10.1 Sequential Search

To begin, suppose that you are looking for a particular CD in your collection. After rummaging around for a while with no luck, you may decide to go through the collection systematically, looking at each CD in turn. If you are patient, you will eventually find the one that you are looking for (if it is there).

This simple technique, called a *sequential search*, can be applied to an array of values.

Example 1

The following method performs a sequential search on an array of `String` values for the value called `item`. If the search is successful, the method returns the index in the array of `item` but, if the search fails, the method returns -l.

```
public static int seqSearch (String[] list, String item)
{
        int location = -1;
        for (int i = 0; i < list.length; i++)
        if (list[i] .equals(item))
                location = i;
        return location;
}
```

Note that the variable `location` is initialized to indicate that the search is unsuccessful. If `item` is not found, this is the value that the method will return. Only if we find item in the list will `location` be changed.

Although this method works, it is not as efficient as we might wish. Even if it finds `item` early in the search, it stupidly goes on looking through the rest of the array. We can correct this defect by using a `boolean` variable that acts as a flag to stop the search as soon as `item` has been found. We use this device in the next example.

Example 2

This method shows an efficient implementation of sequential search that quits examining a list immediately after the search has found `item`.

```
public static int seqSearch (String[] list, String item)
{
        int location = -1;
        boolean found = false;
        for (int i = 0; i < list.length && !found; i++)
                if (list[i] .equals (item))
                {
                        location = i;
                        found = true;
                }
        return location;
}
```

We can achieve even greater efficiency for our sequential search, as shown in the next example.

Example 3

By eliminating the boolean flag found and the corresponding test in the for statement to determine the state of found, we can create the following very efficient sequential search.

```
public static int seqSearch (String[] list, String item)
{
        for (int i = 0; i < list.length; i++)
                if (list[i] .equals(item))
                        return i;
        return -1;
}
```

Now,if a match for `item` is ever found, we exit both the `for` statement and the method immediately. Only if no match is ever found do we complete the `for` statement and return -1.

Although the code of Example 3 is both simpler and more efficient than that of Example 2, some people still argue that the solution in Example 2 is preferable. In Example 3, the first line of the `for` statement would make a reader think that the loop is going to be executed for every value in the array but, if a match for `item` is found, this does not happen. In Example 2, on the other hand, the first line of the `for` statement is quite clear about what is going to happen - the loop will be executed as long as we haven't reached the end of the array and we haven't found what we are seeking.

Exercises 10.1

1. Suppose that the method of Example 1 was used to search an array. What would the method return if `item` appeared in the list more than once?

The method will return the last index of where the same item lies.

For example:

```
public class Question1 {

      public static void main(String[] args) {
            // TODO Auto-generated method stub
```

```
                String [] array = {"red", "yellow","green", "red","bleu"};
                String item = "red";
                System.out.println(seqSearch(array,item));

        }

        public static int seqSearch (String[] list, String item)
        {
                int location = -1;
                for (int i = 0; i < list.length; i++)
                if (list[i] .equals(item))
                        location = i;
                return location;
        }

}
```

In this array the item "red" is being repeated twice in the array at the location of index 0 and 3. Hence, when the method is executed for the item "red", the method returns a 3. This is because the method returns the last index of where the item is located.

2. What changes should be made to the sequential search in Example 2 so that it searches an array of values starting at the top and moving downward?

```
        public static int seqSearch (String[] list, String item)
        {
                int location = -1;
                boolean found = false;
                for (int i = list.length-1; i >= 0 && !found; i--)
                        if (list[i] .equals (item))
                        {
                                location = i;
                                found = true;
                        }
                return location;

        }
```

3. A modification of the basic sequential search operates in the following way. If the item being sought is found, it is interchanged with the item that preceded it. If, for example, we were searching for 7 in the list

```
4.   3   9   5   7   2   8   4
```

then, after finding 7, the list would be rearranged in the order

```
3   9   7   5   2   8   4
```

   A. Write a method that implements this technique to search an array of `int` values.

```
        public static int [] seqSearch (int [] list, int item) {
                int temp = 0;
                for (int x = 0; x < list.length; x++) {
                        if (list[x] == item && x != 0) {
                                temp = list[x];
                                list[x] = list[x-1];
                                list[x-1] = temp;
                                return list;
                        }
                }
                return list;

        }
```

B. Test your method in a program that first asks for the length of the list to be searched and then reads that many integers into an array. The program should then repeatedly prompt the user for values until the user supplies a sentinel of zero. The program should print the initial list and then, for each non-zero value read, it should use your modified sequential search to try to locate that item and then print the resulting array.

```
    System.out.println("Enter the length of the list to be searched:");
                int length = input.nextInt();
                int [] list = new int [length];
                int value = 1;
                for(int x = 0; x < length; x++) {
                        System.out.println("Enter integer " + (x+1));
                        list[x] = input.nextInt();
                }
                int [] temp = new int [list.length];
                for (int x = 0; x < list.length; x++)
                        temp[x] = list[x];

                do {
                        System.out.println("Enter value to be located");
                        value = input.nextInt();
                        if (value == 0)
                                break;
                        int [] resultingList = seqSearch(list, value);
                        System.out.print("Initial Array\t-\t");
                        for (int x = 0; x < temp.length; x++)
                                System.out.print(temp[x] + "\t");
                        System.out.print("\nResulting Array\t-\t");
                        for (int x = 0; x < resultingList.length; x++)
                                System.out.print(resultingList[x] + "\t");
                        System.out.println();
                }

                while (value !=0);
```

C. Why might this modification sometimes improve the efficiency of a sequential search?

   This modification might improve the efficiency of the sequential search every time the user simply wants to search a value in the list and swap it with another value within the list.