DATA STRUCTURES & ALGORITHMS

UNIT ASSIGNMENT REPORT

Russell Sammut-Bonnici
ICS1018
March 20, 2018

Contents

Task 1 – Matching Products

In this task, a program was required to find all two distinct pairs of integers that have the same product. The range of integers in the given list was set from 1 to 1024.

How the problem was solved

The problem was solved by firstly creating a class called Pair. This class would be able to initialize objects with two distinct integers inserted through the parameters, and from which it calculates the product.

The method findProductMatches(rangeList) was then created. Its main function is to read a range of numbers (in this program's case 1-1024) and return all the matches by product of each distinct pair from the list, efficiently. It made use of calling a sorting algorithm to sort its pairs by products.

How the program was tested

The program was firstly tested using small ranges like 1-10, seeing that the output functioned properly it was then increased to 1-25 (shown in the below output listing) and then to 1-1024 as the question requested. Not to much surprise, the time taken for completion increased when the range was increased but thanks to implementing quick-sort, most of the time increase was due to printing more pair matches.

Optimizations made

Not only did making use of objects allow the pairs to be sorted by a specific characteristic (in this case products), but it also allowed organisation and ease of access when it came to printing the pair.

The program made use of quick sort to efficiently organise the large list of numbers from 1-1024. It was called with the method name "sortByProduct(pairList)." This quick sort was made to work with a Pair object list and it would rearrange the pairs with regards to their products in ascending order.

On displaying the matches it would print all the matches on the same line to save on time during printing. A for loop to go through every index in the pairList array was made, and a nested while loop was used to traverse through the products and display every match of the index after to the product at the current index on the same line.

Source Code (task1.py)

```
#Name: Russell Sammut-Bonnici
#ID: 0426299(M)
#Task: 1

import random #used for choosing random pivot

#class for storing pair's details
class Pair:

    #initialises product pair
    def __init__(self,a,b):
        self.a = a
        self.b = b
        self.product = a*b

#recursive method for sorting pairs by products
def sortByProduct(objList):

    #base case for when segment's length is less than or equal to one
    if(len(objList)<=1):
        return objList

    smaller=[] #initialized for storing the smaller segment of the list
    equivalent=[] #initialized for storing the element at the pivot
    greater=[] #initialized for storing the greater segment of the list

    #randomly chosen pivot selected from list of pairs
    pivot = objList[random.randint(0,len(objList)-1)]

    #for loop to go through the list of pairs
    for x in objList:

        #when x is less, append to smaller segment
        if(x.product < pivot.product):
            smaller.append(x)

        #when x is at the pivot, store element into equivalent
        elif(x.product==pivot.product):
            equivalent.append(x)

        #when x is greater, append to greater segment
        elif(x.product > pivot.product):
            greater.append(x)

        else:
            print("An unknown error has occurred during sorting by Product")

    #recursively calls method to work on the smaller and greater segment, then returns
on backtracking
    return sortByProduct(smaller) + equivalent + sortByProduct(greater)

#method for finding product matches
def findProductMatches(rangeList):

    pairList = [] #initialized for storing products pairs and their components

    # outer loop for a equivalent
    for x in rangeList:

        # inner loop for b equivalent
        for y in rangeList:
```

```
            if(y>x): #condition to avoid repeated a*b combinations
                pairList.append(Pair(x,y))

    pairList = sortByProduct(pairList) #sorts by product using quick sort

    i = 0 #initialized index to 0

    #goes through every index in the pairList array
    while(i<len(pairList)):

        #Accesses this during the last iteration when the last index has no matching
products. Avoids OutOfIndex exception, iterates to exit loop
        if(i==len(pairList)-1):
            i+=1

        #If pair at index has a unique product, skips cause no matches
        elif(pairList[i].product != pairList[i+1].product):
            i+=1

        #If current pair is seen to have at least one match, access while loop
        elif(pairList[i].product == pairList[i+1].product):

            print("The pair [%d*%d = %d] matches with"%(pairList[i].a, pairList[i].b,
pairList[i].product)),

            #traverses through products, displaying the matches until there are no more
(stops when at the end)
            while(i!=len(pairList)-1 and pairList[i].product == pairList[i+1].product):
                print("[%d*%d = %d] and"%(pairList[i+1].a, pairList[i+1].b,
pairList[i+1].product)),
                i+=1

            print(",") #prints comma indicating end of line

        #else statement used for unconsidered scenarios
        else:
            print("Error: something wrong has occurred during printing.")

    return 0

print("Matching products from 1 to 1024 are;")
findProductMatches(range(1, 1025)) #or (range(1,26))
```

Console Listing

```
#The console listing for matches from 1-1024 were too large to include in this report
so #instead matches from 1-25 are shown below

Matching products from 1 to 1024 are;
The pair [1*6 = 6] matches with [2*3 = 6] and ,
The pair [1*8 = 8] matches with [2*4 = 8] and ,
The pair [1*10 = 10] matches with [2*5 = 10] and ,
The pair [1*12 = 12] matches with [2*6 = 12] and [3*4 = 12] and ,
The pair [1*14 = 14] matches with [2*7 = 14] and ,
The pair [1*15 = 15] matches with [3*5 = 15] and ,
The pair [1*16 = 16] matches with [2*8 = 16] and ,
The pair [1*18 = 18] matches with [2*9 = 18] and [3*6 = 18] and ,
The pair [1*20 = 20] matches with [2*10 = 20] and [4*5 = 20] and ,
The pair [1*21 = 21] matches with [3*7 = 21] and ,
The pair [1*22 = 22] matches with [2*11 = 22] and ,
```

```
The pair [1*24 = 24] matches with [2*12 = 24] and [3*8 = 24] and [4*6 = 24] and ,
The pair [2*14 = 28] matches with [4*7 = 28] and ,
The pair [2*15 = 30] matches with [3*10 = 30] and [5*6 = 30] and ,
The pair [2*16 = 32] matches with [4*8 = 32] and ,
The pair [2*18 = 36] matches with [3*12 = 36] and [4*9 = 36] and ,
The pair [2*20 = 40] matches with [4*10 = 40] and [5*8 = 40] and ,
The pair [2*21 = 42] matches with [3*14 = 42] and [6*7 = 42] and ,
The pair [2*22 = 44] matches with [4*11 = 44] and ,
The pair [3*15 = 45] matches with [5*9 = 45] and ,
The pair [2*24 = 48] matches with [3*16 = 48] and [4*12 = 48] and [6*8 = 48] and ,
The pair [2*25 = 50] matches with [5*10 = 50] and ,
The pair [3*18 = 54] matches with [6*9 = 54] and ,
The pair [4*14 = 56] matches with [7*8 = 56] and ,
The pair [3*20 = 60] matches with [4*15 = 60] and [5*12 = 60] and [6*10 = 60] and ,
The pair [3*21 = 63] matches with [7*9 = 63] and ,
The pair [3*22 = 66] matches with [6*11 = 66] and ,
The pair [5*14 = 70] matches with [7*10 = 70] and ,
The pair [3*24 = 72] matches with [4*18 = 72] and [6*12 = 72] and [8*9 = 72] and ,
The pair [3*25 = 75] matches with [5*15 = 75] and ,
The pair [4*20 = 80] matches with [5*16 = 80] and [8*10 = 80] and ,
The pair [4*21 = 84] matches with [6*14 = 84] and [7*12 = 84] and ,
The pair [4*22 = 88] matches with [8*11 = 88] and ,
The pair [5*18 = 90] matches with [6*15 = 90] and [9*10 = 90] and ,
The pair [4*24 = 96] matches with [6*16 = 96] and [8*12 = 96] and ,
The pair [4*25 = 100] matches with [5*20 = 100] and ,
The pair [5*21 = 105] matches with [7*15 = 105] and ,
The pair [6*18 = 108] matches with [9*12 = 108] and ,
The pair [5*22 = 110] matches with [10*11 = 110] and ,
The pair [7*16 = 112] matches with [8*14 = 112] and ,
The pair [5*24 = 120] matches with [6*20 = 120] and [8*15 = 120] and [10*12 = 120] and
,
The pair [6*21 = 126] matches with [7*18 = 126] and [9*14 = 126] and ,
The pair [6*22 = 132] matches with [11*12 = 132] and ,
The pair [7*20 = 140] matches with [10*14 = 140] and ,
The pair [6*24 = 144] matches with [8*18 = 144] and [9*16 = 144] and ,
The pair [6*25 = 150] matches with [10*15 = 150] and ,
The pair [7*22 = 154] matches with [11*14 = 154] and ,
The pair [8*20 = 160] matches with [10*16 = 160] and ,
The pair [7*24 = 168] matches with [8*21 = 168] and [12*14 = 168] and ,
The pair [8*22 = 176] matches with [11*16 = 176] and ,
The pair [9*20 = 180] matches with [10*18 = 180] and [12*15 = 180] and ,
The pair [8*24 = 192] matches with [12*16 = 192] and ,
The pair [9*22 = 198] matches with [11*18 = 198] and ,
The pair [8*25 = 200] matches with [10*20 = 200] and ,
The pair [10*21 = 210] matches with [14*15 = 210] and ,
The pair [9*24 = 216] matches with [12*18 = 216] and ,
The pair [10*22 = 220] matches with [11*20 = 220] and ,
The pair [10*24 = 240] matches with [12*20 = 240] and [15*16 = 240] and ,
The pair [12*21 = 252] matches with [14*18 = 252] and ,
The pair [11*24 = 264] matches with [12*22 = 264] and ,
The pair [12*24 = 288] matches with [16*18 = 288] and ,
The pair [12*25 = 300] matches with [15*20 = 300] and ,
The pair [14*24 = 336] matches with [16*21 = 336] and ,
The pair [15*24 = 360] matches with [18*20 = 360] and ,

Process finished with exit code 0
```

Task 2 – RPN Calculator

In this task, a program was required to make use of an Abstract Data Type Stack to evaluate arithmetic expressions in Reverse Polish Notation format. The allowed operators were “+, -, x , /”

How the problem was solved

The requested operators were defined by having each operator character refer to their corresponding operator function in an array.

The class Stack was then made to provide an implementation of an ADT Stack to be used by the RPN Calculator. It allowed initialization, pushing, popping, getting stack size and printing the stack’s contents.

The method rpnCalc(String) would then have an RPN expression passed through it and it would parse the String character by character until the whole expression is evaluated through iterating, referencing operator functions and using stack operations. Note that spaces were used to separate tokens.

How the program was tested

The program was tested using various different RPN expressions. Wrong inputs such as “hello” and “2 2 + -” were used to test the error case. Validation aside, a fairly complex expression settled on for functionality testing was "3 5 x 7 9 / 3 8 + x + 9 +". Its console listings are shown in the next page.

Features

Expression validation was implemented by having error cases where less than two parameters are attempted to be operated on and where invalid characters are detected. Each would return-1 when an error case is resolved as true.

The variable num was type-casted as float in order to work with accurate decimal values when involved with the ‘/’ operator or floats entered as a token. The character ‘.’ Was also included as a token for numbers to provide support for evaluating floats.

Source Code (task2.py)

```
#Name: Russell Sammut-Bonnici
#ID: 0426299(M)
#Task: 2

import operator #for operator functionality

#defines operator characters and corresponding functions
ops = { '+': operator.add,
        '-': operator.sub,
        'x': operator.mul,
        '/': operator.div
}

#class used to implement ADT Stack
class Stack:

    #initializes stack
    def __init__(self):
        self.items = []

    #pushes to the top of the stack
    def push(self, item):
        self.items.append(item)

    #pops top element of the stack
    def pop(self):
        return self.items.pop()

    #gets size of the stack
    def size(self):
        return len(self.items)

    #used for printing the stack
    def __str__(self):
        return str(self.items)

#method used for RPN Calculator
def rpnCalc(String):

    s = Stack() #instance of Stack
    temp = "" #initializes String to temporarily store numbers
    num = 0 #initializes number to store current number

    #scanning each token in the string and evaluating with stack
    for x in String:

        #when token is a number or decimal point
        if x in ['0','1','2','3','5','6','7','8','9','.']:
           temp+=x #concatenate to temp

        #when token is a space
        elif x==" ":

            #if the previous argument was a number, push to stack
            if temp!="":
                num = float(temp)  #convert to float
                s.push(num)  #push into stack
                temp = ""  #clear temp
                print s #print stack

            #note that if operator was before, nothing is performed
```

```
        #when token is an operator
        elif x in ops:

            #when stack has less than two elements
            if s.size()<2:
                print("Error: At least two parameters required before the operator. -1
returned")
                return -1

            b = s.pop()
            a = s.pop()
            op = ops[x]

            s.push(op(a,b))

            print s #print stack

        else:fo
            print("Error: Invalid character \' %s \' detected. -1 returned"%x)
            return -1

    return s.pop() #returns calculated number after stack evaluation

expression = "3 5 x 7 9 / 3 8 + x + 9 +"
print("Evaluating \"%s\" using RPN: "%(expression))
print("Answer = %.1f"%rpnCalc(expression))
```

Console Listing

```
Evaluating "3 5 x 7 9 / 3 8 + x + 9 +" using RPN:

[3.0]

[3.0, 5.0]

[15.0]

[15.0, 7.0]

[15.0, 7.0, 9.0]

[15.0, 0.7777777777777778]

[15.0, 0.7777777777777778, 3.0]

[15.0, 0.7777777777777778, 3.0, 8.0]

[15.0, 0.7777777777777778, 11.0]

[15.0, 8.555555555555555]

[23.555555555555557]

[23.555555555555557, 9.0]

[32.55555555555556]

Answer = 32.6
```

```
Process finished with exit code 0
```

Console Listings for Validity Testing

```
#when expression=”hello”
Evaluating "hello" using RPN:
Error: Invalid character ' h ' detected. -1 returned
Answer = -1.0

Process finished with exit code 0

#when expression=”2 2 + =”
Evaluating "2 2 + -" using RPN:
[2.0]
[2.0, 2.0]
[4.0]
Error: At least two parameters required before the operator. -1 returned
Answer = -1.0

Process finished with exit code 0
```

Task 3 – Sieve of Eratosthenes and Prime Check

In this task, a program was required to check whether a number is prime or not. An implementation of the Sieve of Eratosthenes was also required.

How the problem was solved

The method “isPrime(n)” accepts an integer n as a parameter and checks whether it is prime by returning True or False. It does this by checking if n has any factors before itself. The range on checking is set from 2-n as 2 is known to be a prime number and n does not need to be included as every number is a factor of itself.

The method “findPrimes1(n)” finds all prime numbers less than or equal to n by simply calling isPrime(x) for x in range(2, n+1) and appending to a final list called “primes” when isPrime(x) returns True.

The method “findPrimes2(n)” then makes use of the Sieve of Eratosthenes algorithm to achieve the same thing, but proves itself to be much more efficient with respect to time complexity. Between both functions their execution time was compared by using the time.time() function from the time package.

How the program was tested

The isPrime method was tested using various numbers that were prime and weren’t and has proved to return the appropriate True or False value in each opposite scenario. The number 101 was done as a final test as it is a moderately high prime number.

The findPrimes1 and findPrimes2 methods on the other hand were tested with the number set as 10, 100, 1000, 10,000 then 100,000. In comparison the first few test cases didn’t have such high difference but when n2 (the passed number) was set as 100,000 a difference of 63.766 seconds was evident as the execution time for findPrimes1 was 63.798 seconds and for findPrimes2 it was 0.032 seconds. This goes to show how efficient the Sieve of Eratosthenes algorithm is.

Optimizations made

In findPrimes2(n), in initialization the list sieve made use of do not care values. This is because the variable instance at each index in the for loop isn’t important as they are all initially to be set to True, and do not require any further interaction other than simple initialization.

Source Code (task3.py)

```
#Name: Russell Sammut-Bonnici
#ID: 0426299(M)
#Task: 3

import time #used for execution time comparison

#checks if number is prime
def isPrime(n):

    '''
    The range for the loop is set from 2 to n, as we are concerned with finding factors
    that aren't 1 and n itself. If no factors that aren't 1 and n are found then by
definition
    it is a prime number.
    '''

    #if 2 then prime number
    if(n==2):
        return True
    else:
    #if n is not 2 then test if n has a factor between 2-n
        for x in range(2,n):

            r = n%x #calculating remainder for current test

            if(r==0): #if a factor is found, n is not prime
                return False

        # if a factor isn't found in the loop, n is prime
        return True

#finds all prime numbers less than or equal to n using isPrime()
def findPrimes1(n):

    start_time=time.time() #gets time at start

    primes = [] #initialized to later store prime numbers

    #for loop from range 2-n (including n)
    for x in range(2,n+1):

        # appends prime numbers to list "primes" when isPrime(x) is True
        if(isPrime(x)==True):
            primes.append(x)

    end_time = time.time() #gets time at end
    print("Total time: %0.5fs"%(end_time-start_time)) #prints executuion time

    return primes



#finds all prime numbers less than or equal to n using Sieve of Eratosthenes Algorithm
def findPrimes2(n):

    start_time = time.time() #gets time at start

    primes = [] #initializes list of prime numbers as empty

    #initializes Boolean list from 2-n (including n)
```

```
    sieve = [True for _ in range(n+1)]

    #starts by setting 0 and 1 to False as they are not prime numbers
    sieve[0:1] = [False, False]

    #for loop from 2-n
    for p in range(2, n + 1):

        #if True, access nested loop
        if sieve[p]:

            #set multiples of p to false, starting from 2*p and incrementing by p each
iteration
            for i in range(2 * p, n + 1, p):
                sieve[i] = False

    #for loop from 2-n
    for i in range(2, n + 1):

        #if True, after the procedure above, then prime, so add to primes list
        if sieve[i]:
            primes.append(i)

    end_time = time.time() #gets time at end
    print("Total time: %0.5f" % (end_time - start_time)) #prints executuion time

    return primes

n1=101
print("Is %d a prime number?: %s"%(n1,isPrime(n1)))

print("-------------------------------------")

n2=10000

primes1 = findPrimes1(n2)
print("List of prime numbers up to %d using isPrime() are: "%n2)
print primes1

print("-------------------------------------")

primes2 = findPrimes2(n2)
print("List of prime numbers up to %d using Sieve of Eratosthenes are: "%n2)
print primes2
```

Console Listing

```
Is 101 a prime number?: True

-------------------------------------

Total time: 0.70700

List of prime numbers up to 10000 using isPrime() are:

[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79,
83, 89, 97, 101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
179, 181, 191, 193, 197, 199, 211, 223, 227, 229, 233, 239, 241, 251, 257, 263, 269,
271, 277, 281, 283, 293, 307, 311, 313, 317, 331, 337, 347, 349, 353, 359, 367, 373,
379, 383, 389, 397, 401, 409, 419, 421, 431, 433, 439, 443, 449, 457, 461, 463, 467,
479, 487, 491, 499, 503, 509, 521, 523, 541, 547, 557, 563, 569, 571, 577, 587, 593,
```

```
599, 601, 607, 613, 617, 619, 631, 641, 643, 647, 653, 659, 661, 673, 677, 683, 691,
701, 709, 719, 727, 733, 739, 743, 751, 757, 761, 769, 773, 787, 797, 809, 811, 821,
823, 827, 829, 839, 853, 857, 859, 863, 877, 881, 883, 887, 907, 911, 919, 929, 937,
941, 947, 953, 967, 971, 977, 983, 991, 997, 1009, 1013, 1019, 1021, 1031, 1033, 1039,
1049, 1051, 1061, 1063, 1069, 1087, 1091, 1093, 1097, 1103, 1109, 1117, 1123, 1129,
1151, 1153, 1163, 1171, 1181, 1187, 1193, 1201, 1213, 1217, 1223, 1229, 1231, 1237,
1249, 1259, 1277, 1279, 1283, 1289, 1291, 1297, 1301, 1303, 1307, 1319, 1321, 1327,
1361, 1367, 1373, 1381, 1399, 1409, 1423, 1427, 1429, 1433, 1439, 1447, 1451, 1453,
1459, 1471, 1481, 1483, 1487, 1489, 1493, 1499, 1511, 1523, 1531, 1543, 1549, 1553,
1559, 1567, 1571, 1579, 1583, 1597, 1601, 1607, 1609, 1613, 1619, 1621, 1627, 1637,
1657, 1663, 1667, 1669, 1693, 1697, 1699, 1709, 1721, 1723, 1733, 1741, 1747, 1753,
1759, 1777, 1783, 1787, 1789, 1801, 1811, 1823, 1831, 1847, 1861, 1867, 1871, 1873,
1877, 1879, 1889, 1901, 1907, 1913, 1931, 1933, 1949, 1951, 1973, 1979, 1987, 1993,
1997, 1999, 2003, 2011, 2017, 2027, 2029, 2039, 2053, 2063, 2069, 2081, 2083, 2087,
2089, 2099, 2111, 2113, 2129, 2131, 2137, 2141, 2143, 2153, 2161, 2179, 2203, 2207,
2213, 2221, 2237, 2239, 2243, 2251, 2267, 2269, 2273, 2281, 2287, 2293, 2297, 2309,
2311, 2333, 2339, 2341, 2347, 2351, 2357, 2371, 2377, 2381, 2383, 2389, 2393, 2399,
2411, 2417, 2423, 2437, 2441, 2447, 2459, 2467, 2473, 2477, 2503, 2521, 2531, 2539,
2543, 2549, 2551, 2557, 2579, 2591, 2593, 2609, 2617, 2621, 2633, 2647, 2657, 2659,
2663, 2671, 2677, 2683, 2687, 2689, 2693, 2699, 2707, 2711, 2713, 2719, 2729, 2731,
2741, 2749, 2753, 2767, 2777, 2789, 2791, 2797, 2801, 2803, 2819, 2833, 2837, 2843,
2851, 2857, 2861, 2879, 2887, 2897, 2903, 2909, 2917, 2927, 2939, 2953, 2957, 2963,
2969, 2971, 2999, 3001, 3011, 3019, 3023, 3037, 3041, 3049, 3061, 3067, 3079, 3083,
3089, 3109, 3119, 3121, 3137, 3163, 3167, 3169, 3181, 3187, 3191, 3203, 3209, 3217,
3221, 3229, 3251, 3253, 3257, 3259, 3271, 3299, 3301, 3307, 3313, 3319, 3323, 3329,
3331, 3343, 3347, 3359, 3361, 3371, 3373, 3389, 3391, 3407, 3413, 3433, 3449, 3457,
3461, 3463, 3467, 3469, 3491, 3499, 3511, 3517, 3527, 3529, 3533, 3539, 3541, 3547,
3557, 3559, 3571, 3581, 3583, 3593, 3607, 3613, 3617, 3623, 3631, 3637, 3643, 3659,
3671, 3673, 3677, 3691, 3697, 3701, 3709, 3719, 3727, 3733, 3739, 3761, 3767, 3769,
3779, 3793, 3797, 3803, 3821, 3823, 3833, 3847, 3851, 3853, 3863, 3877, 3881, 3889,
3907, 3911, 3917, 3919, 3923, 3929, 3931, 3943, 3947, 3967, 3989, 4001, 4003, 4007,
4013, 4019, 4021, 4027, 4049, 4051, 4057, 4073, 4079, 4091, 4093, 4099, 4111, 4127,
4129, 4133, 4139, 4153, 4157, 4159, 4177, 4201, 4211, 4217, 4219, 4229, 4231, 4241,
4243, 4253, 4259, 4261, 4271, 4273, 4283, 4289, 4297, 4327, 4337, 4339, 4349, 4357,
4363, 4373, 4391, 4397, 4409, 4421, 4423, 4441, 4447, 4451, 4457, 4463, 4481, 4483,
4493, 4507, 4513, 4517, 4519, 4523, 4547, 4549, 4561, 4567, 4583, 4591, 4597, 4603,
4621, 4637, 4639, 4643, 4649, 4651, 4657, 4663, 4673, 4679, 4691, 4703, 4721, 4723,
4729, 4733, 4751, 4759, 4783, 4787, 4789, 4793, 4799, 4801, 4813, 4817, 4831, 4861,
4871, 4877, 4889, 4903, 4909, 4919, 4931, 4933, 4937, 4943, 4951, 4957, 4967, 4969,
4973, 4987, 4993, 4999, 5003, 5009, 5011, 5021, 5023, 5039, 5051, 5059, 5077, 5081,
5087, 5099, 5101, 5107, 5113, 5119, 5147, 5153, 5167, 5171, 5179, 5189, 5197, 5209,
5227, 5231, 5233, 5237, 5261, 5273, 5279, 5281, 5297, 5303, 5309, 5323, 5333, 5347,
5351, 5381, 5387, 5393, 5399, 5407, 5413, 5417, 5419, 5431, 5437, 5441, 5443, 5449,
5471, 5477, 5479, 5483, 5501, 5503, 5507, 5519, 5521, 5527, 5531, 5557, 5563, 5569,
5573, 5581, 5591, 5623, 5639, 5641, 5647, 5651, 5653, 5657, 5659, 5669, 5683, 5689,
5693, 5701, 5711, 5717, 5737, 5741, 5743, 5749, 5779, 5783, 5791, 5801, 5807, 5813,
5821, 5827, 5839, 5843, 5849, 5851, 5857, 5861, 5867, 5869, 5879, 5881, 5897, 5903,
5923, 5927, 5939, 5953, 5981, 5987, 6007, 6011, 6029, 6037, 6043, 6047, 6053, 6067,
6073, 6079, 6089, 6091, 6101, 6113, 6121, 6131, 6133, 6143, 6151, 6163, 6173, 6197,
6199, 6203, 6211, 6217, 6221, 6229, 6247, 6257, 6263, 6269, 6271, 6277, 6287, 6299,
6301, 6311, 6317, 6323, 6329, 6337, 6343, 6353, 6359, 6361, 6367, 6373, 6379, 6389,
6397, 6421, 6427, 6449, 6451, 6469, 6473, 6481, 6491, 6521, 6529, 6547, 6551, 6553,
6563, 6569, 6571, 6577, 6581, 6599, 6607, 6619, 6637, 6653, 6659, 6661, 6673, 6679,
6689, 6691, 6701, 6703, 6709, 6719, 6733, 6737, 6761, 6763, 6779, 6781, 6791, 6793,
6803, 6823, 6827, 6829, 6833, 6841, 6857, 6863, 6869, 6871, 6883, 6899, 6907, 6911,
6917, 6947, 6949, 6959, 6961, 6967, 6971, 6977, 6983, 6991, 6997, 7001, 7013, 7019,
7027, 7039, 7043, 7057, 7069, 7079, 7103, 7109, 7121, 7127, 7129, 7151, 7159, 7177,
7187, 7193, 7207, 7211, 7213, 7219, 7229, 7237, 7243, 7247, 7253, 7283, 7297, 7307,
7309, 7321, 7331, 7333, 7349, 7351, 7369, 7393, 7411, 7417, 7433, 7451, 7457, 7459,
7477, 7481, 7487, 7489, 7499, 7507, 7517, 7523, 7529, 7537, 7541, 7547, 7549, 7559,
```

```
7561, 7573, 7577, 7583, 7589, 7591, 7603, 7607, 7621, 7639, 7643, 7649, 7669, 7673,
7681, 7687, 7691, 7699, 7703, 7717, 7723, 7727, 7741, 7753, 7757, 7759, 7789, 7793,
7817, 7823, 7829, 7841, 7853, 7867, 7873, 7877, 7879, 7883, 7901, 7907, 7919, 7927,
7933, 7937, 7949, 7951, 7963, 7993, 8009, 8011, 8017, 8039, 8053, 8059, 8069, 8081,
8087, 8089, 8093, 8101, 8111, 8117, 8123, 8147, 8161, 8167, 8171, 8179, 8191, 8209,
8219, 8221, 8231, 8233, 8237, 8243, 8263, 8269, 8273, 8287, 8291, 8293, 8297, 8311,
8317, 8329, 8353, 8363, 8369, 8377, 8387, 8389, 8419, 8423, 8429, 8431, 8443, 8447,
8461, 8467, 8501, 8513, 8521, 8527, 8537, 8539, 8543, 8563, 8573, 8581, 8597, 8599,
8609, 8623, 8627, 8629, 8641, 8647, 8663, 8669, 8677, 8681, 8689, 8693, 8699, 8707,
8713, 8719, 8731, 8737, 8741, 8747, 8753, 8761, 8779, 8783, 8803, 8807, 8819, 8821,
8831, 8837, 8839, 8849, 8861, 8863, 8867, 8887, 8893, 8923, 8929, 8933, 8941, 8951,
8963, 8969, 8971, 8999, 9001, 9007, 9011, 9013, 9029, 9041, 9043, 9049, 9059, 9067,
9091, 9103, 9109, 9127, 9133, 9137, 9151, 9157, 9161, 9173, 9181, 9187, 9199, 9203,
9209, 9221, 9227, 9239, 9241, 9257, 9277, 9281, 9283, 9293, 9311, 9319, 9323, 9337,
9341, 9343, 9349, 9371, 9377, 9391, 9397, 9403, 9413, 9419, 9421, 9431, 9433, 9437,
9439, 9461, 9463, 9467, 9473, 9479, 9491, 9497, 9511, 9521, 9533, 9539, 9547, 9551,
9587, 9601, 9613, 9619, 9623, 9629, 9631, 9643, 9649, 9661, 9677, 9679, 9689, 9697,
9719, 9721, 9733, 9739, 9743, 9749, 9767, 9769, 9781, 9787, 9791, 9803, 9811, 9817,
9829, 9833, 9839, 9851, 9857, 9859, 9871, 9883, 9887, 9901, 9907, 9923, 9929, 9931,
9941, 9949, 9967, 9973]

-------------------------------------

Total time: 0.00000

List of prime numbers up to 10000 using Sieve of Eratosthenes are:

[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79,
83, 89, 97, 101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
179, 181, 191, 193, 197, 199, 211, 223, 227, 229, 233, 239, 241, 251, 257, 263, 269,
271, 277, 281, 283, 293, 307, 311, 313, 317, 331, 337, 347, 349, 353, 359, 367, 373,
379, 383, 389, 397, 401, 409, 419, 421, 431, 433, 439, 443, 449, 457, 461, 463, 467,
479, 487, 491, 499, 503, 509, 521, 523, 541, 547, 557, 563, 569, 571, 577, 587, 593,
599, 601, 607, 613, 617, 619, 631, 641, 643, 647, 653, 659, 661, 673, 677, 683, 691,
701, 709, 719, 727, 733, 739, 743, 751, 757, 761, 769, 773, 787, 797, 809, 811, 821,
823, 827, 829, 839, 853, 857, 859, 863, 877, 881, 883, 887, 907, 911, 919, 929, 937,
941, 947, 953, 967, 971, 977, 983, 991, 997, 1009, 1013, 1019, 1021, 1031, 1033, 1039,
1049, 1051, 1061, 1063, 1069, 1087, 1091, 1093, 1097, 1103, 1109, 1117, 1123, 1129,
1151, 1153, 1163, 1171, 1181, 1187, 1193, 1201, 1213, 1217, 1223, 1229, 1231, 1237,
1249, 1259, 1277, 1279, 1283, 1289, 1291, 1297, 1301, 1303, 1307, 1319, 1321, 1327,
1361, 1367, 1373, 1381, 1399, 1409, 1423, 1427, 1429, 1433, 1439, 1447, 1451, 1453,
1459, 1471, 1481, 1483, 1487, 1489, 1493, 1499, 1511, 1523, 1531, 1543, 1549, 1553,
1559, 1567, 1571, 1579, 1583, 1597, 1601, 1607, 1609, 1613, 1619, 1621, 1627, 1637,
1657, 1663, 1667, 1669, 1693, 1697, 1699, 1709, 1721, 1723, 1733, 1741, 1747, 1753,
1759, 1777, 1783, 1787, 1789, 1801, 1811, 1823, 1831, 1847, 1861, 1867, 1871, 1873,
1877, 1879, 1889, 1901, 1907, 1913, 1931, 1933, 1949, 1951, 1973, 1979, 1987, 1993,
1997, 1999, 2003, 2011, 2017, 2027, 2029, 2039, 2053, 2063, 2069, 2081, 2083, 2087,
2089, 2099, 2111, 2113, 2129, 2131, 2137, 2141, 2143, 2153, 2161, 2179, 2203, 2207,
2213, 2221, 2237, 2239, 2243, 2251, 2267, 2269, 2273, 2281, 2287, 2293, 2297, 2309,
2311, 2333, 2339, 2341, 2347, 2351, 2357, 2371, 2377, 2381, 2383, 2389, 2393, 2399,
2411, 2417, 2423, 2437, 2441, 2447, 2459, 2467, 2473, 2477, 2503, 2521, 2531, 2539,
2543, 2549, 2551, 2557, 2579, 2591, 2593, 2609, 2617, 2621, 2633, 2647, 2657, 2659,
2663, 2671, 2677, 2683, 2687, 2689, 2693, 2699, 2707, 2711, 2713, 2719, 2729, 2731,
2741, 2749, 2753, 2767, 2777, 2789, 2791, 2797, 2801, 2803, 2819, 2833, 2837, 2843,
2851, 2857, 2861, 2879, 2887, 2897, 2903, 2909, 2917, 2927, 2939, 2953, 2957, 2963,
2969, 2971, 2999, 3001, 3011, 3019, 3023, 3037, 3041, 3049, 3061, 3067, 3079, 3083,
3089, 3109, 3119, 3121, 3137, 3163, 3167, 3169, 3181, 3187, 3191, 3203, 3209, 3217,
3221, 3229, 3251, 3253, 3257, 3259, 3271, 3299, 3301, 3307, 3313, 3319, 3323, 3329,
3331, 3343, 3347, 3359, 3361, 3371, 3373, 3389, 3391, 3407, 3413, 3433, 3449, 3457,
3461, 3463, 3467, 3469, 3491, 3499, 3511, 3517, 3527, 3529, 3533, 3539, 3541, 3547,
3557, 3559, 3571, 3581, 3583, 3593, 3607, 3613, 3617, 3623, 3631, 3637, 3643, 3659,
```

```
3671, 3673, 3677, 3691, 3697, 3701, 3709, 3719, 3727, 3733, 3739, 3761, 3767, 3769,
3779, 3793, 3797, 3803, 3821, 3823, 3833, 3847, 3851, 3853, 3863, 3877, 3881, 3889,
3907, 3911, 3917, 3919, 3923, 3929, 3931, 3943, 3947, 3967, 3989, 4001, 4003, 4007,
4013, 4019, 4021, 4027, 4049, 4051, 4057, 4073, 4079, 4091, 4093, 4099, 4111, 4127,
4129, 4133, 4139, 4153, 4157, 4159, 4177, 4201, 4211, 4217, 4219, 4229, 4231, 4241,
4243, 4253, 4259, 4261, 4271, 4273, 4283, 4289, 4297, 4327, 4337, 4339, 4349, 4357,
4363, 4373, 4391, 4397, 4409, 4421, 4423, 4441, 4447, 4451, 4457, 4463, 4481, 4483,
4493, 4507, 4513, 4517, 4519, 4523, 4547, 4549, 4561, 4567, 4583, 4591, 4597, 4603,
4621, 4637, 4639, 4643, 4649, 4651, 4657, 4663, 4673, 4679, 4691, 4703, 4721, 4723,
4729, 4733, 4751, 4759, 4783, 4787, 4789, 4793, 4799, 4801, 4813, 4817, 4831, 4861,
4871, 4877, 4889, 4903, 4909, 4919, 4931, 4933, 4937, 4943, 4951, 4957, 4967, 4969,
4973, 4987, 4993, 4999, 5003, 5009, 5011, 5021, 5023, 5039, 5051, 5059, 5077, 5081,
5087, 5099, 5101, 5107, 5113, 5119, 5147, 5153, 5167, 5171, 5179, 5189, 5197, 5209,
5227, 5231, 5233, 5237, 5261, 5273, 5279, 5281, 5297, 5303, 5309, 5323, 5333, 5347,
5351, 5381, 5387, 5393, 5399, 5407, 5413, 5417, 5419, 5431, 5437, 5441, 5443, 5449,
5471, 5477, 5479, 5483, 5501, 5503, 5507, 5519, 5521, 5527, 5531, 5557, 5563, 5569,
5573, 5581, 5591, 5623, 5639, 5641, 5647, 5651, 5653, 5657, 5659, 5669, 5683, 5689,
5693, 5701, 5711, 5717, 5737, 5741, 5743, 5749, 5779, 5783, 5791, 5801, 5807, 5813,
5821, 5827, 5839, 5843, 5849, 5851, 5857, 5861, 5867, 5869, 5879, 5881, 5897, 5903,
5923, 5927, 5939, 5953, 5981, 5987, 6007, 6011, 6029, 6037, 6043, 6047, 6053, 6067,
6073, 6079, 6089, 6091, 6101, 6113, 6121, 6131, 6133, 6143, 6151, 6163, 6173, 6197,
6199, 6203, 6211, 6217, 6221, 6229, 6247, 6257, 6263, 6269, 6271, 6277, 6287, 6299,
6301, 6311, 6317, 6323, 6329, 6337, 6343, 6353, 6359, 6361, 6367, 6373, 6379, 6389,
6397, 6421, 6427, 6449, 6451, 6469, 6473, 6481, 6491, 6521, 6529, 6547, 6551, 6553,
6563, 6569, 6571, 6577, 6581, 6599, 6607, 6619, 6637, 6653, 6659, 6661, 6673, 6679,
6689, 6691, 6701, 6703, 6709, 6719, 6733, 6737, 6761, 6763, 6779, 6781, 6791, 6793,
6803, 6823, 6827, 6829, 6833, 6841, 6857, 6863, 6869, 6871, 6883, 6899, 6907, 6911,
6917, 6947, 6949, 6959, 6961, 6967, 6971, 6977, 6983, 6991, 6997, 7001, 7013, 7019,
7027, 7039, 7043, 7057, 7069, 7079, 7103, 7109, 7121, 7127, 7129, 7151, 7159, 7177,
7187, 7193, 7207, 7211, 7213, 7219, 7229, 7237, 7243, 7247, 7253, 7283, 7297, 7307,
7309, 7321, 7331, 7333, 7349, 7351, 7369, 7393, 7411, 7417, 7433, 7451, 7457, 7459,
7477, 7481, 7487, 7489, 7499, 7507, 7517, 7523, 7529, 7537, 7541, 7547, 7549, 7559,
7561, 7573, 7577, 7583, 7589, 7591, 7603, 7607, 7621, 7639, 7643, 7649, 7669, 7673,
7681, 7687, 7691, 7699, 7703, 7717, 7723, 7727, 7741, 7753, 7757, 7759, 7789, 7793,
7817, 7823, 7829, 7841, 7853, 7867, 7873, 7877, 7879, 7883, 7901, 7907, 7919, 7927,
7933, 7937, 7949, 7951, 7963, 7993, 8009, 8011, 8017, 8039, 8053, 8059, 8069, 8081,
8087, 8089, 8093, 8101, 8111, 8117, 8123, 8147, 8161, 8167, 8171, 8179, 8191, 8209,
8219, 8221, 8231, 8233, 8237, 8243, 8263, 8269, 8273, 8287, 8291, 8293, 8297, 8311,
8317, 8329, 8353, 8363, 8369, 8377, 8387, 8389, 8419, 8423, 8429, 8431, 8443, 8447,
8461, 8467, 8501, 8513, 8521, 8527, 8537, 8539, 8543, 8563, 8573, 8581, 8597, 8599,
8609, 8623, 8627, 8629, 8641, 8647, 8663, 8669, 8677, 8681, 8689, 8693, 8699, 8707,
8713, 8719, 8731, 8737, 8741, 8747, 8753, 8761, 8779, 8783, 8803, 8807, 8819, 8821,
8831, 8837, 8839, 8849, 8861, 8863, 8867, 8887, 8893, 8923, 8929, 8933, 8941, 8951,
8963, 8969, 8971, 8999, 9001, 9007, 9011, 9013, 9029, 9041, 9043, 9049, 9059, 9067,
9091, 9103, 9109, 9127, 9133, 9137, 9151, 9157, 9161, 9173, 9181, 9187, 9199, 9203,
9209, 9221, 9227, 9239, 9241, 9257, 9277, 9281, 9283, 9293, 9311, 9319, 9323, 9337,
9341, 9343, 9349, 9371, 9377, 9391, 9397, 9403, 9413, 9419, 9421, 9431, 9433, 9437,
9439, 9461, 9463, 9467, 9473, 9479, 9491, 9497, 9511, 9521, 9533, 9539, 9547, 9551,
9587, 9601, 9613, 9619, 9623, 9629, 9631, 9643, 9649, 9661, 9677, 9679, 9689, 9697,
9719, 9721, 9733, 9739, 9743, 9749, 9767, 9769, 9781, 9787, 9791, 9803, 9811, 9817,
9829, 9833, 9839, 9851, 9857, 9859, 9871, 9883, 9887, 9901, 9907, 9923, 9929, 9931,
9941, 9949, 9967, 9973]


Process finished with exit code 0
```

Task 4 – Binary Search Tree

A program was required to input a sequence of integers one at a time and incrementally build a binary search tree as a result, balancing the search tree was not required.

How the problem was solved

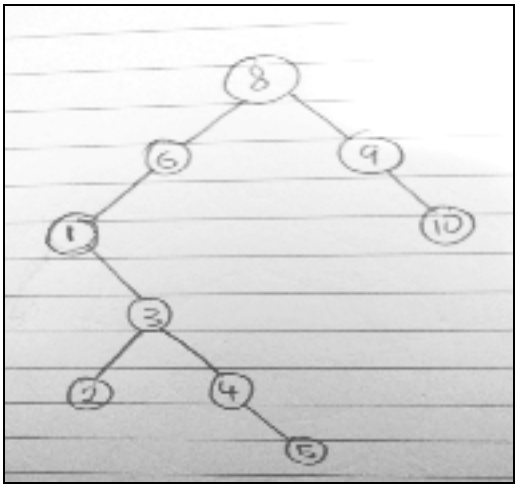
A class called “Node” was made to store the attributes of a single node. In this, the node would have it’s own value, and reference to the left child and the right child. Using an __init__ method the leftChild and rightChild was initialised as “None,” whereas the value is initialised as either a passed parameter or None when no number is passed.

Another class “BinarySearchTree” was then created. It would have a “root” attribute set as None on initialisation. Whenever a node is inserted it is either set as the root, or after it if a root already exists. If the root already exists it calls a recursive function to compare the inserted node to the already existing nodes in the tree and placing it accordingly.

An extra function in the BinarySearchTree class was made to print out the nodes from left to right, with their values and the values of their left child and right child. If statements were used to print different permutations of existing and non-existing children.

How the program was tested

The program was tested by firstly initialising a binary search tree and then inserting a stream of the integers [8, 6, 9, 1, 3, 2, 4, 5, 10] in to it. The expected tree after insertion was supposed to match the draft sketch shown below. The match to this structure can be found described in the output listings when the “printTree” method is called.



Features

Two error cases were made to prevent the insertions from breaking this simplistic implementation of a binary search tree. If a value that has already been inserted before is inserted again the program would stop the program and return a -1 after printing the error message "Error: Inserted value %d is already in the Binary Search Tree therefore it wasn't inserted."%(currNode.value). The other validation is for printing a tree when the tree has no nodes stored inside it. When this occurs it also returns -1 after printing "Error: Tree does not have any nodes to print out".

Source Code (task4.py)

```
#Name: Russell Sammut-Bonnici
#ID: 0426299(M)
#Task: 4

#class concerned with initialising node
class Node:

    #initializes node
    def __init__(self, value=None):
        self.value = value
        self.leftChild = None
        self.rightChild = None


#class concerned with Binary Search Tree functions
class BinarySearchTree:

    #initializes tree
    def __init__(self):
        self.root = None

    #inserts node into the tree, calls _insertNode after root is set
    def insertNode(self, value):

        if self.root == None:
            self.root = Node(value)
        else:
            self._insertNode(value, self.root)



    #recursive function called for inserting node after root
    def _insertNode(self, value, currNode):

        #when the value is less than its previous
        if value < currNode.value:

            #if left child doesn't exist add as left, else add under left
            if currNode.leftChild == None:
                currNode.leftChild = Node(value)
            else:
                self._insertNode(value, currNode.leftChild)

        #when the value is more than its previous
        elif value > currNode.value:
```

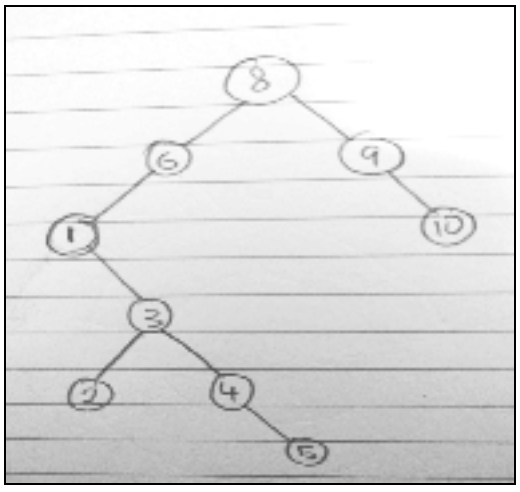
```
            #if right child doesn't exist add as right, else add under right
            if currNode.rightChild == None:
                currNode.rightChild = Node(value)
            else:
                self._insertNode(value, currNode.rightChild)

        #when the value is equal to its previous
        else:
            print "Error: Inserted value %d is already in the Binary Search Tree
therefore it wasn't inserted."%currNode.value
            return -1


    #prints the tree with all its contained nodes
    def printTree(self):

        #prints tree when root is not null with recursive function call
        if self.root!=None:
            self._printTree(self.root)
        else:
            print("Error: Tree does not have any nodes to print out")
            return -1

    #recursive function called for printing nodes left to right
    def _printTree(self,currNode):

        #base case is when the bottom of the tree is reached (when node==None)
        if currNode!=None:

            self._printTree(currNode.leftChild) #goes to the deepest node on the left

            # prints when left and right child don't exist
            if(currNode.leftChild==None and currNode.rightChild==None):
                print("[ %s ] ---> \tLEFT: [%s] \tRIGHT: [%s]" % (currNode.value,
currNode.leftChild, currNode.rightChild))

            # prints when left child only doesn't exist
            elif(currNode.leftChild==None):
                print("[ %s ] ---> \tLEFT: [%s] \tRIGHT: [ %s ]" % (currNode.value,
currNode.leftChild, currNode.rightChild.value))

            # prints when right child only doesn't exist
            elif(currNode.rightChild==None):
                print("[ %s ] ---> \tLEFT: [ %s ] \tRIGHT: [%s]" % (currNode.value,
currNode.leftChild.value, currNode.rightChild))

            # prints when left and right child do exist
            else:
                print("[ %s ] ---> \tLEFT: [ %s ] \tRIGHT: [ %s ]" % (currNode.value,
currNode.leftChild.value, currNode.rightChild.value))

            self._printTree(currNode.rightChild) #goes to the node right from the left,
then up a level

bst = BinarySearchTree() #initializing instance of class

#incrementally building up bst with a sequence of integers
bst.insertNode(8)
bst.insertNode(6)
bst.insertNode(9)
bst.insertNode(1)
bst.insertNode(3)
bst.insertNode(2)
bst.insertNode(4)
```

```
bst.insertNode(5)
bst.insertNode(10)

print("The Binary Search Tree's nodes from left to right are:")
bst.printTree()
```

Console Listing

```
The Binary Search Tree's nodes from left to right are:

[ 1 ] --->     LEFT: [None]   RIGHT: [ 3 ]

[ 2 ] --->     LEFT: [None]   RIGHT: [None]

[ 3 ] --->     LEFT: [ 2 ]    RIGHT: [ 4 ]

[ 4 ] --->     LEFT: [None]   RIGHT: [ 5 ]

[ 5 ] --->     LEFT: [None]   RIGHT: [None]

[ 6 ] --->     LEFT: [ 1 ]    RIGHT: [None]

[ 8 ] --->     LEFT: [ 6 ]    RIGHT: [ 9 ]

[ 9 ] --->     LEFT: [None]   RIGHT: [ 10 ]

[ 10 ] --->    LEFT: [None]   RIGHT: [None]


Process finished with exit code 0
```

```
#Successfully seen to describe the envisioned structure;
```



Task 5 – Newton-Raphson Method

In this task, a program was required to approximate the square root of a given number denoted as n. The approximation was said to be solved using an iterative method, in this program the recommended method was used, the Newton-Raphson Method.

How the problem was solved

The method "NewtonRhapson(n)" was defined, having n, the number to be square rooted, set as its parameter. Two variables x and x1 were initialized as 0 and n respectively. Using the Method's formula (shown below), when the approximation is seen to normalise after two consecutive trials, the program then returns the square root of n accurate to 9 decimal points.

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

How the program was tested

The program was tested using a large float 123456789.0 as n. Comparing the returned value 11111.111060556 to the return value of a scientific calculator's square root, the program was proved to approximate the square root value accurate to 9 decimal places.

Optimizations made

A while loop was used to continuously apply the evaluating formula from the Newton-Raphson Method, until the current approximation is seen to match the previous approximation to 9-decimal places.

This avoided having to set trials as a parameter for the approximation, this way the program is adaptable to the given number and doesn't continue approximating when the approximation keeps on getting evaluated to the same normalised number.

Source Code (task5.py)

```
#Name: Russell Sammut-Bonnici
#ID: 0426299(M)
#Task: 5

#finding the root using Newton-Rhapson method
def NewtonRhapson(n):

    x = 0 #setting initial x as 0
    x1 = n #initializing x1 as n

    #compares current approximation to previous for 9 decimal places
    while(round(x1,9)!=round(x,9)):

        x = x1 #sets current x to previous x1, for function calculation

        function = x**2 - n #f(x) of n
        function_deriv = 2*x #f'(x) of n

        x1 = x - (function/function_deriv) #Newton-Rhapson formula for finding the root

    return round(x,9) #returns answer accurate to 9 d.p.


n=123456789.0
print("Using the Newton-Rhapson Method, the square root of %f is
%.9f"%(n,NewtonRhapson(n)))
```

Console Listing

```
Using the Newton-Rhapson Method, the square root of 123456789.000000 is 11111.111060556

Process finished with exit code 0
```

Task 6 – Finding Repetitions

In this task, a program was required to find all integers repeated more than once from a list. Two different methods were used to achieve this. One I've dubbed as the "2-array method" and the other that uses a Quick-sort implementation.

How the problem was solved

The first method "findRepetition1(array)" uses 2 arrays, unique and duplicate. It starts off by storing the first element in the unique array. Using a for lop, it then goes through the numbers in the inputted array and compares them to the current numbers in the unique array. If no match is found, then the current number x is appended to unique. If a match is found, and is not already in duplicate, x is appended into duplicate. This is because we are not taking into account the amount of repetitions the number has but merely that it is repeated more than once. The duplicate array is then returned from the method.

Though this method provides efficient for small input lists, the larger the input list the more time exponentially increases. This is why an alternative solution using Quick-sort was then implemented, this is the second method.

The second method "findRepetition2(array)" firstly sorts the inputted array using a Quick-Sort implementation similar to the one used in Task 1. It then traverses through the array scanning for matches between one element and the next. When there is a match, the current element is added to the duplicate list and a while loop is used to iterate the index to the next number after the repetitions in the sorted list. After finishing this process, like findRepetition1() the duplicate list is then returned.

The use of execution time measurement was used to compare the two methods to evaluate which was ideal for which scenario. findRepetition1 proved more efficient for smaller lists and findRepetition2 proved more efficient for larger lists.

How the program was tested

The program was tested by passing the same array into both methods. The first array in testing was a small array [1,2,3,4,5,1,6,7,4,0,9,9]. But a larger array was required to challenge the program's efficiency so it was changed to a much larger array length. The larger array consisted of a list of numbers i from 1-99 added to another list ranging from 1-99. This way each number would have 1 repetition, and the program was tested to detect this.

Source Code (task6.py)

```
#Name: Russell Sammut-Bonnici
#ID: 0426299(M)
#Task: 6

import random #used for choosing random pivot
import time #used for execution time comparison

#recursive quick-sort for findRepetitions2(array)
def quickSort(numList):

    #base case for when segment's length is less than or equal to one
    if(len(numList)<=1):
        return numList

    smaller=[] #initialized for storing the smaller segment of the list
    equivalent=[] #initialized for storing the element at the pivot
    greater=[] #initialized for storing the greater segment of the list

    #randomly chosen pivot selected from list of pairs
    pivot = numList[random.randint(0,len(numList)-1)]

    #for loop to go through the list of pairs
    for x in numList:

        #when x is less, append to smaller segment
        if(x < pivot):
            smaller.append(x)

        #when x is at the pivot, store element into equivalent
        elif(x == pivot):
            equivalent.append(x)

        #when x is greater, append to greater segment
        elif(x > pivot):
            greater.append(x)

        else:
            print("An unknown error has occurred during sorting by number")

    #recursively calls method to work on the smaller and greater segment, then returns
on backtracking
    return quickSort(smaller) + equivalent + quickSort(greater)


#method solving problem using 2 arrays
def findRepetition1(array):

    start_time = time.time()  # gets time at start

    unique = [] #for storing unique elements
    duplicate = [] #for storing repeated elements

    #append first element as unique
    unique.append(array[0])

    #loop to scan through array
    for x in array:

        #when unique has contents
        if x in unique and not x in duplicate: #if match found, add to duplicate
provided duplicate is not already detected
            duplicate.append(x)
```

```
        if x not in unique: #if no match is found, add to unique
            unique.append(x)

    end_time = time.time()  # gets time at end
    print("Total time: %0.5fs" % (end_time - start_time))  # prints execution time

    #returns repeated numbers
    return duplicate


#method solving problem using an implementation of quick-sort
def findRepetition2(array):

    start_time = time.time()  # gets time at start

    array = quickSort(array)
    duplicate = []

    i = 0  # initialized index to 0

    # goes through every index in the array array
    while (i < len(array)):

        # Accesses this during the last iteration when the last index has no matching
numbers. Avoids OutOfIndex exception, iterates to exit loop
        if (i == len(array) - 1):
            i += 1

        # If number at index doesnt match next, iterate
        elif (array[i] != array[i + 1]):
            i += 1

        # If current number is seen to match next, access while loop
        elif (array[i] == array[i + 1]):

            duplicate.append(array[i]) #append as duplicate

            # traverses through repeated numbers, iterating over to prepare to scan the
next number
            while (i != len(array)-1 and array[i] == array[i + 1]):
                i += 1

        # else statement used for unconsidered scenarios
        else:
            print("Error: something wrong has occurred during traversing.")

    end_time = time.time()  # gets time at end
    print("Total time: %0.5fs" % (end_time - start_time))  # prints execution time

    #returns repeated numbers
    return duplicate


array=range(1,100)+range(1,100)

repetitions1 = findRepetition1(array)
print("Using the 2-array unique comparison method;")

repetitions2 = findRepetition2(array)
print("Using the Quick-sort implementation method;")
```

Console Listing

```
Total time: 0.00019s
Using the 2-array unique comparison method;
Duplicate numbers in the array [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38,
39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60,
61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82,
83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 1, 2, 3, 4, 5, 6,
7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28,
29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50,
51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72,
73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94,
95, 96, 97, 98, 99] are:
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24,
25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46,
47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68,
69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90,
91, 92, 93, 94, 95, 96, 97, 98, 99]

Total time: 0.00035s
Using the Quick-sort implementation method;
Duplicate numbers in the array [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38,
39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60,
61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82,
83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 1, 2, 3, 4, 5, 6,
7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28,
29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50,
51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72,
73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94,
95, 96, 97, 98, 99] are:
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24,
25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46,
47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68,
69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90,
91, 92, 93, 94, 95, 96, 97, 98, 99]

Process finished with exit code 0
```

Quick-Sort Efficiency

Two other scenarios were executed to prove that though Quick-sort is less efficient for a small array input, it is much more efficient for a large array input. In these console listings the output of the lists was removed as it contained too many numbers to include in this report.

```
#for range(1,1000)+range(1,1000):
Total time: 0.01761s
Using the 2-array unique comparison method;
Total time: 0.00412s
Using the Quick-sort implementation method;

Process finished with exit code 0

#for range(1,10000)+range(1,10000):
Total time: 1.63538s
Using the 2-array unique comparison method;
Total time: 0.04468s
Using the Quick-sort implementation method;

Process finished with exit code 0

#for range(1,100000)+range(1,100000):
Total time: 178.19355s
Using the 2-array unique comparison method;
Total time: 0.47951s
Using the Quick-sort implementation method;

Process finished with exit code 0
```

Task 7 – Finding Maximum Number Recursively

In this task, a program was required to find the largest number of a given list of integers using recursion.

How the problem was solved

The recursive method "_findMax(list)" accepts an integer list as its parameter. The length of the list is stored in a variable length (This is needed at every level, and is updated every time the length shortens).

A base case is used to check if the list is 1 in length, when it is it returns the only element in the list at index 0.

To get to this base case, an algorithm is made were the last two elements are always compared. The smallest element of the two gets deleted from the list and the length of the list shortens. It then calls itself recursively until its at the base case, where the remaining element is the largest from the list. Therefore the maximum is then returned.

How the program was tested

The program was tested using a simple list of [1,2,3,4,5,60,6,7,80,9,10]. From the list the maximum, 80, was returned (as can be seen in the output listing). The list was then changed to be equal to range(1,1000) to test the efficiency of the program, and it returned the maximum value appropriately. Testing even further, at range(1,10000) an unexpected RuntimeError error was seen to be returned, "maximum recursion depth exceeded". This is due to a precaution taken by Python in order to avoid a stack overflow.

Source Code (task7.py)

```
#Name: Russell Sammut-Bonnici
#ID: 0426299(M)
#Task: 7

#recursive method to finding the max number of the list
def _findMax(list):

    length = len(list) #getting current length of list

    #base case to check if list is 1 in length
    if length==1:
        return list[0]

    #compares last two elements
```

```
    if list[length - 1] > list[length - 2]:
        del list[length - 2] #delete element before the last, shift last to the left
    else:
        del list[length - 1] #delete element at the last position

    return _findMax(list) #calls method recursively

list = [1,2,3,4,5,60,6,7,80,9,10]
print("The max number in %s is:"%(str(list)))
print(_findMax(list))
```

Console Listing

```
The max number in [1, 2, 3, 4, 5, 60, 6, 7, 80, 9, 10] is:
80

Process finished with exit code 0
```

Task 8 – Cosine or Sine Series Expansion

In this task, a program was required to compute cosine or sine by taking the first n terms of the appropriate series expansion. In this program both cosine and sine series expansion were implemented.

How the problem was solved

Firstly by analysing MacLaurin's theorem for sin(x) and cos(x) (shown below), we see that the summation can be worked out using a for loop and an initialized variable sum, that increments to itself every iteration. To work this out a method to work out the factorial of a number is required.

$$\begin{aligned}\sin x &= x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots \\ &= \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n+1}}{(2n+1)!}, \\ \cos x &= 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots \\ &= \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n}}{(2n)!}.\end{aligned}$$

The recursive method "_factorial(x)" returns the factorial of number n by multiplying n by the factorial of the number below it. This involves a recursive call which keeps on occurring until the factorial of 0 is reached, where 1 is returned as a fact. The program then backtracks until it returns the correct value of n!.

In the methods "sin(x,n)" and "cos(x,n)" x is taken as the value in radians to be evaluated and t is taken as the amount of trials for the return value to be approximated to.

Both firstly include error checks to see if x is outside of the range from -2 π to 2 π , if it is it uses the modulus operator and sets x as x modulus 2 π . This is because a period outside the range would be accessed in the sin or cos wave. Making use of modulation allows the sum to evaluate accurately from the beginning of this new period.

Both make use of their respective MacLaurin's Theorem and evaluate the sum for a number of iterations specified by the number of trials t. Both the values are displayed to the user accuraet to 9 decimal places.

How the program was tested

The program was tested using small numbers as a (which is passed as x) like -1, -2, 2, 2.5, 3 but inaccurate return values were seen to display when a number like 50 or -50 was set as a. This was fixed by including if conditions to check if x is outside of the -2π - 2π range or not, and if so it modulates it by 2π to set the value from the starting point of its period 2π.

Source Code (task8.py)

```
#Name: Russell Sammut-Bonnici
#ID: 0426299(M)
#Task: 8

import math #for pi

#recursive function for finding the factorial of a number x
def _factorial(x):

    #base case, factorial of 0 is 1
    if(x == 0):
        return 1.0

    # calls recursively until base case
    return x * _factorial(x-1)


#function for calculating sin(x) through sum of first t terms
def sin(x,t):

    sum=0.0 #initialised sum

    #if x is out of the range from -2PI to 2PI, x modulo 2PI
    if(x< -(2 * math.pi) or x>(2 * math.pi)):
        x = x%(2*math.pi)

    #for loop for evaluating sin(x) using MacLaurin's Theorem
    for n in range(0,t+1):

        numerator = ((-1)**n) * (x**((2*n)+1))
        denominator = _factorial((2*n)+1)
        sum += numerator/denominator

    return sum #returns answer

#function for calculating cos(x) through sum of first t terms
def cos(x,t):

    sum=0.0 #initialised sum

    # if x is out of the range from -2PI to 2PI, x modulo 2PI
    if (x < -(2 * math.pi) or x > (2 * math.pi)):
        x = x % (2 * math.pi)

    # for loop for evaluating cosx using MacLaurin's Theorem
    for n in range(0,t+1):

        numerator = ((-1)**n)*(x**(2*n))
        denominator = _factorial(2*n)
```

```
        sum += numerator/denominator

    return sum #returns answer

a=50.0 #for x in radians
t=90 #for t trials
print("sin(%.4f) with %d trials = %.9f"%(a,t,sin(a,t)))
print("cos(%.4f) with %d trials = %.9f"%(a,t,cos(a,t)))
```

Console Listing

```
#a set as 50

sin(50.0000) with 90 trials = -0.262374854
cos(50.0000) with 90 trials = 0.964966028

Process finished with exit code 0


#a set as -50

sin(-50.0000) with 90 trials = 0.262374854
cos(-50.0000) with 90 trials = 0.964966028

Process finished with exit code 0


#a set as 2

sin(2.0000) with 90 trials = 0.909297427
cos(2.0000) with 90 trials = -0.416146837

Process finished with exit code 0


#a set as -2

sin(-2.0000) with 90 trials = -0.909297427
cos(-2.0000) with 90 trials = -0.416146837

Process finished with exit code 0
```

Task 9 – Finding Sum from Fibonacci Sequence

In this task, a program was required to return the sum of the first n terms of the Fibonacci sequence.

How the problem was solved

The method calcSumFib(n) calculates the sum of the first n terms of the Fibonacci sequence and returns it. It works iteratively. The current number x is initialized as 1. Making use of variables prev, sum and temp the current value is then moved on to the next Fibonacci number in the sequence and is incremented to the sum variable in the next iteration. This keeps on looping for n times.

How the program was tested

The program was tested using small values for n like 4, 9 then 15. Massively large amounts like for the first 1000 or 100000 numbers were also tested and proven to work (although at an understandably much longer execution time). The test case included in the output listing is for the first 40 numbers in the sequence.

Why Iteration is More Suitable for Fibonacci

Iteration is a more efficient way of processing Fibonacci numbers because it overwrites the existing information of each element every iteration, whereas in recursion it keeps on building a stack out of it, which for large lengths may return a Stack Overflow error.

Source Code (task9.py)

```
#Name: Russell Sammut-Bonnici
#ID: 0426299(M)
#Task: 9

def calcSumFib(n):

    prev=0 #initialised to 0
    x=1 #set to 1
    sum=0 #initialised to 0
    temp = 0  # initialised to 0, temporarily stores next number

    for i in range(0,n): #loops for n times

        print(x) # prints current number in the sequence
```

```
        #print(x)
        sum+=x

        #incrementing to next elements
        temp = prev + x
        prev = x
        x = temp

    return sum

n=40
print("\nSum of the first %d elements in the Fibonacci sequence is
%d"%(n,calcSumFib(n)))
```

Console Listing

```
1
1
2
3
5
8
13
21
34
55
89
144
233
377
610
987
1597
2584
4181
6765
10946
17711
28657
46368
75025
121393
196418
317811
514229
832040
1346269
2178309
3524578
5702887
9227465
14930352
24157817
39088169
63245986
102334155

Sum of the first 40 elements in the Fibonacci sequence is 267914295

Process finished with exit code 0
```