## **Introduction**

Array is one of the fundamental blocks in data structure. Since a string is just formed by an array of characters, they are both similar. Most interview questions fall into this category.

In this card, we will introduce array and string. After finishing this card, you should:

1. Understand the differences between array and dynamic array;
2. Be familiar with basic operations in the array and dynamic array;
3. Understand multidimensional arrays and be able to use a two-dimensional array;
4. Understand the concept of string and the different features string has;
5. Be able to apply the two-pointer technique to practical problems.

Introduction to Array

 Introduction to Array

 Introduction to Dynamic Array

 Find Pivot Index

 Largest Number At Least Twice of Others

 Plus One

Introduction to 2D Array

 Introduction to 2D Array

 Diagonal Traverse

 Spiral Matrix

 Pascal's Triangle

Introduction to String

 Introduction to String

 Immutable String - Problems & Solutions

 Add Binary

 Implement strStr()

 Longest Common Prefix

Two-Pointer Technique

 Two-pointer Technique - Scenario I

 Reverse String

 Array Partition I

 Two Sum II - Input array is sorted

 Two-pointer Technique - Scenario II

 Remove Element

 Max Consecutive Ones

 Minimum Size Subarray Sum

Conclusion

 Array-related Techniques

 Rotate Array

 Pascal's Triangle II

 Reverse Words in a String

 Reverse Words in a String III

 Remove Duplicates from Sorted Array

**Introduction to Array**

In this chapter, we are going to introduce two important concepts: array and dynamic array.

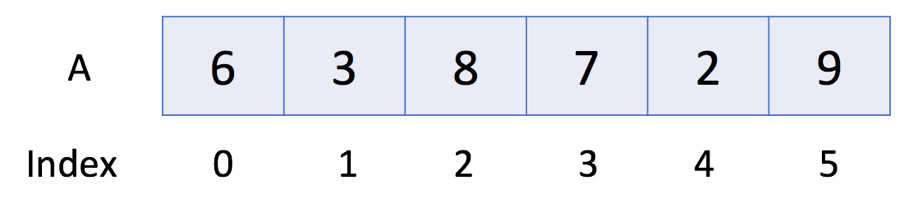
These are the basic data structure you should be familiar with. We also provide a tutorial for you to use the built-in array and dynamic array.

By completing this chapter, you should be able to answer the following questions:

1. What is the difference between array and dynamic array?
2. What is the corresponding built-in data structure of array and dynamic array in your frequently-used language?
3. How to perform basic operations (initialization, data access, modification, iteration, sort, etc) in an array?
4. How to perform basic operations (initialization, data access, modification, iteration, sort, addition, deletion, etc) in a dynamic array?

An array is a basic data structure to store a collection of elements sequentially. But elements can be accessed randomly since each element in the array can be identified by an array index.

An array can have one or more dimensions. Here we start with the one-dimensional array, which is also called the linear array. Here is an example:



In the above example, there are 6 elements in array A. That is to say, the length of A is 6. We can use A[0] to represent the first element in the array. Therefore, A[0] = 6. Similarly, A[1] = 3, A[2] = 8 and so on.

*Operations in Array*

Let's take a look at the usage of the array.

|  |
| --- |
| // "static void main" must be defined in a public class.  public class Main {  public static void main(String[] args) {  // 1. Initialize  int[] a0 = new int[5];  int[] a1 = {1, 2, 3};  // 2. Get Length  System.out.println("The size of a1 is: " + a1.length);  // 3. Access Element  System.out.println("The first element is: " + a1[0]);  // 4. Iterate all Elements  System.out.print("[Version 1] The contents of a1 are:");  for (int i = 0; i < a1.length; ++i) {  System.out.print(" " + a1[i]);  }  System.out.println();  System.out.print("[Version 2] The contents of a1 are:");  for (int item: a1) {  System.out.print(" " + item);  }  System.out.println();  // 5. Modify Element  a1[0] = 4;  // 6. Sort  Arrays.sort(a1);  }  } |

**Introduction to Dynamic Array**

As we mentioned in the previous article, an array has a fixed capacity and we need to specify the size of the array when we initialize it. Sometimes this will be somewhat inconvenient and wasteful.

Therefore, most programming languages offer built-in dynamic array which is still a random access list data structure but with variable size. For example, we have vector in C++ and ArrayList in Java.

### *Operations in Dynamic Array*

|  |
| --- |
| // "static void main" must be defined in a public class.  public class Main {  public static void main(String[] args) {  // 1. initialize  List<Integer> v0 = new ArrayList<>();  List<Integer> v1; // v1 == null  // 2. cast an array to a vector  Integer[] a = {0, 1, 2, 3, 4};  v1 = new ArrayList<>(Arrays.asList(a));  // 3. make a copy  List<Integer> v2 = v1; // another reference to v1  List<Integer> v3 = new ArrayList<>(v1); // make an actual copy of v1  // 3. get length  System.out.println("The size of v1 is: " + v1.size());  // 4. access element  System.out.println("The first element in v1 is: " + v1.get(0));  // 5. iterate the vector  System.out.print("[Version 1] The contents of v1 are:");  for (int i = 0; i < v1.size(); ++i) {  System.out.print(" " + v1.get(i));  }  System.out.println();  System.out.print("[Version 2] The contents of v1 are:");  for (int item : v1) {  System.out.print(" " + item);  }  System.out.println();  // 6. modify element  v2.set(0, 5); // modify v2 will actually modify v1  System.out.println("The first element in v1 is: " + v1.get(0));  v3.set(0, -1);  System.out.println("The first element in v1 is: " + v1.get(0));  // 7. sort  Collections.sort(v1);  // 8. add new element at the end of the vector  v1.add(-1);  v1.add(1, 6);  // 9. delete the last element  v1.remove(v1.size() - 1);  }  } |

**Find Pivot Index Solution**

Given an array of integers nums, write a method that returns the "pivot" index of this array.

We define the pivot index as the index where the sum of all the numbers to the left of the index is equal to the sum of all the numbers to the right of the index.

If no such index exists, we should return -1. If there are multiple pivot indexes, you should return the left-most pivot index.

**Example 1:**

**Input:** nums = [1,7,3,6,5,6]

**Output:** 3

**Explanation:**

The sum of the numbers to the left of index 3 (nums[3] = 6) is equal to the sum of numbers to the right of index 3.

Also, 3 is the first index where this occurs.

**Example 2:**

**Input:** nums = [1,2,3]

**Output:** -1

**Explanation:**

There is no index that satisfies the conditions in the problem statement.

**Constraints:**

* The length of nums will be in the range [0, 10000].
* Each element nums[i] will be an integer in the range [-1000, 1000].

 Hide Hint #1

We can precompute prefix sums P[i] = nums[0] + nums[1] + ... + nums[i-1]. Then for each index, the left sum is P[i], and the right sum is P[P.length - 1] - P[i] - nums[i].

#### **Approach #1: Prefix Sum [Accepted]**

**Intuition and Algorithm**

We need to quickly compute the sum of values to the left and the right of every index.

Let's say we knew S as the sum of the numbers, and we are at index i. If we knew the sum of numbers leftsum that are to the left of index i, then the other sum to the right of the index would just be S - nums[i] - leftsum.

As such, we only need to know about leftsum to check whether an index is a pivot index in constant time. Let's do that: as we iterate through candidate indexes i, we will maintain the correct value of leftsum.

|  |
| --- |
| class Solution {  public int pivotIndex(int[] nums) {  int sum = 0, leftsum = 0;  for (int x: nums) sum += x;  for (int i = 0; i < nums.length; ++i) {  if (leftsum == sum - leftsum - nums[i]) return i;  leftsum += nums[i];  }  return -1;  }  } |

**Largest Number At Least Twice of Others**

In a given integer array nums, there is always exactly one largest element.

Find whether the largest element in the array is at least twice as much as every other number in the array.

If it is, return the **index** of the largest element, otherwise return -1.

**Example 1:**

**Input:** nums = [3, 6, 1, 0]

**Output:** 1

**Explanation:** 6 is the largest integer, and for every other number in the array x,

6 is more than twice as big as x. The index of value 6 is 1, so we return 1.

**Example 2:**

**Input:** nums = [1, 2, 3, 4]

**Output:** -1

**Explanation:** 4 isn't at least as big as twice the value of 3, so we return -1.

**Note:**

1. nums will have a length in the range [1, 50].
2. Every nums[i] will be an integer in the range [0, 99].

Hint #1

Scan through the array to find the unique largest element `m`, keeping track of it's index `maxIndex`. Scan through the array again. If we find some `x != m` with `m < 2\*x`, we should return `-1`. Otherwise, we should return `maxIndex`.

#### **Approach #1: Linear Scan [Accepted]**

**Intuition and Algorithm**

Scan through the array to find the unique largest element m, keeping track of it's index maxIndex.

Scan through the array again. If we find some x != m with m < 2\*x, we should return -1.

Otherwise, we should return maxIndex.

|  |
| --- |
| class Solution {  public int dominantIndex(int[] nums) {  int maxIndex = 0;  for (int i = 0; i < nums.length; ++i) {  if (nums[i] > nums[maxIndex])  maxIndex = i;  }  for (int i = 0; i < nums.length; ++i) {  if (maxIndex != i && nums[maxIndex] < 2 \* nums[i])  return -1;  }  return maxIndex;  }  } |

**Plus One**

Given a **non-empty** array of decimal digits representing a non-negative integer, increment one to the integer.

The digits are stored such that the most significant digit is at the head of the list, and each element in the array contains a single digit.

You may assume the integer does not contain any leading zero, except the number 0 itself.

**Example 1:**

**Input:** digits = [1,2,3]

**Output:** [1,2,4]

**Explanation:** The array represents the integer 123.

**Example 2:**

**Input:** digits = [4,3,2,1]

**Output:** [4,3,2,2]

**Explanation:** The array represents the integer 4321.

**Example 3:**

**Input:** digits = [0]

**Output:** [1]

**Constraints:**

* 1 <= digits.length <= 100
* 0 <= digits[i] <= 9

## **Solution**

#### **Overview**

"Plus One" is a subset of the problem set "Add Number", which shares the same solution pattern.

All these problems could be solved in linear time, and the question here is how to solve it without using the addition operation or how to solve it in constant space complexity.

The choice of algorithm should be based on the format of input. Here we list a few examples:

1. Integers

Usually the addition operation is not allowed for such a case. Use Bit Manipulation Approach. Here is an example: [Add Binary](https://leetcode.com/articles/add-binary/).

1. Strings

Use bit by bit computation. Note, sometimes it might not be feasible to come up a solution with the constant space for languages with immutable strings, e.g. for Java and Python. Here is an example: [Add Binary](https://leetcode.com/articles/add-binary/).

1. Linked Lists

Sentinel Head + Schoolbook Addition with Carry. Here is an example: [Plus One Linked List](https://leetcode.com/articles/plus-one-linked-list/).

1. Arrays (also the current problem)

Schoolbook addition with carry.

Note that, a straightforward idea to convert everything into integers and then apply the addition could be risky, especially for the implementation in Java, due to the potential integer overflow issue.

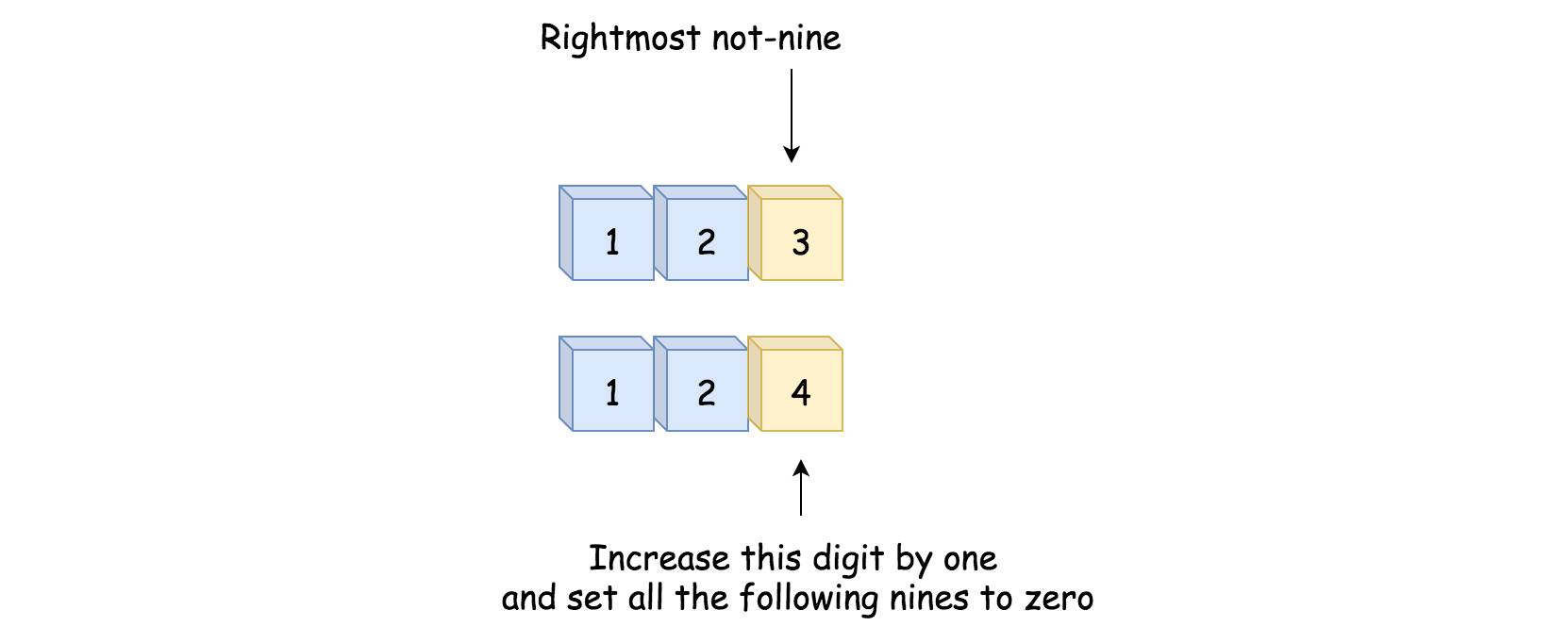
As one can imagine, once the array gets long, the result of conversion cannot fit into the data type of Integer, or Long, or even [BigInteger](https://docs.oracle.com/javase/8/docs/api/java/math/BigInteger.html).

#### **Approach 1: Schoolbook Addition with Carry**

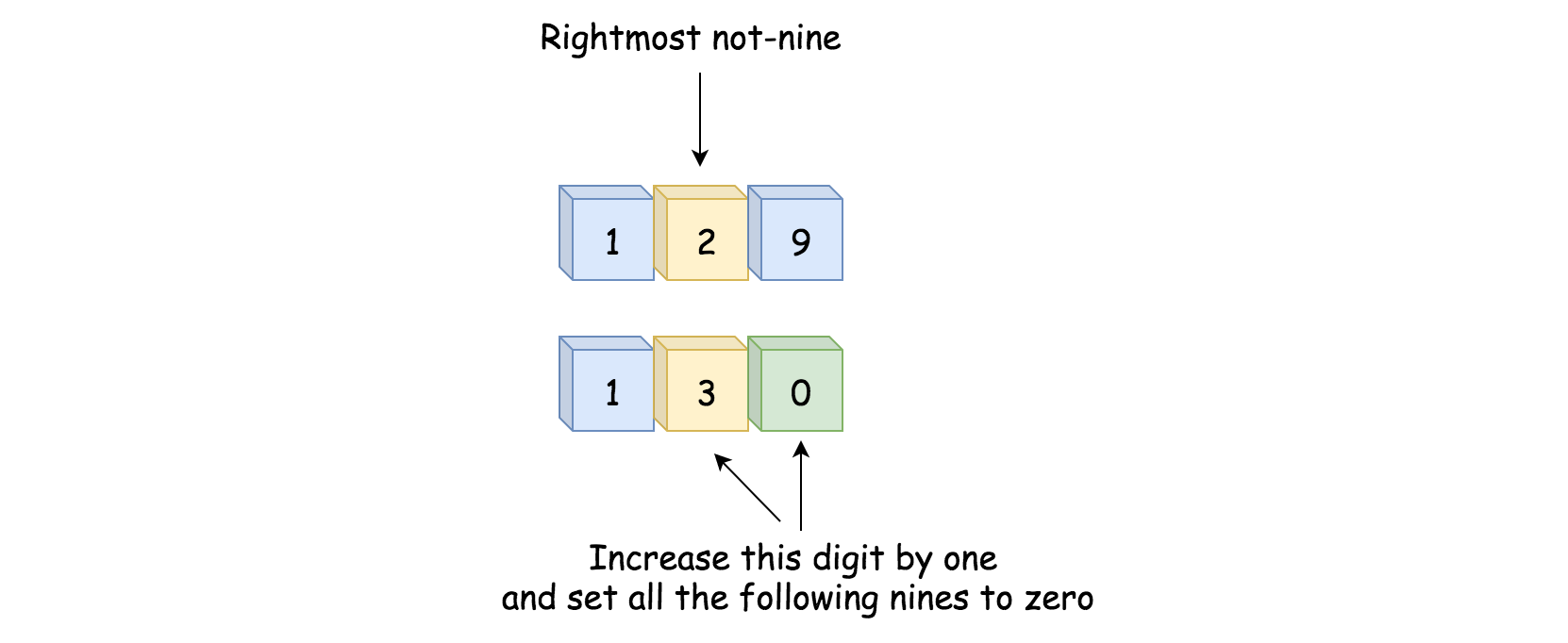
**Intuition**

Let us identify the rightmost digit which is not equal to nine and increase that digit by one. All the following consecutive digits of nine should be set to zero.

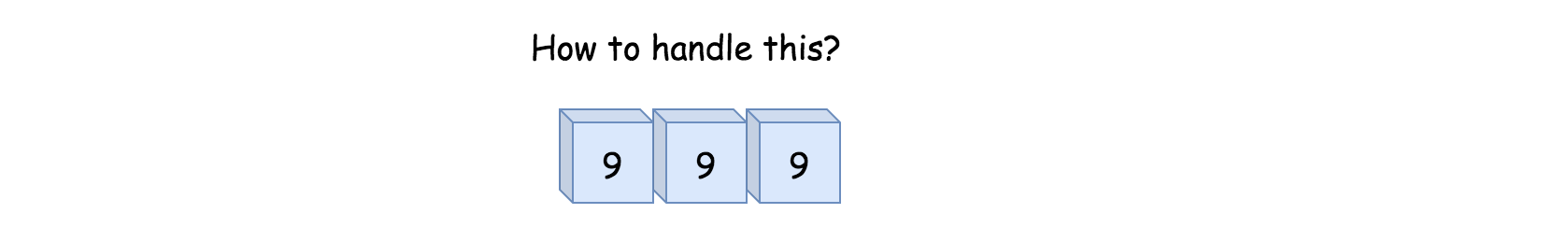
Here is the simplest use case which works fine.



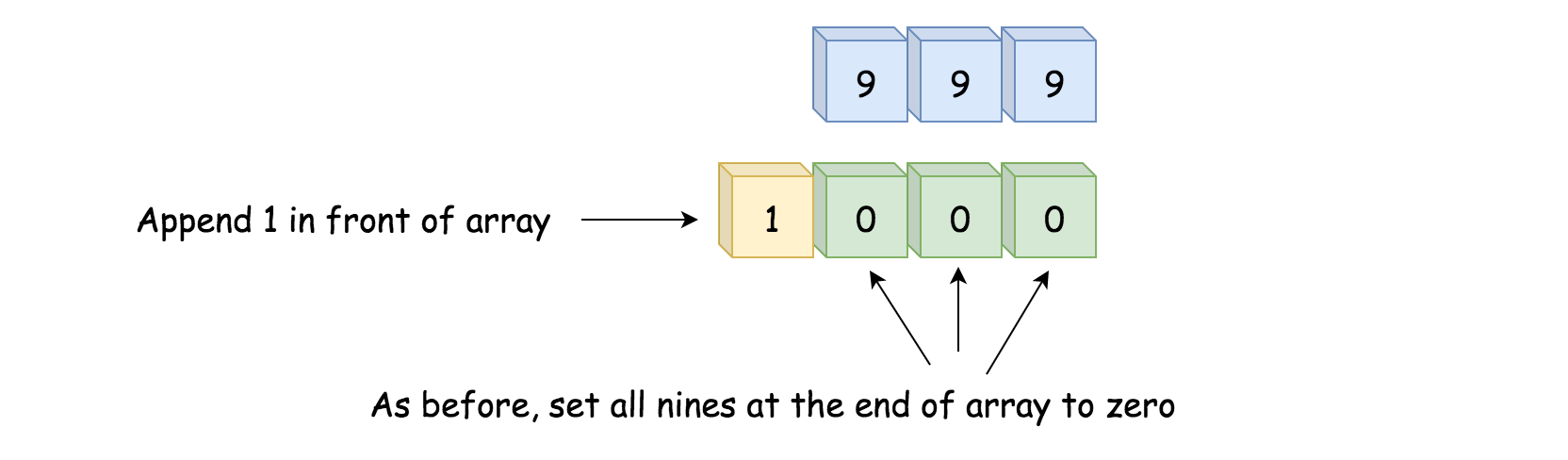
Here is a slightly complicated case which still passes.



And here is the case which breaks everything, because all the digits are nines.



In this case, we need to set all nines to zero and append 1 to the left side of the array.

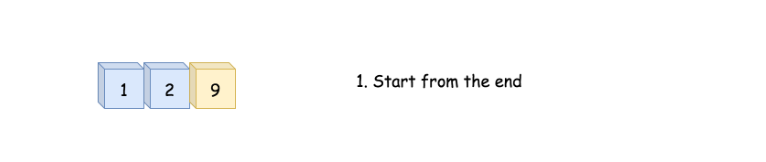


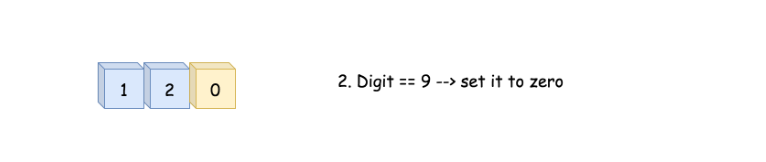
**Algorithm**

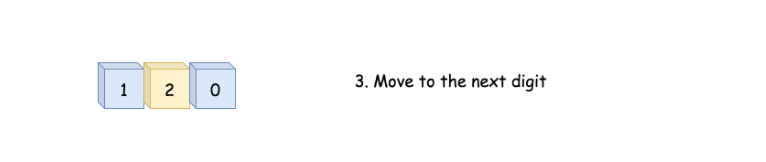
* Move along the input array starting from the end of array.
* Set all the nines at the end of array to zero.
* If we meet a not-nine digit, we would increase it by one. The job is done - return digits.
* We're here because **all** the digits were equal to nine. Now they have all been set to zero. We then append the digit 1 in front of the other digits and return the result.

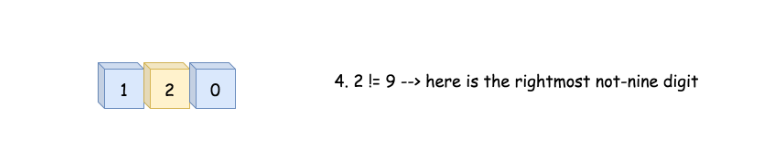
**Implementation**

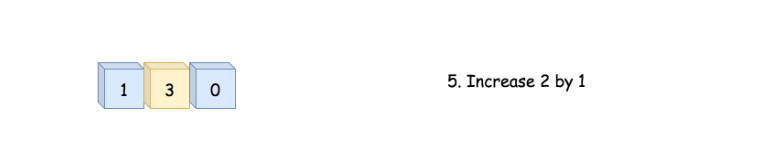


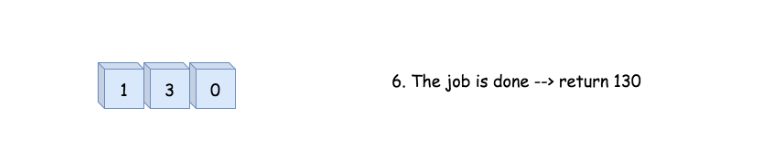












|  |
| --- |
| class Solution {  public int[] plusOne(int[] digits) {  int n = digits.length;  // move along the input array starting from the end  for (int idx = n - 1; idx >= 0; --idx) {  // set all the nines at the end of array to zeros  if (digits[idx] == 9) {  digits[idx] = 0;  }  // here we have the rightmost not-nine  else {  // increase this rightmost not-nine by 1  digits[idx]++;  // and the job is done  return digits;  }  }  // we're here because all the digits are nines  digits = new int[n + 1];  digits[0] = 1;  return digits;  }  } |

**Complexity Analysis**

Let *N* be the number of elements in the input list.

* Time complexity: O(*N*) since it's not more than one pass along the input list.
* Space complexity: O(*N*)
  + Although we perform the operation **in-place** (*i.e.* on the input list itself), in the worst scenario, we would need to allocate an intermediate space to hold the result, which contains the *N*+1 elements. Hence the overall space complexity of the algorithm is O(*N*).

**Introduction to 2D Array**

In the previous chapter, we have learned about the one-dimensional array. However, sometimes, we might need a multidimensional array which is more suitable for more complex structures like table or matrix.

In this chapter, we are going to focus on the two-dimensional array to explain:

* How does a multidimensional array work?
* How to use a two-dimensional array to solve problems?

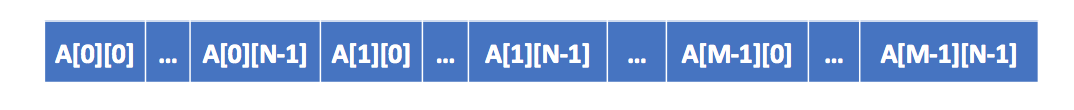
Similar to a one-dimensional array, a two-dimensional array also consists of a sequence of elements. But the elements can be laid out in a rectangular grid rather than a line.

|  |
| --- |
| // "static void main" must be defined in a public class.  public class Main {  private static void printArray(int[][] a) {  for (int i = 0; i < a.length; ++i) {  System.out.println(a[i]);  }  for (int i = 0; i < a.length; ++i) {  for (int j = 0; a[i] != null && j < a[i].length; ++j) {  System.out.print(a[i][j] + " ");  }  System.out.println();  }  }  public static void main(String[] args) {  System.out.println("Example I:");  int[][] a = new int[2][5];  printArray(a);  System.out.println("Example II:");  int[][] b = new int[2][];  printArray(b);  System.out.println("Example III:");  b[0] = new int[3];  b[1] = new int[5];  printArray(b);  }  } |

### *Principle*

In some languages, the multidimensional array is actually implemented internally as a one-dimensional array while in some other languages, there is actually no multidimensional array at all.

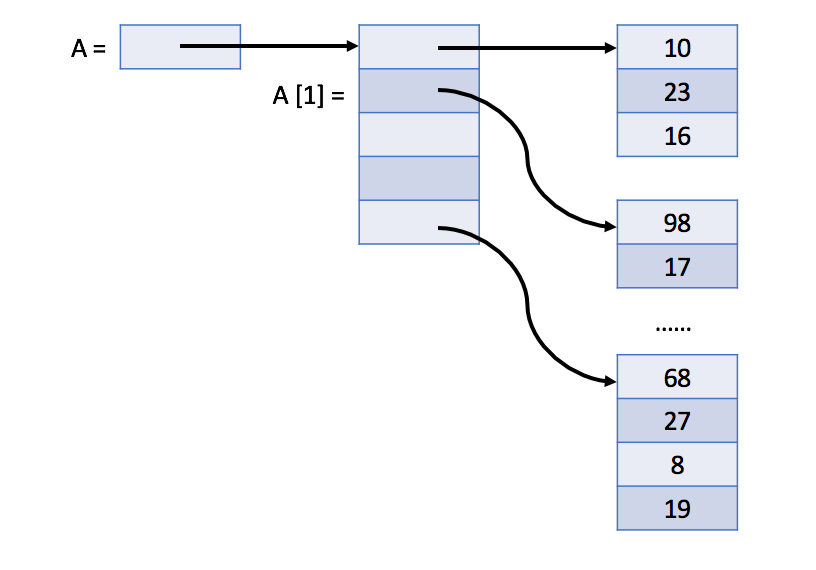
**1. C++ stores the two-dimensional array as a one-dimensional array.**

The picture below shows the actual structure of a *M \* N* array *A*:

So actually *A[i][j]* equals to *A[i \* N + j]* if we defined *A* as a one-dimensional array which also contains *M \* N* elements.

**2. In Java, the two-dimensional array is actually a one-dimensional array which contains *M* elements, each of which is an array of *N* integers.**

The picture below shows the actual structure of a two-dimensional array *A* in Java:



*Dynamic 2D Array*

Similar to the one-dimensional dynamic array, we can also define a dynamic two-dimensional array. Actually, it can be just a nested dynamic array. You can try it out by yourself.

**Diagonal Traverse**

Given a matrix of M x N elements (M rows, N columns), return all elements of the matrix in diagonal order as shown in the below image.

**Example:**

**Input:**

[

[ 1, 2, 3 ],

[ 4, 5, 6 ],

[ 7, 8, 9 ]

]

**Output:** [1,2,4,7,5,3,6,8,9]

**Explanation:**



**Note:**

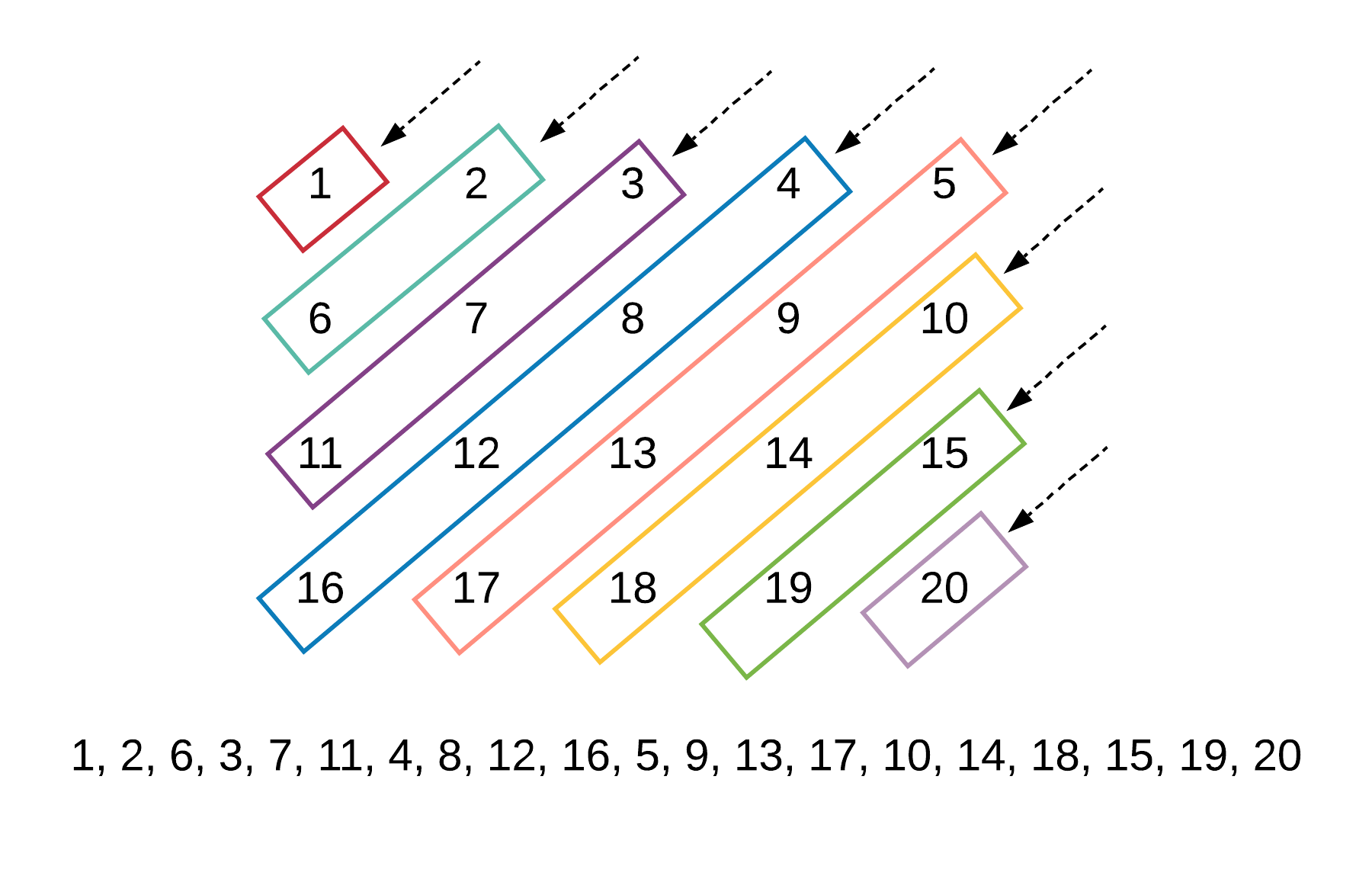
The total number of elements of the given matrix will not exceed 10,000.

## **Solution**

#### **Approach 1: Diagonal Iteration and Reversal**

**Intuition**

A common strategy for solving a lot of programming problem is to first solve a stripped down, simpler version of them and then think what needs to be changed to achieve the original goal. Our first approach to this problem is also based on this very idea. So, instead of thinking about the zig-zag pattern of printing for the diagonals, let's say the problem statement simply asked us to print out the contents of the matrix, one diagonal after the other starting from the first element. Let's see what this problem would look like.



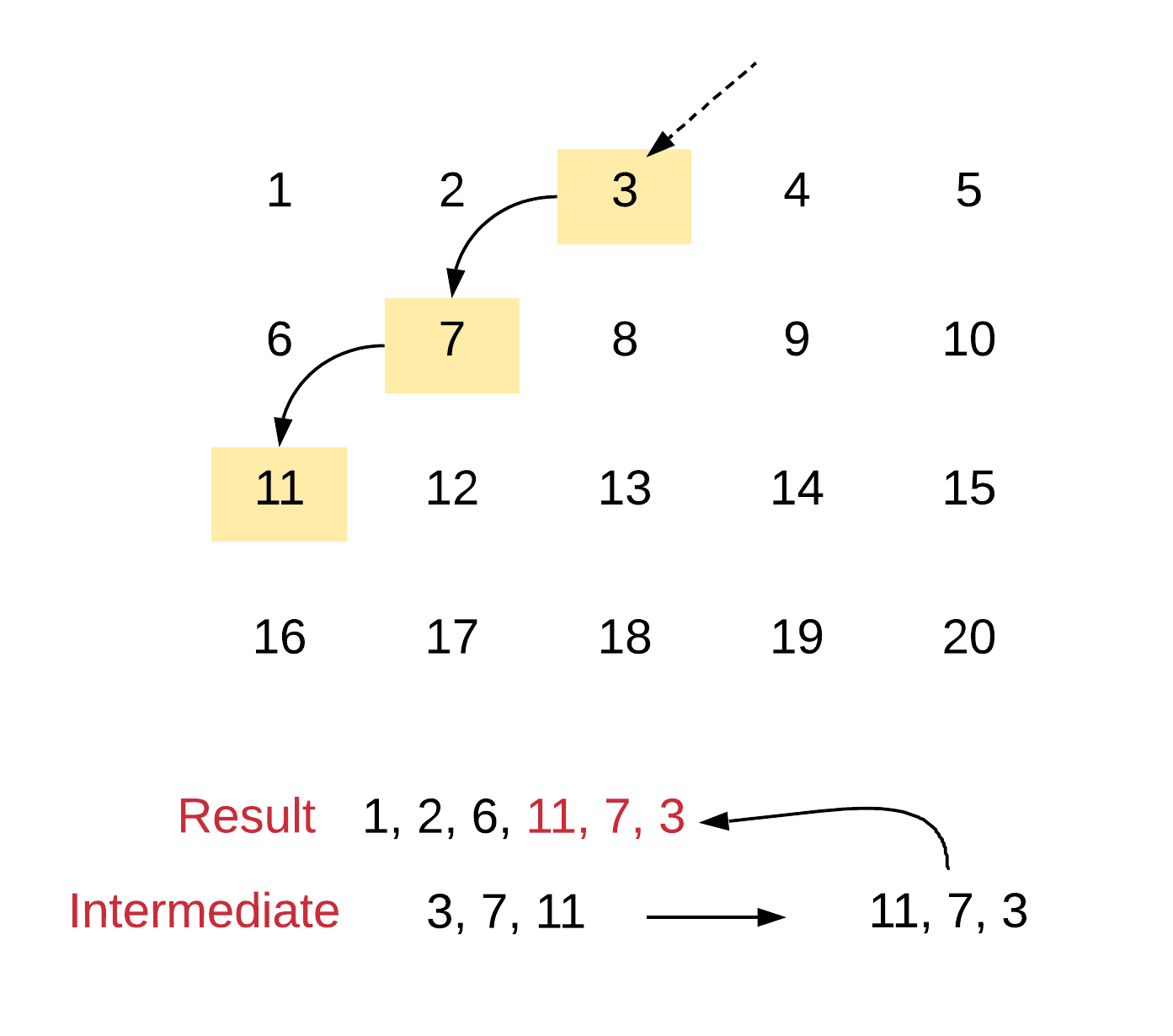
The first row and the last column in this problem would serve as the starting point for the corresponding diagonal. Given an element inside a diagonal, say [*i*,*j*], we can either go up the diagonal by going one row up and one column ahead i.e. [*i*−1,*j*+1] or, we can go down the diagonal by going one row down and one column to the left i.e. [*i*+1,*j*−1]. Note that this applies to diagonals that go from right to left only. The math would change for the ones that go from left to right.

This is a simple problem to solve, right? The only difference between this one and the original problem is that some of the diagonals are not printed in the right order. That's all we need to fix to get the right solution!

We simply need to reverse the odd numbered diagonals before we add the elements to the final result array. So, for e.g. the third diagonal starting from the left would be [3, 7, 11] and before we add these elements to the final result array, we simply reverse them i.e. [11, 7, 3].

**Algorithm**

1. Initialize a result array that we will eventually return.
2. We would have an outer loop that will go over each of the diagonals one by one. As mentioned before, the elements in the first row and the last column would actually be the heads of their corresponding diagonals.
3. We then have an inner while loop that iterates over all the elements in the diagonal. We can calculate the number of elements in the corresponding diagonal by doing some math but we can simply iterate until one of the indices goes out of bounds.
4. For each diagonal we will need a new list or dynamic array like data structure since we don't know what size to allocate. Again, we can do some math and calculate the size of that particular diagonal and allocate memory; but it's not necessary for this explanation.
5. For odd numbered diagonals, we simply need to add the elements in our intermediary array, in reverse order to the final result array.



|  |
| --- |
| class Solution {  public int[] findDiagonalOrder(int[][] matrix) {  // Check for empty matrices  if (matrix == null || matrix.length == 0) {  return new int[0];  }  // Variables to track the size of the matrix  int N = matrix.length;  int M = matrix[0].length;  // The two arrays as explained in the algorithm  int[] result = new int[N\*M];  int k = 0;  ArrayList<Integer> intermediate = new ArrayList<Integer>();    // We have to go over all the elements in the first  // row and the last column to cover all possible diagonals  for (int d = 0; d < N + M - 1; d++) {    // Clear the intermediate array every time we start  // to process another diagonal  intermediate.clear();    // We need to figure out the "head" of this diagonal  // The elements in the first row and the last column  // are the respective heads.  int r = d < M ? 0 : d - M + 1;  int c = d < M ? d : M - 1;    // Iterate until one of the indices goes out of scope  // Take note of the index math to go down the diagonal  while (r < N && c > -1) {    intermediate.add(matrix[r][c]);  ++r;  --c;  }    // Reverse even numbered diagonals. The  // article says we have to reverse odd  // numbered articles but here, the numbering  // is starting from 0 :P  if (d % 2 == 0) {  Collections.reverse(intermediate);  }    for (int i = 0; i < intermediate.size(); i++) {  result[k++] = intermediate.get(i);  }  }  return result;  }  } |

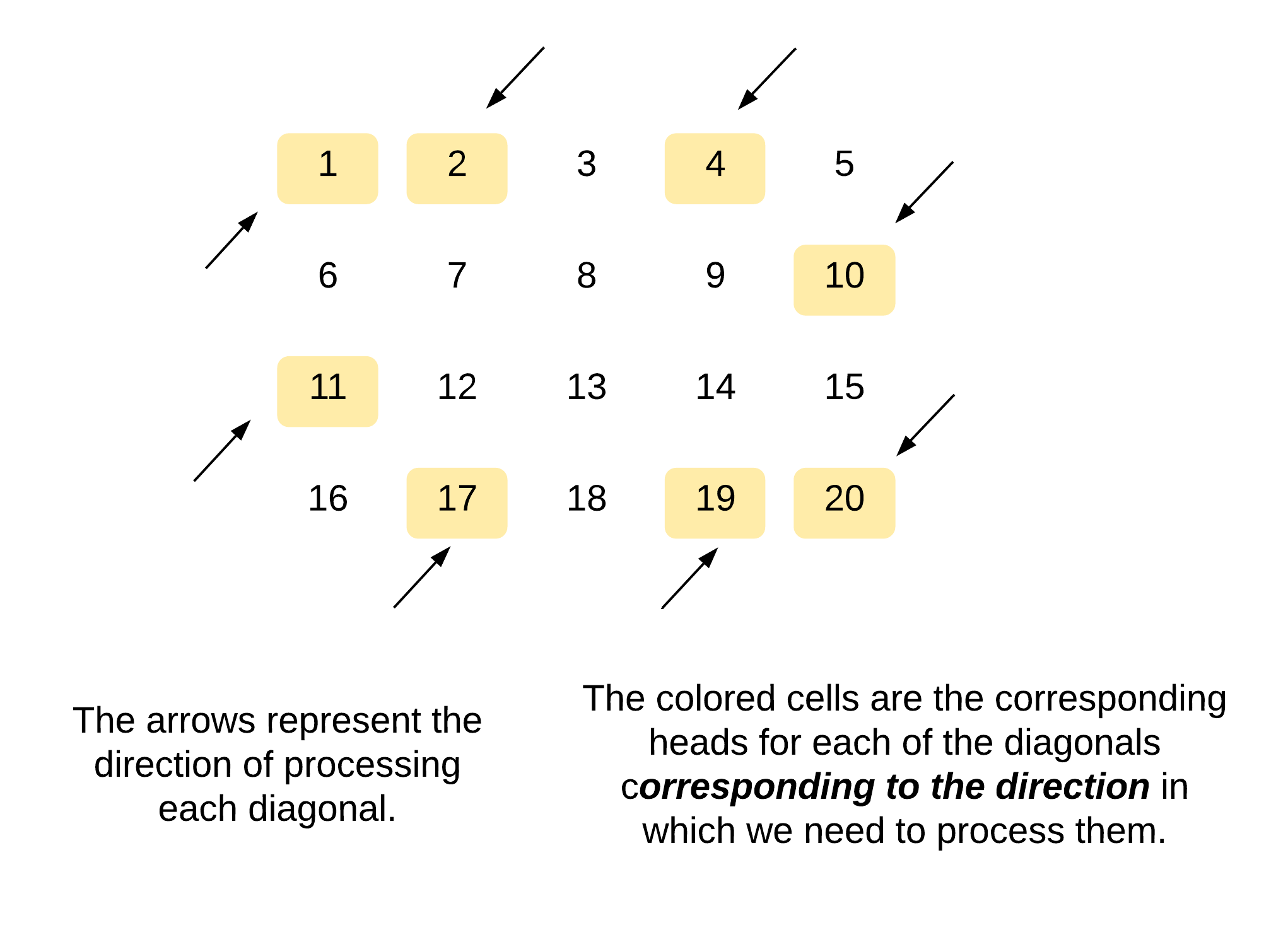
#### **Approach 2: Simulation**

**Intuition**

This approach simply and plainly does what the problem statement asks us to do. It's pure simulation. However, in order to implement this simulation, we need to understand the walking patterns inside the array. Basically, in the previous approach, figuring our the head of the diagonal was pretty easy. In this case, it won't be that easy. We need to figure out two things for each diagonal:

1. The direction in which we want to process it's elements and
2. The head or the starting point for the diagonal depending upon its direction.

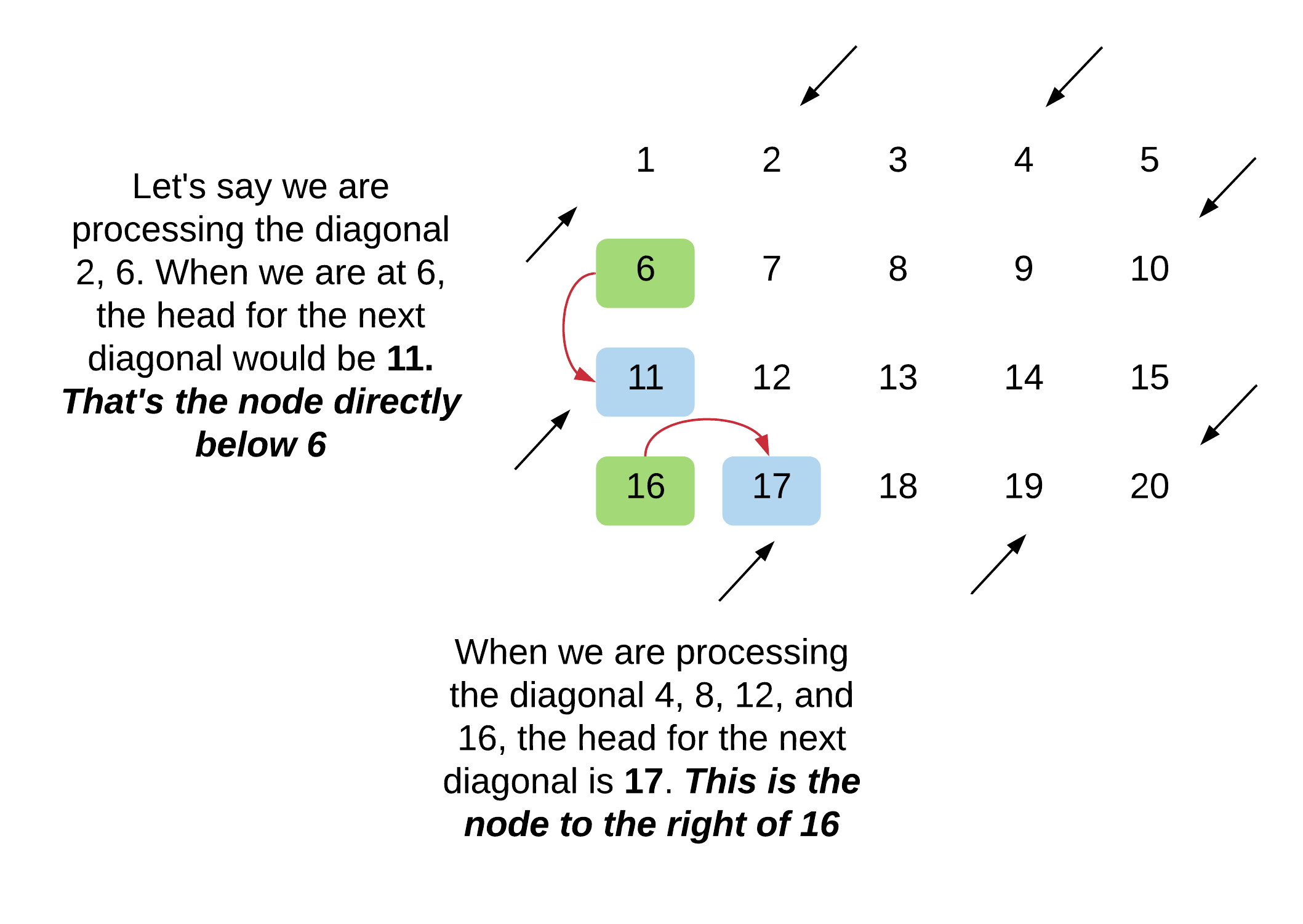
Let's see these two things annotated on a sample matrix.



Now that we know what two things we need to figure out, let's get to the part where we actually do it! The direction is pretty straightforward. We can simply use a boolean variable and keep alternating it to figure out the direction for a diagonal. That part is sorted. The slightly tricky part is figuring out the head of the next diagonal.

The good part is, we already know the end of the previous diagonal. We can use that information to figure out the head of the next diagonal.

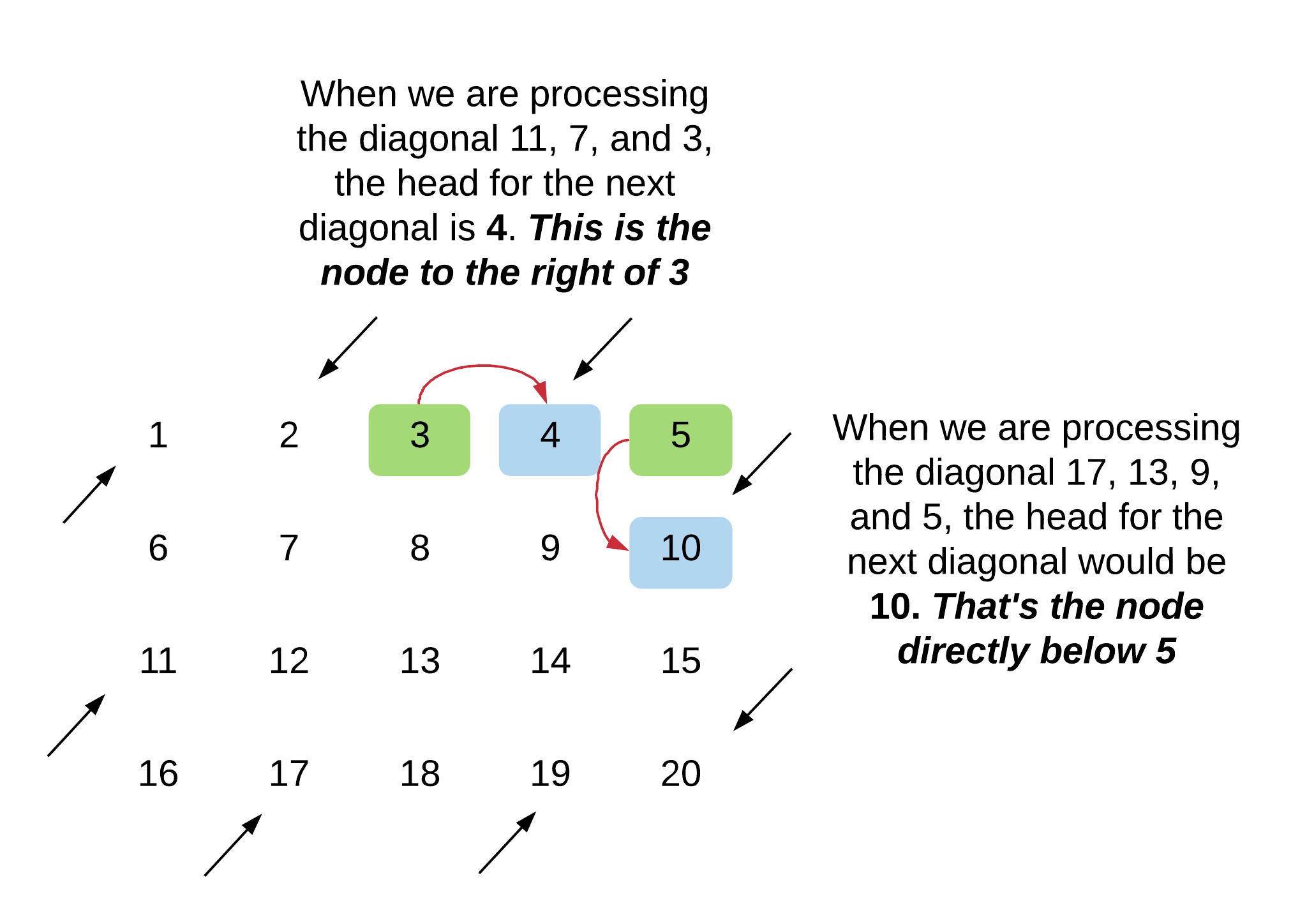
**Next head when going UP**  
Let's look at the two scenarios that we may come across when we are at the tail end of a downwards diagonal and we want to find the head of the next diagonal.



So, the general rule that we will be following when we want to find the head for an upwards going diagonal is that:

The head would be the node directly below the tail of the previous diagonal. Unless the tail lies in the last row of the matrix in which case the head would be the node right next to the tail.

**Next head when going DOWN**  
Let's look at the two scenarios that we may come across when we are at the tail end of an upwards diagonal and we want to find the head of the next diagonal.



So, the general rule that we will be following when we want to find the head for a downwards going diagonal is that:

The head would be the node to the right of the tail of the previous diagonal. Unless the tail lies in the last column of the matrix in which case the head would be the node directly below the tail.

**Algorithm**

1. Initialize a boolean variable called direction which will tell us whether the current diagonal is an upwards or downwards going. Based on the current direction and the tail, we will determine the head of the next diagonal. Initially the direction would be 1 which would indicate up. We will keep alternating this value from one iteration to the next.
2. Assuming we know the head of a diagonal, say *matrix*[*i*][*j*], we will use the direction to progress along the diagonal and process its elements.
   * For an upwards going diagonal, the next element in the diagonal would be *matrix*[*i*−1][*j*+1]
   * For a downwards going diagonal, the next element would be *matrix*[*i*+1][*j*−1].
3. We keep processing the elements of the current diagonal until we go out of the boundaries of the matrix.
4. Now, given that we know the tail of the diagonal (the last node before we went out of bounds), let's see how we can find the next head. Note that in the following pseudocode, the direction is for the current diagonal and we are trying to find the head of the next diagonal. So, if the direction is up, it means the next diagonal would be going down and vice-versa.

|  |
| --- |
| tail = [i, j]  if direction == up, then {  if [i, j + 1] is within bounds, then {  next\_head = [i, j + 1]  } else {  next\_head = [i + 1, j]  }  } else {  if [i + 1, j] is within bounds, then {  next\_head = [i + 1, j]  } else {  next\_head = [i, j + 1]  }  } |

1. We keep processing the elements of a diagonal and once the current diagonal ends, we use the current direction and the tail element to find the next head and we switch over to processing the next diagonal. Also remember to flip the direction bit.

|  |
| --- |
| class Solution {  public int[] findDiagonalOrder(int[][] matrix) {    // Check for empty matrices  if (matrix == null || matrix.length == 0) {  return new int[0];  }    // Variables to track the size of the matrix  int N = matrix.length;  int M = matrix[0].length;    // Incides that will help us progress through  // the matrix, one element at a time.  int row = 0, column = 0;    // As explained in the article, this is the variable  // that helps us keep track of what direction we are  // processing the current diaonal  int direction = 1;    // The final result array  int[] result = new int[N\*M];  int r = 0;    // The uber while loop which will help us iterate over all  // the elements in the array.  while (row < N && column < M) {    // First and foremost, add the current element to  // the result matrix.  result[r++] = matrix[row][column];    // Move along in the current diagonal depending upon  // the current direction.[i, j] -> [i - 1, j + 1] if  // going up and [i, j] -> [i + 1][j - 1] if going down.  int new\_row = row + (direction == 1 ? -1 : 1);  int new\_column = column + (direction == 1 ? 1 : -1);    // Checking if the next element in the diagonal is within the  // bounds of the matrix or not. If it's not within the bounds,  // we have to find the next head.  if (new\_row < 0 || new\_row == N || new\_column < 0 || new\_column == M) {    // If the current diagonal was going in the upwards  // direction.  if (direction == 1) {    // For an upwards going diagonal having [i, j] as its tail  // If [i, j + 1] is within bounds, then it becomes  // the next head. Otherwise, the element directly below  // i.e. the element [i + 1, j] becomes the next head  row += (column == M - 1 ? 1 : 0) ;  column += (column < M - 1 ? 1 : 0);    } else {    // For a downwards going diagonal having [i, j] as its tail  // if [i + 1, j] is within bounds, then it becomes  // the next head. Otherwise, the element directly below  // i.e. the element [i, j + 1] becomes the next head  column += (row == N - 1 ? 1 : 0);  row += (row < N - 1 ? 1 : 0);  }    // Flip the direction  direction = 1 - direction;    } else {    row = new\_row;  column = new\_column;  }  }  return result;  }  } |

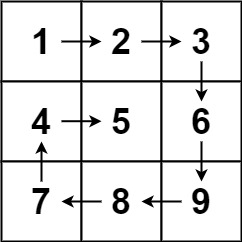
**Complexity Analysis**

* Time Complexity: *O*(*N*⋅*M*) since we process each element of the matrix exactly once.
* Space Complexity: *O*(1) since we don't make use of any additional data structure. Note that the space occupied by the output array doesn't count towards the space complexity since that is a requirement of the problem itself. Space complexity comprises any additional space that we may have used to get to build the final array. For the previous solution, it was the intermediate arrays. In this solution, we don't have any additional space apart from a couple of variables.

**Spiral Matrix**

Given an m x n matrix, return *all elements of the* matrix *in spiral order*.

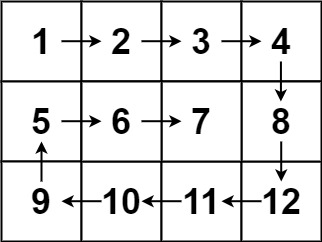
**Example 1:**



**Input:** matrix = [[1,2,3],[4,5,6],[7,8,9]]

**Output:** [1,2,3,6,9,8,7,4,5]

**Example 2:**



**Input:** matrix = [[1,2,3,4],[5,6,7,8],[9,10,11,12]]

**Output:** [1,2,3,4,8,12,11,10,9,5,6,7]

**Constraints:**

* m == matrix.length
* n == matrix[i].length
* 1 <= m, n <= 10
* -100 <= matrix[i][j] <= 100

Hint #1

Well for some problems, the best way really is to come up with some algorithms for simulation. Basically, you need to simulate what the problem asks us to do.

Hint #2

We go boundary by boundary and move inwards. That is the essential operation. First row, last column, last row, first column and then we move inwards by 1 and then repeat. That's all, that is all the simulation that we need.

Hint #3

Think about when you want to switch the progress on one of the indexes. If you progress on

i

out of

[i, j]

, you'd be shifting in the same column. Similarly, by changing values for

j

, you'd be shifting in the same row. Also, keep track of the end of a boundary so that you can move inwards and then keep repeating. It's always best to run the simulation on edge cases like a single column or a single row to see if anything breaks or not.

#### **Approach 1: Simulation**

**Intuition**

Draw the path that the spiral makes. We know that the path should turn clockwise whenever it would go out of bounds or into a cell that was previously visited.

**Algorithm**

Let the array have R rows and C columns. seen[r][c] denotes that the cell on the r-th row and c-th column was previously visited. Our current position is (r, c), facing direction di, and we want to visit R x C total cells.

As we move through the matrix, our candidate next position is (cr, cc). If the candidate is in the bounds of the matrix and unseen, then it becomes our next position; otherwise, our next position is the one after performing a clockwise turn.

|  |
| --- |
| class Solution {  public List<Integer> spiralOrder(int[][] matrix) {  List ans = new ArrayList();  if (matrix.length == 0) return ans;  int R = matrix.length, C = matrix[0].length;  boolean[][] seen = new boolean[R][C];  int[] dr = {0, 1, 0, -1};  int[] dc = {1, 0, -1, 0};  int r = 0, c = 0, di = 0;  for (int i = 0; i < R \* C; i++) {  ans.add(matrix[r][c]);  seen[r][c] = true;  int cr = r + dr[di];  int cc = c + dc[di];  if (0 <= cr && cr < R && 0 <= cc && cc < C && !seen[cr][cc]){  r = cr;  c = cc;  } else {  di = (di + 1) % 4;  r += dr[di];  c += dc[di];  }  }  return ans;  }  } |

**Complexity Analysis**

* Time Complexity: *O*(*N*), where *N* is the total number of elements in the input matrix. We add every element in the matrix to our final answer.
* Space Complexity: *O*(*N*), the information stored in seen and in ans.

#### **Approach 2: Layer-by-Layer**

**Intuition**

The answer will be all the elements in clockwise order from the first-outer layer, followed by the elements from the second-outer layer, and so on.

**Algorithm**

We define the k-th outer layer of a matrix as all elements that have minimum distance to some border equal to k. For example, the following matrix has all elements in the first-outer layer equal to 1, all elements in the second-outer layer equal to 2, and all elements in the third-outer layer equal to 3.

[[1, 1, 1, 1, 1, 1, 1],

[1, 2, 2, 2, 2, 2, 1],

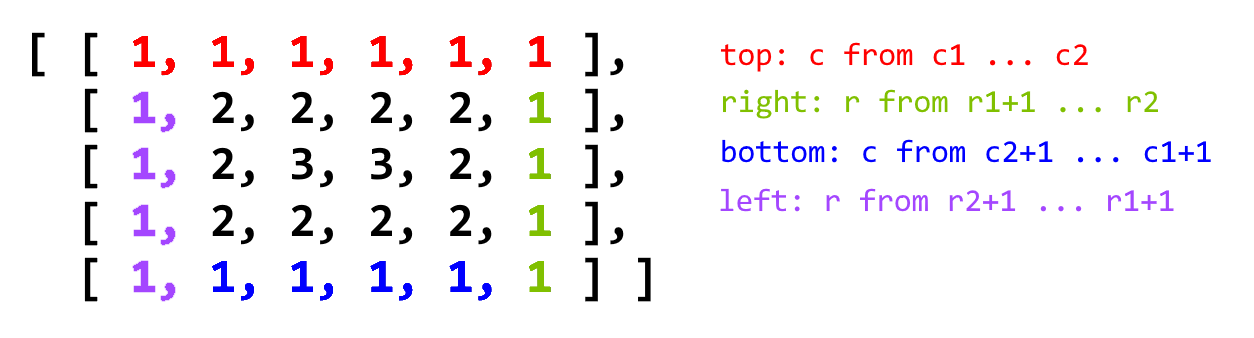
[1, 2, 3, 3, 3, 2, 1],

[1, 2, 2, 2, 2, 2, 1],

[1, 1, 1, 1, 1, 1, 1]]

For each outer layer, we want to iterate through its elements in clockwise order starting from the top left corner. Suppose the current outer layer has top-left coordinates  (r1, c1) and bottom-right coordinates (r2, c2).

Then, the top row is the set of elements  (r1, c) for c = c1,...,c2, in that order. The rest of the right side is the set of elements  (r, c2) for r = r1+1,...,r2, in that order. Then, if there are four sides to this layer (ie.,  r1 < r2 and c1 < c2), we iterate through the bottom side and left side as shown in the solutions below.



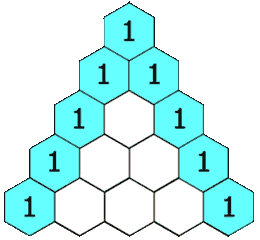
|  |
| --- |
| class Solution {  public List < Integer > spiralOrder(int[][] matrix) {  List ans = new ArrayList();  if (matrix.length == 0)  return ans;  int r1 = 0, r2 = matrix.length - 1;  int c1 = 0, c2 = matrix[0].length - 1;  while (r1 <= r2 && c1 <= c2) {  for (int c = c1; c <= c2; c++) ans.add(matrix[r1][c]);  for (int r = r1 + 1; r <= r2; r++) ans.add(matrix[r][c2]);  if (r1 < r2 && c1 < c2) {  for (int c = c2 - 1; c > c1; c--) ans.add(matrix[r2][c]);  for (int r = r2; r > r1; r--) ans.add(matrix[r][c1]);  }  r1++;  r2--;  c1++;  c2--;  }  return ans;  }  } |

**Complexity Analysis**

* Time Complexity: *O*(*N*), where *N* is the total number of elements in the input matrix. We add every element in the matrix to our final answer.
* Space Complexity:
  + *O*(1) without considering the output array, since we don't use any additional data structures for our computations.
  + *O*(*N*) if the output array is taken into account.

**Pascal's Triangle**

Given a non-negative integer numRows, generate the first numRows of Pascal's triangle.

  
In Pascal's triangle, each number is the sum of the two numbers directly above it.

**Example:**

**Input:** 5

**Output:**

[

[1],

[1,1],

[1,2,1],

[1,3,3,1],

[1,4,6,4,1]

]

#### **Approach 1: Dynamic Programming**

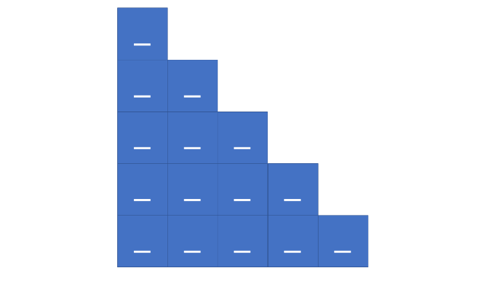
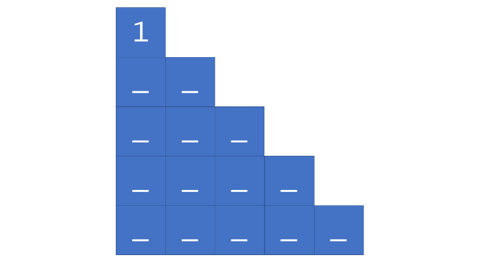
**Intuition**

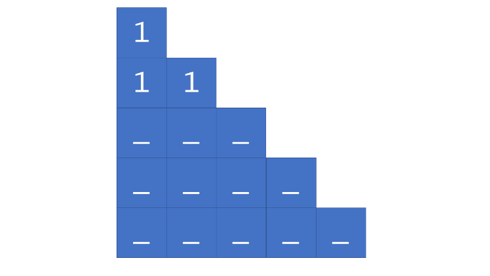
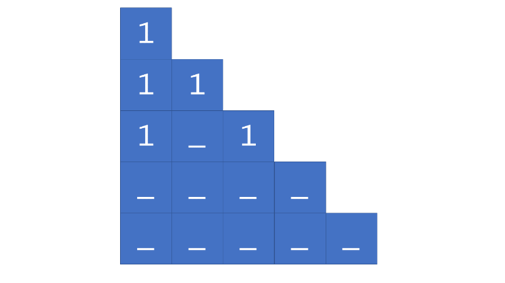
If we have the a row of Pascal's triangle, we can easily compute the next row by each pair of adjacent values.

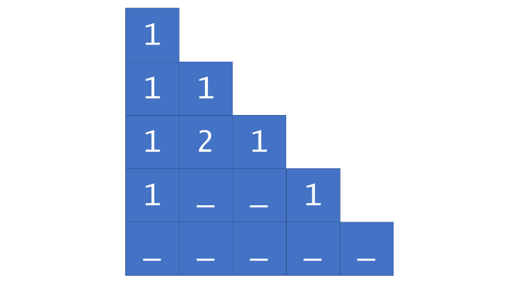
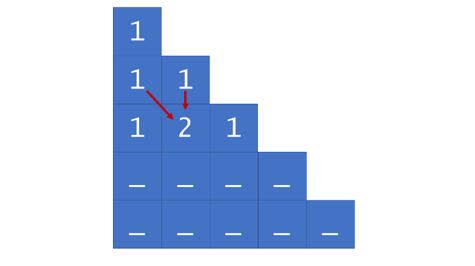
**Algorithm**

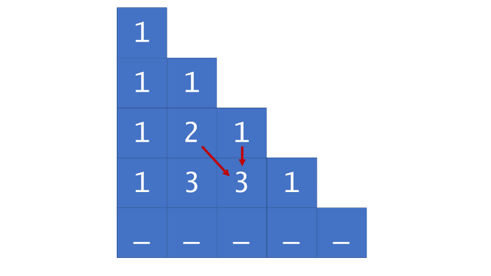
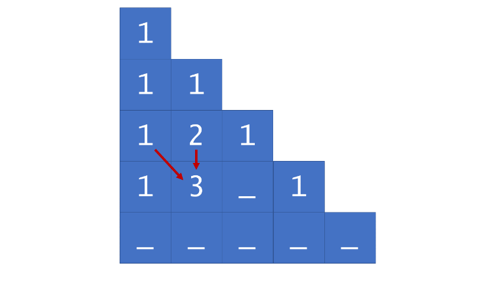
Although the algorithm is very simple, the iterative approach to constructing Pascal's triangle can be classified as dynamic programming because we construct each row based on the previous row.

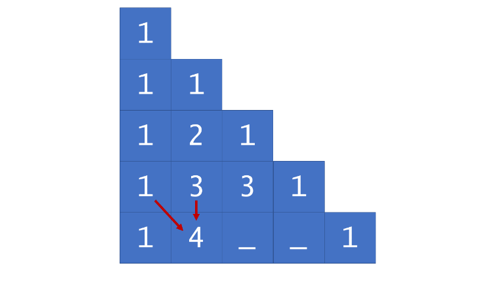
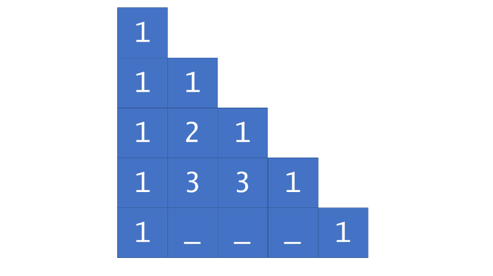
First, we generate the overall triangle list, which will store each row as a sublist. Then, we check for the special case of 0, as we would otherwise return [1]. If *numRows*>0, then we initialize triangle with [1] as its first row, and proceed to fill the rows as follows:

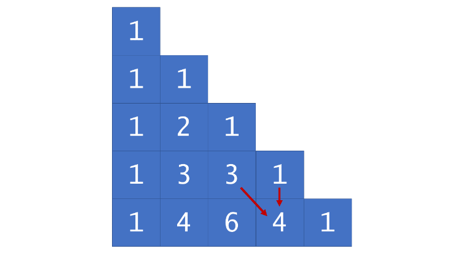
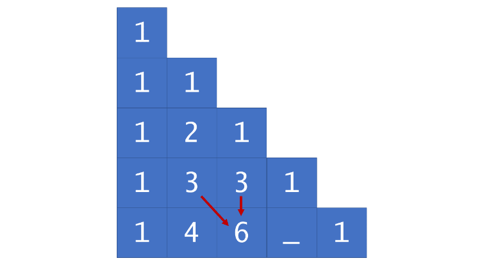
 

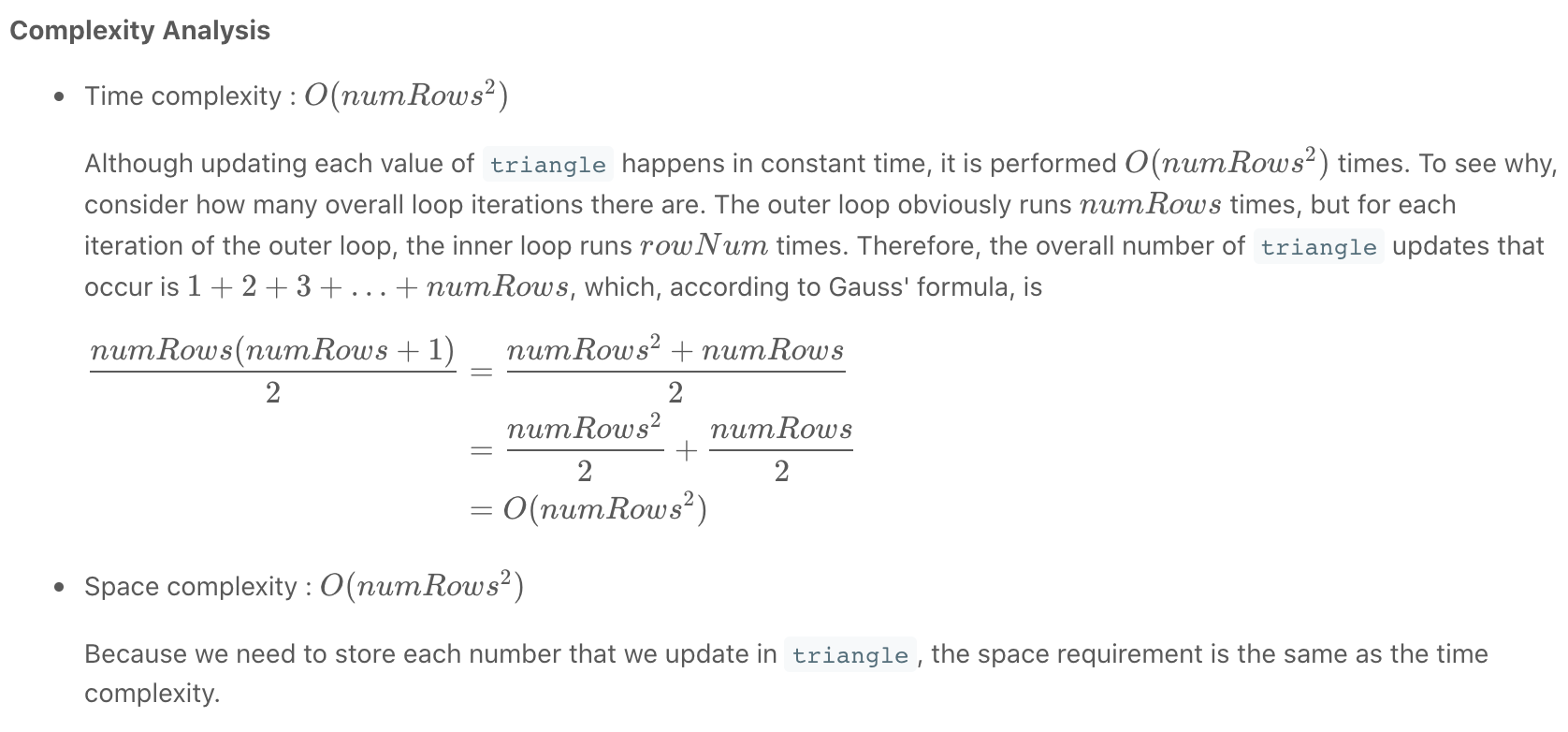








|  |
| --- |
| class Solution {  public List<List<Integer>> generate(int numRows) {  List<List<Integer>> triangle = new ArrayList<List<Integer>>();  // First base case; if user requests zero rows, they get zero rows.  if (numRows == 0) {  return triangle;  }  // Second base case; first row is always [1].  triangle.add(new ArrayList<>());  triangle.get(0).add(1);  for (int rowNum = 1; rowNum < numRows; rowNum++) {  List<Integer> row = new ArrayList<>();  List<Integer> prevRow = triangle.get(rowNum-1);  // The first row element is always 1.  row.add(1);  // Each triangle element (other than the first and last of each row)  // is equal to the sum of the elements above-and-to-the-left and  // above-and-to-the-right.  for (int j = 1; j < rowNum; j++) {  row.add(prevRow.get(j-1) + prevRow.get(j));  }  // The last row element is always 1.  row.add(1);  triangle.add(row);  }  return triangle;  }  } |



**Introduction to String**

As we mentioned in the overview, a string is an array of characters.

In this chapter, we will go deeper into the string. After this chapter, you will:

* Be familiar with the basic operations in a string, especially the unique operations which we don't have in an array
* Understand the differences between different comparison functions;
* Determine whether the string is immutable or not in your favorite language and understand the impact of this feature;
* Be able to solve basic string-related problems.

A string is actually an array of unicode characters. You can perform almost all the operations we used in an array. You can try it out by yourself.

However, there are some differences. In this article, we will go through some of them which you should be aware of when dealing with a string. These features might vary a lot from one language to another.

### *Compare Function*

String has its own compare function (we will show you the usage of compare function in the code below).

However, there is a problem:

Can we use "==" to compare two strings?

It depends on the answer to the question:

Does the language support operator overloading?

1. If the answer is yes (like C++), we may use "==" to compare two strings.
2. If the answer is no (like Java), we may not use "==" to compare two strings. When we use "==", it actually compares whether these two objects are the same object.

Let's run the following example and compare the results:

|  |
| --- |
| // "static void main" must be defined in a public class.  public class Main {  public static void main(String[] args) {  // initialize  String s1 = "Hello World";  System.out.println("s1 is \"" + s1 + "\"");  String s2 = s1;  System.out.println("s2 is another reference to s1.");  String s3 = new String(s1);  System.out.println("s3 is a copy of s1.");  // compare using '=='  System.out.println("Compared by '==':");  // true since string is immutable and s1 is binded to "Hello World"  System.out.println("s1 and \"Hello World\": " + (s1 == "Hello World"));  // true since s1 and s2 is the reference of the same object  System.out.println("s1 and s2: " + (s1 == s2));  // false since s3 is refered to another new object  System.out.println("s1 and s3: " + (s1 == s3));  // compare using 'equals'  System.out.println("Compared by 'equals':");  System.out.println("s1 and \"Hello World\": " + s1.equals("Hello World"));  System.out.println("s1 and s2: " + s1.equals(s2));  System.out.println("s1 and s3: " + s1.equals(s3));  // compare using 'compareTo'  System.out.println("Compared by 'compareTo':");  System.out.println("s1 and \"Hello World\": " + (s1.compareTo("Hello World") == 0));  System.out.println("s1 and s2: " + (s1.compareTo(s2) == 0));  System.out.println("s1 and s3: " + (s1.compareTo(s3) == 0));  }  } |

### *Immutable or Mutable*

Immutable means that you can't change the content of the string once it's initialized.

1. In some languages (like C++), the string is mutable. That is to say, you can modify the string just like what you did in an array.
2. In some other languages (like Java), the string is immutable. This feature will bring several problems. We will illustrate the problems and solutions in the next article.

You can determine whether the string in your favorite language is immutable or mutable by testing the modification operation. Here is an example:

|  |
| --- |
| // "static void main" must be defined in a public class.  public class Main {  public static void main(String[] args) {  String s1 = "Hello World";  s1[5] = ',';  System.out.println(s1);  }  } |

*Extra Operations*

Compare to an array, there are some extra operations we can perform on a string. Here are some examples:

|  |
| --- |
| // "static void main" must be defined in a public class.  public class Main {  public static void main(String[] args) {  String s1 = "Hello World";  // 1. concatenate  s1 += "!";  System.out.println(s1);  // 2. find  System.out.println("The position of first 'o' is: " + s1.indexOf('o'));  System.out.println("The position of last 'o' is: " + s1.lastIndexOf('o'));  // 3. get substring  System.out.println(s1.substring(6, 11));  }  } |

You should be aware of the time complexity of these built-in operations.

For instance, if the length of the string is N, the time complexity of both finding operation and substring operation is O(N).

Also, in languages which the string is immutable, you should be careful with the concatenation operation (we will explain this in next article as well).

Never forget to take the time complexity of built-in operations into consideration when you compute the time complexity for your solution.

**Immutable String - Problems & Solutions**

You should know whether the string in your favourite language is immutable or not in the previous article. If the string is immutable, it will bring some problems. Hopefully, we will also provide the solution at the end.

*Modification Operation*

Obviously, an immutable string cannot be modified. If you want to modify just one of the characters, you have to create a new string.

*Beware of String Concatenation in Java*

You should be very careful with string concatenation. Let's look at an example when we do string concatenation repeatedly in a for loop:

|  |
| --- |
| // "static void main" must be defined in a public class.  public class Main {  public static void main(String[] args) {  String s = "";  int n = 10000;  for (int i = 0; i < n; i++) {  s += "hello";  }  }  } |

Notice how slow string concatenation is for Java? On the other hand, there is no noticeable performance impact in C++.

In Java, since the string is immutable, concatenation works by first allocating enough space for the new string, copy the contents from the old string and append to the new string.

Therefore, the time complexity in total will be:

*5 + 5 × 2 + 5 × 3 + … + 5 × n  
= 5 × (1 + 2 + 3 + … + n)  
= 5 × n × (n + 1) / 2*,

which is O(n2).

*Solutions*

If you want your string to be mutable, there are some substitutions:

**1. If you did want your string to be mutable, you can convert it to a char array.**

|  |
| --- |
| // "static void main" must be defined in a public class.  public class Main {  public static void main(String[] args) {  String s = "Hello World";  char[] str = s.toCharArray();  str[5] = ',';  System.out.println(str);  }  } |

**2. If you have to concatenate strings often, it will be better to use some other data structures like StringBuilder. The below code runs in O(n) complexity.**

|  |
| --- |
| // "static void main" must be defined in a public class.  public class Main {  public static void main(String[] args) {  int n = 10000;  StringBuilder str = new StringBuilder();  for (int i = 0; i < n; i++) {  str.append("hello");  }  String s = str.toString();  }  } |

**Add Binary**

Given two binary strings a and b, return *their sum as a binary string*.

**Example 1:**

**Input:** a = "11", b = "1"

**Output:** "100"

**Example 2:**

**Input:** a = "1010", b = "1011"

**Output:** "10101"

**Constraints:**

* 1 <= a.length, b.length <= 104
* a and b consist only of '0' or '1' characters.
* Each string does not contain leading zeros except for the zero itself.

## **Solution**

#### **Overview**

There is a simple way using built-in functions:

* Convert a and b into integers.
* Compute the sum.
* Convert the sum back into binary form.

|  |
| --- |
| class Solution {  public String addBinary(String a, String b) {  return Integer.toBinaryString(Integer.parseInt(a, 2) + Integer.parseInt(b, 2));  }  } |

The overall algorithm has O(*N*+*M*) time complexity and has two drawbacks which could be used against you during the interview.

1 . In Java this approach is limited by the length of the input strings a and b. Once the string is long enough, the result of conversion into integers will not fit into Integer, Long or BigInteger:

* 33 1-bits - and b doesn't fit into Integer.
* 65 1-bits - and b doesn't fit into Long.
* [500000001](https://docs.oracle.com/javase/8/docs/api/java/math/BigInteger.html) 1-bits - and b doesn't fit into BigInteger.

To fix the issue, one could use standard Bit-by-Bit Computation approach which is suitable for quite long input strings.

2 . This method has quite low performance in the case of large input numbers.

One could use Bit Manipulation approach to speed up the solution.

#### **Approach 1: Bit-by-Bit Computation**

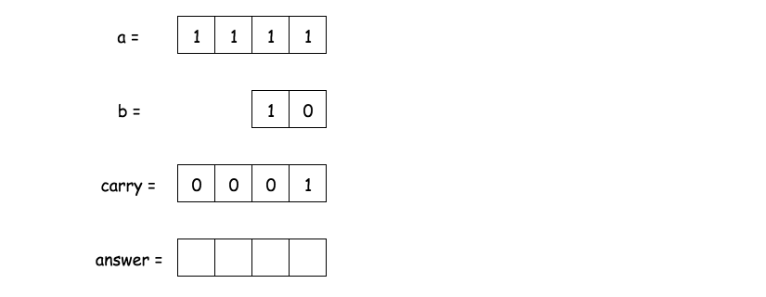
**Algorithm**

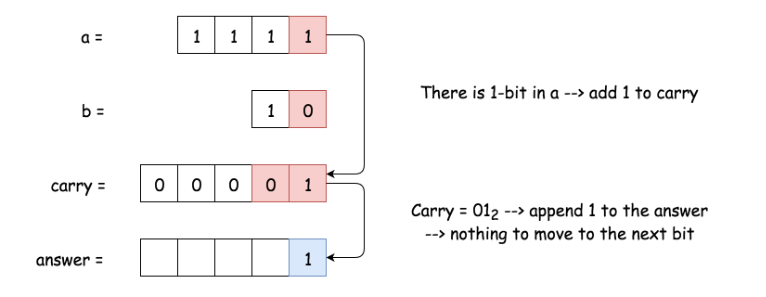
That's a good old classical algorithm, and there is no conversion from binary string to decimal and back here. Let's consider the numbers bit by bit starting from the lowest one and compute the carry this bit will add.

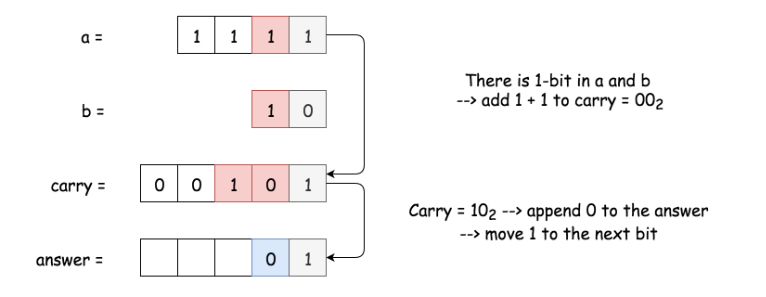
Start from carry = 0. If number a has 1-bit in this lowest bit, add 1 to the carry. The same for number b: if number b has 1-bit in the lowest bit, add 1 to the carry. At this point the carry for the lowest bit could be equal to (00)2​, (01)2​, or (10)2​.

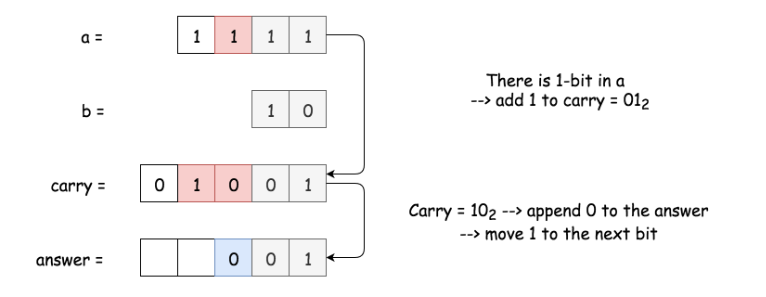
Now append the lowest bit of the carry to the answer, and move the highest bit of the carry to the next order bit.

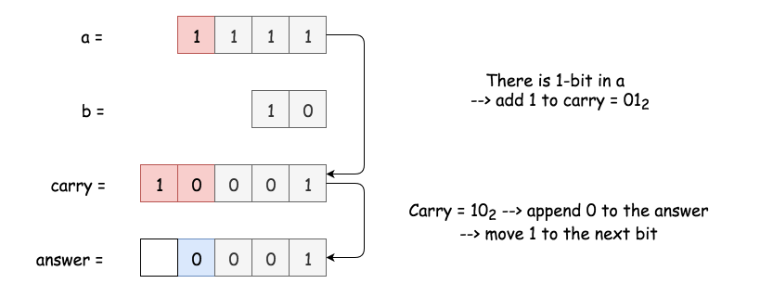
Repeat the same steps again, and again, till all bits in a and b are used up. If there is still nonzero carry to add, add it. Now reverse the answer string and the job is done.

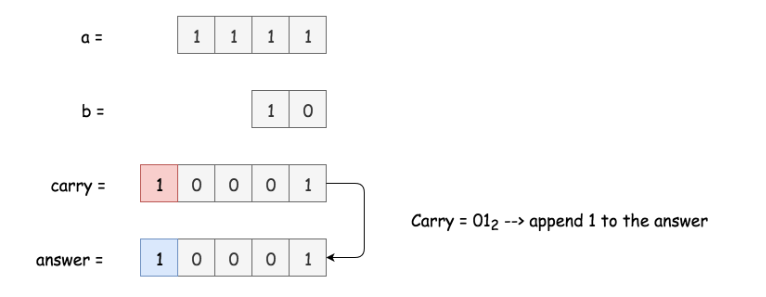


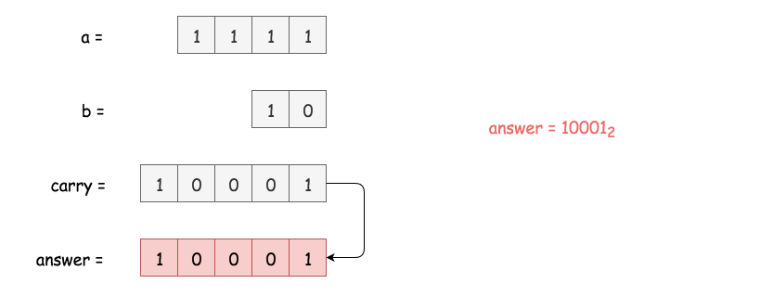












**Implementation**

|  |
| --- |
| class Solution {  public String addBinary(String a, String b) {  int n = a.length(), m = b.length();  if (n < m) return addBinary(b, a);  int L = Math.max(n, m);  StringBuilder sb = new StringBuilder();  int carry = 0, j = m - 1;  for(int i = L - 1; i > -1; --i) {  if (a.charAt(i) == '1') ++carry;  if (j > -1 && b.charAt(j--) == '1') ++carry;  if (carry % 2 == 1) sb.append('1');  else sb.append('0');  carry /= 2;  }  if (carry == 1) sb.append('1');  sb.reverse();  return sb.toString();  }  } |

**Complexity Analysis**

* Time complexity: O(max(*N*,*M*)), where *N* and *M* are lengths of the input strings a and b.
* Space complexity: O(max(*N*,*M*)) to keep the answer.

#### **Approach 2: Bit Manipulation**

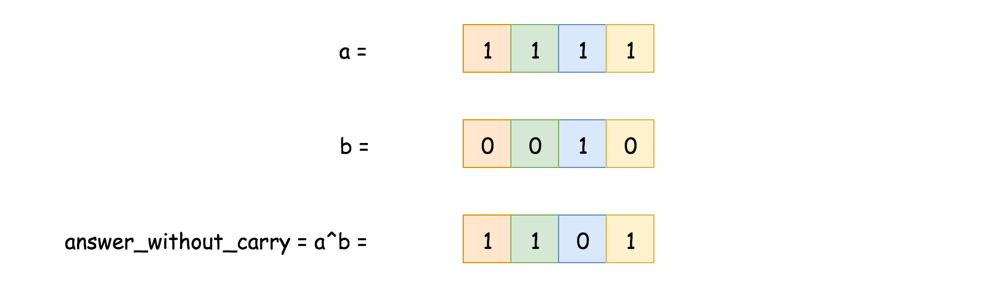
**Intuition**

Here the input is more adapted to push towards Approach 1, but there is popular Facebook variation of this problem when interviewer provides you two numbers and asks to sum them up without using addition operation.

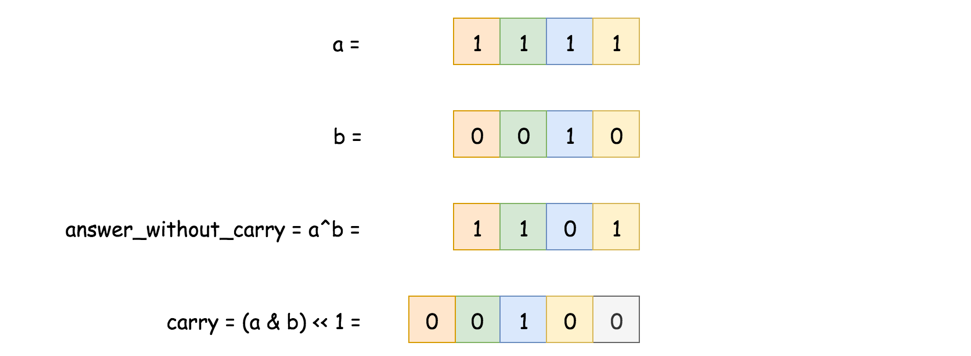
No addition? OK, bit manipulation then.

How to start? There is an interview tip for bit manipulation problems: if you don't know how to start, start from computing XOR for your input data. Strangely, that helps to go out for quite a lot of problems, [Single Number II](https://leetcode.com/articles/single-number-ii/), [Single Number III](https://leetcode.com/articles/single-number-iii/), [Maximum XOR of Two Numbers in an Array](https://leetcode.com/articles/maximum-xor-of-two-numbers-in-an-array/), [Repeated DNA Sequences](https://leetcode.com/articles/repeated-dna-sequences/), [Maximum Product of Word Lengths](https://leetcode.com/articles/maximum-product-of-word-lengths/), etc.

Here XOR is a key as well, because it's a sum of two binaries without taking carry into account.



To find current carry is quite easy as well, it's AND of two input numbers, shifted one bit to the left.



Now the problem is reduced: one has to find the sum of answer without carry and carry. It's the same problem - to sum two numbers, and hence one could solve it in a loop with the condition statement "while carry is not equal to zero".

**Algorithm**

* Convert a and b into integers x and y, x will be used to keep an answer, and y for the carry.
* While carry is nonzero: y != 0:
  + Current answer without carry is XOR of x and y: answer = x^y.
  + Current carry is left-shifted AND of x and y: carry = (x & y) << 1.
  + Job is done, prepare the next loop: x = answer, y = carry.
* Return x in the binary form.

**Implementation**

|  |
| --- |
| import java.math.BigInteger;  class Solution {  public String addBinary(String a, String b) {  BigInteger x = new BigInteger(a, 2);  BigInteger y = new BigInteger(b, 2);  BigInteger zero = new BigInteger("0", 2);  BigInteger carry, answer;  while (y.compareTo(zero) != 0) {  answer = x.xor(y);  carry = x.and(y).shiftLeft(1);  x = answer;  y = carry;  }  return x.toString(2);  }  } |

**Performance Discussion**

Here we deal with input numbers which are greater than 2^100. That forces to use slow [BigInteger](https://docs.oracle.com/javase/8/docs/api/java/math/BigInteger.html) in Java, and hence the performance gain will be present for the Python solution only. Provided here Java solution could make its best with Integers or Longs, but not with BigIntegers.

**Complexity Analysis**

* Time complexity : O(*N*+*M*), where *N* and *M* are lengths of the input strings a and b.
* Space complexity :  O(max(*N*,*M*)) to keep the answer.

**Implement strStr()**

Implement [strStr()](http://www.cplusplus.com/reference/cstring/strstr/" \t "_blank).

Return the index of the first occurrence of needle in haystack, or -1 if needle is not part of haystack.

**Clarification:**

What should we return when needle is an empty string? This is a great question to ask during an interview.

For the purpose of this problem, we will return 0 when needle is an empty string. This is consistent to C's [strstr()](http://www.cplusplus.com/reference/cstring/strstr/" \t "_blank) and Java's [indexOf()](https://docs.oracle.com/javase/7/docs/api/java/lang/String.html" \l "indexOf(java.lang.String)" \t "_blank).

**Example 1:**

**Input:** haystack = "hello", needle = "ll"

**Output:** 2

**Example 2:**

**Input:** haystack = "aaaaa", needle = "bba"

**Output:** -1

**Example 3:**

**Input:** haystack = "", needle = ""

**Output:** 0

**Constraints:**

* 0 <= haystack.length, needle.length <= 5 \* 104
* haystack and needle consist of only lower-case English characters.

## **Solution**

#### **Overview**

The problem is to find needle of length L in the haystack of length N.

Let's discuss three different ideas how to proceed. They are all based on sliding window idea. The key point is how to implement a window slice.

Linear time window slice O(*L*) is quite easy, move the window of length L along the haystack and compare substring in the window with the needle. Overall that would result in O((*N*−*L*)*L*) time complexity.

Could that be improved? Yes. Two pointers approach is still the case of linear time slice, though the comparison happens not for all substrings, and that improves the best time complexity up to O(*N*). The worst time complexity is still O((*N*−*L*)*L*) though.

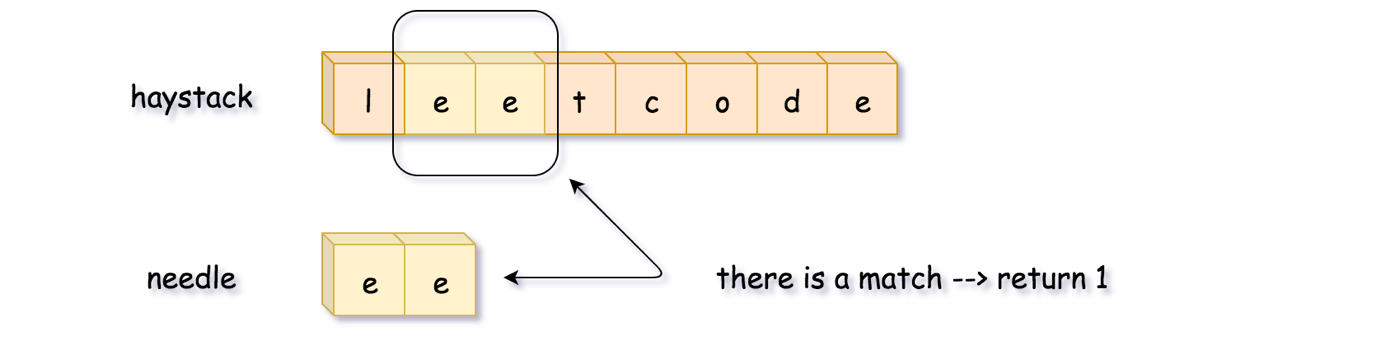
Could that be improved to O(*N*)? Yes, but one has to implement constant time slice O(1). There are two ways to do it:

* Rabin-Karp = constant-time slice using rolling hash algorithm.
* Bit manipulation = constant-time slice using bitmasks.

Bit Manipulation approach in Java is more suitable for the short strings or strings with very limited number of characters, ex. [Repeated DNA Sequences](https://leetcode.com/articles/repeated-dna-sequences/). That's a consequence of overflow issues in Java (in Python there is no such a problem). Here we deal with quite long strings and it's more simple to implement the basic version of Rabin Karp algorithm.

#### **Approach 1: Substring: Linear Time Slice**

Quite straightforward approach - move sliding window along the string and compare substring in the window with the needle.



**Implementation**

|  |
| --- |
| class Solution {  public int strStr(String haystack, String needle) {  int L = needle.length(), n = haystack.length();  for (int start = 0; start < n - L + 1; ++start) {  if (haystack.substring(start, start + L).equals(needle)) {  return start;  }  }  return -1;  }  } |

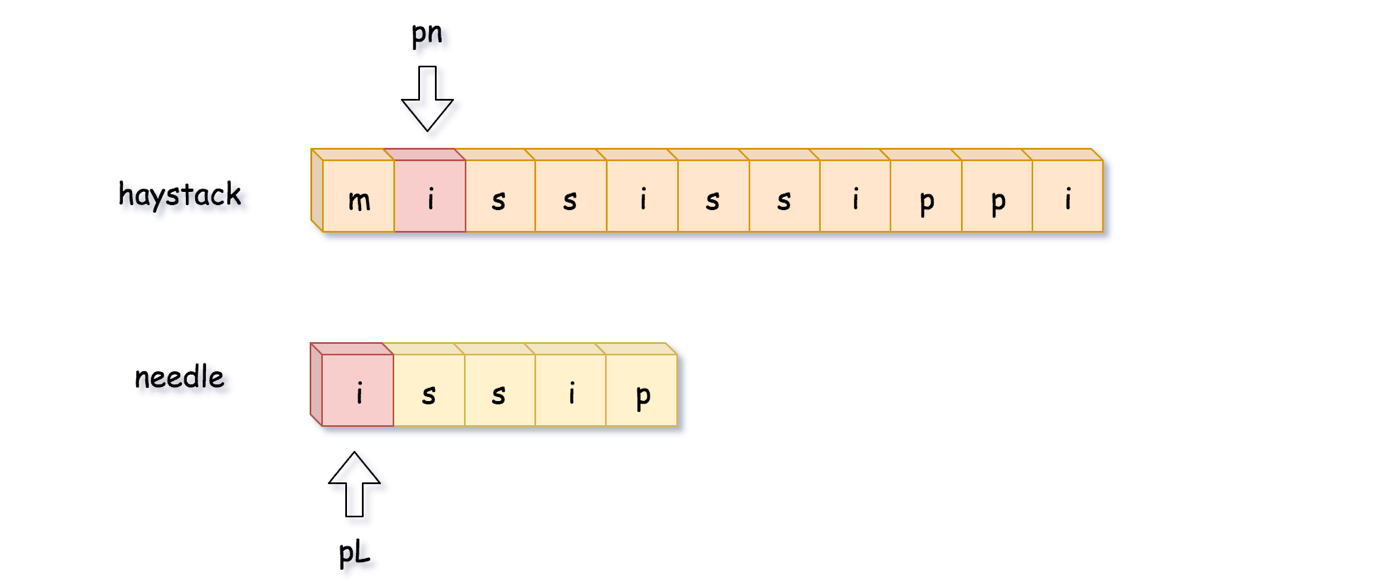
**Complexity Analysis**

* Time complexity: O((*N*−*L*)*L*), where N is a length of haystack and L is a length of needle. We compute a substring of length L in a loop, which is executed (N - L) times.
* Space complexity: O(1).

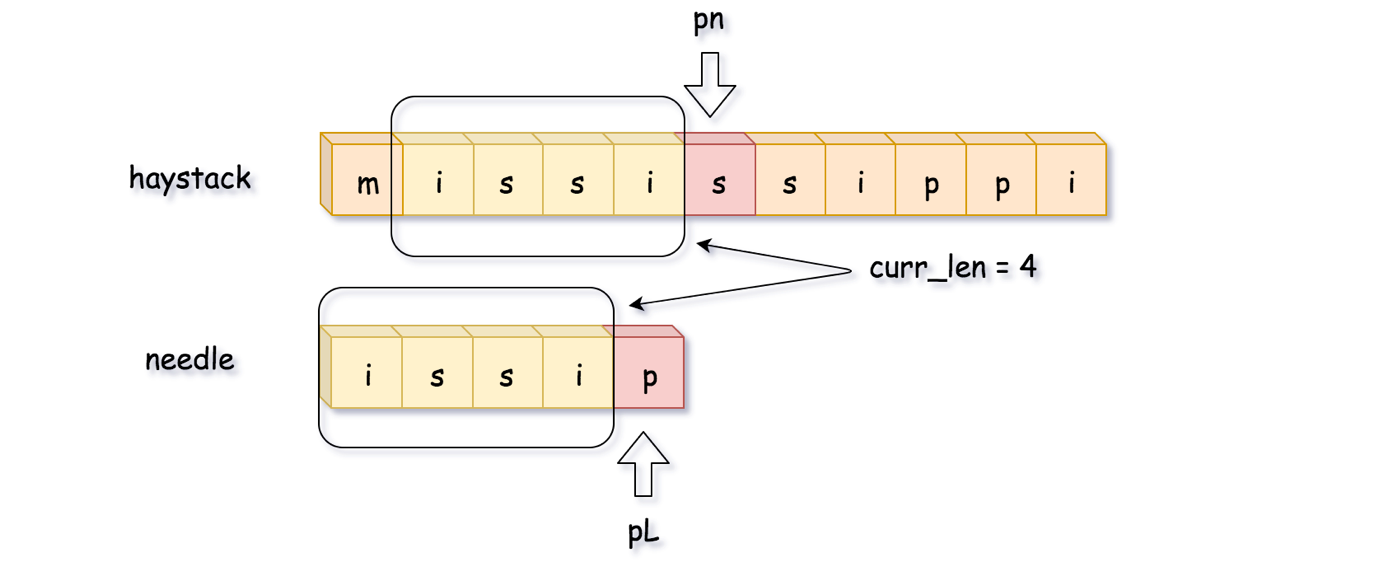
#### **Approach 2: Two Pointers: Linear Time Slice**

Drawback of the previous algorithm is that one compares absolutely all substrings of length L with the needle. There is no need to that.

First, let's compare only substrings which starts from the first character in the needle substring.



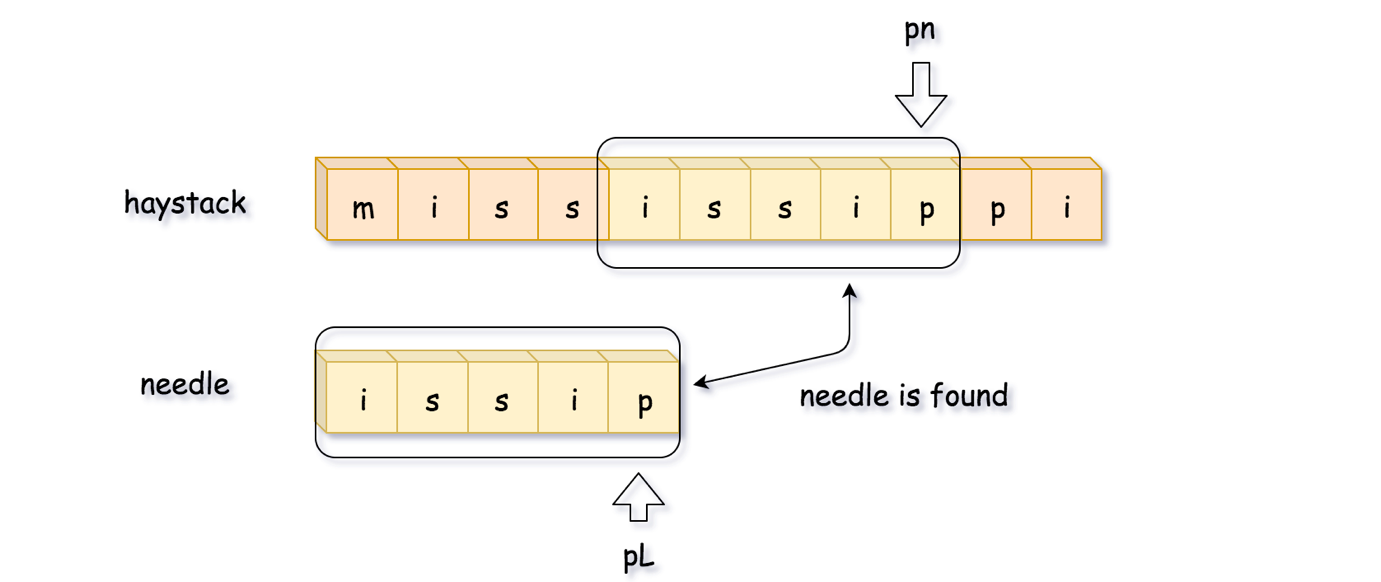
Second, let's compare the characters one by one and stop immediately in the case of mismatch.



Here it was impossible to manage the full match up to the length of needle string, which is L = 5. Let's backtrack then. Note, that we move pn pointer back to the position **pn = pn - curr\_len + 1**, and not to the position **pn = pn - curr\_len**, since this last one was already investigated.



Let's try again. Here we've managed to get the full match during the second attempt, so let's return the start position of that match, **pn - L**.



**Algorithm**

* Move **pn** till you'll find the first character of the needle string in the haystack.
* Compute the max string match by increasing **pn**, **pL** and **curr\_len** in the case of equal characters.
* If you managed to get the full match, **curr\_len == L**, return the start position of that match, **pn - L**.
* If you didn't, backtrack: **pn = pn - curr\_len + 1**, **pL = 0**, **curr\_len = 0**.

**Implementation**

|  |
| --- |
| class Solution {  public int strStr(String haystack, String needle) {  int L = needle.length(), n = haystack.length();  if (L == 0) return 0;  int pn = 0;  while (pn < n - L + 1) {  // find the position of the first needle character  // in the haystack string  while (pn < n - L + 1 && haystack.charAt(pn) != needle.charAt(0)) ++pn;  // compute the max match string  int currLen = 0, pL = 0;  while (pL < L && pn < n && haystack.charAt(pn) == needle.charAt(pL)) {  ++pn;  ++pL;  ++currLen;  }  // if the whole needle string is found,  // return its start position  if (currLen == L) return pn - L;  // otherwise, backtrack  pn = pn - currLen + 1;  }  return -1;  }  } |

**Complexity Analysis**

* Time complexity: O((*N*−*L*)*L*) in the worst case of numerous almost complete false matches, and O(*N*) in the best case of one single match.
* Space complexity: O(1).

#### **Approach 3: Rabin Karp: Constant Time Slice**

Let's now design the algorithm with O(*N*) time complexity even in the worst case. The idea is simple: move along the string, generate hash of substring in the sliding window and compare it with the reference hash of the needle string.

There are two technical problems:

1. How to implement a string slice in a constant time?
2. How to generate substring hash in a constant time?

**String slice in a constant time**

Strings are immutable in Java and Python, and to move sliding window in a constant time one has to convert string to another data structure, for example, to integer array of ascii-values.

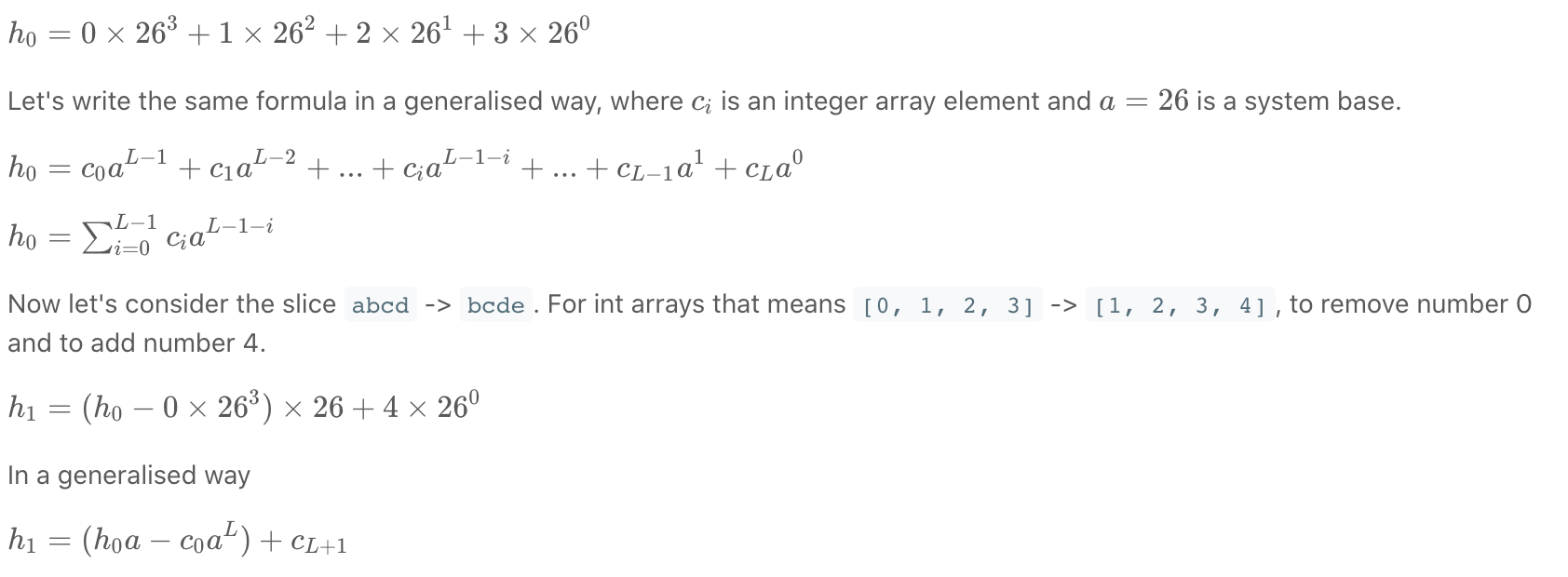
**Rolling hash: hash generation in a constant time**

To generate hash of array of length L, one needs O(*L*) time.

How to have constant time of hash generation? Use the advantage of slice: only one integer in, and only one - out.

That's the idea of [rolling hash](https://en.wikipedia.org/wiki/Rolling_hash). Here we'll implement the simplest one, polynomial rolling hash. Beware that's polynomial rolling hash is NOT the [Rabin fingerprint](https://en.wikipedia.org/wiki/Rolling_hash#Rabin_fingerprint).

Since one deals here with lowercase English letters, all values in the integer array are between 0 and 25 : arr[i] = (int)S.charAt(i) - (int)'a'.  
So one could consider string abcd -> [0, 1, 2, 3] as a number in a [numeral system](https://en.wikipedia.org/wiki/Numeral_system) with the base 26. Hence abcd -> [0, 1, 2, 3] could be hashed as



Now hash regeneration is perfect and fits in a constant time. There is one more issue to address: possible overflow problem.

**How to avoid overflow**

a^L could be a large number and hence the idea is to set limits to avoid the overflow. To set limits means to limit a hash by a given number called modulus and use everywhere not hash itself but h % modulus.

It's quite obvious that modulus should be large enough, but how large? [Here one could read more about the topic](https://en.wikipedia.org/wiki/Linear_congruential_generator#Parameters_in_common_use), for the problem here 2^31 is enough.

**Algorithm**

* Compute the hash of substring haystack.substring(0, L) and reference hash of needle.substring(0, L).
* Iterate over the start position of possible match: from 1 to N - L.
  + Compute rolling hash based on the previous hash value.
  + Return start position if the hash is equal to the reference one.
* Return -1, that means that needle is not found.

**Implementation**

|  |
| --- |
| class Solution {  // function to convert character to integer  public int charToInt(int idx, String s) {  return (int)s.charAt(idx) - (int)'a';  }  public int strStr(String haystack, String needle) {  int L = needle.length(), n = haystack.length();  if (L > n) return -1;  // base value for the rolling hash function  int a = 26;  // modulus value for the rolling hash function to avoid overflow  long modulus = (long)Math.pow(2, 31);  // compute the hash of strings haystack[:L], needle[:L]  long h = 0, ref\_h = 0;  for (int i = 0; i < L; ++i) {  h = (h \* a + charToInt(i, haystack)) % modulus;  ref\_h = (ref\_h \* a + charToInt(i, needle)) % modulus;  }  if (h == ref\_h) return 0;  // const value to be used often : a\*\*L % modulus  long aL = 1;  for (int i = 1; i <= L; ++i) aL = (aL \* a) % modulus;  for (int start = 1; start < n - L + 1; ++start) {  // compute rolling hash in O(1) time  h = (h \* a - charToInt(start - 1, haystack) \* aL  + charToInt(start + L - 1, haystack)) % modulus;  if (h == ref\_h) return start;  }  return -1;  }  } |

**Complexity Analysis**

* Time complexity:  O(*N*), one computes the reference hash of the needle string in O(*L*) time, and then runs a loop of (*N*−*L*) steps with constant time operations in it.
* Space complexity: O(1).

**Longest Common Prefix**

Write a function to find the longest common prefix string amongst an array of strings.

If there is no common prefix, return an empty string "".

**Example 1:**

**Input:** strs = ["flower","flow","flight"]

**Output:** "fl"

**Example 2:**

**Input:** strs = ["dog","racecar","car"]

**Output:** ""

**Explanation:** There is no common prefix among the input strings.

**Constraints:**

* 0 <= strs.length <= 200
* 0 <= strs[i].length <= 200
* strs[i] consists of only lower-case English letters.

## **Solution**

#### **Approach 1: Horizontal scanning**

**Intuition**

For a start we will describe a simple way of finding the longest prefix shared by a set of strings *LCP*(*S*1​…*Sn*​). We will use the observation that :

*LCP*(*S*1​…*Sn*​)=*LCP*(*LCP*(*LCP*(*S*1​,*S*2​),*S*3​),…*Sn*​)

**Algorithm**

To employ this idea, the algorithm iterates through the strings [*S*1​…*Sn*​], finding at each iteration *i* the longest common prefix of strings *LCP*(*S*1​…*Si*​) When *LCP*(*S*1​…*Si*​) is an empty string, the algorithm ends. Otherwise after *n* iterations, the algorithm returns *LCP*(*S*1​…*Sn*​).

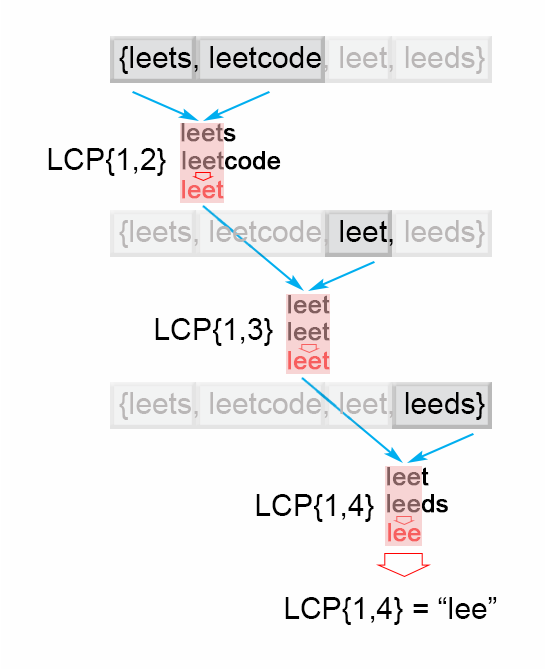


Figure 1. Finding the longest common prefix (Horizontal scanning)

|  |
| --- |
| public String longestCommonPrefix(String[] strs) {  if (strs.length == 0) return "";  String prefix = strs[0];  for (int i = 1; i < strs.length; i++)  while (strs[i].indexOf(prefix) != 0) {  prefix = prefix.substring(0, prefix.length() - 1);  if (prefix.isEmpty()) return "";  }  return prefix;  } |

**Complexity Analysis**

* Time complexity : *O*(*S*) , where S is the sum of all characters in all strings.

In the worst case all *n* strings are the same. The algorithm compares the string *S*1 with the other strings [*S*2​…*Sn*​] There are *S* character comparisons, where *S* is the sum of all characters in the input array.

* Space complexity : *O*(1). We only used constant extra space.

#### **Approach 2: Vertical scanning**

**Algorithm**

Imagine a very short string is at the end of the array. The above approach will still do S*S* comparisons. One way to optimize this case is to do vertical scanning. We compare characters from top to bottom on the same column (same character index of the strings) before moving on to the next column.

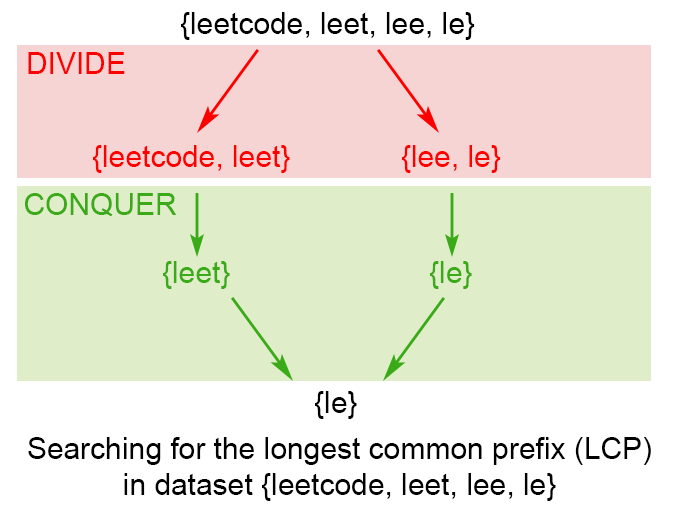
|  |
| --- |
| public String longestCommonPrefix(String[] strs) {  if (strs == null || strs.length == 0) return "";  for (int i = 0; i < strs[0].length() ; i++){  char c = strs[0].charAt(i);  for (int j = 1; j < strs.length; j ++) {  if (i == strs[j].length() || strs[j].charAt(i) != c)  return strs[0].substring(0, i);  }  }  return strs[0];  } |

**Complexity Analysis**

* Time complexity : *O*(*S*) , where S is the sum of all characters in all strings. In the worst case there will be *n* equal strings with length *m* and the algorithm performs *S*=*m*⋅*n* character comparisons. Even though the worst case is still the same as [Approach 1](https://leetcode.com/problems/longest-common-prefix/solution/#approach-1-horizontal-scanning), in the best case there are at most  n⋅*minLen* comparisons where *minLen* is the length of the shortest string in the array.
* Space complexity : *O*(1). We only used constant extra space.

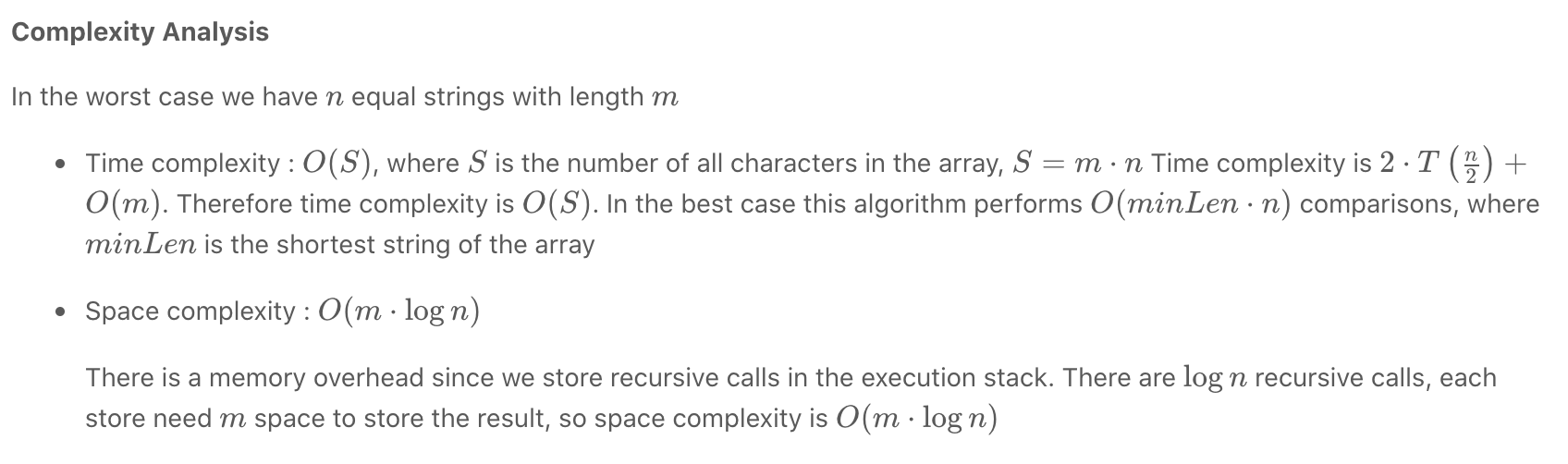
#### **Approach 3: Divide and conquer**





*Figure 2. Finding the longest common prefix of strings using divide and conquer technique*

|  |
| --- |
| public String longestCommonPrefix(String[] strs) {  if (strs == null || strs.length == 0) return "";  return longestCommonPrefix(strs, 0 , strs.length - 1);  }  private String longestCommonPrefix(String[] strs, int l, int r) {  if (l == r) {  return strs[l];  }  else {  int mid = (l + r)/2;  String lcpLeft = longestCommonPrefix(strs, l , mid);  String lcpRight = longestCommonPrefix(strs, mid + 1,r);  return commonPrefix(lcpLeft, lcpRight);  }  }  String commonPrefix(String left,String right) {  int min = Math.min(left.length(), right.length());  for (int i = 0; i < min; i++) {  if ( left.charAt(i) != right.charAt(i) )  return left.substring(0, i);  }  return left.substring(0, min);  } |



#### **Approach 4: Binary search**

The idea is to apply binary search method to find the string with maximum value L, which is common prefix of all of the strings. The algorithm searches space is the interval (0…*minLen*), where minLen is minimum string length and the maximum possible common prefix. Each time search space is divided in two equal parts, one of them is discarded, because it is sure that it doesn't contain the solution. There are two possible cases:

* S[1...mid] is not a common string. This means that for each j > i S[1..j] is not a common string and we discard the second half of the search space.
* S[1...mid] is common string. This means that for for each i < j S[1..i] is a common string and we discard the first half of the search space, because we try to find longer common prefix.

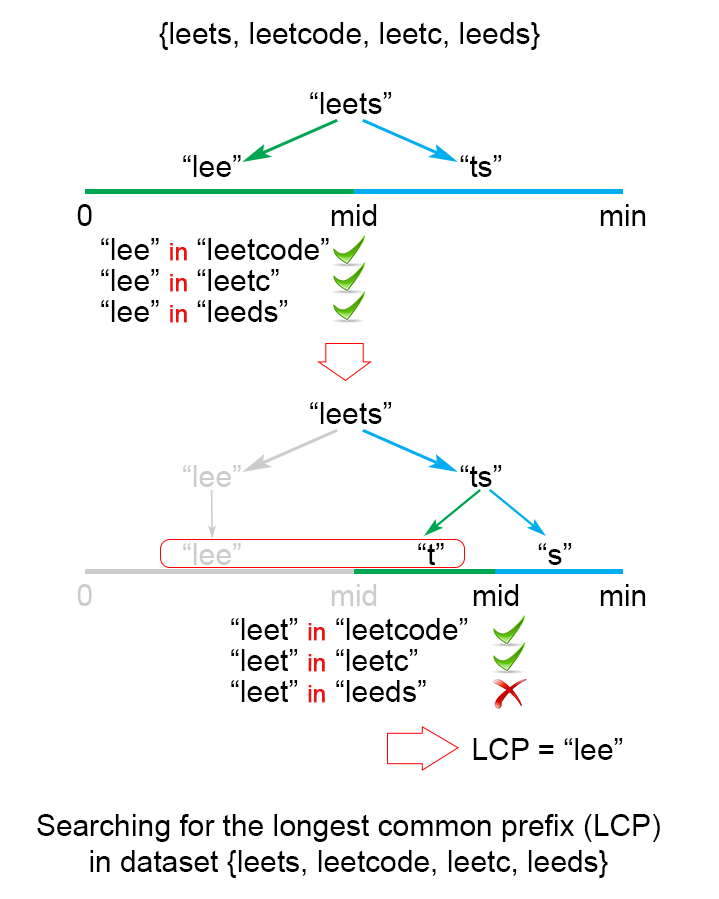


Figure 3. Finding the longest common prefix of strings using binary search technique

|  |
| --- |
| public String longestCommonPrefix(String[] strs) {  if (strs == null || strs.length == 0)  return "";  int minLen = Integer.MAX\_VALUE;  for (String str : strs)  minLen = Math.min(minLen, str.length());  int low = 1;  int high = minLen;  while (low <= high) {  int middle = (low + high) / 2;  if (isCommonPrefix(strs, middle))  low = middle + 1;  else  high = middle - 1;  }  return strs[0].substring(0, (low + high) / 2);  }  private boolean isCommonPrefix(String[] strs, int len){  String str1 = strs[0].substring(0,len);  for (int i = 1; i < strs.length; i++)  if (!strs[i].startsWith(str1))  return false;  return true;  } |

**Complexity Analysis**

In the worst case we have n*n* equal strings with length m*m*

* Time complexity :  *O*(*S*⋅log*m*), where S*S* is the sum of all characters in all strings.

The algorithm makes log*m* iterations, for each of them there are S =*m*⋅*n* comparisons, which gives in total  *O*(*S*⋅log*m*) time complexity.

* Space complexity : *O*(1). We only used constant extra space.

#### **Further Thoughts / Follow up**

Let's take a look at a slightly different problem:

Given a set of keys S = [*S*1​,*S*2​…*Sn*​], find the longest common prefix among a string q and S. This LCP query will be called frequently.

We could optimize LCP queries by storing the set of keys S in a Trie. For more information about Trie, please see this article [Implement a trie (Prefix trie)](https://leetcode.com/articles/implement-trie-prefix-tree/). In a Trie, each node descending from the root represents a common prefix of some keys. But we need to find the longest common prefix of a string q and all key strings. This means that we have to find the deepest path from the root, which satisfies the following conditions:

* it is prefix of query string q
* each node along the path must contain only one child element. Otherwise the found path will not be a common prefix among all strings.
* the path doesn't comprise of nodes which are marked as end of key. Otherwise the path couldn't be a prefix a of key which is shorter than itself.

**Algorithm**

The only question left, is how to find the deepest path in the Trie, that fulfills the requirements above. The most effective way is to build a trie from [*S*1​…*Sn*​] strings. Then find the prefix of query string **q** in the Trie. We traverse the Trie from the root, till it is impossible to continue the path in the Trie because one of the conditions above is not satisfied.

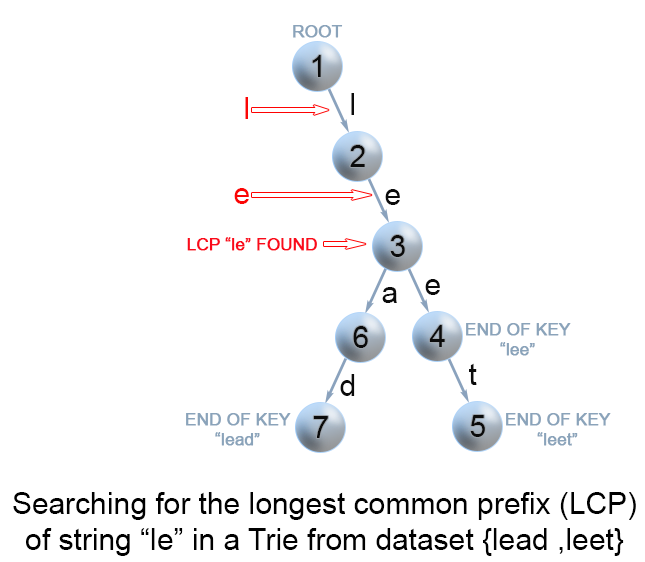


Figure 4. Finding the longest common prefix of strings using Trie

|  |
| --- |
| public String longestCommonPrefix(String q, String[] strs) {  if (strs == null || strs.length == 0)  return "";  if (strs.length == 1)  return strs[0];  Trie trie = new Trie();  for (int i = 1; i < strs.length ; i++) {  trie.insert(strs[i]);  }  return trie.searchLongestPrefix(q);  }  class TrieNode {  // R links to node children  private TrieNode[] links;  private final int R = 26;  private boolean isEnd;  // number of children non null links  private int size;  public void put(char ch, TrieNode node) {  links[ch -'a'] = node;  size++;  }  public int getLinks() {  return size;  }  //assume methods containsKey, isEnd, get, put are implemented as it is described  //in https://leetcode.com/articles/implement-trie-prefix-tree/)  }  public class Trie {  private TrieNode root;  public Trie() {  root = new TrieNode();  }  //assume methods insert, search, searchPrefix are implemented as it is described  //in https://leetcode.com/articles/implement-trie-prefix-tree/)  private String searchLongestPrefix(String word) {  TrieNode node = root;  StringBuilder prefix = new StringBuilder();  for (int i = 0; i < word.length(); i++) {  char curLetter = word.charAt(i);  if (node.containsKey(curLetter) && (node.getLinks() == 1) && (!node.isEnd())) {  prefix.append(curLetter);  node = node.get(curLetter);  }  else  return prefix.toString();  }  return prefix.toString();  }  } |

**Complexity Analysis**

In the worst case query *q* has length m*m* and it is equal to all *n* strings of the array.

* Time complexity : preprocessing *O*(*S*), where *S* is the number of all characters in the array, LCP query *O*(*m*).

Trie build has *O*(*S*) time complexity. To find the common prefix of *q* in the Trie takes in the worst case *O*(*m*).

* Space complexity : *O*(*S*). We only used additional *S* extra space for the Trie.

## **Two-Pointer Technique**

After finishing the previous chapter, we are now familiar with the concept of array and string and are able to perform basic operations in an array or a string.

By using these operations, we are able to solve some basic problems. However, it is obviously not enough.

In this chapter, we are going to talk about the two-pointer technique which can help us with many array/string related problems.

**Two-pointer Technique - Scenario I**

In the previous chapter, we solve some problems by iterating the array. Typically, we only use one pointer starting from the first element and ending at the last one to do iteration. However, sometimes, we might need to use two pointers at the same time to do the iteration.

 Let's start with a classic problem:

Reverse the elements in an array.

The idea is to swap the first element with the end, advance to the next element and swapping repeatedly until it reaches the middle position.

We can use two pointers at the same time to do the iteration: one starts from the first element and another starts from the last element. Continue swapping the elements until the two pointers meet each other.

Here is the code for your reference:

|  |
| --- |
| public static void reverse(int[] v, int N) {  int i = 0;  int j = N - 1;  while (i < j) {  swap(v, i, j); // this is a self-defined function  i++;  j--;  }  } |

*Summary*

To summarize, one of the typical scenarios to use two-pointer technique is that you want to

Iterate the array from two ends to the middle.

So you can use the two-pointer technique:

One pointer starts from the beginning while the other pointer starts from the end.

And it is worth noting that this technique is often used in a sorted array.

**Reverse String**

Write a function that reverses a string. The input string is given as an array of characters char[].

Do not allocate extra space for another array, you must do this by **modifying the input array**[**in-place**](https://en.wikipedia.org/wiki/In-place_algorithm) with O(1) extra memory.

You may assume all the characters consist of [printable ascii characters](https://en.wikipedia.org/wiki/ASCII#Printable_characters).

**Example 1:**

**Input:** ["h","e","l","l","o"]

**Output:** ["o","l","l","e","h"]

**Example 2:**

**Input:** ["H","a","n","n","a","h"]

**Output:** ["h","a","n","n","a","H"]

 Hint #1

The entire logic for reversing a string is based on using the opposite directional two-pointer approach!

## **Solution**

#### **Overview**

Life is short, use Python. (c)

|  |
| --- |
| class Solution:  def reverseString(self, s):  s.reverse() |

Speaking seriously, let's use this problem to discuss two things:

* Does *in-place* mean constant space complexity?
* Two pointers approach.

#### **Approach 1: Recursion, In-Place, O(N) Space**

**Does in-place mean constant space complexity?**

No. [By definition](https://en.wikipedia.org/wiki/In-place_algorithm), an in-place algorithm is an algorithm which transforms input using no auxiliary data structure.

The tricky part is that space is used by many actors, not only by data structures. The classical example is to use recursive function without any auxiliary data structures.

Is it in-place? Yes.

Is it constant space? No, because of recursion stack.



**Algorithm**

Here is an example. Let's implement recursive function helper which receives two pointers, left and right, as arguments.

* Base case: if left >= right, do nothing.
* Otherwise, swap s[left] and s[right] and call helper(left + 1, right - 1).

To solve the problem, call helper function passing the head and tail indexes as arguments: return helper(0, len(s) - 1).

**Implementation**

|  |
| --- |
| class Solution {  public void helper(char[] s, int left, int right) {  if (left >= right) return;  char tmp = s[left];  s[left++] = s[right];  s[right--] = tmp;  helper(s, left, right);  }  public void reverseString(char[] s) {  helper(s, 0, s.length - 1);  }  } |

**Complexity Analysis**

* Time complexity :  O(*N*) time to perform *N*/2 swaps.
* Space complexity :  O(*N*) to keep the recursion stack.

#### **Approach 2: Two Pointers, Iteration,  O(1) Space**

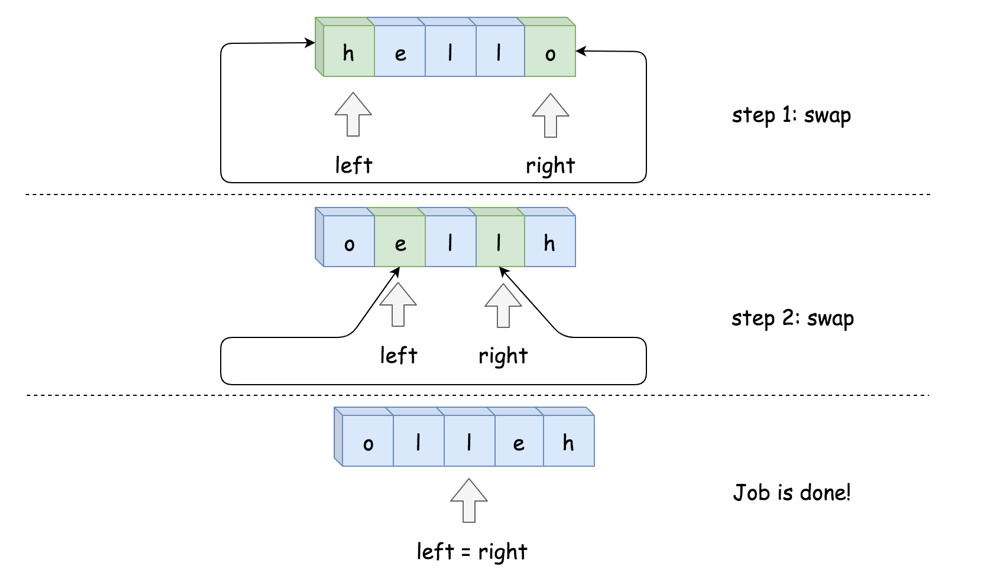
**Two Pointers Approach**

In this approach, two pointers are used to process two array elements at the same time. Usual implementation is to set one pointer in the beginning and one at the end and then to move them until they both meet.

Sometimes one needs to generalize this approach in order to use three pointers, like for classical [Sort Colors problem](https://leetcode.com/articles/sort-colors/).

**Algorithm**

* Set pointer left at index 0, and pointer right at index n - 1, where n is a number of elements in the array.
* While left < right:
  + Swap s[left] and s[right].
  + Move left pointer one step right, and right pointer one step left.



**Implementation**

|  |
| --- |
| class Solution {  public void reverseString(char[] s) {  int left = 0, right = s.length - 1;  while (left < right) {  char tmp = s[left];  s[left++] = s[right];  s[right--] = tmp;  }  }  } |

**Complexity Analysis**

* Time complexity :  O(*N*) to swap *N*/2 element.
* Space complexity :  O(1), it's a constant space solution.

**Array Partition I**

Given an integer array nums of 2n integers, group these integers into n pairs (a1, b1), (a2, b2), ..., (an, bn) such that the sum of min(ai, bi) for all i is **maximized**. Return*the maximized sum*.

**Example 1:**

**Input:** nums = [1,4,3,2]

**Output:** 4

**Explanation:** All possible pairings (ignoring the ordering of elements) are:

1. (1, 4), (2, 3) -> min(1, 4) + min(2, 3) = 1 + 2 = 3

2. (1, 3), (2, 4) -> min(1, 3) + min(2, 4) = 1 + 2 = 3

3. (1, 2), (3, 4) -> min(1, 2) + min(3, 4) = 1 + 3 = 4

So the maximum possible sum is 4.

**Example 2:**

**Input:** nums = [6,2,6,5,1,2]

**Output:** 9

**Explanation:** The optimal pairing is (2, 1), (2, 5), (6, 6). min(2, 1) + min(2, 5) + min(6, 6) = 1 + 2 + 6 = 9.

**Constraints:**

* 1 <= n <= 104
* nums.length == 2 \* n
* -104 <= nums[i] <= 104

Hint #1

Obviously, brute force won't help here. Think of something else, take some example like 1,2,3,4.

Hint #2

How will you make pairs to get the result? There must be some pattern.

Hint #3

Did you observe that- Minimum element gets add into the result in sacrifice of maximum element.

Hint #4

Still won't able to find pairs? Sort the array and try to find the pattern.

## **Solution**

#### **Approach #1 Brute Force [Time Limit Exceeded]**

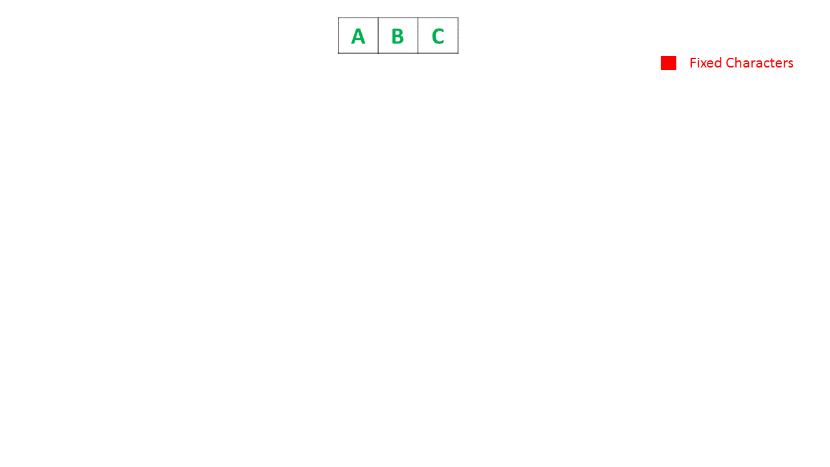
**Algorithm**

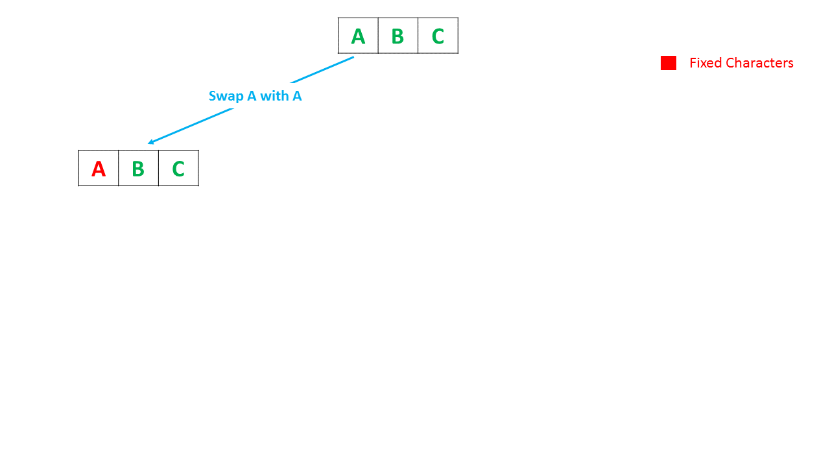
The simplest solution is to consider every possible set of pairings possible by using the elements of the nums*nums* array. For generating all the possible pairings, we make use of a function permute(nums, current\_index). This function creates all the possible permutations of the elements of the given array.

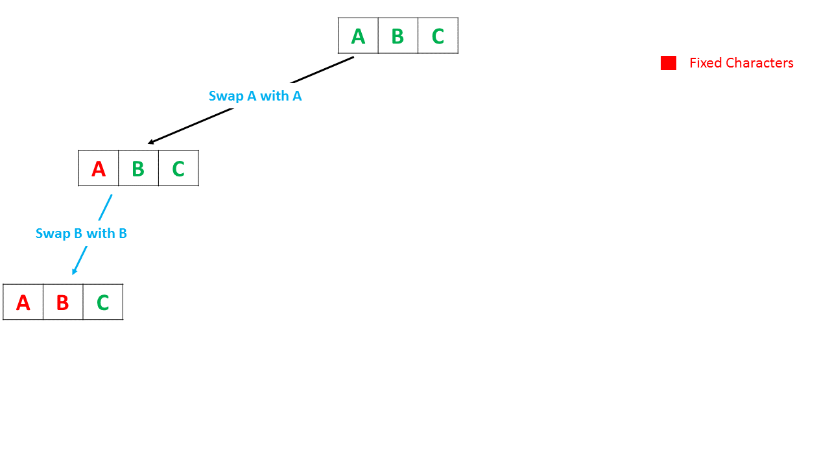
To do so, permute takes the index of the current element current\_index as one of the arguments. Then, it swaps the current element with every other element in the array, lying towards its right, so as to generate a new ordering of the array elements. After the swapping has been done, it makes another call to permute but this time with the index of the next element in the array. While returning back, we reverse the swapping done in the current function call.

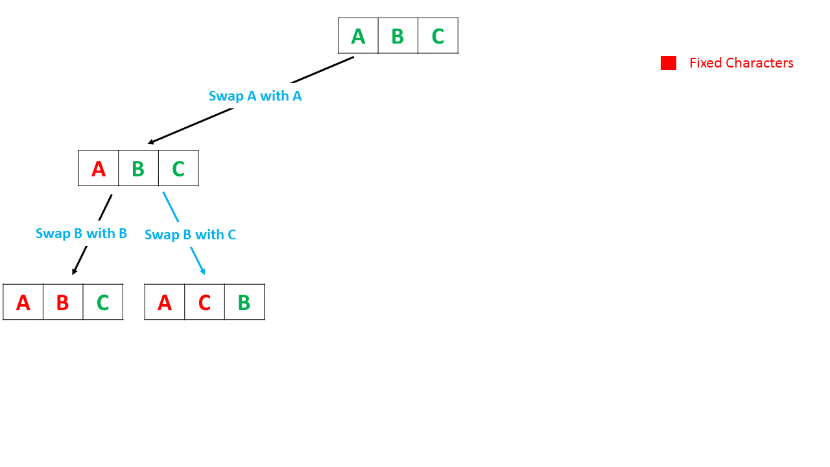
Thus, when we reach the end of the array, a new ordering of the array's elements is generated. We consider the elements to be taken for the pairings such that the first element of every pair comes from the first half of the new array and the second element comes from the last half of the array. Thus, we sum up the minimum elements out of all these possible pairings and find out the maximum sum out of them.

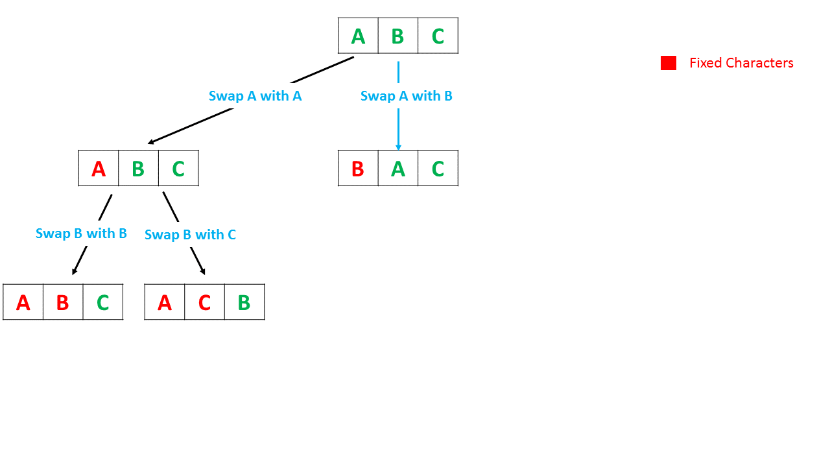
The animation below depicts the ways the permutations are generated.

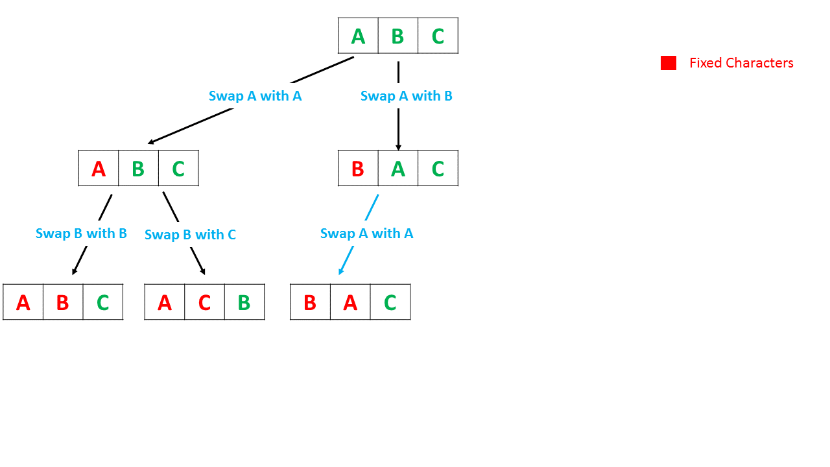


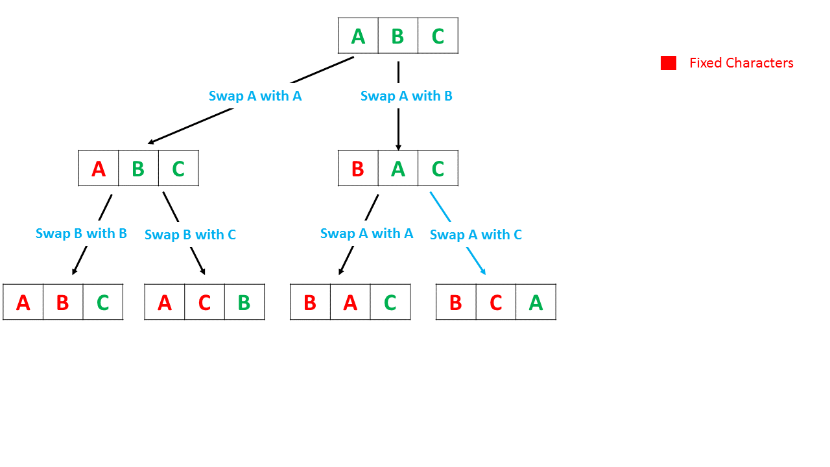




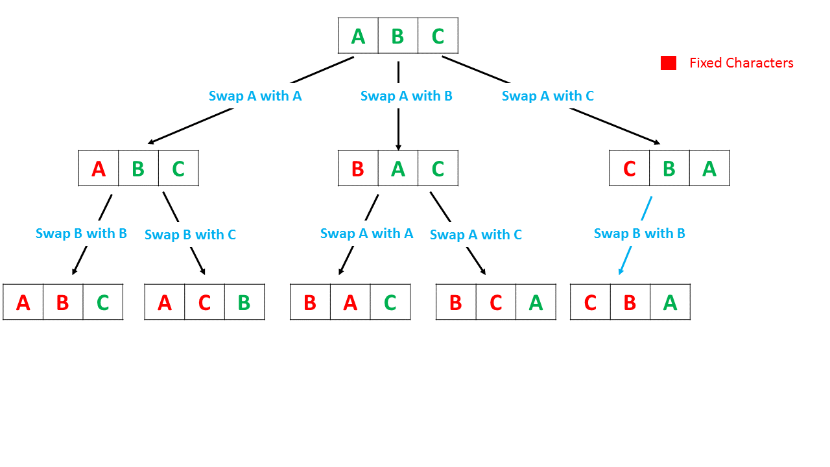


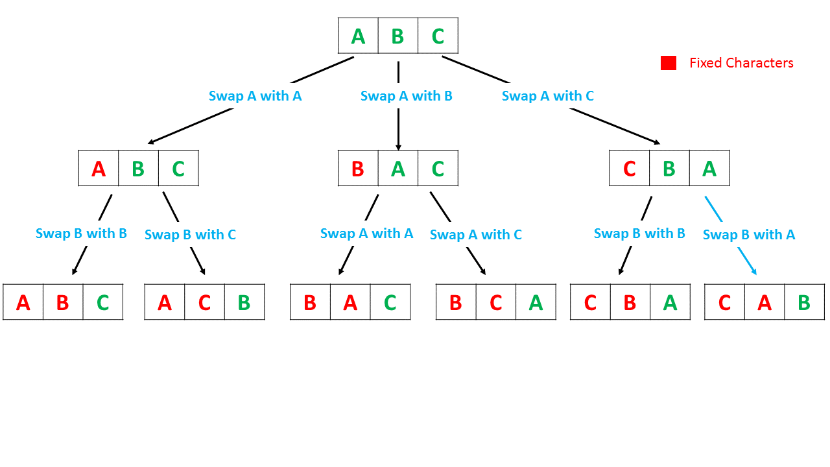














|  |
| --- |
| public class Solution {  int max\_sum = Integer.MIN\_VALUE;  public int arrayPairSum(int[] nums) {  permute(nums, 0);  return max\_sum;  }  public void permute(int[] nums, int l) {  if (l == nums.length - 1) {  int sum = 0;  for (int i = 0; i < nums.length / 2; i++) {  sum += Math.min(nums[i], nums[nums.length / 2 + i]);  }  max\_sum = Math.max(max\_sum, sum);  }  for (int i = l; i < nums.length; i++) {  swap(nums, i, l);  permute(nums, l + 1);  swap(nums, i, l);  }  }  public void swap(int[] nums, int x, int y) {  int temp = nums[x];  nums[x] = nums[y];  nums[y] = temp;  }  } |

**Complexity Analysis**

* Time complexity : *O*(*n*!). A total of *n*! permutations are possible for *n* elements in the array.
* Space complexity : *O*(*N*). We do not allocate any additional data structures but use *O*(*N*) for the recursion stack.

#### **Approach #2 Using Sorting [Accepted]**

**Algorithm**

In order to understand this approach, let us look at the problem from a different perspective. We need to form the pairings of the array's elements such that the overall sum of the minimum out of such pairings is maximum. Thus, we can look at the operation of choosing the minimum out of the pairing, say (*a*,*b*) as incurring a loss of *a*−*b*(if *a*>*b*), in the maximum sum possible.

The total sum will now be maximum if the overall loss incurred from such pairings is minimized. This minimization of loss in every pairing is possible only if the numbers chosen for the pairings lie closer to each other than to the other elements of the array.

Taking this into consideration, we can sort the elements of the given array and form the pairings of the elements directly in the sorted order. This will lead to the pairings of elements with minimum difference between them leading to the maximization of the required sum.

|  |
| --- |
| public class Solution {  public int arrayPairSum(int[] nums) {  Arrays.sort(nums);  int sum = 0;  for (int i = 0; i < nums.length; i += 2) {  sum += nums[i];  }  return sum;  }  } |

**Complexity Analysis**

* Time complexity :  O(*N*log*N*). Sorting takes O(*N*log*N*) time. We iterate over the array only once.
* Space complexity :  O(*N*) or O(log*N*)
  + The space complexity of the sorting algorithm depends on the implementation of each program language.
  + For instance, the list.sort() function in Python is implemented with the [Timsort](https://en.wikipedia.org/wiki/Timsort) algorithm whose space complexity is  O(*N*).
  + In Java, the [Arrays.sort()](https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html" \l "sort-byte:A-) is implemented as a variant of quicksort algorithm whose space complexity is O(log*N*).

#### **Approach #3 Using Extra Array [Accepted]**

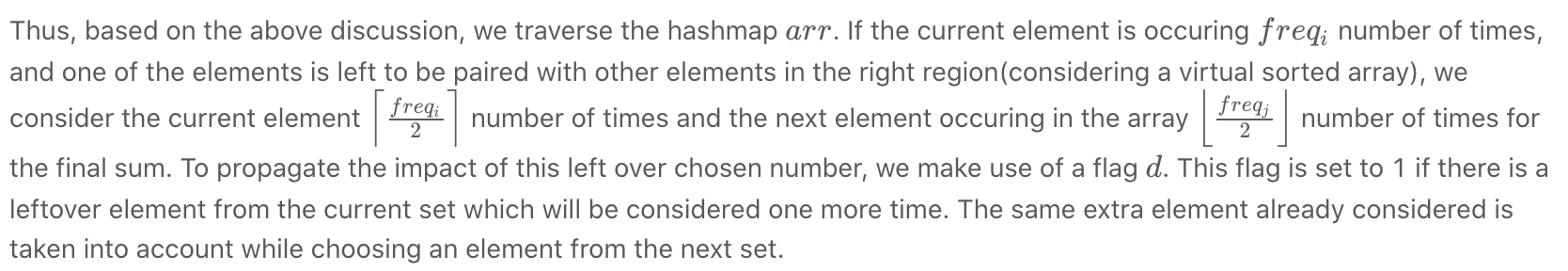
**Algorithm**

This approach is somewhat related to the sorting approach. Since the range of elements in the given array is limited, we can make use of a HashMap ***arr***, such that ***arr***[*i*] stores the frequency of occurrence of (i-10000)^th element. This subtraction is done so as to be able to map the numbers in the range -10000 ≤ i ≤ -1 onto the HashMap.

Thus, now instead of sorting the array's elements, we can directly traverse the HashMap in an ascending order. But, any element could also occur multiple times in the given array. We need to take this factor into account.

For this, consider an example: nums: [a, b, a, b, b, a]. The sorted order of this array will be nums\_sorted: [a, a, a, b, b, b]. (We aren't actually sorting the array in this approach, but the sorted array is taken just for demonstration). From the previous approach, we know that the required set of pairings is (*a*,*a*),(*a*,*b*),(*b*,*b*). Now, we can see that while choosing the minimum elements, a*a* will be chosen twice and b*b* will be chosen once only. This happens because the number of *a*'s to be chosen has already been determined by the frequency of a*a*, leaving the rest of the places to be filled by b*b*. This is because, for the correct result we need to consider the elements in the ascending order. Thus, the lower number always gets priority to be added to the end result.

But, if the sorted elements take the form: nums\_sorted: [a, a, b, b, b, b], the correct pairing will be (*a*,*a*),(*b*,*b*),(*b*,*b*). Again, in this case the number of *a*'s chosen is already predetermined, but since the number of *a*'s is odd, it doesn't impact the choice of b*b* in the final sum.



While traversing the HashMap, we determine the correct number of times each element needs to be considered as discussed above. Note that the flag d*d* and the *sum* remains unchanged if the current element of the HashMap doesn't exist in the array.

Below code is inspired by [@fallcreek](https://leetcode.com/fallcreek)

|  |
| --- |
| public class Solution {  public int arrayPairSum(int[] nums) {  int[] arr = new int[20001];  int lim = 10000;  for (int num: nums)  arr[num + lim]++;  int d = 0, sum = 0;  for (int i = -10000; i <= 10000; i++) {  sum += (arr[i + lim] + 1 - d) / 2 \* i;  d = (2 + arr[i + lim] - d) % 2;  }  return sum;  }  } |

**Complexity Analysis**

* Time complexity : *O*(*n*). The whole HashMap *arr* of size *n* is traversed only once.
* Space complexity : *O*(*n*). A HashMap *arr* of size *n* is used.

**Two Sum II - Input array is sorted**

Given an array of integers that is already ***sorted in ascending order***, find two numbers such that they add up to a specific target number.

The function twoSum should return indices of the two numbers such that they add up to the target, where index1 must be less than index2.

**Note:**

* Your returned answers (both index1 and index2) are not zero-based.
* You may assume that each input would have *exactly* one solution and you may not use the *same* element twice.

**Example 1:**

**Input:** numbers = [2,7,11,15], target = 9

**Output:** [1,2]

**Explanation:** The sum of 2 and 7 is 9. Therefore index1 = 1, index2 = 2.

**Example 2:**

**Input:** numbers = [2,3,4], target = 6

**Output:** [1,3]

**Example 3:**

**Input:** numbers = [-1,0], target = -1

**Output:** [1,2]

**Constraints:**

* 2 <= nums.length <= 3 \* 104
* -1000 <= nums[i] <= 1000
* nums is sorted in **increasing order**.
* -1000 <= target <= 1000

## **Solution**

#### **Approach 1: Two Pointers**

**Algorithm**

We can apply [Two Sum's solutions](https://leetcode.com/articles/two-sum/) directly to get O(n^2) time, *O*(1) space using brute force and *O*(*n*) time, *O*(*n*) space using hash table. However, both existing solutions do not make use of the property where the input array is sorted. We can do better.

We use two indexes, initially pointing to the first and last element respectively. Compare the sum of these two elements with target. If the sum is equal to target, we found the exactly only solution. If it is less than target, we increase the smaller index by one. If it is greater than target, we decrease the larger index by one. Move the indexes and repeat the comparison until the solution is found.

Let [... , a, b, c, ... , d, e, f, ...] be the input array that is sorted in ascending order and the element *b*, *e* be the exactly only solution. Because we are moving the smaller index from left to right, and the larger index from right to left, at some point one of the indexes must reach either one of *b* or *e*. Without loss of generality, suppose the smaller index reaches *b* first. At this time, the sum of these two elements must be greater than target. Based on our algorithm, we will keep moving the larger index to its left until we reach the solution.

|  |
| --- |
| class Solution {  public:  vector<int> twoSum(vector<int>& numbers, int target) {  int low = 0, high = numbers.size() - 1;  while (low < high) {  int sum = numbers[low] + numbers[high];  if (sum == target)  return {low + 1, high + 1};  else if (sum < target)  ++low;  else  --high;  }  return {-1, -1};  }  }; |

Do we need to consider if *numbers*[*low*]+*numbers*[*high*] overflows? The answer is no. Even if adding two elements in the array may overflow, because there is exactly one solution, we will reach the solution first.

**Complexity analysis**

* Time complexity : *O*(*n*). Each of the *n* elements is visited at most once, thus the time complexity is *O*(*n*).
* Space complexity : *O*(1). We only use two indexes, the space complexity is *O*(1).

**Two-pointer Technique - Scenario II**

Sometimes, we can use two pointers with different steps to solve problems.

Let's start with another classic problem:

Given an array and a value, remove all instances of that value [**in-place**](https://en.wikipedia.org/wiki/In-place_algorithm) and return the new length.

If we don't have the limitation of space complexity, it will be easier. We can initialize a new array to store the answer. Iterate the original array and add the element to the new array if the element is not equal to the given target value.

Actually, it is equivalent to using two pointers, one is used for the iteration of the original array and another one always points at the last position of the new array.

*Reconsider the Space Limitation*

Now let's reconsider the space limitation.

We can use a similar strategy. We still use two pointers: one is still used for the iteration while the second one always points at the position for next addition.

Here is the code for your reference:

|  |
| --- |
| public int removeElement(int[] nums, int val) {  int k = 0;  for (int i = 0; i < nums.length; ++i) {  if (nums[i] != val) {  nums[k] = nums[i];  k++;  }  }  return k;  } |

We use two pointers, one faster-runner i and one slower-runner k, in the example above. i moves one step each time while k moves one step only if a new needed value is added.

*Summary*

This is a very common scenario of using the two-pointer technique when you need:

One slow-runner and one fast-runner at the same time.

The key to solving this kind of problems is to

Determine the movement strategy for both pointers.

Similar to the previous scenario, you might sometimes need to sort the array before using the two-pointer technique. And you might need a greedy thought to determine your movement strategy.

**Remove Element**

Given an array *nums* and a value val, remove all instances of that value [**in-place**](https://en.wikipedia.org/wiki/In-place_algorithm) and return the new length.

Do not allocate extra space for another array, you must do this by **modifying the input array**[**in-place**](https://en.wikipedia.org/wiki/In-place_algorithm) with O(1) extra memory.

The order of elements can be changed. It doesn't matter what you leave beyond the new length.

**Clarification:**

Confused why the returned value is an integer but your answer is an array?

Note that the input array is passed in by **reference**, which means a modification to the input array will be known to the caller as well.

Internally you can think of this:

// **nums** is passed in by reference. (i.e., without making a copy)

int len = removeElement(nums, val);

// any modification to **nums** in your function would be known by the caller.

// using the length returned by your function, it prints the first **len** elements.

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

    print(nums[i]);

}

**Example 1:**

**Input:** nums = [3,2,2,3], val = 3

**Output:** 2, nums = [2,2]

**Explanation:** Your function should return length = **2**, with the first two elements of *nums* being **2**.

It doesn't matter what you leave beyond the returned length. For example if you return 2 with nums = [2,2,3,3] or nums = [2,2,0,0], your answer will be accepted.

**Example 2:**

**Input:** nums = [0,1,2,2,3,0,4,2], val = 2

**Output:** 5, nums = [0,1,4,0,3]

**Explanation:** Your function should return length = **5**, with the first five elements of *nums* containing **0**, **1**, **3**, **0**, and **4**. Note that the order of those five elements can be arbitrary. It doesn't matter what values are set beyond the returned length.

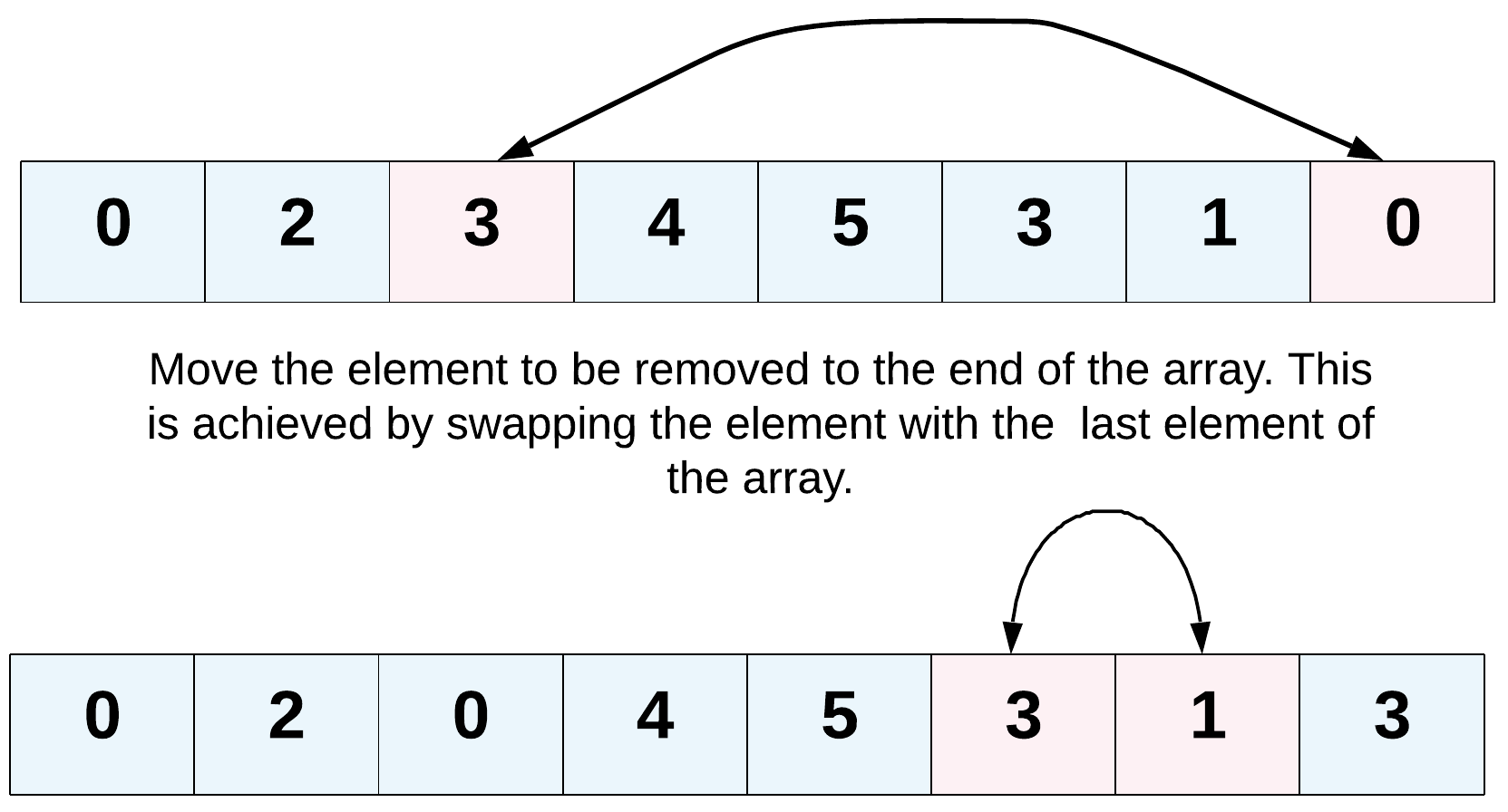
**Constraints:**

* 0 <= nums.length <= 100
* 0 <= nums[i] <= 50
* 0 <= val <= 100

Hint #1

The problem statement clearly asks us to modify the array in-place and it also says that the element beyond the new length of the array can be anything. Given an element, we need to remove all the occurrences of it from the array. We don't technically need to **remove** that element per-say, right?

Hint #2

We can move all the occurrences of this element to the end of the array. Use two pointers!  


Hint #3

Yet another direction of thought is to consider the elements to be removed as non-existent. In a single pass, if we keep copying the visible elements in-place, that should also solve this problem for us.

## **Summary**

This is a pretty easy problem, but one may get confused by the term "in-place" and think it is impossible to remove an element from the array without making a copy of the array.

## **Hints**

1. Try two pointers.
2. Did you use the fact that the order of elements can be changed?
3. What happens when the elements to remove are rare?

## **Solution**

#### **Approach 1: Two Pointers**

**Intuition**

Since this question is asking us to remove all elements of the given value in-place, we have to handle it with *O*(1) extra space. How to solve it? We can keep two pointers i*i* and *j*, where i*i* is the slow-runner while *j* is the fast-runner.

**Algorithm**

When *nums*[*j*] equals to the given value, skip this element by incrementing *j*. As long as  [*j*]!=*val*, we copy *nums*[*j*] to *nums*[*i*] and increment both indexes at the same time. Repeat the process until *j* reaches the end of the array and the new length is *i*.

This solution is very similar to the solution to [Remove Duplicates from Sorted Array](https://leetcode.com/articles/remove-duplicates-from-sorted-array/).

|  |
| --- |
| public int removeElement(int[] nums, int val) {  int i = 0;  for (int j = 0; j < nums.length; j++) {  if (nums[j] != val) {  nums[i] = nums[j];  i++;  }  }  return i;  } |

**Complexity analysis**

* Time complexity : *O*(*n*). Assume the array has a total of *n* elements, both *i* and *j* traverse at most 2*n* steps.
* Space complexity : *O*(1).

#### **Approach 2: Two Pointers - when elements to remove are rare**

**Intuition**

Now consider cases where the array contains few elements to remove. For example, nums=[1,2,3,5,4],*val*=4. The previous algorithm will do unnecessary copy operation of the first four elements. Another example is nums = [4,1,2,3,5], val = 4. It seems unnecessary to move elements [1,2,3,5] one step left as the problem description mentions that the order of elements could be changed.

**Algorithm**

When we encounter *nums*[*i*]=*val*, we can swap the current element out with the last element and dispose the last one. This essentially reduces the array's size by 1.

Note that the last element that was swapped in could be the value you want to remove itself. But don't worry, in the next iteration we will still check this element.

|  |
| --- |
| public int removeElement(int[] nums, int val) {  int i = 0;  int n = nums.length;  while (i < n) {  if (nums[i] == val) {  nums[i] = nums[n - 1];  // reduce array size by one  n--;  } else {  i++;  }  }  return n;  } |

**Complexity analysis**

* Time complexity : *O*(*n*). Both i*i* and *n* traverse at most *n* steps. In this approach, the number of assignment operations is equal to the number of elements to remove. So it is more efficient if elements to remove are rare.
* Space complexity : *O*(1).

**Max Consecutive Ones**

Given a binary array, find the maximum number of consecutive 1s in this array.

**Example 1:**

**Input:** [1,1,0,1,1,1]

**Output:** 3

**Explanation:** The first two digits or the last three digits are consecutive 1s.

The maximum number of consecutive 1s is 3.

**Note:**

* The input array will only contain 0 and 1.
* The length of input array is a positive integer and will not exceed 10,000

Hint #1

You need to think about two things as far as any window is concerned. One is the starting point for the window. How do you detect that a new window of 1s has started? The next part is detecting the ending point for this window. How do you detect the ending point for an existing window? If you figure these two things out, you will be able to detect the windows of consecutive ones. All that remains afterward is to find the longest such window and return the size.

## **Solution**

The constraints for this problem make it easy to understand that it can be done in one iteration.

The length of input array is a positive integer and will not exceed 10,000

How else do you expect to find the number of 1's in an array without making at least one pass through the array. So let's look at the solution.

#### **Approach: One pass**

**Intuition**

The intuition is pretty simple. We keep a count of the number of 1's encountered. And reset the count whenever we encounter anything other than 1 (which is 0 for this problem). Thus, maintaining count of 1's between zeros or rather maintaining counts of contiguous 1's. It's the same as keeping a track of the number of hours of sleep you had, without waking up in between.

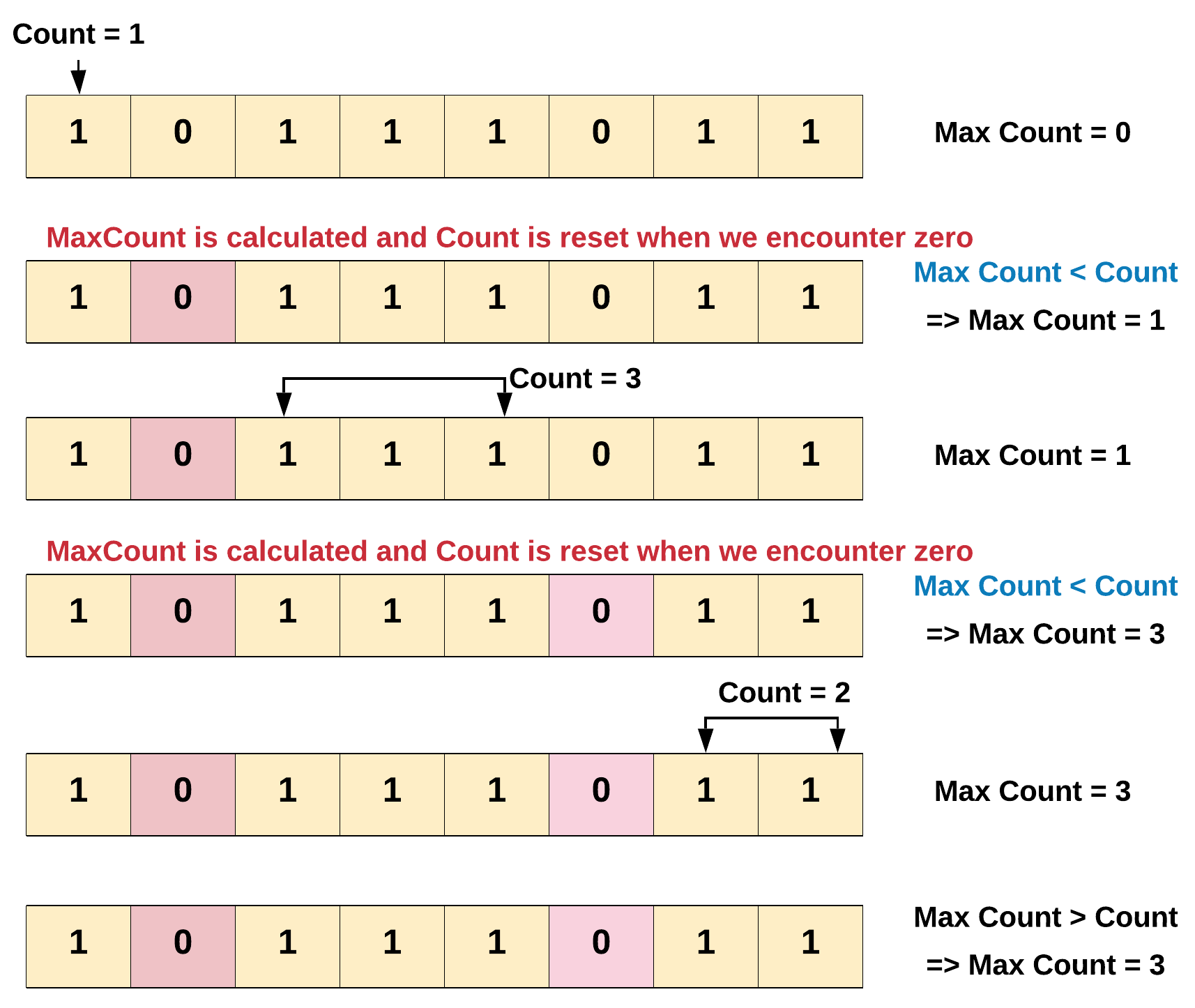
**Algorithm**

1. Maintain a counter for the number of 1's.
2. Increment the counter by 1, whenever you encounter a 1.
3. Whenever you encounter a 0

a. Use the current count of 1's to find the maximum contiguous 1's till now.

b. Afterwards, reset the counter for 1's to 0.

1. Return the maximum in the end.



In the above diagram we found out that the maximum number of consecutive 1's is 3. There were two breaks in the count we encountered while iterating the array. Every time the break i.e. 0 was encountered we had to reset the count of 1 to zero.

Note - The maximum count is only calculated when we encounter a break i.e. 0, since thats where a subarray of 1's ends.

|  |
| --- |
| class Solution {  public int findMaxConsecutiveOnes(int[] nums) {  int count = 0;  int maxCount = 0;  for(int i = 0; i < nums.length; i++) {  if(nums[i] == 1) {  // Increment the count of 1's by one.  count += 1;  } else {  // Find the maximum till now.  maxCount = Math.max(maxCount, count);  // Reset count of 1.  count = 0;  }  }  return Math.max(maxCount, count);  }  } |

**Complexity Analysis**

* Time Complexity: *O*(*N*), where *N* is the number of elements in the array.
* Space Complexity: *O*(1). We do not use any extra space.

#### **Follow up:**

You can also try something fancy one liner [solution](https://leetcode.com/articles/duplicate-zeros/) as shared by [Stefan Pochmann](https://leetcode.com/stefanpochmann/).

def findMaxConsecutiveOnes(self, nums):

return max(map(len, ''.join(map(str, nums)).split('0')))

Note, how he converts the list into a string, using map and join functions. Then, splits the resultant string on 0. The maximum of the lengths of these list of strings of 1's is the answer we are looking for.

**Minimum Size Subarray Sum**

Given an array of **n** positive integers and a positive integer **s**, find the minimal length of a **contiguous** subarray of which the sum ≥ **s**. If there isn't one, return 0 instead.

**Example:**

**Input:** s = 7, nums = [2,3,1,2,4,3]

**Output:** 2

**Explanation:** the subarray [4,3] has the minimal length under the problem constraint.

**Follow up:**

If you have figured out the *O*(*n*) solution, try coding another solution of which the time complexity is *O*(*n* log *n*).

#### **Approach #1 Brute force [Time Limit Exceeded]**

**Intuition**

Do as directed in question. Find the sum for all the possible subarrays and update the ans as and when we get a better subarray that fulfill the requirements (sum≥s).

**Algorithm**

* Initialize {ans}={INT\_MAX}
* Iterate the array from left to right using i*i*:
  + Iterate from the current element to the end of vector using *j*:
    - Find the sum of elements from index *i* to *j*
    - If sum is greater then *s*:
      * Update ans = ans=min(ans,(*j*−*i*+1))
      * Start the next *i*th iteration, since, we got the smallest subarray with sum≥*s* starting from the current index.

|  |
| --- |
| int minSubArrayLen(int s, vector<int>& nums)  {  int n = nums.size();  int ans = INT\_MAX;  for (int i = 0; i < n; i++) {  for (int j = i; j < n; j++) {  int sum = 0;  for (int k = i; k <= j; k++) {  sum += nums[k];  }  if (sum >= s) {  ans = min(ans, (j - i + 1));  break; //Found the smallest subarray with sum>=s starting with index i, hence move to next index  }  }  }  return (ans != INT\_MAX) ? ans : 0;  } |

**Complexity Analysis**

* Time complexity: O(n^3).
  + For each element of array, we find all the subarrays starting from that index which is O(n2).
  + Time complexity to find the sum of each subarray is O(n).
  + Thus, the total time complexity : O(n2 \* n) = *O*(*n*2∗*n*)=*O*(*n*3)
* Space complexity: *O*(1) extra space.

#### **Approach #2 A better brute force [Accepted]**

**Intuition**

In Approach #1, you may notice that the sum is calculated for every surarray in *O*(*n*) time. But, we could easily find the sum in O(1) time by storing the cumulative sum from the beginning(Memoization). After we have stored the cumulative sum in sums, we could easily find the sum of any subarray from *i* to *j*.

**Algorithm**

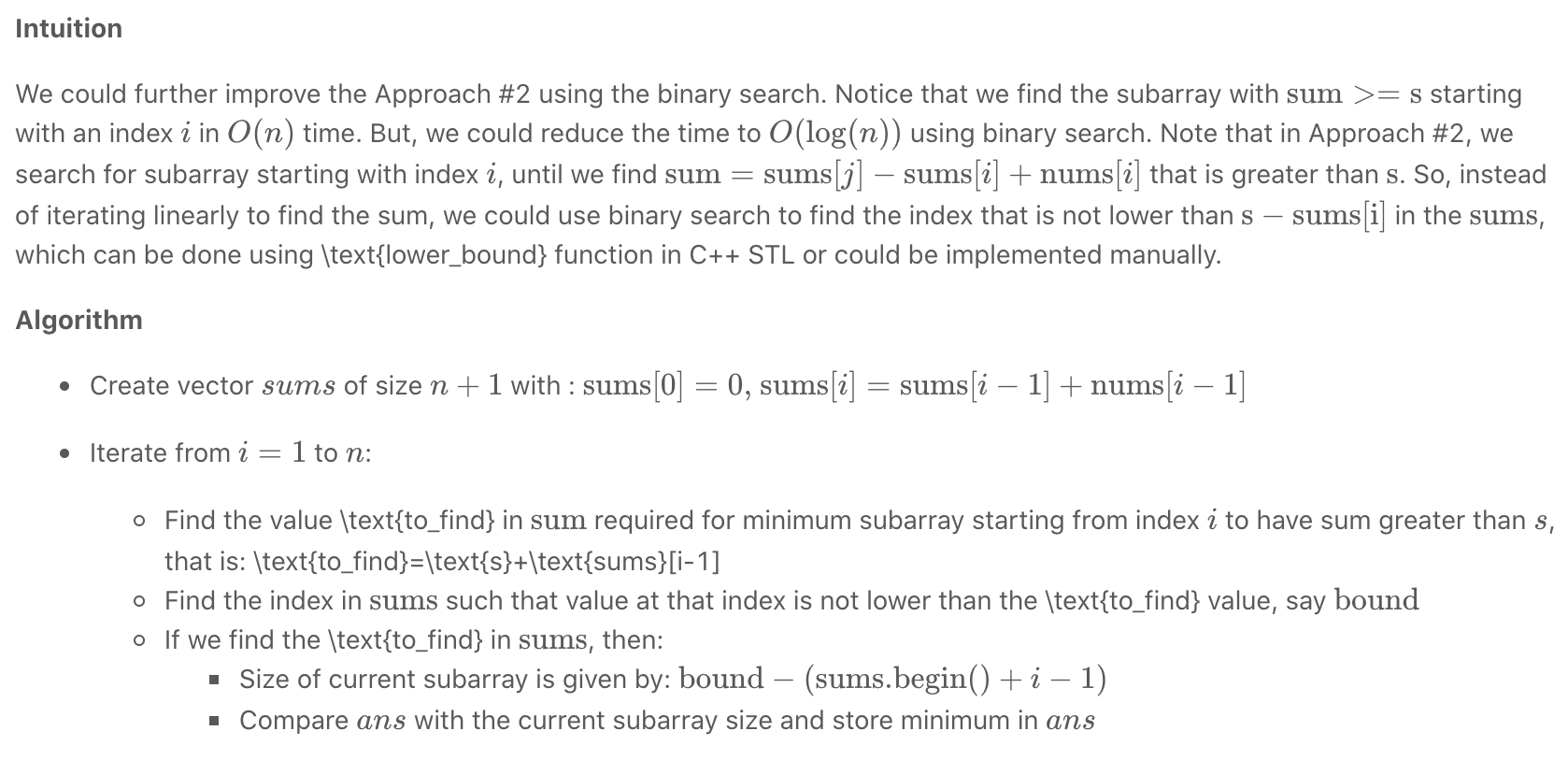
* The algorithm is similar to Approach #1.
* The only difference is in the way of finding the sum of subarrays:
  + Create a vector **sums** of size of **nums**
  + Initialize **sums[0]=nums[0]**
  + Iterate over the sums vector:
    - Update **sums[*i*]=sums[*i*−1]+nums[*i*]**
  + Sum of subarray from ***i***to ***j*** is calculated as:  **sum=sums[*j*]−sums[*i*]+nums[*i*]**, wherein **sums[*j*]−sums[*i*]** is the sum from (*i*+1)th element to the *j*th element.

|  |
| --- |
| int minSubArrayLen(int s, vector<int>& nums)  {  int n = nums.size();  if (n == 0)  return 0;  int ans = INT\_MAX;  vector<int> sums(n);  sums[0] = nums[0];  for (int i = 1; i < n; i++)  sums[i] = sums[i - 1] + nums[i];  for (int i = 0; i < n; i++) {  for (int j = i; j < n; j++) {  int sum = sums[j] - sums[i] + nums[i];  if (sum >= s) {  ans = min(ans, (j - i + 1));  break; //Found the smallest subarray with sum>=s starting with index i, hence move to next index  }  }  }  return (ans != INT\_MAX) ? ans : 0;  } |

**Complexity analysis**

* Time complexity: O(n2).
  + Time complexity to find all the subarrays is O(n2).
  + Sum of the subarrays is calculated in *O*(1) time.
  + Thus, the total time complexity: *O*(*n*2∗1)=*O*(*n*2)
* Space complexity: *O*(*n*) extra space.
  + Additional *O*(*n*) space for sums vector than in Approach #1.

#### **Approach #3 Using Binary search [Accepted]**



|  |
| --- |
| int minSubArrayLen(int s, vector<int>& nums)  {  int n = nums.size();  if (n == 0)  return 0;  int ans = INT\_MAX;  vector<int> sums(n + 1, 0); //size = n+1 for easier calculations  //sums[0]=0 : Meaning that it is the sum of first 0 elements  //sums[1]=A[0] : Sum of first 1 elements  //ans so on...  for (int i = 1; i <= n; i++)  sums[i] = sums[i - 1] + nums[i - 1];  for (int i = 1; i <= n; i++) {  int to\_find = s + sums[i - 1];  auto bound = lower\_bound(sums.begin(), sums.end(), to\_find);  if (bound != sums.end()) {  ans = min(ans, static\_cast<int>(bound - (sums.begin() + i - 1)));  }  }  return (ans != INT\_MAX) ? ans : 0;  } |

**Complexity analysis**

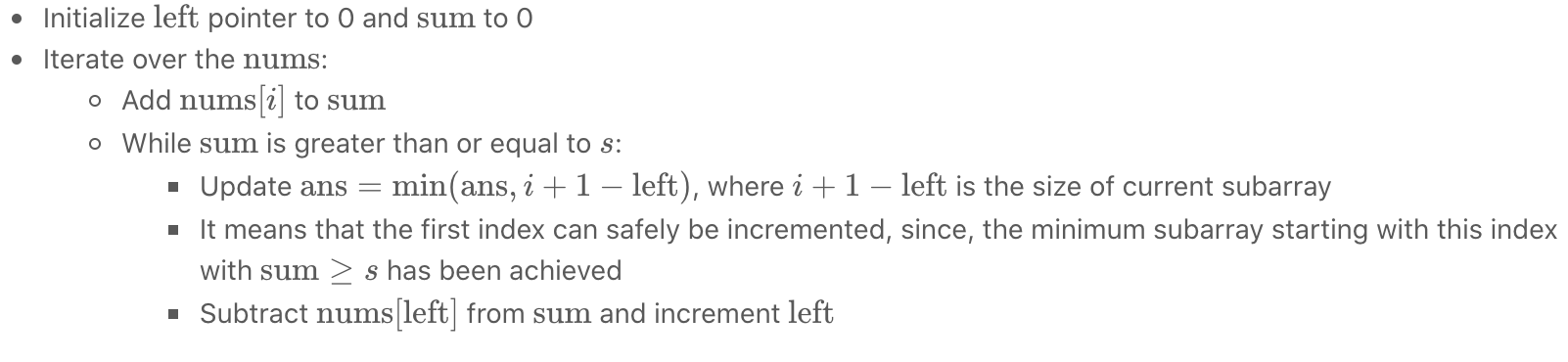
* Time complexity: *O*(*n*log(*n*)).
  + For each element in the vector, find the subarray starting from that index, and having sum greater than s*s* using binary search. Hence, the time required is *O*(*n*) for iteration over the vector and *O*(log(*n*)) for finding the subarray for each index using binary search.
  + Therefore, total time complexity = *O*(*n*∗log(*n*))
* Space complexity: *O*(*n*). Additional *O*(*n*) space for sums vector

#### **Approach #4 Using 2 pointers [Accepted]**

**Intuition**

Until now, we have kept the starting index of subarray fixed, and found the last position. Instead, we could move the starting index of the current subarray as soon as we know that no better could be done with this index as the starting index. We could keep 2 pointer, one for the start and another for the end of the current subarray, and make optimal moves so as to keep the \text{sum}sum greater than s*s* as well as maintain the lowest size possible.

**Algorithm**



|  |
| --- |
| int minSubArrayLen(int s, vector<int>& nums)  {  int n = nums.size();  int ans = INT\_MAX;  int left = 0;  int sum = 0;  for (int i = 0; i < n; i++) {  sum += nums[i];  while (sum >= s) {  ans = min(ans, i + 1 - left);  sum -= nums[left++];  }  }  return (ans != INT\_MAX) ? ans : 0;  } |

**Complexity analysis**

* Time complexity: *O*(*n*). Single iteration of *O*(*n*).
  + Each element can be visited atmost twice, once by the right pointer(*i*) and (atmost)once by the \text{left}left pointer.
* Space complexity: *O*(1) extra space. Only constant space required for left,  sum, ans and *i*.

## **Conclusion**

In this chapter, we provide a list of more array-related data structures or techniques you might want to know in the future. We will keep publishing more cards and update the links in this article to help you conquer these topics one by one.

We also provide some array/string related exercise for you. Try to solve these problems by yourself.

**Array-related Techniques**

There are more array-related data structures or techniques you might want to know. We will not go deeper into most of the concepts in this card but provide the links to the corresponding card in this article.

1. There are some other data structures which are similar to the array but have some different properties:

* [String](https://leetcode.com/explore/learn/card/array-and-string/203/introduction-to-string/) (has been introduced in this card)
* [Hash Table](https://leetcode.com/explore/learn/card/hash-table/)
* [Linked List](https://leetcode.com/explore/learn/card/linked-list/)
* Queue
* Stack

2. As we mentioned, we can call the built-in function to sort an array. But it is useful to understand the principle of some widely-used sorting algorithms and their complexity.

3. [Binary search](https://leetcode.com/explore/learn/card/binary-search/) is also an important technique used to search a specific element in a sorted array.

4. We have introduced two-pointer technique in this chapter. It is not easy to use this technique flexibly. This technique can also be used to solve:

* [Slow-pointer and fast-pointer problem in Linked List](https://leetcode.com/explore/learn/card/linked-list/214/linked-list-two-pointer/)
* Sliding Window Problem

5. The two-pointer technique sometimes will relate to Greedy Algorithm which helps us design our pointers' movement strategy.

We will come up with more cards to introduce these techniques mentioned above and update the link in the near future.

**Rotate Array**

Given an array, rotate the array to the right by *k* steps, where *k* is non-negative.

**Follow up:**

* Try to come up as many solutions as you can, there are at least 3 different ways to solve this problem.
* Could you do it in-place with O(1) extra space?

**Example 1:**

**Input:** nums = [1,2,3,4,5,6,7], k = 3

**Output:** [5,6,7,1,2,3,4]

**Explanation:**

rotate 1 steps to the right: [7,1,2,3,4,5,6]

rotate 2 steps to the right: [6,7,1,2,3,4,5]

rotate 3 steps to the right: [5,6,7,1,2,3,4]

**Example 2:**

**Input:** nums = [-1,-100,3,99], k = 2

**Output:** [3,99,-1,-100]

**Explanation:**

rotate 1 steps to the right: [99,-1,-100,3]

rotate 2 steps to the right: [3,99,-1,-100]

**Constraints:**

* 1 <= nums.length <= 2 \* 104
* -231 <= nums[i] <= 231 - 1
* 0 <= k <= 105

Hint #1

The easiest solution would use additional memory and that is perfectly fine.

Hint #2

The actual trick comes when trying to solve this problem without using any additional memory. This means you need to use the original array somehow to move the elements around. Now, we can place each element in its original location and shift all the elements around it to adjust as that would be too costly and most likely will time out on larger input arrays.

Hint #3

One line of thought is based on reversing the array (or parts of it) to obtain the desired result. Think about how reversal might potentially help us out by using an example.

Hint #4

The other line of thought is a tad bit complicated but essentially it builds on the idea of placing each element in its original position while keeping track of the element originally in that position. Basically, at every step, we place an element in its rightful position and keep track of the element already there or the one being overwritten in an additional variable. We can't do this in one linear pass and the idea here is based on **cyclic-dependencies** between elements.

## **Summary**

We have to rotate the elements of the given array k times to the right.

## **Solution**

#### **Approach 1: Brute Force**

The simplest approach is to rotate all the elements of the array in *k* steps by rotating the elements by 1 unit in each step.

|  |
| --- |
| class Solution {  public void rotate(int[] nums, int k) {  // speed up the rotation  k %= nums.length;  int temp, previous;  for (int i = 0; i < k; i++) {  previous = nums[nums.length - 1];  for (int j = 0; j < nums.length; j++) {  temp = nums[j];  nums[j] = previous;  previous = temp;  }  }  }  } |

**Complexity Analysis**

* Time complexity:  O(*n*×*k*). All the numbers are shifted by one step(O(*n*)) k times.
* Space complexity:  O(1). No extra space is used.

#### **Approach 2: Using Extra Array**

**Algorithm**

We use an extra array in which we place every element of the array at its correct position i.e. the number at index i*i* in the original array is placed at the index (*i*+*k*)% length of array. Then, we copy the new array to the original one.

|  |
| --- |
| class Solution {  public void rotate(int[] nums, int k) {  int[] a = new int[nums.length];  for (int i = 0; i < nums.length; i++) {  a[(i + k) % nums.length] = nums[i];  }  for (int i = 0; i < nums.length; i++) {  nums[i] = a[i];  }  }  } |

**Complexity Analysis**

* Time complexity: O(*n*). One pass is used to put the numbers in the new array. And another pass to copy the new array to the original one.
* Space complexity: O(*n*). Another array of the same size is used.

#### **Approach 3: Using Cyclic Replacements**

**Algorithm**

**Algorithm**

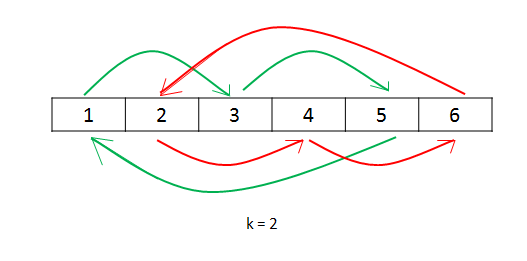
We can directly place every number of the array at its required correct position. But if we do that, we will destroy the original element. Thus, we need to store the number being replaced in a *temp* variable. Then, we can place the replaced number temp at its correct position and so on, *n* times, where *n* is the length of array. We have chosen *n* to be the number of replacements since we have to shift all the elements of the array(which is *n*). But, there could be a problem with this method, if *n*%*k*=0 where *k*=*k*%*n* (since a value of *k* larger than *n* eventually leads to a *k* equivalent to *k*%*n*). In this case, while picking up numbers to be placed at the correct position, we will eventually reach the number from which we originally started. Thus, in such a case, when we hit the original number's index again, we start the same process with the number following it.

Now let's look at the proof of how the above method works. Suppose, we have *n* as the number of elements in the array and k*k* is the number of shifts required. Further, assume *n*%*k*=0. Now, when we start placing the elements at their correct position, in the first cycle all the numbers with their index i*i* satisfying *i*%*k*=0 get placed at their required position. This happens because when we jump k steps every time, we will only hit the numbers k steps apart. We start with index *i*=0, having *i*%*k*=0. Thus, we hit all the numbers satisfying the above condition in the first cycle. When we reach back the original index, we have placed n/*k*​ elements at their correct position, since we hit only that many elements in the first cycle. Now, we increment the index for replacing the numbers. This time, we place other *n/k*​ elements at their correct position, different from the ones placed correctly in the first cycle, because this time we hit all the numbers satisfy the condition *i*%*k*=1. When we hit the starting number again, we increment the index and repeat the same process from *i*=1 for all the indices satisfying *i*%*k*==1. This happens till we reach the number with the index *i*%*k*=0 again, which occurs for *i*=*k*. We will reach such a number after a total of *k* cycles. Now, the total count of numbers exclusive numbers placed at their correct position will be k \times *k*×*n/k*​=*n*. Thus, all the numbers will be placed at their correct position.

Look at the following example to clarify the process:

nums: [1, 2, 3, 4, 5, 6]

k: 2



|  |
| --- |
| class Solution {  public void rotate(int[] nums, int k) {  k = k % nums.length;  int count = 0;  for (int start = 0; count < nums.length; start++) {  int current = start;  int prev = nums[start];  do {  int next = (current + k) % nums.length;  int temp = nums[next];  nums[next] = prev;  prev = temp;  current = next;  count++;  } while (start != current);  }  }  } |

**Complexity Analysis**

* Time complexity: O(*n*). Only one pass is used.
* Space complexity: O(1). Constant extra space is used.

#### **Approach 4: Using Reverse**

**Algorithm**

This approach is based on the fact that when we rotate the array k times, *k* elements from the back end of the array come to the front and the rest of the elements from the front shift backwards.

In this approach, we firstly reverse all the elements of the array. Then, reversing the first k elements followed by reversing the rest *n*−*k* elements gives us the required result.

Let *n*=7 and *k*=3.

Original List : 1 2 3 4 5 6 7

After reversing all numbers : 7 6 5 4 3 2 1

After reversing first k numbers : 5 6 7 4 3 2 1

After revering last n-k numbers : 5 6 7 1 2 3 4 --> Result

|  |
| --- |
| class Solution {  public void rotate(int[] nums, int k) {  k %= nums.length;  reverse(nums, 0, nums.length - 1);  reverse(nums, 0, k - 1);  reverse(nums, k, nums.length - 1);  }  public void reverse(int[] nums, int start, int end) {  while (start < end) {  int temp = nums[start];  nums[start] = nums[end];  nums[end] = temp;  start++;  end--;  }  }  } |

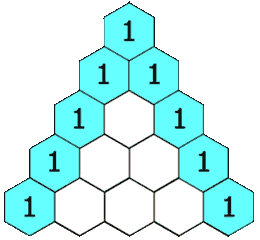
**Complexity Analysis**

* Time complexity: O(*n*). *n* elements are reversed a total of three times.
* Space complexity: O(1). No extra space is used.

**Pascal's Triangle II**

Given an integer rowIndex, return the rowIndexth row of the Pascal's triangle.

Notice that the row index starts from **0**.

  
In Pascal's triangle, each number is the sum of the two numbers directly above it.

**Follow up:**

Could you optimize your algorithm to use only *O*(*k*) extra space?

**Example 1:**

**Input:** rowIndex = 3

**Output:** [1,3,3,1]

**Example 2:**

**Input:** rowIndex = 0

**Output:** [1]

**Example 3:**

**Input:** rowIndex = 1

**Output:** [1,1]

**Constraints:**

* 0 <= rowIndex <= 33

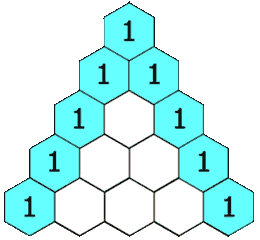
## **Solution**

If you haven't attempted [118. Pascal's Triangle](https://leetcode.com/problems/pascals-triangle/), I would strongly recommend that you try that first.

#### **Approach 1: Brute Force Recursion**

**Intuition**

We'll utilize a nice little property of Pascal's Triangle (given in the problem description):



In Pascal's triangle, each number is the sum of the two numbers directly above it.

[Approach 4](https://leetcode.com/problems/pascals-triangle-ii/solution/#approach-4-math-specifically-combinatorics) will expand more on why it is so.

**Algorithm**

Let's say we had a function getNum(rowIndex, colIndex), which gave us the colIndexth number in the rowIndexth row, we could simply build the *kth* row by repeatedly calling getNum(...) for columns 0 to *k*.

We can formulate our intuition into the following recursion:

getNum(rowIndex, colIndex) = getNum(rowIndex-1, colIndex-1) + getNum(rowIndex-1, colIndex)

The recursion ends in some known base cases:

1. The first row is just a single 11, i.e. getNum(0, ...) = 1
2. The first and last number of each row is 11, i.e. getNum(k, 0) = getNum(k, k) = 1

|  |
| --- |
| class Solution {  private int getNum(int row, int col) {  if (row == 0 || col == 0 || row == col) {  return 1;  }  return getNum(row - 1, col - 1) + getNum(row - 1, col);  }  public List<Integer> getRow(int rowIndex) {  List<Integer> ans = new ArrayList<>();  for (int i = 0; i <= rowIndex; i++) {  ans.add(getNum(rowIndex, i));  }  return ans;  }  } |

#### 

#### **Approach 2: Dynamic Programming**

**Intuition**

In the previous approach, we end up making the same recursive calls repeatedly.



For example, to calculate getNum(5, 3) and getNum(5, 4), we end up calling getNum(3, 2) thrice. To generate, the entire fifth row (0-based row indexing), we'd have to call getNum(3, 2) four times.

It makes sense to store the results of intermediate recursive calls for later use.

**Algorithm**

Simple memoization caches results of deep recursive calls and provides significant savings on runtime.

|  |
| --- |
| class Solution {  private final class RowCol {  private int row, col;  public RowCol(int row, int col) {  this.row = row;  this.col = col;  }  @Override  public int hashCode() {  int result = (int) (row ^ (row >>> 32));  return (result << 5) - 1 + (int) (col ^ (col >>> 32)); // 31 \* result == (result << 5) - 1  }  @Override  public boolean equals(Object o) {  if (this == o) return true;  if (o == null) return false;  if (this.getClass() != o.getClass()) return false;  RowCol rowCol = (RowCol) o;  return row == rowCol.row && col == rowCol.col;  }  }  private Map<RowCol, Integer> cache = new HashMap<>();  private int getNum(int row, int col) {  RowCol rowCol = new RowCol(row, col);  if (cache.containsKey(rowCol)) {  return cache.get(rowCol);  }  int computedVal =  (row == 0 || col == 0 || row == col) ? 1 : getNum(row - 1, col - 1) + getNum(row - 1, col);  cache.put(rowCol, computedVal);  return computedVal;  }  public List<Integer> getRow(int rowIndex) {  List<Integer> ans = new ArrayList<>();  for (int i = 0; i <= rowIndex; i++) {  ans.add(getNum(rowIndex, i));  }  return ans;  }  } |

But, it is worth noting that generating a number for a particular row requires only two numbers from the previous row. Consequently, generating a row only requires numbers from the previous row.

Thus, we could reduce our memory footprint by only keeping the latest row generated, and use that to generate a new row.

|  |
| --- |
| class Solution {  public List<Integer> getRow(int rowIndex) {  List<Integer> curr,  prev =  new ArrayList<>() {  {  add(1);  }  };  for (int i = 1; i <= rowIndex; i++) {  curr =  new ArrayList<>(i + 1) {  {  add(1);  }  };  for (int j = 1; j < i; j++) {  curr.add(prev.get(j - 1) + prev.get(j));  }  curr.add(1);  prev = curr;  }  return prev;  }  } |

The std::move() operator on vectors in C++ is an O(1)*O*(1) operation. [[2]](https://leetcode.com/problems/pascals-triangle-ii/solution/#fn2)



#### **Approach 3: Memory-efficient Dynamic Programming**

**Intuition**

Notice that in the previous approach, we have maintained the previous row in memory on the premise that we need terms from it to build the current row. This is true, but not wholly.

What we actually need, to generate a term in the current row, is just the two terms above it (present in the previous row).

Formally, in memory,

pascal[i][j] = pascal[i-1][j-1] + pascal[i-1][j]

where pascal[i][j] is the number in ith row and jth column of Pascal's triangle.

So, trying to compute pascal[i][j], only the memory regions of pascal[i-1][j-1] and pascal[i-1][j] are required to be accessed.

**Algorithm**

Let's take a step back and analyze the circumstances under which pascal[i][j] might be accessed. Given that we have already employed DP to save us valuable run-time, the access pattern for pascal[i][j] looks a bit like this:

* WRITE pascal[i][j] (after generating it from pascal[i-1][j-1] and pascal[i-1][j])
* READ pascal[i][j] to generate pascal[i+1][j]
* READ pascal[i][j] to generate pascal[i+1][j+1]

That's it! Once we've written out pascal[i][j]:

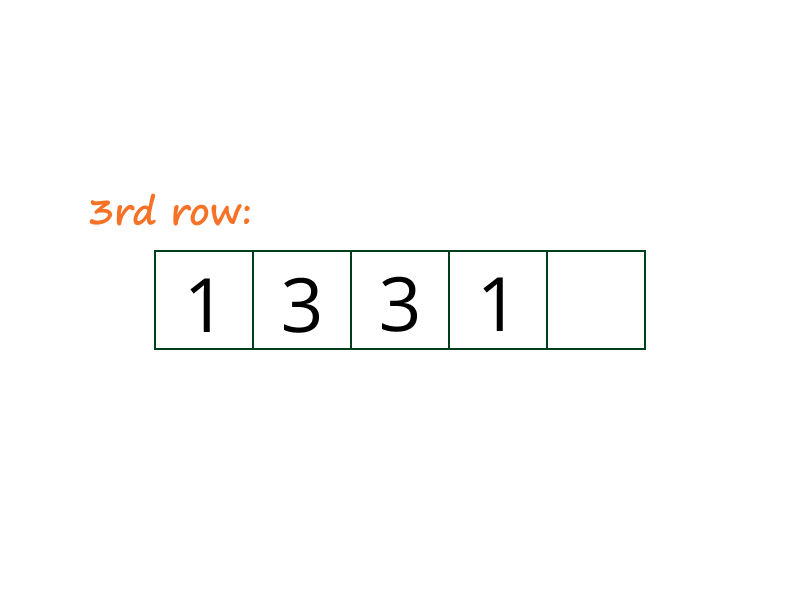
1. We don't ever need to modify it.
2. It's only read a *fixed* number of times, i.e. **twice** (thanks to DP).

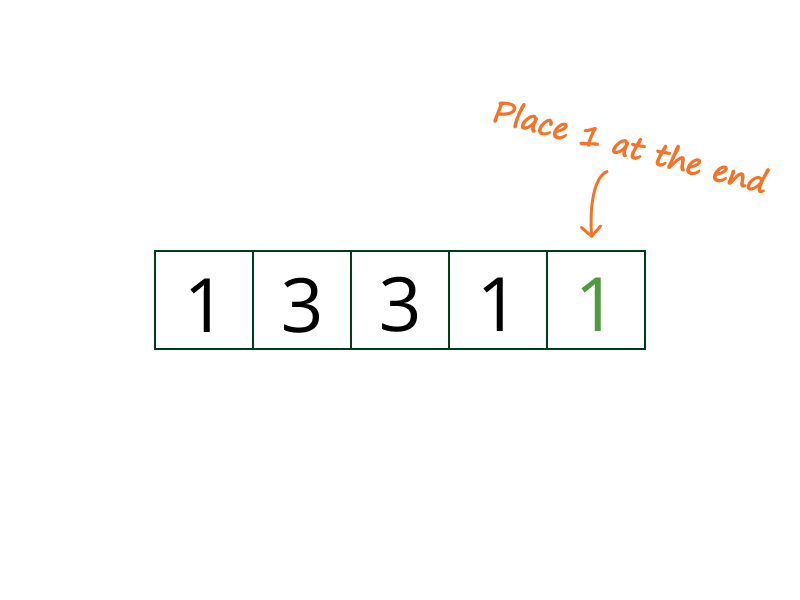
Hypothetically, if we kept the the current row (in the process of being generated) and the previous row, in the same memory block, what kind of access patterns would we see (assume pascal[j] means the *j*th number in a row)?

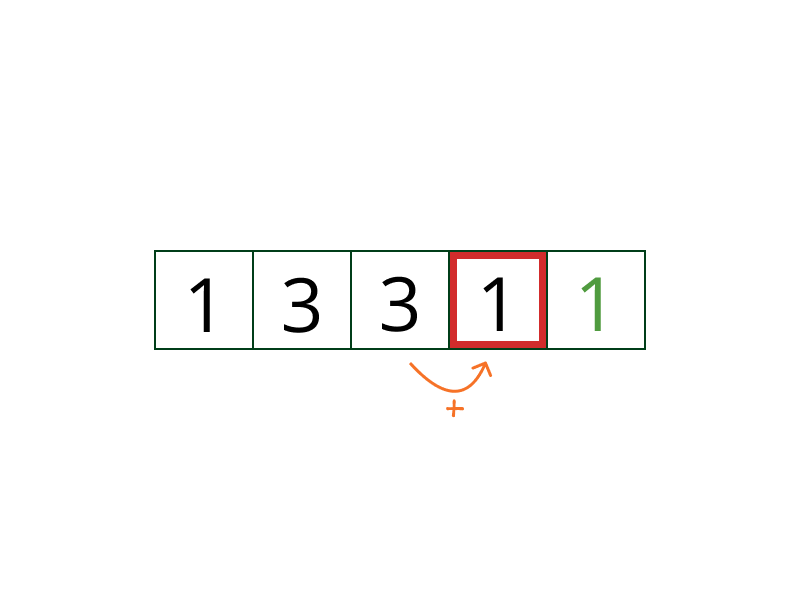
* pascal[j] was somehow generated in a previous instance. Currently, it holds the previous row value.
* pascal[j] (which holds the *j*th number of the previous row) must be read when writing out pascal[j] (the *j*th number of the current row).
  + Obviously they are the same memory location, so a conflict exists: the previous row value of pascal[j] will be lost after the write-out.
  + Is that ok? If we don't need to read the previous row value of pascal[j] anymore, is there any harm in writing out the current row value in its place?
* pascal[j] (which holds the *j*th number of the previous row) must be read when writing out pascal[j+1] (the *j+1*th number of the current row). These are two different memory locations, so there is no conflict.

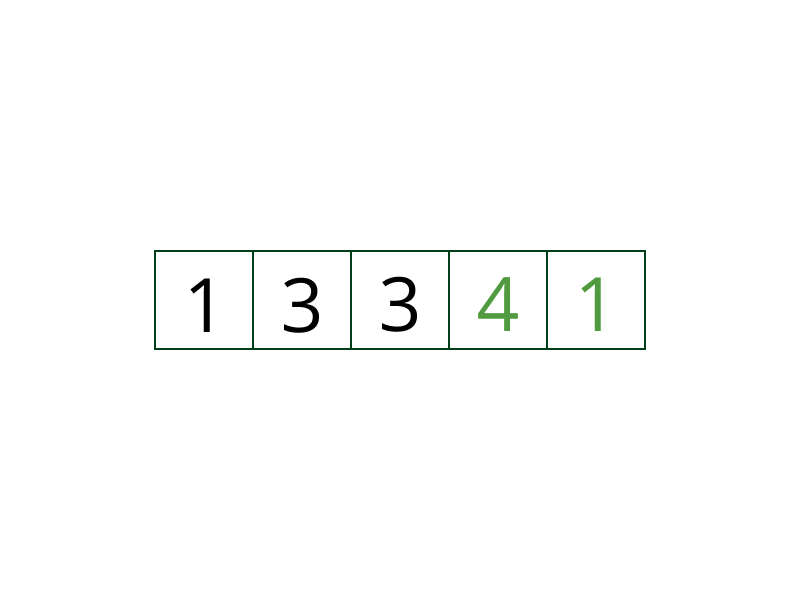
If we managed to keep all read accesses on the previous row value of pascal[j], **before** any write access to pascal[j] for the current row value, we should be good! That's possible by evaluating each row from the end, instead of the beginning. Thus, a new row value of pascal[j+1] must be generated *before* doing so for pascal[j].

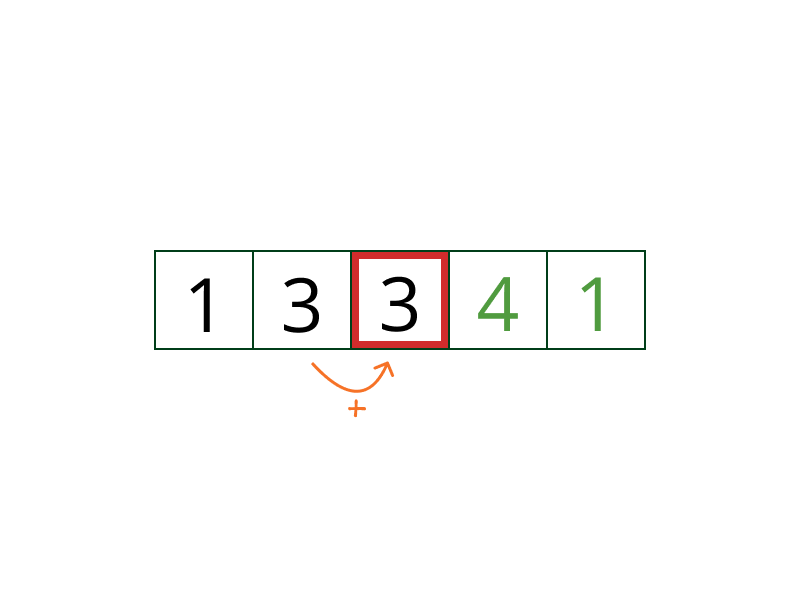
The following animation demonstrates the above algorithm, used to generate the 4th row of Pascal's Triangle, from an existing 3rd row:

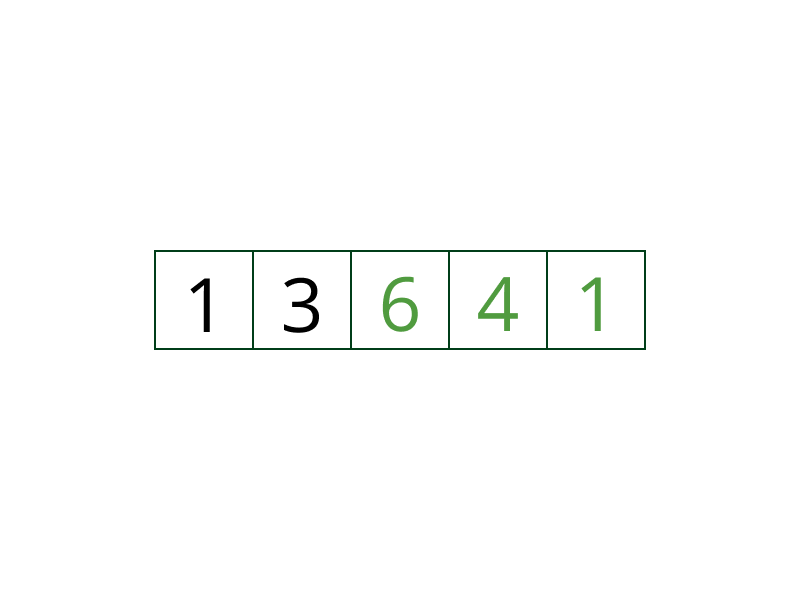


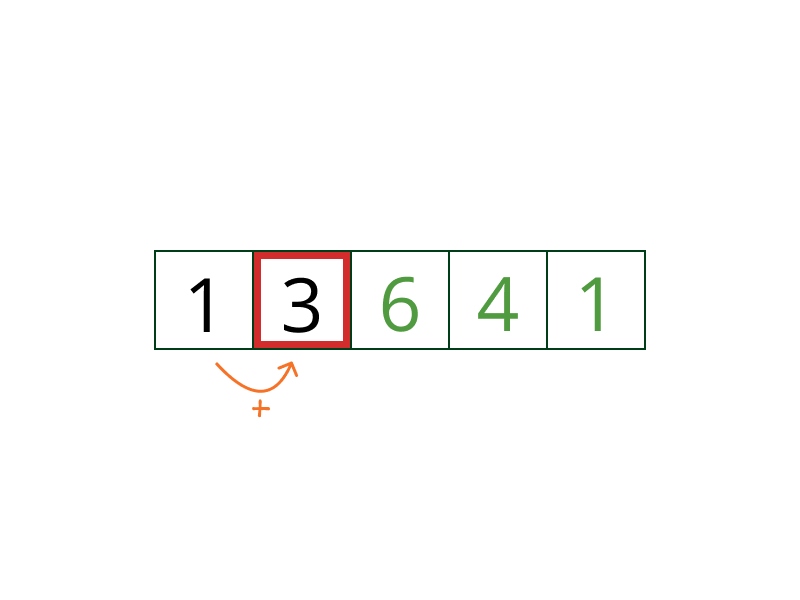














|  |
| --- |
| class Solution {  public List<Integer> getRow(int rowIndex) {  List<Integer> row =  new ArrayList<>(rowIndex + 1) {  {  add(1);  }  };  for (int i = 0; i < rowIndex; i++) {  for (int j = i; j > 0; j--) {  row.set(j, row.get(j) + row.get(j - 1));  }  row.add(1);  }  return row;  }  } |

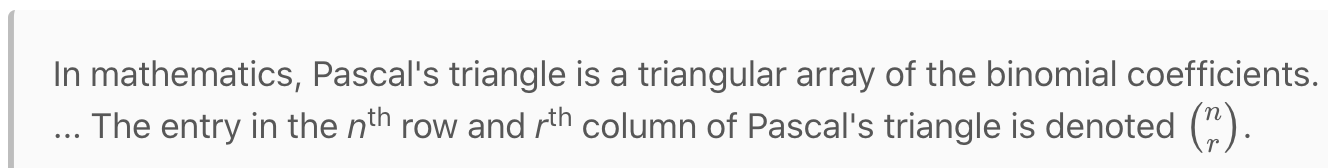
**Complexity Analysis**

* Time complexity : O(k2). Same as the previous dynamic programming approach.
* Space complexity : O(k). No extra space is used other than that required to hold the output.
* Although there is no savings in theoretical computational complexity, in practice there are some minor wins:
  + We have one vector/array instead of two. So memory consumption is roughly half.
  + No time wasted in swapping references to vectors for previous and current row.
  + Locality of reference shines through here. Since every read is for consecutive memory locations in the array/vector, we get a performance boost.

#### **Approach 4: Math! (specifically, Combinatorics)**

**Intuition**

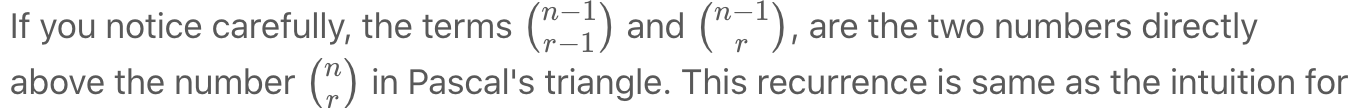
Let's go back to the definition of a Pascal's Triangle:





Binomial coefficients have an additive property, known as [Pascal's rule](https://en.wikipedia.org/wiki/Pascal%27s_rule):

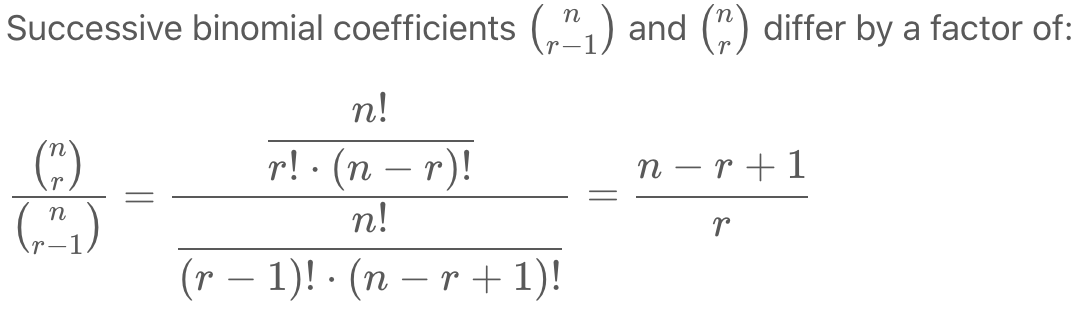




[Approach 1](https://leetcode.com/problems/pascals-triangle-ii/solution/#approach-1-brute-force-recursion).

**Algorithm**

While knowing Pascal's rule does not give us any benefits over previous approaches, knowing that the numbers in Pascal's triangle are just binomial coefficients will come in handy.



Thus, we can derive the next term in a row in Pascal's triangle, from a preceding term. Running a loop should give us the required row.

* We know that each row starts with a 1, so we have a starting point.
* We also know that the *kth* row has exactly *k*+1 terms, so we know how long we need to run the loop.

|  |
| --- |
| class Solution {  public List<Integer> getRow(int n) {  List<Integer> row =  new ArrayList<>() {  {  add(1);  }  };  for (int k = 1; k <= n; k++) {  row.add((int) ((row.get(row.size() - 1) \* (long) (n - k + 1)) / k));  }  return row;  }  } |

**Complexity Analysis**

* Time complexity : *O*(*k*). Each term is calculated once, in constant time.
* Space complexity : *O*(*k*). No extra space required other than that required to hold the output.

#### **Further Thoughts**

* The symmetry of a row in Pascal's triangle allows us to get away with computing just half of each row.

Pop Quiz: Are there any computational complexity benefits of doing this?

Pop Quiz: Can you prove why these rows are symmetrical?

1. This [Stack Overflow answer](https://stackoverflow.com/a/26229383/2844164) has a good explanation. See the parallel between the time complexity recurrence and [Pascal's rule](https://en.wikipedia.org/wiki/Pascal%27s_rule). [↩︎](https://leetcode.com/problems/pascals-triangle-ii/solution/#fnref1)
2. Starting C++11, std:move() can be used to move resources across arguments or references. Since underlying representations are simply moved, and not copied, this can be a very efficient operation to transfer elements across collections or containers. See this [Stack Overflow answer](https://stackoverflow.com/a/12613436/2844164) for more. [↩︎](https://leetcode.com/problems/pascals-triangle-ii/solution/#fnref2)

**Reverse Words in a String**

Given an input string s, reverse the order of the **words**.

A **word** is defined as a sequence of non-space characters. The **words** in s will be separated by at least one space.

Return *a string of the words in reverse order concatenated by a single space.*

**Note** that s may contain leading or trailing spaces or multiple spaces between two words. The returned string should only have a single space separating the words. Do not include any extra spaces.

**Example 1:**

**Input:** s = "the sky is blue"

**Output:** "blue is sky the"

**Example 2:**

**Input:** s = " hello world "

**Output:** "world hello"

**Explanation:** Your reversed string should not contain leading or trailing spaces.

**Example 3:**

**Input:** s = "a good example"

**Output:** "example good a"

**Explanation:** You need to reduce multiple spaces between two words to a single space in the reversed string.

**Example 4:**

**Input:** s = " Bob Loves Alice "

**Output:** "Alice Loves Bob"

**Example 5:**

**Input:** s = "Alice does not even like bob"

**Output:** "bob like even not does Alice"

**Constraints:**

* 1 <= s.length <= 104
* s contains English letters (upper-case and lower-case), digits, and spaces ' '.
* There is **at least one** word in s.

**Follow up:**

* Could you solve it **in-place** with O(1) extra space?

## **Solution**

#### **Overview**

Different interviewers would probably expect different approaches for this problem. The holy war question is to use or not use built-in methods. As you may notice, most of design problems on Leetcode are voted down because of two main reasons:

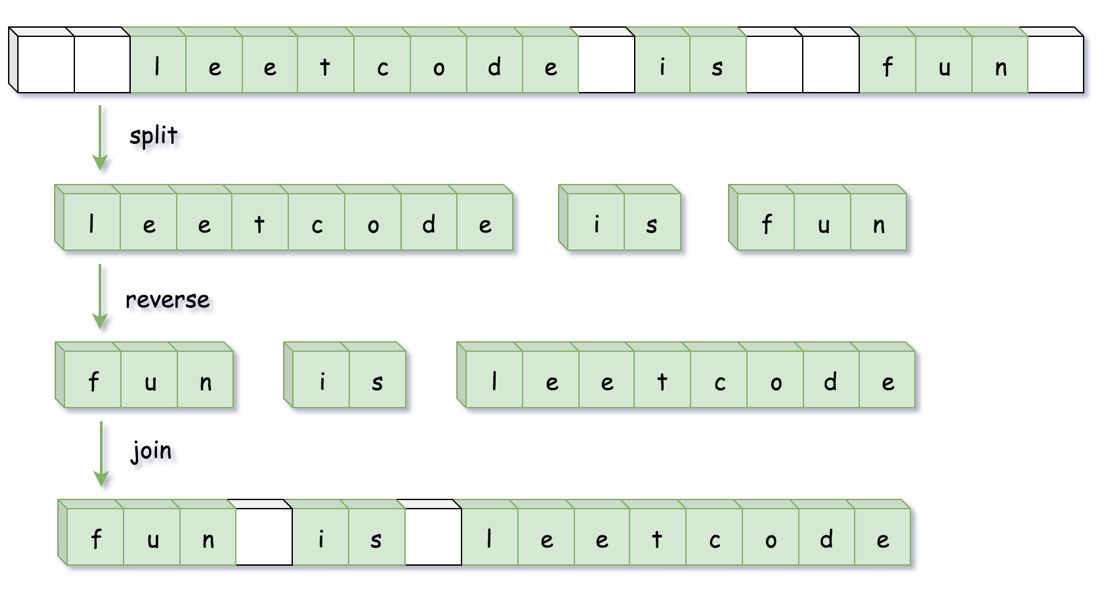
1. There was no approach with built-in methods / data structures in the article.
2. One of the approaches in the article did contain built-in methods / data structures.

Seems like the community has no common opinion yet, and in practice that means an unpredictable interview experience for some sort of problems.

Here we consider three different solutions of linear time and space complexity:

1. Use built-in split and reverse. Benefits: in-place in Python (in-place, but linear space complexity!) and the simplest one to write.
2. The most straightforward one. Trim the whitespaces, reverse the whole string and then reverse each word.  
   Benefits: could be done in-place for the languages with mutable strings.
3. Two passes approach with a deque. Move along the string, word by word, and push each new word in front of the deque. Convert the deque back into string. Benefits: two passes.

#### **Approach 1: Built-in Split + Reverse**



**Implementation**

|  |
| --- |
| class Solution {  public String reverseWords(String s) {  // remove leading spaces  s = s.trim();  // split by multiple spaces  List<String> wordList = Arrays.asList(s.split("\\s+"));  Collections.reverse(wordList);  return String.join(" ", wordList);  }  } |

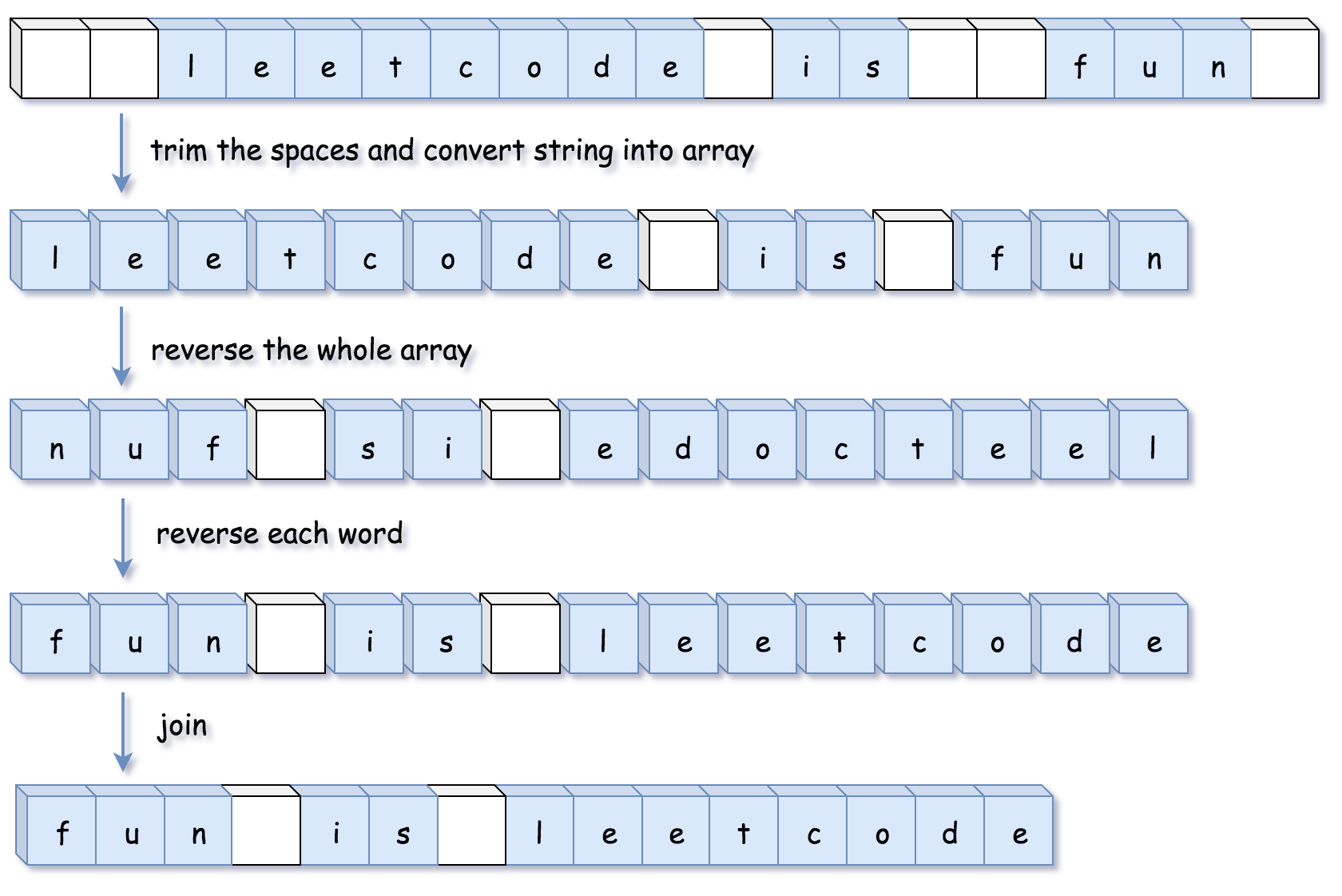
**Complexity Analysis**

* Time complexity: O(*N*), where N is a number of characters in the input string.
* Space complexity: O(*N*), to store the result of split by spaces.

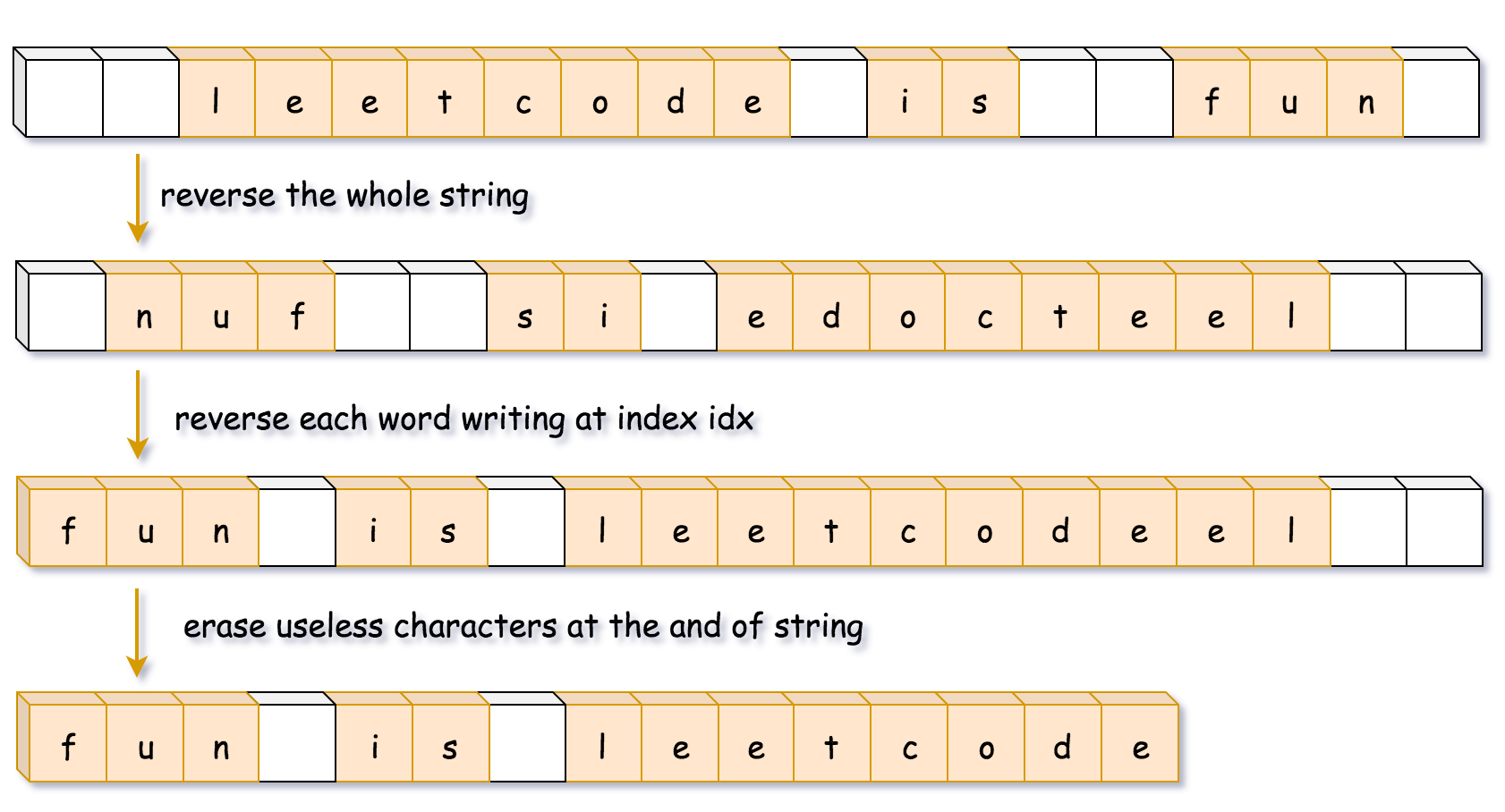
#### **Approach 2: Reverse the Whole String and Then Reverse Each Word**

The implementation of this approach will be different for Java/Python (= immutable strings) and C++ (= mutable strings).

In the case of immutable strings one has first to convert string into mutable data structure, and hence it makes sense to trim all spaces during that conversion.



In the case of mutable strings, there is no need to allocate an additional data structure, one could make all job done in-place. In such a case it makes sense to reverse words and trim spaces at the same time.



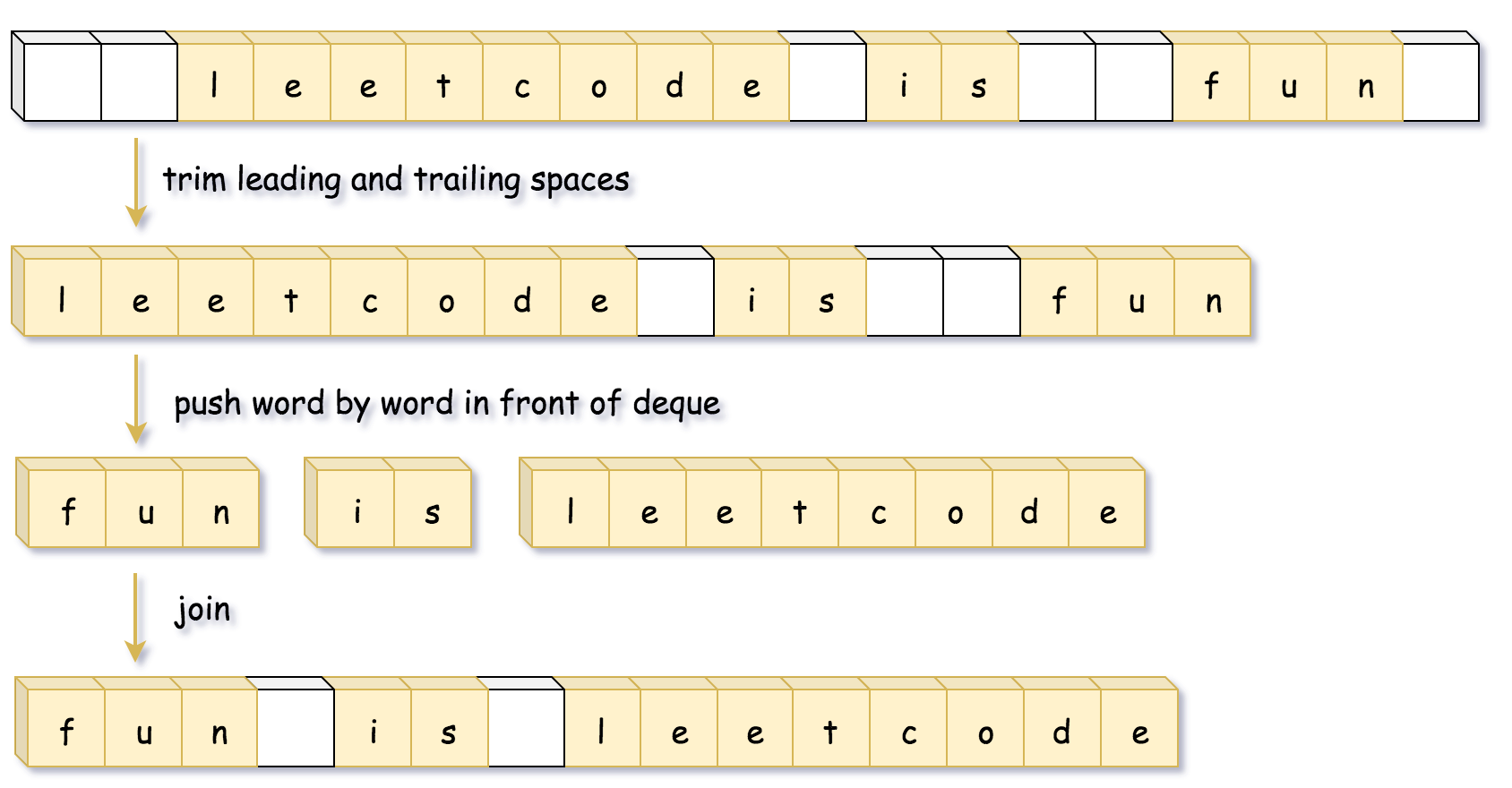
**Implementation**

|  |
| --- |
| class Solution {  public StringBuilder trimSpaces(String s) {  int left = 0, right = s.length() - 1;  // remove leading spaces  while (left <= right && s.charAt(left) == ' ') ++left;  // remove trailing spaces  while (left <= right && s.charAt(right) == ' ') --right;  // reduce multiple spaces to single one  StringBuilder sb = new StringBuilder();  while (left <= right) {  char c = s.charAt(left);  if (c != ' ') sb.append(c);  else if (sb.charAt(sb.length() - 1) != ' ') sb.append(c);  ++left;  }  return sb;  }  public void reverse(StringBuilder sb, int left, int right) {  while (left < right) {  char tmp = sb.charAt(left);  sb.setCharAt(left++, sb.charAt(right));  sb.setCharAt(right--, tmp);  }  }  public void reverseEachWord(StringBuilder sb) {  int n = sb.length();  int start = 0, end = 0;  while (start < n) {  // go to the end of the word  while (end < n && sb.charAt(end) != ' ') ++end;  // reverse the word  reverse(sb, start, end - 1);  // move to the next word  start = end + 1;  ++end;  }  }  public String reverseWords(String s) {  // converst string to string builder  // and trim spaces at the same time  StringBuilder sb = trimSpaces(s);  // reverse the whole string  reverse(sb, 0, sb.length() - 1);  // reverse each word  reverseEachWord(sb);  return sb.toString();  }  } |

**Complexity Analysis**

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

#### **Approach 3: Deque of Words**



**Implementation**

|  |
| --- |
| class Solution {  public String reverseWords(String s) {  int left = 0, right = s.length() - 1;  // remove leading spaces  while (left <= right && s.charAt(left) == ' ') ++left;  // remove trailing spaces  while (left <= right && s.charAt(right) == ' ') --right;  Deque<String> d = new ArrayDeque();  StringBuilder word = new StringBuilder();  // push word by word in front of deque  while (left <= right) {  char c = s.charAt(left);  if ((word.length() != 0) && (c == ' ')) {  d.offerFirst(word.toString());  word.setLength(0);  } else if (c != ' ') {  word.append(c);  }  ++left;  }  d.offerFirst(word.toString());  return String.join(" ", d);  }  } |

**Complexity Analysis**

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

**Reverse Words in a String III**

Given a string, you need to reverse the order of characters in each word within a sentence while still preserving whitespace and initial word order.

**Example 1:**

**Input:** "Let's take LeetCode contest"

**Output:** "s'teL ekat edoCteeL tsetnoc"

**Note:** In the string, each word is separated by single space and there will not be any extra space in the string.

## **Solution**

#### **Approach #1 Simple Solution[Accepted]**

The first method is really simple. We simply split up the given string based on whitespaces and put the individual words in an array of strings. Then, we reverse each individual string and concatenate the result. We return the result after removing the additional whitespaces at the end.

|  |
| --- |
| public class Solution {  public String reverseWords(String s) {  String words[] = s.split(" ");  StringBuilder res=new StringBuilder();  for (String word: words)  res.append(new StringBuffer(word).reverse().toString() + " ");  return res.toString().trim();  }  } |

**Complexity Analysis**

* Time complexity : *O*(*n*). where *n* is the length of the string.
* Space complexity : *O*(*n*). res*res* of size *n* is used.

#### **Approach #2 Without using pre-defined split and reverse function [Accepted]**

**Algorithm**

We can create our own split and reverse function. Split function splits the string based on the delimiter " "(space) and returns the array of words. Reverse function returns the string after reversing the characters.

|  |
| --- |
| public class Solution {  public String reverseWords(String s) {  String words[] = split(s);  StringBuilder res=new StringBuilder();  for (String word: words)  res.append(reverse(word) + " ");  return res.toString().trim();  }  public String[] split(String s) {  ArrayList < String > words = new ArrayList < > ();  StringBuilder word = new StringBuilder();  for (int i = 0; i < s.length(); i++) {  if (s.charAt(i) == ' ') {  words.add(word.toString());  word = new StringBuilder();  } else  word.append( s.charAt(i));  }  words.add(word.toString());  return words.toArray(new String[words.size()]);  }  public String reverse(String s) {  StringBuilder res=new StringBuilder();  for (int i = 0; i < s.length(); i++)  res.insert(0,s.charAt(i));  return res.toString();  }  } |

\*\*Complexity Analysis\*\*

* Time complexity : *O*(*n*). where *n* is the length of the string.
* Space complexity : *O*(*n*). *res* of size *n* is used.

#### **Approach #3 Using StringBuilder and reverse method [Accepted]**

**Algorithm**

Instead of using split method, we can use temporary string *word* to store the word. We simply append the characters to the *word* until ' ' character is not found. On getting ' ' we append the reverse of the *word* to the resultant string *result*. Also after completion of loop , we still have to append the *reverse* of the *word*(last word) to the *result* string.

Below code is inspired by [@ApolloX](http://leetcode.com/apolloX).

|  |
| --- |
| public class Solution {  public String reverseWords(String input) {  final StringBuilder result = new StringBuilder();  final StringBuilder word = new StringBuilder();  for (int i = 0; i < input.length(); i++) {  if (input.charAt(i) != ' ') {  word.append(input.charAt(i));  } else {  result.append(word.reverse());  result.append(" ");  word.setLength(0);  }  }  result.append(word.reverse());  return result.toString();  }  } |

\*\*Complexity Analysis\*\*

* Time complexity : *O*(*n*). Single loop upto *n* is there, where *n* is the length of the string.
* Space complexity : *O*(*n*). *result* and *word* size will grow upto *n*.

**Remove Duplicates from Sorted Array**

Given a sorted array *nums*, remove the duplicates [**in-place**](https://en.wikipedia.org/wiki/In-place_algorithm) such that each element appears only *once* and returns the new length.

Do not allocate extra space for another array, you must do this by **modifying the input array**[**in-place**](https://en.wikipedia.org/wiki/In-place_algorithm) with O(1) extra memory.

**Clarification:**

Confused why the returned value is an integer but your answer is an array?

Note that the input array is passed in by **reference**, which means a modification to the input array will be known to the caller as well.

Internally you can think of this:

// **nums** is passed in by reference. (i.e., without making a copy)

int len = removeDuplicates(nums);

// any modification to **nums** in your function would be known by the caller.

// using the length returned by your function, it prints the first **len** elements.

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

    print(nums[i]);

}

**Example 1:**

**Input:** nums = [1,1,2]

**Output:** 2, nums = [1,2]

**Explanation:** Your function should return length = **2**, with the first two elements of *nums* being **1** and **2** respectively. It doesn't matter what you leave beyond the returned length.

**Example 2:**

**Input:** nums = [0,0,1,1,1,2,2,3,3,4]

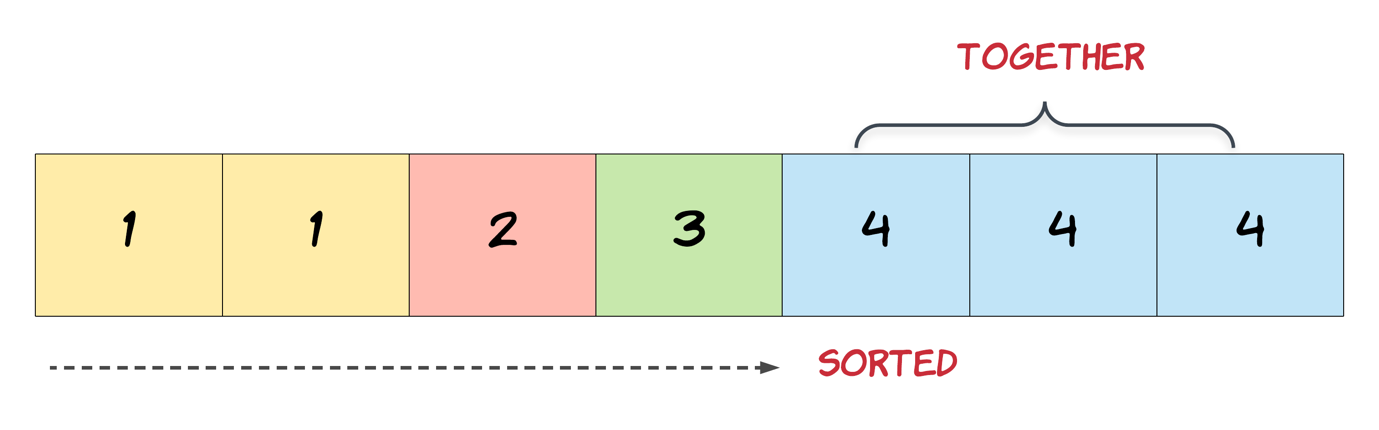
**Output:** 5, nums = [0,1,2,3,4]

**Explanation:** Your function should return length = **5**, with the first five elements of *nums* being modified to **0**, **1**, **2**, **3**, and **4** respectively. It doesn't matter what values are set beyond the returned length.

**Constraints:**

* 0 <= nums.length <= 3 \* 104
* -104 <= nums[i] <= 104
* nums is sorted in ascending order.

Hint #1

In this problem, the key point to focus on is the input array being sorted. As far as duplicate elements are concerned, what is their positioning in the array when the given array is sorted? Look at the image above for the answer. If we know the position of one of the elements, do we also know the positioning of all the duplicate elements?  


Hint #2

We need to modify the array in-place and the size of the final array would potentially be smaller than the size of the input array. So, we ought to use a two-pointer approach here. One, that would keep track of the current element in the original array and another one for just the unique elements.

Hint #3

Essentially, once an element is encountered, you simply need to **bypass** its duplicates and move on to the next unique element.

## **Solution**

#### **Approach 1: Two Pointers**

**Algorithm**

Since the array is already sorted, we can keep two pointers *i* and *j*, where *i* is the slow-runner while *j* is the fast-runner. As long as *nums*[*i*]=*nums*[*j*], we increment *j* to skip the duplicate.

When we encounter *nums*[*j*]!​=*nums*[*i*], the duplicate run has ended so we must copy its value to *nums*[*i*+1]. *i* is then incremented and we repeat the same process again until *j* reaches the end of array.

|  |
| --- |
| public int removeDuplicates(int[] nums) {  if (nums.length == 0) return 0;  int i = 0;  for (int j = 1; j < nums.length; j++) {  if (nums[j] != nums[i]) {  i++;  nums[i] = nums[j];  }  }  return i + 1;  } |

**Complexity analysis**

* Time complextiy : *O*(*n*). Assume that *n* is the length of array. Each of i*i* and *j* traverses at most *n* steps.
* Space complexity : *O*(1).

**Move Zeroes**

Given an array nums, write a function to move all 0's to the end of it while maintaining the relative order of the non-zero elements.

**Example:**

**Input:** [0,1,0,3,12]

**Output:** [1,3,12,0,0]

**Note**:

1. You must do this **in-place** without making a copy of the array.
2. Minimize the total number of operations.

Hint #1

**In-place** means we should not be allocating any space for extra array. But we are allowed to modify the existing array. However, as a first step, try coming up with a solution that makes use of additional space. For this problem as well, first apply the idea discussed using an additional array and the in-place solution will pop up eventually.

Hint #2

A **two-pointer** approach could be helpful here. The idea would be to have one pointer for iterating the array and another pointer that just works on the non-zero elements of the array.

## **Solution**

This question comes under a broad category of "Array Transformation". This category is the meat of tech interviews. Mostly because arrays are such a simple and easy to use data structure. Traversal or representation doesn't require any boilerplate code and most of your code will look like the Pseudocode itself.

The 2 requirements of the question are:

1. Move all the 0's to the end of array.
2. All the non-zero elements must retain their original order.

It's good to realize here that both the requirements are mutually exclusive, i.e., you can solve the individual sub-problems and then combine them for the final solution.

#### **Approach #1 (Space Sub-Optimal) [Accepted]**

**C++**

|  |
| --- |
| void moveZeroes(vector<int>& nums) {  int n = nums.size();  // Count the zeroes  int numZeroes = 0;  for (int i = 0; i < n; i++) {  numZeroes += (nums[i] == 0);  }  // Make all the non-zero elements retain their original order.  vector<int> ans;  for (int i = 0; i < n; i++) {  if (nums[i] != 0) {  ans.push\_back(nums[i]);  }  }  // Move all zeroes to the end  while (numZeroes--) {  ans.push\_back(0);  }  // Combine the result  for (int i = 0; i < n; i++) {  nums[i] = ans[i];  }  } |

**Complexity Analysis**

Space Complexity : *O*(*n*). Since we are creating the "ans" array to store results.

Time Complexity: *O*(*n*). However, the total number of operations are sub-optimal. We can achieve the same result in less number of operations.

If asked in an interview, the above solution would be a good start. You can explain the interviewer(not code) the above and build your base for the next Optimal Solution.

#### **Approach #2 (Space Optimal, Operation Sub-Optimal) [Accepted]**

This approach works the same way as above, i.e. , first fulfills one requirement and then another. The catch? It does it in a clever way. The above problem can also be stated in alternate way, " Bring all the non 0 elements to the front of array keeping their relative order same".

This is a 2 pointer approach. The fast pointer which is denoted by variable "cur" does the job of processing new elements. If the newly found element is not a 0, we record it just after the last found non-0 element. The position of last found non-0 element is denoted by the slow pointer "lastNonZeroFoundAt" variable. As we keep finding new non-0 elements, we just overwrite them at the "lastNonZeroFoundAt + 1" 'th index. This overwrite will not result in any loss of data because we already processed what was there(if it were non-0,it already is now written at it's corresponding index,or if it were 0 it will be handled later in time).

After the "cur" index reaches the end of array, we now know that all the non-0 elements have been moved to beginning of array in their original order. Now comes the time to fulfil other requirement, "Move all 0's to the end". We now simply need to fill all the indexes after the "lastNonZeroFoundAt" index with 0.

**C++**

|  |
| --- |
| void moveZeroes(vector<int>& nums) {  int lastNonZeroFoundAt = 0;  // If the current element is not 0, then we need to  // append it just in front of last non 0 element we found.  for (int i = 0; i < nums.size(); i++) {  if (nums[i] != 0) {  nums[lastNonZeroFoundAt++] = nums[i];  }  }  // After we have finished processing new elements,  // all the non-zero elements are already at beginning of array.  // We just need to fill remaining array with 0's.  for (int i = lastNonZeroFoundAt; i < nums.size(); i++) {  nums[i] = 0;  }  } |

**Complexity Analysis**

Space Complexity : *O*(1). Only constant space is used.

Time Complexity: *O*(*n*). However, the total number of operations are still sub-optimal. The total operations (array writes) that code does is *n* (Total number of elements).

#### **Approach #3 (Optimal) [Accepted]**

The total number of operations of the previous approach is sub-optimal. For example, the array which has all (except last) leading zeroes: [0, 0, 0, ..., 0, 1].How many write operations to the array? For the previous approach, it writes 0's n-1*n*−1 times, which is not necessary. We could have instead written just once. How? ..... By only fixing the non-0 element,i.e., 1.

The optimal approach is again a subtle extension of above solution. A simple realization is if the current element is non-0, its' correct position can at best be it's current position or a position earlier. If it's the latter one, the current position will be eventually occupied by a non-0 ,or a 0, which lies at a index greater than 'cur' index. We fill the current position by 0 right away,so that unlike the previous solution, we don't need to come back here in next iteration.

In other words, the code will maintain the following invariant:

1. All elements before the slow pointer (lastNonZeroFoundAt) are non-zeroes.
2. All elements between the current and slow pointer are zeroes.

Therefore, when we encounter a non-zero element, we need to swap elements pointed by current and slow pointer, then advance both pointers. If it's zero element, we just advance current pointer.

With this invariant in-place, it's easy to see that the algorithm will work.

**C++**

|  |
| --- |
| void moveZeroes(vector<int>& nums) {  for (int lastNonZeroFoundAt = 0, cur = 0; cur < nums.size(); cur++) {  if (nums[cur] != 0) {  swap(nums[lastNonZeroFoundAt++], nums[cur]);  }  }  } |

**Complexity Analysis**

Space Complexity : *O*(1). Only constant space is used.

Time Complexity: *O*(*n*). However, the total number of operations are optimal. The total operations (array writes) that code does is Number of non-0 elements.This gives us a much better best-case (when most of the elements are 0) complexity than last solution. However, the worst-case (when all elements are non-0) complexity for both the algorithms is same.