October 31, 2018

MATHEMATICS FOR TOPCODERS

Dmitri Kamenetsky
dimkadimon

DURATION
15 min

CATEGORIES
Competitive Programming
Tutorials

TAGS
GCD | Complex numbers | JAVA | Advanced Math | Competitive Programming Tutorials

Discuss this article in the forums

INTRODUCTION

I have seen a number of competitors complain that they are unfairly disadvantaged because many topcoder problems are too mathematical. Personally, I love mathematics and thus I am biased in this issue. Nevertheless, I strongly believe that problems should contain at least some math, because mathematics and computer science often go hand in hand. It is hard to imagine a world where these two fields could exist without any interaction with each other. These days, a great deal of applied mathematics is performed on computers such as solving large systems of equations and approximating solutions to differential equations for which no closed formula exists. Mathematics is widely used in computer science research, as well as being heavily applied to graph algorithms and areas of computer vision.

This article discusses the theory and practical application to some of the more common mathematical constructs. The topics covered are: primes, GCD, basic geometry, bases, fractions and complex numbers.

PRIMES

A number is prime if it is only divisible by 1 and itself. So for example 2, 3, 5, 79, 311 and 1931 are all prime, while 21 is not prime because it is divisible by 3 and 7. To find if a number n is prime we could simply check if it divides any numbers below it. We can use the modulus (%) operator to check for divisibility:

```
for (int i = 2; i & lt; n; i++)
 if (n % i == 0) return false;
```

```java
return true;
```

We can make this code run faster by noticing that we only need to check divisibility for values of i that are less or equal to the square root of n (call this m). If n divides a number that is greater than m then the result of that division will be some number less than m and thus n will also divide a number less or equal to m. Another optimization is to realize that there are no even primes greater than 2. Once we’ve checked that n is not even we can safely increment the value of i by 2. We can now write the final method for checking whether a number is prime:

```java
public boolean isPrime(int n) {
  if (n & lt; = 1) return false;
  if (n == 2) return true;
  if (n % 2 == 0) return false;
  int m = Math.sqrt(n);
  <br/>
  for (int i = 3; i & lt; = m; i += 2)
    if (n % i == 0)
      return false;
  <br/>
  return true;
}
```

Now suppose we wanted to find all the primes from 1 to 100000, then we would have to call the above method 100000 times. This would be very inefficient since we would be repeating the same calculations over and over again. In this situation it is best to use a method known as the Sieve of Eratosthenes. The Sieve of Eratosthenes will generate all the primes from 2 to a given number n. It begins by assuming that all numbers are prime. It then takes the first prime number and removes all of its multiples. It then applies the same method to the next prime number. This is continued until all numbers have been processed. For example, consider finding primes in the range 2 to 20. We begin by writing all the numbers down:
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
2 is the first prime. We now cross out all of its multiples, ie every second number:
2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ 9 ~~10~~ 11 ~~12~~ 13 ~~14~~ 15 ~~16~~ 17 ~~18~~ 19 ~~20~~
The next non-crossed out number is 3 and thus it is the second prime. We now cross out all the multiples of 3, ie every third number from 3:
2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ ~~9~~ ~~10~~ 11 ~~12~~ 13 ~~14~~ ~~15~~ ~~16~~ 17 ~~18~~ 19 ~~20~~
All the remaining numbers are prime and we can safely terminate the algorithm. Below is the code for the sieve:

```java
public boolean[] sieve(int n) {
  boolean[] prime = new boolean[n + 1];
  Arrays.fill(prime, true);
  prime[0] = false;
  prime[1] = false;
  int m = Math.sqrt(n);
  <br/>
  for (int i = 2; i & lt; = m; i++)
    if (prime[i])
      for (int k = i * i; k & lt; = n; k += i)
        prime[k] = false;
  <br/>
  return prime;
}
```

In the above method, we create a boolean array prime which stores the primality of each number less of equal than n. If prime[i] is true then number i is prime. The outer loop finds the next prime while the inner loop removes all the multiples of the current prime.

GCD

The greatest common divisor (GCD) of two numbers a and b is the greatest number that divides evenly into both a and b. Naively we could start from the smallest of the two numbers and work our way downwards until we find a number that divides into both of them:

```
for (int i = Math.min(a, b); i & gt; = 1; i--)
 if (a % i == 0 & amp; & amp; b % i == 0)
   return i;
```

Although this method is fast enough for most applications, there is a faster method called Euclid’s algorithm. Euclid’s algorithm iterates over the two numbers until a remainder of 0 is found. For example, suppose we want to find the GCD of 2336 and 1314. We begin by expressing the larger number (2336) in terms of the smaller number (1314) plus a remainder:

```
2336 = 1314 x 1 + 1022
```

We now do the same with 1314 and 1022:

```
1314 = 1022 x 1 + 292
```

We continue this process until we reach a remainder of 0:

```
1022 = 292 x 3 + 146
292 = 146 x 2 + 0
```

The last non-zero remainder is the GCD. So the GCD of 2336 and 1314 is 146. This algorithm can be easily coded as a recursive function:

```
//assume that a and b cannot both be 0
public int GCD(int a, int b) {
  if (b == 0) return a;
  return GCD(b, a % b);
}
```

Using this algorithm we can find the lowest common multiple (LCM) of two numbers. For example the LCM of 6 and 9 is 18 since 18 is the smallest number that divides both 6 and 9. Here is the code for the LCM method:

```
public int LCM(int a, int b) {
  return b * a / GCD(a, b);
}
```

As a final note, Euclid's algorithm can be used to solve linear Diophantine equations. These equations have integer coefficients and are of the form:

```
ax + by = c
```

GEOMETRY

Occasionally problems ask us to find the intersection of rectangles. There are a number of ways to represent a rectangle. For the standard Cartesian plane, a common method is to store the coordinates of the bottom-left and top-right corners.

Suppose we have two rectangles R1 and R2. Let (x1, y1) be the location of the bottom-left corner of R1 and (x2, y2) be the location of its top-right corner. Similarly, let (x3, y3) and (x4, y4) be the respective corner locations for R2. The intersection of R1 and R2 will be a rectangle R3 whose bottom-left corner is at (max(x1, x3), max(y1, y3)) and top-right corner at (min(x2, x4), min(y2, y4)). If max(x1, x3) > min(x2, x4) or max(y1, y3) > min(y2, y4) then R3 does not exist, ie R1 and R2 do not intersect. This method can be extended to intersection in more than 2 dimensions as seen in CuboidJoin (SRM 191, Div 2 Hard).

Often we have to deal with polygons whose vertices have integer coordinates. Such polygons are called lattice polygons. In his tutorial on Geometry Concepts, lbackstrom presents a neat way for finding the area of a lattice polygon given its vertices. Now, suppose we do not know the exact position of the vertices and instead we are given two values:
B = number of lattice points on the boundary of the polygon
I = number of lattice points in the interior of the polygon
Amazingly, the area of this polygon is then given by:

```
Area = B/2 + I - 1
```

The above formula is called Pick's Theorem due to Georg Alexander Pick (1859 – 1943). In order to show that Pick's theorem holds for all lattice polygons we have to prove it in 4 separate parts. In the first part we show that the theorem holds for any lattice rectangle (with sides parallel to axis). Since a right-angled triangle is simply half of a rectangle it is not too difficult to show that the theorem also holds for any right-angled triangle (with sides parallel to axis). The next step is to consider a general triangle, which can be represented as a rectangle with some right-angled triangles cut out from its corners. Finally, we can show that if the theorem holds for any two lattice polygons sharing a common side then it will also hold for the lattice polygon, formed by removing the common side. Combining the previous result with the fact that every simple polygon is a union of triangles gives us the final version of Pick's Theorem. Pick's theorem is useful when we need to find the number of lattice points inside a large polygon.

Another formula worth remembering is Euler's Formula for polygonal nets. A polygonal net is a simple polygon divided into smaller polygons. The smaller polygons are called faces, the sides of the faces are called edges and the vertices of the faces are called vertices. Euler's Formula then states:
V - E + F = 2, where
V = number of vertices
E = number of edges
F = number of faces
For example, consider a square with both diagonals drawn. We have V = 5, E = 8 and F = 5 (the outside of the square is also a face) and so V – E + F = 2.

We can use induction to show that Euler's formula works. We must begin the induction with V = 2, since every vertex has to be on at least one edge. If V = 2 then there is only one type of polygonal net possible. It has two vertices connected by E number of edges. This polygonal net has E faces (E – 1 "in the middle" and 1 "outside"). So V – E + F = 2 – E + E = 2. We now assume that V – E + F = 2 is true for all 2<=V<=n. Let V = n + 1. Choose any vertex w at random. Now suppose w is joined to the rest of the net by G edges. If we remove w and all these edges, we have a net with n vertices, E – G edges and F – G + 1 faces. From our assumption, we have:

```
(n) - (E - G) + (F - G + 1) = 2
thus (n+1) - E + F = 2
```

Since V = n + 1, we have V – E + F = 2. Hence by the principal of mathematical induction we have proven Euler's formula.

BASES

A very common problem faced by topcoder competitors during challenges involves converting to and from binary and decimal representations (amongst others).

So what does the base of the number actually mean? We will begin by working in the standard (decimal) base. Consider the decimal number 4325. 4325 stands for 5 + 2 x 10 + 3 x 10 x 10 + 4 x 10 x 10 x 10. Notice that the “value” of each consequent digit increases by a factor of 10 as we go from right to left.

Binary numbers work in a similar way. They are composed solely from 0 and 1 and the “value” of each digit increases by a factor of 2 as we go from right to left. For example, 1011 in binary stands for 1 + 1 x 2 + 0 x 2 x 2 + 1 x 2 x 2 x 2 = 1 + 2 + 8 = 11 in decimal. We have just converted a binary number to a decimal. The same applies to other bases. Here is code which converts a number n in base b (2<=b<=10) to a decimal number:

```
public int toDecimal(int n, int b) {
  int result = 0;
  int multiplier = 1;

  while (n & gt; 0) {
    result += n % 10 * multiplier;
    multiplier *= b;
    n /= 10;
  }

  return result;
}
```

Java users will be happy to know that the above can be also written as:

```
return Integer.parseInt(""+n,b);
```

To convert from a decimal to a binary is just as easy. Suppose we wanted to convert 43 in decimal to binary. At each step of the method we divide 43 by 2 and memorize the remainder. The final list of remainders is the required binary representation:

```
43/2 = 21 + remainder 1

21/2 = 10 + remainder 1

10/2 = 5 + remainder 0

5/2 = 2 + remainder 1

2/2 = 1 + remainder 0

1/2 = 0 + remainder 1
```

So 43 in decimal is 101011 in binary. By swapping all occurrences of 10 with b in our previous method we create a function which converts from a decimal number n to a number in base b (2<=b<=10):

```
public int fromDecimal(int n, int b) {
  int result = 0;
  int multiplier = 1;

  while (n & gt; 0) {
    result += n % b * multiplier;
    multiplier *= 10;
    n /= b;
  }

  return result;
}
```

If the base b is above 10 then we must use non-numeric characters to represent digits that have a value of 10 and more. We can let 'A' stand for 10, 'B' stand for 11 and so on. The following code will convert from a decimal to any base (up to base 20):

```
public String fromDecimal2(int n, int b) {
  String chars = "0123456789ABCDEFGHIJ";
  String result = "";
  while (n & gt; 0) {
    result = chars.charAt(n % b) + result;
    n /= b;
  }
  return result;
}
```

In Java there are some useful shortcuts when converting from decimal to other common representations, such as binary (base 2), octal (base 8) and hexadecimal (base 16):

```
Integer.toBinaryString(n);
Integer.toOctalString(n);
Integer.toHexString(n);
```

FRACTIONS AND COMPLEX NUMBERS

支持

Fractional numbers can be seen in many problems. Perhaps the most difficult aspect of dealing with fractions is finding the right way of representing them. Although it is possible to create a fractions class containing the required attributes and methods, for most purposes it is sufficient to represent fractions as 2-element arrays (pairs). The idea is that we store the numerator in the first element and the denominator in the second element. We will begin with multiplication of two fractions a and b:

```
public int[] multiplyFractions(int[] a, int[] b) {
  int[] c = {
     a[0] * b[0],
     a[1] * b[1]
  };
  return c;
}
```

Adding fractions is slightly more complicated, since only fractions with the same denominator can be added together. First of all we must find the common denominator of the two fractions and then use multiplication to transform the fractions such that they both have the common denominator as their denominator. The common denominator is a number which can divide both denominators and is simply the LCM (defined earlier) of the two denominators. For example lets add 4/9 and 1/6. LCM of 9 and 6 is 18. Thus to transform the first fraction we need to multiply it by 2/2 and multiply the second one by 3/3:
`4/9 + 1/6 = (4*2)/(9 * 2) + (1 * 3)/(6 * 3) = 8/18 + 3/18`
Once both fractions have the same denominator, we simply add the numerators to get the final answer of 11/18. Subtraction is very similar, except we subtract at the last step:
`4/9 - 1/6 = 8/18 - 3/18 = 5/18`
Here is the code to add two fractions:

```
public int[] addFractions(int[] a, int[] b) {
  int denom = LCM(a[1], b[1]);
  int[] c = {
     denom / a[1] * a[0] + denom / b[1] * b[0],
     denom
  };
  return c;
}
```

Finally it is useful to know how to reduce a fraction to its simplest form. The simplest form of a fraction occurs when the GCD of the numerator and denominator is equal to 1. We do this like so:

```
public void reduceFraction(int[] a) {
  int b = GCD(a[0], a[1]);
```

支持

```
  a[0] /= b;
  a[1] /= b;
}
```

Using a similar approach we can represent other special numbers, such as complex numbers. In general, a complex number is a number of the form a + ib, where a and b are reals and i is the square root of -1. For example, to add two complex numbers m = a + ib and n = c + id we simply group likewise terms:

```
m + n
= (a + ib) + (c + id)
= (a + c) + i(b + d)
```

Multiplying two complex numbers is the same as multiplying two real numbers, except we must use the fact that i^2 = -1:

```
m * n
= (a + ib) * (c + id)
= ac + iad + ibc + (i^2)bd
= (ac - bd) + i(ad + bc)
```

By storing the real part in the first element and the complex part in the second element of the 2-element array we can write code that performs the above multiplication:

```
public int[] multiplyComplex(int[] m, int[] n) {
  int[] prod = {
    m[0] * n[0] - m[1] * n[1],
    m[0] * n[1] + m[1] * n[0]
  };
  return prod;
}
```

CONCLUSION

In conclusion I want to add that one cannot rise to the top of the topcoder rankings without understanding the mathematical constructs and algorithms outlined in this article. Perhaps one of the most common topics in mathematical problems is the topic of primes. This is closely followed by the topic of bases, probably because computers operate in binary and thus one needs to know how to convert from binary to decimal. The concepts of GCD and LCM are common in both pure mathematics as well as geometrical problems. Finally, I have included the last topic not so much for its usefulness in topcoder competitions, but more because it demonstrates a means of treating certain numbers.

支持

88 3