Taller Problemas Búsqueda y Ordenamiento

Búsqueda Binaria

Edinson Antolinez 1151436

Docente:
Ing. Milton Contreras

Búsqueda Binaria

El algoritmo de búsqueda binaria es un algoritmo eficiente para encontrar un elemento en una lista ordenada. En lugar de buscar de manera secuencial a través de cada elemento de la lista, la búsqueda binaria divide repetidamente la lista en dos partes y busca en la mitad que podría contener el elemento buscado.

```
public class BinarySearch{
    static int q = 0;
    // binarySearch(a, k, 0, a.length-1){
    int binarySearch(int a [], int k, int l, int r){
        int ans = 0, m = 0;
        q++;
        if(l>r) ans=-1;
        else{
            m = (int)Math.floor((l+r)/2);
            if (k == a[m]) ans = m;
            else if (k < a[m]) ans = binarySearch(a,k,l,m-1);
            else ans = binarySearch(a,k,m+1,r);
        }
        return ans;
    }
}
```

Código extraído de la uvirtual

1) Resolver problema en la plataforma https://leetcode.com/problems/binary-search/

Dada una matriz de números enteros `nums` que se ordena de forma ascendente y un número entero `target`, escribe una función para buscar `target` en `nums`. Si `target` existe, devuelve su índice. De lo contrario, regresa -1.

Debe escribir un algoritmo con O(log n) complejidad de tiempo de ejecución.

```
Example 1:

Input: nums = [-1,0,3,5,9,12], target = 9
Output: 4
Explanation: 9 exists in nums and its index is 4

Example 2:

Input: nums = [-1,0,3,5,9,12], target = 2
Output: -1
Explanation: 2 does not exist in nums so return -1
```

Para llegar a la solución tomé el algoritmo que se encontraba en el material de estudio, pero se debió adaptar a los requisitos de leetcode, porque en este el método debía recibir solo dos parámetros de entrada.

```
class Solution {
    public int search(int[] nums, int target) {
        int l = 0;
        int r = nums.length - 1;

        while (l <= r) {
            int m = l + (r - l) / 2;

            if (nums[m] == target) {
                return m;
            }

            if (nums[m] < target) {
                l = m + 1;
            } else {
                r = m - 1;
            }
        }
        return -1;
    }
}
```

2) Desarrollar una aplicación que genera al menos 100 casos de prueba para el problema. Los casos deben cubrir todas las posibilidades de casos del problema, de manera equilibrada.

Para generar los casos de prueba realice una clase llamada TestCase para mayor manejo y comodidad. En las que se encuentran las propiedades longitud para referirse al tamaño del listado, búsqueda para saber cuál es el elemento a buscar, pos que se refiere a la posición del elemento a buscar y lista.

También se implementó el método toString de la clase para explorar los resultados además del método arrayToString para mayor profundidad.

```
class TestCase {
    private int longitud;
    private int busqueda;
    private int[] lista;
    private int pos;

    public TestCase() {}

    public TestCase(int l, int b, int[] lista, int pos) {
        this.longitud = l;
        this.busqueda = b;
        this.lista = lista;
        this.pos = pos;
    }

    public int getLongitud() {
        return longitud;
    }
    public int getBusqueda() {
        return busqueda;
    }
    public int[] getLista() {
        return lista;
    }
    public int getPos() {
        return pos;
    }
    public void setPos(int p) {
        this.pos = p;
    }

    @Override
    public String toString() {
        if (this.pos == -1) {
            return "el elemento no se encuentra en el listado";
        } else {
            return "El numero " + busqueda + " se encuentra en la
posicion " + pos + " del listado de longitud "
                    + longitud
                    + " \n  arreglo: " + arrayToString();
        }
    }

    public String arrayToString() {
        StringBuilder sb = new StringBuilder();
        sb.append("[");
        for (int i = 0; i < this.lista.length; i++) {
            sb.append(this.lista[i]);
            if (i < this.lista.length - 1) {
                sb.append(", ");
            }
        }
        sb.append("]");
        return sb.toString();
    }
}
```

Continue creando la clase Pruebas para implementar el método que generara los 100 casos aplicando las siguientes condiciones:

- En los primeros 30 casos el elemento se encuentra en el inicio.
- En los 20(30;50) siguientes casos el elemento se encuentra en el medio.
- En los 30(50;80) siguientes casos el elemento se encuentra al final.
- En los últimos 20(80;100) casos el elemento no se encuentra.

Además que en 50 casos el tamaño del listado será par y la otra mitad será impar.

```
public class Pruebas {

    public static List<TestCase> generateArraysOrdenado() {
        List<TestCase> arrays = new ArrayList<>();

        for (int i = 0; i < 100; i++) {
            int size = i < 50 ? getRandomNumber(2, 100) :
getRandomNumber(1, 99);
            int[] array = generarListado(size, true);

            TestCase temp = new TestCase();

            if (i < 30) {
                // inicio
                temp = new TestCase(size, array[0], array, -1);
            } else if (i >= 30 && i < 50) {
                // mitad
                int mitad = size / 2;
                temp = new TestCase(size, array[mitad], array, -1);
            } else if (i >= 50 && i < 80) {
                // final
                temp = new TestCase(size, array[size - 1], array, -1);
            } else {
                // no esta
                temp = new TestCase(size, size + 1, array, -1);
            }

            arrays.add(temp);
        }
        return arrays;
    }
```

El método getRandomNumber() genera de manera aleatoria los tamaños de los listados, variando los parámetros de entrada para que sean par o impar (2, 100); (1, 99).

```
    public static int getRandomNumber(int min, int max) {
        Random random = new Random();
        return random.nextInt(max - min + 1) + min;
    }
```

El método generarListado() recibe el tamaño del arreglo y a partir de ese parámetro crea una lista de números de manera consecutiva y ascendente, también se encuentran un parámetro order y un condicional para mezclar el listado de manera aleatoria pero eso el caso de prueba de listados en desorden.

```
    public static int[] generarListado(int size, boolean order) {
        int[] array = new int[size];
        List<Integer> numbers = new ArrayList<>();
        for (int i = 0; i < size; i++) {
            numbers.add(i);
        }

        if (!order)
            Collections.shuffle(numbers);

        for (int i = 0; i < size; i++) {
            array[i] = numbers.get(i);
        }
        return array;
    }
```

Todo está implementación se encuentra focalizada en generar listados ya ordenados.

Al final tengo la clase Main.

```
public class Main {
    public static void main(String[] args) {
        List<TestCase> testCases = Pruebas.generateArraysOrdenado();

        for (TestCase testCase : testCases) {
            BusquedaBinaria solution = new BusquedaBinaria();
            int result = solution.busquedaBinaria(testCase.getLista(),
testCase.getBusqueda());
            testCase.setPos(result);
            System.out.println(testCase.toString());
        }
    }
}
```

Hago el llamado a la clase Pruebas el método generateArraysOrdenado() Para generar los 100 casos de pruebas.

Itero los casos generados, por cada uno hago una instancia de las clase BusquedaBinaria y realizo el llamado del algoritmo de búsqueda. El resultado es guardado en una variable entera para luego ser asignada al atributo pos de la clase TestCase y al final hago una impresión en la consola.

Internamente se llamará el método toString de la clase que implemente para los casos.

Resultados:
Impresiones en consola de los casos básicos.

```
El numero 0 se encuentra en la posicion 0 del listado de longitud 28
  arreglo: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27]


El numero 20 se encuentra en la posicion 20 del listado de longitud 40
  arreglo: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33,
34, 35, 36, 37, 38, 39]

El numero 8 se encuentra en la posicion 8 del listado de longitud 9
  arreglo: [0, 1, 2, 3, 4, 5, 6, 7, 8]


el elemento no se encuentra en el listado
```

Todo fueron unos casos muy simples en los que solo creé listados ordenados entre 0 y 100. Empecé a revisar porque no funcionaba en listas desordenadas entonces implemente unos métodos para esas pruebas.

```
    public static List<int[]> generateArraysDesorden(int busqueda) {
        List<int[]> arrays = new ArrayList<>();

        for (int i = 0; i < 100; i++) {
            int size = i < 50 ? getRandomNumber(2, 100) :
getRandomNumber(1, 99);
            int[] array = generarListado(size, false);

            if (i < 30) {
                // inicio
                colocarEnPosicion(array, busqueda, 0);
            } else if (i >= 30 && i < 50) {
                // mitad
                colocarEnPosicion(array, busqueda, 1);
            } else if (i >= 50 && i < 80) {
                // final
                colocarEnPosicion(array, busqueda, 2);
            } else {
                // no esta
                reemplazarNumero(array, busqueda, 1);
            }

            arrays.add(array);
        }

        return arrays;
    }
```

Es básicamente un método en el que replica el comportamiento de las condiciones o reglas del primer método de pruebas. Pero en cada condición realice un método distinto. Todo partiendo que en el método generarListado el parámetro order está en false, por lo que los listados se encuentran desordenados y recibo como parámetro el número que quiero buscar en el listado.

En el método colocarEnPosicion() calculo los 3 casos de posición y dependiendo del parámetro selecciono uno. Tambien evaluo que el numero a buscar no este repetido, de ser asi cambio de posición el valor que se encuentra con la posición del numero repetido y si no solo reemplazo el valor en el listado.

```
    public static void colocarEnPosicion(int[] arreglo, int numero,
int cas) {
        int mitad = arreglo.length / 2;
        int fin = arreglo.length - 1;
        int posicion = 0;

        boolean numeroRepetido = false;
        int posicionNumeroRepetido = -1;
        for (int i = 0; i < arreglo.length; i++) {
            if (arreglo[i] == numero) {
                numeroRepetido = true;
                posicionNumeroRepetido = i;
                break;
            }
        }

        if (cas == 1) {
            posicion = mitad;
        } else if (cas == 2) {
            posicion = fin;
        }

        if (numeroRepetido) {
            int temp = arreglo[posicion];
            arreglo[posicion] = numero;
            arreglo[posicionNumeroRepetido] = temp;
        } else {
            arreglo[posicion] = numero;
        }
    }
```

Al final en la clase Main hago el llamado del método generateArrayDesorden y envió al número que quiero buscar, en este caso al 8.

En cada iteración imprimo el listado antes de mostrar los resultados de la búsqueda.

```
        List<int[]> testCases = Pruebas.generateArraysDesorden(8);

        for (int[] testCase : testCases) {
            System.out.println(Pruebas.arrayToString(testCase));

            BusquedaBinaria solution = new BusquedaBinaria();
            int result = solution.busquedaBinaria(testCase, 8);

            if (result == -1) {
                System.out.println("objetivo no encontrado en la
matriz");
            } else {
                System.out
                        .println(
                                "El objetivo se encuentra en la
posicion: " + result + " longitud: " +
                                        testCase.length);
            }
        }
```

Resultados:
Algunas impresiones de consola en las que no funciona el algoritmo de búsqueda por el desorden de los listados.

```
objetivo no encontrado en la matriz
[7, 3, 4, 6, 1, 2, 0, 8]

objetivo no encontrado en la matriz
[10, 6, 1, 2, 4, 9, 0, 5, 3, 7, 8]

objetivo no encontrado en la matriz
[43, 31, 66, 64, 61, 38, 24, 32, 50, 0, 10, 65, 28, 14, 2, 21, 37, 17,
39, 7, 63, 68, 70, 73, 59, 16, 46, 6, 54, 55, 47, 1, 33, 56, 22, 67,
49, 60, 11, 48, 57, 69, 15, 26, 35, 29, 30, 72, 19, 27, 5, 12, 51, 53,
44, 71, 9, 34, 23, 3, 58, 4, 40, 62, 45, 25, 41, 36, 20, 42, 13, 18,
52, 8]
```

Conclusiones

El algoritmo de búsqueda binaria requiere que los datos estén ordenados para funcionar correctamente. La razón principal por la que no funciona en datos desordenados es que se basa en la propiedad de orden de la lista para tomar decisiones sobre qué mitad de la lista explorar en cada paso.

Cuando la lista no está ordenada, no se puede confiar en la propiedad de orden para tomar decisiones precisas sobre la ubicación del elemento buscado. El algoritmo de búsqueda binaria supone que los elementos en la mitad inferior de la lista son menores que el elemento medio, y los elementos en la mitad superior son mayores. Si los datos están desordenados, esta suposición puede no ser válida, lo que llevará a resultados incorrectos.

Además, si los datos están desordenados, puede haber múltiples instancias del mismo valor dispersas en diferentes partes de la lista. Esto dificulta la determinación de la ubicación precisa del elemento buscado y complica el proceso de búsqueda.

En resumen, el algoritmo de búsqueda binaria requiere una lista ordenada para poder realizar una búsqueda eficiente. Si los datos están desordenados, se necesitaría un algoritmo de búsqueda diferente, como la búsqueda secuencial, que verifica cada elemento en la lista hasta encontrar una coincidencia.