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

if(str.length() >= 2) { // Si la cadena tiene al menos dos caracteres

if(str.charAt(0) != 'a' && str.charAt(1) != 'b') { // Si ni el primero es 'a' ni el segundo es 'b'

return str.substring(2); // Retorna la cadena desde el tercer carácter

} else if(str.charAt(0) != 'a') { // Si el primero no es 'a'

return str.substring(1); // Retorna la cadena desde el segundo carácter

} else if(str.charAt(1) != 'b') { // Si el segundo no es 'b'

return "a" + str.substring(2); // Retorna 'a' seguido de la cadena desde el tercer carácter

}

}

return str; // Retorna la cadena original

}

public String firstHalf(String str) {

return str.substring(0, str.length() / 2); // Retorna la primera mitad de la cadena

}

public String firstTwo(String str) {

if(str.length() < 2) // Si la cadena tiene menos de dos caracteres

return str; // Retorna la cadena original

return str.substring(0, 2); // Retorna los dos primeros caracteres

}

public boolean hasBad(String str) {

if(str.length() == 3) // Si la cadena tiene longitud 3

return str.substring(0, 3).equals("bad"); // Retorna si es igual a "bad"

if(str.length() >= 4) // Si la cadena tiene al menos 4 caracteres

return str.substring(0, 3).equals("bad") || // Retorna si "bad" está en el índice 0

str.substring(1, 4).equals("bad"); // o en el índice 1

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

}

public int countCode(String str) {

int count = 0;

for(int i = 0; i < str.length() - 3; i++) { // Itera hasta la antepenúltima posición

if(str.substring(i, i + 2).equals("co") && str.charAt(i + 3) == 'e') // Si encuentra "co\*e"

count++; // Incrementa el contador

}

return count; // Retorna el número de veces que encontró "co\*e"

}

public int countHi(String str) {

int count = 0;

for(int i = 0; i < str.length() - 1; i++) { // Itera hasta la penúltima posición

if(str.substring(i, i + 2).equals("hi")) // Si encuentra "hi"

count++; // Incrementa el contador

}

return count; // Retorna el número de veces que encontró "hi"

}

public String doubleChar(String str) {

char[] arr = new char[2 \* str.length()]; // Crea un array de caracteres con el doble de longitud

int count = 0;

for(int i = 0; i < str.length(); i++) { // Itera sobre la cadena original

arr[count] = str.charAt(i); // Agrega el carácter al array

count++;

arr[count] = str.charAt(i); // Agrega el carácter duplicado al array

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

for(int i = 0; i < n; i++) // Itera n veces

result.append(end); // Agrega los últimos n caracteres al resultado

return result.toString(); // Retorna la cadena resultante

}

public String wordEnds(String str, String word) {

StringBuffer result = new StringBuffer();

int i = 0;

if(str.length() >= word.length() + 1 && // Si la cadena es lo suficientemente larga

str.substring(0, word.length()).equals(word)) { // y la palabra está al principio

i = word.length() - 1; // Ajusta el índice

result.append(str.charAt(i + 1)); // Agrega el carácter siguiente

}

while(i < str.length() - word.length()) { // Itera sobre la cadena

if(str.substring(i + 1, i + 1 + word.length()).equals(word)) { // Si encuentra la palabra

result.append(str.charAt(i)); // Agrega el carácter anterior

i = i + word.length(); // Avanza el índice

if(i < str.length() - 1) { // Si no está al final

result.append(str.charAt(i + 1)); // Agrega el carácter siguiente

}

} else {

i++; // Avanza el índice si no encuentra la palabra

}

}

return result.toString(); // Retorna la cadena resultante

}

STRING 3

public int countTriple(String str) {

int count = 0;

for(int i = 0; i <= str.length() - 3; i++) { // Itera hasta la antepenúltima posición

if(str.charAt(i) == str.charAt(i+1) && // Si el carácter actual es igual al siguiente

str.charAt(i) == str.charAt(i+2)) // y es igual al siguiente del siguiente

count++; // Incrementa el contador

}

return count; // Retorna el número de triples

}

public int countYZ(String str) {

if(str.length() == 0) // Si la cadena está vacía

return 0; // Retorna 0

int count = 0;

for(int i = 0; i <= str.length() - 2; i++) { // Itera hasta la penúltima posición

if((Character.toLowerCase(str.charAt(i)) == 'y' || // Si el carácter actual es 'y' o 'z' (en minúscula)

Character.toLowerCase(str.charAt(i)) == 'z') &&

!Character.isLetter(str.charAt(i+1))) // y el siguiente carácter no es una letra

count++; // Incrementa el contador

}

if(Character.toLowerCase(str.charAt(str.length() - 1)) == 'y' || // Si el último carácter es 'y' o 'z' (en minúscula)

Character.toLowerCase(str.charAt(str.length() - 1)) == 'z')

count++; // Incrementa el contador

return count; // Retorna el número de palabras que terminan en 'y' o 'z'

}

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(str.charAt(i) == 'g' && str.charAt(i-1) != 'g' && // Si el carácter actual es 'g' y no tiene otra 'g' a la izquierda

str.charAt(i+1) != 'g') // ni a la derecha

return false; // Retorna falso

}

return true; // Retorna verdadero si todas las 'g' son felices

}

public int maxBlock(String str) {

if(str.length() == 0) // Si la cadena está vacía

return 0; // Retorna 0

int largest = 0; // Longitud del bloque más grande encontrado

int current = 1; // Longitud del bloque actual

for(int i = 1; i < str.length(); i++) { // Itera sobre la cadena

if(str.charAt(i) != str.charAt(i-1)) { // Si el carácter actual es diferente al anterior

if(current > largest) // Si la longitud del bloque actual es mayor que la del bloque más grande

largest = current; // Actualiza la longitud del bloque más grande

current = 1; // Reinicia la longitud del bloque actual

} else { // Si el carácter actual es igual al anterior

current++; // Incrementa la longitud del bloque actual

}

}

return Math.max(largest, current); // Retorna la longitud del bloque más grande

}

public String mirrorEnds(String string) {

StringBuilder result = new StringBuilder();

for(int i = 0; i < string.length(); i++) { // Itera sobre la cadena

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

i += 2; // Avanza el índice

} else if(i >= 1 && !Character.isLetter(str.charAt(i-1)) && // Si el índice es válido y el carácter anterior no es una letra

str.substring(i).equals("is")) { // y la subcadena actual es "is"

result.append("is not"); // Agrega "is not" al resultado

i += 2; // Avanza el índice

} else { // Si no se cumple ninguna de las condiciones anteriores

result.append(str.charAt(i)); // Agrega el carácter actual al resultado

i++; // Avanza el índice

}

}

return result.toString(); // Retorna la cadena resultante

}

public String sameEnds(String string) {

int start = (int) Math.ceil((double) string.length() / 2); // Calcula el punto de inicio desde la mitad de la cadena

int end = string.length() / 2; // Calcula el punto final hasta la mitad de la cadena

for(int i = 0; i < string.length() / 2; i++) { // Itera hasta la mitad de la cadena

if(string.substring(0, end).equals(string.substring(start))) { // Si la subcadena al principio es igual a la subcadena al final

return string.substring(0, end); // Retorna la subcadena

} else { // Si no son iguales

start++; // Mueve el punto de inicio hacia adelante

end--; // Mueve el punto final hacia atrás

}

}

return ""; // Si no encuentra ninguna subcadena, retorna una cadena vacía

}

public int sumDigits(String str) {

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.

span = 1 + j - i;

// Si el 'span' es mayor que el valor actual de maxSpan, lo actualizamos.

if(span > maxSpan)

maxSpan = span;

}

// Devolvemos el 'span' máximo encontrado.

return maxSpan;

}ç

public int[] fix34(int[] nums)

{

// Devuelve un arreglo que contiene exactamente los mismos números que el arreglo dado,

// pero reorganizados de modo que cada 3 esté inmediatamente seguido de un 4.

// No muevas los 3, pero cualquier otro número puede moverse.

int j = 1; // Variable para buscar la posición de los 4 en el arreglo.

// Recorremos el arreglo hasta el penúltimo elemento.

for(int i = 0; i < nums.length - 1; i++) {

// Si encontramos un 3 y el siguiente número no es 4, lo reordenamos.

if(nums[i] == 3 && nums[i+1] != 4) {

// Buscamos el siguiente 4 en el arreglo.

for(; nums[j] != 4; j++);

// Colocamos el número que estaba después del 3 en la posición del 4.

nums[j] = nums[i+1];

// Colocamos el 4 en la posición donde estaba el número siguiente al 3.

nums[i+1] = 4;

}

}

// 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;

}

}

// Devolvemos el arreglo modificado.

return nums;

}

public boolean canBalance(int[] nums)

{

// Devuelve `true` si existe un lugar para dividir el arreglo de modo que la suma

// de los números de un lado sea igual a la suma de los números del otro.

int left = 0; // Inicializamos la suma del lado izquierdo.

int right; // Inicializamos la suma del lado derecho.

// Sumamos todos los elementos al lado izquierdo, excepto el último.

for(int i = 0; i < nums.length - 1; i++)

left += nums[i];

right = nums[nums.length-1]; // Inicializamos el lado derecho con el último elemento.

// Recorremos el arreglo desde el penúltimo elemento y vamos ajustando las sumas.

for(int i = nums.length - 2; i > 0; i--) {

if(left == right) // Si las sumas son iguales, hemos encontrado el punto de balance.

return true;

left -= nums[i]; // Restamos el valor del lado izquierdo.

right += nums[i]; // Sumamos el valor al lado derecho.

}

// Comprobamos si las sumas son iguales al final.

return (left == right);

}

public boolean linearIn(int[] outer, int[] inner)

{

// 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).

int[] arr = new int[n\*n]; // Creamos un arreglo de longitud n\*n.

int p; // Variable para manejar la posición de los elementos en el arreglo.

// Llenamos el arreglo con el patrón.

for(int i = 1; i <= n; i++) {

p = n \* i - 1; // Calculamos la posición inicial para cada fila.

for(int j = 1; j <= i; j++, p--) // Llenamos cada fila en el arreglo.

arr[p] = j;

}

// Devolvemos el arreglo con el patrón.

return arr;

}

public int[] seriesUp(int n)

{

// Dado n >= 0, crea un arreglo con el patrón {1, 1, 2, 1, 2, 3, ... 1, 2, 3 .. n}.

// Nota que la longitud del arreglo será 1 + 2 + 3 ... + n,

// lo cual se sabe que suma exactamente n\*(n + 1)/2.

int[] arr = new int[n\*(n+1)/2]; // Creamos el arreglo con el tamaño adecuado.

int p = 0; // Variable para manejar la posición de los elementos en el arreglo.

// Llenamos el arreglo con los valores en el patrón.

for(int i = 1; i <= n; i++) {

for(int j = 1; j <= i; j++, p++) // Colocamos los números en secuencia.

arr[p] = j;

}

// Devolvemos el arreglo con el patrón.

return arr;

}

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.

if(span > maxSpan)

maxSpan = span;

// Buscamos la siguiente sección espejo.

right = lastIndexOf(nums, nums[i], right);

}

}

// Devolvemos la longitud máxima de la sección espejo encontrada.

return maxSpan;

}