Exercise 1

import random

def shuffle(x):

arrayRange = len(x)

if not arrayRange > 2: #checking if array has more than two elements and prints an error if it doesn't

raise AssertionError('Array is too short to shuffle!')

for i in range(arrayRange):

swap = random.randrange(arrayRange - 1)

swap += swap >= i

x[index], x[swap] = x[swap], x[i]

return Array

Array = [1,2,3,4,5,6,7,8,9,10]

print (shuffle(Array))

#O(n), where n is the amount of elements in the array

Exercise 2

def factorial(n):

if n == 0:

return 1

else:

return n \* factorial(n-1)

def trailingZeroes(factorialResult):

stringConverted = str(factorialResult)

return len(stringConverted)-len(stringConverted.rstrip('0'))

n=int(input("Please enter a number"))

print("There are ",trailingZeroes(factorial(n))," trailing zeroes from ", n,"\*",n ," which results in ", factorial(n))

#O(n), each line of code is being run once

Exercise 3

*baseNumber <- 1*

*while baseNumber\* baseNumber <= parameter*

*baseNumber <- baseNumber +1*

*else*

*answer <- baseNumber -1*

parameter = int(input("What is your parameter?"))

n=1 #base case

#multiplying base case by itself until it will be bigger than given parameter

while n\*n<=parameter:

n=n+1

#when it becomes bigger we know that our answer is one less than n

else:

print(n-1)

Exercise 5

*ADDMATRICES(X,Y)*

*answer <- [[0,0],[0,0]]*

*for n in length(X) //to go through rows*

*for m in length(X[0]) //to go through columns*

*answer[n][m] <- X[n][m] + Y[n][m]*

*SUBMATRICES(X,Y)*

*answer <- [[0,0],[0,0]]*

*for n in length(X) //to go through rows*

*for m in length(X[0]) //to go through columns*

*answer[n][m] <- X[n][m] + Y[n][m]*

*MULTMATRICES(X,Y)*

*answer <- [[0,0],[0,0]]*

*for n in length(X) //to go through rows*

*for m in length(X[0]) //to go through columns*

*for l in length(Y)*

*answer[n][m] <- X[n][m] + Y[n][m]*

*//A= N-M=N-(L+L)*

*L=addMatrices(B,C)*

*M=addMatrices(L,L)*

*N=multMatrices(B,C)*

*A=subMatrices(N,M)*

*//run time, due to triple nested loop, is O(n^3)*

Exercise 6

*REVERSESENTECE(sentence)*

*sentence <- sentence.split*

*length <- len(sentence)*

*lastPostion <- length – 1*

*iterations <- length//2*

*for n in length(0, iterations)*

*replacement <- sentence[lastPostion]*

*sentence[lastPostion] <- sentence[n]*

*sentence[n] = replacement*

*lastPostion <- lastPostion - 1*

def sentenceReverse(lst):

lst = lst.split()

length = len(lst)

lastPostion = length - 1

n = length//2 #How many times will the functions go

for n in range(0, n):

temp = lst[lastPostion] # Creating temporary postion for incoming swap

lst[lastPostion] = lst[n] #swap

lst[n] = temp #assining temp back to sequence

lastPostion -= 1 #leaving changed number at the end and going one lower

return lst

lst = "This Is an amazing nonfuctional list wannabe"

print(sentenceReverse(lst))

#O(n), for n is the variable 'n' aka how many times will fuction run

Exercise 7

*PRIMECHECK(VAR, DIVISOR)*

*divisor <- divisor – 1 //Divisor is always smaller than dividend and we want it to go lower with each iteration*

*if divisor = 0 or 1 //when during iteration divisor reaches 1 we know that nothing earlier able to divide our variable*

*and is divisor is 0 it means variable was 1 which is a prime number*

*“this is prime number”*

*else if var%divisor = 0 //if our variable is dividable by divisor without any reminder it means it’s not a prime number*

*“this is not prime number”*

*else*

*return primecheck(var, divisor)*

def primeCheck(val,maxRange):

maxRange= maxRange-1 #setting a divisor

if maxRange <0:

print ("Please enter value higher than 0")

return None

elif maxRange == 0 or maxRange == 1:

print ("This is prime number")

return True

elif val%maxRange==0:

print ("this is not prime nubmer, the number '", val,"' is dividable by", maxRange)

return False

else:

return primeCheck(val,maxRange)

val=int(input("what nubmer do you wish to check?"))

maxRange=val

print(primeCheck(val,maxRange))

#O(n), the worst case is the amount of iterations equal to variable 'maxRange'

Exercise 8

*VOWELSREMOVAL(INPT)*

*If not inpt*

*return inpt*

*elif inpt[0] in vowelsString*

*return vowelsRemoval(inpt[1:])*

*else*

*return inpt[0] + vowelsRemoval(inpt[1:])*

def removeVowels(sentence):

if not sentence: # case for empty string

return sentence

elif sentence[0] in "eaiouyEAIOUY": #checks if first letter is a vovel and removes it if is it gets discarded and fuction is called again

return removeVowels(sentence[1:])

return sentence[0] + removeVowels(sentence[1:]) #if none of the above is true then first letter is probably not a vovel and gets saved

sentence=input("What is your given sentence/word?")

print(removeVowels(sentence))

Exercise 9

*BINARYSEARCH(LST, X, Y)*

*if length(lst) = 0 //safe case*

*return False*

*else*

*middle =length(lst)//2 //amount of iterations*

*if lst[middle]>x and lst[middle]<y*

*return True*

*else*

*if lst[middle]<x*

*return binarySearch(lst[middle+1:],x,y) //if lower number is bigger than middle value of the list higher is being recursively called*

*else*

*return binarySearch(lst[:middle],x,y) //opposite to comment above*

def binarySearch(aList, x, y):

if len(aList)==0:

return False

else:

middle = len(aList)//2

if aList[middle]>x and aList[middle]<y:

return True

else:

if aList[middle]<x:

return binarySearch(aList[middle+1:], x, y)

else:

return binarySearch(aList[:middle], x, y)

theList = [1,2,3,4,5,6,7,8,9,10,11,12]

print (binarySearch(theList, 8, 10))

#O(n), n depends on the amount of numbers in a list

Exercise 12

class BinTreeNode(object):

def \_\_init\_\_(self, value):

self.value=value

self.left=None

self.right=None

def tree\_insert( tree, item):

if tree==None:

tree=BinTreeNode(item)

else:

if(item < tree.value):

if(tree.left==None):

tree.left=BinTreeNode(item)

else:

tree\_insert(tree.left,item)

else:

if(tree.right==None):

tree.right=BinTreeNode(item)

else:

tree\_insert(tree.right,item)

return tree

def inOrder(tree):

lst = [] # initialze stack

done = 0

while not done:

if tree is not None:

lst.append(tree)

tree = tree.left

else:

if(len(lst) >0 ):

tree = lst.pop()

print (tree.value)

tree = tree.right

else:

done = 1

if \_\_name\_\_ == '\_\_main\_\_':

t=tree\_insert(None,6);

tree\_insert(t,10)

tree\_insert(t,5)

tree\_insert(t,2)

tree\_insert(t,3)

tree\_insert(t,4)

tree\_insert(t,11)

inOrder(t)