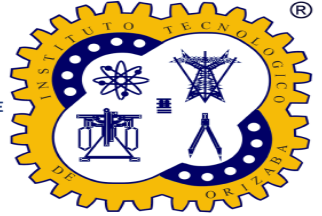




EDUCACIÓN
SECRETARÍA DE EDUCACIÓN PÚBLICA



TECNM
TECNOLOGICO NACIONAL DE
MÉXICO



HOJA DE PRESENTACION:

MATERIA: ESTRUCTURA DE DATOS
INGIENERIA INFORMATICA
Métodos de búsqueda

EQUIPO:

Castro Ramón David Alejandro 20010329

Martínez Ramos Rodrigo 20010347

Carrillo Ávila Juan Pablo 20010327

Guzmán Lino Roberto Rafael 20010339

GRUPO: 4PM – 5PM HRS CLAVE: 3a3A

Fecha de entrega: 22-05-2023

Marco teórico:

Los métodos de búsqueda se utilizan para localizar un elemento específico dentro de una estructura de datos, como un arreglo, una lista enlazada, un árbol, etc. Estos métodos son fundamentales para acceder y manipular la información almacenada en estas estructuras de manera eficiente. La elección del método de búsqueda adecuado depende de las características de la estructura de datos y la eficiencia requerida para la búsqueda en un contexto específico.

A continuación, se presentan algunos métodos de búsqueda comunes utilizados en estructuras de datos:

Búsqueda Lineal

Trata 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.

Para la búsqueda lineal, se consideran los siguientes pasos a seguir:

- Toma el elemento que deseas buscar.
- Comienza desde el primer elemento de la lista y compáralo con el elemento buscado.
- Si el elemento coincide con el buscado, se ha encontrado el resultado y el proceso termina.
- Si el elemento no coincide, pasa al siguiente elemento de la lista y repite el paso 3.
- Continúa este proceso hasta que encuentres el elemento buscado o hayas recorrido toda la lista.
- Si llegas al final de la lista sin encontrar el elemento, se considera que el elemento no está presente.

Ejemplos de búsqueda Lineal:

```
public class busquedas {

    public static boolean secuencialLineal(int a[], int valor){
        boolean existe=false;

        for (int i = 0;i<a.length;i++){
            if(valor==a[i])
                existe = true;
        }

        return existe;
    }
}
```

```
sel=boton(menu);
switch(sel){
    case "Secuencial Lineal":
        int SL[] = {1,2,3,4,5,6,7,8,9,10};
        toolsList.imprimePantalla("Busqueda de un numero del 1 al 10.");
        if(busquedas.secuencialLineal(SL, toolsList.leerByte("Numero a buscar:")))
            toolsList.imprimePantalla("El elemento existe");
        else
            toolsList.imprimePantalla("El elemento no existe");
        break;
}
```

La búsqueda lineal tiene una eficiencia de tiempo de $O(n)$, donde "n" es el tamaño de la lista o arreglo en el que se está realizando la búsqueda. Esto significa que el tiempo de ejecución aumenta linealmente con el tamaño de la lista.

La búsqueda lineal es adecuada cuando la lista o arreglo no está ordenado o cuando el tamaño de la lista es pequeño. Sin embargo, puede volverse ineficiente en listas muy grandes, ya que se deben examinar todos los elementos de la lista en el peor caso.

En términos de eficiencia de espacio, la búsqueda lineal es muy eficiente, ya que solo requiere una cantidad constante de espacio adicional para almacenar las variables de control utilizadas en el proceso de búsqueda.

1. **Peor caso:** El peor caso ocurre cuando el elemento buscado no está presente en el arreglo o se encuentra en la última posición. En este escenario, la búsqueda lineal debe recorrer todos los elementos del arreglo hasta llegar al final o encontrar el valor. Por lo tanto, en el peor caso, la búsqueda lineal tiene una complejidad temporal de $O(n)$, donde n es la longitud del arreglo. Esto significa que el tiempo de ejecución crece linealmente con respecto al tamaño del arreglo.
2. **Promedio caso:** En el promedio caso, asumimos que el elemento buscado tiene igual probabilidad de estar en cualquier posición del arreglo. Si consideramos n elementos en el arreglo, en promedio, se necesitará recorrer la mitad del arreglo para encontrar el valor buscado o determinar que no está presente. Por lo tanto,

en el promedio caso, la complejidad temporal de la búsqueda lineal Sigue siendo $O(n)$.

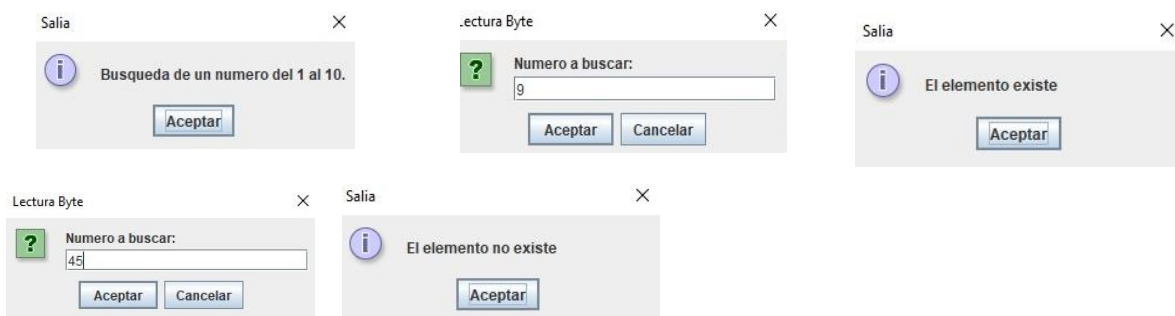
3. **Mejor caso:** El mejor caso ocurre cuando el elemento buscado se encuentra en la primera posición del arreglo. En este escenario, la búsqueda lineal solo necesita realizar una comparación antes de encontrar el valor. Por lo tanto, en el mejor caso, la complejidad temporal de la búsqueda lineal es $O(1)$, es decir, constante. Sin embargo, es importante tener en cuenta que el mejor caso es poco común y no representa el rendimiento típico de la búsqueda lineal.

- **Complejidad en el tiempo**

El algoritmo recorre secuencialmente los elementos del arreglo hasta encontrar el valor buscado o llegar al final del arreglo. En el peor caso, debe recorrer todos los elementos, lo cual implica una complejidad temporal de $O(n)$, donde n es la longitud del arreglo. Esto significa que el tiempo de ejecución aumenta linealmente con el tamaño del arreglo.

- **Complejidad en el espacio**

La búsqueda lineal no requiere memoria adicional más allá del arreglo en el que se realiza la búsqueda. Por lo tanto, la complejidad en el espacio de la búsqueda lineal es $O(1)$, es decir, constante. No importa cuántos elementos haya en el arreglo, la cantidad de memoria utilizada por el algoritmo de búsqueda lineal no cambia.



“BÚSQUEDA BINARIA”

EN QUE CONSISTE: :

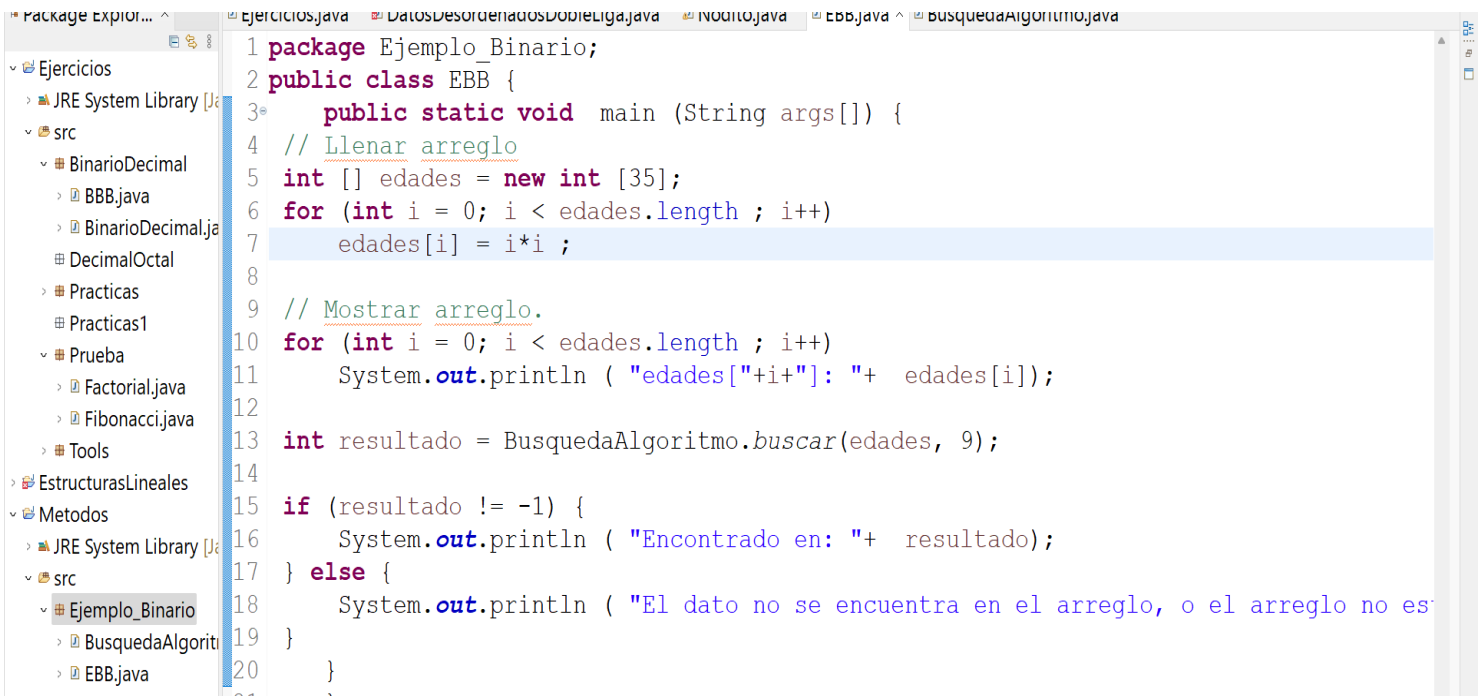
Es un algoritmo eficiente para encontrar un elemento en una lista ordenada de elementos.

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.

IMPLEMENTADO EN UN ALGORITMO: :

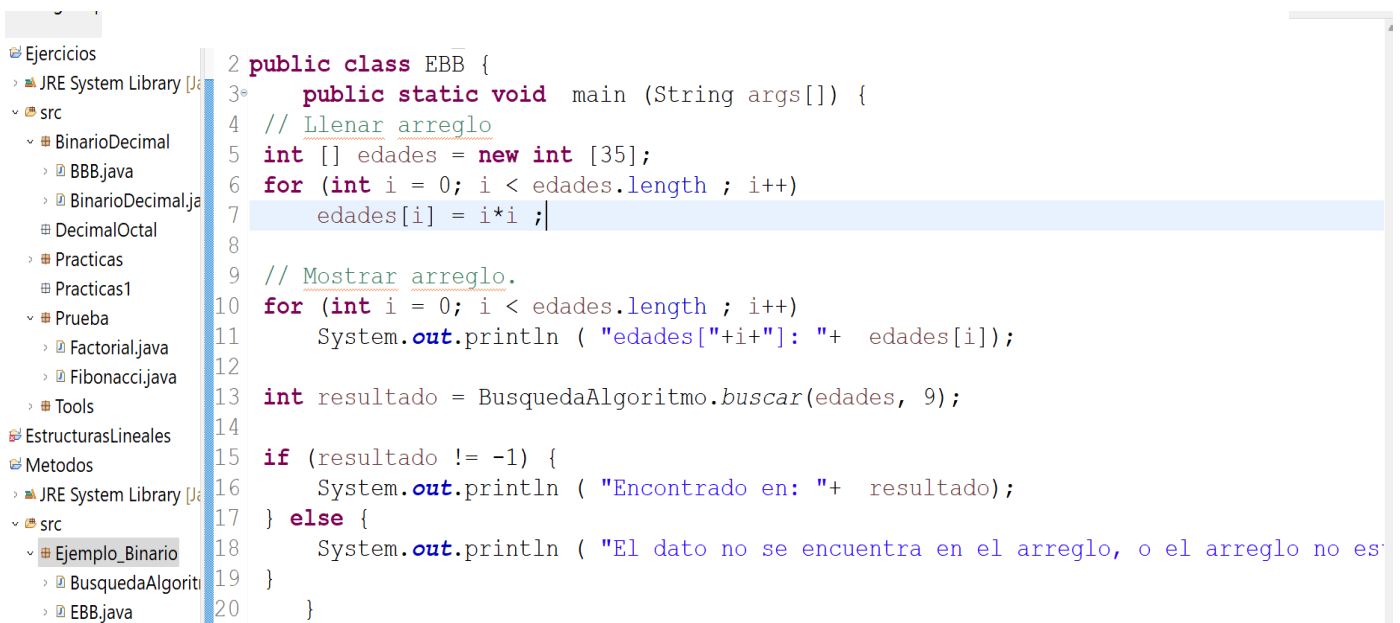
Se utiliza una función estática de la clase BusquedaAlgoritmo.

Recordar que para que funcione correctamente los valores del arreglo deben estar ordenados.



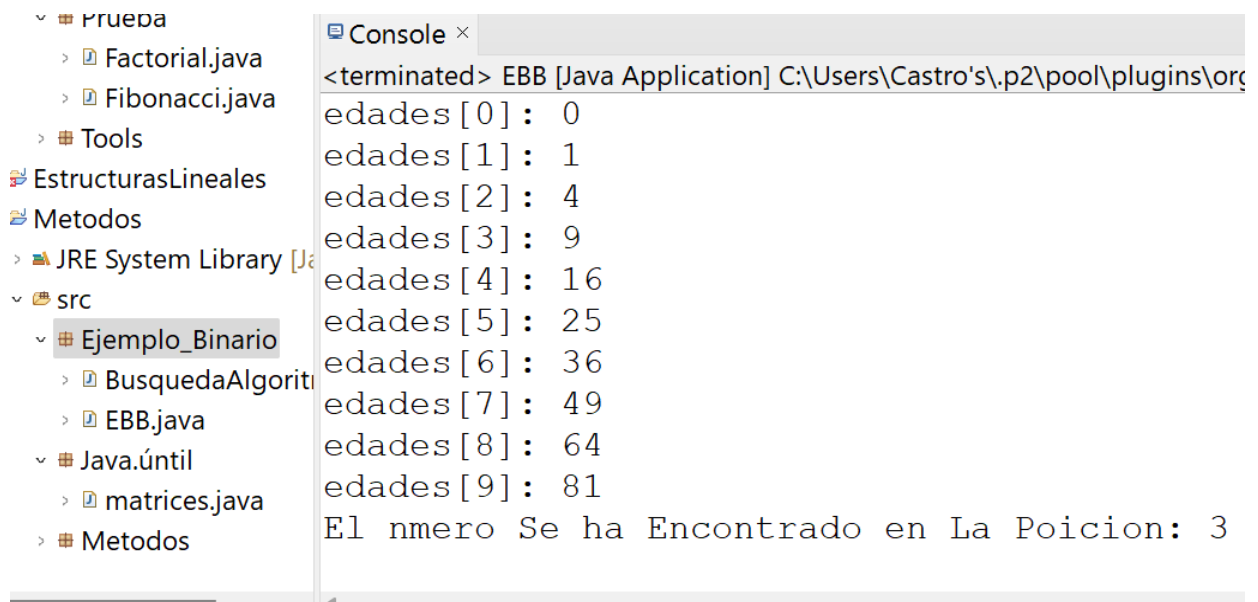
```
1 package Ejemplo_Binario;
2 public class EBB {
3     public static void main (String args[]) {
4         // Llenar arreglo
5         int [] edades = new int [35];
6         for (int i = 0; i < edades.length ; i++)
7             edades[i] = i*i ;
8
9         // Mostrar arreglo.
10        for (int i = 0; i < edades.length ; i++)
11            System.out.println ( "edades["+i+"]: " + edades[i]);
12
13        int resultado = BusquedaAlgoritmo.buscar(edades, 9);
14
15        if (resultado != -1) {
16            System.out.println ( "Encontrado en: " + resultado);
17        } else {
18            System.out.println ( "El dato no se encuentra en el arreglo, o el arreglo no es");
19        }
20    }
21 }
```

Una vez que se puedan ordenar los números será mas fácil para la búsqueda encontrar el número que se necesita.



```
2 public class EBB {
3     public static void main (String args[]) {
4         // Llenar arreglo
5         int [] edades = new int [35];
6         for (int i = 0; i < edades.length ; i++)
7             edades[i] = i*i ;
8
9         // Mostrar arreglo.
10        for (int i = 0; i < edades.length ; i++)
11            System.out.println ( "edades["+i+"]: " + edades[i]);
12
13        int resultado = BusquedaAlgoritmo.buscar(edades, 9);
14
15        if (resultado != -1) {
16            System.out.println ( "Encontrado en: " + resultado);
17        } else {
18            System.out.println ( "El dato no se encuentra en el arreglo, o el arreglo no es");
19        }
20    }
21 }
```

Entonces se ejecuta el código y vemos como se cumplen las condiciones, se ordenan los números y además se muestra en que línea o donde se encuentra de acuerdo a la posición, en este caso se busca el numero 9 el cual se encuentra en la posición #3.



The screenshot shows an IDE interface. On the left is a project explorer with a tree view containing folders like 'Prueba', 'Tools', 'EstructurasLineales', 'Metodos', 'JRE System Library', 'src', and 'Ejemplo_Binario'. The 'src' folder is expanded, showing files like 'BusquedaAlgoritmo.java', 'EBB.java', 'Java.útil', 'matrices.java', and 'Metodos'. On the right is a 'Console' window titled 'Console x'. It displays the output of a Java application named 'EBB'. The output shows an array of ages ('edades') indexed from 0 to 9, with values: 0, 1, 4, 9, 16, 25, 36, 49, 64, 81. The final line of output states: 'El nmero Se ha Encontrado en La Poicion: 3'.

```
<terminated> EBB [Java Application] C:\Users\Castro's\p2\pool\plugins\org
edades [ 0 ] : 0
edades [ 1 ] : 1
edades [ 2 ] : 4
edades [ 3 ] : 9
edades [ 4 ] : 16
edades [ 5 ] : 25
edades [ 6 ] : 36
edades [ 7 ] : 49
edades [ 8 ] : 64
edades [ 9 ] : 81
El nmero Se ha Encontrado en La Poicion: 3
```

En este caso igual podemos implementar este método de búsqueda en nuestro proyecto de dato simple:

```

Ejecutar.java Salto.java DatoSimple.java Test.java
71         if (datos[prev] == x) {
72             return prev;
73         }
74     }
75     return -1;
76 }
77
78 public int buscarBin(Object dato) {
79     int inicio = 0;
80     int fin = datos.length - 1;
81     int pos;
82     while (inicio <= fin) {
83         pos = (inicio+fin) / 2;
84         if ( datos[pos] == dato )
85             return pos;
86         else if ((int) datos[pos] < (int) dato ) {
87             inicio = pos+1;
88         } else {
89             fin = pos-1;
90         }
91     }
92     return -1;
93 }

```

```

Ejecutar.java Salto.java DatoSimple.java Test.java
40     }else {
41         pos = (byte) obj.SaltoBusqueda(ToolsPanel.leerInt(""));
42         if(pos!=-1) {
43             ToolsPanel.imprimePantalla("Se encuentra en la posicion: "+pos);
44         }else {
45             ToolsPanel.imprimeError("Dato no encontrado");
46         }
47     }
48     break;
49     case "BusquedaBin":
50         if(obj.validaVacio()) {
51             ToolsPanel.imprimeError("Array vacio");
52         }else {
53             pos = (byte) obj.buscarBin(ToolsPanel.leerInt("Que dato buscas"));
54             if(pos!=-1) {
55                 ToolsPanel.imprimePantalla("Se encuentra en la posicion: "+pos);
56             }else {
57                 ToolsPanel.imprimeError("Dato no encontrado");
58             }
59         }
60     break;

```

MENU

?

SELECCIONA DANDO CLICK

Agregar

Imprimir

Buscar

BuscarSalto

BuscarBin

Eliminar

Salir

ANALISIS DE EFICIENCIA: :

Cuando tenemos que buscar un dato dentro de un arreglo de datos, uno de los métodos más eficaces de hacerlo es mediante el **algoritmo de búsqueda binaria**, capaz de encontrar un dato en arreglos de gran tamaño en tan sólo unos instantes, gracias a la poca cantidad de operaciones que realiza en memoria, siendo muy efectivo y consumiendo muy pocos recursos.

Como la **búsqueda binaria** divide un problema a la mitad cada vez que se ejecuta, su complejidad es de $O(\log n)$, una de las más efectivas que podemos encontrar en algoritmos. Podemos compararla con la **búsqueda lineal**, de complejidad $O(n)$ y también muy utilizada por lo fácil que puede expresarse en código.

En resumen, se verifica si el valor buscado es igual al de la mitad del arreglo; si es igual, termina la búsqueda; si es mayor, se descarta el lado izquierdo del arreglo y se repite la búsqueda; si es menor, se descarta el lado derecho del arreglo y se repite la búsqueda.

ANALISIS DE CASOS:

En la mayoría de los casos, la **búsqueda binaria** es uno de los métodos de búsqueda más rápidos que podemos implementar, ya que, al ir reduciendo el área de búsqueda por la mitad con cada iteración, reduce el tamaño del problema logarítmicamente, por lo que se dice que su complejidad es de $O(\log n)$.

- Caso promedio

Cuando realizamos la búsqueda binaria, buscamos en una mitad y descartamos la otra mitad, reduciendo el tamaño del array a la mitad cada vez.

La expresión para la complejidad del tiempo viene dada por la recurrencia.

$$T(n) = T(n/2) + k, \text{ k is a constant.}$$

Este resultado de esta recurrencia da $\log n$, y la complejidad temporal es del orden de $O(\log n)$. Es más rápido que la búsqueda lineal y la búsqueda por salto.

- Mejor caso

El mejor de los casos ocurre cuando el elemento del medio 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 $O(\log n)$.

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.

COMPLEJIDAD DEL TIEMPO Y ESPACIO DE LOS MÉTODOS:

ESPACIO: complejidad espacial de este algoritmo es $O(1)$, en el caso de implementación iterativa porque no requiere ninguna estructura de datos más que variables temporales.

En el caso de la implementación recursiva, la complejidad del espacio es $O(\log n)$ debido al espacio requerido por la pila de llamadas recursivas.

TIEMPO: También conocida, como **búsqueda de intervalo medio** o **búsqueda logarítmica**, es un [algoritmo de búsqueda](#) que encuentra la posición de un valor en un [array](#) ordenado.

Compara el valor con el elemento en el medio del array, si no son iguales, la mitad en la cual el valor no puede estar es eliminada y la búsqueda continúa en la mitad restante hasta que el valor se encuentre.

Aunque la idea es simple, implementar la búsqueda binaria correctamente requiere atención a algunos detalles como su condición de parada y el cálculo del punto medio de un intervalo.

Existen numerosas variaciones de la búsqueda binaria. Una variación particular ([cascada fraccional](#)) acelera la búsqueda binaria para un mismo valor en múltiples arreglos.

CONCLUSIÓN:

La **búsqueda binaria** es un algoritmo simple y muy eficaz para encontrar elementos en un arreglo ordenado de datos.

Aunque actualmente los [lenguajes de programación más utilizados](#) de mas alto nivel como Python, Ruby o C# cuentan con métodos de búsqueda integrados muy optimizados, es útil conocer como implementar la búsqueda binaria para poder adaptar este algoritmo a tus necesidades, además de que podrás utilizarlo bajo cualquier entorno, sin importar si el lenguaje o entorno que tengas que utilizar cuente con las librerías donde se almacenen los métodos de búsqueda de dichos lenguajes.

Además, la búsqueda binaria, por su efectividad y simpleza.

Búsqueda de patrones de Knuth Morris Pratt:

En que consiste:

La búsqueda de patrones de Knuth Morris Pratt (KMP) es un algoritmo eficiente para buscar coincidencias en un patrón dado en una cadena de texto, este algoritmo utiliza una estrategia basada en el conocimiento de los caracteres ya comparados en el texto para evitar realizar comparaciones innecesarias.

El punto importante de este algoritmo es la construcción de una tabla de prefijos que contiene información sobre los patrones repetidos dentro del propio patrón a buscar, esto quiere decir que con este algoritmo se evita volver a comparar caracteres que ya se han verificado correctamente en el texto, esta tabla se crea durante la ejecución antes de realizar la búsqueda.

Implementado en un algoritmo:

Para esta búsqueda ya mencionamos que necesitamos dos métodos diferentes, los cuales serían:

Método de Prefijos:

```
1 public class Metodo_Knuth {  
2  
3     private int[] TablaPrefijo(String prefijo) {  
4         int[] Tablaprefijo = new int[prefijo.length()];  
5         int i = 0;  
6         int j = 1;  
7  
8         while (j < prefijo.length()) {  
9             if (prefijo.charAt(i) == prefijo.charAt(j)) {  
10                 Tablaprefijo[j] = i + 1;  
11                 i++;  
12                 j++;  
13             } else {  
14                 if (i != 0) {  
15                     i = Tablaprefijo[i - 1];  
16                 } else {  
17                     Tablaprefijo[j] = 0;  
18                     j++;  
19                 }  
20             }  
21         }  
22  
23         return Tablaprefijo;  
24     }  
-- }
```

Este método recibe como parámetro la cadena que queremos comparar con nuestro texto, así que le sacaremos una tabla de prefijos, la tabla de prefijos nos servirá como referencia para que en la búsqueda no se vuelvan a repetir ciertos caracteres ahorrando así tiempo, empezando desde el último carácter en el que coincidieron.

Método de comparación:

```

26 public int BuscarPatron(String texto, String patron) {
27     int[] Tablaprefijo = TablaPrefijo(patron);
28     int i = 0;
29     int j = 0;
30
31     while (i < texto.length() && j < patron.length()) {
32         if (texto.charAt(i) == patron.charAt(j)) {
33             i++;
34             j++;
35         } else {
36             if (j != 0) {
37                 j = Tablaprefijo[j - 1];
38             } else {
39                 i++;
40             }
41         }
42     }
43
44     if (j == patron.length()) {
45         return i - j;
46     }
47
48     return -1;
49 }
50
51 }

```

Una vez obtenido la tabla de los prefijos se podrían empezar a comparar las cadenas, las dos se recorrerán al mismo tiempo mientras los caracteres coincidan, una vez que estos no coincidan se retrocederá la cadena del patrón hasta donde empiece a coincidir, pongamos un ejemplo para ponerlo más claro:

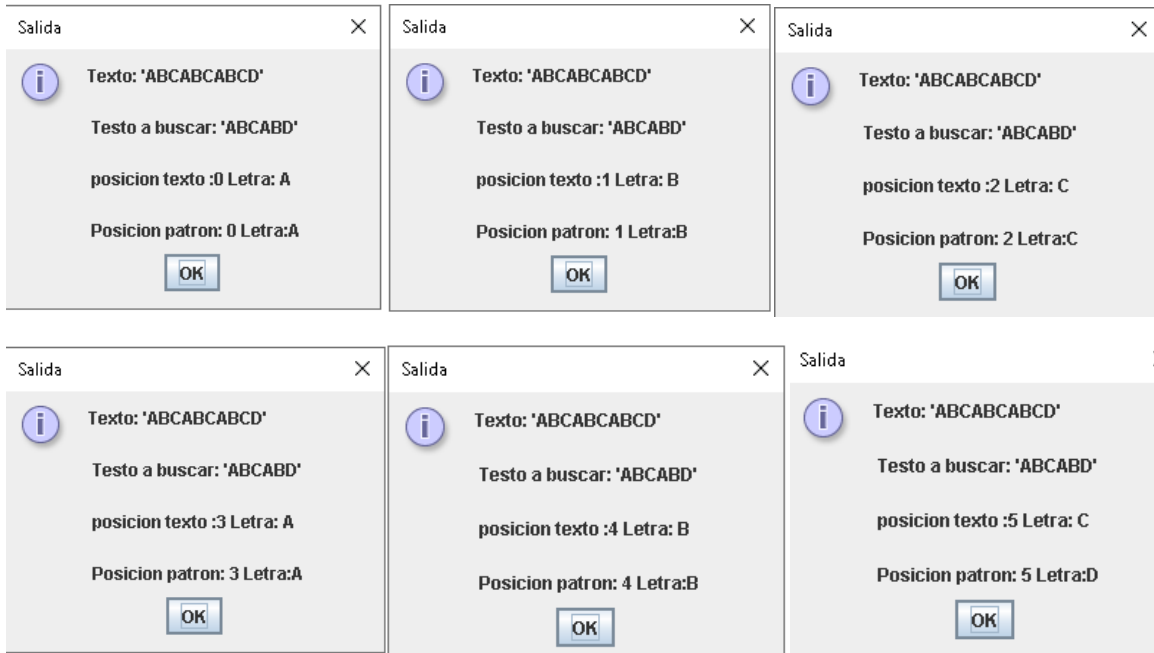
Supongamos que nuestra cadena es: "ABCABCABCD"

Y nuestra cadena a buscar es: "ABCABD"

Primero se sacará la tabla de prefijo la cual nos dirá a que posición retroceder sino coinciden en cierto punto esta tabla saldrá como: [0,0,0,1,2,0] sale así porque si vemos en nuestra cadena a buscar la letra A se repite dos veces así que eso quiere decir que nos podemos saltar hasta la posición 1 ya que sabemos que en la posición 0 se encuentra una A, después vemos que la letra B igual se repite entonces podemos retroceder hasta la posición 2 ya que sabemos que en la 1 se encuentra la b.

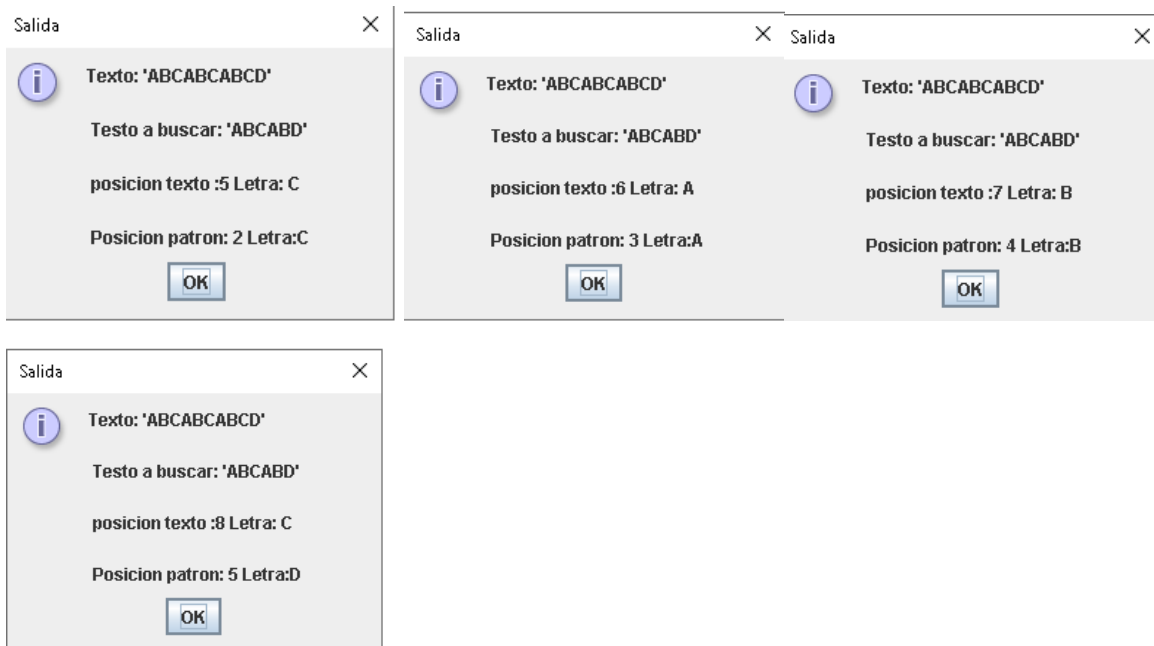
De esta manera nos aseguramos de no volver a repetir la búsqueda en los patrones ya conocidos.

Como podemos ver las cadenas logran avanzar hasta la posición numero 5:

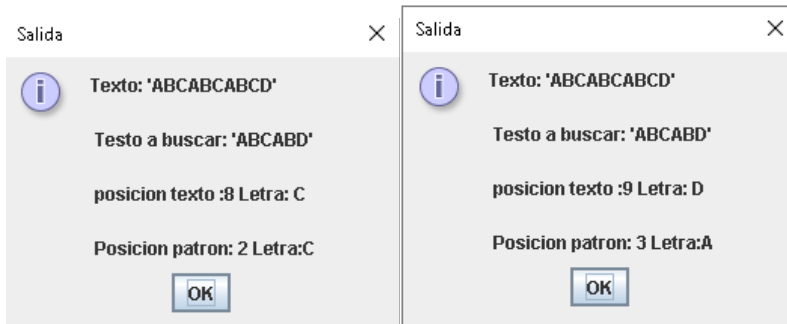


Una vez llegada a esta posicion logramos ver que la letra del tecto en la posicion 5 es una “C” y en la del patron es la “D” esto quiere decir que la cadena no ha coincidido asi que se regresa a la ultima posicion donde se repetia esta posicion recuerden que nos la da la tabla del prefijo que dice que la letra b se reperia asi que podemos retroceder a la posicion 2 en la cual se encuentra la letra “C”.

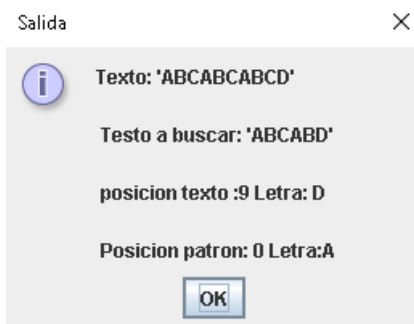
Y se vuelven a comparar a partir de esa posicion:



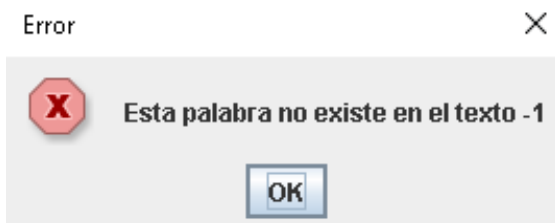
Una vez más volvemos a comprobar que en la posición 5 del patrón no coincide con nuestra cadena de texto, entonces volvemos a retroceder en nuestra tabla de prefijo la cual nos dice que volvamos a la posición 2 o sea a la letra "C".



Ahora podemos ver que logro avanzar solo una vez y no coincidio así que volvemos a retroceder en la tabla del prefijo y esta vez nos dice que retrocedamos a la posición 0 donde se encuentra la letra "A".



Pero ni siquiera esta coincide y la cadena principal ya llegó al final esto significa que no coinciden en ningún momento completamente las dos cadenas.



Analisis de eficiencia:

La búsqueda de Knuth es altamente eficiente ya que maneja una complejidad de $O(n+m)$ donde n es la longitud del texto y m es la longitud del patrón a buscar, En comparación con la búsqueda ordinaria la cual es $O(n*m)$ como podemos ver este número se elevaría mucho a comparación con esta búsqueda, esta capacidad se

debe gracias a evitar las comparaciones innecesarias al utilizar la tabla de prefijos, esto permite saltarse las partes ya conocidas.

Como pudimos ver en el ejemplo anterior cada que una letra no coincidía no teníamos que regresar la frase a buscar hasta el inicio sino que solo lo regresábamos hasta la parte ya conocida.

Aun que tenemos que decir que esta búsqueda solo funciona si la palabra tiene caracteres repetidos, esto quiere decir que si tenemos una cadena a buscar sin caracteres repetidos la búsqueda será igual que si fuera una búsqueda común ya que no abra una tabla de prefijos que nos ayude.

Con lo cual este método de búsqueda solo funciona para cadenas grandes donde sabemos que hay varios caracteres que se repiten.

Analisis de casos:

Mejor de los casos: lo mejor que podría pasar para que fue el mejor de los casos sería que la palabra a buscar coincidiera al inicio de la cadena del texto, dando al instante el resultado y sin tener que ocupar la tabla de prefijos como por ejemplo:

Texto: "ABCDABCABD"

Patrón: "ABCD"

En este caso las primeras 4 posiciones coinciden a la perfección, con lo cual no necesitaríamos seguir recorriendo ni una tabla de prefijos.

Caso medio:

En este caso sería como el ejemplo que hicimos al inicio

Texto: "ABCABCABCD"

Patrón: "ABCABD"

En este caso pudimos ver que si se ocupó una tabla de prefijos la cual fue eficiente para no volver a recorrer el patrón completo sino desde el último dato donde coincidió a pesar de que las cadenas nunca coinciden, la búsqueda fue eficiente gracias a que no se tuvo que recorrer por completo.

Peor de los casos: en el peor de los casos sería que ingresaran una cadena demasiado larga y con coincidencias medias lo cual haría a esta búsqueda un poco ineficiente o sería igual que un algoritmo de fuerza bruta, como por ejemplo:

Texto: "ABABACABBABABABABABCABAAAABBBBCABCABC"

Patrón: "ABCABC"

Tabla de prefijos: [0,0,0,1,2,3]

Como podemos ver en el texto se repiten demasiadas veces el conjunto "AB" esto haría que fuera mas tardado y se tendría que recorrer varias veces el patrón, teniendo solo hasta el final la coincidencia.

Complejidad del tiempo y espacio de los métodos:

Tiempo: El tiempo variara dependiendo de que tan larga sea la cadena para analizarla, al igual que el patrón a buscar y su tabla de prefijos como vimos en el mejor de los casos será sumamente rápido y en el caso medio y peor tardara un poco más, pero será un poco mejor que los métodos de fuerza bruta.

Espacio: Al igual que el tiempo esto dependerá de la cadena de texto y su patrón de búsqueda el cual creará un espacio en la memoria para almacenar la tabla de prefijos.

Conclusión:

En conclusión, podemos decir que este método de búsqueda es eficiente en cadenas de texto con patrones de búsqueda repetidos ya que no necesitaremos analizar el patrón completo constantemente, sino que solo recorreremos desde las partes ya conocidas así mejorando el tiempo de ejecución para encontrar dicho patrón, aun que en cadenas donde no existen patrones no es eficiente ya que será necesario recorrer constantemente el patrón a buscar.

“BÚSQUEDA POR SALTO DE BUSQUEDA”

EN QUE CONSISTE: :

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.

IMPLEMENTADO EN UN ALGORITMO: :

En este caso este algoritmo lo podemos implementar en nuestro proyecto dato simple:

```

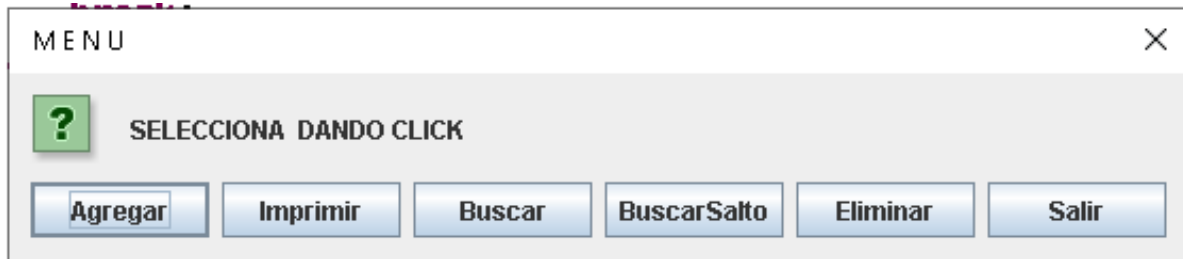
Ejecutar.java  Salto.java  DatoSimple.java  Test.java
49      }
50
51  public int SaltoBusqueda(Object x) {
52      int n = datos.length;
53      int step = (int) Math.floor(Math.sqrt(n));
54      int prev = 0;
55
56      while ((int)datos[Math.min(step, n) - 1] < (int)x) {
57          prev = step;
58          step += (int) Math.floor(Math.sqrt(n));
59          if (prev >= n) {
60              return -1;
61          }
62      }
63
64      while ((int)datos[prev] < (int)x) {
65          prev++;
66          if (prev == Math.min(step, n)) {
67              return -1;
68          }
69      }
70
71      if (datos[prev] == x) {
72          return prev;
73      }
74
75      return -1;
76  }

```

```

Ejecutar.java  Salto.java  DatoSimple.java  Test.java
26      if(obj.validaVacio()) {
27          ToolsPanel.imprimeError("Array vacio");
28      }else {
29          pos = obj.buscarSecuencial(ToolsPanel.leerInt("Dato a buscar"));
30          if(pos!=-2) {
31              ToolsPanel.imprimePantalla("Se encuentra en la posicion: "+pos);
32          }else {
33              ToolsPanel.imprimeError("Dato no encontrado");
34          }
35      }
36      break;
37  case "BuscarSalto":
38      if(obj.validaVacio()) {
39          ToolsPanel.imprimeError("Array vacio");
40      }else {
41          pos = (byte) obj.SaltoBusqueda(ToolsPanel.leerInt(""));
42          if(pos!=-1) {
43              ToolsPanel.imprimePantalla("Se encuentra en la posicion: "+pos);
44          }else {
45              ToolsPanel.imprimeError("Dato no encontrado");
46          }
47      }
48      break;

```



ANALISIS DE EFICIENCIA: :

Supongamos que tenemos el siguiente arreglo de 15 elementos: **[1, 3, 4, 7, 9, 11, 13, 16, 19, 18, 20, 22, 24, 27, 30]**

- Necesitamos saber la posición del número **22** dentro del arreglo.

Cómo determinar el número de elementos a brincar en cada iteración

Usualmente sería \sqrt{N} donde N es el total de elementos. En este caso $\sqrt{15} = 3.87$

Redondeando, serían 4.

Con este tamaño de paso y este arreglo en particular, el peor escenario serían **5** iteraciones para encontrar el número **27**.

El algoritmo sería el siguiente:

1. Se determina el número de posiciones que estaremos brincando en cada iteración, lo llamaremos **x**.
2. Inicializamos nuestro apuntador a la posición inicial, lo llamaremos **i**.
3. Se valida el elemento en la posición **i**, si es el que buscamos, termina el algoritmo.
4. Si no, brincamos x número de posiciones hacia adelante y volvemos a comparar.
5. Si el número a comparar es mayor al buscado, recorremos hacia atrás desde la posición **i** hasta encontrar el valor deseado.

El valor sería encontrado en **4** iteraciones de nuestro algoritmo, esto nos da una notación de tiempo de **O (\sqrt{n})**. Este tipo de algoritmo es

considerado ***in place***, ya que no requiere estructura de dato adicional (otro arreglo) para encontrar el resultado.

Si saltar hacia atrás en una lista toma mucho más tiempo que saltar hacia adelante, entonces se debe usar este algoritmo, es por eso que se le considera muy eficiente a la hora de realizar búsquedas dentro las estructuras de datos.

ANALISIS DE CASOS:

- 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.

- 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 \sqrt{n} , haciendo que la complejidad de tiempo sea $n/\sqrt{n} + \sqrt{n}$, es decir, \sqrt{n} . La complejidad de tiempo del algoritmo Jump Search es $O(\sqrt{n})$.

- 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 $O(\sqrt{n})$.

COMPLEJIDAD DEL TIEMPO Y ESPACIO DE LOS MÉTODOS:

ESPACIO: Supongamos que tenemos un array sin clasificar $A[]$ que contiene n elementos, y queremos encontrar un elemento X .

- Comienza desde el primer conjunto de elementos i como 0 y el tamaño del bloque m como \sqrt{n} .
- Mientras $A[\min(m,n)-1] < X$ y $i < n$.
 - Configure i como m e incremente m en \sqrt{n} .
- Si $i \geq n$ devuelve -1.
- Mientras $A[i] < X$ haz lo siguiente:
 - incremento i
 - si i es igual a $\min(m, n)$ devuelve -1
- Si $A[i] == X$ devuelve i .
- De lo contrario, devuelve -1.

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

CONCLUSIÓN:

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 la colección se ordene.

En la búsqueda por salto, saltamos en el intervalo $\sqrt{\text{arraylength}}$ hacia adelante hasta que alcanzamos 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.

Método de búsqueda de interpolación.

En que consiste la búsqueda de interpolación:

La búsqueda de interpolación es un método que podemos utilizar para obtener un valor aproximado de una variable dentro de un rango conocido, utilizando datos discretos. Lo que queremos hacer es estimar el valor desconocido basándonos en la relación lineal o no lineal entre los datos conocidos.

Existen diferentes técnicas de interpolación, estas son la interpolación lineal, la interpolación polinómica y la interpolación Spline.

Interpolación lineal: funciona asumiendo que los datos conocidos siguen una tendencia lineal y esto se utiliza para hacer la estimación.

Interpolación polinómica: Se utiliza un polinomio que se ajusta a los puntos de datos disponibles. Esta técnica se basa en suponer que los datos siguen una relación polinómica.

Interpolación Spline: A diferencia de la anterior esta divide el conjunto de datos en segmentos mas pequeños y utiliza un polinomio diferente en cada segmento.

Ejemplos de implementación del algoritmo en estructura de datos:

Interpolación lineal:

```
1 public class Interpolacion {
2
3     public static double InterpolacionLin(double[] x, double[] y, double xValue) {
4         int n = x.length;
5         for (int i = 0; i < n - 1; i++) {
6             if (x[i] <= xValue && xValue <= x[i + 1]) {
7                 double yValue = y[i] + (y[i + 1] - y[i]) * (xValue - x[i]) / (x[i + 1] - x[i]);
8                 return yValue;
9             }
10        }
11        return -1;
12    }
13
14 }
15
```

En este código hacemos la interpolación lineal, primero recibimos los parámetros los cuales serán las listas de datos que tenemos y el valor a interpolar en x.

Una vez tenemos estos datos tendremos que recorrer el arreglo x para lograr encontrar el intervalo donde se encuentra xValue, una vez que se encuentra, se calculará su valor interpolado utilizando la fórmula la cual sería: $yValue = y1 + (y2 - y1) * ((xValue - x1) / (x2 - x1))$.

Una vez interpolado este numero se devuelve el resultado aproximado, si el valor en x no se encuentra regresara un -1 indicando que ese valor no existe en x.

Ejemplo:

Supongamos que tenemos datos:

```
double[] x = {1.0, 2.0, 3.0, 4.0, 5.0}  
double[] y = {10.0, 20.0, 30.0, 40.0, 50.0}
```

```
double xValue = 4.5
```

Ahora el programa se encargaría de estimar el valor en la lista x que nosotros le demos a buscar, si por ejemplo le queremos dar un 4.5 el resultado seria 45.0 como podemos ver el programa intenta predecir el dato que estará en y

Análisis de eficiencia:

Es importante tener en cuenta que la eficiencia de este método de búsqueda es eficiente en cuanto al tema de tiempo y espacio sin embargo hay que tener en cuenta que puede verse afectado por factores externos como puede ser la calidad y distribución de los datos, así como la precisión requerida en la estimación interpolada.

Mejor de los casos:

Seria cuando el valor de la interpolación se encuentra exactamente en uno de los puntos de datos conocidos, en este caso no seria necesario recorrer los datos ni realizar cálculos adicionales.

Caso medio:

Seria cuando el valor de interpolación se encuentra dentro del rango de los datos conocidos, pero no coincide exactamente con ninguno de ellos, en este caso se necesitaría recorrer los datos para encontrar el intervalo adecuado y luego realizar los cálculos de interpolación lineal.

Peor de los casos:

Esto ocurriría cuando el valor de interpolación se encuentra fuera del rango de los datos conocidos, en este caso se recorrerán todos los puntos de datos sin encontrar el intervalo adecuado, lo que llevara a la iteración completa de los datos.

Complejidad de tiempo y espacio:

Complejidad temporal: La búsqueda requiere iterar a través de los datos conocidos para encontrar el intervalo adecuado donde se encuentra el valor de interpolación, esto implica comparación para cada par de puntos consecutivos, por lo tanto, la complejidad es $O(n)$ donde "n" es el numero de puntos de datos conocidos.

Complejidad espacial: La búsqueda no requiere almacenar ninguna estructura de datos adicional más allá de los datos conocidos, los cuales ya se asumen disponibles, por lo tanto, es $O(1)$.

Búsqueda Exponencial

La búsqueda exponencial es un algoritmo de búsqueda utilizado en estructuras de datos ordenadas. Se basa en el principio de acelerar la búsqueda mediante la exploración de posiciones exponencialmente distantes en lugar de realizar pasos lineales.

El algoritmo de búsqueda exponencial funciona de la siguiente manera:

- A. Comienza con un rango inicial que cubre toda la estructura de datos ordenada.
- B. Compara el elemento buscado con el elemento en el medio del rango.
 - Si son iguales, se ha encontrado el elemento y la búsqueda termina.
 - Si el elemento buscado es menor, se reduce a la mitad el rango y se repite el paso 2 con la mitad inferior.
 - Si el elemento buscado es mayor, se duplica el tamaño del rango y se repite el paso 2 con el nuevo rango.

Este proceso se repite hasta encontrar el elemento buscado o hasta que el rango se reduce a un solo elemento. Si el rango se reduce a un solo elemento y ese elemento no coincide con el buscado, significa que el elemento no está presente en la estructura de datos.

Ejemplo de implementación de búsqueda exponencial:

```

public static int busquedaExponencial(int[] a, int valor) {
    int tamaño = a.length;
    if (a[0] == valor) {
        return 0;
    }

    int i = 1;
    while (i < tamaño && a[i] <= valor) {
        i *= 2;
    }

    return busquedaBinaria(a, valor, i / 2, Math.min(i, tamaño - 1));
}

public static int busquedaBinaria(int[] a, int clave, int inicio, int fin) {
    if (fin >= inicio) {
        int medio = inicio + (fin - inicio) / 2;

        if (a[medio] == clave) {
            return medio;
        }

        if (a[medio] > clave) {
            return busquedaBinaria(a, clave, inicio, medio - 1);
        }

        return busquedaBinaria(a, clave, medio + 1, fin);
    }

    return -1;
}

```

```

case "Exponencial":
    int[] EXP = {2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26};
    toolsList.imprimePantalla(busquedas.imprimeOrdenados(EXP));
    if (busquedas.interpolacion(EXP, toolsList.leerInt("Numero a buscar:")) >= 0)
        toolsList.imprimePantalla("El dato existe");
    else
        toolsList.imprimePantalla("El dato no existe.");
    break;

```

En términos de eficiencia, la búsqueda exponencial puede ser más rápida que la búsqueda binaria tradicional en ciertos casos, especialmente cuando el elemento buscado está más cerca del principio del arreglo. Sin embargo, en casos en los que el elemento buscado está cerca del arreglo final, la búsqueda exponencial puede requerir más iteraciones que la búsqueda binaria para encontrar el elemento.

El análisis de los casos de la búsqueda exponencial es el siguiente:

1. Mejor de los casos:

El elemento buscado se encuentra en el primer intento, es decir, en el índice 0. La complejidad en tiempo es $O(1)$, ya que solo se necesita una comparación.

2. Caso medio:

El elemento buscado está ubicado en alguna posición aleatoria dentro del arreglo. La complejidad en tiempo es $O(\log n)$, donde "n" es el tamaño del arreglo. Esto se debe a que la búsqueda se reduce a la búsqueda binaria en un rango específico del arreglo.

3. Peor de los casos:

El elemento buscado no está presente en el arreglo. La complejidad en tiempo es $O(\log n)$, ya que la búsqueda se reduce a la búsqueda binaria en un rango específico del arreglo, pero en última instancia no se encuentra el elemento.

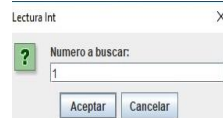
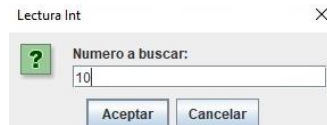
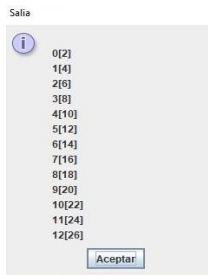
En cuanto a la complejidad en el tiempo y la complejidad espacial:

4. Complejidad en el tiempo:

En el peor de los casos, la complejidad en el tiempo es $O(\log n)$, donde "n" es el tamaño del arreglo. Esto se debe a que la búsqueda se reduce a la búsqueda binaria en un rango específico. Sin embargo, en el mejor de los casos, la complejidad en el tiempo es $O(1)$, ya que el elemento buscado se encuentra en el primer intento.

5. Complejidad espacial:

La complejidad espacial es $O(1)$ ya que no se requiere espacio adicional dependiente del tamaño del arreglo. Solo se utilizan variables adicionales para mantener el estado de la búsqueda.



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 implementación en Java de la búsqueda de Fibonacci:

```
case "Busqueda Fibonacci":
    int BF[] = {10,13,15,26,28,50,56,88,94,127,159,356,480,567,689,699,780};
    toolsList.imprimePantalla(busquedas.imprimeOrdenados(BF));
    if(busquedas.fibonacciSearch(BF, toolsList.leerInt("Numero a buscar:"))>=0)
        toolsList.imprimePantalla("El dato existe");
    else
        toolsList.imprimePantalla("El dato no existe.");
    break;

public static int fibonacciSearch(int[] a, int target) {
    int n = a.length;
    int fib2 = 0;
    int fib1 = 1;
    int fib = fib2 + fib1;
    //calcular fibonacci
    while (fib < n) {
        fib2 = fib1;
        fib1 = fib;
        fib = fib2 + fib1;
    }
    int dif = -1;
    while (fib > 1) {
        int i = Math.min(dif + fib2, n - 1);
        if (a[i] < target) {
            fib = fib1;
            fib1 = fib2;
            fib2 = fib - fib1;
            dif = i;
        }
        else
            if (a[i] > target) {
                fib = fib2;
                fib1 = fib1 - fib2;
                fib2 = fib - fib1;
            }
        else {
            return i;
        }
    }
    if (fib1 == 1 && a[dif + 1] == target) {
        return dif + 1;
    }
    return -1;
}
```

1. Caso medio

Reducimos el espacio de búsqueda en un tercio / dos tercios en cada iteración y, por lo tanto, el algoritmo tiene una complejidad logarítmica. La complejidad de tiempo del algoritmo de búsqueda de Fibonacci es $O(\log n)$.

2. 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 que comparamos.

3. Peor caso

El peor de los casos ocurre cuando el elemento objetivo X siempre está presente en el subarreglo más grande. La complejidad de tiempo en el peor de los casos es $O(\log n)$. Es lo mismo que la complejidad del tiempo promedio de los casos.

Complejidad en el tiempo: La complejidad en el tiempo de la búsqueda de Fibonacci es $O(\log n)$. Esto se debe a que en cada iteración, el tamaño del rango de búsqueda se reduce aproximadamente a la mitad.

Complejidad espacial: La complejidad espacial de la búsqueda de Fibonacci es $O(1)$, ya que no se requiere almacenamiento adicional en función del tamaño de los datos de entrada.

