Sieve of Eratosthenes - Wikipedia, the free encyclopedia

Sieve of Eratosthenes: algorithm steps for primes below 121 (including optimization of starting from prime's square).

In mathematics, the **sieve of Eratosthenes** (Greek: κόσκινον Ἐρατοσθένους), one of a number of prime number sieves, is a simple, ancient algorithm for finding all prime numbers up to any given limit. It does so by iteratively marking as composite (i.e. not prime) the multiples of each prime, starting with the multiples of 2.[1]

The multiples of a given prime are generated starting from that prime, as a sequence of numbers with the same difference, equal to that prime, between consecutive numbers.[1] This is the sieve's key distinction from using trial division to sequentially test each candidate number for divisibility by each prime.[2]

The sieve of Eratosthenes is one of the most efficient ways to find all of the smaller primes (below 10 million or so).[3] It is named after Eratosthenes of Cyrene, a Greek mathematician; although none of his works have survived, the sieve was described and attributed to Eratosthenes in the *Introduction to Arithmetic* by Nicomachus.[4]

Algorithm description[edit]

Sift the Two's and Sift the Three's,
The Sieve of Eratosthenes.
When the multiples sublime,
The numbers that remain are Prime.

A prime number is a natural number which has exactly two distinct natural number divisors: 1 and itself.

To find all the prime numbers less than or equal to a given integer n by Eratosthenes' method:

1. Create a list of consecutive integers from 2 to n: (2, 3, 4, ..., n).
2. Initially, let p equal 2, the first prime number.
3. Starting from p, count up in increments of p and mark each of these numbers greater than p itself in the list. These will be multiples of p: $2p$, $3p$, $4p$, etc.; note that some of them may have already been marked.
4. Find the first number greater than p in the list that is not marked. If there was no such number, stop. Otherwise, let p now equal this number (which is the next prime), and repeat from step 3.

When the algorithm terminates, all the numbers in the list that are not marked are prime.

The main idea here is that every value for p is prime, because we have already marked all the multiples of the numbers less than p.

As a refinement, it is sufficient to mark the numbers in step 3 starting from p^2, as all the smaller multiples of p will have already been marked at that point. This means that the algorithm is allowed to terminate in step 4

when p^2 is greater than n.[1]

Another refinement is to initially list odd numbers only, (3, 5, ..., n), and count up using an increment of $2p$ in step 3, thus marking only odd multiples of p greater than p itself. This actually appears in the original algorithm.[1] This can be generalized with wheel factorization, forming the initial list only from numbers coprime with the first few primes and not just from odds, i.e. numbers coprime with *2*.[8]

Incremental sieve[edit]

An incremental formulation of the sieve[2] generates primes indefinitely (i.e. without an upper bound) by interleaving the generation of primes with the generation of their multiples (so that primes can be found in gaps between the multiples), where the multiples of each prime p are generated directly, by counting up from the square of the prime in increments of p (or $2p$ for odd primes).

Trial division[edit]

Trial division can be used to produce primes by filtering out the composites found by testing each candidate number for divisibility by its preceding primes. It is often confused with the sieve of Eratosthenes, although the latter directly generates the composites instead of testing for them. Trial division has worse theoretical complexity than that of the sieve of Eratosthenes in generating ranges of primes.[2]

When testing each candidate number, the optimal trial division algorithm uses just those prime numbers not exceeding its square root. The widely known 1975 functional code by David Turner[9] is often presented as an example of the sieve of Eratosthenes[8] but is actually a sub-optimal trial division algorithm.[2]

Example[edit]

To find all the prime numbers less than or equal to 30, proceed as follows.

First generate a list of integers from 2 to 30:

2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30

First number in the list is 2; cross out every 2nd number in the list after it (by counting up in increments of 2), i.e. all the multiples of 2:

2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ 9 ~~10~~ 11 ~~12~~ 13 ~~14~~ 15 ~~16~~ 17 ~~18~~ 19 ~~20~~ 21 ~~22~~ 23 ~~24~~ 25 ~~26~~ 27 ~~28~~ 29 ~~30~~

Next number in the list after 2 is 3; cross out every 3rd number in the list after it (by counting up in increments of 3), i.e. all the multiples of 3:

2 3 ~~4~~ 5 ~~6~~ 7 ~~8 9 10~~ 11 ~~12~~ 13 ~~14 15 16~~ 17 ~~18~~ 19 ~~20 21 22~~ 23 ~~24~~ 25 ~~26 27 28~~ 29 ~~30~~

Next number not yet crossed out in the list after 3 is 5; cross out every 5th number in the list after it (by counting up in increments of 5), i.e. all the multiples of 5:

2 3 ~~4~~ 5 ~~6~~ 7 ~~8 9 10~~ 11 ~~12~~ 13 ~~14 15 16~~ 17 ~~18~~ 19 ~~20 21 22~~ 23 ~~24 25 26 27 28~~ 29 ~~30~~

Next number not yet crossed out in the list after 5 is 7; the next step would be to cross out every 7th number in the list after it, but they are all already crossed out at this point, as these numbers (14, 21, 28) are also

multiples of smaller primes because 7*7 is greater than 30. The numbers left not crossed out in the list at this point are all the prime numbers below 30:

```
 2  3     5     7           11    13          17    19          23                  29
```

Algorithm complexity[edit]

Time complexity in the random access machine model is $O(n \log \log n)$ operations, a direct consequence of the fact that the prime harmonic series asymptotically approaches $\log \log n$.

The bit complexity of the algorithm is $O(n(\log n)(\log \log n))$ bit operations with a memory requirement of $O(n)$.[10]

The segmented version of the sieve of Eratosthenes, with basic optimizations, uses $O(n)$ operations and $O(n^{1/2} \log \log n / \log n)$ bits of memory.[11]

Implementation[edit]

In pseudocode:[12][13]

```
Input: an integer n > 1

Let A be an array of Boolean values, indexed by integers 2 to n,
initially all set to true.

for i = 2, 3, 4, ..., √n :
  if A[i] is true:
    for j = i^2, i^2+i, i^2+2i, ..., n:
      A[j] := false

Now all i such that A[i] is true are prime.
```