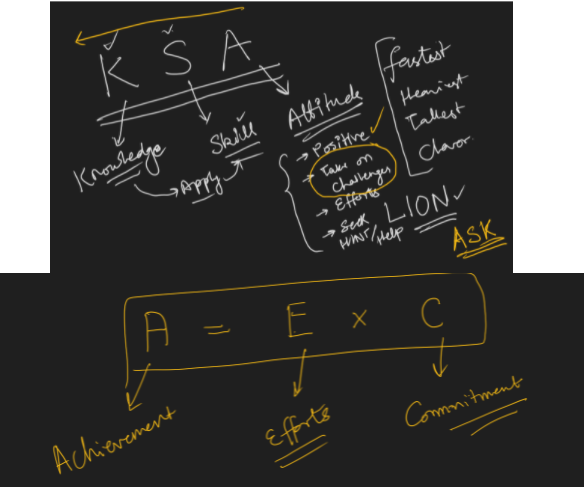
## Learning Objectives

1. Teaching Philosophy/Methodology
   * KSA
   * "Progress should be celebrated more than success".



1. Problem Solving Process
   * Comprehend the Problem
   * Approach (es)
   * Analyze
   * Pick the optimal one based on current problem
   * Code the optimal solution
   * Debug and Fix the Code — Avoid this (proactive think)
2. Concept of Functions
   * Defining
   * Calling
   * No Printing of Answers (Always Returning)
   * Every Problem Discussed will be Function
3. Array vs Dynamic Array
4. Character Array vs String
5. Matrices as List of List - Refer This
   * Magic Square Filling
   * Diagonal Traversals
   * Rotation
6. Online Coding vs Interview

### **Matrix using Dynamic Arrays or Lists**

* Many coding problems require us to return a 2D matrix of dynamic size (rows or columns). So, it is very important for us to understand how to create and return such a list. or How to even represent 2D Matrix as a List of List of Integers.

## UNDERSTANDING THE CONCEPT

* One List of 5 integers represents a 1D Array.  
  List = [1 2 3 4 5]
* 3 Lists of Size 5 Each can represent a 2D Matrix  
  List[0] = [1 2 3 4 5]  
  List[1] = [0 1 2 3 4]  
  List[2] = [5 6 7 8 9]  
    
  Such list will be represented as List< List > listOfLists.
* Let’s Look at the Code now.

**[PROBLEM]** - Read the input of Two D Matrix and create List of Lists (Dynamic in size) and print that list back in the form of matrix (rows and cols).

|  |  |
| --- | --- |
| INPUT FORMAT | Line 1: Number of Test Cases.  Each Test Case Has following Lines:  Test Case Line 1: Integers R C - Rows and Columns  Next R lines contain C Numbers per line representing matrix elements. |
| OUTPUT FORMAT | Print 2D Matrix by making it a list of list. Print it row wise. Print Empty Line of Output between two Matrices. |
| SAMPLE INPUT | SAMPLE OUTPUT |
| 2 3 4 1 2 3 4 4 5 6 7 7 8 9 0 4 9 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 9 9 8 7 6 5 4 3 2 1 0 1 2 3 4 9 8 7 6 | 1 2 3 4 4 5 6 7 7 8 9 0  1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 9 9 8 7 6 5 4 3 2 1 0 1 2 3 4 9 8 7 6 |

#include <cstdio>

#include <vector>

#include <iostream>

using namespace *std*;

void printMatrix(*vector*< *vector*<int> > mat)

{

for (int i = 0; i < mat.*size*(); i++) {

for (int j = 0; j < mat[i].*size*(); j++)

*printf*("%d ", mat[i][j]);

*printf*("\n");

}

*printf*("\n");

}

int main() {

int t; *cin* >> t;

while (t--) {

int r, c;

*cin* >> r >> c;

*vector*< *vector*<int> > v(r);

for (int i = 0; i < r; i++) {

for (int j = 0; j < c; j++) {

int x; *cin* >> x;

v[i].*push\_back*(x);

}

}

printMatrix(v);

}

return 0;

}

1. Complexity Analysis.
   * Time taken by code is - Input, Processing, Output
   * FASTIO to optimized IO (Buffered/Non-Buffered)
2. Sample Problems.
3. Course Layout.

**[PROBLEM]** - Given R and C, dimensions of a 2D MATRIX. Given a cell (i, j) and Direction to move one step 'R' or 'L' or 'D' or 'U' (Right, Left, Down, Up).

Calculate Cell after movement.

**Note:** Matrix is CIRCULAR matrix..

* Moving right from RIGHT MOST COLUMN takes us to LEFT MOST COLUMN.
* MOVING DOWN from last ROW, takes us to FIRST RON.

**INPUT FORMAT** and sample

R c i j D

5 4 1 3 R

**OUTPUT FORMAT** and sample

ni nj

1 6

**PSEUDO CODE:**

*Input*: *i*, *j*, *r*, *c*, dir

if dir == 'R':

if we are in last COL

*print*(*i*, 0)

else

*print*(*i*, *j* + 1)

else if dir == 'L':

if are in *first* COL

*print*(*i*, *c* - 1)

else

*print*(*i*, *j* - 1)

**MODULO ARITHMETIC**

(a % R) = (a + R) % R = (a + 2 \* R) % R = (a + k \* R) % R;

(a + *b*) % R = (a%R + *b* % R) % R;

(a - *b*) % R = (a%R - *b* % R + R) % R;

(a \* *b*) % R = (a%R \* *b*%R) % R;

But

(a / *b*) % R can not be calculated directly.

if (dir == 'R')

j = (j + 1) % c;

else if (dir == 'L')

j = (j - 1 + c) % c;

else if (dir == 'U')

i = (i - 1 + r) % r;

else if (dir == 'D')

i = (i + 1) % r;

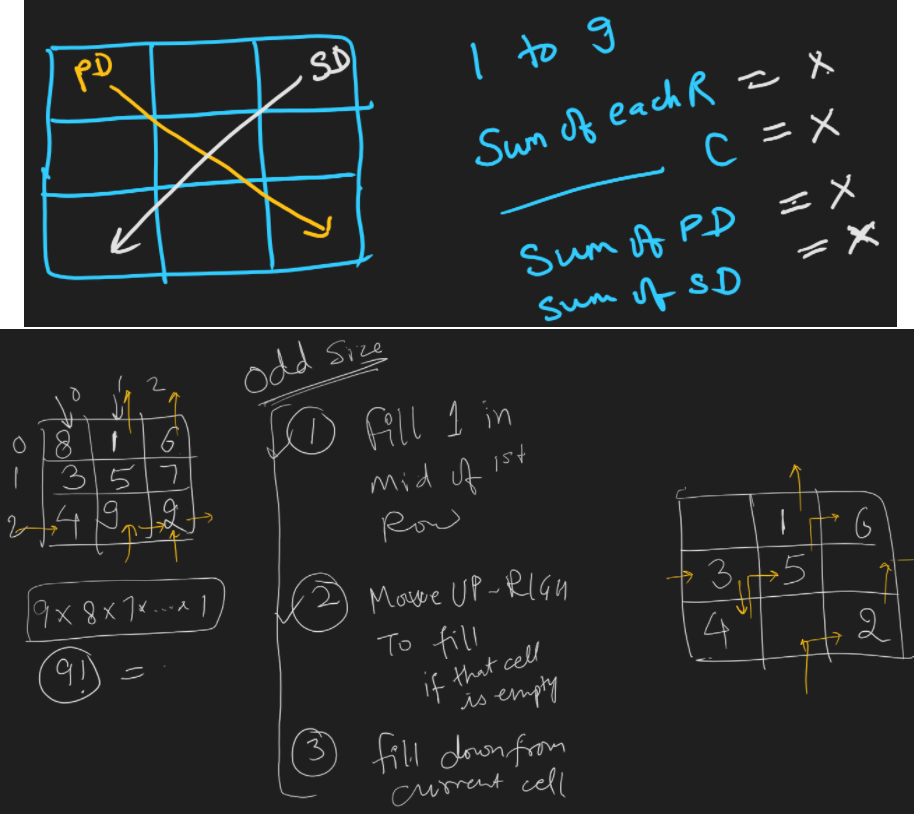
*print*(i, j) // NEXT CELL AFTER MOVEMENT

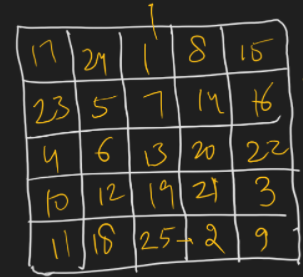
## PRE-REQUISITES

1. Data Types, Variables, Constants
   1. int a = 021; // C++, JAVA, JS - Leading 0 is octal
   2. int a = 0b110; // C++, JAVA — Binary Number
   3. int a = 0x23; // Hex Integral Constant
   4. print(a); // What will this print? 21, 17
2. Input Output Statements
   1. FASTIO to optimized IO (Buffered/Non-Buffered)
3. Operators
   1. Arithmetic Operators
      1. +, -, \*, l, %
         1. DIVISION
            1. float a=4/5;
            2. print(a); // Not give 0.8, because integral div happens first
            3. / in most languages behaves according to data types.
            4. / - when used with integers, gives an integral answer only.
            5. PY has / (floating) and // (integral)
         2. MODULO
            1. % is used to rotate in MATRIX/ARRAY
            2. % is used to extract the rightmost digit from integral values.
   2. Relational operators
      1. <, <=, >, >=, ==, !=
   3. Assignment
      1. =
   4. Logical Operators
      1. &&, ||, ! (AND, OR, NOT)
4. Conditional Statements
5. Loops

### [**PROBLEM**] FILL IN SQUARE

Fill 1 to 9 only once…





### CODING THE APPROACH

Always think in terms of **FUNCTIONS**

* Never print the answer
* Take Input
* Return the Answer

**INPUT:**

* N - Dimension of Square Matrix

**OUTPUT:**

* 2D MATRIX of NXN size

**FUNCTION SIGNATURE:**

* RETURN\_TYPE FUNCTIONAME(P1, P2, ..., PN)

|  |
| --- |
| **C++**  *vector*<*vector*<int>> getMagicSquareOdedSize(int n) {  }  // Fill magic square in given matrix itself, which is pre-allocated  // and provided  void getMagicSquareOdedSize(*vector*<*vector*<int>> &m) {  int r = m.*size*();  int c = m[0].*size*();  } |

## Array vs Dynamic Array

### Static Array

C/C++: int a[*n*]; int b[10];

### Static Size Dynamic Allocated Array

C/C++: int \*a = (int \*)*malloc*(sizeof(int) \* 10); // HEAP

C++: int \*a = new int[*n*];

C#/JAVA: int[] a = new int[*n*];

### Dynamic Array

* Internally uses static size dynamic array.

// ArrayList<Integer> al; // add(4);

// vector<int> v; push\_back(4);

// Python: append(4)

class DynamicArray {

int \*a; // Dynamic Allocated Array

int size;

int capacity;

DyanmicArray() {

capacity = 2;

}

DyanmicArray(int initCapacity) {

capacity = initCapacity;

size = 0;

a = *allocate capacity size memory*

}

void append(int x) {

size++;

if (size >= capacity) {

capacity = 2 \* capacity;

int \*b = *allocate capacity size memory*

*Copy all elements of a to b*

*Point a to b // a = b*

*Free up old space*

}

int getValueAtIndex(int idx) {

return a[idx]; // Without checks

}

int size() {

return this.*size*;

}

bool isEmpty() {

return this.*size* == 0;

}

}

};

DynamicArray a = new DynamicArray(); // Not of fixed size

a.append(4);

## ARRAY and DYNAMIC ARRAY

int a[*n*]; // C++

int a[] = new int[*n*]; // JAVA/C#

*vector*<int> a; // C++

*List*<Integer> a = new ArrayList<>(); // JAVA

*List*<int> a = new *List*<int>(); // C#

a = [] # Python

**[PROBLEM]** Given an array of integers, return a new array having only even integers from the input array.

**INPUT:**

*A* = { 1, 3, 2, 4, 5, 6, 8, 7, 9 }

**OUTPUT:**

2 4 6 8

**C++**

*vector*<int> getEvenOnIy(*vector*<int> a) {

}

# Complexity Analysis

## Time Complexity of a Function:

* Number of iterations (loop iterations) a function performs.
* It is expressed as a mathematical expression in terms of input size.
* No loop will be considered as 1 iteration.

## Space Complexity of a Function:

* Max number of variables (in addition to input parameters) used by the function is called space
* complexity.
* If we use an array of size N, we count them as N variables.
* Expressed as a mathematical formula in terms of input size.
* Consider scope rules also (prefer using scope variables)

**[PROBLEM]** Given N, tell if N is even or not.

TC: 1

SC: 0

bool isEven(int n) {

if (n % 2 == 0) return true;

else return false;

}

bool isEven(int n) {

return (n % 2 == 0);

}

**[PROBLEM]** Given N, find and return the sum of numbers from 1 to N.

**CONSTRAINTS**

1 <= N <= 1012

* Integers have max value of **231-1** (2147483647) **(2 \* 109)**
* long (in JAVA), long long in C++ have range up to 263-1 **(8 \* 1018)**

INPUT:

N = 5

OUTPUT:

15

long long sumOfOneToN(int n) {

long long s = 0;

for (int i = 1; i <= n; i++)

s = s + i;

return s;

}

//TC: n

//SC: 2

long long sumOfOneToN(int n) {

return (n \* (n + 1) / 2); // Check if this can still overflow

}

//TC: 1

//SC: 0

**If a program performs more than 108 iterations, we get verdict as TLE (Time Limit Exceeded)**

**[PROBLEM]** Given M and N, Find and return sum of all numbers between M and N both inclusive.

**CONSTRAINTS**

1 <= M, N <= 109

INPUT:

M=4

N=6

OUTPUT:

15

EXAMPLE:

M=4, N=7 [4+5+6+7]

28 - 6 = 22

M = 4, N = 6 [4+5+6]

21 - 6 = 15

|  |  |
| --- | --- |
| long long sumOfRange(int m, int n) {  if (n < m){  int temp;  //swap n and m using temp  }  long long s = 0;  for (int i = m; i <= n; i++)  s = s + i;  return s;  } | long long sumOfRange(long long m, long long n) {  if (n < m)  {  // swap n and m  }  return (n\*(n + 1)) / 2 - (m\*(m - 1)) / 2;  } |
| TLE | ACCEPTED |

* Comprehension.
* Start from Brute Force.
* Pay attention to constraints.
* Analyzing the Complexity of Solution.

# 3 More Scenarios

## Sequential Loops Add Up

TC: n+m

SC: 2

void testingLoops(int n, int m) {

int s = 0;

for (int i = 0; i <= n; i++) // n + 1

s = s + i;

for (int j = 1; j < m; j++) // m -1

s = s + j;

*print*(s);

}

## Nested Independent Loops Multiply

TC: (n+1)\*(m-1)

SC: 3

void testingLoops(int n, int m) {

int s = 0;

for (int i = 0; i <= n; i++) // (n+1)

for (int j = 1; j < m; j++) // (m-1)

s = s + i + j;

*print*(s);

}

## Nested Dependent Loops

TC: **(n+1)(n+2)/2**

SC: 3

void testingLoops(int n, int m) {

int s = 0;

for (int i = 0; i <= n; i++)

for (int j = 1; j <= (i + 1); j++)

s = s + i + j;

*print*(s);

}

* i =0, inner loop runs 1 time.
* i =1, runs 2 times
* i =2, runs 3 times
* i =n, runs (n+1) times
* 1 + 2 + ... + (n+1) = SIGMA(n+1) = ((n+1) \* (n+1+1) )/ 2 =**(n+1)(n+2)/2**