Table of Contents:

Proof 1 : A^C ≡ B^C (mod N) For any positive integer C

Since A ≡ B (mod N)

A and B are in the same modulo set.

Then:

(A^C) % N =((A%N)^C)%N
and
(B^C) % N =((B%N)^C)%N

because we know that (A*A) % N = ((A%N)*(A%N))%N
which is true because A and A%N are in the same modular set.

This basically means that, instead of calculating the exponent first,
you can calculate the mod of the base, and raise that to the N, then take the mod of that.

With that said, since A%N = B%N, then
((A%N)^C)%N = ((B%N)^C)%N
therefore
(A^C) % N =(B^C) % N

Proof 1 version 2:

A^C ≡ B^C (mod N) For any positive integer C

Since A ≡ B (mod N)

A and B are in the same modulo set.

We know that

A * C ≡ B * C (mod N)

If A’ = A%N

Then A ≡ A’(mod N) since A and A’ are in the same modulo set.

Then A*A ≡ A’*A(mod N).

If B’ = A%N

Then B ≡ B’(mod N) since B and B’ are in the same modulo set.

Then B*B ≡ B’*B(mod N).

It follows that
B*B*B ≡ B’*B’*B(mod N)
A*A*A ≡A’*A’*A(mod N)

If A * C ≡ B * C (mod N)
And C = B’ * B’ = A’ * A’ (because A’ = B’ since A≡B(mod N))
Then
A*C ≡ B*C (mod N)
Therefore
A*A*A ≡ B*B*B (mod N)

A^3 ≡ B^3 (mod N)

If A^C ≡ B^C (mod N)
Then A^(C+1) ≡ B^(C+1) (mod N)

(A^C)*A’ ≡ (B^C)*B’ (mod N)
since A’ = B’

then A^(C+1) = B^(C+1)

This proves the second, third and all further cases.

MODEXP Function:

```
int modexp(int x, int n, int p) {
        if (n==0) return 1;
        if (n==1) return x%p;
        if ( (n % 2) == 0 ) // n is even
                return modexp((x*x)%p,(n/2),p);
        else // n is odd
                return (modexp((x*x)%p,(n/2),p)*x)%p;
}
```

MODEXP Algorithm Explanation and Analysis

The modexp function exploits the fact that:

X^N = (X^2)^(N/2) i.e. X^4 = (X^2)^2

And the associative modular property:

X^N % P = (X^(N-1) * (X % P)) % P

Combining to:

X^N % P = [((X^2) % P)^(N/2)] % P

First, for exponents, you can square the inside and half the exponent, which reduces the number of multiplications from n to $\log_2 n$. This speeds the process up.
Second, you can peel off an x or two, and mod them, then multiply the mod back to the equation, without changing the final results. This allows us to take the mod of huge exponents without ever getting a huge intermediate value.

Since N is halved each iteration,

The number of recursions is $\log_2 N$.

T(N) = O ($\log_2 n$) (Growth Rate)

Because

F(n) = F(n/2) + O (C) , Let O(C)= k

F(0) = 0
F(1) = k + F(0) = k
F(2) = k + F(1) = k + k
F(4) = k + F(2) = k + k + k
F(4) = k + F(4) = k + k + k + k
F(4) = k + F(8) = k + k + k + k + k
F(4) = k + F(16) = k + k + k + k + k + k
F(N) = k + F(N/2) = k $\log_2 N$.

F(n) = F(n/2) + O (C)

a = 1, b = 2 , k = 0 , a = b^k = 1

when a = b^k
$T(n) = O(n^k \log_b n)$
$= O(n^0 \log_2 n)$
$= O(\log_2 n)$

Example:

To compute MODEX(X,62,P)
The algorithm does the following calculations:

X^3 =((X^2)%P*X) %P
X^7 =(((X^3)^2)%P*X) %P
X^15 =(((X^7)^2)%P*X) %P
X^31 =(((X^15)^2)%P*X) %P
X^62 =(((X^31)^2)%P) %P

9 mutiplications, 9 mods

This is about $4(\log_2 N)$ calculations ($2 \log_2 N$ mult. And $2 \log_2 N$ mods) verses N.
This equates to a growth rate of $O(\log_2 N)$.

MODEXP Proof 1

Suppose the N is even.
Show that if MODEXP(X,N/2,P)=Beta
Then (Beta * Beta) % P = MODEX(X,N,P)

Suppose MODEXP(X,N/2,P) = Beta

Then X^(N/2) % P = Beta

We want to show that
(Beta * Beta) % P = MODEXP(X,N,P)

Beta = (X^(N/2)) % P

Let A’ = A%P

Then A = A’ mod P , since A and A’ are in the same modulo set.

Then by property 3 (A^C = B^C (mod P))

A * A = A^2 (mod P)
A^2 = A’^2 (mod P)
A’^2 = (A’ * A’)(mod P)

Line * : then (A * A) % P = (A’ * A’) % P = (A%P * A%P) % P

Let A = X^(N/2)

Then (A * A) % P = (X ^ N) % P

Substituting in A = X^(N/2)
(X ^ N) % P = (X ^ (N/2) * X ^ (N/2)) % P

Using Line * :
(X ^ (N/2) * X ^ (N/2)) % P = (X ^ (N/2) % P * X ^ (N/2) % P) % P = (Beta * Beta) % P

So:
X ^ N % P = (Beta * Beta) % P

And

MODEXP(X,N,P) = (Beta * Beta) % P

MODEX Proof 2

Suppose the N is odd.
Show that if MODEXP(X,N/2,P)=Beta
Then (X * Beta * Beta) % P = MODEXP(X,N,P)

We know that: **(Beta * Beta) % P = MODEXP(X,N,P) when N is even.**
While Beta = (X^(N/2)) % P

We will call this Neven
Therefore,
Beta = (X^(Neven/2)) % P
And
MODEXP(X,Neven,P) = X ^ Neven %P = (X ^(Neven/2)%P * X ^(Neven/2)%P) %P = (Beta*Beta)%P

More simply: X ^ Neven = (Beta*Beta) (mod P)

Definition 1 states that A*C = B*C(mod N)
therefore
X*(X ^ Neven) = (X*Beta*Beta) (mod P)

X ^ (Neven+1) = (X*Beta*Beta) (mod P)

So if N = Neven+1 then

X ^ N = (X*Beta*Beta) (mod P)

Modexp(X,N,P) = X ^ N % P = (X*Beta*Beta) % P

GCD Algorithm :

```
int gcd( const int & A,  const int & B) {
if (B != 0)
{
return gcd(B,A%B);
}
return A;
}
```

GCD Algorithm Explanation and Analysis:

The gcd recursive algorithm uses the fact that gcd(A,B) = gcd(A,alpha), where alpha = A % B.
The algorithm recursively calls gcd(B,alpha) until alpha = 0. When alpha = 0, we know that B is the gcd, because B is divisible by B, and 0 is divisible by anything, including B.

GCD Algorithm Analysis:

The algorithm runs in $O(\log_2 A + \log_2 B)$ time.

If we look at this function:

F(n) = F(gcd(B,A%B)) + O(C)
K = 0, a = 1, b = ?

It looks like b might be 2.

We can see that A%B < A/2

Because the largest value of A%B occurs when B is one more than half of A. In that case,
A % B = B – 1 = A – B

Sub Proof: AmodB < A/2

Case 1: If A>B then AmodB < A/2, since the remainder is smaller than B.

Case 2: If B>A/2 then B goes into A once with a remainder of M – N < M/2.

Therefore, after 2 iterations the remainder is at most half its original value, thus

$$T(N) = 2 \log_2 N = O(\log_2 N)$$

N could be A or B, whichever takes the longest to reduce to 0.
Therefore, $T(N) = \max(2 \log_2 A, 2 \log_2 B)$.

An easy upper bound for this is just to add them as...

$T(N) = \max(2 \log_2 A, 2 \log_2 B) \leq 2 \log_2 A + 2 \log_2 B$.

Thus,

$T(N) = O(\log_2 A + \log_2 B)$

GCD Proof 1 gcd(A,B) = gcd(A-B,B)

gcd(A,B) = gcd(A-B,B) = D

First we will show that if D divides A and B it also divides A-B and B.

q and r are integers such that

q = A/D r = B/D

then

(A-B) / D = (qD - rD) / D = q – r which is a positive integer when A > B.

Therefore, and common divisor of A,B is a common divisor of A-B, B.

NEXT

We want to show that there is no integer > D that divides both A-B and B.

If e is an integer, e > d, and e divides both A-B and B.

Let u be an integer such that: u = B / e

Let v be an integer such that: v = (A-B)/e

Then: B = eu and A-B = ev

A – eu = ev
A = ev + eu
A = e (v + u)
A/e = (v + u) (Note that u + v is an integer since u is and integer and v is an integer.)

Therefore e divides A, e divides B
Therefore e is the gcd(A,B) since it divides A and B and is greater than D.
This contradicts the notion that D is gcd(A,B) which we know is true.

Therefore, if D is the gcd(A-B,B) there is no e which is larger than D and is the gcd(A,B).

GCD Proof 2 gcd(A,B) = gcd(B,alpha)

A = alpha (mod B)

A - alpha are divisible by B

(A – alpha) / B = k where k is an integer

A = kB + alpha

From the last proof, we know that gcd(A,B) = gcd(A-B,B) [while A > B]

Therefore

Gcd(A,B) = gcd(kB + alpha, B) = gcd(kB + alpha - B, B)

If we apply this k times we get gcd (alpha,B)

Which is the same as gcd(B,alpha)

Because we know that gcd(A,B) = gcd(B,A)

MMI Algorithm

```
void getxy(const int & A, const int & P, int & X, int & Y) {

            if (P==0) {X=1; Y=0; return;}

            getxy(P,A%P,X,Y);

            int Xpre=X;
            X=Y;
            Y=Xpre-(A/P)*Y;
}

int mmi( const int & A,  const int & P) {
            int X=0;
            int Y=0;
            if(A>P) {
                        getxy(A,P,X,Y);
                        if(Y<0)
            return Y+A;
            else return Y;
            }
            else
                        getxy(P,A,Y,X);
                                    if(X<0)
            return X+P;
            else return X;

}
```

MMI Algorithm Explanation and Analysis

Mmi recursively calls getxy until A or B = 0.

The algorithm runs in $O(\log_2 A + \log_2 B)$ time.

If N = max (A or B)
The elements are swapped each iteration and are reduced then, only every other iteration.

Every other iteration of getxy reduces N to N % P.

We know that N < N / 2 and
$T(N) = 2 \log_2 N = O(\log_2 N)$
because:

The largest value of A%B occurs when B is one more than half of A. In that case,
A % B = B – 1 = A – B

Sub Proof: AmodB < A/2

Case 1: If A>B then AmodB < A/2, since the remainder is smaller than B.

Case 2: If B>A/2 then B goes into A once with a remainder of M – N < M/2.

Therefore, after 2 iterations the remainder is at most half its original value, thus

$T(N) = 2 \log_2 N = O(\log_2 N)$

The algorithm quits when A or B reaches 0.
Suppose A reaches zero first. Then the algorithm took no more than $2 \log_2 A$ iterations.

We know that $\log_2 A$ reduces more slowly that the mod function.
The 2 prefix exists because A is reduced every other iteration.
However, the 2 disappears when we put it in Oh notation.
i.e. $T(n) = O(\log_2 A)$

Since we don't know which hits the ground first (A or B), we use $T(n) = O(\log_2 A + \log_2 B)$.

N could be A or B, whichever takes the longest to reduce to 0.
Therefore, $T(N) = \max(2 \log_2 A, 2 \log_2 B)$.

An easy upper bound for this is just to add them.

$T(N) = \max(2 \log_2 A, 2 \log_2 B) \leq 2 \log_2 A + 2 \log_2 B$.

$T(N) = O(\log_2 A + \log_2 B)$

---

mmi Proof 1, show that for mmi(A,P) to exist, gcd(A,P) = 1

METHOD ONE:

If gcd(A , P) = N

and

A * X = 1 (mod P)

A * X – 1 = k * P (for some integer k)

Since N | P then N | k * P (| is used to mean 'divides', thus N divides P with no remainder.)

Since k * P = A * X – 1 and N | k * P

Then N | A * X – 1

Since N =gcd (A , P) then N | A then N | A * X

If N | A * X and N | A * X – 1

Then N | -1

So N must be 1 , for the condition to hold.

METHOD TWO:

suppose gcd(A,P) = n

and A = a*n, P = p*n

then

gcd(a,p) = 1

then

mmi(a,p) = x or a * x = 1 (mod p)

If A*X = 1 (mod P)

then a * n * X = 1 (mod n*p)

a*n*X % n*p is always a multiple of n

because

a*n*X – Y * n * P = n (a*X – Y * P)

therefore

A*X = n (mod P)

Another way to show this is:

a * n * x – y * n * p = 1

n (a * x – y * p) = 1

since a , x , y , and p are all integers
this only holds when n = 1 and (a * x – y * p) = 1

Therefore
a * n * X = 1(mod p * n) has no solution unless n = 1

Therefore
A*X = 1 (mod P) has no solution because A and P have a common divisor greater than 1

mmi proof 2:

if mmi(A,P) = X
Show that there must be an integer Y such than
A*X+P*Y=1

Mmi(A,P) means that

A*X % P = 1

Therefore, A*X minus some multiple of P = 1

A*X – PZ = 1

If Y = -Z then

A*X + PY = 1

mmi proof 3:
Show that :
for AX + PY = 1
another solution exists with X’ = X + k*P (where k is any integer)

mmi(A,P) = X

A*X + P*Y = 1

X’ = X + k*P

A*(X+k*P) + P*Y = 1

A*X + A*k*P + P*Y = 1

A*X + P*(Y+A*k) = 1

Therefore, another solution exists for X’ = X + k*P

Where Y’ = Y + A*k

mmi Proof 4 : show MX + NY = 1 , if N = 0

MX + 0Y = MX = 1

Y drops out of the equation so its value is irrelevant.

If M and X are integers, they must both be 1 for this to hold.
Thus, proving the relationship.

This case is analogous to the base case:

1*1 = 1 mod N

We don't know N, but we know every other element.

MMI Proof 5 : M*X + N*Y = 1, if N * X1 + R * Y1 = 1

Where R = M%N and X = Y1 and Y = X1 – [M / N] * Y1

If:
M*X + N*Y = 1

Set X = Y1 and Y = X1 – [M / N] * Y1

Then
M*Y1 + N(X1-[M/N] * Y1) = 1 (I use [ ] to indicate lower bound 'integer' division.)

M*Y1 + N*X1- N[M/N] * Y1 = 1

M*Y1 + N*X1- (M - M%N) * Y1 = 1 (Note that N[M/N] = (M - M%N)
because [M/N] = M/N - (M%N)/N)

M*Y1 + N*X1- M*Y1 + (M%N) * Y1 = 1

N*X1 + (M%N) * Y1 = 1

Since R = M%N then:

N*X1 + R * Y1 = 1

If this smaller problem has a solution and is true (which we know from the problem statement), then we have proven that the larger problem is also true.
