# Problem 1:

**Array Product**

Given an array of numbers *nums* of size *n*, find an array of numbers *products* of size *n*, such that *products[i]* is the product of all numbers *nums[j]*, where *j* != *i*.

**Example One**

Input:

5

1

2

3

4

5

Output:

120

60

40

30

24

Resultant Product array *products* = [products[0], products[1], products[2], products[3], products[4]]

= [(nums[1]\*nums[2]\*nums[3]\*nums[4]), (nums[0]\*nums[2]\*nums[3]\*nums[4]), (nums[0]\*nums[1]\*nums[3]\*nums[4]), (nums[0]\*nums[1]\*nums[2]\*nums[4]), (nums[0]\*nums[1]\*nums[2]\*nums[3])]

= [(2\*3\*4\*5), (1\*3\*4\*5), (1\*2\*4\*5), (1\*2\*3\*5), (1\*2\*3\*4)]

= [120, 60, 40, 30, 24]

**Example Two**

Input:

3

4

9

10

**Output:**

90

40

36

Resultant Product array products = [products[0], products[1], products[2]]

= [(nums[1]\*nums[2]), (nums[0]\*nums[2]), (nums[0]\*nums[1])]

= [(9\*10), (4\*10), (4\*9)]

 = [90, 40, 36]

Output: Return an array of numbers *products*, denoting the required product array where *products[i]* is the (product of all numbers *nums[j]*)% (10^9 + 7), where *j* != *i*.

Constraints:

* You can't use division anywhere in solution.
* 2 <= *n* <= 100000
* -10^9 <= *nums*[i] <= 10^9, i = 0, 1, 2, … , n-1
* *products*[i] >=0, i = 0, 1, 2, ... , n-1
* You are allowed to use only constant extra space and the resultant product array will not be considered extra space.

Usage of resultant products array will not be considered as extra space used. Without using division is the key constraint to remember.

# Solution

Are you getting a wrong answer for some of the test cases, but still think your logic is correct?

Check for the overflow.

If a = 10^9, b = 10^9 and we do, int c = (a \* b) % (10^9 + 7), then it will overflow. Instead, use something like int c = (a \* (long long int) 1 \* b) % (10^9 + 7), to avoid overflow. By multiplying with (long long int) 1, we make sure that the calculation is done in long long int, instead of int.

We provided two solutions.

**1) brute\_force\_solution.java**

A naive approach would be to find the i-th element of the output array (i.e. products[i]), iterate over the entire input array to get the product of all elements nums[j], such that j != i.

**Time Complexity:**

O(*n\*n*) where *n* is length of the input array. As we are iterating over the entire array to find products[i] and as it can be 0<=i<=n-1. Each calculation of element of products array will take O(n) so total complexity will be O(n\*n).

**Auxiliary Space Used:**

O(1).

As we are not storing anything extra and excluding space used to store output array products.

**Space Complexity:**

O(*n*) where *n* is length of the input array.

Storing the input array will take O(n) and auxiliary space used is O(1). So, O(n) + O(1) → O(n).

**2) optimal\_solution.java**

Notice that for products[i], product of all input array elements other than i-th element is nothing but (product of all elements nums[j], 0 <= j <= (i-1)) \* (product of all elements nums[j], (i+1) <= j <= (n-1)) = (nums[0]\*nums[1]\*...\*nums[i-1]) \* (nums[i+1]\*nums[i+2]\*...\*nums[n-1]).

So, iterate over the input array twice to fill output array products, once for updating products[i] with (nums[0] \* nums[1] \*...\*nums[i-1]), and next one for updating products[i] with (nums[i+1] \* nums[i+2] \* … \* nums[n-1]).

**Time Complexity:**

O(*n*) where *n* is the length of the input array.

As we are iterating over the input array two times it will take O(n).

**Auxiliary Space Used:**

O(1).

We are not storing anything extra.

**Space Complexity:**

O(*n*) where *n* is the length of the input array.

Storing the input array will take O(n) and the auxiliary space used is O(1). So, O(n) + O(1) → O(n).

# Problem 2:

**Merge Overlapping Intervals**

Given an array of time intervals (in any order) *inputArray*, of size *n*, merge all overlapping intervals into one and return the resulting array *outputArray*, such that no two intervals in *outputArray* are overlapping. In other words, the result array should contain only mutually exclusive intervals. Hence, in *outputArray*, no pair of intervals i and j exists, such that

*outputArray*[i][0] <= *outputArray*[j][0] <= *outputArray*[i][1].

Consider all the intervals as closed intervals. i.e. endpoints of the intervals are inclusive.

**Example One**

Input:

4

2

1 3

5 7

2 4

6 8

Output:

1 4

5 8

The intervals {1,3} and {2,4} overlap with each other, so they should be merged and become {1,4}.

Similarly {5,7} and {6,8} should be merged and become {5,8}.

**Example Two**

Input:

7

2

100 154

13 47

1 5

2 9

7 11

51 51

47 50

Output:

1 11

13 50

51 51

100 154

The intervals {1,5} and {2,9} overlap with each other, so they should be merged and become {1,9}.

Also, {1,9} and {7,11} overlap with each other, so they should be merged and become {1,11}

Similarly, The intervals {13,47} and {47,50} should be merged and become {13,50}.

Intervals {51,51} and {100,154} are kept as it is as they are not overlapping with any other intervals.

Constraints:

* 1 <= *n* <= 10^5
* -10^9 <= *inputArray*[i][0] <= *inputArray*[i][1] <= 10^9,   i=0, 1, ..., (n-1)

# Solution

We provided three solutions.

**1) brute\_force\_solution.java**

A naive approach would be iterating over inputArray,

For 0<=i<=n-1, Check if inputArray[i] is a removed interval.

1. If it’s a removed interval, continue.
2. If it's not a removed interval, compare inputArray[i] with all other intervals for overlapping. Let us say it overlaps with interval inputArray[k], then remove inputArray[k] from array and merge it into the inputArray[i].

For removing an interval from array, one way is to make the interval invalid (i.e. start>end), so that later we can

check if it is removed or not. See implementation for better understanding.

**Time Complexity:**

O(*n\*n*) where *n* is length of inputArray.

As we have to iterate entire input interval array for each interval, time complexity will be O(*n\*n*).

**Auxiliary Space Used:**

O(1).

Here, all updates can be done in inputArray. No extra space is used.

**Space Complexity:**

O(*n*) where *n* is length of inputArray.

For inputArray, it takes O(n) and the auxiliary space used is O(1). So, O(n) + O(1) → O(n).

// -------- START --------

static int[][] getMergedIntervals(int[][] inputArray) {

for (int i = 0; i < inputArray.length; i++) {

// Check if it is an invalid interval

if (isInvalidInterval(inputArray[i])) {

continue;

}

for (int j = 0; j < inputArray.length; j++) {

// Check if it is an invalid interval

if (i == j || isInvalidInterval(inputArray[j])) {

continue;

}

// Check if interval[i] and interval[j] are overlapping

if (is\_overlapping(inputArray[i], inputArray[j])) {

inputArray[i][0] = Math.min(inputArray[i][0], inputArray[j][0]);

inputArray[i][1] = Math.max(inputArray[i][1], inputArray[j][1]);

invalidateInterval(inputArray[j]);

// Here, for removing an interval from inputArray, we will make it an invalid interval

// (i.e. interval.start > interval.end) as actually removing it from array will be an inefficient way.

}

}

}

int outputArraySize = 0;

for (int i = 0; i < inputArray.length; i++) {

// Check if it is an invalid interval.

// Invalid intervals are the ones which are removed(merged into some other intervals).

if (!isInvalidInterval(inputArray[i])) {

outputArraySize++;

}

}

int[][] outputArray = new int[outputArraySize][2];

int ptr = 0;

for (int i = 0; i < inputArray.length; i++) {

if (!isInvalidInterval(inputArray[i])) {

outputArray[ptr++] = inputArray[i];

}

}

return outputArray;

}

private static boolean isInvalidInterval(int[] inputArray) {

return (inputArray[0] > inputArray[1]);

}

private static void invalidateInterval(int[] interval) {

interval[0] = 1;

interval[1] = 0;

}

private static boolean is\_overlapping(int[] interval1, int[] interval2) {

return !isInvalidInterval(interval1) && !isInvalidInterval(interval2) &&

!(interval1[1] < interval2[0] || interval2[1] < interval1[0]);

}

// -------- END --------

}

**2) other\_solution.java**

A more efficient approach.

Sort the interval array in increasing order of start point. Once we have sorted intervals, we can combine all intervals in a linear traversal.

Following is the detailed step by step algorithm.

1. Sort the intervals based on increasing order of starting time.
2. Push the first interval on to a stack.
3. For each interval do the following
4. If the current interval does not overlap with the stack top, push it.
5. If the current interval overlaps with stack top and ending time of current interval is more than that of stack top, update stack top with the ending  time of current interval.
6. At the end stack contains the merged intervals.

**Time Complexity:**

O(*n\*log(n)*) where *n* is length of inputArray.

As we have to sort the interval array, followed by linear traversal, time complexity will be

O(n\*log(n)) + O(n) → O(n\*log(n)).

**Auxiliary Space Used:**

O(*n*) where *n* is length of inputArray.

Here we used a stack. So, auxiliary space used is O(n).

(We ignore the auxiliary space used by the built-in sort function that we use. Depending on implementation, library, language, it can be different.)

**Space Complexity:**

O(*n*) where *n* is length of inputArray.

For inputArray, it takes O(n) and the auxiliary space used is O(n). So, O(n) + O(n) → O(n).

**3) optimal\_solution.java**

Auxiliary space used in the above approach is O(n). It can be reduced.

The idea remains the same as discussed in the previous approach. Sort the interval array in increasing order of starting point.

Once you have sorted intervals, you can combine all intervals in a linear traversal.

Following is the detailed step by step algorithm:

Let last be the last interval of non overlapping intervals. last=0.

Iterating over inputArray, starting from second interval (1<=i<=n-1)

1. Check if inputArray[i] is overlapping with inputArray[last]
2. If overlapping, merge inputArray[i] and inputArray[last], For merging them, it is sufficient to update only endpoint of inputArray[last] as it is guaranteed that starting point of inputArray[last] <= starting point of inputArray[i][0] as array is sorted by starting point.
3. If non overlapping, we increment last and moving on, inputArray[i] is the new interval under test of overlapping with following intervals.
4. repeat step 1 for i=i+1.

**Time Complexity:**

O(*n\*log(n)*) where *n* is length of inputArray.

As we have to sort the interval array, followed by linear traversal, time complexity will be

O(n\*log(n)) + O(n) → O(n\*log(n)).

**Auxiliary Space Used:**

O(1).

Here, all updates can be done in inputArray. So, no extra space is used.

(We ignore the auxiliary space used by the built-in sort function that we use. Depending on implementation, library, language, it can be different.)

**Space Complexity:**

O(*n*) where *n* is length of inputArray.

For inputArray, it takes O(n) and the auxiliary space used is O(1). So, O(n) + O(1) → O(n).

This is not optimal solution; it still takes the space of O(n) for output array because copyOfRange creates copy of array. So space complexity is O(n) and not O(1).

static int[][] getMergedIntervals(int[][] inputArray) {

// Sorting the input interval array by their starting points in increasing order

Arrays.sort(inputArray, (int object1[], int object2[]) -> {

if (object1[0] != object2[0]){

return object1[0] - object2[0];

}

return object1[1] - object2[1]; }

);

int last = 0;

for (int i = 1; i < inputArray.length; i++) {

// Checking if inputArray[last] and inputArray[i] are overlapping or not

if (inputArray[last][1] >= inputArray[i][0]) {

// If overlapping, then merge inputArray[i] into inputArray[last]

// For merging them, it is sufficient to update only endpoint of inputArray[last] as

// it is guaranteed that inputArray[last][0]<=inputArray[i][0] , last<i

inputArray[last][1] = Math.max(inputArray[last][1], inputArray[i][1]);

} else {

// inputArray[last] and inputArray[i] are found non-overlapping.

// Moving on, inputArray[i] is the new interval under test of overlapping with following intervals.

last++;

inputArray[last] = inputArray[i];

}

}

// From index 0 to last of inputArray will contain all non overlapping intervals

return Arrays.copyOfRange(inputArray, 0, last+1);

}

// -------- END --------

}

# Problem 3:

*2D Array Search  
\* You are given a sorted two-dimensional integer array arr of size r \* c,  
\* where all the numbers are in non-decreasing order from left to right and  
\* top to bottom. I.e. arr[i][j] <= arr[i+1][j] and arr[i][j] <= arr[i][j+1]  
\* for all i = 0,1,...,(r - 2) and j = 0,1,...,(c - 2).  
\*  
\* Check if a given number x exists in arr or not. Given an arr,  
\* you have to answer q such queries.  
\*  
\* Example One  
\* Input: arr = [[1, 2, 3, 12], [4, 5, 6, 45], [7, 8, 9, 78]], queries = [6, 7, 23]  
\* Output: [“present”, “present”, “not present”]  
\*  
\* Given number x=6 is present at arr[1][2] and x=7 is present at arr[2][0].  
\* Hence, "present" returned for them, while  
\*  
\* x=23 is not present in arr, hence "not present" returned  
\*  
\* Example Two  
\* Input: arr = [[3, 4], [5, 10]], queries = [12, 32]  
\*  
\* Output: [“not present”, “not present”]  
\* Given number x=12 and x=32 are not present in arr. Hence, "not present"  
\* returned for both of the queries.  
\*  
\* Constraints:  
\*  
\* 1 <= r <= 10^3  
\* 1 <= c <= 10^3  
\* 1 <= q <= 10^4  
\* -10^9 <= arr[i][j] <= 10^9, (i = 0,1,...,(r - 1) and j = 0,1,...,(c - 1))  
\* -10^9 <= x <= 10^9*

# Solution

We provided three solutions.

**1) brute\_force\_solution.java**

A naive approach would be to iterate over the entire input array arr to check if x is present or not.

**Time Complexity:**

O(*r\*c\*q*) where *r* denotes number of rows of *arr*, *c* denotes number of columns of *arr* and *q* denotes number of queries.

As we are iterating over entire array for each query, time complexity will be O(*r\*c*) (for each query) and as there are *q* queries so total time complexity will be O(*r\*c\*q*).

**Auxiliary Space Used:**

O(1).

As we are not storing anything extra.

**Space Complexity:**

O(*r\*c*) where *r* denotes number of rows of *arr* and *c* denotes number of columns of *arr*.

To store input, it would take O(*r\*c*), auxiliary space used is O(1).

So, total space complexity will be O(*r\*c*).

**2) optimal\_solution.java**

1) Start with top right element arr[0][c-1]

2) Loop: compare this element arr[i][j] with x

    -> If arr[i][j] == x, then return "present"

    -> If arr[i][j] < x then move to next row (i.e. arr[i+1][j])

    -> If arr[i][j] > x then move to column to its left (i.e. arr[i][j-1])

3) repeat the steps in #2 till you find element and return "present" OR if out of bound of matrix then break and return "not present"

Let say x is not present in the first i-1 rows.

Let's say in i-th row, arr[i][j] is the largest number smaller than or equal to x.

-> If it is equal to x, then problem solved, directly return “present”.

-> If arr[i][j] < x, it can be implied that x cannot be present at arr[l][m], i < l and j < m as array is row wise and column wise sorted (ascending). So, moving on to the next row, i+1-th row, we can start checking from j-th column (i.e. arr[i+1][j]).

-> If arr[i][j] > x, means element x can be present in the left side of column jth as row and column are sorted in ascending order. So, we start checking it with arr[i][j-1].

**Time Complexity:**

O(*(r+c)\*q*) where *r* denotes number of rows of *arr*, *c* denotes number of columns of *arr* and *q* denotes number of queries.

As for each query maximum iteration over array can be of O(r+c) and as there can be q queries so, total complexity will be O((r+c)\*q).

**Auxiliary Space Used:**

O(1).

As we are not storing anything extra.

**Space Complexity:**

O(*r\*c*) where *r* denotes number of rows of *arr* and *c* denotes number of columns of *arr*.

To store input, it would take O(*r\*c*), auxiliary space used is O(1).

So, total space complexity will be O(*r\*c*).

# Problem 4:

*/\*\*  
 \* Area under histogram  
 \*  
 \* You will be given an array arr of height of bars, of size n.  
 \* You have to answer q queries, where in each query, you will be given left index l  
 \* and right index r. For each query, return the largest rectangular area possible  
 \* in a histogram formed using (right-left+1) bars with array of heights as  
 \* [arr[left], arr[left+1], arr[left+2], ..., arr[right]].  
 \* Largest rectangle can be made of a number of contiguous bars.  
 \* For simplicity, you can assume that all bars have the same width and the  
 \* width is 1 unit.  
 \*  
 \* For example, consider the following histogram with 7 bars of heights  
 \* [6, 2, 5, 4, 5, 1, 6]. The largest possible rectangle possible is 12  
 \*  
 \* Example One  
 \* Input:  
 \*  
 \* arr = [6, 2, 5, 4, 5, 1, 6]  
 \* q = 1  
 \* For 1st query: l = 0 and r = 6.  
 \*  
 \* Output:  
 \* 12  
 \*  
 \* 1st query: A rectangle of area 12 can be formed using 2nd to 4th bar  
 \* (0-based indexing) and has maximum area possible in histogram out of all  
 \* possible rectangles that can be formed using contiguous bar with given  
 \* array of heights [arr[0],…,arr[6]] = [6, 2, 5, 4, 5, 1, 6] as l=0 and r=6.  
 \*  
 \* Example Two  
 \* Input:  
 \* arr = [2, 4, 6, 5, 8]  
 \* q = 2  
 \*  
 \* For 1st query: l = 0 and r = 4.  
 \*  
 \* For 2nd query: l = 3 and r = 3.  
 \*  
 \* Output:  
 \* 16  
 \* 5  
 \*  
 \* 1st query: A rectangle of area 16 can be formed using 1st to 4th bar  
 \* (0-based indexing) and has maximum area possible in histogram out of all  
 \* possible rectangles that can be formed using contiguous bar with given array  
 \* of heights [arr[0], …, arr[4]] = [2, 4, 6, 5, 8] as l=0 and r=4.  
 \*  
 \* 2nd query: A rectangle of area 5 can be formed using 3rd to 3rd bar  
 \* (0-based indexing) and has maximum area possible in histogram out of all  
 \* possible rectangles that can be formed using contiguous bar with given array of  
 \* heights [arr[3]] = [5] as l=3 and r=3.  
  
 \* Constraints:  
 \*  
 \* 1 <= n <= 2\*10^5  
 \* 1 <= q <= 10  
 \* 1 <= arr[i] <= 10^9, i=(0,1,2,3,...,n-1)  
 \* 0 <= l <= r < n for each query.  
 \*  
 \* Approach  
 \*  
 \* Brute Force Approach  
 \*  
 \* 1) Check for all possible rectangles that start at i and ends at j  
 \* 2) Keep the track of smallest height.  
 \* 3) Width will be j - i + 1  
 \* 4) Find the smallest height between i and j and then calculate area  
 \* area = (hsmall) \* (j - i + 1)  
 \* 5) Keep track of maxArea  
 \*  
 \* Time Complexity: O(n^2)  
 \* Space Complexity: O(1)  
 \*  
 \* Optimal Approach  
 \*  
 \* 1) We need to know the left index and right index for bar of height h.  
 \* 2) With the stack approach, the right index is the index when we encounter smaller bar.  
 \* 3) Left index is the prev index.  
 \* 4) We traverse from left to right.  
 \* 5) A bar is pushed to stack if it's greater than the top of the stack  
 \* 6) A bar is popped from stack, when bar of smaller height is seen. When bar is popped  
 \* we calculate the area.  
 \* area = bar \* (right index - left index - 1)  
 \* 7) How do we get the left index and right index of popped bar?  
 \* right index of popped bar == i which is current index.  
 \* Left index = Index of prev item in stack.  
 \*  
 \* 1) Create an empty stack  
 \* 2) i = 1 to r  
 \* i) a[i] >= stack[top]  
 \* push  
 \* ii) a[i] < stack[top]  
 \* pop until top of stack is greater  
 \* for each popped item calculate area  
 \* area = a[pop] \* (i - prev stack - 1)  
 \* 3) If stack is not empty, then keep popping and calculate area  
 \*  
 \* Time Complexity: O(n)  
 \* Space Complexity: O(n)  
 \*  
 \* resources/AreaUnderHistogram1.jpg  
 \* resources/AreaUnderHistogram2.jpg  
 \* resources/AreaUnderHistogram3.jpg  
 \* resources/AreaUnderHistogram4.jpg  
 \* resources/AreaUnderHistogram5.jpg  
 \* resources/AreaUnderHistogram6.jpg  
 \* resources/AreaUnderHistogram7.jpg  
 \*  
 \*/*

# Solution

We provided three solutions.

**1) brute\_force\_solution.java**

A naive approach would be to check the area of all possible rectangles that can be made using bars in the given histogram and find the max area rectangle.

To implement this approach, iterate over all bars j for each bar i, j>=i, find the smallest height bar 'hsmall' from (i, i+1, i+2, ..., j)th bars. Then area of largest rectangle made using (i, i+1, i+2 , ..., j)th bars would be currentArea = (hsmall \* (j - i + 1)). Compare with max area 'maxArea' found till now and replace it if maxArea < currentArea.

**Time Complexity:**

O(*n\*n*) where *n* is length of input array arr.

As we are iterating over all possible subarrays, it will take O(n\*n).

**Auxiliary Space Used:**

O(1).

As we are not storing anything extra.

**Space Complexity:**

O(*n*) where *n* is length of input array arr.

To store array it will take O(*n*) and as auxiliary space used is O(1).

So, O(*n*) + O(1) → O(*n*).

**3) optimal\_solution2.java**

For every bar ‘x’, we calculate the area with ‘x’ as the smallest bar in the rectangle. If we calculate such an area for every bar ‘x’ and find the maximum of all areas, our task is done. How to calculate the area with ‘x’ as the smallest bar? We need to know the index of the first smaller (smaller than ‘x’) bar on the left of ‘x’ and the index of the first smaller bar on the right of ‘x’. Let us call these indexes ‘left index’ and ‘right index’ respectively.

We traverse all bars from left to right, maintaining a stack of bars.

Every bar is pushed to stack once. A bar is popped from stack when a bar of smaller height is seen. When a bar is popped, we calculate the area with the popped bar as the smallest bar. How do we get the left and right indexes of the popped bar – the current index tells us the ‘right index’ and index of previous item in stack is the ‘left index’. Following is the complete algorithm.

1) Create an empty stack.

2) Start from the first bar, and do the following for every bar ‘arr[i]’ where ‘i’ varies from l to r (As we are calling this function for each query with given l and r).

……a) If stack is empty or arr[i] is higher than the bar at top of stack, then push ‘i’ to stack.

……b) If this bar is smaller than the top of stack, then keep removing the top of stack while top of the stack is greater. Let the removed bar be arr[tp]. Calculate area of rectangle with arr[tp] as smallest bar. For arr[tp], the ‘left index’ is the previous (previous to tp) item in stack and ‘right index’ is ‘i’ (current index).

3) If the stack is not empty, then one by one remove all bars from the stack and do step 2.b for every removed bar.

**Time Complexity:**

O(*n*) where *n* is length of input array arr.

As we are traversing through array to calculate our answer and size of array is r-l+1 in worst case it can be n. So, time complexity will be O(n).

**Auxiliary Space Used:**

O(*n*) where *n* is length of input array arr.

To maintain a stack of size r-l+1 in the worst case it can be n which is length of input array arr. So, auxiliary space used will be O(n).

**Space Complexity:**

O(*n*) where *n* is length of input array arr.

As to store input array it will take O(*n*) and auxiliary space used is O(*n*).

Hence, O(*n*) + O(*n*) → O(*n*).

# Problem 5

*/\*\*  
 \* Pascal's Triangle  
 \*  
 \* Pascal’s triangle is a triangular array of the binomial coefficients.  
 \* Write a function that takes an integer value n as  
 \* input and returns a two-dimensional array representing pascal’s triangle.  
 \*  
 \* pascalTriangleArray is a two-dimensional array of size n\*n, where  
 \* pascalTriangleArray[i][j] = BinomialCoefficient(i, j); if j<=i,  
 \* pascalTriangleArray[i][j] = 0; if j>i  
 \*  
 \* Following are the first 6 rows of Pascal’s Triangle:  
 \*  
 \* 1  
 \* 1 1  
 \* 1 2 1  
 \* 1 3 3 1  
 \* 1 4 6 4 1  
 \* 1 5 10 10 5 1  
 \*  
 \* Example One  
 \* Input: 4  
 \*  
 \* Output:  
 \* 1  
 \* 1 1  
 \* 1 2 1  
 \* 1 3 3 1  
 \*  
 \* Pascal's Triangle for given n=4:  
 \*  
 \* Using equation,  
 \* pascalTriangleArray[i][j] = BinomialCoefficient(i, j); if j<=i,  
 \* pascalTriangleArray[i][j] = 0; if j>i  
 \*  
 \* Generated pascal’s triangle will be:  
 \* 1  
 \* 1 1  
 \* 1 2 1  
 \* 1 3 3 1  
 \*  
 \* Notes  
 \*  
 \* Input Parameters: There is only one argument n, denoting the number of lines  
 \* of Pascal's triangle to be considered.  
 \*  
 \* Output: Return a two-dimensional integer array result, denoting pascal’s  
 \* triangle where each value must be modulo with (10^9 + 7).  
 \* Size of result[i] for 0 <= i < n should be (i + 1)  
 \* i.e. 0s for pascalTriangleArray[i][j] = 0; if j>i, should be ignored.  
 \*  
 \* Constraints:  
 \* 1 <= n <= 1700  
 \**

# Solution

We provided two solutions.

**1) brute\_force\_solution.java**

A naive approach would be to calculate each (binomial coefficient % mod) separately. Binomial coefficient nCr = n!/((n-r)! \* r!). So, calculate numerator = (n! % mod) , denominator = (((n-r)! \* r!) % mod). Finally nCr can be found as ((numerator \* moduloInverse(denominator)) % mod).

**Time Complexity:**

O(*n*^3) where *n* is the given number.

As there are n rows and each row can have n element in worst  cases. For calculating nCr for each element it will take O(n). Hence for n\*n elements it will take O(n^3).

**Auxiliary Space Used:**

O(1).

As we are not storing anything extra. (Here we are ignoring space used to store output 2d array result which will be O(n\*n))

**Space Complexity:**

O(n \* n).

As input is O(1), auxiliary space used is O(1) and output space is O(n \* n).

**2) optimal\_solution.java**

As we know, for Pascal's triangle pascalsTriangle[i][j] = pascalsTriangle[i-1][j] + pascalsTriangle[i-1][j-1] and pascalsTrianlge[i][0] = 1 and pascalsTriangle[i][i]=1. For 0<=i<n and 0<=j<=i.

We use these facts and iterate each row and find out the pascalsTriangle.

**Time Complexity:**

O(*n*^2) where *n* is the given number.

As there are n rows and each row can have n element in worst  cases so, to iterate over n\*n elements it will take O(n^2).

**Auxiliary Space Used:**

O(*1*).

As we are not storing anything extra. (Here we are ignoring space used to store output 2d array result which will be O(n\*n))

**Space Complexity:**

O(n \* n).

As input is O(1), auxiliary space used is O(1) and output space is O(n \* n).

# Problem 6:

**Sum Zero**

Given an array of integers, find any non-empty subarray whose elements sum up to zero.

**Example One**

Input: [5, 1, 2, -3, 7, -4]

Output: [1, 3]

Sum of [1, 2, -3] subarray is zero. It starts at index 1 and ends at index 3 of the given array, so [1, 3] is a correct answer. [3, 5] is another correct answer.

**Example Two**

Input: [1, 2, 3, 5, -9]

Output: [-1]

There is no non-empty subarray with sum zero.

**Notes**

Input Format: Function has one parameter, an integer array.

Output: Return an array of integers. If no zero sum subarray is found, return [-1]. Otherwise return [start\_index, end\_index] of a non-empty zero sum subarray; zero-based indices; both start\_index and end\_index are included in the subarray. If there are multiple such subarrays, any one is a correct answer.

Constraints:

* 1 <= *n* <= 5\*10^5
* -10^9 <= *arr*[i] <= 10^9, (i = 0,1,...,(*n*-1))

Output Format:

There are two cases here:

1. If a valid sum zero subarray exists in *arr*, then there will be two lines for output. First line will have an integer *res*[0] and second line will have an integer *res*[1], denoting starting index and ending index of required subarray (0 based indexing, both inclusive).

2. Otherwise if there is no valid sum zero subarray, there will be only one line for output, having an integer -1.

# Solution

We provided two solutions.

**1) brute\_force\_solution.java**

A naive approach would be to iterate over all possible subarrays of input array arr, such that while on subarray [i,j], i.e. subarray starting from i-th index and ending at jth index, find sum of elements in it and if it's zero, return [i,j]. If no such subarray is found, return [-1].

**Time Complexity:**

O(*n\*n*) where *n* is length of input *arr*.

As we are iterating over all possible subarrays of input array arr, time complexity will be O(*n\*n*).

**Auxiliary Space Used:**

O(1).

We are not storing anything extra.

**Space Complexity:**

O(*n*) where *n* is length of input *arr*.

To store input it takes O(*n*) and as auxiliary space used is O(1).

Hence, O(*n*) + O(1) → O(*n*).

**2) optimal\_solution.java**

An optimal approach would be as follows:

Notice that if there exists a zero sum subarray [i,j] in a given input array arr, then prefix sum (denote it as prefix where prefix[k] = arr[0] + arr[1] + arr[2] + ... + arr[k]) prefix[j] should be equal to prefix[i-1], as prefix[j] = prefix[i-1] + (arr[i] + arr[i+1] + ... + arr[j]), where the term in bracket is sum of subarray [i,j], which is 0.

Considering this fact, build prefix sum array prefix. If for some i, j, 0 <= i <= j < n, prefix[i-1] = prefix[j], then subarray [i,j] is the zero sum subarray.

**Time Complexity:**

O(*n*) where *n* is length of input *arr*.

To find out if any two sums of subarrays are equal or not we will store them in HashMap as prefix[k] (i.e. sum) as key and k as value. To maintain a hashmap it will take O(*n*) time complexity in the worst case to get and store n sums.

**Auxiliary Space Used:**

O(*n*) where *n* is length of input *arr*.

We are using hashmap to store sums. It will take O(n) of space.

**Space Complexity:**

O(*n*) where *n* is length of input *arr*.

To store input it takes O(*n*) and as auxiliary space used is O(*n*).

Hence, O(*n*) + O(*n*) → O(*n*).