Example: Finding the primes below n

Problem: Find all the primes below n

Naive algorithm

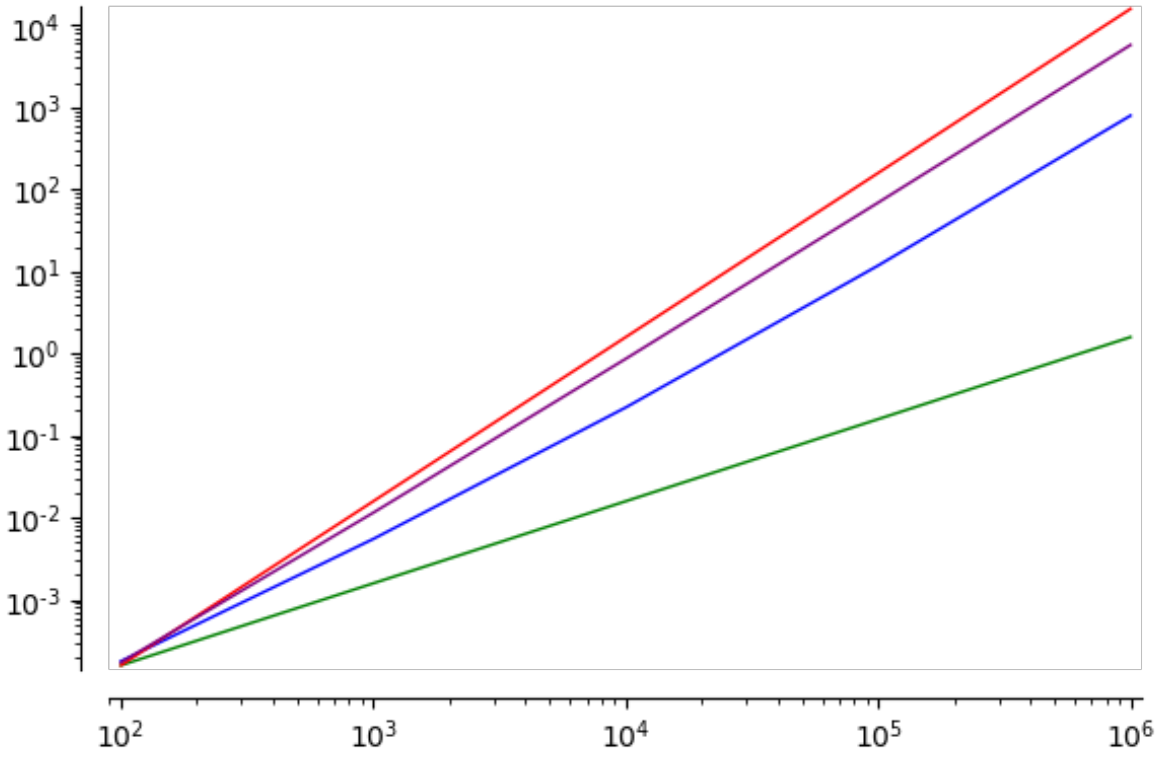
The naive algorithm tests for each i from 2 to n if n is divisible by a smaller prime. If it is, it is not a prime. If it is not, it is a prime.

The cost of the algorithm is n iterations of the outer loop. Each inner loop costs a division per smaller prime. Approximating the number of primes below i as i/log i, we get approximately n^2/log n divisions as the cost.

In [101]:
```
%%time

n = 1000000

primes = []
for i in range(2,n+1):
    IsPrime = True
    for j in primes:
        if i % j == 0:
            IsPrime = False
            break
    if IsPrime:
        primes.append(i)

#print(primes)
```

```
CPU times: user 13min 10s, sys: 1.55 s, total: 13min 12s
Wall time: 13min 13s
```

In [114]:
```
timings = [ (100, 0.000178), (1000, 0.0055), (10000, 0.220), (10000
0, 11.8), (1000000, 793) ]
P1 = list_plot_loglog(timings, color='blue', plotjoined=True)
# lines to compare against.
P2 = list_plot_loglog([(i,i*10^-5.8) for i in [ 100, 1000, 10000, 1
00000, 1000000 ]], color='green', plotjoined=True)
P3 = list_plot_loglog([(i,i*i*10^-7.8) for i in [ 100, 1000, 10000,
100000, 1000000 ]], color='red', plotjoined=True)
P4 = list_plot_loglog([(i,i*i/log(i)*10^-7.1) for i in [ 100, 1000,
10000, 100000, 1000000 ]], color='purple', plotjoined=True)
show(P1 + P2 + P3 + P4)
```



Sieve of Eratosthenes

We keep a list of numbers that may be prime. First we discard 0 and 1. Then the smallest remaining number must be prime. Then we discard all the multiples of that prime. And repeat.

The cost of the algorithm is n iterations of the outer loop, but most of those iterations are trivial. Instead, for each prime, we do n/i discards. The primes are mostly small, so we approximate the cost per prime as n operations. Which means we get (n/log n) n operations.

However, this estimate is too large. We get that the cost is sum n/i for primes i, which is equal to n(sum 1/i). We can approximate the sum of the reciprocals of the primes as log log n, which gives us a cost of n log log n.

In [46]:
```
%%time

n = 10000000

numbers = [ True ]*n
primes = []
numbers[0] = False
numbers[1] = False

for i in range(n):
    if numbers[i]: # i is prime
        primes.append(i)
        for j in range(i+i, n, i):
            numbers[j] = False

#print(primes)
```

```
CPU times: user 4.26 s, sys: 52 ms, total: 4.31 s
Wall time: 4.33 s
```

In [113]:
```
timings = [ (100, 0.000065), (1000, 0.000400), (10000, 0.0043), (10
0000, 0.044), (1000000, 0.400),  (10000000, 4.33)]
P1 = list_plot_loglog(timings, color='blue', plotjoined=True)
# lines to compare against.
P2 = list_plot_loglog([(i,i*10^(-6.2)) for i in [ 100, 1000, 10000,
100000, 1000000, 10000000 ]], color='green', plotjoined=True)
P3 = list_plot_loglog([(i,i*i*10^(-8.2)) for i in [ 100, 1000, 1000
0, 100000, 1000000, 10000000 ]], color='red', plotjoined=True)
show(P1+P2+P3)
```