Arrays_1

@kbbhatt04

@June 17, 2023

Check if Array Is Sorted and Rotated

- Approach
  - Brute-force
    - Concatenate array with itself
    - Find the break-point
    - `from i=break-point+1 to len(nums)` check `if arr[i] ≤ arr[i+1]`
    - Time Complexity: $O(n)$
    - Space Complexity: $O(n)$
  - Better
    - Find the break-point
    - `while (i+1) % len(nums) ≠ break-point` check `if nums[i] ≤ nums[i+1]`
    - Time Complexity: $O(n)$
    - Space Complexity: $O(1)$
  - Optimal
    - There can be at most 1 break-point
    - Count the number of break-points
    - One-Pass
    - Time Complexity: $O(n)$
    - Space Complexity: $O(1)$

```
# Python3
# Brute-force Solution
class Solution:
    def check(self, nums: List[int]) -> bool:
        i = 0
        while i < len(nums) - 1 and nums[i] <= nums[i+1]:
            i += 1

        if i == (len(nums) - 1):
            if nums[i-1] <= nums[i]:
                return True
            if nums[i] < nums[i-1] and nums[i] <= num[0]:
                return True
            return False

        arr = nums + nums
        for j in range(i+1, len(nums)):
            if arr[j] > arr[j+1]:
                return False
        return True
```

```
# Python3
# Better Solution
class Solution:
    def check(self, nums: List[int]) -> bool:
        i = 0
        while i < len(nums) - 1 and nums[i] <= nums[i+1]:
            i += 1

        if i == (len(nums) - 1):
            if nums[i-1] <= nums[i]:
                return True
            if nums[i] < nums[i-1] and nums[i] <= num[0]:
                return True
            return False

        br = i
        i += 1
        while i != br:
            if nums[i] > nums[(i+1)%len(nums)]:
                return False
            i = (i+1) % len(nums)
        return True
```

```
# Python3
# Optimal Solution
class Solution:
    def check(self, nums: List[int]) -> bool:
        count = 0
        for i in range(len(nums)):
            if nums[i] > nums[(i+1) % len(nums)]:
                count += 1
        return count <= 1
```

```
// C++
// Optimal Solution
class Solution {
public:
    bool check(vector<int>& A) {
        for (int i = 0, k = 0; i < A.size(); ++i)
            if (A[i] > A[(i + 1) % A.size()] && ++k > 1)
                return false;
        return true;
    }
};
```

Remove Duplicates from Sorted Array

- Approach
  - Brute-force
    - Use set to remove duplicates
    - Time Complexity: $O(nlogn)$
    - Space Complexity: $O(n)$
  - Better
    - Use stack

        - if stack.top() ≠ curr_element then push curr_element onto the stack
        - Time Complexity: $O(n)$
        - Space Complexity: $O(n)$
    - Optimal
        - Initialize 2 pointers `i = 0`, `j = 1` where i maintains unique elements count
        - `if nums[i] == nums[j]` then increment j only
        - `else swap(nums[i+1], nums[j])` and increment both pointers
        - Time Complexity: $O(n)$
        - Space Complexity: $O(1)$

```
# Python3
# Brute-force Solution
def removeDuplicates(self, nums: List[int]) -> int:
    nums[:] = sorted(set(nums))
    return len(nums)
```

```
# Python3
# Better Solution
class Solution:
    def removeDuplicates(self, nums: List[int]) -> int:
        stack = [nums[0]]
        for i in range(1, len(nums)):
            if stack[-1] != nums[i]:
                stack += nums[i],
        nums[:] = stack
```

```
# Python3
# Optimal Solution
class Solution:
    def removeDuplicates(self, nums: List[int]) -> int:
        if len(nums) == 1:  return True
        i = 0
        j = 1
        while j != len(nums):
            if nums[i] == nums[j]:
                j += 1
            else:
                nums[i+1], nums[j] = nums[j], nums[i+1]
                i += 1
                j += 1

        return i + 1
```

```
// C++
// Optimal Solution
class Solution {
public:
    int removeDuplicates(vector<int>& nums) {
        if (nums.size() == 1) return 1;
        int i = 0, j = 1;
        while (j != nums.size()) {
            if (nums[i] == nums[j]) {
                j++;
```

```
            }
            else {
                swap(nums[i+1], nums[j]);
                i++;
                j++;
            }
        }
        return i+1;
    }
};
```

Rotate Array to the right by `k` steps

- Approach
  - Brute-force
    - Rotate one element in every iteration
    - Time Complexity: $O(n * k)$
    - Space Complexity: $O(1)$
  - Better
    - Take another array
    - First, Copy the last `k` elements in the new array
    - Then, Copy the first `n-k` elements in the new array
    - Time Complexity: $O(n)$
    - Space Complexity: $O(n)$
  - Optimal
    - Reverse the last `k` elements
    - Reverse first `n-k` elements
    - Reverse the whole array
    - Time Complexity: $O(n)$
    - Space Complexity: $O(1)$

```
# Python3
# Brute-force Solution
class Solution:
    def rotate(self, nums: List[int], k: int) -> None:
        for i in range(k % len(nums)):
            temp = nums[-1]
            for j in range(len(nums) - 1, 0, -1):
                nums[j] = nums[j - 1]
            nums[0] = temp
```

```
# Python3
# Better Solution
class Solution:
    def rotate(self, nums: List[int], k: int) -> None:
        k = k % len(nums)
        ans = []
```

```
        for i in range(len(nums) - k, len(nums)):
            ans += nums[i],
        for i in range(len(nums) - k):
            ans += nums[i],
        nums[:] = ans
```

```
# Python3
# Optimal Solution
class Solution:
    def reverse(self, nums, start, end):
        while start < end:
            nums[start], nums[end] = nums[end], nums[start]
            start += 1
            end -= 1

    def rotate(self, nums: List[int], k: int) -> None:
        n = len(nums)
        k = k % n
        self.reverse(nums, 0, n - k - 1)
        self.reverse(nums, n - k, n - 1)
        self.reverse(nums, 0, n - 1)
```

```
// C++
// Optimal Solution
class Solution {
public:
    void rotate(vector<int>& nums, int k) {
        k = k % nums.size();
        reverse(nums.begin(), nums.end() - k);
        reverse(nums.end() - k, nums.end());
        reverse(nums.begin(), nums.end());
    }
};
```

Single Number

Find the number that appears once, and the other numbers twice - Strivers A2Z DSA

Detailed solution for Find the number that appears once, and the other numbers twice - Problem Statement: Given a non-empty array of integers arr, every element appears twice except for one. Find that single one. Examples: Example 1: Input Format: arr[] = {2,2,1} Result:

https://takeuforward.org/arrays/find-the-number-that-appears-once-and-the-other-numbers-twice/

- Approach
  - Brute-force
    - For every element, count the frequency
    - Time Complexity: $O(n^2)$
    - Space Complexity: $O(1)$
  - Better-1
    - Use unordered_map/dict to store the frequency of each element
    - Time Complexity: $O(n)$
    - Space Complexity: $O(n)$

- Better-2
  - XOR of two same numbers is always 0 i.e. a ^ a = 0
  - XOR of a number with 0 will result in the number itself i.e. 0 ^ a = a
  - Perform the XOR of all the numbers of the array elements, we will be left with a single number
  - Optimal if array is unsorted
  - Time Complexity: $O(n)$
  - Space Complexity: $O(1)$
- Optimal
  - Works IFF array is sorted
  - Binary Search
  - Time Complexity: $O(logn)$
  - Space Complexity: $O(1)$

```
# Python3
# Brute-force Solution
class Solution:
    def singleNumber(self, nums: List[int]) -> int:
        for i in nums:
            cnt = 0
            for j in nums:
                if j == i:
                    cnt += 1
            if cnt == 1:
                return i
```

```
# Python3
# Better Solution - 1
class Solution:
    def singleNumber(self, nums: List[int]) -> int:
        dct = dict()
        for i in nums:
            dct[i] = 0
        for i in nums:
            dct[i] += 1
        for i in dct:
            if dct[i] == 1:
                return i
```

```
// C++
// Better Solution - 1
class Solution {
public:
    int singleNumber(vector<int>& nums) {
        unordered_map<int, int> umap;
        for (int i = 0; i < nums.size(); i++) {
            umap[nums[i]]++;
        }
        for (auto it: umap) {
            if (it.second == 1) return it.first;
        }
        return -1;
```

```
    }
};
```

```
# Python3
# Better Solution - 2
class Solution:
    def search(self, A, N):
        xor = 0
        for i in A:
            xor ^= i
        return xor
```

```
# Python3
# Optimal Solution
# Works iff array is sorted
```

```
// C++
// Optimal Solution
// Works iff array is sorted
class Solution{
public:
  int search(int A[], int N){
        int low = 0;
        int high = N-1;
        while(low <= high)
        {
            int mid = low + (high - low) / 2;
            if(mid % 2 == 0)
            {
                if(A[mid] == A[mid+1])
                    low = mid + 1;
                else
                    high = mid - 1;
            }
            else
            {
                if(A[mid] == A[mid + 1])
                    high = mid-1;
                else
                    low = mid + 1;
            }
        }
        return A[low];
    }
};
```

Next Permutation

- Total Permutations = n!
- Follow Dictionary Sorted order i.e. `{{1,2,3}, {1,3,2}, {2,1,3}, {2,3,1}, {3,1,2}, {3,2,1}}`
- Approach:
    - Brute-force
        - Generate all permutations and apply linear search
        - Time Complexity: $O(n! * n)$
        - Space Complexity: $O(n)$

- Optimal
  - First, we find the breakpoint i.e. point where this condition `nums[i] < nums[i+1]` is met from the back. If there is no such point then the list is already in descending order and the next permutation is the reverse of the given list.
  - When a breakpoint is found we swap it with the first greater element from the back of the list and reverse the list elements on the right of breakpoint.
  - Time Complexity: $O(n)$
  - Space Complexity: $O(1)$

```
# Python3
# Optimal Solution
class Solution:
    def reverse(self, nums, start, stop):
        size = stop + start
        for i in range(start, (size + 1) // 2):
            nums[i], nums[size - i] = nums[size - i], nums[i]

    def nextPermutation(self, nums: List[int]) -> None:
        ind = -1
        # find breakpoint from back
        for i in range(len(nums)-2, -1, -1):
            if nums[i] < nums[i+1]:
                ind = i     # breakpoint found
                break
        if ind == -1:   # no breakpoint found
            nums.reverse()  # in-place
            # nums[:] = nums[::-1] # also works but is not in-place
            # Note that nums = nums[::-1] doesn't work as it is not in-place
            # nums[::-1] creates and returns new list object
        else:
            for i in range(len(nums)-1, ind, -1):
                if nums[i] > nums[ind]:
                    nums[i], nums[ind] = nums[ind], nums[i]
                    break

            # reverse list elements on the right of breakpoint
            self.reverse(nums, ind+1, len(nums)-1) # in-place
            # nums[ind+1:] = nums[ind+1:][::-1] # also works but is not in-place
            # but nums[2:4].reverse() is not correct
```

```
// C++
// Optimal Solution
class Solution {
public:
    void nextPermutation(vector<int>& nums) {
        int ind = -1;
        for (int i = nums.size()-2; i >= 0; i--) {
            if (nums[i] < nums[i+1]) {
                ind = i;
                break;
            }
        }
        if (ind == -1) {
            reverse(nums.begin(), nums.end());
        }
        else {
            for (int i = nums.size()-1; i >= 0; i--) {
                if (nums[i] > nums[ind]) {
                    swap(nums[i], nums[ind]);
                    break;
                }
```

```
            }
            reverse(nums.begin()+ind+1, nums.end());
        }
    }
};
```

Maximum Subarray Sum (Kadane's Algorithm)

- Approach
  - Brute-force
    - Check the sum of every possible subarray and consider the maximum among them
    - Time Complexity: $O(n^3)$
    - Space Complexity: $O(1)$
  - Better
    - If we carefully observe, we can notice that to get the sum of the current subarray we just need to add the current element(i.e. arr[j]) to the sum of the previous subarray i.e. arr[i….j-1].
    - Time Complexity: $O(n^2)$
    - Space Complexity: $O(1)$
  - Optimal
    - The intuition of the algorithm is not to consider the subarray as a part of the answer if its sum is less than 0. A subarray with a sum less than 0 will always reduce our answer and so this type of subarray cannot be a part of the subarray with maximum sum.
    - Here, we will iterate the given array with a single loop and while iterating we will add the elements in a sum variable. Now, if at any point the sum becomes less than 0, we will set the sum as 0 as we are not going to consider any subarray with a negative sum. Among all the sums calculated, we will consider the maximum one.
    - Time Complexity: $O(n)$
    - Space Complexity: $O(1)$

```
# Python3
# Brute-force Solution
def maxSubarraySum(arr, n):
    maxi = -sys.maxsize - 1  # maximum sum

    for i in range(n):
        for j in range(i, n):
            # subarray = arr[i.....j]
            sum = 0

            # add all the elements of subarray:
            for k in range(i, j+1):
                sum += arr[k]

            maxi = max(maxi, sum)

    return maxi
```

```python
# Python3
# Better Solution
def maxSubarraySum(arr, n):
    maxi = -sys.maxsize - 1 # maximum sum

    for i in range(n):
        sum = 0
        for j in range(i, n):
            # current subarray = arr[i.....j]

            #add the current element arr[j]
            # to the sum i.e. sum of arr[i...j-1]
            sum += arr[j]

            maxi = max(maxi, sum) # getting the maximum

    return maxi
```

```python
# Python3
# Optimal Solution (Kadane's Algorithm)
class Solution:
    def maxSubArray(self, nums: List[int]) -> int:
        maxsum = nums[0]
        tempsum = 0

        for i in nums:
            tempsum += i
            maxsum = max(maxsum, tempsum)
            if tempsum < 0:  tempsum = 0
        return maxsum
```

```cpp
// C++
// Optimal Solution (Kadane's Algorithm)
class Solution {
public:
    int maxSubArray(vector<int>& nums) {
        int maxsum = nums[0];
        int tempsum = 0;

        for (auto i: nums) {
            tempsum += i;
            if (tempsum > maxsum) {
                maxsum = ts;
            }
            if (tempsum < 0) {
                tempsum = 0;
            }
        }

        return maxsum;
    }
};
```

Print Subarray with Maximum Sum (variation of Kadane’s Algorithm)

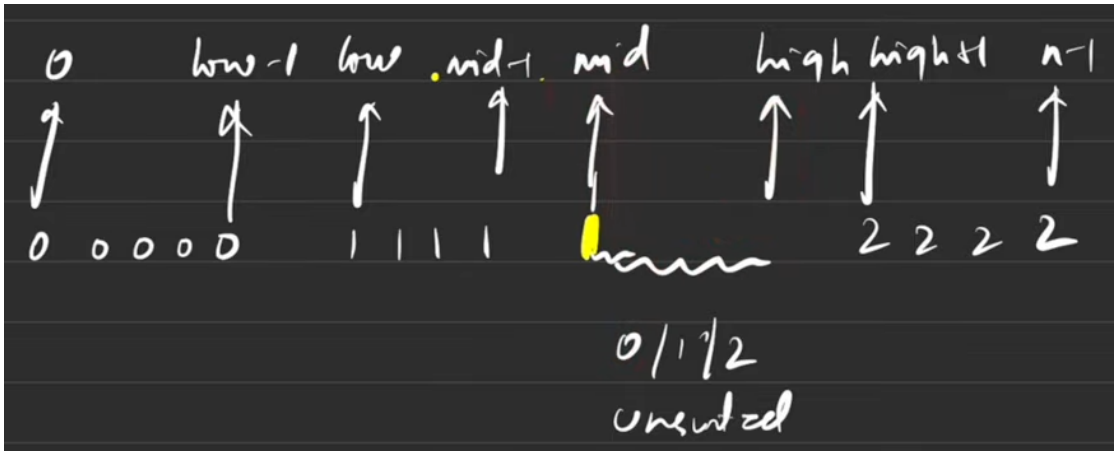
- Approach
  - Optimal
    - Store the starting index and the ending index of the subarray. Thus we can easily get the subarray afterwards without actually storing the subarray elements.

        - Time Complexity: $O(n)$
        - Space Complexity: $O(1)$

```
# Python3
# Optimal Solution
def maxSubarraySum(arr, n):
    maxi = -sys.maxsize - 1  # maximum sum
    sum = 0

    start = 0
    ansStart, ansEnd = -1, -1
    for i in range(n):
        if sum == 0:
            start = i  # starting index

        sum += arr[i]
        if sum > maxi:
            maxi = sum

            ansStart = start
            ansEnd = i

        if sum < 0:
            sum = 0

    print(arr[ansStart:ansEnd+1])
```

Sort an array of 0's, 1's and 2's (Dutch National Flag Algorithm)

- Approach
    - Brute-force
        - Apply Merge-Sort Algorithm or any in-built Sorting Algorithm
        - Time Complexity: $O(nlogn)$
        - Space Complexity: $O(n)$
    - Better
        - Maintain 3 counter variables that store the number of occurrences of 0's, 1's and 2's
        - Two-Pass
        - Time Complexity: $O(n)$
        - Space Complexity: $O(1)$
    - Optimal
        - Maintain 3 pointers `low=0`, `mid=0`, `high=len(nums)-1`
        - Note that the array is sorted between `0 to low-1`, `low to mid-1`, and `high to len(nums)-1` and unsorted between `mid to high-1`
        - `while mid ≤ high` we check `nums[mid]` and swap it with low/high accordingly and maintain pointers as necessary



- One-Pass
- Time Complexity: $O(n)$
- Space Complexity: $O(1)$

```
# Python3
# Brute-force Solution
class Solution:
    def sortColors(self, nums: List[int]) -> None:
        nums.sort()
```

```
# Python3
# Better Solution (Two Pass)
class Solution:
    def sortColors(self, nums: List[int]) -> None:
        zero = 0
        one = 0
        two = 0
        for i in nums:
            if i == 0:
                zero += 1
            elif i == 1:
                one += 1
            else:
                two += 1
        for i in range(zero):
            nums[i] = 0
        for i in range(zero, one+zero):
            nums[i] = 1
        for i in range(one+zero, two+one+zero):
            nums[i] = 2
```

```
# Python3
# Optimal Solution (Dutch National Flag Algorithm - One Pass)
class Solution:
    def sortColors(self, nums: List[int]) -> None:
        low = 0
        mid = 0
        high = len(nums) - 1

        while mid <= high:
            if nums[mid] == 0:
                nums[low], nums[mid] = nums[mid], nums[low]
                low += 1
```

```
                mid += 1
            elif nums[mid] == 1:
                mid += 1
            else:
                nums[mid], nums[high] = nums[high], nums[mid]
                high -= 1
```

```
// C++
// Optimal Solution (Dutch National Flag Algorithm)
#include <bits/stdc++.h>
class Solution {
public:
    void sortColors(vector<int>& nums) {
        int low = 0;
        int mid = 0;
        int high = nums.size() - 1;

        while (mid <= high) {
            if (nums[mid] == 0) {
                swap(nums[low], nums[mid]);
                low++;
                mid++;
            }
            else if (nums[mid] == 1) {
                mid++;
            }
            else {
                swap(nums[mid], nums[high]);
                high--;
            }
        }
    }
};
```

Best Time to Buy and Sell Stock

- Approach
  - Brute-force
    - Use 2 loops and track every transaction and maintain a variable to contain the max value among all transactions.
    - Time Complexity: $O(n^2)$
    - Space Complexity: $O(1)$
  - Optimal
    - Iterate over the array and check whether `cur_price - min_price > max_profit` and if `cur_price < min_price`
    - Time Complexity: $O(n)$
    - Space Complexity: $O(1)$

```
# Python3
# Brute-force Solution
class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        max_profit = 0
        for i in range(len(prices)):
            for j in range(i+1, len(prices)):
```

```
                max_profit = max(max_profit, prices[j]-prices[i])
        return max_profit
```

```
# Python3
# Optimal Solution
class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        min_price = prices[0]
        max_profit = 0
        for i in range(1, len(prices)):
            # if current_price - min_price is greater than max_profit, update max_profit
            max_profit = max(max_profit, prices[i]-min_price)

            # if current_price is less than min_price, update min_price
            min_price = min(min_price, prices[i])
        return max_profit
```

```
// C++
// Optimal Solution
int maxProfit(vector<int> &arr) {
    int maxPro = 0;
    int n = arr.size();
    int minPrice = INT_MAX;

    for (int i = 0; i < arr.size(); i++) {
        minPrice = min(minPrice, arr[i]);
        maxPro = max(maxPro, arr[i] - minPrice);
    }

    return maxPro;
```

Using Kadane's Algorithm

- Will also work when the prices array is given as price_difference array i.e. `{1, 7, 4, 11}` will be given as `{0, 6, -3, 7}`

```
# Python3
# Optimal Solution (variation of Kadane's Algorithm)
class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        n = len(prices)
        ans = 0
        curSum = 0
        for i in range(n-1):
            curSum += prices[i+1] - prices[i]
            if curSum < 0:
                curSum = 0
            ans = max(ans, curSum)
        return ans
```

Merge 2 Sorted Arrays

- Approach
  - Brute-force
    - Concatenate both arrays and apply any (in-built) sorting algorithm
    - Time Complexity: $O((m+n)log(m+n))$

        - Space Complexity: $O(1)$ (in-place)
    - Better
        - Maintain a separate array `res` and two pointers `a` and `b` (for given arrays).
        - `while (a < m and b < n)` check `if nums1[a] < nums2[b]`
        - Time Complexity: $O(m + n)$
        - Space Complexity: $O(m + n)$
    - Optimal
        - Maintain three pointers and iterate from backwards of both arrays and take the larger element between the two and store it at the back of `nums1` array
        - Time Complexity: $O(m + n)$
        - Space Complexity: $O(1)$

```
# Python3
# Better Solution
class Solution:
    def merge(self, nums1: List[int], m: int, nums2: List[int], n: int) -> None:
        a = b = 0
        res = []
        while (a < m and b < n):
            if nums1[a] < nums2[b]:
                res.append(nums1[a])
                a += 1
            else:
                res.append(nums2[b])
                b += 1
        while (a < m):
            res.append(nums1[a])
            a += 1
        while (b < n):
            res.append(nums2[b])
            b += 1
        nums1[:] = res.copy()
```

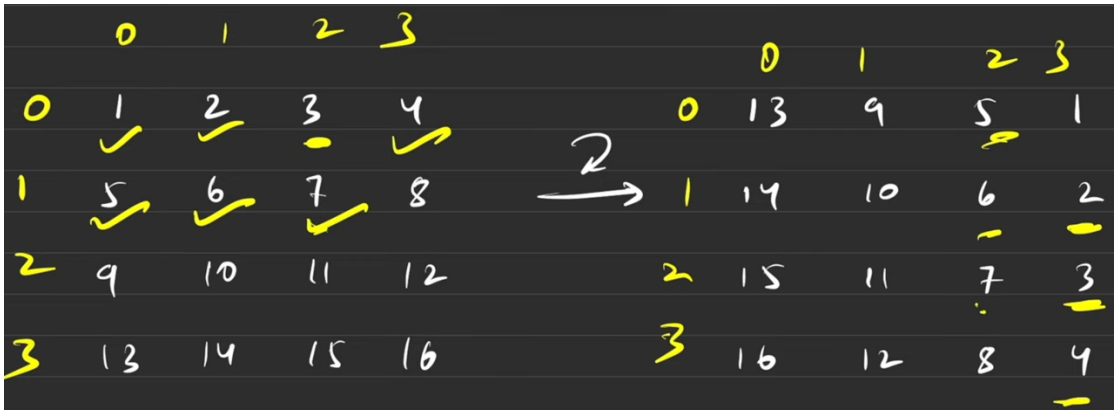
```
# Python3
# Optimal Solution
class Solution:
    def merge(self, nums1: List[int], m: int, nums2: List[int], n: int) -> None:
        a, b, write_index = m-1, n-1, m + n - 1

        while b >= 0:
            if a >= 0 and nums1[a] > nums2[b]:
                nums1[write_index] = nums1[a]
                a -= 1
            else:
                nums1[write_index] = nums2[b]
                b -= 1

            write_index -= 1
```

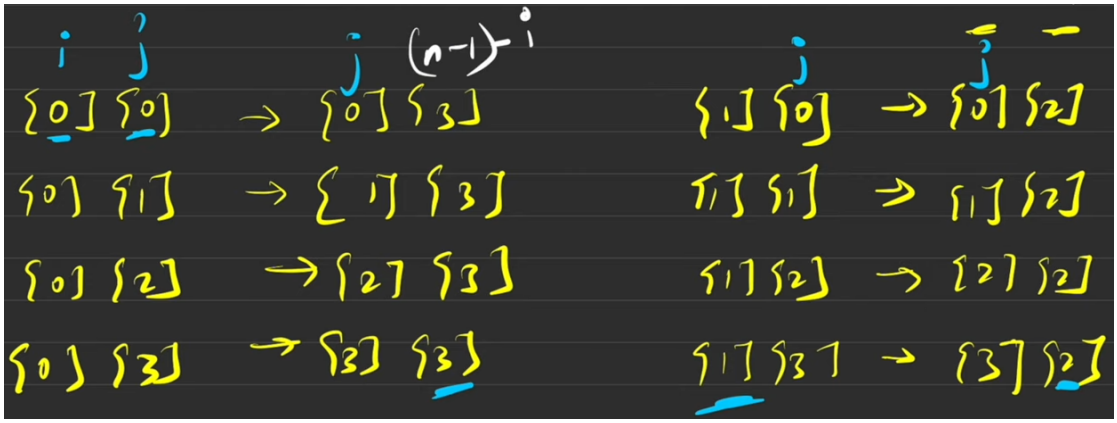
```
// C++
// Optimal Solution
class Solution {
public:
    void merge(vector<int>& nums1, int m, vector<int>& nums2, int n) {
```

```
        int a = m - 1;
        int b = n - 1;
        int c = m + n - 1;

        while (b >= 0) {
            if (a >= 0 && nums1[a] > nums2[b]) {
                nums1[c] = nums1[a];
                a--;
            }
            else {
                nums1[c] = nums2[b];
                b--;
            }
            c--;
        }
    }
};
```

Rotate the Matrix by 90 Degrees (Clockwise)

- Approach
  - Brute-force
    - Take an element in the matrix and put it in its proper place in another matrix.
    - Time Complexity: $O(n^2)$
    - Space Complexity: $O(n^2)$





  - Optimal
    - Transpose the matrix and reverse all the rows.

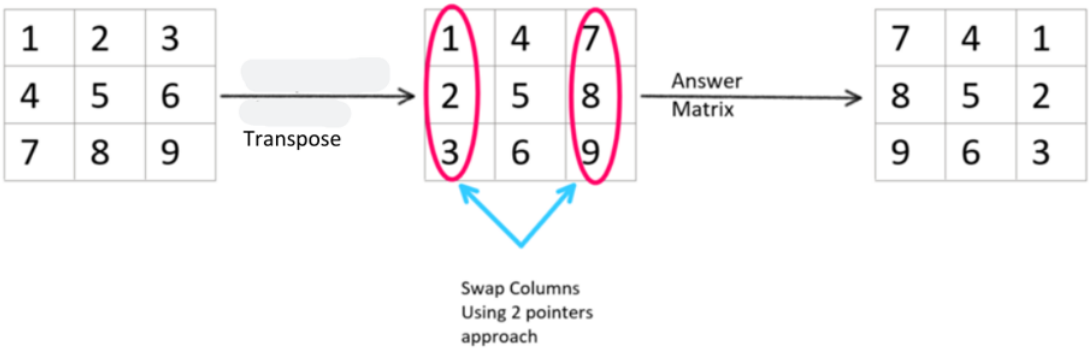
- Time Complexity: $O(n^2)$
- Space Complexity: $O(1)$ (in-place)

```
# Python3
# Brute-force Solution
class Solution:
    def rotate(self, matrix: List[List[int]]) -> None:
        n = len(matrix)
        ans = [[0 for i in range(n)]for j in range(n)]

        for i in range(n):
            for j in range(n):
                ans[j][n-i-1] = matrix[i][j]

        for i in range(n):
            for j in range(n):
                matrix[i][j] = ans[i][j]
```

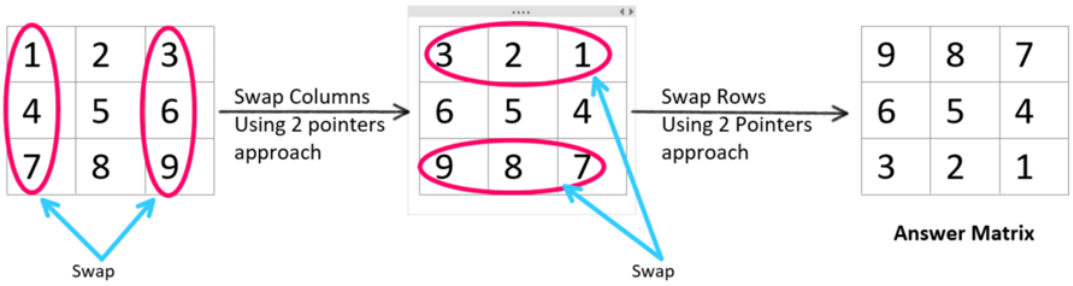
```
# Python3
# Optimal Solution
class Solution:
    def rotate(self, matrix: List[List[int]]) -> None:
        n = len(matrix)
        for i in range(1, n):
            for j in range(i):
                matrix[i][j], matrix[j][i] = matrix[j][i], matrix[i][j]

        for i in matrix:
            i.reverse()
```

```
// C++
// Optimal Solution
void rotate(vector<vector<int>>& m) {
    int n = m.size();

    for(int i=0; i<n; i++)
        for(int j=0; j<i; j++)
            swap(m[i][j], m[j][i]);

    for(int i=0; i<n; i++)
        reverse(m[i].begin(), m[i].end());
}
```

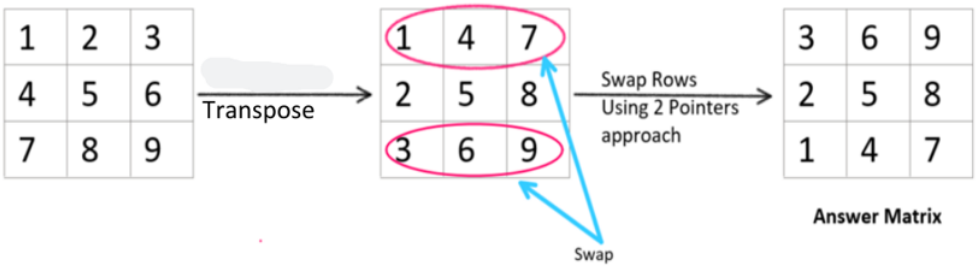
Rotate 90°



Rotate 180°



Rotate 270°



Merge Overlapping Sub-intervals

- Approach
  - Brute-force

- Sort the array first and then check for every interval[i], what all intervals[i+1] to intervals[n-1] could be merged with the current interval.
- Time Complexity: $O(nlogn) + O(2n)$
- Space Complexity: $O(n)$

- Optimal
  - Sort the array
  - Use stack to check if `interval[i] ≤ stack.top()[1]` then update the stack top with `max(stack.top()[1], interval[i][1])` else push the interval[i] onto the stack
  - Time Complexity: $O(nlogn) + O(n)$
  - Space Complexity: $O(n)$

```
# Python3
# Brute-force Solution
def mergeOverlappingIntervals(arr: List[List[int]]) -> List[List[int]]:
    n = len(arr) # size of the array

    # sort the given intervals:
    arr.sort()

    ans = []

    for i in range(n): # select an interval:
        start, end = arr[i][0], arr[i][1]

        # Skip all the merged intervals:
        if ans and end <= ans[-1][1]:
            continue

        # check the rest of the intervals:
        for j in range(i + 1, n):
            if arr[j][0] <= end:
                end = max(end, arr[j][1])
            else:
                break
        ans.append([start, end])
    return ans
```

```
# Python3
# Optimal Solution
class Solution:
    def merge(self, intervals: List[List[int]]) -> List[List[int]]:
        intervals.sort()
        ans = [intervals[0]]
        for i in range(1, len(intervals)):
            if intervals[i][0] > ans[-1][1]:
                ans += intervals[i],
            else:
                ans[-1][1] = max(ans[-1][1], intervals[i][1])
        return ans
```

```
// C++
// Optimal Solution
vector<vector<int>> mergeOverlappingIntervals(vector<vector<int>> &arr) {
    int n = arr.size(); // size of the array

    //sort the given intervals:
```

```
    sort(arr.begin(), arr.end());

    vector<vector<int>> ans;

    for (int i = 0; i < n; i++) {
        // if the current interval does not
        // lie in the last interval:
        if (ans.empty() || arr[i][0] > ans.back()[1]) {
            ans.push_back(arr[i]);
        }
        // if the current interval
        // lies in the last interval:
        else {
            ans.back()[1] = max(ans.back()[1], arr[i][1]);
        }
    }
    return ans;
}
```

Template

- Approach
  - Brute-force
    -
    - Time Complexity: $O(n^3)$
    - Space Complexity: $O(1)$
  - Better
    -
    - Time Complexity: $O(n^3)$
    - Space Complexity: $O(1)$
  - Optimal
    -
    - Time Complexity: $O(n^3)$
    - Space Complexity: $O(1)$

```
# Python3
# Brute-force Solution
```

```
# Python3
# Better Solution
```

```
# Python3
# Optimal Solution
```

```
// C++
// Optimal Solution
```