# Lab 1:

## 1 First:

Is it easier to implement this function recursively or iteratively?

Answer: For someone who has not worked with recursion the iterative version would seem to be easiest to implement. The recursive version does however contain fewer lines of code and it’s a very simple execution. The recursive version is an easier implementation of this function.

Are there any pros/cons of the two approaches?

Answer: The recursive version is not simple to understand at a glance compared to the iterative version. To understand the return-unfolding phase takes some figuring out. The recursive function does not need an array what-so-ever compared to the iterative version that iterates the array and fills it up with the chars then reads the array backwards.

## 2 Second:

### Recursive:

The tests written for the recursive function works by keep prompting the user for input. If the user enters exit then the program will shut down. If anything else is entered then the program will return that string in reverse. The tests validate the correctness of the implementation because it allows the user to enter whatever string they want in order to check that it works.

### Iterative:

The tests written for the iterative function works by giving the user several different commands. Every command represents a function within the program. The user has the choice to test all functions within the stack or to test the actual reverse-string implementation. The tests validate the correctness of the implementation because it allows the user to test all functions in order to see if they work as intended. The user can push a number to the stack and then pop it in order to see that the stack works. The user can also choose to reverse a string in order to check that the returned string is reversed.

## 3 Third:

The tests written for the implementation works by giving the user several different commands. Every command represents a function within the program. The user has the choice to test all functions within the generic iterable FIFO-queue based on a double linked circular list. The user can choose to enqueue an element followed by checking the size of the queue in order to check that the queue works as expected. After each insertion/deletion the program prints all of the elements in the queue followed by all of the nodes and their next/prev node. This was in order to validate that the queue fulfilled the requirements of a doubly-circular-linked-list. The tests validate the correctness of the implementation because it allows the user to test all functions in order to see if they work as intended.

## 4 Fourth:

The tests written for the implementation works by giving the user several different commands. Every commands represents a function within the program. The user has the choice to test all functions within the generic iterable circular linked list. The user can add an element by either using addToStart or addToEnd and remove elements by using either removeFromStart or removeFromEnd. After each insertion/deletion the program prints all of the elements in the queue followed by all of the nodes and their next node. This was in order to validate that the queue fulfilled the requirements of a circular-linked-list. The tests validate the correctness of the implementation because it allows the user to test all function in order to see if they work as intended.

## 5 Fifth:

The tests written for the implementation works by giving the user several different commands. Every command represents a function within the program. The user has the choice to test all functions within the generalized queue. The user can enqueue 3 different elements with the most recent element having index 1 and then remove index 2 for example which will remove the element in the middle of the queue. The user can among other things also check the size of the queue in order to check that everything works as intended this validates the correctness of the implementation.

## 6 Sixth:

The tests written for the implementation works by giving the user several different commands. Every command represents a function within the program. The user has the choice to test all functions within the ordered queue. If the user adds an element to the queue with the value 10 and then proceeds to add another either more than or less than 10. The queue will order it in ascending order. Since this happens at insertion, using dequeue works as usual. The tests validate the correctness of the implementation because it allows the user to test all function in order to see if they work as intended.

## 7 Seventh:

The tests written for the implementation works by giving the user several different commands. Every command represents a function within the program. The user has the option to test the actual stack or to test the balance-parentheses function. The program will keep prompting for input until the user enters ‘exit’ which will terminate the instance of the program. The tests validate the correctness of the implementation because it allows the user to test the balance parentheses function or the stack separately.

The choice of algorithm was chosen because it fulfilled the demands of storing the open parentheses.

The time complexity of the 7 Seventh part is O(n) since it iterates every character only once.

The memory complexity of the 7 Seventh parts is also O(n). The program only pushes the open parentheses to the stack and does not store the closed parentheses in any particular way. This results in n \* ½ since only half of the characters are stored. The constants when calculating complexity are overlooked which in turn leads to a memory complexity of O(n).