Instituto Tecnológico De Orizaba

Estructura De Datos

Equipo 2

Integrantes:

Brayan Hernández Acevedo

Cesar Ríos Méndez

Pedro Sentíes Robles

Pedro Tezoco Cruz

Tema 6 Búsquedas Estructura de Datos

Fecha de entrega 22/05/2023

	Tema	Subtema-Reporte de Practicas
6	Métodos de búsqueda	Marco Teórico deberá de incluir los subtemas investigados: • Búsqueda lineal • Búsqueda binaria • Búsqueda de patrones de Knuth Morris Pratt • Saltar búsqueda • Búsqueda de interpolación • Búsqueda exponencial • Búsqueda de fibonacci

Algoritmos de búsqueda secuencial lineal

Algoritmos de búsqueda secuencial lineal usando datos desordenados (método analizado e implementado en el tema 1, e implementado en listas enlazadas simples y dobles)

Investigar

- En que consiste la búsqueda secuencial
- Ejemplos de implementaciones del algoritmo en las estructuras de datos: memoria estática, memoria dinámica (listas) y archivos.
- Análisis de eficiencia
- Análisis de los casos: mejor de los casos, caso medio y peor de los casos con ejemplos:
- Complejidad en el tiempo y complejidad espacial.

En que consiste la búsqueda secuencial.

El algoritmo de búsqueda secuencial, también conocido como búsqueda lineal, es un método simple para buscar un elemento en una lista de manera secuencial. Consiste en recorrer cada elemento de la lista, uno por uno, comparándolo con el elemento buscado hasta encontrar una coincidencia o llegar al final de la lista.

El proceso de búsqueda secuencial se realiza siguiendo los siguientes pasos:

1. Se toma el primer elemento de la lista.
2. Se compara el elemento actual con el elemento buscado.

3. Si hay una coincidencia, se devuelve la posición o el elemento encontrado y se termina la búsqueda.
4. Si no hay coincidencia, se pasa al siguiente elemento de la lista y se repite el paso 2.
5. Si se llega al final de la lista sin encontrar el elemento buscado, se indica que el elemento no está presente en la lista.

El algoritmo de búsqueda secuencial es simple de implementar y no requiere que la lista esté ordenada. Sin embargo, puede ser ineficiente para listas grandes, ya que debe revisar todos los elementos en el peor de los casos.

Ventaja:

Ø Es un método sumamente simple que resulta útil cuando se tiene un conjunto de datos pequeños (Hasta aproximadamente 500 elementos)

Ø Es fácil adaptar la búsqueda secuencial para que utilice una lista enlazada ordenada, lo que hace la búsqueda más eficaz.

Ø Si los datos buscados no están en orden es el único método que puede emplearse para hacer dichas búsquedas.

Desventaja:

Ø Este método tiende hacer muy lento.

Ø Si los valores de la clave no son únicos, para encontrar todos los elementos con una clave particular, se requiere buscar en todo el arreglo, lo que hace el proceso muy largo.

Ejemplos de implementaciones del algoritmo en las estructuras de datos: memoria estática, memoria dinámica (listas) y archivos.

Estos son algunos ejemplos de algoritmos de búsqueda secuencial en estructura de datos: memoria estática, memoria dinámica(Listas) y archivos. Todos los algoritmos fueron hechos en clase con ayuda de la maestra.

Memoria estática.

```
public byte busSecuenciaOrdenada (int dato)

{

byte i=0;
```

```
while (i<ordenados.length && ordenados[i]<dato)

i++;

if (i<ordenados.length || ordenados[i]>dato)

return (byte) -i;

else

return i;

}
```

Memoria dinámica (listas).

```
public Nodo<T> busSecLista(T dato) {

Nodo i=inicio;

while(i.sig!=null && !i.info.equals(dato) )

i=i.sig;

return i;}

public Nodito busSecLista(T dato) {

Nodito b=inicio;

while(b!=null && !(dato.equals((Object)b.getInfo())))

b=b.getDer();

return (b);

}
```

Archivos.

```
public class BusquedaSecuencialEnArchivos {

public static int busquedaSecuencialEnArchivo(String nombreArchivo, String
elementoBuscado) {

int indice = 0;
```

```
try (BufferedReader br = new BufferedReader(new FileReader(nombreArchivo))) {
String linea;
while ((linea = br.readLine()) != null) {
indice++;
if (linea.equals(elementoBuscado)) {
return indice;
}}
} catch (IOException e) {
e.printStackTrace();
}
return -1;
}}
```

Análisis de eficiencia.

La búsqueda secuencial es un algoritmo simple utilizado para buscar un elemento en una lista o arreglo de manera secuencial. realizaremos un análisis de eficiencia de la búsqueda secuencial:

Mejor de los casos: El mejor caso ocurre cuando el elemento buscado se encuentra en la primera posición de la lista o arreglo. En este caso, la búsqueda se completa después de una sola comparación y se devuelve el resultado. Por lo tanto, el tiempo de ejecución en el mejor de los casos es O(1), es decir, constante.

Caso medio: En el caso medio, la eficiencia de la búsqueda secuencial depende de la distribución de los elementos en la lista o arreglo. En promedio, se espera que la búsqueda tenga que recorrer la mitad de la lista o arreglo antes de encontrar el elemento buscado. Por lo tanto, en promedio, el tiempo de ejecución en el caso medio es proporcional a n/2, donde n es la longitud de la lista o arreglo. Se puede aproximar como O(n), donde n es la longitud de la lista o arreglo.

Peor de los casos: El peor caso ocurre cuando el elemento buscado está en la última posición de la lista o arreglo, o no está presente en absoluto. En este caso, la búsqueda secuencial tendría que recorrer toda la lista o arreglo antes de determinar que el elemento no está presente. Por lo tanto, el tiempo de ejecución en el peor de los casos es proporcional a n, donde n es la longitud de la lista o

arreglo. Se puede representar como O(n), donde n es la longitud de la lista o arreglo.

En resumen, la eficiencia de la búsqueda secuencial es lineal, O(n), en el caso medio y el peor de los casos, donde n es la longitud de la lista o arreglo. En el mejor de los casos, la eficiencia es constante, O(1). Es importante tener en cuenta que la eficiencia de la búsqueda secuencial puede variar dependiendo de la distribución de los elementos y la cantidad de comparaciones necesarias para encontrar el elemento buscado.

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

De ejemplo tomaremos uno de los códigos anterior mente presentados y analizaremos todos los casos de ese código dando ejemplos del mismo.

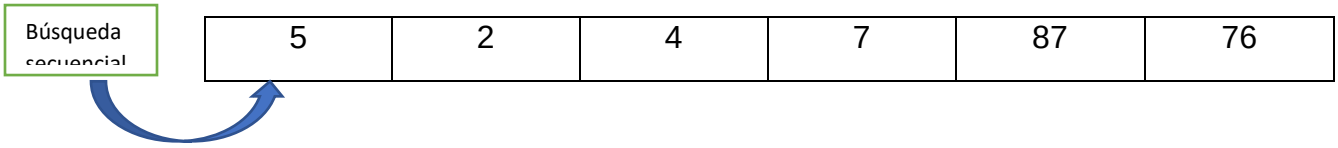
```
public byte busSecuenciaOrdenada (int dato)

{

byte i=0;

while (i<ordenados.length && ordenados[i]<dato)

i++;

if (i<ordenados.length || ordenados[i]>dato)

return (byte) -i;

else

return i;

}
```

1.-**Mejor de los casos:**

- En el mejor de los casos, el elemento dato se encuentra en la primera posición del arreglo ordenados.
- En este caso, el bucle while se ejecutará solo una vez, ya que ordenados[0] será igual a dato.
- Después de la primera iteración, se verificará la condición i < ordenados.length || ordenados[i] > dato, que será falsa, ya que i es igual a 0 y ordenados[i] es igual a dato.

- Por lo tanto, se devuelve i, que es 0.
- El tiempo de ejecución en el mejor de los casos es constante, O(1).

Ejemplo: tenemos que buscar un elemento en este caso el 5

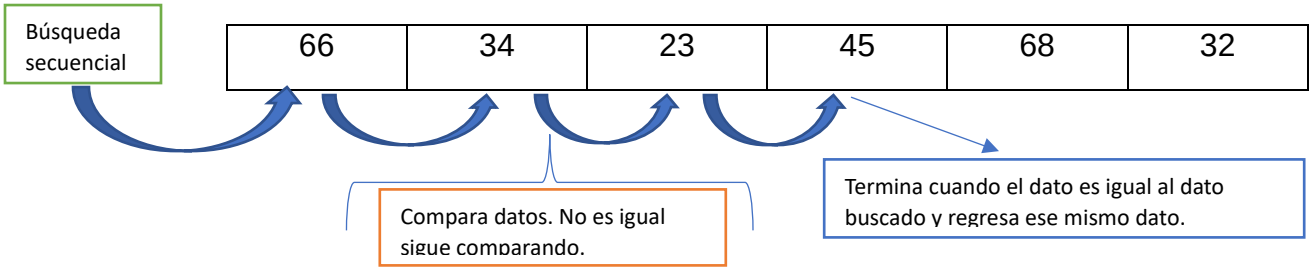


En el mejor de los casos la búsqueda no tendrá que hacer mayor recorrido ya que el dato a buscar está en la primera posición.

2.-**Caso medio:**

- En el caso medio, el elemento dato se encuentra en alguna posición intermedia del arreglo ordenados.
- En este caso, el bucle while se ejecutará un número de veces proporcional a la posición del elemento dato en el arreglo.
- Después de la iteración que encuentra el elemento dato, se realizará la verificación i < ordenados.length || ordenados[i] > dato, que será falsa.
- Por lo tanto, se devuelve i, que representa la posición del elemento dato en el arreglo.
- En el caso medio, el tiempo de ejecución es proporcional a la posición del elemento dato en el arreglo, y puede considerarse lineal, O(n), donde n es la longitud del arreglo ordenados.

Ejemplo: en esta ocasión el elemento a buscar es 45.



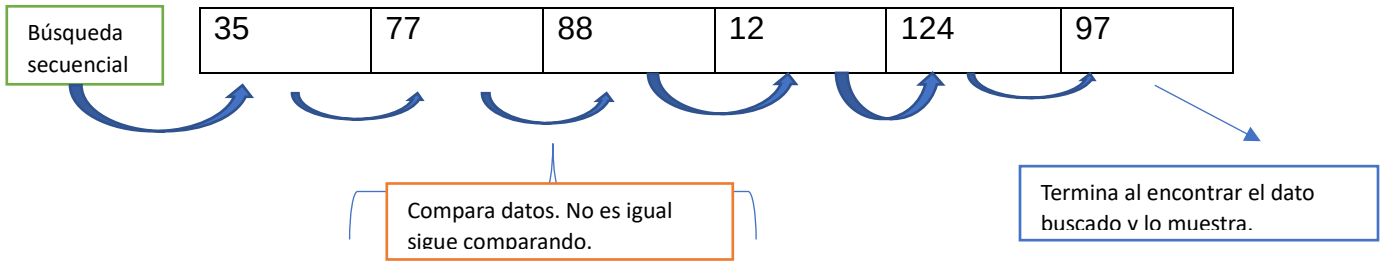
Como sabemos en este segundo caso la búsqueda será un poco mas larga ya que tendrá que hacer mas interacciones para llegar al elemento deseado.

3.-**Peor de los casos:**

- En el peor de los casos, el elemento dato no se encuentra en el arreglo ordenados o se encuentra en la última posición.
- En este caso, el bucle while recorrerá todo el arreglo ordenados hasta el final, ya que no se cumple la condición ordenados[i] < dato.

- Después de recorrer todo el arreglo, se realizará la verificación i < ordenados.length || ordenados[i] > dato, que será verdadera, ya que i será igual a ordenados.length.
- Por lo tanto, se devuelve -i, que es un valor negativo que indica que el elemento dato no se encontró en la secuencia ordenada.
- El tiempo de ejecución en el peor de los casos es lineal, O(n), donde n es la longitud del arreglo ordenados.

Ejemplo: en esta ocasión el elemento a buscar es 97.



En este caso pueden pasar dos cosas o el dato está en lo último o simplemente no se encuentra dentro en ambos casos el método tendrá que recorrer todo para encontrarlo haciendo mas largo las internaciones y el proceso.

Complejidad en el tiempo y complejidad espacial.
La búsqueda secuencial tiene una complejidad en tiempo y espacio que se puede analizar de la siguiente manera.

Complejidad en tiempo:

- En el peor de los casos, la búsqueda secuencial necesita recorrer todos los elementos de la lista o arreglo antes de encontrar el elemento buscado o determinar que no está presente. Por lo tanto, en el peor de los casos, la complejidad en tiempo es lineal, O(n), donde n es la longitud de la lista o arreglo.

- En el mejor de los casos, cuando el elemento buscado se encuentra en la primera posición, la búsqueda se completa en una sola comparación. En este caso, la complejidad en tiempo es constante, O(1).
- En el caso medio, la complejidad en tiempo es proporcional a la mitad de la longitud de la lista o arreglo, ya que se espera que la búsqueda tenga que recorrer en promedio la mitad de los elementos antes de encontrar el elemento buscado. Por lo tanto, en el caso medio, la complejidad en tiempo es O(n/2), que se puede aproximar como O(n), donde n es la longitud de la lista o arreglo.

Complejidad en espacio:

- La búsqueda secuencial no requiere espacio adicional más allá de la lista o arreglo original y algunas variables de control utilizadas durante el proceso de búsqueda. Por lo tanto, la complejidad en espacio de la búsqueda secuencial es constante, O(1), independientemente del tamaño de la lista o arreglo.

Ejemplo:

Des siguiente algoritmo.

```
public Nodo<T> busSecLista(T dato) {

Nodo i=inicio;

while(i.sig!=null && !i.info.equals(dato) )

i=i.sig;

return i;}
```

Su complejidad en tiempo será:

En el peor de los casos, la búsqueda secuencial necesita recorrer todos los nodos de la lista enlazada hasta llegar al último nodo o encontrar el nodo que contiene el elemento buscado. Esto implica realizar una iteración a través de los nodos de la lista hasta que se cumpla la condición i.sig != null && !i.info.equals(dato).

Si la lista enlazada tiene n nodos, la cantidad de iteraciones necesarias en el peor de los casos es n. Por lo tanto, la complejidad en tiempo en el peor de los casos es lineal, O(n), donde n es el tamaño de la lista enlazada.

Su complejidad en espacio será:

El algoritmo no requiere espacio adicional más allá de las variables locales utilizadas, como el nodo i. Por lo tanto, la complejidad en espacio es constante, O(1), independientemente del tamaño de la lista enlazada.

Algoritmos de búsqueda binaria

Sustituir el algoritmo de búsqueda secuencial ordenada, por el algoritmo de búsqueda binaria en el proyecto de datos ordenados (Segundo proyecto del tema 1).

Investigar

- En que consiste la búsqueda binaria
- Ejemplos de implementaciones del algoritmo en las estructuras de datos
- Análisis de eficiencia
- Análisis de los casos: mejor de los casos, caso medio y peor de los casos con ejemplos:
- Complejidad en el tiempo y complejidad espacial.

En que consiste la búsqueda binaria

La búsqueda binaria es el método, donde si el arreglo o vector esta bien ordenado, se reduce sucesivamente la operación eliminando repetidas veces la mitad de la lista restante.

El proceso comienza comparando el elemento central del arreglo con el elemento buscado. Si ambos coinciden finaliza la búsqueda. Si no ocurre así, el elemento buscado será mayor o menor en sentido estricto que el elemento central del arreglo. Si el elemento buscado es mayor se procede a hacer búsqueda binaria en el subarray superior, si el elemento buscado es menor que el contenido de la casilla central, se debe cambiar el segmento a considerar al segmento que está a la izquierda de tal sitio central.

Este método se puede aplicar tanto a datos en listas lineales como en árboles binarios de búsqueda.

La búsqueda binaria es un algoritmo de búsqueda eficiente utilizado para encontrar un elemento específico en una lista ordenada. La lista debe estar ordenada previamente en orden ascendente o descendente para que la búsqueda binaria funcione correctamente. El algoritmo sigue los siguientes pasos:

1. Comparación inicial: Se compara el elemento buscado con el elemento del medio de la lista. Si son iguales, la búsqueda se completa y se devuelve la posición del elemento.
2. División en mitades: Si el elemento buscado es menor que el elemento del medio, se descarta la mitad superior de la lista y se repite el proceso en la mitad inferior. Si el elemento buscado es mayor que el elemento del medio, se descarta la mitad inferior y se repite el proceso en la mitad superior.
3. Comparación y división continua: Se repite el proceso de comparación y división en mitades sucesivas hasta que se encuentre el elemento buscado o se determine que no está presente en la lista.

4. Búsqueda completada: Una vez que se encuentra el elemento buscado, se devuelve la posición en la lista. Si el elemento no está presente, se indica que la búsqueda ha sido infructuosa.

El proceso de búsqueda binaria reduce a la mitad el espacio de búsqueda en cada iteración, lo que la convierte en un algoritmo eficiente. La complejidad en tiempo de la búsqueda binaria es logarítmica, O(log n), donde n es el número de elementos en la lista. Esto contrasta con la búsqueda secuencial, que tiene una complejidad lineal, O(n), en el peor de los casos.

Ventajas:

Ø Se puede aplicar tanto a datos en listas lineales como en árboles binarios de búsqueda.

Ø Es el método más eficiente para encontrar elementos en un arreglo ordenado.

Desventajas:

Este método funciona solamente con arreglos ordenados, por lo cual si nos encontramos con arreglos que no están en orden, este método, no nos ayudaría en nada.

Ejemplos de implementaciones del algoritmo en las estructuras de datos.

Estos son algunos ejemplos de implementaciones del algoritmo de búsqueda binaria en diferentes estructuras de datos:

1.-**Búsqueda binaria en un arreglo:**

```
public static int busquedaBinaria(int[] arreglo, int dato) {

while (inicio <= fin) {

int medio = (inicio + fin) / 2;

if (arreglo[medio] == dato) {

return medio;

}

if (arreglo[medio] < dato) {

inicio = medio + 1;

} else {
```

```
fin = medio - 1;

}}

return -1; }
```

2.-**Búsqueda binaria en una lista enlazada ordenada:**

```
public static ListNode busquedaBinaria(ListNode head, int dato) {

while (inicio != fin) {

ListNode medio = getNodoMedio(inicio, fin);

if (medio.val == dato) {

return medio;

}

if (medio.val < dato) {

inicio = medio.next;

} else {

fin = medio;

}}

return null;

}
```

3.-**Búsqueda binaria en un árbol de búsqueda binaria:**

```
public static TreeNode busquedaBinaria(TreeNode root, int dato) {

if (root == null || root.val == dato) {

return root;

}

if (root.val < dato) {
```

```
return busquedaBinaria(root.right, dato);

} else {

return busquedaBinaria(root.left, dato);

}

}
```

Estos ejemplos ilustran cómo implementar la búsqueda binaria en diferentes estructuras de datos, como un arreglo, una lista enlazada ordenada y un árbol de búsqueda binaria. La idea central es dividir el conjunto de datos en mitades y comparar el valor buscado con el valor en la mitad para determinar la dirección de la búsqueda. Esto permite una búsqueda eficiente en estructuras de datos ordenadas.

Análisis de eficiencia

La eficiencia de la búsqueda binaria se puede analizar en términos de su complejidad en tiempo y espacio.

Complejidad en tiempo:

La búsqueda binaria tiene una complejidad en tiempo de O(log n), donde n es el tamaño del conjunto de datos sobre el cual se realiza la búsqueda. Esto se debe a que en cada iteración del algoritmo, el tamaño del rango de búsqueda se reduce aproximadamente a la mitad. Por lo tanto, el número de comparaciones necesarias para encontrar el elemento objetivo aumenta en escala logarítmica con respecto al tamaño del conjunto de datos. Esta propiedad hace que la búsqueda binaria sea muy eficiente en comparación con la búsqueda secuencial, especialmente cuando el conjunto de datos es grande.

Complejidad en espacio:

La búsqueda binaria tiene una complejidad en espacio de O(1), es decir, utiliza una cantidad constante de espacio adicional independientemente del tamaño del conjunto de datos. Esto se debe a que solo se utilizan variables adicionales para realizar los cálculos y no se requiere una estructura de datos auxiliar.

En resumen, la búsqueda binaria es altamente eficiente en términos de tiempo, ya que su complejidad es logarítmica en función del tamaño del conjunto de datos. Además, tiene una complejidad en espacio constante, lo que la hace aún más eficiente en términos de uso de memoria. Esto la convierte en una excelente

opción para buscar elementos en conjuntos de datos ordenados. Sin embargo, es importante tener en cuenta que la búsqueda binaria requiere que los datos estén ordenados de antemano para obtener resultados correctos.

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

Para este caso utilizaremos un código echo en clase, pero modificado para búsqueda binaria a continuación mostraremos el código:

```
public byte busquedaBinaria(int dato) {

while (inicio <= fin) {

int medio = (inicio + fin) / 2;

if (ordenados[medio] == dato) {

return (byte) medio;

}

if (ordenados[medio] < dato) {

inicio = medio + 1;

} else {

fin = medio - 1;

}}

return (byte) -inicio; }
```

1.-**Mejor de los casos:**

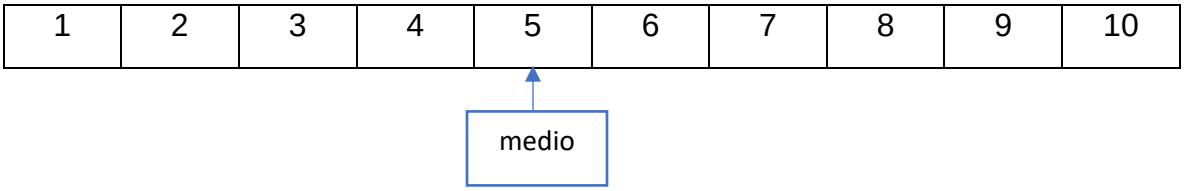
El mejor caso ocurre cuando el elemento buscado se encuentra exactamente en el medio del rango de búsqueda en cada iteración. En este caso, la búsqueda binaria encontrará el elemento deseado rápidamente y devolverá su posición en la primera iteración. La complejidad de tiempo en el mejor caso es O(1), ya que se encuentra el elemento de forma instantánea.

Ejemplo:

Supongamos que tenemos un arreglo ordenado de números del 1 al 10. Queremos buscar el número 5 utilizando el algoritmo de búsqueda binaria. A medida que avanzamos en las iteraciones, podemos visualizar cómo se reduce el rango de búsqueda hasta encontrar el número deseado:

1.-En la primera iteración, el rango de búsqueda abarca todo el arreglo: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10].

- El índice de inicio es 0 y el índice de fin es 9.
- Calculamos el índice del elemento medio: (0 + 9) / 2 = 4.
- El elemento en la posición media es 5, que es igual al número buscado. La búsqueda se detiene y se devuelve la posición 4.



En este ejemplo, el número buscado se encuentra exactamente en el medio del rango de búsqueda. Como resultado, la búsqueda binaria encuentra el elemento deseado en la primera iteración. La complejidad de tiempo en este mejor caso es O(1), ya que se encuentra el elemento de forma instantánea

2.-**Caso medio:**

En el caso medio, se asume que no hay patrones particulares en la distribución de los elementos en el arreglo. La búsqueda binaria sigue dividiendo el rango de búsqueda por la mitad en cada iteración, lo que reduce significativamente la cantidad de elementos a revisar. En promedio, el número de iteraciones necesarias para encontrar el elemento objetivo es logarítmico en función del tamaño del rango de búsqueda. Por lo tanto, la complejidad de tiempo en el caso medio es O(log n), donde n es la diferencia entre el índice de inicio y el índice de fin.

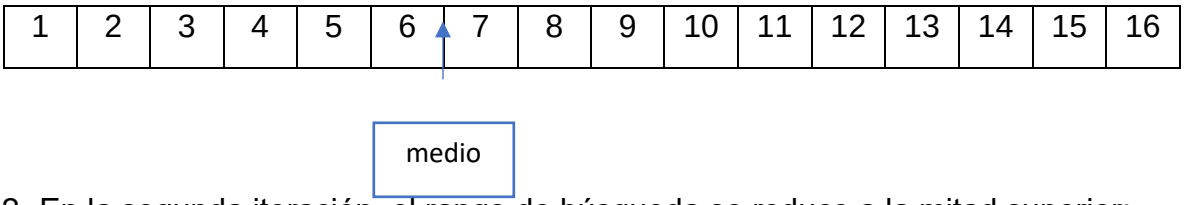
Ejemplo:

Supongamos que tenemos un arreglo ordenado de números del 1 al 16. Queremos buscar el número 11 utilizando el algoritmo de búsqueda binaria. A medida que avanzamos en las iteraciones, podemos visualizar cómo se reduce el rango de búsqueda hasta encontrar el número deseado:

1.-En la primera iteración, el rango de búsqueda abarca todo el arreglo: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16].

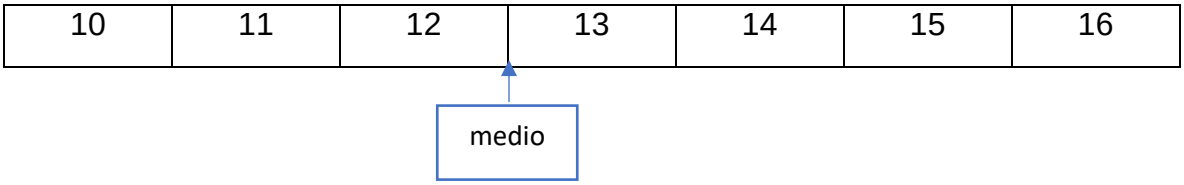
- El índice de inicio es 0 y el índice de fin es 15.
- Calculamos el índice del elemento medio: (0 + 15) / 2 = 7.

- El elemento en la posición media es 8, que es menor que el número buscado (11). Sabemos que el número buscado está en la mitad superior del rango.



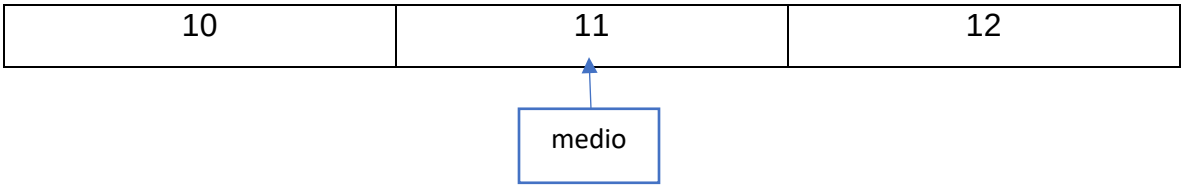
2.-En la segunda iteración, el rango de búsqueda se reduce a la mitad superior: [10, 11, 12, 13, 14, 15, 16].

- El índice de inicio es 8 y el índice de fin es 15.
- Calculamos el índice del elemento medio: (8 + 15) / 2 = 11.
- El elemento en la posición media es 13, que es mayor que el número buscado (11). Sabemos que el número buscado está en la mitad inferior del rango.



3.-En la tercera iteración, el rango de búsqueda se reduce aún más: [10, 11, 12].

- El índice de inicio es 8 y el índice de fin es 10.
- Calculamos el índice del elemento medio: (8 + 10) / 2 = 9.
- El elemento en la posición media es 11, que es igual al número buscado. La búsqueda se detiene y se devuelve la posición 9.



3.- **Peor de los casos:**

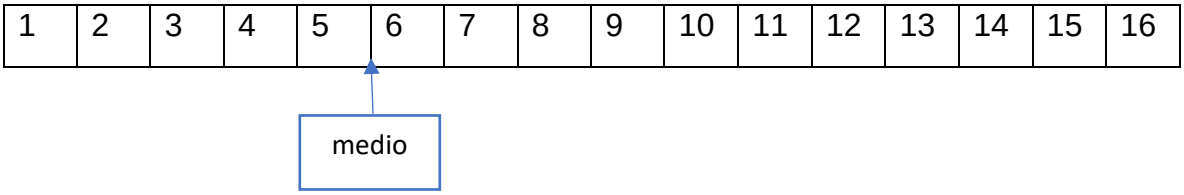
El peor caso ocurre cuando el elemento buscado no está presente en el arreglo o está en el extremo del rango de búsqueda. En este caso, la búsqueda binaria seguirá dividiendo el rango de búsqueda hasta que el rango se reduzca a cero. Esto implica que se realizarán log n iteraciones en el peor caso, donde n es la diferencia entre el índice de inicio y el índice de fin. La complejidad de tiempo en el peor caso también es O(log n).

Ejemplo:

Supongamos que tenemos un arreglo ordenado de números del 1 al 16. Queremos buscar el número 20 utilizando el algoritmo de búsqueda binaria. A medida que avanzamos en las iteraciones, podemos visualizar cómo se reduce el rango de búsqueda hasta que se alcance el peor caso:

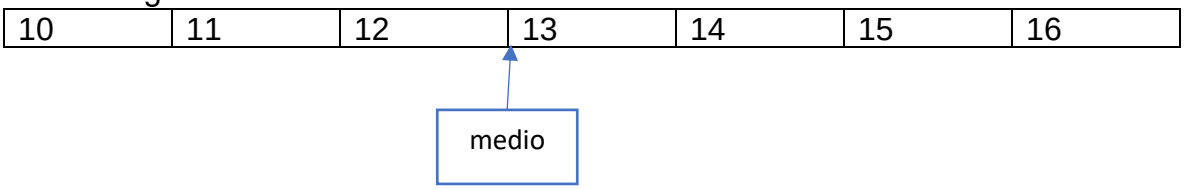
1.-En la primera iteración, el rango de búsqueda abarca todo el arreglo: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16].

- El índice de inicio es 0 y el índice de fin es 15.
- Calculamos el índice del elemento medio: (0 + 15) / 2 = 7.
- El elemento en la posición media es 8, que es menor que el número buscado (20). Sabemos que el número buscado está en la mitad superior del rango.



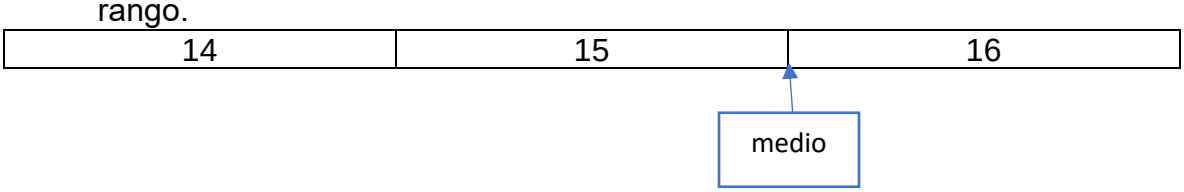
2.-En la segunda iteración, el rango de búsqueda se reduce a la mitad superior: [10, 11, 12, 13, 14, 15, 16].

- El índice de inicio es 8 y el índice de fin es 15.
- Calculamos el índice del elemento medio: (8 + 15) / 2 = 11.
- El elemento en la posición media es 13, que es menor que el número buscado (20). Sabemos que el número buscado está en la mitad superior del rango.



3.-En la tercera iteración, el rango de búsqueda se reduce aún más: [14, 15, 16].

- El índice de inicio es 13 y el índice de fin es 15.
- Calculamos el índice del elemento medio: (13 + 15) / 2 = 14.
- El elemento en la posición media es 15, que es menor que el número buscado (20). Sabemos que el número buscado está en la mitad superior del rango.



4.-En la cuarta iteración, el rango de búsqueda se reduce aún más: [16].

- El índice de inicio es 15 y el índice de fin es 15.
- Calculamos el índice del elemento medio: (15 + 15) / 2 = 15.
- El elemento en la posición media es 16, que es menor que el número buscado (20). Sabemos que el número buscado está en la mitad superior del rango.

16

En este ejemplo, la búsqueda binaria no encuentra el número buscado.

Complejidad en el tiempo y complejidad espacial.

La complejidad en el tiempo de la búsqueda binaria es O(log n), donde n es el tamaño del conjunto de datos en el que se realiza la búsqueda. Esto significa que el tiempo de ejecución del algoritmo crece de forma logarítmica a medida que aumenta el tamaño del conjunto de datos. En cada iteración, la búsqueda binaria divide el rango de búsqueda por la mitad, reduciendo eficientemente la cantidad de elementos a examinar.

La complejidad logarítmica de la búsqueda binaria se debe a su enfoque de "divide y conquista". En cada iteración, se reduce el rango de búsqueda a la mitad, lo que permite descartar una gran cantidad de elementos en comparación con otros algoritmos de búsqueda. Esto hace que la búsqueda binaria sea muy eficiente para conjuntos de datos grandes, ya que incluso para conjuntos muy grandes, el número de operaciones requeridas para encontrar el elemento objetivo sigue siendo relativamente bajo.

En cuanto a la complejidad espacial, la búsqueda binaria requiere un espacio constante, ya que solo se necesitan variables adicionales para realizar el seguimiento de los índices de inicio y fin del rango de búsqueda, así como una variable para almacenar el valor del elemento medio. No se requiere espacio adicional proporcional al tamaño del conjunto de datos. Por lo tanto, la complejidad espacial de la búsqueda binaria es O(1), es decir, espacio constante.

En resumen:

Complejidad en el tiempo: O(log n)

Complejidad espacial: O(1) (espacio constante)

La búsqueda binaria es uno de los algoritmos de búsqueda más eficientes para conjuntos de datos ordenados, y su complejidad logarítmica hace que sea una opción preferida en situaciones en las que se necesita una búsqueda rápida en grandes conjuntos de datos.

Búsqueda de patrones de Knuth Morris Prat

- **En que consiste:**

La búsqueda de patrones de Knuth-Morris-Pratt (KMP) es un algoritmo eficiente para buscar ocurrencias de un patrón dado en una cadena de texto. Fue desarrollado por Donald Knuth, James H. Morris y Vaughan Pratt en 1977.

El algoritmo KMP utiliza una estrategia de "salto" para evitar la repetición de comparaciones innecesarias durante la búsqueda. La idea clave detrás del algoritmo es que cuando ocurre una falta de coincidencia entre el patrón y el texto en una posición determinada, se puede aprovechar la información ya comparada para determinar los próximos índices en los que se debe iniciar la comparación.

El proceso de construcción del algoritmo KMP implica la creación de una tabla de "fallas" o "límites" que almacena información sobre los prefijos que coinciden con los sufijos de un patrón dado. Esta tabla de fallas se utiliza para calcular los desplazamientos necesarios en caso de que ocurra una falta de coincidencia.

El algoritmo KMP tiene una complejidad de tiempo de O(n + m), donde n es la longitud del texto y m es la longitud del patrón. Esto lo hace más eficiente en comparación con otros algoritmos de búsqueda de patrones, como el algoritmo de fuerza bruta, que tiene una complejidad de tiempo de O(n * m).

En resumen, la búsqueda de patrones de Knuth-Morris-Pratt es un algoritmo eficiente para encontrar ocurrencias de un patrón en una cadena de texto. Utiliza una estrategia de "salto" y una tabla de fallas para evitar comparaciones innecesarias y lograr un mejor rendimiento en comparación con otros algoritmos de búsqueda de patrones.

- **Ejemplo de Implementación:**

```
import java.util.ArrayList;
import java.util.List;

class KnuthMorrisPratt {
    public static int[] calcularTablaFallas(String patron) {
        int m = patron.length();
        int[] tabla = new int[m];
        int i = 1, j = 0;

        while (i < m) {
            if (patron.charAt(i) == patron.charAt(j)) {
                j++;
                tabla[i] = j;
                i++;
            } else {
                if (j != 0) {
                    j = tabla[j - 1];
                } else {
                    tabla[i] = 0;
                    i++;
                }
            }
        }

        return tabla;
    }

    public static List<Integer> buscarPatron(String texto, String patron) {
```

```
    int n = texto.length();
    int m = patron.length();
    int[] tabla = calcularTablaFallas(patron);
    List<Integer> resultados = new ArrayList<>();

    int i = 0, j = 0;
    while (i < n) {
        if (texto.charAt(i) == patron.charAt(j)) {
            i++;
            j++;

            if (j == m) {
                resultados.add(i - j);
                j = tabla[j - 1];
            }
        } else {
            if (j != 0) {
                j = tabla[j - 1];
            } else {
                i++;
            }
        }
    }

    return resultados;
}

public static void main(String[] args) {
```

```
        String texto = "ABCABCDABABCDABCDABDE";
        String patron = "ABCDABD";
        List<Integer> resultados = buscarPatron(texto, patron);

        System.out.print("Ocurrencias encontradas en las posiciones: ");
        for (int resultado : resultados) {
            System.out.print(resultado + " ");
        }
    }
}
```

Salida:

```
Ocurrencias encontradas en las posiciones: 10
```

- **Análisis de eficiencia**

La búsqueda de patrones de Knuth-Morris-Pratt (KMP) es conocida por su eficiencia en comparación con otros algoritmos de búsqueda de patrones, como el algoritmo de fuerza bruta. Su complejidad de tiempo es de O(n + m), donde n es la longitud del texto y m es la longitud del patrón.

En el proceso de construcción de la tabla de fallas, que es la etapa previa a la búsqueda, se realizan comparaciones entre caracteres del patrón. Esto se hace una sola vez y tiene una complejidad de tiempo de O(m), donde m es la longitud del patrón.

Durante la etapa de búsqueda, se realiza una comparación entre los caracteres del texto y el patrón. En el peor caso, el algoritmo no realizará comparaciones adicionales y avanzará sin retroceder en el patrón. Esto se debe a que la tabla de fallas proporciona información precisa sobre los desplazamientos necesarios cuando se produce una falta de coincidencia.

Por lo tanto, en el peor caso, el algoritmo KMP realizará un total de n comparaciones entre caracteres del texto y el patrón. Esto da lugar a una complejidad de tiempo de O(n).

En resumen, la eficiencia del algoritmo KMP radica en su capacidad para evitar comparaciones innecesarias al utilizar la información almacenada en la tabla de fallas. Esto permite que el algoritmo tenga una complejidad de tiempo lineal O(n +

m), lo cual es muy favorable en comparación con otros algoritmos que pueden tener una complejidad cuadrática en el peor caso. Esto hace que el algoritmo KMP sea especialmente útil cuando se busca un patrón en un texto grande o cuando se necesita realizar múltiples búsquedas de patrones en diferentes textos.

- **Análisis de los casos:**

Peor de los casos

El peor caso de la búsqueda de patrones de Knuth-Morris-Pratt (KMP) ocurre cuando el patrón no se encuentra en el texto o se encuentra al final del texto. En este caso, el algoritmo debe realizar el máximo número de comparaciones posible.

La complejidad de tiempo en el peor caso para el algoritmo KMP es de O(n + m), donde n es la longitud del texto y m es la longitud del patrón.

En el peor caso, el algoritmo KMP realizará n comparaciones entre caracteres del texto y el patrón. Esto se debe a que cada posición en el texto debe ser visitada al menos una vez.

La etapa de construcción de la tabla de fallas tiene una complejidad de tiempo de O(m), ya que se realizan comparaciones entre caracteres del patrón para determinar los valores de la tabla.

Sin embargo, en el peor caso, el algoritmo KMP no necesita retroceder en el texto ni realizar comparaciones adicionales debido a la información almacenada en la tabla de fallas. Por lo tanto, el número total de comparaciones realizadas es proporcional a la longitud del texto, es decir, O(n).

En términos de complejidad espacial, el algoritmo KMP requiere O(m) de espacio para almacenar la tabla de fallas.

En resumen, en el peor caso, el algoritmo KMP tiene una complejidad de tiempo de O(n + m) y una complejidad espacial de O(m). Aunque el peor caso se produce cuando el patrón no se encuentra en el texto o se encuentra al final del texto, el algoritmo KMP sigue siendo eficiente en comparación con otros algoritmos de búsqueda de patrones, como el algoritmo de fuerza bruta, que tiene una complejidad cuadrática en el peor caso.

A continuación, se muestra un ejemplo del peor caso de KMP:

Texto: "ABCDEF" Patrón: "XYZ"

En este caso, el patrón "XYZ" no se encuentra en el texto "ABCDEF". El algoritmo KMP realizará comparaciones entre caracteres hasta recorrer todo el texto sin encontrar ninguna coincidencia.

En el peor caso, el algoritmo KMP realizará un número de comparaciones igual a la longitud del texto. Esto se debe a que cada posición en el texto debe ser visitada al menos una vez antes de determinar que no hay coincidencias.

Aunque el peor caso es menos común en la práctica, es importante tener en cuenta que el algoritmo KMP sigue siendo eficiente en comparación con otros algoritmos de búsqueda de patrones, ya que su complejidad de tiempo en el peor caso es de O(n + m), donde n es la longitud del texto y m es la longitud del patrón. El uso de la tabla de fallas permite al algoritmo evitar comparaciones innecesarias y realizar búsquedas eficientes en la mayoría de los casos.

Caso medio

El caso medio de la búsqueda de patrones de Knuth-Morris-Pratt (KMP) ocurre cuando el patrón y el texto tienen características aleatorias, sin patrones repetitivos o estructuras específicas.

La complejidad de tiempo en el caso medio para el algoritmo KMP es de O(n + m), donde n es la longitud del texto y m es la longitud del patrón.

En el caso medio, el algoritmo KMP sigue aprovechando la información almacenada en la tabla de fallas para evitar comparaciones innecesarias. A medida que se realiza la búsqueda, el algoritmo puede saltar posiciones en el texto y realizar menos comparaciones que en el peor caso.

La etapa de construcción de la tabla de fallas sigue teniendo una complejidad de tiempo de O(m), ya que se realizan comparaciones entre caracteres del patrón para determinar los valores de la tabla.

En el caso medio, el número de comparaciones realizadas por el algoritmo KMP es menor que en el peor caso, pero puede variar dependiendo de la estructura y distribución de los caracteres en el texto y el patrón.

En términos de complejidad espacial, el algoritmo KMP requiere O(m) de espacio para almacenar la tabla de fallas.

En resumen, en el caso medio, el algoritmo KMP tiene una complejidad de tiempo de O(n + m) y una complejidad espacial de O(m). Aunque el número de comparaciones realizadas puede variar según las características específicas del texto y el patrón, el algoritmo KMP sigue siendo eficiente en comparación con otros algoritmos de búsqueda de patrones, especialmente en casos donde el patrón no se encuentra al final del texto o cuando el texto y el patrón tienen una distribución aleatoria.

A continuación, se muestra un ejemplo del caso medio de KMP:

Texto: "ABACABABCD" Patrón: "ABCD"

En este caso, el patrón "ABCD" se encuentra en el texto "ABACABABCD", pero no al principio ni al final. El algoritmo KMP utilizará la tabla de fallas para realizar comparaciones eficientes y encontrar la posición de la coincidencia.

El algoritmo KMP realizará un número de comparaciones proporcional a la longitud del texto y el patrón, pero evitará comparaciones innecesarias utilizando la información almacenada en la tabla de fallas. En este ejemplo, el algoritmo realizará comparaciones entre caracteres hasta encontrar la coincidencia exacta.

El caso medio es representativo de situaciones en las que el patrón y el texto no tienen patrones repetitivos o estructuras específicas. En estos casos, el algoritmo KMP sigue siendo eficiente debido a su capacidad para evitar comparaciones innecesarias mediante el uso de la tabla de fallas. La complejidad de tiempo en el caso medio para el algoritmo KMP sigue siendo de O(n + m), donde n es la longitud del texto y m es la longitud del patrón.

Mejor de los casos

El mejor caso de la búsqueda de patrones de Knuth-Morris-Pratt (KMP) ocurre cuando se encuentra una coincidencia exacta del patrón al principio del texto. En este caso, el algoritmo puede identificar la coincidencia en una sola iteración sin necesidad de realizar comparaciones adicionales.

La complejidad de tiempo en el mejor caso para el algoritmo KMP es de O(m), donde m es la longitud del patrón. Esto se debe a que en el mejor caso, se encuentra una coincidencia exacta al principio del texto y no es necesario realizar ninguna comparación adicional en el resto del texto.

La etapa de construcción de la tabla de fallas sigue teniendo una complejidad de tiempo de O(m), ya que se realizan comparaciones entre caracteres del patrón para determinar los valores de la tabla.

En el mejor caso, el algoritmo KMP realiza un número mínimo de comparaciones y puede identificar la posición inicial de la coincidencia de manera eficiente.

En términos de complejidad espacial, el algoritmo KMP requiere O(m) de espacio para almacenar la tabla de fallas.

Es importante tener en cuenta que el mejor caso se produce en situaciones específicas donde hay una coincidencia exacta al principio del texto. En la mayoría de los casos prácticos, es más común enfrentarse a casos medios o peores, donde la eficiencia del algoritmo KMP brilla en comparación con otros algoritmos de búsqueda de patrones.

A continuación, se muestra un ejemplo del mejor caso de KMP:

Texto: "ABCDEF" Patrón: "ABC"

En este caso, el patrón "ABC" se encuentra al principio del texto "ABCDEF". El algoritmo KMP identificará la coincidencia en la primera iteración y no realizará ninguna comparación adicional.

El mejor caso ocurre cuando el patrón se encuentra al principio del texto y coincide exactamente. En este caso, la complejidad de tiempo del algoritmo KMP es de O(m), donde m es la longitud del patrón. No se requieren comparaciones adicionales en el resto del texto, lo que resulta en una búsqueda muy eficiente.

- **Complejidad en el tiempo y complejidad espacial.**

La complejidad de tiempo de la búsqueda de patrones de Knuth-Morris-Pratt (KMP) es de O(n + m), donde n es la longitud del texto y m es la longitud del patrón.

La etapa de construcción de la tabla de fallas tiene una complejidad de tiempo de O(m), ya que se realizan comparaciones entre caracteres del patrón para determinar los valores de la tabla.

En la etapa de búsqueda, se realiza un bucle que recorre el texto una sola vez, realizando comparaciones entre caracteres del texto y el patrón. En cada iteración, el algoritmo realiza una cantidad constante de operaciones (comparaciones y posibles actualizaciones de variables). En el peor caso, si no se encuentra ninguna coincidencia, se realizarán n comparaciones entre caracteres.

Por lo tanto, la complejidad de tiempo total del algoritmo KMP es de O(n + m).

Es importante destacar que la complejidad de tiempo lineal de O(n + m) del algoritmo KMP lo hace eficiente en comparación con otros algoritmos de búsqueda de patrones. En particular, el algoritmo KMP es beneficioso cuando se busca un patrón en un texto largo o cuando se necesita realizar múltiples búsquedas de patrones en diferentes textos.
La complejidad espacial de la búsqueda de patrones de Knuth-Morris-Pratt (KMP) es de O(m), donde m es la longitud del patrón.

La principal estructura de datos utilizada en el algoritmo KMP es la tabla de fallas, que se utiliza para almacenar información sobre los desplazamientos en caso de una falta de coincidencia. La tabla de fallas tiene una longitud igual a la longitud del patrón y requiere O(m) de espacio.

Además de la tabla de fallas, el algoritmo KMP utiliza variables adicionales para mantener el seguimiento de las posiciones en el texto y el patrón durante la etapa de búsqueda. Estas variables requieren un espacio constante, por lo que no afectan la complejidad espacial.

En resumen, la complejidad espacial del algoritmo KMP es O(m), donde m es la longitud del patrón. Esto significa que el espacio requerido por el algoritmo aumenta linealmente con respecto a la longitud del patrón.

Método Saltar búsqueda

- **En que consiste la búsqueda Saltar búsqueda**

El método de salto de búsqueda, también conocido como búsqueda por salto o búsqueda por interpolación, es una técnica utilizada en estructuras de datos para buscar eficientemente un elemento en una secuencia ordenada, como un arreglo o una lista.

A diferencia de la búsqueda binaria que divide la secuencia en dos partes iguales, el método de salto de búsqueda calcula una estimación más precisa de la posición del elemento buscado basándose en la distribución de los valores en la secuencia.

El algoritmo de salto de búsqueda sigue los siguientes pasos:

1. Estima la posición del elemento buscado en función de los valores mínimos y máximos de la secuencia y del valor buscado.
2. Compara el elemento en la posición estimada con el valor buscado.
    - Si son iguales, se ha encontrado el elemento buscado y se devuelve su posición.
    - Si el elemento en la posición estimada es mayor que el valor buscado, la búsqueda se realizará en la parte anterior de la secuencia.
    - Si el elemento en la posición estimada es menor que el valor buscado, la búsqueda se realizará en la parte posterior de la secuencia.
3. Divide la secuencia en una fracción más pequeña y repite los pasos 1 y 2 en la parte seleccionada.
4. El proceso se repite hasta que se encuentre el elemento buscado o hasta que no queden elementos por revisar en la secuencia.

El método de salto de búsqueda es especialmente eficiente en secuencias con una distribución uniforme de los valores, ya que puede realizar estimaciones más precisas. Sin embargo, si la distribución no es uniforme, puede comportarse similar a una búsqueda binaria.

Es importante destacar que al igual que la búsqueda binaria, el método de salto de búsqueda solo se puede aplicar en secuencias ordenadas. Si la secuencia no está

ordenada, es necesario realizar una ordenación previa o utilizar otro algoritmo de búsqueda, como la búsqueda secuencial.

- **Ejemplos de implementaciones del algoritmo en las estructuras de datos:**

```
public class SaltoBusqueda {

    public static int saltoBusqueda(int[] arr, int elemento) {

        int n = arr.length;

        int salto = (int) Math.sqrt(n); // Calcula el tamaño del salto inicial


        // Encuentra el bloque donde puede estar el elemento

        int previo = 0;

        while (arr[Math.min(salto, n) - 1] < elemento) {

            previo = salto;

            salto += (int) Math.sqrt(n);

            if (previo >= n) { // Si el bloque actual es el último

                return -1; // Elemento no encontrado

            }

        }


        // Realiza una búsqueda lineal en el bloque encontrado

        while (arr[previo] < elemento) {

            previo++;

            if (previo == Math.min(salto, n)) {

                return -1; // Elemento no encontrado
```

```
            }
        }
        if (arr[previo] == elemento) {
            return previo; // Elemento encontrado
        }
        return -1; // Elemento no encontrado
    }

    public static void main(String[] args) {
        int[] arreglo = {2, 5, 8, 12, 16, 23, 38, 56, 72, 91};
        int elementoBuscado = 16;
        int indice = saltoBusqueda(arreglo, elementoBuscado);
        if (indice != -1) {
            System.out.println("Elemento encontrado en el índice: " + indice);
        } else {
            System.out.println("Elemento no encontrado");
        }
    }
}
```

- **Análisis de eficiencia**

La eficiencia del método de salto de búsqueda en estructuras de datos depende de la distribución de los datos y del tamaño de la estructura de datos. A continuación, se presentan las consideraciones de eficiencia para este método:

Tiempo de ejecución promedio: El tiempo de ejecución promedio del método de salto de búsqueda es O(sqrt(n)), donde n es el tamaño de la estructura de datos. Esto se debe a que en cada iteración, el tamaño del salto aumenta en la raíz cuadrada del tamaño de la estructura de datos. La búsqueda lineal dentro de cada bloque requiere un número constante de comparaciones. En general, el método de salto de búsqueda es más rápido que una búsqueda secuencial lineal, pero no es tan eficiente como la búsqueda binaria, que tiene un tiempo de ejecución de O(log(n)).

Espacio adicional requerido: El método de salto de búsqueda no requiere espacio adicional más allá del arreglo o estructura de datos existente en la que se realiza la búsqueda. No se crean estructuras de datos adicionales durante el proceso de búsqueda.

Ordenamiento previo: El método de salto de búsqueda asume que los datos están ordenados. Si los datos no están ordenados, se requerirá un paso adicional para ordenarlos, lo cual puede tener un tiempo de ejecución adicional dependiendo del algoritmo de ordenamiento utilizado. Sin embargo, una vez que los datos están ordenados, el método de salto de búsqueda ofrece una búsqueda eficiente.

Distribución de datos: La eficiencia del método de salto de búsqueda puede verse afectada por la distribución de los datos en la estructura de datos. Si los datos están uniformemente distribuidos, el método de salto de búsqueda puede proporcionar un rendimiento óptimo. Sin embargo, si los datos están agrupados o tienen una distribución no uniforme, el método de salto de búsqueda puede comportarse de manera similar a una búsqueda secuencial lineal, lo que resulta en un peor rendimiento.

En resumen, el método de salto de búsqueda ofrece una búsqueda eficiente en estructuras de datos ordenadas, con un tiempo de ejecución promedio de O(sqrt(n)). Sin embargo, su rendimiento puede verse afectado por la distribución de los datos en la estructura de datos. Es recomendable considerar la distribución de los datos y evaluar otras técnicas de búsqueda, como la búsqueda binaria, dependiendo de los requisitos específicos del problema.

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

Mejor de los casos
En el mejor de los casos, el método de salto de búsqueda puede tener un tiempo de ejecución muy eficiente. Esto ocurre cuando el elemento buscado se encuentra exactamente en la primera comparación, es decir, en el elemento inicial o en el bloque inicial estimado.

En el mejor de los casos, el método de salto de búsqueda tiene un tiempo de ejecución de O(1), ya que encuentra el elemento de inmediato con una única comparación. Esto se debe a que el elemento buscado se encuentra en el primer elemento o en el bloque estimado en la primera iteración del algoritmo.

Por ejemplo, si tenemos un arreglo ordenado con elementos [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] y buscamos el elemento 1, el método de salto de búsqueda lo encontraría de inmediato en la primera comparación, ya que el primer elemento es el elemento buscado.

Sin embargo, es importante tener en cuenta que este escenario ideal del mejor caso puede ser poco frecuente en la práctica. En la mayoría de los casos, se asume una distribución más aleatoria de los datos y el método de salto de búsqueda tendrá un rendimiento promedio de O(sqrt(n)), como se mencionó anteriormente.

Por lo tanto, al analizar la eficiencia del método de salto de búsqueda, es más común considerar su rendimiento promedio y peor de los casos, ya que ofrecen una visión más completa de su desempeño en diferentes escenarios.

Ejemplo: Supongamos que tenemos un arreglo ordenado de números del 1 al 100 y queremos buscar el número 50.

```
public class SaltoBusqueda {

    public static int saltoBusqueda(int[] arr, int elemento) {

        int n = arr.length;

        int salto = (int) Math.sqrt(n); // Calcula el tamaño del salto inicial



        // Encuentra el bloque donde puede estar el elemento

        int previo = 0;

        while (arr[Math.min(salto, n) - 1] < elemento) {

            previo = salto;

            salto += (int) Math.sqrt(n);

            if (previo >= n) { // Si el bloque actual es el último

                return -1; // Elemento no encontrado
```

```
        }
    }

    // Realiza una búsqueda lineal en el bloque encontrado
    while (arr[previo] < elemento) {
        previo++;
        if (previo == Math.min(salto, n)) {
            return -1; // Elemento no encontrado
        }
    }
    if (arr[previo] == elemento) {
        return previo; // Elemento encontrado
    }
    return -1; // Elemento no encontrado
}

public static void main(String[] args) {
    int[] arreglo = new int[100];
    for (int i = 0; i < 100; i++) {
        arreglo[i] = i + 1;
    }

    int elementoBuscado = 50;
```

```
        int indice = saltoBusqueda(arreglo, elementoBuscado);


        if (indice != -1) {
            System.out.println("Elemento encontrado en el índice: " + indice);
        } else {
            System.out.println("Elemento no encontrado");
        }
    }
}
```

Caso Medio

El caso medio del método de salto de búsqueda es más difícil de determinar con precisión, ya que depende de la distribución de los datos y la ubicación del elemento buscado dentro de la estructura de datos. Sin embargo, en general, se puede esperar un rendimiento promedio mejor que una búsqueda secuencial lineal, pero no tan eficiente como una búsqueda binaria.

El método de salto de búsqueda divide la estructura de datos en bloques y realiza una búsqueda lineal en el bloque donde se estima que se encuentra el elemento buscado. El tamaño del salto inicial se calcula como la raíz cuadrada del tamaño total de la estructura de datos. Esto implica que el método de salto de búsqueda puede saltar a través de bloques más grandes en comparación con una búsqueda secuencial lineal, lo que reduce el número de comparaciones necesarias.

En el caso medio, el tiempo de ejecución del método de salto de búsqueda se acerca a O(sqrt(n)), donde n es el tamaño de la estructura de datos. Sin embargo, este rendimiento puede variar dependiendo de la distribución de los datos y la ubicación del elemento buscado. Si los datos están uniformemente distribuidos, se puede esperar un buen rendimiento promedio. Sin embargo, si los datos están agrupados o tienen una distribución no uniforme, el método de salto de búsqueda puede comportarse de manera similar a una búsqueda secuencial lineal en el peor de los casos.

En resumen, el caso medio del método de salto de búsqueda se caracteriza por un rendimiento promedio mejor que una búsqueda secuencial lineal, pero no tan eficiente como una búsqueda binaria. La eficiencia en el caso medio depende de

la distribución de los datos y la ubicación del elemento buscado dentro de la estructura de datos.

Ejemplo: Supongamos que tenemos un arreglo ordenado de números del 1 al 100 y queremos buscar el número 75.

```
def salto_busqueda(arr, elemento):
    n = len(arr)
    salto = int(n**0.5)  # Calcula el tamaño del salto inicial

    # Encuentra el bloque donde puede estar el elemento
    previo = 0
    while arr[min(salto, n) - 1] < elemento:
        previo = salto
        salto += int(n**0.5)
        if previo >= n:  # Si el bloque actual es el último
            return -1  # Elemento no encontrado

    # Realiza una búsqueda lineal en el bloque encontrado
    while arr[previo] < elemento:
        previo += 1
        if previo == min(salto, n):
            return -1  # Elemento no encontrado
    if arr[previo] == elemento:
        return previo  # Elemento encontrado
    return -1  # Elemento no encontrado
```

```
# Arreglo ordenado del 1 al 100

arreglo = [i for i in range(1, 101)]


# Elemento a buscar: 75

elemento_buscar = 75


# Llamada al método de salto de búsqueda

indice = salto_busqueda(arreglo, elemento_buscar)


if indice != -1:

    print("Elemento encontrado en el índice:", indice)

else:

    print("Elemento no encontrado")
```

Peor de los casos

El peor de los casos en el método de salto de búsqueda ocurre cuando el elemento buscado no está presente en la estructura de datos y se necesita recorrer todos los bloques hasta llegar al último bloque. En este caso, la complejidad en el tiempo del método de salto de búsqueda se acerca a O(n), donde n es el tamaño de la estructura de datos.

En el peor de los casos, el método de salto de búsqueda requerirá saltar a través de bloques y realizar una búsqueda lineal en cada bloque hasta encontrar el final de la estructura de datos sin encontrar el elemento buscado. Esto implica realizar un número significativo de comparaciones, ya que se deben explorar todos los elementos de la estructura de datos.

Aunque el método de salto de búsqueda reduce el número de comparaciones en comparación con una búsqueda secuencial lineal, en el peor de los casos, puede requerir una cantidad considerable de operaciones para explorar todos los bloques y elementos de la estructura de datos antes de determinar que el elemento no está presente.

Es importante tener en cuenta que el peor de los casos ocurre cuando el elemento buscado no está presente en la estructura de datos. Si el elemento se encuentra cerca del principio de la estructura de datos, el rendimiento será mejor y se necesitarán menos comparaciones.

En términos de complejidad espacial, sigue siendo O(n) en el peor de los casos, ya que se requiere almacenar todos los elementos en un arreglo.

Ejemplo:

```
public class SaltoBusqueda {


    public static int saltoBusqueda(int[] arr, int elemento) {
        int n = arr.length;
        int salto = (int) Math.sqrt(n); // Calcula el tamaño del salto inicial


        // Encuentra el bloque donde puede estar el elemento
        int previo = 0;
        while (arr[Math.min(salto, n) - 1] < elemento) {
            previo = salto;
            salto += (int) Math.sqrt(n);
            if (previo >= n) { // Si el bloque actual es el último
                return -1; // Elemento no encontrado
            }
        }


        // Realiza una búsqueda lineal en el bloque encontrado
```

```
        while (arr[previo] < elemento) {
            previo++;
            if (previo == Math.min(salto, n)) {
                return -1; // Elemento no encontrado
            }
        }
        if (arr[previo] == elemento) {
            return previo; // Elemento encontrado
        }
        return -1; // Elemento no encontrado
    }

    public static void main(String[] args) {
        // Arreglo ordenado del 1 al 100
        int[] arreglo = new int[100];
        for (int i = 0; i < 100; i++) {
            arreglo[i] = i + 1;
        }

        // Elemento a buscar: 101 (no presente en el arreglo)
        int elementoBuscar = 101;

        // Llamada al método de salto de búsqueda
```

```
        int indice = saltoBusqueda(arreglo, elementoBuscar);


        if (indice != -1) {
            System.out.println("Elemento encontrado en el índice: " + indice);
        } else {
            System.out.println("Elemento no encontrado");
        }
    }
}
```

- **Complejidad en el tiempo y complejidad espacial.**

La complejidad en el tiempo del método de salto de búsqueda depende del tamaño de la estructura de datos en la que se realiza la búsqueda, que se denota como "n". La complejidad en el tiempo se puede analizar en términos de las comparaciones realizadas durante la ejecución del algoritmo.

En el peor de los casos, cuando el elemento buscado no está presente en la estructura de datos, el método de salto de búsqueda realizará aproximadamente "sqrt(n)" comparaciones durante la fase de salto para encontrar el bloque donde el elemento podría estar. Luego, se realizarán comparaciones adicionales en la fase de búsqueda lineal dentro de ese bloque, que puede requerir hasta "sqrt(n)" comparaciones adicionales.

Por lo tanto, en el peor de los casos, la complejidad en el tiempo del método de salto de búsqueda es aproximadamente O(sqrt(n)), ya que el número total de comparaciones realizadas está relacionado con la raíz cuadrada del tamaño de la estructura de datos.

Es importante tener en cuenta que este análisis se basa en el peor de los casos y asume que el arreglo está ordenado de manera ascendente. En el mejor de los casos, cuando el elemento buscado se encuentra rápidamente en el primer bloque, la complejidad en el tiempo puede ser mejor que O(sqrt(n)). Sin embargo, en el caso promedio, la complejidad en el tiempo sigue siendo cercana a O(sqrt(n)), aunque puede variar dependiendo de la distribución de los datos y la ubicación del elemento buscado.

La complejidad espacial del método de salto de búsqueda es relativamente baja y está determinada principalmente por el tamaño de la estructura de datos y algunas variables auxiliares utilizadas durante la ejecución del algoritmo.

En el método de salto de búsqueda, se requiere un arreglo para almacenar los elementos de la estructura de datos. El tamaño de este arreglo es "n", que representa el número de elementos en la estructura de datos. Por lo tanto, la complejidad espacial del arreglo es O(n), ya que se necesita espacio para almacenar todos los elementos.

Además del arreglo, también se utilizan algunas variables auxiliares, como "salto", "previo" y "elemento" para realizar los cálculos y controlar el flujo del algoritmo. Estas variables son de tipo entero y su espacio de almacenamiento es constante, por lo que su contribución a la complejidad espacial se considera insignificante.

En resumen, la complejidad espacial del método de salto de búsqueda es O(n), ya que se necesita espacio para almacenar los elementos de la estructura de datos en un arreglo.

Búsqueda de interpolación

Investigar

- **En que consiste la búsqueda de interpolación.**

El algoritmo de búsqueda de interpolación es una mejora del algoritmo de búsqueda binaria que utiliza la interpolación lineal para calcular una estimación más precisa de la ubicación del elemento buscado. En lugar de dividir el rango de búsqueda en dos partes iguales, el algoritmo de interpolación utiliza una estimación lineal basada en los valores extremos de la estructura de datos.

A continuación, se presenta un ejemplo simplificado de cómo funciona el algoritmo de interpolación:

1. Supongamos que tenemos un arreglo ordenado de elementos y queremos buscar un valor específico llamado "elementoBuscado".
2. Comparamos "elementoBuscado" con el primer y último elemento del arreglo.
3. Utilizando la fórmula de interpolación, calculamos una estimación de la posición del "elementoBuscado" dentro del rango.
4. Si el "elementoBuscado" coincide con la estimación, se ha encontrado el elemento y se devuelve su posición.

5. Si el "elementoBuscado" es menor que la estimación, se reduce el rango de búsqueda al subarreglo izquierdo y se repiten los pasos 2 a 4.
6. Si el "elementoBuscado" es mayor que la estimación, se reduce el rango de búsqueda al subarreglo derecho y se repiten los pasos 2 a 4.
7. Si el rango de búsqueda se reduce a un solo elemento y no coincide con el "elementoBuscado", entonces el elemento no está presente en la estructura de datos y se devuelve un valor de indicador (-1 o null, dependiendo del lenguaje de programación).

Es importante destacar que el algoritmo de interpolación asume una distribución uniforme de los elementos en la estructura de datos. Si la distribución no es uniforme, la eficiencia del algoritmo puede degradarse y puede ser más adecuado utilizar otros métodos de búsqueda, como la búsqueda binaria estándar.

En resumen, la búsqueda de interpolación es un algoritmo de búsqueda que utiliza la interpolación lineal para realizar estimaciones más precisas de la posición de un elemento dentro de una estructura de datos ordenada.

- **Ejemplos de implementaciones del algoritmo en las estructuras de datos:**

```
public class InterpolationSearch {

    public static int interpolationSearch(int[] arr, int target) {

        int low = 0;

        int high = arr.length - 1;



        while (low <= high && target >= arr[low] && target <= arr[high]) {

            if (low == high) {

                if (arr[low] == target) {

                    return low;

                }

                return -1;

            }
```

```
        // Fórmula de interpolación para calcular la posición estimada
        int pos = low + (int) (((double) (high - low) / (arr[high] - arr[low])) * (target - arr[low]));

        if (arr[pos] == target) {
            return pos;
        }

        if (arr[pos] < target) {
            low = pos + 1;
        } else {
            high = pos - 1;
        }
    }

    return -1;
}

public static void main(String[] args) {
    int[] arr = {1, 2, 3, 4, 5, 6, 7, 8, 9};
    int target = 6;
    int result = interpolationSearch(arr, target);
    if (result == -1) {
```

```
            System.out.println("Element not found");
        } else {
            System.out.println("Element found at index " + result);
        }
    }
}
```

En este ejemplo, la función **interpolationSearch** toma un arreglo **arr** y un valor objetivo **target** como parámetros. Devuelve la posición del valor objetivo en el arreglo si se encuentra, de lo contrario, devuelve -1.

- **Análisis de eficiencia**

Es importante tener en cuenta que la eficiencia del algoritmo de búsqueda de interpolación depende en gran medida de la distribución de los elementos en la estructura de datos. Si los elementos están uniformemente distribuidos, el algoritmo puede ser muy eficiente. Sin embargo, si la distribución no es uniforme, la eficiencia puede verse afectada negativamente.

En resumen, el algoritmo de búsqueda de interpolación puede ser eficiente en el mejor caso y en promedio cuando los elementos están uniformemente distribuidos. Sin embargo, su eficiencia puede degradarse en el peor caso o cuando la distribución de los elementos no es uniforme. Por lo tanto, es importante considerar la distribución de los datos al decidir utilizar este algoritmo en una estructura de datos específica.

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

Mejor de los casos
En el mejor caso para la búsqueda de interpolación en estructuras de datos, se asume que los elementos están uniformemente distribuidos. En este escenario ideal, la fórmula de interpolación puede realizar estimaciones precisas y alcanzar una búsqueda constante, lo que resulta en una eficiencia óptima.

En el mejor caso, el algoritmo de búsqueda de interpolación tiene una complejidad temporal de O(1), lo que significa que el tiempo de ejecución no depende del tamaño de la estructura de datos. La búsqueda se realiza directamente en la posición estimada, y si el valor buscado coincide con el valor en esa posición, se encuentra de inmediato.

El mejor caso ocurre cuando el valor buscado es exactamente igual al valor en la posición estimada. En este escenario, la interpolación lineal proporciona una estimación precisa, y no es necesario realizar más comparaciones o ajustar los límites del rango de búsqueda.

Sin embargo, es importante tener en cuenta que el mejor caso es un escenario ideal y puede ser poco probable en situaciones reales. En la práctica, la distribución de los elementos puede variar y afectar la precisión de la estimación, lo que lleva a casos promedio o peores en términos de eficiencia.

Por lo tanto, al analizar la eficiencia de la búsqueda de interpolación en estructuras de datos, es esencial considerar no solo el mejor caso, sino también el promedio y el peor caso, ya que estos proporcionan una visión más completa de su rendimiento en diferentes situaciones.

Ejemplo: Supongamos que tenemos un arreglo ordenado de números del 1 al 100. Deseamos buscar el número 50 en este arreglo utilizando el algoritmo de búsqueda de interpolación.

En el mejor caso, la fórmula de interpolación calculará la posición estimada de manera precisa, y el valor buscado coincidirá exactamente con el valor en esa posición estimada.

```
public class InterpolationSearch {

    public static int interpolationSearch(int[] arr, int target) {

        int low = 0;

        int high = arr.length - 1;



        // Fórmula de interpolación para calcular la posición estimada

        int pos = low + (int) (((double) (target - arr[low]) / (arr[high] - arr[low])) * (high - low));



        if (arr[pos] == target) {

            return pos;

        } else {
```

```
            return -1;
        }
    }

    public static void main(String[] args) {
        int[] arr = new int[100];
        for (int i = 0; i < 100; i++) {
            arr[i] = i + 1;
        }
        int target = 50;

        int result = interpolationSearch(arr, target);
        if (result != -1) {
            System.out.println("El número " + target + " se encuentra en la posición " + result);
        } else {
            System.out.println("El número " + target + " no se encontró en el arreglo.");
        }
    }
}
```

Caso medio

El caso medio para la búsqueda de interpolación en estructuras de datos ocurre cuando los elementos no están uniformemente distribuidos y la estimación de posición no es precisa en todos los casos. En este escenario, el rendimiento del algoritmo puede variar dependiendo de la distribución de los datos.

La complejidad temporal promedio del algoritmo de búsqueda de interpolación en el caso medio se estima en O(log(log(n))), donde n es el tamaño de la estructura de datos. Sin embargo, es importante tener en cuenta que esta estimación puede variar en función de la distribución específica de los elementos.

En el caso medio, el algoritmo de búsqueda de interpolación realizará una serie de estimaciones y ajustes de rango para acercarse al valor buscado. La fórmula de interpolación se utilizará para calcular la posición estimada, y luego se realizarán comparaciones y ajustes según el valor en esa posición.

La eficiencia en el caso medio dependerá de la distribución de los datos y de la capacidad de la fórmula de interpolación para realizar estimaciones precisas. Si los datos están más o menos uniformemente distribuidos, el algoritmo puede ser bastante eficiente en el caso medio. Sin embargo, si la distribución es desigual o si hay agrupaciones de valores, el rendimiento del algoritmo puede verse afectado negativamente.

En resumen, en el caso medio, el algoritmo de búsqueda de interpolación tiene una complejidad temporal estimada de O(log(log(n))). Sin embargo, la eficiencia real puede variar dependiendo de la distribución específica de los elementos en la estructura de datos.

Ejemplo: Supongamos que tenemos un arreglo ordenado de números del 1 al 1000, pero los valores están más concentrados hacia el extremo inferior del rango. Deseamos buscar el número 750 en este arreglo utilizando el algoritmo de búsqueda de interpolación.

```
public class InterpolationSearch {

    public static int interpolationSearch(int[] arr, int target) {

        int low = 0;

        int high = arr.length - 1;



        while (low <= high && target >= arr[low] && target <= arr[high]) {

            // Fórmula de interpolación para calcular la posición estimada

            int pos = low + (int) (((double) (target - arr[low]) / (arr[high] - arr[low])) *
(high - low));
```

```
        if (arr[pos] == target) {
            return pos;
        }

        if (arr[pos] < target) {
            low = pos + 1;
        } else {
            high = pos - 1;
        }
    }

    return -1;
}

public static void main(String[] args) {
    int[] arr = new int[1000];
    for (int i = 0; i < 1000; i++) {
        arr[i] = i + 1;
    }
    int target = 750;

    int result = interpolationSearch(arr, target);
    if (result != -1) {
```

```
            System.out.println("El número " + target + " se encuentra en la posición " +
result);

        } else {

            System.out.println("El número " + target + " no se encontró en el arreglo.");

        }

    }

}
```

Peor de los casos

En el peor de los casos, la búsqueda de interpolación puede tener una complejidad temporal lineal, es decir, O(n), donde n es el tamaño de la estructura de datos. Esto ocurre cuando los elementos están distribuidos de una manera que no permite realizar estimaciones precisas de la posición buscada.

Un ejemplo del peor caso es cuando la estructura de datos está compuesta por elementos repetidos o tiene una distribución irregular. En este escenario, la fórmula de interpolación no puede hacer estimaciones precisas y el algoritmo de búsqueda termina realizando una serie de comparaciones lineales para encontrar el valor buscado.

Ejemplo:

```
 public class InterpolationSearch {

     public static int interpolationSearch(int[] arr, int target) {

         int low = 0;

         int high = arr.length - 1;



         while (low <= high && target >= arr[low] && target <= arr[high]) {

             // Fórmula de interpolación para calcular la posición estimada

             int pos = low + (int) (((double) (target - arr[low]) / (arr[high] - arr[low])) *
 (high - low));
```

```
            if (arr[pos] == target) {
                return pos;
            }

            if (arr[pos] < target) {
                low = pos + 1;
            } else {
                high = pos - 1;
            }
        }

        return -1;
    }

    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};  // Arreglo con valores repetidos
        int target = 7;

        int result = interpolationSearch(arr, target);
        if (result != -1) {
            System.out.println("El número " + target + " se encuentra en la posición " +
result);
        } else {
```

```
            System.out.println("El número " + target + " no se encontró en el arreglo.");
        }
    }
}
```

- **Complejidad en el tiempo y complejidad espacial.**

Complejidad temporal:

- En el mejor caso, cuando los elementos están uniformemente distribuidos en la estructura de datos, la complejidad del algoritmo de búsqueda de interpolación es O(1), es decir, una búsqueda constante. Esto ocurre cuando el valor buscado coincide exactamente con el valor en la posición estimada.
- En el peor caso, la complejidad del algoritmo de búsqueda de interpolación es O(n), donde n es el número de elementos en la estructura de datos. Esto ocurre cuando los elementos están muy dispersos y la fórmula de interpolación no logra estimar la posición de manera precisa, resultando en un recorrido lineal del rango de búsqueda.
- En promedio, la complejidad del algoritmo de búsqueda de interpolación es aproximadamente O(log(log(n))) si los elementos están distribuidos de manera uniforme. Sin embargo, si la distribución de los elementos no es uniforme, la complejidad puede acercarse a O(n).

Es importante tener en cuenta que la eficiencia del algoritmo de búsqueda de interpolación depende en gran medida de la distribución de los elementos en la estructura de datos. Si los elementos están uniformemente distribuidos, el algoritmo puede ser muy eficiente. Sin embargo, si la distribución no es uniforme, la eficiencia puede verse afectada negativamente.

Complejidad espacial:

- El algoritmo de búsqueda de interpolación no requiere espacio adicional más allá de los requerimientos de la estructura de datos utilizada para almacenar los elementos. Por lo tanto, la complejidad espacial es O(1), es decir, constante.

Es importante destacar que el algoritmo de búsqueda de interpolación es más eficiente en estructuras de datos ordenadas, ya que requiere una distribución de valores para realizar una estimación precisa. Además, el algoritmo puede no ser adecuado para estructuras de datos en las que el acceso aleatorio no sea eficiente, como las listas enlazadas.

Búsqueda exponencial

- **En que consiste la búsqueda exponencial.**

El método de búsqueda exponencial, también conocido como búsqueda por saltos o búsqueda binaria exponencial, es un algoritmo utilizado para buscar un elemento en una secuencia ordenada. Este método es una mejora de la búsqueda binaria convencional y aprovecha la estructura ordenada de los datos para reducir el número de comparaciones realizadas.

El algoritmo de búsqueda exponencial sigue los siguientes pasos:

1. Comienza con un rango de búsqueda inicial que abarca toda la secuencia ordenada.

2. Compara el elemento buscado con el elemento en la posición inicial del rango.

- Si son iguales, se ha encontrado el elemento y se devuelve su posición.

- Si el elemento buscado es menor, se termina la búsqueda, ya que el elemento no está en la secuencia.

- Si el elemento buscado es mayor, continúa con el siguiente paso.

3. Duplica el tamaño del rango de búsqueda en cada iteración (salto exponencial) hasta que se encuentre un elemento mayor o igual al elemento buscado o se supere el tamaño de la secuencia.

4. Realiza una búsqueda binaria en el rango actualizado, limitado por el rango anterior y el tamaño de la secuencia, para encontrar la posición exacta del elemento buscado.

El método de búsqueda exponencial aprovecha el hecho de que si el elemento buscado está cerca del final del rango actualizado, es más probable que esté en la segunda mitad del rango, y por lo tanto, se reduce la búsqueda a la mitad en cada iteración. Esto permite una búsqueda más eficiente en comparación con la búsqueda binaria convencional.

Es importante destacar que el método de búsqueda exponencial requiere que los datos estén ordenados y que el acceso a elementos en posiciones arbitrarias sea eficiente (por ejemplo, en un arreglo). También es importante considerar que, en algunos casos, la búsqueda exponencial puede ser menos eficiente que la búsqueda binaria convencional, especialmente si los datos están dispersos o no siguen una distribución uniforme.

- **Ejemplos de implementaciones del algoritmo en las estructuras de datos:**

El método de búsqueda exponencial se puede implementar en varias estructuras de datos en Java.

```
public class ExponentialSearch {
```

```
public static int exponentialSearch(int[] arr, int target) {
    if (arr[0] == target)
        return 0;
    int size = arr.length;
    int range = 1;
    while (range < size && arr[range] <= target)
        range *= 2;
    return binarySearch(arr, target, range / 2, Math.min(range, size - 1));   }
private static int binarySearch(int[] arr, int target, int low, int high) {
    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (arr[mid] == target)
            return mid;
        if (arr[mid] < target)
            low = mid + 1;
        else
            high = mid - 1;   }
    return -1;    }
public static void main(String[] args) {
    int[] arr = { 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 };
    int target = 12;
    int result = exponentialSearch(arr, target);
    if (result == -1)
        System.out.println("El elemento no se encuentra en el arreglo.");
    else
        System.out.println("El elemento se encuentra en el índice: " + result);    }}
```

En este ejemplo, tenemos el método exponentialSearch que toma un arreglo ordenado arr y un valor objetivo target. El algoritmo utiliza el método de búsqueda

exponencial para encontrar la posición del target en el arreglo. Si el elemento se encuentra, se devuelve su índice; de lo contrario, se devuelve -1.

Dentro del método exponentialSearch, se utiliza un bucle para duplicar el tamaño del rango de búsqueda en cada iteración hasta encontrar un elemento mayor o igual al objetivo, o hasta que se supere el tamaño del arreglo. Luego, se llama al método binarySearch para realizar una búsqueda binaria en el rango actualizado y encontrar la posición exacta del elemento buscado.

En el método binarySearch, se implementa la búsqueda binaria convencional para encontrar el elemento en el rango especificado.

En el método main, se crea un arreglo ordenado arr y se establece el valor objetivo target como 12. Luego, se llama al método exponentialSearch y se muestra el resultado en la consola.

- **Análisis de eficiencia**

El análisis de eficiencia del algoritmo de búsqueda exponencial se basa en su tiempo de ejecución y su complejidad en función del tamaño de los datos de entrada.

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

Mejor caso: El mejor caso ocurre cuando el elemento buscado se encuentra en la primera posición del arreglo. En este caso, el algoritmo de búsqueda exponencial tiene una complejidad de tiempo de O(1). Esto se debe a que el elemento objetivo se encuentra directamente en la posición inicial, y no es necesario realizar ninguna comparación adicional ni realizar la búsqueda binaria.

```
int[] arr = { 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 };

int target = 2;

int result = exponentialSearch(arr, target);
```

En este ejemplo, el elemento buscado es 2, que se encuentra en la primera posición del arreglo. Como resultado, el algoritmo devuelve el índice 0 sin necesidad de realizar más comparaciones.

Caso medio: El caso medio ocurre cuando el elemento buscado se encuentra en una posición aleatoria dentro del arreglo. En este caso, la complejidad de tiempo del algoritmo de búsqueda exponencial es O(log i), donde i es la posición del elemento buscado.

```
int[] arr = { 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 };

int target = 12;
```

int result = exponentialSearch(arr, target);

En este ejemplo, el elemento buscado es 12, que se encuentra en la posición 5 del arreglo. El algoritmo primero realiza saltos exponenciales para encontrar un rango aproximado y luego realiza una búsqueda binaria en ese rango para encontrar el elemento exacto. En promedio, la cantidad de iteraciones necesarias para encontrar el elemento buscado es log i, donde i es la posición del elemento.

Peor caso: El peor caso ocurre cuando el elemento buscado está en la última posición del arreglo o no está presente en absoluto. En este caso, el algoritmo de búsqueda exponencial tiene una complejidad de tiempo de O(log n), donde n es el tamaño del arreglo.

int[] arr = { 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 };

int target = 30;

int result = exponentialSearch(arr, target);

En este ejemplo, el elemento buscado es 30, que no está presente en el arreglo. El algoritmo realizará saltos exponenciales hasta que el rango de búsqueda exceda el tamaño del arreglo, y luego realizará una búsqueda binaria en ese rango. En el peor caso, la cantidad de iteraciones necesarias es log n, donde n es el tamaño del arreglo.

- **Complejidad en el tiempo y complejidad espacial.**

La complejidad en el tiempo y la complejidad espacial del algoritmo de búsqueda exponencial son las siguientes:

Complejidad en el tiempo:

- Mejor caso: O(1) - Si el elemento buscado se encuentra en la primera posición del arreglo, no se requieren iteraciones adicionales.

- Caso medio: O(log i) - Donde `i` es la posición del elemento buscado en el arreglo. En promedio, se requieren log i iteraciones para encontrar el elemento.

- Peor caso: O(log n) - Donde `n` es el tamaño del arreglo. En el peor caso, se requieren log n iteraciones para determinar que el elemento no está presente en el arreglo.

La complejidad en el tiempo del algoritmo de búsqueda exponencial es más eficiente que la búsqueda lineal O(n), pero puede ser menos eficiente que la búsqueda binaria O(log n) en algunos casos.

Complejidad espacial:

La complejidad espacial del algoritmo de búsqueda exponencial es O(1), lo que significa que no requiere memoria adicional que crezca con el tamaño del arreglo.

Solo se utilizan variables adicionales para almacenar las posiciones y límites de búsqueda, y no dependen del tamaño del arreglo.

En resumen, el algoritmo de búsqueda exponencial tiene una complejidad en el tiempo de O(log i) en el caso medio y O(log n) en el peor caso, y una complejidad espacial de O(1). La eficiencia del algoritmo depende de la posición del elemento buscado en el arreglo y del tamaño del arreglo. En promedio, es más rápido que la búsqueda lineal, pero en algunos casos puede ser menos eficiente que la búsqueda binaria.

Búsqueda de Fibonacci

Investigar

- **En que consiste la búsqueda de Fibonacci**

El método de búsqueda de Fibonacci es un algoritmo de búsqueda utilizado para encontrar un elemento en una secuencia ordenada. Este método es una variante de la búsqueda binaria y utiliza la sucesión de Fibonacci para determinar los puntos de división en el rango de búsqueda.

El algoritmo de búsqueda de Fibonacci sigue los siguientes pasos:

1. Inicializa los dos primeros números de la sucesión de Fibonacci: F(0) = 0 y F(1) = 1.

2. Calcula el número de Fibonacci más pequeño que sea igual o mayor que el tamaño del arreglo en el que se va a buscar.

3. Establece tres variables: `offset` para almacenar el índice base actual, `prevOffset` para almacenar el índice base previo y `mid` para almacenar el punto de división actual.

4. Compara el elemento buscado con el elemento en la posición `mid` del arreglo.

- Si son iguales, se ha encontrado el elemento y se devuelve su posición.

- Si el elemento buscado es menor, se actualizan las variables `offset`, `prevOffset` y `mid` usando la sucesión de Fibonacci y se repite el paso 4.

- Si el elemento buscado es mayor, se actualizan las variables `offset`, `prevOffset` y `mid` usando la sucesión de Fibonacci y se repite el paso 4.

5. Si el rango de búsqueda se ha reducido a un solo elemento y no es igual al elemento buscado, se termina la búsqueda y se devuelve -1 para indicar que el elemento no está presente en el arreglo.

El método de búsqueda de Fibonacci utiliza la sucesión de Fibonacci para determinar los puntos de división en el rango de búsqueda, de manera similar a cómo la búsqueda binaria utiliza la mitad del rango. Esto permite reducir el rango de búsqueda de manera más eficiente en comparación con la búsqueda binaria convencional.

Es importante destacar que el método de búsqueda de Fibonacci requiere que los datos estén ordenados y que el acceso a elementos en posiciones arbitrarias sea eficiente (por ejemplo, en un arreglo). También es importante considerar que, en algunos casos, la búsqueda de Fibonacci puede ser menos eficiente que la búsqueda binaria, especialmente si los datos están dispersos o no siguen una distribución uniforme.

- **Ejemplos de implementaciones del algoritmo en las estructuras de datos:**

```
public class FibonacciSearch {
    public static int fibonacciSearch(int[] arr, int target) {
        int fibM2 = 0; // Fibonacci(i-2)
        int fibM1 = 1; // Fibonacci(i-1)
        int fib = fibM2 + fibM1; // Fibonacci(i)
        while (fib < arr.length) {
            fibM2 = fibM1;
            fibM1 = fib;
            fib = fibM2 + fibM1;
        }
        int offset = -1; // Offset inicial
        while (fib > 1) {
            int i = Math.min(offset + fibM2, arr.length - 1);
            if (arr[i] == target)
                return i;
            if (arr[i] < target) {
                fib = fibM1;
                fibM1 = fibM2;
```

```
            fibM2 = fib - fibM1;
            offset = i;
        } else {
            fib = fibM2;
            fibM1 = fibM1 - fibM2;
            fibM2 = fib - fibM1;
        }
    }
    if (fibM1 == 1 && arr[offset + 1] == target)
        return offset + 1;

    return -1;
}
public static void main(String[] args) {
    int[] arr = { 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 };
    int target = 12;
    int result = fibonacciSearch(arr, target);
    if (result == -1)
        System.out.println("El elemento no se encuentra en el arreglo.");
    else
        System.out.println("El elemento se encuentra en el índice: " + result);}}
```

En este ejemplo, tenemos el método fibonacciSearch que toma un arreglo ordenado arr y un valor objetivo target. El algoritmo utiliza el método de búsqueda de Fibonacci para encontrar la posición del target en el arreglo. Si el elemento se encuentra, se devuelve su índice; de lo contrario, se devuelve -1.

Dentro del método fibonacciSearch, se inicializan las variables para la sucesión de Fibonacci y se calcula el número de Fibonacci más pequeño que sea igual o mayor que el tamaño del arreglo. Luego, se realiza el proceso de búsqueda utilizando las variables fibM2, fibM1 y fib para controlar los puntos de división en el rango de búsqueda. Se comparan los elementos en la posición correspondiente y se actualizan las variables de acuerdo a si el elemento buscado es menor o mayor.

En el método main, se crea un arreglo ordenado arr y se establece el valor objetivo target como 12. Luego, se llama al método fibonacciSearch y se muestra el resultado en la consola.

- **Análisis de eficiencia**

El análisis de los casos en el algoritmo de búsqueda de Fibonacci se basa en el número de iteraciones requeridas para encontrar un elemento en una secuencia ordenada.

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

Mejor caso: El mejor caso ocurre cuando el elemento buscado se encuentra en la primera posición del arreglo. En este caso, el algoritmo de búsqueda de Fibonacci tiene una eficiencia óptima, ya que solo requerirá una comparación y devolverá el resultado.

```
int[] arr = { 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 };

int target = 2;

int result = fibonacciSearch(arr, target);
```

En este ejemplo, el elemento buscado es 2, que se encuentra en la primera posición del arreglo. El algoritmo realizará solo una comparación y devolverá el índice 0 como resultado.

Caso medio: El caso medio ocurre cuando el elemento buscado se encuentra en una posición aleatoria dentro del arreglo. En este caso, el algoritmo de búsqueda de Fibonacci tiene una eficiencia aceptable y requiere un número moderado de iteraciones para encontrar el elemento.

```
int[] arr = { 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 };

int target = 12;

int result = fibonacciSearch(arr, target);
```

En este ejemplo, el elemento buscado es 12, que se encuentra en la posición 5 del arreglo. El algoritmo realizará un número moderado de iteraciones utilizando la sucesión de Fibonacci para determinar los puntos de división en el rango de búsqueda. La cantidad de iteraciones dependerá de la posición del elemento y puede variar.

Peor caso: El peor caso ocurre cuando el elemento buscado está en la última posición del arreglo o no está presente en absoluto. En este caso, el algoritmo de búsqueda de Fibonacci tiene una eficiencia subóptima, ya que requerirá el mayor número de iteraciones para determinar que el elemento no está presente.

```
int[] arr = { 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 };

int target = 30;

int result = fibonacciSearch(arr, target);
```

En este ejemplo, el elemento buscado es 30, que no está presente en el arreglo. El algoritmo realizará un mayor número de iteraciones utilizando la sucesión de Fibonacci para reducir el rango de búsqueda. Eventualmente, el algoritmo determinará que el elemento no está presente y devolverá -1 como resultado.

- **Complejidad en el tiempo y complejidad espacial**

Complejidad en el tiempo:

- Mejor caso: O(1) - En el mejor caso, el elemento buscado se encuentra en la primera posición del arreglo y se devuelve después de una única comparación.

- Caso medio y peor caso: O(log n) - En el caso medio y en el peor caso, el algoritmo de búsqueda de Fibonacci realiza un número de iteraciones proporcional al logaritmo en base 2 del tamaño del arreglo `n`. A medida que el tamaño del arreglo aumenta, el número de iteraciones necesarias para encontrar el elemento buscado también aumenta, pero a una tasa menor que lineal.

Complejidad espacial:

La complejidad espacial del algoritmo de búsqueda de Fibonacci es O(1), es decir, constante. El algoritmo no requiere memoria adicional que crezca con el tamaño del arreglo. Solo se utilizan algunas variables adicionales para almacenar los números de Fibonacci y las posiciones de búsqueda, pero su cantidad es constante y no depende del tamaño del arreglo.

En resumen, el algoritmo de búsqueda de Fibonacci tiene una complejidad en el tiempo de O(1) en el mejor caso y O(log n) en el caso medio y peor caso. La complejidad espacial es O(1). La eficiencia del algoritmo es notable, ya que el número de iteraciones necesarias para encontrar un elemento en un arreglo ordenado aumenta de manera logarítmica en lugar de lineal.