Chapter 8

Search and Sort

Goals

This chapter begins by showing two algorithms used with arrays: selection sort and binary search. After studying this chapter, you will be able to

- understand how binary search finds elements more quickly than sequential search
- arrange array elements into ascending or descending order (sort them)
- Analyze the runtime of algorithms

8.1 Binary Search

The binary search algorithm accomplishes the same function as sequential search (see Chapter 8, “Arrays”). The binary search presented in this section finds things more quickly. One of the preconditions is that the collection must be sorted (a sorting algorithm is shown later).

The binary search algorithm works like this. If the array is sorted, half of the elements can be eliminated from the search each time a comparison is made. This is summarized in the following algorithm:

Algorithm: Binary Search, used with sorted arrays

```
while the element is not found and it still may be in the array {
    if the element in the middle of the array is the element being searched for
        store the reference and signal that the element was found so the loop can terminate
    else
        arrange it so that the correct half of the array is eliminated from further search
}
```

Each time the search element is not the element in the middle, the search can be narrowed. If the search item is less than the middle element, you search only the half that precedes the middle element. If the item being sought is greater than the middle element, search only the elements that are greater. The binary search effectively eliminates half of the array elements from the search. By contrast, the sequential search only eliminates one element from the search field with each comparison. Assuming that an array of strings is sorted in alphabetic order, sequentially searching for `"Ableson"` does not take long. `"Ableson"` is likely to be located near the front of the array elements. However, sequentially searching for `"Zevon"` takes much more time—especially if the array is very big (with millions of elements).

The sequential search algorithm used in the `indexOf` method of the previous chapter would have to compare all of the names beginning with A through Y before arriving at any names beginning with Z. Binary search gets to `"Zevon"` much more quickly. When an array is very large, binary search is much faster than sequential search. The binary search algorithm has the following preconditions:

1. The array must be sorted (in ascending order, for now).
2. The indexes that reference the first and last elements must represent the entire range of meaningful elements.

The index of the element in the middle is computed as the average of the first and last indexes. These three indexes—named `first`, `mid`, and `last`—are shown below the array to be searched.

```java
int n = 7;
String[] name = new String[n];
name[0] = "ABE";
name[1] = "CLAY";
name[2] = "KIM";
name[3] = "LAU";
name[4] = "LISA";
name[5] = "PELE";
name[6] = "ROY";
// Binary search needs several assignments to get things going
```

```
int first = 0;
int last = n - 1;
int mid = (first + last) / 2;
String searchString = "LISA";
// -1 will mean that the element has not yet been found
int indexInArray = -1;
```

Here is a more refined algorithm that will search as long as there are more elements to look at and the element has not yet been found.

Algorithm: Binary Search *(more refined, while still assuming that the items have been sorted)*

```
while indexInArray is -1 and there are more array elements to look through {
   if searchString is equal to name[mid] then
       let indexInArray = mid  // This indicates that the array element equaled searchString
   else if searchString alphabetically precedes name[mid]
      eliminate mid . . . last elements from the search
   else
       eliminate first . . . mid elements from the search
   mid = (first + last) / 2;  // Compute a new mid for the next loop iteration (if there is one)
}
// At this point, indexInArray is either -1, indicating that searchString was not found,
// or in the range of 0 through n - 1, indicating that searchString was found.
```

As the search begins, one of three things can happen (the code is searching for a `String` that equals `searchString`):

1. The element in the middle of the array equals `searchString`. The search is complete. Store `mid` into `indexInArray` to indicate where the `String` was found.
2. `searchString` is less than (alphabetically precedes) the middle element. The second half of the array can be eliminated from the search field (`last = mid - 1`).
3. `searchString` is greater than (alphabetically follows) the middle element. The first half of the array can be eliminated from the search field (`first = mid + 1`).

In the following code, if the `String` being searched for is not found, `indexInArray` remains `-1`. As soon as an array element is found to equal `searchString`, the loop terminates. The second part of the loop test stops the loop when there are no more elements to look at, when `first` becomes greater than `last`, or when the entire array has been examined.

```
// Binary search if searchString
// is not found and there are more elements to compare.
while (indexInArray == -1 && (first <= last)) {
  // Check the three possibilities
  if (searchString.equals(name[mid]))
    indexInArray = mid; // 1. searchString is found
  else if (searchString.compareTo(name[mid]) < 0)
    last = mid - 1; // 2. searchString may be in first half
  else
    first = mid + 1; // 3. searchString may be in second half
```

```
      // Compute a new array index in the middle of the search area
      mid = (first + last) / 2;
    } // End while

    // indexInArray now either is -1 to indicate the String is not in the array
    // or when indexInArray >= 0 it is the index of the first equal string found.
```

At the beginning of the first loop iteration, the variables `first`, `mid`, and `last` are set as shown below. Notice that the array is in ascending order (binary search won't work otherwise).

Array and binary search indexes before comparing `searchString` (`"LISA"`) to `name[mid]` (`"LAU"`):

```
    name[0]     "ABE"  ⇐ first == 0
    name[1]     "CLAY"
    name[2]     "KIM"
    name[3]     "LAU"  ⇐ mid == 3
    name[4]     "LISA"
    name[5]     "PELE"
    name[6]     "ROY"  ⇐ last == 6
```

After comparing `searchString` to `name[mid]`, `first` is increased from `0` to `mid + 1`, or `4`; `last` remains `6`; and a new `mid` is computed as (4 + 6) / 2 = `5`.

~~name[0] "ABE"~~ Because "LISA" is greater than name[mid],
~~name[1] "CLAY"~~ the objects name[0] through name[3] no longer
~~name[2] "KIM"~~ need to be searched through and can be eliminated from
~~name[3] "LAU"~~ subsequent searches. That leaves only three possibilities.

```
name[4]     "LISA"⇐    first == 4
name[5]     "PELE"⇐      mid == 5
name[6]     "ROY" ⇐     last == 6
```

With `mid == 5`, `"LISA".compareTo("PELE") < 0` is true. So `last` is decreased (5 - 1 = 4), `first` remains `4`, and a new `mid` is computed as `mid = (4 + 4) / 2 = 4`.

~~name[0] "ABE"~~
~~name[1] "CLAY"~~
~~name[2] "KIM"~~
~~name[3] "LAU"~~

```
name[4]     "LISA"⇐ mid == 4    ⇐ first == 4    ⇐ last == 4
```

~~name[5] "PELE"~~
~~name[6] "ROY"~~ Because "LISA" is less than name[mid], eliminate name[6].

Now name[mid] does equal searchString ("LISA".equals("LISA")), so `indexInArray = mid`. The loop terminates because `indexInArray` is no longer -1. The following code after the loop and the output confirm that `"LISA"` was found in the array.

```
    if (indexInArray == -1)
      System.out.println(searchString + " not found");
    else
      System.out.println(searchString + " found at index " + indexInArray);
```

Output

```
LISA found at index 4
```

Terminating when searchName Is Not Found

Now consider the possibility that the data being searched for is not in the array; if `searchString` is `"DEVON"`, for example.

```
    // Get the index of DEVON if found in the array
    String searchName = "DEVON";
```

This time the values of `first`, `mid`, and `last` progress as follows:

	first	***mid***	***last***	***Comment***
#1	0	3	6	Compare "DEVON" to "LAU"
#2	0	1	2	Compare "DEVON" to "CLAY"
#3	2	2	2	Compare "DEVON" to "KIM"
#4	2	2	1	`first <= last` is false—the loop terminates

When the `searchString` (`"DEVON"`) is not in the array, `last` becomes less than `first` (`first > last`). The two indexes have crossed each other. Here is another trace of binary search when the searched for element is not in the array.

		#1	#2	#3	#4
`name[0]`	`"ABE"`	⇐ first	⇐ first		
`name[1]`	`"CLAY"`		⇐ mid		**last**
`name[2]`	`"KIM"`		⇐ last	⇐ first, mid, last	**first**
`name[3]`	`"LAU"`	⇐ mid			
`name[4]`	`"LISA"`				
`name[5]`	`"PELE"`				
`name[6]`	`"ROY"`	⇐ last			

After `searchString` ("DEVON") is compared to `name[2]` (`"KIM"`), no further comparisons are necessary. Since `DEVON` is less than `KIM`, `last` becomes `mid` - 1, or 1. The new `mid` is computed to be 2, but it is never used as an index. This time, the second part of the loop test terminates the loop.

```
    while(indexInArray == -1 && (first <= last))
```

Since `first` is no longer less than or equal to `last`, `searchString` cannot be in the array. The `indexInArray` remains `-1` to indicate that the element was not found.

Comparing Running Times

The binary search algorithm can be more efficient than the sequential search algorithm. Whereas sequential search only eliminates one element from the search per comparison, binary search eliminates half of the elements for each comparison. For example, when the number of elements (n) == 1,024, a binary search eliminates 512 elements from further search in the first comparison, 256 during the second comparison, then 128, 64, 32, 16, 4, 2, and 1.

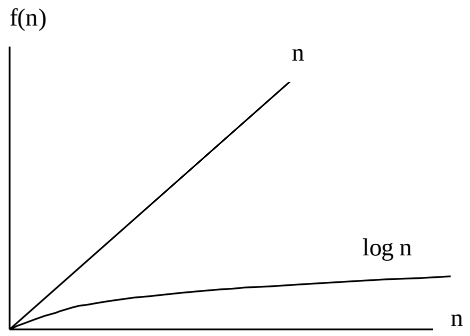
When n is small, the binary search is not much faster than sequential search. However, when n gets large, the difference in the time required to search for something can make the difference between selling the software and having it flop. Consider how many comparisons are necessary when n grows by powers of two. Each doubling of n would require potentially twice as many loop iterations for sequential search. However, the same doubling of n would require potentially only one more comparison for binary search.

The Maximum Number of Comparisons during Two Different Search Algorithms

Power of 2	n	Sequential Search	Binary Search
2^2	4	4	2
2^4	16	16	4
2^8	256	256	8
2^{12}	4,096	4,096	12
2^{24}	16,777,216	16,777,216	24

As n gets very large, sequential search has to do a lot more work. The numbers above represent the maximum number of iterations to find an element or to realize it is not there. The difference between 24 comparisons and almost 17 million comparisons is quite dramatic even on a fast computer.

In general, as the number of elements to search (n) doubles, binary search requires only one iteration to eliminate half of the elements from the search. The growth of this function is said to be logarithmic. The following graph illustrates the difference between linear search and binary search as the size of the array grows.



Self-Check

8-1 Give at least one precondition for a successful binary search.

8-2 What is the maximum number of comparisons (approximately) performed on a list of 1,024 elements during a binary search? (*Hint:* After one comparison, only 512 array elements need be searched; after two searches, only 256 elements need be searched, and so on.)

8-3 During a binary search, what condition signals that the search element does not exist in an array?

8-4 What changes would be made to the binary search when the elements are sorted in descending order?

8.2 One Sorting Algorithm

The elements of a collection are often arranged into either ascending or descending order through a process known as **sorting**. To sort an array, the elements must be compared. For `int` and `double`, < or > suffices. For `String`, `Integer`, and `BankAccount` objects, the `compareTo` method is used.

There are many sorting algorithms. Even though others are more efficient (run faster), the relatively simple selection sort is presented here. The goal here is to arrange an array of integers into ascending order, the natural ordering of integers.

Object Name	Unsorted Array	Sorted Array
`data[0]`	`76.0`	`62.0`
`data[1]`	`91.0`	`76.0`
`data[2]`	`100.0`	`89.0`
`data[3]`	`62.0`	`91.0`
`data[4]`	`89.0`	`100.0`

With the selection sort algorithm, the largest integer must end up in `data[n - 1]` (where n is the number of meaningful array elements). The smallest number should end up in `data[0]`. In general, an array `x` of size `n` is sorted in ascending order if `x[j] <= x[j + 1]` for `j = 0` to `n-2`.

The selection sort begins by locating the smallest element in the array by searching from the first element (`data[0]`) through the last (`data[4]`). The smallest element, `data[2]` in this array, is then swapped with the top element, `data[0]`. Once this is done, the array is sorted at least through the first element.

top == 0	Before	After	Sorted
`data[0]`	***76.0***	***62.0***	⇐
`data[1]`	`91.0`	`91.0`	
`data[2]`	`100.0`	`100.0`	
`data[3]`	***62.0***	***76.0***	
`data[4]`	`89.0`	`89.0`	

Placing the Largest Value in the "Top" Position (index 0)

The task of finding the smallest element is accomplished by examining all array elements and keeping track of the index with the smallest integer. After this, the smallest array element is swapped with `data[0]`. Here is an algorithm that accomplishes these two tasks:

Algorithm: Finding the smallest in the array and switching it with the topmost element

```
(a)        top = 0
// At first, assume that the first element is the smallest
(b)        indexOfSmallest = top
// Check the rest of the array (data[top + 1] through data[n - 1])
(c)        for index ranging from top + 1 through n - 1
             (c1)  if data[index] < data[indexOfSmallest]
                     indexOfSmallest = index
// Place the smallest element into the first position and place the first array
// element into the location where the smallest array element was located.
(d)        swap data[indexOfSmallest] with data[top]
```

The following algorithm walkthrough shows how the array is sorted through the first element. The smallest integer in the array will be stored at the "top" of the array—`data[0]`. Notice that `indexOfSmallest` changes only when an array element is found to be less than the one stored in `data[indexOfSmallest]`. This happens the first and third times step c1 executes.

		`indexOf`							
Step	`top`	`Smallest`	`index`	`[0]`	`[1]`	`[2]`	`[3]`	`[4]`	`n`
	?	?	?	76.0	91.0	100.0	62.0	89.0	5
(a)	0	"	"	"	"	"	"	"	"
(b)	"	0	"	"	"	"	"	"	"
(c)	"	"	1	"	"	"	"	"	"
(c1)	"	***1***	"	"	"	"	"	"	"
(c)	"	"	2	"	"	"	"	"	"
(c1)	"	"	"	"	"	"	"	"	"
(c)	"	"	3	"	"	"	"	"	"
(c1)	"	***2***	"	"	"	"	"	"	"
(c)	"	"	4	"	"	"	"	"	"
(c1)	"	"	"	"	"	"	"	"	"
(c)	"	"	5	"	"	"	"	"	"
(d)	"	"	"	***62.0***	"	"	***76.0***	"	"

This algorithm walkthrough shows `indexOfSmallest` changing twice to represent the index of the smallest integer in the array. After traversing the entire array, the smallest element is swapped with the top array element. Specifically, the preceding algorithm swaps the values of the first and fourth array elements, so `62.0` is stored in `data[0]` and `76.0` is stored in `data[3]`. The array is now sorted through the first element!

The same algorithm can be used to place the second smallest element into `data[1]`. The second traversal must begin at the new "top" of the array—index 1 rather than 0. This is accomplished by incrementing `top` from 0 to 1. Now a second traversal of the array begins at the second element rather than the first. The smallest element in the unsorted portion of the array is swapped with the second element. A second traversal of the array ensures that the first two elements are in order. In this example array, `data[3]` is swapped with `data[1]` and the array is sorted through the first two elements.

top == 1	Before	After	Sorted
`data[0]`	`62.0`	`62.0`	⇐
`data[1]`	***`91.0`***	***`76.0`***	⇐
`data[2]`	`100.0`	`100.0`	
`data[3]`	***`76.0`***	***`91.0`***	
`data[4]`	`89.0`	`89.0`	

This process repeats a total of n - 1 times.

top == 2	Before	After	Sorted
`data[0]`	`62.0`	`62.0`	⇐
`data[1]`	`76.0`	`76.0`	⇐
`data[2]`	***`100.0`***	***`89.0`***	⇐
`data[3]`	`91.0`	`91.0`	
`data[4]`	***`89.0`***	***`100.0`***	

And an element may even be swapped with itself.

top == 3	Before	After	Sorted
`data[0]`	`62.0`	`62.0`	⇐
`data[1]`	`76.0`	`76.0`	⇐
`data[2]`	`89.0`	`89.0`	⇐
`data[3]`	***91.0*** ⇄	***91.0***	⇐
`data[4]`	`100.0`	`100.0`	

When `top` goes to `data[4]`, the outer loop stops. The last element need not compared to anything. It is unnecessary to find the smallest element in an array of size 1. This element in `data[n - 1]` must be the largest (or equal to the largest), since all of the elements preceding the last element are already sorted in ascending order.

top == 3 and 4	Before	After	Sorted
`data[0]`	`62.0`	`62.0`	⇐
`data[1]`	`76.0`	***76.0***	⇐
`data[2]`	`89.0`	`89.0`	⇐
`data[3]`	`91.0`	`91.0`	⇐
`data[4]`	`100.0`	`100.0`	⇐

Therefore, the outer loop changes the index `top` from 0 through n - 2. The loop to find the smallest index in a portion of the array is nested inside a loop that changes `top` from 0 through n - 2 inclusive.

Algorithm: Selection Sort

```
for top ranging from 0 through n - 2  {
   indexOfSmallest = top
   for index ranging from top + 1 through n - 1    {
       if data[indexOfSmallest] < data[index] then
          indexOfSmallest = index
   }
   swap  data[indexOfSmallest] with data[top]
}
```

Here is the Java code that uses selection sort to sort the array of numbers shown. The array is printed before and after the numbers are sorted into ascending order.

```java
double[] data = { 76.0, 91.0, 100.0, 62.0, 89.0 };
int n = data.length;

System.out.print("Before sorting: ");
for(int j = 0; j < data.length; j++)
  System.out.print(data[j] + " ");
System.out.println();
```

```
int indexOfSmallest = 0;

for(int top = 0; top < n - 1; top++) {
  // First assume that the smallest is the first element in the subarray
  indexOfSmallest = top;

  // Then compare all of the other elements, looking for the smallest
  for(int index = top + 1; index < data.length; index++)
  { // Compare elements in the subarray
    if(data[index] < data[indexOfSmallest])
      indexOfSmallest = index;
  }

  // Then make sure the smallest from data[top] through data.size
  // is in data[top]. This message swaps two array elements.
  double temp = data[top]; // Hold on to this value temporarily
  data[top] = data[indexOfSmallest];
  data[indexOfSmallest] = temp;
}
System.out.print(" After sorting: ");
for (int j = 0; j < data.length; j++)
  System.out.print(data[j] + " ");
System.out.println();
```

Output

```
Before sorting: 76.0 91.0 100.0 62.0 89.0
 After sorting: 62.0 76.0 89.0 91.0 100.0
```

Sorting an array usually involves elements that are more complex. The sorting code is most often located in a method. This more typical context for sorting will be presented later.

This selection sort code arranged the array into ascending numeric order. Most sort routines arrange the elements from smallest to largest. However, with just a few simple changes, any primitive type of data (such as `int`, `char`, and `double`) may be arranged into descending order using the > operator.

```
if(data[index] < data[indexOfSmallest])
   indexOfSmallest = index;
```

becomes

```
if(data[index] > data[indexOfLargest])
  indexOfLargest = index;
```

Only primitive types can be sorted with the relational operators < and >. Arrays of other objects, `String` and `BankAccount` for example, have a `compareTo` method to check the relationship of one object to another.

Self-Check

8-5 Alphabetizing an array of strings requires a sort in which order, ascending or descending?

8-6 If the smallest element in an array already exists as `first`, what happens when the swap function is called for the first time (when `top` = 0)?

8-7 Write code that searches for and stores the largest element of array `x` into `largest`. Assume that all elements from `x[0]` through `x[n - 1]` have been given meaningful values.

Answers to Self-Check Questions

8-1 The array is sorted.

8-2 1,024; 512; 256; 128; 64; 32; 16; 8; 4; 2; 1 == 11

8-3 When `first` becomes greater than `last`.

8-4 Change the comparison from less than to greater than.

```
if(searchString.compareTo(str[mid]) > 0)
  last = mid - 1;
else
   first= mid + 1;  // ...
```

8-5 Ascending

8-6 The first element is swapped with itself.

8-7

```
int largest = x[0];
for(int j = 0; j < n; j++) {
  if(x[j] > largest)
     largest = x[j];
}
```