Computational Number Theory

Stuff with integers.

By Yaseen's brother

Converting to different bases

Is that even number theory?

When converting form base i to base j, we can use base 10 as in intermediary step.

The reason for this is it is conceptually easier to go to base 10.

Base i to base 10

Example:

Firstly we are going to use the following format for number in a certain base

A number in base 13 would look like this:

[1 , 2, 12, 10]

The first digit is the units digit. Coding it like this makes it easier to iterate over.

The above example in base 10 is the following:

$1*13^0 + 2*13^1 + 12*13^2 + 10*13^3 = 24025$ (base 10)

Notice that:

$[X_0, X_1, X_2, ...]$ (base i)

$= X_0 * i^0 + X_1 * i^1 + X_2 * i^2 + ...$ (base 10)

With this knowledge we can write the following code:

Code for base i to base 10

```
int t_base_10 (const vector<int> num,const int c_base){
        //num => the number you want to change to base 10
        // c_base => the base of num
        int result = 0;
        //result => what num is in base 10
        int multi = 1;
        for (int i = 0; i < num.size(); ++i){
                result += num[i] * multi;
                multi *= c_base;
        }
        return result;
}
```

Base 10 to base j

First we find the highest power of j that divides the number we want to convert.

If n is the number then the highest power is:

floor(Log_j n)

We then run through each power of j from floor(Log_j n) to 0.
In each iteration we do the following:

Find the multiple of j to some power that fits into n. We then write down the multiple and subtract the product from n.

Code for base 10 to base j

```
vector<int> t_base_i ( int num,const int c_base){
        //log(x)/log(y) = log(y,x)
        //int rounds down
        int h_pow = log(num)/log(c_base);
        int divider = pow(c_base, h_pow);
        vector<int> result;
        for (int i = 0; i < h_pow + 1; ++i){
                result.push_back(num/divider);
                num = num % divider;
                divider /= c_base;
        }
        //NOTE: result is in reverse
        reverse(result.begin(), result.end());
        return result;
}
```

Finding Prime numbers

Prime numbers are numbers with only 2 factors: 1 and itself

There are two main methods:

Trial division

Or

Sieve of Eratosthenes

Trial Division

It is the simplest method.

Check all numbers below the prime (excluding 1) and check if any number divides it.

If a number divides it, it is not prime.

Otherwise it is prime.

Things to Note:

You only need to check primes below it.

You only need to check up to the square root of the number

This method is often too slow as it takes roughly $O(n^{0.5})$
or
$O((n/\ln (n))^{0.5})$ if you are only checking primes below it. For each iteration.

Meaning if you calculate all primes below n with this method it will take
$O(n^{3/2})_{ish}$ time.

Sieve of Eratosthenes

This finds primes up to a given n
Time complexity : O(n log log n)

Step 1: generate a list of numbers up to n, excluding 1. Mark everything in the list as a prime.

Step 2: process the first element.

Step 3: if the number you iterate over is prime add it to your prime list. And do the following. If it is not prime skip to the last step.

Step 4: mark all multiples of that prime as non-prime.

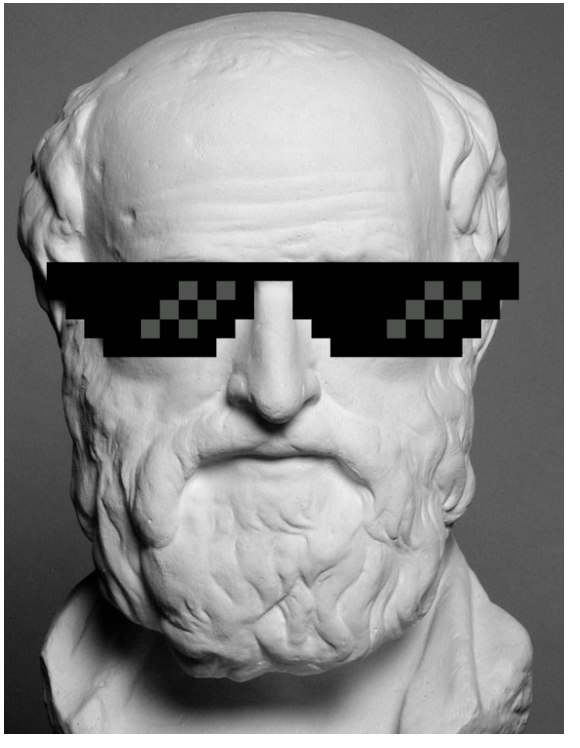
Step 5: process the next element

This will give you all the primes under n.

You can make this code much more efficient:

Efficiency 1: generate in your starting list only odd numbers your prime list should thus start with 2 in it in the beginning.

Efficiency 2: when marking off numbers as non prime, start at the square of the prime. e.g. if you are processing 11, start at 11*11 = 121 and mark off from there.



Sieve of EraWhatSitFace code

```
vector<int> primes = {2};

int con (int index){
        return (index*2 + 3) ;
}
int noc (int num){
        return ((num - 3)/ 2);
}

int sieve (int limit){
        vector<bool> buff;
        for (int i = 3; i < limit; i+= 2){
                buff.push_back(true);
        }
        for (int i = 0; i < buff.size(); ++i){
                if (buff[i] == true){
                        primes.push_back(con(i));
                        for (int j = noc(con(i)*con(i)); j < buff.size() ; j+= con(i)){
                                buff[j] = false;
                        }
                }
        }
        return primes.size();
```

Note: vector<int> primes is global.
Since I'm using bools instead of int. I convert the index to numbers with the function "con"
And numbers to index with "noc".
Since primes is global you also don't have to return anything of use in the function.

GCD and LCM

a%b = c
Means a divided by b leaves remainder c

GCD = greatest common divisor

LCM = lowest common multiple

W.L.O.G a >= b
If c is the GCD(a ,b). then c divides a and c divides b.
c also divides a – b. From this we see that
c divides a % b.

Thus GCD(a ,b) = GCD (a%b,b) and GCD(a,0) = a

We can use this to calculate the GCD of any two numbers

We can find the LCM with the GCD-LCM link which states:

LCM(a,b) = (a*b)/GCD(a,b)

Code for GCD and LCM

```
int GCD(int a,int b){
        if (a < b){
                int temp;
                temp = a;
                a = b;
                b = temp;
        }
        if (b == 0){
                return a;
        }
        else if (b == 1){
                return 1;
        }
        else{
                return GCD(a%b,b);
        }
}
int LCM (int a, int b){
        return (a*b)/GCD(a,b);
}
```

Modular inverses

What is a modular Inverse?

If you have an a and b such that GCD(a,b) = 1

Then k is the modular inverse of a MOD b iff

(a*k) %b == 1

e.g. the modular inverse of 7 MOD 31 is 9.

7*9 = 63
63 % 31 = 1

How to calculate modular inverses?

We first need to learn Euclid's division algorithm and the extended algorithm

Euclid's Division algorithm is what we used to calculate the GCD of two numbers, except written out properly.
It's best to show by example.

EXAMPLE 1:
a = 78 , b = 30

78 = 2(30) + 18

30 = 1(18) + 12

18 = 1(12) + 6

12 = 2(6) + 0

Therefore GCD(78,30) = 6

EXAMPLE 2:
a = 125 , b = 36

125 = 3(36) + 17

36 = 2(17) + 2

17 = 8(2) + 1

2 = 2(1) + 0

Therefore GCD(125,36) = 1

Extended Euclid's Division algorithm

It's the same thing but backwards.

Normal

a = 125 , b = 36

125 = 3(36) + 17

36 = 2(17) + 2

17 = 8(2) + 1

Extended

1 = [1](17) + [-8](2)
1 = [1](17) + [-8](36 – 2*17)
1 = [-8](36) + [17](17)
1 = [-8](36) +[17](125 – 3(36))
1 = [17](125) + [-59](36)

Thus the modular inverse of 125 MOD 36 is 17

BUT THERE'S A BETTER WAY THAT ISN'T CONFUSING AS HELL AND PRONE TO ERRORS!!!!

Do the division algorithm but only record the coefficients (the thing not in brackets)

a = 125 , b = 36

125 = 3(36) + 17 p[3] = 3

36 = 2(17) + 2 p[1] = 2

17 = 8(2) + 1 p[0] = 8

Then do the following recursion

x = 0, y = 1

(x, y)
(y, x – y*p[i])

e.g

(0, 1)
(1, 0 -1*8)
(1, -8)
(-8, 1 -(-8)*2)
(-8, 17)
(17, -8 -(17)*3)
(17, -59)

Which means 1 = 17(125)-59(36)

Code for using the previous method

```
vector<int> mod_inv (int a, int b){
        //assumes GCD(a,b) == 1
        vector<int> record;
        int temp;
        if (a < b){
                temp = a;
                a = b;
                b = temp;
        }
        int rem_a = a;
        int rem_b = b;
        while (a%b != 1){
                record.push_back(a/b);
                temp = a % b;
                a = b;
                b = temp;
        }
        record.push_back(a/b);
        int x, y;
        x = 0;
        y = 1;
        for (int i = 0; i < record.size(); ++i){
                temp = y;
                y = x - y*record[record.size() - i - 1];
                x = temp;
                //cout << x << " " << y << " " << record[i] <<"\n";
        }
        vector<int> ans = { x, y};
```

Bezout coefficients

Efficient exponentiation

```
long long e_pow(long long num, long long exp, long long mod){
        long long temp;
        //Note: Number theory dictates that you can mod the base.
        num = num % mod;
        if (exp == 1){
                // a^1 = a
                //This is the base case
                return num;
        }
        else if (exp % 2 == 1){
                // a^2k+1 = a * (a^k)^2
                temp = e_pow(num, ((exp - 1 )/ 2),mod) % mod;
                return (num * temp*temp) % mod;
        }
        else{
                // a^2k = (a^k)^2
                temp = e_pow(num, (exp/2),mod) % mod;
                return (temp*temp) % mod;
        }
}
```