ALGO-101

Week 3 - Math

Fatih Çakır

ITU ACM

November 2023

Topics

Topics covered at week 3:

- Prime Numbers & Sieve
- Prime Factorization
- GCD - LCD
- Permutation & Combination
- Fast Exponentiation

Primality Test

A number is prime if it is only evenly divisible by one and itself. How to find if a number is prime or not?

Naive Approach:
We should check the numbers from 2 to n-1 to see if any number evenly divides n. The divisor of a number must be equal to or less than itself, so we just need to look at numbers less than n.

How to optimize the naive approach?
To iterate the numbers to $\sqrt{n}$ instead of n. If the number (n) has a divisor bigger than $\sqrt{n}$, then the other multiplier must be smaller than $\sqrt{n}$. And, we already checked the numbers to $\sqrt{n}$, so we don't need to look at the rest of it.

Optimized Primality Test

```
1
2  bool isPrime(int n){
3
4      if(n == 0 || n == 1) return false;
5
6      for(int i = 2; i*i <= n; i++){
7          // If there is a number divide n evenly, n is not a prime
8          if(n % i == 0) return false;
9      }
10
11     return true;
12 }
```

Finding Primes Up to N

We learned how to find if a number is prime or not. Now, we will learn how to find primes up to N.

Naive Approach:
We can use isPrime function for every number from 1 to N. Its time complexity is $O(N \times \sqrt{N})$.

Sieve of Eratosthenes Approach:
It is one of the most efficient ways to find primes up to N. It is based on marking multiples of prime numbers as not prime. Let's look at the example.

Sieve Algorithm

For $N = 50$

1	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
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50

Sieve Algorithm

We start from 2 and marked multiples of 2.

1	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
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50

Sieve Algorithm

If the next number is unmarked, it means that the number is not a multiple of any number less than itself. So, it's a prime number. 3 is unmarked, mark multiples of 3. Skip 4 because it is marked.

1	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
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50

Sieve Algorithm

Unmarked numbers are prime numbers. Like as optimized naive approach, we check up to $\sqrt{N}$.

1	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
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50

Sieve Algorithm Code

```
vector<bool> Sieve_of_Erotosthenes(int n){

    vector<bool> isPrime (n+1, true);
    isPrime[0] = false;
    isPrime[1] = false;

    for(int i = 2; i*i <= n; i++){
        if(isPrime[i]){
            for(int j = i*2; j <= n; j+=i){
                isPrime[j] = false;
            }
        }
    }
    return isPrime;
}
```

Time Complexity of Sieve Algorithm

Inside for loop does n/p_i step if p_i is prime. Total processes is equal to sum of $1/p_i$ from 1 up to $\sqrt{n}$.

$$\frac{n}{2} + \frac{n}{3} + \frac{n}{5} + \ldots + \frac{n}{p_i} = n \times \left\{ \frac{1}{2} + \frac{1}{3} + \frac{1}{5} + \ldots + \frac{1}{p_i} \right\}$$

According to prime harmonic series, the sum $(1/i)$ where i is prime is $\log \log n$.

$\sum_{i=2}^{n} \frac{1}{p_i} = \log \log n$.

Time complexity of sieve approach is $O(n \times \log logn)$

Factorization Algorithm

How to find all factors of given number n?

Naive Approach:
Check the numbers from 1 to n if divides n, then it is a factor of n.

Optimized naive approach:
Instead of checking up to n, check the numbers up to $\sqrt{n}$, then add i and n/i to the array.
The logic of this approach is the same as for optimized primality test.

Prime Factorization

To find prime factors of a number (n), we can use a similar method to sieve approach. Create an array that stores the smallest prime factor of every number from 2 to n.

We add arr[n], the first prime factor of n, to the array of prime factors. Then we divide n by its smallest prime factor. The result will be n/arr[n]. We repeat this operation until the result is 1. Let's look at the example.

Prime Factorization

For N = 18
Create an array size of $n + 1$, then assign every index to index value.

index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
value	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

n

Starting from two, assign all its multiples to i.

index	1	**2**	3	**4**	5	**6**	7	**8**	9	**10**	11	**12**	13	**14**	15	**16**	17	**18**
value	1	**2**	3	**2**	5	**2**	7	**2**	9	**2**	11	**2**	13	**2**	15	**2**	17	**2**

Prime Factorization

For the next index if arr[i] equals i (i is a prime number) mark all multiples. If it is not, that means i is not a prime number, skip it.
While marking the multiples, if it is already marked with another value, skip it. Because every index should store the smallest prime factor of itself. In this example, skip 6, 12, 18.

index	1	**2**	**3**	**4**	5	**6**	7	**8**	**9**	**10**	11	**12**	13	**14**	**15**	**16**	17	**18**
value	1	**2**	**3**	**2**	5	**2**	7	**2**	**3**	**2**	11	**2**	13	**2**	**3**	**2**	17	**2**

It is redundant to look at any bigger than $\sqrt{n}$, because the rest unsigned are prime numbers.

Prime Factorization

After creating the array, to find prime factors;

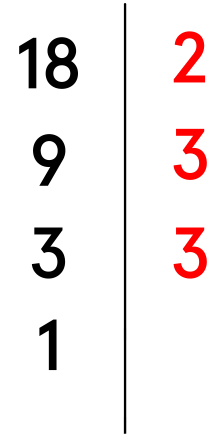
$n = 18$ arr[18] = **2**;
$n = 18 \ / \ 2$;

$n = 9$ arr[9] = **3**;
$n = 9 \ / \ 3$;

$n = 3$ arr[3] = **3**;
$n = 3 \ / \ 3$;

$n = 1$ end condition

Prime factors = {2, 3, 3}



GCD - Greatest of Common Divisors

Take two integers:

$$x = p_1^{a_1}.p_2^{a_2}.p_3^{a_3}...$$
$$y = p_1^{b_1}.p_2^{b_2}.p_3^{b_3}...$$

Common divisor z: Must both divide x and y.

$$z = p_1^{c_1}.p_2^{c_2}.p_3^{c_3}...$$

GREATEST common divisor $\gcd(x, y)$: $c_i = \min(a_i, b_i)$
Applicable to any number of integers, not just two

GCD - Eucludean Algorithm

$$\gcd(x, y) = \gcd(x \bmod y,\ y) = \gcd(x,\ y \bmod x)$$

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

$$O(logmin(x, y))!$$

LCM - Least of Common Multiples

Common multiple z: Must both divisible by both x and y.

$$z = p_1^{c_1}.p_2^{c_2}.p_3^{c_3}...$$

GREATEST common divisor lcm(x, y): $c_i = \max(a_i, b_i)$
Applicable to any number of integers, not just two

Permutation

Permutation is the number of possible orders of a given set. Formula;

$$P(n, k) = \frac{n!}{(n-k)!}$$

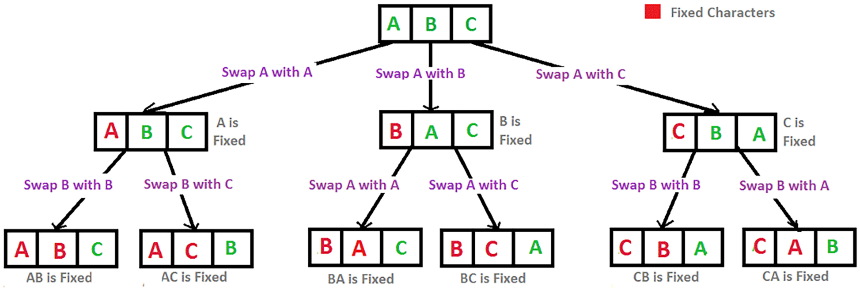
We can find the p with a for loop and its time complexity is k. For bigger numbers it will be find with fast exponentiation. But how can we find the all permutation of given set?

There are many ways to find all permutations. However, all of them time complexity is $O(n \times n!)$. We solve this problem by swapping or more known name backtracking.

Generating Permutation

First, swap with 0 index element to other indexes starting from 0.
First element was swapped. Skip to second element, and swap with other elements starting from second element.
It goes like until no element left to swap. For example;

$$s = \{A, B, C\};$$



Recursion Tree for Permutations of String "ABC"

Permutation Code

```
1  class Solution {
2      // store all permutations
3      vector<vector<int>> permutations;
4
5      public:
6      vector<vector<int>> permute(vector<int>& set) {
7          // swap starting 0 index
8          swap_function(set, 0);
9          return permutations;
10     }
```

Permutation Code

```
1      void swap_function(vector<int>& subset, int index){
2          // return when there is no element left to swap
3          if(index == subset.size()){
4              permutations.push_back(subset);
5              return;
6          }
7          for(int i = index; i < subset.size(); i++){
8              swap(subset[index], subset[i]);
9              // 0 to index value are swapped and fixed
10             // so call the swap_function for the rest of it
11             swap_function(subset, index + 1);
12
13             // swap again and revert to original
14             swap(subset[index], subset[i]);
15         }
16     }
17 };
```

Combination

Combination is to find subsets length k of n length set. Unlike permutation, order does not matter for combination. Formula for combination;

$$C(n,k) = \frac{n!}{(n-k)! \times k!}$$

It can be solved by a factorial function and its time complexity is $O(n)$.
In optimized version, time complexity would be $O(r)$, where r is minimum of $n-k$ and k.

While finding combination and permutation, overflow may happen if the result is bigger than limit of "int" in any step . To avoid this, we will use the modulo operation.

Modular Division

We can find n mod m with n % m, but if we try to find $\frac{n}{k}$ mod m. That equation not equals $\frac{n\%m}{k\%m}$. Example;

$24/8 \bmod 5 = 3$
$24 \bmod 5 = 4$
$8 \bmod 5 = 3$
$\frac{24\%5}{8\%5} = 4/3$

How can we solve this?
(n mod m) $\times$ (inverse of k under modulo m)
$24 \bmod 5 \times$ (inverse of 8 under modulo 5)
$4 \times 2 = 8$
$8 \bmod 5 = 3$

Modular Inverse

The product of a number and its multiplicative inverse is 1.

$$n \times \frac{1}{n} = 1$$

Modular multiplicative inverse of n under modulo p is n^{-1};

$$n \times n^{-1} (mod\ p) = 1$$

Example;

n = 3 p = 11 $n^{-1} = ?$

$(3 \times 4)\ \%\ 11 = 1$

Note: p must be co-prime to n, and the modular inverse of n should be in the range 1 to p-1.
How to find modular inverse of n under modulo p?

Naive Approach:

Look in the range [1,m-1], and check a number (k) for $(n \times\ k)(mod\ m)$ is 1.

Fermat's Little Theorem

Fermat's little theorem states that if p is a prime number, then for any integer a, the number $a^p - a$ is multiple of p.

$$a^p \equiv a \, (mod \; p).$$
$$a^{p-1} \equiv 1 \, (mod \; p).$$

Special Case: If a is not divisible by p, we can multiply both sides by a^{-1}, and we get;

$$a^{p-2} = a^{-1} (mod \; p)$$
$$a^{-1} = a^{p-2} (mod \; p)$$

So we can find modular inverse of a under modulo p as a^{p-2}.

Exponentiation

First way that come to mind for to find n^k is multiply n to itself k times in a for loop. It is naive approach of exponentiation.

However there is a way this problem can be solved by another approach with $O(\log k)$ time complexity called **fast exponentiation**. That base on this rule

if k is **even** $n^k = n^{k/2} \times n^{k/2}$

if k is **odd** $n^k = n^{k-1/2} \times n^{k-1/2} \times n$

Fast Exponentiation Code

```cpp
#define MOD 1000000007

long long fastExponentiation(long long n, long long m){

    if (n == 1 || m == 0) return 1;
    if (n == 0) return 0;

    long long x =  fastExponentiation(n, m/2) % MOD;

    x = (x*x) % MOD;

    if (m%2 == 0) return x;
    else return (x*n) % MOD;
}
```

Problem Session

Math Problems
Count Primes - Prime Numbers
Permutations - Permutation
Prime Arrangements - Prime and Permutation
C(10,5) - Combination
ZSUM - Fast Exponentiation

References

https://www.cs.cmu.edu/afs/cs/academic/class/15859-f01/www/notes/comb.pdf
https://www.geeksforgeeks.org/
https://www.geeksforgeeks.org/modular-exponentiation-power-in-modular-arithmetic/
https://www.geeksforgeeks.org/write-a-c-program-to-print-all-permutations-of-a-given-string/
https://www.geeksforgeeks.org/compute-ncr-p-set-3-using-fermat-little-theorem/
https://www.geeksforgeeks.org/multiplicative-inverse-under-modulo-m/
https://www.geeksforgeeks.org/c-bitset-and-its-application/
https://github.com/inzva/Algorithm-Program/blob/master/bundles/03-math-1/03_math1.pdf
https://codility.com/media/train/9-Sieve.pdf
https://en.wikipedia.org/wiki/Harmonic_(*mathematics*)
https://en.wikipedia.org/wiki/Euclidean_*algorithm*