EECS 1012: LAB 8 – Recursion (part of computational thinking)

A. IMPORTANT REMINDERS

1. Lab8 is due on **Friday (Aug 5)** 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 8 posted on eClass no later than the due date of this lab, i.e., 1:30pm on Friday (Aug 5).
3. You are welcome to attend the lab session on Aug 5th, 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 through them carefully.
2. Review and implement the following recursive functions: **Factorial(n)** and **Fibonacci(n)** from slides.
3. Review the following links:

   https://www.freecodecamp.org/news/quick-intro-to-recursion/
   https://www.youtube.com/watch?v=YZcO_jRhvxs
   https://www.youtube.com/watch?v=B0NtAFf4bvU
   https://levelup.gitconnected.com/introduction-to-recursion-7848231b0d1b

4. Trace the following four calls: **Factorial(3)**, **Factorial(4)**, **Factorial(5), Fibonacci(4)**, and **Fibonacci(5).**

C. GOALS/OUTCOMES FOR LAB

1. To practice more computational thinking by using recursion.
2. To become familiar with how to trace recursive functions.

D. TASKS

Solving the computational thinking problem using Recursion (not solely iteration) and trace them for some sample inputs.

E. SUBMISSIONS

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

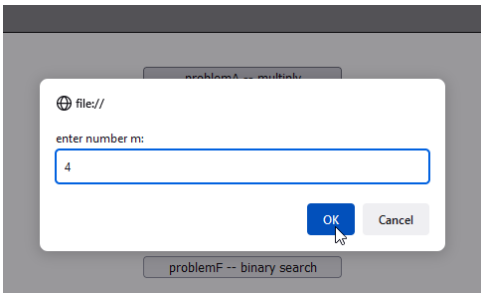
F. FURTHER DETAILS

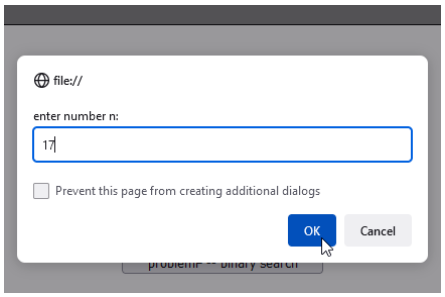
In this exercise, you complete the several recursive functions and trace them for some sample inputs.

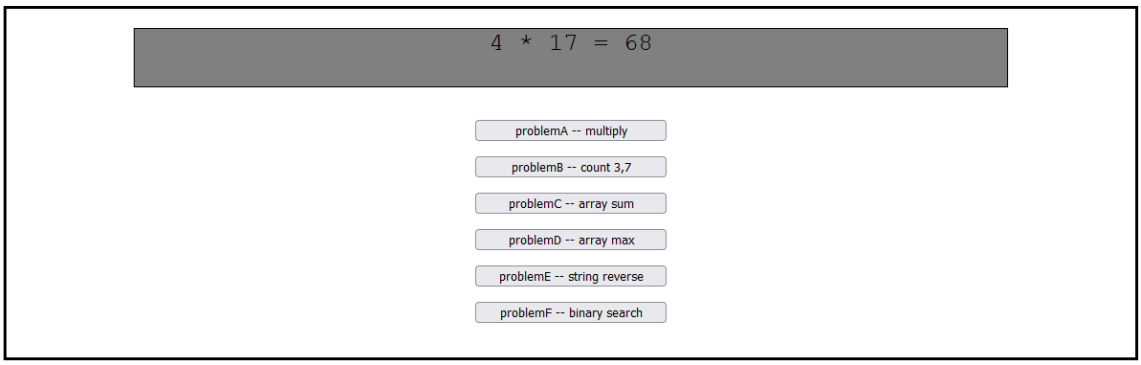
~~**Task 0:** The given HTML file includes one button for problem A. Add five more buttons as shown in samples blow, and add corresponding calls to function **problemB~F**.~~

Task A. Function **problemA()** receives two natural numbers and then calls sub-program function **multiply(m, n)** to compute their multiplication. For instance, if the inputs are 5 and 6, the function should output 30. **multiply**() should be a recursive function. You are NOT allowed to use the multiplication operator (i.e. *), also should not have loops in the function.

Approach: In the body of function **multiply**(), check if n is equal to 0 and/or check if n is equal to 1 and return a reasonable value. This is the base case. Hint: what is m*0 or m*1? For the rest of the body of your function does a recursive call, using the recursive definition, as discussed in class.







Now draw the following trace table and trace your function **multiply(m,n)** for the value of **m** and **n** initially being 5 and 6, respectively. As a guideline, we have populated the tables with the values for the first 2 calls and the corresponding returned values for each of those calls. Note that in this example, the first table is populated top-down (i.e. time elapses from row x to row x+1) whereas the returned value table is populated bottom-up (i.e. time elapses from row x to row x-1). You may want to use debugging features or simply console.log to observe these values.

call #	m	n
1	5	6
2	5	5

returned value
30
25

time

Once you are done, take a screenshot of picture of the table and name it **taskA.jpg**

Task B. Function **problemB()** receives a natural number and then call sub-program function **count37(n)** to count how many digits in the number are equal to 3 or 7. For instance, if the input is 772, the program should output 2, because there are two sevens there. Another example, if the input is 14367, the program should output 2 as there is a 3 and a 7 in it. You have solved this problem in lab5 using while loops. Here **count37()** should be implemented as a recursive function where no loop should be used.

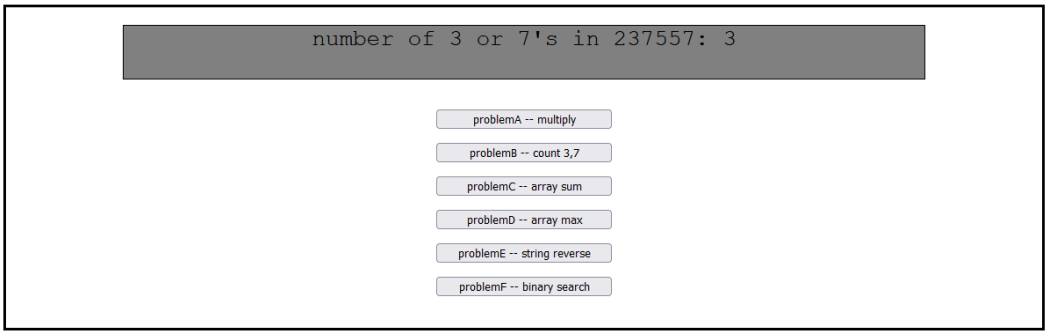
Approach: As demonstrated in class, in the body of the function, we can first check if parameter n is equal to 0. This is the base case. If so, just return 0. Otherwise separate its least significant digit as you did in lab5; If the least significant digit is equal to 3 or 7, recursively call your function with n' which is n with last digit removed (which is a smaller version of problem and assume we have such a solution), and add it up to 1 (because you just observed a digit 3 or 7). If the least significant digit is not a 3 or 7, just recursively call your function on n'. Note that n' eventually is reduced to 0, which is the base case and the program terminates.

Also as demonstrated in class, we can have a different base case. We can first check if n is a single digit (< 10).

If it is, then we can just return 1 if it is 3 or 7, and return 0 if it is not. This is the base case and the program terminates. If n is not a single digit, we separate n and proceed recursively as above. Note that n' eventually is reduced to a single digit, which is the base case and the program terminates.

You can choose one of the approaches for defining base case. After your function works, you are encouraged to try another approach.

You should just work with numbers in this question. Converting the number to string results in a 0 for this question. Also, writing any explicit loop in your code results in a 0 for this question.



Now draw the following trace table and trace your function **count37(n)** for when n is initially 237557. As a guideline, we have populated the tables with the values for the first few calls and the corresponding returned values for each of those calls.

Note that in this example, the first table is populated top-down (i.e. time elapses from row x to row x+1) whereas the returned value table is populated bottom-up (i.e. time elapses from row x to row x-1). You may want to use debugging features or simply console.log to observe these values.

Once you are done, take a screenshot of picture of the table and name it **taskB.jpg**

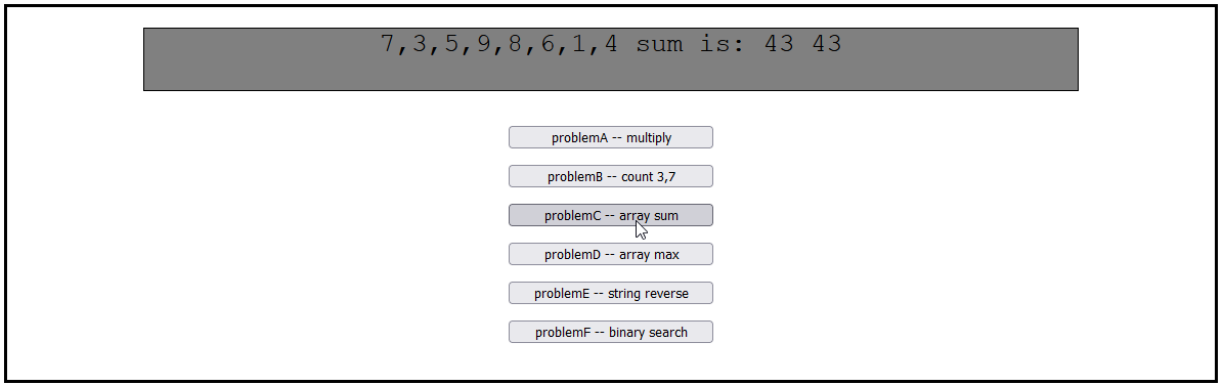
call #	n	lsd
1	237557	7
2	23755	5
3	2375	5

returned value
3
2
2

time

Task C. Function **problemC()** creates an array of numbers and then call two recursive sub functions to calculate the sum of values in the array. Thinking recursively, the idea is that the sum can be calculate by adding first element with the sum of the rest of the array (which is a smaller version of problem and assume we have such a solution). How to pass the rest of the array to recursive function call? As mentioned in class, one approach is to pass a new sub-array in each recursive call. This has been implemented for you in function **arraySum0(),** using *slice()* method in JS to create a sub-array. Another approach, which you are going to implement in function **arraySum()** , is to pass the original array and a "sliding" index, starting from 0. The index signifies the function that elements with smaller indexes have been processed and thus this call should start processing at the current index. Both approaches should generate the same results. You may want to

change the values in the array to test your solution.



Now draw the following trace table and trace the given function **arraySum0(arr)** for when arr is initially [7, 3, 5, 9, 8, 6,1, 4]. As a guideline, we have populated the tables with the values for the first few calls and the corresponding returned values for each of those calls.

Once you are done, take a screenshot of picture of the table and name it **taskC-1.jpg**

call #	arr	returned value
1	[7, 3, 5, 9, 8, 6, 1, 4]	43
2	[3, 5, 9, 8, 6, 1, 4]	36
3	[5, 9, 8, 6, 1, 4]	33
…	…	…

time

Now draw the following trace table and trace your function **arraySum (arr, index)** for when arr is initially [7, 3, 5, 6, 9, 8, 1, 4] and index is initially 0. As a guideline, we have populated the tables with the values for the first 4 calls and the corresponding returned values for each of those calls.

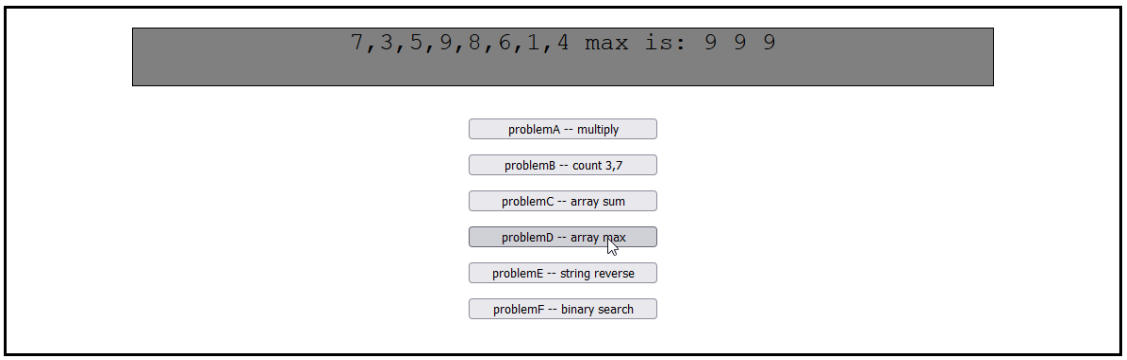
Once you are done, take a screenshot of picture of the table and name it **taskC-2.jpg**

call #	arr	index	returned value
1	[7, 3, 5, 6, 9, 8, 1, 4]	0	43
2	[7,3, 5, 6, 9, 8, 1, 4]	1	36
3	[7,3,5, 6, 9, 8, 1, 4]	2	33
…	…	…	…

time

Task D. Function **problemD()** creates an array of numbers and then call three sub-program functions to find the max value in the array. One function is iterative, the other two are recursive. You are to implement both of them.

Thinking recursively, the max value can be determined by comparing the first element with the max of the rest of the array (which is a smaller version of problem and assume we have such a solution). Again, to pass the rest of the array to a recursive function call we can pass a new sub-array in each recursive call or pass original array and a "sliding" index, starting from 0. Here you need to implement both the two approaches, i.e., passing the (sub)array, or passing original array with a sliding index. Both functions should generate the same result. In function **arrayMaxRecursiveA(arr)** where you pass the (sub)array only, you use slice() to generate and pass the subarray. In function **arrayMaxRecursiveB(arr, index)**, you always pass original array with a index. In this function, you should not use slice() in this function.



Now draw the following trace table and trace your function **arrayMaxRecursiveA (arr)** for which initially arr is [7, 3, 5, 9, 8, 6, 1, 4]. As a guideline, we have populated the tables with the values for the first few calls and the corresponding returned values for each of those calls.

Once you are done, take a screenshot of picture of the table and name it **taskD-1.jpg**

call #	arr	returned value
1	[7, 3, 5, 9, 8, 6, 1, 4]	9
2	[3, 5, 9, 8, 6, 1, 4]	9
3	[5, 9, 8, 6, 1, 4]	9
...	...	...

time

Now draw the following trace table and trace your function **arrayMaxRecursiveB (arr, index)** for which initially arr is [7, 3, 5, 9, 8, 6, 1, 4] and the index is 0. As a guideline, we have populated the tables with the values for the first few calls and the corresponding returned values for each of those calls.

Once you are done, take a screenshot of picture of the table and name it **taskD-2.jpg**

call #	arr	index	returned value
1	[7, 3, 5, 9, 8, 6, 1, 4]	0	9
2	[7,3, 5, 9, 8, 6, 1, 4]	1	9
3	[7,3,5, 9, 8, 6, 1, 4]	2	9
...	...	...	...

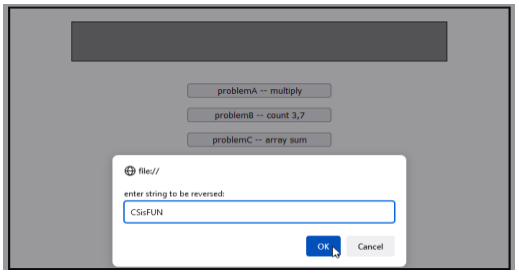
time

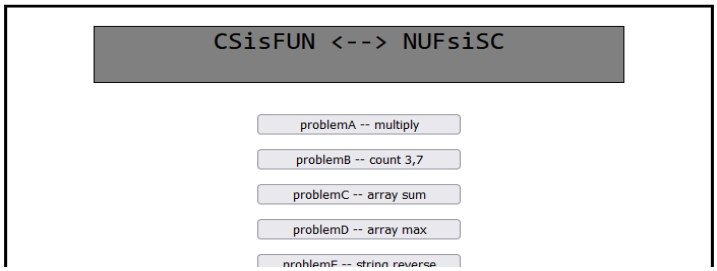
Task E. Function **problemE()** receives a string and call a sub-program **reverse(s)** to compute its reverse. For instance, if the input is *CSisFUN*, the function should output *NUFsiSC*. The sub-program function **reverse(s)** is a recursive function.

Thinking recursively, the function first checks if the length of s is less than 2. If so, just return s because the reverse of a string of length less than 2 is the same string. This is the base case. If length of s is no smaller than 2, make a new string that excludes the first character of the old string. In JS, the substring() method can do this. And now call the function recursively with the new string (which is a smaller version of problem because the array size is reduced by 1). Assuming the solution to the smaller version of problem works, we can concatenate the result with the character that you excluded. As a hint, part of your code will look like this:

```
return  reverse(newString)  +  s[0];
```

Writing any explicit loop in your code results a 0 mark for this question.





Now draw the following trace table and trace your function **reverse(s)** when the input S is initially *CSisFUN*. As a guideline, we have populated the tables with the values for the first few calls and the corresponding returned values for each of those calls.

Once you are done, take a screenshot of picture of the table and name it **taskE.jpg**

call #	s	s[0]	returned value
1	CSisFUN	C	NUFsiSC
2	SisFUN	S	NUFsiS
3	isFUN	i	NUFsi
…	…	…	…

time

Task F. Function **problemF()** creates an sorted array of 20 non-descending numbers exists, and it receives a number from user and then call a sub-program **find()** to determine if that number exists in the array or not.

For instance, assume the array is:

[8, 9, 9, 12, 13, 13, 13, 15, 20, 100, 100, 101, 123, 129, 300, 330,390, 400, 403,407]

Now, assume the user enters 129, the program should output *true*. But, if the user enters 150, the program should output *false*.

Does this sound familiar? Yes, you have seen this question in Assignment 1.

Approach: since the array is sorted, instead of a sequential search, we can do the so-called "binary search". You have done binary search in Assignment1, using iterative (loops) solutions. Here we solve the problem with recursive approach.

Recall that in assignment1, we compare the search key with the middle value of the array. If the key equals to the middle value, we find it and stop. If the key is smaller than the middle value, we search the first half of the array as the key could not be in the 2nd half. Otherwise search the 2nd half of the array as the key could not be in the 1st half. In assignment 1, we used loop and indexes to direct the search area.

Thinking recursively, if the key is smaller than the middle element, then the answer to whether key is in the array depends on whether the key is in the left subarray, which is a smaller version of problem and assume we have the solution. If the key is larger than the middle element, then the answer to whether key is in the array depends on whether the key is in the right subarray, which is a smaller version of problem and assume we have the solution. In either case we can return the answer to the smaller version of problem. Thus, in the

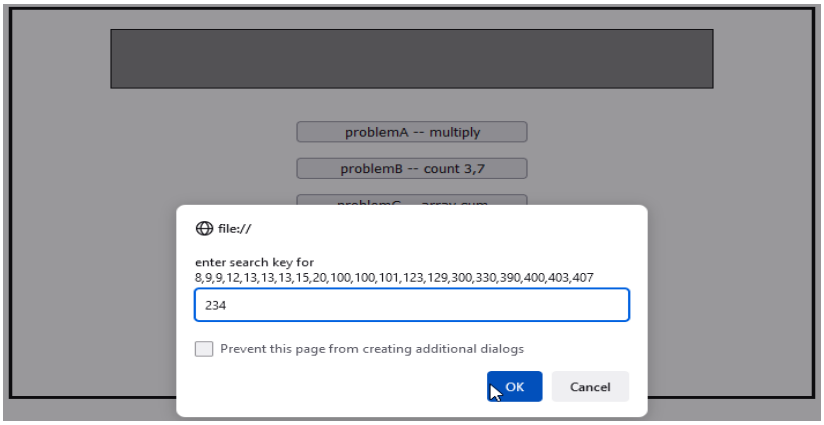
corresponding sub-array, we repeat the work recursively, returning the result of the recursive call. We pass the sub-array to be searched next to a recursive call. To pass sub-arrays to the recursive call, we can pass a new sub-array or pass two sliding indexes. Here we use the sliding indexes. The signature of our function is **find(x, A, i, j)** where **x** is the number we are looking for in array **A**, and the first index of the array is **i** and its last index is **j**. That means we want to determine whether **x** exists in **A** anywhere from index **i** to index **j**.

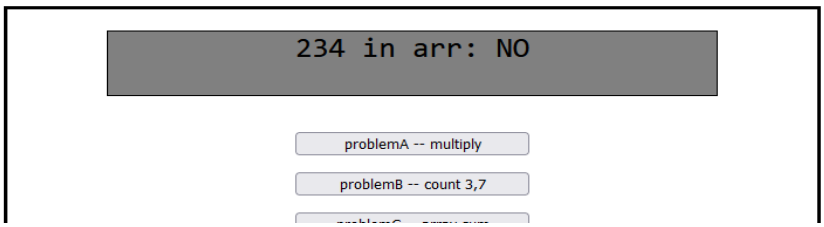
In the body of your function, you can compare **x** with the item that is in the middle of the array. The middle of the array is at index **(i+j)/2**, let's call it **mid**. If **x==a[mid]**, the program returns true. If **x<a[mid]**, you should recursively call your function to search the first half of the array, i.e. from **i** to **mid-1**. If **x>a[mid]**, you should recursively call your function to search the second half of the array, i.e. from **mid+1** to **j**. As a hint, part of your code will look like these:

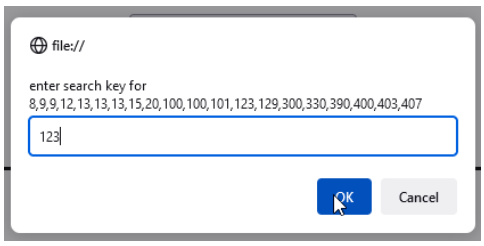
```
if …

  return  find(x,A,i,mid-1);    // returns whether the key is in the left subarray

else …

  return  find(x,A,mid+1,j);  // returns whether the key is in the right subarray
```

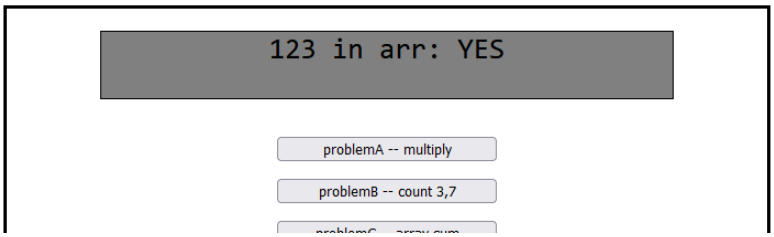
Note that **i** and **j** are getting closer to each other by every recursive call. Once **i** and **j** 'cross' each other, i.e.,**i** becomes greater than **j**, that means all potential elements has been compared, and **x** does not exist in **A** and the algorithm should return **false**. This is the base case.

Writing any explicit loop in your code results in a 0 mark for this question.









Now draw the following trace table and trace your function **find(x, A, i, j)** for when **x** is intially 123, and **A** is the array given in the example above. As a guideline, we have populated the tables with the values for the first few calls and the corresponding returned values for each of those calls.

Once you are done, take a screenshot of picture of the table and name it **taskF.jpg**

call #	A	X	i	j	mid	A[mid]	returned value
1	[8..407]	123	0	19	9	100	true
2	[8..407]	123	10	19	14	300	true
…		…			…		…

time

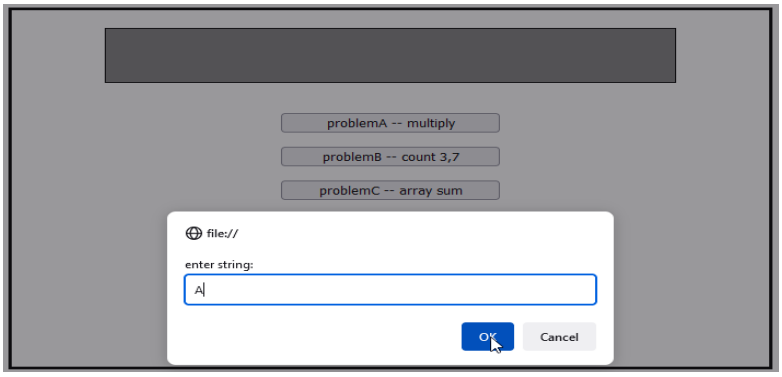
Task G. Function **problemG()** reads a string and call a sub-program **symmetric ()** to determine if the string is symmetric or not.

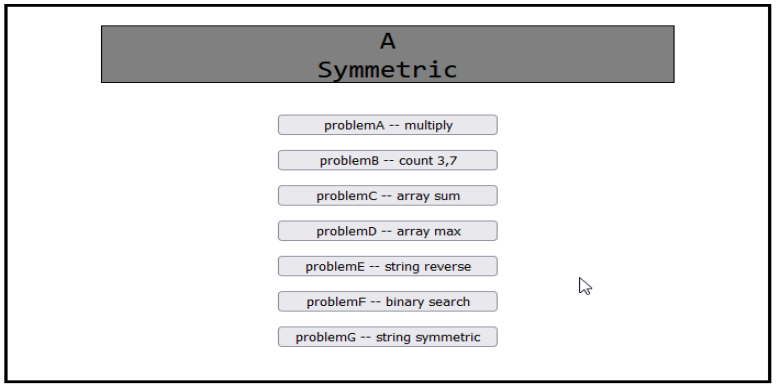
As defined in Assignment 1, a string is symmetric if it reads the same backward as forward, e.g., "*a*" "*aa*", "*dad*", "*madam*", "*that is a si taht*". On the other hand, "*ab*" "*random*" or "*Dad*" "*this is that*" are not symmetric.

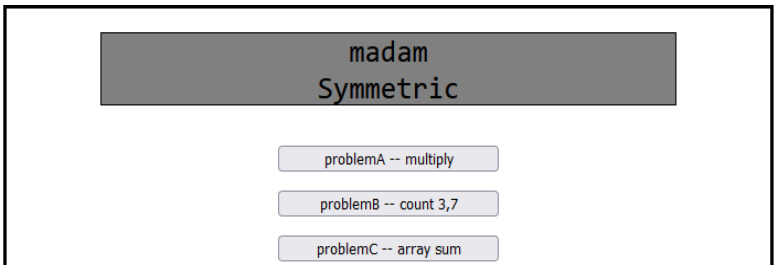
Approach: In assignment 1, you developed an iterative solution. You checked the first and the last characters, and if they are the same, use loop to check 2nd and the 2nd last characters, etc. At any point, if mismatch is found, stop the loop and output false. If the loop comes to an end and all pairs of characters are the same, output 'yes'.

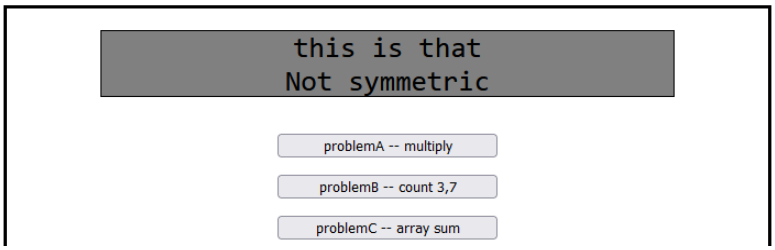
Thinking recursively, if the first and last characters are same, then whether the string is symmetric depends on whether the rest string (with the first and last character excluded) is symmetric, which is the smaller version of problem and assume we have answer to it. So we return the answer to the smaller version of problem. Specifically, we repeat the work recursively. We pass the sub-string with first and last character excluded to a recursive call. The recursive call checks the first and last character of the passed sub-string. To pass sub-strings to the recursive call, we can create a new sub-string or use sliding indexes. Here we use the sliding indexes. We pass the original array and two sliding indexes. The signature of our function is **symmetric (str, i, j)** where **str** is the original string, and **i** and **j** are the indexes of the characters in the string to be compared. If characters at index **i** and **j** are same, update **i** and **j** and pass to the recursive call.

Note that **i** and **j** are getting closer to each other by every recursive call. If **i** and **j** 'meets' or crosses, that means all corresponding characters in the string have been compared, and no mismatch is found – **i** and **j** won't meet if mismatch was found. In this case the algorithm returns true as the string is symmetric. This is the base case.









Now draw the following trace table and trace your function **symmetric(str, i, j)** for when **str** is **"abc2xyz2cba"**. As a guideline, we have populated the tables with the values for the first few calls and the corresponding returned values for each of those calls.

Once you are done, take a screenshot of picture of the table and name it **taskG.jpg**

call #	str	i	j	str[i]	str[j]	returned value
1	"abc2xyz2cba"	0	11	a	a	false
2	"abc2xyz2cba"	1	10	b	b	false
…				…		…

time

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

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

- Count elements in an array which satisfy some criterion
    - Even/odd number, > 5, multiples of 5…
- ~~Check if a string/array is a Palindrome "madam" "abc010cba"~~
- Find min value in an integer array

2. Add more questions to your learning kits. If you add recursion problems you saw in this lab and recursion lecture slides, and don't not have flowcharts, **you can use the trace table instead of the flowcharts**.

H. SUBMISSION

eClass submission

You should already have a **Lab08** folder that contains the following files:

lab08.html and **lab08.js** and **style.css**

taskA.jpg, taskB.jpg, taskC-1.jpg taskC-2.jpg, taskD-1.jpg, taskD-2.jpg, taskE.jpg, taskF.jpg taskG.jpg

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

Compress the Lab08 folder (.zip **or .**tar **or** .gz**), and then submit the (single) compressed file on eClass.**