Number Theory

Number Theory - I (Saturday 26th Jun)

1-D Prefix Sum

Statement:

Given an array `arr[]` of size `n`, its prefix sum array is another array `prefixSum[]` of the same size, such that the value of `prefixSum[i]` is `arr[0] + arr[1] + arr[2] … arr[i]`

In other words we can say the every element in the `prefixSum[]` array is sum of all of its previous elements (including it)

Example:

```
Input  : arr[] = {10, 20, 10, 5, 15}
Output : prefixSum[] = {10, 30, 40, 45, 60}

Explanation : While traversing the array, update
the element by adding it with its previous element.
prefixSum[0] = 10,
prefixSum[1] = prefixSum[0] + arr[1] = 30,
prefixSum[2] = prefixSum[1] + arr[2] = 40 and so on.
```

Lets see this in Code:

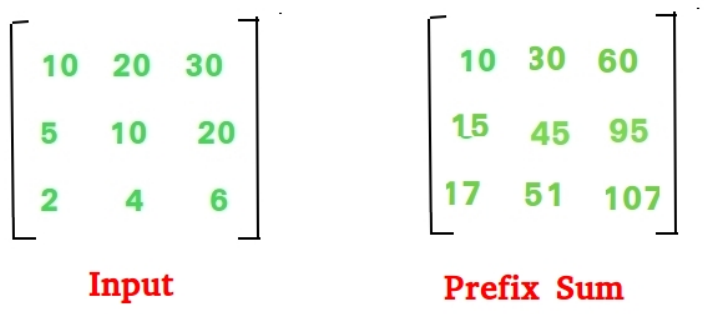
```
// C++ program for Implementing
// prefix sum array
#include <bits/stdc++.h>
using namespace std;

// Fills prefix sum array
void fillPrefixSum(int arr[], int n, int prefixSum[])
{
    prefixSum[0] = arr[0];

    // Adding present element
    // with previous element
    for (int i = 1; i < n; i++)
        prefixSum[i] = prefixSum[i - 1] + arr[i];
}

// Driver Code
int main()
{
    int arr[] = { 10, 4, 16, 20 };
    int n = sizeof(arr) / sizeof(arr[0]);
    int prefixSum[n];

    fillPrefixSum(arr, n, prefixSum);
    for (int i = 0; i < n; i++)
        cout << prefixSum[i] << " ";
}
```

2-D Prefix Sum

Given a 2-D Array or a Matrix find prefix sum of matrix.

Prefix sum matrix: every element of the matrix is the sum elements above and left of it. i.e

`prefixSum[i][j] = arr[i][j] + arr[i-1][j]...arr[0][j] + arr[i][j-1] +... arr[i][0]`



Sliding Window Technique

This technique shows how a nested for loop in some problems can be converted to a single for loop to reduce the time complexity.

Example:

```
Given an array of integers of size ‘n’.
Our aim is to calculate the maximum sum of ‘k’
consecutive elements in the array.

Input  : arr[] = {100, 200, 300, 400}
         k = 2
Output : 700

Input  : arr[] = {1, 4, 2, 10, 23, 3, 1, 0, 20}
         k = 4
Output : 39
We get maximum sum by adding subarray {4, 2, 10, 23}
of size 4.

Input  : arr[] = {2, 3}
         k = 3
Output : Invalid
There is no subarray of size 3 as size of whole
array is 2.
```

Naive Brute force approch

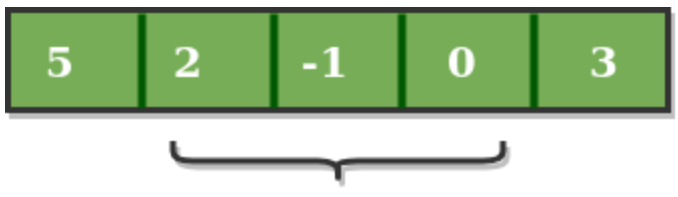
We start with first index and sum till`k-th element. We do it for all possible consecutive blocks or groups of k elements.

```
// O(n*k) solution for finding maximum sum of
// a subarray of size k
#include <bits/stdc++.h>
using namespace std;

// Returns maximum sum in a subarray of size k.
int maxSum(int arr[], int n, int k)
{
    // Initialize result
    int max_sum = INT_MIN;

    // Consider all blocks starting with i.
    for (int i = 0; i < n - k + 1; i++) {
        int current_sum = 0;
        for (int j = 0; j < k; j++)
            current_sum = current_sum + arr[i + j];

        // Update result if required.
        max_sum = max(current_sum, max_sum);
    }

    return max_sum;
}

// Driver code
int main()
{
    int arr[] = { 1, 4, 2, 10, 2, 3, 1, 0, 20 };
    int k = 4;
    int n = sizeof(arr) / sizeof(arr[0]);
    cout << maxSum(arr, n, k);
```

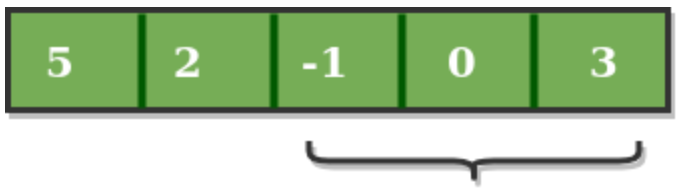
```
    return 0;
}
```

Applying sliding window technique :

- We compute the sum of first k elements out of n terms using a linear loop and store the sum in variable window_sum.
- Then we will graze linearly over the array till it reaches the end and simultaneously keep track of maximum sum.
- To get the current sum of block of k elements just subtract the first element from the previous block and add the last element of the current block .



current_sum = window_sum + (-5) + (0)



current_sum = window_sum + (-2) + (3)

Code For above process

```
// O(n) solution for finding maximum sum of
// a subarray of size k
#include <iostream>
using namespace std;

// Returns maximum sum in a subarray of size k.
int maxSum(int arr[], int n, int k)
{
    // n must be greater
    if (n < k) {
        cout << "Invalid";
        return -1;
    }

    // Compute sum of first window of size k
    int max_sum = 0;
    for (int i = 0; i < k; i++)
        max_sum += arr[i];

    // Compute sums of remaining windows by
    // removing first element of previous
    // window and adding last element of
    // current window.
    int window_sum = max_sum;
    for (int i = k; i < n; i++) {
        window_sum += arr[i] - arr[i - k];
        max_sum = max(max_sum, window_sum);
    }

    return max_sum;
}

// Driver code
int main()
{
    int arr[] = { 1, 4, 2, 10, 2, 3, 1, 0, 20 };
    int k = 4;
    int n = sizeof(arr) / sizeof(arr[0]);
    cout << maxSum(arr, n, k);
    return 0;
}
```

Boom!! Now time Complexity is O(N)

Modular Arithmetic

Modular Addition:

`(a + b) % m = ((a % m) + (b % m)) % m`

Example:

```
(15 + 17) % 7
= ((15 % 7) + (17 % 7)) % 7
= (1 + 3) % 7
= 4 % 7
= 4
```

Modular Multiplication:

`(a x b) % m = ((a % m) x (b % m)) % m`

Example:

```
(12 x 13) % 5
= ((12 % 5) x (13 % 5)) % 5
= (2 x 3) % 5
= 6 % 5
= 1
```

Modular Division

Modular division is totally different from modular addition, subtraction and multiplication. It also does not exist always.

`(a / b) % m != ((a % m) / (b % m)) % m`

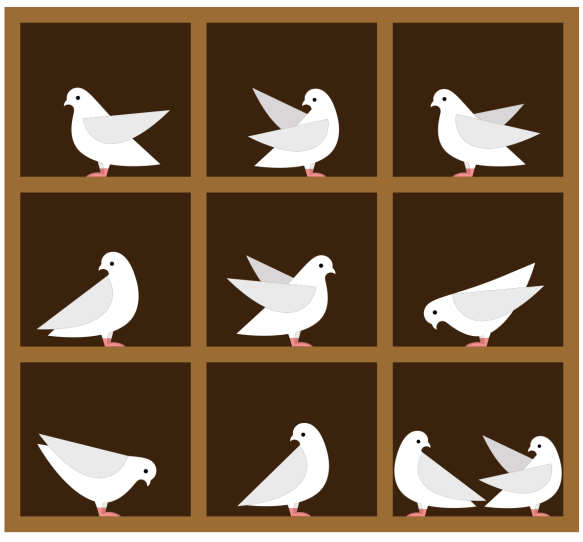
`(a / b) % m = (a x (inverse of b if exists)) % m`

Number Theory - II (Sunday 27th Jun)

Pigeonhole principle

If n pigeonholes are occupied by n+1 or more pigeons, then at least one pigeonhole is occupied by greater than one pigeon.

Generalised pigeonhole principle is:



- If n pigeonholes are occupied by kn+1 or more pigeons, where k is a positive integer, then at least one pigeonhole is occupied by k+1 or more pigeons.

Easy Peezy, Isn't it?

Example:

If `(Kn+1)` pigeons are kept in `n` pigeon holes where `K` is a positive integer, what is the average no. of pigeons per pigeon hole?

- average number of pigeons per hole = `(Kn+1)/n` = `k + 1/n`

Example:

A bag contains 10 red marbles, 10 white marbles, and 10 blue marbles. What is the minimum no. of marbles you have to choose randomly from the bag to ensure that we get 4 marbles of same colour?

- Apply Pigeonhole Principle

No. of colors (pigeonholes) `n` = 3

No. of marbles (pigeons) `K+1` = 4

Therefore the minimum no. of marbles required = `Kn+1`

On solving the equations we get, `Kn+1 = 10`

Verification: ceil[Average] is `[Kn+1/n] = 4`

`[Kn+1/3] = 4`

`Kn+1 = 10`

i.e., 3 red + 3 white + 3 blue + 1(red or white or blue) = 10

Prime Sieve or Sieve of Eratosthenes

Given a number n, print all primes smaller than or equal to n.

```
Input : n =10
Output : 2 3 5 7

Input : n = 20
Output: 2 3 5 7 11 13 17 19
```

The sieve of Eratosthenes is one of the most efficient ways to find all primes smaller than n when n is smaller than 10 million or so

Appoarch:

Lets take an example when n = 50. We need to find all prime numbers which are smaller than or equal to 50.

- we create a array of numbers from 2 to 50

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50

Note the array we are going to create will be a 1-D array for visualisation purpose it has been illustrated as 2D array

- Mark all numbers which are divisible by 2

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50

- Now move to next unmarked number i.e 3 and mark all numbers divisible by 3

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50

- Now move to next unmarked number i.e 5 and mark all numbers divisible by 5

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50

- Continue similar process until the last unmarked number and the table will look something like this

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50

The unmarked numbers are our prime numbers, Lets see this thing in code

```
// C++ program to print all primes
// smaller than or equal to
// n using Sieve of Eratosthenes
#include <bits/stdc++.h>
using namespace std;

void SieveOfEratosthenes(int n)
{
    // Create a boolean array
    // "prime[0..n]" and initialize
    // all entries it as true.
    // A value in prime[i] will
    // finally be false if i is
    // Not a prime, else true.
    bool prime[n + 1];
    memset(prime, true, sizeof(prime));

    for (int p = 2; p * p <= n; p++)
    {
        // If prime[p] is not changed,
        // then it is a prime
        if (prime[p] == true)
        {
            // Update all multiples
            // of p greater than or
            // equal to the square of it
            // numbers which are multiple
            // of p and are less than p^2
            // are already been marked.
            for (int i = p * p; i <= n; i += p)
                prime[i] = false;
        }
    }

    // Print all prime numbers
    for (int p = 2; p <= n; p++)
        if (prime[p])
```

```
            cout << p << " ";
}

// Driver Code
int main()
{
    int n = 30;
    cout << "Following are the prime numbers smaller "
         << " than or equal to " << n << endl;
    SieveOfEratosthenes(n);
    return 0;
}
```

GCD Euclid Theorem

- **G**reatest **C**ommon **D**ivisor (**GCD**) of two or more numbers is the largest positive number that divides all the numbers which are being taken into consideration.
- A simple way to find GCD is to factorize both numbers and multiply common prime factors.

For example:

GCD of 6, 10 is 2 since 2 is the largest positive number that divides both 6 and 10.

Naive Approach

```
int GCD(int A, int B) {
    int m = min(A, B), gcd;
    for(int i = m; i > 0; --i)
        if(A % i == 0 && B % i == 0) {
            gcd = i;
            return gcd;
        }
}
```

Time Complexity: $O(min(A, B))$

Euclid's Algorithm Approach

The algorithm is based on the below facts.

- If we subtract a smaller number from a larger (we reduce a larger number), GCD doesn't change. So if we keep subtracting repeatedly the larger of two, we end up with GCD.
- Now instead of subtraction, if we divide the smaller number, the algorithm stops when we find remainder 0.

Euclid's Algorithm is $GCD(A, B) = GCD(B, A\%B)$.

The algorithm will recurse until $A\%B = 0$.

```
//recursive approach
int GCD(int A, int B) {
    if(B==0)
        return A;
    else
        return GCD(B, A % B);
}
```

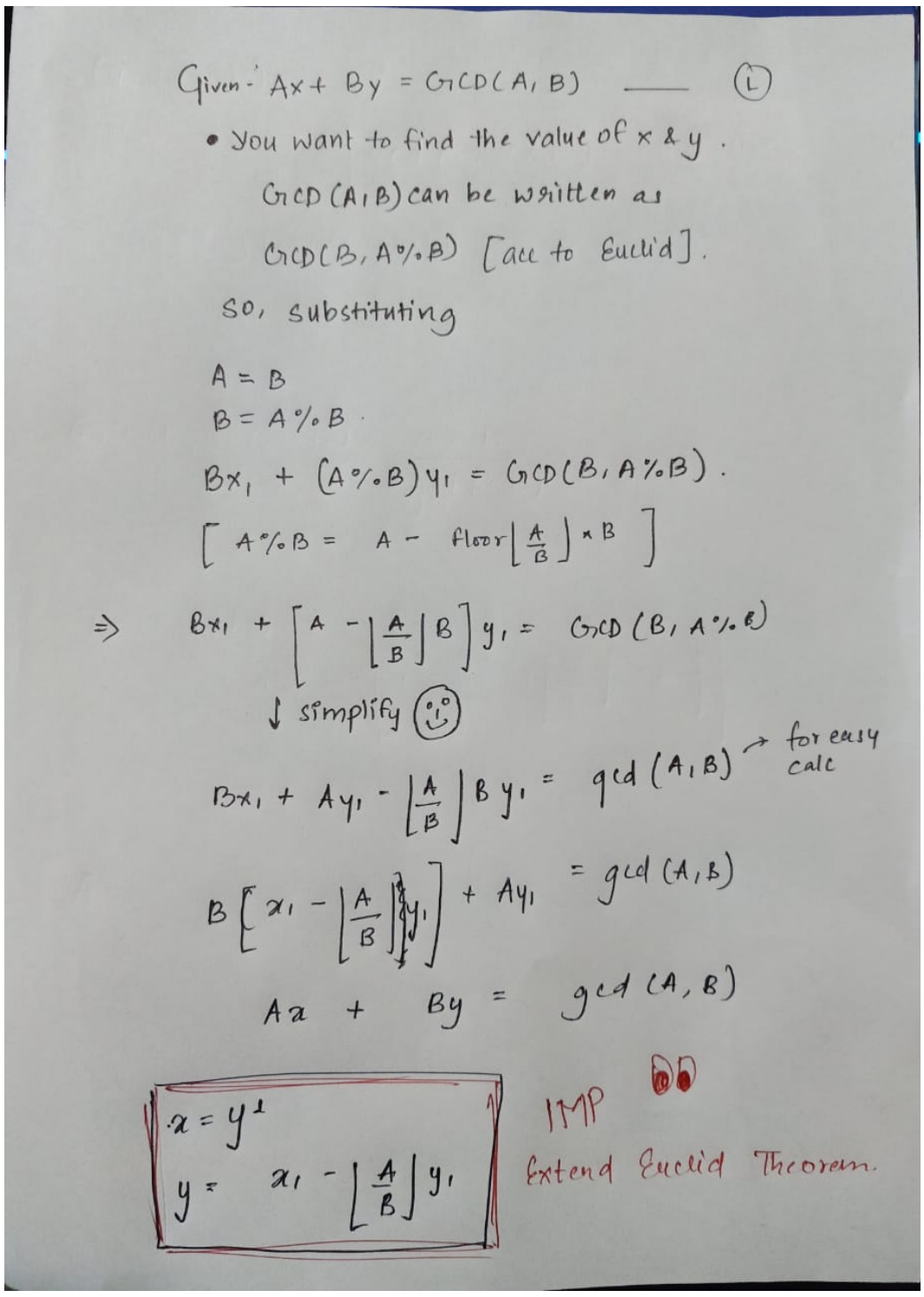
Time Complexity: $O(log(max(A, B)))$

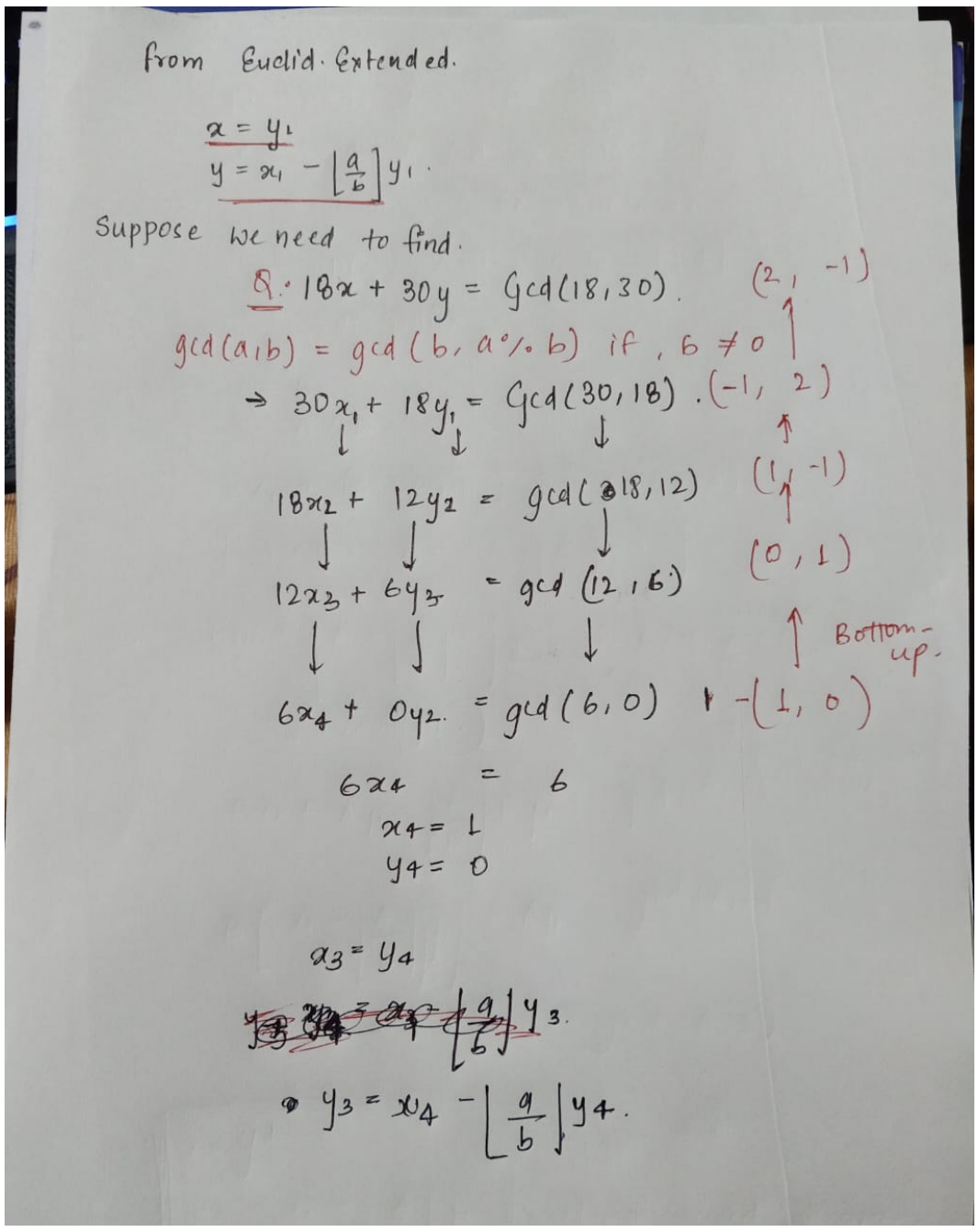
$$O(min(A, B)) > O(log(min(A, B)))$$

Extended GCD Euclid Theorem

- $GCD(A, B)$ has a special property that it can always be represented in the form of an equation, i.e $Ax + By = GCD(A, B)$
- This algorithm takes two inputs as A and B and returns $GCD(A, B)$ and coefficients (x and y**)** of the above equations as output.
- These coefficients can be zero or negative in their value.

Proof





Implementation

```
int d, x, y;
void extendedEuclid(int A, int B) {
    if(B == 0) {
        d = A;
        x = 1;
        y = 0;
    }
    else {
        extendedEuclid(B, A%B);
        int temp = x;
        x = y;
        y = temp - (A/B)*y;
    }
}
```

Time Complexity: $O(log(max(A, B)))$