Stable_sort() ; uses merge sort
TC → $O(n(\log n)^2)$

⑦ Intersection of two arrays :

Simply print intersection of 2 Arrays

```
int f(int a[], int b[], int n, int m)
{
    if(n<m)
        f(b,a,m,n);
    map<int, int> m;
    int ans=0;
    for(int i=0;i<m;i++)
        m[b[i]] += 1;
    for(int i=0;i<n;i++)
    {
        if(m.find(a[i])!=m.end() && m[a[i]]>=1)
        {
            ans++;
            m[a[i]]=0;
        }
    }return ans;
}
```

n = 5, m = 3
a[] = {89, 24, 75, 11, 23}
b[] = {89, 2, 4}

o/p → 1

(8) Count the no. of possible Δ :

n = 5

a[] = {6, 4, 9, 7, 8} o(n log n)

```
int f(int a[], int n)
{
    sort(a, a+n, greater<int>());
    int c=0;
    for(int i=0; i<n-2; i++)
    {
        int l=i+1, h=n-1;
        while(l<h)
        {
            if(a[i]<a[l]+a[h])
            {
                c+=(h-l);
                l++;
            }
            else
                h--;
        }
    }return c;
}
```

(9) Sort by Absolute Difference:

Given an array of N elements and a number K, the task is to arrange array elements according to the absolute difference with K.

* If two or more elements are equal distance arrange them in same sequence as in the given array.

N=5, K=7, a[] = {10, 5, 3, 9, 2}

o/p → 5 9 10 3 2

Using Structure :

(I)
```
struct mycomp{
    int k1;
    mycomp(int k){ k1=k; }

    bool operator() (int a, int b)
    {
        return abs(a-k1) < abs(b-k1);
    }
};
void sortABS( int a[], int n, int k)
{
    stable_sort(a, a+n, mycomp(k));
}
```

(II) Using Class

```
class mycomp{
    int k1;
    public:
    mycomp(int k) : k1(k){}
```

→ Same

if k pass करना है तो ये तरीका follow

(III) Using lambda function :

```
void sortABS(int A[], int n, int k)
{
    stable_sort( a, a+n, [=](int a, int b)
    {
        return abs(a-k) < abs(b-k);
    });
}
```

lambda function

Lambda Expression

→ used to write inline anonymous function
↓
small, simple, small function

→ lambda creation
we need three things [ ] ( ) { };
- inside definition of lambda expression.
- used to pass parameter
- used to capture clause

- We use lambda b/z it is easy to read, cleaner code, keeps everything at same place.

- To pass the value of variable ~~to~~ present in main, we use capture clause. (We capture the variable & pass it to the lambda)

- Remember value of this variable can't be changed inside lambda function (eg: x value in previous example)

- ~~To~~ If you want to change value ~~by~~ pass by reference [&d] ( ) { };

- To pass multiple variable [d, c, k]
If want to change value pass by reference [&d, &c, &k]

★ To pass all variable by reference [&]
To pass ——— value [=]

Example: vector<int> v = {2, 3, 4, 5, 6};
for_each(v.begin(), v.end(), [ ](int x) { cout<<x<<" ";

O/P → 2 3 4 5 6

Brute → $O(n^2)$
Find all pairs $\frac{n(n-1)}{2}$
& sort them

(19) **Smallest Absolute Difference :**

Given → A[], n, K
find K^{th} smallest absolute difference

Binary Search + Sliding Window

Eg. a[] = {1, 2, 5, 10}
absolute differences are {1, 3, 9, 4, 8, 9}
let K=4 o/p → 5 (4th smallest abs difference)

```
int f(int a[], int n, int k)
{
    sort(a, a+n);
    int l = a[1]-a[0], h = a[n-1]-a[0];
    for(int i=1 ; i<n; i++)
        l = min(l, a[i]-a[i-1]);
    while(l<=h)
    {
        int m = (l+h)/2;
        int left=0, count=0;
        for(int right=1; right<n; right++)
        {
            while( a[right]-a[left] <=m)
                left++;
            count += right-left;
        }
        if(count <k)
            l = m+1;
        else
            h = m-1;
    }
    return l;
}
```

By default C++ creates a max-heap for priority Queue

(11) Nearly Sorted Algo:

- Given - a[], k

each element is almost k away from its correct position.

```
vector<int> f(int a[], int n, int k)
{
    priority_queue<int, vector<int>, greater<int>> pq;
    // creating a min-heap priority queue
    vector<int> ans;
    for(int i=0; i<n; i++)
    {
        if(i<k)
            pq.push(a[i]);
        else
        {   pq.push(a[i]);
            ans.push_back(pq.top());
            pq.pop();
        }
    }
    while(!pq.empty())
    {
        ans.push_back(pq.top());
        pq.pop();
    }

    return ans;
}
```

I/p - a[] = {2, 6, 3, 12, 56, 8} k=3

O/p - 2 3 6 8 12 56

(12) Largest Number

- Given non-negative integer Array, arrange them such that they form largest number & return it.
- Since result may be large, return as a String

I/p → [3, 30, 34, 5, 9]
O/p → "9534330"

```
string f(vector<int> &v)
{
    vector<string> s;
    for(int i=0; i<v.size(); i++)
        s.push_back(v[i]);

    sort(s.begin(), s.end(), [](string s1, string s2)
    {
        return s1+s2 > s2+s1;
    });

    string s1;
    for(int i=0; i<s.size(); i++)
    {
        s1 += s[i];
    }
    while(s1[0]=='0' && s1.length()>1)
        s1.pop_back();

    return s1;
}
```

→ lambda function

→ for [0,0,0] array

Matrix

- 2D Array a[3][2] 2 for loop needed to traverse
  3D Array a[2][3][3] 3 ———————

Important Points:

- C++ follow Row-major order to store 2D Matrix
  a[3][2] = {{10,20}
  {30,40},
  {50,60}}

10	20	30	40	50	60
2000	2004	2008	- - -	2020	→ address

- Internal curly brackets are optional in Multidimensional Array.

  a[3][2] = {10,20,30,40,50,60} → not recommended (though to read)

- Only the first dimension can be omitted when we initialize. (compiler automatically figures out)

  eg: a[][2] = {{1,2},{3,4}}
  a[][2][2] = {{{1,2},{3,4}},
  {{5,6},{7,8}}}

- C++ provide variable-sized Array
  int m=3, n=2;
  int a[m][n]; // variable sized Array.

→ More Ways to create multidimensional Array:

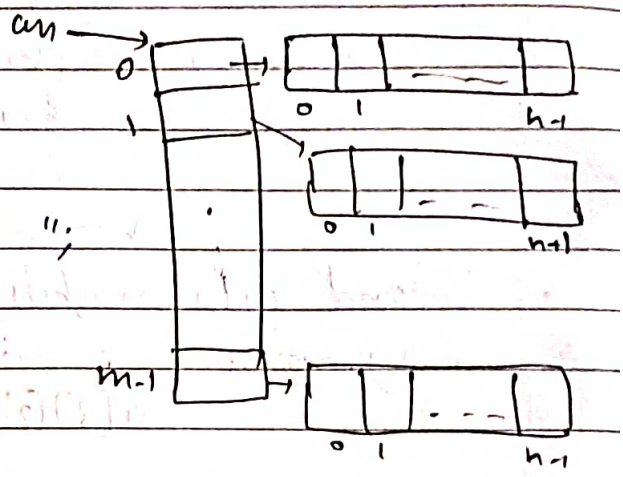
① Using Double Pointer:
- It provide individual row of different sizes.
- DP points to Array of Pointer.

2D Array creation

```
int m=3, n=2;
int **arr;                          // array of pointer
arr = new int *[m];                 // column of size m
for(int i=0; i<m; i++)
    arr[i] = new int[n];            // each row size n
```

```
for(int i=0; i<m; i++)
    for(int j=0; j<n; j++)
    {   a[i][j]=10;
        cout<< a[i][j] <<" ";
    }
```

O/p: 10 10 10 10 10 10



- DP is not cache friendly, b/z there array are not at continuous location.

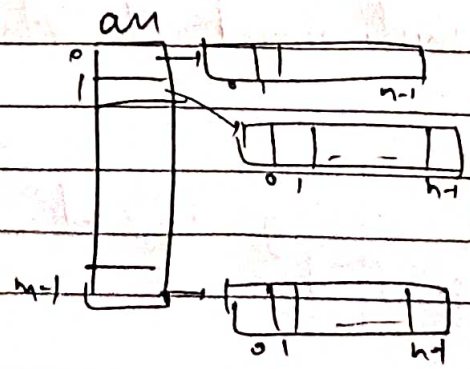
② Array of Pointers

```
int m=3, n=2;
int *arr[m];
for(int i=0; i<m; i++)
    arr[i]= new int[n];
```

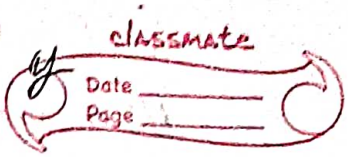
2D Array of size m×n

- Diffrence in above method & this method

D → Earlier Array of pointer was allocated on heap but now ____ is allocated on Stack.



Jagged Array: A 2D array is called JA if every row is of different size



③ STL vector
vector<int> a[m];
- Not cache friendly as simple 2D Array
- Individual row are of dynamic size
- Easy to pass to a function

④ Vector of vector
vector<vector<int>> v;
- Dynamic
- Not cache friendly

→ Passing matrix to a function

① For double pointer
void f(int **a, int n, int m)

② For Array of Pointer
void f(int *a[], int n, int m)

③ Vector
void f(vector<int> v)

④ Array of vector
void f(vector<int> a[], int n)

⑤ Vector of Vector
void f(vector<vector<int>> &v)

[1, 2, 3, 4, 5] ⟶ reverse [5, 4, 3, 2, 1]

// To reverse each element of row
reverse(v[i].begin(), v[i].end());

(I) Transpose of Matrix:

```
void f(vector<vector<int>> &a, int n)
{
    for(int i=0; i<n-1; i++)
    {
        for(int j=i+1; j<n; j++)
            swap(a[i][j], a[j][i]);
    }
}
```

swap

eg:

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Transpose ⟶

1	5	9	13
2	6	10	14
3	7	11	15
4	8	12	16

(II) Rotate by 90°

```
void f(vector<vector<int>> &a, int n)
{
    for(int i=0; i<n-1; i++)
    {
        for(int j=i+1; j<n; j++)
            swap(a[i][j], a[j][i]);
    }
    reverse(a.begin(), a.end()); // swap all rows
}
```

eg:

1	2	3
4	5	6
7	8	9

Transpose ⟶

1	4	7
2	5	8
3	6	9

reverse ⟶

3	6	9
2	5	8
1	4	7

- Multiply 2D matrix → O(N²)
- for upper △ j>i

* for Boundary Traversal Question remember edge case (i) when no. of row & column are 1, there needs to implemented explicitly

(III)

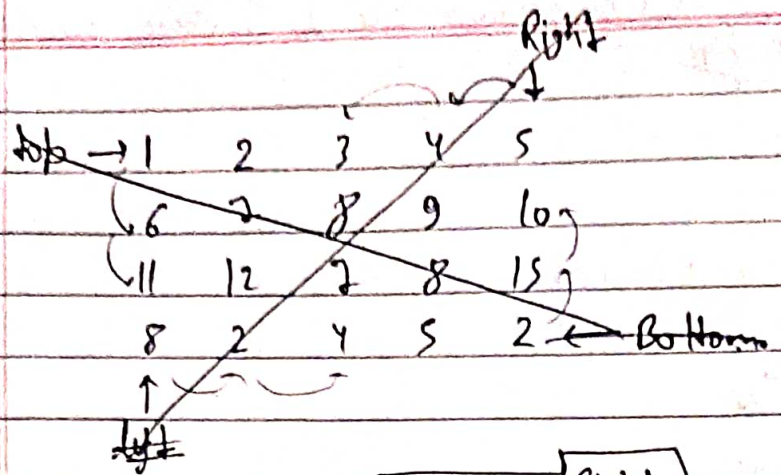
Spiral Matrix :

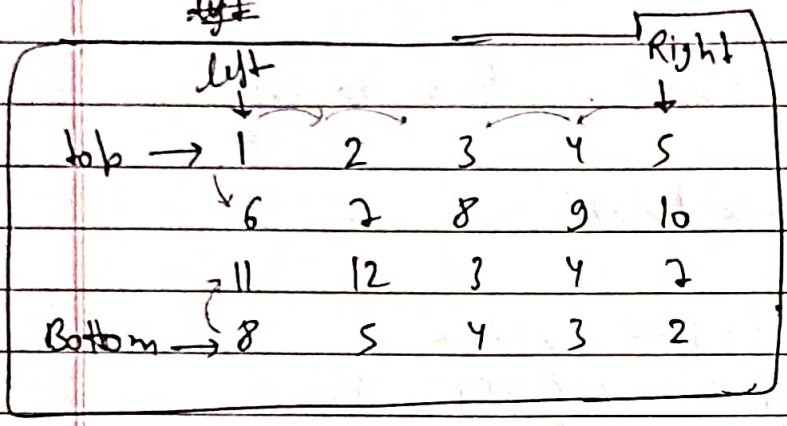
Row → l, r
Column → t, b

```
vector<int> f(vector<vector<int>> &v)
{
    int n=v.size();
    vector<int> ans;
    int l=0, r=v[0].size()-1, t=0, b=n-1;
    while(l<=r && t<=b)
    {
        for(int i=l; i<=r; i++)
            ans.push_back(v[t][i]);
        t++;
        for(int i=t; i<=b; i++)
            ans.push_back(v[i][r]);
        r--;
        if(t<=b)
        {   for(int i=r; i>=l; i--)
                ans.push_back(v[b][i]);
            b--;
        }
        if(l<=r)
        {   for(int i=b; i>=t; i--)
                ans.push_back(v[i][l]);
            l++;
        }
    }
    return ans;
}
```

Using Switch (arr & direction variable)

IInd approach





left & Right now ही traverse के काम आ रहे हैं।

top & bottom column ही traverse कर रहे हैं।

☆ (IV) Search in 2D Matrix : O(R+C)

find target t in matrix m×n

```
bool f(vector<vector<int>> &v, int t)
{
    int n = v.size(), m = v[0].size(), i=0, j=v[0].size()-1;
    while(i<=n-1 && j>=0)
    {
        if(v[i][j]==t)
            return 1;
        else if(t<v[i][j])
            j--;
        else
            i++;
    }
    return 0;
}
```

We start from here

	↑		↓
1	3	5	7
10	11	16	20
23	30	34	60

each row sorted Given
each col sorted "

(V) Make Matrix Beautiful:

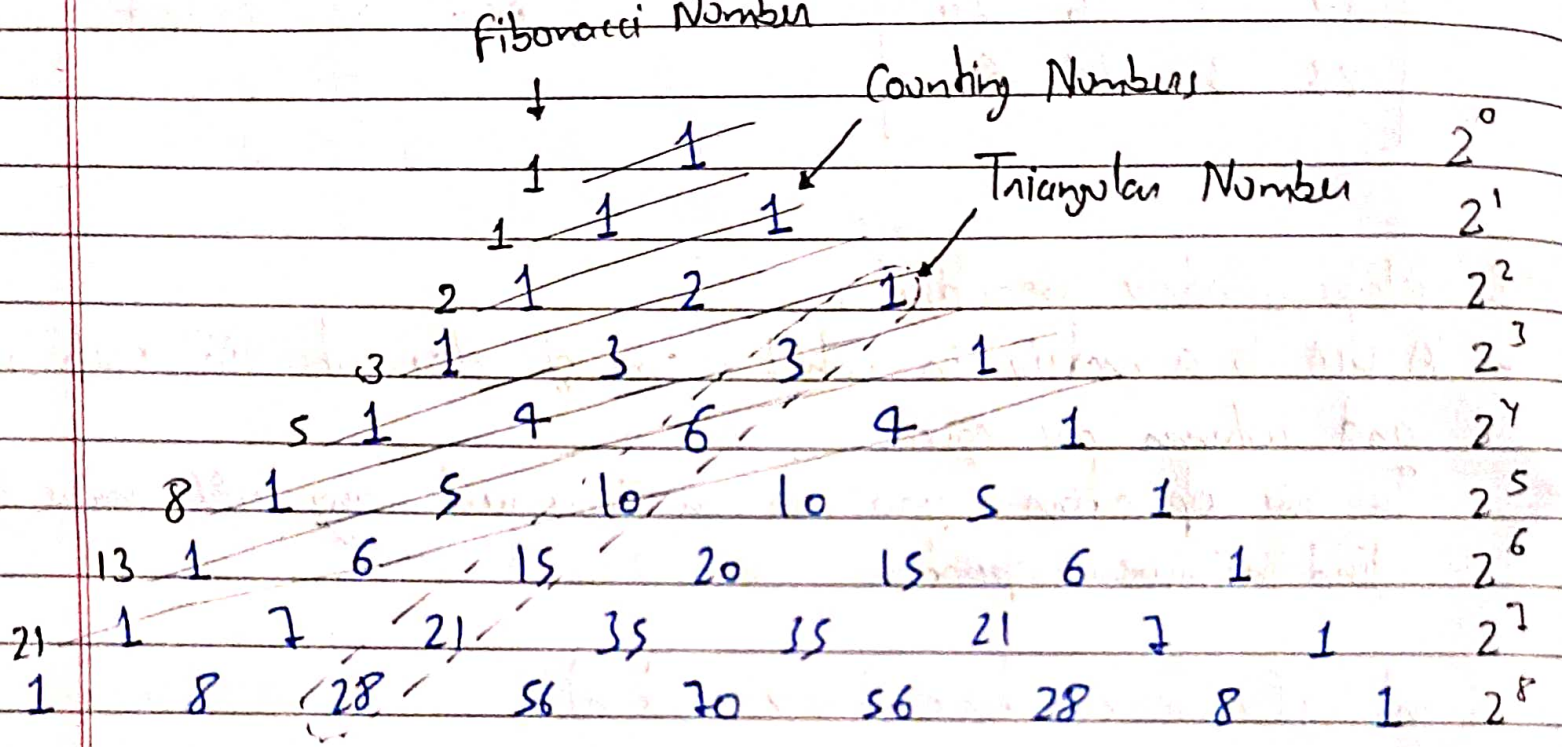
- A BM is a matrix in which sum of elements in each row and column are equal.
- In one operation you can increment any cell value by 1, find minimum operation

```
int f(vector<vector<int>> v, int n)
{
    int sum1=0, sum2=0, ns=0;
    for(int i=0; i<v.size(); i++)
        sum1 += max(sum1, accumulate(v[i].begin(), v[i].end(), 0));

    for(int j=0; j<v.size(); j++)
    {
        ns=0;
        for(int i=0; i<v.size(); i++)
            ns += v[i][j];
        sum2 = max(ns, sum2);
    }
    int sum3 = max(sum1, sum2);
    int total=0;
    for(int i=0; i<v.size(); i++)
        total += accumulate(v[i].begin(), v[i].end(), 0);

    return sum3*(v.size()) - total;
}
```

I/p $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 2 & 3 \\ 3 & 2 & 1 \end{bmatrix}$

O/p → 6

Pascal Triangle



Hashing

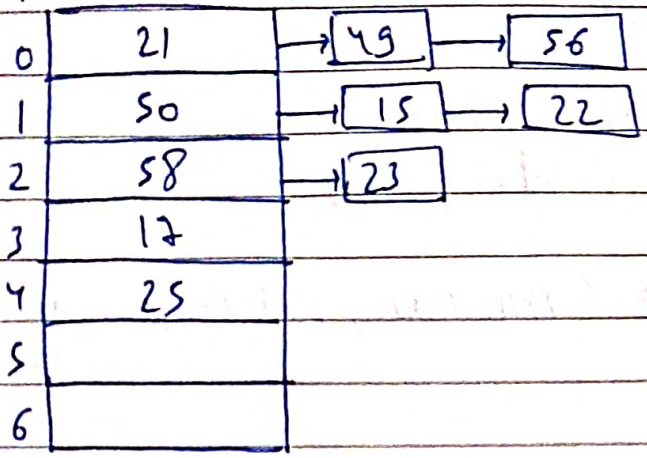
→ Collision detection in hashing

I^st Method:

Chaining

- We maintain Array of linked list

Hash Table



chances of collision is less when we take prime no.

7 linked list header

Hash function = key %. 7 ⇒ hash(key) = key %. 7

key = {50, 21, 58, 17, 15, 49, 56, 22, 23, 25}

- Whenever collision happen we insert element (key) at end of linked list.

Performance of Chaining:

n = no. of slots in Hash Table

n = no. of keys to be inserted

load factor $\alpha = n/m$

- when the keys are uniformly distributed (n keys in n slots)
  ∴ Hashing is O(1) $\alpha = 1$
- Expected time to Search, Insert, Delete = $O(1+\alpha) = O(1)$

→ Data Structure for Storing Chains :

l = chain length

① Linked list
- Search, Delete, Insert $O(l)$
- LL is not cache friendly, nodes at different location

② Dynamic Sized Arrays (Vector)
- S, D, I $O(l)$ • cache friendly

③ Self Balancing BST (AVL Tree, Red Black Tree)
- S, D, I $O(\log l)$
- not cache friendly