Searching & Sorting in Java – Binary Search

If the data you wish to search is already in order, a sequential search will still, on average, take the same amount of time to find the item you want. It is possible, however, to greatly improve the speed of a search on sorted data.

The *binary search* algorithm is one such way to improve performance using sorted data. This algorithm is an example of a *divide and conquer* algorithm, of which there are many other examples. This type of algorithm solves a problem by quickly reducing its size. For the binary search, at each stage of the problem we cut the size of the problem roughly in half.

To illustrate, consider the following list, and suppose we are searching for the value 47.

16	19	22	24	27	29	37	40	43	44	47	52	56	60	64
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

To start the process, we initially examine the item in the middle of the array. The middle item, 40, is not the one we want, but it is less than the value we are looking for. Since the list is sorted, we use this information to eliminate all of the items in the lower half of the list. Our search now only looks at the remaining (upper) half of the list.

16	19	22	24	27	29	37	40	43	44	47	52	56	60	64
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

We repeat our strategy on these items. The middle value is now 52, which is too high, so we eliminate the upper half of the remaining list.

16	19	22	24	27	29	37	40	43	44	47	52	56	60	64
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

The middle value is now 44, which is too small. Eliminating everything below this value leaves us with only a single item that hasn't been eliminated, which is the location of our target value.

16	19	22	24	27	29	37	40	43	44	47	52	56	60	64
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

To implement this algorithm in Java, we will search for `item` in an array called `list`. Through the process of elimination, the upper and lower bounds of the array that we need to search will change, so we will track them with `int` variables called `top` and `bottom`. Similarly, we need to track the `middle` value, also an `int`.

For each iteration, we can find the value of middle by taking the average of the top and bottom. If the value at middle is equal to item, then obviously our search is done. If our value is too low, the bottom becomes middle + 1. If our value is too high, the top becomes middle - 1.

If our search value is not in the list, this process will continue until bottom and top and middle are all equal to each other (i.e., we are looking at a single element of the array). On the next step, top or bottom will change such that top < bottom, which signals the end of our search, at which point we return a value to indicate a failure (-1).

```java
public static int binSearch ( double[] list, double item)
{
  int bottom = 0;                // lower bound of searching
  int top = list.length - 1;     // upper bound of searching
  int middle;                    // current search candidate
  boolean found = false;
  int location = -1;             // location of item, -1 for failure

  while (bottom <= top && !found)
  {
    middle = (bottom + top)/2;   // integer division, auto-truncate
    if (list[middle] == item)
    {
      location = middle;         // success!
      found = true;
    }
    else if (list[middle] < item)
    {
      bottom = middle + 1;       // look only in top half
    }
    else
    {
      top = middle - 1;          // look only in bottom half
    }
  }
  return location;
}
```

Suppose we want to perform a binary search for the value 75 on the following data.

12	34	47	62	75	87	90

Initially, we need to search the entire array, so bottom and top are set to 0 and 6, while middle is set to 3.

`bottom`	`middle`	`top`
0	3	6

12	34	47	62	75	87	90
0	1	2	3	4	5	6

Since 62 < 75, the item we are seeking cannot be in the left half of the array. We discard this half by setting `bottom` to `middle + 1 = 4`. The middle of the remaining interval is `(4 + 6)/2 = 5`.

bottom	middle	top
4	5	6

0	1	2	3	4	5	6
12	34	47	62	75	87	90

Since `87 > 75`, the value 75 cannot be in the upper half of the sublist, so we discard it by setting `top` to `middle - 1 = 4`. The new value of middle will be `(4 + 4)/2 = 4`.

bottom	middle	top
4	4	4

0	1	2	3	4	5	6
12	34	47	62	75	87	90

Once the value has been found at middle, the search ends successfully.

Now let us consider a failed search, were the final element was not equal to our search value.

bottom	middle	top
4	4	4

0	1	2	3	4	5	6
12	34	47	62	80	87	90

Since `80 > 75`, the value 75 cannot be in the upper half of the sublist, so we discard it by setting `top` to `middle - 1 = 3`. Now we have the situation where `top < bottom`, so our searching ends without a successful result.

Exercises

1. Suppose that an array contains the following elements.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
23	27	30	34	41	49	51	55	57	60	67	72	78	83	96

   Trace the execution of the method `binSearch` shown in this section as it searches for the following values of `item`. In each trace, show the progress of the search by using diagrams similar to those in previous examples.

   (a) 72 (b) 41 (c) 62

2. What changes would have to be made to binSearch so that it will search an array in *descending* order?

3. Rewrite binSearch so that, if a search is unsuccessful, the method will return the index of the value *nearest* to item, instead of returning -1. If there is a tie, return the smaller index).

4. What is the maximum number of comparisons that might be necessary to perform a binary search on a list containing seven items?

5. Repeat the previous question for lists with indicated sizes.
   (a) 3 (c) 31 (e) 100 (g) 1000
   (b) 15 (d) 63 (f) 500 (h) 10000