EECE 230X – Introduction to Computation and Programming
Programming Assignment 5

- This programming assignment consists of 4 problems.
- Prerequisites: Topics 6 and 7:
  - Problem 1: Topic 6
  - Problems 2,3,4: Topic 7
- Related material: Files, exception handling, plotting, and Monte Carlo simulation

Problem 1. Files

a) **Find word in a text file.** Implement the function `isWordInFile(fileName, word)`, which given a string `fileName` and a string `word`, checks whether or not `word` is in the file named `fileName`.

(*Hint:* Read the file in single shot into a string `s` and feel free to use the the membership test operator `in`: `word in s`)

Test it on the file named "test.txt" consisting of:

```
This is a test
for
Problem 1 of Programming Assignment 5
```

Test program:

```
print(isWordInFile("test.txt","Programming"))
print(isWordInFile("test.txt","programming"))
```

Output:

```
True
False
```

b) **Find word in text file: return line number.** This is a variation of Part (a) which requires reading the file line by line. Implement the function `wordSearch(fileName, word)`, which given a string `fileName` and a string `word` (not containing the new line character), searches for `word` in the file named `fileName`. If found, it should return the line number of the first occurrence. Otherwise, it should return 0. Feel free to use the membership test operator `in` for strings.

Test it on the file "test.txt" given in Part (a).

Test program:

```
print(wordSearch("test.txt","Programming"))
print(wordSearch("test.txt","programming"))
```

Output:

```
3
0
```

c) **Duplicate lines.** Implement the function `duplicateLines(fileName)`, which given a string `fileName`, opens the file name `fileName` for reading and creates a new file whose content is like `fileName` but with all lines duplicated. This function assumes that extension of `fileName` is ".txt", i.e., as a string, `fileName` is of the form `name.txt`, for some string `name`. The new file should be called `nameDuplicated.txt`. Use `assert` to stop the program if the extension of `fileName` is not ".txt". Display an appropriate error message.

Test your function on the file "test.txt" given in Part (a). It should create a new file named "testDuplicated.txt" consisting of:

```
This is a test
This is a test
for
for
Problem 1 of Programming Assignment 5
Problem 1 of Programming Assignment 5
```

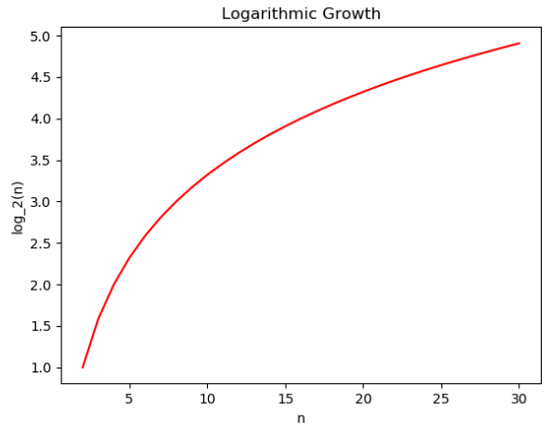
Test it also a file whose extension is not ".txt".

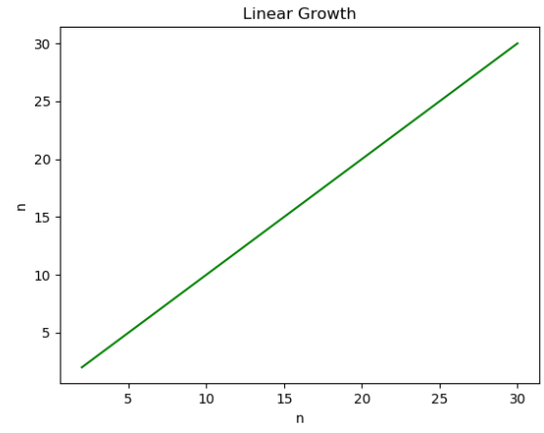
Problem 2. Plotting: growth of functions

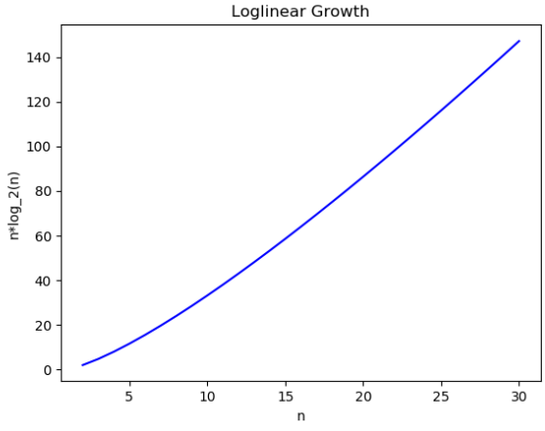
In this problem, we will use the `pyplot` module from the `matplot` library:

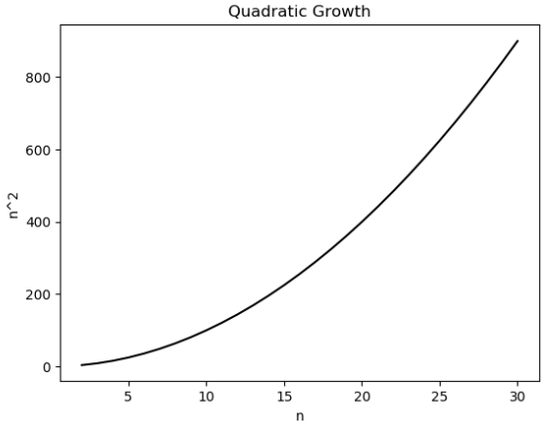
```
import matplotlib.pyplot as plt
```

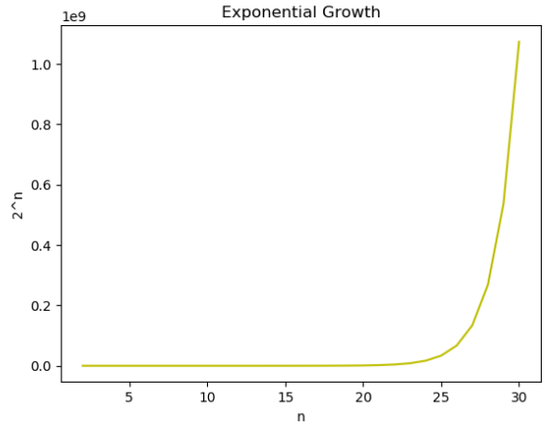
a) **Separate plots.** Write a Python script to create 5 figures plotting the functions $\log_2(n)$, n, $n\log_2(n)$, n^2, and 2^n, for $n = 2, 3, \ldots, 30$. As shown below, in each figure, the x-axis should be labeled n, the y-axis should be labeled with the corresponding function (use the symbol ^ to represent powers), and the title should correspondingly be "Logarithmic Growth", "Linear Growth", "Loglinear Growth", "Quadratic Growth", and "Exponential Growth". Use different colors for the different figures. Use the `log` function from the `math` module.



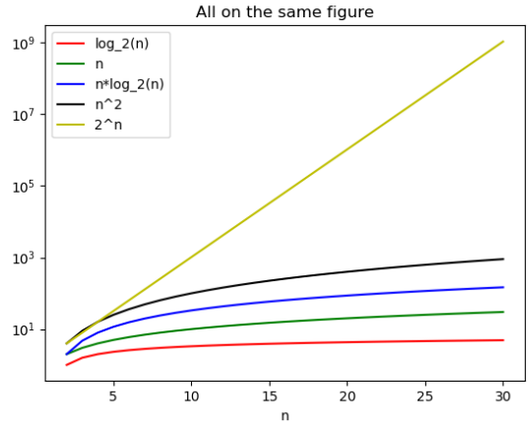




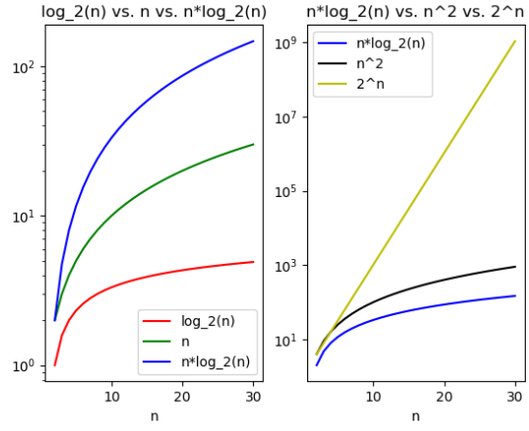




b) **Plots on the same figure, y-logscale.** Instead of plotting each function on a separate figure, plot them all on the same figure as shown below. Include a *legend* and use *logscale* on the y-axis (see the slides on plotting). Try it also without logscale to see why logscale is needed.



c) **Subplot, y-logscale.** Now, plot the 5 functions on the same figure divided into 2 subplots as shown below: use the `subplot` function and use logscale on the y-axis.



Problem 3. Primes revisited: generating primes, plotting density of prime

a) **Generate primes.** Implement the function `generatePrimes(n)`, which given a integer `n`, returns the tuple `(P,B)`, where

– `P` is a list consisitng of all prime number less than `n`,

– `B` is a length-`n` boolean list given by: for $i = 0, \ldots, n-1$, `B[i]=True` if `i` is prime and `B[i]=False` otherwise.

Note that even though 0 and 1 are not primes, we are including $i = 0, 1$ for convinience as indexing in lists starts from 0.

We can solve this problem as in Problem 3.b of Programming Assignment 2: loop on $i = 2, \ldots, n-1$, and for each i check if it is prime as in Problem 3.a in Programming Assignment 2. In total, this algorithm takes $O(n\sqrt{n}) = O(n^{3/2})$ arithmetic operations. In this problem, you will use lists to do it more efficiently as done by the Ancient Greek scientist Eratosthenes around 200 BC.

The idea of Eratosthenes is to write down the integers $2, 3, 4, \ldots, n-1$. Then cross all multiple all multiples of 2 less than n, then all multiples of 3 less than n, then move the next uncrossed number (i.e., 5) and cross all its multiples which are less than n, and so on. At some point there will be no next uncrossed number, in which case we stop. The uncrossed numbers are the prime numbers. Check the following wikipedia animation:
`https://en.wikipedia.org/wiki/File:Sieve_of_Eratosthenes_animation.gif`

Note that the above animation shows that the primes are being displayed as the numbers are being crossed (colored). You are not asked to do that. Instead, append them to the list P, initalized to the empty list `[]`. Use the `list.append` method.

Test program:

```
(P,B)=generatePrimes(2)
print(P)
(P,B)=generatePrimes(10)
print(P)
(P,B)=generatePrimes(20)
print(P)
(P,B)=generatePrimes(100)
print(P)
# Note: B will be used in Part (b) below
```

Output:

```
[]
[2, 3, 5, 7]
[2, 3, 5, 7, 11, 13, 17, 19]
[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]
```

Note: It can be shown that this takes $O(n \log\log n)$ arithmetic operations, which is significantly faster than $\Theta(n^{3/2})$. Namely, up to a constant factor, it takes in the order of $N = \frac{n}{2} + \frac{n}{3} + \frac{n}{5} + \frac{n}{7} + \ldots + \frac{n}{q}$ arithmetic operations, where q is the largest prime less than n. It can be shown that $N = O(n \log\log n)$. This is beyond the scope of this course.

b) **Density of primes.** Using your function in Part (a), implement the function `primesCount(n)`, which given a integer `n`, returns the length-`n` list `y`, given by `y[i]` = the number of prime numbers less than or equal to i, for $i = 0, \ldots, n-1$.

Test script:

```
print(primesCount(5))
print(primesCount(10))
```

Output:

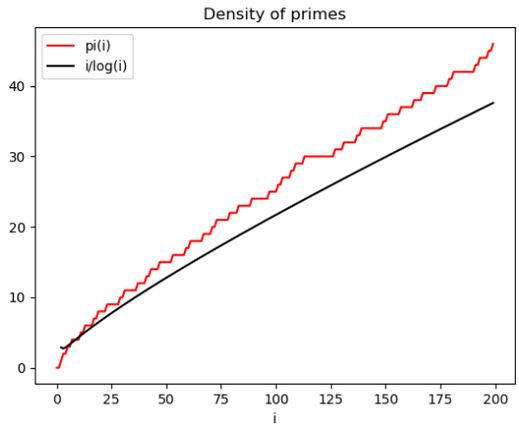
```
[0, 0, 1, 2, 2]
[0, 0, 1, 2, 2, 3, 3, 4, 4, 4]
```

c) **Plot density of primes.** Recall the definition of density of primes from Problem 3 in Programming Assignment 2.

If i is a nonnegative integer, the number of prime numbers less than or equal to i is denoted by $\pi(i)$.

Using your function in Part (c), write a Python script which plots $\pi(i)$ as a function of i, for $i = 0, \ldots, 199$. Compare this graph with the that of the function $\frac{i}{\log i}$, for $i = 2, \ldots, 99$ (see the note at the end Problem 3 in Programming Assignment 2).

Your script should produce the below figure.



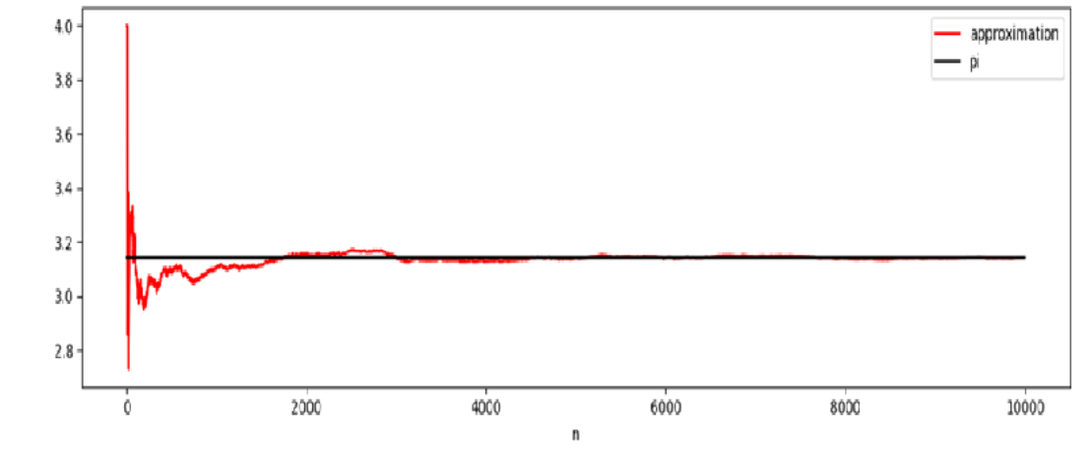
Problem 4. Monte Carlo simulation and plotting

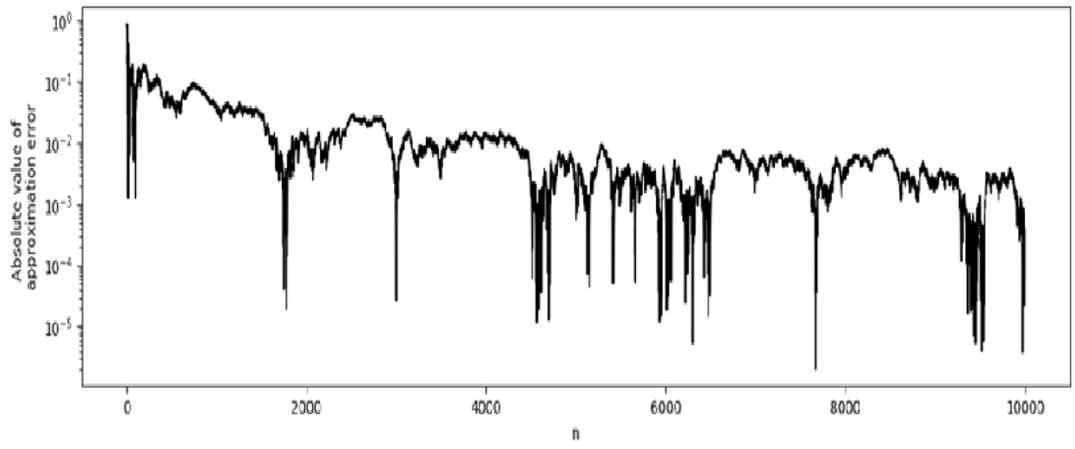
We studied in Topic 7 Monte Carlo simulation for approximating π. Namely, we wrote a function which given n, returns the approximate value of π using n samples. In this problem, you are asked to implement the function `monteCarloPiApproximation(N)`, which given an integer N returns a length-N list `approximatePi`, where for $n = 1, \ldots, N$, `approximatePi[n-1]` is the approximate value of π using n samples. Instead of sampling n points for each value of n (i.e., two nested loops: outer loop on n and inner loop to generate the n samples), do it using one loop on n. At each iteration, generate one sample and keep track of the number m of points in the the circle. At the end of each iteration, append $4m/n$ to a list `approximatePi`.

Then use your function to plot the approximate value of π as a function of n as well as the absolute value of the error (i.e., $|approximatePi[n-1] - \pi|$, for $n = 1, \ldots, N$) as shown below. Use the `subplot` function and use log-scale on the y-axis in the error graph.

Needed modules:

```
import numpy.random as rand
import matplotlib.pyplot as plt
from math import pi
```


approximation
pi
4.0
3.8
3.6
3.4
3.2
3.0
2.8
0
2000
4000
6000
8000
10000
n


Absolute value of approximation error
10^{0}
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
0
2000
4000
6000
8000
10000
n