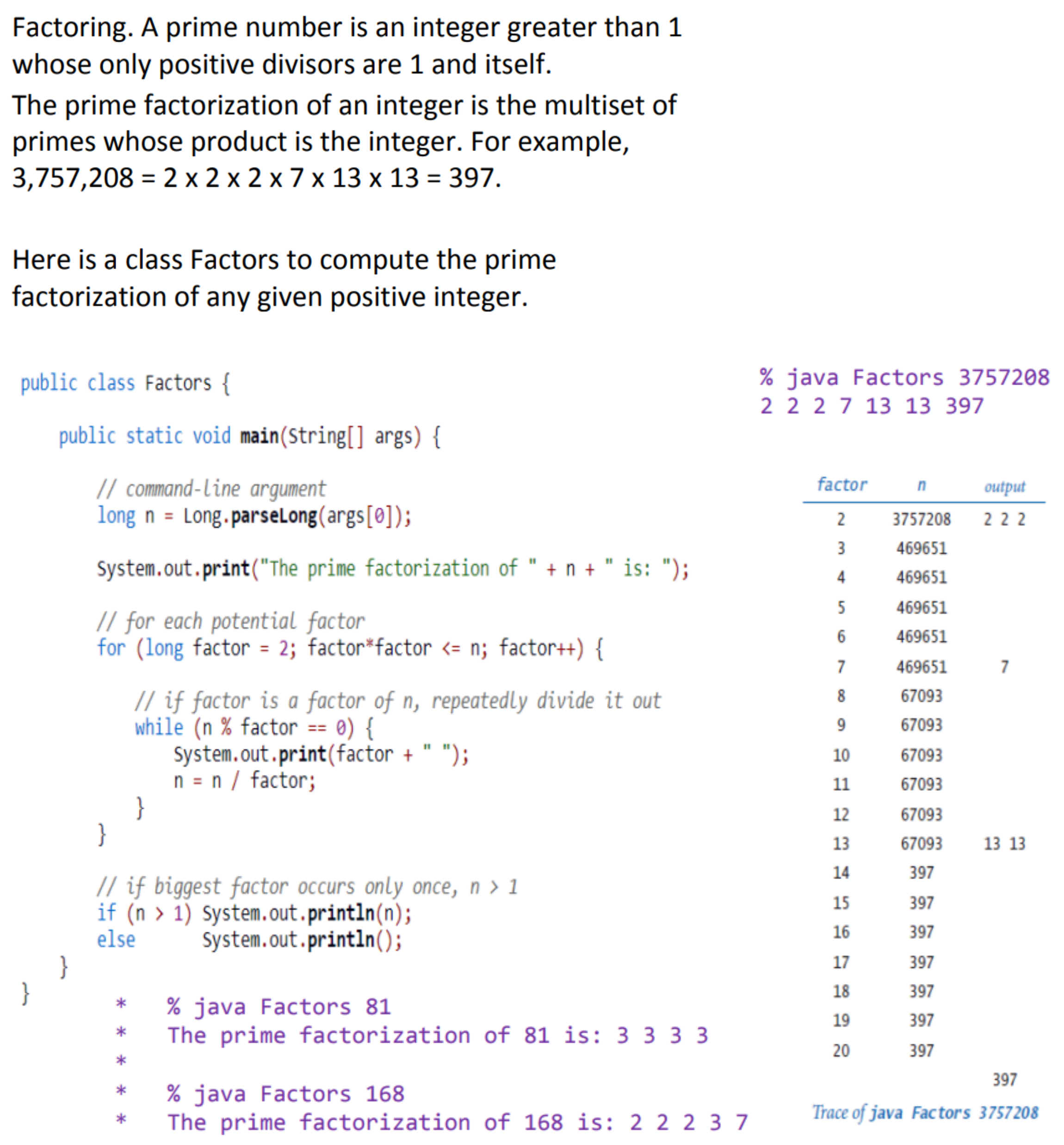
In-Class Question

Lecture1

Finding factors

找到一个数由哪些质数的乘积组成



Factoring. A prime number is an integer greater than 1 whose only positive divisors are 1 and itself.
The prime factorization of an integer is the multiset of primes whose product is the integer. For example, 3,757,208 = 2 x 2 x 2 x 7 x 13 x 13 = 397.

Here is a class Factors to compute the prime factorization of any given positive integer.

```java
public class Factors {

    public static void main(String[] args) {

        // command-line argument
        long n = Long.parseLong(args[0]);

        System.out.print("The prime factorization of " + n + " is: ");

        // for each potential factor
        for (long factor = 2; factor*factor <= n; factor++) {

            // if factor is a factor of n, repeatedly divide it out
            while (n % factor == 0) {
                System.out.print(factor + " ");
                n = n / factor;
            }
        }

        // if biggest factor occurs only once, n > 1
        if (n > 1) System.out.println(n);
        else       System.out.println();
    }
}
```

```
 *   % java Factors 81
 *   The prime factorization of 81 is: 3 3 3 3
 *
 *   % java Factors 168
 *   The prime factorization of 168 is: 2 2 2 3 7
```

```
% java Factors 3757208
2 2 2 7 13 13 397
```

factor	n	output
2	3757208	2 2 2
3	469651	
4	469651	
5	469651	
6	469651	
7	469651	7
8	67093	
9	67093	
10	67093	
11	67093	
12	67093	
13	67093	13 13
14	397	
15	397	
16	397	
17	397	
18	397	
19	397	
20	397	
		397

Trace of `java Factors 3757208`

Prime Counting

计算到 n 之前有多少个质数

An algorithm for making tables of primes. Sequentially write down the integers from 2 to the highest number n you wish to include in the table. Cross out all numbers >2 which are divisible by 2 (every second number). Find the smallest remaining number >2. It is 3. So cross out all numbers >3 which are divisible by 3 (every third number). Find the smallest remaining number >3. It is 5. So cross out all numbers >5 which are divisible by 5 (every fifth number).

The sieve of Eratosthenes can be used to compute the prime counting function.

The prime counting function π(*n*) is the number of primes less than or equal to *n*. For example π(17) = 7 since the first seven primes are 2, 3, 5, 7, 11, 13, and 17.

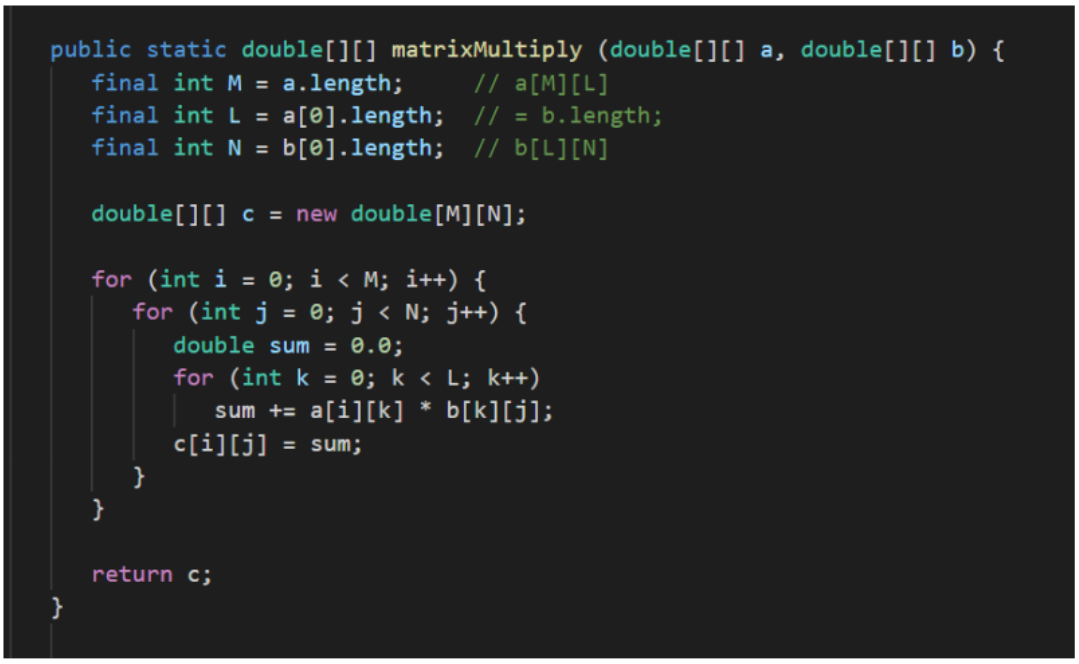
```java
public class PrimeSieve {
    public static void main(String[] args) {
        int n = Integer.parseInt(args[0]);

        // initially assume all integers are prime
        boolean[] isPrime = new boolean[n+1];
        for (int i = 2; i <= n; i++) {
            isPrime[i] = true;
        }

        // mark non-primes <= n using Sieve of Eratosthenes
        for (int factor = 2; factor*factor <= n; factor++) {

            // if factor is prime, then mark multiples of factor as nonprime
            // suffices to consider mutiples factor, factor+1, ...,  n/factor
            if (isPrime[factor]) {
                for (int j = factor; factor*j <= n; j++) {
                    isPrime[factor*j] = false;
                }
            }
        }

        // count primes
        int primes = 0;
        for (int i = 2; i <= n; i++) {
            if (isPrime[i]) primes++;
        }
        System.out.println("The number of primes <= " + n + " is " + primes);
    }
}
```

Matrix Multiplication

```java
double[][] c = new double[n][n];
for (int i = 0; i < n; i++) {
  for (int j = 0; j < n; j++)  {
    for (int k = 0; k < n; k++)  {
      c[i][j] += a[i][k]*b[k][j];
    }
  }
}
```

Increasing the locality for the inner loop operations 增加内部循环操作的局部性，so that the repeated access of `c[i][j]` could be substitute with local variable sum which might be optimized by assigning a register!

```
double[][] c = new double[n][n];
for (int i = 0; i < n; i++) {
  for (int j = 0; j < n; j++)  {
    double sum = 0.0;
    for (int k = 0; k < n; k++)  {
      sum += a[i][k]*b[k][j];
    }
    c[i][j] = sum;
  }
}
```

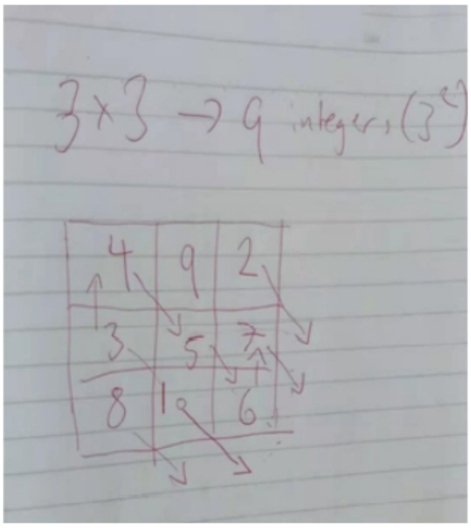


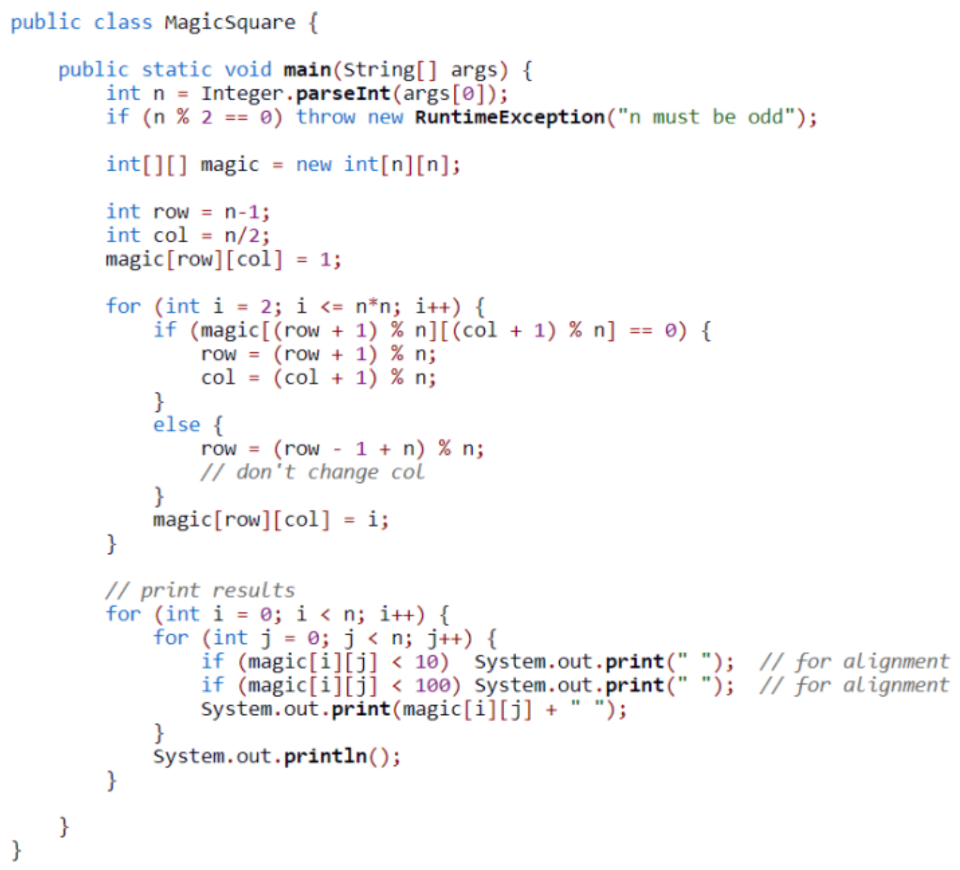
```
public static double[][] matrixMultiply (double[][] a, double[][] b) {
    final int M = a.length;     // a[M][L]
    final int L = a[0].length;  // = b.length;
    final int N = b[0].length;  // b[L][N]

    double[][] c = new double[M][N];

    for (int i = 0; i < M; i++) {
        for (int j = 0; j < N; j++) {
            double sum = 0.0;
            for (int k = 0; k < L; k++)
                sum += a[i][k] * b[k][j];
            c[i][j] = sum;
        }
    }

    return c;
}
```

Magic Square

A n by n magic square is an n x n matrix with the integer 1 .. n^2 exactly once, with the constraints that the rows, columns and diagonals all are equal.

One simple algorithm is to assign the integers 1 to n^2 in ascending order, starting at the bottom, middle cell. Repeatedly assign the next integer to the cell adjacent diagonally to the right and down. If this cell has already been assigned another integer, instead use the cell adjacently above. Use wrap-around to handle border cases.





```java
public class MagicSquare {

    public static void main(String[] args) {
        int n = Integer.parseInt(args[0]);
        if (n % 2 == 0) throw new RuntimeException("n must be odd");

        int[][] magic = new int[n][n];

        int row = n-1;
        int col = n/2;
        magic[row][col] = 1;

        for (int i = 2; i <= n*n; i++) {
            if (magic[(row + 1) % n][(col + 1) % n] == 0) {
                row = (row + 1) % n;
                col = (col + 1) % n;
            }
            else {
                row = (row - 1 + n) % n;
                // don't change col
            }
            magic[row][col] = i;
        }

        // print results
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (magic[i][j] < 10)  System.out.print(" ");  // for alignment
                if (magic[i][j] < 100) System.out.print(" ");  // for alignment
                System.out.print(magic[i][j] + " ");
            }
            System.out.println();
        }

    }
}
```