EJERCICIOS JAVA

HandBook I (1-25)

Contenido

1. Two Sum 49.2% Easy

Dado un array de enteros ***nums*** y un entero ***target***, devuelva los índices de los dos números tales que sumen ***target***.

Puede suponer que cada entrada tendría ***exactamente una solución,*** y no puede utilizar el mismo elemento dos veces.

Puede devolver la respuesta en cualquier orden.

Enfoque

1. El método ***twoSum*** debe devolver un array de enteros con los índices de dos números en el array original que suman el entero dado.
2. Dentro del método twoSum, se crea un ***map*** vacío llamado *map* para almacenar los números y sus índices.
3. Se itera sobre el array de números y para cada número, se calcula complemento, el ***número necesario*** para alcanzar el ***objetivo*** dado.
4. Si el complemento está en el ***map***, entonces se han encontrado los dos números necesarios. Devolvemos un array con los índices de cada número.
5. Si el complemento no está en el ***map***, añadimos el número actual al ***map*** para futuras referencias.
6. Si el número objetivo no se encuentra, la función lanza una excepción.

```
public class Solution {
    public int[] twoSum(int[] nums, int target) {
        // Creamos un mapa para almacenar cada número y su índice
correspondiente
        Map<Integer, Integer> map = new HashMap<>();
        // Iteramos sobre el arreglo de números
        for (int i = 0; i < nums.length; i++) {
            // Calculamos el complemento del número actual, es decir,
el número que necesitamos para alcanzar el objetivo
            int complemento = target - nums[i];
            // Si el complemento está en el mapa, entonces hemos
encontrado los dos números que necesitamos
            if (map.containsKey(complemento)) {
                // Devolvemos un arreglo con los índices de cada
número
                return new int[] { map.get(complemento), i };
            }
            // Si el complemento no está en el mapa, entonces añadimos
el número actual al mapa para futuras referencias
            map.put(nums[i], i);
        }
        // Si no encontramos los números, entonces lanzamos una
excepción
        throw new IllegalArgumentException("No se ha encontrado
solución");
    }
}
```

2. Add Two Numbers 40.0% Medium

Se le dan dos listas enlazadas ***no vacías*** que representan dos números enteros no negativos. Los dígitos se almacenan en ***orden inverso***, y cada uno de sus nodos contiene un solo dígito. Suma los dos números y devuelve la suma como una lista enlazada.

Puede suponer que los dos números no contienen ningún cero a la izquierda, excepto el propio número 0.

Enfoque

La función addTwoNumbers tiene como parámetros de entrada ***dos listas enlazadas***, *l1* y *l2*, y devuelve una nueva lista enlazada que es la suma de las dos entradas(invertidas). Los pasos de implementaciónson los siguientes:

1. Se comprueba si l1 o l2 son nulos. Si alguno de ellos es nulo, se devuelve ***null***.
2. Se crea una nueva lista enlazada llamada ***res*** y un ***nodo actual*** que apunta a ***res***. Se inicializa una variable de ***arrastre*** en 0.
3. Se aplica un bucle *while* que itera mientras l1 o l2 no sean nulos:
    - Se suman los valores de los nodos actuales de l1 y l2, así como el valor de ***arrastre***.
    - Se actualiza el valor de ***arrastre*** a la división entera de la ***suma*** por 10.
    - Se crea un ***nuevo nodo*** con el valor de la suma mod 10 y se añade al final de la lista enlazada resultante.
    - Se actualiza el ***nodo actua***l para que apunte al ***nuevo nodo***.
    - Se avanza al siguiente nodo de l1 y l2, si es que existen.
4. Si el valor de arrastre es mayor que 0, se crea un nuevo nodo con el valor de arrastre y se añade al final de la lista enlazada resultante.
5. Se devuelve la lista enlazada ***resultante***, excluyendo el nodo ficticio inicial.

```java
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode() {}
 *     ListNode(int val) { this.val = val; }
 *     ListNode(int val, ListNode next) { this.val = val; this.next =
next; }
 * }
 */

class Solution {
    public  ListNode addTwoNumbers(ListNode l1, ListNode l2) {
        if (l1 == null | l2 == null) return null; // Manejo de entrada
nula
        ListNode res = new ListNode(0); // lista enlazada resultante
        ListNode actual = res; // nodo actual en la lista enlazada
resultante
        int arrastre = 0; // arrastre
        while (l1 != null || l2 != null) {
            // suma los valores de los nodos actuales y el valor de
arrastre
            int suma = arrastre;
            if (l1 != null) suma += l1.val;
            if (l2 != null) suma += l2.val;
            // actualización del valor de arrastre y el nodo actual en
la lista enlazada resultante
            arrastre = suma / 10;
            actual.next = new ListNode(suma % 10);
            actual = actual.next;
            // pasar al siguiente nodo de la lista enlazada de entrada
            if (l1 != null) l1 = l1.next;
            if (l2 != null) l2 = l2.next;
        }
        // manejar cualquier valor de arrastre restante
        if (arrastre > 0) actual.next = new ListNode(arrastre);
        return res.next; // devuelve la lista enlazada resultante sin
el nodo ficticio
    }
}
```

3. Longest Substring Without Repeating Characters 33.8% Medium

Dada una cadena ***s***, hallar la longitud de la ***subcadena más larga*** más larga sin repetir caracteres.

Enfoque

1. Comprobar las entradas de la cadena que no sea nula ni de longitud 0.
2. Utilizar una estructura de datos ***set*** para rastrear la ***subcadena actual*** y ***dos punteros*** (*l* y *r*) para delimitar la subcadena. Cada vez que se agrega un nuevo carácter a la ***subcadena actual*** (es decir, se agrega a ***set***), se aumenta *r*.
3. Si se encuentra que el carácter ya está en el conjunto, se elimina el primer carácter de la subcadena (*l* se aumenta en 1) y se sigue intentando agregar caracteres hasta que no se encuentren caracteres repetidos.
4. La longitud de la ***subcadena*** más larga se actualiza cada vez que se encuentra una ***subcadena*** más larga sin caracteres repetidos.
5. Al final el bucle, devuelve la longitud de la ***subcadena*** más larga.

```
class Solution {

    int lengthOfLongestSubstring(String s) {
        if (s == null || s.length() == 0) return 0; // manejar entrada
inválida
        Set<Character> set = new HashSet<>(); // set para rastrear la
subcadena actual
        int l = 0, r = 0, maxLength = 0; // límites y longitud de la
subcadena más larga
        while (r < s.length()) {
            if (set.add(s.charAt(r))) {
                r++;
            } else {
                set.remove(s.charAt(l));
                l++;
            }
            // actualiza la longitud de la subcadena más larga
            maxLength = Math.max(maxLength, r - l);
        }
        return maxLength;
    }
}
```

4. Median of Two Sorted Arrays 35.5% Hard

Dadas dos matrices ordenadas **nums1** y **nums2** de tamaño **m** y **n** respectivamente, devuelve la mediana de las dos matrices ordenadas.

La complejidad global del tiempo de ejecución debe ser *O(log (m+n))*.

Enfoque

1. Declarar array ***mix***. m y n son las longitudes de los arrays de entrada nums1 y nums2, respectivamente. ***mix*** es un array que combina los elementos de nums1 y nums2 en orden.
2. Iniciar un bucle que continúa mientras *i* sea menor que *m* y *j* sea menor que *n*.
3. Comparar nums1[i] y nums2[j]. Si nums1[i] es menor, agregar nums1[i] a mix[k] y aumentar *i* en 1. Si no, agregar nums2[j] a mix[k] y aumentar *j* en 1. Aumentar *k* en 1 al final del bloque.
4. Segundo bucle que continúa mientras *i* sea menor que *m*. Agrega cada elemento restante de nums1 a ***mix*** y aumenta *i* y *k* en 1 en cada iteración.
5. Tercer bucle que continúa mientras *j* sea menor que *n*. Agrega cada elemento restante de nums2 a ***mix*** y aumenta *j* y *k* en 1 en cada iteración.
6. Verificación de si la longitud de ***mix*** es par o impar. Si es par, se devuelve la media aritmética de mix[mix.length/2] y mix[mix.length/2-1]. Si es impar, se devuelve mix[mix.length/2].

```java
class Solution {
    public double findMedianSortedArrays(int[] nums1, int[] nums2) {
        int m = nums1.length;
        int n = nums2.length;
        int[] mix = new int[m+n];

        int i = 0, j = 0, k = 0;

        while (i < m && j < n) {
            if (nums1[i] < nums2[j]) {
                mix[k] = nums1[i];
                i++;
            } else {
                mix[k] = nums2[j];
                j++;
            }
            k++;
        }
        // Suma los elementos restantes de nums1
        while (i < m) {
            mix[k] = nums1[i];
            i++;
            k++;
        }
        // Suma los elementos restantes de nums2
        while (j < n) {
            mix[k] = nums2[j];
            j++;
            k++;
        }
        if (mix.length % 2 == 0) {
            // Número par de elementos
            return (mix[mix.length/2] + mix[mix.length/2-1]) / 2.0;
        } else {
            // Número impar de elementos
            return mix[mix.length/2];
        }
    }
}
```

5. Longest Palindromic Substring 32.4% Medium

Dada una cadena ***s***, devuelve la subcadena palindrómica más larga en ***s***.

Enfoque

1. La función primero verifica si la cadena de entrada es **null** o tiene una longitud de 0.
2. Si la cadena de entrada no está vacía, se declara una variable **n** igual a la longitud de la cadena de entrada y se crea una matriz booleana de dos dimensiones **check** con dimensiones **n** por **n**.
3. Se crea un bucle anidado, con el bucle externo iterando sobre los índices de **s** en orden inverso, y el bucle interno iterando sobre los índices de **s** a partir del índice del bucle externo. Para cada iteración del bucle interno, se establece el valor de **check[i][j]** en **true** si los caracteres en los índices **i** y **j** de **s** son iguales y **j - i < 3** o **check[i + 1][j - 1]** es **true**.
4. Después de establecer el valor de **check[i][j]**, la función debe verificar si **check[i][j]** es **true** y la longitud de la subcadena definida por **i** y **j** es mayor que la longitud de la cadena actual **resultado**. Si ambas condiciones son verdaderas, la función establecerá que **resultado** igual a la subcadena definida por **i** y **j**.
5. Finalmente, la función devuelve el valor de **resultado**.

```
class Solution {
    public String longestPalindrome(String s) {
        if (s == null || s.length() == 0) {
            return "";
        }
        int n = s.length();
        boolean[][] check = new boolean[n][n];
        String resultado = "";
        for (int i = n - 1; i >= 0; i--) {
            for (int j = i; j < n; j++) {
                check[i][j] = (s.charAt(i) == s.charAt(j)) && (j - i <
3 || check[i + 1][j - 1]);

                if (check[i][j] && (j - i + 1 > resultado.length())) {
                    resultado = s.substring(i, j + 1);
                }
            }
        }
        return resultado;
    }
}
```

6. Zigzag Conversion 43.4% Medium

La cadena "***PAYPALISHIRING***" está escrita en un patrón de ***zigzag*** en un *número determinado de filas* como esta: (es posible que desee mostrar este patrón en una fuente fija para una mejor legibilidad).

Enfoque

1. Si el número de filas es 1, devolver la cadena sin ningún cambios.
2. Crear una objeto de **StringBuilder** para construir la cadena resultante.
3. Calcular el paso utilizado para ir a la siguiente fila en la disposición de Z.
4. Iterar a través de cada fila de la matriz.
5. Si estamos en la primera o última fila, itera a través de la cadena, saltando cada **paso** caracteres y agregando cada carácter al **StringBuilder**.
6. Si no estamos en la primera o última fila, calcular el paso de vuelta utilizado para ir a la fila anterior y iterar a través de la cadena, saltando cada **paso** caracteres y agregando cada carácter y el carácter de vuelta al **StringBuilder** si está dentro de los límites de la cadena.
7. Dar la cadena resultante.

```java
class Solution {

    public String convert(String s, int numRows) {
        // Si numRows es 1, devolvemos la cadena sin cambios
        if (numRows == 1) return s;
        // Utilizamos un StringBuilder para construir la cadena
resultante
        StringBuilder sb = new StringBuilder();
        // Calculamos el paso utilizado para ir a la siguiente fila
        int paso = 2 * numRows - 2;
        // Iteramos a través de cada fila de la matriz
        for (int i = 0; i < numRows; i++) {
            // Si estamos en la primera o última fila, el patrón es
diferente
            if (i == 0 || i == numRows - 1) {
                // Iteramos a través de la cadena, saltando cada paso
caracteres
                for (int j = i; j < s.length(); j += paso) {
                    // Añadimos el carácter a nuestro StringBuilder
                    sb.append(s.charAt(j));
                }
            } else {
                // Si no estamos en la primera o última fila, el
patrón es diferente
                // Calculamos el paso de vuelta utilizado para ir a la
fila anterior
                int backStep = paso - 2 * i;
                // Iteramos a través de la cadena, saltando cada paso
caracteres
                for (int j = i; j < s.length(); j += paso) {
                    // Añadimos el carácter a nuestro StringBuilder
                    sb.append(s.charAt(j));
                    // Añadimos el carácter de vuelta si está dentro
de los límites de la cadena
                    if (j + backStep < s.length()) {
                        sb.append(s.charAt(j + backStep));
                    }
                }
            }
        }
        // Devolvemos la cadena resultante
        return sb.toString();
    }
}
```

7. Reverse Integer 27.3% Medium

Dado un entero de 32 bits con signo ***x***, devuelve ***x*** con sus dígitos invertidos. Si la inversión de ***x*** hace que el valor salga del rango de enteros de *32 bits con signo [-231, 231 - 1]*, entonces devuelve 0.

Se supone que el entorno no permite almacenar enteros de 64 bits (con o sin signo).

Enfoque

1. Bucle: Mientras **x** sea diferente de 0, realiza los siguientes pasos:
2. Calcular el último dígito de **x** utilizando el módulo 10.
3. Dividir **x** entero por 10 utilizando la división de enteros.
4. Verificar si **resultado** es mayor que el valor máximo permitido para un entero en Java (**Integer.MAX_VALUE**) dividido por 10 o si es igual a ese valor y el último dígito es mayor que 7. Si alguna condición se cumple, devolver 0.
5. Verificar si **resultado** es menor que el valor mínimo permitido para un entero en Java (**Integer.MIN_VALUE**) dividido por 10 o si es igual a ese valor y el último dígito es menor que -8. Si cualquiera de estas condiciones se cumple, devolver 0.
6. Actualizar el valor de **resultado** multiplicándolo por 10 y agregando el último dígito.
7. Devolución del **resultado**.

```
class Solution {
    public int reverse(int x) {
        int resultado = 0;
        while (x != 0) {
            int ultimoDigito = x % 10;
            x /= 10;
            if (resultado > Integer.MAX_VALUE/10 || resultado ==
Integer.MAX_VALUE/10 && ultimoDigito > 7) return 0;
            if (resultado < Integer.MIN_VALUE/10 || resultado ==
Integer.MIN_VALUE/10 && ultimoDigito < -8) return 0;
            resultado = resultado * 10 + ultimoDigito;
        }
        return resultado;
    }
}
```

8. String to Integer (atoi) 16.6% Medium

Implemente la función myAtoi(string s), que convierte una ***cadena*** en un ***entero*** con ***signo*** de 32 bits (similar a la función atoi de C/C++). El algoritmo para myAtoi(string s) es el siguiente:

Lea e ignore cualquier espacio en blanco inicial.

Compruebe si el siguiente carácter (si no está ya al final de la cadena) es '-' o '+'. Lea este personaje si lo es. Esto determina si el resultado final es negativo o positivo respectivamente. Suponga que el resultado es positivo si ninguno está presente.

Lea los siguientes caracteres hasta que se alcance el siguiente carácter que no sea un dígito o el final de la entrada. El resto de la cadena se ignora.

Convierta estos dígitos en un número entero (es decir, "123" -> 123, "0032" -> 32). Si no se leyeron dígitos, entonces el número entero es 0. Cambie el signo según sea necesario (desde el paso 2).

Si el entero está fuera del rango de enteros con signo de 32 bits [-231, 231 - 1], sujete el entero para que permanezca en el rango. Específicamente, los números enteros inferiores a -231 deben fijarse a -231, y los enteros superiores a 231 - 1 deben fijarse a 231 - 1.

Devuelve el entero como resultado final.

Nota:

Solo el carácter de espacio ' ' se considera un carácter de espacio en blanco.

No ignore ningún carácter que no sea el espacio en blanco inicial o el resto de la cadena después de los dígitos.

Enfoque

1. Eliminar los espacios en blanco al principio y al final de la cadena utilizando el método **strip()**.
2. Si la cadena de entrada es vacía, devolver 0.
3. Si el primer carácter de la cadena es '+' o '-', determinar el signo de la conversión y eliminar ese carácter de la cadena. Si el primer carácter es diferente a '+' o '-', se asume que el signo es positivo.
4. Iterar a través de cada carácter de la cadena. Si el carácter no es un dígito, termina el bucle. Si el carácter es un dígito, se agrega a **num** multiplicando **num** por 10 y sumando el valor del carácter menos '0'.
5. Verificar si el signo de **num** multiplicado por el valor actual de **num** es menor o igual a el valor mínimo permitido para un entero en Java (**Integer.MIN_VALUE**). Si es así, devolver **Integer.MIN_VALUE**.
6. Verificar si el signo de **num** multiplicado por el valor actual de **num** es mayor o igual a el valor máximo permitido para un entero en Java (**Integer.MAX_VALUE**). Si es así, devolver **Integer.MAX_VALUE**.
7. Devolver el signo de **num** multiplicado por el valor de **num** convertido a entero.

```java
class Solution {

    public int myAtoi(String s) {
        s = s.strip();
        if (s.isEmpty())
            return 0;

        final int signo = s.charAt(0) == '-' ? -1 : 1;
        if (s.charAt(0) == '+' || s.charAt(0) == '-')
            s = s.substring(1);

        long num = 0;

        for (final char c : s.toCharArray()) {
            if (!Character.isDigit(c))
                break;
            num = num * 10 + (c - '0');
            if (signo * num <= Integer.MIN_VALUE)
                return Integer.MIN_VALUE;
            if (signo * num >= Integer.MAX_VALUE)
                return Integer.MAX_VALUE;
        }
        return signo * (int) num;
    }
}
```

9. Palindrome Number 53.1% Easy

Dado un entero ***x***, devuelve verdadero si ***x*** es un palíndromo, y falso en caso contrario.

Enfoque

1. Convertir la entrada *int* **x** en una cadena utilizando el método **String.valueOf(x)**.
2. Crear una instancia de **StringBuilder** a partir de la cadena obtenida e invertir utilizando el método **reverse()**.
3. Convertir la cadena invertida de nuevo en una cadena utilizando el método **toString()**.
4. Comparación de la cadena original con la cadena invertida utilizando el método **equals()**.
5. Si son iguales, devuelve **true**, de lo contrario, devuelve **false**.

```
class Solution {
        public static boolean isPalindrome(int x) {
            // Convertir el int en un String
            String xStr = String.valueOf(x);
            // Comprobar si la cadena es la misma hacia delante y
hacia atrás
            return xStr.equals(new
StringBuilder(xStr).reverse().toString());
        }
}
```

10. Regular Expression Matching 28.2% Hard

Dada una cadena de entrada s y un patrón p, implemente la correspondencia de expresiones regulares con soporte para '.' y '*' donde:

1. '.' Coincide con cualquier carácter.
2. '*' Coincide con cero o más del elemento precedente.

La coincidencia debe abarcar toda la cadena de entrada (no parcial).

Enfoque

1. Si **p** es una cadena vacía, la función devolverá **true** si y solo si **s** también es una cadena vacía.
2. Comprobación de si el primer carácter de **s** y **p** son iguales o si el primer carácter de **p** es un punto ('.'). Si cualquiera de estas condiciones se cumple, se establece la variable **firstMatch** en **true**, de lo contrario, en **false**.
3. Si el segundo carácter de **p** es un asterisco ('*'), se devuelve **true** si **isMatch(s, p.substring(2))** es **true** o si **firstMatch** es **true** y **isMatch(s.substring(1), p)** es **true**. Si el segundo carácter de **p** no es un asterisco, se devuelve **true** si **firstMatch** es **true** y **isMatch(s.substring(1), p.substring(1))** es **true**.

```
class Solution {
    public boolean isMatch(String s, String p) {
        if (p.isEmpty()) {
            return s.isEmpty();
        }
        boolean firstMatch = !s.isEmpty() && (s.charAt(0) ==
p.charAt(0) || p.charAt(0) == '.');
        if (p.length() >= 2 && p.charAt(1) == '*') {
            return isMatch(s, p.substring(2)) || firstMatch &&
isMatch(s.substring(1), p);
        } else {
            return firstMatch && isMatch(s.substring(1),
p.substring(1));
        }
    }
}
```

11. Container With Most Water 54.2% Medium

Se te da una matriz entera altura de longitud n. Hay n rectas verticales trazadas tales que los dos puntos extremos de la recta i-ésima son (i, 0) e (i, altura[i]).

Encuentra dos rectas que junto con el eje x formen un contenedor, tal que el contenedor contenga la mayor cantidad de agua.

Devuelve la cantidad máxima de agua que puede almacenar un contenedor.

Ten en cuenta que no puedes inclinar el contenedor.

Enfoque

1. Se inicializan dos variables **izq** y **der** a 0 y el tamaño del arreglo **height**, respectivamente. También la variable **maxArea** a 0.
2. Mientras **izq** sea menor que **der**:
3. Se calcula el área entre el elemento **height[izq]** y **height[der]** como el mínimo de estos dos elementos multiplicado por la distancia entre **izq** y **der**.
4. Se actualiza la variable **maxArea** si la área calculada es mayor que su valor actual.
5. Si **height[izq]** es menor que **height[der]**, aumentar **izq** en 1. Si no, disminuir **der** en 1.
6. Devolver el resultado **maxArea**.

```
public int maxArea(int[] height) {
    int izq = 0;
    int der = height.length - 1;
    int maxArea = 0;
    while (izq < der) {
        int area = Math.min(height[izq], height[der]) * (der - izq);
        maxArea = Math.max(maxArea, area);
        if (height[izq] < height[der]) {
            izq++;
        } else {
            der--;
        }
    }
    return maxArea;
}
}
```

12. Integer to Roman 61.6% Medium

Los números romanos se representan con siete símbolos diferentes: I, V, X, L, C, D y M.

Symbol	Value
I	1
V	5
X	10
L	50
C	100
D	500
M	1000

Por ejemplo, 2 se escribe como II en número romano, simplemente dos uno sumados. El 12 se escribe como XII, que es simplemente X + II. El número 27 se escribe XXVII, que es XX + V + II.

Los números romanos suelen escribirse de mayor a menor, de izquierda a derecha. Sin embargo, el número cuatro no es IIII. En su lugar, el número cuatro se escribe IV. Como el uno está antes del cinco, lo restamos y obtenemos cuatro. El mismo principio se aplica al número nueve, que se escribe IX. Hay seis casos en los que se utiliza la resta:

- I puede colocarse antes de V (5) y X (10) para formar 4 y 9.
- X puede anteponerse a L (50) y C (100) para formar 40 y 90.
- C puede colocarse antes de D (500) y M (1000) para obtener 400 y 900.

Dado un número entero, conviértelo en un número romano.

Enfoque

1. Declara cuatro arrays: **M**, **C**, **X**, e **I**. Cada uno de estos arrays contiene las cadenas de caracteres romanos correspondientes a cada unidad de mil, centena, decena e unidad, respectivamente.
2. Devolver la concatenación de las siguientes cuatro cadenas:
    - El elemento **M[num / 1000]**, que representa las millares.
    - El elemento **C[(num % 1000) / 100]**, que representa las centenas.
    - El elemento **X[(num % 100) / 10]**, que representa las decenas.
    - El elemento **I[num % 10]**, que representa las unidades.

```
class Solution {
    public String intToRoman(int num) {
        String[] M = {"", "M", "MM", "MMM"};
        String[] C = {"", "C", "CC", "CCC", "CD", "D", "DC", "DCC",
"DCCC", "CM"};
        String[] X = {"", "X", "XX", "XXX", "XL", "L", "LX", "LXX",
"LXXX", "XC"};
        String[] I = {"", "I", "II", "III", "IV", "V", "VI", "VII",
"VIII", "IX"};

        return M[num / 1000] + C[(num % 1000) / 100] + X[(num % 100) /
10] + I[num % 10];
    }
}
```

13. Roman to Integer 58.2% Easy

Los números romanos están representados por siete símbolos diferentes: I, V, X, L, C, D y M.

Symbol	Value
I	1
V	5
X	10
L	50
C	100
D	500
M	1000

Por ejemplo, 2 se escribe como II en números romanos, simplemente dos unos sumados. El 12 se escribe como XII, que es simplemente X + II. El número 27 se escribe XXVII, que es XX + V + II.

Los números romanos suelen escribirse de mayor a menor, de izquierda a derecha. Sin embargo, el número cuatro no es IIII. En su lugar, el número cuatro se escribe IV. Como el uno está antes del cinco, lo restamos y obtenemos cuatro. El mismo principio se aplica al número nueve, que se escribe IX. Hay seis casos en los que se utiliza la resta:

- I puede colocarse antes de V (5) y X (10) para formar 4 y 9.
- X puede anteponerse a L (50) y C (100) para formar 40 y 90.
- C puede colocarse antes de D (500) y M (1000) para formar 400 y 900.

Dado un número romano, conviértelo en un número entero.

Enfoque

1. Inicializar dos variables: **res** a 0 y **num** a 0.
2. Iterar a través de cada carácter de **s** desde el último hasta el primero.
3. Utilizar un **switch** para asignar a **num** el valor numérico correspondiente al carácter actual de en números romanos.
4. Si **4** * **num** es menor que **res**, disminuir **res** en **num**. Si no, aumentar **res** en **num**.
5. Devolver resultado **res**.

```java
class Solution {
    public int romanToInt(String s) {
        int res = 0, num = 0;
        for (int i = s.length()-1; i >= 0; i--) {
            switch(s.charAt(i)) {
                case 'I': num = 1; break;
                case 'V': num = 5; break;
                case 'X': num = 10; break;
                case 'L': num = 50; break;
                case 'C': num = 100; break;
                case 'D': num = 500; break;
                case 'M': num = 1000; break;
            }
            if (4 * num < res) res -= num;

            else res += num;
        }
        return res;
    }
}
```

14. Longest Common Prefix 40.8% Easy

Escriba una función para encontrar la cadena de prefijo común más larga entre una matriz de cadenas.

Si no hay un prefijo común, devuelve una cadena vacía "".

Enfoque

1. Si el tamaño del arreglo **strs** es 0, devolver una cadena vacía.
2. Asignar el primer elemento del array **strs** a la variable **prefix**.
3. Iterar a través de cada elemento de **strs** desde el segundo hasta el último.
4. Mientras la posición de la cadena **prefix** dentro de **strs[i]** sea diferente de 0, disminuir el tamaño de **prefix** en 1 y verificar si **prefix** es una cadena vacía. Si es así, devolver una cadena vacía.
5. Devolver **prefix**.

```
public String longestCommonPrefix(String[] strs) {
    if (strs.length == 0) {
        return "";
    }
    String prefix = strs[0];
    for (int i = 1; i < strs.length; i++) {
        while (strs[i].indexOf(prefix) != 0) {
            prefix = prefix.substring(0, prefix.length() - 1);
            if (prefix.isEmpty()) {
                return "";
            }
        }
    }
    return prefix;
}
}
```

15. 3Sum 32.4% Medium

Dada una matriz de enteros nums, devolver todos los tripletes [nums[i], nums[j], nums[k]] tales que i != j, i != k, y j != k, y nums[i] + números[j] + números[k] == 0.

Tenga en cuenta que el conjunto de soluciones no debe contener tripletes duplicados.

Enfoque

1. Ordenar el array **nums** de forma ascendente.
2. Inicializar la variable **n** con el tamaño del arreglo **nums**.
3. Nueva lista vacía llamada **triplets** para almacenar los resultados.
4. Iterar a través de cada elemento de **nums** desde el primero hasta el último.
5. Si el elemento actual es igual al anterior, se salta a la siguiente iteración.
6. Inicializa las variables **j** y **k** con el elemento siguiente al actual y el último elemento de **nums**, respectivamente.
7. Mientras **j** sea menor que **k**, realiza los siguientes pasos:
    - Si la suma de **nums[i]**, **nums[j]** y **nums[k]** es igual a 0, se agrega una lista con estos tres elementos a **triplets** y se aumenta **j** en 1. Si el elemento siguiente a **j** es igual al elemento actual, saltar a la siguiente iteración.
    - Si la suma de **nums[i]**, **nums[j]** y **nums[k]** es menor que 0, aumentar **j** en 1.
    - Si la suma de **nums[i]**, **nums[j]** y **nums[k]** es mayor que 0, se disminuye **k** en 1.
8. Dovelver la lista **triplets**.

```java
class Solution {

    public List<List<Integer>> threeSum(int[] nums) {
        // Ordenar el array
        Arrays.sort(nums);
        // Longitud del array
        int n = nums.length;
        // List resultante
        List<List<Integer>> triplets = new ArrayList<>();
        // Bucle
        for (int i = 0; i < n; i++) {
            // Eliminar duplicados
            if (i > 0 && nums[i] == nums[i - 1]) {
                continue;
            }
            // Pointers izquierdo y derecho
            int j = i + 1;
            int k = n - 1;
            // Bucle para los pares restantes
            while (j < k) {
                if (nums[i] + nums[j] + nums[k] == 0) {
                    triplets.add(Arrays.asList(nums[i], nums[j],
nums[k]));
                    j++;
                    while (j < k && nums[j] == nums[j - 1]) {
                        j++;
                    }
                } else if (nums[i] + nums[j] + nums[k] < 0) {
                    j++;
                } else {
                    k--;
                }
            }
        }
        return triplets;
    }
}
```

16. 3Sum Closest 46.0% Medium

Dada una matriz de enteros nums de longitud n y un objetivo entero, encuentre tres enteros en nums tales que la suma sea la más cercana al objetivo.

Devuelve la suma de los tres enteros.

Puede suponer que cada entrada tendría exactamente una solución.

Enfoque

1. Inicializar la variable **n** con el tamaño del array **nums**.
2. Inicializar la variable **mejorSuma** con el valor máximo de un entero.
3. Ordenar el array **nums** en orden ascendente.
4. Iterar a través de cada elemento de **nums** desde el primero hasta el penúltimo.
5. Inicializar las variables **izq** y **der** con el elemento siguiente al actual y el último elemento de **nums**, respectivamente.
6. Mientras **izq** sea menor que **der**, se realizan los siguientes pasos:
    - Calcular la suma de **nums[i]**, **nums[izq]** y **nums[der]**.
    - Si la diferencia absoluta entre esta suma y el objetivo es menor que la diferencia absoluta entre **mejorSuma** y el objetivo, asignar esta suma a **mejorSuma**.
    - Si la suma es menor que el objetivo, aumenta **izq** en 1. Si es mayor, se disminuye **der** en 1. Si es igual, devuelve **mejorSuma**.
7. Devolver resultado de la variable **mejorSuma**.

```java
public static int threeSumClosest(int[] nums, int target) {
    int n = nums.length;
    int mejorSuma = Integer.MAX_VALUE;
    // Ordenar el array nums de menor a mayor
    Arrays.sort(nums);
    // Recorrer el array nums y buscar los otros dos enteros que sumen
lo más cerca posible
    // del objetivo - nums[i]
    for (int i = 0; i < n - 2; i++) {
        int izq = i + 1;
        int der = n - 1;
        while (izq < der) {
            int sum = nums[i] + nums[izq] + nums[der];
            if (Math.abs(sum - target) < Math.abs(mejorSuma - target))
{
                mejorSuma = sum;
            }
            if (sum < target) {
                izq++;
            } else if (sum > target) {
                der--;
            } else {
                // La suma es igual al objetivo, por lo que no podemos
encontrar una suma más cercana
                return mejorSuma;
            }
        }
    }
    return mejorSuma;
}
}
```

17. Letter Combinations of a Phone Number 55.9% Medium

Dada una cadena que contiene dígitos del 2 al 9 inclusive, devuelva todas las combinaciones de letras posibles que el número podría representar. Devuelve la respuesta en cualquier orden.

A continuación se muestra una asignación de dígitos a letras (igual que en los botones del teléfono). Tenga en cuenta que 1 no se asigna a ninguna letra.

Enfoque

1. Inicializar una lista de String **combinaciones** y verificar si **digits** es nulo o está vacío. Si es así, devolver el String **combinaciones**.
2. Inicializar el array **mappingNumerosLetras** con los mapeos de números a letras.
3. Llamar a función **findCombinations** con los argumentos **combinaciones**, **digits**, un nuevo objeto **StringBuilder**, 0 y **mappingNumerosLetras**.
4. Devolver **combinaciones**.

```java
class Solution {

    public static List<String> letterCombinations(String digits) {
        // Lista resultante
        List<String> combinaciones = new ArrayList<>();

        if (digits == null || digits.isEmpty()) {
            return combinaciones;
        }
        // Mappings de letras y números
        String[] mappingNumerosLetras = new String[]{"Anirudh","is
awesome","abc","def","ghi","jkl","mno","pqrs","tuv","wxyz"};
        findCombinations(combinaciones, digits, new StringBuilder(),
0, mappingNumerosLetras);
        return combinaciones;
    }
    private static void findCombinations(List<String> combinaciones,
String digits, StringBuilder previous, int index, String[]
mappingNumerosLetras) {
        //  Condición base para parar la recursión
        if (index == digits.length()) {
            combinaciones.add(previous.toString());
            return;
        }
        // Obtener las letras correspondientes al índice actual de
cadena de dígitos
        String letters = mappingNumerosLetras[digits.charAt(index) -
'0'];
        // Bucle de todos los caracteres en la combinación actual de
letras
        for (char c : letters.toCharArray()) {
            findCombinations(combinaciones, digits,
previous.append(c), index + 1, mappingNumerosLetras);
            previous.deleteCharAt(previous.length() - 1);
        }
    }
}
```

18. 4Sum 36.3% Medium

Dada una matriz nums de n enteros, devuelve una matriz de todos los cuartetos únicos [nums[a], nums[b], nums[c], nums[d]] tales que:

- 0 <= a, b, c, d < n
- a, b, c y d son distintos.
- números[a] + números[b] + números[c] + números[d] == objetivo

Puede devolver la respuesta en cualquier orden.

Enfoque

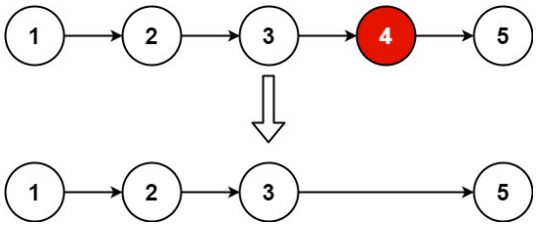
1. Declara una lista llamada "sol" que almacenará la solución del problema.
2. Ordena los elementos del arreglo "nums" utilizando el método "sort" de la clase "Arrays".
3. Llama al método privado "nSum" con los parámetros necesarios: "nums", "4", "target", "0", "nums.length-1", una lista vacía y "sol".
4. El método "nSum" verifica si es posible encontrar una solución válida dado el rango de los índices y el valor objetivo. Si no es posible, la función regresa.
5. Si el valor de "n" es igual a 2, se utilizan dos puntos y se recorren el arreglo, si se encuentra una combinación de dos números que sumen el valor objetivo entonces se añade a la lista solución y se actualizan los índices para seguir buscando.
6. Si el valor de "n" es diferente a 2, se utiliza recursión. se recorre el arreglo, se agrega el elemento actual a la lista temporal "path" y se llama de nuevo al método "nSum" con un valor de "n" decrementado en 1 y el objetivo decrementado en el valor del elemento actual. al final se elimina el elemento agregado temporalmente.
7. Finalmente, se entrega la solución.

```java
class Solution {
    public List<List<Integer>> fourSum(int[] nums, int target) {
        List<List<Integer>> sol = new ArrayList<>();
        Arrays.sort(nums);

        nSum(nums, 4, target, 0, nums.length - 1, new ArrayList<>(),
sol);
        return sol;
    }
    private void nSum(int[] nums, long n, long target, int l, int r,
List<Integer> path, List<List<Integer>> ans) {
        if (r - l + 1 < n || target < nums[l] * n || target > nums[r]
* n)
            return;
        if (n == 2) {
            while (l < r) {
                final int sum = nums[l] + nums[r];
                if (sum == target) {
                    path.add(nums[l]);
                    path.add(nums[r]);
                    ans.add(new ArrayList<>(path));
                    path.remove(path.size() - 1);
                    path.remove(path.size() - 1);
                    ++l;
                    --r;
                    while (l < r && nums[l] == nums[l - 1])
                        ++l;
                    while (l < r && nums[r] == nums[r + 1])
                        --r;
                } else if (sum < target) {
                    ++l;
                } else {
                    --r;
                }
            }
            return;
        }
        for (int i = l; i <= r; ++i) {
            if (i > l && nums[i] == nums[i - 1])
                continue;
            path.add(nums[i]);
            nSum(nums, n - 1, target - nums[i], i + 1, r, path, ans);
            path.remove(path.size() - 1);
        }
    }
}
```

19. Remove Nth Node From End of List 40.3% Medium

Dado el encabezado de una lista enlazada, elimine el nodo n del final de la lista y devuelva su encabezado.



Enfoque

1. Verifica si la cabeza de la lista ligada es nula, y si lo es, devuelve la cabeza de la lista.
2. Inicializa dos punteros, p1 y p2, apuntando a la cabeza de la lista ligada.
3. Avanza el puntero p1 n pasos.
4. Si p1 es nulo, significa que la lista ligada tiene menos de n nodos, por lo que se devuelve la cabeza de la lista modificada (que es la lista original con el primer nodo eliminado).
5. Mueve los punteros p1 y p2 un paso cada vez hasta que p1 alcanza el final de la lista ligada.
6. Establece el puntero "next" de p2 al nodo después del n-ésimo nodo desde el final (que actualmente está apuntado por p2.next.next)
7. Devuelve la lista ligada modificada

```java
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode() {}
 *     ListNode(int val) { this.val = val; }
 *     ListNode(int val, ListNode next) { this.val = val; this.next = next; }
 * }
 */
class Solution {
    public ListNode removeNthFromEnd(ListNode head, int n) {
        // Edge case: head is null
        if (head == null) {
            return head;
        }
        // Initialize two pointers, p1 and p2, to the head of the
linked list
        ListNode p1 = head;
        ListNode p2 = head;
        // Move p1 forward by n steps
        for (int i = 0; i < n; i++) {
            p1 = p1.next;
        }
        // If p1 is null, it means the linked list has fewer than n
nodes,
        // so we return the head of the modified linked list (which is
the original list with the first node removed)
        if (p1 == null) {
            return head.next;
        }
        // Move p1 and p2 forward one step at a time until p1 reaches
the end of the linked list
        while (p1.next != null) {
            p1 = p1.next;
            p2 = p2.next;
        }
        // Set the next pointer of p2 to the node after the n-th node
from the end (which is currently pointed to by p2.next.next)
        p2.next = p2.next.next;
        // Return the modified linked list
        return head;
    }
}
```

20. Valid Parentheses 40.5% Easy

Dada una cadena s que sólo contiene los caracteres '(', ')', '{', '}', '[' y ']', determine si la cadena de entrada es válida.

Una cadena de entrada es válida si:

Los corchetes abiertos deben estar cerrados por el mismo tipo de corchetes.

Los corchetes abiertos deben cerrarse en el orden correcto.

Cada corchete cerrado tiene su correspondiente corchete abierto del mismo tipo.

Enfoque

1. Crear una pila vacía de caracteres llamada "stack".
2. Recorrer la cadena de caracteres utilizando un bucle "for" y guardando cada caracter en una variable "c"
3. Dentro del bucle "for", si el caracter "c" es igual a '(', '{' o '[', agregar el caracter correspondiente a la pila.
4. Si el caracter "c" es igual a ')', '}' o ']', verificar si la pila está vacía o si el último caracter insertado en la pila no es igual a "c". Si cualquiera de estas condiciones se cumple, retornar "false" inmediatamente.
5. Fuera del bucle "for", si la pila está vacía, retornar "true" ya que todos los paréntesis se han emparejado correctamente. En caso contrario, retornar "false" ya que faltan emparejar algunos paréntesis.

```java
class Solution {
    public boolean isValid(String s) {
        Stack<Character> stack = new Stack<>();
        for (char c : s.toCharArray()) {
            if (c == '(') {
                stack.push(')');
            } else if (c == '{') {
                stack.push('}');
            } else if (c == '[') {
                stack.push(']');
            } else if (stack.isEmpty() || stack.pop() != c) {
                return false;
            }
        }
        return stack.isEmpty();
    }
}
```

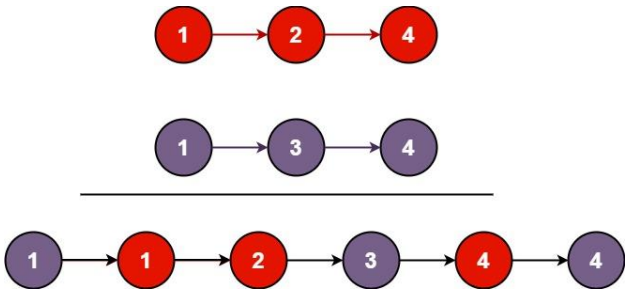
21. Merge Two Sorted Lists 62.1% Easy

Se le dan las cabezas de dos listas ordenadas enlazadas lista1 y lista2.

Fusione las dos listas en una sola lista ordenada. La lista debe hacerse empalmando los nodos de las dos primeras listas.

Devuelve la cabeza de la lista enlazada combinada.

Ejemplo1:



1. Crea un nodo "temp" con un valor de cero y un puntero "next" nulo para que sea la cabeza de la lista resultante.
2. Crea un puntero "actual" para recorrer la lista resultante, inicializado con el nodo "temp"
3. Recorre las dos listas de entrada "l1" y "l2" mediante un bucle "while"
4. Dentro del bucle, si el valor del primer elemento de "l1" es menor o igual al primer elemento de "l2", entonces se enlaza el nodo actual de "actual" con el primer elemento de "l1" y se avanza el puntero de "l1" al siguiente elemento.
5. Si el valor del primer elemento de "l1" es mayor al primer elemento de "l2", entonces se enlaza el nodo actual de "actual" con el primer elemento de "l2" y se avanza el puntero de "l2" al siguiente elemento.
6. Luego se avanza el puntero "actual" al siguiente nodo
7. Una vez que una de las dos listas llega al final, se enlaza el puntero "next" del último nodo "actual" con el resto de la lista no vacía, ya sea "l1" o "l2"
8. Finalmente se devuelve el siguiente nodo de "temp" como cabeza de la lista resultante

```java
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode() {}
 *     ListNode(int val) { this.val = val; }
 *     ListNode(int val, ListNode next) { this.val = val; this.next = next; }
 * }
 */
class Solution {
    public ListNode mergeTwoLists(ListNode list1, ListNode list2) {
        ListNode temp = new ListNode(0);
        ListNode actual = temp;

        while (list1 != null && list2 != null) {
            if (list1.val < list2.val) {
                actual.next = list1;
                list1 = list1.next;
            } else {
                actual.next = list2;
                list2 = list2.next;
            }
            actual = actual.next;
        }
        if (list1 != null) {
            actual.next = list1;
        } else {
            actual.next = list2;
        }
        return temp.next;
    }
}
```

22. Generate Parentheses 72.1% Medium

Dados n pares de paréntesis, escribe una función para generar todas las combinaciones de paréntesis bien formados.

Enfoque

1. Crear una lista vacía llamada "sol" que almacenará las combinaciones de paréntesis válidos.
2. Llamar al método "generateParenthesis" con los parámetros necesarios: "sol", una cadena vacía "s", dos contadores "open" y "close" inicializados en cero y el número de pares de paréntesis "n".
3. Dentro del método "generateParenthesis": a. Establecer un caso base en el que si el contador "open" y "close" son iguales al número de pares de paréntesis "n", se agrega la cadena actual "s" a la lista "sol" y se retorna. b. Si el contador "open" es menor al número de pares de paréntesis "n", se llama de nuevo a "generateParenthesis" con la cadena actual "s" concatenada con un paréntesis abierto, el contador "open" incrementado en uno, el contador "close" no cambia y el número de pares de paréntesis "n". c. Si el contador "close" es menor al contador "open", se llama de nuevo a "generateParenthesis" con la cadena actual "s" concatenada con un paréntesis cerrado, el contador "open" no cambia, el contador "close" incrementado en uno y el número de pares de paréntesis "n".
4. Finalmente, retornar la lista "sol" con las combinaciones de paréntesis válidos.

```java
class Solution {
    public List<String> generateParenthesis(int n) {
        // Lista final de sol
        List<String> sol = new ArrayList<>();
        /// Generación recursiva de paréntesis
        generateParenthesis(sol, "", 0, 0, n);
        return sol;
    }
    private void generateParenthesis(List<String> sol, String s, int
open, int close, int n) {
        // Caso base
        if (open == n && close == n) {
            sol.add(s);
            return;
        }
        // Si sobran paréntesis
        if (open < n) {
            generateParenthesis(sol, s + "(", open + 1, close, n);
        }
        // Si faltan paréntesis
        if (close < open) {
            generateParenthesis(sol, s + ")", open, close + 1, n);
        }
    }
}
```

23. Merge k Sorted Lists 48.6% Hard

Se le da un array de k listas enlazadas, cada lista enlazada está ordenada en orden ascendente.

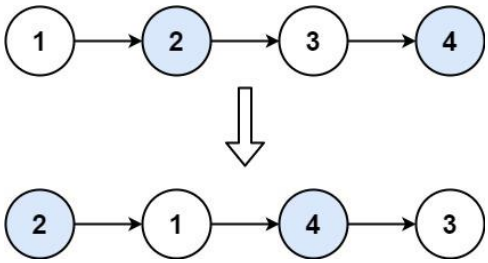
Fusiona todas las listas enlazadas en una lista enlazada ordenada y devuélvela.

1. Verificar si el array de listas "lists" es nulo o vacío, y si es así, retornar nulo.
2. Crear una cola de prioridad "queue" que compare los valores de cada nodo de las listas utilizando el comparador "(n1, n2) -> n1.val - n2.val".
3. Iterar a través del array de listas y agregar el primer nodo de cada lista a la cola de prioridad siempre y cuando no sea nulo.
4. Crear un nodo dummy con un valor de cero y un puntero "next" nulo para que sea la cabeza de la lista.
5. Crear un puntero "current" igual al nodo dummy.
6. Mientras la cola de prioridad no esté vacía, obtener el nodo con el valor más pequeño de la cola de prioridad y asignarlo como el siguiente nodo de "current"
7. Si el nodo actual tiene un next no nulo, agregar ese nodo al final de la cola de prioridad.
8. Retornar el siguiente nodo del nodo dummy, que es la cabeza de la lista ordenada resultante.

```
class Solution {
    public ListNode mergeKLists(ListNode[] lists) {
        if (lists == null || lists.length == 0) {
            return null;
        }
        PriorityQueue<ListNode> queue = new PriorityQueue<>((n1, n2) -
> n1.val - n2.val);
        // put the first node of each list into the priority queue
        for (ListNode node : lists) {
            if (node != null) {
                queue.offer(node);
            }
        }
        ListNode dummy = new ListNode(0);
        ListNode current = dummy;
        while (!queue.isEmpty()) {
            // get the node with the smallest value
            ListNode node = queue.poll();
            current.next = node;
            current = current.next;
            if (node.next != null) {
                queue.offer(node.next);
            }
        }
        return dummy.next;
    }
}
```

24. Swap Nodes in Pairs 60.7% Medium

Dada una lista enlazada, intercambie cada dos nodos adyacentes y devuelva su cabeza. Debe resolver el problema sin modificar los valores en los nodos de la lista (es decir, solo se pueden cambiar los nodos).



1. La función debe comienzar verificando si la cabeza es nula o si no tiene un siguiente nodo. Si es así, devuelve la cabeza tal como está.
2. Se crea un "dummy head" (cabezal ficticio) para simplificar el código
3. Inicializar dos punteros: "curr" y "next"
4. Utilizar un bucle "while" para recorrer la lista y intercambiar los nodos en pares
5. Dentro del bucle, intercambiar los nodos utilizando un nodo temporal.
6. Actualizar los punteros "curr" y "next"
7. Al final del bucle, devolver la lista modificada, comenzando desde el siguiente nodo del cabezal ficticio.

```java
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode() {}
 *     ListNode(int val) { this.val = val; }
 *     ListNode(int val, ListNode next) { this.val = val; this.next = next; }
 * }
 */
class Solution {
    public ListNode swapPairs(ListNode head) {
        // Edge case: head is null or head has no next node
        if (head == null || head.next == null) {
            return head;
        }
        // Create a dummy head to simplify the code
        ListNode dummy = new ListNode();
        dummy.next = head;
        // Initialize two pointers: curr and next
        ListNode curr = dummy;
        ListNode next = head;
        // Loop through the linked list and swap nodes in pairs
        while (next != null && next.next != null) {
            // Swap the nodes
            ListNode temp = next.next;
            curr.next = temp;
            next.next = temp.next;
            temp.next = next;
            // Update curr and next
            curr = next;
            next = curr.next;
        }
        // Return the modified linked list, starting from the dummy
head's next node
        return dummy.next;
    }
}
```

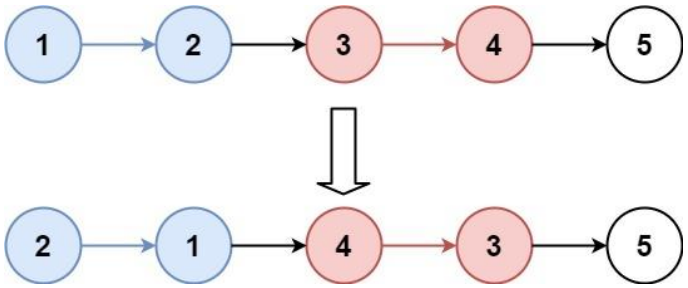
25. Reverse Nodes in k-Group 54.0% Hard

Dada la cabecera de una lista enlazada, invierte los nodos de la lista k cada vez, y devuelve la lista modificada.

k es un número entero positivo y es menor o igual que la longitud de la lista enlazada. Si el número de nodos no es múltiplo de k, los nodos omitidos, al final, deben permanecer como están.

No se pueden modificar los valores de los nodos de la lista, sólo se pueden modificar los nodos en sí.

Ejemplo1 :



1. La función reverseKGroup toma como entrada una referencia al primer nodo de una lista ligada y un entero k, y devuelve una referencia al primer nodo de la lista ligada con los k grupos consecutivos de nodos invertidos en orden.
2. La función utiliza dos bucles anidados para iterar sobre los nodos de la lista y realizar las operaciones de inversión necesarias.
3. El primer bucle externo itera sobre los grupos de k nodos en la lista, mientras que el bucle interno itera sobre los nodos individuales dentro de cada grupo y los reordena en orden inverso.

```java
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode() {}
 *     ListNode(int val) { this.val = val; }
 *     ListNode(int val, ListNode next) { this.val = val; this.next = next; }
 * }
 */
class Solution {
    public ListNode reverseKGroup(ListNode head, int k) {
        if (head == null || k == 1)
            return head;

        final int length = getLength(head);
        ListNode dummy = new ListNode(0, head);
        ListNode prev = dummy;
        ListNode curr = head;

        for (int i = 0; i < length / k; ++i) {
            for (int j = 0; j < k - 1; ++j) {
                ListNode next = curr.next;
                curr.next = next.next;
                next.next = prev.next;
                prev.next = next;
            }
            prev = curr;
            curr = curr.next;
        }
        return dummy.next;
    }
    private int getLength(ListNode head) {
        int length = 0;
        for (ListNode curr = head; curr != null; curr = curr.next)
            ++length;
        return length;
    }
}
```