# Basic Maths Concept

→ Extraction of digits

Q- N = 7789 , perform extraction of digits.

↳ If we do 7789 % 10 we get (9)

[→ When 7789 is divided by 10, the nearest no will be 7780, the remainder will be 9]

↳ If we do 7789 // 10 we get (778).9

integer

Now, 778 % 10 = (8)

we get.

Again divide by 10, and do modulo 10, you get (7).

77 % 10 = 7

Again divide by 10, and do modulo 10, you get (7)

7 % 10 = 7

Now 7 // 10 = (0)

→ Extraction has been done by reverse order

ie 9, 8, 7, 7.

---

Pseudocode

```
→ while (N > 0) {
    Lastdigit = N % 10;
    N = N / 10;
  }
```

# Count Digits

Pseudocode

```
cnt = 0 (counter variable)
while (N > 0)
{
  Last digit = N % 10;
  cnt = cnt + 1;
  N = N / 10;
}
print(cnt)
```

---

Another way

↳ We can say, that the no of time it is getting divisible by 10, is the count of the digit.

ex. $\log_{10}(7789)$

$\leq 3.89\ldots$

(∵ 1 is added to it

$3.89 + 1 = (4.89)$

(Integer of this : 4)

---

Code

```
#include <bits/stdc++.h>
int cnt = (int) log10(n) + 1;
```

↑ Typecasting to integer

```
return cnt;
```

Time complexity $O(\log_{10}(N))$ → how many time the loop running?

↳ the no of time it's getting divided by 10.

```
rest dgt = N.%10;
cnt = cnt+1;
```
} These are unit operations

For 7789 → it is 3.84 ie nearly 4 time

→ Whenever there is division,

If division is happening by 10, we say $\log_{10}(N)$

If division is happening by 2, we say $\log_2(N)$

" " " " 5, we say $\log_5(N)$

" whenever we are writing a logic where the no of iteration depends on division, or we are dividing, there logarithmic will come to T.C "

↳ Here T.C won't be $O(N)$

---

## Reverse of a Number

Eg: 7789 → Reverse 9877

N = 7789 % 10 = 9

$N/10$ → 778 % 10 = 8

$N/10$ → 77 % 10 = 7

$N/10$ → 7 % 10 = 7

$N/10$ → 0

## Pseudocode

```
rev N = 0
while (N > 0) {
    lastRight = N % 10;
    N = N / 10;
    revN = revN x 10 + lastRight;
}
```

## By Run

(0 x 10) + 9 = 9
(9 x 10) + 8 = 98
(98 x 10) + 7 = 987
(987 x 10) + 7 = 9877 — just what we wanted.

9 x 10 = 90, +8 → 98

---

## Palindrome of a Number

→ Any number, which on reverse, is the same number, is a palindrome number.
  ↳ Palindrome: Reverse of a Number (is same).

→ If we generate reverse of a number, and compare it with original number, and if they come out to be same,
  ↳ they are palindrome.

---

## Pseudocode

```
rev Num = 0
dup = n // storing a copy of N, so that we can use it to compare it with reverse of a number.
while (n > 0) {
    lastDigit = n % 10;
    n = n / 10;
}
```

(∵ we are doing operations with N, N becomes zero at the end)

```
revNum = revNum x 10 + lastDigit;
}
```

if (revNum == Dup) → Yes

else → No.

---

## Armstrong Number

$N = 371 = 3^3 + 7^3 + 1^3 = 371$

↳ If taking the cube of the digits of a number, and adding them up, sums up to the number itself, that is an Armstrong Number.

$N = 1634 \neq 1^3 + 6^3 + 3^3 + 4^3 = 1634$ (Not an armstrong no.)

$35 \neq 3^3 + 5^3$ (Not an armstrong no.)

---

```
int dup = n;
int Sum = 0
while (n > 0) {
int lastDigit = n % 10;
Sum = Sum + (lastDigit * lastDigit * lastDigit);
n = n / 10;
}
if (dup == Sum) → Yes
else → No
```

# Print all Divisors

Eg: (36) → print all numbers, that divide 36?

↳ 1, 2, 3, 4, 6, 9, 12, 18, 36 → completely divide 36.

‖

These are the divisors of 36

‖

You have to print all of them, in any particular order.

→ All the divisors will be between 1 to N itself.
(or factors)

i.e [1 → N]

Any anything greater than N, will never divide N.

↳ ∴ loop from 1 → N

→ if (i) is completely dividing n, then it is a factor

‖

should leave a remainder of 0.

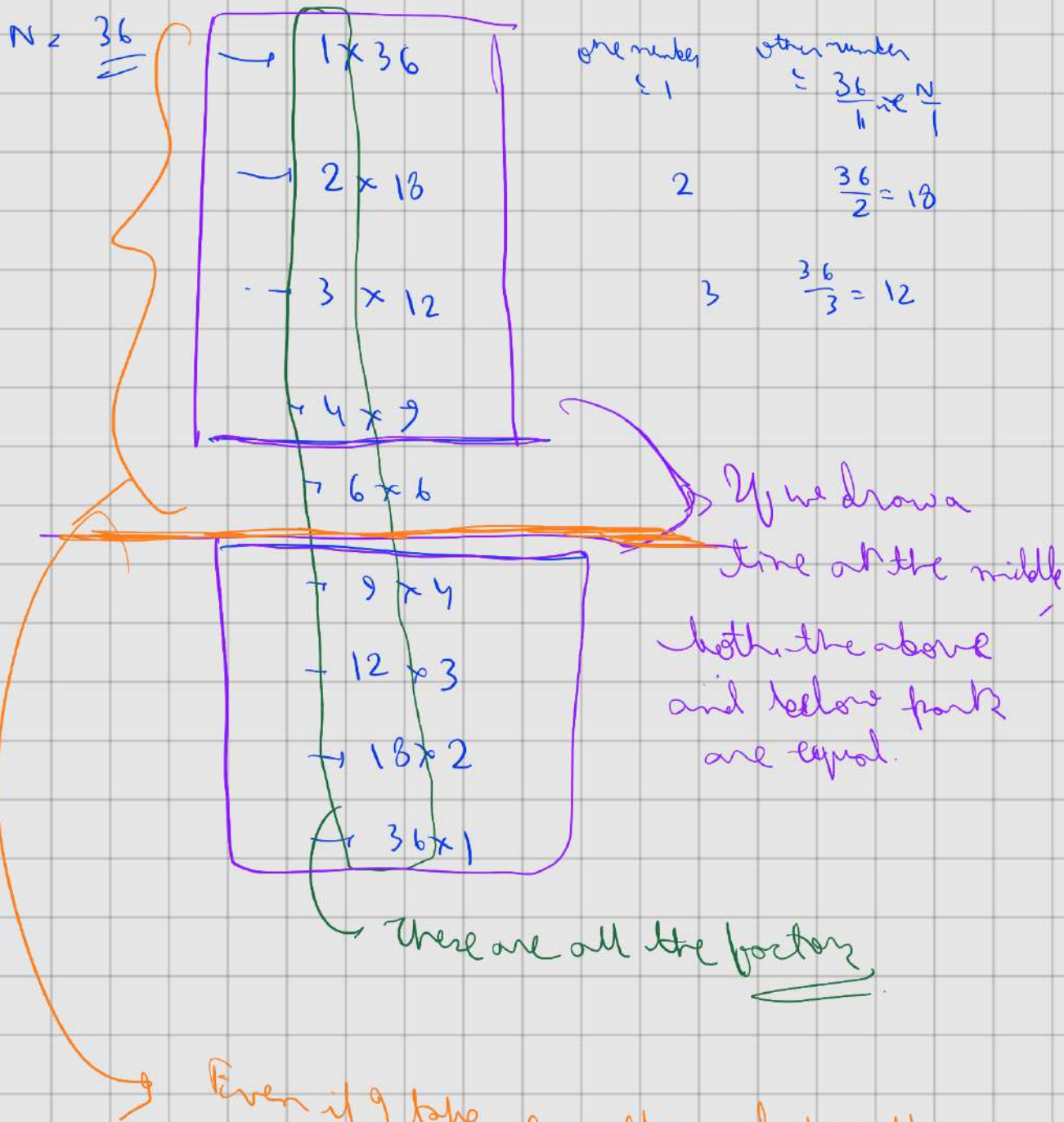
```
∴ for(i=1; i<=N; i++){
    if(N%i==0){
        print(i);
    }
}
```

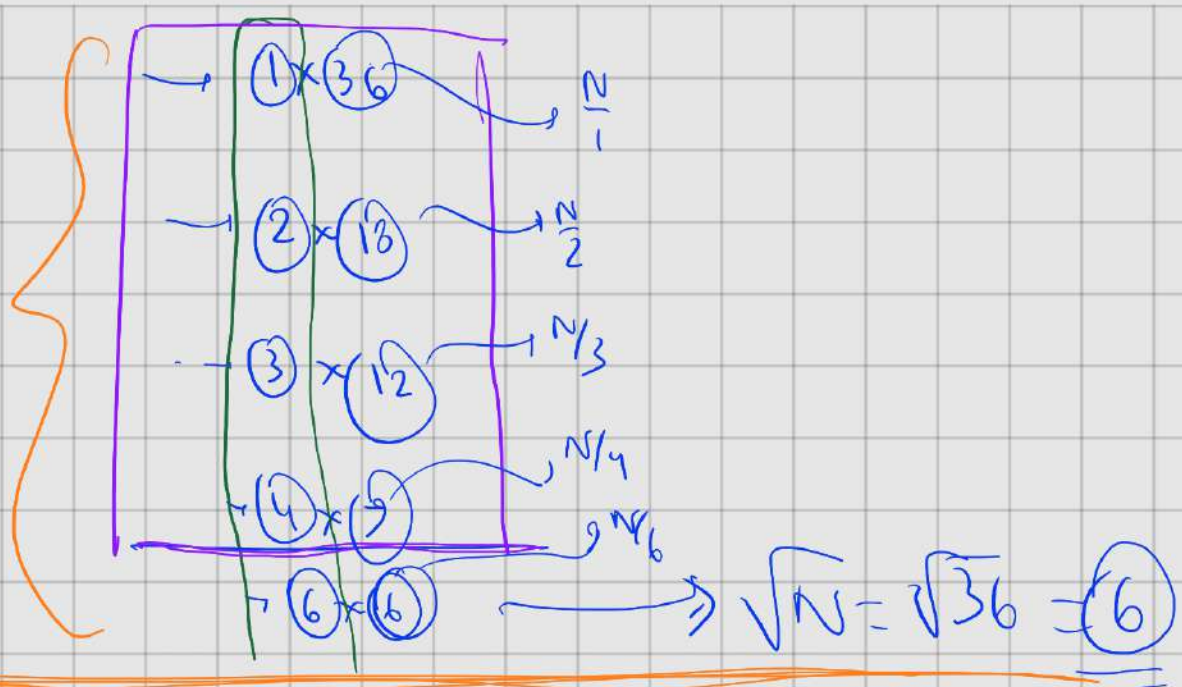
// Time Complexity: loop running from 1 → N, taking N iterations ⇒ ∴ O(N)

Improvement - 2

## Mathematical observation :-

For 36, 1 is a factor. So, 1 has to be multiplied with something in order to get 36.






These numbers will be replication of the upper half.

Even if you loop till sqrt of N, you can actually get all of the factors.

---

Pseudocodes

```
for(i=1; i <= sqrt(n); i++)
{
    if (n%i==0)   // n%i is a factor
    {
        print(i) // print 'i' or one of the factors
                 // other factor is n/i
```

```
if ((n/i) != i)
    print (n/i);
}
}
```

// making sure both n/i and i are not same:

But if, if i is 6, other factor will also be 6, then are not two different factors, so make sure $\left(\frac{n}{i}\right)$ is not equal to i

$\frac{n}{i}$ → second factor

i → one factor

The output will be

| 1 | 36 | 2 | 18 | 3 | 12 | 4 | 9 | 6 |
|---|---|---|---|---|---|---|---|---|

All the factors are printed, but not in a sorted way.

So, we can store them in a data structure, i.e. vector, since we don't know what will be the size, or no. of factors

Then store them, then sort them.

---

## Check for Prime

→ Using definition

A number that is divisible by 1 and itself

Using definition

According to this definition, 1 is a prime
↳ 1 divisible by 1
↳ 1 divisible by itself.

Correct Definition: The number that has exactly two factors, 1 and itself.

N = 11 → has factors of 1 & 11 itself
↳ Prime no.

N = 13 → has factors of 1 & 13 itself
↳ Prime no

N = 4 → Not Prime
↳ it is divisible by 1, 2, 4.

Simplest (Brute Force) method :-

```
cnt = 0
for(i=1; i<=n; i++)
{
  if(n%i==0)
    cnt++;
}
if(cnt==2) → Prime nov
else → Not a Prime no
```

↳ Time Complexity : O(N)

36 =

(1) × (36)
(2) × (18)
(3) × (12)
(4) × (9)
(6) × (6)

Every factor is the other corresponding number, with which it has to be multiplied, in order to get multiplied in order to get the number

sqrt(n) ⟶ Sqrt(n)

more efficient approach

n-2

```
cnt = 0
for(int i=1; i*i <= N; i++)
{
    if (n % i == 0)
    {
        cnt++;
        if ((n/i) != i)
            cnt++;
    }
}
if (cnt == 2) → Prime ✓
else → Not Prime ✗
```

Here Time Complexity is $O(\sqrt{n})$

# GCD / HCF

HCF → Highest Common Factor

GCD → Greatest Common Divisor

If $N_1 = 9$, $N_2 = 12$,

↳ Find the HCF or GCD, that actually divides 9 & 12.

Factors of 9 → (1), (3), 9

Factors of 12 → (1), 2, 6, 12, (3), 4

↳ There are 2 common factors.

↳ out of these two, highest is (3).

∴ GCD of 9, 12 is (3)

↳ Coz 3 is the largest number that divides 9 and 12.

Q. GCD of 11, 13

Ans (1).

(1), 11 (1), 13

↳ There will always be a GCD, coz, 1 is a number that divides every other number.

Q GCD of 20, 40

A: (20)

↳ So, for two given numbers, one of them can also be a GCD.

Brute

eg: $n_1, n_2$, $n_1 = 9$, $n_2 = 12$

gcd = 1 // we know, for any given two numbers, will always have a GCD of 1.

for(i = 1; i <= min(n1, n2); i++) {

if(n1 % i == 0 && n2 % i == 0) — runs till min(n1, n2)

gcd = i;

}

// time complexity is O(min(n1, n2))

Better

$n_1 = 20$, $n_2 = 40$

for(i = min(n1, n2); i >= 1; i--)

{

if(n1 % i == 0 && n2 % i == 0) — 20 % 20 == 0, 40 % 20 == 0

{

print(i);

break;

}

}

it is a conditional statement, not a loop

task of break is it always breaks out from the outer loop.

↳ In this way, it will turn out better for a lot of ways. But, still the worst case will be $O(\min(n_1, n_2))$

eg: $n_1 = 11$, $n_2 = 13$

↳ here,

it will loop from 11 to 1 → no matter what or $O(\min(n_1, n_2))$

→ If both the numbers have gcd as 1, then it ends up running completely.

---

Optimal

## Euclidean Algorithm

↳ If given two numbers $n_1, n_2$, the GCD of $n_1, n_2$, (whatever be the GCD)

that's equivalent, to the GCD of $(n_1 - n_2, n_2)$, where

$n_1 > n_2$

i.e

$n_1, n_2$ $\qquad$ $n_1 > n_2$

$$gcd(n_1, n_2) = gcd(n_1 - n_2, n_2)$$

$$\boxed{gcd(a, b) = gcd(a - b, b)}, \quad a > b$$

Proof By Induction,

Eg:- $\gcd(20, 15)$

We know $\gcd(20, 15) = 5$

$\therefore \gcd(20, 15) = \gcd(5, 15)$ (20 − 15, 15)

$\gcd(20, 15) \to 5$ ; $\gcd(5, 15) \to 5$

Now, applying Euclidean to (15, 5).

$\therefore \gcd(15, 5) = \gcd(10, 5)$

$\therefore \gcd(10, 5) = \gcd(5, 5)$

$\gcd(5, 5) = \gcd(0, 5)$

The moment, one of the numbers become zero, the other number is actually a GCD.

$\therefore \gcd(20, 15) = 5$

---

$\gcd(a, b) \longrightarrow \gcd(a-b, b) - - - - \longrightarrow 0$

(It will take much time)

$a = 52, b = 10$

$\gcd(52, 10) \to \gcd(32, 10) \to \gcd(32, 10) \to \gcd(22, 10)$

$\gcd(12, 10)$

or we could have said $(52-10, 10) = (42, 10)$ Instead 5 times

gcd(12,10) → gcd(2,10) → gcd(10,2) → gcd(8,2)

gcd(6,2) → (4,2) → (2,2) → (0,2)

(10%2,2) = (0,2)

This is a lot of steps (might not increase the time complexity)

the algorithm is actually:

gcd(a,b) == gcd(a%b, b), always take the greater number, a > b

if one of them is zero, other is GCD

i.e greater % smaller → one of them is 0 → other is GCD

Pseudocode

a b

```
while(a>0 && b>0)
{
  if(a>b)  a = a%b;
  else     b = b%a;
}
if(a==0)  print(b)
else      print(a)
```

↳ : division happening, no of iterations will be in $\log(x)$

: time Complexity → $O(\log_{\phi}(\min(a,b)))$

↳ In digit solution, it was $n/10$, so $\log_{10}$

$\phi$ since not sure, what will be a and b when they are getting divided.