PrimeMultiplicationTable

Objective

Write a program that prints out a multiplication table of the first 10 prime number.

- The program must run from the command line and print one table to STDOU
- The first row and column of the table should have the 10 primes, with each containing the product of the primes for the corresponding row and column.

Note:

- Consider complexity. How fast does your code run? How does it scale?
- Consider cases where we want N primes.
- Do not use the Prime class from stdlib (write your own code).

Execute

```
$ cd <project directory>
$ python PrimeMultiplicationTable.py
```

Result

```
C:\Users\jeffrey\Documents\Workspace\PrimeMultiplicationTable
λ python PrimeMultiplicationTable.py
       |    2    3    5    7   11   13   17   19   23   29
-----------------------------------------------------------
     2 |    4    6   10   14   22   26   34   38   46   58
     3 |    6    9   15   21   33   39   51   57   69   87
     5 |   10   15   25   35   55   65   85   95  115  145
     7 |   14   21   35   49   77   91  119  133  161  203
    11 |   22   33   55   77  121  143  187  209  253  319
    13 |   26   39   65   91  143  169  221  247  299  377
    17 |   34   51   85  119  187  221  289  323  391  493
    19 |   38   57   95  133  209  247  323  361  437  551
    23 |   46   69  115  161  253  299  391  437  529  667
    29 |   58   87  145  203  319  377  493  551  667  841
```

My Thinking Process

After review the question, I found it can be divided into two parts. First one is to find the first N primes, second is to generate the table nicely. And how to realize the first part is crucial here.

Solution 1

The first idea I got is based on the definition of Prime number.

> *A prime number (or a prime) is a natural number greater than 1 that has no positive divisors other than 1 and itself.*

So, basically, I can try to divide each number N by all numbers between 2 and N-1 to see if it can be divided evenly. If it can't we find a prime. The time complexity of this solution is O(n^2).

Then, I found there are actually some ways to improve the algorithm above.

- No even number except 2 can be a prime number. So we don't need to check if a number can be divided evenly by an even number. This fact can reduce half of number we need to try.
- If a number is not a prime. It should be a multiplication of at least two numbers. Then one of them must be smaller than equal to sqrt(N). So, we just need to try to divide each number by 2 to sqrt(N) instead of N.
- After optimized the code by the two ideas above, I find there are still some duplicated division operations. For instance, to check 101, we will try dividends 3,5,7,9. However, 9 is not necessary to be checked since 3 has been checked already. So we just need to try to divide the number by all primes we have found so far.

It is implemented in method **get_primes_1** with time complexity **lower than O(n^1.5)**.

Solution 2

After completing the code, I start thinking if there is better way to solve the problem. Division operations are generally slower than other operations, because it includes a lot of shifting and subtractions. Then, I come up with a theory I learned before named Sieve of Eratosthenes. It is a typical example to exchange time with space.

Make a list of all the integers less than or equal to n (and greater than one). Strike out the multiples of all primes less than or equal to the square root of n, then the numbers that are left are the primes.

A challenge here is we need to know the upper bound of nth prime firstly. To find it out, I use the Prime Number Theorem. The prime distribution is $\pi(N) \sim N / \log(N)$, where $\pi(N)$ is the prime-counting function and $\log(N)$ is the natural logarithm of N. By using this algorithm, we can approximate the nth prime number $P(N) \sim N\log(N)$. Since there will be some deviations (<20%), I expand the table size by 30% to make sure it covers all prime numbers we need.

It is implemented in method **get_primes_2** with time complexity **O(nloglogn)**.

Solution 3

Then, I'm thinking if we can move a little bit further to make it become O(n). In solution 2, some composite numbers are struck out more than once. If we can stipulate that each composite number must be struck out by its smallest prime factor, then we can say each composite number will only be touched once, which makes the time complexity become O(n) .

It is implemented in method **get_primes_3** with time complexity **O(n)**.

Future Improvement

We can store prime status in bits rather than bytes (boolean type) to reduce the space complexity. This can also help to reduce the cache miss and improve the performance.

Scalability

The algorithm can work with larger numbers. The table column width will be adjusted dynamically.

```
C:\Users\jeffrey\Documents\Workspace\PrimeMultiplicationTable
λ python PrimeMultiplicationTable.py
```

	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
2	4	6	10	14	22	26	34	38	46	58	62	74	82	86	94	106	118	122	134	142	146	158	166	178	194	202	206	214	218	226
3	6	9	15	21	33	39	51	57	69	87	93	111	123	129	141	159	177	183	201	213	219	237	249	267	291	303	309	321	327	339
5	10	15	25	35	55	65	85	95	115	145	155	185	205	215	235	265	295	305	335	355	365	395	415	445	485	505	515	535	545	565
7	14	21	35	49	77	91	119	133	161	203	217	259	287	301	329	371	413	427	469	497	511	553	581	623	679	707	721	749	763	791
11	22	33	55	77	121	143	187	209	253	319	341	407	451	473	517	583	649	671	737	781	803	869	913	979	1067	1111	1133	1177	1199	1243
13	26	39	65	91	143	169	221	247	299	377	403	481	533	559	611	689	767	793	871	923	949	1027	1079	1157	1261	1313	1339	1391	1417	1469
17	34	51	85	119	187	221	289	323	391	493	527	629	697	731	799	901	1003	1037	1139	1207	1241	1343	1411	1513	1649	1717	1751	1819	1853	1921
19	38	57	95	133	209	247	323	361	437	551	589	703	779	817	893	1007	1121	1159	1273	1349	1387	1501	1577	1691	1843	1919	1957	2033	2071	2147
23	46	69	115	161	253	299	391	437	529	667	713	851	943	989	1081	1219	1357	1403	1541	1633	1679	1817	1909	2047	2231	2323	2369	2461	2507	2599
29	58	87	145	203	319	377	493	551	667	841	899	1073	1189	1247	1363	1537	1711	1769	1943	2059	2117	2291	2407	2581	2813	2929	2987	3103	3161	3277
31	62	93	155	217	341	403	527	589	713	899	961	1147	1271	1333	1457	1643	1829	1891	2077	2201	2263	2449	2573	2759	3007	3131	3193	3317	3379	3503
37	74	111	185	259	407	481	629	703	851	1073	1147	1369	1517	1591	1739	1961	2183	2257	2479	2627	2701	2923	3071	3293	3589	3737	3811	3959	4033	4181
41	82	123	205	287	451	533	697	779	943	1189	1271	1517	1681	1763	1927	2173	2419	2501	2747	2911	2993	3239	3403	3649	3977	4141	4223	4387	4469	4633
43	86	129	215	301	473	559	731	817	989	1247	1333	1591	1763	1849	2021	2279	2537	2623	2881	3053	3139	3397	3569	3827	4171	4343	4429	4601	4687	4859
47	94	141	235	329	517	611	799	893	1081	1363	1457	1739	1927	2021	2209	2491	2773	2867	3149	3337	3431	3713	3901	4183	4559	4747	4841	5029	5123	5311
53	106	159	265	371	583	689	901	1007	1219	1537	1643	1961	2173	2279	2491	2809	3127	3233	3551	3763	3869	4187	4399	4717	5141	5353	5459	5671	5777	5989
59	118	177	295	413	649	767	1003	1121	1357	1711	1829	2183	2419	2537	2773	3127	3481	3599	3953	4189	4307	4661	4897	5251	5723	5959	6077	6313	6431	6667
61	122	183	305	427	671	793	1037	1159	1403	1769	1891	2257	2501	2623	2867	3233	3599	3721	4087	4331	4453	4819	5063	5429	5917	6161	6283	6527	6649	6893
67	134	201	335	469	737	871	1139	1273	1541	1943	2077	2479	2747	2881	3149	3551	3953	4087	4489	4757	4891	5293	5561	5963	6499	6767	6901	7169	7303	7571
71	142	213	355	497	781	923	1207	1349	1633	2059	2201	2627	2911	3053	3337	3763	4189	4331	4757	5041	5183	5609	5893	6319	6887	7171	7313	7597	7739	8023
73	146	219	365	511	803	949	1241	1387	1679	2117	2263	2701	2993	3139	3431	3869	4307	4453	4891	5183	5329	5767	6059	6497	7081	7373	7519	7811	7957	8249
79	158	237	395	553	869	1027	1343	1501	1817	2291	2449	2923	3239	3397	3713	4187	4661	4819	5293	5609	5767	6241	6557	7031	7663	7979	8137	8453	8611	8927
83	166	249	415	581	913	1079	1411	1577	1909	2407	2573	3071	3403	3569	3901	4399	4897	5063	5561	5893	6059	6557	6889	7387	8051	8383	8549	8881	9047	9379
89	178	267	445	623	979	1157	1513	1691	2047	2581	2759	3293	3649	3827	4183	4717	5251	5429	5963	6319	6497	7031	7387	7921	8633	8989	9167	9523	9701	10057
97	194	291	485	679	1067	1261	1649	1843	2231	2813	3007	3589	3977	4171	4559	5141	5723	5917	6499	6887	7081	7663	8051	8633	9409	9797	9991	10379	10573	10961
101	202	303	505	707	1111	1313	1717	1919	2323	2929	3131	3737	4141	4343	4747	5353	5959	6161	6767	7171	7373	7979	8383	8989	9797	10201	10403	10807	11009	11413
103	206	309	515	721	1133	1339	1751	1957	2369	2987	3193	3811	4223	4429	4841	5459	6077	6283	6901	7313	7519	8137	8549	9167	9991	10403	10609	11021	11227	11639
107	214	321	535	749	1177	1391	1819	2033	2461	3103	3317	3959	4387	4601	5029	5671	6313	6527	7169	7597	7811	8453	8881	9523	10379	10807	11021	11449	11663	12091
109	218	327	545	763	1199	1417	1853	2071	2507	3161	3379	4033	4469	4687	5123	5777	6431	6649	7303	7739	7957	8611	9047	9701	10573	11009	11227	11663	11881	12317
113	226	339	565	791	1243	1469	1921	2147	2599	3277	3503	4181	4633	4859	5311	5989	6667	6893	7571	8023	8249	8927	9379	10057	10961	11413	11639	12091	12317	12769

Here is the result for 30 primes

Handle super large number

Solution 1 and Solution 2(sieve of Eratosthenes) can be scaled on machine clusters by map-reduce mode. In map function code, we divide the candidate numbers into groups and distribute the data to different machines. Then in reduce module, we collect the primes found by each machine and create the result.

TDD Test Case

I created two types of test cases.

- First group includes numbers and primes of certain number captured from official prime table
- Second group includes random number N and the Nth prime. It Avoids putting in super long prime lists and keeps the file readable.

Test Result

```
C:\Users\jeffrey\Documents\Workspace\PrimeMultiplicationTable
λ python test.py
test get primes method with the PRIME LIST returned
test num = -1
test num = 0
test num = 1
test num = 2
test num = 3
test num = 4
test num = 5
test num = 10
test num = 20
test num = 50
test num = 100
test num = 500
test num = 1000
test get primes method with the LAST PRIME returned
test  num = 355
test  num = 999
test  num = 9999
test  num = 99999
test  num = 100001
test  num = 1000000
.
----------------------------------------------------------------------
Ran 1 test in 9.080s

OK
```