

Ch.02

Characteristics of Good Data Structures & Algorithms

(kmitl) cs-department

Outline

- Good Characteristic : correctness and efficiency
- (Efficiency) Benchmark
- (Efficiency) Operations Counting
- (Asymptotic Analysis) ($O(g(n))$, $\Omega(g(n))$, $\theta(g(n))$)

Recap (Last Chapter)

- Data Structures: Structures for keeping data
- Algorithms: Steps for solving problem
- Data Structures and Algorithms works together to solve problems.

Good Characteristics

- Good characteristics of Data Structures & Algorithms

- **Correctness**

- **Efficiency**

- Time
 - Space

- Efficiency (in space)

- At the early age, we did care about space
 - After memory becomes cheap, we tends to care for speed more
 - In the age of Big Data, space becomes important again, in a whole new level.
 - We typically use the same tool for evaluate efficiency in speed and space.

- Another key good characteristic is **readability**

- **Proof of correctness**

- **Formal reasoning**, a.k.a. Mathematical proof

- (Example of proving techniques)
 - Direct proof
 - Proof by contradiction
 - Proof by contraposition
 - Proof by Mathematical Induction

- **Empirical analysis**

- Good for confirming theory
 - Good enough if we don't have theory?
 - Your "test cases" should be complete enough.
 - Beware of confirmation bias

Correctness

```
1 public class FindMax {  
2     public static void main(String args[]) {  
3         int a[] = {15, 2, 17, 4, 9, 12, 16};  
4         int max = -1;  
5         for(int i=0; i<a.length; i++) {  
6             if(a[i]>max) {  
7                 max = a[i];  
8             }  
9         }  
10        System.out.print("Max is "+max);  
11    }  
12 }
```

```
>java FindMax  
Max is 17
```

```
1 public class FindMax {  
2     public static void main(String args[]) {  
3         int a[] = {-7, -12, -8, -5, -21, -6, -11};  
4         int max = -1;  
5         for(int i=0; i<a.length; i++) {  
6             if(a[i]>max) {  
7                 max = a[i];  
8             }  
9         }  
10        System.out.print("Max is "+max);  
11    }  
12 }
```

```
>java FindMax  
Max is -1
```

Recap

- Characteristics of good Data Structures and Algorithms are **correctness** and **efficiency** (and readability)
- To prove the correctness, we can test it (empirical analysis), or we can formally (mathematically) prove it.
 - Formal proof can guarantee the correctness.
 - **Empirical analysis** need good test cases.

• Measuring Efficiency

Benchmark vs. Asymptotic Analysis

- Benchmark
 - Similar to empirical test
 - Good for measuring end products.
- Asymptotic analysis
 - Similar to Mathematical proof.
 - Concentrate on how time or space are increase as the size of input increase.
 - Compare the increasing trend to a familiar function.

CountPiN.java

- *CountPiN(n)* is a function that return number of prime (e.g. *CountPiN(100) = 25*)

```
public class CountPiN {  
    static boolean isPrime0(int n) {  
        if(n==1) return false;  
        if(n<=3) return true;  
        int m = n/2;  
        for(int i=2; i<=m; i++) {  
            if(n%i==0) return false;  
        }  
        return true;  
    }  
}
```

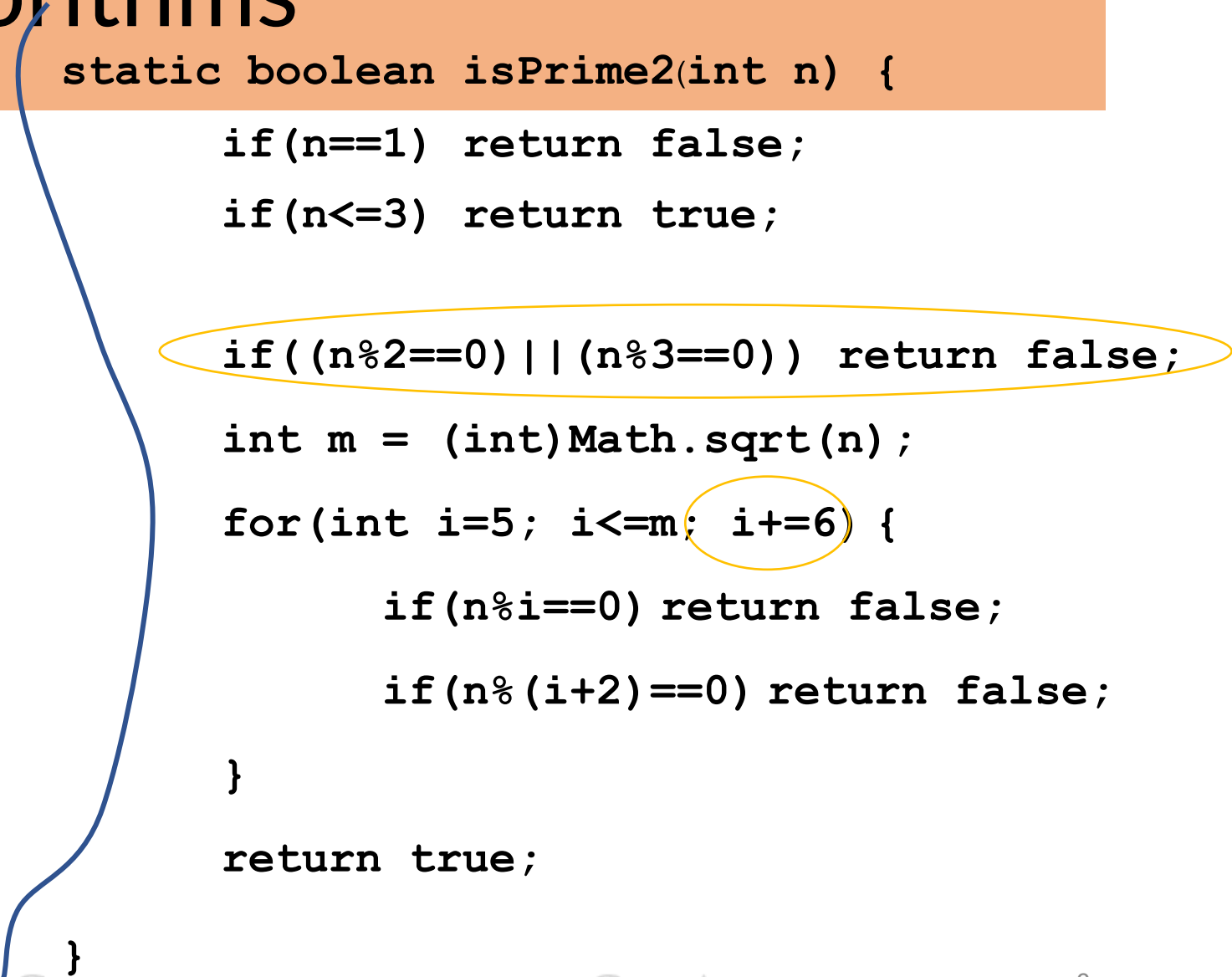
```
public static void main(String[] args) {  
    int count = 0;  
    int N = 100;  
    for(int n=1; n<N; n++) {  
        if(isPrime0(n)) count++;  
    }  
    System.out.println("Pi("+N+")="+count);  
}
```

Another two algorithms

```
static boolean isPrime1(int n) {  
    if(n==1) return false;  
    if(n<=3) return true;  
    int m = (int)Math.sqrt(n);  
    for(int i=2; i<=m; i++) {  
        if(n%i==0) return false;  
    }  
    return true;  
}
```


Another two algorithms

```
static boolean isPrime2(int n) {  
    if(n==1) return false;  
    if(n<=3) return true;  
    if((n%2==0) || (n%3==0)) return false;  
    int m = (int)Math.sqrt(n);  
    for(int i=5; i<=m; i+=6) {  
        if(n%i==0) return false;  
        if(n%(i+2)==0) return false;  
    }  
    return true;  
}
```



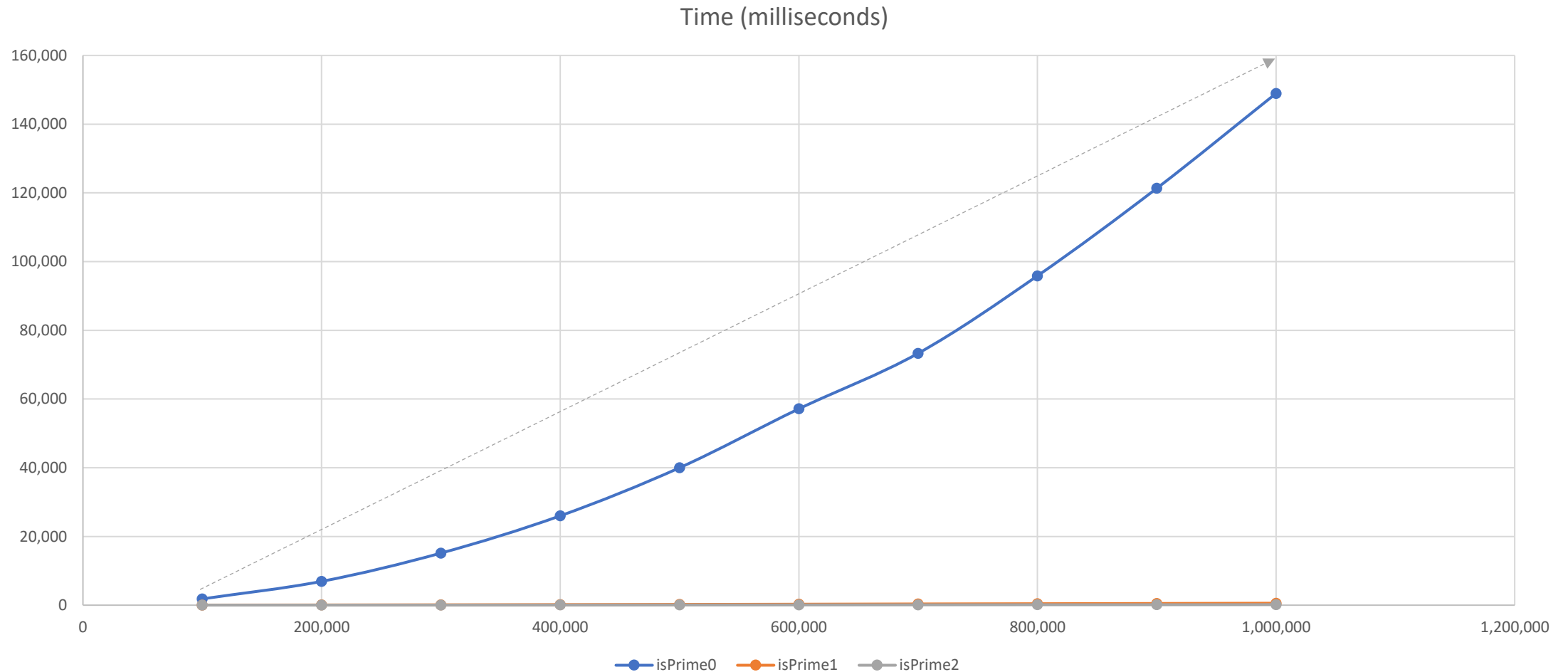
Modified main() method

```
public static void bench_isPrime(L2_IsPrimeInterface obj) {  
    int your_cpu_factor = 1; /* increase by 10 times */  
    int N = 100;  
    int count = 0;  
    for (N = 100_000; N <= 1_000_000 * your_cpu_factor; N+= 100_000 * your_cpu_factor) {  
        long start = System.currentTimeMillis();  
        for (int n = 1; n < N; n++) {  
            if (obj.isPrime(n)) count++;  
        }  
        long time = (System.currentTimeMillis() - start);  
        System.out.println(N + "\t" + count + "\t" + time);  
    }  
}
```

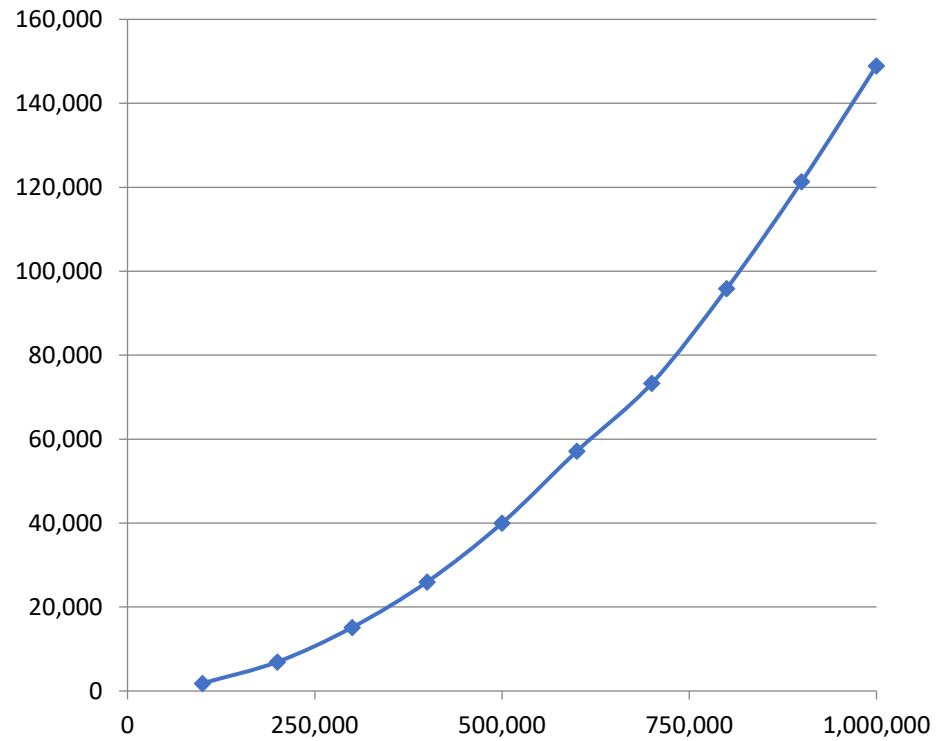
Recorded Results

N	$\pi(n)$	Times(milliseconds)		
		isPrime0	isPrime1	isPrime2
100,000	9,592	1,828	38	26
200,000	17,984	6,927	66	29
300,000	25,997	15,153	108	37
400,000	33,860	26,004	163	54
500,000	41,538	39,993	219	75
600,000	49,098	57,139	280	94
700,000	56,543	73,301	353	118
800,000	63,951	95,851	419	139
900,000	71,274	121,327	492	141
1,000,000	78,498	148,958	576	164

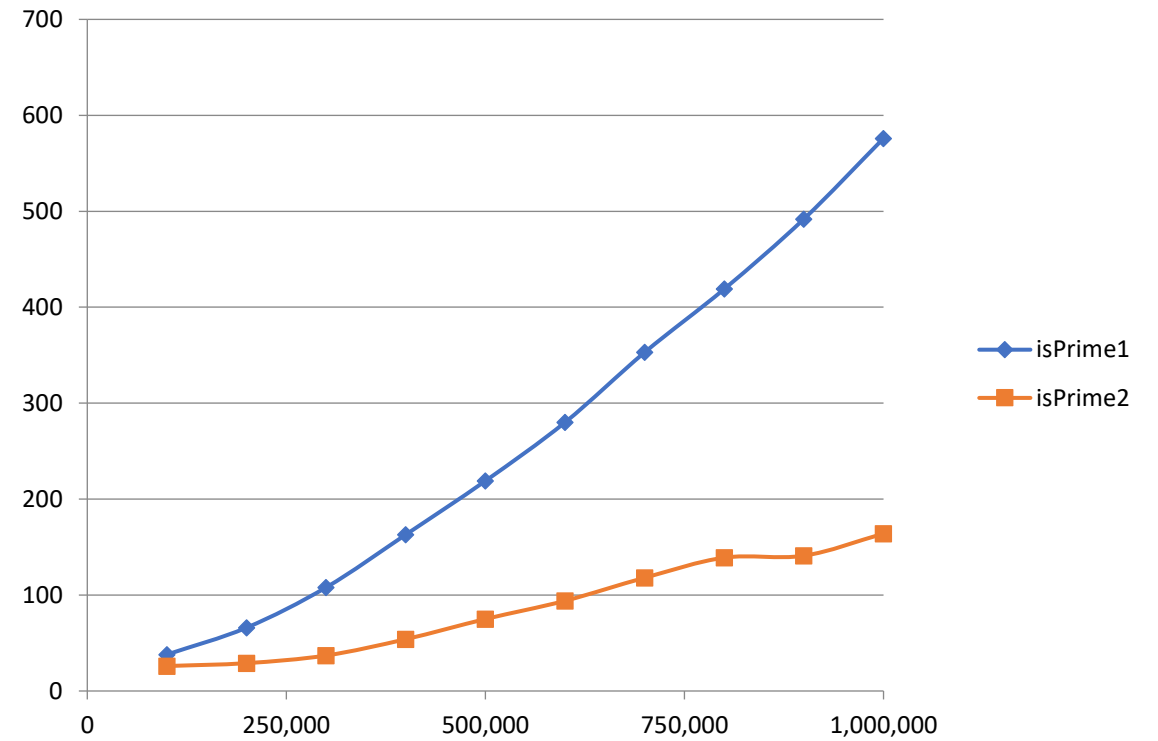
Performance Graph



Performance Graphs



isPrime0

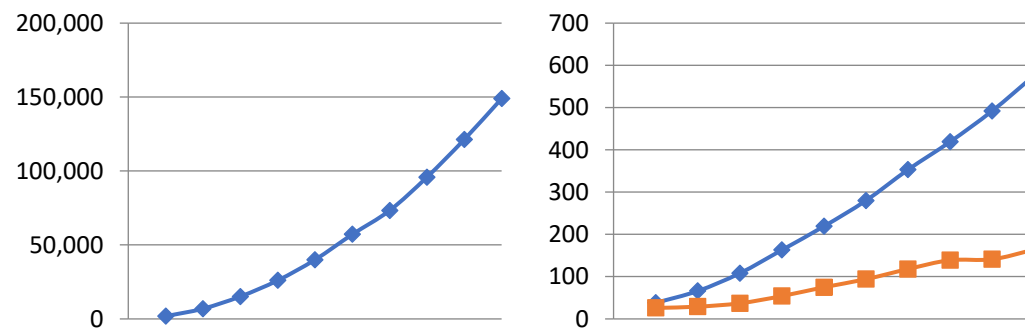


isPrime1

isPrime2

Observations

- As N grows, the computing time is longer.
- isPrime0 is noticeably slower, comparing to the other two
- isPrime1 and isPrime2 are comparatively similar
- We can safely say that isPrime0 is inferior to the other two
- However, if our program only need to compute $\pi(n)$ where n is relatively small, and only for a few times, any methods will do.



Benchmarking

- For comparing algorithms, using benchmark has its **limitations**. We need to be careful in these issues:
 - Must be done in the same environment, including hardware, operating system, selected computer language, etc.
 - Implementation details should be the same
 - May not reflect the real environment
 - May not reflect size of data, especially in the future.
- Good for measuring finished products.
- We see them a lot in hardware testing.
- Key Properties
 - Relevance: Focus on vital features.
 - Representativeness: Accepted performance metrics.
 - Equity: Should be fairly compared.
 - Repeatability: Can be verified.
 - Cost-effectiveness: Should be economical.
 - Scalability: Should be able to test all range of system.
 - Transparency: Should be easy to understand.

Why don't we use real world clock?

- Real world clock measurement is possible but has many drawback
 - System dependency
 - Too complex (we have to build the system.)
 - Too specific
- 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

Estimating Time by Counting Operations

- We will start by assuming that one command, one programming statement, takes a fixed amount of time.
 - If we have m statement, it will take km milliseconds to follow them.
 - Linearly proportional to each other.
- For simplicity, we will assume that any statement takes the same amount time to compute.
 - For example, `x=3`; or `Math.abs(-178)` are consider to take the same amount of time to compute.
 - This might not be true, but good enough for our purpose.
 - Note that `int m=n/2`; is count as 3 operations: declaration, assignment, and division.
- Let's start counting

isPrime0()

code		operation count
1	static boolean isPrime0(int n) {	
2	if(n==1) return false;	1
3	if(n<=3) return true;	1
4	int m=n/2;	3
5	for(int i=2; i<=m; i++) {	$2 + ((m-1)+1) + (m-1) = 2m+1$
6	if(n%i==0) return false;	$2(m-1)$
7	}	
8	return true;	1
9	}	
total		$= 4m + 5$
		$= 2n + 5$

isPrime1()

code		operation count
1	static boolean isPrime1(int n) {	
2	if(n==1) return false;	1
3	if(n<=3) return true;	1
4	int m= (int)Math.sqrt(n);	4
5	for(int i=2; i<=m; i++) {	2 + ((m-1)+1) + (m-1) = 2m+1
6	if(n%i==0) return false;	2(m-1)
7	}	
8	return true;	1
9	}	
total		= 4m + 6
		= 4√n + 6

isPrime2()

	code	operation count
1 2 3 4 5 6 7 8 9 10 11	<pre> static boolean isPrime2(int n) { if(n==1) return false; if(n<=3) return true; if((n%2==0) (n%3==0)) return false; int m = (int)Math.sqrt(n); for(int i=5; i<=m; i+=6) { if(n%i==0) return false; if(n%(i+2)==0) return false; } return true; } </pre>	<div>1</div> <div>1</div> <div>5</div> <div>4</div> <div> $2 + (\frac{m}{6} + 1) + \frac{m}{6} = 2\frac{m}{6} + 3$ </div> <div> $2\frac{m}{6}$ </div> <div> $3\frac{m}{6}$ </div> <div>1</div>
total		$= 7\frac{m}{6} + 15$
		$= 7\frac{\sqrt{n}}{6} + 15$

Operation Function

- These are functions of number operations where n is the size of the input

$$f_0(n) = 2n + 5$$

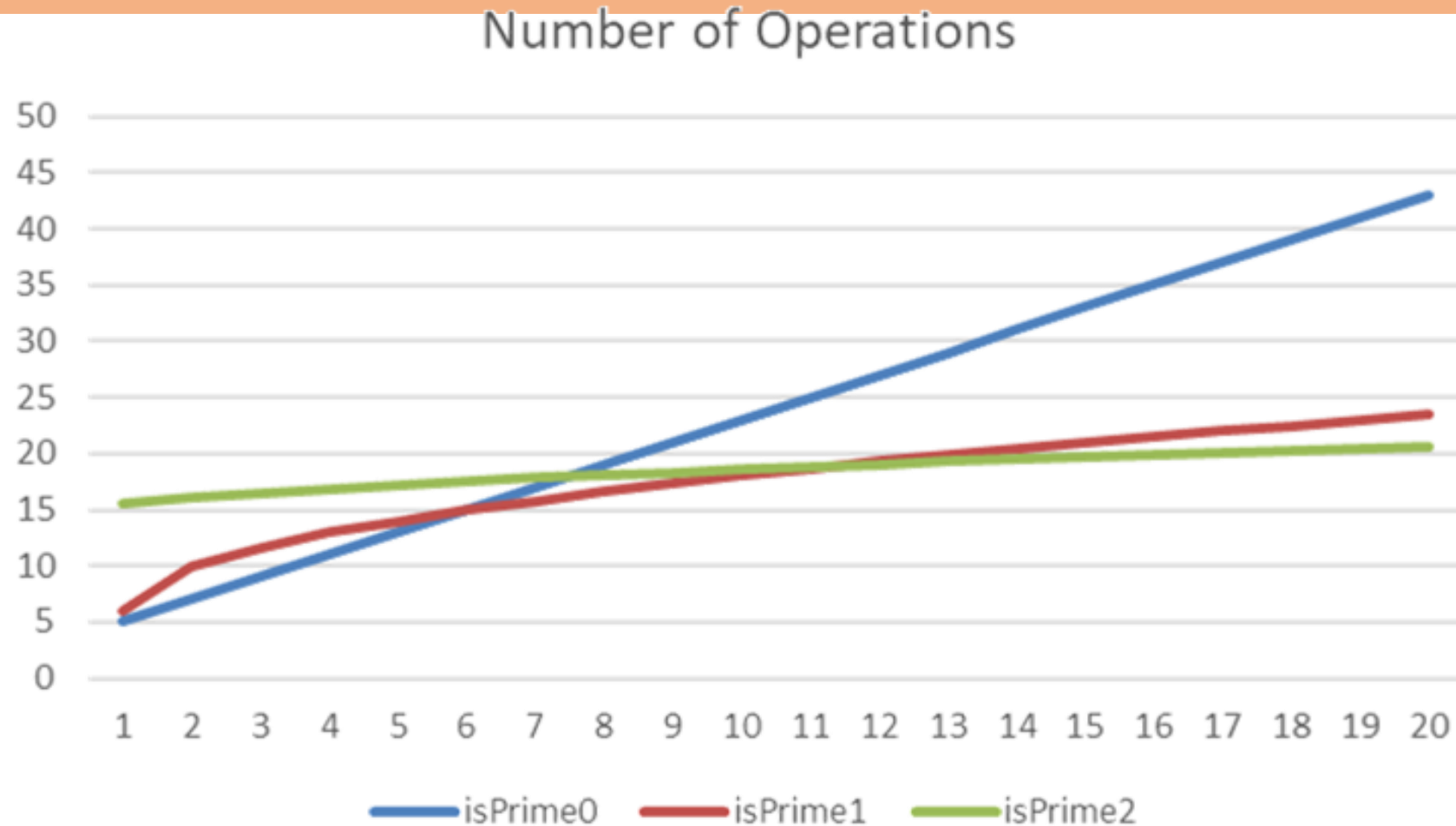
$$f_1(n) = 4\sqrt{n} + 6$$

$$f_2(n) = 7\frac{\sqrt{n}}{6} + 15$$

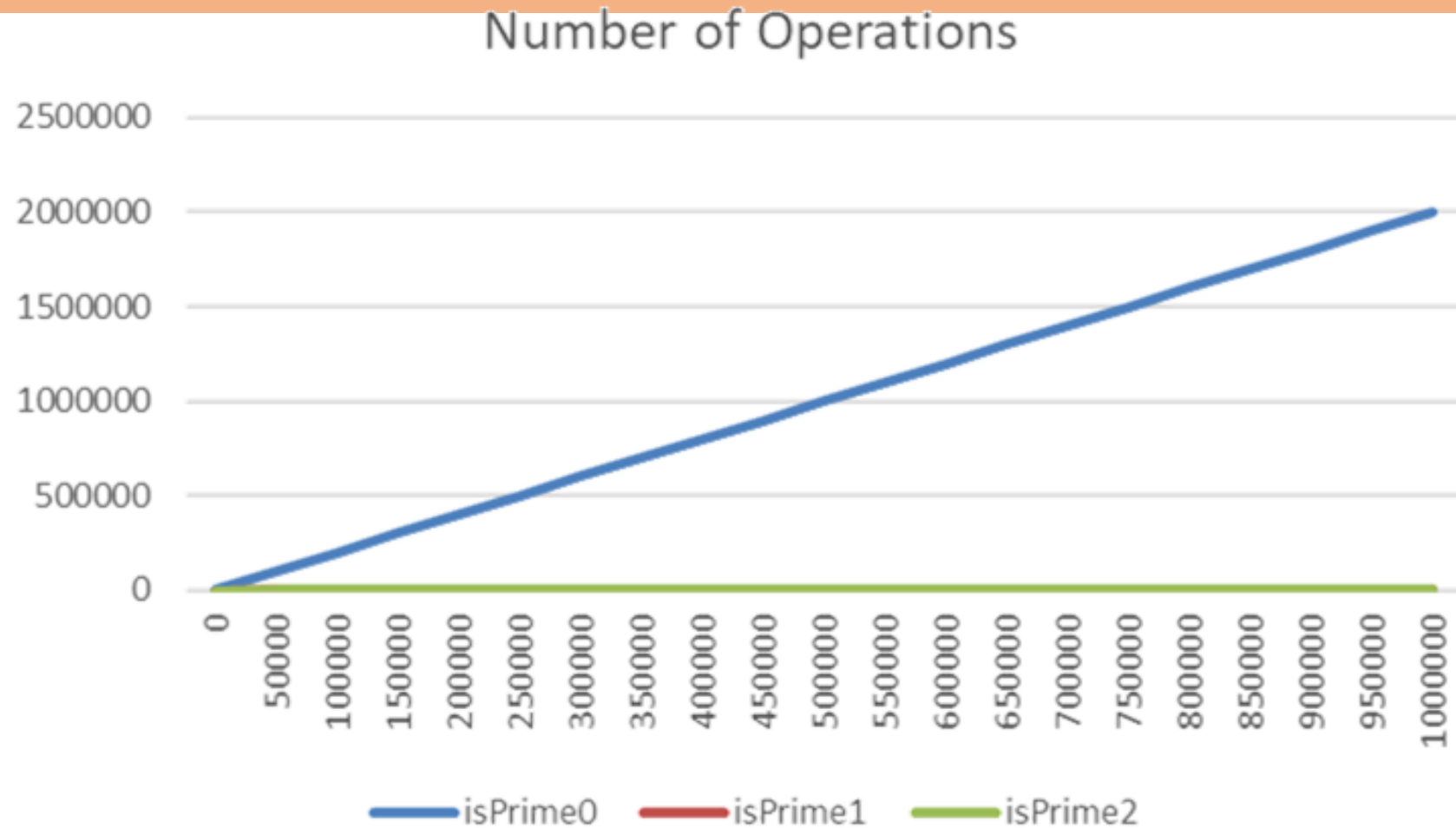
Let's call them Operation Function

- Let's plot some graph

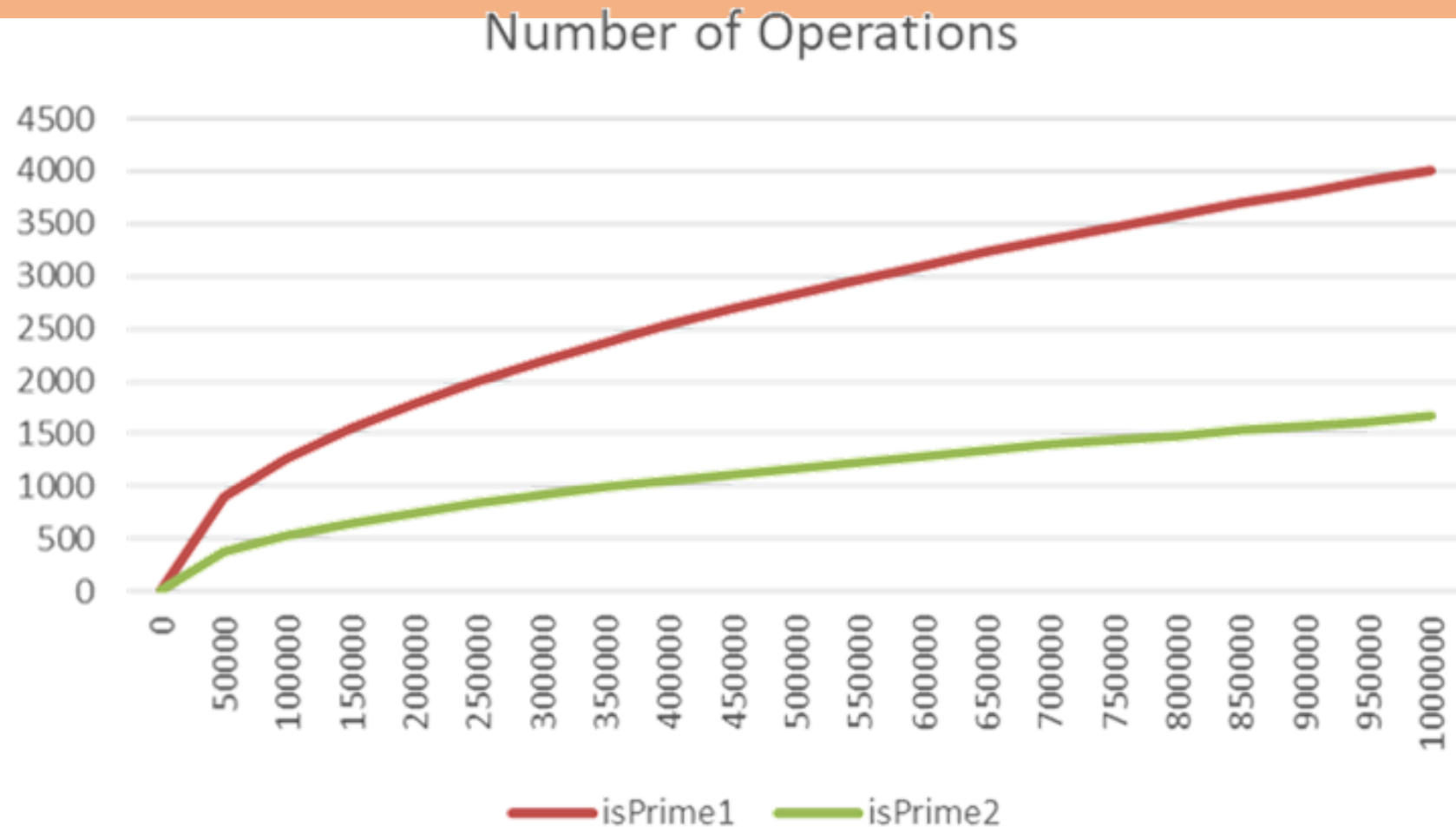
Small input ($n \leq 20$)



Large Input



Large Input w/o isPrime0



Some Discussion

$$f_0(n) = 2n + 5 \quad f_1(n) = 4\sqrt{n} + 6 \quad f_2(n) = 7\frac{\sqrt{n}}{6} + 15$$

- The result of statement counting is similar to our benchmark results.
 - isPrime0 grows at the same rate as the input.
 - isPrime1 and isPrime2 grows proportion to the square root of the input.
 - isPrime1 and isPrime2 are comparable, while isPrime0 is by far the slowest.
- It is typically possible to count number of operations of algorithms.
- (Remark) While the actual time depends on machines, number of operations does not.
 - Make it good for directly compare algorithms
- It is also applicable to counting space.
- Rather than remember the functions of all known algorithms, we will compare them to well-known functions using **Asymptotic Analysis**.

Recap (cont.)

- Measuring efficiency can be done by benchmark and asymptotic analysis.
 - Benchmark good for measuring end products
 - Asymptotic analysis measure **growth rate** of time or space comparing to the growth rate input.
 - Rather than measuring the time, we can count number of operations in algorithms
 - Number of operations are just an estimate number, which is good enough for our purpose.
 - If we know the function of operations of algorithms, we can compare them easily.
- However, we will not have to memorize the exact function of operations of each algorithm, we will use **Asymptotic Analysis** to compare them to well-known function.

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
- Emphasize long term trend

Why?

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

Why should we care for large n ?

A=	$5n^2 + 4n + 4$	B=	$3n^2 + 3n + 4$		n^2		$T(A(n))/T(B(n))$
n	raw_time	growth	raw_time	growth	raw_time	growth	
10	544		334		100		1.628742515
20	2084	3.830882353	1264	3.784431138	400	4.0	1.648734177
40	8164	3.917466411	4924	3.895569620	1600	4.0	1.658001625
80	32324	3.959333660	19444	3.948822096	6400	4.0	1.662415141
160	128644	3.979829229	77284	3.974696564	25600	4.0	1.664561876
320	513284	3.989956780	308164	3.987423011	102400	4.0	1.665619605
640	2050564	3.994989129	1230724	3.993730611	409600	4.0	1.666144481
1280	8197124	3.997497274	4919044	3.996870135	1638400	4.0	1.666405911
2560	32778244	3.998749317	19668484	3.998436282	6553600	4.0	1.666536374
5120	1.31E+08	3.996553324	78658564	3.999218445	26214400	4.0	1.665425776
10240	5.24E+08	4.000000000	3.15E+08	4.004649767	1.05E+08	4.0	1.663492063
20480	2.10E+09	4.007633588	1.26E+09	4.000000000	4.19E+08	4.0	1.666666667

Comparing Growth Rate

$$f_0(n) = 2n + 5 \quad f_1(n) = 4\sqrt{n} + 6 \quad f_2(n) = 7\frac{\sqrt{n}}{6} + 15$$

- These are functions from size of input to number of computing operation of isPrime0, isPrime1, and isPrime2
- We can say that the growth rates of $f_1(n)$ and $f_2(n)$ are mathematically similar taking a limit as n approach infinity

$$\lim_{n \rightarrow \infty} \frac{f_1(n)}{f_2(n)} = \lim_{n \rightarrow \infty} \frac{4\sqrt{n} + 6}{7\frac{\sqrt{n}}{6} + 15} = \frac{24}{7}$$

- These means that, $f_1(n)$ and $f_2(n)$ grows together to infinity.

Comparing Growth Rate

- How about $f_0(n)$ comparing to $f_1(n)$?

$$\lim_{n \rightarrow \infty} \frac{f_0(n)}{f_1(n)} = \lim_{n \rightarrow \infty} \frac{2n + 5}{4\sqrt{n} + 6} = \infty$$

- This means that $f_0(n)$ is bigger than $f_1(n)$ if n is large enough.

- Now, let compare $f_2(n)$ to $f_0(n)$:

$$\lim_{n \rightarrow \infty} \frac{f_2(n)}{f_0(n)} = \lim_{n \rightarrow \infty} \frac{7\frac{\sqrt{n}}{6} + 15}{2n + 5} = 0$$

- This means $f_2(n)$ is small comparing to $f_0(n)$ if n is large enough.

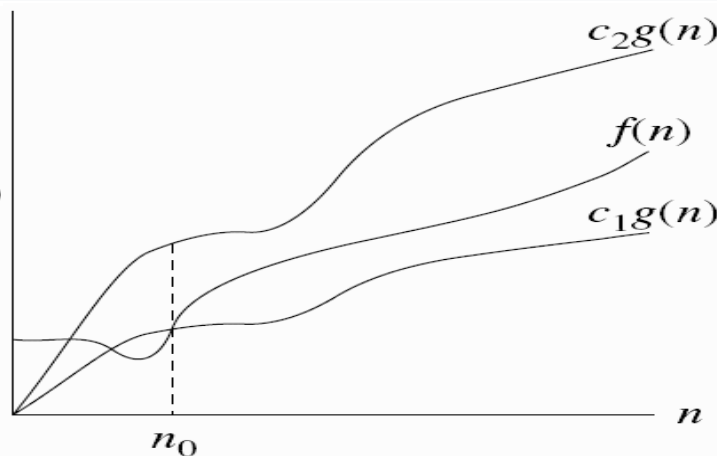
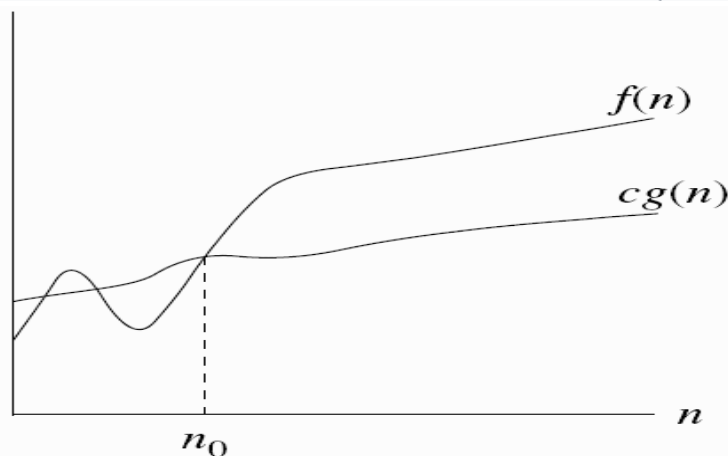
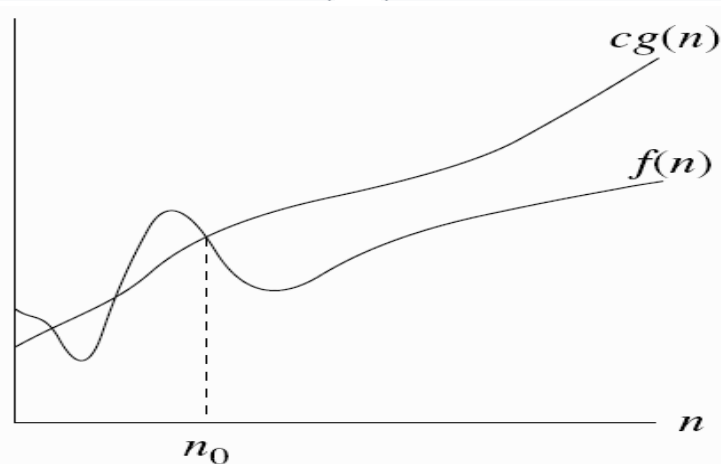
Asymptotic Analysis ($O(g(n))$, $\Omega(g(n))$, $\theta(g(n))$)

- We are to compare growth rates of functions, as their input increase (up to infinity), to another function.
- There are three asymptotic analysis commonly used in Computer Science:
 - $f(n) \in O(g(n))$ read **Big-O** of $f(n)$ is $g(n)$, means that $g(n)$ is an upper bound of $f(n)$ or $f(n)$ grows asymptotically **no faster than $g(n)$** .
 - Upper bound implies not worse than
 - $f(n) \in \Omega(g(n))$ read **Big-Omega** of $f(n)$ is $g(n)$, means that $g(n)$ is a lower bound of $f(n)$ or $f(n)$ grows asymptotically **faster than $g(n)$** .
 - Lower bound implies no better than
 - $f(n) \in \theta(g(n))$ read **Big-Theta** of $f(n)$ is $g(n)$, means that $f(n)$ and $g(n)$ are growing at that same rate or $f(n)$ grows asymptotically **as fast as $g(n)$** .

Note that $f(n)$ and $g(n)$ are defined on unbound subset of positive real number and $g(n)$ is strictly positive for all large enough n .

Formal Definition

Notation	Formal Definition	Limit Definition ^{[24][25][26][23][18]}
$f(n) = O(g(n))$	$\exists k > 0 \exists n_0 \forall n > n_0: f(n) \leq k \cdot g(n)$	$\limsup_{n \rightarrow \infty} \frac{ f(n) }{g(n)} < \infty$
$f(n) = \Theta(g(n))$	$\exists k_1 > 0 \exists k_2 > 0 \exists n_0 \forall n > n_0:$ $k_1 \cdot g(n) \leq f(n) \leq k_2 \cdot g(n)$	$f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$ (Knuth version)
$f(n) = \Omega(g(n))$	$\exists k > 0 \exists n_0 \forall n > n_0: f(n) \geq k \cdot g(n)$	$\liminf_{n \rightarrow \infty} \frac{f(n)}{g(n)} > 0$



Simple $g(n)$

- Rather than comparing to each other, we will compare each function with a simple function, for example

$$f_0(n) = 2n + 5 \in O(n)$$

$$f_1(n) = 4\sqrt{n} + 6 \in O(\sqrt{n})$$

$$f_2(n) = 7\frac{\sqrt{n}}{6} + 13 \in O(\sqrt{n})$$

- So, we are saying that isPrime0 has $O(n)$ while isPrime1 and isPrime2 has $O(\sqrt{n})$.
 - Thus, we regards isPrime1 and isPrime2 as equal and superior to isPrime0

Note on Big-O

- In Computer Science, we often use only Big-O for asymptotic analysis.
- However, people are expecting the smallest Big-O.
 - Although we say Big-O, we are expecting **Big-Theta**.
 - To prevent side effect of big-o broad definition ($< \infty$), though people normally imply Big-O in tight upper-bound manner.
 - For example, if you say $f_1(n) = 4\sqrt{n} + 6 \in O(n)$ is **technically correct**, but people will be expecting $O(\sqrt{n})$ and **tends to think you are wrong**.
 - Thus, to be the safe side, although we say Big-O, we should use Big-Theta whenever possible.

Comparing Growth Rate (more example)

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

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

- Hence $f(n)$ grows similar to $g(n)$
 - Therefore $f(n) = \theta(n^2)$ // usually we determine whether it is theta by
// examining the code
- (More example growth rate) $f(n) = \log(n)$ vs. $g(n) = \text{sqrt}(n)$?

In practice – determine the big O (most frequently executed line)

```
function partition(array, low, high)
    // Choose the last element as the pivot
    pivot = array[high]
    i = low - 1 // Index of smaller element

    for j = low to high - 1 do
        if array[j] <= pivot then
            i = i + 1
            swap array[i] and array[j]
        end if
    end for

    // Place the pivot in its correct position
    swap array[i + 1] and array[high]

    return i + 1 // Return the pivot index

end function
```

In practice – determine the big O (most frequently executed line)

```
function hoare_partition(arr, low, high):  
    # Choose a pivot (e.g., the first element)  
    pivot = arr[low]  
    i = low - 1  
    j = high + 1  
  
    while True: # Move i to the right until an element >= pivot is found  
        do  
            i = i + 1  
        while arr[i] < pivot # Move j to the left until an element <= pivot is found  
        do  
            j = j - 1  
        while arr[j] > pivot  
  
        if i >= j: # If i and j have crossed, partitioning is done  
            return j # or return i, either is fine  
  
    swap(arr[i], arr[j]) # Swap the elements at i and j
```

More on $O(n)$ vs. $\theta(n)$

```

/*****
 * VERSION A – early exit when we discover the first divisor *
 *****/
static boolean isPrimeEarly(int n) {
    if (n <= 1) return false;
    if (n == 2) return true;
    if (n % 2 == 0) return false;           // ← may finish in  $O(1)$ 

    int limit = (int) Math.sqrt(n);
    for (int d = 3; d <= limit; d += 2) {
        if (n % d == 0) return false;       // ← EARLY-EXIT
    }
    return true;                           // ← only happens if n is prime
}
/*
  ▶ Complexity
    best-case :  $O(1)$            (when n is even)
    worst-case:  $O(\sqrt{n})$      (when n is prime)
  ▶ We cannot pin it down to a single  $\theta$ -expression, because the running
    time varies between  $\theta(1)$  and  $\theta(\sqrt{n})$ . All we can safely say is that
    it is Big- $O(\sqrt{n})$ .
*/

```

```

/*****
 * VERSION B – scan the whole range, no early return *
 *****/
static boolean isPrimeFullLoop(int n) {
    if (n <= 1) return false;

    boolean prime = true;
    int limit = (int) Math.sqrt(n);
    for (int d = 2; d <= limit; d++) {       // ← ALWAYS  $\sqrt{n}$  iterations
        if (n % d == 0) prime = false;       // (but keep looping)
    }
    return prime;
}

```

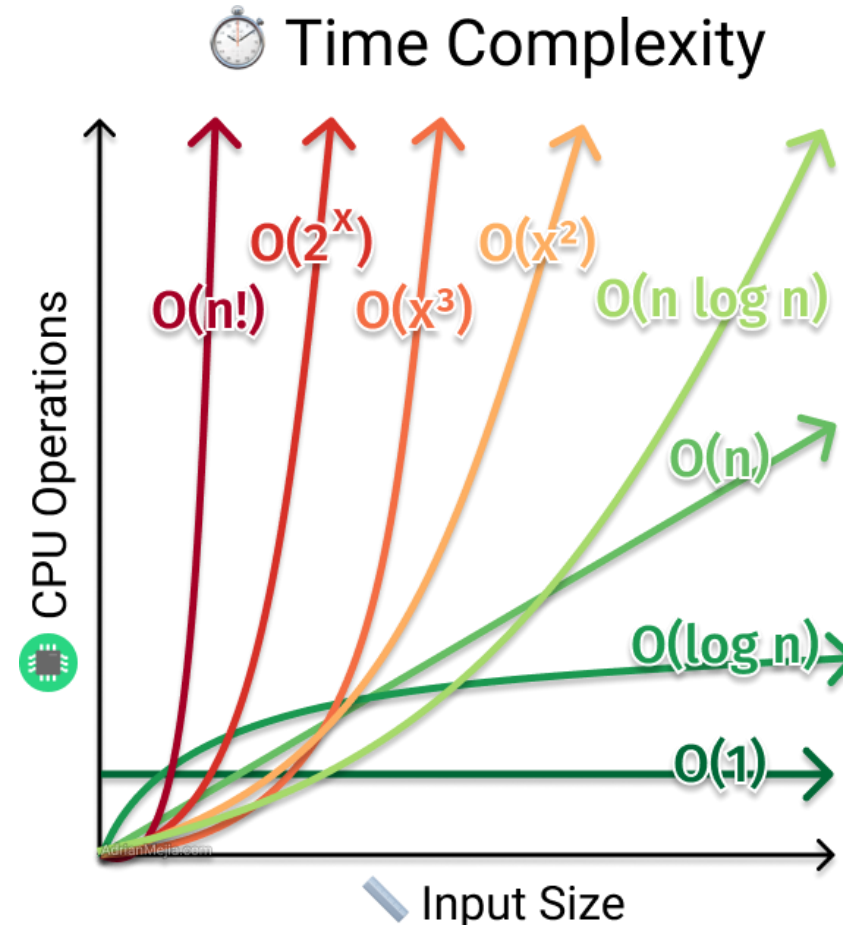
```

/*
  ▶ Complexity
    every input :  $c \cdot \sqrt{n} + O(1)$  operations
  ▶ Here the loop executes exactly  $\lfloor \sqrt{n} \rfloor - 1$  times no matter what.
    The work is therefore bounded both above and below by  $k \cdot \sqrt{n}$  for
    positive constants  $k$ , giving a tight Big- $\theta(\sqrt{n})$  bound.
*/

```

Typical Big-O

Big-O	Name
$O(1)$	Constant
$O(\log n)$	Logarithmic
$O(n)$	Linear
$O(n \log n)$	Just $n \log n$
$O(n^2)$	Quadratic
$O(n^k)$	Polynomial
$O(c^n)$	Exponential
$O(n!)$	Factorial
$O(n^n)$	Very bad!



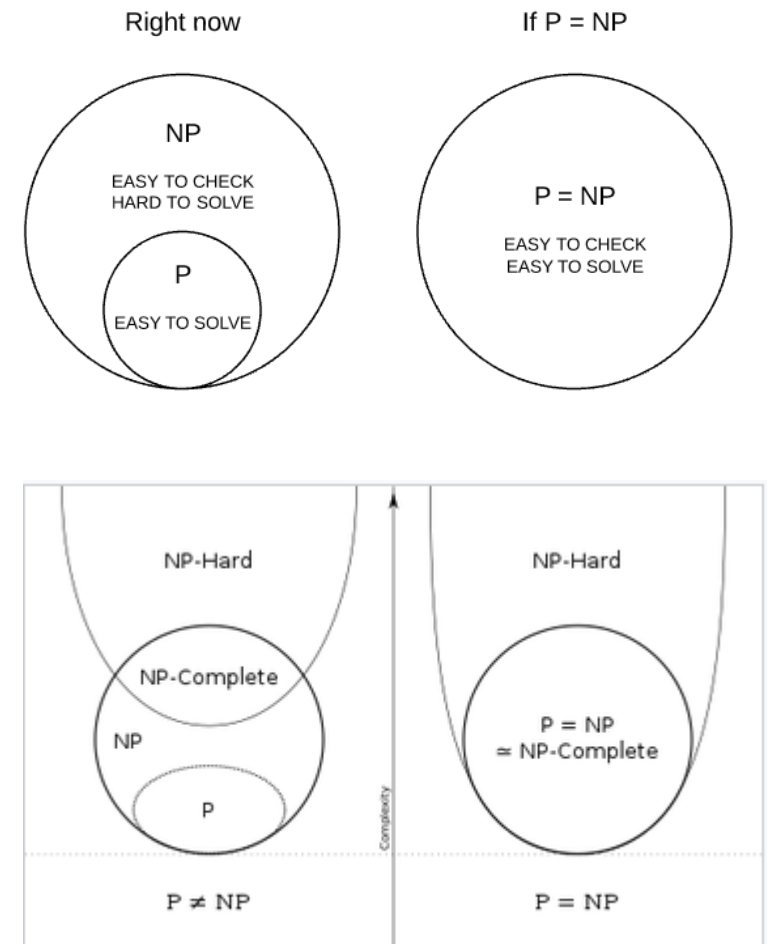
* image from adrianmejia.com

P vs NP problems

- If a problem is solvable with polynomial time, $O(n^k)$, or less, we call them a P class problem.
- NP-complete problem means a problem with no polynomial time solution.
- Although widely accepted, there is **no definite proof** that $P \neq NP$

*NP is short for
“nondeterministic polynomial time”*

Millennium 7 Prize Problems	
1. P=NP	Can the dispatcher be lazy ? 1970s
2. Hodge	Can high dim space be decomposed ? 1930s
3. Poincare	Apple is not same as donut ! 1900s
4. Riemann	Is prime distribution regular ? 1850s
5. Yang-Mills	There is always a ghost ! ? 1950s
6. Navier-Stokes	How many solutions ? 1820s
7. Birch Swinnerton-Dyer	Only oval fly!



Summary

- To proof the correctness, we can test it (empirical analysis), or we can formally (mathematically) proof it.
 - Empirical analysis need good test cases.
 - Formal proof can guarantee the correctness.
- Measuring efficiency can be done by benchmark and asymptotic analysis.
 - Benchmark good for measuring end products
 - Asymptotic analysis measure growth rate of time or space comparing to the growth rate input.
 - Asymptotic analysis measures computational complexity by comparing them asymptotically to well known functions.
 - There are several well known functions that we are typically compare our function to.