INSTITÚTO TECNOLÓGICO DE MEXICO

CAMPUS ORIZABA

NOMBRE DE LA ASIGNATURA:
ESTRUCTURA DE DATOS

TEMA:
REPORTE DE PRACTICA TEMA 6 MÉTODOS DE BÚSQUEDA

NOMBRE DE LOS INTEGRANTES:
HERNÁNDEZ DÍAZ ARIEL - 21010195
PELACIOS MENDEZ XIMENA MONSERRAT – 21010209
VELAZQUEZ SARMIENTO CELESTE - 21010223
ESPINDOLA OLIVERA PALOMA - 21010186

NOMBRE DE LA PROFESORA:
MARIA JACINTA MARTINEZ CASTILLO

HORARIO OFICIAL DE LA CLASE:
15:00 – 16:00 HRS

PERIODO:
ENERO - JUNIO 2023

INTRODUCCIÓN

Con mucha frecuencia los programadores trabajan con grandes cantidades de datos almacenados en arrays y registros, y por ello será necesario determinar si un array contiene un valor que coincida con un cierto valor clave. El proceso de encontrar un elemento específico de un arrays se denomina búsqueda. La operación de búsqueda nos permite encontrar datos que están previamente almacenados. La operación puede ser un éxito, si se localiza el elemento buscado o un fracaso en otros casos.

La búsqueda se puede realizar sobre un conjunto de datos ordenados, lo cual hace la tarea más fácil y consume menos tiempo; o se puede realizar sobre elementos desordenados, tarea más laboriosa y de mayor insumo de tiempo.

Una de las funciones que con mayor frecuencia se utiliza en los sistemas de información, es el de las consultas a los datos, se hace necesario utilizar algoritmos, que permitan realizar búsquedas de forma rápida y eficiente. La búsqueda, se puede decir que es la acción de recuperar datos o información, siendo una de las actividades que más aplicaciones tiene en los sistemas de información.

Más formalmente se puede definir como “La operación de búsqueda sobre una estructura de datos es aquella que permite localizar un nodo en particular si es que éste existe” (Euán, 1989).

COMPETENCIA(S) A DESARROLLAR

Conoce, comprende y aplica los algoritmos de ordenamiento para el uso adecuado en el desarrollo de aplicaciones que permita solucionar problemas del entorno.

MARCO TEORICO

Búsqueda lineal

la búsqueda secuencial es un método para encontrar un valor objetivo dentro de una lista. Ésta comprueba secuencialmente cada elemento de la lista para el valor objetivo hasta que es encontrado o hasta que todos los elementos hayan sido comparados.

Búsqueda lineal es en tiempo el peor, y marca como máximo n comparaciones, donde n es la longitud de la lista. Si la probabilidad de cada elemento para ser buscado es el mismo, entonces la búsqueda lineal tiene una media de n/2 comparaciones, pero esta media puede ser afectado si las probabilidades de búsqueda para cada elemento varían. La búsqueda lineal es poco práctica porque otros algoritmos de búsqueda y esquemas, como el algoritmo de búsqueda binaria y Tabla hash, es significativamente más rápido buscando todo menos listas cortas.

La búsqueda secuencial consiste en recorrer secuencialmente un array desde el primer elemento hasta el último y comprobar si alguno de los elementos del array contiene el vector buscado, es decir, comparar cada elemento del array con el valor buscado. Dado que el array no está en ningún orden particular, existe la misma probabilidad de que el valor se encuentre, ya sea en el primer elemento como en el último. Por tanto, en promedio, el programa tendrá que comparar el valor buscado con la mitad de los elementos del arreglo.

La búsqueda secuencial requiere, para el peor de los casos, cuando el elemento a buscar es el último o no se encuentra, recorrer todo el vector y realizar un número de comparaciones igual al tamaño del vector, de lo que deducimos que para vectores con muchos elementos esta búsqueda puede no ser conveniente.

El método de búsqueda lineal funciona bien para arrays pequeños o para arrays no ordenados. Si el array está ordenado, se puede utilizar la técnica de alta velocidad de búsqueda binaria.

Ejemplo de búsqueda lineal

Supongamos que tenemos el array: (5, 3, 4, 2, 1, 6).

Caso 1: Queremos buscar X = 5.

El primer elemento en sí es una coincidencia y devolvemos el índice 0. (Mejor caso)

Caso 2: Queremos buscar X = 1.

Atravesamos el array y llegamos al índice 4 para encontrar una coincidencia y devolver ese índice. (Caso promedio)

Caso 3: Queremos buscar X = 9

Atravesamos el array pero no encontramos una coincidencia cuando llegamos al último elemento del array. Devolvemos -1. (Peor de los casos)

Búsqueda binaria

Se llama también dicotonámica. Se utiliza cuando el vector en el que queremos determinar la existencia de un elemento está previamente ordenado. Este algoritmo reduce el tiempo de búsqueda considerablemente, ya que disminuye exponencialmente el número de iteraciones necesarias.

Está altamente recomendado para buscar en arrays de gran tamaño. Por ejemplo, en uno conteniendo 50.000.000 elementos, realiza como máximo 26 comparaciones (en el peor de los casos).

Para implementar este algoritmo se compara el elemento a buscar con un elemento cualquiera del array (normalmente el elemento central): si el valor de éste es mayor que el del elemento buscado se repite el procedimiento en la parte del array que va desde el inicio de éste hasta el elemento tomado, en caso contrario se toma la parte del array que va desde el elemento tomado hasta el final. De esta manera obtenemos intervalos cada vez más pequeños, hasta que se obtenga un intervalo indivisible. Si el elemento no se encuentra dentro de este último entonces se deduce que el elemento buscado no se encuentra en todo el array.

Búsqueda de patrones de Knuth Morris Pratt

El algoritmo KMP es un algoritmo de búsqueda de subcadenas simple. Por lo tanto su objetivo es buscar la existencia de una subcadena (habitualmente llamada patrón) dentro de otra cadena. La búsqueda se lleva a cabo utilizando información basada en los fallos previos. Información obtenida del propio patrón creando previamente una tabla de valores sobre su propio contenido. El objetivo de crear dicha tabla es poder determinar donde podría darse la siguiente existencia, sin necesidad de analizar más de 1 vez los caracteres de la cadena donde se busca.

En 1970 S.A. Cook sugirió la existencia de un algoritmo que resuelve la búsqueda en un tiempo equivalente a M+N, en el peor caso, a raíz de unos resultados teóricos que obtuvo mientras probaba un tipo de máquina abstracta.1 Knuth y Pratt siguiendo

los pasos que Cook usó para probar su teorema, lograron crear el algoritmo que lleva sus nombres. De modo independiente Morris mientras trabajaba en un editor de texto acabó descubriendo el mismo algoritmo, para satisfacer la cuestión de la búsqueda eficiente de subcadenas, y que publicaron juntos los tres en 1976.

El algoritmo KMP trata de localizar la posición de comienzo de una cadena dentro de otra. Previamente sobre el patrón a localizar se calcula una tabla de saltos (conocida como tabla de fallos) que después se utiliza (al examinar las cadenas) para hacer saltos cuando se localiza un fallo de coincidencia en la comparación de un carácter.

Supongamos una tabla 'F' ya precalculada, y supongamos que el patrón a buscar esté contenido en la matriz 'P()', y la cadena donde buscamos esté contenida en la matriz 'T()'. Entonces ambas cadenas comienzan a compararse usando un puntero de avance para el patrón, si ocurre un fallo (de coincidencia), el puntero salta hacia donde indica el valor en la posición del puntero actual sobre la tabla de fallos . El array 'T' utiliza un puntero de avance absoluto que determina el comienzo de una sección del mismo tamaño que el patrón. Dentro de dicha sección se desplaza con el puntero del patrón, analizando cada carácter en la sección con cada carácter en el patrón. Se dan 2 situaciones:

Mientras existan coincidencias el puntero de avance de 'P', se va incrementando (apunta al siguiente par de caracteres a comparar entre el patrón y la sección actual en el array 'T') y si alcanza el final del patrón se devuelve la posición actual del puntero absoluto del array 'T'.

Si se da un fallo, la sección del array 'T' (el puntero absoluto de avance) se actualiza, sumando el valor del puntero de 'P' + el valor que contiene la tabla 'F' en el mismo índice que 'P'. Además de actualizar la sección bajo examen, debe igualmente alinearse el patrón bajo dicha sección, para coincidir bajo una de 2 circunstancias; Si el valor de 'F' es mayor que -1 el puntero de 'P', toma el valor que indica la tabla de salto 'F', en caso contrario vuelve a recomenzar su valor en 0 (comienzo del patrón y de la siguiente sección).

Saltar búsqueda

Jump Search es un algoritmo de búsqueda por intervalos. Es un algoritmo relativamente nuevo que funciona solo en matrices ordenadas. Intenta reducir el número de comparaciones requeridas que la búsqueda lineal al no escanear cada elemento como la búsqueda lineal. En la búsqueda por salto, el array se divide en bloques m. Busca el elemento en un bloque y, si el elemento no está presente, pasa al siguiente bloque. Cuando el algoritmo encuentra el bloque que contiene el elemento, utiliza el algoritmo de búsqueda lineal para encontrar el índice exacto. Este algoritmo es más rápido que la búsqueda lineal pero más lento que la búsqueda binaria.

Ejemplo de búsqueda con salto

Supongamos que tenemos el array: (1, 2, 3, 4, 5, 6, 7, 8, 9), y queremos encontrar X - 7.

- Como hay 9 elementos, tenemos n como 9.
- Establece i como 0 y m como √9, es decir, 3.
- A[2] es menor que X. Establezca i como 3 y m como 6.
- A[5] es menor que X. Establezca i como 6 y m como 9.
- A[8] es igual a X. Sal del bucle.
- i como 6 es menor que n.
- A[6] == 7. Salir del bucle
- Desde A[6] == 7, devuelve 6.

Búsqueda de interpolación

La búsqueda por interpolación es un algoritmo de búsqueda rápido y eficiente. Mejora el algoritmo de búsqueda binaria para escenarios donde los elementos del array se distribuyen uniformemente sobre el array ordenada. Funciona en la posición de palpación del valor requerido. A diferencia de la búsqueda binaria, no siempre va a la mitad del array, pero puede ir a cualquier posición dependiendo del valor de la clave que se buscará. Comparamos el valor en la posición estimada y reducimos el espacio de búsqueda a la parte posterior o anterior. Por ejemplo, cuando buscamos una palabra en el diccionario, pasamos las páginas de acuerdo con la posición de las letras en su interior y no dividiendo el espacio de búsqueda en dos mitades cada vez.

Ejemplo de búsqueda de interpolación

Supongamos que tenemos el array (1, 3, 7, 8, 11, 15, 17, 18, 21), y queremos encontrar X - 18.

- Establecer lo = 0, mid = -1 y hi = 8.
- Calcula mid como 6 utilizando la fórmula - 0 + (18 - 1)*(8 - 0)/(21-1).
- Luego comparamos A[6] con X para ver que es más pequeño y establecemos lo como 7.
- Calcula mid usando 7 + (18 - 18) * (8 - 7) / (21 - 18).
- Luego comparamos A[7] con X para ver que es igual a 18 y devolvemos el índice 7.

Búsqueda exponencial

La búsqueda exponencial, también conocida como búsqueda duplicada o búsqueda con los dedos, es un algoritmo creado para buscar elementos en matrices de gran

tamaño. Es un proceso de dos pasos. Primero, el algoritmo intenta encontrar el rango (L, R) en el que está presente el elemento objetivo y luego utiliza la búsqueda binaria dentro de este rango para encontrar la ubicación exacta del objetivo.

Se denomina búsqueda exponencial porque encuentra el elemento que contiene el rango buscando el primer exponente k para el que el elemento en el índice pow(2,k) es mayor que el objetivo. Aunque su nombre es búsqueda exponencial, la complejidad temporal de este algoritmo es logarítmica. Es muy útil cuando los Arrays son de tamaño infinito y convergen a una solución mucho más rápido que la búsqueda binaria.

Ejemplo de búsqueda exponencial

Supongamos que tenemos el array: (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11), y queremos encontrar X - 10.

- Inicializar i como 1
- A[1] = 2 < 10, así que incremente i a 2.
- A[2] = 3 < 10, así que incremente i a 4.
- A[4] = 5 < 10, así que incremente i a 8.
- A[8] = 9 < 10, así que incremente i a 16.
- i = 16 > n Por lo tanto, llame a la búsqueda binaria en el rango i/2, es decir, 8 a min(i,n-1), es decir, min(16,10) =10
- Inicializa lo como i/2 = 8 y hi como min(i,n-1) = 10.
- calcular mid como 9.
- 10 == 10 es decir, A[9] == X, por lo tanto, devuelve 9.

Búsqueda de Fibonacci

La búsqueda de Fibonacci es un algoritmo de búsqueda de intervalo eficiente. Es similar a búsqueda binaria en el sentido de que también se basa en la estrategia de divide y vencerás y también necesita el array para ser ordenado. Además, la complejidad del tiempo para ambos algoritmos es logarítmica. Se llama búsqueda de Fibonacci porque utiliza la serie de Fibonacci (el número actual es la suma de dos predecesores F[i] = F[i-1] + F[i-2], F[0] = 0 y F[1]= 1 son los dos primeros números Series) y divide el array en dos partes con el tamaño dado por los números de Fibonacci. Es un método fácil de calcular que usa solo operaciones de suma y resta en comparación con la división, multiplicación y cambios de bits requeridos por la búsqueda binaria.

Ejemplo de búsqueda de Fibonacci

Supongamos que tenemos el array (1, 2, 3, 4, 5, 6, 7). Tenemos que buscar el elemento X = 6.

el array tiene 7 elementos. Entonces, n = 7. El número de Fibonacci más pequeño mayor que n es 8.

- Obtenemos fib(m) = 8, fib(m-1) = 5 y fib(m-2) = 3.
- Primera iteración
- Calculamos el índice del elemento como min(-1 + 3, 6) dándonos el elemento como A[2] = 3.
- 3 < 6 es decir, A[2] < X, por lo tanto, descartamos A[0....2] y establecemos offset como 2.
- También actualizamos la secuencia de Fibonacci para mover fib(m-2) a 2, fib(m-1) a 3 y fib(m) a 5.
- Segunda iteración
- Calculamos el índice del elemento como min(2 + 2, 6) dándonos el elemento como A[4] = 5.
- 5 < 6 es decir, A[4] < X, por lo tanto, descartamos A[2 .... 4] y establecemos offset como 4.
- También actualizamos la secuencia de Fibonacci para mover fib(m-2) a 1, fib(m-1) a 2 y fib(m) a 3.
- Tercera iteración
- Calculamos el índice del elemento como min(4 + 1, 6) dándonos el elemento como A[5] = 6.
- 6 == 6 es decir, A[5] == X, devolvemos el índice 5.

RECURSOS, MATERIALES Y EQUIPO

- Computadora
- Java
- Lectura de los materiales de apoyo del tema 6 (AULA PADLET)
- Notas de clase (problemas resueltos en clase y materiales de trabajo de la profesora)

MÉTODO BÚSQUEDA LINEAL

En que consiste la búsqueda lineal

Consiste en recorrer y examinar cada uno de los elementos del array hasta encontrar el o los elementos buscados, o hasta que se han mirado todos los elementos del array.

Este es el método de búsqueda más lento, pero si nuestra información se encuentra completamente desordenada es el único que nos podrá ayudar a encontrar el dato que buscamos.

Ejemplos de implementaciones del algoritmo en las estructuras de datos:

El siguiente algoritmo ilustra un esquema de implementación del algoritmo de búsqueda secuencial:

```
for(i=j=0;i<N;i++)
  if(array[i]==elemento)
  {
    solucion[j]=i;
    j++;
  }
```

Este algoritmo se puede optimizar cuando el array está ordenado, en cuyo caso la condición de salida cambiaría a:

```
for(i=j=0;array[i]<=elemento;i++)
```

o cuando sólo interesa conocer la primera ocurrencia del elemento en el array:

```
for(i=0;i<N;i++)
 if(array[i]==elemento)
   break;
```

En este último caso, cuando sólo interesa la primera posición, se puede utilizar un centinela, esto es, dar a la posición siguiente al último elemento de array el valor del elemento, para estar seguro de que se encuentra el elemento, y no tener que comprobar a cada paso si seguimos buscando dentro de los límites del array:

```
array[N]=elemento;
for(i=0;;i++)
 if(array[i]==elemento)
   break;
```

Si al acabar el bucle, i vale N esto indica que no se encontró el elemento. El número medio de comparaciones que hay que hacer antes de encontrar el elemento buscado es de (N+1)/2.

Análisis de eficiencia

Una métrica útil sería general, aplicable a cualquier algoritmo (de búsqueda). Dado que un algoritmo más eficiente tardaría menos en ejecutarse, un enfoque sería escribir un programa para cada uno de los algoritmos que se van a comparar, ejecutarlos y medir el tiempo que tarda cada uno en finalizar. Sin embargo, una

métrica más eficiente sería aquella que permita evaluar los algoritmos antes de implementarlos.

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

Mejor caso

El algoritmo de búsqueda lineal termina tan pronto como encuentra el elemento buscado en el array. Si tenemos suerte, puede ser que la primera posición examinada contenga el elemento que buscamos, en cuyo caso el algoritmo informará que tuvo éxito después de una sola comparación. Por tanto, la complejidad en este caso será O(1).

Peor caso

Sucede cuando encontramos X en la última posición del array. Como se requieren n ejecuciones del bucle mientras, la cantidad de tiempo es proporcional a la longitud del array n, más un cierto tiempo para realizar las instrucciones del bucle mientras y para la llamada al método. Por lo tanto, la cantidad de tiempo es de la forma an + b (instrucciones del mientras * tamaño del arreglo + llamada al método) para ciertas constantes a y b, que representan el coste del bucle mientras y el costo de llamar el método respectivamente. Representando esto en notación O, O(an+b) = O(n).

Caso promedio

Supongamos que cada elemento almacenado en el array es igualmente probable de ser buscado. La media puede calcularse tomando el tiempo total de encontrar todos los elementos y dividiéndolo por n:

Total = a (1 + 2 + ...+n) + bn = a (n(n+1) / 2) + bn, a representa el costo constante asociado a la ejecución del ciclo y b el costo constante asociado a la evaluación de la condición. 1, 2 , ..n, representan el costo de encontrar el elemento en la primera, segunda, ..., enésima posición dentro del arreglo.

Media = (Total / n) = a((n+1) / 2) + b que es O(n).

Este es el algoritmo de más simple implementación, pero no el más efectivo. En el peor de los casos se recorre el array completo y el valor no se encuentra o se recorre el array completo si el valor buscado está en la última posición del array. La ventaja es su implementación sencilla y rápida, la desventaja, su ineficiencia.

Complejidad en el tiempo y en el espacio

Complejidad en el Tiempo: O(n)

Complejidad en el Espacio: O (1)

IMPLEMENTACIÓN Y PRUEBA DEL MÉTODO BÚSQUEDA SECUENCIAL (LINEAL)

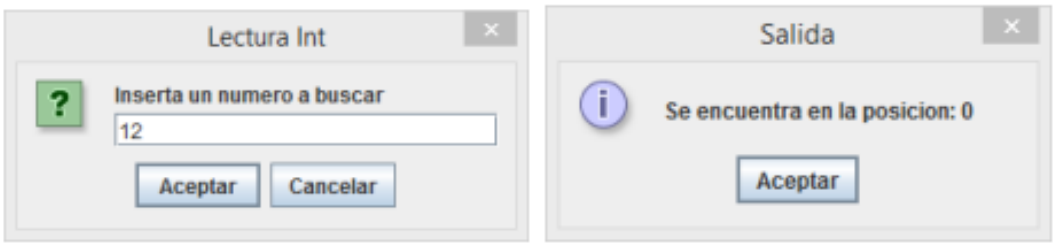
```
case "BuscarDesordenado":
    if(obj.validaVacio()) {
        ToolsPanel.imprimeErrorMsje("Array vacio");
    }
    else {
    pos = obj.busquedaSecuencial(ToolsPanel.leerInt("Inserta un numero a buscar"));
    if(pos>=0) {
        ToolsPanel.imprime("Se encuentra en la posicion: " +pos);
    }else {
        ToolsPanel.imprimeErrorMsje("Dato no encontrado");
    }
}
    break;
```

En este método lo que se busca es de que se busque algún elemento del arreglo y esto se hace de forma lineal. Que quiere decir que va a ser el recorrido posición por posición hasta encontrar el dato que queremos. Ejemplo tenemos un arreglo {1,2,3,4,5}.

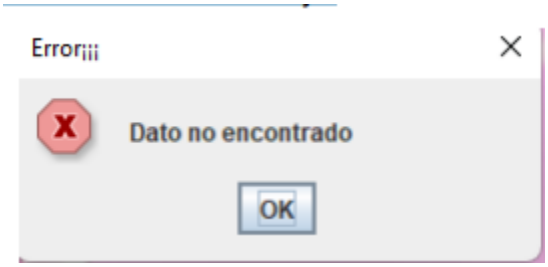
Para esto queremos encontrar el valor 4 entonces desde la posición cero va a ir haciendo el recorrido va a comparar si el valor que está en la posición cero es igual al valor que queremos buscar y nos va a decir que no, va a ir con el siguiente e igual va a comparar y nos dirá que no es el mismo, así con cada uno hasta llegar a la posición 3 y en esta igual hará la comparación que si el valor que se encuentra en la posición 3 es igual a 4, entonces automáticamente nos dirá que el dato sí existe y nos mandará a imprimir la posición en la que se encuentra, en caso contrario si ingresamos un dato no existente en el arreglo hará el mismo recorrido hasta terminar con el arreglo se dará cuenta que no existe dicho elemento, y es como nos mandará una pantalla con un mensaje que diga que el elemento no existe.

EJECUCIÓN DEL MÉTODO BÚSQUEDA SECUENCIAL (LINEAL)

Aquí agregamos el número que deseamos buscar en este caso el número 12 que si se encuentra dentro del arreglo



y si ingresamos un número que no se encuentra dentro del arreglo como el 3 nos mandara este mensaje



MÉTODO BÚSQUEDA BINARIA

En que consiste la búsqueda binaria

Consiste en un algoritmo eficiente para encontrar un elemento en una lista ordenada de elementos. Funciona al dividir repetidamente a la mitad la porción de la lista que podría contener al elemento, hasta reducir las ubicaciones posibles a solo una.

Ejemplos de implementaciones del algoritmo en las estructuras de datos:

Dado un vector A de n elementos con valores A_0 ... A_{n-1}, ordenados tal que $A_0 \leq ... \leq A_{n-1}$, y un valor buscado T, el siguiente procedimiento usa búsqueda binaria para encontrar el índice de T en A.

1. *Asignar* 0 a L y a R ($n - 1$).
2. Si L > R, la búsqueda termina sin encontrar el valor.
3. Sea m (la posición del elemento del medio) igual a la parte entera de (L + R)/2.
4. Si A_m < T, igualar L a m + 1 e ir al paso 2.
5. Si A_m > T, igualar R a m – 1 e ir al paso 2.
6. Si A_m = T, la búsqueda terminó, retornar m.

Este procedimiento iterativo mantiene los límites de la búsqueda mediante dos variables. Algunas implementaciones realizan la comparación de igualdad al final del algoritmo, como resultando se obtiene un ciclo más rápido de comparaciones, pero se aumenta en uno la cantidad de iteraciones promedio.

Ejemplo 2:

```
public class BinarySearch {


    public static int busquedaBinaria(int[] array, int elemento) {

        int izquierda = 0;

        int derecha = array.length - 1;
```

```
        while (izquierda <= derecha) {
            int medio = izquierda + (derecha - izquierda) / 2;

            if (array[medio] == elemento) {
                return medio;
            }

            if (array[medio] < elemento) {
                izquierda = medio + 1;
            } else {
                derecha = medio - 1;
            }
        }

        return -1; // Elemento no encontrado
    }

    public static void main(String[] args) {
        int[] array = {2, 5, 8, 12, 16, 23, 38, 56, 72, 91};
        int elemento = 38;

        int indice = busquedaBinaria(array, elemento);

        if (indice != -1) {
            ToolsPanel.imprimePantalla("El elemento " + elemento + " se encuentra en
el índice " + indice);
        } else {
            ToolsPanel.imprimePantalla("El elemento " + elemento + " no se encuentra
en el arreglo.");
```

```
    }
  }
}
```

Análisis de eficiencia

La búsqueda binaria funciona más rápido que la búsqueda lineal, aparte de que el número de comparaciones es menor que la búsqueda. Además de que es simple de implementar fácilmente

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

Mejor caso

En el mejor caso, la búsqueda binaria podría toparse con el elemento buscado en el primer punto medio, requiriéndose sólo una comparación de elementos.

Peor caso

En el peor de los casos se realizan iteraciones (del ciclo de comparaciones), donde la notación denota la parte entera por debajo de la función. Esta cantidad de iteraciones es alcanzada cuando la búsqueda alcanza el nivel más profundo del árbol, equivalente a una búsqueda binaria que se reduce a un solo elemento, y en cada iteración, siempre elimina el arreglo más pequeño de los dos si no tienen la misma cantidad de elementos.

Caso-promedio

Como promedio, asumiendo que cada elemento es igualmente probable de ser buscado, después que la búsqueda termine, el valor buscado será más probable de ser encontrado en el segundo nivel del árbol.

Complejidad en el tiempo y en el espacio

La complejidad en el tiempo es de O (log2n)

La complejidad en el espacio es de O (1)

IMPLEMENTACIÓN Y PRUEBA DEL MÉTODO BÚSQUEDA BINARIA METODO BUSQUEDA BINARIA

```
public int busquedaBinaria( int x)
{
    int l = 0, r = datos.length - 1;
    while (l <= r) {
        int m = l + (r - l) / 2;


        if (datos[m] == x)
            return m;


        if (datos[m] < x)
            l = m + 1;


        else
            r = m - 1;
    }


    return -1;
}
```

El arreglo debe darse en orden ordenado. Primero, el índice medio se encuentra en la matriz, el elemento buscado luego se compara con el elemento de ese índice medio. Si los dos elementos son iguales, se imprimirá la posición del elemento. Si el elemento buscado es más pequeño que el elemento del índice medio se buscará en el subarreglo izquierdo y si el elemento es mayor que el elemento del índice medio se buscará en el subarreglo derecho.

Para los subarreglos nuevamente, se seguirá el mismo procedimiento para encontrar el índice medio. El mecanismo de búsqueda binaria puede explicarse mediante una analogía de una guía telefónica. Cuando buscamos un nombre en particular, primero abrimos la página central del directorio y luego decidimos en qué parte necesitamos buscar. Este procedimiento continúa hasta que encontramos el contacto buscado.

Por ejemplo, tenemos un arreglo con los siguientes valores: {54,68,78,82,99}

La matriz está ordenada. Supongamos que el elemento de búsqueda es 82 . Ahora, por primera vez en este caso, el índice inferior es 0 , el índice superior es 5 - 1 = 4 (ya que el número de elementos es 5). Ahora el valor medio se calculará como:

medio = (índice superior + índice inferior)/2

= (4 + 0)/2

= 2

Ahora, el elemento del índice 2 es 78 , que se comparará con el elemento buscado 82 . No son iguales pero el elemento buscado es mayor que el elemento en posición media. Ahora, se buscará el subarreglo derecho. Para buscar en la submatriz derecha, el índice inferior se establecerá en mid+1. En este caso, el índice inferior será 2 + 1 = 3 , el índice superior seguirá siendo el mismo, es decir, 4 .

Así que ahora la matriz a buscar es: el 82

{~~54,68,78~~,82,99}

Ahora nuevamente el valor medio se calculará como: medio = (índice superior + índice inferior)/2

= (4 + 3)/2

= 7/2

= 3

Por lo tanto, ahora el valor del índice 3 (aquí, 82) se comparará con el elemento buscado (82), ya que ambos elementos son iguales, lo que significa que el elemento buscado está presente en la matriz, se imprimirá la posición del elemento.

Si el elemento está presente en el subconjunto izquierdo, el índice superior se establecerá en medio 1 y el índice inferior permanecerá igual. Si el elemento no está presente en la matriz, se imprimirá un mensaje de "elemento no encontrado".

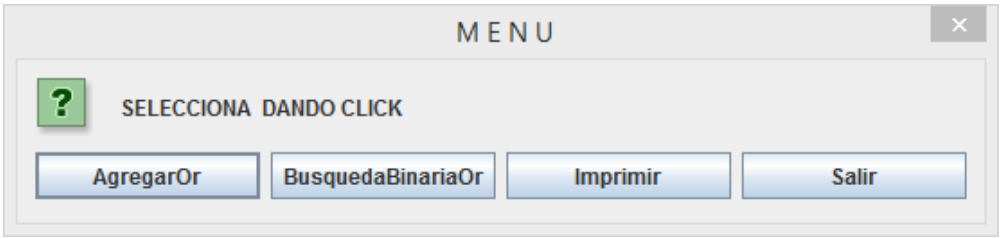
CASO EN MENU BUSQUEDA BINARIO ORDENADO

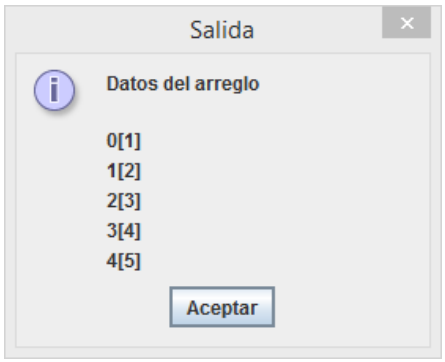
```
case "BusquedaBinariaOr":
    if(obj.validaVacio()) {
        Tools.imprimeErrorMsje("Array vacio");
    }
    else {
    pos = (byte) obj.busquedaBinaria(Tools.leerInt("Inserta un numero a buscar"));
    if(pos>=0) {
        Tools.imprime("Se encuentra en la posicion: " +pos);
    }else {
        Tools.imprimeErrorMsje("Dato no encontrado");
    }
}
    break;
```

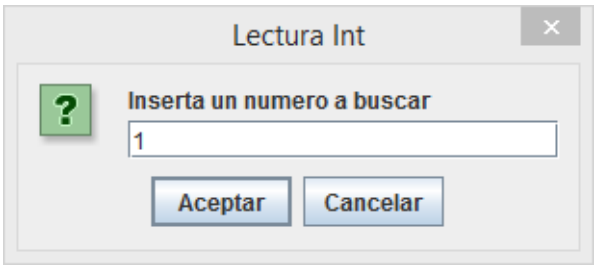
Como primera instancia, vamos a validar si el array está vacío, en caso contrario insertamos el número que vamos a buscar en el arreglo, si POS llega ser mayor a 0 nos dirá en que posición se encuentra, sino el dato no existe o no se encontró.

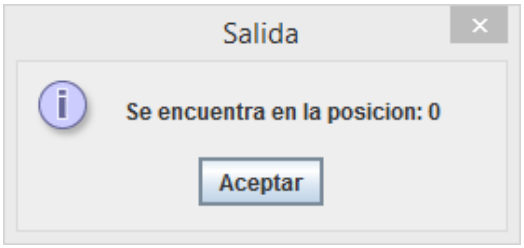
EJECUCIÓN DEL MÉTODO BÚSQUEDA BINARIA

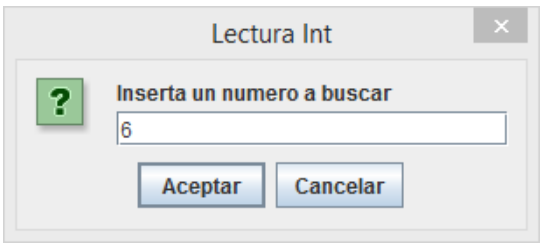
Para esto hacemos la ejecución del método para ver si funciona, y también ver si encuentra o no el dato

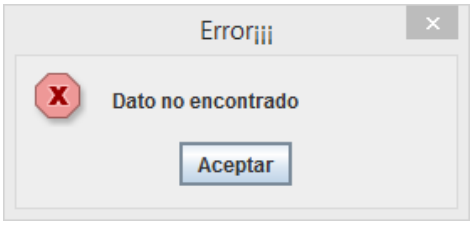












Con este resultado nos damos cuenta de que el método si es funcional y si encontró el dato y además en que posición se encuentra. Sino llegara a existir un x dato, mandara un mensaje de que el dato no se encuentra.

MÉTODO BÚSQUEDA DE PATRONES DE KNUTH MORRIS PRATT

En qué consiste la búsqueda de Knuth Morris Pratt

Consiste en buscar un patrón en O(n) tiempo, ya que nunca vuelve a comparar un símbolo de texto que coincida con un símbolo de patrón. Sin embargo, utiliza una tabla de coincidencias parciales para analizar la estructura del patrón. La construcción de una tabla de coincidencias parcial toma O(m) tiempo.

Ejemplos de implementaciones del algoritmo en las estructuras de datos:

```
public static void KMP(String text, String pattern)
   {
      // caso base 1: el patrón es nulo o está vacío
      if (pattern == null || pattern.length() == 0)
      {
         System.out.println("The pattern occurs with shift 0");
         return;
      }

      // caso base 2: el texto es NULL, o la longitud del texto es menor que la del patrón
      if (text == null || pattern.length() > text.length())
      {
         System.out.println("Pattern not found");
         return;
      }

      char[] chars = pattern.toCharArray();

      // next[i] almacena el índice de la siguiente mejor coincidencia parcial
      int[] next = new int[pattern.length() + 1];
      for (int i = 1; i < pattern.length(); i++)
      {
         int j = next[i];

         while (j > 0 && chars[j] != chars[i]) {
            j = next[j];
```

```
        }

        if (j > 0 || chars[j] == chars[i]) {
            next[i + 1] = j + 1;
        }
    }

    for (int i = 0, j = 0; i < text.length(); i++)
    {
        if (j < pattern.length() && text.charAt(i) == pattern.charAt(j))
        {
            if (++j == pattern.length()) {
                System.out.println("The pattern occurs with shift " + (i - j + 1));
            }
        }
        else if (j > 0)
        {
            j = next[j];
            i--;    // ya que `i` se incrementará en la siguiente iteración
        }
    }
}

// Programa para implementar el algoritmo KMP en Java
public static void main(String[] args)
{
    String text = "ABCABAABCABAC";
    String pattern = "CAB";
```

```
    KMP(text, pattern);
  }
```

Análisis de eficiencia

Debido a que el algoritmo precisa de 2 partes donde se analiza una cadena en cada parte, la complejidad promedio resultante es O(k) y O(n), cuya suma resulta ser O(n + k).

Genéricamente puede despreciarse el tiempo de cálculo de la tabla de fallos del patrón, salvo que su tamaño sea excesivamente grande. También en los casos en que el mismo patrón se buscará en muchos diferentes textos y que es la razón por la que conviene precalcular la tabla de fallos del patrón de forma independiente de la función de búsqueda.

La capacidad del algoritmo (para demostrar que no examina caracteres más de dos veces) queda patente al apreciar como logra saltar varios caracteres a la vez cuando hay un fallo. Esto se aprecia mejor, mirando la tabla 'F', cuantos mayor es el valor en la tabla tanto más grande es el salto resultante (salto debido a que dichos caracteres ya han sido examinados) y cuantos más ceros haya, señala que ha ido avanzando el puntero absoluto de uno en uno.

En la práctica el algoritmo no será mucho mejor que el algoritmo lineal de fuerza bruta, en condicioes normales (texto del lenguaje hablado, por ejemplo), pero mejora sustancialmente, respecto del mismo cuando sale de dichas condiciones normales, en tanto que algoritmo presente no presenta demasiada sobrecarga.

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

Respecto del patrón, no existe caso mejor ni peor (considerado por sí solo), puede demostrarse que en una tabla de fallos donde son todo ceros (o lo que es lo mismo, el primer carácter del patrón aparece una única vez (por ejemplo P ="ABBBBBBB")), tiene el mismo coste que cuando el patrón se compone de 2 únicos caracteres donde 1 se repite en todo el patrón y el otro aparece al final, (por ejemplo: P= "AAAAAAAB", (obsérvese la tabla F para dicho texto)) y tiene el mismo coste que en cualquier otra situación intermedia entre dichos extremos:

i	0	1	2	3	4	5	6	7

P[i]	A	A	A	A	A	A	A	B
F[i]	-1	0	1	2	3	4	5	6

En el caso de que el primer carácter aparezca una única vez (ver tabla aquí debajo), examina la A, si coincide, examina la B, en caso de fallo simplemente salta 1 carácter. Si el patrón existiere finalmente en el texto, será cuando haya oportunidad de examinar el resto de los caracteres, antes de alcanzar el final del texto.

i	0	1	2	3	4	5	6	7
P[i]	A	B	B	B	B	B	B	B
F[i]	-1	0	0	0	0	0	0	0

Igualmente cuando el primer carácter se repite excepto en el último carácter. Recorrería, hasta el último que genera el fallo, pero igualmente el puntero absoluto del texto solo avanza 1, con cada fallo.

La diferencia entre uno y otro casos consiste básicamente en donde se localiza el puntero en el patrón y en qué momento se recorre el resto del patrón si al principio o al final del texto.

Para el texto, el mejor caso se da cuando el patrón se localiza en la posición 0. El peor caso se da cuando el patrón se localiza en la última posición. Si el texto no se encuentra, es igual o ligeramente mejor que el peor caso, y es en tal caso dependiente del patrón (ver sección anterior y siguiente).

Considerando conjuntamente el patrón y el texto, el peor caso se puede dar en función de donde y cuantas veces surja la ruptura del patrón. Cada carácter fallido necesita ser comprobado otra vez con aquel con el que se alinea. Esto implica que el peor caso puede llegar a tener un costo de O(n) + O(k*2), si bien en la práctica estos casos son raros, pues no es frecuente que deban realizarse búsqueda con patrones o textos cuyos caracteres tengan una alta frecuencia de reptición (como los que s emuestran de ejemplo a continuación)..

Ejemplos: Patrón (tabla F):Texto = nº de comparaciones precisas (en todos los ejemplos el tamaño del patrón y del texto es igual: 6:30)

AAAAAZ (-1 0 1 2 3 4):AAAAACAAAAACAAAAACAAAAACAAAAAZ = 51

AAAAAZ (-1 0 1 2 3 4):AAAAAAAAAAAAAAAAAAAAAAAAAAAZ = 54

ABBBBB (-1 0 0 0 0 0):ABBBBZABBBBZABBBBZABBBBZABBBBB = 35 (coste típico m+n).

Complejidad en el tiempo y complejidad en el espacio

La complejidad temporal O(N+M) donde N es la longitud del texto y M es la longitud del patrón a encontrar.

La complejidad en el espacio: (M)

IMPLEMENTACION Y PRUEBA DEL ALGORITMO DEL METODO DE BUSQUEDA KMP

METODO MAIN PARA AGREGAR TEXTO Y PATRON

```
public static void main(String[] args) {
String texto =Tools.leerString("Ingresa un texto");
String patron =Tools.leerString("Ingresa un patron");
List<Integer> resultados = buscarPatron(texto, patron);
if (!resultados.isEmpty()) {
Tools.imprime("El patrón se encontró en las siguientes posiciones:" +resultados);
} else {
Tools.imprimeErrorMsje("El patrón no se encontró en el texto.");
}
}
```

En esta parte del main pedimos cualquier texto, y por consiguiente un patron (letra que se encuentra en dicho texto), para esto creamos un objeto llamado resultados el cual tomara como parámetro los valores que metimos en la variable texto y la variable patron. Si no llegara a estar vacio el objeto resultado, entonces imprimirá las posiciones en las que se encuentra el patron.

Por otro lado si no llegara a ser verdad, entonces nos dira que el patron no se encontró en el texto.

METODO CALCULAR KMP

```
public static int[] calcularLPS(String patron) {
    int m = patron.length();
    int[] lps = new int[m];
    int longitud = 0;
    int i = 1;
    while (i < m) {
    if (patron.charAt(i) == patron.charAt(longitud)) {
    longitud++;
    lps[i] = longitud;
    i++;
    } else {
    if (longitud != 0) {
    longitud = lps[longitud - 1];
    } else {
    lps[i] = 0;
    i++;
    }
    }
    }
    return lps;
    }
```

En este método calcularemos la longitud del patrón de tal manera que por medio de este sepamos la posición en la que se encuentra el patron, y esto se hace sacando cada letra del texto para identificar el patron. Y por ultimo va a retornar la posición en la que se encuentra el patron del texto.

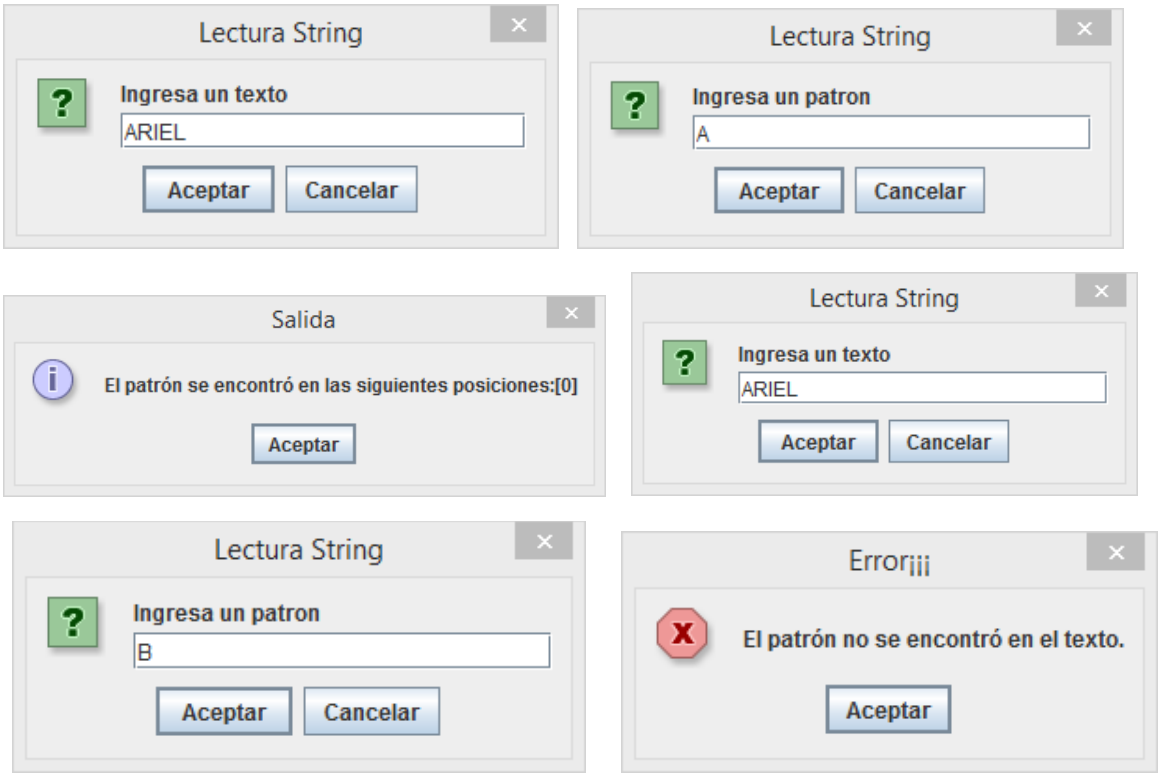
METODO BUSCAR PATRON

```
public static List<Integer> buscarPatron(String texto, String patron) {
int n = texto.length();
int m = patron.length();
int[] lps = calcularLPS(patron);
List<Integer> resultados = new ArrayList<>();
int i = 0;
int j = 0;
while (i < n) {
if (texto.charAt(i) == patron.charAt(j)) {
i++;
j++;
if (j == m) {
resultados.add(i - j);
j = lps[j - 1];
}
} else {
if (j != 0) {
j = lps[j - 1];
} else {
i++;
}
}
}
return resultados;
}
```

Ahora para este método, tomaremos las variables m y n en este caso la longitud de estos mismos, en caso de n que es largo del texto en donde se buscar el patrón, y y m denotara el largo del patrón a buscar. E igual a la variable lps le asignaremos calcularPatron (lo llamamos indirectamente), en este le ponemos como parámetro lo que hay en la variable patron.Y después vamos a ir devolviendo la posición del carácter que le pasemos como parámetro en el interior de la cadena. Si la cadena contiene varias ocurrencias del carácter que le pasemos como parámetro, el método charAt devolverá solamente la posición de la primera de ellas. Y entonces si el patrón llegara a ser parte del texto, nos devolverá el resultado, el cual será llamado en el método main.

EJECUTABLE METODO BUSQUEDA KMP

Para esto hacemos la ejecución del método para ver si funciona, y también ver si encuentra o no el patrón en el texto



Con este resultado nos damos cuenta de que el método si es funcional y si encontró el patrón en el texto y además en que posición se encuentra. Sino llegara a existir un x patrón, mandara un mensaje de que el patrón no se encuentra.

MÉTODO SALTAR BÚSQUEDA

En que consiste la búsqueda Saltar búsqueda

Consiste en verificar menos elementos saltando hacia adelante con pasos fijos algunos elementos en lugar de buscar elemento por elemento.

Ejemplos de implementaciones del algoritmo en las estructuras de datos:

```java
public class JumpSearchAlgorithm {

  public static void main(String[] args) {
    int[] arr = {10, 20, 30, 40, 50, 60, 70, 80, 90};
    int ele = 60;

    int foundIndex = jumpSearch(arr, ele);
    System.out.println(foundIndex > 0 ? "Found at index : " + foundIndex : "Element Not Fou
  }

  public static int jumpSearch(int[] arr, int ele) {

    int prev = 0;
    int n = arr.length;
    int step = (int) Math.floor(Math.sqrt(n));

    //loop until current element is less than the given search element
    while (arr[Math.min(step, n) - 1] < ele) {
      prev = step;
      step += (int) Math.floor(Math.sqrt(n));
      if (prev >= n) return -1;
    }

    //perform linear search prev index element to given element
    while (arr[prev] < ele) {
      prev++;
      if (prev == Math.min(step, n)) return -1;
    }

    // Return index if element is found
    if (arr[prev] == ele) return prev;

    return -1;
  }
}
```

Ejemplo 2:

```java
import java.util.Scanner;
//defining a class
class JumpSearch{
    //data members(variables)
    int a[]=new int[10],n,x,strt,end,f=0;
    //member methods

//function to take inputs
void accept()
{
    int i;
    Scanner sc=new Scanner(System.in);
    //taking the inputs
    System.out.println("\nEnter the number of elements: ");
    n=sc.nextInt();
    System.out.println("\nEnter the elements of the array: ");
    for(i=0;i<n;i++)
        a[i]=sc.nextInt();
    //taking the element to be searched
    System.out.println("\nEnter the element to search: ");
    x=sc.nextInt();
}
//function to search
void search()
{
    int i;
    //searching the element in the array
    strt = 0;
    end = (int) Math.sqrt(n);
    while(a[end] <= x && end < n) {
      strt = end;
      end += Math.sqrt(n
      if(end > n - 1)
         end = n;
   }
   for(i = strt; i<end; i++) {
      if(a[i] == x)
      {
          System.out.println("The element "+x+" is found at index "+i);
        f=1;
      }
   }
   if(f==0)
       System.out.println("element not found");

}

//main function
public static void main(String[] args)
{
    //creating an object
    JumpSearch j=new JumpSearch();
    //calling the objects to perform the jump search
    j.accept();
    j.search();
}
}
```

Análisis de eficiencia

En el mejor de los casos, Jump Search encontraría el elemento de destino en el borde del primer bloque en el que busca. A su vez, esto hace que Jump Search tenga una eficiencia en el mejor de los casos de complejidad O(1) en términos de notación Big-O .

En consecuencia, Jump Search mantiene una eficiencia de caso peor y promedio de complejidad O(√n) .

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

Caso promedio

El algoritmo de clasificación de salto se ejecuta n / m veces donde n es el número de elementos y m es el tamaño del bloque. La búsqueda lineal requiere comparaciones m-1 haciendo que la expresión de tiempo total sea n / m + m-1. El valor más óptimo de m minimizando la expresión de tiempo es √n, haciendo que la complejidad de tiempo sea n/√n + √n, es decir, √n. La complejidad de tiempo del algoritmo Jump Search es O(√n).

Mejor caso

La complejidad del tiempo en el mejor de los casos es O(1). Ocurre cuando el elemento a buscar es el primer elemento presente dentro del array.

Peor caso

El peor de los casos ocurre cuando hacemos saltos n/m, y el último valor que comprobamos es mayor que el elemento que estamos buscando, y se realizan comparaciones m-1 para búsqueda lineal. La complejidad de tiempo en el peor de los casos es O(√n).

Complejidad en el tiempo y complejidad espacial

La complejidad en el tiempo de este algoritmo es O(√n) dónde n es el tamaño de la matriz.

La complejidad espacial de este algoritmo es O(1) porque no requiere ninguna estructura de datos más que variables temporales.

IMPLEMENTACION Y PRUEBA DEL ALGORITMO DEL METODO DE SALTO DE BUSQUEDA

METODO AGREGAR ELEMENTOS AL ARREGLO

```
int a[]=new int[10],n,x,strt,end,f=0;

public void agregar()
{
int i;
String cad="";
n=Tools.leerInt("\nIngresa el numero de elementos para el array: ");

for(i=0;i<n;i++) {
a[i]=Tools.leerInt("\nIngresa los elementos del arreglo: ");
    cad+= i + "[" + a [ i ]+ "]" + "\n" ;
}
Tools.imprime("Datos del arreglo\n "+cad);

}
```

Aquí en esta parte vamos a agregar elementos al arreglo, primero preguntamos cuantos valores vamos a insertar, y después de ello ingresamos los elementos que queramos para que posteriormente se impriman en pantalla.

METODO BUSQUEDA SALTO

```
public void busquedaSalto()
{
x=Tools.leerInt("\nIngresa el elemento a buscar: ");
int i;

strt = 0;
end = (int) Math.sqrt(n);
while(a[end] <= x && end < n) {
strt = end;
end += Math.sqrt(n);
if(end > n - 1)
end = n;
}
for(i = strt; i<end; i++) {
if(a[i] == x)
{
Tools.imprime("El elemento "+x+" si se encuentra en la posicion "+i);
f=1;
}

}
if(f==0)
    Tools.imprimeErrorMsje("Elemento no encontrado");

}
```

En este método lo que se quiere buscar es un elemento en el arreglo, para esto debemos de haber ingresado elementos en un arreglo, para posteriormente buscar x elemento, Por consiguiente, esto comienza desde pedir el elemento que queremos

buscar, Ya en el método, recorremos el arreglo ordenado y avanzamos con el tamaño de bloque específico. Determinaremos el tamaño del bloque para saltar obteniendo la raíz cuadrada del tamaño del arreglo. es decir , √n donde n es la longitud del arreglo ordenado.

Cuando encontramos un elemento mayor que el elemento de búsqueda, se realiza una búsqueda lineal en el bloque actual. El elemento, si está presente en el arreglo, se encontrará en el bloque. Sino entonces mandara un mensaje de que el elemento no se encontró.

Por ejemplo, tenemos un arreglo con los siguientes valores: {1,2,3,4,5,6}

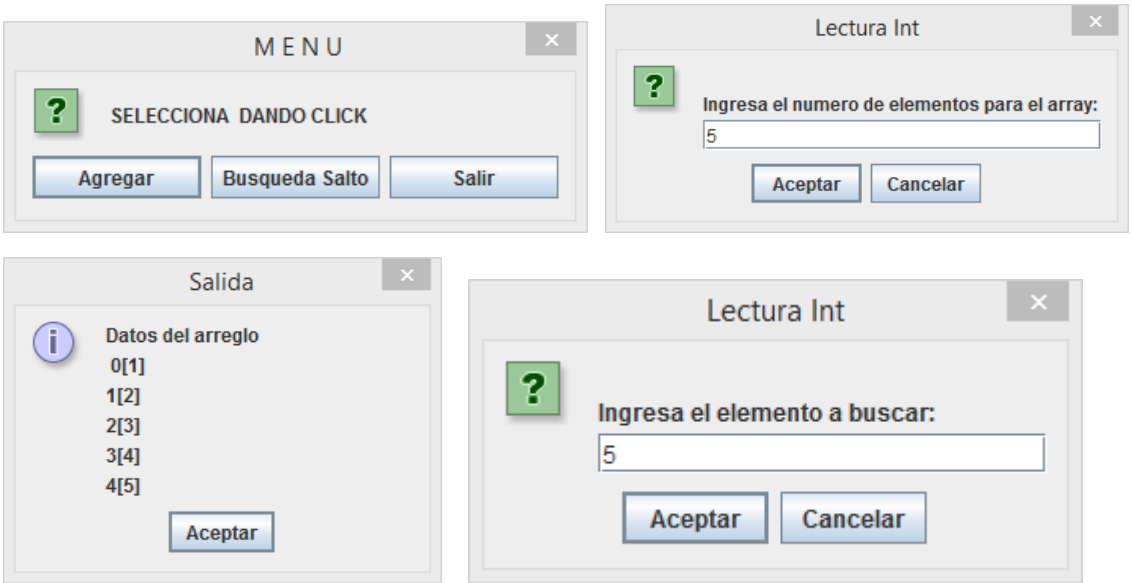
La longitud de la matriz es 6 .Entonces, el tamaño del bloque = 6 = 2

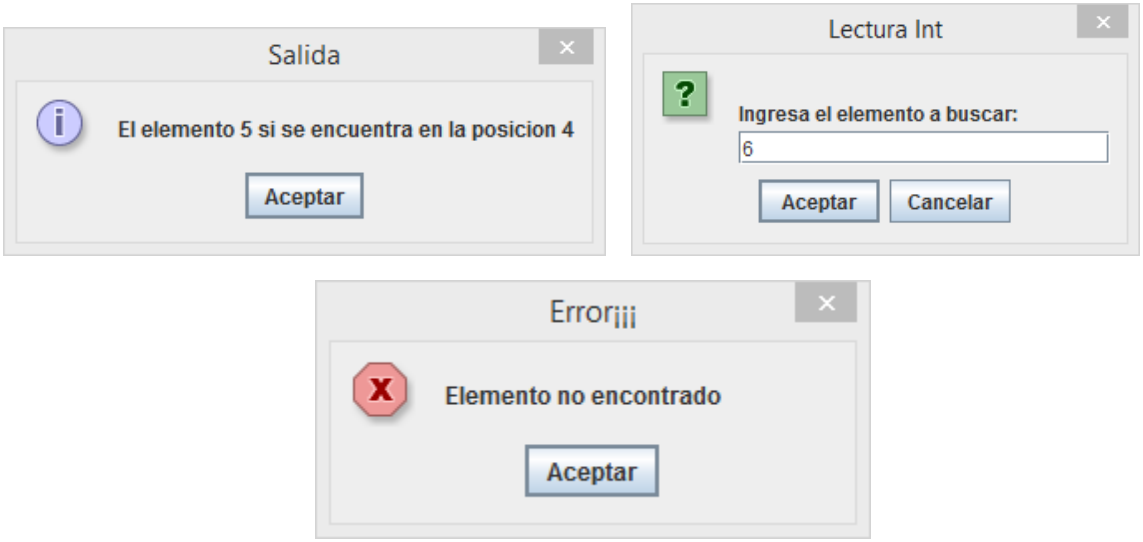
Deje que el elemento a buscar sea 4 . Ahora saltará del índice 0 al índice 0 + 2 = 2. el elemento buscado es el 4 que se compara con el 3 (en la posición 2). Como el 4 es mayor a la posición 2, es decir, el 3, el índice volverá a saltar de 2 a 2 que da resultado 4. Después, el elemento buscado es el 4, que es menor que el valor en la posición 4, es decir, 5, luego nuevamente, el índice volverá a saltar a 2. Ahora la búsqueda lineal se realizará a partir del índice 2 +1 = 3. El elemento se comparará con 4.

Como el elemento coincidió, imprimirá la posición del elemento en el arreglo. Para esto hacemos la prueba del método, para ver su funcionalidad:

EJECUTABLE METODO BUSQUEDA SALTO

Para esto hacemos la ejecución del método para ver si funciona, y también ver si encuentra o no el dato





Con este resultado nos damos cuenta de que el método si es funcional y si encontró el dato y además en que posición se encuentra. Sino llegara a existir un x dato, mandara un mensaje de que el dato no se encuentra.

MÉTODO BÚSQUEDA DE INTERPOLACIÓN

En que consiste la búsqueda de interpolación.

Consiste en calcular en qué espacio de búsqueda restante podría estar presente el, en función de los valores alto y bajo del espacio de búsqueda y el valor del objetivo. El valor encontrado en esta posición estimada se compara luego con el valor objetivo. Si no es igual, el espacio de búsqueda restante se reduce a la parte anterior o posterior a la posición estimada según la comparación. Este método solo funcionará si los cálculos sobre el tamaño de las diferencias entre los valores de objetivo son sensatos.

Ejemplos de implementaciones del algoritmo en las estructuras de datos:

```
public static int interpolationSearch(int arr[], int lo,Int hi, int x)
  {
     Int pos;


     // Since array is sorted, an element
     // present in array must be in range
     // defined by corner
     If (lo <= hi && x >= arr[lo] && x <= arr[hi]) {
```

```
        // Probing the position with keeping
        // uniform distribution in mind.
        pos = lo
             + (((hi – lo) / (arr[hi] – arr[lo]))
                • (x – arr[lo]));

        // Condition of target found
        If (arr[pos] == x)
            return pos;

        // If x is larger, x is in right sub array
        If (arr[pos] < x)
            return interpolationSearch(arr, pos + 1, hi,x);

        // If x is smaller, x is in left sub array
        If (arr[pos] > x)
            return interpolationSearch(arr, lo, pos – 1,  x);
    }
    return -1;
}

// Driver Code
public static void main(String[] args)
{

    // Array of items on which search will
    // be conducted.
    Int arr[] = { 10, 12, 13, 16, 18, 19, 20, 21,
```

```
                22, 23, 24, 33, 35, 42, 47 };

        Int n = arr.length;

        // Element to be searched
        Int x = 18;
        Int index = interpolationSearch(arr, 0, n – 1, x);

        // If element was found
        If (index ¡= -1)
            System.out.println(“Element found at index “
                        + index);
        else
            System.out.println(“Element not found.”);
    }
}
```

Ejemplo 2:

```
public int interpolationSearch(int[] data, int item) {

    Int highEnd = (data.length – 1);
    Int lowEnd = 0;

    while (item >= data[lowEnd] && item <= data[highEnd] && lowEnd <= highEnd) {

        Int probe
          = lowEnd + (highEnd – lowEnd) * (item – data[lowEnd]) / (data[highEnd] –
data[lowEnd]);
```

```
        If (highEnd == lowEnd) {
            If (data[lowEnd] == item) {
                return lowEnd;
            } else {
                return -1;
            }
        }

        If (data[probe] == item) {
            return probe;
        }

        If (data[probe] < item) {
            lowEnd = probe + 1;
        } else {
            highEnd = probe – 1;
        }
    }
    return -1;
}
```

Análisis de eficiencia

Para que la búsqueda por interpolación funcione de manera eficiente, la matriz debe ordenarse y distribuirse uniformemente; de lo contrario, la complejidad del tiempo puede llegar a O (n), que es peor que la búsqueda binaria.

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

Caso promedio

La complejidad del tiempo de caso promedio del algoritmo es del orden de O(log (logn)). Ocurre cuando todos los elementos dentro del array se distribuyen uniformemente.

Mejor caso

El mejor de los casos ocurre cuando el elemento que estamos buscando es el primer elemento sondeado por búsqueda de interpolación. La complejidad de tiempo del algoritmo en el mejor de los casos es O(1).

Peor caso

El peor de los casos ocurre cuando los valores numéricos de los objetivos aumentan exponencialmente. La complejidad temporal del algoritmo en el peor de los casos es O(n).

Complejidad en el tiempo y complejidad espacial.

Complejidad de tiempo: O(log2(log2 n)) para el caso promedio y O(n) para el peor de los casos.

Complejidad del espacio auxiliar: O(1)

IMPLEMENTACION Y PRUEBA DEL ALGORITMO DEL METODO DE BUSQUEDA DE INTERPOLACION

METODO AGREGAR ELEMENTOS AL ARREGLO

```
int a [] = new int [ 10 ], n , x , l , h , pos ;
public void agregar()
{
int i;
String cad="";
n=Tools.leerInt("\nIngresa el numero de elementos para el array: ");

for(i=0;i<n;i++) {
a[i]=Tools.leerInt("\nIngresa los elementos del arreglo: ");
    cad+= i + "[" + a [ i ]+ "]" + "\n" ;
}
Tools.imprime("Datos del arreglo\n "+cad);

}
```

Aquí en esta parte vamos a agregar elementos al arreglo, primero preguntamos cuantos valores vamos a insertar, y después de ello ingresamos los elementos que queramos para que posteriormente se impriman en pantalla.

METODO BUSQUEDA DE INTERPOLACION

```
public void busquedaInterpolacion() {
    x=Tools.leerInt("\nIngresa el elemento a buscar: ");
    //buscando el elemento en el arreglo
    l = 0 ; h = n - 1 ;
    while ( l <= h && x >= a [ l ] && x <= a [ h ])
    {
        if (l == h ){
            if ( a [ l ] == x )
                Tools.imprime( "El elemento " + x + " se encuentra en el índice " + l );
            else
                Tools.imprimeErrorMsje("Elemento no encontrado");
            return;
        }
    pos = (int) ( l + ((( double )( h - l ) / ( a [ h ] - a [ l ])) * ( x - a [l])));
    if ( a [pos] == x )
    {
        Tools.imprime ( "El elemento " + x + " se encuentra en el índice " +pos );
        return;
    }
    if ( a [ pos ] < x )
        l = pos + 1 ;
    else
        h = pos - 1 ;
    }
    Tools.imprimeErrorMsje ( "elemento no encontrado" );
}
```


En este método lo que se quiere buscar es un elemento en el arreglo, para esto debemos de haber ingresado elementos en un arreglo, para posteriormente buscar x elemento, Por consiguiente, esto comienza desde pedir el elemento que queremos buscar, después función de la posición de sondeo del valor requerido. Pero antes la lista debe ordenarse para realizar la búsqueda de interpolación. En cada paso de esta técnica de búsqueda, se calcula el espacio de búsqueda restante para el valor a encontrar. El cálculo se realiza en función de los valores en los límites del espacio de búsqueda y el valor a buscar. Para encontrar la posición del elemento buscado se utiliza la siguiente fórmula:

Posición = índice inferior + [(matriz de elementos buscados [índice inferior]) * (índice superior – índice inferior) /(matriz [índice superior] – matriz [índice inferior])

Que esta misma se implementó en código en Eclipse. Básicamente lo que hace en el código es hacer una operación matemática para llegar a la posición del elemento que vamos a buscar, y veremos si el dato se encontró o no. Para resumir el código, nos dice que si el elemento coincide con el entonces se imprimirá la posición.

Si el elemento es menor que array[position] , se calcula nuevamente la posición de la sonda para el subconjunto izquierdo y, si es mayor, la posición se calcula para el subconjunto derecho.

Estos pasos se repetirán hasta que obtengamos una coincidencia o las sub-matrices se reduzcan a 0

Por ejemplo, tenemos un arreglo con los siguientes valores: {14,52,63,74,85,90}

El elemento que queremos buscar es el 14. Primero el índice inferior = 0, Índice superior = 5

Posición= 0 + [(14 - 14) *(5 - 0)/ (90 - 14)]

= 0 + [0*5/76]

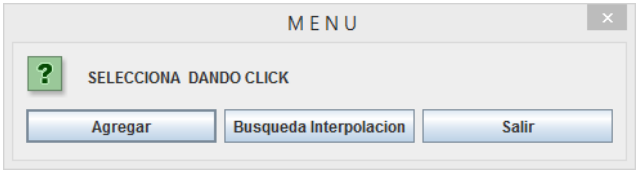
= 0 + [0/76]

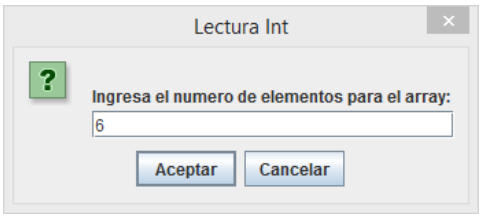
= 0

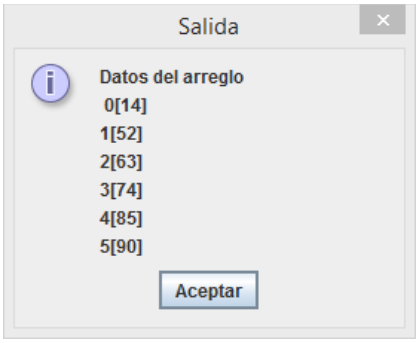
Como el elemento coincidió, imprimirá la posición del elemento en el arreglo. Para esto hacemos la prueba del método, para ver su funcionalidad:

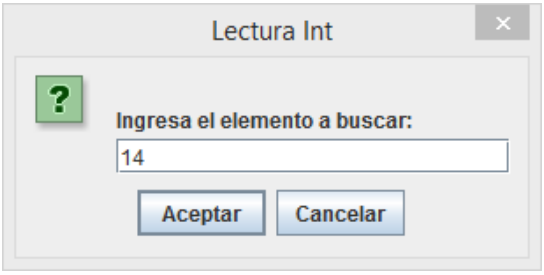
EJECUTABLE METODO BUSQUEDA DE INTERPOLACION

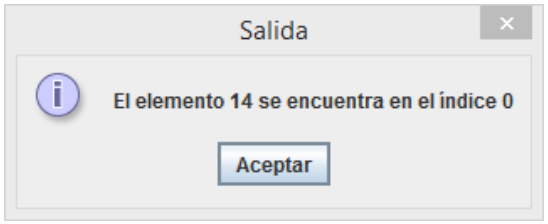
Para esto hacemos la ejecución del método para ver si funciona, y también ver si encuentra o no el dato

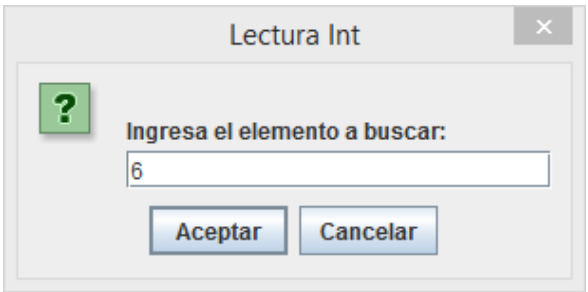


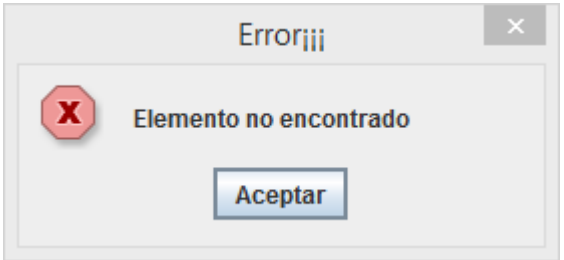












Con este resultado nos damos cuenta de que el método si es funcional y si encontró el dato y además en que posición se encuentra. Sino llegara a existir un x dato, mandara un mensaje de que el dato no se encuentra.

MÉTODO BÚSQUEDA EXPONENCIAL

En que consiste la búsqueda exponencial

Consiste en un proceso de dos pasos. Primero, el algoritmo intenta encontrar el rango (L, R) en el que está presente el elemento objetivo y luego utiliza la búsqueda binaria dentro de este rango para encontrar la ubicación exacta del objetivo.

Ejemplos de implementaciones del algoritmo en las estructuras de datos:

Supongamos que tenemos el array: (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11), y queremos encontrar X - 10.

Ejemplo 1:

1. Inicializar i como 1
2. A[1] = 2 < 10, así que incremente i a 2.
3. A[2] = 3 < 10, así que incremente i a 4.
4. A[4] = 5 < 10, así que incremente i a 8.
5. A[8] = 9 < 10, así que incremente i a 16.
6. i = 16 > n Por lo tanto, llame a la búsqueda binaria en el rango i/2, es decir, 8 a min(i,n-1), es decir, min(16,10) =10
7. Inicializa lo como i/2 = 8 y hi como min(i,n-1) = 10.
8. calcular mid como 9.
9. 10 == 10 es decir, A[9] == X, por lo tanto, devuelve 9.

Ejemplo 2:

```
// Java program to

// find an element x in a

// sorted array using

// Exponential search.

import java.util.Arrays;


class GFG

{

        // Returns position of
```

```
// first occurrence of
// x in array
static int exponentialSearch(int arr[],
                                                    int n, int x)
{
        // If x is present at first location itself
        if (arr[0] == x)
                return 0;

        // Find range for binary search by
        // repeated doubling
        int i = 1;
        while (i < n && arr[i] <= x)
                i = i*2;

        // Call binary search for the found range.
        return Arrays.binarySearch(arr, i/2,
                                            Math.min(i, n-1), x);
}

// Driver code
public static void main(String args[])
{
        int arr[] = {2, 3, 4, 10, 40};
        int x = 10;
        int result = exponentialSearch(arr,
                                                    arr.length, x);
```

```
            System.out.println((result < 0) ?
            "Element is not present in array" :
            "Element is present at index " +
                                                result);
        }
}
```

Análisis de eficiencia

Esta técnica de búsqueda es más eficiente que la búsqueda binaria si el elemento está presente más cerca del primer elemento. En algunos casos, la búsqueda exponencial es más rápida que la búsqueda binaria.

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

Caso promedio

La complejidad del tiempo de caso promedio es O(logi) donde i es el índice del elemento objetivo X dentro del array. Incluso supera a la búsqueda binaria cuando el elemento está cerca del comienzo del array.

Mejor caso

El mejor de los casos ocurre cuando el elemento que comparamos es el elemento que estamos buscando y se devuelve en la primera iteración. La complejidad del tiempo en el mejor de los casos es O(1).

Peor caso

La complejidad del tiempo del caso más desfavorable es la misma que la complejidad del tiempo del caso medio. La complejidad de tiempo en el peor de los casos es O(logi).Complejidad en el tiempo y complejidad espacial

Complejidad en el tiempo y complejidad en el espacio

La complejidad del tiempo es O(Log n)

La complejidad del espacio de este algoritmo es O(1) porque no requiere más espacio que las variables temporales.

IMPLEMENTACION Y PRUEBA DEL ALGORITMO DEL METODO DE BUSQUEDA DE EXPONENCIAL

METODO AGREGAR ELEMENTOS AL ARREGLO

```
int a [] = new int [ 10 ], n , x , l , h ,mid,bound,indx ;
public void agregar()
{
int i;
String cad="";
n=Tools.leerInt("\nIngresa el numero de elementos para el array: ");

for(i=0;i<n;i++) {
a[i]=Tools.leerInt("\nIngresa los elementos del arreglo: ");
    cad+= i + "[" + a [ i ]+ "]" + "\n" ;
}
Tools.imprime("Datos del arreglo\n "+cad);

}
```

Aquí en esta parte vamos a agregar elementos al arreglo, primero preguntamos cuantos valores vamos a insertar, y después de ello ingresamos los elementos que queramos para que posteriormente se impriman en pantalla.

METODO EXTRA BUSQUEDA BINARIA

```
int busq_binaria(int a[],int l,int h,int x){
if(l>h)
return -1;

mid=(l+h)/2;

if (x==a[mid])
return mid;

else if(x<a[mid])
return busq_binaria(a,l,mid-1,x);
else
return busq_binaria(a,mid+1,h,x);
}
```

En este método lo que haremos es tomar como parámetro las variables que declaramos al inicio, ya que las vamos a ocupar para realizar las operaciones de buscar el índice del elemento a buscar, para esto este método será de forma recursiva, porque así recorrerá cada posición hasta llegar al elemento buscado, además de que se hará de esa forma la operación. Por medio de este método, va a ser llamado desde la método de búsqueda exponencial. En pocas palabras necesitara de la búsqueda binaria para realizar la búsqueda exponencial.

METODO BUSQUEDA DE EXPONENCIAL

```
public void busquedaExponencial()
{
x=Tools.leerInt("\nIngresa el elemento a buscar: ");
bound=1;

while(bound<n&&a[bound]<x)
bound*=2;

indx=busq_binaria(a,bound/2,Math.min(bound,n),x);
if(indx!=-1)
    Tools.imprime("El elemento "+x+" se encuentra en el índice "+indx);
else
    Tools.imprimeErrorMsje("Elemento no encontrado");
}
```

En este método lo que se quiere buscar es un elemento en el arreglo, para esto debemos de haber ingresado elementos en un arreglo, para posteriormente buscar x elemento, una mejor manera de explicar este método es el siguiente:

primero se encuentra un rango entre el cual existe el valor buscado. Luego, en ese rango en particular, se aplica la técnica de búsqueda binaria. Para la búsqueda exponencial, la lista debe estar ordenada. Primero, el tamaño del subarreglo se toma como 1, luego el último elemento de ese rango en particular se compara con el valor buscado, luego el tamaño del subarreglo se toma como 2, luego 4, y así sucesivamente.

Cuando el último elemento del subarreglo es mayor que el elemento buscado, el proceso se detiene. Ahora, podemos decir que el elemento buscado está presente entre i/2 e i (si el último índice del subarreglo mayor que el elemento buscado es i). El rango es i/2 e i porque en la iteración anterior el último elemento no era mayor que el elemento buscado.

Que esta misma se implementó en código en Eclipse. Básicamente lo que hace en el código es hacer uso del método de búsqueda de forma recursiva, y en el método exponencial hace una llamada indirecta (método de búsqueda binaria) para llegar a la posición del elemento que vamos a buscar, y veremos si el dato se encontró o no. Para resumir el código, nos dice que si el elemento coincide con el entonces se imprimirá la posición.

Por ejemplo, tenemos un arreglo con los siguientes valores: {15,20,35,47,59}

El elemento que queremos buscar es el 47. Primero, el rango se establece en 1 . Ahora el valor del índice 1 (es decir, 20) se comparará con el elemento buscado (47)

Por consiguiente, como el elemento buscado no es menor que el valor del índice 1 , el siguiente rango se establecerá en 2 * 1 = 2 . Ahora nuevamente el valor del índice 2 (es decir, 35) se comparará con el elemento buscado (47) .

Después como 35 es menor que 47 , el límite se establecerá en 2 * 2 = 4 . El elemento buscado 47 ahora se comparará con el valor en el índice 4, es decir, 59 .

Ahora el elemento no es menor que 47 . Entonces, el incremento del límite se detendrá. Ahora, la búsqueda binaria se realizará entre los índices 2 y 4 .

Ahora bien como el índice inferior es 2 y el índice superior es 4 .Entonces, el índice medio se calculará como: medio = (índice inferior + índice superior)/2

= (2 + 4)/2

= 6/2

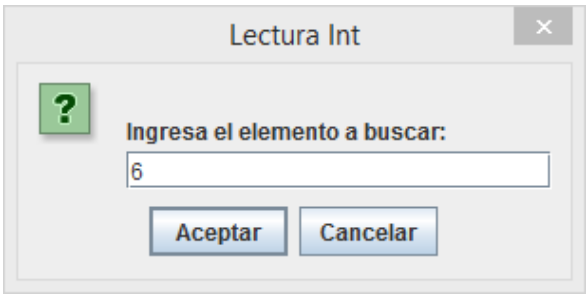
= 3

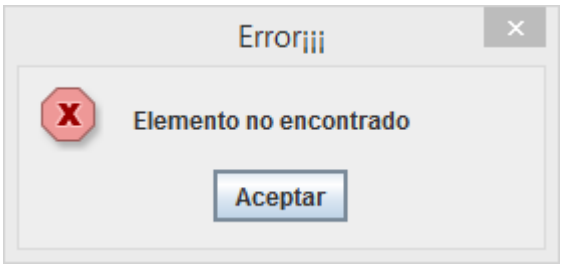
Ahora el valor buscado (47) se comparará con el valor del índice 3(47) .

A medida que sus valores coincidan, se imprimirá la posición del elemento .

EJECUTABLE METODO BUSQUEDA DE EXPONENCIAL

Para esto hacemos la ejecución del método para ver si funciona, y también ver si encuentra o no el dato





Con este resultado nos damos cuenta de que el método si es funcional y si encontró el dato y además en que posición se encuentra. Sino llegara a existir un x dato, mandara un mensaje de que el dato no se encuentra.

MÉTODO BÚSQUEDA DE FIBONACCI

En que consiste la búsqueda Fibonacci

Consiste que para esta técnica de búsqueda , tenemos que encontrar el número de Fibonacci más pequeño que sea mayor o igual que n (el número de elementos de la lista). Sean los dos números de Fibonacci anteriores F(m - 1) y F(m - 2)

Ejemplos de implementaciones del algoritmo en las estructuras de datos:

Ejemplo 1:

```
import java.io.*;
import java.util.Scanner;
public class FibonacciSearch {
  static int min(int a, int b) {
    return (a > b) ? b : a;
  }
  static int fibonacci_search(int arr[], int n, int key) {
    int offset = -1;
    int Fm2 = 0;
    int Fm1 = 1;
    int Fm = Fm2 + Fm1;
    while (Fm < n) {
      Fm2 = Fm1;
```

```
      Fm1 = Fm;
      Fm = Fm2 + Fm1;
    }
    while (Fm > 1) {
      int i = min(offset + Fm2, n - 1);
      if (arr[i] < key) {
        Fm = Fm1;
        Fm1 = Fm2;
        Fm2 = Fm - Fm1;
        offset = i;
      } else if (arr[i] > key) {
        Fm = Fm2;
        Fm1 = Fm1 - Fm2;
        Fm2 = Fm - Fm1;
      } else
        return i;
    }
    if (Fm1 == 1 && arr[offset + 1] == key)
      return offset + 1;
    return -1;
  }
  public static void main(String args[]) {
    int i, n, key;
    int arr[] = {6, 11, 19, 24, 33, 54, 67, 81, 94, 99};
    n = 10;
    key = 67;
    int pos = fibonacci_search(arr, n, key);
    if(pos >= 0)
```

```
        System.out.print("The element is found at index " + pos);
      else
        System.out.print("Unsuccessful Search");
  }
}
```

Ejemplo 2:

```
// Java program for Fibonacci Search
import java.util.*;

class Fibonacci {
    // Utility function to find minimum
    // of two elements
    public static int min(int x, int y)
    {
        return (x <= y) ? x : y;
    }

    /* Returns index of x if present, else returns -1 */
    public static int fibMonaccianSearch(int arr[], int x,
                                         int n)
    {
        /* Initialize fibonacci numbers */
        int fibMMm2 = 0; // (m-2)'th Fibonacci No.
        int fibMMm1 = 1; // (m-1)'th Fibonacci No.
        int fibM = fibMMm2 + fibMMm1; // m'th Fibonacci
```

```
/* fibM is going to store the smallest
Fibonacci Number greater than or equal to n */
while (fibM < n) {
    fibMMm2 = fibMMm1;
    fibMMm1 = fibM;
    fibM = fibMMm2 + fibMMm1;
}

// Marks the eliminated range from front
int offset = -1;

/* while there are elements to be inspected.
Note that we compare arr[fibMm2] with x.
When fibM becomes 1, fibMm2 becomes 0 */
while (fibM > 1) {
    // Check if fibMm2 is a valid location
    int i = min(offset + fibMMm2, n - 1);

    /* If x is greater than the value at
    index fibMm2, cut the subarray array
    from offset to i */
    if (arr[i] < x) {
        fibM = fibMMm1;
        fibMMm1 = fibMMm2;
        fibMMm2 = fibM - fibMMm1;
        offset = i;
    }
```

```
        /* If x is less than the value at index
        fibMm2, cut the subarray after i+1 */
        else if (arr[i] > x) {
            fibM = fibMMm2;
            fibMMm1 = fibMMm1 - fibMMm2;
            fibMMm2 = fibM - fibMMm1;
        }

        /* element found. return index */
        else
            return i;
    }

    /* comparing the last element with x */
    if (fibMMm1 == 1 && arr[n-1] == x)
        return n-1;

    /*element not found. return -1 */
    return -1;
}

// driver code
public static void main(String[] args)
{
    int arr[] = { 10, 22, 35, 40, 45, 50,
                80, 82, 85, 90, 100,235};
    int n = 12;
    int x = 235;
```

```
    int ind = fibMonaccianSearch(arr, x, n);
      if(ind>=0)
      System.out.print("Found at index: "+ind);
    else
      System.out.print(x+" isn't present in the array");
  }
}
```

Análisis de eficiencia

En el caso de la memoria de acceso aleatorio, esta técnica puede reducir el tiempo de búsqueda, además de que solo sumas y restas, mientras que la búsqueda binaria requiere de operaciones de multiplicación y división.

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

Caso promedio

En promedio, reducimos el espacio de búsqueda en 1/3, por lo que la complejidad de tiempo de caso promedio del algoritmo de búsqueda de Fibonacci es O(log n)

Mejor caso

Podemos decir fácilmente que la complejidad temporal del mejor caso de la búsqueda de Fibonacci es O(1) Encontraremos este caso cuando la clave de búsqueda sea el primer elemento que empecemos a comparar.

Peor caso

Podemos ver que el peor caso para el método de búsqueda de Fibonacci ocurre cuando la clave siempre está presente en el subarreglo más grande. Su complejidad de tiempo en el peor de los casos es O(log n).

Complejidad en el tiempo y complejidad espacial

La complejidad del tiempo es O (log N) donde la base del logaritmo es 3.

La complejidad espacial de este algoritmo es O(1) porque no se utiliza ningún espacio adicional aparte de las variables temporales.

IMPLEMENTACION Y PRUEBA DEL ALGORITMO DEL METODO DE BUSQUEDA DE FIBONACCI

METODO AGREGAR ELEMENTOS AL ARREGLO

```
int ar[]=new int[10],n,x,a,b,c,offset;
public void agregar()
{
int i;
String cad="";
n=Tools.leerInt("\nIngresa el numero de elementos para el array: ");

for(i=0;i<n;i++) {
ar[i]=Tools.leerInt("\nIngresa los elementos del arreglo: ");
    cad+= i + "[" + ar [ i ]+ "]" + "\n" ;
}
Tools.imprime("Datos del arreglo\n "+cad);

}
```

Aquí en esta parte vamos a agregar elementos al arreglo, primero preguntamos cuantos valores vamos a insertar, y después de ello ingresamos los elementos que queramos para que posteriormente se impriman en pantalla.

METODO BUSQUEDA FIBONACCI

```
public void busquedaFibonacci(){
int i;
x=Tools.leerInt("\nIngresa el elemento a buscar: ");
a=0;
b=1;
c=a+b;
while (c<n){
a = b;
b = c;
c = a+b;
}
offset = -1;
while(c > 1){
i=Math.min(offset+a, n-1);
if (ar[i] < x){
c = b;
b = a;
a = c - b;
offset = i;
}
else if (ar[i] > x){
c= a;
b = b -a;
a= c - b;
}
```

```
else{
    Tools.imprime("El elemento "+x+" se encuentra en el índice "+i);
return;
}
}
if(b!=0 && ar[offset+1]==x){
    Tools.imprime("El elemento "+x+" se encuentra en el índice "+offset);
    return;
}
Tools.imprimeErrorMsje("Elemento no encontrado");
}
```

Ahora bien explicaremos un poco para este pequeño código, Primero, el elemento buscado se comparará con el rango F(m - 2), si el valor coincide, se imprimirá la posición del elemento. Si x es menor que el elemento, la tercera variable de Fibonacci se moverá dos Fibonacci hacia abajo, lo que eliminará los 2/3 de la lista no buscada.

Si el valor buscado es mayor que el elemento, la tercera variable de Fibonacci se moverá un Fibonacci hacia abajo, lo que eliminará 1/3 del valor no buscado del frente. Para esto ponemos un ejemplo aplicando lo anterior dicho, para que quede más claro:

Supongamos que las listas de elementos son: {50,58,64,74,80,100,120}

Supongamos que el elemento es 100 que se va a buscar . El valor de n es 7

Entonces, el número de Fibonacci más pequeño que es igual o mayor que 7 es 8 . F(m) = 8, F(m - 1) = 5 y F(m - 2) = 3

Primera compensación = -1.

i = min(compensación + 3, n - 1).

= mínimo (-1 + 3, 6)

=mín (2,6)

= 2

Entonces, el valor del índice 2 es 64 y se comparará con el elemento buscado (100). Como 64 es menor que 100

F(m) = 5, F(m - 1) = 3, F(m - 2) = 2 y el desplazamiento = 2

Por lo tanto, i = min (compensación + 3, n - 1)

= mínimo (2 + 3, 6)

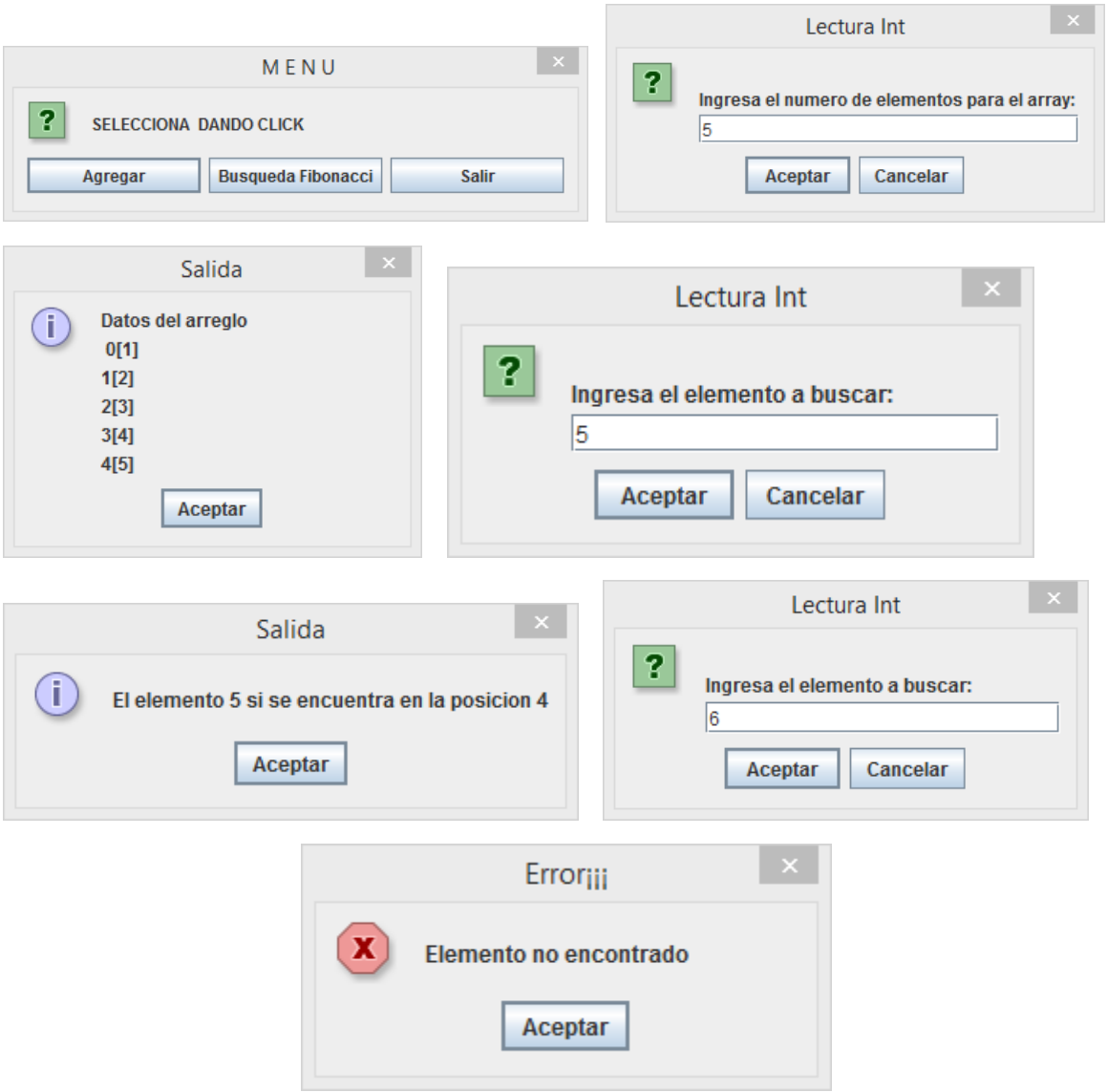
= mínimo (5, 6)

= 5

De nuevo , el valor del índice 5 es100 que es lo mismo que el elemento buscado 100 . A medida que los elementos coincidan, se imprimirá la posición de este elemento. Sino llegara a encontrar el dato dira que el elemento no fue encontrado.

EJECUTABLE METODO BUSQUEDA FIBONACCI

Para esto hacemos la ejecución del método para ver si funciona, y también ver si encuentra o no el dato



Con este resultado nos damos cuenta de que el método si es funcional y si encontró el dato y además en que posición se encuentra. Sino llegara a existir un x dato, mandara un mensaje de que el dato no se encuentra.

CONCLUSION

Una de las funciones que con mayor frecuencia se utiliza en los sistemas de información, es el de las consultas a los datos, se hace necesario utilizar algoritmos, que permitan realizar búsquedas de forma rápida y eficiente. La búsqueda, se puede decir que es la acción de recuperar datos o información, siendo una de las actividades que más aplicaciones tiene en los sistemas de información, más formalmente se puede definir como “La operación de búsqueda sobre una estructura de datos es aquella que permite localizar un nodo en particular si es que éste existe”.

Un aprendizaje que se obtuvo al realizar esta investigación es que se conoció cada uno de los tipos de métodos de búsqueda ya que al momento de hacer un programa extenso podríamos utilizar algunos de estos métodos y así encontrar la información más rápida y no perder tanto tiempo. Al comparar con las demás definiciones y funciones de cada uno de los métodos de búsqueda pude notar que este método es más eficiente y entendible que los demás. Es algoritmo conocido como Quicksort (ordenación rápida) recibe el nombre de su autor, Tony Hoare. La idea del algoritmo es simple, se basa en la división em particiones de la lista a ordenar, por lo que se puede considerar que aplica la técnica divide y vencerás. El método es, posiblemente, el más pequeño de código, más rápido, más elegante, más interesante y eficiente de los algoritmos de ordenación conocidos.

BIBLIOGRAFIA

- Jindal, H. (2021a, March 11). Búsqueda binaria. Delft Stack. https://www.delftstack.com/es/tutorial/algorithm/binary-search/
- Jindal, H. (2021b, March 11). Búsqueda de interpolación. Delft Stack. https://www.delftstack.com/es/tutorial/algorithm/interpolation-search/
- Jindal, H. (2021c, March 11). Búsqueda exponencial. Delft Stack. https://www.delftstack.com/es/tutorial/algorithm/exponential-search/
- Jindal, H. (2021d, March 11). Búsqueda lineal. Delft Stack. https://www.delftstack.com/es/tutorial/algorithm/linear-search/
- Jindal, H. (2021e, March 11). Saltar búsqueda. Delft Stack. https://www.delftstack.com/es/tutorial/algorithm/jump-search/
- Wikipedia contributors. (n.d.). Algoritmo Knuth-Morris-Pratt. Wikipedia, The Free Encyclopedia. https://es.wikipedia.org/w/index.php?title=Algoritmo_Knuth-Morris-Pratt&oldid=147437435
- Proyecto: Sistema de ayuda al C. (s. f.). https://ccia.ugr.es/~jfv/ed1/c/cdrom/cap5/f_cap56.htm
- Algoritmos de Búsqueda. (s. f.-b). http://artemisa.unicauca.edu.co/~nediaz/EDDI/cap02.htm