

Due by Tuesday, March. 26 at 11:00 PM. 25 total points

**1** (15 points). For this problem, you need to know a little about the DOM. If an element in our document has an `id` attribute with value, say, "v", then we get a reference to an object corresponding to the element (and assign it to a variable `obj`) with

```
obj = document.getElementById("v");
```

We can then assign the element content by assigning to the object's `innerHTML` attribute, e.g.,

```
obj.innerHTML = "An example";
```

This goes between the opening and closing tags of the element (hence "inner"), and it may itself contain markup. We can append content to the end of what is already there with statements such as

```
obj.innerHTML += " of dynamic content";
```

As an example, consider the following HTML document and JavaScript script.

```
<html>
<head>
  <meta charset="utf-8">
  <title>Example for Assignment 3, Problem 1</title>
  <script type="text/javascript" src="problex.js">
  </script>
</head>
<body>
  <p id="pid">The number is </p>
  <table id="tid" border="1" width="50%">
    <caption>The table</caption>
    <table>
  <p onclick="start()">Go!</p>
</body>
</html>
```

```
function start()
{
  var pobj = document.getElementById( "pid" ),
      tobj = document.getElementById( "tid" );

  pobj.innerHTML += 3;
  tobj.innerHTML += "<tr><td>1</td><td>2</td></tr>"
                  + "<tr><td>3</td><td>4</td></tr>";
}
```

The first `p` element has `id` "pid" and already has some content. The `table` element has `id` "tid" and already has as content a `caption` element. The `p` element is rendered as **Go!**, which, when clicked, results in function `start()` being called. This function gets references to the objects for the first `p` and the table. It appends "3" to the content of the `p` and two rows (with two cells each) to the content of the table (after the caption). When the page is first loaded, it is rendered as shown in the first screenshot at right. After **Go!** is clicked, it is rendered as shown in the second.

**The number is**

**The table**

**Go!**

**The number is 3**

**The table**

1	2
3	4

**Go!**

For this problem, you are given the following HTML document.

```
<html>
<head>
  <meta charset="utf-8">
  <title>Assignment 3, Problem 1</title>
  <script type="text/javascript" src="probl.js">
  </script>
</head>
<body onload="start()">
  <table id="input" border="1" width="50%">
    <caption>The Input Array</caption>
  </table>
  <p id="sum">The sum of all elements is </p>
  <table id="rowS" border="1" width="50%">
    <caption>Row Sum</caption>
  </table>
  <table id="colS" border="1" width="100px">
    <caption>Column Sum</caption>
  </table>
</body>
</html>
```

Once the document is loaded, it calls function `start()`, defined in `probl.js` as

```
function start()
{
  var rows = prompt( "Number of rows", 1 ),
      cols = prompt( "Number of columns", 1 ),
      inAr;

  inAr = inputArray(rows, cols);
  sumAll( inAr );
  rowSum( inAr );
  colSum( inAr );
}
```

You write the functions `inputArray()`, `sumAll()`, `rowSum()`, and `colSum()`. Function `inputArray()` is passed the number of rows and columns in the 2D array it builds by prompting the user for integers. (You don't have to verify input.) It returns this array, but, before doing so, it constructs a string that represents the array as table rows (`tr` elements containing `td` elements) and appends this to the content of the first `table` element (just after the

**caption** element). This requires a nested loop. You can get by with one nested loop by constructing the string as the values are obtained from the user.

Function `sumAll()` is passed the array, sums all the values (across all rows and columns), and concatenates this sum to the end of the content of the `p` element.

Function `rowSum()` adds the rows together (i.e., forms the sum of each column) and concatenates a `tr` whose `td`'s contain the column sums to the end of the content of the second table (after the caption). Similarly, `colSum()` adds the columns together (i.e., forms the sum of each row) and concatenates to the end of the content of the third table a sequence of `tr`'s, each with one `td`, which contains the sum of the respective row. Neither of these functions requires an additional array; just form the string as you produce the sums.

As an example, the following shows the rendering when the user answered 2 for the number of rows and 3 for the number of columns and then entered in the prompts 1, 2, 3 (giving the first row), and 4, 5, 6 (for the second row).

The Input Array		
1	2	3
4	5	6

The sum of all elements is 21

Row Sum		
5	7	9

Column Sum

6
15

2 ( 10 pts.). For this problem, you will maintain an associative array whose keys are people's names. An associated value is an object with three properties: **deposits** (an array of the person's deposits), **withdrawals** (an array of the person's withdrawals), and **balance** (the person's current balance). We assume that an account starts out with a zero balance so that the current balance is the sum of the deposits minus the sum of the withdrawals.

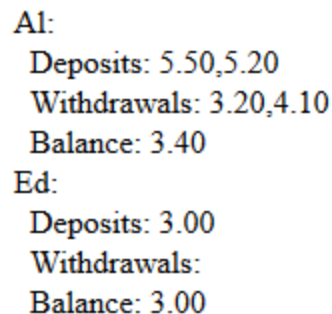
Write an HTML document **prob2.html** that loads a JavaScript file **prob2.js**. When the body is finished loading, function **go()** is called. The body of the document consists of an empty **div** element with **id** value **"recs"**.

Function **go()** in **prob2.js** first calls function **init()**, which prompts for names until the user clicks **Cancel**. Each name becomes a key in an associative array, and the associated value is an object with its **deposits** and **withdrawals** properties initialized to **[]**, and its **balance** property initialized to 0. It returns this associative array, which is assigned to, say, **recs** in **go()**.

Function **go()** next calls **update()**, passing it **recs**. This function loops through round after round as long as the user clicks **OK** (not **Cancel**) in the confirmation box asking about another round. In each round, the function loops over all the names of people. For each person, it asks for their deposit and their withdrawal for that round. The user enters "0" if no deposit or no withdrawal was made. As long as the deposit is not "0", it is added to the end of the array that is the value of the **deposits** property; similarly, the withdrawal is added to **withdrawals**. (An easy way to add something to the end of an array is to use the array's **push()** method. If **ar** is an array and **x** is some value, **ar.push(x)** adds **x** to the end of **ar**, increasing its length by 1.) The person's balance is also updated each round.

Finally, **go()** calls **outRecs()**, passing it **recs**. This function assigns to, say, **recsRef** a reference to the **div** element (with **id "recs"**). It constructs in, say, **recStr** a string with all the information in associative array **recs**, and this string is assigned to the **innerHTML** property of **recsRef**. For each element of **recs**, the string has one line with the name followed by a **':'**. The next three lines have, respectively, the array of deposits, the array of withdrawals, and the balance for the person named. Each of these three lines begins with two spaces. (Use the character entity reference **&nbsp;** to have the browser insert a space.) Use the **toString()** method of the arrays of deposits and withdrawals to get a quick string representation of the contents of these arrays; do not spend time on appearances. (The contents of these arrays will be strings that have not been converted to numbers. These strings are converted to numbers when the balance is updated since that involves subtraction.)

The screenshot at right shows the output when names "Al" and "Ed" were supplied, and the update went through two rounds. In the first round, Al deposited 5.50 and withdrew 3.20 while Ed deposited 3.00 and withdrew nothing. In the second round, Al deposited 5.20 and withdrew 4.10 while Ed neither deposited nor withdrew.



```
Al:
  Deposits: 5.50,5.20
  Withdrawals: 3.20,4.10
  Balance: 3.40
Ed:
  Deposits: 3.00
  Withdrawals:
  Balance: 3.00
```