Number theory and combinatorics fundamentals for competitive programming

CPPoliTo

- Nima Naderi Ghotbodini
- Ghassane Ben El Aattar

Number theory

Divisors of a number (1)

Computing efficiently the divisors of a number is an important part of many number theory problems in competitive programming.

The most used algorithm to compute the divisors of a number works in O(sqrt(n)) complexity, where n is the number for which we want to find its divisors.

Divisors of a number (2)

```
vector<int> findDivisors(int n){
    vector<int> divisors;
    for(int i = 1; i * i <= n; i++){
        if(n % i == 0){ //if i divides n
            if((n / i) == i){ //this avoids taking a divisors twice
                //if n is a perfect square
                divisors.push_back(i); //add i to the divisors
            }
            else{ //these are both divisors. One of them is above
                //sqrt(n) and the other below it
                divisors.push_back(i);
                divisors.push_back(n / i);
            }
        }
    }
    return divisors;
}
```

Prime numbers (1)

Prime numbers are an important concept in number theory. Understanding their properties and how to handle them is a prerequisite to solve many number theory problems in competitive programming.

- Prime number have only two divisors: 1 and the number itself;
- All of the primes except 2 are odd numbers;
- We can check if a number is prime by computing its divisors, but this isn’t always the most efficient way.

Prime numbers (2)

There exists certain conjectures regarding primes that are useful in problems.

- **Goldbach's conjecture**: Each even integer $n > 2$ can be represented as a sum $n = a + b$ so that both a and b are primes.
- **Twin prime conjecture**: There is an infinite number of pairs of the form $\{p, p+2\}$, where both p and $p+2$ are primes.
- **Legendre's conjecture**: There is always a prime between numbers n^2 and $(n+1)^2$, where n is any positive integer.

Prime factorization

Every natural number **n > 1** has a unique prime factorization.

$$n = p_1^{\alpha_1} p_2^{\alpha_2} \cdots p_k^{\alpha_k},$$

where $p_1, p_2, \ldots, p_k$ are distinct primes and $\alpha_1, \alpha_2, \ldots, \alpha_k$ are positive numbers. For example, the prime factorization for 84 is

$$84 = 2^2 \cdot 3^1 \cdot 7^1.$$

Greatest common divisors and least common multiple (1)

Some useful facts about GCD and LCM of two numbers **a** and **b**:

- LCM(A, B) = (A * B) / GCD(A, B)
- GCD(A, A) = A
- GCD(0, A) = A
- GCD(A, B) = GCD(B, A)

Greatest common divisors and least common multiple (2)

Notice that we can easily find the GCD and LCM of two numbers by using their prime factorization:

- For the GCD, take only the common primes factors between the two values with the lowest exponent;
- For the LCM, take all of the factors from the two values with the highest exponent.

Sieve of Eratosthenes (1)

Algorithm to check whether number in the range [2, n] are primes or not and also compute the shortest prime factor for each of them.

The algorithms works in O(n loglogn).

```
for (int x = 2; x <= n; x++) {
    if (sieve[x]) continue;
    for (int u = 2*x; u <= n; u += x) {
        sieve[u] = x;
    }
}
```

Sieve of Eratosthenes (2)

This is the resulting “sieve” array for n = 20. The inner for loop will only be accessed when “x” is a prime.

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	0	2	0	3	0	2	3	5	0	3	0	7	5	2	0	3	0	5

Prime factorization using Sieve

```
vector<int> getFactorization(int x) {
    vector<int> primes;
    if(x == 1){
        return primes;
    }
    while (x != 0) {
        if(sieve[x] == 0){ //this is a prime
            //append to array and set to 0 to quit
            primes.push_back(x);
            x = 0;
        }
        else{ //not a prime, append to array smallest prime factor
            //and divide by it
            primes.push_back(sieve[x]);
            x /= sieve[x];
        }
    }
    return primes;
}
```

Basic modular arithmetic (1)

Arithmetic regarding remainders of divisions.

$$\begin{aligned}
131 &\equiv 2 \pmod{3} \\
131 &\equiv 1 \pmod{5} \\
131 &\equiv 5 \pmod{7} \\
131 &\equiv 3 \pmod{8}
\end{aligned}$$

Basic modular arithmetic (2)

$4 \bmod 4 = 0$	$8 \bmod 4 = 0$
$5 \bmod 4 = 1$	$9 \bmod 4 = 1$
$6 \bmod 4 = 2$	$10 \bmod 4 = 2$
$7 \bmod 4 = 3$	$11 \bmod 4 = 3$

Basic modular arithmetic (3)

There are many modular arithmetic properties that are useful in problems, we'll show some of them here:

- (a + b) mod m = [(a mod m) + (b mod m)] mod m
- (a * b) mod m = [(a mod m) * (b mod m)] mod m

These two properties are crucial to know to solve problems (especially counting problems) where the answer is expected modulo a large prime.

Combinatorics

Permutations & Factorial

Permutations refer to the arrangements of objects in a specific order.
The permutation of numbers in problems refers to arrays where elements are from 1 to n (all distinct) in some order.

There are $n!$ permutations of length n.

$$n! = n \times (n-1) \times (n-2) \times \ldots \times 1$$

$$1! = 1$$
$$2! = 2 \times 1 = 2$$
$$3! = 3 \times 2 \times 1 = 6$$
$$4! = 4 \times 3 \times 2 \times 1 = 24$$
$$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$$

Permutations & Factorial

The formula for permutations of a set of n distinct objects taken r at a time is given by:

$n! / (n - r)!$

n! represents the total number of arrangements when all n objects are considered. By dividing $n!$ by $(n - r)!$, we effectively eliminate the arrangements of the remaining $(n - r)$ objects and only count the arrangements of the chosen r objects.

Binomial coefficients

The **binomial coefficient** $\binom{n}{k}$ equals the number of ways we can choose a subset of k elements from a set of n elements. For example, $\binom{5}{3} = 10$, because the set $\{1,2,3,4,5\}$ has 10 subsets of 3 elements:

$$\{1,2,3\},\{1,2,4\},\{1,2,5\},\{1,3,4\},\{1,3,5\},\{1,4,5\},\{2,3,4\},\{2,3,5\},\{2,4,5\},\{3,4,5\}$$

Binomial coefficients

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

Sample Problem

A fair coin is tossed 10 times. What is the probability of getting exactly 4 heads?

Sample Problem

A fair coin is tossed 10 times. What is the probability of getting exactly 4 heads?

Probability = (Number of ways to get 4 heads) / (Total number of possible outcomes)

The number of ways to get 4 heads can be represented by the binomial coefficient "10 choose 4," denoted as C(10, 4).

C(10, 4) = 10! / (4! * (10 - 4)!) = 210.

The total number of possible outcomes in 10 coin tosses is 2^10 = 1024.

Probability = 210 / 1024 ≈ 0.2051 (20.51%)

Binomial coefficients calculation (1st Method)

Formula 1

Binomial coefficients can be recursively calculated as follows:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

The idea is to fix an element x in the set. If x is included in the subset, we have to choose $k-1$ elements from $n-1$ elements, and if x is not included in the subset, we have to choose k elements from $n-1$ elements.

The base cases for the recursion are

$$\binom{n}{0} = \binom{n}{n} = 1,$$

because there is always exactly one way to construct an empty subset and a subset that contains all the elements.

Binomial coefficients calculation (2nd Method)

Formula 2

Another way to calculate binomial coefficients is as follows:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}.$$

There are $n!$ permutations of n elements. We go through all permutations and always include the first k elements of the permutation in the subset. Since the order of the elements in the subset and outside the subset does not matter, the result is divided by $k!$ and $(n-k)!$

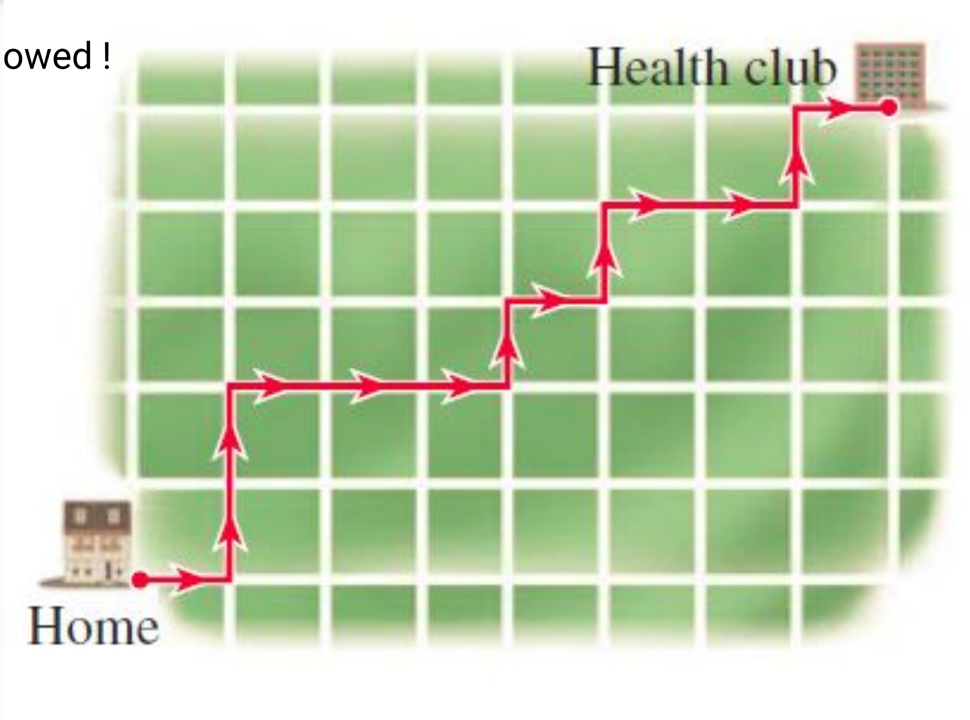
Some Facts

$$\binom{n}{0} + \binom{n}{1} + \binom{n}{2} + \ldots + \binom{n}{n} = 2^n.$$

$$(a+b)^n = \binom{n}{0} a^n b^0 + \binom{n}{1} a^{n-1} b^1 + \ldots + \binom{n}{n-1} a^1 b^{n-1} + \binom{n}{n} a^0 b^n.$$

Number of ways in n x m grid

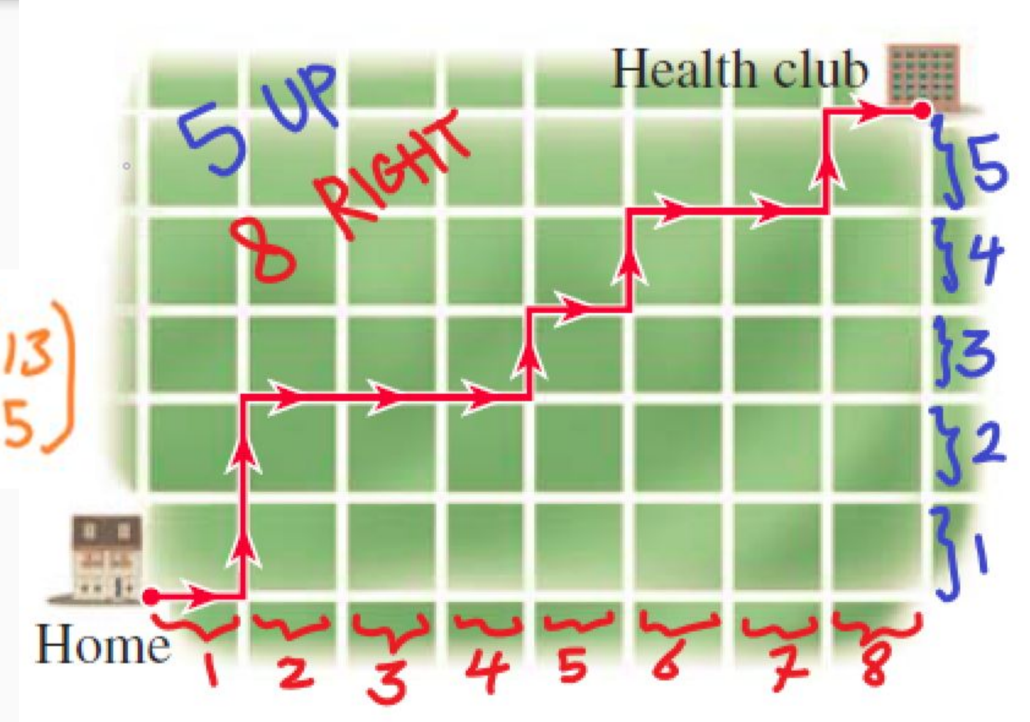
Only Right and Up are allowed !



Number of ways in n x m grid

Only Right and Up are allowed !

$$\binom{8+5}{8} = \binom{8+5}{5} = \binom{13}{5}$$



Finally Code!

```
#include <bits/stdc++>
using namespace std;

// Recursive function to calculate the binomial coefficient
int nCr(int n, int k) {
    if (k == 0 || k == n)
        return 1;
    else
        return nCr(n - 1, k - 1) + nCr(n - 1, k);
}

int main() {
    int n, k;
    cout << "Enter the values of n and k: \n";
    cin >> n >> k;

    int result = nCr(n, k);
    cout << result << '\n';

    return 0;
}
```

Finally Code!

```
#include <bits/stdc++.h>
using namespace std;

typedef long long ll;
const ll MXN = 1e3 + 10;

int n, k;
ll C[MXN][MXN];

int main() {
    cin >> n >> k;

    for (int i = 0; i <= n; i ++){
        C[i][0] = 1;
    }
    for (int i = 1; i <= n; i ++) {
        for (int j = 1; j <= i; j ++) {
            C[i][j] = C[i - 1][j - 1] + C[i - 1][j];
        }
    }

    cout << C[n][k] << '\n';
    return 0;
}
//! N.N
```

Finally Code!

```cpp
#include <bits/stdc++.h>
using namespace std;

typedef long long ll;
const ll MXN = 1e3 + 10;
const ll Mod = 1e9 + 7;

int n, k;
int C[MXN][MXN];

int main() {
    cin >> n >> k;

    for (int i = 0; i <= n; i ++){
        C[i][0] = 1;
    }
    for (int i = 1; i <= n; i ++) {
        for (int j = 1; j <= i; j ++) {
            C[i][j] = (C[i - 1][j - 1] + C[i - 1][j]) % Mod;
        }
    }

    cout << C[n][k] << '\n';
    return 0;
}
//! N.N
```

Subsequences vs Subarrays

2	7	3	4	1	6	5

$$2\ \underbrace{7\ \ 3\ \ 4}_{\text{Subarray}}\ 1\ \ 6\ \ 5$$

$$\#\text{Subarrays} = \frac{n(n+1)}{2} = \binom{n+1}{2}$$

Subsequences vs Subarrays

2	7	3	4	1	6	5

[7, 3, 1, 5]

Subsequence

$\#\text{Subsequences} = 2^n$

Subsequences vs Subarrays

2	7	3	4	1	6	5

[7, 3, 1, 5]

Subsequence

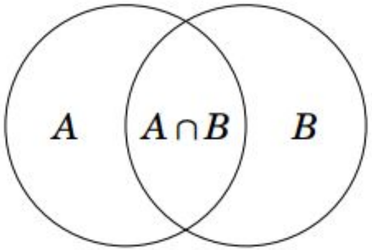
$$\underbrace{\#\text{Subsequences} = 2^n - 1}_{\text{non-empty}}$$

Principle of Inclusion-exclusion

Inclusion-exclusion is a technique that can be used for counting the size of a union of sets when the sizes of the intersections are known, and vice versa. A simple example of the technique is the formula

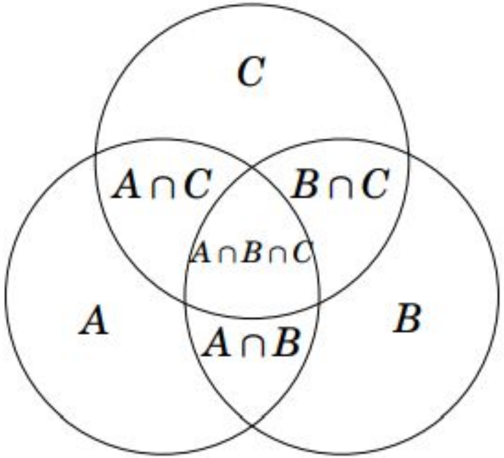
$$|A \cup B| = |A| + |B| - |A \cap B|,$$

where A and B are sets and $|X|$ denotes the size of X. The formula can be illustrated as follows:



Principle of Inclusion-exclusion

$$|A \cup B \cup C| = |A| + |B| + |C| - |A \cap B| - |A \cap C| - |B \cap C| + |A \cap B \cap C|$$



Thank you for your attention!

Links and contacts:
https://linktr.ee/politocompetitiveprogramming