# GUÍA DE APRENDIZAJE PARA CONCURSANTES ICPC Y IOI: BÚSQUEDA BINARIA

## 1. Introducción

En muchos ejercicios parte del algoritmo solución o el mismo algoritmo solución radica en saber si un elemento esta dentro de los valores almacenados en una estructura de datos. Para solucionar lo anterior en muchos casos realizamos lo que comúnmente se llama una búsqueda lineal.

La búsqueda lineal radica en recorrer todos los elementos de la coleción en busca de la existencia o no de un determinado valor. Dicho algoritmo en el peor de los casos puede tener una complejidad de $O(n)$ ya que bien el elemento bien no pudiera estar en la colección o estar en la última posición que se visite en el recorrido.

Ahora si antes de buscar un elemento dentro de una coleción conocemos de antemano dicha que coleción sus valores están ordenados. Podría esto ayudar a mejorar el tiempo de búsqueda ? .

## 2. Conocimientos previos

**Arreglos** Es un tipo de datos estructurado que está formado de una colección finita y ordenada de datos del mismo tipo. Es la estructura natural para modelar listas de elementos iguales. Están formados por un conjunto de elementos de un mismo tipo de datos que se almacenan bajo un mismo nombre, y se diferencian por la posición que tiene cada elemento dentro del arreglo de datos. Al declarar un arreglo, se debe inicializar sus elementos antes de utilizarlos. Para declarar un arreglo tiene que indicar su tipo, un nombre único y la cantidad de elementos que va a contener. Accediendo a su indice podemos acceder a cada elemento del arreglo, conociendo que el primer elemento tiene indice 0. En muchas ocasiones para buscar un elemento que desconocemos su indice debemos hacer uso de bucles.

## 3. Desarrollo

Trataremos de aprovechar el hecho de que la colección está ordenada y vamos a hacer algo distinto: nuestro espacio de búsqueda se irá achicando a segmentos cada vez menores de la colección original. La idea es descartar segmentos de la lista donde el valor seguro que no puede estar:

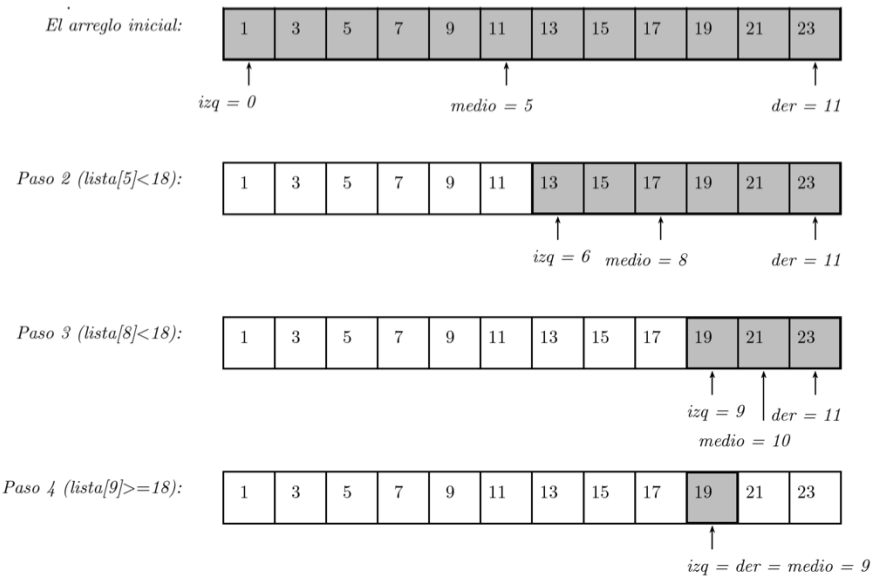
1. Consideramos como segmento inicial de búsqueda a la lista completa.
2. Analizamos el punto medio del segmento (el valor central), si es el valor buscado, devolvemos el índice del punto medio.
3. Si el valor central es mayor al buscado, podemos descartar el segmento que está desde el punto medio hacia la a derecha.
4. Si el valor central es menor al buscado, podemos descartar el segmento que está desde el punto medio hacia la izquierda.
5. Una vez descartado el segmento que no nos interesa, volvemos a analizar el segmento restante, de la misma forma.

6. Si en algún momento el segmento a analizar tiene longitud 0 o negativa significa que el valor buscado no se encuentra en la coleción.

Este algoritmo es un gran ejemplo de una estrategia de dividir y conquistar. Dividir y conquistar significa que dividimos el problema en partes más pequeñas, resolvemos dichas partes más pequeñas de alguna manera y luego reensamblamos todo el problema para obtener el resultado. Cuando realizamos una búsqueda binaria en una colección, primero verificamos el ítem central. Si el ítem que estamos buscando es menor que el ítem central, podemos simplemente realizar una búsqueda binaria en la mitad izquierda de la colección original. Del mismo modo, si el ítem es mayor, podemos realizar una búsqueda binaria en la mitad derecha.

Para señalar la porción del segmento que se está analizando a cada paso, utilizaremos dos variables (izq y der) que contienen la posición de inicio y la posición de fin del segmento que se está considerando. De la misma manera usaremos la varible medio para contener la posición del punto medio del segmento.

En el gráfico que se incluye a continuación, vemos qué pasa cuando se busca el valor 18 en la lista [1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23].




# 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.

## 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;
}
```

### 4.2. 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;
}

/*las clases Arrays y Colections poseen el metodo busqueda binaria
* para buscar un elemento bien sea dentro de un arreglo o coleccion,
* en ambos caso devuelve la posicion donde se encuentra el elemento
* buscado. En caso de no encontrarse el valor devuelto es -1. El arreglo
* o colecion debe esta ordenado de forma ascendente  previamente */
```

## 5. 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.

## 6. Aplicaciones

La aplicación de está técnica se nota que es la de buscar un elemento dentro de una coleción de la cual tenemos que tener como precondición que **los elementos están ordenados** si lo anterior no se cumpliera no podríamos aplicar esta técnica. Ahora existen otros contextos donde pueden participar como parte de la solución.

1. La búsqueda del elemento dentro de una coleción ordenada.
2. La búsqueda de un valor entero dentro de un intervalo (a veces puede ser el mínimo/máximo o único) que cumple con una determinada condición en dicho intervalo. En este caso los números enteros que componen el intervalo actúan como los valores de la coleción ordenada.

## 7. Ejercicios propuestos

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