

Lab 2

[New Attempt](#)

Due Feb 14 by 11:59pm **Points** 100 **Submitting** a file upload **File Types** zip
Available after Feb 7 at 12am

CS-546 Lab 2

The purpose of this lab is to familiarize yourself with Node.js modules and further your understanding of JavaScript syntax.

In addition, you must have error checking for the arguments of all your functions. If an argument fails error checking, you should throw a string describing which argument was wrong, and what went wrong. You can read more about error handling on the [MDN \(https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/throw\)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/throw).

You can download the starter template here: [lab2_stub-1.zip](https://sit.instructure.com/courses/64568/files/11122625?wrap=1)

<https://sit.instructure.com/courses/64568/files/11122625?wrap=1> ↓

https://sit.instructure.com/courses/64568/files/11122625/download?download_frd=1 PLEASE NOTE: THE STUB DOES NOT INCLUDE THE PACKAGE.JSON FILE. YOU WILL NEED TO CREATE IT! DO NOT FORGET TO ADD THE START COMMAND AND DO NOT FORGET TO ADD THE "type": "module" PROPERTY TO THE PACKAGE.JSON. THERE IS A HELPERS.JS FILE IN THE STUB, YOU CAN USE THIS IF YOU CREATE ANY HELPER FUNCTIONS THAT YOU CAN CALL FROM YOUR OTHER MODULES. YOU DO NOT HAVE TO USE THIS FILE FOR THE ASSIGNMENT IF YOU DO NOT WANT/NEED TO.

Initializing a Node.js Package

For all of the labs going forward, you will be creating Node.js packages, which have a `package.json`. To create a package, simply create a new folder and within that folder, run the command `npm init`. When it asks for a package name, name it **cs-546-lab-2**. You may leave the version as default and add a description if you wish. The entry file will be `app.js`.

All of the remaining fields are optional **except** author. For the author field, you **must** specify your first and last name, along with your CWID. **In addition**, You must also have a start script for your package, which will be invoked with `npm start`. You can set a start script within the `scripts` field of your `package.json`. Also add the `"type": "module"` property to the `package.json`

Here's an example of a valid package.json:

```
{
  "name": "cs-546-lab-2",
  "version": "1.0.0",
  "description": "My lab 2 module",
```

```
{
  "main": "app.js",
  "type": "module",
  "scripts": {
    "start": "node app.js"
  },
  "author": "John Smith 12345678",
  "license": "ISC"
}
```

arrayUtils.js

This file will export 3 functions, each of which will pertain to arrays.

sortAndFilter(array, [sortByField1, order], [sortByField2, order], filterBy, filterByTerm)

Given:

- An array of objects
- An array with a key to sort a field first on and the order of the sort (ascending/descending) ,
- An array with a key to sort a field second on and the order of the sort (ascending/descending),
- A key to filter by
- A value to be filtered, this function must return an array sorted by the sortByField1 in the order given first and then by the sortByField2 key in the order given and filtered by the filterBy key.

You must ensure that:

- The `array` parameter exists
- The `array` parameter is an array
- The `array` parameter is not empty
- each element in the `array` parameter is an `object` and there are at least two objects supplied in the `array` parameter.
- each object in the `array` parameter is not an empty object
- all objects in the array parameter have all the same keys.
- all values for for all keys in each object in the `array` parameter are strings (not just empty spaces)
- the 2nd array parameter `[sortByField1, order]` exists.
- the 2nd array parameter `[sortByField1, order]` is not empty.
- the 2nd array parameter `[sortByField1, order]` has two and only two elements.
- Each element in the 2nd array parameter `[sortByField1, order]` are strings (not just empty spaces)
- the 2nd array parameter `[sortByField1, order]` index 0 is a key that exists as a key in the array of objects and each object in the array of objects passed in as the first input parameter has that key.
- the 2nd array parameter `[sortByField1, order]` index 1 is either one of these values "asc" for ascending or "desc" for descending. This element can ONLY have a value of "asc" or "desc"
- the 3rd array parameter `[sortByField2, order]` exists.
- the 3rd array parameter `[sortByField2, order]` is not empty.
- the 3rd array parameter `[sortByField2, order]` has two and only two elements.

- Each element in the 3rd array parameter `[sortByField2, order]` are strings (not just empty spaces)
- the 3rd array parameter `[sortByField2, order]` index 0 is a key that exists as a key in the array of objects and each object in the array of objects passed in as the first input parameter has that key.
- the 3rd array parameter `[sortByField2, order]` index 1 is either one of these values "asc" for ascending or "desc" for descending. This element can ONLY have a value of "asc" or "desc"
- the `filterBy` key exists in the objects passed in the array of objects
- the `filterByTerm` exists (meaning there is at least one object that has that value and is a string (not just empty spaces))

Example:

```
let people = [
{name: 'Ryan', age: '22', location: 'Hoboken', role: 'Student'},
{name: 'Matt', age: '21', location: 'New York', role: 'Student'},
{name: 'Matt', age: '25', location: 'New Jersey', role: 'Student'},
{name: 'Greg', age: '22', location: 'New York', role: 'Student'},
{name: 'Mike', age: '21', location: 'Chicago', role: 'Teacher'} ];

console.log(sortAndFilter(people, ['name', 'asc'], ['location', 'asc'], 'role', 'Student'));
/* output:
[{name: 'Greg', age: '22', location: 'New York', role: 'Student'},
{name: 'Matt', age: '25', location: 'New Jersey', role: 'Student'},
{name: 'Matt', age: '21', location: 'New York', role: 'Student'},
{name: 'Ryan', age: '22', location: 'Hoboken', role: 'Student'}]
*/

console.log(sortAndFilter(people, ['name', 'asc'], ['location', 'desc'], 'role', 'Student'));
/* output:
[{name: 'Greg', age: '22', location: 'New York', role: 'Student'},
{name: 'Matt', age: '21', location: 'New York', role: 'Student'},
{name: 'Matt', age: '25', location: 'New Jersey', role: 'Student'},
{name: 'Ryan', age: '22', location: 'Hoboken', role: 'Student'}]
*/

console.log(sortAndFilter(people, ['location', 'asc'], ['name', 'asc'], 'age', '22'));
/* output:
[{name: 'Ryan', age: '22', location: 'Hoboken', role: 'Student'},
{name: 'Greg', age: '22', location: 'New York', role: 'Student'}]
*/

console.log(sortAndFilter(people, ['ssn', 'asc'], ['name', 'asc'], 'age', '22'));
/* output:
Error: the sortByField1 is not a key in each object of the array
*/

console.log(sortAndFilter(people, ['location', 'none'], ['name', 'asc'], 'age', '22'));
/* output:
Error: the order of sortByField1 must be either 'asc' or 'desc'
*/

console.log(sortAndFilter(people, ['location', 'asc'], ['name', 'asc'], 'phone', '22'));
/* output:
Error: the filterBy key is not a key in each object of the array
*/

console.log(sortAndFilter(['location', 'asc'], ['name', 'asc'], 'age', '22'));
/* output:
Error: the array does not exist
*/

console.log(sortAndFilter(['string', {}], ['location', 'asc'], ['name', 'asc'], 'age', '22'));
/* output:
Error: each element in the array must be an object
*/

console.log(sortAndFilter(people, ['location', 'asc'], ['name', 'asc'], 'age', 22));
```

```

/* output:
Error: the filterByTerm must be a string
*/

console.log(sortAndFilter([ {name: 'Ryan', age: '22', location: 'Hoboken', role: 'Student'}, {name: 'Greg', age: 22, location: 'New York', role: 'Student'}], 'location', 'age', '22'));
/* output:
Error: each value for each key in each object in the array must be a string
*/

```

merge(...args)

For this function, **you will have to take into account a variable number of input parameters**. You will take in `arrays` as input. You will merge the arrays into one array. You will sort that array numerically first, and then alphabetically (if there are strings in the array). If you

called `merge([3,0,1,2,4], [1,2,8,15], [6,3,10,25,29])` You would return:

`[0,1,1,2,2,3,3,4,6,8,10,15,25,29]` If you called `merge([3,0,"Lab2",2,"Aiden"], ["CS-546", "Computer Science",8,15], [6,3,"Patrick",25,29])` You would return: `[0,2,3,3,6,8,15,25,29,"Aiden","CS-546","Computer Science", "Lab2", "Patrick"]`

For the elements that are strings, you will use the ASCII sort order to sort them For example: If you called `merge([3,0,"Lab2",2,"Aiden"], ["CS-546", "Computer Science",8,15], [6,3,"!Patrick",25,29])` You would return: `[0,2,3,3,6,8,15,25,29,"!Patrick","Aiden","CS-546","Computer Science", "Lab2"]`

You must also account for nested array cases For example: If you called

`merge(["bar", 0, 1, [[5, "foo"]]], [7, "buzz", ["fizz", 8]])`

You would return:

`[0, 1, 5, 7, 8, "bar", "buzz", "fizz", "foo"]`

You must check:

- At least one array is supplied as input
- That each input is an array
- Each array is of the proper type (meaning, it's an array)
- Each array is not empty and has at least one element
- Each array element is either a string, number or an array that has either strings or numbers as elements. You will need to flatten the array first (strings with just spaces are allowed as a space can be sorted using the ASCII sort order method)

If any of those conditions fail, you will throw an error.

matrixMultiply(...args)


For this function, **you will take a variable amount of array inputs**. Each array would represent a matrix, a set of numbers arranged within rows and columns. So, for example, a 2x3 matrix would be `[[1, 2, 3], [4, 5, 6]]` and a 5x2 matrix would be `[[1, 2], [3, 4], [5, 6], [7, 8], [9, 10]]`.

Perform one or more matrix multiplications given the number of matrices (or array of arrays) given as input. Return the resulting matrix from these multiplications. If matrix multiplication is not possible, throw an error.

You must ensure that:

- There are at least two inputs
- Each input is an array
- Each array is not empty
- The outer array must have only arrays as elements
- The inner arrays must only have numbers as elements
- Each inner array is of the same length

In order for any two matrices to be multiplied together, **the number of columns within the first matrix must match the number of rows within the second matrix**. So, for a 2x3 matrix and a 3x2 matrix, we can multiply these since the first matrix has 3 columns and the second matrix has 3 rows. For a 2x4 matrix and a 1x4 matrix, we cannot multiply these together since the first matrix has 4 columns while the second matrix has 1 row.

For those that aren't familiar with performing matrix multiplication, here is a link summarizing how it works: <https://www.mathsisfun.com/algebra/matrix-multiplying.html> 
(<https://www.mathsisfun.com/algebra/matrix-multiplying.html>)

Examples:

```
matrixMultiply([ [2,3], [3,4], [4,5] ], [ [1,1,1], [2,2,2] ], [ [3], [2], [1] ]) would return [ [48], [66], [84] ]  
  
matrixMultiply([ [3,5] ], [ [4], [4] ]) would return [ [32] ]  
  
matrixMultiply([]) //throws an error  
matrixMultiply([ [1,2], [3,3] ]) //throws an error  
matrixMultiply([ [1,2] ], [ ['foobar'], [6] ]) //throws an error
```

stringUtils.js

This file will export 3 functions, each are useful functions when dealing with strings in JavaScript.

palindromes(strings)

Given an array of strings, you will return an object that contains each string element as a key (after it's been converted to lowercase and stripped of any non alphanumeric characters) and a boolean for the value of that key which will state if that key is a palindrome or not.

A palindrome is a phrase that is spelled the same way, backwards and forwards (ignoring spacing and punctuation). For example, the following phrases are palindromes:

- Madam
- Was it a cat I saw?
- He did, eh?

- Go hang a salami, I'm a lasagna hog.
- Poor Dan is in a droop

For each string element in the array, you will:

1. Lowercase the text
2. Strip all non alphanumeric text; this includes spaces.

For example, `Hello, 2 the world!` becomes `hello, 2 the world!` when lowercased and then `hello2theworld` when stripped of all non alphanumeric text

3. Determine whether or not the text is a palindrome

You must check:

- That the array exists
- The array is of the proper type (meaning, it's an array)
- The array is not empty
- Each array element in the array is a string (No strings with empty spaces)
- Each array element in the array consists of at least one alphanumeric character (No strings consisting of only non-alphanumeric characters)
- That each `string` element exists.

If any of those conditions fails, the function will throw.

```
palindromes(["Madam", "Loot", "Was it a cat I saw?", "Poor Dan is in a droop", "Anna", "Nope" ]);  
// Returns: {madam: true, loot: false, wasitacatisaw: true, poordanisinadroop: true, anna: true, nope: false}  
palindromes(); // throws error  
palindromes("hi"); // throws error  
palindromes(" "); // throws error  
palindromes(1); //throws error
```

sensorWords(str, badWordsList)

Given an input string and an array of strings to be censored, return the input string while replacing each word that is present in the input string and in the bad words list with special characters (each bad word should maintain its original length). How it works is like so: you will use `!`, `@`, `$`, `#` in that particular order and replace each character in each bad word with these starting with `!`. For example if the bad word is "pineapple", the censored word would be "`!@$#!@$#!`". For every bad word encountered, the pattern resumes from where it last left off, so after encountering "pineapple" and we later encounter "pretzel" as the next bad word in the input string, we will censor it as "`@$#!@$#!`". Also, you must censor any strings in the bad words list that appear as substrings in the input string - an example is provided below.

You must ensure the following:

- the input string exists and is a string (not just empty spaces)
- The bad words list exists and is an array

- The bad words list is not empty
- Each element in the bad words list is a string
- Each element in the bad words list must exist in the input string

Examples:

```
let badWords = ["bread","chocolate","pop"];

console.log(censorWords("I like bread that has chocolate chips in it but I do not like lollipops", badWords))
/*
output: "I like !@$#! that has @$#!@$#!@ chips in it but I do not like lolli$#!s"
*/

console.log(censorWords("", badWords))
/*
output: Error: input string cannot be an empty string
*/

console.log(censorWords("I like bread that has chocolate chips in it", [2, "wow"]))
/*
output: Error: each element in the bad words list must be a string
*/
```

distance(string, word1, word2)

Given `string`, `word1`, and `word2` return the minimum distance between `word1` and `word2` in the `string`, where `word1` appears before `word2`. You should be calculating the distance based on index, meaning `word2` is inclusive in the distance. If `word1` is at index 1 and `word2` is at index 8, then the distance is $8-1=7$.

You must check:

- That `string`, `word1`, and `word2` exist
- That `string`, `word1`, and `word2` are of type string
- That `string`, `word1`, and `word2` are not just empty strings
- That `string`, `word1`, and `word2` are not just strings made of punctuation symbols
- That `string` is at least two words long
- That `word1` and `word2` are not the same
- That `word1` and `word2` exist in the `string`
- That `word1` appears before `word2` in the `string`

If any of those conditions fail, the function will throw.

This function is **case insensitive**.

```
distance() // throws error
distance([],true) // throws error
distance("", "", "") // throws error
distance("Hello World!", " !?!", " ... ") // throws error
distance("Patrick", "Patrick", "Patrick") // throws error
distance(123, "CS", "Patrick") // throws error
distance("Hello there", "hello", "") // throws error
distance("Give me music suggestions", "rock", "pop") // throws error
distance("Bob met Adam on wednesday", "Adam", "Bob") // throws error
distance("I was going to buy preworkout powder yesterday", "going to", "workout powder") // throws error
```



```
distance("The brown fox jumped over the lazy dog", "fox", "dog") // returns 5
distance("I was going to buy workout powder yesterday", "going to", "workout powder") // returns 2
distance("sphinx of black quartz, judge my vow", "QUARTZ", "vOW"); // returns 3
distance("I really hope it will snow soon because I like snow", "I", "snow") // returns 2
distance("I like sweet and salty but I like sweet more", "salty", "sweet") // returns 4
```

objUtils.js

This file will export 3 functions that are useful when dealing with objects in JavaScript.

areObjectsEqual(...args)

This method takes in **a variable number of objects** and checks each field (at every level deep) for equality. It will return **true** if each field in each of the supplied objects is equal, and **false** if not.

Note: Empty objects can be passed into this function.

For example, if given the following:

```
const first = {a: 2, b: 3};
const second = {a: 2, b: 4};
const third = {a: 2, b: 3};
const forth = {a: {sA: "Hello", sB: "There", sC: "Class"}, b: 7, c: true, d: "Test"}
const fifth = {c: true, b: 7, d: "Test", a: {sB: "There", sC: "Class", sA: "Hello"}}
const sixth = {name: {firstName: "Patrick", lastName: "Hill"}, age: 47, dob: '9/25/1975', hobbies: ["Play ing music", "Movies", "Spending time with family"]}
const seventh = {age: 47, name: {firstName: "Patrick", lastName: "Hill"}, hobbies: ["Playing music", "Mov ies", "Spending time with family"], dob: '9/25/1975'}
const eighth = {b:3, a:2}
console.log(areObjectsEqual(first, second, third)); // false
console.log(areObjectsEqual(forth, fifth)); // true
console.log(areObjectsEqual(forth, third, sixth)); // false
console.log(areObjectsEqual(sixth, seventh)); // true
console.log(areObjectsEqual(first, eighth, third)); // true
console.log(areObjectsEqual({}, {}, {}, {}, {})); // true
console.log(areObjectsEqual([1,2,3], [1,2,3])); // throws error
console.log(areObjectsEqual("foo", "bar")); // throws error
```

You must check:

- That input exists and is of proper type (an Object). If not, throw an error.
- There are at least two objects passed into the function, if not, throw an error

Hint: Using recursion is the best way to solve this one.

Remember: The order of the keys is not important so: **{a: 2, b: 4}** is equal to **{b: 4, a: 2}**

calculateObject(object, funcs)

Given an object and an array of functions, evaluate the functions in order on the values of the object, using the output of the previous function as the input of the next function. Return a new object with the results. Note, on the result, please use the `toFixed(2)` function to only display 2 decimal places rounded.

You must check:

- That the **object** exists and is of proper type (an object). If not, throw an error.

- That `funcs` exists and is of proper type (an array). If not, throw an error.
- That the `object` values are all numbers (positive, negative, decimal). If not, throw an error.
- That the `funcs` array has at least one element and that the elements are of proper type (functions). If not, throw an error.

You can assume that the correct types will be passed into the `funcs` parameter element functions since you are checking the types of the values of the object beforehand.

```
calculateObject({ a: 3, b: 7, c: 5 }, [(n => n * 2), (n => Math.sqrt(n))]);
/* Returns:
{
  a: 2.45,
  b: 3.74,
  c: 3.16
}
*/

calculateObject({ a: 'Hello', b: 7, c: false }, [(n => n * n)]);
/* Throws an error */

calculateObject({ a: 1, b: 2, c: 3}, [false]);
/* Throws an error */
```

combineObjects(...args)

Given a variable amount of objects, you will find all the keys that appear in at least any two objects and return a new object with all these keys where the value of each of those keys is the value from the first object that had the key. In other words, if a key `'k'` appears in objects `args[0]` and `args[2]`, the result object will include the key `k` and its value will be that of `args[0]['k']`.

You must check:

- That `args` has at least two objects. If not, throw an error
- That each object in `args` is of proper type (an object), has **at least 1 key**. If not, throw an error.

Examples:

```
combineObjects(
  { a: 3, b: 7, c: 5 },
  { d: 4, e: 9 },
  { a: 8, d: 2 }
);
/* Returns:
{
  a: 3,
  d: 4
}
*/

combineObjects(
  { b: 7, c: 5 },
  { d: 4, e: 9, a: 'waffle' },
  { a: 8, d: 2 },
  { d: 3, e: 'hello' }
);
/* Returns:
{
  a: 'waffle',
  d: 4,
```

```
e: 9
}

combineObjects(
  { apple: 'orange', orange: 'pear' },
  { pear: 'blueberry', fruit: 4 },
  { cool: false, intelligence: -2 }
);
/* Returns:
{ }
*/

combineObjects(
  { wow: 'crazy', super: 'duper' },
  false
);
/* Throws an error */
```

Testing

In your `app.js` file, you must import all the functions the modules you created above export and create one passing and one failing test case for each function in each module. So you will have a total of 18 function calls (there are 9 total functions)

For example: (these are just generic function call examples, you would use the functions that you created, specified above)

```
try {
  // Should Pass
  const meanOne = mean([2, 3, 4]);
  console.log('mean passed successfully');
} catch (e) {
  console.error('mean failed test case');
}
try {
  // Should Fail
  const meanTwo = mean(1234);
  console.error('mean did not error');
} catch (e) {
  console.log('mean failed successfully');
}
```

Requirements

1. Write each function in the specified file and export the function so that it may be used in other files.
2. Ensure to properly error check for different cases such as arguments existing and of the proper type as well as throw if anything is out of bounds such as invalid array index.
3. Import ALL exported module functions and write 2 test cases for each in `app.js`.
4. Submit all files (including `package.json`) in a zip with your name in the following format:
`LastName_FirstName.zip`.
5. do NOT have the files in any folders, they should be in the root of the zip file
6. **You are not allowed to use any npm dependencies for this lab.**

Lab 2 Rubric

Criteria	Ratings		Pts
arrayUtils.js - sortAndFilter Test cases used for grading will be different from assignment examples.	13.5 to >0.0 pts 1 Points/Error Handling Test Case & 1 Points/Valid Input Test Case 4 Error Handling Test Cases & 2 Valid Input Test Cases. The function is implemented correctly, and few or all test cases pass.	0 pts All Test Cases Failed Incorrect implementation or none of the test cases pass.	13.5 pts
arrayUtils.js - merge Test cases used for grading will be different from assignment examples.	13.5 to >0.0 pts 1.5 Points/Error Handling Test Case & 3.75 Points/Valid Input Test Case 4 Error Handling Test Cases & 2 Valid Input Test Cases. The function is implemented correctly, and few or all test cases pass.	0 pts All Test Cases Failed Incorrect implementation or none of the test cases pass.	13.5 pts
arrayUtils.js - matrixMultiply Test cases used for grading will be different from assignment examples.	6 to >0.0 pts 1.5 Points/Error Handling Test Case & 3.75 Points/Valid Input Test Case 4 Error Handling Test Cases & 2 Valid Input Test Cases. The function is implemented correctly, and few or all test cases pass.	0 pts All Test Cases Failed Incorrect implementation or none of the test cases pass.	6 pts
stringUtils.js - palindromes Test cases used for grading will be different from assignment examples.	13.5 to >0.0 pts 1.5 Points/Error Handling Test Case & 3.75 Points/Valid Input Test Case 4 Error Handling Test Cases & 2 Valid Input Test Cases. The function is implemented correctly, and few or all test cases pass.	0 pts All Test Cases Failed Incorrect implementation or none of the test cases pass.	13.5 pts
stringUtils.js - censorWords Test cases used for grading will be different from assignment examples.	13.5 to >0.0 pts 1.5 Points/Error Handling Test Case & 3.75 Points/Valid Input Test Case 4 Error Handling Test Cases & 2 Valid Input Test Cases. The function is implemented correctly, and few or all test cases pass.	0 pts All Test Cases Failed Incorrect implementation or none of the test cases pass.	13.5 pts
stringUtils.js - distance Test cases used for grading will be different from assignment examples.	6 to >0.0 pts 1 Points/Error Handling Test Case & 1 Points/Valid Input Test Case 4 Error Handling Test Cases & 2 Valid Input Test Cases. The function is implemented correctly, and few or all test cases pass.	0 pts All Test Cases Failed Incorrect implementation or none of the test cases pass.	6 pts

Criteria	Ratings		Pts
objUtils.js - areObjectsEqual Test cases used for grading will be different from assignment examples.	13.5 to >0.0 pts 1.5 Points/Error Handling Test Case & 3.75 Points/Valid Input Test Case 4 Error Handling Test Cases & 2 Valid Input Test Cases. The function is implemented correctly, and few or all test cases pass.	0 pts All Test Cases Failed Incorrect implementation or none of the test cases pass.	13.5 pts
objUtils.js - calculateObject Test cases used for grading will be different from assignment examples.	13.5 to >0.0 pts 1.5 Points/Error Handling Test Case & 3.75 Points/Valid Input Test Case 4 Error Handling Test Cases & 2 Valid Input Test Cases. The function is implemented correctly, and few or all test cases pass.	0 pts All Test Cases Failed Incorrect implementation or none of the test cases pass.	13.5 pts
objUtils.js - combineObjects Test cases used for grading will be different from assignment examples.	6 to >0.0 pts 1 Points/Error Handling Test Case & 1 Points/Valid Input Test Case 4 Error Handling Test Cases & 2 Valid Input Test Cases. The function is implemented correctly, and few or all test cases pass.	0 pts All Test Cases Failed Incorrect implementation or none of the test cases pass.	6 pts
Assignment Submitted	1 pts Assignment Submitted At least one of the test cases passes in the assignment.	0 pts All Test Cases Failed Incorrect implementation or none of the test cases pass in the assignment.	1 pts
Total Points: 100			