CSCE 120: Learning To Code

Module 12.0: Organizing Data I
Searching, Sorting & Data Visualization

This module is designed to familiarize you with basic data organization algorithms including basic searching and sorting algorithms as well as how to utilize a language's libraries to do so. We will also examine ways to organize data visually using the D3js library.

Searching

A fundamental operation on data is to to search for a particular element or elements that meet certain criteria. This could be as simple as finding a particular value in an array of numbers or as complex as finding all student records for a particular major, taking classes in a particular department.

Linear Search

A straightforward search technique is *linear search*. The basic idea is that we iterate through an array of elements, one-by-one, checking if each element meets our criteria. If we find an element that does, we can stop searching and continue processing it. We may also need to handle the situation where the element we're searching for does not exist. In any case, we may be forced to search the entire array of elements in the worst case. This is why this technique is called "linear" search: the amount of work it takes to find our element is linear with respect to the number of elements in the array.

We could also consider several variations on this problem: we may want to find the first, last or every element that meets our criteria or we may want to count the number of such elements, or just simply determine whether or not an array contains such an element.

Recall that ES6, the `find` function was added for arrays which allows us to provide a callback that returns `true`/`false` if our search condition is satisfied. It returns the first value in the array that satisfies our search condition. Prior to ES6, the only function

provided to search was the `indexOf` which only works on numbers and strings. To accomplish a general linear search in earlier versions of JavaScript, we would need to implement code like the following.

```
/**
 * Returns the index of the first occurrence of
 * item in the array, -1 if not found.
 */
function findIndexOfElement(arr, item) {
  for(var i=0; i<arr.length; i++) {
    if(arr[i] === item) {
      return i;
    }
  }
  return -1;
}

/**
 * Returns the index of the first occurrence of
 * the element satisfying the search condition
 * specified by the provided callback
 */
function findIndexOf(arr, matches) {
  for(var i=0; i<arr.length; i++) {
    if(matches(arr[i]) {
      return i;
    }
  }
  return -1;
}
```

Binary Search

Another, more efficient search technique when dealing with arrays is to use a *binary search*. Suppose that the array we're searching is sorted and we're searching for an item k. Compare k to the middle element in the array, m. If $k = m$, then we've found our element and we're done. Otherwise:

- If $k < m$ then we know that if k exists, it must be in the first half of the array
- If $k > m$ then we know that if k exists, it must be in the upper half of the array

In either case, we've cut the search size of the array in half. We can repeat this process until we've found k or have found that it is not in the array. Binary search is *exponentially* more efficient than a straightforward search like the functions above. The caveat is that we require the array to be sorted.

Unfortunately, JavaScript does not provide a function to perform a binary search. Though there is no default functionality, we *can* provide our own binary search by creating a "shim" and making it part of the set of functions that belong to *any* array by adding the function to the `Array` object's `prototype`

```
/**
 * Binary Search extension for JavaScript Array
 * Requires that the array be sorted or it may
 * not return correct results.  Takes a comparator
 * function predicate(a, b) and an element item
 * to be searched for.
 *
 * Returns the (an) index i such that
 *   predicate(arr[i], item) === 0
 * Returns -1 if no such value can be found
 */
if (!Array.prototype.binarySearch) {
  Array.prototype.binarySearch = function(predicate, item) {
    if (this === null) {
      throw new TypeError('Array.prototype.binarySearch called on null or undefin
    }
    if (typeof predicate !== 'function') {
      throw new TypeError('predicate must be a function');
    }
    var i = 0;
    var j = this.length-1;
    while(i <= j) {
      var m = Math.floor((i + j) / 2);
      var val = this[m];
      var x = predicate.call(null, item, val);
      if(x < 0) { //item < this[m]
        j = m-1;
      } else if(x > 0) { //item > this[m]
        i = m+1;
      } else {
        return m;
      }
    }
    return -1;
  };
}
```

Sorting Data

Another fundamental operation when organizing data is to *sort* it Sorting is a well-studied problem and there are many different types of sorting algorithms with different properties.

Example: Selection Sort

Selection sort is a simple sorting algorithm that works as follows. First, it iterates through the array to find the minimal element and then places it at the front of the array. It repeats this process for the remaining elements, finding the second smallest, third smallest, etc. and placing each in its proper spot. Example code for sorting an array of integers is provided below.

```
function selectionSort(arr) {
  for(var i=0; i<arr.length-1; i++) {
    var minIndex = i;
    for(var j=i+1; j<arr.length; j++) {
      if(arr[j] < arr[minIndex]) {
        minIndex = j;
      }
    }
    //swap a[i] and a[minIndex]
    var t = arr[i];
    arr[i] = arr[minIndex];
    arr[minIndex] = t;
  }
}
```

Though selection sort is simple conceptually, it is actually one of the least efficient sorting algorithms. There are much better algorithms such as Quick Sort, Merge Sort, and Heap Sort that use a divide-and-conquer strategy or smart data structures (a heap) to sort elements.

Sorting the Correct Way

Any decent programming language will have a built-in sorting function as part of its standard library of functions. In JavaScript, arrays can be sorted using the `sort` function. An example:

```
var animals = ['dog', 'cat', 'gorilla', 'jaguar', 'tiger', 'bat'];
animals.sort();
//animals is now ['bat', 'cat', 'dog', 'gorilla', 'jaguar', 'tiger']
```

However, the default behavior of the `sort` function is to sort elements in lexicographic

order. That is, it sorts them by converting the elements to strings and then sorting them in alpha-numeric order. This leads to some very odd behavior if you attempt to sort numbers or objects.

```
var arr = [2, 6, 1, 23, 4, 3, 1];
arr.sort();
//arr is now [1, 1, 2, 23, 3, 4, 6]
```

This is obviously *not* what we want. Moreover, what if we wanted to sort objects that represented students by last name or GPA, or NUID or some combination of these? What if we wanted to sort elements in decreasing order instead of increasing order?

Comparator Functions

The solution is to provide a *comparator function* to the `sort` function when you call it. A comparator function is a function that takes two elements, a, b and returns

- A negative number if a comes before b (they are in order)
- Zero if a is equal to b
- A positive number if a comes after b (they are out of order)

A comparator function allows us to define an *ordering* without having to worry about the details of sorting.

For example, to properly sort an array of numbers, we could use an anonymous function as follows.

```
var arr = [2, 6, 1, 23, 4, 3, 1];
arr.sort(function(a, b) {
  if(a < b) {
    return -1;
  } else if(a > b) {
    return 1;
  } else {
    return 0;
  }
});
//arr is now [1, 1, 2, 3, 4, 6, 23]
```

One way we could simplify this is to use a simple trick: instead of complex if-else-if statements, we could use simple arithmetic.

```
arr.sort(function(a, b) {
  return (a - b);
});
```

If $a < b$, then the difference, $a - b$ will be negative; if $a = b$ then $a - b = 0$; and finally

if $a > b$ then $a - b$ will be positive. This is exactly the behavior that we want with only one line of code. To sort in a reversed order, we would simply reverse the logic in the comparator function.

```
arr.sort(function(a, b) {
  return (b - a);
});
```

Finally, to sort objects, we can employ logic as complex as needed to order them. Consider the student objects stored in a `roster` array that we've used before. To sort students by last-name/first-name, we could use the following logic: compare the last names and if one comes before the other, return the appropriate value. However, if they have the same last name, we then look at the first name as Amanda Smith should come before Zora Smith.

```
var roster = [...];
/* student object looks like:
{
  firstName: "John",
  lastName: "Doe",
  nuid: "12345678",
  gpa: 3.75
}
*/
roster.sort(function(s1, s2) {
  if(s1.lastName < s2.lastName) {
    return -1;
  } else if(s1.lastName > s2.lastName) {
    return 1;
  else {
    //last names are equal, so order by first names...
    if(s1.firstName < s2.firstName) {
      return -1;
    } else if(s1.firstName > s2.firstName) {
      return 1;
    } else {
      return 0;
    }
  }
});
```

A final note about JavaScript's `sort()` function: it is not *stable*. A sorting algorithm is stable if it preserves the original relative ordering of "equal" elements. That is, if we sort by some field, if a and b have the same value, and a came before b in the original ordering, then a should come before b in the new ordering. An unstable sorting algorithm may not preserve this ordering and may place b before a.

Data Visualization

Organizing data is only the first step in making sense of data. Humans are generally not very good at discerning patterns from raw data. However, when data is represented visually, humans can easily discern patterns, trends, and connections that may not be immediately obvious.

There are several JavaScript libraries that provide functionality to visualize data.

Data-Driven Documents (D3js, http://d3js.org/) provides functionality to bind arbitrary data to the Document Object Model and to apply transitions and interactivity to that data. Similar to jQuery, D3js allows you to apply selectors to DOM objects and modify styles, bind data, etc. It operates on HTML, CSS, and SVG. SVG are emph-scalable vector graphics, graphics that are specified by metadata rather than individual pixels. This allows you to think of a graphic as a collection of objects that interact with each other and that can be modified or *transformed*.

The learning curve for D3js is rather high. However, we have provided several custom examples to get an idea of what is possible. In addition, you may find the following resources useful in learning D3js.

- http://d3js.org/
- https://github.com/mbostock/d3/wiki/Tutorials
- https://github.com/mbostock/d3/wiki
- https://www.youtube.com/user/d3Vienno/videos

GoJS (http://gojs.net/) is another library that provides ways to create interactive data visualization elements.