CRACKING THE PLACEMENT TESTS

A WEEKLY GUIDE CONTAINING TOPICS, QUESTIONS AND TRICKS COMMONLY ENCOUNTERED IN PLACEMENT TESTS

JULY 25, 2020
CODING CLUB, IIT GUWAHATI

Coding Club
IIT Guwahati

WEEK 4

NUMBER THEORY

Number Theory is an integral component to Competitive Programming. In this lecture, we will look upon its various aspects starting with the very basic Euclid's Theorems.

EUCLID'S THEOREMS

There are two theorems given by Euclid.

(1) p|ab => p|a or p|b
(2) There are infinitely many primes

Now we move on to the more useful algorithms given by Euclid.

EUCLIDEAN ALGORITHM FOR COMPUTING THE GREATEST COMMON DIVISOR

The algorithm is extremely simple:

$$\gcd(a,b) = \begin{cases} a, & \text{if } b = 0 \\ \gcd(b, a \bmod b), & \text{otherwise.} \end{cases}$$

Recursive Implementation:

```
1. int gcd (int a, int b) {
2.     if (b == 0)
3.         return a;
4.     else
5.         return gcd (b, a % b);
6. }
```

Iterative Implementation:

```
1. int gcd (int a, int b) {
2.     while (b) {
3.         a %= b;
4.         swap(a, b);
5.     }
6.     return a;
```

```
7.  }
```

Time Complexity:

$$O(\log log\ (a,b)\ )$$

This is an extremely efficient algorithm, as the number of steps required in this algorithm is at most 5 times the number of digits of the smaller number.

NOTE: This algorithm is tail recursive.

EXTENDED EUCLIDEAN ALGORITHM

For this, let us first state Bezout's identity

The bezout's identity states that if d = gcd(a, b) then there always exist integers x and y such that ax + by = d. (Of course, the theory of linear Diophantine equations assures existence of infinitely many solutions, if one exists). It is also worth noting that k = d is the smallest positive integer for which ax + by = k has a solution with integral x and y.

While the Euclidean algorithm calculates only the greatest common divisor (GCD) of two integers a and b, the extended version also finds a way to represent GCD in terms of a and b, i.e. coefficients x and y for which
ax + by = gcd(a,b)

Recursive Implementation:

```
1.  int gcd(int a, int b, int& x, int& y) {
2.      if (b == 0) {
3.          x = 1;
4.          y = 0;
5.          return a;
6.      }
7.      int x1, y1;
8.      int d = gcd(b, a % b, x1, y1);
9.      x = y1;
10.     y = x1 - y1 * (a / b);
11.     return d;
```

```
12.}
```

Iterative Implementation:

```
1. int gcd(int a, int b, int& x, int& y) {
2.     x = 1, y = 0;
3.     int x1 = 0, y1 = 1, a1 = a, b1 = b;
4.     while (b1) {
5.         int q = a1 / b1;
6.         tie(x, x1) = make tuple(x1, x - q * x1);
7.         tie(y, y1) = make tuple(y1, y - q * y1);
8.         tie(a1, b1) = make tuple(b1, a1 - q * b1);
9.     }
10.    return a1;
11.}
```

Now we move on to our next topic in Number Theory namely, Integer Factorization.

INTEGER FACTORIZATION

(1) Trial division method
(2) Using Sieve of Eratosthenes

Let us first describe the Trial division method

This is the most basic algorithm to find a prime factorization.

We divide by each possible divisor d. We can notice, that it is impossible that all prime factors of a composite number n are bigger than $\sqrt{n}$. Therefore, we only need to test the divisors $2 \leq d \leq \sqrt{n}$, which gives us the prime factorization in $O(\sqrt{n})$

The smallest divisor has to be a prime number. We remove the factor from the number, and repeat the process. If we cannot find any divisor in the range $[2; \sqrt{n}]$ then the number itself has to be prime.

Implementation:

```
1. vector<long long> trial division1(long long n) {
2.     vector<long long> factorization;
3.     for (long long d = 2; d * d <= n; d++) {
4.         while (n % d == 0) {
5.             factorization.push back(d);
6.             n /= d;
7.         }
8.     }
9.     if (n > 1)
10.        factorization.push back(n);
11.    return factorization;
12.}
```

Now we describe Sieve of Eratosthenes Method

In this method, we compute all prime numbers till $\sqrt{n}$ and then check for each prime individually. We will describe Sieve of Eratosthenes in detail under its own heading.

Also, if we need to factorize all numbers between 1 to N, this task can be done using a single run of this algorithm - For every integer k between 1 to N, we can maintain a single pair - the smallest prime that divides k, and its highest power, say (p,a). The remaining prime factors of k are then same as that of $k/(p^a)$.

The above trick will be more clear when I present the code for it. But before that, lets know more about Sieve of Eratosthenes.

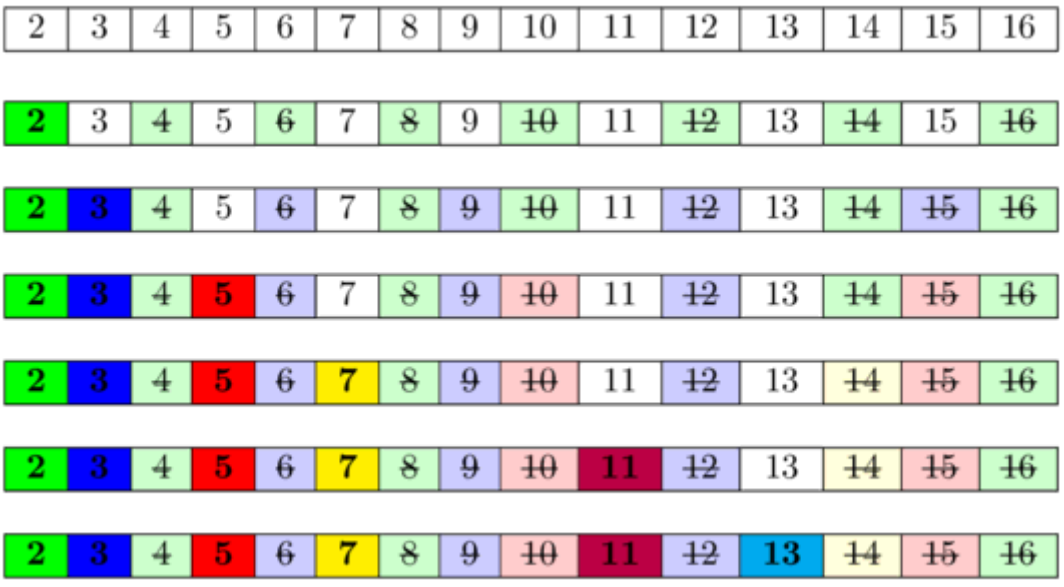
SIEVE OF ERATOSTHENES

Sieve of Eratosthenes is an algorithm for finding all the prime numbers in a segment $[1; n]$ using $O(n \log log \log log n)$ operations.

The algorithm is very simple: at the beginning we write down all numbers between 2 and n. We mark all proper multiples of 2 (since 2 is the smallest prime number) as composite. A proper multiple of a number x is a number greater than x and divisible by x. Then we find the next number that hasn't

been marked as composite, in this case it is 3. Which means 3 is prime, and we mark all proper multiples of 3 as composite. The next unmarked number is 5, which is the next prime number, and we mark all proper multiples of it. And we continue this procedure until we processed all numbers in the row.

In the following image you can see a visualization of the algorithm for computing all prime numbers in the range [1;16]. It can be seen that quite often we mark numbers as composite multiple times.



The idea behind is this: A number is prime, if none of the smaller prime numbers divides it. Since we iterate over the prime numbers in order, we already marked all numbers, who are divisible by at least one of the prime numbers, as divisible. Hence if we reach a cell and it is not marked, then it isn't divisible by any smaller prime number and therefore has to be prime.

Implementation:

```
vector<long long> prime;

void sieve(long long n){
    vector<bool> is prime(n+1, true);
    is prime[0] = is prime[1] = false;
    for (long long i = 2; i * i <= n; i++) {
        if (is prime[i]) {
            for (long long j = i * i; j <= n; j += i)
                is prime[j] = false;
        }
    }
    for(int i=2;i<=n;i++)
    {
        if (is prime[i])
            prime.push back(i);
    }
}
```

Now I present the code for the trick mentioned before, i.e. for factorizing all integers from 1 to n.

```
vector<int> smallest factor;

void factorization from 1 to n(long long n){
    smallest factor.resize(n+1);
    for(int i=1;i<=n;i++)
    {
        smallest factor[i]=i;
    }
    for (long long i = 2; i * i <= n; i++) {
        if (smallest factor[i]==i) {
            for (long long j = i * i; j <= n; j += i){
                if(smallest factor[j]==j){
                    smallest factor[j]=i;
                }
            }
        }
    }
```

```
18.}
```

Now for every number k such that $1 \leq k \leq n$, we know its smallest prime factor. Hence we can determine its complete factorization using the code below

```
1. while(k>1){
2.     int prime = smallest factor[k];
3.     int exponent=0;
4.     while(k%prime==0){
5.         k/=prime;
6.         exponent++;
7.     }
8.     factorization.push back(make pair(prime,exponent));
9. }
```

There are more modifications to the Sieve of Eratosthenes that we will not discuss here.

(1) Finding primes in a range [L;R] where R-L is small (around 1e6) but R can be very large (around 1e12).
https://cp-algorithms.com/algebra/sieve-of-eratosthenes.html#toc-tgt-7
(2) Sieve of Eratosthenes Having Linear Time Complexity
https://cp-algorithms.com/algebra/prime-sieve-linear.html

Now we move on to our next topic which is Primality Tests, i.e. tests to determine whether a given number is prime or composite

PRIMALITY TESTS

https://cp-algorithms.com/algebra/primality_tests.html

MODULAR INVERSE

https://cp-algorithms.com/algebra/module-inverse.html

FACTORIAL modulo p

https://cp-algorithms.com/algebra/factorial-modulo.html

Coding Club
IIT Guwahati

WEEK 4

NUMBER OF DIVISORS, SUM OF DIVISORS

https://cp-algorithms.com/algebra/divisors.html

BIT MANIPULATION

https://www.hackerearth.com/practice/basic-programming/bit-manipulation/basics-of-bit-manipulation/tutorial/

https://www.geeksforgeeks.org/bits-manipulation-important-tactics/

PROBLEMS

https://www.interviewbit.com/courses/programming/topics/math/

https://www.interviewbit.com/courses/programming/topics/bit-manipulation/

https://www.hackerearth.com/practice/math/number-theory/basic-number-theory-1/practice-problems/

https://www.hackerearth.com/practice/basic-programming/bit-manipulation/basics-of-bit-manipulation/practice-problems/

https://www.geeksforgeeks.org/number-theory-competitive-programming/

https://www.geeksforgeeks.org/bitwise-algorithms/

https://leetcode.com/tag/math/

https://leetcode.com/tag/bit-manipulation/