Mathematics

Count digits:

$x > 0$

1) I/P: $x = 9235$ O/P: 4

I/P: $x = 38$ O/P: 2

I/P: $x = 7$ O/P: 1

```
int CountDigits (int x)
{
  int res=0;
  while (x>0)
  {
    x = x/10;
    res++;
  }
  return res;
}
```

$x = 789$

Initially: res = 0

After I st Iteration

$x = 78$

res = 1

After II nd Iteration

$x = 7$

res = 2

After III rd Iteration

$x = 0$

res = 3

Digits => d

T.C = $\theta(d)$

---

palindrome number

=> $n \geq 0$

I/P: n = 78987 O/P: yes

I/P: n = 8668 O/P: yes

I/P: n = 8 O/P: yes

I/P: n = 21 O/P: No

I/P: n = 367 O/P: NO

Idea: Traverse digits from right to left to find reverse of given number

last digit => x%10 Remaining => x = x/10

(1) n = 367

rev = 0

7: rev = rev * 10 + 7 = 7

6: rev = rev * 10 + 6 = 76

3: rev = rev * 10 + 3 = 763

(2) n = 363

rev = 0

3: rev = rev * 10 + 3 = 3

6: rev = rev * 10 + 6 = 36

3: rev = rev * 10 + 3 = 363

```
static boolean ispal (int n)
{ int rev=0;
  int temp=n;
  while (temp!=0)
  {
    int ld= temp%10;
    rev = rev*10 +ld;
    temp = temp/10;
  }
  return (rev==n);
}
```

n=4553
rev=0
temp=4553

I st iteration:
ld=3
rev=3
temp=455

II nd iteration:
ld=5
rev=35
temp=45

III rd iteration:
ld=5
rev=355
temp=4

IV th iteration
ld=4
rev=3554
temp=0

Digits ⇒ d
T.C ⇒ θ(d)

Factorial of a number:

(n≥0)

I/P: n=4 O/P: 24
I/P: n=6 O/P: 720
I/P: n=0 O/P: 1

1*2*3*4 - - - - (n-1)*n

Iterative:

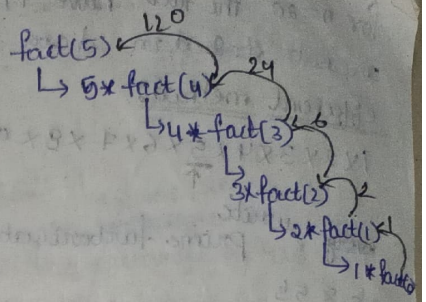
```
int fact (int n).
{ ires=1
  for(i=2; i<=n; i++)
  {
    res=res*i;
  }
  return res;
}
```

n=5
res=1
i=2 ; res=2
i=3 ; res=6
i=4 ; res=24
i=5 ; res=120

T.C = θ(n) } better
A.S = θ(n)
Auxiliary space

Recursive:

```
int fact (int n).
{ if(n==0)
    return 1;
  Return n*fact(n-1);
}
```



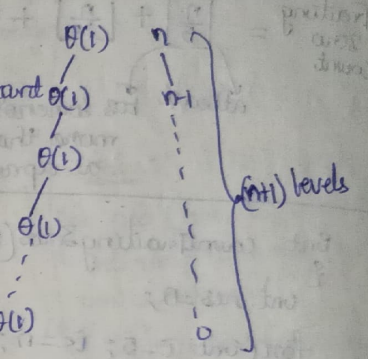
Assumption: n≥0

T.C: T(n) = T(n-1) + θ(1)

at every level work is constant

θ(n+1) * θ(1) = θ(n)

T.C = θ(n) } worst when compared to iterative
A.S = θ(n)
Auxiliary space



Trailing zeros in factorial

I/P: n=5
O/P: 1
1*2*3*4*5 = 120

I/P: n=10
O/P: 2
1*2*3 - - - 9*10
3628800

I/P: n=100
O/P: 24

Naive method

```
int countZeros(int n)
{ int fact = 1;
  for (int i=2; i<=n; i++)
      fact = fact * i;        } θ(n)
  int res=0;
  while (fact%10==0)
  {
    res++;
    fact = fact/10;           } <θ(n)
  }
  return res;
}
```

T.C = θ(n)

n=10
fact = 3628800
res = 0

I st Iteration of while:
res=1
fact = 362880

II nd Iteration of while:
res=2
fact = 36288

for n=20 the fact have 19 digits but that much of digits are not stored in data types => overflow issue.

effecient method:

1x2x3x4x5x6x7x8x9x10 . . . . 25 . . . n

write prime factorisation of that numbers & count 2's & 5's

every 5th no is having 5 as a prime factor

Trailing zero count = $\lfloor \frac{n}{5} \rfloor + \lfloor \frac{n}{25} \rfloor + \lfloor \frac{n}{125} \rfloor$ . . .

atleast bcs some no's have more than 1 primes as a prime factor

n=251
res=0

Ist Iteration: i=5
res=res+50
=50

IInd Iteration: i=25
res=res+10
=60

IIIrd Iteration: i=125
res=res+2
=62

```
int countTrailingZeros(int n)
{
  int res=0;
  for(int i=5; i<=n; i=i*5){
    res = res+n/i;
  return res;
}
```

T.C: $5, 5^2, 5^3, \ldots 5^k$

$5^k \le n$

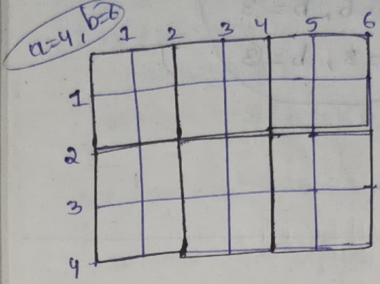
$k \le \log_5 n$ => $O(\log n)$

Greatest Common Divisor

I/P: a=4, b=6
O/P: 2

I/P: a=100, b=200
O/P: 100

I/P: a=7, b=13
O/P: 1



concept => Largest square tile that can fill the rectangle of 4x6 is 2x2 => so o/p is 2.

largest 2x2 tile is fill the whole rectangle

Naive soln:

C++:

```
int gcd (int a, int b)
{
  int res = min(a,b);
  while(res>0)
  { if (a%res==0 && b%res==0)
    {
      break;
    }
    res--;
  }
  return res;
}
```

a=10, b=15
res=10
res=9
res=8
res=7
res=6
res=5 ✓

Java:

```
int gcd (int a, int b)
{ int res = Math.min(a,b);
  while (res>0)
  { if (a%res==0 && b%res==0)
    { break;
    } res--;
  } return res;
}
```

T.C: ~~O(min(a,b))~~

O(min(a,b)) worst case.

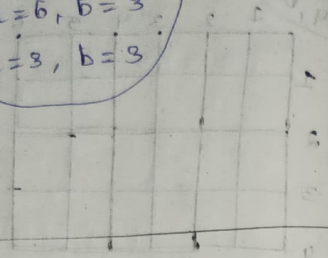
Euclidean Algorithm:

Basic idea: let 'b' be smaller than 'a'
gcd(a,b) = gcd(a-b,b)

why? let 'g' be GCD of 'a' and 'b' a=gx b=gy
and GCD(x,y)=1 => a-b = g(x-y)

```
int gcd(int a, int b)
{
  while (a != b)
  {
    if (a > b)
      a = a - b;
    else
      b = b - a;
  }
  return a;
}
```

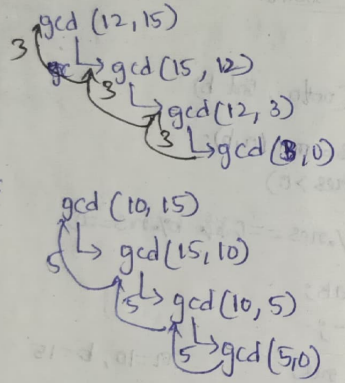
a = 12, b = 15
a = 12, b = 3
a = 9, b = 3
a = 6, b = 3
a = 3, b = 3



Optimised Implementation :-

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

T.C = O(log(min(a,b)))



Lcm of Two Numbers

I/p: a = 4, b = 6
O/p: 12

I/p: a = 12, b = 15
O/p: 60

I/p: a = 2, b = 8 (worst case)
O/p: 8

I/p: a = 3, b = 7
O/p: 21

```
int lcm(int a, int b)
{
  // java: res = Math.max(a,b);
  int res = max(a,b);
  while (true) {
    if (res % a == 0 && res % b == 0)
      return res;
    res++;
  }
  return res;
}
```

a = 4, b = 6
res = 6
res = 7
res = 8
res = 9
res = 10
res = 11
res = 12

T.C: O(a*b - max(a,b))

Efficient Solution :-

a * b = gcd(a,b) * lcm(a,b)

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

=> T.C = O(log(min(a,b)))

Check for prime

n > 0

I/p: n = 13
O/p: yes
13 has divisors as 1 and 13 only

I/p: n = 14
O/p: NO
14 has divisors as 1, 2, 7 and 14.

I/p: n = 101
O/p: Yes
101 has divisors as 1 and 101 only.

First few prime numbers: 2, 3, 5, 7, 11, 13, 17, 19, ...

Naive method :

```
boolean isPrime(int n)
{
  if (n == 1) return false;
  for (int i = 2; i < n; i++)
    if (n % i == 0)
      return false;
  return true;
}
```

n = 65
i = 2
i = 3
i = 4
i = 5
O/p: NO

n = 7
i = 1
i = 2
i = 3
i = 4
i = 5
i = 6
O/p: yes

Idea: check 2 to n-1

Because: if n is even

worst case: if n is prime O(n-2) => O(n)

optimized soln: $O(n) \rightarrow O(\sqrt{n})$

efficient method:

Idea: divisors always appear in pairs [check 2 to $\sqrt{n}$]

30: (1,30), (2,15), (3,10), (5,6)

65: (1,65), (5,13),

25: (1,25), (5,5)

$T.C = O(\sqrt{n})$

```
boolean isprime(int n)
{ if (n==1) return false;
  for (int i=2; i*i<=n; i++)
    if (n%i==0)
      return false;
  return true;
}
```

n=65	n=37
i=2	i=2
i=3	i=3
i=4	i=4
i=5	i=5
No	i=6
	Yes

more efficient method: [for large n]

Idea: By checking $n\%2==0$ and $n\%3==0$, we can save many iterations for large n;

~~1~~ ~~2~~ ~~3~~ ~~4~~ 5 ~~6~~ 7 ~~8~~ ~~9~~ ~~10~~

11 ~~12~~ 13 ~~14~~ ~~15~~ ~~16~~ 17 ~~18~~ 19 ~~20~~

~~21~~ ~~22~~ 23 ~~24~~ 25 ---- $\sqrt{n}$

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

prime factors

n>1

I/P: n=12

O/P: 2 2 3

I/P: n=13

O/P: 13

I/P: n=315

O/P: 3 3 5 7

prime numbers: 2, 3, 5, 7, 11, 13...

Naive solution

```
void primefactors(int n){
  for(int i=2; i<n; i++){ ⇒ O(n)
    if(isprime(i)){ ⇒ dependent on n
      int x=i;
      while(n%x==0){ ⇒ atmost O(logn)
        print(i);
        x=x*i;
      }
    }
  }
}
```

Why we are using another variable x?

multiple powers of i can also divide n.

n=12

i=2 divides n & 4 also divides n.

n=12

i=2:

2

2

i=3:

3

i=4

i=5

⋮

i=11

$T.C = O(n^2 \log n)$

efficient solution:

① Divisors always appear in pairs

30: (1,30), (2,15), (3,10), (5,6)

② A number n can be written as multiplications of power of prime factors

$12 = 2^2 * 3$

$450 = 2^1 * 3^2 * 5^2$

$13 = 13^1$

Let (x, y) be a pair

$x * y = n$

If $x \le y$ then

$x * x \le n$

$x \le \sqrt{n}$

① method to get prime divisors is iterate from 2 to $\sqrt{n}$

```
Void printprimefactors (int n)
{
  if (n<=1) return ;
  for (int i=2 ; i*i<=n ; i++)
  {
    while (n%i==0)
    {
      print (i);
      n=n/i ;
    }
  }
  if (n>1)
    print (n);
}
```

NOTE:
$450 = 2^1 \times 3^2 \times 5^2$

n=450
i=2:
print(2)
n=225
i=3:
print(3)
n=75
print(3)
n=25
i=4:
i=5:
print(5)
n=5
print(5)
n=1

n=84
i=2:
print(2)
n=42
print(2)
n=21
i=3:
print(3)
n=7
After loop:
print(7)

$84 = 2^2 \times 3 \times 7$

Further optimisation

2, 3, 4, (5), 6, (7), 8, 9, 10, (11)
12, (13), 14, 15, 16, (17), 18, (19)
20, 21 - - - - - - $\sqrt{n}$

more effecient solution :-

```
Void printprimefactor (int n)
{ if (n<=1) return ;
  while (n%2==0)
  { print(2); n=n/2; }
  while (n%3==0)
  { print(3); n=n/3; }
  for (int i=5; i*i<=n; i=i+6)
  { while (n%i==0)
    { print(i) ; n=n/i ; }
    while (n%(i+2)==0)
    { print(i+2); n=n/(i+2) ; }
  }
  if (n>3)
  print (n);
}
```

n=450
print(2)
n=225
print(3)
n=75
print(3)
n=25
print(5)
print(5)

T.C ⇒ for prime Nos ⇒ $\theta(\sqrt{n})$

⇒ If n is prime we don't need to execute inner for loop ⇒ $\theta(\sqrt{n})$

⇒ If n is not prime In inner for loop dividing the no. with i so n is drastically reducing

ALL Divisors of a Number

I/P: n=15
O/P: 1 3 5 15

I/P: n=100
O/P: 1 2 4 5 10 20 25 50 100

I/P: n=7
O/P: 1 7

Naive solution:

```
Void printDivisors (int n){
for (int i=1; i<=n; i++){
  if (n%i==0)
  { print(i);
  }
}
}
```

n=6
i=1: print(1)
i=2: print(2)
i=3: print(4)
i=4 :
i=5 :
i=6 : print(6)

T.C = $\theta(n)$
A.S = $\theta(1)$

Efficient solutions:

1) Divisors always appear in pairs
   so : (1, 30) ; (2, 15) , (3, 10) , (5, 6)

2) one of the divisors in every pair is smaller than or equal to $\sqrt{n}$

   for a pair (x, y)

   let x be the smaller, i.e; $x \le y$

   $x * x \le xy$ (where $xy = n$)

   $x \le \sqrt{n}$

A solution that prints all divisors but not in order:

```
Void printDivisors (int n)
{
  for (i=1; i*i<=n ; i++)
  {
    if (n % i == 0)
    {
      print (i);
      if (i != n/i)
      {
        print (n/i);
      }
    }
  }
}
```

n = 25

i = 1 : print (1), print (25)
i = 2 :
i = 3 :
i = 4 :
i = 5 : print (5)

T.C = $\theta(\sqrt{n})$

A solution that prints all divisors in order:

```
Void printDivisor (int n)
{
  int i;
  for (i=1; i*i<n ; i++)
  {
    if (n % i == 0)
      print (i)
  }
  for (i ; i>=1 ; i--)
    if (n % i == 0)
      print (n/i)
}
```

$\theta(\sqrt{n})$ prints all divisors from 1 (inclusive) to $\sqrt{n}$ (exclusive)

$\theta(\sqrt{n})$ prints all divisors from $\sqrt{n}$ (exclusive) to n (inclusive)

T.C = $\theta(\sqrt{n}) + \theta(\sqrt{n}) = \theta(\sqrt{n})$

n = 15

I^st loop:
i = 1 : print (1)
i = 2 :
i = 3 : print (3)

II^nd loop:
i = 4 :
i = 3 : print (5)
i = 2 :
i = 1 : print (15)

Sieve of Eratosthenes :-

I/p : n = 10 → O/p : 2, 3, 5, 7

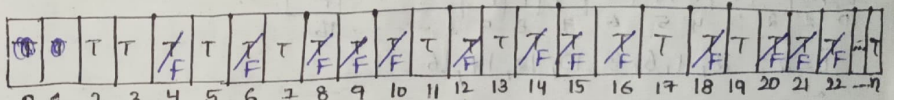
I/p : n = 23 → O/p : 2 3 5 7 9 11 13 17 19 23

Naive Solution:

```
Void printprimes (int n)
{
  for (int i=2; i<=n ; i++)
  {
    if (is prime (i))
      print (i);
  }
}
```

T.C = $O(n * \sqrt{n}) \Rightarrow O(n^{3/2})$

n = 10
i = 2 : print (2)
i = 3 : print (3)
i = 4 :
i = 5 : print (5)
i = 6 :
i = 7 : print (7)
i = 8 :
i = 9 :
i = 10 :



Initially all nos are considered as True.
for multiples of 2, 3, 5, 7 --- marks as False

Multiples of 2, 3 and 5:
2 : 4, 6, 8, 10, 12, 14, 16, 18, 20, - - - -
3 : 6, 9, 12, 15, 18, - - - - - - - -
5 : 10, 15, 20, - - - - - -

Java : create a boolean array & use arrays.fill() fn to fill all the values as true

```
Void Sieve (int n)
{
  vector<bool> isprime(n+1, true);
  for (int i=2; i*i<=n; i++)
  {
    if (isprime[i])
    {
      for (int j=2*i; j<=n; j=j+i)
        isprime[j] = false;
    }
  }
  for (int i=2; i<=n; i++)
  {
    if (isprime[i])
      cout<<i<<" ";
  }
}
// Simple Implementation of Sieve
```

n = 9

0	1	2	3	4	5	6	7	8	9
/	/	T	T	T	T	T	T	T	T

i=2 : print 2
mark 4, 6, and 8 as false

0	1	2	3	4	5	6	7	8	9
/	/	T	T	F	T	F	T	F	T

i=3 : print 3
mark 3, 6, 9 and 9 as False

0	1	2	3	4	5	6	7	8	9
/	/	T	T	F	T	F	T	F	F

n = 16

0	1	2	3	4	5	6	7	8	9	...	16
/	/	T	T	F	T	F	T	F	F	...	F

i=2 : 2, 4, 6, 8, 10, 12, 14, 16
i=3 : 3, 6, 9, 12, 15
i=4 : Doesnot go inside if

optimised Implementation :

```
Void Sieve (int n)
{
  vector<bool> isprime(n+1, true);
  for (int i=2; i*i<=n; i++)
  {
    if (isprime[i])
    {
      for (int j=i*i; j<=n; j=j+i)
        isprime[j] = false;
    }
  }
  for (int i=2; i<=n; i++)
  {
    if (isprime[i])
      cout<<i<<" ";
  }
}
```

Replaced by 2*i

$i^2, i^2+i, i^2+2i, \ldots$

earlier :
5 : 10, 15, 20, 25, 30 ...

Now : (removed by 2 & 3)
5 : 25, 30 ...

Composite numbers smaller than i
i * (i-1)
i * (i-2)

Shorter Implementation of the optimised Sieve

performance & T.C wise same as before code but code length is reduced.

```
Void Sieve (int n)
{
  vector<bool> isprime(n+1, true);
  for (int i=2; i<=n; i++)
  {
    if (isprime[i])
    {
      cout<<i<<" ";
      for (int j=i*i; j<=n; j=j+i)
        isprime[j] = false;
    }
  }
}
```

T.C of Sieve algo : $O(n \log\log n)$

$O(n^{3/2}) \geq O(n \log\log n)$

naive approach — better optimised

Computing pow(x, n)

$n \geq 0$

ip: x=2, n=3
op: 8

ip: x=3, n=4
op: 81

ip: x=5, n=0
op: 1

ip: x=5, n=1
op: 5

Naive Solution:

```
int power (int x, int n)
{
  int res = 1;
  for (int i=0; i<n; i++)
      res = res * x;
  return res;
}
```

⇓

T.C = Θ(n)

x=3, n=4
res=1
i=0 : res=3
i=1 : res=9
i=2 : res=27
i=3 : res=81

efficient solution:

$$power(x,n) = \begin{cases} \text{if } n\%2 == 0 \\ \quad power(x, n/2) * power(x, n/2) \\ \text{else} \\ \quad power(x, n-1) * x \end{cases}$$

```
int power (int x, int n)
{
  if (n==0)
    return 1;
  int temp = power(x, n/2);
  temp = temp * temp;
  if (n%2 == 0)
    return temp;
  else
    return temp * x;
}
```

pow(3,5) =
→ temp = pow(3,2) = 3*3
→ temp = pow(3,1) = 3
→ temp = pow(3,0) = 1
→ return 3 * 1 * temp
→ return 3*3
→ return (3*3) * (3*3) * 3

T.C: $T(n) = T(\lfloor n/2 \rfloor) + \Theta(1)$

height = log n

Θ(1) n=16
Θ(1) → n=8
Θ(1) → n=4

at every level Θ(1) work

T.C = Θ(log n)

Auxiliary Space = Θ(log n)

Iterative (Binary Exponentiation)

T.C = Θ(log n) A.S = Θ(1)

$3^{10} = 3^8 \times 3^2$
$3^{19} = 3^{16} \times 3^2 \times 3^1$

10 : 1 0 1 0
19 : 1 0 0 1 1

- Every number can be written as sum of powers of 2 (set bits in binary representation)
- we can traverse through all bits of a number (from LSB to MSB) in O(log n) time.

```
while (n > 0)
{
  if (n%2 != 0)
    // Bit is 1
  else
    // Bit is 0
  n = n/2;
  x = x * x;
}
```

```
int power (int x, int n)
{
  int res = 1;
  while (n > 0)
  {
    if (n%2 != 0)
      res = res * x;
    x = x * x;
    n = n/2;
  }
  return res;
}
```

c++: n%2 != 0 not java

x = 4, n = 5
Initially: res = 1

I st iteration: res = 4, x = 16, n = 2

II nd iteration: res = 4, x = 16*16 = 256, n = 1

III rd iteration: res = 4*256 = 1024, x = 256*256 = 65536, n = 0

5 : 1 0 1
$4^1 4^0 4^1$
$4^1 * 4^1$ res = $4^1 * 4^1$

5 : 1 0 0 1
$4^1 4^0 4^0 4^1$

T.C = Θ(log n)
A.S = Θ(1)

for large nos to overcome overflow:

```
int power (int x, int n, int m)
{ int res = 1;
  while (n > 0)
  { if (n % 2 != 0)
      res = (res * x) % m;
    x = (x * x) % m
    n = n >> 1;
  }
  return res;
}
```