Advanced Computer Contest Preparation

Lecture 3

NUMBERS AND NUMERIC ALGORITHMS

Contents

- Floating Point Numbers
- Binary
- Data Type Limits
- Modulo Arithmetic
- Quick Power
- Primality Test
- Greatest Common Denominator
- Bases
- Infinity

Floating Point Numbers

Floating Point

- A way of representing real numbers
- In C++, **`float`**, **`double`**, and **`long double`** are floating point numbers
- A floating point number consists of a significand and an exponent in some fixed base (normally 2, 10, or 16)
  - significand * $base^{exponent}$
  - $3.4529 = 34529 * 10^{-4}$
- Significand has fixed amount of significant digits it can represent

Floating Point Usage

- Floating point numbers are not always accurate
- Errors in calculations can result from rounding errors
- Avoid using floating point numbers if possible
  - For example, store the square of distance between points rather than the distance
  - Square of distance will always be integer if points consist of integer coordinates

Floating Point Usage Tips

- Floating point numbers are sometimes unavoidable
- `double` is usually good enough if floating point numbers are necessary

Floating Point Usage Tips

- Printing floating point numbers with a number of digits after the decimal
  - Option 1: **`printf`** can take a precision modifier (**`.x`**), specifying the number of digits after the decimal
    - **`printf("%.6lf",myDouble);`**
  - Option 2: Use **`cout.precision()`** and **`std::fixed`**
    - **`std::cout.precision(6);`**
    - **`std::cout << std::fixed << myDouble;`**

Floating Point Usage Tips

- Error
  - Problems might sometimes state that an answer may have an absolute or relative error of less than 10^{-x}
    - e.g. 10^{-6}
  - Your answer may not differ from the amount given
    - Absolute = |your answer – judge answer|
    - Relative = Absolute error / |judge answer|
  - Generally, include at least x digits after the decimal in your answer

Floating Point Usage Tips

- Epsilon
  - An upper bound on errors due to rounding
    - How far a result can differ from being correct
  - Commonly used to compare floating point numbers
    - e.g. `eps = 1e-9`
    - `if (float1 == float2 ± eps)` becomes:
    - `if (abs(float1 - float2) <= eps)`

Binary

Binary Representation

- Computers store things in memory in binary
- We can use binary operations to manipulate numbers since they are stored in binary
- Binary operations are very fast and are used in advanced concepts such as state compression

Binary Operations – AND (&)

- Takes the corresponding bits of two numbers and performs the AND function on each pair, forming a new number

```
7 & 13 = 5
 00111
&01101
 00101
```

&	0	1
0	0	0
1	0	1

Binary Operations – OR (|)

- Takes the corresponding bits of two numbers and performs the OR function on each pair, forming a new number

```
5 | 9 = 13
  00101
 |01001
  01101
```

|	0	1
0	0	1
1	1	1

Binary Operations – XOR (^)

- Takes the corresponding bits of two numbers and performs the XOR function on each pair, forming a new number

```
7 ^ 13 = 10
 00111
^01101
 01010
```

^	0	1
0	0	1
1	1	0

Binary Operations – Complement (~)

- Flips all of the bits of a single number

```
  ~13
= ~(0...0001101)
= 1...1110010
= -14
```

Binary Operations – Left Shift (<<)

- Shifts all of the bits of a number to the left by a specified amount
  - Same effect as doubling that many times

```
  11 << 2
= 0...0001011 << 2
= 0...0101100
= 44
```

Binary Operations – Right Shift (>>)

- Shifts all of the bits of a number to the left by a specified amount
  - Same effect as halving that many times

```
  45 >> 3
= 0...101101 >> 3
= 0...000101
= 5
```

Data Type Limits

Data Type Limits

- Computers are not infinite!
- All data types have a limit as to what they can hold

Data Type Limits – Integer

- Integer data types have different sizes
  - e.g. C++ has:
    - `short` (2 bytes, 16 bits)
    - `int` (4 bytes, 32 bits)
    - `long long` (8 bytes, 64 bits)
  - CAUTION! `long` in C++ can be either 4 bytes (32 bits) or 8 bytes (64 bits) depending on computer, so avoid using it

Data Type Limits – Integer

- Given the size of your integer data type, you can determine its largest/smallest values
- The largest number you can hold is the largest binary number with the number of digits equal to the number of bits
- A signed integer (integer data types in C++ are by default signed) has its leftmost bit representing positive/negative and does not count as a usable digit
- **`signed int`** (32 bit) limits in C++: -2^{31} to $2^{31}-1$
- **`unsigned int`** (32 bit) limits in C++: 0 to $2^{32}-1$

Data Type Limits – Floating Point Numbers

- Floating point number implementation can vary depending on language
- Consult your language specs for the limits of floating point numbers
  - e.g. C++ **`double`** limit is 1.7E +/- 308 with 15 (base 10) digits of precision
  - A double can be this large/small, but at maximum, will have 15 significant digits

Overflowing

- Going over the limits of a variable is called **overflowing**
- The desired result will differ from the actual result since bits are discarded
- Never overflow unless the problem **specifically tells you to do so**
- If you suspect that `int` will over flow, switch to `long long`

Modulo Arithmetic

Modulo Operation

- The modulo operation (%) computes the remainder of the division of two integers
  - e.g. 5 % 3 = 2 since (5/3 = 1 R 2)
- A problem may ask you to output your answer modulo some large number
  - Typically $10^9 + 7$, which is a prime
- Reason is that your number may become very large and overflow your variables

Modulo Arithmetic – Addition and Subtraction

$$(A \pm B) \bmod M = (A \bmod M \pm B \bmod M) \bmod M$$

- Whenever you add or subtract, mod by M afterward
- Note: if you subtract, your number might become negative
  - If you are sure that the final answer is positive and your result is negative, add M

Modulo Arithmetic – Multiplication

$$(A * B) \bmod M = ((A \bmod M) * (B \bmod M)) \bmod M$$

- Multiplication is similar to addition and subtraction
  - Whenever you multiply, mod by M afterward

Modular Arithmetic – Example

```
int mod = 1000000007;
int num1 = 0, num2 = 0, num3 = 1;
for (int i = 0; i < N; i++){
      num1 += arr[i];
      num1 %= mod;
      num2 -= arr[i];
      num2 %= mod;
      num3 *= arr[i];
      num3 %= mod;
}
num1 -= 10 //assume actual value is positive
if (num1 < 0) num1 += mod;
```

Modulo Arithmetic – Division

$$(A \div B) \bmod M \neq ((A \bmod M) \div (B \bmod M)) \bmod M$$

- Division is not as straight forward as other operations
- Counter example:

$M = 1{,}000{,}000{,}000$

$(1{,}000{,}000{,}002 / 2) \% M = 500{,}000{,}001$

$(1{,}000{,}000{,}002 \% M) / (2 \% M) = 2/2 = 1$

Modular Arithmetic – Division

- Recall dividing fractions – we do not actually divide fractions; we multiply by its reciprocal
  - *fraction * reciprocal = 1*
- Similarly, instead of dividing by a number, we multiply by its inverse, where
  - *(number * inverse) mod M = 1*

Modular Inverse

- Given a number, how can its inverse be found?
  - $(number * inverse) \bmod M = 1$
- Using Fermat's little theorem, it is true that
  - $a^{p-1} \bmod p = 1$, if p is prime
- If we manipulate and let $a = number$,
  - $number * number^{p-2} \bmod p = 1$, if p is prime
- Therefore, the inverse of a number is $number^{M-2}$ **if M is prime**
- This is why M is often $10^9 + 7$, which is prime
- Finding the inverse when M is not prime is more difficult and might not exist
  - e.g. No integer x exists that satisfies $12*x \bmod 10 = 1$

Modular Arithmetic – Division

- To conclude, if M is prime,

$$(A \div B) \bmod M = ((A \bmod M) * (B^{M-2} \bmod M)) \bmod M$$

Modular Division – Example

```
int mod = 17;
int num = some_value % mod;
int div = 3;
num *= (int)pow(div,mod-2) % mod;
num %= mod; //num = (some_value/div)%mod
```

Quick Power

Exponentiation

- Multiplication is built in the processor
  - i.e. multiplication is very fast
- Exponentiation is **NOT** built in
- What is the most efficient way to get x^y, where x and y are integers?
  - C++ `pow()` cannot be used since it returns a `double`, which should be avoided unless necessary

Brute Force Exponentiation

```
int ans = 1;
while(y--){
      ans *= x;
}
```

What is the time complexity of this code?

O(N)

Quick Power Algorithm

- An $O(\log N)$ method of producing the answer to x^y
- Instead of decreasing the value of y by 1 every time, we can halve y every time and multiply our current answer by itself (rather than the oritinal value of x)
- Also called **exponentiation by squaring**

Quick Power Code

```
int pow(int x, int y){
      if (y == 0) return 1;
      if (y % 2 == 0) return pow(x*x,y/2);
      else return x*pow(x*x,y/2);
}
```

Quick Power Uses

- Since exponentiation can form very large numbers, use modulo if necessary
  - Perform after every single multiplication
- Used when any integer exponentiation is required (avoid built-in pow() method if possible)
- Used when modular division is necessary
  - Recall that modular inverse is $number^{M-2}$

Primality Test

Primality Test

- A method that checks if a number is prime or not.
- Two well-known methods:
  - Check if a single integer is prime
  - Generate a list of primes starting from 2 (Sieve of Eratosthenes)

Single Integer Primality Test

- Get rid of simple cases:
  - Integer is less or equal to 1 (not prime)
  - Integer is 2 or 3 (prime)
  - Integer is divisible by 2 or 3 (not prime)
- Note that all primes except 2 and 3 can be expressed as $6k \pm 1$
- The largest factor we need to check is $\sqrt{N}$
- Time complexity: $O(\sqrt{N})$, where N is the number to check

Single Integer Primality Test – Code

```
bool isPrime(int n){
      if (n <= 1) return 0;
      if (n <= 3) return 1;
      if (n%2==0 || n%3==0) return 0;
      for (int i = 5; i*i <= n; i+=6){
            if (n%i==0 || n%(i+2)==0)
                  return 0;
      }
      return 1;
}
```

Sieve of Eratosthenes

- A technique that will generate prime numbers starting from 2
- Uses a boolean array to keep track of whether or not a number is prime (true means prime)
- Initially, all numbers are true except 1
- Starting from 2, set all multiples of 2 (except 2) to false
- Repeat with remaining prime numbers
- If our current prime is x, we can start setting numbers to false from x^2 since multiples beforehand must have already been set to false
- Time complexity: $O(N \log(\log N))$, N is the largest number to check

Sieve of Eratosthenes – Code

```
bool isPrime[lim];
memset(isPrime,isPrime+lim,1); //set values to true
isPrime[0] = isPrime[1] = 0;
for (int i = 2; i < lim; i++){
      if (isPrime[i]){
            for (int j = i*i; j < lim; j++){
                  isPrime[j] = 0;
            }
      }
}
```

Greatest Common Denominator (GCD)

Greatest Common Denominator

- The Greatest Common Denominator (GCD) of two integers is another integer that can evenly divide both of them.
  - e.g. $GCD(12,16) = 4$
- GCD is required for some problems
- Lowest Common Multiple (LCM) requires GCD since $LCM(x,y) = x*y/GCD(x,y)$
- To find the $GCD(x,y)$ we could check all numbers from 2 to $\sqrt{min(x,y)}$, but this is method is $O(\sqrt{N})$

Euclidean Algorithm

- The Euclidean Algorithm finds the GCD of two integers in $O(\log N)$ time, which is better than $O(\sqrt{N})$
- Relies on the modulo operation

Euclidean Algorithm – Code

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

Bases

Built-in Bases

- C++ allows you to read/write numbers in 3 different bases by default:
  - Base 10, 8, and 16
- Base 8 numbers are prefixed by “0”
  - e.g. 0136726
  - Every digit represents 3 bits
- Base 16 numbers are prefixed by “0x”
  - e.g 0x3afdebc6
  - Every digit represents 4 bits
- NOTE: Without external libraries or methods, numbers cannot be read/written in binary

Infinity

Infinity

- Integer data types in C++ have no way of representing infinity
- Floating point numbers do have a way of representing infinity (`inf` and `-inf`)
  - You can set such a number to infinity by:
    - Dividing a non-zero number by zero
    - Using the `INFINITY` macro (must import cmath)

Integer Infinity

- Typically, an integer representing infinity is just a very large value
- This value should not be able to be reached from other computations
  - e.g. cannot form by adding or multiplying the numbers given in the problem
- For a C++ `int`, infinity is normally either 0x3f3f3f3f or 2e9
  - 0x3f3f3f3f can be added to one another without overflowing
  - Easy to fill a block of memory (e.g. array) with 0x3f3f3f3f using `memset`
    - `memset(ar,0x3f,sizeof(ar))`

THANK YOU!