Binary Search / Dictionary Search

When trying to find an item in a sequence, a natural first approach is to look at the items successively and check whether it is the one wanted. This strategy is simple, but would you use this strategy if asked to find a name in NYC's phone book? Your answer should be "OF COURSE NOT!" There is something that makes searching a phone book, a dictionary, or an old-fashioned library catalog easy: the entries are *sorted*!

A common strategy to search for an element in an *ordered* array a of length n,

$$a = \{a[0], a[1], a[2], \ldots, a[n-1]\},$$

is called Binary Search (a.k.a. lexicographic or dictionary search). This search algorithm maintains a search window $[L, R]$ such that the target value z is within that window, i.e., $a[L] \leq z < a[R]$. This search window is repeatedly *halved*, hence the name *binary* search. Note the use of a strict inequality ($<$ instead of $\leq$) in the latter part of the compound expression $a[L] \leq z < a[R]$.

The algorithm begins with setting L and R so that a[L] $\leq z <$ a[R] is true. Then at each iteration the "middle" index of the search window is calculated: $m = \frac{L+R}{2}$. [1] So now $a[m] \leq z$ or $a[m] > z$. If $a[m] \leq z$ then the next search window is $[m, R]$, otherwise the next search window is updated to be $[L, m]$. You can see that eventually the search window will be small ($R = L+1$) and *it will bracket the target value* z, even if z itself is not in array a. So the binary search algorithm doesn't just reveal whether a target is in a sorted array; it actually indicates where the target could be inserted! Since the algorithm maintains a search window such that $a[L] \leq z < a[R]$, the single returned value of a binary search is really L, the index that marks the *low* end of the search window when the search stops.

Now think about how to set the initial search window $[L, R]$ so that the compound relation a[L] $\leq z < a[R]$ is true. Since we cannot guarantee that z is in array a, we need to set the initial search window to include the possibilies for $z < a[0]$ and $z > a[n-1]$. This means that initially we set L to be smaller than the lowest possible index and R to be bigger than the highest possible index! The values -1 and n are not valid indices for an array of length n, but we use them to indicate the correct search window. When you write and analyze the code, you will see that (or make sure that) the code never actually tries to access $a[-1]$ or $a[n]$.

Below are some examples for an array a of length 8, $a =$ { -4, -2, 0, 5, 5, 5, 8, 9 }:

Target z	Returned value	Note
-2	1	z found at position 1
5	5	z appears at positions 3, 4, 5, so any one of these indices may be returned.
-4	0	z found at first position, position 0
9	7	z found at last position, position 7
4	2	The smallest possible window to bracket z is [2,3]. In other words, $a[2] < z < a[3]$. Your method should print a message saying value not found.
-6	-1	Of course -1 isn't a valid index. This is a signal that $z < a[0]$. Your method should print a message saying value not found.
11	7	z isn't found and $a[7] < z$. Your method should print a message saying value not found.

[1] Remember than in Java $W/2$ will be an integer if W is an integer.

Questions:

How many iterations would it take to find a value in an array of length 1000? _____

How many iterations would it take for an array of length 10000000 (10 million)? _____

Write the method `binarySearch` to perform a binary search as explained above.

```
/* Binary search:
 * =Return the value i such that a[i]<=z<a[i+1], array a is sorted in non-decreasing order.
 * If z is not in array a, a warning message will display and the returned value indicates
 *   the position after which z may be inserted in the sorted array a.  A return value -1
 *   is not a valid index but indicates that z<a[0].
 */
public static int binarySearch(int[] a, int z) {

  //Your code here.  It doesn't take the whole page  :-)




}
```