EECS 1012: LAB 07 – more on JavaScript; intro to unit-testing

A. IMPORTANT REMINDERS

1) Lab7 is due on **Wednesday (July 20)** at 11pm. No late submission will be accepted.
2) The pre-lab mini quiz is considered part of this lab. You are required to complete the pre-lab mini quiz 7 posted on eClass no later than the beginning of lab01's lab time, i.e., 1:30pm on Friday July 15.
3) You are welcome to attend the lab session on July 15, if you stuck on any of the steps below. TAs and instructor will be available to help you. The location is WSC105. Attendance is optional. You can come to either the morning or the afternoon lab sessions. (Later for lab tests, you need to come to your official lab session.)
4) Feel free to signal a TA for help if you stuck on any of the steps below. Yet, note that TAs would need to help other students too.
5) You can submit your lab work any time before the specified deadline.

B. IMPORTANT PRE-LAB WORKS YOU NEED TO DO BEFORE GOING TO THE LAB

1) Download this lab files and read them carefully to the end.
2) You should have a good understanding of
   - JavaScript objects, such as **Math**, **Date**, and DOM **document**
   - **array** and **string** in JavaScript
3) Also, see assert API here: https://www.chaijs.com/api/assert/

C. GOALS/OUTCOMES FOR LAB

1) To practice more concepts in programming, including variables, arrays, functions (aka sub-algorithms), and program control statements.
2) To use more JS objects, such as **document**, **Math**, and **Date**.
3) To get some idea of unit-testing.

D. TASKS

Part 0: For unit-testing in JavaScript, you need to install **Mocha** and **Chai** on your own computer. For that,

1) Install **node js**
2) Navigate to your lab7 folder and issue command: **npm init** (initialize app and create a package.json file)
3) Install **Mocha** and **Chai**: issue commandt **npm install mocha chai --save-dev**

See the end of the document for detailed installation instructions on different systems.

Part 1: download and unzip **lab07.zip**. the package contains subfolder **images** and **test**

1) TASK 0: revisit the light (five bulb) problem. Practice call function with parameters
2) TASK 1: Simple "HEADS" or "TAILS" button output with an if-statement. You also run some already prepared unit testing code for this task.
3) TASK 2: Passing variables to functions. You also run some already prepared unit testing code for this task.
4) TASK 3: Passing variables and for-loop. You also run some already prepared unit testing code for this task.
5) TASK 4: Random + string concatenation + if-statement. You also run some already prepared unit testing code for this task.
6) TASK 5: Date object + array + string concatenation.
7) TASK 6: Global variable and if-statement
8) TASK 7: More on global variables

E. SUBMISSIONS

eClass submission. More information can be found at the end of this document.

F. FURTHER DETAILS

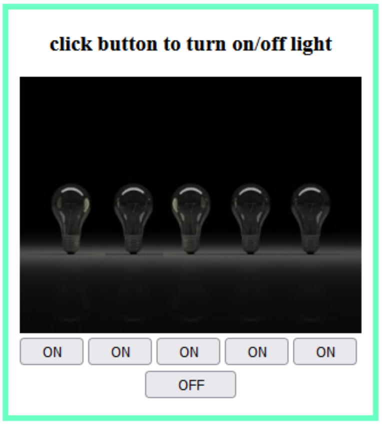
Task 0: revisit the “five bulbs” problem.

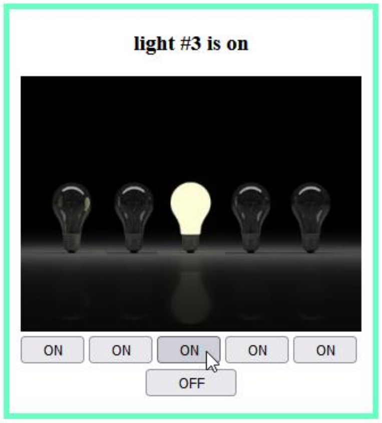
Recall the five-bulb problem we did in lab3. In the pdf it was mentioned that “since we just started learning JS, we use a simple approach: have a separate function per button. Later when we learn more JS, you will see that another approach would be to have a single function and pass a parameter. That is, click any button will call the same function, but with different input (as parameter to the function). Then the function changes the images according to the input“. Now it is the time to explore this cleaner approach.

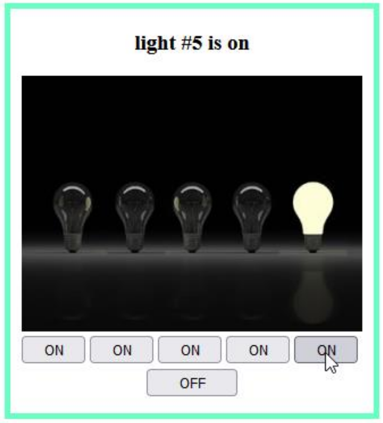
Complete the provided file **light.html**, which call the same JS function *turnOn()* with different parameters. For example, for first button you can call with parameter 1, for off button, you can call with 0 or 6.

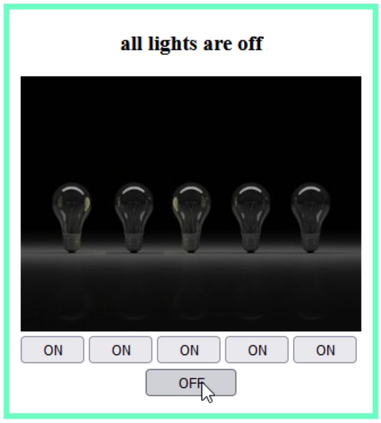
Then in the JS, implement the function, changing the image and text according to the parameter. Preferably you can use switch case, instead of if else to do the job.

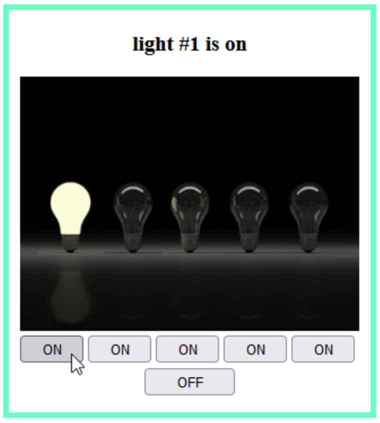
The result should be the same as in lab3, which is repeated here again.





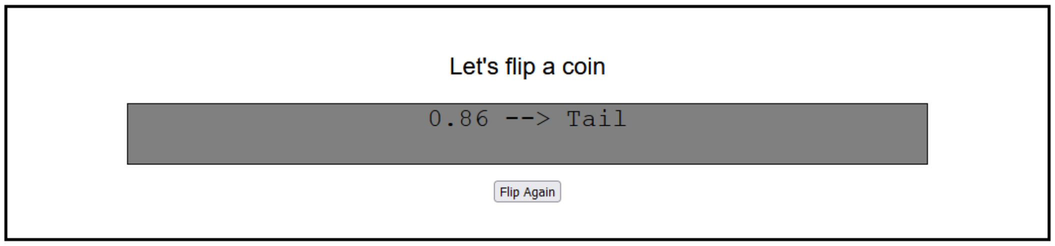




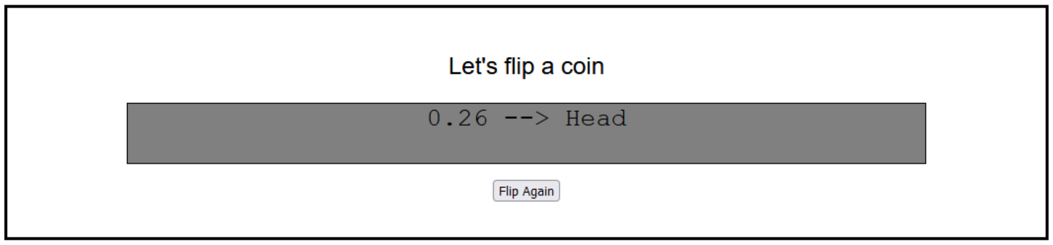


Task 1: Edit task1.js (you do not need to edit the HTML file).

For this task, we have already declared the JavaScript function myFunction() for you. Your function should do the following: Each time the button is clicked, your myFunction() code should call a sub-algorithm that generates a random number between 0 and 1 (including 0 and excluding 1). If the random number is less than 0.5, then have the innerHTML of the paragraph variable set to “num --> HEADS”, otherwise set it to “num --> TAILS”, where num is the random number generated, with 2 decimal digits. See example outputs below.

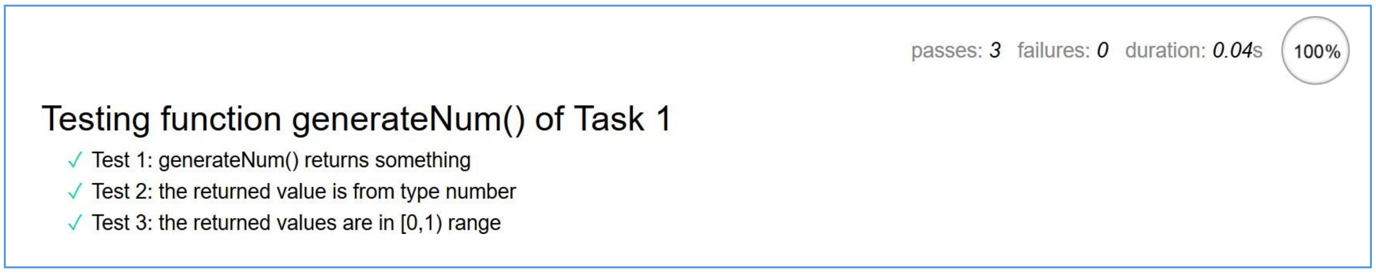

Let's flip a coin

```
0.86 --> Tail
```


Let's flip a coin

```
0.26 --> Head
```

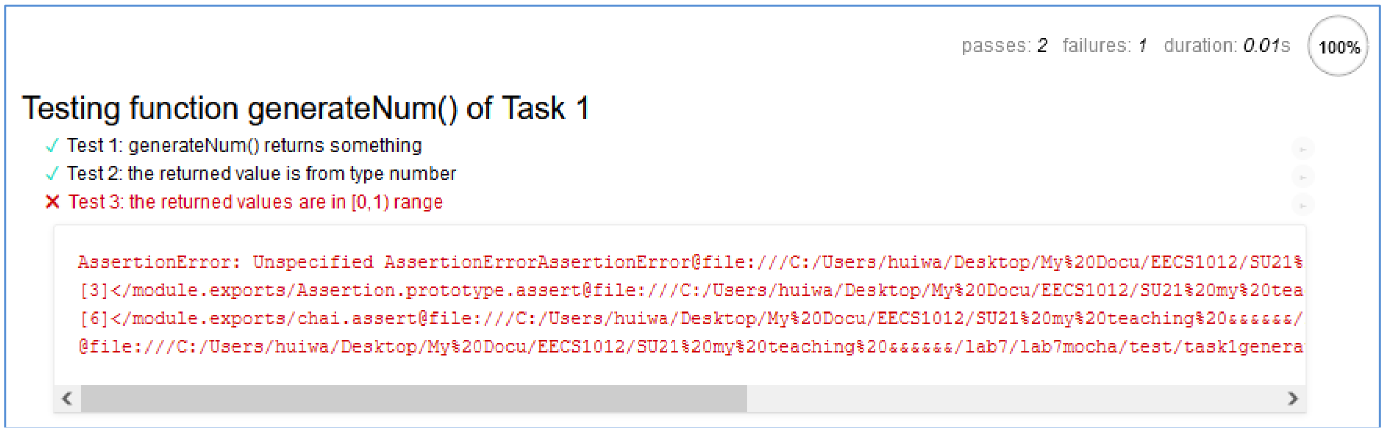
Once you are done, you want to test the program. How to do that? You can repeatedly click the button (say, for 3000 times) and check the output, but this is very tedious and time-consuming. This is how a test framework can help. Once you are done, we can test the function using unit test. After you have successfully installed **node js** and **mocha** and **chai** on your system, you can open **testRunner1.html** with Firefox. If you have implemented **task1.js** properly, you should see the following information as well as green check marks in your web browser, meaning that all 3 tests have passed. If any test fails, you must go back to your task1.js and debug your code. Note: to pass the tests, don't do the "2 decimal digits" formatting in the function generateNum(). Do it in main function.


passes: 3 failures: 0 duration: 0.04s 100%

Testing function generateNum() of Task 1

✓ Test 1: generateNum() returns something
✓ Test 2: the returned value is from type number
✓ Test 3: the returned values are in [0,1) range

If you did not see the error message -- congratulations :), then you can change your function to return a value that is not in the range of [0,1) (e.g., current return value +1), then refresh the **testRunner1.html** web page, and it will see something like the following.


passes: 2 failures: 1 duration: 0.01s 100%

Testing function generateNum() of Task 1

✓ Test 1: generateNum() returns something
✓ Test 2: the returned value is from type number
✗ Test 3: the returned values are in [0,1) range

```
AssertionError: Unspecified AssertionErrorAssertionError@file:///C:/Users/huiwa/Desktop/My%20Docu/EECS1012/SU21%
[3]</module.exports/Assertion.prototype.assert@file:///C:/Users/huiwa/Desktop/My%20Docu/EECS1012/SU21%20my%20tea
[6]</module.exports/chai.assert@file:///C:/Users/huiwa/Desktop/My%20Docu/EECS1012/SU21%20my%20teaching%20&&&&&&/
@file:///C:/Users/huiwa/Desktop/My%20Docu/EECS1012/SU21%20my%20teaching%20&&&&&&/lab7/lab7mocha/test/task1genera
```

Now change back the correct result in JS, and refresh the web page again, and you should see all the green check mark again. **testRunner1.html** links to file **task1generateNumTest.js** located in **test** folder. This file contains 3 test cases. Now open this file and try to understand the test cases. E.g., test case3 call the function 3000 times to generate 3000 random numbers and check if each one is in the range of [0, 1).

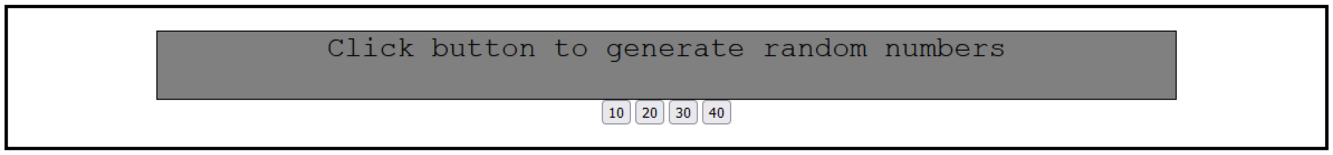
Task 2. Edit task2.html and task2.js

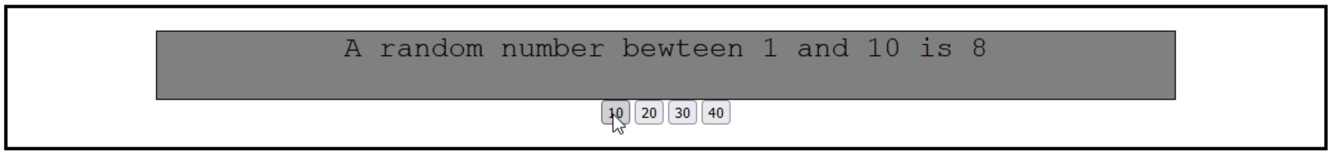
(1) Link your task2.js to your HTML code.

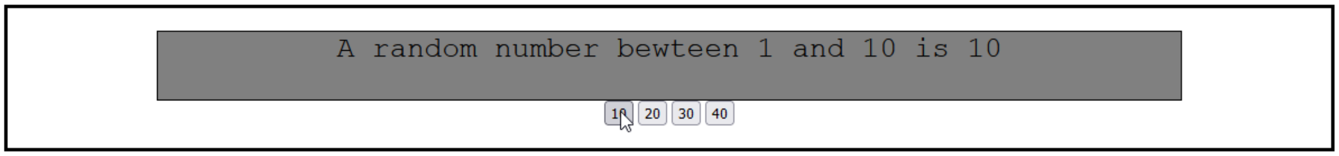
(2) Have the text in the paragraph "mydata" start with initial text as "click button to… " (see below).

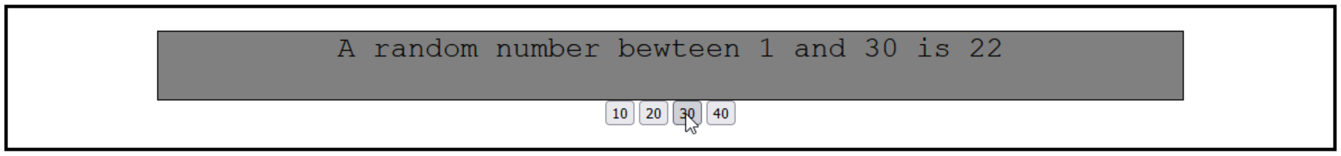
(3) Add four buttons to your **Task2.html** as shown below.

(4) Write a function in JavaScript that has one parameter. When a button is pressed, it should pass the value shown in the button (e.g., 10, 20, 30, or 40) to a function named **passNum**(). In your JavaScript code, your function should call a sub-algorithm that generates a random whole number between 1 and the passed value, inclusively. See example below for when the buttons that passe value 10 and 30 are pressed.

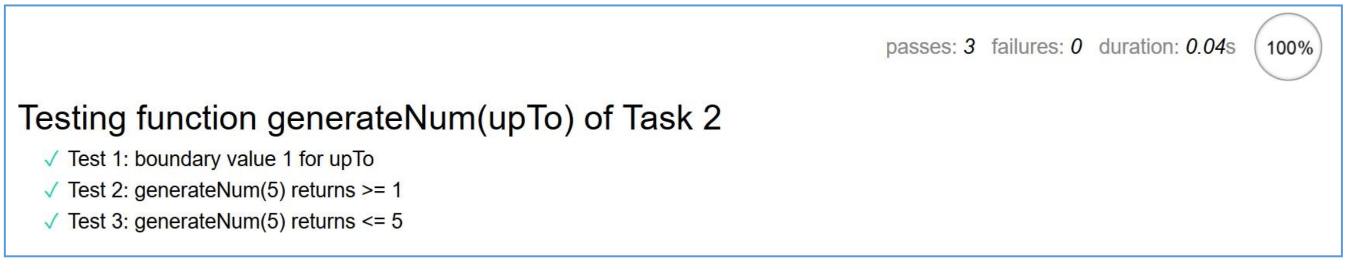






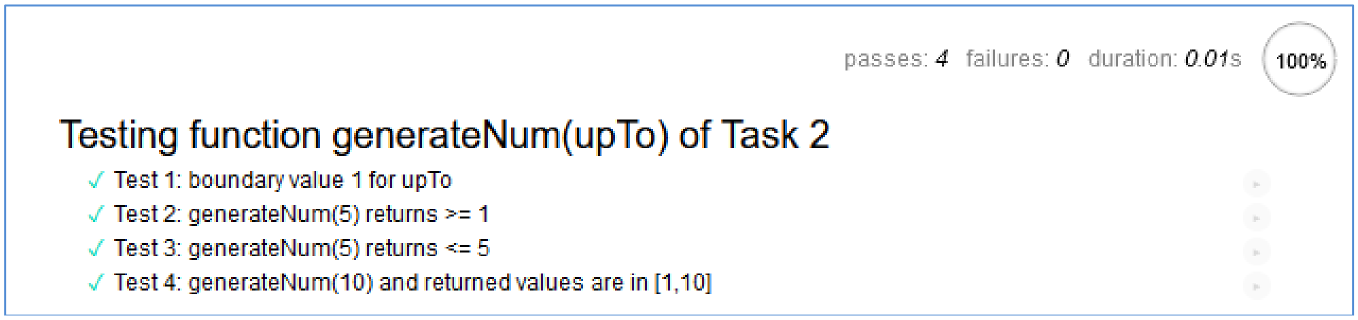


Once you are done, open **testRunner2.html** with Firefox. If you have implemented task2.js properly, you should see the following information in your web browser, meaning that all 3 tests have passed. If any test fails, you must go back to your task2.js and debug your code.



```
passes: 3   failures: 0   duration: 0.04s   100%

Testing function generateNum(upTo) of Task 2
  ✓ Test 1: boundary value 1 for upTo
  ✓ Test 2: generateNum(5) returns >= 1
  ✓ Test 3: generateNum(5) returns <= 5
```

testRunner2.html links to file **task2generateNumTest.js** located in **test** folder. This file contains 3 test cases. Now open this file and try to understand the test cases. E.g., test case3 call the function 1000 times with argument 5 to generate 1000 random numbers and check if each one is ≤ 5.
Now, complete the test 4 which calls the function 1000 times with argument 10 to generate 1000 random numbers and check if each one is ≥1 and also ≤ 10, i.e., in the range of [1, 10]. If implemented correctly, open **testRunner2.html** again in Firefox should get the following result:

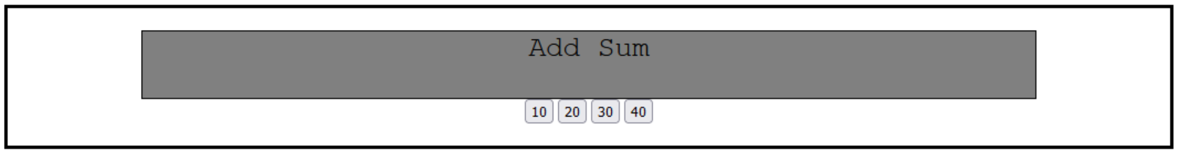


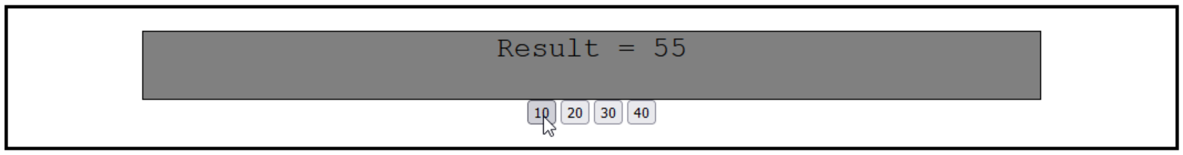
```
passes: 4   failures: 0   duration: 0.01s   100%

Testing function generateNum(upTo) of Task 2
  ✓ Test 1: boundary value 1 for upTo
  ✓ Test 2: generateNum(5) returns >= 1
  ✓ Test 3: generateNum(5) returns <= 5
  ✓ Test 4: generateNum(10) and returned values are in [1,10]
```

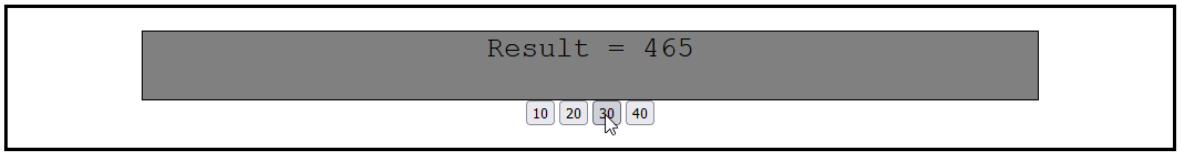
Task 3. Edit task3.html and **task3.js**

Write a function in JavaScript that has one parameter. When each button is pressed, it should pass the *value* shown in the button (e.g., 10, 20, 30, or 40). Use a for-loop to compute the sum of 0 to the passed *value*.

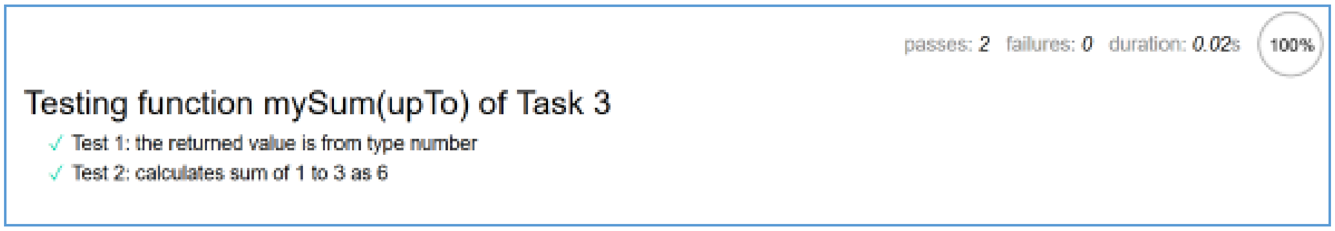
For example, if the value passed is 10, then compute 0+1+2+3+4+5+6+7+8+9+10=55. See below.





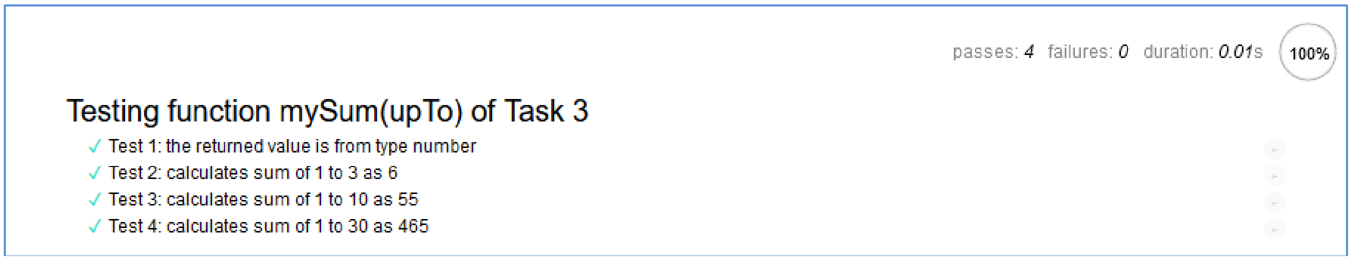


Once you are done, open **testRunner3.html** in Firefox. If you have implemented task3.js properly, you should see the following information in your web browser, meaning that both tests have passed. If any test fails, you must go back to your task3.js and debug your code.



```
passes: 2  failures: 0  duration: 0.02s  100%

Testing function mySum(upTo) of Task 3
  ✓ Test 1: the returned value is from type number
  ✓ Test 2: calculates sum of 1 to 3 as 6
```

The test cases for Task 3 are given in file **/test/task3mySumTest.js.** Study the code of the test cases in this file, and then add two more test cases, one for testing sum 1 to 10, and one for testing sum 1 to 30.
If implemented correctly, open **testRunner3.html** again in Firefox and you should get the following result:



```
passes: 4  failures: 0  duration: 0.01s  100%

Testing function mySum(upTo) of Task 3
  ✓ Test 1: the returned value is from type number
  ✓ Test 2: calculates sum of 1 to 3 as 6
  ✓ Test 3: calculates sum of 1 to 10 as 55
  ✓ Test 4: calculates sum of 1 to 30 as 465
```

Task 4. Modify task4.html and task4.js

(1) Link your JavaScrpit file to your HTML file.

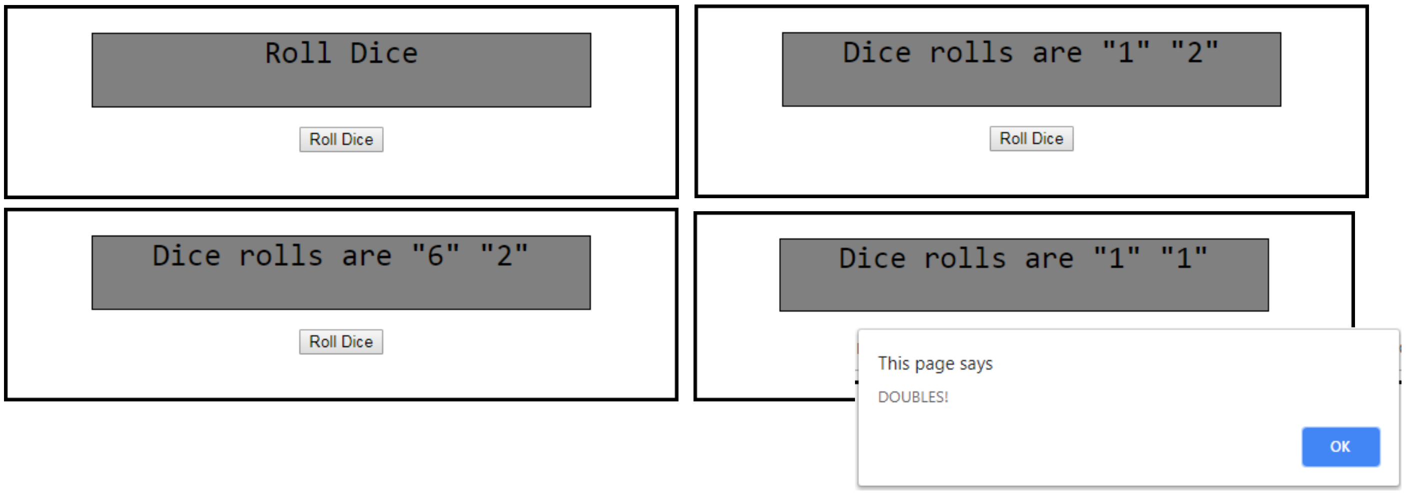
(2) Have the text in the paragraph "mydata" start with **Roll Dice**. Add a button "Roll Dice". Have this button respond the click event.

(3) Have the onlick for your button link to your JavaScript function. The function does not have parameters.

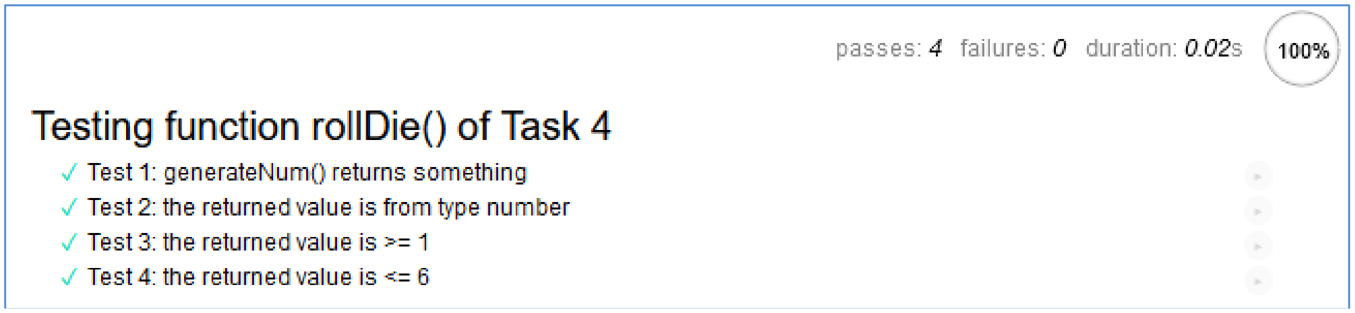
(4) Each time you click, have your JavaScript function generate two random numbers from 1 to 6, inclusive. These represent a dice. Change the innerHTML to say Dice rolls are "value1" and "value2", where value1 and value2 are the results of your random numbers. Note that the values are quoted in the display.

(5) If the two numbers are the equal, pop up an alert that says "DOUBLES!".

See examples below.

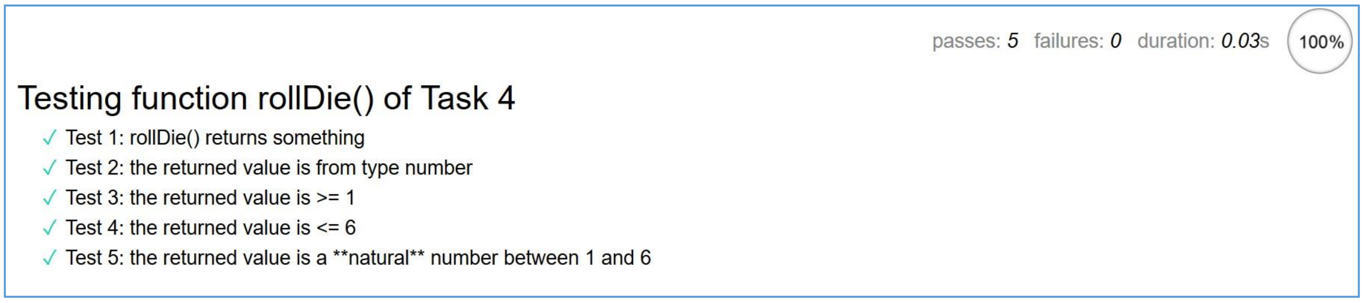


Once you have completed the steps above, open **testRunner4.html** in Firefox. You should see the following figure. Otherwise, you must go to your task4.js and debug your code.



```
passes: 4   failures: 0   duration: 0.02s   100%

Testing function rollDie() of Task 4
✓ Test 1: generateNum() returns something
✓ Test 2: the returned value is from type number
✓ Test 3: the returned value is >= 1
✓ Test 4: the returned value is <= 6
```

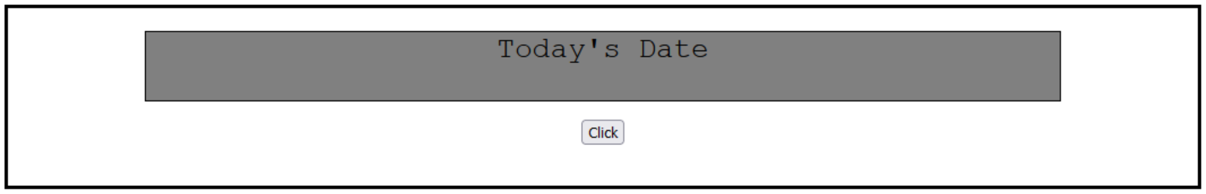
The test cases for Task 4 are given in file **/test/task4rollDieTest.js.** Study the code of the test cases in this file. The tests cases check the min and max value of generated numbers but the numbers can be real numbers, e.g., 2.75. Now complete Test5, which checks if the generated numbers are nature number 1,2,3,4,5 or 6.
If implemented correctly, open **testRunner4.html** again in Firefox and you should get the following result:

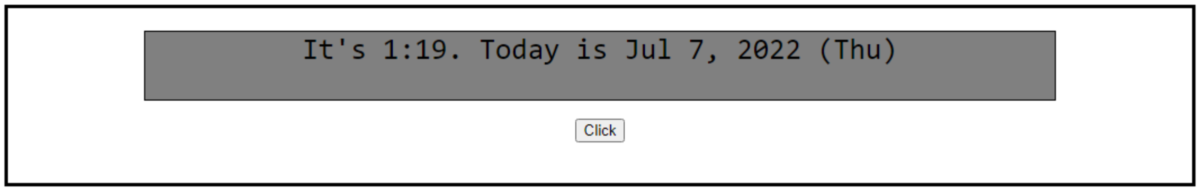


```
passes: 5   failures: 0   duration: 0.03s   100%

Testing function rollDie() of Task 4
✓ Test 1: rollDie() returns something
✓ Test 2: the returned value is from type number
✓ Test 3: the returned value is >= 1
✓ Test 4: the returned value is <= 6
✓ Test 5: the returned value is a **natural** number between 1 and 6
```

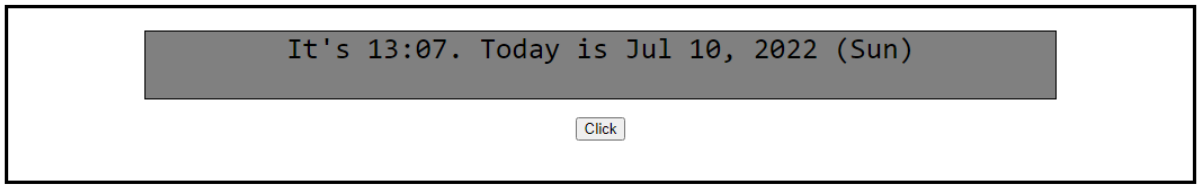
Task 5. Modify task5.js (you do not need to edit the HTML file).

In task5.js, complete the sub-program such that when task5.html runs, the current time and date are shown with a format similar to the figures below.





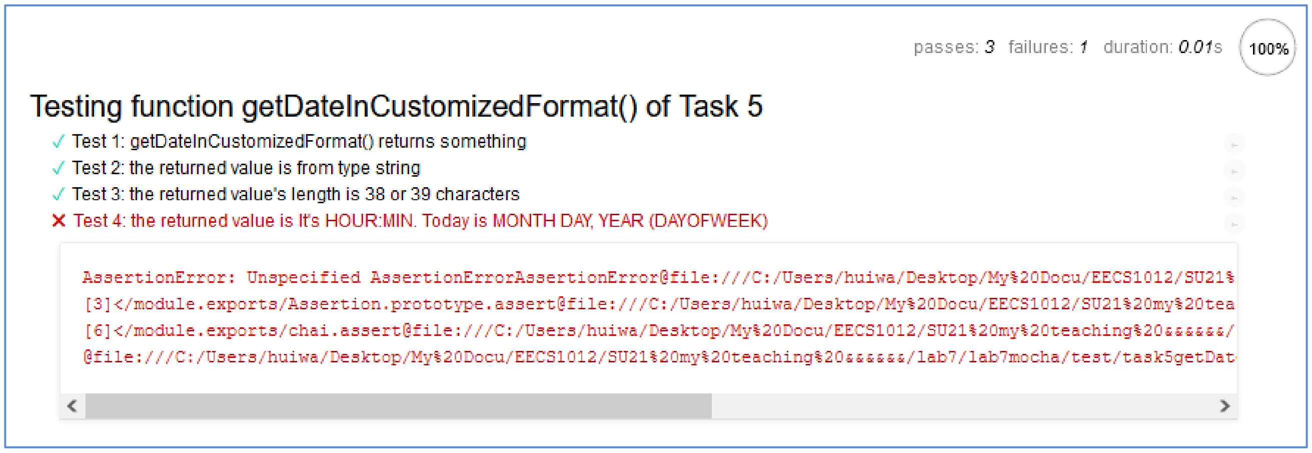
```
It's 1:19. Today is Jul 7, 2022 (Thu)
```



```
It's 13:07. Today is Jul 10, 2022 (Sun)
```

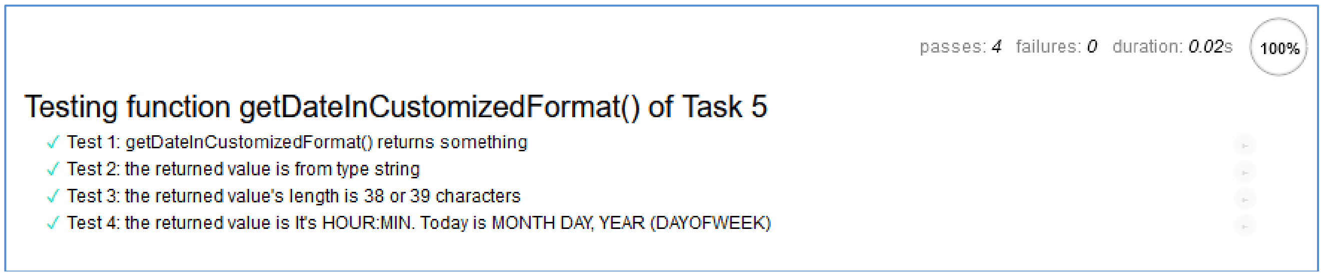
As discussed in class, we can create a Date object to retrieve the current time and date info. But the info retrieved from the object may not be in the desired form, e.g., getMonth () method returns 0 if the current month is January, and returns 1 if current month is February etc. To generate the word “Jan”, “Feb”, one way is to use if else or switch case. E.g., suppose the retrieved month data is stored in a variable *mon*, then we can use *if (mon == 6) output “Jul”*.. Another solution, which you will try in this exercise, is to put the expected word in an array, say named arr, with “Jan” having index 0 and “Feb” having index 1 etc. Then the expected word will be retrieved via *arr[mon].*
Note that as the example shows, for 7 minute pass 13:00, it should display 13:07, not 13:7.

Once you have completed the step above, open **testRunner5.html** in Firefox. You should see the following figure. If any of the first 3 tests have failed, you must go to your **task5.js** and debug your code.



```
passes: 3   failures: 1   duration: 0.01s   100%

Testing function getDateInCustomizedFormat() of Task 5
  ✓ Test 1: getDateInCustomizedFormat() returns something
  ✓ Test 2: the returned value is from type string
  ✓ Test 3: the returned value's length is 38 or 39 characters
  ✗ Test 4: the returned value is It's HOUR:MIN. Today is MONTH DAY, YEAR (DAYOFWEEK)

    AssertionError: Unspecified AssertionErrorAssertionError@file:///C:/Users/huiwa/Desktop/My%20Docu/EECS1012/SU21%
    [3]</module.exports/Assertion.prototype.assert@file:///C:/Users/huiwa/Desktop/My%20Docu/EECS1012/SU21%20my%20tea
    [6]</module.exports/chai.assert@file:///C:/Users/huiwa/Desktop/My%20Docu/EECS1012/SU21%20my%20teaching%20&&&&&&/
    @file:///C:/Users/huiwa/Desktop/My%20Docu/EECS1012/SU21%20my%20teaching%20&&&&&&/lab7/lab7mocha/test/task5getDat
```

The reason test 4 failed is that the test case cannot test the current time. To pass this test, open **test/task5getDateInCustomizedFormatTest.js** and go to the lines related to Test 4, replace the message with the exact current time and date of your computer, and then run the test immediately (within the same minute). You should see the following result then.



```
passes: 4   failures: 0   duration: 0.02s   100%

Testing function getDateInCustomizedFormat() of Task 5
  ✓ Test 1: getDateInCustomizedFormat() returns something
  ✓ Test 2: the returned value is from type string
  ✓ Test 3: the returned value's length is 38 or 39 characters
  ✓ Test 4: the returned value is It's HOUR:MIN. Today is MONTH DAY, YEAR (DAYOFWEEK)
```

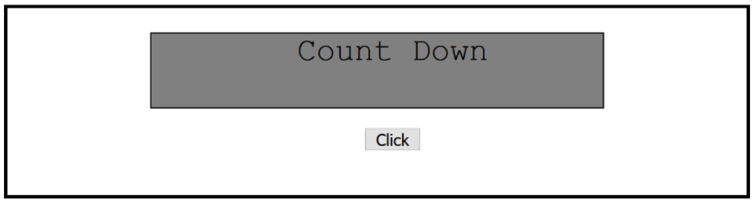
Task 6. Modify task6.js (you do not need to edit the HTML file).

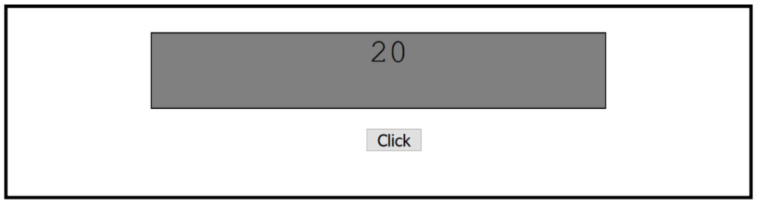
In task6.js, declare a **global variable**. A global variable is a variable that is created outside your function. Inside your function, you do not need to declare it again. If you modify the variable, the modification will be remembered next time you access the function. See example code here.

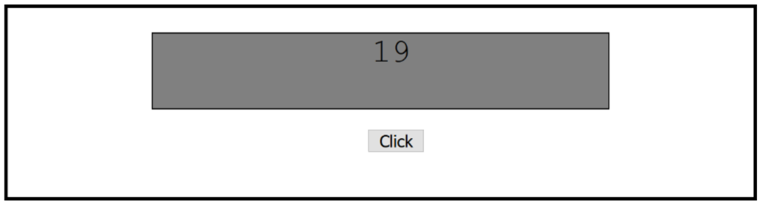
```
var i = 21;

function myFunction()
{
  i = i - 1;   // the value of i will be remembered next function call
}
```

Each time your button is clicked, you should reduce the global variable by 1 and show the result. Your innerHTML of the paragraph with id "mydata" should show the current value of the global variable. When the variable gets to 0, the next clicks have the innerHTML change to BOOM!

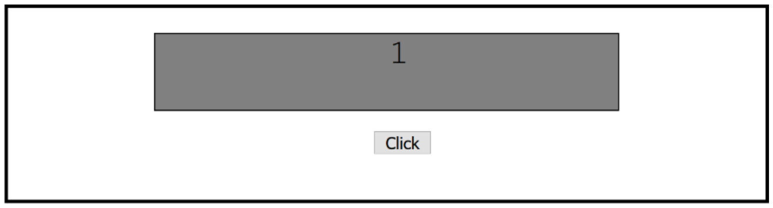
Your code should produce the exact results as the following figures. (i.e. it should not show 21, and it should show count down all the way to 0, and then show BOOM! for the following clicks on the button.)

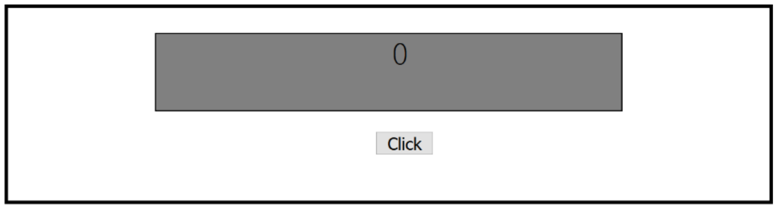


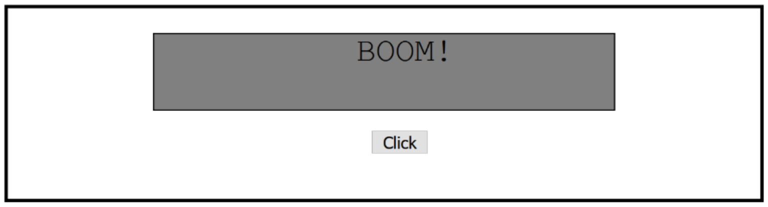




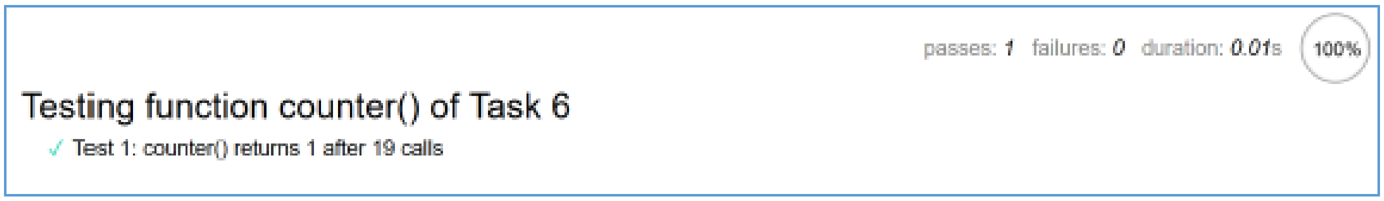
...





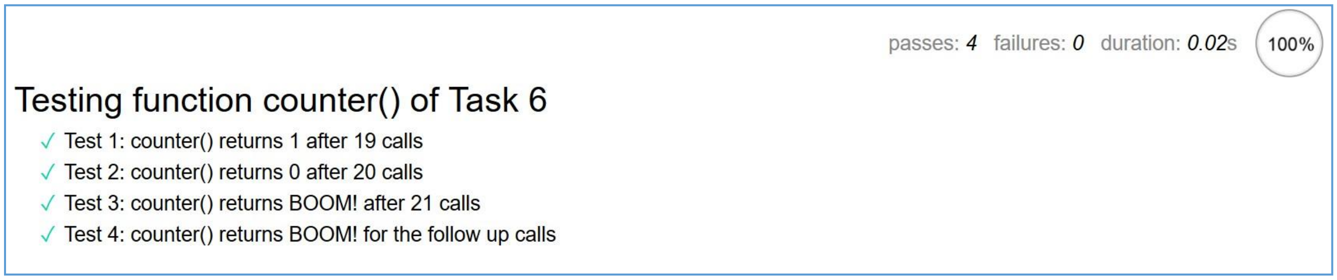


Once you have completed the step above, open **testRunner6.html** in Firefox. You should see the following figure. If one or more tests fail, you must go to your **task6.js** and debug your code.



```
passes: 1  failures: 0  duration: 0.01s  100%

Testing function counter() of Task 6
  ✓ Test 1: counter() returns 1 after 19 calls
```

Now, add 3 more test cases as explained in the `/test/task6counterTest.js`. Once you complete it properly, open **testRunner6.html** again in Firefox and you should see the following figure.



```
passes: 4   failures: 0   duration: 0.02s   100%

Testing function counter() of Task 6
  ✓ Test 1: counter() returns 1 after 19 calls
  ✓ Test 2: counter() returns 0 after 20 calls
  ✓ Test 3: counter() returns BOOM! after 21 calls
  ✓ Test 4: counter() returns BOOM! for the follow up calls
```

Task 7. Extending Task 6. Copy task6.html and task6.js to task7.html and task7.js

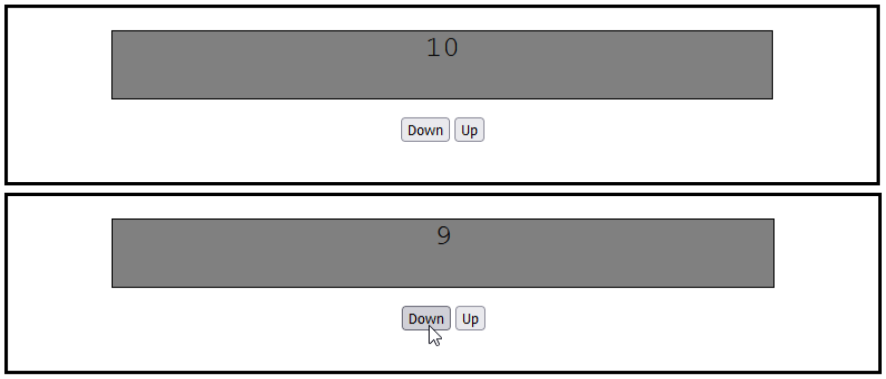
Modify **task7.html** so that the existing button's text is *Down*. Also add one more button with text *Up*. Add an onclick event to button *Up*, and call a JS function myFunctionUp(). This JS function will increment the global variable i by 1 on each call.

In this version, when the number decremented to 0, following calls to myFunction() produce "BOOM!" and the value of i will not be decremented further, i.e., i will stay 0. Next call to myFunctionUp() will increment it to 1.

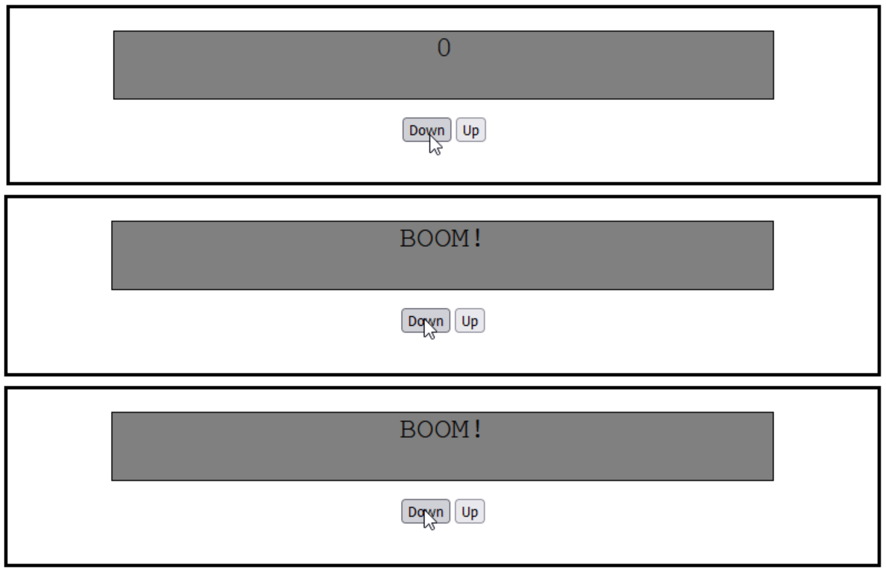
When the variable is incremented to 21, it should show "BOOM up!". After this, following calls to function myFunctionUIp() will not increment i value, i.e., i stays at 21. Next call to function myFunction() will decrement it to 20.

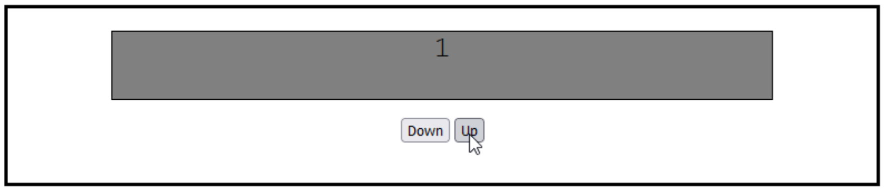
You can increment i directly in function myFuncitonUp(), or, in myFunctionUp() call another sub-program to do the incrementation, similar to how count down is implemented.

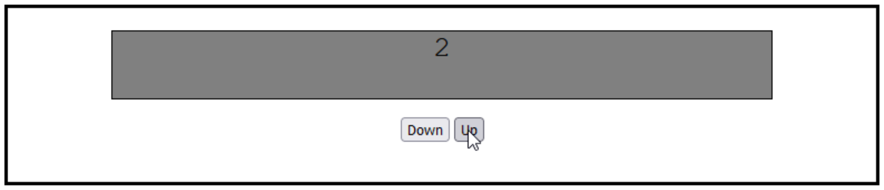
You can see that different functions can access and manipulate the same global variable, as shown below.



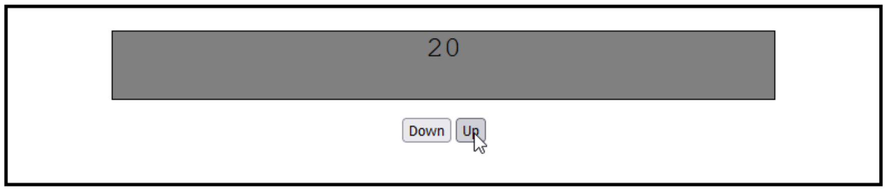
…

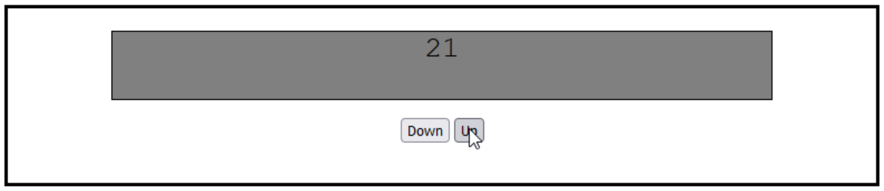


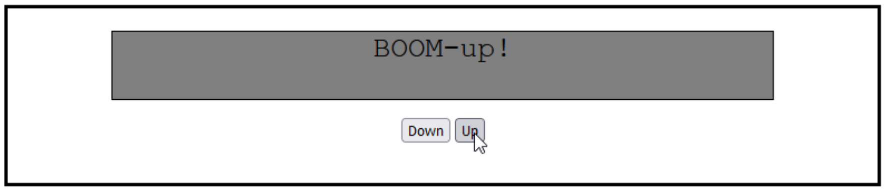


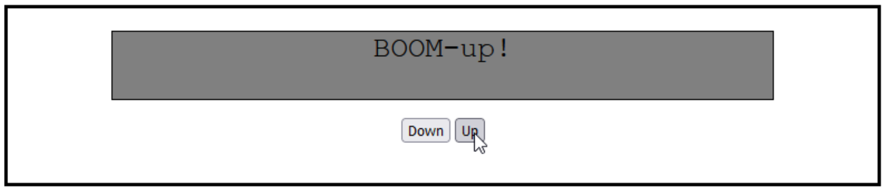


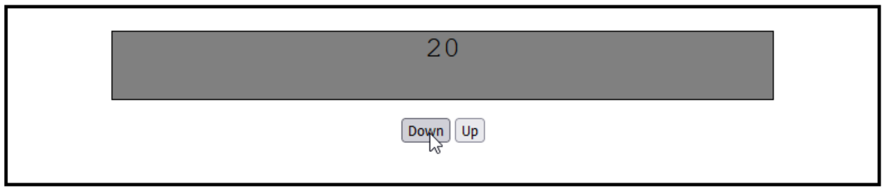
.....

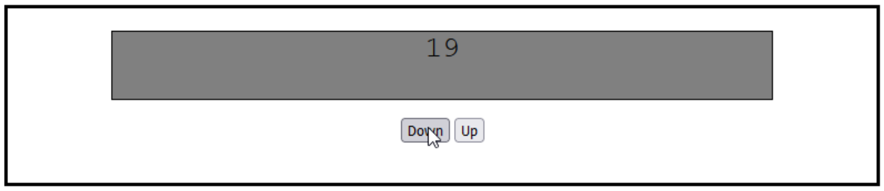


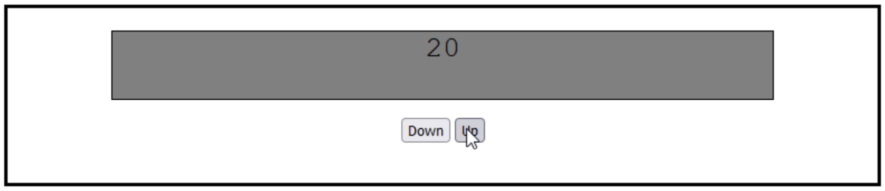












G. AFTER-LAB WORKS (THIS PART WILL NOT BE GRADED)

In order to review what you have learned in this lab as well as expanding your skills further, we recommend the following questions and extra practices:

1) Revisit objects **Math**, **Date**, and **document** in w3schools and explore more methods or properties of them by writing simple JavaScript codes.
    a. For object Math, learn more about **PI, round, pow, sqrt, abs, ceil, floor, sin, min, max**, etc. (https://www.w3schools.com/js/js_math.asp)
    b. For object **Date**, see 4 different constructors. (https://www.w3schools.com/js/js_dates.asp)
    c. For object **document**, see different methods such as **getElementsByTagName**, etc. (https://www.w3schools.com/js/js_htmldom_document.asp)
    d. Also see this fun animation made by html, css, JavaScript (https://www.w3schools.com/js/js_htmldom_animate.asp), and use your creativity to create some similar animations. As an example, make the red square go from the bottom-left corner to the top-right corner.
2) Revisit **arrays** and **strings** of JavaScript in w3schools. These two data structures have many applications in computer science and in your follow up courses. Interesting methods of strings include **indexOf(), lastIndexOf(), search(), slice(), substring(), substr(), replace(), concat(), trim(), charAT(), split() ,** etc. More on strings here: https://www.w3schools.com/js/js_strings.asp More on arrays here: https://www.w3schools.com/js/js_arrays.asp

3) Add more algorithms and programs to your **Learning Kit** Project. You would eventually need to have at least 30 buttons for 30 problems, algorithms, and JavaScript solutions in your Learning kit. You are highly encouraged to choose some problems that need some nested loops or sub-algorithms. Please feel free to discuss any of these questions in the course forum or see the TAs and/or Instructors for help.

H. SUBMISSION

eClass submission

You should already have a **Lab07** folder that contains folder **images**, **test, node_modules** and the following structure:

/node_modules

light.html, light.js, light.css

task{1,2,3,4,5,6,7}.html and **task{1,2,3,4,5,6,7}.js** **style.css**

testrunner{1,2,3,4,5,6}.html

/test/task{1,2,3,4,5,6}….Test.js

/images/light/light_{1,2,3,4,5}.jpg

Your HTML should pass the HTML validator at https://validator.w3.org

Compress the whole Lab07 folder (zip **or** tar **or** .gz**), and then submit the (single) compressed file on eClass.**

- In your zip folder, you need to include sub-folder **node_modules** so that the test cases can be run by the TAs.

Installation of Mocha and Chai unit test tools

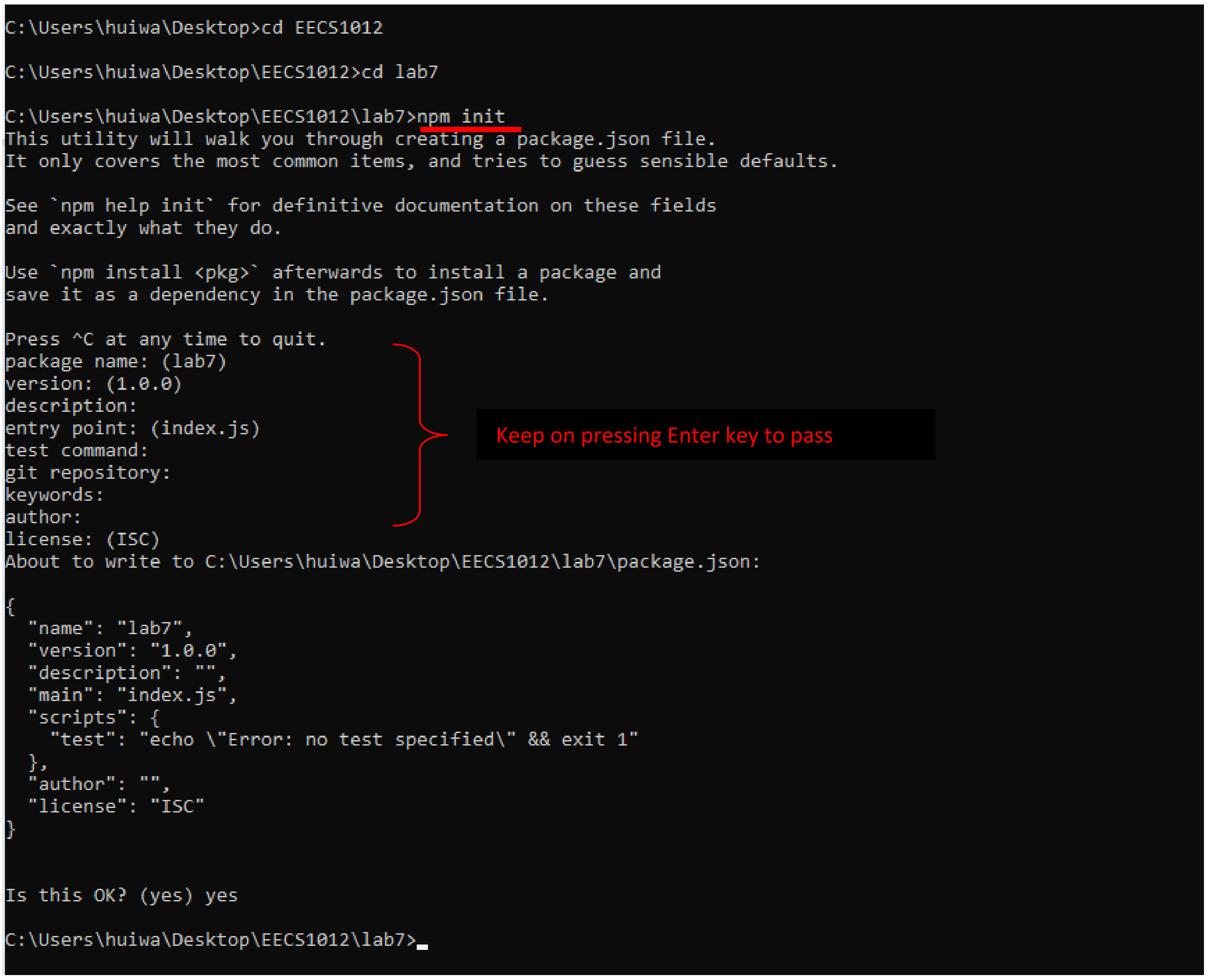
For windows users:

1. Go to https://nodejs.org/en/download/

2. Download the appropriate Windows Installer (.msi package) for your system, either 32-bit or 64-bit. If you don't know what version of Windows you are running, read this article for help: https://support.microsoft.com/en-us/windows/32-bit-and-64-bit-windows-frequently-asked-questions-c6ca9541-8dce-4d48-0415-94a3faa2e13d

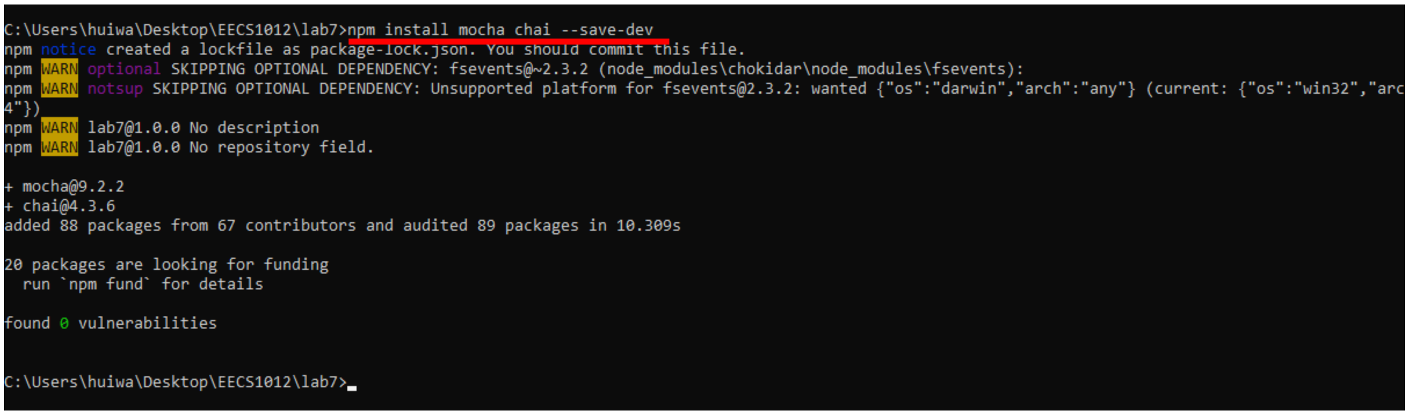
3. Open the .msi file, install Node.js (probably to the default directory provided by the installer, to avoid any problems later), follow the steps. You don't need to install extra packages.

4. Once the installer is finished, open the Command Prompt window and navigate to your lab7 folder (use *cd* command). Once in the lab7 folder, issue ***npm init*** to initialize environment. Keep on pressing Enter key to pass the steps. Eventually, type 'yes'. Now you should see your prompt symbol again.



```
C:\Users\huiwa\Desktop>cd EECS1012

C:\Users\huiwa\Desktop\EECS1012>cd lab7

C:\Users\huiwa\Desktop\EECS1012\lab7>npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help init` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (lab7)
version: (1.0.0)
description:
entry point: (index.js)
test command:
git repository:
keywords:
author:
license: (ISC)
About to write to C:\Users\huiwa\Desktop\EECS1012\lab7\package.json:

{
  "name": "lab7",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}


Is this OK? (yes) yes

C:\Users\huiwa\Desktop\EECS1012\lab7>
```

Next, install mocha chai by typing ***npm install mocha chai --save-dev***



```
C:\Users\huiwa\Desktop\EECS1012\lab7>npm install mocha chai --save-dev
npm notice created a lockfile as package-lock.json. You should commit this file.
npm WARN optional SKIPPING OPTIONAL DEPENDENCY: fsevents@~2.3.2 (node_modules\chokidar\node_modules\fsevents):
npm WARN notsup SKIPPING OPTIONAL DEPENDENCY: Unsupported platform for fsevents@2.3.2: wanted {"os":"darwin","arch":"any"} (current: {"os":"win32","arc
4"})
npm WARN lab7@1.0.0 No description
npm WARN lab7@1.0.0 No repository field.

+ mocha@9.2.2
+ chai@4.3.6
added 88 packages from 67 contributors and audited 89 packages in 10.309s

20 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities


C:\Users\huiwa\Desktop\EECS1012\lab7>_
```

(optional) Finally, open the generated package.json file, and replace test line with

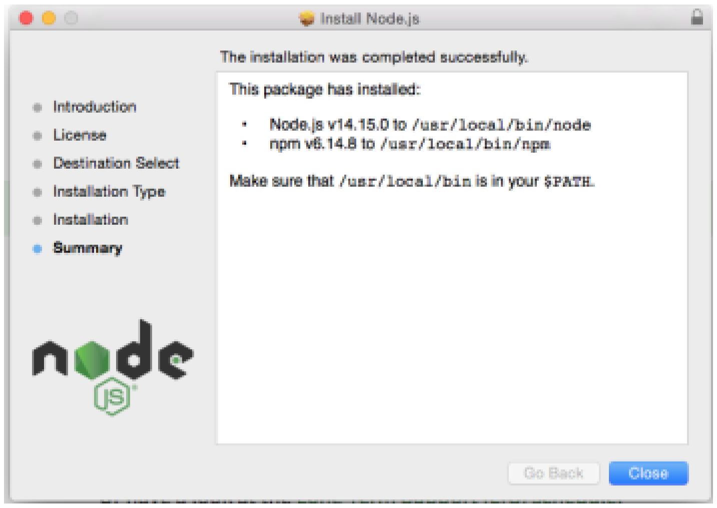
```
"scripts": {
"test": "mocha || true"

    }
```

Now, you should be able to run the testing html files provided.

For MAC users:

1. Obtain the macOS installer (.pkg) file from here: https://nodejs.org/en/download/

2. Open the .pkg file, and let Node.js install following the on-screen prompts. You will probably be asked to enter your system's password in order for the installer to proceed.



The installation was completed successfully.

This package has installed:

- Node.js v14.15.0 to `/usr/local/bin/node`
- npm v6.14.8 to `/usr/local/bin/npm`

Make sure that `/usr/local/bin` is in your `$PATH`.

3. Once Node.js is installed, open a new Terminal window (if you don't know what this is, open your Launchpad and type in "Terminal")

4. Now, navigate to your lab7 folder (use *cd* command), and then simply type in ***npm init*** and then keep on pressing Enter for the questions. Eventually, when you see your prompt symbol again, type ***npm install mocha chai --save-dev*** as shown in the window instructions.

Note: if you use the lab computer, the node.js is already installed there. So you just need to navigate to your lab7 directory, and issue ***npm init*** and then issue ***npm install mocha chai --save-dev***