Binary Search

The simplest method for searching an array for a specific value (called the search value), is the **Linear search**. If the content of the array is unsorted it is the **only** method that can be used, and the potential exists that the search value may need to be compared to every element in the array before we can determine whether or not it is present in the array. **If we are searching for a value that is not there,** then we won't know for sure until the search value is compared to the last element. If the array is sorted in ascending order, then we can stop performing comparisons once a value is found that is **greater** than the search value, somewhat improving the algorithm. But still in the worst case (the search value > value stored in the last element), then we would still need to visit all elements in the array.

The **binary search** algorithm is a far better method for searching that takes advantage of the fact that the array is **sorted**. It uses a **divide and conquer strategy**, where the size of the array being searched is divided in **half** with each comparison. It can be implemented using either iteration or **recursion**. Suppose that we have an array of **N** integers sorted in ascending order.

3	5	7	9	10	12	17	20	24	29	31	37	44	50	62	71	72	80	93	100
1																			**N**
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

Suppose the value we seek is 80. We call the method with the index number of the first and last elements in the **range** of elements we are still considering. We begin our search at the middle element of the array which is **(first+last)/2**. In our example N=19, we calculate (19+0)/2 = 9. We compare our search value to the value stored in the **9th** element of the array which is **29.**

29 is less than 80. This means that the value we seek must like between elements **10** and **19**. So we need only concern ourselves with the **upper** portion of the array, and we set "first" to **mid index +1** or **10** in this case

31	**37**	**44**	**50**	**62**	**71**	**72**	**80**	**93**	**100**
10	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**

Now we determine the middle element of this portion of the array: **(10+19)/2 = 14**

Element **14** contains the value **62** which again is less that **80**, so element **15** becomes the "lower bounds" of the elements that might contain our value.

71	**72**	**80**	**93**	**100**
15	**16**	**17**	**18**	**19**

We determine the middle again which is **(15 + 19)/2** which is 17.. and the value in element **17** is **80**, we have found the desired value and can return 17 as the result of our search.

If the value we are searching for is not in the array, then eventually we will end up looking at a portion of the array that consists of a **single element**, if it does not match our search value we know that the value is not in the array and we can simply return a value (**-1**) indicating that the value was not found, also we have to insure that the index of the "first" element is never greater than the index of the "last"

It is plain that the binary search algorithm yields a significant improvement over the linear search. A linear search would have performed 18 comparisons before finding our value, while the binary search performed only 3.

So how can we implement this algorithm in code? Consider the following, assume the array is named **arr,** and our search value is contained in a variable named **val:**

1) With each comparison, we are looking at a range of elements in the array. We can indicate these elements by specifying a **low** and **high** index value that indicates the first and last element in the range being considered. In the first search, **low**=0 and **high** = **N-1**, since we are considering the entire array.
2) We compute the middle index as **mid = (low + high)/2**
3) if **val = arr[mid]** then return **mid**
4) **if low = high or low > high** return -1.. the value is not found
5) if **val < arr[mid]** our search value must be to the left of the middle element, so set **high = mid -1 ,** and search again
6) if **val > arr[mid]** our search value must be to the right of the middle element, so set **low = mid +1,** and search again

Iterative Binary Search

The binary search algorithm can be implemented using iteration (looping) instead of recursion:

```
 public static int bsearch(int[] arr, int searchVal) {

        int result=-1, mid;
        int low=0, high = arr.length-1;


        while (low <= high) {

        mid = (low + high) / 2;

        if (arr[mid] == searchVal) {
                result = mid;
                break;
        }

        else if ( searchVal < arr[mid] )
                high = mid-1;

        else  low = mid + 1;

        } // end while

        return result;
}// end bsearch
```

It should also be obvious that the binary search algorithm can be easily implemented as a recursive method, but we need more information. We need the array to be searched, the search key, and the range of elements to consider in each execution of the search method.