Complexity Analysis

How fast is our code?

Nattee Niparnan

Overview

- Introducing a measurement of efficiency of a program
- Why we need measurement
- How we measure
    - How to analyze our code to get a measurement
- What is measurement
    - Asymptotic Notation

Key Idea

- We need a useful way to describe efficiency of our code
  - Useful = able to be easily use to predict how much resource (time / memory) that our program will need
  - Useful = not overly complex in analysis
  - Need to balance between usefulness and complexity
- Ultimately, we introduce a class of efficiency that says how our code use resource with respect to size of data
  - Focus on growth of resource usage

Preview

```
int find_max(vector<int> v) {
  int m = v[0];
  for (size_t i = 0;i < v.size();i++)
    if (v[i] > m)
      m = v[i];
  return m;
}
```

- This code takes time directly proportional to the size of the data
  - Size N takes time T
  - Size 5N should take time 5T

```
int count_pair_sum(vector<int> v,int k) {
  int count = 0;
  for (size_t i = 0;i < v.size();i++)
    for (size_t j = 0;j < v.size();j++)
      if (i != j && v[i]  + v[j] == k)
        count++;
  return count/2;
}
```

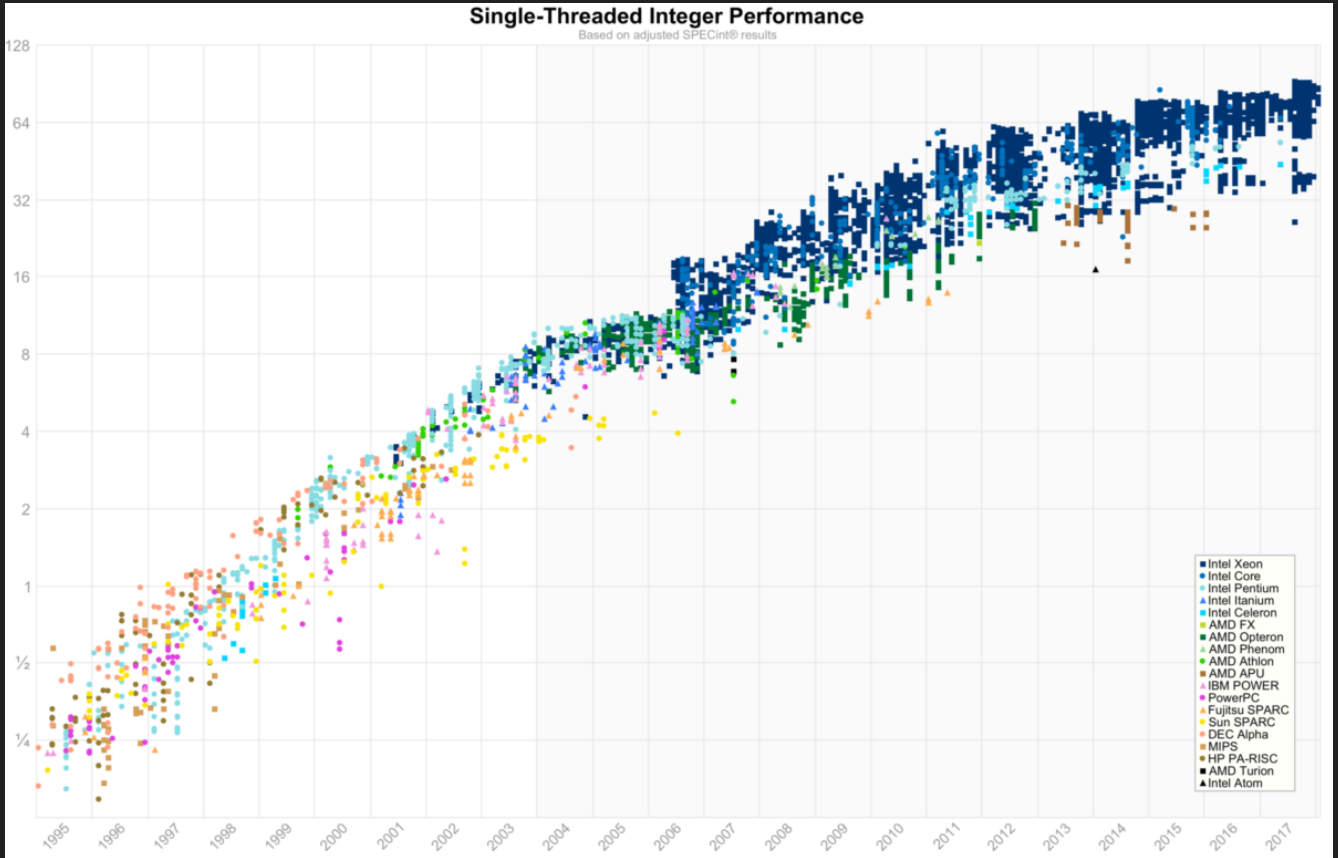
- This code takes time directly proportional to the square of the size of the data
  - Size N takes time T
  - Size 5N should take time 25T

Why don’t use real world clock?

- Ultimately, we want to know how long our program takes to do each operation
  - Use in design, How much resource we need
  - Help us choose appropriate data structure
- Real world clock measurement is possible but has many drawback
  - System dependency
  - Too complex (we have to build our system)
  - Too Specific

Dependency of System

In summary, we want a measurement that is system independent



https://preshing.com/20120208/a-look-back-at-single-threaded-cpu-performance/

- Same program in different system = different time
  - CPU
  - RAM
  - Compiler
  - Operating parameter
    - Heat? Other running program?

How to measure

This code use $4 + 4n + 5n^2$ instruction

- Counting instruction
  - Depend on code only
- Dependency on size of data
  - Focus on large data

$$\begin{aligned}\text{Total} &= \overset{(a)}{1} + \overset{(b)}{2+n+n} + \overset{(c)}{(2+n+n)*n} + \overset{(d)}{2n^2} + \overset{(e)}{n^2} + \overset{(f)}{1} \\ &= 4 + 4n + 5n^2\end{aligned}$$

```
int count_pair_sum(vector<int> v,int k) {
  int count = 0;
  for (size_t i = 0; i < v.size(); i++)
    for (size_t j = 0; j < v.size(); j++)
      if (i != j && v[i] + v[j] == k)
        count++;
  return count/2;
}
```

(a) 1
(b) 2 + n + n (n = v.size())
(c) (2 + n + n) * n
(d) 2 * n * n
(e) 1 * ???? (<= (d))
(f) 1

Time per instruction

- In reality, different CPU instructions use different time
- Same instruction but different CPU also use different number of cycle
- However, we just ignore it
  - For now

https://www.agner.org/optimize/instruction_tables.pdf

AMD Ryzen 3000 MUL

MUL, IMUL	r32/m32	2	3	1	
MUL, IMUL	r64/m64	2	3	1	
IMUL	r,r	1	3	1	
IMUL	r,m	1		1	

AMD Ryzen 3000 DIV

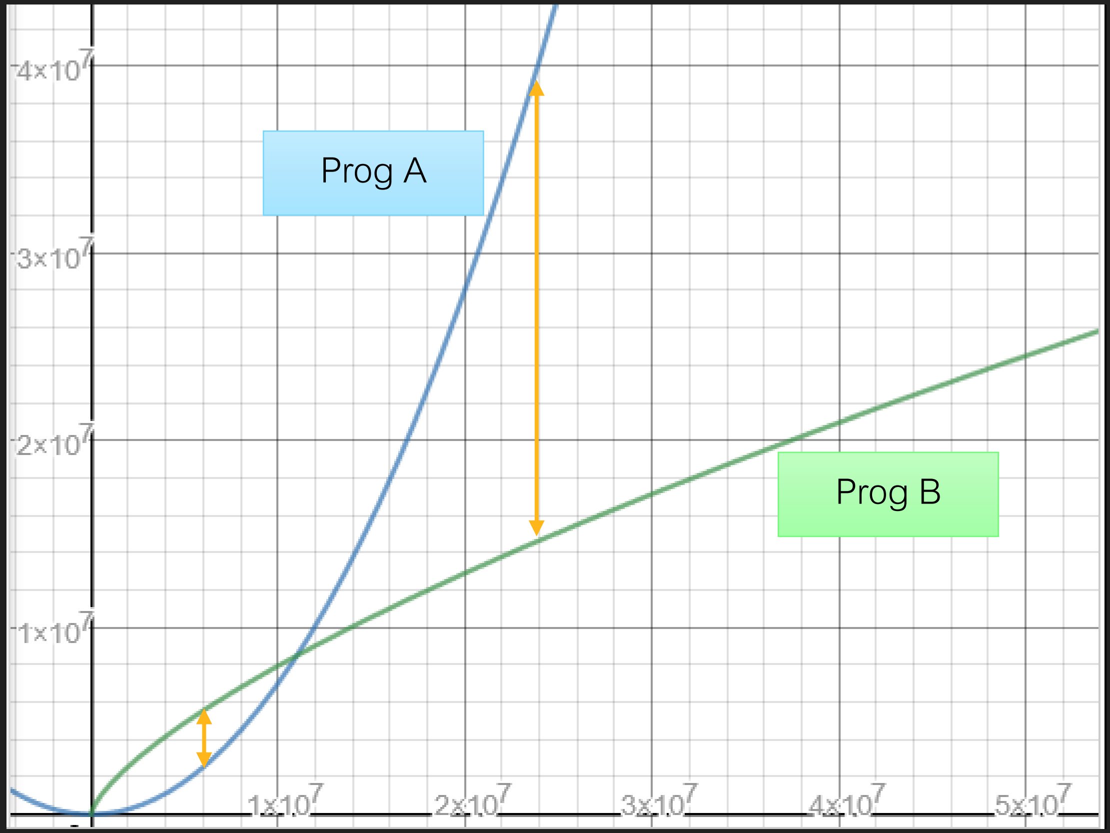
DIV	r8/m8	1	12-15	12-15	
DIV	r16/m16	2	13-20	13-20	
DIV	r32/m32	2	13-28	13-28	
DIV	r64/m64	2	13-44	13-44	

Intel Coffee Lake DIV

DIV	r8	10	10	p0 p1 p5 p6	23	6
DIV	r16	10	10	p0 p1 p5 p6	23	6
DIV	r32	10	10	p0 p1 p5 p6	26	6
DIV	r64	36	36	p0 p1 p5 p6	35-88	21-83

Focusing on large data

- In many cases, the size of the data we are working with will affect the time our code use
- Large data usually mean longer time
- What matter is when the data is large



Growth rate simplifies analysis

```
int count_pair_sum(vector<int> v,int k) {
  int count = 0;
  for (size_t i = 0; i < v.size(); i++)
    for (size_t j = i+1; j < v.size(); j++)
      if (v[i] + v[j] == k)
        count++;
  return count;
}
```

```
i=0: 1 + n-0 + n-1
i=1: 1 + n-1 + n-2
i=2: 1 + n-2 + n-3
.
.
i=n-1: 1 + 1 + 0
```

sum = $\Sigma(1+2n-2i-1) = n^2 + n$

$$\text{Total} = \underbrace{1}_{(a)} + \underbrace{2+n+n}_{(b)} + \underbrace{n^2+n}_{(c)} + \underbrace{n^2}_{(d)} + \underbrace{n^2}_{(e)} + \underbrace{1}_{(f)}$$

$$= 4 + 3n + 3n^2$$

This code uses $4 + 3n + 3n^2$ instruction

Small Detail

n	$5n^2+4n+4$	$3n^2+3n+4$	n^2
10	544	334	100
20	2084	1264	400
40	8164	4924	1600
80	32324	19444	6400
160	128644	77284	25600
320	513284	308164	102400
640	2050564	1230724	409600
1280	8197124	4919044	1638400
2560	32778244	19668484	6553600
5120	1.31E+08	78658564	26214400
10240	5.24E+08	3.15E+08	1.05E+08
20480	2.1E+09	1.26E+09	4.19E+08

When counting instruction, it is usually OK to focus on most executed line

Measurement by Growth Rate

What?

- Growth rate = how much resource usage growth with respect to change of input
  - Resource usage = number of instruction used
  - Input = size of data
- Emphasizes long term trend

Why?

- System independent
  - The result can be used to predict behavior on any system
- Focus on change of resource usage with respect to size of input
- Can disregard small detail
  - Simple to calculate
  - Applicable in real world

Asymptotic Notation

Classification of growth rate

Overview

- Formally, it is a set of function having the growth rate related to something
- The definition focus on growth of the function while disregard small detail
- Also provide some workaround on dependency of value of input

What is?

- A notation written as O(f(n))
  - O can be one of $O, \Theta, \Omega, o, \omega$
  - f(n) is some expression
- Example $O(n)$ or $\Theta(n^2+3)$ or $\omega(n^2 \log(n))$
- Usage
  - "This code is O(n)" (read as Big-Oh of n)
  - "This function takes time in $\Theta(\log(n))$" (read as Big-theta of log n)
  - "Time complexity of this program is O(n^2)"

Meaning

- "A is Θ(f(x))" means the growth rate of A is equal to the growth rate of f(x)
- "B is O(f(x))" means the growth rate of B is less than or equal to the growth rate of f(x)
- For Ω, o, ω, (Big-Omega, little-oh, little-omega), we won't use it for now but the meaning is similar (which are more than or equal, less than, more than, respectively)
- Convention, we usually use N for the size of the data

Usage

- Let $f(n)$ be the number of instruction need by code A when the size of data is n
- We will calculate asymptotic notation that $f(n)$ is a member of
  - Find $g(n)$ such that $f(n)$ is $O(g(n))$ or $\Theta(g(n))$ (or $\Omega(g(n))$ or ….)
- Let's say we have analyzed that $f(n)$ is $O(g(n))$
  - We now understand that the growth rate of instruction required by code A grows slower or the same as how $g(n)$ grow

Comparing growth rate of f(n) and g(n)

- The relation of growth rate of f(n) and g(n) depends on the value of f(n)/g(n) when n approach infinity

$$\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right)$$

0	f(n) grows slower than g(n)
c	f(n) grows similar than g(n)
∞	f(n) grows faster than g(n)

O(g(n)) = set of all functions that does not grow faster than g(n)

Θ(g(n)) = set of all functions that grows similar to g(n)

Example

- $f(n) = 4 + 3n + 4n^2$
- $g(n) = n^2$

vanish as n approach infinity

$$\lim_{n\to\infty}\left(\frac{4n^2+3n+4}{n^2}\right) = \lim_{n\to\infty}\left(4 + \boxed{3/n + 4/n^2}\right) = 4$$

- Hence f(n) grows similar to g(n)
  - Therefore $f(n) = \Theta(n^2)$

Another Example

- $f(n) = 0.00005\, n^2$
- $g(n) = 100000\, n$

$$\lim_{n\to\infty}\left(\frac{0.00005n^2}{100000n}\right) = \lim_{n\to\infty}(10^{-10}n) = \infty$$

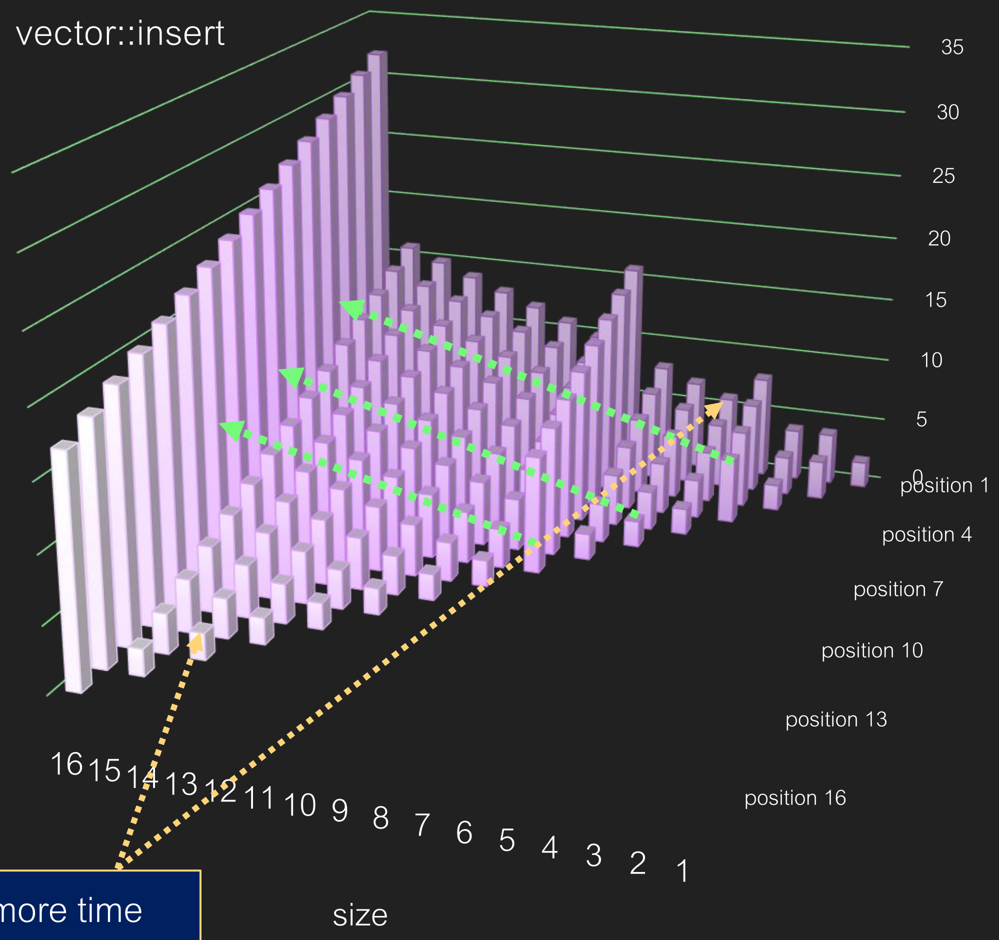
- Hence f(n) grows faster than g(n) (also means g(n) grows slower than f(n))
  - Therefore $g(n) = O(0.00005n^2)$
  - and $f(n) = \Omega(100000n)$

Big-O notation

- Formally, O(g(n)) is a set of all functions that grows either the same or slower than g(n)
- f(n) is O(g(n)) means $\lim(\frac{f(n)}{g(n)})$ is either 0 or a constant
  - Which implies that f(n) growth rate does not exceed that of g(n)

Dependency on the value of input

- Consider vector::insert(iterator it, T value)
- The time it takes depends on both the size of the vector and the value of it
    - Larger size --> more time
    - Closer to end() --> less time



Insert at begin() of size 4 use more time than insert at end of size 13

Big-O describes upper bound

- vector::insert is O(n)
  - Its growth rate does not exceed n
  - There are case that it maybe grow less than n (insert at end)
- This is very useful in real world
  - Knowing maximum load
  - Not overly complex in analysis

Big-O Example

- Find is O(n)
  - At worse, it can’t find value and a and b points to begin() and end()
    - This case, find growth rate is n
  - At bets, it always find value at the first position (a)
    - This case, find growth as 1

```
bool find(iterator a, iterator b, T value) {
  while (a < b) {
    if (*a == value)
      return true;
    a++;
  }
  return false;
}
```

l'Hôpital's Rule

- Can help
- F(n) = log n
- g(n) = $n^{0.5}$

$$\lim_{n\to\infty}\frac{\log n}{\sqrt{n}} = \lim_{n\to\infty}\frac{\ln(n)}{\ln(10)\sqrt{n}}$$
$$= \frac{1}{\ln(10)}\lim_{n\to\infty}\frac{\ln(n)}{\sqrt{n}}$$
$$= \frac{1}{\ln(10)}\lim_{n\to\infty}\frac{1/n}{1/(2\sqrt{n})}$$
$$= \frac{1}{\ln(10)}\lim_{n\to\infty}\frac{2}{\sqrt{n}} = 0$$

$$\lim_{n\to\infty}\frac{f(x)}{g(x)} = \lim_{n\to\infty}\frac{f'(x)}{g'(x)}$$

f(x) must be diffable

g(x) must be diffable

g(x) non-zero

lim(f'/g') must exists

log n grow slower than square root n

use l'Hopital

$$\frac{1/n}{1/(2\sqrt{n})} = \frac{2}{\sqrt{n}}$$

Exercise

- $f(n) = (\log n)^c$
- $g(n) = n^k$
- We know that $c > 0$, $k > 0$
- Does $f(n)$ grow slower than $g(n)$?

Big-Theta is tight bound

- std::count always go through entire array
- Regardless of the value in the array, it always perform

```
if (*first == value)
```

```
size_t count(iterator first, iterator last, const T& value) {
  size_t ret = 0;                       (a) 1
  for (; first != last; ++first) {      (b) n + n
    if (*first == value)                (c) 1 * n
      ret++;                            (d) 1 * ???? ( <= (d) )
  }
  return ret;                           (e) 1
}
```

```
          (a)  (b)    (c)    (d)   (e)
Total = 1 + n+n + 1*n + 1*n + 1
      = 2 + 4n
```

More Example

Observation: multiplicative and addition constants in f(n) can usually be ignored, since it will be disregard by lim. Degree cannot.

Θ(1)	**Θ**(n)	**Θ**(n^2)
f(n) = 5	f(n) = n	f(n) = n*n
f(n) = 0	f(n) = n + 3	f(n) = C(n,2) = n(n-1)/2
f(n) = c	f(n) = n/1200 + 86	f(n) = $400n^2$ + an + b
	f(n) = 40000000n	

Observation: O(g(n)) always include **Θ**(g(n)) by definition

O(1)

O(n)

O(n^2

)

More Example

Θ(n)
f(n) = n
f(n) = n + 3
f(n) = n/1200 + 86
f(n) = 40000000n

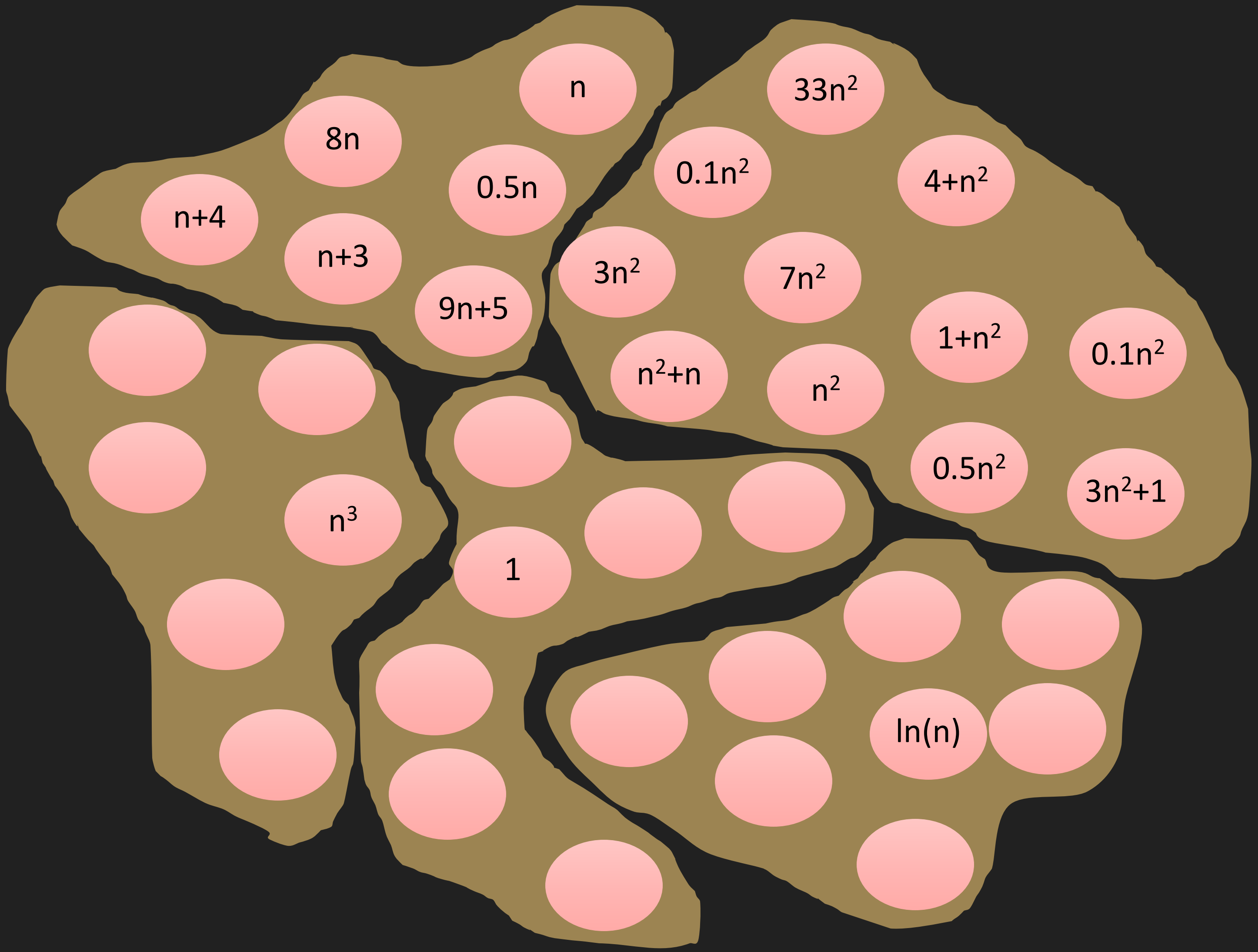
f(n) = n + 3 is O(n)

f(n) = n is O(4000000n)

O(n) is O(400000n)

Which is also O(0.585n + 3)

Classification



Well known growth rate class

grow slow

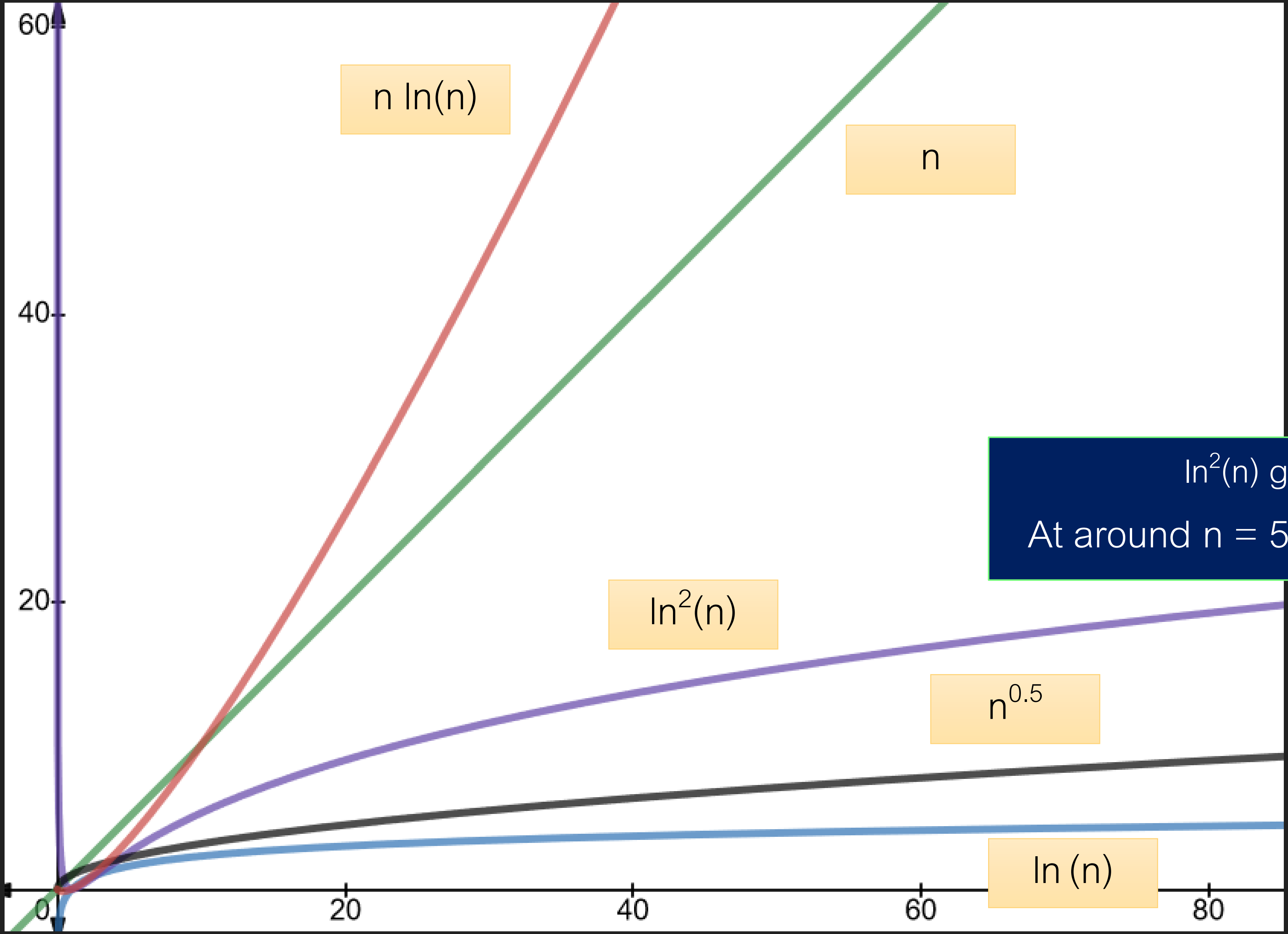
Grow fast

$\Theta(1)$	Constant
$\Theta(\log(n))$	Logarithm
$\Theta(\log^c(n))$, $c >= 1$	Polylogarithm
$\Theta(n^a)$, $0 < a < 1$	Sublinear
$\Theta(n)$	Linear
$\Theta(n \log(n))$	Linearithmic
$\Theta(n^2)$	Quadratic
$\Theta(n^c)$, $c >= 1$	Polynomial
$\Theta(c^n)$, $c > 1$	Exponential
$\Theta(n!)$	Factorial

Exercise:

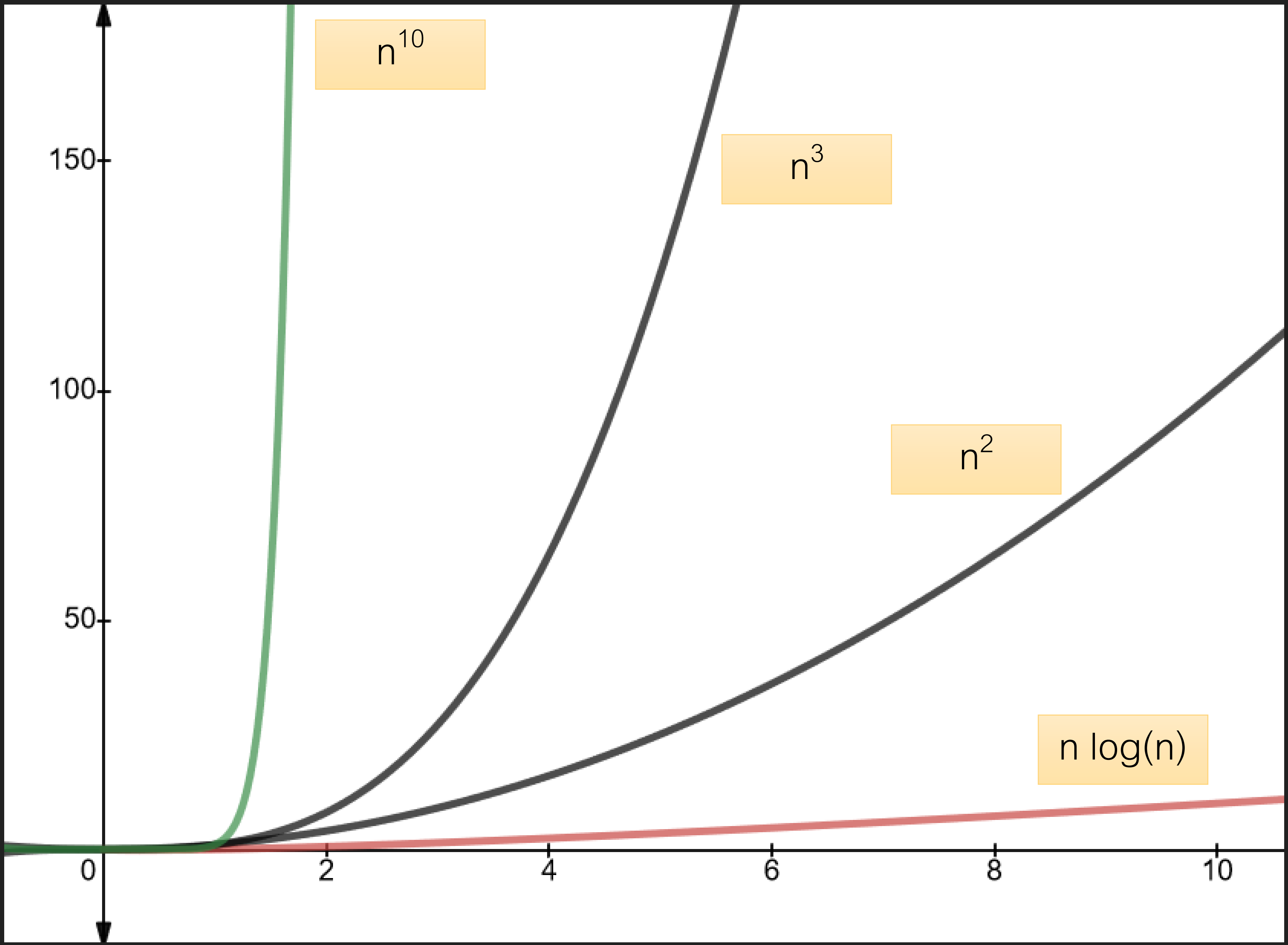
Try comparing following functions using $\lim \left(\frac{f(n)}{g(n)} \right)$

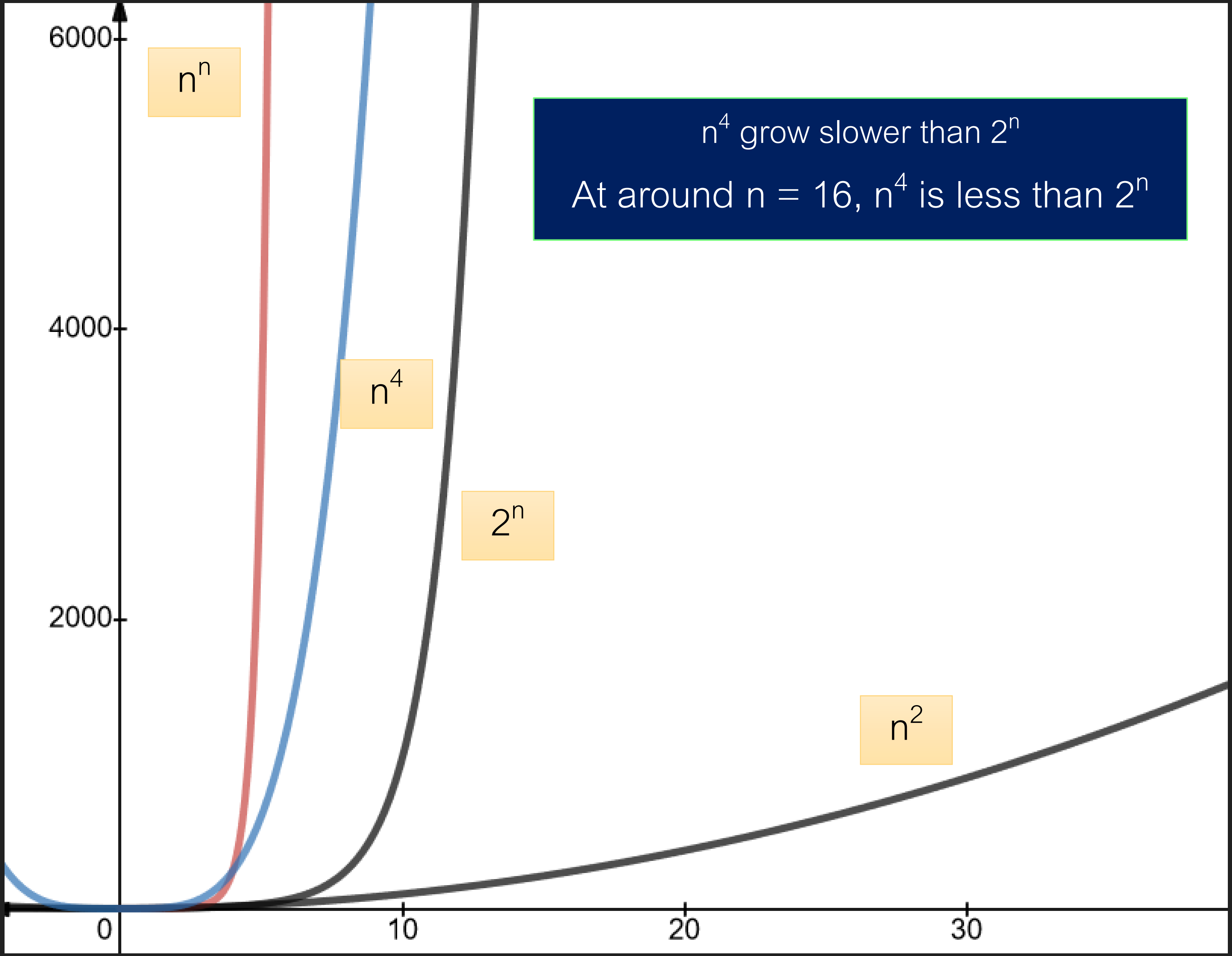
F(n)	G(n)	lim(F(n) / G(n))
$\log_3(n)$	$\log_8(n)$	
n^2	$\log_2(n)$	
2^n	n^4	
$\log_2(n)$	$\log_2(n^8)$	



$\ln^2(n)$ grow slower than $n^{0.5}$

At around n = 5600, $\ln^2(n)$ is less than $n^{0.5}$


n^10
n^3
n^2
n log(n)
150
100
50
0
2
4
6
8
10



6000
4000
2000
0
10
20
30
n^n
n^4
2^n
n^2
n^4 grow slower than 2^n
At around n = 16, n^4 is less than 2^n

Beware

- It is wrong to say that vector::insert is $\Theta(n)$
  - Because there is a case that it grows faster than N
- It is wrong to say that vector::push_back is $\Theta(n)$
  - Because there is a case that it grows slower than N
- It is ok to say that std::count is O(n)
  - Because, while it always grows as N, it does not grow faster than N
  - O is upper bound
  - But it is better to say that std::count is $\Theta(n)$

How to analyze using asymptotic notation

1: Write a code

2: Calculate the function F(n) that counts the number of instruction of the code when the data is of size n

Usually, just focus on most executed line

3: Find g(n) and a notation X such that f(n) is X(g(n))

It's either Big-O or Big-Theta

If there is a case that it can grow slower than G(n), use Big O

```
void erase(iterator it) {
  while((it+1)!=end()) {
    *it = *(it+1);
    it++;
  }
  mSize--;
}
```

Most executed line

$1 <= F(n) <= n$

vector::erase is O(n)

Another Example

- Let's analyze vector::push_back

```
iterator insert(iterator it,const T& element) {
  size_t pos = it - begin();
  ensureCapacity(mSize + 1);
  for(size_t i = mSize;i > pos;i--) {
    mData[i] = mData[i-1];                Most executed line (b)
  }
  mData[pos] = element;
  mSize++;
  return begin()+pos;
}
void push_back(const T& element) {
  insert(end(),element);
}
```

```
void expand(size_t capacity) {
  T *arr = new T[capacity]();
  for (size_t i = 0;i < mSize;i++) {
    arr[i] = mData[i];                Most executed line (a)
  }
  delete [] mData;
  mData = arr;
  mCap = capacity;
}

void ensureCapacity(size_t capacity) {
  if (capacity > mCap) {
    size_t s = (capacity > 2 * mCap) ? capacity : 2 * mCap;
    expand(s);
  }
}
```

(a) Can be 0 to n

(b) Can also be 0 to n

Best case 0+0

Worst case n + n

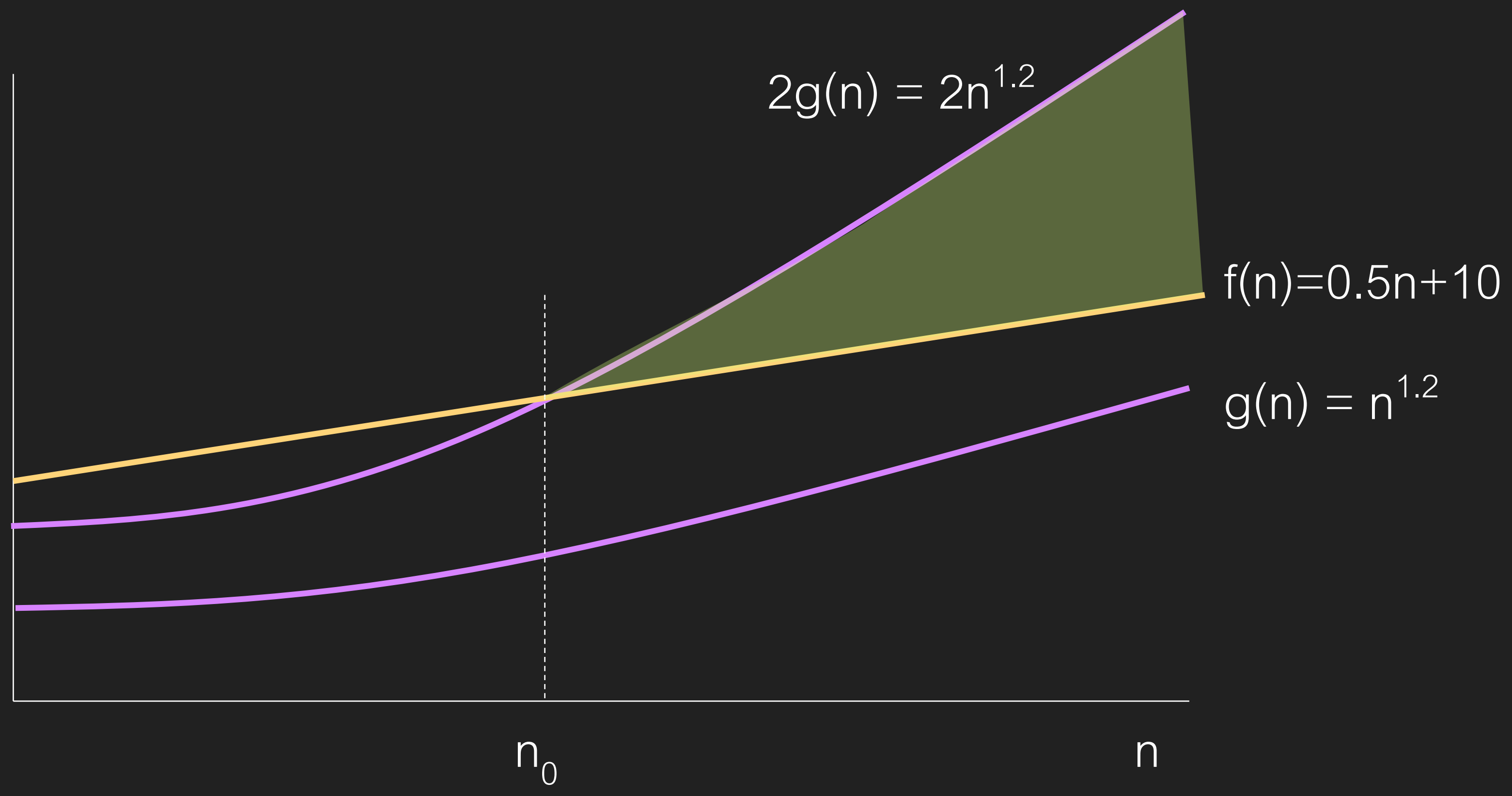
vector::push_back is O(n)

Another Definition for O and Θ

- Using set builder notation
- $O(g(n)) = \{ f(n) \mid \text{there exists } c > 0 \text{ and } n_0 >= 0$
  $\text{such that } f(n) <= cg(n) \text{ for } n >= n_0 \}$
- $\Theta(g(n)) = \{ f(n) \mid \text{there exists } c_1 > 0, c_2 > 0 \text{ and } n_0 >= 0$
  $\text{such that } c_1 g(n) <= f(n) <= c_2 g(n) \text{ for } n >= n_0 \}$
- The result is the same as definition using lim

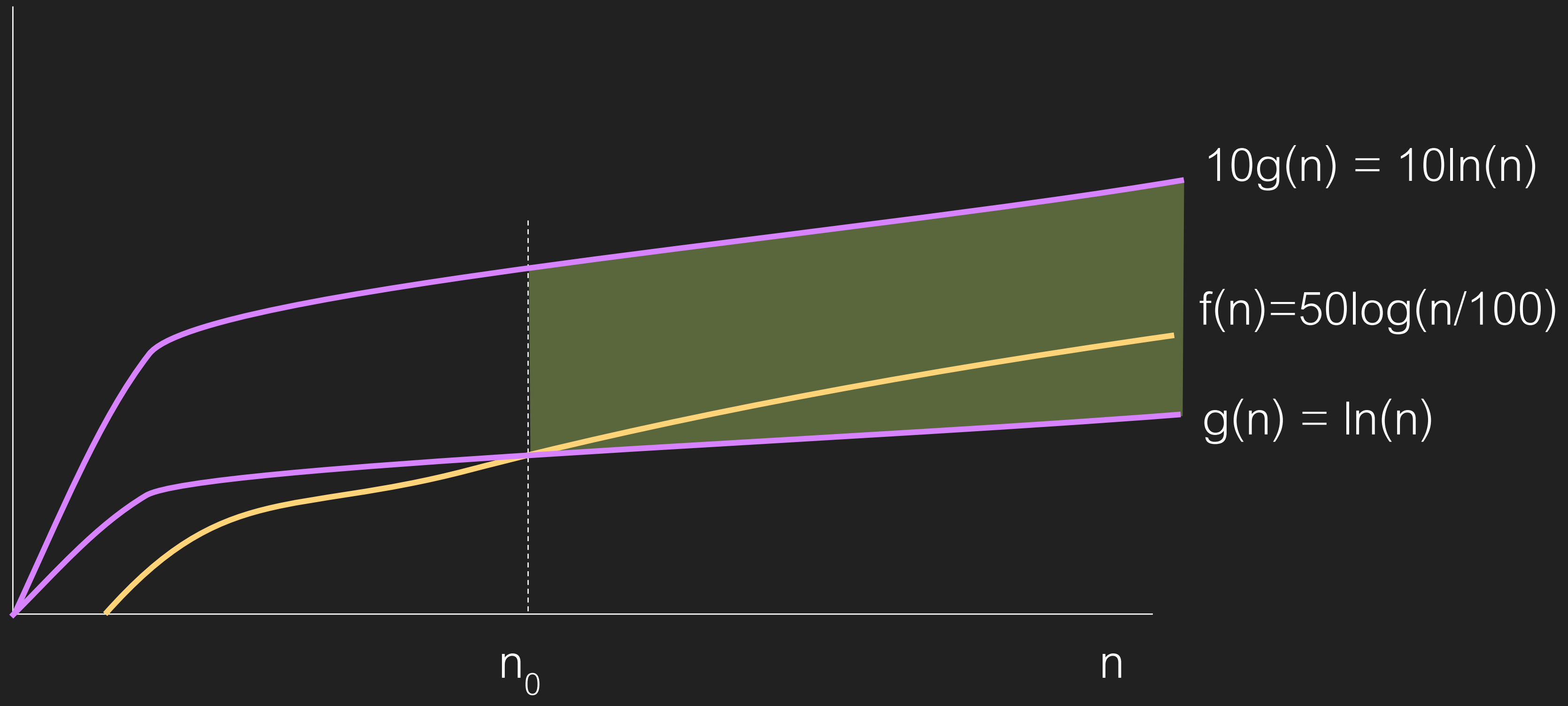
O as set builder notation

- $O(g(n)) = \{ f(n) \mid \text{there exists } c > 0 \text{ and } n_0 >= 0$
  $\text{such that } f(n) <= cg(n) \text{ for } n >= n_0 \}$



Θ as set builder notation

- $\Theta(g(n)) = \{ f(n) \mid \text{there exists } c_1 > 0, c_2 > 0 \text{ and } n_0 >= 0$
  $\text{such that } c_1 g(n) <= f(n) <= c_2 g(n) \text{ for } n >= n_0 \}$



Summary

- We use Asymptotic Notation to describe efficiency of a program
  - Measure instruction count instead of time
  - Focus on growth rate of instruction count
- Find most frequently executed line in the code and count it
- Maps to Big-Theta if we have tight bound
- Use Big-O if we have upper bound

More Example

Analyze test1

```
int test1(vector<int> v) {
  int sum = 0;
  for (auto &x : v)
    sum += x;                    Most executed line
  return sum;
}
```

$f(n) = n$

$\Theta(n)$	Good
$O(n)$	OK
$O(n^2)$	Bad, but not wrong
$\Theta(n^2)$	Wrong

Analyze test1

```
int test1(vector<int> v) {
  int sum = 0;
  for (int i = 0;i < v.size();i+= 2)
    sum += v[i];                    Most executed line
  return sum;
}
```

$f(n) = n/2$

$\Theta(n)$	Good
$O(n)$	OK
$O(n^2)$	Bad, but not wrong
$\Theta(n^2)$	Wrong

Analyze test2

```
int test2(vector<int> v) {
  int sum = 0;
  for (int i = 0;i < v.size();i++)
    for (int j = i+1;j < v.size();j++)
      sum += v[i] + v[j];
  return sum;
}
```

Most executed line

$$f(n) = \boxed{\sum_{i=0}^{n-1}} \boxed{\sum_{j=i+1}^{n-1}} 1$$

$$= \sum_{i=0}^{n-1} n - i - 1$$

$$= n(n-1)/2$$

$$= \frac{n^2}{2} + \frac{n}{2}$$

$\Theta(n^2)$	Good
$O(n^2)$	OK
$O(n^8)$	Bad, but not wrong
$\Theta(n)$, $\Theta(n^3)$ $O(n)$, $O(1)$	Wrong

For polynomial,

use the one that has highest degree

also discard constants

Analyze test3

```
int test1(vector<int> v) {
  int sum = 0;
  for (int i = 0;i < v.size();i+= 2)
    sum += v[i];
  return sum;
}
```

```
int test2(vector<int> v) {
  int sum = 0;
  for (int i = 0;i < v.size();i++)
    for (int j = i+1;j < v.size();j++)
      sum += v[i] + v[j];
  return sum;
}
```

```
int test3(vector<int> v) {
  test1(v);
  test2(v);
}
```

For summation of multiple terms,

use the one that grow fastest

$$f(n) = \Theta(n) + \Theta(n^2)$$

$\Theta(n^2)$	Good
$O(n^2)$	OK
$O(n^8)$	Bad, but not wrong
$\Theta(n)$, $\Theta(n^3)$ $O(n)$, $O(1)$	Wrong

Analyze test3

```
int test1(vector<int> v) {
  int sum = 0;
  for (int i = 0;i < v.size();i+= 2)
    sum += v[i];
  return sum;
}
```

```
int test2(vector<int> v) {
  int sum = 0;
  for (int i = 0;i < v.size();i++)
    for (int j = i+1;j < v.size();j++)
      sum += v[i] + v[j];
  return sum;
}
```

```
int test3(vector<int> v) {
  if (v.size() % 2 == 0)
    test1(v);
  else
    test2(v);
}
```

With conditional statement where it can be either f1() or f2()

Use O(max(f1(), f2()))

$$f(n) = \begin{cases} \Theta(n) & ; n \text{ is even} \\ \Theta(n^2) & ; n \text{ is odd} \end{cases}$$

$\Theta(n^2)$	Wrong
$O(n^2)$	Good
$O(n^8)$	Bad, but not wrong
$\Theta(n)$, $\Theta(n^3)$, $O(n)$, $O(1)$	Wrong

Analyze test4

```
int test4(int n) {
  int sum = 0;
  while (n > 0) {
    sum += 1;          Most executed line
    n = n / 2;
  }
}
```

In each loop, n reduce by half

1: n

2: n/2

3: n/2/2

4: n/2/2/2 ….

$f(n) = \log_2(n)$

$\Theta(\lg n)$	Good
$O(\lg n)$	OK
$O(n)$	Bad, but not wrong
$\Theta(n^2)$	Wrong

Analyze test5

```
int test5(int n) {
  int sum = 0;
  for (int i = 0;i < 100000;i++)
    while (n > 0) {
      sum += 1;                    Most executed line
      n = n / 10;
    }
}
```

$f(n) = 100000 * \log_{10}(n)$

$\Theta(\lg n)$	Good
$O(\lg n)$	OK
$O(n)$	Bad, but not wrong
$\Theta(n^2)$	Wrong

Showing $\log_2(n!)$ is $\Theta(n \log_2 n)$

- Will use set definition of Big-Theta

{ $f(n)$ | there exists $c_1 > 0$, $c_2 > 0$ and $n_0 >= 0$

such that $c_1 g(n) <= f(n) <= c_2 g(n)$ for $n >= n_0$ }

- Need to find c_1, c_2 and n_0

Finding c_1 and c_2

Found c_2 = 1 for upper bound

$$n! = n \times n-1 \times n-2 \times n-3 \times \cdots \times 1$$

$$n! \le n \times n \quad \times n \quad \times n \quad \times \cdots \times 1$$

$$\log n! \le \log n^n$$

$\log n^n = n \log n$ when $n \ge 1$

$\log n! \le n \log n$ when $n \ge 1$

Found c_1 = 32 for lower bound

$$n! = n \times n-1 \times \cdots \times \left\lfloor \frac{n}{2} \right\rfloor \times \left(\left\lfloor \frac{n}{2} \right\rfloor - 1\right) \times \cdots \times 1$$

$$n! \ge \left\lfloor \frac{n}{2} \right\rfloor \times \left\lfloor \frac{n}{2} \right\rfloor \times \cdots \times \left\lfloor \frac{n}{2} \right\rfloor \times 1 \times \cdots \times 1$$

$$n! \ge (n/2)^{n/2}$$

$$\log n! \ge \log((n/2)^{n/2})$$

$\log n! \ge 0.4\, n \log n$ when $n \ge 32$

$\log(n/2)^{n/2} = (n/2)\log \mathrm{n} - \mathrm{n}/2$

$0.5n \log n - 0.5n \ge 0.4\, n \log n$ when $n \ge 32$

$$0.1n \log n \ge 0.5n$$
$$0.1 \log n \ge 0.5$$
$$\log n \ge 5$$
$$n \ge 32$$

$0.4\, n \log n \le \log n! \le n \log n$

when $n \ge 32$

$c_1 = 0.4 \quad c_2 = 1 \quad n_0 = 32$

$\log n!$ is $\Theta(n \log n)$

$n \log n$ is $\Theta(\log n!)$