**Longest Subarray with sum K | [Postives and Negatives]**

***Approach:*** *Prefix index map*

**Time Complexity:**

* If we use unordered map in Java (like hashmap) – avg and best time complexity to insert and get is O(1), but worst is O(N). Hence, worst time complexity will be O(N.N)
* If we use ordered map in Java (like TreeMap, keys are returned in sorted order in this map – implemented using red-black tree) – avg, best and worst time complexity to insert and get item is O(logN). Hence, worst time complexity will be O(N.logN)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# Sort an array of 0s, 1s and 2s

**Dutch National flag algorithm**

This algorithm contains 3 pointers i.e. low, mid, and high, and 3 main rules. The rules are the following:

* arr[0….low-1] contains 0. [Extreme left part]
* arr[low….mid-1] contains 1.
* arr[high+1….n-1] contains 2. [Extreme right part], n = size of the array

The middle part i.e. arr[mid….high] is the unsorted segment. So, hypothetically the array with different markers will look like the following:

A diagram of numbers and a line

Description automatically generated with medium confidence

In our case, we can assume that the entire given array is unsorted and so we will place the pointers accordingly. For example, if the given array is: [2,0,2,1,1,0], the array with the 3 pointers will look like the following:

A number of objects in a row

Description automatically generated with medium confidence

Here, as the entire array is unsorted, we have placed the mid pointer in the first index and the high pointer in the last index. The low is also pointing to the first index as we have no other index before 0. Here, we are mostly interested in placing the ‘mid’ pointer and the ‘high’ pointer as they represent the unsorted part in the hypothetical array.

Now, let’s understand how the pointers will work to make the array sorted.

**Approach:**

1. First, we will run a loop that will continue until mid <= high.
2. There can be three different values of mid pointer i.e. arr[mid]

* If arr[mid] == 0, we will swap arr[low] and arr[mid] and will increment both low and mid. Now the subarray from index 0 to (low-1) only contains 0. *(solve for [2,0,1] and you will realise why swap is imp.)*
* If arr[mid] == 1, we will just increment the mid pointer and then the index (mid-1) will point to 1 as it should according to the rules.
* If arr[mid] == 2, we will swap arr[mid] and arr[high] and will decrement high. Now the subarray from index high+1 to (n-1) only contains 2.

In this step, we will do nothing to the mid-pointer as even after swapping, the subarray from mid to high(after decrementing high) might be unsorted. So, we will check the value of mid again in the next iteration.

Finally, our array should be sorted.

A screenshot of a graph

Description automatically generatedA screenshot of a computer program

Description automatically generated

A green rectangular object with black text

Description automatically generated

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# next\_permutation : find next lexicographically greater permutation

Intution, given arr  {2,1,5,4,3,0,0}, the next permutation will be {2,3,0,0,1,4,5}.

How did we arrive to it?

🡪 Look from back of arr and check for a drop (it will tell that bigger no is possible). Here, drop is at no 1. Now right after 21xxxxx, will be 23xxxxx. So, we need to replace drop no with a no just greater than him lying on right side (3). After that, we need to rearrange nos left in the right part to get smallest no (sorted order it). So, final no will be – 2300145.

**Steps:**

1. Find the break-point, i: Break-point means the first index i from the back of the given array where arr[i] becomes smaller than arr[i+1].

For example, if the given array is {2,1,5,4,3,0,0}, the break-point will be index 1(0-based indexing). Here from the back of the array, index 1 is the first index where arr[1] i.e. 1 is smaller than arr[i+1] i.e. 5.

To find the break-point, using a loop we will traverse the array backward and store the index i where arr[i] is less than the value at index (i+1) i.e. arr[i+1].

1. If such a break-point does not exist i.e. if the array is sorted in decreasing order, the given permutation is the last one in the sorted order of all possible permutations. So, the next permutation must be the first i.e. the permutation in increasing order.

So, in this case, we will reverse the whole array and will return it as our answer.

1. If a break-point exists:
2. Find the smallest number i.e. > arr[i] and in the right half of index i(i.e. from index i+1 to n-1) and swap it with arr[i].
3. Reverse the entire right half(i.e. from index i+1 to n-1) of index i. And finally, return the array.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# Set Matrix Zero

**Approach 1: Using O(n) space**

1. First, we will declare two arrays: a row array of size N and a col array of size M and both are initialized with 0.
2. Then, we will use two loops(nested loops) to traverse all the cells of the matrix.
3. If any cell (i,j) contains the value 0, we will mark ith index of row array i.e. row[i] and jth index of col array col[j] as 1. It signifies that all the elements in the ith row and jth column will be 0 in the final matrix.
4. We will perform step 3 for every cell containing 0.
5. Finally, we will again traverse the entire matrix and we will put 0 into all the cells (i, j) for which either row[i] or col[j] is marked as 1.
6. Thus we will get our final matrix.

**Approach 2: Using space within matrix**

Intutuion: we keep first row and first column of matrix to store any 0 we encounter within inner array. We additionally keep 2 bool variables to check if the 1st row/col, itself contains 0 or not (In that case, complete qst row/col will be filled with 0).

1. Iterate through matrix and store the 0 info in 1st row/col. If it itself contain 0, make bool variable true
2. Iterate inner matrix and fill row/col zeros acc to value set.
3. Iterate 1st row/col if bool variable is true, and make them 0

int n = matrix.length;

        int m = matrix[0].length;

        boolean rowContainsZero = false, colContainsZero = false;

        for (int r = 0; r < n; r++) {

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

                if (matrix[r][c] == 0) {

                    if(c==0) rowContainsZero = true;

                    if(r==0) colContainsZero = true;

                    matrix[r][0] = 0;

                    matrix[0][c] = 0;

                }

            }

        }

        // Iterate and set 0 for inner matrix

        for (int r = 1; r < n; r++) {

            for (int c = 1; c < m; c++) {

                if (matrix[r][0] == 0 || matrix[0][c] == 0 ) {

                    matrix[r][c] = 0;

                }

            }

        }

        if(rowContainsZero) {

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

                matrix[r][0] = 0;

        }

        if(colContainsZero) {

            for (int c = 0; c < m; c++)

                matrix[0][c] = 0;

        }

    }

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# Spiral Traversal of Matrix

A red arrow pointing to a square with numbers

Description automatically generated

In this approach, we will be using four loops to print all four sides of the matrix.

**1st loop:** This will print the elements from left to right.

**2nd loop:** This will print the elements from top to bottom.

**3rd loop:** This will print the elements from right to left.

**4th loop:** This will print the elements from bottom to top.

**Steps:**

1. Create and initialize variables top as starting row index, bottom as ending row index left as starting column index, and right as ending column index. As shown in the image given below.

A grid with numbers and arrows

Description automatically generated

1. In each outer loop traversal print the elements of a square in a clockwise manner.
2. Print the top row, i.e. Print the elements of the top row from column index left to right and increase the count of the top so that it will move to the next row.
3. Print the right column, i.e. Print the rightmost column from row index top to bottom and decrease the count of right.
4. Print the bottom row, i.e. if top <= bottom, then print the elements of a bottom row from column right to left and decrease the count of bottom
5. Print the left column, i.e. if left <= right, then print the elements of the left column from the bottom row to the top row and increase the count of left.
6. Run a loop until all the squares of loops are printed.

**Note:** As we can see in the code snippet below, two edge conditions are being added in the last two ‘for’ loops: when we are moving from right to left and from bottom to top.

These conditions are added to check if the matrix is a single column or a single row. So, whenever the elements in a single row are traversed they cannot be traversed again backward so the condition is checked in the right-to-left loop. When a single column is present, the condition is checked in the bottom-to-top loop as elements from bottom to top cannot be traversed again.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_