Segmented Sieve

Segmented Sieve, similar to simple sieve, is also used to find all prime numbers from 1 to n or to find all the primes in a given range.

Why segmented sieve?

- Segmented sieve is an optimization in the usage of memory over simple sieve which makes it useful for finding primes from 1 to n efficiently as compared to simple sieve.
- It provides a better locality of reference which in turn allows better caching by the CPU.

Space optimization in segmented sieve

The idea of segmented sieve is to basically divide the whole range from 1 to n, into smaller segments(blocks) and find all the prime numbers in those segments i.e sieve each segment(block) separately. The division of the whole range into segments(blocks) will allow us to not keep the whole range in memory at once, instead keep one segment(block) in memory at a time which in turn allows better caching by the CPU.

Division into segments

Similar to simple sieve**,** we need primes up to sqrt(n) only for the complete process of sieving. The difference lies in the process of sieving.

Let 'b' be the size of each segment(block), then the total number of segments(blocks) required will be ceil(n/b). For each block index 'i' (0,1,2....,floor(n/b)), denoted by b_i, the numbers present in the block b_i are in the range [i*b, i*b+b-1]. Generally, the size of the block is taken to be sqrt(n).

Hence, sieving each such segment(block) means going through all the prime numbers from 1 to sqrt(n) and perform a simple sieve on this segment(block) i.e for each prime 'p'(unmarked number), mark all its multiple within the range of the current segment(block).

Algorithm:

- Find all prime numbers from 1 to sqrt(n), using a simple sieve.
- Divide the whole range(1 to n), into segments(blocks) of size sqrt(n).
- For every segment, perform a sieve -
    - Let b_i be the current block, let 'left' denote the starting number and 'right' denote the ending number in b_i
    - Create an array of size right-left+1 and initially unmark all the values in the array.
    - Now iterate through all the primes found through simple sieve and for each unmarked number, mark all its multiples within the range of the current segment(block) b_i.

Pseudocode:

```
//   Input n is a positive integer, finds all the prime numbers less than or equal to sqrt(n)
function sieveOfEratosthenes(n)

        /*
                Declaring a sieve array whose value for the ith index is false if the ith
                value is composite, and is true if the ith value is prime
        */

        bool  sieve[n + 1], primes

        //   Initializing sieve array to be true
        for i = 1 to n
                sieve[i] = true
        sieve[1] = false

        /*
                Now iterating from 2 and for every prime value marking the multiples of
                that prime up to n as composites i.e false
        */
        for i = 2; i * i<=n; i++
                if sieve[i] equals true
                        //   Storing the prime number sieve[i]
                        primes.insert(sieve[i])

                        //   Marking all multiples of i till n to be false(composites)
                        for j = i * i; j<=n; j += i
                                sieve[j] = false

        return primes

/* Input n is a positive integer, finds all the prime numbers from 2 to n. */
function segmentedSieve(n)

        //   Calling simple sieve on n and storing all the prime numbers upto sqrt(n) in
primes
        primes  = sieveOfEratosthenes(n)

        //   Block size
        blk_sz = sqrt(n)

        //   Total blocks
        tot_blk = ceil(n/blk_sz)
```

```
//   Iterating over the block indexes representing different intervals from 0 to
tot_blk-1

for idx = 0 to tot_blk-1
        /*
                Finding the starting number and the ending number representing
                the current block
        */
        start = idx*blk_sz+1
        end = min((idx+1)*blk_sz, n)

        /*
                Declaring a block array representing the current interval of size
                'end-start+1', initialized to true
        */
        block[end-start+1]

        /*
                Iterating over all prime numbers in primes(unmarked) and
                marking all multiples of the current unmarked number till the
                end.
        */
        for prm = 0 to primes.size()-1
                /*
                        Finding the smallest number greater than or equal to start
                        that is a multiple of current prime prm[i]
                */
                prm_start = (start / prm[i]) * prm[i]
                if prm_start < prm[i]
                        prm_start = prm_start + prm[i]
                if prm_start == prm[i]
                        prm_start = prm_start + prm[i]

                /*
                        Marking all the multiples of the prime prm[i], here block[i]
                        represents the value i+start, hence in a way compressing
                        the values.
                */
                for itr = prm_start; itr <= end; itr += prm[i]
                        block[itr-start] = false

        for i = start to end
```

```
                    if block[i-start] equals true
                        print(i)
```

Time complexity: O(n*$\log_2(\log_2 n)$), where n is the number up to which the primes are to be found.

The time and space complexity for the segmented sieve is the same as that of sieve, as it follows the procedure similar to that of sieve, the only difference lies in the way we deal with space, where we take different segments(blocks) representing different ranges of numbers at a time, thus reducing the space to be kept in memory at once.