Lesson 39: Binary Search

Consider the following array of ordered integers. It is very important that they **be in order** so that the techniques we are to discuss will work.

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Value	-7	15	21	22	43	49	51	67	78	81	84	89	95	97

Our task is to examine this array and see if 22 is present, and if it is, report its index. If it is not present, report an "index" of –1. Of course, we can instantly tell at a glance that 22 is, indeed, present, and that its index is 3. That's easy for us as humans, but is it this straightforward for a computer program? (Actually, it's not easy for humans either if the size of the array is several thousand or, perish the thought, in the millions.)

The binary search technique looks at a range of index numbers that is determined by a lower bound (*lb*) and an upper bound (*ub*), subdivides that range in halves, and then continues the search in the appropriate half. We illustrate this by initially setting *lb* = 0 and *ub* =13 and realizing that the number we seek lies somewhere on or between them (the shaded region).

	lb													ub
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Value	-7	15	21	22	43	49	51	67	78	81	84	89	95	97

Fig 50-2 *lb* and *ub* are initially set.

Find the midpoint *m* using integer arithmetic, 6 = (lb + ub)/2. This position is illustrated below.

	lb						m							ub
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Value	-7	15	21	22	43	49	51	67	78	81	84	89	95	97

Fig 50-3 Midpoint *m* is determined.

Next, we ask if the number we seek (22) is at position *m*. In this case it is not, so we next ask if 22 is less than or greater than the value at position *m*. 22 is, of course, less that 51 so we indicate the new search area by the shaded area below and redefine *ub* = 5.

	lb					ub								
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Value	-7	15	21	22	43	49	51	67	78	81	84	89	95	97

Fig 50-4 A new *ub* and search region are defined.

Again, we calculate m = (lb + ub) / 2 using integer arithmetic and get 2. This new *m* value is shown in Fig. 50-5.

	lb		m			ub								
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Value	-7	15	21	22	43	49	51	67	78	81	84	89	95	97

Fig 50-5 A new *m* value of 2 has been determined as the midpoint.

Repeat what we have done before by determining if the number we seek (22) is equal to the value at position *m* (21). It is not, so as before, we ask if 22 is less than or greater than 21. Of course, it's greater than 21, so we now redefine *lb* = 3 and have the new search area shown in Fig 50-6.

				lb		ub								
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Value	-7	15	21	22	43	49	51	67	78	81	84	89	95	97

Fig 50-6 A new, smaller search area defined by *lb* and *ub*.

Are you beginning to see a pattern here? We are cutting our search area in half each time. At this rate we could examine an array of over a million integers with only 20 iterations of this process.

Let's average *lb* and *ub* as before, and this time *m* is 4. The result is shown in Fig 50-7 below.

				lb	m	ub								
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Value	-7	15	21	22	43	49	51	67	78	81	84	89	95	97

Fig 50-7 Ready to test if the value stored at index 4 is what we seek.

Ask if the 22 (the value we seek) is stored at index *m* = 4. It is not, so now ask if 22 is less than or greater than 43. It's less than 43 so we redefine *ub* as 3. Notice now that *lb* and *ub* are the same and we have just about "squeezed" the search area down to nothing. The redefined boundaries and the resulting recalculation of *m* are shown below. If we don't find the number we seek in this single cell that's left, the number is not in the array, and we would have to report a "failure" value of –1 as a result of our search.

				lb = m = ub										
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Value	-7	15	21	22	43	49	51	67	78	81	84	89	95	97

Fig 50-8 The lower bound, upper bound and the midpoint all coincide.

Now, when we ask if 22 (the number we seek) is at index 3, the answer is, "yes", so we exit this process and report index 3 as the answer.

Ending the process:

We will code this entire process shortly, however, there is something worth noticing here. As we squeezed down to ever-smaller ranges, the indices *lb* and *ub* move closer to each other. If the number we seek is not in the array, we will find that the algorithm actually moves these two value "right past each other". Therefore we can use the condition *ub* < *lb* as a condition to exit the loop, or… *ub* >= *lb* to stay in the loop.

Implementing our own binary search:

Our first illustration of code implementing the binary search will be to search the integer array {-7, 15, 21, 22, 43, 49, 51, 67, 78, 81, 84, 89, 95, 97}. Notice that this is the array

used in the example above. We will continue the tradition of using the variables *lb* and *ub*; however, instead of *m* we will use *mid*. The student should be aware that when looking at binary searches written by other programmers, other variables are often used in the place of our *lb* and *ub*. Popular choices are *left* & *right*, *front*& *back*, or *start*& *end*.

The following class provides a *main* method for testing and a *binarySearch* method where the search is actually done.

```
public class Tester
{
	public static void main(String args[])
	{
		int i[] ={-7, 15, 21, 22, 43, 49, 51, 67, 78, 81, 84, 89, 95, 97};
		System.out.println(binarySearch(i, 22)); //prints 3
		System.out.println(binarySearch(i, 89)); //prints 11
		System.out.println(binarySearch(i, -100)); //prints -1
		System.out.println(binarySearch(i, 72)); //prints -1
		System.out.println(binarySearch(i, 102)); //prints -1
	}

	//Look for srchVal in the a[] array and return the index of where it's found
	//Return –1 if not found

	private static int binarySearch(int a[], int srchVal)
	{
		int lb = 0;
		int ub = a.length - 1;
		while(lb <= ub)
		{
			int mid = (lb + ub)/2;
			if(a[mid] = = srchVal)
			{
				return mid;
			}
			else if (srchVal > a[mid])
			{
				lb = mid + 1; //set a new lowerbound
			}
			else
			{
				ub = mid -1; //set a new upper bound
			}
		}
		return -1; //srchVal not found
	}
}
```

Recursive Binary Search:

We offer one last way to do a binary search. The class at the top of the next page uses recursion to implement a binary search. Notice that the *binarySearch* method uses four parameters.

```
public class Tester
{
	public static void main(String args[])
	{
		int i[] ={-7, 15, 21, 22, 43, 49, 51, 67, 78, 81, 84, 89, 95, 97};
		int lb = 0;
		int ub = i.length -1;
		System.out.println(binarySearch(i, 22, lb, ub)); //prints 3
		System.out.println(binarySearch(i, 89, lb, ub)); //prints 11
		System.out.println(binarySearch(i, -100, lb, ub)); //prints -1
		System.out.println(binarySearch(i, 72, lb, ub)); //prints -1
		System.out.println(binarySearch(i, 102, lb, ub)); //prints -1
	}

	private static int binarySearch(int a[], int srchVal, int lb, int ub) //recursive
	{
		if(lb > ub)
		{
			return -1;
		}
		else
		{
			int mid = (lb + ub)/2;
			if(a[mid] = = srchVal)
			{
				return mid;
			}
			else if (srchVal > a[mid])
			{
				return binarySearch(a, srchVal, mid + 1, ub);
			}
			else
			{
				return binarySearch(a, srchVal, lb, mid -1);
			}
		}
	}
}
```

****

Pros and cons:

Quite often we have to ask ourselves if a binary search is really worthwhile, especially when compared to doing a linear search. (A linear search is typically done on an **unordered** array.)

A linear search is done by starting at index 0 of an array and just marching all the way to the end asking at each index, "Is this the value?" With good luck we will get a match on the first index we try, but on the other hand, with bad luck, it will be the very last possible index. Therefore, the Big O value for a linear search is O(n) and it is not very efficient.

Since a binary search is much, much faster on the average than a linear search, we have to ask ourselves if sorting the array (as required before doing a binary search) is worthwhile. If the array is to be searched only once, then a linear search would probably save time in the long run. However, if several searches are to be done, then the time investment in sorting the array would probably pay off, and the binary search would be preferred.

It's all in the *Arrays* class:

We would be remiss if it was not mentioned that the *Arrays* class in the *java.util* package has *sort* and *binarySearch* methods. These have already been discussed in Lesson 19; however, here is a summary of that information:

a. **Sort** the array in ascending order.

```
public static void sort(int a[])          //Signature
```

Example:

```
int b[] = {14, 2, 109, . . . 23, 5, 199};
Arrays.sort(b);                           //the b array is now in ascending order
```

b. Perform a **binary search** of an array for a particular value (this assumes the array has already been sorted in ascending order). This method returns the index of the last array element containing the value of *key*. If *key* is not found, a negative number is returned… *–k –1* where *k* is the index before which the key would be inserted.

```
public int binarySearch(int a[], int key) //Signature
```

Example:

```
//Assume array b[] already exists and has been sorted in ascending order.
//Suppose the b array now reads {2,17, 36, 203, 289, 567, 1000}.
int indx = Arrays.binarySearch(b, 203); //search for 203 in the array
System.out.println(indx);       //3
```

Unfortunately, *Arrays.binarySearch* does not work on any objects other than *String* and

Comparable object arrays. It *does* work on all the primitive data type arrays.

****

Some miscellaneous facts:

Finally, here are a few random facts about binary searches (on an array of *n* elements).

1. Array must already be sorted.
2. Big O value is O(log n) for both iterative and recursive types.
3. $\log_2 n$ is the worst case number of times we must "halve" the list.
4. The largest size array that can be searched while comparing at most *n* elements is 2^n-1.