GUÍA DE APRENDIZAJE PARA CONCURSANTES ICPC Y IOI: BÚSQUEDA BINARIA

1. Introducción

La búsqueda binaria es un método que permite una búsqueda más rápida de algo al dividir el intervalo de búsqueda en dos. Su aplicación más común es la búsqueda de valores en matrices ordenadas; sin embargo, la idea de división es crucial en muchas otras tareas típicas.

2. Conocimientos previos

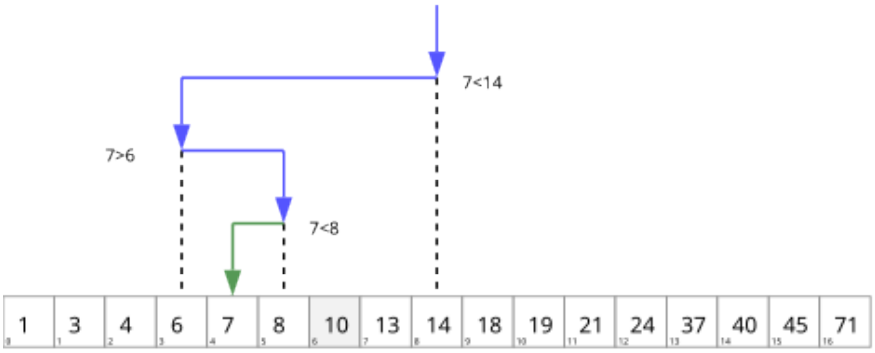
2.1. Arreglo

En programación, un arreglo (llamados en inglés array) es una zona de almacenamiento continuo, que contiene una serie de elementos del mismo tipo.

3. Desarrollo

3.1. Buscar en matriz ordenada

El problema más típico que lleva a la búsqueda binaria es el siguiente. Te dan una matriz ordenada $A_0 \leq A_1 \leq \cdots \leq A_{n-1}$, comprobar si k está presente dentro de la secuencia. La solución más sencilla sería comprobar cada elemento uno por uno y compararlo con k (la llamada búsqueda lineal). Este enfoque funciona en $O(n)$, pero no utiliza el hecho de que la matriz esté ordenada.



Ahora supongamos que conocemos dos índices $L < R$ tal que $A_L \leq k \leq A_R$. Como la matriz está ordenada, podemos deducir que k cualquiera de los dos ocurre entre $A_L, A_{L+1}, \ldots, A_R$ o no aparece en la matriz en absoluto. Si elegimos un índice arbitrario M tal que $L < M < R$ y comprobar si k es menor o mayor que A_M. Tenemos dos casos posibles:

- $A_L \leq k \leq A_M$. En este caso, reducimos el problema de $[L, R]$ a $[L, M]$.
- $A_M \leq k \leq A_R$. En este caso, reducimos el problema de $[L, R]$ a $[M, R]$.

Cuando es imposible elegir M, Eso es cuando $R = L+1$, comparamos directamente k con A_L y A_R. De lo contrario, querríamos elegir M de tal manera que reduzca el segmento activo a un solo elemento lo más rápidamente posible en el peor de los casos.

Dado que en el peor de los casos siempre reduciremos a un segmento mayor de $[L, M]$ y $[M, R]$. Así, en el peor de los casos la reducción sería de $R - L$ a $\max(M - L, R - M)$. Para minimizar este valor, debemos elegir $M \approx \frac{L+R}{2}$, entonces:

$$M - L \approx \frac{R - L}{2} \approx R - M.$$

En otras palabras, desde la perspectiva del peor de los casos, lo óptimo es elegir siempre M en el medio de $[L, R]$ y dividirlo por la mitad. Por lo tanto, el segmento activo se reduce a la mitad en cada paso hasta que alcanza el tamaño 1. Entonces, si el proceso necesita h pasos, al final reduce la diferencia entre R y l de $R - L$ a $\frac{R-L}{2^h} \approx 1$, dándonos la ecuación $2^h \approx R - L$.

Tomando $\log_2$ en ambos lados, obtenemos $h \approx \log_2(R - L) \in O(\log n)$.

El número logarítmico de pasos es drásticamente mejor que el de la búsqueda lineal. Por ejemplo, para $n \approx 2^{20} \approx 10^6$ necesitarías realizar aproximadamente un millón de operaciones para la búsqueda lineal, pero solo alrededor 20 operaciones con la búsqueda binaria.

3.1.1. Límite inferior y límite superior

A menudo es conveniente encontrar la posición del primer elemento que no sea menor que k (llamado límite inferior de k en la matriz) o la posición del primer elemento que es mayor que k (llamado límite superior de k) en lugar de la posición exacta del elemento.

Juntos, los límites inferior y superior producen un medio intervalo posiblemente vacío de los elementos de la matriz que son iguales a k. Para comprobar si k está presente en la matriz, es suficiente encontrar su límite inferior y verificar si el elemento correspondiente equivale a k.

La explicación anterior proporciona una descripción aproximada del algoritmo. Para los detalles de implementación, necesitaríamos ser más precisos.

Mantendremos un par $L < R$ tal que $A_L \leq k < A_R$. Lo que significa que el intervalo de búsqueda activa es $[L, R)$. Usamos medio intervalo aquí en lugar de un segmento. $[L, R]$ ya que resulta que requiere menos trabajo de caso.

Cuando $R = L + 1$, podemos deducir de las definiciones anteriores que R es el límite superior de k. Es conveniente inicializar R con índice pasado el final, es decir $R = n$ y l con índice antes del comienzo, es decir $L = -1$. Está bien siempre y cuando nunca evaluemos A_L y A_R en nuestro algoritmo directamente, tratándolo formalmente como $A_L = -\infty$ y $A_R = +\infty$.

Finalmente, para ser específico sobre el valor de M elegimos, nos quedaremos con $M = \lfloor \frac{L+R}{2} \rfloor$.

3.2. Buscar en predicado arbitrario

Sea $f : \{0, 1, \ldots, n - 1\} \to \{0, 1\}$ ser una función booleana definida en $f : \{0, 1, \ldots, n - 1\} \to \{0, 1\}$ tal que aumenta monótonamente, es decir

$$f(0) \leq f(1) \leq \cdots \leq f(n - 1).$$

La búsqueda binaria, como se describe arriba, encuentra la partición de la matriz por el predicado $f(M)$, manteniendo el valor booleano de $k < A_M$ expresión. Es posible utilizar un predicado monótono arbitrario en lugar de $k < A_M$. Es particularmente útil cuando el cálculo de $f(k)$ esto requiere demasiado tiempo para calcularlo para cada valor posible. En otras palabras, la búsqueda binaria encuentra el índice único L tal que $f(L) = 0$ y $f(R) = f(L+1) = 1$ si tal punto de transición existe, o nos da $L = n - 1$ si $f(0) = \cdots = f(n-1) = 0$ o $L = -1$ si $f(0) = \cdots = f(n-1) = 1$.

Prueba de corrección suponiendo que exista un punto de transición, es decir $f(0) = 0$ y $f(n-1) = 1$: La implementación mantiene el bucle invariante $f(l) = 0, f(r) = 1$. . Cuando $r - l > 1$, la elección de m medio $r - l$ siempre disminuirá. El bucle termina cuando $r - l = 1$, dándonos nuestro punto de transición deseado.

Esta situación ocurre a menudo cuando se nos pide que calculemos algún valor, pero solo somos capaces de verificar si este valor es al menos i. Por ejemplo, te dan una matriz $a_1, \ldots, a_n$ y se le pide que encuentre la suma promedio máxima fijada

$$\left\lfloor \frac{a_l + a_{l+1} + \cdots + a_r}{r - l + 1} \right\rfloor$$

entre todos los pares posibles de l, r tal que $r - l \geq x$. Una de las formas sencillas de resolver este problema es comprobar si la respuesta es al menos λ , eso si hay un par l, r tal que lo siguiente es cierto:

$$\frac{a_l + a_{l+1} + \cdots + a_r}{r - l + 1} \geq \lambda.$$

De manera equivalente, se reescribe como

$$(a_l - \lambda) + (a_{l+1} - \lambda) + \cdots + (a_r - \lambda) \geq 0,$$

entonces ahora necesitamos verificar si hay un subarreglo de un nuevo arreglo $a_i - \lambda$ de longitud al menos $x + 1$ con suma no negativa, lo cual es factible con algunas sumas de prefijo.

3.3. Búsqueda continua

Dejar $f : \mathbf{R} \to \mathbf{R}$ ser una función de valor real que es continua en un segmento $[L, R]$.

Sin pérdida de generalidad supongamos que $f(L) \leq f(R)$. Del teorema del valor intermedio se deduce que para cualquier $y \in [f(L), f(R)]$ hay $x \in [L, R]$ tal que $f(x) = y$. Tenga en cuenta que, a diferencia de los párrafos anteriores, no es necesario que la función sea monótona.

El valor x podría aproximarse hasta $\pm\delta$ en $O\left(\log \frac{RL}{\delta}\right)$ tiempo para cualquier valor específico de δ. La idea es esencialmente la misma, si tomamos $M \in (L, R)$ entonces podríamos reducir el intervalo de búsqueda a cualquiera de los dos $[L, M]$ o $[M, R]$ dependiendo de si $f(M)$ es mas grande que y . Un ejemplo común aquí sería encontrar raíces de polinomios de grados impares.

Por ejemplo, dejemos $f(x) = x^3 + ax^2 + bx + c$. Entonces $f(L) \to -\infty$ y $f(R) \to +\infty$ con $L \to -\infty$ y $R \to +\infty$. Lo que significa que siempre es posible encontrar cantidades suficientemente pequeñas l y suficientemente grande R tal que $f(L) < 0$ y $f(R) > 0$. Entonces, es posible encontrar con búsqueda binaria un intervalo arbitrariamente pequeño que contenga x tal que $f(x) = 0$.

3.4. Buscar con potencias de 2

Otra forma notable de realizar una búsqueda binaria es, en lugar de mantener un segmento activo, mantener el puntero actual i y la potencia actual k. El puntero comienza en $i = L$ y luego en cada iteración se prueba el predicado en el punto $i + 2^k$. Si el predicado sigue siendo 0, el puntero avanza desde i a $i + 2^k$, de lo contrario permanece igual, entonces el poder k se reduce en 1.

Este paradigma se usa ampliamente en tareas relacionadas con árboles, como encontrar el ancestro común más bajo de dos vértices o encontrar un ancestro de un vértice específico que tenga una altura determinada. También podría adaptarse para, por ejemplo, encontrar el k-ésimo elemento distinto de cero en un árbol de Fenwick.

4. Implementación

La implementación de la búsqueda binaria tiene dos variantes una recursiva y otra iterativa. En este caso vamos a presentar la iterativa por ser mas conveniente para lo que deseamos utilizar.

Calcular $m = (r + l)/2$ puede provocar un desbordamiento si l y r son dos números enteros positivos, y este error duró aproximadamente 9 años en JDK de Java . Algunos enfoques alternativos incluyen, por ejemplo, escribir $m = l + (r - l)/2$, que siempre funciona para números enteros positivos l y r, pero aún podría desbordarse si les un número negativo. Si utiliza C++ 20, ofrece una solución alternativa que $m = std :: midpoint(l, r)$ siempre funciona correctamente.

4.1. C++

```
#define MAX_N 1000001

int n, x;
int niz[MAX_N];


inline int b_search(int left, int right, int x){
   int i = left;
   int j = right;
   while (i < j){
      int mid = (i+j)/2;
      if (niz[mid] == x) return mid;
      if (niz[mid] < x) i = mid+1;
      else j = mid-1;
   }
   if (niz[i] == x) return i;
```

```
    return -1;
}
```

4.1.1. Funciones de C++ relacionados con la búsqueda binaria

1. **binary_search:** Devuelve verdadero si algún elemento en el rango $[begin, end)$ es equivalente a val y falso en caso contrario. Los elementos se comparan usando operator $<$ para la primera versión y comp para la segunda. Dos elementos, a y b, se consideran equivalentes si $(!(a < b)\&\&!(b < a))$ o si $(!comp(a,b)\&\&!comp(b,a))$. Los elementos del rango ya deberán estar ordenados según este mismo criterio (operador<o comp), o al menos particionados respecto a val.

   Parámetros:

   - **first, last:** Reenvía iteradores a las posiciones inicial y final de una secuencia ordenada (o correctamente dividida). El rango utilizado es [primero, último), que contiene todos los elementos entre el primero y el último, incluido el elemento señalado por el primero pero no el elemento señalado por el último.
   - **val:** Valor a buscar en el rango. Para (1), T será un tipo que admita la comparación con elementos del rango [first, last) como cualquiera de los operandos del operador $<$.
   - **comp:** Función binaria que acepta dos argumentos del tipo señalado por ForwardIterator (y de tipo T), y devuelve un valor convertible a bool. El valor devuelto indica si se considera que el primer argumento va antes del segundo. La función no modificará ninguno de sus argumentos. Puede ser un puntero de función o un objeto de función.

```
#include <iostream>      // std::cout
#include <algorithm>     // std::binary_search, std::sort
#include <vector>        // std::vector

bool myfunction (int i,int j) { return (i<j); }

int main () {
   int myints[] = {1,2,3,4,5,4,3,2,1};
   std::vector<int> v(myints,myints+9); // 1 2 3 4 5 4 3 2 1

   // usando comparacion predeterminada:
   std::sort (v.begin(), v.end());

   std::cout << "buscando un 3... ";
   if (std::binary_search (v.begin(), v.end(), 3))
   std::cout << "encontrado!\n"; else std::cout << "no encontrado.\n";

   // usando mi funcion como comparacion:
   std::sort (v.begin(), v.end(), myfunction);

   std::cout << "buscando un 6... ";
   if (std::binary_search (v.begin(), v.end(), 6, myfunction))
   std::cout << "encontrado!\n"; else std::cout << "no encontrado.\n";
```

```
    return 0;
}
```

2. **lower_bound:** Devuelve un iterador que apunta al primer elemento del rango $[begin, end)$ que no compara menos que val. Los elementos se comparan usando operator $<$ para la primera versión y comp para la segunda. Los elementos del rango ya deberán estar ordenados según este mismo criterio (operador $<$ o comp), o al menos particionados respecto a val. La función optimiza el número de comparaciones realizadas comparando elementos no consecutivos del rango ordenado, lo cual es especialmente eficiente para iteradores de acceso aleatorio. A diferencia de upper_bound, el valor señalado por el iterador devuelto por esta función también puede ser equivalente a val, y no solo mayor.

   Parámetros:

   - **first, last:** Reenvía iteradores a las posiciones inicial y final de una secuencia ordenada (o correctamente dividida). El rango utilizado es [primero, último), que contiene todos los elementos entre el primero y el último, incluido el elemento señalado por el primero pero no el elemento señalado por el último.
   - **val:** Valor del límite inferior a buscar en el rango. Para (1), T será un tipo que admita la comparación con elementos del rango [primero, último) como operando del lado derecho del operador <.
   - **comp:** Función binaria que acepta dos argumentos (el primero del tipo señalado por ForwardIterator, y el segundo, siempre val), y devuelve un valor convertible a bool. El valor devuelto indica si se considera que el primer argumento va antes del segundo. La función no modificará ninguno de sus argumentos. Puede ser un puntero de función o un objeto de función.

```
#include <iostream>  // std::cout
#include <algorithm> // std::lower_bound, std::upper_bound, std::sort
#include <vector>    // std::vector

int main () {
   int myints[] = {10,20,30,30,20,10,10,20};
   std::vector<int> v(myints,myints+8);           // 10 20 30 30 20 10 10 20

   std::sort (v.begin(), v.end());                // 10 10 10 20 20 20 30 30

   std::vector<int>::iterator low,up;
   low=std::lower_bound (v.begin(), v.end(), 20); //          ^
   up= std::upper_bound (v.begin(), v.end(), 20); //                   ^

   std::cout << "limite inferior en la posicion " << (low- v.begin()) << '\n';
   std::cout << "limite superior en la posicion " << (up - v.begin()) << '\n';

   return 0;
}
```

3. **upper_bound:** Devuelve un iterador que apunta al primer elemento del rango $[begin, end)$ que compara mayor que val. Los elementos se comparan usando operator $<$ para la primera versión y comp para la segunda. Los elementos del rango ya deberán estar ordenados según este mismo criterio (operador $<$ o comp), o al menos particionados respecto a val. La función optimiza el número de comparaciones realizadas comparando elementos no consecutivos del rango ordenado, lo cual es especialmente eficiente para iteradores de acceso aleatorio. A diferencia de lower_bound, el valor señalado por el iterador devuelto por esta función no puede ser equivalente a val, solo mayor.

   Parámetros:

   - **first, last:** Reenvía iteradores a las posiciones inicial y final de una secuencia ordenada (o correctamente dividida). El rango utilizado es [primero, último), que contiene todos los elementos entre el primero y el último, incluido el elemento señalado por el primero pero no el elemento señalado por el último.
   - **val:** Valor del límite superior a buscar en el rango. Para (1), T será un tipo que admita la comparación con elementos del rango [primero, último) como operando del lado izquierdo del operador <.
   - **comp:** Función binaria que acepta dos argumentos (el primero del tipo señalado por ForwardIterator, y el segundo, siempre val), y devuelve un valor convertible a bool. El valor devuelto indica si se considera que el primer argumento va antes del segundo. La función no modificará ninguno de sus argumentos. Puede ser un puntero de función o un objeto de función.

```cpp
#include <iostream>  // std::cout
#include <algorithm> // std::lower_bound, std::upper_bound, std::sort
#include <vector>    // std::vector

int main () {
   int myints[] = {10,20,30,30,20,10,10,20};
   std::vector<int> v(myints,myints+8);           // 10 20 30 30 20 10 10 20

   std::sort (v.begin(), v.end());                // 10 10 10 20 20 20 30 30

   std::vector<int>::iterator low,up;
   low=std::lower_bound (v.begin(), v.end(), 20); //          ^
   up= std::upper_bound (v.begin(), v.end(), 20); //                   ^

   std::cout << "limite inferior en la posicion " << (low- v.begin()) << '\n';
   std::cout << "limite superior en la posicion " << (up - v.begin()) << '\n';

   return 0;
}
```

4. **equal_range:** Devuelve los límites del subrango que incluye todos los elementos del rango $[begin, end)$ con valores equivalentes a val. Los elementos se comparan usando operator $<$ para la primera versión y comp para la segunda. Dos elementos, a y b, se consideran equi-

valentes si $(!(a < b)\&\&!(b < a))$ o si $(!comp(a, b)\&\&!comp(b, a))$. Los elementos del rango ya deberán estar ordenados según este mismo criterio (operador $<$ o comp), o al menos particionados respecto a val. Si val no es equivalente a ningún valor en el rango, el subrango devuelto tiene una longitud de cero, con ambos iteradores apuntando al valor más cercano mayor que val, si lo hay, o al último, si val se compara con un valor mayor que todos los elementos en el rango. rango.

Parámetros:

- **first, last:** Reenvía iteradores a las posiciones inicial y final de una secuencia ordenada (o correctamente dividida). El rango utilizado es [primero, último), que contiene todos los elementos entre el primero y el último, incluido el elemento señalado por el primero pero no el elemento señalado por el último.
- **val:** Valor del subrango a buscar en el rango. Para (1), T será un tipo que admita la comparación con elementos del rango [primero, último) como cualquiera de los operandos del operador $<$.
- **comp:** Función binaria que acepta dos argumentos del tipo señalado por ForwardIterator (y de tipo T), y devuelve un valor convertible a bool. El valor devuelto indica si se considera que el primer argumento va antes del segundo. La función no modificará ninguno de sus argumentos. Puede ser un puntero de función o un objeto de función.

```
#include <iostream>      // std::cout
#include <algorithm>     // std::equal_range, std::sort
#include <vector>        // std::vector

bool mygreater (int i,int j) { return (i>j); }

int main () {
   int myints[] = {10,20,30,30,20,10,10,20};
   std::vector<int> v(myints,myints+8); // 10 20 30 30 20 10 10 20
   std::pair<std::vector<int>::iterator,std::vector<int>::iterator> bounds;

   // usando comparacion predeterminada:
   std::sort (v.begin(), v.end());                    // 10 10 10 20 20 20 30 30
   bounds=std::equal_range (v.begin(), v.end(), 20) //          ^        ^

   // usando "mygreater" como comp:
   std::sort (v.begin(), v.end(), mygreater);                    // 30 30 20 20 20
        10 10 10
   bounds=std::equal_range (v.begin(), v.end(), 20, mygreater); //       ^
             ^

   std::cout << "limites en posiciones " << (bounds.first - v.begin());
   std::cout << " y " << (bounds.second - v.begin()) << '\n';

   return 0;
}
```

4.2. Java

```java
/* Busqueda de un elemento en un arreglo
* Retorna -1 si el elemento no esta si no la posicion del elemento*/

int binarySearch(int arr[], int x) {
   int l = 0, r = arr.length - 1;
   while (l <= r) {
      int m = l + (r - l) / 2;
      if (arr[m] == x) {
         return m;
      }

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

4.2.1. Funciones de Java relacionados con la búsqueda binaria

1. **Arrays.binarySearch():** El método busca en la matriz especificada del tipo de datos dado el valor especificado utilizando el algoritmo de búsqueda binaria. La matriz debe ordenarse según el método Arrays.sort() antes de realizar esta llamada. Si no está ordenado, los resultados no están definidos. Si la matriz contiene varios elementos con el valor especificado, no hay garantía de cuál se encontrará.

   Parámetros:

   - La matriz a buscar
   - El valor a buscar

   Retorna el índice de la clave de búsqueda, si está contenida en la matriz; de lo contrario, (-1). El punto de inserción se define como el punto en el que la clave se insertaría en la matriz: el índice del primer elemento mayor que la clave, o a.length si todos los elementos de la matriz son menores que la clave especificada. Tenga en cuenta que esto garantiza que el valor de retorno será >= 0 si y sólo si se encuentra la clave. Hay ciertos puntos importantes a tener en cuenta, como sigue:

   - Si la lista de entrada no está ordenada, los resultados no están definidos.
   - Si hay duplicados, no hay garantía de cuál se encontrará.

```java
public class Main {
   public static void main(String[] args){
      int arr[] = { 10, 20, 15, 22, 35 };
      Arrays.sort(arr);
```

```
        int key = 22;
        int res = Arrays.binarySearch(arr, key);

        if (res >= 0) System.out.println( key + " encontrada en la posicion = " +
            res);
        else System.out.println(key + " No encontrada");

        key = 40;
        res = Arrays.binarySearch(arr, key);
        if (res >= 0) System.out.println( key + " encontrada en el indice = " + res
            );
        else System.out.println(key + " No encontrada");
          }
}
```

2. **Collections.binarySearch():**

   Parámetros:

   - La colección a buscar
   - El valor a buscar

   Retorna el índice de la clave de búsqueda, si está contenida en la colección; de lo contrario, (-1). El punto de inserción se define como el punto en el que la clave se insertaría en la matriz: el índice del primer elemento mayor que la clave, o a.length si todos los elementos de la matriz son menores que la clave especificada. Tenga en cuenta que esto garantiza que el valor de retorno será >= 0 si y sólo si se encuentra la clave. Hay ciertos puntos importantes a tener en cuenta, como sigue:

   - Si la lista de entrada no está ordenada, los resultados no están definidos.
   - Si hay duplicados, no hay garantía de cuál se encontrará.

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

public class Main {
   public static void main(String[] args){
      List<Integer> al = new ArrayList<Integer>();
      al.add(100); al.add(50);
      al.add(30); al.add(10);
      al.add(2);

      // El ultimo parametro especifica el comparador
      // metodo utilizado para ordenar.
      int index = Collections.binarySearch(al, 50, Collections.reverseOrder());
      System.out.println("Encontrado en la posicion " + index);
   }
}
```

Buscando en una lista de objetos de clase definidos por el usuario:

```
import java.util.*;

class Main {
   public static void main(String[] args) {
      List<Domain> l = new ArrayList<Domain>();
      l.add(new Domain(10, "www.google.com"));
      l.add(new Domain(20, "eva.umcc.cu"));
      l.add(new Domain(30, "youtube.com"));
      l.add(new Domain(40, "dmoj.uclv.edu.cu"));

      Comparator<Domain> c = new Comparator<Domain>() {
         public int compare(Domain u1, Domain u2){
            return u1.getId().compareTo(u2.getId());
         }
      };

      // Buscando un dominio con valor clave 10. Para buscar
      // creamos un objeto de dominio con clave 10.
      int index = Collections.binarySearch(l, new Domain(10, null), c);
      System.out.println("Encontrado en el indice " + index);

      // Buscando un articulo con la clave 5
      index = Collections.binarySearch(l, new Domain(5, null), c);
      System.out.println(index);
   }
}

class Domain {
   private int id;
   private String url;

   // Constructor
   public Domain(int id, String url) {
      this.id = id;
      this.url = url;
   }
   public Integer getId() { return Integer.valueOf(id); }
}
```

5. Aplicaciones

La búsqueda binaria es un algoritmo eficiente para encontrar un elemento específico en una lista ordenada. Algunas aplicaciones comunes de la búsqueda binaria incluyen:

1. **Búsqueda de elementos en bases de datos:** La búsqueda binaria se utiliza frecuentemente en bases de datos para buscar registros de manera eficiente.
2. **Búsqueda en árboles binarios de búsqueda:** La búsqueda binaria se utiliza en árboles binarios de búsqueda para encontrar un elemento específico de manera rápida.

3. **Búsqueda en listas ordenadas:** La búsqueda binaria es útil para buscar elementos en listas ordenadas, ya que reduce significativamente el tiempo de búsqueda en comparación con la búsqueda lineal.
4. **Búsqueda en algoritmos de ordenamiento:** La búsqueda binaria se utiliza a menudo en algoritmos de ordenamiento como el algoritmo de ordenamiento rápido (QuickSort) y el algoritmo de ordenamiento por mezcla (MergeSort).

En resumen, la búsqueda binaria es una técnica poderosa y eficiente que se utiliza en una variedad de aplicaciones para encontrar elementos específicos en conjuntos de datos ordenados.

6. Complejidad

Veamos en el peor caso, es decir, que se descartaron varias veces partes de la coleción para finalmente llegar a una coleción vacía y porque el valor buscado no se encontraba en la coleción.

En cada paso la coleción se divide por la mitad y se desecha una de esas mitades, y en cada paso se hace una comparación con el valor buscado. Por lo tanto, la cantidad de comparaciones que hacen con el valor buscado es aproximadamente igual a la cantidad de pasos necesarios para llegar a un segmento de tamaño 1. Veamos el caso más sencillo para razonar, y supongamos que la longitud de la coleción es una potencia de 2.

Por lo tanto este programa hace aproximadamente k comparaciones con el valor buscado cuando la cantidad de elementos de la coleción es 2^k. Pero si despejamos k de la ecuación anterior, podemos ver que este programa realiza aproximadamente $\log(N)$ de comparaciones donde N es la cantidad de elementos de la coleción.

Cuando la cantidad de elementos de la coleción no es una potencia de 2 el razonamiento es menos prolijo, pero también vale que este programa realiza aproximadamente $\log(N)$ comparaciones.

Vemos entonces que si la coleción esta ordenada, la búsqueda binaria es muchísimo más eficiente que la búsqueda lineal.

7. Ejercicios

A continuación una lista de ejercicios que se resuelven utilizando esta técnica:

- DMOJ -Repartiendo Caramelos.
- DMOJ - Gasto Mensual.
- MOG - Fito en el Sahara.
- Codeforces - Vanya and Lanterns.
- Codeforces - T-primes
- LeetCode - Find First and Last Position of Element in Sorted Array

- LeetCode - Search Insert Position
- LeetCode - First Bad Version
- LeetCode - Valid Perfect Square
- LeetCode - Find Peak Element
- LeetCode - Search in Rotated Sorted Array
- LeetCode - Find Right Interval
- Codeforces - Interesting Drink
- Codeforces - Magic Powder - 1
- Codeforces - Another Problem on Strings
- Codeforces - Frodo and pillows
- Codeforces - GukiZ hates Boxes
- Codeforces - Enduring Exodus
- Codeforces - Chip 'n Dale Rescue Rangers