Programming Pathshala

4

M1 : Gearing Up

An **array** is a collection of elements of the same data type stored in contiguous memory cells. It has a fixed size and is by default passed by reference to a function.

```
Eg. int arr[6] = {4, -3, 8, 5, -1, 6};
```

It initializes an integer array 'arr' storing 6 elements.

Assuming the size of an integer to be 4 bytes and base cell address to be 2000. It can be represented as -

2000	2004	2008	2012	2016	2020
4	-3	8	5	-1	6

A **dynamic array** is similar to a static array but it has the ability to automatically resize itself when an element is inserted or deleted.

They are available as vectors in C++ and likewise lists in Java.

```
Eg. list<int> l;
    vector<int> v;
```

Vectors are slightly less efficient than static arrays due to the occasional resizing and copying of elements.

The amortized time complexity of insertion in a dynamic array is O(1).

[Amortized Time complexity = No. of operations / No. of pushbacks]

By default, vectors are passed by value to a function.

Note: Sometimes you may get TLE on passing vectors to a function. The intention there can be to use Pass by Reference instead of Pass by Value.
Pass by Reference: ***O(1)***
Pass by Value: ***O(N)***

Pre-computation Techniques

In this lecture, we will learn the applications of **pre-computation techniques** like prefix sum, prefix max, suffix max etc. Pre-processing helps in efficiently answering multiple queries and is used with linear data structures like arrays and vectors.

In the given problem, there is an integer array 'Arr[N]' and we have to print the individual sum of 'Q' subarrays whose first index - 'l' and the last index - 'r' is given.

Approach:

Brute force - Create two nested loops to print the sum of different subarrays respectively.
Time Complexity: **O(Q*N)**
Space Complexity: **O(1)**

Prefix sum - We can pre-compute the prefix sum of the array and store it in an array PS[N].

PS[i] = Arr[i] + PS[i - 1];

We can then print the sum of subarray(l, r) within O(1) time complexity.

Sum of Subarray(l, r) = PS[r] - PS[l - 1] (l>=1)
                      = PS[r]   (l==0)

Time Complexity: **O(N+Q)**

Space Complexity: **O(N)**

Note: It can also be solved in O(1) space complexity if we use the original array to store the prefix sum.

Maximize the Expression

In this lecture, we will look at another problem based on the pre-computation technique. Here, we have been given an array 'Arr[N]' and we have to maximise 's'
s = p*Arri + q*Arrj + r*Arrk where p, q, r ∈ Z and i < j < k

Approach:

Brute force - Run three nested loops to find the sum of all possible combinations and hence the maximum value of 's'.
Time Complexity: **O(N^3)**
Space Complexity: **O(1)**

How about choosing the first three largest elements to calculate 's'?
No, it may not work since we are not dealing with positive numbers alone. Let us see an example -

Arr[5] = { 1, 2, 3, 4, -5}, p = 1, q = 2, r = -3
According to the above approach,
s = 1(2) + 2(3) + (-3)(4) = -4, which is not the correct answer.

What if we know all the three maximum terms separately?
Let a = p*Arri, b = q*Arrj and c = r*Arrk. If we can fix the second term (b) and find the maximum possible value of a & c corresponding to that b. And if we repeat the process for all the possible values of b then we can easily find the maximum value of 's'.

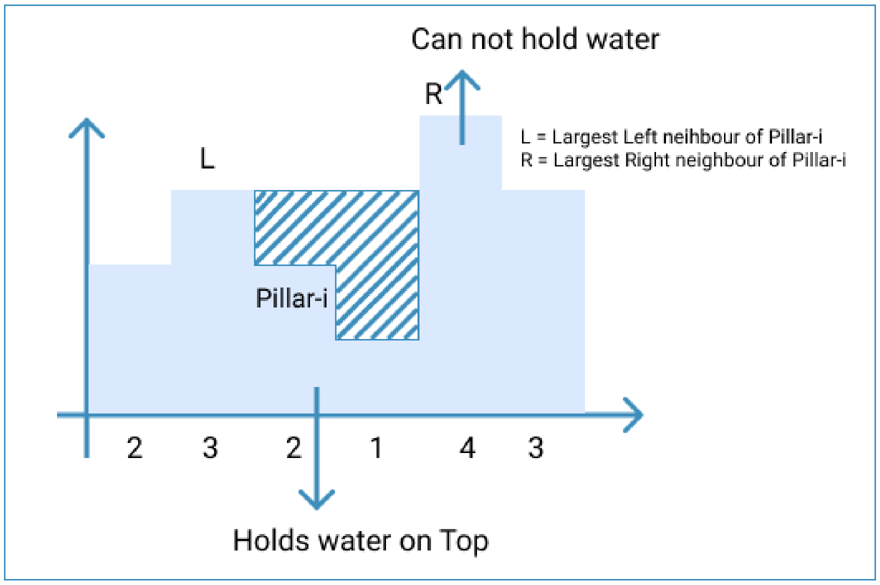
M1 : Gearing Up

Histogram Problem

In this lecture, we will discuss the Rainwater collection problem. Here, we have been given an integer array 'Arr[N]' containing the heights of 'N' pillars. We have to find the total height of the water column trapped between the pillars after the rainfall.

Approach:

Which pillars will hold water on their tops?


Can not hold water
R
L
L = Largest Left neihbour of Pillar-i
R = Largest Right neighbour of Pillar-i
Pillar-i
2
3
2
1
4
3
Holds water on Top

It can be seen in the above illustration, only the pillars with a larger pillar in their left and right neighbourhood can hold water.

Will the boundary pillars hold water?
No, since there is no supporting pillar on the other side.
Solution - If we know the largest left and right neighbours for each pillar then we can find the amount of water that it can hold on its top.

Amount of water on the top of pillar 'i' = max(0, (min(largest left neighbour, largest right neighbour) - height of pillar i))

Implementation - Can you think of how to apply a **pre-computation technique** that was taught in the previous lecture?
For pre-processing, we can calculate the prefix and suffix max for each pillar and use it to find the amount of water they will hold on their top.

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

Maximum Chunks

Here we have been given a permutation array 'Arr[N]' - containing all the elements from 0 to N-1. We have to split the array into the maximum number of chunks (contiguous subarrays) such that after sorting all the chunks individually, we get a sorted array.

Input: Arr[5] = { 1, 2, 0, 3, 4 }
Output: 3

We can divide the array into three chunks - [1 0 2 3 4]. After sorting each chunk individually we get a sorted array - [0, 1, 2, 3, 4]

Approach:

What will be the minimum number of chunks?
One, if we consider the whole array as a chunk then sorting the chunk can yield us a sorted array. [5 4 3 2 1] => [1 2 3 4 5]

Observation - Every chunk from index i to j should be a permutation of numbers from i to j.
Eg.   [1 2 0 3 4]
Index: 0 1 2 3 4

Implementation -

Brute Force - Create two nested loops to check each subarray starting from 'i', if it can be chunked or not. If it can be chunked then we can increase the count of the answer variable and move our iterator after that chunk.
Time complexity: **O(N^2)**
Space complexity: **O(1)**

Prefix Max - From the above observation, we can also infer that the chunks are divided at the index where the prefix max is equal to the array index. Since a chunk from the index, i to j is basically a permutation of numbers from i to j.

```
Eg.          [ 1 2 0  3  4 ]
Index:         0 1 2  3  4
Prefix Max:    1 2 2  3  4
```

Time complexity: **O(N)**
Space complexity: **O(1)** - we can use the input array for calculating prefix max.

M1 : Gearing Up

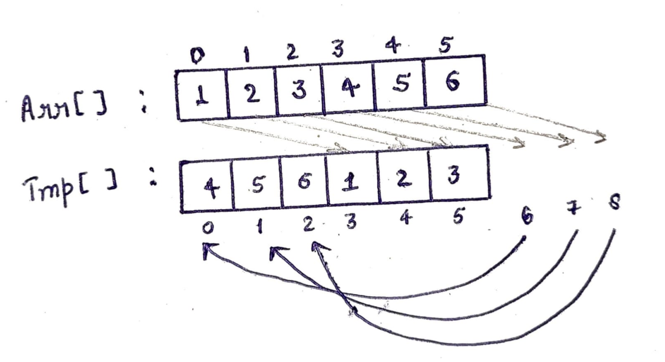
In this lecture, we will learn how to rotate an array by 'k' units clockwise using different methods.

Input: Arr[6] = { 1, 2, 3, 4, 5, 6 }, k = 3
O: Arr[6] = { 4, 5, 6, 1, 2, 3 }

Approach:

Brute Force - Rotate the array clockwise by 1 unit and repeat the steps 'k' times.
Time complexity: **O(kN)**
Space complexity: **O(1)**

Using a temporary array - We can create a temporary array and store the elements at their new position after rotation.



Element Arr[i]	Old Position (i)	New Position
1	0	3
2	1	4
3	2	5
4	3	0(=6%6)
5	4	1(=7%6)
6	5	2(=8%6)

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

3. **Using Array Reversal** -

Input: Arr[5] = { 1 7 3 4 5 }, k = 3

Output: Arr[5] = [3 4 5 1 7]

[1 7 3 4 5] → [3 4 5 1 7]

We can divide the array into two subarrays of length N-k and k. Now, if we carefully look at the reverse of the two subarrays -

reverse of [1 7] → [7 1]

reverse of [3 4 5] → [5 4 3]

Combining the above two reversed sub-arrays, we get → [7 1 5 4 3]

And on reversing the above array, we get → [3 4 5 1 7], which is the desired output.

Note: The above method is based on the logic that an array if reversed twice yields the same array.

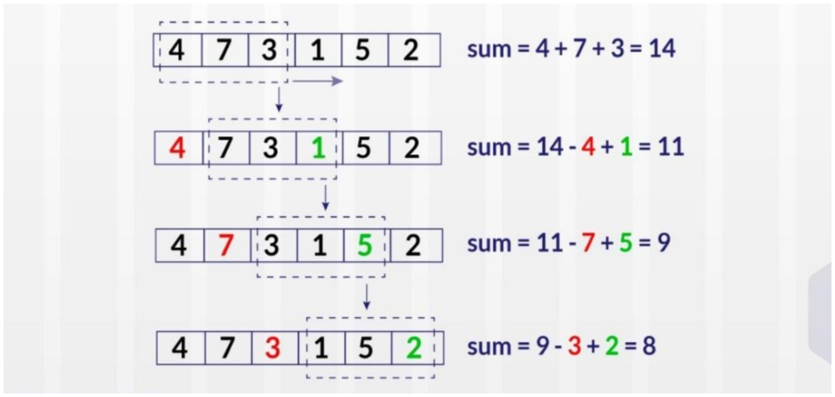
Time complexity: **O(N)**
Space complexity: **O(1)**

M1 : Gearing Up

Approach:

Brute Force - Create two nested loops to find all the subarrays of size 'k' and print their sum.
Time complexity: **O(kN)**
Space complexity: **O(1)**

Sliding Window Technique - In this technique, we first calculate the sum of a window (subarray of size 'k') and then slide the window by one unit each, to find the sum of the remaining subarrays respectively.



Time complexity: **O(N)**
Space complexity: **O(1)**

Reverse Lookup in 1-Dimenssion

In this lecture, we will learn the application of reverse lookup in a one-dimensional array. Here, we have been given an integer array 'Arr[N]' and we have to find the sum of all its subarrays.

Approach:

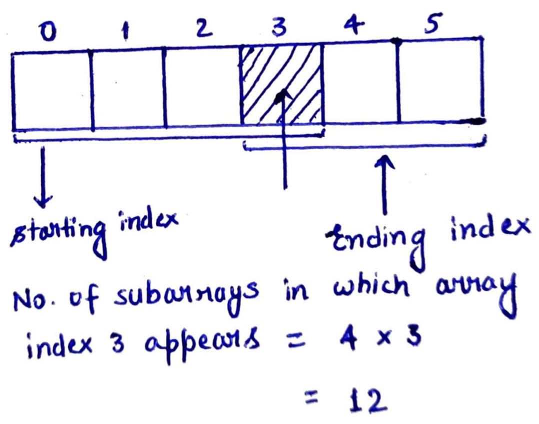
Brute Force - Create two nested loops to find all the subarrays of array 'Arr', followed by another loop to calculate their sum.
Time complexity: **O(N^3)**
Space complexity: **O(1)**

Optimised Brute Force - We can simultaneously compute the sum while finding the subarrays of array 'Arr'.
Time complexity: **O(N^2)**
Space complexity: **O(1)**

An array has N(N+1)/2 subarrays, therefore we can not solve this problem in less than O(N2) time complexity if we are trying to find all the subarrays.

Is there a way to solve it without finding the different subarrays?

Using Reverse Lookup - This method insists to think in a reverse manner. Try to think how many times an element will appear if we consider all the subarrays.



An element at index i will be a part of all the subarrays whose starting index lies between 0 & i and the ending index lies between i to N-1.

Therefore, using the **Rule of Products**, the total number of times an element Arr[i] appears = (i + 1)*(N - i).
Total sum = Σ(Arr[i]*(i+1)*(N-i)) *[from i=0 to i=N-1]*

Time complexity: **O(N)**
Space complexity: **O(1)**

Note: The above method involves integer multiplication and the product may go out of the valid integer range. Therefore, we may have to calculate the modulus of the product each time with some large prime number, for example, 10^9 + 7.

M1 : Gearing Up

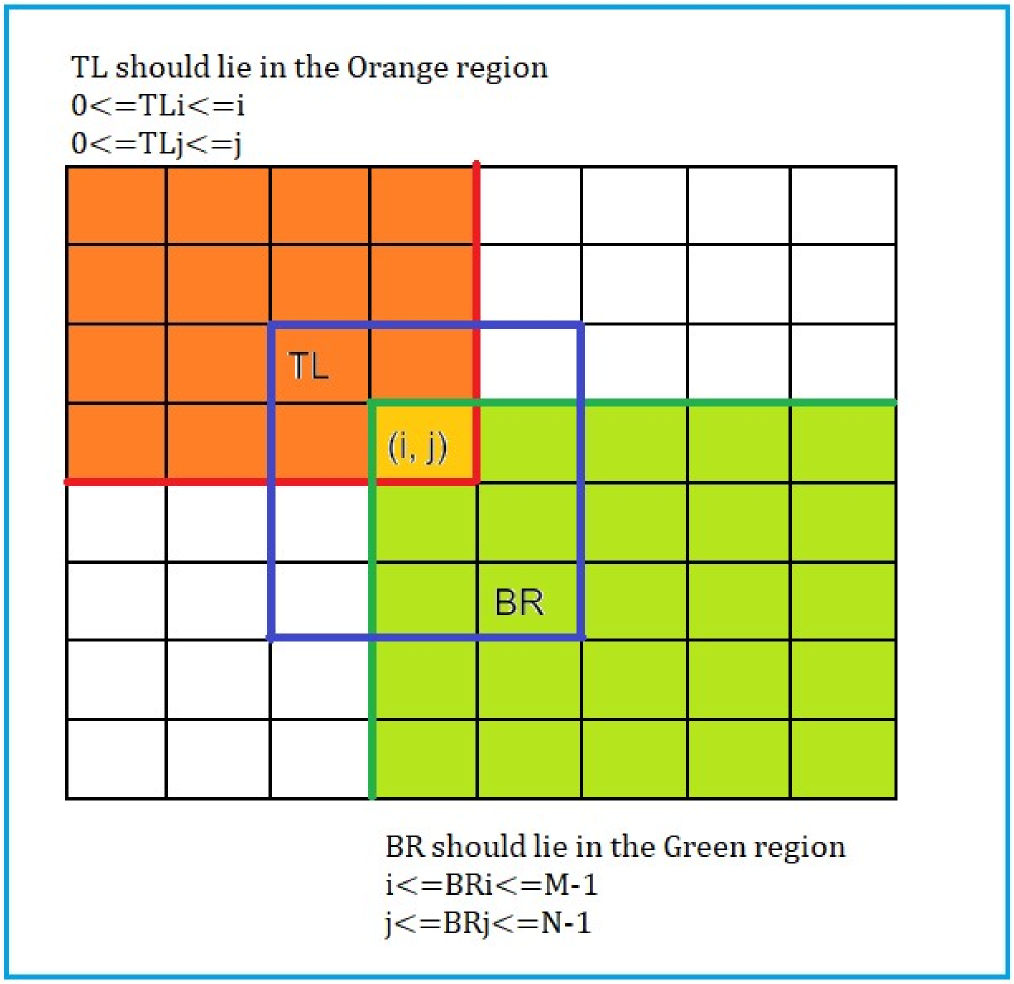
How to define a submatrix uniquely?
We can define a unique submatrix by using the indices of either -

Top Left (TL) & Bottom Right (BR) cell
Bottom Left (BL) & Top Right (TR) cell

Approach:

Brute Force - Create nested loops to iterate through every possible TL & BR valid cell combination to find all the submatrices and calculate their total sum respectively.
Time complexity: **O((M^3)*(N^3))**
Space complexity: **O(1)**

Using Reverse Lookup - Can we use the approach followed in the previous question to find out the contribution of each array element to the total sum?


TL should lie in the Orange region
0<=TLi<=i
0<=TLj<=j
TL
(i, j)
BR
BR should lie in the Green region
i<=BRi<=M-1
j<=BRj<=N-1

An element Arr[i][j] will be a part of all those submatrices who have -

0<=TLi<=i & 0<=TLj<=j
i<=BRi<=M-1 & j<=BRj<=N-1

Therefore, by the Rule of Products, Arr[i][j] appears = (i+1)*(j+1)*(M-i)*(N-j)

Total sum = **Σi(Σj(Arr[i][j]*(i+1)*(j+1)*(M-i)*(N-j)))** *[from i=0 to M-1 & j=0 to N-1]*

Time complexity: **O(MN)**
Space complexity: **O(1)**

Note: The above method involves integer multiplication and the product may be out of the valid integer range. Therefore we may have to calculate the modulus of the product each time by some large prime number say, 10^9 + 7.

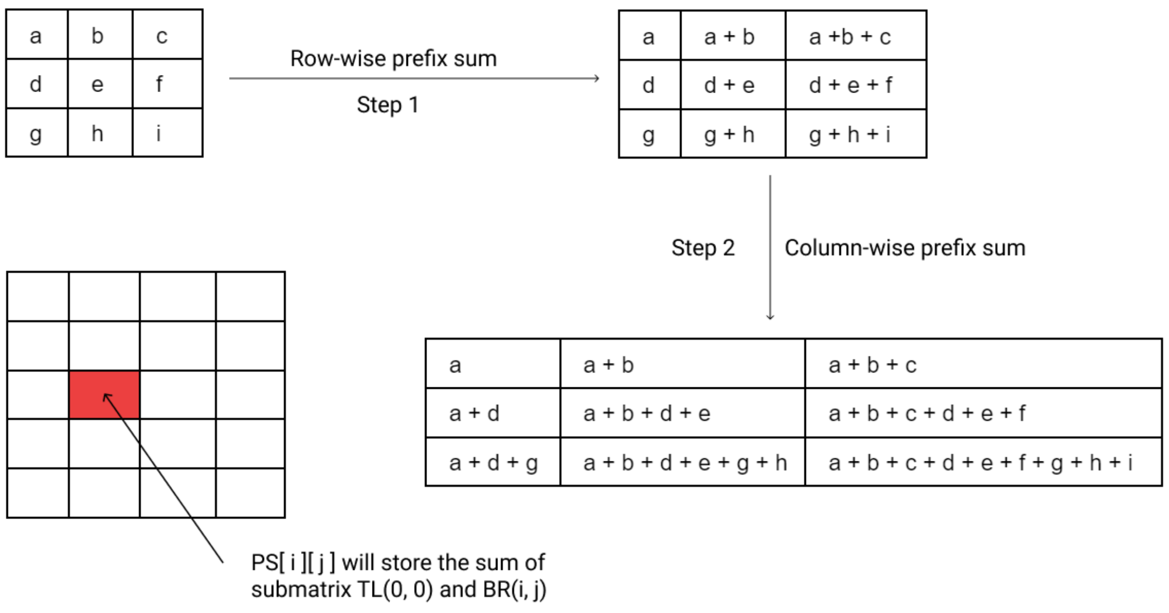
Processing Queries Efficiently

In this lecture, we will learn how to process multiple queries for a 2d matrix efficiently.

Here, we have been given a 2d matrix Arr[M][N] and we have to find the sum of the 'Q' sub-matrices defined by TL(i1, j1) & BR(i2, j2) cells.

Approach:

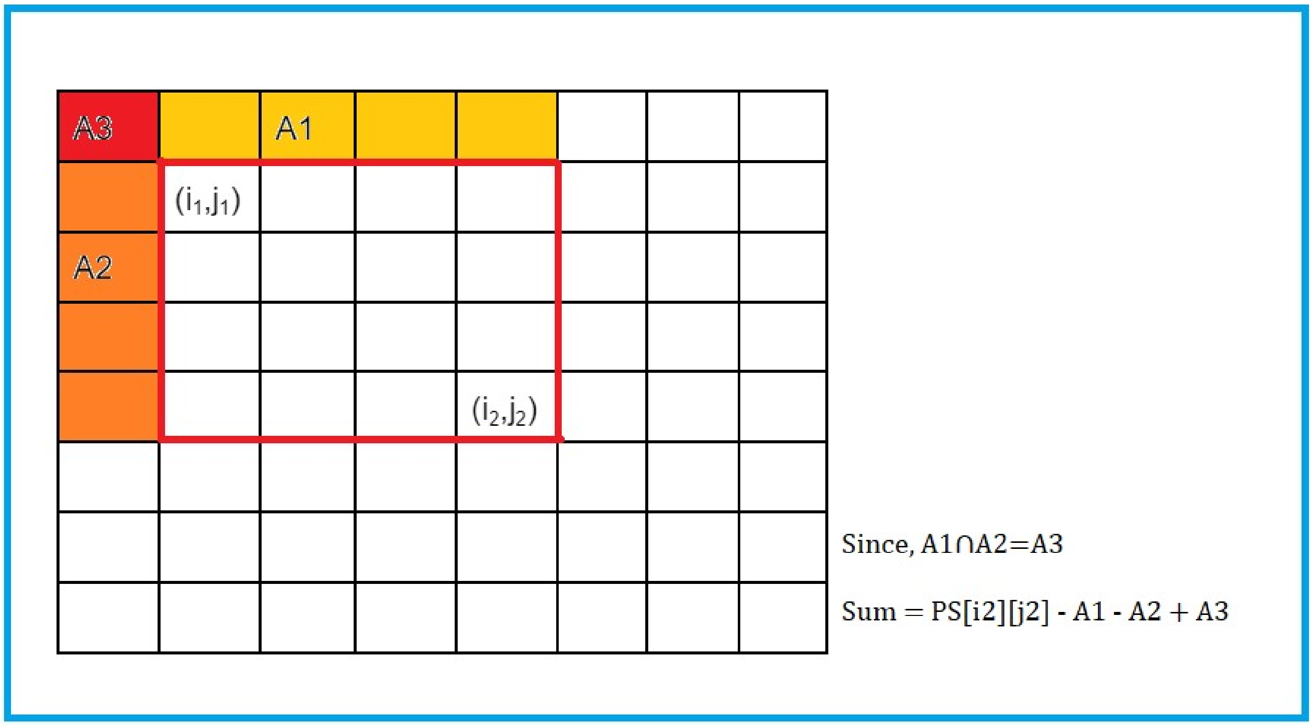
M1 : Gearing Up



a b c
d e f
g h i
Row-wise prefix sum
Step 1
a | a + b | a +b + c
d | d + e | d + e + f
g | g + h | g + h + i
Step 2
Column-wise prefix sum
a | a + b | a + b + c
a + d | a + b + d + e | a + b + c + d + e + f
a + d + g | a + b + d + e + g + h | a + b + c + d + e + f + g + h + i
PS[i][j] will store the sum of submatrix TL(0, 0) and BR(i, j)

Since we now know how to define the Prefix sum for a 2D array, we can create a prefix sum matrix PS[M][N].

For a submatrix: TL(i1, j1) and BR(i2, j2) -



Required Sum = PS[i2][j2] - A1 - A2 + A3 *where,*

A1 = PS[i1-1][j2] ; i1>=1

A2 = PS[i2][j1-1] ;j1>=1
A3 = PS[i1-1][j1-1] ;i1>=1 & j1>=1

Time complexity: **O(Q+MN)**
Space complexity: **O(MN)**

Note: Please check the boundary conditions to ensure that the array indices are non-negative.

← M1 : Gearing Up

the matrix otherwise return (-1, -1).

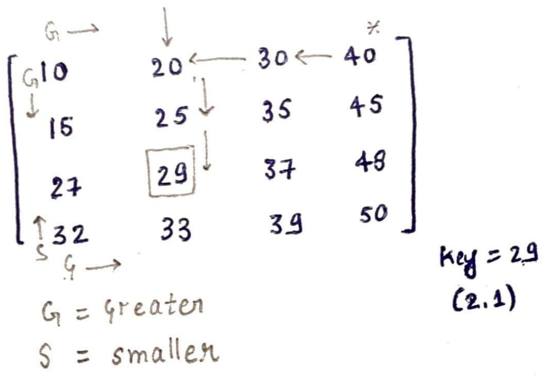
Approach:

Brute Force - Traverse the matrix to search for the key 'k' and return its coordinates if present otherwise returns (-1,-1).
Time complexity: **O(MN)**
Space complexity: **O(1)**

Binary Search - Since the rows are sorted, therefore we can think of applying binary search while searching in each row.
Time complexity: **O(MlogN)**
Space complexity: **O(1)**

The above time complexity could have been achieved even if only the rows were sorted but here the columns are also sorted too. Can we use this fact to optimise it further?

Can we somehow curtail the search space by traversing the matrix in a clever way?
This approach involves observing the elements of the matrix and smartly traversing it to find the key. We can start traversing either from the Top Right(TR) or the Bottom Left(BL) cell of the matrix and move accordingly, comparing the key with the matrix elements.



Time complexity: **O(M+N)**
Space complexity: **O(1)**

Note: We can not start the traversal from Top Left(TL) or Bottom Right(BR) cell because their neighbouring elements are either greater or smaller to them, therefore we do not get any clue on which direction we should move next.

Max Gap Problem: Bucketing

In this lecture, we will learn about the technique of Bucketing. Here we have been given an integer array 'Arr[N]' and we have to return the maximum difference between two consecutive elements when the array is in a sorted state.

Approach:

Brute Force - Sort the array and calculate the maximum difference between two consecutive elements.
Time complexity: **O(NlogN)**
Space complexity: **O(1)**

Using Bucketization - Can you think of the minimum possible value of the answer? Let's call it *gap*.

gap = ceil((Max - Min)/(N - 1))

Eg. 10 _ _ _ _ 20 ; N=6

In this case, the *gap* will be 2, i.e. when the elements are 12, 14, 16, 18 respectively.

Similarly for, 10 _ _ _ _ 21 ; N=6

The *gap* will be 3, i.e. when the elements are 12, 14, 16, 18 respectively.

Since we know that, *Answer >= gap*. Thus, if somehow we are able to create buckets of size '*gap*', then there will not be any need to calculate the difference between the elements which are lying in those particular buckets.

And if we observe carefully, we will realise that our answer will be the maximum of the differences between the max & min elements from consecutive buckets.

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

Summary of the steps:

Find the minimum possible value of the answer ie *gap* = **ceil((max-min)/(N-1))**

Create two temporary arrays maxArr[N] and minArr[N] for storing the max & min of each bucket.

Calculate the bucket number of each element ie **bucket = (Arr[i] - min)/gap** and store the min & max of each bucket in minArr & maxArr.

Note: In the above equation, 'gap' can not be zero (0/0 division), therefore take care of test cases containing all elements as equal.

Iterate on maxArr & minArr to calculate the difference between the maximum & minimum elements from the consecutive buckets.

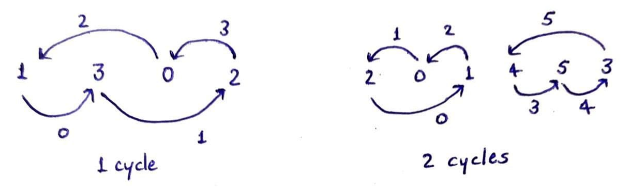
M1 : Gearing Up

Approach:

Using a temporary array - When we try to place the elements at their new indices at that time we may lose the old elements present at those indices. Therefore we can use a temporary array to store the elements at their desired position.
Time complexity: **O(N)**
Space complexity: **O(N)**

Can we do it without using any auxiliary array?

The problem while replacing the old value with the new value is that the previous elements are being overwritten. But can not we do it in a cycle?
Yes, we can!



But as evident from the figure, there may not necessarily be a single cycle. It can contain multiple cycles and thus we would need to track if an element has already been modified in some other cycle or not.

For this, we can make use of the fact that **0<=Arr[i]<=N-1**, therefore we can add 1 to the elements and multiply them by -1.
ie. If element at index 'i' has been visited then, **Arr[i] -> - (Arr[i] + 1)**

Time complexity: **O(N)**
Space complexity: **O(1)**

Array Rearrangement: Alternate Technique

In this lecture, we will tackle the previous question on array rearrangement with an alternate approach.

The problem that we were facing earlier was to preserve the old value while placing the new value at its position.

So, can we think of a number that can represent both the old and the new value together?
Hint: Use the fact that 0<=Arr[i]<=N-1

Let the modified element be x such that **x = N * new_value + old_value**. We would be able to extract both the new & the old value from x as -

old_value = x%N
new_value = x/N

Once we have modified all the elements then we can divide them by N to find out the new values at each array index.
Time complexity: **O(N)**
Space complexity: **O(1)**

Note:

Can we do it the other way around ie. x = N*old_value + new_value.
We can not! since for any untouched element, x/N = 0 (Arr[i]<N-1), therefore we will not be able to extract the old value.
In the above method, we are multiplying N with the new_value, which may lead to integer overflow if N is larger than 10^4. Therefore, it may not work in every case.