19.4 Binary Search

The **binary search algorithm** is more efficient than linear search, but it requires that the array be sorted. The first iteration of this algorithm tests the *middle* element in the array. If this matches the search key, the algorithm ends. Assuming the array is sorted in *ascending* order, then if the search key is *less than* the middle element, it cannot match any element in the second half of the array and the algorithm continues with only the first half of the array (i.e., the first element up to, but not including, the middle element). If the search key is *greater than* the middle element, it cannot match any element in the first half of the array and the algorithm continues with only the second half (i.e., the element *after* the middle element through the last element). Each iteration tests the middle value of the remaining portion of the array. If the search key does not match the element, the algorithm eliminates half of the remaining elements. The algorithm ends by either finding an element that matches the search key or reducing the subarray to zero size.

Example

As an example consider the sorted 15-element array

```
   2  3  5  10  27  30  34  51  56  65  77  81  82  93
```

and a search key of 65. A program implementing the binary search algorithm would first check whether 51 is the search key (because 51 is the *middle* element of the array). The search key (65) is larger than 51, so 51 is ignored along with the first half of the array (all elements smaller than 51), leaving

```
   56  65  77  81  82  93  99
```

Next, the algorithm checks whether 81 (the middle element of the remainder of the array) matches the search key. The search key (65) is smaller than 81, so 81 is discarded along with the elements larger than 81. After just two tests, the algorithm has narrowed the number of values to check to only three (56, 65 and 77). It then checks 65 (which indeed matches the search key) and returns the index of the array element containing 65. This algorithm required just three comparisons to determine whether the search key matched an element of the array. Using a linear search algorithm would have required 10 comparisons. [*Note:* In this example, we've chosen to use an array with 15 elements so that there will always be an obvious middle element in the array. With an even number of elements, the middle of the array lies between two elements. We implement the algorithm to choose the higher of those two elements.]

19.4.1 Binary Search

Implementation

Class `BinarySearchTest` (Fig. 19.3) contains:

- `static` method `binarySearch` to search an `int` array for a specified key,
- `static` method `remainingElements` to display the remaining elements in the array being searched, and
- `main` to test method `binarySearch`.

8

The `main` method (lines 60–89) is nearly identical to `main` in Fig. 19.2. In this program, line 65 creates a 15-element stream (Java SE 8) of random values, sorts the values into ascending order, then converts the stream's elements to an array of `int`s. Recall that the binary search algorithm will work only on a *sorted* array. The first line of output from this program shows the sorted array of `int`s. When the user instructs the program to search for `18`, the program first tests the middle element, which is `57` (as indicated by `*`) in our sample execution. The search key is less than `57`, so the program eliminates the second half of the array and tests the middle element from the first half. The search key is smaller than `36`, so the program eliminates the second half of the array, leaving only three elements. Finally, the program checks `18` (which matches the search key) and returns the index `1`.

```
1   // Fig. 19.3: BinarySearchTest.java
2   // Use binary search to locate an item in an arr
3   import java.security.SecureRandom;
```

```
4   import java.util.Arrays;
5   import java.util.Scanner;
6
7   public class BinarySearchTest {
8     // perform a binary search on the data
9     public static int binarySearch(int[] data, in
10       int low = 0; // low end of the search area
11       int high = data.length - 1; // high end of
12       int middle = (low + high + 1) / 2; // midd
13       int location = -1; // return value; -1 if
14
15       do { // loop to search for element
16         // print remaining elements of array
17         System.out.print(remainingElements(data
18
19         // output spaces for alignment
20         for (int i = 0; i < middle; i++) {
21           System.out.print("  ");
22         }
23         System.out.println(" * "); // indicate
24
25         // if the element is found at the middl
26         if (key == data[middle]) {
27           location = middle; // location is th
28         }
29         else if (key < data[middle]) { // middl
30           high = middle - 1; // eliminate the
31         }
32         else { // middle element is too low
33           low = middle + 1; // eliminate the l
34         }
35
36         middle = (low + high + 1) / 2; // recal
37       } while ((low <= high) && (location == -1)
38
39       return location; // return location of sea
40     }
41
42     // method to output certain values in array
43     private static String remainingElements(
```

```
   int[] data, int low, int high) {
      StringBuilder temporary = new StringBuilde
      // append spaces for alignment
      for (int i = 0; i < low; i++) {
         temporary.append("  ");
      }
      // append elements left in array
      for (int i = low; i <= high; i++) {
         temporary.append(data[i] + " ");
      }
      return String.format("%s%n", temporary);
   }
   public static void main(String[] args) {
      Scanner input = new Scanner(System.in);
      SecureRandom generator = new SecureRandom(
      // create array of 15 random integers in s
      int[] data = generator.ints(15, 10, 91).so
      System.out.printf("%s%n%n", Arrays.toStrin
      // get input from user
      System.out.print("Please enter an integer
      int searchInt = input.nextInt();
      // repeatedly input an integer; -1 termina
      while (searchInt != -1) {
         // perform search
         int location = binarySearch(data, searc
         if (location == -1) { // not found
            System.out.printf("%d was not found%
         }
         else { // found
            System.out.printf("%d was found in p
               searchInt, location);
         }
```

```
                        84
       85            // get input from user
 86            System.out.print("Please enter an integ
    87             searchInt = input.nextInt();
                    88         }
                     89      }
                      90   }
```

```
[13, 18, 29, 36, 42, 47, 56, 57, 63, 68, 80, 81, 82,
  Please enter an integer value (-1 to quit): 18
   13 18 29 36 42 47 56 57 63 68 80 81 82 88 88
                                     *
               13 18 29 36 42 47 56
                                *
                     13 18 29
                           *
            18 was found in position 1
```

```
  Please enter an integer value (-1 to quit): 82
   13 18 29 36 42 47 56 57 63 68 80 81 82 88 88
                                     *
                          63 68 80 81 82 88 88
                                        *
                                   82 88 88
                                        *
                                     82
                                      *
           82 was found in position 12

  Please enter an integer value (-1 to quit): 69
   13 18 29 36 42 47 56 57 63 68 80 81 82 88 88
                                     *
                          63 68 80 81 82 88 88
                                        *
                                63 68 80
                                     *
```

```
                                          80
                                           *

                  69 was not found

    Please enter an integer value (-1 to quit): -1
```

Fig. 19.3

Use binary search to locate an item in an array (the `*` in the output marks the middle element).

Lines 9–40 declare method `binarySearch`, which receives as parameters the array to search (`data`) and the search key (`key`). Lines 10–12 calculate the `low` end index, `high` end index and `middle` index of the portion of the array that the program is currently searching. At the beginning of the method, the `low` end is `0`, the `high` end is the length of the array minus `1` and the `middle` is the average of these two values. Line 13 initializes the `location` of the element to `-1`—the value that will be returned if the `key` is not found. Lines 15–37 loop until `low` is greater than `high` (this occurs when the `key` is not found) or `location` does not equal `-1` (indicating that the `key` was found). Line 26 tests whether the value in the `middle` element is equal to the `key`. If so, line 27 assigns `middle` to `location`, the loop terminates and `location` is returned to the caller. Each iteration of the loop tests a single value (line 26) and *eliminates half of the remaining values in the array* (lines 29–31 or 32–34) if the value is not the `key`.

19.4.2 Efficiency of the Binary Search

In the worst-case scenario, searching a *sorted* array of 1023 elements takes *only 10 comparisons* when using a binary search. Repeatedly dividing 1023 by 2 (because after each comparison we can eliminate half the array) and rounding down (because we also remove the middle element) yields the values 511, 255, 127, 63, 31, 15, 7, 3, 1 and 0. The number 1023 $(2^{10} - 1)$ is divided by 2 only 10 times to get the value 0, which indicates that there are no more elements to test. Dividing by 2 is equivalent to one comparison in the binary search algorithm. Thus, an array of 1,048,575 $(2^{20} - 1)$ elements takes a *maximum of 20 comparisons* to find the key, and an array of over one billion elements takes a *maximum of 30 comparisons* to find the key. This is a tremendous improvement in performance over the linear search. For a one-billion-element array, this is a difference between an average of 500 million comparisons for the linear search and a maximum of only 30 comparisons for the binary search! The maximum number of comparisons needed for the binary search of any sorted array is the exponent of the first power of 2 greater than the number of elements in the array, which is represented as $\log_2 n$. All logarithms grow at roughly the same rate, so in big O notation the base can be omitted. This results in a big O of ***O*(log *n*)** for a binary search, which is also known as **logarithmic run time** and pronounced as “order log n.”