Kernel: SageMath 10.7

Prime Explorations

MATH 242 Modern Computational Mathematics

Runtime of the Sieve of Eratosthenes

Here is an implementation of the sieve of Eratosthenes.

In [3]:
```
# function: sieveEratos
# input: integer nMax, which must be at least 2
# output: a list of primes up to nMax
def sieveEratos(nMax):
    # create a list of integers up to nMax
    # if you start the list from zero, then each number in the list is EQUAL to its index in the list, which is very convenient
    nums = list(range(nMax+1))

    # replace 1 in the list with 0
    nums[1] = 0

    # compute sqrt(nMax)
    nroot = floor(sqrt(nMax))

    # loop over each list item less than or equal to nroot
    i = 0
    while nums[i] <= nroot:
        # if nums[i] is not zero, then i is prime
        if nums[i] != 0:
            # i is prime, so set all of its multiples to zero
            j = i^2
            while j <= nMax:
                nums[j] = 0
                j += i
        i += 1

    # now extract all nonzero numbers from the list
    return [n for n in nums if n != 0]
```

In [4]:
```
sieveEratos(20)
```

Out[4]:
```
[2, 3, 5, 7, 11, 13, 17, 19]
```

How efficient is the sieve of Eratosthenes? Previously, we used the iPython magic commands **%time** and **%timeit** to measure runtime, like this:

In [5]:
```
%time primeList = sieveEratos(10^6)
```

Out[5]:
```
CPU times: user 1.35 s, sys: 73.7 ms, total: 1.42 s
Wall time: 1.67 s
```

In [6]:
```
%timeit primeList = sieveEratos(10^6)
```

Out[6]:
```
1.39 s ± 154 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

To store the runtime in a variable that we can use for further calculations, we must use a different approach. First, we import the Python's **time** module. (A Python *module* is a file containing code that can be used by other programs.) We can then record the time at the beginning and end of a calculation. The difference between these times is the amount of time that it took to do the calculation. The next code cell shows how to do this.

In [7]:
```
import time

# record the time before we run the sieve of Eratosthenes
startTime = time.time()
print(startTime)

# run the sieve of Eratosthenes
primeList = sieveEratos(10^6)

# record the time after we run the sieve of Eratosthenes
endTime = time.time()
print(endTime)

# compute the difference in the times that we recorded
timeElapsed = endTime - startTime
timeElapsed
```

Out[7]:
```
1775675182.97756
1775675185.3432143

2.365654230117798
```

We could also write a function that measures and returns the runtime, like this:

In [8]:
```
# function: runtimeEratos
# input: integer nMax, which must be at least 2
# output: the time required to use sieveEratos to make a list of primes
up to nMax
def runtimeEratos(nMax):
    # record the time before we run the sieve of Eratosthenes
    startTime = time.time()

    # run the sieve of Eratosthenes
    primeList = sieveEratos(nMax)

    # record the time after we run the sieve of Eratosthenes
    endTime = time.time()

    # return the difference in the times that we recorded
    return endTime - startTime
```

Try it out:

In [9]:
```
runtimeEratos(10^6)
```

Out[9]:
```
1.8957505226135254
```

It's useful to see how the runtime changes as we increase the value of nMax. To measure runtimes for many values of nMax, we could use a list comprehension, as in the next code cell.

In [10]:
```
# define a list of values for nMax
nMaxVals = range(10^5, 10^6 + 1, 10^5)
print("nMaxVals: ", list(nMaxVals))

# measure runtimes for these values
runTimes = [runtimeEratos(m) for m in nMaxVals]
print("runtimes: ", runTimes)
```

Out[10]:
```
nMaxVals:  [100000, 200000, 300000, 400000, 500000, 600000, 700000, 800000,
900000, 1000000]
runtimes:  [0.3640444278717041, 0.38044285774230957, 0.4409778118133545,
0.6127324104309082, 0.7409360408782959, 0.8677821159362793,
1.1671245098114014, 1.1227784156799316, 1.096097707748413,
1.276942491531372]
```

It would be nice to make a plot with nMax on the horizontal axis and runtimes on the vertical axis. For this, we need to convert the two lists above into a single list of ordered pairs. Each pair should be of the form (nMax, runtime). Fortunately, Python provides the zip function that converts two lists into a single list of pairs. Here is an example:

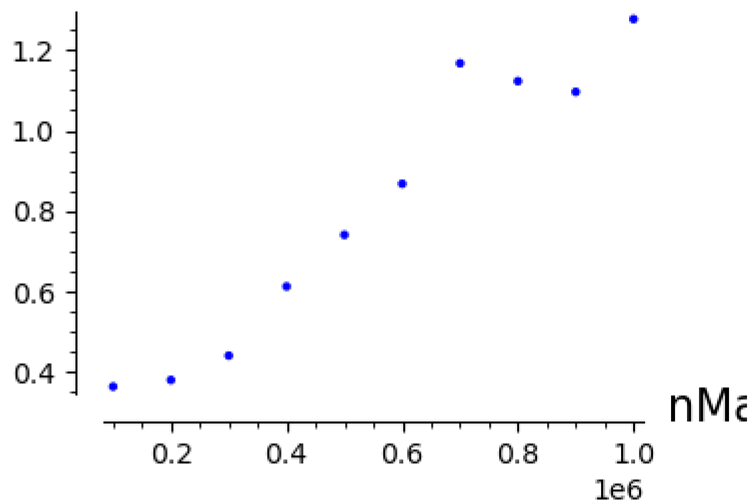
In [11]:
```
list(zip(nMaxVals, runTimes))
```

Out[11]:
```
[(100000, 0.3640444278717041),
 (200000, 0.38044285774230957),
 (300000, 0.4409778118133545),
 (400000, 0.6127324104309082),
 (500000, 0.7409360408782959),
 (600000, 0.8677821159362793),
 (700000, 1.1671245098114014),
 (800000, 1.1227784156799316),
 (900000, 1.096097707748413),
 (1000000, 1.276942491531372)]
```

Now we can make a nice plot of our runtime measurements:

In [12]:
```
list_plot( list(zip(nMaxVals, runTimes)), axes_labels=["nMax","runtime"],
figsize=4)
```

Out[12]:



Exploration: Units Digits of Primes

The only primes with a units digit of 2 or 5 are the primes 2 and 5. All other primes have a units digit of 1, 3, 7, or 9. How often do these digits occur?

First consider the primes less than 100. Of these primes, count how many have each units digit 1, 3, 7, and 9.

Here is one way to do it:

In [13]:

```
# first, make a list of primes
primeList = sieveEratos(100)
print(primeList)

#now count how many primes in the list end with each digit
digits = [1,3,7,9]
for d in digits:
    numPrimes = len([p for p in primeList if p%10 == d])
    print(f"units digit {d}: {numPrimes} primes")
```

Out[13]:

```
[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]
units digit 1: 5 primes
units digit 3: 7 primes
units digit 7: 6 primes
units digit 9: 5 primes
```

In [0]:

```
```

In [0]:

```
```

Now replace 100 by some other integer M. How many primes less than or equal to M have each units digit 1, 3, 7, and 9? Consider various values of M.

- What patterns do you observe in your counts?

- What questions arise during your exploration?
- What conjectures can you make?

In [14]:
```
# primes up to 200
nMax = 200
primeList = sieveEratos(nMax)
print(primeList)

#now count how many primes in the list end with each digit
digits = [1,3,7,9]
for d in digits:
    numPrimes = len([p for p in primeList if p%10 == d])
    print(f"units digit {d}: {numPrimes} primes")
```

Out[14]:
```
[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]
units digit 1: 10 primes
units digit 3: 12 primes
units digit 7: 12 primes
units digit 9: 10 primes
```

In [16]:
```
# primes up to 1000
nMax = 1000
primeList = sieveEratos(nMax)
print(primeList)

#now count how many primes in the list end with each digit
digits = [1,3,7,9]
for d in digits:
    numPrimes = len([p for p in primeList if p%10 == d])
    print(f"units digit {d}: {numPrimes} primes")
```

Out[16]:
```
[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]
units digit 1: 40 primes
units digit 3: 42 primes
units digit 7: 46 primes
units digit 9: 38 primes
```

In [0]:
```
```

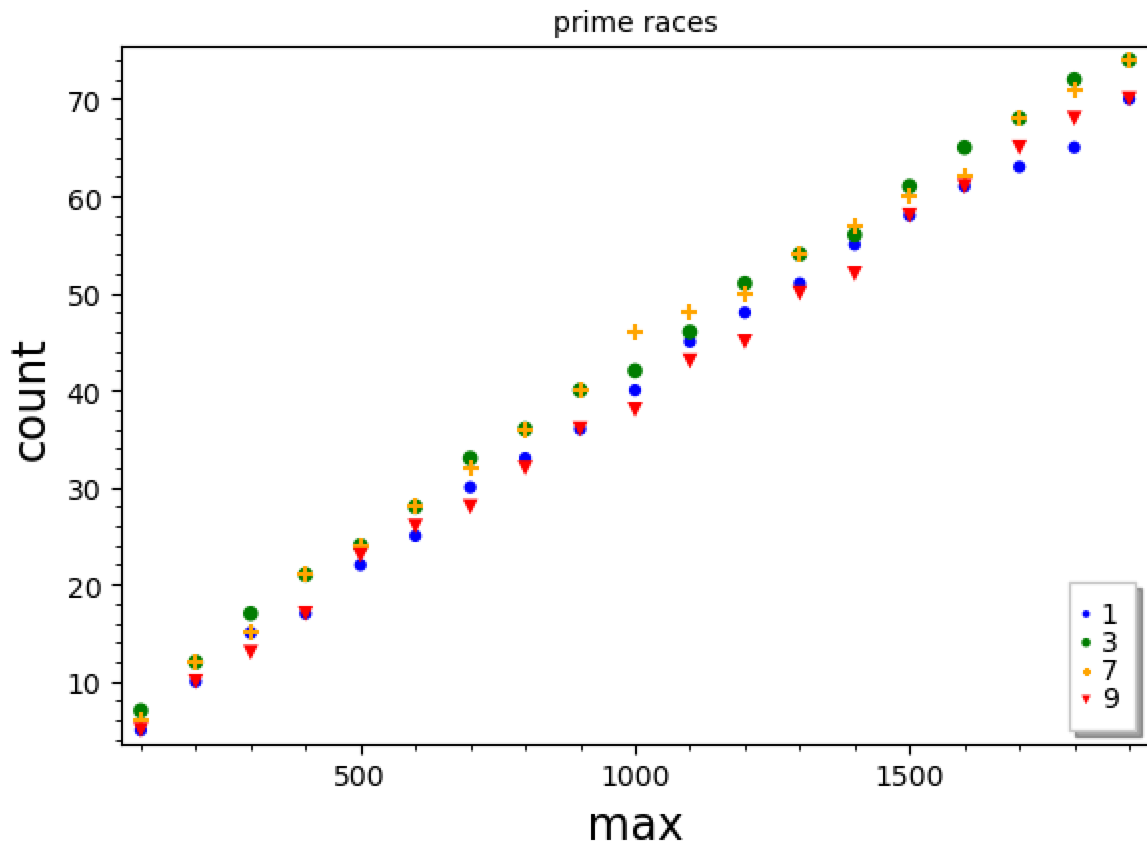
Here is a more thorough exploration with a plot of the counts:

In [23]:

```
# function to count the primes in a given list with each units digit
def countByUnit(primeList, units = [1,3,7,9]):
    counts = [0]*len(units)
    for p in primeList:
        u = p % 10
        for i in range(len(units)):
            if u == units[i]:
                counts[i] += 1
                break
    return counts

# make a big list of primes
primeList = sieveEratos(100000)

# specify a range of nMax values
nMaxVals = range(100,3000,100)

# set up lists to store counts for each units digit
ones = []
threes = []
sevens = []
nines = []

# count the primes by units digit up to each nMaxVal
for m in nMaxVals:
    pList = [p for p in primeList if p <= m]
    temp = countByUnit(pList)
    ones.append(temp[0])
    threes.append(temp[1])
    sevens.append(temp[2])
    nines.append(temp[3])

#testing
print(list(zip(nMaxVals, ones)))

# create plots for each units digit
plot1 = list_plot(list(zip(mVals,ones)), legend_label="1", size=20)
plot3 = list_plot(list(zip(mVals,threes)), legend_label="3", marker="o",
size=30, color="green")
plot7 = list_plot(list(zip(mVals,sevens)), legend_label="7", marker="+",
size=30, color="orange")
plot9 = list_plot(list(zip(mVals,nines)), legend_label="9", marker="v",
size=30, color="red")

# show a combined plot
combined = plot1 + plot3 + plot7 + plot9
combined.axes_labels(["max","count"])
combined.show(title="prime races", frame=True, legend_loc="lower right",
figsize=6)
```

Out[23]:

```
[(100, 5), (200, 10), (300, 15), (400, 17), (500, 22), (600, 25), (700, 30),
(800, 33), (900, 36), (1000, 40), (1100, 45), (1200, 48), (1300, 51), (1400,
55), (1500, 58), (1600, 61), (1700, 63), (1800, 65), (1900, 70), (2000, 73),
(2100, 75), (2200, 79), (2300, 82), (2400, 87), (2500, 89), (2600, 93),
(2700, 95), (2800, 99), (2900, 102)]
```



Extension

Instead of counting primes by their units digits, what if you count the primes by their remainders after division by some other number? For example:

- Are there more primes less than or equal to M that are 1 more than a multiple of 3 or 2 more than a multiple of 3? How does this depend on M?
- How many primes less than or equal to M are 1 more than a multiple of 8? ...3 more than a multiple of 8? ...5 more than a multiple of 8? ...7 more than a multiple of 8? How do these counts depend on M?
- What questions arise during your exploration?
- What conjectures can you make?

In [0]:

In [0]:

In [0]:

In [0]: