STRING 1

```
public String atFirst(String str) {
  if(str.length() == 0) // Si la cadena está vacía
    return "@@"; // Retorna "@@"

  if(str.length() == 1) // Si la cadena tiene un solo carácter
    return str + "@"; // Retorna el carácter seguido de "@"

  return str.substring(0, 2); // Retorna los dos primeros caracteres
}
public String comboString(String a, String b) {
  if(b.length() < a.length()) { // Si b es más corta que a
    String temp = a; // Intercambia a y b
    a = b;
    b = temp;
  }

  return a + b + a; // Retorna la concatenación corta + larga + corta
}
public String conCat(String a, String b) {
  if(a.length() == 0 || b.length() == 0) // Si alguna cadena está vacía
    return a + b; // Retorna la concatenación simple

  if(a.charAt(a.length() - 1) == b.charAt(0)) // Si hay doble carácter
    return a + b.substring(1); // Omite el primer carácter de b

  return a + b; // Retorna la concatenación simple
}
public String deFront(String str) {
  if(str.length() == 1 && str.charAt(0) != 'a') // Si la cadena tiene un carácter y no es 'a'
    return ""; // Retorna una cadena vacía
```

```
    return false; // Retorna falso si no se cumple ninguna condición
}
public String twoChar(String str, int index) {
    if(index + 2 > str.length() || index < 0) // Si el índice es inválido
        return str.substring(0, 2); // Retorna los dos primeros caracteres

    return str.substring(index, index + 2); // Retorna la subcadena de longitud 2
}

public boolean bobThere(String str) {
    for(int i = 0; i < str.length() - 2; i++) { // Itera hasta la penúltima posición
        if(str.charAt(i) == 'b' && str.charAt(i + 2) == 'b') // Verifica si es "b*b"
            return true; // Retorna verdadero si encuentra "b*b"
    }

    return false; // Retorna falso si no encuentra "b*b"
}
public boolean catDog(String str) {
    int cat = 0;
    int dog = 0;

    for(int i = 0; i < str.length() - 2; i++) { // Itera hasta la antepenúltima posición
        if(str.substring(i, i + 3).equals("cat")) // Si encuentra "cat"
            cat++; // Incrementa el contador de gatos

        if(str.substring(i, i + 3).equals("dog")) // Si encuentra "dog"
            dog++; // Incrementa el contador de perros
    }

    return cat == dog; // Retorna si el número de gatos es igual al número de perros
```

```
        count++;
    }

    return new String(arr); // Retorna la nueva cadena con caracteres duplicados
}
public String mixString(String a, String b) {
    char[] arr;
    String end;
    int count = 0;

    if(a.length() < b.length()) { // Si a es más corta que b
        arr = new char[2 * a.length()]; // Crea un array con el doble de la longitud de a
        end = b.substring(a.length()); // Guarda los caracteres restantes de b
    } else { // Si b es más corta o igual que a
        arr = new char[2 * b.length()]; // Crea un array con el doble de la longitud de b
        end = a.substring(b.length()); // Guarda los caracteres restantes de a
    }

    for(int i = 0; i < arr.length / 2; i++) { // Itera sobre la longitud de la cadena más corta
        arr[count] = a.charAt(i); // Agrega el carácter de a al array
        count++;
        arr[count] = b.charAt(i); // Agrega el carácter de b al array
        count++;
    }

    return new String(arr) + end; // Retorna la cadena entrelazada más los caracteres
restantes
}
public String repeatEnd(String str, int n) {
    StringBuffer result = new StringBuffer();
    String end = str.substring(str.length() - n); // Obtiene los últimos n caracteres
```

```
}
public boolean equalIsNot(String str) {
  int is = 0; // Contador de "is"
  int not = 0; // Contador de "not"

  for(int i = 0; i <= str.length() - 3; i++) { // Itera hasta la antepenúltima posición
    if(str.substring(i, i + 2).equals("is")) { // Si encuentra "is"
      is++; // Incrementa el contador de "is"
    } else if(str.substring(i, i + 3).equals("not")) { // Si encuentra "not"
      not++; // Incrementa el contador de "not"
    }
  }

  if(str.length() >= 2 && str.substring(str.length() - 2).equals("is")) // Si la cadena termina
con "is"
    is++; // Incrementa el contador de "is"

  return is == not; // Retorna si el número de "is" es igual al número de "not"
}
public boolean gHappy(String str) {
  if(str.length() == 1 && str.charAt(0) == 'g') // Si la cadena tiene un solo carácter y es 'g'
    return false; // Retorna falso

  if(str.length() >= 2 && // Si la cadena tiene al menos dos caracteres
    (str.charAt(0) == 'g' && str.charAt(1) != 'g' || // y la primera 'g' no tiene otra 'g' a la
derecha
     str.charAt(str.length()-1) == 'g' && // o la última 'g' no tiene otra 'g' a la izquierda
     str.charAt(str.length()-2) != 'g'))
    return false; // Retorna falso

  for(int i = 1; i <= str.length() - 2; i++) { // Itera desde la segunda posición hasta la
penúltima
```

```
    if(string.charAt(i) == string.charAt(string.length() - i - 1)) // Si el carácter actual es igual al carácter correspondiente desde el final
      result.append(string.charAt(i)); // Agrega el carácter al resultado
    else
      break; // Si no son iguales, detiene la iteración
  }

  return result.toString(); // Retorna la subcadena especular
}
public String notReplace(String str) {
  if(str.equals("is")) // Si la cadena es "is"
    return "is not"; // Retorna "is not"

  StringBuilder result = new StringBuilder();
  int i = 0;

  if(str.length() >= 3 && str.substring(0,2).equals("is") && // Si la cadena comienza con "is"
    !Character.isLetter(str.charAt(2))) { // y no está seguida por una letra
    result.append("is not"); // Agrega "is not" al resultado
    i = 2; // Avanza el índice
  }

  while(i < str.length()) { // Itera sobre la cadena
    if(!Character.isLetter(str.charAt(i))) { // Si el carácter actual no es una letra
      result.append(str.charAt(i)); // Agrega el carácter al resultado
      i++; // Avanza el índice
    } else if(i >= 1 && i <= str.length()-3 && // Si el índice está dentro de los límites
        !Character.isLetter(str.charAt(i-1)) && // y el carácter anterior no es una letra
        str.substring(i,i+2).equals("is") && // y la subcadena actual es "is"
        !Character.isLetter(str.charAt(i+2))) { // y el carácter siguiente no es una letra
      result.append("is not"); // Agrega "is not" al resultado
```

```
  int sum = 0;

  for(int i = 0; i < str.length(); i++) { // Itera sobre la cadena
    if(Character.isDigit(str.charAt(i))) // Si el carácter actual es un dígito
      sum = sum + str.charAt(i) - '0'; // Agrega el valor del dígito a la suma
  }

  return sum; // Retorna la suma de los dígitos
}
```

ARRAYS

ARRAY 3:

```
public int maxSpan(int[] nums)
{
  // Considera las apariciones más a la izquierda y más a la derecha de un valor en un arreglo.
  // Diremos que el 'span' es el número de elementos entre los dos, inclusive.
  // Un solo valor tiene un 'span' de 1.
  // Devuelve el 'span' más grande encontrado en el arreglo dado. (La eficiencia no es una prioridad.)

  int maxSpan = 0; // Inicializamos maxSpan en 0, para almacenar el mayor 'span' encontrado.
  int span; // Variable para almacenar el valor de cada 'span'.
  int j; // Variable para recorrer el arreglo desde la derecha.

  // Recorremos el arreglo buscando, para cada valor, su última aparición a la derecha.
  for(int i = 0; i < nums.length; i++) {
    // Buscamos la última aparición del número en el arreglo (más a la derecha).
    for(j = nums.length - 1; nums[i] != nums[j]; j--);

    // Calculamos el 'span' sumando 1 a la diferencia entre las posiciones.
```

```
    }
  }

  // Devolvemos el arreglo modificado.
  return nums;
}
public int[] fix45(int[] nums)
{
  // (Esta es una versión ligeramente más difícil del problema `fix34`).
  // Devuelve un arreglo que contiene exactamente los mismos números que el arreglo dado,
  // pero reorganizados de modo que cada 4 esté inmediatamente seguido de un 5.
  // No muevas los 4, pero cualquier otro número puede moverse.

  int j = 0; // Variable para buscar las posiciones de los 5 en el arreglo.

  // Recorremos el arreglo hasta el penúltimo elemento.
  for(int i = 0; i < nums.length - 1; i++) {
    // Si encontramos un 4 y el siguiente número no es 5, lo reordenamos.
    if(nums[i] == 4 && nums[i+1] != 5) {
      // Buscamos el siguiente 5 en el arreglo, asegurándonos de que no esté justo después de un 4.
      for(; !(nums[j] == 5 && (j == 0 || j > 0 && nums[j-1] != 4)); j++);

      // Colocamos el número que estaba después del 4 en la posición del 5.
      nums[j] = nums[i+1];

      // Colocamos el 5 en la posición donde estaba el número siguiente al 4.
      nums[i+1] = 5;
    }
  }
```

```
  // Dado dos arreglos de enteros ordenados de manera creciente, 'outer' e 'inner',
  // devuelve `true` si todos los números en 'inner' aparecen en 'outer'.
  // La mejor solución hace solo una pasada "lineal" de ambos arreglos,
  // aprovechando que ambos arreglos ya están ordenados.

  boolean notFound; // Variable para verificar si el número en 'inner' fue encontrado en 'outer'.

  // Recorremos el arreglo 'inner' para buscar cada número en el arreglo 'outer'.
  for(int inI = 0, outI = 0; inI < inner.length; inI++) {
    notFound = true; // Inicializamos 'notFound' en 'true' al inicio de cada búsqueda.

    // Buscamos el número de 'inner' en 'outer' en una sola pasada.
    for(; outI < outer.length && notFound; outI++) {
      if(inner[inI] == outer[outI]) // Si encontramos el número, lo marcamos como encontrado.
        notFound = false;
    }

    // Si no encontramos el número en 'outer', devolvemos 'false'.
    if(notFound)
      return false;
  }

  // Si todos los números en 'inner' fueron encontrados en 'outer', devolvemos 'true'.
  return true;
}
public int[] squareUp(int n)
{
  // Dado n >= 0, crea un arreglo de longitud n*n con el siguiente patrón,
  // mostrado aquí para n=3: {0, 0, 1,  0, 2, 1,  3, 2, 1}
  // (espacios añadidos para mostrar los 3 grupos).
```

```
}
public int maxMirror(int[] nums)
{
  // Decimos que una "sección espejo" en un arreglo es un grupo de elementos contiguos
  // tal que en algún lugar del arreglo, el mismo grupo aparece en orden inverso.
  // Por ejemplo, la mayor sección espejo en {1, 2, 3, 8, 9, 3, 2, 1} es de longitud 3
  // (la parte {1, 2, 3}). Devuelve el tamaño de la mayor sección espejo encontrada en el arreglo dado.

  int span; // Variable para almacenar la longitud de la sección espejo encontrada.
  int maxSpan = 0; // Variable para almacenar la longitud máxima de la sección espejo.
  int left; // Variable para la posición inicial de la sección espejo.
  int right; // Variable para la posición final de la sección espejo.

  // Recorremos el arreglo buscando secciones espejo.
  for(int i = 0; i < nums.length; i++) {
    left = i; // Establecemos la posición izquierda de la sección espejo.
    right = lastIndexOf(nums, nums[i], nums.length - 1); // Buscamos la última aparición de un número.

    while(right != -1) {
      span = 0; // Inicializamos el tamaño de la sección espejo.
      left = i; // Reiniciamos la posición izquierda.

      // Comparamos los elementos de izquierda a derecha y de derecha a izquierda.
      do {
        left++;
        right--;
        span++; // Incrementamos el tamaño de la sección espejo.
      } while(left < nums.length && right >= 0 && nums[left] == nums[right]);

      // Si encontramos una sección espejo mayor, la actualizamos.
```