Competitive Programming
Class - 1

Modular Arithmetic

Modulo Operator %

- Produces Remainder of an integer division
- Cannot be applied to floating point number
- Eg.
  - 11 %7 = 4
  - 19%2 = 1
  - 13%5 = 3

Modulo Addition

- $(a + b) \% m = ?$

Modulo Addition

- **(a + b) %m = (a%m + b%m)%m**

- Example :
  - (8 + 9)%5 = (8%5 + 9%5)%5
  - 17%5 = (3+4)%5
  - 2 = 2

Modulo Subtraction

- **(a-b)%m = (a%m - b%m + m)%m**

- Example:
  - (18-7)%5 = (18%5 - 7%5 + 5)%5
  - = (3-2+5)%5
  - = (6)%5
  - = 1

Modulo Multiplication

- (a*b)%m = (a%m * b%m)%m

Modulo Multiplication

- **(a*b)%m = (a%m * b%m)%m**

- Why is expansion of modulo equations required ?
- To solve the problem of integer overflow.
- Eg. $(10^{18} * 10^{18})\%7$
- Before the modulo operator is applied , above expression will lead to int overflow

Modulo Division

- **$(a/b)\%m = (a\%m * b^{-1}\%m)\%m$**

- b^{-1} is the multiplicative inverse of b wrt m
- $(b*b^{-1})\%m = 1$
- If m is prime $b^{-1}\ \%\ m = b^{m-2}\%m$ (Proof - Fermat's Little Theorem)

Time complexity

Time complexity is the number of operations an algorithm performs to complete its task.

1)
```
for(i = 1; i <= n; i++) {
    printf("%d", i);
}
```

2)
```
for(i = 1; i <= n; i++) {
    for(int j = 1; j <= n; j++) {
        printf("%d", i);
    }
}
```

3)
```
for(int i = 1; i <= n; i++)
    for(int j = 1; j <= m; j++)
        for(int k = 1; k <= p; k++)
            printf("Hello");
```

4)
```
for(int i = 1; i <= n; i++)
    for(int j = 1; j <= n*n; j++)
        printf("%d", j);
```

Answers

1) O(n)
2) O(n^2)
3) O(n * m * p)
4) O(n^3)

Modular exponentiation

- (a^n) % m ??

Naive approach, complexity: O(n)

```c
// Program to calculate (a^n) % m
#include <stdio.h>

int main() {
    long int a, n, m, i, ans;
    ans = 1;
    // Fetching the value of base, exponent and modulus from user
    scanf("%ld %ld %ld", &a, &n, &m);
    for(i = 1; i <= n; i++) {
        ans = (ans * a) % m;
    }
    printf("%ld\n", ans);
    return 0;
}

```

Optimized approach

$X^{2n} = (X^n)^2$

$X^{2n+1} = X * (X^n)^2$

Keep on diving the exponent into two parts until exponent = 0

Recursive code for fast modular exponentiation

```c
// Program to calculate (a^n) % m
#include <stdio.h>

long int modpower(long int a, long int n, long int m) {
    if(n == 0)
        return 1;
    ll x = modpower(a, n/2, m);
    x = (x * x) % m;
    if(n & 1)
        x = (x * a) % m;
    return x;
}

int main() {
    long int a, n, m;
    // Fetching the value of base, exponent and modulus from user
    scanf("%ld %ld %ld", &a, &n, &m);
    printf("%ld\n", modpower(a, n, m));
    return 0;
}
```

Complexity: **O($\log_2$n)**

O($\log_2$(exponent)) to be precise

Greatest Common Divisor (GCD)

- gcd(a,b) ??

Naive approach, complexity: O(min(a, b))

```c
// Program to calculate gcd of two numbers
#include <stdio.h>

int min(int a, int b) {
    return (a < b) ? a : b;
}

int main() {
    int a, b, i, gcd;
    scanf("%d %d", &a, &b);
    for(i = min(a, b); i >= 1; i--) {
        if(a % i == 0 && b % i == 0) {
            gcd = i;
            break;
        }
    }
    printf("%d\n", gcd);
    return 0;
}
```

Optimal approach, Euclidean algorithm

- GCD (A, B) = GCD (B, A % B)
- Until A % B == 0

Code

```c
// Program to calculate gcd of two numbers

int gcd(int a, int b) {
    // Base case
    if(b == 0)
        return a;
    return gcd(b, a % b);
}

int main() {
    int a, b, i, gcd;
    scanf("%d %d", &a, &b);
    printf("%d\n", gcd(a, b));
    return 0;
}
```

Complexity:
$O(\log_2(\max(a, b)))$