JAVA NOTES

DATA STRUCTURES AND ALGORITHMS

Terry Marris July 2001

5 BINARY SEARCH

5.1 LEARNING OUTCOMES

By the end of this lesson the student should be able to

- explain the principles of the binary search algorithm
- state the essential pre-requisites for a binary search algorithm
- dry run, implement and test a binary search method

5.2 PRE-REQUISITES

The student should know what is meant by an ordered or sorted array - see §4.

5.3 INTRODUCTION

In §3 we looked at the linear search method. To determine that an item was not in the array we had to look at each element in the array in turn.

We could speed up the search if the items held in the array were sorted into some natural order. Then, to determine that an item was not in the array we need to look at no more than half the elements in the array. For example, think of a number between 1 and 100 inclusive.

1	.. ..	25	.. ..	50	.. ..	75	.. ..	100

Is it less than fifty? If you answered yes I can immediately discard the numbers 50..100 in my attempt to find the number you thought of and concentrate my search on the numbers 1..49.

1	.. ..	25	.. ..	50	.. ..	75	.. ..	100

If you answered no I can immediately discard the numbers 1..49 and concentrate on 50..100.

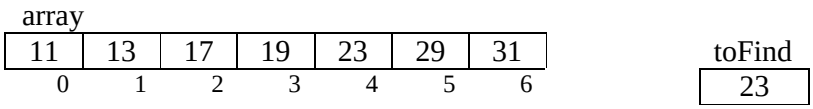
1	.. ..	25	.. ..	50	.. ..	75	.. ..	100

The idea works because we consider the numbers 1..100 in numerical order.

5.4 BINARY SEARCH

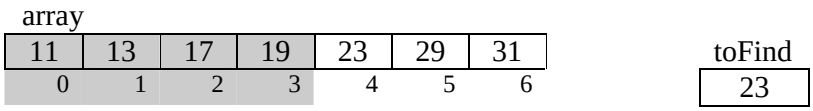
The essence of the binary search method is that we continually eliminate parts of the sorted array that the item we are looking for cannot possibly be in.

We start with the array shown below. Notice that the array is sorted - all its objects are held in ascending numerical order. We want to establish whether 23 is in the array.



Find the middle index:	(0 + 6) / 2 = 3
Is the required item (23) the same as array[middle] (19)?	No
Is the required item less than array[middle]	No
Is the required item more than array[middle]?	Yes

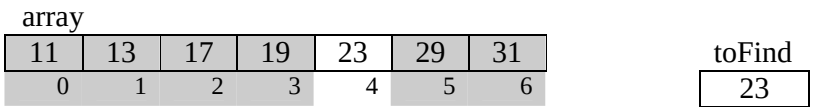
Eliminate left-hand part of array from the search.



Search the right-hand part of the array.

Find the middle index:	(4 + 6)/2 = 5
Is the required item (23) the same as array[middle] (29)?	No
Is the required item less than array[middle]?	Yes

Eliminate right-hand part of the array segment from the search

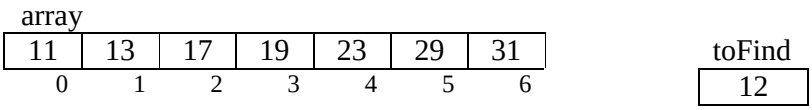


Search the left-hand part of the array.

Find the middle index:	(4 + 4)/2 = 4
Is the required item (23) the same as array[middle] (23)?	Yes

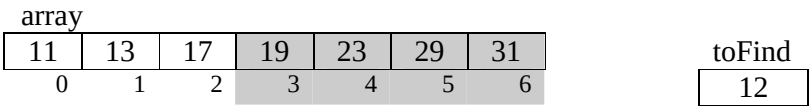
Return success.

We can see with our eyeballs that 12 is not in the array. Does the binary search method described above still work?



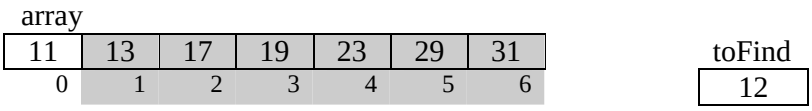
Find the middle index:	(0 + 6) / 2 = 3
Is the required item (12) the same as array[middle] (19)?	No
Is the required item less than array[middle]?	Yes

Eliminate right-hand part of array from the search.



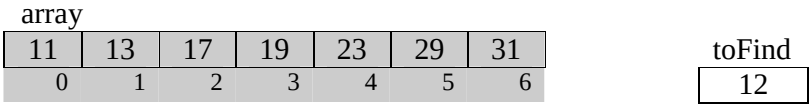
Find the middle index:	(0 + 2) / 2 = 1
Is the required item (12) located at array[middle] (13)?	No
Is the required item less than item at array[middle] ?	Yes

Eliminate right-hand part of array from the search.



Find the middle index:	(0 + 0) / 2 = 0
Is the required item (12) the same as array[middle] (11)?	No
Is the required item less than array[middle]?	No
Is the required item more than array[middle]?	Yes

Eliminate left-hand part of array from the search.



There is no more array left to partition. Therefore we conclude that 12 is not in the array. Return failure.

Here is the code. Each time round the loop, *low* and *high* get closer together, until either they meet and go past each other or the method is exited with a *return success* statement.

```
static String search(Object[] array, Object toFind)
{
  int low = 0;
  int high = array.length - 1;

  while (low <= high) {     // loop until low passes high
    int mid = (low + high) / 2;
    Object midVal = array[mid];
    int cmp = ((Comparable)toFind).
                compareTo((Comparable)midVal);
    if (cmp == 0)           // found it
      return "success";
    else if (cmp < 0)       // toFind < array[middle]
      high = mid - 1;       // eliminate right half
    else if (cmp > 0)       // toFind > array[middle]
      low = mid + 1;        // eliminate left half
  }
  return "failure";         // not found
}
```

The complete class is shown below.

```
/* BinarySearch.java
   Terry Marris  23 July 2001
*/

public class BinarySearch {
  static String search(Object[] array, Object toFind)
  {
    int low = 0;
    int high = array.length - 1;
    while (low <= high) {
      int mid = (low + high) / 2;
      Object midVal = array[mid];
      int cmp = ((Comparable)toFind).
                  compareTo((Comparable)midVal);
      if (cmp == 0)
        return "success";
      else if (cmp < 0)
        high = mid - 1;
      else if (cmp > 0)
        low = mid + 1;
    }
    return "failure";
  }
}
```

5.5 TESTING THE BINARY SEARCH METHOD

We populate an array with seven *Integer*s, taking care to see that they are stored in ascending order. We search for the items at both ends of the array, and for items just less than the first one, and just more than the last one in the array.

```
/* TestBinarySearch.java
   Terry Marris  23 July 2001
*/

public class TestBinarySearch {
  public static void populate(Integer[] array)
  {
    array[0] = new Integer(11);  array[1] = new Integer(13);
    array[2] = new Integer(17);  array[3] = new Integer(19);
    array[4] = new Integer(23);  array[5] = new Integer(29);
    array[6] = new Integer(31);
  }


  public static void main(String[] s)
  {
    Integer[] array = new Integer[7];  // indexed 0..6
    populate(array);
    System.out.println("Searching for 10 ... " +
                BinarySearch.search(array, new Integer(10)));

    System.out.println("Searching for 11 ... " +
                BinarySearch.search(array, new Integer(11)));

    System.out.println("Searching for 31 ... " +
                BinarySearch.search(array, new Integer(31)));

    System.out.println("Searching for 33 ... " +
                BinarySearch.search(array, new Integer(33)));
  }
}
```

Outcome

```
Searching for 10 ... failure
Searching for 11 ... success
Searching for 31 ... success
Searching for 33 ... failure
```

A test plan always has five columns: Test #, Test Data, Reasons and Expected Results. The primary purpose of testing is to demonstrate the presence of errors. Errors are likely to occur around boundary points - the ends of an array, arrays with just one element, loops that execute zero times and just once, each branch of a selection statement. So we pay particular attention to them when drawing up test plans.

Test Plan for BinarySearch.search(Object[] array, Object toFind)

#	Test Data		Reason	Expected Results
	Array Contents	Search Values		
1	array[0] = 0;	-1	array with *one* element	failure
2		0		success
3		1		failure
4	array[0] = 0;	-1	*all* array values are *equal*	failure
5	array[1] = 0;	0		success
6	array[2] = 0;	1		failure
	array[3] = 0;			
	array[4] = 0;			
7	array[0] = 0;	-1	just less than *first* value in the array	failure
8	array[1] = 2;	0	equal to the *first* value in the array	success
9	array[2] = 4;	1	just more than *first* value in the array	failure
	array[3] = 6;			
10	array[4] = 8;	7	just less than *last* value in array	failure
11		8	equal to *last* value in array	success
12		9	just more than *last* value in array	failure
13		2	value at an *odd* array location	success
14		3	value at an *even* array location	success

5.6 REVIEW

5.7 FURTHER READING

DROMEY RG *How to Solve it by Computer* pp 16, 227

In the next lesson we look at tables - arrays with two dimensions.

5.8 EXERCISES

1 With the aid of a sequence of diagrams explain how the binary search algorithm might determine whether **(a)** else, **(b)** class and **(c)** import were items in the array shown below.

array

case	class	else	if	while
0	1	2	3	4

2 Dry run the binary search algorithm implemented in §5.4 above and determine whether **(a)** 31 and **(b)** 15 are in the array shown below.

array

10	12	23	27	30	31	39	42
0	1	2	3	4	5	6	7

3 State two essential pre-conditions for the binary search method to function as expected.

4 Amend the binary search method implemented in §5.4 above so that it returns the index of an item found in the array, or -1 if the item is not found in the array. Test your implementation.