Search Algorithms
(Binary Search)

Logically, to find a value in an unsorted array, we could look through elements of an array one by one, until a target value is found. This, however, can take a lot of CPU time, plus if the target value is absent from the array, we'll have wasted a lot of processing to check all elements. On average, the complexity of such an algorithm is proportional to the length of the array.

The situation changes significantly when the array is sorted. Random access can be utilized very efficiently to find a target value. The cost of a search algorithm is reduced to the binary logarithm of the array length. For example, in an array of a million elements, the number of searches = $\log_2(1{,}000{,}000) \approx 20$. It means that ***in the worst case***, the algorithm makes 20 steps to find a value in a sorted array of a million elements.

Binary Search Algorithm

The algorithm is quite simple. It can be done either ***recursively*** or ***iteratively***:

1. get the middle element;
2. if the middle element equals to the target value, the algorithm stops;
3. otherwise, two cases are possible:
    - target value is less than the middle element. In this case, go to the step 1 for the portion of the array that ends before the middle element.
    - target value is greater than the middle element. In this case, go to the step 1 for the portion of the array that begins after the middle element.

We should also define when the iterations should stop.
The first case is when searched element is found.
The second case is when the array has no elements. In this case, we can conclude that the target value isn't found in the array.

Example 1

Find 6 in {-1, 5, 6, 18, 19, 25, 46, 78, 102, 114}.

Step 1 (middle element is 19 > 6): search **{ -1 5 6 18 }** 19 25 46 78 102, 114

Step 2 (middle element is 5 < 6): search -1 5 **{ 6 18 }** 19 25 46 78 102, 114

Step 3 (middle element is 6 == 6): found -1 5 **{ 6 18 }** 19 25 46 78 102, 114

Example 2

Find 103 in {-1, 5, 6, 18, 19, 25, 46, 78, 102, 114}.

Step 1 (middle element is 19 < 103): search -1 5 6 18 19 **{ 25 46 78 102 114 }**

Step 2 (middle element is 78 < 103): search -1 5 6 18 19 25 46 78 **{ 102 114 }**

Step 3 (middle element is 102 < 103): search -1 5 6 18 19 25 46 78 102 **{ 114 }**

Step 4 (middle element is 114 > 103): search -1 5 6 18 19 25 46 78 102 **{ }** 114

Step 5 (search array is empty): target value is absent

Complexity analysis

The huge advantage of this algorithm is that its complexity depends on the array size logarithmically ***in the worst case*****.** In practice this means that the algorithm will execute at most $\log_2(n)$ iterations, which is a very small number even for big arrays.

This can be proven very easily, where every step of the algorithm reduces the size of the search array by half. The algorithm stops when there are no elements in the search array.

$$n / 2^{iterations} > 0$$

resulting in

$$iterations <= \log_2(n).$$

This means that the binary search algorithm time complexity is $O(\log_2(n))$.

The 'O' stands for Big-Oh.

The following method is an implementation of a binary search. This would be called from your mainline code.
This algorithm assumes that your array has ***already been sorted***.
The method returns the position of the target in the array, or a -1 if target not found…

The Iterative Algorithm

```
/*
* NOTE: This is a method, not a class.
*
* Functional Description:
*   searches for a value in a sorted array
* @Parameters:
*   arr is the array to search
*   target is search target value
*   left is the index of left boundary of array
*   right is the index of right boundary of array
* returns array index of target,
*      or -1 if target not found.
*/
int binarySearch ( int arr[], int target, int left, int right ) {
      while (left <= right) {
            int middle = Math.floor((left + right) / 2);
            if (arr[middle] == value)
                  return middle;
            else if (arr[middle] > value)
                  right = middle - 1;
            else
                  left = middle + 1;
      }
      return -1;
}
```

Sample call:

```
int[] intSortArray = new int[] {5, 23, 45, 48, 89, 80, 81, 89, 102};
int target = 45;
int result = -1;
result =  binarySearch( intSortArray, target, 0, intSortArray.length-1 ));
if ( result != -1 )
      System.out.println ( "Target found in cell " + result );
else
      System.out.println ( "Target not found.” );
```

The Recursive Algorithm

```
/**
 * @param array
 *      array to search in
 * @param value
 *      target value
 * @param left
 *      index of left boundary
 * @param right
 *      index of right boundary
 * @return position of searched value,
 *      if present in the array, or -1 if it is absent
 */
int binarySearch(int[] array, int value, int left, int right) {
      if (left > right)
            return -1;
      int middle = (left + right) / 2;
      if (array[middle] == value)
            return middle;
      else if (array[middle] > value)

            // this is the recursive call
            return binarySearch(array, value, left, middle - 1);
      else
            return binarySearch(array, value, middle +1, right);
}
```