



# ACEING THE CODING ROUND

125+ MOST POPULAR INTERVIEW PROBLEMS

A Definitive Guide to Crack Coding  
Interviews of Product Companies



WITH FREE ACCESS TO



Learn. Code. Upskill.

# **ACING THE CODING ROUND**

## **125+ MOST POPULAR INTERVIEW PROBLEMS**

A DEFINITIVE GUIDE TO CRACK CODING INTERVIEWS OF PRODUCT COMPANIES

LEARN. CODE. UPSKIL

Copyright © 2021 Coding Ninjas All rights reserved

No part of this book may be reproduced, or stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without express written permission of the publisher.

# Table of Contents

[1. Arrays](#)

[2. Multidimensional Arrays](#)

[3. Strings](#)

[4. Stacks and Queues](#)

[5. Linked Lists – I](#)

[6. Linked Lists – II](#)

[7. Trees](#)

[8. Binary Search Tree \(BST\)](#)

[9. Priority Queue and Heaps](#)

[10. Graphs](#)

[11. Recursion and Backtracking](#)

[12. Dynamic Programming and Memoization](#)

[13. Hashmaps, Circular Queues, and Deques](#)

[14. Tries and String Algorithms](#)

# Preface

Acing the Coding Round is a remarkable initiative taken by Coding Ninjas to assist the final-year undergraduates in kick-starting their professional journey. This book offers a research-based planned trajectory to prepare the aspiring candidates to excel at the technical rounds of their interviews.

The book is equally suited for professionals and post-graduates who want to brush through their DSA skills before appearing for their next interview.

The book is divided into 14 chapters. Each chapter or a set of chapters typically covers only one of the data structures. Thus providing an opportunity to study and practice the concepts in depth.

Each chapter begins with an overview of the conceptual basics to help the reader scan through the main ideas used in the chapter. The basics are followed by a set of practice questions arranged in an increasing level of difficulty.

Each question has been solved using multiple approaches, from naive to most optimised, thus giving the readers an opportunity to solve a question in an approach that suits them the best. The book is language-agnostic, so you can solve questions in the language you are most comfortable with.

The problems discussed in the book are carefully selected from a pool of 2000+ coding problems on our platform—[Codestudio](#). You are highly encouraged to explore it and enhance your skills further by attempting the given challenges. Codestudio is a great place to learn, practice, and participate among a constantly increasing community of novice to expert programmers.

With more than 125 questions to practice from this book, we believe that you will be able to master the fundamentals of data structures and algorithms, and feel confident in attempting the coding problems asked in the recruitment process.

Best wishes,  
Coding Ninjas Team

# How to use this book

1. Click on the problem or the url given below it. The link will take you to the corresponding problem on the CodeStudio platform.

The screenshot shows a web page with a light blue header bar containing the URL [https://www.codingninjas.com/codestudio/problems/overlapping-intervals\\_630417](https://www.codingninjas.com/codestudio/problems/overlapping-intervals_630417). Below the header, the title "5. Overlapping Intervals" is displayed in bold. A mouse cursor is hovering over the URL bar. The main content area contains a "Problem Statement" section with the text: "You have been given the start and end times of N intervals. Write a function to check if any two intervals overlap with each other." It also includes a "Note" section about intervals starting at the same time, and "Input Format" and "Output Format" sections with detailed descriptions.

2. CodeStudio link for the problem is now open. You can start coding on the right-side in the available code editor.

The screenshot shows the "Overlapping Intervals" problem on the CodeStudio platform. The left side of the screen displays the problem statement, notes, input format, and output format. The right side features a code editor with a dark theme and a light border. Inside the code editor, there is a placeholder text "Your code goes here." A mouse cursor is positioned over this text, with a callout bubble containing the text "'Write code here'". The code editor has tabs for Java (SE 1.8) and C/C++.

**3. The e-book offers step-by-step multiple approaches to solve the problem. You may refer to these approaches while solving the problem.**

Approach 1: Brute Force Approach

**Steps:**

1. Iterate over the intervals one by one.
2. For every interval, check if the end value of one interval is less than the start value of the other interval.
3. If this condition fails, the intervals are overlapping. Therefore return false.
4. Otherwise, return true.

**Time Complexity:**  $O(N^2)$ , where  $N$  is the total number of intervals.  
For every interval, we check the remaining  $(N - i - 1)$  intervals where  $i$  is the current interval.  
Thus, we would end up with quadratic complexity.

**Space Complexity:**  $O(1)$  since we don't use any extra space.

Approach 2: Sorting Approach

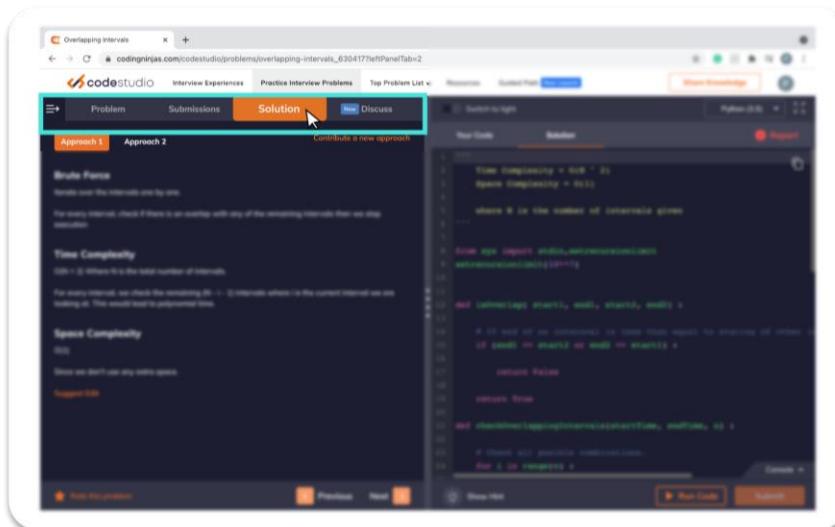
**Steps:**

1. Sort the list of intervals first on the basis of their start time and then iterate through the array
2. If the start time of an interval is less than the end of the previous interval, then there is an overlap, and we can return true.
3. If we iterate through the entire array without finding an overlap, we can return false.

**Time Complexity:**  $O(N \log N)$ , where  $N$  is the total number of intervals. Sorting the list would take  $O(N \log N)$  operations. Later, iterating over the list again to find the overlap would take  $N$  operations. Hence the overall time complexity boils down to  $O(N \log N)$

**Space Complexity:**  $O(N)$ , where  $N$  is the total number of intervals.  
Sorting takes  $O(N)$  extra space.

**4. If you need additional support, you can open the 'Solution' tab in the CodeStudio and see the actual solution. Once you have seen the solution, you will be able to go back and code. However, you wouldn't get any points to solve the problem.**



*This page is intentionally left blank.*

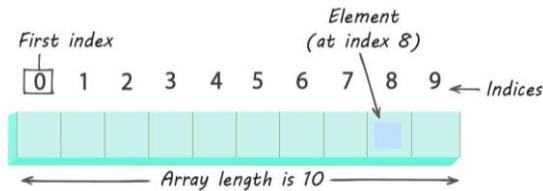
# 1. Arrays

---

An array is defined as a **fixed-size** collection of elements of the **same data type** stored in **contiguous memory** locations. It is the simplest data structure where each element of the array can be accessed using its index.

## Properties of Arrays

- Each element of the array is of the same data type and the same size.
- The elements of the array are accessed by using their index. The index of the first element is always 0. Thus the index of an array of size N ranges from **0** to **N – 1**.



In the given example, we see that the element at the 8th index—array [8] is the array's ninth element.

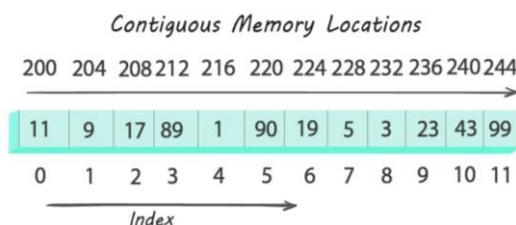
- Elements of the array are stored in contiguous memory locations.

## Accessing Array Elements

Every array is identified by its **base address**—that is, the location of the first element of the array in the memory. The base address helps in identifying the address of all the elements of the array.

For example, the given representation shows the memory allocation of an array that stores integer data types. Given that the integer data type occupies 4 bytes, the **nth** element's location of the given array can be easily calculated by adding the offset to the base address.

For instance, the address of element at index 6 will be  $200 + 4 \times (6 - 0) = 224$ .



## Where can Arrays be Used?

- Arrays are typically used where the number of elements to be stored is already known.
- Arrays work well when we have to organise data in a **multi-dimensional format**. We can declare arrays of as many dimensions as we want.
- Arrays are commonly used in computer programs to organize data so that a related set of values can be easily sorted or searched.
- Generally, when we require **fast access times**, we prefer arrays since they provide **O(1)** access times. Elements of an array **arr** can be simply accessed using statements like **arr[i]**, where **i** is an integer within the range of indices of the array. When the index of the element to be modified is known beforehand, it can be efficiently modified using arrays due to **quick access time** and **mutability**.

## Drawbacks of Using Arrays

- The size of the array needs to be declared in advance and cannot be altered later as the structure of an array is rigid and non-dynamic in nature.
- Declaring the size of the array in advance can be a challenge. If extra storage is allocated, it leads to the wastage of space. The elements, if accessed, display garbage values. And likewise, if enough storage is not provided, errors will be displayed for the exceeding indices.
- We know that the elements of an array are stored in contiguous memory locations. Thus for every insertion and deletion, we need to shift the elements to create and re-organise space, and this operation is typically time-consuming and costly.
- Allocating more memory than the requirement leads to wastage of memory space and less allocation of memory also leads to a problem that we don't have sufficient space to store our data.

## Time Complexity of Various Operations

- **Accessing elements:** Since elements in an array are stored at contiguous memory locations, they can be accessed very efficiently (random access) in  $O(1)$  time when their indices are given.
- **Finding elements:** Finding an element in an array takes  $O(N)$  time in the worst case, where  $N$  is the size of the array, as you may need to traverse the entire array. By using different algorithms, the time can be reduced significantly.
- **Inserting elements:**

**Case I:** Insertion of elements at the end of the array

This takes **O(1)** time provided that the space is available to insert the new element.

**Case II:** Insertion of elements at the beginning or at any given index

This involves shifting the elements and is possible only if the required space is available.

- ❑ If we want to insert an element at index  $i$ , all the elements starting from the index,  $i$  need to be shifted by one position. Thus, the time complexity for inserting an element at index  $i$  is **O(N)**.
- ❑ Inserting an element at the beginning of the array involves moving all elements by one position and takes **O(N)** time.

- **Deleting elements:**

**Case I:** Similar to the insertion operation, the deletion of any element from the end of the array takes **O(1)** time.

**Case II:** Deleting elements from any index other than the last index involves moving elements and re-adjusting the space.

1. If we want to delete an element at index  $i$ , all the elements starting from index  $(i + 1)$  need to be shifted by one position. Thus, the time complexity for deleting an element at index  $i$  is **O(N)**.
  2. Deleting an element from the beginning involves moving all elements starting from index 1 by one position and takes **O(N)** time.
- 

## Practice Problems

### 1. Smallest Subarray with K Distinct Elements [<https://coding.ninja/P1>]

**Problem Statement:** Given an array **A** of integers, find the smallest (contiguous) subarray of **A** containing exactly **K** distinct integers. For example, [1, 2, 2, 3, 1, 3] has three distinct integers: 1, 2, and 3. If there is no such subarray found, print -1.

**Note:** If more than one such contiguous subarrays exist, print the subarray having the smallest leftmost index. In the above example, the subarrays: [1,2], [2,3], [3,1], [1,3] are the smallest subarrays containing two distinct elements. In this case, we will print the starting and ending index of subarray [1,2] separated by space—that is, 0 1.

#### Input Format:

The **first line** contains two integers—**N** and **K**, the total number of integers and the number of distinct integers, respectively.

The **second line** contains **N** space-separated integers describing elements of the array **A**.

#### **Output Format:**

Two integers denoting the start and end index of the subarray, if it exists, otherwise print -1.

#### **Sample Input:**

```
8 3  
4 2 2 2 3 4 4 3
```

#### **Sample Output:**

```
3 5
```

#### **Explanation:**

For the given array [4, 2, 2, 2, 3, 4, 4, 4], the array[3..5]—that is, [2, 3, 4] contains three distinct elements which is the smallest possible array with three distinct elements.

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

#### **Steps:**

1. Pick each element from the array as the starting element(i) of the subarray.
  - a. Initialize an empty set to store distinct elements in the subarray.
  - b. Pick each remaining element ( $i, i + 1, \dots, n - 1$ ) from the array as the last element ( $j^{\text{th}}$ ).
    - i. Add the current element to the set.
    - ii. If the set size equals K, then update the results and break from the inner loop. We have already found K distinct elements. Increasing the size of the subarray will either make more distinct elements or increase the subarray size with repeated elements that are not to be considered in the required results.
  - c. If  $j == \text{size of the array}$ , it means we have not found any desired subarray starting from the  $i^{\text{th}}$  index and going forward we will be having fewer elements to consider.
  - d. Break from the outer loop.
2. Print the output if found—otherwise, print -1.

**Time Complexity:**  $O(N^2)$ , where **N** is the number of elements as the ends of the subarray are picked using two nested loops (one inside another).

**Space Complexity:**  $O(N)$ , in the worst case, we can have all **N** elements in our set.

### **Approach 2: Sliding Window Approach**

#### **Steps:**

Using the sliding window technique,

1. Initialize the map to store the count of each element.

2. Initialize start and end to store the index of the required subarray.
3. Here  $i, j$  denote the start and end of the window respectively, initially:  $i = 0, j = 0$ .
4. While the end < size of the array,
  1. In the map, increase the count of the end index's character by one. If the character is not present in the map, insert the character in the map, with count as one.
  2. While the count of distinct elements in the map equals  $K$  (—that is, the size of the map is equal to  $K$ ).
    - i. Find the length of subarray as  $j - i$ .
    - ii. Compare it with the length of the minimum length you've found so far (initially 0).
    - iii. If the current length is less than the minimum, update the minimum length.
    - iv. Decrease the count of  $i$ th character by one. If it is already one, remove that character from the map.
    - v. Move  $i$  forward by one.
5. If end is equal to  $n$ , that means no subarray was found with  $K$  distinct characters, and we can print  $-1$ . Otherwise, we can print the start and end index we found.

**Time Complexity:**  $O(N)$ , where  $N$  is the number of elements. In the worst case, each element will be added once and removed once from the set.

**Space Complexity:**  $O(K)$ . In the worst case, we will be only having  $K$  elements in our map.

## 2. Sum of Infinite Array [<https://coding.ninja/P2>]

**Problem Statement:** Given an array,  $\mathbf{A}$  of  $\mathbf{N}$  integers and you have also defined the new array  $\mathbf{B}$  as a concatenation of array " $\mathbf{A}$ " for an infinite number of times.

For example, if the given array " $\mathbf{A}$ " is  $[1, 2, 3]$  then, infinite array " $\mathbf{B}$ " is  $[1, 2, 3, 1, 2, 3, 1, 2, 3, \dots]$ .

Now you are given  $\mathbf{Q}$  queries. Each query consists of two integers, " $\mathbf{L}$ " and " $\mathbf{R}$ ". Your task is to find the sum of the subarray from index " $\mathbf{L}$ " to " $\mathbf{R}$ " (both inclusive) in the infinite array " $\mathbf{B}$ " for each query.

### Note:

1. The value of the sum can be very large. Return the answer as modulo  $10^9 + 7$ .
2. Here **1-based** indexing is used.

### Input Format:

The **first line** of input contains a single integer  $\mathbf{T}$ , representing the number of test cases or queries to be run.

For each test case:

The **first line** contains a single integer  $\mathbf{N}$ , denoting the size of the array " $\mathbf{A}$ ".

The **second line** contains  $N$  single space-separated integers, elements of the array " $\mathbf{A}$ ".

The **third line** contains a single integer  $\mathbf{Q}$ , denoting the number of queries.

Then each of the **Q** lines of each test case contains two space-separated integers **L**, and **R** denoting the left and the right index of the infinite array “**B**”, whose sum is to be returned.

#### **Output Format:**

For each test case, print **Q** space-separated integers that denote the answers of the given **Q** queries. Print the answer to each test case in a separate line.

#### **Sample Input:**

```
1
3
1 2 3
2
1 3
1 5
```

#### **Sample Output :**

```
6 9
```

#### **Explanation:**

For this example, the given array **A** is [1, 2, 3] therefore the infinite array “**B**” will be [1, 2, 3, 1, 2, 3, 1, 2, 3, ...]. Therefore, the answer for the given query is 6 because the sum of the subarray from index 1 to 3 of infinite array “**B**”—that is, (**B[1]+B[2]+B[3]**) is 6.

For the second query, the answer is 9 because the sum of the subarray from index 1 to 5 of array “**B**”—that is, (**B[1]+B[2]+B[3]+B[4]+B[5]**) is 9.

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

#### **Steps:**

1. Instead of creating a new infinite array **B** which has a repeated array **A** elements in the form [**A<sub>1</sub>, A<sub>2</sub>, ..., A<sub>N</sub>, A<sub>1</sub>, A<sub>2</sub>, ..., A<sub>N</sub>, A<sub>1</sub>, A<sub>2</sub>, ..., A<sub>N</sub>, ...**], we will traverse array **A** from start to end repetitively. This means that when we reach **A<sub>N</sub>**, we will go back to **A<sub>1</sub>** and start our traversal again.
2. Therefore, using the brute force approach, we follow the following steps:
  - a. We run a loop from **L** to **R**.
  - b. For each index **i**, we add the value at index **i % N** of the array **A**—that is, **A[i % N]** to our sum.
  - c. This way, we find the sum of the required subarray from index **L** to **R** in an infinite array **B**.

#### **Example:**

Let us dry run this for the sample input given above. Our array **A** = [1, 2, 3] and indices **L** = 1 and **R** = 3. Clearly, **N** = 3. Hence in this case, sum will be

$$\text{sum} = \mathbf{A[1 \% 3]} + \mathbf{A[2 \% 3]} + \mathbf{A[3 \% 3]}$$

**sum = 2 + 3 + 1**

**sum = 6**

**Time Complexity:** In the worst case, we will be running a loop from L to R that takes **(O(R - L))** time. Thus a total of **O(R - L)** time will be required.

**Space Complexity:** **O(1)**, as in the worst case, only a constant space is required.

### Approach 2: Prefix Sum Array Approach

The better idea is first to create the **sum array** in which **sumArray[i]** stores the sum from **(A[0] to A[i])**. Now instead of iterating from **L** to **R** like in the previous approach, we find the sum using our **sumArray[]**.

Let's take an example, where array **A = [1, 2, 3]**, and we have to find the sum of the subarray of the infinite array (as shown in below fig) from index 3 to 10.



#### Steps:

1. What can be clearly seen is that iterating from index 3 to 10 would lead to traversal of array **A** again and again. So, instead of doing this, we can first find the sum from index 0 to index 10, say **a**, and then find the sum from index 0 to 2, say **b**, and subtract **b** from **a** as **a - b**. This would give us the sum of subarray from index 3 to 10 in an infinite array **B**.
2. Now to find the sum, from index 0 to any index X, we first find how many times the given array **A** can occur completely up till the index X.
3. This can be simply found by **X / N**. Let variable **count = X / N**. Thus, the sum will be **count \* sumArray[N]**, where **N** is the length of the array **A**.
4. Now for the remaining part of the subarray, sum can be found by **sumArray[ (X % N) ]**.

Hence, using this way, we can find the sum without iterating from **L** to **R**.

**Time Complexity:** **O(Q + N)** per test case, where **N** is the size of the given array, and **Q** is the number of given queries. In the worst case, we will traverse the given array, **O(N)**, only one time, and for each query **O(Q)**, we will be calculating the sum in constant time. Thus a total of **O(N+Q)** time will be required.

**Space Complexity:** **O(N)**, where **N** is the size of the given array. In the worst case, we will be storing the prefix sums in an array of size **N**. Thus a total of **O(N)** space will be required.

### 3. Flip Bits [\[https://coding.ninja/P3\]](https://coding.ninja/P3)

**Problem Statement:** You are given an array **ARR** of size **N** consisting of 0s and 1s. You have to select a subset and flip the bits of that subset. You have to return the count of maximum 1s that you can obtain by flipping the chosen subarray at most once.

A flip operation is an operation in which you turn 1 into 0 and 0 into 1.

For example, if you are given an array [1,1,0,0,1], then you will have to return the count of maximum 1s you can obtain by flipping any one chosen sub-array at most once. It is clear in the mentioned example that you will have to choose a subarray from index '2' to index '3'. So, the final array comes out to be [1,1,1,1,1], which contains five 1s. Hence, your final answer will be 5.

#### **Input Format:**

The **first line** of input consists of a single integer **T** denoting the total number of the test case.

For each test case:

The **first line** contains an integer **N**, which represents the array's size.

The **second line** contains **N** space-separated integers representing the array elements.

#### **Output Format:**

For each test case, return a single integer representing the maximum number of 1's you can have in the array after at most one flip operation.

#### **Sample Input:**

1

5

1 0 0 1 0

#### **Sample Output :**

4

#### **Explanation:**

For the given test case, we can perform a flip operation in the range [1,2]. After the flip operation, the array is [1 1 1 0], thus the answer will be 4.

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

The idea is to check for all the possible subarrays and inside each subarray, check for the highest value of the difference between the count for zeroes and ones for this. Let's consider this highest difference to be **MAX** and initialize it to zero. So, formally **MAX** = (Count of zeros in the subarray - count of ones in the subarray).

#### **Steps:**

1. Initialize **TOTALONES** to zero, which will store the total count of ones in the array.
2. Now start two nested loops, the outer one starting from index  $i = 0$  and the inner one from index  $i$ , both running till the end of the array.
3. Inside the loop—if you encounter one—increase **TOTALONES** by 1.
4. Initialize **COUNTONE** and **COUNTZERO** to zero, which will store the count of one and zero, respectively.
5. Consider all subarrays starting from  $i$  and find the difference between 1s and 0s.

Update **MAX** on every iteration to store the answer.

6. Finally, return the count of all the ones in the array by the sum **TOTALONES + MAX**.

**Time Complexity:**  $O(N^2)$ , where '**N**' denotes the number of elements in the array as we are using two nested loops.

**Space Complexity:**  $O(1)$ , since we are using constant space.

### Approach 2: Kadane's Algorithm

This problem can be interpreted as a version of Kadane's Algorithm.

We want to consider a subarray that maximizes the difference between the count of ones and zeroes. If we change 1s to -1s and change 0s to 1s, then the sum of values will give us the maximum difference between the counts(**MAX**). So, we have to find a subarray with the maximum sum, which can be done by **Kadane's Algorithm**. Finally, we return the **MAX** plus count of ones in the original array.

#### Steps:

1. Initialize **MAX** and **CURRMAX** to zero; these variables store overall max diff for any subarray and the current difference in the subarray respectively.
2. Initialize **TOTALONES** to zero, which will store the total count of ones in the array.
3. Now run a loop from index  $i = 0$  and increment **TOTALONES** by one if one is encountered.
4. Consider the value of 1 as -1 and 0 as 1—that is,  $\text{val} = (\text{ARR}[i] == 1)? -1 : 1$ .
5. Update **CURRMAX** and **MAX**—that is,  $\text{CURRMAX} = \text{Math.max}(\text{VAL}, \text{CURRMAX} + \text{VAL})$ ; and  $\text{MAX} = \text{Math.max}(\text{MAX}, \text{CURRMAX})$ .
6. Finally, return the count of all the ones in the array by returning the **TOTALONES + MAX**.

**Time Complexity:**  $O(N)$ , where '**N**' denotes the number of elements in the array since we are traversing the array once.

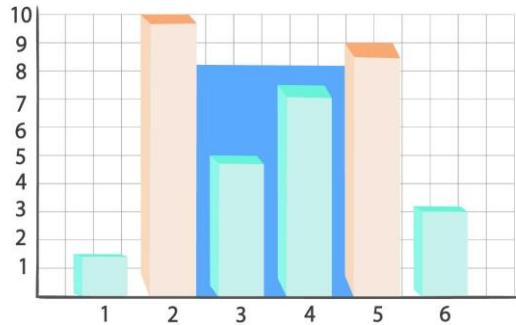
**Space Complexity:**  $O(1)$ , we are using constant space.

## 4. Container with Most Water [<https://coding.ninja/P4>]

**Problem Statement:** Given a sequence of ' $n$ ' space-separated non-negative integers **a[1], a[2], a[3],...a[i].....a[n]**, where each number of the sequence represents the height of the line drawn at the point  $i$ . Hence on the cartesian plane, each line is drawn from coordinate  $(i,0)$  to coordinate  $(i, a[i])$ , here  $i$  ranges from 1 to ' $n$ '. Find two lines, which, together with the x-axis, form a container, such that the container contains the most area of water.

**Note:** The container would be perfectly vertical—that is, the height of the water is equal to the minimum height of the two lines which define the container.

### Example:



For the above diagram, the first red-marked line is formed between coordinates (2,0) and (2,10), and the second red-marked line is formed between coordinates (5,0) and (5,9). The area of water contained between these two lines is  $(\text{height} * \text{width}) = (5 - 2) * 9 = 27$ , which is the maximum area contained between any two lines present on the plane. So in this case, we will return  $3 * 9 = 27$ .

### Input Format:

The **first line** of input contains an integer **T** denoting the number of test cases.

The **next  $2 * T$  lines** represent the **T** test cases. For each test case:

The **first line** contains the number of elements in the sequence.

The **second line** contains ' $n$ ' space-separated integers, which is the given sequence.

### Output Format:

Return the area of the container, which can hold the maximum amount of water using any pair of lines from the given sequence.

### Sample Input:

```
1
5
4 3 2 1 4
```

### Sample Output :

```
16
```

### Explanation:

We can create ' $n(n+1)/2$ ' different containers using  $n$  lines. Suppose we are considering lines  $i$  and  $j$ . To find the area, we shall use the following formula:

$$(j - i) * \min(a[i], a[j])$$

For instance, in the given example, with the use of 1<sup>st</sup> and 3<sup>rd</sup> line, we can create a container with area  $= (3 - 1) * \min(4, 2) = 4$ .

Let us look at all such possible containers.

Lines used	Area obtained
------------	---------------

4,3	area = min(4,3) * 1 = 3
4,2	area = min(4,2) * 2 = 4
4,1	area = min(4,1) * 3 = 3
4,4	area = min(4,4) * 4 = 16
3,2	area = min(3,2) * 1 = 2
3,1	area = min(3,1) * 2 = 2
3,4	area = min(3,4) * 3 = 9
2,1	area = min(2,1) * 1 = 1
2,4	area = min(2,4) * 2 = 4
1,4	area = min(1,4) * 1 = 1

But among all such containers, the one with the maximum area will be formed by using the first and last line, the area of which is  $(5 - 1) * \min(4,4) = 16$ .

Hence, the output will be 16.

### Approach 1: Brute Force Approach

Since we need to find the container with the most water, let us try to find all possible containers and choose the one which has the maximum area.

So how can we find the area of all possible containers?

Let  $i$  and  $j$  represent points on the horizontal axis. For each line with the position,  $i$  find another line  $j$ , such that  $j > i$ .

And to find the amount of water contained, we can use the formula,  $(j - i) * \min(a_i, a_j)$ , where  $a_i$  and  $a_j$  represents the height of the lines at index  $i$  and  $j$ , respectively. We can do this in the following way:

#### Steps:

1. Take a variable **globalMax** to store the overall maximum area and initialize it to some very small value.
2. Let  $n$  be the number of lines in the sequence.
3. Let **height[i]** be the height of the first line of any container. Then for each  $i$ , we find **height[j]** where  $i < j < n-1$ , which is the height of the second line for the container. We compute its current area in a variable **currentArea** which is  $(j - i) * \min(\text{height}[j], \text{height}[i])$ , and compare with the maximum area—that is, **globalMax** achieved till now. If the **currentArea** is greater than **globalMax**, we update the maximum area.
4. In the end, we return the value of **globalMax**.

**Time Complexity:**  $O(N^2)$ , where **N** denotes the number of elements in the given sequence.

For each starting height, we calculated the area for the container. This is done for all **N** lines in the container, and in the worst case, this is done **N** times. Hence the complexity is of order  $N^2$ .

**Space Complexity:**  $O(1)$ , we are using constant space.

### Approach 2: Two Pointers Approach

We know that  $\text{area} = \text{width} * \text{length}$ . So, increasing the length of line or width between two lines will increase the area. If we start with the two extremes of the sequence—the first and last line, we can achieve maximum width. But this does not ensure that the area is maximum as it is possible that some pair of lines with a lesser width and much greater length than the current extreme lengths can have a greater area. Hence, it is obvious that if we decrease the width, we need to find such a pair of lines that the length of the shorter line is more than the maximum of the previous pair of lines.

Keeping all that in mind, we can use the following strategy to find the maximum area:

**Steps:**

1. Let the ' $i^{\text{th}}$ ' line be used to denote the first line of the container, and the ' $j^{\text{th}}$ ' line be used to denote the second line and ' $n$ ' be the number of lines in the sequence.
2. We start from the widest container—that is,  $i = 1$  and  $j = n$ . Now for a better answer with a lesser width, it is necessary that the height is greater than the current heights.
3. Now discard the height, which is lower among the two heights  $i$  and  $j$  because having a greater height will allow us to have a better answer. If it is  $i$ , then move the  $i^{\text{th}}$  pointer to the right and if it is  $j$  move it to the left.
4. Calculate the area of the new container formed and update the maximum area if the newly found area is greater than the previous maximum area.
5. Do this until  $i$  is less than  $j$ .
6. In the end, we return the maximum area found at the end of the loop.

**Time Complexity:**  $O(n)$ , where  $n$  denotes the number of elements in the given sequence. Since we traverse the sequence once to find the container with maximum water, the time complexity is the order of  $n$ .

**Space Complexity:**  $O(1)$ , we are using constant space.

## **5. Overlapping Intervals** [<https://coding.ninja/P5>]

**Problem Statement:** You have been given the start and end times of  $N$  intervals. Write a function to check if any two intervals overlap with each other.

**Note:**

If an interval ends at time  $T$  and another interval starts at the same time, they are not considered overlapping intervals.

**Input Format:**

The **first line** contains an integer  $t$  which denotes the number of test cases or queries to be run.

For each test case

The **first line** or query contains an integer '**N**' representing the total number of intervals.

The **second line** contains '**N**' space-separated integers representing the starting time of the intervals.

The **third line** contains '**N**' space-separated integers representing the end time of the intervals.

#### **Output Format:**

For each test case, return true if overlapping intervals are present. Otherwise, return false.

Output for every test case will be printed in a separate line.

#### **Sample Input:**

```
1
3
1 2
2 2
3 4
```

#### **Sample Output :**

```
false
```

#### **Explanation:**

Here the first interval starts at 1 and ends at 2. The second interval starts and ends at 2 itself, and the third interval starts at 3 and ends at 4. Hence, none of the intervals overlap each other. Thus, the output is false.

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

#### **Steps:**

1. Iterate over the intervals one by one.
2. For every interval, check if the end value of one interval is less than the start value of the other interval.
3. If this condition fails, the intervals are overlapping. Therefore return false.
4. Otherwise, return true.

**Time Complexity:**  $O(N^2)$ , where **N** is the total number of intervals.

For every interval, we check the remaining ( $N - i - 1$ ) intervals where  $i$  is the current interval. Thus, we would end up with quadratic complexity.

**Space Complexity:**  $O(1)$  since we don't use any extra space.

### Approach 2: Sorting Approach

#### Steps:

1. Sort the list of intervals first on the basis of their start time and then iterate through the array
2. If the start time of an interval is less than the end of the previous interval, then there is an overlap, and we can return true.
3. If we iterate through the entire array without finding an overlap, we can return false.

**Time Complexity:**  $O(N \log N)$ , where  $N$  is the total number of intervals. Sorting the list would take  $O(N \log N)$  operations. Later, iterating over the list again to find the overlap would take  $N$  operations. Hence the overall time complexity boils down to  $O(N \log N)$

**Space Complexity:**  $O(N)$ , where  $N$  is the total number of intervals.

Sorting takes  $O(N)$  extra space.

## 6. Pair Sum [<https://coding.ninja/P6>]

**Problem Statement:** Given an integer array **arr** of size **N** and an integer **S**. Return the list of all pairs of elements such that for each pair, the sum of elements equals **S**.

#### Note:

Each pair should be sorted—that is, the first value should be less than or equal to the second value. Return the list of pairs sorted in non-decreasing order of their first value. In case if two pairs have the same first value, the pair with a smaller second value should come first.

#### Input Format:

The **first line** contains two space-separated integers **N** and **S**, denoting the size of the input array and the value of **S**.

The **second line** contains **N** space-separated integers, denoting the elements of the input array: **arr[i]** where  $0 \leq i < N$ .

#### Output Format:

Print **C** lines. Each line contains one pair—that is, two space-separated integers, where **C** denotes the count of pairs having sum equal to the given value **S**.

#### Sample Input:

```
5 5
1 2 3 4 5
```

### **Sample Output:**

1 4

2 3

### **Explanation:**

The sum of the 1st and the 4th element (1 and 4) and the 2nd and the 3rd element (2 and 3) is 5.

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

#### **Steps:**

1. Initialize a list to store our results.
2. For each element in the array **arr[i]**,  $0 \leq i < N$ , check if  $(\text{arr}[i] + \text{arr}[j])$ , equals to given sum or not, where  $i < j < N$ .
3. If the condition matches, add the **pair(arr[i], arr[j])** to the list.
4. Sort the list of pairs as per the given output format and return this list.

**Time Complexity:**  $O(N^2)$ , where **N** is the size of the input array.

In the worst case, for each element, we will be checking all other elements in the array.

**Space Complexity:**  $O(1)$  in the worst case, as constant extra space is required.

### **Approach 2: Using HashMap**

An efficient way to solve this problem is by using the technique of hashing. We will use a hashmap to check for a particular **arr[i]** if there exists any **S - arr[i]**, which on adding gives us the desired sum **S**.

#### **Steps:**

1. Initialize a list to store our results and a Hashmap to store the count.
2. For each element in the array **arr[i]**,  $0 \leq i < N$ , we will check whether there exists an element equals to  $(S - \text{arr}[i])$  already in the map.
3. If it exists, we will add the **pair(arr[i], S - arr[i])** count number of times to the list, where count represents the frequency of  $(S - \text{arr}[i])$  in the map.
4. Increment the frequency of the current element (**arr[i]**) in the map.
5. Sort the list of pairs as per the given output format and return this list.

**Time Complexity:**  $O(N)$ , where **N** is the number of elements in the array.

In the worst case, for each element, we will be adding all the pairs formed by the elements in the array.

**Space Complexity:**  $O(N)$ , where **N** is the number of elements in the array.

In the worst case,  $O(N)$  extra space is required for the hashmap to store the frequency of each element.

## **7. Sort 0 1 2 [<https://coding.ninja/P7>]**

**Problem Statement:** You have been given an integer array/list (**ARR**) of size **N**. It only contains 0s, 1s, and 2s. Write a solution to sort this array/list.

**Note :**

Try to solve the problem in '**Single Scan**'. ' Single Scan' refers to iterating over the array/list just once, or to put it in other words, you will be visiting each element in the array/list just once.

**Input Format:**

The **first line** contains an integer **T** which denotes the number of test cases or queries to be run.  
For each test case:

The **first line** contains an integer **N** denoting the size of the array/list.

The **second line** contains **N** space-separated integers denoting the array/list.

**Output Format:**

For each test case/query, print the sorted array/list (**ARR**) as space-separated Integers.

Output for every test case will be printed in a separate line.

**Sample Input:**

```
1
6
0 1 2 2 1 0
```

**Sample Output :**

```
0 0 1 1 2 2
```

**Explanation:**

The given array is [0, 1, 2, 2, 1, 0] and sorted array is [0, 0, 1, 1, 2, 2]

**Approach 1: Sorting**

1. Use any good sorting algorithm like Merge Sort, Quick Sort, or inbuilt sorting function of different languages.
2. Sort the array and just return.

**Time Complexity:  $O(N \log N)$** , where **N** is the size of the array.

We are using an inbuilt sort algorithm which has Overall Time Complexity  **$O(N \log N)$**

**Space Complexity:  $O(1)$** , as we are using constant space.

**Approach 2: Frequency of 0,1 and 2**

Note that this approach uses two scans to achieve the solution. We cannot use it when we have to specifically solve the problem in a single scan.

We can exploit the property that we will only have 0s, 1s, and 2s in our array.

#### Steps:

1. Count the frequency of 0s, 1s, and 2s in the array.
2. Start from  $i = 0$  and first fill the number of 0s present in the array.
3. Then fill in the number of 1s present in the array.
4. Then fill in the number of 2s present in the array.
5. Finally, we have the sorted array.

**Time Complexity:  $O(N)$** , where **N** is the size of the array. We are iterating over the array 2 times. First time counting the frequency of 0s 1s and 2s and then for filling the array. So the overall time complexity is  **$O(N)$** .

**Space Complexity:  $O(1)$** , as we are using constant space.

#### Approach 3: Three Pointer Approach

We'll use a three-pointer approach to solve this problem.

#### Steps:

1. The three-pointers will be **current**, **zeroPos**, and **twoPos**.:
  1. **current**: This will hold the position of the current element that we're on during the iteration of the array. This will be initialized to zero.
  2. **zeroPos**: This will hold the index where we will push any 0s that we may encounter. This will be initialized to zero.
  3. **twoPos**: This will hold the index where we will push any 2s that we may encounter. This will be initialized to  $n - 1$ , where  $n$  is the size of the array.
2. We'll iterate through the array using the current pointer. Every element is either 0, 1, or 2, so let's see what we'll be doing in each of these cases:
  1. If **arr[current] = 0**: In this case, we need to push the element towards the front of the array. To do that, we can swap **arr[current]** and **arr[zeroPos]**, then we will increase both current and zeroPos by one.
  2. If **arr[current] = 1**: In this case, we will just increase the current by one since we are only concerned with pushing 0s to the front and 2s to the end of the array.
  3. If **arr[current] = 2**: In this case, we need to push the element towards the end of the array. Again, to do this, we'll just swap **arr[current]** and **arr[twoPos]**. We will decrease **twoPos** by one. However, in this case, we will not increase the current by one.

#### Note:

1. What will be the condition that must be satisfied so that our loop can end? You might think that it's when current reaches the end of the array, but that's not the case here. Let's see why. Can you see what exactly the two pointers, **zeroPos**, and **twoPos** are doing? As we go through the array, every element before **zeroPos** is a 0, and every

element after **twoPos** is a 2. Also, every element after **zeroPos** but before the current is a 1. Therefore, all these elements are 'sorted'. The element that remains to be sorted is the ones that lie between the indices current and **twoPos**. Therefore our loop will terminate when the current reaches the value of **twoPos**.

- Now, let's understand why we can't increase the value of current when **arr[current]** = 2. When we swap **arr[current]** with **arr[twoPos]**, we don't know what value was initially at index **twoPos** (before the swap happened), it could be any of the values 0, 1, or 2. So, we can't increase the value of current without checking what value was swapped with **twoPos**. We didn't have to worry about this in the case where we were swapping **arr[current]** with **arr[zeroPos]** because then we would always be swapping 0 and 1.

**Time Complexity:** **O(N)**, where **N** is the size of the array. We are only doing a single pass of the array. So the overall Time Complexity is **O(N)**.

**Space Complexity:** **O(1)**, as we are using constant space.

## **8. Search In Rotated Sorted Array [<https://coding.ninja/P8>]**

**Problem Statement:** Aahad and Harshit always have fun by solving problems. Harshit took a sorted array and rotated it clockwise by an unknown amount. For example, he took a sorted array = [1, 2, 3, 4, 5] and if he rotates it by 2, then the array becomes: [4, 5, 1, 2, 3].

After rotating a sorted array, Aahad needs to answer **Q** queries asked by Harshit, each of them is described by one integer **Q[i]** which Harshit wanted him to search in the array. For each query, if he finds it, he had to shout the index of the number, otherwise, he had to shout -1.

### **Input Format:**

The first line contains the size of the array: **N**

The second line contains **N** space-separated integers: **A[i]**.

The third line contains the number of queries: **Q**

The next **Q** lines contain the number which Harshit wants Aahad to search: **Q[i]**

### **Output Format:**

For each test case, print the index of the number if found, otherwise -1.

Output for every test case will be printed in a separate line.

### **Sample Input:**

4

2 5 -3 0

2

5

0

### **Sample Output:**

1  
3

### Explanation:

5 is present at index 1 while 0 is present at index 3 in the given array.

### Approach 1: Brute Force Approach

The idea here is to do a linear approach which apparently is a brute force way to do this.

#### Steps:

1. Visit every element one by one.
2. Check if the current element that you are looking at is the value that needs to be searched. If the element is found, return the index at which you find it.
3. Once all the elements are visited and you don't find the value, return -1

**Time Complexity: O(N)**, where **N** is the total number of elements in the array.

In the worst case, we have to visit every element.

**Space Complexity: O(1)**, since a constant amount of space is used.

### Approach 2: Binary Search

Before we discuss the algorithm, there's an interesting property about sorted and rotated arrays that must be noted.

#### Example:

If we divide the array into two halves, at least one of the two halves will always be sorted.

Let's consider the array: [5, 6, 7, 1, 2, 3, 4]

5 6 7 1 2 3 4

**Mid-index** is calculated as **startIndex + (endIndex - startIndex) / 2**.

The **mid-index** for the above-depicted list/array will be 3.

The value at the **mid-index** is 1. If we have a closer look at three values, that are, value at start, end, and mid then we can deduce which subarray is sorted or not.

Since the value at the mid-index is less than the value at the start index, we clearly can say that the left subarray is not sorted or violates the property of a sorted array.

Similarly, we can deduce if the right subarray is sorted or not by comparing the values at **mid-index** and end index.

Now, once we know what part of the array is sorted, we can compare the value(key) to be searched in reference to the sorted subarray.

#### Steps:

1. Find the **mid-index**
2. If the value (key) being searched for is at the mid index, then return the mid index.
3. Compare values at **startIndex**, **endIndex**, and **mid-index**:

1. If the left subarray is sorted, check if the value (key) to be searched lies in the range:
  1. If it does, then search space reduces to **[startIndex, (mid-index - 1)]**.
  2. Otherwise, the search space reduces to **[(mid-index + 1), endIndex]**
2. If the right subarray is sorted, check if the value(key) to be searched lies in the range:
  1. If it does, then search space reduces to **[(mid-index + 1), endIndex]**.
  2. Otherwise, the search space reduces to **[startIndex, (mid-index - 1)]**
4. Repeat from step -1 until the key is found.
5. Return -1 if the key is never found

**Time Complexity:**  $O(\log(N))$ , Where **N** is the total number of elements in the array.

Every time we split the array into two halves.

**Space Complexity:**  $O(1)$ , since a constant amount of space is used.

## 9. Find All Triplets with Zero Sum [<https://coding.ninja/P9>]

**Problem Statement:** You are given an array **arr** consisting of **n** integers. You, need to find all the distinct triplets present in the array, which adds up to zero.

An array is said to have a triplet **{arr[i], arr[j], arr[k]}** with 0 sum if there exists three indices *i*, *j* and *k* such that  $i \neq j, j \neq k$  and  $i \neq k$  and  $\text{arr}[i] + \text{arr}[j] + \text{arr}[k] = 0$ .

### Note:

1. You can return the list of values in any order. For example, if a valid triplet is {1, 2, -3}, then (2, -3, 1), (-3, 2, 1) etc is also a valid triplet. Also, the ordering of different triplets can be random—that is, if there are more than one valid triplets, you can return them in any order.
2. The elements in the array need not be distinct.
3. If no such triplet is present in the array, then return an empty list, and the output printed for such a test case will be "-1".

### Input Format:

The **first line** contains an integer **T**, denoting the number of test cases.

For each test case:

The **first line** contains the integer **N**, denoting the size of the array.

The **second line** contains **N** space-separated integers denoting the array elements.

### Output Format:

For each test case, every line of output contains three spaced integers which correspond to the elements which add to zero.

### Sample Input:

```
1
5
-10 5 5 -5 2
```

#### Sample Output:

```
-10 5 5
```

#### Explanation:

-10 5 5 is the only triplet that adds up to zero. Note that the order of output does not matter, so 5 -10 5 or 5 5 -10 are also acceptable.

### Approach 1: Brute Force Approach

#### Steps:

1. The most trivial approach would be to find all triplets of the array and count all such triplets whose **sum = 0**.
2. We can find the answer using three nested loops for three different indices and check if the sum of values at those indices adds up to zero.
3. We will then create a set to keep the track of triplets we have visited. Run the first loop from  $i = 0$  to  $i = n - 3$ , second loop from  $j = i + 1$  to  $j = n - 2$  and the third loop from  $k = j + 1$  to  $k = n - 1$ .
4. Check **if  $\text{arr}[i] + \text{arr}[j] + \text{arr}[k] = 0$** 
  - If the triplet is not present in the set, then print the triplet and insert triplet into the set since we need distinct triplets only.
  - Else continue.

**Time Complexity:**  $O(N^3)$ , where N is the number of elements in the array.

**Space Complexity:**  $O(1)$ , as we are using constant extra memory.

### Approach 2: Sorting and Two Pointers Approach

#### Steps:

1. Sort the array in non-decreasing order because after the array is sorted, we don't have to process the same elements multiple times and hence we don't have to explicitly keep track of distinct triplets.
2. Now since we want triplets such that  $x + y + z = 0$ , we have  $x + y = -z$ . Let us fix z as **arr[i]**. Hence we want to find two numbers **x and y**, such that their sum is equal to  $-\text{arr}[i]$  in the array.
3. Let us assume that we are at the ith index of the array and initialize variable target to  $-\text{arr}[i]$ .
4. So now we just need to find two elements x, y such that **target = x + y**.
5. We will use two pointers, one will start from  $i+1$ , and the other will start from the end of the array.

6. Let the two pointers be front and back, where **front = i + 1** and **back = n - 1**. Let sum = x + y, where x = arr[front] and y = arr[back]. We have to find the triplets such that **target = sum**. To do that, we will run a loop **while front < back**. This loop will involve three cases:
  - a. **if(sum < target)**, we will have to increase the sum and hence increment the front pointer.
  - b. **Else if(sum > target)**, we will have to decrease the sum and hence decrease the back pointer.
  - c. **Else** print the triplet and since we want distinct triplets, do the following.
    - i. Increment the front pointer until **arr[front] = x and front < back**.
    - ii. Decrement the back pointer until **arr[back] = y and front < back**.
7. While **arr[i] = arr[i+1]**, keep on incrementing i.

**Time Complexity:**  $O(N^2)$ , where **N** is the number of elements in the array.

For every possible candidate for target, we can find if there are valid x and y in  $O(N)$  time. Thus the complexity will be  $O(N*N)$ .

**Space Complexity:**  $O(1)$ , as we are using constant extra space.

## 10. Majority Element [<https://coding.ninja/P10>]

**Problem Statement:** You have been given an array/list ARR consisting of N integers. Your task is to find the majority element in the array. If there is no majority element present, print -1.

Note: A majority element is an element that occurs more than  $\text{floor}(N/2)$  times in the array.

### Input Format:

The **first line** contains an integer **T** representing the number of test cases or queries to be processed.

For each test case:

The **first line** contains a single positive **N** representing the size of the array/list.

The **second line** contains **N** space-separated integers representing the array elements.

### Output Format:

For each test case, print the majority element.

Print the output of each test case in a separate line.

### Sample Input:

```
1
5
2 3 9 2 2
```

### Sample Output:

```
2
```

### **Explanation:**

Frequencies of occurrences of different elements are  $2 \rightarrow 3$  times,  $3 \rightarrow 1$  time,  $9 \rightarrow 1$  time. As  $2$  occurs more than  $\text{floor}(5/2)$  ( $= \text{floor}(2.5) = 2$ ) times, it is the majority element.

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

#### **Steps:**

1. First, traverse the array and count the frequency of each element.
2. We will run two nested loops till  $N$  and store the count of each array element in **maxCount**. If for any element, **maxCount** becomes greater than  $N/2$ , we will return that element as the majority element.
3. If no majority element is found, we will return -1.

**Time Complexity:**  $O(N^2)$ , where  $N$  denotes the size of the given array/list. We are running two nested loops till ' $N$ ' to count the frequency of each element. So, the overall time complexity is  $O(N^2)$ .

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

### **Approach 2: Using HashMap**

#### **Steps:**

1. We will maintain a hashmap to store element-frequency pairs.
2. We will traverse the array and store the frequency of each element in the hashmap.
3. If the frequency of any element becomes greater than the **floor( $N/2$ )**, we will return it as the majority element.
4. If no majority element is found, we will return -1.

**Time Complexity:**  $O(N)$ , where  $N$  denotes the size of the given array/list. We are traversing the array exactly one time. Hence, the time complexity is **linear**.

**Space Complexity:**  $O(N)$ , where  $N$  denotes the size of the given array/list. We are storing the elements of the array and their respective frequencies in a hashmap. In the worst case, when the array contains ' $N$ ' distinct elements, then the hashmap takes  $O(N)$  space.

### **Approach 3: Boyer-Moore Majority Vote Algorithm**

We can find the majority element in linear time and constant space using **Moore's voting algorithm**. It is based on the fact that since the majority element occurs more than  $\text{floor}(N/2)$  times, its frequency will be greater than the combined frequencies of all other elements. The algorithm gives the correct answer only if the majority element exists in the array. So, in the end, we have to check the frequency of the majority element to confirm.

#### **Steps:**

1. We will maintain '**majorityElement**' to keep track of the possible candidate of the majority element.
2. We will initialize '**count**' to zero to store the count of **majorityElement**.
3. Loop through array elements.
  1. If **count** = 0:
    1. We set **majorityElement** to the current element, set count to one, and continue iterating.
  2. Else:
    1. If the current element is equal to the **majorityElement**, we increment the count by 1.
    2. Else, we decrement the count by one.
4. Now, we again traverse through the array and find the count of **majorityElement**.
5. If the count is greater than  $\text{floor}(N/2)$ , we return the **majorityElement** as the answer. Else, we return -1.

**Time Complexity:**  $O(N)$ , where **N** denotes the size of the given array/list. Hence, the time complexity is linear. We are traversing the array exactly two times. So, the overall time complexity is  $O(N)$ .

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

# 2. Multidimensional Arrays

---

## Introduction to Multidimensional Arrays

An array with more than one dimension is called a **multidimensional array**. A *multidimensional array* identifies each element in the array with multiple indexes.

### Two-dimensional Array

Imagine a classroom that has a seating arrangement of 5 by 5. It means that the seats are arranged in five rows of five columns each. This arrangement can be represented through a two-dimensional array. Two-dimensional arrays use two indices for each element of the array. A two-dimensional array is often called a *table* or *matrix*.

Y: 0	0	0	0	0	0
1	0	0	0	0	0
2	0	0	1	0	0
3	0	1	1	1	0
4	1	1	1	1	1

X: 0    1    2    3    4

We can observe that each row of the matrix is an array in itself. Thus in the above example, the given two-dimensional arrays consist of five 1-dimensional arrays.

There is no upper limit to the number of dimensions that an array can have. However, due to memory constraints, arrays up to three dimensions are usually used. It is very easy for us to conceptualise three-dimensional arrays as well. Let us extend the previous example.

### Three-dimensional Array

Imagine that the classroom discussed in the previous example has two more floors, this gives a 3D array of size  $5 \times 5 \times 3$ .

Z: 0	0	0	0	0	0
1	0	0	0	0	0
2	0	0	1	0	0
3	0	1	1	1	0
4	1	1	1	1	1

Y: 0    1    2    3    4

X: 0    1    2    3    4

## Properties of Multidimensional Arrays

- Similar to 1D arrays, the elements of a 2D array are stored in contiguous memory locations, where each element of the array occupies a fixed memory size. For example, for integers—each element occupies four bytes.

An example of a **2D array** of size  $3 \times 5$  is:

	0	1	2	3	4
0	a[0] [0]	a[0] [1]	a[0] [2]	a[0] [3]	a[0] [4]
1	a[1] [0]	a[1] [1]	a[1] [2]	a[1] [3]	a[1] [4]
2	a[2] [0]	a[2] [1]	a[2] [2]	a[2] [3]	a[2] [4]

- The elements in the arrays are represented using **N** indices. In the case of a 2D array, the first number in brackets is the number of rows, and the second number in brackets is the number of columns. The first element (top-left) of any grid would be element [0][0]. The element to its right would be [0][1], the element below it would be [1][0], and so on.

## Accessing Array Elements

- The elements of the array are accessed by using their index. The row index of a 2D array of size  $\mathbf{N} \times \mathbf{M}$  ranges from **0** to **N - 1**. Similarly, the column index ranges from **0** to **M - 1**. For example, the array[5][7] is the element present at the sixth row and eighth column in the given array.
- Every array is identified by its **base address**—that is, the location of the first element of the first row of the array in memory. The base address helps in identifying the address of all the elements of the array. Since the elements of an array are stored in **contiguous memory locations**, the address of any element can be accessed from the base address itself.

For example, 200 is the base address of the entire array. If the number of columns in the array is equal to 10 then the address of the element stored at the index Array[5][7] is equal to  $200 + (5 * 10 + 7) * (\text{size of(int)}) = 428$ .

## Applications of Multidimensional Arrays

- Multidimensional arrays are used to store the data in a tabular form. For example, storing a student's **roll number and marks** can be easily done using multidimensional arrays. The first dimension would store the roll number, and the second dimension would store the marks. Such an array may look like: [[1, 25], [2, 45], [3, 68], [4, 10]]. In this

example, we have stored the data of four students and their marks. Another common usage is to **store the images in 3D arrays**.

- In **dynamic programming questions**, multidimensional arrays are used to **represent the states** of the problem. If we have one parameter that describes the state, for example, Fibonacci sequence, we use a **1D array**. Because  $\text{arr}[0]$  represents the value of the function for state 0,  $\text{arr}[1]$  represents the value of the function for state 1 and so on. If we have  $k$  parameters that represent the state of the function, then we have a  $k$ -dimensional **array**. Here  $\text{arr}[\mathbf{x}_1][\mathbf{x}_2][\mathbf{x}_3]..\,[\mathbf{x}_k]$  represents the value of the function for the state  $(\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \dots, \mathbf{x}_k)$ .
- Apart from these, they also have applications in many standard algorithmic problems like:
  - Matrix Multiplication
  - Adjacency matrix representation in graphs
  - Grid search problems

## Time Complexity of Various Operations

- **Accessing elements:** Since elements in an array are stored at contiguous memory locations, they can be accessed efficiently (random access) in **O(1)** time using indices.
- **Finding elements:** Finding an element in an array takes **O(N \* M)** time in the worst case—where  $N$  is the number of rows of the array and  $M$  is the number of columns of the array—as it may involve traversing the entire array.

---

## Practice Problems

### 1. Matrix Is Symmetric [<https://coding.ninja/P11>]

**Problem Statement:** You are given a square matrix, print whether the matrix is symmetric or not.

A symmetric matrix is that matrix whose transpose is equal to the matrix itself.

Example of a symmetric matrix:

$$\begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 5 \\ 3 & 5 & 8 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 5 \\ 3 & 5 & 8 \end{bmatrix}^T$$

*Symmetric matrix*

### **Input Format :**

The **first line** contains an Integer **t** which denotes the number of test cases or queries to be run.

For each test case:

The **first line** contains the size of the square matrix **N**.

Each test case's **second and final line contains** the **N \* N** Integers separated by a single space (the matrix is entered row-wise).

### **Output Format :**

For each test case/query, print 'Yes' if the given matrix is symmetric otherwise, print 'No'.

### **Sample Input:**

1

3

1 2 3 2 4 5 3 5 8

### **Sample Output:**

Yes

### **Explanation:**

We can see that the matrix, when transposed, is equal to the original matrix. Therefore the answer is 'Yes'.

$$\begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 5 \\ 3 & 5 & 8 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 5 \\ 3 & 5 & 8 \end{bmatrix}^T$$

*Symmetric matrix*

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

The simplest way to verify that a matrix is symmetric or not is by creating the transpose of the matrix and checking if the transpose and the given matrix are the same or not.

#### **Steps:**

1. Create another matrix of the same size, initially having all zeros.
2. Replace the new matrix( $i,j$ ) with the old matrix( $j,i$ ) and traverse both matrices element by element. If anywhere there is inequality return false, else continue.
3. In the end, return true as the entire matrix is traversed.

**Time Complexity: O(N<sup>2</sup>)**, as every element is visited once and there are **N<sup>2</sup>** elements in total, where **N** is the size of the square matrix.

**Space Complexity: O(N<sup>2</sup>)**, as an extra matrix of size **N \* N** is created while making the transpose of the matrix, where **N** is the size of the square matrix.

## Approach 2: Optimised Approach

A more optimised way to check whether the matrix is symmetric or not is to compare matrix elements without creating any other matrix. This can be done by comparing **matrix[i][j]** with **matrix[j][i]**.

### Steps:

1. Traverse the matrix element by element and compare  $\text{matrix}(i,j)$  with  $\text{matrix}(j,i)$ .
2. If at any point inequality exists between them, return **False**, declare the result as 'No'. Otherwise, continue till all the elements are compared one by one.
3. When the loop continues till the end and doesn't return false at any point, return **True**. Declare the result 'Yes' as the matrix turns out to be symmetric.

**Time Complexity:**  $O(N^2)$ , as every element is visited once and there are  $N^2$  elements in total, where **N** is the size of the square matrix.

**Space Complexity:**  $O(1)$ , as constant space required for variables, where **N** is the size of the square matrix.

## 2. Summed Matrix [\[https://coding.ninja/P12\]](https://coding.ninja/P12)

**Problem Statement:** There is a matrix **M** of size **n \* n**. The value at every cell of the matrix is given as **M(i,j) = i + j**, where **M(i,j)** is the value of a cell, **i** is the row number, and **j** is the column number. Given an integer **q** as an input, return the number of cells having the value **q**.

Note: Assume the array follows **1-based indexing**.

### Input Format:

The **first line** of input contains an integer **T** denoting the number of test cases.

For each test case:

A line contains two space-separated integers, **n** and **q**—where **n** is the size of the matrix and **q** is the value we need to find in the matrix.

### Output Format:

For each test case, return the number of cells in the array which have **q** stored as a value.

### Sample Input:

1  
4 2

### Sample Output:

1

### Explanation:

The matrix we get after using **M(i,j) = i + j** is:

2 3 4 5  
3 4 5 6  
4 5 6 7  
5 6 7 8

The value 2 occurs only once—at [1][1]. Hence the result is 1.

Only cell (1,1) holds the value 2.

### Approach 1: Brute Force Approach

In this approach, we would use a variable **total** to store the number of cells having value q. Simply loop over the elements to find the answer.

#### Steps:

1. Create a variable **total** to store the number of cells that have the value **q**. Initialise **total** to 0.
2. Next, start two nested loops—the outer one starting from index  $i = 1$  and the inner one from index  $j = 1$ —both running until the array's end.
3. For every instance, where  $i + j = q$ , increase the **total** by one.
4. Return the variable **total**.

**Time Complexity:**  $O(N^2)$ , where **N** is the size of the array. Here we are nesting two loops—one is going from 1 to **N**, and the second is going from 1 to **N** for every  $i$  in the first loop—making the time complexity  $N^2$ .

**Space Complexity:**  $O(1)$ , as we use constant space.

### Approach 2: Mathematical Approach

Let's take the matrix:

2 3 4  
3 4 5  
4 5 6

Now we can see till 4 all the numbers are occurring (number - 1) times. And if we take 4 as the longest diagonal, the numbers on both sides at an equal distance have the same number of occurrences. So after the biggest diagonal, the occurrence is  $2 * n + 1 - q$ .

#### Steps:

1. If **q** is less than equal to  $1 + n$ . Then we return  $q - 1$  as the answer.
2. In all other cases, we will return  $2 * n - q + 1$  as the answer.

**Time Complexity:**  $O(1)$ , as we are using constant time.

**Space Complexity:**  $O(1)$ , as we are using constant space.

### **3. Search in a Row Wise and Column Wise Sorted Matrix**

[<https://coding.ninja/P13>]

**Problem Statement:** You are given an  $N * N$  matrix of integers where each row and column are sorted in increasing order. You are given a target integer  $X$ . Find the position of  $X$  in the matrix. If it exists, then return the pair  $[i, j]$  where  $i$  represents the row, and  $j$  represents the column of the array, otherwise return  $\{-1, -1\}$ .

**For example:** If the given matrix is:

$\begin{bmatrix} 1, 2, 5 \\ 3, 4, 9 \\ 6, 7, 10 \end{bmatrix}$

We have to find the position of 4.

We will return  $[1, 1]$  since  $A[1][1] = 4$ .

#### **Input Format:**

The first line of input contains a single integer  $T$ , representing the number of test cases or queries to be run.

For each test case:

**The first line** contains two space-separated integers,  $N$  and  $X$ , representing the size of the matrix and target element, respectively.

Each of the following  $N$  lines contains  $N$  space-separated integers representing the elements of the matrix.

#### **Output Format:**

For each test case, print the position of  $X$  if it exists. Otherwise, print  $-1 -1$ .

#### **Example:**

```
1
3 4
1 2 5
3 4 9
6 7 10
```

#### **Output:**

```
1 1
```

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

The simple idea is to traverse the array and to search element one by one.

#### **Steps:**

1. Run a nested loop for each row and each column and iterate through each element of the given matrix.
2. Check if the element at the current row and column is the target element.

3. If it is, return the position of the current row and column.
4. If the element is not found in the entire matrix, return  $\{-1, -1\}$

**Time Complexity:**  $O(N^2)$ , we have to traverse the whole matrix that would take  $O(N^2)$  time in the worst case.

**Space Complexity:**  $O(1)$ , as only a constant space is required in the worst case.

### Approach 2: Binary Search Approach

#### Steps:

1. Run a loop for each column—that is, iterate through each column.
2. Using binary search on each row, find if the element exists in the current row. Binary Search is done because each row is sorted.
  - a. If the middle element is equal to the target element, return its position.
  - b. If the middle element is greater than the target element, decrease the right pointer. Otherwise, increase the left pointer.
3. If the element is not found in the entire matrix, return  $\{-1, -1\}$ .

**Time Complexity:**  $O(N * \log N)$ , since we traverse through all the rows and apply binary search on it.

**Space Complexity:**  $O(1)$ , as only a constant space is required in the worst case.

### Approach 3: Optimal Solution

1. Start your search from the top-right corner of the matrix.
2. There can be three cases:
  - a. The current element is equal to target: return this position.
  - b. The current element is greater than the target: This means that any element on the current column is also greater. Thus we move to the previous column.
  - c. The current element is smaller than the target: This means that any element on the current row is also smaller. Thus we move to the next row.

**Time Complexity:**  $O(N)$ , in the worst case, only one traversal from the top-right corner to the bottom left corner is needed—that is,  $i$  from 0 to  $N - 1$  and  $j$  from  $N - 1$  to 0 with at most  $2 * N$  steps.

**Space Complexity:**  $O(1)$ , as only a constant space is required in the worst case.

## 4. Rotate Matrix to the Right [<https://coding.ninja/P14>]

**Problem Statement:** You have been given a matrix **MAT** of size **N \* M** (where N and M denote the number of rows and columns, respectively) and a positive integer **K**. Your task is to rotate the matrix to the right **K** times.

**Note:**

Right rotation on a matrix is shifting each column to the right side (the last column moves to the first column).

**Input Format:**

The first line of input contains an integer **T** representing the number of test cases or queries to be processed.

For each test case:

The **first line** contains three single space-separated integers **N**, **M** and **K**, respectively. **N** and **M** represent the rows and columns of the matrix, and **K** denotes the number of right rotations to be performed.

Then each of the next **N** lines contains **M** space-separated integers representing the elements in a row.

**Output Format:**

For each test case, print the elements of the matrix row-wise after rotation in a single line.

**Sample Input:**

```
1
3 3 2
10 20 30
40 50 60
70 80 90
```

**Sample Output:**

```
20 30 10 50 60 40 80 90 70
```

**Explanation:**

Performing right rotation for the first time ( $K = 1$ ), we get:

```
30 10 20
60 40 50
90 70 80
```

Performing right rotation for the second time ( $K = 2$ ), we get:

```
20 30 10
50 60 40
80 90 70
```

The matrix after rotations will be printed in a single line row-wise. Therefore, the output is:

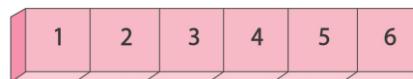
20 30 10 50 60 40 80 90 70

### Approach 1: Row-wise rotation

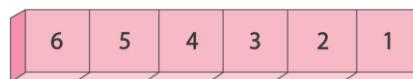
This approach is based on the fact that when we rotate an array to the right by  $K$  times, it transfers  $K$  elements from the end to the beginning of the array, while the remaining elements shift towards the end.

#### Steps:

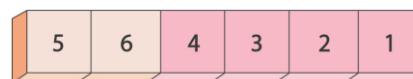
1. The effective rotations in **MAT** can be from 0 to  $M - 1$ , as we get the same matrix **MAT** after every  $M$  rotations. So, we will set  $K$  to  $K \% M$ .
2. Now, we traverse **MAT** row-wise. We will follow the below steps to rotate the  $i^{\text{th}}$  row of **MAT**-
  - a. Taking an example, let the **MAT[i]** be -



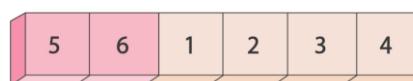
- b. First, we reverse all the elements of the row **MAT[i]**.



- c. We then reverse the first  $K$  elements. (For the sake of example, let's take  $K = 2$ )



- d. Finally, we reverse the next  $N - K$  elements.



3. We will initialise a vector/list **result** and append each row after rotating it.
4. Finally, we return the **result**.

**Time Complexity:**  $O(N \times M)$ , where  $N$  and  $M$  are the number of rows and columns of the given matrix. The time complexity to rotate one row is  $O(M)$ , and there are  $N$  rows. So, the overall time complexity is  $O(N \times M)$

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

## 5. Inplace Rotate Matrix 90 Degree [\[https://coding.ninja/P15\]](https://coding.ninja/P15)

**Problem Statement:** You are given a square matrix of non-negative integers. Your task is to rotate that array by 90 degrees in an anti-clockwise direction without using any extra space.

**For example:** For given 2D array:

```
[[1, 2, 3],  
 [4, 5, 6],
```

[ 7, 8, 9 ]]

After 90 degree rotation in anticlockwise direction, it will become:

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

#### **Input Format:**

The first line contains an integer **T** representing the number of the test case.

For each test case:

The **first line** contains an integer **N** representing the size of the square matrix **MAT**.

Each of the **next N lines** contains **N** space-separated integers representing the elements of the matrix **MAT**.

#### **Output Format:**

For each test case, return the rotated matrix.

#### **Sample Input:**

1 2 3  
4 5 6  
7 8 9

#### **Sample Output:**

3 6 9  
2 5 8  
1 4 7

#### **Explanation:**

The array has been rotated by 90 degrees in an anticlockwise direction as the first row is now the first column inverted and so on for second and third rows.

#### **Approach 1: Cycle Rotation**

Here, we will divide our matrix in cycles. For example, a  $4 \times 4$  cycle will have 2 cycles. The first cycle is formed by its 1<sup>st</sup> row, last column, last row and 1<sup>st</sup> column. The second cycle is formed by 2<sup>nd</sup> row, second-last column, second-last row and 2<sup>nd</sup> column. The idea is for each square cycle, swap the elements involved with the corresponding cell in the matrix in an anti-clockwise direction—that is, from top to left, left to bottom, bottom to right and from right to top one at a time using nothing but a temporary variable to achieve this.

#### **Steps:**

1. There are  **$N/2$**  cycles in a matrix of size **N**.
2. We traverse in the matrix from the outermost cycle—that is, (0,0) to the innermost cycle—that is, **((N / 2) - 1, (N / 2) - 1)**.

3. For each cycle, we'll swap the elements of the matrix in a group of four elements—that is, for each  $i \leq j < N - i - 1$  for each  $0 \leq i \leq (N / 2) - 1$  we swap:
  - a.  $\text{ARR}[i][j]$  with  $\text{ARR}[j, N - 1 - i]$
  - b.  $\text{ARR}[j, N - 1 - i]$  with  $\text{ARR}[N - 1 - i, N - 1 - j]$
  - c.  $\text{ARR}[N - 1 - i, N - 1 - j]$  with  $\text{ARR}[N - 1 - j, i]$
  - d.  $\text{ARR}[N - 1 - j, i]$  with  $\text{ARR}[i][j]$
4. At the end of these loops, we'll get a rotated matrix.
5. Print the matrix.

**Time Complexity:**  $O(N^2)$ , where  $N \times N$  is the dimension of the matrix. Since only a single traversal of the matrix is needed, the time complexity is  $O(N^2)$ .

**Space Complexity:**  $O(1)$  as a constant space is needed because we are doing an in-place rotation.

### Approach 2: Transpose and Reverse

1. To solve the problem, we have to perform two tasks.
  - a. Finding the transpose
  - b. Reversing the columns
2. For finding the transpose of the matrix we swap  $\text{ARR}[i][j]$  with  $\text{ARR}[j][i]$  for each  $0 \leq i < N$  and  $i \leq j < N$ —that is, we swap elements across the principal diagonal of the matrix.
3. Then we reverse the columns by swapping  $\text{ARR}[j][i]$  to  $\text{ARR}[k][i]$  for each  $0 \leq i < N$  representing the columns and  $0 \leq j, k < N$  where,
  - a.  $j$  is incremented from 0 to the point where  $j \geq k$ .
  - b.  $k$  is decremented from  $N - 1$  to the point where  $k \leq j$ .

**Time Complexity:**  $O(N^2)$ , where  $N \times N$  is the dimension of the matrix. Since the matrix is traversed twice, once while taking transpose and then while reversing columns, the time complexity is  $O(N^2)$  is the time complexity.

**Space Complexity:**  $O(1)$  as a constant space is needed because we are doing an in-place rotation.

## 6. Matrix Flip Bit <https://coding.ninja/P16>

**Problem Statement:** Given a binary matrix  $\text{mat}$  of size  $n * n$ . Let  $i, j$  denote the row and column of the matrix, respectively. If  $\text{mat}[i][j]$  is equal to 0, flip every element, which is, a 1 in the  $i$ th row and  $j$ th column—that is, in the same row and column as 0. Return the total number of flips done over all the elements of the matrix.

**For example,** let the matrix be

1	0	1
1	1	0
1	1	1

As shown in the figure, the cells marked in red will be counted as they lie in the same row or column as 0 and will be flipped. Hence, we return 6.

1	0	1
1	1	0
1	1	1

#### Note:

1. Each element in the matrix is either 0 or 1.
2. Only do the flip operation for the elements 0 in the original matrix, not for the elements that were 1 in the original matrix but became 0 due to flip operation.
3. If a cell is already flipped, don't flip it again.
4. Just return the minimum number of flips needed. You don't need to print anything.

#### Input Format :

The first line of input contains **T**, the number of test cases.

For each test case:

The **first line** contains **n**, which represent the dimensions of the  **$n * n$**  matrix.

The following **n** lines contain **n** space-separated integers denoting the numbers in matrix

**mat**.

#### Output Format:

For each test case, return the number of flips required for the given matrix.

#### Sample Input:

```
1
4
1 0 1 1
1 1 0 1
1 1 1 1
1 1 0 1
```

**Sample Output:**

11

**Explanation:**

	0	1	2	3
0	1	0	1	1
1	1	1	0	1
2	1	1	1	1
3	1	1	0	1

We can see that **mat[0][1]** is zero. Hence, we flip all elements which are **1** in the  $i^{\text{th}}$  row—that is, **0th row** and all elements in the  $j^{\text{th}}$  column—that is, **1st column**. We can see that there are three 1s in the 0th row and three 1s in the 1st column. Hence for the 0 at **mat[0][1]** we make count  $3 + 3 = 6$ . We mark the already counted cells red, so we don't count them twice.

	0	1	2	3
0	1	0	1	1
1	1	1	0	1
2	1	1	1	1
3	1	1	0	1

Now we encounter the next zero at **mat[1][2]**. We flip the bits which are unflipped in the second row—that is, **mat[1][0]** and **mat[1][3]**, and in the third column, which are unflipped and are 1—that is, **mat[2][2]**. Finally, the count is  $6 + 3 = 9$ , as shown with the cells marked in red.

	0	1	2	3
0	1	0	1	1
1	1	1	0	1
2	1	1	1	1
3	1	1	0	1

Now the last cell, which is '0'—that is,  $\text{mat}[3][2]$ , we check for all the cells in the fourth row and third column and mark all the '1' which are not marked red and increment the count. Finally, we will have the following matrix.

	0	1	2	3
0	1	0	1	1
1	1	1	0	1
2	1	1	1	1
3	1	1	0	1

Finally, we see that there are 11 cells marked red—that is, we flipped 11 cells. Hence we return 11.

### Approach 1: Brute Force Approach

We need to count the number of 1s which are in the same row or column as any of the 0 in the given matrix. To do that, we iterate through the matrix with the variable  $i$  for the row and  $j$  for the column.

If for any cell  $\text{mat}[i][j] = 0$ , we iterate through the  $i^{\text{th}}$  row and  $j^{\text{th}}$  column and count the number of 1s in them and mark them as -1 to not revisit them. Finally, we return the count.

#### Steps:

1. Take a variable **count** to count the number of bits to be flipped.
2. Iterate through the matrix with the variable  $i$  for row and variable  $j$  for column and check for each element of the matrix if  $\text{mat}[i][j]$  is zero.
3. If  $\text{mat}[i][j]$  is zero we iterate in the  $i^{\text{th}}$  row and  $j^{\text{th}}$  column using a variable **k** and increment the count if  $\text{mat}[i][k]$  or  $\text{mat}[k][j]$  is one.
4. If the above conditions are satisfied, we also mark the cell as -1 to not count it again.

- Finally, we return the **count**.

**Time Complexity:**  $O(n^3)$ , where **n** is the number of rows and columns of the matrix. In the worst case, we can have all 0s, and in that case, each element will have an order of **n** loop. Thus the overall complexity will be the order of  $n^3$ .

**Space Complexity:**  $O(1)$ , as we are using constant space.

### Approach 2: Optimised Approach

If we ensure that a particular row or column is already checked, we need not recheck it. We can do this by maintaining a visited array for rows and columns and checking for every time we encounter a '0' that if its row or column is already visited, we need not recheck it.

#### Steps:

- Make two arrays **visitedRow** and **visitedCol** each of size **n** to store if any particular row and column is already checked or not.
- Traverse the matrix and store the position of the zeroes in an array of pairs.
- Then we traverse the vector, and for each element, if the element is a zero, we add its row and column position as a pair in the array.
- Then we traverse the array with the position of zeroes and for each element of the array, we check if columns and rows are visited or not.
- If not visited, we traverse the particular row or column. We check the number of 1s in the column or row and increment the count. We also replace the 1 with -1 so we do not count it again.
- Finally, we return the count.

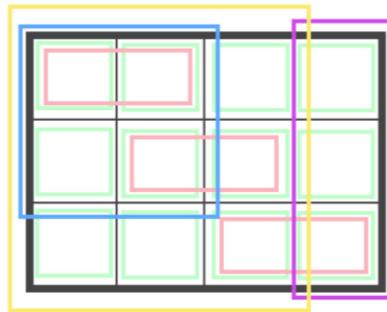
**Time Complexity:**  $O(n^2)$ , where **n** is the number of rows and columns of the matrix. We need to go through all the elements in the matrix, which will take an order of  $n^2$  time.

**Space Complexity:**  $O(n)$ , where **n** is the dimension of the matrix. We need two visited arrays to store whether any particular row or column is visited or not.

## 7. Number of Rectangles in M x N Grid [\[https://coding.ninja/P17\]](https://coding.ninja/P17)

**Problem Statement:** You are given an arbitrary grid with **M** rows and **N** columns. You have to print the total number of rectangles that can be formed using the rows and columns of this grid. In simple words, print the number of unique rectangles in the grid.

**For example**, consider the grid shown below. The dark black boundary encloses a grid of dimension 3 x 4.



The **green** colour represents rectangles of dimension  $1 \times 1$ . The **pink** colour represents the rectangles of dimension  $1 \times 2$ . The **blue** colour represents the rectangles of dimension  $2 \times 2$ . The **yellow** colour represents the rectangles of dimension  $3 \times 3$ . The **magenta** colour represents the rectangles of dimension  $3 \times 1$ . There can be many different other possibilities as well. You need to print the total number of all such rectangles.

#### Note:

Two rectangles are said to be unique if at least one of their 4 sides is non-overlapping.

#### Follow up:

Try to solve this problem in **O(1)** time.

#### Input Format :

The first line contains a single integer **T**, representing the number of test cases.

For each test case:

There are two space-separated positive integers representing **M** (number of rows) and **N** (number of columns), respectively.

#### Output Format:

For each test case, print the number of rectangles that can be formed using the rows and columns of the given grid.

#### Example:

1  
3 4

#### Output:

60

#### Explanation:

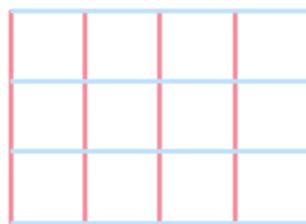
In a grid of  $3 \times 4$  dimensions, there are 30 rectangles of  $1 \times N$  where  $N = 1, 2, 3, 4$ . There are 20 rectangles of  $2 \times N$  where  $N = 1, 2, 3, 4$ . There are 10 rectangles of  $3 \times N$  where  $N = 1, 2, 3, 4$ . Adding all these values, we get  $30 + 20 + 10 = 60$ . Hence, there are 60 rectangles in a grid of dimension  $3 \times 4$ .

## Approach 1: Constant Time Solution

If we look carefully at the given problem, we observe that to create a rectangle in a grid, we just need to select two horizontal and two vertical lines among all the lines present in the grid.

In a grid with  $N$  columns, we know that there are  $(N + 1)$  vertical lines. Also, in a grid with  $M$  rows, there are  $(M + 1)$  horizontal lines (including the outer boundaries).

To better understand this approach, consider the grid shown below:



This grid had 3 rows and 4 columns. Hence,  $M = 3$  and  $N = 4$ . Vertical lines are coloured **pink** and horizontal lines are coloured **blue**.

As is clearly visible, there are  $(M + 1) = 4$  horizontal lines and there are  $(N + 1) = 5$  vertical lines. Now, to select two horizontal lines out of 4, there are  ${}^4C_2 = 6$  ways. Similarly, to select two vertical lines out of 5, there are  ${}^5C_2 = 10$  ways.

The total number of ways to select two vertical and two horizontal lines out of five vertical and four horizontal lines is  $= (6 * 10) = 60$ . Hence, there are a total of 60 rectangles in the given grid.

Our formula becomes  $\rightarrow {}^{(M+1)}C_2 * {}^{(N+1)}C_2$ , where  ${}^nC_r$  is defined as the total number of unique ways to choose  $r$  objects from a set containing  $n$  different objects.

On simplification, this formula evaluates to  $\rightarrow (M * (M + 1) * N * (N + 1)) / 4$ . We can directly use this formula to get the total number of rectangles in a given grid with  $M$  rows and  $N$  columns.

**Time Complexity:**  $O(1)$ , as all operations will be executed in constant time.

**Space Complexity:**  $O(1)$ , as we are using constant extra memory.

## 8. Matrix Median [\[https://coding.ninja/P18\]](https://coding.ninja/P18)

**Problem Statement:** You have been given a matrix of  $N$  rows and  $M$  columns filled with integers where every row is sorted in non-decreasing order. Your task is to find the overall median of the

matrix—that is, if all elements of the matrix are written in a single line, then you need to return the median of that linear array.

The median of a finite list of numbers is the “middle” number when those numbers are listed from smallest to greatest. If there is an odd number of observations, the middle one is picked.

**For example**, consider the list of numbers [1, 3, 3, 6, 7, 8, 9]. This list contains seven numbers. The median is the fourth of them, which is 6.

**Note:**

$N \times M$  is always an odd number.

**Input Format:**

The first line contains a single integer **T** representing the number of test cases.

For each test case:

The **first line** contains two integers **N** and **M**, denoting the number of rows and columns.

Next **N lines** contain **M** space-separated integers, each denoting the elements in the matrix.

**Output Format:**

For each test case, print an integer which is the overall median of the given matrix.

Output for every test case will be printed in a separate line.

**Sample Input:**

```
1
1 3
1 2 3
```

**Sample Output:**

```
2
```

**Explanation:**

The overall median of the matrix is 2.

**Approach 1: Naive Approach**

Since the median is the middle number in a sorted—ascending or descending—list of numbers, our basic approach is to generate the list of integers from the given matrix in a sorted manner.

**Steps:**

1. Create an auxiliary array/list of  **$N \times M$**  length.
2. Traverse the matrix and insert all the elements in that array/list.
3. Sort that list/array in non-decreasing order.
4. The element on the  $((N \times M)/2)^{th}$  index will be the overall median of the matrix.

**Time Complexity:**  $O((N*M) * \log(N*M))$ , where **N** is the number of rows and **M** is the number of columns in the given matrix. Since we are sorting a list/array of integers whose length is **N \* M**, its time complexity will be  **$O(N*M)$** . The overall complexity will be  **$O((N*M) * \log(N*M))$** .

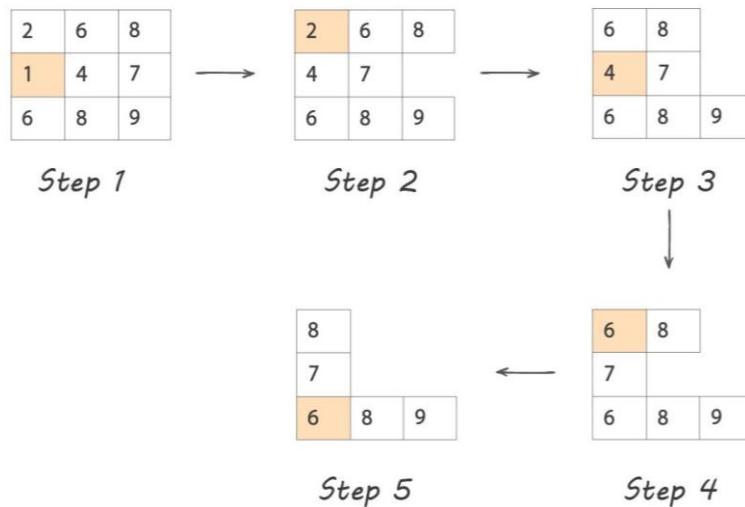
**Space Complexity:**  $O(N*M)$ , where **N** is the number of rows, and **M** is the number of columns in the given matrix. The only extra space we are using is to store the elements of the matrix, which are **N \* M** in total. Therefore, the overall space complexity will be  **$O(N*M)$** .

### Approach 2: Optimised Sorting

The basic idea of this approach is to utilise the fact that elements in the row are already stored in a sorted manner.

Suppose we want to create a sorted (in non-decreasing order) list of integers from the given matrix. Now let's try to append the integers in the list one by one. Consider the first element of each row. The minimum of those elements will be the minimum element of the matrix itself. Why? Because the first element of each row is the minimum element of that row.

To find the second smallest element, we will remove the smallest element, and we will consider the next element ( $2^{\text{nd}}$  element) of the row where the minimum element was found. Following the steps mentioned above, we can easily find the sorted list of integers.



The above image explains the first five steps of the algorithm. The highlighted element is the minimum element of the matrix in each step. We can observe that we just need to consider elements of the first column to find the minimum. Therefore, we can use a min-heap data structure to operate efficiently.

Since the median is always the middle element in the sorted list, we simply do the above operation for  $(N*M)/2$  times. And then, the next smallest element will be the median.

### Steps:

1. Create a min-heap that stores a **pair**  $\langle x, \langle \text{row}, \text{col} \rangle \rangle$ , where **x** is the element of the given matrix and  $\langle \text{row}, \text{col} \rangle$  is the index of that element in the matrix. The reason behind storing the index of that element is that after removing that element, we will have to consider the next element from the same row for finding the next smallest element.
2. Initialise a **count** variable to zero which stores the number of elements that were removed from the min-heap.
3. Extract the minimum element from the min-heap and increment the **count**.
4. If **count** is  $(N*M)/2 + 1$ , the current minimum element is the overall median of the matrix.
5. Using the index of the minimum element, find the next element in that row and insert the pair of the next element and its index in the min-heap.
6. If that row doesn't have any more elements, repeat the same process (from step 3).

**Time Complexity:**  $O(N*M*\log(N))$ , where **N** is the number of rows, and **M** is the number of columns in the given matrix. We are using a min-heap to extract the minimum element. The time complexity of removing a minimum element from a min-heap is  $\log(N)$ . Since we will be doing the above operation a maximum of **N \* M** times and in the worst case, there would be a maximum of **N** elements in the min-heap, the overall time complexity will be  $O(N*M*\log(N))$ .

**Space Complexity:**  $O(N)$ , where **N** is the number of rows, and **M** is the number of columns in the given matrix. We are using the min-heap, which will store a maximum of **N** elements. The overall space complexity will be  $O(N)$ .

### Approach 3: Space + Time Optimised Approach

The core idea of this approach is to utilise the property of the median and the sorted rows. Note that for a given list of integers whose length is **N \* M**, the median is the smallest number greater than or equal to at least  $(N * M)/2$  elements in the list.

### Steps:

Let **x** be the potential median of the matrix. Now, **x** can be any number from the matrix. We can do a binary search to find the **x**.

1. Let **MIN** and **MAX** be the minimum and maximum element of the matrix, respectively. **x** can be any number in the range of **[MIN, MAX]**.
2. Initialise **low** and **high** for the binary search algorithm to **MIN** and **MAX**, respectively. In other words, initialise our search space to **[MIN, MAX]**.
3. We will iterate until **low** is not equal to **high**
  - a. Initialise **mid** as  $(\text{low} + \text{high})/2$ .
  - b. Now, our task is to find the number of elements in the matrix that are less than or equal to the **mid** (say it is **count**).
    - i. Iterate through each row one by one.

- ii. Since each row is sorted in non-decreasing order, we can easily find the total number of elements in that row that are less than or equal to **mid** using the **upper\_bound()**.
  - iii. Take the sum of such elements over each row.
  - c. Now, if **count** is less than or equal to **(N\*M)/2** then, the median will be undoubtedly greater than mid. Therefore, change our search space to **[mid + 1, high]**—that is, assign **mid + 1** to **low**.
  - d. Else, the median will indeed lie in the range of **[low, mid]**—that is, assign **mid** to **high**.
  - e. Repeat this process while our search space doesn't converge to a single element.
4. Since the steps mentioned above will terminate when **low** is equal to **high**, we'll get the overall median of the matrix which is **low**.

**Time Complexity:**  $O(N * \log(M) * \log(\text{MAX} - \text{MIN}))$ , where **N** is the number of rows and **M** is the number of columns in the given matrix. And **MAX** and **MIN** are the minimum and maximum element of the matrix, respectively. At each iteration of the outer loop (which runs until **low** is equal to **high**), we are iterating through every row and doing a binary search on it. Since there are **M** elements in each row, the **upper\_bound()** function will take  $O(\log(M))$  time. And we'll be doing **N** number of such operations in each iteration. Therefore, it will take  $O(N * \log(M))$  time for each iteration.

And the outer loop which is again a binary search, will run for  $O(\log(\text{MAX} - \text{MIN}))$  time. Therefore, the overall time complexity will be  $O(N * \log(M) * \log(\text{MAX} - \text{MIN}))$ .

**Space Complexity:**  $O(1)$ . Since we are not using any extra space, we are only doing the binary search. So, the overall space complexity will be  $O(1)$ .

## 9. Empty Cells in a Matrix [\[https://coding.ninja/P19\]](https://coding.ninja/P19)

**Problem Statement:** You are given an integer **N** denoting the size of a **N \* N** matrix. Initially, each cell of the matrix is empty. You are also given an integer **K** denoting the number of tasks. In each task, you are given two integers  $(i, j)$  where  $i$  represents the  $i^{th}$  row, and  $j$  represents the  $j^{th}$  column of the matrix.

You have to perform each task in the given order. For each task, you have to place 0 in each cell of  $i^{th}$  row and  $j^{th}$  column. After completing all the tasks, you have to return a matrix of size **K** where the **n<sup>th</sup>** element of the matrix is the number of empty cells in the matrix after the **n<sup>th</sup>** task.

### Note:

We call a cell empty only if it does not contain any value. → Indexing is 0-based.

**For example**, consider an empty matrix of size **N \* N** where **N = 3**.

```
[[NULL, NULL, NULL]
 [NULL, NULL, NULL]
 [NULL, NULL, NULL]]
```

Suppose the value of **K** is 2, which means we have to perform two tasks.

**Task 1:** (0, 0)

Matrix after placing 0 in each cell of 0<sup>th</sup> row and 0<sup>th</sup> column:

```
[ [0,    0,    0]  
[ 0, NULL, NULL]  
[ 0, NULL, NULL] ]
```

The number of empty cells now: 4

**Input Format :**

The **first line** of input contains two space-separated integers, **N** and **K**.

The **next N** rows contain **N** space separated integers representing the elements of the **N \* N** matrix.

**Output Format:**

For each test case, print the matrix of size **K** where the **n<sup>th</sup>** element of the matrix is the number of empty cells in the matrix after the **n<sup>th</sup>** task.

**Sample Input:**

```
2 2  
0 0  
0 1
```

**Sample Output:**

```
1 0
```

**Explanation:**

Initial Matrix:

```
[ [NULL, NULL]  
[NULL, NULL] ]
```

Matrix after task (0,0)

```
[ [0,    0 ]  
[ 0, NULL] ]
```

As we can see, there is only one empty cell. Hence, we will print 1.

Matrix after task (0,1)

```
[ [0,0]  
[0,0] ]
```

Now there is no empty cell in the matrix, hence print 0.

**Approach 1: Brute Force**

We will create a matrix of size **N \* N** and initialise all of its elements with a number say -1. For every task  $(i,j)$ , we will replace -1 with 0 for every cell of the  $i$ th row and  $j$ th column.

Now we will simply count the number of -1 in the matrix using two loops and print it.

#### Steps:

1. Create an integer matrix of size **N \* N** and initialise every cell with -1.
2. Create an integer array of size **K**, namely **outputArray**.
3. For every task  $(i,j)$ ,
  - a. Run a loop over the  $i^{th}$  row and replace every cell of this row with 0.
  - b. Run a loop over the  $j^{th}$  column and replace every cell of this column with 0.
  - c. Count the number of -1 in the matrix and insert it to the **outputArray**.
4. Return the **outputArray**.

**Time Complexity:**  $O(K*N*N)$ , where **N** is the number of rows and columns of the matrix, and **K** is the number of tasks.

**Space Complexity:**  $O(N*N)$ , where **N** is the number of rows and columns of the matrix.

#### Approach 2: HashMap Approach

Suppose we have a task  $(i,j)$ . We will update  $i^{th}$  row and  $j^{th}$  column of the matrix by 0. What can we conclude from this? We can say that the  $i^{th}$  row and  $j^{th}$  column will not contribute any empty cells. So we can just consider a matrix not having  $i^{th}$  row and  $j^{th}$  column. This means our number of rows has been reduced by one and the number of columns has also been reduced by one.

#### Steps:

1. Initialise two variables, rows by **N** and columns by **N**.
2. Create an integer array of size **K** namely **outputArray**.
3. Take two sets, **rowSet** and **colSet**, to keep track of the rows and columns we are eliminating.
4. For every task  $(i,j)$ , repeat the steps from 4 to 6.
5. Check for  $i$ 
  - a. If  $i$  is not present in **rowSet**, add it to the **rowSet** and reduce the number of rows by one.
  - b. If  $i$  is already present in the **rowSet**, we don't have to do anything because this row has already been eliminated.
6. Check for  $j$ 
  - a. If  $j$  is not present in **colSet**, add it to the **colSet** and reduce the number of columns by one.
  - b. If  $j$  is already present in the **colSet**, we don't have to do anything because this column has already been eliminated.
7. Now the number of empty cells would be the product of the remaining rows and columns, insert it to the **outputArray**.
8. Return the **outputArray**.

**Time Complexity:**  $O(K)$ , where  $K$  is the number of tasks. For each task, we are just checking and updating the set. It takes constant time. Thus, the time needed for completing a task is  $O(1)$ . As we have  $K$  tasks, the total time complexity would be  **$O(K)$  in the worst case**.

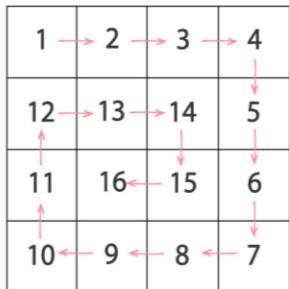
**Space Complexity:**  $O(K)$ , where  $K$  is the number of tasks. We are not creating any matrix. Instead, we are using two maps only. In the worst case, the size of each map can go to  $K$ .

## 10. Find nth Elements of Spiral Matrix [\[https://coding.ninja/P20\]](https://coding.ninja/P20)

**Problem Statement:** Given a matrix with  $n$  rows and  $m$  columns and an integer  $k$ , your task is to find the  $k^{\text{th}}$  element, which is obtained while traversing the matrix in spiral form.

### For Example:

The below picture shows how to traverse a matrix in spiral form.



Spiral traversing in the matrix

### Input Format:

The **first line** contains three space-separated integers,  $n$ ,  $m$ , and  $k$ , respectively. Each of the following  $n$  lines contains  $m$  space-separated integers representing the elements in a row.

### Output Format:

For each test case, print the  $k^{\text{th}}$  element which is obtained while traversing the matrix in spiral form.

### Sample Input:

```
3 4 8
1 2 3 4
5 6 7 8
7 9 2 1
```

### Sample Output:

```
9
```

### Explanation:

The given matrix can be represented as follows:

1	2	3	4
5	6	7	8
7	9	2	1

Spiral form traversal of the given matrix is  $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 8 \rightarrow 1 \rightarrow 2 \rightarrow 9 \rightarrow 7 \rightarrow 5 \rightarrow 6 \rightarrow 7$ . In a spiral form traversal of the given matrix, the integer **9** is at the **eighth** position of the matrix. Hence, we would return the integer '**9**'.

### Approach 1: Spiral Order Traversal

The basic idea is to start traversing the matrix in spiral form and find the **k<sup>th</sup>** element in the traversal.

#### Steps:

1. We traverse the matrix in spiral form. (refer to the image in the problem statement)
2. Keep two variables **startRow = 0**, **startCol = 0**, **n** and **m** are already ending indices of row and column.
3. Keep a variable **count = 0**, if **k == count**, we return.
4. Start iterating **i** from **startCol** to **m - 1** for the first row and keep incrementing count for every element visited.  
For any **i** if **count == k**, return '**matrix[startRow][i]**'.
5. Update **startRow = startRow + 1**.
6. Then iterate **i** from **startRow** to **n - 1** for the last column and keep incrementing count for every element visited.  
For any **i** if **count == k**, return **matrix[i][m - 1]**.
7. Update **m = m - 1**.
8. Then iterate **i** from **m - 1** to **startCol** with **-1** jumps and keep incrementing count for every element visited. We are iterating the last row from right to left.  
For any **i** if **count == k**, return **matrix[n - 1][i]**.
9. Update **n = n - 1**.
10. Then iterate **i** from **n - 1** to '**startRow**' with **-1** jumps and keep incrementing count for every element visited. We are iterating the first column from bottom to top.  
For any **i** if **count == k**, return **matrix[i][startCol]**.
11. Update **startCol = startCol + 1**.
12. Repeat the same step until **count** is not equal to the **k**.

In other words, keep iterating the matrix from left to right for the row marked by **startRow**, followed by top to bottom for the column marked by **m**, followed by right to left for the row marked by **n**, followed by bottom to top for the column marked by **startCol**, and keep incrementing the value of **count** by 1 until we get **count** to be equal to **k**

**Time Complexity:**  $O(N*M)$ , where **N** is the number of rows and **M** is the number of columns. We are iterating every element of the matrix.

**Space Complexity:**  $O(1)$ , as we are using constant space.

### Approach 2: Level Wise Traversal

While traversing the array in spiral order, a loop is used to traverse the sides. Hence, if it can be found out that the  $k^{th}$  element is present in the given side, then the  $k^{th}$  element can be found out in constant time. We can approach this recursively as well as iteratively.

#### Steps:

1. Use two variables **startRow = 1** and **startCol = 1**.
2. Iterate a loop level from 0 to  $\max(n, m)$ . There can be four cases.
  - a. If **startRow <= k <= m**
    - i. Means  $k^{th}$  element is present in **level<sup>th</sup>** row.
    - ii. Then we can find the  $k^{th}$  element in constant time in this row.
  - b. If **m - level <= k <= n**
    - i. Means  $k^{th}$  element is present in **(m - level)<sup>th</sup>** column.
    - ii. Then we can find the ' $k^{th}$ ' element in constant time in this column.
  - c. If **(n - 1) \* m + 1 <= k <= n \* m**
    - i. Means  $k^{th}$  element is present in **n** row.
    - ii. Then we can find the  $k^{th}$  element in constant time in this row.
  - d. If **startCol - level <= k <= n - 1**
    - i. Means  $k^{th}$  element is present in **startCol - level** column.
    - ii. Then we can find the  $k^{th}$  element in constant time in this column.
3. If any of the above conditions is true, then find the respective  $k^{th}$  element and return else, update **startCol = startCol + 1**, **startRow = startRow + 1**, **n = n - 1** and **m = m - 1** and iterate for the next level.

**Time Complexity:**  $O(\max(N, M))$ , where **N** is the number of rows and **M** is the number of columns. We are iterating a loop **level max(N, M)** times.

**Space Complexity:**  $O(1)$ , as we are using constant space.

# 3. Strings

---

## Introduction to Strings

A string is a data type in programming that is defined as a sequence of **characters**. It is implemented as an array of bytes (or words) that stores a sequence of elements, typically characters using some character encoding.

Depending on the programming language, strings can be of two types:

- **Mutable Strings:** Mutable strings are strings that can be modified.
- **Immutable Strings:** Immutable strings are strings that cannot be modified. Making any changes to the string involves creating a copy of the original string and deallocating it.

## Operations on Strings

Common string operations include finding the **length**, **copying**, **concatenating**, **replacing**, and **counting** the occurrences of characters in a string. Such operations on strings can be performed easily with built-in functions provided by any programming language.

Some of the standard operations involving strings are:

- **Access characters:** This operation involves retrieving characters of a string. Similar to arrays, strings follow 0-based indexing.  
For example, **S = “apple”**, ‘a’ is the character at 0th index (**S[0]**) and ‘e’ is the character at 4th index of the string S (**S[4]**).
- **Concatenation:** Joining the characters of one string to another is a concatenation operation. The result of this operation is a joined string.  
For example:  
**S1 = “Hello ”, S2 = “World.”**, the concatenation of strings S1 and S2 represented by S3 is **S3 = S1 + S2 = “Hello World.”**, while **S3 = S2 + S1 = “World.Hello”**.
- **Substring:** A contiguous sequence of characters in a string. For example, substrings of the string “boy” are: “”, “b”, “o”, “y”, “bo”, “oy”, “boy” (an empty string is also a substring ).
  - **Prefix:** A prefix is any leading contiguous part of the string. For example, “”, “g”, “ga”, “gar”, “gard”, “garde”, “garden” are all prefixes of the string **S = “garden”**.
  - **Suffix:** A suffix is any trailing contiguous part of the string. For example, “”, “n”, “en”, “den”, “rden”, “arden”, “garden” are all suffixes of the same string S.

**Note:** A string of length **N** has  $(N * (N + 1)) / 2$  substrings.

## Applications of Strings

- String matching algorithms, which involve searching for a pattern in a given text have various applications in the real world. These algorithms contribute to efficiently implementing spell checkers, spam filters, intrusion detection systems, plagiarism detection, bioinformatics, and digital forensics.
- 

## Practice Problems

### 1. Uncommon Characters [\[https://coding.ninja/P21\]](https://coding.ninja/P21)

**Problem Statement:** Given two strings **S1** and **S2** of lowercase alphabets, find the list of uncommon characters for the two strings.

#### Note:

- A character is uncommon if it is present only in one of the strings—that is, it is either present in **S1** or **S2**, but not in both.
- Both the strings contain only lowercase alphabets and can contain duplicates. Return the uncommon characters in lexicographically sorted order.

#### Input Format:

The **first line** contains an integer **T** representing the number of test cases.

For each test case:

There are two strings, **S1** and **S2**, separated by a single space.

#### Output Format:

For each test case, the uncommon characters are printed in sorted order.

#### Sample Input:

```
1
coding ninjas
```

#### Sample Output:

```
acdgjos
```

#### Explanation:

For the given two strings, **{c, o, d, g}** are the characters that are present only in the first string and **{j, a, s}** are characters that are present only in the second string. Thus, **{c, o, d, g, j, a, s}** are the uncommon characters for the given two strings and are printed in the sorted order.

#### Approach 1: Naive Approach

The basic solution for this problem involves using loops. We use a set, **uncommonChars**, to store the uncommon characters of the strings. This would reduce our effort to handle duplicates of a single uncommon alphabet (because a set does not contain duplicates). We solve the problem in two parts:

1. Finding the uncommon characters that are present in string **S1**, but not in string **S2**:  
For every character in string **S1**, we check if that character is present in string **S2** or not. This can be easily done using two loops. If the character is not present in **S2**, we add it to the set **uncommonChars**.
2. Finding the uncommon characters that are present in string **S2**, but not in string **S1**:  
Similarly, for every character in string **S2**, we check if that character is present in string **S1** or not. If the character is not present in **S1**, we add it to the set **uncommonChars**.

After the above two steps, the set **uncommonChars** will contain all the uncommon characters for both the strings.

After insertion inside a set, the reordering of elements takes place and the set is sorted. Hence, the characters will be in lexicographically sorted order.

**Time Complexity:**  $O(N * M)$ , where **N** and **M** are lengths of the strings **S1** and **S2**.

The time complexity of finding the characters present in one string and not in another using two loops is  $O(N * M)$ . Also, inserting an element in a set takes  $O(\log(K))$  time, where **K** is the size of the set. Here, the set can have a maximum size of 26. Thus the worst-case complexity of inserting an element in the set will be  $O(\log(26))$ . Now, since we are using the strategy of loops two times, the final complexity is  $O(2 * N * M * \log(26)) = O(N * M)$ .

**Space Complexity:**  $O(1)$ , since the set **uncommonChars**, can contain at most 26 elements (since there are 26 lowercase alphabets), constant space is required.

## Approach 2: Optimized Approach using Hashing

The main idea behind this approach is to use **hashing**. We create a hash table of size 26 for all the lowercase alphabets and initialise all the elements as 0. Here, 0 denotes that the character is not present in either of the strings.

### Steps:

We start traversing the first string. For every character, we update its value in the hash table as 1. Here, 1 indicates that the character is present in the first string. If the character is found again, we keep the value as 1 only, signifying that it has already been found before.

After the above step, we start traversing the second string. For each character in the second string, we do the following:

1. If its current value in the hash table is 1, then this means that the character is present in both the strings. In this case, we set the value in the hash table as -1, where -1 denotes that the character is present in both the strings.
2. If its current value is 0, it means that the character is present only in the second string. In this case, we set its value in the hash table as 2, indicating that the character is present only in string 2.
3. If its current value is 2 or -1, it means that the character has already been encountered in the second string. In this case, nothing else needs to be done.

Now, the characters with a value of 1 or 2 in the hash table are the uncommon characters for the given two strings.

Since a hash table is maintained in a sorted manner, the characters will, by default be in lexicographically sorted order.

**Time Complexity:**  $O(N + M)$ , where  $N$  and  $M$  are lengths of the strings  $S1$  and  $S2$ . Traversing a string takes linear time. Thus, traversal of string  $S1$  takes  $O(N)$  time, and likewise traversing  $S2$  takes  $O(M)$  time. Therefore, the final complexity is  $O(N + M)$ .

**Space Complexity:**  $O(1)$ , constant extra space is required to create the hash table and the final set of uncommon characters. Thus the space complexity is  $O(1)$ .

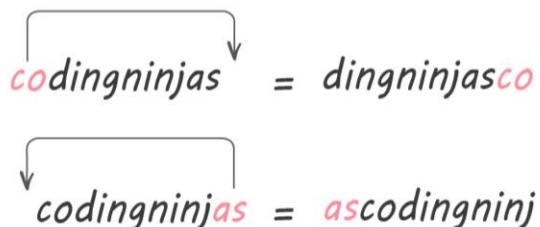
## 2. Left And Right Rotation Of A String [<https://coding.ninja/P22>]

**Problem Statement:** You are given string  $str$  and an integer  $D$ . The task is to rotate the same string once to the left and once to the right by  $D$  units from the starting index. You are required to return the rotated string in each case.

**Example:** Suppose the input string  $str = \text{"codingninja"}$ , and  $d = 2$

Hence, left rotation by 2 units will result in the string — “dingninja”.

And right rotation by 2 units will result in the string — “ascodingninja”.



### Input Format:

The **first line** contains an integer  $T$  which denotes the number of test cases or queries.

For each test case:

The **first line** contains the string  $str$ .

The **second line** contains an integer  $D$ , representing the number of units by which the string is to be rotated left and right.

### **Output Format:**

For each test case, print the left and right rotations of the string separated by single space respectively.

### **Sample Input:**

```
1  
codingninjas  
3
```

### **Sample Output :**

```
ingnijascod jascodingnin
```

### **Explanation:**

In string **codingninjas**, the substring of length **D = 3**, starting from the beginning is 'cod'. This substring is removed from the beginning and attached to the end of the string in the left rotation. Similarly, in the right rotation, the substring of length **D = 3** from the end is 'jas'. This substring is removed from the end and attached to the beginning of the string.

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

The idea is to use an additional string to store the copies of required substrings. Hence, we will initialise an empty string named **ans**.

#### **Steps:**

1. For the left rotation of the given string, append the last **N - D** characters to **ans**, then append the remaining first **D** characters to the **ans**. The string **ans** after the left rotation is the required string. Print the current state of **ans**.
2. For the right rotation, again initialise **ans** with an empty string (i.e. **ans = ""**) and append the last **D** characters of the given string to the **ans**, then append the remaining **N - D** characters of the given string from beginning to the **ans**. The string **ans** after the right rotation is the required string. Print the current state of **ans**.

#### **Explanation with an example:**

Let the given string be **abcdef** and **D = 2**.

Initialize **ans = ""** for left rotation:

1. **ans = "" + "cdef" = "cdef"**
2. **ans = "cdef" + "ab" = "cdefab"**

For right rotation again initialize **ans** to empty:

1. **ans = "" + "ef" = "ef"**
2. **ans = "ef" + "abcd" = "efabcd"**

**Time Complexity:**  $O(N)$ , where  $N$  is the length of the string. In the worst case, we have to traverse the whole string in each rotation.

**Space Complexity:**  $O(N)$ , where  $N$  is the length of the string. In the worst case, the temporary string variable used to store the string after rotation would be equal to the length of the string.

### Approach 2: In-Place Rotation

For this method, we keep in mind the following.

- In the case of left rotation of the string, reverse the substring of length  $D$  starting from the beginning. Then reverse the substring of length  $N - D$ , starting from the  $(D - 1)^{th}$  index. Now finally, rotate the whole string. The result we get is the required string after left rotation.
- In the case of right rotation, it's the same as the left rotation. The difference is that we reverse the substring of length  $N - D$  from the beginning. Hence, rotating a string by  $2$  units in the left direction is the same as rotating the string by  $N - 2$  units in the right direction.

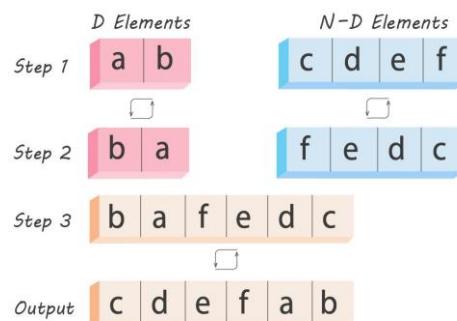
### Explanation with an example:

Let the given string is “**abcdef**” and  $D = 2$ .

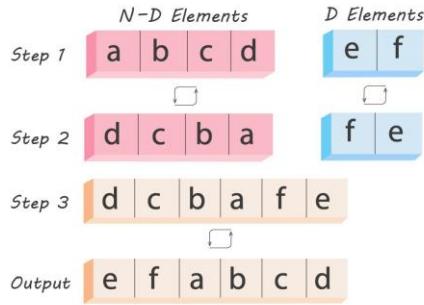
- In Left Rotation: “**bacdef**”  $\rightarrow$  “**bafedc**”  $\rightarrow$  “**cdefab**”.
- In Right Rotation : **LeftRotate(string,4)**  $\rightarrow$  “**dcbaef**” $\rightarrow$  “**dcbafe**”  $\rightarrow$  “**efabcd**”

From the above example, it is clear that rotating the string in the left direction by 2 units is the same as rotating the string by  $N - 2$  units in the right direction, where  $N - 2 = 6 - 2 = 4$ .

### Visualisation for Left Rotation:



### Visualisation for Right Rotation:



**Time Complexity:**  $O(N)$ , where  $N$  is the length of the string. In the worst case, to reverse the string, you need to traverse the whole string.

**Space Complexity:**  $O(1)$ , as only constant extra space is required when we reverse the string in-place in the worst case.

### 3. Anagram Difference [\[https://coding.ninja/P23\]](https://coding.ninja/P23)

**Problem Statement:** You have been given two strings, let's say **str1** and **str2** of equal lengths. You are supposed to return the minimum number of manipulations required to make the two strings anagrams.

**Note:** An **anagram** is a word or phrase formed by rearranging the letters of a different word or phrase.

**For example:**

String "eat" and "ate" are anagram to each other, but strings "buy" and "bye" are not.

**Input Format:**

The **first line** contains an integer **T**, which denotes the number of test cases or queries to be run. For each test case:

The **first line and second line** contain the string **str1** and **str2**, respectively.

**Output Format:**

For each test case/query, print the minimum number of manipulations required to make **str1** and **str2** anagram.

**Sample Input:**

```
1
except
accept
```

**Sample Output :**

```
2
```

**Explanation:**

In the example, we can either change two character of **str1**—that is, **{'e','x'}** to **{'a','c'}** or we can change two character of **str2**—that is, **{'a','c'}** to **{'e','x'}**, to make strings anagram. Hence, the minimum number of manipulations to make **str1** and **str2** anagrams will be 2.

### Approach 1: Brute Force

The straightforward intuition is to loop over all characters in any one given string and search for each character in the other string. If the character of the first string exists in the second string, then change it to '#' so that we do not include this character again. If the character does not exist, then increment the count.

#### Steps:

1. Iterate over all characters of **str1** and search it into **str2**.
2. For any  $i^{th}$  character, if **str1[i]** exists in **str2**, then make **str2[j]** to '#', where  $j$  is the index of **str2** at which **str1[i]** is found. Remember that we need to change only the first occurrence of the character to '#', not all the occurrences.
3. Otherwise, increment the count of *anagram difference*. Let's name it as **minAnagramDiff**. This represents the number of changes that need to be made in order to make **str1** and **str2** anagrams of each other. Maintaining the difference would do the trick, as we can change either of the strings to achieve our goal.
4. Return the **minAnagramDiff**.

**Time Complexity:**  $O(N^2)$ , where **N** is the length of the given string. Since we will check every character of **str1** and check whether the current character exists into **str2** or not, the worst case happens when none of the characters of string **str1** exist into **str2**. In this case, if there are **N** characters of **str1**, we will be spending **N** time to search into **str2**. Therefore, the overall time complexity will be  $O(N^2)$ .

**Space Complexity:**  $O(1)$ , we are not using any extra space. Therefore, space complexity will be constant.

### Approach 2: Using Frequency

The intuition behind this approach is that we will first store the frequency of any one string, and then we go for each character in the other string, and we decrease the frequency by one.

Let's say we have stored the frequencies into **freq**, where **freq** is the array of length 26, and initially **freq** contains zero for all characters. We will iterate over all characters of the other string and decrease the frequency by one corresponding to stored frequencies of that character.

In the end, we have the difference in frequencies between the first string and the second string into the **freq** array.

Then, we will store the sum of *absolute frequencies difference* from **freq**. Now, we may change half of the mismatched characters of the first string to the required characters for the second string and half of the mismatched characters of the second string to the required characters for the first string. Doing so, they become anagrams.

### Steps:

1. Take an array of size 26 to store the frequencies of characters, initially let it contain 0 for every element. Let's call it **freq**.
2. Iterate over all characters of the first string—that is, **str1** and store the frequencies of characters.
3. Iterate over all characters of the second string and decrease the frequency of characters from **freq**.
4. Now calculate the sum of absolute frequency of all 26 characters. Let's call **freqDiff**.
5. The minimum number of manipulations needed to make the strings anagram will be  $(\text{freqDiff} / 2)$  because we can change half of the mismatched character from **str1** to the required characters for **str2**, or we can change half of the mismatched characters from **str2** to the required characters for **str1**.

To understand this better, let us take the following example:

**str1 = except**  
**str2 = accept**

Let us calculate the **freqDiff** for these two strings. As we can see, there are four characters different in these two strings (**{e, x}** in **str1** and **{a, c}** in **str2**). Hence, the **freqDiff** is equal to 4. Now, we can either change e, x to a, c in str1 or a, c to e, x in str2. Therefore, our answer is **freqDiff/2**, which is **4/2 = 2**.

**Time Complexity:** **O(N)**, where **N** is the length of the string **str1**.

We are storing the frequencies of **N** elements into an array and also decreasing the frequencies by the character of the second string, which takes constant time for the array. Therefore, the overall time complexity will be **O(N)**.

**Space Complexity:** **O(N)**, where **N** is the length of the given string.

We are using HashMap to store the frequencies of the character of the given string. Hence, in the worst case, when all characters of the string will be distinct, the HashMap will take **O(N)** space. Therefore, overall space complexity will be **O(N)**.

## **4. Find Kth Character of Decrypted String** [<https://coding.ninja/P24>]

**Problem Statement:** You have been given an encrypted string where repetitions of substrings are represented as substring followed by the count of substrings. For example, the string "aabbbcdcdcd" will be encrypted as "a2b3cd3".

You need to find the **K<sup>th</sup>** character of the decrypted string. The decrypted string would have **1-based indexing**.

**Input Format:**

The **first line** contains an encrypted string **S**.

The **second line** contains the integer value **K**.

#### Output Format:

Print the **K<sup>th</sup>** character of the decrypted string corresponding to the given encrypted string **S**.

#### Sample Input:

a2b3cd3

8

#### Sample Output :

c

#### Explanation:

**S = "a2b3cd3". Corresponding** decrypted string = "**aabbcdcdcd**" . According to 1-based indexing for **S**, the **8<sup>th</sup>** character is 'c'.

#### Approach 1: Decryption

We will just iterate through encrypted string **S** and will create a decrypted string. Initially, take an empty decrypted string. Then decompress the string and its frequency one by one and append the current substring in the decrypted string by its frequency. Repeat the process till the end of string and print the **K<sup>th</sup>** character from the decrypted string.

#### Steps:

1. We will find the substring by traversing the string until no digit is found.
2. Then find the frequency of the preceding substring by traversing the string until no lowercase alphabet is found. We can use this relation to create the integer frequency from the string:

This will tell us how many times the string will be repeated in the decompressed string.

$$\text{freqOfSubstring} = \text{freqOfSubstring} * 10 + (\text{S}[j] - '0')$$

- a. Suppose we need to find the frequency of '**cd**' in '**cd231a2**'. We will traverse '**cd231a2**', beginning from '**2**' until no lower case alphabet is found while applying the above formula.
- b. In the first iteration, **freqOfSubstring** will be zero, and **s[j] = '2'**. Now,

$$\begin{aligned}\text{frequencyOfSubstring} &= 0 * 10 + ('2' - '0') \\ \text{frequencyOfSubstring} &= 0 * 10 + 2 \\ \text{frequencyOfSubstring} &= 2\end{aligned}$$

- c. In the second iteration, **freqOfSubstring** is **2**, and **s[j] = '3'**.

```

frequencyOfSubstring = 2 * 10 + ('3' - '0')
frequencyOfSubstring = 2 * 10 + 3
frequencyOfSubstring = 23

```

- d. In the third iteration, **freqOfSubstring** will be **23**, and **s[j]** = **'1'**.

```

frequencyOfSubstring = 23 * 10 + ('1' - '0')
frequencyOfSubstring = 23 * 10 + 1
frequencyOfSubstring = 231

```

- e. Finally, **frequencyOfSubstring** = **231**. After the last iteration, character '**a**' is found, thus terminating our loop.
3. We will append the substring **freqOfSubstring** times in decrypted string.
  4. We will keep repeating the above process until the whole string **S** is traversed.
  5. Finally, we have our decrypted string. Print the **(K - 1)<sup>th</sup>** index of the decrypted string.

**Time Complexity:** **O(N)**, where **N** is the length of the decrypted string corresponding to the encrypted string **S**.

**Space Complexity:** **O(N)**, where **N** is the length of the decrypted string corresponding to the encrypted string **S**.

### Approach 2: Without Decryption

We will iterate through Encrypted String **S** and will keep track of length passed in Decrypted String.

1. Find the length of the substring (**substringLength**) by traversing the string until no digit is found.
2. Next, find the frequency of the preceding substring by traversing the string until no lowercase alphabet is found. We can use this relation to create the Integer frequency from the string:

```
freqOfSubstring = freqOfSubstring * 10 + (S[j] - '0');
```

3. Find the length of the **resultant string** in the decrypted string.  

$$\text{resultantLength} = \text{freqOfSubstring} * \text{substringLength}.$$
For example, the **resultantLength** for substring '**cd23**' in the string "**cd23b4**" will be 46.
4. If the **resultantLength** of the repeated substring is less than **K**, then the required character is present in some later substring.
5. Subtract the **resultantLength** of the repeated substring from **K** to keep the number of characters required to be visited.

$$K = K - \text{resultantLength}$$

For example, if K was 48, our new K would be  $48 - 46 = 2$ . This means the Kth character we are looking for lies on the 2nd index of the string after 'cd23', which is 'b4'. Thus, we will repeat our entire process for 'b4'.

6. If the length of repeated substring is greater than or equal to **K**, the required character lies in the current substring, return the **(K - 1) % substringLength** character from the current substring.
7. We will keep repeating the above process until we reach our **K<sup>th</sup>** character.

**Time Complexity:** **O(N)**, where **N** is the length of encrypted string **S**.

**Space Complexity:** **O(1)**, as we are not creating any decrypted string and just keeping the length of string and substring in constant space.

## 5. Shortest substring with all characters [<https://coding.ninja/P25>]

**Problem Statement:** You have been given a string **S** which only consists of lowercase English-Alphabet letters. Your task is to find the shortest (minimum length) substring of **S** which contains all the characters of **S** at least once. If there are many substrings with the shortest length, find one which appears earlier in the string—that is, a substring whose starting index is lowest.

**For example:**

If the given string is **S = "abcba"**, then the possible substrings are "**abc**" and "**cba**". As "**abc**" starts with a lower index—that is, 0 and "**cba**" starts with index 2, we will print "**abc**" as our shortest substring that contains all characters of **S**.

**Sample Input:**

The only line of input contains a string **S**—that is, the given string.

**Sample Output:**

The only line of output contains a string—that is, the shortest substring of **S** which contains all the characters of **S** at least once.

**Sample Input:**

aabcabb

**Sample Output:**

abc

**Explanation:**

Some of the possible substrings are "aabcabb", "aabc", "abca", "abc", etc. Out of all these substrings, we will have "abc", "bca", and "cab" with the shortest length. As "abc" appears earliest in the string, we will print "abc" in the output.

## Approach 1: Naïve Solution

The problem boils down to counting distinct characters present in the string and then finding the minimum length substring that contains these many distinct characters at least once. We can check all substrings by two nested loops and maintain a count of distinct characters for each substring with the help of a hashmap. We process this substring whenever the **count** of distinct characters for a substring equals the count of distinct characters in the given string. We will choose the minimum length substring out of all these substrings.

### Steps:

1. Store all distinct characters of the string in a hashmap.
2. Take a variable **count** and initialise it with value zero.
3. Generate the substrings by using two pointers.
4. Now check whether generated substring is valid or not:
  - a. If we find that the character of the substring generated has not been encountered before, increment **count** by 1.
  - b. We can use a visited array of maximum chars size to find whether the current character has been encountered before or not.
  - c. If **count** is equal to the size of hashmap, the substring generated is valid.
  - d. If it is a valid substring, compare it with the minimum length substring already generated.

**Time Complexity:**  $O(N^2)$ , where **N** is the length of the string. We will run two nested loops for checking all the  $N*(N+1)/2$  substrings of the given string and hashmap operations take constant time.

**Space Complexity:**  $O(K)$ , where **K** is the number of distinct characters in the string. We are using two hashmaps and their size will not exceed the count of distinct characters in the string.

## Approach 2: Two Pointers

The idea is to use two pointers technique with a sliding window of variable length. The current window will be our current processing substring.

- If the **count** of distinct characters of the current window is equal to that of the given string, it makes no sense in expanding the window because the **count** will only increase by doing so.
- Similarly, if the **count** of the distinct characters is less than that of the given string, there is no benefit in shrinking the window.

We will keep track of the minimum length substring obtained so far and only update it when we find a substring of a smaller length.

### **Steps:**

1. Add all the characters of the given string to a **hashSet**.
2. Initialise **distCount** to the size of **hashSet**.
3. Initialise **start** and **end** pointers to zero, as currently, our window has size one.
4. Initialise **count** to zero, which will keep track of the number of distinct characters in the current window/substring.
5. Declare a hashmap **frequency** to store current window characters.
6. Initialise **minimumLength** to the length of the given string and **startIndex** to zero.
7. Loop till **end < length** of the string
  - a. Add **s[end]** to the window—that is, increment **frequency[s[end]]** by one.  
If after this **frequency[s[end]] == 1**, increment ‘count’ by one.
  - b. Shrink window till **count** is equal to **distCount**
    - i. Update **minimumLength** and **startIndex** if the current window is shorter.
    - ii. Remove **s[start]** from the window—that is, decrement **frequency[s[start]]** by one.  
If after this **frequency[s[start]] == 0**, decrement **count** by one.
    - iii. Shrink window size—that is, Increment **start** pointer by one.
  - c. Expand window size—that is, Increment **end** pointer by one.
8. Return **minimumLength**.

**Time Complexity:**  $O(N)$ , where **N** is the length of the given string. When all the characters in the string are the same, we will have our minimum length substring as a single character (size = 1). So, in the worst case, we will traverse every character twice, one time with each pointer.

**Space Complexity:**  $O(K)$ , where **K** is the number of distinct characters in the string. We are using two hashmaps and their size will not exceed the count of distinct characters in the string.

## **6. Match Specific Pattern** [<https://coding.ninja/P26>]

**Problem Statement:** Ninja has given you a list of **WORDS** and a **PATTERN** string. Your task is to find all such words in the list which match the given pattern. A valid match is found if and only if every character in the pattern is uniquely mapped to a character in a word.

### **Example:**

Let the list of words be {cod, zcz} and the pattern be “nin”. For each word in the list, we will check whether the word matches the pattern or not:

**For the word “cod”:** Letter ‘n’ in the pattern maps to the letter ‘c’ in the word. The letter ‘i’ in the pattern maps to the letter ‘o’ in the word. Letter ‘n’ in the pattern maps to the letter ‘d’ in the word. The same letter ‘n’ in the pattern maps to two different letters, ‘c’ and ‘d’, in the word. Hence, “cod” is not a valid match.

**For the word “zcz”:** Letter ‘n’ in the pattern maps to letter ‘z’ in the word. Letter ‘i’ in the pattern maps to letter ‘c’ in the word. Letter ‘n’ in the pattern maps to letter ‘z’ in the word. As every

letter in the pattern maps uniquely to a corresponding letter in the word. Hence “zc” is a valid match.

**Example:**

Words: cdd pcm

Pattern: foo

**Output:**

cdd

**Explanation:**

The list of words is {cdd, pcm} and the pattern is “foo”. For the word “cdd”: The letters ‘f’, ‘o’, ‘o’ of the pattern, map to letters ‘c’, ‘d’, ‘d’ of the word, respectively. As every letter in the pattern maps uniquely to a corresponding letter in the word. Hence, it is a valid match. For the word “pcm”: The letters ‘f’, ‘o’, ‘o’ of the pattern map to letters ‘p’, ‘c’, ‘m’ of the word, respectively. As the same letter ‘o’, in the pattern, maps to two different letters, ‘c’ and ‘m’ in the word, hence, “pcm” is not a valid match.

### Approach 1: Convert Word To String of Integers

The idea is to generate hashes of each word present in the list and compare this hash value with the hash of the pattern string.

- Firstly, it should be noted that a word can match the pattern only if the length of the word is equal to the length of the pattern. Therefore, before generating the hash value, we must compare the lengths to check whether they are equal or not to get the answer with ease.
- Otherwise, If the hash value of the pattern and the word matches, it tells us that the pattern and the word have the same structure — that is, it's a valid word.

How do we generate a hash for a given word or pattern?

- A simple way to generate the hash would be to assign a unique integer to every distinct character in the word (and pattern) one by one, thereby generating a string of integers. This string of integers acts as a hash value for the word (or pattern).
- If the hash value of the word matches with the pattern, the word is added to the list of valid words. Hence, we can use this check for the similarity between the word and the pattern.

**Let us understand this by an example.**

Let the word be “aababdeeeec” and the pattern be “ppqpssttvu”.

Generating the string of integers (hash) for the word:

a → 1

b → 2

d → 3

e → 4

$c \rightarrow 5$   
 $f \rightarrow 6$

Hash for the word: **11213344456**

Generating the string of integers (hash) for the pattern:

$p \rightarrow 1$   
 $q \rightarrow 2$   
 $s \rightarrow 3$   
 $t \rightarrow 4$   
 $v \rightarrow 5$   
 $u \rightarrow 6$

Hash for the pattern: **11213344456**

As hashes for word and pattern match each other, the word is a valid match.

**Time Complexity:**  $O(N * \text{length(pattern)})$ , where N is the number of words in the list. In the worst case, we convert every word (including pattern) to a string of integers. This gives us the complexity as  $O(N * \text{max}(\text{length of all provided strings}))$ . We can optimise it by checking if the length of the pattern is equal to the length of the word.

**Space Complexity:**  $O(\text{length(pattern)})$ . In the worst case, extra space is required by the map and the string of integers.

### Approach 2: Map Word to Pattern

Instead of generating a hash for every word and comparing it with the hash of the pattern, a better approach is to map the letters of the pattern and the corresponding letters in the word to each other.

Then, check whether every character in the pattern maps uniquely to a character in a word. And also, check whether every character in the word maps uniquely to a character in the pattern.

During the process of mapping, three cases may arise:

- **Case 1:** The **character in the pattern as well as the character in the word are not mapped** to any other character.
  - In such a case, map these characters to each other.
- **Case 2:** The **character in the word is already mapped** to some character in the pattern.  
In such a case, check whether the new value to which it is being mapped is the same as the value to which it is already mapped.
  - If it is the same, continue.
  - Otherwise, the same character in the word is getting mapped to two different characters of the pattern. Hence the word is **not a valid match**.
- **Case 3:** The **character in the pattern is already mapped** to some character in the word.  
In such a case, check whether the new value to which it is being mapped is the same as the value to which it is already mapped.
  - If it is the same, continue.

- Otherwise, the same character in the pattern is getting mapped to two different characters of the word. Hence the word is **not a valid match**.
- Return the list of valid words.

### **Let us understand this by an example:**

Let's assume the list of words is {abb, pqr} and the pattern given is "zxx".

Mapping the pattern and the word "abb":

```

z ↔ a
x ↔ b
x ↔ b

```

As every letter in the pattern maps uniquely to a corresponding letter in the word. Hence "abb" is a valid match.

Mapping the pattern and the word "pqr":

```

z ↔ p
x ↔ q
x ↔ r

```

The same letter 'x' in the pattern is getting mapped to two different letters, 'q' and 'r' in the word. Hence "pqr" is not a valid match.

### **Steps:**

- Create two maps, **mapPat** to store the mapping between the characters of the pattern to the characters of the word and another map, **mapWord**, to store the mapping between the characters of the word to the characters of the pattern.
- Loop 1:** For every word in the list:
  - Check if  $\text{length}(\text{word}) = \text{length}(\text{pattern})$ :
    - If true: move to step 2b.
    - Otherwise, the current word is not a valid match. Move to the next word.
  - Loop:** For  $i = 0$  to  $\text{length}(\text{pattern})$ :
    - Let  $\text{chW} = \text{word}[i]$  and  $\text{chP} = \text{pattern}[i]$ .
    - Check for Case 1: If  $\text{mapPat}[\text{chP}] = \text{NULL}$  and  $\text{mapWord}[\text{chW}] = \text{NULL}$ : then set  $\text{mapPat}[\text{chP}] = \text{chW}$  and  $\text{mapWord}[\text{chW}] = \text{chP}$ .
    - Otherwise, check for Case 2: If  $\text{mapPat}[\text{chP}] \neq \text{NULL}$  and  $\text{mapPat}[\text{chP}] \neq \text{chW}$ : then the current word is not a valid match, break.
    - Otherwise, check for Case 3: If  $\text{mapWord}[\text{chW}] \neq \text{NULL}$  and  $\text{mapWord}[\text{chW}] \neq \text{chP}$ : then the current word is not a valid match, break.
  - If the word is valid, add it to the list of valid words.

**Time Complexity:**  $O(N * \text{length}(\text{pattern}))$ , where N is the number of words in the list. In the worst case, we convert every word (including pattern) to a string of integers. This gives us the complexity as  **$O(N * \text{max}(\text{length of all provided strings}))$** . We can optimise it by checking if the length of the pattern is equal to the length of the word or not.

**Space Complexity:**  **$O(1)$** . In the worst case, constant extra space is required for the map.

## **7. Anagram Substring Search** [<https://coding.ninja/P27>]

**Problem Statement:** Given two strings **str** and **ptr**. Find all the starting indices of **ptr** anagram substring in **str**. Two strings are anagram if and only if one string can be converted into another string by rearranging the characters.

### **Note:**

Strings **str** and **ptr** consist only of letters of English alphabets in uppercase. The length of string **str** will always be greater than or equal to the length of string **ptr**. The index is 0-based. In case there is no anagram substring, then return an empty sequence.

### **Example:**

For example, the given **str** is '**BACDGABCD**' and **ptr** is '**ABCD**'. Indices are given as follows:

0-3 - String formed by the characters at 0, 1, 2 and 3 index in str is "BACD" which is an anagram with "ABCD"

1-4 in **str** index 1,2,3,4 are '**ACDG**', and it is not anagram with '**ABCD**'.

2-5 in **str** index 2,3,4,5 are '**CDGA**', and it is not anagram with '**ABCD**'.

3-6 in **str** index 3,4,5,6 are '**DGAB**', and it is not anagram with '**ABCD**'.

4-7 in **str** index 4,5,6,7 are '**GABC**', and it is not anagram with '**ABCD**'.

5-8 in **str** index 5,6,7,8 are '**ABCD**', and it is an anagram with '**ABCD**'.

Hence there are **two** starting indices of substrings in the string **str** that are anagram with given **ptr**. They are index 0 and 5.

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

In this approach, we will use two nested loops.

#### **Steps:**

- Consider **n** is the number of characters in the given strings **str** and **m** is the number of characters in the given string **ptr**.
- The outer loop runs from 0 to **n - m - 1** for starting indices '**i**' of anagram.
- For each character of **str**, we iterate from **i** to **i + m - 1** and check for an anagram with the given string **str**.
- For checking, we will sort all characters of **str** from **i** to **i + m - 1** with the help of another temporary character array and check if all characters of **str** and **ptr** are the same or not.
- If all characters of **str** and **ptr** are same, we consider **i** in our **answer** sequence, in which we are storing all the starting indices of the anagram.
- Then check for the next **i**.
- Finally, return the **answer** sequence.

**Time Complexity:**  $O(N * M * \log(M))$ , where **N** is the number of characters in the given string **str** and **M** is the number of characters in the given string **ptr**. For each, indices we are sorting the length of the **M** size characters array.

**Space Complexity:**  $O(\max(M, K))$ , where **M** is the number of characters in the given string **ptr** and **K** is the size of our **answer** sequence. In this approach, the **answer** sequence is a sequence in which we store all the starting indices which are anagrams.

### Approach 2: Sliding Window

In this approach, we will use the concept of a sliding window.

#### Steps:

1. Consider **n** is the number of characters in the given string **str** and **m** is the number of characters in the given string **ptr**.
2. We use array **strMap** to store the frequency of characters in **str** and **ptrMap** to store the frequency of characters in **ptr**.
3. The initial size of **strMap** and **ptrMap** is 26 and the value is '0'.
4. To make the frequency map, we simply increment the value at index '0' of the map if we encounter 'A' in our string. Similarly, we increment the value at index '1' by one if we encounter 'B' in our string.
5. Iterate *i* from 0 to **n - 1** on the string **str**. The value of **strMap[0]** will be incremented by one, if the character at **str[i]** is 'A', similarly the value of **strMap[1]** will increase by one if the character at **str[i]** is 'B'. Once the iteration over **str** is complete, our frequency map **strMap** will be ready.
6. Follow the same procedure to create **ptrMap** from the string **ptr**.
7. Check if **strMap** and **ptrMap** are same or not If both are same, then add '0' in the **answer** sequence in which we are storing all the starting indices of the anagram.
8. Now iterate *i* from **m** to **n - 1**.
9. In **strMap**, decrease the value of **str[i - m]** and increase **str[i]** by one.
10. In every iteration, check if **strMap** and **ptrMap** are same or not. If they are same, then add **i - m + 1** in the **answer** sequence.
11. Finally, return the **answer** sequence.

**Time Complexity:**  $O(N)$ , where **N** is the number of characters in given string **str**. A single iteration through the sequence is required.

**Space Complexity:**  $O(K)$ , where **K** is the size of our **answer** sequence. In this approach, the **answer** sequence is a sequence in which we store all the starting indices which are anagrams.

## 8. Next Smaller Palindrome [<https://coding.ninja/P28>]

**Problem Statement:** You are given a number **N** in the form of a string **S**, which is a palindrome. You need to find the greatest number strictly less than N which is also a palindrome.

**Note:**

1. A palindrome is a word, number, phrase, or another sequence of characters that reads the same backward as forward, such as 'madam', 'racecar', and 1234321.
2. The numerical value of the given string **S** will be greater than 0.
3. A single-digit number is also considered a palindrome.
4. The answer number should not contain any leading zeros, except when the answer is 0.
5. Note that the length of the string is nothing but the number of digits in **N**.

**Sample Input:**

12321

**Sample Output:**

12221

**Explanation:**

The next smaller palindrome to 12321 is 12221, as it is strictly less than 12321, and it reads the same from the front and back.

**Approach 1: Find the first place from the centre where you can update the number**

Given the fact that the input number is a palindrome itself makes this problem very trivial.

Let us assume that the given string **str** is a palindrome, so we know that **str** comprises of two halves **s1** and **s2**—that is, **str = s1 + s2**, where **s1** and **s2** are two strings, and **s2** is the reverse of **s1** (in case of odd length palindromes **s2** won't have the last character of **s1**). So a change in any of the half must also reflect in the other half to keep the output string a palindrome.

Hence, we can either traverse **s1** from the end or **s2** from the beginning—that is, from the middle of the string **str** to any of the directions, because in this way, we would have the option of updating lesser significant digits first.

Let us traverse **s2** from the beginning:

- If we find a position in **s2** such that **s2[pos] != 0** (digit at **s2[pos]** is not equal to 0), we can simply update **s2[pos] = s1[n - 1 - pos] = s2[pos] - 1**. And we are done—that is, we have reduced the current number and we simply return the string.
- However, if **s2[pos] = 0**, then we can't decrease this digit, and we will look further for non-zero digits. But before going ahead, we need to make **s2[pos] = s1[n - 1 - pos] + 9**, as we want the greatest number less than the current number, so if we keep this digit (and its corresponding digit in **s1**) as it is and decrease some more significant digit, then

this will not be the correct answer. For example, the next smaller palindrome of 2002 will be 1991 and not 1001.

There is one last case in this approach: consider the number 1001. For this number we can't reduce the most significant 1(and its corresponding 1 in s2) to 0 because in such a case, the number would have leading zeros. Therefore, in this case, we convert two zeros to 9 and two ones will be discarded, and then one extra 9 will be added. This is done because, as we can see, 1001 is a four-digit number and 99 is a two digit number, so we add an extra 9 and thus the answer will be 999.

**Time Complexity:**  $O(N)$ , where  $N$  is the length of the given palindrome.

**Space Complexity:**  $O(1)$ , as we are using constant extra memory.

## **9. Longest Substring Without Repeating Characters**

[<https://coding.ninja/P29>]

**Problem Statement:** Given a string  $S$  of length  $L$ , return the length of the longest substring without repeating characters.

**Sample Input:**

xyxyz

**Sample Output:**

3

**Explanation:**

The substrings without repeating characters are "xy", "yx", "xyz", "yz", "z". The longest substring out of these substrings is "xyz" of length 3.

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

In the brute force approach, we will use two nested loops. The outer loop is used to select the starting index of the substring and the inner loop is used to fix the ending index of the substring. After selecting the substring, we will use another loop (or a method) to check whether the substring contains all unique characters or not using a HashSet.

**Time Complexity:**  $O(L^3)$ , where  $L$  is the length of the input string. In the worst case, we will be selecting all substrings by choosing two pointers  $i, j$  using two nested loops, which will be of  $O(L^2)$  complexity. For each selected substring, we are checking whether it contains all unique characters or not, which constitutes  $O(L)$  complexity. Thus, overall time complexity will be  $O(L^2 * L) = O(L^3)$ .

**Space Complexity:**  $O(L)$ , where  $L$  is the length of the input string. In the worst case, we will be storing all the characters of the string in the set.

### Approach 2: Optimized Brute Force

- In this approach, we will use two nested loops. The outer loop is used to select the starting index of the substring and the inner loop is used to fix the ending index of the substring.
- Inside the outer loop, we will initialise a HashSet to store the unique characters. Now we will keep adding the characters to this set using the inner loop. Before adding each character—if the character is already present in the set—it means that from  $i^{\text{th}}$  index to the current  $j^{\text{th}}$  index, this character is repeated. Hence, we will not consider this substring and any other substring starting from the  $i^{\text{th}}$  index and ending at  $j$  or after. In this case, we will break from the inner loop. Otherwise, we will update the longest length and add the character to the set.

**Time Complexity:**  $O(L * D)$ , where  $L$  is the length of the input string and  $D$  is the maximum number of distinct characters. In the worst case, we will be only selecting all substrings by choosing two pointers  $i, j$  using two nested loops, and whenever we encounter a repeated character, we will break from the inner loop. Thus the complexity will only remain  $O(N * D)$ .

**Space Complexity:**  $O(D)$ , where  $D$  is the maximum number of distinct characters. In the worst case, we will be storing all the distinct characters of the string in the set.

### Approach 3: Binary Search

- In this approach, we will use a binary search to select the length of the substring. The smallest value possible will be 1 as all the characters are unique substrings themselves. The maximum value for binary search will be  $L$  as there is no substring with a length more than  $L$  in the given string where  $L$  is the length of the given input string.
- Now, we will perform binary search on the range **start = 1** to **end = L**. We will check for  $\text{mid} = (\text{start} + \text{end})/2$  length. We need to check whether there is any substring with length equal to **mid** having unique characters or not. If such substring exists, we will update the answer and move to the right side — **start = mid + 1**. Otherwise, we will check on the left side — **end = mid - 1**.
- Now to check whether a substring of length **mid** exists or not, we will iterate on the whole string.

#### Steps:

1. Initialize the map to store frequencies.
2. For each character in the input—**i = 0 to L - 1**, where  $L$  is the length of the input string.
  - a. Increment the current character's frequency in the map.
  - b. If **i >= mid**, decrease the frequency of  $(i - \text{mid})^{\text{th}}$  character. If it is zero, remove it from the map.

- c. If the map's current size equals **mid-value**, return true.
- 3. If no substring of length equal to **mid** found, return false.

**Time Complexity:**  $O(L * \log(L))$ , where **L** is the length of the input string. In the worst case, we will be selecting length using binary search  $O(\log(L))$  and for all such lengths iterating the string to check unique substring existence.

**Space Complexity:**  $O(L)$ , where **L** is the length of the input string. In the worst case, we will be storing all the characters of the string in the map.

#### Approach 4: Sliding Window Approach

We will keep a window that will keep a range of unique characters where the range is defined by start and end indices. Initially, both start and end will be equal to zero, denoting that we have only one character in the current window—the character at 0th index. Along with this, we will also keep a **HashSet** to store the characters in the window.

Now, we will keep incrementing the end to iterate on each character of the string. If the current character at the end index is not present in the window, we will simply add it to the window (add to the **HashSet**), increment the end, and update the maximum length. Otherwise, we will remove the start character from the window and increment the start index.

**Time Complexity:**  $O(L)$ , where **L** is the length of the input string. In the worst case, we will be adding and removing each character of the string only once from the HashSet. This leads to total  $(2 * L)$  ( $L$  times adding and  $L$  times removing) operations, and hence the time complexity will be  $O(L)$ .

**Space Complexity:**  $O(L)$ , where **L** is the length of the input string. In the worst case, we will be storing all the characters of the string in the set.

#### Approach 5: Optimized Sliding Window Approach

In approach 4, we have used  $2 * L$  steps. We can optimise the number of steps to **L** using **HashMap**. Suppose the current element exists in the window. In that case, we can skip all the characters from the start to the previous index of the current character and update the start accordingly. Also, add/update the index of the current character in **HashMap**.

**Time Complexity:**  $O(L)$ , where **L** is the length of the input string. In the worst case, we will be adding each character of the string only once from the HashSet.

**Space Complexity:**  $O(L)$ , where **L** is the length of the input string. In the worst case, we will be storing all the characters of the string in the map.

## 10. Minimum Operations to Make Strings Equal [<https://coding.ninja/P30>]

**Problem Statement:** Given two strings **A** and **B** consisting of lower case English letters. The task is to count the minimum number of pre-processing moves on the string **A** required to make it equal to string **B** after applying below operations:

1. Choose any index  $i$  ( $0 \leq i < n$ ) and swap characters  $a[i]$  and  $b[i]$ .
2. Choose any index  $i$  ( $0 \leq i < n$ ) and swap characters  $a[i]$  and  $a[n - i - 1]$ .
3. Choose any index  $i$  ( $0 \leq i < n$ ) and swap characters  $b[i]$  and  $b[n - i - 1]$ .

**In one pre-process move, you can replace a character in A with any other English alphabet letter.**

**Note:**

The number of changes you make after the preprocess move does not matter. You cannot apply to preprocess moves to String B or make any preprocess moves after the first change is made.

**Sample Input:**

abacaba bacabaa

**Sample Output:**

4

**Explanation:**

For the first test case, preprocess moves are as follows: **A[0] = 'b'**, **A[2] = 'c'**, **A[3] = 'a'** and **A[4] = 'b'**. Afterwards, **A = "bbcabba"**. Then we can obtain equal strings by the following sequence of changes: swap(A[2], B[2]) and swap(A[2], A[6]). There is no way to use fewer than 4 preprocess moves before a sequence of changes to make strings equal. Hence the answer in this test case is 4.

**Approach 1: Using HashMap**

Let's divide all characters of both strings into groups so that characters in each group can be swapped with each other with changes. So, there will be the following groups: {A[0], A[n - 1]}, {B[0], B[n - 1]}, {A[1], A[n - 2]}, {B[1], B[n - 2]} and so on... Since these groups don't affect each other, we can calculate the number of pre-processing moves in each group and then add them.

Now the question is, how to determine if a group does not need any pre-processing moves?

For a group consisting of 2 characters (there will be one such group if  $n$  is odd), that's easy - If the characters in this group are equal, the answer is 0; otherwise, it's 1.

To determine the required number of pre-processing moves for a group consisting of four characters, we may use the following fact: this group doesn't require any pre-processing moves if the characters in this group can be divided into pairs. So if the group contains four equal

characters or two pairs of equal characters, then the answer for this group is 0. Otherwise, we may check that replacing only one character of  $A[i]$  and  $A[n - i - 1]$  will be enough. If so, then the answer is 1; otherwise, it's 2.

We will use HashMap to count the number of distinct characters in a particular group. The key in the HashMap will be the character itself, and the value corresponding to that key will be its frequency.

**Time Complexity:  $O(N)$** , where  $N$  is the length of the given strings. Since we are traversing in the given string only once. Also, insertion and searching operations take constant time into the HashMap. So, overall time complexity will be  $O(N)$ .

**Space Complexity:  $O(1)$** . Since no extra memory is used except a HashMap of size 4, which is constant for the length of the string. Therefore, overall space complexity will be constant.

### Approach 2: Using Variables

Let's divide all characters of both strings into groups in such a way that characters in each group can be swapped with each other with changes. So, there will be the following groups:  $\{A[0], A[n - 1], B[0], B[n - 1]\}$ ,  $\{A[1], A[n - 2], B[1], B[n - 2]\}$  and so on. Since these groups don't affect each other, we can calculate the number of pre-processing moves in each group and then sum it up.

Now, for each group of letters  $A[i]$ ,  $A[n - i - 1]$ ,  $B[i]$  and  $B[n - i - 1]$  (further denoted as  $c1$ ,  $c2$ ,  $c3$  and  $c4$  respectively), we have all these possible cases:

- The group that doesn't require any preprocessing move, if:
  - All four characters are same—that is,  $c1 == c2 == c3 == c4$   
OR
  - Pairwise equal characters—that is,
    - $c1 == c2 \&& c3 == c4$
    - $c1 == c3 \&& c2 == c4$
    - $c1 == c4 \&& c2 == c3$
- The group that requires only one preprocessing move, if:
  - $c1 == c2$
  - $c1 == c3$
  - $c1 == c4$
  - $c2 == c3$
  - $c2 == c4$
  - $c3 == c4$
- All the remaining cases require two preprocessing moves.

For every group of letters, we analyse according to the above cases. The strings with odd numbers of letters represent one edge case. Therefore, the letter in the middle must be taken into account separately—that is, if the characters in this group are equal, the answer is 0. Otherwise, it's 1, because we need to change it.

**Time Complexity:  $O(N)$** , where  $N$  is the length of the given strings. Since we are traversing the string only once. Thus, the time complexity will be  **$O(N)$** .

**Space Complexity:  $O(1)$** —that is, constant. Since no extra memory is used except for some variables, thus, the space complexity will be  **$O(1)$** .

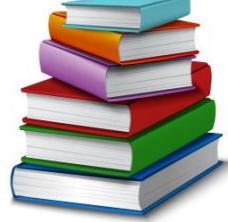
# 4. Stacks and Queues

---

## Introduction to Stacks

Stacks are data structures that allow us to store and retrieve data sequentially. A stack is a linear data structure like arrays and linked lists. It is an abstract data type (**ADT**). In a stack, the order in which the data arrives is essential. It follows the LIFO order of data insertion. LIFO stands for **Last In First Out**.

Consider the example of a pile of books:



Unless the book at the topmost position is removed from the pile, we can't have access to the second book from the top and so on. When we apply the same technique to data in our program, this pile-type structure is called a **stack**.

In a stack, the top-most element is the first one to be retrieved or removed. In a pile of books, we can insert a book only at the top of the pile. Correspondingly, the element that made its entry at the end in a stack would be the first one to be retrieved. This establishes the **Last In First Out** order.

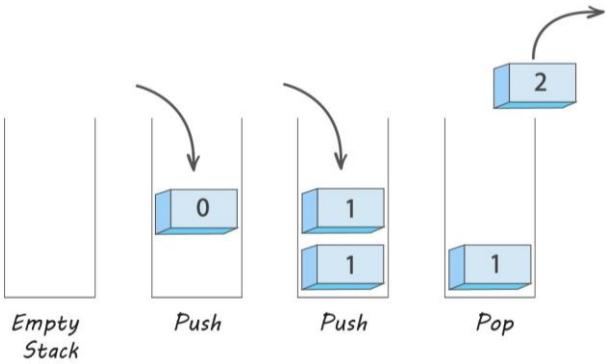
## Operations on Stacks

In a stack, insertion and deletion are done only at one of the stack ends, which is called the **top**.

- **Insertion:** Inserting an element from the top is called a **push** operation.
- **Deletion:** Deleting an element from the top is called a **pop** operation.

### Main stack operations

- **push (data):** Inserts data onto the stack.



- **pop ()**: Removes the last inserted element from the stack.

### Auxiliary stack operations

- **top()**: Returns the last added element without removing it.
- **size()**: Returns the number of elements stored in the stack.
- **isEmpty()**: Indicates whether any elements are stored in the stack or not—that is, whether the stack is empty or not.

## Implementation of Stacks

Stacks can be implemented using arrays, linked lists, or queues. The underlying algorithm for implementing operations of stack remains the same.

### Push operation

```
function push(data, stack)

    // data: the data to be inserted into the stack.

    If stack is full
        return null
    /*
        top: It refers to the position (index in arrays) of the last
        element into the stack
    */
    top = top + 1
    stack[top] = data
```

### Pop operation

```

function pop(stack)

    If stack is empty
        return null

    // Retrieving data of the element to be popped.
    data = stack[top]

    // Decrementing top
    top = top - 1

```

### Top operation

```

function top(stack)

    If stack is empty
        return null
    else
        return stack[top]

```

### isEmpty operation

```

function isEmpty(stack)

    If top is null
        return true
    else
        return false

```

## Time Complexity of Various Operations

Let **n** be the number of elements present in the stack. The complexities of stack operations with this representation can be given as:

Operations	Time Complexity
Push(data)	O(1)
Pop()	O(1)
int top()	O(1)
boolean isEmpty()	O(1)
int size()	O(1)
boolean isFull()	O(1)

## Exceptions

Attempting the execution of an operation may sometimes cause an error condition, which is called an exception.

Exceptions are thrown by an operation that cannot execute. Following are the two exceptions we see in case of stacks:

- Attempting the execution of pop() on an empty stack throws an exception called Stack Underflow.
- Trying to push an element in a full-stack throws an exception called Stack Overflow.

## Applications of Stacks

- Stacks are useful when we need dynamic addition and deletion of elements in our data structure. Since stacks require O(1) time complexity for all the operations, we can achieve our tasks very efficiently.
- It also has applications in various **popular algorithmic problems** such as:
  - Tower of Hanoi,
  - Balancing Parenthesis,
  - Infix to postfix, and
  - Backtracking problems
- It is useful in many graph algorithms like topological sorting and finding strongly connected components.
- Apart from the above, stacks are also used in practical applications like:
  - Undo and Redo operations in editors
  - Forward and backward buttons in browsers
  - Allocation of memory by an operating system while executing a process.

## Introduction to Queues

**Queues** are simple data structures in which insertions are done at one end, and deletions are done at the other end. It is a linear data structure such as arrays. It is an abstract data type (**ADT**).

The first element to be inserted is the first element to be deleted. It follows **FIFO (First In First Out)** order.

Consider the queue as the line of people.



When we enter a queue, we stand at the end of the line, and the person at the front of the line is served first. After the first person is served, he will exit the queue, and as a result, the next person will come at the start of the queue. Similarly, the first person in the queue will keep exiting the queue as we move towards the front of the queue. This behaviour is useful when we need to **maintain the order of arrival**.—that is, **FIFO**.

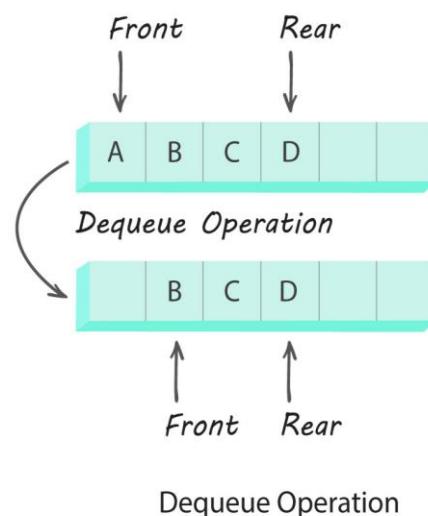
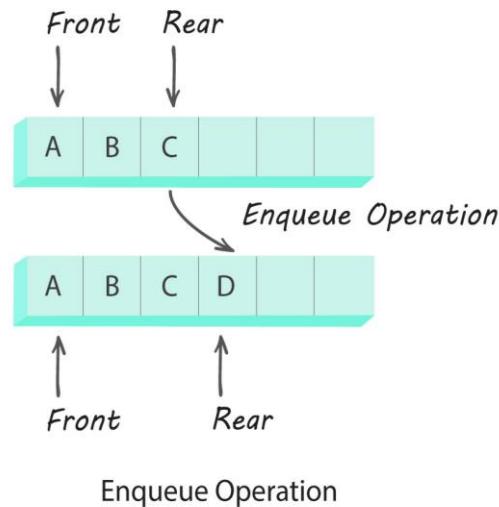
## Operations on Queues

In queues, insertion is done at one end (rear), and deletion is done at the other end (front):

- **Insertion** (Adding elements at the rear): Inserting an element into the queue is called **enqueue**. Enqueueing an element when the queue is full is called **overflow**.
- **Deletion** (Removing elements from the front): The operation of deleting an element from the queue is called **dequeue**. Deleting an element from the queue when the queue is empty is called **underflow**.

### Main Queue Operations:

- `enqueue(data)`: Insert data in the queue (at rear)
- `dequeue()` : Deletes and returns the first element of the queue (at front).



#### Auxiliary Queue Operations:

- `front()`: returns the first element of the queue
- `size()`: returns the number of elements present in the queue
- `isEmpty()`: returns whether the queue is empty or not

## Implementation of Queues

Queues can be implemented using arrays, linked lists, or stacks. The basic algorithm for implementing a queue remains the same. We maintain two variables for all the operations—**front** and **rear**.

#### Enqueue Operation

```
function enqueue(data)
```

```

if queue is full
    return "Full Queue Exception"

else
    rear++
    queue[rear] = data

```

### Dequeue Operation

```

function dequeue()

if queue is empty
    return "Empty Queue Exception"

else
    temp = queue[front]
    front++

```

### getFront() Operation

```

function getFront()

if queue is empty
    return "Empty Queue Exception"

else
    temp = queue[front]
    return temp

```

## Time Complexity of Various Operations

Let **n** be the number of elements present in the queue. The complexities of queue operations with this representation can be given as:

Operations	Time Complexity
<b>enqueue(data)</b>	$O(1)$
<b>dequeue</b>	$O(1)$
<b>int getFront()</b>	$O(1)$
<b>boolean isEmpty()</b>	$O(1)$
<b>int size()</b>	$O(1)$

## Applications of Queues

## Real Life Applications

- Scheduling of jobs in the order of arrival by the operating system
- Implementing printer spooler so that printing jobs can be executed as per their order of arrival
- Handling of interrupts in the real-time systems. The interrupts are generally handled in the same order as they arrive—that is, FIFO.
- Handling multi-user, multiprogramming, and time-sharing environments. Nowadays, systems handle several jobs at a time. To handle these jobs efficiently, queues are used.

## Application in solving DSA problems

- Queues are useful when we need **dynamic addition and deletion of elements in our data structure**. Since queues require O(1) time complexity for all the operations, we can achieve our tasks very efficiently.
- It is useful in many **graph algorithms** such as:
  - Breadth first search,
  - Dijkstra's algorithm
  - Prim's algorithm.

---

## Practice Problems

### **1. Interleave the First Half of the Queue with the Second Half**

[<https://coding.ninja/P31>]

**Problem Statement:** You have been given a queue of integers. You need to rearrange the queue elements by interleaving the elements of the first half of the queue with the second half.

**Note:**

The given queue will always be of even length.

**For Example:**

**N = 10**

**Q = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]**

the output will be **Q = [10, 60, 20, 70, 30, 80, 40, 90, 50, 100]**.

**Input Format:**

The **first line** contains an integer **T** denoting the number of test cases or queries to be run. For each of the test cases:

The **first line** contains an integer **N** which denotes the size of the queue.

The **second line** contains elements of the queue separated by a single space.

#### **Output Format:**

For each test case, print the queue elements after interleaving the elements of the first half with the second half.

#### **Sample Input:**

```
1
10
10 20 30 40 50 60 70 80 90 100
```

#### **Sample Output:**

```
10 60 20 70 30 80 40 90 50 100
```

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

The basic idea is to use a stack to interleave the elements of the queue.

#### **Steps:**

1. Create an empty stack.
2. Push the elements of the first half of the queue to stack.
3. Enqueue back the elements of the stack (enqueue operation in queue happens from the back/rear side)
4. Dequeue the elements of the first half of the queue and enqueue them back at the rear side.
5. Again push the first half elements into the stack.
6. Interleave the elements of queue and stack (alternately push the top element of the stack and front element of the queue into the queue).

**Time Complexity:**  $O(N)$ , where **N** is the size of the queue.

**Space Complexity:**  $O(N)$ , where **N** is the size of the stack. The reason is an auxiliary stack of the size to that of the original queue is used in interleaving operation.

## **2. Next Greater Element** [<https://coding.ninja/P32>]

**Problem Statement:** You are given an array **arr** of length **N**. You have to return a list of integers containing the **NGE** (next greater element) of each element of the given array. The **NGE** for an element **X** is the first greater element on the right side of **X** in the array. Elements for which no greater element exists, consider the **NGE** as -1.

**Example:** If the given array is [1, 3, 2], you need to return [3, -1, -1].

For 1: 3 is the next greater element

For 3: There is no greater element to its right. Thus, -1 is returned

For 2: There is no greater element to its right. Thus, -1 is returned.

#### **Input Format:**

The **first line** of the input contains a single integer **T**, representing the number of test cases or queries to be run.

For each test case:

The **first line** contains an integer **N**, representing the length of the input array (**arr**).

The **second line** contains **N** space-separated integers representing elements of the array **arr**.

#### **Output Format:**

For each test case, print a list of integers, each representing the **NGE** (next greater element) of each element of the given array in a single line.

#### **Sample Input:**

1

4

9 3 6 5

#### **Sample Output:**

-1 6 -1 -1

#### **Explanation:**

For element 9, there is no next greater element. Hence, -1 would be the next greater element.

For element 3, 6 is the next greater element. For element 6, there is no next greater element.

Hence, -1 would be the next greater element. For element 5, there is no next greater element.

Hence, -1 would be the next greater element.

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

#### **Steps:**

1. Initialise an array **ans** of length **N** to store the next greater number of **arr[i]** at each index *i*.
2. Traverse the given array and for every element at index *i*.
  - a. Initialise a variable **next** = -1.
  - b. Run a loop from *i* + 1 to **N**, and check whether the current element is greater than **arr[i]** and then update the **next** to the current element and break the loop.
  - c. Store **next** at **ans[i]**.
3. Return **ans** array.

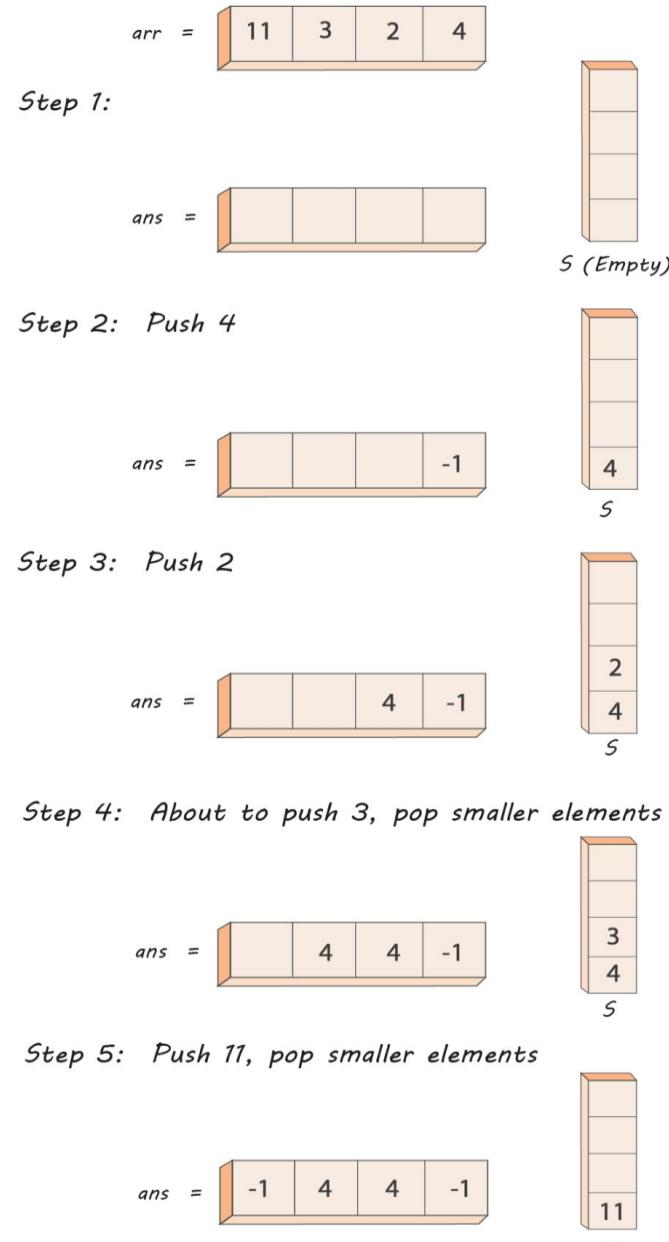
**Time Complexity:**  $O(N^2)$ , where  $N$  is the length of the array **arr**. In the worst case, we will be searching for the next greater element of each element. For searching the next greater element in **arr**, it can take  $O(N)$  time. Thus total time would be  $O(N^2)$ .

**Space Complexity:**  $O(1)$ , only a constant extra space is required in the worst case.

### Approach 2: Optimal Approach (Using Stacks)

#### Steps:

1. Initialise an array **ans** of length  $N$  to store the next greater number of **arr[i]** at each index  $i$ .
2. Create a stack of type integer, where we will store the smaller element at the top.
3. Traverse the array from backwards because when we arrive at a certain index, its next greater element will be already in the stack, and we can easily get this element.
4. For every element at index  $i$ ,
  - a. Pop the elements from the stack till we get the greater element on top of the stack from the current element. The same element will be the **NGE** for the current element.
  - b. If the stack gets empty while doing the pop operation, then the answer would be  $-1$ .
  - c. Store the next greater element in the array and push the current element of the array into the stack.
5. Finally, return the array **ans**.



**Time Complexity:**  $O(N)$ , where  $N$  is the length of the array  $\text{arr}$ . In the worst case, we will be pushing and popping all the elements into the stack only once. Thus time complexity would be  $O(N)$ .

**Space Complexity:**  $O(N)$ , in the worst case, all the elements can be in the stack.

### 3. Valid Parentheses [\[https://coding.ninja/P33\]](https://coding.ninja/P33)

**Problem Statement:** You're given string  $\text{str}$  consisting solely of “{”, “}”, “(”, “)”, “[” and “]”. Determine whether the parentheses are balanced.

**Input Format:**

The **first line** of the input contains an integer **T** which denotes the number of test cases or queries to be run.

For each test case:

The **first and the only line** contains a string, as described in the task.

#### **Output Format:**

For each test case, the first and the only line of output prints whether the string or expression is balanced or not.

#### **Sample Input 1:**

1  
[{}]{[()]}()

#### **Sample Output 1 :**

Balanced

#### **Explanation:**

There is always an opening brace before a closing brace,i.e. i.e '{' before '}', '(' before ')', '[' before ']'.

#### **Sample Input 2:**

[()]

#### **Sample Output 2:**

Not Balanced

#### **Explanation:**

There is a closing brace for '[' i.e. ']' before closing brace for '(' i.e. ')'. The balanced sequence should be '[()'].

### **Approach 1: Valid Parentheses**

Make use of the stack. Traverse the string and push the current character in the stack; if it is an opening brace, else pop from the stack. If it is the corresponding starting brace for current closing brace, move to the next character of the string; otherwise, return false.

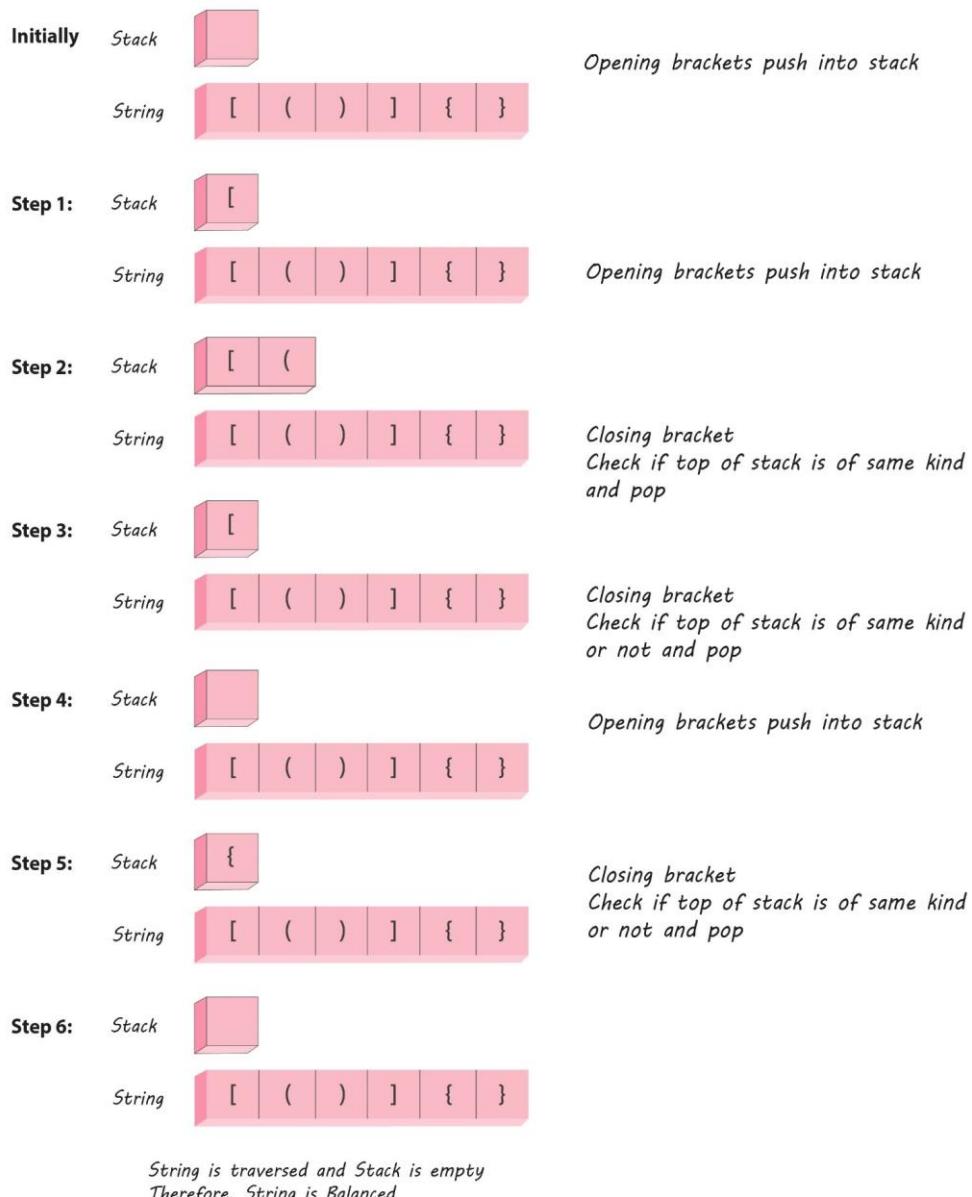
After complete traversal, if the stack is empty, the string is balanced, else it is not balanced.

#### **Steps:**

1. Declare a character stack.
2. Now traverse the expression string.
  - a. If the current character is a starting bracket ( '**' or '**' or '**' ), then push it onto the stack.******

- b. If the current character is a closing bracket ( '}' or '}' or ']' ) then pop it out from the stack, and if the popped character is the matching starting bracket, then they are balanced, else they are not.
- 3. After complete traversal, if there is some starting bracket left in the stack, then (print) or (return) or (the string is) "**not balanced**".
- 4. Otherwise, the string is **balanced**.

**Explanation** str = [(){}]



**Time Complexity:**  $O(N)$ , as a traversal of the string or expression is done only once, where  $N$  is the length of the input string.

**Space Complexity:**  $O(N)$ , as the maximum stack size reaches the length of the string, where  $N$  is the length of the input string.

## 4. Two Stacks [<https://coding.ninja/P34>]

**Problem Statement:** Design a data structure representing two stacks using only one array common for both stacks. The data structure should support the following operations:

**push1(num):** Push **num** into **stack1**.

**push2(num):** Push **num** into **stack2**.

**pop1():** Pop out the top element from **stack1** and return the popped element in case of underflow return -1.

**pop2():** Pop out the top element from **stack2** and return the popped element in case of underflow return -1.

**There are two types of queries in the input.**

**Type 1** - These queries correspond to **Push** operation.

**Type 2** - These queries correspond to **Pop** operation.

**Note:**

1. You are given the **size** of the array.
2. You need to perform **push**, **just**, and **pop** operations; postfix expression evaluation is faster, i.e., we can push elements in the stack till there is some empty space available in the array.
3. While performing Push operations, do nothing in case of the overflow of the stack.

**Input format:**

The **first line** contains two space-separated integers, **s** and **q**, denoting the size of the array and the number of operations to be performed, respectively.

The **next q lines** contain operations, one per line.

**Each line** begins with two integers, '**type**' and '**stackNo**', denoting the type of query mentioned above and the stack number on which this operation will be performed.

If the '**type**' is 1, then it will be followed by one more integer '**num**' denoting the element needed to be pushed to stack '**stackNo**'.

**Output format:**

For each operation of Type 2 (Pop Operation), print an integer on a single line, the popped element from the stack if the stack is already empty, print -1.

**Sample Input:**

3 5

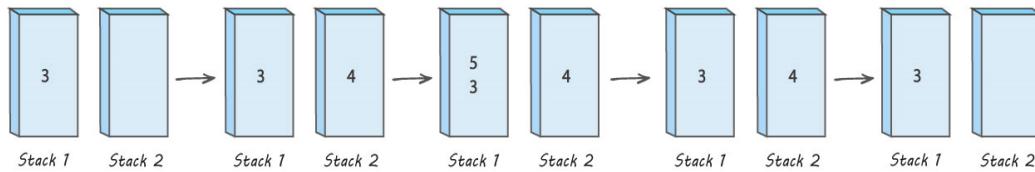
1 1 3

```
2 1 4  
1 1 5  
2 1  
2 2
```

### Output:

```
5 4
```

### Explanation:



Here every step shows a snapshot of two stacks after each operation.

Initialising the size of the array to 3, `twoStack = new TwoStack(3)`.

Then operation on two stacks occurs as follows:

```
twoStack.push1(3)    // pushing 3 in stack1  
twoStack.push2(4)    // pushing 4 in stack2  
twoStack.push1(5)    // pushing 5 in stack1  
twoStack.pop1()      // popping out from stack2, it returns 5  
twoStack.pop2()      // popping out from stack2, it returns 4.
```

### Approach 1: Optimal solution

1. To utilise the array space optimally, we start the top of both stacks from the two extremes of the array.
  - a. Let the array used by both the stacks is **Arr**, having size equal to **s**.
  - b. Let's say **top1** and **top2** points to the top of the stacks, **stack1** and **stack2** respectively.
  - c. As the size of the array is **s**, we assign -1 to **top1** and **s** to **top2** (keeping 0-based indexing in mind) which denotes both stacks are currently empty.
2. To complete push operations (or operation of Type 1), we do as follows:
  - a. **push1(num):**
    - i. Check if **Arr** has enough space to push an integer or not, i.e, space between top elements of both stacks. For this, we check **top1 + 1 < top2**. If this is the case, we have enough space. Thus, we increment **top1** by 1 and assign **num** to **Arr[top1]**.
    - ii. If there is insufficient space for pushing another element, the condition would reach **top1 + 1 == top2**, and we will return.
  - b. **push2(num):**

- i. Check if **Arr** has enough space to push an integer or not. For this, we check **top2 - 1 > top1**. If this is the case, then we have enough space. Thus we decrement **top2** by 1 and assign **num** to **Arr[top2]**.
  - ii. If there is insufficient space for pushing another element, the following condition would reach: **top2 - 1 == top1**, hence. we return.
3. In order to complete pop operations (or operations of Type 2), we do as follows:
- a. **pop1()**:
    - i. Firstly, check if **stack1** is empty or not. To do that, we need to check if **top1** is -1 or not. If it is equal to -1, then it is the condition of underflow (empty stack). Hence, we return -1.
    - ii. If **stack1** is not empty, then we decrement **top1** by 1 and return the value of **Arr[top1 + 1]**.
  - b. **pop2()**:
    - i. First, we check if **stack2** is empty or not, for which we just need to check if **top2** is equal to 's' or not. If it is equal to 's' then, it is the condition of underflow, and we just return -1.
    - ii. If **stack2** is not empty, we increment **top2** by 1 and return **Arr[top2 - 1]**.
4. Through this method, we can utilise the total empty space available of the array at any instant of time.

**Time Complexity:** **O(1)** for push operation as it only takes constant time to push an element into the stack.

**O(1)** for pop operation as it only takes constant time to pop an element out of the stack.

**Space Complexity:** **O(N)**, where **N** is the size of the array used for the two stacks implementation. As we need to store stack values in the array and at some instant, the sizes of both the stacks can be **N**.

## **5. Evaluation of Postfix Expression** [<https://coding.ninja/P35>]

**Problem Statement:** Given a postfix expression, the task is to evaluate the expression. The answer could be very large output your answer modulo ( $10^9+7$ ). Also, use modular division when required.

An expression is called the postfix expression if the operator appears in the expression after the operands. The evaluation of the postfix expression is faster as compared to the infix notation as parenthesis are not required in postfix.

### **Example :**

Infix expression (where operators are used in-between operands): **A + B \* C - D**

Postfix expression (where the operator is written after the operands): **A B + C D - \***

### **Note:**

1. Operators will only include the basic arithmetic operators like **\***, **/**, **+**, and **-**.

2. The operands can be of multiple digits.
3. The operators and operands will have space as a separator between them.
4. There won't be any brackets in the postfix expression.
5. Do not print anything, just return the evaluated answer.

**Input Format:**

The **first line** contains an integer **T** denoting the number of test cases.

The next **T** lines represent the **T** test cases as follows

The **first and only line** of each test case contains a postfix expression.

**Output Format:**

For each test case, evaluate the postfix expression and return the value.

**Sample Input:**

2 3 1 \* + 9 -

**Sample Output:**

-4

**Explanation:**

```

2 3 1 * + 9 -
- : ( ) - ( )
9 : ( ) - (9)
+ : ( ( ) + ( ) ) - (9)
'*': ( ( ) + ( ( ) * ( ) ) ) - (9)
1 : ( ( ) + ( ( ) * (1) ) ) - (9)
3 : ( ( ) + ( (3) * (1) ) ) - (9)
2 : ( (2) + ( (3) * (1) ) ) - (9)

```

$$\text{Result} = (2 + 3) - 9 = 5 - 9 = -4$$

**Approach 1: Evaluation of Postfix Expression.**

The idea is to use **stack** to store the operands. Whenever an operator is encountered, we pop top two numbers from the stack, perform the operation and push the result back to the stack. Finally, when the traversal is completed, the number left in the stack is the final answer.

**Steps:**

1. Create a stack to store the operands.
2. Scan the given expression from left to right. For every scanned element do the following:
  - a. If the element is a number, extract the full number (for numbers with more than one digit). To do that, we increment the extracted digit and form a multi-digit

number till space is encountered. Space encountered would mean that the number has no more digits.

- b. If the element is an operand, pop the last two operands from the stack. Let the operands are **B** (topmost) and **A** (second from the top), and the operator is \*. Now, evaluate the operator—that is, perform **A \* B** (Second element is taken first, then the operator and finally the topmost element) and push the result into the stack.
- c. If space is encountered, send the control back to the loop using the *continue* statement.

3. When the expression ends, the number in the stack is the final answer.

#### Explanation:

Character Scanned	Stack	Explanation
2	2	2 is an operand; push it into the stack
3	2 3	3 is an operand; push it into the stack
1	2 3 1	1 is an operand; push it into the stack
*	2 3(3 * 1)	* is an operator, pop 3 & 1 and push the result into the stack
+	5(2 + 3)	+ is an operator, pop 2 & 3 and push the result into the stack
9	5 9	9 is an operand; push it into the stack
-	-4 (5 - 9)	- is an operator, pop 5 & 9 and push the result into the stack

**Time Complexity:**  $O(N)$ , where **N** is the length of the postfix expression.

**Space Complexity:**  $O(N)$ , where **N** is the length of the postfix expression. Stack space for **N** elements is used.

## 6. Queue Using Stack [<https://coding.ninja/P36>]

**Problem Statement:** Implement a queue data structure which follows **FIFO (First In First Out)** property using only the instances of the stack data structure.

Here the implementation means you need to complete the predefined functions which are supported by a normal queue such that it can efficiently handle the given input queries, which are defined below:

**The implemented queue must support the following operations of a normal queue:**

1. **enQueue(data):** This function should take one argument of type integer and place the integer to the back of the queue.
2. **deQueue():** This function should remove an integer from the front of the queue and also return that integer. If the queue is empty, it should return -1.
3. **peek():** This function returns the element present in the front of the queue. If the queue is empty, it should return -1.
4. **isEmpty():** This function should return true if the queue is empty and false otherwise.

**You will be given q queries of four types:**

- '1 val': Insert the integer **val** to the back of the queue.
- '2': Remove the element from the front of the queue and also return it.
- '3': Return the element present at the front of the queue (No need to remove it from the queue).
- '4': Return true if the queue is empty and false otherwise.

**Note:**

1. You can only use the instances of the stack data structure—that is, you can use the standard stack operations such as push to the top, pop the element from the top, etc.
2. You can also use the inbuilt stack data structure in some languages such as C++, Java.
3. You can assume that the maximum capacity of the queue may be infinite.

**Input Format:**

The **first line** contains an integer **q**, which denotes the number of queries that will be run against the implemented queue.

Then **q** lines follow.

The  $i^{\text{th}}$  line contains the  $i^{\text{th}}$  query in the format as in the problem statement.

For the **enQueue** operation, the input line will contain two integers separated by a single space, representing the type of the operation in integer and the integer data being pushed into the queue.

The input line will contain only one integer value for the rest of the queries, representing the query being performed.

**Output Format:**

For **Query 1**, you do not need to return anything.

For **Query 2**, print the integer being deQueued from the queue.

For **Query 3**, print the integer present in the front of the queue.

For **Query 4**, print “true” if the queue is empty, “false” otherwise.

The output of each query of type 2, 3 or 4 has to be printed in a separate line.

#### Sample Input:

```
1 1  
1 2  
1 3  
2  
2  
2  
3
```

#### Sample Output:

```
1 2 3 -1
```

#### Explanation of the Sample Input 1:

Here we have seven queries in total.

Query 1: Insert '1' to the back of the queue. Updated queue: 1

Query 2: Insert '2' to the back of the queue. Updated queue: 1 2

Query 3: Insert '3' to the back of the queue. Updated queue: 1 2 3

Query 4: Remove element from the front: Updated queue: 2 3

Query 5: Remove the element from the front: Updated queue: 2

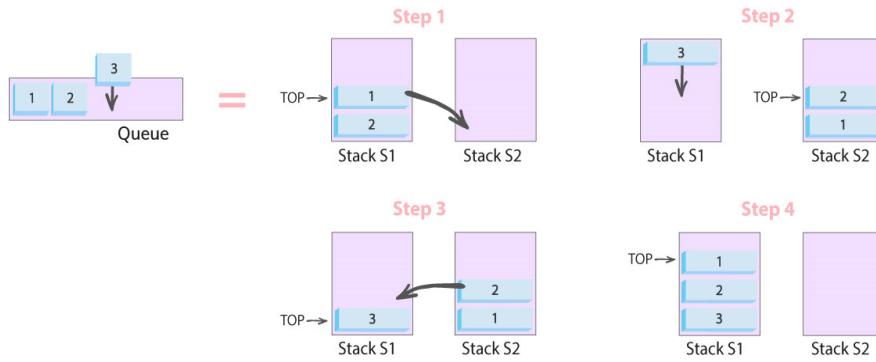
Query 6: Remove the element from the front: Updated queue:

Query 7: As the queue is empty, returned -1.

**Approach 1: Costly enQueue** - We can make the **enQueue** operation costly and perform the **deQueue** and all other operations in constant time.

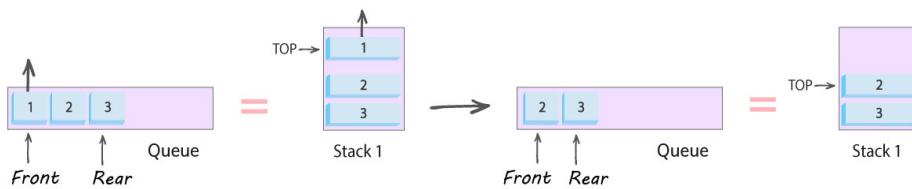
- We use two instances of the stack. Whenever we need to insert an element in the queue, we transfer all elements from stack 1 to stack 2, push the element in stack 1, and then again push all elements from stack 2 to stack 1.
- As the latest entered element is supposed to be at the back of the queue, this method ensures that this element is at the bottom of the stack.
- The above strategy ensures that the oldest entered element will always remain at the top of stack 1 so that to perform the **deQueue** or the peek operation, we simply return the top element of stack 1.

**Enqueue: O(n)**



*Pushing element “3” to the queue*

### Dequeue: O(1)



*Popping element “1” from the queue*

**Time Complexity:**  $O(N \cdot q)$ , where **N** is the maximum number of elements in the stack at any time and **q** is the total number of queries to be performed.

The time complexity of individual operations is:

**enQueue:**  $O(N)$ , where **N** is the number of elements in the stack.

**deQueue:**  $O(1)$

**peek:**  $O(1)$

**isEmpty:**  $O(1)$

**Space Complexity:**  $O(N)$ , where **N** is the maximum number of elements that are in the stack at a time.

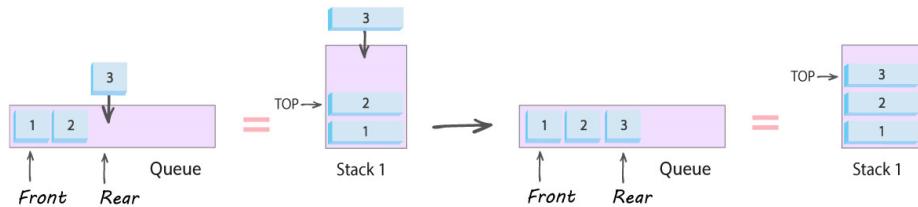
### Approach 2: Costly deQueue -

We can make the **deQueue** operation costly and perform the insertion in constant time.

- Use two stacks, let us say **s1** and **s2**. To insert into the queue, simply push the element at the top of **s1**.
- Now to perform the **deQueue** operation, we know that if elements are inserted in the order. Let us say that the order is: **a1, a2, a3**; then the removal will also be in the same order, i.e., **a1** then **a2** then **a3**. Thus, for the removal operation, we push all elements of **s1** to **s2** (to reverse their order of removal) and pop the top element of **s2**.

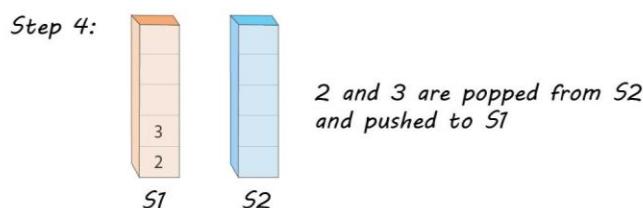
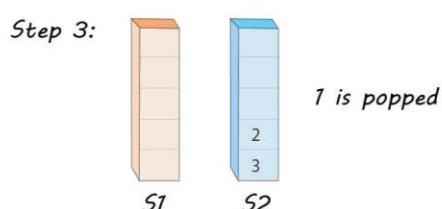
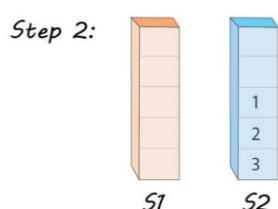
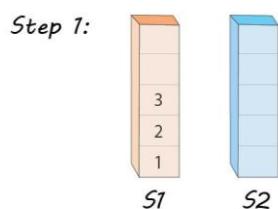
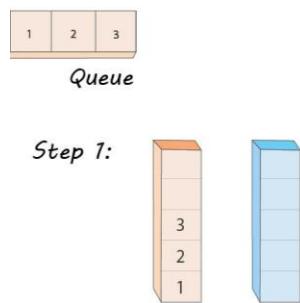
- The advantage of this approach over the first one is that we don't need to transfer all elements from **s1** to **s2** every time. As long as **s2** is not empty, we don't need to transfer the elements from **s1** to **s2**.

### Enqueue: O(1)



*Pushing element "3" to the queue*

### Dequeue: O(n)



**Time Complexity:**  $O(N*q)$ , where **N** is the maximum number of elements in the stack at any time and **q** is the total number of queries to be performed.

The time complexity of individual operations is:

**enQueue:**  $O(1)$

**deQueue:**  $O(N)$ , where **N** is the maximum number of elements in the stack at any time.

**peek**  $O(N)$ , where **N** is the maximum number of elements in the stack at any time.

**isEmpty:**  $O(1)$

**Space Complexity:**  $O(N)$ , where **N** is the maximum number of elements that are in the stack at a time.

**Approach 3: One Stack** - This approach is slightly better than the second approach because here we use only one user stack and one function-call stack which implements recursion. The logic behind this approach is very much similar to approach 2, except that we use only one user stack for the **deQueue** operation.

- Let us assume that the stack defined is **s1**. To **enQueue**, we simply push the element to the top of **s1**.
- To perform the **deQueue** operation, we recursively pop the element from the top of **s1** while the stack is not empty.
- When we reach the end of the stack, we return the element and then push all the other elements to the stack to restore them and return the last element in each recursion.
- The difference between **deQueue** and peek operations is that, in peek operation, we do not pop the last element of the stack but simply return it.
- This method is slightly better than approach 2 because, after each **deQueue** operation, all elements are restored in a single call.

**Time Complexity:**  $O(N*q)$ , where **N** is the maximum number of elements in the stack at any time and **q** is the total number of queries to be performed.

The time complexities of individual operations are:

**enQueue:**  $O(1)$

**deQueue:**  $O(N)$ , where **N** is the maximum number of elements in the stack at any time.

**peek:**  $O(N)$ , where **N** is the maximum number of elements in the stack at any time.

**isEmpty:**  $O(1)$

**Space Complexity:**  $O(N)$ , where **N** is the maximum number of elements that are in the stack at a time.

## **7. Stack Using Queue** [<https://coding.ninja/P37>]

**Problem Statement:** Implement a stack data structure specifically to store integer data using two **queues**.

You may use the inbuilt **queue**.

### Implement the following public functions :

1. **Constructor:** It initialises the data members (queues) as required.
2. **push(data):** This function should take one argument of type integer. It pushes the element into the stack and returns nothing.
3. **pop():** It pops the element from the top of the stack and, in turn, returns the element being popped or deleted. In case the stack is empty, it returns -1.
4. **top:** It returns the element being kept at the top of the stack. In case the stack is empty, it returns -1.
5. **size():** It returns the size of the stack at any given instance of time.
6. **isEmpty():** It returns a boolean value indicating whether the stack is empty or not.

### Operations Performed on the Stack:

- '1': Pushes integer data to the stack (**push** function).
- '2': Pops the data kept at the top of the stack and returns it to the caller (**pop** function).
- '3': Fetches and returns the data being kept at the top of the stack but doesn't remove it, unlike the pop function (**top** function).
- '4': Returns the current size of the stack (**size** function).
- '5': Returns a boolean value denoting whether the stack is empty or not (**isEmpty** function).

### Input Format:

The first line of the input contains an integer **q**, which denotes the number of queries to be run. The next **q** lines represent an operation that needs to be performed.

For the **push operation**, the input line will contain two integers separated by a single space, representing the type of the operation in integer and the integer data being pushed into the stack.

For the **rest of the operations** on the stack, the input line will contain only one integer value, representing the type of operation query being performed on the stack.

### Output Format:

For query **1**, you do not need to return anything.

For query **2**, print the data being popped from the stack.

For query **3**, print the data kept on the top of the stack.

For query **4**, print the current size of the stack.

For query **5**, print 'true' or 'false' (without quotes).

Output for every query will be printed in a separate line.

### Sample Input:

```
6
1 13
1 47
4
5
2
```

3

#### **Sample Output:**

2  
false  
47  
13

#### **Explanation:**

Here we have six queries in total.

Query 1: Integer 1 represents the push function. Hence we push element '13' onto the stack.  
Query 2: Integer 1 represents the push function. Hence we push element '47' onto the stack.  
Query 3: Integer 4 represents the size function. Hence we print the size of the stack that is 2.  
Query 4: Integer 5 represents the isEmpty function. Hence here, we print false because the stack is not empty.  
Query 5: Integer 2 represents the pop function. The stack contains the element '47' at the top.  
We remove/pop '47' from the stack and print '47' on the new line.  
Query 6: Integer 3 represents the top function. Because we have element '13' at the top of the stack, we print '13' on the new line.

#### **Approach 1: Making push operation costly**

This method ensures that every new element entered in queue **q1** is always at the front. Hence, during pop operation, we just dequeue from **q1**. For this, we need another queue, **q2**, which is used to keep every new element to the front of **q1**.

#### **Steps:**

1. During push operation:
  - a. Enqueue new element **x** to queue **q2**.
  - b. One by one, dequeue everything from **q1** and enqueue to **q2**.
  - c. Swap the names of **q1** and **q2**.
2. During pop operation:
  - a. Dequeue an element from **q1** and return it.
3. For the size function, return the size of the queue **q1** and for the empty function, check if **q1** is empty.

**Time Complexity:**  $O(N * q)$ , where **N** denotes the maximum number of elements in the queue, and **q** denotes the number of queries. During each push operation, we transfer all the elements of **q1** to **q2**.

**Space Complexity:**  $O(\max(n1, n2))$ , we use two queues of size **n1** and **n2**.

#### **Approach 2: Making pop operation costly**

In this method, we will be using two queues, **q1** and **q2**. The difference in this method is that in a push operation, the new element is always enqueued to **q1**.

#### **Steps:**

1. During push operation:
  - a. Enqueue new element **x** to **q1**.
2. During pop operation:
  - a. One by one, dequeue everything except the last element from **q1** and enqueue to **q2**.
  - b. Dequeue the last item of **q1**, and the dequeued item is the result, store it.
  - c. Swap the names of **q1** and **q2**.
  - d. Return the item stored in step 2.
3. For the size function, return the size of the queue **q1** and for the empty function, check if **q1** is empty.

**Time Complexity:**  $O(N * q)$ , where **N** denotes the maximum number of elements in the queue, and **q** denotes the number of queries. During each pop operation, we transfer all the elements of **q1** to **q2**.

**Space Complexity:**  $O(\max(n1, n2))$ , we use two queues of size **n1** and **n2**.

#### **Approach 3: Using a single queue**

In this method, we use a single queue **q1**. In a push operation, we can calculate the size of the queue **q1**. Hence, we can enqueue new data to the queue. Now suppose before inserting new data, the queue size is **x**, then we dequeue **x** elements from the queue and push them back again into the same queue. This would push the new element to the front of the queue.

#### **Steps:**

1. During pop and top operation:
  - a. The element we are searching for is the front of the queue.
  - b. Hence during pop operation, simply access the front of the queue and remove the element.
  - c. During top operation, simply access the front element of the queue.
2. Hence assuming we know the queue size at any instance, we can solve this problem using a single queue.

**Time Complexity:**  $O(N * q)$ , where **N** denotes the maximum number of elements in the queue, and **q** denotes the number of queries. During each push operation, we re-transfer all the elements of **q1** back to **q1**.

**Space Complexity:**  $O(N)$ , as we use a queue of size **N**.

## **8. Reverse First K Elements of Queue** [<https://coding.ninja/P38>]

**Problem Statement:** You are given a **queue** containing **N** integers and an integer **K**. You need to reverse the order of the first **K** elements of the queue, leaving the other elements in the same relative order.

You can only use the standard operations of the QUEUE STL:

1. **enqueue(x)**: Adds an item **x** to rear of the queue
2. **dequeue()**: Removes an item from the front of the queue
3. **size()**: Returns the number of elements in the queue.
4. **front()**: Finds the front element.

#### For Example:

Let the given **queue** be {1, 2, 3, 4, 5} and **K** be 3. You need to reverse the first **K** integers of the **queue**, which are 1, 2, and 3. Thus, the final output will be {3, 2, 1, 4, 5}.

#### Input Format:

The **first line** contains an integer **T** denoting the number of queries or test cases.

Next **T** lines follow-

The **first line** of each input consists of two space-separated integers **N** and **K**, denoting the queue size and number of elements to be reversed from the front.

The **second line** of each input consists of **N** space-separated integers denoting the elements of the queue.

#### Output Format:

For each test case, print the queue elements after reversing the order of first **K** elements in a separate line.

#### Sample Input:

```
4 2
6 2 4 1
```

#### Sample Output:

```
2 6 4 1
```

#### Explanation:

The queue after reversing the first 2 elements—that is, 6 and 2 will be {2, 6, 4, 1}.

#### Approach 1: Using Array

The very first approach can be to use an array to reverse the elements.

#### Steps:

1. Initialize **N** = queue.size
2. Create an array **ARR** of size equal to the size of the queue.
3. Run a loop from  $i = 0$  till  $i = N$ , **pop the queue elements, and push** to the array.

4. Reverse the first K elements of the array.
5. Run a loop from  $i = 0$  till  $\mathbf{N}$  and push the array elements to the queue.
6. Return queue.

**Time Complexity:**  $O(N)$  per test case, where  $\mathbf{N}$  is the queue size. In the worst case, we are pushing and popping all the elements of the queue.

**Space Complexity:**  $O(N)$  per test case, where  $\mathbf{N}$  is the queue size. In the worst case, we are creating an array of size  $\mathbf{N}$ .

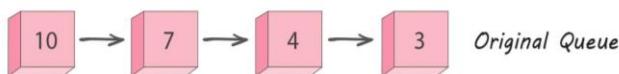
### Approach 2: Using Stack

Another approach can be to use a stack to reverse the elements. We will store the first  $\mathbf{K}$  elements of the queue into a stack. Then, we will push all elements of the stack into the queue. Finally, we pop and push  $\mathbf{N} - \mathbf{K}$  elements continuously in the queue.

#### Steps:

1. Initialise an array of size equal to the size of the queue
2. Create a stack S
3. Run a loop till  $K$  and push the front of the queue to the stack, and pop the element from the queue
4. Run a while loop till the stack is not empty. We push the stack's top into the queue and pop top from stack S in this loop.
5. Run a loop till  $\mathbf{N} - \mathbf{K}$  queue size to push and pop  $\mathbf{N} - \mathbf{K}$  times.
6. Finally, return the queue.

Let us take an example where the queue is **{10, 7, 4, 3}** and  $\mathbf{K} = 2$ .



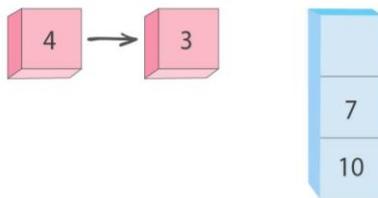
**We will create an empty stack {} and push K elements (in this case,  $K = 2$ ) from the front of the queue to the stack.**

Step 1: Pop 10 from the queue and push in the stack.

- a. Stack: {} 10 {}.
- b. Queue: {} 7, 4, 3 {}

Step 2: Pop 7 from the queue and push in the stack.

- c. Stack: {} 10, 7 {}.
- d. Queue: {} 4, 3 {}



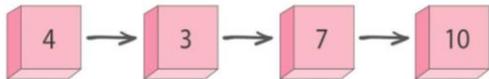
**Now, we pop elements from the stack and push them in the queue.**

Step 3: Pop 7 from the stack and push in the queue.

- a. Stack: { 10 }
- b. Queue: { 4, 3, 7 }

Step 4: Pop 10 from the stack and push in the queue.

- c. Stack: { }
- d. Queue: { 4, 3, 7, 10 }



**Now pop (N - K) elements from the queue and push them back in the same order**

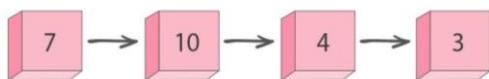
Step 5: Push and pop 4 in the queue.

- a. Queue: { 3, 7, 10, 4 }

Step 6: Push and pop 3 in the queue.

- b. Queue: { 7, 10, 4, 3 }

**Thus, the final output will be {7, 10, 4, 3}.**



**Time Complexity:**  $O(N)$  per test case, where  $N$  is the size of the queue. In the worst case, we are pushing and popping all the elements of the queue.

**Space Complexity:**  $O(K)$  per test case, where  $K$  is the number of elements whose order is reversed. In the worst case, we are creating a stack of size  $K$ .

## **9. Reversing a Queue**

[<https://coding.ninja/P39>]

**Problem Statement:** You are given a queue of  $N$  elements. Your task is to reverse the order of elements present in the queue.

**You can only use the standard operations of the QUEUE STL.**

1. **enqueue (x):** Add an item x to the rear of the queue.
2. **dequeue():** Removes an item from the front of the queue.
3. **size():** Returns the number of elements in the queue.
4. **front():** Returns front element.
5. **empty():** Checks whether the queue is empty or not.

#### **Input Format:**

The **first line** of the input contains an integer **T** denoting the number of test cases.

Next **T** lines follow-

The **first line** of each test case contains an integer **N** denoting the number of elements present in the queue.

The **second line** contains '**N**' space-separated integers denoting the elements present in the queue.

#### **Output Format:**

For each test case, print space-separated integers denoting the elements of the queue in the reverse order.

#### **Sample Input:**

```
5
10 6 8 12 3
```

#### **Sample Output:**

```
3 12 8 6 10
```

#### **Explanation:**

The queue has elements in the order: 10, 6, 8, 12, 3. Reversing the queue changes the order of elements to 3, 12, 8, 6, 10.

#### **Approach 1: Using Another Queue**

The idea is to use another queue to solve this problem. This queue will be used to store the final result.

#### **Steps:**

- Create an empty queue, say **result**.
- Remove the front element from the **given** queue.
- Each time you remove an element, push it back to the same queue.
- After repeating the previous two steps, **N - 1** times (where **N** is the queue size), the last element of the (given) queue will reach the front of the queue.
- Dequeue the front element and enqueue it to **result**.
- The size of the given queue has now decreased by one, and the size of the **result** queue has increased by one.
- Repeat the steps until the given queue becomes empty.

- The **result** queue holds the final answer.

**Time Complexity:**  $O(N^2)$  per test case. In the worst case, we require  $O(N)$  operations for each element in the queue. Therefore, to reverse  $N$  elements, the overall time complexity becomes  $O(N^2)$ .

**Space Complexity:**  $O(N)$  per test case. In the worst case, extra space is required by the queue.

### Approach 2: Using an Array

The queue data structure follows **FIFO (First In First Out)** order. So, we cannot reverse the queue in-place. We require the help of another data structure to reverse the queue. One of the simplest data structures which we can use is an array.

#### Steps:

- Create an array of size equal to the number of elements in the queue.
- The idea is to fill the array from the back instead of the front.
- One by one, dequeue the elements from the queue and add them to the array.
- Now, iterate over the array and push each element back to the queue.

**Time Complexity:**  $O(N)$  per test case. In the worst case, we require  $N$  enqueue and  $N$  dequeue operations. Thereby, the overall time complexity is  $O(N)$ .

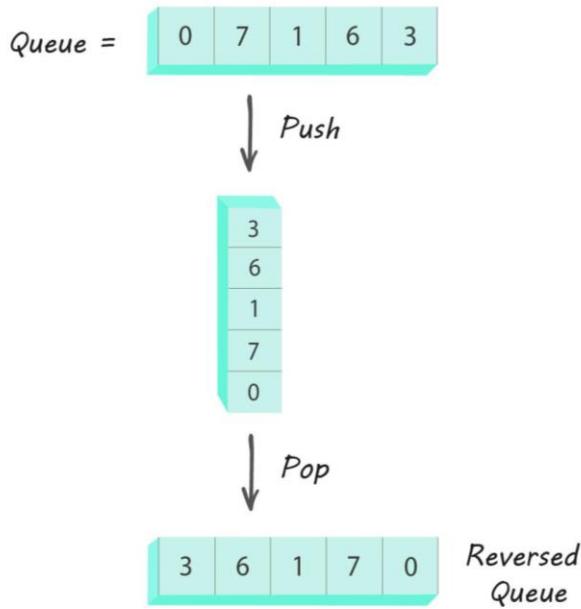
**Space Complexity:**  $O(N)$  per test case. In the worst case, the array is required to store the elements of the queue in the worst case.

### Approach 3: Using Stack

Stack data structure follows a **LIFO (Last In First Out)** order—that is, the element which was inserted last in the stack is the first one to pop out. Hence, a stack can be used to reverse the order of elements.

#### Steps:

- Empty all the elements of the queue into a stack.
- Pop the elements from the stack and insert them back into the queue.
- The queue now stores the elements in reverse order.



**Time Complexity:**  $O(N)$  per test case. In the worst case, we require  $N$  push,  $N$  pop,  $N$  enqueue and  $N$  dequeue operations. Thereby, the overall time complexity is  $O(N)$ .

**Space Complexity:**  $O(N)$  per test case. In the worst case, extra space is required by the stack.

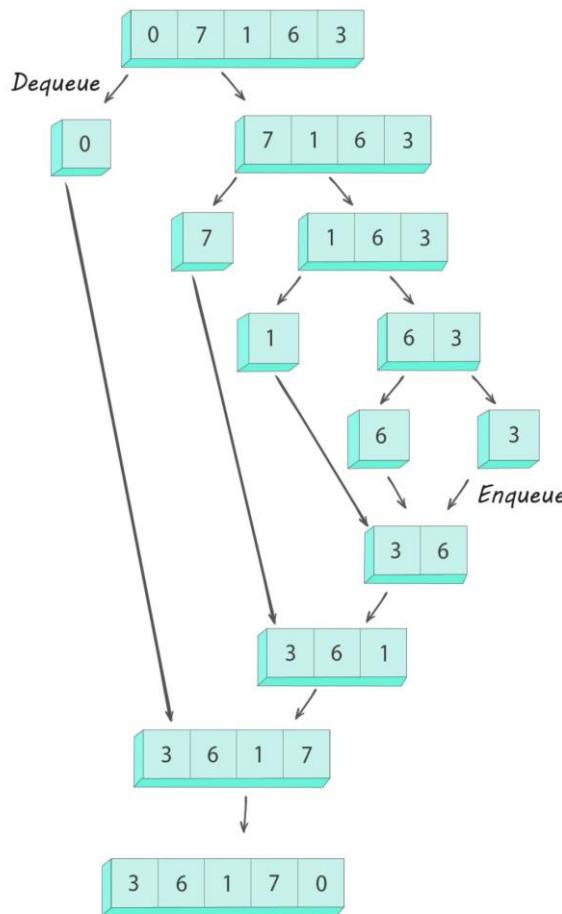
### Approach 3: Using Recursion

Here, instead of explicitly creating a stack, we use the concept of recursion where the OS has to maintain the stack.

The idea is to dequeue the front element from the queue and recursively call the reverse function for the remaining queue. In this way, we are dividing the larger problem into smaller sub-problems. This is repeated until the queue becomes empty. A base condition can be easily formed around it. Now, after returning from the recursive call, push the elements back into the queue. In this way, we have reversed the order of elements in the queue.

#### Steps:

1. Initialise the base case—that is, if the queue **is empty**, return the queue.
2. Dequeue the front element and store it in a variable, say **E**.
3. Recursively call the reverse function.
4. Enqueue **E** back to the queue.
5. Return the queue.



**Time Complexity:**  $O(N)$  per test case. In the worst case, we perform  $N$  enqueue and  $N$  dequeue operations. Therefore, the overall time complexity is  $O(N)$ .

**Space Complexity:**  $O(N)$  per test case. In the worst case, extra space is required by the recursion stack.

## 10. LRU Cache Implementation [\[https://coding.ninja/P40\]](https://coding.ninja/P40)

**Problem Statement:** Design and implement a data structure for Least Recently Used (LRU) cache to support the following operations:

**get(key):** Return the value of the key if the key exists in the cache, otherwise return -1.

**put(key, value):** Insert the value in the cache if the key is not already present or update the value of the given key if the key is already present. When the cache reaches its capacity, it should invalidate the least recently used item before inserting the new item.

**Two types of queries denote these operations:**

**Type 1:** for **get(key)** operation.

**Type 2:** for **put(key, value)** operation.

The cache is initialised with a capacity (the maximum number of unique keys it can hold at a time).

**Note:**

Access to an item or key is defined as a get or a put operation on the key. The least recently used key is the one with the oldest access time.

**Input Format:**

The **first line** of input contains two space-separated integers **C** and **Q**, denoting the capacity of the cache and the number of operations to be performed, respectively.

The next **Q** lines contain operations, one per line. Each operation starts with an integer which represents the type of operation.

If it is **0**, then it is of the first type and is followed by one integer key.

If it is **1**, it is of the second type and is followed by two space-separated integers, key and value(in this order).

The meanings of key and value are well explained in the statement.

**Output Format:**

For each operation of type 0, print an integer on a single line the value of the key if the key exists, otherwise -1.

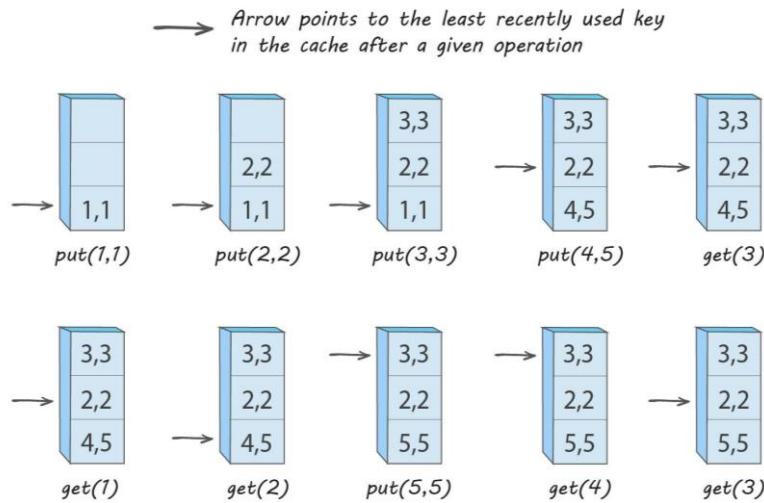
**Sample Input:**

```
3 10
1 1 1
1 2 2
1 3 3
1 4 5
0 3
0 1
0 2
1 5 5
0 4
0 3
```

**Sample Output:**

```
3
-1
5
-1
3
3
```

**Explanation:**



Initializing a cache of capacity 3

Then each operation is performed as shown in the above figure.

```

cache.put(1,1)
cache.put(2,2)
cache.put(3,3)
cache.put(4,5)
cache.get(3)    // returns 3
cache.get(1)    // returns -1
cache.get(2)    // returns 2
cache.put(5,5)
cache.get(4)    // returns -1
cache.get(3)    // returns 3

```

### Approach 1: Using Arrays

We will use an array of type **Pair<key, value>** to implement our LRU Cache where the larger the index is, the more recently the key is used. This means that the 0th index denotes the least recently used pair, and the last index denotes the most recently used pair.

The key will be considered as accessed if we try to perform any operation on it. So while performing the get operation on a key, we will do a linear search. If the key is found in the cache at an index **id**, we will left shift all keys starting from **id + 1** in the cache by one and put the key at the end (marking it as the most recently used key in the cache).

Similarly, while performing the put operation on a key, we will do a linear search. If the key is found at index **id**, we will shift left all keys starting from **id + 1** in the cache by one and put the key at the end. Otherwise, we will check the current size of the cache. If the size equals capacity,

we will remove the first (0th) key from the cache by doing the left shift on all the keys by one. Now we can simply insert the key at the end.

**size:** denotes the current size of the cache.

**capacity:** denotes the maximum keys cache can hold at one time.

**cache:** denotes the array of type pair to store key-value pairs.

**Pair:** Pair type will store these values, key, value.

### **Steps for each method:**

**getIndex(int key):** This method will return the index of the key if it exists in the cache, otherwise -1.

1. For all the pairs in the cache, from  $i = 0$  to  $\text{size} - 1$ .
  - a. If  $\text{currentPair} \rightarrow \text{key} == \text{key}$ , return  $i$ .
2. Return -1, as the given key does not exist in the cache.

**leftShift(int start):** This method will shift left all the pairs starting from the **start** index by 1.

1. If  $\text{start} == \text{size}$ , no pair exists to shift, return.
2. For all pairs in the cache, from  $i = \text{start}$  to  $\text{size} - 1$ ,
  - a.  $\text{cache}[i - 1] = \text{cache}[i]$

### **int get(key):**

1. **ID** =  $\text{getIndex}(\text{key})$ .
2. If **ID** == -1, then the key does not exist in the cache, return -1.
3. Get the pair at index = **ID**, name it **pair**.
4. Do  $\text{leftShift}(\text{ID} + 1)$ .
5. Assign  $\text{cache}[\text{size} - 1] = \text{pair}$ .
6. Return **pair->value**.

### **void put(key, value):**

1. **ID** =  $\text{getIndex}(\text{key})$ .
2. If **ID** is not equal to -1, the key already exists in the cache. In this case:
  - a. Get the pair at index = **ID**, name it **pair**.
  - b. Do  $\text{leftShift}(\text{ID} + 1)$ .
  - c. Assign **pair->value** = value.
  - d. Assign  $\text{cache}[\text{size} - 1] = \text{pair}$
  - e. Return from the function.
3. Else, create a new pair of (key, value), name it **pair**.
4. If the size is less than the capacity,
  - a. Insert **pair** in cache at index = size, using  $\text{cache}[\text{size}] = \text{pair}$
  - b. Increment size by one.
5. Otherwise,

- a. Remove the first pair from the cache by performing a left shift
- b. Insert pair in cache at index = size - 1, using cache[size - 1] = pair.

#### **Time Complexity:**

**get(key):** O(capacity),

**put(key, value):** O(capacity),

We will be iterating on the cache to left shift all pairs by one in the worst case.

**Space Complexity: O(capacity)**, where capacity is the maximum number of keys LRU Cache can store. In the worst case, we will only be maintaining the 'capacity' number of keys in storage.

#### **Approach 2: HashMap with Queue**

We will use two data structures to implement our LRU Cache.

1. **Queue<Node>:** To store the nodes into cache where the least recently used key will be the head node and the most recently used key will be the tail node.
2. **Map<K, Node>:** To map the keys to the corresponding nodes.

The key will be considered as accessed if we try to perform any operation on it. So while performing the get operation on a key, if the key already exists in the cache, we will detach the key from the cache and put it at the tail end (marking it as the most recently used key in the cache). Similarly, while performing the put operation on a key, we will detach it from the cache and put it at the tail end if the key already exists. Otherwise, we will put the key at the tail end and check the current size of the cache. If the size exceeds capacity, we will remove the head key from the cache.

To perform both of these operations at the minimum cost (O(1)). We will use a **doubly-linked list** to implement our queue.

**head:** head node of the queue, to store least recently used key-value pair.

**tail:** tail node of the queue to store the most recently used key-value pair.

**size:** denotes the current size of the cache.

**capacity:** denotes the maximum keys cache can hold at one time.

**Node:** Node type will store these values: next, prev, key, and value.

#### **Algorithm:**

##### **addToRear(Node node):**

1. If head == null, head = **node**.
2. Else tail→next = **node** and **node**→prev = tail
3. tail = **node**

##### **deleteNode(Node node):**

1. If the given **node** is the head node, **head** = **head**→next.
2. Else **node**→prev→next = **node**→next
3. If the given node is the tail node, **tail** = **tail**.prev
4. Else **node**→next→prev = **node**.prev
5. **node**→next = null
6. **node**→prev = null

**int get(key):**

1. If the key does not exist, return -1.
2. Fetch the node corresponding to the given key from the map, let it be **nod**.
3. **deleteNode(nod)**
4. **addToRear(nod)**
5. Return **nod** →value

**void put(key, value):**

1. If the key already exists in the cache, remove the key from the cache and put it at the tail end with the new value.
  - a. Fetch the node corresponding to the given key from the map, let it be **nod**.
  - b. **nod**→value = value
  - c. **deleteNode(nod)**
  - d. **addToRear(nod)**
  - e. return
2. Initialize a new node with the given key-value pair. Let it be **nod**, **nod** = **node(key,value)**.
  - a. **map.put(key, nod)**
  - b. If the cache size reaches its capacity
    - i. **map.remove(key)**.
    - ii. **deleteNode(nod)**
    - iii. **addToRear(nod)**
  - c. Else,
    - i. **addToRear(nod)**
    - ii. Increment size by 1

**Time Complexity:**

**get(key): O(1)**

**put(key, value): O(1)**

As we are only updating the pointers of a queue implemented using a doubly-linked list.

**Space Complexity: O(capacity):** where capacity is the maximum number of keys LRU Cache can store. In the worst case, we will only be maintaining the capacity number of keys in storage.

# 5. Linked Lists – I

---

## Introduction to Linked Lists

A linked list is a collection of nodes in **non-contiguous** memory locations where every node contains data and a pointer to the next node of the same data type. In other words, the node stores the address of the next node in the sequence.

### Important Terms:

1. **Node:** In a singly linked list, a node contains two fields:
  - a. **Data** field stores the data at the node
  - b. **Next pointer** contains the address of the next node in the sequence
2. **Head:** The address of the first node in a linked list is referred to as the head. The head is always used as a reference to traverse the list.
3. **Tail:** The address of the last node in a linked list is referred to as the tail. In the case of a singly linked list, the tail pointer always contains a pointer to NULL, denoting the end of a linked list.

## Properties of Linked Lists

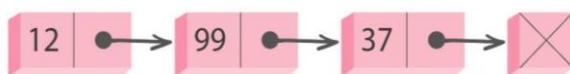
- A linked list is a **dynamic** data structure, which means the list can grow or shrink easily as the nodes are stored in memory in a non-contiguous fashion.
- The size of a linked list is limited to memory size, and the size need not be declared in advance.

**Note:** We must never lose the address of the **head** node as it references the starting address of the linked list, and if lost, we would lose the entire list.

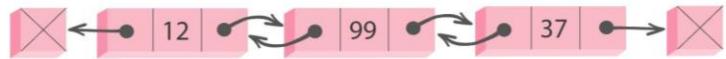
## Types of Linked Lists

There are generally three types of linked list:

- **Singly Linked List:** Each node contains only one pointer, which points to the subsequent node in the list.



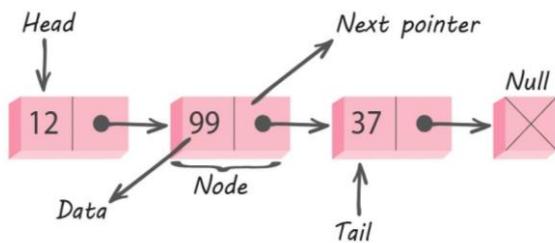
- **Doubly Linked List:** It's a two-way linked list as each node points to the next pointer and the previous pointer. The previous pointer of the head node points to NULL because there is no node before it and the next pointer of the tail node also points to NULL because there is no node after it.



- **Circular Linked List:** There is no tail node in this list as the pointer inside the last node points to the head node. It means that there is no point that points to NULL in these lists.



In this chapter, we will work with **singly linked lists**. A singly linked list allows the traversal of data only in one direction. We will discuss different types of linked lists briefly here and work with them in the upcoming chapter.



## Singly Linked Lists

Let us discuss the following operations on singly linked lists one by one:

1. Insertion
2. Deletion
3. Search
4. Display

### Insertion Operation

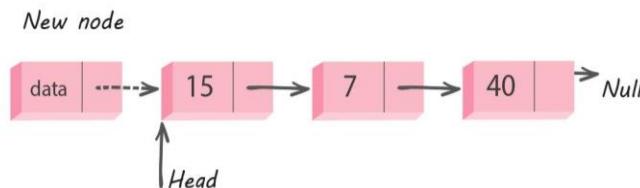
There are three possible cases:

- Inserting a new node at the beginning
- Inserting a new node at the end of the list
- Inserting a new node in the middle at a specified location

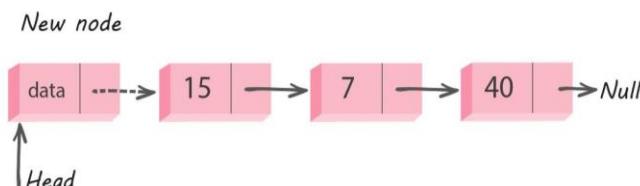
### Case 1: Insert node at the beginning of the list:

In this case, a new node is inserted before the current head node. In such a case, only one next pointer needs to be modified (new node's next pointer), which can be done in the following two steps:

**Step 1:** Create a **new node**. Update the **next** pointer of the **new node** to point to the current **head**.



**Step 2:** Update **head** pointer to point to the **new node**.



### PseudoCode:

```
function insertAtBeginning(x)
    /*
        create a new node named 'newNode'
        set data of 'newNode' to x
    */

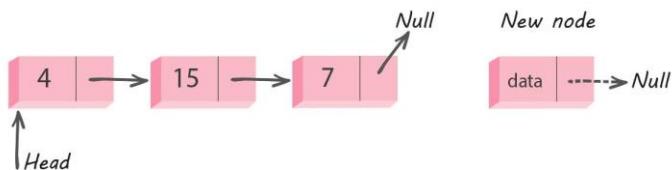
    newNode.data = x
    newNode.next = null

    /*
        If the list is not empty, make the next pointer of 'newNode' point to the head of
        existing list, thereby converting first node in the existing list to the second node
        in the list
    */
    else
        newNode.next = head
        head = newNode
    return head
```

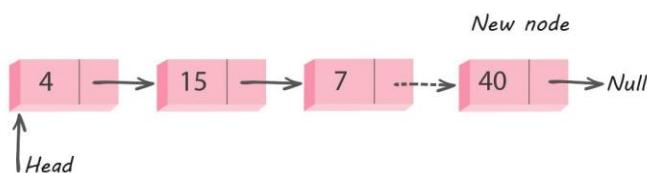
## Case 2. Insert node at the end

In this case, we need to modify one next pointer—the last node's next pointer.

**Step 1:** New node's **next** pointer points to **null**.



**Step 2:** Last node's **next** pointer points to the **new node**.



**Pseudo Code:**

```
function insertAtEnd(x)

    /*
        create a new node named 'newNode'
        set data of 'newNode' to x
    */

    newNode.data = x
    newNode.next = null

    // If the list is empty, set head as 'newNode.'

    if head is null
        head = newNode
        return head

    /*
        Otherwise, create a 'cur' node pointer and keep moving it
        until it reaches the last node to reach the end of list
    */

    cur = head // get the start of list in another pointer to not update and hence lose the
    head of linked list
    while cur.next is not null
        cur = cur.next

    /*
        Now 'cur' points to the last node of the linked list; link the 'newNode' and the
    
```

```
'cur' node - 'newNode' is the next node of 'cur' node.  
*/
```

```
cur.next = newNode  
return head
```

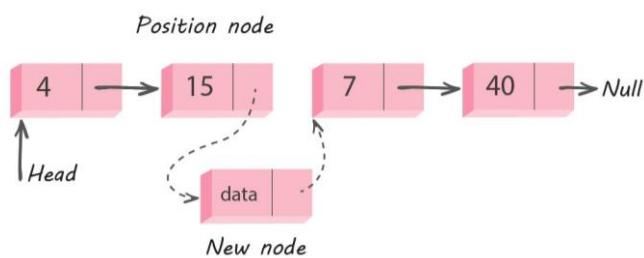
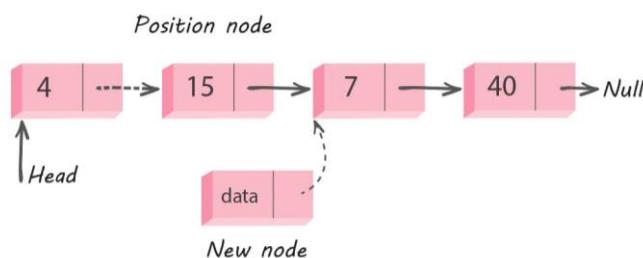
### Case 3: Insert node anywhere in the middle at a specified index

If the new node's location is given and is at the non-ending nodes, we need to update the address of two next pointers.

Let us assume that we need to add a node between the existing second and third nodes; we take the following steps:

**Step 1:** We traverse the first two nodes and stop at the second node to insert the **new node**.

**Step 2:** For simplicity, let us assume that the second node is called the **position node**. The **new node** points to the next node of the position where we want to add this node.



- The position node's **next** pointer now points to the new node.

### Pseudo Code:

```

function insertAtIndex(idx, x)

    /*
        create a new node named 'newNode'
        set newNode's data to data
    */

    newNode.data = x
    newNode.next = null

    if idx is 0
        // case of insertion at beginning
        insertAtBeginning(x)
    else
        /*
            Have a pointer 'position_node' to reach the previous node at the given 'index', thereby
            making 'position_node' point to the node after which the 'newNode' is to be inserted.
        */

        count = 0
        position_node = head

        while count < idx - 1 and position_node.next is not null
            count = count + 1
            position_node = position_node.next

        /*
            If count does not reach (idx - 1), then the given index
            is greater than the current size of the list
        */

        If count < idx - 1
            print "invalid index"
            return head

        /*
            Link 'newNode' with the second part of the linked list by setting the
            next pointer of 'newNode' to the
            address of the node present at position idx
        */

        newNode.next = position_node.next

        /*
            Link 'newNode' with the first part of the linked list by updating the
            'position_node' node's next pointer to the address of
            the 'newNode'
        */

        position_node.next = newNode

```

```
return head
```

## Deletion Operation

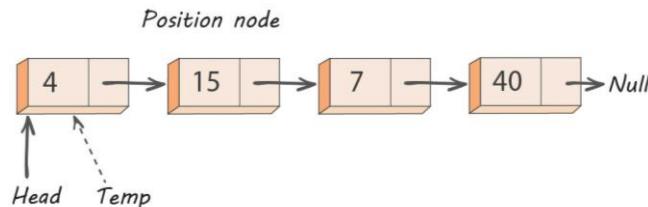
Similar to insertion, we have three cases for the deletion operation. They are as follows:

- Deleting the first node
- Deleting the last node
- Deleting an intermediate node.

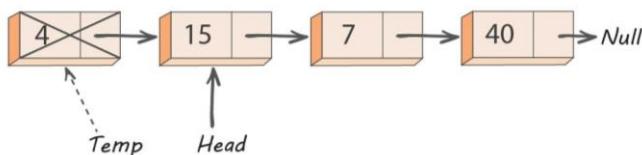
### Case 1: Deleting the first node

It can be done in two steps:

**Step 1:** Create a **temporary** node which will point to the same node that the **head** points to.



**Step 2:** Move the **head** node's pointer to the **next** node and delete the temporary node.



### Pseudo Code:

```
function deleteFromBeginning()  
    // If the list is empty, return null (there is no node to be deleted)  
    if head is null  
        print "Linked List is empty"  
        return null  
  
    /*  
     * Otherwise, set the new head to the second node in the linked list and delete  
     */
```

the first node from the linked list.

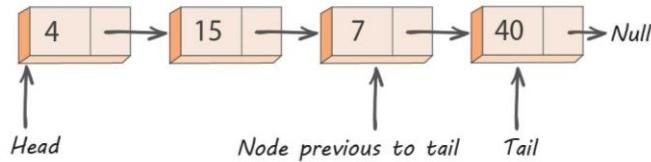
\*/

```
temp = head  
head = head.next  
delete temp  
return head
```

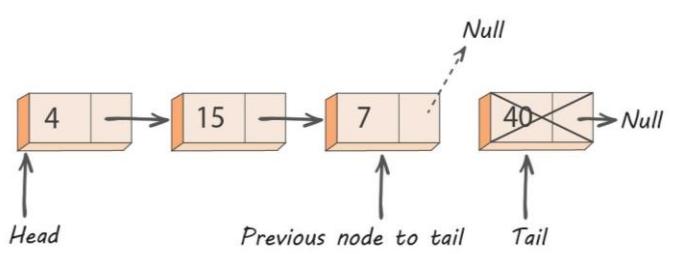
## Case 2: Deleting the Last node

In this case, the last node is removed from the list. This operation is a bit trickier than removing the first node because the algorithm should find a node, which is previous to the tail. It can be done in three steps:

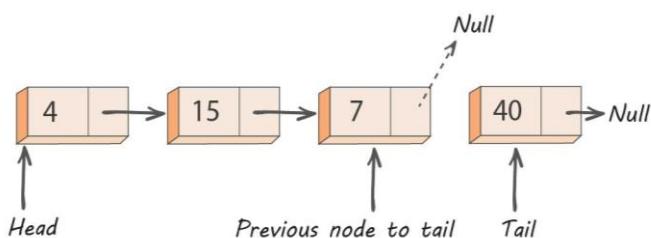
**Step 1:** Traverse the list and,, while traversing maintain the previous node address as well. By the time we reach the end of the list, we will have two pointers, one pointing to the **tail** node and the other pointing to the node **before** the tail node.



**Step 2:** Update the second last node's next pointer as **null**.



**Step 3:** Delete the tail node.



### Pseudo Code:

```
function deleteFromEnd()

    // If the list is empty, return null (there is no node to be deleted)
    If head is null
        print "Linked List is empty"
        return null

    // If the list contains a single node, delete it

    If head.next is null
        temp = head
        head = head.next
        delete temp
        return head

    /*
        Use two pointers - 'tail' and 'cur'. 'prev' always represents the previous node of
        the current node. Make 'tail' point to the last node of the linked list and 'prev'
        point to the second last node.*/
    
    tail = head.next
    prev = head
    while tail.next is not null
        prev = tail
        tail = tail.next

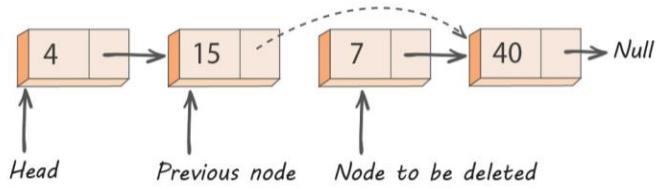
    /*
        Make the second last node pointed by 'prev' the new last node in the linked
        list and delete the old last node pointed by 'tail'.
    */

    prev.next = null
    delete tail
    return head
```

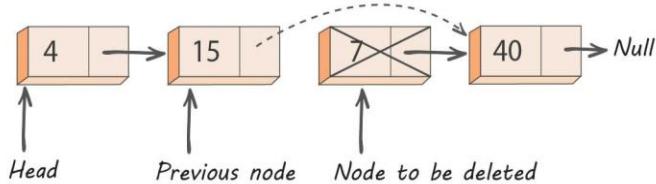
### Case 3: Deleting an Intermediate Node

In this case, the node to be removed is always located between two nodes. Head and tail links are not updated in this case. Such removal can be done in two steps:

**Step 1:** Similar to Case 2 discussed above, maintain the previous node while traversing the list. Once we find the node to be deleted, change the previous node's next pointer to the next pointer of the node to be deleted.



**Step 2:** Delete the current node.



**Pseudo Code:**

```

function deleteFromIndex(idx)

    // If the list is empty, return null (there is no node to be deleted)
    If head is null
        print "Linked List is empty"
        return null

    if idx is 0
        // Case of deletion from beginning
        return deleteFromBeginning()
    else
        count = 0
        previous = head

    /*
        Have a pointer 'previous' to reach the previous node at the given
        'index', thereby making 'previous' point to the node whose next node is
        to be deleted.
    */

    while count < idx - 1 and previous is not null
        count = count + 1
        previous = previous.next

    /*
        If the 'previous' node reaches the tail of the list, then the given index is
        greater than the size of the list.
    */

    if previous is null or previous.next is null

```

```

print "invalid index"
return head


$$/*$$

    Otherwise, link the 'previous' node (previous node of the node to be deleted)
    with the next node of the node to be deleted.

$$*/$$


temp = previous.next
previous.next = temp.next
delete temp

return head

```

## Search Operation

This operation involves searching a linked list for a node with given data. To search any value in the linked list, we can traverse the linked list and compare the value present in each node till we find the given value or reach the end of the list.

```

function search(x)

    // Create a 'cur' node pointer initialized to head of the linked list
    cur = head

    
$$/*$$

        Traverse the linked list, and compare the data of 'cur' node and the data to be
        searched
    
$$*/$$


    while cur is not null
        if cur.data equals data
            return true
        cur = cur.next

    // If data is not found, return false
    return false

```

## Display Operation

This operation involves displaying the data content of each node in a given linked list.

```

function display()
    // If the list is empty, print nothing
    if the head is null
        print "list empty"
    return

```

```

// Create a 'cur' node pointer initialized to head of the linked list
cur = head

/*
    Traverse the linked list and print the data of the 'cur' node
*/

while cur is not null
    print cur.data
    cur = cur.next

```

## Time Complexity of Various Operations

Let **n** be the number of nodes in a singly linked list. The time complexity of linked list operations in the worst case can be given as:

Operations	Time Complexity
Insertion at beginning(x)	O(1)
Insertion at end(x)	O(n)
Insertion at given index(idx,data)	O(n)
Deletion from beginning()	O(1)
Deletion from end()	O(n)
Deletion from given index(idx)	O(n)
Search(x)	O(n)
Display	O(n)

## Applications of Linked Lists

Following are the few applications of linked lists:

- Linked Lists can be used to implement useful data structures like stacks and queues.
- They can be used to implement hash tables; each hash table bucket can be a linked list.
- They can be used to implement graphs (Adjacency List representation of graphs).
- Linked Lists can be utilised in a refined form to implement connectivity in an otherwise large file system.

## Advantages of Linked Lists

- It is a dynamic data structure, which can grow or shrink according to the requirements.
- Insertion and deletion of elements (or nodes) can be done easily and does not require movement of all elements (or nodes), unlike arrays.
- Allocation and deallocation of memory can be done easily during execution.

- Insertion at the beginning is a constant time operation and takes  $O(1)$  time, whereas in an array, inserting an element at the beginning takes  $O(n)$  time, where  $n$  is the number of elements in the array.

## Disadvantages of Linked Lists

- Linked lists consume more memory than arrays as each node in the list contains a data item and a reference to the next node.
  - As the elements (or nodes) in a linked list are not stored in a contiguous fashion in memory, it requires more time to access the elements (or nodes) than arrays.
  - Appending an element (or node) to a linked list is a costly operation and takes  $O(n)$  time, where  $n$  is the number of elements (or nodes) in the linked list, unlike in arrays where it takes  $O(1)$  time.
- 

## Practice Problems

### 1. Left Short [<https://coding.ninja/P41>]

**Problem Statement:** You are given a singly linked list of integers. Modify the linked list by removing the nodes from the list with a greater valued node on their adjacent left in the given linked list.

#### Input Format:

The **first line** of the input contains the elements of the singly linked list separated by a single space and terminated by -1.

#### Output Format:

The modified, linked list. The elements of the modified list should be separated by single-space and terminated by -1.

#### Sample Input:

10 14 6 80 55 56 77 -1

#### Sample Output:

10 14 80 56 77 -1

#### Explanation:

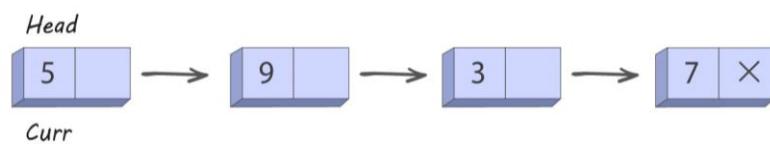
For the given linked list, nodes with the values 6 and 55 **have a greater valued node to their left** (14 and 80, respectively). Thus, we remove these 2 nodes from the linked list and get the

modified list as shown in the output. Note that the node with the value 56 should not be deleted as its initial left adjacent node had the value 55, which is not greater than 56.

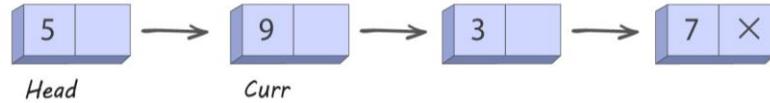
### Approach 1: Intuitive Solution

We use a variable **deletedValue** to keep track of the value of the last node deleted. This variable is initialised with the value -1 as it can never be a list element. If **deletedValue** is not equal to -1, this means that we have just deleted a node, and we need to compare the value of the next node with **deletedValue**. We iterate through the linked list, checking if the value of the next node is less than the value of the current node. Thus, the following situations arise:

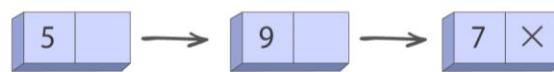
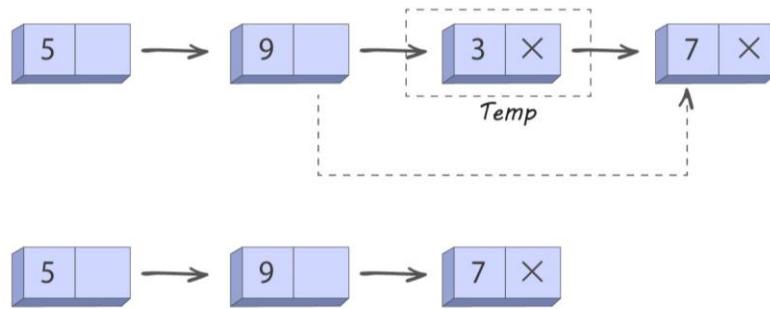
1. If the value of the next node is less than the value of the current node and **deletedValue** = -1, set **deletedValue** to the value of the next node. Then, delete the next node.
2. Else, if the **deletedValue** != -1 and the value of the next node is less than **deletedValue**, set **deletedValue** to the value of the next node. Then, delete the next node.
3. Else, for all other cases, set **deletedValue** to -1 and move to the next node.



Step 1:  $5 < 9$  and **deletedvalue** = -1



Step 2:  $9 > 3$  and **deletedvalue** = 3



**Time Complexity:** **O(N)**, N is the number of nodes in the linked list. It takes **O(N)** time to traverse through the entire list. Also, the deletion of a node takes **O(1)** time.

**Space Complexity:** **O(1)**, as we don't require any additional space.

### Approach 2: By reversing the list

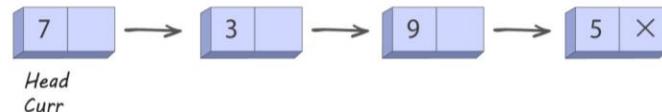
The main idea behind this approach is that it would be easier to start deleting the required nodes from the right as we need to check the left adjacent node for each node in the given linked list. This can be done by reversing the linked list.

### Steps:

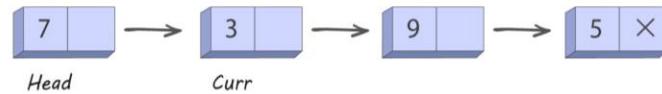
1. Reverse the given linked list.
2. After reversing the list, we traverse the list while checking if the next node is greater than the current node. Two situations arise:
  - a. If it is greater, we delete the current node and move it to the next node.
  - b. If it is not greater, we simply move to the next node.



Step 1: Reverse the list



As  $7 > 3$ ; curr = curr.next.



Step 2: As  $3 < 9$ ; delete curr and move curr to next node.



Step 3: As  $9 > 5$ ; curr = curr.next.



3. After iterating through the entire list, we reverse the linked list again to get the final modified list.

**Time Complexity:**  $O(N)$ , where  $N$  is the number of nodes in the linked list. It takes  $O(N)$  time to reverse the linked list. Also, traversing through the entire list takes  $O(N)$  times, and deletion of a node takes  $O(1)$  time. Thus, the overall time complexity is  $O(N + N * 1) = O(N)$ .

**Space Complexity:**  $O(1)$  as we don't require any additional space.

## 2. Sort Linked List [<https://coding.ninja/P42>]

**Problem Statement:** You are given a singly linked list of integers which is sorted based on absolute value. You have to sort the linked list based on actual values. The absolute value of a real number  $x$ , denoted  $|x|$ , is the non-negative value of  $x$ . The negative sign is ignored in the absolute value.

**For Example:**

If the given list is  $\{1 \rightarrow -2 \rightarrow 3\}$  (which is sorted on absolute value), the returned list should be  $\{-2 \rightarrow 1 \rightarrow 3\}$ .

**Input Format:**

The **first line** of input contains a single integer  $T$ , representing the number of test cases or queries to be run.

Then the  $T$  test cases follow.

The **first line** of each test case contains the elements of the singly linked list separated by a single space and terminated by  $-1$ . Hence,  $-1$  would never be a list element.

It is guaranteed that the given list is sorted based on absolute value.

**Output Format:**

For each test case, print the sorted linked list. The elements of the sorted list should be single-space separated, terminated by  $-1$ .

**Sample Input:**

```
1
1 -2 3 -1
```

**Sample Output:**

```
-2 1 3
```

**Approach 1: Insertion Sort**

In Insertion sort, the array is divided into three parts: Current element, Sorted part and Unsorted part. Everything before the current element is sorted, and everything after the current element is an unsorted part. In each iteration, we pick the first element from the unsorted part and then find the position of that element within the sorted list and insert it at that position. This is repeated until no element is present in the unsorted part.

**Steps:**

1. Initialise a node **curr = head**, which stores the current element.
2. Create and initialise a node **sorted\_head** to track the head of the sorted list. Initialize it as **sorted\_head = NULL**
3. Iterate the list until **curr != NULL**.
4. Store the next element after the **curr** in a node—that is, **currNext = curr→next**.
5. Insert **curr** in the partially sorted part and update the head of the sorted list.
6. Set **curr = currNext**.

7. Return **sorted\_head**.

**Time Complexity:**  $O(N^2)$ , where **N** is the number of nodes in the linked list. In the worst case, it takes  $O(N)$  time to insert the node in **sorted\_head** and to traverse the list takes  $O(N)$  time. Thus, the final time complexity is  $O(N^2)$ .

**Space Complexity:**  $O(N)$ , where **N** is the number of nodes in the linked list.  $O(N)$  additional space is required to store the **sorted\_head**.

### Approach 2: Merge Sort

We use the Merge sort algorithm to sort the given linked list. Merge sort is a divide and conquer algorithm. This algorithm will divide the list into two parts, recursively sort the two parts, and finally merge them so that the resultant list will be sorted.

#### Steps:

1. If the list contains only one node, return the head of the list.
2. Else, divide the list into two sublists. For this, we will take pointers **mid** and **tail** which will initially point to the head node. We will change **mid** and **tail** as follows until **tail** becomes NULL.
  - a. **mid = mid→next**
  - b. **tail = tail→next→next**The above operation will ensure that the **mid** will point to the middle node of the list. **mid** will be the head of the second sublist. Hence, we will change the **next** value of the node, which is before **mid** to NULL, thereby breaking the original linked list into two sublists.
3. Call the same algorithm for the two sublists using recursion.
4. Merge the two sublists and return the head of the merged list. For this, we will make a list **mergeList** which will be initially empty.
5. If the head of the first sublist has a value less than the head of the second sublist, then we will add the head of the first sublist in the **mergeList** and change the head to its next value.
6. Else if the head of the first sublist has a value greater than the head of the second sublist, we will add the head of the second sub list.
7. In the end, if any of the sublists becomes empty and the other sublist is non-empty, we will add all nodes of the non-empty sublist in the **mergeList**.

**Time Complexity:**  $O(N * \log(N))$ , where **N** is the number of nodes in the linked list. The algorithm used (Merge Sort) is divide and conquer. Therefore, for each traversal, we divide the list into two parts. Thus there are  $\log(N)$  levels, and the final complexity is  $O(N * \log(N))$ .

**Space Complexity:**  $O(\log(N))$ , where  $N$  is the number of nodes in the linked list. Since the algorithm is recursive and there are  $\log(N)$  levels, it requires  $O(\log(N))$  stack space.

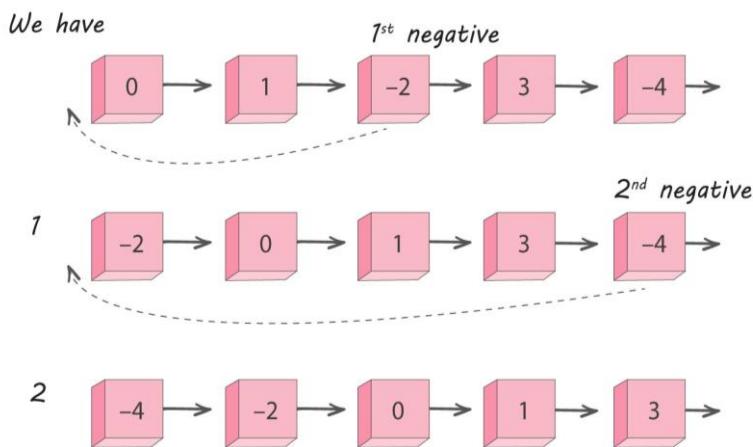
### Approach 3: Reverse Negative Values

We are given a singly linked list of integers as an input, which is sorted based on absolute value. This would mean that in the given list, all non-negative values are already present in sorted order. Also, if we look carefully, all negative values are present in reverse order. Hence, the idea of this approach is very simple yet intuitive; we have to reverse the negative values and move them before the first non-negative value (head of the linked list).

For example, say the given list is  $\{0 \rightarrow 1 \rightarrow -2 \rightarrow 3 \rightarrow -4\}$ . In this list, non-negative values  $\{0, 1, 3\}$  are present in sorted order, and the negative values  $\{-2, -4\}$  are present in reverse order.

#### Steps:

1. Traverse the linked list.
2. If we encounter a negative value node, we will move this node before the head node, thereby making this node the new head of the linked list.
3. After doing this, for all negative values, we will have a sorted list.



**Time Complexity:**  $O(N)$ , where  $N$  is the number of nodes in the linked list. We consider each node in the list exactly once. Thus the final time complexity is  $O(N)$ .

**Space Complexity:**  $O(1)$ , as only constant memory is utilised.

## 3. Make Maximum Number [<https://coding.ninja/P43>]

**Problem Statement:** Given a linked list such that each node represents a digit. Construct the maximum number possible from the given digits.

You just need to print the maximum Integer that can be formed.

#### **Input format :**

A **single line of input** contains the elements of the singly linked list separated by a single space and terminated by -1, denoting the end of the list.

#### **Output format :**

The maximum Number formed using the digits present in the linked list.

#### **Sample Input:**

1 2 2 0 9 -1

#### **Sample Output:**

92210

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

In this approach, we use the concept of sorting in descending order.

#### **Steps:**

1. Traverse in the linked list and store the elements in the vector or array.
2. Sort the array or vector and then reverse it.
3. Using the digits in the sorted structure, form a string of the number, which will be the largest possible number.

**Time Complexity:**  $O(N \log(N))$ , where **N** is the number of nodes in the linked list. Since sorting is involved in forming the largest number from digits, the overall time complexity is  $O(N \log(N))$ .

**Space Complexity:**  $O(N)$ , where N is the number of nodes in the linked list. The extra space is used to create a vector of the same size as the length of the linked list.

#### **Approach 2: Optimised Solution**

In this approach, we will create a frequency array. This will help us reduce the time complexity.

#### **Steps:**

1. Traverse the linked list from start to end and make a frequency array of all digits (0-9) present in the linked list.
2. Create a string which will store the digits.
3. From the frequency array, first append all the 9's in the answer string, then append all the 8's at the end of the answer string and so on.
4. Repeat the same process for all digits from 9 to 1.

5. The answer string will thus hold the largest possible number.

**Time Complexity:** **O(N)**, where **N** is the number of nodes in the linked list. Since we only need to traverse in the linked list once, the overall time complexity is **O(N)**

**Space Complexity:** **O(1)**, as constant space is used to create a fixed-size array of size 10 as a frequency array.

## **4. Add First and Second Half** [<https://coding.ninja/P44>]

**Problem Statement:** You are given a linked list of **N** nodes where each node represents a single digit. Return the head node of the linked list, storing the digits of the sum (most significant digit at the head) formed by adding the first half and second half of the given linked list.

**Note:**

1. When **N** is odd, consider the middle element to be a part of the first half.
2. The sum should not contain any leading zero except the sum 0 itself.
3. The most Significant Digit is the first digit of a number. For example, in 1234, '**1**' is the most significant digit. Also, '**4**' is the least significant digit.

**For example:**

Given a linked list: 1 → 2 → 3 → 4 → 5 → 5 → 6

First half: 1 → 2 → 3 → 4 = 1234

Second half: 5 → 5 → 6 = 556

Output linked list: (1234 + 556 = 1790) = 1 → 7 → 9 → 0

**Input format :**

A single line of input contains the elements of the singly linked list separated by a single space and terminated by -1, denoting the end of the list.

**Output format :**

A single line contains a string, the required sum.

**Sample Input 1:**

1 2 4 5 6 -1

**Sample Output 1:**

180

**Explanation for Sample Input 1:**

The first half of the given linked list is: 1 → 2 → 4

The second half of the given linked list is: 5 → 6

Sum of both parts = 124 + 56 = 180

**Sample Input 2:**

3 9 0 1 1 0 -1

### Sample Output 2:

500

### Explanation of Sample Input 2:

The first half of the given linked list is: 3 → 9 → 0

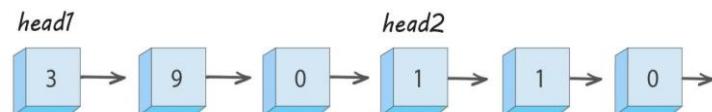
The second half of the given linked list is: 1 → 1 → 0

Sum of both parts =  $390 + 110 = 500$

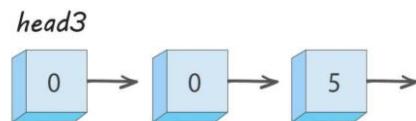
### Approach 1: Brute Force

#### Steps:

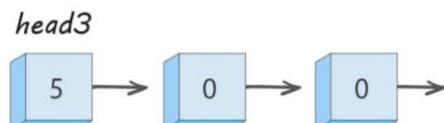
4. Iterate through the linked list and find its length.
5. Initialise two linked lists with their corresponding heads as **head1** and **head2** to store each half represented by numbers in the given linked list.
6. Iterate on the linkedlist till we reach the first half and create a new list to store the first half with head equal to **head1**. Similarly, we will keep the second half in a new list with head equal to **head2**.



7. Reverse both the lists so that their head nodes contain the least significant digit of both numbers. The least significant digit of a number refers to the last digit. For example, in 1234, '4' is the least significant digit.
8. Add two numbers represented by two linkedlist where the head of each denotes the least significant digit of two numbers.
9. To add them, we will iterate on both simultaneously and keep on adding the corresponding digits with a flag for **carry** and store the sum in a new list, say **head3**.
10. This list will have the required sum with its least significant digit in the **head3**.



11. Reverse the resultant list and return its head so that **head3** node now points to the most significant digit of the resultant sum.



**Time Complexity: O(N), where N is the total number of nodes in the given linked list.**

In the worst case, we will be iterating on the list twice. So the running time complexity is  **$2 * N$**  with a complexity of  **$O(N)$** .

**Space Complexity:  $O(N)$ , where  $N$  is the total number of nodes in the given linked list. In the worst case, extra space is required to maintain the 3 new linked lists (2 for each half and 1 for the resultant sum list).**

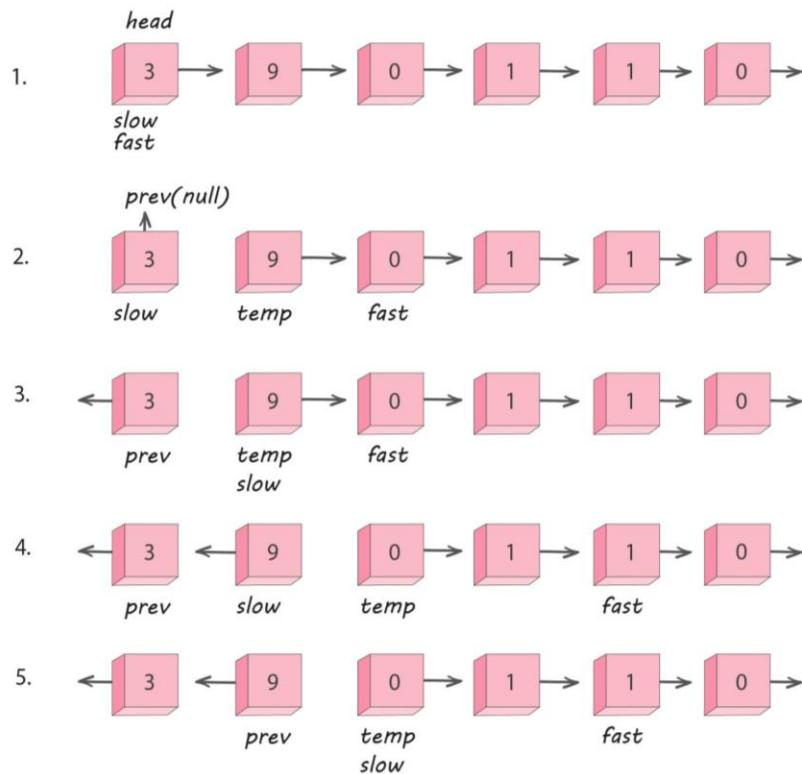
### Approach 2: Single Iteration

In this approach, instead of finding the length of the given linked list in one traversal, we will use two pointers slow and fast to traverse the list. In each iteration, the slow pointer will point to the next node, and the fast pointer will skip one node and point to the next node. This way, the **head1** and **head2** pointers find their correct positions much faster.

#### Steps:

1. Traverse the given linked list and store the two heads — **head1** and **head2**, for both halves using the fast and slow pointer approach.
2. While traversing the given list, reverse it until the fast pointer becomes null. In this way, we will have our first half with head **head1** having the least significant digit.
3. To reverse the linked list while traversing, we make use of two more pointers, **prev** and **temp**. The method is:
  - a. Initialise **prev** pointer to null.
  - b. Begin the iteration till **fast** or **fast→next** are not null.
  - c. During iteration, we
    - i. Do **fast = fast→next→next** (Skip one node)
    - ii. Create new node **temp = slow→next**.
    - iii. **slow.next = prev**
    - iv. **prev = slow**
    - v. **slow = temp**

Refer to the image below for a better understanding.



- Similarly, we will store the second half in the reversed order while traversing to the rest half using the **slow** pointer.
- Finally, we add both the lists and store them into the same pointer **head1**.
- Reverse **head1**, and return the answer. (Same as approach 1).

**Time Complexity:** O(N), where N is the number of nodes in the linkedlist. In the worst case, we will be iterating on the list only once using two pointers.

**Space Complexity:** O(1), as in the worst case, constant space is required.

## 5. Reverse List in K groups [\[https://coding.ninja/P45\]](https://coding.ninja/P45)

**Problem Statement:** You are given a linked list of **N** nodes and an integer **K**. You have to reverse the given linked list in groups of size **K**. If the list contains **x** nodes numbered from 1 to **x**, then you need to reverse each of the groups **(1, k),(k + 1, 2 \* k)**, and so on.

For example, if the list is 1 2 3 4 5 6 and **K** = 2, then the new list will be 2 1 4 3 6 5.

**Follow up:** Try to solve the question using constant extra space.

### Note:

- In case the number of elements in the list cannot be evenly divided into groups of size **K**, then just reverse the last group(irrespective of its size). For example, if the list is 1 2 3 4 5 and **K** = 3, then the answer would be 3 2 1 5 4.

2. All the node values will be distinct.

#### **Input Format:**

The **first line** of input contains an integer **T** representing the number of test cases.

For each test case:

The **first line** contains a linked list whose elements are separated by space, and the linked list is terminated by -1.

The **second line** contains an integer **K**.

#### **Output Format:**

The only line of each test case contains the modified, linked list.

#### **Sample Input:**

```
1
1 2 3 4 5 6 -1
2
```

#### **Sample Output:**

```
2 1 4 3 6 5
```

#### **Explanation:**

We optimise by reversing the nodes in groups of two for the given test case to keep track of, and get the modified, linked list as 2 1 4 3 6 5.

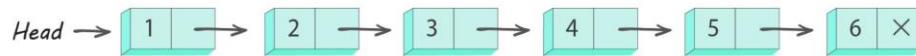
#### **Approach 1: Recursive Approach**

We can make use of recursion to solve this problem. The main idea is to reverse the first **K** nodes by ourselves and let the recursive function reverse the rest of the linked list.

#### **Steps:**

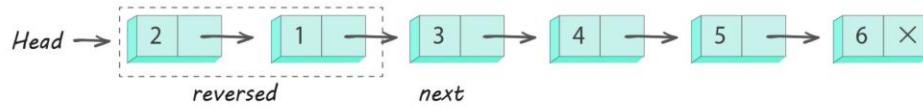
1. Let **reverseList()** be the recursive function that takes the head of the linked list and integer **K** as the parameters.
2. We can iteratively reverse the first **K** nodes of the linked list.
3. If we still have some nodes left in the linked list, we can call **reverseList()** until the nodes are exhausted.
4. To keep track of the exhaustion of nodes, we will simply check whether the next of the current node is null or not.

*Initially,*

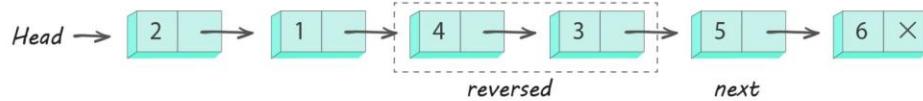


$k = 2$

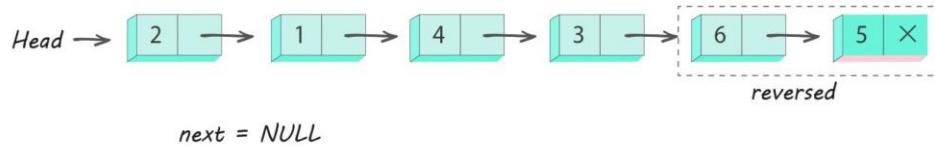
*First function call:*



*Second function call:*



*Third Recursive function call*



$next = \text{NULL}$

$\therefore$  Linked list is reversed in groups of  $K$  size

**Time Complexity:**  $O(N)$ , where  $N$  is the size of the linked list.

**Space Complexity:**  $O(N)$ , as we require a stack space for recursive calls of  $N$  nodes of the linked list.

## Approach 2: Space Optimised Approach

Since we have to optimize on space, instead of using recursion, we can use some pointers to create the links between the different sets of  $K$  nodes.

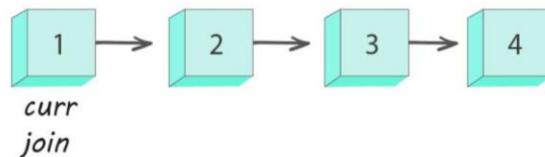
### Steps:

1. Create some pointers to keep the track of a current set of  $K$  nodes and the previously modified linked list.
2. Let **curr** denote the current node that we are on. Let **prev** denote the previous pointer. The **tail** pointer keeps the track of the last node of the  $K$  set of nodes, and **join** pointer points to the current set of nodes which we have to join to the modified, linked list.
3. Now keep on traversing the list until we reach the end and do the following:
  - a. Let **join = curr** and **prev = NULL**. Now reverse the  $K$  set of nodes in the linked list.
  - b. If we have not assigned the **newHead**, then assign the **newHead** to the **prev** pointer.
  - c. If we already have the tail of the linked list, then the next of tail should point to the **prev**—that is, **tail → next = prev**.
  - d. Now update the tail to the last node of the K-reversed linked list—that is, **tail = join**.

For example, let the linked list be 1 2 3 4 and K = 2.

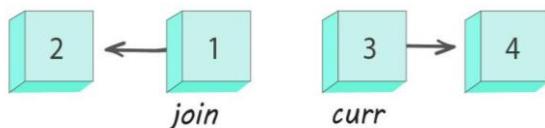
Initially, we will have **curr** pointing to the head of the list. Until we reach the end, we do the following steps:

1. Set **join** = **curr** and **prev** = **NULL**. Hence **join** will point to 1 initially.

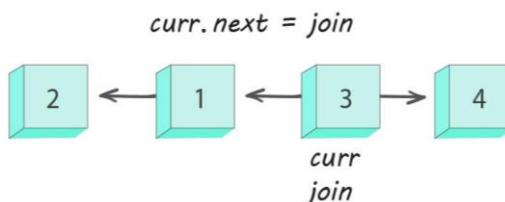


2. After reversing the first **K** nodes, **prev** will point to 2.
3. Assign the tail to **join** because the **join** was initially pointing to the head. After reversing the **K** nodes, it will point to the last node of the current set of nodes.

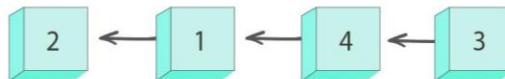
*After reversing K = 2 nodes*



4. We repeat the same process again, and hence we will get the required output.



*After reversing next K = 2*



**Time Complexity:**  $O(N)$ , where **N** is the size of the Linked List.

**Space Complexity:**  $O(1)$ , as we are using constant extra space.

## 6. Implement Stack with Linked List [<https://coding.ninja/P46>]

**Problem Statement:** Implement the Stack data structure using a Singly Linked List.

Create a class named **Stack** which supports the following operations (all in  $O(1)$  time):

**getSize**: Returns an integer. Gets the current size of the stack.

**isEmpty**: Returns a boolean. Gets whether the stack is empty.

**push**: Returns nothing. Accepts an integer. Puts that integer at the top of the stack.

**pop**: Returns nothing. Removes the top element of the stack. Does nothing if the stack is empty.

**getTop**: Returns an integer. Gets the top element of the stack. Returns -1 if the stack is empty.

#### **Input Format:**

The **first line** of the input will contain the number of queries, **T**.

The **next t lines** will contain the **queries**. They can be of the following five types:

'**1**': Print the current size of the stack.

'**2**': Find whether the stack is empty. Print "true" if yes, print "false" otherwise.

'**3**': This query will be given like "3 val" where val can be any non-negative integer. Put val on the top of the stack. Print nothing.

'**4**': Remove the top element of the stack. Print nothing.

'**5**': Print the top element of the stack. If the stack is empty, print -1.

#### **Output Format:**

Print the result of each query on a separate line. If the query is '3' or '4', print nothing (not even an empty line).

#### **Sample Input:**

```
4
3 5
3 4
1
2
```

#### **Sample Output:**

```
2
false
```

#### **Explanation:**

The first two queries ('3') push 5 and 4 on the stack. Hence, the size of stack becomes 2.

Therefore the third query ('1') prints the size. Since the stack is not empty, the fourth and final query ('2') prints "false".

<b>No of queries</b>	<b>5</b>
3 - Push	5
3 - Push	4
1 - size of stack	2
2 - stack is empty or not	False

### Approach:

Maintain a linked list. Keep track of its head and size at all times, and update them accordingly whenever a new operation is performed.

We can implement all functions of the stack using linked list in the following manner:

### Steps:

1. First, initialise a **head** node and the **size** of the list as **NULL** and **0**, respectively.
2. For the **push** function, insert new elements at the head of the list—that is, the new element will become the new head of the list, and the previous head will become the next new head. Also, increase the size by one.
3. For **pop**, move the head to its next and delete the original head (if required). Also, decrease the **size** by one. If the head was null or none—that is, the list is empty, do nothing.
4. For **getTop**, return the head's data or **-1** if the list is empty.
5. For **getSize**, return the size of the list.
6. For **isEmpty**, return true if the size is **0**; otherwise, return false.

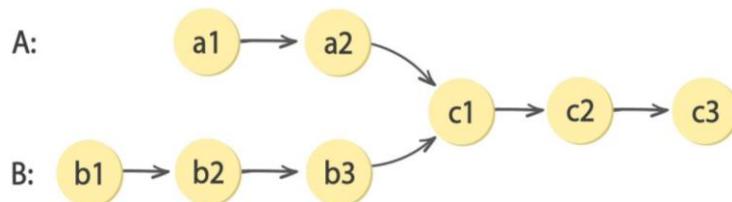
**Time Complexity:** **O(1)** for all operations. Since we are inserting and deleting elements from the **head** of the list itself, constant time is spent on pop and push functions. The other functions are also simply returning a value, so they also take constant time.

**Space Complexity:** **O(N)**, where **N** is the number of elements pushed in the stack.

## 7. Intersection of Linked Lists [\[https://coding.ninja/P47\]](https://coding.ninja/P47)

**Problem Statement:** You are given two singly linked lists of integers, which are merging at some node. Find the data of the node at which merging starts. If there is no merging, return **-1**.

For example, the given Linked Lists are merging at node **c1**.



### Input Format:

All three lines contain the elements of the singly linked list separated by a single space and terminated by **-1**.

First line would contain : **a1, a2, ...an, c1, -1**.

The second line would contain: **b1,b2,...bm,c1,-1**.

The third line would contain : **c2, c3, ....ck, -1**.

### Output Format:

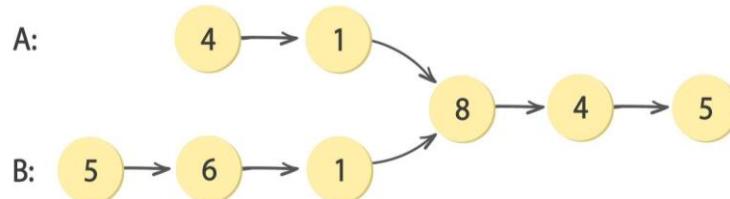
The only line of output contains data of the first merged node. If there is no merging, the output should contain -1.

#### Sample Input:

4 1 8 -1

5 6 1 8 -1

4 5 -1



#### Sample Output :

8

#### Approach 1: Brute Force

For each node in the first list, traverse the entire second list. Check if any node in the second list coincides with the first list:

1. If it does, return the node's data.
2. If it doesn't, return -1.

**Time Complexity:**  $O(N * M)$ , where **N** and **M** are the lengths of the first and second linked lists respectively. Since we are traversing the second list for each element of the first list, the complexity is  $O(N * M)$ .

**Space Complexity:**  $O(1)$ , as no extra memory has been allocated.

#### Approach 2: Hashing Approach

In this approach, to improve our time complexity, we shall make use of hashing. Instead of traversing the second linked list for each node of the first linked list, we shall store their references in a hash set/map/dictionary, traversing over them separately.

#### Steps:

1. Traverse the first list and store the address/reference to each node in a hash set.
2. Then check every node in the second list:
  - a. If the address of the node of the second list appears in the hash set, then the node is the intersection node. Return it.
  - b. If we do not find any address of the second list node in the hash set, return -1.

**Time Complexity:**  $O(N + M)$ , where **N** and **M** are the lengths of the first and second linked lists respectively. Since we are traversing on the first and second linked list separately, the time complexity is  $O(N + M)$ .

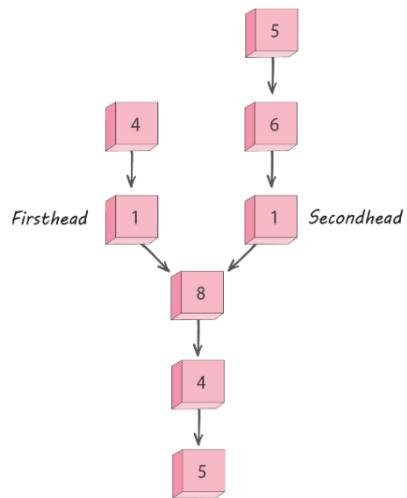
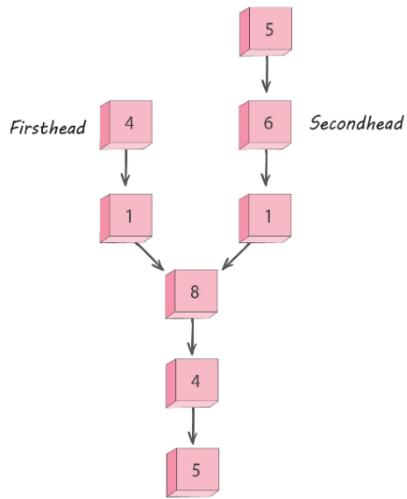
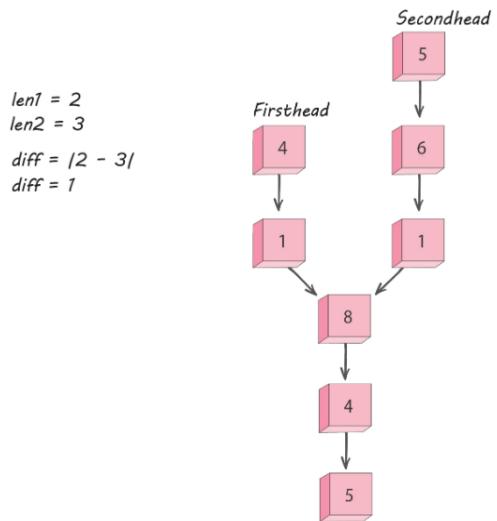
**Space Complexity:**  $O(N)$  or  $O(M)$  depending on which list's nodes you store in the hash set.

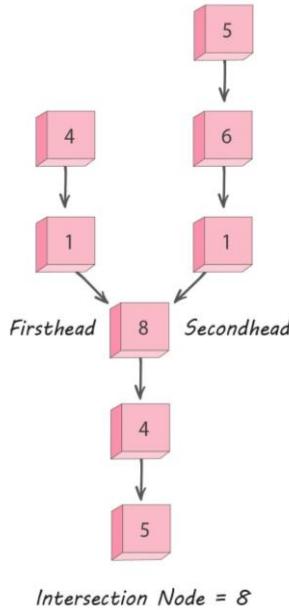
### Approach 3: Parallel Traversal Approach

In the previous approach, we used some extra space for our hashmap. Instead of doing that, we can traverse our linked lists parallelly to find our answer.

#### Steps:

1. Calculate the length of both the lists, say **len1** and **len2**.
2. Get the absolute difference of the lengths, **diff** =  $|len1 - len2|$ .
3. Now traverse the longer list from the first node till **diff** nodes so that from there onwards both the lists have an equal number of nodes.
4. Then traverse both the lists in parallel and check whether a common node is reached  
(Note that getting a common node is done by comparing the address of the nodes, not the data.)
  1. If **yes**, return that node's data.
  2. If **no**, return -1.





**Time Complexity:**  $O(N + M)$ , where  $N$  and  $M$  are the lengths of the first and second linked lists respectively. Since we have traversed both the linked lists twice, the time complexity is  $O(N + M)$ .

**Space Complexity:**  $O(1)$ , since we only use constant space.

## **8. Sum Between Zeroes** [<https://coding.ninja/P48>]

**Problem Statement:** You are given a singly linked list that contains a series of integers separated by a '0'. Between two zeroes, you have to merge all the nodes lying between them into a single node which contains the sum of all the merged nodes. The output list shall not contain the partitioning 0s. You have to perform this in place.

### **Example:**

If the given linked list is:  $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 0 \rightarrow 4 \rightarrow 5 \rightarrow 0 \rightarrow 6 \rightarrow 0 \rightarrow \text{null}$ , then the linked list is converted to  $6 \rightarrow 9 \rightarrow 6 \rightarrow \text{null}$ .

**Explanation:** Taking 0s as the start and end in reference to a sequence, we can see that there are three sequences. They are:

1.  $1 \rightarrow 2 \rightarrow 3$ , which sum up to 6
2.  $4 \rightarrow 5$ , which sum up to 9
3.  $6$ , which sum up to 6 only.

### **Note:**

It is guaranteed that there will be no two consecutive zeros, and there will always be a zero at the beginning and end of the linked list.

### **Input Format:**

The **first line** of input contains the elements of the singly linked list separated by a single space and terminated by a -1. Hence, -1 will not be an element of the list.

#### Output Format:

The **first and the only** output line contains the integers present in the linked list after all the merging operations have been performed.

#### Sample Input:

0 1 2 4 8 16 0 -1

#### Sample Output:

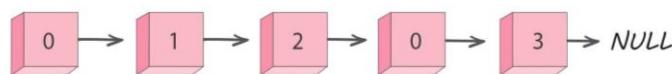
31 -1

#### Approach: Two Pointer Approach

In this approach, we shall use two pointers to iterate over our list. Let us initialise two pointers, **newHead**, and **newTail**, with NULL (These will be the head and tail of the final list).

#### Steps:

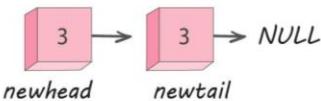
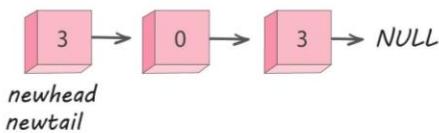
1. Traverse the given list.
2. Ignore the first zero.
3. As you encounter non-zero nodes, add their values in a variable called **sum**. As soon as you encounter a node with data 0, change that node's value to **sum**, and
  - a. If **newHead** is NULL, this node becomes the new head and tail of the list. (this means that we have summed up the first sequence present between the 0s, which now becomes the head of our resultant linked list.)
  - b. If **newHead** is not NULL, then connect this node with the tail of the list and assign it to be the **newTail**.



*newhead = NULL*

*newtail = NULL*

$$\text{Sum} = 1 + 2$$



Note that we are not creating a new list because we are not creating new nodes. Instead, we are changing the values of existing nodes and modifying their connections.

**Time Complexity:**  $O(N)$ , where  $N$  is the size of the linked list. Traversing the linked list once takes  $O(N)$  time.

**Space Complexity:**  $O(1)$ , as constant space is required.

## 9. Rearrange Linked List [<https://coding.ninja/P49>]

**Problem Statement:** You have been given a singly linked list in the form of  $L_1 \rightarrow L_2 \rightarrow L_3 \rightarrow \dots L_n$ . Your task is to rearrange the nodes of this list to make it in the form of  $L_1 \rightarrow L_n \rightarrow L_2 \rightarrow L_{n-1} \rightarrow \dots$ . You are not allowed to alter the data of the nodes of the given linked list.  
If it is impossible to rearrange the list into the specified form, return an empty list, that is, NULL.

### **Input Format:**

The first line of input contains an integer  $T$ , representing the number of test cases or queries to be processed. Then the test case follows.

The **only line** of each test case contains the elements of the singly linked list separated by a single space and terminated by -1. Hence, -1 would never be a list element.

### **Output Format:**

Print the linked list in the specified form. The elements of the linked list must be separated by a single space and terminated by -1.

### **Sample Input:**

1 2 3 4 5 6 -1

### **Sample Output:**

1 6 2 5 3 4 -1

### **Explanation:**

For the given test case-

$L_1 = 1, L_2 = 2, L_3 = 3, L_4 = 4, L_5 = 5, L_6 = 6$ .

Our answer will be  $L_1 \rightarrow L_6 \rightarrow L_2 \rightarrow L_5 \rightarrow L_3 \rightarrow L_4$ .

Hence, the output is  $1 \rightarrow 6 \rightarrow 2 \rightarrow 5 \rightarrow 3 \rightarrow 4$

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

The naive solution is to maintain a pointer **front** that will process the nodes from the front side of the linked list and keep removing the nodes from the backside. While removing nodes from the back, we will put them in front of the list after the **front** pointer and move the **front** pointer ahead.

### Steps:

1. Initialise the **front** pointer to head.
2. Loop till next of **front** pointer is not NULL.
  - a. Extract the last node from the list and insert it as the next of **front** pointer; thus, **front→next = last→next**.
  - b. Move the **front** pointer to the next of **last**.

**Time Complexity:**  $O(N^2)$ , where **N** is the number of nodes in the linked list. Extracting the last node from the linked list will take linear time, and we will be extracting at most  $N/2$  times. Hence, the overall complexity will be  $N/2 * O(N)$ , that is,  $O(N^2)$ .

**Space Complexity:**  $O(1)$ , as no extra space is required.

### Approach 2: Split and Merge

A very clever approach is to split the linked list and then merge it in a manner that gives us our answer.

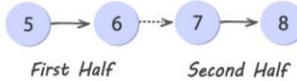
### Steps:

1. Find the middle node of the given linked list.
2. Split the linked list into two halves from the middle node.
3. Then, the second half of the linked list is reversed.
4. Now, merge the nodes of both halves alternatively to make a list into the specified form.

*Before*



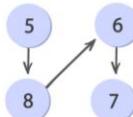
*After locating midpoint*



*After reversing second half*



*After merging the two halves (updating the linkage)*



**Time Complexity:**  $O(N)$ , where  $N$  is the number of nodes in the linked list. Finding the middle node, reversing the second half of the list, and merging the two halves all these operations will take linear time.

**Space Complexity:**  $O(1)$ , as no extra space is required.

## 10. Merge Two Sorted Linked Lists [<https://coding.ninja/P50>]

**Problem Statement:** You are given two sorted linked lists. You have to merge them to produce a combined sorted linked list. You need to return the head of the final linked list.

**For example:**

If the first list is:  $1 \rightarrow 4 \rightarrow 5 \rightarrow \text{NULL}$  and the second list is:  $2 \rightarrow 3 \rightarrow 5 \rightarrow \text{NULL}$ . Thus the final list would be  $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 5 \rightarrow \text{NULL}$ .

**Note:**

1. The given linked lists may or may not be null.
2. Try to solve this problem in linear time complexity and constant space complexity.

**Input Format:**

The first line of input contains an integer  $T$  representing the number of test cases or queries to be processed. Then the test case follows.

The first line of each test case contains the elements of the first linked list separated by a single space and terminated by  $-1$ . Hence,  $-1$  would never be a list element.

The second line of each test case contains the elements of the second linked list separated by a single space and terminated by  $-1$ .

**Output Format:**

Print the final linked list. The elements of the linked list must be separated by a single space and terminated by  $-1$ .

**Sample Input:**

```
2 1 4 5 -1
2 3 5 -1
```

**Sample Output:**

```
1 2 2 3 4 5 5 -1
```

**Explanation:**

The first list is:  $2 \rightarrow 1 \rightarrow 4 \rightarrow 5 \rightarrow \text{NULL}$ .

The second list is:  $2 \rightarrow 3 \rightarrow 5 \rightarrow \text{NULL}$ .

The final list would be  $1 \rightarrow 2 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 5 \rightarrow \text{NULL}$ .

**Approach 1: Recursive**

Without loss of generality, let's say the head of the first linked list is smaller than the head of the second linked list. Now, to create our merged linked list, we follow the following procedure.

#### Steps:

1. Among the respective heads of the first and second list, the current node would point to the smaller of the two.
2. We run a recursive function, pointing to the next element of the head of the first linked list.
3. If only one of the lists is empty, we return the head of the other list.
4. If both the lists are empty, we return null.

**Time Complexity:**  $O(N + M)$ , where **N** and **M** are the number of nodes in both linked lists. We will be traversing both linked lists once, which will take linear time on both.

**Space Complexity:**  $O(N + M)$ , where **N** and **M** are the number of nodes in both linked lists—considering the recursive stack space.

#### Approach 2: Iterative

In this approach, instead of using recursion, we simply follow a looping logic.

#### Steps:

1. If one of the given lists is NULL, return the head of the other list.
2. The data of the head of the first list should be less than or equal to the data of the head of the second list.
3. Traverse both the linked lists simultaneously.
4. If the current node of the second linked list is smaller than the current node of the first linked list, insert the current node of the second linked list before the current node of the first linked list.
5. This is done by making the next node of the previous node of the first linked list as the current node of the second linked list. Further, make the next node of this node as the current node of the first linked list.
6. If the first list has reached the end and the second list hasn't, point its next node to the head of the second list.

**Time Complexity:**  $O(N + M)$ , where **N** and **M** are the number of nodes in both linked lists.

**Space Complexity:**  $O(1)$ . We are storing the result of the addition in one of the given linked lists. Hence, we are not using any extra space.



# 6. Linked Lists – II

---

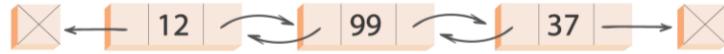
In the previous chapter, we discussed linked lists. We also discussed different types of linked lists and worked extensively on singly linked lists.

In this chapter, we will discuss the other two types of linked lists—doubly linked lists and circular lists in detail. Later we would work on some problems involving different types of linked lists.

## Doubly Linked Lists

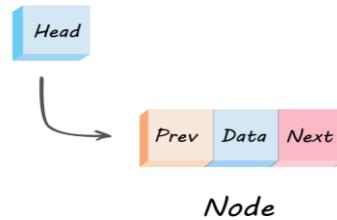
In a doubly linked list, each node contains an extra pointer pointing towards the previous node in addition to the pointer pointing to the next node.

The advantage of using doubly linked lists is that we can navigate in both directions.



Doubly linked lists contain a head pointer that points to the first node in the list. Head is null if the doubly linked list is empty.

Each node in a doubly-linked list has three elements: **data**, **previous** pointer (address of the previous node), **next** pointer (address of the next node).



## Operations on Doubly Linked Lists

Let us understand the two main operations involving doubly linked lists—that is, insertion and deletion.

### Insertion Operation

#### Case 1: Insert at the beginning of the linked list

**InsertAtBeginning(x):** This operation involves inserting a node with the given data at the head or beginning of a doubly linked list.

#### Pseudo code

```
function insertAtBeginning(x)
/*
    create a new node named: 'newNode'
    set data of new node's data to x
*/
newNode.data = x
newNode.prev = null
newNode.next = null
// If list is empty, set head as 'newNode'
if head is null
    head = newNode
    return head
/* If the list is not empty, make the next pointer of 'newNode' point to the head
of the existing list and the previous pointer of the first node point to the 'newNode',
thereby converting the first node in the existing list to the second node in the list and
set the head to 'newNode' making it the first node in the list */
newNode.next = head
head.previous = newNode
head = newNode
return head
```

**Case 2: InsertAtEnd(x):** This operation involves inserting a node with the given data at the end of a doubly linked list.

#### Pseudo code:

```
function insertAtEnd(x)
/*
```

```

        create a new node named 'newNode'

        set data of newNode's data to data

    */

    newNode.data = x

    newNode.prev = null

    newNode.next = null

    // If list is empty, set head as 'newNode'

    if head is null

        head = newNode

        return head

    /*

        Otherwise, create a 'cur' node pointer and keep moving it
        until it reaches the last node to reach the end of the list

    */

    cur = head // We store the head in a separate variable for traversal to retain
    // the original head

    while cur.next is not null

        cur = cur.next

    /*

        The pointer 'cur' is now at the end of the list, so 'newNode' will be linked to
        'cur', so that the next of 'cur' points to 'newNode'. Also, the previous of
        'newNode' will point to 'cur'.

    */

    cur.next = newNode

    newNode.previous = cur

    return head

```

**Case 3: InsertAtIndex(idx, data):** This operation involves inserting a node with the given data at the given index.

## Pseudo Code

```
function insertAtGivenIndex(idx, x)
    /*
        create a new node: newNode
        set newNode's data to x
    */
    newNode.data = x
    newNode.prev = null
    newNode.next = null
    // call insertAtBeginning if idx = 0
    if idx is 0
        // case of insertion at beginning
        return insertAtBeginning(data)
    count = 0
    cur = head
    while count < idx - 1 and cur.next is not null
        count = count + 1
        cur = cur.next

    /*
        If count does not reach (idx - 1), then the given index
        is greater than the current size of the list
    */
    if count < idx - 1
        print "invalid index"
        return head
    /*
        Link 'newNode' with the first part of the linked list by creating a two-way
        link between the 'cur' node and the 'newNode' otherwise setting the
        newNode next field as the address of the node present at position idx
    */

```

```

*/



newNode.next = cur.next

cur.next = newNode

newNode.prev = cur

if nextNode is not null

    newNode.next.prev = newNode

return head

```

## Deletion Operation

Case 1: DeleteFromBeginning(): Deleting a node from the head (or beginning) of a linked list.

### Pseudo Code

```

function deleteFromBeginning()

    // if the list is empty, return head is null (there is no node to be deleted), return

    if the head is null

        print "Linked List is Empty"

        return head

    /*

        Otherwise, set the new head to the second node in the linked list, make
        the 'prev' pointer of the second node point to null, and finally delete the
        first node from the linked list.

    */

    temp = head

    head = head.next

    head.prev = null

    delete temp

    return head

```

**Case 2: DeleteFromEnd():** Deleting a node from the end of a linked list.

#### Pseudo Code

```
function deleteFromEnd()

    // If the list is empty, return null (there is no node to be deleted)

    if head is null

        print "Linked List is Empty"

        return head

    /*

        Keep a 'cur' pointer and let it point to the head initially; move the 'cur'
        pointer to the last node in the list

    */

    cur = head

    while cur.next is not null

        cur = cur.next

        prevNode = cur.prev

        prevNode.next = null

        delete cur

    return head
```

**Case 3: DeleteFromIndex(idx):** Deleting a node at a given index.

#### PseudoCode

```
function deleteFromGivenIndex(idx)

    // If the list is empty, return null (there is no node to be deleted)

    If head is null

        print "Linked List is empty"

        return null
```

```
// call deleteFromBeginning if idx = 0  
  
if idx == 0  
  
    // case of deletion from beginning  
  
    deleteFromBeginning()  
  
    return head  
  
count = 0  
  
cur = head
```

```
/*
```

Have a pointer 'temp' to reach the previous node at the given 'index', thereby making 'temp' point to that node whose next node is to be deleted.

```
*/
```

```
while count < idx - 1 and cur.next is not null
```

```
    count = count + 1
```

```
    cur = cur.next
```

```
/*
```

If the 'cur' node reaches null or the tail of the list, then the given index is greater than the size of the list.

```
*/
```

```
If cur. next is null
```

```
    print "Invalid Index"
```

```
    return head
```

// Otherwise, link the 'cur' node (previous node of the node to be deleted) with the next node of the node to be deleted.

```
    nextNode = cur.next
```

```
    prevNode = cur.prev
```

```
    prevNode.next = nextNode
```

```
    nextNode.prev = prevNode
```

```
    delete cur
```

**return head**

## Time Complexity of Various Operations

Let  $n$  be the number of nodes in a doubly linked list. The complexity of linked list operations in the worst case can be given as:

Operations	Time Complexity
insertAtBeginning(x)	O(1)
insertAtEnd(x)	O(n)
insertAtGivenIdx(idx, x)	O(n)
deleteFromBeginning()	O(1)
deleteFromEnd()	O(n)
deleteFromGivenIdx(idx)	O(n)

## Advantages of Doubly Linked Lists

- Doubly Linked Lists can be traversed in both directions.
- Deletion is easy in doubly linked lists if we know the node's address to be deleted, whereas, in singly-linked lists, we need to traverse the list to get the previous node.

## Disadvantages of Doubly Linked Lists

- We need to maintain an extra pointer for each node.
- Every operation in doubly linked lists requires updating of an extra pointer.

## Applications of Doubly Linked Lists

- Web browsers use doubly linked lists for backward and forward navigation of web pages.
- LRU (Least recently used) / MRU (most recently used) cache is constructed using doubly linked lists.
- They are used by various applications to maintain undo and redo functionalities.
- In operating systems, the thread scheduler maintains a doubly-linked list to keep track of processes that are being executed at that time.

## Circular Linked Lists

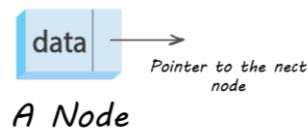
Circular linked lists are linked lists where the 'next' pointer of the last node is pointing to the first node of the list.

### Why Circular Linked Lists?

The advantage of using a circular linked list is that when we want to traverse in only one direction and move to the first node cyclically, we don't need to store additional pointers to mark the first and the last node. Typical usage of this concept is to implement queues using one pointer.



Each node in a circular linked list has two properties: **data**, **next (pointer to the next node)**.



## Operations on Circular Linked Lists

The operations on a circular linked list are the same as a singly linked list. The following three functions are, however, slightly modified, which is discussed in detail ahead:

- **insertAtBeginning(x):** Inserting a node with given data at the head (or beginning) of a circular linked list.
- **deleteFromBeginning():** Deleting a node from the head (or beginning) of a circular linked list.
- **display():** Display the contents of a circular linked list

### Insert at beginning

```
function insertAtBeginning(x)
```

```
/*
```

```

create a new node named 'newNode'

set data of 'newNode' to x

 $\ast\ast$ 

newNode.data = x

newNode.next = null

// If the list is empty, newNode will be the first node inserted to the list
hence make head point to newNode

if head is null

    head = newNode

    head.next = head

    return head

// traverse to the end of the circular list where the 'next' pointer of a node is
equivalent to the head node and the 'temp' node points to the last node in the list

temp = head

while temp.next is not equal to head

    temp = temp.next

// link 'newNode' and the first node represented by 'head' pointer. Then, link
the 'temp' node and 'newNode' by setting the 'next' pointer of 'temp' to 'newNode'.
Finally, set the 'newNode' as the new head.

    newNode.next = head

    temp.next = newNode

    head = newNode

    return head

```

## Delete from beginning

```

function deleteFromBeginning()

// if the list is empty, return null (there is no node to be deleted)

if head is null

    print "Linked List is Empty"

```

```

    return head

    // if there is a single node in the list, delete it

    if head.next equals head

        head = NULL

        return head

    // traverse to the end of the circular list where the 'next' pointer of a node is
    // equivalent to the head node and the 'temp' node points to the last node in the list

    temp = head

    while temp.next is not equal to head

        temp = temp.next

    // link the last node (represented by 'temp') and the second node (represented by
    // the next pointer of head) and delete the head, thereby deleting the first node in the list.

    temp.next = head.next

    delete head

    // finally, set the new head as the second node in the existing list.

    head = temp.next

    return head

```

### Display (function does not return anything)

```

function display()

    // If the list is empty, print nothing

    if the head is null

        print "Linked List is empty"

        return

    // traverse to the end of the circular list where the 'next' pointer of a node is
    // equivalent to the head node and the 'temp' node points to the last node in the list

    temp = head

    while temp.next is not equal to head

        print temp.data

```

```

temp = temp.next
print temp.data
return

```

## Time Complexity of the Discussed Operations

Let **n** be the number of nodes in a circular linked list. The complexity of linked list operations in the worst case can be given as:

<b>Operations</b>	<b>Time Complexity</b>
<code>insertAtBeginning(x)</code>	$O(n)$
<code>deleteFromBeginning()</code>	$O(n)$
<code>display()</code>	$O(n)$

## Advantages of Circular Linked Lists

- We can start traversing the list from any node. We just have to keep track of the first visited node.
- Circular linked lists make the implementation of data structures like queues a lot easier and more space-efficient than singly linked lists.
- It saves time when we have to go to the first node from the last node. It can be done in a single step because there is no need to traverse the in between nodes.
- Instead of the head pointer, if we keep track of the last node in a Circular Linked List, insertion and deletion operations can be done in  $O(1)$  time.

## Disadvantages of Circular Linked Lists

- Operations like insertion and deletion from the beginning become very expensive as compared to singly linked lists. We need to maintain an extra pointer that marks the beginning of the list to prevent getting into an infinite loop.

## Applications of Circular Linked Lists

- Circular lists can be used to implement Fibonacci Heap efficiently.
- They can be used to implement circular queues, which have applications in CPU scheduling algorithms like round-robin scheduling.

- They are used to switch between players in multiplayer games.
- 

## Practice Problems

### 1. Circularly Linked [<https://coding.ninja/P51>]

**Problem Statement:** You are given the head of a linked list containing integers. You need to find out whether the given linked list is circular or not.

#### Note:

1. A linked list is said to be circular if it has no node having its next pointer equal to NULL and all the nodes form a circle; that is, the next pointer of the last node points to the first node.
2. All the integers in the linked list are unique.
3. If the linked list is non-circular, the next pointer of a node with  $i^{\text{th}}$  integer is linked to the node with data  $(i+1)^{\text{th}}$  integer. If there is no such  $(i+1)^{\text{th}}$  integer, then the next pointer of such node is set to NULL.

#### Input Format:

The **first line** of input contains an integer **T**, denoting the number of test cases.

The **first line** of each test case consists of an integer **N**, denoting the number of links in the linked list.

The **second line** of each test case consists of **N** space-separated integers denoting the linked list data and linking between nodes of the linked list as described in the notes above.

#### Output Format:

For each test case, print True or False, in a separate line depending on whether the linked list is circular or not.

#### Sample Input:

5

1 2 3 4 1

### Sample Output:

True

### Explanation:

The given linked list would look like as shown here. Since the fourth node links to the first node, it is a circular link and thus True is returned.



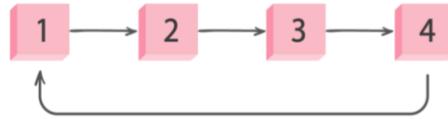
### Approach: Slow and Fast Pointer

In this approach, we will use the concept of fast and slow pointers to find out whether the list is circular or not.

### Steps

1. Take two node pointers and initialise them to head of the linked list.
2. Iterate through the linked list as follows:
  - a. In each iteration, move the slow pointer by one node and fast by two nodes (if they exist).
3. If a fast pointer becomes NULL at some point, the linked list has some endpoint; hence, it is not circular.
4. If slow and fast pointers meet at the same node at some point in time, then this means the linked list contains a loop.
5. To determine whether the linked list is circular or not, check the node at which slow and fast pointers become equal (meet).
  - a. If the node at which they become equal is the head node, then the given linked list is circular.
  - b. If it is not the head node, then the given linked list is not circular.

In the linked list given below:



1. Here, the head is the pointer to the node with data = 1, now initialise slow and fast with head.
2. After the 1<sup>st</sup> iteration, slow will point to the node with data = 2, and fast will point to the node with data = 3.
3. After the 2<sup>nd</sup> iteration, slow will point to the node with data = 3, and fast will point to the node with data = 1.
4. After the 3<sup>rd</sup> iteration, slow will point to the node with data = 4, and fast will point to the node with data = 3.
5. After the 4<sup>th</sup> iteration, slow will point to the node with data = 1, and fast will point to the node with data = 1.
6. At this point, slow and fast pointers start pointing to the same node, which is the same as the head; hence the given linked list is circular.

**Time Complexity:**  $O(N)$ , where  $N$  denotes the length of the linked list.

**Space Complexity:**  $O(1)$ , as we are using constant memory.

## 2. Find Pair with a Given Sum in a Doubly Linked List

[<https://coding.ninja/P521>]

**Problem Statement:** You are given a sorted doubly linked list of size  $N$ , consisting of positive integers and also provided a number  $K$ . Your task is to find out whether there exists a pair in the doubly linked list with the sum  $K$  or not. If there exists a pair, then you can return TRUE else, return FALSE.

**Note:**

A doubly linked list is a type of linked list that is bi-directional—that is, it can be traversed in both directions, both forward and backward.

**Input Format:**

The **first line** of input contains a single integer  $T$ , representing the number of test cases.

Then the  $T$  test cases follow.

The **first line** contains an integer  $K$  which represents the sum.

The **second line** contains the elements of the doubly linked list separated by a single space and terminated by -1. Hence, -1 would never be a list element.

### **Output Format:**

For each test case, print YES if there exists a pair with a sum equal to K; else print NO.

The output of every test case will be printed in a separate line.

### **Sample Input:**

1

4

1 2 3 4 9 -1

### **Sample Output:**

YES

### **Explanation:**

For the doubly linked list,  $1 \leftarrow 2 \leftarrow 3 \leftarrow 4 \leftarrow 9$ , there exists a pair (1,3) with a sum equal to 4. Hence the output is YES.

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

We will find all the possible pairs by iterating over the whole doubly linked list **N** times, where **N** is the length of the linked list. At any time, if the sum of any pair is equal to **K**, we will return TRUE; else, we will keep on iterating till we traverse the complete list.

#### **Steps:**

1. Let's say the head of the given linked list is **HEAD**.
2. Initialise a pointer **PTR1 = HEAD**, and it will be used to iterate over the linked list.
3. Iterate while **PTR1** is not NULL.
  - a. Initialise a pointer that would point to the next of **PTR1**, like **PTR2 = PTR1 → NEXT**.
  - b. Iterate while **PTR2** is not NULL
    - i. If the sum of values at **PTR1** and **PTR2** is equal to **K**, then return true.
    - ii. Move **PTR2** ahead using **PTR2 = PTR2 → NEXT**.
  - c. Move **PTR1** ahead using **PTR1 = PTR1 → NEXT**.
4. If no such pair is found, return false.

**Time Complexity:**  $O(N^2)$ , where **N** denotes the length of the doubly linked list.

**Space Complexity:**  $O(1)$ , as we are using constant memory.

### **Approach 2: Hashing**

We will scan the doubly linked list from left to right, and we will use the Hash to store all previously encountered elements. So at each step, we will check if K - (current node data) already exists in Hash; if yes, we can say we have found a pair.

#### Steps:

1. Let's say the head of the given linked list is **HEAD**.
2. Initialise a pointer **PTR1 = HEAD**, which will be used to iterate over the linked list.
3. Initialise an unordered\_map called **MYMAP** as a hash.
4. Iterate while **PTR1** is not NULL.
  - a. Take an integer variable **VAL = PTR1 → DATA**.
  - b. If **(K - VAL)** exists in **MYMAP**, then return true.
  - c. Else, since we found a **VAL**, we will mark its frequency as one in the map, as **MYMAP[VAL] = 1**
  - d. Move **PTR1** ahead to its next to further the iteration.
5. If no such pair is found, return false.

**Time Complexity:** **O(N)**, where **N** is the length of the double linked list.

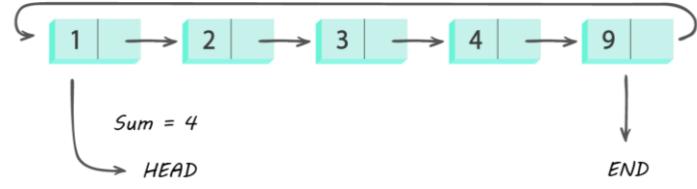
**Space Complexity:** **O(N)**, where **N** is the length of the double linked list. Since we have used the unordered\_map to store **N** elements, the space complexity is **O(N)**.

#### Approach 3: Two Pointer

As the given linked list is sorted so we can use two pointer techniques to find the pairs in **O(N)** time complexity.

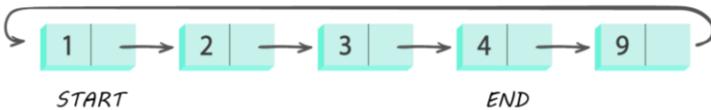
#### Steps:

1. We will take two pointer variables to find the candidate elements in the sorted doubly linked list.
2. Initialise **START = HEAD**, and **END** will be the last element of the linked list. We will find it by traversing the linked list.
3. If the sum of data at **START** and **END** is less than **K**, then we move **START** in forward direction.
4. For each **START and END**, if their sum is equal to **K**, return true.
5. If the current sum of data at **START** and **END** element is greater than **K**, we move **END** in the backward direction.
6. The loop will terminate when either of the two pointers become NULL, or they become the same (**START == END**).
7. If no such pairs are found, return false.



Step 1 :  $START + END = 1 + 9 = 10 > K(4)$

$END = END - 1$



Step 2 :  $START + END = 1 + 4 = 5 > K(4)$

$END = END - 1$



Step 3 :  $START + END = 1 + 3 = 4 = K$

Return True

**Time Complexity:**  $O(N)$ , where  $N$  denotes the length of the doubly linked list.

**Space Complexity:**  $O(1)$ , as we are using constant memory.

### 3. Count Triplets in a Sorted Doubly Linked List Whose Sum is Equal to a Given Value X [<https://coding.ninja/P53>]

**Problem Statement:** You are given a sorted doubly linked list of distinct nodes that means no two nodes present in the list have the same data. You are also given an integer  $X$ . Your task is to count the number of triplets in the list that sum up to a given value  $X$ .

For example, the given doubly linked list is  $1 \leftrightarrow 2 \leftrightarrow 3 \leftrightarrow 4 \leftrightarrow \text{NULL}$ , and the given value of  $X$  is 9. Here the number of triplets which have a sum of 9 is only one, and that triplet is (2, 3, 4).

#### Note:

If no such triplets exist, return zero. At least three elements will be present in the linked list.

### **Input Format:**

The first line of input contains an integer **T** denoting the number of test cases.

The next **2 \* T** lines represent the **T** test cases.

The **first line** of each test case contains space-separated integers denoting the nodes of the linked list. Each line is guaranteed to have -1 at the end to signify the end of the linked list.

The **second line** of each test case contains a single integer **X** denoting the value of the triplet sum.

### **Sample Input:**

1

1 2 3 8 9 -1

13

### **Sample Output:**

2

### **Explanation:**

For the given test case, the linked list is  $1 \leftrightarrow 2 \leftrightarrow 3 \leftrightarrow 8 \leftrightarrow 9 \leftrightarrow \text{NULL}$ .

We can clearly see that **two** triplets exist for this case—(2, 3, 8) and (1, 3, 9).

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

The idea is to traverse the linked list using three nested loops; thus, each possible combination is tried to see if it sums up to the value **X** and if it does increment the counter for the result.

### **Steps:**

1. Initialise the **COUNT** to zero.
2. For the first loop, initialise the pointer **PTR1** to the head of the given linked list, **HEAD**.

3. For the next nested loop, we initialise the pointer **PTR2** to the next loop pointer—that is, **PTR1→NEXT**.
4. Similarly, in the third loop, we initialise the pointer **PTR3** to the next of the second loop pointer—that is, **PTR2→NEXT**.
5. For each iteration in the last loop, check whether elements in the triplet sum up to X or not.
6. If the sum corresponds to X, which means if **(PTR1→DATA + PTR2→DATA + PTR3→DATA) = X**, increase the **COUNT**.
7. Else do nothing and move forward to the next iteration.

**Time Complexity:**  $O(N^3)$ , where **N** denotes the number of elements in the linked list. Since we are using three nested loops, hence time complexity will be  $O(N^3)$ .

**Space Complexity:**  $O(1)$ , we are using constant space.

### Approach 2: Using HashMap

The idea is to create a hashmap with (key, value) pair in which the key will be the **DLLNODE.DATA** (node of DLL class) and value will be a pointer to that node. We will traverse the doubly linked list and store each node's data and its pointer pair in the hash map and check whether  $(X - PAIRSUM)$  exists in the hash map or not. If it exists, then we will first verify that the two nodes in the pair are not the same as the node associated with  $(X - PAIRSUM)$  in the hash table. Finally, return **count/3** as the answer as each triplet is counted three times in the above process.

#### Steps:

1. Initialise the **COUNT** to zero.
2. Now initialise the hashmap as **MAP**, which stores the key as a **DLLNODE.DATA** and value as a pointer to that node.
3. Insert the pair in a hashmap by iterating the loop from head to null.
4. Further, iterate two nested loops in which the first loop iterates to head to null and the second nested loop iterates from pointer to next to null.
5. Store the **PAIRSUM**, which stores the sum of data of two pointers—that is, **PTR1.DATA + PTR2.DATA**.
6. Check if **MAP** contains the **(X - PAIRSUM) pair** and check if the two nodes in the pair are not the same as the node associated with **(X - PAIRSUM)** in the hashmap.
7. If the condition satisfies, increment the **COUNT**.

- Finally, return **COUNT/3** as each triplet is counted three times in the above process.

**Time Complexity:**  $O(N^2)$ , where **N** denotes the number of elements in the linked list. Since we are using two nested loops for generating all the possible pairs, hence time complexity is  $O(N^2)$ .

**Space Complexity:**  $O(N)$ , as we are using a HashMap to store all the key-value pairs.

### Approach 3: Using Two Pointers

The idea is to have two pointers in which we iterate through the linked list, checking each value in the **COUNTPAIRS** function. The **COUNTPAIRS** function will count the number of pairs whose sum equals the given value and then traverses the doubly linked list from left to right.

#### Steps:

- Initialise a variable **COUNT** to zero.
- Get the **LAST** pointer to the last node of the doubly linked list.
- Traverse the linked list, using a pointer, say **CURR**.
- For each current node during the traversal, initialise two pointers: **FIRST**, which points to the node next to the current node—that is, **CURRENT→NEXT** and **LAST**, that points to the last node of the list.
- Find such pairs in the linked list such that **CURR + FIRST + LAST = K**.
- Declare a variable **VALUE** such that **VALUE = K - CURR**.
- Thus, if we find pairs such that **FIRST + LAST = VALUE**, we find a triplet whose sum is **K** (**FIRST + LAST + CURR**).
- To calculate the pairs that satisfy the constraint, initialise a count to zero, let us name it **COUNTP**.
- Run a loop that terminates when either of two pointers become null or cross each other—that is, **SECOND→NEXT == FIRST**, or they become the same (**FIRST == SECOND**).
- Inside this loop, check for pairs such that **FIRST→DATA + SECOND→DATA equals VALUE**.
  - Increment the pointer **FIRST** to forward direction **FIRST = FIRST→NEXT** and **SECOND** in backward direction **SECOND = SECOND→PREV**.
- Else if **FIRST→DATA + SECOND→DATA** is greater than **VALUE**, shift the second pointer backwards, **SECOND = SECOND→PREV**.

13. Else if **FIRST→DATA + SECOND→DATA** is lesser than **VALUE**, shift first pointer ahead, **FIRST = FIRST→NEXT**.
14. After the loop is finished, **COUNTP** holds the number of pairs corresponding to a triplet we need to find.
15. This **COUNTP** will be added to **COUNT**.
16. After the traversal over the entire list is complete, we will have the number of triplets in variable **COUNT**.

**Time Complexity:**  $O(N^2)$ , where **N** denotes the number of elements in the linked list. Since we are using two nested loops in the **COUNTPAIRS** function and another one for calling the **COUNTPAIRS** function for every iteration.

**Space Complexity:**  $O(1)$ , we are using constant space.

#### **4. Deletion in Circular Linked List** [<https://coding.ninja/P54>]

**Problem Statement:** You are given a circular linked list of integers and an integer **key**. You have to write a function that finds the given **key** in the list and deletes it. If no such key is present, then the list remains unchanged.

##### **Note:**

1. The integer **-1** marks the end of the linked list; however, the tail of the linked list would be pointing to the head, making it circular.
2. The circular linked list before/after deletion may happen to be empty. In that case, only print **-1**.
3. All integers in the list are unique.

##### **Input Format:**

The **first input** line contains the integers present in the circular linked list in order.

The **second input** line contains a single integer **key** which is to be deleted.

##### **Output Format:**

The single output line contains the updated circular linked list post deletion.

### Sample Input:

1 2 3 4 5 -1

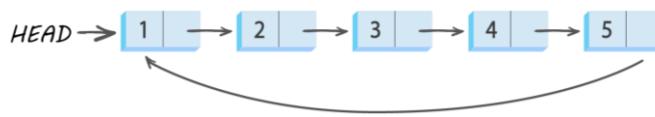
3

### Sample Output:

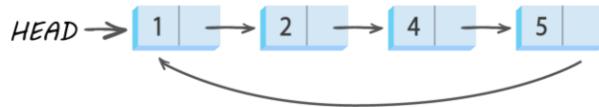
1 2 4 5 -1

### Explanation:

The given linked list before deletion:



After deletion:



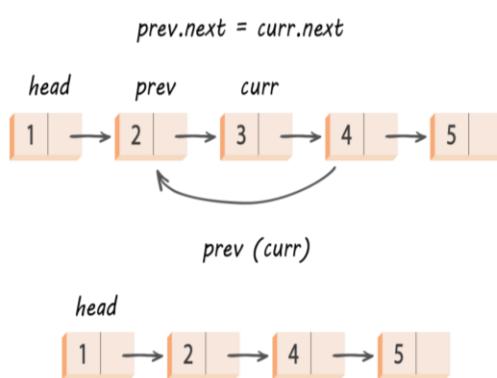
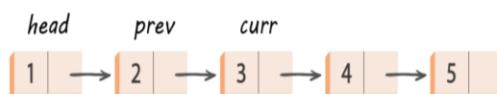
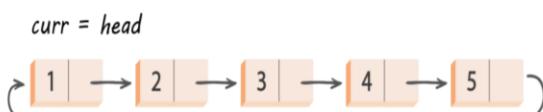
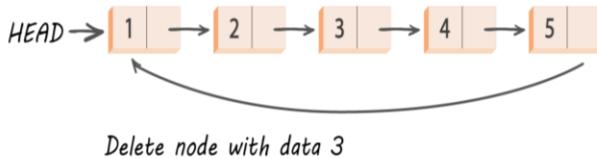
### Approach 1: Curr And Prev Approach

In this approach, we are basically going to use two pointers to iterate over our linked list. We will begin traversal and follow the following steps.

### Steps:

1. If the list is empty, return.
2. Else, we define two pointers, **curr** and **prev** and initialise the pointer **curr** with the **head** node.
3. Traverse the circular linked list using **curr** to find the node to be deleted, and before moving **curr** to the next node every time, set **prev = curr**, so that previous will be on one node behind current.
4. If the node is found, check if it is the only node in the circular linked list. If yes, do **head = NULL**.
5. If the circular linked list has more than one node, check if it is the first node of the list. To confirm this, check **(curr == head)**. If yes:
  - a. Move **prev** pointer until it reaches the last node.

- b. After the **prev** pointer reaches the last node, set **head = head→next** and **prev→next = head**
  - c. Delete the node **curr** by applying **free(curr)**.
6. If **curr** is not the first node, we check if it is the last node in the list. Check if **curr→next == head**. If **curr** is the last node. Do **prev→next = head** and delete the node **curr** by applying **free(curr)**.
  7. If the node to be deleted is neither the first node nor the last node, then set **prev→next = curr.next** and delete the node **curr** by applying **free(curr)**



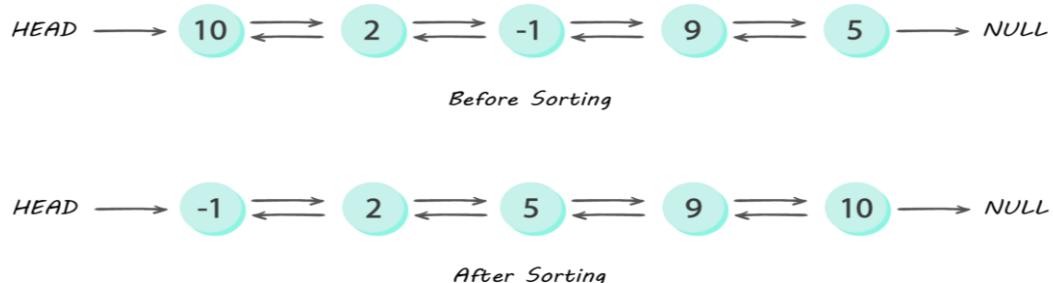
**Time Complexity:** **O(N)**, where **N** denotes the number of nodes in the linked list. In the worst case, we have to traverse the whole list to find the node to be deleted; time complexity is **O(N)**.

**Space Complexity:** **O(1)**, since constant space is utilised.

## 5. QuickSort on Doubly Linked List [<https://coding.ninja/P55>]

**Problem Statement:** You are given the head of a doubly linked list containing **N** nodes. Each node will have an integer value stored in it. You need to return the head of the doubly linked list after sorting it using the **QuickSort** algorithm.

**For example:**



**Input Format:**

The **first line** of the input contains an integer **T** denoting the number of test cases.

The **first line** of each test case contains a single positive integer **N** denoting the number of the element nodes present in the doubly linked list.

The **second line** of each test case contains **N** space-separated integers denoting the elements of the double linked list.

**Output Format:**

The only line of output of each test case should contain **N** space-separated integer in the sorted order.

**Sample Input:**

1

4

4 2 -3 4

**Sample Output:**

-3 2 4 4

### **Explanation:**

After sorting the list  $[4 \leftrightarrow 2 \leftrightarrow -3 \leftrightarrow 4]$  will look like  $[-3 \leftrightarrow 2 \leftrightarrow 4 \leftrightarrow 4]$ .

### **Approach 1: QuickSort**

The algorithm will remain the same as for arrays.

### **Steps:**

1. Choose a pivot as the last node in our doubly linked list and define its position in a sorted linked list in linear time. Pivot element is the one whose correct position is located in the list to decide partitioning.
2. We will place all the elements smaller than pivot to the left of the pivot, and all the elements greater than pivot will remain at the right side of the pivot.
3. We traverse the list and maintain a pointer that is pointing at the starting of the list.
4. If the current value is smaller than the pivot, then:
  - a. Swap the values present at the current node to the starting node pointer
  - b. Increment the pointer a step forward.
5. Finally, swap values present at the pointer and pivot node. In this way, the pivot will go to its correct position.
6. Now the array is divided into two halves around the pivot.
7. Sort them separately by using the same process again using recursion.

**Time Complexity:**  $O(N^2)$ , where  $N$  denotes the length of the doubly linked list. In each recursive call, we do  $O(N)$  operations. And in the worst case, we may have  $N$  recursive calls in the stack, which will make the time complexity to be  $O(N^2)$ .

**Space Complexity:**  $O(N)$ , where  $N$  denotes the length of the doubly linked list. In the worst case, the recursive calls would be made  $N$  times; hence space complexity is  $O(N)$ .

## **6. Reverse DLL Nodes in Groups**

[<https://coding.ninja/P56>]

**Problem Statement:** You are given a doubly linked list of integers and a positive integer  $K$  representing the group size. Modify the linked list by reversing every group of  $K$  nodes in the linked list.

### **Input Format:**

The **first line** of the input contains an integer **T** denoting the number of test cases.

The **first line** of every test case contains the elements of the singly linked list separated by a single space and terminated by -1. Hence, -1 would never be a list element.

The **second line** of every test case contains the positive integer **K**.

### **Output Format:**

For every test case, print the modified, linked list. The elements of the modified list should be single-space separated and terminated by -1.

### **Sample Input:**

```
1
1 2 3 4 5 6 7 -1
2
```

### **Sample Output:**

```
2 1 4 3 6 5 7 -1
```

### **Explanation:**

In the given linked list, we have to reverse the first two nodes, then reverse the next two nodes, and so on until all the nodes are processed in the same way. The modified linked list after the above process is 2 1 4 3 6 5 7.

### **Approach 1: Recursion**

We will process **K** nodes at a time. Firstly, we will reverse the first **K** nodes of the doubly linked list, and then we will do this recursively for the remaining linked list.

### **Steps:**

1. If the current node does not exist, then return NULL.
2. If there are less than **K** nodes, reverse them and return the reversed linked list.
3. Else reverse the first **K** nodes of the linked list.

4. After reversing the  $K$  nodes, the head points to the  $K^{\text{th}}$  node and the  $K^{\text{th}}$  node in the original linked list will become the new head.
5. To connect the reversed part with the remaining part of the linked list, update the next of the head to  $(K + 1)^{\text{th}}$  node and the previous of the  $(K + 1)^{\text{th}}$  node to the head node.
6. Now, recursively modify the rest of the linked list and link the two sub-linked lists.
7. Return the new head of the linked list.

**Time Complexity:**  $O(N)$ , where  $N$  denotes the number of nodes in the linked list.

**Space Complexity:**  $O(N/K)$ , where  $N$  denotes the number of nodes in the linked list and  $K$  is the given positive integer. In each recursive step, we are processing  $K$  nodes of the linked list. Thus,  $O(N/K)$  is the total recursion stack space used by the algorithm.

### Approach 2: Iterative Approach

The idea is the same as used in the previous approach. This time, we will do it iteratively.

#### Steps:

1. Head is pointing to the first node of the linked list.
2. Initialise a pointer to a Node **newHead** that will point to the head of the final modified linked list.
3. Repeat the steps below until all the nodes are processed in the same way.
  - a. Reverse the first  $K$  nodes of the linked list.
  - b. After reversing the  $K$  nodes, the head points to the  $K^{\text{th}}$  node.
  - c. If the **newHead** is NULL,
    - i. If the  $K^{\text{th}}$  node in the original linked list exists, it will be the **newHead**.
    - ii. Else, the last node in the original linked list will be the **newHead**.
  - d. To connect the reversed part with the remaining part of the linked list, update the next of the head to  $(K + 1)^{\text{th}}$  node and the previous of the  $(K + 1)^{\text{th}}$  node to the head node.
4. Return the new head of the linked list.

**Time Complexity:**  $O(N)$ , where  $N$  is the length of the double linked list.

**Space Complexity:**  $O(1)$ , as constant memory is utilised.

## 7. Count Inversion [<https://coding.ninja/P57>]

**Problem Statement:** Given a singly linked list of integers, return its inversion count. The inversion count of a linked list is a measure of how far it is from being sorted.

Two nodes **n1** and **n2**, form an inversion if the value of **n1** is greater than the value of **n2** and **n1** appears before **n2** in the linked list.

### **For Example:**

If we have a list like [3 2 1 5 4], after sorting, it will look like [1 2 3 4 5]. To achieve this sorted list, we need to make 4 inversions. They are -

1. [ 2 3 1 5 4 ] - We inverted (3, 2)
2. [ 2 1 3 5 4 ] - We inverted (3, 1)
3. [ 1 2 3 5 4 ] - We inverted (2, 1)
4. [ 1 2 3 4 5 ] - We inverted (5, 4)

### **Input Format:**

The **first and the only line** of the input contains the elements of the singly linked list separated by a single space and terminated by -1. Hence, -1 would never be a list element.

### **Output Format:**

A single integer denoting the inversion count of the given list.

### **Sample Input:**

3 2 1 5 4 -1

### **Sample Output:**

4

### **Explanation:**

For the given linked list, there are 4 inversions: (3, 2), (3, 1), (2, 1), and (5, 4). Thus, the inversion count is 4.

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

We use a pointer **cur**, initialised to the head of the linked list, to traverse the entire list. We also use a pointer **temp** to traverse the part of the linked list after **cur**. We use the **temp** pointer to start traversal from the node next to the **cur** for every node pointed by the **cur**. We compare the

values of the nodes pointed by the **temp** and the **cur** pointer. If the value pointed by **cur** is greater than that pointed by **temp**, then the two nodes form an inversion pair.

We repeat the entire process till **cur** reaches the end of the list.

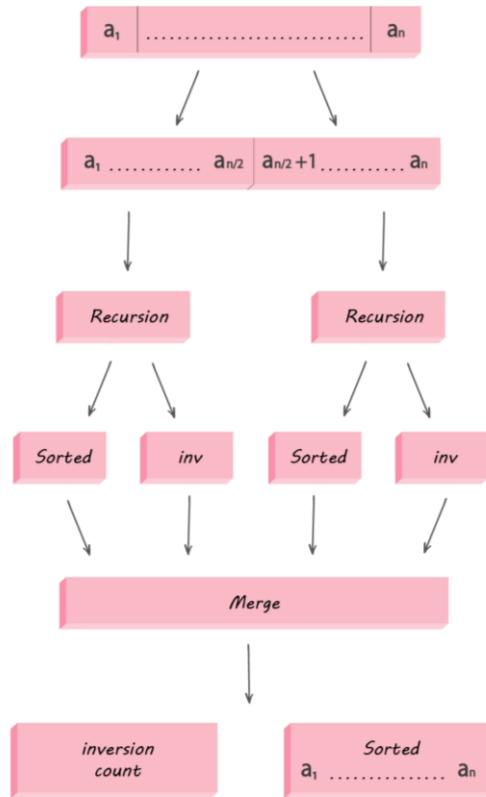
**Time Complexity:**  $O(N^2)$ , where **N** is the number of nodes in the linked list. For each node, we are iterating through the linked list once in  $O(N)$  time, and there are **N** nodes; thus, the total time complexity is  $O(N * N) = O(N^2)$ .

**Space Complexity:**  $O(1)$ , as constant space is required for this problem.

### Approach 2: Using Merge Sort

This approach is based on the concept of merge sort, and we count the number of inversions in the list during the merge step. We will have two sorted lists—left and right. We need to merge them and concurrently count the inversions. We will keep track of the current position in each list. Initially, this position is 0 (denoting the beginning of the list), and it can also point after the end of the list.

1. If both the current positions are inside their lists, then we have two cases:
  - a. The left current element is less than or equal to the right current element: We copy the left current element to the resulting list and increment the left position.
  - b. The left current element is greater than the right current element: We copy the right current element into the resulting list and increment both the right position pointer and the inversions count by the number of remaining elements in the left list (including the left current element).
2. If there are no left elements present anymore, then we append the remaining right list to the resulting list.
3. Otherwise, we append the remaining left list to the resulting list.



**Time Complexity:**  $O(N * \log_2(N))$ , where  $N$  denotes the number of nodes in the linked list. The algorithm used (Merge Sort) is divide and conquer. Thus, for each traversal, we divide the array into two parts; thus, there are  $\log_2(N)$  levels. Hence, the final complexity is  $O(N * \log_2(N))$ .

**Space Complexity:**  $O(\log_2(N))$ , as the algorithm is recursive and there are  $\log_2(N)$  levels, it requires  $O(\log_2(N))$  stack space.

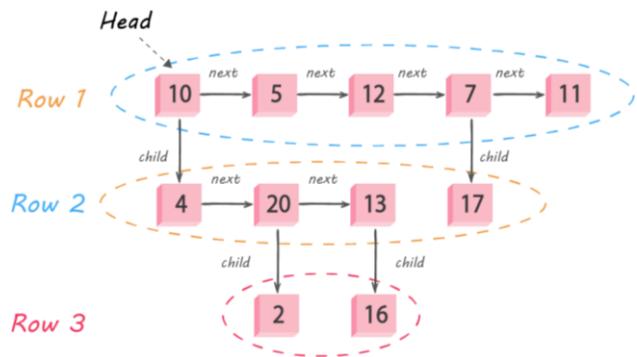
## 8. Flatten The Multi-Level Linked List [\[https://coding.ninja/P58\]](https://coding.ninja/P58)

**Problem Statement:** You are given a multi-level linked list of  $N$  nodes; each node has a next and child pointer which may or may not point to a separate node. Flatten the multi-level linked list into a singly linked list. You need to return the head of the updated linked list.

### Note:

Flattening a multi-level linked list means merging all the different rows into a single row.

### Example

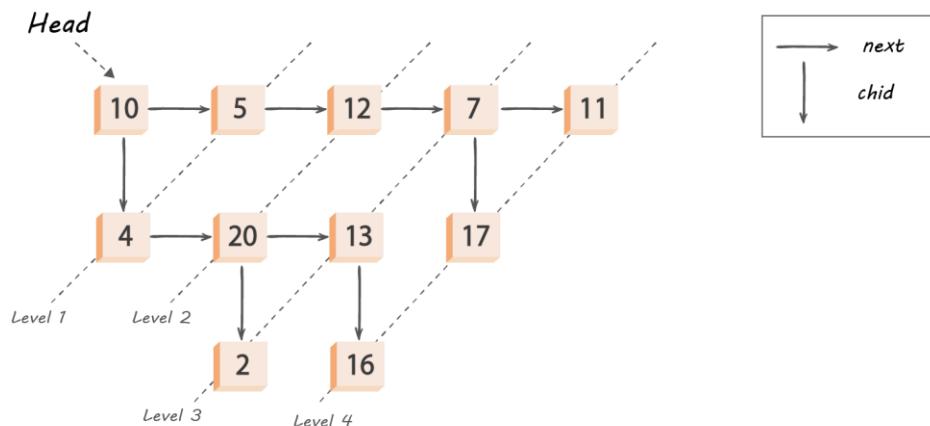


*Input : 10 5 4 12 -1 20 -1 7 -1 13 2 11 17 1 16 -1 -1 -1 -1 -1 -1 -1*

*Output : 10 5 12 7 11 4 20 13 17 2 16*

### Input :

The **first line** of input contains the elements of the multi-level linked list in the level order form. The line consists of values of nodes separated by a single space. In case a node (next or child pointer) is null, we take -1 in its place. For example, the input for the multi-level linked list depicted in the below image would be:



10 5 4 12 -1 20 -1 7 -1 13 2 11 17 -1 16 -1 -1 -1 -1 -1 -1 -1

### Explanation :

**Level 0:** The head node of the multi-level linked list is 10.

**Level 1:** Next pointer of 10 = 5; Child pointer of 10 = 4

**Level 2:** Next pointer of 5 = 12; Child pointer of 5 = null (-1)

Next pointer of 4 = 20; Child pointer of 4 = null (-1)

**Level 3:** Next pointer of 12= 7; Child pointer of 12 = null (-1)

Next pointer of 20 = 13; Child pointer of 20 = 2

**Level 4:** Next pointer of 7 = 11; Child pointer of 7 = 17

Next pointer of 13 = null (-1); Child pointer of 13 = 16

Next pointer of 2 = null (-1); Child pointer of 2 = null (-1)

The first not-null node (of the previous level) is treated as the parent of the first two nodes of the current level (next and child pointers). The second not-null node (of the previous level) is treated as the parent node for the next two nodes of the current level and so on. The input ends when all nodes at the last level are null (-1).

#### Sample Input:

1 -1 2 -1 3 -1 4 -1 -1

#### Sample Output :

1 2 3 4

#### Explanation:

The given multi-level linked list will be represented as shown below.



*Input : 1-1 2-1 3-1 4-1 -1*

*Output : 1 2 3 4*

#### Approach: Using Tail Pointer

The naive solution is to maintain a pointer **tail** that initially points to the end of level 0 (the level that contains head). Make necessary connections so that all nodes are traversed through the tail via the next parameter only. Take another pointer **curr** and traverse the list till the end of the list(null). Make a new connection—that is, **tail** → **next** = **curr** → **child** (append current row head to previous row tail). Also, maintain a pointer **temp**, which maintains the tail of the flatten linked list.

### **Steps:**

1. If the **head** is null, return the **head**.
2. Initialise a **tail** node with the head and iterate till the end of the first row so that the tail node will store the end node of each row processed.
3. Initialise two nodes **curr = head** and **temp** = null, where **curr** is used to traverse the whole list and temp is used to update the tail.
4. While **curr** is not null,
  - a. If the **curr** node has a child node,
    - i. then we will append this child to the end of the list. Now the next row will be attached to the end of the previous row.
    - ii. Using the temp node iterates to the end of this row until the next node is null.
    - iii. Update the tail node with the temp, i.e., the new tail of the flattened list.
  - b. Shift the **curr** to its next node—that is, **curr = curr→next**

**Time Complexity:**  $O(N)$ , where **N** denotes the number of nodes in the multi-level linked list. As every node of the multi-level linked list can be visited up to two times.

**Space Complexity:**  $O(1)$ , only constant space is required.

# 7. Trees

---

## Introduction to Trees

A tree is a data structure similar to linked lists, but instead of each node pointing to a single next node in a linear fashion, each node points to several nodes. The tree is a **non-linear** data structure. A tree structure is a way of representing a hierarchical nature of a structure in graphical form.

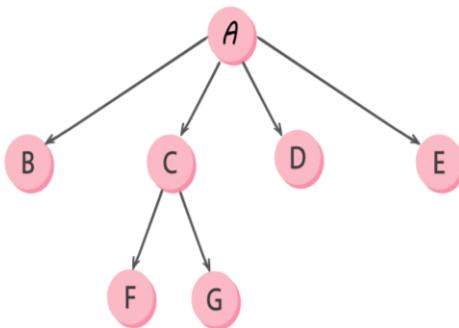


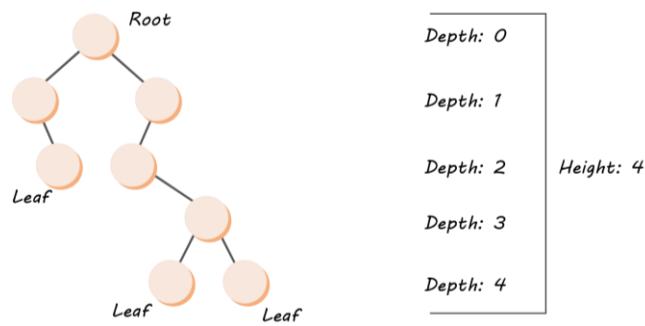
Figure 1

## Properties of Trees

- **Root:** The root node of the tree is the node with no parents. There can be at most one root node in the tree. **A** is the root node in the example shown in figure 1.
- **Parent:** For a given node, its immediate predecessor is known as its parent node. In other words, nodes having one or more children are parent nodes. Here **A** is the parent node of **B**, **C**, **D**, and **E**. Similarly, **C** is the parent node of **F** and **G**.
- **Edge:** An edge refers to the link from the parent node to the child node.
- **Leaf:** Nodes with no children are called leaf nodes. **B**, **F**, **G**, **D**, and **E** are leaf nodes in the given example.
- **Siblings:** Children with the same parent are called siblings. Here **B**, **C**, **D**, and **E** are siblings. Similarly, **F** and **G** are siblings.
- **Ancestors:** A node **x** is an ancestor of node **y** if there exists a path from the root to node **y** such that **x** appears on the path. For example, **A** is an ancestor of nodes **F** and **G**.

- **Descendants:** A descendant is the inverse relationship of an ancestor. Since **x** is the ancestor of node **y**, node **y** is treated as the descendant of node **x**. For example, **F** and **G** are the descendants of node **A**.
- **Depth:** The depth of a node in the tree is the length of the path from the root to that node. The depth of the tree is the maximum depth among all the nodes in the tree. The depth of the tree in figure 2 is 4.

Figure 2



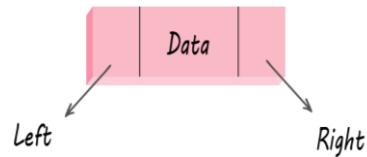
- **Height:** The node's height is the length of the path from that node to the deepest node in the tree. The tree's height would be the length of the path from the root node to the deepest node.
  - The height of the tree in figure 2 is four. We always count the edges and not the nodes while calculating the height or depth.
  - A tree with only **one** node has **zero** height.
  - For a given tree, depth and height return the same value; however, they may differ for individual nodes.
- **Skewed Trees:** If every node in a tree has only one child, we call such a tree a skewed tree.
  - If a tree's each node has only a left child, we call it a **left-skewed tree**. If a tree's each node has only a right child, we call it a **right-skewed tree**.



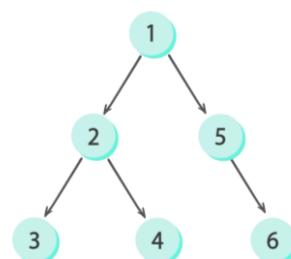
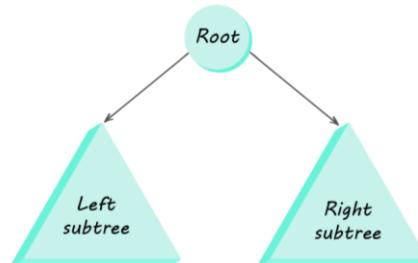
LEFT SKEWED      RIGHT SKEWED

## Binary Trees

A tree is called a **binary tree** if every node in the tree has zero, one, or two children. An empty binary tree is also a valid binary tree. A binary tree is made of nodes that constitute a left pointer, a right pointer, and a data element.

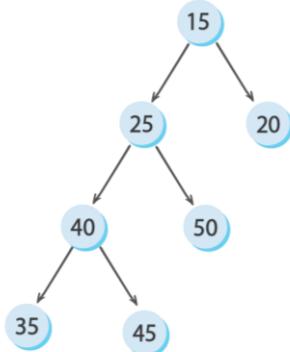


We can visualise a binary tree as the **root node** and **two disjoint binary trees**, called the **left subtree** and the **right subtree**.

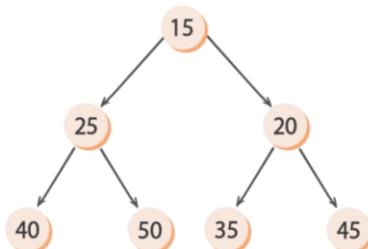


## Types of Binary Trees

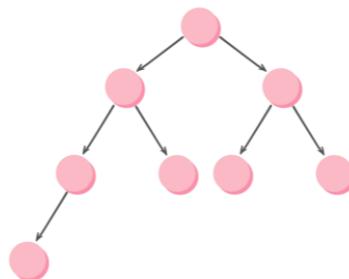
- **Strict Binary Tree:** A binary tree in which each node has exactly **zero or two children**.



- **Full Binary Tree:** A binary tree in which each node has **exactly two children** and all the leaf nodes are at the **same level**.

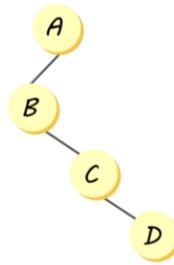


- **Complete Binary Tree:** A complete binary tree has all the levels filled except for the last level, which has all its nodes as far left as possible.



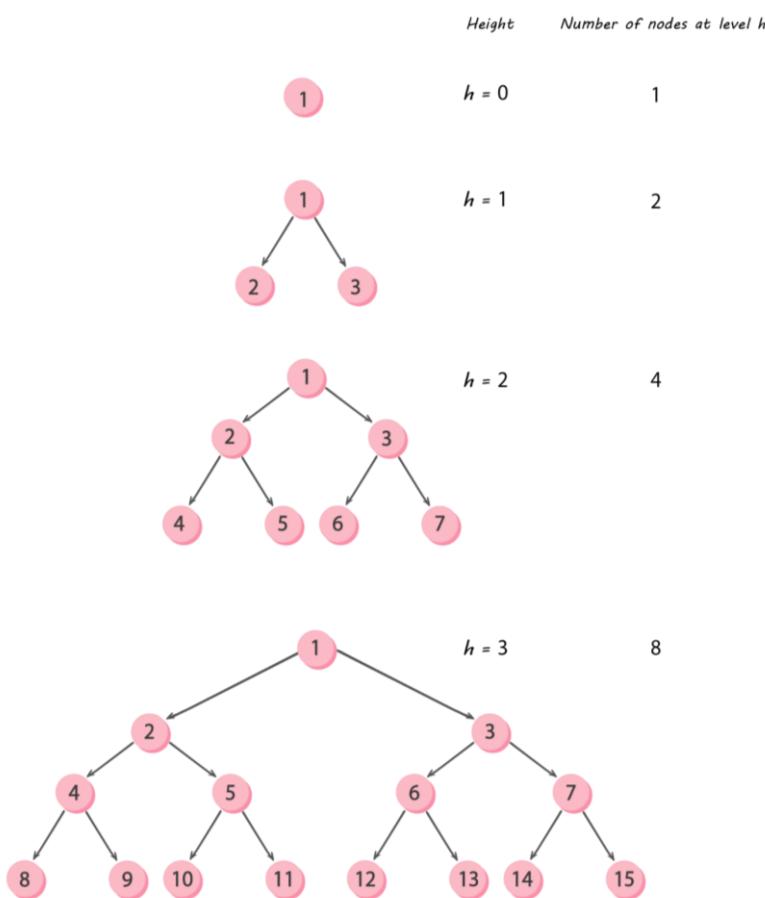
*Complete tree*

- **Degenerate Tree:** Each internal node has only one child in a degenerate tree, either on the left or right side.



## Properties of a Binary Tree

Let's assume that the tree has a height of  $h$ . The root node is at the height of 0.



From the diagram, we can say that

- The number of nodes in a full binary tree is  $2^{(h+1)} - 1$ .
- The number of leaf nodes in a full binary tree is  $2^h$ .
- The number of node links in a complete binary tree of  $n$  nodes is  $n - 1$ .

## Binary Tree Traversals

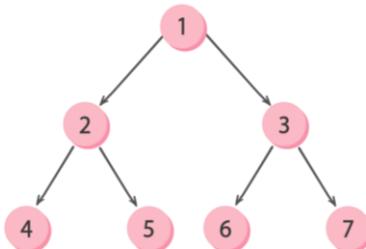
The process of visiting all the nodes of the tree is called **tree traversal**. Each node is processed only once but may be visited more than once. We will be considering the below tree for all the traversals.

We will also use the following notations to represent the different parts of a binary tree:

**N**: Current node

**L**: Left Subtree

**R**: Right Subtree



### Preorder Traversal (NLR)

In preorder traversal, the root node is visited first, then the left node, and then the right node.

In the preorder traversal of a tree, **each node is processed before processing its subtrees**.

Like in the above example, 1 is processed first, then the left subtree, followed by the right subtree. Therefore, processing must return to the right subtree after finishing the processing of the left subtree. To move to the right subtree after processing the left subtree, we must maintain the root information.

#### Steps for preorder traversal:

- Visit the root.
- Traverse the left subtree in preorder—that is, call `preOrderTraversal(left.subtree)`.
- Traverse the right subtree in preorder—that is, call `preOrderTraversal(right.subtree)`.

```
function preOrderTraversal(root)
```

```

if root is null
    return

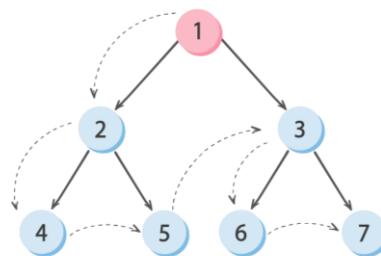
    // process current node
    print "root.data"

    // calling function recursively for left and right subtrees
    preOrderTraversal(root.left)
    preOrderTraversal(root.right)

```

**Preorder output of above tree: 1 2 4 5 3 6 7**

The recursive traversal takes place like this:



### InOrder Traversal (LNR)

The left subtree is visited first in inorder traversal, then the root and later the right subtree.

#### Steps for inorder traversal:

- Traverse the left subtree in Inorder.
- Visit the root.
- Traverse the right subtree in Inorder.

```

function inOrderTraversal(root)
    if root is null
        return

    // calling function recursively for left subtree
    inOrderTraversal(root.left)

```

```

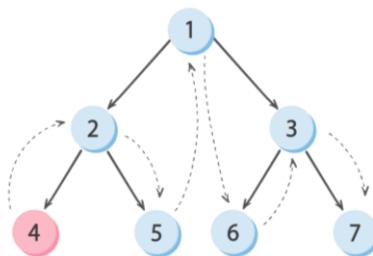
// process current node
print "root.data"

// calling function recursively for right subtree
inOrderTraversal(root.right)

```

**Inorder output of above tree: 4 2 5 1 6 3 7**

The recursive traversal takes place like this:



### PostOrder Traversal(LRN)

In postorder traversal, the root is visited after the left and right subtrees—that is, we traverse the left subtree, then the right subtree, and finally the root node.

#### Steps for postorder traversal

- Traverse the left subtree in postorder.
- Traverse the right subtree in postorder.
- Visit the root.

```

function postOrderTraversal(root)

    if root is null
        return

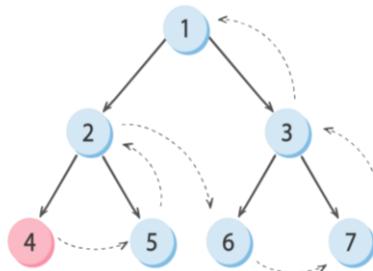
    // calling function recursively for left and right subtrees
    postOrderTraversal(root.left)
    postOrderTraversal(root.right)

```

```
// process current node  
print "root.data"
```

**Postorder output of above tree: 4 5 2 6 7 3 1**

The recursive traversal looks like this:



### LevelOrder Traversal

Level-order traversal is a breadth-first traversal, in which all children of a node are visited before any of the children's children is visited. Each node is visited level-wise in increasing order of levels from left to right.

#### Steps for levelOrder traversal

- Visit the root.
- While traversing **level I**, add all the elements at **(I + 1)** in the queue.
- Go to the next level and visit all the nodes at that level.
- Repeat this until all levels are completed.

```
function levelOrderTraversal()  
    If root is null  
        return  
  
    // Create queue of type Node and add root to the queue  
    queue.add(root).  
  
    while the queue is not empty
```

```

// remove the front item from the queue
removedNode= queue.remove()

// process the removedNode
print "removeNode.data"

/*
    Add the left and right child of the removedNode if they are not null
*/

if removedNode.left is not null
    queue.add(removedNode.left)

if removedNode.right is not null
    queue.add(removedNode.right)

end

```

**LevelOrder output of above tree: 1 2 3 4 5 6 7**

## Some Operations on a Binary tree

Following are the main operations done on a binary tree

- Count nodes in a binary tree
- Find the maximum value node in a binary tree
- Search for a value in a binary tree
- Find the height of a binary tree

Let us discuss them one by one below:

### Count nodes in a binary tree

Given a binary tree, calculate the total number of nodes in the tree.

```
function countNodesInABinaryTree(root)

    if the root is null
        return 0

    /*
        Count nodes in the left subtree and right subtree recursively
        and add 1 for self node
    */

    leftCount = countNodesInABinaryTree(root.left)
    rightCount = countNodesInABinaryTree(root.right)

    return leftCount + rightCount + 1
```

### Find maximum value in a binary tree:

Given a binary tree, calculate the node with the maximum value in the tree.

```
function findMax(root)

    if root is null
        return Integer.MIN_VALUE

    /*
        Find Maximum in the left subtree and right subtree recursively
        and compare self data, max of left subtree, max of right
        subtree and return max of these three
    */

    *
```

```

max = root.data

leftMax = findMax(root.left)

rightMax = findMax(root.right)

if leftMax > max

    max = leftMax

if rightMax > max

    max = rightMax

return max

```

### **Search for a value in a binary tree.**

Given an integer value, Search for the node with the value in the given tree.

```

function search(root, val)

    if the root is null

        return false

    /*

        If the current node's data is equal to val then return true

        Else call the function for left and right subtree.

    */

    if root.data equals val

        return true

    return search(root.left, val) or search(root.right, val)

```

Height of a binary tree.

Find the height of the given binary tree.

```
function findHeight(root)
    if root is null
        return 0

    /*
        Find the height of the left subtree and right subtree.
        Then add 1 (for current node) to max value from leftHeight
        and rightHeight
    */

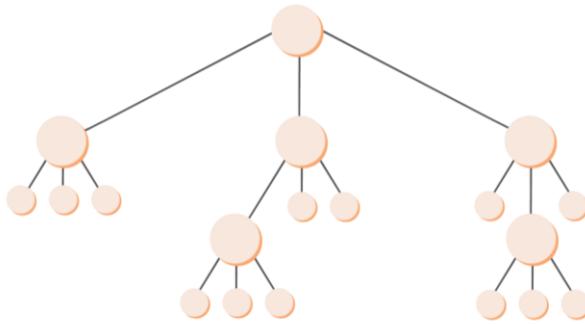
    leftHeight = findHeight(root.left)
    rightHeight = findHeight(root.right)

    return 1 + max(leftHeight, rightHeight)
```

## Time Complexity of Various Operations

Operations	Time Complexity
preOrderTraversal()	$O(n)$
inOrderTraversal()	$O(n)$
postOrderTraversal()	$O(n)$
levelOrderTraversal()	$O(n)$
findMax()	$O(n)$
search(val)	$O(n)$
countNodesInABinaryTree()	$O(n)$
findHeight()	$O(n)$

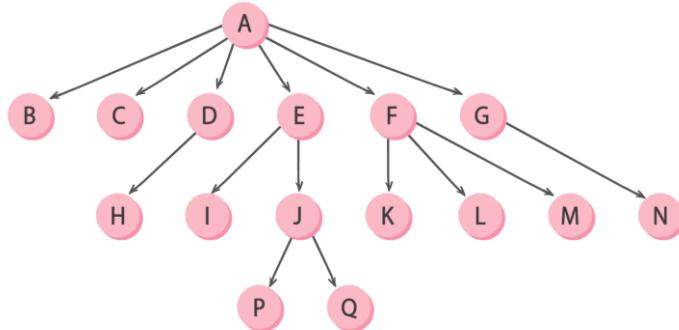
## Generic Trees (N-ary Trees)



**N-ary tree**, also known as a **generic tree**, is a tree data structure where any parent node can have at most **n** number of children. Here **n** is a non-negative integer.

A generic tree consists of a **list of pointers to the child nodes** and their respective **data** element. Similar to the binary trees, the **root** pointer is the topmost node in the tree.

### Printing a Generic Tree (recursively)



```

function printTree(root)

    // process current node
    print "root.data"

    // calling function recursively for childNodes

    for every childNode in the root.children list
        printTree(childNode)
    
```

**The output of the above tree: A B C D H E I J P Q F K L M G N**

### **LevelOrder Printing of Generic Tree**

#### **Steps for levelOrder traversal**

- Visit the root.
- While traversing **level l**, add all the elements at **(l + 1)** in the queue.
- Go to the next level and visit all the nodes at that level.
- Repeat this until all levels are completed.

```
function levelOrderTraversal()  
    If the root is null  
        return  
  
    // Create queue of type Node and add root to the queue  
    queue.add(root)  
  
    while the queue is not empty  
  
        // remove the front item from the queue  
        removedNode= queue.remove()  
  
        print "removedNode.data"  
  
        /*  
            Add the childNodes of the removedNode to the queue  
        */  
        for every childNode in removedNode.children list  
            queue.add(childNode)  
        end
```

**LevelOrder output of above tree: A B C D E F G H I J K L M N P Q**

## **Application of Trees**

Trees enable us to establish hierarchical relationships. For example, they are used in the implementation of file structures on a system.

A variety of trees are used to serve different purposes. Following are a few examples:

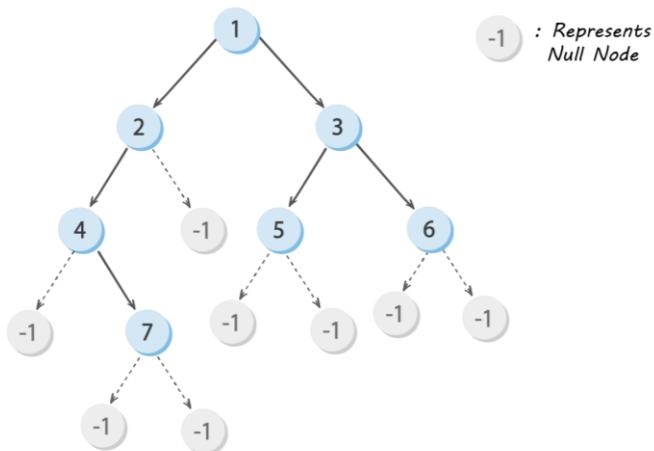
- Expression trees are used in compilers.
  - Huffman coding trees are used in compression algorithms, such as those used by the .jpeg and .mp3 file-formats.
  - Binary trees are used in almost every high-bandwidth router for storing router tables.
  - Heaps are used in implementing efficient priority queues.
- 

## Practice Problems

### ***Input Format for all the practice problems***

Every binary tree is represented in **level order form**.

For example, the input for the tree depicted in the below image would be :



**Note: Null nodes (-1) are for representation purposes only. They are not part of the actual tree.**

**Level 0:** 1

**Level 1:** 2 3

**Level 2:** 4 -1 5 6

**Level 3:** -1 7 -1 -1 -1 -1

**Level 4:** -1 -1

Thus the final input for the given tree can be 1 2 3 4 -1 5 6 -1 7 -1 -1 -1 -1 -1.

## 1. Maximum Width In Binary Tree [<https://coding.ninja/P59>]

**Problem Statement:** You have been given a **binary tree** of integers. You are supposed to return the maximum width of the given binary tree. The maximum width of the tree is the maximum width among all the levels of the given tree.

The width of one level is defined as the length between its leftmost and the rightmost non-null nodes. The null nodes in between the leftmost and rightmost are excluded into the calculation.

### Input Format:

The **first line** contains elements in the level order form. The line consists of values of nodes separated by a single space. In case of a null node, we take -1 in its place.

### Output Format:

Print the maximum width for the given tree.

### Sample Input:

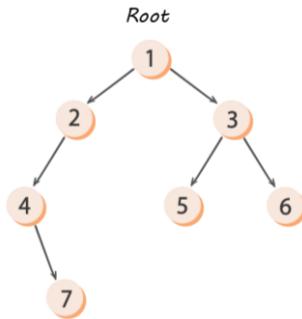
1 2 3 4 -1 5 6 -1 7 -1 -1 -1 -1 -1 -1

### Sample Output:

3

### Explanation:

For the given binary tree, the maximum width will be at the third level with the length of 3—that is, 4, 5, and 6.



### Approach 1: Brute Force

The straightforward intuition is to find the maximum levels possible in the given tree, called the **height** of the tree. And then, for each level for the given height, find the number of nodes. The maximum number of nodes among all levels will be the maximum width of the given binary tree. The implementation of our intuition takes below steps:

#### Steps:

1. Begin with finding the **height** of the tree.
  - a. If the root is **NULL**, return 0 as height is 0.
  - b. Else, call the recursive function for left and right subtrees. The maximum height among both would be the height of the subtree for the **root** node. So the height of the **root** node will be:
$$1 + \max(\text{leftHeight}, \text{rightHeight})$$
  - c. Return the height of the **root** node.
2. For each level from level 1 to height, find the number of nodes present in it. The maximum number of nodes among all levels will be the maximum width of the given tree. For finding the number of nodes at any level, let's say **level**, steps are as follows:
  - a. If the root is **NULL**, return 0 as there will be no node present.
  - b. If the **level** becomes 1, then return 1 as there will be only **root** node present.

- c. Else, call recursive function on left subtree with **level - 1** and store in **leftNodes** and call recursive on right subtree with **level - 1** and store in **rightNodes**. So the total number of nodes for the current level will be (**leftNodes + rightNodes**).
- 3. The maximum number of nodes among all levels will be the maximum width of the given tree.

**Time Complexity:**  $O(N \times H)$ , where **N** denotes the number of nodes in the given binary tree and **H** is the height of the binary tree. Height is found at a **cost  $O(H)$** . In the worst case (Skewed Trees), **H** is equal to **N**, and we will find the number of nodes in each level for each level in height. Therefore, the overall time complexity will be  $O(N \times H)$ .

**Space Complexity:**  $O(H)$ , where **H** denotes the height of the binary tree. In the worst case (Skewed Trees), **H** is equal to **N**. Therefore; overall space complexity would be  $O(H)$ .

### Approach 2: Using Depth First Traversal

In the previous approach, we used a recursive algorithm to get the total number of nodes on each level. That took  $O(N \times H)$ , where **N** is the number of nodes in the given binary tree and **H** is the height of the binary tree. In this approach, our intuition is to use any of the traversal methods among preorder, postorder, or inorder traversal and visit each element only once. While calling the selected order function, we will also pass the level corresponding to that node. When we visit any node belonging to the same level, we will increment the total number of nodes for that level.

#### Steps:

1. Use a **HashMap** to store the total number of nodes for each level, let's say **nodesAtLevel**. The key of **nodesAtLevel** will represent the level, and the value corresponding to that key will represent the total number of nodes at that level.
2. Apply preorder traversal for travelling each node, and our **level** will start from 0; steps are as follows:
  - a. If the root is **NULL**, we can not get any node for the current level.
  - b. Else if the root is not **NULL**, we can go down another level. Hence we increment the count of nodes by one.

$$\text{nodesAtLevel}[level] = \text{nodesAtLevel}[level] + 1.$$

- c. Call the recursive function for the left and right subtree with **level + 1**.
- 3. Every value corresponding to the key of **nodesAtLevel** will represent the total number of nodes at that level in the given binary tree.
- 4. Return the maximum number of nodes in **nodesAtLevel**. That will be our maximum width for the given binary tree.

**Time Complexity:**  $O(N)$ , where **N** denotes the number of nodes in the given binary tree. The pre-order traversal algorithm is used, which takes  $O(N)$  time to visit each node. We also store the total number of nodes for all levels into the HashMap, where insertion will take  $O(1)$  time for the HashMap. Therefore, the overall time complexity will be  $O(N)$ .

**Space Complexity:**  $O(H)$ , where **H** denotes the height of the binary tree. We are using a HashMap to store the total number of nodes for each level, so there will be the total **H** level in the tree. In the worst case (Skewed Trees), **H** is equal to **N**. There can be at most **N** height. Therefore overall space complexity will be  $O(H)$ .

### Approach 3: Level Order Traversal

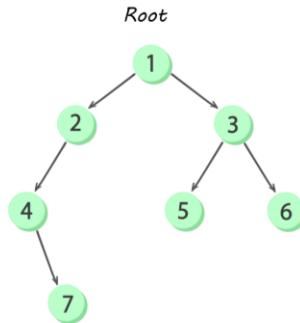
In this approach, we are going to use the level order traversal algorithm. While traversing in level order, we will be storing all the children of the nodes at the current level in the queue. And once we store all the children nodes for the current level, our queue will have all the next-level nodes. We can get the total number of nodes for the next level from the size of the queue. We maintain the maximum number of nodes in a level (—that is, the maximum size of the queue at any level) that we have till now. In the end, we have a maximum number of nodes that could be at any level.

#### Steps:

1. Using the level order traversal, push the root into the queue. And iterate through the queue until it becomes empty.
2. Get the queue's size at each level.
3. If the queue size is greater than the maximum number of nodes that we got till now, then we update our answer.
4. Dequeue the root node and push all the children of all the root nodes of the current level into the queue.

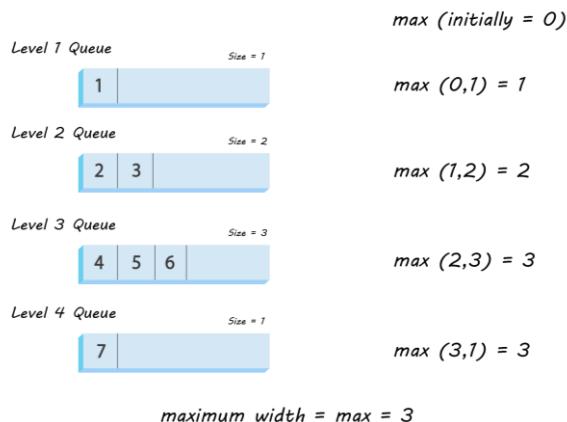
- When the queue becomes empty, return the final answer, which has stored the queue's maximum size at any level that will be the maximum width of the given binary tree.

Let us understand this approach using the following tree.



We maintain a queue and a **max** variable to store the maximum number of nodes at any given level (size of the queue).

The **max** variable will store the maximum value among the previous max and the current queue's size.



**Time Complexity:** **O(N)**, where **N** denotes the number of nodes in the given binary tree. We are using the level order traversal, in which there will be **N** push and **N** pop operations, where push and pop operation will take constant time for the queue data structure. Therefore, the overall time complexity will be **O(N)**.

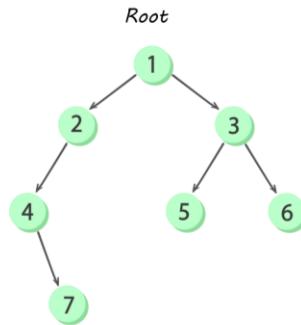
**Space Complexity:** **O(W)**, where **W** denotes the maximum width of the tree. Since in the level order traversal of a tree, a queue is maintained whose maximum size at any moment can go upto the maximum width of the binary tree.

## **2. Print Leaf Nodes** [<https://coding.ninja/P60>]

**Problem Statement:** Given a binary tree, write a function that returns a list containing all the leaf nodes of the binary tree in the order in which they appear from left to right.

### **Remember/Consider:**

If both horizontal and vertical distances are the same for two leaf nodes, print the one with smaller node data. For example, in the following tree., for



Leaf nodes 5 and 6 are at the same distance from left-most node 7 (horizontal distance) and are present at the same depth from the root node (vertical distance). We will print 5 first and 6 later as 5 is smaller in value than 6.

### **Horizontal distance:**

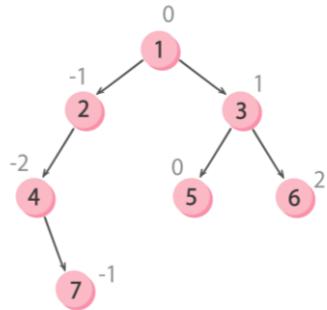
For the nodes of a binary tree, the horizontal distance is defined as follows:

Horizontal distance of the root = 0

Horizontal distance of a left child = horizontal distance of its parent - 1

Horizontal distance of a right child = horizontal distance of its parent + 1

For the above tree, the horizontal distances are:



### **Input Format:**

Elements in the level order form. The input consists of values of nodes separated by a single space in a single line. In case a node is null, we take -1 on its place.

### **Output Format:**

You have to return a list containing all the leaf nodes of the binary tree in the order in which they appear from left to right.

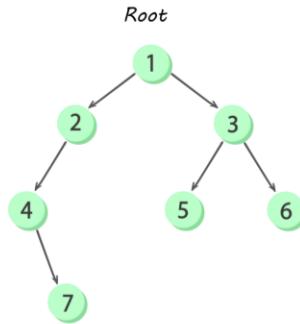
### **Sample Input:**

1 2 3 4 -1 5 6 -1 7 -1 -1 -1 -1 -1

### **Sample Output :**

7 5 6

### **Explanation:**



For the above tree, the leaf nodes are 7, 5, and 6.

### **Approach: Vertical Order Traversal**

In this approach, we will be storing the horizontal level of each node. The horizontal level of the root node is considered to be zero. Whenever we go to the left child of any node, its horizontal level is said to be one less than that of the parent node. Whereas, whenever we go to the right child of any node, its horizontal level is said to be one more than that of the parent node.

#### **Steps:**

Traverse the tree recursively while maintaining the horizontal level and the vertical depth and a list that stores triplets of the node's value, its horizontal level, and its vertical level

1. Add the current node to the list only if it's a leaf node.
2. In case the current node is not a leaf node, traverse it further on to its children. For the left child, the horizontal level is the horizontal level of the current node minus one, and the vertical level is the vertical level of the current node plus one.
3. Similarly, for the right child, the horizontal level of the right child is the horizontal level of the current node plus one, and the vertical level is the vertical level of the current node plus one.
4. If the current node is null, return without doing anything.
5. After the traversal is done and the list is filled, sort the list according to the required conditions.

6. Then iterate through this list from start to end to copy only the node's value (not the levels) to a different list called **ret**.
7. Finally, return **ret**.

**Time Complexity:**  $O(N)$ , where **N** denotes the total number of nodes in the binary tree.

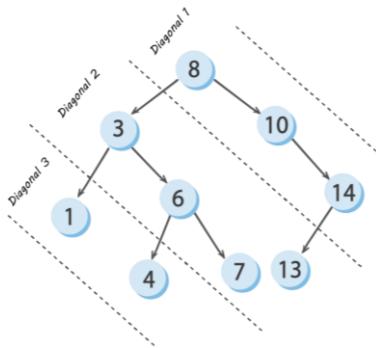
**Space Complexity:**  $O(N)$ , where **N** denotes the total number of nodes in the binary tree.

### 3. Diagonal Anagram [<https://coding.ninja/P61>]

**Problem Statement:** You are given two binary trees, you need to return true if the diagonals of both the trees are anagram to each other, otherwise return false.

**Note:** Anagrams are words/numbers you spell by rearranging the letters/digits of another word/number.

Diagonals of a binary tree are shown below:



There are three diagonals in the given tree are:

**Diagonal 1:** 8 10 14

**Diagonal 2:** 3 6 7 13

**Diagonal 3:** 1 4

#### **Input Format:**

The **first line of input** contains elements of the first binary tree in the level order form. The line consists of values of nodes separated by a single space. In case a node is null, we take -1 on its place.

The **second line of input** contains elements of the second binary tree in the level order form. The line consists of values of nodes separated by a single space. In case a node is null, we take -1 on its place.

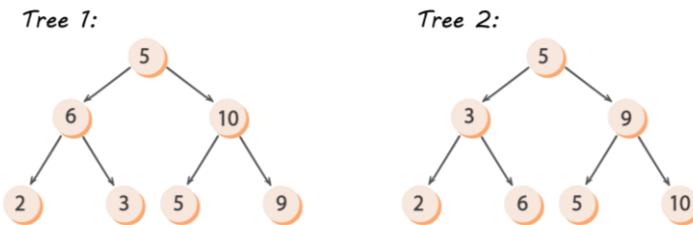
### Output Format:

The only line of output will contain "True" or "False".

### Sample Input:

5 6 10 2 3 5 9 -1 -1 -1 -1 -1 -1 -1 -1

5 3 9 2 6 5 10 -1 -1 -1 -1 -1 -1 -1 -1



### Sample Output:

True

### Explanation:

There are three diagonals in each tree.

#### Tree1:

Diagonal 1: 5 10 9

Diagonal 2: 6 3 5

Diagonal 3: 2

## Tree2:

Diagonal 1: 5 9 10

Diagonal 2: 3 6 5

Diagonal 3: 2

Since the corresponding diagonals of tree 1 and tree 2 are anagrams of each other, the output should be True.

## Approach 1: Recursive Approach

To store all diagonals separately, **HashMap** is used with its key as a diagonal number and value as the list containing the elements in that diagonal. Make recursive calls for the left and right and update the list corresponding to the diagonal, which stores the diagonal nodes.

### Steps:

1. Assign a distance **d = 0** to the root.
2. Create a class **maxSlope** that will store the maximum slope of the tree that is, total no of diagonal.
3. Run a helper function with parameter **root, distance** as 0, and **map**.
4. In the helper function,
  - a. Check if the root is null, then return.
  - b. Update the **maxSlope** to a maximum of **maxSlope** and **d**.
  - c. Update the list of that particular diagonal by adding the current node data.
  - d. Make a recursive call for left by incrementing **d** by one.
  - e. Make a recursive call for right but do not change **d**.
5. Finally, check for anagrams; if all the diagonals are anagram to each other, return true otherwise, return false.

For checking anagram, the following steps are followed.

### **Steps:**

1. If **maxSlope** of both the trees are not the same, then return false.
2. For every slope of the tree, get the list from the map and check for the anagram. If they are not an anagram, then return false.
3. Create two maps for both diagonals.
4. Store all the elements of the list in the map with its frequency.
5. Check if the frequency of all the keys is the same in both the map, then return true otherwise, return false.

**Time Complexity:**  $O(N)$ , where **N** denotes the total number of nodes in the given binary tree. In the worst case, we will be traversing each node only once.

**Space Complexity:**  $O(N)$ , where **N** denotes the total number of nodes in the given binary tree. In the worst case, we will have all the nodes of the Binary Tree in the recursion stack and also stored in the map. Hence, complexity is linear.

### **Approach 2: Level order traversal**

To store elements for each diagonal separately, **HashMap** is used with its key as diagonal number and value as the list containing the elements in that diagonal. Using level order traversal and maintaining a queue for it will help us achieve this task efficiently.

### **Steps:**

1. Assign a distance **d = 0** to the root.
2. To store the vertical distance, create another map **nodeDist** where the binary tree node will be the key, and vertical distance **d** would be the value.
3. Create a class **maxSlope** that will store the maximum slope of the tree that is, the total number of diagonals.
4. Create a **queue** for level order traversal.
5. Push the root node into the **queue** and store its vertical distance 0 into the map **nodeDist**.
6. While the **queue** is not empty
  - a. Get the front node of the **queue** as the **current** node.
  - b. While the **current** node is not null
  - c. Add the element of the node to its corresponding map key.

- d. If the left child of the node is not null, then add the left child into the **queue** and store the vertical distance **d + 1** in **nodeDist**.
  - e. Update the **current** node to the right of the **current** node.
7. Finally, check for anagrams; if all the diagonals are anagram to each other, then return true otherwise, return false.

For checking anagram, the following steps are followed.

### **Steps:**

1. If **maxSlope** of both the trees are not the same then return false.
2. For every slope of the tree, get the list from the map and check for the anagram. If they are not anagram then return false.
3. Create two maps for both the diagonals.
4. Store all the elements of the list in the map with its frequency.
5. Check if the frequency of all the keys is the same in both the map then return true otherwise, return false.

**Time Complexity:**  $O(N)$ , where **N** denotes the total number of nodes in the given binary tree. In the worst case, we will be traversing each node only once.

**Space Complexity:**  $O(N)$ , where **N** denotes the total number of nodes in the given binary tree as we will be storing all the nodes in the maps.

## **4. Time to Burn Tree** [<https://coding.ninja/P62>]

**Problem Statement:** You have been given a binary tree of **N** unique nodes and a **start** node from where the tree will start to burn. The **start** node will always exist in the tree print the time (minutes) that it will take to burn the whole tree.

**Note:** It is given to us that it takes 1 minute for the fire to travel from the burning node to its adjacent node and burn down the adjacent node.

### **Input Format:**

The **first line** of input will contain level order traversal of the binary tree.

The **second line** of input will contain the value of the start node.

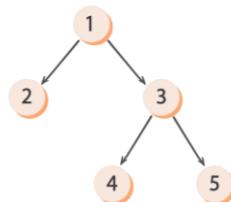
### Output Format:

A single integer, the time in minutes.

### Sample Input

1 2 3 -1 -1 4 5 -1 -1 -1 -1

3



### Sample Output:

2

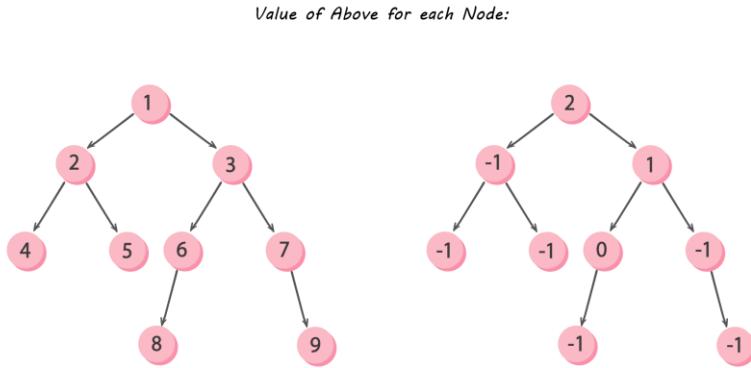
**Explanation:** In the zeroth minute, Node 3 will start to burn. After one minute, Nodes (1, 4, 5) that are adjacent to 3 will burn completely. After two minutes, the only remaining Node 2 will be burnt, and there will be no nodes remaining in the binary tree, i.e. the whole tree will burn in 2 minutes.

### Approach 1: Distance Of Farthest Node

The total time taken by the tree to burn completely will be equal to the distance of the farthest node from the **start node** in the tree. Hence, we will find three values for each node:

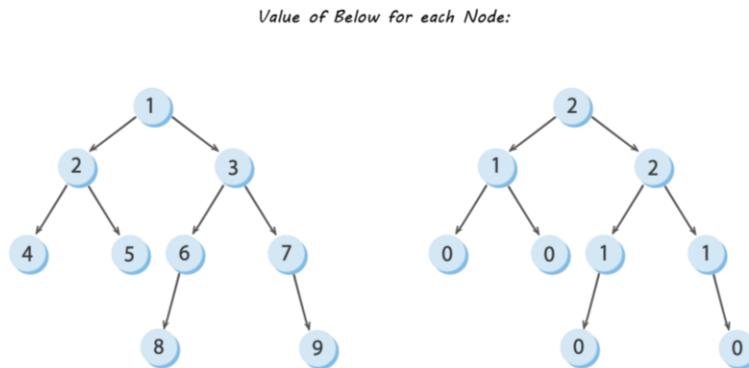
- **Above:** If the current node is an ancestor of the start node or if it is the start node itself, then the value of **above** for that node will be the shortest distance between the current node and the start node; otherwise, it will be -1.

For the given binary tree and **Start node** 6:



- **Below:** If the current node is an ancestor of the start node or if it is the start node itself, then **below** denotes the maximum number of edges from the **Start node** to a leaf node in the **Start node's** subtree. Otherwise, it denotes the number of edges in the longest path from the current node to a leaf node (in the left or right subtree).

For the given binary tree and **Start Node** 3,



- **Max:** This value denotes the length of the longest path that has been achieved so far if the **Start** node is found, otherwise -1. This can be calculated simply by finding the maximum of these three values:
  - The longest path below the starting node from the **Start node** to the leaf node (in the subtree of the **Start** node) — This value will be equal to **below** if we have found the **Start node** in any of the subtrees.
  - The longest path including the current node and **Start** node — This can be calculated as:

(the number of edges between the current node and **Start node** (if found) equals to **Above**) + (number of edges in the longest path in the subtree without **Start node** equals to **Below**) + 1 or (below + above + 1)

- For example:

For **Node 6**

$$\text{longest path} = (\text{below} + \text{above} + 1)$$

$$= (1 + 0 + 1)$$

$$= 2$$

- The **Max** value of the subtree in which the **Start** node is present.
- Finally, the **Max** of the root node will store the total time to burn the whole tree.

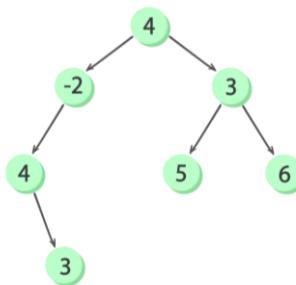
**Time Complexity:**  $O(N)$ , where **N** denotes the total number of nodes in the tree. We will be visiting all the nodes in the given tree; hence the time complexity is  $O(N)$ .

**Space Complexity:**  $O(H)$ , where **H** denotes the height of the tree. Since the recursion depth can extend up to **H**, extra space will be required for the recursion's stack.

## 5. Maximum Sum Path From the Leaf to Root [\[https://coding.ninja/P63\]](https://coding.ninja/P63)

**Problem Statement:** You are given a binary tree of **N** nodes. Find the path from the leaf node to the root node, which has the maximum path sum among all the root to leaf paths.

**Example:**



All the possible root to leaf paths are:

1. **3, 4, -2, 4** with sum **9**

2. **5, 3, 4** with sum **12**
3. **6, 3, 4** with sum **13**

Here, the maximum sum is **13**; thus, the output path will be **6, 3, 4**.

#### **Input Format:**

The **very first line** of input contains an integer **T** denoting the number of queries or test cases.

The **first and only line** of every test case contains elements of the binary tree in the level order form. The line consists of values of nodes separated by a single space. In case a node is null, we take 0 on its place.

#### **Output Format:**

For each test case, print the path from the leaf node to the root node, which has the maximum sum separated by spaces in a single line.

#### **Sample Input:**

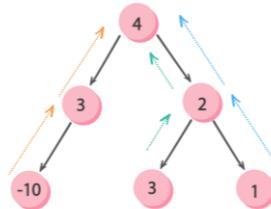
```
1
4 3 2 -10 0 3 1 0 0 0 0 0 0
```

#### **Sample Output:**

```
3 2 4
```

#### **Explanation:**

The tree given in the input can be represented as:-



Possible paths

10,3,4	→ sum = -3
3,2,4	→ sum = 9
1,2,4	→ sum = 7

All possible paths are

1. **-10, 3, 4** with sum **-3**.
2. **3, 2, 4** with sum **9**.
3. **1, 2, 4** with sum **7**.

Here, the maximum is **9**, hence the path is **3, 2, 4**.

### Approach 1: Recursive Subproblem

The idea here is that we will do a recursive solution by asking the children of the current node for the max sum path and then choose the path with the max sum. We will find the max sum path for the left subtree and the max sum path for the right subtree. Then, we will select the one with the max sum from both and insert the current node's data into it. Finally, we shall return the updated path.

#### Steps:

1. Create a function that accepts the **root** node as the parameter and stores the path in a list.
2. If the root is NULL, return an empty list.
3. Initialise a variable **leftPath** which would store the answer returned to us by calling our function on **root.left**.
4. Similarly, initialise a variable **rightPath** which would store the answer returned to us by calling our function on **root.right**.
5. Append **root.data** to both the lists returned from **leftPath** and **rightPath**.

6. Find the sum of both the lists.
7. Return the list with the greater sum.

**Time Complexity:**  $O(N + H^2)$  per test case, where **N** denotes the number of nodes in the binary tree and **H** is the height of the binary tree. In the worst case, for each level, we will be storing the path from the leaf node to every level and calculating the sum for the same. If the given tree is a skew tree, then time complexity will be  $O(N^2)$ .

**Space Complexity:**  $O(N)$  per test case where **N** denotes the number of nodes in the binary tree.

In the worst case, we are storing at max  $2 * N$  nodes.

### Approach 2: Recursive

The idea here is that we will recursively get the possible maximum path sum from root to leaf; we will store this in a variable, say **maxSum**, then find the path with a sum equivalent to **maxSum**.

We will recursively call to get the maximum path sum for left and right subtree. We will then return the maximum sum to be the current node's data + max (left maximum path sum, right maximum path sum). After getting the maximum path sum, find the path with a sum equivalent to the max path sum using recursion. We will maintain three functions to perform the above task:

- One function would get the possible maximum path sum from root to leaf, **maxSum**.
- One function would find the path with a sum equivalent to **maxSum**.
- The last function will simply be a helper function that calls to the first two functions and returns the final answer.

#### Steps for getting **maxSum** using **getSum(root)**:

1. Initialize **sum** = INT\_MIN
2. If root's left is not NULL:
  - a. Call **sum** = max(**sum**, **getSum(root.left)**)
3. If root's right is not NULL:
  - a. Call **sum** = max(**sum**, **getSum(root.right)**)

4. If sum == INT\_MIN, update sum with 0.

5. Finally, return **sum + root.data**.

#### **Steps for getting path using getPath(root, path, maxSum, currSum = 0):**

1. Initially, **path** is an empty list. Using recursive technique, we will append the final path in it.
2. If root == NULL, return true if **ans == sum**, otherwise, return **false**.
3. Add root's data to **sum**.
4. Append root's data to the **path**.
5. If **getPath(root.left, path, ans, currSum)** is true, return true.
6. If **getPath(root.right, path, ans, currSum)** is true, return true.
7. If the path is not found yet, delete the last element from the path and subtract root.data from the sum.

#### **Steps for helper function:**

- Initialise **ans = getSum(root)**; this will give you the maximum possible sum among all the root to leaf paths.
- Initialise a list **path = getPath(root, path, ans)**; this will give you the required path from the root to leaf.
- Reverse the **path** and return.

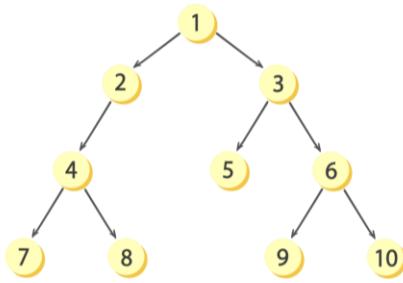
**Time Complexity:** **O(N)** per test case, where **N** denotes the number of nodes in the binary tree. In the worst case, we are traversing every node twice.

**Space Complexity:** **O(H)** per test case, where **H** denotes the height of the binary tree. In the worst case, extra space is used by the recursion stack, which goes to a maximum depth of **H**.

## **6. LCA of Three Nodes** [<https://coding.ninja/P64>]

**Problem Statement:** You have been given a binary tree of **N** nodes with integer values and three integers **N1, N2, and N3**. Find the LCA(Lowest Common Ancestor) of the three nodes represented by the given three (N1, N2, N3) integer values in the Binary Tree.

**For example:**



For the given binary tree: the LCA of (7,8,10) is 1

**Note:**

All of the node values of the binary tree will be unique. **N1, N2, and N3** will always exist in the binary tree.

**Input Format:**

The **first line** of input contains a single integer **T**, representing the number of test cases or queries to be run.

Then the **T** test cases follow.

The **first line** of each test case contains three single space-separated integers **N1, N2, and N3**, denoting the nodes of which LCA is to be calculated.

The **second line** of each test case contains elements in the level order form. The line consists of values of nodes separated by a single space. In case a node is null, we take -1 in its place.

**Output Format:**

For each test case, print the integer value of the node representing LCA in a single line.

**Sample Input:**

5 6 7

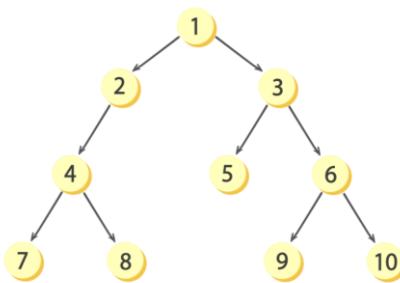
1 2 3 4 -1 5 6 7 8 -1 -1 9 10 -1 -1 -1 -1 -1 -1 -1 -1

### Sample Output:

1

### Explanation:

The binary tree will be represented as



Ancestors of 5: 3,1

Ancestors of 6: 3,1

Ancestors of 7: 4,2,1

Thus, the LCA (Lowest Common Ancestor) is 1

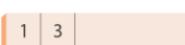
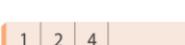
### Approach 1: By Storing Paths

In this approach, we will be storing paths from each node to the root. This way, we will have a list of ancestors to compare and find the lowest common ancestor.

### Steps:

1. A simple solution would be to store the path from the root to **N1**, the path from the root to **N2**, and the path from the root to **N3** in the three auxiliary Lists/Array.
2. Now, to store the path from the root to any node **X**, we create a recursive function that traverses the different path in the binary tree to find any node **X**:
  - a. If **root == null** return true
  - b. Add the root data to List/Array
  - c. If root data = **X** return true
  - d. If **X** is found in any of the subtrees, either left or right, then return true.
  - e. Else remove the current root data from the List/Array and then return false
3. Then we traverse all lists simultaneously till the values in the list match. The last matched value will be the LCA. If the end of one array is reached, then the last seen value is LCA.

*Path from root to Node*

(5) $N_1$ -	
(6) $N_2$ -	
(7) $N_3$ -	

**Time Complexity:**  $O(N)$ , where **N** denotes the total number of nodes in the given binary tree. In the worst case, the tree is traversed thrice to find paths of **N1**, **N2**, and **N3**, whose complexity will be  $O(N)$ , and then the stored path list  $O(N)$  is traversed. Therefore, the time complexity is a total of  $O(N)$ .

**Space Complexity:**  $O(N)$ , where **N** denotes the total number of nodes in the given binary tree. We will store all the nodes in the list as a path from the root to the node in the worst case. And we will have all the nodes of the binary tree in the recursion stack  $O(N)$ . Thus a total of  $O(N)$ .

### Approach 2: By Preorder Traversal

We will traverse the tree in a depth-first manner (recursively) to find the LCA in this method. We will return the node when we encounter either of three nodes, **N1**, **N2**, or **N3**. The LCA would then be the node for which both the subtree recursions return a non-NULL node. LCA can also be the node which itself is one of **N1**, **N2**, or **N3** and for which one of the subtree recursions returns that particular node.

**Steps:**

1. Traverse the tree from the root node.
2. If the current node is **N1**, **N2**, or **N3** node, we will return that node.
3. If the left or right subtree returns not null nodes, this means one of three nodes was found below in the subtree.
4. If at any point in the traversal, both the left and right subtree return some node, the current node can be LCA and LCA will be a top-level node in which the subtree is returning some node.

**Time Complexity:**  $O(N)$ , where **N** denotes the number of nodes in the binary tree.

**Space Complexity:**  $O(N)$ , where **N** denotes the number of nodes in the binary tree. In the worst case, we will have all the nodes of the binary tree in the recursion stack; hence, the space complexity is linear.

## **7. Count Special Nodes in Generic Tree** [<https://coding.ninja/P65>]

**Problem Statement:** You have been given a generic tree of integers. The task is to count the number of **Special Nodes**.

A node is a **Special Node** if there is a path from the root to that Node with all distinct elements.

**Input Format:**

The **first line** contains an integer **T** which denotes the number of test cases or queries to be run. Then the test cases follow.

The **first and only line** of each test case or query contains elements in the level order form. The order of the input is: data for the root node, number of children to root node, data of each of child nodes and so on and so forth for each node. The data of the nodes of the tree are separated by space.

**Output Format:**

For each test case/query, print the number of **special nodes** present in the given generic tree.

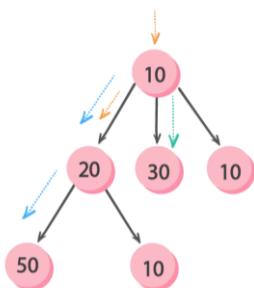
**Sample Input:**

10 3 20 30 10 2 50 10 0 0 0 0

**Sample Output:**

4

**Explanation:**



10  
10 → 20  
10 → 30  
10 → 20 → 50

We have 4 Special Nodes:

10 Path: 10

20 Path: 10 → 20

30 Path: 10 → 30

50 Path: 10 → 20 → 50

**Approach 1: Recursive Search**

We will do a recursive search on our tree, keeping a vector/list to check for the distinct element.

1. Start the recursive search from the root node with an empty vector/list.

2. Let's say we are at some node **x** of the tree with a vector/list containing all the elements from root to that node.
3. If the value of the node **x** is present in the vector/list, then it means our distinct element condition is violating. Hence, return 0.
4. Else insert the value of node **x** in vector/list and initialise the count of special nodes with one.
5. Recurse for children of node **x** with updated vector/list to count the special nodes in subtree of **x**.
6. Delete the value of node **x** from vector/list to make sure that all other child elements have been removed from the vector/list when we move to the next child.
7. Return the total count of special nodes.

**Time Complexity:**  $O(N^2)$ , where **N** denotes the number of nodes in the generic tree. We are visiting each node of the tree once and checking if the node's value already exists or not, then pushing the value of nodes in the vector/list, which is  $O(N)$  on average. Therefore, the overall time complexity is  $O(N^2)$ .

**Space Complexity:**  $O(N)$ , where **N** denotes the number of nodes in the Generic Tree. As we are doing recursive DFS, which can at max have a depth of  $O(N)$  and also our vector/list can have max size **N**.

### Approach 2: HashMap/Dictionary Approach

We will do a recursive search keeping a HashMap/Dictionary to check for the distinct element.

1. Start the recursive search from the root node with an empty HashMap/Dictionary.
2. Let's say we are at some node **x** of the Tree with a HashMap/Dictionary containing all the elements from root to that node.
3. If the value of the node **x** is present in HashMap/Dictionary, then it means our distinct element condition is violating, so return 0
4. Else insert the value of node **x** in HashMap/Dictionary and initialise the count of special nodes with one.
5. Recurse for children of node **x** with updated HashMap/Dictionary to count the special nodes in subtree of **x**.
6. Delete the value of node **x** from HashMap/Dictionary to make sure that all other child elements have been removed from the vector/list when we move to the next child.

7. Return the total count of special nodes.

**Time Complexity:** **O(N)**, where **N** denotes the number of nodes in the generic tree. We are visiting each node of the tree once and checking if the node's value already exists or not, then pushing the value of nodes in HashMap, which is **O(1)** on average. Therefore, overall time complexity is **O(N)**.

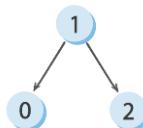
**Space Complexity:** **O(N)**, where **N** denotes the number of nodes in the generic tree.

## **8. Binary Tree from Parent Array [<https://coding.ninja/P66>]**

**Problem Statement:** Given an array parent, which represents a binary tree such that the parent-child relationship is defined by **(parent[i], i)**, which means that parent of i is parent[i]. The value of the root node will be **i** if -1 is present at **parent[i]**. Your task is to create the binary tree from the given parent array.

**For example:**

For the parent array [1, -1, 1], the tree will be:

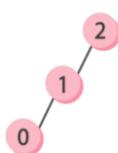


**(parent[0],0):** Parent of 0 is parent[0] = 1.

**(parent[1],1):** Parent of 1 is parent[1] = -1, which means it is the root node.

**(parent[2],2):** Parent of 2 is parent[2] = 1.

Similarly, for the parent array [1, 2, -1], the tree will be:



### Note:

From the parent array, multiple binary trees may be possible. You need to create a binary tree in such a way that these conditions satisfy:

- If the node has a left child as well as a right child, make sure the left child is smaller than the right child.
- If the node has only one child, make sure it has an only left child.

For example, for the parent array [1, -1], the accepted tree will be:



And not.



### Input Format:

The **first line of input** contains an integer **T** denoting the number of queries or test cases.

For each test case:

The **first line** contains an integer **N** which denotes the size of the parent array.

The **second line** contains **N** single space-separated integers, representing the elements of the parent array.

### Output Format:

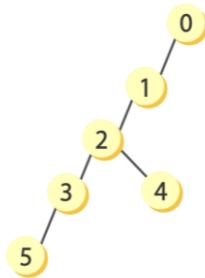
For each test case, print level order of Binary tree in a separate line.

### Sample Input:

```
1  
6  
-1 0 1 2 2 3
```

### Sample Output:

```
0 1 -1 2 -1 3 4 5 -1 -1 -1 -1 -1
```



### Approach 1: Brute force

The main idea is to repeatedly find out the children of a particular node, attach them with their parent node, and work on the child nodes separately.

### Steps:

1. Find out the root node by searching for `-1` in the given parent array.
2. Pass this root node as well as the parent array into a helper function.
3. The helper function returns us the final root of the binary tree by doing the following things:
  - a. If the root is `NULL`—that is, we are given the parent array of a `NULL` tree, return `NULL`.
  - b. Otherwise, traverse the parent array and find out the children of the current root. Store the left child in **First** and the right child in **Second**.

- c. Assign **First** to the **left child** of root and recursively call for creating the subtree which has **First** as its root.
  - d. Assign **Second** to the **right child** of root and recursively call for the creation of the subtree which has **Second** as its root.
4. Return the root fetched from the helper function.

**Time Complexity:**  $O(N^2)$ , where **N** denotes the size of the parent array. The tree we are creating has **N** nodes, and we are traversing at each node recursively. Additionally, for every node, we are traversing the parent array as well. This makes the time complexity is  $O(N * N)$ .

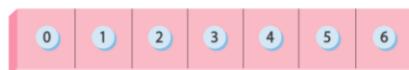
**Space Complexity:**  $O(H)$ , where **H** denotes the height of the tree being created. Extra space is used by a recursion stack that goes up to the height of the resulting tree, which, in the worst case, can go till **N**.

### Approach 2: Optimized Approach

Let us walk through the steps and an example where the parent array is [1, 5, 5, 2, 2, -1, 3].

#### Steps:

1. Create a list/array that stores tree nodes. Note that tree node values can be 0 to  $N - 1$ , thus make tree nodes for all these values and store them in a list say **NODE**.  
For the above example, the list will be:



2. Create a root node, say **root**, which will be updated by the root of the binary tree we are forming.
3. Now, we traverse the parent array from start to end and perform the following:-
  - a. **If  $\text{parent}[index] = -1$** , we have got the root node's value, which is an **index**.  
Remember, we have already created a node of the value **index**, which is stored at **NODE[index]**. Thus, we update the **root** with **NODE[index]**.

- b. Otherwise, we need to create a tree possible from **parent[index]**, as we have already made nodes for the value **parent[index]** and **index**.

Firstly, we will make left child:

**NODE[parent[index]] → left = NODE[index]**

If the left child is not NULL, that is, we already have left the child for a particular **parent[index]**, we create a right child:

**NODE[parent[index]] → right = NODE[index]**

Let us walk through the visualisations of the above example.

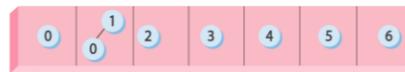
**Note:** The array given below is only to show the representation of each node at the  $i^{\text{th}}$  index of the NODE array and its subtree. This is only for the representation and clarity sake. In the actual memory, each index of the NODE array will only store the reference to a single node, not the complete tree separately.

1. Parent array: [1, 5, 5, 2, 2, -1, 3]

For index = 0, as  $\text{parent}[index] \neq -1$ , we perform the following steps-

**NODE[parent[0]] → left = NODE[0]**

**NODE[1] → left = NODE[0]**

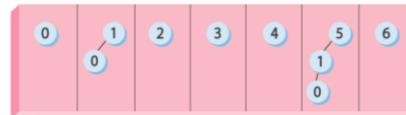


2. Parent array: [1, 5, 5, 2, 2, -1, 3]

For index = 1, as  $\text{parent}[index] \neq -1$ , we perform the following steps-

**NODE[parent[1]] → left = NODE[1]**

**NODE[5] → left = NODE[1]**



3. Parent array: [1, 5, 5, 2, 2, -1, 3]

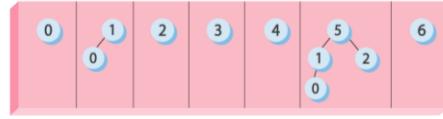
For index = 2 as  $\text{parent}[index] \neq -1$ , we perform the following steps-

$\text{NODE}[\text{parent}[2]] \rightarrow \text{left} = \text{NODE}[2]$

$\text{NODE}[5] \rightarrow \text{left} = \text{NODE}[2]$

We see that  $\text{NODE}[5] \rightarrow \text{left}$  is not NULL; therefore, we update it with its right child.

$\text{NODE}[5] \rightarrow \text{right} = \text{NODE}[2]$

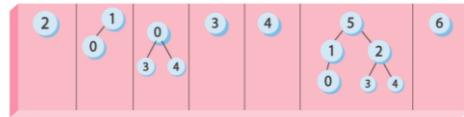


4. Parent array: [1, 5, 5, 2, 2, -1, 3]

Similarly, for index 3 and 4, the NODE list becomes:

$\text{NODE}[2] \rightarrow \text{left} = \text{NODE}[3]$

$\text{NODE}[2] \rightarrow \text{right} = \text{NODE}[4]$



5. Parent array: [1, 5, 5, 2, 2, -1, 3]

If index = 5, we find -1; thus, we make  $\text{NODE}[5]$  as the root.

Thus, the NODE list will be:



6. Now, return the root—that is,  $\text{NODE}[5]$

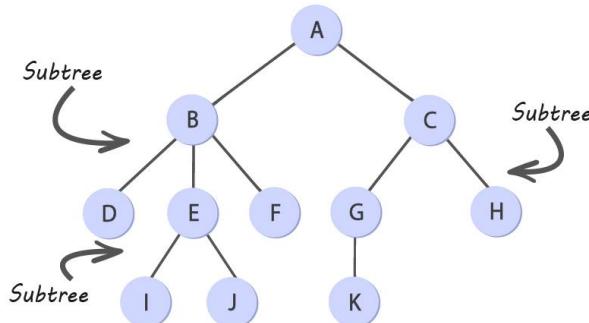
**Time Complexity:**  $O(N)$  per test case, where **N** denotes the size of the parent array. We are traversing the parent array once; hence the time complexity is  $O(N)$ .

**Space Complexity:**  $O(N)$  per test case, where  $N$  denotes the size of the parent array. We are creating a list of nodes of size  $N$ .

## 9. Count Elements in all Subtrees [\[https://coding.ninja/P67\]](https://coding.ninja/P67)

**Problem Statement:** You are given an arbitrary tree consisting of  $N$  nodes numbered from 0 to  $N - 1$ . You need to find the total number of elements in all the subtrees of the given tree. In other words, given a generic tree, find the count of elements in the subtrees rooted at each node.

A subtree of a tree  $T$  is a tree  $S$  consisting of a node in  $T$  and all of its descendants in  $T$ . The subtree corresponding to the root node is the entire tree. For better understanding, refer to the image:-



### Note:

1. The tree will always be rooted at 0.
2. You have to return the count of nodes in the subtree rooted at each node in the increasing order of node number.
3. The subtree's root is also counted in the subtree; thus the count of nodes in a subtree rooted at a leaf node is one.

### Input Format:

The **first line** of the input contains a single integer  $T$ , representing the number of test cases.

The **first line** of each test case contains a single integer  $N$ , representing the number of nodes in the given tree.

The **next  $N - 1$  lines** of each test case contain two space-separated integers  $u$  and  $v$ , denoting an edge between the node  $u$  and the node  $v$ .

### Output Format:

For each test case, print the number of nodes in all subtrees of the given tree.

**Sample Input:**

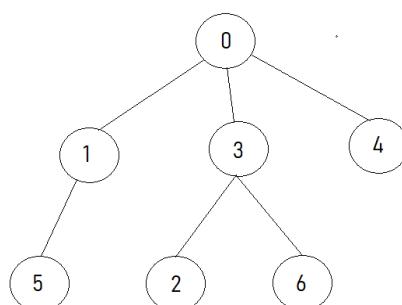
```
1
7
0 1
0 3
0 4
1 5
3 2
3 6
```

**Sample Output:**

```
7 2 1 3 1 1 1
```

**Explanation:**

The given tree looks like -



The subtrees rooted at nodes 5, 2, 6, and 4 have 1 node each.

The subtree rooted at node 1 has 2 nodes.

The subtree rooted at 3 has 3 nodes.

The subtree rooted at 0 has 7 nodes.

Hence, the output is [7, 2, 1, 3, 1, 1, 1].

**DFS approach:**

We run a modified DFS over the given tree to solve this problem.

1. Create an output array, **arr** of size **N**, where **N** is the number of nodes in the tree.

- Call the DFS function with source **src** as 0 and parent of the source as -1 (root has no parent), and the given adjacency list. The DFS function signature looks like:

```
countNodesDFS(adj, arr, *it, src);
```

- In the DFS function, initialise the count of nodes in the subtree rooted at **src** with 1.
- Then, iterate through all the immediate children of **src** and recursively call the DFS function with a current neighbour as the source and original **src** as its parent.
- Finally, update the count of nodes in the current subtree by adding the number of nodes in all the neighbour's subtrees to it:

```
Arr[src] += arr[*it];
```

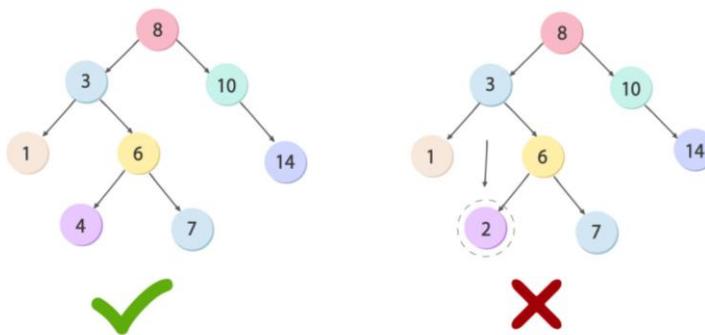
**Time Complexity:**  $O(N)$ , where **N** denotes the number of nodes in the tree.

**Space Complexity:**  $O(N)$ , where **N** denotes the number of nodes in the tree. Since we create an auxiliary array to store the answers, the space complexity is  $O(N)$ .

# 8. Binary Search Tree (BST)

## Introduction

- BST are specific types of binary trees.
- The concept of BST is inspired by the **Binary Search Algorithm**.
- In the BST structure, all the elements in the left subtree of any node have their values less than the current node's value, and all the elements in the right subtree have their values greater than the current node value (see image below for reference).
- The time complexity of operations like Insertion, Deletion, and Searching reduces significantly as it works on the principle of binary search rather than linear search.



The second image here is not a valid BST since the node with value 2 lies in the right branch of value 3, which violates the BST condition.

## How are BSTs stored?

In BSTs, a new node with **smaller** data than the current node is always inserted in the **left** subtree of the current node and the one with the **larger** data in the **right** subtree of the current node until we get a right place (a NULL node). To put it in other words, consider the root node data of BST to be N, then:

- Everything smaller than the value N will be placed in the left subtree.
- Everything greater than the value N will be placed in the right subtree.

A better understanding of insertion in a BST can be visualised with the help of the below example:

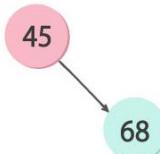
**For Example:** Insert [45, 68, 35, 42, 15, 64, 78] in a BST in the order they are given.

**Solution:** Follow the steps below:

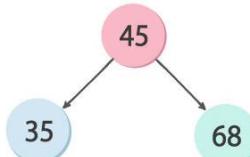
1. Since the tree (BST) is initially empty, the first node will automatically be the root node—that is, **45**.

45

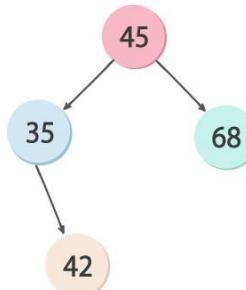
2. The next element to be inserted is **68**, which is greater than 45, so it will go to the right side of the root node.



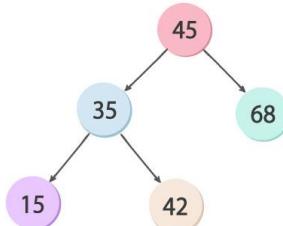
3. To insert **35**, we will start comparing the value from the root node. Since 35 is smaller than 45, it will be inserted on the left side.



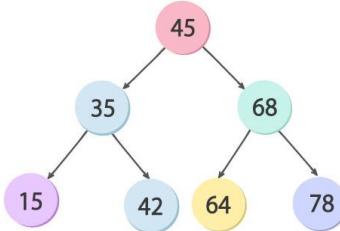
4. Next, we need to insert **42**; we can see that 42 is less than 45, so it will be placed on the left side of the root node. Next, we will check for the right place for 42 in the left subtree. We can see that  $42 > 35$ ; this means 42 will be the right child of 35, which is still a part of the left subtree of the root node.



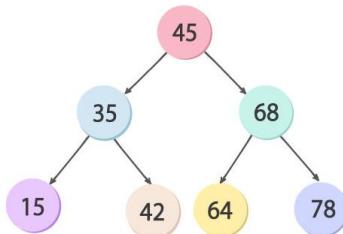
5. To insert **15**, we will follow the same approach starting from the root node. Here,  $15 < 45$ , thus we will choose the left subtree. Again,  $15 < 35$ , therefore we continue to move towards the left. As the left subtree of 35 is empty, so 15 will be the left child of 35.



6. Continuing further, to insert **64**, we find that  $64 > 45$  (root node's data). Since 45 is less than 68, it will be the left child of 68.



7. Finally, we insert **78**; we can see that  $78 > 45$  and  $78 > 68$ , so it will be the right child of 68.



The above figure shows the final BST obtained after inserting elements in the given order.

If we follow the **inorder traversal** of a BST, we get all the elements in **sorted order (ascending order)**; hence searching in BST can be thought of as applying a Binary Search Algorithm which considerably reduces the searching time.

As seen above, to insert an element, we will either traverse only the left subtree or only the right subtree of a node ignoring the other half completely till we reach a leaf node. Hence, the **time complexity of insertion** for each node is  **$O(h)$**  (where  **$h$**  is the height of the BST).

Hence for inserting  **$n$**  nodes, the time complexity will be  **$O(n*h)$** .

**Note:** The value of  **$h$**  is equal to  $\log(n)$  on average but can even go up to  **$O(n)$**  in the worst case (as in the case of skewed trees).

## Main operations involving Binary Search Tree (BST)

### Searching BST:

This operation involves verifying if a given value is present in a BST or not.

### Steps to search in the BST:

- Visit the root.
- If it is null, return false.
- If it is equal to the value to be searched, return true.

- If the value is less than the root's data, search in the left subtree. Else search in the right subtree.

```

function search(root, val)
    if the root is null
        return false

    /*
        If the current node's data is equal to val, then return true
        Else call the function for left or right subtree depending on
        the value of val
    */

    if root.data equals val
        return true

    if val < root.data
        return search(root.left, val)

    return (root.right, val)

```

### **Insertion in BST:**

This operation involves adding a given value to a BST. We assume that the same value is not present already.

### **Steps to insert a value in the BST:**

- Visit the root.
- If it is null, create a new node with this value, set it as root and return it.
- If the value is less than the root's data, make a recursive call with the new root as root→left (since the insertion of the desired node would now be in the left subtree).
- If the value is greater than the root's data, make a recursive call with the new root as root→right (since the insertion of the desired node would now be in the right subtree).

```

function insert(root, val)
    if the root is null
        /*
            We have reached the correct place, so we create a new
            node with data val
        */
        temp = newNode(val)
        root=temp
        return temp

    /*

```

```

else we recursively insert to the left and the right subtree
depending on the val
*/
if val < root.data
    root.left = insert(root.left, val)

else
    root.right = insert(root.right, val)

```

## **Deletion in BST:**

This operation involves deleting a given value from a BST.

### **Steps for deletion in the BST**

- Visit the root.
- If the root is null, return null.
- If the value is greater than the root's data, call delete in the right subtree.
- If the value is lesser than the root's data, call delete in the left subtree.
- If the value is equal to the root's data, then
  - If it is a leaf, then delete it and return null.
  - If it has only one child, then return that single child.
  - Else replace the root's data with the minimum in the right subtree (this is done to maintain BST property even after the deletion) and then delete that minimum valued node.

```

function deleteData(val, root)

    // Base case
    if root equals null
        return null

    // Finding that root by traversing the tree
    if (val > root.data)
        root.right = deleteData(val, root.right)
        return root

    else if (val < root.data)
        root.left = deleteData(val, root.left)
        return root

    // found the node with val as data
    else
        if (root.left equals null and root.right equals null)

```

```

        // Leaf node
        delete root
        return null

    else if (root.left equals null)
        // root having only right child
        return root.right

    else if (root.right equals null)
        // root having only left child
        return root.left

    else
        // root having both the childs
        minNode = root.right;

        // finding the minimum value node in the right subtree
        while (minNode.left not equals null)
            minNode= minNode.left

        rightMin = minNode.data
        root.data = rightMin

        // now deleting that replaced node using recursion
        root.right = deleteData(rightMin, root.right)
        return root

```

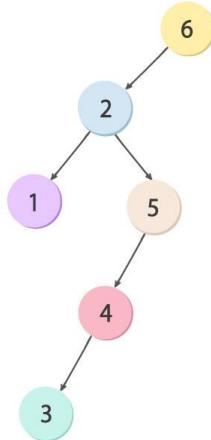
## Time Complexity of Various Operations

If **n** is the total number of nodes in a **BST**, and **h** is the height of the tree—which is equal to **O(logn)** on average and can be **O(n)** in worst case (for skewed trees)tree—then the time complexities of various operations for a single node in the worst case are as follows:

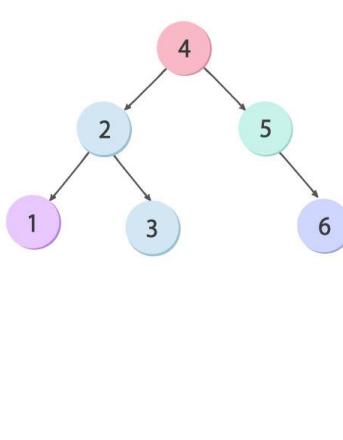
Operations	Time Complexity
Search(val)	$O(h)$
Insert(val)	$O(h)$
delete(val)	$O(h)$

## Balanced BST

- It is a special kind of a **BST** in which, for every node, the difference between the heights of the left and the right subtrees is at most **one**.
- Balanced BST's follow all the properties of a normal BST.



**(a) Not a balanced binary search tree**



**(b) Balanced binary search tree**

Here the first image is not a balanced tree since the difference between the left subtree height and the right subtree's height for the root node is 2, which is greater than 1.

Thus for a balanced BST the below equation must hold true:

$$|\text{Height\_of\_left\_subtree} - \text{Height\_of\_right\_subtree}| \leq 1$$

This equation must be valid for every node present in the BST for it to be a balanced BST.

It can be proved that the height of a Balanced BST is  $O(\log n)$ , where  $n$  is the number of nodes in the given tree.

This can be summarised as the time complexity of operations like searching, insertion, and deletion can be performed in  $O(\log n)$ .

Many Binary search trees are balanced BSTs like.

- AVL Trees (also known as self-balancing BST, uses rotation to balance)
- Red-Black Trees
- 2 - 4 Trees

## Applications of Binary Search Tree

Following are some of the applications of BSTs:

- BSTs are used to implement hashmaps and tree sets.
- They are used to implement dictionaries in various programming languages.
- They are also used to implement multi-level indexing in Databases.

## Practice Problems

## **1. BST Delete** [<https://coding.ninja/P68>]

**Problem Statement:** You are given a binary search tree (BST) and a key-value **K**. You need to delete the node with value **K**. It is guaranteed that the node with value **K** will be present in the tree.

**Note:** All the elements of the binary search tree are unique.

### **Input Format:**

The first line of input contains an integer **T**, representing the number of test cases.

The **first line** of every test case contains elements in the level order form. The input consists of values of nodes separated by a single space in a single line. In case a node is null, we take -1 on its place.

The **second line** of every test case contains a single integer **K** denoting the node's value to be deleted.

### **Output Format:**

For each test case, print the resultant binary search tree after the deletion in the level order form.

**Note:** Deletion in BST should be thought of as a two-step process. First to search the required node in the BST and then deleting it.

### **Sample Input:**

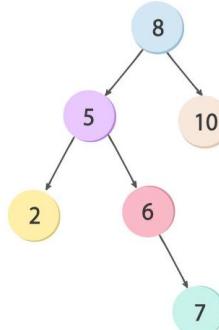
```
1
8 5 10 2 6 -1 -1 -1 -1 -1 7 -1 -1
6
```

### **Sample Output :**

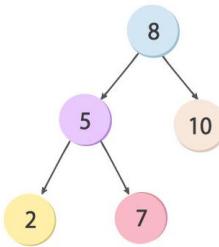
```
8 5 10 2 7 -1 -1 -1 -1 -1 -1
```

### **Explanation**

The BST formed from the sample input is as shown:



For the given example, we need to delete the node with the value 6. So after deletion, the BST would look like:



### Approach 1: Recursive Solution 1

We will traverse the tree by recursion and find the node to be deleted. Once the node is found, it can be of three types — has no children, has 1 child, or has 2 children. Deletion in each of the three cases would vary a bit.

#### Steps:

1. Once the required node which is to be deleted is found, figure out its type. Depending on the number of its children, as discussed above, there can be three cases:
  - a. The node has 0 children, and thus node is a leaf. In this case, simply remove the node from the tree.
  - b. The node has 1 child. In this case, delete the node and copy the child to this node.
  - c. The node has 2 children. In this case, copy the inorder successor to this node (minimum of the right subtree). You could have also used the inorder predecessor (maximum of the left subtree).
2. Finally, delete the given node and return the root of the new binary tree.

**Time Complexity:**  $O(H)$ , where  $H$  denotes the height of the binary search tree. In the worst case, the height can be  $H = N$ , where  $N$  is the number of nodes in the BST for a skewed binary tree.

**Space Complexity:**  $O(H)$ , where  $H$  denotes the height of the binary search tree. In the worst case, the height can be  $H = N$ , where  $N$  is the number of nodes in the BST for a skewed binary tree. A stack takes up this space for recursive calls of  $N$  nodes of the binary tree.

### Approach 2: Optimised Approach

As discussed in the previous approach, there can be three cases when a node is to be deleted. In this approach, we will optimise the third case.

#### Steps:

1. First, identify the type of node to be deleted.  
 Case I: The node has 0 children, and thus the node is a leaf. In this case, simply remove the node from the tree.

Case 2: The node has a single child. In this case, delete the node and copy the child to this node.

Case 3: The node has 2 children. In this case, delete the node and copy the inorder successor to this node. (We could have also used the inorder predecessor).

- i. In approach 1, we found the inorder successor node every time we found that the node has two children.
  - ii. Here we update the node's value to be deleted with the value of the found inorder successor and then make a recursive call with the new value to be deleted as the inorder successor node value.
2. Finally, delete the given node and return the root of the new binary tree.

**Time Complexity:**  $O(H)$ , where  $H$  denotes the height of the binary tree. In the worst case, the height can be  $H = N$ , where  $N$  is the number of nodes in the BST for a skewed binary tree.

**Space Complexity:**  $O(H)$ , where  $H$  denotes the height of the binary tree. In the worst case, the height can be  $H = N$ , where  $N$  is the number of nodes in the BST for a skewed binary tree. A stack takes up this space for recursive calls of  $N$  nodes of the binary tree.

## **2. Validate BST** [<https://coding.ninja/P69>]

**Problem Statement:** Given a binary tree with  $N$  number of nodes, check if that input binary tree is a BST or not. If it is a BST, return true, otherwise, return false.

### **Input Format:**

The **first line** contains an integer  $T$ , denoting the number of test cases or queries to be run. Then the test cases follow.

The **first line** of every test case contains elements in the level order form. The input consists of values of nodes separated by a single space in a single line. In case a node is null, we take -1 on its place.

### **Output Format:**

For each test case, return true if the binary tree is a BST, else return false. A separate line will denote output for every test case; otherwise.

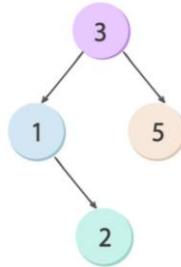
### **Sample Input:**

3 1 5 -1 2 -1 -1 -1 -1

### **Sample Output:**

true

**Explanation:** The input binary tree looks as:



**Note:** Every node in the BST should follow the property that all the node values in the left subtree and all the node values in the right subtree should be smaller and greater than the current node value, respectively.

Checking the above BST condition for all the nodes in the given example:

**Level 1:** For node 3, all the nodes in the left subtree (1, 2) are less than 3, and all the nodes in the right subtree (5) are greater than 3.

**Level 2:** For node 1, the left subtree is empty, and all the nodes in the right subtree (2) are greater than 1.

For node 5: both left and right subtrees are empty.

**Level 3:** For node 2, both left and right subtrees are empty.

In the given example, since all the nodes follow the property of a binary search tree, the function should return true.

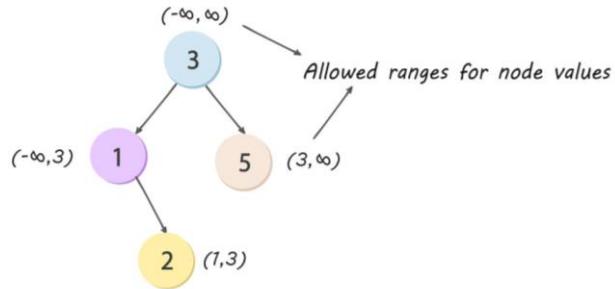
### Approach 1: Brute Force

In this approach, we traverse each node of the tree recursively and find out if it fulfills the property of a BST.

#### Steps:

1. For each node, store the minimum and maximum value allowed for that node.
2. Initially, as all the integer values are allowed for the root node, the minimum value should be  $-10^9$  and the maximum value should be  $10^9$ . Here we can also use built-in **INT\_MIN** and **INT\_MAX** constants.
3. If the value of that node is not in the allowed range of minimum and maximum value, then return false.
4. For the left subtree of a node with data **x**, update the maximum value to **x**.
5. For the right subtree of a node with data **x**, update the minimum value to **x**.  
This will ensure every node in the left subtree will be less than or equal to **x**, and every node in the right subtree will be greater than or equal to **x**.
6. Check this for each node of the tree; if every node is in the range, then the tree is a binary search tree, else not.

The above algorithm can be illustrated for the given example as follows:



**Time Complexity:**  $O(N)$ , where  $N$  denotes the number of nodes in the binary tree. We are recursively traversing through all the nodes of the tree.

**Space Complexity:**  $O(N)$ , this space is taken by the stack for recursive calls of  $N$  nodes of the binary tree.

### Approach 2: Inorder Traversal

This approach considers that if an inorder traversal is performed on a binary search tree, then the elements are in ascending order.

#### Steps:

1. While performing the traversal, keep track of the previous node value and the current node value.
2. If, for any node, the previous node value is greater than the current node value, then the binary tree is certainly not a binary search tree, thus return false.
3. If the previous node value is smaller than the current node value for all nodes, then the traversal is in the ascending order, and thus true is returned.

**Time Complexity:**  $O(N)$ , where  $N$  denotes the number of nodes in the binary tree. We are recursively traversing through all the nodes of the tree.

**Space Complexity:**  $O(N)$ , this space is taken by the stack for recursive calls of  $N$  nodes of the binary tree.

## 3. Binary Tree To BST [<https://coding.ninja/P70>]

**Problem Statement:** You are given a binary tree consisting of  $N$  nodes where every node is a distinct integer. Your task is to convert the given binary tree to a binary search tree (BST) without changing the original structure of the input binary tree.

#### Note:

1. The conversion must be done in such a way that the original structure of the input binary tree is intact.
2. Each node of the input binary tree is associated with a unique integer value.

### **Input Format:**

The **first line** contains an integer **T** which denotes the number of test cases or queries to be run. Then the test cases follow.

The **first line** of each test case contains the tree elements in the level order form separated by a single space. In case a node is null, we take -1 on its place.

### **Output Format:**

For each test case, print the level order traversal of the modified binary tree.

### **Sample Input:**

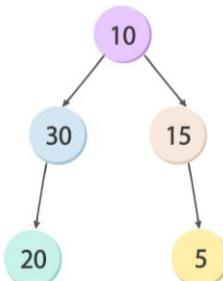
10 30 15 20 -1 -1 5 -1 -1 -1 -1

### **Sample Output:**

15 10 20 5 30

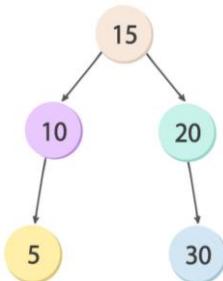
### **Explanation:**

The given binary tree is represented as follows:



The inorder traversal of the given input tree is 20 30 10 15 5.

After converting this tree to BST. It will look as follows:



The inorder traversal of the modified tree will be 5 10 15 20 30. Since the inorder is sorted and it also meets the condition of restoring the original structure of the given input tree, the above tree is a valid BST.

### **Approach: Inorder Traversal**

Here we make use of the fact that the inorder traversal of a BST is sorted.

### **Steps:**

1. We will traverse the given tree and store the data value of nodes in an array/list.

2. Now, sort the array in increasing order.
3. Traverse the tree again, but now in an inorder fashion, and replace the node values with the appropriate values present in the array/list.

**Time Complexity:**  $O(N \log N)$ , where **N** denotes the number of nodes in the binary tree. Since we store the node's values in the array and then sort it in increasing order, the time complexity is  $O(N \log N)$  because an additional sorting step is involved.

**Space Complexity:**  $O(N)$ , where **N** denotes the number of nodes in the binary tree. In the worst case (skewed trees), we will have all the nodes of the binary tree in the recursion stack and the array/list. Hence, the space complexity is linear.

## 4. Predecessor And Successor In BST [<https://coding.ninja/P71>]

**Problem Statement:** You are given a binary search tree of integers with **N** nodes. You are also given a **KEY** which represents data of a node of this tree. Your task is to find the predecessor and successor of the given node in the BST.

### Note:

1. The predecessor of a node in BST is that node that is visited just before the given node in the inorder traversal of the tree. If the given node is visited first in the inorder traversal, then its predecessor is NULL.
2. The successor of a node in BST is that node that is visited immediately after the given node in the inorder traversal of the tree. If the given node is visited last in the inorder traversal, then its successor is NULL.
3. The node for which predecessor and successor are to be found will always be present in the given tree.

### Input Format:

The first line contains an integer **T** which denotes the number of test cases or queries to be run. Then the test cases follow.

The **first line** of each test case contains the elements of the tree in the level order form separated by a single space. If any node does not have a left or right child, take -1 in its place.

The **second line** of each test case contains a **KEY** representing the data of the node for which predecessor and successor are to be found.

### Output Format:

For each test case, print two single space-separated integers representing data values of the predecessor and the successor node, respectively. If any of the two doesn't exist, print -1 in place of it.

### Sample Input:

15 10 20 8 12 16 25 -1 -1 -1 -1 -1 -1 -1 -1

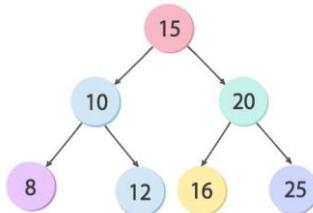
10

### Sample Output:

8 12

### Explanation:

The tree can be represented as follows:



The inorder traversal of this tree will be 8 10 12 15 16 20 25. Since the node with data 8 is on the immediate left of the node with data 10 in the inorder traversal, the node with data 8 is the predecessor. Since the node with data 12 is on the immediate right of the node with data 10 in the inorder traversal, the node with data 12 is the successor.

### Approach 1: Inorder Traversal

The fact that all the data values are unique makes the solution look very intuitive. We can simply store the inorder traversal of the given tree in the array and find the element present before and after the given node in the array.

#### Steps:

1. Create an array **inOrder**, and using the inorder traversal technique, store the values of the BST inside this array
2. After the traversal, we can find the given node in the inorder array and return its predecessor and successor, if any.

**Time Complexity: O(N)**, where **N** denotes the number of nodes in the BST. As we are traversing each node of the BST once, the time complexity will be linear.

**Space Complexity: O(N)**, where **N** denotes the number of nodes in the BST. In the worst case (skewed trees), we will have all the nodes of the BST in the recursion stack. Also, the maximum possible size of the array used to store inorder traversal will be equal to **N**. Hence; the space complexity is linear.

### Approach 2: Optimised Approach

This approach is based on a very astute observation. A node's predecessor in a BST is the greatest value present in its left subtree, which also means the rightmost value in the left subtree (if the left subtree of the node is not NULL).

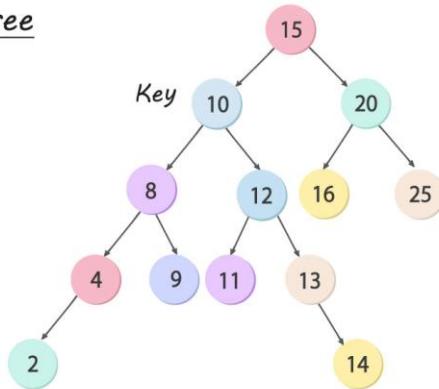
Similarly, a node's successor in a BST is the smallest value present in its right subtree, which also means the leftmost value present in the right subtree ( if the right subtree of the node is not NULL).

Thus we may follow the following steps.

**Steps:**

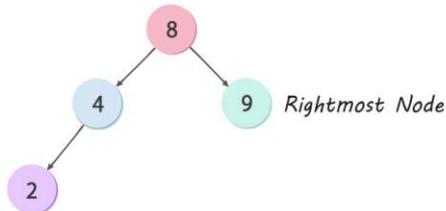
1. Initialise two variables — **predecessor** = -1 and **successor** = -1. These variables would be used to keep track of the possible predecessor and successor of the required node.
2. Recursively if the key is found in the tree.
  - If the left subtree of the found node is not NULL :
    - Run a loop to find the maximum of the left subtree and then set the predecessor to this maximum value.
  - If the right subtree of the found node is not NULL:
    - Run a loop to find the minimum of the right subtree and then set the successor to this minimum value.
3. If the current node value is greater than the key :
  - Update the successor variable to the current node value.
  - Then search for the key recursively in the left subtree.
4. If the current node value is smaller than the key :
  - Update the predecessor variable to the current node value.
  - Then search for the key recursively in the right subtree.

Tree

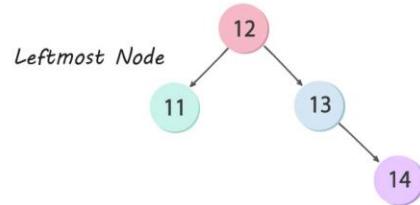


Inorder: 2, 4, 8, 9, 10, 11, 12, 13, 14, 15, 16, 20, 25  
 Key

Left Subtree of Key



Right Subtree of Key



In the above example, from the inorder traversal, it can be seen that 9 is the predecessor of 10, and 11 is the successor of 10.

Drawing the left subtree of 10 shows us that the predecessor 9 is the rightmost node of the left subtree.

Drawing the right subtree of 10 shows that the successor 11 is the leftmost node of the right subtree.

**Time Complexity:**  $O(N)$ , where  $N$  denotes the number of nodes in the BST. In the worst case(skewed tree), we will have to traverse all the nodes in the BST. Hence the time complexity is  $O(N)$ .

**Space Complexity:**  $O(1)$ , as constant space is used.

## **5. Sorted Linked List to Balanced BST** [<https://coding.ninja/P72>]

**Problem Statement:** You are given a singly linked list in which nodes are present in increasing order. Your task is to construct a balanced binary search tree with the same data elements as the given linked list. A balanced BST is defined as a BST in which the height of two subtrees of every node differs no more than 1.

### **Input Format:**

The **first line** of input contains an integer **T** denoting the number of test cases to run.

Then the test case follows.

The **only line** of each test case contains the elements of the singly linked list separated by a single space and terminated by -1. Hence, -1 would never be a list element.

### **Output Format:**

Output for every test case will be printed in a separate line in the level order form. Put -1 wherever the node is NULL.

### **Sample Input:**

1 2 3 4 5 -1

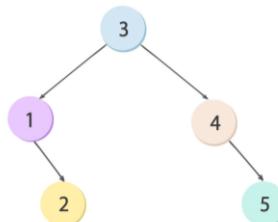
### **Sample Output:**

3 1 4 -1 2 -1 5 -1 -1 -1 -1

### **Explanation:**

The balanced BST corresponding to the given input linked list is:

**Note:** For a given list, there could be various other valid BST's too.

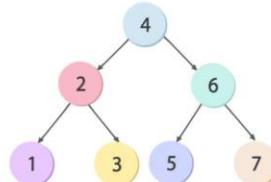
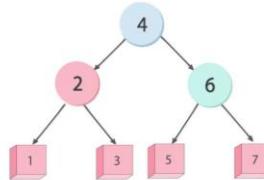
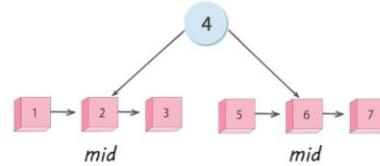
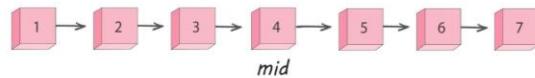


### **Approach 1: Recursion**

The key observation here is that the middle node of the linked list would be the root of the BST. Therefore the nodes which lie to the left of the middle node will necessarily form the left subtree of the BST. Similarly, the nodes which lie to the right to it will form the right subtree. Therefore, we will devise a recursive solution based on this observation.

### **Steps:**

1. Keep the head pointer of the linked list as one of our parameters of the recursion function.
2. Find the pointer to the middle node of the linked list by the standard **two pointers approach** (Hare-tortoise Approach).
  - a. The data of this node will be copied to the current root node of the subtree.
  - b. Let the previous pointer to the middle node be **prev**. We will set **prev → next = NULL**.
3. Recurse on the left half of this linked list and by passing the **head** pointer as our parameter. This will necessarily become the left subtree of the current root node.
4. Recurse on the right half of the linked list by passing the **middle → next** pointer as our parameter of the head of the linked list. This will necessarily become the right subtree of the current root node.



The above image depicts the stepwise formation of the required BST from the given input Linked List.

**Time Complexity: O(NlogN)**, where **N** denotes the number of nodes in the BST. As the binary tree is balanced, its height will be at most **logN**. Since we are iterating over the complete linked list to find the middle element for every level, the time complexity will be **O(NlogN)**.

**Space Complexity:**  $O(\log N)$ , where  $N$  denotes the number of nodes in the binary tree. As the binary tree is balanced, its height will be at most  $\log N$ . Hence, the size of the recursive stack will be at most  $\log N$ .

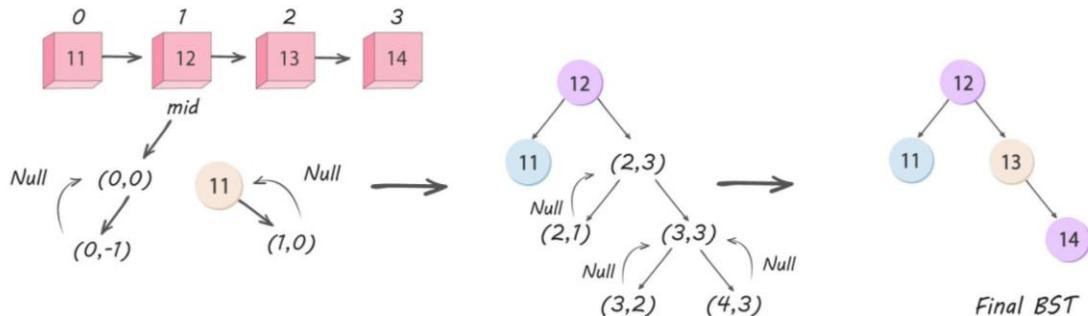
### Approach 2: Inorder Traversal of BST

The inorder traversal of the BST gives elements in sorted order. Therefore, the linked list representation is the inorder traversal of a BST. So we can devise a recursive solution similar to the inorder traversal of a BST.

#### Steps:

1. First, we will find the size of the linked list by iterating through it. Let this size be  $N$ .
2. We will keep (**leftBoundary**, **rightBoundary**) as our recursion parameters, with initial values equal to  $(0, N - 1)$ .
3. We can find the middle element '*mid*' index of the linked list by  $(\text{leftBoundary} + \text{rightBoundary}) / 2$ .
  - a. Now, we recurse on (**leftBoundary**, **mid - 1**), which will be the left subtree of the BST.
4. We will assign **head** → **data** to the current node of BST and reassign the head pointer as **head** → **next**.
5. Next, we recurse on (**mid + 1**, **rightBoundary**), which will be the right subtree of the BST.
6. At any point if (**leftBoundary** > **rightBoundary**), we will return NULL.
7. Finally, attach the returned left and right subtrees to the newly formed node as its left and right children, respectively and return the root of the formed BST.

For better understanding, we look at an example of  $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$  as our given input linked list:



**NOTE:** The numbers within parentheses refer to the **leftBoundary** and the **rightBoundary** variables in the recursive calls.

**Time Complexity:**  $O(N)$ , where  $N$  denotes the number of nodes of the BST. The time required to find the size of the linked list will be  $O(N)$ , as every node on the linked list will be visited at most once.

**Space Complexity:**  $O(\log N)$ , where  $N$  denotes the number of nodes in the binary tree. As the binary tree is balanced, its height will be at most  $\log N$ . So the size of the recursive stack will be at most  $\log N$ .

## 6. Total Number of BSTs using Array Elements as the Root Node

[<https://coding.ninja/P73>]

**Problem Statement:** Given a sequence  $\text{arr}$  of  $N$  integers. For each  $\text{arr}[i]$  where  $0 \leq i < N$ , find the number of binary search trees (BST) possible with elements of the given sequence  $\text{arr}$  as nodes of the BST and the element  $\text{arr}[i]$  as the root node. The answer could be very large; output your answer modulo  $(10^9 + 7)$ . Also, use modular division when required.

### Input Format:

The first line of input contains an integer  $T$  denoting the number of test cases.

The next  $2 * T$  lines represent the  $T$  test cases.

The first line of each test case contains an integer  $N$  denoting the number of elements in the given sequence.

The next line contains  $N$  space-separated integers denoting the sequence  $\text{arr}$ .

### Output Format:

For each test case, return a sequence of  $N$  integers representing the number of BSTs that can be formed using each element of the given sequence as the root node.

### Sample Input:

```
1
3
1 2 3
```

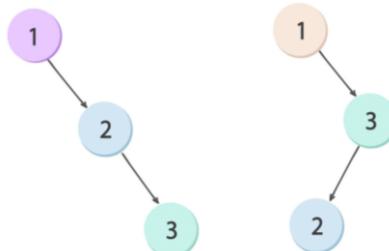
### Sample Output:

```
2 1 2
```

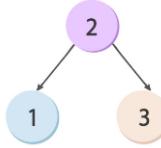
### Explanation:

The possible BSTs with the  $\text{arr}[i]$  as 0, 1, and 2 respectively are as follows:

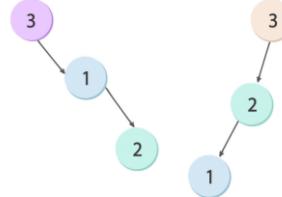
- There are two BSTs possible with 1 as a root node:



- There is one BST possible with 2 as a root node:



- There is one BST possible with 3 as a root node:



Hence we return the sequence: 2 1 2.

### Approach 1: Brute Force Approach

The Catalan number gives the number of binary search trees with N nodes.

For any element  $\text{arr}[i]$ , where  $\text{arr}[i]$  is the  $i^{\text{th}}$  element of the given sequence, to be the root of a binary search tree, it is necessary that the elements in its left subtree are less than  $\text{arr}[i]$  and elements in its right subtree are greater than  $\text{arr}[i]$ . Let the number of elements less than  $\text{arr}[i]$  in the given sequence be  $kL$ , and the number of elements greater than  $\text{arr}[i]$  in the given sequence be  $kR$ .

Then the number of unique BSTs with  $\text{arr}[i]$  as the root node will be given by the expression:

**Catalan( $kL$ ) \* Catalan( $kR$ )**, where **Catalan( $i$ )** denotes the Catalan number for the integer  $i$ .

**Catalan** numbers can be calculated using the following formula-

$$C_0 = 1 \quad \text{and} \quad C_{n+1} = \sum_{i=0}^n C_i C_{n-i}$$

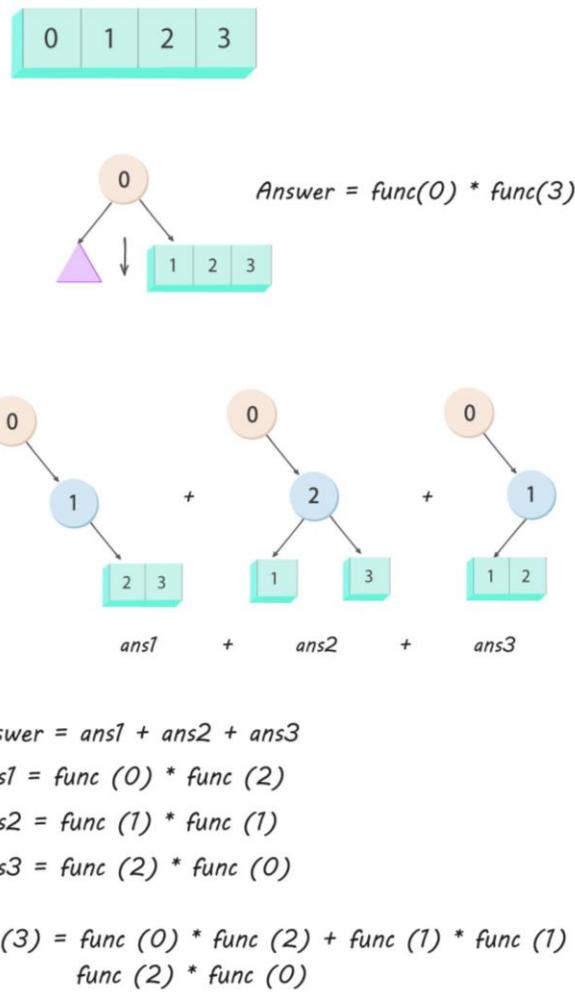
(this will make much more sense when we look at some example problems)

e.g.  $C_4 = C_0 C_3 + C_1 C_2 + C_2 C_1 + C_3 C_0$



The intuition behind the above formulae can be illustrated with the help of an example as shown below:

In the example, the number of BST's formed by taking 0 as the root node is discussed considering the input array be {0,1,2,3}:



### Steps:

1. Take a variable  $k$ , and for each  $k$  in  $0 \leq k < N$ , find the number of elements less than  $\text{arr}[k]$  where  $\text{arr}$  is the given sequence.
2. Let the number of elements less than  $\text{arr}[i]$  be  $s$ . Hence, the number of elements greater than  $\text{arr}[i]$  will be:

$$\begin{aligned}&\text{Total number of elements} - s - 1 \\ &= N - s - 1\end{aligned}$$

3. Find the Catalan number of  $s$  and  $N - s - 1$ . Return their product.

**Time Complexity:**  $O(N^2)$ , where  $N$  denotes the number of elements present in the sequence.

Each iteration will take order of **N** time as we need to iterate through total **N** elements. Finding the Catalan number takes an order of **N** time. Hence, the overall time complexity is the order of **N<sup>2</sup>**.

**Space Complexity:** **O(N)**, where **N** denotes the number of elements present in the sequence. Output takes a space of order of **N** as we need an array of **N** elements to store the output sequence.

### Approach 2: Optimised Approach

In the previous approach, we saw that calculating the number of elements less than and greater than the current element took linear time. Also, we calculated the Catalan number for each element in linear time for each element in the sequence. We can do the above operations faster if we precompute the factorials till  $2 * N$  in an array called **factorials**. This way, we can find the Catalan numbers in **O(1)** time.

For finding the number of elements less than and greater than the current element efficiently, we first make a copy of the given sequence and sort it. Then, we find the number of elements less than the current element in **O(logN)** by binary searching the current element in the sorted sequence.

#### Steps:

1. Create an array **factorials** and compute factorials of numbers till  $2 * N$ .
2. Make a copy of the given sequence and name it **sortedCopy** and sort it.
3. Iterate through the given sequence, and for each element, we find its position in **sortedCopy** using binary search.
4. Let **pos** be the position of the current element in the sorted array.
5. Find the Catalan number of **pos** and store it in **ansLeft**.
6. Find the Catalan number of  $N - pos - 1$  and store it in **ansRight**.
7. Finally, return the product of **ansLeft** and **ansRight**.

**Time Complexity:** **O(N \* log(N))**, where **N** is the length of the given sequence.

We need to iterate the sequence, and for each element, we need to search for its position in the sorted array using binary search, which takes an order of **log(N)** time. Hence, the overall time complexity is the order of **N \* log(N)**.

**Space Complexity:** **O(N)**, where **N** denotes the length of the given sequence. We are making two arrays—one of size **N** and one of size  $2 * N$ , hence the space complexity is the order of **N**.

## 7. Find Kth Smallest Element in BST [<https://coding.ninja/P74>]

**Problem Statement:** Given a binary search tree and an integer  $K$ . Your task is to find the  $K^{\text{th}}$  smallest element in the given BST (binary search tree).

**Input Format:**

The first line of input contains an integer  $T$  denoting the number of test cases.

The next  $T$  lines represent the  $T$  test cases.

The first line of input contains a single integer  $K$ .

The second line of input contains the tree elements in the level order form separated by a single space.

If any node does not have a left or right child, take -1 in its place.

**Output Format:**

For every test case, return the  $K^{\text{th}}$  smallest element of the given BST.

**Sample Input:**

1

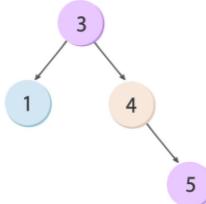
3

3 1 4 -1 -1 -1 5 -1 -1

**Sample Output:**

4

**Explanation:** The input BST is as follows:



The order of elements in the increasing order in the given BST is [1, 3, 4, 5]. Here  $K = 3$ , hence the 3<sup>rd</sup> smallest element is 4.

**Approach 1: Recursion In-order**

The basic idea is to maintain a global array, traverse the whole tree in an inorder manner, and add the node values in the array. Then, in the end, return the  $(k - 1)^{\text{th}}$  element of the array since the inorder of a BST is sorted in ascending order.

**Steps:**

1. Initialise a global array **arr**, which stores the elements of the BST.
2. Use an **inorder(root)** recursive function to add all the nodes in **arr** in the order.
3. In the end, return **arr[K-1]**.
4. If  $K$  is greater than the length of **arr**, then return -1.

**Time Complexity:**  $O(N)$ , where  $N$  denotes the number of nodes in the input BST.

**Space Complexity:**  $O(N)$ , where  $N$  denotes the number of nodes in the BST. We are using a linear space recursion call stack and an extra array.

### Approach 2: Iterative In-order

Here, the basic idea is to traverse the tree in an inorder manner with the help of iteration, so we don't need a size  $N$  call stack. A call stack is required to store all the calls from a recursive function that is currently being executed.

Whereas, by maintaining a stack ourselves and using an iterative methodology, we can optimise it to  $O(H)$  space, where  $H$  is the height of the BST.

#### Steps:

1. Maintain a stack  $s$ .
2. At any point in time, the current node is  $curr$ .
3. Initialise  $curr$  as the **root** of the BST.
4. Until  $curr$  becomes **NULL**, add the  $curr$  node to the stack  $s$  and update  $curr$  to  $curr \rightarrow left$  using a while loop.
5. If the stack  $s$  is empty, return -1.
6. Else, until  $s$  is not empty,
  - a. Pop the top element of  $s$ , call it  $curr$ , and decrement  $K$  by one, and make  $curr = curr \rightarrow right$ .
    - i. If  $K$  is zero, then return the current node's value.
    - ii. Else repeat the above algorithm from step 4.
7. If the stack is empty and  $K$  is not 0, then return -1.

#### Example:

The dry run of the above iterative inorder algorithm for the given input BST is depicted as below:

- Initially,  $s$  is empty.  $s = \{\}$  and  $curr$  is pointing to the root node.
- While  $curr \neq NULL$ , update  $curr$  to  $curr.left$  and add the nodes to the stack  $s$ . So  $s$  now becomes {1,3}
- Now  $curr$  is **NULL**, but  $s$  is not empty.
  - Pop the top of stack  $s$ , decrement  $k$  by 1 (the new value of  $k$  becomes 2) and update  $curr$  to the right child of the popped node.
  - Again  $curr$  is **NULL**, but  $s$  is not empty therefore repeating the above step.  $curr$  becomes 4,  $k$  becomes 1, and  $s$  becomes empty.
- Now repeat step 2 until  $curr$  becomes **NULL**.
- Finally when  $curr = NULL$  the stack  $s$  is not empty repeating step 3, we get the third smallest element in BST as 4.

**Time Complexity:**  $O(K + H)$ , where  $K$  is the given integer and  $H$  is the height of BST. We are iterating a loop until we reach the  $K^{th}$  element. Hence, before popping out one element from  $s$ ,

we have to go down in the left subtree until we reach a NULL node. Therefore, we need **H** extra time for this step.

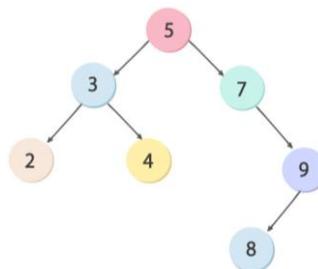
**Space Complexity:**  $O(H)$ , where **H** denotes the height of BST. We are maintaining a stack **s**, which can have at max **H** number of elements at a time.

### Approach 3: Morris Traversal

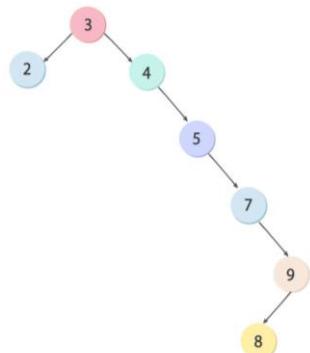
In this approach, we will use the Morris traversal algorithm, which is the space optimisation of approach 2.

#### Example:

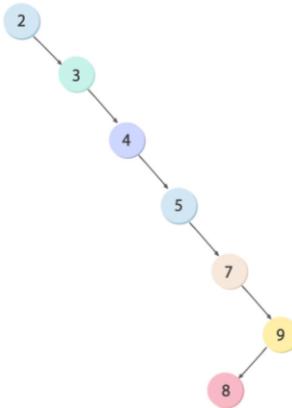
- Consider this binary search tree.



- Currently, node 5 is the root; node 3 is the left subtree. Find the rightmost node of node 3, which is 4.
- Build a connection between node 4 and root node 5.
- Break the connection between 5 and node 3, using **root → left = NULL**.
- The current new root is node 3.



- Repeat the same process until the left branch of the root node is NULL. Now the BST will look like this.



- Node 2 has no left subtree, so the first element of in-order traversal will be 2.
- The inorder for the above tree becomes { 2, 3, 4, 5, 7, 8, 9 }.

**Steps:**

1. Initialise, **curr** as **root**.
2. While **curr** is not NULL, we follow these steps:
  - a. If **curr** does not have left child:
    - i. Decrease **K** by one.
    - ii. If **K** becomes zero, then return **curr** node's value.
    - iii. Go to the right subtree, using **curr = curr → right**.
  - b. Else
    - i. In the **curr's** left subtree
      1. Assign **curr** as a right child of the rightmost child node. The rightmost child node can be found using a simple while loop.
      2. If the rightmost node is a leaf node:
        - a. Make a connection between the rightmost node and **curr**, and go to curr's left child using **curr = curr → left**.
    - iii. Else
      - a. Break the connection of **curr** with its subtrees.
      - b. Decrement **K** by one.
        - i. If **K** becomes zero, return **curr's** data.
        - c. Check **curr** node's right subtree, using **curr = curr → right**.
  3. If after traversing the whole BST **K** is not zero, return -1.

**Time Complexity:**  $O(K + H)$ , where **K** is the given integer and **H** is the height of BST. We are iterating a loop until we reach the  $K^{\text{th}}$  element. Since before popping out one element from the stack, we have to go down in the left subtree until we reach the NULL node, we need **H** extra time.

**Space Complexity:**  $O(1)$ , as we are using constant space.

## 8. Two Sum in a BST/ Pair Sum In a BST [<https://coding.ninja/P75>]

**Problem Statement:** You are given a binary search tree and a target value. You need to find out whether there exists a pair of node values in the BST, such that their sum is the same as the target value.

**Input Format:**

The first line of input contains a single integer **T**, representing the number of test cases or queries to be run.

Then the **T** test cases follow.

The **first line** of each test case contains elements in the level order format. The line consists of values of nodes separated by a single space. In case a node is null, we take -1 in its place.

The **second line** of each test case contains a single integer representing the target value.

**Output Format:**

For each test case, print True or False in a separate line.

**Sample Input:**

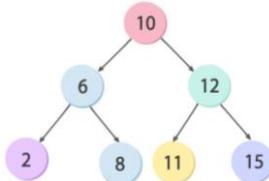
```
1
10 6 12 2 8 11 15 -1 -1 -1 -1 -1 -1 -1 -1
14
```

**Sample Output:**

True

**Explanation:**

The input BST is as shown:



The sum of the nodes with values 2 and 12 equals the target value of 14. Thus the output is True.

**Approach 1: Brute Force**

The idea here is to traverse the BST in a level-order manner. Let's denote the value of the current node as **A**. For each node, check whether the node with value (**target - A**) is present in the BST or not. We can search a node in the BST with this value using the BST property—that is, at any node we go to left subtree or right subtree by comparing the node value to (**target - A**), and if the value is found in the BST, return true else return false.

**Time Complexity:**  $O(N * H)$ , where **N** is the number of nodes, and **H** is the height of the given BST (Binary Search Tree). In the worst case, we will be traversing each node of BST  $O(N)$ , and for each node, we are searching in a given BST, which can take  $O(H)$  time. Hence, the overall time complexity will be  $O(N * H)$ .

**Space Complexity:**  $O(N)$ , where **N** denotes the total number of nodes in the given BST. In the worst case, almost all the tree nodes will be pushed into the queue (since we are traversing the tree in the level order manner), and extra space will be required for the recursion stack. Hence, the overall space complexity will be **linear  $O(N)$** .

### Approach 2: Using Set

In the previous approach, for each node in the BST, we searched again and again for a second node with the value (**target - value of the current node**) in the BST. We can optimise this search by using a **HashSet** to keep track of the elements of the BST that we have visited till now. We will traverse the tree using recursion, and for any current node of value **A**, we check if (**target - A**) is present in the HashSet. If so, we can say that there are two nodes with the values such that their sum is equal to the **target**, and we return true. Otherwise, we put the current node value inside the **HashSet**.

#### Steps:

1. Create an empty HashSet. Let's call it **st** to store the values of the nodes of the BST visited so far.
2. We create a recursive function that traverses each node in the given tree to check the pair sum:
3. If **root == null**, then return false.
4. If **st** contains **target - root.data**, then return true.
5. Else add **root.data** to **st**.
6. If either of the left or right subtrees return true, then return true.
7. Else return false.

**Time Complexity:**  $O(N)$ , where **N** denotes the number of nodes in the given BST (Binary Search Tree). In the worst case, we will be traversing each node of BST, and for each node value, we check the value (**target value - current node value**) in **HashSet**, which takes constant time. Hence, the overall time complexity will be  $O(N)$ .

**Space Complexity:**  $O(N)$ , where **N** denotes the number of nodes in a given BST(Binary Search Tree). In the worst case, we store each node value of BST in the **HashSet**, and extra space is required for the recursion stack. Hence, the overall space complexity will be  $O(N)$ .

### Approach 3: Using Two Pointers

We can use the property of the inorder traversal of the BST—that is, the inorder traversal of a BST always traverses the nodes in increasing order of their values. Therefore, we perform

inorder traversal on the BST and store the values in an auxiliary array. As this array is sorted, we can use the standard two-pointer technique on this sorted array to determine if two elements exist in the array, which sums up to the **target** value.

#### Steps:

1. Create an empty array, let's say **inorderArray**, to store BST values in the inorder manner.
2. Create an **inorder** function and pass the **root** and **inorderArray** as two arguments and store the value of each node in the array.
3. Then take two pointers, let's say *i* and *j*, where *i* points to the start of the array and *j* points to the end of the array.
4. Run a loop, while  $i < j$ , and do:
  - a. Add the value of **inorderArray[i]** and **inorderArray[j]**. If this sum equals the **target** value, we return **true**.
  - b. If the sum is greater than the **target** value, then decrease *j* by one.
  - c. Else increase *i* by one.
5. At last, if there is no pair, we return **false**.

**Time Complexity:** **O(N)**, where **N** denotes the number of nodes in the given BST (Binary Search Tree). In the worst case, we will be traversing each node of BST, which is of the order of **O(N)** and traversing the **inorderArray**, which is again the order of **O(N)** time. Hence, the overall time complexity will be **O(N)**.

**Space Complexity:** **O(N)**, where **N** denotes the number of nodes in a given BST (Binary Search Tree). In the worst case, we store each node of BST in the **inorderArray**, and extra space is required for the recursion stack. Hence, the overall space complexity will be **O(N)**.

#### Approach 4: Two Pointer on BST

The idea here is the same as the previous approach with space optimisation, by applying the two-pointer technique on the given BST itself rather than creating an auxiliary array of node values. In this approach, we will maintain a forward and a backward iterator that will iterate the BST in the order of in-order and reverse in-order traversal, respectively, using two stacks.

One stack (**start**) will be made by maintaining the leftmost value of the BST, and the other stack (**end**) will be made by maintaining the rightmost value in the BST. Now, using the two-pointer technique, we check if the sum of values at the **start** and **end** nodes are equal to the **target** or not. If they are equal to the **target** value, then return true. If the values are lesser than the **target** value, we make the forward iterator point to the next element using stack. Else, if the values are greater than the **target** value, we make a backward iterator point to the previous element using the second stack.

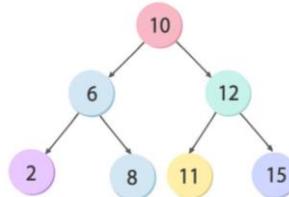
At last, if we find no such two elements, then return false.

#### Steps:

1. Create two empty stacks, **start** and **end**, to store the nodes in normal inorder and reverse inorder traversal respectively.
2. Take a node, let's say **currNode** and make it point to the root.
3. While the **currNode** is not NULL, do:
  - a. Push the **currNode** to **start** stack.
  - b. **currNode = currNode → left.**
4. After the loop is over, make **currNode** point to **root** again and while the **currNode** is not NULL, do:
  - a. Push the **currNode** to the **end** stack.
  - b. **currNode = currNode → right.**
5. While the top element of both the stacks are not equal, do:
  - a. Create two variables **val1** and **val2**, to store the top element of the stack, **start** and **end**, respectively.
  - b. If the sum of **val1** and **val2** is equal to the **target**, then return true.
  - c. If the sum of **val1** and **val2** is less than the **target**, then make the **currNode** point to the right child of the top element of the **start** stack and pop this node from the **start** stack.
  - d. While the **currNode** is not NULL, do:
    - I. Push the **currNode** to **start** stack.
    - II. **currNode = currNode → left.**
  - e. Else if the sum of **val1** and **val2** is greater than the **target**, then make the **currNode** point to the left child of the top element of the **end** stack and pop this node.
  - f. While the **currNode** is not NULL, do:
    - I. Push the **currNode** to the end stack.
    - II. **currNode = currNode → right.**
6. Finally, return false if we come out of the loop.

### Example:

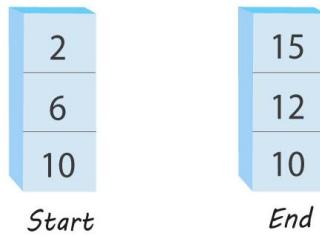
Let us dry run the above algorithm for the following example :



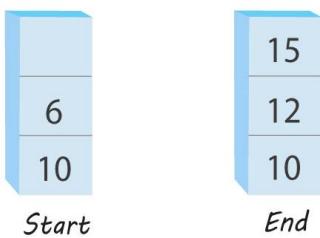
Find if there exist any two nodes, such that the sum of their values is equal to 20 in the shown BST.

Step 1: Initially, both the **start** and **end** stacks are empty, and the **currNode** pointer points to the root node of the BST.

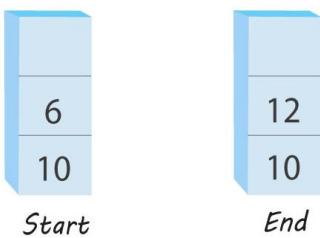
Step 2: Following step 3 and step 4 of the mentioned algorithm, we have the start and end stacks as follows:



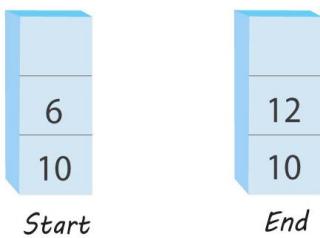
Step 3:  $2 + 15 < 20$ , therefore the currNode pointer points to the right child of node 2 and adds elements onto the start stack until the currNode becomes NULL (step 5d of the algorithm).



Step 4:  $6+15 > 20$ ; therefore, the currNode pointer points to the left child of the node 15 and adds elements onto the end stack until the currNode becomes NULL (step 5f of the algorithm).



Step 5:  $6 + 12 < 20$ , therefore the currNode pointer points to the right child of the node 6 and adds elements onto the start until the currNode becomes NULL (step 5d of the algorithm).



Step 6:  $8 + 12 = 20$ , hence we found a valid pair of nodes whose sum of values is equal to the required number. Therefore return true.

**Time Complexity:**  $O(N)$ , where  $N$  denotes the number of nodes in the given BST (Binary Search Tree). In the worst case, we will be traversing each node of BST. Hence, the overall time complexity will be  $O(N)$ .

**Space Complexity:**  $O(H)$ , where  $H$  denotes the height of the given BST (Binary Search Tree). In the worst case, we store total elements equal to the height of BST in the stack. Hence, the overall space complexity will be  $O(H)$ .

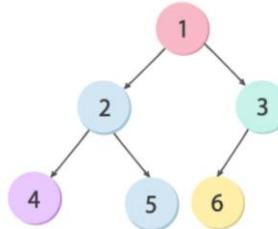
# 9. Priority Queue and Heaps

---

## Heaps

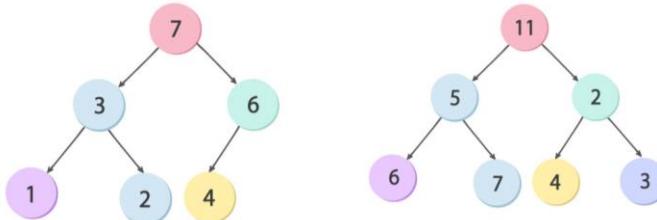
- A heap is a binary tree with two special properties.
  - Firstly, the value of a node must be greater than equals to or less than equals to the values of its children. This is called the **heap property**.
  - Secondly, the heap should always be a **complete binary tree**—that is, the elements in a heap are always inserted in the **last level from left to right**.
- The second property for the heap ensures that all the leaf nodes in a heap would lie in the  **$h$  or  $h-1$**  level, where  $h$  is the height of the heap.

### Example of a min-heap



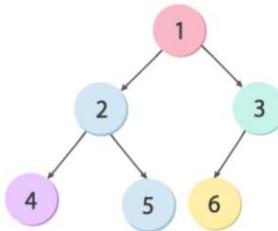
In the examples below, the left tree is a heap since each element is greater than its children, and it is also a complete binary tree.

The right tree is not a heap as 11 is greater than 2 and 5, whereas the rest of the nodes are less than their children, thus violating the heap property.

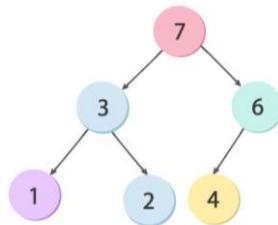


## Types of Heap

- **Min heap:** The value of every node must be less than or equal to its children.



- **Max heap:** The value of every node must be greater than or equal to its children.

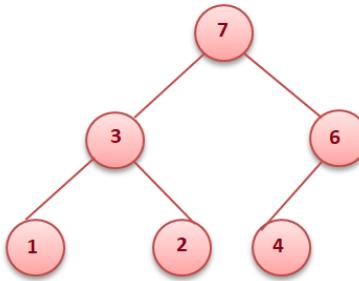


### Representing heaps

Before looking at the various heap operations, let us see how heaps can be represented or stored in memory. One possibility is using arrays. Since heaps form complete binary trees, there will not be any wastage of locations or memory in the array used for storing the heap elements. For the following discussion, let us assume that elements are stored in an array, starting at **index 0**. The previous max heap can be represented as:

Index	0	1	2	3	4	5
7	3	6	1	2	4	
Index	0	1	2	3	4	5

**Parent of a Node:** For a node at index  $i$ , its parent is at index  $(i - 1)/2$ . In the figure below, element 6 is at the 2<sup>nd</sup> index and its parent is at the 0<sup>th</sup> index—that is, for the element at  $i = 2$ , the parent's position =  $(i - 1)/2 = (2 - 1)/2 = 0$ .



**Children of a Node:** For a node at index  $i$ , its children are at index  $(2 * i + 1)$  and  $(2 * i + 2)$ . For example in the given figure, 3 is at index 1 and has children 1 and 2, at index 3 ( $2 * 1 + 1$ ) and 4 ( $2 * 1 + 2$ ) respectively .

**Maximum/minimum element:** The maximum element in a max heap, or the minimum element in a min-heap, is always the root node of the heap. It will be stored at the 0<sup>th</sup> index.

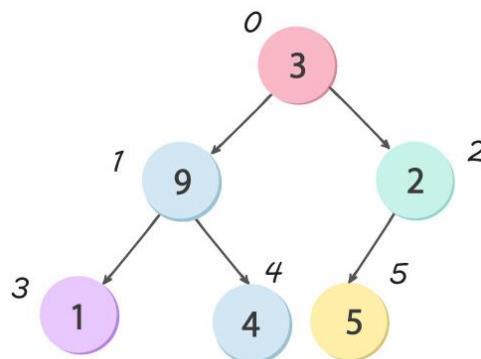
## Heapify

Heapify is defined as the process of creating a heap data structure from a binary tree. It is used for creating a min or a max heap.

1. Let the input array be as shown here.



2. Create a complete binary tree from the array.



3. Start from the first index of the non-leaf node whose index is given by  $n/2 - 1$ .
4. Set current element  $i$  as **largest**.
5. The indices of the left child and right child are  $2i + 1$  and  $2i + 2$ , respectively.

6. If **leftChild** is greater than **currentElement**—the element at the *i*th index, set **largest** as **leftChildIndex**.
7. If **rightChild** is greater than the element in **largest**, set **largest** as **rightChildIndex**.
8. Swap **largest** with **currentElement**.
9. Repeat steps 3–7 until the **largest** becomes equal to the **currentElement** indicating the subtrees are also heapified.

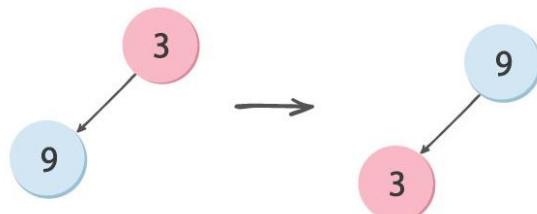
For example, the following are the steps to form a max-heap from the given input array :

3	9	2	1	4	5
0	1	2	3	4	5

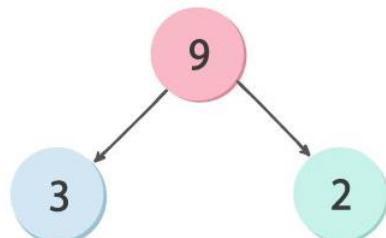
**Step 1:** Insert 3 into the max-heap. Heap looks like:



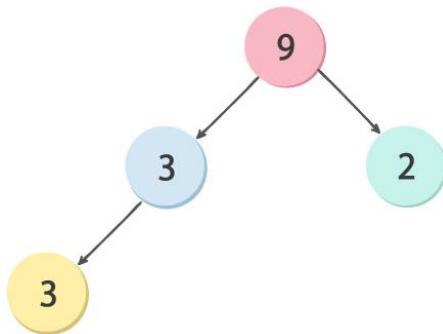
**Step 2:** Insert 9 to the max-heap . The parent of 9, that is, 3, is smaller than 9, therefore swapping the two elements.



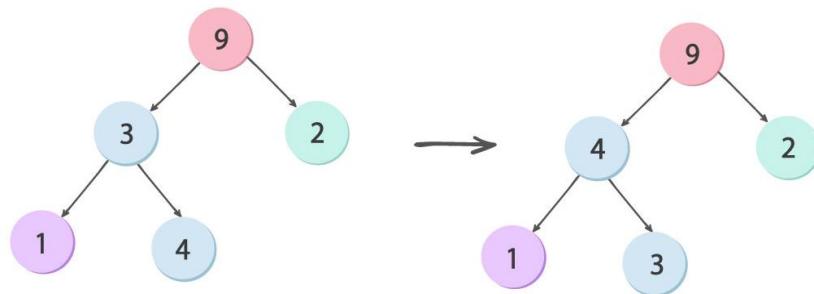
**Step 3:** Insert 2 to the max-heap. Heap looks like:



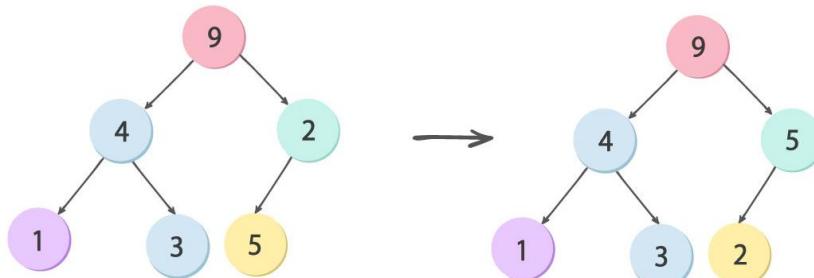
**Step 4:** Insert 1 to the max-heap. Heap looks like:



**Step 5:** Insert 4 to the max-heap. The parent of 4, that is, 3, is smaller than 4, therefore swapping the two elements.



**Step 6:** Insert 5 to the max-heap. The parent of 5, that is, 2, is smaller than 5, therefore swapping the two elements.



**Note:** The above steps are for Max-Heap. For Min-Heap, both **leftChild** and **rightChild** must be **greater** than the parent for all nodes.

Below is the pseudo-code for the heapify process.

```
function heapify(int i)
    largest = i
    l = 2 * i + 1           // Index of Left Child
    r = 2 * i + 2           // Index of Right Child

    if l < n && heap[i] < heap[l]
```

```

largest = i

if (r < n && heap[largest] < heap[r])
    largest = r

if largest is not equal to i
    temp = heap[i]
    heap[i] = heap[largest]
    heap[largest] = temp
    heapify(largest)

```

## Inserting into a Heap

Insertion into a heap can be done in two steps:

- Insert the new element at the end of the heap to preserve the complete binary tree property of the heap.
- Move that element to its correct position in a heap using the **heapify** (upheapify) process.

Upheapify refers to the process of re - heapifying the heap after the insertion of the new element present at the last position of the complete binary tree.

```

function insert(element) {

    // add an element to the end of the heap list.
    heap.add(element)

    childIndex = heap.length - 1
    parentIndex = (childIndex - 1) / 2

    while(childIndex > 0)
        if heap[parentIndex] < heap[childIndex]      // swap the elements

            temp = heap[parentIndex]
            heap[parentIndex] = heap[childIndex]
            heap[childIndex] = temp
            childIndex = parentIndex
            parentIndex = (childIndex - 1) / 2

        else
            break

```

## Delete Max Element from Max Heap

Since the max element in the max heap is always present at the 0th index, therefore, follow the following steps to delete the max element from the heap:

- Swap the 0th element with the last element.
- Remove the last element from the heap.
- Heapify the 0th element (**downheapify** process). Downheapify refers to the process of re-heapifying the heap after the deletion of the element at the last position of the complete binary tree.

If you want to return the max element, then store it before replacing it with the element at the last index, and return it in the end.

```
function removeMax(){

    if heap is empty
        print "heap is empty"
        return

    retVal = heap[0]
    heap[0] = heap[heap.length - 1]
    heap.remove(heap.length - 1)

    if(heap.size() > 1)
        heapify(0)

    return retVal
}
```

## Introduction to Priority Queues

- Priority queues are abstract data structures where each data/value in the queue has a certain priority.
- A priority queue is a special type of queue in which each element is served according to its priority and not merely by the **FIFO** principle of a queue.
- If elements with the same priority occur, they are served according to the order in which they are present in the queue.
- A priority queue is called an **ascending-priority queue** if the element with the smallest key has the highest priority. Similarly, a priority queue is called a **descending-priority queue** if the element with the largest key has the highest priority.

- Since the two operations are symmetric, we will be discussing ascending priority queues.

### Difference between Priority Queue and Normal Queue

A queue follows the **First-In-First-Out(FIFO)** principle. In a priority queue, the values are removed based on the **priority of the elements**. The element with the highest priority is removed first.

## Operation on Priority Queues

### Main Priority Queue Operations

- **Insert (key, data):** Inserts data with a priority to the priority queue. Elements are ordered based on priority.
- **DeleteMin/DeleteMax:** Remove and return the element with the smallest/largest priority.
- **GetMinimum/GetMaximum:** Return the element with the smallest/largest priority without deleting it.

### Auxiliary Priority Queues Operations

- **kth-Smallest/kth-Largest:** Returns the element with the kth-smallest/kth-largest priority element.
- **Size:** Returns the number of elements present in the priority queue.
- **Heap Sort:** Sorts the elements in the priority queue based on priority.

## Ways of Implementing Priority Queues

Priority queue can be implemented using an array, a linked list, a heap data structure, or a binary search tree.

Among these data structures, the heap data structure provides the most efficient implementation of priority queues. Hence, we will be using the heap data structure to implement the priority queue.

Before discussing the actual implementations, let us enumerate the possible options available to us.

- **Unordered Array Implementation:** Elements are inserted into the array without bothering the order. Deletions are performed by searching the minimum or maximum priority element and then deleting.

- **Unordered List Implementation:** It is similar to array implementation, but instead of an array, this implementation uses linked lists to store the elements.
- **Ordered Array Implementation:** Elements are inserted into the array in a sorted order based on the priority field. Deletions are performed only at one end of the array.
- **Ordered list implementation:** Elements are inserted into the list in sorted order based on the priority field. Deletions are performed at only one end of the list, hence preserving the status of the priority queue. All other functionalities associated with a linked list are performed without modification.
- **Binary Search Trees Implementation:** Insertion and deletions are performed in such a way that the property of BST is preserved. Both insertion and deletion operations take  $O(\log n)$  time on average and  $O(n)$  time in the worst case.
- **Balanced Binary Search Trees Implementation:** Insertion and deletion operations are performed such that the property of BST is preserved—that is, the balancing factor of each node is  $-1$ ,  $0$ , or  $1$ . Both the operations take  $O(\log n)$  time.
- **Binary Heap Implementation:** Both the operations of Insertion and Deletion in the Binary Heaps are of the order of the height of the tree that is  $O(\log n)$  time, whereas the operation of accessing the minimum or the maximum element is carried out in constant time as it is always present at the root.

## Time Complexities of Various Implementations

Implementation	Insertion	Deletion (Delete max/min)	Find (max/min)
Unordered Array	$O(1)$	$O(n)$	$O(n)$
Unordered List	$O(1)$	$O(n)$	$O(n)$
Ordered Array	$O(n)$	$O(1)$	$O(1)$
Ordered List	$O(n)$	$O(1)$	$O(1)$
Binary Search Trees	$O(\log n)$ average	$O(\log n)$ average	$O(\log n)$ average
Balanced Binary Search Trees	$O(\log n)$	$O(\log n)$	$O(\log n)$
Binary Heaps	$O(\log n)$	$O(\log n)$	$O(1)$

## Insertion and Deletion in a Priority Queue

The first step would be to represent the priority queue in the form of a max/min-heap. Once it is heapified, the insertion and deletion operations in the priority queue can be performed similarly to that in a heap.

### Heap Sort

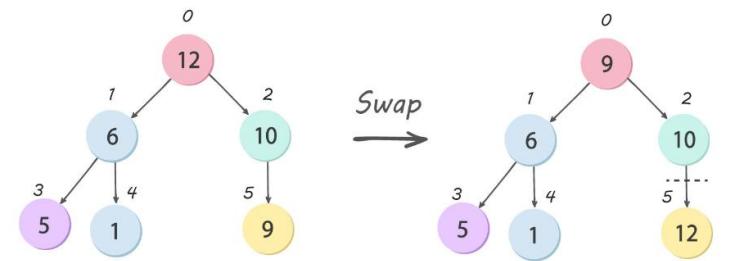
- Heap Sort is another example of an efficient sorting algorithm. Its main advantage is that even its worst-case runtime complexity is of **O(nlog(n))** order regardless of the input data.
- Heap sort is one of the simplest sorting algorithms to implement apart from its additional advantage of being a fairly efficient algorithm.

### Algorithm

1. The heap-sort algorithm inserts all elements (from an unsorted array) into a maxheap.
2. Note that heap sort can be done in-place within the array given to be sorted, and it is referred to as **in-place-heap-sort**.
3. Since the tree satisfies the max-heap property, the largest element is always stored at the root node.
4. **Swap:** The root element is swapped with the element at the last index ( $n - 1$ ).
5. **Remove:** Reduce the size of the heap by 1.
6. **Heapify:** Heapify the root element again to have the highest element at the root (**downheapify** process).
7. The process is repeated until all the items in the list are sorted (repeated n times).

### Example:

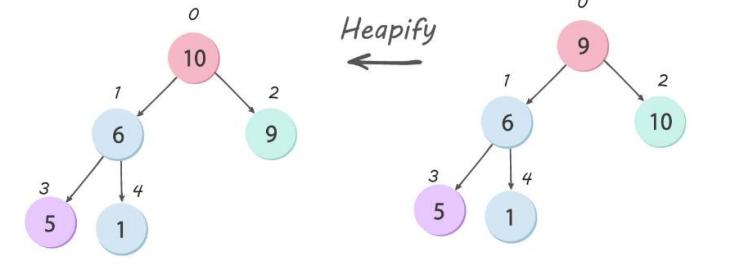
Below is the dry run of the above heap sort algorithm when the unsorted array given as an input is [12,6,10,5,1,9].



Swap



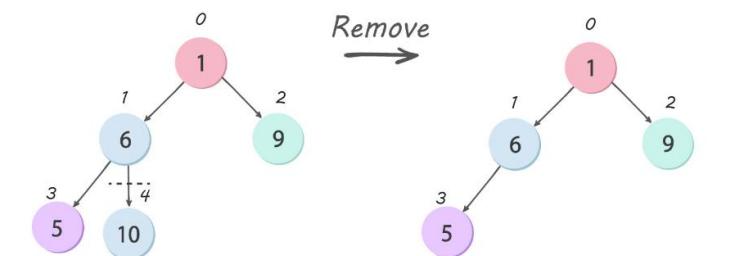
Remove



Heapify

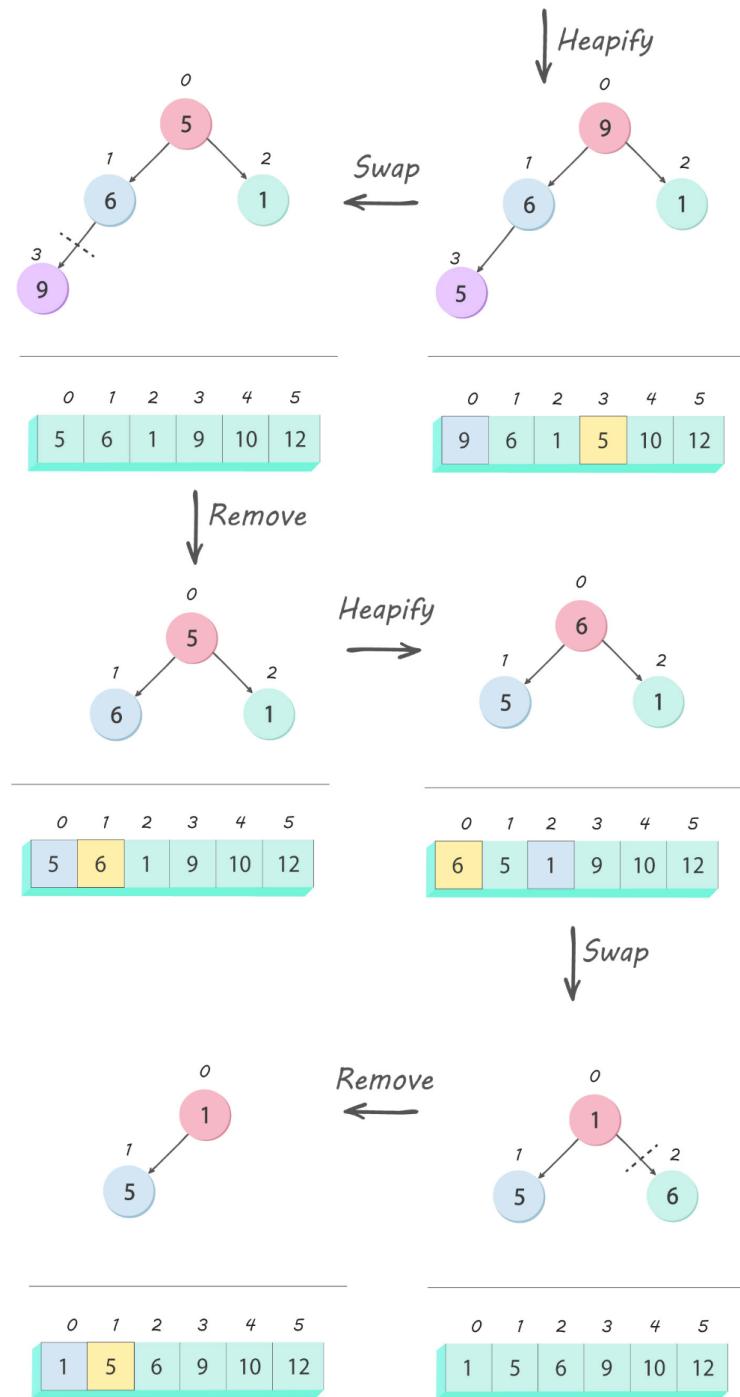


Swap



Remove





Below is the pseudo code for the above heap sort algorithm :

```

function heapSort(heap, startIndex, endIndex)

    // making a max heap of the given input unsorted array

    i = heap.length/2 - 1
    while i is greater than or equals to 0
        heapify(input, i, input.length)
        i--
    // heap sort

    n = heap.length
    i = n - 1
    while i is greater than equal to 0
        // Move current root to end
        temp = heap[0]
        heap[0] = heap[i]
        heap[i] = temp
        i--

    // call heapify on the reduced heap
    heapify(input, 0, i)

// heapify pseudo code

function heapify(heap, index, arrLength)

    largest = index
    left = 2 * index + 1
    right = 2 * index + 2

    if left < arrLength and heap[left] > heap[largest]
        largest = left

    if right < arrLength and input[right] > input[largest]
        largest = right

    if largest != index
        k = input[index]
        input[index] = input[largest]
        input[largest] = k
        heapify(input, largest, arrLength)

```

## Applications of Priority Queues

Following are some of the applications of priority queues:

- They are used in Huffman Coding Algorithm, which is used for data compression.
  - Used in Dijkstra's Algorithm (a shortest path algorithm)
  - Used in Prim's algorithm (a minimum spanning tree algorithm)
  - Used in event-driven simulations like customers in a line.
  - Used in selection problems such as  $k^{\text{th}}$  - smallest element.
- 

## Practice Problems

### **1. Is Binary Heap Tree** [<https://coding.ninja/P76>]

**Problem Statement:** Given a binary tree. You need to check if it is a binary heap tree or not.

A binary tree is a tree in which each node has at most two children.

A binary heap tree has the following properties.

1. It must be a complete binary tree; in the complete binary tree, every level, except the last level, is completely filled and the last level is filled as far left as possible.
2. Every node must be greater than all its children nodes.

#### **Note:**

There are two types of binary heap trees:

- a. **Min-heap** - if all the nodes have their values less than its children nodes.
- b. **Max-heap** - if all the nodes have their values greater than its children nodes.

In this problem, consider the binary heap tree to be a **binary max-heap tree**.

#### **Input Format:**

The first line contains an integer **T** denoting the number of test cases.

For each test case:

Each line contains the elements of the tree in the **level order** form separated by a single space. Whenever a node is NULL, -1 is used in its place.

#### **Output Format:**

For every test case, return **True** or **False**.

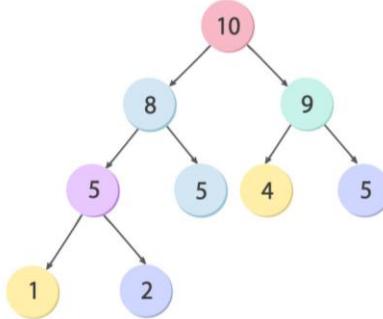
#### **Sample Input:**

10 8 9 5 5 4 5 1 2 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1

### Sample Output:

True

**Explanation:** The input binary tree can be represented diagrammatically as follows:



- (a) It is a complete binary tree because every level except the last level is completely filled, and every node of the last level is as far left as possible. Thus the first condition of a heap binary tree is fulfilled.
- (b) Secondly, we need to check if every node has a value greater than its children nodes or not.

Here we see that:

Node 10 is greater than 8, 9, 5, 5, 4, 5, 1, and 2.

Node 8 is greater than 5, 5, 1, and 2.

Node 9 is greater than 4 and 5.

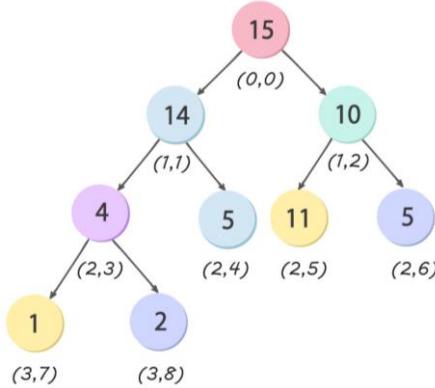
Node 5 is greater than 1 and 2.

All the other nodes have no children.

Since all the nodes have values that are greater than its children nodes, the second condition of a heap tree is also fulfilled. Thus the given input binary tree is a binary max-heap tree.

### Approach 1: Recursion

Consider the binary tree represented by the input array as: {15, 14, 10, 4, 5, 11, 5, 1, 2, -1, -1, -1, -1, -1, -1}



**Note:** The elements begin from index 1; therefore, for any node at position  $i$ , its left child would be at position  $2*i + 1$  and the right child would be at position  $2*i + 2$ . In the above figure, (3,7) indicates that the level of the node (1) is 3 and that its index is 7.

The idea here is that we check for both the properties of a binary heap tree one by one. First, we check if the given tree is a complete binary tree, and then we check whether all the nodes have their values greater than their children nodes or not.

#### Steps:

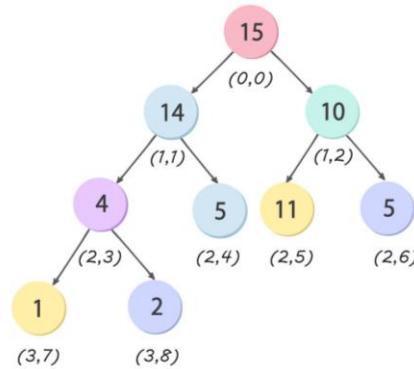
1. Checking if the given input binary tree is a complete binary tree or not:
  - a. In the array representation of a binary tree, if a node is assigned an index of  $i$ , then the left child gets assigned an index of  $2*i + 1$  while the right child is assigned an index of  $2*i + 2$ . Assign indices to the different nodes of the tree above from top to down and left to right.
    - i. Count the number of nodes in the binary tree.
    - ii. Start with root node assigning it index 0 and then recursively check for the left and right child indices respectively for all the nodes.
  - b. If, at any point in time, the examined node's assigned index is greater than the total number of nodes, then it is not a complete binary tree and thus returns **False**.
  - c. In the end, if all nodes' assigned indices are less than the number of nodes in the tree, then it is ensured that the given tree is a complete binary tree.
2. Given a binary tree is a complete binary tree, now check for the second heap property—that is, every node must be greater than its children nodes.
  - a. Start from the root and check for every node.
    - i. If, at any point in time, the parent node is less than any of its children nodes, then return **False** because it is not holding the binary max-heap property.
    - ii. Else recursively check for its left child and right child.
3. If the binary tree also holds the second property of the binary heap tree, then we can be sure that the given tree is a binary heap tree. Thus return **True**.

**Time Complexity:**  $O(N)$ , where  $N$  denotes the number of nodes in the binary tree. We access all the nodes of the tree in linear time.

**Space Complexity:**  $O(N)$ , where  $N$  denotes the number of nodes in a binary tree. We are calling every node recursively.

### Approach 2: Breadth-first search

Consider the binary tree represented by the array:  $\{15, 14, 10, 4, 5, 11, 5, 1, 2, -1, -1, -1, -1, -1, -1, -1\}$



The idea is that we place all nodes (**position**) as shown in the above figure.

Suppose we are at a node (**position**), then the left child of this node will be at  $(2 * \text{position} + 1)$ , and the right child of this node will be at  $(2 * \text{position} + 2)$ .

In the end, we check if the last node position is equal to the number of nodes in the given tree. It will ensure that the given binary tree is a complete binary tree. For binary heap trees, we check if every node is greater than both its left and right child.

#### Steps:

1. We maintain a queue/array of pairs called **nodesArr** in which we store the pair (node and their positions) of nodes.
2. Initially, **nodesArr** will have only one pair (root, 1), where the root is the **root** node, and 1 is the position of the root node.
3. We iterate **nodesArr** with  $i$  as the iterator of the loop
  - a.  $\text{curNode} = \text{nodesArr}[i]$ 
    - i. **Current node** is **curNode.first**, and **the position** is **curNode.second**
4. If **curNode** is greater than both its children, insert both the child nodes into **nodesArr** that is  $(\text{curNode} \rightarrow \text{left}, \text{position} * 2)$  and  $(\text{curNode} \rightarrow \text{right}, \text{position} * 2 + 1)$ .
5. If **curNode** is not greater than its children, return **False** because it doesn't hold the property of the binary max-heap tree anymore.

6. In the end, after iterating all the nodes of the binary tree or **nodesArr**, check if the size of **nodesArr** is equal to the last nodes' **nodesArr.second** value.
  - a. If both values are equal, then return **True**.
  - b. Else return **False**.

**Time Complexity:**  $O(N)$ , where **N** denotes the number of nodes in a binary tree. We are iterating through the binary tree once.

**Space Complexity:**  $O(N)$ , where **N** denotes the number of nodes in a binary tree. We are using an array/vector of pairs to store every node.

## **2. Convert Min-Heap to Max-Heap** [<https://coding.ninja/P77>]

**Problem Statement:** You are given an array/list (arr) representing a min-heap. Your task is to write a function that converts the given min-heap to a max-heap.

### **Input Format:**

The **first line** contains an integer **t** which denotes the number of test cases or queries to be run. For each test case:

The **first line** contains an integer **N** denoting the size of the array/list.

The **second line** contains **N** single space-separated in a heap integers representing the elements in the array/list which form a min-heap.

### **Output Format:**

For each test case or query, if the built max-heap satisfies the max-heap conditions, print **true**. Otherwise, print **false**.

### **Sample Input:**

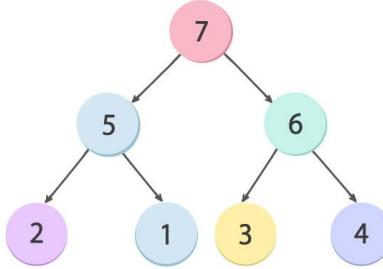
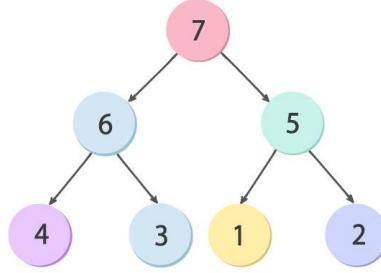
```
1
7
1 2 3 4 5 6 7
```

### **Sample Output:**

```
true
```

### **Explanation:**

For a given min-heap, there can be multiple max heaps. For example, for the above input, below are the two possible max-heaps:



Since it is possible to create a max-heap, the answer should be true.

### Approach 1: Brute Force Approach

For this approach, we will first copy the elements of the min-heap array to a soon-to-be-max-heap array using a 'for' loop. After copying each element to the soon-to-be-max-heap array, we will heapify it to move all its elements to the correct positions.

Following is the procedure to heapify:

1. For every index in the heap, find its left child and right child. This can be done by calculating  $2i + 1$  and  $2i + 2$ , where  $i$  is the index of the current element (0-based indexing).
2. Then find the maximum element among the left and right children and the current element.
3. If the current element is the maximum, we will move on to the next index; otherwise, we will swap the current element with the maximum element and recurse the heapify function on the maximum element index.

**Time Complexity:  $O(N \log(N))$** , Where **N** denotes the total number of elements in the heap. Since we're performing the heapify process for every element in the array,  **$O(N \log(N))$  is the time complexity**.

**Space Complexity:  $O(N)$**  as we are making a new array for the max heap.

### Approach 2: Rightmost Layer Of Min Heap

We will convert the min-heap to a max-heap by heapifying all the elements in the heap starting

from the rightmost node of the bottom-most layer of the min-heap. Let  $n$  be the size of the input heap. Following is the procedure for heapifying:

1. For every index in the heap from  $i = n/2 - 1$  to  $i = 0$ , find its left child and right child. This can be done by calculating  $2i + 1$  and  $2i + 2$ , where  $i$  is the index of the current element (0-based indexing).
2. Then find the maximum element among the left and right children and the current element.
3. If the current element is the maximum, we can move on to the next index; otherwise, we will swap the current element with the maximum element and recurse the heapify function on the maximum element index.

**Time Complexity:  $O(N)$**  You may go through [this](#) stack overflow page for a detailed explanation to understand this time complexity.

**Space Complexity:  $O(\log(N))$**  for the recursion stack.

### **3. Kth Largest Element in the Unsorted Array** [<https://coding.ninja/P78>]

**Problem Statement:** You are given an array consisting of  $N$  distinct positive integers and a number  $K$ ; your task is to find the  $K^{\text{th}}$  largest element in the array.

**Note:**

1.  $K^{\text{th}}$  largest element in an array is the  $K^{\text{th}}$  element of the array when sorted in non-increasing order. For example, consider the array [2,1,5,6,3,8] and  $K = 3$ , the sorted array will be [8,6,5,3,2,1], and the 3rd largest element will be 5.
2. All the array elements are pairwise distinct—that is, all the elements in the input array are unique.

**Input Format:**

The first line of the input contains an integer  $T$  denoting the number of test cases or queries to be run.

The first line of each test case contains two space-separated integers  $N$  and  $K$ , as described in the problem statement.

The second line of each test case consists of  $N$  integers separated by a single space, representing the given input array.

**Output Format:**

The only line of output of each test case consists of an integer—that is, the  $K^{\text{th}}$  largest element of the array.

**Sample Input:**

1  
4 2

5 6 7 8

#### Sample Output:

7

#### Explanation:

Here 7 is the second-largest element in the array [8,7,6,5].

#### Approach 1: Brute Force

The most obvious brute force approach would be to sort the array in descending order and return the **K<sup>th</sup>** element from the beginning of the array.

1. Sort the array in descending order. For sorting, most of the languages have their inbuilt sort methods, which are usually very efficient.
2. After sorting, return the element **arr[K - 1]** considering 0-based indexing.

**Time Complexity: O(N\*log(N))**, where **N** denotes the size of the array, as the inbuilt sort functions have a runtime complexity of **O(N\*log(N))**.

**Space Complexity: O(1)**, as we are using constant extra memory.

#### Approach 2: Priority Queue(Min-Heap):

We can achieve better time complexity than in approach 1 by using some specific data structures. Notice the fact that if there are only **K** elements in the array, then the **K<sup>th</sup>** largest element of the given input array (of size **N**) will be nothing but the smallest element of this **K** sized array. This observation clearly hints at using heaps as they help find the minimum or maximum element efficiently. We will be using a min-heap since we are interested in **the smallest element** among the **K** largest elements of the array.

1. Maintain a priority queue giving priority to the min element (min-heap), and insert the first **K** elements of the array into the priority queue.
2. Now traverse the remaining **N - K** elements one by one and check if the element at the top of the priority queue is smaller than the traversed element, then pop the element present at the top of the queue and insert the element being traversed in the priority queue.
3. Do the same for all the remaining **N - K** elements. We want to make sure that, at any **instance, our queue will only have K elements**.
4. After performing the above operation, return the element at the top of the priority queue, which will be our answer. Because after performing the above operations, the priority queue will be containing the **k** largest elements of the array. And the **min** of these **K** elements will be the **K<sup>th</sup>** largest element which will be present at the top of the priority queue (min-heap).

#### Example:

Let's examine the working of the above algorithm with the help of an example:

Consider the array [2, 3, 4, 8, 1, 7, 5, 6] and let **K = 4**.

Let's take an empty priority queue(min-heap).

- After performing the first step of the algorithm, the queue will be:  
2-3-4-8 (in the same order—that is, 2 will be at the front of the queue and 8 at the end).
- Now we start traversing the rest of the  $N - K$  elements—that is, step 2 of the algorithm.
- We start from 1 since it is smaller than 2, so we will continue.
- Then we come to 7 since it is greater than 2, so we will pop the element from the top(2) and push 7 into the queue. Now, the queue will be 3-7-4-8 (in the same order).
- Then we come to 5 since it is greater than 3, so we will pop the element from the top of the queue(3) and push 5 into the queue. Now, the queue will be 4-5-8-7 (in the same order).
- Then we come to 6 since it is greater than 4, so we will pop the element from the top of the queue(4) and push 6 into the queue. Now, the queue will be like 5-6-8-7 (in the same order).
- After completing this step, we can observe that our queue contains the four largest elements of the array, with the 4th largest element at the top of the queue.
- Now we return the element at the top of the queue (5), which is our answer.

**Time Complexity:**  $O(K + (N-K)*\log(K))$ , where **N** denotes the size of the array, and **K** is the order of the largest element to be found.

**Space Complexity:**  $O(K)$ , where **K** is the order of the largest element to be found.

## 4. Minimum K Product [<https://coding.ninja/P79>]

**Problem Statement:** You are given an array **arr** of **N** positive integers and a positive integer **K**. You need to find the minimum product of any **K** integers of the given array. Since the answer could be very large, you need to return the product modulo  $10^9 + 7$ .

**For Example:** If the given array is [1, 4, 2, 6, 3] and  $K = 3$ , then your answer should be 6 by taking the product of 3 integers which are 1, 2, and 3.

### Input Format:

The **first line** contains a single integer **T**, representing the number of test cases or queries to be run.

For each test case:

The **first line** contains two positive integers **N** and **K**, where **N** is the size of the given input array and **K** is the number of elements of the array of which the minimum product is to be found.

The **second line** contains **N** single space-separated positive integers representing the elements of the array.

### Output Format:

For each test case, print the minimum product of **K** integers from the array modulo  **$10^9 + 7$**  in a single line.

**Sample Input:**

```
1
5 3
5 2 3 8 3
```

**Sample Output:**

```
18
```

**Explanation:**

The minimum product of three integers (2, 3, 3) is 18. Thus 18 is the output.

**Approach 1: Sorting**

1. Sort the given array in increasing order.
2. Initialise a variable **ans** to one.
3. Run a loop from **0** to **K - 1** index and store the product of elements in the **ans** variable by taking the modulo at each multiplication.
4. Return the **ans%modulo**.

**Time Complexity:**  **$O(N * \log N)$** , where **N** denotes the length of the array. In the worst case, we will be sorting the array that takes  **$O(N \log N)$**  time. For calculating the product,  **$O(K)$**  time is required. The total time complexity will be  **$O(N \log N) + O(K) = O(N * \log N)$** .

**Space Complexity:**  **$O(1)$**  as we are using constant space.

**Approach 2: Using Heaps**

1. Initialise a variable **ans** to 1 to store the product.
2. Build a **Max-Heap** for the first **K** elements of the given array.
3. For each element from index **K** to **N - 1**, compare it with the top element of the max heap.
  - a. If the element is greater than the top element, then ignore it.
  - b. Else pop the top element and insert the current element into the heap.
4. Multiply all the elements that are currently in the max heap by taking the modulo at each multiplication and update the variable **ans**.
5. Return the **ans%modulo**.

**Time Complexity:**  **$O(N * \log K)$** , where **N** denotes the length of the array. In the worst case, for each element of the array, we are heapifying the elements of the max heap of size **K**. Thus, heapifying takes  **$O(\log K)$**  time, and since we are doing it **N** times, thus time complexity would be  **$O(N * \log K)$** .

**Space Complexity:**  $O(K)$ , where  $K$  is the number of integers for which the minimum product is to be found. A max heap of size  $K$  is required justifying the space complexity.

## 5. String Transformation [<https://coding.ninja/P80>]

**Problem Statement:** Given a string (**STR**) of length **N**, you are required to create a new string by performing the following operation:

Take the smallest character from the first **K** characters of **STR**, remove it from **STR** and append it to the new string. Repeat this operation until **STR** becomes empty. Return the newly formed string.

### Note:

1. The input string (**STR**) does not contain any spaces.
2. All the characters in **STR** are lower case letters.
3. If characters less than '**K**' remain in the string, then append them in a sorted way to the new string.

### Input Format:

The first line contains an integer **t** which denotes the number of test cases or queries to be run. Then the test cases follow.

The first and the only line of each test case or query contains a string(**STR**) and an integer(**K**) separated by a single space.

### Output Format:

For each test case, print the new string formed by applying the given operation.

### Sample Input:

```
1
edcba 4
```

### Sample Output:

```
bacde
```

### Explanation:

Let the input string be “**edcba**” with **K** = 4.

- Let the new string to be formed is initially empty, **newString** = “”.
- The first set of 4 characters are ('e', 'd', 'c', 'b'). Out of these 4 characters, the smallest one is 'b' and hence we append it to the **newString** and it becomes “**b**”.
- The next set of 4 characters are ('e', 'd', 'c', 'a'). Out of these 4 characters, the smallest one is 'a' and hence we append it to the **newString** and it becomes “**ba**” .
- Now we are left with “**edc**”, and since we can't get a window of size 4, we sort them in increasing order and append them to the **newString**.

- Hence, **newString** thus formed will be "bacde".

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

1. Create a new "**answer**" string that will contain the modified string to be returned.
2. While the input string's length is greater than 0, repeat the following three steps:
  - a. Find the minimum character in the first **K** characters of the string (or the entire string if its length is less than **K**).
  - b. Append that character to the "**answer**" string.
  - c. Remove that character from the input string.
3. Return the "**answer**" string.

**Time Complexity:**  $O(N * K)$ , where **N** denotes the length of the string.

**Space Complexity:**  $O(1)$ , since we are using constant space.

### **Approach 2: Heap Approach :**

1. Take the first **K** characters and build a **min-heap** out of it.
2. Get the smallest character from this heap. Add it to the new string that we are building.
3. Next, remove the smallest character from the heap and insert the next character from the remaining pool of **N - K** characters of the given input string.

**Time Complexity:**  $O(N * \log K)$ , where **N** denotes the length of the string. We will be extracting the smallest character from the heap  $(N - K + 1)$  times. Extraction in a heap takes constant time, but at the same time, we will be inserting a new character into the heap. Inserting will take logarithmic time. So overall time complexity will be  $O(N * \log K)$ .

**Space Complexity:**  $O(K)$ , since we are maintaining a heap of size **K**.

## **6. Running Median**

[<https://coding.ninja/P81>]

**Problem Statement:** You are given a stream of **N** integers. For every  $i^{\text{th}}$  integer ( $0 \leq i < N$ ) added to the running list of integers, print the resulting median of the integers.

**Note:**

Print only the integral part of the median.

### **Input Format:**

The **first line** contains a positive integer **N**, where **N** is the size of the given input stream of numbers. The **second line** contains **N** single space-separated positive integers representing the stream of numbers.

### **Example:**

6

6 2 1 3 7 5

**Output:**

6 4 2 2 3 4

**Explanation:**

$S = \{6\}$ , median = 6

$S = \{6, 2\} \rightarrow \{2, 6\}$ , median = 4

$S = \{6, 2, 1\} \rightarrow \{1, 2, 6\}$ , median = 2

$S = \{6, 2, 1, 3\} \rightarrow \{1, 2, 3, 6\}$ , median = 2

$S = \{6, 2, 1, 3, 7\} \rightarrow \{1, 2, 3, 6, 7\}$ , median = 3

$S = \{6, 2, 1, 3, 7, 5\} \rightarrow \{1, 2, 3, 5, 6, 7\}$ , median = 4

The median of  $n$  numbers is:

- the middle-most element among the numbers when sorted in ascending order when  $n$  is odd
- the average of the two middle-most elements when  $n$  is even.

**Approach 1: Sorting Approach**

To find the median, we perform the following steps:

- For every new addition of a number to the stream, we sort all the currently present numbers.
- Now the middle-most elements would be the middle elements of this sorted array of numbers.
- Calculate the median for the current stream.

**Time Complexity:**  $O(N * N * \log(N))$ , where  $N$  denotes the number of integers in the stream.

There are  $N$  additions, and after each addition, we sort the numbers, which takes  $O(N * \log N)$  time.

**Space Complexity:**  $O(1)$ , as no extra memory is utilised.

**Approach 2: Inserting the new element at the correct position**

According to the previously discussed approach, every time there is a new addition, we have a sorted array right before the addition. Instead of sorting it again, we can simply insert this new element in the right position. The new array thus formed will still be sorted.

Now we can find the median through the middle positions of the sorted array.

**Time Complexity:**  $O(N * N)$ , where  $N$  is the number of integers. There are  $N$  additions, and each new addition takes  $O(N)$  time for insertion in the correct position.

**Space Complexity:**  $O(N)$ , where  $N$  denotes the number of integers. We maintain a vector/list to support insertions of elements at the correct positions.

### Approach 3: Min And Max Heap Approach

The idea here is to maintain two equal halves of the current stream:

- The higher half contains greater valued elements of the current stream of numbers.
- The lower half contains lower-valued elements of the current stream of numbers.

For each new addition, the transition between elements in the two halves goes this way:

- If the new element is less than the maximum element of the lower half, then this element gets added to the lower half.
- If, after the addition, the difference between the sizes of the two halves becomes greater than 1, then to maintain the balance, the maximum element is removed from the lower half and added to the higher half.
- Else, if the new element is greater than the maximum element of the lower half, then this element gets added to the higher half.
- Similarly, if after addition, the difference between the sizes of the two halves becomes greater than 1. To maintain balance, the minimum element of the higher half is removed and added to the lower half.

By maintaining the two halves in this way, we assure that the elements of the stream are equally distributed between the two halves.

Hence, after each addition, the maximum element of the lower half and the minimum element of the higher half become the middle-most elements of the numbers present in the current stream, which we can use to compute the median.

Considering the case of the odd number of elements, whichever half has more elements—the maximum of the lower half or the minimum of the higher half—will be the median.

Considering the case of the even number of elements, the average of the maximum of the lower half and the minimum of the higher half will be the median.

The above logic can easily be achieved by maintaining a **max heap** for the **lower half** of elements and a **min-heap** for the **upper half** of elements.

**Time Complexity:**  $O(N * \log N)$ , where  $N$  denotes the number of integers. There are  $N$  additions and each new addition takes  $O(\log N)$  time for transition (insertion/deletion) of elements between the two heaps.

**Space Complexity:**  $O(N)$ , where  $N$  denotes the number of integers. Maintaining the two heaps will require  $O(N)$  memory.

## 7. Minimum Character Deletion [<https://coding.ninja/P82>]

**Problem Statement:** You are given a string '**input**'; you need to find and return the minimum number of characters to be deleted from the '**input**' string so that the frequency of each character in the string becomes unique.

**For Example:** If the given string is "aabbc" then the frequency of characters present in the string is: {a:2, b:2, c:1}.

Now, as 'a' and 'b' both have the same frequency (2), we need to delete one character: either one 'a' or one 'b', to make their frequencies different. After deleting either of the characters, we will get their respective frequencies as 1, 2, and 1. Since they are all now distinct, we got our answer as 1 (the number of characters deleted).

### **Input Format:**

The **first line** contains a single integer **T**, representing the number of test cases or queries to be run.

For each test case:

Each line contains the input string.

### **Output Format:**

For each test case, print a single integer representing the minimum number of characters to be deleted.

### **Sample Input:**

1

aaabbbcc

### **Sample Output:**

2

### **Explanation:**

After removing 2 a's or 2 b's, the resulting string formed will have distinct frequencies of each character.

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

As we are only allowed to delete a character, the resulting string after deletion of some characters would be a subsequence of the string. So, we have to find such a subsequence with the unique frequency for each character present in it.

### **Steps:**

1. Initialise a global variable **ans** to **INT\_MAX**, which will store the minimum number of characters needed to be removed from the string.
2. Create all the subsequences of the string and check whether it satisfies the condition, simultaneously update the **ans** with the minimum of **ans** and (**N - length of subsequence**).
3. Finally, return the **ans**.

### **For creating the subsequence of string:**

1. Call a helper function to create all the subsequences.
2. Make two recursive calls, one by including the current character and the other by excluding the current character.
3. The base case would be when input length becomes zero, then check for the condition of unique frequency of characters, and update the **ans** accordingly.

### **For checking the unique frequency condition:**

1. Create a map to store the frequency of each character of the subsequence string.
2. Traverse the subsequence and store the frequencies in the map.
3. Create a set.
4. Traverse the keys of the map and store their frequency in the set.
  - (a) If the frequency is already present in the set, then return **false**, indicating a repeated frequency of a character.
  - (b) If the complete map is traversed, then return **true**.

**Time Complexity:**  $O(N * 2^N)$ , where **N** denotes the length of the string. In the worst case, we have two choices for every index of the given string—that is, either to include that character or to exclude that character. Also, we are traversing each subsequence to store frequencies that will take **O(N)** time. Thus, the overall complexity would be  $O(N * 2^N)$ .

**Space Complexity:**  $O(N)$ : In the worst case, we are storing **N** subsequences of the string in the stack space, and also extra space is required to store the frequencies of the characters in the map.

### **Approach 2: Using Max Heap**

1. Create a **max heap** of integers, which will store the frequency of each character in the given input string.
2. Initialise a variable **count** that will store the number of deleted characters.
3. Create a **map** and store the frequency of each character by traversing the string.
4. Store all the key's frequency in the **max heap**.
5. While **heap** is not empty:
  - a. Remove the top of the heap (the frequency of the maximum occurring character).

- b. If the current top element of the heap is equal to the frequency of the highest occurring character, then **decrement** the frequency of the highest occurring character by **one** and push it back into the heap only if the new frequency is not zero and also increment the **count** by **one**, indicating one deletion.
  - c. Else continue.
6. Finally, return the **count**.

**Time Complexity:**  $O(N * \log M)$ , where **N** denotes the length of the string and **M** is the number of unique characters in the string. In the worst case, we are traversing the complete string ( $O(N)$ ). The complexity of counting the characters to remove is  $O(\log(M))$ . Thus, the overall complexity would be  $O(N * \log M)$ .

**Space Complexity:**  $O(M)$ , where **M** denotes the number of unique characters in the string. In the worst case, we will be using extra space to store frequencies of all unique characters in a map of size **M** and a heap of size **M**.

## 8. Minimum and Maximum Cost to Buy N Candies

[<https://coding.ninja/P83>]

**Problem Statement:** Ram went to a special candy store in Ninjaland, which has **N** candies with different costs. The candy shop gives a special offer to its customers. A customer can buy a single candy from the store and get at most **K** different candies for free. Now, Ram is interested in knowing the maximum and the minimum amount he needs to spend for buying all the **N** candies available in the store.

### Note:

Ram must utilise the offer in both cases—that is, if **K** or more candies are available, he must take **K** candies for every candy purchase. If less than **K** candies are available, he must take all the candies for a candy purchase for free.

### For Example :

For **N** = 5 and **K** = 2, Let the cost of different candies in the store be: [9 8 2 6 4]

#### For the minimum cost:

Ram can buy candy with cost 2 and take candies with costs 9 and 8 for free.

Then, he can buy candy with cost 4 and take candy with cost 6 for free.

Thus, the minimum cost will be 6, that is: (2 + 4).

#### For the maximum cost:

Ram can buy candy with a cost 9 and take candies with costs of 2 and 4 for free.

Then, he can buy candy at cost 8 and take candy at cost 6 for free.

Thus, the maximum cost will be 17 (9 + 8).

Thus, minimum cost = 6 and maximum cost = 17.

### Input Format :

The **first line** contains an integer '**T**' which denotes the number of test cases or queries to be run. Then, the **T** test cases follow.

The **first line** of each test case or query contains two space-separated integers **N** and **K**, representing the number of candies available and the number of candies you get free for a single purchase, respectively.

The **second line** of each test case contains **N** single space-separated integers, representing the costs of the candies.

#### **Output Format:**

For each test case, print two space-separated integers **A** and **B**, where **A** is the minimum amount and **B** is the maximum amount in which Ram can buy all the candies.

#### **Example:**

```
1
4 2
3 2 1 4
```

#### **Output:**

```
3 7
```

#### **Explanation:**

For the minimum amount, Ram can buy a candy at cost 1 and take candies at costs 3 and 4 for free. Then, he can buy the remaining candy at the cost of 2. Thus, the minimum cost will be  $3 + 2 = 5$ .

For the maximum amount, Ram can buy a candy at cost 4 and take candies at costs 1 and 2 for free. Then, he can buy the remaining candy at the cost of 3. Thus, the minimum cost will be  $7 + 3 = 10$ .

### **Approach 1: Using Sorting**

Let us apply the **greedy** approach:

**For minimum cost:** Ram buys the candy with minimum cost and then takes top **K** expensive candies for free.

**For maximum cost:** Ram buys the candy with maximum cost and then takes top **K** cheapest candies for free.

#### **For Minimum:**

1. Sort the cost array in **increasing** order.
2. Now, until the candies are available:

- a. Start picking candies from the **left** that is the cheapest one.
  - b. Add the cost of the picked candy in a variable, say, **minCost**.
  - c. Drop **K** candies from right as you get them for free.
3. Return minCost.

For **Maximum**:

1. Sort the cost array in **decreasing** order.
2. Now, until the candies are available:
  - a. Start picking candies from the **left** that is the one with the maximum cost.
  - b. Add the cost of the picked candy in a variable, say, **maxCost**.
  - c. Drop **K** candies from right as you get them for free.
3. Return maxCost.

**Time Complexity:**  $O(N \log N)$ , where  $N$  denotes the number of candies in the store. In the worst case, we are sorting the cost array, which takes  $O(N \log N)$  time.

**Space Complexity:**  $O(1)$ , per test case. Constant extra space is used.

## Approach 2: Using Priority-Queue

The idea is to use a priority queue.

- Firstly, we need to calculate  $M$ —that is, the number of candies that are to be actually bought,  $M$  can be calculated simply as  $\text{Ceil}[ N/(K+1) ]$  as Ram gets **K + 1** candies at one purchase where **1** candy is bought by Ram and **K** candies are for free.  
For example, when Ram buys one candy for  $N = 8$  and  $K = 2$ , he gets 2 candies for free. Therefore, in one buy, he gets 3 ( $2+1$ ) candies. Thus, the total purchases required to buy all the 8 candies are  $\text{Ceil}[ 8 / 3 ] = 3$ .
- The idea is basically that for the minimum cost, take **M** cheapest candies and for the maximum cost, take **M** costliest candies.
- For the minimum cost, create a min priority queue containing the costs of all the candies and then pick top **M** elements.
- Similarly, for maximum cost, create a max priority queue containing the costs of all the candies and then pick top **M** elements.

**Time Complexity:**  $O(N + M \log(N))$  per test case where  $N$  is the number of candies in the store and  $M$  is the number of candies for which Ram has to pay. In the worst case, the time analysis is as follows:

**$O(N)$**  for building the priority queue.

**$O(M \log N)$**  for popping top  $M$  elements from the priority queue containing **N** integers.

**Space Complexity:**  $O(N)$  per test case where  $N$  denotes the number of candies in the store. We are creating a priority queue of size  $N$ .

## 9. Last Stone Weight [<https://coding.ninja/P84>]

**Problem Statement:** You are given a collection of  $N$  stones, where each stone has a positive integer weight.

On each turn, you choose the two heaviest stones present and then smash them together.

Suppose the stones have weights  $x$  and  $y$  with  $x \leq y$ . The result of this smash will be:

1. If  $x$  is equal to  $y$ , both stones are totally destroyed.
2. If  $x$  is not equal to  $y$ , the stone of weight  $x$  is totally destroyed, and the stone of weight  $y$  has a new weight equal to  $y - x$ .

In the end, there is at most 1 stone left. Return the weight of this stone or 0 if there are no stones left.

### Input Format:

The **first line** contains a positive integer  $N$ , where  $N$  is the number of stones.

The **second line** contains  $N$  single space-separated positive integers representing the weight of the stones.

**Output Format:** A single integer representing the weight of the final stone left. Print 0 if no stone is left.

### Example:

3

1 9 5

### Output:

3

### Explanation:

The two heaviest stones are 9 and 5, so after smashing, we will get a stone with a weight of  $9 - 5 = 4$ .

Now we have two stones with weights 4 and 1, and after smash, only one stone will be left with weight  $4 - 1 = 3$ .

So the answer is 3.

### Approach 1: Removing From List

We will store the stones in a vector/list and run a while loop until the size of the vector/list is greater than 1.

1. Find the index and weight of the two heaviest stones.

2. Remove those stones from the vector/list using their index.
3. Add the absolute difference of their weights to the vector /list only if the absolute difference is greater than 0.

**Time Complexity:**  $O(N^2)$ , the number of stones is  $N$ , and so the first loop will run at most  $N$  times. To find the two heaviest stones, it will take  $O(N)$ , and to remove the two heaviest stones, it will again take  $O(N)$ . Adding another stone if weight > 0 will again cost  $O(N)$  time. So the overall time complexity is  $O(N*(N+N+N))$  or  $O(N^2)$ .

**Space Complexity:**  $O(1)$  because we are not using any extra space. We are changing in the given vector itself.

### Approach 2: Priority Queues

We can use a max priority queue to solve this problem.

Store the weights of all the stones in a **max priority queue**; **it** and then while its size is greater than 1 do the following:

1. Pop two elements from the priority queue. These will be the two heaviest weighing stones from the existing elements.
2. Then, find the absolute difference between the two popped elements, and if it's greater than zero, insert the difference back into the queue.

After the above process is completed

- If there is no element left in the max priority queue, then simply return 0.
- Else return the value of the only element present in the max priority queue.

**Time Complexity:**  $O(N*LogN)$ , making the max priority queue takes  $O(N)$  time, and the popping/inserting of weights takes  $O(NlogN)$  time. So overall complexity is  $O(NlogN)$ . Here  $N$ , is the total number of stones we have in the beginning.

**Space Complexity:**  $O(N)$  because we're making a max priority queue to store the weights of  $N$  stones.

## 10. K Most Frequent Words [<https://coding.ninja/P85>]

**Problem Statement:** Given a list of  $N$  non-empty words and an integer  $K$ . Return the  $K$  most frequent words sorted by their frequency from highest to lowest.

### Note:

If two words have the same frequency, then the lexicographically smaller word comes first in your answer.

### **Input Format:**

The **first line** contains two positive integers **N** and **K**, where **N** is the number of words and **K** is the number of most frequent words to be printed.

The **second line** contains **N** single space-separated words.

### **Output Format:**

For each test case, print the **K** most frequent words.

### **Example:**

8 3

the sky is blue; the weather is hot

### **Output:**

is the blue

### **Explanation:**

"is" and "the" are the words with a frequency of 2. "sky", "blue", "weather", and "hot" are the words with a frequency of 1.

The words with a frequency of 2 are the most frequent words, and the lexicographically smallest word from the words with a frequency of 1 is "blue".

So the answer is: is the blue.

### **Approach 1: Using Sorting**

1. We use a **hashmap<key, value>** to store the frequencies of all unique words in the list, where the key is the word, and the value is the frequency of that word in the list. While iterating over the list of words, we will increment the frequency corresponding to the current word in the hashmap by 1.
2. Now, we need to find the **K** largest frequencies among them. For this, we store each word and its frequency as a pair in a list. We then sort the list based on the criteria that the word having a higher frequency should come first. If two words have the same frequency, we compare the words, and the lexicographically smaller word comes first.
3. After sorting, the first **K** words of the list would be our required answer.

**Time Complexity:** **O(NlogN)**, where **N** denotes the number of words in the file. Since hashing each word would require **O(N)** time and sorting the list would require **O(NlogN)** time.

**Space Complexity:** **O(N)**, where **N** denotes the number of words in the file. We will be storing all the **N** words in the map and in the list, and both require **O(N)** space.

### **Approach 2: Using Min Heap**

1. Similar to the first approach, we use a **hashmap<key, value>** to store the frequencies of all unique words in the list, where the key is the word, and the value is the frequency of that word in the list.

2. While iterating over the list of words, we will increment the frequency corresponding to the current word in the hashmap by 1.
3. Now, we need to find the  $K$  largest frequencies among them. For this, we will first initialise a **min-heap** of size  $K$  that will keep the least frequency word as the **top** (root) element.
4. When two words have the same frequencies, we will keep the lexicographically larger word at the top of other words in the min-heap (as the lexicographically larger word will have less priority than the other words having the same frequency).
5. Now, iterating through the Hashmap, we will add the first  $K$  words into the min-heap. We will compare the current word frequency with the minimum frequency from the heap (top element of min-heap) for the next words.
6. If the current frequency is larger than the minimum frequency from the heap, we will pop the top element from the min-heap and insert our current word with its frequency into the min-heap.
7. Otherwise, if the current frequency is equal to the minimum frequency, then we will compare both the words and only keep the lexicographically smaller word from them in the heap.
8. Now, our min heap will be storing the top  $K$  most frequent words. We will pop all words into a list. This list will now contain the least frequent word out of these  $K$  words at the 0th position and the most frequent word at the  $(K - 1)$ th position. This means we need to print the reverse of this list as our answer.

**Time Complexity:**  $O(N * \log K)$ , where  $N$  denotes the total number of words, and  $K$  is the number of words to return. We iterate on all  $N$  words of the list and add them to the min-heap of size  $K$  that requires  $O(\log K)$  time, and when the min-heap is full of its capacity, removing one word also takes  $O(\log K)$  time.

**Space Complexity:**  $O(N)$ , where  $N$  denotes the total number of words. We will be storing all the  $N$  words in a map that requires  $O(N)$  space in the worst case. Also,  $O(K)$  space will be required to maintain a heap of size  $K$ .

# 10. Graphs

---

## Introduction to Graphs

A graph **G** is defined as an ordered set  $(V, E)$ , where  $V(G)$  represents the set of **vertices**, and  $E(G)$  represents the set of **edges** that connect these vertices.

The vertices **x** and **y** of an edge  $\{x, y\}$  are called the **endpoints** of the edge. A vertex may or may not belong to any edge endpoints of the graph.

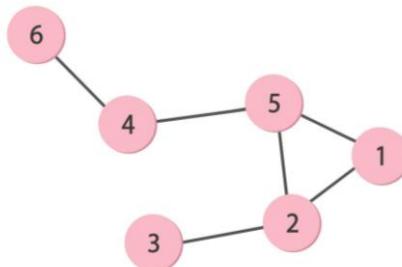
A graph network could be understood with the help of an example of a road network. Since this network is a non-uniform and non-hierarchical structure, a tree data structure cannot be used to represent it. In such cases, graphs are used.

Suppose the cities are represented in the form of numbers from (1 to 6), and the bidirectional road network connecting them is as below:

$$V(G) = \{1, 2, 3, 4, 5, 6\}$$

$$E(G) = \{(1, 2), (1, 5), (2, 5), (5, 4), (2, 3), (4, 6)\}$$

Then the graph for the given network can be represented as follows:



Basis certain properties of the graphs, they can be identified as follows:

- **Undirected Graphs:** In undirected graphs, edges **do not** have any direction associated with them. In other words, if an edge is present between nodes **A** and **B**, then the nodes can be traversed in both the directions from **A** to **B** as well as from **B** to **A**.
- **Directed Graphs:** In directed graphs, the edges form an ordered pair. For example, if there is an edge from **A** to **B**, then there is a direct path from **A** to **B** but not from **B** to **A**. The edge  $(A, B)$  is said to initiate from node **A** (**initial node**) and terminate at node **B** (**terminal node**).

Undirected Graph

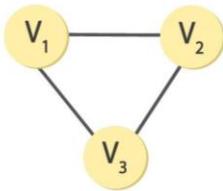


Figure 1: An Undirected Graph

Directed Graph

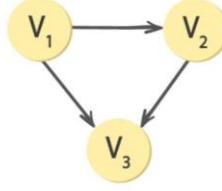
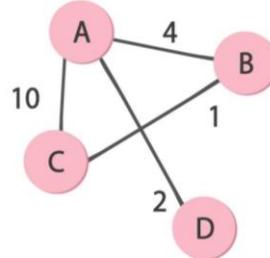


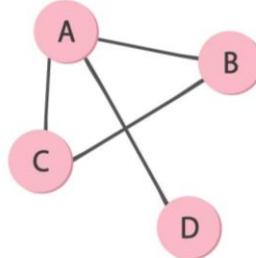
Figure 2: A Directed Graph

- **Weighted Graphs:** In weighted graphs, the edges have a designated weight associated with them.
- **Unweighted Graphs:** In unweighted graphs, the edges do not have any weight associated with them.

Weighted Graph

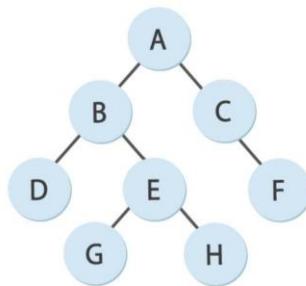


Unweighted Graph

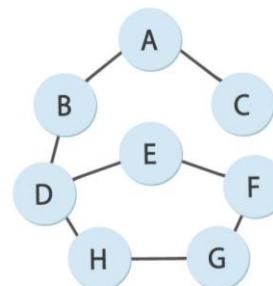


## Relationship Between Graphs and Trees

- A tree is a special type of graph in which a node can be reached from any other node using some unique path, unlike the graphs where this condition may or may not hold true.
- A **tree** is an **undirected connected graph** with **N** vertices and exactly **N - 1** edges.
- A **tree** does not have any cycles in it, while a graph may have cycles in it.



TREE



GRAPH

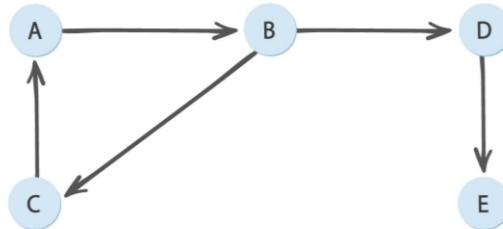
## Important Points

- A graph is said to be **connected** if there is a path between every pair of vertices.

- If the graph is not connected, then all the maximally connected subsets of the graph are called its **connected components**. Each component is connected within itself, but any two different components of the graph are never connected.
- The minimum number of edges in a connected graph will be **(N - 1)**, where **N** is the number of vertices in the graph.
- In a **complete graph** (where each vertex is connected to every other vertex by a direct edge), there are  $N C_2 = (N * (N - 1)) / 2$  number of edges, where **N** is the number of vertices. This also represents the maximum number of edges that a simple graph can have.
- Hence, if an algorithm works on the terms of edges, let's say **O(E)**, where **E** is the number of edges, then in the worst case, the algorithm will take **O(N<sup>2</sup>)** time, where **N** is the number of vertices.

## Graph Representations:

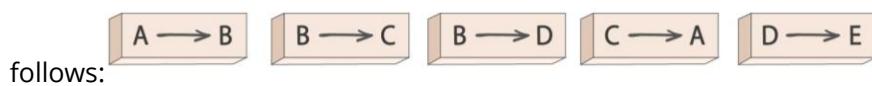
Consider a graph as follows:



There are the following ways to implement a graph:

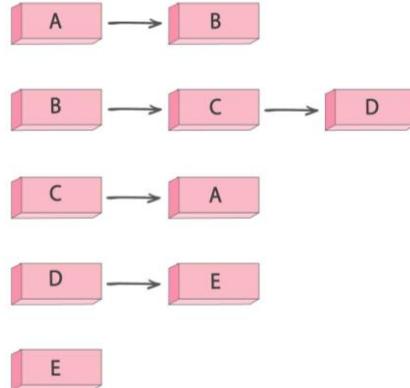
1. **Using edge list:** We can create a class that could store an array of edges. The array of edges will contain all the pairs of vertices that are connected in the graph, all put together in one place. This representation is usually not preferred for checking the presence of a distinct edge as a complete array traversal is required leading to **O(n<sup>2</sup>)** time complexity in the worst case. The space complexity of an edge list is of the order of **O(E)**, and it would be **O(V<sup>2</sup>)** in the worst case.

Pictorial representation for the above graph using the edge list representation is as



2. **Adjacency list:** An adjacency list has an array of vertices, where each vertex will have its own list of edges connecting itself to any other vertex in the graph. The primary advantages of using an adjacency list to store a graph are:
- They are easy to follow and clearly show the adjacent nodes of a particular node.

The example given in the figure can be represented as below using an adjacency list:



In the worst case, the memory requirement for storing a graph, using an adjacency list, is of the order of  $O(E)$ —that is,  $O(V^2)$ .

3. **Adjacency matrix:** An adjacency matrix is a 2 by 2 matrix. If there is an edge between the two nodes,  $i$  and  $j$ , the cell  $(i, j)$  would have a non-zero value. Likewise, in case of a no direct link among the two nodes, the cell  $(i, j)$  would contain a 0 value.

For the given graph, the adjacency matrix can be created as follows:

	A	B	C	D	E
A	0	1	0	0	0
B	0	0	1	1	0
C	1	0	0	0	0
D	0	0	0	0	1
E	0	0	0	0	0

- For a simple graph, which has no self-loops or multiple edges, the adjacency matrix has 0s on its diagonals.
- The adjacency matrix of an undirected graph is always symmetric. A symmetric matrix is the one where  $\text{mat}[i][j]=\text{mat}[j][i]$ .
- The memory usage of an adjacency matrix for  $n$  nodes is  $O(n^2)$ , where  $n$  is the number of nodes in the graph.

- The adjacency matrix for a **weighted graph** contains the weights of the edges connecting the nodes—that is,  $\text{mat}[i][j]$  = the weight of the edge present between the vertices  $i$  and  $j$ .

## Graph Traversal Algorithms

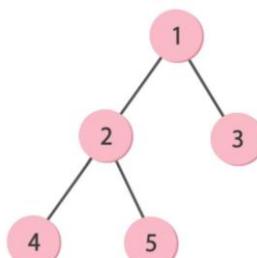
Traversing a graph means traversing all the nodes and edges of the graph. There are two standard methods of graph traversals—**Depth-first search (DFS) and Breadth-first search (BFS)**.

While breadth-first search uses a **queue** as an auxiliary data structure to store the nodes for further processing, depth-first search uses a **stack**. Both the algorithms make use of a boolean array, which is commonly called **VISITED**. During the execution of the algorithm, every node in the graph will have the value in the **VISITED** array set to either **false** or **true**, depending on whether the node has been processed/visited or not.

### Depth-first search (DFS)

The **depth-first search(DFS)** algorithm, as the name suggests, first goes into the depth in one direction exploring all the nodes and then recursively does the same in other directions. It progresses by expanding the starting node of G and then going deeper and deeper until the goal node is found for a simple graph with no self-loops or multiple edgesThe or until a node that has no children is encountered. When a dead-end is reached, the algorithm backtracks itself, returning to the most recent node that has not been completely explored.

In other words, the depth-first search begins at a starting node **A** which becomes the current node. Then, it examines each node along with a path P which begins at **A** while going deeper and deeper in a particular direction. That is, we process a neighbour of **A**, then a neighbour of the processed node, and so on until a dead end is reached. During the algorithm's execution, if we reach a path with a node that has already been processed, we backtrack to the current node. Otherwise, the unvisited (unprocessed) node, if any, becomes the current node.



**DFS Traversal:** For the given tree, starting from node 1, we can have the following DFS traversal:

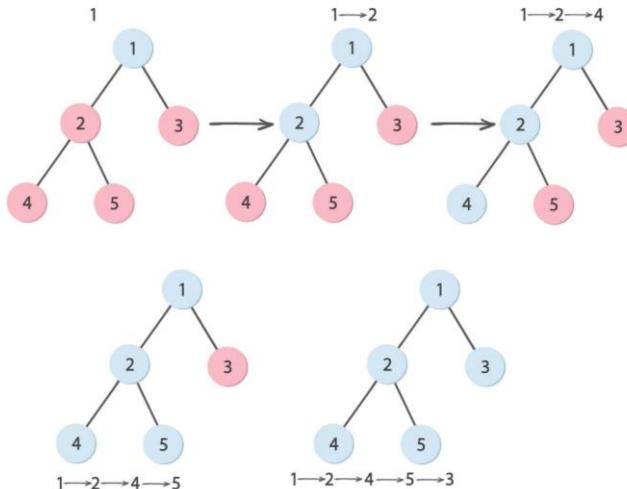
$1 \rightarrow 2 \rightarrow 4 \rightarrow 5 \rightarrow 3$ .

Other possible DFS traversals for the graph can be:

- $1 \rightarrow 3 \rightarrow 2 \rightarrow 4 \rightarrow 5$
- $1 \rightarrow 2 \rightarrow 5 \rightarrow 4 \rightarrow 3$
- $1 \rightarrow 3 \rightarrow 2 \rightarrow 5 \rightarrow 4$

We can clearly observe that there can be more than one DFS traversals for the same graph.

## DFS



### Implementation of DFS (Iterative):

```
function DFS_iterative(graph, source)

/*
    Let St be a stack representing the visited vertices.
    Push the source vertex to the stack.
*/
St.push(source)

// Mark source vertex as visited.
visited[source] = true

// Iterate through the vertices present in the stack

while St is not empty
    // Pop a vertex from the stack to visit its neighbors
    cur = St.top()
    St.pop()

    /*
        Visit the neighbors of the current vertex
        cur has neighbors 1, 2, 3, 4, 5
        If neighbor i is not visited, push it onto the stack
    */
    if (neighbor i is not visited)
        St.push(neighbor i)
        visited[neighbor i] = true
```

```

Push all the neighbours of the cur vertex that have not been
visited yet to the stack and mark them as visited.
*/
for all neighbors v of cur in graph:
    if visited[v] is false
        St.push(v)
        visited[v] = true

return

```

### Implementation of DFS (Recursive)

```

function DFS_recursive(graph,cur)

    // Mark the cur vertex as visited.
    visited[cur] = true

    /*
        Recur for all the neighbours of the cur vertex that have not been
        visited yet.
    */

    for all neighbors v of cur in graph:
        if visited[v] is false
            DFS_recursive(graph,v)
    return

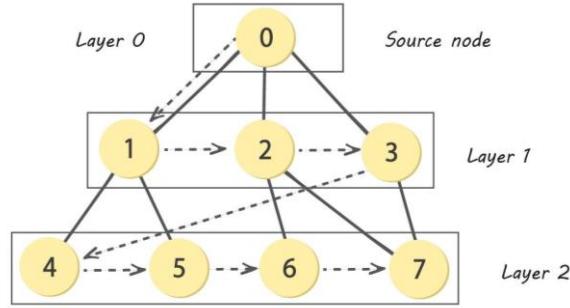
```

**Time Complexity:** The time complexity of a depth-first search is proportional to the sum of the number of vertices and the number of edges in the traversed graph. The time complexity can be represented as  $(O(|V| + |E|))$ , where  $|V|$  is the number of vertices and  $|E|$  is the number of edges in the graph, considering the graph is represented by an adjacency list.

### Breadth-first search (BFS) :

Breadth-first search (BFS) is a graph traversal algorithm in which we start from a selected node and traverse the entire graph level-wise or layer-wise, thus exploring the neighbour nodes and then moving on to the next-level neighbour nodes and so on.

As the name suggests, we first move horizontally in a level, visit all the nodes of the current layer, and then move to the next layer of the graph.



**BFS Traversal:** For the given graph, starting from node 0, we can have the following BFS traversal:

0→1→2→3→4→5→6→7

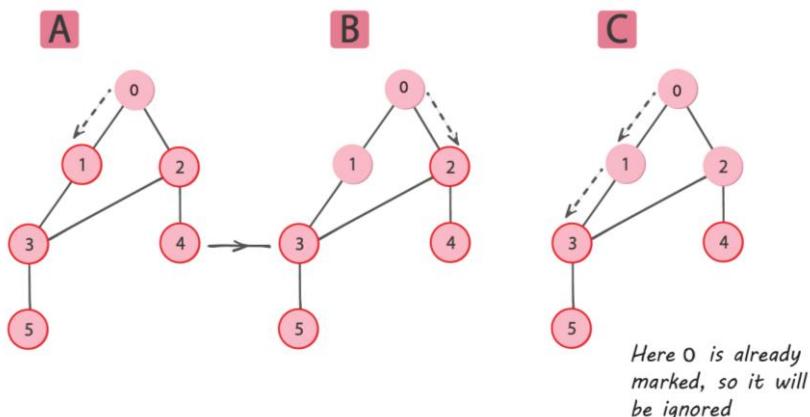
Other possible BFS traversals for the shown graph can be:

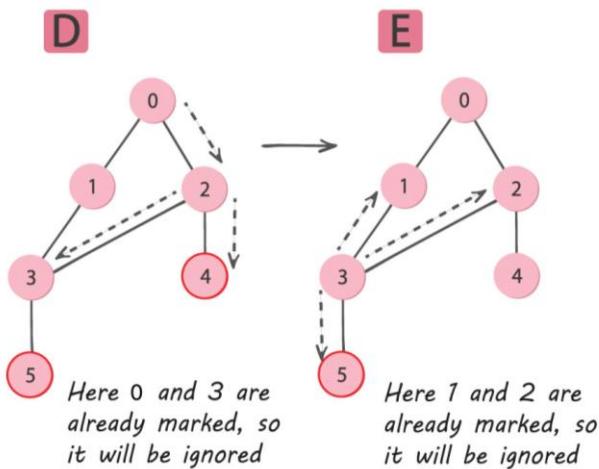
- 0→3→2→1→7→6→5→4
- 0→1→2→3→5→4→7→6

We can clearly observe that there can be more than one BFS traversals for the same graph.

If we start examining a graph from, say, node **A**, then all the neighbours of **A** are examined next in the BFS scheme of graph traversal. Thus we need to keep track of the neighbours of the current node that are not visited till now and guarantee that every node in the graph is processed and no node is processed more than once. This is accomplished by using a queue data structure that will hold the nodes that are waiting for further processing and a boolean array named **VISITED** that is used to represent the current state of every node.

For example, BFS for the following graph is:





Explanation:

- Initially, the queue contains the node 0. Therefore the node 0 is our first traversed node in BFS.
- Node 0 is popped from the queue, and its unvisited children, that is, nodes 1 and 2, are added to the queue.
- Similarly, in the following steps, nodes 1 and node 2 are removed from the queue and their unvisited children—that is, node 3 and node 4 are added to the queue, respectively.
- Finally, the unvisited child of node 3—that is, node 5 is added to the queue and is the last node to be traversed in the BFS.

Hence one possible BFS traversal for the given graph is {0, 1, 2, 3, 4, 5}.

### Implementation of BFS Iteratively :

```
function BFS(graph,source)
    /*
        Let Q be a queue that stores the set vertices that have been visited,
        but their neighbours have not been processed yet.
        Push the source vertex to the queue.
        Mark the source vertex as visited.
    */
    Q.enqueue(source)
```

```

visited[source] = true

// Iterate through the vertices in the queue.
while Q is not empty
    // Pop a vertex from the queue to visit its neighbors
    cur = Q.front()
    Q.dequeue()

    /*
        Push all the neighbours of the cur vertex that have not been
        visited yet to the queue and mark them as visited.
    */

    for all neighbors v of cur in graph:
        if visited[v] is false
            Q.enqueue(v)
            visited[v] = true

return

```

### Implementation of BFS recursively :

```

/*
    Let Q be a queue that stores the set vertices that have been visited,
    but their neighbours have not been processed yet. Initially, it has the source
    node in it.

    Let visited be a boolean array storing whether a node has been visited or
    not.

*/
function BFS(graph,visited,Q)
    // base case : if Q is empty then simply return

    if(Q.size()==0){
        Return;
    }

    Vertex v=Q.front();
    Q.pop();

```

```

    // Iterate through the neighbouring vertices of vertex v.

    For all neighbours u of v :

    /*
        Push all the neighbours of the cur vertex that have not been
        visited yet to the queue and mark them as visited.
    */

    if(!visited[u]){
        visited[u]=true;
        Q.push(u);

        // recursive call

        BFS(graph,visited,Q)
    }

    return

```

**Time Complexity:** The time complexity for breadth-first search traversal is proportional to the sum of the number of vertices and the number of edges in the graph being traversed. The time complexity can be represented as  $(O(|V| + |E|))$ , where  $|V|$  is the number of vertices in the graph and  $|E|$  is the number of edges in the graph, considering the graph is represented by an adjacency list.

#### Completeness of BFS and DFS algorithms:

**BFS** is said to be a **complete** and an **optimal** algorithm, while **DFS** does not always guarantee to be complete. Completeness of an algorithm means that if at least one solution exists, then the algorithm is guaranteed to find a solution in a finite amount of time.

#### Applications of Breadth-First Search algorithm and Depth-First Search algorithm :

- Finding a minimum spanning tree for the given graph.
- Detecting cycles in the given graph.
- Finding a path between two given vertices u and v of the graph.

- Finding the strongly connected components in a directed graph.

### Important Points

- In disconnected graphs, for traversing the whole graph, we need to call DFS/BFS for each **unvisited** vertex again until all the vertices of the graph are visited.
- For finding the number of connected components present in a graph, we need to count the number of times the DFS/BFS traversal algorithm is called on the graph on an **unvisited** vertex.

## Applications of Graphs

Graphs are very powerful data structures. They are very important in modelling data, thus many real-life problems can be reduced to some well-known graph problems. Some of the important applications of graphs are:

- Social network graphs
- Transportation networks

The other well known practical fields of applications where graphs play an integral role include:

- Neural Networks
- Epidemiology
- Scene graphs
- Design of Compilers
- Robot planning

## Practice Problems

### **1. Shortest Path using Dijkstra's algorithm:** [<https://coding.ninja/P86>]

**Problem Statement:** You are given an undirected graph of **V** vertices (labelled from  $0, 1, \dots, V - 1$ ) and **E** edges. Each edge connects two nodes (**u, v**) and has a weight associated with it, denoting the distance between node **u** and node **v**. Your task is to find the shortest path distance from the source node to all the other nodes of the graph. The source node is labelled as 0.

#### Input Format:

The **first line** of input contains an integer **T** representing the number of test cases.

The **first line** of each test case contains two integers **V** and **E**, denoting the number of vertices and the number of edges in the undirected graph, respectively.

The next **E** lines contain three space-separated integers **u**, **v**, and **distance**, denoting a node **u**, a node **v**, and the distance between nodes **(u,v)**, respectively.

#### Output Format:

For each test case, return the shortest path distance from the source node, which is the node labelled as 0, to all other vertices given in the graph in ascending order of the node number.

#### Sample Input:

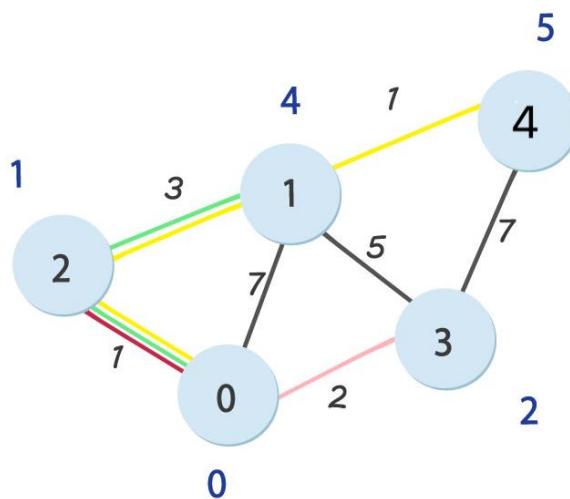
```
1
5 7
0 1 7
0 2 1
0 3 2
1 2 3
1 3 5
1 4 1
3 4 7
```

#### Sample Output:

```
0 4 1 2 5
```

#### Explanation:

The input graph with the associated weights for the edges can be represented as shown:



The source node is the node labelled as 0.

1. The shortest distance from node 0 to node 0 is **0**.
2. The shortest distance from node 0 to node 1 is **4**, represented in **green**. The path goes from node 0 → 2 → 1, giving the shortest distance as  **$1 + 3 = 4$** .
3. The shortest distance from node 0 to node 2 is **1**, represented in **red**. The path goes from node 0 → 2, giving the shortest distance as **1**.

4. The shortest distance from node 0 to node 3 is **2**, represented in **pink**. The path goes from node 0 → 3, giving the shortest distance as **2**.
5. The shortest distance from node 0 to node 4 is **5**, represented in **yellow**. The path goes from node 0 → 2 → 1 → 4, giving the shortest distance as  **$1 + 3 + 1 = 5$** .

### Approach 1: Using Adjacency Matrix

The idea here is to maintain two arrays. The first would store the status of all the nodes (**visited**)—that is, whether they are visited or not. The second array will store the **distance**—the element at index  $i$  of this array will store the distance from node 0 to node  $i$ .

We pick a minimum distance vertex (the one having the least value in the **distance** array) among all the unvisited vertices till now, then update the distance of all its adjacent vertices if required by considering a path from the source node to the adjacent vertex to pass through the picked minimum distance vertex. Repeat this process until all the nodes are visited.

#### Steps:

1. Consider a matrix of size **V \* V**, initialise all its positions to 0. Now, update the distance between two vertices in the matrix (denoted by the row number and column number) to the given distance in the input.
2. Initialise a **distance** array to **INT\_MAX** and a boolean **visited** array to false.
3. Since the distance of the source node—that is, node 0 to itself is 0, update **distance[0]** to 0.
4. Find the shortest path for all the vertices using the steps given below:
  - a. Pick the minimum distance vertex, which is not yet visited; let's call it **u**.
  - b. Mark it as true in the visited array—that is, **visited[u] = true**.
  - c. Update the distances of its adjacent vertices. Update **distance[v]** only if vertex **v** is not already visited and the current distance of the vertex **v** (the value stored at a distance [v]) is greater than the sum of the distances from source vertex 0 to vertex **u** and the distance from **u** to **v**. This indicates that the shortest path from the source vertex to vertex **v** passes through the vertex **u**.
  - d. Repeat the above steps till all the vertices are visited.
5. Return the distance array, which is our required answer.

**Time Complexity:** **O(V<sup>2</sup>)**, where **V** is the number of vertices in the graph. Finding the minimum distance vertex, which is unvisited, takes to traverse the entire distance array every time, costing **O(V)** order time; thus overall complexity is **O(V<sup>2</sup>)**.

**Space Complexity:** **O(V<sup>2</sup>)**, where **V** is the number of vertices in the graph. The adjacency matrix takes the size of the order **V<sup>2</sup>**.

### Approach 2: Using the Adjacency List

The idea here is to use a **priority queue** which creates a **min-heap** for us. In the priority queue, we will be storing the node **v** along with its corresponding distance from the source node. In our case, the priority will be set based on the attribute **distance**. We will pick an unvisited minimum distance vertex which will be the first element of the priority queue, then update the **distance** of all its adjacent vertices (using its adjacency list) by considering a path from the source node to the adjacent vertex to pass through the picked minimum distance vertex. Repeat the process till the priority queue is empty.

### **Steps:**

1. Form an adjacency list for the graph from the given input information.
2. Initialise the distance array to **INT\_MAX** and the boolean visited array to **false**.
3. Since the distance of the source node to itself is 0, update **distance[0]** to 0.
4. Find the shortest path for all vertices using the steps below:
  - a. Pick the first element of the priority queue, which will give the minimum distance vertex which is not yet visited, let's say **u**. Pop it from the priority queue.
  - b. Mark it as true in the visited array—that is, **visited[u]=true**.
  - c. Iterate through the adjacency list of this minimum distance vertex to find all the adjacent vertices to the vertex **u**.
  - d. Update the distances of the adjacent vertices. Update **distance[v]** only if vertex **v** is not already visited and the current distance of the vertex **v** is greater than the sum of the distances from source vertex 0 to vertex **u** and the distance from **u** to **v**. This means that the shortest path from the source vertex to vertex **v** passes through vertex **u**. Push the adjacent vertices along with their distances to the priority queue.
  - e. Repeat the above steps till the priority queue is not empty.
5. Return the distance array, which is our required solution.

**Time Complexity:**  $O(E * \log(V))$ , where **E** is the number of edges and **V** is the number of vertices in the graph. The time taken to add each vertex to the priority queue is **log(V)**. Hence, for **V** vertices, the time taken will be  $(V * \log(V))$ . To traverse all the **E** edges and update the minimum distance for the vertices in the worst case would cost  $(E * \log(V))$  order time.

So overall complexity will be  $O(V\log(V)) + O(E\log(V)) = O((E+V)\log(V)) = O(E\log(V))$

**Space Complexity:**  $O(V^2)$ , where **V** is the number of vertices in a graph. The adjacency list, in the worst case, could have a size of the order of  $O(V^2)$  for a complete graph.

## **2. Minimum Spanning Tree** [<https://coding.ninja/P87>]

**Problem Statement:** You are given an undirected, connected, and weighted graph **G(V, E)**, consisting of **V** number of vertices (numbered from 0 to **V - 1**) and **E** number of edges. Find and print the total weight of the Minimum Spanning Tree (MST) for the given graph using **Kruskal's algorithm**.

By definition, a minimum weight spanning tree is a subset of the edges of a connected, edge-weighted, undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight.

#### **Input Format:**

The **first line** of input consists of two integers, **N** and **M**, the number of vertices and the edges in the graph, respectively.

The next **M** input lines contain three integers **X**, **Y**, and **W**, representing an edge of the graph. Here the edge **X**, **Y**, **W** represent an edge between the vertices **X** and **Y** of weight **W**.

#### **Output Format:**

Print the total weight of the minimum spanning tree.

#### **Sample Input:**

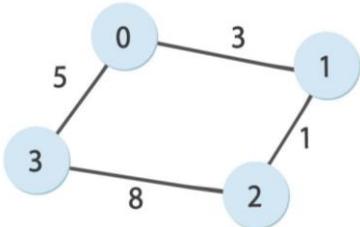
```
4 4
0 1 3
0 3 5
1 2 1
2 3 8
```

#### **Sample Output:**

```
9
```

#### **Explanation:**

The input graph can be represented as :



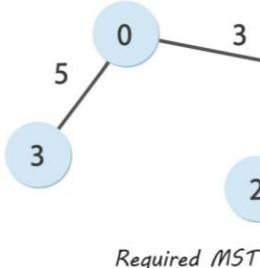
The edge connecting the vertices 2 and 3, having weight 8, will be excluded from the graph to form the required MST. Thus the total weight of the MST will be  $1 + 3 + 5 = 9$ .

#### **Kruskal's Algorithm:**

The algorithm first puts all the graph's vertices as a collection of single-vertex trees, separated from each other. They are then gradually combined together to form a single tree.

Before the algorithm is executed, all the graph edges are sorted in order of non-decreasing weights. Then we start selecting edges in the sorted order—that is, minimum first. If the endpoints of the currently selected edge belong to different subtrees, then we connect those subtrees, and that edge is added to the pool of the selected edges for the MST.

After we've iterated through all the edges of the graph or the pool of the selected edges for the MST contains exactly  $n - 1$  edges, then all the vertices will belong to the same subtree, which will give us the desired MST.



### Steps:

1. Sort all the edges of the graph in order of non-decreasing weights.
2. Put each vertex of the graph in their own tree—that is, set the parent of each vertex to itself.
3. Iterate through all the edges in the sorted order.
4. If the two vertices of an edge belong to two different subtrees—that is, if they have different parents, add that edge to the MST and make the parents of each vertex in the two connected subtrees equal.
5. After iterating through all the edges, you will end up having a collection of exactly  $n - 1$  edges that will make up the MST.

### DSU (Disjoint Union Set):

Disjoint Set Union (DSU) is a graph algorithm that helps us to efficiently find whether two vertices of a graph belong to the same connected component or not. Initially, each vertex of the graph is assigned to a new set, represented by a set number, equal to the value of the vertex itself.

#### Using DSU to figure out whether two vertices belong to the same subtree or not:

To find in which subtree a vertex lies in, we simply check for the set number corresponding to that element in the DSU. If the set number does not match the vertex value of the element, we then recursively check for the 'set number' itself, and we do this until  $(\text{current value}) \neq (\text{set number})$ . When the two values become equal, we actually reach the 'head' of the set—that is, the topmost parent of the current vertex subtree.

To merge the two sets, we simply point the head of the smaller set to the head of the larger set (Union By Rank). Hence, this is an  $O(1)$  operation.

In the worst case, the 'find' operation will be of order  $O(N)$ . To avoid this scenario, we use an algorithm called **Path Compression**. Putting it briefly, suppose the heads of the sets point in the fashion shown:

1→2→3→4

If we want to look for the set number of 1, we have to reach 4. After we know that the set head is 4, we simply point all values encountered so far, like 2 and 3 to 4, so it would look something like:

```
1→4←3
↑
2
```

This is Path Compression in essence. It has an amortised (per operation analysis) constant time complexity.

#### Time Complexity:

$O(E * \log(V))$ , where  $E$  and  $V$  are the numbers of edges and vertices in the graph, respectively.

$O(E * \log(E))$  or  $O(E * \log(V))$  time is taken up to sort the edges.

We can replace  $E * \log(E)$  with  $E * \log(V)$  above because  $E \leq V * V$ .

$$\Rightarrow \log(E) \leq \log(V^2)$$

$$\Rightarrow \log(E) \leq 2 * \log(V)$$

Therefore the overall time complexity is  $O(E * \log(V))$ .

**Space Complexity:**  $O(E + V)$ .  $O(E)$  space is needed to store the edges and  $O(V)$  to store the parents of each vertex. Therefore the overall space complexity is  $O(E + V)$ .

### 3. Alien Dictionary [<https://coding.ninja/P88>]

**Problem Statement:** You are given a sorted (lexical order) dictionary of an alien language. Write a function that finds the order of the characters in the alien language. The dictionary given to you is in the form of an array of strings called 'dictionary' of size  $N$ .

#### Example:

If the dictionary consists of the following words "caa", "aaa", "aab" then, the order of alphabets in the alien language are 'c', 'a', and 'b'.

**Remember:** If the alien language consists of only four unique letters, then the four letters should be the starting four letters of the English alphabet (a, b, c, and d). However, their order might differ in the alien language.

#### Input Format:

The **first line** contains an integer  $T$  which denotes the number of test cases or queries to be run. Then the test cases follow:

The **first line** of each test case or query contains an integer  $N$  representing the size of the alien dictionary.

The second line contains  $N$  single space-separated strings representing the words in the alien dictionary.

#### Output Format:

For each test case, return an array of characters representing the order in which they will appear in the alien language. You need not print anything. True or False in the output shows whether the returned array of characters is in the correct order or not.

#### Sample Input:

```
2
3
a aa aaa
3
a b c
```

#### Sample Output:

```
true
true
```

#### Explanation:

The words given in the dictionary are 'a', 'aa', and 'aaa'. Since the only unique character here is 'a', so the array to be returned will just be ['a']. The 'true' that is being printed in the output, just signifies that the output returned by the function is valid. For the second test case, the words are 'a', 'b', and 'c'. The unique characters here are 'a', 'b', and 'c' (in that order), so the array to be returned will be ['a', 'b', 'c']

### Approach 1: Permutations Approach (Brute Force):

In this approach, we find all the possible permutations of distinct characters to check if the words are sorted according to this sequence.

#### Steps:

1. Find all the distinct characters present in all the words given in the dictionary.
2. Generate all possible permutations of these distinct characters.
3. Treat each of the permutations as a correct sequence of characters.
4. Check if the given words in the dictionary are sorted according to this sequence by following the steps as listed :
  - a. For all words from 1 to  $n - 1$ , let the current word be **currWord** and the next word be **nextWord**.
  - b. One by one, compare characters of both the words and find the first mismatching character. In case there was no mismatching character, continue to the next word.
  - c. Let us say that the mismatching characters were **ch1** and **ch2** in **currWord** and **nextWord**, respectively.
  - d. If these words (**currWord** and **nextWord**) follow the dictionary, **ch1** will occur before **ch2** in the sequence of permutation chosen to be correct.
5. In case the words are sorted according to the current sequence, return this sequence.

**Time Complexity:**  $O(K! * N * L)$ , where **K** is the number of distinct characters, **N** represents the number of words present in the dictionary, and **L** is the maximum length of a word in the dictionary. Generating all the possible permutations of the distinct characters in the dictionary costs  $K!$  order time, and then using the formed sequence to check if it is a valid sequence or not will cost  $N * L$  order time. Therefore the overall time complexity is  $O(K! * N * L)$ .

**Space Complexity:**  $O(K)$ , since we are storing **K** distinct characters.

### Approach 2: Using Topological Sorting

Consider ["wrt", "wrf", ....] as the first two of the words of the given dictionary. Looking at the first mismatch in the characters, we can say '**t**' comes before '**f**'.

We could denote this relation by, '**t** → **f**', this implies an edge directed from '**t**' to '**f**'. Thus we can represent this relation using a directed graph.

#### Steps:

1. Iterate over all the words present in the dictionary and make a directed graph representing the above relations. All the distinct characters will be the vertices of the graph whereas, the relation mapping which character comes before another character will be the directed edge.
2. Do a topological sort over the built graph and return one of the possible orders for the same.
  - a. Maintain an indegree array, which stores the indegree of every vertex present in the array.
  - b. Maintain a queue, which has only the vertices with zero indegree. Initially, insert all the vertices for which the indegree is currently zero, and add the current vertex to the answer array.
  - c. Remove each element one by one from the queue and start traversing its neighbours. Now break the edge between the removed vertex and its current neighbour vertex—that is, decrease the indegree of the neighbour by one; if it becomes zero, add it to the queue.
  - d. Repeat the above-mentioned steps until the queue becomes empty.
  - e. If all the elements are visited in the end, return the answer array; else it is not possible to find the order of characters in the alien language.

**Time Complexity:**  $O(N + K)$ , where **N** is the total number of words and **K** is the total number of distinct characters present in the list of **N** words.

The time complexity of Topological Sort is  $O(V + E)$  which is  $O(K + N)$

**Space Complexity:**  $O(K)$ , where **K** represents the total number of distinct characters.

## **4. Snake and Ladder** [<https://coding.ninja/P89>]

**Problem Statement:** You are given a snake and ladder board with **N** rows and **N** columns with the numbers written from 1 to (**N \* N**) starting from the bottom-left corner of the board and in an alternating direction in each row. For example, for a 6 x 6 board, the numbers appearing on the board would be as follows.

36	35	34	33	32	31
25		27	28	29	30
24		22	21	20	19
13	14	15	16	17	18
12	11	10	9	8	7
1	2	3	4	5	6

You start from **square 1** of the board (which is always the last row and first column cell of the board). On each square, say **X**, you can throw a dice which can have six possible outcomes, and you have total control over the outcome of the dice throw, and you want to find out the **minimum** number of throws required to reach the last cell of the board.

Some of the squares contain Snakes and Ladders, and these are the possibilities of a throw at square **X**.

You choose a destination square '**S**' with a number **X + 1, X + 2, X + 3, X + 4, X + 5, or X + 6**, provided this number is  $\leq N * N$ . If **S** has a snake or ladder, you move to the destination of that snake or ladder. Otherwise, you move to destination square **S**.

**A board square on row 'i' and column 'j' has a "Snake or Ladder" if  $board[i][j] \neq -1$ . The destination of that snake or ladder is represented by the value  $board[i][j]$ .**

### **Note:**

You can only take a snake or ladder at most once per move—that is, if the destination square to a snake or ladder is the start of another snake or ladder, you do not continue moving. In this case, you have to ignore the snake or ladder present on that square.

For example, if the board is: { -1 1 -1 , -1 -1 9 , -1 4 -1 }. Let's say on the first move your destination square is 2 [at row 2, column 1], then you finish your first move at square 4 [at row 1, column 2] because you do not continue moving to 9 [at row 0, column 2] by taking the ladder from square 4.

A square can also have a Snake or Ladder, which will end at the same cell. For example, if the board is: { -1 3 -1, -1 5 -1, -1 -1 9 }. Here we can see Snake/Ladder on square 5 [at row 1, column 1] will end on the same square 5.

### **Input Format:**

The **first line** contains a single integer value, representing the number of rows and columns for the two-dimensional square matrix.

The **next N** lines contain **N** space-separated integers, representing the  $i^{\text{th}}$  row values.

#### Output Format:

The only line of output prints the minimum number of throws required to reach the last cell of the board. If it is impossible to reach the last cell of the board, then print -1.

#### Sample Input:

```
3
-1 1 -1
-1 -1 9
-1 4 -1
```

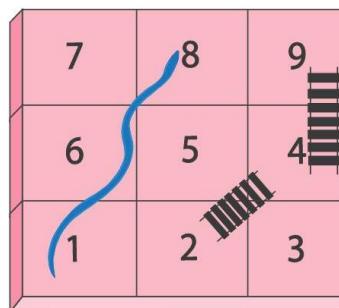
#### Sample Output:

```
1
```

#### Explanation:

In the beginning, you start at square 1 [at row 2, column 0]. The optimal path will be:

You decided to move to square 4 [at row 1, column 2] and then take the ladder to square 9 [at row 0, column 2], which is the last square. Thus you finished at the last cell of the board in just 1 dice throw, and hence the answer is 1.



#### BFS Approach:

We will use Breadth-First Search to find the shortest path from cell number 1 to cell number  $\mathbf{N} * \mathbf{N}$ .

#### Steps:

1. Maintain a queue that will store the cell numbers, where the front of the queue always contains a cell number with the minimum number of steps to reach it from the starting cell (cell numbered 1)
2. Create an array, say **minDiceThrow**, of size  $\mathbf{N} * \mathbf{N}$ , Initialize it with the maximum value (**INT\_MAX**).

3. Start with pushing cell number 1 to the queue and updating **minDiceThrow[1] = 0** because the number of throws required to reach cell number 1 is 0.
4. Iterate till we reach the destination cell or our queue becomes empty.
5. In each iteration, pop the front cell number and store it in a variable, say **cellNumber**. This will have the minimum dice throw value till now from the starting cell. Update cells that are reachable from the popped cell at the cost of one dice throw—that is, increment the dice throw value for all the other reachable cells from that cell by 1.
6. The cells which can be reached at one dice throw are:
  - a. **cellNumber + 1**
  - b. **cellNumber + 2**
  - c. **cellNumber + 3**
  - d. **cellNumber + 4**
  - e. **cellNumber + 5**
  - f. **cellNumber + 6**
7. Store them in a variable, say **nextCell**. If **nextCell** is greater than the value **N\*N**, then ignore this unreachable cell.
8. If these cells contain any snake or ladder, then change the **nextCell** to the end square of the snake or ladder.
9. If the updated **minDiceThrow[]** value of **nextCell** is more than the prior **minDiceThrow[]** value, update the **minDiceThrow** Array and push the **nextCell** in the queue.
10. Finally, if **minDiceThrow[N \* N] = maximum value (INT\_MAX)**, then we can't reach the last cell of the board. Hence, we return -1.
11. Else if the **minDiceThrow[N \* N]** is not equal to maximum value, return the value at **minDiceThrow[N \* N]**.

**Time Complexity:**  $O(N^2)$ , where **N** is the number of rows and number of columns on the board. At max, we can push and pop each cell in our queue. There are total **N \* N** cells. Therefore the overall complexity will be  $O(N^2)$ .

**Space Complexity:**  $O(N^2)$ , where **N** is the number of rows and number of columns on the board. We are using a **minDiceThrow** array of size **N \* N** to store the minimum number of dice throws required to reach each cell. Therefore the overall space complexity will be  $O(N^2)$ .

## 5. Is it a Tree? [\[https://coding.ninja/P90\]](https://coding.ninja/P90)

**Problem Statement:** Given a graph with **V** vertices numbered from **0** to **(V - 1)** and **E** edges. Determine if it is a tree or not.

### Input Format :

The **first line** of the input contains two integers **V** and **E**, separated by a single space. They denote the total number of vertices and edges of the graph, respectively.

The next **E** lines contain two space-separated integers, **a** and **b**, representing an edge between the vertex **a** and vertex **b**.

#### Output Format :

The only line of output prints 'True' if the given graph is a tree; otherwise prints 'False'.

#### Sample Input:

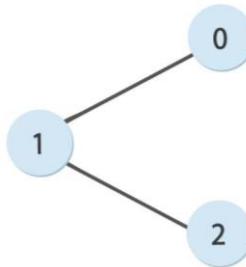
```
3 2
0 1
1 2
```

#### Sample Output:

```
True
```

#### Explanation:

We clearly can see that the given graph is a tree since it satisfies the properties of a tree.



A graph is a tree if the following two conditions are satisfied:

1. There are no cycles present in the graph.
2. The graph is connected.

Hence, we will check if the graph has any cycle present in it or not. Once this is done, we will also check if the graph is connected or not. If both the conditions are satisfied, we can conclude that the graph is a tree.

### Approach 1: Using DFS

#### Steps:

1. Create the graph using the given set of input edges and vertices.
2. Run a DFS starting from any vertex while keeping track of the parent of the visited nodes (initially, the parent of every vertex is -1).
3. Recur for all the vertices adjacent to the current vertex.
4. If an adjacent vertex is not visited, then recur for that adjacent vertex—that is, call the DFS function again for that adjacent vertex.

5. If an adjacent vertex is visited and it is not the parent of the current vertex, this shows there is a back edge, thereby indicating the presence of a cycle and thereby concluding that the given graph is not a tree.
6. If the adjacent vertex is the parent of the current vertex, then continue.
7. To check whether the graph is connected or not, use DFS again.
  - a. After DFS, check if all the vertices are visited or not.
  - b. If all the vertices are visited, then the graph is connected; otherwise it is not connected.

**Time Complexity:**  $O(V + E)$ , where **V** is the number of vertices and **E** is the number of edges.

**Space Complexity:**  $O(V + E)$ , as we are using an adjacency list.

### Approach 2: Using BFS

The only difference in this approach is that, instead of using the DFS algorithm, we will use the BFS algorithm to check for the two conditions of the graph to be a tree.

#### Steps:

1. Create the graph using the given set of input edges and vertices.
2. Run a BFS starting from any vertex while keeping track of the parent of visited nodes.
3. Push the vertex into a queue and mark its parent as -1 since it is the starting vertex for the BFS algorithm.
4. Run a loop while the queue becomes empty:
  - a. Dequeue vertex from the queue, say **u**, and mark it as visited.
  - b. For vertex **u**, if there is an adjacent vertex **v** such that **v** is already visited and is also not its parent, then there is a cycle in the graph.
  - c. For vertex **u**, if there is an adjacent vertex **v** such that **v** is not already visited, add the vertex **v** to the queue and set its parent to **u**.

To check if the graph is connected or not, use BFS again.

- a. After BFS, check if all the vertices are visited or not.
- b. If all the vertices are visited, then the graph is connected; otherwise it is not connected.

**Time Complexity:**  $O(V + E)$ , where **V** is the number of vertices and **E** is the number of edges in the graph.

**Space Complexity:**  $O(V + E)$ , as we are using an adjacency list.

## 6. M-Coloring Problem [\[https://coding.ninja/P91\]](https://coding.ninja/P91)

**Problem Statement:** You are given an undirected graph in the form of an adjacency matrix along with an integer **M**. You need to tell if you can colour the vertices of the graph using at most **M** colours such that no two adjacent vertices in the graph are of the same colour.

#### Input Format:

The first line of input contains a single integer **T**, representing the number of test cases or queries to be run.

Then the **T** test cases follow.

The **first line** of each test case contains two space-separated integers **V** and **M**, denoting the number of vertices in the undirected graph and the maximum number of colours available respectively.

Each of the next **V** lines contains **V** space-separated integers denoting the adjacency matrix of the given undirected graph.

#### **Output Format:**

For each test case, you need to print **YES** if we can colour the given graph with at most **M** colours. Otherwise, print **NO**.

#### **Sample Input:**

```
1
3 3
0 1 0
1 0 1
0 1 0
```

#### **Sample Output:**

YES

#### **Explanation:**

The given adjacency matrix is: [ [0 1 0] [1 0 1] [0 1 0] ] and **M** = 3.

The given adjacency matrix tells us that 1 is connected to 2 and 2 is connected to 3. So if we colour vertex 1 with 2, vertex 2 with 1, and vertex 3 with 2, it is possible to colour the given graph with 2 colours which are  $\leq M$  where **M** is given as 3.

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

In this approach, we generate all the possible combinations of colours that could be assigned for colouring the given graph and then check if the chosen combination satisfies the given condition or not.

##### **Steps:**

1. Generate all the possible combinations of colours that could be assigned for colouring the given graph.
2. To do that, recursively assign each node all the colours from 1 to **M**. This procedure will be repeated for every graph node.
3. For each such combination, check if the adjacent vertices don't have the same colour.
4. If such a combination of vertices is found, print YES.
5. Else, print NO.

**Time Complexity:**  $O(M^V)$ , where **V** is the number of vertices in the graph and **M** is the maximum number of colours allowed. This is because for each node, there are a total of **M** possibilities to colour it and there are **V** nodes, which gives us  $M * ... * M$  (**V times**) =  $M^V$  order time complexity.

**Space Complexity:**  $O(V)$ , because we make an array of size **V** for storing the colours of each vertex.

#### **Approach 2: Backtracking**

In the brute force method, we generated all the possible ways to assign **M** colours to the nodes of the graph and then checked if it was possible to colour the vertices of the graph using at most **M** colours such that no two adjacent vertices in the graph are of the same colour.

In this approach, the optimisation is that we will assign the colours to nodes only after checking the above condition.

**Steps:**

1. Generate all the possible combinations of colours that could be assigned for colouring the given graph.
2. Optimisation in this method is that we would assign the colours after checking if it is possible to make the vertex of that colour or not.
3. Assign each vertex a colour from 1 to  $M$ , check if its adjacent vertex has a different colour or not.
4. If we get a configuration such that each node is coloured from 1 to  $M$  and all its adjacent vertices are of different colours, print YES.
5. Else, print NO.

**Time Complexity:**  $O(M^V)$ , where  $V$  is the number of vertices in the graph and  $M$  is the maximum number of colours allowed. This is because for each node, there are  $M$  possibilities to colour it and there are  $V$  nodes, which gives us  $M * \dots * M$  ( $V$  times) =  $M^V$  order time complexity.

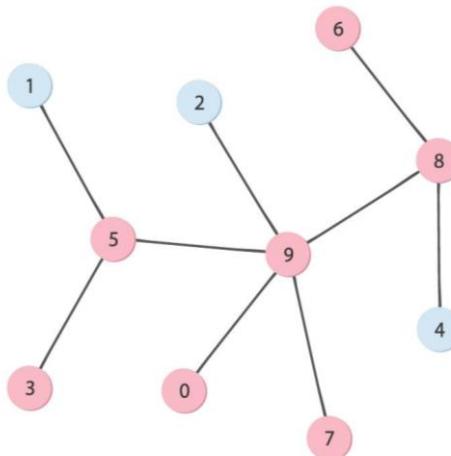
Although the worst-case time complexity would be the same as brute force, in this approach, the average time complexity is better than the brute force method.

**Space Complexity:**  $O(V)$ , because we make an array of size  $V$  for storing the colours of each vertex.

## 7. Count Nodes Within K-Distance [\[https://coding.ninja/P92\]](https://coding.ninja/P92)

**Problem Statement:** You are given a connected, undirected, and acyclic graph with  $M$  number of nodes marked in it and a positive number  $K$ . You need to return the count of all such nodes in the graph which have a distance less than or equal to  $K$  from all the marked nodes, which means every node whose distance from all marked nodes is less than or equal to  $K$ , should be counted in the result.

**Example:**



Marked nodes are circled with red colour.

Now consider the example of the graph shown in the figure. Here, nodes 1, 2 and 4 are marked. Let us take the value of  $K$  as 3. Thus, we need to find all the nodes that are at a distance less than or equal to 3 from all the marked nodes. We can see that nodes with indices: 5, 9, 8, 2, 0, 7 have distances less than or equal to 3 from all marked nodes. Therefore, the total count of nodes is 6.

**Input Format:**

The **first line** of input contains two space-separated integers **V** and **E**, denoting the number of vertices and edges in the graph.

The **next E lines** contain two space-separated integers denoting the vertices between whom edges exist.

The **next line** contains a single integer **K** denoting the distance.

The **next line** contains a single integer **M** representing the number of marked nodes in the graph.

The next line contains **M** space-separated integers representing marked nodes in the graph.

**Output Format:**

For each test case, return an integer denoting the count of the nodes which are at a distance less than or equal to **K** from all the marked nodes.

**Sample Input:**

10 9

2 1

1 4

1 9

3 4

4 6

4 7

4 8

5 6

6 10

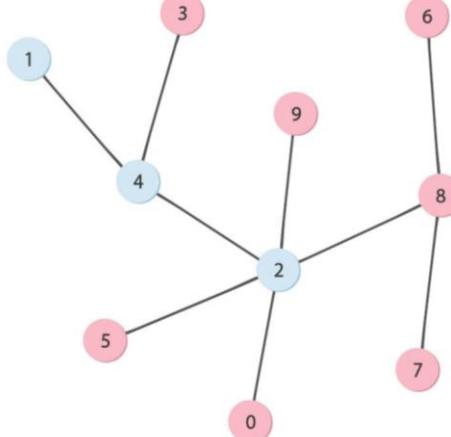
3

3

1 2 4

**Sample Output:**

8

**Explanation:**

From the test case, we can draw the following graph as shown in the figure, and it is given that nodes with values 1, 2, and 4 are marked, and the value for **K** is 3. In the above graph, we can see that nodes with values 1, 2, 3, 4, 9, 6, 7, and 8 have distances less than or equal to 3 from all the marked nodes shown in the figure. Thus the answer will be 8.

## Approach: Using BFS Traversal

The problem can be solved using the BFS traversal on the graph.

The main idea here is to find the two extreme marked nodes with the maximum distance between them.

After finding the two extreme marked nodes, we will check each node's distance from both of these nodes (which are farthest from each other). If the node's distance is less than **K** from both nodes, that means it will be at a distance less than **K** from all the other marked nodes as well, because these two nodes represent the extreme limit of all the given marked nodes in the graph. Now by doing a BFS traversal on the graph starting from any node, we can find the first extreme of both the nodes. And by again doing BFS from this found node, we can find the second extreme node. Then we can find the distances of all the nodes from both these extreme nodes using BFS traversals and store these distances in two different arrays.

After getting all the distances, we can check whether the distances are less than **K** or not and update the count, which would be our answer.

### Steps:

- Create a function called **BFSUTIL**, which we will use to traverse the graph in a BFS fashion and return the most extreme marked node from node **U** by calculating the distance of all the nodes encountered during BFS traversal.  
The function signature would look like **BFSUTIL (GRAPH, MARK, U, DISTANCES)**, where **GRAPH** stores the graph in an adjacency list format, **MARK** is a boolean array for marked nodes, **U** is a node to start BFS from and **DISTANCES** for storing distances of the nodes from **U**.
- Now inside the **BFSUTIL** function do the following:
  - 1) Declare a variable for storing the previously marked node, say, **PREMARKED**.
  - 2) Initialise a queue for BFS traversal **Q** and push node **U** in the queue and initialise its distance as 0.
  - 3) Now run a loop until all the nodes are processed, or the queue is empty.
    1. Dequeue from the queue and check if the node is already marked in the boolean array **MARK** or not; if it is already marked, change **PREMARKED** as the current node.
    2. Now loop over all neighbours of **U** and update their distance before pushing in the queue only if its value is the initial value of -1.
    3. Finally, return the **PREMARKED** node.
- Now for the main function, which returns the required count of nodes, prepare the graph.
  - 1) Declare three arrays **TEMP**, **DISTANCEFROMFIRSTEXTREME**, **DISTFROMSECONDEXTREME**, to find the first extreme node and to store distances from both extreme nodes, respectively.
  - 2) Run three BFS traversals as discussed in the approach.
  - 3) Finally, check for the distances which are less than **K** and add to the final **COUNT**, which was initialised as ZERO initially.
  - 4) Return **COUNT**, which is our answer.

**Time Complexity:**  $O(V + E)$ , as we are doing a BFS traversal, which takes  $O(V + E)$  time complexity, where **V** denotes the number of vertices in the graph, and **E** denotes the number of edges in the graph.

**Space Complexity:**  $O(V)$ , where **V** denotes the number of vertices in the graph as we are using an array to store distances of the vertices.

## **8. Shortest Path in an Unweighted Graph** [<https://coding.ninja/P93>]

**Problem Statement:** The city of Ninjaland is analogous to an unweighted graph. The city has **N** houses numbered from 1 to **N** and are connected by **M** bidirectional roads. If a road is connecting two houses, say, **X** and **Y**, it means you can go from **X** to **Y** or **Y** to **X**. It is guaranteed that you can reach any house from any other house via some combination of roads. Two houses are directly connected at max by one road.

A path between house **S** to house **T** is defined as a sequence of some vertices from **S** to **T** where **S** is the starting house and **T** is the ending house, and there is a road connecting two consecutive houses in the chosen sequence. Basically, the path looks like (**S**, h<sub>1</sub>, h<sub>2</sub>, h<sub>3</sub>, ..., **T**); you have to find the shortest path from house **S** to house **T**.

### **Input Format:**

The **first line** of input contains two space-separated integers **N** and **M**, denoting the number of houses and the number of bidirectional roads, respectively.

The **second line** of input contains two space-separated integers, **S** and **T**, denoting the source house and the destination house numbers, respectively.

The next **M** lines contain two space-separated integers denoting the houses between which a bidirectional road exists.

### **Output Format:**

Return the smallest path from house **S** to house **T**.

### **Sample Input:**

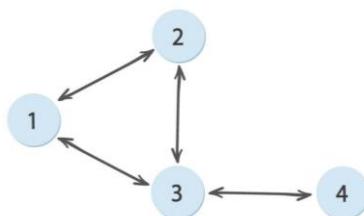
```
4 4
1 4
1 2
2 3
3 4
1 3
```

### **Output:**

```
(1, 3, 4)
```

### **Explanation:**

In the given graph, where 1 and 4 are the **starting** and **ending** houses in the path, there are two ways to go from house 1 to house 4, which are (1, 2, 3, 4) and (1, 3, 4). Here the second path (1, 3, 4) is the shortest path and, therefore the desired answer to be returned.



### **Approach 1: BFS Traversal**

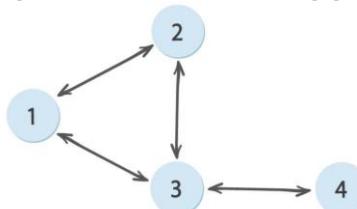
The idea here is to use BFS traversal in order to find the shortest path between any two given vertices.

We start our BFS from the given start node and then explore its neighbours like in any BFS traversal and repeat until the desired destination node is obtained.

1. Store all the edges in the form of an adjacency list **ADJ**. If  $ADJ[X][j] = Y$ , it means there is an undirected edge between X and Y.
2. Declare a queue **Q** and push **S** in it. Also, declare two arrays **VISITED** and **DISTANCE**, which will store whether a house is visited or not and the distance of the house from house **S**, respectively.
3. store the **PARENT**, which stores the node from which we will reach the current node. It will help us in building the path afterwards.
4. Mark **S** as visited and set **DISTANCE[S] = 0**
5. Do the following operations until the queue **Q** is not empty:
  - a. Select the vertex that is at the front end of the queue, say **X**. Remove it from the queue.
  - b. Iterate to all the neighbouring vertices of **X** using **ADJ**. Let's assume it to be **Y**. If **Y** is unvisited, do the following:
    - i. Push **Y** in the **Q**.
    - ii. Mark **Y** as visited.
    - iii. Set **DISTANCE[Y] = DISTANCE[X] + 1**
    - iv. Make the **PARENT[Y] = X**.
6. Once we do this, we will get the minimum distance of all vertices from **S**, and now we only need to rebuild the path. For this, we can take an array **PATH** and push **T** in it and mark **CUR = T**.
7. To rebuild the path, do the following until **CUR** is not equal to **S**:
  - a. **CUR = PARENT[CUR]**
  - b. Add **CUR** into the **PATH**
  - c. By this, we will trace the path in the reverse direction.
8. By doing this, we have recreated the path and now just reverse the **PATH** and return it.

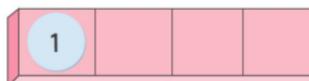
### Example:

Let us dry run the above algorithm for the following graph:

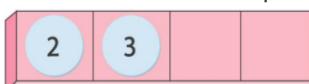


Here **N = 4** and **M = 4** and there exist edges between (1,2), (2,3), (1,3), and (3,4). Also Here we have **S = 1** and **T = 4**, so we start our BFS from node 1.

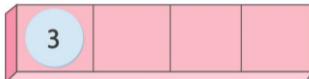
- Initially, the queue contains the node 1 in it.



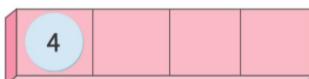
- In the first iteration, we have node 1 in front of Q, remove it from the queue, and add its neighbours—that is, 2 and 3 (since both the nodes are unvisited) in the queue and set distance of 2 and 3 equal to 2.



- Now in the second iteration, we have 2 in the front of Q. Remove it from the queue. Since there are no unvisited neighbours of 2 so we will move on.
- In the third iteration, we have 3 in the front of the queue; remove it. Since there is only one unvisited neighbour (4), set the distance of 4 equal to 3 and add 4 to the queue.



- Now in the fourth iteration, we have 4 in the front of Q. Remove it from the queue. Since there are no unvisited neighbours of 4, we will move on.



- Now the queue is empty, and we backtrack to get the shortest path as (1, 2, 3).

**Time Complexity:**  $O(N + M)$ , where **N** is the number of nodes and **M** is the number of edges. This is the time required for the BFS traversal of a graph with **N** nodes and **M** edges.

**Space Complexity:**  $O(N^2)$ , where **N** is the number of nodes in the graph. In the worst case, if the graph is connected and complete, the space requirement to store the graph using an adjacency list is of the order of  $O(N^2)$

## 9. Fire in the Cells [\[https://coding.ninja/P94\]](https://coding.ninja/P94)

**Problem Statement:** You are given a  $N \times M$  sized matrix, where **N** is the number of rows and **M** is the number of columns in the matrix. Value 0 in a cell represents that the cell is set on fire initially (at time  $t = 0$ ). The cells which don't have fire initially have a value = 1 and are called safe cells. If a cell is on fire, then in one second, the fire will expand to its adjacent cells (left, right, top, bottom) if they are not already on fire.

You are given the initial position of a person as  $(X, Y)$ , where **X** and **Y** are the integers denoting the cell  $(X, Y)$ . In one second, the person can move from its current cell to any one of its adjacent cells (left, right, top, bottom), provided they are not on fire. You have to determine if the person can move through the matrix to one of the escape cells without burning himself—that is, without going through the fire cells. If it's possible, return time taken, else return -1.

### Note:

- Escape cells in this problem are all such cells that lie on the edges of the matrix but not on the corner—that is, all such cells which are in the first row, first column, last row, and last column—excluding the four corner cells—are considered as valid escape cells.
- A cell, once set on fire, continues to remain on fire till the end.
- Note that rows are indexed from 0 to **N - 1**, and columns are indexed from 0 to **M - 1**.
- The escape cells may also be initially on fire or can catch fire from some neighbouring cell.

### Input Format:

The **first line** of input contains two space-separated integers **N** and **M**, denoting the number of rows and the number of columns in the matrix, respectively.

The next **N** lines contain **M** space-separated integers denoting the initial status of every cell. A cell has a value 0 if it is on fire initially otherwise, it has a value of 1, indicating it to be a safe cell.

The **next line** contains two space-separated integers **X** and **Y**, denoting the initial coordinates of the person in the grid.

### Output Format:

Return the time required by the person to reach an escape cell unburnt. If it is not possible, return -1.

**Sample Input:**

```
5 4  
0 1 1 1  
1 0 1 1  
1 1 1 1  
0 1 1 1  
0 0 0 0  
2 1
```

**Output:**

```
-1
```

**Explanation:**

The minimum possible time for the fire to reach any of the individual cells of the given matrix is as follows:

0 1 2 3	(Column index)
↓ ↓ ↓ ↓	
0 → 0 1 2 3	
1 → 1 0 1 2	
2 → 1 1 2 2	
3 → 0 1 1 1	
4 → 0 0 0 0	

(Row index)

The person's initial cell is (2, 1). At time  $t = 1$ , the person can only go to cell (2, 2) as the fire won't have reached the cell yet. At time  $t = 2$ , the person cannot go to any other cell as all the adjacent cells are at fire, and moving causes him to get burnt. So the answer is -1 as he wouldn't be able to reach any of the escape cells.

**Approach 1: BFS Approach**

1. We know that the fire is expanding with each passing second, and also there are multiple cells at fire initially (at time  $t = 0$ ). Let's create an auxiliary matrix **timeOfFire** of size **N\*M** to store the **minimum time** it will take for each cell to catch fire.
2. Iterate through the given input matrix **mat[][]** and set **timeOfFire[][]** for a cell  $(i, j)$  to 0, if **mat[i][j] = 0**, (source of fire) because they are on fire from the beginning only.
3. Since there are multiple sources of fire, the fire will expand in all four directions(up, down, left, right) from the fired cells and will continue to do so until all the cells are set to fire.
4. Thus a better choice is to use **multisource bfs** to store all the values of the matrix **timeOfFire[][]** for each cell in the matrix.
5. To do **multisource BFS**, do the following:
  1. Initialise a visited array/map. And mark all cells as false.
  2. Those cells which are set to 0 in **mat[][]**, push them into a queue and mark them as visited. Also set every cell's **timeOfFire[][]** = timer. Initialise **timer** as 0.
  3. **Timer** acts as clock time.
  4. Multi-source BFS works level by level from each of its source nodes. After each second, source nodes will change. Thus when pushing any cell in the queue, the minimum time for this cell to catch fire will be the minimum time taken by its parent +1. Here the children nodes will be nothing but the adjacent cells (left, right, top, and bottom).

- The next step is to find whether the person can reach an escape cell or not and if yes, then to figure out the minimum time required to do the same.
- For this, initialise **timer** = 0, and do BFS from the initial cell of the person (**X**, **Y**).
- Now to figure out if the next cell is possible to be visited by the person or not, we check for the next second—that is, use **timer + 1**.
- If **timer + 1 < timeOffFire** of the cell to be visited next, then it is possible to visit that cell safely; otherwise the person will get burnt. So in each iteration, insert only those cells in the queue for which **timer + 1 < timeOffFire** for the next cell.
- To identify if the person has reached escape cell via BFS or not, we can check for any of the below two conditions to hold true (since we are given that the escape cells are the cells present on the edges of the matrix but excluding the corner cells):
  1. Either row = 0 or row = **N - 1**:
    - a. col  $\geq 1$  and col  $< M - 1$ .
  2. Either col = 0 or col = **M - 1**.
    - a. row  $\geq 1$  and row  $< N - 1$ .
- If the person escapes, then return timer value, else return -1.

**Time Complexity:**  $O(N * M)$ , where **N** is the number of rows and **M** is the number of columns in the given matrix.

**Space Complexity:**  $O(N * M)$ , where **N** is the number of rows and **M** is the number of columns in the given matrix.

## 10. Strongly Connected Components (Tarjan's Algorithm)

[<https://coding.ninja/P95>]

**Problem Statement:** You are given an unweighted directed graph of **V** vertices and **E** edges. Your task is to print all the strongly connected components (SCCs) present in the graph.

### Input Format:

The **first line** of input contains an integer **T** denoting the number of test cases.

The **first line** of every test case contains two space-separated integers **V** and **E**, denoting the number of vertices and the number of edges present in the graph.

The next **E** lines contain two space-separated integers, **a** and **b**, denoting a directed edge from vertex **a** to vertex **b**.

### Output Format:

For each test case, print space-separated vertices present in the strongly connected components of the graph, print the output for one SCC on each line.

### Sample Input:

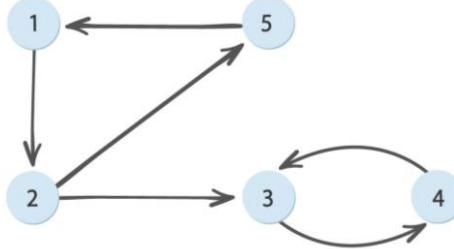
```
1
5 6
1 2
2 3
2 5
3 4
4 3
```

**Sample Output:**

```
0 1 4
2 3
```

**Explanation:**

For the given test case, the input directed graph is as shown in the figure. There are two SCCs in the graph, which are enclosed in the boxes as shown in the image.

**Tarjan's Algorithm**

- Tarjan's algorithm can determine the SCCs in a graph using just a single DFS traversal. Before getting into details of Tarjan's algorithm, we should understand the meaning of the terms: **discovery time of a node** and **low-link value**.
- The **discovery time** of a node is the time at which the node was discovered during the DFS traversal. This can be represented by an integer value which also specifies the order in which the nodes are visited during the DFS.
- The **low-link value** of a node is the smallest (lowest) discovery time reachable from that node during the DFS, including the node itself.
- For an SCC in a graph, every node which is a part of the same SCC will have the same **low-link** value.
- As the discovery time of a node depends on the order in which the graph is traversed in a DFS, it may be possible that multiple nodes may have the same low-link value even though they are a part of different SCC.
- This is where Tarjan's SCC algorithm comes in. The algorithm maintains an invariant (**stack**), which prevents the **low-link** values of two different SCCs from interfering with each other.
- The invariant used is a **stack**, which is different from the stack being used for DFS (the recursion stack). The stack stores the nodes forming a subtree from which the SCC will be found. Nodes are placed in the stack in the order in which they are visited during the DFS traversal. But the nodes are removed only when a complete SCC is found.

The algorithm for the discussed approach is as follows:

**findSCC(vector<vector<int>> graph):**

1. Create two arrays—**low** and **disc**, to store the low-link value and discovery time of the nodes, respectively.
2. Initialise all the entries of **low** and **disc** to -1 as no node has been visited initially.
3. Create an empty stack, which will be used as an invariant (as explained in the approach).
4. Create a global variable **time**, which will be used to mark the discovery time as the nodes are visited during DFS. Initialise **time** to 0.

5. Call the DFS procedure for all the unvisited nodes of the graph.
6. Return all the strongly connected components so found.

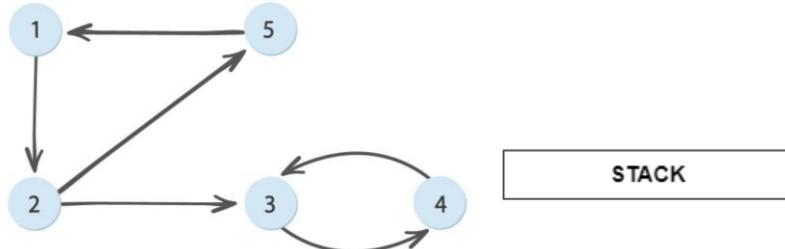
**DFS(node u, low[ ], disc[ ], stack):**

1. Initialise the low-link value and discovery time of the current node with **time**. Also, increment the **time**.
2. Push the **node u** in the (invariant) stack.
3. Loop: For each **node v**, which is adjacent to the **node u**:
  - a. Case 1: Node **v** is not visited till now. Therefore edge **u→v** is a **tree edge** in the DFS tree. So,
    - i. Recursively call DFS procedure on node **v**.
    - ii. Then update the low-link value of node **u** as **low[u] = min(low[u], low[v])**.
  - b. Case 2: Node **v** has already been visited using the DFS. Therefore edge **u→v** is a **back edge** in the DFS tree. So,
    - i. Check whether node **v** is present in the invariant stack or not.
    - ii. If node **v** is present, then update the low-link value of node **u** as **low[u] = min(low[u], disc[v])**.
4. Check whether the current **node u** is the head/root of an SCC—that is, it satisfies the condition: **low[u] = disc[u]**:
  - a. We have found a new SCC.
  - b. Pop all the elements from the stack until the **node u** gets popped.
  - c. Add every element to a list.
  - d. Add this list to the list of SCCs.

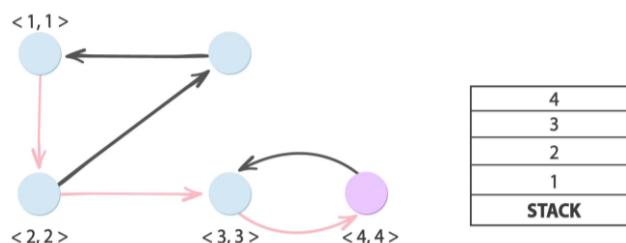
**Example:**

Let us understand this by an example:

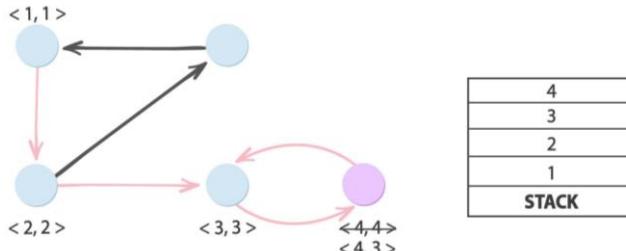
- Consider a graph as shown below. Initially, our invariant stack is empty. The green node marks the current node in the DFS traversal.



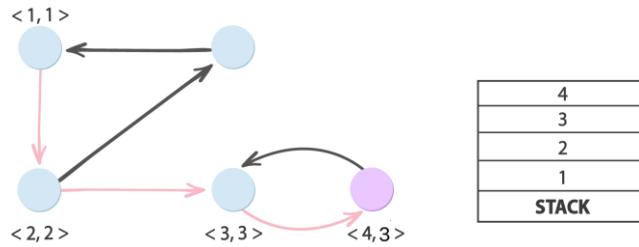
Starting the DFS traversal from the previously marked green node. We move along the edges marked by orange colour. As we traverse the graph, nodes are pushed into the stack in the order in which they are visited during the DFS traversal. The discovery time and the low-link value of any node **u** are shown in the image as **<disc[u], low[u]>**.



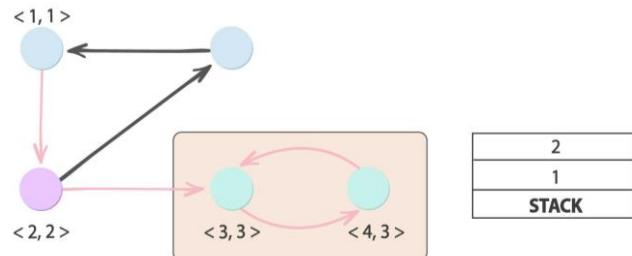
- During the exploration of node 4, we find that the node has a back edge to node 3, Hence we update the low-link value of node 4 as  $\text{low}[4] = \min(\text{low}[4], \text{disc}[3]) = \min(4, 3) = 3$ .



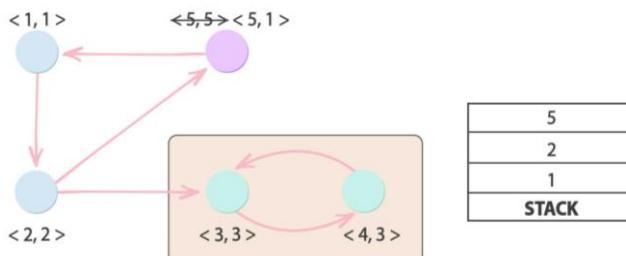
- Now the node 4 is completely explored (hence it is now marked orange), but it is not removed from the stack. We move back to node 3.
- The exploration of node 3 is completed as all its adjacent nodes have been visited (so it's marked orange).
- According to step 4 in the DFS algorithm. We find that  $\text{low}[3] = \text{disc}[3]$  indicates that node 3 is the head of an SCC; therefore the algorithm has found an SCC with node 3 as the head node.



So we pop all the elements from the stack until node 3 gets popped. These form an SCC of the graph. This SCC is highlighted using a box, as shown in the image.

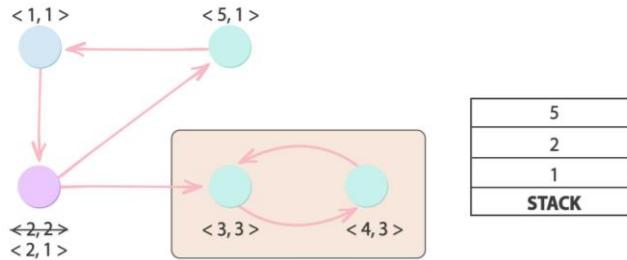


- We move back to node 2 to explore its remaining neighbours.
- There is only one unvisited neighbour of node 2; its discovery time is set to 5. As node 5 has a back edge to node 1, also node 1 is present in the stack, hence we update the low-link value of node 5 as  $\text{low}[5] = \min(\text{low}[5], \text{disc}[1]) = \min(5, 1) = 1$ .

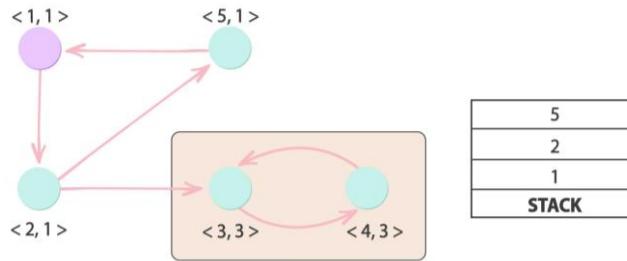


Now node 5 has been explored, but it is not removed from the stack. We move back to node 2.

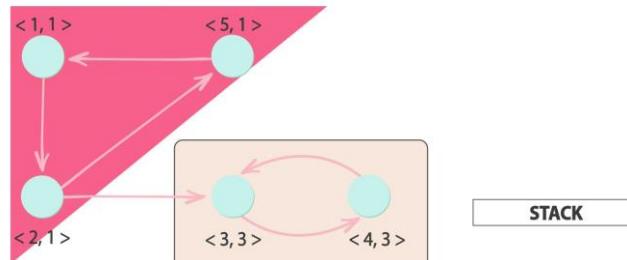
- As the neighbour node 5 is completely explored, we can now update the low value of node 2 (according to step 3, CASE 1) as  $\text{low}[2] = \min(\text{low}[2], \text{low}[5]) = \min(2, 1) = 1$ .



- Now the exploration of node 2 is completed, but it is not removed from the stack.
- We move back to node 1, according to step 4 in the DFS algorithm. We find that  $\text{low}[1] = \text{disc}[1]$  indicates that node 1 is the head of an SCC.



So, we pop all the elements from the stack until node 1 gets popped. These form an SCC of the graph. This SCC is highlighted using a blue box, as shown in the image.



- There are two SCCs in the graph:  $\{1, 2, 5\}$  and  $\{3, 4\}$ .

**Time Complexity:**  $O(V + E)$  per test case, where **V** is the number of vertices and **E** represents the number of edges in the graph. In the worst case, Tarjan's Algorithm performs a single DFS to obtain all the SCCs.

**Space Complexity:**  $O(V)$  per test case. In the worst case, **O(V)** space is required for storing the low-link values, discovery time, invariant stack, and the recursion stack of DFS.

# 11. Recursion and Backtracking

---

## Introduction to Recursion

When a function calls itself, it is called recursion. **A recursive method solves a problem by calling a copy of itself to work on a smaller version (smaller input) of the same problem.** In recursion, every time the function calls itself with a slightly simpler version of the original problem, however, this sequence of smaller problems must eventually converge on a base case. A recursive approach can be summarised to be constituted of the following three steps:

- **Base Case:** A recursive function must have a terminating condition and such a condition is known as the base case. In the absence of a base case, the function will keep on calling itself and get stuck in an infinite loop. Soon, the recursion depth will exceed the threshold value and it will throw an error.
- **Recursive Call (Smaller problem):** The recursive function will invoke itself on a **smaller version** of the main problem. We need to be careful while writing this step as it is crucial to correctly figure out what the smaller problem is.
- **Small Calculation (Self-work):** Generally, we perform a calculation step in each recursive call. We can achieve this calculation step before or after the recursive call depending upon the nature of the problem.

**Note:** Recursion uses an in-built stack that stores all the recursive calls. Hence, the number of recursive calls must be as small as possible to avoid memory-overflow. If the number of recursive calls exceeds the maximum permissible amount, this condition is called **stack overflow** which throws an error.

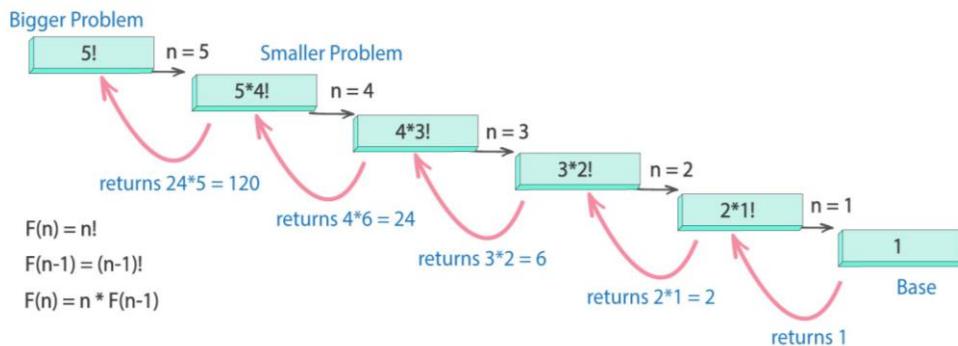
Now, let us see how recursion is used to solve some problems.

### Finding the Factorial of a Given Number n

Factorial of any number n is denoted as  $n!$  and can be calculated as  $n! = n * (n - 1) * (n - 2) * \dots * 1$ .

For example,  $5! = 5 * 4 * 3 * 2 * 1 = 120$ .

Now the recursive calls can be made to calculate the factorial of 5.



In recursion, the main idea is to represent the problem in terms of a smaller problem of the same nature.

We know that  $5! = 5 * 4!$ . Let's assume that recursion gives us the correct answer for  $4!$ . Now to get the solution to our problem, we need to simply multiply the calculated answer by recursion with 5 and then return the result as our answer. Similarly, when we give a recursive call for  $4!$ ; we assume recursion correctly calculates the answer for  $3!$  and so on. Since the same work is done (a smaller problem is of the nature as the main problem) in all these steps, we write only one function and give it a call recursively.

What if there is no base case? Let's start with  $1!$ . It will give a call to  $0!$ ,  $0!$  will then give a call to  $-1!$  which does not exist and this will continue. Soon the function call stack will be full of method calls and this will give us an error of **Stack Overflow**. Thus to avoid this, we need a base case to terminate.

```
function factorial(n)
    // base case
    if n equals to 0
        return 1

    // getting answer of the smaller problem
    recursionResult = factorial(n - 1)

    // self work
    ans = n * recursionResult
    return ans
```

## Check if an Array is Sorted or not recursively

For example:

- If the input array is [2, 4, 8, 9, 9, 15], then the output should be **true**, since the array is sorted.
- If the input array is [5, 8, 2, 9, 3], then the output should be **false**, since the array is unsorted.

We consider a single element present in the array to be sorted.

So now we could ask recursion to return whether the remaining array is sorted or not and then compare the current element with the last element of the remaining portion of the array.

Below is the pseudo code for the same:

```
function isArraySorted(arr, idx) // 0 is passed in idx

    // base case
    if idx equals arr.length - 1 // indicating one single element is sorted
        return true

    // getting answer of the smaller problem
    recursionResult = isArraySorted(arr, idx + 1)

    // self work
    ans = recursionResult & arr[idx] <= arr[idx + 1]
    return ans
```

## Divide and Conquer

**Divide and conquer** is an algorithm design paradigm. A divide-and-conquer algorithm recursively breaks down a problem into two or more sub-problems of the same or related type, until these subproblems become simple enough to be solved directly. The solutions to these sub-problems are then combined together to give a solution to the original problem.

Following are a few sorting techniques using divide and conquer:

## Merge Sort

Merge sort requires dividing a given array/list into two halves until it can no longer be divided. By definition, if there is only one element in the list, it is considered to be sorted. Then, merge sort combines/merges these smaller sorted lists together,,, keeping the new list sorted too.

- If there is only one element in the list,, it is considered to be sorted, return.
- Divide the list recursively into two halves until it can't be divided further.
- Merge the smaller sorted lists into a new list in a sorted order.

```

/*
    array from leftidx(0) to rightidx(arr.length - 1) is considered
*/
function mergeSort(arr, leftidx , rightidx )

    // base case: only 1 element is left to be sorted
    if leftidx == rightidx
        return

    middle =leftidx + (rightidx - leftidx) / 2

    // smaller problems
    mergerSort(arr, leftidx, middle)
    mergeSort(arr, middle + 1, rightidx)

    // selfwork
    merge(arr,leftidx , middle, rightidx)

function merge(arr, leftidx , middle , rightidx)

    leftlo = leftidx
    rightlo = middle+1
    idx = 0
    // create an array temp of size (rightidx - leftidx + 1)

    while leftlo <= middle AND rightlo <= rightidx
        if arr[leftlo] < arr[rightlo]
            temp[idx] = arr[leftlo]
            leftlo++
            idx++
        else
            temp[idx] = arr[rightlo]
            rightlo++
            idx++

    while leftlo <= middle
        temp[idx] = arr[leftlo]
        leftlo++
        idx++

    while rightlo <= rightidx
        temp[idx] = arr[rightlo]
        rightlo++
        idx++

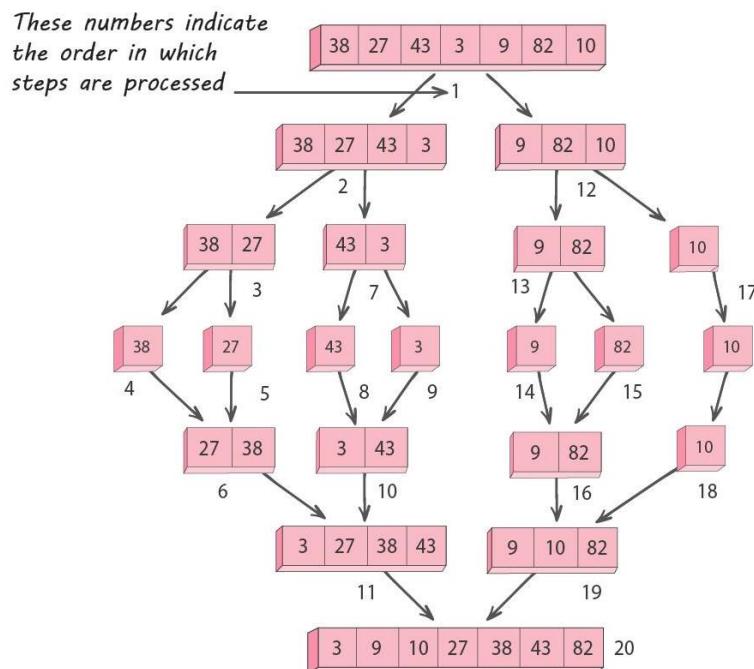
    // copy temp to original array
    count = 0
    while count < temp.length AND leftidx <= rightidx
        arr[leftidx] = temp[count]
        leftidx++

```

count++

The following diagram shows the complete merge sort process for an example array [38, 27, 43, 3, 9, 82, 10].

If we look at the diagram, we can see that the array is recursively divided into two halves till the size becomes 1. Once the size becomes 1, the merge processes come into action and start merging arrays back till the complete array is merged.



Merge Sort suffers from the disadvantage of **not being an in-place sorting** technique—that is, it creates a copy of the array and then works on that copy.

**Time Complexity:**  $O(N \log(N))$ , where **N** is the number of elements in the array. Since at every step we are dividing the array into two halves requiring  $\log(N)$  time and then copying the array in a new array which requires  $O(N)$  time, thereby making the overall complexity be  $O(N \log(N))$ .

**Space Complexity:**  $O(N)$ , since we are making an auxiliary array of size **N**.

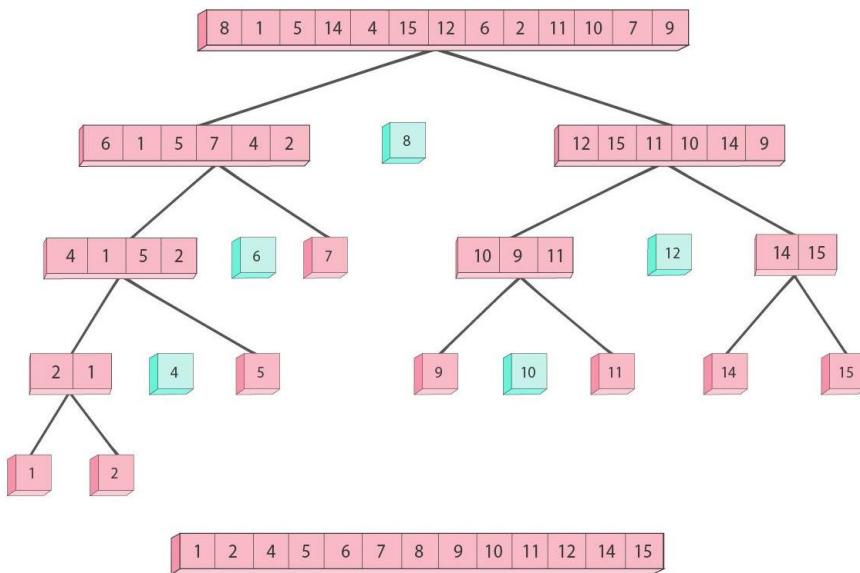
## QuickSort:

Quick sort is an in-place sorting algorithm based on the divide and conquer algorithm paradigm and is thus considered to be better than Merge Sort as it does not make use of any auxiliary array.

Based on the Divide-and-Conquer approach, the quicksort algorithm can be explained as follows:

- **Divide:** The array is divided into subparts taking a pivot as the partitioning point. The elements that are **smaller** than the pivot element are placed to the **left** of the pivot,, and the elements that are **greater** than the pivot element are placed to the **right** side of the pivot in the array.
  - **Conquer:** The left and the right sub-parts are again partitioned by selecting the respective pivot elements for them. This can be achieved by recursively passing the subparts into the algorithm.
  - **Combine:** This part does not play a significant role in quicksort since the array is already sorted at the end of the conquer step.

The following diagram shows the complete working of a quicksort process for an example array [8, 1, 5, 14, 4, 15, 12, 6, 2, 11, 10, 7, 9], where every time the first element is chosen as the pivot element.



- In step 1, 8 is taken as the pivot.
  - In step 2, 6 and 12 are taken as pivots.
  - In step 3, 4 and 10 are taken as pivots.
  - We keep dividing the list about pivots till there is only one element left in the sublist.

```
/*
    array from lo(0) to hi(arr.length-1) is considered
*/
function quickSort(arr, lo, hi )
```

```

return

pivot = arr[lo]

// partitioning
left = lo
right = hi

while left <= right

    // move left pointer
    while arr[left] < pivot
        left++

    // move right pointer
    while arr[right] > pivot
        right--

    // problem solve : swap
    if left <= right
        temp = arr[left]
        arr[left] = arr[right]
        arr[right] = temp
        left++
        right--

    // smaller problems
    quickSort(arr, lo, right)
    quickSort(arr, left, hi)

```

**Time Complexity:**  $O(N \log N)$ , where **N** is the number of elements in the array. In the worst case, it is  $O(N^2)$ .

**Space Complexity:**  $O(N)$  as a call stack is being used.

## Introduction to Backtracking

**Backtracking** is a famous algorithmic technique for solving/resolving problems recursively by trying to build a solution incrementally. Solving one piece at a time and removing those solutions that fail to satisfy the constraints given in the problem at any point in time is called the process of backtracking.

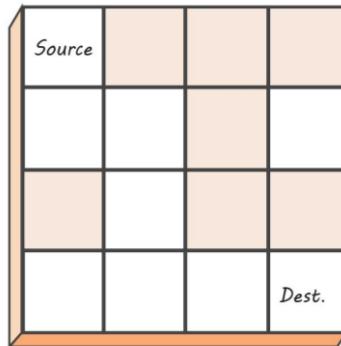
In other words, backtracking can be termed as a general algorithmic technique that considers searching every possible combination of a solution to solve a computational problem.

## Sample Problems

### Rat in a Maze Problem:

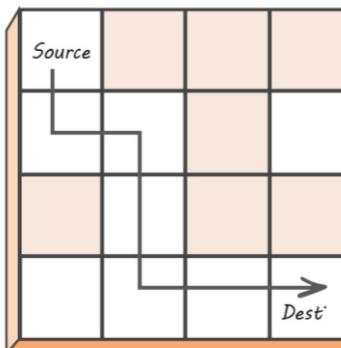
**Problem statement:** Given a 2D grid —where some cells are marked free,, and some are marked as blocked(for simplicity, assume that the grid is a **square** matrix of size **N**). Given a source cell **S** and a destination cell **D**, we need to find whether there is a path from source to destination cell in the maze that passes only through the **free cells** of the matrix.

Let us assume that the source **S** is at the **top-left corner** of the maze and destination **D** is at the **bottom-right corner** of the maze, and the movements allowed are moving to the **right** or **down** from the current cell of the maze.



The **brown-coloured** cells represent the **blocked** cells,, and **white-coloured** cells represent the **free** cells.

A possible path from **source** to **destination** traversing through the free cells is highlighted below:



**Note:** There can be multiple possible paths from the source to the destination cell.

### Approach:

We will make use of the backtracking technique to find whether there exists a path from source to destination cell in the given maze or not.

### **Steps:**

1. If the destination cell—that is, **the bottom-right cell of the maze** is reached, return true.
2. Else:
  - a. Check if the current position is a **valid** position—that is, the position is within the **bounds** of the maze and also a **free** position in the maze.
  - b. Mark the current position as 1, denoting that the position can lead to a possible solution.
  - c. Move **forward**—that is, move to the **right cell** from the current position and recursively check if this move leads us to reach the destination cell.
  - d. If the above move fails to reach the destination, then move **downward** and recursively check if this move leads us to reach the destination cell.
  - e. If none of the above moves leads to the destination cell, mark the current position as 0, denoting that it is **impossible** to reach the destination from this position.

### **Pseudocode:**

```
function isValid(x, y, N)

/*
    Check if the position is within the bounds of the maze and that the position is
    not a blocked cell.
*/

if(x <= N and y <= N and (x, y) cell is not blocked)
    return true
else
    return false

function RatMaze(maze[][], x, y, N)

/*
    (x, y) is the current position of the rat in the maze.
    Check if the current position is valid.
*/

if isValid(x, y, N)
    mark[x][y] = 1
else
```

```

return false

 $/*$ 
If the current position is the bottom-right corner of the maze, then we have
found a solution.
 $*/$ 

if x equals N and y equals N
    mark[x][y] = 1
    return true

 $/*$ 
Otherwise, try moving forward or downward to look for other possible
solutions.
 $*/$ 

bool found = RatMaze(maze,x+1,y,N) OR RatMaze(maze,x,y+1,N)

 $/*$ 
If a solution is found from the current position by moving forward or
downward, then return true, otherwise mark the current position as 0, as it is
not possible to reach the destination cell of the maze from the current
position.
 $*/$ 

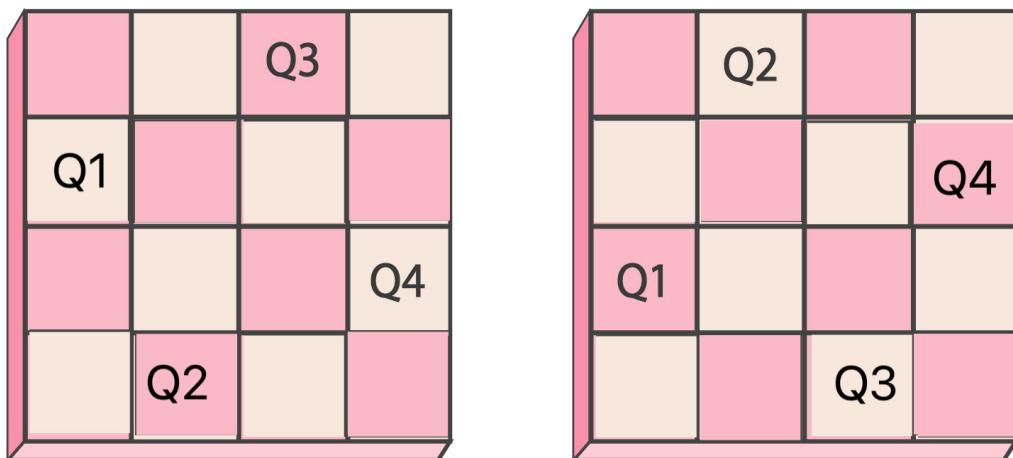
if(found)
    return true
else
    mark[x][y] = 0
    return false

```

## N-Queens Problem

**Problem statement:** Given an **N × N** chessboard, we are required to arrange **N queens** on the board in such a way that **no two queens** attack each other. A queen can attack horizontally, vertically, or diagonally.

Below is a diagram showing how **four queens** can be placed on a chessboard of size **4 × 4**, such that no two queens attack each other.



The above figure shows two possible **valid arrangements** of four queens (represented by **Q1**, **Q2**, **Q3**, **Q4**) on the chessboard of size  $4 \times 4$ .

#### Approach:

1. There can be multiple valid arrangements for the queens on the chessboard.
2. We will use the backtracking technique to find a possible valid arrangement of the queens on the given chessboard.
3. We place the first queen arbitrarily anywhere on the chessboard and then continue to place the next queen at a position that can not be attacked by any other queen placed so far. If no such position is present, we backtrack and change the position of the previous queen.
4. A solution is found if we can place all the queens on the chessboard.

#### Steps:

1. Place a queen **arbitrarily** at any position on the chessboard.
2. Check if this position is **safe**—that is, it is not being attacked by any other queen placed so far.
3. If the position is not safe, then look for other positions on the board, and if no such position is found, then return false as we cannot place any more queens.
4. If the position is safe, then recursively check for **Q - 1** queens. If the function returns true, it indicates that we were successfully able to place all the queens on the chessboard, satisfying the problem constraints.

**Note:** The **Q** in the function argument denotes the number of queens to be placed, which is initially **N** as we need to place **N** queens on the board.

**Pseudocode:**

```
function isValid(x,y,board[],N)

/*
    Check if the position (x,y) is not attacked by any other
    queen.
    1. Check if no position is marked 1 in the same row.
    2. Check if no position is marked 1 in the same column.
    3. Check if no position is marked 1 in the diagonals.
*/

for i = 0 to N - 1
    if board[x][i] equals 1 or board[i][y] equals 1
        return false

tempx = x
tempy = y

while tempx >= 0 and tempy < N
    if board[tempx][tempy] equals 1
        return false
    tempx -= 1
    tempy += 1

tempx = x
tempy = y

while tempx < N and tempy >= 0
    if board[tempx][tempy] equals 1
        return false
    tempx += 1
    tempy -= 1

tempx = x
tempy = y

while tempx >= 0 and tempy >= 0
    if board[tempx][tempy] equals 1
        return false
    tempx -= 1
    tempy -= 1

tempx = x
```

```

tempy = y

while tempx < N and tempy < N
    if board[tempx][tempy] equals 1
        return false
    tempx += 1
    tempy += 1

    // The position is safe, return true.
    return true

function N-Queens(Q, board[], N)

    // Q represents the number of queens to be placed on the board.

    // Base Case, when all queens have been placed
    if Q equals 0
        return true

    /*
        For each possible position on the board, check if the position is safe.i.e. it is
        not attacked by any other queen placed so far.
    */

    for i = 0 to N - 1
        for j = 0 to N - 1
            bool can = isValid(i, j, board, N)

            /*
                If the position is safe, mark the board's position check
                recursively for Q - 1 queens if they can be placed successfully.
            */

            if isValid is true
                board[i][j] = 1
                bool solve = N-Queens(Q - 1,board,N)

            /*
                The remaining queens can be placed successfully, return
                true, otherwise, unmark the position on the board, and
                check for other possible options.
            */

```

```

        if solve is true
            return true
        else
            board[i][j] = 0
        else
            continue
    /*
        Since there was no possible option to place a queen on the board, so return
        false.
    */

    return false

```

## Practice Problems

### **1. Reverse a Stack Using Recursion :** [\[https://coding.ninja/P96\]](https://coding.ninja/P96)

**Problem Statement:** Reverse a given stack of integers using recursion.

**Note:**

You cannot use any extra space other than the internal stack space used due to recursion.  
You are not allowed to use the loop constructs such as for, for-each, while, etc. The only inbuilt stack methods allowed are:

**push(x):** Push element **x** onto the stack.

**pop():** Remove the element present on the top of the stack.

**top():** Get the topmost element of the stack.

**Input Format:**

The **first line** of input contains an integer value **N**, denoting the size of the input stack.  
The **second line** contains **N** single space-separated integers, denoting the stack elements, where the last (**N<sup>th</sup>**) element is the **topmost** element of the input stack.

**Output Format:**

**N** single space-separated integers in a single line, where the first integer denotes the **TOP** element of the reversed stack.

**Sample Input:**

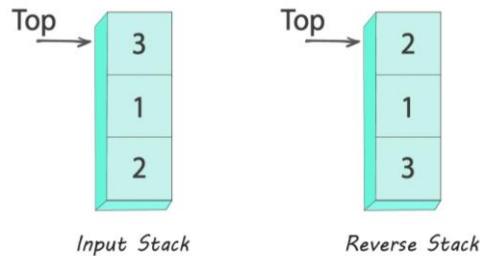
3  
2 1 3

### Sample Output:

2 1 3

### Explanation:

The input stack and the required reversed stack are as shown in the figure:



Printing the reverse stack starting from the top element gives 2 1 3 as the output.

### Recursive Approach:

We will be using two recursive methods:

1. **To Reverse the Stack:** We will use recursion to iterate through the stack. For each top element, we will pop it and use recursion to reverse the remaining stack. After getting the stack reversed by recursion, we can simply push the popped element to the bottom of the stack. The procedure to push the element at the bottom of the stack is explained in the following paragraph.
2. **To Push an element to the Bottom of the Stack:** We will be again using recursion for this. Now to push a given element to the bottom of the stack, we will iterate through the stack recursively. For each top element, we will pop it and call recursion to insert a given element at the bottom of the remaining stack. After the recursive function does its work and adds the given element to the bottom of the stack, we can simply push the popped element to the top of the stack.

### Steps:

1. **reverseStack(stack):** The function responsible for reversing the input stack.
  - a. If the stack is empty, then return.
  - b. Pop the top element from the stack as the **top**.
  - c. Reverse the remaining elements in the stack, call **reverseStack(stack)** method.
  - d. Insert the **top** element to the bottom of the stack, call **insertAtBottom(stack, top)** method.
2. **insertAtBottom(stack, ele):** The function responsible for pushing an element to the bottom of the stack.
  - a. If the stack is empty, push **ele** to stack, return.
  - b. Pop the top element from the stack as the **top**.

- c. Insert the **ele** to the bottom of the stack, call **insertAtBottom(stack, ele)** method.
- d. Push **top** to the stack.

**Time Complexity:**  $O(N^2)$ , where **N** denotes the total number of elements in the given stack. The **reverseStack** method will be called for each element in the stack that requires  $O(N)$  time. And for each call of the **reverseStack**, we call **insertAtBottom**, which also takes  $O(N)$  time, making the total time complexity of  $O(N^2)$ .

**Space Complexity:**  $O(N)$ , where **N** denotes the total number of elements in the given stack.

## 2. Return Subsets Sum to K [<https://coding.ninja/P97>]

**Problem Statement:** Given an integer array **arr** of size **N** and an integer **K**, return all the subsets of **arr** that sum to **K**. Subset of an array **arr** is a tuple obtained from **arr** by removing some or none of the elements of **arr**.

### Input Format:

The **first line** of input consists of a single integer **N**, which denotes the size of the array.  
 The **second line** consists of **N** single-space separated integers representing the elements of the array.  
 The **third line** consists of a single integer **K**, which denotes the integer to which the subsets should sum to.

### Output Format:

Print each subset in a separate line. The elements of each subset should be single-space separated.

### Sample Input:

```
3
2 4 6
6
```

### Sample Output:

```
2 4
6
```

### Explanation:

For the array **arr** = {2, 4, 6}, we can have subsets {}, {2}, {4}, {6}, {2, 4}, {2, 6}, {4, 6}, {2, 4, 6}. Out of these 8 subsets, {2, 4} and {6} sum to the given **K**, that is, 6.

### Brute Force Approach

In this approach, we recursively generate all the subsets and keep track of the sum of the elements in the current subset. Subsets can be generated in the following way. For every element of the array, there are two options:

1. **Include the element in the current subset:** If we include the element in the current subset, we decrease K's value by the value of the element.
2. **Do not include the element in the current subset:** There is no effect on the value of K,, and we can simply move onto the next element.

In any step, if the value of K becomes 0, then we have found a valid subset that sums to K. We store all these subsets and return them.

**Time Complexity:**  $O(2^N)$ , where  $N$  denotes the number of elements in the integer array **arr**. Total  $2^N$  subsets are possible for an integer array **arr** of size  $N$ , with a subset having a maximum size as  $N$ .

**Space Complexity:**  $O(N)$ ,  $N$  denotes the number of elements in the integer array A. Total  $2^N$  subsets are possible for an integer array **arr** of size  $N$ , with a subset having a maximum size as  $N$ .

### **3. Reach the Destination :** [\[https://coding.ninja/P98\]](https://coding.ninja/P98)

**Problem Statement:** Given a source point (**sx, sy**) and a destination point (**dx, dy**), your task is to check if it is possible to reach the destination point from the given source point using the set of following valid moves:

$(a, b) \rightarrow (a + b, b)$

$(a, b) \rightarrow (a, a + b)$

Return true if it is possible to reach the destination point using only the above mentioned valid moves, else return false.

#### **Input Format:**

The **first line** of the input contains an integer **T** representing the number of test cases.

Then the test cases follow.

The **only line** of each test case contains four space-separated integers, **sx, sy, dx and dy**, where (**sx, sy**) represent the coordinates of the source point and (**dx, dy**) represent the coordinates of the destination point.

#### **Sample Input:**

```
1
1 1 3 5
```

#### **Sample Output:**

True

#### **Explanation:**

The output will be true since the destination point (3, 5) can be reached using the following sequence of moves in order:

(1, 1) → (1, 2) → (3, 2) → (3, 5).

### Approach 1: Brute Force Approach

The naive approach to solve this problem is to consider every possible valid move until we reach the destination point. This can be done using recursion.

#### Steps:

1. Check for the base cases:
  - a. If the source and destination coordinates are the same, return true.
  - b. Return false if the **x (or y)** coordinate of the source point is greater than the **x (or y)** coordinate of the destination point as there is no path to reach the destination.
2. Perform the following two set of recursive calls to check if the destination is reachable from either of the two:
  - a. Check if the destination point can be reached by replacing the **x** coordinate with the sum of the coordinates of the source point in the recursive call.
  - b. Check if the destination point can be reached by replacing the **y** coordinate with the sum of the coordinates of the source point in the recursive call.
  - c. Return true; if either of the above two calls returns true, else return false.

**Time Complexity:**  $O(2^{\max(x, y)})$ , where **(x, y)** is the coordinate of the destination point. Since, for every valid pair of coordinates, each recursive call will end up in two recursive calls. Thus, the final time complexity will be exponential—that is,  $O(2^{\max(x, y)})$ .

**Space Complexity:**  $O(\max(x, y))$ , where **(x, y)** is the coordinate of the destination point. We are using  $O(H)$  extra space for the call stack used in recursion, where **H** is the height of the recursion tree. The value of **H** could be  $\max(x, y)$  in the worst case.

### Approach 2: Optimised Approach

The main idea of this approach is to iterate backwards from the destination point (**dx, dy**) to the source point (**sx, sy**). We can do so by determining which coordinate (**x** or **y** of the destination point) was last changed by selecting the larger of the two. This will reduce the time complexity from exponential to linear since we make just one recursive call for each valid pair of coordinates.

#### Steps:

1. Check for the base cases:
  - a. If the source and destination coordinates are the same, return true.
  - b. Return false if the **x (or y)** coordinate of the source point is greater than the **x (or y)** coordinate of the destination point respectively as there is no path to reach the destination.
2. Perform the following recursive call:
  - a. If **dx > dy**, make a recursive call with parameters **(sx, sy, dx - dy, dy)**.
  - b. Else make a recursive call with parameters **(sx, sy, dx, dy - dx)**.

**Time Complexity:**  $O(\max(x, y))$ , where  $(x, y)$  is the coordinate of the destination point. Since, for every valid pair of coordinates, each recursive call will end up in only one recursive call. Thus, the final time complexity will be  $O(\max(x, y))$ .

**Space Complexity:**  $O(\max(x, y))$ , where  $(x, y)$  is the coordinate of the destination point. We are using  $O(H)$  extra space for the call stack, where  $H$  is the height of the recursion tree. The value of  $H$  could be  $\max(x, y)$  in the worst case.

## 4. Binary Strings With No Consecutive 1s [\[https://coding.ninja/P99\]](https://coding.ninja/P99)

**Problem Statement:** Given an integer **K**, generate all the possible binary strings of length **K** such that there are no consecutive **1's** in the strings. This means that the binary string should not contain any instance of **1's** coming together one after the other.

A binary string is a string that contains only '**0**' and '**1**' as its characters.

### Note:

1. Each string must be a binary string.
2. There should be no consecutive **1's** in the string.

### Input Format:

The **first line** of input contains a single integer **T** denoting the total number of test cases.

The **first and only line** of each test case contains a single integer **K** denoting the length of the binary strings to be generated.

### Output Format:

Return an array of all possible binary strings without consecutive **1's** of the length **K**.

### Sample Input:

```
1
4
```

### Sample Output:

```
0000 0001 0010 0100 0101 1000 1001 1010
```

### Explanation:

For  $K = 4$ , we get the following set of valid strings: 0000 0001 0010 0100 0101 1000 1001 1010.

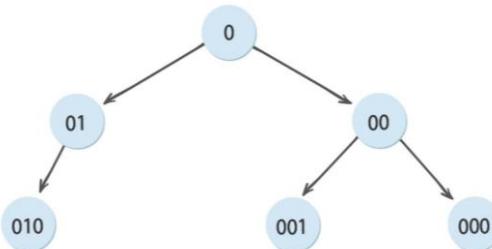
Note that none of the strings has consecutive **1's** in them. Also, note that they are in a lexicographically increasing order.

### Recursive Approach:

Since we need to generate the set of all possible valid strings which do not have consecutive **1's** in them, we can simply start by adding new characters to the current string until the length of the string becomes  $K$ .

We need to take care that if the last character of the current string is '**1**', we cannot add another '**1**' as it will result in consecutive **1's**. If the last character is '**0**', we have two options—to either add '**1**' or '**0**'. We explore both options recursively until we have a string of desired length  $K$ .

For example, let's say that  $K = 3$  and the first character is '**0**'.

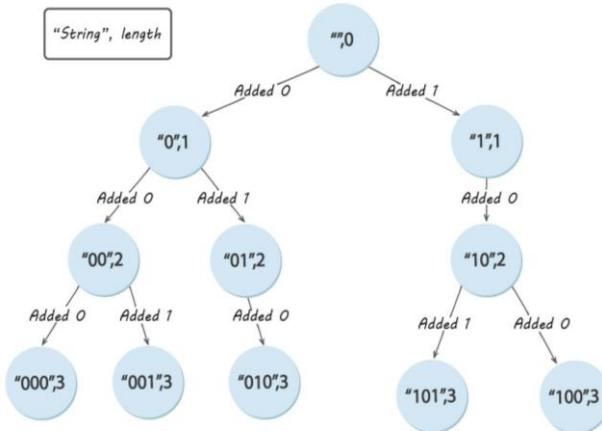


From the figure, it is clear that we get three unique strings with the starting character as '**0**'. Similarly, we will get all possible valid strings of length ( $K = 3$ ) with '**1**' as the starting character.

### Steps:

1. Check the last character of the current string. If the last character is '**0**', we can add a '**0**' or '**1**' at the end of the current string.
2. If the last character is '**1**', we can add only '**0**' and not '**1**'. This is because if we add '**1**', we violate the condition that there must be no consecutive '**1**' in the string.
3. Once the length of the string is equal to  $K$ , we add it to our answer array.
4. Lastly, we sort the array and return all possible valid strings so formed of length  $K$  with no consecutive **1's** in them.

Following is the detailed order in which the recursion calls occur for length = 3.



**Time Complexity:**  $O(2^N)$ , where  $N$  denotes the length of the string. For every character, we have 2 choices—that is, to put either a '0' or a '1'. Hence, for  $N$  choices, we will have a total time complexity of  $O(2^N)$ .

**Space Complexity:**  $O(N)$ , where  $N$  denotes the length of the string.

## 5. Replace 'O' with 'X' [\[https://coding.ninja/P100\]](https://coding.ninja/P100)

**Problem Statement:** Given a 2D array/grid  $\mathbf{G}$  of **O's** and **X's**. The task is to replace all '**O'** contained in an island with '**X**'. Here an island is a set of **O's** connected horizontally or vertically and surrounded by '**X**' from all of its boundaries. (Boundary includes top, bottom, left, and right cells in the grid).

### Example:

```

{{X, X, O, X, X, X},
 {X, X, O, X, O, X},
 {X, X, X, O, O, X},
 {X, O, X, X, X, X},
 {O, X, O, O, X, X},
 {X, X, X, X, O, X}}

```

Consider 0 based indexing of rows and columns in the grid where the top-left cell of the grid is represented by (0,0) and the bottom-right cell by (5,5).

In the above example, there are three islands. These islands can be represented as the combination of the following cells in the grid:

**Island 1:** Formed by three elements at (1, 4), (2, 3), (2, 4) positions.

**Island 2:** Formed by a single element at (3, 1) position.

**Island 3:** Formed by two elements at (4, 2), (4, 3) positions.

	0	1	2	3	4	5
0	X	X	O	X	X	X
1	X	X	O	X	O	X
2	X	X	X	O	O	X
3	X	O	X	X	X	X
4	O	X	O	O	X	X
5	X	X	X	X	O	X

**Note:** Elements at positions (0, 2) and (1,2) do not form an island as there is no 'X' surrounding it from the top in the grid. Similarly, the element at position (4, 0) does not form an island as there is no 'X' on its left. Also, the element at position (5, 4) does have no X at its bottom.

#### Input Format:

The **first line** of the input contains two space-separated integers, **N** and **M**, representing the total number of rows and columns respectively.

The next **N** lines contain **M** space-separated characters describing rows of the grid **G** (each element of **G** is either 'O' or 'X').

#### Output Format:

The updated grid will be displayed as below:

**N** lines each containing **M** space-separated characters describing the rows of the updated grid **G**.

#### Sample Input:

```
3 4
XXOX
XOXX
XXOO
```

#### Sample Output:

```
XXOX
XXXX
XXOO
```

#### Approach 1: Brute Force Approach

This is the basic naive approach, where we explore all the possible islands.

#### Steps:

1. Create a new 2D array (**arr**) of the same size as the input 2D array (**matrix**).
2. Iterate through the **matrix** and check for the value of the current element:
  - a. If it is equal to '**X**', we'll assign **arr** as '**X**' at the same index.
  - b. Otherwise, check whether there exists a path from the current element to any edge element such that all elements in the path are equal to '**O**'. Here, edge elements are present in either the first row/column or the last row/column.
    - i. If such a path exists, we'll assign **arr** as '**O**' at the same index; otherwise, we'll assign **arr** as '**X**'.
3. Finally, copy all elements of **arr** to the **matrix**.

**Time Complexity:**  $O(N^2 * M^2)$ , where **N** and **M** are the number of rows and columns in the grid respectively. In the worst case, for each entry in the matrix, we will be visiting all paths—possibly in all the four directions, that is, top, left, right and bottom edges of the matrix—from that cell to an edge cell. In all paths, it can only include at most **N \* M** cells. Therefore, the time complexity will be  $O(N^2 * M^2)$ .

**Space Complexity:**  $O(N * M)$  as all the changes are done in the new matrix of **N** rows and **M** columns.

### Approach 2: Flood Fill Algorithm:

In this approach, we follow the flood fill algorithm.

#### Steps:

1. Traverse the given matrix and replace all '**O**' with '**R**' or any other symbol except '**O**' and '**X**'.
2. We look for '**R**' as a character in the cell for each edge in the given matrix.
  - a. For every '**R**' in the current edge, we do the following:
    - i. Visit all adjacent cells/entries (left, right, top, down) recursively.
    - ii. If there is an '**R**', we replace it with '**O**'.
3. Traverse the modified matrix again and replace all the remaining '**R**' with '**X**'.

**Time Complexity:**  $O(N * M)$ , where **N** and **M** are the number of rows and columns in the grid respectively.

**Space Complexity:**  $O(N * M)$ . In the worst case, recursive calls can go to a maximum depth of **N \* M**, thus causing an extra space for the recursive stack.

## 6. Closest Distance Pair [<https://coding.ninja/P101>]

**Problem Statement:** You are given an array containing **N** points in the plane. Your task is to find out the distance of the two closest points given in the array.

#### Note :

The distance between two points  $(x_1, y_1)$  and  $(x_2, y_2)$  is calculated as  $(x_1 - x_2)^2 + (y_1 - y_2)^2$ .

#### Input Format:

The **first line** contains an integer **N** which denotes the number of points in the input array.

The **next N lines** contain two integers separated by a single space. The first integer represents the **x** coordinate and the second integer represents the **y** coordinate of that point on the plane.

#### Output Format:

The only line of the output contains the minimum distance between any two points in the array.

#### Sample Input:

```
5
1 2
2 3
3 4
5 6
2 1
```

#### Sample Output:

```
2
```

### Approach 1: Brute Force Approach

In this naive approach, we calculate the distances between all the possible pair of points given in the array and figure out the minimum as our answer.

#### Steps:

1. For every pair of points  $(x_1, y_1)$  and  $(x_2, y_2)$ , calculate the distance by the following formula:  
$$\text{distance} = (x_1 - x_2)^2 + (y_1 - y_2)^2.$$
2. If the minimum distance till now is greater than **distance**, then update the minimum distance.
3. Return the minimum distance.

**Time Complexity:**  $O(N^2)$ , where **N** denotes the number of input points. Since we are calculating the distance between every pair, therefore the time complexity is  $O(N^2)$ .

**Space Complexity:**  $O(1)$ , as constant extra space is used.

### Approach 2: Divide And Conquer

We can also use the **Divide and Conquer Algorithm** to solve this problem.

### Steps:

1. Sort the points with respect to **x** coordinate and store them in a new array (**Pair**).
2. Find the middle point in the sorted array, we can take **Pair[n/2]** as the middle point.
3. Divide the given array into two halves. The first subarray contains points from **Pair[0]** to **Pair[n/2]**. The second subarray contains points from **Pair[n/2 + 1]** to **Pair[n - 1]**.
4. Recursively find the smallest distances in both the subarrays. Let the distances be **leftDistance** and **rightDistance** respectively. Find the minimum of both these distances (**minDistance**). Our answer is the value in the **minDistance** if both points of closest distance lie in any one half only.
5. Now we need to consider the pairs such that one point in the pair is from the left half and the other is from the right half of the array. We will only consider those pairs of points which have a distance less than **minDistance** as we already have our minimum set to **minDistance**.
6. Consider the vertical line passing through the middle point—that is, **Pair[n/2]**. Find all the points whose **x** coordinate is closer than **minDistance** to this middle vertical line at **Pair[n/2]**. Build an array **Arr** of all such points.
7. Sort this array **Arr** according to **y** coordinates then for each point **pi**, in the array **Arr**, consider all points **pj** that have **y** coordinates less than **y** coordinates of (**pi + minDistance**) point.
8. For each pair, calculate the distance and compare it with the current best distance.
9. Return the **minDistance**.

### Time Complexity:

Let the time complexity of the above algorithm be  $T(n)$ . Let us assume that we use a  $O(n \log n)$  sorting algorithm. The above algorithm divides all points into two sets and recursively calls for two sets. After dividing,

- it finds the array **Arr** in  $O(n)$  time,
- sorts the array **Arr** in  $O(n \log n)$  time, and
- finally finds the closest points in the array **Arr** in  $O(n)$  time.

$$T(n) = 2T(n/2) + O(n) + O(n \log n) + O(n)$$

$$T(n) = O(n^*(\log n)^2)$$

**Space Complexity:  $O(N)$** , as in every recursion stack, it uses a strip array/list to store points that lie nearer to the middle point chosen, making the maximum length of the recursion stack of the order of  $O(N)$ .

## 7. Print Permutations – String [<https://coding.ninja/P102>]

**Problem Statement:** Given an input string **S**, find and return all the possible permutations of the input string.

**Note:**

The input string may contain repetitive characters, so there can also be some repeating permutations. The order of permutations doesn't matter.

**Input Format:**

The only input line consists of a string **S** of alphabets in lower case characters.

**Output Format:**

Print each permutation in a new line.

**Sample Input:**

abc

**Sample Output:**

abc

acb

bac

bca

cab

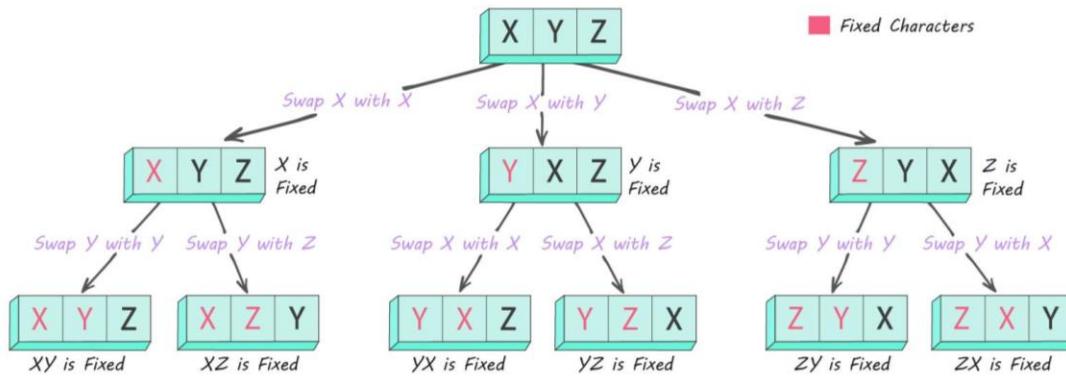
cba

**Backtracking Approach:**

We can find all the permutations of the given string by the backtracking approach.

**Steps:**

1. Fix a character and then swap all the remaining characters of the string with the fixed character.
2. Next, find all the possible permutations for the remaining characters by recursion.
3. The base case for the recursion will be when only one character is left unprocessed in the string.



*Recursion Tree for Permutation of String “XYZ”*

**Time Complexity:**  $O(N * N!)$ , where **N** denotes the length of the string. In total, there will be **N!** permutations of the string,, and for each permutation, it requires  $O(N)$  time to print. Therefore, the overall time complexity is  $O(N * N!)$ .

**Space Complexity:**  $O(N)$ , where **N** denotes the length of the string. In the worst case, extra space is required for the recursion stack.

## 8. Sudoku [<https://coding.ninja/P103>]

**Problem Statement:** You are given a  $9 \times 9$  2D matrix with some cells filled with digits from 1–9 and some empty cells denoted by 0. You need to find whether there exists a way to fill all the empty cells with the digits from 1–9 such that the final matrix is a valid Sudoku solution.

A Sudoku solution must satisfy all the following conditions:

1. Each of the digits from 1–9 must occur exactly once in each row.
2. Each of the digits from 1–9 must occur exactly once in each column.
3. Each of the digits from 1–9 must occur exactly once in each of the nine  $3 \times 3$  sub-matrices of the matrix.

### Note:

1. There will always be at least one cell in the matrix which is empty.
2. The given initial matrix will always be consistent according to the rules mentioned in the problem statement.

### Input Format:

The **first line** contains a single integer **T**, representing the number of test cases.

The **next  $9 * T$  lines** contain the description of each test case.

There will be 9 lines for each test case,, with each line containing 9 space-separated digits (0 if the cell is empty, digits 1–9 otherwise).

### Output Format:

Print **yes** if there is a way, otherwise print **no**.

**Sample Input:**

```
1
9 0 0 0 2 0 7 5 0
6 0 0 0 5 0 0 4 0
0 2 0 4 0 0 0 1 0
2 0 8 0 0 0 0 0 0
0 7 0 5 0 9 0 6 0
0 0 0 0 0 4 0 1
0 1 0 0 0 5 0 8 0
0 9 0 0 7 0 0 0 4
0 8 2 0 4 0 0 0 6
```

**Sample Output:**

yes

**Explanation:**

One of the possible solutions is:

```
9 4 1 3 2 6 7 5 8
6 3 7 1 5 8 2 4 9
8 2 5 4 9 7 6 1 3
2 6 8 7 1 4 3 9 5
1 7 4 5 3 9 8 6 2
3 5 9 6 8 2 4 7 1
4 1 3 2 6 5 9 8 7
5 9 6 8 7 3 1 2 4
7 8 2 9 4 1 5 3 6
```

**Approach 1: Brute Force**

In this approach, we will find out all the possible arrangements of digits in the given sudoku and then figure out if any one of them solves the sudoku.

**Steps:**

1. For all the empty cells of the matrix, try all the arrangements of the digits from(1–9)—that is, filling all the empty cells with some digit.
2. For each arrangement, check whether the matrix now violates any of the given rules; if it does, then check for the next arrangement; otherwise, we found a valid sudoku solution.
3. If,, after checking all the arrangements, we are unable to find a solution, then there doesn't exist any solution to the given input sudoku.

**Time Complexity:  $O(9^K)$ ,** where  $K$  denotes the total number of empty cells in the given matrix.

The time complexity is exponential as for each cell, we are trying all the digits from 1 to 9.

**Space Complexity:**  $O(K)$ , where  $K$  is the number of empty cells. We are checking the original matrix itself, and the recursion stack can grow in the order of  $K$ , which can take the max value of  $9 * 9 = 81$ .

### Approach 2: Backtracking Approach

In this approach, instead of trying all the digits from (1 to 9) in the empty cells, we try to find a solution using only those digits which don't violate any of our conditions to be a valid sudoku—when placed in that given cell.

#### Steps:

1. Before assigning a number to an empty cell, check if it satisfies all the rules. If it does, then assign this number to the cell and then recursively check if it leads to a successful filling of all the other empty cells or not. If it doesn't, try some other valid number for the current cell and then repeat the recursion for all other empty cells.
2. If at some stage we can't find a number that is valid for the current empty cell, we **backtrack**—that is, move one step behind and check for some other possible digits for the previously filled cell.
3. The backtracking might continue until we get a valid configuration of the matrix or if it has tried all the possibilities. If the latter happens, then there is no solution for the given input matrix to be a valid sudoku solution.

**Time Complexity:**  $O(9^K)$ , where  $K$  denotes the total number of empty cells in the given matrix. The complexity might look the same, however, given that there will be some pre-filled entries, it will be much better than the sheer brute force approach discussed previously.

**Space Complexity:**  $O(K)$ , where  $K$  is the number of empty cells. We are checking the original matrix itself,, and the recursion stack can grow in the order of  $K$ , which can take max value as  $9 * 9 = 81$ .

## 9. IP Address [<https://coding.ninja/P104>]

**Problem Statement:** You are given a string  $S$  containing only digits. Your task is to find all the possible valid IP addresses that can be obtained from the string  $S$  in lexicographical order.

#### Note:

A valid IP address consists of exactly four integers. Each integer lies between 0 and 255,; hence both inclusive separated by single dots, and cannot have leading zeros except in the case of zero itself.

For example:

The IP addresses 0.1.24.255 and 18.5.244.1 are valid, and 0.01.24.255 (as 01 contains one leading zero) and 18.312.244.1 (as 312 does not lie between 0 and 255) are invalid IP addresses.

### **Input Format :**

The first line of input contains a single integer **T**, representing the number of test cases or queries to be run.

Then the **T** test cases follow.

The only line of each test case contains the input string **S**.

### **Output Format:**

For each test case, print a single line containing all possible valid IP addresses that can be obtained from **S** in lexicographical order in a separate line.

Each IP address obtained from the string **S** is written within quotes (""), and separated by comma (,) and a space. All the IP addresses of the given string **S** are written inside square brackets [].

### **Sample Input:**

```
1
23579
```

### **Sample Output:**

```
["2.3.5.79", "2.3.57.9", "2.35.7.9", "23.5.7.9"]
```

### **Explanation:**

All possible valid IP addresses are shown below.

```
2.3.5.79 , 2.3.57.9 , 2.35.7.9 , 23.5.7.9
```

### **Approach 1: Brute force**

The idea is to divide the given string into four parts using 3 dots, where each part corresponds to a number. We will then add the current IP address to our **answer** array if it satisfies the following conditions:

1. Each number lies between 0 and 255 (both numbers inclusive).
2. Each number must not contain any leading zeros apart from zero itself.

### **Steps:**

1. Create a function, let's say **check**, which takes an argument **s** as an input string and is used to check the validity of a number in the current IP address. It will return **true** if **s** lies between 0 and 255 and **s** does not contain leading zeroes.
2. Create an empty list of strings, let's say **answer**, to store all possible valid IP addresses.
3. Denote the length of the given input string using **n**.
4. Use three nested loops to partition the string into four parts and check if all the parts are valid.
5. Run a loop from **i = 0 to min(3, n - 4)**:
  - a. Run a loop from **j = i + 1 to min(i + 3, n - 3)**:

- i. Run a loop from  $k = j + 1$  to  $\min(j + 3, n - 2)$  and perform the following steps:
  1. Denote substring from  $x$  to  $y$  as **substr(x, y)**.
  2. Check if **check(substr(0, i))**, **check(substr(i + 1, j))**, **check(substr(j + 1, k))**, and **check(substr(k + 1, n - 1))** are all true. If yes, add this partitioned string as a valid IP address to the **answer** list.
  3. Else move to the next combination of partitions.
6. Repeat step 5 until you check for all the partitions.

**Time Complexity: O(1).** In the worst case, we are using three ‘for’ loops which takes constant time, and the check function also takes constant time. Hence, the overall time complexity will be **O(1)**.

**Space Complexity: O(1).** In the worst case, constant space is required.

### Approach 2: Backtracking Approach

This problem can also be solved using a backtracking approach. The idea here is to use the backtracking technique to generate all the possible valid IP addresses. The key observation here is that we can select either 1 or 2 or 3 digits at a time and put it into one segment.

Therefore before moving to the next segment, in each step, we will choose 1 or 2 or 3 digits and we will check if the current segment is valid or not. If it is invalid then there is no need to visit this path and other combinations of the segment are tried.

#### Steps:

1. Create a function, let's say **check** which takes an argument **s** as a string that is used to check the validity of a number in an IP address. It will return **true** if **s** lies between 0 and 255 and **s** does not contain any leading zeros.
2. Create an empty list of strings, let's say an **answer** that is used to store all the possible valid IP addresses.
3. Denote the length of the string using **n**.
4. Create a function, let's say **backtracking**, which will take five arguments **answer**, **s** (denoting original string), **curlIndex** (denoting current index), **segments** (used to store segments) and **segmentIndex**, (denoting the number of segments formed so far).
5. We will pass **curlIndex = 0** and **segmentIndex = 0** as arguments in the backtracking function.
6. Call **backtracking** function to get the answer.
7. Define base cases as:
  - a. If **curlIndex = n**:
    - i. If **segmentIndex = 4**, then add the current path to **answer**, and return.
  - b. If **curlIndex < n**:
    - i. If **segmentIndex > 4**, break the recursion.

8. Run a loop from **steps** = 0 to 2 and perform the following:
  - a. Add **s[curIndex + steps]** to **segment[segmentIndex]**.
  - b. If **curIndex < n** and **segmentIndex > 4**, then break the loop.
  - c. If the current segment is valid then we will recur for the next segment and increment both **curIndex** and **segmentIndex** by one.
  - d. Else we will move to the next iteration.

**Time Complexity: O(1).** In the worst case, the total number of IP addresses is fixed. Thus our function will remain constant after some particular value of the input. Hence, the time complexity will be **O(1)**.

**Space Complexity: O(1).** In the worst case the total number of IP addresses are fixed, hence space complexity will also remain constant after some particular value of the input. Therefore, the space complexity will be **O(1)**.



# 12. Dynamic Programming and Memoization

---

## What is Dynamic Programming?

Dynamic Programming is a programming paradigm, where the idea is to store the already computed values instead of calculating them again and again in the recursive calls. This optimization can reduce our run time significantly and sometimes reduce it from exponential-time complexities to polynomial time complexities.

Let us understand further with the help of the following example:

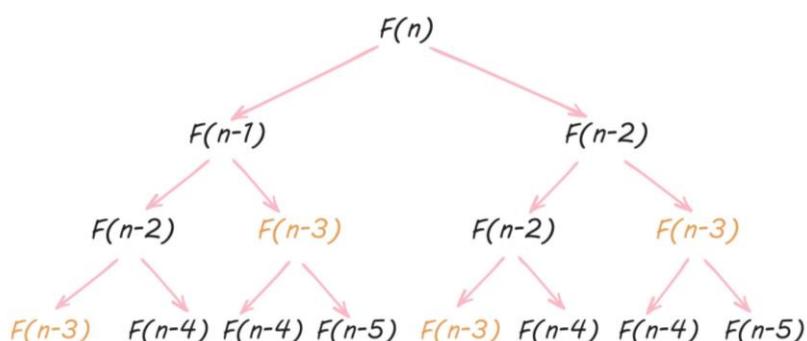
We know that Fibonacci numbers are calculated as follows:

$$\begin{aligned}\mathbf{fibo(n)} &= n && \text{(for } n = 0 \text{ and } n = 1\text{)} \\ \mathbf{fibo(n)} &= \mathbf{fibo(n - 1)} + \mathbf{fibo(n - 2)} && \text{(for } n > 1\text{)}\end{aligned}$$

The pseudo code to find the  $n^{\text{th}}$  Fibonacci number using recursion is as follows:

```
function fibo(n):
    if(n <= 1):
        return n
    return fibo(n - 1) + fibo(n - 2)
```

Here, for every  $n$ , we need to make a recursive call to  $\text{fibo}(n - 1)$  and  $\text{fibo}(n - 2)$ . For  $\text{fibo}(n - 1)$ , we will again make recursive calls to  $\text{fibo}(n - 2)$  and  $\text{fibo}(n - 3)$ . Similarly, for  $\text{fibo}(n - 2)$ , recursive calls are made to  $\text{fibo}(n - 3)$  and  $\text{fibo}(n - 4)$  and so on until we reach the base case.



**Time Complexity:**

At every recursive call, we are doing constant work (**K**), which is adding previous outputs to obtain the current result). Since reaching 1 from **n** will take **n** calls, therefore, at the last level, we are doing  $2^{n-1} * K$  work. Thus the total work can be calculated as:

$$(2^0 + 2^1 + 2^2 + \dots + 2^{n-1}) * K \approx 2^n K$$

Thus till any level, we are doing  $2^n * K$  work (where  $n = 0, 1, 2, \dots$ ). Hence, it means time complexity will be **O( $2^n$ )**—that is, exponential.

We can observe that there are repeating recursive calls made in the program, which increase the time complexity of the entire program as shown in the above figure. To overcome this problem, we store the output of previously encountered values—preferably in arrays as they are the most efficient way to traverse and extract data. By storing these values, we can directly use them in our calculations instead of calculating them repeatedly while making recursive calls, thereby improving the run time of our code.

## Memoization

The process of storing each recursive call's output and then using it directly for any further calculations is called Memoization.

- Initialise an **answer** array to -1. Now while making a recursive call, we will first check if the value stored in this **answer** array corresponding to that position is -1 or not. If it is -1, it means we haven't calculated the value yet, and thus further processing needs to be done by making appropriate recursive calls for that value.
- After obtaining the output, we need to store this in the **answer** array so that next time, if the same value is encountered, it can be directly used from this **answer** array.

Now in this process of memoization, considering the above Fibonacci numbers example, it can be observed that the total number of unique calls will be at most (**n + 1**), thus making the run time complexity reduced to linear—that is, **O(N)**.

Let's look at the memoization code for Fibonacci numbers example:

```
function fibo(n, dp)
    // base case
    if n equals 0 or n equals 1
        return n
    // checking if the value has been already calculated
    if dp[n] not equals -1
        return dp[n]
    // final ans
    myAns = fibo(n - 1, dp) + fibo(n - 2, dp)
```

```
dp[n] = myAns
```

```
return myAns
```

## Top-down approach

- Memoization is a **top-down approach**, where we save the answers to the recursive calls calculated so far so that they can be used to calculate future answers when the same recursive calls are to be evaluated again, thereby improving the run time complexity to a much greater extent.
- Recursive calls terminate over the base cases, which means we are already aware of the answers that should be stored in the base case's index.
- Finally, by making a recursive call once for every index and using the previously stored values in the answer array, the value for every index of the array is calculated .
- In case of Fibonacci numbers, these indexes are 0 and 1 as **fibo(0) = 0** and **fibo(1) = 1**. So we can use these two values as our base cases and evaluate **fibo(2) (= fibo(1) + fibo(0))**, and so on for every other index.

## Bottom-up approach

- In the top-down approach, we were trying to figure out the dependency of the current value on the previously stored values in the array and then use them to calculate our new values.
- In the bottom-up approach, we look for those values which do not depend on other values for their evaluation, which means they are independent (the base case's values as these are the smallest problems about which we are already aware of).

Let us now look at the Bottom Up approach code for calculating the  $n^{\text{th}}$  Fibonacci number:

```
function fibonacci(n):
    f = array[n+1]
    // base case
    f[0] = 0
    f[1] = 1

    for i from 2 to n:
        // calculating the f[i] based on the last two values
        f[i] = f[i-1] + f[i-2]

    return f[n]
```

**Note:** Generally, **memoization** is a **recursive** approach, and **DP** is an **iterative** approach.

### Basic differences between the tabulation and memoization approach

Memoization	Tabulation
It is a Top-Down DP approach.	It is a Bottom-Up DP approach.
In the memoization approach for DP, the lookup table is not filled in any specific order, instead the values are filled on demand.	Starting from the first entry in the lookup table, all the remaining entries are filled in a specific order.
It makes use of the recursive approach.	It is an iterative approach to DP.

### General Steps to approach a problem:

1. Figure out the most straightforward approach for solving the problem using brute-force recursion.
2. Now, if the problem has some overlapping subproblems, then try to optimise the recursive approach by storing the previous answers using memoization.
3. Finally, replace recursion by iteration using dynamic programming (Bottom-Up approach).

## Sample Problems

### Min Cost Path

**Problem Statement:** Given an integer matrix of size **m\*n**, you need to determine the minimum cost to reach the cell **(m - 1, n - 1)** from the cell **(0, 0)**.

**Note:** From a cell **(i, j)**, you can move in three directions: **(i + 1, j)**, **(i, j + 1)**, and **(i + 1, j + 1)**. The cost of a path is defined as the sum of values of each cell through which the path passes.

For example, if the given input is as follows:

```
3 4  
3 4 1 2  
2 1 8 9  
4 7 8 1
```

The path that should be followed for achieving minimum cost is **3 → 1 → 8 → 1**. Hence the output should be **13**.

## Approach 1: Recursive Brute Force Solution

Using the **recursive approach** to reach from the cell **(0, 0)** to **(m - 1, n - 1)**, we need to decide the next direction from the three available valid directions for every current cell. We call recursion over all the three choices available to us, and finally, we will be considering the one with minimum cost and add the current cell's value to it.

Let's now look at the recursive code for this problem:

```
function minCost(cost, m, n)
    // base case
    If m == 0 AND n == 0
        return cost[m][n]

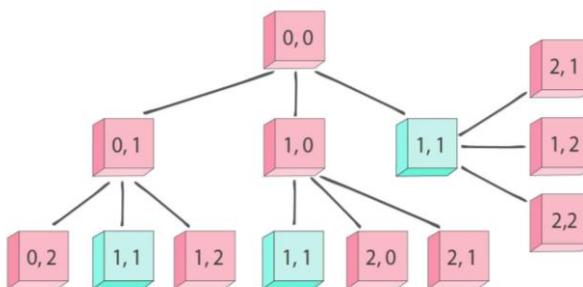
    // outside the grid
    if m < 0 or n < 0
        return infinity

    recursionResult1 = minCost(cost, m - 1, n)
    recursionResult1 = minCost(cost, m, n - 1)
    recursionResult1 = minCost(cost, m - 1, n - 1)
    myResult = min(recursionResult1, recursionResult2, recursionResult3) + cost[m][n]

    return myResult
```

The recursion tree when **m = 4 and n = 5**; would look something like as shown :

$m = 4, n = 5$



Here, we can see that there are many repeated/overlapping recursive calls leading to exponential time complexity—that is, **O(3<sup>n</sup>)** since every recursive call leads to three more recursive calls.

Now, to optimise the above recursive approach, let's move on to the **Memoization approach**.

## Approach 2: Memoization Approach

In memoization, we avoid repeated overlapping recursive calls by storing the output of each recursive call in an array. In this case, we will be using a 2D array instead of 1D, as the storage used for the memoization depends on the **states**, which are basically the necessary variables whose values at a particular instant are required to calculate the desired result.

Following is the pseudo code for the memoization approach:

```
function minCost(cost, m, n, dp)
    // base case
    If m == 0 AND n == 0
        return cost[m][n]

    // outside the grid
    if m < 0 or n < 0
        return infinity

    if dp[m][n] != -1
        return dp[m][n]

    recursionResult1 = minCost(cost, m - 1, n , dp)
    recursionResult1 = minCost(cost, m, n - 1 , dp)
    recursionResult1 = minCost(cost, m - 1, n - 1 , dp)
    myResult = min(recursionResult1, recursionResult2, recursionResult3) + cost[m][n]

    // store in dp array
    dp[m][n] = myresult

    return myResult
```

## Approach 3: Bottom-Up Iterative approach

We will now proceed towards the **DP approach (Bottom-Up Approach)**.

1. The DP approach is simple. We just need to create a 2D solution array (**ans**), where:

**ans[i][j]** = minimum cost to reach from **(i, j)** to **(m - 1, n - 1)**

2. Now we initialize the last row and last column of the matrix (**ans**) with the sum of their values and the value just after it. This is because, in the last row or column, we can reach the destination cell (**m - 1, n - 1**) by traversing through the forward cell only:

**ans[m - 1][n - 1] = cost[m - 1][n - 1]**

**ans[m - 1][j] = ans[m - 1][j + 1] + cost[m - 1][j] (for 0 < j < n - 1)**

**ans[i][n - 1] = ans[i + 1][n - 1] + cost[i][n - 1] (for 0 < i < m - 1)**

3. Next, we fill the rest of the answer matrix (**ans**) by checking out the minimum among the values from where we could reach the current cell. For this, we will use the same formula as used in the recursive approach:

**ans[i][j] = min(ans[i + 1][j], ans[i + 1][j + 1], ans[i][j + 1]) + cost[i][j]**

Finally, we will get our answer at the cell (0, 0), which we will return.

The pseudo code for the above discussed DP approach is as follows:

```

funcion minCost(cost, m, n)
    ans = array[m][n]
    ans[m - 1][n - 1] = cost[m - 1][n - 1]

    // Initialize last column of ans array
    for i from m - 2 to 0
        ans[i][n - 1] = ans[i + 1][n - 1] + cost[i][n - 1]

    // Initialize last row of ans array
    for j from n - 2 to 0
        ans[m - 1][j] = ans[m - 1][j + 1] + cost[m - 1][j]

    // Construct rest of the ans array
    for i from m - 2 to 0
        for j from n - 2 to 0
            min_temp = min(ans[i + 1][j + 1], ans[i + 1][j], ans[i][j + 1])
            ans[i][j] = min_temp + cost[i][j]

    return ans[0][0]

```

**Time Complexity:** Here, we can observe that as we move from the **cell ( $m - 1, n - 1$ ) to cell ( $0,0$ )**, the  $i^{\text{th}}$  row varies from  $m - 1$  to 0, and the  $j^{\text{th}}$  column runs from  $n - 1$  to 0. Hence, the unique recursive calls will be a maximum of  $(m - 1) * (n - 1)$ , which leads to the time complexity of  $O(m*n)$ .

**Space Complexity:** Since we use an array of size  $(m*n)$  to store the results for recursive calls, the space complexity turns out to be  $O(m*n)$ .

## **LCS (Longest Common Subsequence):**

**Problem statement:** A longest common subsequence (LCS) is defined as the longest subsequence that is common to all the given sequences. You are given two input strings **s1** and **s2**; you are supposed to calculate the length of the longest common subsequence present between the given two input strings.

**Note:** A subsequence is a part of a string that can be made by omitting none or some of the characters from the string while maintaining the order of the rest of the characters in the string.

If  $s_1$  and  $s_2$  are two given strings, then  $z$  is the common subsequence of  $s_1$  and  $s_2$ , if  $z$  is a subsequence of both of them.

### **Example 1:**

$s_1 = \text{"abcdef"}$

$s_2 = \text{"xyczeff"}$

Here, in the example, the longest common subsequence is '**cef**'; hence the answer is **3** (the length of the LCS).

### **Approach: Brute Force Recursive approach**

Let's first think of a brute-force approach using simple **recursion**.

For LCS, we have to match the starting characters of both the strings. If they match, then simply we can break the problem as shown below:

$s_1 = \text{"x|yzar"}$

$s_2 = \text{"x|qwea"}$

But, if the first characters do not match, then we have to figure out that by traversing which of the following strings, we will get our answer. This is done by traversing over both of them one by one and checking for the maximum value of LCS obtained among them.

For example, suppose string  $s = "xyz"$  and string  $t = "zxay"$ .

We can see that their first characters do not match; therefore, we call recursion over them through either of the following ways:

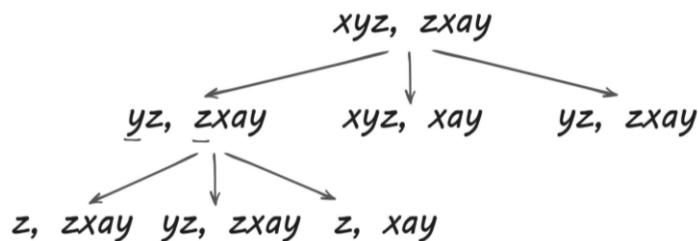
a.  $S \rightarrow \boxed{x}yz$       Skipping the first character from string S  
 $T \rightarrow \boxed{z}xay$

b.  $S \rightarrow \boxed{x}\boxed{yz}$       Skipping the first character from string T  
 $T \rightarrow z\boxed{xay}$

c.  $S \rightarrow x\boxed{\boxed{yz}}$       Skipping the first character from both the strings  
 $T \rightarrow z\boxed{xay}$

Finally, our answer would be **LCS = max(A, B, C)**.

If we dry run this over the example:  $s = "xyz"$  and  $t = "zxay"$ , it will look something like below:



Here, as for each node, we will be making three recursive calls, but the third recursive call will be covered by the first two, so the time complexity will be exponential and is represented by  $O(2^{m+n})$ , where  $m$  and  $n$  are the lengths of both strings.

**Pseudocode:**

```

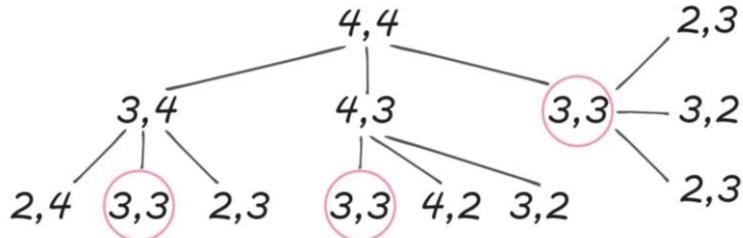
function lcs(s, t, m, n):
    // Base Case
    if m equals 0 or n equals 0
        return 0

    // match (m - 1)th character of s with (n - 1)th character of t
    if s[m - 1] equals t[n - 1]
        return 1 + lcs(s, t, m - 1, n - 1)

    else:
        return max(lcs(s, t, m, n - 1), lcs(s, t, m - 1, n))
  
```

Consider the diagram shown, where the input strings are of lengths four each:

$$\begin{aligned} S &= abcd \longrightarrow 4 \\ T &= xyzw \longrightarrow 4 \end{aligned}$$



As we can see, there are multiple overlapping recursive calls, the solution can be optimised using **memoization** followed by DP.

### Approach: Dynamic Programming

For string **s**, we can make at most **length(s)** recursive calls, and similarly, for string **t**, we can make at most **length(t)** recursive calls, which are also dependent on each other's solution.

Hence, our result can be directly stored in the form of a 2-dimensional array of size **(length(s)+1) \* (length(t) + 1)**.

So for every index *i* in string **s** and *j* in string **t**, we will choose one of the following two options:

1. If the character **s[i]** matches the character **t[j]**, the length of the common subsequence would be one plus the length of the common subsequence till the **(i - 1)<sup>th</sup>** and **(j - 1)<sup>th</sup>** indices in the two respective strings.
2. If the character **s[i]** does not match the character **t[j]**, we will take the longest subsequence by either skipping **i<sup>th</sup> or j<sup>th</sup> character** from the respective strings.

Hence, we will get the final answer at the position **matrix[length(s)][length(t)]**.

The pseudo code for the discussed Top-Down approach is as follows:

```
function LCS(s, t, i, j, memo)
    // one or both of the strings are fully traversed
    if i equals len(s) or j equals len(t)
        return 0

    // if result for the current pair is already present in the table
    if memo[i][j] not equals -1
        return memo[i][j]

    memo[i][j] = max(
        LCS(s, t, i+1, j, memo),
        LCS(s, t, i, j+1, memo),
        LCS(s, t, i+1, j+1, memo) + 1
    )
```

```

if s[i] equals t[j]
    // check for the next characters in both the strings
    memo[i][j] = lcs(s, t, i + 1, j + 1, memo) + 1

else
    // check for the maximum answer using the two options
    memo[i][j] = max(lcs(s, t, i, j + 1, memo), lcs(s, t, i + 1, j, memo))

return memo[i][j]

```

The pseudo-code for the discussed Bottom-Up approach is as follows:

```

function LCS(s , t)

    // find the length of the strings
    m = len(s)
    n = len(t)

    // creating an array to store the dp values
    L = array[m + 1][n + 1]

    for i from 0 to m
        for j from 0 to n
            if i equals 0 or j equals 0
                L[i][j] = 0
            else if s[i-1] equals t[j - 1]
                L[i][j] = L[i - 1][j - 1]+1
            else:
                L[i][j] = max(L[i - 1][j] , L[i][j - 1])

    // L[m][n] contains the length of LCS of strings s and t
    return L[m][n]

```

**Time Complexity:** We can see that the time complexity of the DP and memoization approach is reduced to **O(m\*n)** where **m** and **n** are the lengths of the given strings.

**Space Complexity:** Since we are using an array of size **(m\*n)**, the space complexity turns out to be **O(m\*n)**.

## Applications of Dynamic Programming

- They are often used in machine learning algorithms, for example, Markov decision process in reinforcement learning.
  - They are used for applications in interval scheduling.
  - They are also used in various algorithmic problems and graph algorithms like Floyd warshall's algorithms for the shortest path and the sum of nodes in a subtree.
- 

## Practice Problems

### 1. Count Ways to Reach Nth Stair : [<https://coding.ninja/P105>]

**Problem Statement:** You have been given several stairs. Initially, you are at the 0<sup>th</sup> stair, and you need to reach the N<sup>th</sup> stair. Each time you can either climb one step or two steps. You are supposed to return the total number of distinct ways to climb from the 0<sup>th</sup> step to the N<sup>th</sup> step.

#### Input Format:

The **first line** contains an integer **T**, which denotes the number of test cases or queries to be run. Then the test cases follow.

The **first and the only argument** of each test case contains an integer **N**, representing the number of stairs.

#### Output Format:

For each test case/query, print the total number of distinct ways to reach the topmost stair **N**. Since the number can be huge, so return output modulo 10<sup>9</sup>+7.

#### Sample Input:

1  
3

#### Sample Output:

3

#### Explanation:



We can climb one step at a time—that is, {(0 → 1),(1 → 2), (2 → 3)} or we could climb first, two steps and then one step—that is, {(0 → 2),(2 → 3)} or

we could climb first one step and then two steps—that is,  $\{(0 \rightarrow 1), (1 \rightarrow 3)\}$ .

Therefore the total number of distinct ways to reach the third stair is **3**.

### Approach 1: Brute Force Recursive Approach

One basic approach is to explore all the possible ways to reach the topmost stair. At every step, we have two options—that is, we could climb either one step or two steps at a time from the current stair. So at every step, we explore both the options to climb the next stair to get all the possible ways.

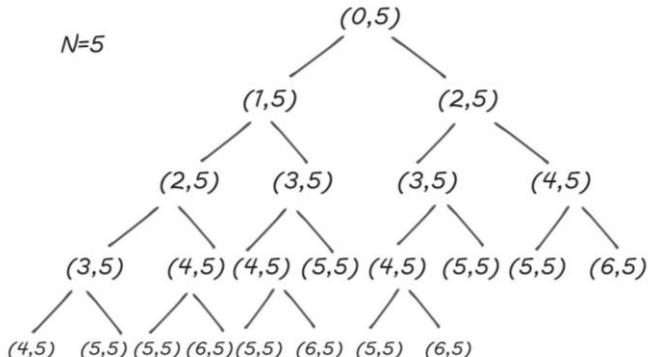
#### Steps:

1. Create a function, say **countDistinctWayToClimbStair**, and recurse over all the possible scenarios.
2. The total number of ways to reach the topmost stair would be the sum of the possible options to climb one or two steps at a time—that is,

**countDistinctWayToClimbStair (currStep, N)**

= **countDistinctWayToClimbStair (currStep + 1, N) + countDistinctWayToClimbStair (currStep + 2, N)**,

where **currStep** denotes the current step on which the person is standing, and **N** denotes the destination step.



*Number of Nodes =  $O(2^n)$*

**Time Complexity:**  $O(2^N)$ , where **N** denotes the total number of stairs. Since every recursive call boils down to further two recursive calls, thereby making the time complexity of the order of  $O(2^N)$ .

**Space Complexity:**  $O(N)$ , where **N** denotes the total number of stairs. In the worst-case scenario, when we climb one step each time, then the depth of the recursion tree will be **N**. Therefore, the space complexity will be **O(N)**.

### Approach 2: Using Memoization approach :

In the previous approach, we were naively calculating the results for every step. Hence, there were lots of redundant/repetitive function calls. If we look at the recursion tree, we notice there can only be at most **N** distinct function calls.

So in order to avoid these repetitive function calls, we store the result at each step and directly return the result for that step whenever a function is called again for that step.

#### Steps:

1. Create a global array, **dp**, of size **N + 1**.
2. Initialise all the elements in **dp** with -1.
3. Let **dp[currStep]** be defined as the total number of ways to reach **currStep** from the 0<sup>th</sup> step.
4. Using recursion, we will find the answer for (**currStep - 1**) and (**currStep - 2**).
5. Each time we recurse, we will check if **dp[currStep]** is -1 or not:
  - a. If it is -1, this means we have not yet calculated the answer for this **currStep** index; hence we will calculate it by calling recursion over that step.
  - b. Else, return **dp[currStep]**, because the answer for this **currStep** has already been calculated and stored in the **dp** array.
6. Finally, store the calculated answer for **currStep** inside **dp[currStep]**.

**Time Complexity:** **O(N)**, where **N** denotes the total number of stairs. Since there will be only **N** distinct function calls as we are saving the result at every step so there will be no redundant/repetitive function calls. Hence the overall time complexity will be **O(N)**.

**Space Complexity:** **O(N)**, where **N** denotes the total number of stairs. We store the result of every step in the **dp** array of size **N**. Hence the space complexity is **O(N)**.

#### Approach 3: Using Dynamic Programming:

In the Dynamic programming approach, instead of storing the result through recursion in an array, we are going to store the result iteratively in an array. Our intuition here is explained as below:

The total number of ways to reach the **currStep** by taking one step or two steps at a time:

- we can take the one-step from (**currStep - 1**)<sup>th</sup> step or
- we can take the two steps from (**currStep - 2**)<sup>th</sup> step.

Thus, the total number of ways to reach **currStep** will be the sum of the total number of ways to reach at the (**currStep - 1**)<sup>th</sup> step and the total number of ways to reach the (**currStep - 2**)<sup>th</sup> step.

#### Steps:

1. Create a **dp** array of size **N + 1**.
2. Let **dp[currStep]** be defined as the total number of ways to reach the **currStep** from the 0<sup>th</sup> step. Hence, **dp[currStep] = dp[currStep - 1] + dp[currStep - 2]**.

3. For the initial case, we need to define **dp[0]** and **dp[1]** ourselves. The number of distinct ways to climb for both the cases when **currStep = 0** and **currStep = 1** will be **1**. Thus, **dp[0] = 1** and **dp[1] = 1**.
4. After iterating through the loop, the final answer will be stored in **dp[N]**.

**Time Complexity:** **O(N)**, where **N** denotes the total number of stairs. We are traversing in the **dp** array only once. Therefore, time complexity will be **O(N)**.

**Space Complexity:** **O(N)**, where **N** denotes the total number of stairs. Since we are storing the result into an array of size **N + 1**, the space complexity will be **O(N)**.

## **2. Palindrome Partitioning :** [\[https://coding.ninja/P106\]](https://coding.ninja/P106)

**Problem Statement:** Given a string **str**. Find the minimum number of partitions to make in the string such that every partition of the string is at least of length one and is also a palindrome.

Consider “AACCB” we can make a valid partition like A | A | CC | B. Among all such valid partitions, return the minimum number of cuts to be made such that all the resulting substrings in the partitions are palindromes.

The minimum number of cuts for the above example would be two cuts as suggested below:

**AA | CC | B**

### **Input Format:**

The first line of input contains a single integer **T** denoting the number of test cases.

The next **T** lines represent the **T** test cases.

Each test case on a separate line contains a string **str** denoting the string to be partitioned.

### **Output Format:**

For each test case, return the minimum number of cuts to be made so that each partitioned substring is a palindrome.

### **Sample Input:**

1

APPLE

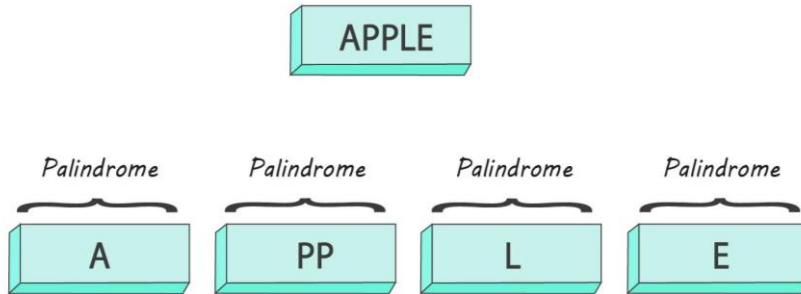
### **Sample Output:**

3

### **Explanation:**

Take the example of APPLE as shown below:

We can clearly see that we need at least 3 cuts to partition the string into palindromic substrings.



### Approach 1: Brute Force Approach

This is a recursive approach. We can break the problem into a set of related subproblems of smaller size, which partitions the given string in such a way that it yields the lowest possible number of cuts.

#### Steps:

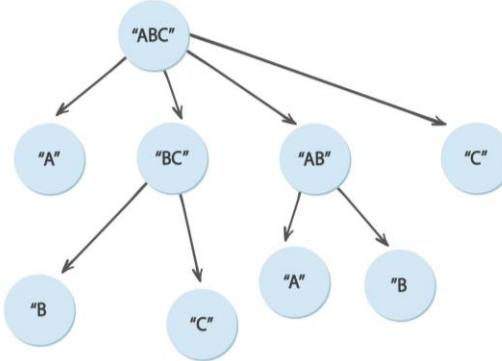
- In each recursive function call, we divide the string into two substrings of all possible sizes.
- Let  $i, j$  be the starting and ending indices of a substring, respectively.
- If  $i$  is equal to  $j$  or  $\text{str}[i...j]$  is a palindrome, we return 0, indicating no cut is required to be made.
- Otherwise, we start a loop with variable  $k$  from the starting index of string  $i$  till ending index of string  $j - 1$  and then recursively call the function for the substring with starting index  $i$  and ending index  $k$  to find the minimum cuts in each substring.
- Do this for all possible positions where we can cut the string and take the minimum over all of them.
- In the end, the recursive function would return the minimum number of partitions needed for the complete string.

**Time Complexity:**  $O(N * 2^N)$ , where  $N$  denotes the length of the string. In the worst case, all possible combinations for length  $N$  are recursively found, and in each case, we check if it is a palindrome that takes order of  $N$  time. Therefore the overall time complexity is  $O(N * 2^N)$

**Space Complexity:**  $O(N)$ , where  $N$  denotes the length of the string. The recursive stack uses space of the order  $N$  in the worst case.

### Approach 2: Dynamic Programming Approach

Following is the recursion tree for the string "ABC"



We can observe that the problem has an optimal substructure (asking for a minimum number of cuts) and has overlapping subproblems. Thus, it can be solved using the dynamic programming approach. The idea here is to store the results of the subproblems so that we do not have to recompute them when they are needed again.

The below approach computes two two-dimensional arrays **isPalindrome** and **cuts**.

- **isPalindrome[i][j]** stores if a substring with starting index  $i$  and ending index  $j$  is a palindrome or not. (**isPalindrome[i][i]** is true as every string of length one is a palindrome)
- **cuts[i][j]** stores the minimum number of cuts needed for a substring with starting index  $i$  and ending index  $j$  to have all its partitions to be as palindromes.

#### Steps:

- Run a loop where  $2 \leq L \leq N$ . Consider  $L$  as the length of the substring.
- For each substring of length  $L$ , set different possible starting indices  $i$ , where  $i$  ranges from 0 to  $L - 1$ .
- Calculate **cuts[i][j]** where  $j = i + L - 1$ —  
This represents the minimum cuts needed for the string **str[ i...j ]** to have all its partitions to be as palindromes, where last index is  $j$ , starting index is  $i$  and the length of the string is  $L$
- If  $L$  is equal to 2, we just need to compare two characters. Else we need to check for the two corner characters at indices  $i$  and  $j$  and the value of **isPalindrome[i + 1][j - 1]**.
- If **str[ i....j ]** is palindrome then **cuts[i][j] = 0**.
- Otherwise, we take a variable  $k$  where  $i \leq k \leq j$  and make a cut at every  $k^{\text{th}}$  location to find the number of cuts. We repeat this at every possible location starting from  $i$  to  $j$  and get the minimum number of cuts and store it at **cuts[i][j]**.
- Lastly, return **cuts[0][n - 1]** which stores the minimum number of cuts needed.

**Time Complexity:**  $O(N^3)$ , where  $N$  denotes the length of the string. We have an order of  $N^2$  loop for each substring, thereby making the overall complexity to be of the order of  $O(N^3)$ .

**Space complexity:**  $O(N^2)$ , where  $N$  denotes the length of the string as we are making an auxiliary two-dimensional array of size  $N * N$ .

### **Approach 3: Highly-Optimised Dynamic Programming Approach**

In the previous approach, we calculated the minimum number of cuts while finding all the palindromic substrings. If we find all the palindromic substrings first and then calculate the minimum cuts, then the solution can be further optimized.

We can do that in the following way:

- Compute a two-dimensional array **isPalindrome[][]** and a single-dimensional array **cuts[]** where:
  - **isPalindrome[i][j]** stores if a substring with starting index *i* and ending index *j* is a palindrome or not and,
  - **cuts[i]** stores the minimum number of cuts needed for a palindrome partitioning of substring str[0...i].
- Mark **isPalindrome[i][i]** as true as every substring of length 1 is a palindrome. Now, find all the palindromic substrings in the given string for each length 'L' and for every starting index *i* in the following way:
  - If the value of 'L' is 2, just compare the two characters.
  - Otherwise, check the first and last character of the substring and also if **isPalindrome[i + 1][j - 1]** is true. If yes, we mark the current substring as a palindrome.

Now that we know all the palindromic substrings, we can efficiently find the minimum cuts in the following way:

#### **Steps:**

- If **isPalindrome[0][i]** is true, make **cuts[i] = 0**.
- Otherwise, first, initialize the value of **cuts[i]** to be infinite.
- Then for each *i*, take a variable *j* and initialize it to 0.
- Then loop through *j* such that  $0 \leq j < i$  and update **cuts[i]** :  
if  $1 + \text{cuts}[j]$  is less than the current value of **cuts[i]**—which represents that we found a better way of partitioning the substring str[0...i] with a lesser number of cuts than the previous one—then update the value of **cuts[i]** with the present value.
- An extra one is added because the entire substring is not palindrome, and thus we need to make a cut.
- Finally, the answer lies at **cuts[n - 1]** which is the final answer.

**Time Complexity:**  $O(N^2)$ , where **N** denotes the length of the string. As finding all palindromic substrings takes an order of  $N^2$  time, the overall complexity is of the order of  $N^2$ .

**Space Complexity:** Since we are using a two-dimensional array of size  $N \times N$ , the space complexity would be  $O(N^2)$ , where **N** denotes the length of the string.

### **3. Longest Palindromic Substring** [<https://coding.ninja/P107>]

**Problem Statement:** You are given a string (**STR**) of length **N**. Find the longest palindromic substring in the given string. If there is more than one palindromic substring with the maximum length, return the one with the smaller start index.

**Input Format:**

The first line of the input contains a single integer **T**, representing the number of test cases or queries to be run.

Then the **T** test cases follow.

The **first and only** line of each test case contains a string (**STR**).

**Output Format:**

For every test case, print the longest palindromic substring in the given input string.

**Sample Input:**

```
1
ababc
```

**Sample Output:**

```
aba
```

**Explanation:**

The longest palindromic substring of the given input string “**ababc**” is “**aba**”, since “**aba**” is the longest substring of length 3 which is also a palindrome in the given input string.

There is another palindromic substring of length 3—“**bab**”. However, the starting index of “**aba**” is lesser than “**bab**”; therefore “**aba**” is the required answer.

**Approach 1: Brute Force**

In this approach, we find out all the possible substrings of the given input string and check whether it is a palindrome or not.

**Steps:**

1. Generate all possible substrings of the given input string such that the substring having greater length is generated first.
2. To do this, run a loop where iterator **len** will go from **N** to **1**, where **N** is the length of the given string.
3. Run a nested loop and fix an iterator **j** that will point at the starting index of the current substring.
4. Get the substring from **j** to **j + len**.
5. If the chosen substring from **j** to **j + len** is a palindrome, return the substring as the answer (since this substring will be of the longest length and of minimum starting index).

**Time Complexity:**  $O(N^3)$ , where  $N$  denotes the length of the given string. We are creating every substring, which takes  $N^2$  time. For checking whether the substring is palindrome or not, it takes  $O(\text{length of substring})$  time, which can be maximum  $O(N)$ . Thus, the overall time complexity becomes  $O(N^3)$ .

**Space Complexity:**  $O(N)$ , where  $N$  denotes the length of the given input string. As we are storing the generated substrings in a variable that could go up to a maximum length of  $N$ , the overall space complexity becomes of the order of  $O(N)$ .

## Approach 2: Dynamic Programming

### Steps:

1. If the input string (**input**) length is less than or equal to 1, return the string.
2. Initialise a 2D **dp** array of data type boolean where **dp[i][j]** stores **false** if the substring **input[i, j]** is not palindromic; otherwise it stores **true**.
3. Store all the diagonal elements—that is, **dp[i][i]** as true, since **input[i, i]** will always be palindromic (string of length one is always palindromic).
4. For substring of length 2, check if **input[i]** is equal to **input[i + 1]**, if yes, then store **dp[i][i + 1]** as **true**, else **false**.
5. Run a loop for length of substring greater than 2. To calculate **dp[i][j]**, check if the value of **dp[i + 1][j - 1]** is **true** and **input[i]** is the same as **input[j]**, then make **dp[i][j]** true, otherwise **false**.
6. For every **dp[i][j]** as **true**, update the length and the starting index of the **answer** substring.
7. Return the **answer** substring of the string having minimum starting index and of maximum length.

**Time Complexity:**  $O(N^2)$ , where  $N$  denotes the length of the given input string. As we are traversing the **dp** array once, the overall time complexity is  $O(N^2)$ .

**Space Complexity:**  $O(N^2)$ , Where  $N$  denotes the length of the given input string, as we are using a **dp** array of  $N \times N$  dimensions, the overall space complexity is  $O(N^2)$ .

## Approach 3: Expanding around the centres

In this approach, the idea is to generate all the even and odd length palindromes and keep track of the longest palindrome seen so far and return the same at the end.

### Steps:

1. If the string length is less than or equal to one, then return the string, as a single character is always palindromic.

2. To generate odd length palindromes, run a loop where  $i$  varies between 0 to  $\mathbf{N} - 1$ , fix center as  $i$ , and expand in both directions for longer palindromes; odd length palindromes will have a character at the centre.
3. Similarly, for even length palindrome, fix the centres as  $i$  and  $i + 1$ , and expand in both directions; the even palindrome will have a partition between  $i^{\text{th}}$  char and  $(i + 1)^{\text{th}}$  char as the centre.
4. If the length of the current palindromic substring is greater than the current maximum, update the **start** and the **len** variables as that of the current substring.
5. Return the substring of the string having starting index as **start** and of length **len**.

**For expanding :**

1. For expanding around a centre  $i$  for odd length palindrome, initialise two variables **left** and **right** to  $i$  and go until **left** and **right** are in the range of the string and **input[left] == input[right]**. Decrement the **left** and increment the **right** variables.
2. For expanding around a centre  $i$  and  $i + 1$  for even length palindrome, initialise two variables **left = i** and **right = i + 1**, and go until **left** and **right** are in the range of the string and **input[left] == input[right]**. Decrement the **left** and increment the **right** variables.

**Time Complexity:**  $O(N^2)$ , where  $\mathbf{N}$  denotes the length of the given input string. As we are expanding twice for every index (one for odd length and one for even length ) and since expanding takes  $O(N)$  in the worst case, therefore the overall time complexity is of the order of  $O(N^2)$ .

**Space Complexity:**  $O(1)$  as we are not using any extra space; therefore the space complexity is constant.

## 4. Longest Palindromic Subsequence [<https://coding.ninja/P108>]

**Problem Statement:** Given an input string **A** consisting of lowercase english letters, your task is to find the length of the longest palindromic subsequence in **A**.

A subsequence is a sequence generated from a string after deleting some or no characters of the given string without changing the order of the remaining string characters.

**Example:** “ace” is a subsequence of the string “abcde” while “aec” is not a subsequence of the given string.

**Input Format:**

The **first line** of input contains a single integer **T** representing the number of test cases.

Then the test cases follow.

The **only line** of each test case contains a single string **A** consisting of only lowercase english letters.

### **Output Format:**

For each test case, print a single integer denoting the length of the longest palindromic subsequence in the input string **A**.

### **Sample Input:**

1  
babcbcab

### **Sample Output:**

7

### **Explanation:**

For the given test case, the longest palindromic subsequence is “**babcbab**”, which has a length of 7. Strings “**bbbbbb**”, “**bbabb**” and “**bbccb**” are also palindromic subsequences of the given string; however they are not the longest ones.

### **Approach 1: Recursive Approach**

The main idea behind this approach is to use recursion. The idea is to compare the first character of the string **A[i...j]** with its last character. Let **A[0...n - 1]** be the input string of length **n** and **L(0, n - 1)** be the length of the longest palindromic subsequence of **A[0...n - 1]**.

Then there are two possibilities:

1. If the first character of the string is the same as the last character, we include the first and last characters in the palindrome and do a recursive call for the remaining substring **A[i + 1, j - 1]**.
2. If the last character of the string is different from the first character, we return a maximum of the two values we get by:
  - a. removing the last character and doing a recursive call for the remaining substring **A[i, j - 1]**.
  - b. removing the first character and doing a recursive call for the remaining substring **A[i + 1, j]**.

### **Steps:**

1.  $L(i, i) = 1$  for all indices **i** in the given string because every single character is a palindrome of length 1.
2. If the first and the last characters are not the same—that is,  $A[i] \neq A[j]$ ,  
$$L(i, j) = \max \{L(i + 1, j), L(i, j - 1)\}$$
3. If the first and last characters are the same—that is,  $A[i] == A[j]$ ,  
$$L(i, j) = L(i + 1, j - 1) + 2$$

**Time Complexity:**  $O(2^N)$ , where **N** denotes the length of the given input string. In the worst case,

when all the characters in the given string are unique (LPS length is 1), each recursive call will end up in two recursive calls, thereby making the final time complexity  $O(2^N)$ .

**Space Complexity:**  $O(N)$ , where  $N$  denotes the length of the given input string. We are using  $O(H)$  extra space for the call stack where  $H$  is the height of the recursion tree, and the height of the recursion tree could be  $N$  in the worst case when all the characters present in the given string are unique. Thus, the final space complexity is  $O(N)$ .

### Approach 2: DP Approach

The LPS (Longest Palindromic Subsequence) problem has an optimal substructure and also exhibits overlapping subproblems. So we create a 2 D array  $dp$  to store the results. We will consider the state  $dp[i][j]$  to store the length of the longest palindromic subsequence of  $substring[i, j]$  of the given string.

#### Steps:

1. Store the answer to the base subproblems—that is,  $dp[i][i] = 1$ , since every single character is a palindrome of length 1.
2. Iterate over all possible lengths in increasing order, as we need to start from smaller subproblems and reach the bigger ones.
3. For a substring in range  $i, j$ —that is,  $substring[i, j]$ :
  - a. If both the end characters of the range match, then the current answer in  $dp[i][j]$  will be equal to the answer of the smaller range ( $dp[i + 1][j - 1]$ ) + 2.
  - b. Otherwise, if both the end characters don't match, we check for the answers for the smaller ranges—that is, the values of  $dp[i + 1][j]$  or  $dp[i][j - 1]$  and store the maximum between both the options at  $dp[i][j]$ .
4. The answer to the complete string is stored at  $dp[0][n - 1]$ .

**Time Complexity:**  $O(N^2)$ , where  $N$  denotes the length of the given input string. Since there are  $N * N$  states in total, the final time complexity is  $O(N^2)$ .

**Space Complexity:**  $O(N^2)$ , where  $N$  denotes the length of the given input string. Since we are using a dp array of size  $N * N$ , the final space complexity is  $O(N^2)$ .

## 5. Minimum Cost to Reach N [\[https://coding.ninja/P109\]](https://coding.ninja/P109)

**Problem Statement:** You are given a number  $N$ . Initially, you are at 0, and you are supposed to reach  $N$  by performing any one of the following operations any number of times:

1. Add 1, at the cost of  $A$ .
2. Double the number at the cost of  $B$ .

Your task is to find the minimum cost required to reach the number  $N$  starting from 0.

**Note:** There can be two or more paths with the same cost; you can consider any one of them, but the overall cost to reach from 0 to **N** must be minimised.

#### **Input Format:**

The **first line** of input contains an integer **T** representing the total number of the test cases.

Then the test cases follow:

The **first and only line** of each test case contains three space-separated integers **N**, **A** and **B**.

#### **Output Format:**

For each test case, print the minimum cost to reach from 0 to **N** using the two allowed operations.

#### **Sample Input:**

```
1
5 2 1
```

#### **Sample Output:**

```
6
```

#### **Explanation:**

We are given **N** = 5, **A** = 2, and **B** = 1, then the sequence of operations to get the minimum cost to reach 5 from 0 could be :

1. Adding 1 to 0 with cost 2. Total cost becomes 2.
2. Doubling 1 with cost 1. Therefore the total cost becomes 3.
3. Doubling 2 will result in 4 with a cost of 1. Therefore the total cost becomes 4.
4. Adding 1 in 4 will result in 5 (which is the desired number) with a cost of 2. Thus the minimum cost to reach 5 from 0 is **6**.

### **Approach 1: Recursion**

In this approach, we will recursively explore all the possible ways to reach **N** from 0. Instead of reaching from 0 to **N**, we could also go from **N** to 0. Hence, we have two choices—subtract 1 from **N** at the cost of **A**, or halve **N** at the cost of **B**.

#### **Steps:**

1. If **N** is 1, return **A**, as all we need to do is reach 0 by subtracting 1 at the cost of **A**.
2. If **N** is odd, there is no point in halving **N**, hence:
  - a. Recursively call for the result that will be obtained by subtracting one from **N**. Store value in **RESULT**.

- b. Return **A + RESULT**.
- 3. If **N** is even, explore both the possibilities of subtracting one from **N** and halving **N**:
  - a. Recursively call for the result that will be obtained by subtracting one. Store the value in a variable **ADDCOST**.
  - b. Add **A** to **ADDCOST** to get the final cost.
  - c. Recursively call for the result that will be obtained by halving **N**. Store the result in a variable **DOUBLECOST**.
  - d. Add **B** to **DOUBLECOST** to get the final cost.
  - e. Store the minimum of **ADDCOST** and **DOUBLECOST** in your final answer.

**Time Complexity:**  $O(2^N)$ , where **N** denotes the number given in the question. For each step, we may have two choices—to either subtract 1 from **N** or halve **N**. A recursive tree will be generated for these two options; hence the overall time complexity is  $O(2^N)$

**Space Complexity:**  $O(N)$ , because of the stack space required in memory for recursive calls, which might go up to the order of **N** in the worst case.

## Approach 2: Memoization

In this approach, we will optimise the recursive approach using memoization.

### Steps:

1. Initialise a single-dimensional memoization array **MEMO** of size **N + 1** with -1.
2. If **N** is 1, return **A**, as all we need to do is to reach 0 by subtracting 1 at the cost of **A**.
3. If **MEMO[N] > 0**, return **MEMO[N]**.
4. If **N** is odd, there is no point in halving **N**, hence:
  - a. Recursively call for the result that will be obtained by subtracting one from **N**. Store value in **RESULT**.
  - b. Store **A + RESULT** in **MEMO[N]** and return **MEMO[N]** as the answer.
5. If **N** is even, explore both the possibilities of subtracting one from **N** and halving **N**:
  - a. Recursively call for the result that will be obtained by subtracting one. Store value in a variable **ADDCOST**.
  - b. Add **A** to **ADDCOST** to get the final cost.
  - c. Recursively call for the result that will be obtained by halving **N**. Store the result in a variable **DOUBLECOST**.
  - d. Add **B** to **DOUBLECOST** to get the final cost.
  - e. Store minimum of **ADDCOST** and **DOUBLECOST** in **MEMO[N]** and return it.

**Time Complexity:**  $O(N)$ . Since every element till number **N** will be visited once and once the result for a number is stored in **MEMO**, it can be fetched directly in constant time; hence the overall time complexity becomes  $O(N)$ .

**Space Complexity:**  $O(N)$  as an additional array of size  $N + 1$  is used to store the results for the recursive calls.

### Approach 3: Dynamic Programming

In this approach, we will follow the iterative procedure :

1. If  $N$  is 1, return  $A$ , as all we need to do is to just reach 0 by subtracting 1 at the cost of  $A$ .
2. Create an array  $DP$  of size  $N + 1$  and initialize  $DP[0]$  to 0,  $DP[1]$  to  $A$  and  $DP[2]$  to  $A + \min(A, B)$ .
3. Iterate over  $DP[i]$  for each  $3 \leq i \leq N$ :
  - a. Check if  $N$  is even:
    - i. Find the minimum of  $DP[i - 1] + A$  and  $DP[i/2] + B$  and store it in  $DP[i]$ .
  - b. Else:
    - i. Store  $A + DP[i - 1]$  in  $DP[i]$ .
4. Return  $DP[N]$  as the answer.

**Time Complexity:**  $O(N)$ . Since every element till number  $N$  will be visited once, and once the result for a number is stored in  $DP$ , it can be fetched directly in constant time, thus the overall time complexity becomes  $O(N)$ .

**Space Complexity:**  $O(N)$ , as we use an additional array in memory of size  $N+1$  .

## 6. Rod Cutting Problem [<https://coding.ninja/P110>]

**Problem Statement:** You are given a rod of length  $N$  units. The rod can be cut into different sizes, and each size has a cost associated with it. Determine the maximum cost obtained by cutting the rod into different pieces and selling those pieces.

**Note:** The sizes of the different pieces will range from 1 to  $N$  and will be integers. The sum of the length of the cut pieces should always be equal to  $N$ .

#### Input Format:

The **first line** of input contains an integer  $T$  denoting the number of test cases.

The next  $2 * T$  lines represent the  $T$  test cases.

The **first line** of each test case contains an integer  $N$  denoting the length of the rod.

The **second line** of each test case contains a vector  $A$  of size  $N$  representing the cost of different lengths, where each index of the array is the sub-length, and the element at that index is the cost for that sub-length.

**Note:**

Since 1-based indexing is considered, the 0th index of the vector **A** will represent a sub-length 1 of the rod. Hence the  $(n - 1)^{\text{th}}$  index would represent cost for the length **N**.

**Output Format:**

For each test case, return the maximum cost obtained by selling the pieces.

**Sample Input:**

1

5

2 5 7 8 10

**Sample Output:**

12

**Explanation:**

All possible partitions for the rod of length 5 are:

1, 1, 1, 1, 1      max\_cost =  $(2 + 2 + 2 + 2 + 2) = 10$

1, 1, 1, 2      max\_cost =  $(2 + 2 + 2 + 5) = 11$

1, 1, 3      max\_cost =  $(2 + 2 + 7) = 11$

1, 4      max\_cost =  $(2 + 8) = 10$

5      max\_cost =  $(10) = 10$

2, 3      max\_cost =  $(5 + 7) = 12$

1, 2, 2      max\_cost =  $(2 + 5 + 5) = 12$

Clearly, if we cut the rod into lengths (1, 2, 2) or (2, 3), we get the maximum cost which is **12**.

**Approach 1: Recursive Approach**

Here we will follow a simple recursive approach to check for all the possible cuts in the rod.

**Steps:**

1. Initialize a variable **max\_cost** to **INT\_MIN**.
2. In each recursive function call, run a loop where **i** ranges from 0 to **N - 1**, and partition the rod of length **N** into two parts, **i** and **N - i**. For example, for a rod of length 5, for **i** = 2, the two partitions will be 2 and 3.
  - a. Take **i** as the first cut in the rod and keep it constant. Now, a rod of length **N - i** remains. Pass the remaining rod of size **N - i** to the recursion.
  - b. Step 1 continues until it hits the base condition—that is, when the length of the rod becomes zero, for which the answer is zero.

- c. When the recursion hits the base condition, the cost of a particular configuration of cuts to the rod is obtained.
  - d. Compare the returned value with the **max\_cost** variable and store the maximum of the cost in variable **max\_cost**.
3. For every **i** as an initial sub-length, different configurations are obtained and compared to the **max\_cost** variable.
  4. Lastly, **max\_cost** contains the final answer.

**Time Complexity:**  $O(2^N)$ , where **N** denotes the length of the rod. In the worst case, all possible combinations for length **N** are recursively found.

**Space Complexity:**  $O(N)$ , where **N** denotes the length of the rod. The recursive stack uses space of the order **N**.

### Approach 2: Top-down DP approach

The idea is to select a sub-length of rod out of all possible sub-lengths, as one option and ignore this sub-length, as another option.

In this way, we will get all possible configurations, and we calculate the **max\_cost** for all the possible configurations and return the maximum of them all.

#### Steps:

1. Take a two-state dp, **cost[n][max\_len]**, where **N** denotes the sub-length of the rod to be processed and **max\_length** denotes the rod left uncut.
2. Calculate all possible configurations using recursion.
  - a. If the length of the sub-length of the rod considered is less than or equal to the **max\_length**, that is, the rod left uncut, then for a particular sub-length, **x**.
    - i. Either include it and reduce the max-length by **x** and recur.
    - ii. Or exclude it and keep the max-length unchanged and recur.
  - b. Else, recur with reducing the value of **N** but keeping **max\_length** unchanged.
3. Keep storing the maximum cost at every step in the dp array, **cost[ ][ ]** so that it need not be re-calculated further.
4. The value of the greater lengths is calculated by selecting the maximum value from all the possible configurations of shorter lengths.
5. Lastly, **cost[n][max\_len]** stores the maximum cost obtained for a rod of length **N**.

**Time Complexity:**  $O(N^2)$ , where **N** denotes the length of the rod. In the worst case, all possible combinations of partitions are found, and because subproblems are stored, they need not be recalculated.

**Space Complexity:**  $O(N^2)$ , where **N** denotes the length of the rod. The recursive stack uses space of the order **N**.

### **Approach 3: Bottom-up DP approach**

We can observe that the problem has optimal substructure and overlapping subproblems and hence can be solved using the dynamic programming approach.

Below is a bottom-up approach in which the smaller subproblems are solved and stored first, which are further used to solve the larger sub-problems. The below approach computes **cost[i]**, which stores the maximum profit achieved from the rod of length **i** for each  $1 \leq i \leq n$ .

#### **Steps:**

1. Declare an array, **cost** of size **N + 1** and initialize **cost[0] = 0**. We use this 1D array to store the previous cost, and with every iteration, we update the **cost** array.
2. Run a loop where  $1 \leq i \leq N$ . Consider **i** as the length of the rod.
3. Divide the rod of length **i** into two rods of length **j** and  $i - j$  using a nested loop in which  $0 \leq j < i$ . Find the cost of this division using **A[j] + cost[i - j]** and compare it with the already existing value in the array **cost[]**.
4. Store the maximum value at every index **x**, which represents the maximum value obtained for partitioning a rod of length **x**.
5. Lastly, **cost[n]** stores the maximum cost obtained for a rod of length **N**.

**Time Complexity:**  $O(N^2)$ , where **N** denotes the length of the rod. In the worst case, all possible combinations of partitions are found, and because subproblems are stored, they need not be recalculated.

**Space Complexity:**  $O(N)$ , where **N** denotes the length of the rod. An array of length **N + 1** is used to store the results of the subproblems.

## **7. Optimal Strategy for a Game** [<https://coding.ninja/P111>]

**Problem Statement:** You and your friend Ninjax are playing a game of coins. Ninjax places **N** number of coins in a straight line.

The rules of the game are as follows:

1. Each coin has a value associated with it.
2. It is a two-player game played with alternating turns.
3. In each turn, the player picks either the first or the last coin from the line and permanently removes it.
4. The value associated with the coin picked by the player adds up to the total amount of that player.

Ninjax is a good friend of yours and asks you to start first. Your task is to find the maximum amount that you can definitely win at the end of this game.

#### **Note:**

**N** is always an even number. Ninjax wants to win the game as well, which means he will also play to maximise the amount he wins.

#### **Input Format:**

The **first line** contains an integer **T** denoting the total number of test cases or queries to be run.

For each test case:

The **first line** contains an integer **N** denoting the number of coins present in the line initially.

The **second line** contains **N** space-separated integers denoting the values associated with the coins placed by Ninjax.

#### **Output Format:**

For each test case, print the maximum amount that you can earn in a separate line.

#### **Sample Input:**

1

4

9 5 21 7

#### **Sample Output:**

30

#### **Explanation:**

The values associated with four coins are [9, 5, 21, 7].

Let's say that initially, you pick 9, and Ninjax picks 7. Then, you pick 21, and Ninjax picks 5. So, you win a total amount of (9 + 21)— that is, 30.

In case you would have picked up 7 initially, and Ninjax would have picked 21 (as he plays optimally). Then, you would pick 9, and Ninjax would choose 5. So, you win a total amount of (7 + 9)— that is, 16.

Thus, the maximum amount you can win is **30**, which is the desired answer.

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

In this approach, we find all the possible combinations recursively.

- 1) Suppose it's your turn, and you are left with coins in the index range  $[i, j]$  (all other coins have already been picked up in previous turns). You have the option to pick either  $i^{\text{th}}$  or  $j^{\text{th}}$  coin. Of these two options, you would select the one which maximises your winning amount:
  - a) If **you pick the  $i^{\text{th}}$  coin**. The other player will have the option to pick  $(i+1)^{\text{th}}$  or  $j^{\text{th}}$  coin :

- If the **other player picks the  $(i + 1)^{\text{th}}$  coin**. You can pick either end of the **range  $[i + 2, j]$** .
- If the **other player picks  $j^{\text{th}}$  coin**. You can pick either end of the **range  $[i + 1, j - 1]$** .

As the other player wants to maximise his amount (thereby minimizing the amount you can win).

Hence, of the two ranges which you are left with (mentioned above), you can only use that range that gives you the minimum amount.

- Similarly, if **you pick the  $j^{\text{th}}$  coin, the** other player will have the option to pick  **$i^{\text{th}}$**  or  **$(j - 1)^{\text{th}}$  coin**.
    - If the **other player picks the  $i^{\text{th}}$  coin**. You can pick either end of the **range  $[i + 1, j - 1]$** .
    - If the **other player picks  $(j-1)^{\text{th}}$  coin**. You can pick either end of the **range  $[i, j - 2]$** .
- Similarly, here, of the two ranges which you are left with (mentioned above), you can only use that range which gives us the minimum amount.
- We'll use a helper function that would return the maximum amount you can win for a given sub-array of coins and return the answer .

### Steps:

**Int helper(vector<int> coins, int i, int j):** The helper function that returns the maximum amount which you can earn:

- Base Case 1: if  $i > j$ , this is not possible, so return 0.
- Base Case 2: if  $i = j$ , means only one element in the sub-array, hence return **coins[i]**.
- Select the  $i^{\text{th}}$  coin, and recursively call the helper function for the two possible cases and select the one which gives the minimum.
  - X = coins[i] + min( helper(coins, i + 1, j - 1), helper(coins, i + 2, j))**
- Select the  $j^{\text{th}}$  coin, and recursively call the helper function for the two possible cases and select the one which gives the minimum.
  - Y = coins[j] + min( helper(coins, i + 1, j - 1), helper(coins, i, j - 2))**
- For the above two possible amounts, select the maximum one.
- AMOUNT = Max(X, Y)**
- Return this selected amount.

**Int optimalStrategyOfGame(vector<int> coins, int n):**

- Call **helper(coins, 0, n - 1)**
- Return the result generated by the helper function.

**Time Complexity:  $O(4^N)$**  where **N** is the number of coins lined up initially. In the worst case, the helper function calls four recursive calls in it.

The time complexity can be derived from the recurrence relation:

$$\begin{aligned}
 T(N) &= T(N-1) + T(N-1) + T(N-1) + T(N-1) \\
 &= 4 * T(N-1)
 \end{aligned}$$

**Space Complexity:**  $O(N)$  where  $N$  denotes the number of coins lined up initially. In the worst case, extra space will be used by the recursion stack as there can be a maximum of  $N$  number of recursive calls at a time.

### Approach 2: Optimized Recursive Brute Force

In the previous approach, we were making four recursive calls every time we called the helper function but a better approach to solve would be as follows:

#### Steps:

1. Use a helper function that will return the maximum amount you can win for a given subarray of coins.
2. Suppose it's your turn and you are left with coins in the index range  $[i, j]$  (other coins have already been picked up in the previous turns). You have the option to pick either  $i^{\text{th}}$  or  $j^{\text{th}}$  coin. Let **SUM** store the sum of coins in the subarray  $[i, j]$ .
  - a. If **you pick the  $i^{\text{th}}$  coin**. The other player will have the option to pick  $(i + 1)^{\text{th}}$  or  $j^{\text{th}}$  coin. The other player will choose the coin such that it minimizes the amount you can earn.  
So you can collect the value **coins[i] + (SUM - coins[i]) - helper(i + 1, j, SUM - coins[i])**. The expression can be simplified to **SUM - helper(i + 1, j, SUM - coins[i])**.
  - b. If **you pick the  $j^{\text{th}}$  coin**. The other player will have the option to pick  $i^{\text{th}}$  or  $(j - 1)^{\text{th}}$  coin. The other player will choose the coin such that it minimizes the amount you can earn.  
So you can collect the value **coins[j] + (SUM - coins[j]) - helper(i, j - 1, SUM - coins[j])**. The expression can be simplified to **SUM - helper(i + 1, j, SUM - coins[j])**.
  - c. As we want to maximise our amount, we will return the maximum out of the two.

Algorithm for the helper function would be

→ **Int helper(vector<int> coins, int i, int j, int SUM):**

1. Base Case: if  $i = j + 1$ , return the maximum of the two coins.
2. Return Maximum of **(SUM - helper(i+1, j, SUM - coins[i]), (SUM - helper(i, j - 1, SUM - coins[j]))**

**Time Complexity:**  $O(2^N)$  per test case where  $N$  denotes the number of coins lined up initially. In the worst case, the helper function makes two recursive calls for at most  $N$  times.

The time complexity can be derived from the recurrence relation:

$$T(N) = T(N - 1) + T(N - 1)$$

$$= 2 * T(N - 1)$$

**Space Complexity:**  $O(N)$  per test case where  $N$  denotes the number of coins. In the worst case, extra space will be used by the recursion stack which can go to a maximum depth of  $N$ .

### Approach 3: DP Memoization

Approach 2 solves the same sub-problems multiple times—that is, the helper function gets called for the same sub-array (index range) multiple times and we use a 2D DP array called the **lookup** to store the answers.

#### Steps:

**Int helper(vector<int> coins, int i, int j, int SUM, lookUp[][]):**

1. Base Case: **if  $i=j+1$** , return the maximum of the two coins.
2. If **lookUp[i][j] != -1**, return **lookUp[i][j]**.  
Otherwise, move to the next step.
3. **AMOUNT = Maximum of (SUM - helper(coins,i+1, j, SUM - coins[i],lookUp), SUM - helper(coins,i, j-1, SUM - coins[j],lookUp))**
4. Store the **AMOUNT** in **lookUp[i][j]**.
5. Return the **AMOUNT**

**Time Complexity:**  $O(N^2)$  per test case where  $N$  denotes the number of coins lined up initially. In the worst case, we will be filling the lookup table of size  $N * N$  to find out the required solution for all the subarrays (ranges).

**Space Complexity:**  $O(N^2)$  per test case where  $N$  denotes the number of coins. In the worst case, the extra space is required by the look-up table of size  $N * N$ .

### Approach 5: DP Tabulation

#### Steps:

1. The idea is to create a 2D table of size  $N * N$ .
2. Each entry in the table stores the solution to a subproblem i.e. **table[i][j]** represents the maximum amount you can win for the **subarray coins[i, j]** given that you start first.
3. The **algorithm** for filling the table is as follows:
  - a. Loop 1: For **len = 1** to  $N$ :
  - b. Loop 2: For  $i = 0$  to  $(N - len)$ :  
→ Let  $j = i + len - 1$   
→ If  $(i + 1) < N \&& (j - 1) \geq 0$  then,  $(A = dp[i + 1][j - 1], \text{otherwise } A = 0.)$   
→ If  $(i + 2) < N$  then,  $B = dp[i + 2][j], \text{otherwise } B = 0.$   
→ If  $(j - 2) \geq 0$  then,  $C = dp[i][j - 2], \text{otherwise } C = 0.$   
→ Update  $dp[i][j] = \max(\text{coins}[i] + \min(A, B), \text{coins}[j] + \min(A, C)).$

- c. End Loop 2.
  - d. End Loop 1.
4.  $\text{dp}[0][n - 1]$  stores the final answer.

**Example:**

Let us understand the above algorithm in detail with an example:

For the test case where  $N = 4$  and coins List = [5, 40, 4, 1].

- We will create a table of dimension  $N * N$  ( $4 * 4$ ), with all elements initialised to 0.
- Each entry in the table stores the solution to a subproblem—that is,  $\text{table}[i][j]$  represents the maximum amount you can win for the **subarray coins[i, j]** given that you start first.
- The 'len' variable in the algorithm represents the length of the subarray being considered in the current iteration.
- For the first iteration, len = 1, consider all the subarrays with length 1. All the cells with  $i = j$  are filled in this iteration. Filling the table according to the conditions mentioned in the algorithm above, we get:

5	0	0	0
0	40	0	0
0	0	4	0
0	0	0	1

- For the second iteration, len = 2, so consider all the subarrays with length 2. All the cells with  $(i + 1 = j)$  are filled in this iteration. Filling the table according to the conditions mentioned in the algorithm above, we get:

5	40	0	0
0	40	40	0
0	0	4	4
0	0	0	1

$\max(40 + \min(0, 0), 4 + \min(0, 0))$

- For the third iteration, len = 3, so consider all the subarrays with length 3. All the cells with  $(i + 2 = j)$  are filled in this iteration. Filling the table according to the conditions mentioned in the algorithm above, we get:

5	40	9	0
0	40	40	41
0	0	4	4
0	0	0	1

$\max(40 + \min(4, 1), 4 + \min(4, 40))$

- For the fourth iteration,  $\text{len} = 4 (=N)$ , so consider all the subarrays with length 4. All the cells with  $(i + 3 = j)$  are filled in this iteration. Filling the table according to the conditions mentioned in the algorithm above, we get:

5	40	9	41
0	40	40	41
0	0	4	4
0	0	0	1

$\max(5 + \min(40, 4),$   
 $1 + \min(40, 4))$

- The cell marked as red (Table[0][n-1]) stores the final answer.

**Time Complexity:**  $O(N^2)$  per test case where  $N$  denotes the number of coins. In the worst case, we will be filling the table of size  $N * N$  to find out the required solution for all the subarrays (ranges).

**Space Complexity:**  $O(N^2)$  per test case where  $N$  denotes the number of coins. In the worst case, the extra space is required by the DP table of size  $N * N$ .

**Note:** There is one approach to this problem using memoization, which can be accessed through the CodeStudio under Solutions tab.

## 8. Edit Distance [\[https://coding.ninja/P112\]](https://coding.ninja/P112)

**Problem Statement:** You are given two strings **S** and **T** of lengths **N** and **M**, respectively. Find the 'Edit Distance' between the strings.

Edit Distance of two strings is the minimum number of steps required to make one string equal to the other. In order to do so, you may perform the following three operations any number of times:

1. Delete a character
2. Replace a character with another character
3. Insert a character

### Input Format:

The **first line** of input contains the string **S** of length **N**.

The **second line** of the input contains the String **T** of length **M**.

### Output Format:

The only line of output prints the minimum 'Edit Distance' between the two given input strings.

### Sample Input:

abc  
dc

### Sample Output:

2

### Explanation:

In 2 operations, we can make the string **T** equal to string **S**.

1. Firstly, insert the character '**a**' to string **T**, which makes it "**adc**".
2. Secondly, replace the character '**d**' of the string **T** with '**b**'. This would make string **T** to "**abc**" which is also the same as string **S**.

Hence, the minimum edit distance for the given sample of input is **2**.

### Approach 1: Recursive Approach

Here we follow a naive recursive approach discussed as below:

#### Steps:

1. The base case would be if the length of the first string is zero, return the length of the second string. Similarly, if the length of the second string is zero, return the length of the first string.
2. Now the recurrence relation is as follows:
  - a. If the last characters of two strings are not the same, try all the three allowed operations. Here *i* is initially the last index of **s1**, and *j* is initially the last index of **s2** and **f(i, j)** is the edit distance of the two strings **s1** and **s2** till index *i* and *j*, respectively. Then:
$$f(i, j) = 1 + \min(f(i - 1, j), f(i, j - 1), f(i - 1, j - 1))$$
Here the three recursive calls are for the operations of insertion, removal and replacement, respectively.
  - b. If the last characters of the two strings are the same—that is, **s1[i] == s2[j]** then the relation would be:
$$f(i, j) = f(i - 1, j - 1)$$

**Time Complexity:**  $O(3^{(N*M)})$ , where **N** denotes the length of the first string and **M** is the length of the second string.

**Space Complexity:**  $O(N + M)$  since the recursion stack might go upto a depth of maximum **N + M**, thereby making the space complexity  $O(N + M)$ .

### Approach 2: Memoization

The idea here is to write a recursive approach and memoize it.

#### Steps:

1. Make a **2D** array of size  $(N + 1) * (M + 1)$  where **N** and **M** are the lengths of the two strings **s1** and **s2** and initialise it with -1, which will indicate that the answer for that state has not yet been calculated.
2. The base case would be if the length of the first string is zero, return the length of the second string. Similarly, if the length of the second string is zero, return the length of the first string.
3. Else check the value of the dp array at index[i, j]. If it's not equal to -1, simply return its value.
4. Now the recurrence relation is as follows:
  - a. If the last characters of two strings are not the same, try all the three allowed operations. Here *i* is initially the last index of **s1**, and *j* is initially the last index of **s2**, and **f(i, j)** is the edit distance of the two strings **s1** and **s2** till index *i* and *j*, respectively. Then:
$$f(i, j) = 1 + \min(f(i - 1, j), f(i, j - 1), f(i - 1, j - 1))$$
Here the three recursive calls are for the operations of insertion, removal and replacement, respectively.
  - b. If the last characters of the two strings are the same—that is, **s1[i] == s2[j]** then the relation would be:
$$f(i, j) = f(i - 1, j - 1)$$

5. Before returning the result, store it in the array at index [i, j].
6. The final answer would be stored at the index [N, M] in the matrix.

**Time Complexity:**  $O(N * M)$ , where **N** denotes the size of the first string and **M** is the size of the second string. Since we fill the entire memoization array and we visit each cell just once via recursion, therefore, the time complexity is  $O(N * M)$ Initialise.

**Space Complexity:**  $O(N * M)$ , where **N** denotes the size of the first string and **M** is the size of the second string. Since we are making an auxiliary two-dimensional array of size **N \* M**, the overall space complexity  $O(N * M)$ .

#### Approach 3: Iterative DP

Here we use the dynamic programming approach to solve the problem. We will have a two-dimensional array, **dp** where:

**dp[i][j]** stores the edit distance of the  $(i + 1)^{th}$  length substring of **str1** and  $(j + 1)^{th}$  length substring of **str2** starting from index 0.

#### Steps:

- When the size of first-string is zero, then the edit distance will be equal to the size of other string—that is when :

$i = 0$  then  $\text{dp}[i][j] = j$  and when  $j = 0$  then  $\text{dp}[i][j] = i$ .

- Now we will use our recurrence relation: If  $\text{str1}[i - 1] == \text{str2}[j - 1]$ :

$$\text{dp}[i][j] = \text{dp}[i - 1][j - 1]$$

- Otherwise:

$$\text{dp}[i][j] = 1 + \min(\text{dp}[i][j - 1], \text{dp}[i - 1][j], \text{dp}[i - 1][j - 1])$$

- We will have our final answer stored at the index  $(N, M)$  in the  $\text{dp}$  matrix.

**Time Complexity:**  $O(N * M)$ , where **N** denotes the size of the first string and **M** is the size of the second string. Since we are running two nested loops of sizes **N** and **M** therefore the time complexity is  $O(N * M)$ .

**Space Complexity:**  $O(N * M)$ , where **N** denotes the size of the first string and **M** is the size of the second string. Since we are making an auxiliary two-dimensional array of size **N \* M**, the space complexity is  $O(N * M)$ .

## 9. Maximum Sum of Two Subarrays of Given Size

[<https://coding.ninja/P113>]

**Problem Statement:** You are given an array/list **ARR** of integers and a positive integer **K**. Your task is to find two non-overlapping subarrays (contiguous) each of length **K** such that the total sum of the elements in these subarrays is maximum.

### Input Format:

The **first line** of input contains an integer **T** representing the number of test cases or queries to be run.

Then **T** test cases follow:

The **first line** of each test case contains two single space-separated integers **N** and **K**, respectively, where **N** represents the size of the input array/list and **K** represents the length of the two subarrays.

The **second line** contains **N** single space-separated integers representing the array/list elements.

### Output Format:

For each test case, print the total maximum possible sum of two non-overlapping subarrays of size **K** in the given input array/list.

### Sample Input:

```
1  
7 2  
2 5 1 2 7 3 0
```

### Sample Output:

```
17
```

### Explanation

We are given **ARR** = [2, 5, 1, 2, 7, 3, 0] and **K** = 2, the output for the given input is **17**. We can choose two non-overlapping subarrays as [2, 5] and [7, 3] to get a total sum of 17 (2 + 5 + 7 + 3) which is the maximum possible sum.

### Approach 1: Brute Force

The problem boils down to finding the sum of all pairs of non-overlapping subarrays of size **K** in the given array. We can naively find all non-overlapping subarray pairs by taking two nested loops, with the outer loop (loop variable i) running from 0 to **N - 2K** and inner loop (loop variable j) running from  $i + K$  to **N - K**.

The range of loops is taken in such a way that it will always prevent any sort of overlapping of the two chosen subarrays.

### Steps:

1. Initialize **maxSum** to **INT\_MIN**.
2. Run a loop (loop variable i) from 0 to **N - 2\*K**.
  - a. Find the sum of a subarray of size **K** starting from index *i*. Let this sum be **sum1**.
  - b. Now, run a loop (loop variable j) from  $i + K$  to **N - K** as the starting index for the second subarray:
    - i. Find the sum of another subarray of size **K** starting from index *j*. Let this sum be **sum2**.
    - ii. If **maxSum** is less than the total sum (**sum1 + sum2**), update it—that is,  
**maxSum = sum1 + sum2**.
3. Finally, return **maxSum** as the answer.

**Time Complexity: O(K\*N<sup>2</sup>)**, where **N** denotes the size of array/list **ARR**.

The inner loop variable *j* varies with the outer loop variable *i* as  $i + K$ .

For  $i = 0$ , *j* varies from  $K$  to  $N - K$ . So, the loop runs  $N - 2K$  times.

For  $i = 1$ , *j* varies from  $K + 1$  to  $N - K$ . So, the loop runs  $N - 2K - 1$  times.

For  $i = 2$ , *j* varies from  $K + 2$  to  $N - K$ . So, the loop runs  $N - 2K - 2$  times, and so on.

For  $i = N - 2K$ , *j* varies from  $N - K$  to  $N - K$ . So, the loop runs 1 time.

Thus the total number of times the function (getSum) is executed is

$$1 + 2 + 3 + 4 + \dots + (N - 2K - 2) + (N - 2K - 1) + (N - 2K) = (N - 2K)(N - 2K + 1)/2$$

The time complexity to find the sum of K elements using a loop in `getSum` function is **O(K)**. So, the overall time complexity is **O(N<sup>2\*K</sup>)**.

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

## Approach 2: Dynamic Programming

In this approach, we store the prefix sum (the sum of all elements from first index until the current index) in the **prefixSum** array to calculate the sum of subarrays of size **K** in constant time.

We also maintain a **dp** array where **dp[i]** denotes the subarray sum that is maximum among all possible subarrays of size **K** till index **i**.

For the subarray of size **K** ending at index **i**—that is, subarray (**[i - K + 1: i]**), the maximum subarray sum of size **K** that ends before this subarray is given by **dp[i - K]**.

### Steps:

1. Initialize **maxSum** to **INT\_MIN**.
2. Create a **prefixSum** array.
3. Create **dp** array as follows:
  - a. Initialise **dp[K]** to the sum of the first subarray of size **K**—that is, **prefixSum[K]**.
  - b. Run a loop from **K + 1** to **N**.
    - i. Set **dp[i]** to the maximum of **dp[i - 1]** and (**prefixSum[i] - prefixSum[i - K]**).
4. Run a loop from **2 \* K** to **N**. The beginning of this loop is chosen as such to prevent any overlap between the two subarrays :
  - a. If the total of subarray sum—that is, (**dp[i] + dp[i - K]**) (maximum subarray sum ending on or before index **i** and at index **i - K** respectively) is more than **maxSum**, then update it.
5. Finally, return **maxSum** as the answer.

**Time Complexity:** **O(N)**, where **N** denotes the size of the input array.

The time complexity to create a prefix sum array is **O(N)**. The time complexity to create a DP array is **O(N)**. The time complexity of looping from **2 \* K** to **N** is **O(N)**.

Thus, the overall time complexity is **O(N)**.

**Space Complexity:** **O(N)**, where **N** denotes the size of the input array. Arrays of size **N + 1** are used to store the prefix sum and maximum subarray sum calculated so far. Thus, the overall space complexity is **O(N)**.

## 10. Colourful Knapsack [<https://coding.ninja/P114>]

**Problem Statement:** You are given **N** stones labelled from 1 to **N**. The  $i^{th}$  stone has the weight **W[i]**. There are **M** colors labelled by integers from 1 to **M**. The  $i^{th}$  stone has the color **C[i]**, which is an integer between 1 to **b**, both inclusive.

You are required to fill a knapsack with these stones. The knapsack can hold a total maximum weight of **X**.

You are required to select exactly **M** stones, one of each color such that the sum of the weights of the stones chosen must not exceed **X**. Since you paid a premium for a Knapsack with capacity **X**, you are required to fill the Knapsack as much as possible.

#### **Input Format:**

The **first line** contains three integers **N**, **M**, and **X**, separated by a single space. Where **N** represents the total number of stones, **M** represents the total number of colours available, and **X** represents the total weight of the knapsack.

The **second line** contains **N** integers, **W[1]**, **W[2]**, **W[3]** ..**W[i]**... **W[N]**, separated by a single space.

The **third line** contains **N** integers **C[1]**, **C[2]**, **C[3]** ..**C[i]**... **C[N]**, separated by a single space.

#### **Output Format:**

The output prints the minimum unused capacity of the Knapsack (a single integer). If there is no way to fill the Knapsack, print -1.

#### **Sample Input:**

```
8 3 13
2 3 4 2 4 5 2 3
1 1 1 2 2 2 3 3
```

#### **Sample Output:**

```
1
```

#### **Explanation:**

We can choose the stones as follows:

1. With colour as 1 of weight 4,
2. With colour as 2 of weight 5, and
3. With colour as 3 of weight 3

So we have a total weight  $4 + 5 + 3 = 12$ . Hence the minimum unused capacity is  $13 - 12 = 1$ .

We cannot gain more than 12 with any other combination. Hence the answer is  $13 - 12 = 1$ .

#### **Approach 1: Recursive Approach**

In this approach, we find all the possible combinations using recursion.

### Steps:

1. Create an array of vectors of size  $m + 1$ . Let's call this as **weights**, and add each weight of  $i^{\text{th}}$  color at the index  $i$ .
2. Call a recursive function with color 1 having current weight as 0. The base case would be when we reach the  $(m + 1)^{\text{th}}$  color, which means we have exhausted the colors; hence we return 0.
3. For  $i^{\text{th}}$  color, we have a list of stones. Use the stone if the **current weight + weight** of this stone is less than or equal to  $X$  and call the recursive function with the next color (as we can use only one stone of a particular color).
4. Set new weight as **current weight + weight** of this stone.
5. Initialise answer with a minimum value of integer **INT\_MIN** and update the answer with maximum from the recursive calls.
6. In the main function, return -1 if the **helper function answer** is less than 0, otherwise, return **(X - helper function answer)**. Note that the **helper function** is the recursive function itself, the value it returns is the **answer**.

**Time Complexity:**  $O(2^N)$ , as for every stone we will have two options to either add the stone to the knapsack or not.

**Space Complexity:**  $O(N)$ , as in the worst case, there will be  $N$  recursion calls stored in the call stack.

### Approach 2: Memoization

In this approach, we will optimise the recursive method by using memoization to store the answers to the overlapping subproblems.

### Steps:

1. Create an array of vectors of size  $m + 1$ , let's call it **weights**, and add each weight of  $i^{\text{th}}$  color at the index  $i$ .
2. Create a 2D integer array of size  $(M + 1) * (X + 1)$ . Let's call this **DP**. Fill it with -1.
3. Call a recursive function with color 1 and having current weight as 0. The base case would be when we reach the  $(m + 1)^{\text{th}}$  color, which means that we have exhausted the colors, so return 0. If **dp[color][currentWeight]** is not equal to -1, simply return it.
4. Now for  $i^{\text{th}}$  color, we will have a list of stones. Use the stone if the **current weight + weight** of this stone is less than or equal to  $X$  and call the recursive function with the next color (as we can use only one stone of a particular color).
5. Set the new weight as **current weight + weight** of this stone.
6. Initialise answer with a minimum value of integer **INT\_MIN** and update the answer with maximum from the recursive calls.
7. In the main function, return -1 if the helper function answer is less than 0, otherwise return **(x - helper function answer)**.

**Time Complexity:**  $O(N * X)$ , where **N** denotes the number of stones and **X** is the weight of the knapsack. Since we are traversing the entire matrix of size  $(N*X)$  once, therefore the time complexity is  $O(N * X)$ .

**Space Complexity:**  $O(M * X)$ , where **M** denotes the number of colors and **X** is the weight of the knapsack as we are creating a two-dimensional array of size  $M * X$ .

### Approach 3: Iterative DP

In this approach, we will simply use iteration instead of recursion to solve the problem.

#### Steps:

1. Create an array of vectors of size **m + 1**. Let's call it **weights**, and add each weight of  $i^{th}$  colour at the index *i*.
2. Create a 2D boolean array of size  $(M + 1) * (X + 1)$ . Let's call this **DP**.
3. Initialise **DP[0][0]** as true, as we can always get 0 weight.
4. Run three loops: first for  $i^{th}$  colour (from 1 to **M**), second loop for  $j^{th}$  weight (from 0 to **X**), and third loop to use each stone of  $i^{th}$  colour.
5. If the selected stone of the  $i^{th}$  colour is **W** and this is used at  $j^{th}$  weight, then do the following:  
**if** ( $j - W \geq 0$ )  
     $dp[i][j] = dp[i - 1][j - W]$
6. Initialise the answer with -1.
7. The maximum weight that can be added to the knapsack will be at the rightmost index of **dp[m]**, which has 'True' value. Hence, for this run a loop from 1 to **X**, and whenever **dp[m][i]** is 'True', update the answer with *i*.
8. If the answer is still -1, no answer exists. Otherwise, update the answer with **X - answer** as we have to find the unused capacity.

**Time Complexity:**  $O(N * X)$ , where **N** denotes the number of stones and **X** is the weight of the knapsack.

**Space Complexity:**  $O(M * X)$ , where **M** denotes the number of colors and **X** is the weight of the knapsack. This is because we are creating a two-dimensional array of size  $M * X$ .

# 13. Hashmaps, Circular Queues, and Deques

---

## Hashmaps

HashMap is a data structure where the data or value is stored in the form of keys against which some value is assigned. Keys and values need not be of the same data type. To get a better picture of the key-value pair in a hashmap, consider the following situation:

Suppose we are given a string or a character array, and we are asked to find the maximum occurring character in it. This task could be quickly done using arrays:

We can create an array of size 256, initialise it to zero, and then traverse the given string and increment the count of each character against its ASCII value in the frequency array. In this way, we figure out the maximum occurring character in the given string.

The above method will work fine for all the 256 characters whose ASCII values are known. But what if we want to store the maximum frequency string out of a given array of strings? It can't be done using a simple frequency array, as the strings do not possess any specific identification value like the ASCII values for making the corresponding frequency array. For this, we will be using a different data structure called hashmaps.

If we consider the above example in which we were trying to find the most frequent string among the given array of strings, using hashmaps, the individual strings will be regarded as keys, and the value stored against them will be considered as their respective frequency.

For example, If the given string array is:

```
str[] = {"abc", "def", "ab", "abc", "def", "abc"}
```

The hashmap will look as follows with given **unique strings as keys** and their corresponding **frequencies as values**:

Key (datatype = string)	Value (datatype = int)
"abc"	3
"def"	2

"ab"	1
------	---

From here, we can directly check for the frequency of each string in the given array of strings and hence figure out the most frequent string among them all.

An array limits the indices (keys) of the values to be only whole numbers. Hashmaps successfully overcome this. Here the keys can be non-whole numbers as well.

The values in a hashmap are stored corresponding to their respective keys and can be invoked using those keys. To insert a key-value pair in a hashmap, we can do either of the following:

- `hashmap[key] = value`
- `hashmap.insert(key, value)`

The functions that are required to perform various operations in a hashmap are:

- **insert(k key, v value):** To insert a value **v** against the key **k**.
- **getValue(k key):** To get the value stored against the key **k**.
- **deleteKey(k key):** To delete the key **k**, and hence the value corresponding to it in the hashmap.

**Note:** Here, **v** and **k** represent the generic data types for **value** and **key** variables, respectively.

We will be discussing the implementation of the various operations involving Hash Maps in the practice problems section.

## Ways to implement HashMaps

To implement the hashmaps, we can use the following data structures:

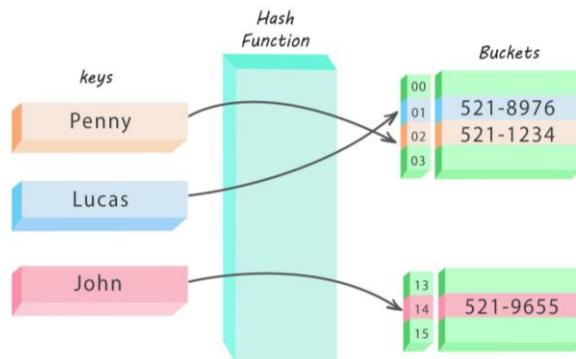
1. **Linked Lists:** To perform insertion, deletion, and search operations in the linked list, the time complexity will be **O(n)** for each as all the three operations require us to traverse the length of the linked list to figure out if the **key** already exists or not in the list. If yes, we perform the required operation; else we return an invalid operation.
2. **Binary Search Trees (BST):** We can use a balanced BST so that the height always remains of the order **O(logN)**. For using a BST, we will need some sort of comparison between the keys. In the case of strings, we can do the same. Hence, insertion, search, and deletion operations are directly proportional to the height of the BST. Thus the time complexity reduces to **O(logN)** for each operation.

**3. Hash tables:** Using a hash table, the time complexity of insertion, deletion, and search operations, could be improved to **O(1)**. We will be studying the implementation of HashMaps using hash tables in further sections.

## Bucket Array and Hash Function

Since we know arrays provide us with one of the fastest ways to extract data as compared to other data structures, as the time complexity of accessing the data in an array is **O(1)**, so we will try to use them in implementing the hashmaps.

Now, we want to store the **key-value** pairs in an array named **bucket array**. For this, we need an integer corresponding to the **key** so that it could be mapped to an index in the bucket array. To do so, we use a **hash function**. A hash function converts the **key** into an integer, which acts as the index for storing the **value** corresponding to that **key** in the bucket array. Refer to the given image for more clarity.



In the above figure: When we pass the key **Penny** through the hash function, it returns us the index or hash value as 02, similarly, for the keys **Lucas** and **John**, we get the hash values like 01 and 14, respectively.

Now, let us consider a situation.

Suppose we want to store a string in a hash table. After passing the string through the hash function, the integer we obtain is equal to 10593, but the bucket array's size is only 20. So, we can't store that string in the array as the value 10593 cannot be used as an index in the array of size 20.

To overcome this problem, we divide the implementation of hashmap into two parts:

- Hash code
- Compression function

The first step to store a value into the bucket array is to convert the **key** into an integer (this could be any integer irrespective of the size of the bucket array). This part is achieved by using a **hash code**. For different types of **keys**, we can have different kinds of hash codes.

Now we will pass this value through a compression function, which will convert that value within the range of our bucket array's size. Now, we can directly store that key at the index obtained after passing through the compression function.

The compression function can be used as (**%bucket\_size**). Since the modulo function always allows the index to stay within the bounds of the **bucket\_size**.

One example of a hash code could be: (Example input: "abcd")

$$\text{"abcd"} = (\text{'a'} * p^3) + (\text{'b'} * p^2) + (\text{'c'} * p^1) + (\text{'d'} * p^0)$$

**Where p is generally taken as a prime number so that they are well distributed.**

But, there is still a possibility that after passing the key through the hash code, when we pass the integer obtained through the compression function, we can get the same values of indices.

For example, let s1 = "ab" and s2 = "cd". Now using the above hash function for p = 2, we get , h1 = 292 and h2 = 298. Let the bucket size be equal to 2. Now, if we pass the hash codes obtained through the compression function, we will get:

$$\text{Compression\_function1} = 292 \% 2 = 0$$

$$\text{Compression\_function2} = 298 \% 2 = 0$$

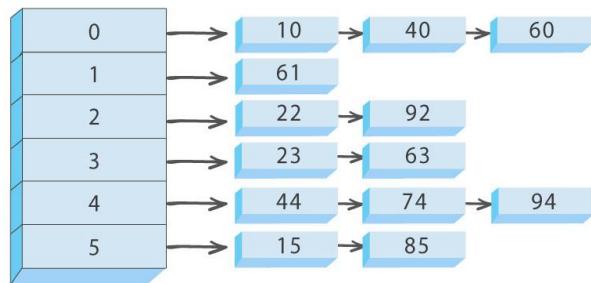
This means they both lead to the same index 0. This is known as a collision.

## Collision Handling

We can handle collisions in two ways:

- Closed hashing (or closed addressing)
- Open addressing

In **closed hashing**, each entry of the array will be a linked list. This means it should be able to store every possible value that corresponds to this index. The array position holds the address to the head of the linked list, and we can traverse the linked list by using the head pointer for the same and add a new element at the end of that linked list. This is also known as **separate chaining**.



On the other hand, in **open addressing**, we check for the index in the bucket array if it is empty or not. If it is empty, then we directly insert the **key-value** pair at that index. If not, then will we find an alternate position for the same.

To find the alternate position, we can use the following:

$$h_i(a) = h_f(a) + f(i)$$

where **hf(a)** is the original hash function, and **f(i)** is the **i<sup>th</sup>** try over the hash function to obtain the final position **h<sub>i</sub>(a)**.

To find out **f(i)**, we could use any of the following techniques:

- 1. Linear probing:** In this method, we linearly probe to the next slot until we find an empty index. Thus here, **f(i) = i**.
- 2. Quadratic probing:** As the name suggests, we will look for alternate **i<sup>2</sup>** positions ahead of the filled ones, that is, **f(i) = i<sup>2</sup>**.
- 3. Double hashing:** According to this method, **f(i) = i \* H(a)**, where **H(a)** is some other hash function.

In practice, we generally prefer to use **separate chaining** over **open addressing**, as it is easier to implement and is even more efficient.

## Advantages of HashMaps

- Hash functions help us have fast random access to the memory.
- HashMaps can use negative and non-integral values as keys to access the corresponding values stored in them; similarly unlike in arrays which use just whole numbers as indexes to access the corresponding values.
- All the keys stored in the hashmap can be easily accessed by iterating over the map.

## Disadvantages of HashMaps

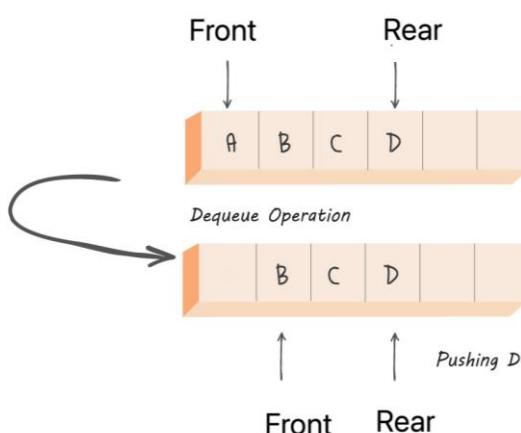
- Collisions can cause large penalties and increase the time complexity to linear.
- When the number of keys is large, a single hash function often causes collisions.

## Applications of HashMaps

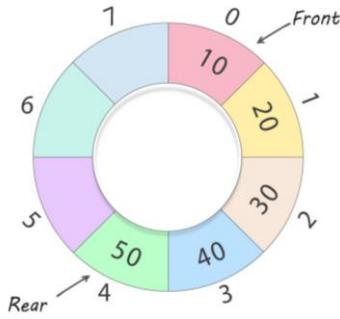
- HashMaps have applications in the implementation of cache where memory locations are mapped to small sets.
- They are used to index tuples in database management systems.
- They are also used in algorithms such as the Rabin-Karp pattern matching algorithm.

## Circular Queues

We know that while performing the dequeue operation on queues, a lot of space gets wasted, as whenever an element is deleted from the front, the front pointer starts pointing to the next index. With every deletion, the front spaces keep getting wasted. Therefore, a simple array implementation for a queue is not memory efficient.



To solve this problem, we assume the arrays as circular arrays. With this representation, if we have any free slots at the beginning, the rear pointer can go to its next free slot.



## How can circular queues be implemented?

Queues can be implemented using arrays and linked lists. The basic algorithm for implementing a circular queue remains the same.

We maintain three variables for all the operations: **front**, **rear**, and **curSize**.

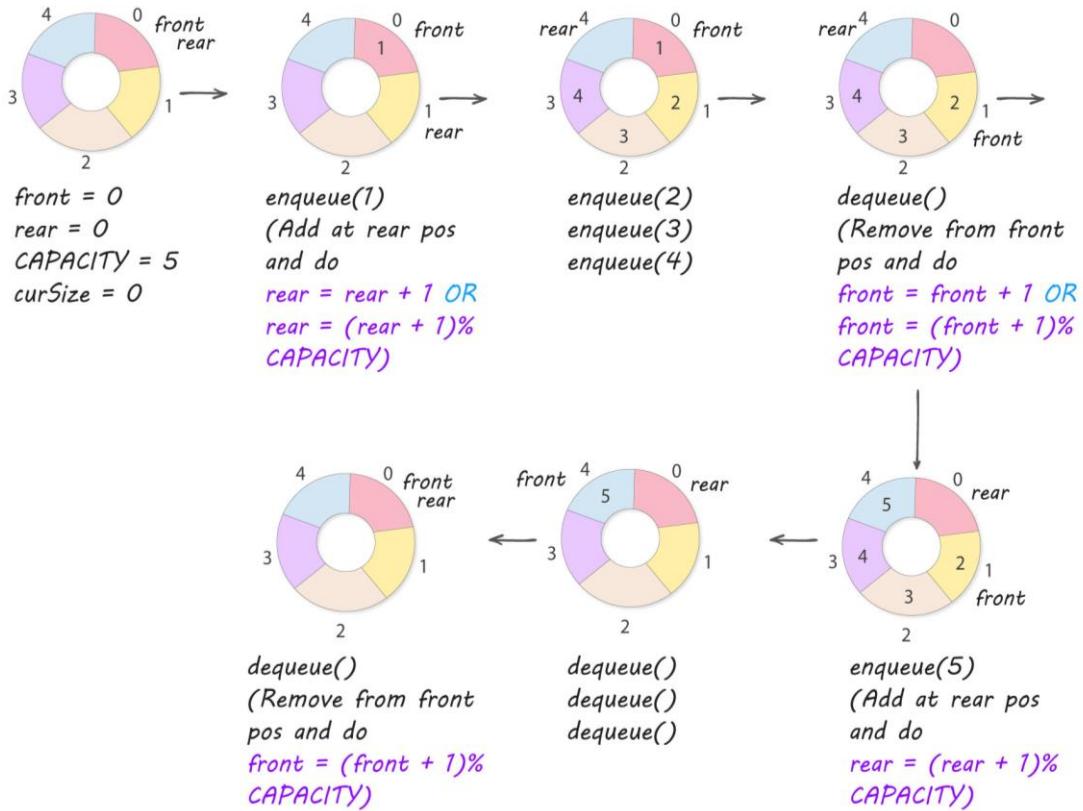
**CAPACITY** of the queue is the size passed during the initialization of the array.

For the **front** and **rear** variables to always remain within the valid bounds of indexing, we update :

**front** as **(front + 1) % CAPACITY** and

**rear** as **(rear + 1) % CAPACITY**.

This allows **front** and **rear** to never result in an index which gives an out of bounds exception.



Notice in figure 5 we cannot do  $\text{rear} = \text{rear} + 1$  as it will result in an index out of bound exception, but there is an empty position at index 0, so we do  $\text{rear} = (\text{rear} + 1) \% \text{CAPACITY}$ . Similarly, in figure 7, we cannot do  $\text{front} = \text{front} + 1$  as it will also give an exception therefore we do  $\text{front} = (\text{front} + 1) \% \text{CAPACITY}$ .

The pseudo-codes for the various operations in circular queues are as follows :

- **Enqueue Operation :**

```
function enqueue(data)
    // curSize == CAPACITY when queue is full
    if queue is full
        return "Full Queue Exception"
    curSize++
    queue[rear] = data
    // updating rear variable to the next empty position in the circular queue while
```

**keeping it within the bounds of the array**  
**rear = (rear + 1) % CAPACITY**

- **Dequeue Operation :**

```
function dequeue()

    // front == rear OR curSize == 0 when queue is empty
    if queue is empty
        return "Empty Queue Exception"

    curSize--
    temp = queue[front]
    front = (front + 1) % CAPACITY
    return temp
```

- **getFront Operation**

```
function getFront()

    // front == rear OR curSize == 0 when queue is empty
    if queue is empty
        return "Empty Queue Exception"

    temp = queue[front]
    return temp
```

## Time Complexity of Various Operations

Let '**n**' be the number of elements in the circular queue. The complexities of various operations with this representation is as follows:

Operations	Time Complexity
enqueue(data)	O(1)
dequeue()	O(1)
getFront()	O(1)
boolean isEmpty()	O(1)
int size()	O(1)

## Advantages of Circular Queues

- All operations in a circular queue occur in **constant time**.
- **None of the operations requires** reorganising/copying data.

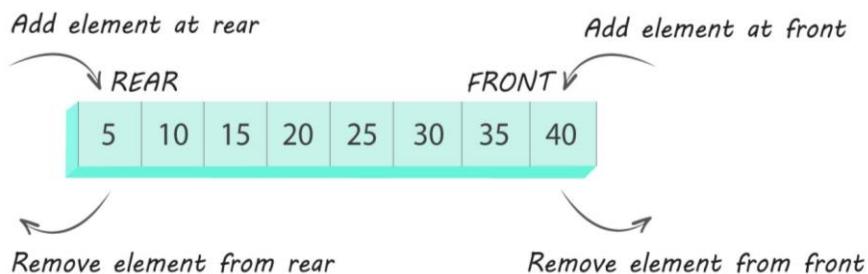
- Circular queues are **memory efficient**.

## Application of Circular Queues

- They are used in the **looped execution** of slides of a presentation.
- They are used in browsing through the open windows applications (**alt + tab** in Microsoft Windows)
- It is used for the **Round-Robin execution** of jobs in multiprogramming OS.
- They are also used in **page replacement algorithms**—a circular list of pages is maintained, and when a page needs to be replaced, the page in the front of the queue is chosen.

## Deques

A deque, also known as the **double-ended queue**, is an ordered list in which elements can be inserted or deleted at either of the ends. It is also known as a **head-tail linked list** because elements can be added or removed from either the front (head) or the back (tail) of the list.



## Properties of Deques

- Deques can be used both as **stack** and **queue** to allow insertion and deletion of elements from both ends.
- Deques do not require **LIFO** and **FIFO** orderings enforced by data structures like stacks and queues.
- There are two variants of double-ended queues:
  - **Input restricted deque:** In this deque, insertions can only be done at one end, while deletions can be done from either end.
  - **Output restricted deque:** In this deque, deletions can only be done at one end, while insertions can be done on either end.

## Operations on Deques

- **enqueue\_front(data):** Insert an element at the front end.
- **enqueue\_rear(data):** Insert an element at the rear end.
- **dequeue\_front():** Delete an element from the front end.
- **dequeue\_rear():** Delete an element from the rear end.
- **front():** Return the front element of the dequeue.
- **rear():** Return the rear element of the dequeue.
- **isEmpty():** Returns true if the deque is empty.
- **isFull():** Returns true if the deque is full.

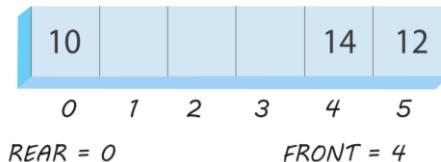
## Implementation of Deques

Deques can be implemented using data structures like **circular arrays** or **doubly-linked lists**.

Below is the circular array implementation for deques; the same approach can be used to implement deque using doubly-linked lists.

We maintain two variables: **front** and **rear**, front represents the **front end** of the deque, and rear represents the **rear end** of the deque.

The circular array is represented as "**carr**", and its size is represented by **size**, having elements indexed from **0** to **size - 1**.



Here in the above image, the array **carr** has a **size** equal to **6**, and the **front** and **rear** variables are pointing at index 4 and 0, respectively.

- **Inserting** an element at the **front** end involves **decrementing** the front pointer.
- **Deleting** an element from the **front** end involves **incrementing** the front pointer.
- **Inserting** an element at the **rear** end involves **incrementing** the rear pointer.
- **Deleting** an element from the **rear** end involves **decrementing** the rear pointer.

The **front** and **rear** pointers need to be maintained such that they remain within the bounds of indexing of **0** to **size - 1** of the circular array.

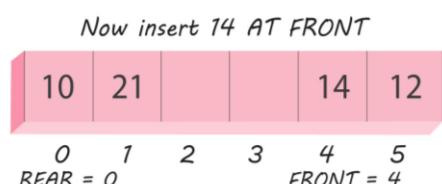
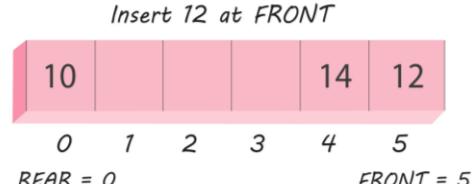
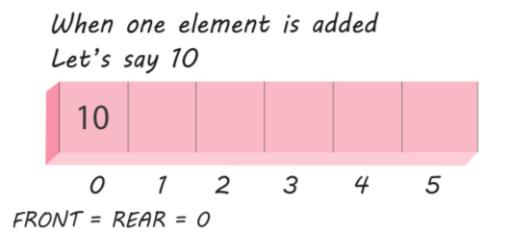
Initially, the deque is empty, so front and rear pointers are initialised to **-1**, denoting that the deque contains no element.

## **Implementation of various operations in Deque:**

## Enqueue\_front operation

## **Steps:**

1. If the array is full, the data can't be inserted.
  2. If **front** is equal to -1 then set **front = size- 1** and **arr[front] = data.**
  3. Else decrement **front** and set **carr[front]** as data.



```
function enqueue_front(data)

    // Check if deque is full
    if (front equals 0 and rear equals size - 1) or (front equals rear + 1)
        print ("Overflow")
        return

    /*
        Check if the deque is empty; insertion of an element from the front or rear
        will be equivalent.
        So update both front and rear to 0.
    */

    if front equals -1
        front = 0
        rear = 0
        carr[front] = data
        return
```

```

// Otherwise check if the front is 0
if front equals 0
/*
    Updating front to size -1 so that front remains within the bounds of
    the circular array.
*/
front = size - 1
else
    front = front - 1

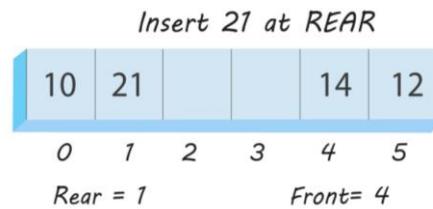
carr[front] = data
return

```

### Enqueue\_rear operation

#### Steps:

1. If the array is already full, then it is not possible to insert more elements.
2. If there is no element in the deque—that is, **rear** is equal to -1, increase **front** and **rear** and set **carr[rear]** as data.
3. Else increment rear and set **carr[rear]** as data.



```

function enqueue_rear(data)

    // Check if deque is full
    if (front equals 0 and rear equals size - 1) or (front equals rear + 1)
        print ("Overflow")
        return

    /*
        Check if the deque is empty; insertion of an element from the front or rear
        will be equivalent.
        So update both front and rear to 0.
    */

    if front equals -1
        front = 0
        rear = 0
        carr[front] = data
        return

```

```

// Otherwise, check if rear equals size -1

if rear equals size -1
/*
    Updating rear to 0 so that rear remains within the bounds of the
    circular array.
*/
rear = 0
else
    rear = rear + 1

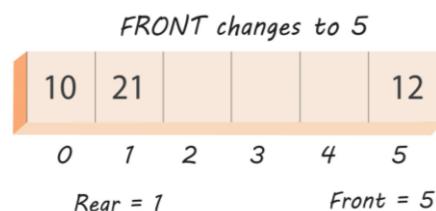
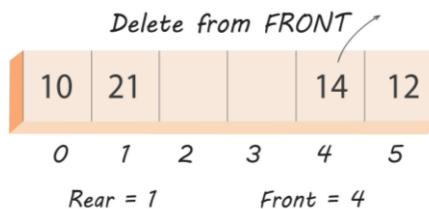
carr[rear] = data
return

```

### Dequeue\_front operation

#### Steps:

1. If the deque is empty, return.
2. If there is only one element in the deque—that is, **front** equals **rear**, set **front** and **rear** as -1.
3. Else increment **front** by 1.



```

function dequeue_front()

// Check if deque is empty
if front equals -1
    print ("Underflow")
    return

/*
Otherwise, check if the deque has a single element, i.e. front and rear are
equal and non-negative as the deque is non-empty.

```

```

*/
if front equals the rear
/*
    Update front and rear back to -1, as the deque becomes empty.
*/
front = -1
rear = -1
return

// If the deque contains at least 2 elements.

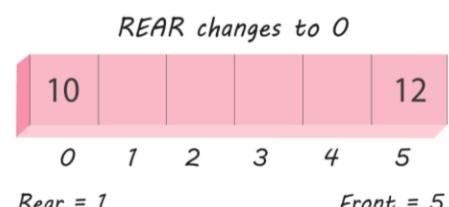
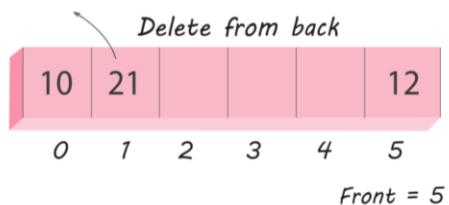
if front equals size - 1
    // Bring front back to the start of the circular array.
    front = 0
else
    front = front + 1
return

```

### Dequeue\_rear operation

Steps:

1. If the deque is empty, return.
2. If there's just one element in the Deque—that is, **rear** equals **front**, set **front** and **rear** as **-1**.
3. Else decrement **rear** by one.



```
function dequeue_rear()
```

```

// Check if deque is empty
if front equals -1
    print ("Underflow")
    return

/*

```

```

Otherwise, check if the deque has a single element.; else i.e. front and rear are
equal and non-negative as the deque is non-empty.

*/
if front equals the rear
/*
    Update front and rear back to -1, as the deque becomes empty.
*/
front = -1
rear = -1
return

// If the deque contains at least 2 elements.

if rear equals 0
    // Bring rear back to the last index i.e. size - 1 of the circular array.
    rear = size - 1
else
    rear = rear - 1
return

```

## Front operation

### Steps:

1. If the deque is empty, return.
2. Else return **carr[front]**.

```

function front()

// Check if deque is empty
if front equals -1
    print ("Deque is empty")
    return

// Otherwise return the element present at the front end
return carr[front]

```

## Rear operation

### Steps:

1. If the deque is empty, return.
2. Else return **carr[rear]**.

```

function rear()

// Check if deque is empty
if front equals -1
    print ("Deque is empty")

```

```
    return  
  
    // Otherwise return the element present at the rear end  
    return carr[rear]
```

### IsEmpty operation

#### Steps:

If front equals -1, the deque is empty, else it's not.

```
function isEmpty()  
  
    // Check if front is -1 i.e. no elements are present in deque.  
    if front equals -1  
        return true  
    else  
        return false
```

### IsFull operation

#### Steps:

If front equals 0 and rear equals size -1, or front equals rear + 1, then the deque is full.

Otherwise, it's not. Here “**size**” is the size of the circular array.

```
function isFull()  
  
/*  
     Check if the front is 0 and rear is size -1 or front == rear + 1; in both cases, we  
     cannot move front and rear to perform any insertions.,  
*/  
  
if (front equals 0 and rear equals size - 1) or (front equals rear + 1)  
    return true  
else  
    return false
```

## Time Complexity of Various Operations

Let ‘n’ be the number of elements in the deque. The time complexities of various deque operations in the worst case are given as:

Operations	Time Complexity
Enqueue_front(data)	O(1)
Enqueue_rear(data)	O(1)
Dequeue_front()	O(1)
Dequeue_rear()	O(1)
Front()	O(1)
Rear()	O(1)
isEmpty()	O(1)
isFull()	O(1)

## Advantages of Deques

- Deques are powerful data structures that offer the functionalities of both stacks and queues.
- They are highly useful for operations that involve the addition/deletion of elements from either end, as all operations can be performed in constant **O(1)** time.

## Applications of Deques

- Since deques can be used as stacks and queues, they can be used to perform **undo-redo** operations in software applications.
  - Deques are used in the **A-steal job scheduling algorithm** that implements task scheduling for multiple processors (multiprocessor scheduling).
- 

## Practice Problems

### 1. Implementation: Hashmap [<https://coding.ninja/P115>]

**Problem Statement:** You are required to design a data structure that stores a mapping of a key to a given value and supports the following operations in constant time.

1. **insert(key, value):** Inserts an integer value to the data structure against a string type key if not already present. If the key is already present, it updates the value of the key with the new one.  
This function will not return anything.
2. **delete(key):** Removes the key from the data structure if present. It doesn't return anything.
3. **search(key):** It searches for the key in the data structure. If the key is present, it returns true. Otherwise, it returns false.
4. **get(key):** It returns the integer value stored against the given key. If the key is not present, return -1.

5. **getSize()**: It returns an integer value denoting the size of the data structure, which represents the number of key-value pairs already present in the data structure.
6. **isEmpty()**: It returns a boolean value, denoting whether the data structure is empty or not.

**Note:**

1. **Key** is always a string value.
2. **Value** corresponding to any **key** can never be -1.

**Input Format:**

The **first line** contains an integer **N** which denotes the number of operations to be performed on the data structure.

Then the operations follow.

**N lines** follow where every line represents an operation that needs to be performed.

For **insert** operation, the input line will have three input values separated by a single space—the type of the operation in integer, the **key** to be inserted as a string, and the **value** against the key as an integer, respectively.

For the **delete(key)**, **search(key)**, and **get(key)** operations, the input line will have two values separated by a single space—the type of the operation in integer and the **key** to be inserted.

For the **getSize() and isEmpty() operations**, the input will contain a single integer, denoting only the type of the operation in integer.

**Output Format:**

For 1st operation (**insert(key, value)**), you need not return anything.

For 2nd operation (**delete(key)**), remove the element from the data structure and don't return anything.

For the 3rd operation (**search(key)**), return **true** if the **key** is present in the data structure. Else, return **false**.

For the 4th operation (**get(key)**), return the value stored against the **key**. If the **key** is not present, return -1.

For 5th operation (**getSize()**), return the size of the data structure.

For the 6th operation (**isEmpty()**), return the boolean denoting whether the data structure is empty or not.

**Sample Input:**

```
6
1 qwerty 35
1 qwerty 50
2 cd
5
```

2 qwerty

2 abcde

#### Sample Output:

1

#### Explanation

The first query inserts the key '**qwerty**' in the hashmap with the value **35**.

The second query attempts to insert '**qwerty**' again in the hashmap. But since it is already present, so the value is updated to **50**

The third query asks to delete the key '**cd**' from the hashmap. Since the key is not present in the hashmap hence nothing is returned.

The fourth query asks the size of the hashmap, which is 1 as it only contains single key-value pair **{'qwerty': 50}**

The fifth query asks us to delete the word '**qwerty**' from the hashmap. Now our hashmap becomes empty.

The sixth query asks us to delete the word '**abcde**' from the hashmap. Since our hashmap is empty, we simply do nothing.

#### Approach: Implementation of HashMap using Separate Chaining :

Here hashmap is implemented using arrays, and to avoid a collision, the technique of separate chaining is used, which is explained as below:

- HashMap is the data structure used to store **key-value** pairs, where the average retrieval time for the operations like **get()** and **insert()** is constant, i.e. **O(1)**.
- Hashmap uses an array of Singly Linked List internally for its implementation.
- The array is called the **buckets**, and every bucket(i-th index of the array) holds a pointer to the head of the Singly Linked List.
- Every Linked List Node has fields called the '**key**', '**value**', and certainly the pointer to the next node.
- For every entry of a key, we try to generate a '**unique value**' pointing to the bucket where it can be stored. It can also be said that we try to find an index against every key that needs to be inserted or updated in the array/buckets.

**Note:** There might be a scenario where the unique value which we want against each **key** has a **collision**. That means two or more unique keys may have the same unique value/index that we are trying to calculate.

The first essential step is calculating **the index in the array/bucket** where the node can be placed.

As discussed earlier, we can encounter collisions while calculating the indices of the strings. The collision is when two or more elements have the same index or need to be put in the same bucket.

To make a hashmap work even in collision cases, we keep a singly linked list in every bucket. Let us suppose that the two keys ('firstKeyValue' and 'secondKeyValue') to be inserted in the hashmap have the same index—that is, the hash value, as two; then we could visualise the list at the second bucket as:

head → 'secondKeyValue' → 'firstKeyValue'.

So while updating the **key**, we first calculate the index, then iterate over the LinkedList at that index to get the item. To calculate the index against every **key** that needs to be inserted, we use **hashing algorithms**. The values thus computed are called **Hash Values**. This hash value serves as the location or index of the bucket.

### Steps to implement insert(key, value):

1. As discussed, first compute the **hash value or location** in the buckets where the **key** needs to be inserted using any of the hashing algorithms available. In case the location doesn't have a list already, we make it as the head, or if the location already contains a list (case of collision), we insert the new **key** at the head.
2. We define **load factor**, which is the total load being put on each of the buckets. We try to keep it less than or equal to 0.7 to maintain the operations performed are constant.  
$$loadFactor = (size * 1.0) / buckets.size()$$
3. To ensure that every insert operation happens in constant time, we keep a threshold of 0.7 (less than 1) over the load factor. If the load factor exceeds the threshold value, we run a **rehash function**.
4. **rehash():** This function basically increases the total bucket size either by the multiple of 2 or by the power of 2, and every key present is inserted into the updated buckets.

### Steps to implement delete(key):

1. Generate the **hash value** against the **key** to be deleted and thus get the bucket corresponding to the **key** by taking modulo with the total bucket size.
2. If the required **key** is present, then delete the **key-value** pair from the data structure; otherwise, do nothing.

### Steps to implement search(key):

1. Generate the **hash value** against the **key** to be searched and get the bucket corresponding to the **key** by taking modulo with the total bucket size.
2. If the required **key** is present, then return **true**, else return **false**.

### Steps to implement get(key):

1. Generate the **hash value** against the **key** to be searched and get the bucket corresponding to the **key** by taking modulo with the total bucket size.
2. If the key is present, then return the value stored against the key in the hashmap, else return -1.

### Steps to implement getSize():

Return the size of the hashmap, which is equal to the number of `<key, value>` pairs in the map.

#### **Steps to implement `isEmpty()`:**

Return `true` if the size of the hashmap is zero, else return `false`.

**Time Complexity:**  $O(1)$  as all the functions take constant time.

**Space Complexity:**  $O(N)$  to store all the elements.

## **2. Longest Subarray Zero Sum** [<https://coding.ninja/P116>]

**Problem Statement:** You are given an array `arr` of length `N` consisting of positive and negative integers; you are required to return the length of the longest subarray whose sum is zero.

#### **Input Format:**

The first line of input contains an integer `N`, denoting the length of the array.

The second line of input contains `N` single space-separated integers, denoting the elements of the array.

#### **Output Format:**

The single line contains an integer, the length of the longest subarray in the given array whose sum is zero.

#### **Sample Input:**

```
5
3 -2 1 -2 1
```

#### **Sample Output:**

```
4
```

#### **Explanation:**

In the given input array, there are two subarrays whose sum is equal to zero:

- Subarray 1 from index 0 to 3—that is, `[3, -2, 1, -2]`.
- Subarray 2 from index 2 to 4—that is, `[1, -2, 1]`.

The longest subarray of these is of length 4, and hence the answer is **4**.

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

In this approach, we will calculate the sum for all possible subarrays of the given input array.

#### **Steps:**

1. For each element ( $i^{\text{th}}$ , where  $0 \leq i < N$ ) in the array, fix it as the first end of the subarray.
2. Initialise the subarray sum with zero, and iterate on the remaining elements ( $j^{\text{th}}$ , where  $i \leq j < N$ ) of the array.

3. Keep adding the current element—that is,  $\text{arr}[j]$ , to the subarray sum and fix it as the second end of the array.
4. If the sum becomes equals to zero, then update the largest length.

**Time Complexity:**  $O(N^2)$ , where  $N$  denotes the total number of integers in the array. In the worst case, we will be checking for all possible subarrays. As there will be  $N^2$  number of subarrays possible for a given array of length  $N$ , the time complexity will be of the order of  $N^2$ .

**Space Complexity:**  $O(1)$ , since constant extra space is required.

### Approach 2: Using Hashmap

#### Steps:

1. Use a hashmap to store the **prefix sums** as **keys** and the **value** as the **lowest index** till which the prefix sum occurred.
2. If any prefix sum occurs again in the array (let's say  $i$  is the first index and  $j$  is the new array index at which the prefix sum occurs), then the sum of elements from  $(i + 1)^{\text{th}}$  to  $j^{\text{th}}$  index in the array equals to zero.
3. Update the largest length.

**Note:** We will only keep just the first index (lowest index) of a prefix sum as we need to find the largest length.

**Time Complexity:**  $O(N)$ , where  $N$  denotes the number of integers in the array. In the worst case, we will be iterating on each element only once, and hence the time complexity is  $O(N)$ .

**Space Complexity:**  $O(N)$ , where  $N$  denotes the number of integers in the array. In the worst case, we will be storing the prefix sums for all indices in the map, and hence the space complexity is  $O(N)$ .

## 3. Bursting Balloons [<https://coding.ninja/P117>]

**Problem Statement:** There are  $N$  balloons lined up. Your aim is to destroy all these balloons. Now, a balloon can only be destroyed if the player shoots its head. To do this, one needs to shoot an arrow from the left to the right side of the platform from a chosen height.

The arrow moves from left to right, at a chosen height  $H$ , until it finds a balloon. The moment when an arrow touches a balloon, the balloon gets destroyed and disappears, and the arrow continues towards the right at a height decreased by one. If the arrow was moving at height  $H$ , after destroying the balloon, it travels at height  $H - 1$ . The player wins this game if he destroys all the balloons in minimum arrows.

You are given an array of **N** integers. Each integer represents the height of a balloon. You have to print the minimum arrows required to complete the task.

#### **Input Format:**

The **first line** of input contains an integer **N** representing the size of the array.

The **second line** of input contains **N** space-separated integers representing the height of the balloons.

#### **Output Format:**

Print the minimum arrows required to complete the task.

#### **Sample Input:**

5

2 1 5 4 3

#### **Sample Output:**

2

#### **Explanation:**

To shoot all the balloons, we can release two arrows—at the height of 2 and another at the height of 5.

The arrow shot for height **2** destroys balloons at the heights **[2, 1]**.

The arrow shot for height **5** destroys balloons at the heights **[5, 4, 3]**.

Therefore we require a minimum of **2** arrows to shoot all the balloons.

#### **Using Hashmaps:**

We can see here that balloons need to be destroyed in a minimum number of arrows, so we need to reuse the arrows. Also, we need a new arrow for every balloon, which is not getting destroyed by the previous arrows.

We can solve this problem greedily by adding a new arrow for every balloon, which is not getting destroyed by any of the previously shot arrows.

#### **Steps:**

1. Use a map data structure that will store the height of fired arrows.
2. Start a loop from  $i = 0$  to  $i = N - 1$ , and for the  $i^{\text{th}}$  balloon, if the map contains the value **Arr[i]**, then it means this balloon will get hit by some previous arrow.
3. Therefore decrease the count of **Arr[i]** in the map and if it becomes 0, remove the key from the map.
4. Increase the count of **Arr[i] - 1** in the map because there is an arrow at this height available.
5. If the map does not contain the **Arr[i]**, then fire a new arrow so increase the count of **Arr[i]** in the map.
6. Find the sum of all the arrows present in the map and return it.

**Time Complexity:**  $O(N)$ , where  $N$  denotes the number of balloons. As we are visiting every balloon only once and we are also searching and accessing the hashmap  $N$  times, the overall time complexity is  $O(N)$ .

**Space complexity:**  $O(N)$ , where  $N$  denotes the number of balloons. As we are storing the heights of balloons in the hashmap and its size can at max grow up to  $O(N)$ .

## 4. Longest Harmonious Subsequence [<https://coding.ninja/P118>]

**Problem Statement:** You are given an array '**ARR**' of  $N$  integers. Your task is to find the longest harmonious subsequence of this array.

A sequence is said to be a harmonious sequence if the difference between the maximum and the minimum of all the elements of that sequence is exactly 1.

### **Input Format:**

The **first line** of input contains an integer  $T$ , denoting the number of test cases.

Then the test cases follow.

The **first line** of each test case contains an integer  $N$ , which denotes the number of integers in the array **ARR**.

The **second line** of each test case contains  $N$  integers of the array **ARR**, separated by space.

### **Output Format :**

For each test case, print the size of the longest harmonious subsequence.

Print the output of each test case in a new line.

### **Sample Input:**

1

4

1 2 2 1

### **Sample Output:**

4

### **Explanation:**

The given array is [1, 2, 2, 1].

If we take the complete array, then the maximum of all the elements is 2, and the minimum of all the elements is 1.

So, the difference between the maximum and the minimum =  $2 - 1 = 1$ .

Hence, the longest harmonious subsequence is [1, 2, 2, 1], and its length is 4.

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

The idea here is to generate all the possible subsequences of the given input array using the bit masking technique. For each subsequence, check if it is a harmonious subsequence or not. Take the maximum length of all the possible found harmonious subsequences.

#### Steps:

1. Initialise **answer** to 0, which denotes the length of the longest harmonious subsequence in the given array.
2. Iterate from  $i = 0$  to  $2^N - 1$ , where **N** is the length of the given array, **arr**.
  - a. Initialise three integer variables **minValue**, **maxValue**, and **countLength** with **INT\_MAX**, **INT\_MIN**, and **0**, respectively, which stores the minimum and maximum values in the current subsequence and the length of the current subsequence, respectively.
  - b. Iterate from  $j = 0$  to **N - 1**:
    - i. If the  $j^{th}$  bit of  $i$  is set and **minValue** is less than **arr[j]**, then update **minValue** to **arr[j]**.
    - ii. If the  $j^{th}$  bit of  $i$  is set and **maxValue** is greater than **arr[j]**, then update **maxValue** to **arr[j]**.
    - iii. Increment **countLength** as we are including the current element in the subsequence.
  - c. If the difference between the **maxValue** and **minValue** is 1. It means it is a valid harmonious subsequence—update **answer** to a maximum of **answer** and **countLength**.
  - d. Return **answer** as the final answer.

**Time Complexity:**  $O(N * 2^N)$ , where **N** denotes the length of the given array **arr**. We generate the binary values from 0 to  $2^N - 1$  and then iterating through the array of length **N**. Hence the overall time complexity is  $O(N * 2^N)$ .

**Space Complexity:**  $O(1)$ , as we are using constant space.

#### Approach 2: Optimised approach

The idea here is to pivot each element of the array and then take only those elements of the array which are equal or have a **difference** = 1 so that the maximum difference of elements in the chosen subsequence is 1.

#### Steps:

1. Initialise **answer** to 0, which denotes the length of the longest harmonic subsequence in the given array.
2. Iterate from  $i = 0$  to **N - 1**:
  - a. Initialise **countLength** with 0, which stores the length of the current subsequence.
  - b. Initialise a boolean **flag** to **false**, which will be **true** if it is possible to make the harmonic subsequence. Otherwise, it will be **false**.
  - c. Iterate from  $j = 0$  to **N - 1**:

- i. If the element at the  $j^{th}$  index is the same as the element at the  $i^{th}$  index, then increment **countLength**.
  - ii. If the element at the  $j^{th}$  index is 1 greater than the element at the  $i^{th}$  index, then increment **countLength** and set the **flag** to **true** as the maximum difference of the elements in the current subsequence is 1.
  - d. If the **flag** is **true**, Update the **answer** to the maximum of **answer** and **currentLength**.
3. Return **answer** as the final answer.

**Time Complexity:**  $O(N^2)$ , where **N** denotes the size of the given array. We are pivoting each element of the array, and then for each pivot element, we are iterating through the complete array once, which will take  $O(N)$  time. Hence, the overall time complexity is  $O(N^2)$ .

**Space Complexity:**  $O(1)$ , as we are using constant space.

### Approach 3: Using Hashmaps

The idea here is to store the frequency of each element in a HashMap. In each iteration, we will check two things:

1. If  $\text{arr}[i] + 1$  is present in the HashMap or not. If present, then our current length of the subsequence will be the number of occurrences of  $\text{arr}[i]$  + number of occurrences of  $(\text{arr}[i] + 1)$ .
2. If  $\text{arr}[i] - 1$  is present in the HashMap or not. If present, then our current length of the subsequence will be a number of occurrences of  $\text{arr}[i]$  + number of occurrences of  $(\text{arr}[i] - 1)$ .

### Steps:

1. Initialise **answer** to 0, which denotes the longest of the length harmonic subsequence in the given array.
2. Define a HashMap **frequency** that stores the number of occurrences of each element in the array.
3. Iterate from  $i = 0$  to  $\mathbf{N} - 1$ :
  - a. Increment the count of  $\text{arr}[i]$  in the HashMap.
  - b. If  $\text{arr}[i] + 1$  is present in the HashMap, that is,  $\text{frequency}[\text{arr}[i] + 1] > 0$ , then update the **answer** to the maximum of **answer** and  $(\text{frequency}[\text{arr}[i]] + \text{frequency}[\text{arr}[i] + 1])$ .
  - c. If  $\text{arr}[i] - 1$  is present in the HashMap, that is,  $\text{frequency}[\text{arr}[i] - 1] > 0$ , then update the **answer** to the maximum of the **answer** and  $(\text{frequency}[\text{arr}[i]] + \text{frequency}[\text{arr}[i] - 1])$ .
4. Return **answer**.

**Time Complexity:**  $O(N)$ , where **N** denotes the size of the given array. We are iterating over the array and incrementing the count of every element in the HashMap, which has amortised time complexity of  $O(1)$ . Hence, the overall time complexity is  $O(N)$ .

**Space Complexity:**  $O(N)$ , where  $N$  denotes the size of the given array. As the maximum size of the HashMap can grow up to the size of the given array. Hence, the overall space complexity is  $O(N)$ .

## 5. Predict the Winner [<https://coding.ninja/P119>]

**Problem Statement:**  $N$  people are standing in a circle. There is a number written on the back of every person's shirt. One person among them has a ball in his hands, and the number written on his shirt is 1. The number written on the shirt of every other person is 1 more than the number written on the shirt of the person standing to the left of him. We have also been provided with an integer  $K$  denoting the jump parameter. They start playing a game. The game proceeds as follows:

1. If only one person is remaining in the game, then the game stops immediately, and the person who is left is declared as the winner.
2. The player who currently has the ball in his hand passes the ball to the person standing to his right.
3. Step 2 happens exactly  $K - 1$  times.
4. The game pauses here, and the person who has the ball in his hand currently passes the ball to the person standing on his right and has to leave the game immediately.
5. Again the game resumes with the remaining players.

You have been provided with positive integers  $N$  and  $K$ . Your task is to find the number written on the back of the winner when the game is played with  $N$  members using the jump parameter  $K$ .

### **Input Format:**

The **first line** of input contains an integer  $T$  denoting the number of test cases.

The **next  $2 * T$  lines** represent the  $T$  test cases.

The **first line** of each test case contains an integer  $N$  denoting the number of players in the game.

The **second line** of each test case contains an integer  $K$  denoting the jump parameter in the game.

### **Output Format:**

For each test case, print the number written on the back of the winner when the game is played with  $N$  members using the jump parameter  $K$ .

### **Sample Input:**

```
1
4
3
```

### **Sample Output:**

### **Explanation:**

- The game starts with Person 1. He passes the ball to person 2. Person 2 passes the ball to Person 3. The game pauses here. Person 3 passes the ball to Person 4 and leaves the game.
- Then the game resumes. Person 4 passes the ball to person 1. Person 1 passes the ball to person 2. The game pauses here. Person 2 passes the ball to person 4 and leaves the game.
- Then the game resumes with Person 4 having the ball in his hand. He passes the ball to person 1. Person 1 passes the ball to Person 4. The game pauses here. Person 4 passes the ball to person 1 and leaves the game.
- Now only person 1 remains, and the game stops here.

Hence, the winner of the game will be Person **1**, and the answer will be **1**.

### **Approach 1: Brute Force using Circular Linked List**

The idea here is to simulate the game as mentioned in the question and find the winner. We need to use a circular data structure, and elements from it can be deleted from anywhere in constant time.

Therefore, we use a **circular singly linked list** to simulate the process as it is circular in nature, and we can delete a node from it anywhere in constant time.

- First of all, we create a circular linked list of length **N** with values of nodes assigned 1,2,..., **N**.
- Then we will start simulating the game until the length of the linked list is greater than 1.
- To simulate one iteration of the game, we first initialise an iterator at the head of the linked list and move that iterator **K - 1** times to the right.
- Delete the node currently at that position.
- Move the iterator to the right and restart the game.

### **Steps:**

1. Define a struct node having a pointer **next** and an integer **shirtNo** denoting shirt number.
2. Initialise the head of the linked list with **shirtNo = 1**.
3. Initialise a pointer named **curr** currently pointing to the head.
4. Run a for loop for *i* from 2 to **N**, and do:
  - a. For each, *i* Initialize a new node **temp** with **shirtNo = i**.
  - b. Set **curr → next = temp**
  - c. Set **curr = temp**
5. Make this linked list as a circular linked list by doing **curr → next = head**.

6. Initialise a pointer **ballInHand** pointing to the head and a **currentPlayers** integer variable to **N**.
7. While **currentPlayers** is greater than one:
  - a. Do a for loop from  $i = 1$  to  $K - 1$  and set **ballInHand** to **ballInHand** → **next**.
  - b. Set **ballInHand** → **shirtNo** to **(ballInHand** → **next**) → **shirtNo**.
  - c. Initialise a node **temp** as **ballInHand** → **next**.
  - d. Set **ballInHand** → **next** as **temp** → **next** and delete the node to which **temp** is pointing.
  - e. Decrease **currentPlayers** by 1.
8. At last, return **ballInHand** → **shirtNo** as the winner of the game.

**Time Complexity:**  $O(N * K)$ , where **N** denotes the number of players of the game and **K** is the jump parameter.

In the worst case, it takes **K** iterations to eliminate one person from the game, and we need to eliminate **N - 1** person from the game. Hence, the overall time complexity is  $O(N * K)$ .

**Space Complexity:**  $O(N)$ , where **N** denotes the number of players in the game. The number of elements in the circular linked list is **N**. Hence, the overall space complexity is  $O(N)$ .

### Approach 2: Optimized Iterative approach

The idea here is to first think of a recursive relation to predict the winner of the game and then optimise the recursive approach to an iterative approach to solve the problem optimally.

Let **predictTheWinner(N, K)** denote the winner of the game when it is played with **N** players using the jump parameter **K**.

We can find the winner of the game using the below recursive relation:

**predictTheWinner(N, K) = (predictTheWinner(N - 1, K) + K) % N + 1** with the base case **predictTheWinner(1, K) = 1**.

We can think of this recursive relation as follows:

When the first person (**K<sup>th</sup>** from starting) is eliminated, **N - 1** persons are left. Thus we call

**predictTheWinner(N - 1, K)** to find the winner when **N - 1** players are playing.

But the **shirtNo** returned by **predictTheWinner(N - 1, K)** as the winner will treat the player with **shirtNo K % N + 1** as the first player. Hence, we need to make changes to the **shirtNo** returned by **predictTheWinner(N - 1, K)**.

Now we will convert the recursive solution in an iterative approach.

### Steps:

1. Initialise a variable **winnerOfTheGame** with a value of 0.
2. Run a for loop for  $i = 1$  to **N** and in each iteration
  1. Set **winnerOfTheGame** as **(winnerOfTheGame + K) % i**
3. At last, return **winnerOfTheGame + 1** as the final winner of the game.

**Time Complexity:**  $O(N)$ , where  $N$  denotes the number of players of the game. In the worst case, it takes  $N$  iterations of the for loop to predict the winner.

**Space Complexity:**  $O(1)$ . In the worst case, constant extra space is required.

## 6. Deque [<https://coding.ninja/P120>]

**Problem Statement:** You need to implement a class for Deque—that is, for a double-ended queue. In this queue, elements can be inserted and deleted from both ends. You don't need to double the capacity.

You need to implement the following functions -

1. **pushFront(X):** Inserts an element  $X$  to the front of the deque. Returns true if the element is inserted, otherwise false.
2. **pushRear(X):** Inserts an element  $X$  to the back of the deque. Returns true if the element is inserted, otherwise false.
3. **popFront():** Pops an element from the front of the deque. Returns -1 if the deque is empty, otherwise returns the popped element.
4. **popRear():** Pops an element from the back of the deque. Returns -1 if the deque is empty, otherwise returns the popped element.
5. **getFront():** Returns the first element of the deque. If the deque is empty, it returns -1.
6. **getRear():** Returns the last element of the deque. If the deque is empty, it returns -1.
7. **isEmpty():** Returns true if the deque is empty, otherwise false.
8. **isFull():** Returns true if the deque is full, otherwise false.

**The following types of queries denote these operations:**

Type 1: for **pushFront(X)** operation.

Type 2: for **pushRear(X)** operation.

Type 3: for **popFront()** operation.

Type 4: for **popRear()** operation.

Type 5: for **getFront()** operation.

Type 6: for **getRear()** operation.

Type 7: for **isEmpty()** operation.

Type 8: for **isFull()** operation.

### **Input Format:**

The **first line** of input contains two space-separated integers  $N$  and  $Q$ , denoting the size of the deque and the number of queries to be performed, respectively.

The next  $Q$  lines specify the type of operation/query to be performed on the data structure.

Each query contains an integer  $P$  denoting the type of query.

For the query of type 1 and 2, the integer  $P$  is followed by a single integer  $X$  denoting the element on which operation is to be performed.

For the queries of type 3 to 8, a single integer **P** is given, denoting the query type.

**Output Format:**

For each query, print the output returned after performing the corresponding operation on the data structure.

**Sample Input:**

```
5 7  
7  
1 10  
1 20  
2 30  
5  
4  
4
```

**Sample Output:**

```
True  
True  
True  
True  
20  
30  
10
```

**Explanation:**

For the given input, we have the number of queries, Q = 7.

Operations performed on the deque are as follows:

**isEmpty()**: Deque is initially empty. So, this returns true.

**pushFront(10)**: Insert the element '10' in the front of the deque. This returns true.

**pushFront(20)**: Insert the element '20' in the front of the deque. This returns true.

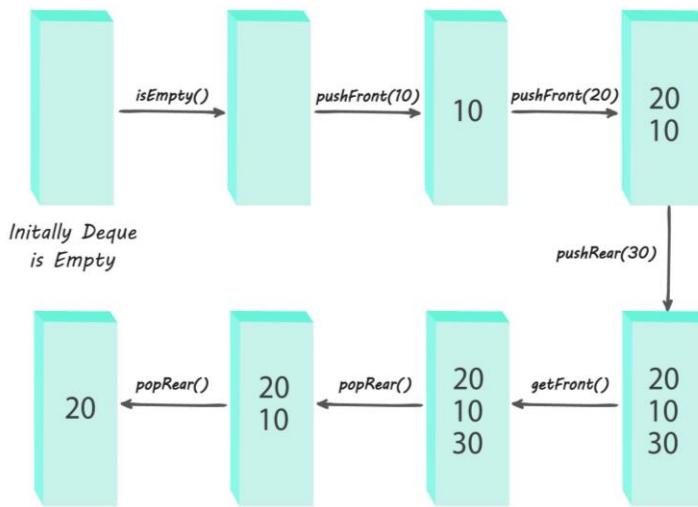
**pushRear(30)**: Insert the element '30' in the back of the deque. This returns true.

**getFront()**: Returns the front element of the deque i.e. 20.

**popRear()**: Pop an element from the back of the deque. This returns 30.

**popRear()**: Pop an element from the back of the deque. This returns 10.

The following image shows the snapshots of the deque after each operation:



### Approach 1: Using Arrays

1. A deque (double-ended queue) is a generalised form queue, which supports insert and delete operations from either end of the queue.
2. We can easily implement deque with the help of a circular array.
3. We can think of a circular array as an array in which 0th position occurs after  $(N - 1)^{th}$  position and/or  $(N - 1)^{th}$  position occurs before 0<sup>th</sup> position, resulting in the two ends of the array wrapping up to make a circle.
4. As the insertion and deletion can occur at both ends in a deque, we maintain two pointers—**front** and **rear**, to keep track of the first and the last filled positions in the array.

#### Steps:

1. Create an array **arr** of size **N**. This will be used to implement the deque.
2. Initialise two pointers, **front** and **rear**, to -1.
3. **PushFront(X)** Operation:
  - a. If the deque is full, return false.
  - b. Otherwise, if the deque is empty, initialise **front** and **rear** to 0.
  - c. Otherwise, if the deque is NOT empty, we decrement the **front** by one (if **front** is equal to 0, it becomes **N - 1** as the array is circular).
  - d. Insert the element at the front of the queue—that is, at position **front**, which implies **arr[front] = X**.
  - e. Return true.
4. **PushRear(X)** Operation:
  - a. If the deque is full, return false.
  - b. Otherwise, if the deque is empty, initialise **front** and **rear** to 0.
  - c. Otherwise, if the deque is NOT empty, we increment the **rear** by 1 (if **rear** is equal to **N - 1**, it becomes 0 as the array is circular).
  - d. Insert the element at the back of the queue—that is, at position **rear**, which implies **arr[rear]=X**.
  - e. Return true.

5. **PopFront** Operation:

- a. If the deque is empty, return -1.
- b. Get the first element of the deque and store it in a variable, say **item**.
- c. If **front = rear**, then deque has only one element, and by removing it, the deque becomes empty. So, set **front** and **rear** to -1.
- d. Otherwise, delete the element from the front by incrementing the **front** by 1 (If the **front** is equal to **N - 1**, **front** becomes 0 as the array is circular).
- e. Return **item**.

6. **PopRear** Operation:

- a. If the deque is empty, return -1.
- b. Get the last element and store it in a variable, say **item**.
- c. If **front = rear**, then deque has only one element, and by removing it, the deque becomes empty. So, set **front** and **rear** to -1.
- d. Otherwise, delete the element from the back by decrementing the **rear** by 1 (If the **rear** is equal to 0, it becomes **N - 1** as the array is circular).
- e. Return **item**.

7. **GetFront** Operation:

- a. If the deque is empty, return -1.
- b. Otherwise return the first element of the deque—that is, value at **front**, which implies return **arr[front]**.

8. **GetRear** Operation:

- a. If the deque is empty, return -1.
- b. Otherwise, return the last element of the deque—that is, value at **rear**, that is return **arr[rear]**.

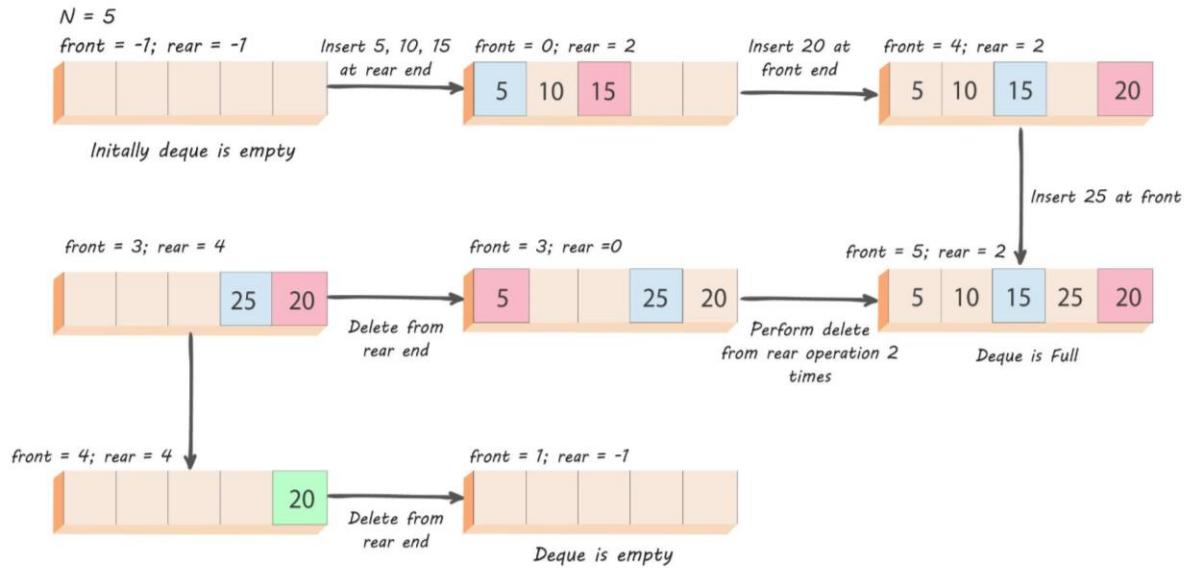
9. **isEmpty** Operation:

- a. If **front = -1**, deque is empty. So, return true.
- b. Otherwise, return false.

10. **isFull** Operation:

- a. Deque is full when all the array positions are occupied—that is when the array becomes full. This can happen in two situations—when **front** is 0 and **rear** is **N - 1**, or when **front** is equal to (**rear + 1**).
- b. Hence, if (**front = 0 AND rear = N - 1**) OR (**front = rear + 1**), deque is full. So, return **true**, else return **false**.

The following image shows the snapshots of the deque on performing some of the operations. Here green cell represents the position pointed by **front**, the red cell represents the position pointed by **rear**, and yellow cell represents the position which is pointed by both **front** and **rear** pointers:



**Time Complexity:**  $O(1)$  for every operation. All the operations can be performed in a constant time.

**Space Complexity:**  $O(N)$ , where  $N$  denotes the size of the deque.  $O(N)$  extra space is required for the array to store all the elements in the deque.

## 7. Sliding Maximum [\[https://coding.ninja/P121\]](https://coding.ninja/P121)

**Problem Statement:** You are given an array of integers of length  $\mathbf{N}$  and a positive integer  $\mathbf{K}$ . You need to find the maximum elements for each contiguous subarray of size  $\mathbf{K}$  of the array.

**Example:** If the given input array is  $[3, 4, -1, 1, 5]$  and  $K = 3$ .

Then the output should be :  $[4, 4, 5]$  because

For the first subarray of length 3 that is :  $[3, 4, -1]$ , its maximum element is 4,

For the second subarray of length 3 that is :  $[4, -1, 1]$  its maximum element is 4

For the last subarray of length 3 that is :  $[-1, 1, 5]$  its maximum element is 5

So the answer is : $[4, 4, 5]$ .

### Input Format:

The **first line** of input contains a single integer  $T$ , representing the number of test cases or queries to be run.

For each test case:

The **first line** contains two positive integers,  $\mathbf{N}$  and  $\mathbf{k}$ , representing the length of the array and length of the subarray, respectively.

The **Second line** contains  $\mathbf{N}$  space-separated integers representing the elements of the array.

**Output Format:**

For each test case, return the maximum elements for each contiguous subarray of size **K** of the array.

**Sample Input:**

```
1  
5 3  
3 2 -6 1 0
```

**Sample Output:**

```
3 2 1
```

**Explanation:**

The subarray of length **K**     maximum element of the subarray.

3 2 -6	3
2 -6 1	2
-6 1 0	1

Thus the answer is: "3 2 1".

**Approach 1: Brute Force Approach**

1. Create a nested loop. The outer loop will go from  $i = 0$  to  $i = N - K$ , where  $i$  represents the starting indices of all possible **K sized**- subarrays for the given input array.
2. The inner loop will go from  $j = i$  to  $j = i + K - 1$ . This will cover all the elements of the **K sized** subarray chosen having the starting index  $i$ .
3. Keep track of the maximum element in the inner loop and print it.

**Time Complexity:**  $O(N * K)$  where **N** denotes the length of the input array, and **K** is the size of the subarrays.

**Space Complexity:**  $O(1)$  because the extra space being used (looping variables, maximum element) is constant for any value of **N** and **K**.

**Approach 2: MaxHeap****Steps:**

1. Create a class pair where we will have two variables, i.e. elements of the given array and their corresponding indexes.
2. Create an array **ans** of size  $N - K + 1$  size to store the maximum element of each subarray of size **K**.
3. Create a maxheap and store the first **K** pair of elements and their indexes in it.
4. Max-heapify the elements of the heap and store the top element in the array **ans**.
5. Run a loop from **K** to **N**, and in every iteration, add the current element of the array with its index in the heap.

6. Keep removing all the top elements which got out of the window by checking the condition, which is **current index-heap.peek().index >= K**, as it will tell which elements are not in the current window
7. Now the top element will be the maximum of the current subarray. Add this maximum element of the subarray(top element of the heap) in the array **ans**.
8. Return the **ans** array.

**Time Complexity:**  $O(N * \log K)$ , where **N** denotes the array's length, and **K** is the subarray size. We are pushing and popping out the elements from the heap only once, and in each iteration, we are heapifying the elements of heap that will take  **$O(\text{size of heap})$** ; therefore, total time complexity will be  $O(N * \log K)$ .

**Space Complexity:**  $O(K)$ , where **N** denotes the array's length, and **K** is the subarray size. We are using heap, and heap will store at max **K** element at a time.

### Approach 3: Dequeue

The idea is to use the deque to hold the index of the maximum element and restrict the deque size to **K**. We can use a double-ended queue to keep only the indices of those useful elements. The use of deque is to add and drop elements from both the ends of a queue. We will slide the window of **K** elements by “dropping” the first element and “adding” the next element after the window to move it forward.

#### Steps:

1. Create a deque to store **K** elements and an array of size  **$N - K + 1$**  to store the answer of each subarray of size **K**.
2. Run a loop and insert the first **K** elements in the deque. While inserting the element, if the element at the back of the deque is smaller than the current element, then remove all those elements and then insert the current element at the end of the deque.
3. Now, run a loop from **K** to the end of the array.
4. Store the front element of the deque in the array.
5. Remove the element from the front of the deque if they are out of the current window.
6. Insert the next element in the deque. While inserting the element, if the element at the back of the deque is smaller than the current element, then remove all those elements and then insert the current element.
7. Before returning the final array, store the maximum element of the last window in the array.
8. Return the array.

**Time Complexity:**  $O(N)$ , where **N** denotes the array's length, and **K** is the subarray size as we are inserting and deleting every element of the array in the deque only once.

**Space Complexity:  $O(K)$ ,** where  $N$  denotes the length of the array and  $K$  is the size of the subarray; we are using a deque of size  $K$  to store the current window elements.

# 14. Tries and String Algorithms

---

## Introduction to Tries

Suppose we want to implement a word dictionary and perform the following functions:

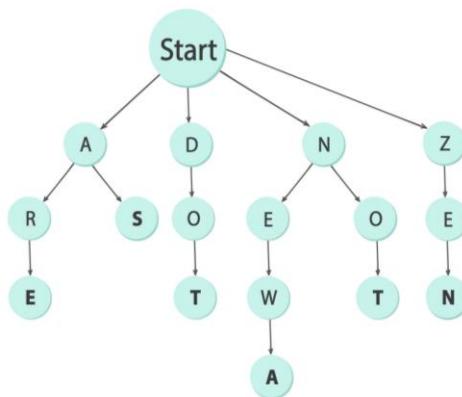
- Adding a word
- Searching a word
- Removing a word

To perform the given functions, we could consider hashmaps as an efficient data structure with the **average case time complexity** for the insertion, deletion, and retrieval operations as **O(1)**.

For the same purpose, we can also use another data structure known as **tries**.

A trie is a tree-like structure whose nodes contain letters, and by structuring these nodes in a particular way, words/strings can be retrieved by traversing down a branch of the tree.

Below is the visual representation of a trie:



Here, the node at the top, which is named as **start**, is the root node of our **n-ary tree**.

## Trie Node Structure

The trie node consists of the following components :

- Char **ch** (representing the character).
- An array **children** of size 26 of trie node pointers.
- Bool **isTerminal** (boolean variable representing the termination of a word)

## Searching a Word

- To search in a trie, we would need to traverse the complete length of the string, which would take **O(word\_length)** time.
- Let us take an example. Suppose we want to search for the word **DOT** in the above trie; we begin with searching for the letter **D** as the direct child of the start node and then

search for **O** and **T**, respectively. If the word is present in the trie, **true** is returned, else **false**. In this case, **true** would be returned as the word **DOT** is present in the trie.

- Now, if we want to search the word **AR** in the given trie, following the above approach, we would return **true** as when going from the root node, we encounter character **A** and then character **R**.

However, ideally, we should return **false** as the actual words that were inserted in the trie was **ARE** and not **AR**.

To overcome this, we mark the last character of every word entered in the **TRIE** as **bolded ( isTerminal property set to True)**, representing the **termination** of a word starting from the root node. Hence, while searching for a word, we should always be careful about the last letter that should be bolded to represent the termination of that word.

## Inserting a Word

- Suppose we want to insert the word **ARE** in the trie. We begin from the root, search for the first letter **A** and then insert **R** as its child and then similarly insert the letter **E** as the child of node **R**. You can see it as the left-most path in the above trie.
- Similarly, for inserting the word **AS** in the trie. We follow the same procedure as above. First-of-all, we find the letter **A** as the child of the root node, and then we search for **S**. If **S** is already present as the child of **A**, then we do nothing as the given word is already present; otherwise, we insert **S**. Similarly, we add other words in the trie.
- This approach also takes **O(word\_length)** time for **insertion** of a string as every time we explore through one of the branches of the trie to check for the prefix of the word already present.

## Deleting a Word

- While inserting a word in a trie, the last letter of the word should have its **isTerminal** property set to **True** as a mark of termination of the word. Thus to remove a word from the trie, we simply set the **isTerminal** property as **False**.
- For example: If we want to remove the string **DO** from the above trie while preserving the occurrence of string **DOT**, we reach at **O** and then set the **isTerminal** property as **False**. This way, the word **DO** is removed, but at the same time, another word **DOT** which was on the same path as that of word **DO**, is still preserved in the trie structure.

- For the **removal** of a word from trie, the time complexity is still **O(word\_length)** as we are traversing the complete length of the word to reach the last letter to set the **isTerminal** property to **False**.

## Comparison between Hashmaps and Tries

While implementing a dictionary, using tries helps us save a lot of space over hashmaps. Suppose we have 1,000 words starting from character **A** that we want to store. If we try to hold these words using a hashmap, then for each word, we will have to store the character **A** separately for each of the 1000 strings. But in the case of tries, we need to store the character **A** just once. Thus we save a considerable amount of space using tries.

## Operations on Tries

### Insertion in a Trie

Given a word which is to be inserted into a trie. We assume that the word is already **not present** in the trie.

### Steps for insertion in the Trie

- Start from the root node
- For each character of the string, we want to insert, check if it has a child node corresponding to itself in the root. If the required child node does not exist, then create a new node corresponding to that character.
- Descend from the root to the child node corresponding to the current character.
- Set the last character's terminal property to be **true**.

```
function insert(root, word)

    for each character c in word
        /*
            check if it is already present and
            if not, then create a new node
        */
        index = c - 'a'
        if root.child[index] equals null
            root.child[index] = new node

        root = root.child[index]

    // mark the last character to be the end of the word
```

```
root.isTerminal = true  
return root
```

## Searching in a Trie

Given a word, we want to find out whether it is present in the trie or not. If the search word is present in the trie, then return **true** else, return **false**.

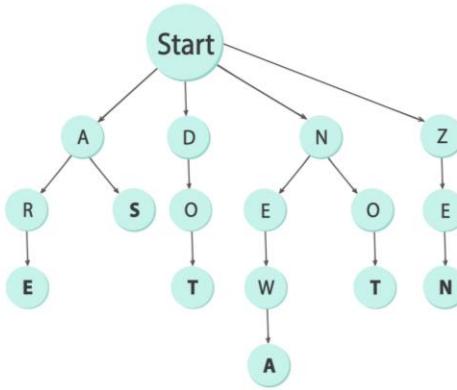
### Steps for searching in a trie :

- Start from the root node.
- For each character of the string we want to search, check if it has a child node corresponding to itself in the root or not. If the required child node does not exist, return **false**.
- Descend from the root to the child node corresponding to the current character.
- Return **true** on successful completion of the above loop only if the last node is not null and its terminal property is set to **true**, else return **false**.

```
function search(root, word)  
  
    for each character c in word  
        /*  
            check if it is already present and  
            if not then return false  
        */  
        index = c - 'a'  
  
        if root.child[index] equals null  
            return false  
  
        root = root.child[index]  
  
    if root != null and root.isTerminal equals True  
        return true  
    else  
        return false
```

## Deletion in a in Trie

Given a **word**, we want to remove it from the trie if it is present.



Consider the above example. Suppose we want to remove **NEWS** from the trie. We recursively call the delete function, and as soon as we reach **S**, we mark its **isTerminal** property as false. Now we see that it has no children remaining, so we delete it. After returning, we get to **W**. We see that the condition: **isTerminal** is false, and no children are remaining for the current node; hence we just return again. Proceeding the same way back up to **E** and **N**, we delete the word **NEWS** from the trie.

#### Steps for deletion in a Trie

- Call the function delete for the root of the trie.
- Update the child corresponding to the current character by calling delete for the child.
- If we have reached the final character, mark the **isTerminal** property of the node as **False** and delete it if all its children are **NULL**.
- If all the children of this node have been deleted and this character is not the prefix of any other word, then also delete the current node.

```

function delete(root, depth, word)
    if root is null
        /*
            The word does not exist
            hence return null
        */
        return null

    if depth == word.size
        /*
            mark the isTerminal as false and delete if no child is
            present
        */

        root.isTerminal = false
        if root.children are null

```

```

delete(root)
root = null

return root

index = word[depth] - 'a'
/*
    update the child of the root corresponding to the current
    character
*/
root.children[index] = delete(root.children[index], depth + 1 , word)

if(root.children are null and root.isTerminal == false)
    delete(root)
    root = null

return root

```

## Time Complexity of Various Functions

For a given string of length **L**, the time complexities of various operations in a Trie are as follows:

Operations	Time Complexity
<b>insert(word)</b>	O(L)
<b>search(word)</b>	O(L)
<b>delete(word)</b>	O(L)

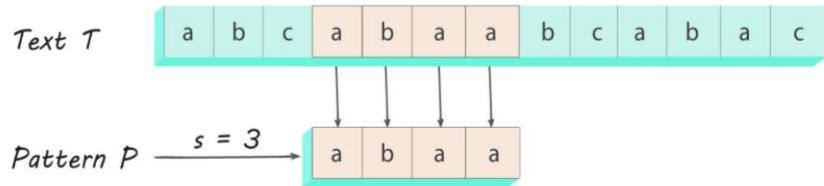
## Applications of Tries

- Tries are used as data structures to implement functionalities like **dictionaries, lookup tables, matching algorithms**, etc.
- They are also used for many practical applications such as **auto-complete in editors and mobile applications**.
- They are used in **phone book search applications** where efficient searching of a large number of records is required.

## Introduction to String Algorithms

In this section, we will primarily walk through two string searching algorithms—**Knuth Morris Pratt algorithm** and **Z-Algorithm**.

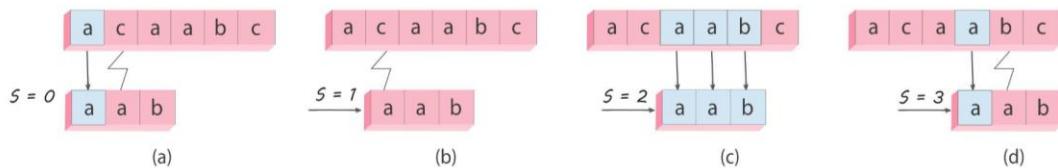
Suppose you are given a string **text** of length **n** and a string **pattern** of length **m**. You need to find all the occurrences of the **pattern** in the **text** or report that no such instance exists.



In the above example, the **pattern** string to be searched, that is, “**abaa**” appears at position 3 (0-indexed) in the **text** string “**abcabaabcabac**”.

## Naive Algorithm

A naive way to implement this pattern searching algorithm is to move from each position in the **text** string and start matching the **pattern** from that position until we encounter a mismatch between the characters of the two strings or say that the current position is a valid one.



In the given picture, The length of the **text** string is 5, and the length of the **pattern** string is 3. For each position from 0 to 3 in the **text** string, we choose it as the starting position and then try to match the next 3 positions with the **pattern**.

### Naive pattern matching algorithm

- Run a loop for  $i$  from 0 to  $n - m$ , where  $n$  and  $m$  are the lengths of the **text** and the **pattern** string, respectively.
  - Run a loop for  $j$  from 0 to  $m - 1$ , try to match the  $j$ th character of the **pattern** with  $(i + j)^{th}$  character of the **text** string.
  - If a mismatch occurs, skip this instance and continue to the next iteration.
  - Else output this position as a matching position.

```
function NaivePatternSearch(text, pattern)
    // iterate for each candidate position
    for i from 0 to text.length - pattern.length
```

```

// boolean variable to check if any mismatch occurs
match = True

for j from 0 to pattern.length - 1
    // if mismatch make match = False
    if text[i + j] not equals pattern[j]
        match = False

    // if no mismatch print this position
    if match == True
        print the occurrence i
return

```

## Knuth Morris Pratt Algorithm

We first define a prefix function of the string - The prefix function of a string **s** is an array **lps** of length same as that of string **s** such that **lps[i]** stores the length of the maximum prefix that is also the suffix of the substring **s[0..i]**.

### For example:

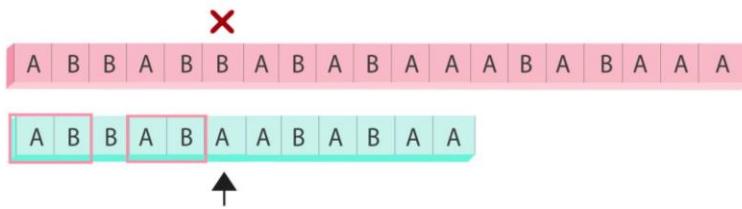
For the pattern "AAABAAA",

**lps[]** is [0, 1, 2, 0, 1, 2, 3]

**lps[0]** is 0 by definition.

The longest prefix that is also the suffix of string **s[0..1]** which is AA is 1. (Note that we are only considering the proper prefix and suffix).

Similarly, For the whole string AAABAAA it is 3; hence the **lps[6]** is 3.



### Algorithm for computing the LPS array

- We compute the prefix values **lps[i]** in a loop by iterating from 1 to **n - 1**.
- To calculate the current value **lps[i]** we set a variable **j** denoting the length of best suffix possible till index **i - 1**. So **j = lps[i - 1]**.
- Test if the suffix of length **j + 1** is also a prefix by comparing **s[j]** with **s[i]**. If they are equal then we assign **lps[i] = j + 1** else reduce **j = lps[j - 1]**.
- If we reach **j = 0**, we assign **lps[i] = 0** and continue to the next iteration .

```

function PrefixArray(s)
    n = s.length;
    // initialize to all zeroes
    lps = array[n];

    for i from 1 to n - 1
        j = lps[i - 1];
        // update j untill s[i] becomes equal to s[j] or j becomes zero
        while j greater than 0 && s[i] is not equal to s[j]
            j = lps[j - 1];

        // if extra character matches increase j
        if s[i] equal to s[j]
            j += 1;

        // update lps[i]
        lps[i] = j;

    // return the array
    return lps

```

### Algorithm for searching the pattern using KMP Algorithm

Consider a new string **S' = pattern + '#' + 'text'** where + denotes the concatenation operator.

Now, the condition for the **pattern** string to appear at a position  $[i - M + 1 \dots i]$  in the string **text**, the **lps[i]** should be equal to **M** for the corresponding position of **i** in **S'**.

Note that **lps[i]** cannot be larger than **M** because of the '#' character.

- Create  $S' = \text{pattern} + \# + \text{text}$
- Compute the **lps** array of  $S.'$
- For each  $i$  from  $2*M$  to  $M + N$  check the value of **lps[i]**.
- If it is equal to **M**, then we have found an occurrence at the position  $i - 2*M$  in the string **text** else,; else continue to the next iteration.

```

function StringSearchKMP(text, pattern)
    // construct the new string
    S' = pattern + '#' + text

    // compute its prefix array
    lps = PrefixArray(S')
    N = text.length
    M = pattern.length

```

```

for i from 2*M to M + N
    // longest prefix match is equal to the length of pattern
    if lps[i] == M
        // print the corresponding position
        print the occurrence as i - 2*M
return

```

Let us understand the above algorithm with the help of the following example:

text[] = "AAAAAZAAZA"

pattern[] = "AAAA"

lps[] = {0, 1, 2, 3}

i = 0, j = 0

text[] = "AAAAAZAAZA"

pattern[] = "AAAA"

text[i] and pattern[j] match, do i++, j++

i = 1, j = 1

text[] = "AAAAAZAAZA"

pattern[] = "AAAA"

text[i] and pattern[j] match, do i++, j++

i = 2, j = 2

text[] = "AAAAAZAAZA"

pattern[] = "AAAA"

pattern[i] and pattern[j] match, do i++, j++

i = 3, j = 3

text[] = "AAAAAZAAZA"

pattern[] = "AAAA"

text[i] and pattern[j] match, do i++, j++

i = 4, j = 4

Since j == M, print pattern found and reset j,

j = lps[j-1] = lps[3] = 3

Here, unlike the Naive algorithm, we do not match the first three characters of this window.

Value of  $\text{lps}[j-1]$  (in the above step) gave us an index of the next character to match.

$i = 4, j = 3$

$\text{text}[] = "AAAAAZAAZA"$

$\text{pattern}[] = "AAAA"$

$\text{text}[i]$  and  $\text{pattern}[j]$  match, do  $i++$ ,  $j++$

$i = 5, j = 4$

Since  $j == M$ , print pattern found and reset  $j$ ,

$j = \text{lps}[j-1] = \text{lps}[3] = 3$

Again unlike Naive algorithm, we do not match first three characters of this window. Value of  $\text{lps}[j-1]$  (in above step) gave us an index of the next character to match.

$i = 5, j = 3$

$\text{text}[] = "AAAAAZAAZA"$

$\text{pattern}[] = "AAAA"$

$\text{text}[i]$  and  $\text{pattern}[j]$  do NOT match and  $j > 0$ , change only  $j$

$j = \text{lps}[j-1] = \text{lps}[2] = 2$

$i = 5, j = 2$

$\text{text}[] = "AAAAAZAAZA"$

$\text{pattern}[] = "AAAA"$

$\text{text}[i]$  and  $\text{pattern}[j]$  do NOT match and  $j > 0$ , change only  $j$

$j = \text{lps}[j-1] = \text{lps}[1] = 1$

$i = 5, j = 1$

$\text{text}[] = "AAAAAZAAZA"$

$\text{pattern}[] = "AAAA"$

$\text{text}[i]$  and  $\text{pattern}[j]$  do NOT match and  $j > 0$ , change only  $j$

$j = \text{lps}[j-1] = \text{lps}[0] = 0$

$i = 5, j = 0$

```
text[] = "AAAAAZAAZA"  
pattern[] =      "AAAA"  
text[i] and pattern[j] do NOT match and j is 0, we do i++.
```

```
i = 6, j = 0  
text[] = "AAAAAZAAZA"  
pattern[] =      "AAAA"  
text[i] and pattern[j] match, do i++ and j++
```

```
i = 7, j = 1  
text[] = "AAAAAZAAZA"  
pattern[] =      "AAAA"  
text[i] and pattern[j] match, do i++ and j++ // the algorithm continues in the same fashion
```

## Applications of Tries

- Used in plagiarism detection between documents.
  - Used in bioinformatics and DNA sequencing to match DNA and RNA patterns.
  - Used in various editors and spell checkers to correct the spellings.
- 

## Practice Problems

### 1. Implementation: Trie [<https://coding.ninja/P122>]

**Problem Statement :** Implement a trie data structure to support the following operations:

- insert(word)** - To insert a string "**word**" in the trie.
- search(word)** - To check if string "**word**" is present in the trie or not.
- startsWith(word)** - To check if there is any string in the trie that starts with the given prefix string "**word**".

Three types of queries denote these operations:

**Type 1:** To insert a string "**word**" in trie represented as **1 word**

**Type 2:** To check if the string "**word**" is present in the trie or not represented as **2 word**

**Type 3:** To check if there is any string in the trie that starts with the given prefix string "**word**" which is represented as **word**

**Input Format:**

The first line contains an Integer '**Q**', which denotes the total number of queries to be run. Then next '**Q**' lines denote each query:

The first and only line of each query contains the type of query (as described above) and a string "**word**" separated by a single space.

#### **Output Format:**

For each query of **Type 2**, print the string "**true**" if string "**word**" is present in trie or "**false**" otherwise.

For each query of **Type 3**, print the string "**true**" if there is any string in the trie that starts with the given prefix string "**word**" or "**false**" otherwise.

#### **Sample Input:**

```
5
1 hello
1 help
2 help
3 hel
2 hel
```

#### **Output :**

```
true
true
false
```

#### **Explanation:**

Query 1: "**hello**" is inserted into the trie.

Query 2: "**help**" is inserted into the trie.

Query 3: "**true**" is printed as "help" is present into the trie.

Query 4: "**true**" is printed as "**hello**" and "**help**" strings are present in the trie having the prefix as "**hel**"

Query 5: "**false**" is printed as the string "**hel**" is not present in the trie.

### **Approach 1: Hashmap/Dictionary Approach**

Firstly we define the **node class of trie** having members as follows:

1. **child** - storing the address of child nodes (hashmap of character number and address of the corresponding child trie nodes)
2. **isEnd** - a bool variable for marking this node as an end of some word.

Then we define our **trie class** having members as follows:

1. **root** - The root node for the whole trie, every word starts from this root node.
2. **insert(word)** - The function used to insert a string "**word**" in the trie.
3. **search(word)** - The function to check if string "**word**" is present in the trie or not.

4. **startsWith(word)** - The function checks if there is any string in the trie that starts with the given prefix string "**word**".

Steps to implement the function **insert(word)**:

1. Initialise the current node with the root node of the trie.
2. Iterate over the word from left to right and if there is no node present for the next character of the input word, then create a **new node** and store it in the appropriate child member of the previous character's node.
3. Keep updating the current node to the corresponding node for the next character of the given input word.
4. Finally, when we reach the end of the word, mark the **isEnd member** of the current node to **true** as it marks the termination of a word in the trie.

Steps to implement the function **search(word)**:

1. Initialise the current node with the **root** node of trie.
2. Iterate over the word from left to right and if there is no node present for the next character of the given input word, then return **false** as the word is not present in the trie.
3. Keep updating the current node to the corresponding node for the next character of the given input word.
4. Finally, if we reach the end of the word, return **True** only if the **isEnd** member of the current node is **true**, else return **false**.

Steps to implement the function **startsWith(word)**:

1. Initialise the current node with the **root** node of trie.
2. Iterate over the word from left to right and if there is no node present for the next character of the word, then return **false** as no word is present in the trie with this word as a Prefix.
3. Keep updating the current node to the corresponding node for the next character of the given input word.
4. Finally, when we reach the end of the word, return **true** as some string must be present in the trie with the input **word** as the prefix string.

**Time Complexity:** All the operations, namely **search(word)**, **insert(word)**, **startsWith(word)**, have individual time complexities of **O(W)**.

Therefore the overall time complexity for processing **N** queries is **O(N\*W)**, where **N** is the number of queries and **W** is the average length of the words.

**Space Complexity:** **O(N\*W)** where **N** denotes the number of words inserted and **W** is the average length of words because we are making nodes for each character.

## Approach 2: Array Approach

Firstly we define the **node class of trie** having members as follows:

1. **child** - storing the address of child nodes (hashmap of character number and address of the corresponding child trie Nodes)

2. **isEnd** - a bool variable for marking this node as an end of some word.

Then we define our trie class having members as follows:

1. **root** - The root node for the whole trie, every word starts from this root node.
2. **insert(word)** - The function used to insert a string "**word**" in the trie
3. **search(word)** - The function to check if string "**word**" is present in the trie or not.
4. **startsWith(word)** - The function checks if there is any string in the trie that starts with the given prefix string "**word**".

Steps to implement the function **insert(word)**:

1. Initialise the current node with the **root** node of trie.
2. Iterate over the word from left to right, and if there is NULL in the node's address for the next character of the word, then create a new node and store the address of this new node in the corresponding child member of the previous character's node.
3. Keep updating the current node to the corresponding node for the next character of the given input **word**.
4. Finally, if we reach the end of the word, mark the **isEnd member** of the current node to **true** as it marks the termination of a word in the trie.

Steps to implement the function **search(word)**:

1. Initialise the current node with the **root** node of trie.
2. Iterate over the word from left to right and if there is NULL in the node's address for the next character of the word, then return **false** as the word is not present in the trie.
3. Keep updating the current node to the corresponding node for the next character of the given input word.
4. Finally, if we reach the end of the word, return the **isEnd** bool variable of the current node as it will denote whether the word is present in the trie or not.

Steps to implement the function **startsWith(word)**:

1. Initialise the current node with the **root** node of trie.
2. Iterate over the word from left to right and if there is NULL in the address of the node for the next character of the word, then return **false** as it denotes that there is no word present in the trie with this word as a Prefix.
3. Keep updating the current node to the corresponding node for the next character of the given input word.
4. Finally, if we reach the end of the word, return **true** as it denotes that some word must be present in the trie with the input **word** as the prefix string.

**Note:** This array approach of storing a child is more time-efficient than a hashmap approach for storing a child as the constant factor of **O(1)** in hashmaps is more costly.

**Time Complexity:** All the operations, namely **search(word)**, **insert(word)**, **startsWith(word)**, have individual time complexities of **O(W)**.

Therefore the overall time complexity for processing **N** queries is **O(N\*W)**, where **N** is the number of queries and **W** is the average length of the words.

**Space Complexity:** **O(N \* W)** where **N** denotes the number of queries and **W** is the average length of words. As we are making nodes for each character of the word so, at max, we can have all unique sequences of the words.

Thus, overall space can be at max **26 \* N \* W**, so overall space complexity is **O(N\*W)**

## 2. Word Ladder [<https://coding.ninja/P123>]

**Problem Statement:** You are given two strings, **BEGIN** and **END** and an array of strings **DICT**; your task is to find the length of the **shortest** transformation sequence from **BEGIN** to **END** such that in every transformation, you can change exactly one alphabet and the word formed after each transformation exists in **DICT**.

### **Input Format :**

The **first line** of input contains an integer **T** denoting the number of test cases.

The **first line** of each test case contains a string **BEGIN**.

The **second line** of each test case contains a string **END**.

The **third line** of each test case contains a single integer **N** denoting the length of the **DICT**.

The **fourth line** of each test case contains **N** space-separated strings denoting the strings present in the **DICT** array.

### **Output Format :**

For each test case, return a single integer representing the length of the **shortest** transformation sequence from **BEGIN** to **END**. The output of each test case will be printed in a separate line.

### **Sample Input:**

```
1
goa
pro
5
poa pro bro say pra
```

### **Sample Output:**

```
4
```

### **Explanation :**

The given **BEGIN** string is '**goa**', and the **end** string is '**pro**', and we have the array [poa, pro, bro, say, pra].

Firstly we can move from '**goa**' to '**poa**'.

Now we can move from '**poa**' to '**pra**', and finally, we can move from '**pra**' to '**pro**'.

Thus the series of transformations is "goa" → "poa" → "pra" → "pro"; therefore, the length of the shortest transformation sequence from 'goa' to 'pro' is 4.

### Approach 1: Using BFS

The idea here is to use **BFS** traversal by considering an edge between any two **adjacent words** (words that will have a difference of only one alphabet).

Now using BFS, we need to find the shortest path between the **start** word and the **target** word.

#### Steps:

1. As the BFS procedure works, start from **BEGIN** word, add it to the queue **QUEUE** and also declare a variable **COUNT** = 1 to store the answer.
2. Now till the **QUEUE** goes empty, run a while loop—that is, pop each word from the **QUEUE** in the respective iterations.
3. If the current word becomes equal to **END**, then just return the **COUNT**.
4. Else just iterate the word, check how many alphabets are different from the target word, store it in a variable, say **CHECK**.
5. If this **CHECK** variable becomes 1, then that means we have reached the **END** word and return **COUNT**.
6. Else just add the number which is adjacent to the word and add the corresponding word to the **QUEUE**.
7. Now in the final statement, if the function doesn't hit the return statement, then that means there is no possible path so just return -1.

**Time Complexity:**  $O((N^2) * |S|)$ , where **N** denotes the length of the **DICT** and  $|S|$  is the length of each string. Since every word can be added to the queue, and each word needs to be compared with every word in the **DICT**. Each comparison takes  $O(|S|)$ . Thus, the final time complexity is  $O((N^2) * |S|)$ .

**Space Complexity:**  $O(N * |S|)$ , we are using a queue to store all words.

### Approach 2: Using tries

The idea here is to use **BFS** traversal but, along with that, make use of **trie** data structure for storing the words present in **DICT**.

#### Steps:

1. Create a class **TRIE** and insert all the words in the **DICT**.
2. As the **bfs** procedure works, start from the **BEGIN** word, add it to the queue **QUEUE**.
3. We maintain a hashmap for storing the visited words.
4. Now till the **QUEUE** goes empty, run a while loop—that is, pop each word from the **QUEUE** in the respective iterations and mark the entry of the corresponding word in the hashmap as **true**.

5. Here we will calculate all the possible permutations of the word that differ just by one character. In order to generate all the possible permutations, we use a function **ALLPERMUTATIONS**.
6. In this function, we change a character one by one and generate all possible words and check whether it is present in the **DICT** or not. If it is present, we add it to the arraylist.
7. Now add those words in the queue which are not yet visited in the hashmap.
8. We repeat the same procedure until the queue goes empty.
9. Now, if we encounter the target word—that is, **END** then just return from there with the minimum distance; else return -1 finally.

**Time Complexity:**  $O(N * (|S| * |S|))$ , where **N** denotes the length of the **DICT** and  $|S|$  is the length of each string of **DICT**. Since every word can be added to the queue and we are checking all possible permutations for each word, so total complexity for generating permutations will be  $O(|S| * |S|)$ .

**Space Complexity:**  $O(N * (|S| * |S|))$ , where **N** denotes the length of the **DICT** and  $|S|$  is the length of each string of **DICT**. Since for storing all permutations, we will require  $|S| * |S|$  space which is for each iteration. Therefore the total space requirement is of the order of  $(N * |S| * |S|)$ .

### **3. Common Elements** [<https://coding.ninja/P124>]

**Problem Statement:** You are given two one-dimensional arrays containing strings of lowercase alphabets. You are supposed to print the elements that are common to both the arrays—that is, the strings that are present in both the arrays in the order in which these common elements appear in the second array.

#### **Note :**

An element of one array can be mapped only to one element of the other array.

For example, **Array 1 = {“ab”, “dc”, “ab”, “ab”}** **Array 2 = {“dc”, “ab”, “ab”}**

Then the common elements for the above example will be “dc”, “ab” and “ab”.

#### **Input Format:**

The **first line** of the input contains an integer '**T**' representing the number of test cases.

For each test case:

The **first line** contains two integers, **N** and **M**, representing the sizes of both the arrays.

The **second line** contains **N** single space-separated strings representing the elements of the first array.

The **third line** contains **M** single space-separated strings representing the elements of the second array.

#### **Output Format:**

For each test case, the common elements of both the arrays are printed in the order in which they appear in the second array. The elements are printed in a single space-separated manner.

**Sample Input:**

```
1
3 5
at bat rat
rat bar bat rat to
```

**Sample Output:**

```
rat bat
```

**Explanation:**

The common elements of both the arrays are “**bat**” and “**rat**”.

Thus, these two elements are printed in the order in which they appear in the second array.

**Approach 1: Brute Force approach**

The main idea behind this approach is to solve the problem intuitively, and for each string in the second array, check if it exists in the first array or not (we are checking for each string in the second array because we need the output in the order in which they appear in the second array).

**Steps:**

1. Start traversing the second array and for each string in the second array, do the following:
  - a. Loop through the first array and see if the string is present in the first array or not. If it is present in the first array, add the string to the **answer** array as it is present in both the arrays.
  - b. Also, in the first array, replace the string with an empty string so that the same string is not considered again in the subsequent iterations.
2. The **answer** array will contain all the strings in the order in which they appear in the second array. Lastly, return the **answer**.

**Time Complexity:  $O(N * M)$ , where **N** and **M** are the sizes of both the arrays**

For each string in the second array, we are traversing the first array. Thus, the final time complexity will be  **$O(N * M)$** .

**Space Complexity:  $O(K)$ , where  $K = \min(N, M)$** —that is, **K** is the minimum of the sizes of both the arrays. Additional space is used to store the answer—that is, the common strings of both the arrays. The number of common strings of both the arrays could be at max,  **$K = \min(N, M)$** , and thus, the final space complexity is  **$O(K)$** .

**Approach 2: Using Hashmap**

The main idea behind this approach is to use a hashmap to store the count of the strings present in the first array.

### Steps:

1. Traverse the first array and for every string in the first array, do the following:
  - a. If the string is not already present in the hashmap, insert it in the hashmap with a count of 1.
  - b. Else increment the count of the string in the hashmap by 1.
2. Now, traverse the second array and for every string, do the following:
  - a. If the string is present in the hashmap, this string is a common element of both the arrays. Add this string to the **answer** array and decrement its count in the hashmap.  
Now, if the count becomes 0, this means that all the instances of the current string in the first array have been considered. So, delete this string from the hashmap.
  - b. If the string is not present in the hashmap, then this string is not present in the first array and will not be a part of the **answer** array .
3. After both the traversals, the **answer** array will have all the common strings in the order in which they appear in the second array. Thus, we need to return the answer simply.

**Time Complexity:**  $O(K * S)$ , where  $K = \max(N, M)$ —that is,  $K$  is the maximum of the sizes of both the arrays and  $S$  is the average size of the strings. The traversal of an array takes linear time. Also, inserting a string in the hashmap will take  $O(S)$  time on average, where  $S$  is the size of the string.

Thus, the final time complexity will be  $O(K * S)$ , where  $K = \max(N, M)$ .

**Space Complexity:**  $O(N)$ , where  $N$  is the size of the first array. We are storing all the strings of the first array in the hashmap. Thus, the final space complexity is  $O(N)$ .

### Approach 3: Using Trie

The main idea behind this approach is to use a trie to store the frequencies of the strings in the first array. We add an additional field **count** in the implementation of the node of the trie. This field will be equal to the number/occurrences of the current string in the first array. For the nodes that do not represent the end of a string, the value of the count will be zero.

### Steps:

1. We traverse the first array and insert the strings in a trie.
2. While inserting a string in the trie, we increment the variable **count** of the last node—that is, the node representing the last character of the string.
3. After the creation of the trie, we traverse the second array, and for each string in the second array, we check if it is present in the trie.
4. If the string exists in the trie, we add it to the **answer** array and decrement the variable **count** for the last node—that is, the node representing the last character of the string.
5. After the traversal of the second array, the **answer** array will contain the common strings in the order in which they appear in the second array.

**Time Complexity:**  $O(K * S)$ , where  $K = \max(N, M)$  and  $S$  is the average length of strings  
Both inserting and searching operations of a string in a trie take  $O(S)$  time, respectively, where  $S$  is the length of the string.  
Thus, inserting  $N$  strings of the first array will take  $O(N * S)$  time and searching for  $M$  strings of the second array will take  $O(M * S)$  time.  
Thus, the final time complexity will be  $O(N * S + M * S) = O((N + M) * S) = O(K * S)$  where  $K = \max(N, M)$ .

**Space Complexity:**  $O(26 * N * S)$ , where  $N$  denotes the size of the first array and  $S$  is the average length of the strings. For every string in the first array of length  $S$ , we will create  $S$  nodes, and every node will have an array of size 26 (to store the children of the node). Since there are  $N$  strings in the first array, the final space complexity will be  $O(26 * N * S)$ .

## 4. Count Distinct Substrings [<https://coding.ninja/P125>]

**Problem Statement:** Given a string  $S$ , you are supposed to return the number of distinct substrings (including empty substring) of the given string.

**Note:**

A string  $B$  is said to be a substring of a string  $A$  if  $B$  can be obtained by deletion of zero or more characters from the start or end of  $A$ .

Two strings  $X$  and  $Y$  are considered different if there is at least one index  $i$  such that the character of  $X$  at index  $i$  is different from the character of  $Y$  at index  $i$ , that is,  $(X[i] \neq Y[i])$ .

**Input Format :**

The **first line** contains a single integer  $T$  denoting the number of test cases.

The test cases follow.

The **first line** of each test case contains a string.

**Output Format:**

For each test case, print the number of distinct substrings of the given string.

**Sample Input**

1

sds

**Sample Output:**

6

**Explanation:**

In the above test case, the 6 distinct substrings are { 's', 'd', "sd", "ds", "sds", "" }

**Approach 1: HashSet based Approach**

The idea is to find all substrings and insert each substring into a **HashSet**. Since there will be no duplicates possible into the **HashSet**, after we have inserted all the substrings in the **HashSet**, the size of the **HashSet** will be equal to the number of distinct substrings.

#### Steps:

1. Iterate through the current string and pivot each index; let's say the pivoted index is  $i$ , as the starting index of the current substring.
  - a. Iterate from the  $i$  till the end of the loop; let's say the current index is  $j$ .
    - i. Insert the substring from index  $i$  to index  $j$  into the **HashSet**.
2. Return the size of the **HashSet** + 1(extra 1 for empty substring), as that will be equal to the number of total distinct substrings in the given string.

**Time Complexity:**  $O(|S|^3)$ , where  $|S|$  is the length of the given string. We are iterating through the given string and pivoting each element. There will be  $|S|$  elements to pivot, and for each pivot, we are iterating till the end of the string and inserting each substring into the **HashSet**. Calculating the substring will take  $O(|S|)$  time. Also, the insert operations in the **HashSet** will take time equal to the length of the string being inserted. Thus, the total time complexity will be  $O(|S|^3)$ .

**Space Complexity:**  $O(|S|^2)$ , where  $|S|$  is the length of the given string. We are inserting each substring into the **HashSet**. So in the worst case, when all the characters in the string are distinct, there will be  $|S|^2$  strings in the **HashSet**. Thus, the overall space complexity will be  $O(|S|^2)$ .

#### Approach 2: Trie(HashMap) based Approach

The idea here is to build a trie for all suffixes of the given string.

We will make use of the fact that every substring of **S** can be represented as a prefix of some suffix string of **S**.

Hence, if we create a trie of all suffixes of **S**, then the number of distinct substrings will be equal to the total number of nodes in the trie. This is because, for every substring, there will be only one path from the root node of the trie to the last character of the substring; hence no duplicate substrings will be present in the trie.

#### Steps:

1. Iterate through the given string; let's say our current index is  $i$ .
  - a. Insert the suffix starting at index  $i$  of the given string into the trie.
2. Now to count the number of nodes in the trie, we use recursion.
3. We define a function **countNodes(root)** where **root** denotes the current node of the trie. This function will return the total number of nodes in the subtree of the **root**.
  - a. Initialise a variable **subtreeNodes** to one; we will use this variable to count the number of nodes in the subtree of the current node. We are initialising the variable to one because the current root node is also counted in the subtree.

- b. Iterate through the children nodes of the current node; let's say we are currently at the  $i^{\text{th}}$  child.
    - i. If the current child node is not null, then recursively call the function for all children of the current node, i.e. **countNodes(root → child[i])** and add the value returned by this function call to **subtreeNodes**.
  - c. Return the value of **subtreeNodes**.
4. Return the number of trie nodes as this will be the number of distinct substrings in the given string.

**Time Complexity:**  $O(|S|^2)$ , where  $|S|$  is the length of the given string. We are inserting each suffix of the given string into the trie. There will be  $|S|$  different suffixes, and inserting each suffix will take  $O(|S|)$  time. Thus, the total time complexity will be  $O(|S|^2)$ .

**Space Complexity:**  $O(|S|^2)$ , where  $|S|$  is the length of the given string. We are inserting each suffix into a trie. So in the worst case, when all the characters in the string are distinct, there will be  $|S|^2$  nodes in the trie. Thus, the overall space complexity will be  $O(|S|^2)$ .

## 5. Implement indexOf() [<https://coding.ninja/P126>]

**Problem Statement:** You are given two strings, **A** and **B**. Find the index of the first occurrence of string **A** in string **B**. If string **A** is not present in string **B**, then return -1.

**For Example:**

**A** = "bc", **B** = "abcdabc".

String **A** is present at indices 1 and 5 (0-based indexing) in string **B**, but we will return 1 as our answer since it is the first occurrence of string **A** in string **B**.

**Input Format:**

The **first line** contains an integer **T** which denotes the number of test cases or queries to be run. Then, the **T** test cases follow.

The **first and only line** of each test case contains two strings **A** and **B**, separated by a single space.

**Output Format:**

For each test case, return the index of the first occurrence of **A** in **B**; if string **A** is not present in string **B**, then return -1.

**Sample Input :**

1  
ninjas codingninjas

**Sample Output:**

6

### **Explanation:**

For the given test case, “**ninjas**” is present at the 6th index of the string “**codingninjas**”.

### **Approach 1: Brute-force**

In this approach, we take every possible index of string **B** as the starting index for the matching sequence of string **A**.

#### **Steps:**

1. If the length of string **B** is less than the length of string **A**, then we simply return -1.
2. Now let's denote the length of string **A** as **M** and the length of string **B** as **N**.
3. Now, we run a loop from 0 to **N - M**, and for each index *i* in the range of 0 to **N - M**, we run another loop from 0 to **M**. Let's denote this inner loop counter by *j*.
4. Then match the characters at **(i + j)<sup>th</sup>** index of **B** with the *j*th index of **A**. If at any point characters mismatch, then we break the inner loop and increment *i* by 1. If all these characters match, and we are out of the inner loop, then we return *i*, as it is the index of the first occurrence of **A** in **B**.
5. If we reach the end of the outer loop, then we return -1, as **A** is not present in **B**.

**Time Complexity:** **O(N \* M)**, where **N** denotes the length of the string **B** and **M** is the length of string **A**.

In the worst case, we will be traversing the whole string **B**, and for each index of **B**, we will traverse the string **A**. Hence the time complexity is **O(N \* M)**.

**Space Complexity:** **O(1)**. Since we are not using any sort of auxiliary space, a constant space is required.

### **Approach 2: KMP Algorithm :**

In the brute force approach, for each index of the string **B**, we check for the next **M** characters, and whenever we find a mismatch, we move one index ahead in string **B**.

By doing this, we may have to check for the same characters again and again, and the time complexity increases to **O(N \* M)** in the worst case, where **M** and **N** are the lengths of strings **A** and **B**, respectively.

However, by using the [KMP algorithm](#), we precompute the **LPS** (Longest Proper Prefix that is also a Suffix) array of length **M**. So, we already know some matching characters in the next window. In this way, we avoid checking the same matching characters again and again.

#### **Steps:**

1. If the length of string **B** is less than the length of string **A**, then we simply return -1.
2. We first compute the **LPS** array. Call the function, **kmpPreProcess(A,M)**.
3. Declare two pointers *i* and *j*. Initialise both of them to 0.
4. Run a loop while *i* is less than **N** and *j* is less than **M**, where **M** and **N** are the lengths of **A** and **B**, respectively.

5. If the  $i^{th}$  character of **B** matches with the  $j^{th}$  character of **A**, then increment both  $i$  and  $j$ .
6. If the  $i^{th}$  character of **B** doesn't match with the  $j^{th}$  character of **A**, then check the value of  $j$ :
  - a. If  $j > 0$ , then  $j$  is updated to **Ips[j - 1]**. Else, increment  $i$  by one.
7. Finally, when we come out of the loop, we check for the value of  $j$ . If  $j$  is equal to **M**, then return  $i - j$ , else return **-1**.

#### Steps for calculating the **Ips** array:

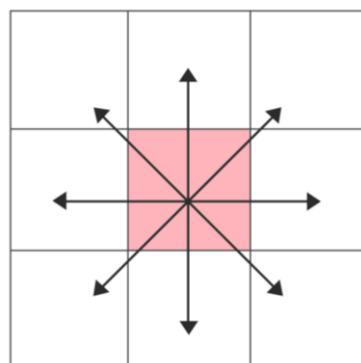
1. Create an **Ips** array of size **M**, where **M** is the length of string **A** and initialise all elements in the array to 0.
2. Keep two pointers,  $i$  and  $j$ . Initialise  $i$  as 1 and  $j$  as 0.
3. Run a loop while  $i < M$ , and do:
  - a. If the  $i^{th}$  index of **A** matches with the  $j^{th}$  index of **A**, then store the value  $j + 1$  at **Ips[i]** and increment both  $i$  and  $j$ .
  - b. If the  $i^{th}$  index of **A** doesn't match with the  $j^{th}$  index of **A**, then check whether  $j > 0$ . If  $j > 0$ , then  $j$  is updated to **Ips[j - 1]**, else mark **Ips[i]** as 0 and increment  $i$  by 1.

**Time Complexity:** **O(N + M)**, where **M** and **N** are the lengths of the string **A** and **B**, respectively. In the worst case, we will be traversing the whole string **A** and **B**, but since the **Ips** array for **A** is already calculated, the value of  $j$  just redirects to **Ips[j - 1]** if any mismatch occurs. Hence the overall complexity will be **O(N + M)**.

**Space Complexity:** **O(M)**, where **M** denotes the length of the string **A**. We create the **Ips** array of size **M**. Hence the overall space complexity will be **O(M)**.

## 6. Word Search [<https://coding.ninja/P127>]

**Problem Statement:** You are given a two-dimensional grid having **N** rows and **M** columns, consisting of upper case characters. You are also given a string **word**. You have to find the total number of occurrences of that **word** in the given grid. Starting from a given cell, a word can be formed by moving in all possible eight directions: left, right, up, down, and all four diagonal directions.



**For Example:**

There are **three** occurrences of the word '**NINJA**' in the following input grid of uppercase characters:

A	A	N	I	Q
P	J	I	N	T
N	I	N	J	A
B	L	J	I	J
P	R	A	D	N

**Note:**

1. Given **word** will not be a single character.
2. If occurrences of the given **word** start from two different cells, then these two occurrences are considered to be different.

**Input Format:**

The **first input** line contains two space-separated integers, **N** and **M**, denoting the number of rows and columns respectively in the grid.

The **next N input lines** represent the rows of the grid, containing a string, each of length **M**.

The **next input line** contains a string **WORD**, whose total occurrences are to be found in the grid.

**Output Format:**

In the only output line, print the total number of occurrences of the given string in the grid.

**Sample Input:**

5 5  
AANIQ  
PJINT  
NINJA  
BLJII  
PRADN  
NINJA

**Sample Output:**

3

**Explanation:** All the three different occurrences of the word **NINJA** are depicted in the figure below:

A	A	N	I	Q
P	J	I	N	T
N	I	N	J	A
B	L	J	I	J
P	R	A	D	N

## Approach 1: Brute-force

1. For each cell, we first check if the character in this cell matches with the first letter of the given **word**. If not, then there lies no point in checking all eight directions, so we go to the next cell.
2. To move in all eight directions, we create two arrays, **dx[]** and **dy[]**, which corresponds to the representation of these 8 directions in the **X** and **Y** axes, respectively.
3. We keep going in a particular direction as long as the string matches. If we find a mismatch, we simply break and start looking in other directions, whereas if we reach the end of the given **word**, we find an occurrence of the **word** in the grid, and thus add 1 to the answer.

**Time Complexity:**  $O(N * M * X)$ , where **N** and **M** are the number of rows and columns of the grid, and **X** is the length of the given word.

There are **N \* M** cells to consider, and for each cell, we go till a distance **X** in the worst case for all eight directions.

**Space Complexity:**  $O(1)$ . No auxiliary space is used.

## Approach 2: KMP Algorithm

Before applying this algorithm, we first compute the prefix function of the given **word**. The prefix function is described as follows:

**Ips[i]:** The maximum length of the suffix of string[0, i], which is also a proper prefix of the string. Here, the proper prefix means that we exclude the empty string and the string itself.

Creating the '**text**' strings for the rows and columns is simple.

For the diagonals:

In case of upper diagonals, all these diagonals will start from the left and top boundaries of the grid.

```
for(i=0; i<n; i++)
    for(j=0; i+j<n && j<m; j++)
        // All elements diagonally, starting from top to bottom
```

```
for(int j=1; j<m; j++)
    for(i=0; i<n && j+i<m; j++)
        // All elements diagonally, starting from left to bottom
```

In case of lower diagonals, all these diagonals will start from the left and bottom boundaries of the grid.

```
for(int i=0; i<n; i++)
    for(int j=0; i-j >= 0 && j < m; j++)
```

```

// All elements diagonally, starting from left to top

for(int j=1; j<m; j++)
    for(int i=n- 1; i >= 0 && j+(n- 1)-i < m; i--)
        // All elements diagonally, starting from bottom to top

```

Remember, we have to check for the reverse directions as well! In order to do that, we simply apply the same algorithm to the reverse of the above strings.

**Note:** The required steps to create the lps array and then using it for the string pattern matching algorithm (KMP algorithm) is discussed in the theory section as well as in the previous problems.

**Time Complexity:**  $O(N*M + X)$ , where **N** and **M** are the number of rows and columns of the grid, and **X** is the length of the given word.

Computing the prefix array for the given word takes  **$O(X)$**  time.

Considering all the rows, columns, and diagonals as **text** string will take  **$O(N * M)$**  time at maximum.

**Space Complexity:**  $O(\max(N, M) + X)$ , where **N** and **M** are the number of rows and columns of the grid and **X** is the length of the given word.

The prefix array for the given word takes  **$O(X)$**  space, and creating strings for the rows, columns, and diagonals take  **$O(\max(N, M))$**  space.

## 7. Longest Common Prefix After Rotation [\[https://coding.ninja/P128\]](https://coding.ninja/P128)

**Problem Statement:** You are given two strings, **A** and **B**. Where string **A** is fixed, but you can perform left shift operations on string **B** any number of times. Your task is to find out the minimum number of left-shift operations required in order to obtain the longest common prefix of string **A** and **B**.

For example:

If **A** = “an”, **B** = “can”,

After performing one left shift operation, string **B** becomes “anc”.

After performing two left shift operations, string **B** becomes “nca”.

### Input Format:

The first line contains an integer **T** which denotes the number of test cases or queries to be run.

Then, the **T** test cases follow.

The **first line** of each test case contains the string **A**.

The **second line** of each test case contains the string **B**.

### Output Format:

For each test case, print the minimum number of left-shift operations required in order to obtain the longest common prefix of string **A** and **B**.

### Sample Input:

1  
on  
soon

### Sample Output:

2

### Explanation:

the common prefix of **A** and **B** initially is ""

After one left shift, the string **B** becomes "**oons**", now the common prefix is "**o**".

After two left shifts, the string **B** becomes "**onso**", now the common prefix is "**on**".

After three left shifts, the string **B** becomes "**nsoo**", now the common prefix is "".

(We do not need to perform one more left shift, because if the number of left-shift operations is equal to the length of the string, then we get the original string. For example, here, if we perform one more shift, then we get the string "soon" which was the original string.)

So after two left shifts, we get the longest common—that is, "**on**". Hence, the answer is **2**.

### Approach 1: Brute-force

#### Steps:

1. Let's denote the length of **A** as **M** and the length of **B** as **N**.
2. Declare an **ans** variable to store the minimum number of left shifts required and a **max** variable in order to store the length of the longest common prefix encountered so far. Initialise the **ans** variable to 0.
3. Find out the common prefix of **A** and **B** and initialise **max** to the length of the common prefix. The length of the common prefix between two strings can be obtained by implementing a function let's say **lengthOfTheCommonPrefix**, and passing **A** and **B** as arguments to this function.
4. We run a loop from 1 to **N** – 1. Let's denote the loop counter as *i*. In each iteration, we left shift the string **B** by one character and find the length of the common prefix of **A** and **B** by calling the **lengthOfTheCommonPrefix** function. If this length is greater than the **max** variable, then update the **max** to this length and update the **ans** to *i*.
5. Finally, return the **ans** variable after the loop is over.

#### **lengthOfTheCommonPrefix(A, B):**

1. Let's denote the length of **A** as **M** and the length of **B** as **N**.
2. Initialise a **count** variable to 0, which will store the length of the common prefix of **A** and **B**.
3. Run a loop from 0 to **min(N, M)**. Let's denote the loop counter as *i*, and do:
  - a. If **A[i] != B[i]**, then break out of the loop, else increase the **count** by 1.
4. Finally, return the **count** variable after the loop ends.

**Time Complexity:**  $O(N * (N + M))$ , where **N** denotes the length of string **B** and **M** is the length of string **A**.

In the worst case, we will be traversing the whole string **B**, and for each iteration, we will perform the left-shift operation, which takes  $O(N)$  time, and then we will traverse the whole string **A** in the **lengthOfTheCommonPrefix** function, which takes  $O(M)$  time. So, the time needed for one iteration is  $O(N+M)$ . Hence, the overall time complexity is  $O(N * (N + M))$ .

**Space Complexity:**  $O(1)$ . No auxiliary space is used.

### Approach 2: KMP Algorithm

In the brute force approach, we perform the left shift operation **N - 1** number of times on the string **B** and find the length of the common prefix of **A** and **B** after each left shift operation. By doing this, we may have to check for the same characters again and again and hence, the time complexity increases to  $O(N * (N + M))$  in the worst case, where **M** and **N** are the lengths of strings **A** and **B** respectively.

By observing the property of left shift operation, we get that after **K** left shift operations, the first **K** characters of string **B** are removed and inserted to the end of the string. And we can perform maximum **N - 1** left shifts because, after the **N<sup>th</sup>** left shift, the string **B** is converted again to its original form.

So, instead of performing the left shift, again and again, we can concatenate string **B** with itself. After doing so, the only task that remains is to find out the longest prefix of string **A** that is present in string **B** and the index of this longest prefix in string **B**, which is exactly the number of shift operations needed.

For finding the index of this longest prefix, we use the KMP algorithm in which we precompute the LPS (Longest Proper Prefix that is also a Suffix) array of length **M**. So; we already know some characters in the next window.

In this way, we avoid checking the same matching characters again and again.

#### Steps:

1. Concatenate string **B** with itself. Let's denote the length of the string **A** and the string **B** as **M** and **N**, respectively.
2. Declare an **ans** variable to store the minimum number of left shifts required and a **max** variable in order to store the length of the longest common prefix encountered so far. Initialise both of them to 0.
3. We first compute the LPS array, for which we can implement another function, let's say **kmpPreProcess** and pass the string **A** and its length **M** as arguments to this function.
4. Keep two pointers, *i* and *j*. Initialise both of them to 0.
5. Run a loop while *i* is less than **N** and *j* is less than **M**, and do:
  - a. If the *i*th character of **B** matches with the *j*th character of **A**, then increment both *i* and *j*.

- b. If the  $i$ th character of **B** doesn't match with the  $j$ th character of **A**, then check the value of  $j$ .
  - c. Else If  $j > 0$ , then  $j$  is redirected to **Ips[j - 1]**, and continue the loop.
  - d. Else, increment  $i$  by 1.
  - e. If the value of  $j$  exceeds the **max** variable, then update the max to  $j$  and **ans** to  $i - j$ .
6. Finally, return the **ans** variable after the loop is over.

#### **For calculating the Ips array:**

1. Create an **Ips** array of size **M**, where **M** is the length of string **A** and initialise all elements in the array to 0.
2. Keep two pointers,  $i$  and  $j$ . Initialise  $i$  as 1 and  $j$  as 0.
3. Run a loop till  $i < M$ , and do:
  - a. If the  $i^{th}$  index of **A** matches with the  $j^{th}$  index of **A**, then store the value  $j + 1$  at **Ips[i]** and increment both  $i$  and  $j$ .
  - b. If the  $i^{th}$  index of **A** doesn't match with the  $j^{th}$  index of **A**, then check whether  $j > 0$ . If  $j > 0$ , then  $j$  redirects to **Ips[j - 1]**, else mark **Ips[i]** as 0 and increment  $i$  by 1.

**Time Complexity:**  $O(N + M)$ , where **M** and **N** are the lengths of the string **A** and **B**, respectively. In the worst case, we will be traversing the whole string **A** and **B**, but since the **Ips** array for **A** is already calculated, the value of  $j$  gets redirected to **Ips[j - 1]** whenever there is any mismatch. Hence, the overall complexity will be  $O(N + M)$ .

**Space Complexity:**  $O(M)$ , where **M** denotes the length of the string **A**. In the worst case, we are creating the **Ips** array of size **M**. Hence the overall space complexity will be  $O(M)$ .

## **8. Z-Algorithm** [<https://coding.ninja/P129>]

**Problem Statement:** You are given a string **S** of length **N** and a string **P** of length **M**. Your task is to find the number of occurrences of string **P** in the string **S**.

#### **Note:**

The string only consists of lowercase English alphabets.

#### **Input Format:**

The **first line** of input contains **T**, the number of test cases.

The **first line** of each test case contains two integers **N** and **M**, denoting the length of string **S** and **P**, respectively.

The **second line** of each test case contains the string **S**.

The **third line** of each test case contains the string **P**.

#### **Output Format:**

The only line of output of each test case should contain an integer denoting the number of occurrences of string **P** in string **S**.

**Sample Input:**

```
1
5 2
ababa
ab
```

**Sample Output:**

```
2
```

**Explanation:**

String **ab** occurs two times in the string "**ababa**".

The first occurrence is from position 1 to position 2, and the second occurrence is from position 4 to position 5.

**Approach 1: Brute Force Approach**

A basic approach is to check all substrings of string **S** and see if any of the substrings is equal to **P**.

**Steps:**

1. Initialize an integer variable **count** = 0.
2. Start by running an outer loop (loop variable *i*) from 0 to **N - M**.
3. Then run a nested loop(loop variable *j*) from 0 to **M**.
4. For index *i*, check if **S[i + j] != P[j]**; if this condition is true, we simply break the inner loop and move to the next *i*. We do this to check every substring of length **M** if it is equal to **P** or not.
5. Later on, we check if *j* is equal to **M** or not; if this condition is true, we increment our **count** by one because we have found a valid substring of length **M** which is equal to string **P**.
6. Finally, we return the **count**, which is our final answer.

**Time Complexity:**  $O(N * M)$ , where **N** and **M** are the lengths of String **S** and **P**, respectively.

Since for every substring, we are checking if it is equal to the string **P** or not; therefore, the time complexity is of  $O(N * M)$ .

**Space Complexity:**  $O(1)$ , We are not using any auxiliary space.

**Approach 2: Z-Algorithm Approach**

In this approach, we maintain a **Z array**. The **Z array** for any string **S** of length **N** is an array of size **N** where the *i*th element is equal to the greatest number of characters starting from the position *i* that coincide with the first characters of **S**.

The idea here is to concatenate both **S** and **P** and make a string **P#S**, where '#' is a special character that we are using. Now, for this concatenated string, we make a **Z** array. In this **Z** array, if the value at any index (**z[i]**) is equal to **M**, we increment our **answer** (initially = 0) by 1 because this indicates that **P** is present at that index.

### Steps:

1. Start by making a concatenated string **C**, where **C = P + # + S** and initialize an integer variable **count = 0**.
2. Make an array (**Z** array) of size **K**, where **K** is the length of string **C**.
3. Now initialize two integer values **L = 0 and R = 0**. We will be using these variables to create an interval to check if the string inside this interval matches **P** or not.
4. Now run a loop (loop variable **i**) from 1 till **K** and check if **i > R** or not.
  - a. If **i > R**. This means that there is no substring that is starting before **i** and ending after **i**. So we reset the values of **L** and **R** and calculate this new interval of **[L, R]** by using the Z array method where we use a while loop. We simply check if **C[R - L] = C[R]** and **R < K** (to check if we are still inside the string **C**), and till the time both of these conditions are met, we simply increment our **R** by 1 and finally obtain the Z array element(**z[i]**) by using **z[i] = R - L** and then we decrease **R** by 1.
  - b. If **i <= R**. Start by making another integer variable **pos = i - L**.
    - i. We first check if **z[pos]** is less than the remaining interval **(R - i + 1)** or not. If it is so then, we simply make **z[i] = z[pos]** because this means that there is no prefix substring that starts at **C[i]** because if it was, then **z[pos]** would have been larger than the remaining interval **(R - i + 1)**.
    - ii. If **z[pos]** was greater than or equal to the remaining interval **(R - i + 1)**. This condition implies that we can still extend our **[L, R]** interval. To do so, we set **L = i**. Now we start with **R** and manually check as we did in the case of **i > R** where we calculated **z[i] = R - L**, and then we decrease **R** by 1.
5. Since our **Z** array is created, we just traverse our array once and check if the current value of the array is equal to **M** or not. If it is equal to **M**, we increment the **count** by one.
6. Finally, we return **count**, which is our final answer.

Let us understand the above algorithm with the help of the following example :

**For example- String S = abba and String P = ab**

Here **N = 4 and M = 2**

1. We start by making a string **C = ab + "#" + abba = ab#abba**
2. Make a Z array, whose size is **K**, where **K** is the length of the string **C** (here, **K = 7**) and initialise it with zero.
3. Make two integer variables **L = 0** and **R = 0**.
4. Run a loop from 1 till **K** (loop variable **i**) and start by checking if **i** less **R** or not.

5. Since our  $i > R$  we reset  $L = i$  and  $R = i$ . While  $R < K(7)$  and  $C[R - L] = C[R]$  we keep incrementing  $R$  and after coming out of this while loop we calculate  $z[1]$  and since this while loop is never executed because of  $C[0] \neq C[1]$ . Hence,  $z[1] = R - L = 1 - 1 = 0$ .



A	B	#	A	B	B	A
0	0	0	0	0	0	0

6. Next for index 2 we see that  $i > R$  just like index 1. Here also since  $C[0] \neq C[2]$  we find  $z[2] = R - L = 2 - 2 = 0$ .



A	B	#	A	B	B	A
0	0	0	0	0	0	0

7. For index 3 still  $i < R$ . But here we see that  $C[0] = C[3]$ , hence we increment  $R$  by 1 and still  $C[1] = C[4]$ , hence we again increment by 1 and finally since  $C[2] \neq C[5]$  we get out of the while loop and calculate  $z[3] = R - L = 5 - 3 = 2$  and change  $R$  to 4.



A	B	#	A	B	B	A
0	0	0	2	0	0	0

8. For index 4 we see that still  $i \geq R$ . We make another variable  $pos = 4 - 3 = 1$ . And since  $z[1] < R - i + 1$ ,  $z[4] = z[1] = 0$ .



A	B	#	A	B	B	A
0	0	0	2	0	0	0

9. Similarly, for the next index, also  $z[5]=0$ .



A	B	#	A	B	B	A
0	0	0	2	0	0	0

10. For the final index  $z[6]$ . Since our  $i \geq R$ . We make another variable  $pos = 6 - 5 = 1$ . And since  $z[1] < R - i + 1$ ,  $z[6] = z[1] = 0$ .



A	B	#	A	B	B	A
0	0	0	2	0	0	1

11. Our final Z array is  $[0,0,0,2,0,0,1]$ , and in this array, there is only one element equal to  $M(2)$ , which is our answer.

**Time Complexity:**  $O(N + M)$ , where  $N$  and  $M$  are the lengths of String  $S$  and  $P$ , respectively. As we are traversing the string  $C$  only once and the length of  $C$  is  $N + M$ .

**Space Complexity:  $O(K)$** , here  $K = N+M$  , where **N** and **M** are the lengths of the string **S** and **P** respectively, as we are making an array of length **K**.

# Acing the Coding Round: 125+ Most Popular Interview Problems

## Book Description

CodingNinjas is pleased to announce its newest offering—a topic-wise curated coding problems book with an objective to prepare the aspiring students for the most frequently asked questions in the tech rounds of their coming interview process.

**Acing the Coding Round** offers the most frequently asked interview problems and the different approaches to solve them in a reader-friendly manner.

## Topics Covered

- |  |   |  |   |
|--|---|--|---|
| ● Arrays<br>● Multi-dimensional Arrays<br>● Strings<br>● Stacks and Queues | ● Linked Lists<br>● Trees<br>● Binary Search Trees<br>● Priority Queues | ● Graphs<br>● Recursion<br>● Backtracking<br>● Dynamic Programming | ● Hashmaps<br>● Circular Queues<br>● Dequeues<br>● Tries<br>● String Algorithms |
|--|---|--|---|

## Main Features of the book

- **Introductory theory** covering the essential concepts to solve the problems
- **Coding Interview Problems** curated specifically for each topic
- **Naive to most-optimised approaches** explained visually and step-by-step
- **CodeStudio links** for hands-on practice of each coding problem

## About CodeStudio

With over 2000+ problems for you to practice, CodeStudio is an online community to compete in challenges, prepare for tech interviews, and hone your skills as a programmer.

It offers guided paths on key interview topics such as DSA, Operating Systems, and aptitude preparation and has become a go-to-place to practice for the technical interview rounds and land your dream job.



## About CodingNinjas

At Coding Ninjas, our mission is to continuously innovate the best ways to train the next generation of developers and transform the way tech education is delivered.

We are a state-of-art learning platform with faculty alumni of IIT, Stanford, IIIT and Facebook teaching 17+ programming courses with over 40,000 students.