Check if a given number is prime

One of the simplest approaches is to check whether the number N is divisible by any number from 2 to N-1, as from the definition of a prime number it's clear that the prime number is only divisible by 1 and itself, so if it is divisible by any number from {2, N-1}, then the number is not prime.

Pseudocode:

```
/* Input n is a non-negative integer, checks whether n is prime or not */
function isPrime(n)

        if n equals 0 or 1
                return false

        //   Iterate from 2 to n-1 and check the divisibility of n.
        for i = 2 to n-1
                //   If n becomes divisible by i, then return false
                if n mod i equals 0
                        return false

        return true
```

Time complexity: O(n), where n is the number that is to be checked for primality.

However, it turns out that it is not necessary to check the divisibility of the number N from 2 to all the way to N-1, rather it is sufficient to check its divisibility from 2 to sqrt(N).

Suppose N is a composite number, then N can be written as N = a * b, where 2 <= a,b <= N-1 as a composite number has at least one factor other than one and itself. Then let one factor be x, the other factor can simply be N/x.

Now, we can claim that at least one of a and b cannot be greater than sqrt(N), as if a > sqrt(N) and b > sqrt(N), then a * b > N, which is a contradiction. Hence, we can say that for a number to be composite it must have a factor less than or equal to sqrt(N).

Therefore we can modify our range of {2, N-1} to {2, sqrt(N)}, to check if the given number is prime.

Pseudocode:

```
/* Input n is a non-negative integer, checks whether n is prime or not */
```

```
function isPrime(n)

        if n equals 0 or 1
                return false

        //   Iterate from 2 to sqrt(n) and check the divisibility of n.
        for i = 2 ; i * i <= n ; i++
                //   If n becomes divisible by i, then return false
                if n mod i equals 0
                        return false

        return true
```

Time complexity: O(sqrt(n)), where n is the number that is to be checked for primality.

Find primes from 1 to n

Using the above approach to check whether a given number is prime, we can easily find all prime numbers from 1 to n, by iterating over the values from 1 to n and checking for its primality.

Pseudocode:

```
/* Input n is a positive integer, checks whether n is prime or not */
function isPrime(n)

        if n equals 1
                return false

        //   Iterate from 2 to sqrt(n) and check the divisibility of n.
        for i = 2 ; i * i <=n ; i++
                //   If n becomes divisible by i, then return false
                if n mod i equals 0
                        return false

        return true

/* Input n is a positive integer, finds all primes from 1 to n */
function findPrimes(n)

        //   Iterate over the values from 1 to n and check for its primality
        for idx = 1 to n
```

```
//   Check if the current number 'idx' is prime
if isPrime(idx)
        print(idx)
```

Time complexity: O(n * sqrt(n)), where n is the number up to which primes are to be found.

However this approach is not efficient, we can use **Sieve of Eratosthenes** which finds all primes from 1 to n in $O(n * \log_2(\log_2 n))$ time complexity and O(n) space complexity.