comparing Growth Rates

Big O:

The notation **f = O(g)** is read "f is big-Oh of g".

$$f(n) \leq c\,g(n)$$



Loosely speaking, f is O(g) means there is a constant c such that when $f(n)$ and c·$g(n)$ are graphed, the graph of c·$g(n)$ will eventually cross $f(n)$ and will remain higher than $f(n)$, as n gets large.

Comparing Growth Rates – Big O

Let *f* and *g* be two functions from $\mathbf{Z}^+$ to $\mathbf{Z}^+$. Then *f* = O(*g*) if there are positive constants c and n_0 such that for any $n \geq n_0$,

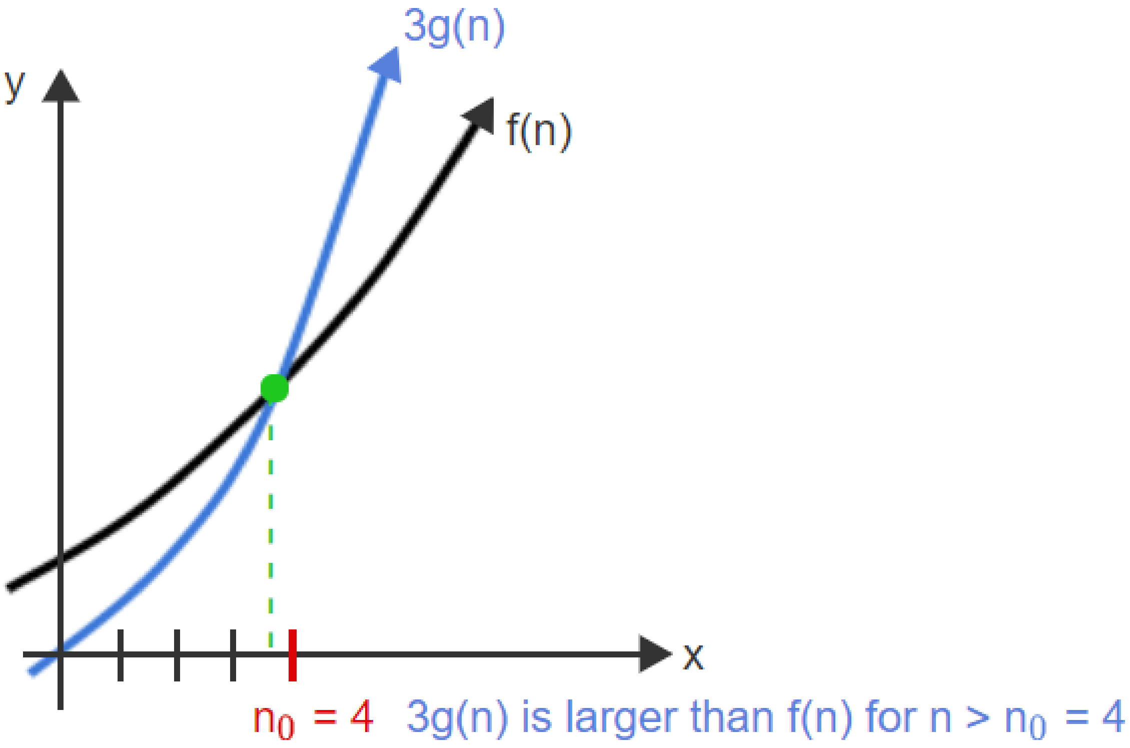
$$f(n) \leq c\, g(n).$$

$$f(n) = 2n^3 + 3n^2 + 7$$

$$g(n) = n^3$$

$$f(n) \leq c\, g(n), \qquad \forall\, n \geq n_0$$

$$c = 3\,,\ n_0 = 4$$



3g(n) is larger than f(n) for n > n_0 = 4

Big O – Example

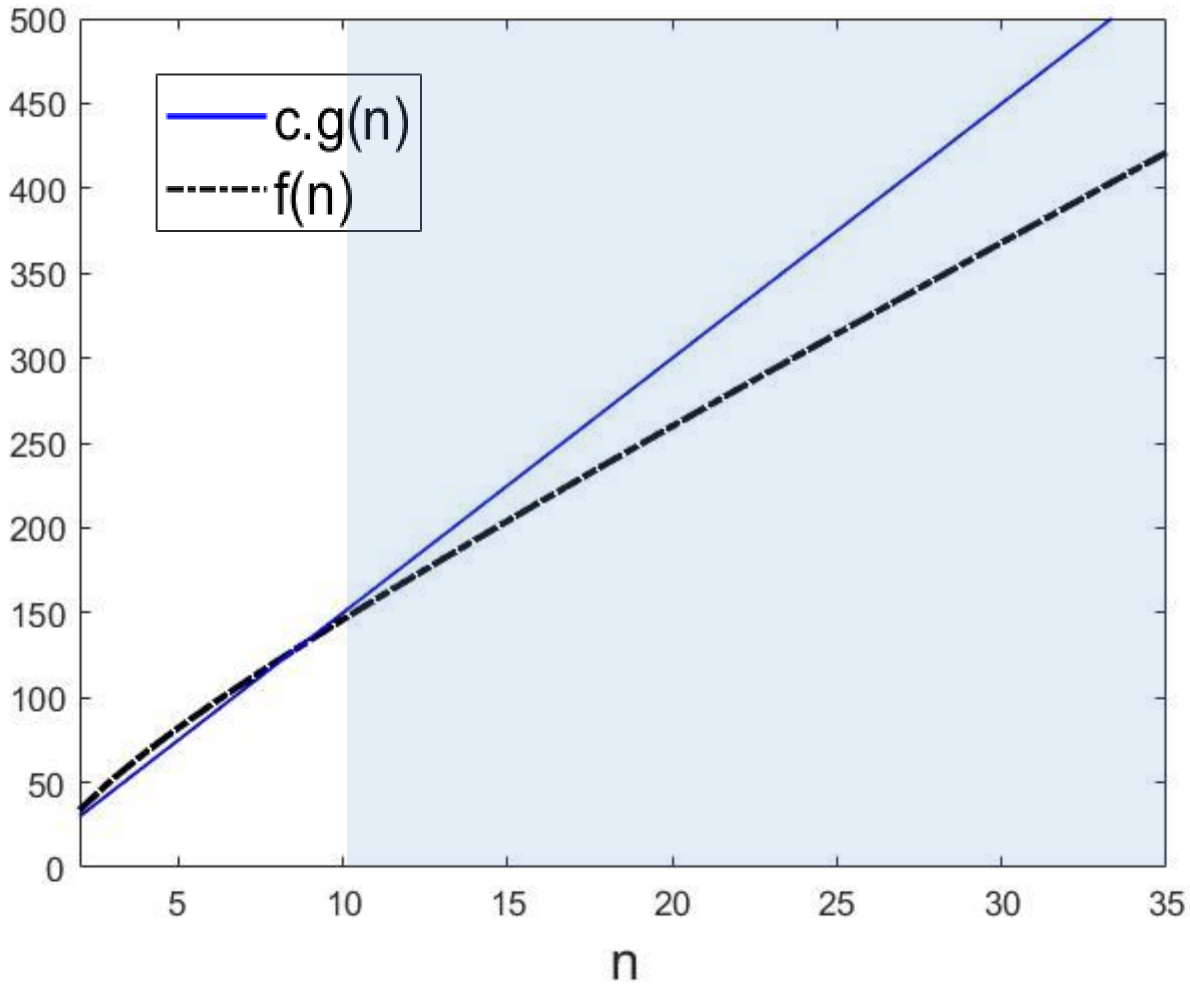
Example:

$$f(n) = 20 \log n + 10\, n$$

$$g(n) = n$$

f is O(n)

- Here $c = 15$, and $n_0 = 10$.
- Note that for $n \geq n_0$, $cg(n)$ is an upper bound of $f(n)$.



Comparing Growth Rates – Big O

Rules for Asymptotic Growth:

Let f, g, and h be functions from $\mathbf{R}^+$ to $\mathbf{R}^+$:

If $f = O(h)$ AND $g = O(h)$, then

$f+g = O(h)$.

$f(n) \leq c_1\ h(n), \forall n \geq n_1$

$g(n) \leq c_2\ h(n), \forall n \geq n_2$

$$f(n)+g(n) \leq (c_1 + c_2)h, \forall n \geq (n_1 + n_2)$$

$$\boxed{f+g \leq ch} \longleftrightarrow \boxed{f+g = O(h)}$$

Comparing Growth Rates – Big O

Rules for Asymptotic Growth:

Let f, g, and h be functions from $\mathbf{R}^+$ to $\mathbf{R}^+$:

- If $f = O(h)$ AND $g = O(h)$, then

$$f+g = O(h).$$

- If $f = O(g)$ and c is a constant greater than 0, then

$$c \cdot f = O(g).$$

- If $f = O(g)$ and $g = O(h)$, then

$$f = O(h).$$

Comparing Growth Rates – Big O

Example:

$f(n) = 5n^3 + 16(n \log n) + 5 \cdot 2^n$

$5n^3$ is $O(n^3)$ and n^3 is $O(2^n)$ $\Rightarrow$ $5n^3$ is $O(2^n)$

$16(n \log n)$ is $O(n \log n)$ and $(n \log n)$ is $O(2^n)$ $\Rightarrow$ $16(n \log n)$ is $O(2^n)$

$5 \cdot 2^n$ is $O(2^n)$

$f(n) = 5n^3 + 16(n \log n) + 5 \cdot 2^n$ is $O(2^n)$

Comparing Growth Rates – Big O

Remark:

Big-O is an **upper limit bound** that says the algorithm will do **no worse** than such-and-such.

If you don't have a **good** upper bound, the information from big-O can be meaningless.

In other words, if an algorithm does $200n$ operations, it's $O(n)$, but it's also $O(n^2)$ and $O(n!)$ according to the definition. Which one tell us the most about the algorithm?

Comparing Growth Rates – Big O

Summary:

We write	$f(n) = O(g(n))$
We say	$f(n)$ is big O ("Oh") of $g(n)$
Remember	O(.) is solely an **upper bound**

Examples:

$$200n = O\left(\frac{n}{100}\right) = O(n)$$

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

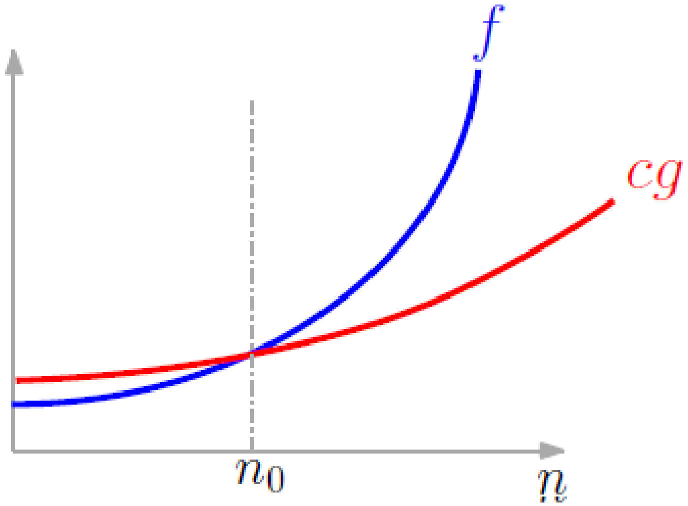
$$O(n \log n) + O(n^3) = O(n^3)$$

Comparing Growth Rates – Big Ω

Big Omega Ω :

The notation $f = \Omega(g)$ is read "*f* is Omega of *g*".

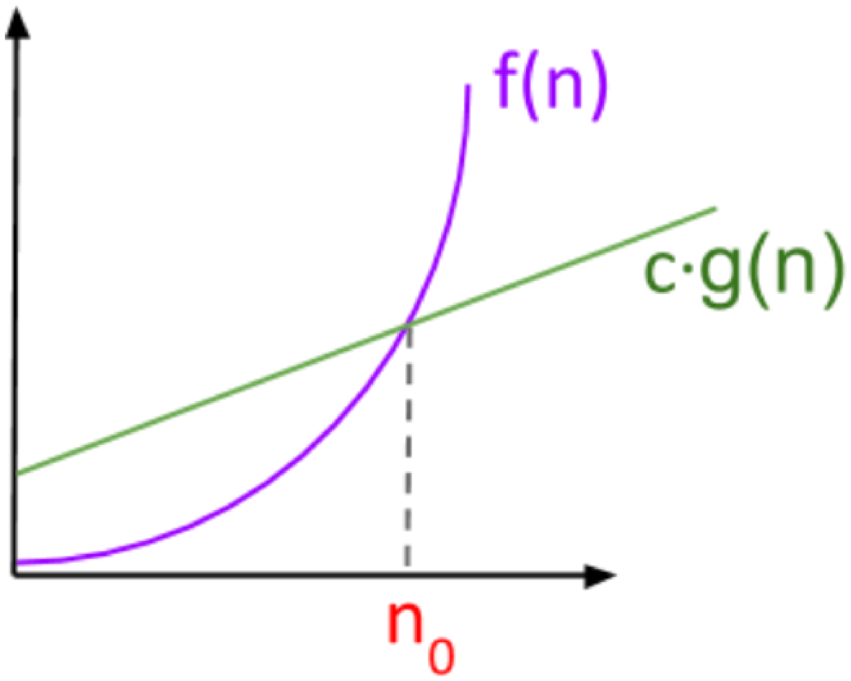
$$f(n) \qquad \geq \qquad c\,g(n)$$



Comparing Growth Rates – Big Ω

Let f and g be two functions from $\mathbf{Z}^+$ to $\mathbf{Z}^+$. Then $f = \Omega(g)$ if there are positive constants c and n_0 such that for any $n \geq n_0$,

$$f(n) \geq c\, g(n).$$



Comparing Growth Rates – Big Ω

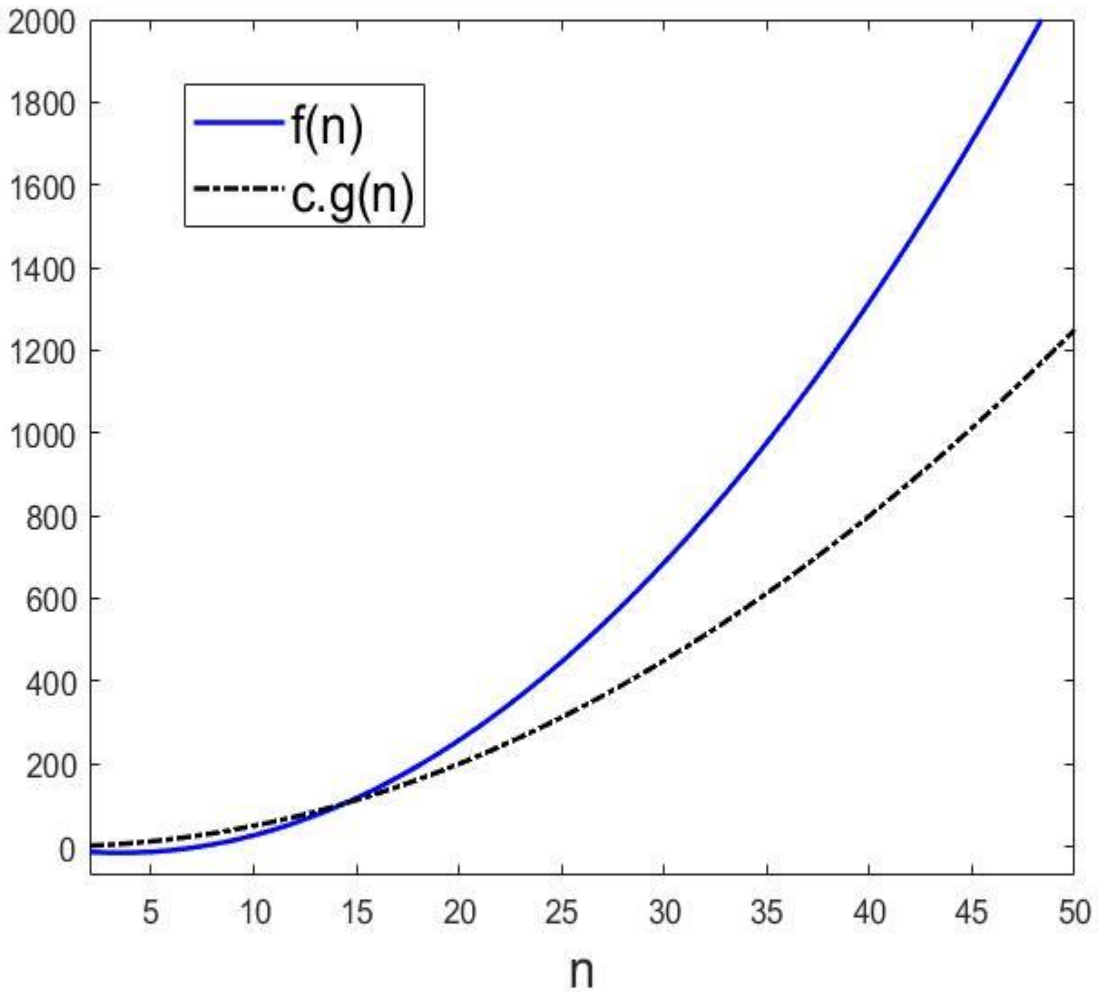
Example:

$$f(n) = n^2 - 7n - 3$$

$$g(n) = n^2$$

f is $\Omega(n^2)$

- Here $c = 1/2$, and $n_0 = 20$.
- Note that for $n \geq n_0$, $g(n)$ is a lower bound of $f(n)$.



Comparing Growth Rates – Big Ω

Relationship of O and Ω Notations.

Let f and g be two functions from $\mathbf{Z}^+$ to $\mathbf{Z}^+$. Then,

$f = \Omega(g)$ if and only if $g = O(f)$.

$$f = \Omega(g)$$
$$f \geq cg$$
$$(1/c)\, f \geq g$$

$$g \leq (1/c)\, f$$
$$g = O(f)$$

Comparing Growth Rates – Big Ω

Rules for Asymptotic Growth:

Let f, g, and h be functions from $\mathbf{R}^+$ to $\mathbf{R}^+$:

- If $f = \Omega(h)$ **OR** $g = \Omega(h)$, then

  $$f+g = \Omega(h).$$

- If $f = \Omega(g)$ and c is a constant greater than 0, then

  $$c \cdot f = \Omega(g).$$

- If f = $\Omega(g)$ and $g = \Omega(h)$, then

  $$f = \Omega(h).$$

Comparing Growth Rates – Big Ω

Remark:

Big- Ω is a **lower limit bound** that says the algorithm will do **at least** such-and-such.

If you don't have a **good** lower bound, the information from big-Ω can be meaningless.

In other words, if an algorithm does $200n^5$ operations, it's $\Omega(n^5)$, but it's also $\Omega(n)$ and $\Omega(\log n)$ according to the definition. Which one tell us the most about the algorithm?

Comparing Growth Rates – Big O

Summary:

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

We say $f(n)$ is big O ("Oh") of $g(n)$

Remember O(.) is solely an **upper bound**

Examples:

$$200n = O\left(\frac{n}{100}\right) = O(n)$$

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

$$O(n \log n) + O(n^3) = O(n^3)$$

Comparing Growth Rates – Big O

Summary:

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

We say $f(n)$ is big O (“Oh”) of $g(n)$

Remember O(.) is solely an **upper bound**

Examples:

$$200n = O\left(\frac{n}{100}\right) = O(n)$$

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

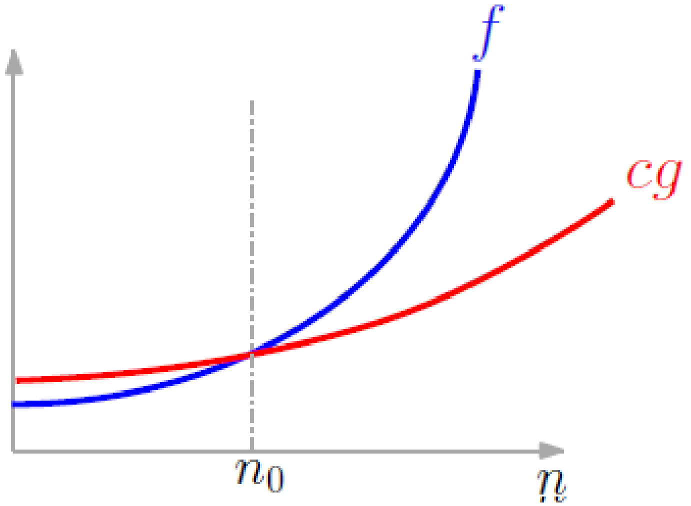
$$O(n \log n) + O(n^3) = O(n^3)$$

Comparing Growth Rates – Big Ω

Big Omega Ω :

The notation $f = \Omega(g)$ is read “*f* is Omega of *g*”.

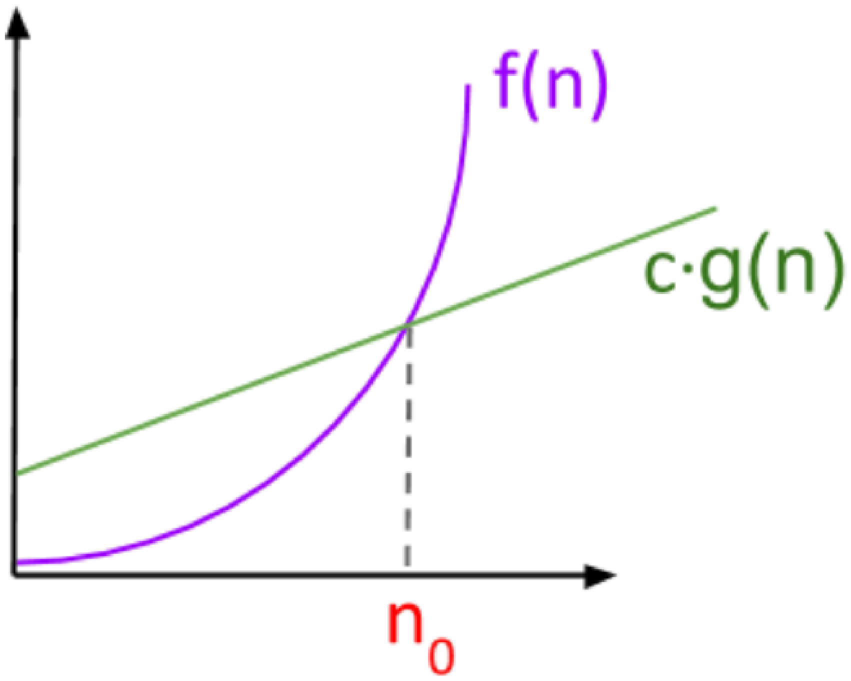
$$f(n) \geq c\,g(n)$$



Comparing Growth Rates – Big Ω

Let f and g be two functions from Z^+ to Z^+. Then $f = \Omega(g)$ if there are positive constants c and n_0 such that for any $n \geq n_0$,

$$f(n) \geq c\, g(n).$$



Comparing Growth Rates – Big Ω

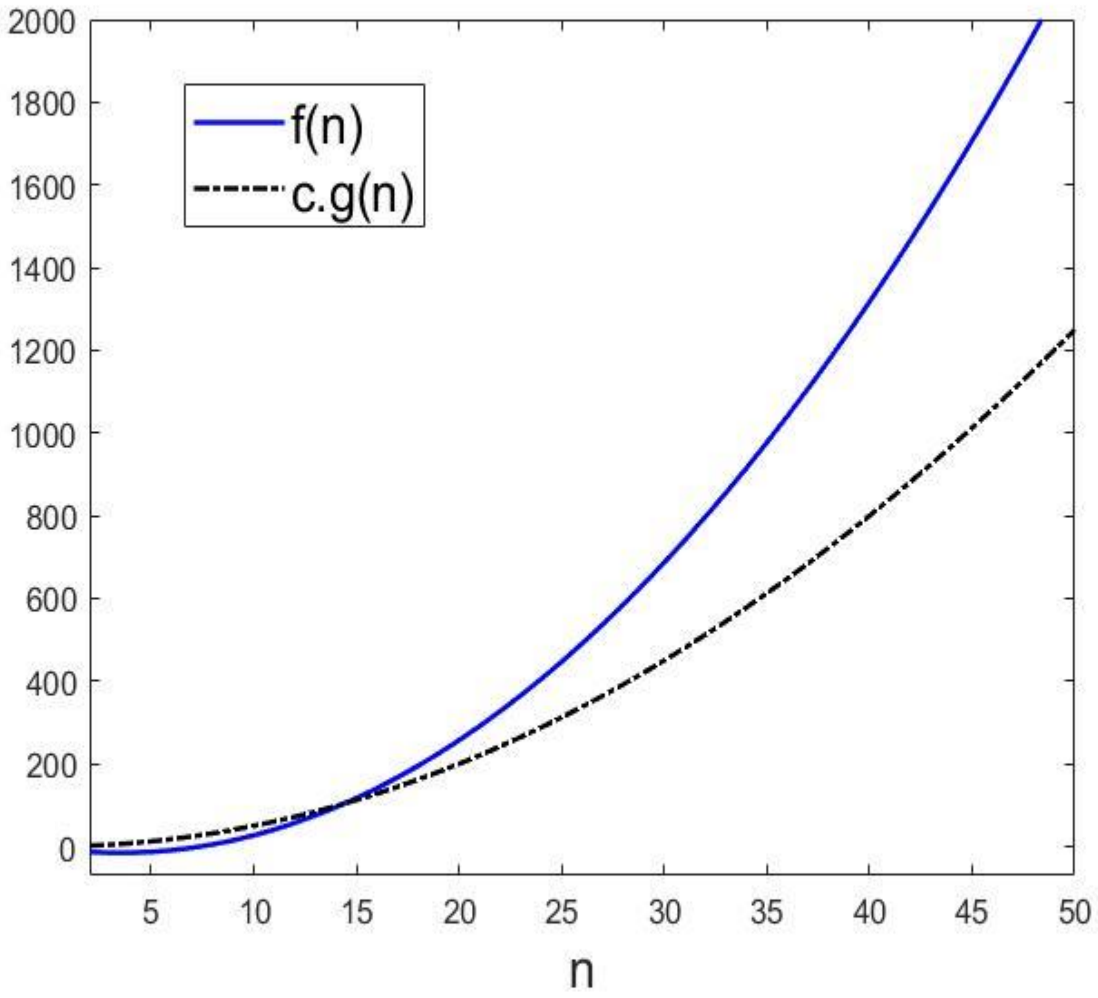
Example:

$$f(n) = n^2 - 7n - 3$$

$$g(n) = n^2$$

f is $\Omega(n^2)$

- Here $c = 1/2$, and $n_0 = 20$.
- Note that for $n \geq n_0$, $cg(n)$ is a lower bound of $f(n)$.



Comparing Growth Rates – Big Ω

Relationship of O and Ω Notations.

Let f and g be two functions from $\mathbf{Z}^+$ to $\mathbf{Z}^+$. Then,

$f = \Omega(g)$ if and only if $g = O(f)$.

$$f = \Omega(g)$$
$$f \geq cg$$
$$(1/c)f \geq g$$

$$g \leq (1/c)f$$
$$g = O(f)$$

Comparing Growth Rates – Big Ω

Rules for Asymptotic Growth:

Let f, g, and h be functions from $\mathbf{R}^+$ to $\mathbf{R}^+$:

- If $f = \Omega(h)$ **OR** $g = \Omega(h)$, then
$$f+g = \Omega(h).$$
- If $f = \Omega(g)$ and c is a constant greater than 0, then
$$c\cdot f = \Omega(g).$$
- If f = $\Omega(g)$ and $g = \Omega(h)$, then
$$f = \Omega(h).$$

Comparing Growth Rates – Big Ω

Remark:

- Big- Ω is a **lower limit bound** that says the algorithm will do **at least** such-and-such.
- If you don't have a **good** lower bound, the information from big-Ω can be meaningless.
- In other words, if an algorithm does $200n^5$ operations, it's $\Omega(n^5)$, but it's also $\Omega(n)$ and $\Omega(\log n)$ according to the definition. Which one tell us the most about the algorithm?

Comparing Growth Rates – Big Ω

Summary:

We write	$f(n) = \Omega(g(n))$
We say	$f(n)$ is big omega of $g(n)$
Remember	Ω(.) is simply a **lower bound**

Examples:

$$200n = \Omega\left(\frac{n}{100}\right) = \Omega(n)$$

$$n^2 = \Omega\ (n)$$

$$n^2 = \Omega(n \log n)$$

Comparing Growth Rates – Big Theta (Θ)

Big Theta Θ :

The notation $\boldsymbol{f = \Theta(g)}$ is read "f is big theta of g".

Loosely speaking, f is $\Theta(g)$ means the algorithm sandwiched between the same upper and lower bound. This gives us a precise measure of work.

Comparing Growth Rates – Big Theta (Θ)

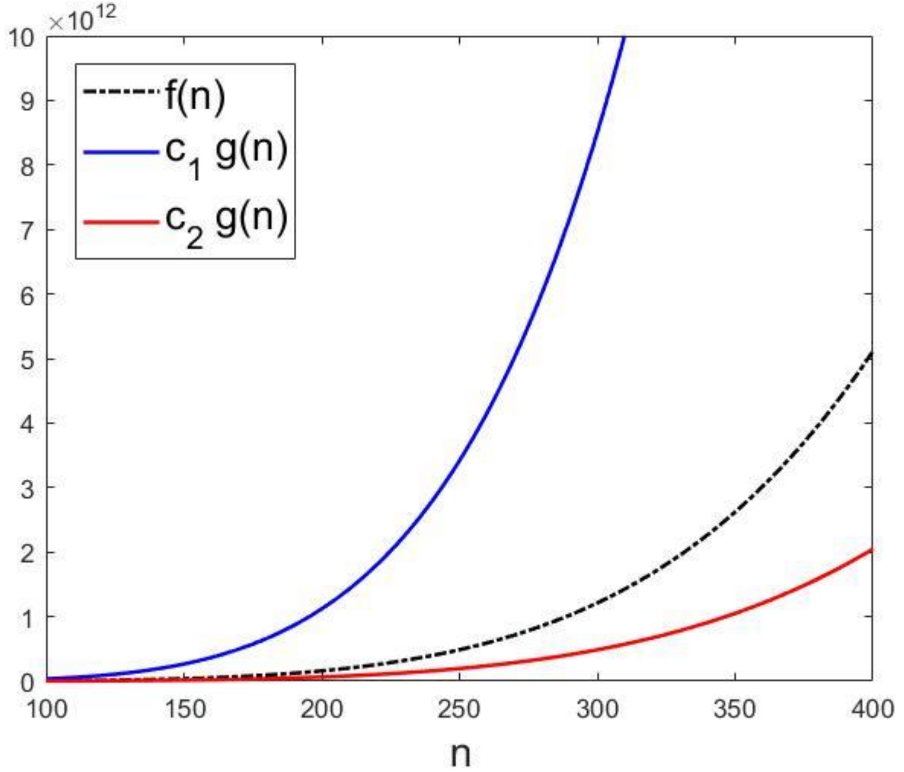
Let f and g be two functions from Z^+ to Z^+. Then $f = \Theta(g)$ if and only if

$$f = O(g) \text{ and } f = \Omega(g).$$

$$f(n) = 0.5n^5 - 100n^3 + 3n - 1$$

$$g(n) = n^5$$

f is $\Theta(n^5)$



Comparing Growth Rates – Big Theta (Θ)

We write $f(n) = \Theta(g(n))$

We say $f(n)$ is big theta of $g(n)$

Translation: If two functions have a running time that differs by a constant, we say they have the same growth rate.

Comparing Growth Rates – Big Theta (Θ)

Example: Show: $\frac{n(n+1)}{2} = \Theta(n^2)$; $n \geq 0$

Approach: We need to demonstrate two things. Show that:

1. the function $\frac{n(n+1)}{2}$ does at most n^2 work, $O(n^2)$
2. the function $\frac{n(n+1)}{2}$ does at least n^2 work. $\Omega(n^2)$

Conclude that n^2 is a tight bound on the work and we can say it is $\Theta(n^2)$.

Comparing Growth Rates – Big Theta (Θ)

Example: Show: $\frac{n(n+1)}{2} = \Theta(n^2)$; $n \geq 0$

$$\frac{n(n+1)}{2} \leq cn^2 \qquad ???$$

How about $c = 2$,

$$\frac{n(n+1)}{2} \leq 2n^2 \qquad ✓$$

$\frac{n(n+1)}{2}$ is **O(n^2)**

Comparing Growth Rates – Big Theta (Θ)

Example: Show: $\frac{n(n+1)}{2} = \Theta(n^2); \quad n \geq 0$

$$\frac{n(n+1)}{2} \geq cn^2 \qquad ???$$

How about $c = 1/2$

$$\frac{n(n+1)}{2} \geq (1/2)n^2 \qquad ✓$$

$$\frac{n(n+1)}{2} \text{ is } \boldsymbol{\Omega\ (n^2)}$$

Comparing Growth Rates – Big Theta (Θ)

Example: Show: $\frac{n(n+1)}{2} = \Theta(n^2)$; $n \geq 0$

$\frac{n(n+1)}{2}$ is $O(n^2)$.

$\frac{n(n+1)}{2}$ is $\Omega(n^2)$.

Conclusion: $\frac{n(n+1)}{2}$ is $\Theta(n^2)$, which is a tight bound on the amount of work.

Comparing Growth Rates – Big Theta (Θ)

Example: Show: $\log(n + 1) = \Theta(\log n)$

Approach: Again, we need to demonstrate two things to make our big-theta case...

1. The function does at most $\log(n)$ work to say it is $O(\log n)$.
2. The function does at least $\log(n)$ work to say that it is $\Omega(\log n)$.
3. Conclude the function does **$\Theta(\log n)$** work, which is a tight bound.

Comparing Growth Rates – Big Theta (Θ)

Example: Show: $\log(n + 1) = \Theta(\log n)$

1. Show $\log (n + 1)$ is $O(\log n)$.
   - We know

     $$n < n + 1 < n^2 \quad \text{for } n \geq 2.$$

   - So it follows that

     $$\log n < \log (n + 1) < \log n^2 = 2 \log n \quad \text{for } n \geq 2.$$

   - Since $\log (n + 1) < 2 \log n$, we conclude that

     $$\boldsymbol{\log (n + 1) = O(\log n)}.$$

Comparing Growth Rates – Big Theta (Θ)

Example: Show: $\log(n + 1) = \Theta(\log n)$

2. Show $\log (n + 1)$ is $\Omega(\log n)$
   - We observe that

     $$\log (n + 1) > \log n \,;$$

   - So we can conclude it is **$\Omega(\log n)$.**

3. Since $\log (n + 1)$ is both $O(\log n)$ and $\Omega(\log n)$ we conclude that the function **$\log(n + 1) = \Theta(\log n)$**.

Comparing Growth Rates – Big Theta (Θ)

Theorem: If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$ and $c \neq 0$ and $c \neq \infty$, then

$$f(n) = \Theta\left(g(n)\right)$$

Translation: If two functions have a running time that differs by a constant, we say they have the same growth rate.

	n = 32	n = 1000	n = 10,000	n = 1,000,000	n = 100,000,000
$f(n) = \frac{n(n-1)}{2}$	496	499,500	49,995,000	499,999,500,000	4,999,999,950,000,000
$g(n) = n^2$	1024	10^6	10^8	10^{12}	10^{18}
$\frac{f(n)}{g(n)}$	0.4843	0.4995	0.49995	0.4999995	0.499999995

Comparing Growth Rates – Big Theta (Θ)

Let’s look at how the ratio of rates of growth change with algorithms that run at different rates.

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

	n = 1000	n = 10,000	n = 1,000,000	n = 100,000,000
$g(n) = n$; $\frac{f(n)}{g(n)}$	499	49,995	499,999	4,999,999
$g(n) = \sqrt{n}$; $\frac{f(n)}{g(n)}$	15,795	499,950	499,999,500	499,999,995,000
$g(n) = \log n$; $\frac{f(n)}{g(n)}$	166,500	12,498,750	83,333,250,000	555,555,550,111,111
$g(n) = \boldsymbol{n^2}$; $\boldsymbol{\frac{f(n)}{g(n)}}$	0.4995	0.49995	0.4999995	0.499999995

Comparing Growth Rates – Big Theta (Θ)

Order of

If $f = \Theta(g)$, then f is said to be order of g.

Constant function

A function that does not depend on n at all is called a constant function.

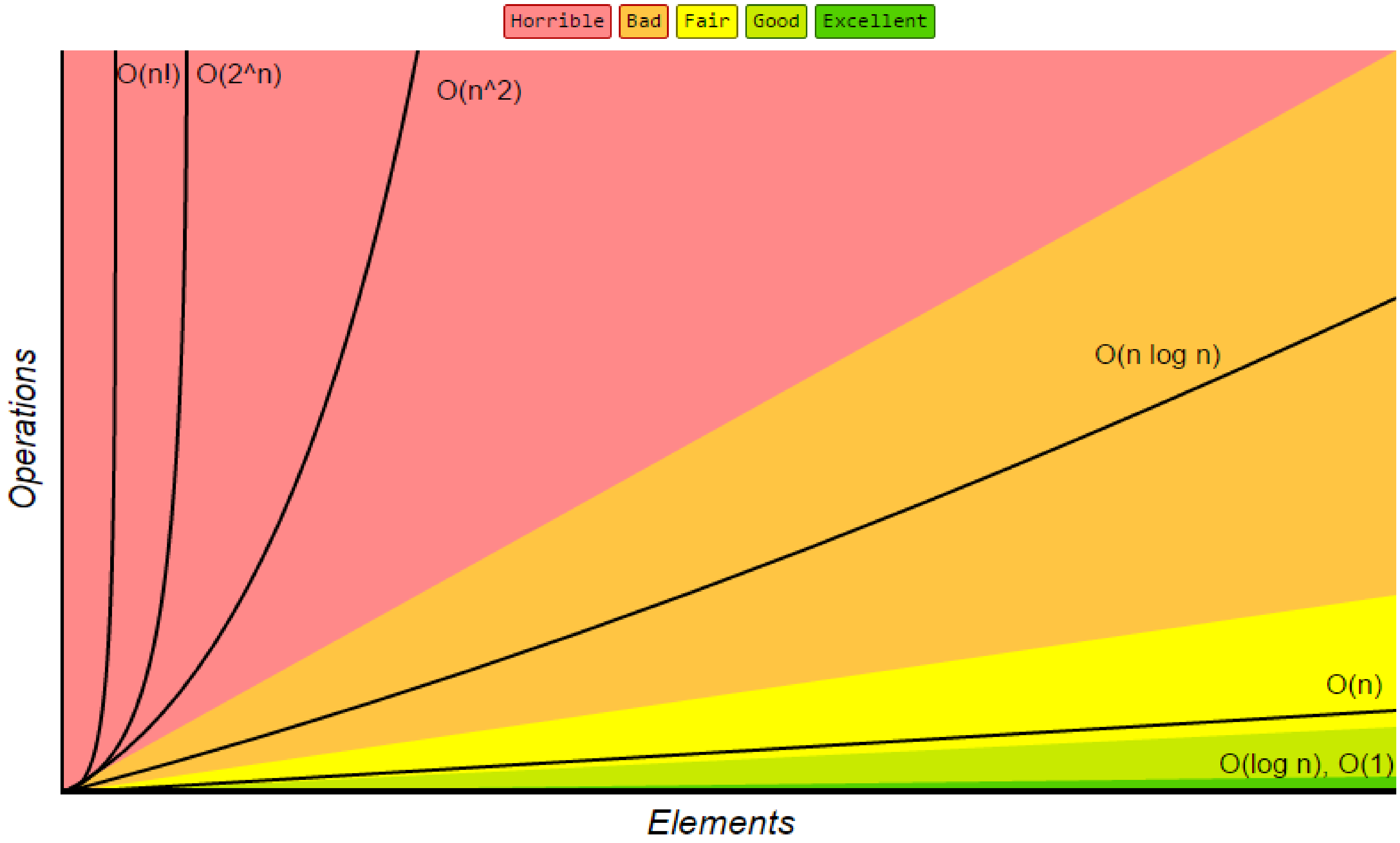
Polynomial

A function $f(n)$ is said to be polynomial if $f(n)$ is $\Theta(n^k)$ for some constant $k \geq 1$.

Comparing Growth Rates – Big Omega (Ω)

Function	**Name**
$\Theta(1)$	Constant
$\Theta(\log \log n)$	Log log
$\Theta(\log n)$	Logarithmic
$\Theta(n)$	Linear
$\Theta(n \log n)$	$n \log n$
$\Theta(n^2)$	Quadratic
$\Theta(n^3)$	Cubic
$\Theta(c^n)$, $c > 1$	Exponential
$\Theta(n!)$	Factorial

Comparing Growth Rates



Comparing Growth Rates

Exercise:

Indicate the "order of growth" relationships between the following expressions from lowest order to highest order. If two expressions have the same order, place them in a set together.

$$n!,\ \ n^2,\ \ \log n,\ \ 3^n,\ \ n^3,\ \ n \log n,\ \ \log(\log n),\ \ 2^n$$

$$\log(\log n),\ \ \log n,\ \ n \log n,\ \ n^2,\ \ n^3,\ \ 2^n,\ \ 3^n,\ \ n!$$

Comparing Growth Rates

Exercise (Round 2):

Indicate the "order of growth" relationships between the following expressions from lowest order to highest order. If two expressions have the same order, place them in a set together.

n^2, $n \log n$, $2n$, $\log(n^2)$, $(\log n)^2$, 2^n, n^3, log n, $\log(\log n)$

$\log(\log n)$, $\log(n)$, $\log(n^2)$, $(\log n)^2$, $2n$, $n \log n$, n^2, n^3, 2^n

Algorithms Analysis

Example: Algorithm to find the smallest in a sequence of numbers.

Input: a_1, a_2, ..., a_n, n the length of the sequence

Output: the smallest number in the sequence

```
min := $a_1$
For i = 2 to n
    if ( $a_i$ < min ) min := $a_i$
End-for
Return (min)
```

1 assignment op

loop iterated n - 1 times

For loop tests i and increments i (2 ops)
1 op for comparison + 1 op (in worst-case)
for assignment

1 op for return

f(n) = # of operations on a sequence of length n:

$$f(n) = (n - 1)c + d = cn - c + d = \Theta(n)$$

Algorithms Analysis

Example: Analysis of the algorithm SearchSequence.

Input: $a_1, \ldots, a_n$, the sequence, n, length of sequence, x, a number to search for
Output: Index of first occurence of x in the sequence
or -1 if x does not occur in the sequence

```
i = 1
While ( $a_i \neq$ x and i < n)
    i := i + 1
End-while
If ( $a_i$ = x ), Return ( i )
Return (-1)
```

1 assignment op

3 ops
1 assignment op

Number of iterations
in Loop ≤ n-1

if a:≠x for all:, number
of iterations = n-1

2 ops

f(n) = # of ops on sequence of length n:

$$f(n) \leq 1 + \underbrace{(3 + 1) \cdot (n-1)}_{\text{While loop}} + 2 \leq 4n - 1 = O(n)$$

Worst-case:

$$f(n) \geq 1 + (3 + 1)(n-1) + 2 = 4n - 1 = \Omega(n)$$

Algorithms Analysis

Example: Worst-case time complexity - finding the maximum value of a function.

```
Input: $a_1, a_2,…,a_n$
        n, the length of the sequence.
Output: The largest values of M on input values from the sequence.
```

```
max := M($a_1$, $a_1$, $a_1$)
```

```
For i = 1 to n
    For j = 1 to n
        For k = 1 to n
            new := M($a_i$, $a_j$, $a_k$)
            If ( new > max ), max := new
        End-for
    End-for
End-for
```

```
Return( max )
```

- All we know about **M** is that it takes three positive integers as inputs and outputs a positive integer.
- We can assume that it takes **O(1)** operations to compute the value of **M** on any input.
- At least one operation is performed in the innermost loop, so the time complexity of the algorithm is $\Omega(n^3)$.
- The number of operations performed in the worst case is at most $cn^3 + 2$, which is $O(n^3)$.
- The worst-case time complexity of the algorithm is $\Theta(n^3)$.