M1 : Gearing Up

GCD (Greatest Common Divisor) or **HCF (Highest Common Factor)** of two numbers 'a' & 'b' (a,b >=0) is the largest number that divides both 'a' & 'b'.
Eg. GCD(10,15) = 5

How to calculate GCD?

Approach:

Brute Force - Since 1 **<= GCD(a,b) <= min(a,b)** ; a, b > 0
We can create a for loop from min(a,b) to 1 and check if the number is divisible by both 'a' and 'b'.
Time complexity: **O(min(a, b))**
Space complexity: **O(1)**
Euclid's Division Lemma - We can use the concept of repeated divisions to find the GCD of two numbers 'a' and 'b'.
Here, the **divisor = max(a, b)** and **dividend = min(a, b)**. We repeatedly perform division operations until the remainder becomes zero.

For a>=b, after every division,
divisor = a%b
dividend = b

Time complexity: **O(log2(max(a,b))**
Space complexity: **O(1)**

Note: GCD of any non-zero number with zero is the number itself.

More on GCD

In this lecture, we will learn how to calculate the GCD of integers stored in an array.

For three numbers 'a', 'b' and 'c',

GCD(a, b, c) = GCD(GCD(a, b), c) = GCD(GCD(a, c), b) = GCD(a, GCD(b, c))

Using the above logic, the GCD of an array 'Arr[N]' can be calculated as -

```
int GCD = Arr[0];
for(int i = 1; i < N; i++){
     GCD = gcd(GCD, Arr[ i ]);
}
```

Q. We have been given an integer array 'Arr[N]': Return '1' if the array contains a subsequence with GCD = 1 otherwise return '0'.

Approach:

Brute Force - Find all the subsequences of the array and calculate their GCD.
Time complexity: **O(2^N)**
For every element we have two options either to select it in the subsequence or not, thus there will be 2^N number of total subsequences.
Space complexity: **O(1)**
Can we check all the pairs in the array to find if there is any subsequence with GCD = 1?
No, the above method may not work in every case.
Eg. For Arr[3] = {6, 10, 15}, GCD(6, 10) = 2; GCD(6, 15) = 3; GCD(10, 15) = 5 but GCD(6, 10, 15) = 1
Since we know that the GCD of any number with 1 is 1 itself. Can we use this fact to solve the problem?
We can calculate the GCD of the entire array and if it is equal to 1, it means the required subsequence exists inside the array, otherwise it doesn't.
Time complexity: **O(Nlog2(max(Arr[i])))**
Space complexity: **O(1)**

Lowest Common Multiple

LCM (Least Common Multiple) of two numbers, 'a' and 'b' is the smallest number which is divisible by both 'a' & 'b'.
Eg. LCM(10,15) = 30

How to calculate LCM of two numbers?

Approach:

Brute Force - Since **max(a, b) <= lcm(a,b) <= a*b** ; a,b>0
Therefore, we can iterate from the minimum to the maximum possible value of LCM to find the LCM.
Time complexity: **O((a*b)-max(a,b))**
Space complexity: **O(1)**
Using Mathematical Relation - HCF * LCM = a * b
Hence, **LCM = (a*b)/HCF**

How to calculate the LCM of three numbers?

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

Similarly, we can calculate the LCM of the elements of an array 'Arr[N]'

```
int LCM = Arr[0];
for(int i = 1; i < N; i++){
     LCM = (LCM*Arr[i])/GCD(LCM, Arr[i]);
}
```

Generating All Factors

Factors of a number 'a' are all the different numbers that divide 'a'.
Eg. Factors of 12 = {1, 2, 3, 4, 6, 12}

Approach:

Brute Force - Iterate from 1 to N and check if they divide N or not.
Time complexity: **O(N)**
Optimised Brute Force - Since the factors of 'N' are symmetric about its square root ie √N.

← M1 : Gearing Up

Therefore, we can iterate from 1 to √N. And if 'i' divides N, it means both 'i' and 'N/i' are the factors of N excluding the case where i=N/i or i =√N - to avoid duplication while printing.
Time complexity: O(√N)

Note: If an integer is a perfect square then it has an odd number of factors otherwise it will have an even number of factors.

Open Close Problem

There are N doors that are initially closed. There are N rounds and In every round 'i', we have to toggle the states of all the doors which are a multiple of 'i'. Find the number of doors open at the end of the game.

Approach: We know that a door will be toggled only if it is divisible by 'i'. This implies that a door will be toggled as many times as the number of factors it has.
Since we know that a perfect square has an odd number of factors while an imperfect square has an even number of factors. Therefore, only the state of those doors will change that are a perfect square or the doors having an odd number of factors.

Answer = (int)sqrt(N)

Primality Test

If a number 'x'(>1) has only two divisors - 1 and 'x' ie. the number itself then it is called a Prime Number. In this lecture, we will learn how to check the primality of any number 'N' in O(N) time complexity.

Approach:

Brute Force - Count the factors of 'N' by iterating from 1 to 'N'. It will be a prime number only if the count is 2.
Time complexity: **O(N)**
Optimised Brute Force - Since we know that the factors of 'N' are symmetric about √N. Therefore, we can iterate from 1 to √N and count the total number of factors of 'N'. It will be a prime number if the count is 1.
Time complexity: **O(√N)**

Primality Test in a range

In this lecture, we will learn how to find all the prime numbers in the range of 1 to N using the **Sieve of Eratosthenes**. It is the most efficient pre-processing technique to find all the prime numbers less than 10^7.
Note: It is not advisable to use it if N>10^7 or if the range is not starting from 1.

Approach:

Brute Force - We can iterate from 1 to N and can check the primality of each number individually.
Time complexity: **O(N√N)**
Space complexity: **O(1)**
Using Sieve of Eratosthenes - In this technique, we create a boolean array to represent the state of the numbers from 1 to N. Initially we assume each one of them to be prime and we iterate from 2 to N marking all their divisors to be non-prime in the boolean array.
We follow this process only for the numbers that are marked as prime in the boolean array. At the end, we will be left with the true nature of the numbers in the boolean array.
Time complexity: *N/2+N/3+N/5+N/7+...+=N[1/2+1/3+1/5+1/7+...+]* = **O(Nlog(logN))**
Space complexity: **O(N)**

Implementing Sieve

In this lecture, we will look at the implementation of the sieve of Eratosthenes.

```
bool Primes[N+1];           //To store the nature of all the numbers from 1 to N
for(int i = 1; i<N+1; i++){
    Primes[i]=1;            //All the numbers are marked as Primes
}
Primes[0]=0;                //Non-primes are represented by 0
for(int i = 2; i*i<=N; i++){
    if(Primes[i]==1){       //If a number is prime then mark its multiples as non-prime
      for(int j=i*i;j<=N;j+=i){
          Primes[j]=0;
      }
    }
}
```

Drawback:

If we have to find the prime numbers in the range - 10^9 to 10^10. Then this approach may not be helpful because the space required by the boolean array will not fit in the memory of the program. For such cases, the brute force approach of O(N√N) will be a better choice.

Prime Factorization

Expressing a number as a product of its prime factors is called Prime factorisation. N = P1*P2*P3..Pi where P1, P2,...Pi are primes.

Eg. 12=2*2*3

Approach:

Brute Force - Iterate 'i' from 2 to N and keep dividing N till it is divisible by 'i'. Maintain the count of each prime factor to find their power in the prime factorisation of N.
Can it ever happen that i divides N but it is a composite number?
No, it will not happen as we are starting with the lowest possible prime number i.e. 2.
Time complexity: O(Nlog2N)

Since $\sqrt{N}*\sqrt{N}=N$, therefore if one of the prime factors is larger than √N then the rest have to be smaller than √N otherwise the product of prime factors will be larger than N, which is not possible.
Hence we can optimize the previous brute force approach by iterating 'i' from 2 to √N and keep dividing N till it is divisible by 'i'. At the end of the process, we have to check the remaining number. If it is not equal to 1 then it is to be counted as a prime factor of N.
Eg. 404=2*2*101
Note: The maximum number of terms the prime factorisation of a number would have is equal to log2N.
Time complexity: **O(N + log2N) = O(N)**

M1 : Gearing Up

Print the prime factorisation of n1, n2, n3,.., nq (ni<=N).

Approach:

Using Sieve of Eratosthenes - Create an array of prime numbers (<=N) using the Sieve of Eratosthenes. Iterate through the array to find all the prime factors of ni.
Time complexity: **O(Nlog(logN)+(number of primes x (logN)))**
Space complexity: **O(N)**
Using Smallest Prime Factor (SPF) - This method helps in getting rid of useless prime numbers by finding the smallest prime factor of all the numbers till N. We implement this algorithm by slightly modifying the code of the sieve of Eratosthenes.
Time complexity: **O(NloglogN + QlogN)**
Space complexity: **O(N)**

Counting Divisors Faster

In continuation with the previous lecture on Fast Factorization, we will learn how to efficiently count the factors of a number 'N' by using the Rule of Products.

We have been given an integer, N = (P1)^c1(P2)^c2(P3)^c3..(Pi)^ci; where P1, P2, P3,.., Pi are prime numbers, then print the value of c1 + c2 + c3 +..+ ci.

Approach: Since we know that any factor of N consists of the product of some or all the prime factors of N. And for every prime factor Pi with power ci, we have (ci+1) different powers to consider in the product.

Therefore by Rule of Products, **total number of divisors = (c1+1)(c2+1)(c3+1)..(ci+1)**.

Segmented Sieve

In this lecture, we will see the limitation of the Sieve of Eratosthenes in finding primes in the range [L, R] where L~10^9 and (R-L) is small.

Print all the primes in the range [L, R] such that L = 10^10, R=10^10+500.

Using Segmented Sieve:
The major drawback of the sieve of Eratosthenes is the space taken by the auxiliary array. Can we somehow reduce the extra space?

Consider that we have an array representing the state of numbers between L & R. Then what will be the required prime numbers to strike off the composites in [L, R]?
=> Primes between 2 to √R

For example, For L = 90 & R = 99, the prime numbers between 2 to √99(~10) are sufficient to strike off all the composites.
90 91 92 93 94 95 96 97 98 99

Numbers divisible by 2
90 91 92 93 94 95 96 97 98 99

Numbers divisible by 3 out of the remaining numbers
91 93 95 97 99

Numbers divisible by 5 out of the remaining numbers
91 95 97

Numbers divisible by 7 out of the remaining numbers
91 97

Primes found: 97
Since we are able to achieve our objective by using prime numbers till √R. Therefore there is no need to find the state of all the numbers from 1 to R.
We can use the Sieve of Eratosthenes to find the prime numbers between 1 to √R and can store them in a vector P[ ]. It will help in making the process of striking more efficient by removing the composites between 1 to √R.
Time complexity: O(√RloglogR)
Space complexity: O(√R)
Create a boolean array isPrime[ ] of size (R-L+1) to track the state of numbers in [L, R] and mark them as prime initially.
Iterate on the primes array P[ ] and strike out their divisors by marking them as non-prime in the isPrime[ ] array.
The first multiple of prime P[i] in the range [L,R] can be found as P[i]*k where k = ceil((L*1.0)/P[i])

Note:

ceil() returns the Greatest Integer close to the decimal value inside it.
ceil(3.2) = 4
ceil(3.0) = 3
L/P[i] results in an integer division therefore we multiply L with 1.0 for a float division.

Implementing Segmented Sieve

Here we will look at the implementation of the segmented sieve, discussed in the previous lecture.

Find prime numbers in the range [Li, Ri] for the given 'Q' queries - (L1, R1), (L2, R2), (L3, R3)....(Lq, Rq) & R<=10^11.

Code:

```
//Extracting primes between 1 to √R(~10^6)

vector<int> getPrimes( ){
     vector<bool> isPrime(10^6+1, true);  //labeling numbers till √R as prime(true)
     isPrime[1] = false;
     for(int i = 2; i*i<=10^6; i++){
          if(isPrime[i]){   //Iterating to strike multiples of prime no. 'i'
             for(int j = i*i; j<=10^6; j+=i){
                  isPrime[j]=false;  //marking non-primes as false
}
          }
       }
       vector<int> primes;
       //storing all the prime numbers in a vector - primes[ ]
       for(int i =1; i<10^6; i++){
            if(isPrime[i]){
              primes.push_back(i);
            }
        }
}
```

```
//finding primes in the range [L, R]

//labeling numbers in [L, R] as prime(true)
vector<bool> isPrime(R-L+1, true)

//iterating over the primes array to strike off their multiples in [L, R]
for(int i = 0; i<primes.size(); i++){
```

M1 : Gearing Up

```
//iterate 'Q' times to print the primes in the range [Li, Ri]
```

Time complexity: O(√Rloglog√R + (R - L)(loglog√R))
Space complexity: O(√R)

Fundamentals of Modular Arithmetic

Modulus of two numbers a & b, i.e. a%b represents the remainder obtained after dividing a by b.

Properties:

x%m ∈ [0, m-1]
There is a periodic repetition of x%m for a particular value of m & x ∈ N U {0}. This property is also known as **Modular congruence**.
Eg. for m = 3
0%3 = 3%3 = 6%3 = 0
1%3 = 4%3 =1
2%3 = 5%3 = 2

0 ≡ 3 mod 3 ≡ 6 mod 3
The remainder of the sum is equal to the sum of remainders -
(a+b)%m = (a%m + b%m)%m
(a-b)%m = (a%m) - (b%m) if (a%m)>(b%m)
= (a%m) - (b%m) + m if (a%m)<(b%m)
(a*b)%m = ((a%m)*(b%m))%m

Counting Pairs

Here we have been given an integer array 'Arr[N]' and an integer 'k'. We have to find the total number of pairs (i, j) such that (Arr[i]+Arr[j]) is divisible by 'k' where i≠j, Arr[i]>=0.

Note: Both (i,j) and (j,i) are counted as the same

Approach:

Brute Force - Consider all the possible pairs of the array & count ones whose sum is divisible by 'k'.
Time complexity: **O(N^2)**
Space complexity: **O(1)**
Using the property of Modulus - The remainder of the sum is equal to the sum of remainders.
(ai+aj)%k = (ai%k+aj%k)%k
If ai%k = r1 & aj%k = r2
then, (ai+aj)%k = (r1+r2)%k where 0<=r1,r2<=k-1
The above equation will yield zero if,

r1 + r2 = 0
It is possible when: r1 = 0 & r2 = 0
r1 + r2 = k
It is possible when: r1 = i & r2 = k-i where 1<=i<=k-1
We can create an array cnt[k] to store the frequency of remainders from 0 to k-1. By the rule of products and the rule of combinatorics, the answer will be given as -

$$\text{Answer} = {}^{cnt[0]}C_2 + \sum_{i=1}^{i=k/2-1} cnt[i]cnt[k-i] \quad \text{if k is odd}$$
$$= {}^{cnt[0]}C_2 + \sum_{i=1}^{i=k/2-1} cnt[i]cnt[k-i] + {}^{cnt[k/2]}C_2 \quad \text{if k is even}$$

Time complexity: **O(N+k)**
Space complexity: **O(k)**

Counting Triplets

We have been given an integer array 'Arr[N]' and an integer 'm'. We have to find the count of triplets (i, j, k) such that (Arr[i]+Arr[j]+Arr[k]) is divisible by 'm' where Arr[i]>=0.

Note: All permutations of the triplet (i, j, k) have been considered as the same

Approach:

Brute Force - Consider all the triplets of Arr[N] & count ones whose sum is divisible by 'k'.
Time complexity: O(N^3)
Space complexity: O(1)
Using the property of modulus - The remainder of the sum is equal to the sum of remainders.
(ai+aj+ak)%m = ((ai+aj)%m+ak%m)%m = ((ai%m+aj%m)%m+ak%m)%m
If ai%m=ri, aj%m=rj & ak%m=rk
i.e. ((ri+rj)%m+rk)%m where, 0<=ri,rj,rk<=k-1

The above equation will be zero when:

If (ri+rj)%m = 0 then rk = 0
If (ri+rj)%m = x then rk = k - x

So we can iterate for the first two remainders which will automatically fix the third one. And by using the rule of combinatorics & the rule of products we can easily calculate the answer.
Time complexity: **O(N+m^2)**
Space complexity: **O(m)**

Note: The second approach is better than the first approach only when N>M. Eg. N=100 & m = 100. If it is the opposite, then using the first approach is a better idea.

Basics of Combinatorics

Product Rule - If we can complete an Event by performing a sequence of 'k' subtasks and each of these tasks can be further performed in n1, n2,..,nk ways respectively, then the total number of ways in which the event can be performed is n1*n2*n3*...*nk.

A bitstring is composed of 0s and 1s. So, for every position, there are 2 possible digits. Thus by the Product rule, the number of configurations will be 2N.

Q. How many distinct subsets does a set 'S' with 'n' element have?
For every element we have two options - either to choose it in the subset or not, therefore by the Product rule the number of subsets will be 2n.

Q. How many distinct permutations are possible for a set S = [1, 0, 2]
For the first position, we have three choices - 1, 0 & 2
For the second position, we have two choices
For the third position, we have only one choice

Therefore by the Product Rule, the total number of distinct permutations will be 6
_3_ * _2_ * _1_ = 6 (or 3!)

Note: The number of distinct permutations of N objects is N!

Understanding nCr

A binomial coefficient nCr or C(n, r) is the coefficient of xr in the expansion of (1+x)^n. It also tells us the number of ways in which 'r' objects can be chosen from 'n' objects, ignoring the order.

nCr = nCn-r = n!/r!(n-r)!

Q. How many different combinations are possible by picking 2 alphabets out of A, B, C, D, E?

Number of choices for first alphabet = 5
Number of choices for the second alphabet = 4
Total combinations = 5*4 = 20

But combinations like AB and BA are considered the same, thus total combinations = (5*4)/2 = 10

Note: The number of different combinations to choose 2 objects out of N objects is NC2

Q. There are N bits. Find no. of bitstrings of size 'N' with exactly 'k' 1s.

Number of bitstring with exact 'k' 1s = Number of ways to choose 'k' positions = NCK

Q. We have to pick any three walls out of N walls and colour them with RGB. How many different configurations can be made?

Number of ways to choose 3 walls out of N walls = NC3
Total configurations = NC3*3!

Pascal Triangle for nCr

We have been given 'N' boys and 'M' girls and we need to choose a group containing exactly 'T' people having no less than 4 boys and 1 girl. Print the number of distinct combinations of groups that are possible. 4<=N<=30 & 1<=M<=30

Approach:

Suppose we have 5 boys & 5 girls and we want to make a team of 5, then the number of distinct combinations will be 10C5
If we want all the above combinations except those consisting of 2 boys & 3 girls then the number of distinct combinations will be 10C5 - 5C2*5C3.
Taking a hint from above, the number of distinct groups consisting of T people out of N+M people will be (N+M)CT.
Since 4<=N and M>=1, therefore

$$\text{Answer} = {}^{N+M}C_T - {}^{N}C_3\,{}^{M}C_{T-3} - {}^{N}C_2\,{}^{M}C_{T-2} - {}^{N}C_1\,{}^{M}C_{T-1} - {}^{N}C_0\,{}^{M}C_T - {}^{N}C_T\,{}^{M}C_0$$

Note: *In NCr, N>=r. Pay attention to this rule while writing the code. For eg. In NC1*MCT, T-1>=M.*

For N=M=30, we will need to calculate 60C30 = 60!/(30!*30!). But 60! can not be stored in any of the known data types. But we also know that 60C30 lies in the integer range. This is where Pascal's Triangle comes to the rescue.

Pascal's Triangle is a numerical pattern consisting of numbers arranged in a triangular fashion.

1

M1 : Gearing Up

1 3 3 1
1 4 6 4 1
1 5 10 10 5 1

$$(\because {}^{n}C_{r} = {}^{n-1}C_{r-1} + {}^{n-1}C_{r})$$

∴ We can create a 2D matrix of size 61x61 and use the above formula to calculate different nCr values.

1					
1	1				
1	2	1			
1	3	3	1		
1	4	6	4	1	
1	5	10	10	5	1

```
//code for finding nCr values for N=M=30

long long P[61][61];
P[0][0]=1;
for(int i = 0; i<61; i++){
     for(int j = 0; j<=i; j++){
          if(i==j or j==0) P[i][j]=1;
          else P[i][j]=P[i-1][j-1] + P[i-1][j];
     }
}
```

Introduction to Catalan Numbers

In this lecture, we will study an interesting set of numbers called **Catalan Numbers** and see how they can help us in solving the **Balanced parenthesis problem**.

Q. Print the count of distinct Balanced parentheses consisting of 'N' pairs of braces
For N=3
(( ))( ) - Balanced
( ))(( ) - Unbalanced

Approach:

We can find the number of balanced parentheses for lower values of N.

For N=0 → Empty
cnt[0] = 1

For N=1 → ( )
cnt[1] = 1

For N=2 → ( )( ), (( ))
cnt[2] = 2

For N=3 → ( )( )( ), ( )(( )), (( )))( ), (( )( )), ((( )))
cnt[3] = 5
In a balanced parentheses, for any prefix string cnt('(') > = cnt(')'). Let us see how we can use this property together with the Product Rule to find out the number of balanced parentheses for higher values of N.
For N=5, the balanced parentheses can be represented as
(Inside[i]) Outside[O]

We can analyze it by putting all possible bracket combinations in the inside & outside area.
(0 I) 4 O = cnt[0]*cnt[4]
(1 I) 3 O = cnt[1]*cnt[3]
(2 I) 2 O = cnt[2]*cnt[2]
(3 I) 1 O = cnt[3]*cnt[1]
(4 I) 0 O = cnt[4]*cnt[0]

cnt[5] = cnt[0]*cnt[4] + cnt[1]*cnt[3] + cnt[2]*cnt[2] + cnt[3]*cnt[1] + cnt[4]*cnt[0]

The above pattern makes an important family of numbers known as Catalan numbers. We can create an array cnt[N+1] to store the answers for different values of N.

Catalan Numbers: $cnt[N] = \sum_{i=0}^{i=N-1} cnt[i]*cnt[N-i-1]$ where cnt[0], cnt[1] = 1

← M1 : Gearing Up

Applications of Catalan Numbers

We have a 2D grid of dimension MxN. Initially, we are at source(0, 0) and we want to reach the destination(M-1, N-1). At a time we can either move 1 unit downwards or rightwards. Print the total number of unique paths to reach the destination.

Approach:

Any path that we choose is a combination of right('R') and down('D') moves. For a 3x3 grid, the paths can be expressed as-

RRDD
RDRD
RDDR
DDRR
DRDR
DRRD

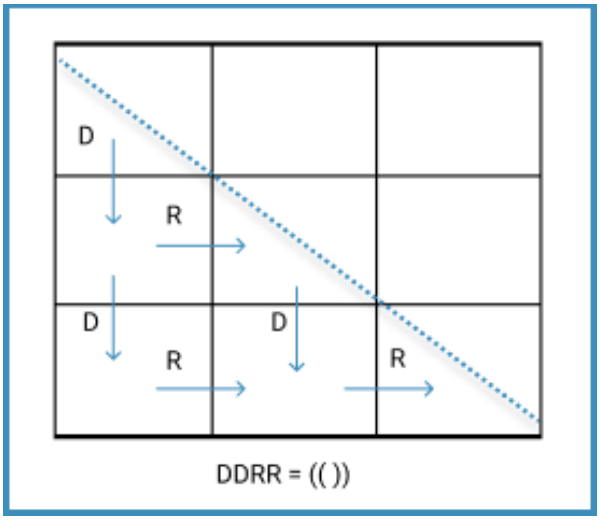
Since each path can be expressed as a string consisting of (M-1) Ds & (N-1) Rs. Therefore, we can say that the number of unique paths will be equal to the number of unique strings consisting of (M-1) Ds and (N-1) Rs.

Answer = $^{M+N-2}C_{M-1}$ or $^{M+N-2}C_{N-1}$

Q. We have been given a square matrix of dimension NxN. Find the total number of unique paths to travel from the source(0, 0) to destination(N-1, N-1) such that no path crosses the major diagonal (i>=j)

Approach:

Since we know the total number of unique paths from source to destination, the answer should be half of it i.e. (2N-2)C(N-1)/2. Is this correct?
No, it is not, test the hypothesis with some examples.
If you observe the right paths carefully then you will realize that for all of them i>=j i.e. for any prefix of their equivalent path strings, cnt('**D**') >= cnt('**R**'). Also for every D, there exists a corresponding R.



The equivalent path string exhibits a behaviour similar to the balanced parentheses. Thus, we can easily map the logic of Catalan numbers here and our answer will be the (N-1)th Catalan number.

Answer: $cnt(N-1) = \sum_{i=0}^{N-2} cnt[i]*cnt[N-i-2]$

Time complexity: O(N^2)
Space complexity: O(N)

M1 : Gearing Up