**GitHub link:** <https://github.com/DamianLC/210CT>

# Week 1

**Task 1**

import random

##function to randomly shuffle a list of numbers

def randShuffle(n):

shuffledLst = [] #empty list

while len(n) > 0:

randNum = random.choice(n) #picks a random number

shuffledLst.append(randNum) #adds the number to the list

n.remove(randNum) #removes the number from the orignal list

return(shuffledLst)

answer = randShuffle([5,3,8,6,1,9,2,7])

print("The shuffled list is: ", answer)

**Task 2**

##function to count the amount of trailing 0s in a factorial number

def factorialZeros(number):

zeroCount = 0 #counter for the trailing 0s

factorialNum = 1 #start of the factorial caluclation

revFactorial = []

for i in range(1,number+1): #iterates through each number of the user input to calculate the factorial

factorialNum = factorialNum \* i

for reverse in str(factorialNum): #reverses the factorial by placing each integer at the beginning

revFactorial.insert(0, reverse)

for zeros in revFactorial: #counts the amount of zeros in a factorial until it reaches a number that isn't a zero

if zeros == "0":

zeroCount += 1

else:

break

return(zeroCount)

**Task 3**

1. Calling the function is the input and the shuffled list is the output.
2. It will terminate when there are no more elements in the original list.
3. Yes.
4. It will produce a shuffled list every time.
5. Calling the function is the input and the number of trailing zeros is the output.
6. The function will terminate when it reaches a number that is not a 0 when counting the amount of 0s.
7. Yes.
8. It will output the number of trailing zeros for any number.

# Week 2

**Task 1**

**Pseudocode**

import MATH

PERFSQUARE(NUMBER)

num <- square root number

num <- round down num

num <- square num

RETURN num

**Code**

import math

##function to find the highest perfect square which is less or equal to its parameter

def perfSquare(number):

num = math.sqrt(number)

num = math.floor(num) #rounds down num

num = num\*\*2

return(num)

**Task 2**

The bigO notation for Week 1 Task 1 is O(n).

The bigO notation for Week 1 Task 2 is O(n).

**Task 3**

MATRIXADD(A,B)

result <- list

rowA <- length(A)

colA <- length(A[0])

rowB <- length(B)

colB <- length(B[0])

if rowA rowB

RETURN false

if colA colB

RETURN false

for i to range[rowA]

row <- empty list

for j to range[colA]

row <- append A[i][j] + B[i][j]

result <- append(row)

RETURN result

MATRIXSUB(A,B)

result <- list

rowA <- length(A)

colA <- length(A[0])

if rowA rowB

RETURN false

if colA colB

RETURN false

for i to range[rowA]

row <- empty list

for j to range[colA]

row <- append A[i][j] - B[i][j]

result <- append(row)

RETURN result

MATRIXMULT(A,B)

rowA <- length(A)

colA <- length(A[0])

rowB <- length(B)

colB <- length(B[0])

if colA rowB

RETURN false

result <- list

for i to range[rowA]

for j to range[cola]

for k to range[rowB]

result[i][j] += a[i][k] \* b[k][j]

RETURN result

MATRIXFACTORMULT(A,B)

result <- list

rowA <- length(A)

colA <- length(A[0])

for i to range(rowA)

row <- list

for j to range(cola)

row <- append A[i][j] \* b

result <-append(row)

RETRUN result

A = B\*C – 2\*(B+C)

A1 <- MATRIXMULT(B,C)

A2 <- MATRIXADD(B,C)

A3 <- MATRIXFACTORMULT(A2,2)

A <- A1 – A3

# Week 3

**Task 1**

**Pseudocode**

REVWORDS(STRING)

splitString <- STRING.split()

revLst <- empty list

revString <- empty string

for reverse to splitString

revLst.insert(0, reverse)

for word to revLst

revString <- revString + word + “ “

RETURN revString

**Code**

##function to reverse the words in a strings

def revWords(string):

splitString = string.split() #splits the string into a list

#print(splitString)

revLst = []

revString = ""

for reverse in splitString: #inserts each word in the string as the first

revLst.insert(0,reverse)

#print(revLst)

for word in revLst: #reconstructs the string

revString += word + " "

return(revString)

**BigO**

O(n)

**Task 2**

**Pseudocode**

ISPRIME(NUMBER, PRIMECHECK = NONE)

if NUNBER <= 1

RETURN false

if PRIMECHECK is NONE

PRIMECHECK = NUMBER – 1

While PRIMECHECK >= 2

If NUMBER mod PRIMECHECK = 0

RETURN false

else

RETURN ISPRIME(NUMBER, PRIMECHECK – 1)

RETURN true

**Code**

##function to check whether a number is prime or not by dividing the number by all numbers below it

def isPrime(number,primeCheck = None):

if number <= 1: #special case

print(number, "is not a prime number")

return False

if primeCheck is None: #assigns a number to the second parameter of the function

primeCheck = number - 1 #this is the number that will be reduced with each recursve call to check if a number is prime

while primeCheck >= 2: #base case

if number % primeCheck == 0: #if the reminder of number divided by primeCheck is equal to 0, number isn't prime

print (number, "is not a prime number")

return False

else:

return isPrime(number, primeCheck-1) #call the function again decreasing the primeCheck by 1

print (number, "is a prime number")

return True

**Big O**

O(n)

**Task 3**

**Pseudocode**

REMOVEVOWELS(STRING)

If STRING = empty

RETURN false

elif STRING[0] to “aeiouAEIOU”

RETURN REMOVEVOWELS(STRING[1:])

RETURN STRING[0] + REMOVEVOWELS(STRING[1:])

**Code**

def removeVowels(string):

if string == "":

return string

elif string[0] in "aeiouAEIOU": #checks the character which has index 0 if its a vowel, if true it

calls the function again starting from the

return removeVowels(string[1:]) #next character

return string[0] + removeVowels(string[1:]) #if the character at index 0 is not a vowel, it calls the

function from the next character but adds the

#one it was checking to the beginning of the string

**Big O**

O(n)

# Week 4

**Task 1**

**Pseudocode**

BINARYSEARCH(SEQUENCE, TARGETLOW, TARGETHIGH)

first <- 0

last <- length(sequence) – 1

found <- false

while first<= last and not found

midpoint <- (first+last) // 2

if TARGETLOW <= SEQUENCE[midpoint] <=TARGETHIGH

found <- true

else

if TARGETHIGH < SEQUENCE[midpoint]

last <- midpoint – 1

else

first <- midpoint +1

RETURN found

**Code**

##binary search function that searches for a value between two given points

def binarySearch(sequence, targetLow, targetHigh):

first = 0

last = len(sequence) - 1

found = False

while first<=last and not found:

midpoint = (first + last)//2

if targetLow <= sequence[midpoint] <= targetHigh: #if the midpoint is between the two given

found = True # points then the number is in the list

else:

if targetHigh < sequence[midpoint]: #if the higher point is smaller than the midpoint

last = midpoint - 1 #look in the left hand side of the list

else:

first = midpoint + 1 #if the higher point is bigger than the midpoint

return found #look in the right hand side of the list

print(binarySearch([2,3,5,7,9,13,],10,14))

**Big O**

O(log n)

# Week 5

**Task 1**

##function to find the longest sub sequence in ascending order

def longestSubSequence(n):

longestSeqFound = []

currentSeq = []

for i in range(len(n)):

if i < len(n)-1 and n[i] < n[i+1]: #if the index which i is currently on is smaller than the length of the sequence

currentSeq.append(n[i]) #and the number is smaller than next one it will append that number to the list

#print(currentSeq)

else: #appends the number to the list if the conditions of the if statement aren't met

currentSeq.append(n[i]) #last number of the current sequence

#print(currentSeq)

if len(currentSeq) > len(longestSeqFound): #compares the length of sequences to check which one is bigger

longestSeqFound = currentSeq

#print(longestSeqFound)

currentSeq = [] #clears the list so the new sequence can be placed here and compared to the longest one

return(longestSeqFound)

print(longestSubSequence([1,2,3,4,1,6,1,7,8,1,2,3,4,5,6,7]))

**Task 2**

###Doesn't work as it should

##def remove (self,n):

## if n.prev != None:

## n.prev.next = n.next

## else:

## self.head = n.next

##

## if n.next != None:

## n.next.prev = n.prev

##

## else:

## self.tail = n.prev

# Week 6

**Task 1**

def inOrder(tree):

stack = []

currentNode = tree #current node of the tree

while True: #the loop will always run until it has gone through the entire tree

while currentNode != None: #until the node is not equal to node it will append the value of the current to the stack

stack.append(currentNode)

currentNode = currentNode.left #moves to the left side of the tree

if len(stack) == 0: #when the length of the stack is 0 it has traversed the entire tree and breaks out of the loop

break

currentNode = stack.pop() #pops the value that is at the top of the stack, also backtracks through the stack

print (currentNode.value)

currentNode = currentNode.right #sets the node to go to the right