Computer Science Correspondence School

discere domi

Lecture 20

CISC 131
Introduction to Programming and Problem Solving
Spring 2020
Searching an Array

Due: **Tuesday, May 5, 2020, at start of class**
Points: **none**

Solutions to Problems from Lecture 19

Here are my solutions to the functions you were to write. Your implementations may differ and, if you followed the requirements, that is okay. There are usually several ways to implement the logic of an algorithm.

```
function toNumber ( data )
{
 var result;

 result = NaN;
 data = ( data + "" ).trim();  // force data to be a String
 if ( data.length > 0 ) result = Number( data );

 return result;
}

function isNumeric ( data ) { return ! isNaN ( toNumber( data ) ); }

function isInteger ( data )
{
 return isNumeric ( data ) &&
        Math.ceil(toNumber(data)) === Math.floor(toNumber(data));
}
```

Start of Today's Lecture

Searching an array is a common activity in programs and there are several approaches that can be taken. Some of these approaches depend on the order of the values in the array and others do not.

Searching is not like finding the minimum value or the maximum value in an array. If an array contains at least one element, then the array has both a minimum value and a maximum value. The algorithm does not search for the minimum value or the maximum value. Instead, it identifies the location of a value that it *knows* is in the array by accessing each value in the array and comparing it to the current minimum (or maximum). This is quite different than searching. When searching, the algorithm can end as soon as the searched for item is found and, therefore, may not have to examine each element in the array.

There are two possible outcomes when searching for a value in an array: the value is in the array somewhere or the value is not in the array at all.

All search algorithms must have a way of distinguishing these two possible outcomes. While this can be done in a few different ways, the most general approach is to have the algorithm return the element number where the search for value occurs if that value is in the array. If the algorithm cannot find the searched for value, it returns negative one. By having the algorithm return a valid subscript value (zero through *array.length* – 1) for a successful search and an invalid subscript value (negative one) for an unsuccessful search, the algorithm is able to both indicate if the search was successful and also, in the case of a successful search, tell you the location in the array of the searched for value. This can be essential when using associated arrays.

Several different approaches are shown below. You should implement each one and test it out. Make sure it works and make sure you understand why it works.

Basic Array Search

Shown below is a basic array search element. It is important that you remember how to write this basic algorithm. It has a tendency to appear on exams.

```
function searchArray ( array, forWhat )
{
 // find occurrence nearest the beginning of the array
 var i;
 var result;

 result = -1;
 if ( array != null )
 {
  for ( i=0; i < array.length && array [i] !== forWhat; i++ ) {}
  if ( i < array.length ) result = i;
 }

 return result;
}
```

This algorithm makes no assumption about the order of the values in the array. They can be in any order and the algorithm will work correctly. If *forWhat* is in the array more than one time, the function returns the location of *forWhat* that is nearest the beginning of the array.

You may be a little puzzled by the *for* loop. Its body contains no statements because all of the logic necessary to do the search is in the *for* statement itself. The loop ends as soon as it locates

the value *forWhat* so the loop may not examine all the elements of the array. However, if *forWhat* is not in the array, it will only realize that after examining all the elements in the array.

Here is a variation of this search. In this variation, if *forWhat* is in the array more than one time, the function returns the location of *forWhat* that is nearest the end of the array.

```
function searchArray ( array, forWhat )
{
 // find the occurrence nearest the end of the array
 var i;
 var result;

 result = -1;
 if ( array != null && array.length > 0 )
 {
  for ( i=0; i < array.length; i++  )
  {
   if ( array [i] === forWhat ) result = i;
  }  // for
 }  // if

 return result;
}
```

In this variation, every element of the array must be examined because the algorithm is trying to identify the one nearest the end of the array. It doesn't matter if the search is successful (*forWhat* is in the array somewhere) or unsuccessful (*forWhat* is not in the array), each element of the array is examined. Why is there an additional condition on the *if* statement?

Examine the following function and analyze its behavior by comparing it to the two functions shown above. What does it do? How much of the array does it process?

```
function searchArray ( array, forWhat )
{
 var i;

 i = -1;
 if ( array != null )
 {
  for ( i =array.length-1; i>=0 && array [i] !== forWhat; i-- ) {}
 }

 return i;
}
```

Basic Sorted Array Search

A more efficient search algorithm can be written if the values in the array are in sorted order. In the basic search algorithm, an unsuccessful search required all of the elements of the array to be examined. If the array is sorted, then the algorithm can stop looking for the searched for value as soon as it encounters an array element whose value is greater than or equal to the searched for value.

The algorithm will still need to look at every value in the array sometimes. For example, if the searched for value is the value of the last element in the array or if the searched for value is greater than any value in the array. On average, though, searching a sorted array is more efficient that searching an unsorted array.

```
function searchSortedArray ( array, forWhat )
{
 var hold;
 var i;
 var result;

 result = -1;
 if ( array !== null )
 {
  for ( i=0; i< array.length && array [ i ] < forWhat; i++) {}
  if ( i < array.length  && forWhat === array [ i ] ) result = i;
 }

 return result;
}
```

This algorithm is very similar to that of the basic search algorithm. It differs mainly in one of the conditions in the loop. In the basic search algorithm, === or !== was used when comparing the array element value to the searched for value. In this loop, the test is <. This will cause the loop to end as soon as it accesses an array element value that is greater than or equal to the searched for value. The *if* statement following the loop resolves the ambiguity as to whether the search was successful or not successful.

Modified Sorted Array Search

When accessing an array, it is very important to never attempt to access an element before the beginning of the array or after the last element of the array. That is, a subscript value must always be a non-negative integer that is less than the array length. It is for this reason that the above algorithms always initialize *i* to a valid subscript value as the first condition in the loop. The algorithm shown below does not use that loop condition but still prevents the array from being accessed outside its bounds (zero through *array.length-1*). Can you determine how it does this?

```
function searchSortedArray1 ( array, forWhat )
{
 var hold;
 var i;
 var result;

 result = -1;
 if ( array !== null && array.length > 0 && forWhat >= array[0] )
 {
  hold = array [ array.length-1 ];
  array [ array.length-1 ] = forWhat;

  for ( i=0; array [ i ] < forWhat; i++ ){}

  // restore the original array
  array [ array.length-1 ] = hold;

  // check to see if forWhat was found
  if ( forWhat === array [ i ] ) result = i;
 }

 return result;
}
```

This algorithm is slightly more efficient that the basic sorted array search algorithm because it does one fewer comparisons each time through the loop. That is, there is no && in the loop condition.

Binary Search of Sorted Array

A binary search algorithm can only be used on an array whose values are in sorted order. Its strategy is to examine the value of the middle element of the array. If that value matches the searched for value, then the search is successfully completed. If it does not match, then, since the array values are in sorted order, if the searched for value is in the array it is either in the top half (among the values that are less than the value of the middle element) or in the bottom half (among the values that are greater than the value of the middle element). This approach is repeated until the searched for value is found or there are no places left to look for it in the array.

Binary search algorithms are often described using recursion or nested *if* statements. These implementations tend to hide the underlying aspects of the algorithm. To make the algorithm easier to understand, two algorithms are implemented below. One algorithm calculates the middle element and compares its value to the searched for value. If it matches the searched for value, an array of length one is returned that contains the element number in the array where the match was found. If the value of the middle element does not match the searched for value, an array of length two is returned. The values in these two elements define the range of the array to be searched during the next iteration of the algorithm. If the algorithm sees that there is no place left to search, it returns a zero length array. Here is the first algorithm:

```
function searchPartOfArray ( array, forWhat, start, end )
{
 var i;
 var middle;
 var result;

 // this does one iteration of a binary search
 if ( end < start )
   result = [];      // a zero length array
   else
   {
    middle = Math.floor ( (start + end) / 2 );
    if ( forWhat === array [ middle ] ) result = [ middle ];
    if ( forWhat <   array [ middle ] ) result = [ start,  middle-1 ];
    if ( forWhat >   array [ middle ] ) result = [ middle+1, end ];
   }
 return result;
}
```

In previous assignments, you have created arrays using either *new Array(n)* or by creating and intializing the array in a single statement as shown below.

```
var age;
age = [ 18, 22, 24, 20, 26 ];
```

The assignment statement creates an array of five elements initialized to the values shown.

The *if* statements use in the above function use the same approach. The following statement will create an array name *result* that contains one element and will initialize that element to whatever the value of the variable *middle* is.

```
result = [ middle ];
```

Similarly, the following statement will create an array named *result* that contains two elements. The first element will be initialized to whatever value of the variable start is and the second element will be initialized to one less than whatever the value of the variable middle is.

```
result = [ start,  middle-1 ]
```

The second algorithm (shown below) calls the above algorithm and, based on the array length that is returned, either calls it again or stops – either identifying a successful search or an unsuccessful search. As in the above algorithms, the unsuccessful search causes negative one to be returned. A successful search returns the element number in the array where the searched for value was found. Here is this second algorithm:

```
function binarySearch ( array, forWhat )
{
 var hold;
 var result;

 result = -1;
 if ( array !== null && array.length > 0 )
 {
  hold = searchPartOfArray ( array, forWhat, 0, array.length-1 );
  while ( hold.length === 2 )
  {
   hold = searchPartOfArray ( array, forWhat, hold[0], hold[1] );
  }
 }  // if

 if ( hold.length === 1 ) result = hold [ 0 ];

 return result;
}
```

Implement and test the binary search algorithm. Study it carefully. It may be helpful to draw a sorted array on paper and follow the logic. The idea of binary search is very important and occurs in many places.

Imagine you had an array that contained 1024 elements. If you use a basic search, the worst case is that the element you are searching for is not in the array but you would have to look at all 1024 elements before the funciton realized that. In a binary search, only 10 elements would need to be examined before the searched for element was found or the search failed. If you double the number of elements, binary search would need to examine only one additional element while the basic search would, once again, need to examine them all. Remember though, binary search will only work if the array is in sorted order.