Find the sum of digits

Add remainders after each division.

```
public static int findSumOfDigitsOfNum(int num) {
   int rem = 0;
   int sum = 0;
   while (num > 0) {
     rem = num % 10;
     num = num / 10;
     sum = sum + rem;
   }
   return sum;
 }
```

Reverse String without using third variable

Iterate through half of elements till end and swap .

```
private static String reverseString(String str) {
   char[] array = str.toCharArray();
   for (int i = 0; i < array.length / 2; i++) {
     char temp = array[i];
     array[i] = array[array.length - i - 1];
     array[array.length - i - 1] = temp;
   }
   return new String(array);
}
```

Find all prime numbers till N.

Iterate till N . find all numbers which are divisible by it's previous numbers starting from 2.

```
public static ArrayList<Integer> getAllPrimeNumbers(int number) {
            ArrayList<Integer> primeNumbers = new ArrayList<>();
            boolean isPrime = true;
            for (int i = 1; i <= number; i++) {
               for (int j = 2; j < i; j++) {
                        if (i % j == 0) {
                           isPrime = false;
                           break;
                        }else{
                           isPrime = true;
                        }
               }
               if (isPrime) {
                        primeNumbers.add(i);
               }
            }
            return primeNumbers;
   }
```

Convert string to integer.

Iterate through each character, fetch it's integer value using ch-'0' and make sum of all integer values.

```
private static void stringToInt(String str) {
   char[] array = str.toCharArray();
   int num = 0;
   for (char ch : array) {
     num = ch - '0';
     num += num * 10;
   }
   System.out.print(num);
 }
```

Find a number using Linear Search

```
 public static boolean contains(int[] a, int b) {
        for (int i : a) {
           if (i == b) {
                return true;
           }
        }
        return false;
 }
```

Find a number using Binary Search.

Binary search requires that the collection is already sorted. For example by Quicksort or Mergesort. Binary search checks the element in the middle of the collection. If the search element smaller or greater then the found element then a sub-array is defined which is then search again. If the searched element is smaller then the found element then the sub-array is from the start of the array until the found element. If the searched element is larger then the found element then the sub-array is from the found element until the end of the array. Once the searched element is found or the collection is empty then the search is over.

```
  public static int binarySearch(int[] array, int num) {
        int low = 0;
        int high = array.length - 1;
        while (low <= high) {
           int mid = low + (high - low) / 2;
           int midElement = array[mid];
           if (num == midElement) {
                return mid;
           } else if (num < midElement) {
                high = mid - 1;
           } else {
                low = mid + 1;
           }
        }
        return -1;
  }

  public static int binarySearchWithRecursion(int[] array, int num, int low,
           int high) {
        int mid = low + (high - low) / 2;
        int midElement = array[mid];
        if (num == midElement) {
           return mid;
        } else if (num < midElement) {
           mid = binarySearchWithRecursion(array, num, low, mid - 1);
```

```
        } else {
            mid = binarySearchWithRecursion(array, num, mid + 1, high);
        }
        return mid;
    }
```

Quick Sort array in Java

Quicksort is a divide and conquer algorithm. It first divides a large list into two smaller sub-lists and then recursively sort the two sub-lists. If we want to sort an array without any extra space, quicksort is a good option. On average, time complexity is O(n log(n)).

The basic step of sorting an array are as follows:

- Select a pivot, normally the middle one
- From both ends, swap elements and make left elements < pivot and all right > pivot
- Recursively sort left part and right part

Here is a very good explanation of quicksort.

```
public static void quickSort(int[] arr, int low, int high) {
        if (low >= high || high > array.length  || array.length == 0) {
                return;
        }

        // pick the pivot
        int middle = low + (high - low) / 2;
        int pivot = arr[middle];

        // make left < pivot and right > pivot
        int i = low, j = high;
        while (i <= j) {
                while (arr[i] < pivot) {
                        i++;
                }

                while (arr[j] > pivot) {
                        j--;
                }

                if (i <= j) {
                        int temp = arr[i];
                        arr[i] = arr[j];
                        arr[j] = temp;
                        i++;
                        j--;
                }
        }

        // recursively sort two sub parts
        if (low < j)
                quickSort(arr, low, j);
        if (high > i)
                quickSort(arr, i, high);
    }

  call with quickSort(array, 0, size-1);
```

Bubble Sort

```
public static void bubbleSort(int[] array) {
   for (int i = 0; i < array.length - 1; i++) {
     for (int j = 0; j < array.length - i - 1; j++) {
       if (array[j + 1] < array[j]) {
        int temp = array[j];
        array[j] = array[j + 1];
        array[j + 1] = temp;
      }
     }
   }
 }
```

Two Sum

Given an array of integers, find two numbers such that they add up to a specific target number.

```
private static int[] twoSum(int[] array, int target) {
        int[] indexArray = new int[2];
        for (int i = 0; i < array.length; i++) {
           for (int j = 1; j < array.length - 1; j++) {
                if (array[i] + array[j] == target) {
                   indexArray[0] = array[i];
                   indexArray[1] = array[j];
                }
           }
        }
        return indexArray;
   }
```

Roman to Numeric conversion

When a letter of smaller value is followed by a letter of larger value, the smaller value is subtracted from the larger value. For example, IV represents 5 - 1, or 4. And MCMXCV is interpreted as M + CM + XC + V, or 1000 + (1000 - 100) + (100 - 10) + 5, which is 1995. In standard Roman numerals, no more than thee consecutive copies of the same letter are used. Following these rules, every number between 1 and 3999 can be represented as a Roman numeral made up of the following one- and two-letter combinations:

M	1000	X	10
CM	900	IX	9
D	500	V	5
CD	400	IV	4
C	100	I	1
XC	90		
L	50		
XL	40		

```
   private final static HashMap<Character, Integer> ROMAN_MAP = new HashMap<>();;

   public RomanToNumeric() {
        ROMAN_MAP.put('M', 1000);
        ROMAN_MAP.put('D', 500);
        ROMAN_MAP.put('C', 100);
        ROMAN_MAP.put('L', 50);
        ROMAN_MAP.put('X', 10);
        ROMAN_MAP.put('V', 5);
        ROMAN_MAP.put('I', 1);
        System.out.println("convertRomanToDecimal:"
                + convertToArabic("IIXVVIIIVXX") + "");
   }
```

```
  private static int getRomanNumeralValue(char ch) {
          if (ROMAN_MAP.containsKey(ch)) {
             return ROMAN_MAP.get(ch);
          } else {
             throw new RuntimeException(
                      "Roman numeral string contains invalid characters " + ch);
          }
  }

  private static int convertToArabic(String romanNumberString) {
          int romanNumberInt = 0;
          int lastIndex = romanNumberString.length() - 1;
          romanNumberInt = getRomanNumeralValue(romanNumberString
                  .charAt(lastIndex));
          for (int i = 0; i <= lastIndex - 1; i++) {
             if (getRomanNumeralValue(romanNumberString.charAt(i)) < getRomanNumeralValue(romanNumberString
                      .charAt(i + 1))) {
                  romanNumberInt -= getRomanNumeralValue(romanNumberString
                          .charAt(i));
             } else {
                  romanNumberInt += getRomanNumeralValue(romanNumberString
                          .charAt(i));
             }
          }
          return romanNumberInt;
  }
```

Stack :

```
public class Stack<E> {
  private int top = 0;
  private int stackSize = 0;
  private E[] elements;

  public Stack(int capacity) {
    stackSize = capacity;
    top = -1;
    elements = (E[]) new Object[stackSize];
  }

  public void push(E e) {
    if (top == stackSize) {
      throw new StackException("StackOverFlow. Stack is full");
    }
    elements[++top] = e;
  }

  public E pop() {
    if (top == -1) {
      throw new StackException("StackUnderFlow.Stack is empty");
    }
    return elements[top--];
  }
```

```
  public E peek() {
    if (top == -1) {
      throw new StackException("StackUnderFlow.Stack is empty");
    }
    return elements[top];
  }

  public boolean isEmpty() {
    return top == -1;
  }

  @Override
  public String toString() {
    StringBuffer stringBuffer = new StringBuffer("[");
    while (!isEmpty()) {
      stringBuffer.append(pop() + " ");
    }
    stringBuffer.append("]");
    return stringBuffer.toString();
  }

  private static final class StackException extends RuntimeException {

    private static final long serialVersionUID = 1L;

    private String exceptionEessage;

    private StackException(String message) {
      exceptionEessage = message;
    }

  }

}
```