ECE36800 Data structures

Asymptotic Notation
Chapter 2 (pp. 27-66)

Lu Su

School of Electrical and Computer Engineering
Purdue University

Slides Courtesy: Prof. Cheng-Kok Koh

Overview

- Analyzing algorithms
- Asymptotic notation
- O(): big-O
- Hierarchy of functions
- Ω(): big-Omega
- Θ(): big-Theta
- o() and ω(): little-o and little-omega

- Reference pages: pp. 27-66

A simple algorithm

Add up the numbers from 1 to *n*

Pseudocode:

```
total ← 0;
for i ← 1 to n
    total ← total + i;
return total;
```

C program:

```
int total = 0;
for (int i = 1; i <= n; i++)
    total = total + i;
return total;
```

- Does it compute the result correctly?
- Is it as fast as it can be?
- How many machine instructions (in terms of *n*) does it take?

A simple algorithm

Add up the numbers from 1 to *n*

	cost (runtime per instruction)	frequency
`total ← 0;`	C_1	1
`for i ← 1 to n`	C_2	$n+1$
`   total ← total + i;`	C_3	n
`return total;`	C_4	1

Total runtime:

$$C_1 + C_2(n+1) + C_3 n + C_4 = (C_2+C_3)n + (C_1+C_2+C_4)$$

Another Simple Algorithm

Add up the sum of the squares of numbers from 1 to *n*, as long as the square is divisible by three or seven

	cost	frequency
`total ← 0;`	C_1	1
`for i ← 1 to n`	C_2	$n+1$
`    if (((i*i % 3) == 0) ||`	C_3	n
`        ((i*i % 7) == 0))`	C_4	$\lceil 2n/3 \rceil$
`      total ← total + i*i;`	C_5	$\lfloor n/3 \rfloor + \lfloor n/7 \rfloor - \lfloor n/21 \rfloor$
`return total;`	C_6	1

Linear Search Example

Determine the worst case time complexity

Worst case: Item not in array

```
Search (A, Item)
1   n ← length(A)
2   for i: 0 to n-1
3      if A[i] = Item
4         return True
5   return False
```

cost	frequency
C_1	1
C_2	n+1
C_3	n
C_4	0
C_5	1

Matrix multiplication

```
Matrix-multiply(A[1..n][1..n], B[1..n][1..n])        cost    frequency

for i $\leftarrow$ 1 to n                               $c_1$     $n+1$
    for j $\leftarrow$ 1 to i                           $c_2$     $\sum_{i=1}^{n}(i+1)$
        $C_{ij} \leftarrow 0$                           $c_3$     $\sum_{i=1}^{n}\sum_{j=1}^{i}1$
        for k $\leftarrow$ j to i                       $c_4$     $\sum_{i=1}^{n}\sum_{j=1}^{i}(i-j+2)$
            $C_{ij} \leftarrow C_{ij} + A_{ik} * B_{kj}$  $c_5$     $\sum_{i=1}^{n}\sum_{j=1}^{i}\sum_{k=j}^{i}1$
return C                                               $c_6$     $1$
```

Order of Growth

- The number of instructions (or total execution time) is a function of *n*, the number of elements
- As *n* gets bigger, so does the number of instructions required
- Juris Hartmanis (an old CS prof at Cornell) and Ralph Stearns noticed this back in the 1960s and decided to try to classify algorithms based on what kinds of function of *n* they were
- Some problems may be defined by multiple parameters
  - Matrix multiplication of *l*-by-*m* matrix and *m*-by-*n* matrix to get a *l*-by-*n* matrix has three parameters, *l*, *m*, and *n*

What Matters in Algorithms

- The function for the number of instructions executed!
- Getting a "better" algorithm usually means finding a way to solve the problem where the function evaluates to a smaller value
- The actual function is usually unwieldy
- Simplify the analysis using asymptotic notation

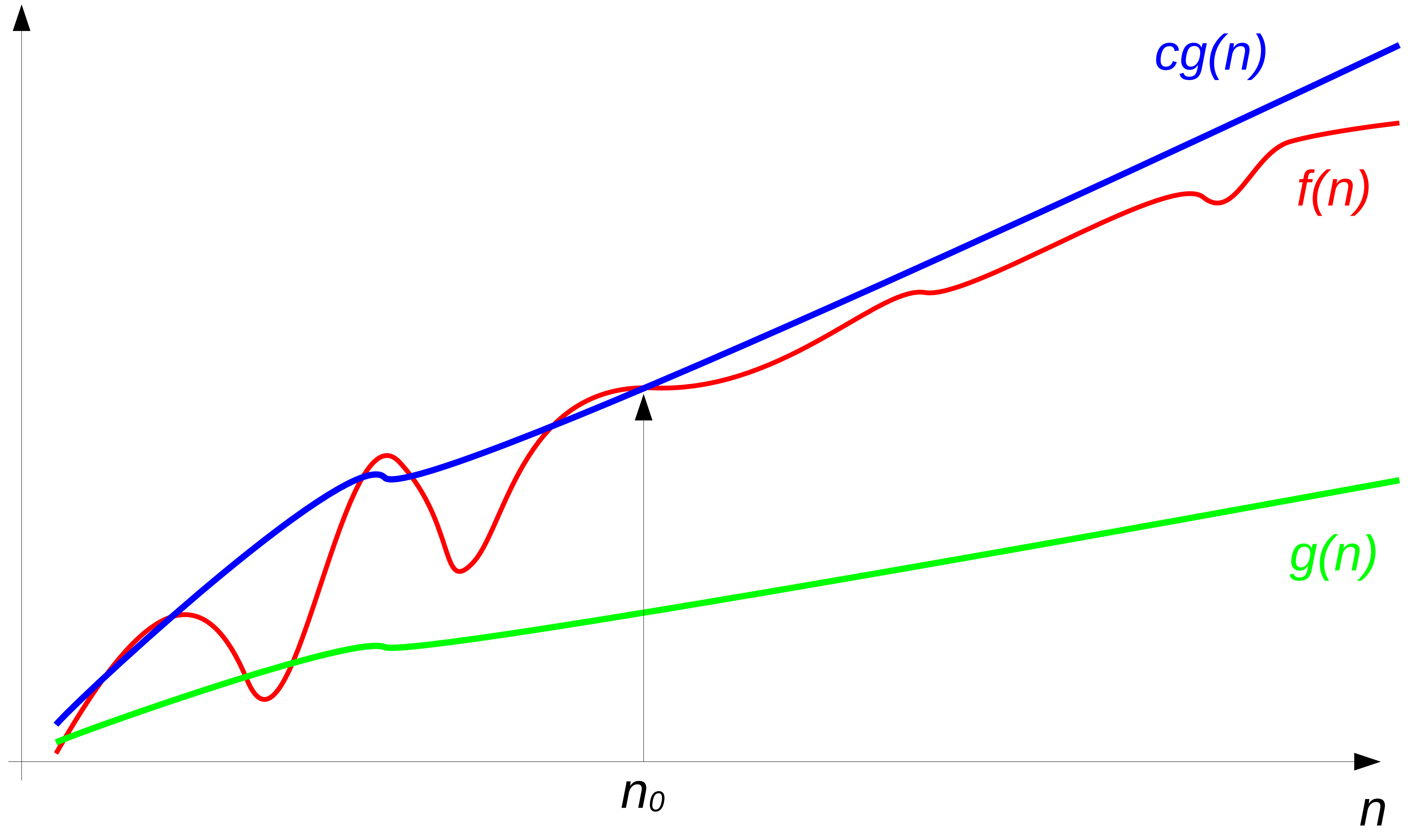
Asymptotic Notation: *O*()

Def: $f(n) = O(g(n))$ ("*f is of order g*") iff (if and only if):

- $\exists$ (there exist) positive constants c, n_0: $\forall$ (for all) $n \geq n_0$: $f(n) \leq cg(n)$
- Can find constants c and n_0 such that $f(n) \leq cg(n)$ $\forall$ $n \geq n_0$
- Apart from some constant factor, $g(n)$ will be larger than $f(n)$ for all "large" n
- $cg(n)$ gives an upper bound of $f(n)$ for large n
- *f grows no faster than g*

$O()$ notes:

- To prove $f(n) = O(g(n))$, we can choose to find c and n_0
- Actually we don't care about the constant c or small n's
- Alternatively, $\lim_{n \to \infty} f(n)/g(n) \neq \infty$
- Usually $g(n)$ is a simpler function than $f(n)$
- Examples of $g(n)$: 1, n, $n \log n$, n^2, 2^n
- Attempt to find the simplest and "smallest" $g(n)$ for $f(n)$


$cg(n)$
$f(n)$
$g(n)$
n_0
n

O() examples:

- 100 = ???
  - O(n): 100 ≤ cn for c = 100 and n_0 = 1
  - O(1): 100 ≤ c for c = 101 (for any n)
- 5n – 14 + 3/n = ???
  - O(n^2): $5n - 14 + 3/n \leq 5n + 14 + 3$ for $n \geq 1$

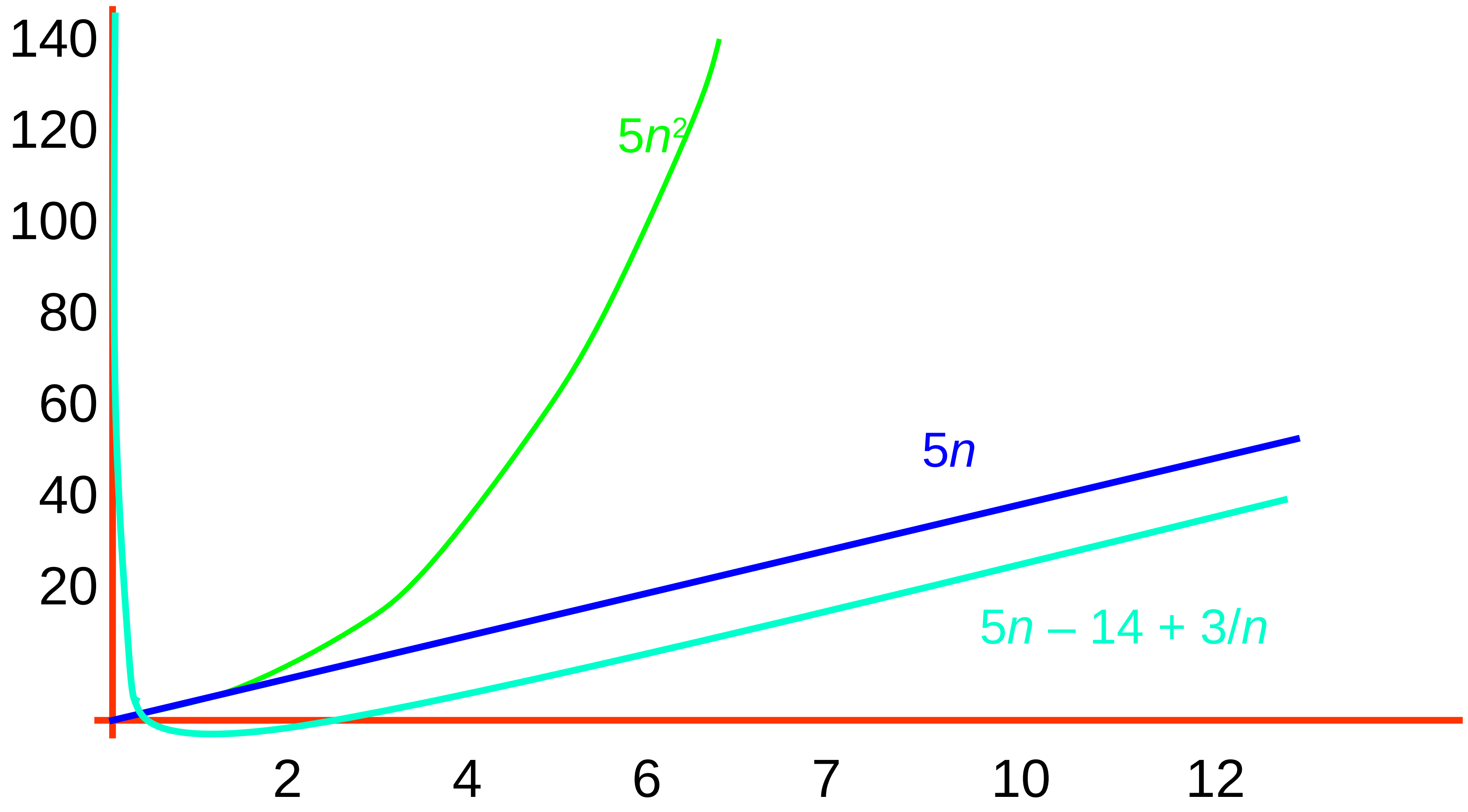
    $5n + 17 \leq 5n^2$ for $n \geq 3$

    thus, c = 5 and n_0 = 3
  - O(n): $|5n - 14 + 3/n| \leq 5n - 14 + 1$ for $n \geq 3$

    $5n - 13 \leq 5n$ for $n \geq 3$

    thus, c = 5 and n_0 = 3

$O()$ examples: graph



$O()$ properties:

- $f(n) = O(f(n))$
- If $f(n) = O(g(n))$ and $g(n) = O(h(n))$,
  then $f(n) = O(h(n))$
  - i.e., if f is of order g and g is of order h then f is also of order h
  - Therefore, there is a hierarchy of increasingly "larger" functions
    $f(n)$, $g(n)$, $h(n)$

$O()$ more properties:

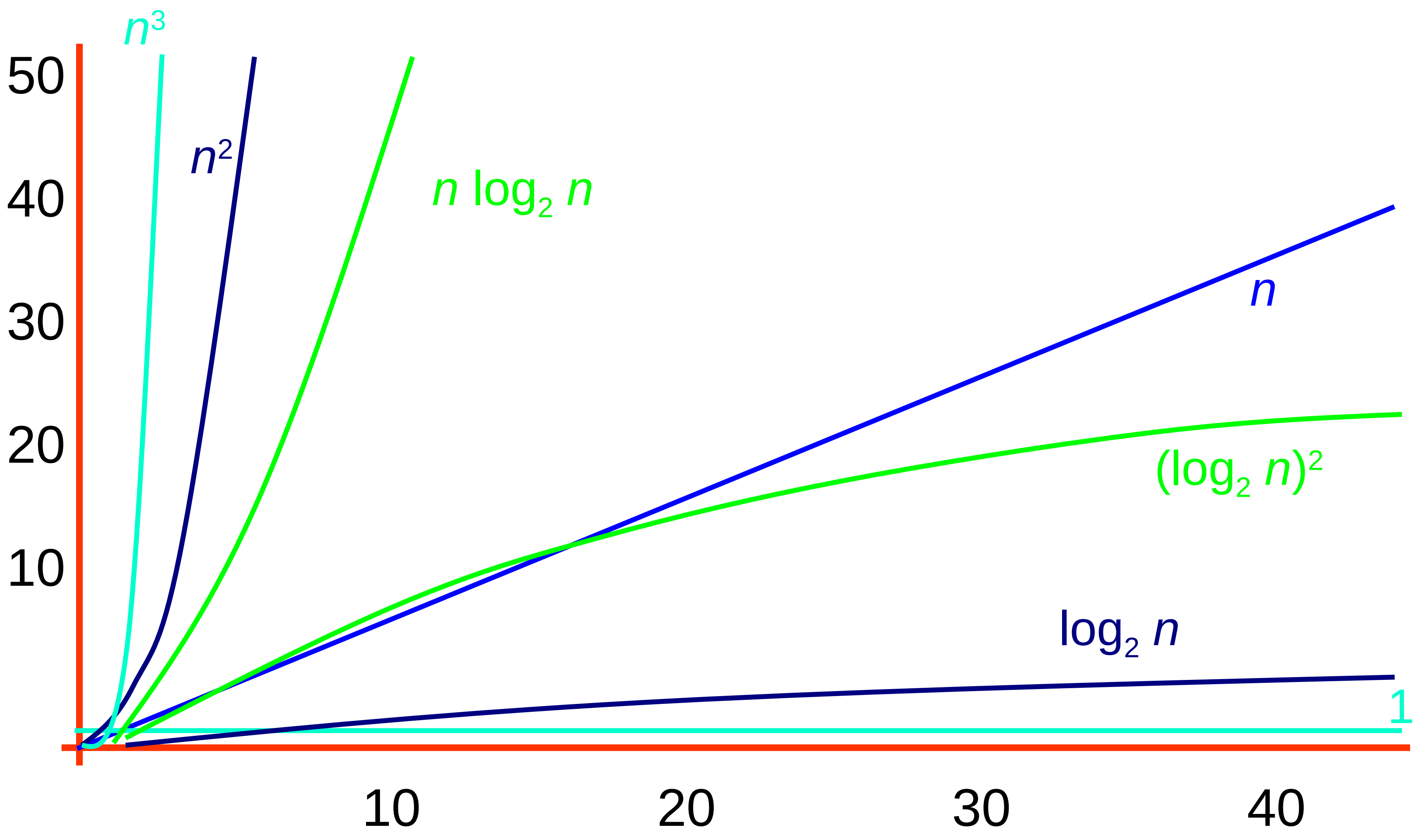
- $f(n) = O(f(n))$
- If $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then $f(n) + g(n) = O(h(n))$
  - “smaller” functions in sums do not matter for order
  - e.g., $5n + 100 = O(n) + O(1) = O(n)$
- If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$, then $f_1(n)\, f_2(n) = O(g_1(n)\, g_2(n))$
  - Product of functions is of the order of the product of orders of the functions

$O()$ Hierarchy of Functions:

Polynomial order (j, k, l, m are constants)

- $k = O(1)$
- $k = O(\log_m n)$
- $k \log_m n = O(\log_b n) = O(\log n)$ (b is irrelevant)
- $k (\log_m n)^j = O(n)$
- $k\, n = O(n)$
- $k\, n = O(n \log n)$
- $k\, n \log_m n = O(n \log n)$
- $k\, n \log n = O(n^2)$
- $k\, n^j = O(n^l)$ for all $l \geq j$

O() functions: graph



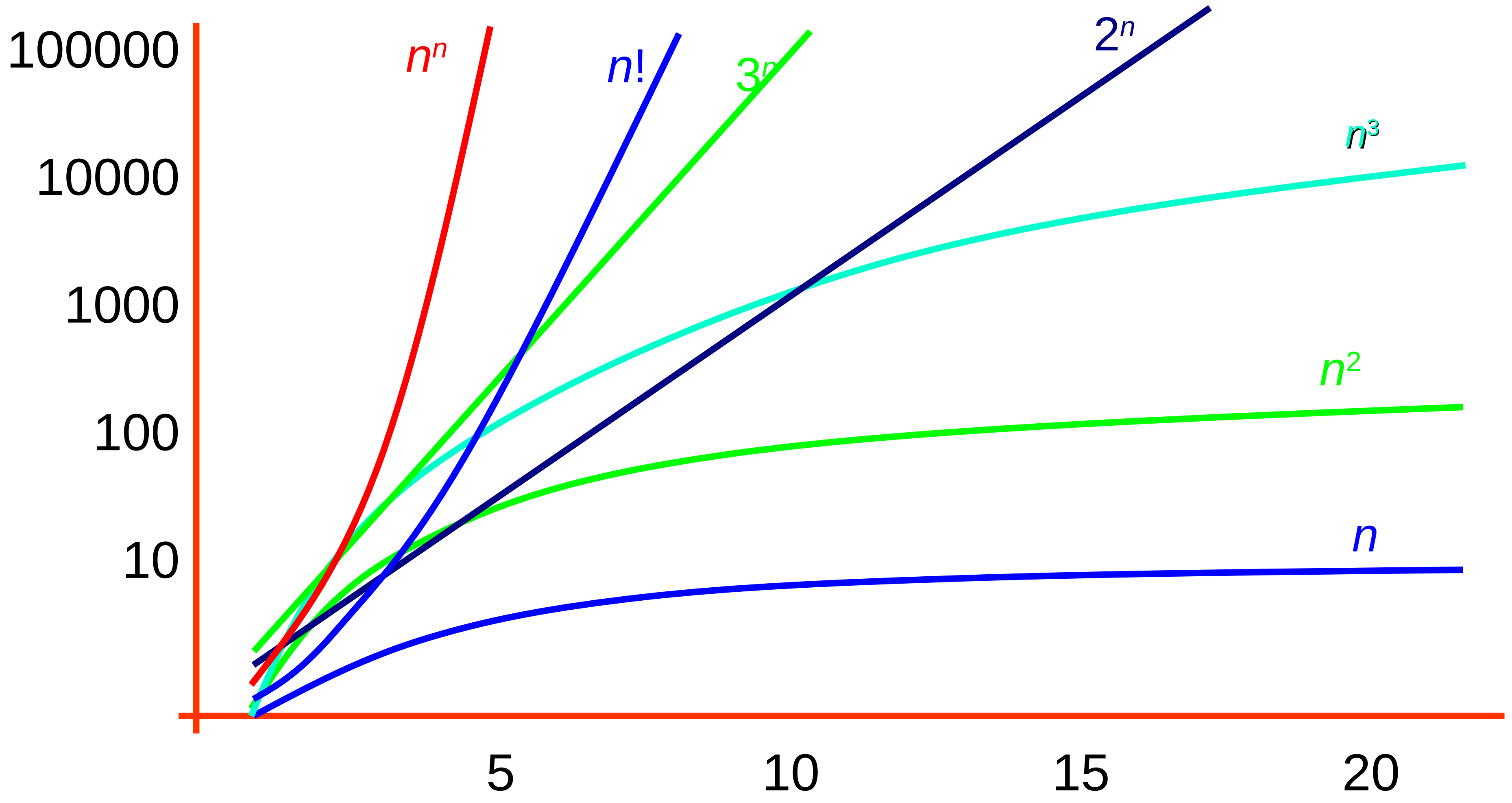
$O()$ Hierarchy of Functions:

Non-Polynomial order (*d, j, k, l* are constants)

- $k\,n^j = O(d^n)$ for any $d > 1$
- $k\,d^n = O(d^n)$ for $d > 1$
- $k\,d_1^n = O(d_2^n)$ for $d_1 > 1$ and $d_1 < d_2$
- $k\,d^n = O(n!)$
- $k\,n! = O(n!)$
- $k\,n! = O(n^n)$
- $k\,n^n = O(n^n)$

Functions that are of order $O(d^n)$ but not of order $O(n^l)$ are said to be of exponential order

$O()$ examples: graph



Asymptotic Notation: *O*() (revisited)

Def: $f(n) = O(g(n))$ ("*f is of order g*") iff (if and only if):

- $\exists$ (there exist) positive constants c, n_0: $\forall$ (for all) $n \geq n_0$: $f(n) \leq cg(n)$
- Can find constants c and n_0 such that $f(n) \leq cg(n)$ $\forall$ $n \geq n_0$
- Apart from some constant factor, $g(n)$ will be larger than $f(n)$ for all "large" n
- $cg(n)$ gives an upper bound of $f(n)$ for large n
- *f grows no faster than g*

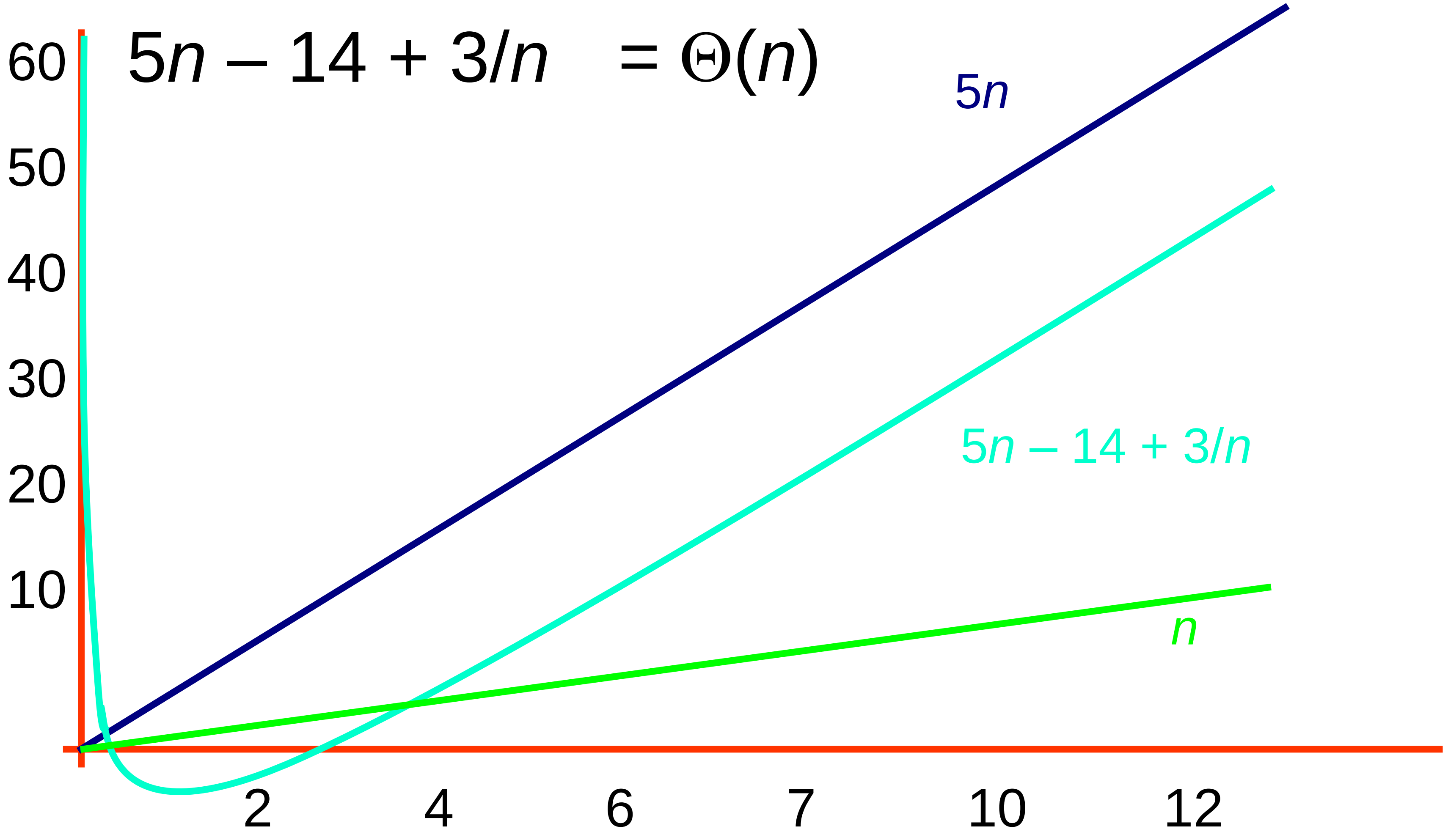
Asymptotic Notation: Big-Ω()

- Def: $f(n) = \Omega(g(n))$ iff:
  - $\exists$ positive constants c, n_0: $\forall\ n \geq n_0 : f(n) \geq cg(n)$
  - c and n_0 exist, such that $f(n) \geq cg(n)\ \forall\ n \geq n_0$
  - Just like O() except that $cg(n)$ is a lower bound
  - f grows at least as fast as g
  - $\lim_{n \to \infty} f(n)/g(n) \neq 0$
- Fact: f is $\Omega(g)$ iff g is $O(f)$
  - i.e., *f* is “larger” than *g* if *g* is “smaller” than *f*

Asymptotic Notation: Big-Θ()

- Def: $f(n) = \Theta(g(n))$ iff:
  - $\exists$ Constants c_1, c_2, n_0: $\forall\, n \geq n_0 : c_1 g(n) \leq f(n) \leq c_2 g(n)$
  - i.e., *g* is “smaller” and “larger” than *f*
  - i.e., *g* and *f* show the same asymptotic behavior, *f* and *g* grow at the same rate
- Fact: *f* is $\Theta(g)$ iff *f* is $O(g)$ and *f* is $\Omega(g)$
- Fact: *f* is $\Theta(g)$ iff *f* is $O(g)$ and *g* is $O(f)$
- Fact: *f* is $\Theta(g)$ implies *f* is $O(g)$
- Fact: *f* is $\Theta(g)$ implies *f* is $\Omega(g)$
- Fact: *f* is $\Theta(g)$, then $\lim_{n\to\infty} f(n) / g(n)$ exists and is greater than 0

Θ() example: graph



Asymptotic Notation: Little-*o*()

- Def: $f(n)$ *is* $o(g)$ if $\lim_{n\to\infty} f(n) / g(n) = 0$
- Def: $\forall$ positive constant c, $\exists$ positive constant n_0, $\forall\ n \geq n_0 : f(n) \leq cg(n)$
- f grows slower than g
- Fact: f is $o(g)$ implies f is $O(g)$
  - i.e., if f is $o(g)$ then f is $O(g)$, but if f is $O(g)$, f might not be $o(g)$
  - Example for the latter case: $5n^2$ is $O(n^2)$ but $5n^2$ is not $o(n^2)$. In fact, $5n^2$ is not only $O(n^2)$ but also $\Theta(n^2)$.
- Fact: if f is $O(g)$ but not $\Theta(g)$, it is $o(g)$

Little-ω: f is $\omega(g)$ if g is $o(f)$, $\lim_{n\to\infty} f(n) / g(n) = \infty$

  - Similar relationship as between $O()$ and $\Omega()$

Asymptotic Notation: *O*() (revisited)

Def: $f(n) = O(g(n))$ ("*f is of order g*") iff (if and only if):

- $\exists$ (there exist) positive constants c, n_0: $\forall$ (for all) $n \geq n_0$: $f(n) \leq cg(n)$
- Can find constants c and n_0 such that $f(n) \leq cg(n)$ $\forall$ $n \geq n_0$
- Apart from some constant factor, $g(n)$ will be larger than $f(n)$ for all "large" n
- $cg(n)$ gives an upper bound of $f(n)$ for large n
- *f grows no faster than g*

Asymptotic Notation: comparison

Analogy between asymptotic comparison of two functions *f*(*n*) and *g*(*n*) and the comparison of two real numbers *a* and *b*:

- $f(n) = O(g(n)) \quad \approx \quad a \le b$
- $f(n) = \Omega(g(n)) \quad \approx \quad a \ge b$
- $f(n) = \Theta(g(n)) \quad \approx \quad a = b$
- $f(n) = o(g(n)) \quad \approx \quad a < b$
- $f(n) = \omega(g(n)) \quad \approx \quad a > b$

Note, while real number can be compared, not all functions are asymptotically comparable (for example, sin x vs. cos x)

Back to Tokyo …

- Scenario 1: A flat rate per ride
  - $O(n\log n)$ to find the largest possible subset of requests that did not overlap in time
- Scenario 2: Customers bid for services
  - $O(n\log n)$ to find the set of non-overlapping requests that maximized income
- Scenario 3: Customers bid for a set of time-period requests
  - Have to try all 2^n possible subsets of requests
  - As a competent consultant, you should not promise to find the best solution
  - your program should have tried to find a good but not optimal solution