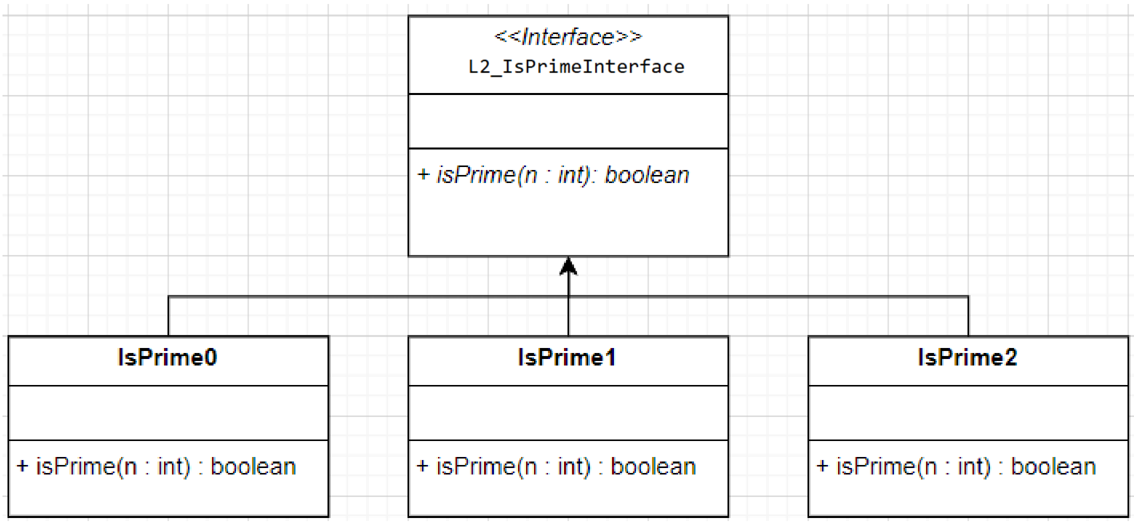
Objective(s):

- To practice on analyzing algorithms' runtime

Task 1: Implement IsPrime0, IsPrime1 and IsPrim2. (in .\Lab02\pack)




```java
package pack;

public interface L2_IsPrimeInterface {
    boolean isPrime(int n);
}
```

```java
// IsPrime0
public boolean isPrime(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;
}
```

```java
// IsPrime1
public boolean isPrime(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;
}
```

```java
// IsPrime2
public boolean isPrime(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;
}
```

The method isPrime0(n) takes any positive integer and returns true if it is a prime, false otherwise. The method run through all integer from 2 to n/2 and check if n is divisible by any of them.

There are two more methods, isPrime1(n) and isPrime2(n). The method isPrime1(n) is similar to isPrime0(n) but only run from 2 to $\sqrt{n}$. The method isPrime2(n) improves upon isPrime1(n) by take out anything divisible by 2 and 3 and not going to test divisibility of number that are multiple of 2 and 3.

For testing, we can use the following program:

```
private static void testIsPrime012() {
        int N = 100;
        int count = 0;
        L2_IsPrimeInterface obj = new IsPrime0();
        for (int n = 1; n < N; n++) {
            if (obj.isPrime(n)) count++;
        }
        System.out.println("Pi ("+ N + ")= " + count);
        count = 0;
        obj = new IsPrime1();
        for (int n = 1; n < N; n++) {
            if (obj.isPrime(n)) count++;
        }
        System.out.println("Pi ("+ N + ")= " + count);
        count = 0;
        obj = new IsPrime2();
        for (int n = 1; n < N; n++) {
            if (obj.isPrime(n)) count++;
        }
        System.out.println("Pi ("+ N + ")= " + count);
}
```

Remark : There are 25 prime numbers between 2 to 100. Check the correctness of your implementation.

Task 2: run the program with isPrime0, isPrime1, and isPrime2. Record your result into the following table.

Running-time table					
n	numPrime(n)	time (milliseconds)			
		Lab's isPrime0	isPrime0	isPrime1	isPrime2
100,000		353			
200,000		1,283			
300,000		2,792			
400,000		4,820			
500,000		7,370			
600,000		15,580			
700,000		24,557			
800,000		31,716			
900,000		39,964			
1,000,000		48,785			

```java
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) {
        count = 0;
        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);
        // System.out.printf("%s\t %s\t %s",String.format("%,d",N),
String.format("%,d",count), String.format("%,d",time));
    }
}
```

Taks 3 : Analyze whether time increased on isPrime0 is linear.

When running algorithms on different input sizes, it's important to understand how the runtime grows. This helps us understand the behavior of the growth rate, and how it compares to the expected Big O complexity.

(current) time_ratio
(prev) time_ratio

Complete the table below.

Running-Time Analysis							
n	Data size ratio	Lab's (example) isPrime0	Time Ratio (compared to n)	(Time) Increased Factor	your isPrime0	Time Ratio (compared to n)	(Time) Increased Factor
100,000	n	353	1.0000	-			
200,000	2n	1,283	3.6345	3.6345			
300,000	3n	2,792	7.9095				
400,000	4n	4,820	13.6543				
500,000	5n	7,370	20.4413				
600,000	6n	15,580	44.1359				
700,000	7n	24,557	69.5665				
800,000	8n	31,716	89.8470				
900,000	9n	39,964	113.2124				
1,000,000	10n	48,785	138.2011				

What Should We Expect?

1. Should growth per step stabilize?

   - Yes, for linear algorithms ⟶ growth per 100k should stay ~constant.

   - If it increases, the algorithm may be O(n log n) or O(n^2).

2. Should increase factor converge?

- Yes — this reflects the asymptotic behavior.
- In this case, it approaches ~1.2 — a sign of super-linear growth.

3. Does Machine Matter?

If we run the same algorithm on different machines (e.g., Windows vs macOS):

- Raw runtimes will differ due to hardware/OS differences.
- But the time ratios and increase factors should follow the same pattern.

Time ratio stays the same, even if absolute values change.

Key Takeaways

- Time ratio and growth analysis help determine algorithm complexity.
- Growth trends and increase factors help distinguish between O(n), O(n log n), and O(n^2).
- As input size grows, patterns converge, even on different machines.

Task 4: Plot 2 runtime graphs your isPrime0’s vs. your isPrime1’s and isPrime1’s vs. isPrime2’s

The following code might be helpful.

```
import matplotlib.pyplot as plt

# Sample input sizes
input_sizes = [100_000, 200_000, 300_000, 400_000, 500_000, 600_000, 700_000,
800_000, 900_000, 1_000_000]

# Sample runtimes in milliseconds (replace with real data)
runtime_A = [100, 200, 300, 400, 500, 600, 700, 800, 900, 1000]          #
O(n)
runtime_B = [130, 290, 470, 680, 920, 1190, 1490, 1810, 2150, 2500]       #
O(n log n)
runtime_C = [90, 350, 780, 1400, 2200, 3200, 4400, 5800, 7400, 9200]      #
O(n^2)

# Plotting
plt.figure(figsize=(10, 6))
plt.plot(input_sizes, runtime_A, marker='o', label='Algorithm A (O(n))')
plt.plot(input_sizes, runtime_B, marker='s', label='Algorithm B (O(n log n))')
plt.plot(input_sizes, runtime_C, marker='^', label='Algorithm C (O(n²))')

plt.title('Runtime Comparison of Three Algorithms')
plt.xlabel('Input Size (n)')
plt.ylabel('Runtime (ms)')
plt.grid(True)
plt.legend()
plt.tight_layout()
#export to PNG/PDF using plt.savefig('filename.png')
plt.show()
```

Submission: this pdf. L2_IsPrime0.java L2_IsPrime1.java and L2_IsPrime2.java

Due Date: TBA