CISC 131
Introduction to Programming and Problem Solving
Spring 2020
Minimum, Maximum, and Sorting an Array

Due: **Monday, April 27, 2020**
Points: **none**

Start of Today's Lecture

Arrays are a convenient way to hold and organize multiple values and there are several functions that are frequently used with arrays:

- find the smallest (minimum) value in the array
- find the largest (maximum) value in the array
- search an array to determine if it contains a specific value
- reorganize the order of the values in the array to place them in ascending or descending order

Today's lecture will discuss some of these important array processing activities. But before we get into that, there is a final bit of *for* loop syntax to introduce.

++ and –

Most modern languages have syntax that is based on the very old C language. C has a lot of operators whose purpose is to reduce the amount of typing done by the programmer. My view of programming is that clarity is more important than brevity but the world is as we find it, not as we create it. While Javascript has adopted all of these C shortcut operators, the style we are using in class permits their use only in the *for* statement and only in that part of the statement that changes the value of the loop variable after each iteration of the loop. For example, if the loop

```
for(i=0; i<9; i=i+1)
```

the *i=i+1* part is the change to the loop variable that is done after each iteration of the loop. This change can be anything but, in many cases, it is either an increment (adding one to something) or a decrement (subtracting one from something.

Most programmers in languages that use the C syntax would not write the increment statement the way it is shown in the above *for* loop. Instead, they would use one of the C shortcut operators. In the C based languages, `++` is an operator that, when it appears after a variable name, combines increment and assignment. Similarly, `--` is an operator that, when it appears after a variable name, combines decrement and assignment. For example,

`i = i + 1`	is equivalent to	`i++`
`j = j - 1`	is equivalent to	`j--`
`x--`	is equivalent to	`x = x -1`
`y++`	is equivalent to	`y = y + 1`

Although there are many such two character operators, we will use only the `++` and the `-` operators. Do not use any of the others. In addition, `++` and `--` may only be used in the change part of the for loop. Do not use them in any other place.

You may be wondering why we are using them at all. It is purely a matter of perception by other programmers. Other programmers will expect to see these operators in the *for* loop. That is the only reason we are doing it. So, instead of writing for loops this way:

```
for ( i=0; i<source.length; i=i+1 )
```

write them this way

```
for ( i=0; i<source.length; i++ )
```

Minimum and Maximum Value in an Array

If an array contains at least one element, then the array has an element with a minimum value and an element with the maximum value. If the array has no elements, then, of course it cannot contain a minimum or maximum value.

The minimum (maximum) value in an array could occur more than one time in the array. For example in this array;

0	8
1	2
2	4
3	5
4	2
5	8
6	6

the minimum is 2 and occurs at element number one and element number four. The maximum is 8 and occurs at element number zero and element number five.

There are two approaches we could take when writing a function that returns the minimum or maximum value:

- return the value itself
- return the element number where the value stored

To illustrate, here is an array containing names:

0	sue
1	bob
2	vinh
3	rich
4	chris
5	tsung
6	ann

Using the two approaches, would produce these results:

	Minimum	**Maximum**
return the value	ann	vinh
return the element number	6	2

Returning the value has a couple of problems associated with it. What value should the function return if it is passed a zero length array? The function could return *undefined* or *null* but there is no guarantee that those values are not already stored in the array. So, returning the value thas the problem of not being able to distinguish between an array that has no minimum/maximum (always a zero length array) and one that contains those special values. In addition, returning the value is a less general approach. This happens when there are *associated arrays*.

Associated arrays are two or more arrays each having the same number of elements and where element *i* of each array contains information related to element *i* of the other associated arrays. Here is an an example showing three associated arrays, name, age, and city:

0	sue	17	Takaka
1	bob	18	Dehli
2	vinh	22	Miami
3	rich	21	St. Paul
4	chris	20	St. Paul
5	tsung	18	Minneapolis
6	ann	20	Hutchinson

The associated arrays indicate that *bob* is 18 and lives in Dehli and *ann* is 20 and lives in Hutchinson. If we use the value approach and find the maximum value in the age array, it will return 22 but this doesn't allow us to know the name of the person with the maximum age. Returning the value is not the better approach.

If, on the other hand, we use the element number approach and find the maximum value in the age array, the function will return 2. Now we can use that to determine that *vinh* has the maximum age (because vinh is at element 2 of the name array) and vinh lives in Miami (because Miami is at element 2 of the city array).

Returning the element number rather than the value also indicates whether the array in question is zero length or not. The element number is a subscript into the array. Valid subscripts are greater than or equal to zero and less than the number of elements in the array. If the function returns an invalid subscript (typically *-1*) the function can tell us that the array contains no elements. There are a lot of ways to write such a function and most would use an *if* statement as shown here:

```
function findMinimum ( array )
{
 // return the element number of the minimum value in the
 // array or -1 if the array contains no elements

 var i;
 var result;

 if ( array.length === 0 )
  result = -1;
  else
  result =  0;

 for ( i=0; i<array.length; i++ )
 {
  if ( array[i] < array[result] ) { result = i; }
 }

 return result;
}
```

The *for* loop could have initialized *i* to one instead of zero but not much time would be saved. In the examples shown in this lecture, we won't worry abou the loop executing one more time than it has to.

Since the minimum or maximum could be anywhere in the array, the function must access every element in the array. Each element must be compared to the current minimum to determine if this new element holds a value that is less than the current minimum. That is done in the *for* loop shown above.

The current minimum needs to be initialized before the loop but to what element number? It doesn't matter. It could be any element in the array. The typical approach would assume that element zero holds the minimum value so it is initialized to that before the loop. But, like I said, it could be initialized to any element in the array. Here is an approach that doesn't use an *if* statement. Study it. What element is used as the beginning minimum element. What happens if the array parameter is a zero length array?

```
function findMinimum ( array )
{
 // return the element number of the minimum value in the
 // array or -1 if the array contains no elements

 var i;
 var result;

 result = array.length-1;
 for ( i=array.length-1; i>=0; i-- )
 {
  if ( array[i] < array[result] ) { result = i; }
 }

 return result;
}
```

A function that returns the element number of the *maximum* value is nearly idential to one that returns the element number of the minimum value. All that is necessary is to change the less than sign into a greater than sign inside the *for* loop.

If the array contains more than one value that is the minimum, the function must be written to return either the one nearest the beginning of the array or the one nearest the end of the array. For exmple, in this array

0	8
1	4
2	11
3	5
4	4
5	6
6	4

The *findMinimum* function shown above that contains the *if* statement would return 1. The *findMinimum* function shown above that does not contain *if* statement would return 6. Don't be confused. The function is returning the location in the array where the minimum occurs, not the minimum value itserlf. Does it matter which one is returned? Usually not but there are situations where one or the other is the correct approach.

As an experiment, change the less than sign to a less than or equal sign in each of the two *findMinimum* functions shown above. How does this change their behavior?

Imagine that you wanted to find the minimum value in only a portion of the array. That is, instead of always starting at the beginning of the array and going to the end, you wanted to start after the beginning, say at element number one or element number five. For example, depending on where you start looking any of the elements of the arry could hold the minium value:

Here is a an example:

0	7
1	4
2	11
3	5
4	21
5	19
6	23

This shows the element number of the minimum value depending on where you begin looking:

If you start dooking at This Element Number	This is the element number of the minimum value
0	1
1	1
2	3
3	3
4	5
5	5
6	6

Here is a function that would do that. You send it the element number of where to start looking and it returns the element number that contains the minimum value on or after the starting element number.

```
function findMinimum ( array, startHere )
{

 var i;
 var result;

 if ( startHere >= array.length )
  result = -1;
  else
  result =  startHere;

 for ( i=startHere; i<array.length; i++ )
 {
  if ( array[i] < array[result] ) { result = i; }
 }

 return result;
}
```

You will notice that, if you pass zero to the function, it will give you the minimum of the entire array. What would happen if you were to swap the values of the elements at the *startHere* value and the location of the minimum value returned by the function?

For example, let's say you call the function passing the array shown immediately above and zero as the *startHere* actual parameter value. The function would return one and you would swap then values of element zero and element one. That would change the array to this:

0	**4**
1	**7**
2	11
3	5
4	21
5	19
6	23

Now call the function again and send one as the actual parameter value. The function would return three. Swapping the values at element number one and element number three changes the array to:

0	4
1	**5**
2	11
3	**7**
4	21
5	19
6	23

Now call the function sending two as the actual parameter value. The function would return three. Swapping the values at element number two and element number three changes the array to:

0	4
1	5
2	**7**
3	**11**
4	21
5	19
6	23

Calling the function sending three as the actual parameter value causes it to return three. Swapping the values of element three with itself doesn't change the array:

0	4
1	5
2	7
3	**11**
4	21
5	19
6	23

We can repeat this processing by send four to the array:

0	4
1	5
2	7
3	11
4	**19**
5	**21**
6	23

And then five:

0	4
1	5
2	7
3	11
4	19
5	**21**
6	23

And finally six (although we know that the minimum/maximum when starting at the last element of the array will always be found in the last element of the array.

0	4
1	5
2	7
3	11
4	19
5	21
6	**23**

If you examine the array, you will see that it is in *ascending order*. Ascending order is more accurately called *non-dscending order* because the array may contain duplicate values. You could reverse this order by using a maximum function rather than a minimum one or writing a function that reversed the order of the values in the array.

Excercise: Sorting an Array

Placing the values of an array into non-ascending or non-descending order is called *sorting* the array. Study the above example and then write a function named:

```
function sortNonDescending ( array )
```

The function is passed an array and, when the function returns, the values in the array will be in non-descending order. The function uses the *findMinimum (array, startHere)* function as illustrated above. The function does not return anything. Instead, it changes the order of the values in the parameter array. Test thoroughly using arrays that contain numbers and arrays that contain *Strings*.