# Mayorante y Minorante

**Binary Search**

El tema del texto como tal se deriva del **Binary Search** y es por eso que empezaremos con este con una breve definición.

Definición

Es un algoritmo de búsqueda que encuentra la posición de un valor específico que encuentra la posición de un valor específico en un arreglo ordenado. En breves palabras este algoritmo nos permite encontrar valores en un conjunto de datos ordenados, lo interesante de esto es la eficiencia con la que lo hace.

Funcionamiento

Lo que hace este algoritmo es comparar el valor determinado con el elemento intermedio del arreglo, si este no es igual al valor que buscamos entonces pregunta si es que es menor y entonces se mueve a una de las mitades y vuelve a hacer la pregunta con esta mitad, de igual forma si es que es mayor. A partir de aquí podemos empezar a intuir la importancia de que el arreglo esté ordenado, ya que al estarlo podemos descartar una de las mitades con facilidad.

Complejidad y un poco más

Este algoritmo en el peor de los casos tiene una complejidad de **O(log n)** donde **n** es el elemento de elementos en un arreglo, lo cual es una gran mejora a la complejidad de una búsqueda lineal, excepto para arreglos muy pequeños

Implementación

En este punto existen dos posibilidades que hagas todo la implementación o que uses la función ya implementada en la biblioteca de **algorithm**, empecemos con la segunda para fines prácticos.

- Implementación en la biblioteca **algorithm**.

  Nuestro primer paso será ordenar un arreglo o vector, para fines didácticos sólo mostraremos el proceso con un vector,

con la ya conocida función **sort** que también es parte de esta biblioteca.

```
int miArreglo[n];

sort(miArreglo, miArreglo+n);
```

Después sólo tienes que llamar a la función **binary_search()**, que al ser una función del tipo **bool** nos devuelve **true** si encuentra el valor buscado y **false** en caso de que no.

```
binary_search(miArreglo, miArreglo+n, valor);
```

- Implementación propia

Para hacer una implementación propia se necesita un poco más de destreza, sin embargo tampoco es imposible. Cabe recalcar que para hacer esta implementación tenemos dos opciones, la primera es de forma recursiva y la segunda iterativa, esta última es la más recomendable por el tiempo de ejecución que para nosotros es importantísima.

```
int miArreglo[n];

sort(miArrelgo, miArreglo+n);
```

En esta primera parte repetimos un poco del proceso anterior, al declarar nuestro conjunto y después lo ordenamos para poder llevar a cabo el algoritmo.

```
int BusquedaBinaria(int valor) {
    int izq=0;
    int der=n;
    int mitad;
    while(izq+1<der) {
      mitad=(izq+der)/2;
      if(miArreglo[mitad]<valor) {
            izq=mitad;
      }else if(miArreglo[mitad]=>valor) {
            der=mitad;
      }
    }
    return izq+1;
}
```

Aquí en el códifo mismo podemos ver el algoritmo de **Binary Search** en acción, para esto hacemos uso de un ciclo que nos permite hacer las iteraciones necesarias hasta antes de que las variables que apuntan a la mitad izquierda y derecha se toquen.

**Mayorante (Upper Bound)**

Cuando estamos buscando el **mayorante** en realidad estamos buscando en un conjunto ordenado el primer elemento cuyo valor es mayor a un valor dado.

Implementación

La función que lleva a cabo la tarea también es parte de la biblioteca **algorithm**, de hecho hace uso del **Binary Search** para encontrar el valor en nuestro arreglo, por tanto primero tenemos que ordenar nuestro arreglo y después mandamos a llamar a nuestra función.

```
int miArreglo[n];
sort(miArreglo, miArreglo+n);
upper_bound(miArreglo, miArreglo+n, valor);
```

**Minorante (Lower Bound)**

Como contraparte de lo anterior al buscar el **minorante** estamos buscando en un conjunto ordenado el primer elemento cuyo valor no es menor o igual a un valor dado.

Implementación

La función que lleva a cabo la tarea también es parte de la biblioteca **algorithm**, de hecho hace uso del **Binary Search** para encontrar el valor en nuestro arreglo, por tanto primero tenemos que ordenar nuestro arreglo y después mandamos a llamar a nuestra función.

```
int miArreglo[n];
sort(miArreglo, miArreglo+n);
lower_bound(miArreglo, miArreglo+n, valor);
```