COSC 2P90 – Assignment 1
Due: Friday, October 4th @5:00pm

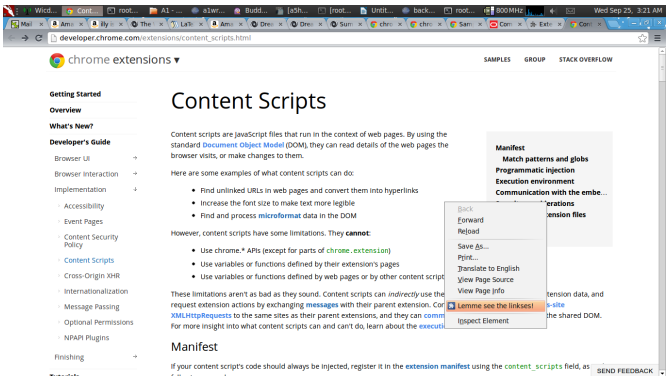
Goals:

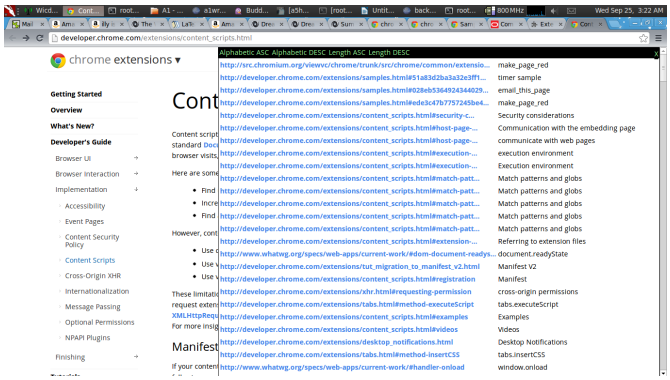
- To promote exploration of a new language (Javascript)
- To learn about comparators
- To bundle up a solution for deployment (make a Chrome extension)
- To gain familiarity with the Document Object Model (DOM)

Prelude:
Since you'll be using Chrome eventually, I'd suggest starting it up and going into the Console for the developer tools (you can get to it by clicking the button beside the 'omnibox', clicking 'Tools', and choosing 'JavaScript Console'; or you can just right-click on nearly any webpage, choose 'Inspect Element', and then click on the 'Console' tab). You can use it to easily test snippets of code. Sure, there are tools that give you command-line Javascript, but, like I said, you'll need Chrome eventually anyhoo.

Post-Prelude:
The idea for this assignment is that you'd be doing most of the research yourself. I'll give you a task, and you learn how to solve it, with relatively little hand-holding. Personally, I'd suggest doing it that way. However, since not everybody is comfortable with that approach (for some reason, *some* people don't like feeling lost, confused and frustrated. Go figure), I've also included a somewhat hand-holdy explanation as well. If that's your preference, don't feel bad using it. However, this guide carries two warnings: First, since I wrote it for *anyone*, I felt compelled to go into quite a bit of detail, which you might find tedious (i.e. this sucker's LONG!). Second, in spite of being so long, this still isn't in any way an actual guide to Javascript, the DOM, or creating extensions.

Your Task:
Your task is simply to write an extension for Google Chrome that adds a context menu (a 'right-click menu') to all webpages that lets you harvest and organize all of the links (`a` anchor tags with an `href` component) to display them in a box that's been added to the top-right corner of the document. The user must be able to click to re-sort the links based on lexicographic order (both ascending and descending), or based on the length of the links (again, both ascending and descending). The user should also be able to dismiss the box.





Background:
Recall for a moment the `Comparable` interface in Java, which indicates that a class may be placed into a total ordering. Or, more explicitly, an instance of that class, when presented with another instance of the same type, can indicate which instance should precede the other according to the ordering (or that they have equivalent ranking).

Though `Comparable` is indeed very useful, it still has one very obvious limitation: it can only impose a single ordering onto its members.

For contrast, let's look at a hypothetical situation:
You have a database of animals, with each animal having both a name and an average weight. If an Animal class were to implement the `Comparable` interface, then you'd only be able to impose one ordering. That ordering could be based on the alphabetical order of the names, the weights, or could even consider both (e.g. sort first by weights, and use lexicographic sequence to break ties), but you'd still have to choose one sequence and stick with it.
But what if you want to sort by weight for some circumstances and alphabetical order for others?
`Comparable` simply does not permit this.

Java does, however, provide another interface: `Comparator` (part of the `java.util` package). A `Comparator` is a separate tool that, when presented with two elements, will compare them to give you an ordering. Ostensibly, it sounds identical to `Comparable`, but with the tedium of having to deal with a separate file, right? Ah, but the catch is you can use one `Comparator` for one ordering, and another `Comparator` for a different ordering *for the same elements*.

Thus, in our animal example, we could have a `LexiComparator` and a `HeftComparator`. Refer to the Java API specifications for additional details on Comparators.

So, that's what we're going to be doing for this assignment, right? ... right?
Erm, nope. Instead, we're going to be making use of the exact same principle, but for a different language and use.

JavaScript
You've almost certainly heard of Javascript, a language primarily (though not exclusively) used for client-side scripts; often in webpages. Javascript has a robust feature set that provides helpful and extensible tools to developers. (blah blah blah: it's got a lot of neat tricks to facilitate being lazy: woot!) For example, its arrays have several handy-dandy built-in functions, including a sort.
If you have an array of values with a natural ordering, you can simply invoke its sort() function to rearrange its elements.
e.g.

```
var mice=Array('Pinky','Brain','Richard Simmons');
mice.sort();
```

That was... pretty easy, right?
However, in this case, it'll sort alphabetically (`["Brain","Pinky","Richard Simmons"]`). What if we don't want that? What if we want to sort by string length (and, while we're being tricky, in descending order)?

Now there's a sticky wicket, innit?

Well, we *could* just do everything manually, but, again, we're assuming that "less work"="good work" (though remarkably close to laziness, simplifying your task and making use of included tools *does* tend to reduce the chance for errors).
The creators of Javascript thought of that, and they included an optional parameter where you can pass in a comparator function.

wait... did I just say that you can pass a FUNCTION as a PARAMETER? THAT'S RIDONKULOUS! Actually, we'll be learning later on that several languages can do this. In this case, it's mostly because Javascript doesn't care what a variable points to (without getting into nitty gritty details, you can mostly think of everything in

Javascript as being a peculiar Object; with some simply being better at their identity than others. Think of it this way and it isn't really much of a stretch).

So, first, we should learn how to write a function in Javascript, shouldn't we?

```
function doSomething() {
  alert('howdy!');
}
```

A few things to note:

1. If you're typing this manually into the console, you don't want to hit 'Enter' after each line. Hit 'Shift-Enter' instead. That lets it know that you're still typing things out. When you're done, *then* hit 'Enter'.
2. Note that I'm inconsistent for how I mark the boundaries of strings. Sometimes I use apostrophes; othertimes quotation marks. That's because javascript is just fine with either. I'm sure you can see reasons for either (e.g. using quotation marks when the string itself contains an apostrophe, or vice-versa). If you like, you can escape characters just like in Java (i.e. 'Earl\'s awesome, eh?').
3. The 'alert' function is a handy way to make a pop-up to display a string. Be especially careful putting it into loops.
4. When you type that in, you then see 'undefined'. This is because, when there's a return from an expression, it outputs it into the console (feel free to play around with that, or ask me about it). Similarly, if you ever forget the contents of a variable, you can simply type its name to see the result (e.g. typing 'mice;' will show you the contents of the array, if you were following along).

Okay, so that's all well and good, but we haven't tried it out yet. Let's, shall we?

```
doSomething();
```

(Are you impressed yet? No? Dernit)
Try typing '`doSomething.`', and then looking at the context menu that pops up. You'll see that functions have several built-in properties (including a `toString()` function! which... also has a `toString()` function... I'd make a 'funcception' joke, but I think I'd prefer to point out that we heard you liked functions in your functions, so we put a function in your function in your function so you can... ... moving on...).

Additionally, you can assign that function to a variable.
e.g.

```
var monkey=doSomething;
```

(by the way, yes, I realize the '`var`' hasn't been necessary for these examples. To avoid a long, rambling diatribe on scope and compatibility and such, can you just humour me and keep using '`var`' to declare variables?)

Great, but now our function is a variable, so how do we run it? Just like the original:

```
monkey();
```

So, you might be wondering what the point is, yes? Well, remember that I said you'd be passing a function in as a parameter; this is to start to show you how the sort function can make use of it. How's about we make that comparator we were talking about? One that lets us sort by length, descending?

```
function diminishingReturns(a,b) {
  return a.length==b.length?0:a.length>b.length?-1:1;
}
```

(You had to know I was going to throw in those irritating ternary conditional operators, right?)
If you don't like how unreadable that is, here's an equivalent alternative:

```
function dr(a,b) {
  if (a.length==b.length) return 0;
  if (a.length>b.length) return -1;
  return 1;
}
```

Let's test it out. We could use alert (and, let's face it, the world could use more lerts), but there's really no need:
`dr("abba","bab");`
should give -1.

`dr("abb","bab");`
should give 0.

`dr("ab","bab");`
should give 1.

(If that seems slightly backwards, remember that we use a negative value to indicate that the first parameter should *precede* the second parameter, etc.)

So, remember our mice?
```
var mice=Array('Pinky','Brain','Richard Simmons');
mice.sort(dr);
```

Note that we get it in order of length, descending, just as we wanted.
Also note that "Pinky" is still before "Brain", as it was in the original. I'd love to be able to tell you that Javascript only uses stable sorts, but unfortunately that varies by implementation. Chrome's, for example, unfortunately, does not (well, it both does and does, if you're familiar with methods of speeding up QuickSort via Insertion Sort for small cases, but short version: not stable).
Unfortunately, this means that, unless we write our own (stable) sorting function, we can't do *subsorts*.

Anyhoo, it should be easy by now to see how powerful this is. Want to sort by length, decreasing? Pass in a function that does that for just two elements. Want it to sort by length, but ascending? Rewrite that function, but with flipped signs. And, of course, you can come up with all sorts of other criteria.

INSANE JAVASCRIPT MOMENT:
```
dr.cheesePreference='Roquefort';
diminishingReturns.cheesePreference='Jarlsberg';
```

There. My two comparators now have preferences for cheese.
`alert(dr.cheesePreference);`

Yup. Definitely does.
(Why is this remotely helpful? If you do ever get interested in complicated, 'real' Javascript work, then it can be helpful for all sorts of things. Refer to the included reference for Advanced JavaScript to see examples like caching results of previous function calls to speed up future invocations)

RECAP (midcap?)
Okay, so now we know a smidgen about Javascript. But how does that help us with our task? Our remaining two big concerns are making a Chrome Extension and analyzing/manipulating a webpage. Since it's relatively pointless to try making an Extension that doesn't do anything yet, let's move on to the latter.

HTML documents have evolved quite a bit over the years (or, possibly more importantly, our representations of them have). Load up a webpage, right-click and choose 'Inspect Element' again. Depending on where you

clicked, the inspector will jump to a corresponding location in the document. But notice that it's showing a hierarchy. What's more, you can collapse and expand elements to see/hide their contents.
An ordered list (`ol`), for example, will have all of the list items (`li`) placed at higher depth in the tree, off of the ordered list. Collapse the list, and all of the items get hidden. (Side note: If you hover your cursor over different elements, Chrome should shade their corresponding items in the actual web document. If you click on an element in the inspector and hit 'delete', you can remove items from the page. It's like a craptacular AdBlock!)
This is what we call a DOM tree (DOM=Document Object Model). I mention this only to aid your Google-Fu.

How does this help us? Well, if you aren't already there, scroll to the top of the inspector. You should see the html tag that holds the entire document (note: depending on the page you're viewing, and whether or not it uses frames, you might actually see multiple documents, but keep it simple for now). If it isn't already, expand the html tag to see the head and body tags. The head is where you'll typically see things like the title, links to CSS, etc. It isn't particularly helpful for this exercise, so let's look at the body instead.
The body is where we have the actual displayable content. If you click on something and look to the right, you should see the CSS rules that apply to that element (both directly, and inherited from parent elements). We'll probably need to make use of that eventually, but not yet.
If you scroll down, you'll probably eventually see some 'anchor' tags (`<a></a>`). Anchor tags are used for several things; two of the biggies are links to other pages (with `href`), and to mark destinations for bookmarks (no, not bookmarks to pages; those little #dealie tags you often see in URLs). (Of course, as mentioned earlier, if you want to jump straight to an anchor tag, you can just right-click a link, and choose to 'Inspect Element'. You should be taken directly to it)

Since our final goal is to be able to grab information about all of these anchors, we're going to need an automated way of inspecting these anchors. BUT THEY'RE ALL OVER THE PLACE!
No problem. Back to our Javascript console...
```
document;
```

Notice that it shows what appears to be a document, but... with no contents? Oh, hey! One of those collapsey-expandy dealies! Click it!
Look familiar?
Just for ha-ha's, also try typing '`document.`' and seeing what pops up.
Quite a bit, eh? Everything from properties, to methods, to event callbacks (e.g. `onkeypress`, which is triggered when you... I don't need to explain this one, right?).

One small distinction to note: this 'document' corresponds to the entire structure; not the visible text. Note that you need to go in a level or two to see anything interesting.

So, how can we access, say, the html tag within it? Maybe try this?
```
document.html;
```

Dernit. Nope. That wasn't helpful.
Well... we said this was a tree, right? And this tree has elements (or nodes) that are deeper in... Oh! Children!
```
document.childNodes;
```

Hey! We got something! Note that it's an array of one or more elements! (For mine, it's a DOCTYPE, some commented-out details, and then the html subtree)
If your html tag happens to be the first child, you could just say:
```
document.firstChild;
```

And you can get the second child thusly:
```
document.firstChild.nextSibling;
```

However, like I said, the children are in an array. So, for me, I needed the third child:
```
document.childNodes[2];
```

Now, depending on what I wanted to do, I could do a lot of interesting things by following these links further and further (changing behaviours, adding/removing elements, changing CSS, etc.). To make things simpler, I might make variables to mark important subtrees. For example, for the page I happened to be viewing:
```
var body=document.childNodes[2].childNodes[2];
```

But there's a problem, isn't there? This is all fine and good for doing everything manually, but there's no way any of that code could be appropriate for general use. Heck, I doubt it would make much sense for the pages half of the class happened to choose to test on!

But that's okay. In this case, we're looking for something very specific, aren't we? It might not *seem* specific (after all, we don't know how many anchor tags there'll be, what they'll contain, etc.), but we *specifically* want to apply some sort of behaviour on each anchor tag. This means that, were we to have some sort of data structure containing *only* anchor tags, we could very easily do some inspection/manipulation (yes, I realize we could also write an algorithm that manually traversed each branch instead, but note the key word, "easily"). Well, we're in luck, because our document object has multiple functions for that sort of use!
Try just typing '`document.get`', and seeing the suggestions. We could grab a particular element based on its '`id`' (`id`'s are *supposed* to be unique in each document, so they don't include a function for grabbing all of them with the same id); that's neat, and generally helpful for Chrome extensions, but less-helpful for what we want. Ooh, what's this? `document.getElementsbyTagName` (there's also a `getElementsByTagNameNS`, but that's for searching within particular namespaces; it's neat, but not immediately useful).

Let's try it out!
```
document.getElementsByTagName("a");
```

Whoa! Depending on the page you chose, that could be quite a few, eh? Let's try hanging onto those for a moment:
```
var allAnchors=document.getElementsByTagName("a");
```

That certainly seems handy, eh? Unfortunately, depending on the page you chose, you might have some anchors that aren't really links to other pages (e.g. used for bookmarks). The only ones we want are those with `href` tags.

This is probably a good time to mention something. allAnchors isn't actually an array. We can access its contents like one, but it isn't really one (I muddled the terms a bit above, to avoid throwing too many things at once at you). `allAnchors` is an object, but not an array. What's the difference? Well, first off, we can't sort it. BUT HOW ARE WE GOING TO BE ABLE TO USE COMPARATORS?!?!?

There are two approaches. The elegant approach would be to manually invoke Array's slice function on the object, which could certainly create the necessary array. However, we'd still have the problem that some elements aren't actually links (even if this isn't the case for your test document, we still can't assume that'll always be true). So, let's do this old-school *and* kill two birds with one stone, eh?

```
var linkAnchors=Array();
for (var i=0;i<allAnchors.length;i++) {
  if (allAnchors[i].href)
    linkAnchors.push(allAnchors[i]);
}
```

Now, if we look at `linkAnchors`, we'll see that it can do everything an array can do *and* it only contains those anchors that had links! We're certainly getting closer.
(By the way, just as an aside, if you'd like an example of a page with anchors that don't have links, check this out: http://docs.oracle.com/javase/7/docs/api/java/util/Comparator.html)

As a final thought before we continue, we might want to filter out even more links. Some links might not have any text at all (e.g. for buttons), some might or might not have titles, etc. For this exercise, I'd suggest just accepting that some of our results might look mildly silly.

Now, all we need to do is to sort them.
So, how do we sort? We can start by, say, ascending length of link addresses (href)?
Let's write a comparator:

```
function hrefSizeAsc(a, b) {
  return a.href.length==b.href.length?0:a.href.length<b.href.length?-1:1;
}
```

Notice that I'm now starting to use more helpful names for things. That's because we're getting closer to the final product.
Just to verify that it works:

```
linkAnchors.sort(hrefSizeAsc);
```

Typing:

```
linkAnchors[0].href;
```

followed by

```
linkAnchors[1].href;
```

should show that the second link is longer than (or equal in length to) the first link.

So, now we have an array of links. It might not seem as though we've accomplished much, but we really have. For the sake of not having to look around to copy&paste, let's just assemble everything we have so far together:

```
function hrefSizeAsc(a, b) {
  return a.href.length==b.href.length?0:a.href.length<b.href.length?-1:1;
}
function grabLinksFromPage() {
  var allAnchors=document.getElementsByTagName("a");
  var linkAnchors=Array();
  for (var i=0;i<allAnchors.length;i++) {
    if (allAnchors[i].href)
      linkAnchors.push(allAnchors[i]);
  }
  linkAnchors.sort(hrefSizeAsc);
  return linkAnchors;
}
```

Try loading up a different page, entering all of that into the console, and then testing it out:

```
var anchors=grabLinksFromPage();
```

We could choose a different sorting criteria now (e.g. alphabetical order, based on an anchors `.innerText`), but let's move on to the next step.

Remember how we said we can add elements to the document? Well, let's do that!

```
var anchorBox=document.createElement("DIV");
document.body.appendChild(anchorBox);
anchorBox.innerText='hi';
```

Scroll down to the bottom of the page, and you should see your new box. It won't look very boxy yet; it's just the text, 'hi'.
First things first, let's move it up to the top of the page. Since we're now talking about layout, style seems to be the obvious thing to modify.
If you type '`anchorBox.style.`' and look at the options, you'll see that there are quite a few. Luckily, we don't need very many of them. (WARNING! WARNING! To any web developers in the class, this is going to be teeerrible! We're going for "get'er done", not "follow the standards perfectly to present a modular and compliant implementation")

```
anchorBox.style.zIndex='1000';
anchorBox.style.top='0px';
anchorBox.style.right='0px';
anchorBox.style.position='absolute';
```

Scroll to the top, and you should see a little 'hi' in the upper-right corner. Note, if we chosed a position of 'fixed' instead of 'absolute', then it would *always* be in the upper-right corner (i.e. it wouldn't scroll with the document). That could potentially be neat, but would get irritating fast if there were more lines of links than could fit on a single page.

```
var linkTable=document.createElement("TABLE");
linkTable.id='tooPeaNinety';
anchorBox.appendChild(linkTable);
anchorBox;
```

Note: that last line is just to inspect the current state of our `DIV`. Note that we can see the text, followed by the table (we don't see the table on the page yet because it still doesn't have anything inside it).
Let's sidetrack for a moment just to verify that we're really doing anything:

```
var test=document.createElement("TR");
linkTable.appendChild(test);
test.appendChild(document.createElement("TD"));
test.firstChild.innerText="Hello!";
```

You should see the text now. Note that it's entirely possible (likely) that the 'table' won't actually have a border. Now's as good a time as any to remedy that:

```
linkTable.style.borderStyle='solid';
linkTable.style.borderWidth='1px';
linkTable.style.borderColor='black';
linkTable.style.backgroundColor='white';
```

There we go! At least now it's a box, which is kinda sorta vaguely table-y!
Since we now have an actual table to work on, that 'hi' isn't really helpful. Let's remove it:

```
anchorBox.removeChild(anchorBox.firstChild);
```

If you aren't familiar with HTML, `TR` stands for 'Table Row', and `TD` stands for 'Table DCell, but the D is silent' (Actually, it stands for 'Table Data', but I like mine better).
So, we know how to add rows to the table, and we know how to add cells to a row. Theoretically, we should now have all of the tools we need to automagically insert entries into the table from an array.
Of course, we should first probably remove our test work:

```
linkTable.removeChild(linkTable.firstChild);
```

This is where you try writing it yourself, and I do the same, and we compare results. (No peeking!)

You're done?

Great! Let's compare! Here's what I have:

```
function appendLink(target,anchor) {
  var row=target.insertRow(-1);
  var cell=row.insertCell(-1);
  cell.innerHTML="<a href='"+anchor.href+"'>"+(anchor.href.length>72?
     anchor.href.substring(0,70)+"...":anchor.href)+"</a>";
  var cell=row.insertCell(-1);
  cell.innerText=anchor.title?anchor.title:anchor.innerText?anchor.innerText:
     (anchor.href.length>72?anchor.href.substring(0,70)+"...":anchor.href);
}
function populateTable(target,elements) {
  for (var i=0;i<elements.length;i++) {
    appendLink(target,elements[i]);
  }
}
populateTable(linkTable,anchors);
```

(Note that I tended to limit the lengths of displayed strings. This was arguably unnecessary, but does still prevent the row widths from getting terribly long)

So, we have now figured out how to retrieve all of the anchors, sort them according to some (replaceable) comparator, and insert a report of the result into the document. This is probably a good time to point out that the style of the source page will be inherited by the components. That is, if the page you're analyzing has a silly font, then your new table may also have a silly font. The same goes for colour, etc. An easy solution would be to change the style of each element as you're adding them. You can even set their style attributes to 'initial' to revert to whatever the 'default' style would have otherwise been, if you don't want to choose one of your own. I'll leave it up to you to decide if you wish to do this or not.

The next step should probably be to write the other comparators:

```
function hrefSizeDesc(a, b) {
  return a.href.length==b.href.length?0:a.href.length>b.href.length?-1:1;
}
function lexicoAsc(a,b) {
  var aText=a.title?a.title:a.innerText?a.innerText:a.href;
  var bText=b.title?b.title:b.innerText?b.innerText:b.href;
  return aText==bText?0:aText<bText?-1:1;
}
function lexicoDesc(a,b) {
  var aText=a.title?a.title:a.innerText?a.innerText:a.href;
  var bText=b.title?b.title:b.innerText?b.innerText:b.href;
  return aText==bText?0:aText>bText?-1:1;
}
```

If you've been following along step-by-step, then instead of having to reload and re-enter all of the code, you can simply test this way:

```
anchors.sort(lexicoDesc);
linkTable.innerHTML=null; //(or "" or '')
populateTable(linkTable,anchors);
```

We're making some great progress here. But we'd still like to be able to re-sort without having to manually clear it out and invoke it. Of course, considering how simple it is to do it manually, we can be pretty confident that it won't be terribly difficult to automate it, either.
If we wanted, we could add a header to the table (there are several easy ways to do this). However, since we'd then have to deal with merging cells to keep everything lined up and such... why not just add something before the table itself?

```
var lexAscButton=document.createElement("SPAN");
var lexDescButton=document.createElement("SPAN");
var lengthAscButton=document.createElement("SPAN");
var lengthDescButton=document.createElement("SPAN");
lexAscButton.style.color=lexDescButton.style.color=lengthAscButton.style.color
     =lengthDescButton.style.color='lightgreen';
lexAscButton.style.backgroundColor
     =lexDescButton.style.backgroundColor=lengthAscButton.style.backgroundColor
     =lengthDescButton.style.backgroundColor='black';
lexAscButton.style.margin=lexDescButton.style.margin
     =lengthAscButton.style.margin=lengthDescButton.style.margin='3px';
lexAscButton.innerText='Alphabetic ASC';
lexDescButton.innerText='Alphabetic DESC';
lengthAscButton.innerText='Length ASC';
lengthDescButton.innerText='Length DESC';
anchorBox.insertBefore(lexAscButton,linkTable);
anchorBox.insertBefore(lexDescButton,linkTable);
anchorBox.insertBefore(lengthAscButton,linkTable);
anchorBox.insertBefore(lengthDescButton,linkTable);
```

Now, I think I know what you're thinking: "But, Earl, that's butt-ugly!"
Why yes. Yes, it is.
You're welcome to do better. This is a demonstration, not a design class.
What we want to be able to do is to click one of those buttons and have it automatically re-sort the list.
Note that, since we always retain an array of the actual data, there's really no need to hang onto any elements of the table itself between sortings, so we'll just be dropping the table's contents each time (just as we did with the sample code above).
Those buttons are going to trigger that code, but to do so we'll need to make use of *events*. In this case, '`onclick`' events. Before we get to the real attempt:

```
lexAscButton.onclick=function(){alert('hi');};
```

Now, whenever we click on the "Alphabetic ASC" button, it will execute that "anonymous function" to generate a mildly annoying popup!

Even though the code for resetting is only three lines long, I'd still feel more comfortable turning it into a function:

```
function resortTable(anchors,table,comparator) {
  anchors.sort(comparator);
  table.innerHTML=null;
  populateTable(table,anchors);
}
lexAscButton.onclick=function(){resortTable(anchors,linkTable,lexicoAsc);};
```

There's still a missing element here; something basic. Choosing to display all of the links on a page doesn't mean you're necessarily done with that page yet. However, once the links are displayed, there's no convenient method of dismissing the box.
This is actually pretty easy to do. How do you normally close a window in typical software? With the close button! So, why not just add a close button?

```
var closeButton=document.createElement("SPAN");
closeButton.style.color='lightgreen';
closeButton.style.backgroundColor='black';
closeButton.style.margin='3px';
closeButton.style.right='0px';
closeButton.style.position='absolute';
closeButton.innerText='X';
anchorBox.insertBefore(closeButton,linkTable);
closeButton.onclick=function(){abolishAnchorBox();};
```

There we go! A bit basic, but it works. However, even though we can do what we want now, there's still one slight quirk with the functionality. Try displaying the box of links, and then try doing it again. i.e. try running the script twice in a row.
We end up with two boxes, with one of those boxes being twice the size (which isn't surprising, since it also harvested links from the previous popup box). This one's incredibly easy to fix. Notice that, for the last part, we wrote a function that only tries to remove the box if it's really there? Kinda odd way to phrase it, since we were having it triggered by a close button on the box (that is, how could it *not* be there, if it was necessary for running it?). That was actually phrased that way for this part. The solution is simple: Before doing anything else for the code, simply run that function. If this is the first execution (i.e. there isn't already a box of links up), then it simply won't do anything.

If you grab all of the pieces we had above (ignoring things like adding 'hi' and other things we undid), you'll see that we actually have sufficient code for a working solution.
In fact, it'd probably look something like this (more or less):

```
//Cleanup
function abolishAnchorBox() {
  if (document.getElementById('tooPeaNinety')) {
    document.body.removeChild(document.getElementById("tooPeaNinety")
           .parentNode);
  }
}
abolishAnchorBox();

//===COMPARATORS===//
function hrefSizeAsc(a, b) {
  return a.href.length==b.href.length?0:a.href.length<b.href.length?-1:1;
}
function hrefSizeDesc(a, b) {
  return a.href.length==b.href.length?0:a.href.length>b.href.length?-1:1;
}
function lexicoAsc(a,b) {
  var aText=a.title?a.title:a.innerText?a.innerText:a.href;
  var bText=b.title?b.title:b.innerText?b.innerText:b.href;
  return aText==bText?0:aText<bText?-1:1;
}
function lexicoDesc(a,b) {
  var aText=a.title?a.title:a.innerText?a.innerText:a.href;
  var bText=b.title?b.title:b.innerText?b.innerText:b.href;
  return aText==bText?0:aText>bText?-1:1;
}
//===END COMPARATORS===//

function grabLinksFromPage() {
  var allAnchors=document.getElementsByTagName("a");
  var linkAnchors=Array();
```

```
  for (var i=0;i<allAnchors.length;i++) {
    if (allAnchors[i].href)
      linkAnchors.push(allAnchors[i]);
  }
  linkAnchors.sort(hrefSizeAsc);
  return linkAnchors;
}

var anchors=grabLinksFromPage();

var anchorBox=document.createElement("DIV");
document.body.appendChild(anchorBox);

anchorBox.style.zIndex='1000';
anchorBox.style.top='0px';
anchorBox.style.right='0px';
anchorBox.style.position='absolute';
anchorBox.style.backgroundColor='black';

var linkTable=document.createElement("TABLE");
linkTable.id='tooPeaNinety';
anchorBox.appendChild(linkTable);

linkTable.style.borderStyle='solid';
linkTable.style.borderWidth='1px';
linkTable.style.borderColor='black';
linkTable.style.backgroundColor='white';

function limitLength(s) {
     return s.length>72?s.substring(0,70)+"...":s;
}

//Adds a single anchor's entry to a target table
function appendLink(target,anchor) {
  var row=target.insertRow(-1);
  var cell=row.insertCell(-1);
  cell.innerHTML="<a href='"+anchor.href+"'>"+limitLength(anchor.href)+"</a>";
  var cell=row.insertCell(-1);
  cell.innerText=anchor.title?limitLength(anchor.title):
     anchor.innerText?limitLength(anchor.innerText):limitLength(anchor.href);
}

//Populates a specified table (target) with elements (elements)
function populateTable(target,elements) {
  for (var i=0;i<elements.length;i++) {
    appendLink(target,elements[i]);
  }
}
populateTable(linkTable,anchors);

var lexAscButton=document.createElement("SPAN");
var lexDescButton=document.createElement("SPAN");
var lengthAscButton=document.createElement("SPAN");
var lengthDescButton=document.createElement("SPAN");
lexAscButton.style.color=lexDescButton.style.color=lengthAscButton.style.color
     =lengthDescButton.style.color='lightgreen';
lexAscButton.style.backgroundColor=lexDescButton.style.backgroundColor
```

```
      =lengthAscButton.style.backgroundColor
      =lengthDescButton.style.backgroundColor='black';
lexAscButton.style.margin=lexDescButton.style.margin
      =lengthAscButton.style.margin=lengthDescButton.style.margin='3px';
lexAscButton.innerText='Alphabetic ASC';
lexDescButton.innerText='Alphabetic DESC';
lengthAscButton.innerText='Length ASC';
lengthDescButton.innerText='Length DESC';
anchorBox.insertBefore(lexAscButton,linkTable);
anchorBox.insertBefore(lexDescButton,linkTable);
anchorBox.insertBefore(lengthAscButton,linkTable);
anchorBox.insertBefore(lengthDescButton,linkTable);

var closeButton=document.createElement("SPAN");
closeButton.style.color='lightgreen';
closeButton.style.backgroundColor='black';
closeButton.style.margin='3px';
closeButton.style.right='0px';
closeButton.style.position='absolute';
closeButton.innerText='X';
anchorBox.insertBefore(closeButton,linkTable);
closeButton.onclick=function(){abolishAnchorBox();};

function resortTable(anchors,table,comparator) {
  anchors.sort(comparator);
  table.innerHTML=null;
  populateTable(table,anchors);
}
lexAscButton.onclick=function(){resortTable(anchors,linkTable,lexicoAsc);};
lexDescButton.onclick=function(){resortTable(anchors,linkTable,lexicoDesc);};
lengthAscButton.onclick=function(){resortTable(anchors,linkTable,hrefSizeAsc);};
lengthDescButton.onclick=function()
{resortTable(anchors,linkTable,hrefSizeDesc);};
```

Of course, this still requires that we manually run the code on each page. That's a pain, not to mention not our goal.

Chrome Extensions

Like any browser worth its salt (or salt substitute), Google Chrome can easily have its functionality expanded through the addition of extensions. Extensions may be used for a great number of tasks, from modifying a page, to retrieving and/or storing resources from other pages, to even simple games.

Depending on the use, there are numerous different ways to have an extension operate on or generate page data. Three of the big ones (that we probably won't be using here) are *Browser Actions*, *Page Actions*, and *Content Scripts*. I'd strongly encourage you to look up all three of those, because they're very useful for realizing what you can do with extensions. If our desired outcome were only minutely different, we'd probably be using a browser action (with a button on the toolbar) that would just present a separate popup. However, we'll just be making a context menu item to create a `DIV`. We'll use a *Background Script* to actually create it, and to define and assign the necessary code to be triggered whenever the menu item is chosen.

Background Pages and Background Scripts are also things you should consider researching on your own, but, basically, a background script is something that automatically executes, without requiring a page for context. It's best used for initializing connections to the browser's elements, establishing callbacks, coordinating other components, etc.

You are *very* strongly encouraged to review Google's Developer tutorials for extensions. However, as with the previous parts, you can simply follow along here if you prefer.

Before you do anything else, create a folder that will *only* contain files for this extension.

Chrome Extensions are pretty simple in terms of their components. They contain their necessary script files/pages, resources (e.g. images for icons), and also what's known as a *manifest*.
A manifest for an extension is very similar to a manifest for a Java .jar. It mostly just documents the extension and identifies the roles of included resources. Let's look at a really simple one:

```
{
  "name" : "ContextualScraper",
  "version" : "0.0.0.1",
  "description" : "Collect ALL THE LINKS! from the current page",
  "background" : { "scripts": ["background.js"] },
  "permissions" : [
    "contextMenus",
    "tabs",
    "http://*/*",
    "https://*/*"
   ],
  "minimum_chrome_version" : "6.0.0.0",
  "manifest_version": 2
}
```

This needs to be in a file called `manifest.json` that needs to be in the folder we created earlier. Let's look at the individual components here:
`name` — This should be obvious
`version` — Ditto
`description` — You'll see this message when managing your extensions within Chrome
`background` — Here's where we start with the 'real' stuff. There are two basic types of background item: a page or a script. A page can be handy if you need to create a .html file that can be brought up later. A script can be useful for registering for event handlers, etc.
`permissions` — Chrome is relatively security-conscious. The basic idea is that letting an extension access content across domains may be a good or bad idea, depending on the code, but that the user is entitled to choose whether or not to let that happen. In this case, we're reserving the right to manipulate the context menu, to access tabs, and to access content on pretty much any website.
`minimum_chrome_version` — This is the number of pelicans I had for breakfast today. (seriously, this one's also easy to identify)
`manifest_version` — If you're using a recent version of Chrome (and you should be), it won't let you pack an extension using an older manifest version (if you're learning all of this for the first time now, then this doesn't really affect you much).

Since we've promised a script called background.js, let's make that!

```
function getClickHandler(info,tab) {
  return function(info, tab) {
    alert('howdy');
  };
};
```

```
chrome.contextMenus.create({
  "title" : "Lemme see the linkses!",
  "type" : "normal",
  "onclick" : getClickHandler()
});
```

When we're creating our context menu, note that we need to both give it something to display (its title), and something to do when chosen. You probably don't need to do it as convoluted as I did, but it really doesn't hurt. When the user chooses to 'see the linkses', a function is called that returns an anonymous function to be executed. Before we get any further, let's try this out (yes, even though we know it won't do anything helpful).

Remember that you should have a folder with two files: `manifest.json` and `background.js`.
Load up Chrome (if you haven't already), choose Tools and click Extensions (or simply type 'chrome://extensions' into the address bar).
At the top, towards the right, you should see a checkbox for, "Developer mode". Check that box. You should now be able to "Load unpacked extension". Click that and choose the folder mentioned above.
Now, when you go to a page and right-click the document, you should be able to see the new context menu item. Try it out!

Great, so we can definitely execute script. But that isn't the script we want. We *could* try using the code that we wrote in the previous section, but unfortunately that won't work. Why not? Let's take a look:

```
function getClickHandler(info,tab) {
  return function(info, tab) {
    dealie();
  };
};

function dealie() {
     alert(document.location.href);
}
```

If you go back to the Extensions tab, you should notice a small 'Reload' link at the bottom of your extension. Click that. Now, once again, try choosing the context menu item on some webpage.

Uh oh. That's a problem. That's what it thinks the document's URL is. So, if that's what it thinks the address is, you can probably guess that it doesn't know the contents of the page we want, either. That is, since it's running from within the context of the extension, that's also the 'document' it's trying to use! What we need to do is to trick the real document into running arbitrary code for us.

```
function getClickHandler(info,tab) {
  return function(info, tab) {
    chrome.tabs.executeScript(tab.id,{
          code: 'dealie();'
    });
  };
};

function dealie() {
     alert(document.location.href);
}
```

Save, reload, etc. Try it out!
... ah crud. Didn't work, did it? If you bring up the Javascript console again, you'll see the problem: It's never heard of dealie! But, wait, this means that, before, we could run the code since it was in our background script,

but couldn't access the document because it was in another context. Now, we can probably access the document, but we can no longer access our background script because now *that*'s in the other context? Ah, but there's a (very lazy) solution!

```
function getClickHandler(info,tab) {
  return function(info, tab) {
    chrome.tabs.executeScript(tab.id,{
      code: 'dealie='+dealie+';dealie();'
    });
  };
};

function dealie() {
     alert(document.location.href);
}
```

There we go! Now we can inject code we want, to be executed in the context we want!
So, instead of '`dealie`', choose something more appropriate/cool (I chose `omniFunc`). Then, simply take all of the code from the previous part, and place it inside that function! (Yes, I'm intentionally not showing the final code, so you can't just copy&paste from the end of this guide with no reading at all)

Now that we have it all working, there's just one last trivial step left: Actually turning it into a distributable extension. This part's easy. On Chrome's Extension page again, choose "Pack extension...". For the Extension root directory, just pick the same folder you've been using for testing. You don't need to worry about a key. Click "Pack Extension", and it should have created a .crx file (the actual extension).
In my case, it made `WithContextMenu.crx`. Remove/Uninstall the testing version of your extension (click the garbage can beside it), and drag the .crx file from your file browser back onto your Extension page. Note that it asks you to confirm the permissions you're granting. Click Add.

And that's it! You're done!

Submission:
Electronic: Put all of your files into the same folder. Use this folder for the submission script: **`submit2p90`**
Don't forget to include the .crx file as well!
Physical: Take a screenshot of your work in action. Print out all of your code (including your manifest). Fill out a departmental cover form and staple it to the front. Submit your assignment into the appropriate departmental dropbox.

Additional Reading:
http://www.w3schools.com/jsref/
http://ejohn.org/apps/learn/
http://developer.chrome.com/extensions/getstarted.html
http://developer.chrome.com/extensions/contextMenus.html
http://hangar.runway7.net/javascript/guide
http://developer.chrome.com/extensions/samples.html#contextMenus