Instituto tecnológico de Orizaba

Sistemas Computacionales

Estructura de Datos

Unidad 6

Profesora: María Jacinta Martínez Castillo

Integrantes del equipo:

Moran De la Cruz Aziel 21011006

Jiménez Jiménez Carlos Yael 21010975

Sebastián Brito García 21010929

Bandala Hernández Sebastián 21010921

Algoritmos de Búsqueda y Ordenamiento

ALGORITMOS DE BÚSQUEDA

Búsqueda Secuencial

La búsqueda lineal o secuencial es el más simple de los algoritmos de búsqueda. Si bien ciertamente es el más simple, definitivamente no es el más común debido a su ineficiencia. Es un algoritmo de fuerza bruta. Muy rara vez se usa en producción y, en la mayoría de los casos, es superado por otros algoritmos.

La búsqueda lineal no tiene requisitos previos para el estado de la estructura de datos subyacente.

La búsqueda lineal implica la búsqueda secuencial de un elemento en la estructura de datos dada hasta que se encuentra el elemento o se alcanza el final de la estructura.
Si se encuentra el elemento, generalmente solo devolvemos su posición en la estructura de datos. Si no, solemos volver -1.

Implementación

Ahora veamos cómo implementar la búsqueda lineal en Java:

```java
public static int linearSearch(int arr[], int elementToSearch) {

    for (int index = 0; index < arr.length; index++) {
        if (arr[index] == elementToSearch)
            return index;
    }
    return -1;
}
```

Para probarlo, usaremos una matriz simple de enteros:

```java
int index = linearSearch(new int[]{89, 57, 91, 47, 95, 3, 27, 22, 67, 99}, 67);
print(67, index);
```

Con un método de ayuda simple para imprimir el resultado:

```java
public static void print(int elementToSearch, int index) {
    if (index == -1){
        System.out.println(elementToSearch + " not found.");
    }
    else {
        System.out.println(elementToSearch + " found at index: " + index);
    }
}
```

Producción:

```
67 found at index: 8
```

Análisis de eficiencia:
La búsqueda secuencial es un algoritmo simple pero lineal en términos de tiempo, lo que significa que su eficiencia puede disminuir significativamente en grandes conjuntos de datos. Si se dispone de información adicional sobre la estructura de datos, como una lista ordenada, otros algoritmos de búsqueda más eficientes, como la búsqueda binaria, pueden ser más apropiados.

Análisis de casos:

- Mejor caso: El elemento buscado se encuentra al principio de la lista/arreglo/archivo, lo que resulta en una eficiencia O(1) ya que se encuentra en el primer elemento comparado.
- Caso medio: En promedio, el elemento buscado se encuentra alrededor del centro de la lista/arreglo/archivo. En este caso, la eficiencia es O(n/2), lo cual se aproxima a O(n).
- Peor caso: El elemento buscado se encuentra al final de la lista/arreglo/archivo o no se encuentra en absoluto. En este caso, se debe recorrer toda la lista/arreglo/archivo, lo que resulta en una eficiencia O(n), donde "n" es el número de elementos.

Complejidad espacial y temporal:
Complejidad en tiempo: La complejidad en tiempo del algoritmo de búsqueda secuencial es lineal, O(n), donde "n" representa el tamaño de la lista o arreglo en el que se realiza la búsqueda. Esto se debe a que, en el peor caso, se debe recorrer todos los elementos de la lista hasta llegar al elemento buscado o hasta el final de la lista. En el mejor caso, si el elemento buscado está al principio de la lista, la búsqueda puede finalizar rápidamente, pero aun así se considera una complejidad lineal.

Complejidad en espacio: La complejidad en espacio del algoritmo de búsqueda secuencial es constante, O(1), ya que no requiere utilizar espacio adicional en función del tamaño de la lista. Solo se necesitan variables adicionales para mantener el índice actual y los valores temporales durante la búsqueda, pero estos requerimientos de espacio son constantes y no dependen del tamaño de la lista.

Búsqueda Binaria
La búsqueda binaria o logarítmica es uno de los algoritmos de búsqueda más utilizados principalmente debido a su rápido tiempo de búsqueda.

Este tipo de búsqueda utiliza la metodología Divide and Conquer y requiere que el conjunto de datos se ordene de antemano.

Divide la colección de entrada en mitades iguales, y con cada iteración compara el elemento objetivo con el elemento en el medio.

Si se encuentra el elemento, la búsqueda finaliza. De lo contrario, continuamos buscando el elemento dividiendo y seleccionando la partición apropiada de la matriz, en función de si el elemento objetivo es más pequeño o más grande que el elemento del medio.

Por eso es importante tener una colección ordenada para la búsqueda binaria.
La búsqueda termina cuando firstIndex(nuestro puntero) pasa lastIndex(el último elemento), lo que implica que hemos buscado en toda la matriz y el elemento no está presente.
Hay dos formas de implementar este algoritmo: iterativo y recursivo .
No debería haber una diferencia con respecto a la complejidad del tiempo y el espacio entre estas dos implementaciones, aunque esto no es cierto para todos los idiomas.

Implementación
Iterativo
Primero echemos un vistazo al enfoque iterativo:

```
public static int binarySearch(int arr[], int elementToSearch) {

    int firstIndex = 0;
    int lastIndex = arr.length - 1;

    // termination condition (element isn't present)
    while(firstIndex <= lastIndex) {
        int middleIndex = (firstIndex + lastIndex) / 2;
        // if the middle element is our goal element, return its index
        if (arr[middleIndex] == elementToSearch) {
            return middleIndex;
        }

        // if the middle element is smaller
        // point our index to the middle+1, taking the first half out of consideration
        else if (arr[middleIndex] < elementToSearch)
            firstIndex = middleIndex + 1;

        // if the middle element is bigger
        // point our index to the middle-1, taking the second half out of consideration
        else if (arr[middleIndex] > elementToSearch)
            lastIndex = middleIndex - 1;

    }
    return -1;
}
```

```
int index = binarySearch(new int[]{89, 57, 91, 47, 95, 3, 27, 22, 67, 99}, 67);
print(67, index);
```

Producción:

```
67 found at index: 5
```

Recursivo

Y ahora echemos un vistazo a la implementación recursiva:

```
public static int recursiveBinarySearch(int arr[], int firstElement, int lastElement, int elementToSearch) {

    // termination condition
    if (lastElement >= firstElement) {
        int mid = firstElement + (lastElement - firstElement) / 2;

        // if the middle element is our goal element, return its index
        if (arr[mid] == elementToSearch)
            return mid;

        // if the middle element is bigger than the goal element
        // recursively call the method with narrowed data
        if (arr[mid] > elementToSearch)
            return recursiveBinarySearch(arr, firstElement, mid - 1, elementToSearch);

        // else, recursively call the method with narrowed data
        return recursiveBinarySearch(arr, mid + 1, lastElement, elementToSearch);
    }

    return -1;
}
```

La diferencia en el enfoque recursivo es que invocamos el mismo método una vez que obtenemos la nueva partición. En el enfoque iterativo, cada vez que determinamos la nueva partición, modificamos el primer y el último elemento y repetimos el proceso en el mismo ciclo.

Otra diferencia aquí es que las llamadas recursivas se insertan en la pila de llamadas del método y ocupan una unidad de espacio por llamada recursiva.
Podemos usar este algoritmo así:

```
int index = binarySearch(new int[]{3, 22, 27, 47, 57, 67, 89, 91, 95, 99}, 0, 10, 67);
print(67, index);
```

Producción:

```
67 found at index: 5
```

Análisis de eficiencia:
La búsqueda binaria es un algoritmo altamente eficiente para buscar elementos en una lista ordenada, con una complejidad en el tiempo logarítmica O(log n). Su eficiencia se basa en la reducción del rango de búsqueda a la mitad en cada iteración. Además, la complejidad espacial es constante O(1), lo que significa que no se necesita espacio adicional proporcional al tamaño de la lista.

Análisis de los casos:

- Mejor caso: El elemento buscado está en el centro de la lista. La búsqueda binaria encuentra el elemento en el primer paso, lo que resulta en una eficiencia constante, O(1).
- Caso medio: En promedio, la búsqueda binaria divide a la mitad el rango de búsqueda en cada paso. Esto resulta en una eficiencia logarítmica, O(log n), donde "n" es el número de elementos en la lista.
- Peor caso: El elemento buscado no está presente en la lista. La búsqueda binaria debe recorrer todo el rango de búsqueda, lo que resulta en una eficiencia logarítmica, O(log n).

Complejidad en el tiempo y complejidad espacial:

- Complejidad en el tiempo: La búsqueda binaria tiene una complejidad en el tiempo de O(log n), donde "n" es el número de elementos en la lista. Esto se debe a que en cada paso, el rango de búsqueda se reduce a la mitad, lo que permite encontrar el elemento buscado en un número reducido de comparaciones.
- Complejidad espacial: La complejidad espacial de la búsqueda binaria es constante, O(1), ya que no se requiere espacio adicional en función del tamaño de la lista. Solo se necesitan variables adicionales para mantener el rango de búsqueda actual.

Búsqueda de patrones de Knuth Morris Pratt
Como su nombre lo indica, es un algoritmo para encontrar un patrón en el texto dado. Este algoritmo fue desarrollado por Donald Knuth, Vaughan Pratt y James Morris, de ahí el nombre.

En esta búsqueda, primero se compila el patrón dado . Al compilarlo, tratamos de encontrar el prefijo y el sufijo de la cadena de patrón. Esto nos ayuda cuando ocurre una discrepancia: no comenzaremos a buscar la siguiente coincidencia desde el comienzo del índice.

En cambio, omitimos la parte de la cadena de texto que ya hemos comparado y comenzamos a comparar más allá de esa parte. Determinamos esta parte conociendo el prefijo y el sufijo para estar seguros de qué parte ya se comparó y se puede omitir de manera segura.

Como resultado de esta omisión, podemos ahorrar muchas comparaciones y KMP funciona más rápido que un algoritmo de fuerza bruta ingenuo.

Implementación
Vamos a crear el compilePatternArray()método, que será utilizado más tarde por el algoritmo de búsqueda KMP:

```
public static int[] compilePatternArray(String pattern) {
    int patternLength = pattern.length();
    int len = 0;
    int i = 1;
    int[] compliedPatternArray = new int[patternLength];
    compliedPatternArray[0] = 0;

    while (i < patternLength) {
        if (pattern.charAt(i) == pattern.charAt(len)) {
            len++;
            compliedPatternArray[i] = len;
            i++;
        } else {
            if (len != 0) {
                len = compliedPatternArray[len - 1];
            } else {
                compliedPatternArray[i] = len;
                i++;
            }
        }
    }
    System.out.println("Compiled Pattern Array " + Arrays.toString(compliedPatternArra
    return compliedPatternArray;
}
```

La matriz de patrones compilada se puede considerar como una matriz que almacena el patrón de caracteres en la matriz de patrones. El objetivo principal detrás de la creación de esta matriz es encontrar el prefijo y el sufijo en el patrón. Si conocemos estos elementos en el patrón, podemos evitar la comparación desde el comienzo del texto y simplemente comparar el siguiente carácter después de que haya ocurrido la falta de coincidencia.

La matriz compilada almacena la posición de índice de la aparición anterior del carácter actual en la matriz de patrones.

Implementemos el algoritmo en sí:

```
public static List<Integer> performKMPSearch(String text, String pattern) {
    int[] compliedPatternArray = compilePatternArray(pattern);

    int textIndex = 0;
    int patternIndex = 0;

    List<Integer> foundIndexes = new ArrayList<>();

    while (textIndex < text.length()) {
        if (pattern.charAt(patternIndex) == text.charAt(textIndex)) {
            patternIndex++;
            textIndex++;
        }
        if (patternIndex == pattern.length()) {
            foundIndexes.add(textIndex - patternIndex);
            patternIndex = compliedPatternArray[patternIndex - 1];
        }

        else if (textIndex < text.length() && pattern.charAt(patternIndex) != text.charAt(textIn
            if (patternIndex != 0)
                patternIndex = compliedPatternArray[patternIndex - 1];
            else
                textIndex = textIndex + 1;
        }
    }
    return foundIndexes;
}
```

Aquí comenzamos comparando secuencialmente los caracteres en el patrón y la matriz de texto. Seguimos avanzando hasta que seguimos obteniendo una coincidencia de patrones y matrices de texto. De esta forma, si llegamos al final de la matriz de patrones mientras lo emparejamos, significa que hemos encontrado una ocurrencia del patrón en el texto.

Sin embargo, sí encontramos una discrepancia al comparar las dos matrices, movemos el índice de la matriz de caracteres del patrón al valor en compiledPatternArray()y también pasamos al siguiente carácter en la matriz de texto. Aquí es donde la búsqueda

KMP supera al enfoque de fuerza bruta, ya que no compara los caracteres de texto más de una vez si no coinciden.

Intentemos ejecutar el algoritmo:

```
String pattern = "AAABAAA";
String text = "ASBNSAAAAAABAAAAABAAAAAGAHUHDJKDDKSHAAJF";

List<Integer> foundIndexes = KnuthMorrisPrathPatternSearch.performKMPSearch(text, pattern);

if (foundIndexes.isEmpty()) {
    System.out.println("Pattern not found in the given text String");
} else {
    System.out.println("Pattern found in the given text String at positions: " + .stream().map(O
}
```

En el texto del patrón AAABAAA, se observa el siguiente patrón y se codifica en la matriz de patrones:

- El patrón A(Single A) se repite en el índice 1 y nuevamente en el 4.
- El patrón AA(Doble A) se repite en el índice 2 y nuevamente en el índice 5.
- El patrón AAA(3 A) se repite en el índice 6.

Veamos el resultado para validar nuestra discusión hasta ahora:

```
Compiled Pattern Array [0, 1, 2, 0, 1, 2, 3]
Pattern found in the given text String at positions: 8, 14
```

El patrón que describimos se nos muestra claramente en la matriz de patrones cumplida en la salida.

Con la ayuda de esta matriz compilada, el algoritmo de búsqueda de KMP puede buscar el patrón dado en el texto sin retroceder en la matriz de texto.

Análisis de eficiencia:

La búsqueda de patrones de Knuth-Morris-Pratt ofrece un rendimiento eficiente con una complejidad en tiempo de O(n + m) y una complejidad espacial de O(m). El preprocesamiento del patrón permite evitar comparaciones innecesarias durante la búsqueda y acelera el proceso de encontrar ocurrencias del patrón en el texto. Esto hace que el algoritmo de KMP sea una opción preferida cuando se trata de buscar patrones en textos extensos.

Análisis de casos

- Mejor caso: El mejor caso ocurre cuando el patrón no está presente en el texto. En este caso, la búsqueda de patrones de KMP tiene una complejidad en el tiempo de O(n), donde "n" es la longitud del texto. Esto se debe a que se recorre todo el texto una vez sin encontrar coincidencias.
- Caso medio: En promedio, la búsqueda de patrones de KMP tiene una complejidad en el tiempo de O(n + m), donde "n" es la longitud del texto y "m" es la longitud del patrón. Esto se debe a que se realiza el preprocesamiento del patrón en O(m) y luego se recorre el texto realizando comparaciones en O(n), utilizando el arreglo "lps" para evitar comparaciones innecesarias.
- Peor caso: El peor caso ocurre cuando todas las posiciones del texto y el patrón deben ser comparadas en cada iteración. En este caso, la complejidad en el tiempo es O(n + m), similar al caso medio.

Complejidad en el tiempo y complejidad espacial:

- Complejidad en el tiempo: La búsqueda de patrones de KMP tiene una complejidad en el tiempo de O(n + m), donde "n" es la longitud del texto y "m" es la longitud del patrón. Esto se debe a que el preprocesamiento del patrón requiere O(m) operaciones y la búsqueda en el texto requiere O(n) operaciones en el peor caso.
- Complejidad espacial: La complejidad espacial de la búsqueda de patrones de KMP es O(m), donde "m" es la longitud del patrón. Esto se debe a que se utiliza el arreglo "lps" de longitud m para almacenar información sobre los prefijos que también son sufijos.

Método Saltar búsqueda
Esta búsqueda es similar a la búsqueda binaria, pero en lugar de saltar tanto hacia adelante como hacia atrás, solo saltaremos hacia adelante. Tenga en cuenta que Jump Search también requiere que se ordene la colección.

En Jump Search, saltamos en el intervalo sqrt(arraylength)anterior hasta llegar a un elemento mayor que el elemento actual o el final de la matriz. En cada salto, se registra el paso anterior.

Si nos encontramos con un elemento mayor que el elemento que estamos buscando, dejamos de saltar. Luego, ejecutamos una búsqueda lineal entre el paso anterior y el paso actual.
Esto hace que el espacio de búsqueda sea mucho más pequeño para la búsqueda lineal y, por lo tanto, se convierte en una opción viable.

Implementación:

```
public static int jumpSearch(int[] integers, int elementToSearch) {

    int arrayLength = integers.length;
    int jumpStep = (int) Math.sqrt(integers.length);
    int previousStep = 0;

    while (integers[Math.min(jumpStep, arrayLength) - 1] < elementToSearch) {
        previousStep = jumpStep;
        jumpStep += (int)(Math.sqrt(arrayLength));
        if (previousStep >= arrayLength)
            return -1;
    }
    while (integers[previousStep] < elementToSearch) {
        previousStep++;
        if (previousStep == Math.min(jumpStep, arrayLength))
            return -1;
    }

    if (integers[previousStep] == elementToSearch)
        return previousStep;
    return -1;
}
```

Comenzamos con la jumpstepraíz cuadrada del tamaño de la longitud de la matriz y seguimos saltando hacia adelante con este mismo tamaño hasta que encontremos un elemento que sea igual o mayor que el elemento que estamos buscando.

Primero visitamos el elemento en integers[jumpStep], luego integers[2jumpStep], integers[3jumpStep]y así sucesivamente. También almacenamos el elemento anterior visitado en la previousStepvariable.

Una vez que encontramos un valor tal que integers[previousStep]< elementToSearch< integers[jumpStep], realizamos una búsqueda lineal entre integers[previousStep]y integers[jumpStep]o un elemento mayor que elementToSearch.
Podemos usar el algoritmo así:

```
int index = jumpSearch(new int[]{3, 22, 27, 47, 57, 67, 89, 91, 95, 99}, 67);
print(67, index);
```

Producción:

```
67 found at Index 5
```

Análisis de eficiencia:

Es un algoritmo eficiente en términos de comparaciones realizadas, especialmente en matrices grandes y ordenadas. Sin embargo, en promedio, puede requerir más comparaciones que la búsqueda binaria, lo que lo hace menos eficiente en ese aspecto.

Análisis de casos:

- Mejor caso: El mejor caso ocurre cuando el elemento buscado está en la primera posición del intervalo inicial. En este caso, la complejidad en tiempo es O(1), ya que se encuentra el elemento en la primera comparación.

- Caso medio: En promedio, el número de comparaciones realizadas en Jump Search es menor que en la búsqueda lineal pero mayor que en la búsqueda binaria. La complejidad en tiempo es O(sqrt(n)), donde "n" es la longitud de la matriz. Esto se debe a que el algoritmo realiza saltos en intervalos de tamaño sqrt(n) y luego realiza una búsqueda lineal en el intervalo actual.

- Peor caso: El peor caso ocurre cuando el elemento buscado es el último elemento del intervalo o no está presente en la matriz. La complejidad en tiempo sigue siendo O(sqrt(n)), ya que se requieren saltos y búsquedas lineales en cada intervalo.

Complejidad temporal y espacial:

La complejidad espacial del algoritmo Jump Search es O(1), lo que significa que no requiere espacio adicional más allá de las variables utilizadas para realizar los saltos y las comparaciones. No se requiere memoria adicional en función del tamaño de la matriz. La complejidad temporal del algoritmo Jump Search es O(sqrt(n)), donde "n" es el tamaño de la matriz. Esto indica que el tiempo requerido para encontrar un elemento utilizando Jump Search aumenta proporcionalmente a la raíz cuadrada del tamaño de la matriz. Es más eficiente que la búsqueda lineal, pero menos eficiente que la búsqueda binaria, que tiene una complejidad temporal de O(log n).

Búsqueda de interpolación

La búsqueda por interpolación se utiliza para buscar elementos en una matriz ordenada. Esta búsqueda es particularmente útil si sabemos que los datos en la estructura subyacente están distribuidos uniformemente.

Si los datos se distribuyen uniformemente, adivinar la ubicación de un elemento puede ser más preciso, a diferencia de la búsqueda binaria, en la que siempre tratamos de encontrar el elemento en el medio de la matriz.

La búsqueda de interpolación utiliza fórmulas de interpolación para encontrar el lugar más probable donde se puede encontrar el elemento en la matriz. Sin embargo, para que estas fórmulas sean efectivas, la matriz de búsqueda debe ser grande; de lo contrario, funciona como la búsqueda lineal:

```
public static int interpolationSearch(int[] integers, int elementToSearch) {

    int startIndex = 0;
    int lastIndex = (integers.length - 1);

    while ((startIndex <= lastIndex) && (elementToSearch >= integers[startIndex]) &&
           (elementToSearch <= integers[lastIndex])) {
        // using interpolation formulae to find the best probable position for this el
        int pos = startIndex + (((lastIndex-startIndex) /
          (integers[lastIndex]-integers[startIndex]))*
                        (elementToSearch - integers[startIndex]));

        if (integers[pos] == elementToSearch)
            return pos;

        if (integers[pos] < elementToSearch)
            startIndex = pos + 1;

        else
            lastIndex = pos - 1;
    }
    return -1;
}
```

Podemos usar este algoritmo así:

```
int index = interpolationSearch(new int[]{1,2,3,4,5,6,7,8}, 6);
print(67, index);
```

Producción:

```
6 found at Index 5
```

Echemos un vistazo a cómo las fórmulas de interpolación hacen su magia para buscar 6:

```
startIndex = 0
lastIndex = 7
integers[lastIndex] = 8
integers[startIndex] = 1
elementToSearch = 6
```

Ahora apliquemos estos valores a las fórmulas para estimar el índice del elemento de búsqueda:

$$inortedmiX = 0 + (7 - 0)/(8 - 1) * (6 - 1) = 5$$

El elemento en integers[5]es 6, que es el elemento que buscábamos. Como podemos ver aquí, el índice del elemento se calcula en un solo paso ya que los datos se distribuyen uniformemente.

Análisis de eficiencia:
La búsqueda de interpolación puede ser eficiente en situaciones donde la distribución de los valores en la colección es uniforme, ya que puede proporcionar una estimación precisa de la posición del elemento buscado. Sin embargo, su rendimiento puede degradarse en casos de distribuciones no uniformes, lo que puede resultar en una complejidad temporal similar a la búsqueda lineal en el peor caso.
Análisis de casos:

- Mejor caso: El mejor caso ocurre cuando el elemento buscado se encuentra en el medio de la colección. En este caso, la búsqueda de interpolación puede encontrar el elemento en tiempo constante O(1).
- Caso medio: En el caso medio, el rendimiento de la búsqueda de interpolación depende de la distribución de los valores en la colección. Si la distribución es aproximadamente uniforme, el algoritmo tiene una complejidad temporal de O(log(log(n))), donde "n" es el tamaño de la colección. Sin embargo, si la distribución no es uniforme, el rendimiento puede degradarse a O(n).
- Peor caso: El peor caso ocurre cuando los valores en la colección están muy dispersos y no siguen una distribución uniforme. En este caso, la búsqueda de interpolación puede tener un rendimiento similar a la búsqueda lineal, con una complejidad temporal de O(n).

Complejidad en el tiempo y complejidad espacial:

- La complejidad en el tiempo de la búsqueda de interpolación en el caso promedio es O(log(log(n))) si la distribución es uniforme, pero puede llegar a ser O(n) en casos extremos de distribución no uniforme.
- La complejidad espacial es O(1) ya que no se requiere espacio adicional más allá de las variables utilizadas para realizar las operaciones de búsqueda.

Búsqueda exponencial
La búsqueda exponencial se utiliza para buscar elementos saltando en posiciones exponenciales, es decir, en potencias de 2.

En esta búsqueda, básicamente estamos tratando de encontrar un rango comparativamente más pequeño en el que podamos buscar el elemento utilizando otros algoritmos de búsqueda acotados como la búsqueda binaria.

No hace falta decir que la colección debe ordenarse para que esto funcione.

Implementación:

```
public static int exponentialSearch(int[] integers, int elementToSearch) {

    if (integers[0] == elementToSearch)
        return 0;
    if (integers[integers.length - 1] == elementToSearch)
        return integers.length;

    int range = 1;

    while (range < integers.length && integers[range] <= elementToSearch) {
        range = range * 2;
    }

    return Arrays.binarySearch(integers, range / 2, Math.min(range, integers.length),
}
```

Podemos usar este algoritmo así:

Así es como funciona el algoritmo:
Intentamos encontrar un elemento que sea mayor que el elemento que estamos buscando. Hacemos esto para minimizar el rango de elementos que estamos buscando. Aumentamos el rango multiplicándolo por 2 y verificamos nuevamente si llegamos a un elemento mayor que el elemento que estamos buscando o al final de la matriz. Una vez que se logra cualquiera de esto, salimos del bucle. Luego realizamos una búsqueda binaria con startIndexas range/2y lastIndexas range.

```
int index = exponentialSearch(new int[]{3, 22, 27, 47, 57, 67, 89, 91, 95, 99}, 67);
print(67, index);
```

En nuestro caso, este valor de rango se logra en 8 y el elemento en integers[8]es 95. Entonces, el rango donde realizamos la búsqueda binaria es:

```
startIndex = range/2 = 4

lastIndex = range = 8
```

Con esto la llamada de búsqueda binaria se convierte en:

```
Arrays.binarySearch(integers, 4, 8, 6);
```

Producción:

```
67 found at Index 5
```

Una cosa importante a tener en cuenta aquí es que podemos acelerar la multiplicación por 2 usando el operador de desplazamiento a la izquierda range << 1en lugar del *operador.

Análisis de eficiencia:
En general, la búsqueda exponencial es eficiente en la mayoría de los casos, especialmente en colecciones ordenadas. Sin embargo, su rendimiento puede ser afectado en el peor caso cuando el elemento buscado está al final de la colección o no está presente.

Análisis de los casos: mejor de los casos, caso medio y peor de los casos con ejemplos:

- Mejor caso: El mejor caso ocurre cuando el elemento buscado se encuentra en la posición 0 del arreglo. En este caso, la búsqueda exponencial puede encontrar el elemento en tiempo constante O(1), ya que no se necesita realizar ninguna iteración adicional.
- Caso medio: En el caso medio, la búsqueda exponencial realiza saltos exponenciales para determinar el rango de búsqueda, seguido de una búsqueda binaria dentro de ese rango. En promedio, tiene una complejidad temporal de O(log(n)), donde "n" es el tamaño de la colección. Esto significa que el tiempo de ejecución crece de manera logarítmica a medida que aumenta el tamaño de la colección.
- Peor caso: El peor caso ocurre cuando el elemento buscado se encuentra al final del arreglo o no está presente en él. En este caso, la búsqueda exponencial realiza saltos exponenciales hasta que el límite superior del rango supera al elemento buscado, y luego realiza una búsqueda binaria entre los límites inferior y superior. En el peor caso, la complejidad temporal es O(log(n)), donde "n" es el tamaño de la colección.
- Complejidad espacial: La búsqueda exponencial tiene una complejidad espacial de O(1), ya que no requiere espacio adicional más allá de las variables utilizadas para realizar las operaciones de búsqueda.

Complejidad en el tiempo y complejidad espacial.
La complejidad en el tiempo en el peor caso es O(log(n)), donde "n" es el tamaño de la colección. Esto significa que el tiempo de ejecución crece de manera logarítmica a medida que aumenta el tamaño de la colección.

La complejidad espacial, la búsqueda exponencial tiene una complejidad espacial de O(1), ya que no requiere espacio adicional en relación con el tamaño de la colección. Solo se utilizan variables adicionales para almacenar información relacionada con la búsqueda, como índices y límites.

Búsqueda de Fibonacci
La búsqueda de Fibonacci emplea el enfoque divide y vencerás en el que dividimos elementos de manera desigual según la serie de Fibonacci. Esta búsqueda requiere que la matriz esté ordenada.

A diferencia de la búsqueda binaria, donde dividimos los elementos en mitades iguales para reducir el rango de la matriz, en la búsqueda de Fibonacci intentamos usar la suma o la resta para obtener un rango más pequeño.

Recuerda que la fórmula de la serie de Fibonacci es:

$$Fibo(norte) = Fibo(norte - 1) + Fibo(norte - 2)$$

Los dos primeros números de esta serie son Fibo(0) = 0y Fibo(1) = 1. Entonces, según esta fórmula, la serie se ve así 0, 1, 1, 2, 3, 5, 8, 13, 21... Las observaciones interesantes a tener en cuenta aquí son las siguientes:

Fibo(N-2)es aproximadamente 1/3 deFibo(N)
Fibo(N-1)es aproximadamente 2/3 deFibo(N)

Entonces, cuando usamos los números de la serie de Fibonacci para dividir el rango, se divide en la misma proporción que arriba.

Implementación

Echemos un vistazo a la implementación para tener una idea más clara:

```
public static int fibonacciSearch(int[] integers, int elementToSearch) {

    int fibonacciMinus2 = 0;
    int fibonacciMinus1 = 1;
    int fibonacciNumber = fibonacciMinus2 + fibonacciMinus1;
    int arrayLength = integers.length;

    while (fibonacciNumber < arrayLength) {
        fibonacciMinus2 = fibonacciMinus1;
        fibonacciMinus1 = fibonacciNumber;
        fibonacciNumber = fibonacciMinus2 + fibonacciMinus1;
    }

    int offset = -1;

    while (fibonacciNumber > 1) {
        int i = Math.min(offset+fibonacciMinus2, arrayLength-1);

        if (integers[i] < elementToSearch) {
            fibonacciNumber = fibonacciMinus1;
            fibonacciMinus1 = fibonacciMinus2;
            fibonacciMinus2 = fibonacciNumber - fibonacciMinus1;
            offset = i;
        }

        else if (integers[i] > elementToSearch) {
            fibonacciNumber = fibonacciMinus2;
            fibonacciMinus1 = fibonacciMinus1 - fibonacciMinus2;
            fibonacciMinus2 = fibonacciNumber - fibonacciMinus1;
        }

        else return i;
    }

    if (fibonacciMinus1 == 1 && integers[offset+1] == elementToSearch)
        return offset+1;

    return -1;
}
```

Podemos ejecutar este algoritmo así:

```
int index = exponentialSearch(new int[]{3, 22, 27, 47, 57, 67, 89, 91, 95, 99}, 67);
print(67, index);
```

Así es como funciona el algoritmo:

Comienza por encontrar primero el número en la serie de Fibonacci más cercano pero mayor que la longitud de la matriz. Esto sucede cuando fibonacciNumberestá en 13, que es solo más que la longitud de la matriz: 10.

A continuación, comparamos los elementos de la matriz y, sobre la base de esa comparación, tomamos una de las siguientes acciones:

- Compare el elemento a buscar con el elemento en fibonacciMinus2y devuelva el índice si el valor coincide.
- Si elementToSearches mayor que el elemento actual, retrocedemos un paso en la serie de Fibonacci y cambiamos los valores de fibonacciNumber, fibonacciMinus1& fibonacciMinus2en consecuencia. El desplazamiento se restablece al índice actual.
- Si elementToSearches más pequeño que el elemento actual, retrocedemos dos pasos en la serie de Fibonacci y cambiamos los valores de fibonacciNumber, fibonacciMinus1& fibonacciMinus2en consecuencia.

Producción:

```
67 found at Index 5
```

Análisis de eficiencia:
la búsqueda de Fibonacci es eficiente en la mayoría de los casos, especialmente en colecciones ordenadas. El mejor caso tiene una complejidad temporal de O(1), mientras que el caso medio y el peor caso tienen una complejidad temporal de aproximadamente O(log(n)).

Análisis de los casos: mejor de los casos, caso medio y peor de los casos con ejemplos:

- Mejor caso: El mejor caso ocurre cuando el elemento buscado se encuentra en la primera posición de la colección. En este caso, la búsqueda de Fibonacci puede encontrar el elemento de manera inmediata sin necesidad de realizar ninguna iteración adicional. La complejidad temporal en el mejor caso es O(1), lo cual lo convierte en el caso más eficiente.
- Caso medio: En el caso medio, la búsqueda de Fibonacci realiza una serie de saltos exponenciales y comparaciones para acercarse al elemento buscado. El número de iteraciones necesarias en el caso medio depende de la ubicación del elemento en la secuencia de Fibonacci. La complejidad temporal en el caso medio puede aproximarse a O(log(n)), donde "n" es el tamaño de la colección. Esto

significa que el tiempo de ejecución crece de manera logarítmica a medida que aumenta el tamaño de la colección, pero con una dependencia en la secuencia de Fibonacci.

- Peor caso: El peor caso ocurre cuando el elemento buscado está al final de la colección o no está presente en ella. En este caso, la búsqueda de Fibonacci puede requerir un mayor número de iteraciones para acercarse al elemento buscado, ya que la secuencia de Fibonacci puede no ser lo suficientemente grande para cubrir todo el rango de búsqueda. La complejidad temporal en el peor caso es O(log(n)), donde "n" es el tamaño de la colección. Aunque sigue siendo eficiente en comparación con la búsqueda lineal, el rendimiento puede degradarse en el peor caso.

Complejidad en el tiempo y complejidad espacial:

- Complejidad temporal: La complejidad temporal en el peor caso de la búsqueda de Fibonacci es O(log(n)), donde "n" es el tamaño de la colección. Esto se debe a que en cada iteración, los índices se ajustan utilizando la secuencia de Fibonacci, lo que permite realizar saltos exponenciales en lugar de lineales o binarios. Esto reduce significativamente el número de comparaciones necesarias y acelera la búsqueda.
- Complejidad espacial: La búsqueda de Fibonacci tiene una complejidad espacial de O(1), ya que no requiere espacio adicional más allá de las variables utilizadas para realizar las operaciones de búsqueda.