Divisibility (1)

Euclidean algorithm for computing the greatest common divisor

Given two **non-negative** integers a and b, we have to find their GCD (greatest common divisor), i.e. the largest number which is a divisor of both a and b. It's commonly denoted by $\gcd(a, b)$. When one of the numbers is zero, while the other is non-zero, their greatest common divisor, by definition, is the second number. When both numbers are zero, their greatest common divisor is **undefined**, but it is convenient to define it as zero to preserve the associativity of $\gcd$.

Algorithm

Originally, the Euclidean algorithm was formulated as follows: subtract the smaller number from the larger one until one of the numbers is zero. Indeed, if g divides a and b, it also divides $a-b$. On the other hand, if g divides $a-b$ and b, then it also divides $a = b + (a-b)$, which means that the sets of the common divisors of $\{a, b\}$ and $\{b, a-b\}$ coincide. The complexity of this is $\mathcal{O}(a + b)$ (take $\gcd(500, 1)$ as an example) which is not good enough.

Note that a remains the larger number until b is subtracted from it at least $\left\lfloor \frac{a}{b} \right\rfloor$ times. Therefore, to speed things up, $a-b$ is substituted with $a - \left\lfloor \frac{a}{b} \right\rfloor b = a \bmod b$. Now, the time complexity is $\mathcal{O}(\log(\min(a, b)))$. The algorithm is formulated in an extremely simple way:

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

Implementation

```
int gcd(int a, int b) {
    while (b) {
        a %= b;
        swap(a, b);
    }
    return a;
}

int gcd_rec(int a, int b) {
    if (b == 0)
        return a;
    return gcd(b, a % b);
}
```

Least Common Multiple

Calculating the least common multiple (commonly denoted **LCM**) can be reduced to calculating the GCD with the following simple formula: $\operatorname{lcm}(a,b) = \frac{a \cdot b}{\gcd(a,b)}$

A possible implementation, that cleverly avoids integer overflows by first dividing a with the GCD, is given here (zero cases are not taken into account, don't forget about division by 0):

```
int lcm (int a, int b) {
return a / gcd(a, b) * b;
}
```

> ### Binary GCD
>
> The Binary GCD algorithm is an **optimization** to the normal Euclidean algorithm. The slow part of the normal algorithm are the modulo operations. Modulo operations, although we see them as O(1), are a lot slower than simpler operations like addition, subtraction or bitwise operations. So, it would be better to avoid those. Read more here, but keep in mind that such an optimization is usually **NOT** necessary.

Integer factorization

In number theory, integer factorization is the decomposition of a positive integer into a product of integers. If the factors are further restricted to be prime numbers, the process is called prime factorization.

Even though the factorization of an integer can be found using countless algorithms, the *basic* one with a time complexity of $O(\sqrt{n})$ actually gets the job done just fine most of the time. A byproduct of this algorithm is checking whether a number is prime (n is prime if it has no divisor d such that $1 < d < n$).

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 $2 \le d \le \sqrt{n}$, 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

```cpp
void factorization(long long n)
{
    if (n == 1)
    {
        cout << "1\n";
        return;
    }

    int cnt = 0;

    while (n % 2 == 0)
        cnt++, n /= 2;
    if (cnt)
        cout << 2 << " " << cnt << '\n', cnt = 0;

    for (long long d = 3; d * d <= n; d += 2)
    {
        cnt = 0;
        while (n % d == 0)
            cnt++, n /= d;

        if (cnt)
            cout << d << " " << cnt << '\n', cnt = 0;
    }
    if (n > 1)
        cout << n << " 1\n";
}
```

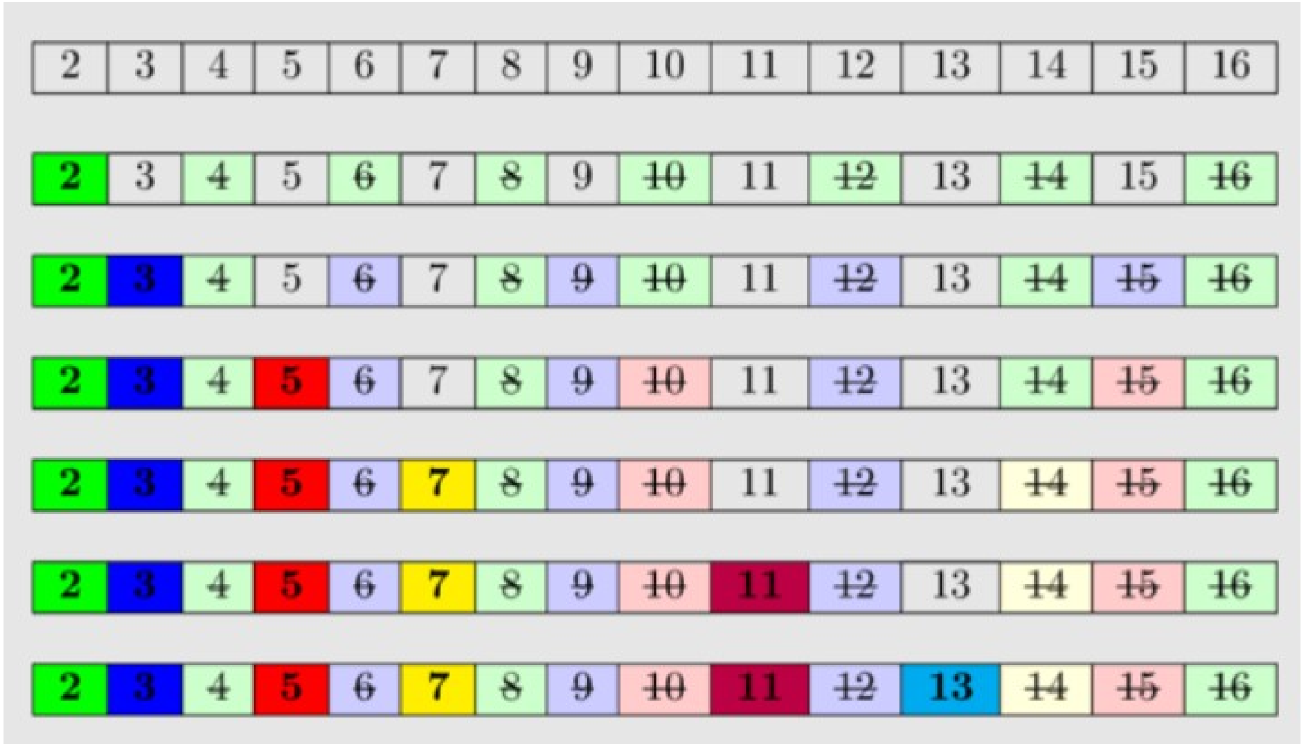
But what if we need to factorize (or check primality for) many numbers? In that case, we can use:

Sieve of Eratosthenes

Sieve of Eratosthenes is an algorithm for finding all the prime numbers in an interval $[1, n]$ using $O(n \log \log n)$ 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 divide it. Since we iterate over the prime numbers in order, we already marked as composite all numbers which are divisible by at least one of the prime numbers. Hence, if we reach a cell and it is not marked, then it isn't divisible by any smaller prime number and therefore is prime.

Implementation

```
// this implementation is far from optimal
// but it should suffice most of the times
vector<int> get_primes(int n)
{
    vector<int> primes;
    vector<bool> is_prime(n + 1, true); // initialize array with `true`
    is_prime[0] = is_prime[1] = false;  // non-prime by definition

    for (int i = 2; i <= n; i++)
        if (is_prime[i]) // has not been marked before
        {
            primes.push_back(i);
            for (long long j = (long long)i * i; j <= n; j += i) // composite numbers less than i*i were already marked as non-
                                                                  // using long long to avoid overflow | (n-1)^2 can easily over
                is_prime[j] = false;
        }
    return primes;
}
```

Time complexity is $\mathcal{O}(n \log\log n)$, but the actual speed depends heavily on implementation details such as cache friendliness. Memory complexity is $\mathcal{O}(n)$.

Keep in mind that when factorizing integers less than N it is enough to generate the prime numbers less than $\sqrt{N}$. Also, the number of primes less than N is approximately $\frac{N}{\ln N}$. So, $\frac{\sqrt{N}}{\ln \sqrt{N}}$ should be a small enough number to get a pretty fast factorization.

Linear sieve

The standard way of computing primes less than n is to use the sieve of Eratosthenes. The algorithm given here achieves the same thing and also facilitates fast factorizations ($\mathcal{O}(\log_2 x)$) for all $x \leq n$ as a side effect, all of that in $\mathcal{O}(n)$.

The weakness of the given algorithm is in using more memory than the classic sieve of Eratosthenes': it requires an array of n numbers, while for the classic sieve of Eratosthenes it is enough to have n bits of memory (which is 32 times less).

Although the running time of $\mathcal{O}(n)$ is better than the $\mathcal{O}(n \log\log n)$ of the classic sieve of Eratosthenes, the difference between them is not so big. In practice the linear sieve runs about as fast as a typical implementation of the sieve of Eratosthenes. In comparison to optimized versions of the sieve of Erathosthenes (e.g. the segmented sieve) it is much slower.

However, its redeeming quality is that this algorithm calculates an array $lp[]$ (least prime), which allows us, for example, to compute the factorization of any number x less than n in $\mathcal{O}(\ell og_2 x)$ complexity. Moreover, using just one extra array will allow us to avoid divisions (which are slow) when doing factorization.

Implementation

```
vector<int> lp(n + 1), pr;
for (int i = 2; i <= n; i++)
{
    if (lp[i] == 0)
        lp[i] = i, pr.push_back(i);
    for (int j = 0; i * pr[j] <= n; ++j)
    {
        lp[i * pr[j]] = pr[j];
        if (pr[j] == lp[i])
            break;
    }
}
```

Notice that every number i has exactly one representation in the form $i = lp[i] \cdot x$, where $lp[i]$ is the minimal prime factor of i, and the number x doesn't have any prime factors less than $lp[i]$, i.e. $lp[i] \leq lp[x]$. Now, let's compare this with the actions of our algorithm: in fact, for every x it goes through all prime numbers it could be multiplied by, i.e. all prime numbers up to $lp[x]$ inclusive, in order to get the numbers in the form given above.

Exercises

- Check the GCD algorithm itself here (infoarena, pbinfo) and the sieve here (infoarena, pbinfo)
- Some simple problems. Just figure out how to apply the above learned algorithms: pbinfo, pbinfo, pbinfo, pbinfo
- Dreptunghi (infoarena): GCD shows up out of nowhere
- Common divisors (CSES): combine the technique from the sieve with the definition of the GCD
- Sherlock and his girlfriend (CF): try rephrasing the statement
- Weakened Common Divisor (CF): use the complexity of the GCD as a hint
- Big Vova (CF): take it step by step and, again, use the complexity of the GCD as a hint
- Enlarge GCD (CF): don't forget what the GCD really is