

**Peor caso: Se repite n veces.
Dato no se encuentra en el arreglo.**

```
int Lineal(int A[], int n, int dato)
```


```
{
```

```
    int posicion = 0;  O(1)
```

```
    while(posicion < n)
```

```
    {
```

```
        if(dato == A[posicion])
```

```
            return posicion;  O(1)
```

```
        posicion++;  O(1)
```

```
    }
```

```
    return -1;
```

```
}
```

O(1)

O(n)

```

int Binaria(int A[], int n, int dato)
{
    //centro: subíndice central del intervalo
    //inf: límite inferior del intervalo
    //sup: límite superior del intervalo
    int centro, inf = 0, sup = n-1;
    while(inf <= sup)
    {
        centro = ((sup + inf)/2);
        if(A[centro] == dato)
        {
            /*Para imprimir la posicion y el dato dentro del arreglo
            printf("SI %d : %d", A[centro], centro);*/
            return centro;
        }
        else if (dato < A[centro])
            sup = centro - 1;
        else
            inf = centro + 1;
    }
    return -1;
}

```

Peor caso: El dato no esta en el arreglo y se recorre completo. Se repite n veces

$O(1)$

$O(1)$

$O(1)$

$O(1)$

$O(1)$

$O(1)$

$O(\log_2 n)$

```

int ABB(Arbin *a, int elemento)
{
    posicion a_aux = *a;
    int numero; // Auxiliar para comparar
    do
    {
        numero = a_aux -> raiz;
        if(numero == elemento)
        {
            a_aux = NULL;
            return 0;
        }
        else if(numero < elemento)
        {
            a_aux = a_aux -> der;
        }
        else
        {
            a_aux = a_aux -> izq;
        }
    }
    while (a_aux != NULL); // Apuntador nulo
    return -1;
}

```

Con un árbol binario completamente lleno, tendrá $2^{(n-1)}$ elementos. El orden de complejidad ideal con números desordenados será de $O(\log_2 n)$

$O(1)$

$O(1)$

$O(1)$

$O(1)$

$O(1)$

$O(1)$

$O(1)$

El peor de los casos es que el número no se encuentre y recorra el árbol hasta el último nodo.

$O(1)$