Project Euler: Problem 351

Contents

1 Introduction

Problem 351 is a very important problem. As we will see, the solution for $H(n)$, the number of hidden points in a hexagonal orchard of order n, involves something known as *Euler's totient summatory function*, or $\Phi(n)$.

The main goal of this paper is to investigate the different methods for calculating the summatory function, starting out with the relatively inefficient $O(n^2 \log n)$ algorithm and gradually working our way to the very fast and efficient $O(n^{2/3}(\log \log n)^{1/3})$ algorithm.

Even though we will be focusing on the totient summatory function, the underlying logic behind the derivations and problem-solving strategies (especially for the fast sublinear methods) is essential for tackling comparable problems that don't necessarily involve totients and have limits much higher than the 10^8 bound posed in Problem 351.

2 Solving Problem 351

We aim to calculate $H(n)$, the number of hidden points in a hexagonal orchard of order n. A hidden point is a point that isn't directly visible from the origin due to a closer point lying within the same line of sight.

However, the first thing we should note before we perform any calculations is the six-way non-overlapping symmetry in the orchard as shown below (hidden points are highlighted in green):

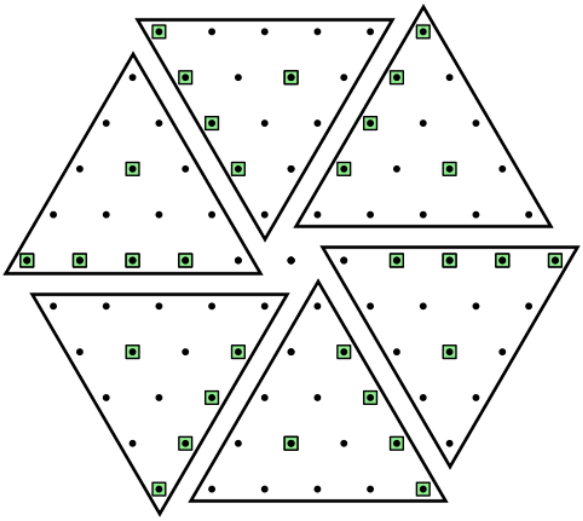

Figure 1: Six-way symmetry

Author: Marcus Andrews

We refer to each grouping as a single *region*. Each region contains the same number of hidden and visible points, so whatever result we get for one region can be multiplied by 6. Of course, we must not forget to include the origin point separately when appropriate.

2.1 Deriving $H(n)$

To make things easier to process, let's take the origin point and all the points in a single region and apply a skew-transformation (which will still preserve all visibility properties) so that we can map everything to a grid graph coordinate system:

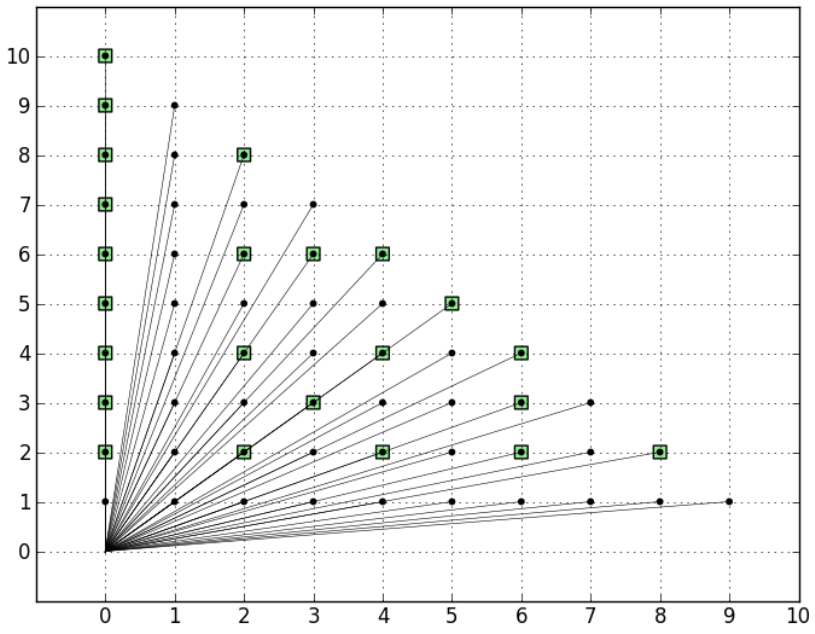

Figure 2: A single region mapped to a grid graph (orchard of order 10)

Let a given point's coordinates be (x', y'). If we draw lines of visibility from the origin to all other points in the region, some points will rest on the same lines. For example, the points $(2, 1)$, $(4, 2)$, $(6, 3)$, etc all rest on the same line. However, only point $(2, 1)$ is visible because it is the closest such point to the origin, and thus the other points behind it are hidden.

This tells us something interesting: if the quotient of a point's coordinates $\frac{y'}{x'}$ can be reduced to a simpler fraction, then we immediately get another point on the same line closer to the origin. Therefore, the closest point to the origin on such a line is the one for which x' is coprime to y'.

Thus, to get the number of hidden points $H(n)$, we subtract the number of visible points $V(n)$ from the number of total points $T(n)$:

$$H(n) = T(n) - V(n) \tag{2.1.1}$$

$T(n)$ and $V(n)$ can be broken down into $t(n)$ and $v(n)$ (the counts per region):

$$H(n) = (1 + 6t(n)) - (1 + 6v(n)) = 6(t(n) - v(n)) \tag{2.1.2}$$

We see from Figure 2 that all visible points in a region fall within $1 \leq y' \leq n$ and $0 \leq x' \leq n - y'$ with $\gcd(x', y') = 1$. If we let $x = x' + y'$ and $y = y'$, then we have $1 \leq y \leq x \leq n$ with $\gcd(x - y, y) = 1$. However, since $\gcd(x - y, y) = \gcd(x, y)$, we can use the latter version. This brings us to the following expression[1]:

$$v(n) = \sum_{x=1}^{n} \sum_{y=1}^{x} [\gcd(x, y) = 1] \cdot (1) \tag{2.1.3}$$

The loop over y in $v(n)$ is better known as *Euler's totient function*, $\phi(x)$, the count of numbers up to x that are coprime to x. More broadly, $v(n)$ itself is the *totient summatory function*, $\Phi(n)$:

$$\phi(x) = \sum_{y=1}^{x} [\gcd(x, y) = 1] \cdot (1) \tag{2.1.4}$$

$$\Phi(n) = v(n) = \sum_{x=1}^{n} \phi(x) \tag{2.1.5}$$

Moving onto the total number of points, we see that $t(n)$ is directly computable. It is the same expression as $v(n)$, only without the condition on gcd. Therefore:

$$t(n) = \sum_{x=1}^{n} \sum_{y=1}^{x} 1 = \sum_{x=1}^{n} x = \frac{n(n+1)}{2} \tag{2.1.6}$$

Giving us:

$$H(n) = 6 \left(\frac{n(n+1)}{2} - \sum_{x=1}^{n} \sum_{y=1}^{x} [\gcd(x, y) = 1] \cdot (1) \right) = 3n(n+1) - 6\Phi(n) \tag{2.1.7}$$

In the upcoming sections, we will investigate ways to calculate $\Phi(n)$.

[1] The [ ] brackets are called *Iverson brackets*, which are Boolean conditionals that evaluate to 1 if the condition is true, and 0 if the condition is false.

3 Quadratic-time algorithms for $\Phi(n)$

3.1 $O(n^2 \log n)$ time algorithm

One of the slowest ways to compute the summatory function is by directly evaluating expression 2.1.3 without any modification (looping over every pair x, y and calculating the gcd to check for coprimality). The gcd calculations themselves can be done via the Euclidean algorithm.

Algorithm 1 Totient summatory function in $O(n^2 \log n)$ time

```
1: function GCD(a,b)
2:     r ← a mod b
3:     while r ≠ 0 do
4:         a ← b
5:         b ← r
6:         r ← a mod b
7:     end while
8:     return b
9: end function
10:
11: function TotientSum(n)
12:     res ← 0
13:     for x = 1 to n do
14:         for y = 1 to x do
15:             if GCD(x,y) = 1 then
16:                 res ← res + 1
17:             end if
18:         end for
19:     end for
20:     return res
21: end function
```

3.2 $O(n^2)$ time algorithm

A useful strategy in many problems is to iterate over fixed outcomes of something. In the previous section, we used the Euclidean algorithm to calculate the gcd for every possible (x, y), but how many pairs have $\gcd(x, y) = g$ for fixed g?

If we have some point (a, b) where $\gcd(a, b) = 1$ and $a \geq b$, then the point (ga, gb) has $\gcd(ga, gb) = g$. This can continue as long as $ga \leq n$. In other words, for a fixed g, we have $1 \leq b \leq a \leq \lfloor \frac{n}{g} \rfloor$ where $\gcd(a, b) = 1$. This, in fact, is the same definition as $v(n)$ itself, only instead of n we are using $\lfloor \frac{n}{g} \rfloor$ as our upper bound[2].

[2]The *floor function*, $\lfloor x \rfloor$, rounds x down to the nearest integer value.

This suggests the following identity:

$$t(n) = \sum_{g=1}^{n} v\left(\left\lfloor \frac{n}{g} \right\rfloor\right) \tag{3.2.1}$$

Some trivial rearrangement gives us a recursive formula for $v(n)$:

$$v(n) = \frac{n(n+1)}{2} - \sum_{g=2}^{n} v\left(\left\lfloor \frac{n}{g} \right\rfloor\right) \tag{3.2.2}$$

Another way to derive equation 3.2.2: to get the coprime count, we take the total, subtract everything not coprime, and then reduce the subtracted portion by g:

$$\begin{aligned} v(n) &= \sum_{x=1}^{n}\sum_{y=1}^{x}[\gcd(x,y)=1]\cdot(1) \\ &= t(n) - \sum_{x=1}^{n}\sum_{y=1}^{x}[\gcd(x,y)>1]\cdot(1) \\ &= t(n) - \sum_{g=2}^{n}\sum_{x=1}^{n}\sum_{y=1}^{x}[\gcd(x,y)=g]\cdot(1) \\ &= t(n) - \sum_{g=2}^{n}\sum_{a=1}^{\lfloor \frac{n}{g} \rfloor}\sum_{b=1}^{a}[\gcd(a,b)=1]\cdot(1) \\ &= \frac{n(n+1)}{2} - \sum_{g=2}^{n} v\left(\left\lfloor \frac{n}{g} \right\rfloor\right) \end{aligned} \tag{3.2.3}$$

We can use dynamic programming with an iterative translation of $v(n)$ in order to re-use old results and avoid recomputing the same things repeatedly.

Algorithm 2 Totient summatory function in $O(n^2)$ time

```
1:  function TotientSum(n)
2:      v ← [0] * (n + 1)                ▷ 0-indexed array containing n + 1 values of 0
3:      for x = 1 to n do
4:          v[x] ← x * (x + 1)/2
5:          for g = 2 to x do
6:              v[x] ← v[x] − v[floor(x/g)]
7:          end for
8:      end for
9:      return v[n]
10: end function
```


Author: Marcus Andrews

4 Subquadratic-time algorithms for $\Phi(n)$

4.1 $O(n^{3/2})$ time algorithm

Consider factorizing x for all $1 \leq x \leq n$ in order to calculate each $\phi(x)$ instead. If x is coprime to some smaller number y, then x and y have no prime factors in common by definition. Thus, if we know how many numbers less than x have at least one prime factor in common, then we can subtract this count from x to get $\phi(x)$.

Let's use an example and say $x = 60$, which has three prime factors: 2, 3, and 5. How many numbers up to x are divisible by prime p? Exactly $\frac{x}{p}$.

So we subtract from 60 the counts of $\frac{60}{2} = 30$ numbers divisible by 2, $\frac{60}{3} = 20$ numbers divisible by 3, and $\frac{60}{5} = 12$ numbers divisible by 5.

However, as we learned from Project Euler Problem 1, there are double-counting issues with such methods, so we must add back the counts of $\frac{60}{(2)(3)} = 10$ numbers divisible by both 2 and 3, $\frac{60}{(2)(5)} = 6$ numbers divisible by both 2 and 5, and $\frac{60}{(3)(5)} = 4$ numbers divisible by both 3 and 5.

But we're still not done. As a result of applying the double-counting correction, we have now added back too much for the numbers that were initially triple-counted in the very first step, so we must subtract $\frac{60}{(2)(3)(5)} = 2$ for the numbers that are divisible by all three primes 2, 3, and 5. Therefore, we have the result: $\phi(60) = 60 - (30 + 20 + 12) + (10 + 6 + 4) - (2) = 16$.

In general, this is an example of the *inclusion-exclusion principle* at work. Can we find a simple, generalized formula for $\phi(x)$ if x contains k primes?

If $k = 1$ (our "base" case):

$$\begin{aligned}\phi(x) &= x - \frac{x}{p_1} \\ &= x\left(1 - \frac{1}{p_1}\right)\end{aligned}$$

If $k = 2$:

$$\begin{aligned}\phi(x) &= x - \frac{x}{p_1} - \frac{x}{p_2} + \frac{x}{p_1 p_2} \\ &= x\left(\left(1 - \frac{1}{p_1}\right) - \frac{1}{p_2} + \frac{1}{p_1 p_2}\right) \\ &= x\left(\left(1 - \frac{1}{p_1}\right) - \frac{1}{p_2}\left(1 - \frac{1}{p_1}\right)\right) \\ &= x\left(1 - \frac{1}{p_1}\right)\left(1 - \frac{1}{p_2}\right)\end{aligned}$$

If $k = 3$:

$$\begin{aligned}\phi(x) &= x - \frac{x}{p_1} - \frac{x}{p_2} - \frac{x}{p_3} + \frac{x}{p_1 p_2} + \frac{x}{p_1 p_3} + \frac{x}{p_2 p_3} - \frac{x}{p_1 p_2 p_3} \\ &= x\left(\left(1 - \frac{1}{p_1}\right)\left(1 - \frac{1}{p_2}\right) - \frac{1}{p_3} + \frac{1}{p_1 p_3} + \frac{1}{p_2 p_3} - \frac{1}{p_1 p_2 p_3}\right) \\ &= x\left(\left(1 - \frac{1}{p_1}\right)\left(1 - \frac{1}{p_2}\right) - \frac{1}{p_3}\left(1 - \frac{1}{p_1} - \frac{1}{p_2} + \frac{1}{p_1 p_2}\right)\right) \\ &= x\left(\left(1 - \frac{1}{p_1}\right)\left(1 - \frac{1}{p_2}\right) - \frac{1}{p_3}\left(1 - \frac{1}{p_1}\right)\left(1 - \frac{1}{p_2}\right)\right) \\ &= x\left(1 - \frac{1}{p_1}\right)\left(1 - \frac{1}{p_2}\right)\left(1 - \frac{1}{p_3}\right)\end{aligned}$$

The same pattern holds by induction for any number of primes. The final result is *Euler's product formula* for the totient function:

$$\phi(x) = x \prod_{\substack{p|x \\ p \in \texttt{PRIMES}}} \left(1 - \frac{1}{p}\right) \tag{4.1.1}$$

For each x, trial division yields the prime factors of x. For each prime factorization, the Euler product formula yields $\phi(x)$. The sum over all $\phi(x)$ yields $\Phi(n)$:

Algorithm 3 Totient summatory function in $O(n^{3/2})$ time

```
1:  function EulerPhi(x)
2:      res ← x
3:
4:      if x mod 2 = 0 then                    ▷ We treat 2 separately, as it is the only even prime
5:          res ← res − res/2                  ▷ res ∗ (1 − 1/2)
6:          while x mod 2 = 0 do               ▷ While x still has a factor of 2
7:              x ← x/2                        ▷ Remove the factor of 2
8:          end while
9:      end if
10:
11:     p ← 3                                  ▷ Start at 3 and iterate up by odd numbers
12:     while p ∗ p ≤ x do
13:         if x mod p = 0 then
14:             res ← res − res/p              ▷ res ∗ (1 − 1/p)
15:             while x mod p = 0 do           ▷ While x still has a factor of p
16:                 x ← x/p                    ▷ Remove the factor of p
17:             end while
18:         end if
19:         p ← p + 2
20:     end while
21:
22:     if x > 1 then                          ▷ In the event a prime existed past the square root point
23:         res ← res − res/x
24:     end if
25:
26:     return res
27: end function
28:
29: function TotientSum(n)
30:     sum ← 0
31:
32:     for x = 1 to n do
33:         sum ← sum + EulerPhi(x)
34:     end for
35:
36:     return sum
37: end function
```

4.2 $O(n \log \log n)$ time algorithm

Instead of factorizing x into primes for each x, why not iterate over fixed primes directly?

For example, consider the expanded Euler product equations for $\phi(x)$ over all x in $\Phi(10)$:

$\phi(1) = 1$

$\phi(2) = 2\left(1 - \frac{1}{2}\right)$

$\phi(3) = 3\left(1 - \frac{1}{3}\right)$

$\phi(4) = 4\left(1 - \frac{1}{2}\right)$

$\phi(5) = 5\left(1 - \frac{1}{5}\right)$

$\phi(6) = 6\left(1 - \frac{1}{2}\right)\left(1 - \frac{1}{3}\right)$

$\phi(7) = 7\left(1 - \frac{1}{7}\right)$

$\phi(8) = 8\left(1 - \frac{1}{2}\right)$

$\phi(9) = 9\left(1 - \frac{1}{3}\right)$

$\phi(10) = 10\left(1 - \frac{1}{2}\right)\left(1 - \frac{1}{5}\right)$

Let's start with the prime $p = 2$. We know that 2 will be part of the prime factorizations for $x = 2$, 4, 6, 8, and 10, and therefore $\left(1 - \frac{1}{2}\right)$ will be present in the equations for those $\phi(x)$. Moving onto the next prime $p = 3$, we know that 3 will be part of the prime factorizations for $x = 3$, 6, and 9, and therefore $\left(1 - \frac{1}{3}\right)$ will be present in the equations for those $\phi(x)$, and so on and so forth.

This yields a type of *sieving* algorithm, a popular choice for Problem 351. It is easy to implement and relatively fast for the limit of $n = 10^8$. Furthermore, since this algorithm requires $O(n)$ memory, the array itself is only a few hundred megabytes in size.

Algorithm 4 Totient summatory function in $O(n \log \log n)$ time

```
1:  function TotientSum(n)
2:      tots ← [0, 1, 2, ..., n]                ▷ 0-indexed array containing values 0 to n
3:      res ← 1                                  ▷ φ(1) = 1
4:      for p = 2 to n do
5:          if p = tots[p] then                  ▷ p is prime, begin sieving
6:              k ← p
7:              while k ≤ n do
8:                  tots[k] ← tots[k] − tots[k]/p    ▷ Multiplying by (1 − 1/p)
9:                  k ← k + p
10:             end while
11:         end if
12:         res ← res + tots[p]
13:     end for
14:     return res
15: end function
```


Author: Marcus Andrews

5 Sublinear-time algorithms for $\Phi(n)$

5.1 Overview of numbers of form $\lfloor \frac{n}{g} \rfloor$

The heart of the sublinear algorithm actually requires us to revisit equation 3.2.2.

This expression possesses an input parameter of form $\lfloor \frac{n}{g} \rfloor$ for fixed n. It is worth taking a brief moment to examine this type of number.

Let $n = 100$, and consider the values of $\lfloor \frac{n}{g} \rfloor$ across $1 \leq g \leq n$. Notice the corresponding numbers, marked in color:

g	$\lfloor \frac{n}{g} \rfloor$
1	100
2	50
3	33
4	25
5	20
6	16
7	14
8	12
9	11
10	10
11	9
12	8
13	7
14	7
15	6
16	6
17	5
18	5
19	5
20	5
21	4
22	4
23	4
24	4
25	4
26	3
27	3
28	3
29	3
30	3
31	3
32	3
33	3

g	$\lfloor \frac{n}{g} \rfloor$
34	2
35	2
36	2
37	2
38	2
39	2
40	2
41	2
42	2
43	2
44	2
45	2
46	2
47	2
48	2
49	2
50	2
51	1
52	1
53	1
54	1
55	1
56	1
57	1
58	1
59	1
60	1
61	1
62	1
63	1
64	1
65	1
66	1

g	$\lfloor \frac{n}{g} \rfloor$
67	1
68	1
69	1
70	1
71	1
72	1
73	1
74	1
75	1
76	1
77	1
78	1
79	1
80	1
81	1
82	1
83	1
84	1
85	1
86	1
87	1
88	1
89	1
90	1
91	1
92	1
93	1
94	1
95	1
96	1
97	1
98	1
99	1
100	1

Once $g = \sqrt{n}$, the values of $\lfloor \frac{n}{g} \rfloor$ begin to iterate down in sequential order with many repeating values. These clusters of repetition become larger and larger in size for each new value of $\lfloor \frac{n}{g} \rfloor$ that appears.

More importantly, the values of g up to $\sqrt{n}$ are the same values of $\lfloor \frac{n}{g} \rfloor$ past $\sqrt{n}$! Thus, it is possible to describe everything about these clusters of repetition without even needing to iterate past $\sqrt{n}$.

Now we ask: given some $z \leq \sqrt{n}$, for which values of $g \geq \sqrt{n}$ will $\lfloor \frac{n}{g} \rfloor = z$? Due to the floor function, $z \leq \frac{n}{g} < z + 1$ must hold for $\lfloor \frac{n}{g} \rfloor = z$ to be true. Solve the constraints for g:

$$\left\lfloor \frac{n}{z} \right\rfloor \geq g \geq 1 + \left\lfloor \frac{n}{z+1} \right\rfloor \tag{5.1.1}$$

For example, let $z = 3$ (corresponding to orange on the previous page). Based on the constraints, $\lfloor \frac{n}{g} \rfloor = 3$ from $g = 1 + \lfloor \frac{100}{3+1} \rfloor = 26$ to $g = \lfloor \frac{100}{3} \rfloor = 33$, inclusive.

This is very good news when it comes to summations. Consider $\sum_{g=1}^{n} f(\lfloor \frac{n}{g} \rfloor)$. Then, for example, at step $g = 3$ we would calculate $f(\lfloor \frac{n}{3} \rfloor)$ as we normally would and add it to the running total. But since $\lfloor \frac{n}{g} \rfloor = 3$ for $(\lfloor \frac{n}{3} \rfloor - \lfloor \frac{n}{3+1} \rfloor)$ values of $g \geq \sqrt{n}$, we would *also* calculate $f(3)$, multiply it by $(\lfloor \frac{n}{3} \rfloor - \lfloor \frac{n}{3+1} \rfloor)$, and add that to the total as well. Of course, if $z = \lfloor \frac{n}{z} \rfloor$ exactly, then we must be sure not to add the same result twice.

In short, the properties of $\lfloor \frac{n}{g} \rfloor$ pave the way for sublinear evaluation. Instead of calculating $\lfloor \frac{n}{g} \rfloor$ over every single $g \leq n$, we only need to calculate $< 2\sqrt{n}$ distinct values in total.

5.2 $O(n^{3/4})$ time algorithm

We modify equation 3.2.2 by computing $v(\lfloor \frac{n}{g} \rfloor)$ normally for g up to $\lfloor \sqrt{n} \rfloor$, but then for the rest we use z up to $\lfloor \sqrt{n} \rfloor$ to count how many times $\lfloor \frac{n}{g} \rfloor = z$ for $g \geq \sqrt{n}$:

$$v(n) = \frac{n(n+1)}{2} - \left(\sum_{g=2}^{\lfloor \sqrt{n} \rfloor} v\left(\left\lfloor \frac{n}{g} \right\rfloor \right) \right) - \left(\sum_{z=1}^{\lfloor \sqrt{n} \rfloor} [z \neq \left\lfloor \frac{n}{z} \right\rfloor] \cdot \left(\left\lfloor \frac{n}{z} \right\rfloor - \left\lfloor \frac{n}{z+1} \right\rfloor \right) v(z) \right) \tag{5.2.1}$$

While this expression can be kept as-is and computed with memoization and recursion (effectively a type of dynamic programming), we will still go over the iterative form of this algorithm for better understanding. Algorithms for both versions will be provided later.

Focusing purely on the z-loop, we see that given some n, no matter how deep the recursion goes, $v(z)$ will always start at $v(1)$ and never go any higher than $v(\lfloor \sqrt{n} \rfloor)$ overall. This means we can use dynamic programming to compute $v(x)$ for $1 \leq x \leq \lfloor \sqrt{n} \rfloor$ and take care of all z-loops stemming from recursion on $v(n)$.

However, the g-loop is less intuitive. For some $v(x)$, some of the values of $\lfloor \frac{x}{g} \rfloor$ may be $\leq \lfloor \sqrt{n} \rfloor$, but others may not. For example, right from the very start of the recursion, when we call $v(n)$ we are

also calling $v(\lfloor \frac{n}{2} \rfloor), v(\lfloor \frac{n}{3} \rfloor), \cdots, v(\lfloor \frac{n}{\lfloor \sqrt{n} \rfloor} \rfloor)$. For large n, it is clear that we can't exactly use cached values of $v(x)$ for $x \leq \lfloor \sqrt{n} \rfloor$ for most of these.

Thus, we must also cache values of $v(\lfloor \frac{x}{g} \rfloor)$ for $x > \lfloor \sqrt{n} \rfloor$, but how do we know what to cache when there are theoretically so many possible values of x and g? It seems like it could turn into a huge mess. For large n, we call $v(\lfloor \frac{n}{2} \rfloor)$ from the first g-loop, and then in the g-loop of the next level of recursion we make function calls to $v(\lfloor \lfloor \frac{n}{2} \rfloor / 2 \rfloor)$, $v(\lfloor \lfloor \frac{n}{2} \rfloor / 3 \rfloor)$, $v(\lfloor \lfloor \frac{n}{2} \rfloor / 4 \rfloor)$, and so on and so forth for each of these, etc.

Fortunately, this is not a problem thanks to the identity $\lfloor \lfloor \frac{n}{a} \rfloor / b \rfloor = \lfloor \frac{n}{ab} \rfloor$. It doesn't matter how crazy the recursion gets with respect to the g-loop because any function parameter invoked will be equivalent to one of the values of $\lfloor \frac{n}{x} \rfloor$. No need to worry!

In other words, after we compute/cache every $v(x)$ for $1 \leq x \leq \lfloor \sqrt{n} \rfloor$, all we have to do is compute/cache $v(\lfloor \frac{n}{x} \rfloor)$ for $\lfloor \frac{n}{\lfloor \sqrt{n} \rfloor} \rfloor \geq x \geq 1$, ending with $v(\lfloor \frac{n}{1} \rfloor) = v(n)$, our final calculation.

In the following algorithms, ISQRT(x) is the integer (floor) square-root function, $\lfloor \sqrt{x} \rfloor$. We'll start with the recursive version and then examine the iterative version that we just explained.

Algorithm 5 Totient summatory function in $O(n^{3/4})$ time (recursive version)

```
1:  function V(n, cache)
2:      if cache[n] is defined then
3:          return cache[n]
4:      end if
5:
6:      res ← n * (n + 1)/2
7:
8:      for g = 2 to ISQRT(n) do
9:          res ← res − V(FLOOR(n/g), cache)
10:     end for
11:
12:     for z = 1 to ISQRT(n) do
13:         if FLOOR(n/z) ≠ z then
14:             res ← res − (FLOOR(n/z) − FLOOR(n/(z + 1))) * V(z, cache)
15:         end if
16:     end for
17:
18:     cache[n] ← res
19:     return res
20: end function
21:
22: function TOTIENTSUM(n)
23:     cache = dict()                                  ▷ Dictionary object
24:     return V(n, cache)
25: end function
```

Algorithm 6 Totient summatory function in $O(n^{3/4})$ time (iterative version)

```
1: function TotientSum(n)
2:     L ← isqrt(n)
3:     v ← [0] * (L + 1)                          ▷ 0-indexed array containing L + 1 values of 0
4:     bigV ← [0] * (floor(n/L) + 1)              ▷ bigV[m] will correspond to v(⌊n/m⌋)
5:
6:     for x = 1 to L do
7:         res ← x * (x + 1)/2
8:
9:         for g = 2 to isqrt(x) do
10:            res ← res − v[floor(x/g)]
11:        end for
12:
13:        for z = 1 to isqrt(x) do
14:            if z ≠ floor(x/z) then
15:                res ← res − (floor(x/z) − floor(x/(z + 1))) * v[z]
16:            end if
17:        end for
18:
19:        v[x] ← res
20:    end for
21:
22:    for x = floor(n/L) to 1 do
23:        k ← floor(n/x)
24:        res ← k * (k + 1)/2
25:
26:        for g = 2 to isqrt(k) do
27:            if floor(k/g) ≤ L then
28:                res ← res − v[floor(k/g)]
29:            else
30:                res ← res − bigV[x * g]
31:            end if
32:        end for
33:
34:        for z = 1 to isqrt(k) do
35:            if z ≠ floor(k/z) then
36:                res ← res − (floor(k/z) − floor(k/(z + 1))) * v[z]
37:            end if
38:        end for
39:
40:        bigV[x] ← res
41:    end for
42:
43:    return bigV[1]
44: end function
```

5.3 $O(n^{2/3}(\log\log n)^{1/3})$ time algorithm

This algorithm is basically the same as before, except we split at $L = O\left((n/(\log\log n))^{2/3}\right)$ instead of $L = \lfloor\sqrt{n}\rfloor$. The sieving strategy from section 4.2 is used for all values below L, and the sublinear algorithm is used for those above. The resulting time complexity will be $O(n^{2/3}(\log\log n)^{1/3})$, and this limit is achieved by mathematically determining the cutoff point at which the runtimes of the sieve and the recursion are asymptotically equivalent (see the Appendix for the runtime derivations).

In the following algorithms, ISQRT(x) is the integer (floor) square-root function, $\lfloor\sqrt{x}\rfloor$. We'll start with the recursive version and then examine the iterative version.

Algorithm 7 Totient summatory function in $O(n^{2/3}(n\log\log n)^{1/3})$ time (recursive version)

```
1:  function V(n, L, cache, sieve)
2:      if n ≤ L then
3:          return sieve[n]
4:      else if cache[n] is defined then
5:          return cache[n]
6:      end if
7:
8:      res ← n * (n + 1)/2
9:
10:     for g = 2 to ISQRT(n) do
11:         res ← res − V(FLOOR(n/g), L, cache, sieve)
12:     end for
13:
14:     for z = 1 to ISQRT(n) do
15:         if FLOOR(n/z) ≠ z then
16:             res ← res − (FLOOR(n/z) − FLOOR(n/(z + 1))) * V(z, L, cache, sieve)
17:         end if
18:     end for
19:
20:     cache[n] ← res
21:     return res
22: end function
23:
24: function TotientSum(n)
25:     L ← FLOOR((n/(LOG(LOG(n))))^(2/3))
26:     cache = dict()                                  ▷ Dictionary object
27:     sieve ← [0, 1, 2, ..., L]                       ▷ 0-indexed array containing values 0 to L
28:
29:     for p = 2 to L do
30:         if p = sieve[p] then
31:             k ← p
32:             while k ≤ L do
33:                 sieve[k] ← sieve[k] − sieve[k]/p
34:                 k ← k + p
35:             end while
36:         end if
37:         sieve[p] ← sieve[p] + sieve[p − 1]
38:     end for
39:
40:     return V(n, L, cache, sieve)
41: end function
```

Algorithm 8 Totient summatory function in $O(n^{2/3}(n \log\log n)^{1/3})$ time (iterative version)

```
1: function TotientSum(n)
2:     L ← floor((n/(log(log(n))))^(2/3))
3:     sieve ← [0, 1, 2, ..., L]                    ▷ 0-indexed array containing values 0 to L
4:     bigV ← [0] * (floor(n/L) + 1)                ▷ bigV[m] will correspond to v(⌊n/m⌋)
5:
6:     for p = 2 to L do
7:         if p = sieve[p] then
8:             k ← p
9:             while k ≤ L do
10:                sieve[k] ← sieve[k] − sieve[k]/p
11:                k ← k + p
12:            end while
13:        end if
14:        sieve[p] ← sieve[p] + sieve[p − 1]
15:    end for
16:
17:    for x = floor(n/L) to 1 do
18:        k ← floor(n/x)
19:        res ← k * (k + 1)/2
20:
21:        for g = 2 to isqrt(k) do
22:            if floor(k/g) ≤ L then
23:                res ← res − sieve[floor(k/g)]
24:            else
25:                res ← res − bigV[x * g]
26:            end if
27:        end for
28:
29:        for z = 1 to isqrt(k) do
30:            if z ≠ floor(k/z) then
31:                res ← res − (floor(k/z) − floor(k/(z + 1))) * sieve[z]
32:            end if
33:        end for
34:
35:        bigV[x] ← res
36:    end for
37:
38:    return bigV[1]
39: end function
```

6 Moebius transformations

In this section, we will consider another strategy that will give us a different expression for $\Phi(n)$ that can be evaluated using the sublinear algorithm.

6.1 Deriving the Moebius function

Taking a step back, consider our derivation of the Euler product formula, equation 4.1.1. Using inclusion-exclusion, we started with x and subtracted the counts of divisors containing ≥ 1 prime ($\frac{x}{p}$ for prime p), added back the counts of divisors containing ≥ 2 primes ($\frac{x}{pq}$ for primes p and q), subtracted back out the counts of divisors containing ≥ 3 primes ($\frac{x}{pqr}$ for primes p, q, r), etc.

If the denominators had an odd number of primes, the terms were subtracted, but if they had an even number of primes, they were added. These denominators span all possible combinations of 1-power prime factors from the factorization of x. These are called *squarefree* numbers, as they cannot be divided by a square number k^2 (except $k = 1$). We might try restating Euler's product formula by iterating over the divisors of x:

$$\phi(x) = \sum_{d|x} \mu(d) \left(\frac{x}{d}\right) \tag{6.1.1}$$

where $\mu(d) = 1$ if d is squarefree and has an even number of primes, $\mu(d) = -1$ if d is squarefree and has an odd number of primes, and $\mu(d) = 0$ if d is not squarefree. $\mu(d)$ is better known as the *Moebius function*.

6.2 Moebius inversion

Using our new Moebius-based form of the totient function, we rewrite $v(n)$ as follows:

$$v(n) = \sum_{x=1}^{n} \sum_{d|x} \mu(d) \left(\frac{x}{d}\right) \tag{6.2.1}$$

Can this be simplified? Using similar strategies as before, let us try to iterate over each fixed divisor d instead of computing the divisors at each step of x.

Given a fixed divisor d, for how many values of x, for $1 \leq x \leq n$, will d be a divisor? Exactly $\lfloor \frac{n}{d} \rfloor$ values of x: $(d, 2d, 3d, ..., \lfloor \frac{n}{d} \rfloor d)$.

Furthermore, every time that divisor d is encountered in the above expression, the amount $\mu(d) \left(\frac{x}{d}\right)$ is added to the total. But we already know which values of x we will encounter for each d (we just listed them!), and thus:

$$v(n) = \sum_{d=1}^{n} \sum_{k=1}^{\lfloor \frac{n}{d} \rfloor} \mu(d) \left(\frac{kd}{d}\right) = \sum_{d=1}^{n} \mu(d) \frac{\lfloor \frac{n}{d} \rfloor (\lfloor \frac{n}{d} \rfloor + 1)}{2} = \sum_{d=1}^{n} \mu(d) t \left(\left\lfloor \frac{n}{d} \right\rfloor\right) \tag{6.2.2}$$

This is no coincidence: A generalization of *Moebius inversion* allows us to relate $t(n)$ and $v(n)$ as transformations of each other:

$$t(n) = \sum_{g=1}^{n} v\left(\left\lfloor \frac{n}{g} \right\rfloor\right) \qquad v(n) = \sum_{d=1}^{n} \mu(d) t\left(\left\lfloor \frac{n}{d} \right\rfloor\right) \tag{6.2.3}$$

where $t(n) = \frac{n(n+1)}{2}$. We have also found another way to arrive at equation 3.2.1. In fact, this inversion relationship will hold for any arithmetic function[3] $t(n)$. Note that the new equation for $v(n)$ contains $\lfloor \frac{n}{d} \rfloor$, which is exactly the type of parameter we need for our sublinear algorithm.

6.3 The Mertens function

Equation 6.2.2 can be approached with similar sublinear logic as before, but there is a minor difference: the Moebius function is present in the equation. For fixed $z \leq \sqrt{n}$, even though $\lfloor \frac{n}{d} \rfloor = z$ over many $d \geq \lfloor \sqrt{n} \rfloor$, the values of $\mu(d)$ won't be. Therefore we cannot simply multiply the repeated value of $t(z)$ by $(\lfloor \frac{n}{z} \rfloor - \lfloor \frac{n}{z+1} \rfloor)$, but rather by the *sum* of $\mu(d)$ over that range. The sum of the Moebius function is called the *Mertens function*:

$$M(n) = \sum_{k=1}^{n} \mu(k) \tag{6.3.1}$$

Thus, when we convert equation 6.2.2 to sublinear form, we have the following expression:

$$v(n) = \left(\sum_{d=1}^{\lfloor \sqrt{n} \rfloor} \mu(d) t\left(\left\lfloor \frac{n}{d} \right\rfloor\right)\right) + \left(\sum_{z=1}^{\lfloor \sqrt{n} \rfloor} [z \neq \left\lfloor \frac{n}{z} \right\rfloor] \cdot \left(M\left(\left\lfloor \frac{n}{z} \right\rfloor\right) - M\left(\left\lfloor \frac{n}{z+1} \right\rfloor\right)\right) t(z)\right) \tag{6.3.2}$$

Using Moebius inversion on $M(n)$ lets us transform it into the form we need:

$$1 = \sum_{g=1}^{n} M\left(\left\lfloor \frac{n}{g} \right\rfloor\right) \tag{6.3.3}$$

Now we separate the $g = 1$ piece, solve for $M(n)$, and convert it to sublinear form:

$$M(n) = 1 - \left(\sum_{g=2}^{\lfloor \sqrt{n} \rfloor} M\left(\left\lfloor \frac{n}{g} \right\rfloor\right)\right) - \left(\sum_{z=1}^{\lfloor \sqrt{n} \rfloor} [z \neq \left\lfloor \frac{n}{z} \right\rfloor] \cdot \left(\left\lfloor \frac{n}{z} \right\rfloor - \left\lfloor \frac{n}{z+1} \right\rfloor\right) M(z)\right) \tag{6.3.4}$$

We can also sieve the first $L = O\left((n/(\log\log n))^{2/3}\right)$ values of the Mertens function, which requires sieving the Moebius function, in turn.

[3] More formally, an *arithmetic function* is a complex-valued function $f(n)$ defined for all $n \in \mathbb{N}$ so that $f : \mathbb{N} \to \mathbb{C}$.

6.4 Moebius sieving

We can use a sieve to calculate the Moebius function over a range of n in $O(n \log \log n)$ time.

The idea is fairly straightforward: starting with an array of values initialized to 1, we then take each prime p and sieve the entire range, flipping each element encountered by -1. At the end, all elements corresponding to numbers with an odd number of distinct prime factors will be -1 (or 1 otherwise). However, not all of these numbers are squarefree, so we also sieve p^2 for each prime and set all elements encountered to 0.

Algorithm 9 Moebius sieve

```
1:  function MoebiusSieve(n)
2:      P ← primeSieve(n + 1)                    ▷ returns all primes below limit
3:      Mu = [1] * (n + 1)                       ▷ 0-indexed array containing n + 1 values of 1
4:      Mu[0] ← 0
5:
6:      for p in P do
7:          q ← p
8:          while q ≤ n do
9:              Mu[q] ← Mu[q] * (−1)             ▷ Flip Mu[q] by (−1) every time a prime makes contact
10:             q ← q + p
11:         end while
12:
13:         q ← p * p
14:         while q ≤ n do
15:             Mu[q] ← 0                        ▷ Mu[q] is 0 if q is not squarefree
16:             q ← q + p * p
17:         end while
18:     end for
19:
20:     return Mu
21: end function
```

7 Closing Remarks / Recap

Problem 351 is a fantastic place to start when it comes to practicing the problem-solving strategies behind the sublinear algorithm. Once you understand $\Phi(n)$, it will become easier to take on more difficult problems with functions of greater complexity.

Let's recap some of the major tips/strategies:

- If we can rearrange an expression to involve input parameters of form $\lfloor \frac{n}{k} \rfloor$, then we may be able to apply the sublinear method.
- Instead of computing divisors/primes/gcds/etc at each step of a summation, try iterating over fixed values. Whenever we ask how many times a fixed value of k can fit under a limit n, we immediately begin dealing with $\lfloor \frac{n}{k} \rfloor$-type numbers.
- Inclusion-exclusion is a powerful tool. Rather than try to figure out how many times a specific condition is met, it may be easier to start with the total (without the condition) and subtract the counts where the condition is *not* met. If there are multiple conditions at play, then each one may have to be dealt with separately. However, this can also lead to a nice generalized formula or recursion.
- If all else fails, try a sieving strategy if it makes sense to do so. It's also a great way to precache values to speed up a separate process.
- When dealing with a particular type of number (in our case, $\lfloor \frac{n}{k} \rfloor$), it helps to output the values over a medium-scale range. There may be a pattern or relationship that can be leveraged to cut down the runtime.
- Any time we find ourselves looking into products of distinct one-power primes, we have square-free numbers, which means the Moebius function may be of use.
- Moebius inversion can convert ordinary arithmetic functions into something with $\lfloor \frac{n}{k} \rfloor$-type numbers and vice-versa. If anything, it is worth investigating simply because it introduces new relationships and insights that may have been harder to see before.
- Insights from other Project Euler problems are often useful. The various concepts that we relied on in this paper (inclusion-exclusion, sieving, iterating over fixed values, totient functions, dynamic programming, recursion, memoization, prime factorization, greatest common divisors, etc) have all been introduced in easier problems.
- Don't be afraid to take a step back. Notice that while we were able to logically push our way to the $O(n \log \log n)$ algorithm from scratch, we actually derived the sublinear method by examining an interesting attribute of our slower $O(n^2)$ algorithm! We also arrived at the sublinear method through Moebius inversion, which we in turn derived by taking a step back to re-examine the Euler product formula used in the $O(n^{3/2})$ algorithm. Slower algorithms tend to be less fundamentally complex, which makes them easier to investigate and modify.

Good luck, and happy solving!

8 Appendix

8.1 Timings

Quadratic algorithms

n	$\Phi(n)$	$O(n^2 \log n)$	$O(n^2)$
10^3	304192	0.0537 seconds	0.00465 seconds
10^4	30397486	6.02 seconds	0.452 seconds
10^5	3039650754	$>$ 10 minutes	$>$ 1.5 minutes
10^6	303963552392	$>$ 1 day	$>$ 3 hours
10^7	30396356427242	$>$ 3.5 months	$>$ 2 weeks
10^8	3039635516365908	$>$ 33 years	$>$ 3.5 years
10^9	303963551173008414	$>$ 3.8 millennia	$>$ 3.5 centuries
10^{10}	30396355092886216366	$>$ 440 millennia	$>$ 35 millennia
10^{11}	303963550928338621140	$>$ 50 million years	$>$ 3.5 million years
10^{12}	3039635509270598040259 10	$>$ 6 billion years	$>$ 350 million years

Subquadratic algorithms

n	$\Phi(n)$	$O(n^{3/2})$	$O(n \log \log n)$
10^3	304192	0.000291 seconds	0.0000457 seconds
10^4	30397486	0.00528 seconds	0.000407 seconds
10^5	3039650754	0.111 seconds	0.00418 seconds
10^6	303963552392	2.27 seconds	0.0486 seconds
10^7	30396356427242	37.9 seconds	0.665 seconds
10^8	3039635516365908	$>$ 10 minutes	6.98 seconds
10^9	303963551173008414	$>$ 4 hours	$>$ 1 minute
10^{10}	30396355092886216366	$>$ 3 days	$>$ 15 minutes
10^{11}	303963550928338621140	$>$ 2 months	$>$ 2.5 hours
10^{12}	3039635509270598040259 10	$>$ 3 years	$>$ 1 day

Sublinear algorithms

n	$\Phi(n)$	$O(n^{3/4})$	$O(n^{2/3}(\log \log n)^{1/3})$
10^3	304192	0.000318 seconds	0.0000432 seconds
10^4	30397486	0.00104 seconds	0.000278 seconds
10^5	3039650754	0.00567 seconds	0.000751 seconds
10^6	303963552392	0.0361 seconds	0.00309 seconds
10^7	30396356427242	0.242 seconds	0.0123 seconds
10^8	3039635516365908	1.49 seconds	0.061 seconds
10^9	303963551173008414	9.50 seconds	0.306 seconds
10^{10}	30396355092886216366	44.9 seconds	1.41 seconds
10^{11}	303963550928338621140	$>$ 4 minutes	6.22 seconds
10^{12}	3039635509270598040259 10	$>$ 20 minutes	29.5 seconds

The algorithms were written in C++ (GCC 4.7.2) and tested on a Windows 7 laptop with 16 GB RAM and an Intel(R) Core(TM) i7-3630QM CPU @ 2.40 GHz. All times above 30 minutes are estimations. The sublinear algorithms used were the recursive versions.

For Problem 351, it is clear that quadratic algorithms are not viable, as they would take years to finish at the $n = 10^8$ level. However, the next-best approach, the $O(n^{3/2})$ algorithm, can cut things down to a matter of minutes.

The $O(n \log \log n)$ sieve approach is a decent option as well, capable of finishing Problem 351 in a time under 10 seconds (easily meeting the Project Euler minute-rule). The $O(n^{2/3}(\log \log n)^{1/3})$ algorithm can compute the result in well under a second.

Looking to larger values of n, we see that even the subquadratic algorithms aren't unreasonable. It is not necessarily out of the question to wait a few days for the $O(n^{3/2})$ algorithm to finish computing at the $n = 10^{10}$ level, but at 10^{11} and beyond it becomes a lot less practical. If you have the memory for it, the $O(n \log \log n)$ algorithm is a superior alternative, but if memory is a concern, there is always the segmented sieve approach (which we did not cover).

The sublinear algorithms are the best of both worlds. They use less memory and finish much more quickly. For $n = 10^{12}$, the $O(n^{2/3}(\log \log n)^{1/3})$ algorithm can do in under 30 seconds what would take the $O(n^2 \log n)$ algorithm nearly half the age of the universe to compute. It pays to spend some time figuring out the better algorithms!

8.2 Runtime derivations

These derivations are not meant to be fully-rigorous proofs, but rather proof-sketches that should be sufficient in getting the general ideas across. All operations are assumed to run in $O(1)$ time (e.g. addition, subtraction, multiplication, division, taking the exponent/logarithm/modulus, etc).

For positive runtime functions f and g on real domains, define $f(n) \sim g(n)$ as follows:

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 1 \tag{8.2.1}$$

This means that if $f(n) \sim g(n)$, then the two runtime functions are asymptotically equivalent and therefore have the same time-complexity.

Going forward, $f(n)$ will represent the estimated runtime of function $v(n)$.

$O(n^2 \log n)$ algorithm

The first thing we should do is determine the time-complexity of the Euclidean algorithm for two integers x and y where $x \geq y$. It is sufficient to show that after two steps, y is reduced to under half of its original value, implying that the number of steps is bounded from above by $O(\log y)$.

Let x_i and y_i be the values of x and y after the ith step of the Euclidean algorithm. Thus, $x_i = y_{i-1}$ and $y_i = (x_{i-1} \bmod y_{i-1})$.

As we know, $(a \bmod b)$, the remainder of a/b, is always less than b by definition. Our algorithm always begins with $x_0 \geq y_0$. Now, if $y_0 \leq \frac{x_0}{2}$, then since $y_1 = (x_0 \bmod y_0) < y_0$, we have $y_1 < \frac{x_0}{2}$. If $y_0 > \frac{x_0}{2}$ instead, then $y_1 = (x_0 \bmod y_0) = x_0 - y_0 < \frac{x_0}{2}$ as well.

And since $x_i \geq y_i$ for all i, this analysis will hold over all i. After the first step, $x_1 = y_0$ and $y_1 < \frac{x_0}{2}$. After the second step, we have $y_2 < \frac{x_1}{2}$. Plugging in y_0, this also means $y_2 < \frac{y_0}{2}$. In other words, after two steps, y drops to under half its original value. Thus, we can characterize $\gcd(x, y)$ as running in $O(\log y)$ time, assuming constant-time operations.

To estimate the runtime of the entire algorithm:

$$f(n) = \sum_{x=1}^{n} \sum_{y=1}^{x} \log y \tag{8.2.2}$$

The well-known log sum formula $\log_b j + \log_b k = \log_b jk$ tells us that the inner loop runs in $O(\log(1 \cdot 2 \cdot 3 \cdot ... \cdot x)) = O(\log x!)$ time. Since $x! \leq x^x$, the inner loop also runs in $O(\log x^x)$ or $O(x \log x)$ time.

$$\begin{aligned} f(n) = \sum_{x=1}^{n} \log x! \leq \sum_{x=1}^{n} \log x^x = \sum_{x=1}^{n} x \log x \leq \sum_{x=1}^{n} n \log n = n^2 \log n \\ \implies f \in O(n^2 \log n) \end{aligned} \tag{8.2.3}$$

$O(n^2)$ algorithm

This algorithm is fairly straightforward, as we are simply iterating over two loops and re-using cached results at each step.

To estimate the runtime of the entire algorithm:

$$\begin{aligned} f(n) = \sum_{x=1}^{n} \sum_{g=2}^{x} (1) = \sum_{x=1}^{n} (x-1) = \frac{n^2 - n}{2} \\ \implies f \in O(n^2) \end{aligned} \tag{8.2.4}$$

$O(n^{3/2})$ algorithm

At each step x for $1 \leq x \leq n$, we calculate the prime factorization of x using standard trial division methods. The general idea behind trial division is that we take some prime p and divide x by it repeatedly until it cannot be divided any more by that factor. Then we increase p and continue the process. The process ends once $p^2 > x$, which implies an upper bound of $O(\sqrt{x})$ runtime for the factorization process.

To estimate the runtime of the entire algorithm:

$$f(n) = \sum_{x=1}^{n} \sqrt{x} \tag{8.2.5}$$

The summation can be approximated with an integral:

$$\int \sqrt{x}\, dx = \frac{2x^{3/2}}{3} + C \tag{8.2.6}$$

Finishing up the original estimation:

$$\begin{aligned} f(n) &\sim \frac{2n^{3/2}}{3} + C \\ \implies f &\in O(n^{3/2}) \end{aligned} \tag{8.2.7}$$

$O(n \log \log n)$ algorithm

At each step p for $2 \leq p \leq n$, we see that if p is prime, then we sieve the rest of the array stepwise p, which implies $\frac{n}{p}$ steps per prime encountered.

To estimate the runtime of the entire algorithm:

$$f(n) = \sum_{p=2}^{n} [p \in \texttt{PRIMES}] \cdot \frac{n}{p} \tag{8.2.8}$$

According to the prime number theorem, if we let $\pi(n)$ be the count of prime numbers less than or equal to n, then $\pi(n) \sim \frac{n}{\log n}$. Furthermore, for the kth prime p_k we have $p_k \sim k \log k$. With these two approximations, the runtime is restated as follows:

$$f(n) \sim n \sum_{x=1}^{n/\log n} \frac{1}{x \log x} \tag{8.2.9}$$

The summation can be approximated with an integral:

$$\int \frac{1}{x \log x}\, dx = \log\log x + C \tag{8.2.10}$$

Finishing up the original estimation:

$$\begin{aligned} f(n) \sim n(\log\log(n/\log n) + C) \sim n(\log(\log n - \log\log n) + C) \leq n\log\log n + Cn \\ \implies f \in O(n\log\log n) \end{aligned} \tag{8.2.11}$$

$O(n^{3/4})$ algorithm

The runtime is easier to understand if we use the iterative version as our guide.

In the first main loop, which iterates x from 1 to $\lfloor\sqrt{n}\rfloor$, we have a g-loop iterating from 2 to $\lfloor\sqrt{x}\rfloor$, as well as a z-loop iterating from 1 to $\lfloor\sqrt{x}\rfloor$.

In the second main loop, which iterates x from $\lfloor\frac{n}{\lfloor\sqrt{n}\rfloor}\rfloor$ down to 1, we have a g-loop iterating from 2 to $\lfloor\sqrt{\lfloor\frac{n}{x}\rfloor}\rfloor$, as well as a z-loop iterating from 1 to $\lfloor\sqrt{\lfloor\frac{n}{x}\rfloor}\rfloor$.

To estimate the runtime of the entire algorithm:

$$f(n) = \left(\sum_{x=1}^{\sqrt{n}}\left(\sum_{g=2}^{\sqrt{x}}(1) + \sum_{z=1}^{\sqrt{x}}(1)\right)\right) + \left(\sum_{x=1}^{\frac{n}{\sqrt{n}}}\left(\sum_{g=2}^{\sqrt{\frac{n}{x}}}(1) + \sum_{z=1}^{\sqrt{\frac{n}{x}}}(1)\right)\right) \tag{8.2.12}$$

Simplifying/approximating:

$$f(n) \approx 2\sum_{x=1}^{\sqrt{n}}\sqrt{x} + 2\sum_{x=1}^{\sqrt{n}}\sqrt{\frac{n}{x}} \tag{8.2.13}$$

The summations can be approximated with integrals:

$$\int \sqrt{x}\, dx = \frac{2x^{3/2}}{3} + C_1 \tag{8.2.14}$$

$$\int \sqrt{\frac{n}{x}}\, dx = 2x\sqrt{\frac{n}{x}} + C_2 \tag{8.2.15}$$

Finishing up the original estimation:

$$\begin{aligned} f(n) \sim 2\left(\frac{2(\sqrt{n})^{3/2}}{3} + C_1\right) + 2\left(2\sqrt{n}\sqrt{\frac{n}{\sqrt{n}}} + C_2\right) \sim \frac{16n^{3/4}}{3} + 2(C_1 + C_2) \\ \implies f \in O(n^{3/4}) \end{aligned} \tag{8.2.16}$$

$O(n^{2/3}(\log\log n)^{1/3})$ algorithm

This algorithm is very similar to the last one in overall structure, except that we first run a sieve to precalculate the first L values. Then the sieve runs in $O(L\log\log L)$ time (as per our previous derivation for the sieving algorithm). Let's solve for L and then derive the time complexity. The idea is to set L such that the runtime complexity of the sieving process is asymptotically equivalent to the runtime complexity of the recursive process.

We no longer need the first main loop from the $O(n^{3/4})$ algorithm. Only the second main loop is necessary, and it begins at $x = \lfloor \frac{n}{L} \rfloor$ and works its way down to 1. Let $f(n) = s(n) + r(n)$, the runtime of the sieve and the recursive process, respectively, where $s(n) \sim L\log\log L$.

To estimate the runtime of $r(n)$:

$$r(n) = \sum_{x=1}^{\frac{n}{L}} \left(\sum_{g=2}^{\sqrt{\frac{n}{x}}} (1) + \sum_{z=1}^{\sqrt{\frac{n}{x}}} (1) \right) \approx 2 \sum_{x=1}^{\frac{n}{L}} \sqrt{\frac{n}{x}} \tag{8.2.17}$$

Using the integral approximation from 8.2.15, we have:

$$r(n) \sim \frac{2n}{L}\sqrt{\frac{n}{\frac{n}{L}}} + C \sim \frac{2n}{\sqrt{L}} + C \tag{8.2.18}$$

Now, we set the sieving runtime equal to that of the recursion to determine the sieve limit L:

$$s(n) \sim L\log\log L \leq L\log\log n = \frac{2n}{\sqrt{L}} + C \tag{8.2.19}$$

Solving for L (ignoring the constant C):

$$L \approx 2\left(n/(\log\log n)\right)^{2/3} \tag{8.2.20}$$

Plugging L into the total runtime estimation $f(n) = s(n) + r(n)$ and simplifying:

$$\begin{aligned} f(n) &\approx 3n^{2/3}(\log\log n)^{1/3} + C \\ \implies f &\in O(n^{2/3}(\log\log n)^{1/3}) \end{aligned} \tag{8.2.21}$$

A quick note on the splitting-point L. You may find that, in practice, you get faster runtimes using something like $L = O(n^{2/3})$. When using that larger L, the resulting time complexity increases to $O(n^{2/3} \log\log n)$.

That may sound confusing at first. How can a greater time-complexity result in faster runtimes? In O-notation, time-complexities are asymptotic, which means that they describe what happens as n goes to infinity. At infinity, all sorts of constants and lesser-powers get hidden since they effectively go to 0 as the leading terms take over.

However, at levels such as $n = 10^{12}$, we see that $\log\log n$ is very small and hasn't yet grown large enough to overtake the constant factors, which still have a nontrivial influence over the runtimes. This is why it isn't uncommon to hear that this algorithm has a time-complexity of $O(n^{2/3})$. It really doesn't, but at our particular levels of n, the $\log\log n$ factor is too small to matter much, and so some find it easier to omit it altogether.

As a consequence, it can be prudent to use a cutoff point that more accurately balances the times out for intended levels of usage. We could also use a binary search routine to determine the optimal value for L that minimizes the overall runtime on average.