

Basics

1. Count Digits

Given an integer n, count the total number of digits present in the number.

Example

Input: n = 12345

Digits are 1, 2, 3, 4, 5 → Output: 5

Approach

Algorithm

1. If n is 0, the number of digits is 1.
2. Initialize a counter as 0.
3. Repeatedly divide n by 10.
4. Increment the counter for each division.
5. Stop when n becomes 0.
6. Return the counter.

Code

```
#include <bits/stdc++.h>

using namespace std;
```

```
int countDigits(int n) {
```

```
    if (n == 0) return 1;
```

```
    int count = 0;
```

```
    while (n != 0) {
```

```
        count++;
```

```
        n /= 10;
```

```
}
```

```

    return count;
}

int main() {
    int n = 12345;
    cout << countDigits(n);
    return 0;
}

```

Complexity Analysis

- Time Complexity: $O(\log_{10} n)$, number of digits
 - Space Complexity: $O(1)$
-

2. Reverse a Number

Given an integer n , reverse its digits.

Example

Input: $n = 1234$

Output: 4321

Approach

Algorithm

1. Initialize $rev = 0$.
2. While n is not 0:
 - o Extract last digit using $n \% 10$.
 - o Update $rev = rev * 10 + digit$.

- o Remove last digit from n using $n \text{ /= } 10$.
3. Return rev.

Code

```
#include <bits/stdc++.h>

using namespace std;

int reverseNumber(int n) {

    int rev = 0;

    while (n != 0) {

        rev = rev * 10 + (n % 10);

        n /= 10;

    }

    return rev;
}

int main() {

    int n = 1234;

    cout << reverseNumber(n);

    return 0;
}
```

Complexity Analysis

- Time Complexity: $O(\log_{10} n)$
 - Space Complexity: $O(1)$
-

3. Check Palindrome

Given an integer n , check whether it is a palindrome.

Example

Input: $n = 121$

Output: true

Approach

Algorithm

1. Store the original number.
2. Reverse the number.
3. Compare reversed number with original.
4. If equal, it is a palindrome.

Code

```
#include <bits/stdc++.h>

using namespace std;

bool isPalindrome(int n) {

    int original = n, rev = 0;

    while (n != 0) {

        rev = rev * 10 + (n % 10);

        n /= 10;
    }
}
```

```

    }

    return rev == original;

}

int main() {
    int n = 121;

    cout << isPalindrome(n);

    return 0;
}

```

Complexity Analysis

- Time Complexity: $O(\log_{10} n)$
 - Space Complexity: $O(1)$
-

4. GCD or HCF

Given two integers a and b, find their GCD (Greatest Common Divisor).

Example

Input: a = 12, b = 18

Output: 6

Approach

(Euclidean Algorithm)

Algorithm

1. While b is not 0:

- o $a = a \% b$
 - o Swap a and b
2. When b becomes 0, a is the GCD.

Code

```
#include <bits/stdc++.h>
```

```
using namespace std;
```

```
int gcd(int a, int b) {
```

```
    while (b != 0) {
```

```
        int temp = b;
```

```
        b = a % b;
```

```
        a = temp;
```

```
}
```

```
    return a;
```

```
}
```

```
int main() {
```

```
    int a = 12, b = 18;
```

```
    cout << gcd(a, b);
```

```
    return 0;
```

```
}
```

Complexity Analysis

- Time Complexity: $O(\log \min(a, b))$
 - Space Complexity: $O(1)$
-

5. Armstrong Numbers

A number is an Armstrong number if the sum of its digits raised to the power of number of digits equals the number itself.

Example

Input: $n = 153$

Explanation: $1^3 + 5^3 + 3^3 = 153$

Output: true

Approach

Algorithm

1. Count the number of digits.
2. Traverse each digit.
3. Add $\text{digit}^{\text{count}}$ to sum.
4. Compare sum with original number.

Code

```
#include <bits/stdc++.h>
```

```
using namespace std;
```

```
bool isArmstrong(int n) {
```

```
    int original = n;
```

```
    int digits = log10(n) + 1;
```

```

int sum = 0;

while (n != 0) {

    int d = n % 10;

    sum += pow(d, digits);

    n /= 10;

}

return sum == original;
}

```

```

int main() {

    int n = 153;

    cout << isArmstrong(n);

    return 0;

}

```

Complexity Analysis

- Time Complexity: $O(\log_{10} n)$
- Space Complexity: $O(1)$

6. Print All Divisors

Given an integer n , print all its divisors.

Example

Input: n = 12

Output: 1 2 3 4 6 12

Approach

Algorithm

1. Loop from 1 to \sqrt{n} .
2. If i divides n:
 - o Print i
 - o If $i \neq n/i$, print n/i .

Code

```
#include <bits/stdc++.h>

using namespace std;

void printDivisors(int n) {

    for (int i = 1; i * i <= n; i++) {
        if (n % i == 0) {
            cout << i << " ";
            if (i != n / i)
                cout << n / i << " ";
        }
    }
}
```

```
int main() {  
  
    int n = 12;  
  
    printDivisors(n);  
  
    return 0;  
  
}
```

Complexity Analysis

- Time Complexity: $O(\sqrt{n})$
 - Space Complexity: $O(1)$
-

7. Check for Prime

Given an integer n , check if it is a prime number.

Example

Input: $n = 7$

Output: true

Approach

Algorithm

1. If $n \leq 1$, not prime.
2. Loop from 2 to \sqrt{n} .
3. If any divisor is found, not prime.
4. Otherwise, prime.

Code

```
#include <bits/stdc++.h>
```

```
using namespace std;
```

```

bool isPrime(int n) {

    if (n <= 1) return false;

    for (int i = 2; i * i <= n; i++) {

        if (n % i == 0)

            return false;

    }

    return true;

}

```

```

int main() {

    int n = 7;

    cout << isPrime(n);

    return 0;

}

```

Complexity Analysis

- Time Complexity: $O(\sqrt{n})$
- Space Complexity: $O(1)$

8. Understand Recursion by Print Something N Times

Given an integer N, print a message "Hello" exactly N times using recursion.

Approach

Algorithm

1. If N becomes 0, stop recursion.
2. Print "Hello".
3. Call the function again with N-1.

Code

```
#include <bits/stdc++.h>
using namespace std;

void printHello(int n) {
    if (n == 0) return;
    cout << "Hello\n";
    printHello(n - 1);
}

int main() {
    int n = 3;
    printHello(n);
    return 0;
}
```

Complexity Analysis

- Time Complexity: $O(N)$, function is called N times.
 - Space Complexity: $O(N)$, recursion stack.
-

9. Print Name N Times Using Recursion

Given an integer N, print your name N times using recursion.

Approach

Algorithm

1. If N == 0, stop.
2. Print the name.

3. Call the function with N-1.

Code

```
#include <bits/stdc++.h>
using namespace std;
```

```
void printName(int n) {
    if (n == 0) return;
    cout << "Rahul\n";
    printName(n - 1);
}
```

```
int main() {
    int n = 4;
    printName(n);
    return 0;
}
```

Complexity Analysis

- Time Complexity: O(N)
 - Space Complexity: O(N)
-

10. Print 1 to N Using Recursion

Given an integer N, print numbers from 1 to N using recursion.

Approach

Algorithm

1. If N == 0, stop.
2. First call recursion with N-1.
3. Print N.

Code

```
#include <bits/stdc++.h>
using namespace std;
```

```
void print1toN(int n) {
    if (n == 0) return;
```

```

print1toN(n - 1);
cout << n << " ";
}

int main() {
    int n = 5;
    print1toN(n);
    return 0;
}

```

Complexity Analysis

- Time Complexity: O(N)
 - Space Complexity: O(N)
-

11. Print N to 1 Using Recursion

Given an integer N, print numbers from N to 1 using recursion.

Approach

Algorithm

1. If N == 0, stop.
2. Print N.
3. Call recursion with N-1.

Code

```
#include <bits/stdc++.h>
```

```
using namespace std;
```

```
void printNto1(int n) {
    if (n == 0) return;
    cout << n << " ";
    printNto1(n - 1);
}
```

```
int main() {
    int n = 5;
    printNto1(n);
    return 0;
}
```

}

Complexity Analysis

- Time Complexity: $O(N)$
 - Space Complexity: $O(N)$
-

12. Sum of First N Numbers

Given an integer N , find the sum of first N natural numbers using recursion.

Approach

Algorithm

1. If $N == 0$, return 0.
2. Return $N + \text{sum}(N-1)$.

Code

```
#include <bits/stdc++.h>
using namespace std;

int sumN(int n) {
    if (n == 0) return 0;
    return n + sumN(n - 1);
}

int main() {
    int n = 5;
    cout << sumN(n);
    return 0;
}
```

Complexity Analysis

- Time Complexity: $O(N)$
 - Space Complexity: $O(N)$
-

13. Factorial of N Numbers

Given an integer N, calculate N! using recursion.

Approach

Algorithm

1. If $N == 0$, return 1.
2. Return $N * \text{factorial}(N-1)$.

Code

```
#include <bits/stdc++.h>
using namespace std;

int factorial(int n) {
    if (n == 0) return 1;
    return n * factorial(n - 1);
}

int main() {
    int n = 5;
    cout << factorial(n);
    return 0;
}
```

Complexity Analysis

- Time Complexity: $O(N)$
 - Space Complexity: $O(N)$
-

14. Reverse an Array

Given an array, reverse it using recursion.

Approach

Algorithm

1. Use two pointers: l and r .

2. If $l \geq r$, stop.
3. Swap $\text{arr}[l]$ and $\text{arr}[r]$.
4. Call recursion with $l+1$ and $r-1$.

Code

```
#include <bits/stdc++.h>
using namespace std;

void reverseArray(vector<int>& arr, int l, int r) {
    if (l >= r) return;
    swap(arr[l], arr[r]);
    reverseArray(arr, l + 1, r - 1);
}

int main() {
    vector<int> arr = {1, 2, 3, 4, 5};
    reverseArray(arr, 0, arr.size() - 1);
    for (int x : arr) cout << x << " ";
    return 0;
}
```

Complexity Analysis

- Time Complexity: $O(N)$
 - Space Complexity: $O(N)$
-

15. Check if a String is Palindrome or Not

Given a string, check whether it is a palindrome using recursion.

Approach

Algorithm

1. Compare first and last characters.
2. If not equal, return false.
3. Call recursion for the inner substring.

Code

```
#include <bits/stdc++.h>
using namespace std;
```

```

bool isPalindrome(string &s, int l, int r) {
    if (l >= r) return true;
    if (s[l] != s[r]) return false;
    return isPalindrome(s, l + 1, r - 1);
}

int main() {
    string s = "madam";
    cout << isPalindrome(s, 0, s.size() - 1);
    return 0;
}

```

Complexity Analysis

- Time Complexity: O(N)
 - Space Complexity: O(N)
-

16. Fibonacci Number

Given an integer N, find the Nth Fibonacci number using recursion.

Approach

Algorithm

1. If $N \leq 1$, return N .
2. Return $\text{fib}(N-1) + \text{fib}(N-2)$.

Code

```

#include <bits/stdc++.h>
using namespace std;

int fibonacci(int n) {
    if (n <= 1) return n;
    return fibonacci(n - 1) + fibonacci(n - 2);
}

int main() {
    int n = 6;
    cout << fibonacci(n);
}

```

```
    return 0;  
}
```

Complexity Analysis

- Time Complexity: $O(2^N)$, due to repeated calls.
- Space Complexity: $O(N)$, recursion stack.

17. Hashing Theory

Hashing is a technique used to **store and retrieve data efficiently**.

It maps data (keys) to a fixed-size array using a **hash function**, which allows fast access.

The main idea is:

- Convert a value into an index using a hash function
- Store the value at that index
- Retrieve it in constant time

Hashing is mainly used for **frequency counting, searching, and lookup operations**.

Example

Array: [1, 2, 2, 3, 1]

We want to know how many times each number appears.

Using hashing:

- 1 → 2 times
- 2 → 2 times
- 3 → 1 time

Approach

Algorithm

1. Create a hash table (array, map, or unordered_map).

2. Traverse the input data.
3. For each element:
 - o Use it as a key in the hash table.
 - o Increase its count.
4. The hash table now stores frequency or presence information.

Hashing helps avoid nested loops and reduces time complexity.

Code

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    int arr[] = {1, 2, 2, 3, 1};
    int n = 5;

    unordered_map<int, int> freq;

    for (int i = 0; i < n; i++) {
        freq[arr[i]]++;
    }

    for (auto it : freq) {
        cout << it.first << " -> " << it.second << endl;
    }
    return 0;
}
```

Complexity Analysis

- Time Complexity: $O(N)$ average case
 - Space Complexity: $O(N)$ for hash table
-

18. Counting Frequencies of Array Elements

Given an array of integers, count the **frequency of each element**.

Example

Input: [10, 5, 10, 15, 10, 5]

Output:

- 10 → 3
 - 5 → 2
 - 15 → 1
-

Approach

Algorithm

1. Create an unordered_map to store element → frequency.
 2. Traverse the array.
 3. For each element, increment its count in the map.
 4. Print the stored frequencies.
-

Code

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    int arr[] = {10, 5, 10, 15, 10, 5};
    int n = 6;

    unordered_map<int, int> freq;

    for (int i = 0; i < n; i++) {
        freq[arr[i]]++;
    }

    for (auto it : freq) {
        cout << it.first << " occurs " << it.second << " times" << endl;
    }
    return 0;
}
```

- Time Complexity: O(N)

- Space Complexity: $O(N)$

Reason:

- We traverse the array once.
 - Hash map stores at most N elements.
-

19. Find the Highest / Lowest Frequency Element

Given an array, find:

- The element with the **highest frequency**
- The element with the **lowest frequency**

If multiple answers exist, any one is acceptable unless specified.

Example

Input: [1, 1, 2, 2, 2, 3]

Highest frequency → 2

Lowest frequency → 3

Approach

Algorithm

1. Count frequencies using hashing.
 2. Initialize:
 - `maxFreq = -∞`
 - `minFreq = +∞`
 3. Traverse the hash map:
 - Update max frequency and corresponding element.
 - Update min frequency and corresponding element.
 4. Print both elements.
-

Code

```
#include <bits/stdc++.h>
using namespace std;
```

```

int main() {
    int arr[] = {1, 1, 2, 2, 2, 3};
    int n = 6;

    unordered_map<int, int> freq;

    for (int i = 0; i < n; i++) {
        freq[arr[i]]++;
    }

    int maxFreq = INT_MIN, minFreq = INT_MAX;
    int maxElement = -1, minElement = -1;

    for (auto it : freq) {
        if (it.second > maxFreq) {
            maxFreq = it.second;
            maxElement = it.first;
        }
        if (it.second < minFreq) {
            minFreq = it.second;
            minElement = it.first;
        }
    }

    cout << "Highest frequency element: " << maxElement << endl;
    cout << "Lowest frequency element: " << minElement << endl;

    return 0;
}

```

Complexity Analysis

- Time Complexity: $O(N)$
 - One pass for frequency counting
 - One pass over hash map
 - Space Complexity: $O(N)$
 - Hash map stores frequencies of elements
-

Sorting

20. Selection Sort

Selection sort works by **repeatedly finding the minimum element** from the unsorted part of the array and placing it at the beginning.

At every step, the array is divided into two parts:

- Left part → already sorted
- Right part → unsorted

We select the smallest element from the unsorted part and swap it with the first unsorted position.

Example

Array: [64, 25, 12, 22, 11]

Pass 1: minimum = 11 → swap with 64 → [11, 25, 12, 22, 64]

Pass 2: minimum = 12 → [11, 12, 25, 22, 64]

Pass 3: minimum = 22 → [11, 12, 22, 25, 64]

Algorithm

1. Loop from index 0 to n-2.
2. Assume the current index is the minimum.
3. Traverse the remaining array to find the actual minimum element.
4. Swap the minimum element with the current index.
5. Repeat until the array is sorted.

Code

```
#include <bits/stdc++.h>
using namespace std;

void selectionSort(vector<int>& arr) {
    int n = arr.size();
    for (int i = 0; i < n - 1; i++) {
        int minIndex = i;
        for (int j = i + 1; j < n; j++) {
```

```

        if (arr[j] < arr[minIndex])
            minIndex = j;
    }
    swap(arr[i], arr[minIndex]);
}
}

int main() {
    vector<int> arr = {64, 25, 12, 22, 11};
    selectionSort(arr);
    for (int x : arr) cout << x << " ";
    return 0;
}

```

Complexity Analysis

- **Time Complexity:** $O(n^2)$
Two nested loops always run, regardless of input.
 - **Space Complexity:** $O(1)$
Sorting is done in-place.
-

21. Bubble Sort

Bubble sort repeatedly **compares adjacent elements** and swaps them if they are in the wrong order.

After each pass, the **largest element moves to the end**, like a bubble rising up.

Example

Array: [5, 1, 4, 2, 8]

Pass 1: [1, 4, 2, 5, 8]

Pass 2: [1, 2, 4, 5, 8]

Array becomes sorted.

Algorithm

1. Repeat passes over the array.
2. Compare adjacent elements.
3. Swap if left element is greater than right.

4. After each pass, the largest element settles at the end.
5. Stop early if no swaps occur in a pass.

Code

```
#include <bits/stdc++.h>
using namespace std;

void bubbleSort(vector<int>& arr) {
    int n = arr.size();
    for (int i = 0; i < n - 1; i++) {
        bool swapped = false;
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
                swapped = true;
            }
        }
        if (!swapped) break;
    }
}

int main() {
    vector<int> arr = {5, 1, 4, 2, 8};
    bubbleSort(arr);
    for (int x : arr) cout << x << " ";
    return 0;
}
```

Complexity Analysis

- **Time Complexity:**
 - Worst / Average: $O(n^2)$
 - Best (already sorted): $O(n)$
- **Space Complexity:** $O(1)$
In-place sorting.

22. Insertion Sort

Insertion sort builds the sorted array **one element at a time**, similar to how we sort playing cards in our hand.

The current element is inserted into its correct position in the already sorted part.

Example

Array: [8, 3, 5, 2]

Insert 3 → [3, 8, 5, 2]

Insert 5 → [3, 5, 8, 2]

Insert 2 → [2, 3, 5, 8]

Algorithm

1. Start from the second element.
2. Compare it with elements before it.
3. Shift larger elements one position to the right.
4. Insert the current element at the correct position.
5. Repeat for all elements.

Code

```
#include <bits/stdc++.h>
using namespace std;

void insertionSort(vector<int>& arr) {
    int n = arr.size();
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}

int main() {
    vector<int> arr = {8, 3, 5, 2};
    insertionSort(arr);
    for (int x : arr) cout << x << " ";
    return 0;
}
```

Complexity Analysis

- **Time Complexity:**
 - Worst / Average: $O(n^2)$
 - Best (already sorted): $O(n)$
 - **Space Complexity:** $O(1)$
Sorting is done in-place.
-

Quick Comparison

Algorithm	Best Case	Average Case	Worst Case	Stable
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	No
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	Yes
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	Yes

These sorting algorithms are mainly used for **learning purposes and small datasets**.

23. Merge Sort

Merge Sort is a **divide and conquer** algorithm.

It divides the array into two halves, sorts each half recursively, and then **merges** the two sorted halves into one sorted array.

Example

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

Divide → [5, 2, 4] and [6, 1, 3]

Sort left → [2, 4, 5]

Sort right → [1, 3, 6]

Merge → [1, 2, 3, 4, 5, 6]

Algorithm

1. If the array has only one element, it is already sorted.
2. Find the middle of the array.
3. Recursively apply merge sort on left half.
4. Recursively apply merge sort on right half.
5. Merge the two sorted halves into one sorted array.

Code

```
#include <bits/stdc++.h>
using namespace std;

void merge(vector<int>& arr, int l, int m, int r) {
    int n1 = m - l + 1;
    int n2 = r - m;

    vector<int> left(n1), right(n2);

    for (int i = 0; i < n1; i++)
        left[i] = arr[l + i];
    for (int j = 0; j < n2; j++)
        right[j] = arr[m + 1 + j];

    int i = 0, j = 0, k = l;

    while (i < n1 && j < n2) {
        if (left[i] <= right[j])
            arr[k++] = left[i++];
        else
            arr[k++] = right[j++];
    }

    while (i < n1)
        arr[k++] = left[i++];

    while (j < n2)
        arr[k++] = right[j++];
}

void mergeSort(vector<int>& arr, int l, int r) {
    if (l >= r) return;

    int m = l + (r - l) / 2;
    mergeSort(arr, l, m);
```

```

    mergeSort(arr, m + 1, r);
    merge(arr, l, m, r);
}

int main() {
    vector<int> arr = {5, 2, 4, 6, 1, 3};
    mergeSort(arr, 0, arr.size() - 1);
    for (int x : arr) cout << x << " ";
    return 0;
}

```

Complexity Analysis

- **Time Complexity:** $O(n \log n)$
Array is divided into halves and merged at each level.
 - **Space Complexity:** $O(n)$
Extra space is used for temporary arrays during merging.
-

24. Recursive Bubble Sort

Recursive Bubble Sort performs the same operation as normal bubble sort, but instead of using loops, it uses **recursion**.

In each recursive call, the **largest element moves to the end**, and the problem size reduces by one.

Example

Array: [4, 3, 2, 1]

Pass 1 → [3, 2, 1, 4]

Pass 2 → [2, 1, 3, 4]

Pass 3 → [1, 2, 3, 4]

Algorithm

1. If the size of the array is 1, stop.
2. Perform one full pass to bubble the largest element to the end.
3. Recursively call bubble sort for the remaining array of size $n-1$.

Code

```
#include <bits/stdc++.h>
using namespace std;

void recursiveBubbleSort(vector<int>& arr, int n) {
    if (n == 1) return;

    for (int i = 0; i < n - 1; i++) {
        if (arr[i] > arr[i + 1])
            swap(arr[i], arr[i + 1]);
    }

    recursiveBubbleSort(arr, n - 1);
}

int main() {
    vector<int> arr = {4, 3, 2, 1};
    recursiveBubbleSort(arr, arr.size());
    for (int x : arr) cout << x << " ";
    return 0;
}
```

Complexity Analysis

- **Time Complexity:** $O(n^2)$
Same number of comparisons as iterative bubble sort.
 - **Space Complexity:** $O(n)$
Recursive call stack uses extra space.
-

25. Recursive Insertion Sort

Recursive Insertion Sort sorts the array by **recursively sorting the first $n-1$ elements** and then inserting the last element into its correct position.

Example

Array: [5, 3, 4, 1]

Sort [5, 3, 4] → [3, 4, 5]

Insert 1 → [1, 3, 4, 5]

Algorithm

1. If the size of the array is 1, stop.
2. Recursively sort the first $n-1$ elements.
3. Insert the last element into the sorted part by shifting elements.

Code

```
#include <bits/stdc++.h>
using namespace std;

void recursiveInsertionSort(vector<int>& arr, int n) {
    if (n <= 1) return;

    recursiveInsertionSort(arr, n - 1);

    int last = arr[n - 1];
    int j = n - 2;

    while (j >= 0 && arr[j] > last) {
        arr[j + 1] = arr[j];
        j--;
    }
    arr[j + 1] = last;
}

int main() {
    vector<int> arr = {5, 3, 4, 1};
    recursiveInsertionSort(arr, arr.size());
    for (int x : arr) cout << x << " ";
    return 0;
}
```

Complexity Analysis

- **Time Complexity:**
 - Worst / Average: $O(n^2)$
 - Best (already sorted): $O(n)$
 - **Space Complexity:** $O(n)$
Due to recursive call stack.
-

Summary Comparison

Algorithm	Time Complexity	Space Complexity	Stable
Merge Sort	$O(n \log n)$	$O(n)$	Yes
Recursive Bubble Sort	$O(n^2)$	$O(n)$	Yes
Recursive Insertion Sort	$O(n^2) / O(n \text{ best})$	$O(n)$	Yes

26. Quick Sort

Quick Sort is a **divide and conquer** sorting algorithm.

It works by selecting a **pivot element**, placing it at its correct position, and then recursively sorting the elements on the left and right of the pivot.

Unlike merge sort, quick sort sorts the array **in-place**.

Question Explanation

You are given an array of integers.

Your task is to sort the array using the **Quick Sort** algorithm.

The idea is:

- Pick one element as a pivot.
 - Rearrange the array so that:
 - All elements smaller than the pivot are on the left.
 - All elements greater than the pivot are on the right.
 - Recursively apply the same steps to the left and right subarrays.
-

Approach

Quick Sort (Using Lomuto Partition)

Algorithm

1. Choose the **last element** of the array as the pivot.
 2. Rearrange the array such that:
 - o Elements smaller than or equal to pivot come before it.
 - o Elements greater than pivot come after it.
 3. The pivot is now at its correct position.
 4. Recursively apply quick sort on:
 - o Left subarray (elements before pivot)
 - o Right subarray (elements after pivot)
 5. Repeat until subarray size becomes 0 or 1.
-

Example Explanation

Array: [10, 7, 8, 9, 1, 5]

Pivot = 5

After partition: [1, 5, 8, 9, 10, 7]

Pivot 5 is at correct position.

Left subarray: [1]

Right subarray: [8, 9, 10, 7]

Apply quick sort recursively to both parts until fully sorted.

Code

```
#include <bits/stdc++.h>
using namespace std;

int partition(vector<int>& arr, int low, int high) {
    int pivot = arr[high];
    int i = low - 1;

    for (int j = low; j < high; j++) {
        if (arr[j] <= pivot) {
            i++;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[high]);
    return i + 1;
}
```

```

void quickSort(vector<int>& arr, int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

int main() {
    vector<int> arr = {10, 7, 8, 9, 1, 5};
    quickSort(arr, 0, arr.size() - 1);
    for (int x : arr) cout << x << " ";
    return 0;
}

```

Complexity Analysis

- **Time Complexity:**
 - Best Case: $O(n \log n)$ (balanced partitions)
 - Average Case: $O(n \log n)$
 - Worst Case: $O(n^2)$ (already sorted array with bad pivot choice)
 - Worst case happens because the pivot always ends up at one extreme, leading to highly unbalanced partitions.
 - **Space Complexity:** $O(\log n)$
Due to recursion stack in the average case.
-

Important Notes

- Quick sort is **not stable**.
- It is usually **faster in practice** than merge sort due to low constant factors.
- Pivot selection plays a very important role in performance.
- Common pivot choices: first element, last element, random element, median of three.

Arrays

Array ko int main me define karo to garbage value se initialize hota hai, Agar globally karo to 0 se initialize hota hai.

10 power 6 is maximum length of array we can define (int main k andar)

10 power 7 is maximum length of array we can define (global declaration)

1. Find the Largest element in an array

★ APPROACH 1 → Using sort()

```
#include<bits/stdc++.h>
using namespace std;

int sortArr(vector<int>& arr) {
    sort(arr.begin(),arr.end());
    return arr[arr.size()-1];
}

int main() {
    vector<int> arr1 = {2,5,1,3,0};
    vector<int> arr2 = {8,10,5,7,9};

    cout<<"The Largest element in the array is: "<<sortArr(arr1)<<endl;
    cout<<"The Largest element in the array is: "<<sortArr(arr2);

    return 0;
}
```

Complexity Analysis

Time Complexity: O(N*log(N))

Space Complexity: O(1)

★ APPROACH 2 → Linear Scan (BEST)

```
#include <bits/stdc++.h>

using namespace std;
int findLargestElement(int arr[], int n) {
```

```

int max = arr[0];
for (int i = 0; i < n; i++) {
    if (max < arr[i]) {
        max = arr[i];
    }
}
return max;
}

int main() {
    int arr1[] = {2,5,1,3,0};
    int n = 5;
    int max = findLargestElement(arr1, n);
    cout << "The largest element in the array is: " << max << endl;

    int arr2[] = {8,10,5,7,9};
    n = 5;
    max = findLargestElement(arr2, n);
    cout << "The largest element in the array is: " << max << endl;
    return 0;
}

```

Complexity Analysis

Time Complexity: O(N)

Space Complexity: O(1)

2. Find Second Smallest and Second Largest Element in an array

★ APPROACH 1 → Sorting

```

#include<bits/stdc++.h>
using namespace std;
void getElements(int arr[],int n)
{

```

```

if(n==0 || n==1)
    cout<<-1<<" "<<-1<<endl; // edge case when only one element is present in array
sort(arr,arr+n);
int small=arr[1];
int large=arr[n-2];
cout<<"Second smallest is "<<small<<endl;
cout<<"Second largest is "<<large<<endl;
}
int main()
{
    int arr[]={1,2,4,6,7,5};
    int n=sizeof(arr)/sizeof(arr[0]);
    getElements(arr,n);
    return 0;
}

```

Complexity Analysis

Time Complexity: O(NlogN), For sorting the array

Space Complexity: O(1)

- Isme 1 aur case hai jaise 1 2 4 5 7 7, to isme 7 2nd largest nhi hai, hume n-2 se back loop run karni padegi 2nd largest k liye.

★ APPROACH 2 → Two Pass Approach (Better)

```

#include<bits/stdc++.h>
using namespace std;
void getElements(int arr[],int n)
{
    if(n==0 || n==1)
        cout<<-1<<" "<<-1<<endl; // edge case when only one element is present in array
    int small=INT_MAX,second_small=INT_MAX;
    int large=INT_MIN,second_large=INT_MIN;
    int i;
    for(i=0;i<n;i++)
    {
        small=min(small,arr[i]);
        large=max(large,arr[i]);
    }
    for(i=0;i<n;i++)
    {

```

```

        if(arr[i]<second_small && arr[i]!=small)
            second_small=arr[i];
        if(arr[i]>second_large && arr[i]!=large)
            second_large=arr[i];
    }

    cout<<"Second smallest is "<<second_small<<endl;
    cout<<"Second largest is "<<second_large<<endl;
}
int main()
{
    int arr[]={1,2,4,6,7,5};
    int n=sizeof(arr)/sizeof(arr[0]);
    getElements(arr,n);
    return 0;
}

```

Complexity Analysis

Time Complexity: O(N), We do two linear traversals in our array

Space Complexity: O(1)

★ APPROACH 3 → Optimal (Single Pass)

```

#include<bits/stdc++.h>
using namespace std;
int secondSmallest(int arr[],int n)
{
    if(n<2)
        return -1;
    int small = INT_MAX;
    int second_small = INT_MAX;
    int i;
    for(i = 0; i < n; i++)
    {
        if(arr[i] < small)
        {
            second_small = small;
            small = arr[i];
        }
    }
}

```

```

        else if(arr[i] < second_small && arr[i] != small)
    {
        second_small = arr[i];
    }
}
return second_small;
}
int secondLargest(int arr[],int n)
{
    if(n<2)
        return -1;
    int large=INT_MIN,second_large=INT_MIN;
    int i;
    for (i = 0; i < n; i++)
    {
        if (arr[i] > large)
        {
            second_large = large;
            large = arr[i];
        }

        else if (arr[i] > second_large && arr[i] != large)
        {
            second_large = arr[i];
        }
    }
    return second_large;
}

int main() {
    int arr[]={1,2,4,7,7,5};
    int n=sizeof(arr)/sizeof(arr[0]);
    int sS=secondSmallest(arr,n);
    int sL=secondLargest(arr,n);
    cout<<"Second smallest is "<<sS<<endl;
    cout<<"Second largest is "<<sL<<endl;
    return 0;
}

```

Complexity Analysis

Time Complexity: O(N), Single-pass solution

Space Complexity: O(1)

3. Check if an Array is Sorted

🚫 Approach 1: Brute Force

```
#include <bits/stdc++.h>

using namespace std;

bool isSorted(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[i])
                return false;
        }
    }
    return true;
}

int main() {

    int arr[] = {1, 2, 3, 4, 5}, n = 5;
    bool ans = isSorted(arr, n);
    if (ans) cout << "True" << endl;
    else cout << "False" << endl;
    return 0;
}
```

Complexity Analysis

Time Complexity: $O(N^2)$

Space Complexity: $O(1)$

🌟 Approach 2: Optimal ($O(N)$)

```
#include<bits/stdc++.h>

using namespace std;

bool isSorted(int arr[], int n) {
    for (int i = 1; i < n; i++) {
        if (arr[i] < arr[i - 1])
            return false;
    }
    return true;
}

int main() {

    int arr[] = {1, 2, 3, 4, 5}, n = 5;

    printf("%s", isSorted(arr, n) ? "True" : "False");

}
```

Complexity Analysis

Time Complexity: $O(N)$

Space Complexity: $O(1)$

4. Remove Duplicates in-place from Sorted Array

Input: 0 0 1 1 1 2 2 3 3 4

Output Array: 0 1 2 3 4 ...aage kuch bhi ho

k = 5

✗ APPROACH 1: Using unordered_set

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
    int removeDuplicates(vector<int>& nums) {
        unordered_set<int> seen;
        int index = 0;
        for (int num : nums) {
            // If num is not in seen, it's unique
            if (seen.find(num) == seen.end()) {
                // Add this num to the set of seen numbers
                seen.insert(num);

                // Overwrite nums[index] with this unique num
                nums[index] = num;

                // Move index forward
                index++;
            }
        }
        // Return count of unique elements
        return index;
    }
};

int main() {
    vector<int> nums = {0,0,1,1,1,2,2,3,3,4};

    Solution sol;
    int k = sol.removeDuplicates(nums);
```

```

cout << "k = " << k << "\nArray after removing duplicates: ";
for (int i = 0; i < k; i++) {
    cout << nums[i] << " ";
}
cout << endl;
}

```

Complexity Analysis

Time Complexity: O(N), We traverse the entire array and insert elements into set.

Space Complexity: O(N), additional space used to store elements in set.

★ APPROACH 2 (Optimal): Two Pointer Technique

```

#include <bits/stdc++.h>
using namespace std;

// Class to hold the solution logic
class Solution {
public:
    // Function to remove duplicates from sorted array in-place
    int removeDuplicates(vector<int>& nums) {
        // If array is empty, return 0 directly
        if (nums.empty()) return 0;

        // Pointer for the position of last unique element
        int i = 0;

        // Traverse the array starting from the second element
        for (int j = 1; j < nums.size(); j++) {
            // If current element is different from last unique element
            if (nums[j] != nums[i]) {
                // Move pointer for unique element forward
                i++;
                // Place the new unique element at the next position
                nums[i] = nums[j];
            }
        }

        // i is index of last unique element, count = i + 1
        return i + 1;
    }
}

```

```

    }
};

int main() {
    vector<int> nums = {0,0,1,1,1,2,2,3,3,4};

    Solution sol;
    int k = sol.removeDuplicates(nums);

    cout << "Unique count = " << k << "\n";
    cout << "Array after removing duplicates: ";
    for (int x = 0; x < k; x++) {
        cout << nums[x] << " ";
    }
    cout << endl;
}

```

Complexity Analysis

Time Complexity: O(N), We traverse the entire original array only once.

Space Complexity: O(1), constant additional space is used to check unique elements.

Dry run:

i=0 → nums[i]=0

j=1 → nums[1]=0 == nums[0] → skip duplicate

j=2 → nums[2]=1 != nums[0]

→ i = 1

→ nums[1] = 1

Array → 0 1 _ _ _ _

j=3 → nums[3]=1 == nums[1] → skip

j=4 → nums[4]=1 == nums[1] → skip

j=5 → nums[5]=2 != nums[1]

→ i = 2

→ nums[2] = 2

j=7 → nums[7]=3 != nums[2]

→ i=3

→ nums[3]=3

j=9 → nums[9]=4 != nums[3]

→ i=4

→ nums[4]=4

5. Left Rotate the Array by One

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

Output: [2, 3, 4, 5, 1]

✗ Approach 1: Using Extra Array (Space = O(N))

```
#include<bits/stdc++.h>
```

```
using namespace std;
void solve(int arr[], int n) {
    int temp[n];
    for (int i = 1; i < n; i++) {
        temp[i - 1] = arr[i];
    }
    temp[n - 1] = arr[0];
    for (int i = 0; i < n; i++) {
        cout << temp[i] << " ";
    }
    cout << endl;
}
int main() {
    int n=5;

    int arr[]={1,2,3,4,5};
    solve(arr, n);
}
```

Complexity Analysis

Time Complexity: $O(n)$, as we iterate through the array only once.

Space Complexity: $O(n)$ as we are using another array of size, same as the given array.

★ Approach 2 (Optimal): In-place Rotation ($O(1)$ space)

```
#include<bits/stdc++.h>
```

```

using namespace std;
void solve(int arr[], int n) {
    int temp = arr[0]; // storing the first element of array in a variable
    for (int i = 0; i < n - 1; i++) {
        arr[i] = arr[i + 1];
    }
    arr[n - 1] = temp; // assigned the value of variable at the last index
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
}
int main() {
    int n=5;

    int arr[]={1,2,3,4,5};
    solve(arr, n);
}

```

Complexity Analysis

Time Complexity: O(n), as we iterate through the array only once.

Space Complexity: O(1) as no extra space is used

6. Rotate array by K elements

Right rotate by k: elements move to the right; items falling off the end appear at the start.

e.g. `arr=[1, 2, 3, 4, 5], k=2 → [4, 5, 1, 2, 3]`

Left rotate by k: elements move to the left; first elements go to end.

e.g. `arr=[1, 2, 3, 4, 5], k=2 → [3, 4, 5, 1, 2]`

Always normalize `k`: do `k = k % n`. If `k < 0` you can treat it as left rotate (`-k`) or convert to equivalent right rotate.

1) Method A — Temporary array (easy, uses O(k) extra space)

Idea: copy the **k** tail elements (for right-rotate) into temp, shift the rest right, then copy temp to beginning.

(Right Rotation)

```
#include <iostream>
using namespace std;
void Rotatetoright(int arr[], int n, int k)
{
    if (n == 0)
        return;
    k = k % n;
    if (k > n)
        return;
    int temp[k];
    for (int i = n - k; i < n; i++)
    {
        temp[i - n + k] = arr[i];
    }
    for (int i = n - k - 1; i >= 0; i--)
    {
        arr[i + k] = arr[i];
    }
    for (int i = 0; i < k; i++)
    {
        arr[i] = temp[i];
    }
}
int main()
{
    int n = 7;
    int arr[] = {1, 2, 3, 4, 5, 6, 7};
    int k = 2;
    Rotatetoright(arr, n, k);
    cout << "After Rotating the elements to right " << endl;
    for (int i = 0; i < n; i++)
    {
        cout << arr[i] << " ";
    }
    return 0;
}
```

Time Complexity: O(n)

Space Complexity: O(k) since k array element needs to be stored in temp array

(Left Rotate)

```
#include <iostream>
using namespace std;
void Rotatetoleft(int arr[], int n, int k)
{
    if (n == 0)
        return;
    k = k % n;
    if (k > n)
        return;
    int temp[k];
    for (int i = 0; i < k; i++)
    {
        temp[i] = arr[i];
    }
    for (int i = 0; i < n - k; i++)
    {
        arr[i] = arr[i + k];
    }
    for (int i = n - k; i < n; i++)
    {
        arr[i] = temp[i - n + k];
    }
}
int main()
{
    int n = 7;
    int arr[] = {1, 2, 3, 4, 5, 6, 7};
    int k = 2;
    Rotatetoleft(arr, n, k);
    cout << "After Rotating the elements to left " << endl;
    for (int i = 0; i < n; i++)
    {
        cout << arr[i] << " ";
```

```
    }
    return 0;
}
```

Time Complexity: $O(n)$

Space Complexity: $O(k)$ since k array element needs to be stored in temp array

2) Method B — Reverse trick (recommended: $O(n)$ time, $O(1)$ space)

Idea (right rotate by k):

1. Reverse the whole array.
2. Reverse the first k elements.
3. Reverse the remaining $n-k$ elements.
That results in a right rotation.

Equivalently some implement: reverse first $n-k$, then reverse last k , then reverse whole array
— both yield same result.

Proof intuition

Reversing rearranges blocks; using three reverses moves the tail block to front while preserving internal order.

(Right Rotate)

```
#include <iostream>
using namespace std;
// Function to Reverse the array
void Reverse(int arr[], int start, int end)
{
    while (start <= end)
    {
        int temp = arr[start];
        arr[start] = arr[end];
        arr[end] = temp;
        start++;
    }
}
```

```

    end--;
}
}
// Function to Rotate k elements to right
void Rotateeletoright(int arr[], int n, int k)
{
    // Reverse first n-k elements
    Reverse(arr, 0, n - k - 1);
    // Reverse last k elements
    Reverse(arr, n - k, n - 1);
    // Reverse whole array
    Reverse(arr, 0, n - 1);
}
int main()
{
    int arr[] = {1, 2, 3, 4, 5, 6, 7};
    int n = 7;
    int k = 2;
    Rotateeletoright(arr, n, k);
    cout << "After Rotating the k elements to right ";
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;
    return 0;
}

```

Time Complexity - O(N) where N is the number of elements in an array

Space Complexity - O(1) since no extra space is required

(Left Rotate)

```

#include <iostream>
using namespace std;
// Function to Reverse the array
void Reverse(int arr[], int start, int end)
{
    while (start <= end)
    {
        int temp = arr[start];
        arr[start] = arr[end];
        arr[end] = temp;
        start++;
    }
}

```

```

        end--;
    }
}
// Function to Rotate k elements to left
void Rotateeletoleft(int arr[], int n, int k)
{
    // Reverse first k elements
    Reverse(arr, 0, k - 1);
    // Reverse last n-k elements
    Reverse(arr, k, n - 1);
    // Reverse whole array
    Reverse(arr, 0, n - 1);
}
int main()
{
    int arr[] = {1, 2, 3, 4, 5, 6, 7};
    int n = 7;
    int k = 2;
    Rotateeletoleft(arr, n, k);
    cout << "After Rotating the k elements to left ";
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;
    return 0;
}

```

Time Complexity - O(N) where N is the number of elements in an array

Space Complexity - O(1) since no extra space is required

7. Move all Zeros to the end of the array

Input: 0 1 0 3 12

Output: 1 3 12 0 0

★ APPROACH 1 — Using Extra Array (Brute Force)

✓ Intuition

- Ek temp array banao (all 0s initially)
- Non-zero numbers ko left side me fill karo
- Baad me temp array ko original me copy kar do

```
#include <bits/stdc++.h>
using namespace std;

// Solution class
class Solution {
public:
    // Function to move all zeroes to end
    vector<int> moveZeroes(vector<int>& arr) {
        // Create temp array
        vector<int> temp(arr.size(), 0);

        // Pointer to fill temp
        int index = 0;

        // Traverse input array
        for (int i = 0; i < arr.size(); i++) {
            // If non-zero, add to temp
            if (arr[i] != 0) {
                temp[index] = arr[i];
                index++;
            }
        }

        // Copy back temp to original
        for (int i = 0; i < arr.size(); i++) {
            arr[i] = temp[i];
        }

        // Return updated array
        return arr;
    }
};

// Main function
int main() {
    vector<int> arr = {0, 1, 0, 3, 12};
    Solution sol;
```

```

vector<int> result = sol.moveZeroes(arr);

// Print result
cout << "Array after moving zeroes: ";
for (int num : result) {
    cout << num << " ";
}
cout << endl;
return 0;
}

```

Time Complexity: O(N), we can move all zeroes to end in linear time.

Space Complexity: O(N), additional space used for temporary array.

★ APPROACH 2 — Optimal Two-Pointer / Swapping (O(1) space)

✓ Intuition (Simple)

first zero ka index chahiye:

Phir array left-to-right traverse karo:

- Jab bhi non-zero mile
 - usse j-th index (first zero position) ke sath swap kar do
 - j++ (next zero ka index update ho jayega).

This keeps relative order preserved.

```

#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
    // Function to move zeroes to the end
    void moveZeroes(vector<int>& nums) {
        // Pointer to the first zero
        int j = -1;

        // Find the first zero

```

```

for (int i = 0; i < nums.size(); i++) {
    if (nums[i] == 0) {
        j = i;
        break;
    }
}

// If no zero found, return
if (j == -1) return;

// Start from the next index of first zero
for (int i = j + 1; i < nums.size(); i++) {
    // If current element is non-zero
    if (nums[i] != 0) {
        // Swap with nums[j]
        swap(nums[i], nums[j]);
        // Move j to next zero
        j++;
    }
}
};

int main() {
    Solution sol;
    vector<int> nums = {0, 1, 0, 3, 12};
    sol.moveZeroes(nums);

    // Print the result
    for (int num : nums) cout << num << " ";
    cout << endl;
    return 0;
}

```

Time Complexity: O(N), we can move all zeroes to end in linear time.

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

8. Linear Search in C

```
#include<stdio.h>

int search(int arr[],int n,int num)
{
    int i;
    for(i=0;i<n;i++)
    {
        if(arr[i]==num)
            return i;
    }
    return -1;
}
int main()
{
    int arr[]={1,2,3,4,5};
    int num = 4;
    int n = sizeof(arr)/sizeof(arr[0]);
    int val = search(arr,n,num);
    printf("%d",val);
}
```

Time Complexity: $O(n)$, where n is the length of the array.

Space Complexity: $O(1)$

9. Union of Two Sorted Arrays

Approach A — **map / set** (easy, uses tree)

```
#include <bits/stdc++.h>

using namespace std;
```

```

vector < int > FindUnion(int arr1[], int arr2[], int n, int m) {
    map < int, int > freq;
    vector < int > Union;
    for (int i = 0; i < n; i++)
        freq[arr1[i]]++;
    for (int i = 0; i < m; i++)
        freq[arr2[i]]++;
    for (auto & it: freq)
        Union.push_back(it.first);
    return Union;
}

int main() {
    int n = 10, m = 7;
    int arr1[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int arr2[] = {2, 3, 4, 4, 5, 11, 12};
    vector < int > Union = FindUnion(arr1, arr2, n, m);
    cout << "Union of arr1 and arr2 is " << endl;
    for (auto & val: Union)
        cout << val << " ";
    return 0;
}

```

Time Complexity : $O((m+n)\log(m+n))$. Inserting a key in map takes $\log N$ times, where N is no of elements in map. At max map can store $m+n$ elements {when there are no common elements and elements in arr,arr2 are distinct}. So Inserting $m+n$ th element takes $\log(m+n)$ time. Upon approximation across insertion of all elements in worst it would take $O((m+n)\log(m+n))$ time.

Using HashMap also takes the same time, On average insertion in unordered_map takes $O(1)$ time but sorting the union vector takes $O((m+n)\log(m+n))$ time. Because at max union vector can have $m+n$ elements.

Space Complexity : $O(m+n)$ {If Space of Union ArrayList is considered}

$O(1)$ {If Space of union ArrayList is not considered}

Now using set

```
#include <bits/stdc++.h>

using namespace std;
```

```

vector < int > FindUnion(int arr1[], int arr2[], int n, int m) {
    set < int > s;
    vector < int > Union;
    for (int i = 0; i < n; i++)
        s.insert(arr1[i]);
    for (int i = 0; i < m; i++)
        s.insert(arr2[i]);
    for (auto & it: s)
        Union.push_back(it);
    return Union;
}

int main()
{
    int n = 10, m = 7;
    int arr1[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int arr2[] = {2, 3, 4, 4, 5, 11, 12};
    vector < int > Union = FindUnion(arr1, arr2, n, m);
    cout << "Union of arr1 and arr2 is " << endl;
    for (auto & val: Union)
        cout << val << " ";
    return 0;
}

```

Time Complexity : $O((m+n)\log(m+n))$. Inserting an element in a set takes $\log N$ time, where N is no of elements in the set. At max set can store $m+n$ elements {when there are no common elements and elements in arr,arr2 are distinct}. So Inserting $m+n$ th element takes $\log(m+n)$ time. Upon approximation across inserting all elements in worst, it would take $O((m+n)\log(m+n))$ time.

Using HashSet also takes the same time, On average insertion in unordered_set takes $O(1)$ time but sorting the union vector takes $O((m+n)\log(m+n))$ time. Because at max union vector can have $m+n$ elements.

Space Complexity : $O(m+n)$ {If Space of Union ArrayList is considered}

$O(1)$ {If Space of union ArrayList is not considered}

Approach (BEST when inputs are sorted) — **Two-pointer merge**

```
#include <bits/stdc++.h>

using namespace std;

vector < int > FindUnion(int arr1[], int arr2[], int n, int m) {
    int i = 0, j = 0; // pointers
    vector < int > Union; // Union vector
    while (i < n && j < m) {
        if (arr1[i] <= arr2[j]) // Case 1 and 2
        {
            if (Union.size() == 0 || Union.back() != arr1[i])
                Union.push_back(arr1[i]);
            i++;
        } else // case 3
        {
            if (Union.size() == 0 || Union.back() != arr2[j])
                Union.push_back(arr2[j]);
            j++;
        }
    }
    while (i < n) // If any element left in arr1
    {
        if (Union.back() != arr1[i])
            Union.push_back(arr1[i]);
        i++;
    }
    while (j < m) // If any elements left in arr2
    {
        if (Union.back() != arr2[j])
            Union.push_back(arr2[j]);
        j++;
    }
    return Union;
}

int main()

{
    int n = 10, m = 7;
    int arr1[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int arr2[] = {2, 3, 4, 4, 5, 11, 12};
```

```

vector < int > Union = FindUnion(arr1, arr2, n, m);
cout << "Union of arr1 and arr2 is " << endl;
for (auto & val: Union)
    cout << val << " ";
return 0;
}

```

Time Complexity: $O(m+n)$, Because at max i runs for n times and j runs for m times. When there are no common elements in arr1 and arr2 and all elements in arr1, arr2 are distinct.

Space Complexity : $O(m+n)$ {If Space of Union ArrayList is considered}

$O(1)$ {If Space of union ArrayList is not considered}

If arrays are already sorted → always use two-pointer.

If arrays unsorted and you need sorted union → either use set (simple) or sort both then two-pointer.

Intersection of Two Sorted Arrays

1) Brute Force ($O(n_1 \cdot n_2)$) — visited array ke saath

(A) Multiplicity (e.g., [1,2,2,3] & [2,2,2] → [2,2])

```

#include <bits/stdc++.h>
using namespace std;

// arr1 size n1, arr2 size n2 (SORTED assumed, but brute force works anyway)
vector<int> intersectionBruteMultiplicity(int arr1[], int n1, int arr2[], int n2) {
    vector<int> res;
    vector<int> visited(n2, 0); // mark matched indices in arr2

    for (int i = 0; i < n1; i++) {
        for (int j = 0; j < n2; j++) {
            if (!visited[j] && arr1[i] == arr2[j]) {
                res.push_back(arr1[i]);
                visited[j] = 1; // this arr2[j] consumed
                break; // move to next arr1[i]
            }
            if (arr2[j] > arr1[i]) break; // small optimization if sorted
        }
    }
}

```

```

    }
    return res;
}

```

- **Time:** $O(n_1 \cdot n_2)$
- **Space:** $O(n_2)$

(B) Unique intersection (deduplicate result)

```

vector<int> intersectionBruteUnique(int arr1[], int n1, int arr2[], int n2) {
    vector<int> res;
    vector<int> visited(n2, 0);

    for (int i = 0; i < n1; i++) {
        // skip internal duplicates from arr1 to avoid re-adding same value
        if (i > 0 && arr1[i] == arr1[i-1]) continue;

        for (int j = 0; j < n2; j++) {
            if (!visited[j] && arr1[i] == arr2[j]) {
                res.push_back(arr1[i]);
                // we could mark visited[j]=1, but for unique result we can break directly
                break;
            }
            if (arr2[j] > arr1[i]) break;
        }
    }
    return res;
}

```

2) Two-Pointer ($O(n_1+n_2)$) — BEST for sorted arrays

(A) Unique intersection

```

#include <bits/stdc++.h>
using namespace std;

vector<int> intersectionTwoPointerUnique(int a[], int n1, int b[], int n2) {

```

```

int i = 0, j = 0;
vector<int> res; res.reserve(min(n1, n2));

while (i < n1 && j < n2) {
    if (a[i] == b[j]) {
        // push only once per value
        if (res.empty() || res.back() != a[i]) res.push_back(a[i]);
        // skip duplicates in both arrays
        int val = a[i];
        while (i < n1 && a[i] == val) i++;
        while (j < n2 && b[j] == val) j++;
    } else if (a[i] < b[j]) {
        i++;
    } else {
        j++;
    }
}
return res;
}

```

Time: O(n1 + n2)

Space: O(1) extra (output ke alawa)

(B) Multiplicity (counted intersection)

```

vector<int> intersectionTwoPointerMultiplicity(int a[], int n1, int b[], int n2) {
    int i = 0, j = 0;
    vector<int> res; res.reserve(min(n1, n2));

    while (i < n1 && j < n2) {
        if (a[i] == b[j]) {
            res.push_back(a[i]);
            i++; j++;
        } else if (a[i] < b[j]) {
            i++;
        } else {
            j++;
        }
    }
    return res;
}

```

Time: $O(n_1 + n_2)$

Space: $O(1)$ extra

Arrays sorted → Two-pointer (fastest, cleanest)

Unique result chahiye → skip duplicates (as shown)

Multiplicity chahiye → equal pe push, both pointers ++ (no extra dedup)

10. Find the missing number in an array

1) Brute force (nested loops) —

- For each i in $1 \dots N$, linearly search in array

```
#include <bits/stdc++.h>
using namespace std;

int missingNumber(vector<int>&a, int N) {

    // Outer loop that runs from 1 to N:
    for (int i = 1; i <= N; i++) {

        // flag variable to check
        // if an element exists
        int flag = 0;

        //Search the element using linear search:
        for (int j = 0; j < N - 1; j++) {
            if (a[j] == i) {

                // i is present in the array:
                flag = 1;
            }
        }
    }

    return flag;
}
```

```

        break;
    }
}

// check if the element is missing
// i.e flag == 0:

if (flag == 0) return i;
}

// The following line will never execute.
// It is just to avoid warnings.
return -1;
}

int main()
{
    int N = 5;
    vector<int> a = {1, 2, 4, 5};
    int ans = missingNumber(a, N);
    cout << "The missing number is: " << ans << endl;
    return 0;
}

```

Time Complexity: $O(N^2)$, where N = size of the array+1.

Reason: In the worst case i.e. if the missing number is N itself, the outer loop will run for N times, and for every single number the inner loop will also run for approximately N times. So, the total time complexity will be $O(N^2)$.

Space Complexity: $O(1)$ as we are not using any extra space.

2) Hashing / freq array — ✓

```

#include <bits/stdc++.h>
using namespace std;

int missingNumber(vector<int>&a, int N) {

    int hash[N + 1] = {0}; //hash array

```

```

// storing the frequencies:
for (int i = 0; i < N - 1; i++)
    hash[a[i]]++;

//checking the frequencies for numbers 1 to N:
for (int i = 1; i <= N; i++) {
    if (hash[i] == 0) {
        return i;
    }
}

// The following line will never execute.
// It is just to avoid warnings.
return -1;
}

int main()
{
    int N = 5;
    vector<int> a = {1, 2, 4, 5};
    int ans = missingNumber(a, N);
    cout << "The missing number is: " << ans << endl;
    return 0;
}

```

Time Complexity: $O(N) + O(N) \sim O(2*N)$, where $N = \text{size of the array}+1$.

Reason: For storing the frequencies in the hash array, the program takes $O(N)$ time complexity and for checking the frequencies in the second step again $O(N)$ is required. So, the total time complexity is $O(N) + O(N)$.

Space Complexity: $O(N)$, where $N = \text{size of the array}+1$. Here we are using an extra hash array of size $N+1$.

3) Sum formula — (watch overflow)

- $\text{Sum}(1..N) - \text{sum(array)} = \text{missing}$.

```
#include <bits/stdc++.h>
using namespace std;
```

```

int missingNumber(vector<int>&a, int N) {

    //Summation of first N numbers:
    int sum = (N * (N + 1)) / 2;

    //Summation of all array elements:
    int s2 = 0;
    for (int i = 0; i < N - 1; i++) {
        s2 += a[i];
    }

    int missingNum = sum - s2;
    return missingNum;
}

int main()
{
    int N = 5;
    vector<int> a = {1, 2, 4, 5};
    int ans = missingNumber(a, N);
    cout << "The missing number is: " << ans << endl;
    return 0;
}

```

Time Complexity: O(N), where N = size of array+1.

Reason: Here, we need only 1 loop to get the sum of the array elements. The loop runs for approx. N times. So, the time complexity is O(N).

Space Complexity: O(1) as we are not using any extra space.

4) XOR method — ★ Best (no overflow, O(1) space)
 $(1 \wedge 2 \wedge \dots \wedge N) \wedge (a[0] \wedge a[1] \wedge \dots) = \text{missing}$

```

#include <bits/stdc++.h>
using namespace std;

int missingNumber(vector<int>&a, int N) {

    int xor1 = 0, xor2 = 0;

```

```

for (int i = 0; i < N - 1; i++) {
    xor2 = xor2 ^ a[i]; // XOR of array elements
    xor1 = xor1 ^ (i + 1); //XOR up to [1...N-1]
}
xor1 = xor1 ^ N; //XOR up to [1...N]

return (xor1 ^ xor2); // the missing number
}

int main()
{
    int N = 5;
    vector<int> a = {1, 2, 4, 5};
    int ans = missingNumber(a, N);
    cout << "The missing number is: " << ans << endl;
    return 0;
}

```

Time Complexity: O(N), where N = size of array+1.

Reason: Here, we need only 1 loop to calculate the XOR. The loop runs for approx. N times. So, the time complexity is O(N).

Space Complexity: O(1) as we are not using any extra space.

11. Count Maximum Consecutive One's in the array

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

Output: 3

Explanation: The maximum consecutive ones are the last three 1's.

Intuition

Hum array me traverse karte hue ek counter `cnt` rakhenge,
jo batata hai abhi tak kitne consecutive 1's chal rahe hain.

- Jab bhi 1 mile → cnt++
- Jab 0 mile → reset (cnt = 0)
- Har step par maxi = max(maxi, cnt) rakhenge

End me maxi hoga **maximum continuous 1s**.

```
#include <bits/stdc++.h>

using namespace std;
class Solution {
public:
    int findMaxConsecutiveOnes(vector < int > & nums) {
        int cnt = 0;
        int maxi = 0;
        for (int i = 0; i < nums.size(); i++) {
            if (nums[i] == 1) {
                cnt++;
            } else {
                cnt = 0;
            }
            maxi = max(maxi, cnt);
        }
        return maxi;
    }
};

int main() {
    vector < int > nums = { 1, 1, 0, 1, 1, 1 };
    Solution obj;
    int ans = obj.findMaxConsecutiveOnes(nums);
    cout << "The maximum consecutive 1's are " << ans;
    return 0;
}
```

Time Complexity: O(N) since the solution involves only a single pass.

Space Complexity: O(1) because no extra space is used.

12. Find the number that appears once, and the other numbers twice

Approach 1: Brute Force (Double Loop)

Idea: Har element ke liye count nikaalo, agar count = 1 → wahi answer.

```
#include <bits/stdc++.h>
using namespace std;

int getSingleElement(vector<int> &arr) {
    // Size of the array:
    int n = arr.size();

    //Run a loop for selecting elements:
    for (int i = 0; i < n; i++) {
        int num = arr[i]; // selected element
        int cnt = 0;

        //find the occurrence using linear search:
        for (int j = 0; j < n; j++) {
            if (arr[j] == num)
                cnt++;
        }

        // if the occurrence is 1 return ans:
        if (cnt == 1) return num;
    }

    //This line will never execute
    //if the array contains a single element.
    return -1;
}

int main()
{
    vector<int> arr = {4, 1, 2, 1, 2};
```

```

int ans = getSingleElement(arr);
cout << "The single element is: " << ans << endl;
return 0;
}

```

Time Complexity: $O(N^2)$, where N = size of the given array.

Reason: For every element, we are performing a linear search to count its occurrence. The linear search takes $O(N)$ time complexity. And there are N elements in the array. So, the total time complexity will be $O(N^2)$.

Space Complexity: $O(1)$ as we are not using any extra space.

Approach 2: Hashing / Frequency Array

Idea: Store count of each number in a hash (array or map).

```

#include <bits/stdc++.h>
using namespace std;

int getSingleElement(vector<int> &arr) {

    //size of the array:
    int n = arr.size();

    // Find the maximum element:
    int maxi = arr[0];
    for (int i = 0; i < n; i++) {
        maxi = max(maxi, arr[i]);
    }

    // Declare hash array of size maxi+1
    // And hash the given array:
    vector<int> hash(maxi + 1, 0);
    for (int i = 0; i < n; i++) {
        hash[arr[i]]++;
    }

    //Find the single element and return the answer:

```

```

for (int i = 0; i < n; i++) {
    if (hash[arr[i]] == 1)
        return arr[i];
}

//This line will never execute
//if the array contains a single element.
return -1;
}

int main()
{
    vector<int> arr = {4, 1, 2, 1, 2};
    int ans = getSingleElement(arr);
    cout << "The single element is: " << ans << endl;
    return 0;
}

```

Time Complexity: $O(N)+O(N)+O(N)$, where N = size of the array

Reason: One $O(N)$ is for finding the maximum, the second one is to hash the elements and the third one is to search the single element in the array.

Space Complexity: $O(\text{maxElement}+1)$ where maxElement = the maximum element of the array.

Approach 3: Using `unordered_map`

Idea: Count frequency using hash map (dynamic key storage).

```

#include <bits/stdc++.h>
using namespace std;

int getSingleElement(vector<int> &arr) {

    //size of the array:
    int n = arr.size();

    // Declare the hashmap.
    // And hash the given array:
    map<int, int> mpp;

```

```

for (int i = 0; i < n; i++) {
    mpp[arr[i]]++;
}

//Find the single element and return the answer:
for (auto it : mpp) {
    if (it.second == 1)
        return it.first;
}

//This line will never execute
//if the array contains a single element.
return -1;
}

int main()
{
    vector<int> arr = {4, 1, 2, 1, 2};
    int ans = getSingleElement(arr);
    cout << "The single element is: " << ans << endl;
    return 0;
}

```

Time Complexity: $O(N \log M) + O(M)$, where $M = \text{size of the map i.e. } M = (N/2)+1$. $N = \text{size of the array}$.

Reason: We are inserting N elements in the map data structure and insertion takes $\log M$ time (where $M = \text{size of the map}$). So this results in the first term $O(N \log M)$. The second term is to iterate the map and search the single element. In Java, HashMap generally takes $O(1)$ time complexity for insertion and search. In that case, the time complexity will be $O(N) + O(M)$.

Note: The time complexity will be changed depending on which map data structure we are using. If we use unordered_map in C++, the time complexity will be $O(N)$ for the best and average case instead of $O(N \log M)$. But in the worst case (which rarely happens), it will be nearly $O(N^2)$.

Space Complexity: $O(M)$ as we are using a map data structure. Here $M = \text{size of the map i.e. } M = (N/2)+1$.

Approach 4: XOR Trick (Optimal 🔥)

Concept:

- $a \wedge a = 0$
- $a \wedge 0 = a$
- XOR is associative and commutative

So if you XOR all numbers, the duplicate ones cancel out, leaving the unique one.

```
#include <bits/stdc++.h>
using namespace std;

int getSingleElement(vector<int> &arr) {
    //size of the array:
    int n = arr.size();

    // XOR all the elements:
    int xorrr = 0;
    for (int i = 0; i < n; i++) {
        xorrr = xorrr ^ arr[i];
    }
    return xorrr;
}

int main()
{
    vector<int> arr = {4, 1, 2, 1, 2};
    int ans = getSingleElement(arr);
    cout << "The single element is: " << ans << endl;
    return 0;
}
```

Time Complexity: $O(N)$, where N = size of the array.

Reason: We are iterating the array only once.

Space Complexity: $O(1)$ as we are not using any extra space.

13. Longest Subarray with given Sum K(Positives)

Input: arr = [2, 3, 5, 1, 9], K = 10

Output: 3

Explanation: The subarray [2, 3, 5] has sum = 10 and length = 3.

1. Brute Force — 3 Loops

Idea

Try **every possible subarray**

and for each subarray compute sum manually.

```
#include <bits/stdc++.h>
using namespace std;

int getLongestSubarray(vector<int>& a, long long k) {
    int n = a.size(); // size of the array.

    int len = 0;
    for (int i = 0; i < n; i++) { // starting index
        for (int j = i; j < n; j++) { // ending index
            // add all the elements of
            // subarray = a[i...j]:
            long long s = 0;
            for (int K = i; K <= j; K++) {
                s += a[K];
            }

            if (s == k)
                len = max(len, j - i + 1);
        }
    }
    return len;
}

int main()
{
    vector<int> a = {2, 3, 5, 1, 9};
```

```

long long k = 10;
int len = getLongestSubarray(a, k);
cout << "The length of the longest subarray is: " << len << "\n";
return 0;
}

```

Time Complexity: $O(N^3)$ approx., where N = size of the array.

Reason: We are using three nested loops, each running approximately N times.

Space Complexity: $O(1)$ as we are not using any extra space.

2. Better Brute — 2 Loops

Idea

Instead of recalculating sum in a third loop, keep a running sum inside the inner loop.

```

#include <bits/stdc++.h>
using namespace std;

int getLongestSubarray(vector<int>& a, long long k) {
    int n = a.size(); // size of the array.

    int len = 0;
    for (int i = 0; i < n; i++) { // starting index
        long long s = 0; // Sum variable
        for (int j = i; j < n; j++) { // ending index
            // add the current element to
            // the subarray a[i...j-1]:
            s += a[j];

            if (s == k)
                len = max(len, j - i + 1);
        }
    }
    return len;
}

int main()
{

```

```

vector<int> a = {2, 3, 5, 1, 9};
long long k = 10;
int len = getLongestSubarray(a, k);
cout << "The length of the longest subarray is: " << len << "\n";
return 0;
}

```

Time Complexity: O(N²) approx., where N = size of the array.

Reason: We are using two nested loops, each running approximately N times.

Space Complexity: O(1) as we are not using any extra space.

3. Optimal (using Prefix Sum + Hash Map)



If we maintain the prefix sum up to each index,
then:

`sum(i...j) = prefix[j] - prefix[i-1]`

If `prefix[j] - k` exists before, then there's a subarray with sum = k.

We store each prefix sum in a map with its first index.

```

#include <bits/stdc++.h>
using namespace std;

int getLongestSubarray(vector<int>& a, long long k) {
    int n = a.size(); // size of the array.

    map<long long, int> preSumMap;
    long long sum = 0;
    int maxLen = 0;
    for (int i = 0; i < n; i++) {
        //calculate the prefix sum till index i:
        sum += a[i];

```

```

// if the sum = k, update the maxLen:
if (sum == k) {
    maxLen = max(maxLen, i + 1);
}

// calculate the sum of remaining part i.e. x-k:
long long rem = sum - k;

//Calculate the length and update maxLen:
if (preSumMap.find(rem) != preSumMap.end()) {
    int len = i - preSumMap[rem];
    maxLen = max(maxLen, len);
}

//Finally, update the map checking the conditions:
if (preSumMap.find(sum) == preSumMap.end()) {
    preSumMap[sum] = i;
}
}

return maxLen;
}

int main()
{
    vector<int> a = {2, 3, 5, 1, 9};
    long long k = 10;
    int len = getLongestSubarray(a, k);
    cout << "The length of the longest subarray is: " << len << "\n";
    return 0;
}

```

Time Complexity: O(N) or O(N*logN) depending on which map data structure we are using, where N = size of the array.

Reason: For example, if we are using an unordered_map data structure in C++ the time complexity will be O(N)(though in the worst case, unordered_map takes O(N) to find an element and the time complexity becomes O(N²)) but if we are using a map data structure, the time complexity will be O(N*logN). The least complexity will be O(N) as we are using a loop to traverse the array.

Space Complexity: O(N) as we are using a map data structure.

4. Sliding Window (Two Pointers) — Only for Positive Numbers

Idea

Since all numbers are positive,
we can use a sliding window that expands and shrinks intelligently.

- Add elements to window (increase `right`)
- If $\text{sum} > k \rightarrow$ shrink from left
- If $\text{sum} == k \rightarrow$ check length

```
#include <bits/stdc++.h>
using namespace std;

int getLongestSubarray(vector<int>& a, long long k) {
    int n = a.size(); // size of the array.

    int left = 0, right = 0; // 2 pointers
    long long sum = a[0];
    int maxLen = 0;
    while (right < n) {
        // if sum > k, reduce the subarray from left
        // until sum becomes less or equal to k:
        while (left <= right && sum > k) {
            sum -= a[left];
            left++;
        }

        // if sum = k, update the maxLen i.e. answer:
        if (sum == k) {
            maxLen = max(maxLen, right - left + 1);
        }

        // Move forward the right pointer:
        right++;
        if (right < n) sum += a[right];
    }

    return maxLen;
}
```

```

int main()
{
    vector<int> a = {2, 3, 5, 1, 9};
    long long k = 10;
    int len = getLongestSubarray(a, k);
    cout << "The length of the longest subarray is: " << len << "\n";
    return 0;
}

```

Time Complexity: $O(2^N)$, where N = size of the given array.

Reason: The outer while loop i.e. the right pointer can move up to index $n-1$ (the last index). Now, the inner while loop i.e. the left pointer can move up to the right pointer at most. So, every time the inner loop does not run for n times rather it can run for n times in total. So, the time complexity will be $O(2^N)$ instead of $O(N^2)$.

Space Complexity: $O(1)$ as we are not using any extra space.

Space Complexity: $O(1)$ as we are not using any extra space.

14. Longest Subarray with sum K | [Postives and Negatives]

Brute Force

```

#include <bits/stdc++.h>
using namespace std;

int getLongestSubarray(vector<int>& a, int k) {
    int n = a.size(); // size of the array.

    int len = 0;
    for (int i = 0; i < n; i++) { // starting index
        for (int j = i; j < n; j++) { // ending index
            // add all the elements of

```

```

// subarray = a[i...j]:
int s = 0;
for (int K = i; K <= j; K++) {
    s += a[K];
}

if (s == k)
    len = max(len, j - i + 1);
}
}
return len;
}

int main()
{
    vector<int> a = { -1, 1, 1 };
    int k = 1;
    int len = getLongestSubarray(a, k);
    cout << "The length of the longest subarray is: " << len << "\n";
    return 0;
}

```

Time Complexity: O(N³) approx., where N = size of the array.

Reason: We are using three nested loops, each running approximately N times.

Space Complexity: O(1) as we are not using any extra space.

Brute force se thoda behtar

```

#include <bits/stdc++.h>
using namespace std;

int getLongestSubarray(vector<int>& a, int k) {
    int n = a.size(); // size of the array.

    int len = 0;
    for (int i = 0; i < n; i++) { // starting index
        int s = 0;
        for (int j = i; j < n; j++) { // ending index
            // add the current element to

```

```

// the subarray a[i...j-1]:
s += a[j];

if (s == k)
    len = max(len, j - i + 1);
}
}
return len;
}

int main()
{
    vector<int> a = { -1, 1, 1 };
    int k = 1;
    int len = getLongestSubarray(a, k);
    cout << "The length of the longest subarray is: " << len << "\n";
    return 0;
}

```

Time Complexity: O(N²) approx., where N = size of the array.

Reason: We are using two nested loops, each running approximately N times.

Space Complexity: O(1) as we are not using any extra space.

Optimal code (use `unordered_map` for O(N) avg)

Core idea (Prefix sum + map)

Let `pref[i] = a[0] + a[1] + ... + a[i]`.

Agar kisi index `i` par `pref[i] = S` hai, aur pehle kabhi `pref[j] = S - K` mila tha ($j < i$), to subarray `(j+1 ... i)` ka sum K hoga (kyunki `pref[i] - pref[j] = K`).

Isi liye:

- Har prefix sum ka pehla index map me store karo (max length chahiye, isliye first occurrence hi best hota hai).
- Har step par `rem = pref - K` dekho: agar rem pehle dikha, to length = `i - firstIndex[rem]`.

```

#include <bits/stdc++.h>
using namespace std;

int getLongestSubarray(vector<int>& a, int k) {
    int n = a.size(); // size of the array.

    map<int, int> preSumMap;
    int sum = 0;
    int maxLen = 0;
    for (int i = 0; i < n; i++) {
        //calculate the prefix sum till index i:
        sum += a[i];

        // if the sum = k, update the maxLen:
        if (sum == k) {
            maxLen = max(maxLen, i + 1);
        }

        // calculate the sum of remaining part i.e. x-k:
        int rem = sum - k;

        //Calculate the length and update maxLen:
        if (preSumMap.find(rem) != preSumMap.end()) {
            int len = i - preSumMap[rem];
            maxLen = max(maxLen, len);
        }
    }

    //Finally, update the map checking the conditions:
    if (preSumMap.find(sum) == preSumMap.end()) {
        preSumMap[sum] = i;
    }
}

return maxLen;
}

int main()
{
    vector<int> a = { -1, 1, 1 };
    int k = 1;
    int len = getLongestSubarray(a, k);
    cout << "The length of the longest subarray is: " << len << "\n";
    return 0;
}

```

Time Complexity: $O(N)$ or $O(N \log N)$ depending on which map data structure we are using, where N = size of the array.

Reason: For example, if we are using an `unordered_map` data structure in C++ the time complexity will be $O(N)$ (though in the worst case, `unordered_map` takes $O(N)$ to find an element and the time complexity becomes $O(N^2)$) but if we are using a map data structure, the time complexity will be $O(N \log N)$. The least complexity will be $O(N)$ as we are using a loop to traverse the array.

Space Complexity: $O(N)$ as we are using a map data structure.

15. Two Sum : Check if a pair with given sum exists in Array

Input:

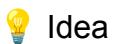
`arr = [2, 6, 5, 8, 11]`

`target = 14`

Output: YES

$(6 + 8 = 14)$

Approach 1: Brute Force (Nested Loops)



Idea

Check every pair (`i, j`) and see if their sum equals `target`.

```
#include <bits/stdc++.h>
using namespace std;

string twoSum(int n, vector<int> &arr, int target) {
```

```

for (int i = 0; i < n; i++) {
    for (int j = i + 1; j < n; j++) {
        if (arr[i] + arr[j] == target) return "YES";
    }
}
return "NO";
}

int main()
{
    int n = 5;
    vector<int> arr = {2, 6, 5, 8, 11};
    int target = 14;
    string ans = twoSum(n, arr, target);
    cout << "This is the answer for variant 1: " << ans << endl;
    return 0;
}

```

Time Complexity: O(N²), where N = size of the array.

Reason: There are two loops(i.e. nested) each running for approximately N times.

Space Complexity: O(1) as we are not using any extra space.

Variety 2:

```

#include <bits/stdc++.h>
using namespace std;

vector<int> twoSum(int n, vector<int> &arr, int target) {
    vector<int> ans;
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            if (arr[i] + arr[j] == target) {
                ans.push_back(i);
                ans.push_back(j);
                return ans;
            }
        }
    }
}

```

```

        return { -1, -1};
    }

int main()
{
    int n = 5;
    vector<int> arr = {2, 6, 5, 8, 11};
    int target = 14;
    vector<int> ans = twoSum(n, arr, target);
    cout << "This is the answer for variant 2: [" << ans[0] << ", "
        << ans[1] << "]" << endl;
    return 0;
}

```

Time Complexity: O(N²), where N = size of the array.

Reason: There are two loops(i.e. nested) each running for approximately N times.

Space Complexity: O(1) as we are not using any extra space.

Approach 2: Hash Map (Optimal)

Idea

As you traverse the array, store each element in a map.

For each number `num`, check if `target - num` exists in the map.

If yes → found a pair.

```

#include <bits/stdc++.h>
using namespace std;

string twoSum(int n, vector<int> &arr, int target) {
    unordered_map<int, int> mpp;
    for (int i = 0; i < n; i++) {
        int num = arr[i];
        int moreNeeded = target - num;
        if (mpp.find(moreNeeded) != mpp.end()) {
            return "YES";
        }
    }
}

```

```

        mpp[num] = i;
    }
    return "NO";
}

int main()
{
    int n = 5;
    vector<int> arr = {2, 6, 5, 8, 11};
    int target = 14;
    string ans = twoSum(n, arr, target);
    cout << "This is the answer for variant 1: " << ans << endl;
    return 0;
}

```

Time Complexity: $O(N)$, where N = size of the array.

Reason: The loop runs N times in the worst case and searching in a hashmap takes $O(1)$ generally. So the time complexity is $O(N)$.

Note: In the worst case(which rarely happens), the unordered_map takes $O(N)$ to find an element. In that case, the time complexity will be $O(N^2)$. If we use map instead of unordered_map, the time complexity will be $O(N * \log N)$ as the map data structure takes $\log N$ time to find an element.

Space Complexity: $O(N)$ as we use the map data structure.

Note: We have optimized this problem enough. But if in the interview, we are not allowed to use the map data structure, then we should move on to the following approach i.e. two pointer approach. This approach will have the same time complexity as the better approach.

Variety 2:

```

#include <bits/stdc++.h>
using namespace std;

vector<int> twoSum(int n, vector<int> &arr, int target) {
    unordered_map<int, int> mpp;
    for (int i = 0; i < n; i++) {
        int num = arr[i];
        int moreNeeded = target - num;
        if (mpp.find(moreNeeded) != mpp.end()) {
            return {mpp[moreNeeded], i};
        }
    }
}

```

```

        mpp[num] = i;
    }
    return { -1, -1};
}

int main()
{
    int n = 5;
    vector<int> arr = {2, 6, 5, 8, 11};
    int target = 14;
    vector<int> ans = twoSum(n, arr, target);
    cout << "This is the answer for variant 2: [" << ans[0] << ", "
        << ans[1] << "]" << endl;
    return 0;
}

```

Time Complexity: O(N), where N = size of the array.

Reason: The loop runs N times in the worst case and searching in a hashmap takes O(1) generally. So the time complexity is O(N).

Note: In the worst case(which rarely happens), the unordered_map takes O(N) to find an element. In that case, the time complexity will be O(N²). If we use map instead of unordered_map, the time complexity will be O(N * logN) as the map data structure takes logN time to find an element.

Space Complexity: O(N) as we use the map data structure.

Approach 3: Two Pointer (Sorted Array)

Idea

If the array is sorted:

- Start one pointer at beginning (`left`)
- Another at end (`right`)
- If sum < target → move `left++`
- If sum > target → move `right--`
- If sum == target → return YES

```

#include <bits/stdc++.h>
using namespace std;

string twoSum(int n, vector<int> &arr, int target) {
    sort(arr.begin(), arr.end());
    int left = 0, right = n - 1;
    while (left < right) {
        int sum = arr[left] + arr[right];
        if (sum == target) {
            return "YES";
        }
        else if (sum < target) left++;
        else right--;
    }
    return "NO";
}

int main()
{
    int n = 5;
    vector<int> arr = {2, 6, 5, 8, 11};
    int target = 14;
    string ans = twoSum(n, arr, target);
    cout << "This is the answer for variant 1: " << ans << endl;
    return 0;
}

```

Note: For variant 2, we can store the elements of the array along with its index in a new array. Then the rest of the code will be similar. And while returning, we need to return the stored indices instead of returning "YES". But for this variant, the recommended approach is approach 2 i.e. hashing approach.

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

Reason: The loop will run at most N times. And sorting the array will take $N \log N$ time complexity.

Space Complex: $O(1)$

16. Sort an array of 0s, 1s and 2s

Sorting (even if it is not the expected solution here but it can be considered as one of the approaches). Since the array contains only 3 integers, 0, 1, and 2. Simply sorting the array would arrange the elements in increasing order.

Complexity Analysis

Time Complexity: $O(N \log N)$

Space Complexity: $O(1)$

Approach 2: Counting Method (a.k.a. Counting Sort for 3 Elements)



Count how many 0s, 1s, and 2s exist, then rewrite the array based on these counts.

```
#include <bits/stdc++.h>
using namespace std;

void sortArray(vector<int>& arr, int n) {

    int cnt0 = 0, cnt1 = 0, cnt2 = 0;

    for (int i = 0; i < n; i++) {
        if (arr[i] == 0) cnt0++;
        else if (arr[i] == 1) cnt1++;
        else cnt2++;
    }

    //Replace the places in the original array:
    for (int i = 0; i < cnt0; i++) arr[i] = 0; // replacing 0's

    for (int i = cnt0; i < cnt0 + cnt1; i++) arr[i] = 1; // replacing 1's

    for (int i = cnt0 + cnt1; i < n; i++) arr[i] = 2; // replacing 2's
```

```

}

int main()
{
    int n = 6;
    vector<int> arr = {0, 2, 1, 2, 0, 1};
    sortArray(arr, n);
    cout << "After sorting:" << endl;
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
    return 0;
}

```

Output:

After sorting:

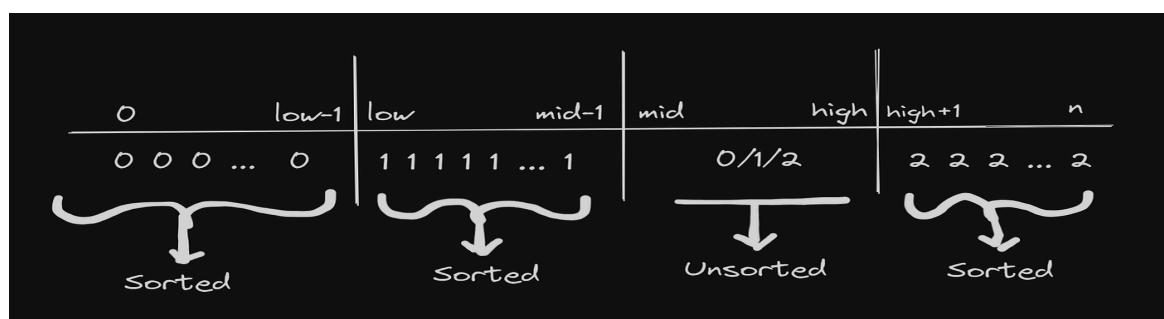
0 0 1 1 2 2

Time Complexity: $O(N) + O(N)$, where N = size of the array. First $O(N)$ for counting the number of 0's, 1's, 2's, and second $O(N)$ for placing them correctly in the original array.

Space Complexity: $O(1)$ as we are not using any extra space.

🚀 Approach 3: Dutch National Flag Algorithm (Most Optimal)

This is the **most efficient and elegant solution** — single traversal, in-place, $O(1)$ space.



```

#include <bits/stdc++.h>
using namespace std;

void sortArray(vector<int>& arr, int n) {

    int low = 0, mid = 0, high = n - 1; // 3 pointers

    while (mid <= high) {
        if (arr[mid] == 0) {
            swap(arr[low], arr[mid]);
            low++;
            Mid++; //0 k sath 1 ko bhi shi jagah lena hai
        }
        else if (arr[mid] == 1) {
            mid++;
        }
        else {
            swap(arr[mid], arr[high]);
            high--;
        }
    }
}

int main()
{
    int n = 6;
    vector<int> arr = {0, 2, 1, 2, 0, 1};
    sortArray(arr, n);
    cout << "After sorting:" << endl;
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
    return 0;
}

```

Output:

After sorting:

0 0 1 1 2 2

Time Complexity: O(N), where N = size of the given array.

Reason: We are using a single loop that can run at most N times.

Space Complexity: O(1) as we are not using any extra space.

17. Find the Majority Element that occurs more than N/2 times

1) Brute force (nested loops)

```
#include <bits/stdc++.h>
using namespace std;

int majorityElement(vector<int> v) {

    //size of the given array:
    int n = v.size();

    for (int i = 0; i < n; i++) {
        //selected element is v[i]
        int cnt = 0;
        for (int j = 0; j < n; j++) {
            // counting the frequency of v[i]
            if (v[j] == v[i]) {
                cnt++;
            }
        }

        // check if frquency is greater than n/2:
        if (cnt > (n / 2))
            return v[i];
    }

    return -1;
}

int main()
{
    vector<int> arr = {2, 2, 1, 1, 1, 2, 2};
    int ans = majorityElement(arr);
    cout << "The majority element is: " << ans << endl;
    return 0;
}
```

Output: The majority element is: 2

Time Complexity: O(N²), where N = size of the given array. Reason: For every element of the array the inner loop runs for N times. And there are N elements in the array. So, the total time complexity is O(N²). Space Complexity: O(1) as we use no extra space.

2) Hash/map counting

```
#include <bits/stdc++.h>
using namespace std;

int majorityElement(vector<int> v) {

    //size of the given array:
    int n = v.size();

    //declaring a map:
    map<int, int> mpp;

    //storing the elements with its occurrence:
    for (int i = 0; i < n; i++) {
        mpp[v[i]]++;
    }

    //searching for the majority element:
    for (auto it : mpp) {
        if (it.second > (n / 2)) {
            return it.first;
        }
    }

    return -1;
}

int main()
{
    vector<int> arr = {2, 2, 1, 1, 1, 2, 2};
    int ans = majorityElement(arr);
    cout << "The majority element is: " << ans << endl;
    return 0;
}
```

Output: The majority element is: 2

Time Complexity: $O(N \log N) + O(N)$, where N = size of the given array.

Reason: We are using a map data structure. Insertion in the map takes $\log N$ time. And we are doing it for N elements. So, it results in the first term $O(N \log N)$. The second $O(N)$ is for checking which element occurs more than $\text{floor}(N/2)$ times. If we use `unordered_map` instead, the first term will be $O(N)$ for the best and average case and for the worst case, it will be $O(N^2)$.

Space Complexity: $O(N)$ as we are using a map data structure.

3) Boyer–Moore Majority Vote (⭐ optimal)

```
#include <bits/stdc++.h>
using namespace std;

int majorityElement(vector<int> v) {

    //size of the given array:
    int n = v.size();
    int cnt = 0; // count
    int el; // Element

    //applying the algorithm:
    for (int i = 0; i < n; i++) {
        if (cnt == 0) {
            cnt = 1;
            el = v[i];
        }
        else if (el == v[i]) cnt++;
        else cnt--;
    }

    //checking if the stored element
    // is the majority element:
    int cnt1 = 0;
    for (int i = 0; i < n; i++) {
        if (v[i] == el) cnt1++;
    }
}
```

```

if (cnt1 > (n / 2)) return el;
return -1;
}

int main()
{
    vector<int> arr = {2, 2, 1, 1, 1, 2, 2};
    int ans = majorityElement(arr);
    cout << "The majority element is: " << ans << endl;
    return 0;
}

```

Output: The majority element is: 2

Time Complexity: $O(N) + O(N)$, where N = size of the given array.

Reason: The first $O(N)$ is to calculate the count and find the expected majority element. The second one is to check if the expected element is the majority one or not.

Note: If the question states that the array must contain a majority element, in that case, we do not need the second check. Then the time complexity will boil down to $O(N)$.

Space Complexity: $O(1)$ as we are not using any extra space.

18. Kadane's Algorithm : Maximum Subarray Sum in an Array

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

Output: 6

Explanation: The subarray [4, -1, 2, 1] has the largest sum = 6

- ◆ **Brute Force (3 loops)**

1. Generate every subarray (*i...j*).
2. Compute its sum using a third loop.
3. Track the maximum sum.

```

#include <bits/stdc++.h>
using namespace std;

int maxSubarraySum(int arr[], int n) {
    int maxi = INT_MIN; // maximum sum

    for (int i = 0; i < n; i++) {
        for (int j = i; j < n; j++) {
            // subarray = arr[i.....j]
            int sum = 0;

            //add all the elements of subarray:
            for (int k = i; k <= j; k++) {
                sum += arr[k];
            }

            maxi = max(maxi, sum);
        }
    }

    return maxi;
}

int main()
{
    int arr[] = { -2, 1, -3, 4, -1, 2, 1, -5, 4 };
    int n = sizeof(arr) / sizeof(arr[0]);
    int maxSum = maxSubarraySum(arr, n);
    cout << "The maximum subarray sum is: " << maxSum << endl;
    return 0;
}

```

Output: The maximum subarray sum is: 6

Time Complexity: $O(N^3)$, where N = size of the array.

Reason: We are using three nested loops, each running approximately N times.

Space Complexity: $O(1)$ as we are not using any extra space.

♦ Improved (2 loops)

Reuse previously computed subarray sums to avoid one inner loop.

```
#include <bits/stdc++.h>
using namespace std;

int maxSubarraySum(int arr[], int n) {
    int maxi = INT_MIN; // maximum sum

    for (int i = 0; i < n; i++) {
        int sum = 0;
        for (int j = i; j < n; j++) {
            // current subarray = arr[i.....j]

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

            maxi = max(maxi, sum); // getting the maximum
        }
    }

    return maxi;
}

int main()
{
    int arr[] = { -2, 1, -3, 4, -1, 2, 1, -5, 4};
    int n = sizeof(arr) / sizeof(arr[0]);
    int maxSum = maxSubarraySum(arr, n);
    cout << "The maximum subarray sum is: " << maxSum << endl;
    return 0;
}
```

Output: The maximum subarray sum is: 6

Time Complexity: O(N²), where N = size of the array.

Reason: We are using two nested loops, each running approximately N times.

Space Complexity: O(1) as we are not using any extra space.

- ◆ **Kadane's Algorithm (O(N))**

At any element `arr[i]`, we have two choices:

1. Continue the previous subarray → `sum + arr[i]`
2. Start a new subarray → `arr[i]`

Whichever gives the **larger sum**, we choose that.

Also, if at any point the running sum becomes **negative**, we reset it to 0 — because a negative prefix can never help future sums.

```
#include <bits/stdc++.h>
using namespace std;

long long maxSubarraySum(int arr[], int n) {
    long long maxi = LONG_MIN; // maximum sum
    long long sum = 0;

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

        sum += arr[i];

        if (sum > maxi) {
            maxi = sum;
        }

        // If sum < 0: discard the sum calculated
        if (sum < 0) {
            sum = 0;
        }
    }

    // To consider the sum of the empty subarray
    // uncomment the following check:

    //if (maxi < 0) maxi = 0;

    return maxi;
}

int main()
{
```

```

int arr[] = { -2, 1, -3, 4, -1, 2, 1, -5, 4};
int n = sizeof(arr) / sizeof(arr[0]);
long long maxSum = maxSubarraySum(arr, n);
cout << "The maximum subarray sum is: " << maxSum << endl;
return 0;
}

```

Output: The maximum subarray sum is: 6

Time Complexity: O(N), where N = size of the array.

Reason: We are using a single loop running N times.

Space Complexity: O(1) as we are not using any extra space.

🎯 To Print the Subarray Too

```

#include <bits/stdc++.h>
using namespace std;

long long maxSubarraySum(int arr[], int n) {
    long long maxi = LONG_MIN; // maximum sum
    long long sum = 0;

    int start = 0;
    int ansStart = -1, ansEnd = -1;
    for (int i = 0; i < n; i++) {

        if (sum == 0) start = i; // starting index

        sum += arr[i];

        if (sum > maxi) {
            maxi = sum;

            ansStart = start;
            ansEnd = i;
        }
    }

    // If sum < 0: discard the sum calculated
    if (sum < 0) {
        sum = 0;
    }
}

```

```

}

//printing the subarray:
cout << "The subarray is: [";
for (int i = ansStart; i <= ansEnd; i++) {
    cout << arr[i] << " ";
}
cout << "]n";

// To consider the sum of the empty subarray
// uncomment the following check:

//if (maxi < 0) maxi = 0;

return maxi;
}

int main()
{
    int arr[] = { -2, 1, -3, 4, -1, 2, 1, -5, 4};
    int n = sizeof(arr) / sizeof(arr[0]);
    long long maxSum = maxSubarraySum(arr, n);
    cout << "The maximum subarray sum is: " << maxSum << endl;
    return 0;
}

```

Output:

The subarray is: [4 -1 2 1]
The maximum subarray sum is: 6

Time Complexity: O(N), where N = size of the array.

Reason: We are using a single loop running N times.

Space Complexity: O(1) as we are not using any extra space.

20. Stock Buy And Sell

Given an array `arr[]` where each element represents the stock price on day `i`, find the **maximum profit** you can achieve by buying and selling **once**.

Input: [7, 1, 5, 3, 6, 4]

Output: 5

Explanation:

Buy on day 2 (price = 1), sell on day 5 (price = 6)

Profit = 6 - 1 = 5

Approach 1: Brute Force ($O(N^2)$)

- Try **all pairs** (i, j) such that $j > i$.
- Compute profit = $\text{arr}[j] - \text{arr}[i]$
- Keep track of the maximum profit.

```
#include<bits/stdc++.h>
using namespace std;

int maxProfit(vector<int> &arr) {
    int maxPro = 0;
    int n = arr.size();

    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            if (arr[j] > arr[i]) {
                maxPro = max(arr[j] - arr[i], maxPro);
            }
        }
    }

    return maxPro;
}

int main() {
    vector<int> arr = {7,1,5,3,6,4};
    int maxPro = maxProfit(arr);
    cout << "Max profit is: " << maxPro << endl;
}
```

Output:

Max profit is: 5

Time complexity: $O(n^2)$

Space Complexity: $O(1)$

Approach 2: Optimized ($O(N)$) — Kadane's style

Now think smartly 

Instead of checking every pair, we can do this in **one pass**.

Intuition

- As we traverse, keep track of the **lowest price so far** (the best day to buy).
- For each price, calculate the **potential profit** = `price - minPriceSoFar`.
- Keep track of the **maximum profit**.

```
#include<bits/stdc++.h>
using namespace std;

int maxProfit(vector<int> &arr) {
    int maxPro = 0;
    int n = arr.size();
    int minPrice = INT_MAX;

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

    return maxPro;
}

int main() {
    vector<int> arr = {7,1,5,3,6,4};
    int maxPro = maxProfit(arr);
```

```
    cout << "Max profit is: " << maxPro << endl;
}
```

Output: Max profit is: 5

Time complexity: O(n)

Space Complexity: O(1)

21. Rearrange Array Elements by Sign

- Rearrange array so **positives and negatives alternate**.
- If counts are **equal** → strictly alternate: + - + - ...
- If counts are **unequal** → alternate as much as possible, then **append leftovers** of the majority sign at the end.



Decide how to treat **0**. Our code treats **0** as **negative** (`A[i] > 0 ? pos : neg`).

Many problems either treat **0** as positive or allow either. Be consistent with the statement.

Case A: Counts are equal (strict alternation)

A1) Two-bucket, write back (simple & readable)

```
#include<bits/stdc++.h>
using namespace std;

vector<int> RearrangebySign(vector<int>A, int n){

    // Define 2 vectors, one for storing positive
    // and other for negative elements of the array.
    vector<int> pos;
    vector<int> neg;
```

```

// Segregate the array into positives and negatives.
for(int i=0;i<n;i++){
    if(A[i]>0) pos.push_back(A[i]);
    else neg.push_back(A[i]);
}

// Positives on even indices, negatives on odd.
for(int i=0;i<n/2;i++){
    A[2*i] = pos[i];
    A[2*i+1] = neg[i];
}

return A;
}

int main() {
    // Array Initialisation.
    int n = 4;
    vector<int> A {1,2,-4,-5};

    vector<int> ans = RearrangebySign(A,n);

    for (int i = 0; i < ans.size(); i++) {
        cout << ans[i] << " ";
    }

    return 0;
}

```

Output :

1 -4 2 -5

Time Complexity: $O(N+N/2)$ { $O(N)$ for traversing the array once for segregating positives and negatives and another $O(N/2)$ for adding those elements alternatively to the array, where N = size of the array A}.

Space Complexity: $O(N/2 + N/2) = O(N)$ { $N/2$ space required for each of the positive and negative element arrays, where N = size of the array A}.

A2) Single pass fill even/odd (no extra buckets for counts)

```
#include<bits/stdc++.h>
using namespace std;

vector<int> RearrangebySign(vector<int> A){

    int n = A.size();

    // Define array for storing the ans separately.
    vector<int> ans(n,0);

    // positive elements start from 0 and negative from 1.
    int posIndex = 0, negIndex = 1;
    for(int i = 0;i<n;i++){

        // Fill negative elements in odd indices and inc by 2.
        if(A[i]<0){
            ans[negIndex] = A[i];
            negIndex+=2;
        }

        // Fill positive elements in even indices and inc by 2.
        else{
            ans[posIndex] = A[i];
            posIndex+=2;
        }
    }

    return ans;
}

int main() {

    // Array Initialisation.

    vector<int> A = {1,2,-4,-5};

    vector<int> ans = RearrangebySign(A);
```

```

for (int i = 0; i < ans.size(); i++) {
    cout << ans[i] << " ";
}
return 0;
}

```

Output :

1 -4 2 -5

Time Complexity: O(N) { O(N) for traversing the array once and substituting positives and negatives simultaneously using pointers, where N = size of the array A}.

Space Complexity: O(N) { Extra Space used to store the rearranged elements separately in an array, where N = size of array A}.

Case B: Counts may be unequal

Strategy:

1. Split into **pos** and **neg**.
2. Fill alternating while both remain.
3. Append the leftover of the majority sign.

```

#include<bits/stdc++.h>
using namespace std;

vector<int> RearrangebySign(vector<int>A, int n){

// Define 2 vectors, one for storing positive
// and other for negative elements of the array.
vector<int> pos;
vector<int> neg;

// Segregate the array into positives and negatives.
for(int i=0;i<n;i++){

    if(A[i]>0) pos.push_back(A[i]);
    else neg.push_back(A[i]);
}

```

```

}

// If positives are lesser than the negatives.
if(pos.size() < neg.size()){

    // First, fill array alternatively till the point
    // where positives and negatives ar equal in number.
    for(int i=0;i<pos.size();i++){

        A[2*i] = pos[i];
        A[2*i+1] = neg[i];
    }

    // Fill the remaining negatives at the end of the array.
    int index = pos.size()*2;
    for(int i = pos.size();i<neg.size();i++){

        A[index] = neg[i];
        index++;
    }
}

// If negatives are lesser than the positives.
else{

    // First, fill array alternatively till the point
    // where positives and negatives ar equal in number.
    for(int i=0;i<neg.size();i++){

        A[2*i] = pos[i];
        A[2*i+1] = neg[i];
    }

    // Fill the remaining positives at the end of the array.
    int index = neg.size()*2;
    for(int i = neg.size();i<pos.size();i++){

        A[index] = pos[i];
        index++;
    }
}

return A;
}

```

```

int main() {

    // Array Initialisation.
    int n = 6;
    vector<int> A {1,2,-4,-5,3,4};

    vector<int> ans = RearrangebySign(A,n);

    for (int i = 0; i < ans.size(); i++) {
        cout << ans[i] << " ";
    }

    return 0;
}

```

Output :

1 -4 2 -5 3 4

Time Complexity: $O(2^*N)$ { The worst case complexity is $O(2^*N)$ which is a combination of $O(N)$ of traversing the array to segregate into neg and pos array and $O(N)$ for adding the elements alternatively to the main array}.

Explanation: The second $O(N)$ is a combination of $O(\min(\text{pos}, \text{neg})) + O(\text{leftover elements})$. There can be two cases: when only positive or only negative elements are present, $O(\min(\text{pos}, \text{neg})) + O(\text{leftover}) = O(0) + O(N)$, and when equal no. of positive and negative elements are present, $O(\min(\text{pos}, \text{neg})) + O(\text{leftover}) = O(N/2) + O(0)$. So, from these two cases, we can say the worst-case time complexity is $O(N)$ for the second part, and by adding the first part we get the total complexity of $O(2^*N)$.

Space Complexity: $O(N/2 + N/2) = O(N)$ { $N/2$ space required for each of the positive and negative element arrays, where N = size of the array A}.

22. next_permutation : find next lexicographically greater permutation

Given a sequence, find the **next** arrangement in dictionary (lexicographic) order.

If you're already at the largest arrangement (strictly decreasing), the next is the **smallest** (sorted ascending).

Brute-Force Approach

The brute force approach to find the next permutation is to find all possible permutations of the array and then look for next permutation.

- Find all possible permutations of elements present and store them.
- Sort the permutations and search input from all possible permutations. Print the next permutation present right after it. If the current permutation is the last, return the first permutation in the list.

Recursion use karenge to generate all the permutation:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

void generatePermutations(vector<vector<int>> &res,
                           vector<int> &arr, int idx) {

    // Base case: if idx reaches the end of array
    if (idx == arr.size() - 1) {
        res.push_back(arr);
        return;
    }

    // Generate all permutations by swapping
    for (int i = idx; i < arr.size(); i++) {
        swap(arr[idx], arr[i]);

        // Recur for the next index
        generatePermutations(res, arr, idx + 1);

        // Backtrack to restore original array
        swap(arr[idx], arr[i]);
    }
}
```

```

}

// Function to find the next permutation
void nextPermutation(vector<int>& arr) {

    vector<vector<int>> res;

    // Generate all permutations
    generatePermutations(res, arr, 0);

    // Sort all permutations lexicographically
    sort(res.begin(), res.end());

    // Find the current permutation index
    for (int i = 0; i < res.size(); i++) {

        // If current permutation matches input
        if (res[i] == arr) {

            // If it's not the last permutation
            if (i < res.size() - 1) {
                arr = res[i + 1];
            }
            else {
                arr = res[0];
            }

            break;
        }
    }
}

int main() {

    vector<int> arr = {2, 4, 1, 7, 5, 0};

    nextPermutation(arr);

    for (int i = 0; i < arr.size(); i++) {
        cout << arr[i] << " ";
    }
}

```

```
    return 0;  
}
```

Optimal Approach

Think of the array as: **prefix | pivot | suffix**

1. **Scan from right to left** to find the first index **i** (pivot) such that **nums[i] < nums[i+1]**.
 - o If no such **i** exists, the array is in descending order → reverse entire array and return.
2. **Find the smallest number > nums[i]** in the suffix (which is decreasing), so just scan from right to left to find the first **j** with **nums[j] > nums[i]**.
3. **Swap** **nums[i]** and **nums[j]**.
4. **Reverse** the suffix (from **i+1** to end).

Why reverse? Because the suffix was decreasing; reversing makes it the **smallest** ascending tail after the new pivot → gives the immediate next permutation.

01
Step

Identify the rightmost element that is smaller than its next element. This element is called the pivot.

0	1	2	3	4	5
2	4	1	7	5	0

pivot

02
Step

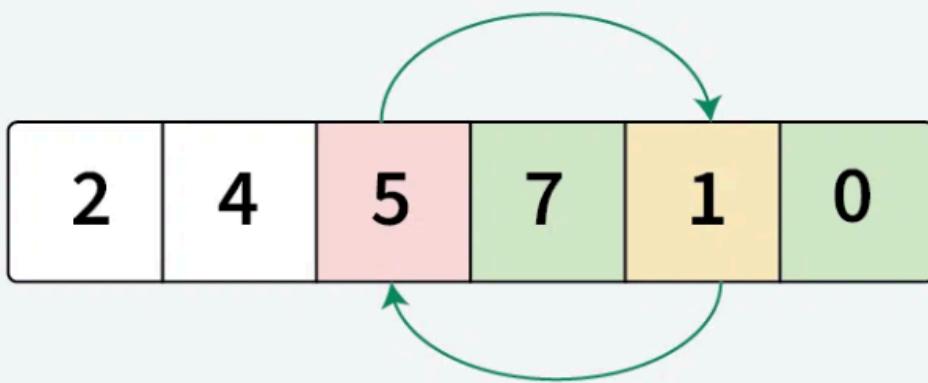
Find the rightmost element that is greater than the pivot.

0	1	2	3	4	5
2	4	1	7	5	0

pivot $5 > 1$

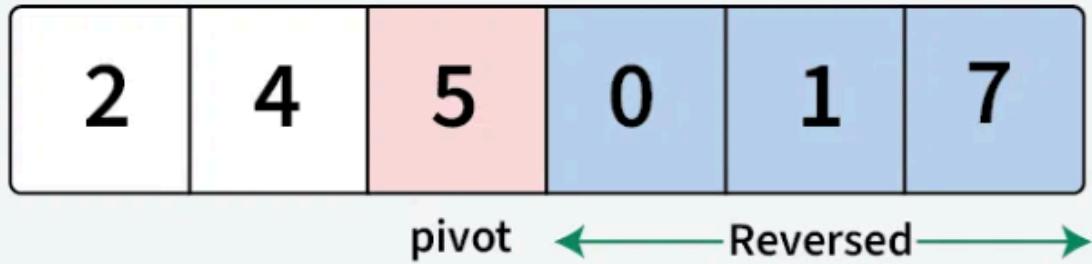
03
Step

Swap Pivot and Rightmost Greater Element



04
Step

Reverse the elements to the right of the pivot.



```
#include <bits/stdc++.h>
using namespace std;

// Solution class
class Solution {
public:
    // Function to find next permutation
    void nextPermutation(vector<int>& nums) {
        // Set index to -1
        int index = -1;

        // Find the first decreasing element from end
        for (int i = nums.size() - 2; i >= 0; i--) {
            // If a smaller element found
            if (nums[i] < nums[i + 1]) {
                // Store index
                index = i;
                break;
            }
        }

        // If no such index found
        if (index == -1) {
            // Reverse the entire array
            reverse(nums.begin(), nums.end());
        }
    }
}
```

```

        return;
    }

    // Find element just greater than nums[index]
    for (int i = nums.size() - 1; i > index; i--) {
        // Swap the two
        if (nums[i] > nums[index]) {
            swap(nums[i], nums[index]);
            break;
        }
    }

    // Reverse the part after index
    reverse(nums.begin() + index + 1, nums.end());
}

};

// Main function
int main() {
    // Input array
    vector<int> nums = {1, 2, 3};

    // Create object
    Solution sol;

    // Call the function
    sol.nextPermutation(nums);

    // Print result
    for (int num : nums) {
        cout << num << " ";
    }
    cout << endl;

    return 0;
}

```

Complexity Analysis

Time Complexity: $O(N)$, we find the breaking point and reverse the subarray in linear time.

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



Bonus: STL one-liner

C++ already gives you this algorithm!

```
vector<int> nums = {1,2,3};  
if (!next_permutation(nums.begin(), nums.end())) {  
    // nums has become the smallest permutation  
}
```

It returns `false` if it wrapped around (i.e., you were at the last permutation).

23. Leaders in an Array

Given an array, an element is called a **Leader** if it is strictly greater than all elements to its right.

The **last element** of every array is **always** a leader (since there's nothing to its right).

arr = [10, 22, 12, 3, 0, 6]
Leaders → 22, 12, 6

Approach 1: Brute Force

Idea:

For each element, check **every element to its right** —
if you find any greater element, it's **not a leader**.

```
#include<bits/stdc++.h>  
using namespace std;  
  
vector<int> printLeadersBruteForce(int arr[], int n) {
```

```

vector<int> ans;

for (int i = 0; i < n; i++) {
    bool leader = true;

    //Checking whether arr[i] is greater than all
    //the elements in its right side
    for (int j = i + 1; j < n; j++)
        if (arr[j] > arr[i]) {

            // If any element found is greater than current leader
            // curr element is not the leader.
            leader = false;
            break;
        }

    // Push all the leaders in ans array.
    if (leader)
        ans.push_back(arr[i]);
}

return ans;
}

int main() {

// Array Initialization.
int n = 6;
int arr[n] = {10, 22, 12, 3, 0, 6};

vector<int> ans = printLeadersBruteForce(arr,n);

for(int i = 0;i<ans.size();i++){

    cout<<ans[i]<<" ";
}

cout<<endl;
return 0;
}

```

Output:

22 12 6

Time Complexity: $O(N^2)$ { Since there are nested loops being used, at the worst case n^2 time would be consumed }.

Space Complexity: $O(N)$ { There is no extra space being used in this approach. But, a $O(N)$ of space for ans array will be used in the worst case }.

⚡ Approach 2: Optimal – Traverse from Right

Since we only need elements greater than all to their **right**,
we can **traverse the array from right to left** while keeping track of the **current maximum**.

```
#include<bits/stdc++.h>
using namespace std;

vector<int> printLeaders(int arr[], int n) {

    vector<int> ans;

    // Last element of an array is always a leader,
    // push into ans array.
    int max = arr[n - 1];
    ans.push_back(arr[n-1]);

    // Start checking from the end whether a number is greater
    // than max no. from right, hence leader.
    for (int i = n - 2; i >= 0; i--)
        if (arr[i] > max) {
            ans.push_back(arr[i]);
            max = arr[i];
        }

    return ans;
}
```

```

int main() {
    // Array Initialization.
    int n = 6;
    int arr[n] = {10, 22, 12, 3, 0, 6};

    vector<int> ans = printLeaders(arr,n);

    for(int i = ans.size()-1;i>=0;i--){
        cout<<ans[i]<<" ";
    }

    cout<<endl;
    return 0;
}

```

Output:

22 12 6

Time Complexity: O(N) { Since the array is traversed single time back to front, it will consume O(N) of time where N = size of the array }.

Space Complexity: O(N) { There is no extra space being used in this approach. But, a O(N) of space for ans array will be used in the worst case }.

24. Longest Consecutive Sequence in an Array

Idea: Har element **x** ke liye check karo **x+1, x+2, ...** array me exist karte hain ya nahi (linearSearch).

Cons: Repeated linear search → **slow**.

Approach 1: Brute force (O(N²))

```
#include <bits/stdc++.h>
using namespace std;
```

```

bool linearSearch(vector<int>&a, int num) {
    int n = a.size(); //size of array
    for (int i = 0; i < n; i++) {
        if (a[i] == num)
            return true;
    }
    return false;
}

int longestSuccessiveElements(vector<int>&a) {
    int n = a.size(); //size of array
    int longest = 1;
    //pick a element and search for its
    //consecutive numbers:
    for (int i = 0; i < n; i++) {
        int x = a[i];
        int cnt = 1;
        //search for consecutive numbers
        //using linear search:
        while (linearSearch(a, x + 1) == true) {
            x += 1;
            cnt += 1;
        }

        longest = max(longest, cnt);
    }
    return longest;
}

int main()
{
    vector<int> a = {100, 200, 1, 2, 3, 4};
    int ans = longestSuccessiveElements(a);
    cout << "The longest consecutive sequence is " << ans << "\n";
    return 0;
}

```

Output: The longest consecutive sequence is 4.

Time Complexity: O(N²), N = size of the given array.

Reason: We are using nested loops each running for approximately N times.

Space Complexity: $O(1)$, as we are not using any extra space to solve this problem.

Approach 2: Sort then scan ($O(N \log N)$)

Idea:

1. Array sort karo.
2. Ek pass me consecutive run length count karo.
3. **Duplicate elements** ko skip karna zaroori hai, warna count toot jayega.

```
#include <bits/stdc++.h>
using namespace std;

int longestConsecutive_sort(vector<int>& a) {
    int n = (int)a.size();
    if (n == 0) return 0;

    sort(a.begin(), a.end());
    int longest = 1, curr = 1;

    for (int i = 1; i < n; i++) {
        if (a[i] == a[i-1]) continue;      // duplicate → ignore
        if (a[i] == a[i-1] + 1) curr++;   // consecutive
        else { longest = max(longest, curr); // break
                curr = 1; }
    }
    longest = max(longest, curr);
    return longest;
}
```

Time Complexity: $O(N\log N) + O(N)$, N = size of the given array.

Reason: $O(N\log N)$ for sorting the array. To find the longest sequence, we are using a loop that results in $O(N)$.

Space Complexity: $O(1)$, as we are not using any extra space to solve this problem.

Approach 3: Hash set ($O(N)$ average)

Best for interviews.

Idea:

- `unordered_set` me sab elements daal do.
- Sirf **starting points** se sequences grow karo: koi number x tab start hoga jab $x-1$ set me na ho.
- Fir $x+1, x+2, \dots$ count karte jao.

```
#include <bits/stdc++.h>
using namespace std;

int longestSuccessiveElements(vector<int>&a) {
    int n = a.size();
    if (n == 0) return 0;

    int longest = 1;
    unordered_set<int> st;
    //put all the array elements into set:
    for (int i = 0; i < n; i++) {
        st.insert(a[i]);
    }

    //Find the longest sequence:
    for (auto it : st) {
        //if 'it' is a starting number:
        if (st.find(it - 1) == st.end()) {
            //find consecutive numbers:
            int cnt = 1;
            int x = it;
            while (st.find(x + 1) != st.end()) {
                x = x + 1;
                cnt = cnt + 1;
            }
            longest = max(longest, cnt);
        }
    }
    return longest;
}
```

```

}

int main()
{
    vector<int> a = {100, 200, 1, 2, 3, 4};
    int ans = longestSuccessiveElements(a);
    cout << "The longest consecutive sequence is " << ans << "\n";
    return 0;
}

```

Output: The longest consecutive sequence is 4.

Time Complexity: $O(N) + O(2^N) \sim O(3^N)$, where N = size of the array.

Reason: $O(N)$ for putting all the elements into the set data structure. After that for every starting element, we are finding the consecutive elements. Though we are using nested loops, the set will be traversed at most twice in the worst case. So, the time complexity is $O(2^N)$ instead of $O(N^2)$.

Space Complexity: $O(N)$, as we are using the set data structure to solve this problem.

Note: The time complexity is computed under the assumption that we are using `unordered_set` and it is taking $O(1)$ for the set operations.

If we consider the worst case the set operations will take $O(N)$ in that case and the total time complexity will be approximately $O(N^2)$.

And if we use the set instead of `unordered_set`, the time complexity for the set operations will be $O(\log N)$ and the total time complexity will be $O(N \log N)$.

25. Set Matrix Zero

If any cell (i, j) is 0 , then **entire row i** and **entire column j** must become 0 .

Approach 1: In-place marking with a sentinel (✗ risky)

You mark same row/col by writing -1 and later turn all $-1 \rightarrow 0$.

- **Problem:** If the input can already contain -1 (or any chosen sentinel), you'll corrupt data.

- **Also:** With many zeros, time can degrade to $O(Z \cdot (N+M)) \leq O(NM(N+M))$.
- **Use only** if the matrix domain guarantees the sentinel is safe.

```
#include <bits/stdc++.h>
using namespace std;

void markRow(vector<vector<int>> &matrix, int n, int m, int i) {
    // set all non-zero elements as -1 in the row i:
    for (int j = 0; j < m; j++) {
        if (matrix[i][j] != 0) {
            matrix[i][j] = -1;
        }
    }
}

void markCol(vector<vector<int>> &matrix, int n, int m, int j) {
    // set all non-zero elements as -1 in the col j:
    for (int i = 0; i < n; i++) {
        if (matrix[i][j] != 0) {
            matrix[i][j] = -1;
        }
    }
}

vector<vector<int>> zeroMatrix(vector<vector<int>> &matrix, int n, int m) {
    // Set -1 for rows and cols
    // that contains 0. Don't mark any 0 as -1:

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            if (matrix[i][j] == 0) {
                markRow(matrix, n, m, i);
                markCol(matrix, n, m, j);
            }
        }
    }

    // Finally, mark all -1 as 0:
    for (int i = 0; i < n; i++) {
```

```

        for (int j = 0; j < m; j++) {
            if (matrix[i][j] == -1) {
                matrix[i][j] = 0;
            }
        }
    }

    return matrix;
}

int main()
{
    vector<vector<int>> matrix = {{1, 1, 1}, {1, 0, 1}, {1, 1, 1}};
    int n = matrix.size();
    int m = matrix[0].size();
    vector<vector<int>> ans = zeroMatrix(matrix, n, m);

    cout << "The Final matrix is: n";
    for (auto it : ans) {
        for (auto ele : it) {
            cout << ele << " ";
        }
        cout << "n";
    }
    return 0;
}

```

Output: The Final matrix is: 1 0 1

0 0 0

1 0 1

Time Complexity: $O((N*M)*(N + M)) + O(N*M)$, where N = no. of rows in the matrix and M = no. of columns in the matrix.

Reason: Firstly, we are traversing the matrix to find the cells with the value 0. It takes $O(N*M)$. Now, whenever we find any such cell we mark that row and column with -1. This process takes $O(N+M)$. So, combining this the whole process, finding and marking, takes $O((N*M)*(N + M))$.

Another $O(N*M)$ is taken to mark all the cells with -1 as 0 finally.

Space Complexity: $O(1)$ as we are not using any extra space.

Approach 2: Extra row/col arrays (✓ simple & safe)

Keep two arrays `row[n]`, `col[m]`. First pass marks which rows/cols to zero; second pass zeros them.

- **Time:** $O(NM)$ (two passes)
- **Space:** $O(N + M)$

```
#include <bits/stdc++.h>
using namespace std;

vector<vector<int>> zeroMatrix(vector<vector<int>> &matrix, int n, int m) {

    int row[n] = {0}; // row array
    int col[m] = {0}; // col array

    // Traverse the matrix:
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            if (matrix[i][j] == 0) {
                // mark ith index of row with 1:
                row[i] = 1;

                // mark jth index of col with 1:
                col[j] = 1;
            }
        }
    }

    // Finally, mark all (i, j) as 0
    // if row[i] or col[j] is marked with 1.
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            if (row[i] || col[j]) {
                matrix[i][j] = 0;
            }
        }
    }

    return matrix;
}
```

```

int main()
{
    vector<vector<int>> matrix = {{1, 1, 1}, {1, 0, 1}, {1, 1, 1}};
    int n = matrix.size();
    int m = matrix[0].size();
    vector<vector<int>> ans = zeroMatrix(matrix, n, m);

    cout << "The Final matrix is: n";
    for (auto it : ans) {
        for (auto ele : it) {
            cout << ele << " ";
        }
        cout << "n";
    }
    return 0;
}

```

Output: The Final matrix is: 1 0 1

```

0 0 0
1 0 1

```

Time Complexity: $O(2*(N*M))$, where N = no. of rows in the matrix and M = no. of columns in the matrix.

Reason: We are traversing the entire matrix 2 times and each traversal is taking $O(N*M)$ time complexity.

Space Complexity: $O(N) + O(M)$, where N = no. of rows in the matrix and M = no. of columns in the matrix.

Reason: $O(N)$ is for using the row array and $O(M)$ is for using the col array.

Approach 3: O(1) extra space using 1st row & 1st column as markers (✓ optimal)

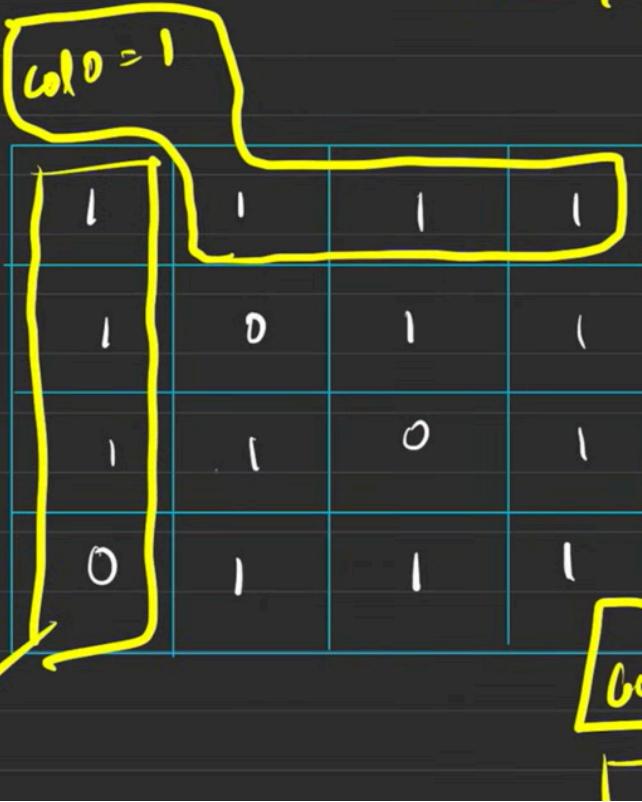
Use `matrix[i][0]` to mark row `i` and `matrix[0][j]` to mark column `j`.

`matrix[0][0]` is ambiguous (marks both), so track first column separately with `col0`.

- **Time:** $O(NM)$ (two passes)
- **Space:** $O(1)$

Set Minimum Zeros

$\text{col}[0] \rightarrow \text{row}[n]$
 $\text{row}[0] \rightarrow \text{col}[m]$



```
#include <bits/stdc++.h>
using namespace std;

vector<vector<int>> zeroMatrix(vector<vector<int>> &matrix, int n, int m) {

    // int row[n] = {0}; --> matrix[..][0]
    // int col[m] = {0}; --> matrix[0][..]

    int col0 = 1;
    // step 1: Traverse the matrix and
    // mark 1st row & col accordingly:
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            if (matrix[i][j] == 0) {
                // mark i-th row:
                matrix[i][0] = 0;
                // mark j-th column:
                matrix[0][j] = 0;
            }
        }
    }
}
```

```

        if (j != 0)
            matrix[0][j] = 0;
        else
            col0 = 0;
    }
}

// Step 2: Mark with 0 from (1,1) to (n-1, m-1):
for (int i = 1; i < n; i++) {
    for (int j = 1; j < m; j++) {
        if (matrix[i][j] != 0) {
            // check for col & row:
            if (matrix[i][0] == 0 || matrix[0][j] == 0) {
                matrix[i][j] = 0;
            }
        }
    }
}

//step 3: Finally mark the 1st col & then 1st row:
if (matrix[0][0] == 0) {
    for (int j = 0; j < m; j++) {
        matrix[0][j] = 0;
    }
}
if (col0 == 0) {
    for (int i = 0; i < n; i++) {
        matrix[i][0] = 0;
    }
}

return matrix;
}

int main()
{
    vector<vector<int>> matrix = {{1, 1, 1}, {1, 0, 1}, {1, 1, 1}};
    int n = matrix.size();
    int m = matrix[0].size();
    vector<vector<int>> ans = zeroMatrix(matrix, n, m);

    cout << "The Final matrix is: n";
    for (auto it : ans) {

```

```

        for (auto ele : it) {
            cout << ele << " ";
        }
        cout << "n";
    }
    return 0;
}

```

Output: The Final matrix is:

1	0	1
0	0	0
1	0	1

Time Complexity: $O(2*(N*M))$, where N = no. of rows in the matrix and M = no. of columns in the matrix.

Reason: In this approach, we are also traversing the entire matrix 2 times and each traversal is taking $O(N*M)$ time complexity.

Space Complexity: $O(1)$ as we are not using any extra space.

26. Rotate Image by 90 degree

Input:

1	2	3
4	5	6
7	8	9

Output (rotated 90° clockwise):

7	4	1
8	5	2
9	6	3

APPROACH 1 — Using Extra Matrix (Brute Force)

Intuition:

Every element (i, j) in the original matrix moves to position $(j, n - 1 - i)$ in the rotated matrix.

So:

```
new[j][n - i - 1] = old[i][j];
```

We simply copy into a new matrix based on that relationship.

```
#include<bits/stdc++.h>

using namespace std;
vector < vector < int >> rotate(vector < vector < int >> & matrix) {
    int n = matrix.size();
    vector < vector < int >> rotated(n, vector < int > (n, 0));
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            rotated[j][n - i - 1] = matrix[i][j];
        }
    }
    return rotated;
}

int main() {
    vector < vector < int >> arr;
    arr = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
    vector < vector < int >> rotated = rotate(arr);
    cout << "Rotated Image" << endl;
    for (int i = 0; i < rotated.size(); i++) {
        for (int j = 0; j < rotated[0].size(); j++) {
            cout << rotated[i][j] << " ";
        }
        cout << "n";
    }
}
```

Output:

Rotated Image

7 4 1
8 5 2
9 6 3

Time Complexity: O(N*N) to linearly iterate and put it into some other matrix.

Space Complexity: O(N*N) to copy it into some other matrix.

⚡ APPROACH 2 — In-place (Optimal)

💡 Intuition:

We can do it **without using extra space**.

We know:

90° rotation = **Transpose + Reverse every row**

Why it works:

Transpose: converts rows → columns
(matrix[i][j] ↔ matrix[j][i])

1 2 3	1 4 7
4 5 6	→ 2 5 8
7 8 9	3 6 9

Reverse each row: rotates the matrix

1 4 7	7 4 1
2 5 8	→ 8 5 2
3 6 9	9 6 3

```
#include<bits/stdc++.h>
```

```
using namespace std;
void rotate(vector < vector < int >> & matrix) {
    int n = matrix.size();
```

```

//transposing the matrix
for (int i = 0; i < n; i++) {
    for (int j = 0; j < i; j++) {
        swap(matrix[i][j], matrix[j][i]);
    }
}
//reversing each row of the matrix
for (int i = 0; i < n; i++) {
    reverse(matrix[i].begin(), matrix[i].end());
}
}

int main() {
    vector<vector<int>> arr;
    arr = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
    rotate(arr);
    cout << "Rotated Image" << endl;
    for (int i = 0; i < arr.size(); i++) {
        for (int j = 0; j < arr[0].size(); j++) {
            cout << arr[i][j] << " ";
        }
        cout << "n";
    }
}

```

Output:

Rotated Image
 7 4 1
 8 5 2
 9 6 3

Time Complexity: $O(N^2)$. One $O(N^2)$ is for transposing the matrix and the other is for reversing the matrix.

Space Complexity: $O(1)$.

27. Spiral Traversal of Matrix

🎯 Problem Statement

Given an $n \times m$ matrix, print all its elements in **spiral order traversal** (clockwise).

1 2 3 4

5 6 7 8

9 10 11 12

13 14 15 16

Output:

1 2 3 4 8 12 16 15 14 13 9 5 6 7 11 10

🧠 Intuition

Imagine the traversal in **layers (rings)**:

1. Traverse the **top row** (\rightarrow)
2. Traverse the **right column** (\downarrow)
3. Traverse the **bottom row** (\leftarrow)
4. Traverse the **left column** (\uparrow)

After each full cycle, we move **one layer inward** by updating:

- `top++`
- `right--`
- `bottom--`
- `left++`

We continue until all layers are covered (`top <= bottom && left <= right`).

```

#include <bits/stdc++.h>

using namespace std;

vector<int> printSpiral(vector<vector<int>> mat) {

    // Define ans array to store the result.
    vector<int> ans;

    int n = mat.size(); // no. of rows
    int m = mat[0].size(); // no. of columns

    // Initialize the pointers reqd for traversal.
    int top = 0, left = 0, bottom = n - 1, right = m - 1;

    // Loop until all elements are not traversed.
    while (top <= bottom && left <= right) {

        // For moving left to right
        for (int i = left; i <= right; i++)
            ans.push_back(mat[top][i]);

        top++;

        // For moving top to bottom.
        for (int i = top; i <= bottom; i++)
            ans.push_back(mat[i][right]);

        right--;

        // For moving right to left.
        if (top <= bottom) {
            for (int i = right; i >= left; i--)
                ans.push_back(mat[bottom][i]);

            bottom--;
        }

        // For moving bottom to top.
        if (left <= right) {
            for (int i = bottom; i >= top; i--)
                ans.push_back(mat[i][left]);

            left++;
        }
    }
}

```

```

    }
}
return ans;
}

int main() {

//Matrix initialization.
vector<vector<int>> mat {{1, 2, 3, 4},
{5, 6, 7, 8},
{9, 10, 11, 12},
{13, 14, 15, 16}};

vector<int> ans = printSpiral(mat);

for(int i = 0;i<ans.size();i++){
    cout<<ans[i]<<" ";
}

cout<<endl;

return 0;
}
Output:

```

1 2 3 4 8 12 16 15 14 13 9 5 6 7 11 10

Time Complexity: $O(m \times n)$ { Since all the elements are being traversed once and there are total $n \times m$ elements (m elements in each row and total n rows) so the time complexity will be $O(n \times m)$ }.

Space Complexity: $O(n)$ { Extra Space used for storing traversal in the ans array }.

28. Count Subarray sum Equals K

1. Brute force $O(N^3)$

Har subarray $[i \dots j]$ ka sum inner loop se nikaalo. (Slow)

```
#include <bits/stdc++.h>
using namespace std;

int findAllSubarraysWithGivenSum(vector < int > & arr, int k) {
    int n = arr.size(); // size of the given array.
    int cnt = 0; // Number of subarrays:

    for (int i = 0 ; i < n; i++) { // starting index i
        for (int j = i; j < n; j++) { // ending index j

            // calculate the sum of subarray [i...j]
            int sum = 0;
            for (int K = i; K <= j; K++)
                sum += arr[K];

            // Increase the count if sum == k:
            if (sum == k)
                cnt++;
        }
    }
    return cnt;
}

int main()
{
    vector arr = {3, 1, 2, 4};
    int k = 6;
    int cnt = findAllSubarraysWithGivenSum(arr, k);
    cout << "The number of subarrays is: " << cnt << "\n";
    return 0;
}
```

Output: The number of subarrays is: 2

Time Complexity: $O(N^3)$, where N = size of the array.

Reason: We are using three nested loops here. Though all are not running for exactly N times, the time complexity will be approximately $O(N^3)$.

Space Complexity: $O(1)$ as we are not using any extra space.

2. Better $O(N^2)$

Har is se start karke running sum rakhte jao: `sum += arr[j]`. (Still quadratic)

```
#include <bits/stdc++.h>
using namespace std;

int findAllSubarraysWithGivenSum(vector < int > & arr, int k) {
    int n = arr.size(); // size of the given array.
    int cnt = 0; // Number of subarrays:

    for (int i = 0 ; i < n; i++) { // starting index i
        int sum = 0;
        for (int j = i; j < n; j++) { // ending index j
            // calculate the sum of subarray [i..j]
            // sum of [i..j-1] + arr[j]
            sum += arr[j];

            // Increase the count if sum == k:
            if (sum == k)
                cnt++;
        }
    }
    return cnt;
}

int main()
{
    vector arr = {3, 1, 2, 4};
    int k = 6;
    int cnt = findAllSubarraysWithGivenSum(arr, k);
    cout << "The number of subarrays is: " << cnt << "\n";
    return 0;
}
```

Output: The number of subarrays is: 2

Time Complexity: O(N²), where N = size of the array.

Reason: We are using two nested loops here. As each of them is running for exactly N times, the time complexity will be approximately O(N²).

Space Complexity: O(1) as we are not using any extra space.

3. Optimal (Prefix-sum + Hash) O(N) avg

Idea: agar `preSum` current tak ka sum hai, to jitni baar `preSum - K` pehle aa chuka hai, utni subarrays end-at-i ka sum K banate hain.

Isliye ek map me **prefix-sum frequency** rakhte hain. Start me `mpp[0] = 1` (empty prefix) zaroori hai.

Note: Two-pointer sliding window **sirf positive numbers** pe kaam karta; is problem me negatives ho sakte hain, to prefix-sum + hash hi best.

```
#include <bits/stdc++.h>
using namespace std;

long long countSubarraysWithSumK(const vector<int>& arr, long long k) {
    unordered_map<long long, long long> freq; // prefix-sum -> count
    long long preSum = 0, cnt = 0;

    freq[0] = 1; // empty prefix

    for (int x : arr) {
        preSum += x;
        long long need = preSum - k; // we want previous prefix = preSum - k
        if (freq.find(need) != freq.end())
            cnt += freq[need];
        freq[preSum]++;
    }
    return cnt;
}

int main() {
    vector<int> arr = {3, 1, 2, 4};
    long long k = 6;
    cout << "The number of subarrays is: " << countSubarraysWithSumK(arr, k) << "\n";
    return 0;
}
```

Output: The number of subarrays is: 2

Time Complexity: O(N) or O(N*logN) depending on which map data structure we are using, where N = size of the array.

Reason: For example, if we are using an unordered_map data structure in C++ the time complexity will be O(N) but if we are using a map data structure, the time complexity will be O(N*logN). The least complexity will be O(N) as we are using a loop to traverse the array.

Space Complexity: O(N) as we are using a map data structure.

Kab long long?

Prefix sums overflow avoid karne ke liye (bade values), `preSum` aur `k` ko `long long` rakhna safe rehta hai.

29. Program to generate Pascal's Triangle

Pascal's Triangle is a triangular arrangement of binomial coefficients.

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
```

Formally:

```
arr[i][j] = arr[i-1][j-1] + arr[i-1][j]
with arr[i][0] = arr[i][i] = 1
```

Approach 1 — Generate Full Pascal's Triangle (Brute Force)

We generate all rows one by one using the rule:

```
row[j] = prevRow[j-1] + prevRow[j]
```

```
#include <bits/stdc++.h>
using namespace std;
```

```

// Class containing Pascal's Triangle generation logic
class Solution {
public:
    // Function to generate Pascal's Triangle up to numRows
    vector<vector<int>> generate(int numRows) {
        // Result vector to hold all rows
        vector<vector<int>> triangle;

        // Loop for each row
        for (int i = 0; i < numRows; i++) {
            // Create a row with size (i+1) and initialize all elements to 1
            vector<int> row(i + 1, 1);

            // Fill elements from index 1 to i-1 (middle values)
            for (int j = 1; j < i; j++) {
                // Each element = sum of two elements above it
                row[j] = triangle[i - 1][j - 1] + triangle[i - 1][j];
            }

            // Add current row to the triangle
            triangle.push_back(row);
        }
        return triangle;
    }
};

int main() {
    Solution obj;
    int n = 5;

    // Generate and print Pascal's Triangle
    vector<vector<int>> result = obj.generate(n);
    for (auto &row : result) {
        for (auto &val : row) cout << val << " ";
        cout << endl;
    }
}

```

Time Complexity: $O(N^2)$, we generate all the elements in first N rows sequentially one by one.
Space Complexity: $O(N^2)$, additional space used for storing the entire pascal triangle.

Approach 2 — Generate Only Nth Row (Efficient)

We can compute the **Nth row directly** using the **binomial formula**:

$$C(n,k) = C(n,k-1) * (n-k)/k$$

This lets us compute the entire row in $O(N)$ time without using the triangle.

```
#include <bits/stdc++.h>
using namespace std;

// Class containing Pascal's Triangle row generation logic
class Solution {
public:
    // Function to generate the Nth row of Pascal's Triangle
    vector<long long> getNthRow(int N) {
        // Result vector to store the row
        vector<long long> row;

        // First value of the row is always 1
        long long val = 1;
        row.push_back(val);

        // Compute remaining values using the relation:
        //  $C(n, k) = C(n, k-1) * (n-k) / k$ 
        for (int k = 1; k < N; k++) {
            val = val * (N - k) / k;
            row.push_back(val);
        }

        return row;
    }
};

int main() {
    int N = 5; // Example: 5th row
    Solution sol;
    vector<long long> result = sol.getNthRow(N);

    // Print the row
    for (auto num : result) {
        cout << num << " ";
    }
    return 0;
}
```

}

Time Complexity: $O(N)$, we iterate N times to compute each element of the row in $O(1)$ time using the direct relation.

Space Complexity: $O(N)$, additional space used for storing the N th row.

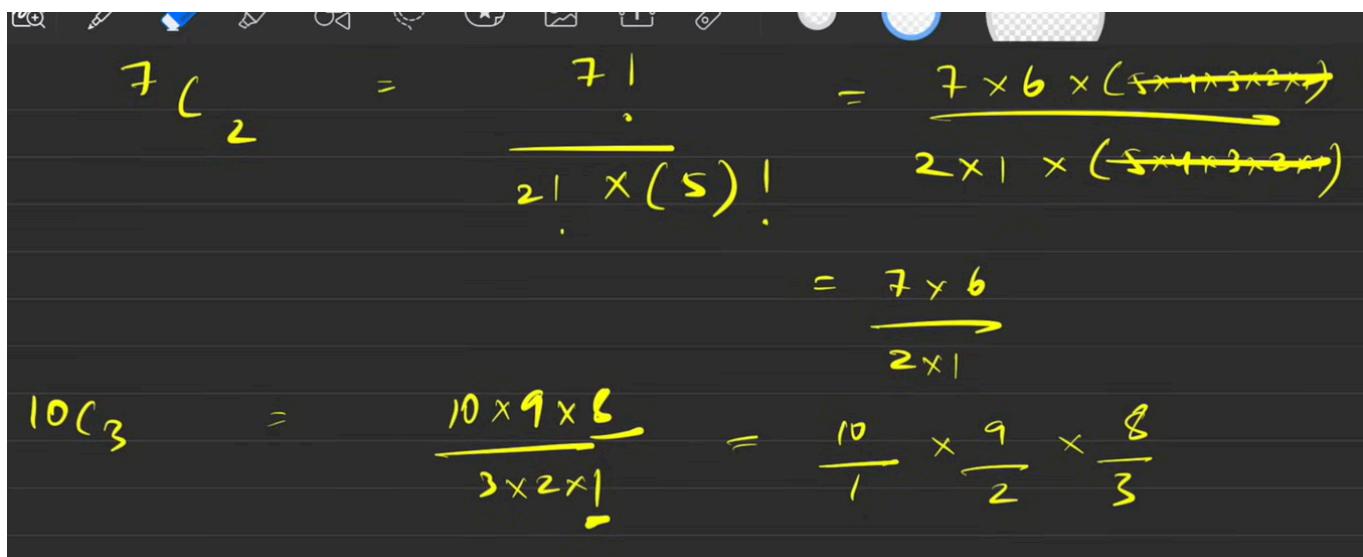
Approach 3 — Find the Element at (r, c)

The element in the r -th row and c -th column of Pascal's triangle is:

$$\text{Element}(r,c) = C(r-1,c-1)$$

We can calculate this using the iterative formula for binomial coefficients:

$$C(n,k) = n! / ((n-k)! * k!)$$


$$\begin{aligned} 7C_2 &= \frac{7!}{2! \times (5)!} = \frac{7 \times 6 \times (5 \times 4 \times 3 \times 2 \times 1)}{2 \times 1 \times (5 \times 4 \times 3 \times 2 \times 1)} \\ &= \frac{7 \times 6}{2 \times 1} \\ 10C_3 &= \frac{10 \times 9 \times 8}{3 \times 2 \times 1} = \frac{10}{1} \times \frac{9}{2} \times \frac{8}{3} \end{aligned}$$

```
#include <bits/stdc++.h>
using namespace std;

// Solution class to find the (r, c) element of Pascal's Triangle
class Solution {
public:
    // Function to compute binomial coefficient (nCr)
```

```

long long findPascalElement(int r, int c) {
    // Element is C(r-1, c-1)
    int n = r - 1;
    int k = c - 1;

    long long result = 1;

    // Compute C(n, k) using iterative formula
    for (int i = 0; i < k; i++) {
        result *= (n - i);
        result /= (i + 1);
    }

    return result;
}

int main() {
    Solution sol;
    int r = 5, c = 3;
    cout << sol.findPascalElement(r, c);
    return 0;
}

```

Time Complexity: $O(\min(c, r-c))$, The loop runs for $\min(c-1, r-c)$ iterations because binomial coefficients are symmetric.

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

30. Majority Elements(>N/3 times) | Find the elements that appears more than N/3 times in the array

1) Brute force (O(N²), O(1))

```
#include <bits/stdc++.h>
using namespace std;

vector<int> majorityElement(vector<int> v) {
    int n = v.size(); //size of the array
    vector<int> ls; // list of answers

    for (int i = 0; i < n; i++) {
        //selected element is v[i]:
        // Checking if v[i] is not already
        // a part of the answer:
        if (ls.size() == 0 || ls[0] != v[i]) {
            int cnt = 0;
            for (int j = 0; j < n; j++) {
                // counting the frequency of v[i]
                if (v[j] == v[i]) {
                    cnt++;
                }
            }

            // check if frquency is greater than n/3:
            if (cnt > (n / 3))
                ls.push_back(v[i]);
        }

        if (ls.size() == 2) break;
    }

    return ls;
}

int main()
{
    vector<int> arr = {11, 33, 33, 11, 33, 11};
    vector<int> ans = majorityElement(arr);
    cout << "The majority elements are: ";
    for (auto it : ans)
```

```

    cout << it << " ";
    cout << "\n";
    return 0;
}

```

Output: The majority elements are: 11 33

Time Complexity: $O(N^2)$, where N = size of the given array.

Reason: For every element of the array the inner loop runs for N times. And there are N elements in the array. So, the total time complexity is $O(N^2)$.

Space Complexity: $O(1)$ as we are using a list that stores a maximum of 2 elements. The space used is so small that it can be considered constant.

2) Hash map / unordered_map ($O(N \log N)$ or $O(N)$, $O(N)$)

Your map version is correct; note that we can push when count becomes $\geq \text{floor}(N/3)+1$. Using `unordered_map` is typically $O(N)$:

```

#include <bits/stdc++.h>
using namespace std;

vector<int> majorityElement(vector<int> v) {
    int n = v.size(); //size of the array
    vector<int> ls; // list of answers

    //declaring a map:
    map<int, int> mpp;

    // least occurrence of the majority element:
    int mini = int(n / 3) + 1;

    //storing the elements with its occurnce:
    for (int i = 0; i < n; i++) {
        mpp[v[i]]++;
    }

    //checking if v[i] is

```

```

// the majority element:
if (mpp[v[i]] == mini) {
    ls.push_back(v[i]);
}
if (ls.size() == 2) break;
}

return ls;
}

int main()
{
    vector<int> arr = {11, 33, 33, 11, 33, 11};
    vector<int> ans = majorityElement(arr);
    cout << "The majority elements are: ";
    for (auto it : ans)
        cout << it << " ";
    cout << "\n";
    return 0;
}
Output: The majority elements are: 33 11

```

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

Reason: We are using a map data structure. Insertion in the map takes $\log N$ time. And we are doing it for N elements. So, it results in the first term $O(N \log N)$.

If we use `unordered_map` instead, the first term will be $O(N)$ for the best and average case and for the worst case, it will be $O(N^2)$.

Space Complexity: $O(N)$ as we are using a map data structure. We are also using a list that stores a maximum of 2 elements. That space used is so small that it can be considered constant.

3) Extended Boyer–Moore (Best: $O(N)$, $O(1)$)

This tracks up to **two candidates** and their counts, then **verifies** them.;

```
#include <bits/stdc++.h>
using namespace std;

vector<int> majorityElement_boyerMoore(const vector<int>& v) {
    int n = v.size();
    int cnt1 = 0, cnt2 = 0;
```

```

int el1 = 0, el2 = 0;

// 1) Find potential candidates
for (int x : v) {
    if (cnt1 > 0 && x == el1) cnt1++;
    else if (cnt2 > 0 && x == el2) cnt2++;
    else if (cnt1 == 0) { el1 = x; cnt1 = 1; }
    else if (cnt2 == 0) { el2 = x; cnt2 = 1; }
    else { cnt1--; cnt2--; }
}

// 2) Verify counts
cnt1 = cnt2 = 0;
for (int x : v) {
    if (x == el1) cnt1++;
    else if (x == el2) cnt2++;
}

vector<int> ans;
if (cnt1 > n/3) ans.push_back(el1);
if (cnt2 > n/3) ans.push_back(el2);
return ans;
}

```

Output: The majority elements are: 11 33

Time Complexity: O(N) + O(N), where N = size of the given array.

Reason: The first O(N) is to calculate the counts and find the expected majority elements. The second one is to check if the calculated elements are the majority ones or not.

Space Complexity: O(1) as we are only using a list that stores a maximum of 2 elements. The space used is so small that it can be considered constant.

31. 3 Sum : Find triplets that add up to a zero

1) Brute force ($O(N^3)$)

Try every (i, j, k) with $i < j < k$.

```
#include <bits/stdc++.h>
using namespace std;

vector<vector<int>> triplet(int n, vector<int> &arr) {
    set<vector<int>> st;

    // check all possible triplets:
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            for (int k = j + 1; k < n; k++) {
                if (arr[i] + arr[j] + arr[k] == 0) {
                    vector<int> temp = {arr[i], arr[j], arr[k]};
                    sort(temp.begin(), temp.end());
                    st.insert(temp);
                }
            }
        }
    }

    //store the set elements in the answer:
    vector<vector<int>> ans(st.begin(), st.end());
    return ans;
}

int main()
{
    vector<int> arr = { -1, 0, 1, 2, -1, -4};
    int n = arr.size();
    vector<vector<int>> ans = triplet(n, arr);
    for (auto it : ans) {
        cout << "[";
        for (auto i : it) {
            cout << i << " ";
        }
        cout << "] ";
    }
}
```

```

cout << "\n";
return 0;
}

```

Output: [-1 -1 2] [-1 0 1]

Time Complexity: $O(N^3 * \log(\text{no. of unique triplets}))$, where $N = \text{size of the array}$.

Reason: Here, we are mainly using 3 nested loops. And inserting triplets into the set takes $O(\log(\text{no. of unique triplets}))$ time complexity. But we are not considering the time complexity of sorting as we are just sorting 3 elements every time.

Space Complexity: $O(2 * \text{no. of the unique triplets})$ as we are using a set data structure and a list to store the triplets.

2) “Two loops + hash set” ($O(N^2)$)

Fix i , then for the subarray $(i+1 \dots n-1)$ you’re basically doing **2-sum** to target $-\text{arr}[i]$.

Keep a `set<int>` (or `unordered_set<int>`) while moving j .

If $\text{third} = -(arr[i] + arr[j])$ exists in the set, you found a triplet `{arr[i], arr[j], third}`.

Still use a `set<vector<int>>` to dedupe final triplets (because order/duplicates can repeat).

```

#include <bits/stdc++.h>
using namespace std;

vector<vector<int>> triplet(int n, vector<int> &arr) {
    set<vector<int>> st;

    for (int i = 0; i < n; i++) {
        set<int> hashset;
        for (int j = i + 1; j < n; j++) {
            //Calculate the 3rd element:
            int third = -(arr[i] + arr[j]);

            //Find the element in the set:
            if (hashset.find(third) != hashset.end()) {
                vector<int> temp = {arr[i], arr[j], third};
                sort(temp.begin(), temp.end());
                st.insert(temp);
            }
        }
    }
}

```

```

        hashset.insert(arr[j]);
    }
}

//store the set in the answer:
vector<vector<int>> ans(st.begin(), st.end());
return ans;
}

```

```

int main()
{
    vector<int> arr = { -1, 0, 1, 2, -1, -4};
    int n = arr.size();
    vector<vector<int>> ans = triplet(n, arr);
    for (auto it : ans) {
        cout << "[";
        for (auto i : it) {
            cout << i << " ";
        }
        cout << "] ";
    }
    cout << "\n";
    return 0;
}

```

Output: [-1 -1 2] [-1 0 1]

Time Complexity: $O(N^2 * \log(\text{no. of unique triplets}))$, where $N = \text{size of the array}$.

Reason: Here, we are mainly using 3 nested loops. And inserting triplets into the set takes $O(\log(\text{no. of unique triplets}))$ time complexity. But we are not considering the time complexity of sorting as we are just sorting 3 elements every time.

Space Complexity: $O(2 * \text{no. of the unique triplets}) + O(N)$ as we are using a set data structure and a list to store the triplets and extra $O(N)$ for storing the array elements in another set.

3) Sort + two pointers (optimal $O(N^2)$, in-place dedupe)

Sort the array once ($O(N \log N)$).

Loop i from $0..n-1$ (skip duplicates for i).

For each i , run a classic two-pointer 2-sum on the subarray ($i+1 \dots n-1$):

- $j = i+1, k = n-1$
- If $\text{arr}[i] + \text{arr}[j] + \text{arr}[k] < 0 \rightarrow j++$

- If $> 0 \rightarrow k--$
- If $= 0 \rightarrow$ record triplet, then move $j++$ and $k--$ and skip duplicates for both sides.

Why it's best: $O(N^2)$ time, $O(1)$ extra space, and duplicate-skipping happens on the fly, so no `set<vector<int>>` needed.

```
#include <bits/stdc++.h>
using namespace std;

vector<vector<int>> triplet(int n, vector<int> &arr) {
    vector<vector<int>> ans;
    sort(arr.begin(), arr.end());
    for (int i = 0; i < n; i++) {
        //remove duplicates:
        if (i != 0 && arr[i] == arr[i - 1]) continue;

        //moving 2 pointers:
        int j = i + 1;
        int k = n - 1;
        while (j < k) {
            int sum = arr[i] + arr[j] + arr[k];
            if (sum < 0) {
                j++;
            }
            else if (sum > 0) {
                k--;
            }
            else {
                vector<int> temp = {arr[i], arr[j], arr[k]};
                ans.push_back(temp);
                j++;
                k--;
                //skip the duplicates:
                while (j < k && arr[j] == arr[j - 1]) j++;
                while (j < k && arr[k] == arr[k + 1]) k--;
            }
        }
    }
    return ans;
}

int main()
```

```

{
vector<int> arr = { -1, 0, 1, 2, -1, -4};
int n = arr.size();
vector<vector<int>> ans = triplet(n, arr);
for (auto it : ans) {
    cout << "[";
    for (auto i : it) {
        cout << i << " ";
    }
    cout << "] ";
}
cout << "\n";
return 0;
}

```

Output: [-1 -1 2] [-1 0 1]

Time Complexity: $O(N \log N) + O(N^2)$, where N = size of the array.

Reason: The pointer i is running for approximately N times. And both the pointers j and k combined can run for approximately N times including the operation of skipping duplicates. So the total time complexity will be $O(N^2)$.

Space Complexity: $O(\text{no. of quadruplets})$, This space is only used to store the answer. We are not using any extra space to solve this problem. So, from that perspective, space complexity can be written as $O(1)$.

32. 4 Sum | Find Quads that add up to a target value

1) Pure brute force (4 nested loops + set) — $O(N^4)$

```

#include <bits/stdc++.h>
using namespace std;

vector<vector<int>> fourSum(vector<int>& nums, int target) {
    int n = nums.size(); //size of the array
    set<vector<int>> st;

    //checking all possible quadruplets:
    for (int i = 0; i < n; i++) {

```

```

        for (int j = i + 1; j < n; j++) {
            for (int k = j + 1; k < n; k++) {
                for (int l = k + 1; l < n; l++) {
                    // taking bigger data type
                    // to avoid integer overflow:
                    long long sum = nums[i] + nums[j];
                    sum += nums[k];
                    sum += nums[l];

                    if (sum == target) {
                        vector<int> temp = {nums[i], nums[j], nums[k], nums[l]};
                        sort(temp.begin(), temp.end());
                        st.insert(temp);
                    }
                }
            }
        }
    }

    vector<vector<int>> ans(st.begin(), st.end());
    return ans;
}

int main()
{
    vector<int> nums = {4, 3, 3, 4, 4, 2, 1, 2, 1, 1};
    int target = 9;
    vector<vector<int>> ans = fourSum(nums, target);
    cout << "The quadruplets are: \n";
    for (auto it : ans) {
        cout << "[";
        for (auto ele : it) {
            cout << ele << " ";
        }
        cout << "] ";
    }
    cout << "\n";
    return 0;
}

```

Output: The quadruplets are: [1 1 3 4] [1 2 2 4] [1 2 3 3]

Time Complexity: O(N4), where N = size of the array.

Reason: Here, we are mainly using 4 nested loops. But we are not considering the time complexity of sorting as we are just sorting 4 elements every time.

Space Complexity: $O(2 * \text{no. of the quadruplets})$ as we are using a set data structure and a list to store the quads.

2) 3 loops + hash set — $O(N^3)$ avg

```
#include <bits/stdc++.h>
using namespace std;

vector<vector<int>> fourSum(vector<int>& nums, int target) {
    int n = nums.size(); //size of the array
    set<vector<int>> st;

    //checking all possible quadruplets:
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            set<long long> hashset;
            for (int k = j + 1; k < n; k++) {
                // taking bigger data type
                // to avoid integer overflow:
                long long sum = nums[i] + nums[j];
                sum += nums[k];
                long long fourth = target - sum;
                if (hashset.find(fourth) != hashset.end()) {
                    vector<int> temp = {nums[i], nums[j], nums[k], (int)(fourth)};
                    sort(temp.begin(), temp.end());
                    st.insert(temp);
                }
                // put the kth element into the hashset:
                hashset.insert(nums[k]);
            }
        }
    }
    vector<vector<int>> ans(st.begin(), st.end());
    return ans;
}

int main()
{
```

```

vector<int> nums = {4, 3, 3, 4, 4, 2, 1, 2, 1, 1};
int target = 9;
vector<vector<int>> ans = fourSum(nums, target);
cout << "The quadruplets are: \n";
for (auto it : ans) {
    cout << "[";
    for (auto ele : it) {
        cout << ele << " ";
    }
    cout << "] ";
}
cout << "\n";
return 0;
}

```

Output: The quadruplets are: [1 1 3 4] [1 2 2 4] [1 2 3 3]

Time Complexity: $O(N^3 * \log(M))$, where N = size of the array, M = no. of elements in the set.

Reason: Here, we are mainly using 3 nested loops, and inside the loops there are some operations on the set data structure which take $\log(M)$ time complexity.

Space Complexity: $O(2 * \text{no. of the quadruplets}) + O(N)$

Reason: we are using a set data structure and a list to store the quads. This results in the first term. And the second space is taken by the set data structure we are using to store the array elements. At most, the set can contain approximately all the array elements and so the space complexity is $O(N)$.

3) Sort + two pointers (optimal) — $O(N^3)$, $O(1)$ extra

- Sort the array.
- Fix i (skip duplicates), fix j (skip duplicates).
- Two pointers $k=j+1$, $l=n-1$:
 - If sum < target $\rightarrow k++$
 - If sum > target $\rightarrow l--$
 - If sum == target \rightarrow push quad, then $k++$, $l--$ and skip duplicates on both sides.
- **Why it's best:** tight $O(N^3)$, no sets, duplicate handling is clean and deterministic.

```
#include <bits/stdc++.h>
using namespace std;
```

```

vector<vector<int>> fourSum(vector<int>& nums, int target) {
    int n = nums.size(); //size of the array
    vector<vector<int>> ans;

    // sort the given array:
    sort(nums.begin(), nums.end());

    //calculating the quadruplets:
    for (int i = 0; i < n; i++) {
        // avoid the duplicates while moving i:
        if (i > 0 && nums[i] == nums[i - 1]) continue;
        for (int j = i + 1; j < n; j++) {
            // avoid the duplicates while moving j:
            if (j > i + 1 && nums[j] == nums[j - 1]) continue;

            // 2 pointers:
            int k = j + 1;
            int l = n - 1;
            while (k < l) {
                long long sum = nums[i];
                sum += nums[j];
                sum += nums[k];
                sum += nums[l];
                if (sum == target) {
                    vector<int> temp = {nums[i], nums[j], nums[k], nums[l]};
                    ans.push_back(temp);
                    k++; l--;
                }
                //skip the duplicates:
                while (k < l && nums[k] == nums[k - 1]) k++;
                while (k < l && nums[l] == nums[l + 1]) l--;
            }
            else if (sum < target) k++;
            else l--;
        }
    }

    return ans;
}

int main()
{
    vector<int> nums = {4, 3, 3, 4, 4, 2, 1, 2, 1, 1};

```

```

int target = 9;
vector<vector<int>> ans = fourSum(nums, target);
cout << "The quadruplets are: \n";
for (auto it : ans) {
    cout << "[";
    for (auto ele : it) {
        cout << ele << " ";
    }
    cout << "] ";
}
cout << "\n";
return 0;
}

```

Output: The quadruplets are: [1 1 3 4] [1 2 2 4] [1 2 3 3]

Time Complexity: O(N³), where N = size of the array.

Reason: Each of the pointers i and j, is running for approximately N times. And both the pointers k and l combined can run for approximately N times including the operation of skipping duplicates. So the total time complexity will be O(N³).

Space Complexity: O(no. of quadruplets), This space is only used to store the answer. We are not using any extra space to solve this problem. So, from that perspective, space complexity can be written as O(1).

33. Length of the longest subarray with zero Sum

Input: [9, -3, 3, -1, 6, -5]

Output: 5

Explanation:

Subarray [-3, 3, -1, 6, -5] ka sum = 0

Brute Force Approach

- ◆ **Logic:**

1. Har index **i** se start karte hue har possible **j** tak sum nikalo.
2. Agar kabhi **sum == 0** ho jaye, to **length = j - i + 1** calculate karo.
3. **maxLen** me maximum length store kar lo.

```

#include <bits/stdc++.h>
using namespace std;

int longestSubarrayBrute(vector<int> &a) {
    int n = a.size();
    int maxLen = 0;

    for (int i = 0; i < n; i++) {
        int sum = 0;
        for (int j = i; j < n; j++) {
            sum += a[j];
            if (sum == 0) {
                maxLen = max(maxLen, j - i + 1);
            }
        }
    }
    return maxLen;
}

int main() {
    vector<int> a = {9, -3, 3, -1, 6, -5};
    cout << longestSubarrayBrute(a);
}

```

Time Complexity: $O(N^2)$ as we have two loops for traversal

Space Complexity: $O(1)$ as we aren't using any extra space.

Can this be done in a single traversal? Let's check :)



Optimized Approach (Using HashMap)

- ◆ **Idea:**

Yahan ek smart trick hai:

Agar kisi prefix sum (sum from 0 to i) ko humne pehle bhi dekha hai,
to iska matlab hai ki dono occurrences ke beech ka subarray ka sum = 0 hai!

◆ Step by Step:

1. Maintain a variable `sum = 0` (prefix sum).
2. Traverse array ke har element par:
 - o `sum += A[i]`
 - o Agar `sum == 0`, matlab start se i tak sum 0 hua $\Rightarrow \text{maxi} = i + 1$
 - o Agar `sum` pehle bhi map me mila, matlab pehle se koi prefix sum same tha.
 \Rightarrow beech ka portion ka sum zero hoga.
`Length = i - mpp[sum]`
`maxi = max(maxi, i - mpp[sum])`
 - o Agar `sum` pehli baar aaya hai $\Rightarrow \text{mpp}[sum] = i$ store kar lo.
3. End me `maxi` return karo.

```
#include <bits/stdc++.h>
using namespace std;

int maxLen(int A[], int n) {
    unordered_map<int, int> mpp;
    int maxi = 0;
    int sum = 0;

    for (int i = 0; i < n; i++) {
        sum += A[i];

        if (sum == 0) {
            maxi = i + 1;
        } else {
            if (mpp.find(sum) != mpp.end()) {
                maxi = max(maxi, i - mpp[sum]);
            } else {
                mpp[sum] = i;
            }
        }
    }
    return maxi;
}

int main() {
```

```

int A[] = {9, -3, 3, -1, 6, -5};
int n = sizeof(A) / sizeof(A[0]);
cout << maxLen(A, n);
}

```

Time Complexity: O(N), as we are traversing the array only once

Space Complexity: O(N), in the worst case we would insert all array elements prefix sum into our hashmap

34. Count the number of subarrays with given xor K

Array diya hai (0/positive/negative sab ho sakte). Hume **continuous subarrays** ki count chahiye jinka **XOR = K** ho.

Example:

a = [4, 2, 2, 6, 4], K = 6 → **answer = 4**

Wo 4 subarrays:

1. [4, 2] ($4 \oplus 2 = 6$)
2. [4, 2, 2, 6, 4] index 0..4 ka kuch part? — actually correct 4 are:
 - o [4, 2] (0..1)
 - o [2, 2, 6] (1..3)
 - o [6] (3..3)
 - o [2, 6, 4] (2..4)

Approach 1: Brute Force ($O(N^3)$)

Idea: har (i, j) subarray banao, uska XOR ek aur loop se nikalo, agar == K to count++.

```
#include <bits/stdc++.h>
using namespace std;
```

```

int subarraysWithXorK(vector<int> a, int k) {
    int n = a.size(); //size of the given array.
    int cnt = 0;

    // Step 1: Generating subarrays:
    for (int i = 0; i < n; i++) {
        for (int j = i; j < n; j++) {

            //step 2:calculate XOR of all
            // elements:
            int xor = 0;
            for (int K = i; K <= j; K++) {
                xor = xor ^ a[K];
            }

            // step 3:check XOR and count:
            if (xor == k) cnt++;
        }
    }
    return cnt;
}

int main()
{
    vector<int> a = {4, 2, 2, 6, 4};
    int k = 6;
    int ans = subarraysWithXorK(a, k);
    cout << "The number of subarrays with XOR k is: "
         << ans << "\n";
    return 0;
}

```

Output: The number of subarrays with XOR k is: 4

Time Complexity: O(N³) approx., where N = size of the array.

Reason: We are using three nested loops, each running approximately N times.

Space Complexity: O(1) as we are not using any extra space.

⚡ Approach 2: Carry-forward XOR ($O(N^2)$)

Observation: Agar hum `i` fix kar dein, to `xr = 0` se start karke `j` badhate jao aur `xr ^= a[j]` rakhte jao. Har step pe check karo `xr==k`.

```
#include <bits/stdc++.h>
using namespace std;

int subarraysWithXorK(vector<int> a, int k) {
    int n = a.size(); //size of the given array.
    int cnt = 0;

    // Step 1: Generating subarrays:
    for (int i = 0; i < n; i++) {
        int xorrr = 0;
        for (int j = i; j < n; j++) {

            //step 2:calculate XOR of all
            // elements:
            xorrr = xorrr ^ a[j];

            // step 3:check XOR and count:
            if (xorrr == k) cnt++;
        }
    }
    return cnt;
}

int main()
{
    vector<int> a = {4, 2, 2, 6, 4};
    int k = 6;
    int ans = subarraysWithXorK(a, k);
    cout << "The number of subarrays with XOR k is: "
         << ans << "\n";
    return 0;
}
```

Output: The number of subarrays with XOR k is: 4

Complexity Analysis

Time Complexity: O(N²), where N = size of the array.

Reason: We are using two nested loops here. As each of them is running for N times, the time complexity will be approximately O(N²).

Space Complexity: O(1) as we are not using any extra space.

🚀 Approach 3: Optimal – Prefix XOR + HashMap (O(N))

Core intuition

Prefix XOR define karo: $\text{pref}[i] = a[0] \wedge a[1] \wedge \dots \wedge a[i]$

Kisi subarray $(l \dots r)$ ka XOR hota hai:

```
subXor = pref[r] ^ pref[l-1]      (l > 0)
subXor = pref[r]                  (l == 0)
```

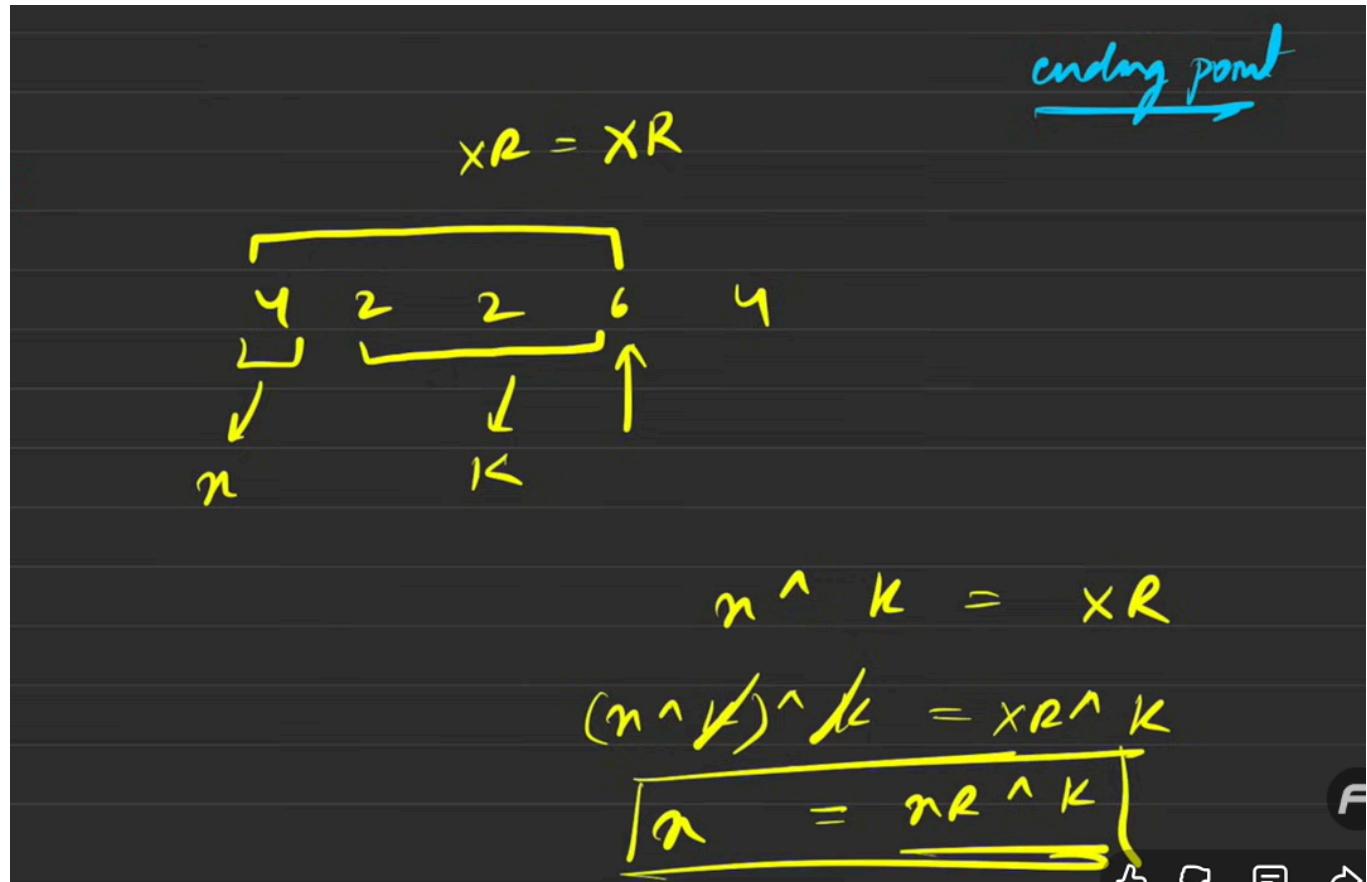
Hume $\text{subXor} == K$ chahiye \Rightarrow

```
pref[r] ^ pref[l-1] == K
pref[l-1] == pref[r] ^ K
```

Matlab: jab hum right end r par hain aur $xr = \text{pref}[r]$, to jitni baar $xr \wedge K$ pehle dekha gaya hai (as a prefix), utni subarrays end at r with XOR K.

Istiyeh:

- ek map rakho: $\text{freq}[\text{prefixXor}]$
- start me $\text{freq}[0] = 1$ (empty prefix)
- har element add karke $xr \wedge= a[i]$
 - $\text{need} = xr \wedge K$
 - $\text{cnt} += \text{freq}[\text{need}]$
 - $\text{freq}[xr]++$



```

#include <bits/stdc++.h>
using namespace std;

int subarraysWithXorK(vector<int> a, int k) {
    int n = a.size(); //size of the given array.
    int xr = 0;
    map<int, int> mpp; //declaring the map.
    mpp[xr]++; //setting the value of 0.
    int cnt = 0;

    for (int i = 0; i < n; i++) {
        // prefix XOR till index i:
        xr = xr ^ a[i];

        //By formula: x = xr^k:
        int x = xr ^ k;

        // add the occurrence of xr^k
    }
}

```

```

// to the count:
cnt += mpp[x];

// Insert the prefix xor till index i
// into the map:
mpp[xr]++;
}
return cnt;
}

int main()
{
    vector<int> a = {4, 2, 2, 6, 4};
    int k = 6;
    int ans = subarraysWithXorK(a, k);
    cout << "The number of subarrays with XOR k is: "
        << ans << "\n";
    return 0;
}

```

Output: The number of subarrays with XOR k is: 4

Complexity Analysis

Time Complexity: O(N) or O(N*logN) depending on which map data structure we are using, where N = size of the array.

Reason: For example, if we are using an unordered_map data structure in C++ the time complexity will be O(N) but if we are using a map data structure, the time complexity will be O(N*logN). The least complexity will be O(N) as we are using a loop to traverse the array. Point to remember for unordered_map in the worst case, the searching time increases to O(N), and hence the overall time complexity increases to O(N2).

Space Complexity: O(N) as we are using a map data structure.

Quick Dry Run (same example)

```

a = [4,2,2,6,4], K=6
freq = {0:1}, xr=0, cnt=0

```

1. x=4 → xr=4, need=4^6=2 → freq[2]=0 → cnt=0 → freq[4]=1
2. x=2 → xr=6, need=6^6=0 → freq[0]=1 → cnt=1 → freq[6]=1
3. x=2 → xr=4, need=4^6=2 → freq[2]=0 → cnt=1 → freq[4]=2

4. $x=6 \rightarrow xr=2$, need= $2^6=4$ $\rightarrow freq[4]=2 \rightarrow cnt=3 \rightarrow freq[2]=1$
 5. $x=4 \rightarrow xr=6$, need= $6^4=0$ $\rightarrow freq[0]=1 \rightarrow cnt=4 \rightarrow freq[6]=2$
- Final $cnt=4$ ✓

35. Merge Overlapping Sub-intervals

Intervals (start, end) diye hain. Hume **overlapping** intervals ko **merge** karke final non-overlapping list return karni hai.

Overlap rule (inclusive): Agar next interval ka `start <= current_end`, to overlap hai.

Example:

`[[1,3], [2,6], [8,10], [15,18]] → [[1,6], [8,10], [15,18]]`

Approach 1: Truly Brute Force ($O(N^2)$) — without sorting

Idea: Har interval ke saath baaki sab ko compare karo; jo overlap kare, unhe merge karke ek block banao; visited mark karke skip karo.

Do intervals `[a, b]` aur `[c, d]` overlap karte hain agar:

`c <= b && a <= d`

`[1,3]` aur `[2,6]` overlap karte hain

kyunki `2 <= 3`

- **Steps:**

1. `visited[n]=false.`
2. For each `i` (not visited), `cur = [L,R] = intervals[i].`

3. For each $j > i$ (not visited): agar $[L, R]$ aur $\text{intervals}[j]$ overlap karein, to $L = \min(L, \text{start}[j])$, $R = \max(R, \text{end}[j])$, $\text{visited}[j] = \text{true}$, and restart inner scan (ya ek while loop se club kar lo).

4. cur ko answer me daal do.

Jb hm sort akrte hai to wo 1st element k according sort hota hai, lekin agar at any moment 1st element is same then It will sort according to 2nd element.

```
#include <bits/stdc++.h>
using namespace std;

bool overlap(const vector<int>& a, const vector<int>& b){
    return !(a[1] < b[0] || b[1] < a[0]); // inclusive overlap
}

vector<vector<int>> mergeBrute(vector<vector<int>> intervals){
    int n = intervals.size();
    vector<vector<int>> ans;
    vector<int> vis(n, 0);

    for(int i = 0; i < n; i++){
        if(vis[i]) continue;
        int L = intervals[i][0], R = intervals[i][1];
        vis[i] = 1;

        bool mergedAny = true;
        while(mergedAny){
            mergedAny = false;
            for(int j = 0; j < n; j++){
                if(vis[j]) continue;
                if(overlap({L,R}, intervals[j])){
                    L = min(L, intervals[j][0]);
                    R = max(R, intervals[j][1]);
                    vis[j] = 1;
                    mergedAny = true;
                }
            }
        }
        ans.push_back({L,R});
    }
    return ans;
}
```

Complexity Analysis

Time Complexity: $O(N^2)$, for every interval we check all future intervals.

Space Complexity: ON , additional space used to store the non-overlapping intervals.

Approach 2: Optimal — Sort + Single Pass ($O(N \log N)$)

Observation: Agar intervals ko start ke hisaab se sort kar doge, to overlaps **sirf pichle merged interval** se check karne padte hain. Isse sirf **ek pass** me merge ho jayega.

- **Steps:**

1. `sort(intervals.begin(), intervals.end());`
2. `merged` empty ho to push first.
3. Har interval par:
 - Agar `interval.start > merged.back().end` → **no overlap** → push.
 - Warna **overlap** → `merged.back().end = max(merged.back().end, interval.end).`

- **Time:** Sorting $O(N \log N)$ + merge pass $O(N) \Rightarrow O(N \log N)$

- **Space:** If you return a new vector, $O(N)$ (output size). In-place bhi kiya ja sakta hai, par interviews me clean return preferred hota hai.

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
    // Function to merge overlapping intervals
    vector<vector<int>> merge(vector<vector<int>>& intervals) {
        // Sort intervals based on starting time
        sort(intervals.begin(), intervals.end());

        // Vector to store final merged intervals
        vector<vector<int>> merged;

        // Traverse each interval
        for (auto interval : intervals) {
            // If merged is empty or current interval does not overlap
```

```

        if (merged.empty() || merged.back()[1] < interval[0]) {
            // Add current interval as a new non-overlapping block
            merged.push_back(interval);
        } else {
            // Overlapping: merge by extending the end time
            merged.back()[1] = max(
                merged.back()[1],
                interval[1]
            );
        }
    }

    return merged;
}
};

int main() {
    Solution sol;
    vector<vector<int>> intervals = {
        {1, 3}, {2, 6}, {8, 10}, {15, 18}
    };

    vector<vector<int>> result = sol.merge(intervals);

    for (auto v : result) {
        cout << "[" << v[0] << "," << v[1] << "] ";
    }

    return 0;
}

```

Complexity Analysis

Time Complexity: $O(N \log N) + O(N)$, we sort the entire array and then merge them in a single pass.

Space Complexity: $O(N)$, additional space used to store the non-overlapping intervals.

36. Merge two Sorted Arrays Without Extra Space

Hume do **sorted arrays** diye gaye hain:

```
nums1 = [1, 3, 5, 0, 0, 0]
nums2 = [2, 4, 6]
```

nums1 ke end me kuch extra **0s** diye gaye hain (placeholders),
taaki uske andar hi **nums2** ke elements merge kiye jaa sake.

👉 Goal: dono ko mila kar ek **sorted array** banana hai —
but **without using any extra array (O(1) space)**.

Expected Output:

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

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
    // Merges nums2 into nums1 in-place in sorted order.
    void merge(vector<int>& nums1, int m, vector<int>& nums2, int n) {
        // i points to last valid element in nums1
        int i = m - 1;

        // j points to last element in nums2
        int j = n - 1;

        // k is the last index of nums1 (including 0 placeholders)
        int k = m + n - 1;

        // Fill nums1 from the end by comparing nums1[i] and nums2[j]
        while (i >= 0 && j >= 0) {
            // Place larger of the two at nums1[k]
            if (nums1[i] > nums2[j]) {
                nums1[k--] = nums1[i--];
            } else {
```

```

        nums1[k--] = nums2[j--];
    }
}

// If nums2 has remaining elements, copy them
while (j >= 0) {
    nums1[k--] = nums2[j--];
}

// No need to copy remaining nums1 elements, as they are already in place
}
};

int main() {
    vector<int> nums1 = {1, 3, 5, 0, 0, 0};
    vector<int> nums2 = {2, 4, 6};
    int m = 3, n = 3;

    Solution().merge(nums1, m, nums2, n);

    // Print merged array
    for (int num : nums1) cout << num << " ";
    return 0;
}

```

Complexity Analysis

Time Complexity: $O(N+M)$, we traverse both the arrays exactly once.

Space Complexity: $O(1)$, constant extra space is used to store pointers.

37. Find the repeating and missing numbers

Given an array **a** of size **n**,
 which contains **numbers from 1 to n**,
 but:

- ek number **repeats twice**,
- aur ek number **missing hai**.

Find both.

Input: a = [3, 1, 2, 5, 4, 6, 7, 5]
Output: Repeating = 5, Missing = 8

⚙️ Approach 1: Brute Force ($O(N^2)$)

💡 Idea:

1. For each number from 1 to n,
 - o count how many times it appears in the array.
2. If any number appears twice → repeating.
3. If any number appears zero times → missing.

```
#include <bits/stdc++.h>
using namespace std;

vector<int> findMissingRepeatingNumbers(vector<int> a) {
    int n = a.size(); // size of the array
    int repeating = -1, missing = -1;

    //Find the repeating and missing number:
    for (int i = 1; i <= n; i++) {
        //Count the occurrences:
        int cnt = 0;
        for (int j = 0; j < n; j++) {
            if (a[j] == i) cnt++;
        }

        if (cnt == 2) repeating = i;
        else if (cnt == 0) missing = i;
    }

    if (repeating != -1 && missing != -1)
        break;
}
return {repeating, missing};
}

int main()
{
    vector<int> a = {3, 1, 2, 5, 4, 6, 7, 5};
    vector<int> ans = findMissingRepeatingNumbers(a);
    cout << "The repeating and missing numbers are: {"
```

```

    << ans[0] << ", " << ans[1] << "}\n";
return 0;
}

```

Output: The repeating and missing numbers are: {5, 8}

Complexity Analysis

Time Complexity: O(N²), where N = size of the given array.

Reason: Here, we are using nested loops to count occurrences of every element between 1 to N.

Space Complexity: O(1) as we are not using any extra space.

Approach 2 — Using Frequency Array (O(N), O(N))

◆ Idea:

Ek **freq array** banao jisme har element ka count store karo.

- Agar freq[i] = 0 → missing
- Agar freq[i] = 2 → repeating

```

#include <bits/stdc++.h>
using namespace std;

vector<int> findMissingRepeatingNumbers(vector<int> a) {
    int n = a.size(); // size of the array
    int hash[n + 1] = {0}; // hash array

    //update the hash array:
    for (int i = 0; i < n; i++) {
        hash[a[i]]++;
    }

    //Find the repeating and missing number:

```

```

int repeating = -1, missing = -1;
for (int i = 1; i <= n; i++) {
    if (hash[i] == 2) repeating = i;
    else if (hash[i] == 0) missing = i;

    if (repeating != -1 && missing != -1)
        break;
}
return {repeating, missing};
}

int main()
{
    vector<int> a = {3, 1, 2, 5, 4, 6, 7, 5};
    vector<int> ans = findMissingRepeatingNumbers(a);
    cout << "The repeating and missing numbers are: {"
        << ans[0] << ", " << ans[1] << "}\n";
    return 0;
}

```

Output: The repeating and missing numbers are: {5, 8}

Complexity Analysis

Time Complexity: O(2N), where N = the size of the given array.

Reason: We are using two loops each running for N times. So, the time complexity will be O(2N).

Space Complexity: O(N) as we are using a hash array to solve this problem.

Approach 3 — Mathematical Formula (O(N), O(1))

- ◆ **Concept:**

Let actual numbers = 1, 2, 3, ..., n

Let given numbers = array elements (with one missing, one repeating)

We know formulas:

$$\text{Sum of } 1..n = n(n+1)/2$$

$$\text{Sum of squares of } 1..n = n(n+1)(2n+1)/6$$

Now let:

- S = actual sum ($1 + 2 + \dots + n$)
- $S2$ = actual sum of squares
- S_{actual} = sum of given array
- $S2_{actual}$ = sum of squares of given array

We have two equations:

$$(1) \quad S_{actual} - S = R - M$$
$$(2) \quad S2_{actual} - S2 = R^2 - M^2 = (R - M)(R + M)$$

From (1): $R - M = val1$

From (2): $R + M = val2 / val1$

Now:

$$R = (val1 + val2 / val1) / 2$$
$$M = R - val1$$

```
#include <bits/stdc++.h>
using namespace std;

vector<int> findMissingRepeatingNumbers(vector<int> a) {
    long long n = a.size(); // size of the array

    // Find Sn and S2n:
    long long SN = (n * (n + 1)) / 2;
    long long S2N = (n * (n + 1) * (2 * n + 1)) / 6;

    // Calculate S and S2:
    long long S = 0, S2 = 0;
    for (int i = 0; i < n; i++) {
        S += a[i];
        S2 += (long long)a[i] * (long long)a[i];
    }

    // S-Sn = X-Y:
    long long val1 = S - SN;

    // S2-S2n = X^2-Y^2:
```

```

long long val2 = S2 - S2N;

//Find X+Y = (X^2-Y^2)/(X-Y):
val2 = val2 / val1;

//Find X and Y: X = ((X+Y)+(X-Y))/2 and Y = X-(X-Y),
// Here, X-Y = val1 and X+Y = val2:
long long x = (val1 + val2) / 2;
long long y = x - val1;

return {(int)x, (int)y};
}

int main()
{
    vector<int> a = {3, 1, 2, 5, 4, 6, 7, 5};
    vector<int> ans = findMissingRepeatingNumbers(a);
    cout << "The repeating and missing numbers are: {" 
        << ans[0] << ", " << ans[1] << "}\n";
    return 0;
}

```

Output: The repeating and missing numbers are: {5, 8}

Complexity Analysis

Time Complexity: O(N), where N = the size of the given array.

Reason: We are using only one loop running for N times. So, the time complexity will be O(N).

Space Complexity: O(1) as we are not using any extra space to solve this problem.

Approach 4 — XOR Method (O(N), O(1)) (Can Ignore)

Concept:

XOR of numbers cancels out equal numbers.

① Take XOR of all elements in the array and all numbers from 1..n:

```
xorAll = (a[0] ^ a[1] ^ ... ^ a[n-1]) ^ (1 ^ 2 ^ ... ^ n)
```

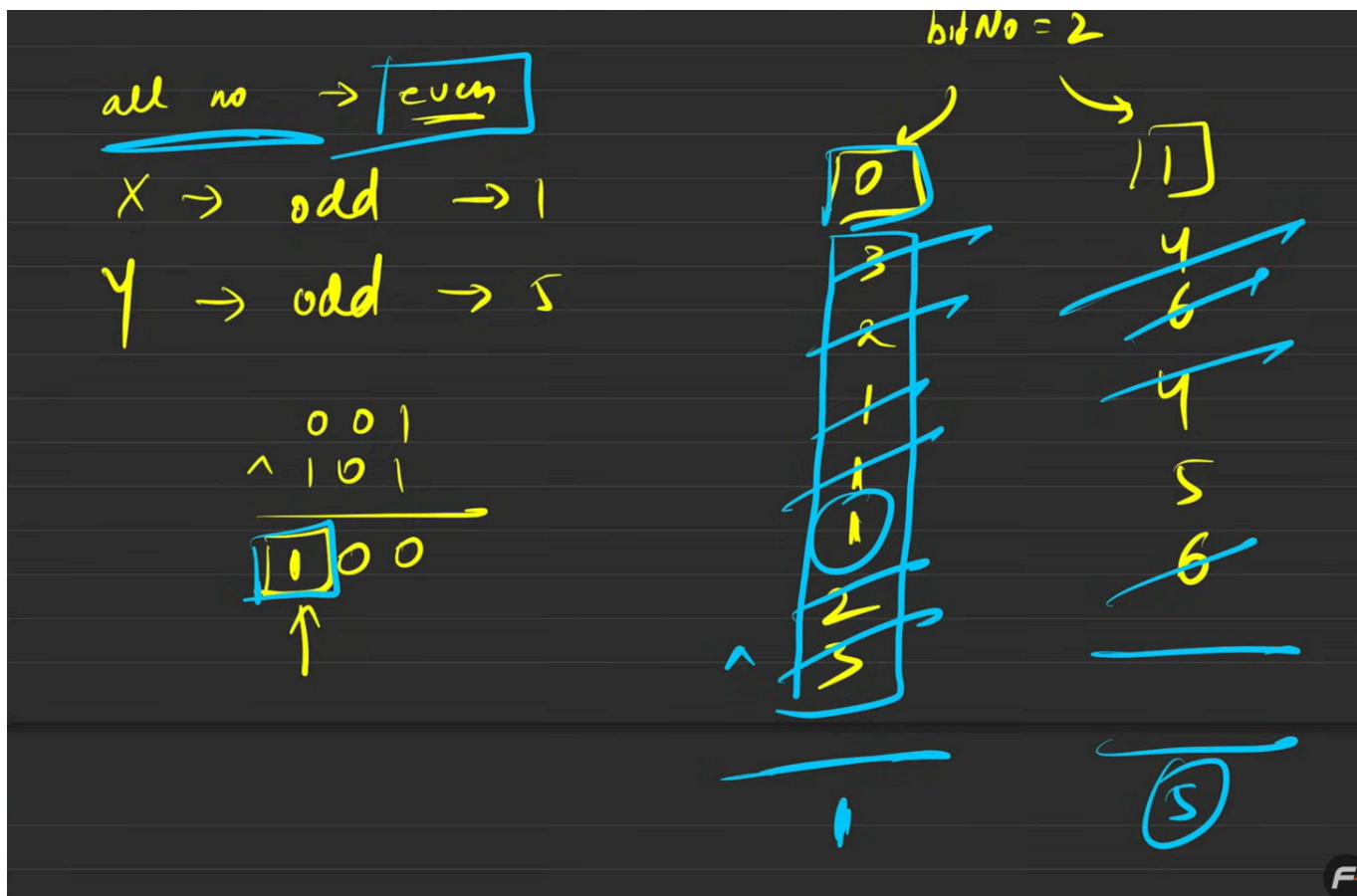
The result = $R \wedge M$ (since all others cancel).

② Find the rightmost set bit in `xorAll` (this helps separate R and M).

③ Divide numbers into two groups based on this bit and XOR separately.

That gives one as R and other as M .

Then, check which one actually repeats in the array.



♦ Step 1: XOR Concept Recap

Important XOR rules:

```
a ^ a = 0      // same numbers cancel out
a ^ 0 = a      // XOR with zero changes nothing
a ^ b = b ^ a  // commutative
```

◆ Step 2: First XOR of everything

We XOR all array elements and all numbers from 1..n.

```
xorAll = (3 ^ 1 ^ 2 ^ 5 ^ 4 ^ 6 ^ 7 ^ 5) ^ (1 ^ 2 ^ 3 ^ 4 ^ 5 ^ 6 ^ 7 ^ 8)
```

Why we do this?

Because every number that appears exactly once
(from both sides) will cancel out due to $a \wedge a = 0$.

Only two numbers will remain:

- the one that repeats (R)
- the one that is missing (M)

So:

```
xorAll = R ^ M
```

👉 For our example:

```
xorAll = 5 ^ 8
```

◆ Step 3: Problem — We only have XOR ($5 \wedge 8$)

We know only that $\text{xorAll} = 5 \wedge 8$,
but we can't directly tell which are 5 and 8 individually.

So we need to separate 5 and 8 somehow.

◆ Step 4: Find a bit where 5 and 8 differ

Let's write their binary:

```
5 = 0101
8 = 1000
-----
xorAll = 1101
```

The `xorAll` bits that are 1 tell us **where 5 and 8 differ**.

If we pick **any one differing bit** (like the rightmost 1-bit), then we can separate numbers based on that bit.

◆ Step 5: Find rightmost set bit

We need to pick **one bit position** where these two differ, so we choose the **rightmost set bit (1)** in `xorAll`.

For `xorAll = 13` (binary 1101)

Rightmost set bit = `0001` (the last '1' bit from the right).

This bit position is **different in 5 and 8**:

```
5 = 0101 → last bit = 1
8 = 1000 → last bit = 0
```

So we can now divide numbers into **two groups**:

- Group 1 → those having this bit = 1
- Group 2 → those having this bit = 0

That's how we'll separate R and M.

◆ Step 6: How to get that rightmost bit in code?

Line:

```
int rightmostSetBit = xorAll & (~xorAll - 1);
```

Let's decode this 

Say:

```
xorAll = 1101 (binary)  
xorAll - 1 = 1100  
~(xorAll - 1) = 0011
```

Now:

```
xorAll & (~(xorAll - 1))  
= 1101  
& 0011  
= 0001
```

 It gives **only the rightmost set bit.**
(i.e., the bit position where the least significant '1' occurs.)

So now `rightmostSetBit = 0001` in binary (means bit position 1).

- ◆ **Step 7: Divide into two groups using this bit**

Now traverse again over:

1. all elements in array, and
2. all numbers from 1 to n.

If that bit (0001) is **set**, put in Group X.

If not set, put in Group Y.

Then XOR all numbers within each group.

Because in each group, same numbers will again cancel out,
and only one unique number (either R or M) will remain.

Example (continue with our array):

Rightmost bit mask = **0001**

Number	Bit (0001)?	Group
3 (0011)	yes	X
1 (0001)	yes	X
2 (0010)	no	Y
5 (0101)	yes	X
4 (0100)	no	Y
6 (0110)	no	Y
7 (0111)	yes	X
5 (0101)	yes	X
1..8	similarly divided	both sides cancel except one in each group

After XORing within groups:

Group X → gives one number (say 5)

Group Y → gives the other (say 8)

Now we have **R** and **M** but don't know which is which.

We just check the array once:

- If number appears twice → repeating
- Otherwise → missing

```

vector<int> findMissingRepeatingNumbers(vector<int> a) {
    int n = a.size();
    int xorAll = 0;

    // Step 1: XOR of all array elements and numbers 1..n
    for (int i = 0; i < n; i++)
        xorAll ^= a[i];
    for (int i = 1; i <= n; i++)
        xorAll ^= i;

    // Step 2: xorAll = R ^ M
    int rightmostSetBit = xorAll & (~xorAll - 1);

    int x = 0, y = 0; // two buckets

    // Step 3: Divide into groups using the rightmost set bit
    for (int i = 0; i < n; i++) {
        if (a[i] & rightmostSetBit)
            x ^= a[i];
        else
            y ^= a[i];
    }
    for (int i = 1; i <= n; i++) {
        if (i & rightmostSetBit)
            x ^= i;
        else
            y ^= i;
    }

    // Step 4: x and y are R and M (check which is repeating)
    int count = 0;
    for (int num : a)
        if (num == x) count++;

    if (count == 2)
        return {x, y}; // x is repeating
    else
        return {y, x}; // y is repeating
}

```

Complexity Analysis

Time Complexity: $O(N)$, where N = the size of the given array.

Reason: We are just using some loops running for N times. So, the time complexity will be approximately $O(N)$.

Space Complexity: $O(1)$ as we are not using any extra space to solve this problem.

38. Count inversions in an array

Inversion: pair (i, j) jahan $i < j$ aur $a[i] > a[j]$.

Example: $a = [5, 4, 3, 2, 1]$

All pairs inverted \rightarrow count = 10.

Approach 1: Brute Force — $O(N^2)$, $O(1)$

Har (i, j) ($i < j$) pair check karo:

```
#include <bits/stdc++.h>
using namespace std;

int numberOfInversions(vector<int>&a, int n) {

    // Count the number of pairs:
    int cnt = 0;
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            if (a[i] > a[j]) cnt++;
        }
    }
}
```

```

    }
    return cnt;
}

int main()
{
    vector<int> a = {5, 4, 3, 2, 1};
    int n = 5;
    int cnt = numberOflnversions(a, n);
    cout << "The number of inversions is: "
        << cnt << endl;
    return 0;
}

```

Output: The number of inversions is: 10

Complexity Analysis

Time Complexity: $O(N^2)$, where N = size of the given array.

Reason: We are using nested loops here and those two loops roughly run for N times.

Space Complexity: $O(1)$ as we are not using any extra space to solve this problem.

Approach 2: Merge Sort se Count — $O(N \log N)$, $O(N)$

Core idea (intuition)

- Agar dono halves individually sorted hain:
 - Left: $L = a[\text{low}..\text{mid}]$
 - Right: $R = a[\text{mid}+1..\text{high}]$
- Merge karte waqt jab $R[\text{right}] < L[\text{left}]$ milta hai, to **Left me $\text{left}..\text{mid}$ ke saare elements $R[\text{right}]$ se bade hote hain** (kyunki left half sorted hoti hai).
Isliye **ek shot me ($\text{mid} - \text{left} + 1$) inversions** add kar dete hain.

```

#include <bits/stdc++.h>
using namespace std;

long long mergeAndCount(vector<int>& arr, int low, int mid, int high){
    vector<int> temp;

```

```

int i = low, j = mid+1;
long long cnt = 0;

while(i <= mid && j <= high){
    if(arr[i] <= arr[j]){
        temp.push_back(arr[i++]);
    } else {
        temp.push_back(arr[j++]);
        // arr[i]..arr[mid] sab > arr[j-1]
        cnt += (mid - i + 1);
    }
}
while(i <= mid) temp.push_back(arr[i++]);
while(j <= high) temp.push_back(arr[j++]);

for(int k=low; k<=high; k++) arr[k] = temp[k - low];
return cnt;
}

long long mergeSortCount(vector<int>& arr, int low, int high){
    if(low >= high) return 0;
    int mid = (low + high) / 2;
    long long cnt = 0;
    cnt += mergeSortCount(arr, low, mid);
    cnt += mergeSortCount(arr, mid+1, high);
    cnt += mergeAndCount(arr, low, mid, high);
    return cnt;
}

long long numberOfInversions(vector<int>& a, int n){
    // NOTE: ye function array ko sort bhi kar deta hai (side-effect).
    // Agar original order preserve karna ho to copy pe kaam karo.
    return mergeSortCount(a, 0, n-1);
}

int main(){
    vector<int> a = {5,4,3,2,1};
    cout << "The number of inversions is: "
         << numberOfInversions(a, (int)a.size()) << "\n";
}

```

Complexity Analysis

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

Reason: We are not changing the merge sort algorithm except by adding a variable to it. So, the time complexity is as same as the merge sort.

Space Complexity: $O(N)$, as in the merge sort We use a temporary array to store elements in sorted order.

39. Count Reverse Pairs

pairs (*i, j*) with *i < j* and $a[i] > 2 * a[j]$.

Example: $a = [4, 1, 2, 3, 1] \rightarrow$ answer 3

Brute Force ($O(N^2)$, $O(1)$)

Har (*i, j*) check karo:

```
#include <bits/stdc++.h>
using namespace std;

int countPairs(vector<int>&a, int n) {

    // Count the number of pairs:
    int cnt = 0;
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            if (a[i] > 2 * a[j]) cnt++;
        }
    }
    return cnt;
}

int team(vector <int> & skill, int n) {
```

```

    return countPairs(skill, n);
}

int main()
{
    vector<int> a = {4, 1, 2, 3, 1};
    int n = 5;
    int cnt = team(a, n);
    cout << "The number of reverse pair is: "
        << cnt << endl;
    return 0;
}

```

Output: The number of reverse pair is: 3

Time Complexity: $O(N^2)$, where N = size of the given array.

Reason: We are using nested loops here and those two loops roughly run for N times.

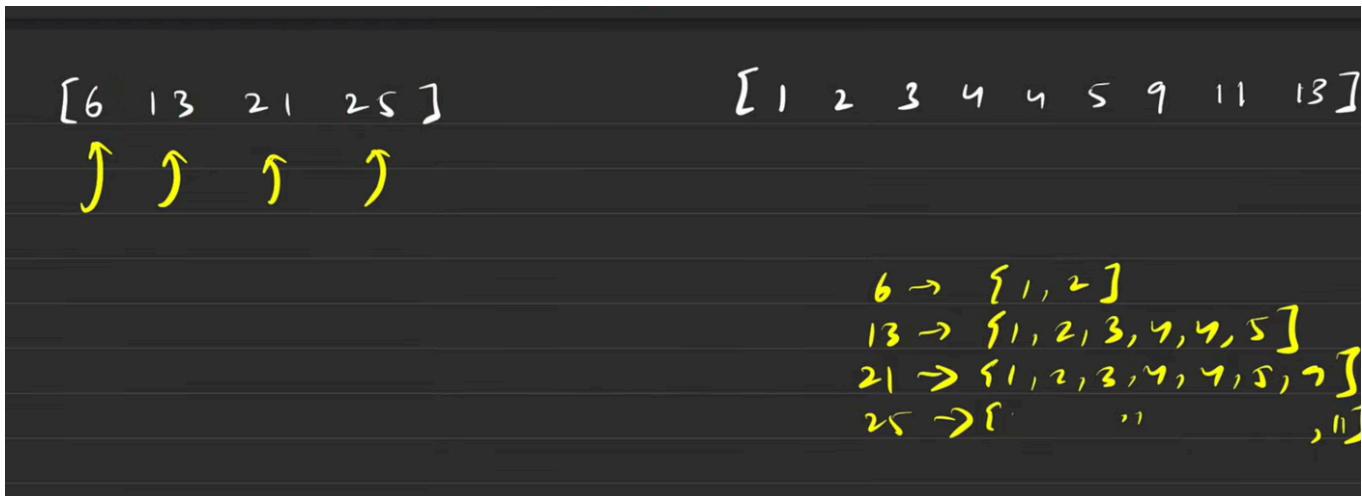
Space Complexity: $O(1)$ as we are not using any extra space to solve this problem.

Optimal: Merge Sort + Two-Pointer Count ($O(N \log N)$, $O(N)$)

Intuition

- Merge sort me halves **sorted** hote hain.
- Left half $L = a[low..mid]$, right half $R = a[mid+1..high]$.
- For each i in L , jitne $R[right]$ aise hain ki $L[i] > 2*R[right]$, **wo sab valid**.

Kyunki R sorted hai, right-pointer ko **sirf aage** badhata hai (reset nahi).



```
#include <bits/stdc++.h>
using namespace std;

void merge(vector<int> &arr, int low, int mid, int high) {
    vector<int> temp; // temporary array
    int left = low; // starting index of left half of arr
    int right = mid + 1; // starting index of right half of arr

    //storing elements in the temporary array in a sorted manner//

    while (left <= mid && right <= high) {
        if (arr[left] <= arr[right]) {
            temp.push_back(arr[left]);
            left++;
        }
        else {
            temp.push_back(arr[right]);
            right++;
        }
    }

    // if elements on the left half are still left //

    while (left <= mid) {
        temp.push_back(arr[left]);
        left++;
    }

    // if elements on the right half are still left //
    while (right <= high) {
```

```

        temp.push_back(arr[right]);
        right++;
    }

// transferring all elements from temporary to arr //
for (int i = low; i <= high; i++) {
    arr[i] = temp[i - low];
}
}

int countPairs(vector<int> &arr, int low, int mid, int high) {
    int right = mid + 1;
    int cnt = 0;
    for (int i = low; i <= mid; i++) {
        while (right <= high && arr[i] > 2 * arr[right]) right++;
        cnt += (right - (mid + 1));
    }
    return cnt;
}

int mergeSort(vector<int> &arr, int low, int high) {
    int cnt = 0;
    if (low >= high) return cnt;
    int mid = (low + high) / 2 ;
    cnt += mergeSort(arr, low, mid); // left half
    cnt += mergeSort(arr, mid + 1, high); // right half
    cnt += countPairs(arr, low, mid, high); //Modification
    merge(arr, low, mid, high); // merging sorted halves
    return cnt;
}

int team(vector <int> & skill, int n)
{
    return mergeSort(skill, 0, n - 1);
}

int main()
{
    vector<int> a = {4, 1, 2, 3, 1};
    int n = 5;
    int cnt = team(a, n);
    cout << "The number of reverse pair is: "
         << cnt << endl;
    return 0;
}

```

}

Output: The number of reverse pair is: 3

Time Complexity: $O(2N \cdot \log N)$, where N = size of the given array.

Reason: Inside the `mergeSort()` we call `merge()` and `countPairs()` except `mergeSort()` itself. Now, inside the function `countPairs()`, though we are running a nested loop, we are actually iterating the left half once and the right half once in total. That is why, the time complexity is $O(N)$. And the `merge()` function also takes $O(N)$. The `mergeSort()` takes $O(\log N)$ time complexity. Therefore, the overall time complexity will be $O(\log N * (N+N)) = O(2N \cdot \log N)$.

Space Complexity: $O(N)$, as in the merge sort We use a temporary array to store elements in sorted order.

40. Maximum Product Subarray in an Array

1) Brute Force — $O(N^2)$, $O(1)$

Idea: har `i` se start karke product carry-forward karo, har `j` par update.

- Kyun? Product recompute karne ke bajay, last product ko `* nums[j]` karke aage badh jao.
- Negatives/zeros handle ho jaate hain kyunki hum sab subarrays try kar rahe.

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
    // This function finds the maximum product
    // of any contiguous subarray using brute force
    int maxProduct(vector<int>& nums) {
        // Initialize the answer with the first element
        int maxProd = nums[0];

        // Outer loop picks the starting index
        for (int i = 0; i < nums.size(); i++) {
            // Initialize current product to 1
```

```

int prod = 1;

// Inner loop picks the ending index
for (int j = i; j < nums.size(); j++) {
    // Multiply current number to product
    prod *= nums[j];

    // Update maximum product if needed
    maxProd = max(maxProd, prod);
}

// Return the result
return maxProd;
};

int main() {
    // Sample input
    vector<int> nums = {2, 3, -2, 4};

    // Create Solution object
    Solution sol;

    // Print the result
    cout << sol.maxProduct(nums);

    return 0;
}

```

Complexity Analysis

Time Complexity: $O(N^2)$, we check the product of all possible subarrays using two nested loops.

Space Complexity: $O(1)$, No extra space is used.

2) Prefix-Suffix Trick — $O(N)$, $O(1)$

Observation:

- **Zero** product ko **break-point** samjho: zero aate hi koi bhi running product 0 ho jaata hai; zero ke baad naya subarray start maan lo.
- **Negative numbers**: odd count of negatives → overall product negative; par agar tum **aage se (prefix)** aur **peeche se (suffix)** dono taraf se multiply karo, to “odd negative ko cut” ho sakta hai

(jaise left se ek negative chhodke, right se ek chhodke).

- Isliye hum **left→right** aur **right→left** ek hi pass me products nikaalte hain, aur beech me zero milte hi running product ko 1 reset kar dete hain. Har step par `ans = max(ans, max(prefixProd, suffixProd))`.

Dry Run (`arr = [2, 3, -2, 4]`):

- Prefix: 2 → 6 → -12 → -48 → max so far = 6 (baad me answer 6 banega)
- Suffix: 4 → -8 → -24 → -48 → max so far = 6
Final = **6** ([2, 3]).

Zero Case (`arr = [0, -2, -3, 0, -2, -4]`):

- Prefix me zero pe reset (pre=1), Suffix me bhi reset.
- Left/right pass milke best chunk pick ho jaata.

Tip: `ans` ko `INT_MIN` rakhna sahi; values bada range ho sakti hain to `long long` prefix/suffix use karna safe hota hai.

Removal of 1 negative out of odd number of negatives will leave us with even number of negatives, hence the idea is to remove 1 negative, so we now see which 1 negative to remove, and on removal how is the array shaped.

```
#include <bits/stdc++.h>
using namespace std;

// This function returns the maximum product subarray
// using prefix and suffix traversal
class Solution {
public:
    int maxProductSubArray(vector<int>& arr) {
        // Store size of array
```

```

int n = arr.size();

// Initialize prefix and suffix product
int pre = 1, suff = 1;

// Initialize answer to negative infinity
int ans = INT_MIN;

// Traverse from both left and right
for (int i = 0; i < n; i++) {
    // Reset prefix if zero
    if (pre == 0) pre = 1;

    // Reset suffix if zero
    if (suff == 0) suff = 1;

    // Multiply prefix with current element from front
    pre *= arr[i];

    // Multiply suffix with current element from back
    suff *= arr[n - i - 1];

    // Update the maximum of all products seen so far
    ans = max(ans, max(pre, suff));
}

// Return the final answer
return ans;
};

int main() {
    // Sample input
    vector<int> arr = {2, 3, -2, 4};

    // Create object of solution
    Solution obj;

    // Call the function and print the result
    cout << obj.maxProductSubArray(arr) << endl;

    return 0;
}

```

Complexity Analysis

Time Complexity: O(N), every element of array is visited once.

Space Complexity: O(1), constant number of variables are used.

3) Optimal DP (Kadane-style for Product) — O(N), O(1) (IGNORE)

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
    // This function returns the maximum product
    // of any contiguous subarray using optimized approach
    int maxProduct(vector<int>& nums) {
        // Initialize answer, max and min product as first element
        int res = nums[0];
        int maxProd = nums[0];
        int minProd = nums[0];

        // Traverse from second element
        for (int i = 1; i < nums.size(); i++) {
            // Store current number
            int curr = nums[i];

            // If current number is negative, swap max and min
            if (curr < 0) swap(maxProd, minProd);

            // Update max and min product ending at current index
            maxProd = max(curr, maxProd * curr);
            minProd = min(curr, minProd * curr);

            // Update global result
            res = max(res, maxProd);
        }

        // Return the result
        return res;
    }
};

int main() {
    vector<int> nums = {2, 3, -2, 4};
```

```
Solution sol;  
cout << sol.maxProduct(nums);  
return 0;  
}
```

Complexity Analysis

Time Complexity: $O(N)$, every element of array is visited once.

Space Complexity: $O(1)$, only constant variables are used.

Binary Search

1. Binary Search

Sorted integer array di hui hai (duplicates nahi). Humein ek target dhoondhna hai — agar mile to uska index return karna, warna -1.

Array sorted hai, isliye har step par beech ka element check karte hain:

- agar beech wala element == target → mil gaya, return index.
- agar target > beech → target right half mein hoga → left = mid+1.
- agar target < beech → target left half mein hoga → right = mid-1.
Is process ko tab tak repeat karo jab tak low <= high. Agar cross ho gaye → nahi mila.

Iterative Code:

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
    // Iterative binary search
    int binarySearch(const vector<int>& nums, int target) {
        int low = 0, high = (int)nums.size() - 1;
        while (low <= high) {
            int mid = low + (high - low) / 2;
            if (nums[mid] == target) return mid;
            else if (nums[mid] < target) low = mid + 1;
            else high = mid - 1;
        }
        return -1;
    }
};

int main() {
    vector<int> a = {3, 4, 6, 7, 9, 12, 16, 17};
    int target = 6;

    Solution s;
    int ind = s.binarySearch(a, target);

    if (ind == -1) cout << "The target is not present.\n";
}
```

```

    else cout << "The target is at index: " << ind << "\n";
    return 0;
}

```

Complexity Analysis

Time Complexity: In the algorithm, in every step, we are basically dividing the search space into 2 equal halves. This is actually equivalent to dividing the size of the array by 2, every time. After a certain number of divisions, the size will reduce to such an extent that we will not be able to divide that anymore and the process will stop. The number of total divisions will be equal to the time complexity. So the overall time complexity is $O(\log N)$, where N = size of the given array.

Space Complexity: $O(1)$, no extra space being used

Recursive Approach:

```

#include <bits/stdc++.h>
using namespace std;

class Solution {
private:
    int binarySearchRec(const vector<int>& nums, int low, int high, int target) {
        if (low > high) return -1;
        int mid = low + (high - low) / 2;
        if (nums[mid] == target) return mid;
        if (nums[mid] < target) return binarySearchRec(nums, mid + 1, high, target);
        return binarySearchRec(nums, low, mid - 1, target);
    }
public:
    int search(const vector<int>& nums, int target) {
        return binarySearchRec(nums, 0, (int)nums.size() - 1, target);
    }
};

int main() {
    vector<int> a = {3, 4, 6, 7, 9, 12, 16, 17};
    int target = 6;

    Solution s;
    int ind = s.search(a, target);
}

```

```

if (ind == -1) cout << "The target is not present.\n";
else cout << "The target is at index: " << ind << "\n";
return 0;
}

```

Complexity Analysis

Time Complexity: In the algorithm, in every step, we are basically dividing the search space into 2 equal halves. This is actually equivalent to dividing the size of the array by 2, every time. After a certain number of divisions, the size will reduce to such an extent that we will not be able to divide that anymore and the process will stop. The number of total divisions will be equal to the time complexity. So the overall time complexity is $O(\log N)$, where N = size of the given array.

Space Complexity: $O(1)$, no extra space being used

2. Implement Lower Bound

Sorted array diya hua hai. Humein **lower bound** dhoondhna hai →
pehla index jahan arr[i] >= x ho.
Agar aisa koi index na mile → return n (array size).

2) Lower Bound Example (simple)

- {1, 2, 2, 3}, x=2 → **index 1** (pehli baar 2 ya usse bada value mila)
- {3, 5, 8, 15, 19}, x=9 → **index 3** ($15 \geq 9$)

Brute Force Approach

- Array ko left se right tak scan karo.
- Jiske pehli baar $\text{arr}[i] \geq x$ mile \rightarrow wahi answer.
- Nahi mila \rightarrow return n.

 Time: $O(N)$

 Space: $O(1)$

```
#include <bits/stdc++.h>

using namespace std;

class LowerBoundFinder {

public:

    int lowerBound(const vector<int>& arr, int n, int x) {

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

            if (arr[i] >= x) return i;

        }

        return n;

    }

};

int main() {

    vector<int> arr = {3, 5, 8, 15, 19};

    int n = arr.size();

    int x = 9;
```

```

LowerBoundFinder f;

cout << "The lower bound is the index: " << f.lowerBound(arr, n, x) << "\n";
return 0;
}

```

Optimal Approach

ahán sorted array ka fayda:

Binary search se sirf **arr[mid] >= x** condition check karna hai.

Cases:

1. **arr[mid] >= x**
 - mid answer ho sakta hai → ans = mid
 - left side me aur chhota valid index ho sakta hai → high = mid - 1

2. **arr[mid] < x**
 - mid chhota hai → bigger element right side me
 - low = mid + 1

Loop khatam → ans return.

 Time: O(log N)
 Space: O(1)

```

#include <bits/stdc++.h>
using namespace std;

class LowerBoundFinder {
public:
    int lowerBound(const vector<int>& arr, int n, int x) {
        int low = 0, high = n - 1;

```

```

int ans = n;

while (low <= high) {
    int mid = low + (high - low) / 2;

    if (arr[mid] >= x) {
        ans = mid;      // potential answer
        high = mid - 1; // search left side
    } else {
        low = mid + 1; // search right side
    }
}
return ans;
};

int main() {
    vector<int> arr = {3, 5, 8, 15, 19};
    int n = arr.size();
    int x = 9;

    LowerBoundFinder f;
    cout << "The lower bound is the index: " << f.lowerBound(arr, n, x) << "\n";
    return 0;
}

```

3. Implement Upper Bound

Sorted array `arr` of size `N` and integer `x` diye hain. **Upper bound** = smallest index `i` such that `arr[i] > x`.

Agar aisa index na mile → return `N`.

Sorted hone ki wajah se first element **strictly greater** than x ko binary search se jaldi find kar sakte ho; otherwise scan linearly.

Brute Force

- Left se iterate karo.
- Pehli position jahan $\text{arr}[i] > x$ ho \rightarrow return i.
- Agar loop finish ho gaya \rightarrow return N.

Why O(N)? worst-case jab sab elements $\leq x$ ho tab tumhe poora array traverse karna padega
 $\rightarrow N$ comparisons.

Space O(1): koi extra data structure ya recursion stack nahi.

```
#include <bits/stdc++.h>
using namespace std;

int upperBoundBrute(const vector<int>& arr, int x) {
    int n = (int)arr.size();
    for (int i = 0; i < n; ++i) {
        if (arr[i] > x) return i;
    }
    return n;
}

int main() {
    vector<int> arr = {1,2,2,3,3,5};
    int x = 2;
    cout << "Upper bound index (brute): " << upperBoundBrute(arr, x) << "\n"; // expected 3
    return 0;
}
```

Optimal (Binary Search)

Use binary search to maintain a candidate ans (initialized to n).

At each step compute $\text{mid} = \text{low} + (\text{high} - \text{low})/2$ (safer against overflow).

- If $\text{arr}[\text{mid}] > x$: mid is a potential answer → store $\text{ans} = \text{mid}$, move left ($\text{high} = \text{mid} - 1$) to find smaller index.
- Else ($\text{arr}[\text{mid}] \leq x$): discard left part ($\text{low} = \text{mid} + 1$).

Loop ends when $\text{low} > \text{high}$. Return ans.

Why $O(\log N)$? har step search space half ho jata hai → number of steps $\sim \log_2 N$.
Space $O(1)$: iterative, no extra memory or recursion.

```
#include <bits/stdc++.h>
using namespace std;

int upperBound(const vector<int>& arr, int x) {
    int n = (int)arr.size();
    int low = 0, high = n - 1;
    int ans = n; // default if no element > x

    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (arr[mid] > x) {
            ans = mid; // possible answer
            high = mid - 1; // search left for smaller index
        } else {
            low = mid + 1; // arr[mid] <= x, ignore left half
        }
    }
    return ans;
}

int main() {
    vector<int> arr = {3, 5, 8, 9, 15, 19};
    int x = 9;
```

```

cout << "Upper bound index: " << upperBound(arr, x) << "\n"; // expected 4
return 0;
}

```

Edge cases to mention

- Empty array → return 0 (because $n = 0$).
- All elements $\leq x$ → return n .
- All elements $> x$ → return 0.
- Duplicates around x (e.g., many elements equal to x) → upper bound returns first index **after** last x .

4. Search Insert Position

You have a **sorted array with distinct values** and a **target x**.

You must return the index where:

- target exists **OR**
- target should be inserted to keep array sorted.

This is the same as **lower bound**, because we need the first index where $arr[i] \geq x$.

Because the array is sorted, the index where x should be inserted is:

👉 **the first element which is greater than or equal to x**

If no such element exists → x should be inserted at the end → return n .

This is exactly the same binary search we used for **lower_bound**.

We maintain:

- $\text{low} = 0$
- $\text{high} = n - 1$
- $\text{ans} = n$ (default → insert at end if nothing $\geq x$)

Steps:

1. Calculate $\text{mid} = \text{low} + (\text{high} - \text{low}) / 2$
2. Compare $\text{arr}[\text{mid}]$ with x
 - If $\text{arr}[\text{mid}] \geq x$
 - possible insertion index
 - update $\text{ans} = \text{mid}$
 - search left half ($\text{high} = \text{mid} - 1$)
 - Else
 - x is on right side
 - $\text{low} = \text{mid} + 1$
3. Continue until $\text{low} > \text{high}$
4. Return ans

```
#include <bits/stdc++.h>

using namespace std;

class Solution {

public:

    int searchInsert(const vector<int>& arr, int x) {

        int n = arr.size();

        int low = 0, high = n - 1;

        int ans = n; // default if x should be placed at the end

        while (low <= high) {

            int mid = low + (high - low) / 2;
```

```

        if (arr[mid] >= x) {
            ans = mid;      // potential insert position
            high = mid - 1; // search left side
        } else {
            low = mid + 1; // search right side
        }
    }

    return ans;
}

};

int main() {
    vector<int> arr = {1, 2, 4, 7};
    int x = 6;

    Solution s;
    cout << "The index is: " << s.searchInsert(arr, x) << "\n";
    return 0;
}

```

5. Floor and Ceil in Sorted Array

Given a **sorted array arr** of size n and a value x, find:

- **Floor(x)**: largest value $\leq x$
- **Ceil(x)**: smallest value $\geq x$

If floor or ceil does not exist → return -1.

Binary Search works perfectly because:

- **Floor** is the **rightmost** value $\leq x$
- **Ceil** is the **leftmost** value $\geq x$

So both can be found by slightly modifying binary search.

Floor Algorithm (Simple Explanation)

Goal → largest element $\leq x$

Steps:

1. low = 0, high = n-1, ans = -1
2. Find mid
3. If $\text{arr}[\text{mid}] \leq x$:
 - it is a valid floor → update $\text{ans} = \text{arr}[\text{mid}]$
 - but maybe a larger valid floor exists → go right ($\text{low} = \text{mid} + 1$)
4. Else ($\text{arr}[\text{mid}] > x$):
 - too big → go left ($\text{high} = \text{mid} - 1$)
5. Continue until $\text{low} > \text{high}$
6. Return ans

Ceil Algorithm (Simple Explanation)

Goal → smallest element $\geq x$

Steps:

1. low = 0, high = n-1, ans = -1
2. Find mid

3. If $\text{arr}[\text{mid}] \geq x$:
 - valid ceil → update $\text{ans} = \text{arr}[\text{mid}]$
 - try to find smaller ceil → go left ($\text{high} = \text{mid} - 1$)
4. Else ($\text{arr}[\text{mid}] < x$):
 - too small → go right ($\text{low} = \text{mid} + 1$)
5. Return ans

```
#include <bits/stdc++.h>
using namespace std;

class FloorCeilFinder {
public:
    int findFloor(const vector<int>& arr, int x) {
        int low = 0, high = arr.size() - 1;
        int ans = -1;

        while (low <= high) {
            int mid = low + (high - low) / 2;

            if (arr[mid] <= x) {
                ans = arr[mid]; // potential floor
                low = mid + 1; // try to find a larger floor
            } else {
                high = mid - 1; // go left
            }
        }
        return ans;
    }

    int findCeil(const vector<int>& arr, int x) {
        int low = 0, high = arr.size() - 1;
        int ans = -1;

        while (low <= high) {
            int mid = low + (high - low) / 2;

            if (arr[mid] >= x) {
                ans = arr[mid]; // potential ceil
                high = mid - 1; // try to find smaller ceil
            } else {
                low = mid + 1; // go right
            }
        }
        return ans;
    }
}
```

```

    }
    return ans;
}

pair<int, int> getFloorAndCeil(const vector<int>& arr, int x) {
    int floorVal = findFloor(arr, x);
    int ceilVal = findCeil(arr, x);
    return {floorVal, ceilVal};
}
};

int main() {
    vector<int> arr = {3, 4, 4, 7, 8, 10};
    int x = 5;

    FloorCeilFinder finder;
    auto ans = finder.getFloorAndCeil(arr, x);

    cout << "Floor and Ceil: " << ans.first << " " << ans.second << "\n";
    return 0;
}

```

Time Complexity: O(log N)

Because both floor and ceil use binary search:

- Search space halves every iteration
- Steps required $\approx \log_2(N)$

Even two searches $\rightarrow O(\log N) + O(\log N) = O(\log N)$.

Space Complexity: O(1)

- Only a few variables: `low`, `high`, `mid`, `ans`
- No recursion, no extra arrays
- Constant memory usage.

Quick Dry Run ($x = 5$, $\text{arr} = \{3,4,4,7,8,10\}$)

Floor

- $3 \leq 5 \rightarrow \text{floor} = 3 \rightarrow \text{go right}$
- $4 \leq 5 \rightarrow \text{floor} = 4 \rightarrow \text{go right}$
- $4 \leq 5 \rightarrow \text{floor} = 4 \rightarrow \text{go right}$
- $7 > 5 \rightarrow \text{go left}$
→ Final floor = **4**

Ceil

- $4 < 5 \rightarrow \text{right}$
- $4 < 5 \rightarrow \text{right}$
- $7 \geq 5 \rightarrow \text{ceil} = 7 \rightarrow \text{left}$
→ Final ceil = **7**

6. Last occurrence in a sorted array

You are given a **sorted array** of size N and a target.

You must return the **index of the last occurrence** of the target
(0-based indexing).

If target does NOT exist → return -1.

Because the array is sorted:

- The duplicates (if any) are grouped together.
- The **last occurrence** will be at the **rightmost** index where $\text{arr}[i] == \text{target}$.

Binary search can do this faster by:

- Finding target
- Then still searching **right side** for any later occurrence.

Brute Force

Start from the **end** of the array and move backward:

- As soon as you find $v[i] == \text{key}$, return i .
- If not found \rightarrow return -1 .

Time Complexity $\rightarrow O(N)$

Worst case: target not found \rightarrow you scan all N elements.

Space Complexity $\rightarrow O(1)$

```
#include <bits/stdc++.h>
using namespace std;

int lastOccurrenceBrute(int n, int key, const vector<int>& v) {
    for (int i = n - 1; i >= 0; i--) {
        if (v[i] == key) return i;
    }
    return -1;
}
```

Optimized Binary Search

We maintain:

- $\text{start} = 0$
- $\text{end} = n - 1$
- $\text{res} = -1$ (default if target not found)

During search:

1. Compute mid
2. If $v[\text{mid}] == \text{key} \rightarrow$
 - store mid in res

- o move start = mid + 1 (because last occurrence may be to the right)
3. If key < v[mid] → search left (end = mid - 1)
 4. Else → search right (start = mid + 1)

Loop continues until start > end.

Why It Works?

Because even after finding the target, we **force binary search to continue rightwards** to capture the last occurrence.

Time Complexity → O(log N)

Binary search halves the search space every step.

Space Complexity → O(1)

Only variables used: start, end, mid, res.

```
#include <bits/stdc++.h>
using namespace std;

int lastOccurrence(int n, int key, const vector<int>& v) {
    int start = 0, end = n - 1;
    int res = -1;

    while (start <= end) {
        int mid = start + (end - start) / 2;

        if (v[mid] == key) {
            res = mid; // store the index
            start = mid + 1; // keep searching on the right side
        }
        else if (key < v[mid]) {
            end = mid - 1; // go left
        }
        else {
            start = mid + 1; // go right
        }
    }
    return res;
}
```

```

}

int main() {
    vector<int> v = {3,4,13,13,13,20,40};
    int key = 13;
    int n = v.size();

    cout << lastOccurrence(n, key, v) << "\n"; // Output: 4
    return 0;
}

```

Quick Dry Run (Example 1)

Array: {3, 4, 13, 13, 13, 20, 40}, key = 13

start	en	mid	v[mid]	action
d				
0	6	3	13	res=3, move right (start=4)
4	6	5	20	key < v[mid] → end=4
4	4	4	13	res=4, move right (start=5)
5	4		stop	

Final answer = **4**

For target = 13 → last occurrence index = 4

For target = 60 → -1 (not found)

7. Count Occurrences in Sorted Array

You have a **sorted array** of integers and a number **X**.

Your job is to find **how many times X occurs** in the array.

Because the array is sorted:

- All occurrences of X will be **together in a block**.
- If we find:
 - **first occurrence index**
 - **last occurrence index**

Then:

```
count = last - first + 1
```

Binary search is perfect for finding first and last position efficiently.

Brute Force Approach

Traverse the array and count how many times `arr[i] == X`.

✓ Time Complexity

$O(N)$ → worst case check all elements.

✓ Space Complexity

$O(1)$ → no extra memory.

```
int countBrute(const vector<int>& arr, int n, int x) {  
    int cnt = 0;  
  
    for (int i = 0; i < n; i++) {  
        if (arr[i] == x) cnt++;  
    }  
    return cnt;  
}
```

```

    }
    return cnt;
}

```

Optimal Approach — Using Binary Search

We use **two modified binary searches**:

1. **firstOccurrence()** – find first index where arr[mid] == x
2. **lastOccurrence()** – find last index where arr[mid] == x

Then compute:

```
count = last - first + 1
```

If **first** == -1 → x not present → return 0.

First Occurrence

- If arr[mid] == x → store mid, move **left** (high = mid - 1)
- Else normal binary search rules.

Last Occurrence

- If arr[mid] == x → store mid, move **right** (low = mid + 1)
- Else normal binary search rules.

This ensures we find the extreme ends of X's block.

```
#include <bits/stdc++.h>

using namespace std;

int firstOccurrence(const vector<int>& arr, int n, int x) {
    int low = 0, high = n - 1, first = -1;
```

```

while (low <= high) {

    int mid = low + (high - low) / 2;

    if (arr[mid] == x) {

        first = mid;      // possible first occurrence

        high = mid - 1;   // go left to find earlier occurrence

    }

    else if (arr[mid] < x) {

        low = mid + 1;   // go right

    }

    else {

        high = mid - 1;   // go left

    }

}

return first;
}

int lastOccurrence(const vector<int>& arr, int n, int x) {

    int low = 0, high = n - 1, last = -1;

    while (low <= high) {

        int mid = low + (high - low) / 2;

```

```

if (arr[mid] == x) {
    last = mid;      // possible last occurrence
    low = mid + 1;   // go right to find later occurrence
}
else if (arr[mid] < x) {
    low = mid + 1;   // go right
}
else {
    high = mid - 1; // go left
}
}

return last;
}

```

```

int countOccurrences(const vector<int>& arr, int n, int x) {
    int first = firstOccurrence(arr, n, x);
    if (first == -1) return 0; // x not present
    int last = lastOccurrence(arr, n, x);
    return last - first + 1;
}

```

```

int main() {
    vector<int> arr = {2, 2, 3, 3, 3, 4};
    int x = 3;
}

```

```

int n = arr.size();

cout << "Total occurrences: " << countOccurrences(arr, n, x) << "\n";

return 0;

}

```

Time Complexity → O(log N)

- first occurrence → O(log N)
- last occurrence → O(log N)
Total → $O(2 \times \log N) = O(\log N)$

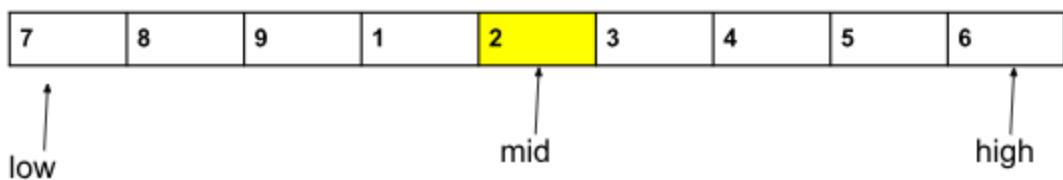
Space Complexity → O(1)

Only a few variables, no extra storage.

8. Search Element in a Rotated Sorted Array

Given a sorted array of **distinct** integers that has been rotated at an unknown pivot, find the index of target k. If k is not present return -1.
(0-based indexing)

Even after rotation, **one** half (left or right of mid) is always sorted. Find which half is sorted, check if k lies inside that sorted half; if yes, search there, otherwise search the other half. Repeat (binary search style).



target = 8,

Considering the comparison made, such as target > arr[mid] (e.g., 8 > 2), we would expect the target to be in the right half. However, due to the array rotation, the number 8 is actually situated in the left half. This rotation creates a challenge in the elimination process.

Brute force

Scan array linearly, return i when $\text{arr}[i] == k$, else -1.

- Time: $O(N)$
- Space: $O(1)$

```
int linearSearch(const vector<int>& a, int k) {
    for (int i = 0; i < (int)a.size(); ++i)
        if (a[i] == k) return i;
    return -1;
}
```

Optimal approach (binary search on rotated array)

Either of one half is sorted

At each step:

- compute mid
- if $a[\text{mid}] == k \rightarrow \text{return mid}$
- else check if left half $a[\text{low}]..a[\text{mid}]$ is sorted ($a[\text{low}] \leq a[\text{mid}]$)
 - if sorted and $a[\text{low}] \leq k < a[\text{mid}] \rightarrow \text{search left (high} = \text{mid} - 1\text{)}$
 - else $\rightarrow \text{search right (low} = \text{mid} + 1\text{)}$

- otherwise right half is sorted
 - if $a[mid] < k \leq a[high]$ → search right ($low = mid + 1$)
 - else → search left ($high = mid - 1$)

Repeat until $low > high$.

This keeps $O(\log N)$ time and $O(1)$ space.

```
#include <bits/stdc++.h>
using namespace std;

// Returns index of k in rotated sorted array a, or -1 if not found.
// Assumes all elements are distinct.
int searchInRotated(const vector<int>& a, int k) {
    int n = (int)a.size();
    int low = 0, high = n - 1;

    while (low <= high) {
        int mid = low + (high - low) / 2;

        if (a[mid] == k) return mid;

        // If left half [low..mid] is sorted
        if (a[low] <= a[mid]) {
            if (a[low] <= k && k < a[mid]) {
                high = mid - 1; // k in left sorted half
            } else {
                low = mid + 1; // k in right half
            }
        }
        else { // right half [mid..high] is sorted
            if (a[mid] < k && k <= a[high]) {
                low = mid + 1; // k in right sorted half
            } else {
                high = mid - 1; // k in left half
            }
        }
    }
    return -1; // not found
}

int main() {
    vector<int> arr = {7,8,9,1,2,3,4,5,6};
    int k = 1;
```

```
int idx = searchInRotated(arr, k);
if (idx == -1) cout << "Target is not present.\n";
else cout << "The index is: " << idx << "\n"; // expected 3
return 0;
}
```

9. Search Element in Rotated Sorted Array II

You are given a rotated sorted array (but now duplicates are allowed).
You must return **True** if element k exists, otherwise **False**.

Example:

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

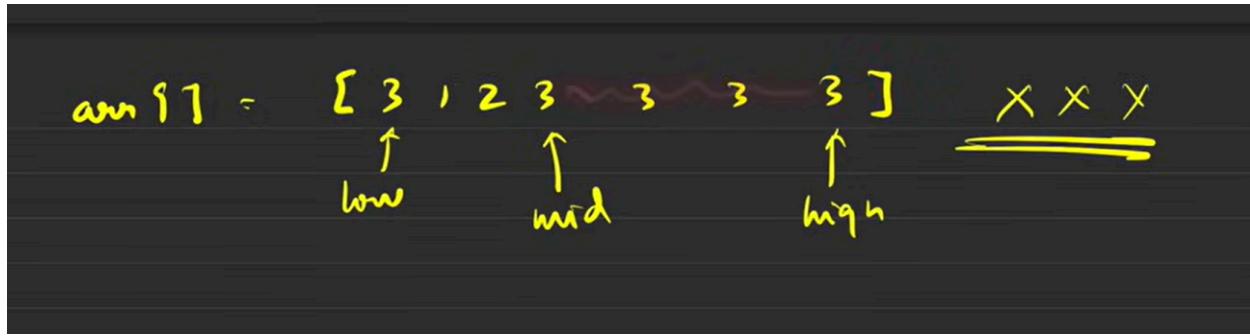
Sorted + rotated + duplicates.

Why this is harder than “Rotated Sorted Array I”? (Distinct Version)

In the distinct version:

👉 One half is **ALWAYS** sorted
(either left or right)

But **in duplicates version** this strict rule sometimes breaks.



Example:

[3, 3, 3, 3, 3]

Here:

`arr[low] = arr[mid] = arr[high] -> 3 = 3 = 3`

Now, can you say left is sorted? right is sorted?

NO.

Both look identical.

The algorithm **cannot determine** the sorted half.

This is the **ONLY** reason why duplicates break the clean logic.

How do we fix the problem?

When you see:

`arr[low] == arr[mid] == arr[high]`

You cannot decide which half is sorted.

So the trick is:

👉 just shrink the array

`low++`

`high--`

This safely removes duplicate boundary noise
without affecting correctness.

After shrinking, one half will again become detectable.

This trick is the heart of the algorithm.

Full Intuition

At every step:

✓ **Step 1: Check mid**

If $\text{arr}[\text{mid}] == \text{k} \rightarrow \text{found} \rightarrow \text{return true}$

✓ **Step 2: Handle the duplicate ambiguity**

If

$\text{arr}[\text{low}] == \text{arr}[\text{mid}] \& \& \text{arr}[\text{mid}] == \text{arr}[\text{high}]$

then:

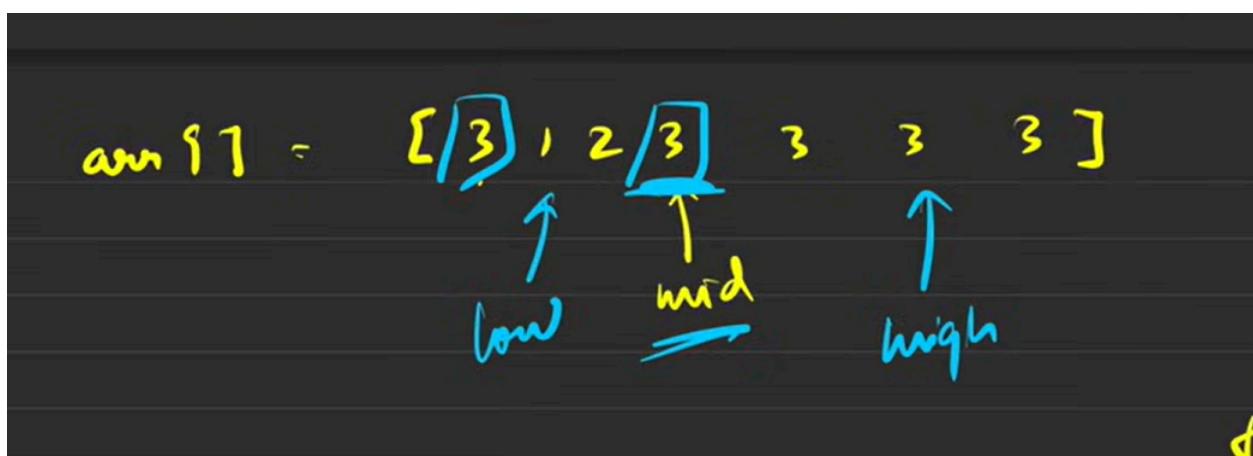
👉 We cannot know which half is sorted

👉 So shrink:

$\text{low}++$

$\text{high}--$

continue



✓ **Step 3: Identify which half is sorted (once duplicates are handled)**

Case A: Left half sorted

```
arr[low] <= arr[mid]
```

Check if k lies in that left sorted range.

Case B: Right half sorted

Else → right half sorted

Check if k lies in that right sorted range.

```
#include <bits/stdc++.h>
using namespace std;

bool searchInRotatedArrayII(vector<int>& arr, int k) {
    int low = 0, high = arr.size() - 1;

    while (low <= high) {

        int mid = low + (high - low) / 2;

        // Case 1: found
        if (arr[mid] == k) return true;

        // Case 2: Cannot determine sorted half (duplicate trap)
        if (arr[low] == arr[mid] && arr[mid] == arr[high]) {
            low++;
            high--;
            continue;
        }

        // Case 3: Left half sorted
        if (arr[low] <= arr[mid]) {
            if (arr[low] <= k && k < arr[mid]) {
                high = mid - 1; // target lies in left
            } else {
                low = mid + 1; // target lies in right
            }
        }

        // Case 4: Right half sorted
        else {
            if (arr[mid] < k && k <= arr[high]) {
                low = mid + 1; // target lies in right
            }
        }
    }
}
```

```

        } else {
            high = mid - 1; // target lies in left
        }
    }

return false;
}

int main() {
    vector<int> arr = {7, 8, 1, 2, 3, 3, 3, 4, 5, 6};
    int k = 3;
    cout << (searchInRotatedArrayII(arr, k) ? "True" : "False");
}

```

This problem CANNOT guarantee $O(\log N)$ for all cases.

✓ **Best/Average: $O(\log N)$**

Because it behaves like normal binary search most of the time.

✓ **Worst Case: $O(N)$**

Why?

When many duplicates are present:

Example:

[3, 3, 3, 3, 3, 3, 3]

Every time:

`arr[low] == arr[mid] == arr[high]`

So binary search degenerates to linear shrinking:

`low++`

`high--`

Thus worst-case = $N/2 \rightarrow O(N)$

10. Minimum in Rotated Sorted Array

Given a **sorted array (distinct values)** that is **rotated at an unknown pivot**, find the **minimum element**.

[4,5,6,7,0,1,2] → 0
[3,4,5,1,2] → 1

A rotated sorted array looks like this:

Original: [1, 2, 3, 4, 5]
Rotated : [4, 5, 1, 2, 3]

Notice:

- 👉 The array is split into **two sorted parts**.
- 👉 The **minimum element** is the **rotation point**.
- 👉 It is the **only element** that is smaller than its previous element.

This makes the problem perfect for **binary search**.

Brute Force

Traverse the entire array and keep track of the smallest element.

- Time: **O(N)**
- Space: **O(1)**

```
int findMinBrute(vector<int>& nums) {  
    int ans = INT_MAX;  
    for (int x : nums) ans = min(ans, x);  
    return ans;  
}
```

Optimal Approach — Binary Search

Even after rotation:

- **Right half is sorted** when $\text{nums}[\text{mid}] \leq \text{nums}[\text{high}]$
- **Left half is sorted** when $\text{nums}[\text{mid}] \geq \text{nums}[\text{low}]$

But we don't care which half is sorted.

We only care:

👉 Is the minimum on the right or left?

The Rule (Very Easy)

Compare $\text{nums}[\text{mid}]$ with $\text{nums}[\text{high}]$:

Case 1: $\text{nums}[\text{mid}] > \text{nums}[\text{high}]$

This means mid is in the **left sorted part**,
so the minimum must be on the **right side**:

```
[4, 5, 6, 7, 0, 1, 2]  
      ↑      ↑  
    mid     high  
mid > high → go right
```

So:

```
low = mid + 1
```

Case 2: $\text{nums}[\text{mid}] < \text{nums}[\text{high}]$

This means the **right part is sorted**,
so the minimum is either at mid or in the left segment:

```
[4, 5, 6, 1, 2]  
      ↑ mid  
nums[mid] < nums[high] → min is left side
```

So:

```
high = mid
```

✓ **Loop ends when low == high**

The index of minimum is found.

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
    int findMin(vector<int>& nums) {
        int low = 0, high = nums.size() - 1;

        while (low < high) {
            int mid = low + (high - low) / 2;

            if (nums[mid] > nums[high]) {
                // Minimum lies in the right half
                low = mid + 1;
            } else {
                // Minimum is in left half (including mid)
                high = mid;
            }
        }

        return nums[low]; // low == high
    };
}
```

Time Complexity: O(log N)

Binary search divides the array into half every step.

Space Complexity: O(1)

Only low, high, mid used.

11. Find out how many times the array has been rotated

Sorted array (distinct values) ko unknown pivot par rotate kiya gaya hai. Hume pata karna hai **kitni baar array rotate** hua — yani original sorted array ke mukable current array ka **smallest element ka index**. Woh index hi rotation count hota hai.

Examples:

- [4, 5, 6, 7, 0, 1, 2, 3] → smallest 0 at index **4** → rotated 4 times
- [3, 4, 5, 1, 2] → smallest 1 at index **3** → rotated 3 times

(Assumption: distinct values — agar duplicates ho to algo thoda modify karna padega.)

Rotation cut point pe order **break** hota hai. Smallest element wahi hota hai jahan break hua. So find index of minimum.

Approach A — Brute force

Scan full array, track minimum value and its index.

Return index.

```
int findRotationsBrute(const vector<int>& a) {  
    int n = a.size();  
    int minIdx = 0;  
    for (int i = 1; i < n; ++i)  
        if (a[i] < a[minIdx]) minIdx = i;  
    return minIdx;  
}
```

Complexities: Time O(N), Space O(1).

Approach B — One-pass break-finding (slightly better idea)

Traverse once and find first i such that $a[i] > a[i+1]$.

Rotation count = $i+1$.

If no such i found \rightarrow array not rotated \rightarrow return 0.

```
int findRotationsOnePass(const vector<int>& a) {  
    int n = a.size();  
    for (int i = 0; i + 1 < n; ++i) {  
        if (a[i] > a[i+1]) return i + 1;  
    }  
    return 0;  
}
```

Complexities: Time $O(N)$, Space $O(1)$.

This is faster in practice than brute (you can stop early at the break).

Approach C — Optimal (Binary Search) — $O(\log N)$

Key idea: Use same trick as *findMin in rotated sorted array*:

- Compare $a[mid]$ with $a[high]$.
 - If $a[mid] > a[high]$ \rightarrow min is in right half $\rightarrow low = mid + 1$.
 - Else \rightarrow min is at mid or left $\rightarrow high = mid$.
- Loop while $low < high$. When ends $low == high \rightarrow$ index of minimum \rightarrow rotation count.

Why this works: In rotated sorted array the smaller half containing the minimum will be identified by $a[mid] \leq a[high]$.

```
#include <bits/stdc++.h>  
using namespace std;  
  
int findRotationCount(const vector<int>& a) {  
    int n = a.size();  
    if (n == 0) return 0; // edge: empty array (define as 0)  
    int low = 0, high = n - 1;
```

```

while (low < high) {
    int mid = low + (high - low) / 2;
    if (a[mid] > a[high]) {
        // min is to the right of mid
        low = mid + 1;
    } else {
        // min is at mid or to the left
        high = mid;
    }
}
// low == high is index of minimum element
return low;
}

int main() {
    vector<int> arr = {4,5,6,7,0,1,2,3};
    cout << findRotationCount(arr) << "\n"; // prints 4
    return 0;
}

```

12. Search Single Element in a sorted array

You're given a **sorted** array where every element appears **exactly twice**, except **one element which appears only once**.

Find that unique element.

Examples:

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

Brute Force – Approach 1 (Neighbor checking)

Because array is sorted, duplicates appear **side-by-side**:

1 1 | 2 2 | 3 3 | 4 | 5 5 | 6 6

The single element is the one

→ **not equal to left neighbor**

→ **not equal to right neighbor.**

✓ Algorithm

1. If array length = 1 → return arr[0].
2. Check first element (index 0):
 - o If $\text{arr}[0] \neq \text{arr}[1]$ → answer is arr[0].
3. Check last element:
 - o If $\text{arr}[n-1] \neq \text{arr}[n-2]$ → answer is arr[n-1].
4. For each middle index i:
 - o If $\text{arr}[i] \neq \text{arr}[i-1]$ AND $\text{arr}[i] \neq \text{arr}[i+1]$ → return arr[i].

```
int singleNonDuplicate(vector<int>& arr) {  
    int n = arr.size();  
    if (n == 1) return arr[0];  
  
    if (arr[0] != arr[1]) return arr[0];  
    if (arr[n-1] != arr[n-2]) return arr[n-1];  
  
    for (int i = 1; i < n-1; i++) {  
        if (arr[i] != arr[i-1] && arr[i] != arr[i+1])  
            return arr[i];  
    }  
  
    return -1; // guaranteed not to happen  
}
```

⌚ Complexity

- **Time:** O(N)
- **Space:** O(1)

Brute Force – Approach 2 (XOR Trick)

Why XOR works?

Properties:

- $a \wedge a = 0$
- $a \wedge 0 = a$

Pairs cancel out, only unique survives:

$$1^1 \wedge 2^2 \wedge 3^3 \wedge 4 = 4$$

```
int singleNonDuplicate(vector<int>& arr) {  
    int ans = 0;  
    for (int x : arr) ans ^= x;  
    return ans;  
}
```

Complexity

- **Time:** O(N)
- **Space:** O(1)

Optimal Approach – Binary Search (O(log N))

Core Insight

Before the unique element:

Pairs are like:

Index: 0 1 2 3 4 5 6 ...
values: a a | b b | c c | ...

- **Every pair starts at even index.**

After the unique element:

Pattern shifts:

```
0 1 2 3 4 5 6 7  
a a b b x c c d d  
↑  
single element
```

After the single element,
pairs now start at **odd** index.

★ Binary Search Strategy

Let's use this pairing rule:

For any index **mid**:

- **If mid is even:**
 - Correct pair → $\text{arr}[\text{mid}] == \text{arr}[\text{mid}+1]$
- **If mid is odd:**
 - Correct pair → $\text{arr}[\text{mid}] == \text{arr}[\text{mid}-1]$

If pairing is correct → unique is on the **right side**

If pairing breaks → unique is on the **left side**

This gives $O(\log N)$ binary elimination.

```
int singleNonDuplicate(vector<int>& arr) {
```

```
    int n = arr.size();
```

```
    if (n == 1) return arr[0];
```

```
    if (arr[0] != arr[1]) return arr[0];
```

```
    if (arr[n-1] != arr[n-2]) return arr[n-1];
```

```
    int low = 1, high = n - 2;
```

```

while (low <= high) {

    int mid = (low + high) / 2;

    // Direct check if mid is unique

    if (arr[mid] != arr[mid+1] && arr[mid] != arr[mid-1])

        return arr[mid];

}

bool isPairCorrect =

(mid % 2 == 0 && arr[mid] == arr[mid+1]) ||

(mid % 2 == 1 && arr[mid] == arr[mid-1]);

if (isPairCorrect) {

    low = mid + 1;    // unique lies right

} else {

    high = mid - 1;    // unique lies left

}

return -1; // won't happen
}

```

⌚ Complexity

Time → O(log N)

Binary search, discarding half each step.

Space → O(1)

No extra memory.

13. Peak element in Array

A **peak element** is one that is **greater than both neighbors**:

$\text{arr}[i-1] < \text{arr}[i] > \text{arr}[i+1]$

At boundaries:

- $i = 0 \rightarrow$ only check right neighbor
- $i = n-1 \rightarrow$ only check left neighbor

If multiple peaks exist \rightarrow return **any one**.

Examples:

[1, 2, 3, 4, 5, 6, 7, 8, 5, 1] \rightarrow peak = 8 at index 7

[1, 2, 1, 3, 5, 6, 4] \rightarrow peaks at indices 1 and 5 \rightarrow return any

2) Brute Force Approach ($O(N)$)

Just check each element and see if it's greater than its neighbors.

```
int findPeakElement(vector<int>& nums) {
    int n = nums.size();

    for (int i = 0; i < n; i++) {
        bool left = (i == 0) || (nums[i] >= nums[i - 1]);
        bool right = (i == n - 1) || (nums[i] >= nums[i + 1]);

        if (left && right)
            return i;
    }
    return -1; // shouldn't happen
}
```

✓ Complexity

- **Time:** $O(N)$
- **Space:** $O(1)$

OPTIMAL APPROACH – Binary Search ($O(\log N)$)

Think of the array like mountains—if you are **going uphill**, a peak **must exist somewhere ahead**.

Let's analyze:

Case A: $\text{nums}[\text{mid}] < \text{nums}[\text{mid} + 1]$

This means:

$\text{mid} \rightarrow \text{mid}+1 \rightarrow$ increasing slope

So the peak lies **to the right**.

We move:

$\text{low} = \text{mid} + 1$

Case B: $\text{nums}[\text{mid}] > \text{nums}[\text{mid} + 1]$

This means:

$\text{mid} \rightarrow \text{mid}+1 \rightarrow$ decreasing slope

So we are **in the right half of a mountain**,
and the peak lies at **mid OR left side**.

We move:

$\text{high} = \text{mid}$

! What guarantees correctness?

Every array with finite ends must have at least 1 peak.

Also:

- Left of peak: strictly increasing region
- Right of peak: strictly decreasing region

Binary search always moves toward a peak.

```

int findPeakElement(vector<int>& nums) {

    int low = 0, high = nums.size() - 1;

    while (low < high) {

        int mid = (low + high) / 2;

        // Descending? → peak is on left side (or mid)

        if (nums[mid] > nums[mid + 1]) {

            high = mid;

        }

        // Ascending? → peak is on right side

        else {

            low = mid + 1;

        }

    }

    return low; // low == high gives a peak index

}

```

Complexity

Operation	Complexity
-----------	------------

Time	O(log N) (binary search halves array each time)
------	--

Space	O(1) (no extra memory)
-------	-------------------------------

14. Finding Sqrt of a number using Binary Search

Given a positive integer n, return:

- **sqrt(n) if n is a perfect square,**
- otherwise return **floor(sqrt(n))**.

Input: 36 → Output: 6

Input: 28 → Output: 5 (since $\text{sqrt}(28) = 5.29\dots$)

Brute Force Approach (O(N))

Try every number from **1 to n**, and track the largest number whose square is $\leq n$.

```
int floorSqrt(int n) {
    int ans = 0;
    for (int i = 1; i <= n; i++) {
        if (1LL * i * i <= n)
            ans = i;
        else
            break;
    }
    return ans;
}
```

⌚ Complexity

- **Time:** O(N)
- **Space:** O(1)

Optimal Approach — Binary Search ($O(\log N)$)

The answer (integer sqrt) must lie in the **range [1...n]**.

Since this range is **sorted**, we can do binary search on it.

At each step:

- Compute `mid`.
- Compare `mid*mid` with `n`.
- If `mid*mid <= n` → `mid` is a *valid* square root → move right to find a bigger possible answer.
- If `mid*mid > n` → `mid` is too large → move left.

We keep track of the last valid `mid` as the answer.

```
int mySqrt(int x) {  
    if (x < 2) return x; // sqrt(0)=0, sqrt(1)=1  
  
    int left = 1, right = x / 2;  
  
    int ans = 0;  
  
    while (left <= right) {  
  
        long long mid = left + (right - left) / 2;  
  
        if (mid * mid <= x) {  
  
            ans = mid; // mid is a possible answer  
  
            left = mid + 1; // try to find a larger one  
  
        } else {  
  
            right = mid - 1; // mid is too large  
  
        }  
    }  
  
    return ans;  
}
```

15. Nth Root of a Number using Binary Search

Given two integers **N** and **M**, find an integer **X** such that:

$$X^n = M$$

If such an integer X does **not** exist → return -1.

Examples

$N = 3, M = 27 \rightarrow \text{Output: } 3$

$N = 4, M = 69 \rightarrow \text{Output: } -1 \quad (\text{no integer 4th root})$

Brute Force Approach — O(M)

Try every number from **1 to M** and check if $i^n == M$.

Stop if $i^n > M$.

```
int nthRoot(int n, int m) {  
    for (int i = 1; i <= m; i++) {  
        long long power = pow(i, n);  
        if (power == m) return i;  
        if (power > m) break;  
    }  
    return -1;}  
}
```

Complexity

- **Time:** $O(M)$
- **Space:** $O(1)$

OPTIMAL APPROACH — Binary Search ($O(\log M)$)

The N -th root of M must lie in the range:

1 to M

Since this range is sorted, we can use **binary search**.

For a given mid:

Compute:

mid^n

We do NOT use $\text{pow}(\text{mid}, n)$ because it:

- is slow
- causes overflow
- is inaccurate for big numbers

Instead, we multiply mid by itself n times **with overflow checking**.

★ Helper function logic

We create a function that compares:

mid^n with M

without overflowing.

```
#include <bits/stdc++.h>

using namespace std;

class Solution {

public:

    // Helper: compute mid^n with early break
    long long power(long long mid, int n, int m) {
```

```

long long ans = 1;

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

    ans *= mid;

    if (ans > m) return ans; // avoid overflow

}

return ans;

}

```

```

// Function to find Nth root of M

int nthRoot(int n, int m) {

    int low = 1, high = m;

    while (low <= high) {

        int mid = (low + high) / 2;

        long long val = power(mid, n, m);

        if (val == m) return mid; // perfect nth root

        else if (val < m)

            low = mid + 1; // try bigger

        else

            high = mid - 1; // try smaller

    }

    return -1; // no integer root

} };

```

```

int main() {
    Solution sol;
    cout << sol.nthRoot(3, 27); // Output: 3
}

```

16. Koko Eating Bananas

Given n piles $\text{piles}[]$ and an integer h (hours). Each hour Koko picks one non-empty pile and eats up to k bananas from it (if fewer than k remain, he finishes that pile and the hour ends).

Find the **minimum integer k** so Koko can finish all bananas within h hours.

If a speed k works (finishes in $\leq h$ hours), then every $k' > k$ also works. So the answer lies in $[1, \max(\text{piles})]$ and we can binary-search that answer space. For a candidate k compute total hours $\sum(\lceil \text{pile}/k \rceil)$.

Important edge case

Because each pile requires at least one hour, total hours $\geq n$. If $h < n$, **no possible k** can finish (each pile requires distinct hour) — return -1 (or handle per problem spec). I include that check in the implementations.

Key micro-optimizations

- Use integer formula $\text{hours} += (\text{pile} + k - 1) / k$ to compute $\lceil \text{pile}/k \rceil$ using integers (no floats).
- Early stop while summing hours if $\text{hours} > h$ — saves work for slow candidates.
- Use `long long` for hour sums to avoid overflow on large inputs.

```
#include <bits/stdc++.h>
```

```

using namespace std;

class Solution {

public:

    // Returns true if speed 'k' allows finishing within 'h' hours

    bool canFinish(const vector<int>& piles, int k, long long h) {

        long long hours = 0;

        for (int p : piles) {

            hours += (p + k - 1) / k;      // integer ceil

            if (hours > h) return false;  // early exit

        }

        return hours <= h;

    }
}

```

```

// Returns minimum k, or -1 if impossible (h < n)

int minEatingSpeed(vector<int>& piles, long long h) {

    int n = piles.size();

    if (h < n) return -1; // impossible: each pile needs at least one hour

    int lo = 1;

    int hi = *max_element(piles.begin(), piles.end());

    int ans = hi;

    while (lo <= hi) {

```

```

int mid = lo + (hi - lo) / 2;

if (canFinish(piles, mid, h)) {

    ans = mid;

    hi = mid - 1; // try smaller speed

} else {

    lo = mid + 1; // need larger speed

}

}

return ans;
}

};

int main() {

    Solution s;

    vector<int> piles1 = {7, 15, 6, 3};

    cout << s.minEatingSpeed(piles1, 8) << '\n'; // 5

    vector<int> piles2 = {25, 12, 8, 14, 19};

    cout << s.minEatingSpeed(piles2, 5) << '\n'; // 25

    // Example impossible case:

    vector<int> piles3 = {3,6,7,11};

    cout << s.minEatingSpeed(piles3, 3) << '\n'; // -1 (since n=4 > h=3)

    return 0;
}

```

17. Minimum days to make M bouquets

You have:

- N roses
- An array $\text{arr}[]$ where $\text{arr}[i]$ = the day the i -th rose will bloom
- You need to make **m** bouquets
- Each bouquet needs **k** adjacent bloomed roses
- You can use a rose only if it is already bloomed (its bloom-day \leq today).

Your goal:

👉 Find the minimum day when you can form at least **m** bouquets.

👉 If it is impossible, return -1.

What does “adjacent bloomed roses” mean?

If $k = 3$, you need **3 consecutive roses that have bloomed**.

For example:

$\text{arr} = [7, 7, 7, 7, 13, 11, 12, 7]$

These are bloom days:

Index: 0 1 2 3 4 5 6 7

Day: 7 7 7 7 13 11 12 7

★ Example

Input:

$N = 8$

$\text{arr} = \{7, 7, 7, 7, 13, 11, 12, 7\}$

$m = 2$ (bouquets)

$k = 3$ (adjacent roses needed)

We must make **2 bouquets**, each having **3 adjacent roses**.

Let's check day = 12:

Index	Bloom day	Is bloomed by day 12?
0	7	✓
1	7	✓
2	7	✓
3	7	✓
4	13	✗
5	11	✓
6	12	✓
7	7	✓

Adjacent bloomed groups:

- [7, 7, 7, 7] → from index 0–3 → **3 adjacent roses available** → 1 bouquet
- [11, 12, 7] → indexes 5–7 → **3 adjacent roses** → 1 bouquet

So, **2 bouquets possible on day 12** → ✓

But on day 11, rose at index 6 (12) isn't bloomed yet → ✗ cannot make two bouquets.

Thus answer = **12**.

Brute Force

1. Check every possible day from `min(arr)` to `max(arr)`.
2. For each day, check if we can form m bouquets.
3. The first day that works → answer.

Time Complexity = $O(N \times (\text{max-min}))$ → too large

Optimal Approach (Binary Search on Days)

The answer (minimum day) lies between:

- **low = minimum bloom day**
- **high = maximum bloom day**

If Koko can make bouquets on day X ,

→ She can definitely make them on any day $> X$.

This monotonic property ⇒ **Binary Search**.

Step 1 — Check impossible case

If $m \times k > N$, immediately return -1.

K adjacent rose, and aise aise M chaiye

Step 2 — Binary search on days

```
low = min(arr)  
high = max(arr)
```

While $low \leq high$:

1. $mid = (low + high) / 2 \rightarrow$ guess a day
2. Check `possible(arr, mid, m, k)`
3. If possible → try smaller day ($high = mid - 1$)
4. If not possible → need more days ($low = mid + 1$)

Final answer → low

Helper Function: **possible(day):**

If $\text{arr}[i] \leq \text{day} \rightarrow \text{rose bloomed}$

Else \rightarrow reset count (not adjacent anymore)

Each time we find k consecutive bloomed roses, we increment bouquet count.

If bouquets $\geq m$, return true.

```
#include <bits/stdc++.h>

using namespace std;

class Solution {

public:

    // Check if it is possible to make m bouquets on or before 'day'

    bool possible(vector<int>& arr, int day, int m, int k) {

        int count = 0;

        int bouquets = 0;

        for (int bloom : arr) {

            if (bloom <= day) {

                count++;

                if (count == k) {
```

```

        bouquets++;

        count = 0;

    }

} else {

    count = 0;

}

return bouquets >= m;

}

// Main function

int minDays(vector<int>& arr, int m, int k) {

    long long needed = 1LL * m * k;

    if (needed > arr.size()) return -1; // Not enough roses

    int low = *min_element(arr.begin(), arr.end());

    int high = *max_element(arr.begin(), arr.end());

    int ans = -1;

    while (low <= high) {

        int mid = (low + high) / 2;

        if (possible(arr, mid, m, k)) {

            ans = mid;

```

```

        high = mid - 1;      // try earlier day
    } else {
        low = mid + 1;      // need more days
    }
}

return ans;
}
};


```

Time Complexity:

$O(N \times \log(\max(\text{arr}) - \min(\text{arr})))$

Because:

- checking possible() takes $O(N)$
- binary search takes $\log(\text{range of days})$

Space Complexity:

$O(1)$

18. Find the Smallest Divisor Given a Threshold

You are given:

- An integer array arr
- A threshold value limit

You must choose a **positive divisor d** such that:

upon dividing all the elements of the given array by it, the sum of the division's result is less than or equal to the given threshold value.

Your objective:

👉 Find the smallest such divisor d.

If divisor = 1, the sum becomes extremely large.

If divisor increases, the sum decreases.

This monotonic behavior \Rightarrow **Binary Search on the divisor.**

✓ Example 1 (Explained)

Input:

arr = {1, 2, 3, 4, 5}, limit = 8

Check divisor = 1

ceil values =

1, 2, 3, 4, 5 \rightarrow sum = 15, too big ✗

Check divisor = 2

ceil(1/2)=1

ceil(2/2)=1

ceil(3/2)=2

$\text{ceil}(4/2)=2$
 $\text{ceil}(5/2)=3$
Sum = **9**, still bigger **✗**

Check divisor = 3

$\text{ceil}(1/3)=1$
 $\text{ceil}(2/3)=1$
 $\text{ceil}(3/3)=1$
 $\text{ceil}(4/3)=2$
 $\text{ceil}(5/3)=2$
Sum = **7**, OKAY **✓**

But is there smaller divisor?

$d = 2 \rightarrow 9 > 8$ **✗**
So **d = 3** is the smallest.

✗ Brute Force Idea (very slow)

1. Try every divisor d from 1 to $\text{max}(\text{arr})$.
2. Compute the sum of $\text{ceil}(\text{arr}[i]/d)$.
3. Return smallest d satisfying condition.

Time Complexity:

$O(\text{max}(\text{arr}) * N) \rightarrow$ too slow for large values

✓ Optimal Solution: Binary Search on Answer

As divisor increases \rightarrow sum decreases
This is a **monotonic function**, perfect for binary search.
low = 1
high = $\text{max}(\text{arr})$

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
    // Helper to compute sum of ceilings
    int computeSum(vector<int>& arr, int d) {
```

```

int sum = 0;
for (int x : arr) {
    sum += (x + d - 1) / d; // integer ceil
}
return sum;
}

int smallestDivisor(vector<int>& arr, int limit) {
    int n = arr.size();

    // If elements > limit, sum of ceil(arr[i]/d) can never be <= limit for d>=1
    if (n > limit) return -1;

    int low = 1;
    int high = *max_element(arr.begin(), arr.end());
    int ans = high;

    while (low <= high) {
        int mid = low + (high - low) / 2;

        if (computeSum(arr, mid) <= limit) {
            ans = mid; // possible divisor
            high = mid - 1; // try smaller
        }
        else {
            low = mid + 1; // need larger divisor
        }
    }

    return ans;
}
};

```

19. Capacity to Ship Packages within D Days

You own a shipping company.

- You have a list of package weights (must be shipped **in order**).
- You have **D days** to ship them.
- Each day you load packages **in order**, but the ship **cannot exceed capacity**.
- If capacity is too small, it will take many days.
- If capacity is huge, everything ships in 1 day.

👉 Your task: **Find the minimum ship capacity so that all packages are shipped in $\leq D$ days.**

★ Example 1

weights = [5,4,5,2,3,4,5,6]

d = 5

We want the **minimum** capacity that allows shipping in **5 days**.

Let's test capacity = **9** (the answer):

Day-wise loading:

- **Day 1:** $5+4 = 9$ (stop)
- **Day 2:** $5+2+3 = 10 > 9 \rightarrow$ only 5,2
- **Day 3:** $3+4 = 7$
- **Day 4:** 5
- **Day 5:** 6

Total = 5 days \Rightarrow **works**

Now check capacity = 8:

- Day 1: $5+4 = 9 > 8 \Rightarrow$ only 5
- Day 2: $4+5 = 9 > 8$
... and you will use more than 5 days.

So 8 doesn't work, 9 is the smallest.

Brute Force Approach:

Idea:

Try every possible capacity from
 $\max(\text{weights}) \rightarrow \sum(\text{weights})$

For each capacity:

- Simulate shipping
- Count number of days taken
- First capacity that fits in D days is the answer

Correct but very slow.

Brute Force: Time Complexity

$O(N * (\sum(\text{weights}) - \max(\text{weights})))$

Too slow ($\sum(\text{weights})$ can be up to 10^9).

Optimal Approach (Binary Search)

Key Insight:

The answer lies between:

- **min capacity = max(weights)**
(because we cannot split a package)

- **max capacity = sum(weights)**
(ship everything in 1 day)

As capacity increases → days required decreases
This is **monotonic**, so we use **Binary Search on capacity**.

low = max(weights)

high = sum(weights)

```
#include <bits/stdc++.h>

using namespace std;

class Solution {

public:

    // How many days needed if ship capacity = cap?

    int daysNeeded(vector<int>& weights, int cap) {

        int days = 1;

        int load = 0;

        for (int w : weights) {

            if (load + w > cap) {

                days++;

                load = w;

            } else {

                load += w;

            }

        }

    }

}
```

```

    }

    return days;
}

int shipWithinDays(vector<int>& weights, int d) {

    int low = *max_element(weights.begin(), weights.end());
    int high = accumulate(weights.begin(), weights.end(), 0);

    while (low < high) {

        int mid = low + (high - low) / 2;

        if (daysNeeded(weights, mid) <= d)
            high = mid;      // try smaller
        else
            low = mid + 1; // need larger capacity
    }

    return low;
};

}

```

20. Kth Missing Positive Number

You are given a strictly increasing array vec of positive integers and a positive integer k. Find the **k-th positive integer that is missing** from vec.
(Example: if $\text{vec} = [4, 7, 9, 10]$, the missing positives are $1, 2, 3, 5, 6, 8, 11, \dots$)

For index i (0-based), the number of missing positive integers **before** $\text{vec}[i]$ is:

```
missing_before_i = vec[i] - (i + 1)
```

Because if there were no missing numbers the value at index i would be $i+1$.
Use this to locate where the k-th missing number lies.

Approach A — Brute force (simple)

Walk the array and for each element $a = \text{vec}[i]$:

- while $k \geq a - \text{current}$ (where current is the next candidate missing), skip forward; simpler implementation: increment k for each $\text{vec}[i] \leq k$.
A clearer and compact brute implementation is:

```
// Brute (O(n))
int missingK_brute(const vector<int>& vec, int k) {
    for (int x : vec) {
        if (x <= k) k++;
        else break;
    }
    return k;
}
```

Why it works: If $x \leq k$ then x "uses up" a missing number $\leq \text{current } k$, so the k-th missing shifts up by 1.

Complexity: $O(n)$, $O(1)$ space.

Approach B — Optimal (binary search) — O(log n)

Use the formula `missing_before_i = vec[i] - (i+1)`.

If `missing_before_last < k` then the answer is beyond the last element:

```
answer = vec.back() + (k - missing_before_last)
```

Otherwise binary search for the largest index `high` such that `missing_before_high < k`.

Then the `k`-th missing is:

```
answer = (high + 1) + k  
// equivalently: k + high + 1  
// or vec[high] + (k - missing_before_high)
```

Reason

At `high` we have fewer than `k` missing numbers. The `k`-th missing lies between `vec[high]` and `vec[high+1]` (or after the array if `high` is last index). The formula `k + high + 1` yields the exact missing integer.

```
#include <bits/stdc++.h>  
using namespace std;  
  
int findKthMissing(const vector<int>& vec, int k) {  
    int n = vec.size();  
    int low = 0, high = n - 1;  
  
    // if total missing until last element is < k, it's after the array  
    int missingLast = vec.back() - n; // vec[n-1] - (n)  
    if (missingLast < k) {  
        return vec.back() + (k - missingLast);  
    }  
  
    // binary search for largest index with missing_before < k  
    while (low <= high) {
```

```

int mid = low + (high - low) / 2;
int missingBeforeMid = vec[mid] - (mid + 1);
if (missingBeforeMid < k) {
    low = mid + 1;
} else {
    high = mid - 1;
}
}

// high is the largest index with missing_before_high < k
// answer = k + high + 1
return k + high + 1;
}

int main() {
    vector<int> vec = {4,7,9,10};
    cout << findKthMissing(vec, 1) << "\n"; // 1
    cout << findKthMissing(vec, 4) << "\n"; // 5
    return 0;
}

```

Complexity

- **Binary search approach:** O(log n) time, O(1) space.
- **Brute approach (scan):** O(n) time, O(1) space.

21. Aggressive Cows

Array me stalls ke positions diye hote hain aur k cows ko aise place karna hota hai ki kisi bhi do cows ke beech ka **minimum distance maximum** ho.

Core Idea

Direct maximum distance nikalna mushkil hai, lekin hum ye check kar sakte hain ki **kya minimum distance d rakh kar saari cows place ho sakti hain ya nahi.**

Agar:

- distance d possible hai → usse chhota bhi possible hogा
- distance d possible nahi hai → usse bada bhi possible nahi hogा

Istiyé answer space **monotonic** hai → Binary Search lagegi.

Search Space

- minimum distance = 1
- maximum distance = $\max(\text{stalls}) - \min(\text{stalls})$

```
low = 1  
high = stalls[last] - stalls[first]
```

canPlace(d) logic

- First cow ko first stall me rakh do

Har next stall par check karo:

- ```
stalls[i] - lastPlaced >= d
```
- - Agar haan → cow place karo
  - Jaise hi k cows place ho jaaye → true
  - End tak nahi ho paayi → false

Greedy approach hai.

---

## Binary Search Flow

```
while(low <= high):
 mid = (low + high) / 2

 if canPlace(mid):
 answer = mid
 low = mid + 1 // distance aur badhane ki koshish
 else:
 high = mid - 1 // distance kam karo
```

Loop ke baad answer hi final maximum minimum distance hoga.

---

## C++ Code

```
bool canPlace(vector<int>& stalls, int cows, int d) {
 int count = 1;
 int lastPos = stalls[0];

 for (int i = 1; i < stalls.size(); i++) {
 if (stalls[i] - lastPos >= d) {
 count++;
 lastPos = stalls[i];
 }
 if (count >= cows) return true;
 }
 return false;
}

int aggressiveCows(vector<int>& stalls, int cows) {
 sort(stalls.begin(), stalls.end());

 int low = 1;
 int high = stalls.back() - stalls.front();
 int ans = 0;
```

```

while (low <= high) {
 int mid = low + (high - low) / 2;

 if (canPlace(stalls, cows, mid)) {
 ans = mid;
 low = mid + 1;
 } else {
 high = mid - 1;
 }
}
return ans;
}

```

---

## Complexity

- Sorting:  $O(N \log N)$
- Binary Search  $\times$  check:  $O(N \log(\maxDist))$
- Space:  $O(1)$

# 22. Allocate Minimum Number of Pages

You are given:

- An integer array  $\text{arr}$  where  $\text{arr}[i]$  denotes the number of pages in the  $i$ -th book
- An integer  $m$  representing the number of students

You must allocate **all books** to the students such that:

1. Each student gets **at least one book**
2. Each book is allocated to **exactly one student**
3. Books allocated to a student must be **contiguous**
4. The **maximum number of pages assigned to any student is minimized**

If it is not possible to allocate books under these conditions, return -1.

---

### **Example 1**

Input:

arr = [12, 34, 67, 90]

m = 2

Output:

113

Explanation:

Possible allocation:

Student 1 → [12, 34, 67] → 113 pages

Student 2 → [90] → 90 pages

Maximum pages = 113 (minimum possible)

---

## Example 2

Input:

```
arr = [25, 46, 28, 49, 24]
```

```
m = 4
```

Output:

71

Explanation:

Allocation:

```
[25, 46] | [28] | [49] | [24]
```

Maximum pages = 71

---

## Key Observation

We cannot directly find the optimal allocation, but we can **check feasibility**.

Question to ask:

If each student is allowed to read **at most X pages**, can we allocate all books using  **$\leq m$  students?**

- If more than  $m$  students are required  $\rightarrow X$  is too small
- If  $m$  or fewer students are sufficient  $\rightarrow X$  is valid

This feasibility condition is monotonic  $\rightarrow$  **Binary Search applies**.

---

## Search Space

- Minimum pages = `max(arr[ ])`  
(a student must read at least one full book)
- Maximum pages = `sum(arr[ ])`  
(one student reads all books)

```
low = max(arr)
high = sum(arr)
```

---

## countStudents(pages)

Greedy allocation:

- Start with one student
- Add books to the current student until page limit exceeds
- Then assign next book to a new student

```
students = 1
currentPages = 0

for each book:
 if currentPages + book <= pages:
 currentPages += book
 else:
 students++
 currentPages = book
```

Return the number of students needed.

---

## Binary Search Logic

```
if m > n: return -1

while(low <= high):
 mid = (low + high) / 2
 students = countStudents(mid)

 if students > m:
 low = mid + 1 // pages too small
 else:
 high = mid - 1 // try smaller maximum
```

After the loop, low gives the **minimum possible maximum pages**.

---

## C++ Code

```
int countStudents(vector<int> &arr, int pages) {
 int students = 1;
 long long currentPages = 0;

 for (int i = 0; i < arr.size(); i++) {
 if (currentPages + arr[i] <= pages) {
 currentPages += arr[i];
 } else {
 students++;
 currentPages = arr[i];
 }
 }
 return students;
}

int findPages(vector<int>& arr, int n, int m) {
 if (m > n) return -1;

 int low = *max_element(arr.begin(), arr.end());
 int high = accumulate(arr.begin(), arr.end(), 0);
```

```

while (low <= high) {
 int mid = low + (high - low) / 2;
 int students = countStudents(arr, mid);

 if (students > m) {
 low = mid + 1;
 } else {
 high = mid - 1;
 }
}
return low;
}

```

---

## Example Insight

`arr = [12, 34, 67, 90], m = 2`

Minimum max pages = 113

Allocation: [12, 34, 67] | [90]

---

## Complexity

- Feasibility check:  $O(N)$
- Binary Search steps:  $O(\log(\text{sum} - \text{max}))$

**Total Time:**  $O(N \log(\text{sum}(arr) - \text{max}(arr)))$

**Space:**  $O(1)$

# 23. Split Array – Largest Sum

## Problem Statement

You are given an integer array A of size N and an integer K.

Split the array into **exactly K non-empty subarrays** such that:

- Each subarray is **contiguous**
- Every element belongs to exactly one subarray
- The **largest sum among all subarrays is minimized**

Return the minimized largest subarray sum.

---

## Example 1

Input:

A = [1, 2, 3, 4, 5]

K = 3

Output:

6

Explanation:

Split as:

[1, 2, 3] | [4] | [5]

Subarray sums = 6, 4, 5

Largest sum = 6

---

## Example 2

Input:

A = [3, 5, 1]

K = 3

Output:

5

Explanation:

Split as:

[3] | [5] | [1]

Largest sum = 5

---

## Approach (Binary Search on Answer)

### Key Idea

Instead of deciding the split directly, decide the **maximum allowed subarray sum** and check if the array can be split into **at most K subarrays**.

- If more than K subarrays are required → allowed sum is too small
- If K or fewer subarrays are enough → allowed sum is valid

The answer space is monotonic, so Binary Search applies.

---

### Search Space

- low =  $\max(A)$
  - high =  $\sum(A)$
- 

### Feasibility Function

Greedy splitting:

- Keep adding elements to the current subarray
- If adding an element exceeds the allowed sum, start a new subarray

```
partitions = 1
currentSum = 0
```

---

## Binary Search Logic

```
while(low <= high):
 mid = (low + high) / 2
 partitions = countPartitions(mid)

 if partitions > K:
 low = mid + 1
 else:
 high = mid - 1
```

Final answer is low.

---

## C++ Code

```
#include <bits/stdc++.h>
using namespace std;

int countPartitions(vector<int>& a, int maxSum) {
 int partitions = 1;
 long long currentSum = 0;

 for (int i = 0; i < a.size(); i++) {
 if (currentSum + a[i] <= maxSum) {
 currentSum += a[i];
 } else {
 partitions++;
 currentSum = a[i];
 }
 }
}
```

```

 }
 return partitions;
}

int splitArrayLargestSum(vector<int>& a, int k) {
 int low = *max_element(a.begin(), a.end());
 int high = accumulate(a.begin(), a.end(), 0);

 while (low <= high) {
 int mid = low + (high - low) / 2;
 int partitions = countPartitions(a, mid);

 if (partitions > k) {
 low = mid + 1;
 } else {
 high = mid - 1;
 }
 }
 return low;
}

int main() {
 vector<int> a = {1, 2, 3, 4, 5};
 int k = 3;
 cout << splitArrayLargestSum(a, k) << endl;
 return 0;
}

```

---

## Complexity

- Time:  $O(N \log(\text{sum} - \text{max}))$
- Space:  $O(1)$

# 24. Painter's Partition Problem

## Problem Statement

You are given an array boards of length N, where boards[i] represents the length of the i-th board.

There are K painters available. Each painter paints at a speed of **1 unit length per unit time**.

Constraints:

- Each painter can paint **only contiguous boards**
- Each board is painted by **exactly one painter**
- All boards must be painted

Return the **minimum time** required to paint all boards.

---

## Example 1

Input:

boards = [5, 5, 5, 5]

K = 2

Output:

10

Explanation:

Split as:

[5, 5] | [5, 5]

Each painter paints 10 units → time = 10

---

## Example 2

Input:

boards = [10, 20, 30, 40]

K = 2

Output:

60

Explanation:

Split as:

[10, 20, 30] | [40]

Painter times = 60, 40

Maximum = 60 (minimum possible)

---

## Approach

### Key Idea

Instead of deciding which painter paints which boards, decide the **maximum time** a painter is allowed to work and check if the job can be completed using at most K painters.

Question to check:

If each painter can work for **at most T time**, can all boards be painted using  $\leq K$  **painters**?

- If more than K painters are needed → T is too small
- If K or fewer painters are enough → T is valid

This condition is monotonic, so Binary Search applies.

---

### Search Space

- Minimum time =  $\max(\text{boards})$   
(a painter must paint at least one full board)

- Maximum time =  $\text{sum}(\text{boards})$   
(one painter paints all boards)

```
low = max(boards)
high = sum(boards)
```

---

## Feasibility Function: countPainters(time)

Greedy allocation:

- Assign boards in order to the current painter
- If adding a board exceeds time, assign it to a new painter

```
painters = 1
currentWork = 0
```

---

## Binary Search Logic

```
while(low <= high):
 mid = (low + high) / 2
 painters = countPainters(mid)

 if painters > K:
 low = mid + 1 // time too small
 else:
 high = mid - 1 // try smaller time
```

Final answer is low.

---

## C++ Code

```
#include <bits/stdc++.h>
using namespace std;
```

```

int countPainters(vector<int>& boards, int time) {
 int painters = 1;
 long long currentWork = 0;

 for (int i = 0; i < boards.size(); i++) {
 if (currentWork + boards[i] <= time) {
 currentWork += boards[i];
 } else {
 painters++;
 currentWork = boards[i];
 }
 }
 return painters;
}

int painterPartition(vector<int>& boards, int k) {
 int low = *max_element(boards.begin(), boards.end());
 int high = accumulate(boards.begin(), boards.end(), 0);

 while (low <= high) {
 int mid = low + (high - low) / 2;
 int painters = countPainters(boards, mid);

 if (painters > k) {
 low = mid + 1;
 } else {
 high = mid - 1;
 }
 }
 return low;
}

int main() {
 vector<int> boards = {10, 20, 30, 40};
 int k = 2;
 cout << painterPartition(boards, k) << endl;
 return 0;
}

```

}

---

## Complexity

- Feasibility check:  $O(N)$
- Binary Search:  $O(\log(\text{sum} - \text{max}))$

**Total Time:**  $O(N \log(\text{sum} - \text{max}))$

**Space:**  $O(1)$

# 25. Minimise Maximum Distance between Gas Stations

## Problem Statement

You are given a **sorted** array  $\text{arr}$  of size  $n$ , where  $\text{arr}[i]$  represents the position of the  $i$ -th gas station on the X-axis.

You are also given an integer  $k$ , representing the number of **new gas stations** to be added.

You can place new gas stations at **any non-negative position**, including non-integer positions.

After adding  $k$  gas stations, let  $\text{dist}$  be the **maximum distance between any two adjacent gas stations**.

Your task is to **minimize this maximum distance** and return its value.

---

## Example 1

Input:

$\text{arr} = [1, 2, 3, 4, 5]$

$k = 4$

Output:

0.5

Explanation:

One possible placement:

[1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5]

Maximum adjacent distance = 0.5

---

## Example 2

Input:

arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

k = 1

Output:

1

Explanation:

Best placement does not reduce the maximum distance below 1.

---

## Approach (Binary Search on Answer)

### Key Idea

Instead of deciding exact positions for new gas stations, we decide the **maximum allowed distance D** between adjacent stations and check feasibility.

Question to check:

If the maximum allowed distance is D, how many gas stations are required to ensure no segment exceeds D?

- If **required stations > k** → D is too small

- If required stations  $\leq k \rightarrow D$  is valid

This feasibility condition is monotonic  $\rightarrow$  Binary Search applies.

---

## Search Space

- low = 0
- high = maximum distance between consecutive existing stations

high = max(arr[i+1] - arr[i])

---

## Feasibility Function

For each gap  $gap = arr[i] - arr[i-1]$ :

Number of stations needed so that each segment  $\leq D$ :

stations = floor(gap / D)

If gap is exactly divisible by D, subtract 1 to avoid placing an extra station.

Sum this over all gaps.

---

## Binary Search Logic

Binary search is done on real numbers, so we stop when precision is small enough.

```
while(high - low > 1e-6):
 mid = (low + high) / 2

 if requiredStations(mid) > k:
 low = mid
 else:
```

```
high = mid
```

Final answer can be high.

---

## C++ Code (Optimal Solution)

```
#include <bits/stdc++.h>
using namespace std;

int numberOfGasStationsRequired(long double dist, vector<int>& arr) {
 int cnt = 0;
 for (int i = 1; i < arr.size(); i++) {
 long double gap = arr[i] - arr[i - 1];
 int stations = gap / dist;
 if (gap == dist * stations) stations--;
 cnt += stations;
 }
 return cnt;
}

long double minimiseMaxDistance(vector<int>& arr, int k) {
 long double low = 0, high = 0;

 for (int i = 0; i < arr.size() - 1; i++) {
 high = max(high, (long double)(arr[i + 1] - arr[i]));
 }

 long double precision = 1e-6;

 while (high - low > precision) {
 long double mid = (low + high) / 2.0;
 if (numberOfGasStationsRequired(mid, arr) > k)
 low = mid;
 else
 high = mid;
 }
 return high;
}
```

```
}

int main() {
 vector<int> arr = {1, 2, 3, 4, 5};
 int k = 4;
 cout << minimiseMaxDistance(arr, k) << endl;
 return 0;
}
```

---

## Complexity

- Feasibility check:  $O(n)$
- Binary Search iterations:  $O(\log(\text{range} / \text{precision}))$

**Total Time:**  $O(n \log(\text{range}))$

**Space:**  $O(1)$

# 26. Median of Two Sorted Arrays of Different Sizes

## Problem Statement

You are given two **sorted** arrays  $\text{arr1}$  and  $\text{arr2}$  of sizes  $n1$  and  $n2$ .

Return the **median** of the combined sorted array formed after merging both arrays.

- If the total number of elements is **odd**, the median is the middle element.

- If the total number of elements is **even**, the median is the **average of the two middle elements**.
- 

## Example 1

Input:

```
arr1 = [2, 4, 6]
arr2 = [1, 3, 5]
```

Output:

3.5

Explanation:

Merged array = [1, 2, 3, 4, 5, 6]

Median =  $(3 + 4) / 2 = 3.5$

---

## Example 2

Input:

```
arr1 = [2, 4, 6]
arr2 = [1, 3]
```

Output:

3

Explanation:

Merged array = [1, 2, 3, 4, 6]

Median = 3

---

## Optimal Approach (Binary Search on Partition)

### Key Idea

Instead of merging the arrays, we **partition** them such that:

- Left half contains exactly half (or half + 1) elements
- All elements in left half  $\leq$  all elements in right half

The median lies at the boundary of these partitions.

Binary search is applied on the **smaller array** to minimize time complexity.

---

## Partition Conditions

Let:

- $\text{cut1}$  = number of elements taken from  $\text{arr1}$
- $\text{cut2}$  = number of elements taken from  $\text{arr2}$

$$\text{cut1} + \text{cut2} = (\text{n1} + \text{n2} + 1) / 2$$

Let:

- $\text{l1}$  = last element on left of  $\text{arr1}$
- $\text{r1}$  = first element on right of  $\text{arr1}$
- $\text{l2}$  = last element on left of  $\text{arr2}$
- $\text{r2}$  = first element on right of  $\text{arr2}$

Correct partition condition:

$$\text{l1} \leq \text{r2} \text{ AND } \text{l2} \leq \text{r1}$$

---

## Median Calculation

If  $(\text{n1} + \text{n2})$  is even:

$$\text{median} = (\max(\text{l1}, \text{l2}) + \min(\text{r1}, \text{r2})) / 2$$



If odd:

```
median = max(l1, l2)
```



---

## C++ Code (Optimal)

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 double findMedianSortedArrays(vector<int>& a, vector<int>& b) {
 // Ensure binary search on smaller array
 if (a.size() > b.size())
 return findMedianSortedArrays(b, a);

 int n1 = a.size(), n2 = b.size();
 int low = 0, high = n1;

 while (low <= high) {
 int cut1 = (low + high) / 2;
 int cut2 = (n1 + n2 + 1) / 2 - cut1;

 int l1 = (cut1 == 0) ? INT_MIN : a[cut1 - 1];
 int l2 = (cut2 == 0) ? INT_MIN : b[cut2 - 1];
 int r1 = (cut1 == n1) ? INT_MAX : a[cut1];
 int r2 = (cut2 == n2) ? INT_MAX : b[cut2];

 if (l1 <= r2 && l2 <= r1) {
 if ((n1 + n2) % 2 == 0)
 return (max(l1, l2) + min(r1, r2)) / 2.0;
 else
 return max(l1, l2);
 }
 }
 }
}
```

```

 else if (l1 > r2) {
 high = cut1 - 1;
 }
 else {
 low = cut1 + 1;
 }
 }
 return 0.0;
}
};

int main() {
 vector<int> arr1 = {2, 4, 6};
 vector<int> arr2 = {1, 3};
 Solution sol;
 cout << sol.findMedianSortedArrays(arr1, arr2) << endl;
 return 0;
}

```

---

## Complexity

- **Time:**  $O(\log(\min(n_1, n_2)))$
- **Space:**  $O(1)$

# 27. K-th Element of Two Sorted Arrays

## Problem Statement

You are given two **sorted** arrays  $a$  and  $b$  of sizes  $m$  and  $n$ .

Find the **k-th element (1-based index)** in the final sorted array formed by merging both arrays.

---

## Example 1

Input:

a = [2, 3, 6, 7, 9]

b = [1, 4, 8, 10]

k = 5

Output:

6

Explanation:

Merged array = [1, 2, 3, 4, 6, 7, 8, 9, 10]

5th element = 6

---

## Example 2

Input:

a = [100, 112, 256, 349, 770]

b = [72, 86, 113, 119, 265, 445, 892]

k = 7

Output:

256

---

## Approach (Binary Search on Partition)

### Key Idea

This problem is an extension of **Median of Two Sorted Arrays**.

We partition both arrays such that:

- Total elements on the **left side** = k
- All elements on the left side  $\leq$  all elements on the right side

Binary search is applied on the **smaller array** to optimize time.

---

## Setup

Let:

- a be the smaller array (swap if needed)
- left = k

Binary search range:

```
low = max(0, k - n)
high = min(k, m)
```

---

## Partition Logic

Choose:

```
mid1 = elements taken from a
mid2 = elements taken from b = k - mid1
```

Define boundary values:

```
l1 = (mid1 == 0) ? -∞ : a[mid1 - 1]
l2 = (mid2 == 0) ? -∞ : b[mid2 - 1]
r1 = (mid1 == m) ? +∞ : a[mid1]
r2 = (mid2 == n) ? +∞ : b[mid2]
```

Correct partition condition:

```
l1 ≤ r2 AND l2 ≤ r1
```

When satisfied:

```
Answer = max(l1, l2)
```

---

## Binary Search Logic

```
while(low <= high):
 mid1 = (low + high) / 2
 mid2 = k - mid1

 if l1 <= r2 and l2 <= r1:
 return max(l1, l2)
 else if l1 > r2:
 high = mid1 - 1
 else:
 low = mid1 + 1
```

---

## C++ Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int kthElement(vector<int>& a, vector<int>& b, int k) {
 int m = a.size(), n = b.size();

 // Always binary search on smaller array
 if (m > n) return kthElement(b, a, k);

 int low = max(0, k - n);
 int high = min(k, m);

 while (low <= high) {
 int mid1 = (low + high) / 2;
 int mid2 = k - mid1;

 int l1 = (mid1 == 0) ? INT_MIN : a[mid1 - 1];
 int l2 = (mid2 == 0) ? INT_MIN : b[mid2 - 1];
```

```

 int r1 = (mid1 == m) ? INT_MAX : a[mid1];
 int r2 = (mid2 == n) ? INT_MAX : b[mid2];

 if (l1 <= r2 && l2 <= r1) {
 return max(l1, l2);
 }
 else if (l1 > r2) {
 high = mid1 - 1;
 }
 else {
 low = mid1 + 1;
 }
 }
 return -1;
}

int main() {
 vector<int> a = {2, 3, 6, 7, 9};
 vector<int> b = {1, 4, 8, 10};
 int k = 5;

 Solution sol;
 cout << sol.kthElement(a, b, k) << endl;
 return 0;
}

```

---

## Complexity

- **Time:**  $O(\log(\min(m, n)))$
- **Space:**  $O(1)$

# 28. Find the Row with Maximum Number of 1's

## Problem Statement

You are given a non-empty binary matrix `mat` with  $n$  rows and  $m$  columns.

Each row is **sorted in ascending order** (all 0's come before 1's).

Your task is to return the **index (0-based)** of the row that contains the **maximum number of 1's**.

Rules:

- If multiple rows have the same maximum number of 1's, return the **smallest index**
  - If the matrix contains **no 1 at all**, return -1
- 

## Example 1

Input:

```
mat =
1 1 1
0 0 1
0 0 0
```

Output:

```
0
```

Explanation:

Row 0 has 3 ones, which is the maximum.

---

## Example 2

Input:

```
mat =
0 0
```

0 0

Output:

-1

Explanation:

No row contains a 1.

---

## Brute Force Approach

### Idea

Check every row and count the number of 1s manually using nested loops.

---

### Steps

Initialize:

```
maxCount = 0
rowIndex = -1
```

- 1.
  2. For each row:
    - o Traverse all columns
    - o Count number of 1s
  3. If current count > maxCount:
    - o Update maxCount
    - o Update rowIndex
  4. Return rowIndex
-

## C++ Code (Brute Force)

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int rowWithMax1s(vector<vector<int>>& mat, int n, int m) {
 int maxCount = 0;
 int rowIndex = -1;

 for (int i = 0; i < n; i++) {
 int countOnes = 0;
 for (int j = 0; j < m; j++) {
 countOnes += mat[i][j];
 }
 if (countOnes > maxCount) {
 maxCount = countOnes;
 rowIndex = i;
 }
 }
 return rowIndex;
 }
};
```

---

## Complexity (Brute)

- **Time:**  $O(n \times m)$
- **Space:**  $O(1)$

## Optimal Approach (Binary Search)

Each row is **sorted**, so:

- All 0's appear before all 1's

- If we find the **first occurrence of 1** in a row,  
then number of 1's =  $m - \text{index\_of\_first\_1}$

Instead of counting all elements, we use **Binary Search** on each row.

---

## Steps

Initialize:

```
maxCount = 0
rowIndex = -1
```

- 1.
2. For each row:

- Use binary search to find the first index where value is 1

Count number of 1's using:

```
ones = m - firstIndex
```

○

If  $\text{ones} > \text{maxCount}$ , update:

```
maxCount = ones
rowIndex = currentRow
```

- 3.
  4. Return `rowIndex`
- 

## Binary Search Logic (Lower Bound for 1)

Find the smallest index  $i$  such that  $\text{row}[i] == 1$ .

---

## C++ Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int firstOneIndex(vector<int>& row, int m) {
 int low = 0, high = m - 1;
 int ans = m;

 while (low <= high) {
 int mid = (low + high) / 2;
 if (row[mid] == 1) {
 ans = mid;
 high = mid - 1;
 } else {
 low = mid + 1;
 }
 }
 return ans;
 }

 int rowWithMax1s(vector<vector<int>>& mat, int n, int m) {
 int maxCount = 0;
 int rowIndex = -1;

 for (int i = 0; i < n; i++) {
 int firstOne = firstOneIndex(mat[i], m);
 int ones = m - firstOne;

 if (ones > maxCount) {
 maxCount = ones;
 rowIndex = i;
 }
 }
 return rowIndex;
 }
}
```

```

 }
};

int main() {
 vector<vector<int>> mat = {
 {1, 1, 1},
 {0, 0, 1},
 {0, 0, 0}
 };

 Solution obj;
 cout << obj.rowWithMax1s(mat, 3, 3) << endl;
 return 0;
}

```

---

## Complexity

- Binary search per row:  $O(\log m)$
- Total rows:  $n$

**Time Complexity:**  $O(n \log m)$

**Space Complexity:**  $O(1)$

# 29. Search in a Sorted 2D Matrix

## Problem Statement

You are given a 2D matrix `mat` of size  $N \times M$  such that:

- Each row is sorted in **non-decreasing order**

- The first element of each row is greater than the last element of the previous row

You are also given an integer target.

Return true if target exists in the matrix, otherwise return false.

---

### Example 1

Input:

```
mat = [
 [1, 2, 3, 4],
 [5, 6, 7, 8],
 [9, 10, 11, 12]
]
target = 8
```

Output:

```
true
```

---

### Example 2

Input:

```
mat = [
 [1, 2, 4],
 [6, 7, 8],
 [9, 10, 34]
]
target = 78
```

Output:

```
false
```

---

## Brute Force Approach

### Idea

Check every element of the matrix one by one.

---

## Steps

1. Traverse each row
  2. Traverse each column inside that row
  3. If any element equals target, return true
  4. If traversal ends, return false
- 

## C++ Code (Brute Force)

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 bool searchMatrix(vector<vector<int>>& matrix, int target) {
 int n = matrix.size();
 int m = matrix[0].size();

 for (int i = 0; i < n; i++) {
 for (int j = 0; j < m; j++) {
 if (matrix[i][j] == target)
 return true;
 }
 }
 return false;
 }
};
```

---

## Complexity (Brute)

- **Time:**  $O(N \times M)$
  - **Space:**  $O(1)$
- 

## Better Approach (Binary Search Row-wise)

### Idea

Each row is sorted.

First identify the row where the target **can possibly exist**, then apply binary search on that row.

A row  $i$  can contain target if:

```
mat[i][0] <= target <= mat[i][m-1]
```

---

### Steps

1. Traverse rows one by one
  2. For a row satisfying the above condition:
    - Apply binary search on that row
  3. If found, return true
  4. Otherwise, return false
- 

### C++ Code (Better)

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
```

```

bool binarySearch(vector<int>& row, int target) {
 int low = 0, high = row.size() - 1;

 while (low <= high) {
 int mid = (low + high) / 2;
 if (row[mid] == target)
 return true;
 else if (row[mid] < target)
 low = mid + 1;
 else
 high = mid - 1;
 }
 return false;
}

bool searchMatrix(vector<vector<int>>& matrix, int target) {
 int n = matrix.size();
 int m = matrix[0].size();

 for (int i = 0; i < n; i++) {
 if (matrix[i][0] <= target && target <= matrix[i][m - 1])
{
 return binarySearch(matrix[i], target);
 }
 }
 return false;
}

```

---

## Complexity (Better)

- **Time:**  $O(N \times \log M)$
  - **Space:**  $O(1)$
-

# Optimal Approach (Binary Search on Virtual 1D Array)

## Idea

Because:

- Rows are sorted
- First element of a row > last element of previous row

The entire matrix behaves like a **single sorted 1D array**.

Instead of flattening, we **map indices**:

```
row = mid / M
col = mid % M
```

---

## Steps

1. Consider search space from 0 to  $N \times M - 1$
  2. Apply binary search on this imaginary array
  3. Convert mid index to  $(\text{row}, \text{col})$
  4. Compare  $\text{mat}[\text{row}][\text{col}]$  with target
- 

## C++ Code (Optimal)

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 bool searchMatrix(vector<vector<int>>& matrix, int target) {
 int n = matrix.size();
 int m = matrix[0].size();
```

```

int low = 0, high = n * m - 1;

while (low <= high) {
 int mid = (low + high) / 2;
 int row = mid / m;
 int col = mid % m;

 if (matrix[row][col] == target)
 return true;
 else if (matrix[row][col] < target)
 low = mid + 1;
 else
 high = mid - 1;
}
return false;
}

```

---

## Complexity (Optimal)

- **Time:**  $O(\log(N \times M))$
  - **Space:**  $O(1)$
- 

## When to Use Which

- Small matrix → Brute force
- Row-wise sorted only → Row-wise binary search
- Fully sorted matrix (this problem) → Virtual 1D binary search

# 30. Search in a Row and Column-wise Sorted Matrix

## Problem Statement

You are given a 2D matrix mat of size  $N \times M$  such that:

- Each **row** is sorted in non-decreasing order
- Each **column** is sorted in non-decreasing order
- The first element of a row is **not necessarily greater** than the last element of the previous row

You are given an integer target.

Your task is to determine whether target exists in the matrix or not.

---

## Example 1

Matrix:

|    |    |    |    |
|----|----|----|----|
| 1  | 4  | 7  | 11 |
| 2  | 5  | 8  | 12 |
| 3  | 6  | 9  | 16 |
| 10 | 13 | 14 | 17 |

Target: 9

Output: Found at (2, 2)

---

## Example 2

Matrix:

|   |    |    |
|---|----|----|
| 5 | 10 | 15 |
| 6 | 12 | 18 |
| 8 | 16 | 20 |

Target: 7

Output: Not Found

---

## Brute Force Approach

### Idea

Traverse the entire matrix and check each element.

---

### Steps

1. Traverse row by row
  2. Traverse column by column inside each row
  3. If any element equals target, return true
  4. If traversal ends, return false
- 

### C++ Code (Brute Force)

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 bool searchMatrix(vector<vector<int>>& mat, int target) {
 int n = mat.size();
 int m = mat[0].size();

 for (int i = 0; i < n; i++) {
 for (int j = 0; j < m; j++) {
 if (mat[i][j] == target)
 return true;
 }
 }
 }
}
```

```
 }
 return false;
 }
};
```

---

## Complexity (Brute)

- **Time:**  $O(N \times M)$
  - **Space:**  $O(1)$
- 

## Better Approach (Binary Search on Each Row)

### Idea

Each row is sorted, so instead of checking all columns:

- Apply **binary search** on each row
- 

### Steps

1. For each row:
    - Apply binary search to find target
  2. If found in any row, return `true`
  3. Otherwise, return `false`
- 

## C++ Code (Better)

```
#include <bits/stdc++.h>
```

```

using namespace std;

class Solution {
public:
 bool binarySearch(vector<int>& row, int target) {
 int low = 0, high = row.size() - 1;

 while (low <= high) {
 int mid = (low + high) / 2;
 if (row[mid] == target)
 return true;
 else if (row[mid] < target)
 low = mid + 1;
 else
 high = mid - 1;
 }
 return false;
 }

 bool searchMatrix(vector<vector<int>>& mat, int target) {
 int n = mat.size();

 for (int i = 0; i < n; i++) {
 if (binarySearch(mat[i], target))
 return true;
 }
 return false;
 }
};

```

---

## Complexity (Better)

- **Time:**  $O(N \log M)$
  - **Space:**  $O(1)$
-

# Optimal Approach (Staircase Search)

## Idea

Start from the **top-right corner** ( $0, M-1$ ).

Why this works:

- Moving **left** decreases the value
- Moving **down** increases the value

So at every step, one full row or column gets eliminated.

---

## Steps

1. Start at ( $\text{row} = 0, \text{col} = M-1$ )
  2. While  $\text{row} < N$  and  $\text{col} \geq 0$ :
    - If  $\text{mat}[\text{row}][\text{col}] == \text{target}$  → return true
    - If  $\text{mat}[\text{row}][\text{col}] > \text{target}$  → move left ( $\text{col}--$ )
    - If  $\text{mat}[\text{row}][\text{col}] < \text{target}$  → move down ( $\text{row}++$ )
  3. If search ends, return false
- 

## C++ Code (Optimal)

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 bool searchMatrix(vector<vector<int>>& mat, int target) {
 int n = mat.size();
```

```

int m = mat[0].size();

int row = 0, col = m - 1;

while (row < n && col >= 0) {
 if (mat[row][col] == target)
 return true;
 else if (mat[row][col] > target)
 col--;
 else
 row++;
}
return false;
};


```

---

## Complexity (Optimal)

- **Time:**  $O(N + M)$
  - **Space:**  $O(1)$
- 

## When to Use Which

- Small matrix → Brute force
- Only row-wise sorted → Binary search on rows
- Row-wise + column-wise sorted → Staircase search (best)

# 31. Find Peak Element (2D Matrix)

## Problem Statement

You are given a **0-indexed** matrix `mat` of size  $n \times m$  such that **no two adjacent cells are equal**.

A **peak element** is an element that is **strictly greater** than its adjacent neighbors:

- top ( $i-1, j$ )
- bottom ( $i+1, j$ )
- left ( $i, j-1$ )
- right ( $i, j+1$ )

The matrix is assumed to be surrounded by a boundary of -1.

Your task is to find **any one peak element** and return its position  $[i, j]$ .

Multiple peaks may exist. Returning **any valid peak index** is acceptable.

---

## Example 1

Input:

```
mat = [
 [5, 10, 8],
 [4, 25, 7],
 [3, 9, 6]
]
```

Output:

```
[1, 1]
```

Explanation:

25 is greater than its neighbors (10, 7, 4, 9)

---

## Example 2

Input:

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

Output:

```
[2, 2]
```

Explanation:

9 is greater than its neighbors (8, 4)

---

## Optimal Approach (Binary Search on Columns)

### Core Idea

This problem extends the **1D peak element** idea to **2D**.

Instead of binary search on rows, we:

- Apply **binary search on columns**
  - In each middle column, find the **maximum element**
  - Compare it with its **left and right neighbors**
- 

### Why This Works

For a chosen middle column:

- Let (row, mid) be the **maximum element** in that column
- Since it is already the largest in its column:

- Only left and right neighbors can invalidate it as a peak

Now:

- If it is greater than both left and right → **peak found**
- If left neighbor is larger → peak exists in **left half**
- If right neighbor is larger → peak exists in **right half**

This guarantees progress just like binary search.

---

## Steps

1. Set `low = 0, high = m - 1`
2. While `low <= high`:
  - `mid = (low + high) / 2`
  - Find the row index of the **maximum element** in column `mid`
  - Compare this element with its left and right neighbors
3. Decide direction:
  - Peak found → return position
  - Left larger → search left
  - Right larger → search right

---

## Helper Function

Find the row index of the **maximum element in a given column**.

---

## C++ Code (Optimal)

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 // Returns row index of max element in a column
 int maxElement(vector<vector<int>>& mat, int col) {
 int n = mat.size();
 int idx = 0;

 for (int i = 1; i < n; i++) {
 if (mat[i][col] > mat[idx][col])
 idx = i;
 }
 return idx;
 }

 vector<int> findPeakGrid(vector<vector<int>>& mat) {
 int n = mat.size();
 int m = mat[0].size();

 int low = 0, high = m - 1;

 while (low <= high) {
 int mid = (low + high) / 2;
 int row = maxElement(mat, mid);

 int left = (mid - 1 >= 0) ? mat[row][mid - 1] : INT_MIN;
 int right = (mid + 1 < m) ? mat[row][mid + 1] : INT_MIN;

 if (mat[row][mid] > left && mat[row][mid] > right) {
 return {row, mid};
 }
 else if (left > mat[row][mid]) {
 high = mid - 1;
 }
 else {

```

```

 low = mid + 1;
 }
}
return {-1, -1};
};

int main() {
 vector<vector<int>> mat = {
 {4, 2, 5, 1, 4, 5},
 {2, 9, 3, 2, 3, 2},
 {1, 7, 6, 0, 1, 3},
 {3, 6, 2, 3, 7, 2}
 };
 Solution sol;
 vector<int> ans = sol.findPeakGrid(mat);
 cout << ans[0] << " " << ans[1] << endl;
}

```

---

## Complexity Analysis

- Finding max in a column:  $O(N)$
- Binary search on columns:  $O(\log M)$

**Time Complexity:**  $O(N \log M)$

**Space Complexity:**  $O(1)$

# 32. Median of Row-Wise Sorted Matrix

## Problem Statement

You are given a matrix of size  $M \times N$  where **each row is sorted in non-decreasing order**.  $M \times N$  is always **odd**.

You need to find the **median** of the matrix.

Median definition:

- Since total elements are odd, median is the **( $M \times N$ )/2-th element (0-based)** in the sorted order of all elements.
- 

## Example 1

Input:

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

Output:

```
5
```

Explanation:

Sorted order:

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

Median = 5

---

## Example 2

Input:

```
1 3 8
2 3 4
1 2 5
```

Output:

3

Explanation:

Sorted order:

1 1 2 2 3 3 4 5 8

Median = 3

---

## Brute Force Approach

### Idea

Store all elements of the matrix into a single array, sort it, and return the middle element.

---

### Steps

1. Create an empty array
  2. Push all elements of the matrix into it
  3. Sort the array
  4. Return element at index  $(M \times N) / 2$
- 

### C++ Code (Brute Force)

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int findMedian(vector<vector<int>>& matrix) {
 vector<int> arr;
```

```

 for (auto& row : matrix) {
 for (int x : row) {
 arr.push_back(x);
 }
 }

 sort(arr.begin(), arr.end());
 return arr[arr.size() / 2];
 }
};

int main() {
 vector<vector<int>> matrix = {
 {1, 4, 9},
 {2, 5, 6},
 {3, 8, 7}
 };

 Solution obj;
 cout << obj.findMedian(matrix) << endl;
 return 0;
}

```

---

## Complexity (Brute Force)

- **Time:**  $O(M \times N \log(M \times N))$
  - **Space:**  $O(M \times N)$
- 

## Optimal Approach (Binary Search on Value Space)

### Key Observation

- Rows are sorted, but the matrix is **not globally sorted**

- Median depends on **value**, not index
  - We binary search on the **range of values**, not positions
- 

## Core Idea

Instead of sorting everything:

- Guess a value `mid`
  - Count how many elements in the matrix are  $\leq \text{mid}$
  - If count < required  $\rightarrow$  median is larger
  - Else  $\rightarrow$  median is smaller or equal
- 

## Search Space

- `low` = minimum element among first elements of all rows
- `high` = maximum element among last elements of all rows

Required count for median:

$$(m \times n + 1) / 2$$

---

## Counting $\leq \text{mid}$

Since each row is sorted:

- Use `upper_bound` on each row
- This gives count of elements  $\leq \text{mid}$  in that row

---

## Steps

1. Find low and high
  2. While  $\text{low} < \text{high}$ :
    - o  $\text{mid} = (\text{low} + \text{high}) / 2$
    - o Count elements  $\leq \text{mid}$
  3. Adjust range based on count
  4. Final low is the median
- 

## C++ Code (Optimal)

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int findMedian(vector<vector<int>>& matrix) {
 int rows = matrix.size();
 int cols = matrix[0].size();

 int low = matrix[0][0];
 int high = matrix[0][cols - 1];

 for (int i = 1; i < rows; i++) {
 low = min(low, matrix[i][0]);
 high = max(high, matrix[i][cols - 1]);
 }

 int required = (rows * cols + 1) / 2;

 while (low < high) {
```

```

 int mid = (low + high) / 2;
 int count = 0;

 for (int i = 0; i < rows; i++) {
 count += upper_bound(matrix[i].begin(),
 matrix[i].end(),
 mid) - matrix[i].begin();
 }

 if (count < required)
 low = mid + 1;
 else
 high = mid;
 }

 return low;
}
};

int main() {
 vector<vector<int>> matrix = {
 {1, 3, 5},
 {2, 6, 9},
 {3, 6, 9}
 };

 Solution obj;
 cout << obj.findMedian(matrix) << endl;
 return 0;
}

```

---

## Complexity (Optimal)

- **Time:**  $O(M \times \log N \times \log(\text{valueRange}))$
- **Space:**  $O(1)$

---

## Comparison Summary

| Approach    | Time                            | Space   |
|-------------|---------------------------------|---------|
| Brute Force | $O(MN \log(MN))$                | $O(MN)$ |
| Optimal     | $O(M \log N \log \text{range})$ | $O(1)$  |

# **STRINGS**

# 1. Remove Outermost Parentheses

## Problem Statement

A **valid parentheses string** is defined as:

- "" (empty string) is valid
- If A is valid, then "( " + A + " )" is valid
- If A and B are valid, then A + B is valid

A **primitive valid parentheses string** is a non-empty valid string that **cannot be split** into two or more non-empty valid parentheses strings.

Given a valid parentheses string s, remove the **outermost parentheses** from **every primitive** in s and return the resulting string.

---

## Example 1

Input:

s = "((()))"

Output:

"(()())"

Explanation:

Primitive = "((()))"

After removing outermost parentheses → "(()())"

---

## Example 2

Input:

s = "(()((())())())"

Output:

"((())())()

Explanation:

Primitives:

"()" , "((()))" , "((())"

After removing outermost parentheses:

"" + "()( )" + "( )"

Final result: "((())())"

---

## Approach

### Core Idea

We track the **depth (level)** of parentheses while traversing the string.

- The **outermost parentheses** of a primitive are:
  - The '<(' when level == 0
  - The ')' when level == 1

So:

- We **skip** these outermost parentheses
  - We **keep** all others
- 

## Steps

1. Initialize:

- level = 0
- result = ""

2. Traverse the string character by character:

- If '(':
  - If level > 0, add '(' to result
  - Increment level
- If ')':
  - Decrement level
  - If level > 0, add ')' to result

3. Return result

---

## Why This Works

- Each primitive starts when level goes from 0 → 1
  - Its outermost parentheses are exactly at:
    - first '(' (level was 0)
    - last ')' (level becomes 0)
  - By checking level, we automatically ignore only those outer ones
- 

## C++ Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
```

```

string removeOuterParentheses(string s) {
 string result = "";
 int level = 0;

 for (char ch : s) {
 if (ch == '(') {
 if (level > 0)
 result += ch;
 level++;
 }
 else { // ch == ')'
 level--;
 if (level > 0)
 result += ch;
 }
 }
 return result;
}

int main() {
 string s = "((())())";
 Solution sol;
 cout << sol.removeOuterParentheses(s) << endl;
 return 0;
}

```

---

## Complexity Analysis

- **Time Complexity:**  $O(n)$  — single traversal of the string
- **Space Complexity:**  $O(1)$  — excluding the output string

## 2. Reverse Words in a String

### Problem Statement

You are given a string  $s$  consisting of:

- Uppercase and lowercase letters
- Digits
- Spaces (' ')

A **word** is defined as a sequence of non-space characters.

Words are separated by **one or more spaces**.

You need to:

- Reverse the **order of words**
  - Ensure **exactly one space** between words
  - Remove any **leading or trailing spaces**
- 

### Example 1

Input:

"welcome to the jungle"

Output:

"jungle the to welcome"

---

### Example 2

Input:

" amazing coding skills "

Output:

"skills coding amazing"

---

## Brute Force Approach

### Idea

Manually extract all words, store them, reverse the list of words, and then join them with a single space.

This approach handles:

- Multiple spaces
  - Leading spaces
  - Trailing spaces
- 

### Steps

1. Initialize an empty list words
  2. Traverse the string character by character
  3. Build words using consecutive non-space characters
  4. Ignore extra spaces
  5. Push each word into the list
  6. Reverse the list of words
  7. Join them using a single space
- 

### C++ Code (Brute Force)

```
#include <bits/stdc++.h>
```

```

using namespace std;

class Solution {
public:
 string reverseWords(string s) {
 vector<string> words;
 string word = "";

 for (char ch : s) {
 if (ch != ' ') {
 word += ch;
 } else if (!word.empty()) {
 words.push_back(word);
 word = "";
 }
 }

 if (!word.empty()) {
 words.push_back(word);
 }

 reverse(words.begin(), words.end());

 string result = "";
 for (int i = 0; i < words.size(); i++) {
 result += words[i];
 if (i < words.size() - 1)
 result += " ";
 }

 return result;
 }
};

int main() {
 Solution obj;
 string s = " amazing coding skills ";
 cout << obj.reverseWords(s) << endl;
}

```

```
 return 0;
}
```

---

## Complexity (Brute Force)

- **Time Complexity:**  $O(N)$
  - **Space Complexity:**  $O(N)$  (extra list to store words)
- 

## Optimal Approach (Right-to-Left Scan)

### Idea

Instead of storing words and reversing later:

- Scan the string **from right to left**
- Extract each word and append it directly to the result

This avoids:

- Extra list
  - Explicit reverse step
- 

### Steps

1. Start from the end of the string
2. Skip spaces
3. Identify a word by moving left until a space
4. Append the word to result

5. Add a space only when needed
  6. Repeat until the beginning
- 

### C++ Code (Optimal)

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 string reverseWords(string s) {
 string result = "";
 int i = s.size() - 1;

 while (i >= 0) {
 while (i >= 0 && s[i] == ' ') //space skip
 i--;
 if (i < 0) break;

 int end = i;

 while (i >= 0 && s[i] != ' ')
 i--;

 string word = s.substr(i + 1, end - i);

 if (!result.empty())
 result += " ";

 result += word;
 }

 return result;
 }
};
```

```
int main() {
 Solution obj;
 string s = " amazing coding skills ";
 cout << obj.reverseWords(s) << endl;
 return 0;
}
```

---

## Complexity (Optimal)

- **Time Complexity:**  $O(N)$
  - **Space Complexity:**  $O(1)$  (excluding output string)
- 

## Comparison Summary

| Approach | Time | Space |
|----------|------|-------|
|----------|------|-------|

|             |        |        |
|-------------|--------|--------|
| Brute Force | $O(N)$ | $O(N)$ |
|-------------|--------|--------|

|         |        |        |
|---------|--------|--------|
| Optimal | $O(N)$ | $O(1)$ |
|---------|--------|--------|

## 3. Largest Odd Number in a String

### Problem Statement

You are given a string  $s$  representing a **large integer**.

Your task is to return the **largest-valued odd integer** (as a string) that can be formed as a **substring** of  $s$ .

Rules:

- The returned number **must be odd**
  - It **must not have leading zeroes**
  - The input string **may have leading zeroes**
  - If no odd number exists, return an **empty string**
- 

## Example 1

Input:

s = "5347"

Output:

"5347"

Explanation:

Odd substrings → 5, 3, 53, 347, 5347

Largest odd number = 5347

---

## Example 2

Input:

s = "0214638"

Output:

"21463"

Explanation:

Odd substrings → 1, 3, 21, 63, 463, 1463, 21463

Substrings starting with 0 are invalid

Largest valid odd number = 21463

---

# Approach

## Key Observations

1. **Odd number rule**  
→ An odd number **must end with an odd digit** (1, 3, 5, 7, 9)
  2. **Largest value rule**  
→ Larger length ⇒ larger number  
→ So we want the **longest possible substring** ending at an odd digit
  3. **Leading zero rule**  
→ We must **skip leading zeroes** in the final answer
- 

## Strategy

1. Scan the string from **right to left**
    - Find the **last odd digit**
    - This will be the **end** of the largest odd number
  2. From the **start of the string**, skip all leading zeroes **up to that index**
  3. Extract the substring between these two positions
- 

## Step-by-Step

For s = "0214638"

- Scan from right → last odd digit is '3' at index 5
- Skip leading zeroes → start from '2'
- Substring → "21463"

---

## C++ Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 string largeOddNum(string& s) {
 int ind = -1;

 // Step 1: find last odd digit
 for (int i = s.length() - 1; i >= 0; i--) {
 if ((s[i] - '0') % 2 == 1) {
 ind = i;
 break;
 }
 }

 // If no odd digit found
 if (ind == -1) return "";

 // Step 2: skip leading zeroes
 int start = 0;
 while (start <= ind && s[start] == '0') {
 start++;
 }

 // Step 3: extract result
 return s.substr(start, ind - start + 1);
 }
};

int main() {
 Solution solution;
 string s = "0214638";
 cout << solution.largeOddNum(s) << endl;
 return 0;
}
```

}

---

## Complexity Analysis

- **Time Complexity:**  $O(N)$   
(single pass from right + single pass from left)
- **Space Complexity:**  $O(1)$   
(only indices used, output excluded)

# 4. Longest Common Prefix

## Problem Statement

Given an array of strings, find the **longest common prefix** shared by **all** strings.  
If no common prefix exists, return an empty string " ".

---

## Example 1

Input :

```
["flower", "flow", "flight"]
```

Output :

```
"fl"
```

All strings start with "fl".

---

## Example 2

Input:

```
["apple", "banana", "grape", "mango"]
```

Output:

```
" "
```

No common starting sequence exists.

---

## Approach (Sorting + Comparison)

### Key Insight

After sorting strings **lexicographically**:

- The **first** and **last** strings will be the **most different**
- Any common prefix shared by **all** strings must also be shared by these two

So instead of comparing all strings:

👉 Only compare the **first** and **last** string after sorting

---

### Step-by-Step Algorithm

1. If the array is empty → return ""
2. Sort the array of strings
3. Take:

- `first = str[0]`
  - `last = str[n-1]`
4. Compare characters from index 0 onwards
  5. Stop when characters differ
  6. Return the matched prefix
- 

## Why This Works

Sorting ensures:

- Strings with **smallest** and **largest** lexicographical order are farthest apart
  - If they match up to some point, **all middle strings must also match**
- 

## C++ Implementation

```
#include <bits/stdc++.h>

using namespace std;

class Solution {

public:

 string longestCommonPrefix(vector<string>& str) {

 if (str.empty()) return "";

 // Sort strings lexicographically
```

```

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

```

string first = str[0];
string last = str[str.size() - 1];
```

```

string ans = "";
int minLen = min(first.size(), last.size());
```

```

// Compare characters of first and last string
for (int i = 0; i < minLen; i++) {
 if (first[i] != last[i])
 break;
 ans += first[i];
}
```

```

return ans;
}
```

```

};

int main() {
 vector<string> input = {"interview", "internet", "internal",
 "interval"};
 Solution sol;
```

```
cout << sol.longestCommonPrefix(input) << endl;
return 0;
}
```

---

## Complexity Analysis

- **Time Complexity:**

$$O(N \log N + M)$$

- Sorting  $\rightarrow O(N \log N)$

- Prefix comparison  $\rightarrow O(M)$   
(N = number of strings, M = min string length)

- **Space Complexity:**

$$O(M) \text{ for storing the prefix}$$

# 5. Isomorphic String

---

## Problem Statement

Given two strings s and t, check whether they are **isomorphic**.

Two strings are isomorphic if:

- Each character in s can be replaced to get t
- **Same character must always map to the same character**

- No two different characters can map to the same character
  - Order of characters must remain the same
- 

## Example 1

Input:

```
s = "paper"
t = "title"
```

Output:

```
true
```

Mapping:

```
p → t
a → i
p → t (same as before ✓)
e → l
r → e
```

Mapping is consistent → **isomorphic**

---

## Example 2

Input:

```
s = "foo"
t = "bar"
```

Output:

```
false
```

Mapping:

```
f → b
o → a
```

$o \rightarrow r$   (conflict)

Same character  $o$  maps to two different characters → **not isomorphic**

---

## Key Idea (Very Important)

Instead of directly storing character-to-character mapping, we compare **patterns of occurrence**.

If two characters:

- First appeared at the same index
- Last appeared at the same index

Then they follow the **same structure**.

---

## Optimal Approach (Pattern Matching using Last Seen Index)

### Core Logic

- Use two arrays of size 256 (for all ASCII characters)
- Store **last seen position** of each character
- Traverse both strings together
- At every index  $i$ :
  - If last seen position of  $s[i] \neq$  last seen position of  $t[i]$  → mapping conflict → return false
  - Otherwise, update both positions

Why  $i + 1$ ?

- Default value is 0
  - Using  $i + 1$  avoids confusion with unseen characters
- 

## Step-by-Step Algorithm

1. If lengths of s and t differ → return false
2. Create two arrays  $m1$  and  $m2$  of size 256, initialized to 0
3. Traverse both strings from left to right
4. At index  $i$ :
  - If  $m1[s[i]] \neq m2[t[i]] \rightarrow$  return false

Update:

```
m1[s[i]] = i + 1
m2[t[i]] = i + 1
```

- 
5. If loop completes → return true
- 

## C++ Implementation

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 bool isomorphicString(string s, string t) {
 // If lengths differ, they cannot be isomorphic
 if (s.size() != t.size()) return false;
```

```

int m1[256] = {0};
int m2[256] = {0};

for (int i = 0; i < s.size(); i++) {
 // If last seen positions do not match → conflict
 if (m1[s[i]] != m2[t[i]])
 return false;

 // Update last seen position
 m1[s[i]] = i + 1;
 m2[t[i]] = i + 1;
}
return true;
}

int main() {
 Solution solution;
 string s = "paper";
 string t = "title";

 cout << (solution.isomorphicString(s, t) ? "true" : "false") <<
endl;
 return 0;
}

```

---

## Complexity Analysis

- **Time Complexity:**  $O(N)$   
Single pass through both strings
  - **Space Complexity:**  $O(1)$   
Fixed-size arrays (256), constant space
- 
- No hashmap overhead

- Very fast
  - Directly checks **structure**, not characters
  - Works for all ASCII characters
- 

## One-Line Intuition

If two strings follow the **same pattern of repetition**, they are isomorphic.

# 6. Check if One String is Rotation of Another

---

## Problem Statement

Given two strings `s` and `goal`, return **true** if `s` can be converted into `goal` using **left rotations**. A left rotation means moving the **first character to the end**.

---

## Example 1

Input:

```
s = "rotation"
goal = "tionrota"
```

Output:

```
true
```

After rotations:

```
rotation → otationr → tationro → ationrot → tionrota
```

---

## Example 2

Input:

```
s = "hello"
goal = "lohelx"
```

Output:

```
false
```

Extra character x makes rotation impossible.

---

## Brute Force Approach

### Idea

- Generate **all rotations** of s
  - Compare each rotation with goal
  - If any match → return true
- 

### Algorithm

1. If lengths are different → return false
2. For  $i = 0$  to  $n-1$ :
  - Create rotation:  $s.substr(i) + s.substr(0, i)$
  - If rotation == goal → return true

3. Return false
- 

### C++ Code (Brute Force)

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 bool rotateString(string& s, string& goal) {
 if (s.length() != goal.length()) return false;

 for (int i = 0; i < s.length(); i++) {
 string rotated = s.substr(i) + s.substr(0, i);
 if (rotated == goal)
 return true;
 }
 return false;
 }
};

int main() {
 Solution sol;
 string s = "rotation";
 string goal = "tionrota";
 cout << (sol.rotateString(s, goal) ? "true" : "false") << endl;
 return 0;
}
```

---

### Complexity

- **Time:**  $O(N^2)$
- **Space:**  $O(N)$

---

## Optimal Approach (Key Trick)

### Core Observation

If `goal` is a rotation of `s`, then:

**goal must be a substring of `s + s`**

Example:

```
s = "abcde"
s+s = "abcdeabcde"
```

All rotations exist inside this string

---

### Why This Works

- Rotation does not change order, only shifts starting point
  - Doubling `s` contains **all possible rotations**
  - Substring check confirms existence
- 

### Algorithm

1. If lengths differ → return false
  2. Create `doubled = s + s`
  3. If `goal` is a substring of `doubled` → return true
  4. Else → return false
-

## C++ Code (Optimal)

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 bool rotateString(string& s, string& goal) {
 if (s.length() != goal.length()) return false;
 string doubled = s + s;
 return doubled.find(goal) != string::npos;
 }
};

int main() {
 Solution sol;
 string s = "rotation";
 string goal = "tionrota";
 cout << (sol.rotateString(s, goal) ? "true" : "false") << endl;
 return 0;
}
```

---

## Complexity

- **Time:**  $O(N)$
  - **Space:**  $O(N)$
- 

## One-Line Intuition

If `goal` is a rotation of `s`, it must appear somewhere inside `s + s`.

# 7. Check if Two Strings Are Anagrams

---

## Problem Statement

Given two strings, check whether they are **anagrams** of each other.

Two strings are anagrams if:

- They contain **exactly the same characters**
  - Each character appears the **same number of times**
  - Order does **not** matter
- 

## Example 1

Input: "CAT", "ACT"

Output: true

Both strings contain: C, A, T (same count)

---

## Example 2

Input: "RULES", "LESRT"

Output: false

Character counts differ (U vs T)

---

## Brute Force Approach (Sorting)

### Idea

If two strings are anagrams, then after sorting:

- Both strings will look **exactly the same**
- 

## Algorithm

1. If lengths differ → return false
  2. Sort both strings
  3. Compare characters one by one
  4. If all match → return true, else false
- 

## C++ Code (Sorting)

```
#include <bits/stdc++.h>
using namespace std;

bool CheckAnagrams(string str1, string str2) {
 if (str1.length() != str2.length())
 return false;

 sort(str1.begin(), str1.end());
 sort(str2.begin(), str2.end());

 for (int i = 0; i < str1.length(); i++) {
 if (str1[i] != str2[i])
 return false;
 }
 return true;
}

int main() {
 string a = "INTEGER";
 string b = "TEGERNI";
```

```
 cout << (CheckAnagrams(a, b) ? "True" : "False") << endl;
 return 0;
}
```

---

## Complexity

- **Time:**  $O(N \log N)$  (due to sorting)
  - **Space:**  $O(1)$  (in-place sort)
- 

## Optimal Approach (Frequency Counting)

### Key Observation

Anagrams depend on **character frequency**, not order.

So:

- Count characters in first string
  - Subtract counts using second string
  - If all counts become zero → anagram
- 

### Algorithm

1. If lengths differ → return false
2. Create frequency array of size 26
3. Traverse first string → increment count
4. Traverse second string → decrement count

5. If any count  $\neq 0 \rightarrow$  return false
  6. Else  $\rightarrow$  return true
- 

### C++ Code (Optimal)

```
#include <bits/stdc++.h>
using namespace std;

bool CheckAnagrams(string str1, string str2) {
 if (str1.length() != str2.length())
 return false;

 int freq[26] = {0};

 for (char c : str1)
 freq[c - 'A']++;

 for (char c : str2)
 freq[c - 'A']--;

 for (int i = 0; i < 26; i++) {
 if (freq[i] != 0)
 return false;
 }
 return true;
}

int main() {
 string a = "INTEGER";
 string b = "TEGERNI";
 cout << (CheckAnagrams(a, b) ? "True" : "False") << endl;
 return 0;
}
```

---

- **Time:**  $O(N)$
  - **Space:**  $O(1)$  (fixed-size array)
- 

## Why Optimal Approach Is Better

- No sorting overhead
  - Linear time
  - Directly checks the core requirement (frequency)
- 

### One-Line Intuition

Two strings are anagrams if **every character appears the same number of times** in both.

## 8. Sort Characters by Frequency

---

### Problem Statement

Given a string  $s$ , return the **unique characters** sorted by:

1. **Higher frequency first**
  2. If frequencies are equal → **alphabetical order**
-

## **Example 1**

Input: "tree"

Output: ['e', 'r', 't']

Frequency:

e → 2

r → 1

t → 1

Same frequency → alphabetical (r before t)

---

## **Example 2**

Input: "raaaajj"

Output: ['a', 'j', 'r']

Frequency:

a → 4

j → 2

r → 1

---

## **Approach (Frequency + Sorting)**

### **Core Idea**

- Count how many times each character appears
- Sort characters based on:
  - **frequency (descending)**
  - **character (ascending)** if frequency ties

---

## Step-by-Step Algorithm

1. Create a frequency array of size 26
  2. Count frequency of each character
  3. Store (frequency, character) pairs
  4. Sort using a custom comparator:
    - o Higher frequency first
    - o Alphabetical order if equal
  5. Collect characters with frequency > 0
- 

## C++ Implementation

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
private:
 // Comparator: frequency desc, character asc
 static bool cmp(pair<int, char> a, pair<int, char> b) {
 if (a.first != b.first)
 return a.first > b.first;
 return a.second < b.second;
 }

public:
 vector<char> frequencySort(string& s) {
 pair<int, char> freq[26];

 // Initialize
 for (int i = 0; i < 26; i++)
```

```

 freq[i] = {0, char('a' + i)};

 // Count frequency
 for (char c : s)
 freq[c - 'a'].first++;

 // Sort by frequency and alphabet
 sort(freq, freq + 26, cmp);

 // Collect result
 vector<char> ans;
 for (int i = 0; i < 26; i++) {
 if (freq[i].first > 0)
 ans.push_back(freq[i].second);
 }

 return ans;
}

};

int main() {
 Solution sol;
 string s = "tree";
 vector<char> res = sol.frequencySort(s);

 for (char c : res)
 cout << c << " ";
 return 0;
}

```

---

## Complexity Analysis

- **Time Complexity:**  $O(N + 26 \log 26) \rightarrow$  effectively  $O(N)$
- **Space Complexity:**  $O(26) \rightarrow$  constant space

# 9. Maximum Nesting Depth of Parenthesis

---

## Problem Statement

Given a **valid parentheses string**  $s$ , return its **maximum nesting depth**.

**Nesting depth** = maximum number of parentheses that are **open at the same time**.

---

## Example 1

Input:

"(1+(2\*3)+((8)/4))+1"

Output:

3

At digit 8:

((8))  
↑ ↑ ↑ → depth = 3

---

## Example 2

Input:

"(1)+((2))+(((3)))"

Output:

3

Deepest nesting happens at 3.

---

## Key Intuition

Think of parentheses like **going up and down floors**:

- ' ( ' → go **one level deeper**
  - ' ) ' → come **one level back**
  - Track the **maximum level reached**
- 

## Step-by-Step Algorithm

1. Initialize:
    - depth = 0 → current nesting level
    - maxDepth = 0 → answer
  2. Traverse the string character by character
  3. If character is ' ( ':
    - Increase depth
    - Update maxDepth
  4. If character is ' ) ':
    - Decrease depth
  5. Ignore all other characters
  6. Return maxDepth
-

## C++ Implementation

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int maxDepth(string s) {
 int depth = 0;
 int maxDepth = 0;

 for (char c : s) {
 if (c == '(') {
 depth++;
 maxDepth = max(maxDepth, depth);
 }
 else if (c == ')') {
 depth--;
 }
 }
 return maxDepth;
 }
};

int main() {
 Solution sol;
 string s = "(1+(2*3)+((8)/4))+1";
 cout << sol.maxDepth(s) << endl;
 return 0;
}
```

---

## Complexity Analysis

- **Time Complexity:**  $O(N)$   
Single traversal of the string

- **Space Complexity:**  $O(1)$   
Only two integer variables used
- 

## Dry Run (Quick)

String: "((a)+(b))"

| Char | depth | maxDepth |
|------|-------|----------|
|------|-------|----------|

|   |   |   |
|---|---|---|
| ( | 1 | 1 |
|---|---|---|

|   |   |   |
|---|---|---|
| ( | 2 | 2 |
|---|---|---|

|   |   |   |
|---|---|---|
| a | 2 | 2 |
|---|---|---|

|   |   |   |
|---|---|---|
| ) | 1 | 2 |
|---|---|---|

|   |   |   |
|---|---|---|
| + | 1 | 2 |
|---|---|---|

|   |   |   |
|---|---|---|
| ( | 2 | 2 |
|---|---|---|

|   |   |   |
|---|---|---|
| b | 2 | 2 |
|---|---|---|

|   |   |   |
|---|---|---|
| ) | 1 | 2 |
|---|---|---|

|   |   |   |
|---|---|---|
| ) | 0 | 2 |
|---|---|---|

Answer = 2

---

## One-Line Intuition

Maximum nesting depth is the highest number of ' ( ' opened at any moment.

# 10. Roman Numerals to Integer

---

## Problem Statement

Roman numerals use the following symbols:

| Symbol | Value |
|--------|-------|
| I      | 1     |
| V      | 5     |
| X      | 10    |
| L      | 50    |
| C      | 100   |
| D      | 500   |
| M      | 1000  |

Normally, values are **added from left to right**.

But in some cases, **subtraction** is used:

- I before V or X → 4, 9
- X before L or C → 40, 90
- C before D or M → 400, 900

Your task: **convert a Roman numeral string into an integer.**

---

## Example 1

Input: "LVIII"

Output: 58

Explanation:

L = 50  
V = 5  
III = 3  
Total = 58

---

## Example 2

Input: "MCMXCIV"  
Output: 1994

Explanation:

M = 1000  
CM = 900  
XC = 90  
IV = 4  
Total = 1994

---

## Key Idea (Most Important)

While scanning the string:

- If **current value < next value** → subtract it
- Else → add it

Why?

- Smaller numeral before a larger one means subtraction (IV, IX, etc.)
-

## Step-by-Step Algorithm

1. Create a map of Roman symbols to values
  2. Initialize **result** = 0
  3. Traverse the string from **left to right**
  4. At index i:
    - o If  $i + 1 < n$  and  $\text{value}(s[i]) < \text{value}(s[i+1])$ 
      - Subtract  $\text{value}(s[i])$
    - o Else
      - Add  $\text{value}(s[i])$
  5. Return **result**
- 

## C++ Implementation

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int romanToInt(string s) {
 unordered_map<char, int> roman = {
 {'I', 1}, {'V', 5}, {'X', 10},
 {'L', 50}, {'C', 100}, {'D', 500}, {'M', 1000}
 };

 int result = 0;
 int n = s.size();

 for (int i = 0; i < n; i++) {
```

```

 // If next character exists and current value is smaller
 if (i + 1 < n && roman[s[i]] < roman[s[i + 1]]) {
 result -= roman[s[i]];
 } else {
 result += roman[s[i]];
 }
 }
 return result;
}

int main() {
 Solution sol;
 cout << sol.romanToInt("LVIII") << endl; // 58
 cout << sol.romanToInt("MCMXCIV") << endl; // 1994
 return 0;
}

```

---

## Dry Run (Very Important)

**Input: "MCMXCIV"**

| Index | Char | Value | Next | Action   | Result |
|-------|------|-------|------|----------|--------|
| 0     | M    | 1000  | C    | add      | 1000   |
| 1     | C    | 100   | M    | subtract | 900    |
| 2     | M    | 1000  | X    | add      | 1900   |
| 3     | X    | 10    | C    | subtract | 1890   |
| 4     | C    | 100   | I    | add      | 1990   |
| 5     | I    | 1     | V    | subtract | 1989   |
| 6     | V    | 5     | —    | add      | 1994   |

---

## Complexity Analysis

- **Time Complexity:**  $O(N)$   
Single pass through the string
- **Space Complexity:**  $O(1)$   
Fixed-size map (7 symbols)

---

# 11. Implement atoi (String to Integer)

---

### Problem Statement

Implement `myAtoi(string s)` that converts a string into a **32-bit signed integer**, following these rules **in order**:

1. **Ignore leading spaces**
  2. **Check sign** (- or +, default is positive)
  3. **Read digits** until a non-digit appears
  4. **If no digits are read → return 0**
  5. **Clamp result** to range  
[-2<sup>31</sup>, 2<sup>31</sup>-1] [-2<sup>31</sup>, 2<sup>31</sup>-1]
- 

### Examples

#### Example 1

Input: "1337c0d3"

Output: 1337

Stops reading at 'c'.

### Example 2

Input: "words and 987"

Output: 0

First character is non-digit → no conversion.

---

## Core Idea (Very Important)

We **scan the string once**, building the number digit by digit, while carefully:

- skipping spaces
  - handling sign
  - stopping at non-digits
  - checking overflow **before it happens**
- 

## Step-by-Step Algorithm

### 1. Initialize:

- `i = 0 (index)`
- `sign = 1`
- `result = 0 (use long to detect overflow)`

### 2. Skip leading spaces

### 3. Check sign

- '-' → sign = -1
- '+' → sign = 1

### 4. Read digits

- result = result \* 10 + digit
- After each step, check:
  - if sign \* result > INT\_MAX → return INT\_MAX
  - if sign \* result < INT\_MIN → return INT\_MIN

### 5. Return sign \* result

---

## Why Overflow Check Is Needed Early

If we wait until the end, the number may already overflow.

So we **check during construction**, not after.

---

## C++ Implementation

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int myAtoi(string s) {
 int i = 0, sign = 1;
 long res = 0;

 // 1. Skip leading whitespaces
```

```

 while (i < s.size() && s[i] == ' ')
 i++;

 // 2. Check sign
 if (i < s.size() && (s[i] == '+' || s[i] == '-')) {
 if (s[i] == '-') sign = -1;
 i++;
 }

 // 3. Convert digits
 while (i < s.size() && isdigit(s[i])) {
 res = res * 10 + (s[i] - '0');

 // 4. Clamp if overflow
 if (sign * res > INT_MAX)
 return INT_MAX;
 if (sign * res < INT_MIN)
 return INT_MIN;

 i++;
 }

 return sign * res;
 }
};

int main() {
 Solution sol;
 cout << sol.myAtoi(" -42") << endl; // -42
 cout << sol.myAtoi("4193 with words") << endl; // 4193
 cout << sol.myAtoi("words and 987") << endl; // 0
 return 0;
}

```

---

## Dry Run (Important)

**Input:** " -42"

```
Skip spaces → index at '-'
sign = -1
Read digits: 4 → 42
Result = -42
```

---

**Input:** "91283472332"

```
Number exceeds INT_MAX
Return 2147483647
```

---

## Complexity Analysis

- **Time Complexity:**  $O(N)$   
Single pass through the string
  - **Space Complexity:**  $O(1)$   
Only a few variables
- 

## Common Mistakes to Avoid

- ✗ Not skipping leading spaces
  - ✗ Not handling sign
  - ✗ Parsing digits after non-digit
  - ✗ Checking overflow only at the end
- 

## One-Line Intuition

Skip spaces → read sign → build number digit by digit → stop early if overflow.

# 12. Count Number of Substrings with Exactly K Distinct Characters

---

## Problem Statement

You are given:

- a string s
- an integer k

Return the **number of substrings** of s that contain **exactly k distinct characters**.

---

## Example 1

Input:

s = "pqpqs", k = 2

Output:

7

Valid substrings:

"pq", "pqp", "pqpq", "qp", "qpq", "pqs", "qs"

---

## Example 2

Input:

s = "abcbaa", k = 3

Output:

Valid substrings:

"abc", "abcb", "abcba", "bcba", "cbaa"

---

## Important Insight (Very Important)

Directly counting substrings with **exactly K** distinct characters is hard.

So we use this trick:

$$\text{Exactly } K = \text{At Most } K - \text{At Most } (K - 1)$$

Why?

- Substrings with **at most K** distinct characters include:
    - substrings with 1, 2, ..., K distinct characters
  - Subtracting **at most (K-1)** removes all substrings with fewer than K distinct characters
  - What remains → **exactly K**
- 

## Helper Function: `atMostKDistinct(s, k)`

This function returns the number of substrings with **at most k distinct characters**.

---

## Sliding Window Logic (Core Concept)

We use:

- left pointer → start of window

- right pointer → end of window
- a frequency map to count characters in the window

## Window Rules

- Expand window by moving right
- If distinct characters exceed k, shrink window from left

For every valid window, add

$(right - left + 1)$

- because all substrings ending at right and starting from left...right are valid
- 

## Step-by-Step Algorithm

### **atMostKDistinct(s, k)**

1. Initialize left = 0, result = 0
  2. Traverse string with right
  3. Add s[right] to frequency map
  4. If distinct characters > k:
    - shrink window from left
  5. Add  $(right - left + 1)$  to result
  6. Return result
-

## Exactly K Distinct

```
answer = atMostKDistinct(s, k)
 - atMostKDistinct(s, k - 1)
```

---

## C++ Implementation

```
#include <bits/stdc++.h>
using namespace std;

// Count substrings with at most k distinct characters
int atMostKDistinct(string s, int k) {
 int left = 0, res = 0;
 unordered_map<char, int> freq;

 for (int right = 0; right < s.size(); right++) {
 freq[s[right]]++;

 // Shrink window if distinct characters exceed k
 while (freq.size() > k) {
 freq[s[left]]--;
 if (freq[s[left]] == 0)
 freq.erase(s[left]);
 left++;
 }

 // Count valid substrings ending at 'right'
 res += (right - left + 1);
 }
 return res;
}

// Count substrings with exactly k distinct characters
int countSubstrings(string s, int k) {
 return atMostKDistinct(s, k) - atMostKDistinct(s, k - 1);
}

int main() {
```

```
string s = "pqpqs";
int k = 2;
cout << countSubstrings(s, k) << endl; // Output: 7
return 0;
}
```

---

## Dry Run (Small Insight)

For  $s = "pqpqs"$ ,  $k = 2$ :

- `atMostKDistinct(2)` counts substrings with 1 or 2 distinct chars
  - `atMostKDistinct(1)` counts substrings with only 1 distinct char
  - Difference → substrings with **exactly 2**
- 

## Complexity Analysis

- **Time Complexity:**  $O(N)$ 
    - Each character enters and leaves the window once
  - **Space Complexity:**  $O(1)$ 
    - Map size bounded by alphabet size ( $\leq 26$  for lowercase)
- 

## One-Line Intuition

Count substrings with at most  $K$  distinct characters, subtract those with at most  $K-1$ .

# 13. Longest Palindromic Substring

---

## Problem Statement

Given a string  $s$ , return the **longest substring** of  $s$  that is a **palindrome**.

A palindrome reads the same **forward and backward**.

---

## Examples

### Example 1

Input: "babad"

Output: "bab"

("aba" is also valid)

### Example 2

Input: "cbbd"

Output: "bb"

---

## Core Idea (Very Important)

Every palindrome has a **center**.

There are **two types** of palindromes:

1. **Odd length** → center at a character  
Example: "bab" (center = a)
2. **Even length** → center between two characters  
Example: "bb" (center between b and b)

👉 So for every index, we try **both cases** and expand outward.

---

## How Expansion Works

From a center:

- Move left to the left
- Move right to the right
- Keep expanding **while characters match**

When they stop matching → palindrome ends.

---

## Step-by-Step Algorithm

1. Initialize `start = 0, end = 0`
2. For each index `i` in the string:
  - Check **odd-length palindrome** (`center = i`)
  - Check **even-length palindrome** (`center = i, i+1`)
3. Take the **maximum length** from both
4. Update `start` and `end` if a longer palindrome is found
5. Return `s.substr(start, end - start + 1)`

---

## Why start and end Formula Works

For a palindrome of length `len` centered at `i`:

`start = i - (len - 1) / 2`

```
end = i + len / 2
```

This formula works for **both odd and even lengths**.

---

## C++ Implementation

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 string longestPalindrome(string s) {
 int start = 0, end = 0;

 for (int i = 0; i < s.length(); i++) {
 int len1 = expandFromCenter(s, i, i); // odd
 int len2 = expandFromCenter(s, i, i + 1); // even
 int len = max(len1, len2);

 if (len > end - start) {
 start = i - (len - 1) / 2;
 end = i + len / 2;
 }
 }
 return s.substr(start, end - start + 1);
 }

private:
 int expandFromCenter(const string& s, int left, int right) {
 while (left >= 0 && right < s.length() && s[left] == s[right])
 {
 left--;
 right++;
 }
 return right - left - 1;
 }
};
```

```
int main() {
 Solution sol;
 cout << sol.longestPalindrome("babad") << endl;
 return 0;
}
```

---

## Dry Run (Quick)

String: "cbbd"

- $i = 1$
- Even center  $(1, 2) \rightarrow "bb"$
- Length = 2  $\rightarrow$  longest so far

Answer = "bb"

---

## Complexity Analysis

- **Time Complexity:**  $O(N^2)$ 
    - Each index expands up to N
  - **Space Complexity:**  $O(1)$ 
    - No extra data structures
- 

## One-Line Intuition

Try every index as a center and expand outward to find the longest palindrome.

# 14. Sum of Beauty of All Substrings

---

## Problem Recap

The **beauty** of a string =

(max frequency of any character) - (min frequency of any character)

👉 Ignore characters with zero frequency

You need to compute the **sum of beauty values of all substrings** of a given string s.

---

## Key Observations

1. Substrings of length **1** always have beauty **0**  
(only one character → max = min)
  2. Beauty depends only on **character frequencies**, not order.
  3. Since the string contains lowercase letters, frequency size is at most **26**.
- 

## Approach (Brute Force but Optimal Enough)

We fix a **starting index i**

Then extend the substring to the right using **ending index j**

For each fixed i:

- Maintain a frequency array/map
- As j increases, update frequency of  $s[j]$
- Compute:

- `maxi` = maximum frequency
  - `mini` = minimum **non-zero** frequency
- Add  $(\text{maxi} - \text{mini})$  to the answer

This avoids recomputing frequencies from scratch.

---

## Why This Works

- Each substring is considered exactly once
  - Frequency update is incremental
  - Checking max/min over at most 26 characters → constant time
- 

## C++ Implementation (Clean & Clear)

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int beautySum(string s) {
 int n = s.size();
 int ans = 0;

 // Fix starting point
 for (int i = 0; i < n; i++) {
 vector<int> freq(26, 0);

 // Extend substring
 for (int j = i; j < n; j++) {
 freq[s[j] - 'a']++;

 // Calculate current beauty
 int maxi = *max_element(freq.begin(), freq.end());
 int mini = *min_element(freq.begin(), freq.end());
 ans += maxi - mini;
 }
 }
 }
}
```

```

int maxi = 0;
int mini = INT_MAX;

// Calculate beauty
for (int k = 0; k < 26; k++) {
 if (freq[k] > 0) {
 maxi = max(maxi, freq[k]);
 mini = min(mini, freq[k]);
 }
}

ans += (maxi - mini);
}
}
return ans;
}
};

int main() {
 Solution sol;
 cout << sol.beautySum("xyx") << endl; // Output: 1
 cout << sol.beautySum("aabcbaa") << endl; // Output: 17
 return 0;
}

```

---

## Dry Run (Short Example)

**Input:** "xyx"

Substrings and beauty:

- "x" → 0
- "y" → 0
- "xy" → 0

- "xy" → x:1, y:1 → 0
- "yx" → y:1, x:1 → 0
- "xyx" → x:2, y:1 → 1

✓ Total = 1

---

## Complexity Analysis

- **Time Complexity:**
  - Outer loop:  $O(n)$
  - Inner loop:  $O(n)$
  - Frequency scan:  $O(26)$   
👉 Overall:  $O(n^2)$
- **Space Complexity:**
  - Frequency array of size 26 →  $O(1)$

---

## One-Line Intuition

Fix a start, grow the substring, track frequencies, and keep adding (max – min).

# 15. Reverse Every Word in a String

---

## Problem Statement

Given an input string containing upper-case and lower-case letters, digits, and spaces (' '), reverse the **order of words** in the string.

- A word is defined as a sequence of non-space characters.
  - Words are separated by at least one space.
  - The output should:
    - have **no leading or trailing spaces**
    - have **exactly one space** between words
- 

## Examples

### Example 1

#### Input

```
s = "welcome to the jungle"
```

#### Output

```
"jungle the to welcome"
```

---

### Example 2

#### Input

```
s = " amazing coding skills "
```

## Output

"skills coding amazing"

---

## Brute Force Approach

### Algorithm

1. Traverse the string and extract all words while ignoring extra spaces.
  2. Store each word in a list/vector.
  3. Reverse the list of words.
  4. Join the words using a single space.
- 

### Code (Brute Force)

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 string reverseWords(string s) {
 vector<string> words;
 int n = s.size();
 int i = 0;

 while (i < n) {
 // Skip spaces
 while (i < n && s[i] == ' ') i++;
 if (i >= n) break;

 int start = i;
 while (i < n && s[i] != ' ') i++;
 words.push_back(s.substr(start, i - start));
 }

 reverse(words.begin(), words.end());
 string result;
 for (int i = 0; i < words.size(); i++) {
 result += words[i];
 if (i < words.size() - 1) result += " ";
 }
 return result;
 }
}
```

```

 words.push_back(s.substr(start, i - start));
 }

 // Reverse words
 reverse(words.begin(), words.end());

 // Join with single space
 string ans = "";
 for (int i = 0; i < words.size(); i++) {
 ans += words[i];
 if (i != words.size() - 1) ans += " ";
 }

 return ans;
}
};

int main() {
 Solution sol;
 string s = " amazing coding skills ";
 cout << sol.reverseWords(s);
 return 0;
}

```

---

## Complexity Analysis (Brute Force)

- **Time Complexity:**  $O(n)$
  - **Space Complexity:**  $O(n)$  (stores words separately)
- 

## Optimal Approach

### Algorithm

1. Reverse the entire string.
  2. Traverse the reversed string word by word.
  3. Reverse each word individually to restore correct character order.
  4. While processing, skip extra spaces and ensure only one space between words.
  5. Remove any trailing space.
- 

## Code (Optimal)

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
private:
 void reversePart(string &s, int l, int r) {
 while (l < r) {
 swap(s[l++], s[r--]);
 }
 }

public:
 string reverseWords(string s) {
 int n = s.size();

 // Reverse entire string
 reversePart(s, 0, n - 1);

 int i = 0, j = 0;

 while (j < n) {
 // Skip spaces
 while (j < n && s[j] == ' ') j++;
 if (j >= n) break;

 int start = i;
```

```

 // Copy characters of word
 while (j < n && s[j] != ' ') {
 s[i++] = s[j++];
 }
//j hamesha i ke barabar ya aage hota hai
// → overwrite ka risk nahi hota.

 // Reverse current word
 reversePart(s, start, i - 1);

 // Add single space
 s[i++] = ' ';
 }

 // Remove trailing space
 if (i > 0) i--;

 return s.substr(0, i);
}
};

int main() {
 Solution sol;
 string s = " amazing coding skills ";
 cout << sol.reverseWords(s);
 return 0;
}

```

---

## Complexity Analysis (Optimal)

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

# HARD PROBLEMS

## 16. Minimum Number of Bracket Reversals Needed to Make an Expression Balanced

Given a string  $s$  consisting only of opening ' $($ ' and closing ' $)$ ' brackets, you have to find the **minimum number of reversals** required to make the expression **balanced**.

A reversal means:

- ' $($ ' can be changed to ' $)$ '
- ' $)$ ' can be changed to ' $($ '

If it is **not possible** to balance the string, return **-1**.

A balanced expression means every opening bracket has a matching closing bracket in the correct order.

### **Example**

$s = ")()((("$

One possible balanced form is " $(((()))()$ "

Minimum reversals required = 3

$s = "((())((()))()$ "

Total length is odd, so it is impossible to balance

Output = -1

---

### **Approach**

Using Counting of Unmatched Brackets

### **Algorithm**

1. If the length of the string is **odd**, it is impossible to balance because brackets must form pairs. Return -1.
2. Initialize two counters:
  - o `openBrackets` → count of unmatched '('
  - o `closeBrackets` → count of unmatched ')'.
3. Traverse the string character by character:
  - o If the character is '(', increase `openBrackets`.
  - o If the character is ')':
    - If there is an unmatched '(', match it by decreasing `openBrackets`.
    - Otherwise, this ')' is unmatched, so increase `closeBrackets`.
4. After traversal, some brackets may remain unmatched.
5. Minimum reversals required:
  - o  $(\text{openBrackets} + 1) / 2$  reversals to fix unmatched '('
  - o  $(\text{closeBrackets} + 1) / 2$  reversals to fix unmatched ')'.
6. Return the sum of both.

### Why this works

- Two unmatched '(' can be fixed with one reversal.
- Same logic applies to unmatched ')'.
- Integer division handles both even and odd counts correctly.

### Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int minReversalsToBalance(string expression) {
 int openBrackets = 0;
 int closeBrackets = 0;

 for (char ch : expression) {
 if (ch == '(') {
 openBrackets++;
 } else {
 if (openBrackets > 0) {
 openBrackets--;
 } else {
 closeBrackets++;
 }
 }
 }
 }
}
```

```

 if ((openBrackets + closeBrackets) % 2 != 0)
 return -1;

 return (openBrackets + 1) / 2 + (closeBrackets + 1) / 2;
 }
};

int main() {
 string expression = ")())((";
 Solution solver;
 cout << solver.minReversalsToBalance(expression);
 return 0;
}

```

## Complexity Analysis

- **Time Complexity:** O(N)  
The string is traversed once.
- **Space Complexity:** O(1)  
Only constant extra variables are used.

# 17. Count and Say

The **count-and-say sequence** is a sequence of strings where each term is generated by **describing the previous term**.

Rules:

- `countAndSay(1) = "1"`
- `countAndSay(n)` is formed by applying **run-length encoding** on `countAndSay(n-1)`

Run-length encoding means counting **consecutive identical digits** and writing:  
`count + digit`

## **Example**

Input: n = 4

Sequence:

- countAndSay(1) = "1"
- countAndSay(2) = "11" → one 1
- countAndSay(3) = "21" → two 1s
- countAndSay(4) = "1211" → one 2, one 1

Output: "1211"

---

## **Approach**

### **Algorithm**

1. Start with the base string "1".
2. Repeat the following process n-1 times:
  - Traverse the current string from left to right.
  - Count how many times the current character repeats consecutively.
  - Append count followed by the character to form the next string.
3. Update the current string with the newly formed string.
4. After completing all iterations, return the final string.

### **Example Walkthrough**

Previous term = "21"

- One 2 → "12"
- One 1 → "11"

Next term = "1211"

### **Code**

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 string countAndSay(int n) {
 string result = "1";

 for (int i = 1; i < n; i++) {
 string current = "";
```

```

int count = 1;

for (int j = 1; j < result.size(); j++) {
 if (result[j] == result[j - 1]) {
 count++;
 } else {
 current += to_string(count) + result[j - 1];
 count = 1;
 }
}

current += to_string(count) + result.back();
result = current;
}
return result;
};

int main() {
 Solution solver;
 int n = 4;
 cout << solver.countAndSay(n);
 return 0;
}

```

## Complexity Analysis

- **Time Complexity:**  $O(n \cdot 2^n)$   
There are  $n$  iterations and the length of the string can grow exponentially.
- **Space Complexity:**  $O(2^n)$   
The final generated string can have exponential length.

# 18. Hashing in Strings

## Introduction:

In this article, we will be discussing the **Hashing in Strings**. Hashing refers to the process of converting a string into a numeric value, called a hash value, using a mathematical formula. It helps in comparing strings, searching efficiently, and solving complex problems like pattern matching.

---

## Need of Hashing:

Hashing is essential as it helps in following operations:

1. **Efficient Comparisons:** Comparing large strings is slow. Hash values allow comparison in constant time.
  2. **Faster Lookups:** Hash tables like dictionaries or unordered maps use hash values to store and retrieve strings quickly.
  3. **Detect Duplicates or Anagrams:** Hashing helps in checking duplicates or determining anagram groups efficiently.
  4. **Pattern Matching:** Algorithms like Rabin-Karp use string hashing to find substrings in linear time.
- 

## How Does String Hashing Work?:

String hashing treats each character as a number and combines them using a mathematical formula. A common formula is **Polynomial Rolling Hash**. Using this hash value is calculated as follows:

$$\text{Hash}(s) = (s[0] * p^0 + s[1] * p^1 + s[2] * p^2 + \dots + s[n-1] * p^{(n-1)}) \% m$$

where,  $s[i]$  is the character's ASCII or mapped value (like 'a' = 1),  
 $p$  is a prime base (commonly 31 or 53) and  
 $m$  is a large prime modulus to avoid overflow and reduce collisions.

---

## Handling Collisions:

Two different strings can have the same hash, called a collision. To reduce this, we use a large m (like  $1e9+7$ ). We can also use double hashing (combine two different hash functions) or use a good base (p) that spreads values well.

TUF

**Text=**"ccaccaaedba"  
**cca** =  $3+3+1 = 7$  , **cac** =  $3+3+1 = 7$  , **aae** =  $1+1+5=7$

**Pattern** = "dba" →  $4 + 2 + 1 = 7$

Example:

Calculate Hash value of string s = "abc".

Solution:

Let us assume p to be 31 and m to be  $1e9+9$ .

Now, we can map characters as following: 'a' = 1, 'b' = 2 and 'c' = 3.

Thus,  $\text{Hash}(s) = (1 * 31^0 + 2 * 31^1 + 3 * 31^2) \% 1e9+9$

$\text{Hash}(s) = 2946$ .

Therefore, "abc" gets hashed to value 2946.

# 19. Rabin Karp Algorithm – Pattern Searching

Given a string **text** and a string **pattern**, you need to find **all starting indices** where pattern occurs in text.

If the pattern does not occur, return an empty list.

The Rabin-Karp algorithm uses **hashing with a sliding window** to efficiently search for patterns in a string.

## Example

text = "ababcabcababc"

pattern = "abc"

The pattern "abc" appears starting at indices 2, 5, and 10.

Output = [2, 5, 10]

---

## Approach

Rabin-Karp Algorithm (Rolling Hash)

## Algorithm

1. If the pattern length is greater than the text length, return an empty list.
2. Choose:
  - o A base value (number of possible characters, usually 256).
  - o A prime modulus to reduce collisions.
3. Compute the hash of the pattern.
4. Compute the hash of the first window of text having the same length as the pattern.
5. Slide the window over the text one character at a time:
  - o If the current window hash matches the pattern hash:
    - Compare characters one by one to confirm the match.
  - o Update the window hash using the rolling hash formula:
    - Remove the leftmost character.
    - Add the next character to the window.

6. Store all indices where a confirmed match is found.
7. Return the list of indices.

### **Why character comparison is needed after hash match**

Different strings can sometimes have the same hash value (hash collision). So, whenever hashes match, we verify by direct comparison.

#### **Code**

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 vector<int> rabinKarp(string text, string pattern) {
 vector<int> result;

 int n = text.length();
 int m = pattern.length();
 if (m > n) return result;

 int base = 256;
 int mod = 101;

 int patternHash = 0;
 int windowHash = 0;
 int h = 1;

 for (int i = 0; i < m - 1; i++) {
 h = (h * base) % mod;
 }

 for (int i = 0; i < m; i++) {
 patternHash = (base * patternHash + pattern[i]) % mod;
 windowHash = (base * windowHash + text[i]) % mod;
 }

 for (int i = 0; i <= n - m; i++) {
 if (patternHash == windowHash) {
 bool match = true;
 for (int j = 0; j < m; j++) {
 if (text[i + j] != pattern[j]) {
 match = false;
 break;
 }
 }
 if (match)
 result.push_back(i);
 }
 }
 }
}
```

```

 }
 if (match) result.push_back(i);
}

if (i < n - m) {
 windowHash = (base * (windowHash - text[i] * h)
 + text[i + m]) % mod;
 if (windowHash < 0) windowHash += mod;
}
}

return result;
}
};

int main() {
 string text = "ababcabcababc";
 string pattern = "abc";

 Solution sol;
 vector<int> ans = sol.rabinKarp(text, pattern);

 for (int idx : ans) {
 cout << idx << " ";
 }
 return 0;
}

```

## Complexity Analysis

- **Time Complexity:**

Average case:  $O(n + m)$  due to rolling hash

Worst case:  $O(n * m)$  when many hash collisions occur and full comparisons are needed

- **Space Complexity:**

$O(1)$  since only constant extra variables are used

# 20. Z Function

Given a **text string** and a **pattern string**, you need to find **all starting indices** where the pattern occurs in the text using the **Z-function algorithm**.

All indices must be returned using **0-based indexing**.

## Example

text = "xyzabxyzabxyz"

pattern = "xyz"

The pattern "xyz" appears starting at indices 0, 5, and 10.

Output = [0, 5, 10]

---

## Approach

Using Z-Function Algorithm

## Algorithm

1. The Z-function helps find prefix matches efficiently in linear time.
2. Construct a new string by joining:
  - o pattern + "\$" + text  
(\$ is a separator that does not appear in pattern or text)
3. Compute the Z-array for this combined string.
  - o Z[i] tells how many characters starting at index i match the prefix of the string.
4. Traverse the Z-array:
  - o If Z[i] == pattern.length(), it means the pattern matches the text starting at:  
 $i - \text{pattern.length()} - 1$
5. Collect all such indices and return them.

## Why this works

- The prefix of the combined string is the pattern.
- A full prefix match at any position directly indicates a pattern occurrence.
- Previously computed matches are reused using a sliding window, avoiding repeated comparisons.

## Code

```
#include <bits/stdc++.h>
using namespace std;
```

```

class Solution {
public:
 vector<int> computeZArray(const string& s) {
 int n = s.length();
 vector<int> z(n, 0);
 int left = 0, right = 0;

 for (int i = 1; i < n; i++) {
 if (i <= right)
 z[i] = min(right - i + 1, z[i - left]);

 while (i + z[i] < n && s[z[i]] == s[i + z[i]])
 z[i]++;
 }

 if (i + z[i] - 1 > right) {
 left = i;
 right = i + z[i] - 1;
 }
 }
 return z;
}

vector<int> zFunctionSearch(const string& text, const string& pattern) {
 string combined = pattern + "$" + text;
 vector<int> z = computeZArray(combined);
 vector<int> result;

 for (int i = pattern.length() + 1; i < combined.length(); i++) {
 if (z[i] == pattern.length()) {
 result.push_back(i - pattern.length() - 1);
 }
 }
 return result;
}
};

int main() {
 string text = "xyzabxyzabxyz";
 string pattern = "xyz";

 Solution sol;
 vector<int> indices = sol.zFunctionSearch(text, pattern);
}

```

```

for (int idx : indices) {
 cout << idx << " ";
}
return 0;
}

```

## Complexity Analysis

- **Time Complexity:**  $O(n + m)$   
The Z-array is computed in linear time using a sliding window technique.
- **Space Complexity:**  $O(n + m)$   
Extra space is used for the combined string and the Z-array.

# 21. KMP Algorithm or LPS Array

Given a **text string** and a **pattern string**, you need to find **all starting indices** where the pattern occurs in the text using **0-based indexing**.

The indices must be returned in **ascending order**.

If the pattern does not occur in the text, return an **empty list**.

The solution must be implemented using the **Knuth-Morris-Pratt (KMP) algorithm**, which is an efficient pattern matching technique.

### Example

text = "abracadabra"

pattern = "abra"

The pattern "abra" occurs at index 0 and 7.

Output = [ 0, 7 ]

### Approach

KMP Algorithm using LPS Array

## Algorithm

1. When matching a pattern in a text using a simple approach, repeated comparisons happen after mismatches.
2. KMP avoids this by using the **LPS (Longest Prefix which is also Suffix)** array.
3. The LPS array tells how many characters can be skipped in the pattern after a mismatch.
4. First, build the LPS array for the pattern:
  - o `lps[i]` stores the length of the longest prefix of the pattern that is also a suffix ending at index *i*.
5. Start comparing characters of the text and pattern:
  - o If characters match, move both pointers forward.
  - o If a mismatch occurs:
    - Use the LPS array to shift the pattern pointer without moving the text pointer backward.
6. When the entire pattern is matched:
  - o Store the starting index.
  - o Continue searching using the LPS value.
7. Continue until the text is fully processed.

## Example Walkthrough

`text = "abcabcabc"`

`pattern = "abc"`

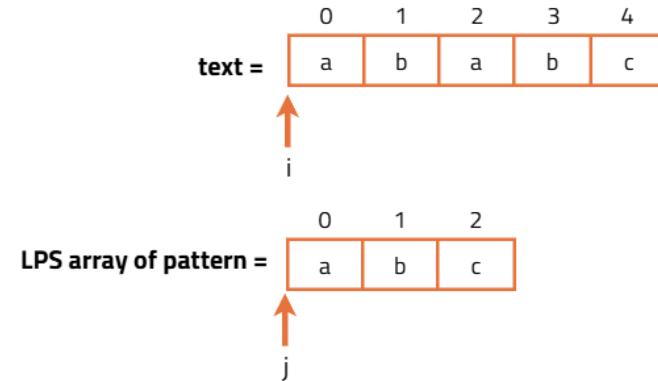
Matches found at:

- index 0
- index 3
- index 6

**text** = "ababc"  
**pattern** = "abc"

**LPS array of pattern** = 

|   |   |   |
|---|---|---|
| 0 | 1 | 2 |
| 0 | 0 | 0 |



**i** = 0, **j** = 0 → "a" = "a", increment i and j

**i** = 1, **j** = 1 → "b" = "b", increment i and j

**i** = 2, **j** = 2 → "a" ≠ "c", **j** = **LPS[j-1]** = 0

**i** = 2, **j** = 0 → "a" = "a", increment i and j

**i** = 3, **j** = 1 → "b" = "b", increment i and j

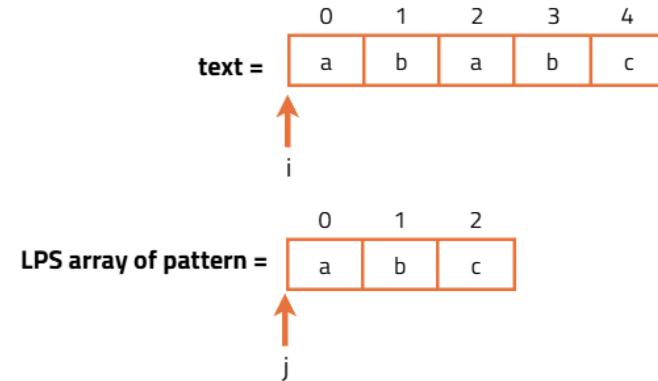
**i** = 4, **j** = 2 → "c" = "c", increment i and j

Hence, **j** = 3, a complete pattern found at  
index, **i-3** = 5-3 = 2

**text** = "ababc"  
**pattern** = "abc"

**LPS array of pattern** = 

|   |   |   |
|---|---|---|
| 0 | 1 | 2 |
| 0 | 0 | 0 |



**i** = 0, **j** = 0 → "a" = "a", increment i and j

**i** = 1, **j** = 1 → "b" = "b", increment i and j

**i** = 2, **j** = 2 → "a" ≠ "c", **j** = **LPS[j-1]** = 0

**i** = 2, **j** = 0 → "a" = "a", increment i and j

**i** = 3, **j** = 1 → "b" = "b", increment i and j

**i** = 4, **j** = 2 → "c" = "c", increment i and j

Hence, **j** = 3, a complete pattern found at  
index, **i-3** = 5-3 = 2

---

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 vector<int> computeLPS(string pattern) {
 vector<int> lps(pattern.length(), 0);
 int len = 0;
 int i = 1;

 while (i < pattern.length()) {
 if (pattern[i] == pattern[len]) {
 len++;
 lps[i] = len;
 i++;
 } else {
 if (len != 0) {
 len = lps[len - 1];
 } else {
 lps[i] = 0;
 i++;
 }
 }
 }
 return lps;
 }

 vector<int> KMP(string text, string pattern) {
 vector<int> lps = computeLPS(pattern);
 vector<int> result;

 int i = 0;
 int j = 0;

 while (i < text.length()) {
 if (text[i] == pattern[j]) {
 i++;
 j++;
 }
 }

 if (j == pattern.length()) {
```

```

 result.push_back(i - j);
 j = lps[j - 1];
 }
 else if (i < text.length() && text[i] != pattern[j]) {
 if (j != 0) {
 j = lps[j - 1];
 } else {
 i++;
 }
 }
}
return result;
}

int main() {
 string text = "abracadabra";
 string pattern = "abra";

 Solution sol;
 vector<int> indices = sol.KMP(text, pattern);

 for (int idx : indices) {
 cout << idx << " ";
 }
 return 0;
}

```

## Complexity Analysis

- **Time Complexity:**  $O(n + m)$   
LPS array construction takes  $O(m)$  and pattern matching takes  $O(n)$ .
- **Space Complexity:**  $O(m)$   
Extra space is used only for the LPS array.

# 22. Shortest Palindromic Substring

Given a string, you need to find the **shortest palindromic substring** present in it.

If there are multiple shortest palindromic substrings, return the **lexicographically smallest** one.

A palindrome is a string that reads the same forward and backward.

## Example

Input: "zyzz"

Palindromic substrings are "z", "y", "zz", "yzz y"

Shortest length is 1, lexicographically smallest is "y"

Input: "abab"

Palindromic substrings of length 1 are "a" and "b"

Lexicographically smallest is "a"

---

## Approach 1

Naive Approach (Expand Around Center)

### Algorithm

1. Every palindrome has a center.
2. For each index  $i$  in the string:
  - o Treat  $i$  as the center for **odd length** palindromes.
  - o Treat  $i-1$  and  $i$  as centers for **even length** palindromes.
3. Expand left and right while characters are equal to form palindromes.
4. Store all found palindromic substrings in a vector.
5. Ignore empty substrings.
6. From all stored palindromes, select the **shortest one**.
7. If multiple have the same length, choose the **lexicographically smallest**.

### Example Dry Run

$s = "abab"$

Centers generate palindromes: "a", "b", "a", "b", "aba", "bab"

Shortest length = 1  $\rightarrow$  "a" is lexicographically smallest.

### Code

```
#include <bits/stdc++.h>
using namespace std;
```

```

string ShortestPalindrome(string s) {
 int n = s.length();
 vector<string> v;

 for (int i = 0; i < n; i++) {
 int l = i, r = i;
 string ans1 = "", ans2 = "";

 // odd length palindrome
 while (l >= 0 && r < n && s[l] == s[r]) {
 ans1 += s[l];
 l--;
 r++;
 }

 l = i - 1;
 r = i;

 // even length palindrome
 while (l >= 0 && r < n && s[l] == s[r]) {
 ans2 += s[l];
 l--;
 r++;
 }

 v.push_back(ans1);
 v.push_back(ans2);
 }

 string ans = "";
 for (int i = 0; i < v.size(); i++) {
 if (v[i] != "") {
 if (ans == "" || v[i].length() < ans.length() ||
 (v[i].length() == ans.length() && v[i] < ans)) {
 ans = v[i];
 }
 }
 }
 return ans;
}

int main() {
 string s = "geeksforgeeks";
 cout << ShortestPalindrome(s);
}

```

```
 return 0;
}
```

## Complexity Analysis

- **Time Complexity:**  $O(N^2)$   
Each character is treated as a center and expanded in both directions.
  - **Space Complexity:**  $O(N^2)$   
All palindromic substrings can be stored in the vector in the worst case.
- 

## Approach 2

Efficient Approach (Observation-Based)

### Algorithm

1. A **single character** is always a palindrome.
2. The shortest possible palindromic substring has length 1.
3. So, the answer will always be the **lexicographically smallest character** in the string.
4. Traverse the string and track the minimum character.
5. Return that character.

### Reasoning

Any longer palindrome will have length  $\geq 2$ , which can never be shorter than a single character.  
So checking individual characters is sufficient.

### Code

```
#include <bits/stdc++.h>
using namespace std;

char ShortestPalindrome(string s) {
 char ans = s[0];
 for (int i = 1; i < s.length(); i++) {
 ans = min(ans, s[i]);
 }
 return ans;
}

int main() {
 string s = "geeksforgeeks";
 cout << ShortestPalindrome(s);
 return 0;
}
```

}

## Complexity Analysis

- **Time Complexity:**  $O(N)$   
Single traversal of the string.
- **Space Complexity:**  $O(1)$   
Only one variable is used.

# 23. Longest Happy Prefix

Given a string  $s$ , you need to find the **longest happy prefix** of the string.  
A happy prefix is a string that is **both a proper prefix and a proper suffix**.

A **proper prefix / suffix** means it must **not be equal to the entire string**.

If no such prefix exists, return an **empty string**.

### Example

Input:  $s = "ababab"$

Prefixes: "a", "ab", "aba", "abab", "ababa"

Suffixes: "b", "ab", "bab", "abab", "babab"

Longest common proper prefix and suffix is "abab"

Output: "abab"

---

## Approach 1

Brute Force Approach

### Algorithm

1. The maximum possible length of a proper prefix or suffix is  $n-1$ .
2. Start checking from length  $n-1$  down to 1.

3. For each length:
  - o Compare the prefix  $s[0 \dots len-1]$
  - o With the suffix  $s[n-len \dots n-1]$
4. If both are equal, return that prefix immediately because we are checking from longest to shortest.
5. If no match is found, return an empty string.

This approach works but performs repeated comparisons.

### Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 string longestPrefix(string s) {
 int n = s.size();

 for (int len = n - 1; len > 0; len--) {
 if (s.substr(0, len) == s.substr(n - len, len)) {
 return s.substr(0, len);
 }
 }
 return "";
 }
};

int main() {
 Solution sol;
 string s = "levellevel";
 cout << sol.longestPrefix(s);
 return 0;
}
```

### Complexity Analysis

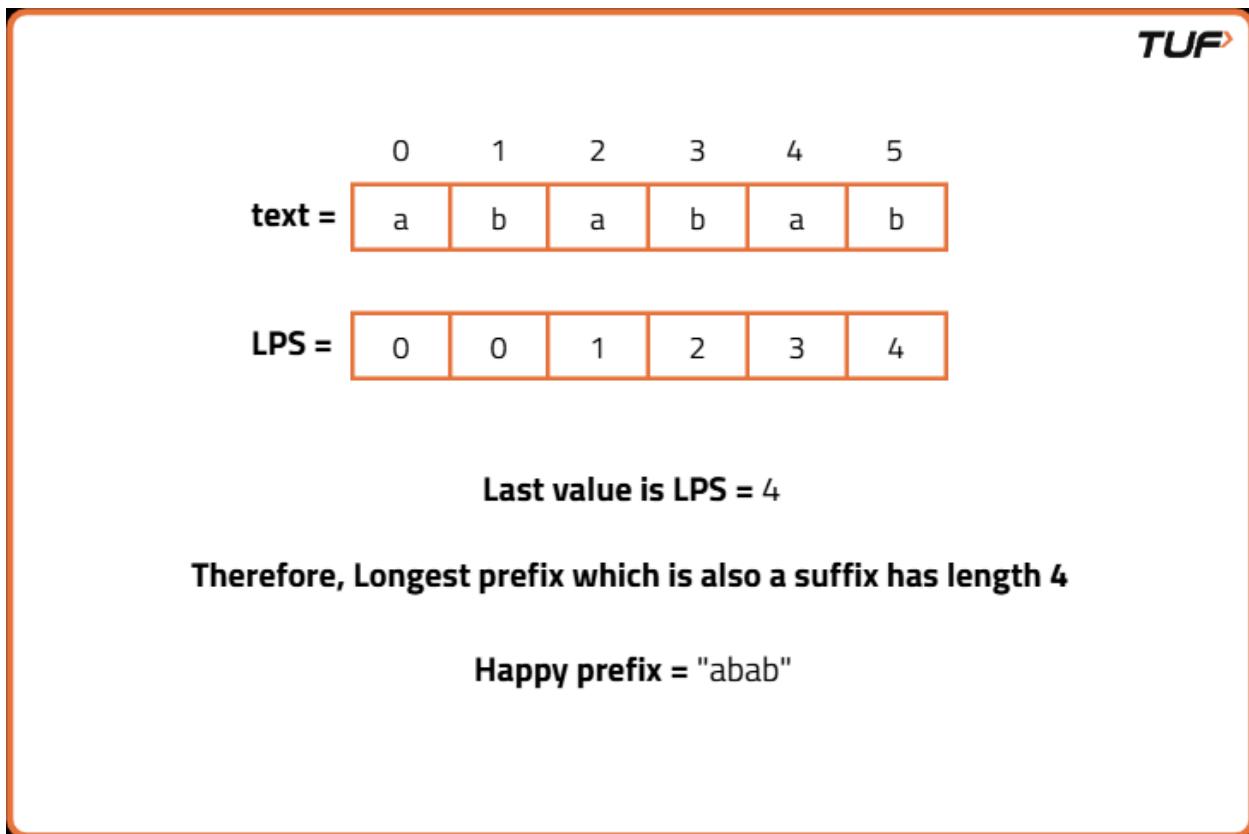
- **Time Complexity:**  $O(n^2)$   
For each possible length, substring comparison can take up to  $n$  time.
  - **Space Complexity:**  $O(n)$   
Temporary substrings are created during comparison.
-

## Approach 2

Optimal Approach (Using KMP / LPS Array)

### Algorithm

1. Build the **LPS (Longest Prefix Suffix)** array for the string.
2.  $\text{lps}[i]$  stores the length of the longest prefix that is also a suffix ending at index  $i$ .
3. Start from the second character and maintain a pointer  $\text{len}$  for the current prefix length.
4. If characters match, increase  $\text{len}$  and store it in  $\text{lps}$ .
5. If a mismatch occurs:
  - o Move  $\text{len}$  back using the LPS array instead of restarting.
6. After processing the whole string:
  - o The last value of the LPS array gives the length of the longest happy prefix.
7. Return the substring from index 0 to that length.



### Example Dry Run

s = "aaaa"

LPS array = [0, 1, 2, 3]

Last LPS value = 3

Answer = "aaa"

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 string longestPrefix(string s) {
 vector<int> lps(s.length(), 0);
 int len = 0;

 for (int i = 1; i < s.length(); i++) {
 if (s[i] == s[len]) {
 len++;
 lps[i] = len;
 } else if (len != 0) {
 len = lps[len - 1];
 i--;
 }
 }

 return s.substr(0, lps[s.length() - 1]);
 }
};

int main() {
 Solution sol;
 string s = "levellevel";
 cout << sol.longestPrefix(s);
 return 0;
}
```

## Complexity Analysis

- **Time Complexity:**  $O(n)$   
The string is processed once while building the LPS array.
- **Space Complexity:**  $O(n)$   
Extra space is used for the LPS array.

# 24. Count Palindromic Subsequence in Given String

Given a string  $s$ , you need to **count the total number of distinct palindromic subsequences** present in it.

A **palindromic subsequence** is obtained by deleting zero or more characters (without changing order) such that the resulting string reads the same forward and backward.

Only **distinct** palindromic subsequences are counted.

## Example

Input:  $s = "bccb"$

Palindromic subsequences are: "b", "c", "bb", "cc", "bcb", "bccb"

Output: 6

Input:  $s = "abcd"$

Only single characters are palindromes: "a", "b", "c", "d"

Output: 4

---

## Approach 1

Brute Force Approach

### Algorithm

1. Generate **all possible subsequences** of the string using recursion.
2. For each subsequence:
  - o Ignore empty subsequences.
  - o Check if it is a palindrome.
3. Count every palindromic subsequence found.
4. Return the total count.

This approach directly follows the definition but is very slow.

### Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
```

```

bool isPalindrome(string s) {
 int l = 0, r = s.size() - 1;
 while (l < r) {
 if (s[l++] != s[r--]) return false;
 }
 return true;
}

int count(string s, string curr, int index) {
 if (index == s.length()) {
 if (!curr.empty() && isPalindrome(curr)) return 1;
 return 0;
 }

 int include = count(s, curr + s[index], index + 1);
 int exclude = count(s, curr, index + 1);

 return include + exclude;
}

int countPalindromicSubsequences(string s) {
 return count(s, "", 0);
}
};

int main() {
 Solution sol;
 string s = "abcb";
 cout << sol.countPalindromicSubsequences(s);
 return 0;
}

```

## Complexity Analysis

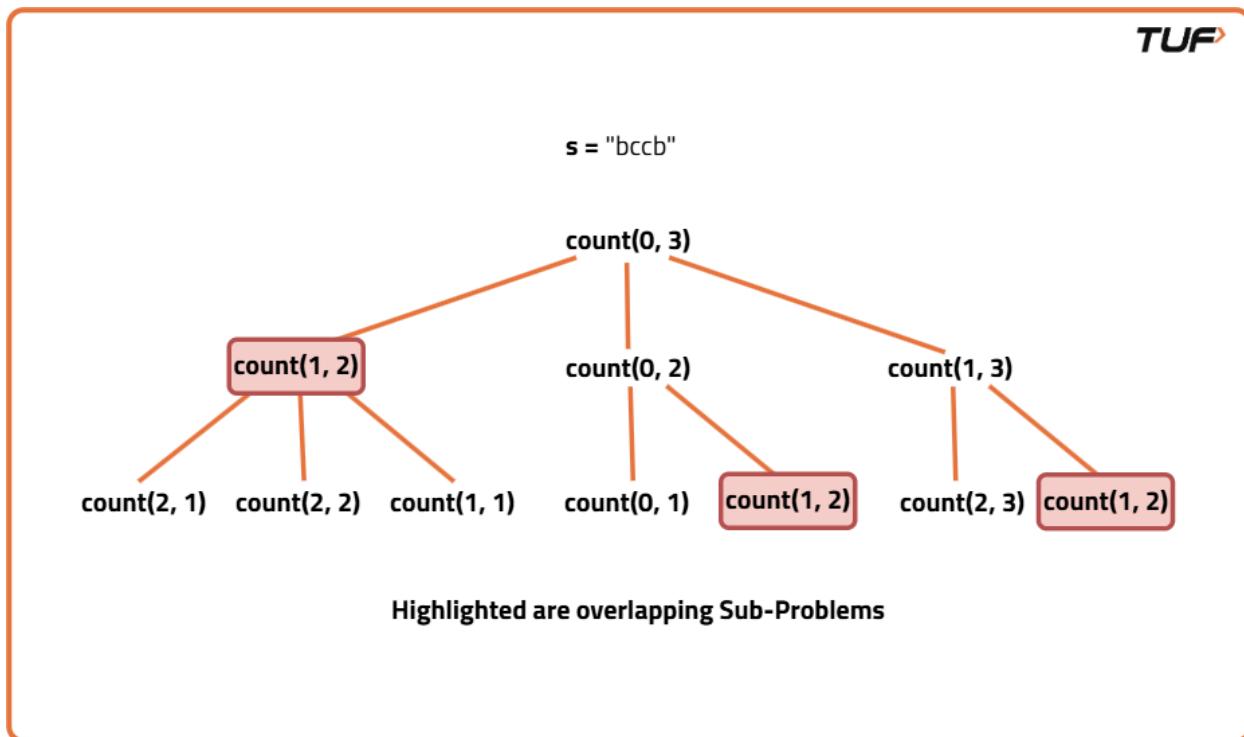
- **Time Complexity:**  $O(2^n \cdot n)$   
All subsequences are generated and each is checked for palindrome.
  - **Space Complexity:**  $O(n)$   
Recursion stack and temporary strings.
- 

## Approach 2

Optimal Approach – Memoization

## Algorithm

1. Use recursion on substring ranges  $[i \dots j]$ .
  2. Base cases:
    - o If  $i > j$ , return 0.
    - o If  $i == j$ , return 1 (single character).
  3. If result already computed, return it.
  4. If  $s[i] == s[j]$ :
    - o Count subsequences excluding first
    - o Count subsequences excluding last
    - o Add 1 for the new palindrome formed by  $s[i]$  and  $s[j]$
  5. If  $s[i] != s[j]$ :
    - o Add counts from excluding first and last
    - o Subtract overlapping part to avoid double count
  6. Store result in DP table.



## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int countPalindromicSubsequences(string s) {
```

```

int n = s.size();
vector<vector<int>> dp(n, vector<int>(n, -1));
return count(s, 0, n - 1, dp);
}

private:
int count(string &s, int i, int j, vector<vector<int>> &dp) {
 if (i > j) return 0;
 if (i == j) return 1;
 if (dp[i][j] != -1) return dp[i][j];

 if (s[i] == s[j])
 dp[i][j] = count(s, i + 1, j, dp) +
 count(s, i, j - 1, dp) + 1;
 else
 dp[i][j] = count(s, i + 1, j, dp) +
 count(s, i, j - 1, dp) -
 count(s, i + 1, j - 1, dp);

 return dp[i][j];
}
};

int main() {
 Solution sol;
 string s = "aab";
 cout << sol.countPalindromicSubsequences(s);
 return 0;
}

```

## Complexity Analysis

- **Time Complexity:**  $O(n^2)$   
Each substring  $[i...j]$  is solved once.
  - **Space Complexity:**  $O(n^2 + n)$   
DP table + recursion stack.
- 

## Approach 3

Optimal Approach – Tabulation

### Algorithm

1. Create a 2D DP table  $dp[i][j]$  to store number of palindromic subsequences in substring  $[i..j]$ .
2. Initialize:
  - o  $dp[i][i] = 1$  (single characters).
3. Consider substrings of increasing length:
  - o If  $s[i] == s[j]$ :  
 $dp[i][j] = dp[i+1][j] + dp[i][j-1] + 1$
  - o Else:  
 $dp[i][j] = dp[i+1][j] + dp[i][j-1] - dp[i+1][j-1]$
4. Final answer is  $dp[0][n-1]$ .

This avoids recursion and is fully iterative.

### Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int countPalindromicSubsequences(string s) {
 int n = s.size();
 vector<vector<int>> dp(n, vector<int>(n, 0));

 for (int i = 0; i < n; i++) {
 dp[i][i] = 1;
 }

 for (int len = 2; len <= n; len++) {
 for (int i = 0; i <= n - len; i++) {
 int j = i + len - 1;

 if (s[i] == s[j])
 dp[i][j] = dp[i + 1][j] + dp[i][j - 1] + 1;
 else
 dp[i][j] = dp[i + 1][j] + dp[i][j - 1] -
 dp[i + 1][j - 1];
 }
 }
 return dp[0][n - 1];
 }
};

int main() {
```

```
Solution sol;
string s = "aab";
cout << sol.countPalindromicSubsequences(s);
return 0;
}
```

## Complexity Analysis

- **Time Complexity:**  $O(n^2)$   
All substrings are processed once.
- **Space Complexity:**  $O(n^2)$   
DP table stores results for all substrings.

# **LinkedList**

# 1. Insert at the Head of a Linked List

You are given the head of a singly linked list and an integer value `val`.

Your task is to **insert a new node with value `val` at the beginning of the linked list** (before the current head) and return the **updated head** of the list.

---

## Examples

### Example 1

Input: `0 -> 1 -> 2 , val = 5`

Output: `5 -> 0 -> 1 -> 2`

### Example 2

Input: `12 -> 5 -> 8 -> 7 , val = 4`

Output: `4 -> 12 -> 5 -> 8 -> 7`

---

## Key Idea

In a linked list, the **head is just a pointer** to the first node.

To insert a node **before the head**, we:

1. Create a new node
2. Point its next to the current head
3. Make this new node the new head

No traversal is required.

---

## Algorithm (Step-by-Step)

1. Create a new node with data = val
  2. Set newNode->next to the current head
  3. Return newNode as the updated head
- 

## Dry Run

### Initial List

head → 2 → 3 → NULL

### Insert value 1 at head

#### Step 1: Create new node

newNode = 1

#### Step 2: Point new node to old head

1 → 2 → 3 → NULL

#### Step 3: Update head

head = newNode

Final List:

1 → 2 → 3 → NULL

---

## C++ Code

```
#include <bits/stdc++.h>
using namespace std;

// Node class to represent each node in the linked list
class Node {
public:
 // Data stored in the node
 int data;
```

```

// Pointer to the next node
Node* next;

// Constructor with data and next pointer
Node(int data1, Node* next1) {
 data = data1;
 next = next1;
}

// Constructor with only data
Node(int data1) {
 data = data1;
 next = nullptr;
}
};

// Solution class to handle linked list operations
class Solution {
public:
 // Function to insert a new node at the head
 Node* insertAtHead(Node* head, int newData) {
 // Create a new node whose next points to current head
 Node* newNode = new Node(newData, head);
 // Return the new node as the head
 return newNode;
 }

 // Function to print the linked list
 void printList(Node* head) {
 Node* temp = head;
 while (temp != nullptr) {
 cout << temp->data << " ";
 temp = temp->next;
 }
 cout << endl;
 }
};

```

```

int main() {
 Solution sol;

 // Creating a sample linked list: 2 -> 3
 Node* head = new Node(2);
 head->next = new Node(3);

 cout << "Original List: ";
 sol.printList(head);

 // Inserting new node at head
 head = sol.insertAtHead(head, 1);

 cout << "After Insertion at Head: ";
 sol.printList(head);

 return 0;
}

```

---

## Edge Case: Empty List

If the list is empty (`head == NULL`):

- The new node's next will be `NULL`
- The new node itself becomes the head

Works correctly without any special handling.

---

## Complexity Analysis

| Type | Complexity |
|------|------------|
|------|------------|

Time Complexity     $O(1)$

Space Complexity     $O(1)$

Only one node is created and no traversal is done.

---

## 2. Delete Last Node of a Linked List

You are given the head of a **singly linked list**.

Your task is to **delete the last node (tail)** of the linked list and return the **updated head** of the list.

---

### Examples

#### Example 1

Input:  $0 \rightarrow 1 \rightarrow 2$

Output:  $0 \rightarrow 1$

#### Explanation:

The last node is 2, so it is removed.

---

#### Example 2

Input:  $12 \rightarrow 5 \rightarrow 8 \rightarrow 7$

Output:  $12 \rightarrow 5 \rightarrow 8$

#### Explanation:

The last node is 7, so it is removed.

---

## Key Idea

In a **singly linked list**, each node only knows about the **next node**.

To delete the last node:

- We must reach the **second last node**
  - Set its next pointer to NULL
  - Delete the old last node
- 

## Important Edge Cases

### ① Empty list (`head == NULL`)

→ Nothing to delete, return NULL

### ② Only one node in the list

→ Delete that node and return NULL

---

## Algorithm (Step-by-Step)

1. If the list is empty or has only one node:
    - Delete the node (if any)
    - Return NULL
  2. Otherwise:
    - Traverse the list until you reach the **second last node**
    - Delete the last node
    - Set `secondLast->next = NULL`
    - Return the head
-

# Dry Run

## Initial List

```
head → 1 → 2 → 3 → NULL
```

## Traversal

We stop when:

```
curr->next->next == NULL
```

So curr points to:

```
1 → 2 → 3 → NULL
```

↑

curr

## Delete last node

```
delete curr->next; // deletes node 3
curr->next = NULL;
```

## Final List

```
1 → 2 → NULL
```

---

## C++ Code

```
#include <bits/stdc++.h>

using namespace std;

// Definition for singly linked list

struct Node {
 int data;
 Node* next;
 Node(int val) {
 data = val;
 next = NULL;
 }
};

class Solution {

public:

 // Function to delete tail node of linked list

 Node* deleteTail(Node* head) {
 // If list is empty or has one node
 if (head == NULL || head->next == NULL) {
 delete head;
 return NULL;
 }
 }
}
```

```

 }

 // Traverse to the second last node

 Node* curr = head;

 while (curr->next->next != NULL) {

 curr = curr->next;

 }

 // Delete tail node

 delete curr->next;

 curr->next = NULL;

 // Return updated head

 return head;

}

};

// Driver code

int main() {

 Node* head = new Node(1);

 head->next = new Node(2);

 head->next->next = new Node(3);
}

```

```
Solution obj;

head = obj.deleteTail(head);

// Print list after deletion

Node* temp = head;

while (temp) {

 cout << temp->data << " ";

 temp = temp->next;

}

return 0;

}
```

---

## Complexity Analysis

| Type | Complexity |
|------|------------|
|------|------------|

|                 |        |
|-----------------|--------|
| Time Complexity | $O(N)$ |
|-----------------|--------|

|                  |        |
|------------------|--------|
| Space Complexity | $O(1)$ |
|------------------|--------|

---

# 3. Find the Length of a Linked List

You are given the **head** of a singly linked list.

Your task is to **find and return the total number of nodes** present in the linked list.

---

## Example 1

Input:  $0 \rightarrow 1 \rightarrow 2$

Output: 3

### Explanation:

There are three nodes in the list, so the length is 3.

---

## Example 2

Input:  $12 \rightarrow 5 \rightarrow 8 \rightarrow 7$

Output: 4

### Explanation:

There are four nodes in the list, so the length is 4.

---

## Key Idea

A linked list is made up of nodes connected using next pointers.

To find its length:

- Start from the **head**
- Move node by node using **next**
- Count how many nodes you visit until you reach **NULL**

---

## Algorithm (Step-by-Step)

1. Initialize a counter count = 0
  2. Create a temporary pointer temp and point it to head
  3. Traverse the list while temp is not NULL
    - o Increment count
    - o Move temp to temp->next
  4. When traversal ends, return count
- 

## Dry Run

### Linked List

head → 10 → 20 → 30 → NULL

### Traversal

| temp points to | coun |
|----------------|------|
|----------------|------|

|    |   |
|----|---|
| 10 | 1 |
|----|---|

|    |   |
|----|---|
| 20 | 2 |
|----|---|

|    |   |
|----|---|
| 30 | 3 |
|----|---|

|      |      |
|------|------|
| NULL | stop |
|------|------|

## Final Answer

Length = 3

---

## C++ Code

```
#include <bits/stdc++.h>

using namespace std;

// Node class to represent each element in the linked list

class Node {

public:

 int data;

 Node* next;

 // Constructor to initialize data and next pointer

 Node(int data1) {

 data = data1;

 next = nullptr;

 }

};
```

```
// Solution class containing the method to find length

class Solution {

public:

 // Function to find the length of the linked list

 int lengthOfLinkedList(Node* head) {

 // Initialize counter to 0

 int count = 0;

 // Initialize a temporary pointer to head

 Node* temp = head;

 // Traverse the linked list

 while (temp != nullptr) {

 // Increment count for each node

 count++;

 // Move to the next node

 temp = temp->next;

 }

 // Return the total count

 return count;

 }

}
```

```

};

int main() {

 // Creating a sample linked list

 Node* head = new Node(10);

 head->next = new Node(20);

 head->next->next = new Node(30);

 // Create Solution object

 Solution obj;

 // Find and print the length of linked list

 cout << "Length of Linked List: "

 << obj.lengthOfLinkedList(head) << endl;

 return 0;
}

```

---

## Edge Cases

### ① Empty list

head = NULL

- Loop does not run
- count = 0
- Correct answer

## 2 Single node

head → X → NULL

- Loop runs once
  - count = 1
- 

## Complexity Analysis

| Type | Complexity |
|------|------------|
|------|------------|

|                 |      |
|-----------------|------|
| Time Complexity | O(N) |
|-----------------|------|

|                  |      |
|------------------|------|
| Space Complexity | O(1) |
|------------------|------|

## 4. Search an Element in a Linked List

You are given the head of a singly linked list and an integer value.  
Check whether the given value exists in the linked list.

**Idea:**

- Traverse the linked list from the head node.
- At each node, compare its data with the given value.
- If a match is found, return true.
- If the traversal ends without a match, return false.

```

#include <bits/stdc++.h>
using namespace std;

class Node {
public:
 int data;
 Node* next;

 Node(int val) {
 data = val;
 next = nullptr;
 }
};

class Solution {
public:
 bool searchValue(Node* head, int val) {
 Node* temp = head;
 while (temp != nullptr) {
 if (temp->data == val)
 return true;
 temp = temp->next;
 }
 return false;
 }
};

int main() {
 Node* head = new Node(10);
 head->next = new Node(20);
 head->next->next = new Node(30);

 Solution sol;
 int val = 20;

 cout << (sol.searchValue(head, val) ? "True" : "False");
 return 0;
}

```

**Complexity:**

Time: O(N)

Space: O(1)

## 5. Introduction to Doubly Linked List

Before a doubly linked list, recall a **singly linked list**.

In a singly linked list, each node stores:

- data
- a pointer to the next node

Traversal is possible only in the forward direction. Moving backward is not possible because there is no reference to the previous node.

---

### Singly Linked List Node Structure

```
#include <bits/stdc++.h>
using namespace std;

class Node {
public:
 int data;
 Node* next;

 Node(int data1, Node* next1) {
 data = data1;
 next = next1;
 }

 Node(int data1) {
 data = data1;
 next = nullptr;
 }
};
```

```
int main() {
 vector<int> arr = {2, 5, 8, 7};

 Node* head = new Node(arr[0]);

 cout << head << endl;
 cout << head->data << endl;

 return 0;
}
```

---

## Doubly Linked List

A doubly linked list improves upon a singly linked list by allowing **two-way traversal**.  
Each node stores:

- data
- pointer to the next node
- pointer to the previous node

Because of the prev pointer, traversal is possible both forward and backward.

---

## Doubly Linked List Node Structure

```
#include <bits/stdc++.h>
using namespace std;

class Node {
public:
 int data;
 Node* next;
 Node* prev;

 Node(int data1, Node* next1, Node* prev1) {
```

```

 data = data1;
 next = next1;
 prev = prev1;
 }

Node(int data1) {
 data = data1;
 next = nullptr;
 prev = nullptr;
}
};

int main() {
 vector<int> arr = {2, 5, 8, 7};

 Node* head = new Node(arr[0]);

 cout << head << endl;
 cout << head->data << endl;

 return 0;
}

```

---

**Key Difference:**

Singly Linked List → only next pointer

Doubly Linked List → both next and prev pointers

## 6. Insert at End of Doubly Linked List

You are given the head of a doubly linked list and a value k.

Insert a new node with value k at the end of the list and return the head.

**Idea:**

If the list is empty, the new node becomes the head.  
Otherwise, traverse to the last node (tail).  
Link the new node after the tail and update both next and prev pointers.

```
#include <bits/stdc++.h>
using namespace std;

class Node {
public:
 int data;
 Node* next;
 Node* prev;

 Node(int val) {
 data = val;
 next = nullptr;
 prev = nullptr;
 }
};

Node* insertAtEnd(Node* head, int k) {
 Node* newNode = new Node(k);

 if (head == nullptr)
 return newNode;

 Node* tail = head;
 while (tail->next != nullptr) {
 tail = tail->next;
 }

 tail->next = newNode;
 newNode->prev = tail;

 return head;
}

void printList(Node* head) {
```

```

 while (head != nullptr) {
 cout << head->data << " ";
 head = head->next;
 }
 }

int main() {
 Node* head = new Node(1);
 head->next = new Node(2);
 head->next->prev = head;
 head->next->next = new Node(3);
 head->next->next->prev = head->next;

 head = insertAtEnd(head, 6);
 printList(head);

 return 0;
}

```

**Complexity:**

Time: O(N)

Space: O(1)

## 7. Delete Last Node of a Doubly Linked List

You are given the head of a doubly linked list.

Delete the last node (tail) of the list and return the updated head.

**Idea:**

If the list is empty, nothing to delete.

If the list has only one node, delete it and return NULL.

Otherwise, traverse to the last node using next.

Use the prev pointer to reach the second last node.  
Update the second last node's next to NULL and delete the tail.

```
#include <bits/stdc++.h>
using namespace std;

class Node {
public:
 int data;
 Node* prev;
 Node* next;

 Node(int val) {
 data = val;
 prev = nullptr;
 next = nullptr;
 }
};

Node* deleteLastNode(Node* head) {
 if (head == nullptr)
 return nullptr;

 if (head->next == nullptr) {
 delete head;
 return nullptr;
 }

 Node* tail = head;
 while (tail->next != nullptr) {
 tail = tail->next;
 }

 Node* secondLast = tail->prev;
 secondLast->next = nullptr;
 delete tail;

 return head;
}
```

```

void printList(Node* head) {
 while (head != nullptr) {
 cout << head->data << " ";
 head = head->next;
 }
}

int main() {
 Node* head = new Node(1);
 head->next = new Node(3);
 head->next->prev = head;
 head->next->next = new Node(4);
 head->next->next->prev = head->next;
 head->next->next->next = new Node(1);
 head->next->next->next->prev = head->next->next;

 head = deleteLastNode(head);
 printList(head);

 return 0;
}

```

### **Complexity:**

Time: O(N)

Space: O(1)

# 8. Reverse a Doubly Linked List

You are given the head of a doubly linked list. Reverse the list and return the new head.

---

## Brute Force Approach

**Idea:**

Store all node values in a stack.

Traverse the list again and replace node values using the stack (LIFO gives reverse order).

```
#include <bits/stdc++.h>
using namespace std;

class Node {
public:
 int data;
 Node* next;
 Node* prev;
 Node(int val) {
 data = val;
 next = nullptr;
 prev = nullptr;
 }
};

Node* reverseDLL(Node* head) {
 if (head == nullptr || head->next == nullptr)
 return head;

 stack<int> st;
 Node* temp = head;

 while (temp != nullptr) {
 st.push(temp->data);
 temp = temp->next;
 }

 temp = head;
 while (temp->next != nullptr) {
 temp->data = st.top();
 st.pop();
 temp = temp->next;
 }

 temp->data = st.top();
 st.pop();

 return head;
}
```

```

 temp = head;
 while (temp != nullptr) {
 temp->data = st.top();
 st.pop();
 temp = temp->next;
 }

 return head;
 }

void printList(Node* head) {
 while (head != nullptr) {
 cout << head->data << " ";
 head = head->next;
 }
}

int main() {
 Node* head = new Node(1);
 head->next = new Node(2);
 head->next->prev = head;
 head->next->next = new Node(3);
 head->next->next->prev = head->next;

 head = reverseDLL(head);
 printList(head);
 return 0;
}

```

### **Complexity:**

Time: O(N)

Space: O(N)

---

## **Optimal Approach**

### **Idea:**

Reverse the list by swapping next and prev pointers for every node.

While traversing, keep updating the head to the current node.  
At the end, the last processed node becomes the new head.

```
#include <bits/stdc++.h>
using namespace std;

class Node {
public:
 int data;
 Node* next;
 Node* prev;
 Node(int val) {
 data = val;
 next = nullptr;
 prev = nullptr; }
};

Node* reverseDLL(Node* head) {
 if (head == nullptr || head->next == nullptr)
 return head;

 Node* curr = head;

 while (curr != nullptr) {
 Node* temp = curr->next;
 curr->next = curr->prev;
 curr->prev = temp;
 head = curr;
 curr = temp;
 }

 return head;
}

void printList(Node* head) {
 while (head != nullptr) {
 cout << head->data << " ";
 head = head->next;
 }
}
```

```

}

int main() {
 Node* head = new Node(10);
 head->next = new Node(20);
 head->next->prev = head;
 head->next->next = new Node(30);
 head->next->next->prev = head->next;

 head = reverseDLL(head);
 printList(head);
 return 0;
}

```

**Complexity:**

Time: O(N)

Space: O(1)

## 9. Find Middle Element in a Linked List

You are given the head of a linked list.

If the list has **odd nodes**, return the middle node.

If the list has **even nodes**, return the **second middle node**.

### Brute Force Approach

**Idea:**

First count total nodes.

Then move to (count / 2) steps from head (0-indexed), which gives the middle.

```
#include <bits/stdc++.h>
```

```
using namespace std;

class Node {
public:
 int data;
 Node* next;
 Node(int val) {
 data = val;
 next = nullptr;
 }
};

Node* findMiddle(Node* head) {
 if (head == nullptr) return nullptr;

 int count = 0;
 Node* temp = head;

 while (temp != nullptr) {
 count++;
 temp = temp->next;
 }
}
```

```

int mid = count / 2;

temp = head;

while (mid--) {

 temp = temp->next;

}

return temp;

}

int main() {

Node* head = new Node(1);

head->next = new Node(2);

head->next->next = new Node(3);

head->next->next->next = new Node(4);

head->next->next->next->next = new Node(5);

Node* ans = findMiddle(head);

cout << ans->data;

return 0;

}

```

### **Complexity:**

Time: O(N)

Space: O(1)

---

## **Optimal Approach (Tortoise & Hare)**

### **Idea:**

Use two pointers:

slow moves 1 step, fast moves 2 steps.

When fast reaches the end, slow is at the middle.

For even length, this automatically gives the **second middle**.

```
#include <bits/stdc++.h>

using namespace std;

class Node {
public:
 int data;
 Node* next;
 Node(int val) {
 data = val;
 next = nullptr;
 }
};

Node* findMiddle(Node* head) {
 if (head == nullptr) return nullptr;
```

```

Node* slow = head;

Node* fast = head;

while (fast != nullptr && fast->next != nullptr) {

 slow = slow->next;

 fast = fast->next->next;

}

return slow;
}

int main() {

Node* head = new Node(1);

head->next = new Node(2);

head->next->next = new Node(3);

head->next->next->next = new Node(4);

head->next->next->next->next = new Node(5);

head->next->next->next->next->next = new Node(6);

Node* ans = findMiddle(head);

cout << ans->data;

return 0;
}

```

**Complexity:**

Time: O(N)

Space: O(1)

# 10. Reverse a Linked List

You are given the head of a **singly linked list**.

Reverse the linked list and return the new head.

---

## Brute Force Approach (Using Stack)

**Idea:**

Store node values in a stack, then overwrite the list in reverse order.

```
#include <bits/stdc++.h>
using namespace std;

struct ListNode {
 int val;
 ListNode* next;
 ListNode(int x) {
 val = x;
 next = NULL;
 }
};

ListNode* reverseList(ListNode* head) {
 stack<int> st;
 ListNode* temp = head;

 while (temp != NULL) {
 st.push(temp->val);
 temp = temp->next;
 }

 temp = head;
 while (temp->next != NULL) {
 temp->val = st.top();
 st.pop();
 temp = temp->next;
 }

 temp->val = st.top();
 st.pop();
}
```

```

 st.push(temp->val);
 temp = temp->next;
 }

 temp = head;
 while (temp != NULL) {
 temp->val = st.top();
 st.pop();
 temp = temp->next;
 }

 return head;
}

int main() {
 ListNode* head = new ListNode(1);
 head->next = new ListNode(2);
 head->next->next = new ListNode(3);

 head = reverseList(head);

 while (head != NULL) {
 cout << head->val << " ";
 head = head->next;
 }
 return 0;
}

```

### **Complexity:**

Time: O(N)

Space: O(N)

---

## **Optimal Approach (Iterative Pointer Reversal)**

### **Idea:**

Reverse links by changing next pointers using three pointers.

```

#include <bits/stdc++.h>
using namespace std;

class ListNode {
public:
 int val;
 ListNode* next;
 ListNode(int x) {
 val = x;
 next = NULL;
 }
};

ListNode* reverseList(ListNode* head) {
 ListNode* prev = NULL;
 ListNode* curr = head;

 while (curr != NULL) {
 ListNode* nextNode = curr->next;
 curr->next = prev;
 prev = curr;
 curr = nextNode;
 }

 return prev;
}

int main() {
 ListNode* head = new ListNode(1);
 head->next = new ListNode(2);
 head->next->next = new ListNode(3);
 head->next->next->next = new ListNode(4);

 head = reverseList(head);

 while (head != NULL) {
 cout << head->val << " ";
 head = head->next;
 }
}

```

```

 }
 return 0;
}

```

### Complexity:

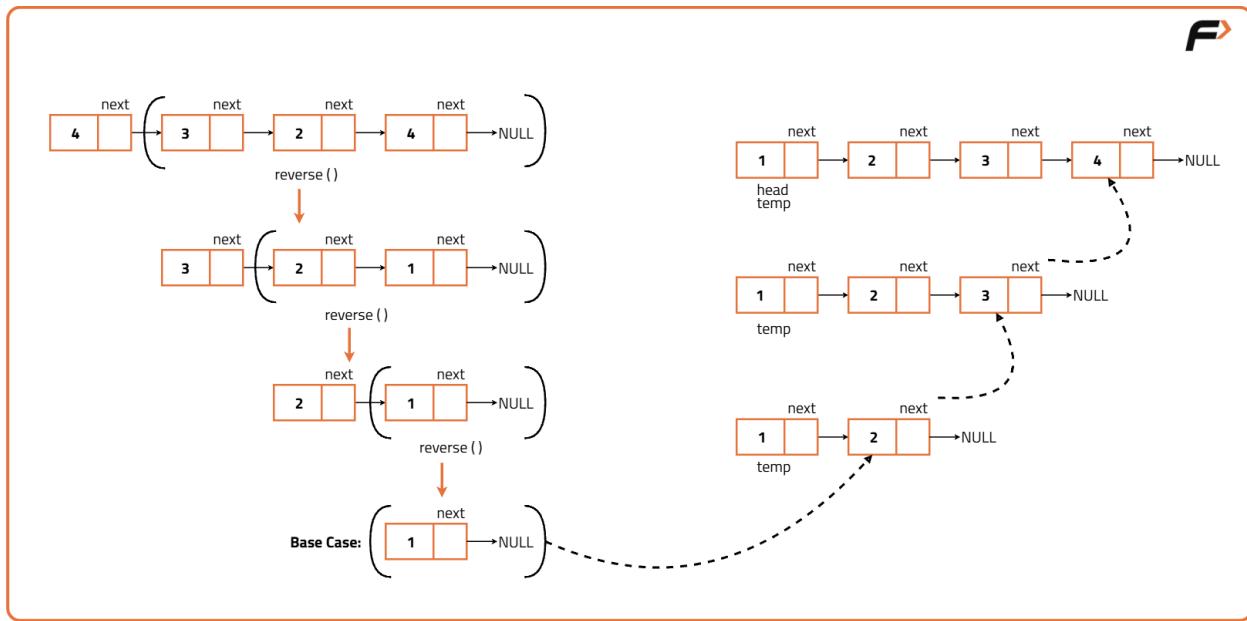
Time: O(N)

Space: O(1)

## Recursive Approach

### Idea:

Reverse rest of the list, then fix current node links while returning.



```

#include <bits/stdc++.h>
using namespace std;

struct ListNode {
 int val;
 ListNode* next;
 ListNode(int x) {
 val = x;
 }
};

```

```

 next = NULL;
 }
};

ListNode* reverseList(ListNode* head) {
 if (head == NULL || head->next == NULL)
 return head;

 ListNode* newHead = reverseList(head->next);

 head->next->next = head;
 head->next = NULL;

 return newHead;
}

int main() {
 ListNode* head = new ListNode(1);
 head->next = new ListNode(2);
 head->next->next = new ListNode(3);
 head->next->next->next = new ListNode(4);

 head = reverseList(head);

 while (head != NULL) {
 cout << head->val << " ";
 head = head->next;
 }
 return 0;
}

```

### **Complexity:**

Time: O(N)

Space: O(N) (recursion stack)

# 11. Detect a Cycle in a Linked List

You are given the head of a singly linked list.

Return **true** if the linked list contains a cycle, otherwise return **false**.

---

## Brute Force Approach (Using Hashing)

### Idea:

Keep track of visited nodes.

If a node is visited again, a cycle exists.

```
#include <bits/stdc++.h>
using namespace std;

class Node {
public:
 int data;
 Node* next;
 Node(int data1) {
 data = data1;
 next = NULL;
 }
};

bool detectCycle(Node* head) {
 unordered_set<Node*> visited;
 Node* temp = head;

 while (temp != NULL) {
 if (visited.count(temp))
 return true;
 visited.insert(temp);
 temp = temp->next;
 }
 return false;
}
```

```

int main() {
 Node* head = new Node(1);
 Node* n2 = new Node(2);
 Node* n3 = new Node(3);
 Node* n4 = new Node(4);

 head->next = n2;
 n2->next = n3;
 n3->next = n4;
 n4->next = n2; // cycle

 cout << detectCycle(head);
 return 0;
}

```

### **Complexity:**

Time:  $O(N)$

Space:  $O(N)$

---

## **Optimal Approach (Floyd's Cycle Detection)**

### **Idea:**

Use two pointers:

- slow moves one step
  - fast moves two steps
- If they meet, a cycle exists.

```

#include <bits/stdc++.h>
using namespace std;

class Node {
public:
 int data;
 Node* next;
 Node(int data1) {
 data = data1;
 }
}

```

```

 next = NULL;
 }
};

bool detectCycle(Node* head) {
 Node* slow = head;
 Node* fast = head;

 while (fast != NULL && fast->next != NULL) {
 slow = slow->next;
 fast = fast->next->next;
 if (slow == fast)
 return true;
 }
 return false;
}

int main() {
 Node* head = new Node(1);
 Node* n2 = new Node(2);
 Node* n3 = new Node(3);
 Node* n4 = new Node(4);

 head->next = n2;
 n2->next = n3;
 n3->next = n4;
 n4->next = n2; // cycle

 cout << detectCycle(head);
 return 0;
}

```

### **Complexity:**

Time: O(N)

Space: O(1)

# 12. Starting Point of Loop in a Linked List

Given the head of a singly linked list, return the node where the cycle begins.  
If there is no cycle, return NULL.

---

## Brute Force Approach (Hashing)

### Idea:

The first node that is visited again during traversal is the starting point of the loop.

```
#include <bits/stdc++.h>
using namespace std;

class ListNode {
public:
 int val;
 ListNode* next;
 ListNode(int x) {
 val = x;
 next = NULL;
 }
};

ListNode* detectCycle(ListNode* head) {
 unordered_set<ListNode*> visited;
 ListNode* temp = head;

 while (temp != NULL) {
 if (visited.count(temp))
 return temp;
 visited.insert(temp);
 temp = temp->next;
 }
}
```

```

 }
 return NULL;
}

int main() {
 ListNode* head = new ListNode(1);
 head->next = new ListNode(2);
 head->next->next = new ListNode(3);
 head->next->next->next = new ListNode(4);
 head->next->next->next->next = new ListNode(5);

 head->next->next->next->next->next = head->next->next; // loop at
3

 ListNode* ans = detectCycle(head);
 if (ans) cout << ans->val;
 else cout << "NULL";
 return 0;
}

```

Time Complexity: O(N)

Space Complexity: O(N)

---

## Optimal Approach (Floyd's Cycle Detection)

### Idea:

First detect the cycle using slow and fast pointers.

After they meet, reset one pointer to head.

Move both one step at a time — the meeting point is the start of the loop.

```

#include <bits/stdc++.h>
using namespace std;

class ListNode {
public:
 int val;
 ListNode* next;
 ListNode(int x) {

```

```

 val = x;
 next = NULL;
 }
};

ListNode* detectCycle(ListNode* head) {
 ListNode* slow = head;
 ListNode* fast = head;

 while (fast != NULL && fast->next != NULL) {
 slow = slow->next;
 fast = fast->next->next;

 if (slow == fast) {
 slow = head;
 while (slow != fast) {
 slow = slow->next;
 fast = fast->next;
 }
 return slow;
 }
 }
 return NULL;
}

int main() {
 ListNode* head = new ListNode(1);
 head->next = new ListNode(2);
 head->next->next = new ListNode(3);
 head->next->next->next = new ListNode(4);
 head->next->next->next->next = new ListNode(5);

 head->next->next->next->next->next = head->next->next; // loop at
3

 ListNode* ans = detectCycle(head);
 if (ans) cout << ans->val;
 else cout << "NULL";
}

```

```
 return 0;
}
```

Time Complexity: O(N)

Space Complexity: O(1)

## Calculation:

$x$  = head se loop start tak distance

$c$  = loop ka length

$y$  = loop start se meeting point tak distance

## Step 1: Slow aur Fast ne kitna distance chala?

### Slow pointer

- Head se chala
- Loop me entry ki
- Meeting point tak gaya

Slow distance =  $x + y$

Numbers daalo:

Slow = 2 + 2 = 4

---

## Fast pointer

- Slow se **2x speed**
- Loop ke andar **extra rounds** laga sakta hai

Isliye:

$$\text{Fast distance} = x + y + k*c$$

Yahan:

- $k$  = loop ke extra rounds
- 

## Step 2: Floyd ka rule use karo

$$\text{Fast} = 2 \times \text{Slow}$$

$$x + y + k*c = 2(x + y)$$

---

## Step 3: Equation solve karo

$$x + y + k*c = 2x + 2y$$

Left se  $x + y$  minus karo:

$$k*c = x + y$$

---

## Step 4: x ke liye likho

$$x = k*c - y$$

---

## Step 5: Numbers daal ke check karo

```
x = 1*4 - 2
x = 2
```

✓ Match karta hai (head → loop start = 2)

---

## Step 6: Iska matlab kya nikla?

```
x = k*c - y
```

Matlab:

- Head se loop start tak distance = **x**
- Meeting point se loop start tak distance:
  - Loop me aage jaate hue =  $c - y$
  - Aur agar extra rounds chahiye →  $k*c - y$

👉 Dono distance **same** hain

# 13. Length of Loop in Linked List

Given the head of a singly linked list, return the **length of the loop** if a cycle exists.  
If there is **no loop**, return **0**.

---

## Brute Force Approach (Hashing + Timer)

### Idea:

Store each visited node with the step number at which it was visited.  
When a node is visited again, the difference in steps gives the loop length.

```
#include <bits/stdc++.h>
using namespace std;

class Node {
public:
 int data;
 Node* next;
 Node(int data1) {
 data = data1;
 next = NULL;
 }
};

int lengthOfLoop(Node* head) {
 unordered_map<Node*, int> visited;
 int timer = 0;
 Node* temp = head;

 while (temp != NULL) {
 if (visited.count(temp)) {
 return timer - visited[temp];
 }
 visited[temp] = timer;
 timer++;
 temp = temp->next;
 }
 return 0;
}

int main() {
 Node* head = new Node(1);
 Node* n2 = new Node(2);
 Node* n3 = new Node(3);
```

```

Node* n4 = new Node(4);
Node* n5 = new Node(5);

head->next = n2;
n2->next = n3;
n3->next = n4;
n4->next = n5;
n5->next = n3; // loop of length 3

cout << lengthOfLoop(head);
return 0;
}

```

Time Complexity: O(N)

Space Complexity: O(N)

---

## Optimal Approach (Floyd's Cycle Detection)

### Idea:

First detect the loop using slow and fast pointers.

Once they meet, move one pointer around the loop until it comes back to the same node, counting steps.

```

#include <bits/stdc++.h>
using namespace std;

class Node {
public:
 int data;
 Node* next;
 Node(int data1) {
 data = data1;
 next = NULL;
 }
};

int lengthOfLoop(Node* head) {
 Node* slow = head;

```

```

Node* fast = head;

while (fast != NULL && fast->next != NULL) {
 slow = slow->next;
 fast = fast->next->next;

 if (slow == fast) {
 int count = 1;
 Node* temp = slow->next;
 while (temp != slow) {
 count++;
 temp = temp->next;
 }
 return count;
 }
}
return 0;
}

int main() {
 Node* head = new Node(1);
 Node* n2 = new Node(2);
 Node* n3 = new Node(3);
 Node* n4 = new Node(4);
 Node* n5 = new Node(5);

 head->next = n2;
 n2->next = n3;
 n3->next = n4;
 n4->next = n5;
 n5->next = n3; // loop of length 3

 cout << lengthOfLoop(head);
 return 0;
}

```

Time Complexity: O(N)

Space Complexity: O(1)

# 14. Check if the given Linked List is Palindrome

You are given the head of a singly linked list where each node stores a digit. Check whether the sequence of digits forms a palindrome.

---

## Brute Force Approach (Using Stack)

**Idea:**

Store all node values in a stack.

Traverse the list again and compare each node with the top of the stack.

```
#include <bits/stdc++.h>
using namespace std;

class Node {
public:
 int data;
 Node* next;
 Node(int data1) {
 data = data1;
 next = NULL;
 }
};

bool isPalindrome(Node* head) {
 stack<int> st;
 Node* temp = head;

 while (temp != NULL) {
 st.push(temp->data);
 temp = temp->next;
 }

 temp = head;
```

```

 while (temp != NULL) {
 if (temp->data != st.top())
 return false;
 st.pop();
 temp = temp->next;
 }
 return true;
 }

int main() {
 Node* head = new Node(3);
 head->next = new Node(7);
 head->next->next = new Node(5);
 head->next->next->next = new Node(7);
 head->next->next->next->next = new Node(3);

 cout << (isPalindrome(head) ? "True" : "False");
 return 0;
}

```

Time Complexity: O(N)

Space Complexity: O(N)

---

## Optimal Approach (Reverse Second Half)

### Idea:

Find the middle using slow and fast pointers.

Reverse the second half of the list.

Compare first half and second half node by node.

Restore the list by reversing back.

```

#include <bits/stdc++.h>
using namespace std;

class Node {
public:
 int data;
 Node* next;

```

```

Node(int data1) {
 data = data1;
 next = NULL;
}
};

Node* reverseList(Node* head) {
 Node* prev = NULL;
 while (head != NULL) {
 Node* nextNode = head->next;
 head->next = prev;
 prev = head;
 head = nextNode;
 }
 return prev;
}

bool isPalindrome(Node* head) {
 if (head == NULL || head->next == NULL) return true;

 Node* slow = head;
 Node* fast = head;

 while (fast->next != NULL && fast->next->next != NULL) {
 slow = slow->next;
 fast = fast->next->next;
 }

 Node* second = reverseList(slow->next);
 Node* first = head;
 Node* temp = second;

 while (temp != NULL) {
 if (first->data != temp->data) {
 reverseList(second);
 return false;
 }
 first = first->next;
 temp = temp->next;
 }
 reverseList(second);
 return true;
}

```

```

 temp = temp->next;
 }

 reverseList(second);
 return true;
}

int main() {
 Node* head = new Node(1);
 head->next = new Node(1);
 head->next->next = new Node(2);
 head->next->next->next = new Node(1);

 cout << (isPalindrome(head) ? "True" : "False");
 return 0;
}

```

Time Complexity: O(N)

Space Complexity: O(1)

## 15. Segregate even and odd nodes in Linked List

You are given the head of a singly linked list.

Reorder the list so that **all even-valued nodes come first**, followed by **all odd-valued nodes**.

The **relative order inside even and odd groups must remain the same**.

---

### Idea

We do **not** change node values.

We only **rearrange links**.

Create two separate lists while traversing once:

- one list for even-valued nodes
- one list for odd-valued nodes

At the end, connect:

even list → odd list

This keeps order intact and avoids extra traversals.

---

### Code (single pass, pointer-based)

```
#include <bits/stdc++.h>
using namespace std;

class Node {
public:
 int data;
 Node* next;
 Node(int x) {
 data = x;
 next = NULL;
 }
};

Node* segregateEvenOdd(Node* head) {
 if (head == NULL) return head;

 Node* evenDummy = new Node(-1);
 Node* oddDummy = new Node(-1);
 Node* evenTail = evenDummy;
 Node* oddTail = oddDummy;

 Node* curr = head;
 while (curr != NULL) {
 if (curr->data % 2 == 0) {
 evenTail->next = curr;
 evenTail = evenTail->next;
 }
 else {
 oddTail->next = curr;
 oddTail = oddTail->next;
 }
 }
 evenTail->next = oddDummy->next;
 oddTail->next = NULL;
 return evenDummy->next;
}
```

```

 } else {
 oddTail->next = curr;
 oddTail = oddTail->next;
 }
 curr = curr->next;
 }

evenTail->next = oddDummy->next;
oddTail->next = NULL;

return evenDummy->next;
}

void printList(Node* head) {
 while (head != NULL) {
 cout << head->data << "->";
 head = head->next;
 }
 cout << "NULL\n";
}

int main() {
 Node* head = new Node(1);
 head->next = new Node(2);
 head->next->next = new Node(3);
 head->next->next->next = new Node(4);
 head->next->next->next->next = new Node(5);
 head->next->next->next->next->next = new Node(6);

 printList(head);
 head = segregateEvenOdd(head);
 printList(head);

 return 0;
}

```

---

Time Complexity: **O(N)**

Space Complexity: **O(1)** (only pointer manipulation, no extra data structures)

## 16. Remove N-th node from the end of a Linked List

You are given the head of a singly linked list and an integer **N**.

Delete the **N-th node from the end** of the list and return the updated head.

---

### Brute Force Approach

#### Idea:

Find the length of the linked list first.

The N-th node from the end is the ( $\text{length} - N + 1$ )-th node from the start.

#### Algorithm:

1. Traverse the list and count total nodes L.
2. If  $N == L$ , delete the head node.
3. Traverse again to the  $(L - N)$ -th node.
4. Adjust pointers to remove the next node.

```
#include <bits/stdc++.h>
```

```
using namespace std;
```

```
class Node {
```

```

public:

 int data;

 Node* next;

 Node(int data1) {

 data = data1;

 next = nullptr;

 }

};

Node* removeNthFromEnd(Node* head, int N) {

 if (!head) return nullptr;

 int len = 0;

 Node* temp = head;

 while (temp) {

 len++;

 temp = temp->next;

 }

 if (N == len) {

 Node* newHead = head->next;

```

```

 delete head;

 return newHead;

 }

temp = head;

for (int i = 1; i < len - N; i++) {

 temp = temp->next;

}

Node* del = temp->next;

temp->next = del->next;

delete del;

return head;

}

int main() {

Node* head = new Node(1);

head->next = new Node(2);

head->next->next = new Node(3);

head->next->next->next = new Node(4);

```

```

head->next->next->next->next = new Node(5);

head = removeNthFromEnd(head, 3);

while (head) {
 cout << head->data << " ";
 head = head->next;
}

return 0;
}

```

### **Complexity:**

Time:  $O(N)$

Space:  $O(1)$

### **Idea (two-pointer method)**

Instead of counting length, use **two pointers**.

Keep a gap of **N nodes** between fast and slow.

- Move fast ahead by  $N+1$  steps
- Move both fast and slow together
- When fast reaches NULL, slow is just **before** the node to delete
- Bypass that node

Using a **dummy node** simplifies edge cases (like deleting the head).

---

### Code (optimal, single pass)

```
#include <bits/stdc++.h>
using namespace std;

class Node {
public:
 int data;
 Node* next;
 Node(int x) {
 data = x;
 next = NULL;
 }
};

Node* removeNthFromEnd(Node* head, int N) {
 Node* dummy = new Node(0);
 dummy->next = head;

 Node* slow = dummy;
 Node* fast = dummy;

 for (int i = 0; i <= N; i++) {
 fast = fast->next;
 }

 while (fast != NULL) {
 slow = slow->next;
 fast = fast->next;
 }

 Node* delNode = slow->next;
 slow->next = slow->next->next;
 delete delNode;

 return dummy->next;
}
```

```

}

void printList(Node* head) {
 while (head != NULL) {
 cout << head->data << "->";
 head = head->next;
 }
 cout << "NULL\n";
}

int main() {
 Node* head = new Node(1);
 head->next = new Node(2);
 head->next->next = new Node(3);
 head->next->next->next = new Node(4);
 head->next->next->next->next = new Node(5);

 int N = 3;

 printList(head);
 head = removeNthFromEnd(head, N);
 printList(head);

 return 0;
}

```

---

## Why this works

When fast reaches the end, it has moved **N nodes more** than slow.  
So slow is exactly **one node before** the target node.

---

## Complexity

Time Complexity: **O(N)**  
Space Complexity: **O(1)**

# 17. Delete the Middle Node of the Linked List

You are given the head of a singly linked list.

Delete the **middle node** and return the updated head.

If the list has an **even number of nodes**, delete the **second middle** node.

---

## Brute Force Approach

### Idea:

Find the length of the list first.

Middle index =  $n / 2$  (0-based logic → this deletes second middle in even case).

Traverse again and delete that node.

### Steps:

1. Count total nodes n.
2. If  $n == 1$ , return NULL.
3. Traverse to node just **before** middle.
4. Adjust pointers and delete the middle node.

```
#include <bits/stdc++.h>
using namespace std;

class Node {
public:
 int data;
 Node* next;
 Node(int data1) {
 data = data1;
 next = nullptr;
 }
};
```

```

Node* deleteMiddle(Node* head) {
 if (head == nullptr || head->next == nullptr) {
 delete head;
 return nullptr;
 }

 int n = 0;
 Node* temp = head;
 while (temp) {
 n++;
 temp = temp->next;
 }

 int mid = n / 2;
 temp = head;

 for (int i = 1; i < mid; i++) {
 temp = temp->next;
 }

 Node* del = temp->next;
 temp->next = del->next;
 delete del;

 return head;
}

void printLL(Node* head) {
 while (head) {
 cout << head->data << " ";
 head = head->next;
 }
 cout << endl;
}

int main() {
 Node* head = new Node(1);

```

```

head->next = new Node(2);
head->next->next = new Node(3);
head->next->next->next = new Node(4);
head->next->next->next->next = new Node(5);

head = deleteMiddle(head);
printLL(head);
return 0;
}

```

### **Complexity:**

Time:  $O(N)$

Space:  $O(1)$

---

## **Optimal Approach (Slow & Fast Pointer)**

### **Idea:**

Use two pointers.

fast moves two steps, slow moves one step.

We stop slow **one node before** the middle, then delete slow->next.

This automatically deletes:

- exact middle for odd length
- **second middle** for even length

### **Steps:**

1. Handle 0 or 1 node case.
2. Initialize slow = head, fast = head->next->next.
3. Move slow by 1 and fast by 2.
4. Delete slow->next.

```

#include <bits/stdc++.h>
using namespace std;

class Node {
public:
 int data;
 Node* next;
 Node(int data1) {
 data = data1;
 next = nullptr;
 }
};

Node* deleteMiddle(Node* head) {
 if (head == nullptr || head->next == nullptr) {
 delete head;
 return nullptr;
 }

 Node* slow = head;
 Node* fast = head->next->next;

 while (fast != nullptr && fast->next != nullptr) {
 slow = slow->next;
 fast = fast->next->next;
 }

 Node* del = slow->next;
 slow->next = del->next;
 delete del;

 return head;
}

void printLL(Node* head) {
 while (head) {
 cout << head->data << " ";
 head = head->next;
 }
}

```

```

 }
 cout << endl;
}

int main() {
 Node* head = new Node(1);
 head->next = new Node(2);
 head->next->next = new Node(3);
 head->next->next->next = new Node(4);

 head = deleteMiddle(head);
 printLL(head);
 return 0;
}

```

**Complexity:**

Time:  $O(N)$

Space:  $O(1)$

## 18. Sort a Linked List

You are given the head of a singly linked list.

Sort the linked list in **ascending order** and return the head of the sorted list.

---

### Brute Force Approach

**Idea:**

Store all node values in an array, sort the array, then overwrite the linked list with sorted values.

**Steps:**

1. Traverse the linked list and store values in an array.
2. Sort the array.

3. Traverse the linked list again and replace node values from the sorted array.

```
#include <bits/stdc++.h>
using namespace std;

class Node {
public:
 int data;
 Node* next;
 Node(int data1) {
 data = data1;
 next = nullptr;
 }
};

Node* sortLL(Node* head) {
 vector<int> arr;
 Node* temp = head;

 while (temp) {
 arr.push_back(temp->data);
 temp = temp->next;
 }

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

 temp = head;
 for (int i = 0; i < arr.size(); i++) {
 temp->data = arr[i];
 temp = temp->next;
 }

 return head;
}

void printLL(Node* head) {
 while (head) {
 cout << head->data << " ";
 }
}
```

```

 head = head->next;
 }
 cout << endl;
}

int main() {
 Node* head = new Node(3);
 head->next = new Node(4);
 head->next->next = new Node(2);
 head->next->next->next = new Node(1);
 head->next->next->next->next = new Node(5);

 head = sortLL(head);
 printLL(head);
 return 0;
}

```

### **Complexity:**

Time:  $O(N \log N)$

Space:  $O(N)$

---

## **Optimal Approach (Merge Sort on Linked List)**

### **Idea:**

Linked lists are best sorted using **Merge Sort** because it does not require random access.

### **Steps:**

1. If list has 0 or 1 node, it is already sorted.
2. Find the middle using slow & fast pointers.
3. Split the list into two halves.
4. Recursively sort both halves.
5. Merge the two sorted halves.

```

#include <bits/stdc++.h>
using namespace std;

class Node {
public:
 int data;
 Node* next;
 Node(int data1) {
 data = data1;
 next = nullptr;
 }
};

Node* merge(Node* l1, Node* l2) {
 Node dummy(-1);
 Node* temp = &dummy;

 while (l1 && l2) {
 if (l1->data <= l2->data) {
 temp->next = l1;
 l1 = l1->next;
 } else {
 temp->next = l2;
 l2 = l2->next;
 }
 temp = temp->next;
 }

 temp->next = (l1 ? l1 : l2);
 return dummy.next;
}

Node* findMiddle(Node* head) {
 Node* slow = head;
 Node* fast = head->next;

 while (fast && fast->next) {
 slow = slow->next;
 }
}

```

```

 fast = fast->next->next;
 }
 return slow;
}

Node* sortLL(Node* head) {
 if (!head || !head->next) return head;

 Node* mid = findMiddle(head);
 Node* right = mid->next;
 mid->next = nullptr;

 Node* left = sortLL(head);
 right = sortLL(right);

 return merge(left, right);
}

void printLL(Node* head) {
 while (head) {
 cout << head->data << " ";
 head = head->next;
 }
 cout << endl;
}

int main() {
 Node* head = new Node(40);
 head->next = new Node(20);
 head->next->next = new Node(60);
 head->next->next->next = new Node(10);
 head->next->next->next->next = new Node(50);
 head->next->next->next->next->next = new Node(30);

 head = sortLL(head);
 printLL(head);
 return 0;
}

```

### **Complexity:**

Time:  $O(N \log N)$

Space:  $O(1)$  (excluding recursion stack)

## **19. Sort a Linked List of 0's, 1's and 2's (by changing links)**

You are given a singly linked list containing only 0, 1, and 2.

You must sort the list by **rearranging links**, not by modifying node data.

---

### **Brute Force Approach**

#### **Idea:**

Count the number of 0s, 1s, and 2s, then rebuild the linked list in sorted order by **creating new links**.

#### **Steps:**

1. Traverse the linked list and count occurrences of 0, 1, and 2.
2. Traverse the list again and overwrite links in order: first 0s, then 1s, then 2s.
3. Do not create new data values, only reuse existing nodes.

```
#include <bits/stdc++.h>
using namespace std;

class Node {
public:
 int data;
 Node* next;
 Node(int d) {
```

```

 data = d;
 next = nullptr;
 }
};

Node* sortList(Node* head) {
 int cnt0 = 0, cnt1 = 0, cnt2 = 0;
 Node* temp = head;

 while (temp) {
 if (temp->data == 0) cnt0++;
 else if (temp->data == 1) cnt1++;
 else cnt2++;
 temp = temp->next;
 }

 temp = head;
 while (cnt0--) {
 temp->data = 0;
 temp = temp->next;
 }
 while (cnt1--) {
 temp->data = 1;
 temp = temp->next;
 }
 while (cnt2--) {
 temp->data = 2;
 temp = temp->next;
 }
 return head;
}

void printLL(Node* head) {
 while (head) {
 cout << head->data << " ";
 head = head->next;
 }
 cout << endl;
}

```

```

}

int main() {
 Node* head = new Node(1);
 head->next = new Node(2);
 head->next->next = new Node(0);
 head->next->next->next = new Node(1);
 head->next->next->next->next = new Node(0);
 head->next->next->next->next->next = new Node(2);

 head = sortList(head);
 printLL(head);
 return 0;
}

```

### **Complexity:**

Time:  $O(N)$

Space:  $O(1)$

---

## **Optimal Approach (Link Rearrangement)**

### **Idea:**

Create **three separate linked lists** for 0, 1, and 2 using dummy nodes.

Traverse once, attach each node to its respective list, then merge all three lists.

### **Steps:**

1. Create dummy heads and tails for lists of 0, 1, and 2.
2. Traverse original list and attach nodes to corresponding list.
3. Connect 0 list → 1 list → 2 list.
4. Update head to the start of sorted list.

```
#include <bits/stdc++.h>
using namespace std;
```

```

class Node {
public:
 int data;
 Node* next;
 Node(int d) {
 data = d;
 next = nullptr;
 }
};

Node* sortList(Node* head) {
 if (!head || !head->next) return head;

 Node* zeroDummy = new Node(-1);
 Node* oneDummy = new Node(-1);
 Node* twoDummy = new Node(-1);

 Node* zero = zeroDummy;
 Node* one = oneDummy;
 Node* two = twoDummy;

 Node* curr = head;
 while (curr) {
 if (curr->data == 0) {
 zero->next = curr;
 zero = zero->next;
 } else if (curr->data == 1) {
 one->next = curr;
 one = one->next;
 } else {
 two->next = curr;
 two = two->next;
 }
 curr = curr->next;
 }

 zero->next = (oneDummy->next) ? oneDummy->next : twoDummy->next;
}

```

```

one->next = twoDummy->next;
two->next = nullptr;

head = zeroDummy->next;

delete zeroDummy;
delete oneDummy;
delete twoDummy;

return head;
}

void printLL(Node* head) {
 while (head) {
 cout << head->data << " ";
 head = head->next;
 }
 cout << endl;
}

int main() {
 Node* head = new Node(2);
 head->next = new Node(1);
 head->next->next = new Node(2);
 head->next->next->next = new Node(0);
 head->next->next->next->next = new Node(0);
 head->next->next->next->next->next = new Node(1);

 head = sortList(head);
 printLL(head);
 return 0;
}

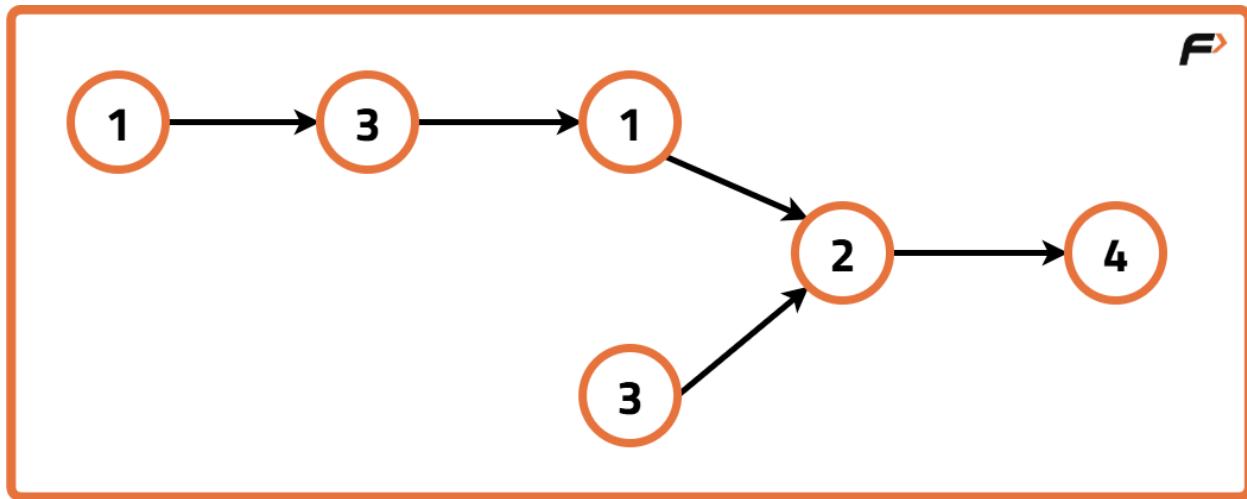
```

### **Complexity:**

Time:  $O(N)$

Space:  $O(1)$

# Find intersection of Two Linked Lists



Example 1: Input:

List 1 = [1, 3, 1, 2, 4], List 2 = [3, 2, 4]

Output:

2

Explanation: Here, both lists intersecting nodes start from node 2.

## Brute Force Approach

### Idea:

Intersection means **same node address**, not same value.

### Algorithm:

- For every node of list2, traverse list1.
- If temp == head2, intersection found.
- If traversal ends, return NULL.

```
node* intersectionPresent(node* head1, node* head2) {
 while (head2 != NULL) {
 node* temp = head1;
 while (temp != NULL) {
 if (temp == head2) return head2;
 temp = temp->next;
 }
 }
}
```

```

 head2 = head2->next;
 }
 return NULL;
}

```

### **Complexity:**

- Time:  $O(m \cdot n)$
  - Space:  $O(1)$
- 

### **Better Approach (Hashing)**

#### **Idea:**

Store node **addresses** of list1, then search while traversing list2.

#### **Algorithm:**

- Traverse list1, store each node in hash set.
- Traverse list2, if node exists in set → intersection.

```

node* intersectionPresent(node* head1, node* head2) {
 unordered_set<node*> st;
 while (head1 != NULL) {
 st.insert(head1);
 head1 = head1->next;
 }
 while (head2 != NULL) {
 if (st.find(head2) != st.end()) return head2;
 head2 = head2->next;
 }
 return NULL;
}

```

### **Complexity:**

- Time:  $O(n + m)$
  - Space:  $O(n)$
- 

## Optimal Approach 1 (Length Difference)

### Idea:

Align both lists at same remaining length, then move together.

### Algorithm:

- Find lengths of both lists.
- Move longer list by  $|len1 - len2|$ .
- Move both pointers together until they meet.

```
node* intersectionPresent(node* head1, node* head2) {
 int len1 = 0, len2 = 0;
 node* t1 = head1;
 node* t2 = head2;

 while (t1) { len1++; t1 = t1->next; }
 while (t2) { len2++; t2 = t2->next; }

 t1 = head1;
 t2 = head2;

 if (len1 > len2)
 for (int i = 0; i < len1 - len2; i++) t1 = t1->next;
 else
 for (int i = 0; i < len2 - len1; i++) t2 = t2->next;

 while (t1 && t2) {
 if (t1 == t2) return t1;
 t1 = t1->next;
 t2 = t2->next;
 }
}
```

```

 }
 return NULL;
}

```

### Complexity:

- Time:  $O(n + m)$
- Space:  $O(1)$

## Optimal Approach 2 (Two Pointer Switching)

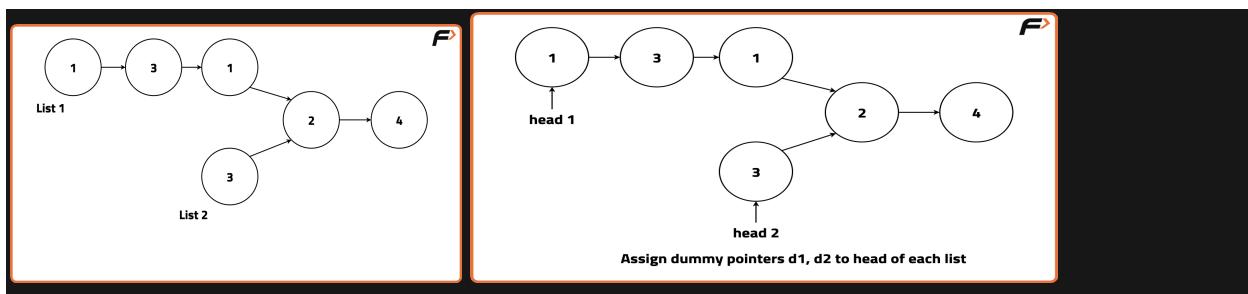
### Idea:

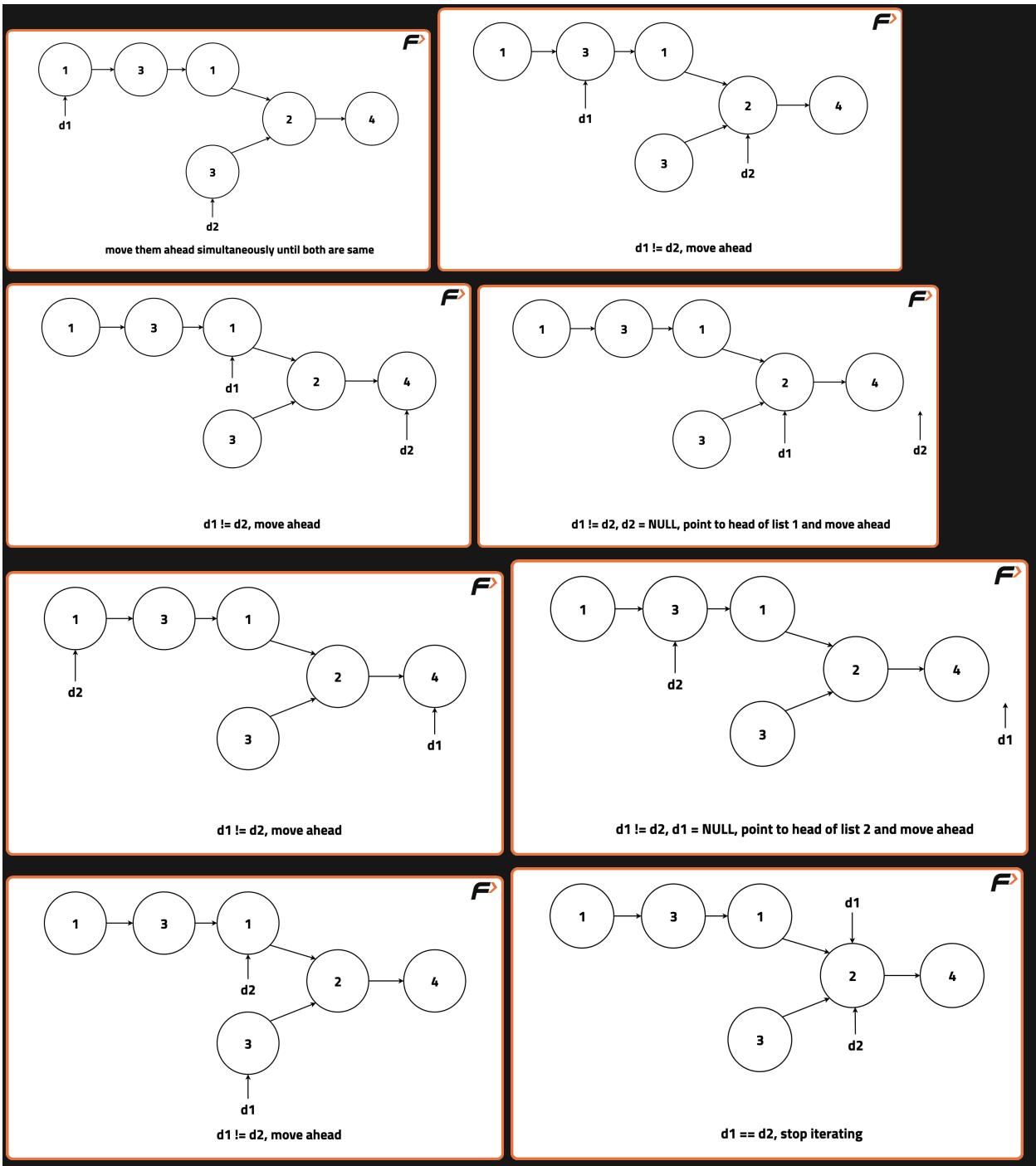
Both pointers traverse equal total distance by switching lists.

### Algorithm:

- Initialize  $d1 = \text{head1}$ ,  $d2 = \text{head2}$ .
- Move both; when one hits NULL, redirect to other list.
- Meeting point is intersection.

### Dry Run:





Dono pointers ko **same total distance** chalwa do.

List1 length =  $L_1$

List2 length =  $L_2$

Common tail length =  $C$

**d1 travels = L1 + L2**

**d2 travels = L2 + L1**

```
#include<bits/stdc++.h>
using namespace std;

class node {
public:
 int num;
 node* next;
 node(int val) {
 num = val;
 next = NULL;
 }
};

// Utility function to insert node at the end of the linked list
void insertNode(node* &head, int val) {
 node* newNode = new node(val);

 if (head == NULL) {
 head = newNode;
 return;
 }

 node* temp = head;
 while (temp->next != NULL) temp = temp->next;

 temp->next = newNode;
}

// Utility function to check presence of intersection
node* intersectionPresent(node* head1, node* head2) {
 node* d1 = head1;
 node* d2 = head2;

 // Traverse both lists, when one reaches the end, redirect it to the head
 // of the other list
 while (d1 != d2) {
 d1 = d1 == NULL ? head2 : d1->next;
 d2 = d2 == NULL ? head1 : d2->next;
 }
}
```

```

 return d1; // If they meet, return the intersection node, otherwise NULL
 }

// Utility function to print linked list
void printList(node* head) {
 while (head->next != NULL) {
 cout << head->num << "->";
 head = head->next;
 }
 cout << head->num << endl;
}

int main() {
 node* head = NULL;
 insertNode(head, 1);
 insertNode(head, 3);
 insertNode(head, 1);
 insertNode(head, 2);
 insertNode(head, 4);
 node* head1 = head;
 head = head->next->next->next; // Intersection point
 node* headSec = NULL;
 insertNode(headSec, 3);
 node* head2 = headSec;
 headSec->next = head; // Creating intersection

 // Printing the lists
 cout << "List1: ";
 printList(head1);
 cout << "List2: ";
 printList(head2);

 // Checking if intersection is present
 node* answerNode = intersectionPresent(head1, head2);
 if (answerNode == NULL)
 cout << "No intersection\n";
 else
 cout << "The intersection point is " << answerNode->num << endl;

 return 0;
}

```

## Complexity:

- Time:  $O(n + m)$
- Space:  $O(1)$

## 20. Add 1 to a number represented by LL

---

### Iterative approach

#### Algorithm

- Reverse the linked list.
- Add 1 starting from the head (least significant digit).
- Propagate carry if needed.
- If carry remains at the end, create a new node.
- Reverse the list again and return head.

```
#include <bits/stdc++.h>
using namespace std;

class Node {
public:
 int data;
 Node* next;
 Node(int value) {
 data = value;
```

```

 next = nullptr;
 }
};

class Solution {
public:
 Node* reverseList(Node* head) {
 Node* prev = nullptr;
 Node* curr = head;
 while (curr) {
 Node* nextNode = curr->next;
 curr->next = prev;
 prev = curr;
 curr = nextNode;
 }
 return prev;
 }

 Node* addOne(Node* head) {
 head = reverseList(head);

 Node* curr = head;
 int carry = 1;

 while (curr && carry) {
 int sum = curr->data + carry;
 curr->data = sum % 10;
 carry = sum / 10;

 if (!curr->next && carry) {
 curr->next = new Node(carry);
 carry = 0;
 }
 curr = curr->next;
 }

 head = reverseList(head);
 return head;
 }
};

```

```

 }
};

int main() {
 Node* head = new Node(4);
 head->next = new Node(5);
 head->next->next = new Node(6);

 Solution sol;
 head = sol.addOne(head);

 while (head) {
 cout << head->data;
 head = head->next;
 }
}

```

## Complexity

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

## Recursive approach

### Algorithm

- Recursively go to the last node.
- Add 1 and propagate carry backwards.
- If carry remains after head, create a new node at front.

```
#include <bits/stdc++.h>
using namespace std;
```

```

class Node {
public:
 int data;
 Node* next;
 Node(int value) {
 data = value;
 next = nullptr;
 }
};

class Solution {
public:
 int addOneUtil(Node* node) {
 if (!node) return 1;

 int carry = addOneUtil(node->next);
 int sum = node->data + carry;
 node->data = sum % 10;
 return sum / 10;
 }

 Node* addOne(Node* head) {
 int carry = addOneUtil(head);
 if (carry) {
 Node* newHead = new Node(carry);
 newHead->next = head;
 head = newHead;
 }
 return head;
 }
};

int main() {
 Node* head = new Node(9);
 head->next = new Node(9);
 head->next->next = new Node(9);

 Solution sol;

```

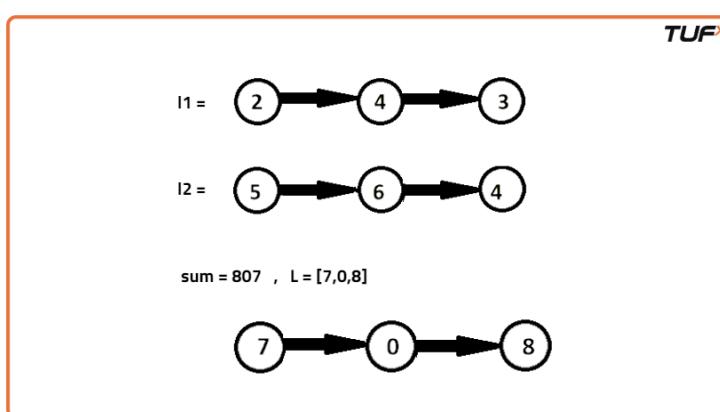
```
head = sol.addOne(head);

while (head) {
 cout << head->data;
 head = head->next;
}
}
```

## Complexity

- Time:  $O(n)$
- Space:  $O(n)$  (recursion stack)

# 21. Add two numbers represented as Linked Lists



## Approach

### Algorithm

- Create a dummy node to simplify result list creation.
- Use a temporary pointer starting at the dummy.
- Maintain a carry initialized to 0.
- Traverse both linked lists while at least one list has nodes left **or** carry is non-zero.
- At each step:
  - Add current digits from both lists (if present) and the carry.
  - Update carry as  $sum / 10$ .
  - Create a new node with value  $sum \% 10$  and attach it.
- Return dummy $\rightarrow$ next as the head of the result list.

## Code

```
#include <bits/stdc++.h>
using namespace std;

// Definition for singly-linked list.
struct ListNode {
 int val;
 ListNode* next;
 ListNode(int x = 0) {
 val = x;
 next = nullptr;
 }
};

class Solution {
public:
 ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
 ListNode* dummy = new ListNode();
 ListNode* temp = dummy;
 int carry = 0;

 while (l1 != nullptr || l2 != nullptr || carry) {
 int sum = carry;

 if (l1) {
 sum += l1->val;
 l1 = l1->next;
 }
 if (l2) {
 sum += l2->val;
 l2 = l2->next;
 }

 carry = sum / 10;
 temp->next = new ListNode(sum % 10);
 temp = temp->next;
 }
 return dummy->next;
 }
};
```

```

 }
};

ListNode* createList(vector<int>& arr) {
 ListNode* head = new ListNode(arr[0]);
 ListNode* temp = head;
 for (int i = 1; i < arr.size(); i++) {
 temp->next = new ListNode(arr[i]);
 temp = temp->next;
 }
 return head;
}

void printList(ListNode* head) {
 while (head) {
 cout << head->val;
 if (head->next) cout << " -> ";
 head = head->next;
 }
 cout << endl;
}

int main() {
 vector<int> num1 = {2, 4, 3};
 vector<int> num2 = {5, 6, 4};

 ListNode* l1 = createList(num1);
 ListNode* l2 = createList(num2);

 Solution sol;
 ListNode* result = sol.addTwoNumbers(l1, l2);

 printList(result); // 7 -> 0 -> 8
 return 0;
}

```

---

## Complexity Analysis

- **Time Complexity:**  $O(\max(m, n))$
- **Space Complexity:**  $O(\max(m, n))$

# 22. Delete all occurrences of a given key in a doubly linked list

---

### Intuition

We just need to **traverse the DLL once** and whenever we find a node whose `data == key`, we **delete that node and fix the links** (`prev` and `next`).

Since it's a doubly linked list, deletion can be done in  **$O(1)$**  time if we already have the node reference.

---

### Approach

#### Algorithm

- If the list is empty, do nothing.
- Use a current pointer starting from head.
- Traverse the list:
  - If `current->data == key`:

- Store `current->next` in a temporary pointer.
- Delete current node using a helper function.
- Move current to the saved next node.
- Else:
  - Move current forward normally.
- Continue till the end of the list.

A helper function `deleteNode` is used to safely delete a node from a doubly linked list by updating adjacent pointers.

---

## Code

```
class Solution {
public:
 Node* deleteAllOccurOfX(Node* head, int key) {
 if (head == NULL) return NULL;

 Node* curr = head;

 while (curr != NULL) {
 if (curr->data == key) {
 Node* nextNode = curr->next;

 // If node to delete is head
 if (curr == head) {
 head = head->next;
 if (head != NULL)
 head->prev = NULL;
 } else {
 // Fix previous node
 curr->prev->next = curr->next;

 // Fix next node
 if (curr->next != NULL)
 curr->next->prev = curr->prev;
 }
 }
 }
 }
}
```

```

 delete curr;
 curr = nextNode;
 } else {
 curr = curr->next;
 }
}

return head;
}
};

```

---

## Complexity Analysis

- **Time Complexity:**  $O(N)$   
Each node is visited once, and deletion is  $O(1)$ .
- **Space Complexity:**  $O(1)$   
No extra data structures are used.

## 23. Find pairs with given sum in doubly linked list

---

List **sorted** hai → Two Pointer best rahega.  
DLL me advantage ye hai ki hum **prev** bhi use kar sakte hain.

- `left = head`
  - `right = last node`
  - Dono side se move karke pairs find karte hain
- 

## Algorithm

1. Agar list empty ho → empty answer return
  2. right pointer ko last node tak le jao
  3. `left = head`
  4. Jab tak left aur right cross na kare:
    - `sum = left->data + right->data`
    - Agar `sum == target` → pair store, `left++`, `right--`
    - Agar `sum < target` → `left++`
    - Agar `sum > target` → `right--`
  5. Answer return karo
- 

## Code (simple, clean, previous trend follow)

```
class Solution {
public:
 vector<pair<int,int>> findPairsWithGivenSum(Node* head, int
target) {
 vector<pair<int,int>> ans;
 if (head == NULL) return ans;
```

```

Node* left = head;
Node* right = head;

// move right to last node
while (right->next != NULL) {
 right = right->next;
}

// two pointer approach
while (left != right && right->next != left) {
 int sum = left->data + right->data;

 if (sum == target) {
 ans.push_back({left->data, right->data});
 left = left->next;
 right = right->prev;
 }
 else if (sum < target) {
 left = left->next;
 }
 else {
 right = right->prev;
 }
}

return ans;
}
};

```

---

## Complexity

- **Time:**  $O(N)$
- **Space:**  $O(1)$  (answer vector ke alawa)

# 24. Remove duplicates from sorted DLL

---

## Idea

List **sorted** hai → duplicates **adjacent** honge.

Har node ko uske next se compare karo.

Agar same value mile → duplicate node delete karo aur links adjust karo.

---

## Algorithm

1. Agar list empty ho → return head
  2. curr pointer ko head par rakho
  3. Jab tak curr aur curr->next exist kare:
    - Agar curr->data == curr->next->data
      - duplicate node delete karo
      - curr->next ko aage wale node se jodo
    - warna curr = curr->next
  4. Head return karo
- 

## Code (simple, previous trend follow)

```
class Solution {
public:
 Node* removeDuplicates(Node* head) {
 if (head == NULL) return head;
 }
```

```

Node* curr = head;

while (curr != NULL && curr->next != NULL) {
 if (curr->data == curr->next->data) {
 Node* dup = curr->next;
 curr->next = dup->next;

 if (dup->next != NULL) {
 dup->next->prev = curr;
 }

 delete dup;
 }
 else {
 curr = curr->next;
 }
}

return head;
}
};

```

---

## Complexity

- **Time:**  $O(N)$
- **Space:**  $O(1)$

# 25. Reverse Linked List in groups of Size K

---

## Idea

Linked list ko **k size ke groups** me todte hain.

Har group me **sirf links reverse** karte hain (data change nahi).

Agar last group me **k se kam nodes** hain → usko as-it-is chhod dete hain.

---

## Algorithm

1. Dummy node use karo taaki head change easily handle ho sake
  2. groupPrev pointer rakho jo previous reversed group ke end ko point kare
  3. Har iteration me:
    - getKthNode se check karo kya k nodes available hain
    - Agar nahi → break
    - Current group ko reverse karo using pointer reversal
    - Previous group ko current reversed group se connect karo
  4. Last me dummy->next return karo
- 

## Code

```
#include <bits/stdc++.h>
using namespace std;
```

```

// Definition of singly-linked list node
class ListNode {
public:
 int val;
 ListNode* next;
 ListNode(int x) {
 val = x;
 next = NULL;
 }
};

class Solution {
public:
 // Function to reverse nodes in groups of k
 ListNode* reverseKGroup(ListNode* head, int k) {
 // Creating a dummy node to handle edge cases easily
 ListNode* dummy = new ListNode(0);
 dummy->next = head;

 // Pointer to keep track of the previous group's tail
 ListNode* groupPrev = dummy;

 while (true) {
 // Finding the k-th node from the groupPrev
 ListNode* kth = getKthNode(groupPrev, k);
 if (!kth) break;

 // Store the next group's head
 ListNode* groupNext = kth->next;

 // Break the chain to reverse current k-group cleanly
 ListNode* prev = groupNext; //next group
 ListNode* curr = groupPrev->next;

 // Reversing k nodes
 for (int i = 0; i < k; i++) {
 ListNode* temp = curr->next;
 curr->next = prev;
 prev = curr;
 curr = temp;
 }

 // Connecting previous group to the reversed group
 ListNode* temp = groupPrev->next;
 groupPrev->next = kth;
 groupPrev = temp;
 }
 }
};

```

```

 // Returning the new head
 return dummy->next;
}

// Helper function to find the k-th node from the current node
ListNode* getKthNode(ListNode* curr, int k) {
 while (curr && k > 0) {
 curr = curr->next;
 k--;
 }
 return curr;
};

// Driver code
int main() {
 Solution obj;

 // Creating the linked list: 1->2->3->4->5
 ListNode* head = new ListNode(1);
 head->next = new ListNode(2);
 head->next->next = new ListNode(3);
 head->next->next->next = new ListNode(4);
 head->next->next->next->next = new ListNode(5);

 int k = 2;

 // Reversing in groups of k
 ListNode* result = obj.reverseKGroup(head, k);

 // Printing the reversed list
 while (result != NULL) {
 cout << result->val << " ";
 result = result->next;
 }
 cout << endl;

 return 0;
}

```

---

## Complexity

- **Time Complexity:**  $O(N)$

- **Space Complexity:**  $O(1)$

## 26. Rotate a Linked List

**Input :** head  $\rightarrow$  1  $\rightarrow$  2  $\rightarrow$  3  $\rightarrow$  4  $\rightarrow$  5, k = 2

**Output :** head  $\rightarrow$  4  $\rightarrow$  5  $\rightarrow$  1  $\rightarrow$  2  $\rightarrow$  3

**Explanation :** List after 1 shift to right: head  $\rightarrow$  5  $\rightarrow$  1  $\rightarrow$  2  $\rightarrow$  3  $\rightarrow$  4.

List after 2 shift to right: head  $\rightarrow$  4  $\rightarrow$  5  $\rightarrow$  1  $\rightarrow$  2  $\rightarrow$  3.

---

### Brute Force Approach

#### Idea

Right rotation ka matlab: **last node ko uthao aur head bana do.**

Ye process **k times** repeat karte hain.

---

#### Algorithm

1. Agar list empty ho, ek hi node ho, ya k = 0 ho  $\rightarrow$  head return karo
  2. k times repeat karo:
    - List ko traverse karke **last aur second-last node** tak jao
    - Last node ko detach karo
    - Usko head ke aage laga do
  3. Final head return karo
- 

#### Code (Brute Force)

```
class ListNode {
public:
```

```

int val;
ListNode* next;
ListNode(int x) {
 val = x;
 next = NULL;
}
};

class Solution {
public:
 ListNode* rotateRight(ListNode* head, int k) {
 if (!head || !head->next || k == 0) return head;

 while (k--) {
 ListNode* curr = head;
 ListNode* prev = NULL;

 while (curr->next) {
 prev = curr;
 curr = curr->next;
 }

 prev->next = NULL;
 curr->next = head;
 head = curr;
 }
 return head;
 }
};

```

---

## Complexity

- **Time:**  $O(k * N)$
  - **Space:**  $O(1)$
-

# Optimal Approach

Har length rotations ke baad list wapas same ho jaati hai.

Isliye:

- $k = k \% \text{length}$
  - List ko **circular** bana do
  - Naya tail aur naya head find karke circle tod do
- 

## Algorithm

1. Edge cases handle karo (empty list, single node,  $k = 0$ )
  2. List ka **length** nikalo aur **tail** tak jao
  3. Tail ko head se connect karke circular list banao
  4.  $k = k \% \text{length}$
  5.  $(\text{length} - k)$  steps chalke **newTail** find karo
  6. **newHead** = **newTail->next**
  7. **newTail->next** = **NULL**
  8. **newHead** return karo
- 

## Code (Optimal)

```
class ListNode {
public:
 int val;
 ListNode* next;
 ListNode(int x) {
```

```

 val = x;
 next = NULL;
 }
};

class Solution {
public:
 ListNode* rotateRight(ListNode* head, int k) {
 if (!head || !head->next || k == 0) return head;

 int length = 1;
 ListNode* tail = head;

 while (tail->next) {
 tail = tail->next;
 length++;
 }

 tail->next = head;
 k = k % length;

 int steps = length - k;
 ListNode* newTail = head;
 for (int i = 1; i < steps; i++) {
 newTail = newTail->next;
 }

 ListNode* newHead = newTail->next;
 newTail->next = NULL;

 return newHead;
 }
};

```

---

- **Time:**  $O(N)$

- **Space:**  $O(1)$

# 27. Flattening a Linked List

---

## Brute Force Approach

### Idea

Saare nodes (top level + child lists) ke values collect karo, sort karo, aur phir ek **nayi single-level linked list** bana do (sirf child pointer use karke).

---

### Algorithm

1. Ek vector lo
  2. next pointer se top-level list traverse karo
  3. Har node ke liye uski child list traverse karke values vector me daalo
  4. Vector ko sort karo
  5. Sorted vector se nayi linked list banao (using child pointer)
  6. Uska head return karo
- 

### Code (Brute Force)

```
struct Node {
 int data;
 Node* next;
 Node* child;
 Node(int x) {
 data = x;
```

```

 next = NULL;
 child = NULL;
 }
};

class Solution {
public:
 Node* buildList(vector<int>& arr) {
 Node* dummy = new Node(-1);
 Node* temp = dummy;

 for (int x : arr) {
 temp->child = new Node(x);
 temp = temp->child;
 }
 return dummy->child;
 }

 Node* flattenLinkedList(Node* head) {
 vector<int> arr;

 while (head) {
 Node* curr = head;
 while (curr) {
 arr.push_back(curr->data);
 curr = curr->child;
 }
 head = head->next;
 }

 sort(arr.begin(), arr.end());
 return buildList(arr);
 }
};

```

---

## Complexity

- **Time:**  $O(N*M \log(N*M))$
  - **Space:**  $O(N*M)$
- 

## Optimal Approach

### Idea

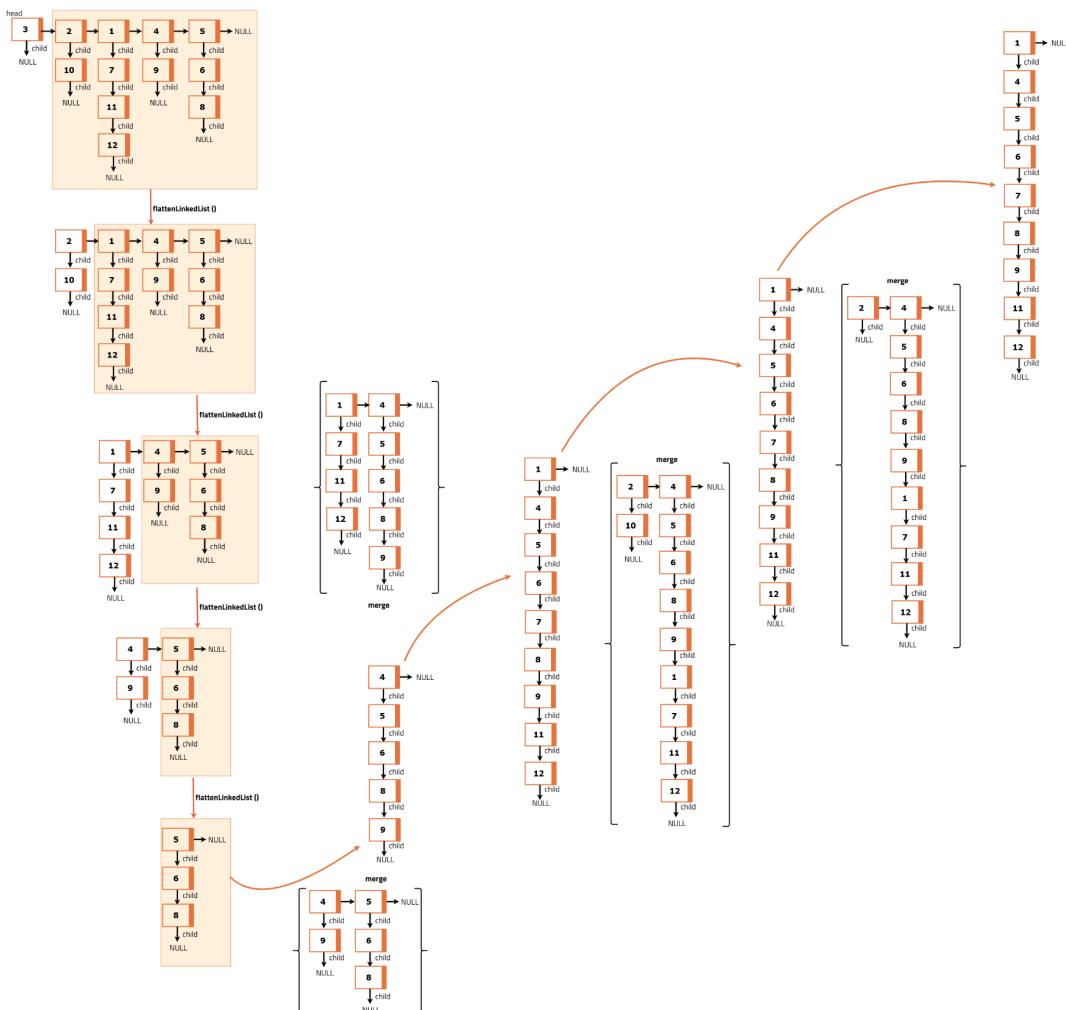
Har child list **already sorted** hai.

Hum recursion ke through **right se left** flatten karte hain aur **do sorted lists ko merge** karte jaate hain (merge sort style).

---

### Algorithm

1. Base case: agar head == NULL ya head->next == NULL → return head
2. Recursively head->next ko flatten karo
3. Ab current list (head) aur flattened list ko merge karo
4. Merge hamesha child pointer se hogा
5. next pointer ko NULL rakho



## Code (Optimal)

```
struct Node {
 int data;
 Node* next;
 Node* child;
 Node(int x) {
 data = x;
 next = NULL;
 child = NULL;
 }
}
```

```

};

class Solution {
public:
 Node* merge(Node* a, Node* b) {
 if (!a) return b;
 if (!b) return a;

 Node* result;
 if (a->data < b->data) {
 result = a;
 result->child = merge(a->child, b);
 } else {
 result = b;
 result->child = merge(a, b->child);
 }
 result->next = NULL;
 return result;
 }

 Node* flattenLinkedList(Node* head) {
 if (!head || !head->next) return head;

 head->next = flattenLinkedList(head->next);
 head = merge(head, head->next);

 return head;
 }
};

```

---

## Complexity

- **Time:**  $O(N*M)$
- **Space:**  $O(N)$  (recursion stack)

# 28. Clone Linked List with Random and Next Pointer

---

Create a 'deep copy' of the given linked list and return it.

Example 1:

Input: [[1, -1], [2, 0], [3, 4], [4, 1], [5, 2]]

Output: 1 2 3 4 5, true

Explanation: All the nodes in the new list have same corresponding values as original nodes.

All the random pointers point to their corresponding nodes in the new list.

'true' represents that the nodes and references were created new.

## Brute Force Approach

### Algorithm

1. Traverse the original list and create a copy node for every original node.
2. Store mapping between **original node** → **copied node** using a hashmap.
  - x
3. Traverse the list again:
  - o Set next pointer of copied node using the map.
  - o Set random pointer of copied node using the map.
4. Return the copied head corresponding to original head.

---

### Code (Brute Force)

```
#include <bits/stdc++.h>
using namespace std;
```

```

class Node {
public:
 int data;
 Node* next;
 Node* random;

 Node(int x) {
 data = x;
 next = NULL;
 random = NULL;
 }
};

class Solution {
public:
 Node* copyRandomList(Node* head) {
 if (!head) return NULL;

 unordered_map<Node*, Node*> mp;
 Node* temp = head;

 while (temp) {
 mp[temp] = new Node(temp->data);
 temp = temp->next;
 }

 temp = head;
 while (temp) {
 mp[temp]->next = mp[temp->next];
 mp[temp]->random = mp[temp->random];
 temp = temp->next;
 }

 return mp[head];
 }
};

```

---

## Complexity

- **Time Complexity:** O(N)
  - **Space Complexity:** O(N)
- 

## Optimal Approach

### Algorithm

1. Traverse the list and insert copied nodes **in between** original nodes.
  2. Traverse again to set random pointers of copied nodes.
  3. Traverse once more to **separate original and copied lists**.
  4. Return head of the copied list.
- 

### Code (Optimal)

```
#include <iostream>
using namespace std;

// Node class to represent
// elements in the linked list
class Node {
public:
 // Data stored in the node
 int data;
 // Pointer to the next node
 Node *next;
 // Pointer to a random
 // node in the list
 Node *random;

 // Constructors for Node class
 Node() : data(0), next(nullptr), random(nullptr) {};
 Node(int x) : data(x), next(nullptr), random(nullptr) {}
 // Constructor with data,
 // next, and random pointers
```

```

Node(int x, Node *nextNode, Node *randomNode) :
 data(x), next(nextNode), random(randomNode) {}
};

// Function to insert a copy of each
// node in between the original nodes
void insertCopyInBetween(Node* head) {
 Node* temp = head;
 while(temp != NULL) {
 Node* nextElement = temp->next;
 // Create a new node with the same data
 Node* copy = new Node(temp->data);

 // Point the copy's next to
 // the original node's next
 copy->next = nextElement;

 // Point the original
 // node's next to the copy
 temp->next = copy;

 // Move to the next original node
 temp = nextElement;
 }
}

// Function to connect random
// pointers of the copied nodes
void connectRandomPointers(Node* head) {
 Node* temp = head;
 while(temp != NULL) {
 // Access the copied node
 Node* copyNode = temp->next;

 // If the original node
 // has a random pointer
 if(temp->random) {
 // Point the copied node's random to the
 // corresponding copied random node
 copyNode->random = temp->random->next;
 }
 else{
 // Set the copied node's random to
 // null if the original random is null
 copyNode->random = NULL;
 }

 // Move to the next original node
 temp = temp->next->next;
 }
}

```

```

 }

 }

// Function to retrieve the
// deep copy of the linked list
Node* getDeepCopyList(Node* head) {
 Node* temp = head;
 // Create a dummy node
 Node* dummyNode = new Node(-1);
 // Initialize a result pointer
 Node* res = dummyNode;

 while(temp != NULL) {
 // Creating a new List by
 // pointing to copied nodes
 res->next = temp->next;
 res = res->next;

 // Disconnect and revert back to the
 // initial state of the original linked list
 temp->next = temp->next->next;
 temp = temp->next;
 }

 // Return the deep copy of the
 // list starting from the dummy node
 return dummyNode->next;
}

// Function to clone the linked list
Node *cloneLL(Node *head) {
 // If the original list
 // is empty, return null
 if(!head) return nullptr;

 // Step 1: Insert copy of
 // nodes in between
 insertCopyInBetween(head);
 // Step 2: Connect random
 // pointers of copied nodes
 connectRandomPointers(head);
 // Step 3: Retrieve the deep
 // copy of the linked list
 return getDeepCopyList(head);
}

// Function to print the cloned linked list
void printClonedLinkedList(Node *head) {
 while (head != nullptr) {

```

```

 cout << "Data: " << head->data;
 if (head->random != nullptr) {
 cout << ", Random: " << head->random->data;
 } else {
 cout << ", Random: nullptr";
 }
 cout << endl;
 // Move to the next node
 head = head->next;
 }
}

// Main function
int main() {
 // Example linked list: 7 -> 14 -> 21 -> 28
 Node* head = new Node(7);
 head->next = new Node(14);
 head->next->next = new Node(21);
 head->next->next->next = new Node(28);

 // Assigning random pointers
 head->random = head->next->next;
 head->next->random = head;
 head->next->next->random = head->next->next->next;
 head->next->next->next->random = head->next;

 cout << "Original Linked List with Random Pointers:" << endl;
 printClonedLinkedList(head);

 // Clone the linked list
 Node* clonedList = cloneLL(head);

 cout << "\nCloned Linked List with Random Pointers:" << endl;
 printClonedLinkedList(clonedList);

 return 0;
}

```

---

## Complexity

- **Time Complexity:** O(N)
- **Space Complexity:** O(1)

# **Recursion**

# 1. Recursive Implementation of atoi()

---

You are given a string  $s$  that may contain leading spaces, an optional sign (+ or -), digits, and other characters.

Your task is to convert this string into a **32-bit signed integer**, exactly like `atoi()`, using **recursion**.

Rules to follow:

- Ignore leading whitespaces
- Check and apply sign (+ or -)
- Read digits until a non-digit character is found
- Stop conversion at the first non-digit
- Clamp the result within [-2147483648, 2147483647]

## Example

Input: " -12345"

Output: -12345

## Approach

We convert the string to an integer using recursion while following these steps:

1. Skip leading whitespaces.
2. Detect sign (+ or -).
3. Recursively read digit characters and build the number.
4. Stop when a non-digit character is found.

5. Clamp the result within 32-bit signed integer limits.
- 

## Code (Recursive)

```
#include <bits/stdc++.h>
using namespace std;

int INT_MIN_VAL = -2147483648;
int INT_MAX_VAL = 2147483647;

// Recursive helper function
int solve(string &s, int idx, long long num, int sign) {
 // Stop if index out of range or character is not a digit
 if (idx >= s.size() || !isdigit(s[idx])) {
 return sign * num;
 }

 // Add current digit
 num = num * 10 + (s[idx] - '0');

 // Handle overflow
 if (sign * num <= INT_MIN_VAL) return INT_MIN_VAL;
 if (sign * num >= INT_MAX_VAL) return INT_MAX_VAL;

 // Recurse to next character
 return solve(s, idx + 1, num, sign);
}

int myAtoi(string s) {
 int i = 0;

 // Skip leading spaces
 while (i < s.size() && s[i] == ' ') i++;

 // Handle sign
 int sign = 1;
 if (i < s.size() && (s[i] == '+' || s[i] == '-')) {
```

```

 if (s[i] == '-') sign = -1;
 i++;
 }

 // Start recursive digit processing
 return solve(s, i, 0, sign);
}

int main() {
 string s1 = " -12345";
 string s2 = "4193 with words";

 cout << myAtoi(s1) << endl; // -12345
 cout << myAtoi(s2) << endl; // 4193

 return 0;
}

```

---

## Complexity Analysis

- **Time Complexity:**  $O(n)$  — each character is processed once
- **Space Complexity:**  $O(n)$  — recursion stack in the worst case

## 2. Implement Pow(x, n) | X raised to the power N

### Explanation

You are given a base x (double) and an integer exponent n.

Your task is to compute  $x^n$ .

Important points:

- If n is **positive**, multiply x by itself n times
- If n is **negative**, result becomes  $1 / (x^n)$
- If n = 0, the answer is always 1
- You must handle large values of n safely

### Example

Input:  $x = 2.0$ ,  $n = -2$

Output: 0.25

Explanation:  $2^{-2} = 1 / (2^2) = 1 / 4$

---

## Brute Force Approach

### Idea

Multiply x with itself  $|n|$  times.

If n is negative, first convert  $x = 1/x$  and make n positive.

---

### Code (Brute Force)

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 double myPow(double x, int n) {
 if (n == 0) return 1.0;

 long long exp = n;
 if (exp < 0) {
 x = 1 / x;
 exp = -exp;
 }

 }
}
```

```

 double ans = 1.0;
 for (long long i = 0; i < exp; i++) {
 ans *= x;
 }
 return ans;
 }

};

int main() {
 Solution sol;
 cout << sol.myPow(2.0, 10) << endl;
 cout << sol.myPow(2.0, -2) << endl;
 return 0;
}

```

## Complexity

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

## Optimal Approach (Fast Power / Binary Exponentiation)

### Idea

Instead of multiplying  $x$  repeatedly, reduce the problem size:

- If  $n$  is even  $\rightarrow x^n = (x^2)^{n/2}$
- If  $n$  is odd  $\rightarrow x^n = x * x^{n-1}$

This reduces calculations drastically.

---

```
#include <bits/stdc++.h>
using namespace std;
```

```

class Solution {
private:
 double power(double x, long long n) {
 if (n == 0) return 1.0;

 double half = power(x, n / 2);

 if (n % 2 == 0)
 return half * half;
 else
 return half * half * x;
 }

public:
 double myPow(double x, int n) {
 long long exp = n;
 if (exp < 0)
 return 1.0 / power(x, -exp);

 return power(x, exp);
 }
};

int main() {
 Solution sol;
 cout << sol.myPow(2.0, 10) << endl;
 cout << sol.myPow(2.0, -2) << endl;
 return 0;
}

```

## Complexity

- Time:  $O(\log n)$
- Space:  $O(\log n)$  (recursion stack)

# 3. Count Good Numbers

## Explanation

A digit string of length  $n$  is called **good** if:

- Digits at **even indices (0-based)** are **even digits**  $\rightarrow \{0, 2, 4, 6, 8\} \rightarrow 5 \text{ choices}$
- Digits at **odd indices** are **prime digits**  $\rightarrow \{2, 3, 5, 7\} \rightarrow 4 \text{ choices}$

You must count how many such strings of length  $n$  exist.

Since the answer can be very large, return it modulo  $10^9 + 7$ .

## Example

Input:  $n = 2$

Index 0  $\rightarrow$  even  $\rightarrow 5$  choices

Index 1  $\rightarrow$  odd  $\rightarrow 4$  choices

Total  $= 5 \times 4 = 20$

---

## Key Observation

- Number of **even positions**  $= (n + 1) / 2$
- Number of **odd positions**  $= n / 2$

For each:

- Even positions  $\rightarrow 5^{(\text{evenCount})}$
- Odd positions  $\rightarrow 4^{(\text{oddCount})}$

Final Answer:

$(5^{(\text{evenCount})} * 4^{(\text{oddCount})}) \% \text{ MOD}$

To compute large powers efficiently, we use **Binary Exponentiation**.

---

## Optimal Approach (Fast Power)

### Code

```
#include <bits/stdc++.h>

using namespace std;

class Solution {

public:

 static const long long MOD = 1000000007;

 // Fast power function

 long long power(long long base, long long exp) {

 long long result = 1;

 base %= MOD;

 while (exp > 0) {

 if (exp & 1)

 result = (result * base) % MOD;

 base = (base * base) % MOD;

 exp >>= 1;

 }

 }

}
```

```

 return result;
}

long long countGoodNumbers(long long n) {
 long long evenCount = (n + 1) / 2;
 long long oddCount = n / 2;

 long long evenWays = power(5, evenCount);
 long long oddWays = power(4, oddCount);

 return (evenWays * oddWays) % MOD;
};

int main() {
 Solution sol;
 cout << sol.countGoodNumbers(1) << endl;
 cout << sol.countGoodNumbers(2) << endl;
 return 0;
}

```

---

# Complexity Analysis

- **Time Complexity:**  $O(\log n)$   
(Binary exponentiation for power calculation)
- **Space Complexity:**  $O(1)$   
(Only constant extra variables used)

## 4. Sort a Stack (Using Recursion)

### Explanation

You are given a stack of integers and you need to sort it in **descending order**, meaning the **largest element should be on the top** of the stack.

You are **not allowed to use loops or any sorting algorithms**. Only **recursion** and basic stack operations (push, pop, top, empty) are allowed.

### Idea

We solve this in two recursive steps:

#### 1. **sortStack**

- Remove the top element.
- Recursively sort the remaining stack.
- Insert the removed element back into the sorted stack at its correct position.

#### 2. **insert**

- Inserts an element into a stack that is already sorted in descending order.
- If the stack is empty or the current top is smaller than or equal to the element, push it.

- Otherwise, pop the top, recursively insert the element, and then push the popped value back.

This way, the stack gets sorted when recursion unwinds.

---

## Code

```
#include <bits/stdc++.h>
using namespace std;

// Inserts an element into the stack in descending order
void insert(stack<int>& s, int x) {
 // If stack is empty or correct position found
 if (s.empty() || s.top() <= x) {
 s.push(x);
 return;
 }

 int temp = s.top();
 s.pop();

 insert(s, x);

 s.push(temp);
}

// Sorts the stack using recursion
void sortStack(stack<int>& s) {
 // Base case
 if (s.empty()) return;

 int temp = s.top();
 s.pop();

 sortStack(s);

 insert(s, temp);
}
```

```

int main() {
 stack<int> s;
 s.push(4);
 s.push(1);
 s.push(3);
 s.push(2);

 sortStack(s);

 // Printing sorted stack (top to bottom)
 while (!s.empty()) {
 cout << s.top() << " ";
 s.pop();
 }

 return 0;
}

```

---

### Example Dry Run (Input: 4 1 3 2)

- Stack is recursively emptied
  - Elements are inserted back in correct descending position
  - Final stack (top → bottom): 4 3 2 1
- 

### Complexity Analysis

- **Time Complexity:**  $O(n^2)$   
Each insertion may take up to  $O(n)$  and is done for  $n$  elements.
- **Space Complexity:**  $O(n)$   
Due to recursion stack.

# 5. Reverse a Stack Using Recursion

## Explanation

You are given a stack of integers and your task is to **reverse the stack in-place using recursion**.

You are **not allowed to use loops or any extra data structures** like arrays, vectors, or queues.

Only standard stack operations (push, pop, top, empty) and recursion are allowed.

## Key Idea

To reverse a stack using recursion, we break the problem into two parts:

### 1. **reverseStack**

- Remove the top element.
- Recursively reverse the remaining stack.
- Insert the removed element at the **bottom** of the stack.

### 2. **insertAtBottom**

- Inserts a given element at the bottom of the stack using recursion.
- If the stack is empty, push the element.
- Otherwise, pop the top, recursively insert at bottom, then push the popped element back.

By repeating this process, the stack gets reversed when recursion unwinds.

---

## Example

Input stack (top → bottom):

[4, 1, 3, 2]

After reversing:

[2, 3, 1, 4]

---

## Code

```
#include <bits/stdc++.h>
using namespace std;

// Function to insert an element at the bottom of the stack
void insertAtBottom(stack<int> &st, int val) {
 // Base case: stack is empty
 if (st.empty()) {
 st.push(val);
 return;
 }

 int topVal = st.top();
 st.pop();

 insertAtBottom(st, val);

 st.push(topVal);
}

// Function to reverse the stack using recursion
void reverseStack(stack<int> &st) {
 // Base case
 if (st.empty()) return;

 int topVal = st.top();
 st.pop();

 reverseStack(st);

 insertAtBottom(st, topVal);
}

int main() {
 stack<int> st;
 st.push(4);
 st.push(1);
 st.push(3);
```

```

 st.push(2);

 reverseStack(st);

 // Print reversed stack (top to bottom)
 while (!st.empty()) {
 cout << st.top() << " ";
 st.pop();
 }

 return 0;
}

```

---

## Complexity Analysis

- **Time Complexity:**  $O(n^2)$   
Each element is inserted at the bottom, which takes  $O(n)$  time, for  $n$  elements.
- **Space Complexity:**  $O(n)$   
Due to recursive call stack.

# 6. Generate All Binary Strings (No Consecutive 1s)

### Explanation

You are given an integer  $n$ . You need to generate **all binary strings of length  $n$**  such that **no two 1s are adjacent**.

The result should be in **lexicographically increasing order**.

A binary string contains only characters '`0`' and '`1`'.

## Key Observation

- At any position, you can **always place '0'**.
- You can place '**1**' **only if the previous character is not '1'**.
- Using recursion, we build the string character by character until its length becomes n.

## Example

For n = 3, valid strings are:

000, 001, 010, 100, 101

Strings like 011 or 110 are not allowed because they contain consecutive 1s.

---

## Approach

1. Use a recursive function with:
    - current string being built
    - result container
  2. If the current string length becomes n, store it.
  3. Always try adding '**0**'.
  4. Add '**1**' only if the last character is not '**1**'.
- 

## Code

```
#include <bits/stdc++.h>
using namespace std;

void generate(int n, string curr, vector<string>& result) {
 // Base case: required length reached
 if (curr.length() == n) {
 result.push_back(curr);
 return;
 }
}
```

```

 }

 // Always allowed to add '0'
 generate(n, curr + "0", result);

 // Add '1' only if previous character is not '1'
 if (curr.empty() || curr.back() != '1') {
 generate(n, curr + "1", result);
 }
}

int main() {
 int n = 3;
 vector<string> result;

 generate(n, "", result);

 for (string &s : result) {
 cout << s << " ";
 }

 return 0;
}

```

---

## Complexity Analysis

- **Time Complexity:**  $O(2^n)$   
In the worst case, each position branches into two recursive calls.
- **Space Complexity:**  $O(n)$   
Due to recursion depth (call stack).

# 7. Generate Parentheses

## Explanation

You are given an integer  $n$ , representing the number of pairs of parentheses.

Your task is to generate **all possible combinations of well-formed parentheses** using exactly  $n$  opening ' $($ ' and  $n$  closing ' $)$ ' brackets.

A parentheses string is **well-formed** if:

- At any point, the number of ' $)$ ' does not exceed ' $($ '.
- At the end, the total number of ' $($ ' equals the number of ' $)$ '.

## Example

For  $n = 3$ , valid combinations are:

((()))  
((())())  
((())())  
((())())  
((())())

---

## Brute Force Approach

### Idea

Generate **all possible strings of length  $2n$**  using ' $($ ' and ' $)$ '.

For each generated string, check whether it is valid using a balance counter.

### Algorithm

1. Generate all sequences of length  $2n$ .
2. For each sequence:
  - Traverse the string.

- Increase balance for '(', decrease for ')'.
  - If balance becomes negative at any point → invalid.
  - At the end, balance must be 0.
3. Store only valid sequences.

## Code

```
#include <bits/stdc++.h>
using namespace std;

// Function to check if parentheses string is valid
bool isValid(string s) {
 int balance = 0;
 for (char c : s) {
 if (c == '(') balance++;
 else balance--;
 if (balance < 0) return false;
 }
 return balance == 0;
}

void generateAll(string curr, int n, vector<string>& res) {
 if (curr.length() == 2 * n) {
 if (isValid(curr))
 res.push_back(curr);
 return;
 }
 generateAll(curr + '(', n, res);
 generateAll(curr + ')', n, res);
}

vector<string> generateParenthesis(int n) {
 vector<string> res;
 generateAll("", n, res);
 return res;
}
```

```

int main() {
 int n = 3;
 vector<string> result = generateParenthesis(n);
 for (string s : result)
 cout << s << endl;
 return 0;
}

```

## Complexity Analysis

- **Time Complexity:**  $O(2^{(2n)} \times n)$   
All possible strings are generated and each is validated.
  - **Space Complexity:**  $O(n)$  recursion depth (excluding result storage).
- 

## Optimal Approach (Backtracking)

### Idea

Instead of generating invalid strings and filtering them, **build only valid sequences**:

- You can add ' (' if  $\text{open} < n$ .
- You can add ')' only if  $\text{close} < \text{open}$ .

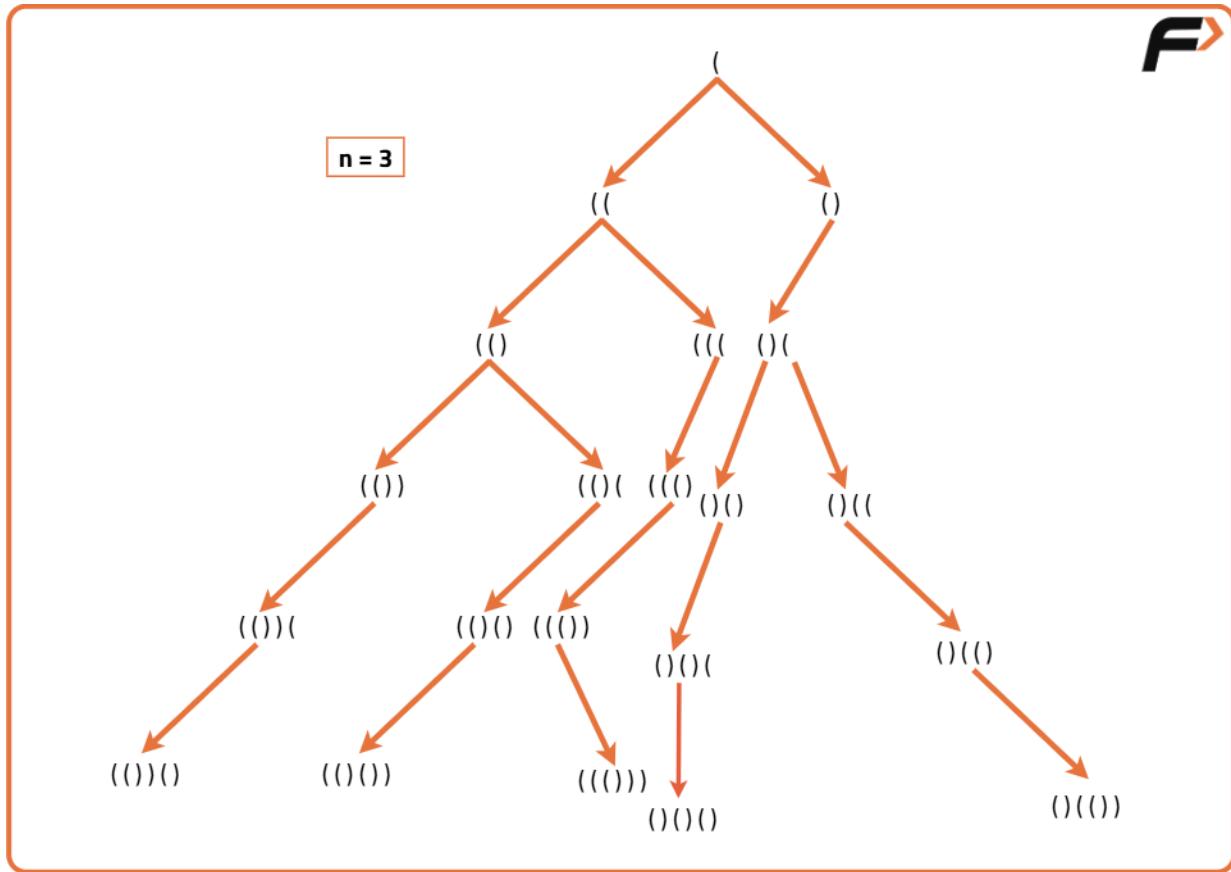
This ensures correctness at every step.

### Algorithm

1. Start with an empty string.
2. Maintain counts of open and close.
3. If  $\text{open} < n$ , add ' (' and recurse.

4. If close < open, add ')' and recurse.

5. When string length becomes  $2n$ , store it.



## Code

```
#include <bits/stdc++.h>
using namespace std;

void backtrack(string curr, int open, int close, int n,
vector<string>& res) {
 if (curr.length() == 2 * n) {
 res.push_back(curr);
 return;
 }
 if (open < n)
 backtrack(curr + '(', open + 1, close, n, res);
 if (close < open)
 backtrack(curr + ')', open, close + 1, n, res);
}
```

```

 backtrack(curr + ')', open, close + 1, n, res);
 }

vector<string> generateParenthesis(int n) {
 vector<string> res;
 backtrack("", 0, 0, n, res);
 return res;
}

int main() {
 int n = 3;
 vector<string> result = generateParenthesis(n);
 for (string s : result)
 cout << s << endl;
 return 0;
}

```

## Complexity Analysis

- **Time Complexity:**  $O(C(n) \times n)$   
where  $C(n)$  is the nth Catalan number (number of valid sequences).
- **Space Complexity:**  $O(n)$  recursion depth  
plus space to store all valid combinations.

# 8. Power Set: Print all the possible subsequences of a String

## Explanation

You are given a string str.

A **subsequence** is formed by choosing **some characters** from the string **without changing**

**their relative order.**

You can either **include or exclude** each character, so for a string of length  $n$ , there are  $2^n$  possible subsequences (including the empty one).

Empty subsequence is usually generated internally, but in many outputs it is skipped.

### **Example**

Input: str = "abc"

Possible subsequences:

a, ab, abc, ac, b, bc, c

---

## **Approach 1: Bit Manipulation**

### **Explanation**

Each character has two choices:

- 0 → exclude
- 1 → include

So every subsequence can be represented using a binary number of length  $n$ .

For "abc" ( $n = 3$ ):

001 → c  
010 → b  
011 → bc  
101 → ac

### **Algorithm**

1. Let  $n$  be the length of the string.
2. Total subsequences =  $2^n$ .
3. Loop from 0 to  $2^n - 1$ .

4. For each number, check its bits:

- o If the i-th bit is set, include  $s[i]$ .

5. Store the generated subsequence.



| Decimal | Binary | Chosen Character | Subsequence |
|---------|--------|------------------|-------------|
| 0       | 000    | none             | " "         |
| 1       | 001    | 'c'              | "c"         |
| 2       | 010    | 'b'              | "b"         |
| 3       | 011    | 'b', 'c'         | "bc"        |
| 4       | 100    | 'a'              | "a"         |
| 5       | 101    | 'a', 'c'         | "ac"        |
| 6       | 110    | 'a', 'b'         | "ab"        |
| 7       | 111    | 'a', 'b', 'c'    | "abc"       |

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 vector<string> getSubsequences(string s) {
 int n = s.size();
 int total = 1 << n;
 vector<string> ans;

 for (int mask = 1; mask < total; mask++) {
 string curr = "";
 for (int i = 0; i < n; i++) {
 if (mask & (1 << i)) {
 // If i-th bit of mask is set, include s[i]
 curr.push_back(s[i]);
 }
 }
 }
 }
}
```

```

 ans.push_back(curr);
 }
 return ans;
}
};

int main() {
 string s = "abc";
 Solution sol;

 vector<string> res = sol.getSubsequences(s);
 for (string x : res) {
 cout << x << endl;
 }
 return 0;
}

```

## Complexity Analysis

- **Time Complexity:**  $O(n * 2^n)$
  - **Space Complexity:**  $O(n * 2^n)$  (to store all subsequences)
- 

## Approach 2: Recursion (Include / Exclude)

### Explanation

For every character, you have two choices:

- Exclude it
- Include it

This forms a binary recursion tree.

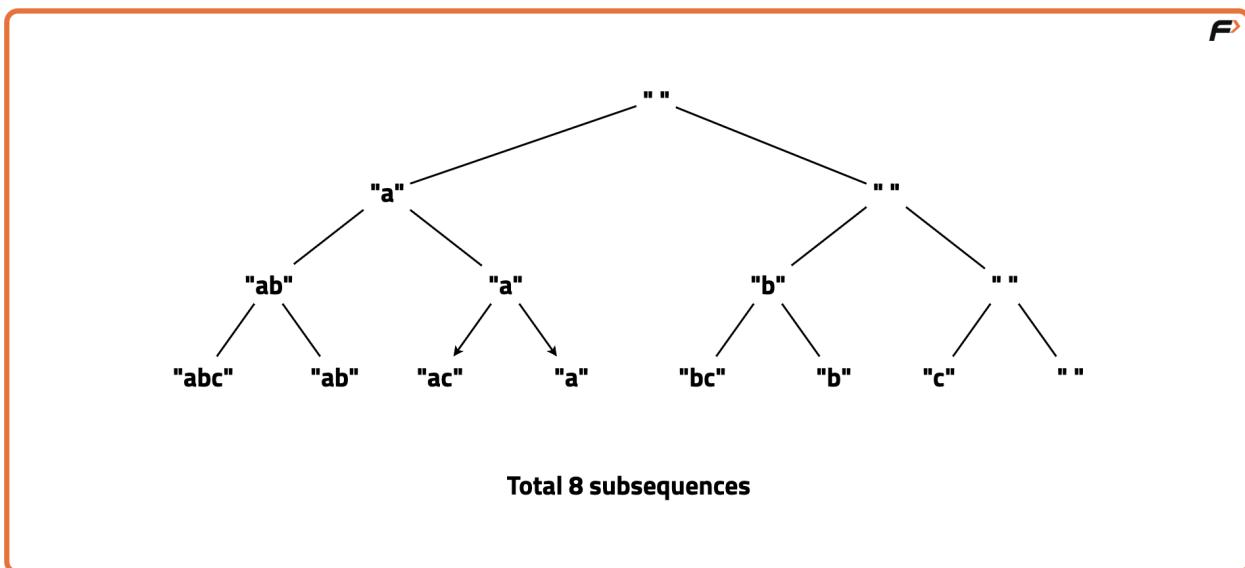
At each index:

- Move forward **without** including the character.
- Move forward **after including** the character.

When you reach the end of the string, the current string is a valid subsequence.

## Algorithm

1. Start from index 0 with an empty string.
2. At each index:
  - Recurse without taking the character.
  - Recurse after taking the character.
3. When index reaches n, store the current subsequence.



## Code

```

#include <bits/stdc++.h>
using namespace std;

class Solution {

```

```

public:
 void helper(string &s, int idx, string &curr, vector<string> &ans)
{
 if (idx == s.size()) {
 if (!curr.empty())
 ans.push_back(curr);
 return;
 }

 // Exclude current character
 helper(s, idx + 1, curr, ans);

 // Include current character
 curr.push_back(s[idx]);
 helper(s, idx + 1, curr, ans);
 curr.pop_back();
}

vector<string> getSubsequences(string s) {
 vector<string> ans;
 string curr = "";
 helper(s, 0, curr, ans);
 return ans;
}
};

int main() {
 string s = "abc";
 Solution sol;

 vector<string> res = sol.getSubsequences(s);
 for (string x : res) {
 cout << x << endl;
 }
 return 0;
}

```

## Complexity Analysis

- **Time Complexity:**  $O(n * 2^n)$
- **Space Complexity:**  $O(n)$  recursion stack  
 $(O(n * 2^n)$  including result storage)

## Learn All Patterns of Subsequences (Theory)

### 9. What are Subsequences?

A **subsequence** is a sequence derived from another sequence by deleting some or no elements without changing the order of the remaining elements. Unlike substrings, the elements of a subsequence are not required to occupy consecutive positions in the original sequence.

For example, in the string "abc", the subsequences include: "a", "b", "c", "ab", "ac", "bc", and "abc".

### Total Subsequences of a String

For a string of length  $n$ , the total number of subsequences (excluding the empty one) is  $2^n - 1$ .

This is because each character has 2 options: either be included or not. Thus,  $2 \times 2 \times \dots$  (n times) =  $2^n$ . We subtract 1 to exclude the empty subsequence if not required.

### Subsequence vs Substring

| Subsequence                            | Substring          |
|----------------------------------------|--------------------|
| Can skip characters, but order matters | Must be contiguous |

|                                   |                                    |
|-----------------------------------|------------------------------------|
| Number of subsequences is $2^n$   | Number of substrings is $n(n+1)/2$ |
| "ace" is a subsequence of "abcde" | "abc" is a substring of "abcde"    |

## Patterns and Problems Based on Subsequences

- Generate All Subsequences – Basic recursion or backtracking
- Count Subsequences with Specific Property – e.g. sum = target
- Longest Increasing Subsequence (LIS) – Classic DP problem
- Subsequences with K elements – Use recursion with element count
- Print all subsequences with sum = K – Variation of subset sum

## Recursive Structure for Generating Subsequences

We use the idea of "pick or not pick" for each character in the string or element in the array.

```
function recurse(index, current):
 if index == n:
 print current
 return
 // Pick the current element
 recurse(index + 1, current + arr[index])
 // Do not pick the current element
 recurse(index + 1, current)
```

## Use Cases in Problem Solving

- Subset sum problems
- Count/print subsequences with a given sum
- Dynamic programming on subsequences (e.g. LIS, LCS, Count Palindromic Subsequences)
- Bitmasking based optimizations

# Time & Space Complexity

| Operation                 | Time Complexity           | Space Complexity |
|---------------------------|---------------------------|------------------|
| Generate All Subsequences | $O(2^n)$                  | $O(n)$           |
| Check for specific sum    | $O(2^n)$                  | $O(n)$           |
| LIS (DP)                  | $O(n^2)$ or $O(n \log n)$ | $O(n)$           |

**Note:** Generating all subsequences has exponential complexity. Use it wisely for small  $n$  (usually  $\leq 20$ ).

## Conclusion

Understanding subsequences and the ability to manipulate them through recursion and dynamic programming unlocks solutions to a wide variety of problems. From classic LIS and subset-sum to interview questions on palindromic subsequences, the concept is foundational in problem-solving.

## 10. Count all subsequences with sum K

### Explanation

You are given an integer array `nums` and an integer `k`.

Your task is to count **all non-empty subsequences** whose **sum of elements is exactly equal to k**.

A **subsequence** means:

- You can choose or skip any element
- Order must remain the same
- Elements do **not** have to be contiguous

So for every element, you always have **two choices**:

1. Include it in the subsequence
2. Exclude it from the subsequence

This naturally leads to a **recursive include / exclude approach**.

---

## Example 1

Input:

```
nums = [4, 9, 2, 5, 1], k = 10
```

Valid subsequences:

```
[9, 1]
[4, 5, 1]
```

Output:

```
2
```

---

## Example 2

Input:

```
nums = [4, 2, 10, 5, 1, 3], k = 5
```

Valid subsequences:

[4, 1]

[2, 3]

[5]

Output:

3

---

## Recursive Approach (Include / Exclude)

### Explanation of the idea

We process the array from left to right.

At every index:

- Either **include** the current element in the sum
- Or **exclude** it and move forward

We stop recursion when:

- The sum becomes 0 → one valid subsequence found
- The sum becomes negative → invalid path
- We reach the end of the array → no valid subsequence

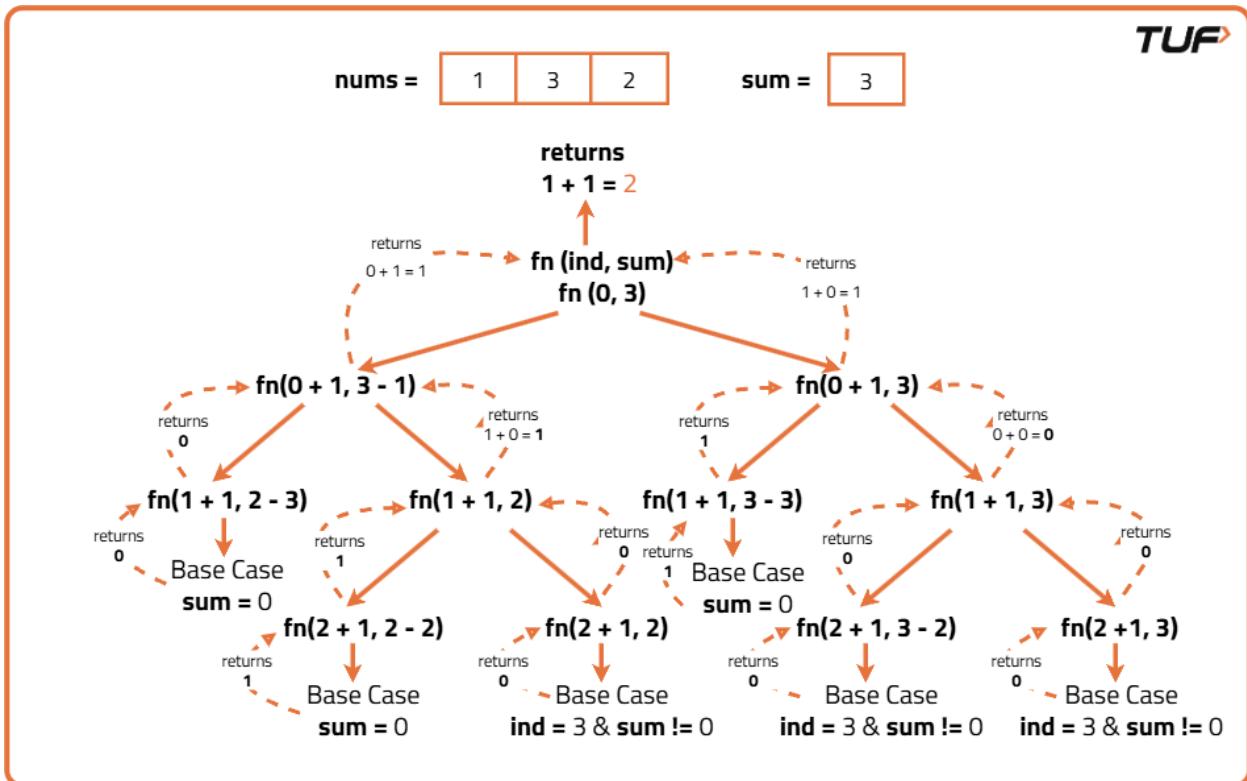
The total answer is the **sum of both recursive choices**.

---

## Algorithm

1. Start recursion from index 0 with target sum k

2. At each index:
  - o Include the current element → reduce sum
  - o Exclude the current element → keep sum same
3. If sum becomes 0, return 1
4. If sum < 0 or index reaches array size, return 0
5. Add results of both choices



## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
```

```

private:
 int solve(int index, int sum, vector<int>& nums) {
 // If exact sum achieved
 if (sum == 0) return 1;

 // If sum becomes negative or index out of range
 if (sum < 0 || index == nums.size()) return 0;

 // Include current element
 int take = solve(index + 1, sum - nums[index], nums);

 // Exclude current element
 int notTake = solve(index + 1, sum, nums);

 return take + notTake;
 }

public:
 int countSubsequenceWithTargetSum(vector<int>& nums, int k) {
 return solve(0, k, nums);
 }
};

int main() {
 Solution sol;
 vector<int> nums = {4, 9, 2, 5, 1};
 int k = 10;

 cout << sol.countSubsequenceWithTargetSum(nums, k) << endl;
 return 0;
}

```

---

## Complexity Analysis

- **Time Complexity:**  $O(2^n)$   
Every element has two choices (include / exclude)

- **Space Complexity:**  $O(n)$   
Due to recursion stack depth

# 11. Check if there exists a subsequence with sum K

## Explanation

You are given an integer array `nums` and an integer `k`.

You need to check **whether at least one non-empty subsequence exists whose sum is exactly k.**

A **subsequence** means:

- You may choose or skip any element
- Order remains the same
- Elements do not need to be contiguous

For every element, you always have **two choices**:

- include it in the sum
- exclude it from the sum

If **any path** leads to sum `k`, return `true`, otherwise return `false`.

---

## Example 1

Input:

```
nums = [1, 2, 3, 4, 5], k = 8
```

Valid subsequences:

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

Output:

```
true
```

---

## Example 2

Input:

```
nums = [4, 3, 9, 2], k = 10
```

No subsequence forms sum 10

Output:

```
false
```

---

## Recursive Include / Exclude Approach

### Idea

We process the array index by index.

At each index:

- include the current element → reduce sum
- exclude the current element → sum remains same

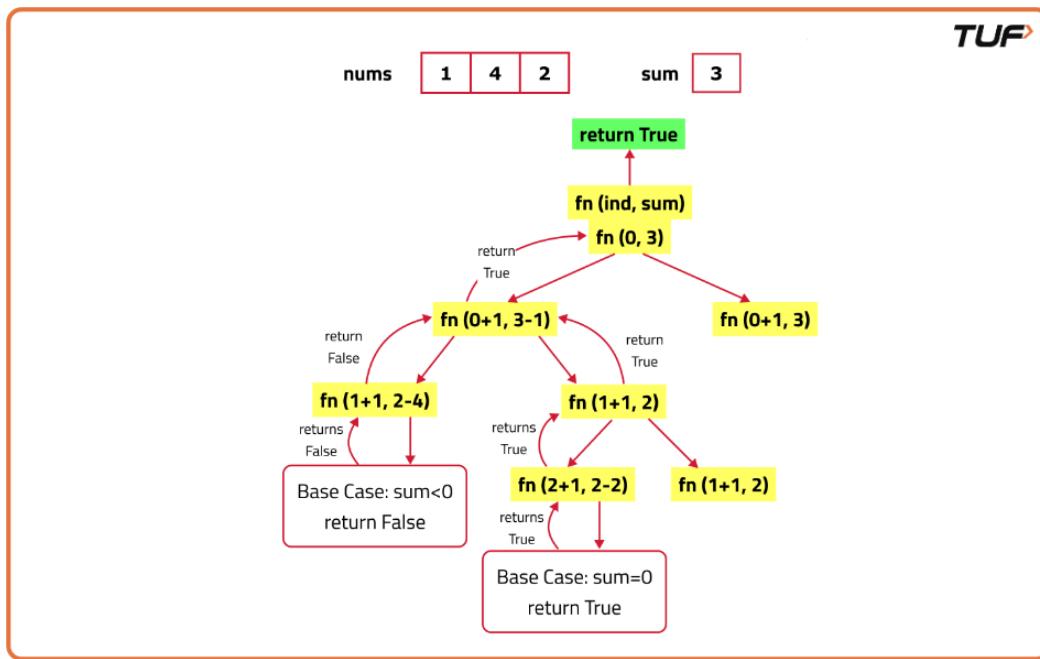
We stop recursion when:

- sum becomes 0 → subsequence found

- index reaches end → no valid subsequence on this path
- 

## Algorithm

1. Start recursion from index 0 with target sum k
2. At each index:
  - include current element
  - exclude current element
3. If sum becomes 0, return true
4. If array ends, return false
5. Return true if **any path** returns true




---

## Code

```
#include <bits/stdc++.h>
using namespace std;
```

```

class Solution {
private:
 bool solve(int index, int sum, vector<int>& nums) {
 // If sum becomes 0, subsequence exists
 if (sum == 0) return true;

 // If index is out of bounds
 if (index == nums.size()) return false;

 // Include current element OR exclude it
 return solve(index + 1, sum - nums[index], nums) ||
 solve(index + 1, sum, nums);
 }

public:
 bool checkSubsequenceSum(vector<int>& nums, int k) {
 return solve(0, k, nums);
 }
};

int main() {
 Solution sol;
 vector<int> nums = {1, 2, 3, 4, 5};
 int k = 8;

 cout << sol.checkSubsequenceSum(nums, k);
 return 0;
}

```

---

## Complexity Analysis

- **Time Complexity:**  $O(2^n)$   
Each element has two choices (include / exclude)
- **Space Complexity:**  $O(n)$   
Due to recursion stack depth

# 12. Combination Sum – I

You are given an array of **distinct integers** and a **target** value.

Your task is to find **all unique combinations** of numbers such that the **sum equals the target**.

Rules:

- You can **use the same number unlimited times**
- Order of numbers inside a combination does not matter
- Each combination must be unique based on frequency of elements

This is a classic **recursion + backtracking** problem.

---

## Example 1

Input:

```
array = [2, 3, 6, 7], target = 7
```

Valid combinations:

```
[2, 2, 3]
[7]
```

---

## Example 2

Input:

```
array = [2], target = 1
```

Output:

```
[]
```

---

## Recursive Pick / Not-Pick Approach

### Idea

At every index, we decide:

- **Pick** the current element (stay on same index because reuse is allowed)
- **Not pick** the current element (move to next index)

We keep track of:

- current index
- remaining target
- current combination (ds)

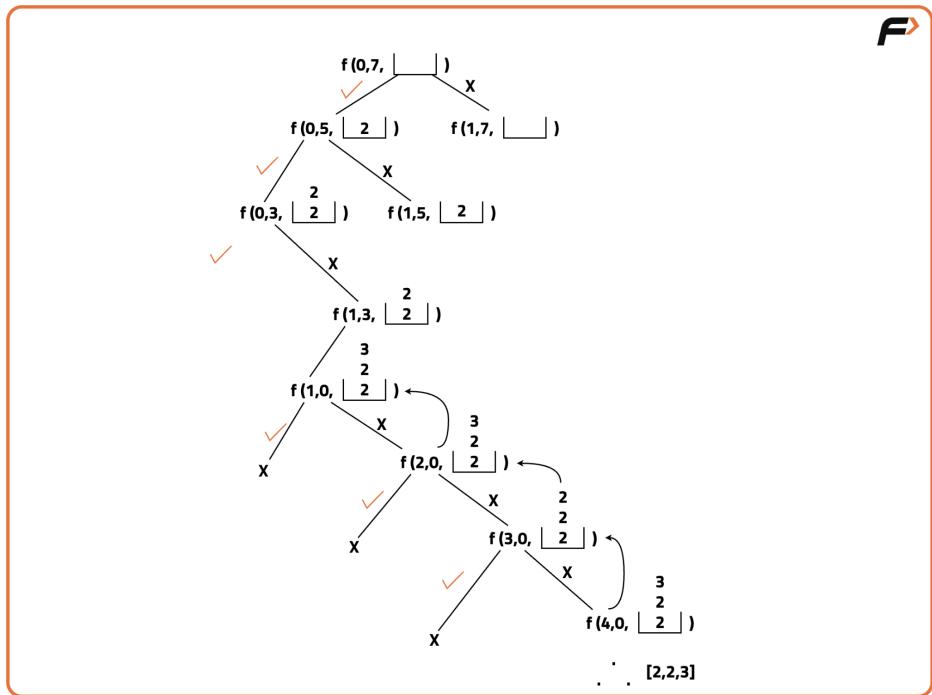
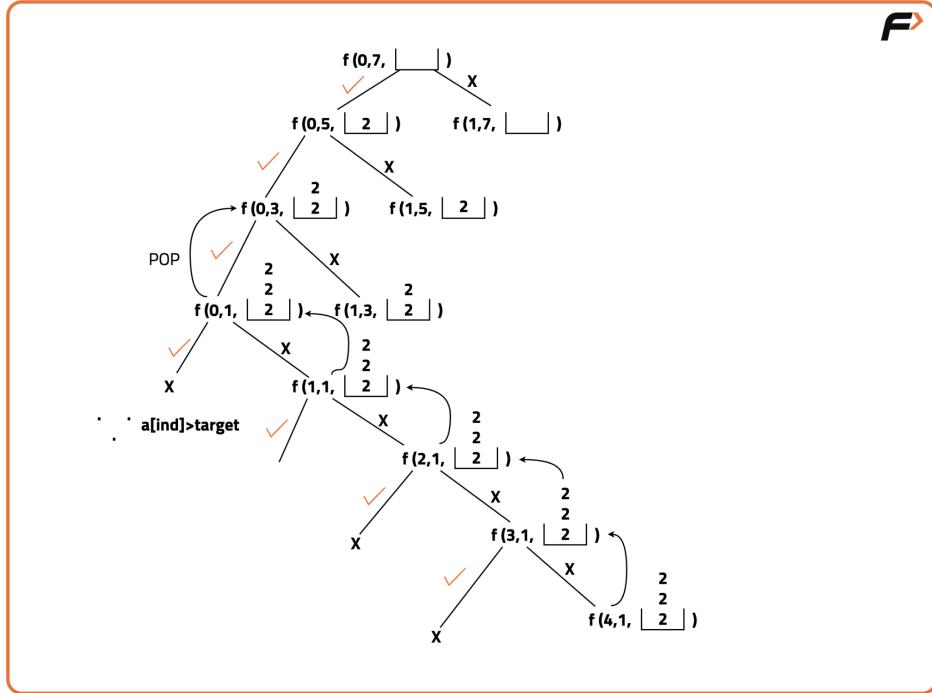
When:

- `target == 0` → valid combination found
  - `index == array.size()` → stop recursion
- 

## Algorithm

1. Start recursion from index 0
2. If current element  $\leq$  target:
  - pick it
  - reduce target
  - stay on same index
3. Always explore the not-pick case:

- o move to next index
4. Backtrack after each recursive call
5. Collect all valid combinations



---

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
private:
 void findCombination(int index, int target, vector<int>& arr,
 vector<vector<int>>& ans, vector<int>& ds) {

 // If we have processed all elements
 if (index == arr.size()) {
 if (target == 0) {
 ans.push_back(ds);
 }
 return;
 }

 // Pick the current element if possible
 if (arr[index] <= target) {
 ds.push_back(arr[index]);
 findCombination(index, target - arr[index], arr, ans, ds);
 ds.pop_back(); // backtrack
 }

 // Do not pick the current element
 findCombination(index + 1, target, arr, ans, ds);
 }

public:
 vector<vector<int>> combinationSum(vector<int>& candidates, int target) {
 vector<vector<int>> ans;
 vector<int> ds;
 findCombination(0, target, candidates, ans, ds);
 return ans;
 }
}
```

```

};

int main() {
 Solution obj;
 vector<int> candidates = {2, 3, 6, 7};
 int target = 7;

 vector<vector<int>> result = obj.combinationSum(candidates,
target);

 for (auto &comb : result) {
 for (int x : comb) cout << x << " ";
 cout << endl;
 }
 return 0;
}

```

---

## Complexity Analysis

- **Time Complexity:** Exponential (depends on number of combinations formed)
- **Space Complexity:**
  - $O(k \times x)$  to store results
  - $O(k)$  recursion depth  
( $k$  = average length of a combination,  $x$  = number of combinations)

This approach strictly follows **pick / not-pick recursion**, allows reuse of elements, and avoids duplicate combinations.

# 13. Combination Sum II – Find All Unique Combinations

## Explanation

You are given an array of integers **candidates** and a target value **target**.

Your task is to find **all unique combinations** where the chosen numbers sum to target.

Rules:

- **Each number can be used at most once** in a combination
- The input array **may contain duplicates**
- The result must not contain duplicate combinations

This problem is very similar to **Combination Sum I**, but with **two key differences**:

1. An element can be picked **only once**
  2. Duplicate combinations must be **avoided**
- 

## Example 1

Input:

```
candidates = [10,1,2,7,6,1,5], target = 8
```

Output:

```
[[1,1,6], [1,2,5], [1,7], [2,6]]
```

---

## Example 2

Input:

```
candidates = [2,5,2,1,2], target = 5
```

Output:

```
[[1,2,2], [5]]
```

---

## Core Idea (Backtracking with Duplicate Handling)

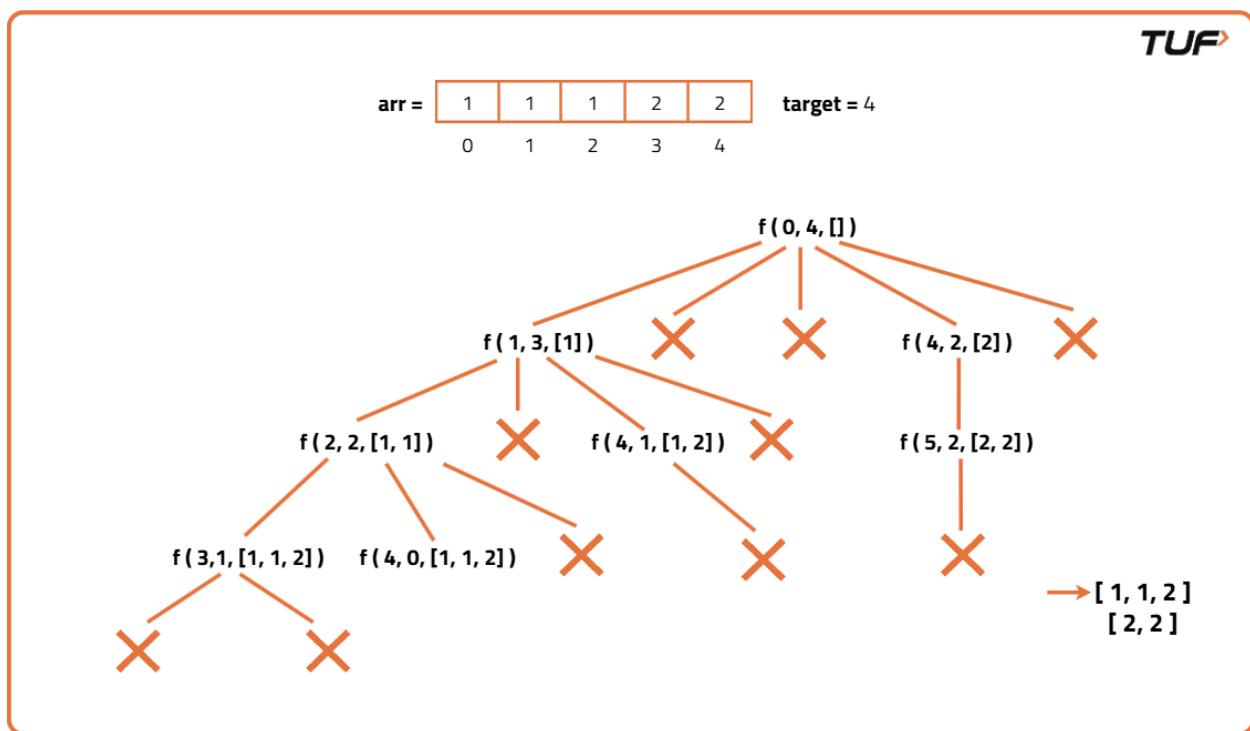
We still use **recursion + backtracking**, but with these rules:

1. **Sort the array first**
    - Helps in skipping duplicates
    - Helps in early stopping when element > target
  2. Use a loop instead of pick / not-pick recursion
    - At every recursive level, try all possible elements starting from `index`
  3. **Skip duplicates**
    - If `arr[i] == arr[i-1]` and `i > index`, skip it
  4. Move to **next index (`i + 1`)** after picking an element
    - Ensures each number is used only once
- 

## Algorithm

1. Sort the candidates array
2. Start recursion from index 0
3. For every index `i`:

- Skip duplicate values
  - Stop if  $\text{arr}[i] > \text{target}$
  - Pick  $\text{arr}[i]$  and recurse with:
    - $\text{target} - \text{arr}[i]$
    - $i + 1$
4. If  $\text{target} == 0$ , store the combination
5. Backtrack after recursion



## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
```

```

private:
 void findCombination(int index, int target, vector<int>& arr,
 vector<vector<int>>& ans, vector<int>& ds) {

 // If target becomes 0, valid combination found
 if (target == 0) {
 ans.push_back(ds);
 return;
 }

 for (int i = index; i < arr.size(); i++) {

 // Skip duplicates
 if (i > index && arr[i] == arr[i - 1]) continue;

 // Early stopping
 if (arr[i] > target) break;

 // Pick current element
 ds.push_back(arr[i]);

 // Move to next index (each element used once)
 findCombination(i + 1, target - arr[i], arr, ans, ds);

 // Backtrack
 ds.pop_back();
 }
 }

public:
 vector<vector<int>> combinationSum2(vector<int>& candidates, int
target) {
 sort(candidates.begin(), candidates.end());
 vector<vector<int>> ans;
 vector<int> ds;
 findCombination(0, target, candidates, ans, ds);
 return ans;
}

```

```

};

int main() {
 Solution sol;
 vector<int> candidates = {10, 1, 2, 7, 6, 1, 5};
 int target = 8;

 vector<vector<int>> result = sol.combinationSum2(candidates,
target);

 for (auto &comb : result) {
 for (int x : comb) cout << x << " ";
 cout << endl;
 }
 return 0;
}

```

---

## Complexity Analysis

- **Time Complexity:**  $O(2^n \times k)$   
Each element can be chosen or skipped, and storing a combination takes  $O(k)$
- **Space Complexity:**
  - $O(k \times x)$  to store all valid combinations
  - $O(k)$  recursion stack  
( $k$  = average length of combination,  $x$  = number of valid combinations)

# 14. Subset Sum – Sum of All Subsets

## Explanation

You are given an array of integers.

Your task is to **generate the sum of every possible subset** of this array and **print all those sums in increasing order**.

A subset can be:

- empty
- contain some elements
- contain all elements

For an array of size N, total subsets =  $2^N$ , so we will also get  $2^N$  subset sums.

---

## Example 1

Input:

```
arr = {5, 2, 1}
```

All subsets:

```
[] → 0
[1] → 1
[2] → 2
[2,1] → 3
[5] → 5
[5,1] → 6
[5,2] → 7
[5,2,1] → 8
```

Output:

```
0 1 2 3 5 6 7 8
```

---

## Example 2

Input:

```
arr = {3, 1, 2}
```

Output:

```
0 1 2 3 3 4 5 6
```

---

## Approach 1: Bitmasking

### Idea

Each subset can be represented using a binary number of length N.

- 0 → element excluded
- 1 → element included

We iterate from 0 to  $(2^N - 1)$  and for each number, calculate the subset sum based on set bits.

---

## Algorithm

1. Let n be the size of the array
2. Iterate mask from 0 to  $(1 \ll n) - 1$
3. For each mask:
  - Initialize sum = 0
  - For each bit i from 0 to n-1

- If  $i$ th bit is set, add  $\text{arr}[i]$  to sum
4. Store sum in result list
  5. Sort the result list
  6. Return / print it
- 

## Code (Bitmasking)

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 vector<int> subsetSums(vector<int>& arr) {
 int n = arr.size();
 vector<int> sums;

 for (int mask = 0; mask < (1 << n); mask++) {
 int sum = 0;
 for (int i = 0; i < n; i++) {
 if (mask & (1 << i)) {
 sum += arr[i];
 }
 }
 sums.push_back(sum);
 }

 sort(sums.begin(), sums.end());
 return sums;
 }
};

int main() {
 Solution sol;
 vector<int> arr = {5, 2, 1};
```

```
vector<int> result = sol.subsetSums(arr);

for (int x : result) cout << x << " ";
return 0;
}
```

---

## Complexity (Bitmasking)

- **Time Complexity:**  $O(2^N * N)$
  - **Space Complexity:**  $O(2^N)$
- 

## Approach 2: Recursion (Pick / Not Pick)

### Idea

At every index, you have **two choices**:

- include the element
- exclude the element

We carry a running sum and once we reach the end of the array, we store that sum.

---

## Algorithm

1. Create a recursive function with parameters:

- index
- currentSum

2. If `index == N`:

- store currentSum
- return

3. Recursive calls:

- include current element → currentSum + arr[index]
- exclude current element → currentSum

4. Start recursion from index 0 and sum 0

5. Sort the result list

---

### **Code (Recursive)**

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 void findSums(int index, int currentSum,
 vector<int>& arr, vector<int>& sums) {

 if (index == arr.size()) {
 sums.push_back(currentSum);
 return;
 }

 // Include current element
 findSums(index + 1, currentSum + arr[index], arr, sums);

 // Exclude current element
 findSums(index + 1, currentSum, arr, sums);
 }

 vector<int> subsetSums(vector<int>& arr) {
 vector<int> sums;
```

```

 findSums(0, 0, arr, sums);
 sort(sums.begin(), sums.end());
 return sums;
 }
};

int main() {
 Solution sol;
 vector<int> arr = {5, 2, 1};
 vector<int> result = sol.subsetSums(arr);

 for (int x : result) cout << x << " ";
 return 0;
}

```

---

## Complexity (Recursive)

- **Time Complexity:**  $O(2^N)$
- **Space Complexity:**
  - $O(2^N)$  for storing sums
  - $O(N)$  recursion stack

# 15. Subset II – Print All Unique Subsets

### Explanation

You are given an integer array `nums` that **may contain duplicate elements**.

Your task is to generate the **power set (all possible subsets)** such that **no duplicate subset appears in the final answer**.

A subset:

- can be empty
- can contain some or all elements
- order inside a subset does not matter

Because duplicates exist in the input array, care must be taken to **avoid printing the same subset more than once**.

---

## Example 1

Input:

```
nums = [1, 2, 2]
```

Output:

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

Explanation:

Although [1, 2] can be formed in multiple ways, it is printed **only once**.

---

## Example 2

Input:

```
nums = [1]
```

Output:

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

---

# Brute Force Approach

## Idea

Generate **all possible subsets** using recursion (pick / not pick).

Since duplicates will appear, store all subsets inside a **set**, which automatically removes duplicates.

---

## Algorithm

1. Sort the array (for consistent subset formation)
  2. Use recursion to generate all subsets:
    - pick the current element
    - do not pick the current element
  3. When the index reaches the end:
    - insert the current subset into a set
  4. Convert the set into a vector and return it
- 

## Code (Brute Force)

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 void findSubsets(int ind, vector<int>& nums,
 vector<int>& ds, set<vector<int>>& st) {
 if (ind == nums.size()) {
 st.insert(ds);
 return;
 }
 for (int i = ind; i < nums.size(); i++) {
 ds.push_back(nums[i]);
 findSubsets(i + 1, nums, ds, st);
 ds.pop_back();
 }
 }
};
```

```

 }

 // Pick
 ds.push_back(nums[ind]);
 findSubsets(ind + 1, nums, ds, st);
 ds.pop_back();

 // Not pick
 findSubsets(ind + 1, nums, ds, st);
}

vector<vector<int>> subsetsWithDup(vector<int>& nums) {
 sort(nums.begin(), nums.end());
 set<vector<int>> st;
 vector<int> ds;

 findSubsets(0, nums, ds, st);

 return vector<vector<int>>(st.begin(), st.end());
}
};

int main() {
 Solution sol;
 vector<int> nums = {1, 2, 2};
 vector<vector<int>> ans = sol.subsetsWithDup(nums);

 for (auto &subset : ans) {
 cout << "[";
 for (int x : subset) cout << x << " ";
 cout << "] ";
 }
 return 0;
}

```

---

## Complexity (Brute Force)

- **Time Complexity:**  $O(N^2 * 2^N)$
- **Space Complexity:**  $O(N * 2^N)$

Reason:

All subsets are generated and stored in a set, which adds extra overhead.

---

## Optimal Approach (Backtracking with Skipping Duplicates)

### Idea

Instead of generating duplicates and removing them later, **avoid creating duplicates in the first place.**

Key observation:

- After sorting, **duplicate elements will be adjacent**
  - While looping, if the current element is the same as the previous one and we are at the same recursion level, **skip it**
- 

## Algorithm

1. Sort the array
2. Use backtracking:
  - Add current subset to result
  - Loop from start index to end
  - If  $i > \text{start}$  and  $\text{nums}[i] == \text{nums}[i-1]$ , skip
  - Pick element → recurse → backtrack

3. Return the result
- 

## Code (Optimal)

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 void backtrack(int start, vector<int>& nums,
 vector<int>& current, vector<vector<int>>& result)
 {

 result.push_back(current);

 for (int i = start; i < nums.size(); i++) {
 if (i > start && nums[i] == nums[i - 1]) continue;

 current.push_back(nums[i]);
 backtrack(i + 1, nums, current, result);
 current.pop_back();
 }
 }

 vector<vector<int>> subsetsWithDup(vector<int>& nums) {
 sort(nums.begin(), nums.end());
 vector<vector<int>> result;
 vector<int> current;

 backtrack(0, nums, current, result);
 return result;
 }
};

int main() {
 Solution sol;
 vector<int> nums = {1, 2, 2};
```

```

vector<vector<int>> ans = sol.subsetsWithDup(nums);

for (auto &subset : ans) {
 cout << "[";
 for (int i = 0; i < subset.size(); i++) {
 cout << subset[i];
 if (i + 1 < subset.size()) cout << ",";
 }
 cout << "] ";
}
return 0;
}

```

---

## Complexity (Optimal)

- **Time Complexity:**  $O(2^N)$
- **Space Complexity:**
  - $O(N)$  recursion stack
  - Output storage:  $O(2^N)$

# 16. Combination Sum III

## Explanation

You are given two integers  $k$  and  $n$ .

You need to find **all unique combinations of exactly  $k$  numbers** such that:

- Numbers are chosen only from **1 to 9**

- Each number is used **at most once**
- The sum of chosen numbers is exactly **n**
- No duplicate combinations are allowed

Order of combinations does not matter.

---

### Example 1

Input:

$k = 3, n = 7$

Output:

`[[1, 2, 4]]`

Explanation:

Only  $1 + 2 + 4 = 7$  using exactly 3 numbers.

---

### Example 2

Input:

$k = 3, n = 9$

Output:

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

---

## Approach

### Idea

This is a classic backtracking problem where we try to build combinations step by step.

At every step:

- Pick a number from 1 to 9
- Use each number only once
- Keep track of remaining sum
- Stop early if:
  - sum becomes negative
  - combination size exceeds k

---

## Algorithm

1. Start from number 1
2. Maintain:
  - remaining sum
  - current combination
3. If remaining sum is 0 and size of combination is k, store it
4. If sum < 0 or size > k, stop recursion
5. Try next numbers from last + 1 to 9
6. Backtrack after each recursive call

---

## Code

```
#include <bits/stdc++.h>
```

```

using namespace std;

class Solution {
private:
 void solve(int sum, int start, int k,
 vector<int>& curr,
 vector<vector<int>>& ans) {

 if (sum == 0 && curr.size() == k) {
 ans.push_back(curr);
 return;
 }

 if (sum < 0 || curr.size() > k) return;

 for (int i = start; i <= 9; i++) {
 curr.push_back(i);
 solve(sum - i, i + 1, k, curr, ans);
 curr.pop_back();
 }
 }
}

public:
 vector<vector<int>> combinationSum3(int k, int n) {
 vector<vector<int>> ans;
 vector<int> curr;
 solve(n, 1, k, curr, ans);
 return ans;
 }
};

int main() {
 Solution sol;
 int k = 3, n = 9;
 vector<vector<int>> res = sol.combinationSum3(k, n);

 for (auto &v : res) {
 for (int x : v) cout << x << " ";

```

```
 cout << endl;
}
return 0;
}
```

---

## Complexity Analysis

- **Time Complexity:**  $O(2^9 * k)$   
We explore all subsets of numbers from 1 to 9.
- **Space Complexity:**  $O(k)$   
Used by recursion stack and current combination.

# 17. Letter Combinations of a Phone Number

### Explanation

You are given a string `digits` containing characters from '2' to '9'.  
Each digit maps to certain letters on a phone keypad (like old mobile phones).  
Your task is to generate **all possible strings** by replacing each digit with one of its mapped letters, keeping the order same.

If the input string has length  $n$ , then each position contributes multiple choices, and we generate combinations by choosing one letter per digit.

---

### Example 1

Input:

```
digits = "34"
```

Explanation:

- 3 → "def"
- 4 → "ghi"

All combinations formed by picking one letter from "def" and one from "ghi":

```
["dg", "dh", "di", "eg", "eh", "ei", "fg", "fh", "fi"]
```

---

## Example 2

Input:

```
digits = "3"
```

Output:

```
["d", "e", "f"]
```

---

## Approach

### Idea

This is a backtracking problem.

At each index, we:

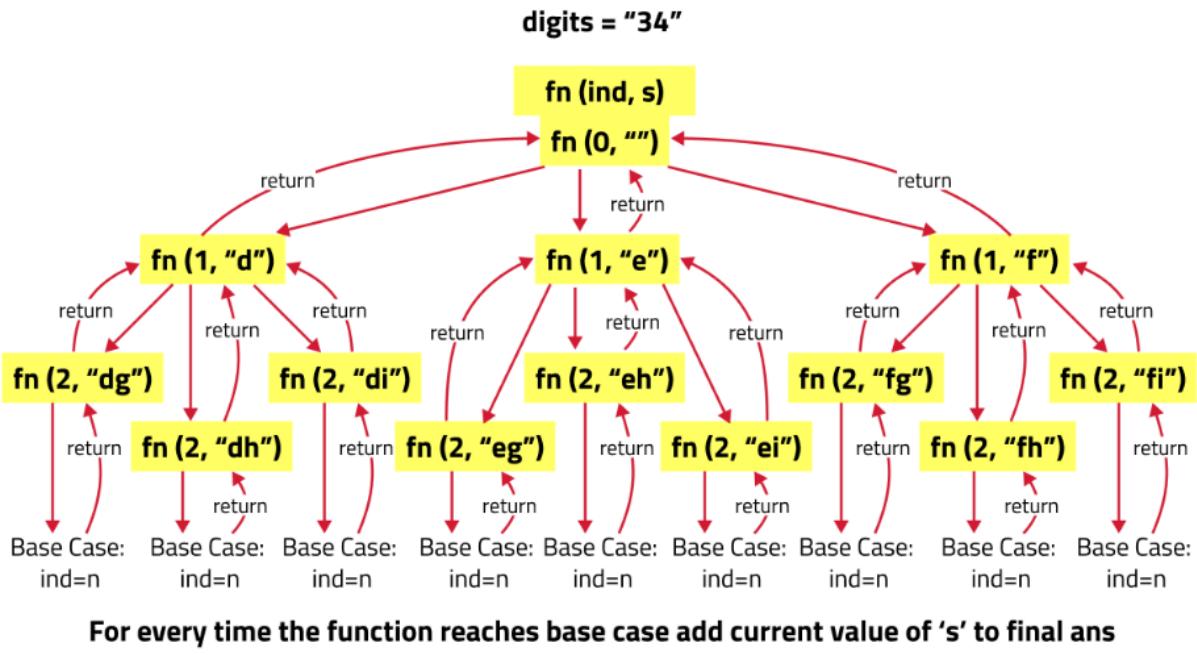
- Look at the digit
- Try all characters mapped to that digit
- Move to the next index

When we reach the end of the digit string, the current built string is a valid combination.

---

## Algorithm

1. Create a mapping from digits (2–9) to letters.
2. Use a recursive function with:
  - current index
  - current string being formed
3. If index equals length of digits, store the string.
4. Otherwise:
  - For the current digit, loop through all mapped letters
  - Append one letter and recurse to next index



## Code

```

#include <bits/stdc++.h>
using namespace std;

class Solution {
private:
 void solve(int index, string &digits, string &curr,
 vector<string> &ans, string mp[]) {

 if (index == digits.size()) {
 ans.push_back(curr);
 return;
 }

 int digit = digits[index] - '0';

```

```

 for (char ch : mp[digit]) {
 curr.push_back(ch);
 solve(index + 1, digits, curr, ans, mp);
 curr.pop_back();
 }
 }

public:
 vector<string> letterCombinations(string digits) {
 if (digits.size() == 0) return {};

 string mp[] = {
 "", "", "abc", "def", "ghi",
 "jkl", "mno", "pqrs", "tuv", "wxyz"
 };

 vector<string> ans;
 string curr = "";
 solve(0, digits, curr, ans, mp);
 return ans;
 }
};

int main() {
 Solution sol;
 string digits = "34";
 vector<string> res = sol.letterCombinations(digits);

 for (auto &s : res) cout << s << " ";
 return 0;
}

```

---

## Complexity Analysis

- **Time Complexity:**  $O(4^n)$   
Each digit can contribute up to 4 letters.

- **Space Complexity:**  $O(n)$   
Recursion stack depth and temporary string length.

# 18. Palindrome Partitioning

## Explanation

You are given a string  $s$ . Your task is to split (partition) the string into substrings such that **every substring is a palindrome**.

A palindrome reads the same from left to right and right to left.

You must return **all possible such partitions**.

---

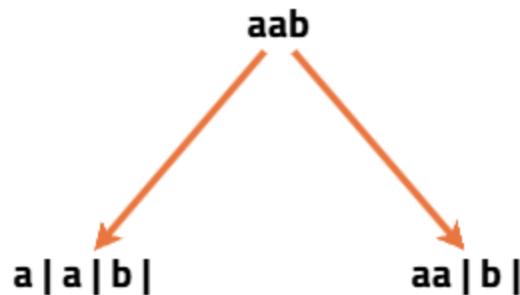
## Example 1

Input:

```
s = "aab"
```

Output:

```
["a", "a", "b"]
["aa", "b"]
```



Explanation:

- "a" | "a" | "b" → all are palindromes
- "aa" | "b" → both are palindromes

---

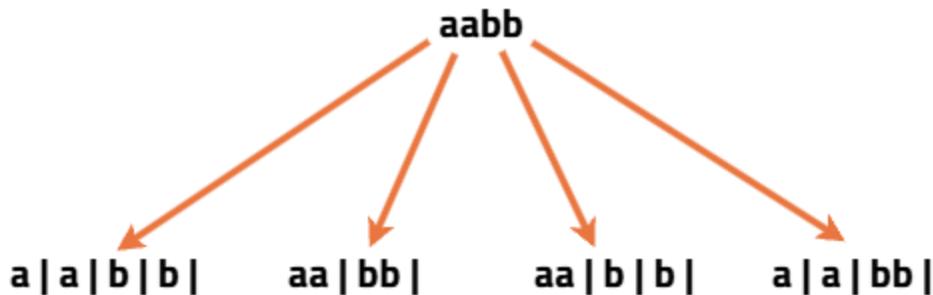
## Example 2

Input:

s = "aabb"

Output:

```
["a" , "a" , "b" , "b"]
["a" , "a" , "bb"]
["aa" , "b" , "b"]
["aa" , "bb"]
```



## Approach

We use **recursion + backtracking**.

At every index, we try to cut the string into substrings.

For each substring:

- If it is a palindrome → choose it
- Recurse for the remaining string
- Backtrack to try other possibilities

The recursion ends when the full string is consumed.

---

## Code

```
#include <bits/stdc++.h>
using namespace std;
```

```

class Solution {
public:
 bool isPalindrome(string &s, int l, int r) {
 while (l < r) {
 if (s[l] != s[r]) return false;
 l++;
 r--;
 }
 return true;
 }

 void solve(int index, string &s, vector<string> &path,
 vector<vector<string>> &ans) {
 if (index == s.size()) {
 ans.push_back(path);
 return;
 }

 for (int i = index; i < s.size(); i++) {
 if (isPalindrome(s, index, i)) {
 path.push_back(s.substr(index, i - index + 1));
 solve(i + 1, s, path, ans);
 path.pop_back();
 }
 }
 }

 vector<vector<string>> partition(string s) {
 vector<vector<string>> ans;
 vector<string> path;
 solve(0, s, path, ans);
 return ans;
 }
};

int main() {
 Solution sol;
 string s = "aab";

```

```

vector<vector<string>> res = sol.partition(s);

for (auto &v : res) {
 for (auto &x : v) cout << x << " ";
 cout << endl;
}
return 0;
}

```

---

## Complexity Analysis

- **Time Complexity:**  $O(2^N * N)$   
All possible partitions are explored, and palindrome checking costs  $O(N)$ .
- **Space Complexity:**  $O(2^N * N) + O(N)$   
For storing all partitions and recursion stack.

# 19. Word Search (LeetCode)

### Explanation

You are given a 2D grid of characters (board) and a string word.

You need to check whether the word can be formed by **moving through adjacent cells** (up, down, left, right).

### Important rules:

- Each character must match the corresponding character in word
- You can move only **horizontally or vertically**
- **The same cell cannot be used more than once** in forming the word

## Example 1

Input:

```
board = [
 ['A', 'B', 'C', 'E'],
 ['S', 'F', 'C', 'S'],
 ['A', 'D', 'E', 'E']
]
word = "ABCED"
```

Output:

```
true
```

Explanation:

The path A → B → C → C → E → D exists in the grid.

---

## Example 2

Input:

```
word = "ABCB"
```

Output:

```
false
```

Explanation:

Although the letters exist, you would need to reuse a cell, which is **not allowed**.

---

## Approach (Backtracking / DFS)

This is a classic **backtracking** problem.

- Start DFS from every cell in the grid
  - If the current cell matches the current character of the word:
    - Mark the cell as visited (temporarily)
    - Try moving in all 4 directions
  - If at any point the full word is matched → return true
  - If a path fails, **backtrack** by restoring the cell
- 

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 bool exist(vector<vector<char>>& board, string word) {
 int rows = board.size();
 int cols = board[0].size();

 for (int i = 0; i < rows; i++) {
 for (int j = 0; j < cols; j++) {
 if (dfs(board, word, i, j, 0))
 return true;
 }
 }
 return false;
 }

private:
 bool dfs(vector<vector<char>>& board, string &word,
 int i, int j, int idx) {

 if (idx == word.size())
 return false;

 if (board[i][j] != word[idx])
 return false;

 char temp = board[i][j];
 board[i][j] = '#';

 if (idx + 1 == word.size())
 return true;

 if (dfs(board, word, i - 1, j, idx + 1) ||
 dfs(board, word, i + 1, j, idx + 1) ||
 dfs(board, word, i, j - 1, idx + 1) ||
 dfs(board, word, i, j + 1, idx + 1))
 return true;

 board[i][j] = temp;
 }
}
```

```

 return true;

 if (i < 0 || j < 0 || i >= board.size() ||
 j >= board[0].size() || board[i][j] != word[idx])
 return false;

 char temp = board[i][j];
 board[i][j] = '#'; // mark visited

 bool found =
 dfs(board, word, i + 1, j, idx + 1) ||
 dfs(board, word, i - 1, j, idx + 1) ||
 dfs(board, word, i, j + 1, idx + 1) ||
 dfs(board, word, i, j - 1, idx + 1);

 board[i][j] = temp; // backtrack
 return found;
}

};

int main() {
 Solution sol;
 vector<vector<char>> board = {
 {'A', 'B', 'C', 'E'},
 {'S', 'F', 'C', 'S'},
 {'A', 'D', 'E', 'E'}
 };

 cout << boolalpha << sol.exist(board, "ABCCED") << endl; // true
 cout << boolalpha << sol.exist(board, "SEE") << endl; // true
 cout << boolalpha << sol.exist(board, "ABCB") << endl; // false
 return 0;
}

```

---

## Complexity Analysis

- **Time Complexity:**  $O(m \times n \times 4^L)$   
From each cell, we try 4 directions for each character of the word.
- **Space Complexity:**  $O(L)$   
Due to recursion stack depth equal to the word length.

## 20. N-Queen Problem | Return all Distinct Solutions to the N-Queens Puzzle

---

### Problem Explanation

You are given an integer **n**, representing an  **$n \times n$  chessboard**.  
Your task is to place **n queens** on the board such that:

1. No two queens attack each other.
2. A queen can attack another queen if they are:
  - In the same **row**
  - In the same **column**
  - On the same **diagonal**

You must return **all distinct valid board configurations**.

Each configuration is represented as:

- 'Q' → Queen placed
- '.' → Empty cell

---

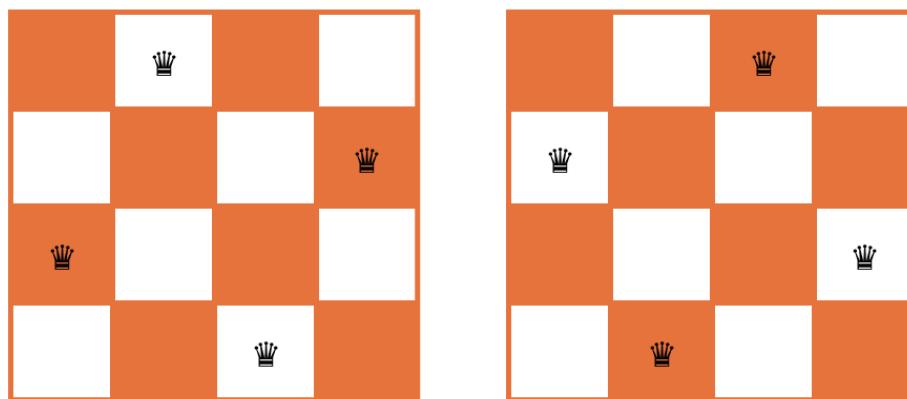
## Example 1

### Input

n = 4

### Output

```
[
 [".Q..", "...Q", "Q...", "..Q."],
 [...Q., "Q...", "...Q", ".Q.."]
]
```



Two arrangements possible for 4 Queens

### Explanation

There are exactly **2 valid ways** to place 4 queens such that none attack each other.

---

## Example 2

### Input

n = 1

### Output

[["Q"]]

### Explanation

Only one cell exists, so placing a single queen is always valid.

## Approach 1: Brute Force / Basic Backtracking

---

### Idea

We place queens **column by column**.

For each column:

- Try placing a queen in **every row**
- Before placing, check if it is **safe**
- If safe → place queen and move to next column
- If not → try next row
- If no row works → backtrack

## Safety Check (Important)

To check if a queen can be placed at (row, col):

1. Check **same row on the left**
2. Check **upper-left diagonal**
3. Check **lower-left diagonal**

Only left side is checked because we place queens column by column from left to right.

---

## Code (Brute Force)

```
#include <bits/stdc++.h>

using namespace std;

class Solution {

public:

 bool isSafe(int row, int col, vector<vector<char>>& board, int n)
 {

 // Check same row (left side)

 for (int j = 0; j < col; j++) {

 if (board[row][j] == 'Q') return false;

 }

 // Check upper-left diagonal

 for (int i = row, j = col; i >= 0 && j >= 0; i--, j--) {

 if (board[i][j] == 'Q') return false;

 }

 // Check lower-left diagonal

 for (int i = row, j = col; i < n && j >= 0; i++, j--) {

 if (board[i][j] == 'Q') return false;

 }

 }

}
```

```

 }

 return true;
}

void solve(int col, vector<vector<char>>& board,
 vector<vector<string>>& ans, int n) {

 if (col == n) {
 vector<string> temp;
 for (int i = 0; i < n; i++) {
 temp.push_back(string(board[i].begin(),
board[i].end()));
 }
 ans.push_back(temp);
 return;
 }

 for (int row = 0; row < n; row++) {
 if (isSafe(row, col, board, n)) {
 board[row][col] = 'Q';
 solve(col + 1, board, ans, n);
 board[row][col] = '.';
 }
 }
}

```

```

 }

}

}

vector<vector<string>> solveNQueens(int n) {

 vector<vector<string>> ans;

 vector<vector<char>> board(n, vector<char>(n, '.'));

 solve(0, board, ans, n);

 return ans;

};

}

```

---

## Complexity (Brute Force)

- **Time Complexity:**  $O(N! \times N)$   
(Trying all permutations + safety checks)
  - **Space Complexity:**  $O(N^2)$   
(Board + recursion stack)
- 

## Key Idea (Backtracking)

This is a classic **backtracking** problem.

We place queens **column by column**:

- In each column, try placing a queen in every row
  - Before placing, check if the position is **safe**
  - If safe → place queen and move to next column
  - If not → try next row
  - If no row works → **backtrack**
- 

## Why Backtracking Works Here

- Each queen placement depends on previous placements
  - Wrong choices must be undone (backtracking)
  - We explore all valid possibilities efficiently
- 

## Optimal Approach (Using Hashing for Fast Safety Check)

Instead of checking the entire board every time, we use **3 helper arrays**:

1. `leftRow[row]` → checks if a row already has a queen
2. `lowerDiagonal[row + col]` → checks \ diagonal
3. `upperDiagonal[n - 1 + col - row]` → checks / diagonal

This allows **O(1)** safety checks.

---

For checking Left now

|   | 0 | 1 | 2 | 3  | 4  | 5  | 6  | 7  |
|---|---|---|---|----|----|----|----|----|
| 0 | 0 | 1 | 2 | 3  | 4  | 5  | 6  | 7  |
| 1 | 1 | 2 | 3 | 4  | 5  | 6  | 7  | 8  |
| 2 | 2 | 3 | 4 | 5  | 6  | 7  | 8  | 9  |
| 3 | 3 | 4 | 5 | 6  | 7  | 8  | 9  | 10 |
| 4 | 4 | 5 | 6 | 7  | 8  | 9  | 10 | 11 |
| 5 | 5 | 6 | 7 | 8  | 9  | 10 | 11 | 12 |
| 6 | 6 | 7 | 8 | 9  | 10 | 11 | 12 | 13 |
| 7 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

In the grid, we will fill the sum of indices of row and columns

We can check that diagonal elements are same in grid

If we are taking  $n \times n$  grid we can take maximum value as  $2 \times n - 1$   
for  $8 \times 8$  grid, maximum value =  $2 \times 8 - 1 = 15$   
means hash size is 15

|   |   |   |                                     |   |   |   |   |   |   |    |    |    |    |
|---|---|---|-------------------------------------|---|---|---|---|---|---|----|----|----|----|
|   |   |   | <input checked="" type="checkbox"/> |   |   |   |   |   |   |    |    |    |    |
| 0 | 1 | 2 | 3                                   | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |

For checking upper diagonal and lower diagonal

|   | 0 | 1 | 2 | 3  | 4  | 5  | 6  | 7  |
|---|---|---|---|----|----|----|----|----|
| 0 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
| 1 | 6 | 7 | 8 | 9  | 10 | 11 | 12 | 13 |
| 2 | 5 | 6 | 7 | 8  | 9  | 10 | 11 | 12 |
| 3 | 4 | 5 | 6 | 7  | 8  | 9  | 10 | 11 |
| 4 | 3 | 4 | 5 | 6  | 7  | 8  | 9  | 10 |
| 5 | 2 | 3 | 4 | 5  | 6  | 7  | 8  | 9  |
| 6 | 1 | 2 | 3 | 4  | 5  | 6  | 7  | 8  |
| 7 | 0 | 1 | 2 | 3  | 4  | 5  | 6  | 7  |

In the grid, we will fill the  $(n-1) + (\text{row} - \text{col})$

We can check that diagonal elements are same in grid

If we are taking  $n \times n$  grid we can take maximum value as  $2 \times n - 1$   
for  $8 \times 8$  grid, maximum value =  $2 \times 8 - 1 = 15$   
means hash size is 15

|   |   |                                     |   |   |   |   |   |   |   |    |    |    |    |
|---|---|-------------------------------------|---|---|---|---|---|---|---|----|----|----|----|
|   |   | <input checked="" type="checkbox"/> |   |   |   |   |   |   |   |    |    |    |    |
| 0 | 1 | 2                                   | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |

F

For checking Left now

|   | 0 | 1 | 2 | 3  | 4  | 5  | 6  | 7  |
|---|---|---|---|----|----|----|----|----|
| 0 | 0 | 1 | 2 | 3  | 4  | 5  | 6  | 7  |
| 1 | 1 | 2 | 3 | 4  | 5  | 6  | 7  | 8  |
| 2 | 2 | 3 | 4 | 5  | 6  | 7  | 8  | 9  |
| 3 | 3 | 4 | 5 | 6  | 7  | 8  | 9  | 10 |
| 4 | 4 | 5 | 6 | 7  | 8  | 9  | 10 | 11 |
| 5 | 5 | 6 | 7 | 8  | 9  | 10 | 11 | 12 |
| 6 | 6 | 7 | 8 | 9  | 10 | 11 | 12 | 13 |
| 7 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

In the grid, we will fill the sum of indices of row and columns

We can check that diagonal elements are same in grid

If we are taking  $n \times n$  grid we can take maximum value as  $2 \times n - 1$   
for  $8 \times 8$  grid, maximum value =  $2 \times 8 - 1 = 15$   
means hash size is 15

|   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |  |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|--|
|   |   |   | ✓ |   |   |   |   |   |   |    |    |    |    |    |  |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |  |

F

For checking upper diagonal and lower diagonal

|   | 0 | 1 | 2 | 3  | 4  | 5  | 6  | 7  |
|---|---|---|---|----|----|----|----|----|
| 0 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
| 1 | 6 | 7 | 8 | 9  | 10 | 11 | 12 | 13 |
| 2 | 5 | 6 | 7 | 8  | 9  | 10 | 11 | 12 |
| 3 | 4 | 5 | 6 | 7  | 8  | 9  | 10 | 11 |
| 4 | 3 | 4 | 5 | 6  | 7  | 8  | 9  | 10 |
| 5 | 2 | 3 | 4 | 5  | 6  | 7  | 8  | 9  |
| 6 | 1 | 2 | 3 | 4  | 5  | 6  | 7  | 8  |
| 7 | 0 | 1 | 2 | 3  | 4  | 5  | 6  | 7  |

In the grid, we will fill the  $(n-1) + (\text{row} - \text{col})$

We can check that diagonal elements are same in grid

If we are taking  $n \times n$  grid we can take maximum value as  $2 \times n - 1$   
for  $8 \times 8$  grid, maximum value =  $2 \times 8 - 1 = 15$   
means hash size is 15

|   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |  |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|--|
|   |   |   | ✓ |   |   |   |   |   |   |    |    |    |    |    |  |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |  |

row

col

row+col

map < (row+col) -> True/False

map < 7 -> True

7 -> TRUE

$O(n^2)$

## Code (Optimal Solution)

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 void solve(int col, vector<string>& board, int n,
 vector<int>& leftRow,
 vector<int>& upperDiagonal,
 vector<int>& lowerDiagonal,
 vector<vector<string>>& ans) {

 // Base case: all queens placed
 if (col == n) {
 ans.push_back(board);
 return;
 }

 // Try placing queen in every row of this column
 for (int row = 0; row < n; row++) {
 if (leftRow[row] == 0 &&
 lowerDiagonal[row + col] == 0 &&
 upperDiagonal[n - 1 + col - row] == 0) {

 // Place queen
 board[row][col] = 'Q';
 leftRow[row] = 1;
 lowerDiagonal[row + col] = 1;
 upperDiagonal[n - 1 + col - row] = 1;

 // Recurse to next column
 solve(col + 1, board, n, leftRow, upperDiagonal,
 lowerDiagonal, ans);

 // Backtrack
 board[row][col] = '.';
 leftRow[row] = 0;
 lowerDiagonal[row + col] = 0;
 }
 }
 }
}
```

```

 upperDiagonal[n - 1 + col - row] = 0;
 }
}
}

vector<vector<string>> solveNQueens(int n) {
 vector<vector<string>> ans;
 vector<string> board(n, string(n, '.'));

 vector<int> leftRow(n, 0);
 vector<int> lowerDiagonal(2 * n - 1, 0);
 vector<int> upperDiagonal(2 * n - 1, 0);

 solve(0, board, n, leftRow, upperDiagonal, lowerDiagonal,
ans);
 return ans;
}
};

int main() {
 Solution obj;
 int n = 4;
 vector<vector<string>> res = obj.solveNQueens(n);

 for (auto& board : res) {
 for (auto& row : board) {
 cout << row << "\n";
 }
 cout << "\n";
 }
 return 0;
}

```

---

## Complexity Analysis

### Time Complexity

$O(N!)$

- We try all possible permutations of queen placements

## Space Complexity

$O(N)$

- Recursion stack + helper arrays

# 21. Rat in a Maze

---

## Question Explanation

You are given a square grid of size  $n \times n$  consisting of 0s and 1s.

- The rat starts at **(0, 0)**
- The destination is **(n-1, n-1)**
- The rat can move in **four directions**:
  - D → Down
  - L → Left
  - R → Right
  - U → Up

## Rules

1. The rat can move only to cells with value 1
  2. A cell with value 0 is blocked
  3. A cell cannot be visited more than once in the same path
  4. If the starting cell  $(0, 0)$  is 0, no path exists
  5. You must return **all possible paths in lexicographical order**
- 

## Example 1

### Input

```
n = 4
grid =
[
 [1, 0, 0, 0],
 [1, 1, 0, 1],
 [1, 1, 0, 0],
 [0, 1, 1, 1]
]
```

### Output

```
["DDRDRR", "DRDDRR"]
```

### Explanation

Two valid paths exist from  $(0, 0)$  to  $(3, 3)$  following the movement rules.

---

## Example 2

## Input

```
n = 2
grid =
[
 [1, 0],
 [1, 0]
]
```

## Output

```
[]
```

## Explanation

The destination cell (1, 1) is blocked, so no valid path exists.

---

## Approach: Backtracking (DFS)

---

## Intuition

Since we need to find **all possible paths**, the rat must explore the maze **step by step**, trying every valid direction.

If the rat:

- Hits a blocked cell
- Goes out of bounds
- Revisits a cell

→ that path is abandoned (backtracking).

This guarantees:

- No infinite loops
  - Every valid path is explored exactly once
- 

## Key Points

- Use a visited matrix to avoid revisiting cells

Always explore directions in **lexicographical order**:

D → L → R → U

- - Mark a cell as visited when moving forward
  - Unmark it while backtracking
- 

## Algorithm Steps

1. If  $\text{maze}[0][0] == 0$ , return empty answer
  2. Start DFS from  $(0, 0)$  with an empty path string
  3. At each cell:
    - Try all 4 directions in order: D, L, R, U
  4. If destination  $(n-1, n-1)$  is reached:
    - Store the current path
  5. Backtrack by unmarking the visited cell
-

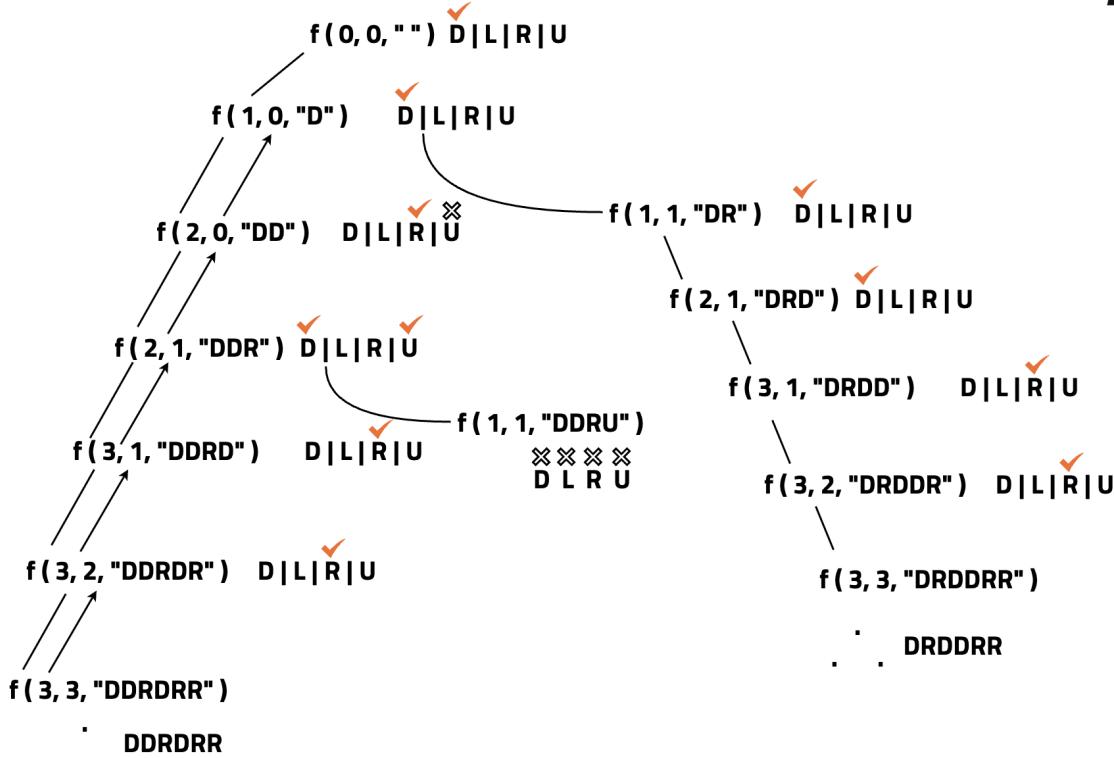
F

|   | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | ✓ |   |   |   |
| 1 | ✓ |   |   |   |
| 2 | ✓ | ✓ |   |   |
| 3 |   | ✓ | ✓ |   |

**Visited**

F

|   | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 0 | 1 |
| 2 | 1 | 1 | 0 | 0 |
| 3 | 0 | 1 | 1 | 1 |



## Code (C++)

```

#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 // Check if the move is valid
 bool isSafe(int x, int y, int n,
 vector<vector<int>>& maze,
 vector<vector<int>>& visited) {
 return (x >= 0 && x < n &&
 y >= 0 && y < n &&
 maze[x][y] == 1 &&
 visited[x][y] == 0);
 }

 // Backtracking function

```

```

void solve(int x, int y, int n,
 vector<vector<int>>& maze,
 vector<vector<int>>& visited,
 string path,
 vector<string>& res) {

 // Destination reached
 if (x == n - 1 && y == n - 1) {
 res.push_back(path);
 return;
 }

 visited[x][y] = 1;

 // Down
 if (isSafe(x + 1, y, n, maze, visited))
 solve(x + 1, y, n, maze, visited, path + "D", res);

 // Left
 if (isSafe(x, y - 1, n, maze, visited))
 solve(x, y - 1, n, maze, visited, path + "L", res);

 // Right
 if (isSafe(x, y + 1, n, maze, visited))
 solve(x, y + 1, n, maze, visited, path + "R", res);

 // Up
 if (isSafe(x - 1, y, n, maze, visited))
 solve(x - 1, y, n, maze, visited, path + "U", res);

 // Backtrack
 visited[x][y] = 0;
}

vector<string> findPath(vector<vector<int>>& maze, int n) {
 vector<string> res;
 vector<vector<int>> visited(n, vector<int>(n, 0));
}

```

```

 if (maze[0][0] == 1) {
 solve(0, 0, n, maze, visited, "", res);
 }

 return res;
 }
};

```

---

## Complexity Analysis

- **Time Complexity:**  $O(4^{n \times n})$   
In the worst case, the rat explores all 4 directions from every cell.
- **Space Complexity:**  $O(n \times n)$   
For the visited matrix and recursion stack.

## 22. Word Break

---

### Question Explanation

You are given:

- A string **s**
- A dictionary **wordDict** containing valid words

Your task is to check **whether the string s can be broken into a sequence of dictionary words.**

## Important points

- You can reuse dictionary words multiple times
  - The order must follow the original string
  - Every part of the string must belong to the dictionary
  - If even one segment is invalid → return false
- 

## Example 1

### Input

```
s = "takeuforward"
wordDict = ["take", "forward", "you", "u"]
```

### Output

```
true
```

### Explanation

The string can be segmented as:

```
"take" + "u" + "forward"
```

All parts exist in the dictionary.

---

## Example 2

### Input

```
s = "applepineapple"
wordDict = ["apple"]
```

## Output

false

## Explanation

Possible split:

"apple" + "pine" + "apple"

But "pine" is **not** in the dictionary, so segmentation fails.

---

## Approach: Dynamic Programming

---

## Intuition

At every index of the string, we ask:

**Can the substring up to this index be segmented using dictionary words?**

If yes, we mark it as valid and continue forward.

We use DP because:

- The same substrings are checked repeatedly
  - DP avoids recomputation
  - Efficient for large strings
-

# DP Definition

Let:

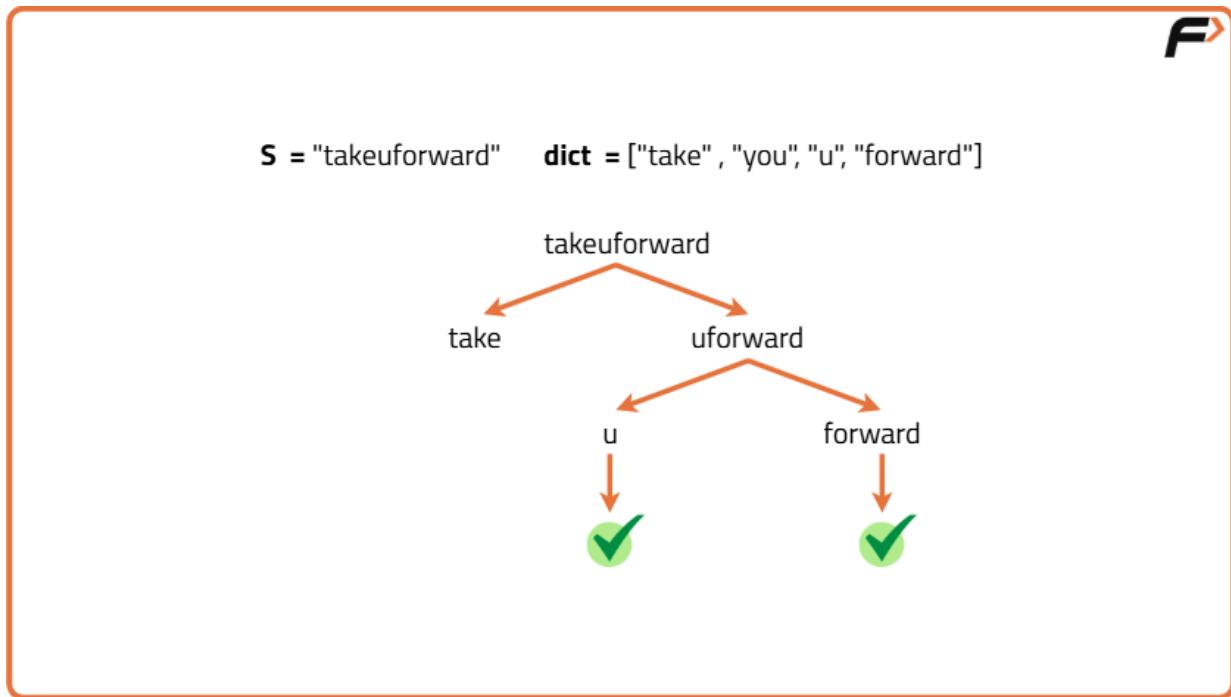
$dp[i] = \text{true}$

if substring  $s[0 \dots i-1]$  **can be segmented**

Final answer will be:

$dp[n]$

where  $n = \text{length of } s$



## Algorithm (Step-by-Step)

1. Convert wordDict into a **set** for fast lookup
2. Create a boolean array dp of size  $n + 1$

Initialize:

dp[0] = true

3. (empty string is always valid)
4. Find the **maximum word length** in dictionary
  - o This limits unnecessary substring checks
5. For every position i from 1 to n:
  - o Try splitting at position j

If:

dp[j] == true  
AND s[j ... i-1] exists in dictionary  
then mark:

dp[i] = true

- o
6. Return dp[n]
- 

## Code (C++)

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 bool wordBreak(string s, vector<string>& wordDict) {
 int n = s.length();

 // Store dictionary words in a set for O(1) lookup
 unordered_set<string> dict(wordDict.begin(), wordDict.end());
```

```

// dp[i] = true if s[0..i-1] can be segmented
vector<bool> dp(n + 1, false);
dp[0] = true; // empty string

// Find maximum length of any word in dictionary
int maxLen = 0;
for (string &w : wordDict) {
 maxLen = max(maxLen, (int)w.size());
}

// Fill dp array
for (int i = 1; i <= n; i++) {
 for (int j = max(0, i - maxLen); j < i; j++) {
 if (dp[j] && dict.count(s.substr(j, i - j))) {
 dp[i] = true;
 break;
 }
 }
}

return dp[n];
}
};


```

---

## Complexity Analysis

### Time Complexity

$O(n * m)$

- $n$  = length of string
- $m$  = maximum word length
- Limited substring checks improve performance

## Space Complexity

$O(n)$

- DP array of size  $n + 1$
- Dictionary set

# 23. M – Coloring Problem

---

## Question Explanation

You are given:

- An **undirected graph** with  $N$  vertices
- A number  $M$  representing available colors

Your task is to check **whether it is possible to color all vertices** such that:

- Each vertex has **one color**
- **No two adjacent vertices** have the **same color**
- You can use **at most  $M$  colors**

Return:

- $1 \rightarrow$  if coloring is possible
- $0 \rightarrow$  if coloring is not possible

---

## Example 1

### Input

```
N = 4
M = 3
Edges = {(0,1), (1,2), (2,3), (3,0), (0,2)}
```

### Output

1

### Explanation

The graph can be colored using 3 colors without any adjacent vertices sharing the same color.

---

## Example 2

### Input

```
N = 3
M = 2
Edges = {(0,1), (1,2), (0,2)}
```

### Output

0

### Explanation

The graph forms a triangle.  
At least **3 colors** are needed, so coloring with 2 colors is not possible.

---

# Approach (Backtracking)

---

## Intuition

We try to assign colors to vertices **one by one**.

For each vertex:

- Try every color from 1 to M
- Check if assigning that color is **safe**
- If safe → move to the next vertex
- If not → try another color
- If all colors fail → backtrack

This explores all possible color assignments.

---

## Algorithm (Step-by-Step)

1. Start coloring from vertex 0
2. For the current vertex:
  - Try each color from 1 to M
3. Use `isSafe()` to check:
  - No adjacent vertex has the same color
4. If safe:
  - Assign the color
  - Recursively color the next vertex

5. If recursion fails:

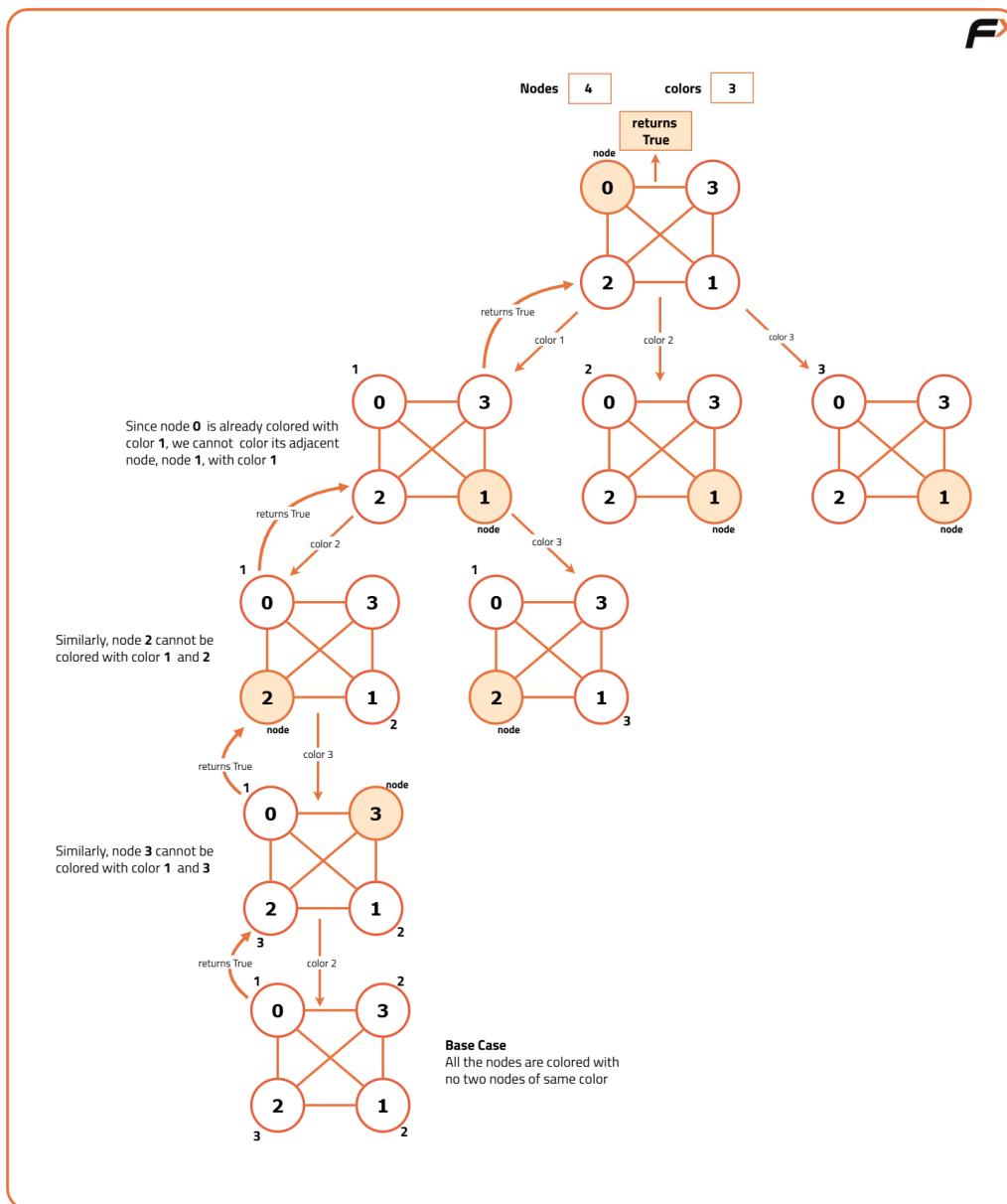
- Remove the color (backtrack)

6. If all vertices are colored successfully:

- Return true

7. If no color works:

- Return false



---

## Code (C++)

```
#include <bits/stdc++.h>
using namespace std;

// Check if it is safe to color the node with given color
bool isSafe(int node, int color[], bool graph[101][101], int n, int
col) {
 for (int k = 0; k < n; k++) {
 if (graph[node][k] && color[k] == col) {
 return false;
 }
 }
 return true;
}

// Backtracking function
bool solve(int node, int color[], int m, int n, bool graph[101][101])
{
 // All nodes are colored
 if (node == n) return true;

 // Try all colors
 for (int col = 1; col <= m; col++) {
 if (isSafe(node, color, graph, n, col)) {
 color[node] = col;

 if (solve(node + 1, color, m, n, graph))
 return true;

 // Backtrack
 color[node] = 0;
 }
 }
 return false;
}
```

```

// Main function
bool graphColoring(bool graph[101][101], int m, int n) {
 int color[n];
 memset(color, 0, sizeof(color));

 return solve(0, color, m, n, graph);
}

int main() {
 int n = 4, m = 3;
 bool graph[101][101] = {0};

 graph[0][1] = graph[1][0] = 1;
 graph[1][2] = graph[2][1] = 1;
 graph[2][3] = graph[3][2] = 1;
 graph[3][0] = graph[0][3] = 1;
 graph[0][2] = graph[2][0] = 1;

 cout << graphColoring(graph, m, n);
 return 0;
}

```

---

## Complexity Analysis

### Time Complexity

$O(M^N)$

- For each vertex, we try M colors

### Space Complexity

$O(N)$

- Color array

- Recursion stack

## 24. Sudoku Solver

---

### Question Explanation

You are given a **9×9 Sudoku board** where some cells are empty and represented by ' . '.

Your task is to **fill all empty cells** such that the final board satisfies **Sudoku rules**:

1. Every row must contain digits 1 to 9 exactly once
2. Every column must contain digits 1 to 9 exactly once
3. Each 3×3 sub-box must contain digits 1 to 9 exactly once

You must modify the board **in-place** and produce **any one valid solution**.

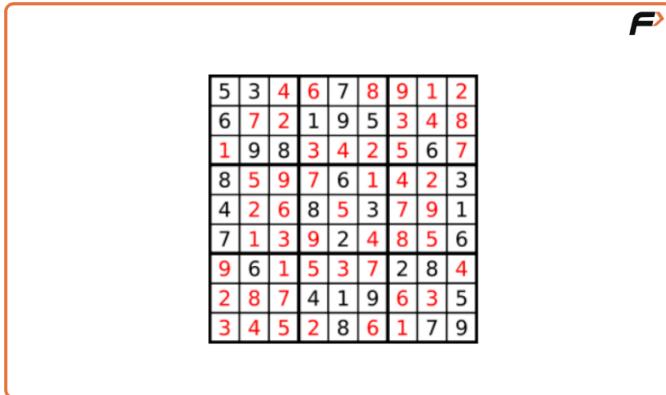
---

### Example (Conceptual)

Input board contains empty cells .

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

Output board is fully filled and valid according to Sudoku rules.



## Approach (Backtracking – as given)

---

### Intuition

Sudoku is a **trial-and-error** problem.

Whenever we find an empty cell:

- Try placing digits from 1 to 9
- Check if the placement is valid
- If valid → move forward recursively
- If no number works → backtrack

This guarantees that all possibilities are explored safely.

---

### Algorithm (Step-by-Step)

1. Traverse the board cell by cell
2. When an empty cell ' .' is found:
  - Try digits from '1' to '9'

3. For each digit, check validity using `isValid()`:

- Row check
- Column check
- $3 \times 3$  sub-box check

4. If valid:

- Place the digit
- Recursively solve the rest of the board

5. If recursion fails:

- Remove the digit (backtrack)

6. If all cells are filled successfully:

- Return true
- 

## Validity Check Logic

For placing digit  $c$  at  $(\text{row}, \text{col})$ :

- Check column  $\rightarrow \text{board}[\text{i}][\text{col}]$
- Check row  $\rightarrow \text{board}[\text{row}][\text{i}]$

Check  $3 \times 3$  box using:

```
rowIndex = 3 * (row / 3) + i / 3
colIndex = 3 * (col / 3) + i % 3
•
```

---



Let's fill board[0][3]...  
Possible values : 2 and 6

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

Filling with  
2

|   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|
| 9 | 5 | 7 | 2 | 1 | 3 | X | 8 | 4 |
| 4 | 8 | 3 | - | 5 | 7 | 1 | - | 6 |
| - | 1 | 2 | - | 4 | 9 | 5 | 3 | 7 |
| 1 | 7 | - | 3 | - | 4 | 9 | - | 2 |
| 5 | - | 4 | 9 | 7 | - | 3 | 6 | - |
| 3 | - | 9 | 5 | - | 8 | 7 | - | 1 |
| 8 | 4 | 5 | 7 | 9 | - | 6 | 1 | 3 |
| - | 9 | 1 | - | 3 | 6 | - | 7 | 5 |
| 7 | - | 6 | 1 | 8 | 5 | 4 | - | 9 |

No possible  
values for  
board[0][6]

Returning  
FALSE

Filling with  
6

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

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

## Code (C++)

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 // Check if placing character c at board[row][col] is valid
 bool isValid(vector<vector<char>> &board, int row, int col, char c) {
 for (int i = 0; i < 9; i++) {
 if (board[i][col] == c) return false;
 if (board[row][i] == c) return false;
 if (board[3 * (row / 3) + i / 3]
 [3 * (col / 3) + i % 3] == c) //integer division
 return false;
 }
 return true;
 }

 // Backtracking solver
 bool solveSudoku(vector<vector<char>> &board) {
 for (int i = 0; i < 9; i++) {
 for (int j = 0; j < 9; j++) {
 if (board[i][j] == '.') {
 for (char c = '1'; c <= '9'; c++) {
 if (isValid(board, i, j, c)) {
 board[i][j] = c;
 if (solveSudoku(board))
 return true;
 board[i][j] = '.';
 }
 }
 }
 return false;
 }
 }
 return true;
 }
}
```

```

};

int main() {
 vector<vector<char>> board {
 {'9', '5', '7', '.', '1', '3', '.', '8', '4'},
 {'4', '8', '3', '.', '5', '7', '1', '.', '6'},
 {'.', '1', '2', '.', '4', '9', '5', '3', '7'},
 {'1', '7', '.', '3', '.', '4', '9', '.', '2'},
 {'5', '.', '4', '9', '7', '.', '3', '6', '.'},
 {'3', '.', '9', '5', '.', '8', '7', '.', '1'},
 {'8', '4', '5', '7', '9', '.', '6', '1', '3'},
 {'.', '9', '1', '.', '3', '6', '.', '7', '5'},
 {'7', '.', '6', '1', '8', '5', '4', '.', '9'}
 };
}

Solution sol;
sol.solveSudoku(board);

for (int i = 0; i < 9; i++) {
 for (int j = 0; j < 9; j++)
 cout << board[i][j] << " ";
 cout << "\n";
}
return 0;
}

```

---

## Complexity Analysis

### Time Complexity

$$O(9^{(n^2)})$$

- For each empty cell, we try up to 9 numbers

### Space Complexity

$$O(1)$$

- Board is modified in-place
- Only recursion stack is used

## 25. Expression Add Operators

---

### Question Explanation

You are given:

- A string num consisting only of digits
- An integer target

Your task is to **insert binary operators (+, -, \*) between the digits** of num such that:

- The resulting mathematical expression evaluates exactly to target
- Operands must **not contain leading zeros** (except the number "0" itself)
- Digits must remain in the same order

You must return **all valid expressions**.

---

### Example

#### Example 1

Input

```
num = "123", target = 6
```

### Output

```
["1*2*3", "1+2+3"]
```

## Example 2

### Input

```
num = "232", target = 8
```

### Output

```
["2*3+2", "2+3*2"]
```

---

## Approach (DFS + Backtracking)

---

### Core Idea

We generate **all possible expressions** by:

- Splitting the string into numbers
- Inserting operators between them
- Evaluating the expression **on the fly** using recursion

This avoids building and re-evaluating expressions repeatedly.

---

### Important Observations

#### 1. Leading Zero Rule

- "0" is allowed

- "05", "00" are NOT allowed  
→ If a number starts with '0' and has more than one digit, skip it.

## 2. Operator Precedence

- \* has higher precedence than + and -
  - To handle this, we track:
    - current\_value: total value so far
    - last\_operand: last number added (used to fix multiplication)
- 

## Recursive DFS Parameters

| Parameter     | Meaning                   |
|---------------|---------------------------|
| start         | Current index in string   |
| expression    | Expression built so far   |
| current_value | Evaluated value till now  |
| last_operand  | Last operand used (for *) |

---

## Base Case

When `start == num.length()`:

- If `current_value == target`
  - Store the expression
-

## Recursive Choices

For each substring starting at `start`:

1. Take substring as a number
  2. If first number → start expression
  3. Else try:
    - +
    - -
    - \* (adjust using `last_operand`)
- 

## Code (C++)

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 void dfs(string& num, int target, int start,
 long long current_value, long long last_operand,
 string expression, vector<string>& result) {

 // Base case
 if (start == num.size()) {
 if (current_value == target)
 result.push_back(expression);
 return;
 }

 // Try all possible numbers starting at 'start'
 for (int i = start; i < num.size(); i++) {

 // Leading zero check
 if (num[i] != '0' || (i > start && num[i-1] != '0')) {
 long long operand = stoll(num.substr(start, i - start));
 long long new_current_value = current_value + operand;
 string new_expression = expression + "+" + num.substr(start, i - start);

 dfs(num, target, i + 1, new_current_value, operand, new_expression, result);

 new_current_value = current_value - operand;
 new_expression = expression + "-" + num.substr(start, i - start);

 dfs(num, target, i + 1, new_current_value, operand, new_expression, result);

 new_current_value = current_value * operand;
 new_expression = expression + "*" + num.substr(start, i - start);

 dfs(num, target, i + 1, new_current_value, operand, new_expression, result);
 }
 }
 }
}
```

```

 if (i > start && num[start] == '0') return;

 string part = num.substr(start, i - start + 1);
 long long value = stoll(part);
 }

 // In C++, std::stol() and std::stoll() are the library functions used to convert the given string to integer
 // value of type long int and long long int respectively
 if (start == 0) {
 // First number, no operator
 dfs(num, target, i + 1, value, value, part, result);
 } else {
 // Addition
 dfs(num, target, i + 1,
 current_value + value, value,
 expression + "+" + part, result);

 // Subtraction
 dfs(num, target, i + 1,
 current_value - value, -value,
 expression + "-" + part, result);

 // Multiplication
 dfs(num, target, i + 1,
 current_value - last_operand + last_operand *
 value,
 last_operand * value,
 expression + "*" + part, result);
 }
}

vector<string> addOperators(string num, int target) {
 vector<string> result;
 dfs(num, target, 0, 0, 0, "", result);
 return result;
}

int main() {
 string num = "123";
}

```

```
int target = 6;

Solution sol;
vector<string> ans = sol.addOperators(num, target);

for (auto &s : ans)
 cout << s << " ";

return 0;
}
```

---

## Complexity Analysis

### Time Complexity

$O(4^n)$

- At each position, we explore multiple splits and operators

### Space Complexity

$O(n)$

- Recursion depth up to string length
- Expression string built incrementally

# **Bit Manipulation**

# 1. Introduction to Bit Manipulation [Theory]

## Introduction

This article serves as an introduction to some fundamental concepts in binary number manipulation, including binary number conversion, one's complement, and two's complement and many more concepts. These concepts are not only fundamental to understanding how computers work but also crucial for solving a variety of problems in computer science.

Having an understanding of binary numbers is extremely important in the field of computer science as everything stored in a computer is in the form of only 0s and 1s.

## Binary Number Conversion

### Decimal to Binary Conversion:

By repeatedly dividing a number by 2 and recording the result, decimal values can be transformed into binary.

**Example:** Converting 13 to its binary equivalent:

- Start with the decimal number 13.
- Divide the number by 2 and record the remainder.
- Repeat the division with the quotient until the number becomes 0.

$$13 \div 2 = 6 \text{ remainder } 1$$

$$6 \div 2 = 3 \text{ remainder } 0$$

$$3 \div 2 = 1 \text{ remainder } 1$$

$$1 \div 2 = 0 \text{ remainder } 1$$

To obtain the binary equivalent of 13, read the remainders from bottom to top: **1101**.

**So, the binary equivalent of 13 is 1101.**

### Binary to Decimal Conversion:

Converting a binary number back to its decimal equivalent involves a reverse process.

**Example:** Converting 1101 to its decimal equivalent:

- Start from the rightmost bit (least significant bit).
- Each bit is multiplied by 2 raised to the power of its position index.

$$1 * 2^0 = 1$$

$$0 * 2^1 = 0$$

```
1 * 2^2 = 4
1 * 2^3 = 8
```

Sum = 1 + 0 + 4 + 8 = **13**.

Hence, the decimal equivalent of the binary number 1101 is **13**.

## Understanding One's Complement and Two's Complement

### One's Complement

The one's complement of a binary number is obtained by flipping all the bits.

**Example:** The one's complement of 13 (binary 1101):

|                  |   |           |
|------------------|---|-----------|
| Binary of 13     | : | 0000 1101 |
| One's Complement | : | 1111 0010 |

### Two's Complement

The two's complement is obtained by taking the one's complement of a number and adding 1.

**Example:** The two's complement of 13 (binary 1101):

|                  |   |           |
|------------------|---|-----------|
| One's Complement | : | 1111 0010 |
| Add 1            | : | 1111 0011 |

## Bitwise Operators

### AND Operator (&)

If both corresponding bits are 1, the resulting bit is 1; otherwise, it is 0.

|     |          |
|-----|----------|
| 13: | 1101     |
| 7:  | 0111     |
| & : | 0101 → 5 |

## **OR Operator (|)**

If either corresponding bit is 1, the resulting bit is 1.

```
13: 1101
7: 0111
| : 1111 → 15
```

## **XOR Operator (^)**

If bits differ, the result is 1; if the same, result is 0.

```
13: 1101
7: 0111
^ : 1010 → 10
```

## **NOT Operator (~)**

Flips all bits of the number.

```
5: 0000 0101
~5: 1111 1010 → -6 (in two's complement)
```

## **Shift Operators**

**Right Shift (>>):** Shifts bits to the right, fills left with 0s.

```
13 >> 1 = 0110 → 6
```

**Left Shift (<<):** Shifts bits to the left, fills right with 0s.

```
13 << 1 = 11010 → 26
```

## **Bit Manipulation Tricks and Techniques**

### **1. Swapping Two Numbers Without a Third Variable**

```
A = A ^ B
```

```
B = A ^ B (old A ^ old B) ^ old B = old A
A = A ^ B (old A ^ old B) ^ old A = old B
```

## 2. Checking if the i-th Bit is Set

```
(1 << i) & num → set if result ≠ 0
(num >> i) & 1 → set if result ≠ 0
```

## 3. Setting the i-th Bit

```
num | (1 << i)
```

## 4. Clearing the i-th Bit

```
num & ~(1 << i)
```

## 5. Toggling the i-th Bit

```
num ^ (1 << i)
```

# 2. Check if the i-th bit is set or not

Given two integers n and i, you need to check whether the i-th bit in the binary representation of n is set (equal to 1). Bit indexing starts from the least significant bit (rightmost) and is 0-based. If the bit is 1, return true, otherwise return false.

Example:

If n = 5, its binary form is 101.

Indexing from right:

0-th bit → 1

1-st bit → 0

2-nd bit → 1

So for i = 0, answer is true.

For i = 1, answer is false.

---

## Approach 1: Brute Force

### Algorithm

First convert the given number n into its binary representation.  
Store each bit starting from the least significant bit.  
Once the binary form is ready, check whether the index i exists.  
If i is greater than or equal to the binary length, the bit is not set.  
Otherwise, check if the bit at index i is equal to '1'.

### Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 bool checkIthBit(int n,int i) {
 string binary="";
 while(n>0){
 if(n%2==0) binary+='0';
 else binary+='1';
 n/=2;
 }
 if(i>=binary.size()) return false;
 return binary[i]=='1';
 }
};

int main(){
 Solution sol;
 int n=5;
 int i=0;
 cout<<sol.checkIthBit(n,i);
 return 0;
}
```

## **Complexity Analysis**

Time Complexity:  $O(\log n)$ , because the number is converted to binary.

Space Complexity:  $O(\log n)$ , for storing the binary representation.

---

## **Approach 2: Optimal Approach (Bit Masking)**

### **Algorithm**

Create a mask by left shifting 1 by  $i$  positions.

This mask has only the  $i$ -th bit set.

Perform a bitwise AND between  $n$  and the mask.

If the result is non-zero, the  $i$ -th bit is set.

If the result is zero, the  $i$ -th bit is not set.

### **Code**

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 bool checkIthBit(int n,int i) {
 return (n & (1<<i))!=0;
 }
};

int main(){
 Solution sol;
 int n=5;
 int i=0;
 cout<<sol.checkIthBit(n,i);
 return 0;
}
```

## **Complexity Analysis**

Time Complexity:  $O(1)$ , only one bitwise operation is used.

Space Complexity:  $O(1)$ , no extra space is required.

### 3. Check if a number is odd or not

Given a non-negative integer n, determine whether the number is odd.

A number is considered odd if it is not divisible by 2, meaning when divided by 2 it leaves a remainder.

If the number is odd, return true, otherwise return false.

Example:

If n = 7, dividing by 2 leaves remainder 1, so it is odd.

If n = 10, dividing by 2 leaves no remainder, so it is not odd.

---

#### Approach 1

##### Algorithm

An odd number always leaves a remainder when divided by 2.

Check the remainder of n when divided by 2.

If the remainder is not equal to 0, the number is odd.

Otherwise, the number is even.

##### Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 bool isOdd(int n) {
 return (n % 2 != 0);
 }
};

int main() {
 Solution sol;
 int n = 7;
 cout << sol.isOdd(n);
 return 0;
}
```

### **Complexity Analysis**

Time Complexity: O(1), only one modulus operation is performed.

Space Complexity: O(1), no extra space is used.

## **4. Check if a number is power of 2 or not**

Given an integer n, determine whether it is a power of two.

A number is called a power of two if it can be written as 2 raised to some integer power, like 1, 2, 4, 8, 16, and so on.

If n satisfies this condition, return true, otherwise return false.

Example:

If n = 16, it can be written as  $2^4$ , so it is a power of two.

If n = 3, it cannot be written as 2 raised to any integer, so it is not a power of two.

---

### **Approach 1**

#### **Algorithm**

A power of two has exactly one bit set to 1 in its binary representation.

When we subtract 1 from such a number, all bits after that single set bit become 1.

Doing a bitwise AND between the number and one less than the number gives 0 only for powers of two.

Also, n must be greater than 0 because non-positive numbers cannot be powers of two.

#### **Code**

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 bool isPowerOfTwo(int n) {
 return n>0 && (n&(n-1))==0; N>0 jaruri
 }
}
```

```

};

int main() {
 Solution sol;
 int n=16;
 cout<<sol.isPowerOfTwo(n);
 return 0;
}

```

### **Complexity Analysis**

Time Complexity: O(1), only constant-time bitwise operations are used.

Space Complexity: O(1), no extra space is required.

## **5. Count the number of set bits**

Given an integer n, you need to count how many bits are set to 1 in its binary representation.

A set bit means a bit with value 1.

Return the total count of such bits.

Example:

If n = 5, its binary representation is 101. There are two 1s, so the answer is 2.

If n = 15, its binary representation is 1111. There are four 1s, so the answer is 4.

---

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

#### **Algorithm**

Initialize a counter to store the number of set bits.

While the number is greater than 0:

Check the least significant bit using bitwise AND with 1.

If it is 1, increment the counter.

Right shift the number by 1 to move to the next bit.

Repeat until all bits are checked.

Return the counter.

#### **Code**

```

#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int countSetBits(int n) {
 int count=0;
 while(n>0){
 if(n&1) count++;
 n>>=1;
 }
 return count;
 }
};

int main(){
 Solution sol;
 int n=5;
 cout<<sol.countSetBits(n);
 return 0;
}

```

### Complexity Analysis

Time Complexity:  $O(\log n)$ , each bit of the number is checked once.

Space Complexity:  $O(1)$ , only a counter variable is used.

---

## Approach 2: Optimal Approach (Brian Kernighan's Algorithm)

### Algorithm

Initialize a counter to store the number of set bits.

While the number is greater than 0:

    Remove the rightmost set bit by doing  $n = n \& (n - 1)$ .

    Increment the counter each time a set bit is removed.

    Repeat until the number becomes 0.

    Return the counter.

This works because each operation removes exactly one set bit.

Ex: n = 1100

n - 1 = 1011

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int countSetBits(int n) {
 int count=0;
 while(n>0){
 n=n&(n-1);
 count++;
 }
 return count;
 }
};

int main(){
 Solution sol;
 int n=5;
 cout<<sol.countSetBits(n);
 return 0;
}
```

## Complexity Analysis

Time Complexity: O(k), where k is the number of set bits in n.

Space Complexity: O(1), no extra space is used.

# 6. Set the rightmost bit

Given a positive integer  $n$ , you need to set the rightmost unset (0) bit in its binary representation to 1 and return the new number.

If all the bits of the number are already set to 1, then the number should remain unchanged.

Example:

If  $n = 10$ , its binary form is 1010.

The rightmost unset bit is at position 0. After setting it to 1, the binary becomes 1011, which is 11.

If  $n = 7$ , its binary form is 111.

All bits are already set, so the result remains 7.

---

## Approach 1

### Algorithm

Add 1 to the number  $n$ .

When 1 is added, the rightmost unset bit in  $n$  becomes set, and all bits to its right become 0.

Perform a bitwise OR between  $n$  and  $(n + 1)$ .

This ensures the rightmost unset bit is set to 1 while all other bits remain unchanged.

Return the result.

$n = 100111$

$n + 1 = 101000$

### Code

```
#include <bits/stdc++.h>
using namespace std;

int setRightmostUnsetBit(int n) {
 return n|(n+1);
}

int main() {
 int n=10;
 cout<<setRightmostUnsetBit(n);
```

```
 return 0;
}
```

### Complexity Analysis

Time Complexity: O(1), only constant-time bitwise operations are used.

Space Complexity: O(1), no extra space is required.

## 7. Swap two numbers

Given two integers a and b, swap their values in-place using only two variables.  
You are not allowed to use any temporary variable.

Example:

If a = 5 and b = 10, after swapping a becomes 10 and b becomes 5.

If a = -100 and b = -200, after swapping a becomes -200 and b becomes -100.

---

### Approach 1

#### Algorithm

Use the XOR operator to swap values without extra space.

First, store  $a \wedge b$  in a.

Then store  $(a \wedge b) \wedge b$  in b, which gives the original value of a.

Finally store  $(a \wedge b) \wedge a$  in a, which gives the original value of b.

This works because XOR of the same numbers becomes 0 and XOR with 0 gives the number itself.

#### Code

```
#include <bits/stdc++.h>
using namespace std;

void swapXOR(int &a, int &b){
 a=a^b;
 b=a^b;
 a=a^b;
```

```

}

int main(){
 int a=5,b=10;
 swapXOR(a,b);
 cout<<"a = "<<a<<, b = "<<b;
 return 0;
}

```

### **Complexity Analysis**

Time Complexity: O(1), only constant number of operations are performed.

Space Complexity: O(1), no extra space is used.

## **8. Divide two integers without using multiplication, division and mod operator**

Given two integers dividend and divisor, divide them without using multiplication, division, or modulo operators.

The result should be truncated toward zero, meaning the fractional part is removed.

If the result overflows the 32-bit signed integer range, return the maximum or minimum allowed value.

Example:

If dividend = 10 and divisor = 3, the division result is 3.33, which becomes 3 after truncation.

If dividend = 7 and divisor = -3, the division result is -2.33, which becomes -2.

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

#### **Algorithm**

First, determine the sign of the final answer based on the signs of dividend and divisor.

Convert both numbers to their absolute values to simplify calculations.

Repeatedly subtract the divisor from the dividend until the remaining value is smaller than the divisor.

Count how many times the subtraction is possible; this count is the quotient.

Handle overflow cases according to the problem statement.

Apply the correct sign to the final answer and return it.

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int divide(int dividend,int divisor){
 if(dividend==INT_MIN && divisor== -1) return INT_MAX;
 bool isPositive=true;
 if((dividend<0 && divisor>0)|| (dividend>0 && divisor<0))
 isPositive=false;
 long long n=llabs((long long)dividend);
 long long d=llabs((long long)divisor);
 long long ans=0,sum=0;
 while(sum+d<=n){
 sum+=d;
 ans++;
 }
 if(ans>INT_MAX && isPositive) return INT_MAX;
 if(ans>INT_MAX && !isPositive) return INT_MIN;
 return isPositive?ans:-ans;
 }
};

int main(){
 Solution sol;
 int dividend=10,divisor=3;
 cout<<sol.divide(dividend,divisor);
 return 0;
}
```

## Complexity Analysis

Time Complexity: O(dividend), in the worst case when divisor is 1.

Space Complexity: O(1), only constant extra variables are used.

---

## Approach 2: Optimal Approach (Using Bit Shifting)

### Algorithm

Determine the sign of the result based on the signs of dividend and divisor.

Convert both numbers to positive values using absolute values.

Instead of subtracting the divisor one by one, subtract the largest possible power-of-two multiple of the divisor.

Use left shifting to find the largest multiple of the divisor that is less than or equal to the remaining dividend.

Subtract that value from the dividend and add the corresponding power of two to the answer.

Repeat this process until the dividend becomes smaller than the divisor.

Clamp the result if it exceeds the 32-bit signed integer range.

Apply the correct sign and return the final quotient.



**dividend = 22**

**divisor = 3**

$$22 = 3 \times 2^2 + 3 \times 2^1 + 3 \times 2^0$$

$$\begin{array}{r} 22 \\ - 12 \\ \hline 10 \end{array} \quad \begin{array}{r} 10 \\ - 6 \\ \hline 4 \end{array} \quad \begin{array}{r} 4 \\ - 3 \\ \hline 1 \end{array}$$

The diagram shows the division of 22 by 3 using bit shifting. It is broken down into three steps: 1) Subtracting 12 (3 \* 2^2) from 22, resulting in 10. 2) Subtracting 6 (3 \* 2^1) from 10, resulting in 4. 3) Subtracting 3 (3 \* 2^0) from 4, resulting in 1. Red dashed boxes group the subtractions, and red arrows point from the first subtraction to the second, and from the second to the third.

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int divide(int dividend,int divisor){
 if(dividend==INT_MIN && divisor== -1) return INT_MAX;
 bool isPositive=true;
 if((dividend<0 && divisor>0)|| (dividend>0 && divisor<0))
 isPositive=false;
 long long n=llabs((long long)dividend);
 long long d=llabs((long long)divisor);
 long long ans=0;
 while(n>=d){
 long long temp=d;
 long long multiple=1;
 while((temp<<1)<=n){
 temp<<=1;
 multiple<<=1;
 }
 n-=temp;
 ans+=multiple;
 }
 if(ans>INT_MAX && isPositive) return INT_MAX;
 if(ans>INT_MAX && !isPositive) return INT_MIN;
 return isPositive?ans:-ans;
 }
};

int main(){
 Solution sol;
 int dividend=10,divisor=3;
 cout<<sol.divide(dividend,divisor);
 return 0;
}
```

### **Complexity Analysis**

Time Complexity:  $O((\log N)^2)$ , where N is the absolute value of the dividend.

Space Complexity:  $O(1)$ , only constant extra variables are used.

## **9. Count number of bits to be flipped to convert A to B**

Given two integers start and goal, you need to find the minimum number of bit flips required to convert start into goal.

A bit flip means changing a bit from 0 to 1 or from 1 to 0 in the binary representation.

Example:

If start = 10 and goal = 7

Binary of 10 is 1010

Binary of 7 is 0111

The bits at three positions are different, so 3 flips are needed.

If start = 3 and goal = 4

Binary of 3 is 011

Binary of 4 is 100

All three bits are different, so 3 flips are needed.

---

### **Approach 1**

#### **Algorithm**

To find differing bits between two numbers, compare them bit by bit.

Use XOR between start and goal.

XOR produces 1 at positions where the bits are different.

Each set bit in the XOR result represents one required flip.

Count the number of set bits in the XOR result to get the answer.

#### **Code**

```
#include <bits/stdc++.h>
using namespace std;
```

```

class Solution {
public:
 int minBitsFlip(int start,int goal){
 int num=start^goal;
 int count=0;
 for(int i=0;i<32;i++){
 count+=(num&1);
 num>>=1;
 }
 return count;
 }
};

int main(){
 Solution sol;
 int start=10,goal=7;
 cout<<sol.minBitsFlip(start,goal);
 return 0;
}

```

### **Complexity Analysis**

Time Complexity: O(1), the loop always runs for 32 bits.

Space Complexity: O(1), only constant extra variables are used.

## **10. Find the number that appears odd number of times**

Given an array of integers where every element appears exactly twice except one element that appears only once, find and return that unique element.

The problem guarantees that exactly one such element exists.

Example:

For nums = [1, 2, 2, 4, 3, 1, 4]

All numbers appear twice except 3, so the answer is 3.

For nums = [5]

Only one element exists and it appears once, so the answer is 5.

---

## Approach 1: Brute Force

### Algorithm

Use a hashmap to store the frequency of each number.

Traverse the array and for each element, increase its count in the map.

After filling the map, traverse it to find the element whose frequency is exactly 1.

Return that element.

If no such element is found (though the problem guarantees one), return -1.

### Code

```
#include <bits/stdc++.h>
using namespace std;

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

int main(){
 vector<int> nums={1,2,2,4,3,1,4};
 Solution sol;
```

```

 cout<<sol.singleNumber(nums);
 return 0;
}

```

### Complexity Analysis

Time Complexity: O(N), where N is the size of the array.

Space Complexity: O(N), hashmap stores frequencies of elements.

---

## Approach 2: Optimal Approach (Using XOR)

### Algorithm

Initialize a variable XOR to 0.

Traverse the array and XOR each element with this variable.

XOR of two equal numbers is 0, so all numbers appearing twice cancel out.

The number that appears once remains as the final XOR value.

Return this value.

### Code

```

#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int singleNumber(vector<int>& nums) {
 int XOR=0;
 for(int i=0;i<nums.size();i++){
 XOR^=nums[i];
 }
 return XOR;
 }
};

int main(){
 vector<int> nums={1,2,2,4,3,1,4};
 Solution sol;
 cout<<sol.singleNumber(nums);
 return 0;
}

```

### **Complexity Analysis**

Time Complexity:  $O(N)$ , the array is traversed once.

Space Complexity:  $O(1)$ , only one variable is used.

## **11. Power Set | Bit Manipulation**

Given an array of integers, generate and print all possible subsets of the array using bitwise operations.

A power set contains every possible combination of elements, including the empty set and the full array.

Example:

If  $\text{nums} = [1, 2, 3]$ , there are 3 elements.

The total number of subsets is  $2^3 = 8$ .

Each subset can be represented using a binary number from 0 to 7, where each bit tells whether to include an element or not.

For example, binary 101 means include index 0 and index 2 → subset [1, 3].

---

### **Approach 1**

#### **Algorithm**

Let  $N$  be the size of the array.

The total number of subsets is  $2^n$ , which can be computed as  $(1 << n)$ .

Each number from 0 to  $(2^n - 1)$  represents one subset.

For each number, check its binary representation.

If the  $i$ -th bit is set, include  $\text{nums}[i]$  in the current subset.

Store each generated subset in the result list.

Return all subsets.

#### **Code**

```
#include <bits/stdc++.h>
using namespace std;
```

```

class Solution {
public:
 vector<vector<int>> getPowerSet(vector<int>& nums) {
 int n=nums.size();
 int total=1<<n;
 vector<vector<int>> ans;
 for(int mask=0;mask<total;mask++){
 vector<int> subset;
 for(int i=0;i<n;i++){
 if(mask&(1<<i)){
 subset.push_back(nums[i]);
 }
 }
 ans.push_back(subset);
 }
 return ans;
 }
};

int main(){
 vector<int> nums={5,7,8};
 Solution sol;
 vector<vector<int>> res=sol.getPowerSet(nums);
 for(auto subset:res){
 cout<<"[";
 for(int x:subset) cout<<x<<" ";
 cout<<"] "<<endl;
 }
 return 0;
}

```

### Complexity Analysis

Time Complexity:  $O(N \times 2^n)$ , for each of the  $2^n$  subsets we may scan all N elements.

Space Complexity:  $O(N \times 2^n)$ , to store all subsets of the power set.

# 12. Find XOR of numbers from L to R

Given two integers L and R, find the XOR of all numbers in the inclusive range [L, R].

XOR is a bitwise operation where equal bits cancel each other and different bits produce 1.

Example:

If L = 3 and R = 5, the result is  $3 \wedge 4 \wedge 5 = 2$ .

If L = 1 and R = 3, the result is  $1 \wedge 2 \wedge 3 = 0$ .

---

## Approach 1: Brute Force

### Algorithm

Initialize a variable ans with 0.

Traverse all numbers from L to R.

For each number, take XOR with ans and update ans.

After finishing the loop, ans contains the XOR of all numbers in the range.

Return ans.

### Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int findRangeXOR(int l,int r){
 int ans=0;
 for(int i=l;i<=r;i++){
 ans^=i;
 }
 return ans;
 }
};

int main(){
 Solution sol;
 int l=3,r=5;
```

```

 cout<<sol.findRangeXOR(l,r);
 return 0;
}

```

### Complexity Analysis

Time Complexity: O(N), where N = R - L + 1.

Space Complexity: O(1), only one variable is used.

---

## Approach 2: Optimal Approach

### Algorithm

The XOR of numbers from 1 to n follows a fixed pattern based on  $n \% 4$ .

If  $n \% 4 == 0$ , XOR from 1 to n is n.

If  $n \% 4 == 1$ , XOR from 1 to n is 1.

If  $n \% 4 == 2$ , XOR from 1 to n is  $n + 1$ .

If  $n \% 4 == 3$ , XOR from 1 to n is 0.

To find XOR from L to R:

Compute XOR from 1 to R.

Compute XOR from 1 to L - 1.

XOR these two results to cancel common values and get the answer.

### Code

```

#include <bits/stdc++.h>
using namespace std;

class Solution {
 int XORtillN(int n){
 if(n%4==0) return n;
 if(n%4==1) return 1;
 if(n%4==2) return n+1;
 return 0;
 }
public:
 int findRangeXOR(int l,int r){
 return XORtillN(l-1)^XORtillN(r);
 }
};

```

```

int main(){
 Solution sol;
 int l=3,r=5;
 cout<<sol.findRangeXOR(l, r);
 return 0;
}

```

### **Complexity Analysis**

Time Complexity: O(1), constant time computation.

Space Complexity: O(1), no extra space is used.

## **13. Find the two numbers appearing odd number of times**

Given an array of integers where every element appears exactly twice except for two elements that appear only once, find those two unique numbers.

Return the two numbers in ascending order.

Example:

For nums = [1, 2, 1, 3, 5, 2]

All numbers appear twice except 3 and 5, so the answer is [3, 5].

For nums = [-1, 0]

Both numbers appear once, so the answer is [-1, 0].

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

#### **Algorithm**

Use a hashmap to store the frequency of each element.

Traverse the array and update the frequency of each number.

After that, traverse the hashmap and collect the elements whose frequency is 1.

Sort the result containing the two numbers.

Return the sorted result.

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 vector<int> singleNumber(vector<int>& nums){
 unordered_map<int, int> mpp;
 for(int i=0;i<nums.size();i++){
 mpp[nums[i]]++;
 }
 vector<int> ans;
 for(auto it:mpp){
 if(it.second==1){
 ans.push_back(it.first);
 }
 }
 sort(ans.begin(),ans.end());
 return ans;
 }
};

int main(){
 vector<int> nums={1,2,1,3,5,2};
 Solution sol;
 vector<int> ans=sol.singleNumber(nums);
 cout<<ans[0]<<" "<<ans[1];
 return 0;
}
```

## Complexity Analysis

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

Space Complexity:  $O(N)$ , hashmap is used to store frequencies.

---

## Approach 2: Optimal Approach (Using XOR)

## Algorithm

XOR all elements of the array. Duplicate elements cancel out, leaving XOR of the two unique numbers.

Find the rightmost set bit in this XOR result. This bit differs between the two unique numbers.

Traverse the array again and divide numbers into two groups based on this bit.

XOR numbers within each group separately.

Each group will give one unique number.

Sort the two numbers and return them.

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 vector<int> singleNumber(vector<int>& nums) {
 long x=0;
 for(int i=0;i<nums.size();i++){
 x^=nums[i];
 }
 int rightmost=(x&(x-1))^x;
 int a=0,b=0;
 for(int i=0;i<nums.size();i++){
 if(nums[i]&rightmost) a^=nums[i];
 else b^=nums[i];
 }
 if(a<b) return {a,b};
 return {b,a};
 }
};

int main(){
 vector<int> nums={1,2,1,3,5,2};
 Solution sol;
 vector<int> ans=sol.singleNumber(nums);
 cout<<ans[0]<<" "<<ans[1];
 return 0;
}
```

## **Complexity Analysis**

Time Complexity: O(N), the array is traversed twice.

Space Complexity: O(1), only constant extra variables are used.

# **14. Prime factors of a number**

Given a number n, find all its prime factors.

A prime factor is a prime number that divides n exactly.

A prime number is a natural number greater than 1 that has exactly two factors: 1 and itself.

Example:

If n = 18, its prime factorization is  $2 \times 3 \times 3$ , so the output is [2, 3, 3].

If n = 25, its prime factorization is  $5 \times 5$ , so the output is [5, 5].

---

## **Approach 1: Naive Approach (Iterative)**

### **Algorithm**

Start checking divisibility from i = 2 up to n.

For each i, if n is divisible by i, then i is a prime factor.

Keep dividing n by i as long as  $n \% i == 0$  and store i each time.

Continue this process until i reaches n.

All stored values are the prime factors.

### **Code**

```
#include <bits/stdc++.h>
using namespace std;

vector<int> primeFactor(int n){
 vector<int> ans;
 for(int i=2;i<=n;i++){
 while(n%i==0 && n>0){
 ans.push_back(i);
 n=n/i;
 }
 }
}
```

```

 }
 return ans;
}

int main(){
 int n=18;
 vector<int> ans=primeFactor(n);
 for(int x:ans){
 cout<<x<<" ";
 }
 return 0;
}

```

### Complexity Analysis

Time Complexity:  $O(n)$ , in the worst case we try all numbers from 2 to  $n$ .

Space Complexity:  $O(1)$ , excluding the space used to store the result.

---

## Approach 2: Expected Approach (Using Square Root)

### Algorithm

First, remove all factors of 2 by repeatedly dividing  $n$  by 2 and storing 2 each time.

After this step,  $n$  becomes odd.

Now check only odd numbers starting from 3 up to  $\sqrt{n}$ .

For each  $i$ , while  $n$  is divisible by  $i$ , store  $i$  and divide  $n$  by  $i$ .

Increment  $i$  by 2 each time to skip even numbers.

After the loop, if  $n$  is greater than 2, then  $n$  itself is a prime factor and should be added.

### Code

```

#include <bits/stdc++.h>
using namespace std;

vector<int> primeFactors(int n){
 vector<int> factors;

 while(n%2==0){
 factors.push_back(2);
 n=n/2;
 }

```

```

for(int i=3;i*i<=n;i=i+2){
 while(n%i==0){
 factors.push_back(i);
 n=n/i;
 }
}

if(n>2){
 factors.push_back(n);
}

return factors;
}

int main(){
 int n=18;
 vector<int> ans=primeFactors(n);
 for(int x:ans){
 cout<<x<<" ";
 }
 return 0;
}

```

### **Complexity Analysis**

Time Complexity:  $O(\sqrt{n})$ , because we only iterate up to the square root of n.

Space Complexity:  $O(1)$ , excluding the space used to store the result.

## **15. All Divisors of a Number**

You are given an integer n. You need to find all the divisors of n and return them in sorted order.

A divisor of a number is a value that divides the number completely without leaving any remainder.

Example:

If  $n = 6$ , the numbers that divide 6 exactly are 1, 2, 3, and 6.

If  $n = 8$ , the numbers that divide 8 exactly are 1, 2, 4, and 8.

---

## Approach 1: Brute Force

### Algorithm

Loop from 1 to  $n$ .

For each number  $i$ , check if  $n \% i == 0$ .

If yes, then  $i$  is a divisor of  $n$ .

Store all such values in a list.

Return the list.

### Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 vector<int> divisors(int n){
 vector<int> ans;
 for(int i=1;i<=n;i++){
 if(n%i==0){
 ans.push_back(i);
 }
 }
 return ans;
 }
};

int main(){
 int n=6;
 Solution sol;
 vector<int> ans=sol.divisors(n);
 for(int x:ans){
 cout<<x<<" ";
 }
 return 0;
}
```

}

### Complexity Analysis

Time Complexity:  $O(N)$ , since we check all numbers from 1 to n.

Space Complexity:  $O(\sqrt{N})$ , as a number can have at most  $2\sqrt{N}$  divisors.

---

## Approach 2: Optimal Approach

### Algorithm

Divisors come in pairs. If i divides n, then  $(n / i)$  is also a divisor.

Iterate only from 1 to  $\sqrt{n}$ .

Whenever i divides n, add i to the list.

If i is not equal to  $n / i$ , also add  $n / i$ .

After collecting all divisors, sort the list.

Return the sorted list.

### Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 vector<int> divisors(int n){
 vector<int> ans;
 int limit=sqrt(n);
 for(int i=1;i<=limit;i++){
 if(n%i==0){
 ans.push_back(i);
 if(i!=n/i){ //duplicate avoid karna
 ans.push_back(n/i);
 }
 }
 }
 sort(ans.begin(),ans.end());
 return ans;
 }
};
```

```

int main(){
 int n=6;
 Solution sol;
 vector<int> ans=sol.divisors(n);
 for(int x:ans){
 cout<<x<<" ";
 }
 return 0;
}

```

### **Complexity Analysis**

Time Complexity:  $O(\sqrt{N} + K \log K)$ , where  $K$  is the number of divisors (sorting step).

Space Complexity:  $O(\sqrt{N})$ , used to store the divisors.

## **16. Sieve of Eratosthenes**

You are given a 2D array queries of size  $n \times 2$ .

Each query  $[L, R]$  asks how many prime numbers exist in the range from  $L$  to  $R$  (inclusive).

You must return an array where each element is the answer for the corresponding query.

A prime number is a number greater than 1 that has exactly two factors: 1 and itself.

Example:

For queries = [[2,5],[4,7]]

The range 2 to 5 has primes 2, 3, 5 → count = 3

The range 4 to 7 has primes 5, 7 → count = 2

### **Approach 1: Sieve of Eratosthenes + Prefix Sum**

#### **Algorithm**

First, find the maximum  $R$  value among all queries.

Create a boolean array  $isPrime$  from 0 to  $\maxR$  and mark all values as true initially.

Set  $isPrime[0]$  and  $isPrime[1]$  to false because they are not prime.

Using the Sieve of Eratosthenes, mark all non-prime numbers as false.

Build a prefix sum array primeCount where primeCount[i] stores how many primes exist from 1 to i.

For each query [L, R], the number of primes is primeCount[R] – primeCount[L – 1].

Store the result for every query and return it.

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 vector<int> primesInRange(vector<vector<int>>& queries) {
 int maxVal = 0;
 for(auto q : queries){
 maxVal = max(maxVal, q[1]);
 }

 vector<bool> isPrime(maxVal + 1, true);
 if(maxVal >= 0) isPrime[0] = false;
 if(maxVal >= 1) isPrime[1] = false;

 for(int i = 2; i * i <= maxVal; i++){
 if(isPrime[i]){
 for(int j = i * i; j <= maxVal; j += i){
 isPrime[j] = false;
 }
 }
 }

 vector<int> primeCount(maxVal + 1, 0);
 for(int i = 1; i <= maxVal; i++){
 primeCount[i] = primeCount[i - 1];
 if(isPrime[i]) primeCount[i]++;
 }

 vector<int> result;
```

```

 for(auto q : queries){
 int L = q[0];
 int R = q[1];
 if(L == 0)
 result.push_back(primeCount[R]);
 else
 result.push_back(primeCount[R] - primeCount[L - 1]);
 }

 return result;
 }
};

int main(){
 vector<vector<int>> queries = {{2,5},{4,7}};
 Solution sol;
 vector<int> ans = sol.primesInRange(queries);
 for(int x : ans){
 cout << x << " ";
 }
 return 0;
}

```

### Complexity Analysis

Time Complexity:  $O(N \log \log N)$ , where  $N$  is the maximum value in the queries, due to the Sieve of Eratosthenes.

Space Complexity:  $O(N)$ , for storing the prime array and prefix sum array.

# 17. Find Prime Factorisation of a Number using Sieve

You are given an integer array queries.

For each number in the array, you need to return its prime factorization in sorted order.

Prime factorization means breaking a number into a product of prime numbers.

Example:

If queries = [2, 3, 4, 5, 6]

2, 3, and 5 are already prime.

$4 = 2 \times 2$

$6 = 2 \times 3$

So the result is [[2], [3], [2, 2], [5], [2, 3]].

---

## Approach 1: Sieve using Smallest Prime Factor (SPF)

### Algorithm

Create an array SPF where SPF[i] stores the smallest prime factor of i.

Initially, set all values of SPF as 1.

Use a modified Sieve of Eratosthenes:

Traverse numbers from 2 to MAX\_N.

If SPF[i] is still 1, then i is a prime number.

Mark i as the smallest prime factor for all its multiples that are not marked yet.

To find prime factorization of a number n:

Repeatedly take SPF[n], add it to the answer, and divide n by SPF[n].

Continue until n becomes 1.

Apply this process for every number in the queries array.

### Code

```
#include <bits/stdc++.h>
using namespace std;

#define MAX_N 100000

vector<int> SPF(MAX_N+1, 1);
```

```

class Solution{
private:
 void sieve(){
 for(int i=2;i<=MAX_N;i++){
 if(SPF[i]==1){
 for(int j=i;j<=MAX_N;j+=i){
 if(SPF[j]==1){
 SPF[j]=i;
 }
 }
 }
 }
 }

 vector<int> primeFact(int n){
 vector<int> ans;
 while(n!=1){
 ans.push_back(SPF[n]);
 n=n/SPF[n];
 }
 return ans;
 }

public:
 vector<vector<int>> primeFactors(vector<int>& queries){
 sieve();
 vector<vector<int>> ans;
 for(int i=0;i<queries.size();i++){
 ans.push_back(primeFact(queries[i]));
 }
 return ans;
 }
};

int main(){
 vector<int> queries={2,3,4,5,6};
 Solution sol;

```

```

vector<vector<int>> ans=sol.primeFactors(queries);
for(int i=0;i<ans.size();i++){
 for(int x:ans[i]){
 cout<<x<<" ";
 }
 cout<<endl;
}
return 0;
}

```

## Complexity Analysis

Time Complexity:

$O(\max\_N \log \log \max\_N)$  for building the SPF array using sieve.

$O(N \log(\text{num}))$  for factorizing N numbers, where num is the value of a number.

Space Complexity:

$O(\max\_N)$  for the SPF array.

$O(N \log(\text{num}))$  for storing the prime factors of all queries.



## Dry run

**MAX\_N = 10**

**i = 2 (prime)**

```

SPF[2] = 2
SPF[4] = 2
SPF[6] = 2
SPF[8] = 2
SPF[10] = 2

```

---

**i = 3 (prime)**

```

SPF[3] = 3
SPF[6] already set (2) → skip
SPF[9] = 3

```

---

**i = 4**

SPF[4] != 1 → skip (composite)

---

**i = 5 (prime)**

SPF[5] = 5

SPF[10] already set → skip

Final SPF:

Index: 2 3 4 5 6 7 8 9 10

SPF: 2 3 2 5 2 7 2 3 2

---

◆ **Step 3: Prime factorization using SPF**

```
vector<int> primeFact(int n){
 vector<int> ans;
 while(n!=1){
 ans.push_back(SPF[n]);
 n = n / SPF[n];
 }
 return ans;
}
```

---



### **Example: n = 36**

36 → SPF[36] = 2 → n = 18  
18 → SPF[18] = 2 → n = 9  
9 → SPF[9] = 3 → n = 3  
3 → SPF[3] = 3 → n = 1

👉 Factors: 2 2 3 3

# 18. Power (n, x)

You are given a number x and an integer n.

You need to calculate x raised to the power n, written as  $x^n$ .

If n is negative, the result should be the reciprocal of x raised to |n|.

Example:

If x = 2 and n = 10, then  $2^{10} = 1024$ .

If x = 2 and n = -2, then  $2^{-2} = 1 / (2^2) = 0.25$ .

---

## Approach 1: Brute Force

### Algorithm

If n is 0, return 1 because any number raised to 0 is 1.

If n is negative, convert the problem by taking reciprocal of x and making n positive.

Initialize result as 1.

Multiply result with x exactly n times using a loop.

Return the final result.

### Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 double myPow(double x, int n){
 if(n==0) return 1;
 long long exp=n;
 if(exp<0){
 x=1/x;
 exp=-exp;
 }
 double ans=1;
 for(long long i=0;i<exp;i++){
 ans*=x;
 }
 return ans;
 }
}
```

```

};

int main(){
 Solution sol;
 cout<<sol.myPow(2.0,10)<<endl;
 cout<<sol.myPow(2.0,-2)<<endl;
 return 0;
}

```

### Complexity Analysis

Time Complexity: O(n), because multiplication is done n times.

Space Complexity: O(1), no extra space is used.

---

## Approach 2: Optimal Approach (Binary Exponentiation)

### Algorithm

Use recursion to reduce the number of multiplications.

If n is 0, return 1.

If n is negative, compute power for -n and return its reciprocal.

If n is even, compute power( $x * x$ ,  $n / 2$ ).

If n is odd, compute  $x * \text{power}(x, n - 1)$ .

This reduces the number of operations significantly.

### Code

```

#include <bits/stdc++.h>
using namespace std;

class Solution {
private:
 double power(double x, long n){
 if(n==0) return 1;
 if(n==1) return x;
 if(n%2==0){
 return power(x*x,n/2);
 }
 return x*power(x,n-1);
 }
public:

```

```

double myPow(double x, int n){
 long num=n;
 if(num<0){
 return 1.0/power(x,-num);
 }
 return power(x,num);
}

int main(){
 Solution sol;
 cout<<sol.myPow(2.0,10)<<endl;
 cout<<sol.myPow(2.0,-2)<<endl;
 return 0;
}

```

### **Complexity Analysis**

Time Complexity:  $O(\log n)$ , because the exponent is reduced at each step.

Space Complexity:  $O(\log n)$ , due to recursive call stack.

# **Stack and Queues**

# 1. Implement Stack using Array

You need to implement a stack data structure using an array.

A stack follows the **Last-In-First-Out (LIFO)** principle, meaning the last element added is the first one removed.

The stack must support these operations:

- `push(x)`: insert an element into the stack
- `pop()`: remove and return the top element
- `top()`: return the top element without removing it
- `isEmpty()`: check whether the stack is empty

Example:

If we push 5, then push 10, the top element becomes 10.

If we pop, 10 is removed.

After that, the stack is not empty because 5 is still present.

---

## Approach 1: Array Implementation

### Algorithm

Create an array with a fixed size to store stack elements.

Maintain an integer variable `topIndex` to track the index of the top element.

Initialize `topIndex` as -1 to indicate an empty stack.

Push operation:

- Check if the stack is full.
- Increase `topIndex` and insert the element at that index.

Pop operation:

- Check if the stack is empty.
- Return the element at `topIndex` and decrease `topIndex`.

Top operation:

- Check if the stack is empty.
- Return the element at topIndex without removing it.

isEmpty operation:

- If topIndex is -1, the stack is empty.

### Code

```
#include <bits/stdc++.h>
using namespace std;

class ArrayStack {
private:
 int* stackArray;
 int capacity;
 int topIndex;

public:
 ArrayStack(int size = 1000) {
 capacity = size;
 stackArray = new int[capacity];
 topIndex = -1;
 }

 ~ArrayStack() {
 delete[] stackArray;
 }

 void push(int x) {
 if (topIndex >= capacity - 1) {
 cout << "Stack overflow" << endl;
 return;
 }
 stackArray[++topIndex] = x;
 }
}
```

```

int pop() {
 if (isEmpty()) {
 cout << "Stack is empty" << endl;
 return -1;
 }
 return stackArray[topIndex--];
}

int top() {
 if (isEmpty()) {
 cout << "Stack is empty" << endl;
 return -1;
 }
 return stackArray[topIndex];
}

bool isEmpty() {
 return topIndex == -1;
}
};

int main() {
 ArrayStack stack;
 stack.push(5);
 stack.push(10);
 cout << stack.top() << " ";
 cout << stack.pop() << " ";
 cout << (stack.isEmpty() ? "true" : "false");
 return 0;
}

```

### Complexity Analysis

Time Complexity: O(1) for push, pop, top, and isEmpty operations.

Space Complexity: O(N), where N is the capacity of the stack array.

## 2. Implement Queue Using Array

You need to implement a queue using an array.

A queue follows the **First-In-First-Out (FIFO)** principle, meaning the element inserted first is removed first.

The queue must support these operations:

- `push(x)`: add an element to the end of the queue
- `pop()`: remove and return the front element
- `peek()`: return the front element without removing it
- `isEmpty()`: check whether the queue is empty

Example:

If we push 5, then push 10, the front element is 5.

If we pop, 5 is removed.

After that, the queue is not empty because 10 is still present.

---

### Approach 1: Circular Queue using Array

#### Algorithm

Create an array of fixed size to store queue elements.

Maintain four variables:

- `start`: index of the front element
- `end`: index of the last element
- `currSize`: current number of elements
- `maxSize`: maximum capacity of the queue

Push operation:

- Check if the queue is full using `currSize`.

- If the queue is empty, initialize start and end to 0.
- Otherwise, move end forward using circular indexing.
- Insert the element and increase currSize.

Pop operation:

- Check if the queue is empty.
- Store the element at start.
- If only one element exists, reset start and end to -1.
- Otherwise, move start forward using circular indexing.
- Decrease currSize and return the removed element.

Peek operation:

- Return the element at start if the queue is not empty.

isEmpty operation:

- If currSize is 0, the queue is empty.

## Code

```
#include <bits/stdc++.h>
using namespace std;

class ArrayQueue {
 int* arr;
 int start,end;
 int currSize,maxSize;

public:
 ArrayQueue() {
 maxSize=10;
 arr=new int[maxSize];
```

```

 start=-1;
 end=-1;
 currSize=0;
 }

void push(int x) {
 if(currSize==maxSize) {
 cout<<"Queue is full"<<endl;
 return;
 }
 if(end==-1) {
 start=0;
 end=0;
 } else {
 end=(end+1)%maxSize;
 }
 arr[end]=x;
 currSize++;
}

int pop() {
 if(isEmpty()) {
 cout<<"Queue is empty"<<endl;
 return -1;
 }
 int val=arr[start];
 if(currSize==1) {
 start=-1;
 end=-1;
 } else {
 start=(start+1)%maxSize;
 }
 currSize--;
 return val;
}

int peek() {
 if(isEmpty()) {

```

```

 cout<<"Queue is empty"<<endl;
 return -1;
 }
 return arr[start];
}

bool isEmpty() {
 return currSize==0;
}
};

int main() {
 ArrayQueue queue;
 queue.push(5);
 queue.push(10);
 cout<<queue.peek()<< " ";
 cout<<queue.pop()<< " ";
 cout<<(queue.isEmpty()?"true":"false");
 return 0;
}

```

### Complexity Analysis

Time Complexity: O(1) for push, pop, peek, and isEmpty operations.

Space Complexity: O(N), where N is the fixed size of the array used for the queue.

## 3. Implement Stack using single Queue

You need to implement a stack using only one queue.

A stack follows the **Last-In-First-Out (LIFO)** rule, while a queue follows **First-In-First-Out (FIFO)**.

The goal is to use queue operations in such a way that stack behavior is achieved.

The stack must support:

- `push(x)`: insert an element
- `pop()`: remove and return the top element
- `top()`: return the top element without removing it
- `isEmpty()`: check if the stack is empty

Example:

If we push 4, then push 8, the top element becomes 8.

If we pop, 8 is removed.

After that, the top element becomes 4.

---

## Approach 1: Using One Queue

### Algorithm

Use a single queue to store all elements.

For `push(x)`:

- First, insert  $x$  into the queue.
- Let the current size of the queue before insertion be  $s$ .
- Rotate the queue  $s$  times by removing the front element and pushing it to the back.
- This brings the newly added element to the front, making it act like the top of a stack.

For `pop()`:

- Remove and return the front element of the queue. This represents the stack's top.

For `top()`:

- Return the front element of the queue without removing it.

For isEmpty():

- If the queue is empty, return true, otherwise false.

### Code

```
#include <bits/stdc++.h>
using namespace std;

class QueueStack {
 queue<int> q;

public:
 void push(int x) {
 int s = q.size();
 q.push(x);
 for(int i = 0; i < s; i++) {
 q.push(q.front());
 q.pop();
 }
 }

 int pop() {
 int val = q.front();
 q.pop();
 return val;
 }

 int top() {
 return q.front();
 }

 bool isEmpty() {
 return q.empty();
 }
};

int main() {
 QueueStack st;
```

```

 st.push(4);
 st.push(8);
 cout << st.pop() << " ";
 cout << st.top() << " ";
 cout << (st.isEmpty() ? "true" : "false");
 return 0;
}

```

### **Complexity Analysis**

Time Complexity:

- `push()`: O(n), because existing elements are rotated.
- `pop()`: O(1)
- `top()`: O(1)
- `isEmpty()`: O(1)

Space Complexity: O(n), where n is the number of elements stored in the queue.

## **4. Implement Queue using Stack**

You need to implement a queue using stacks.

A queue follows the **First-In-First-Out (FIFO)** principle, but stacks follow **Last-In-First-Out (LIFO)**.

Using stack operations carefully, queue behavior can be achieved.

The queue must support:

- `push(x)`: add element to the end of the queue
- `pop()`: remove and return the front element

- `peek()`: return the front element without removing it
- `isEmpty()`: check whether the queue is empty

Example:

If we push 4, then push 8, the front element is 4.

If we pop, 4 is removed.

After that, peek returns 8.

---

## Approach 1: Using Two Stacks (Push is O(N))

### Algorithm

Maintain two stacks: st1 and st2.

Push operation:

- Move all elements from st1 to st2.
- Push the new element into st1.
- Move all elements back from st2 to st1.
- This ensures the front of the queue is always on top of st1.

Pop operation:

- Remove and return the top element of st1.

Peek operation:

- Return the top element of st1 without removing it.

`isEmpty` operation:

- Check if st1 is empty.

### Code

```
#include <bits/stdc++.h>
```

```

using namespace std;

class StackQueue {
 stack<int> st1, st2;

public:
 void push(int x) {
 while(!st1.empty()){
 st2.push(st1.top());
 st1.pop();
 }
 st1.push(x);
 while(!st2.empty()){
 st1.push(st2.top());
 st2.pop();
 }
 }

 int pop() {
 if(st1.empty()) return -1;
 int val = st1.top();
 st1.pop();
 return val;
 }

 int peek() {
 if(st1.empty()) return -1;
 return st1.top();
 }

 bool isEmpty() {
 return st1.empty();
 }
};

int main() {
 StackQueue q;
 q.push(4);

```

```

q.push(8);
cout << q.pop() << " ";
cout << q.peek() << " ";
cout << (q.isEmpty() ? "true" : "false");
return 0;
}

```

## Complexity Analysis

Time Complexity:

- push(): O(N)
- pop(): O(1)
- peek(): O(1)
- isEmpty(): O(1)

Space Complexity: O(N), for storing elements in stacks.

---

## Approach 2: Using Two Stacks (Push is O(1))

### Algorithm

Maintain two stacks: input and output.

Push operation:

- Simply push the element into input stack.

Pop operation:

- If output stack is empty, move all elements from input to output.
- Pop and return the top of output stack.

Peek operation:

- If output stack is empty, move all elements from input to output.

- Return the top of output stack.

isEmpty operation:

- Check if both stacks are empty.

### Code

```
#include <bits/stdc++.h>
using namespace std;

class StackQueue {
 stack<int> input, output;

public:
 void push(int x) {
 input.push(x);
 }

 int pop() {
 if(output.empty()){
 while(!input.empty()){
 output.push(input.top());
 input.pop();
 }
 }
 if(output.empty()) return -1;
 int val = output.top();
 output.pop();
 return val;
 }

 int peek() {
 if(output.empty()){
 while(!input.empty()){
 output.push(input.top());
 input.pop();
 }
 }
 }
}
```

```

 }
 if(output.empty()) return -1;
 return output.top();
 }

bool isEmpty() {
 return input.empty() && output.empty();
}
};

int main() {
 StackQueue q;
 q.push(4);
 q.push(8);
 cout << q.pop() << " ";
 cout << q.peek() << " ";
 cout << (q.isEmpty() ? "true" : "false");
 return 0;
}

```

## Complexity Analysis

Time Complexity:

- push(): O(1)
- pop(): O(N) in worst case, O(1) amortized
- peek(): O(N) in worst case, O(1) amortized
- isEmpty(): O(1)

Space Complexity: O(N), for storing elements in two stacks.

# 5. Implement Stack using Linked List

You need to implement a stack using a singly linked list.

A stack follows the **Last-In-First-Out (LIFO)** principle, where the last inserted element is removed first.

The stack must support:

- `push(x)`: insert an element on top of the stack
- `pop()`: remove and return the top element
- `top()`: return the top element without removing it
- `isEmpty()`: check whether the stack is empty

Example:

If we push 3, then push 7, the top becomes 7.

If we pop, 7 is removed.

After that, the top becomes 3.

---

## Approach 1: Using Singly Linked List

### Algorithm

Use a singly linked list where the head represents the top of the stack.

Push operation:

- Create a new node with value x.
- Point the new node's next to the current head.
- Update head to the new node.

Pop operation:

- If the stack is empty, return -1.
- Store the value of head.
- Move head to head->next.
- Delete the old head and return the stored value.

Top operation:

- If the stack is empty, return -1.
- Return the value stored at head.

isEmpty operation:

- If head is NULL, the stack is empty.

### Code

```
#include <bits/stdc++.h>
using namespace std;
```

```
struct Node {
 int val;
 Node* next;
 Node(int x) {
 val = x;
 next = NULL;
 }
};
```

```
class LinkedListStack {
 Node* head;
 int size;

public:
 LinkedListStack() {
```

```

 head = NULL;
 size = 0;
 }

void push(int x) {
 Node* node = new Node(x);
 node->next = head;
 head = node;
 size++;
}

int pop() {
 if (head == NULL) return -1;
 int val = head->val;
 Node* temp = head;
 head = head->next;
 delete temp;
 size--;
 return val;
}

int top() {
 if (head == NULL) return -1;
 return head->val;
}

bool isEmpty() {
 return size == 0;
}

int main() {
 LinkedListStack st;
 st.push(3);
 st.push(7);
 cout << st.pop() << " ";
 cout << st.top() << " ";
 cout << (st.isEmpty() ? "true" : "false");
}

```

```
 return 0;
}
```

### Complexity Analysis

Time Complexity: O(1) for push, pop, top, and isEmpty operations.

Space Complexity: O(N), where N is the number of elements stored in the stack.

## 6. Implement Queue using Linked List

You need to implement a queue using a singly linked list.

A queue follows the **First-In-First-Out (FIFO)** principle, meaning the element inserted first is removed first.

The queue must support:

- `push(x)`: insert an element at the end of the queue
- `pop()`: remove and return the front element
- `peek()`: return the front element without removing it
- `isEmpty()`: check whether the queue is empty

Example:

If we push 3, then push 7, the front element is 3.

If we pop, 3 is removed.

After that, the queue is not empty because 7 is still present.

---

### Approach 1: Using Singly Linked List

#### Algorithm

Use a singly linked list with two pointers:

- `start` points to the front of the queue

- end points to the rear of the queue

Push operation:

- Create a new node with value x.
- If the queue is empty, set both start and end to the new node.
- Otherwise, link the current end node to the new node and update end.

Pop operation:

- If the queue is empty, return -1.
- Store the value at start.
- Move start to start->next.
- Delete the old front node.
- If the queue becomes empty, end will naturally become NULL.

Peek operation:

- If the queue is empty, return -1.
- Return the value at start.

isEmpty operation:

- If start is NULL (or size is 0), the queue is empty.

## Code

```
#include <bits/stdc++.h>
using namespace std;

struct Node {
 int val;
 Node* next;
```

```

Node(int x) {
 val = x;
 next = NULL;
}
};

class LinkedListQueue {
private:
 Node* start;
 Node* end;
 int size;

public:
 LinkedListQueue() {
 start = end = NULL;
 size = 0;
 }

 void push(int x) {
 Node* node = new Node(x);
 if(start == NULL) {
 start = end = node;
 } else {
 end->next = node;
 end = node;
 }
 size++;
 }

 int pop() {
 if(start == NULL) return -1;
 int val = start->val;
 Node* temp = start;
 start = start->next;
 delete temp;
 size--;
 return val;
 }
}

```

```

int peek() {
 if(start == NULL) return -1;
 return start->val;
}

bool isEmpty() {
 return size == 0;
}

int main() {
 LinkedListQueue q;
 q.push(3);
 q.push(7);
 cout << q.peek() << " ";
 cout << q.pop() << " ";
 cout << (q.isEmpty() ? "true" : "false");
 return 0;
}

```

### Complexity Analysis

Time Complexity: O(1) for push, pop, peek, and isEmpty operations.

Space Complexity: O(N), where N is the number of elements stored in the queue.

## 7. Check for Balanced Parentheses

Given a string containing only the characters (, ), {, }, [ and ], check whether the string is balanced.

A string is balanced if every opening bracket has a matching closing bracket of the same type and the brackets are closed in the correct order.

For example, "()[{}()]" is balanced because every opening bracket is correctly closed.

The string "[(" is not balanced because [ does not have a matching ].

---

## Approach 1: Using Stack

### Algorithm

We need to track the most recent opening bracket and match it with the current closing bracket. A stack follows Last-In-First-Out behavior, which perfectly fits this requirement.

Traverse the string character by character.

If the character is an opening bracket (, {, or [, push it onto the stack.

If the character is a closing bracket:

- If the stack is empty, there is no matching opening bracket, so return false.
- Otherwise, check the top of the stack. If it is not the matching opening bracket, return false.
- If it matches, pop the opening bracket and continue.

After processing the entire string, the stack must be empty.

If it is empty, all brackets were matched correctly, so return true. Otherwise, return false.

### Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 bool isValid(string s) {
 stack<char> st;
 for(char ch : s) {
 if(ch == '(' || ch == '{' || ch == '[') {
 st.push(ch);
 } else {
 if(st.empty()) return false;
 char top = st.top();
 st.pop();
 if((ch == ')' && top == '(') ||
 (ch == ']' && top == '[') ||
 (ch == '}' && top == '{')) {

```

```

 continue;
 } else {
 return false;
 }
}
return st.empty();
}

int main() {
 Solution sol;
 string s = "()[{}()]";
 cout << (sol.isValid(s) ? "True" : "False");
 return 0;
}

```

### **Complexity Analysis**

Time Complexity:  $O(N)$ , where  $N$  is the length of the string, as each character is processed once.

Space Complexity:  $O(N)$ , due to the stack storing opening brackets in the worst case.

## **8. Implement Min Stack : $O(2N)$ and $O(N)$ Space Complexity**

You need to design a stack that supports normal stack operations along with retrieving the minimum element, all in constant time.

The stack must support:

- `push(val)`: insert an element

- `pop()`: remove the top element
- `top()`: return the top element
- `getMin()`: return the minimum element present in the stack

Example:

If we push -2, 0, -3 into the stack, the minimum is -3.

After popping -3, the top becomes 0 and the minimum becomes -2.

---

## Approach 1: Brute Force (Using Stack of Pairs)

In this approach, each element in the stack stores:

- the actual value
- the minimum value up to that point

So for every element, we always know the minimum till that element.

### Algorithm

Use a stack of pairs (`value, currentMinimum`).

Push operation:

- If the stack is empty, push (`value, value`).
- Otherwise, compare `value` with the current minimum.
- Push (`value, min(value, currentMinimum)`).

Pop operation:

- Remove the top pair from the stack.

Top operation:

- Return the first element of the top pair.

GetMin operation:

- Return the second element of the top pair.

### Code

```
#include <bits/stdc++.h>
using namespace std;

class MinStack {
private:
 stack<pair<int,int>> st;

public:
 MinStack() {}

 void push(int value) {
 if(st.empty()) {
 st.push({value,value});
 } else {
 int mini = min(value, st.top().second);
 st.push({value, mini});
 }
 }

 void pop() {
 if(!st.empty()) st.pop();
 }

 int top() {
 if(st.empty()) return -1;
 return st.top().first;
 }

 int getMin() {
 if(st.empty()) return -1;
 return st.top().second;
 }
};
```

```

int main() {
 MinStack s;
 s.push(-2);
 s.push(0);
 s.push(-3);
 cout << s.getMin() << " ";
 s.pop();
 cout << s.top() << " ";
 cout << s.getMin();
 return 0;
}

```

### **Complexity Analysis**

Time Complexity: O(1) for push, pop, top, and getMin.

Space Complexity:  $O(2N) \approx O(N)$ , because each element stores an extra minimum value.

---

## **Approach 2: Optimal Approach (Single Stack with Encoding)**

In this approach, only one stack is used along with a variable to track the current minimum.

The idea is to encode values smaller than the current minimum before pushing them.

### **Algorithm**

Maintain:

- one stack
- one variable `mini` to store the current minimum

Push operation:

- If the stack is empty, push value and set `mini = value`.
- If `value > mini`, push value normally.
- If `value <= mini`, push a modified value ( $2*value - mini$ ) and update `mini` to `value`.

Pop operation:

- If the top value is  $\geq$  mini, pop normally.
- If the top value  $<$  mini, it means it was encoded.
  - Update  $\text{mini} = 2 * \text{mini} - \text{top}$ .

Top operation:

- If  $\text{top} \geq \text{mini}$ , return top.
- Otherwise, return mini.

GetMin operation:

- Return mini.

Main logic:

**Encoded value hamesha current mini se chhoti hoti hai**

Proof:

$$\text{encoded} = 2 * \text{value} - \text{mini}$$

Since  $\text{value} < \text{mini}$

$$\Rightarrow 2 * \text{value} - \text{mini} < \text{value}$$

$$\Rightarrow \text{encoded} < \text{value} < \text{mini}$$

👉 Matlab:

- Stack me koi element **mini se chhota** dikhe  
 $\Rightarrow$  wo **real value nahi**, encoded value hai

## Code

```
#include <bits/stdc++.h>
using namespace std;
```

```

class MinStack {
private:
 stack<int> st;
 int mini;

public:
 MinStack() {}

 void push(int value) {
 if(st.empty()) {
 mini = value;
 st.push(value);
 } else if(value > mini) {
 st.push(value);
 } else {
 st.push(2*value - mini);
 mini = value;
 }
 }

 void pop() {
 if(st.empty()) return;
 int x = st.top();
 st.pop();
 if(x < mini) {
 mini = 2*mini - x;
 }
 }

 int top() {
 if(st.empty()) return -1;
 int x = st.top();
 if(x >= mini) return x;
 return mini;
 }

 int getMin() {

```

```

 return mini;
 }
};

int main() {
 MinStack s;
 s.push(-2);
 s.push(0);
 s.push(-3);
 cout << s.getMin() << " ";
 s.pop();
 cout << s.top() << " ";
 cout << s.getMin();
 return 0;
}

```

### Complexity Analysis

Time Complexity: O(1) for push, pop, top, and getMin.

Space Complexity: O(N), as only one stack is used without storing extra pairs.

## Dry Run

### Push sequence:

```

push(5)
push(3)
push(2)

```

---

### Step 1: push(5)

- Stack empty
- push 5
- mini = 5

Stack: [ 5 ]

mini = 5

---

### Step 2: push(3)

$3 \leq \text{mini}(5)$

encoded =  $2*3 - 5 = 1$

- push 1
- mini = 3

Stack: [ 5, 1 ]

mini = 3

👉 1 actual value nahi hai, sirf **marker** hai

---

### Step 3: push(2)

$2 \leq \text{mini}(3)$

encoded =  $2*2 - 3 = 1$

- push 1
- mini = 2

Stack: [ 5, 1, 1 ]

mini = 2

---

🔥 Ab POP

Pop top = 1

- $1 < \text{mini}(2) \Rightarrow$  encoded value hai

👉 Old mini recover karo:

$$\begin{aligned}\text{oldMini} &= 2 * \text{mini} - \text{encoded} \\ &= 2 * 2 - 1 \\ &= 3\end{aligned}$$

- $\text{mini} = 3$

Stack: [ 5, 1 ]

---

### Pop next = 1

- $1 < \text{mini}(3) \Rightarrow$  encoded

$$\text{oldMini} = 2 * 3 - 1 = 5$$

- $\text{mini} = 5$

Stack: [ 5 ]

---

### Pop 5

- Normal pop
- Stack empty

---

## Iska matlab kya hua?

**Push time:**

```
encoded = 2*newMin - oldMin
```

### Pop time:

```
oldMin = 2*currentMin - encoded
```

👉 Ye **reversible encoding** hai

👉 Isliye ek hi stack kaam kar leta hai

---

*We encode values smaller than the current minimum so that we can recover the previous minimum during pop.*

## 9. Infix to Postfix

You are given an infix expression.

Your task is to convert this infix expression into a postfix expression.

In an **infix expression**, the operator comes between operands, for example a + b.

In a **postfix expression**, the operator comes after operands, for example ab+.

Postfix expressions are easier for computers to evaluate because they do not need parentheses or precedence rules during evaluation.

---

### Question Explanation

Given an infix expression containing operands (letters or digits), operators (+ - \* / ^), and parentheses, convert it into a postfix expression by following operator precedence and associativity rules.

Operator precedence (high to low):

- ^
- \* /

- + -

Associativity:

- $\wedge$  is **right associative**
  - \* / + - are **left associative**
- 

## Example Explanation

Input:

$$(p + q) * (m - n)$$

Step-by-step:

- Convert  $(p + q) \rightarrow pq+$
- Convert  $(m - n) \rightarrow mn-$
- Apply  $* \rightarrow pq+mn-*$

Output:

$$pq+mn-*$$

---

## Approach 1

### Algorithm

1. Create an empty stack to store operators.
2. Traverse the infix expression from left to right.
3. If the current character is an operand, add it to the result.
4. If the character is '(', push it onto the stack.

5. If the character is ')' , pop from the stack and add to result until '(' is found. Remove '('.
  6. If the character is an operator:
    - o Pop operators from the stack while:
      - the stack is not empty, and
      - the precedence of the current operator is **less than** the precedence of the stack top, or
      - the precedence is **equal and the operator is left associative**
    - o Push the current operator onto the stack.
  7. After processing the entire expression, pop all remaining operators from the stack and add them to the result.
  8. The final result is the postfix expression.
- 

## Code

```
#include <bits/stdc++.h>
using namespace std;

// Function to return precedence of operators
int prec(char c) {
 if (c == '^') return 3;
 if (c == '*' || c == '/') return 2;
 if (c == '+' || c == '-') return 1;
 return -1;
}

// Function to check right associativity
bool isRightAssociative(char c) {
 return c == '^';
}
```

```

// Function to convert infix to postfix
string infixToPostfix(string s) {
 stack<char> st;
 string result = "";

 for (int i = 0; i < s.length(); i++) {
 char c = s[i];

 // Operand
 if ((c >= 'a' && c <= 'z') ||
 (c >= 'A' && c <= 'Z') ||
 (c >= '0' && c <= '9')) {
 result += c;
 }
 // Opening bracket
 else if (c == '(') {
 st.push(c);
 }
 // Closing bracket
 else if (c == ')') {
 while (!st.empty() && st.top() != '(') {
 result += st.top();
 st.pop();
 }
 st.pop(); // remove '('
 }
 // Operator
 else {
 while (!st.empty() &&
 (prec(c) < prec(st.top()) ||
 (prec(c) == prec(st.top()) &&
 !isRightAssociative(c)))) {
 result += st.top();
 st.pop();
 }
 st.push(c);
 }
 }
}

```

```

 // Pop remaining operators
 while (!st.empty()) {
 result += st.top();
 st.pop();
 }

 return result;
}

int main() {
 string exp = "(p+q)*(m-n)";
 cout << infixToPostfix(exp);
 return 0;
}

```

---

## Complexity Analysis

### Time Complexity:

$O(N)$ , where  $N$  is the length of the infix expression. Each character is processed once.

### Space Complexity:

$O(N)$ , due to the stack used to store operators and parentheses.

- ◆ **Full dry run**

### Input:

$(p+q)*(m-n)$

### Step-by-step:

| Char | Stac | Result |
|------|------|--------|
| k    |      |        |

|   |   |  |
|---|---|--|
| ( | ( |  |
|---|---|--|

|     |       |         |
|-----|-------|---------|
| p   | (     | p       |
| +   | ( +   | p       |
| q   | ( +   | pq      |
| )   |       | pq+     |
| *   | *     | pq+     |
| (   | * (   | pq+     |
| m   | * (   | pq+m    |
| -   | * ( - | pq+m    |
| n   | * ( - | pq+mn   |
| )   | *     | pq+mn-  |
| end |       | pq+mn-* |

Output:

pq+mn-\*

---

## One-line intuition (interview ready)

*Operators are popped from the stack if they have higher precedence, or equal precedence with left associativity.*

---

- Operand → output
- ( → push
- ) → pop till (
- Operator:
  - pop **stronger** operators

- pop **equal & left associative**
- then push current

## 🔴 CORE RULE

**Postfix me operator tab likhte hain jab usse zyada important kaam ho chuka ho.**

Stack bas **operators ko rok ke rakhne** ke liye hai.

---

### Step 0: Sirf 3 cheezein hoti hain

Expression me aayega:

1. **Operand** (a, b, 1, 2...)
  2. **Operator** (+ - \* / ^)
  3. **Bracket** ( )
- 

## 🟡 Rule 1: Operand aaye

a, b, x, 5

### 👉 Seedha output me daal do

❓ Kyun?

Operand ko kisi ka wait nahi hota.

---

## 🟢 Rule 2: '(' aaye

### 👉 Stack me daal do

Bracket ka matlab:

“Iske andar ka kaam pehle complete karo”

---

### ● Rule 3: ')' aaye

- 👉 Stack se pop karte jao
  - 👉 Jab tak ')' na mil jaye
  - 👉 ')' ko output me nahi daalte
- 

### ● Rule 4 : Operator aaye

Ab dhyaan se 👈

Jab operator c aaye:

- Stack ke top ko dekho
- Agar stack top:
  - **zyada powerful** hai
  - ya **same power ka hai aur left associative** hai  
👉 to pehle usko output me daal do

Fir:

👉 current operator ko stack me push karo

### Dry RUN 2

| Infix expression              | Postfix expression | F        |
|-------------------------------|--------------------|----------|
| $(A + B)^*(C - D)$            | AB + CD - *        |          |
| $(A + B) / (C - D) - (E * F)$ | AB + CD - / EF * - | 2ND WALA |

| <b>Symbol</b> | <b>Action</b>                                                | <b>Stack</b> | <b>Output</b> |
|---------------|--------------------------------------------------------------|--------------|---------------|
| (             | push                                                         | (            |               |
| A             | operand → output                                             | ( A          |               |
| +             | push (top is ( )                                             | ( +          | A             |
| B             | operand                                                      | ( +          | AB            |
| )             | pop till (                                                   |              | AB+           |
| /             | push                                                         | /            | AB+           |
| (             | push                                                         | / (          | AB+           |
| C             | operand                                                      | / (          | AB+C          |
| -             | push                                                         | / ( -        | AB+C          |
| D             | operand                                                      | / ( -        | AB+CD         |
| )             | pop till (                                                   | /            | AB+CD-        |
| -             | compare with / → / has higher precedence → pop<br>now push - |              | AB+CD-/       |
| (             | push                                                         | - (          | AB+CD-/       |
| E             | operand                                                      | - (          | AB+CD-/E      |
| *             | push                                                         | - ( *        | AB+CD-/E      |
| F             | operand                                                      | - ( *        | AB+CD-/EF     |
| )             | pop till (                                                   | -            | AB+CD-/EF     |
|               |                                                              | *            |               |

|     |                         |                 |
|-----|-------------------------|-----------------|
| end | pop remaining operators | AB+CD-/EF<br>*_ |
|-----|-------------------------|-----------------|

## 10. Prefix to Infix Conversion

You are given a valid arithmetic expression written in **prefix notation**.

Your task is to convert it into a **fully parenthesized infix expression**.

In prefix notation, the operator comes **before** its operands.

In infix notation, the operator comes **between** operands and parentheses are used to preserve order.

---

### Question Explanation

In prefix expressions:

- Operators appear before operands.
- Example:  $+ab$  means  $a + b$ .

To convert prefix to infix:

- We must rebuild the expression so that every operation is clearly grouped using parentheses.
  - The final infix expression must be **fully parenthesized** to avoid ambiguity.
- 

### Example Explanation

Input:

\*+ab-cd

Step-by-step:

- +ab becomes (a+b)
- -cd becomes (c-d)
- \* combines both → ((a+b)\*(c-d))

Output:

((a+b)\*(c-d))

---

## Approach 1

### Algorithm

1. Traverse the prefix expression from **right to left**.
2. Use a stack to store operands or partial infix expressions.
3. If the current character is an operand:
  - Push it as a string onto the stack.
4. If the current character is an operator:
  - Pop the top two elements from the stack.
  - Combine them into a new string:  
(operand1 operator operand2)

- Push this new string back onto the stack.
5. After the traversal ends, the stack will contain only one element.
  6. That element is the required fully parenthesized infix expression.
- 

## Code

```
#include <bits/stdc++.h>
using namespace std;

// Function to convert prefix expression to infix
string prefixToInfix(string prefix) {
 stack<string> st;
 int n = prefix.size();

 // Traverse from right to left
 for(int i = n - 1; i >= 0; i--) {
 char c = prefix[i];

 // If operand, push to stack
 if(isalnum(c)) {
 st.push(string(1, c));
 //string(count, character), as direct char nhi de sakte string ko
 //Matlab: count baar character ko repeat karke string banao
 }

 //isalnum() is a programming function (in Python, C, C++) that checks if all characters in a string or a given
 //character are alphanumeric (letters 'a'-'z', 'A'-'Z', or numbers '0'-'9')
 // If operator
 else {
 string op1 = st.top();
 st.pop();
 string op2 = st.top();
```

```

 st.pop();

 string expr = "(" + op1 + c + op2 + ")";
 st.push(expr);
 }

}

return st.top();
}

int main() {
 string expression = "*+ab-cd";
 cout << prefixToInfix(expression);
 return 0;
}

```

---

## Complexity Analysis

### Time Complexity:

$O(N)$ , where  $N$  is the length of the prefix expression. Each character is processed once.

### Space Complexity:

$O(N)$ , due to the stack storing intermediate expressions.

# 11. Prefix to Postfix Conversion

You are given a valid arithmetic expression in **prefix notation**.  
Your task is to convert it into an equivalent **postfix expression**.

- **Prefix (Polish) notation:** Operator comes **before** operands  
Example: +ab
  - **Postfix (Reverse Polish) notation:** Operator comes **after** operands  
Example: ab+
- 

## Key Idea

Prefix expressions are best handled by scanning **from right to left**.  
A **stack** helps us temporarily store operands and intermediate postfix expressions.

---

## Algorithm

1. Traverse the prefix expression **from right to left**.
2. Use a stack of strings.
3. For each character:
  - If it is an **operand**, push it onto the stack.
  - If it is an **operator**:
    - Pop two operands from the stack.
    - Combine them as:  
 $\text{operand1} + \text{operand2} + \text{operator}$

- Push the combined string back onto the stack.
4. After traversal, the stack's top element is the final postfix expression.
- 

## ✨ Example Walkthrough

### Example 1

**Input:** +ab

Steps:

- Push b
- Push a
- Pop a, b, combine → ab+

**Output:** ab+

---

### Example 2

**Input:** \*+ab-cd

Breakdown:

- +ab → ab+
- -cd → cd-
- Combine with \* → ab+cd-\*

**Output:** ab+cd-\*

---



## C++ Implementation

```
#include <bits/stdc++.h>
using namespace std;

// Function to convert prefix expression to postfix
string prefixToPostfix(string prefix) {
 stack<string> st;
 int n = prefix.size();

 // Traverse from right to left
 for (int i = n - 1; i >= 0; i--) {
 char c = prefix[i];

 // If operand, push to stack
 if (isalnum(c)) {
 st.push(string(1, c));
 }
 // If operator
 else {
 string op1 = st.top(); st.pop();
 string op2 = st.top(); st.pop();

 // Postfix = operand1 + operand2 + operator
 st.push(op1 + op2 + c);
 }
 }

 return st.top();
}
```

```
int main() {
 string prefix = "*+ab-cd";
 cout << "Postfix Expression: " <<
prefixToPostfix(prefix) << endl;
 return 0;
}
```

---



## Complexity Analysis

- **Time Complexity:**  $O(n)$   
(Each character is processed once)
- **Space Complexity:**  $O(n)$   
(Stack stores intermediate expressions)

# 12. Postfix to Prefix Conversion

You are given a **valid postfix expression**.

In postfix notation, operators come **after** operands (example: ab+).

Your task is to convert this postfix expression into a **prefix expression**, where operators come **before** operands (example: +ab).

The expression has:

- Single character operands (letters or digits)
- Binary operators (+ - \* /)
- No spaces

- Guaranteed to be valid

## Example Explanation

Example:

Postfix expression = ab+

- a → operand → push to stack
- b → operand → push to stack
- + → operator
  - pop b (op2)
  - pop a (op1)
  - form prefix: +ab
  - push +ab back to stack

Final stack top = +ab (answer)

---

## Approach 1

### Algorithm

- Create a stack of strings.
- Traverse the postfix expression **from left to right**.
- If the character is an operand, push it as a string into the stack.

- If the character is an operator:
  - Pop two elements from the stack.
  - First popped → operand2
  - Second popped → operand1
  - Form prefix expression: operator + operand1 + operand2
  - Push this string back into the stack.
- After traversal, the stack's top element is the prefix expression.

## Code

```
#include <bits/stdc++.h>
using namespace std;

string postfixToPrefix(string postfix) {
 stack<string> st;
 for(char c : postfix) {
 if(isalnum(c)) {
 st.push(string(1,c));
 } else {
 string op2 = st.top(); st.pop();
 string op1 = st.top(); st.pop();
 st.push(c + op1 + op2);
 }
 }
 return st.top();
}

int main() {
```

```
string postfix = "abc*+d-";
cout << postfixToPrefix(postfix);
return 0;
}
```

## Complexity Analysis

- Time Complexity: **O(n)**  
Each character is processed once.
- Space Complexity: **O(n)**  
Stack stores intermediate expressions.

# 13. Postfix to Infix

You are given a **postfix expression** as a string.

In postfix notation, operators come **after** their operands (example: ab+).

Your task is to convert this postfix expression into an **equivalent infix expression** with **proper parentheses**, so that the evaluation order is preserved correctly.

The postfix expression:

- Is valid
- Contains single-character operands (letters or digits)
- Contains binary operators (+ - \* /)
- Has no spaces

## Example Explanation

Example 1:

Postfix = ab+c\*

Step-by-step:

- a → operand → push
- b → operand → push
- + → operator
  - pop b, pop a
  - form ( a+b ) → push
- c → operand → push
- \* → operator
  - pop c, pop ( a+b )
  - form ( ( a+b ) \* c )

Final result: ( a+b ) \* c

---

## Approach 1

### Algorithm

- Traverse the postfix expression from **left to right**.

- Use a stack of strings.
- If the character is an operand, push it as a string.
- If the character is an operator:
  - Pop the top two elements from the stack.
  - First popped → operand2
  - Second popped → operand1
  - Create a new string: (operand1 operator operand2)
  - Push this string back into the stack.
- After traversal, the stack's top element is the infix expression.

### Code

```
#include <bits/stdc++.h>
using namespace std;

string postfixToInfix(string postfix) {
 stack<string> st;
 for(char c : postfix) {
 if(isalnum(c)) {
 st.push(string(1,c));
 } else {
 string op2 = st.top(); st.pop();
 string op1 = st.top(); st.pop();
 st.push("(" + op1 + c + op2 + ")");
 }
 }
 return st.top();
}
```

```

 }

int main() {
 string postfix = "ab+c*";
 cout << postfixToInfix(postfix);
 return 0;
}

```

## Complexity Analysis

- Time Complexity: **O(n)**  
Each character of the postfix expression is processed once.
- Space Complexity: **O(n)**  
Stack is used to store intermediate expressions.

# 14. Infix to Prefix

You are given an **infix expression**, where operators appear between operands (for example  $a+b$ ).

Your task is to convert this infix expression into an equivalent **prefix expression**, where operators appear **before** their operands.

Prefix expressions remove the need for parentheses and make evaluation easier for computers.

---

## Example Explanation

Example 1:

Input:  $x + y * z / w + u$

- Infix has operator precedence (\* and / before +)
- After applying infix to prefix conversion rules, the correct prefix form is:  
++x/\*yzwu

Example 2:

Input: a + b

Output: +ab

---

## Approach 1

### Algorithm

- Reverse the infix expression.
- Replace ( with ) and ) with (.
- Convert the modified infix expression to **postfix** using operator precedence rules.
- Reverse the postfix expression to get the **prefix** expression.
- Use a stack to handle operators during conversion.

### Code

```
#include <bits/stdc++.h>
using namespace std;

bool isOperator(char c) {
 return (!isalpha(c) && !isdigit(c));
}
```

```

int getPriority(char c) {
 if(c=='+' || c=='-') return 1;
 if(c=='*' || c=='/') return 2;
 if(c=='^') return 3;
 return 0;
}

string infixToPostfix(string infix) {
 infix = "(" + infix + ")";
 stack<char> st;
 string output;

 for(char c : infix) {
 if(isalnum(c)) {
 output += c;
 } else if(c=='(') {
 st.push(c);
 } else if(c==')') {
 while(st.top()!='(') {
 output += st.top();
 st.pop();
 }
 st.pop();
 } else {
 while(getPriority(c) < getPriority(st.top())) {
 output += st.top();
 st.pop();
 }
 st.push(c);
 }
 }
 return output;
}

```

```

}

string infixToPrefix(string infix) {
 reverse(infix.begin(), infix.end());
 for(char &c : infix) {
 if(c=='(') c=')';
 else if(c==')') c='(';
 }
 string prefix = infixToPostfix(infix);
 reverse(prefix.begin(), prefix.end());
 return prefix;
}

int main() {
 string s = "x+y*z/w+u";
 cout << infixToPrefix(s);
 return 0;
}

```

## Complexity Analysis

- **Time Complexity:**  $O(N)$   
Each character is processed a constant number of times.
- **Space Complexity:**  $O(N)$   
Stack space used for operators during conversion.

# 15. Next Greater Element Using Stack

You are given an integer array.

For every element in the array, you have to find the **next greater element on its right side**.

The next greater element of a number  $x$  is the first number greater than  $x$  that appears **after it** while moving from left to right.

If no such element exists, return  $-1$  for that position.

---

## Example Explanation

Example 1:

Input: [1, 3, 2, 4]

- For 1, the next greater element is 3
- For 3, the next greater element is 4
- For 2, the next greater element is 4
- For 4, no greater element exists  $\rightarrow -1$

Output: [3, 4, 4, -1]

Example 2:

Input: [6, 8, 0, 1, 3]

- 6  $\rightarrow$  8
- 8  $\rightarrow$  -1
- 0  $\rightarrow$  1

- $1 \rightarrow 3$
- $3 \rightarrow -1$

Output: [8, -1, 1, 3, -1]

---

## Approach 1

### Algorithm

- Use a stack to keep elements in decreasing order.
- Traverse the array from **right to left**.
- For each element:
  - Remove all elements from the stack that are **smaller than or equal** to the current element.
  - If the stack becomes empty, no greater element exists  $\rightarrow$  store -1.
  - Otherwise, the top of the stack is the next greater element.
- Push the current element into the stack.
- Continue until all elements are processed.

### Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
```

```

public:
 vector<int> nextGreater(vector<int>& nums) {
 stack<int> st;
 int n = nums.size();
 vector<int> res(n);

 for(int i = n - 1; i >= 0; i--) {
 while(!st.empty() && st.top() <= nums[i]) {
 st.pop();
 }

 if(st.empty()) res[i] = -1;
 else res[i] = st.top();

 st.push(nums[i]);
 }
 return res;
 }
};

int main() {
 vector<int> nums = {1, 3, 2, 4};
 Solution sol;
 vector<int> ans = sol.nextGreater(nums);

 for(int x : ans) {
 cout << x << " ";
 }
 return 0;
}

```

## Complexity Analysis

- **Time Complexity:** O(N)  
Each element is pushed and popped from the stack at most once.
  - **Space Complexity:** O(N)  
Stack and result array both use linear space.
- 

## 16. Next Greater Element – 2

You are given a **circular integer array**.

For every element, you have to find the **next greater element** while moving in a **clockwise (circular) manner**.

The next greater element of a number  $x$  is the **first number greater than  $x$**  that appears when we move forward in the array.

If we reach the end, we continue from the beginning because the array is circular.

If no such element exists, return -1 for that position.

---

### Example Explanation

Example 1:

Input: [5, 7, 1, 7, 6, 0]

- For 5 → next greater is 7
- For 7 → no greater element exists → -1

- For 1 → next greater is 7
- For 7 → no greater element exists → -1
- For 6 → next greater is 7
- For 0 → next greater is 5 (found after circular traversal)

Output: [7, -1, 7, -1, 7, 5]

---

## Approach 1 (Brute Force)

### Algorithm

- Create an answer array of size n, initialized with -1.
- For each element at index i:
  - Traverse the next  $n-1$  elements in circular order using  $(i + j) \% n$ .
  - If a greater element is found, store it in  $ans[i]$  and stop searching.
- After checking all elements, return the answer array.

### Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
```

```

vector<int> nextGreaterElements(vector<int> arr) {
 int n = arr.size();
 vector<int> ans(n, -1);

 for(int i = 0; i < n; i++) {
 for(int j = 1; j < n; j++) {
 int ind = (i + j) % n;
 if(arr[ind] > arr[i]) {
 ans[i] = arr[ind];
 break;
 }
 }
 }
 return ans;
}

int main() {
 vector<int> arr = {5, 7, 1, 7, 6, 0};
 Solution sol;
 vector<int> ans = sol.nextGreaterElements(arr);

 for(int x : ans) cout << x << " ";
 return 0;
}

```

## Complexity Analysis

- **Time Complexity:**  $O(N^2)$   
For each element, we may scan the entire array.

- **Space Complexity:** O(N)  
Used for the result array.
- 

## Approach 2 (Optimal Using Stack)

### Algorithm

- Initialize an answer array with -1.
- Use a stack to maintain elements in decreasing order.
  - Traverse the array from index  $2*n - 1$  to 0 to simulate circular traversal.
- Use  $i \% n$  to get the actual index.
- While stack is not empty and top  $\leq$  current element, pop from stack.
- If  $i < n$ :
  - If stack is empty  $\rightarrow$  answer is -1
  - Else  $\rightarrow$  top of stack is the next greater element
- Push current element into the stack.
- Return the answer array.

### Code

```
#include <bits/stdc++.h>
using namespace std;
```

```

class Solution {
public:
 vector<int> nextGreaterElements(vector<int> arr) {
 int n = arr.size();
 vector<int> ans(n, -1);
 stack<int> st;

 for(int i = 2*n - 1; i >= 0; i--) {
 int ind = i % n;

 while(!st.empty() && st.top() <= arr[ind]) {
 st.pop();
 }

 if(i < n) {
 if(!st.empty()) ans[ind] = st.top();
 }

 st.push(arr[ind]);
 }
 return ans;
 }
};

int main() {
 vector<int> arr = {5, 7, 1, 7, 6, 0};
 Solution sol;
 vector<int> ans = sol.nextGreaterElements(arr);

 for(int x : ans) cout << x << " ";
 return 0;
}

```

## Complexity Analysis

- **Time Complexity:** O(N)

Each element is pushed and popped from the stack at most once.

- **Space Complexity:** O(N)

Stack and answer array both take linear space.

## Dry Run

arr = [1, 2, 1]

n = 3

Loop: i = 5 → 0

| i | idx | arr[idx] | Stack before | Action        | Stack after | ans        |
|---|-----|----------|--------------|---------------|-------------|------------|
| 5 | 2   | 1        | []           | push          | [1]         | [-1,-1,-1] |
| 4 | 1   | 2        | [1]          | pop 1, push 2 | [2]         | [-1,-1,-1] |
| 3 | 0   | 1        | [2]          | push          | [2,1]       | [-1,-1,-1] |
| 2 | 2   | 1        | [2,1]        | pop 1         | [2]         | ans[2]=2   |
| 1 | 1   | 2        | [2]          | pop 2         | []          | ans[1]=-1  |
| 0 | 0   | 1        | []           | ans[0]=2      |             |            |

# 17. Next Smaller Element

You are given an integer array arr.

For every element in the array, you have to find its **Next Smaller Element (NSE)**.

The **Next Smaller Element** of an element  $x$  is the **first element to the right of  $x$  that is strictly smaller than  $x$** .

If no such element exists on the right side, the answer for that element is -1.

---

## Example Explanation

Example 1:

Input: [4, 8, 5, 2, 25]

- For 4 → next smaller is 2
- For 8 → next smaller is 5
- For 5 → next smaller is 2
- For 2 → no smaller element on the right → -1
- For 25 → no smaller element → -1

Output: [2, 5, 2, -1, -1]

---

## Approach 1 (Brute Force)

### Algorithm

- Create an answer array of size  $n$  and initialize all values with -1
- For each index  $i$  from 0 to  $n-1$ 
  - Traverse elements from  $i+1$  to  $n-1$

- If an element smaller than  $\text{arr}[i]$  is found
  - Store it in  $\text{ans}[i]$
  - Stop searching further
- Return the answer array

### Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 vector<int> nextSmallerElement(vector<int>& arr) {
 int n = arr.size();
 vector<int> ans(n, -1);

 for(int i = 0; i < n; i++) {
 for(int j = i + 1; j < n; j++) {
 if(arr[j] < arr[i]) {
 ans[i] = arr[j];
 break;
 }
 }
 }
 return ans;
 }
};

int main() {
 vector<int> arr = {4, 8, 5, 2, 25};
```

```

Solution sol;
vector<int> ans = sol.nextSmallerElement(arr);

for(int x : ans) cout << x << " ";
return 0;
}

```

## Complexity Analysis

- **Time Complexity:**  $O(N^2)$   
For each element, we may scan the remaining array.
  - **Space Complexity:**  $O(N)$   
Used for the result array.
- 

## Approach 2 (Optimal Using Stack)

### Algorithm

- Create an answer array initialized with -1
- Use a stack to keep elements in increasing order
- Traverse the array from right to left
- While stack is not empty and stack top  $\geq$  current element, pop the stack
- If stack is not empty, top of stack is the next smaller element
- Push current element into the stack
- Return the answer array

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 vector<int> nextSmallerElement(vector<int>& arr) {
 int n = arr.size();
 vector<int> ans(n, -1);
 stack<int> st;

 for(int i = n - 1; i >= 0; i--) {
 while(!st.empty() && st.top() >= arr[i]) {
 st.pop();
 }
 if(!st.empty()) {
 ans[i] = st.top();
 }
 st.push(arr[i]);
 }
 return ans;
 }
};

int main() {
 vector<int> arr = {4, 8, 5, 2, 25};
 Solution sol;
 vector<int> ans = sol.nextSmallerElement(arr);

 for(int x : ans) cout << x << " ";
 return 0;
}
```

## Complexity Analysis

- **Time Complexity:**  $O(N)$   
Each element is pushed and popped at most once.
- **Space Complexity:**  $O(N)$   
Stack and answer array require linear space.

# 18. Number of NGEs to the Right

You are given an integer array `arr` of size  $n$ .

For every element in the array, you have to find its **Next Greater Element (NGE)** on the right side.

The **Next Greater Element** of an element  $x$  is the **nearest element to the right of  $x$  that is strictly greater than  $x$** .

If no such element exists, return  $-1$  for that position.

---

## Example Explanation

Example 1:

Input: [1, 3, 2, 4]

- For 1, the next greater element is 3
- For 3, the next greater element is 4
- For 2, the next greater element is 4

- For 4, there is no greater element on the right → -1

Output: [3, 4, 4, -1]

---

## Approach 1 (Brute Force)

### Algorithm

- Create an answer array initialized with -1
- Traverse the array from left to right
- For each element at index  $i$ , check elements to its right
- The first element greater than the current one is the NGE
- If found, store it and stop checking further
- Return the answer array

### Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 vector<int> nextLargerElement(vector<int> arr) {
 int n = arr.size();
 vector<int> ans(n, -1);

 for(int i = 0; i < n; i++) {
 for(int j = i + 1; j < n; j++) {
```

```

 if(arr[j] > arr[i]) {
 ans[i] = arr[j];
 break;
 }
 }
 return ans;
}

int main() {
 vector<int> arr = {1, 3, 2, 4};
 Solution sol;
 vector<int> ans = sol.nextLargerElement(arr);

 for(int x : ans) cout << x << " ";
 return 0;
}

```

## Complexity Analysis

- **Time Complexity:**  $O(N^2)$   
For each element, we may scan all elements to its right.
  - **Space Complexity:**  $O(N)$   
Used for the answer array.
- 

## Approach 2 (Optimal Using Stack)

### Algorithm

- Create an answer array
- Use a stack to maintain elements in decreasing order
- Traverse the array from right to left
- While stack is not empty and top  $\leq$  current element, pop the stack
- If stack becomes empty, NGE is -1
- Otherwise, top of stack is the NGE
- Push current element into the stack
- Return the answer array

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 vector<int> nextLargerElement(vector<int> arr) {
 int n = arr.size();
 vector<int> ans(n);
 stack<int> st;

 for(int i = n - 1; i >= 0; i--) {
 while(!st.empty() && st.top() <= arr[i]) {
 st.pop();
 }
 if(st.empty()) ans[i] = -1;
 else ans[i] = st.top();
 st.push(arr[i]);
 }
 }
};
```

```

 }
 return ans;
}
};

int main() {
 vector<int> arr = {1, 3, 2, 4};
 Solution sol;
 vector<int> ans = sol.nextLargerElement(arr);

 for(int x : ans) cout << x << " ";
 return 0;
}

```

## Complexity Analysis

- **Time Complexity:** O(N)  
Each element is pushed and popped at most once.
- **Space Complexity:** O(N)  
Stack and answer array use linear space.

# 19. Number of NGEs to the Right

You are given an array of N integers and Q queries.

Each query gives an index, and for that index you have to find **how many elements to the right are strictly greater than the element at that index**.

In simple words, for a given index  $i$ , count all  $j$  such that:

- $j > i$
  - $\text{arr}[j] > \text{arr}[i]$
- 

## Example Explanation

Input array:

```
arr = {3, 4, 2, 7, 5, 8, 10, 6}
```

Queries:

- index = 0
- index = 5

For index 0 (value = 3):

Elements to the right that are greater than 3 → 4, 7, 5, 8, 10, 6

Count = 6

For index 5 (value = 8):

Elements to the right that are greater than 8 → 10

Count = 1

Output:

```
6 1
```

---

## Approach 1 (Naive Approach)

### Algorithm

- For every query index:

- Traverse the array from `index + 1` to the end
- Count how many elements are greater than `arr[index]`
- Print the count for each query

### Code

```
#include <bits/stdc++.h>
using namespace std;

int nextGreaterElements(vector<int>& a, int index) {
 int count = 0;
 int n = a.size();
 for(int i = index + 1; i < n; i++) {
 if(a[i] > a[index]) {
 count++;
 }
 }
 return count;
}

int main() {
 vector<int> a = {3, 4, 2, 7, 5, 8, 10, 6};
 vector<int> queries = {0, 5};

 for(int idx : queries) {
 cout << nextGreaterElements(a, idx) << " ";
 }
 return 0;
}
```

- **Time Complexity:**  $O(N * Q)$   
For each query, we may traverse the entire remaining array.
  - **Space Complexity:**  $O(1)$   
No extra space used apart from variables.
- 

## Approach 2 (Optimized Using Merge Sort Concept)

### Algorithm

- Create a vector of pairs (value, original\_index)
- Use a modified merge sort:
  - While merging, if `left_value < right_value`
    - Then all remaining elements in the right half are greater
    - Add that count to `ans[original_index]`
- This gives the count of elements greater on the right for **every index**
- For each query, simply print the stored answer

### Code

```
#include <bits/stdc++.h>
using namespace std;

void merge(vector<pair<int,int>>& vec, vector<int>& ans,
 int low, int mid, int high) {

 vector<pair<int,int>> left, right;
```

```

 for(int i = low; i <= mid; i++) left.push_back(vec[i]);
 for(int i = mid + 1; i <= high; i++)
 right.push_back(vec[i]);

 int i = 0, j = 0, k = low;

 while(i < left.size() && j < right.size()) {
 if(left[i].first < right[j].first) {
 ans[left[i].second] += (right.size() - j);
 vec[k++] = left[i++];
 } else {
 vec[k++] = right[j++];
 }
 }

 while(i < left.size()) vec[k++] = left[i++];
 while(j < right.size()) vec[k++] = right[j++];
 }

void mergeSort(vector<pair<int,int>>& vec, vector<int>& ans,
 int low, int high) {
 if(low >= high) return;

 int mid = (low + high) / 2;
 mergeSort(vec, ans, low, mid);
 mergeSort(vec, ans, mid + 1, high);
 merge(vec, ans, low, mid, high);
}

void nextGreaterElements(int n, vector<int>& nums,
 vector<int>& queries) {

```

```

vector<pair<int,int>> vec;
for(int i = 0; i < n; i++) {
 vec.push_back({nums[i], i});
}

vector<int> ans(n, 0);
mergeSort(vec, ans, 0, n - 1);

for(int idx : queries) {
 cout << ans[idx] << " ";
}
}

int main() {
 vector<int> nums = {3, 4, 2, 7, 5, 8, 10, 6};
 vector<int> queries = {0, 5};

 nextGreaterElements(nums.size(), nums, queries);
 return 0;
}

```

## Complexity Analysis

- **Time Complexity:**  $O(N \log N)$   
Due to merge sort.
- **Space Complexity:**  $O(N)$   
Extra arrays used during merge process.

# 20. Trapping Rainwater

You are given an array height where each element represents the height of a bar.

All bars have width 1. After raining, water can be trapped between these bars.

Your task is to calculate **the total units of water that can be trapped**.

Water can only be trapped if there are **taller bars on both the left and right sides** of a position.

---

## Example Explanation

### Example 1

Input:

```
height = [0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1]
```

At each index, water trapped depends on the minimum of the tallest bar on the left and right.

Total trapped water = 6 units.

---

### Example 2

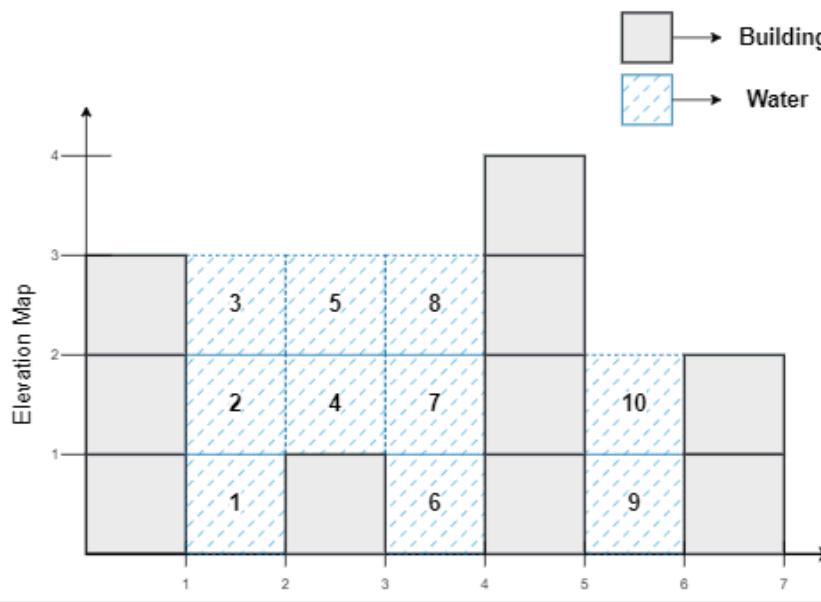
Input:

```
height = [4, 2, 0, 3, 2, 5]
```

Water gets trapped between taller bars on both sides.

Total trapped water = 9 units.

# Trapping Rainwater Problem



## Approach 1 (Brute Force)

### Algorithm

- For every index  $i$ :
  - Find the maximum height on the left of  $i$
  - Find the maximum height on the right of  $i$
  - Water at  $i = \min(\text{maxLeft}, \text{maxRight}) - \text{height}[i]$
- Add water for all indices

### Code

```
#include <bits/stdc++.h>
using namespace std;
```

```

class Solution {
public:
 int trap(vector<int>& height) {
 int n = height.size();
 int totalWater = 0;

 for(int i = 0; i < n; i++) {
 int maxLeft = 0, maxRight = 0;

 for(int j = 0; j <= i; j++)
 maxLeft = max(maxLeft, height[j]);

 for(int j = i; j < n; j++)
 maxRight = max(maxRight, height[j]);

 totalWater += min(maxLeft, maxRight) -
height[i];
 }
 return totalWater;
 }
};

int main() {
 vector<int> height = {0,1,0,2,1,0,1,3,2,1,2,1};
 Solution sol;
 cout << sol.trap(height);
 return 0;
}

```

- **Time Complexity:**  $O(N^2)$   
For each bar, we scan left and right.

- Space Complexity:  $O(1)$
- 

**Ek approach to use prefix and suffix max array, isme prefix array ki jarurat hai nhi wo to traverse karte hue hi kaam chal jayega**

---

## Approach 2 (Optimal Two Pointer Approach)

Intuition: we only need one (either left or right) to calculate water at given index

### Algorithm

- Initialize two pointers: `left = 0, right = n - 1`
- Maintain `maxLeft` and `maxRight`
- While `left <= right`:
  - If `height[left] <= height[right]`:
    - If `height[left] >= maxLeft`, update `maxLeft`
    - Else add `maxLeft - height[left]` to answer
    - Move `left`
  - Else:
    - If `height[right] >= maxRight`, update `maxRight`

- Else add  $\maxRight - \text{height}[\text{right}]$  to answer
- Move right

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int trap(vector<int>& height) {
 int n = height.size();
 int left = 0, right = n - 1;
 int maxLeft = 0, maxRight = 0;
 int totalWater = 0;

 while(left <= right) {
 if(height[left] <= height[right]) {
 if(height[left] >= maxLeft)
 maxLeft = height[left];
 else
 totalWater += maxLeft - height[left];
 left++;
 } else {
 if(height[right] >= maxRight)
 maxRight = height[right];
 else
 totalWater += maxRight - height[right];
 right--;
 }
 }
 }
}
```

```

 return totalWater;
 }
};

int main() {
 vector<int> height = {0,1,0,2,1,0,1,3,2,1,2,1};
 Solution sol;
 cout << sol.trap(height);
 return 0;
}

```

## Complexity Analysis

- **Time Complexity:**  $O(N)$   
Each element is processed once.
- **Space Complexity:**  $O(1)$

# 21. Sum of Subarray Minimums

You are given an integer array `arr` of size  $n$ .

For **every possible contiguous subarray**, find the **minimum element** of that subarray and **add all those minimum values together**.

Since the answer can be very large, return it modulo  $10^9 + 7$ .

In simple words:

👉 Generate all subarrays → find the minimum in each → sum them all.

---

## Example Explanation

### Example 1

Input:

arr = [3, 1, 2, 5]

All subarrays and their minimums:

- [3] → 3
- [1] → 1
- [2] → 2
- [5] → 5
- [3, 1] → 1
- [1, 2] → 1
- [2, 5] → 2
- [3, 1, 2] → 1
- [1, 2, 5] → 1
- [3, 1, 2, 5] → 1

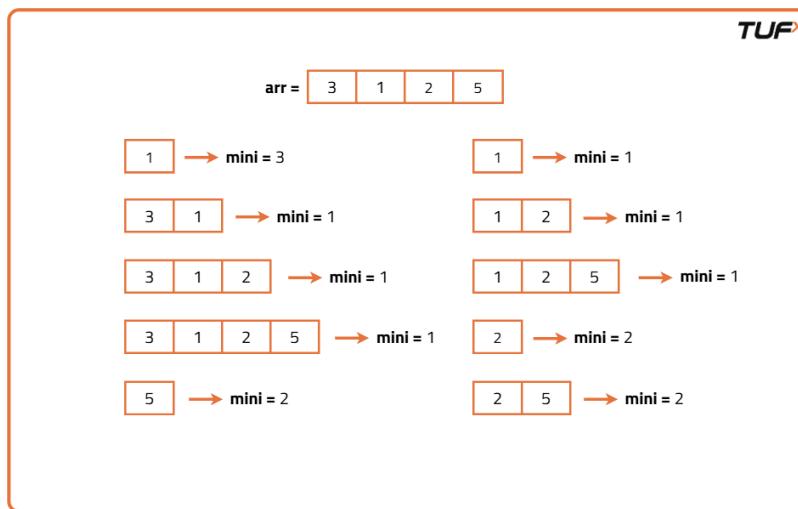
Sum = 18

---

## Approach 1 (Brute Force)

### Algorithm

- Initialize sum = 0
- Fix the starting index i
- Initialize mini = arr[i]
- Extend the subarray to the right using index j
- Update mini for every extension
- Add mini to sum
- Repeat for all starting indices



### Code

```
#include <bits/stdc++.h>
using namespace std;
```

```

class Solution {
public:
 int sumSubarrayMins(vector<int>& arr) {
 int n = arr.size();
 int mod = 1e9 + 7;
 int sum = 0;

 for(int i = 0; i < n; i++) {
 int mini = arr[i];
 for(int j = i; j < n; j++) {
 mini = min(mini, arr[j]);
 sum = (sum + mini) % mod;
 }
 }
 return sum;
 }
};

int main() {
 vector<int> arr = {3,1,2,5};
 Solution sol;
 cout << sol.sumSubarrayMins(arr);
 return 0;
}

```

## Complexity Analysis

- **Time Complexity:**  $O(N^2)$   
Two nested loops generate all subarrays.

- **Space Complexity:**  $O(1)$

Only variables are used.

---

## Approach 2 (Optimal Using Stack)

### Algorithm

Key idea:

Instead of generating subarrays, **count how many subarrays consider  $\text{arr}[i]$  as the minimum.**

Steps:

- Find **Next Smaller Element (NSE)** index for every element
- Find **Previous Smaller or Equal Element (PSEE)** index for every element
- For each index  $i$ :
  - Left choices =  $i - \text{psee}[i]$
  - Right choices =  $\text{nse}[i] - i$
  - Total subarrays where  $\text{arr}[i]$  is minimum =  $\text{left} * \text{right}$
- Contribution =  $\text{arr}[i] * \text{left} * \text{right}$
- Add all contributions modulo  $10^9 + 7$

### Code

```
#include <bits/stdc++.h>
using namespace std;
```

```

class Solution {
private:
 vector<int> findNSE(vector<int>& arr) {
 int n = arr.size();
 vector<int> ans(n);
 stack<int> st;

 for(int i = n - 1; i >= 0; i--) {
 while(!st.empty() && arr[st.top()] >= arr[i])
 st.pop();
 ans[i] = st.empty() ? n : st.top();
 st.push(i);
 }
 return ans;
 }

 vector<int> findPSEE(vector<int>& arr) {
 int n = arr.size();
 vector<int> ans(n);
 stack<int> st;

 for(int i = 0; i < n; i++) {
 while(!st.empty() && arr[st.top()] > arr[i])
 st.pop();
 ans[i] = st.empty() ? -1 : st.top();
 st.push(i);
 }
 return ans;
 }

public:

```

```

int sumSubarrayMins(vector<int>& arr) {
 int n = arr.size();
 int mod = 1e9 + 7;

 vector<int> nse = findNSE(arr);
 vector<int> psee = findPSEE(arr);

 long long sum = 0;

 for(int i = 0; i < n; i++) {
 long long left = i - psee[i];
 long long right = nse[i] - i;
 sum = (sum + (left * right % mod) * arr[i]) %
mod;
 }
 return sum;
}

int main() {
 vector<int> arr = {3,1,2,5};
 Solution sol;
 cout << sol.sumSubarrayMins(arr);
 return 0;
}

```

## Complexity Analysis

- **Time Complexity:**  $O(N)$   
Each element is pushed and popped once from stack.

- **Space Complexity:**  $O(N)$   
Stack and helper arrays are used.

## 22. Asteroid Collision

You are given an integer array `asteroids`.

Each element represents an asteroid moving in a straight line.

- **Absolute value** → size of the asteroid
- **Sign** → direction
  - Positive (+) → moving right
  - Negative (-) → moving left

All asteroids move at the same speed.

When two asteroids collide:

- Smaller asteroid explodes
- If both have the same size, both explode
- Asteroids moving in the same direction never collide

You need to return the **final state of asteroids after all collisions**.

---

## **Example Explanation**

### **Example 1**

Input:

[2, -2]

- 2 moves right, -2 moves left
- Same size → both explode

Output:

[ ]

---

### **Example 2**

Input:

[10, 20, -10]

- 10 and 20 move right
- -10 moves left and collides with 20
- 20 is larger → -10 explodes

Final result:

[10, 20]

---

## **Approach 1**

### **Algorithm**

- Use a stack to keep track of asteroids after collisions
- Traverse the array from left to right
- If the asteroid is moving right ( $> 0$ ), push it into the stack
- If the asteroid is moving left ( $< 0$ ):
  - While the stack is not empty and the top is a smaller right-moving asteroid, pop it
  - If the top asteroid is the same size, pop it and do not push current
  - If the stack is empty or the top is left-moving, push the current asteroid
- After processing all asteroids, the stack contains the final state

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 vector<int> asteroidCollision(vector<int>& asteroids) {
 vector<int> st;

 for(int i = 0; i < asteroids.size(); i++) {
 if(asteroids[i] > 0) {
 st.push_back(asteroids[i]);
 } else {
 while(!st.empty() && st.back() > 0 &&
st.back() < abs(asteroids[i])) {
 st.pop_back();
 }
 }
 }
 return st;
 }
};
```

```

 }

 if(!st.empty() && st.back() ==
abs(asteroids[i])) {
 st.pop_back();
 }
 else if(st.empty() || st.back() < 0) {
 st.push_back(asteroids[i]);
 }
 }
 return st;
}
};

int main() {
 vector<int> asteroids = {10, 20, -10};
 Solution sol;

 vector<int> ans = sol.asteroidCollision(asteroids);

 for(int x : ans) {
 cout << x << " ";
 }
 return 0;
}

```

## Complexity Analysis

- **Time Complexity:**  $O(N)$   
Each asteroid is pushed and popped at most once.

- **Space Complexity:**  $O(N)$   
Stack can store all asteroids in the worst case.

## 23. Sum of Subarray Ranges

You are given an integer array `nums`.

For every **contiguous subarray**, its **range** is defined as:

`range = maximum element - minimum element`

Your task is to calculate the **sum of ranges of all possible subarrays**.

A subarray must be **non-empty and contiguous**.

---

### Example Explanation

#### Example 1

Input:

`nums = [1, 2, 3]`

All subarrays and their ranges:

- $[1] \rightarrow 0$
- $[2] \rightarrow 0$
- $[3] \rightarrow 0$
- $[1, 2] \rightarrow 2 - 1 = 1$

- $[2, 3] \rightarrow 3 - 2 = 1$
- $[1, 2, 3] \rightarrow 3 - 1 = 2$

Sum = 0 + 0 + 0 + 1 + 1 + 2 = 4

Output:

4

---

## Approach 1 (Brute Force)

### Algorithm

- Initialize a variable sum = 0
- Fix the starting index i of the subarray
- Initialize smallest and largest as arr[i]
- Extend the subarray to the right using index j
- Update smallest and largest
- Add (largest - smallest) to sum
- Repeat for all possible subarrays
- Return sum

### Code

```
#include <bits/stdc++.h>
using namespace std;
```

```

class Solution {
public:
 long long subArrayRanges(vector<int>& arr) {
 int n = arr.size();
 long long sum = 0;

 for(int i = 0; i < n; i++) {
 int smallest = arr[i];
 int largest = arr[i];

 for(int j = i; j < n; j++) {
 smallest = min(smallest, arr[j]);
 largest = max(largest, arr[j]);
 sum += (largest - smallest);
 }
 }
 return sum;
 }
};

int main() {
 vector<int> arr = {1, 2, 3};
 Solution sol;
 cout << sol.subArrayRanges(arr);
 return 0;
}

```

## Complexity Analysis

- **Time Complexity:**  $O(N^2)$

- **Space Complexity:**  $O(1)$
- 

## Approach 2 (Optimal Using Monotonic Stack)

### Algorithm

- Every element contributes to the final answer as:
  - how many subarrays where it is **maximum**
  - how many subarrays where it is **minimum**
- Total answer =  
 $\text{sum of subarray maximums} - \text{sum of subarray minimums}$

Steps:

1. Find **Next Smaller Element (NSE)** for each index
2. Find **Previous Smaller or Equal Element (PSEE)**
3. Find **Next Greater Element (NGE)**
4. Find **Previous Greater or Equal Element (PGEE)**
5. Use these indices to count how many subarrays each element is minimum and maximum
6. Compute total maximum contribution
7. Compute total minimum contribution
8. Subtract both results

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
private:
 vector<int> findNSE(vector<int>& arr) {
 int n = arr.size();
 vector<int> ans(n);
 stack<int> st;

 for(int i = n - 1; i >= 0; i--) {
 while(!st.empty() && arr[st.top()] >= arr[i])
 st.pop();
 ans[i] = st.empty() ? n : st.top();
 st.push(i);
 }
 return ans;
 }

 vector<int> findPSEE(vector<int>& arr) {
 int n = arr.size();
 vector<int> ans(n);
 stack<int> st;

 for(int i = 0; i < n; i++) {
 while(!st.empty() && arr[st.top()] > arr[i])
 st.pop();
 ans[i] = st.empty() ? -1 : st.top();
 st.push(i);
 }
 return ans;
 }
}
```

```

}

vector<int> findNGE(vector<int>& arr) {
 int n = arr.size();
 vector<int> ans(n);
 stack<int> st;

 for(int i = n - 1; i >= 0; i--) {
 while(!st.empty() && arr[st.top()] <= arr[i])
 st.pop();
 ans[i] = st.empty() ? n : st.top();
 st.push(i);
 }
 return ans;
}

vector<int> findPGE(vector<int>& arr) {
 int n = arr.size();
 vector<int> ans(n);
 stack<int> st;

 for(int i = 0; i < n; i++) {
 while(!st.empty() && arr[st.top()] < arr[i])
 st.pop();
 ans[i] = st.empty() ? -1 : st.top();
 st.push(i);
 }
 return ans;
}

long long sumSubarrayMins(vector<int>& arr) {
 vector<int> nse = findNSE(arr);

```

```

vector<int> psee = findPSEE(arr);
long long sum = 0;

for(int i = 0; i < arr.size(); i++) {
 long long left = i - psee[i];
 long long right = nse[i] - i;
 sum += left * right * arr[i];
}
return sum;
}

long long sumSubarrayMaxs(vector<int>& arr) {
 vector<int> nge = findNGE(arr);
 vector<int> pgue = findPGEE(arr);
 long long sum = 0;

 for(int i = 0; i < arr.size(); i++) {
 long long left = i - pgue[i];
 long long right = nge[i] - i;
 sum += left * right * arr[i];
 }
 return sum;
}

public:
 long long subArrayRanges(vector<int>& arr) {
 return sumSubarrayMaxs(arr) - sumSubarrayMins(arr);
 }
};

int main() {
 vector<int> arr = {1, 2, 3};

```

```
Solution sol;
cout << sol.subArrayRanges(arr);
return 0;
}
```

## Complexity Analysis

- **Time Complexity:**  $O(N)$
- **Space Complexity:**  $O(N)$

# 24. Remove K Digits

You are given a string `nums` that represents a non-negative integer and an integer `k`.

Your task is to remove exactly `k` digits from `nums` such that the resulting number is the **smallest possible**.

The final answer:

- must not contain **leading zeroes**
- if all digits are removed or only zeroes remain, return "0"

---

## Example Explanation

### Example 1

Input:

nums = "541892", k = 2

We want the smallest number after removing 2 digits.

Steps:

- Remove 5 because next digit 4 is smaller
- Remove 4 because next digit 1 is smaller

Remaining digits → "1892"

Output:

"1892"

---

## Example 2

Input:

nums = "1002991", k = 3

Steps:

- Remove leading 1
- Remove 9 and 9
- Remaining digits → "0021"
- Remove leading zeroes → "21"

Output:

"21"

---

**Approach 1 - Number ko minimum banana hai, isliye left se bade digits ko hatao jab unke right me chhota digit mil jaaye.**

### **Algorithm**

- Use a stack to build the smallest number
- Traverse digits from left to right
- While:
  - stack is not empty
  - $k > 0$
  - top of stack is greater than current digit  
→ pop from stack and decrement k
- Push the current digit into the stack
- If  $k > 0$  after traversal, remove last k digits from stack
- Build the result from the stack
- Remove leading zeroes
- If result becomes empty, return "0"

### **Code**

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
```

```

public:
 string removeKdigits(string nums, int k) {
 stack<char> st;

 for(int i = 0; i < nums.size(); i++) {
 char digit = nums[i];

 while(!st.empty() && k > 0 && st.top() > digit)
{
 st.pop();
 k--;
 }
 st.push(digit);
 }

 while(!st.empty() && k > 0) {
 st.pop();
 k--;
 }

 if(st.empty()) return "0";

 string res = "";
 while(!st.empty()) {
 res.push_back(st.top());
 st.pop();
 }

 while(res.size() > 0 && res.back() == '0') {
 res.pop_back();
 }
 }
}

```

```

 reverse(res.begin(), res.end()));

 if(res.empty()) return "0";
 return res;
 }
};

int main() {
 string nums = "541892";
 int k = 2;

 Solution sol;
 cout << sol.removeKdigits(nums, k);
 return 0;
}

```

## Complexity Analysis

- **Time Complexity:**  $O(N)$   
Each digit is pushed and popped at most once.
- **Space Complexity:**  $O(N)$   
Stack can store up to all digits in the worst case.

# 25. Area of Largest Rectangle in Histogram

You are given an array `heights[ ]` where each element represents the height of a bar in a histogram.

Each bar has a width of 1.

Your task is to find the **maximum rectangular area** that can be formed using contiguous bars in the histogram.

---

## Example Explanation

Input:

```
heights = [2, 1, 5, 6, 2, 3]
```

Possible rectangles:

- Using bar 2 → area =  $2 \times 1 = 2$
- Using bar 1 across all bars → area =  $1 \times 6 = 6$
- Using bars [5, 6] → min height = 5, width = 2 → area = 10
- Other combinations give smaller areas

So, the **maximum area = 10**.

---

## Approach 1: Brute Force

### Algorithm

- Consider every bar as the starting point.
- Extend to the right and keep track of the minimum height in the current range.
- For each range, calculate area = minimum height × width.
- Keep updating the maximum area.

## Code

```
#include <bits/stdc++.h>
using namespace std;

int largestarea(int arr[], int n) {
 int maxArea = 0;

 for (int i = 0; i < n; i++) {
 int minHeight = INT_MAX;

 for (int j = i; j < n; j++) {
 minHeight = min(minHeight, arr[j]);
 int width = j - i + 1;
 maxArea = max(maxArea, minHeight * width);
 }
 }
 return maxArea;
}

int main() {
 int arr[] = {2, 1, 5, 6, 2, 3};
 int n = 6;
 cout << largestarea(arr, n);
 return 0;
}
```

## Complexity Analysis

- **Time Complexity:**  $O(N^2)$
- **Space Complexity:**  $O(1)$

---

## Approach 2: Optimized Using Previous & Next Smaller Elements

### Algorithm

- For each bar, find:
  - nearest smaller element to the left
  - nearest smaller element to the right
- Width = rightSmaller – leftSmaller + 1
- Area = height × width
- Take the maximum among all bars.

### Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int largestRectangleArea(vector<int> &heights) {
 int n = heights.size();
 stack<int> st;
 vector<int> left(n), right(n);

 for (int i = 0; i < n; i++) {
 while (!st.empty() && heights[st.top()] >=
heights[i])
 st.pop();
 if (st.empty())
 left[i] = -1;
 else
 left[i] = st.top();
 st.push(i);
 }

 while (!st.empty())
 st.pop();
 for (int i = n - 1; i >= 0; i--) {
 while (!st.empty() && heights[st.top()] >=
heights[i])
 st.pop();
 if (st.empty())
 right[i] = n;
 else
 right[i] = st.top();
 st.push(i);
 }
 }
}
```

```

 left[i] = st.empty() ? 0 : st.top() + 1;
 st.push(i);
 }

 while (!st.empty()) st.pop();

 for (int i = n - 1; i >= 0; i--) {
 while (!st.empty() && heights[st.top()] >=
heights[i])
 st.pop();
 right[i] = st.empty() ? n - 1 : st.top() - 1;
 st.push(i);
 }

 int maxA = 0;
 for (int i = 0; i < n; i++) {
 int width = right[i] - left[i] + 1;
 maxA = max(maxA, heights[i] * width);
 }
 return maxA;
}

};

int main() {
 vector<int> heights = {2, 1, 5, 6, 2, 3};
 Solution obj;
 cout << obj.largestRectangleArea(heights);
 return 0;
}

```

## Complexity Analysis

- **Time Complexity:** O(N)
  - **Space Complexity:** O(N)
- 

## Approach 3: Single Pass Stack Approach

### Algorithm

- Use a stack to store indices of increasing bars.
- Traverse the array:
  - If current bar is smaller, pop from stack and calculate area.
  - Width is decided using current index and new stack top.
- Add an imaginary bar of height 0 at the end to flush the stack.
- Track the maximum area.

**Jab tak bars increasing hain, hum wait karte hain.  
Jaise hi koi chhoti bar milti hai,  
toh pichhli badi bars ka “right boundary” mil jaata hai.**

### Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int largestRectangleArea(vector<int> &histo) {
```

```

 stack<int> st;
 int maxA = 0;
 int n = histo.size();

 for (int i = 0; i <= n; i++) {
 while (!st.empty() && (i == n || histo[st.top()]
>= histo[i])) {
 int height = histo[st.top()];
 st.pop();

 int width = st.empty() ? i : i - st.top()-1;
 maxA = max(maxA, height * width);
 }
 st.push(i);
 }
 return maxA;
}

};

int main() {
 vector<int> histo = {2, 1, 5, 6, 2, 3};
 Solution obj;
 cout << obj.largestRectangleArea(histo);
 return 0;
}

```

## Complexity Analysis

- **Time Complexity:** O(N)
- **Space Complexity:** O(N)

# 26. Maximal Rectangles

You are given a binary matrix of size  $m \times n$  containing only 0 and 1.

Your task is to find the **largest rectangle** that contains **only 1's** and return its area.

A rectangle must be formed by **contiguous rows and columns**.

---

## Question Explanation

Each cell in the matrix can either contribute to a rectangle (if it is 1) or break it (if it is 0).

To solve this, we convert each row into a **histogram of heights** where each height represents the number of consecutive 1s above that cell (including itself).

Then, for every row's histogram, we calculate the **largest rectangle in a histogram** and keep track of the maximum area found.

---

## Example Explanation

### Input

```
matrix =
[
 [1, 0, 1, 0, 0],
 [1, 0, 1, 1, 1],
 [1, 1, 1, 1, 1],
 [1, 0, 0, 1, 0]
```

]

We build histograms row by row:

Row 0 → [1, 0, 1, 0, 0]

Row 1 → [2, 0, 2, 1, 1]

Row 2 → [3, 1, 3, 2, 2]

Row 3 → [4, 0, 0, 3, 0]

The largest rectangle appears in row 2 with height 2 and width 3, so

**Area = 2 × 3 = 6**

---

## Approach 1

### Algorithm

1. Convert the matrix into histogram heights:
  - If `matrix[i][j] == 1`, increase height from above
  - If `matrix[i][j] == 0`, reset height to 0
2. For each row's histogram:
  - Use a stack-based method to find the largest rectangle in a histogram
3. Track the maximum area across all rows
4. Return the maximum area

---

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
private:
 int largestRectangleArea(vector<int> &heights) {
 int n = heights.size();
 stack<int> st;
 int maxArea = 0;

 for(int i = 0; i <= n; i++) {
 while(!st.empty() && (i == n || heights[st.top()] >= heights[i])) {
 int h = heights[st.top()];
 st.pop();
 int width = st.empty() ? i : i - st.top() - 1;
 maxArea = max(maxArea, h * width);
 }
 st.push(i);
 }
 return maxArea;
 }

public:
 int maximalAreaOfSubMatrixOfAll1(vector<vector<int>> &matrix) {
 int n = matrix.size();
 int m = matrix[0].size();
```

```

vector<vector<int>> prefix(n, vector<int>(m, 0));

for(int j = 0; j < m; j++) {
 int height = 0;
 for(int i = 0; i < n; i++) {
 if(matrix[i][j] == 1) {
 height++;
 } else {
 height = 0;
 }
 prefix[i][j] = height;
 }
}

int maxArea = 0;
for(int i = 0; i < n; i++) {
 maxArea = max(maxArea,
largestRectangleArea(prefix[i]));
}

return maxArea;
}
};

int main() {
 vector<vector<int>> matrix = {
 {1,0,1,0,0},
 {1,0,1,1,1},
 {1,1,1,1,1},
 {1,0,0,1,0}
 };
}

```

```
Solution sol;
cout << sol.maximalAreaOfSubMatrixOfAll1(matrix);
return 0;
}
```

---

## Complexity Analysis

- **Time Complexity:**  $O(N \times M)$   
Each row is processed as a histogram in linear time.
- **Space Complexity:**  $O(N \times M)$   
Prefix matrix plus stack space for histogram calculation.

# 27. Sliding Window Maximum

You are given an integer array `arr` and an integer `k`.

A sliding window of size `k` moves from the left of the array to the right, one step at a time.

For each window position, you must find the **maximum element** inside that window and return all such maximums.

A sliding window always contains **k consecutive elements**.

---

## Question Explanation

At each step, the window includes `k` elements.

When the window moves right by one index:

- One element goes out of the window

- One new element comes into the window

For every such window, we only care about the **largest value** inside it.

---

## Example Explanation

### Input

`arr = [4, 0, -1, 3, 5, 3, 6, 8], k = 3`

Windows and maximums:

- `[4, 0, -1] → max = 4`
- `[0, -1, 3] → max = 3`
- `[-1, 3, 5] → max = 5`
- `[3, 5, 3] → max = 5`
- `[5, 3, 6] → max = 6`
- `[3, 6, 8] → max = 8`

### Output

`[4, 3, 5, 5, 6, 8]`

---

## Approach 1 (Brute Force)

### Algorithm

- Loop through the array from index 0 to  $n - k$
  - For each position, consider the subarray of size  $k$
  - Scan all  $k$  elements to find the maximum
  - Store the maximum in the result array
  - Return the result
- 

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 vector<int> maxSlidingWindow(vector<int>& nums, int k) {
 vector<int> result;
 int n = nums.size();

 for(int i = 0; i <= n - k; i++) {
 int maxi = nums[i];
 for(int j = i; j < i + k; j++) {
 maxi = max(maxi, nums[j]);
 }
 result.push_back(maxi);
 }
 return result;
 }
};
```

```

int main() {
 vector<int> arr = {4,0,-1,3,5,3,6,8};
 int k = 3;

 Solution sol;
 vector<int> ans = sol.maxSlidingWindow(arr, k);

 for(int x : ans) cout << x << " ";
 return 0;
}

```

---

- **Time Complexity:**  $O(N * K)$   
For each window, we scan  $k$  elements.
  - **Space Complexity:**  $O(1)$   
No extra data structures used (output array excluded).
- 

## Approach 2 (Optimal – Deque)

### Algorithm

- Use a deque to store **indices** of useful elements
- Maintain elements in decreasing order inside the deque
- For each index  $i$ :
  - Remove indices from the front if they are outside the window

- Remove indices from the back whose values are smaller than  $\text{arr}[i]$
  - Push current index to the back
  - If window size is at least  $k$ , add  $\text{arr}[\text{dq}.front()]$  to result
  - Return the result
- 

### Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 vector<int> maxSlidingWindow(vector<int>& nums, int k) {
 deque<int> dq;
 vector<int> result;

 for(int i = 0; i < nums.size(); i++) {
 if(!dq.empty() && dq.front() <= i - k) {
 dq.pop_front();
 }

 while(!dq.empty() && nums[dq.back()] < nums[i])
 {
 dq.pop_back();
 }

 dq.push_back(i);
 }

 return result;
 }
}
```

```

 if(i >= k - 1) {
 result.push_back(nums[dq.front()]);
 }
 }
 return result;
}

int main() {
 vector<int> arr = {4,0,-1,3,5,3,6,8};
 int k = 3;

 Solution sol;
 vector<int> ans = sol.maxSlidingWindow(arr, k);

 for(int x : ans) cout << x << " ";
 return 0;
}

```

---

## Complexity Analysis

- **Time Complexity:**  $O(N)$   
Each element is inserted and removed from the deque at most once.
- **Space Complexity:**  $O(K)$   
Deque stores at most  $k$  indices

# 28. Stock Span Problem

You are given an array  $\text{arr}$  of size  $n$ , where  $\text{arr}[i]$  represents the stock price on day  $i$ .

For each day, you need to calculate the **stock span**.

The **span** of a stock on day  $i$  is the maximum number of consecutive days **just before day  $i$  (including day  $i$ )** for which the stock price was **less than or equal to** the price on day  $i$ .

aaj se peechhe kitne continuous din tak stock price  $\leq$  aaj ka price

---

For every day, we look backwards and count how many continuous days (including today) have prices  $\leq$  **today's price**.

The moment we find a price **greater than today's price**, we stop counting.

---

**Example:**

**Input**

$\text{arr} = [120, 100, 60, 80, 90, 110, 115]$

- Day 0 (120): no previous days  $\rightarrow$  span = 1
- Day 1 (100):  $100 < 120 \rightarrow$  span = 1
- Day 2 (60):  $60 < 100 \rightarrow$  span = 1
- Day 3 (80):  $80 \geq 60$ , but  $< 100 \rightarrow$  span = 2
- Day 4 (90):  $90 \geq 60, 80$ , but  $< 100 \rightarrow$  span = 3

- Day 5 (110):  $110 \geq 60, 80, 90, 100 \rightarrow \text{span} = 5$
- Day 6 (115):  $115 \geq \text{all except } 120 \rightarrow \text{span} = 6$

## Output

1 1 1 2 3 5 6

---

## Approach 1 (Brute Force)

### Algorithm

- Create an answer array of size n
- For each index i:
  - Start from i and move backwards
  - Count consecutive days where  $\text{arr}[j] \leq \text{arr}[i]$
  - Stop when a greater element is found
- Store the count as the span for that day
- Return the answer array

---

### Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
```

```

public:
 vector<int> stockSpan(vector<int> arr, int n) {
 vector<int> ans(n);

 for(int i = 0; i < n; i++) {
 int span = 0;
 for(int j = i; j >= 0; j--) {
 if(arr[j] <= arr[i])
 span++;
 else
 break;
 }
 ans[i] = span;
 }
 return ans;
 }

};

int main() {
 vector<int> arr = {120, 100, 60, 80, 90, 110, 115};
 int n = arr.size();

 Solution sol;
 vector<int> ans = sol.stockSpan(arr, n);

 for(int x : ans) cout << x << " ";
 return 0;
}

```

---

## Complexity Analysis

- **Time Complexity:**  $O(N^2)$   
For each day, we may scan all previous days.
  - **Space Complexity:**  $O(1)$   
Only variables used (output array excluded).
- 

## Approach 2 (Optimal Using Stack)

### Algorithm

- Use a stack to store **indices** of days with prices in decreasing order
- Traverse the array from left to right
- For each day:
  - Pop indices from stack while their price  $\leq$  current price
  - If stack becomes empty  $\rightarrow$  no previous greater element  $\rightarrow$  span =  $i + 1$
  - Else  $\rightarrow$  span =  $i - \text{index\_of\_previous\_greater}$
- Push current index into the stack
- Return the span array

---

Jab tak peeche ka price chhota ya barabar hai, wo aaj ke span ka hissa hai — use hata do.

---

### Code

```

#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 vector<int> stockSpan(vector<int> arr, int n) {
 vector<int> ans(n);
 stack<int> st;

 for(int i = 0; i < n; i++) {
 while(!st.empty() && arr[st.top()] <= arr[i]) {
 st.pop();
 }

 if(st.empty())
 ans[i] = i + 1;
 else
 ans[i] = i - st.top();

 st.push(i);
 }
 return ans;
 }
};

int main() {
 vector<int> arr = {120, 100, 60, 80, 90, 110, 115};
 int n = arr.size();

 Solution sol;
 vector<int> ans = sol.stockSpan(arr, n);
}

```

```
 for(int x : ans) cout << x << " ";
 return 0;
}
```

---

## Complexity Analysis

- **Time Complexity:**  $O(N)$

Each element is pushed and popped from the stack at most once.

- **Space Complexity:**  $O(N)$

Stack can store up to  $N$  indices in the worst case.

# 29. Celebrity Problem

A celebrity is someone who **is known by everyone else** but **does not know anyone**.

You are given a square matrix  $M$  of size  $N \times N$  where:

- $M[i][j] = 1$  means person  $i$  knows person  $j$
- $M[i][j] = 0$  means person  $i$  does not know person  $j$
- $M[i][i] = 0$  for all  $i$

Your task is to determine whether a celebrity exists.

If yes, return the **index** of the celebrity, otherwise return  $-1$ .

---

## Example Explanation

### Example 1

```
M = [
 [0, 1, 1, 0],
 [0, 0, 0, 0],
 [1, 1, 0, 0],
 [0, 1, 1, 0]
]
```

- Person 1 knows nobody (row 1 is all 0)
- Everyone else knows person 1  
So, person 1 is the celebrity.

**Output:** 1

---

## Approach 1 (Brute Force)

### Algorithm

- Create two arrays:
  - `knowMe[i]` → how many people know person *i*
  - `IKnow[i]` → how many people person *i* knows
- Traverse the entire matrix:

- If  $M[i][j] == 1$ , increment  $\text{IKnow}[i]$  and  $\text{knowMe}[j]$
  - A person  $i$  is a celebrity if:
    - $\text{knowMe}[i] == N - 1$
    - $\text{IKnow}[i] == 0$
  - If such a person exists, return their index, otherwise return  $-1$
- 

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int celebrity(vector<vector<int>> &M) {
 int n = M.size();
 vector<int> knowMe(n, 0), IKnow(n, 0);

 for(int i = 0; i < n; i++) {
 for(int j = 0; j < n; j++) {
 if(M[i][j] == 1) {
 IKnow[i]++;
 knowMe[j]++;
 }
 }
 }

 for(int i = 0; i < n; i++) {
```

```

 if(knowMe[i] == n - 1 && IKnow[i] == 0)
 return i;
 }
 return -1;
}

int main() {
 vector<vector<int>> M = {
 {0,1,1,0},
 {0,0,0,0},
 {1,1,0,0},
 {0,1,1,0}
 };
}

Solution sol;
cout << sol.celebrity(M);
return 0;
}

```

---

## Complexity Analysis

- **Time Complexity:**  $O(N^2)$   
Full traversal of the matrix.
  - **Space Complexity:**  $O(N)$   
Two auxiliary arrays of size N.
- 

## Approach 2 (Optimal – Two Pointer)

## Algorithm

- Use two pointers:
  - `top = 0`
  - `down = N - 1`
- While `top < down`:
  - If `top` knows `down`, then `top` cannot be celebrity → `top++`
  - Else if `down` knows `top`, then `down` cannot be celebrity → `down--`
  - Else both cannot be celebrity → move both
- After loop, `top` is the potential celebrity
- Verify:
  - Candidate knows nobody
  - Everyone knows the candidate
- If valid, return index, else return -1

---

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
```

```

int celebrity(vector<vector<int>> &M) {
 int n = M.size();
 int top = 0, down = n - 1;

 while(top < down) {
 if(M[top][down] == 1)
 top++;
 else if(M[down][top] == 1)
 down--;
 else {
 top++;
 down--;
 }
 }

 for(int i = 0; i < n; i++) {
 if(i != top && (M[top][i] == 1 || M[i][top] ==
0))
 return -1;
 }
 return top;
}

int main() {
 vector<vector<int>> M = {
 {0,1,1,0},
 {0,0,0,0},
 {1,1,0,0},
 {0,1,1,0}
 };
}

```

```
Solution sol;
cout << sol.celebrity(M);
return 0;
}
```

---

## Complexity Analysis

- **Time Complexity:**  $O(N)$   
One pass to eliminate candidates and one pass to verify.
- **Space Complexity:**  $O(1)$   
Only constant extra variables used.

## 30. Implement LRU Cache

An **LRU (Least Recently Used) Cache** stores a fixed number of key–value pairs.

When the cache is full and a new key is inserted, the **least recently used** key is removed.

The challenge is that both operations must run in  **$O(1)$**  average time:

- `get(key)`
  - `put(key, value)`
-

## Key Idea

To achieve **O(1)** time for both operations, we combine **two data structures**:

### 1. **HashMap**

- Maps key → node
- Allows instant lookup

### 2. **Doubly Linked List**

- Maintains usage order
- Most Recently Used (MRU) → near head
- Least Recently Used (LRU) → near tail

## Why this works

- HashMap gives fast access
- Doubly linked list gives fast insert/delete
- When a key is accessed or updated → move it to the front
- When capacity exceeds → remove the node at the tail

---

## Algorithm

## Initialization

- Create dummy head and tail nodes
- Cache capacity is fixed

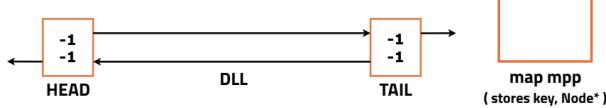
### **get(key)**

- If key does not exist → return -1
- Else:
  - Move the node to the front (MRU)
  - Return its value

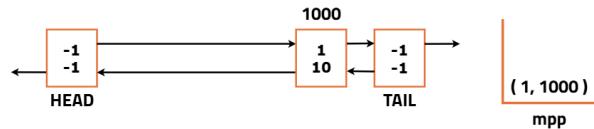
### **put(key, value)**

- If key already exists:
  - Remove old node
- If cache is full:
  - Remove LRU node (node before tail)
- Insert new node at the front

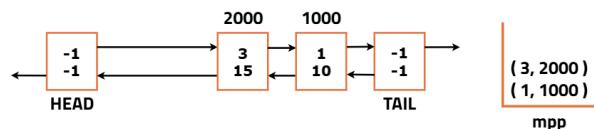
1. LRU cache(3) :



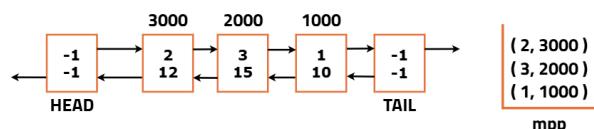
2. Put( 1, 10 ) :



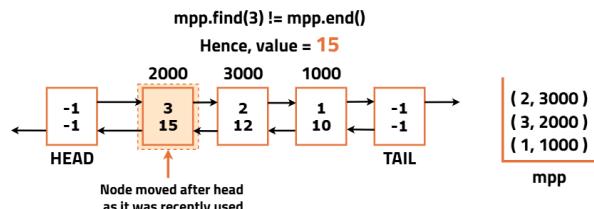
3. Put( 3, 15 ) :



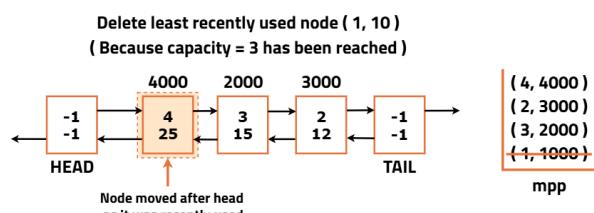
4. Put( 2, 12 ) :



5. Get(3) :



6. Put( 4, 25 ) :



## C++ Implementation

```
#include <bits/stdc++.h>
using namespace std;
```

```

class LRUCache {
public:
 class Node {
public:
 int key, value;
 Node *prev, *next;
 Node(int k, int v) {
 key = k;
 value = v;
 prev = next = nullptr;
 }
 };
 int capacity;
 unordered_map<int, Node*> mp;
 Node *head, *tail;

 LRUCache(int cap) {
 capacity = cap;
 head = new Node(-1, -1);
 tail = new Node(-1, -1);
 head->next = tail;
 tail->prev = head;
 }

 void addNode(Node* node) {
 Node* temp = head->next;
 head->next = node;
 node->prev = head;
 node->next = temp;
 temp->prev = node;
 }
}

```

```

}

void removeNode(Node* node) {
 Node* p = node->prev;
 Node* n = node->next;
 p->next = n;
 n->prev = p;
}

int get(int key) {
 if (mp.find(key) == mp.end())
 return -1;

 Node* node = mp[key];
 removeNode(node);
 addNode(node);
 return node->value;
}

void put(int key, int value) {
 if (mp.find(key) != mp.end()) {
 Node* node = mp[key];
 mp.erase(key);
 removeNode(node);
 }

 if (mp.size() == capacity) {
 Node* lru = tail->prev;
 mp.erase(lru->key);
 removeNode(lru);
 }
}

```

```

 Node* node = new Node(key, value);
 addNode(node);
 mp[key] = node;
 }
};

// Driver code
int main() {
 LRUCache cache(2);

 cache.put(1, 1);
 cache.put(2, 2);
 cout << cache.get(1) << endl; // 1

 cache.put(3, 3);
 cout << cache.get(2) << endl; // -1

 cache.put(4, 4);
 cout << cache.get(1) << endl; // -1
 cout << cache.get(3) << endl; // 3
 cout << cache.get(4) << endl; // 4

 return 0;
}

```

---

- **Time Complexity**

- **get()** → **O(1)**
- **put()** → **O(1)**

- **Space Complexity**

- **O(capacity)**  
(HashMap + Doubly Linked List)

## 31. LFU Cache

You have to design a **Least Frequently Used (LFU) Cache**.  
The cache stores key–value pairs and has a fixed capacity.

Rules of eviction:

- The key with the **lowest frequency of usage** is removed first.
  - If multiple keys have the same frequency, the **least recently used (LRU)** key among them is removed.
  - Every get or put on an existing key increases its frequency by 1.
  - A newly inserted key starts with frequency = 1.
  - Both get and put must work in **O(1) average time**.
- 

### Example Explanation

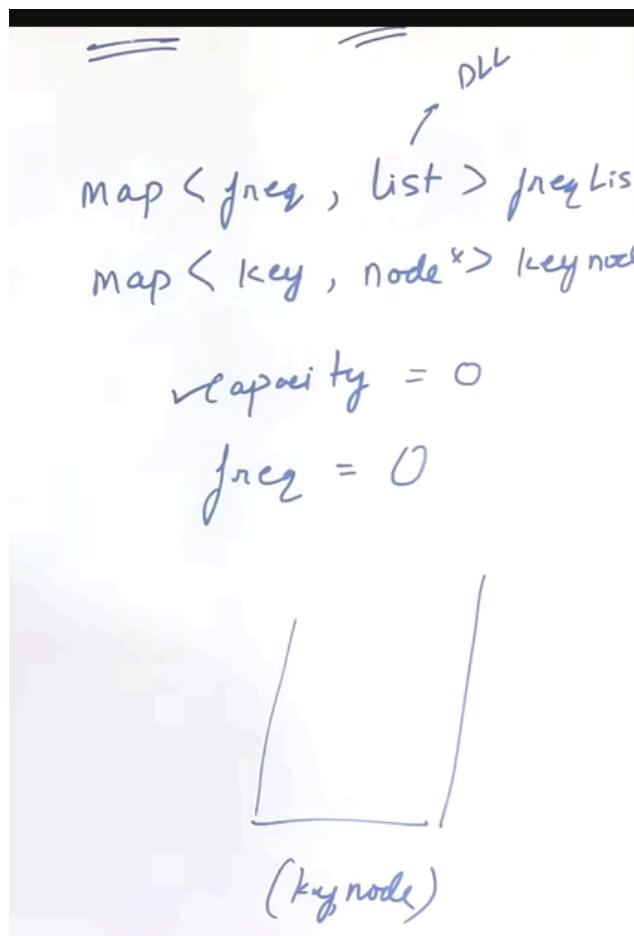
Capacity = 2

Operations:

- $\text{put}(1, 1) \rightarrow \text{cache: } \{1\}, \text{ freq}(1)=1$
- $\text{put}(2, 2) \rightarrow \text{cache: } \{1, 2\}, \text{ freq both } =1$
- $\text{get}(1) \rightarrow \text{return 1, freq}(1)=2$
- $\text{put}(3, 3) \rightarrow \text{cache full, key 2 has lowest freq} \rightarrow \text{evicted}$
- cache becomes  $\{1, 3\}$

This shows:

- LFU key is removed first
- If frequency ties, LRU is removed



---

# Approach

## Algorithm

- Use a **Node** to store key, value, and frequency.
- Use a **doubly linked list** for each frequency to maintain LRU order.
- Maintain:
  - `keyNode` → maps key to its node
  - `freqListMap` → maps frequency to its linked list
- Track:
  - `minFreq` → minimum frequency in cache
  - `curSize` → current cache size

## For `get(key)`:

- If key not found, return -1
- Else:
  - Increase its frequency
  - Move it to the next frequency list
  - Update `minFreq` if needed

## For `put(key, value)`:

- If capacity is 0, do nothing
  - If key exists:
    - Update value
    - Increase frequency
  - Else:
    - If cache is full:
      - Remove LRU node from minFreq list
    - Insert new node with frequency = 1
    - Update minFreq = 1
- 

## Code

```
#include <bits/stdc++.h>
using namespace std;

struct Node {
 int key, value, cnt;
 Node *prev, *next;
 Node(int k, int v) {
 key = k;
 value = v;
 cnt = 1;
 prev = next = NULL;
 }
};
```

```

struct List {
 int size;
 Node *head, *tail;
 List() {
 head = new Node(0, 0);
 tail = new Node(0, 0);
 head->next = tail;
 tail->prev = head;
 size = 0;
 }
 void addFront(Node* node) {
 Node* temp = head->next;
 node->next = temp;
 node->prev = head;
 head->next = node;
 temp->prev = node;
 size++;
 }
 void removeNode(Node* node) {
 node->prev->next = node->next;
 node->next->prev = node->prev;
 size--;
 }
};

class LFUCache {
 int maxSize, curSize, minFreq;
 unordered_map<int, Node*> keyNode;
 unordered_map<int, List*> freqListMap;

public:

```

```

LFUCache(int capacity) {
 maxSize = capacity;
 curSize = 0;
 minFreq = 0;
}

void updateFreq(Node* node) {
 keyNode.erase(node->key);
 freqListMap[node->cnt]->removeNode(node);

 if(node->cnt == minFreq &&
freqListMap[node->cnt]->size == 0)
 minFreq++;

 node->cnt++;
 if(freqListMap.find(node->cnt) == freqListMap.end())
 freqListMap[node->cnt] = new List();

 freqListMap[node->cnt]->addFront(node);
 keyNode[node->key] = node;
}

int get(int key) {
 if(keyNode.find(key) == keyNode.end())
 return -1;

 Node* node = keyNode[key];
 int val = node->value;
 updateFreq(node);
 return val;
}

```

```

void put(int key, int value) {
 if(maxSize == 0) return;

 if(keyNode.find(key) != keyNode.end()) {
 Node* node = keyNode[key];
 node->value = value;
 updateFreq(node);
 } else {
 if(curSize == maxSize) {
 List* list = freqListMap[minFreq];
 keyNode.erase(list->tail->prev->key);
 list->removeNode(list->tail->prev);
 curSize--;
 }

 curSize++;
 minFreq = 1;
 Node* node = new Node(key, value);

 if(freqListMap.find(minFreq) ==
freqListMap.end())
 freqListMap[minFreq] = new List();

 freqListMap[minFreq]->addFront(node);
 keyNode[key] = node;
 }
}

int main() {
 LFUCache cache(2);
 cache.put(1,1);
}

```

```
cache.put(2,2);
cout << cache.get(1) << " ";
cache.put(3,3);
cout << cache.get(2) << " ";
cout << cache.get(3) << " ";
cache.put(4,4);
cout << cache.get(1) << " ";
cout << cache.get(3) << " ";
cout << cache.get(4);
return 0;
}
```

---

## Complexity Analysis

- **Time Complexity:**

get → O(1) average

put → O(1) average

- **Space Complexity:**

O(capacity), as the cache stores at most capacity elements

# **Sliding Window and Two Pointer Combined Problems**

# 1. Length of Longest Substring without any Repeating Character

You are given a string S.

Your task is to find the **length of the longest substring** that contains **no repeating characters**.

A substring must be **continuous**, and all characters inside it must be **distinct**.

---

## Example Explanation

### Example 1

Input: S = "abcddabac"

The substring "abcd" has all unique characters and length 4.

No longer substring without repetition exists.

Output: 4

### Example 2

Input: S = "aaabbccccc"

Possible longest substrings without repeating characters are "ab" and "bc", both of length 2.

Output: 2

---

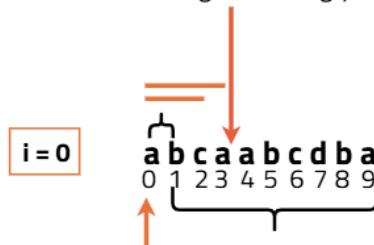
## Approach 1 (Brute Force)

### Algorithm

- Consider every possible starting index of a substring.
- For each starting index, extend the substring character by character.
- Use a hash array to track whether a character has already appeared.
- If a repeated character is found, stop extending the substring.
- Keep updating the maximum length found.
- Return the maximum length at the end.

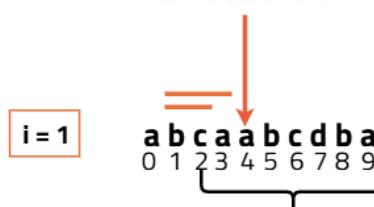


At this position, when hashset is checked and we can find a is already present.  
So, 3 will be the maximum length of string yet



We will iterate for all elements and generate  
all substrings possible by beginning with that element

Same case here



Similarly, we will run more number of iterations  
to get the desired answer

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int longestNonRepeatingSubstring(string &s) {
 int n = s.size();
 int maxLen = 0;

 for(int i = 0; i < n; i++) {
 vector<int> hash(256, 0);
 for(int j = i; j < n; j++) {
 if(hash[s[j]] == 1) break;
 hash[s[j]] = 1;
 int len = j - i + 1;
 maxLen = max(maxLen, len);
 }
 }
 }
}
```

```

 return maxLen;
 }
};

int main() {
 string s = "abcddabac";
 Solution sol;
 cout << sol.longestNonRepeatingSubstring(s);
 return 0;
}

```

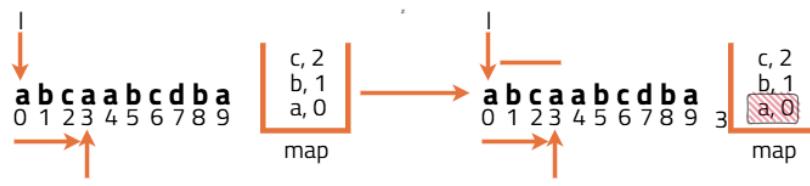
## Complexity Analysis

- **Time Complexity:**  $O(n^2)$   
Nested loops check all possible substrings.
  - **Space Complexity:**  $O(1)$   
Fixed-size hash array of 256 characters.
- 

## Approach 2 (Optimal – Sliding Window)

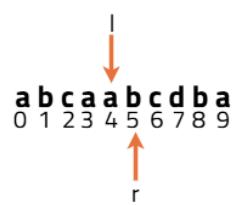
### Algorithm

- Use two pointers  $l$  (left) and  $r$  (right) to maintain a window of unique characters.
- Maintain a hash array storing the **last index** of each character.
- Move the right pointer forward.
- If a character is repeated, move the left pointer just after its last occurrence.
- Update the maximum window length.
- Continue until the end of the string.

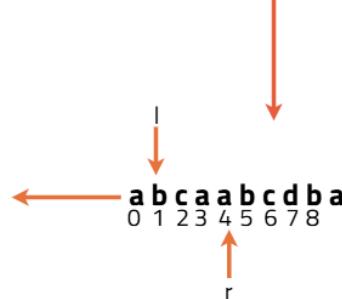


Since 'a' is found in the map, we will update left to 1 index and update the index of a in map as 3.

len = 3



Since 'b' is found at index 5 and left is on 4 it means no 'b' is present in current substring.  
So update b value in map as 5.



'a' again exists in the map, 'a' is initially present at index 3 so update left to index 4.

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int longestNonRepeatingSubstring(string& s) {
 int n = s.size();
 int hash[256];
 for(int i = 0; i < 256; i++) hash[i] = -1;

 int l = 0, r = 0, maxLen = 0;

 while(r < n) {
 if(hash[s[r]] != -1) {
 l = max(l, hash[s[r]] + 1);
 }
 int len = r - l + 1;
 maxLen = max(maxLen, len);
 hash[s[r]] = r;
 r++;
 }
 return maxLen;
 }
}
```

```

 hash[s[r]] = r;
 r++;
 }
 return maxLen;
}

int main() {
 string s = "abcdabac";
 Solution sol;
 cout << sol.longestNonRepeatingSubstring(s);
 return 0;
}

```

## Complexity Analysis

- **Time Complexity:**  $O(n)$   
Each character is processed once using two pointers.
- **Space Complexity:**  $O(1)$   
Fixed-size hash array of 256 characters.

## 2. Max Consecutive Ones III

You are given a binary array `nums` (containing only 0 and 1) and an integer `k`.

You are allowed to flip **at most k zeros into ones**.

Your task is to return the **maximum number of consecutive 1s** that can be obtained after performing at most `k` flips.

---

## Example Explanation

### Example 1

Input:

nums = [1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 0], k = 3

By flipping the zeros at indices 3, 4, 5, the array becomes:

[1, 1, 1, 1, 1, 1, 1, 1, 1, 0]

The longest consecutive sequence of 1s is 10.

Output: 10

---

### Example 2

Input:

nums = [0, 0, 1, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1], k = 3

After flipping suitable zeros, the longest consecutive sequence of 1s is 9.

Output: 9

---

## Approach 1 (Brute Force)

### Algorithm

- Try every possible subarray.
- For each subarray, count how many zeros it contains.
- If the number of zeros is less than or equal to k, the subarray is valid.
- Track the maximum length among all valid subarrays.
- Return the maximum length found.

### Code

```
#include <bits/stdc++.h>
using namespace std;
```

```

class Solution {
public:
 int longestOnes(vector<int>& nums, int k) {
 int n = nums.size();
 int maxLen = 0;

 for(int i = 0; i < n; i++) {
 int zeros = 0;
 for(int j = i; j < n; j++) {
 if(nums[j] == 0) zeros++;
 if(zeros > k) break;
 maxLen = max(maxLen, j - i + 1);
 }
 }
 return maxLen;
 }
};

int main() {
 vector<int> nums = {1,1,1,0,0,0,1,1,1,1,0};
 int k = 3;
 Solution sol;
 cout << sol.longestOnes(nums, k);
 return 0;
}

```

## Complexity Analysis

- **Time Complexity:**  $O(n^2)$  due to nested loops.
  - **Space Complexity:**  $O(1)$  since no extra space is used.
-

## Approach 2 (Better – Sliding Window)

### Algorithm

- Use a sliding window with two pointers `left` and `right`.
- Expand the window using `right`.
- Count zeros inside the window.
- If zeros exceed `k`, shrink the window from the left.
- Update the maximum window size after each step.
- Return the maximum length found.

**Right pointer ek step me sirf 1 zero add kar sakta hai.**

So:

- zerocount **maximum 1 se hi badh sakta hai**
- Isliye window **sirf 1 step invalid hoti hai**
- Usko theek karne ke liye **1 step shrink bhi kaafi hota hai**

### Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int longestOnes(vector<int>& nums, int k) {
 int left = 0, zeros = 0, maxLen = 0;

 for(int right = 0; right < nums.size(); right++) {
 if(nums[right] == 0) zeros++;

 while(zeros > k) {
```

```

 if(nums[left] == 0) zeros--;
 left++;
 }
 maxLen = max(maxLen, right - left + 1);
}
return maxLen;
}

int main() {
 vector<int> nums = {1,1,1,0,0,0,1,1,1,1,0};
 int k = 3;
 Solution sol;
 cout << sol.longestOnes(nums, k);
 return 0;
}

```

## Complexity Analysis

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

## Approach 3 (Optimal – Optimized Sliding Window)

### Algorithm

- Use two pointers `left` and `right`.
- Maintain a count of zeros in the current window.
- Move `right` forward each step.
- If zero count exceeds `k`, move `left` forward once and adjust zero count.

- Update maximum window length at each step.
- This avoids an inner loop and keeps logic simple.

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int longestOnes(vector<int>& nums, int k) {
 int left = 0, zeroCount = 0, maxLen = 0;

 for(int right = 0; right < nums.size(); right++) {
 if(nums[right] == 0) zeroCount++;

 if(zeroCount > k) {
 if(nums[left] == 0) zeroCount--;
 left++;
 }
 maxLen = max(maxLen, right - left + 1);
 }
 return maxLen;
 }
};

int main() {
 vector<int> nums = {1,1,1,0,0,0,1,1,1,1,0};
 int k = 3;
 Solution sol;
 cout << sol.longestOnes(nums, k);
 return 0;
}
```

- **Time Complexity:**  $O(n)$

- **Space Complexity:**  $O(1)$

Doubt clear karne k liye example: `nums = [1,1,1,0,0,0,1]`  
`k = 2`

window thodi der invalid reh sakti hai,  
lekin wo kabhi answer ko bigaadti nahi.

## 3. Fruit Into Baskets

You are given an array `fruits` where `fruits[i]` represents the type of fruit on the  $i$ -th tree.  
You have **only two baskets**, and **each basket can store only one type of fruit**, but in unlimited quantity.

You can start picking fruits from **any tree**, but once you start, you must move **only to the right**, picking **exactly one fruit from each tree**.

If you encounter a fruit type that cannot fit into your two baskets, you must stop.

Your task is to return the **maximum number of fruits** you can collect following these rules.

---

### Example Explanation

#### Example 1

Input: `fruits = [1, 2, 1]`

We start from index 0:

- Basket 1 → fruit 1
- Basket 2 → fruit 2
- Next fruit is 1, which already exists in basket

Total fruits collected = 3

Output: 3

---

### Example 2

Input: fruits = [1, 2, 3, 2, 2]

Best starting point is index 1:

- Fruits collected: [2, 3, 2, 2]
- Only two types (2 and 3)

Total fruits collected = 4

Output: 4

---

## Approach 1 (Brute Force)

### Algorithm

- Try starting from every index.
- From each start index, move right and collect fruits.
- Keep track of fruit types using a map or set.
- Stop when more than 2 fruit types are encountered.
- Track the maximum number of fruits collected.

### Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
```

```

public:
 int totalFruit(vector<int>& fruits) {
 int n = fruits.size();
 int maxFruits = 0;

 for(int start = 0; start < n; start++) {
 unordered_map<int,int> basket;
 int count = 0;

 for(int end = start; end < n; end++) {
 basket[fruits[end]]++;
 if(basket.size() > 2) break;
 count++;
 }
 maxFruits = max(maxFruits, count);
 }
 return maxFruits;
 }
};

int main() {
 vector<int> fruits = {1,2,1};
 Solution sol;
 cout << sol.totalFruit(fruits);
 return 0;
}

```

## Complexity Analysis

- **Time Complexity:**  $O(n^2)$
  - **Space Complexity:**  $O(1)$  (at most 3 fruit types in map)
- 

## Approach 2 (Better – Sliding Window with Hash Map)

## Algorithm

- Use a sliding window with two pointers `left` and `right`.
- Maintain a hash map to count fruit types in the current window.
- Expand window to the right.
- If fruit types exceed 2, shrink window from the left.
- Track the maximum window size.

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int totalFruit(vector<int>& fruits) {
 unordered_map<int, int> basket;
 int left = 0, maxFruits = 0;

 for(int right = 0; right < fruits.size(); right++) {
 basket[fruits[right]]++;

 while(basket.size() > 2) {
 basket[fruits[left]]--;
 if(basket[fruits[left]] == 0)
 basket.erase(fruits[left]);
 left++;
 }
 maxFruits = max(maxFruits, right - left + 1);
 }
 return maxFruits;
 }
};

int main() {
 vector<int> fruits = {1,2,3,2,2};
```

```
Solution sol;
cout << sol.totalFruit(fruits);
return 0;
}
```

## Complexity Analysis

- **Time Complexity:**  $O(n)$
  - **Space Complexity:**  $O(1)$  (map holds at most 2 keys)
- 

## Approach 3 (Optimal – Without Hash Map)

### Algorithm

- Track only the **last two fruit types**.
- Maintain:
  - `lastFruit`
  - `secondLastFruit`
  - `lastFruitCount`
  - `currentWindowSize`
- If current fruit matches one of the two, extend window.
- Otherwise, shrink window to last fruit streak + current fruit.
- Update maximum length.

### Code

```
#include <bits/stdc++.h>
using namespace std;
```

```

class Solution {
public:
 int totalFruit(vector<int>& fruits) {
 int lastFruit = -1, secondLastFruit = -1;
 int lastCount = 0, currLen = 0, maxLen = 0;

 for(int fruit : fruits) {
 if(fruit == lastFruit || fruit == secondLastFruit)
 currLen++;
 else
 currLen = lastCount + 1;

 if(fruit == lastFruit)
 lastCount++;
 else {
 lastCount = 1;
 secondLastFruit = lastFruit;
 lastFruit = fruit;
 }

 maxLen = max(maxLen, currLen);
 }
 return maxLen;
 }
};

int main() {
 vector<int> fruits = {1,2,1,2,3};
 Solution sol;
 cout << sol.totalFruit(fruits);
 return 0;
}

```

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

# 4. Longest Repeating Character Replacement

You are given a string  $s$  consisting of **uppercase English letters** and an integer  $k$ .

You are allowed to **replace at most  $k$  characters** in the string with any other uppercase letter.

After performing at most  $k$  replacements, you need to find the **length of the longest substring that contains only one repeating character**.

---

## Example Explanation

### Example 1

Input:  $s = \text{"BAABAABBAA"}$ ,  $k = 2$

We can replace:

- B at index 0 → A
- B at index 3 → A

New string becomes:

AAAAAABBAA

The longest substring with the same character is "AAAAAA"

Length = 6

Output: 6

---

### Example 2

Input:  $s = \text{"AABABBA"}$ ,  $k = 1$

We can replace one character to form "AABBBA"

The longest repeating substring is "BBBB"

Output: 4

---

# Brute Force Approach

## Algorithm

- Try all possible substrings.
- For each substring:
  - Count frequency of characters.
  - Find the most frequent character.
  - Calculate replacements needed = substring length - max frequency.
- If replacements  $\leq k$ , update the answer.
- Return the maximum valid length found.

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int characterReplacement(string s, int k) {
 int n = s.size();
 int maxLength = 0;

 for(int i = 0; i < n; i++) {
 vector<int> freq(26, 0);
 int maxFreq = 0;

 for(int j = i; j < n; j++) {
 freq[s[j] - 'A']++;
 maxFreq = max(maxFreq, freq[s[j] - 'A']);
 int windowLen = j - i + 1;
```

```

 if(windowLen - maxFreq <= k) {
 maxLength = max(maxLength, windowLen);
 }
 }
 return maxLength;
}
};

int main() {
 Solution sol;
 string s = "AABABBA";
 int k = 1;
 cout << sol.characterReplacement(s, k);
 return 0;
}

```

## Complexity Analysis

- **Time Complexity:**  $O(n^2)$
  - **Space Complexity:**  $O(1)$  (fixed 26-size frequency array)
- 

## Better Approach (Sliding Window)

### Algorithm

- Use two pointers `left` and `right` to form a window.
- Maintain frequency of characters inside the window.
- Track the **maximum frequency** character in the window.
- If  $(\text{window size} - \text{max frequency}) > k$ , shrink window from left.
- Keep updating the maximum window size.

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int characterReplacement(string s, int k) {
 unordered_map<char,int> freq;
 int left = 0, maxFreq = 0, maxLen = 0;

 for(int right = 0; right < s.size(); right++) {
 freq[s[right]]++;
 maxFreq = max(maxFreq, freq[s[right]]);

 while((right - left + 1) - maxFreq > k) {
 freq[s[left]]--;
 left++;
 }

 maxLen = max(maxLen, right - left + 1);
 }
 return maxLen;
 }
};

int main() {
 Solution sol;
 string s = "AABABBA";
 int k = 1;
 cout << sol.characterReplacement(s, k);
 return 0;
}
```

## Complexity Analysis

- **Time Complexity:**  $O(n)$

- **Space Complexity:**  $O(1)$  (at most 26 characters)
- 

## Optimal Approach (Optimized Sliding Window)

### Algorithm

- Use a **fixed-size frequency array (26)** instead of a map.
- Maintain **maxCount** = highest frequency in the window.
- Expand the window to the right.
- If replacements needed exceed k, move left pointer.
- We **do not reduce maxCount** while shrinking, because it does not affect correctness.

### Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int characterReplacement(string s, int k) {
 vector<int> freq(26, 0);
 int left = 0, maxCount = 0, maxLen = 0;

 for(int right = 0; right < s.size(); right++) {
 freq[s[right] - 'A']++;
 maxCount = max(maxCount, freq[s[right] - 'A']);

 if((right - left + 1) - maxCount > k) {
 freq[s[left] - 'A']--;
 left++;
 }
 maxLen = max(maxLen, right - left + 1);
 }
 }
}
```

```

 return maxLen;
 }
};

int main() {
 Solution sol;
 string s = "AABABBA";
 int k = 1;
 cout << sol.characterReplacement(s, k);
 return 0;
}

```

## Complexity Analysis

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

# 5. Binary Subarray With Sum

You are given a **binary array** `nums` (containing only 0 and 1) and an integer `goal`.

You need to count how many **non-empty contiguous subarrays** have a sum **exactly equal to goal**.

A subarray means a continuous part of the array.

---

## Example Explanation

### Example 1

Input: `nums = [1, 0, 0, 1, 1, 0]`, `goal = 2`

Valid subarrays with sum = 2 are:

- [1, 0, 0, 1]
- [0, 0, 1, 1]
- [0, 1, 1]
- [1, 1]
- [1, 1, 0]
- [0, 0, 1, 1, 0]

Total = **6**

Output: 6

---

### Example 2

Input: nums = [0, 0, 0, 0, 0, 0], goal = 0

Every possible subarray has sum 0.

Number of subarrays =  $n(n+1)/2 = 6*7/2 = 21$

Output: 21

---

## Brute Force Approach

### Algorithm

- Try all possible subarrays using two loops.
- Fix a starting index.
- Expand the ending index and keep adding elements to a running sum.
- If the sum equals goal, increase the count.

- After checking all subarrays, return the count.

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int numSubarraysWithSum(vector<int>& nums, int goal) {
 int count = 0;
 int n = nums.size();

 for(int i = 0; i < n; i++) {
 int sum = 0;
 for(int j = i; j < n; j++) {
 sum += nums[j];
 if(sum == goal) {
 count++;
 }
 }
 }
 return count;
 }
};

int main() {
 Solution sol;
 vector<int> nums = {1,0,1,0,1};
 int goal = 2;
 cout << sol.numSubarraysWithSum(nums, goal);
 return 0;
}
```

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

---

## Better Approach (Prefix Sum + Hash Map)

### Algorithm

- Use a prefix sum technique.
- Maintain a hashmap that stores how many times a prefix sum has appeared.
- At any index, if  $(\text{currentSum} - \text{goal})$  exists in the map, it means valid subarrays end here.
- Add its frequency to the answer.
- Update the map with the current prefix sum.

### Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int numSubarraysWithSum(vector<int>& nums, int goal) {
 unordered_map<int,int> mp;
 mp[0] = 1;

 int sum = 0, count = 0;

 for(int x : nums) {
 sum += x;
 if(mp.find(sum - goal) != mp.end()) {
 count += mp[sum - goal];
 }
 mp[sum]++;
 }
 return count;
 }
};
```

```

int main() {
 Solution sol;
 vector<int> nums = {1, 0, 1, 0, 1};
 int goal = 2;
 cout << sol.numSubarraysWithSum(nums, goal);
 return 0;
}

```

## Complexity Analysis

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

## Optimal Approach (Sliding Window using AtMost Trick)

This approach works because the array contains **only non-negative numbers (0 and 1)**.

- Count subarrays with sum **at most goal**
- Count subarrays with sum **at most goal - 1**
- Subtract them to get subarrays with sum **exactly goal**

Why it works:

- Subarrays with exact sum = goal
- = subarrays with sum  $\leq$  goal minus subarrays with sum  $\leq$  goal-1

Steps:

- Use a sliding window with two pointers.
- Expand right pointer and add to sum.
- If sum exceeds limit, shrink from left.
- Add window size to count.
- Final answer =  $\text{atMost}(\text{goal}) - \text{atMost}(\text{goal}-1)$

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int numSubarraysWithSum(vector<int>& nums, int goal) {
 return atMost(nums, goal) - atMost(nums, goal - 1);
 }

private:
 int atMost(vector<int>& nums, int k) {
 if(k < 0) return 0;

 int left = 0, sum = 0, count = 0;

 for(int right = 0; right < nums.size(); right++) {
 sum += nums[right];

 while(sum > k) {
 sum -= nums[left];
 left++;
 }
 count += (right - left + 1);
 }
 return count;
 }
};

int main() {
 Solution sol;
 vector<int> nums = {1,0,1,0,1};
 int goal = 2;
 cout << sol.numSubarraysWithSum(nums, goal);
 return 0;
}
```

## Complexity Analysis

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

## WHY (**right - left + 1**) ?

- Window [left ... right] valid hai (sum  $\leq k$ )

Toh:

- [right ... right]
- [right-1 ... right]
- [right-2 ... right]
- ...
- [left ... right]

**Total = right - left + 1 subarrays**

Aur **sabka sum  $\leq k$  hoga**

## 6. Count Number of Nice Subarrays

You are given an integer array `nums` and an integer `k`.

A subarray is called **nice** if it contains **exactly k odd numbers**.

Your task is to count the number of such nice subarrays.

A subarray is a **continuous** part of the array.

---

## Example Explanation

### Example 1

Input: `nums = [1, 1, 2, 1, 1], k = 3`

Odd numbers appear at indices: 0, 1, 3, 4

Nice subarrays with exactly 3 odd numbers:

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

Output: 2

---

### Example 2

Input: `nums = [4, 8, 2], k = 1`

There are no odd numbers in the array, so no subarray can contain 1 odd number.

Output: 0

---

## Brute Force Approach

### Algorithm

- Try every possible subarray.
- For each starting index, extend the subarray to the right.
- Count how many odd numbers are in the current subarray.
- If odd count becomes greater than k, stop expanding.

- If odd count is exactly k, count it as a nice subarray.
- Return the total count.

## Code

```
#include <bits/stdc++.h>

using namespace std;

class Solution {

public:

 int numberOfSubarrays(vector<int>& nums, int k) {

 int count = 0;

 for(int start = 0; start < nums.size(); start++) {

 int oddCount = 0;

 for(int end = start; end < nums.size(); end++) {

 if(nums[end] % 2 != 0)

 oddCount++;

 if(oddCount > k)

 break;

 }

 if(oddCount == k)

 count++;

 }

 return count;

 }

}
```

```

 count++;

 }

 return count;
}

};

int main() {
 Solution sol;

 vector<int> nums = {1,1,2,1,1};

 int k = 3;

 cout << sol.numberOfSubarrays(nums, k);

 return 0;
}

```

## Complexity Analysis

- **Time Complexity:**  $O(N^2)$
- **Space Complexity:**  $O(1)$

## Better Approach (Prefix Odd Count + Hash Map)

- Convert the problem into counting prefixes.

- Maintain a running count of odd numbers seen so far.
- Use a hashmap where:
  - key = number of odd numbers seen
  - value = how many times this count has occurred
- For each index:
  - If current odd count is oddCount
  - Any previous prefix with oddCount - k forms a nice subarray
- Add its frequency to the answer.
- Update the map.

## Code

```
#include <bits/stdc++.h>

using namespace std;

class Solution {

public:

 int numberOfSubarrays(vector<int>& nums, int k) {

 unordered_map<int,int> freq;
 freq[0] = 1;

 int oddCount = 0;
 int result = 0;

 for(int i=0; i<nums.size(); i++) {
 if(nums[i]<0)
 oddCount++;
 freq[oddCount]++;
 if(oddCount == k)
 result += freq[0];
 else if(oddCount > k)
 result += freq[oddCount-k];
 }
 }
}
```

```

 for(int num : nums) {
 if(num % 2 != 0)
 oddCount++;
 if(freq.count(oddCount - k))
 result += freq[oddCount - k];
 freq[oddCount]++;
 }
 return result;
 }

};

int main() {
 Solution sol;
 vector<int> nums = {1,1,2,1,1};
 int k = 3;
 cout << sol.numberOfSubarrays(nums, k);
 return 0;
}

```

## Complexity Analysis

- **Time Complexity:**  $O(N)$
  - **Space Complexity:**  $O(N)$
- 

## Optimal Approach (Sliding Window – At Most Trick)

### Algorithm

- We cannot directly count subarrays with exactly  $k$  odd numbers using one sliding window.
- Instead:
  - Count subarrays with **at most  $k$  odd numbers**
  - Count subarrays with **at most  $k-1$  odd numbers**
- The difference gives subarrays with **exactly  $k$  odd numbers**.

Formula:

```
exactlyK = atMost(k) - atMost(k-1)
```

Helper function `atMost(k)`:

- Use sliding window.
- Expand right pointer.
- Reduce  $k$  when encountering odd numbers.
- Shrink window when  $k < 0$ .
- Add (`right - left + 1`) to result.

## Code

```
#include <bits/stdc++.h>

using namespace std;

class Solution {

public:

 int countAtMost(vector<int>& nums, int k) {

 int left = 0, res = 0;

 for(int right = 0; right < nums.size(); right++) {

 if(nums[right] % 2 != 0)

 K--;

 // K is allowance

 while(k < 0) {

 if(nums[left] % 2 != 0)

 k++;

 left++;

 }

 res += (right - left + 1);

 }

 return res;

 }

}
```

```

int numberOfSubarrays(vector<int>& nums, int k) {
 return countAtMost(nums, k) - countAtMost(nums, k - 1);
}

};

int main() {
 Solution sol;
 vector<int> nums = {1,1,2,1,1};
 int k = 3;
 cout << sol.numberOfSubarrays(nums, k);
 return 0;
}

```

- **Time Complexity:**  $O(N)$
- **Space Complexity:**  $O(1)$

## 7. Number of Substrings Containing All Three Characters

You are given a string  $s$  that contains only the characters '`a`', '`b`', and '`c`'.

Your task is to count how many **substrings** of  $s$  contain **at least one occurrence of all three characters** '`a`', '`b`', and '`c`'.

A substring is a **continuous** part of the string.

---

## Example Explanation

### Example 1

Input: s = "abcba"

Valid substrings that contain 'a', 'b', and 'c':

- "abc"
- "abcb"
- "abcba"
- "bcba"
- "cba"

Output: 5

---

### Example 2

Input: s = "ccabcc"

Valid substrings:

- "ccab"
- "ccabc"
- "ccabcc"
- "cab"
- "cabc"
- "cabcc"
- "abc"

- "abcc"

Output: 8

---

## Brute Force Approach

### Algorithm

- Try all possible substrings.
- Fix a starting index  $i$ .
- From  $i$ , expand the substring to the right using index  $j$ .
- Maintain a frequency array for characters 'a', 'b', 'c'.
- If at any point all three characters are present, count that substring.
- Continue until all substrings are checked.

### Code

```
#include <bits/stdc++.h>

using namespace std;

class Solution {

public:

 int numberOfSubstrings(string s) {

 int n = s.size();

 int count = 0;
```

```

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

 vector<int> freq(3, 0);

 for(int j = i; j < n; j++) {

 freq[s[j] - 'a']++;

 if(freq[0] > 0 && freq[1] > 0 && freq[2] > 0)

 count++;

 }

 }

 return count;
 }
}

```

```

int main() {

 Solution sol;

 string s = "abcba";

 cout << sol.numberOfSubstrings(s);

 return 0;
}

```

- **Time Complexity:**  $O(N^2)$

All possible substrings are checked using two nested loops.

- **Space Complexity:**  $O(1)$   
Only a fixed-size frequency array of size 3 is used.
- 

## Optimal Approach (Sliding Window)

### Algorithm

- Use a sliding window `[left, right]`.
- Maintain a frequency array for '`a`', '`b`', '`c`'.
- Expand the window using `right`.
- Once the window contains all three characters:
  - All substrings that start at `left` or before and end at `right` are valid.
  - Add  $(n - right)$  to the result.
- Shrink the window from the left and continue.
- This avoids checking every substring explicitly.

Jaise hi current window valid ho jaati hai (`a`, `b`, `c` present),

to us window se start hone wale aur right side tak extend hone wale  
saare substrings automatically valid hote hain.

### Code

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
```

```

public:

 int numberOfSubstrings(string s) {

 int n = s.size();

 vector<int> freq(3, 0);

 int left = 0, result = 0;

 for(int right = 0; right < n; right++) {

 freq[s[right] - 'a']++;

 while(freq[0] > 0 && freq[1] > 0 && freq[2] > 0) {

 result += (n - right); //left - right to left - n

 freq[s[left] - 'a']--;

 left++;

 }

 }

 return result;

 };

}

int main() {

 Solution sol;

 string s = "abcba";

 cout << sol.numberOfSubstrings(s);

```

```
 return 0;
}

}
```

## Complexity Analysis

- **Time Complexity:**  $O(N)$   
Each character is processed at most twice (once by right, once by left).
- **Space Complexity:**  $O(1)$   
Constant extra space for the frequency array.

# 8. Maximum Points You Can Obtain from Cards

You are given an array `cardScore` representing scores of cards placed in a row.

You must choose **exactly k cards**.

At every step, you are allowed to pick **one card either from the beginning or from the end** of the row.

Your task is to return the **maximum total score** you can obtain by picking exactly  $k$  cards following these rules.

---

## Example Explanation

### Example 1

Input: `cardScore = [1, 2, 3, 4, 5, 6], k = 3`

If we choose the last three cards:  $4 + 5 + 6 = 15$ , which is the maximum possible score.

Output: 15

## Example 2

Input: cardScore = [5, 4, 1, 8, 7, 1, 3], k = 3

Best choice:

- Take 5 from start
- Take 4 from start
- Take 3 from end

Total score = 5 + 4 + 3 = 12

Output: 12

---

## Brute Force Approach

### Algorithm

We must pick exactly k cards, but they can be taken from either end.

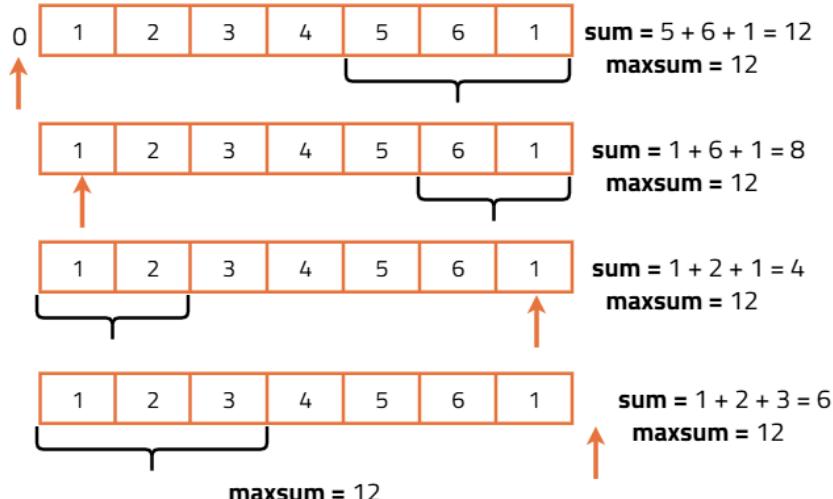
So we try **all possible combinations**:

- Take i cards from the start
- Take k - i cards from the end  
for all i from 0 to k.

Steps:

1. Loop i from 0 to k
  2. Sum first i cards from the beginning
  3. Sum last k - i cards from the end
  4. Track the maximum sum obtained
-

**cards =**  **k = 3**



## Code

```
#include <bits/stdc++.h>

using namespace std;

class Solution {

public:

 int maxScore(vector<int>& cardPoints, int k) {

 int n = cardPoints.size();

 int maxSum = 0;

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

 int currSum = 0;
```

```

 // Take i cards from the front

 for(int j = 0; j < i; j++)

 currSum += cardPoints[j];

 // Take (k - i) cards from the back

 for(int j = 0; j < k - i; j++)

 currSum += cardPoints[n - 1 - j];

 maxSum = max(maxSum, currSum);

 }

 return maxSum;

}

};

int main() {

 Solution sol;

 vector<int> cards = {1, 2, 3, 4, 5, 6};

 int k = 3;

 cout << sol.maxScore(cards, k);

 return 0;

}

```

---

## Complexity Analysis

- **Time Complexity:**  $O(k^2)$   
For each of  $k$  combinations, we sum up to  $k$  elements.
  - **Space Complexity:**  $O(1)$   
Only constant extra variables are used.
- 

## Optimal Approach

### Algorithm

Instead of recalculating sums every time, we use a **sliding window idea**.

Key observation:

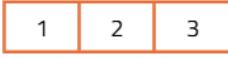
If you pick  $k$  cards from the ends, it is equivalent to **removing a contiguous subarray of length  $n - k$  from the middle** and keeping the rest.

But since the given approach uses front/back shifting, we follow that logic:

Steps:

1. Take all  $k$  cards from the **front** and compute the initial sum.
  2. Then, step by step:
    - Remove one card from the front
    - Add one card from the back
  3. Update the maximum score at each step.
  4. Do this  $k$  times.
-

**cardspoints** =  **k** = 3

 **total** =  $1 + 2 + 3 = 6$  **maxpoints** = 6

 **Remove** : **cardpoints[2]** = 3  
**add** : **cardpoints[6]** = 1

**total** =  $6 - 3 + 1 = 4$  **maxpoints** = 6

 **Remove** : **cardpoints[1]** = 2  
**add** : **cardpoints[5]** = 6

**total** =  $4 - 2 + 6 = 8$  **maxpoints** = 8

 **Remove** : **cardpoints[0]** = 1  
**add** : **cardpoints[4]** = 5

**total** =  $8 - 1 + 5 = 12$  **maxpoints** = 12

**maxpoints** = 12

## Code

```
#include <bits/stdc++.h>

using namespace std;

class Solution {

public:

 int maxScore(vector<int>& cardPoints, int k) {

 int n = cardPoints.size();

 int currSum = 0;

 // Sum of first k cards

 for(int i = 0; i < k; i++)
 currSum += cardPoints[i];
 int totalSum = currSum;
 int maxpoints = currSum;

 for(int i = k; i < n; i++) {
 currSum -= cardPoints[i-k];
 currSum += cardPoints[i];
 if(currSum > maxpoints)
 maxpoints = currSum;
 }
 return maxpoints;
 }
}
```

```

 currSum += cardPoints[i];

 int maxSum = currSum;

 // Shift window: remove from front, add from back

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

 currSum -= cardPoints[k - 1 - i];
 currSum += cardPoints[n - 1 - i];
 maxSum = max(maxSum, currSum);
 }

 return maxSum;
}

};

int main() {
 Solution sol;
 vector<int> cards = {5, 4, 1, 8, 7, 1, 3};
 int k = 3;
 cout << sol.maxScore(cards, k);
 return 0;
}

```

---

## Complexity Analysis

- **Time Complexity:**  $O(k)$   
Initial sum takes  $k$  steps, and sliding takes  $k$  steps.
- **Space Complexity:**  $O(1)$   
Only constant extra variables are used.

# 9. Longest Substring with At Most K Distinct Characters

You are given a string  $s$  and an integer  $k$ .

Your task is to find the **length of the longest substring** that contains **at most  $k$  distinct characters**.

A substring must be **continuous**.

---

## Example Explanation

### Example 1

Input:  $s = "aababbcaacc"$ ,  $k = 2$

The substring "aababb" contains only two distinct characters (a and b) and has length 6, which is the maximum possible.

Output: 6

### Example 2

Input:  $s = "abcdefg"$ ,  $k = 3$

The substring "bcd" contains at most three distinct characters (b, c, d) and has length 4.

Output: 4

---

# Brute Force Approach

## Algorithm

In this approach, we try **all possible substrings** and check whether each substring contains **at most k distinct characters**.

Steps:

1. Fix a starting index  $i$ .
  2. From  $i$ , expand the substring to the right using index  $j$ .
  3. Use a frequency map to track distinct characters in the current substring.
  4. If the number of distinct characters exceeds  $k$ , stop expanding.
  5. If valid, update the maximum length.
  6. Repeat for all starting indices.
- 

## Code

```
#include <bits/stdc++.h>

using namespace std;

class Solution {

public:

 int lengthOfLongestSubstringKDistinct(string s, int k) {

 int maxLength = 0;

 for (int i = 0; i < s.size(); i++) {
```

```

unordered_map<char, int> freq;

for (int j = i; j < s.size(); j++) {
 freq[s[j]]++;

 if (freq.size() > k) break;

 maxLength = max(maxLength, j - i + 1);
}

return maxLength;
};

int main() {
 Solution sol;
 string s = "aababbcaacc";
 int k = 2;
 cout << sol.lengthOfLongestSubstringKDistinct(s, k);
 return 0;
}

```

---

## Complexity Analysis

- **Time Complexity:**  $O(n^2)$   
We try all possible substrings using two nested loops.
  - **Space Complexity:**  $O(k)$   
The frequency map stores at most  $k$  distinct characters.
- 

## Optimal Approach

### Algorithm

Instead of checking all substrings, we use a **sliding window**.

Idea:

- Maintain a window  $[left, right]$  with **at most  $k$  distinct characters**.
- Expand the window by moving `right`.
- If distinct characters exceed  $k$ , shrink the window by moving `left`.
- Track the maximum valid window length.

Steps:

1. Initialize two pointers `left = 0, right = 0`.
  2. Use a frequency map to store character counts.
  3. Expand the window by moving `right`.
  4. If distinct characters exceed  $k$ , shrink from the left.
  5. Update the maximum length for each valid window.
-

## Code

```
#include <bits/stdc++.h>

using namespace std;

class Solution {

public:

 int lengthOfLongestSubstringKDistinct(string s, int k) {

 if (k == 0 || s.empty()) return 0;

 unordered_map<char, int> freq;

 int left = 0, maxLen = 0;

 for (int right = 0; right < s.size(); right++) {

 freq[s[right]]++;

 while (freq.size() > k) {

 freq[s[left]]--;
 if (freq[s[left]] == 0)
 freq.erase(s[left]);
 left++;
 }

 maxLen = max(maxLen, right - left + 1);
 }
 }
}
```

```

 }

 return maxLen;
 }

};

int main() {
 Solution sol;
 string s = "aababbcaacc";
 int k = 2;
 cout << sol.lengthOfLongestSubstringKDistinct(s, k);
 return 0;
}

```

---

## Complexity Analysis

- **Time Complexity:**  $O(n)$   
Each character is added and removed from the window at most once.
- **Space Complexity:**  $O(k)$   
The frequency map stores at most  $k$  distinct characters.

# 10. Subarray with K Different Integers

You are given an integer array `nums` and an integer `k`.

Your task is to count how many **contiguous subarrays** contain **exactly k distinct integers**.

Such subarrays are called **good subarrays**.

A subarray must be continuous, meaning elements are taken from consecutive positions in the array.

---

## Example Explanation

For `nums` = [1, 2, 1, 2, 3] and `k` = 2

The good subarrays with exactly 2 distinct integers are:

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

So the answer is 7.

---

## Brute Force Approach

In this approach, we try **all possible subarrays** and count how many distinct integers each subarray contains.

Steps:

1. Loop over all possible starting indices.
  2. For each starting index, expand the subarray to the right.
  3. Use a frequency map to track how many distinct numbers are present.
  4. If the number of distinct integers becomes exactly k, increment the count.
  5. If it exceeds k, stop expanding further for that start index.
  6. Return the total count.
- 

## Code

```
#include <bits/stdc++.h>

using namespace std;

class Solution {

public:

 int subarraysWithKDistinct(vector<int>& nums, int k) {

 int n = nums.size();

 int count = 0;

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

 unordered_map<int, int> freq;

 for (int j = i; j < n; j++) {

 freq[nums[j]]++;

 if (freq.size() == k)

 count++;

 else if (freq.size() > k)

 break;
 }
 }

 return count;
 }
}
```

```

 freq[nums[j]]++;

 if (freq.size() == k)
 count++;

 if (freq.size() > k)
 break;

 }

 return count;
}

};

int main() {
 Solution sol;
 vector<int> nums = {1,2,1,2,3};
 int k = 2;
 cout << sol.subarraysWithKDistinct(nums, k);
 return 0;
}

```

## Complexity Analysis

- **Time Complexity:**  $O(N^2 * K)$

We examine all subarrays and manage a frequency map that can grow up to  $k$  elements.

- **Space Complexity:**  $O(K)$

The frequency map stores at most  $k$  distinct integers.

---

## Optimal Approach

### Algorithm

We use a **sliding window** idea with a mathematical observation:

Number of subarrays with **exactly  $k$  distinct integers**

- = subarrays with **at most  $k$  distinct**
- subarrays with **at most  $k-1$  distinct**

So we:

1. Write a helper function `atMostK()` that counts subarrays with at most  $K$  distinct elements.
  2. Use sliding window with two pointers.
  3. Maintain a frequency map for the window.
  4. Expand the window using the right pointer.
  5. Shrink the window from the left if distinct elements exceed  $K$ .
  6. For each valid window, add (`right - left + 1`) to the result.
  7. Final answer = `atMostK(k) - atMostK(k - 1)`.
- 

### Code

```
#include <bits/stdc++.h>
```

```

using namespace std;

class Solution {

public:

 int atMostK(vector<int>& nums, int k) {

 unordered_map<int,int> freq;

 int left = 0, count = 0;

 for (int right = 0; right < nums.size(); right++) {

 if (freq[nums[right]] == 0)

 k--;

 freq[nums[right]]++;

 while (k < 0) {

 freq[nums[left]]--;

 if (freq[nums[left]] == 0)

 k++;

 left++;

 }

 count += (right - left + 1);

 }

 return count;
 }
}

```

```

 }

int subarraysWithKDistinct(vector<int>& nums, int k) {
 return atMostK(nums, k) - atMostK(nums, k - 1);
}

};

int main() {
 Solution sol;
 vector<int> nums = {1,2,1,2,3};
 int k = 2;
 cout << sol.subarraysWithKDistinct(nums, k);
 return 0;
}

```

---

## Complexity Analysis

- **Time Complexity:**  $O(N)$   
Each element is processed at most twice using sliding window.
- **Space Complexity:**  $O(K)$   
The frequency map stores up to  $k$  distinct integers.

# 11. Smallest Window in a String Containing All Characters of Another String

## Problem recap

You are given two strings  $s$  and  $p$ .

You need to find the **smallest substring** of  $s$  that contains **all characters of  $p$  including duplicates**.

Rules:

- If no such substring exists, return ""
  - If multiple substrings have the same minimum length, return the one with the **smallest starting index**
- 

## Example 1

### Input

```
s = "timetopractice", p = "toc"
```

### Output

```
"toprac"
```

### Explanation

The substring "toprac" is the smallest window where characters t, o, and c all appear.

---

## Example 2

### Input

```
s = "zoomlazapzo", p = "oza"
```

### Output

```
"apzo"
```

---

# Naive Approach

## Algorithm

1. Generate **all possible substrings** of s.
  2. For each substring:
    - o Check if it contains **all characters of p with correct frequencies**.
  3. Track the substring with the **minimum length** that satisfies the condition.
  4. Return the smallest such substring.
- 

## Code

```
#include <climits>

#include <iostream>

#include <string>

using namespace std;

bool hasAllChars(string &sub, string &p) {

 int count[256] = {0};

 for (char ch : p)
 count[ch]++;
}

for (char ch : sub) {
```

```

 if (count[ch] > 0)

 count[ch]--;
}

for (int i = 0; i < 256; i++) {

 if (count[i] > 0)

 return false;

}

return true;
}

string minWindow(string &s, string &p) {

 int n = s.length();

 int minLen = INT_MAX;

 string res = "";

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

 for (int j = i; j < n; j++) {

 string sub = s.substr(i, j - i + 1);

 if (hasAllChars(sub, p)) {

 if (sub.length() < minLen) {

 minLen = sub.length();

 res = sub;
}

```

```

 }
 }

}

return res;
}

int main() {
 string s = "timetopractice";
 string p = "toc";
 cout << minWindow(s, p);
 return 0;
}

```

---

## Complexity Analysis

- **Time Complexity:**  $O(n^3)$
  - **Space Complexity:**  $O(1)$  (fixed size frequency array)
- 

## Better Approach (Binary Search on Answer)

### Algorithm

1. The minimum window length can range from  $|p|$  to  $|s|$ .
  2. Use **binary search** on window size.
  3. For a given window size  $mid$ :
    - o Check if **any substring of size  $mid$**  contains all characters of  $p$ .
  4. If valid, try smaller window sizes.
  5. Otherwise, try larger sizes.
  6. Store the smallest valid window found.
- 

## Code

```
#include <iostream>

#include <string>

#include <climits>

using namespace std;

bool isValid(string &s, string &p, int mid, int &start) {

 int count[256] = {0};

 int distinct = 0;

 for (char c : p) {

 if (count[c] == 0) distinct++;

 count[c]++;
 }

 for (int i = 0; i + mid - 1 < s.length(); i++) {

 if (distinct == p.length()) {
 start = i;
 return true;
 }

 if (count[s[i + mid]] == 0) distinct--;

 count[s[i + mid]]++;
 }
}
```

```

int matched = 0;

for (int i = 0; i < s.size(); i++) {
 count[s[i]]--;
 if (count[s[i]] == 0)
 matched++;

 if (i >= mid) {
 count[s[i - mid]]++;
 if (count[s[i - mid]] == 1)
 matched--;
 }

 if (i >= mid - 1 && matched == distinct) {
 start = i - mid + 1;
 return true;
 }
}

return false;
}

string smallestWindow(string s, string p) {
 int low = p.length(), high = s.length();

```

```

int minLen = INT_MAX, idx = -1;

while (low <= high) {

 int mid = (low + high) / 2;

 int start;

 if (isValid(s, p, mid, start)) {

 minLen = mid;

 idx = start;

 high = mid - 1;

 } else {

 low = mid + 1;

 }

}

if (idx == -1) return "";

return s.substr(idx, minLen);

}

int main() {

 string s = "timetopractice";

 string p = "toc";

 cout << smallestWindow(s, p);

 return 0;
}

```

}

---

## Complexity Analysis

- **Time Complexity:**  $O(n \log n)$
  - **Space Complexity:**  $O(1)$
- 

## Expected Approach (Sliding Window)

### Algorithm

1. Use **two pointers** (`start`, `end`) to maintain a sliding window on `s`.
  2. Maintain two frequency arrays:
    - One for characters in `p`
    - One for characters in the current window
  3. Expand the window by moving `end`.
  4. Once all characters of `p` are present:
    - Shrink the window from the left to minimize it.
  5. Keep updating the smallest valid window found.
  6. Return the smallest window.
- 

### Code

```
#include <iostream>
```

```

#include <string>
#include <vector>
#include <climits>
using namespace std;

string smallestWindow(string s, string p) {
 int len1 = s.length(), len2 = p.length();
 if (len1 < len2) return "";

 vector<int> countP(256, 0), countS(256, 0);

 for (char c : p)
 countP[c]++;
}

int start = 0, start_idx = -1, min_len = INT_MAX;
int matched = 0;

for (int end = 0; end < len1; end++) {
 countS[s[end]]++;

 if (countP[s[end]] != 0 && countS[s[end]] <= countP[s[end]])
 matched++;
}

```

```

 if (matched == len2) {

 while (countS[s[start]] > countP[s[start]] ||

countP[s[start]] == 0) {

 if (countS[s[start]] > countP[s[start]])

 countS[s[start]]--;

 start++;

 }

 int window_len = end - start + 1;

 if (window_len < min_len) {

 min_len = window_len;

 start_idx = start;

 }

 }

 if (start_idx == -1) return "";

 return s.substr(start_idx, min_len);

 }

int main() {

 string s = "timetopractice";

 string p = "toc";

```

```
 cout << smallestWindow(s, p);

 return 0;

}
```

---

## Complexity Analysis

- **Time Complexity:**  $O(n)$
- **Space Complexity:**  $O(1)$  (fixed size arrays)

# 12. Minimum Window Subsequence

### Problem recap

Given strings  $s_1$  and  $s_2$ , find the **smallest contiguous substring** of  $s_1$  such that  $s_2$  appears **as a subsequence** inside it (order preserved, not necessarily consecutive). If multiple windows have the same minimum length, return the **leftmost** one. If none exists, return "".

---

## 1) Naive Approach — Brute Force

**Idea:** Try all substrings of  $s_1$  and check if  $s_2$  is a subsequence of each.

### Algorithm

1. Generate every substring  $s_1[i \dots j]$ .

2. Check if s2 is a subsequence of that substring.
3. Track the smallest valid substring.

## Complexity

- **Time:**  $O(n^3)$  (substrings  $\times$  subsequence check)
- **Space:**  $O(1)$

## Code

```
#include <iostream>
#include <string>
#include <climits>
using namespace std;

bool isSubsequence(string &sub, string &s2) {
 int i = 0, j = 0;
 while (i < sub.size() && j < s2.size()) {
 if (sub[i] == s2[j]) j++;
 i++;
 }
 return j == s2.size();
}

string minWindow(string &s1, string &s2) {
```

```

int n = s1.size();

string ans = "";

int minLen = INT_MAX;

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

 for (int j = i; j < n; j++) {

 string sub = s1.substr(i, j - i + 1);

 if (isSubsequence(sub, s2)) {

 if ((int)sub.size() < minLen) {

 minLen = sub.size();

 ans = sub;

 }

 break; // smallest for this start

 }

 }

}

return ans;
}

int main() {

 string s1 = "abcdebdde", s2 = "bde";

 cout << minWindow(s1, s2) << endl; // bcde

}

```

---

## 2) Better Approach — Two Pointers + Backtracking

### Idea:

- From each index where  $s1[i] == s2[0]$ , scan forward to match  $s2$ .
- Once matched, **backtrack** to shrink the window as much as possible.

### Algorithm

1. For each  $i$  where  $s1[i] == s2[0]$ :
  - Move forward pointers to match all of  $s2$ .
  - Backtrack from the end to minimize the window.
2. Keep the smallest window.

### Complexity

- **Time:**  $O(n^2)$
- **Space:**  $O(1)$

### Code

```
#include <iostream>

#include <climits>

#include <string>

using namespace std;
```

```

string minWindow(string &s1, string &s2) {

 int n = s1.size(), m = s2.size();

 string ans = "";

 int minLen = INT_MAX;

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

 if (s1[i] != s2[0]) continue;

 int p1 = i, p2 = 0;

 while (p1 < n && p2 < m) {

 if (s1[p1] == s2[p2]) p2++;

 p1++;

 }

 if (p2 == m) {

 int end = p1 - 1;

 p2 = m - 1;

 while (end >= i) {

 if (s1[end] == s2[p2]) p2--;

 if (p2 < 0) break;

 end--;

 }

 int start = end;

 }

 }

}

```

```

 int len = p1 - start;

 if (len < minLen) {

 minLen = len;

 ans = s1.substr(start, len);

 }

 }

 return ans;
}

int main() {

 string s1 = "abcdebdde", s2 = "bde";

 cout << minWindow(s1, s2) << endl; // bcde

}

```

---

### 3) Expected / Optimal Approach — Next Occurrence Preprocessing

#### Idea:

Precompute where the **next occurrence** of each character appears in s1. Then, for any start index, we can jump quickly to match s2.

#### Algorithm

1. Build `nextPos[i][c]`: index of next occurrence of char c at or after i.

2. For each valid start, jump through nextPos to match all chars of s2.
3. Track the minimum window.

## Complexity

- **Time:**  $O(n \times m)$
- **Space:**  $O(n)$  (next occurrence table)

Har possible start se try karo ki s2 ko subsequence ke form me complete kar sakte ho ya nahi, aur fast jumping ke liye nextPos table use karo.

## Code

```
#include <iostream>
#include <vector>
#include <string>
#include <climits>
using namespace std;

string minWindow(string &s1, string &s2) {
 int n = s1.size(), m = s2.size();
 vector<vector<int>> nextPos(n + 1, vector<int>(26, -1));

 for (int c = 0; c < 26; c++)
 nextPos[n][c] = -1;
```

```

for (int i = n - 1; i >= 0; i--) {
 nextPos[i] = nextPos[i + 1];
 nextPos[i][s1[i] - 'a'] = i;
}

string ans = "";
int minLen = INT_MAX;

for (int start = 0; start < n; start++) {
 if (s1[start] != s2[0]) continue;

 int idx = start;
 bool ok = true;

 for (int j = 0; j < m; j++) {
 if (idx == -1) { ok = false; break; }

 idx = nextPos[idx][s2[j] - 'a'];

 if (idx == -1) { ok = false; break; }

 idx++;
 }

 if (ok) {
 int endIdx = idx - 1;
 }
}

```

```

 int len = endIdx - start + 1;

 if (len < minLen) {

 minLen = len;

 ans = s1.substr(start, len);

 }

 }

 return ans;
}

```

```

int main() {

 string s1 = "abcdebdde", s2 = "bde";

 cout << minWindow(s1, s2) << endl; // bcde

}

```

---

```

for (int i = n - 1; i >= 0; i--) {

 nextPos[i] = nextPos[i + 1];

 nextPos[i][s1[i] - 'a'] = i;

}

```

Isko todte hain

Step 1: Peeche wali info copy

```
nextPos[i] = nextPos[i + 1];
```

Matlab:

“Agar maine is position se kuch nahi badla,  
to jo aage se milta tha, wahi yahan se bhi milega.”

Step 2: Current character update

```
nextPos[i][s1[i] - 'a'] = i;
```

Matlab:

“Is index par ye character khud mil raha hai,  
to iska next occurrence yahi hai.”

nextPos use kyun ho raha hai?

Code me ye line dekho:

```
idx = nextPos[idx][s2[j] - 'a'];
```

Iska matlab:

“Current position se,  
s2[j] next kahan milega?”

Mil gaya → jump

Nahi mila → fail

# **Heaps**

# 1. Introduction to Priority Queues using Binary Heaps

## What is Priority Queue ?

A Priority Queue is a special type of queue where each element is assigned a priority, and instead of being processed in the order they arrive (like a normal queue), the element with the highest priority is always processed first.

If two elements have the same priority, they are handled based on their insertion order.

Think of it like an emergency room in a hospital patients are not treated just by arrival time, but by how critical their condition is; someone with a heart attack gets attention before someone with a cold, regardless of who came first. This makes priority queues super useful in scenarios like task scheduling, pathfinding algorithms, and real-time systems.

---

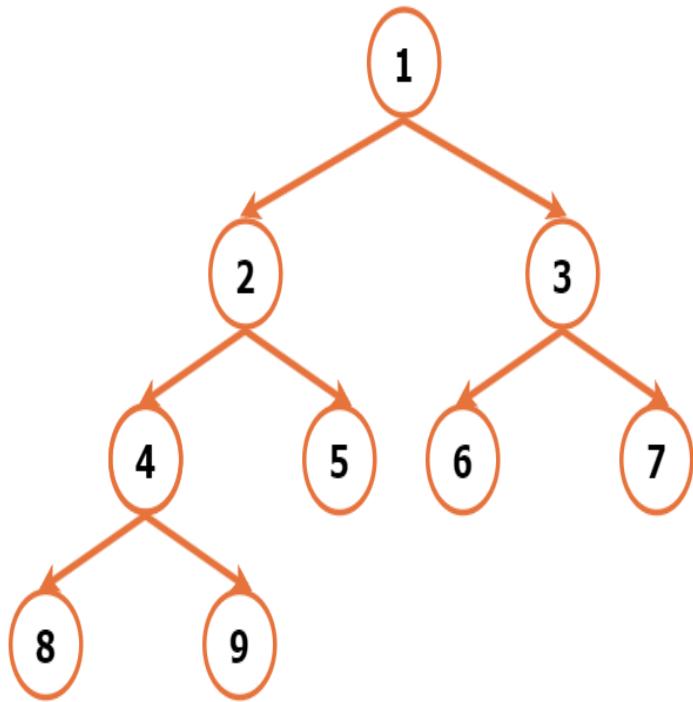
## Binary Heap:

A Binary Heap is a Binary Tree that satisfies the following conditions.

- It should be a Complete Binary Tree.
- It should satisfy the Heap property.

## Complete Binary Tree:

The tree in Which all the levels are completely filled except the last level and last level is filled in such a way that all the keys are as left as possible.

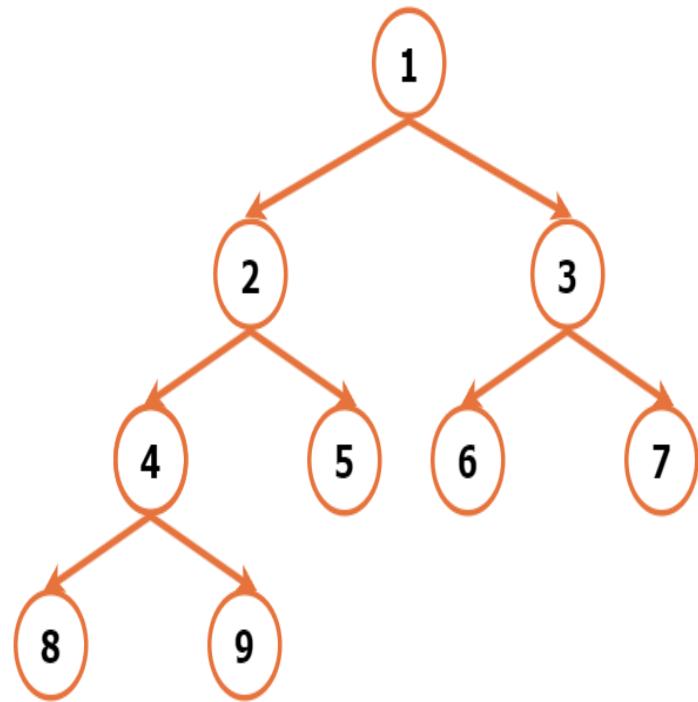


Complete Binary Tree

## Heap Property:

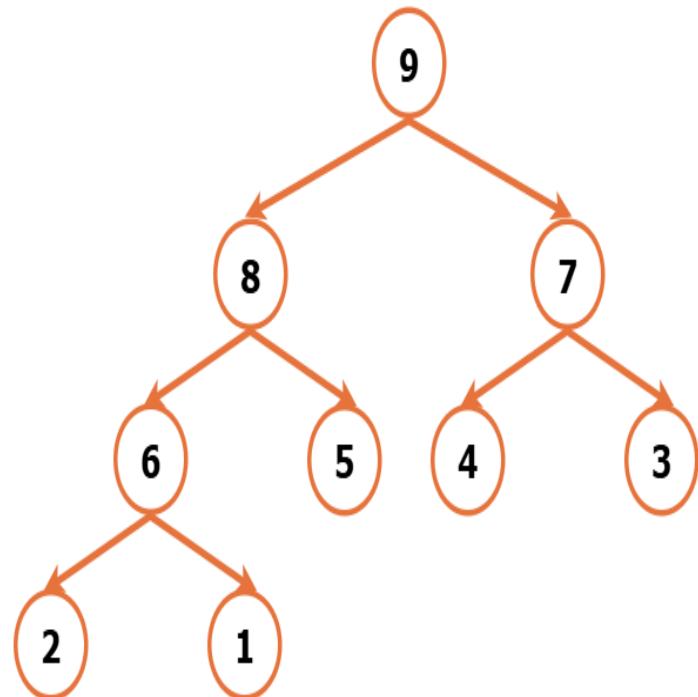
Binary Heap is either a Min Heap or Max Heap. Property of the Binary Heap decides whether it is Min Heap or Max Heap.

- Min Heap property: For every node in a binary heap, if the node value is less than its right and left child's value then Binary Heap is known as Min Heap. The property of Node's value less than its children's value is known as Min Heap property. In Min Heap, the root value is always the Minimum value in Heap.



Min Heap

- Max Heap property: For every node in a binary heap, if the node value is greater than its right and left child's value then Binary Heap is known as Max Heap. The property of being Node's value greater than its children's value is known as Max Heap property. In Max Heap, the root value is always the maximum value in Heap.



## Representation of the Binary Heap

A Binary Heap is represented as an array.

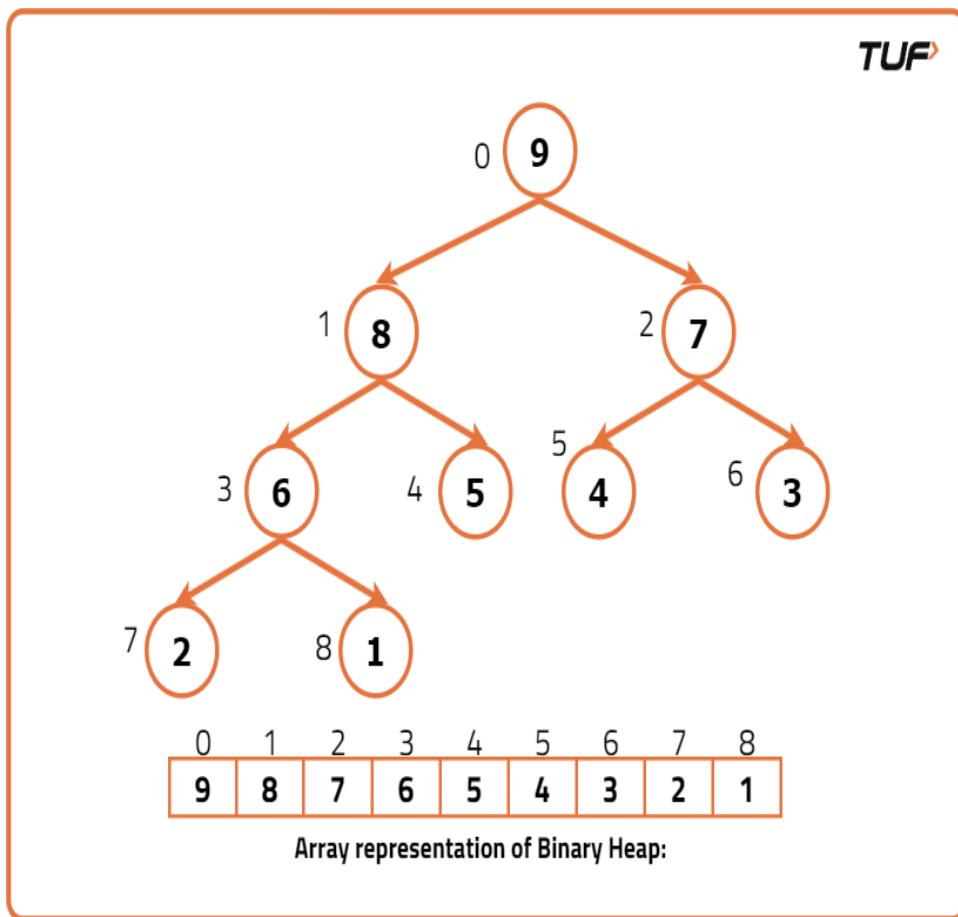
- The root value is at arr[0]

The below table summarizes how the node and its children are stored in an array.

| Node index | Left child Index | Right Child Index |
|------------|------------------|-------------------|
| i          | $2*i+1$          | $2*i+2$           |

If the child is at index  $i$ , then its parent index can be found using the below formula.

| Child's Index | Parent's Index |
|---------------|----------------|
| $i$           | $(i-1)/2$      |



## Operations Associated with Min Heap

- Insert()
- Heapify()
- getMin()

- ExtractMin()
- Delete()
- Decreasekey()

Let's see how these operations are done Using Min Heap.

**Note:** The Binary heap should always be a complete binary tree and should satisfy the corresponding heap property (Min / Max). If any of the two conditions are disturbed we should make necessary modifications in a heap to satisfy the two conditions.

## Insert():

- Insert operation inserts a new key in the Binary Heap.

**Steps Followed for inserting the key in Binary Heap:**

- First Insert the key at the first vacant position from the left on the last level of the heap. IF the last level is completely filled, then insert the key as the left-most element in the next level.
- Inserting a new key at the first vacant position in the last level preserves the Complete binary tree property, The Min heap property may get affected we need to check for it.
- If the inserted key parent is less than the key, then the Min heap property is also not violated.
- If the parent is greater than the inserted key value, then swap the values. Now the heap property may get violated at the parent node. So repeat the same process until the heap property is satisfied.
- Inserting an element takes  $O(\log N)$  Time Complexity. Below shows the animation of how -1 is inserted into a Binary heap by following above described steps.

## Heapify():

- Suppose there exists a Node at some index i, where the Minheap property is Violated.
- We assume that all the subtrees of the tree rooted at index i are valid binary heaps.
- The Heapify function is used to restore the Minheap, by fixing the violation.

**Steps to be followed for Heapify:**

- First find out the Minimum among the Violated Node, left, and right child of Violated Node.

- If the Minimum among them is the left child, then swap the Violated Node value with the Left child value and recursively call the function on the left Child.
- If the Minimum among them is the right child, then swap the Violated Node value with the right child value and recursively call the function right Child.
- Recursive call stops when the heap property is not violated.

### **getMin():**

- It returns the minimum value in the Binary Heap.
- As we all know, the root Node is the Minimum value in Min Heap. Simply return the arr[0].

### **ExtractMin():**

- This removes the Minimum element from the heap.

#### **Steps to be followed to Remove Minimum value/root Node:**

- Copy the last Node value to the Root Node, and decrease the size, this means that the root value is now deleted from the heap, and the size is decreased by 1.
- By doing the above step we ensure that the Complete binary tree property is not violated, as we are copying the last node value to the root node value, the Min Heap property gets violated.
- Call the Heapify function to convert it into a valid heap.

### **Decreasekey():**

- Given an index and a value, we need to update the value at the index with the given value. We assume that the given value is less than the existing value at that index.

#### **Steps to be followed for Decreasekey():**

- Let's the index be i and the value be new\_val. Update existing value at index i with new\_val i.e arr[i] = new\_val.
- By performing the above step, the Complete binary tree property is not violated, but the Min heap property may get violated.
- As the new\_val we are inserting is less than the previously existing value, the min-heap property is not violated in subtrees of this rooted tree.
- It may get violated in its ancestors, so as we do in insert operation, check the value of a current node with its parent node, if it violates the min-heap property

Swap the nodes and recursively do the same.

## Delete() :

- Given an index, delete the value at that index from the min-heap.

Steps to be followed for Delete operation():

- First, update the value at the index that needs to be deleted with INT\_MIN.
- Now call the Decreasekey() function at the index which is need to be deleted. As the value at the index is the least, it reaches the top.
- Now call the ExtractMin() operation which deletes the root node in Minheap.
- In this way, the desired index value is deleted from the Minheap.

Code:

C++JavaPythonJavaScript

```
#include <bits/stdc++.h>

#include
using namespace std;

class BinaryHeap {
public:
 // Maximum elements that can be stored in heap
 int capacity;

 // Current no of elements in heap
 int size;

 // Array for storing the keys
 int* arr;

 BinaryHeap(int cap) {
 // Assigning the capacity
 capacity = cap;

 // Initially size of heap is zero
 size = 0;

 // Creating an array
 arr = new int[capacity];
 }
}
```

```

}

// Returns the parent of ith Node
int parent(int i) {
 return (i - 1) / 2;
}

// Returns the left child of ith Node
int left(int i) {
 return 2 * i + 1;
}

// Returns the right child of the ith Node
int right(int i) {
 return 2 * i + 2;
}

// Insert a new key x
void Insert(int x) {
 if (size == capacity) {
 cout << "Binary Heap Overflow" << endl;
 return;
 }

 // Insert new element at end
 arr[size] = x;

 // Store the index, for checking heap property
 int k = size;

 // Increase the size
 size++;

 // Fix the min heap property
 while (k != 0 && arr[parent(k)] > arr[k]) {
 swap(&arr[parent(k)], &arr[k]);
 k = parent(k);
 }
}

void Heapify(int ind) {

```

```

// Right child
int ri = right(ind);

// Left child
int li = left(ind);

// Initially assume violated value is minimum
int smallest = ind;

if (li < size && arr[li] < arr[smallest])
 smallest = li;

if (ri < size && arr[ri] < arr[smallest])
 smallest = ri;

// If the Minimum among the three nodes is not the
parent itself,
// then swap and call Heapify recursively
if (smallest != ind) {
 swap(&arr[ind], &arr[smallest]);
 Heapify(smallest);
}
}

int getMin() {
 return arr[0];
}

int ExtractMin() {
 if (size <= 0)
 return INT_MAX;

 if (size == 1) {
 size--;
 return arr[0];
 }

 int mini = arr[0];

 // Copy last Node value to root Node
 arr[0] = arr[size - 1];
}

```

```

 size--;

 // Call heapify on root node
 Heapify(0);

 return mini;
 }

void Decreasekey(int i, int val) {
 // Updating the new value
 arr[i] = val;

 // Fixing the Min heap
 while (i != 0 && arr[parent(i)] > arr[i]) {
 swap(&arr[parent(i)], &arr[i]);
 i = parent(i);
 }
}

void Delete(int i) {
 Decreasekey(i, INT_MIN);
 ExtractMin();
}

void swap(int* x, int* y) {
 int temp = *x;
 *x = *y;
 *y = temp;
}

void print() {
 for (int i = 0; i < size; i++)
 cout << arr[i] << " ";
 cout << endl;
}

int main() {
 BinaryHeap h(20);
 h.Insert(4);
}

```

```

 h.Insert(1);
 h.Insert(2);
 h.Insert(6);
 h.Insert(7);
 h.Insert(3);
 h.Insert(8);
 h.Insert(5);

 cout << "Min value is " << h.getMin() << endl;

 h.Insert(-1);
 cout << "Min value is " << h.getMin() << endl;

 h.Decreasekey(3, -2);
 cout << "Min value is " << h.getMin() << endl;

 h.ExtractMin();
 cout << "Min value is " << h.getMin() << endl;

 h.Delete(0);
 cout << "Min value is " << h.getMin() << endl;

 return 0;
}

}

```

## Time Complexities :

| Function   | Time Complexity |
|------------|-----------------|
| Insert() : | $O(\log N)$     |
| Heapify()  | $O(\log N)$     |

|               |         |
|---------------|---------|
| getMin()      | O(1)    |
| ExtractMin()  | O(logN) |
| Decreasekey() | O(logN) |
| Delete()      | O(logN) |

#### ◆ Data members ka role

capacity → max elements  
 size → current elements  
 arr[ ] → heap array

---

#### ◆ INSERT(x) — sabse pehle samjho

##### Steps:

1. New element **end me daalo**
2. Phir **upar bubble up** karao jab tak heap property na ban jaaye

```
arr[size] = x;
int k = size;
size++;
```

##### Bubble-up logic:

```
while (k != 0 && arr[parent(k)] > arr[k]) {
 swap(parent, k);
```

```
k = parent(k);
}
```

👉 Jab child chhota ho parent se → swap

---

### Example

Insert: -1

Before:

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

After inserting at end:

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

Bubble up:

-1 ↔ 6  
-1 ↔ 3  
-1 ↔ 1

Final:

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

---

### ◆ HEAPIFY(ind) — neeche fix karna

Jab root ya koi middle element galat ho jaaye

Steps:

1. Parent, left, right me **minimum** nikaalo

2. Agar parent minimum nahi → swap
3. Recursively neeche jao

```
smallest = ind;
if (left < size && arr[left] < arr[smallest]) smallest = left;
if (right < size && arr[right] < arr[smallest]) smallest = right;

if (smallest != ind) {
 swap(ind, smallest);
 Heapify(smallest);
}
```

---

- ◆ **ExtractMin() — root hataana**

Steps:

1. Root store karo (answer)
2. Last element ko root pe le aao
3. Size--
4. Heapify(0)

```
mini = arr[0];
arr[0] = arr[size-1];
size--;
Heapify(0);
```

👉 Root hataane ke baad heap property toot jaati hai → Heapify se fix

---

- ◆ **getMin()**

```
return arr[0];
```

Simple: root hamesha minimum hota hai

---

- ◆ **DecreaseKey(i, val)**

Jab kisi index ki value **kam** karni ho

Steps:

1. Value update
2. Bubble-up (same as insert logic)

```
arr[i] = val;
while (i != 0 && arr[parent(i)] > arr[i]) {
 swap(i, parent(i));
 i = parent(i);
}
```

---

- ◆ **Delete(i) — clever trick**

Heap me direct delete mushkil hota hai  
Isliye trick use ki:

```
Decreasekey(i, INT_MIN);
ExtractMin();
```

- 👉 Node ko sabse chhota bana diya
  - 👉 Root tak aa gaya
  - 👉 ExtractMin se nikal gaya
- 

- ◆ **swap function**

Tumne pointer-based swap likha hai:

```
void swap(int* x, int* y)
```

Ye bhi sahi hai, STL swap() bhi use kar sakte ho.

---

- ◆ **Main function ka flow**

Insert → heap banti hai

getMin → root

DecreaseKey → bubble up

ExtractMin → root remove

Delete → decrease + extract

---

 **Sabse important intuition (yaad rakhna)**

- Insert / DecreaseKey → bubble UP
- ExtractMin / Delete → bubble DOWN (Heapify)

## 2. Min Heap and Max Heap Implementation

A heap is a complete binary tree that follows a special order property.

In a **Min Heap**, the smallest element is always at the root, meaning every parent node is smaller than its children.

In a **Max Heap**, the largest element is always at the root, meaning every parent node is larger than its children.

Heaps are commonly used to efficiently get the minimum or maximum element, such as in priority queues.

Example:

If we insert elements 10, 5, 20

- Min Heap root will be 5
  - Max Heap root will be 20
- 

## Approach 1: Min Heap Implementation

### Algorithm

1. Use an array to represent the heap.
2. For insertion:
  - o Insert the element at the end.
  - o Move it up (heapify up) while parent is greater than current.
3. For deletion of minimum:
  - o Replace root with last element.
  - o Remove last element.
  - o Move root down (heapify down) to maintain heap property.
4. Root always stores the minimum element.

### Code

```
#include <bits/stdc++.h>
using namespace std;

class MinHeap{
 vector<int> heap;

 void heapifyUp(int index){
 while(index>0){
 int parent=(index-1)/2;
 if(heap[parent]>heap[index]){
 swap(heap[parent],heap[index]);
 }
 }
 }
}
```

```

 index=parent;
 }else break;
}
}

void heapifyDown(int index){
 int n=heap.size();
 while(true){
 int left=2*index+1;
 int right=2*index+2;
 int smallest=index;
 if(left<n && heap[left]<heap[smallest]) smallest=left;
 if(right<n && heap[right]<heap[smallest]) smallest=right;
 if(smallest!=index){
 swap(heap[index],heap[smallest]);
 index=smallest;
 }else break;
 }
}

public:
 void insert(int val){
 heap.push_back(val);
 heapifyUp(heap.size()-1);
 }

 void removeMin(){
 if(heap.empty()) return;
 heap[0]=heap.back();
 heap.pop_back();
 heapifyDown(0);
 }

 int getMin(){
 if(heap.empty()) return -1;
 return heap[0];
 }
};

```

## Complexity Analysis

- Insertion:  $O(\log n)$
  - Deletion:  $O(\log n)$
  - Get Minimum:  $O(1)$
  - Space Complexity:  $O(n)$
- 

## Approach 2: Max Heap Implementation

### Algorithm

1. Use an array to represent the heap.
2. For insertion:
  - Insert the element at the end.
  - Move it up (heapify up) while parent is smaller than current.
3. For deletion of maximum:
  - Replace root with last element.
  - Remove last element.
  - Move root down (heapify down) to maintain heap property.
4. Root always stores the maximum element.

### Code

```
#include <bits/stdc++.h>
using namespace std;

class MaxHeap{
 vector<int> heap;
```

```

void heapifyUp(int index){
 while(index>0){
 int parent=(index-1)/2;
 if(heap[parent]<heap[index]){
 swap(heap[parent],heap[index]);
 index=parent;
 }else break;
 }
}

void heapifyDown(int index){
 int n=heap.size();
 while(true){
 int left=2*index+1;
 int right=2*index+2;
 int largest=index;
 if(left<n && heap[left]>heap[largest]) largest=left;
 if(right<n && heap[right]>heap[largest]) largest=right;
 if(largest!=index){
 swap(heap[index],heap[largest]);
 index=largest;
 }else break;
 }
}

public:
 void insert(int val){
 heap.push_back(val);
 heapifyUp(heap.size()-1);
 }

 void removeMax(){
 if(heap.empty()) return;
 heap[0]=heap.back();
 heap.pop_back();
 heapifyDown(0);
 }
}

```

```

int getMax(){
 if(heap.empty()) return -1;
 return heap[0];
}

```

### Complexity Analysis

- Insertion:  $O(\log n)$
- Deletion:  $O(\log n)$
- Get Maximum:  $O(1)$
- Space Complexity:  $O(n)$

## 3. Check if an Array Represents a Min Heap

Given an array of integers `nums`, check whether it represents a binary min-heap.

In a binary min-heap, every parent node has a value less than or equal to its children. The tree is complete, and this property must hold for all nodes.

Example:

Input: `nums = [10, 20, 30, 21, 23]`

Here, every parent value is smaller than its children, so it is a valid min-heap.

Input: `nums = [10, 20, 30, 25, 15]`

The node 20 has a child 15 which is smaller, so the min-heap property is violated.

## Approach 1

### Algorithm

We store a binary heap in an array.

For an index  $i$ :

- Left child is at  $2*i + 1$
- Right child is at  $2*i + 2$

Leaf nodes do not have children, so they cannot violate the heap property.

In an array of size  $n$ , all non-leaf nodes are from index 0 to  $(n/2) - 1$ .

Steps:

1. Iterate from index 0 to  $(n/2) - 1$ .
2. For each index:
  - Calculate left child index.
  - If left child exists and parent value is greater, return false.
  - Calculate right child index.
  - If right child exists and parent value is greater, return false.
3. If no violation is found, return true.

### Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 bool isMinHeap(vector<int>& nums) {
 int n = nums.size();

 for (int i = 0; i <= (n / 2) - 1; i++) {
 int left = 2 * i + 1;
```

```

 if (left < n && nums[i] > nums[left]) {
 return false;
 }

 int right = 2 * i + 2;
 if (right < n && nums[i] > nums[right]) {
 return false;
 }
 }
 return true;
}
};

int main() {
 Solution obj;
 vector<int> nums = {10, 20, 30, 21, 23};
 cout << (obj.isMinHeap(nums) ? "true" : "false") << endl;
 return 0;
}

```

### Complexity Analysis

- Time Complexity:  $O(n)$  because we check at most  $n/2$  nodes.
- Space Complexity:  $O(1)$  since no extra space is used.

## 4. Convert Min Heap to Max Heap

Given an array that represents a binary **Min Heap**, convert it into a **Max Heap**.

In a Min Heap, every parent node is smaller than its children, while in a Max Heap, every parent node is greater than its children.

The array already represents a complete binary tree, so we only need to rearrange elements to satisfy the Max Heap property.

Example:

Input: arr = {3, 5, 9, 6, 8, 20, 10, 12, 18, 9}

After conversion, the root becomes the maximum element, and all parent nodes are greater than their children.

---

## Algorithm

We do not care that the input is a Min Heap. We directly build a Max Heap from the array.

Key points:

- In an array of size n, all non-leaf nodes are from index 0 to  $(n-2)/2$ .
- Leaf nodes already satisfy heap property.
- We start heapifying from the **last non-leaf node** and move upward to the root.

Steps:

1. Find the last non-leaf node at index  $(n-2)/2$ .
2. For each index from  $(n-2)/2$  down to 0:
  - Apply **MaxHeapify** on that index.
3. MaxHeapify compares a node with its children.
  - If any child is larger, swap with the largest child.
  - Recursively heapify the affected subtree.
4. After finishing, the array becomes a valid Max Heap.

## Code

```
#include <bits/stdc++.h>
using namespace std;

void MaxHeapify(int arr[], int i, int N){
 int left = 2*i + 1;
 int right = 2*i + 2;
```

```

int largest = i;

if(left < N && arr[left] > arr[largest])
 largest = left;
if(right < N && arr[right] > arr[largest])
 largest = right;

if(largest != i){
 swap(arr[i], arr[largest]);
 MaxHeapify(arr, largest, N);
}
}

void convertMaxHeap(int arr[], int N){
 for(int i = (N - 2) / 2; i >= 0; i--){
 MaxHeapify(arr, i, N);
 }
}

void printArray(int arr[], int n){
 for(int i = 0; i < n; i++)
 cout << arr[i] << " ";
}

int main(){
 int arr[] = {3, 5, 9, 6, 8, 20, 10, 12, 18, 9};
 int N = sizeof(arr) / sizeof(arr[0]);

 printArray(arr, N);
 cout << endl;

 convertMaxHeap(arr, N);

 printArray(arr, N);
 return 0;
}

```

- Time Complexity:  $O(n)$  because heapifying all internal nodes takes linear time.
- Space Complexity:  $O(n)$  due to recursion stack in worst case.

# 5. Kth Largest / Smallest Element in an Array

Given an array `nums` and an integer `k`, return the **kth largest element** in the array.

The `k`th largest element means the element which will appear at position `k` if the array is sorted in descending order.

Example:

Input: `nums` = [1, 2, 3, 4, 5], `k` = 2

Sorted (descending): [5, 4, 3, 2, 1]

The 2nd largest element is 4.

Input: `nums` = [-5, 4, 1, 2, -3], `k` = 5

Sorted (descending): [4, 2, 1, -3, -5]

The 5th largest element is -5.

---

## Approach 1: Brute Force (Using Min Heap)

### Algorithm

We maintain a **min-heap of size `K`** that stores the `K` largest elements seen so far.

Steps:

1. Create a min-heap.
2. Insert the first `k` elements of the array into the heap.
3. For each remaining element:
  - o If the element is greater than the top of the heap (smallest among `K` largest),
    - Remove the top element.
    - Insert the current element.
4. After processing all elements, the top of the heap is the `k`th largest element.

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int kthLargestElement(vector<int>& nums, int k) {
 priority_queue<int, vector<int>, greater<int>> pq;

 for(int i = 0; i < k; i++)
 pq.push(nums[i]);

 for(int i = k; i < nums.size(); i++) {
 if(nums[i] > pq.top()) {
 pq.pop();
 pq.push(nums[i]);
 }
 }
 return pq.top();
 }
};

int main() {
 vector<int> nums = {-5, 4, 1, 2, -3};
 int k = 5;

 Solution sol;
 cout << sol.kthLargestElement(nums, k);
 return 0;
}
```

## Complexity Analysis

- Time Complexity:  $O(N \log K)$
- Space Complexity:  $O(K)$

## Approach 2: Optimal (Quick Select)

### Algorithm

This approach is based on partitioning, similar to Quick Sort.

Steps:

1. Choose a random pivot index in the current range.
2. Partition the array such that:
  - o Elements greater than pivot are on the left.
  - o Elements smaller or equal are on the right.
3. Place the pivot in its correct position.
4. If pivot index equals  $k-1$ , return the pivot value.
5. If pivot index is greater than  $k-1$ , repeat on the left part.
6. If pivot index is smaller than  $k-1$ , repeat on the right part.
7. Continue until the  $k$ th largest element is found.

### Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int kthLargestElement(vector<int>& nums, int k) {
 if(k > nums.size()) return -1;

 int left = 0, right = nums.size() - 1;

 while(true) {
 int pivotIndex = randomIndex(left, right);
 pivotIndex = partition(nums, pivotIndex, left, right);
```

```

 if(pivotIndex == k - 1)
 return nums[pivotIndex];
 else if(pivotIndex > k - 1)
 right = pivotIndex - 1;
 else
 left = pivotIndex + 1;
 }
}

private:
 int randomIndex(int left, int right) {
 return left + rand() % (right - left + 1);
 }

 int partition(vector<int>& nums, int pivotIndex, int left, int
right) {
 int pivot = nums[pivotIndex];
 swap(nums[left], nums[pivotIndex]);

 int ind = left + 1;
 for(int i = left + 1; i <= right; i++) {
 if(nums[i] > pivot) {
 swap(nums[i], nums[ind]);
 ind++;
 }
 }
 swap(nums[left], nums[ind - 1]);
 return ind - 1;
 }
};

int main() {
 vector<int> nums = {-5, 4, 1, 2, -3};
 int k = 5;

 Solution sol;
 cout << sol.kthLargestElement(nums, k);
 return 0;
}

```

}

## Complexity Analysis

- Time Complexity:
  - Average Case:  $O(N)$   
In the average case (when the pivot is chosen randomly):
  - Assuming the array gets divided into two equal parts, with every partitioning step, the search range is reduced by half. Thus, the time complexity is  $O(N + N/2 + N/4 + \dots + 1) = O(N)$ .
  - 
  - In the worst-case scenario (when the element at the left or right index is chosen as the pivot):
  - In such cases, the array is divided into two unequal halves, and the search range is reduced by one element with every partitioning step. Thus, the time complexity is  $O(N + N-1 + N-2 + \dots + 1) = O(N^2)$ . However, the probability of this worst-case scenario is negligible.
  - Worst Case:  $O(N^2)$
- Space Complexity:  $O(1)$

## 6. Sort K Sorted Array

Given an array  $\text{arr} [ ]$  and an integer  $k$ , the array is such that every element is at most  $k$  positions away from its correct position in the fully sorted array.

Your task is to completely sort the array.

Example:

Input:  $\text{arr} = [6, 5, 3, 2, 8, 10, 9]$ ,  $k = 3$

After sorting, the array becomes  $[2, 3, 5, 6, 8, 9, 10]$ .

Input:  $\text{arr} = [1, 4, 5, 2, 3, 6, 7, 8, 9, 10]$ ,  $k = 2$

After sorting, the array becomes  $[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$ .

## Approach 1: Brute Force Approach

### Algorithm

In this approach, we ignore the fact that the array is nearly sorted.  
We treat it like a normal unsorted array and sort the entire array.

Steps:

1. Take the entire array as input.
2. Apply a standard sorting algorithm.
3. Return the fully sorted array.

### Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 vector<int> sortNearlySortedArray(vector<int>& arr, int k) {
 sort(arr.begin(), arr.end());
 return arr;
 }
};

int main() {
 vector<int> arr = {6, 5, 3, 2, 8, 10, 9};
 int k = 3;

 Solution obj;
 vector<int> result = obj.sortNearlySortedArray(arr, k);

 for(int x : result)
 cout << x << " ";
 return 0;
}
```

- Time Complexity:  $O(n \log n)$  because the entire array is sorted.

- Space Complexity:  $O(1)$  for in-place sort (ignoring recursion stack).
- 

## Approach 2: Optimal Approach (Using Min Heap)

### Algorithm

Since the array is k-sorted, the smallest element must be present within the first  $k + 1$  elements.

Steps:

1. Create a min heap.
2. Insert the first  $k + 1$  elements into the heap.
3. For each remaining element:
  - Remove the minimum element from the heap and add it to the result.
  - Insert the current element into the heap.
4. After processing all elements, remove all remaining elements from the heap and add them to the result.
5. Return the result array.

### Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 vector<int> sortNearlySortedArray(vector<int>& arr, int k) {
 priority_queue<int, vector<int>, greater<int>> minHeap;
 vector<int> result;

 for(int i = 0; i <= k && i < arr.size(); i++)
 minHeap.push(arr[i]);

 for(int i = k + 1; i < arr.size(); i++) {
 result.push_back(minHeap.top());
 minHeap.pop();
 minHeap.push(arr[i]);
 }

 while(!minHeap.empty())
 result.push_back(minHeap.top());

 return result;
 }
}
```

```

 for(int i = k + 1; i < arr.size(); i++) {
 result.push_back(minHeap.top());
 minHeap.pop();
 minHeap.push(arr[i]);
 }

 while(!minHeap.empty()) {
 result.push_back(minHeap.top());
 minHeap.pop();
 }
 return result;
 }
};

int main() {
 vector<int> arr = {6, 5, 3, 2, 8, 10, 9};
 int k = 3;

 Solution obj;
 vector<int> sortedArr = obj.sortNearlySortedArray(arr, k);

 for(int x : sortedArr)
 cout << x << " ";
 return 0;
}

```

## Complexity Analysis

- Time Complexity:  $O(n \log k)$  because each heap operation takes  $O(\log k)$ .
- Space Complexity:  $O(k)$  due to the min heap of size  $k + 1$ .

# 7. Merge M Sorted Lists

Given an array heads where each element is the head of a sorted linked list, merge all these sorted linked lists into one single sorted linked list and return its head.

All input linked lists are already sorted in ascending order.

Example:

Input: heads = [[1 -> 2 -> 3 -> 4], [-4 -> -3], [-5 -> -3 -> 1 -> 2 -> 3 -> 4]]

After merging all nodes in sorted order, the final list becomes:

-5 -> -4 -> -3 -> -3 -> 1 -> 1 -> 2 -> 2 -> 3 -> 3 -> 4 -> 4

Input: heads = [[-5 -> -4 -> -1], [10 -> 11 -> 12]]

Output: -5 -> -4 -> -1 -> 10 -> 11 -> 12

---

## Approach 1: Brute Force Approach

### Algorithm

In this approach, we do not take advantage of the fact that the lists are already sorted.

Steps:

1. Create an empty array to store all node values.
2. Traverse each linked list one by one.
3. Push every node's value into the array.
4. Sort the array.
5. Create a new linked list using the sorted values.
6. Return the head of the newly created list.

### Code

```
#include <bits/stdc++.h>

using namespace std;
```

```

struct ListNode {

 int val;

 ListNode *next;

 ListNode(int x) : val(x), next(NULL) {}

};

class Solution {

public:

 ListNode* mergeKSortedLists(vector<ListNode*>& lists) {

 vector<int> values;

 for(auto list : lists) {

 while(list != NULL) {

 values.push_back(list->val);

 list = list->next;

 }

 }

 sort(values.begin(), values.end());

 ListNode* dummy = new ListNode(0);

 ListNode* curr = dummy;

```

```

 for(int v : values) {

 curr->next = new ListNode(v);

 curr = curr->next;

 }

 return dummy->next;
 }

};

int main() {

 ListNode* a = new ListNode(1);

 a->next = new ListNode(4);

 a->next->next = new ListNode(5);

 ListNode* b = new ListNode(1);

 b->next = new ListNode(3);

 b->next->next = new ListNode(4);

 ListNode* c = new ListNode(2);

 c->next = new ListNode(6);

 vector<ListNode*> lists = {a, b, c};
}

```

```

Solution obj;

ListNode* res = obj.mergeKSortedLists(lists);

while(res) {

 cout << res->val << " ";
 res = res->next;
}

return 0;
}

```

## Complexity Analysis

- Time Complexity:  $O(N \log N)$  where  $N$  is the total number of nodes.
  - Space Complexity:  $O(N)$  for storing all node values.
- 

## Approach 2: Optimal Approach (Using Min Heap)

### Algorithm

This approach uses the fact that all lists are already sorted.

Steps:

1. Create a min-heap that stores pointers to nodes.
2. Push the head of each non-empty linked list into the heap.
3. Create a dummy node to build the result list.
4. While the heap is not empty:

- Extract the node with the smallest value.
  - Attach it to the result list.
  - If the extracted node has a next node, push that next node into the heap.
5. Return the node next to the dummy head.

### Code

```
#include <bits/stdc++.h>

using namespace std;

struct ListNode {
 int val;
 ListNode* next;
 ListNode(int x) : val(x), next(NULL) {}
};

class Compare {
public:
 bool operator()(ListNode* a, ListNode* b) {
 return a->val > b->val;
 }
};

// kya A ko B ke baad aana chahiye?
};
```

```

class Solution {

public:

 ListNode* mergeKLists(vector<ListNode*>& lists) {

 priority_queue<ListNode*, vector<ListNode*>, Compare> pq;

 for(auto list : lists) {

 if(list != NULL)

 pq.push(list);

 }

 ListNode* dummy = new ListNode(0);

 ListNode* tail = dummy;

 while(!pq.empty()) {

 ListNode* node = pq.top();

 pq.pop();

 tail->next = node;

 tail = tail->next;

 if(node->next != NULL)

 pq.push(node->next);

 }

 return dummy->next;
 }
}

```

```

 }

 return dummy->next;
}

};

int main() {

 ListNode* l1 = new ListNode(1);

 l1->next = new ListNode(4);

 l1->next->next = new ListNode(5);

 ListNode* l2 = new ListNode(1);

 l2->next = new ListNode(3);

 l2->next->next = new ListNode(4);

 ListNode* l3 = new ListNode(2);

 l3->next = new ListNode(6);

 vector<ListNode*> lists = {l1, l2, l3};

 Solution sol;

 ListNode* res = sol.mergeKLists(lists);

 while(res) {

```

```

 cout << res->val << " ";
 res = res->next;
}

return 0;
}

```

### Complexity Analysis

- Time Complexity:  $O(N \log K)$  where N is total nodes and K is number of lists.
- Space Complexity:  $O(K)$  for the min heap.

## 8. Replace Elements by Its Rank in the Array

Given an array of integers, replace each element with its **rank** when the array is sorted.

The smallest element gets rank 1.

If there are duplicate values, they must get the **same rank**.

Example:

Input: 20 15 26 2 98 6

Sorted array: 2 6 15 20 26 98

So ranks are: 2->1, 6->2, 15->3, 20->4, 26->5, 98->6

Output: 4 3 5 1 6 2

Input: 1 5 8 15 8 25 9

Sorted array: 1 5 8 8 9 15 25

Duplicate 8 gets the same rank.

Output: 1 2 3 5 3 6 4

---

## Approach 1: Brute Force Approach

### Algorithm

For each element, we count how many **unique elements are smaller than it**.

Steps:

1. Initialize an empty result array.
2. For each element in the array:
  - o Create an empty set.
  - o Traverse the array again.
  - o If another element is smaller, insert it into the set.
3. Rank = size of the set + 1.
4. Store the rank in the result array.
5. Return the result.

### Code

```
#include <bits/stdc++.h>

using namespace std;

class Solution {

public:

 vector<int> replaceWithRank(vector<int>& arr) {

 vector<int> rankArr;
```

```

 for(int i = 0; i < arr.size(); i++) {
 unordered_set<int> smaller;

 for(int j = 0; j < arr.size(); j++) {
 if(arr[j] < arr[i]) {
 smaller.insert(arr[j]);
 }
 }

 rankArr.push_back(smaller.size() + 1);
 }

 return rankArr;
 }

};

int main() {
 vector<int> arr = {20, 15, 26, 2, 98, 6};
 Solution sol;
 vector<int> res = sol.replaceWithRank(arr);

 for(int x : res)
 cout << x << " ";
 return 0;
}

```

## Complexity Analysis

- Time Complexity:  $O(N^2)$  because we compare every element with every other element.
  - Space Complexity:  $O(N)$  due to the set and result array.
- 

## Approach 2: Optimal Approach

### Algorithm

We sort the array and assign ranks in increasing order, skipping duplicates.

Steps:

1. Create a copy of the original array.
2. Sort the copied array.
3. Use a map to store the rank of each unique element.
4. Traverse the sorted array and assign ranks starting from 1.
5. Traverse the original array and replace each element using the rank map.
6. Return the result array.

### Code

```
#include <bits/stdc++.h>

using namespace std;

class Solution {

public:

 vector<int> replaceWithRank(vector<int>& arr) {
```

```

vector<int> sortedArr = arr;

sort(sortedArr.begin(), sortedArr.end());

unordered_map<int,int> rankMap;

int rank = 1;

for(int num : sortedArr) {

 if(rankMap.find(num) == rankMap.end()) {

 rankMap[num] = rank;

 rank++;

 }

}

vector<int> result;

for(int num : arr) {

 result.push_back(rankMap[num]);

}

return result;

};

int main() {

 vector<int> arr = {1, 5, 8, 15, 8, 25, 9};

```

```

Solution sol;

vector<int> res = sol.replaceWithRank(arr);

for(int x : res)

 cout << x << " ";

return 0;

}

```

## Complexity Analysis

- Time Complexity:  $O(N \log N)$  due to sorting.
- Space Complexity:  $O(N)$  for the map and copied array.

## Approach 3: Heap Based Approach (Using Max Heap)

### Algorithm

We assign ranks by always picking the **maximum element first**.

Steps:

1. Create a max heap storing (value, index) pairs.
2. Push all array elements with their indices into the heap.
3. Initialize rank = n.
4. While heap is not empty:
  - Pop the maximum element.
  - Assign current rank to its original index.

- Decrease rank.
5. The array gets transformed into ranks directly.

### Code

```
#include <bits/stdc++.h>

using namespace std;

void transform(vector<int>& input) {

 priority_queue<pair<int, int>> maxHeap;

 for(int i = 0; i < input.size(); i++) {
 maxHeap.push({input[i], i});
 }

 int rank = input.size();

 while(!maxHeap.empty()) {
 input[maxHeap.top().second] = rank;
 maxHeap.pop();
 rank--;
 }
}
```

```

int main() {

 vector<int> input = {10, 8, 15, 12, 6, 20, 1};

 transform(input);

 for(int x : input)

 cout << x << " ";

 return 0;

}

```

### Complexity Analysis

- Time Complexity:  $O(N \log N)$
- Space Complexity:  $O(N)$

## 9. Task Scheduler

You are given a list of tasks represented by uppercase English letters (A to Z) and an integer  $n$  which represents the cooldown period between two same tasks.

Each task takes exactly 1 unit of CPU time.

The same task must be separated by at least  $n$  intervals, during which the CPU can either execute another task or stay idle.

Return the **minimum number of CPU intervals** required to finish all tasks.

Example:

Input: tasks = ["A", "A", "A", "B", "B", "B"], n = 2

One valid order: A -> B -> idle -> A -> B -> idle -> A -> B

Total intervals = 8

Input: tasks = ["A", "C", "A", "B", "D", "B"], n = 1

One valid order: A -> B -> C -> D -> A -> B

Total intervals = 6

---

## Approach: Simulation Using Max Heap

### Algorithm

We simulate task execution while respecting the cooldown constraint.

Steps:

1. Count the frequency of each task.
2. Push all task frequencies into a max heap.
3. While the heap is not empty:
  - o Create a temporary list for tasks used in the current cycle.
  - o Run a cycle of length n + 1.
  - o In each slot:
    - If heap is not empty, pop the task with highest frequency.
    - Execute it, decrease its count, and store it if it still remains.
    - Increase total time.
  - o Push remaining tasks from the temporary list back into the heap.
  - o If heap is not empty, add idle time for unused slots in the cycle.
4. Return total time.

This ensures that the same task is always separated by at least n intervals.

**Sabse zyada frequent task ko pehle schedule karo,  
kyunki wahi sabse zyada idle create karta hai.**

**Isliye:**

- **Max Heap (highest frequency first)**
- **Cycle = n + 1** (ek task + uske baad n cooldown slots)

**Code**

```
#include <bits/stdc++.h>

using namespace std;

class Solution {

public:

 int leastInterval(vector<char>& tasks, int n) {

 unordered_map<char,int> freq;

 for(char t : tasks)

 freq[t]++;

 priority_queue<int> maxHeap;

 for(auto it : freq)

 maxHeap.push(it.second);

 int time = 0;
```

```

while(!maxHeap.empty()) {

 vector<int> temp;

 int cycle = n + 1;

 int i = 0;

 while(i < cycle && !maxHeap.empty()) {

 int cnt = maxHeap.top();

 maxHeap.pop();

 cnt--;

 if(cnt > 0)

 temp.push_back(cnt);

 time++;

 i++;

 }

 for(int x : temp)

 maxHeap.push(x);

}

if(maxHeap.empty())

 break;

time += (cycle - i);

}

```

```

 return time;
 }

};

int main() {
 Solution obj;
 vector<char> tasks = {'A', 'A', 'A', 'B', 'B', 'B'};
 int n = 2;
 cout << obj.leastInterval(tasks, n);
 return 0;
}

```

### Complexity Analysis

- Time Complexity:  $O(N \log K)$   
N is the number of tasks, K is the number of unique tasks.
- Space Complexity:  $O(K)$   
For frequency map and max heap.

## 10. Hands of Straights

You are given an array `hand` where each element represents the value of a card. You are also given an integer `groupSize`.

Your task is to check whether it is possible to divide all cards into groups such that:

- Each group has exactly `groupSize` cards
- The values in each group are **consecutive integers**

Example:

Input: hand = [1, 2, 3, 6, 2, 3, 4, 7, 8], `groupSize` = 3

One valid grouping is [1, 2, 3] , [2, 3, 4] , [6, 7, 8]

So the answer is true.

Input: hand = [1, 2, 3, 4, 5], `groupSize` = 4

It is not possible to divide all cards into groups of 4 consecutive numbers.

So the answer is false.

---

## Algorithm

We try to always form groups starting from the **smallest available card**.

Steps:

1. If the total number of cards is not divisible by `groupSize`, return false.
2. Store the frequency of each card in a sorted map.
3. Iterate through the map from the smallest card.
4. For each card with frequency greater than 0:
  - Let this card be the start of a group.
  - Try to form count groups of size `groupSize` starting from this card.
  - For each required consecutive card:
    - If it does not exist or its frequency is less than count, return false.
    - Otherwise, decrease its frequency by count.
5. If all groups are formed successfully, return true.

This greedy approach works because starting from the smallest card avoids breaking the consecutive order.

Hamesha sabse chhote available card se group banana shuru karo.

- Agar tum smallest se start nahi karoge,
- to wo chhota card baad me kisi group me fit hi nahi hogा.

Isi wajah se map use hota hai (sorted order).

### Code

```
#include <bits/stdc++.h>

using namespace std;

class Solution {

public:

 bool isNStraightHand(vector<int>& hand, int groupSize) {

 if (hand.size() % groupSize != 0)

 return false;

 map<int,int> freq;

 for(int card : hand)

 freq[card]++;

 for(auto it = freq.begin(); it != freq.end(); it++) {
```

```

 if(it->second == 0) continue;

 int start = it->first;
 int count = it->second;

 }

 for(int i = 0; i < groupSize; i++) {
 if(freq[start + i] < count)
 return false;

 freq[start + i] -= count;
 }

 return true;
}

};

int main() {
 Solution sol;

 vector<int> hand1 = {1,2,3,6,2,3,4,7,8};
 int groupSize1 = 3;
 cout << sol.isNStraightHand(hand1, groupSize1) << endl;

 vector<int> hand2 = {1,2,3,4,5};

```

```

int groupSize2 = 4;

cout << sol.isNStraightHand(hand2, groupSize2) << endl;

return 0;
}

```

### Complexity Analysis

- Time Complexity:  $O(N \log N)$   
Inserting cards into the map takes  $O(N \log N)$ , and forming groups also involves map access.
- Space Complexity:  $O(N)$   
The map stores frequencies of all cards.

## 11. Design Twitter

You need to design a simplified version of Twitter where users can post tweets, follow or unfollow other users, and view the 10 most recent tweets in their news feed.

The news feed should contain tweets from the user themselves and from users they follow, ordered from most recent to least recent.

Example:

Input:

`postTweet(1,2), postTweet(2,6), getNewsFeed(1)`

Output: [2]

After `follow(1,2)`

`getNewsFeed(1)` returns [6,2]

## Approach

### Algorithm

The main challenge is efficiently retrieving the **10 most recent tweets** from multiple users.

Steps:

1. Maintain a map that stores tweets for each user.  
Each tweet is stored as (timestamp, tweetId) so that recency is tracked.
2. Maintain another map that stores which users a given user follows.
3. Use a global timestamp that increases every time a tweet is posted.
4. For postTweet:
  - Add the tweet with the current timestamp to the user's tweet list.
5. For follow and unfollow:
  - Update the follower's follow set.
6. For getNewsFeed:
  - Use a **min-heap of size at most 10**.
  - Push tweets of the user and all followed users into the heap.
  - If heap size exceeds 10, remove the oldest tweet.
  - Finally, extract tweet IDs from the heap in reverse order to get most recent first.
7. Return the result list.

This works because we only keep the top 10 most recent tweets at any time.

### Code

```
#include <bits/stdc++.h>

using namespace std;
```

```

class Twitter {

private:
 unordered_map<int, vector<pair<int, int>>> tweets;
 unordered_map<int, unordered_set<int>> following;
 int time;

public:
 Twitter() {
 time = 0;
 }

 void postTweet(int userId, int tweetId) {
 tweets[userId].push_back({time++, tweetId});
 }

 vector<int> getNewsFeed(int userId) {
 priority_queue<pair<int, int>, vector<pair<int, int>>, greater<>> pq;
 for(auto &t : tweets[userId]) {
 pq.push(t);
 if(pq.size() > 10) pq.pop();
 }
 }
}

```

```

 for(int followee : following[userId]) {
 for(auto &t : tweets[followee]) {
 pq.push(t);
 if(pq.size() > 10) pq.pop();
 }
 }

 vector<int> res;
 while(!pq.empty()) {
 res.push_back(pq.top().second);
 pq.pop();
 }
 reverse(res.begin(), res.end());
 return res;
 }

 void follow(int followerId, int followeeId) {
 following[followerId].insert(followeeId);
 }

 void unfollow(int followerId, int followeeId) {
 following[followerId].erase(followeeId);
 }
}

```

```
}

};

int main() {

 Twitter twitter;

 twitter.postTweet(1, 2);

 twitter.postTweet(2, 6);

 vector<int> a = twitter.getNewsFeed(1);

 for(int x : a) cout << x << " ";

 cout << endl;

 twitter.follow(1, 2);

 vector<int> b = twitter.getNewsFeed(1);

 for(int x : b) cout << x << " ";

 cout << endl;

 twitter.unfollow(1, 2);

 twitter.postTweet(1, 7);

 vector<int> c = twitter.getNewsFeed(1);
```

```
 for(int x : c) cout << x << " ";
 cout << endl;
 return 0;
}
```

## Complexity Analysis

- Time Complexity:
    - `postTweet`:  $O(1)$
    - `follow`:  $O(1)$
    - `unfollow`:  $O(1)$
    - `getNewsFeed`:  $O(T \log 10) \approx O(T)$ , where  $T$  is total tweets checked
  - Space Complexity:
    - $O(N)$  for storing all tweets
    - $O(U)$  for follow relationships
    - Heap uses  $O(10)$  extra space

# 12. Connect N Ropes with Minimum Cost

You are given an array  $\text{arr}[]$  where each element represents the length of a rope. Your task is to connect all ropes into a single rope with the **minimum total cost**. The cost of connecting two ropes is equal to the **sum of their lengths**.

Example:

Input:  $\text{arr} = [4, 3, 2, 6]$

One optimal way is:

- Connect 2 and 3 → cost 5, ropes become  $[4, 5, 6]$
- Connect 4 and 5 → cost 9, ropes become  $[6, 9]$
- Connect 6 and 9 → cost 15  
Total cost =  $5 + 9 + 15 = 29$

If there is only one rope, no cost is required.

---

## Approach 1: Naive Approach (Greedy with Repeated Sorting)

### Algorithm

The idea is to always connect the two shortest ropes first.

Steps:

1. While more than one rope exists:
2. Sort the array.
3. Take the two smallest ropes.
4. Remove them from the array.
5. Add their sum to total cost.
6. Insert the combined rope back into the array.
7. Continue until only one rope remains.

### Code

```
#include <bits/stdc++.h>

using namespace std;

int minCost(vector<int>& arr) {

 int totalCost = 0;

 while (arr.size() > 1) {

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

 int first = arr[0];

 int second = arr[1];

 arr.erase(arr.begin());
 arr.erase(arr.begin());

 int cost = first + second;

 totalCost += cost;

 arr.push_back(cost);

 }

 return totalCost;
}
```

```

int main() {

 vector<int> ropes = {4, 3, 2, 6};

 cout << minCost(ropes);

 return 0;

}

```

## Complexity Analysis

- Time Complexity:  $O(n^2 \log n)$   
Sorting is done repeatedly for each merge.
  - Space Complexity:  $O(n)$   
Used for storing the ropes.
- 

## Approach 2: Optimal Approach (Greedy with Min Heap)

### Algorithm

This approach improves efficiency using a **min heap**.

Steps:

1. Insert all rope lengths into a min heap.
2. Initialize total cost as 0.
3. While heap size is greater than 1:
  - Extract the two smallest ropes.
  - Add their sum to total cost.
  - Push the combined rope back into the heap.
4. Return the total cost.

This works because always merging the two smallest ropes minimizes the cost at each step.

### Code

```
#include <bits/stdc++.h>

using namespace std;

int minCost(vector<int>& arr) {

 priority_queue<int, vector<int>, greater<int>> pq(arr.begin(),
arr.end());

 int res = 0;

 while (pq.size() > 1) {

 int first = pq.top(); pq.pop();

 int second = pq.top(); pq.pop();

 int cost = first + second;

 res += cost;

 pq.push(cost);

 }

 return res;
}

int main() {

 vector<int> ropes = {4, 3, 2, 6};
```

```

 cout << minCost(ropes);

 return 0;

}

```

### Complexity Analysis

- Time Complexity:  $O(n \log n)$   
Each heap operation takes  $\log n$ , done  $n$  times.
- Space Complexity:  $O(n)$   
Min heap stores all rope lengths.

## 13. Kth Largest Element in a Stream of Running Integers

You are given an integer  $k$  and an initial stream of integers.

You need to design a class that can continuously return the **kth largest element** after inserting new values into the stream.

The  $k$ th largest element is based on **sorted order**, not distinct values.

Example:

Initial stream = [1, 2, 3, 4],  $k = 3$   
 Sorted order = [1, 2, 3, 4]  
 3rd largest = 3

After adding new values, the stream grows and the  $k$ th largest must be updated each time.

---

### Approach: Min Heap of Size $K$

#### Algorithm

To efficiently track the kth largest element, we maintain a **min heap of size k**.

Key idea:

- The heap always stores the **k largest elements seen so far**.
- The smallest element in this heap is the **kth largest overall**.

Steps:

1. Create a min heap.
2. Insert elements from the initial stream one by one into the heap.
3. If heap size becomes greater than k, remove the smallest element.
4. When a new value is added:
  - Insert it into the heap.
  - If heap size exceeds k, remove the smallest element.
5. The top of the min heap is always the kth largest element.

## Code

```
#include <bits/stdc++.h>

using namespace std;

class KthLargest {

 priority_queue<int, vector<int>, greater<int>> minHeap;

 int k;

public:

 KthLargest(int k, vector<int>& nums) {
```

```
this->k = k;

for(int num : nums) {
 minHeap.push(num);
 if(minHeap.size() > k)
 minHeap.pop();
}
```

```
int add(int val) {
 minHeap.push(val);
 if(minHeap.size() > k)
 minHeap.pop();
 return minHeap.top();
}
};
```

```
int main() {
 vector<int> nums = {1, 2, 3, 4};
 KthLargest obj(3, nums);

 cout << obj.add(5) << endl; // 3
 cout << obj.add(2) << endl; // 3
 cout << obj.add(7) << endl; // 4
}
```

```
 return 0;
}

}
```

## Complexity Analysis

- Time Complexity:  $O(\log k)$  per insertion  
Each add operation involves heap insertion and possible deletion.
- Space Complexity:  $O(k)$   
The heap stores only  $k$  elements at any time.

# 14. Maximum Sum Combination

You are given two integer arrays `nums1` and `nums2`, and an integer  $k$ .

A valid combination is formed by picking **one element from `nums1` and one element from `nums2`** and adding them.

Your task is to return the **top  $k$  maximum sum combinations in non-increasing order**.

Example:

Input: `nums1 = [7, 3]`, `nums2 = [1, 6]`,  $k = 2$

Possible sums:  $7+1=8$ ,  $7+6=13$ ,  $3+1=4$ ,  $3+6=9$

Sorted sums: `[13, 9, 8, 4]`

Output: `[13, 9]`

---

## Approach 1: Brute Force Approach

### Algorithm

We generate **all possible pair sums** and then select the largest  $k$  values.

Steps:

1. Create an empty array to store all sums.
2. Loop through every element in nums1.
3. For each element in nums1, loop through every element in nums2.
4. Add the pair sum to the list.
5. Sort the list in descending order.
6. Return the first k elements.

### Code

```
#include <bits/stdc++.h>

using namespace std;

class Solution {

public:

 vector<int> maxCombinations(vector<int>& nums1, vector<int>&
nums2, int k) {

 vector<int> allSums;

 for(int i = 0; i < nums1.size(); i++) {
 for(int j = 0; j < nums2.size(); j++) {
 allSums.push_back(nums1[i] + nums2[j]);
 }
 }

 }
}
```

```

 sort(allSums.begin(), allSums.end(), greater<int>());

 vector<int> result(allSums.begin(), allSums.begin() + k);

 return result;
}

};

int main() {
 Solution obj;

 vector<int> nums1 = {7, 3};

 vector<int> nums2 = {1, 6};

 int k = 2;

 vector<int> res = obj.maxCombinations(nums1, nums2, k);

 for(int x : res) cout << x << " ";

 return 0;
}

```

## Complexity Analysis

- Time Complexity:  $O(n*m \log(n*m))$   
All  $n*m$  sums are generated and sorted.
- Space Complexity:  $O(n*m)$   
All pair sums are stored.

---

## Approach 2: Optimal Approach (Using Max Heap)

### Algorithm

Instead of generating all combinations, we only explore the **largest possible sums first**.

Steps:

1. Sort both arrays in descending order.
2. Use a max heap storing (sum, i, j) where  
$$\text{sum} = \text{nums1}[i] + \text{nums2}[j].$$
3. Push the initial combination (0, 0) into the heap.
4. Maintain a set to avoid revisiting index pairs.
5. Repeat k times:
  - o Pop the maximum sum from heap and add it to result.
  - o Push (i+1, j) and (i, j+1) if valid and not visited.
6. Return the result.

This ensures we only process necessary combinations.

### Code

```
#include <bits/stdc++.h>

using namespace std;

class Solution {

public:

 vector<int> maxCombinations(vector<int>& nums1, vector<int>& nums2, int k) {
```

```

sort(nums1.begin(), nums1.end(), greater<int>());
sort(nums2.begin(), nums2.end(), greater<int>());

priority_queue<tuple<int, int, int>> maxHeap;

set<pair<int, int>> visited;

maxHeap.push({nums1[0] + nums2[0], 0, 0});

visited.insert({0, 0});

vector<int> result;

while(k-- && !maxHeap.empty()) {

 auto [sum, i, j] = maxHeap.top();

 maxHeap.pop();

 result.push_back(sum);

 if(i + 1 < nums1.size() && !visited.count({i+1, j})) {

 maxHeap.push({nums1[i+1] + nums2[j], i+1, j});

 visited.insert({i+1, j});

 }

 if(j + 1 < nums2.size() && !visited.count({i, j+1})) {

```

```

 maxHeap.push({nums1[i] + nums2[j+1], i, j+1});

 visited.insert({i, j+1});

 }

 return result;
}

};

int main() {
 Solution sol;

 vector<int> nums1 = {7, 3};

 vector<int> nums2 = {1, 6};

 int k = 2;

 vector<int> res = sol.maxCombinations(nums1, nums2, k);

 for(int x : res) cout << x << " ";

 return 0;
}

```

## Complexity Analysis

- Time Complexity:  $O(k \log k)$   
Each heap operation costs  $\log k$ , repeated  $k$  times.

- Space Complexity:  $O(k)$   
Heap and visited set store at most  $k$  elements.

### Har iteration:

1. Heap se **current maximum sum** nikalo
  2. Uske **neighbors** push karo:
    - o  $(i+1, j)$
    - o  $(i, j+1)$
- 

## Dry Run

### Input

nums1 = [7, 3]

nums2 = [6, 1]

$k = 2$

---

### Initial

Heap:

(13, 0, 0) // 7 + 6

Result: []

---

### Iteration 1

Pop:

(13, 0, 0)

Result:

[13]

Push neighbors:

- $(1, 0) \rightarrow 3 + 6 = 9$
- $(0, 1) \rightarrow 7 + 1 = 8$

Heap:

$(9, 1, 0), (8, 0, 1)$

---

## Iteration 2

Pop:

$(9, 1, 0)$

Result:

[13, 9]

$k = 0 \rightarrow \text{stop}$

---

## Final Answer

[13, 9]

# 15. Find Median from Data Stream

You need to design a data structure that supports adding numbers from a stream and finding the median at any point.

The median is the middle element in sorted order.

If the total number of elements is even, the median is the average of the two middle elements.

Example:

Input: addNum(1), addNum(2), addNum(3)

Sorted: [1, 2, 3] → Median = 2

Input: addNum(1), addNum(6)

Sorted: [1, 6] → Median =  $(1+6)/2 = 3.5$

---

## Approach 1: Brute Force Approach

### Algorithm

We store all numbers and sort them whenever the median is requested.

Steps:

1. Maintain a list of all numbers.
2. On addNum, insert the number into the list.
3. On findMedian:
  - o Sort the list.
  - o If size is odd, return the middle element.
  - o If size is even, return the average of the two middle elements.

### Code

```
#include <bits/stdc++.h>
```

```
using namespace std;

class MedianFinder {

 vector<int> nums;

public:
 MedianFinder() {

 }

 void addNum(int num) {

 nums.push_back(num);

 }

 double findMedian() {

 sort(nums.begin(), nums.end());

 int n = nums.size();

 if(n % 2 == 1)

 return nums[n / 2];

 return (nums[n / 2 - 1] + nums[n / 2]) / 2.0;

 }

};
```

```

int main() {

 MedianFinder mf;

 mf.addNum(1);

 mf.addNum(2);

 mf.addNum(3);

 cout << mf.findMedian() << endl;

 return 0;

}

```

## Complexity Analysis

- Time Complexity:  $O(N \log N)$  for `findMedian` due to sorting.
  - Space Complexity:  $O(N)$  to store all elements.
- 

## Approach 2: Optimal Approach (Two Heaps)

### Algorithm

We use two heaps to maintain balance:

- A **max heap** for the smaller half.
- A **min heap** for the larger half.

Rules:

- The size of max heap is either equal to or one greater than min heap.

- All elements in max heap are  $\leq$  elements in min heap.

Steps:

1. Insert the new number into max heap.
2. Move the largest element from max heap to min heap.
3. If min heap has more elements, move its top back to max heap.
4. To find median:
  - If both heaps have equal size, return average of their tops.
  - Otherwise, return the top of max heap.

## Code

```
#include <bits/stdc++.h>

using namespace std;

class MedianFinder {
 priority_queue<int> maxHeap;
 priority_queue<int, vector<int>, greater<int>> minHeap;

public:
 MedianFinder() {
 }
}
```

```

void addNum(int num) {

 maxHeap.push(num);

 minHeap.push(maxHeap.top());

 maxHeap.pop();

 if(minHeap.size() > maxHeap.size()) {

 maxHeap.push(minHeap.top());

 minHeap.pop();

 }

}

double findMedian() {

 if(maxHeap.size() == minHeap.size())

 return (maxHeap.top() + minHeap.top()) / 2.0;

 else

 return maxHeap.top();

};

int main() {

 MedianFinder mf;

 mf.addNum(1);

 mf.addNum(6);

```

```

cout << mf.findMedian() << endl;
mf.addNum(3);

cout << mf.findMedian() << endl;

return 0;

}

```

## Complexity Analysis

- Time Complexity:
  - addNum:  $O(\log N)$
  - findMedian:  $O(1)$
- Space Complexity:  $O(N)$  for storing elements in heaps.

### **Numbers ko do halves me baant do**

left half (chhotा) → maxHeap  
 right half (bada) → minHeap

Taaki beech ka element turant mil jaaye.

---

## 🔑 Do heaps ka role

### 1 maxHeap (left half)

- Isme chhote elements
- Top = left half ka sabse bada
- Median yahin se aata hai (odd case)

## 2 minHeap (right half)

- Isme bade elements
  - Top = right half ka sabse chhota
- 



## Invariants (ye hamesha true rehna chahiye)

1. `maxHeap.size() == minHeap.size()`  
**ya**  
`maxHeap.size() == minHeap.size() + 1`
2. `maxHeap.top() ≤ minHeap.top()`

Agar ye dono sahi hain → median nikalna trivial.

---

### ◆ addNum(num) ka REAL logic

Code:

```
void addNum(int num) {
 maxHeap.push(num);
 minHeap.push(maxHeap.top());
 maxHeap.pop();

 if(minHeap.size() > maxHeap.size()) {
 maxHeap.push(minHeap.top());
 minHeap.pop();
 }
}
```

Explanation:

### Step 1: Pehle maxHeap me daalo

```
maxHeap.push(num);
```

👉 Kyunki initially hum assume karte hain:

naya number left half ka hissa ho sakta hai

---

### Step 2: maxHeap ka largest minHeap me bhejo

```
minHeap.push(maxHeap.top());
maxHeap.pop();
```

👉 Iska matlab:

- Left half me jo **sabse bada** hai
- Usse right half me shift kar do

Isse ensure hota hai:

```
maxHeap ke saare elements ≤ minHeap ke saare elements
```

---

### Step 3: Size balance karo

```
if(minHeap.size() > maxHeap.size()) {
 maxHeap.push(minHeap.top());
 minHeap.pop();
}
```



Rule:

- maxHeap ko **equal ya ek zyada** element chahiye

Agar minHeap bada ho gaya → ek element wapas bhej do.

---



## findMedian() ka logic

```
if(size equal)
 median = (maxHeap.top + minHeap.top) / 2
else
 median = maxHeap.top
```

Kyun?

- Odd count → beech ka element left half me hota hai
  - Even count → beech ke do elements = dono heaps ke top
- 



## FULL DRY RUN

### Start

```
maxHeap = []
```

```
minHeap = []
```

---

## **addNum(1)**

Step 1:

```
maxHeap = [1]
```

Step 2:

```
minHeap = [1]
```

```
maxHeap = []
```

Step 3 (rebalance):

```
maxHeap = [1]
```

```
minHeap = []
```

Median:

1

---

## **addNum(6)**

Step 1:

```
maxHeap = [6, 1]
```

Step 2:

minHeap = [6]

maxHeap = [1]

Sizes equal → ok

Median:

$$(1 + 6) / 2 = 3.5$$

---

### **addNum(3)**

Step 1:

maxHeap = [3, 1]

Step 2:

minHeap = [3, 6]

maxHeap = [1]

Step 3 (rebalance):

maxHeap = [3, 1]

minHeap = [6]

Median:

3

# 16. Top K Frequent Elements in an Array

Given an array  $\text{arr} [ ]$  and an integer  $k$ , find the **top k elements with the highest frequency**. If multiple elements have the same frequency, **the larger element should be preferred**.

Example:

Input:  $\text{arr} = [3, 1, 4, 4, 5, 2, 6, 1]$ ,  $k = 2$

Frequencies: 4→2, 1→2, others→1

Output: [4, 1]

Input:  $\text{arr} = [7, 10, 11, 5, 2, 5, 5, 7, 11, 8, 9]$ ,  $k = 4$

Output: [5, 11, 7, 10]

---

## Approach 1: Naive Approach (Hash Map + Sorting)

### Algorithm

1. Count frequency of each element using a hash map.
2. Store (element, frequency) pairs in a vector.
3. Sort the vector:
  - Higher frequency first.
  - If frequency is same, larger element first.
4. Return the first  $k$  elements.

### Code

```
#include <bits/stdc++.h>
```

```

using namespace std;

//compare(a, b) ka matlab hota hai:

// "Kya a ko b se pehle aana chahiye?"
• //true → a pehle aayega
• //false → b pehle aayega

static bool compare(pair<int,int>& p1, pair<int,int>& p2) {

 if(p1.second == p2.second)

 return p1.first > p2.first;

 return p1.second > p2.second;

}

vector<int> topKFreq(vector<int>& arr, int k) {

 unordered_map<int,int> mp;

 for(int x : arr) mp[x]++;

 vector<pair<int,int>> freq(mp.begin(), mp.end());

 sort(freq.begin(), freq.end(), compare);

 vector<int> res;

 for(int i = 0; i < k; i++)

 res.push_back(freq[i].first);

 return res;
}

```

```
}
```

```
int main() {

 vector<int> arr = {3,1,4,4,5,2,6,1};

 int k = 2;

 vector<int> res = topKFreq(arr, k);

 for(int x : res) cout << x << " ";

 return 0;

}
```

## Complexity Analysis

- Time Complexity:  $O(n + d \log d)$
- Space Complexity:  $O(d)$   
( $d$  = number of distinct elements)

---

## Approach 2: Expected Approach 1 (Hash Map + Min Heap)

### Algorithm

1. Count frequencies using a hash map.
2. Use a **min heap** of size  $k$  storing (frequency, element).
3. Push each map entry into the heap.
4. If heap size exceeds  $k$ , remove the smallest frequency element.
5. Extract elements from heap in reverse order.

## Code

```
#include <bits/stdc++.h>

using namespace std;

vector<int> topKFreq(vector<int>& arr, int k) {

 unordered_map<int,int> mp;
 for(int x : arr) mp[x]++;
 priority_queue<pair<int,int>, vector<pair<int,int>>, greater<pair<int,int>>> pq;

 for(auto p : mp) {
 pq.push({p.second, p.first});
 if(pq.size() > k)
 pq.pop();
 }

 vector<int> res(k);
 for(int i = k-1; i >= 0; i--) {
 res[i] = pq.top().second;
 pq.pop();
 }

 return res;
}
```

```
}
```

```
int main() {

 vector<int> arr = {3,1,4,4,5,2,6,1};

 int k = 2;

 vector<int> res = topKFreq(arr, k);

 for(int x : res) cout << x << " ";

 return 0;

}
```

### Complexity Analysis

- Time Complexity:  $O(n + k \log k)$
- Space Complexity:  $O(d)$

---

## Approach 3: Expected Approach 2 (Counting Sort / Buckets)

### Algorithm

1. Count frequency of each element.
2. Find maximum frequency.
3. Create buckets where index = frequency.
4. Store elements in corresponding buckets.
5. Traverse buckets from high to low frequency.
6. If same frequency, sort elements in descending order.

7. Collect first k elements.

### Code

```
#include <bits/stdc++.h>

using namespace std;

vector<int> topKFreq(vector<int>& arr, int k) {

 unordered_map<int,int> freq;
 for(int x : arr) freq[x]++;
}

int maxFreq = 0;
for(auto p : freq)
 maxFreq = max(maxFreq, p.second);

vector<vector<int>> buckets(maxFreq + 1);
for(auto p : freq)
 buckets[p.second].push_back(p.first);

vector<int> res;
for(int i = maxFreq; i >= 1; i--) {
 sort(buckets[i].begin(), buckets[i].end(), greater<int>());
 for(int x : buckets[i]) {
 res.push_back(x);
 }
}
```

```

 if(res.size() == k)

 return res;

 }

}

return res;

}

int main() {

 vector<int> arr = {3,1,4,4,5,2,6,1};

 int k = 2;

 vector<int> res = topKFreq(arr, k);

 for(int x : res) cout << x << " ";

 return 0;
}

```

## Complexity Analysis

- Time Complexity:  $O(n \log n)$
- Space Complexity:  $O(n)$

## Approach 4: Alternate Approach (Quick Select)

### Algorithm

1. Count frequencies.

2. Store distinct elements.
3. Use QuickSelect to position top k frequent elements at the end.
4. Sort only those k elements:
  - o Higher frequency first.
  - o If equal frequency, larger value first.
5. Return the result.

### Code

```
#include <bits/stdc++.h>

using namespace std;

int partition(int left, int right, int pivotIdx,
 vector<int>& distinct, unordered_map<int,int>& mp) {

 int pivotFreq = mp[distinct[pivotIdx]];

 int pivotVal = distinct[pivotIdx];

 swap(distinct[pivotIdx], distinct[right]);

 int j = left;
 for(int i = left; i < right; i++) {
 if(mp[distinct[i]] < pivotFreq ||
 (mp[distinct[i]] == pivotFreq && distinct[i] < pivotVal)) {

```

```

 swap(distinct[i], distinct[j]);

 j++;
 }

}

swap(distinct[j], distinct[right]);

return j;
}

void quickselect(int left, int right, int k,
 vector<int>& distinct, unordered_map<int,int>& mp) {

 if(left >= right) return;

 int pivotIdx = left + rand() % (right - left + 1);

 pivotIdx = partition(left, right, pivotIdx, distinct, mp);

 if(pivotIdx == k) return;
 else if(pivotIdx > k)
 quickselect(left, pivotIdx - 1, k, distinct, mp);
 else
 quickselect(pivotIdx + 1, right, k, distinct, mp);
}

vector<int> topKFreq(vector<int>& arr, int k) {

```

```

unordered_map<int,int> mp;

for(int x : arr) mp[x]++;

vector<int> distinct;

for(auto p : mp) distinct.push_back(p.first);

int n = distinct.size();

quickselect(0, n-1, n-k, distinct, mp);

sort(distinct.begin()+n-k, distinct.end(), [&](int a, int b){

 if(mp[a] != mp[b]) return mp[a] > mp[b];

 return a > b;

});

vector<int> res;

for(int i = n-k; i < n; i++)

 res.push_back(distinct[i]);

return res;

}

int main() {

vector<int> arr = {3,1,4,4,5,2,6,1};

int k = 2;

```

```
vector<int> res = topKFreq(arr, k);

for(int x : res) cout << x << " ";

return 0;

}
```

## Complexity Analysis

- Time Complexity: Average  $O(n)$  (QuickSelect), worst  $O(n^2)$
- Space Complexity:  $O(n)$

# **Greedy Algorithms**

# 1. Assign Cookies

You are given two arrays:

- student[] where student[i] is the minimum cookie size needed by the i-th student
- cookie[] where cookie[j] is the size of the j-th cookie

Each student can receive **at most one cookie**, and a cookie can be assigned only if  $\text{cookie}[j] \geq \text{student}[i]$ .

Your task is to **maximize the number of students who receive a cookie**.

Example:

Input: student = [1, 2, 3], cookie = [1, 1]

Only one student can be satisfied, so output is 1.

Input: student = [1, 2], cookie = [1, 2, 3]

Both students can be satisfied, so output is 2.

---

## Approach 1: Memoization (Top-Down DP)

### Algorithm

We try all possible ways of assigning cookies to students using recursion.

Steps:

1. Sort both student and cookie arrays.
2. Use recursion with two indices:
  - i for current student
  - j for current cookie
3. If the cookie cannot satisfy the student, skip the cookie.
4. If it can satisfy:

- Option 1: Assign the cookie and move to next student and cookie.
  - Option 2: Skip the cookie and try the next one.
5. Use a 2D memo table to store already computed results.

### Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int findContentChildren(vector<int>& student, vector<int>& cookie)
 {
 sort(student.begin(), student.end());
 sort(cookie.begin(), cookie.end());

 vector<vector<int>> memo(student.size(),
vector<int>(cookie.size(), -1));
 return helper(0, 0, student, cookie, memo);
 }

private:
 int helper(int i, int j, vector<int>& student, vector<int>& cookie,
 vector<vector<int>>& memo) {
 if(i >= student.size() || j >= cookie.size())
 return 0;

 if(memo[i][j] != -1)
 return memo[i][j];

 int result = 0;

 if(cookie[j] >= student[i]) {
 result = max(result, 1 + helper(i + 1, j + 1, student,
cookie, memo));
 }
 }
}
```

```

 result = max(result, helper(i, j + 1, student, cookie, memo));
 return memo[i][j] = result;
 }
};

int main() {
 vector<int> student = {1, 2, 3};
 vector<int> cookie = {1, 1};

 Solution sol;
 cout << sol.findContentChildren(student, cookie);
 return 0;
}

```

## Complexity Analysis

- Time Complexity:  $O(n * m)$
  - Space Complexity:  $O(n * m)$  (memo table + recursion stack)
- 

## Approach 2: Tabulation (Bottom-Up DP)

### Algorithm

We build a DP table where  $dp[i][j]$  represents the maximum number of students satisfied starting from  $student[i]$  and  $cookie[j]$ .

Steps:

1. Sort both arrays.
2. Create a  $(n+1) \times (m+1)$  DP table initialized with 0.
3. Fill the table from bottom-right to top-left.
4. At each position:

- Option 1: Skip current cookie.
- Option 2: Assign cookie if possible.

5. Final answer is  $dp[0][0]$ .

### Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int findContentChildren(vector<int>& student, vector<int>& cookie)
{
 int n = student.size();
 int m = cookie.size();

 sort(student.begin(), student.end());
 sort(cookie.begin(), cookie.end());

 vector<vector<int>> dp(n + 1, vector<int>(m + 1, 0));

 for(int i = n - 1; i >= 0; i--) {
 for(int j = m - 1; j >= 0; j--) {
 int skip = dp[i][j + 1];
 int take = 0;
 if(cookie[j] >= student[i])
 take = 1 + dp[i + 1][j + 1];

 dp[i][j] = max(skip, take);
 }
 }
 return dp[0][0];
}

int main() {
 vector<int> student = {1, 2};
 vector<int> cookie = {3, 1};
 Solution sol;
 cout << sol.findContentChildren(student, cookie);
}
```

```

vector<int> cookie = {1, 2, 3};

Solution sol;
cout << sol.findContentChildren(student, cookie);
return 0;
}

```

## Complexity Analysis

- Time Complexity:  $O(n * m)$
  - Space Complexity:  $O(n * m)$
- 

## Approach 3: Optimal Greedy Approach (Two Pointers)

### Algorithm

To maximize satisfied students:

- Give the **smallest sufficient cookie** to the **least greedy student**.

Steps:

1. Sort both student and cookie arrays.
2. Use two pointers:
  - One for students
  - One for cookies
3. If current cookie satisfies current student:
  - Assign it and move both pointers.
4. Otherwise:
  - Move to the next cookie.

5. Count how many students are satisfied.

### Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int findContentChildren(vector<int>& student, vector<int>& cookie)
{
 sort(student.begin(), student.end());
 sort(cookie.begin(), cookie.end());

 int i = 0, j = 0;

 while(i < student.size() && j < cookie.size()) {
 if(cookie[j] >= student[i]) {
 i++;
 }
 j++;
 }
 return i;
};

int main() {
 vector<int> student = {1, 2, 3};
 vector<int> cookie = {1, 1};

 Solution sol;
 cout << sol.findContentChildren(student, cookie);
 return 0;
}
```

### Complexity Analysis

- Time Complexity:  $O(n \log n + m \log m)$  due to sorting

- Space Complexity:  $O(1)$

## 2. Fractional Knapsack Problem : Greedy Approach

You are given  $N$  items, where each item has a value and a weight. You are also given a knapsack with maximum capacity  $W$ .

Your goal is to **maximize the total value** inside the knapsack.

You are allowed to:

- Take an item completely, or
- Take **any fraction** of an item.

This makes the problem different from the 0/1 Knapsack problem.

Example:

If an item has value 120 and weight 30, then taking 20 weight from it gives value 80.

---

### Approach: Greedy (Value-to-Weight Ratio)

#### Algorithm

Since items can be broken into smaller parts, the best strategy is to **always take the item with the highest value per unit weight first**.

Steps:

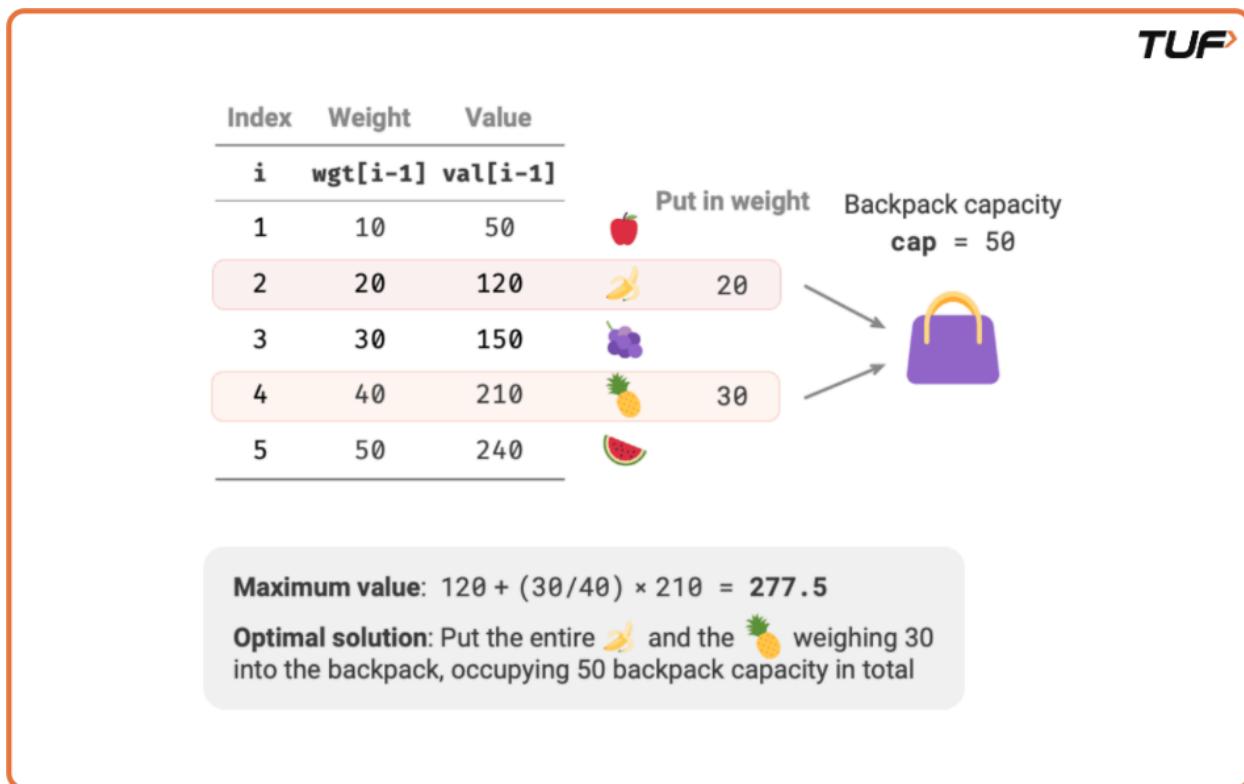
1. For each item, calculate its value-to-weight ratio.
2. Sort all items in **descending order** of value/weight ratio.
3. Initialize current knapsack weight = 0 and total value = 0.

4. Traverse the sorted items:

- If the entire item fits in the remaining capacity, take it fully.
- Otherwise, take only the fraction that fits and add proportional value.

5. Stop once the knapsack is full.

This greedy choice works because taking the highest ratio first always gives maximum value contribution per unit weight.



## Code

```
#include <bits/stdc++.h>
using namespace std;

// Structure to represent an item
struct Item {
 int value;
 int weight;
```

```

};

class Solution {
public:
 // Jis item ka value/weight ratio zyada ho, usse pehle uthao
 static bool comp(Item a, Item b) {
 double r1 = (double)a.value / a.weight;
 double r2 = (double)b.value / b.weight;
 return r1 > r2;
 }
 /comp(a, b) == true ↗ Matlab: a ko b se pehle aana chahiye

 // Function to get maximum value in fractional knapsack
 double fractionalKnapsack(int W, Item arr[], int n) {

 // Sort items by decreasing value/weight ratio
 sort(arr, arr + n, comp);

 int curWeight = 0;
 double finalValue = 0.0;

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

 // If full item can be taken
 if (curWeight + arr[i].weight <= W) {
 curWeight += arr[i].weight;
 finalValue += arr[i].value;
 }
 // Take fractional part
 else {
 int remain = W - curWeight;
 finalValue += (arr[i].value / (double)arr[i].weight) *
remain;
 break;
 }
 }
 return finalValue;
 }
}

```

```

};

int main() {
 int n = 3, W = 50;
 Item arr[n] = {{100,20}, {60,10}, {120,30}};

 Solution obj;
 double ans = obj.fractionalKnapsack(W, arr, n);

 cout << fixed << setprecision(6) << ans;
 return 0;
}

```

---

### Complexity Analysis

- **Time Complexity:**  $O(n \log n)$   
Sorting items by value-to-weight ratio takes  $O(n \log n)$  and iterating once takes  $O(n)$ .
- **Space Complexity:**  $O(1)$   
No extra data structures are used apart from variables.

## 3. Find Minimum Number of Coins

You are given a value  $V$  (in rupees). You have an **infinite supply** of Indian currency denominations:

{1, 2, 5, 10, 20, 50, 100, 500, 1000}.

Your task is to find the **minimum number of coins/notes** required to make the value  $V$ .

Example:

Input:  $V = 70$

We can use 50 + 20, so output is 2.

Input:  $V = 121$

We can use  $100 + 20 + 1$ , so output is 3.

---

## Approach: Greedy Approach

### Algorithm

This problem works perfectly with a greedy strategy because Indian currency denominations are designed to be greedy-friendly.

Steps:

1. Store all coin denominations in an array in increasing order.
2. Start from the **largest denomination**.
3. While the current coin value is less than or equal to  $V$ :
  - o Subtract the coin value from  $V$ .
  - o Store the coin in the answer list.
4. Move to the next smaller denomination.
5. Continue until  $V$  becomes 0.

This ensures we always use the largest possible coin first, minimizing the total count.

---

### Code

```
#include <bits/stdc++.h>
using namespace std;

// Function to compute minimum coins required for value V
vector<int> minCoins(int V) {
 int coins[] = {1, 2, 5, 10, 20, 50, 100, 500, 1000};
 int n = 9;

 vector<int> ans;
```

```

// Traverse from largest to smallest denomination
for (int i = n - 1; i >= 0; i--) {
 while (V >= coins[i]) {
 V -= coins[i];
 ans.push_back(coins[i]);
 }
}
return ans;
}

int main() {
 int V = 121;

 vector<int> result = minCoins(V);

 cout << "Minimum number of coins: " << result.size() << endl;
 cout << "Coins used: ";
 for (int coin : result) {
 cout << coin << " ";
 }
 return 0;
}

```

---

## Complexity Analysis

- **Time Complexity:**  $O(V)$

In the worst case, we may subtract 1 repeatedly (all 1-value coins).

- **Space Complexity:**  $O(V)$

In the worst case, the result array may store  $V$  coins of value 1.

# 4. Lemonade Change

You are given an array `bills` where each element represents the bill a customer pays with.

Each lemonade costs **5\$**. Customers can only pay using **5\$, 10\$, or 20\$** bills.

Initially, you have **no change**.

Your task is to determine whether you can provide the **correct change to every customer in order**.

Return `true` if possible, otherwise return `false`.

Example:

Input: `bills = [5, 5, 5, 10, 20]`

Output: `true`

Because correct change can be given to every customer.

Input: `bills = [5, 5, 10, 10, 20]`

Output: `false`

Because at the last customer, it is not possible to give 15\$ change.

---

## Approach 1: Greedy Counting

### Algorithm

We simulate the process of giving change to each customer using a greedy strategy.

Steps:

1. Maintain two counters:

- `five` → number of 5\$ bills
- `ten` → number of 10\$ bills

2. Traverse the `bills` array one by one:

- If the bill is 5:
  - No change needed, increment `five`.

- If the bill is 10:
    - We must give one 5\$ as change.
    - If `five > 0`, decrement `five` and increment `ten`.
    - Otherwise, return `false`.
  - If the bill is 20:
    - We need to give 15\$ change.
    - Prefer giving one 10\$ and one 5\$ (best option).
    - If not possible, try giving three 5\$ bills.
    - If neither is possible, return `false`.
3. If all customers are processed successfully, return `true`.
- 

### Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 bool lemonadeChange(vector<int>& bills) {
 int five = 0;
 int ten = 0;

 for (int bill : bills) {
 if (bill == 5) {
 five++;
 }
 else if (bill == 10) {
 if (five == 0) return false;
 five--;
 }
 }
 }
}
```

```

 ten++;
 }
 else { // bill == 20
 if (ten > 0 && five > 0) {
 ten--;
 five--;
 }
 else if (five >= 3) {
 five -= 3;
 }
 else {
 return false;
 }
 }
 return true;
}

int main() {
 vector<int> bills = {5, 5, 5, 10, 20};
 Solution obj;
 cout << obj.lemonadeChange(bills);
 return 0;
}

```

---

## Complexity Analysis

- **Time Complexity:**  $O(N)$   
Each customer is processed exactly once.
- **Space Complexity:**  $O(1)$   
Only constant extra variables (`five`, `ten`) are used.

# 5. Valid Parenthesis Checker

You are given a string  $s$  that contains only three characters:

'( ', ')', and '\*'.

The string is considered **valid** if:

- Every '(' has a corresponding ')'.
  - Every ')' has a corresponding '('.
  - '(' always comes before its matching ')'.
    - '\*' can act as:
      - '('
      - ')'
      - an empty string ""

You need to determine whether the string can be interpreted as a valid parenthesis string.

Example:

Input:  $s = (*)$

Here \* can be replaced with '(', making the string (( ), which is valid.

Input:  $s = *()$

No replacement of \* can make the string valid, so the result is false.

---

## Approach 1: Brute Force (Recursion / Backtracking)

### Algorithm

We try **all possible interpretations** of '\*'.

Steps:

1. Use a recursive function with:
  - o current index  $i$
  - o count of currently open brackets  $open$
2. If  $open < 0$ , return false (too many closing brackets).
3. If we reach the end of the string:
  - o return true if  $open == 0$
  - o otherwise return false
4. If current character is:
  - o ' $($ ' → increase  $open$
  - o ' $)$ ' → decrease  $open$
  - o ' $*$ ' → try all three cases:
    - treat as empty
    - treat as ' $($ '
    - treat as ' $)$ '
5. If any recursive path returns true, the string is valid.

### Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 bool isValid(string& s, int i, int open) {
 if (open < 0) return false;

 if (i == s.size())
 return open == 0;
 else
 return isValid(s, i + 1, open + (s[i] == '(' ? 1 : -1));
 }
}
```

```

 return open == 0;

 if (s[i] == '(') {
 return isValid(s, i + 1, open + 1);
 }
 else if (s[i] == ')') {
 return isValid(s, i + 1, open - 1);
 }
 else { // '*'
 return isValid(s, i + 1, open) ||
 isValid(s, i + 1, open + 1) ||
 isValid(s, i + 1, open - 1);
 }
}
};

int main() {
 string s;
 cin >> s;

 Solution sol;
 if (sol.isValid(s, 0, 0))
 cout << "True";
 else
 cout << "False";

 return 0;
}

```

## Complexity Analysis

- Time Complexity:  $O(3^n)$  in the worst case  
Every '\*' creates three recursive branches.
- Space Complexity:  $O(n)$   
Due to recursion stack depth.

## Approach 2: Optimal Greedy Approach

### Algorithm

Instead of checking all combinations, we **track a range of possible open brackets**.

Maintain:

- `minOpen`: minimum possible open brackets
- `maxOpen`: maximum possible open brackets

Steps:

1. Traverse the string from left to right.
2. For each character:
  - ' ( ' → `minOpen++`, `maxOpen++`
  - ' ) ' → `minOpen--`, `maxOpen--`
  - '\*' →
    - `minOpen--` (treat as ' )')
    - `maxOpen++` (treat as ' ( ')
3. If `maxOpen < 0`, return false.
4. Ensure `minOpen` never goes below 0.
5. At the end, if `minOpen == 0`, return true.

### `minOpen`

👉 Is point tak **minimum kitne ' ( open ho sakte hain**  
(optimistically \* ko ' ) ' ya empty maan ke)

### `maxOpen`

👉 Is point tak **maximum kitne ' ( open ho sakte hain**  
(optimistically \* ko ' ( ' maan ke)

This works because we only care whether **some valid interpretation exists**.

### Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 bool isValid(string s) {
 int minOpen = 0, maxOpen = 0;

 for (char c : s) {
 if (c == '(') {
 minOpen++;
 maxOpen++;
 }
 else if (c == ')') {
 minOpen--;
 maxOpen--;
 }
 else { // '*'
 minOpen--;
 maxOpen++;
 }

 if (maxOpen < 0) return false;
 minOpen = max(minOpen, 0);
 }

 return minOpen == 0;
 }
};

int main() {
 string s;
 cin >> s;
```

```

Solution sol;
if (sol.isValid(s))
 cout << "True";
else
 cout << "False";

return 0;
}

```

### Complexity Analysis

- Time Complexity:  $O(n)$   
Single pass through the string.
- Space Complexity:  $O(1)$   
Only constant extra variables are used.

## 6. N Meetings in One Room

You are given two arrays `start[]` and `end[]` of size  $N$ .

For each meeting  $i$ :

- `start[i]` is the starting time
- `end[i]` is the ending time

Only **one meeting** can take place in the room at any time.

Your task is to find the **maximum number of meetings** that can be conducted and **print the order (1-based index)** in which these meetings will be performed.

Example:

Input:

`start = {1, 3, 0, 5, 8, 5}`  
`end = {2, 4, 5, 7, 9, 9}`

Output:

[1, 2, 4, 5]

These meetings do not overlap and allow the maximum count.

---

## Approach: Greedy (Sort by Ending Time)

### Algorithm

To schedule the maximum number of meetings, we always prefer the meeting that **ends earliest**, because it frees the room as soon as possible for the next meetings.

Steps:

1. Store each meeting as a tuple of (end time, start time, meeting index).
2. Sort all meetings by their **end time**.
3. Select the first meeting and set its end time as the last occupied time.
4. Traverse the remaining meetings:
  - o If a meeting's start time is **strictly greater** than the last selected meeting's end time, select it.
  - o Update the last occupied end time.
5. Continue until all meetings are checked.
6. Return the selected meeting indices in order.

This greedy choice guarantees the maximum number of non-overlapping meetings.

---

### Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
```

```

vector<int> maxMeetings(vector<int>& start, vector<int>& end) {
 vector<tuple<int,int,int>> meetings;

 for(int i = 0; i < start.size(); i++) {
 meetings.push_back({end[i], start[i], i + 1});
 }

 sort(meetings.begin(), meetings.end());

 vector<int> result;
 int lastEnd = -1;

 for(auto &m : meetings) {
 int e = get<0>(m);
 int s = get<1>(m);
 int idx = get<2>(m);

 if(s > lastEnd) {
 result.push_back(idx);
 lastEnd = e;
 }
 }
 return result;
}

int main() {
 vector<int> start = {1,3,0,5,8,5};
 vector<int> end = {2,4,5,7,9,9};

 Solution sol;
 vector<int> res = sol.maxMeetings(start, end);

 for(int x : res) cout << x << " ";
 return 0;
}

```

---

## Complexity Analysis

- **Time Complexity:**  $O(N \log N)$   
Sorting meetings by end time dominates the complexity.
- **Space Complexity:**  $O(N)$   
Extra space is used to store meeting tuples and the result list.

# 7. Jump Game – I

You are given an array `nums` where `nums[i]` represents the **maximum jump length** from index `i`.

Starting from index `0`, determine whether you can reach the **last index**.

---

## Intuition (Greedy)

Instead of trying all jump possibilities, track the **farthest index** you can reach at any point.

- If at any index `i`, `i` is **greater than the farthest reachable index**, then this index is unreachable → return `false`.
- Otherwise, update the farthest reachable index using the jump from `i`.

If you finish scanning the array without getting stuck, the last index is reachable.

---

## Algorithm

1. Initialize `maxIndex = 0` (farthest reachable index so far).
2. Traverse the array from left to right.
3. For each index `i`:
  - If `i > maxIndex`, return `false` (we are stuck).

- Update  
     $\maxIndex = \max(\maxIndex, i + \text{nums}[i])$

4. If the loop ends, return true.

---

## Code (C++)

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 bool canJump(vector<int>& nums) {
 int maxIndex = 0;

 for (int i = 0; i < nums.size(); i++) {
 // If current index is not reachable
 if (i > maxIndex) return false;

 // Update farthest reachable index
 maxIndex = max(maxIndex, i + nums[i]);
 }
 return true;
 }

 int main() {
 vector<int> nums = {2, 3, 1, 0, 4};
 Solution sol;

 cout << (sol.canJump(nums) ? "True" : "False") << endl;
 return 0;
 }
}
```

---

## Dry Run (Example)

```
nums = [3, 2, 1, 0, 4]
```

| Index (i) | nums[i] | maxIndex      |
|-----------|---------|---------------|
| 0         | 3       | 3             |
| 1         | 2       | 3             |
| 2         | 1       | 3             |
| 3         | 0       | 3             |
| 4         | 4       | ✗ unreachable |

Since index 4 > maxIndex, return false.

---

## Complexity

- **Time Complexity:**  $O(N)$
- **Space Complexity:**  $O(1)$

# 8. Jump Game 2

You are given a 0-indexed array nums. Each element  $nums[i]$  tells the maximum number of steps you can jump forward from index  $i$ .

You start from index 0 and want to reach the last index  $n-1$  using the minimum number of jumps.

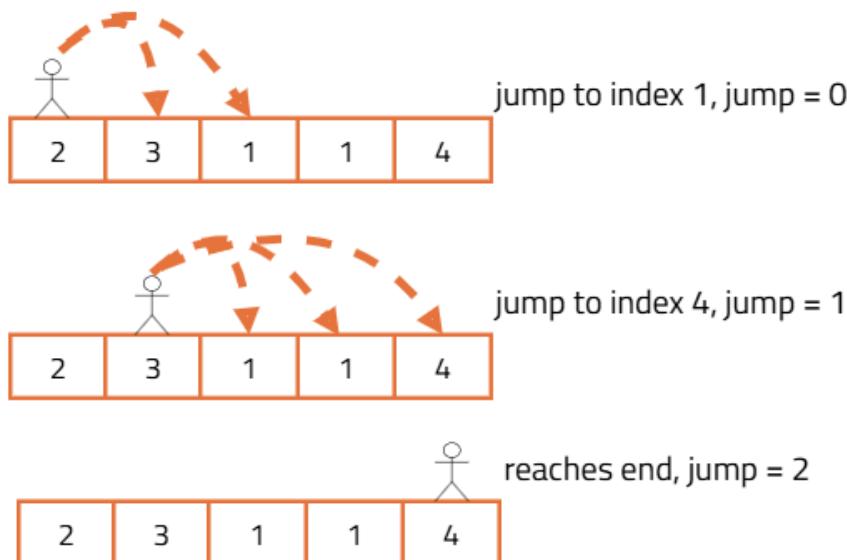
It is guaranteed that the last index is always reachable.

Example:

```
nums = [2, 3, 1, 1, 4]
```

From index 0, you can jump to index 1, and from index 1 you can jump directly to index 4.

Minimum jumps = 2.



## Approach 1: Brute Force (Recursion)

### Algorithm

We try all possible jumps from every index and find the minimum jumps needed to reach the end.

From the current index, we recursively try jumping 1 step, 2 steps, up to `nums[i]` steps. We return the minimum jumps among all possible paths.

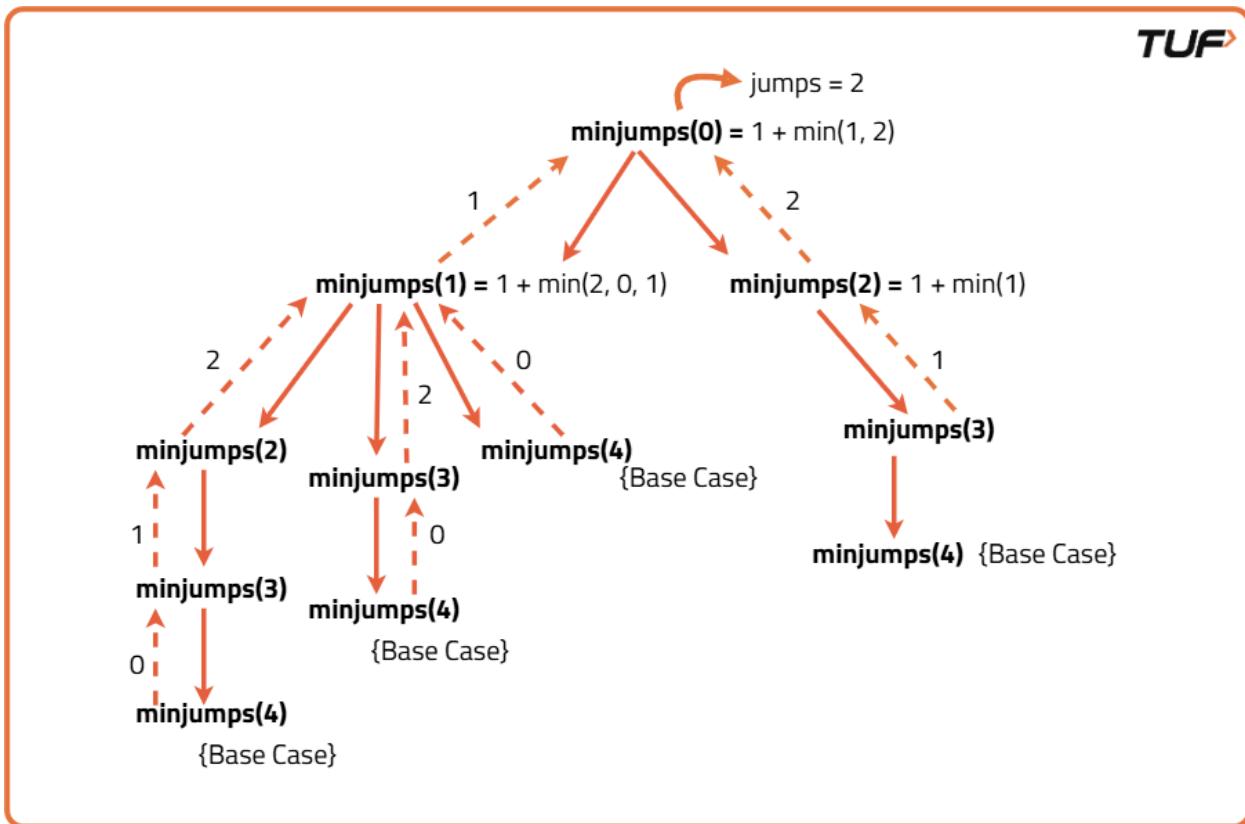
Steps:

- Create a recursive function that takes the current index.
- If the current index is at or beyond the last index, return 0.
- If `nums[i] == 0`, return `INT_MAX` because no jump is possible.
- Try all jumps from 1 to `nums[i]`.
- For each jump, recursively calculate jumps from the new index.

- Return the minimum jumps found.

Example dry run (nums = [2,3,1,1,4]):

- From index 0, try jumps to index 1 and 2.
- From index 1, try jumps to index 2, 3, 4.
- Reaching index 4 returns 0, so total jumps are counted back.



## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int jump(vector<int>& nums) {
 return minJumps(nums, 0);
 }
}
```

```

 }

private:
 int minJumps(vector<int>& nums, int position) {
 if (position >= nums.size() - 1) return 0;
 if (nums[position] == 0) return INT_MAX;

 int minStep = INT_MAX;
 for (int j = 1; j <= nums[position]; j++) {
 int res = minJumps(nums, position + j);
 if (res != INT_MAX)
 minStep = min(minStep, 1 + res);
 }
 return minStep;
 }
};

int main() {
 vector<int> nums = {2,3,1,1,4};
 Solution sol;
 cout << sol.jump(nums);
 return 0;
}

```

## Complexity Analysis

- Time Complexity:  $O(2^N)$   
Every index explores multiple recursive paths, causing exponential calls.
  - Space Complexity:  $O(N)$   
Due to recursion stack depth in the worst case.
- 

## Approach 2: Better Approach (Dynamic Programming)

### Algorithm

We use a DP array where  $dp[i]$  stores the minimum jumps needed to reach index  $i$ .  
We start from index 0 and update all reachable indices from each position.

Steps:

- Create a dp array of size n and initialize all values to INT\_MAX.
- Set  $dp[0] = 0$ .
- For each index  $i$ , try all jumps from 1 to  $\text{nums}[i]$ .
- Update  $dp[i + j] = \min(dp[i + j], dp[i] + 1)$ .
- The answer is  $dp[n-1]$ .

Example dry run ( $\text{nums} = [2,3,1,1,4]$ ):

- $dp[0] = 0$
- From index 0, update  $dp[1] = 1$ ,  $dp[2] = 1$
- From index 1, update  $dp[4] = 2$
- Final answer = 2

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int jump(vector<int>& nums) {
 int n = nums.size();
 vector<int> dp(n, INT_MAX);
 dp[0] = 0;

 for (int i = 0; i < n; i++) {
 for (int j = 1; j <= nums[i] && i + j < n; j++) {
 dp[i + j] = min(dp[i + j], dp[i] + 1);
 }
 }
 }
}
```

```

 return dp[n - 1];
 }
};

int main() {
 vector<int> nums = {2,3,1,1,4};
 Solution sol;
 cout << sol.jump(nums);
 return 0;
}

```

## Complexity Analysis

- Time Complexity:  $O(N^2)$   
Two nested loops where each index may explore up to  $N$  jumps.
  - Space Complexity:  $O(N)$   
Extra DP array is used to store minimum jumps.
- 

## Approach 3: Optimal Approach (Greedy)

### Algorithm

We use a greedy method to always jump as far as possible within the current range.  
Each jump expands the reachable range, and we count jumps when the current range ends.

Steps:

- Initialize jumps = 0, currentEnd = 0, farthest = 0.
- Traverse the array until the second last index.
- Update farthest = max(farthest, i + nums[i]).
- When  $i == \text{currentEnd}$ , increment jumps and update  $\text{currentEnd} = \text{farthest}$ .
- Return jumps.

Example dry run (nums = [2,3,1,1,4]):

- At index 0, farthest = 2, currentEnd reached → jumps = 1, currentEnd = 2
- At index 1, farthest = 4
- At index 2, currentEnd reached → jumps = 2, currentEnd = 4
- End reached with 2 jumps.

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int jump(vector<int>& nums) {
 int jumps = 0, currentEnd = 0, farthest = 0;

 for (int i = 0; i < nums.size() - 1; i++) {
 farthest = max(farthest, i + nums[i]);
 if (i == currentEnd) {
 jumps++;
 currentEnd = farthest;
 }
 }
 return jumps;
 }
};

int main() {
 vector<int> nums = {2,3,1,1,4};
 Solution sol;
 cout << sol.jump(nums);
 return 0;
}
```

## Complexity Analysis

- Time Complexity:  $O(N)$   
The array is traversed only once.
- Space Complexity:  $O(1)$   
Only constant extra variables are used.

## 9. Minimum number of platforms required for a railway

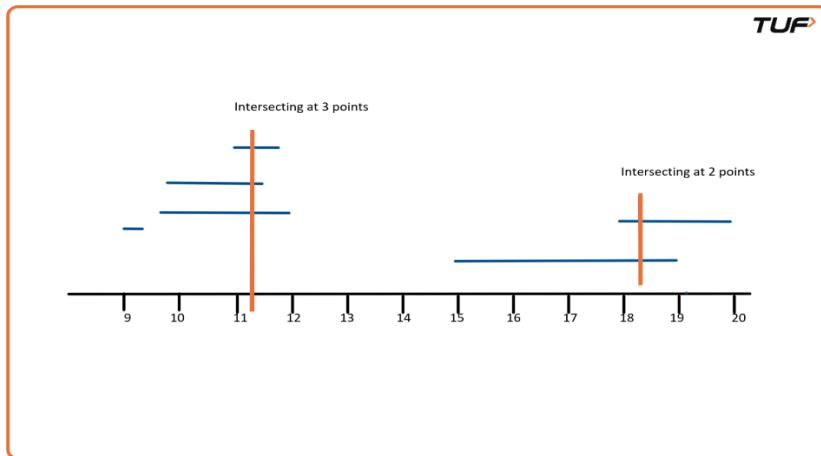
We are given two arrays: one for arrival times and one for departure times of trains at a railway station. Each train needs one platform during the time it is at the station.

The task is to find the minimum number of platforms required so that no train has to wait.

Example:

```
arr = {9:00, 9:45, 9:55, 11:00, 15:00, 18:00}
dep = {9:20, 12:00, 11:30, 11:50, 19:00, 20:00}
```

At time 11:00, three trains are present at the station at the same time, so the answer is 3.




---

### Approach 1: Brute Force

#### Algorithm

We check overlaps between every pair of trains.

For each train, we count how many other trains overlap with it.  
The maximum overlap count at any time is the number of platforms needed.

Steps:

- Initialize answer as 1.
- For each train  $i$ , start a count as 1.
- Compare train  $i$  with every other train  $j$ .
- If their time intervals overlap, increase the count.
- Update the maximum count after checking all overlaps for  $i$ .
- Return the maximum count.

Overlap condition:

Two trains overlap if the arrival of one lies between the arrival and departure of the other.

Example dry run (first example):

- For train at 9:45, it overlaps with trains at 9:55 and 11:00.
- Count becomes 3, which is the maximum.

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int countPlatforms(int n, int arr[], int dep[]) {
 int ans = 1;
 for (int i = 0; i < n; i++) {
 int count = 1;
 for (int j = i + 1; j < n; j++) {
 if ((arr[i] >= arr[j] && arr[i] <= dep[j]) ||
 (arr[j] >= arr[i] && arr[j] <= dep[i])) {
 count++;
 }
 }
 ans = max(ans, count);
 }
 return ans;
 }
}
```

```

 }
 }
 ans = max(ans, count);
}
return ans;
};

int main() {
 int arr[] = {900, 945, 955, 1100, 1500, 1800};
 int dep[] = {920, 1200, 1130, 1150, 1900, 2000};
 int n = sizeof(arr)/sizeof(arr[0]);
 Solution obj;
 cout << obj.countPlatforms(n, arr, dep);
 return 0;
}

```

### **Complexity Analysis**

Time Complexity:  $O(N^2)$

Because for every train, we compare it with all other trains.

Space Complexity:  $O(1)$

Only constant extra variables are used.

---

## **Approach 2: Optimal Approach**

### **Algorithm**

We want to know the maximum number of trains present at the station at the same time.

By sorting arrival and departure times separately, we can simulate train movements.

Steps:

- Sort the arrival array and departure array.
- Use two pointers: one for arrivals and one for departures.
- Start with one platform.

- If the next train arrives before or at the same time as the earliest departure, increase platforms.
- If a train departs before the next arrival, decrease platforms.
- Track the maximum number of platforms used at any time.

Example dry run (first example):

- At 9:00 → platforms = 1
- At 9:45 → platforms = 2
- At 9:55 → platforms = 3
- Maximum platforms needed = 3


  
**arr =**

|      |      |      |
|------|------|------|
| 9:00 | 9:05 | 9:10 |
|------|------|------|

  
**dep =**

|      |      |      |
|------|------|------|
| 9:07 | 9:12 | 9:20 |
|------|------|------|

| i | j | Time | Event                 | Platform | max Platform | Active Trains                      |
|---|---|------|-----------------------|----------|--------------|------------------------------------|
| 0 | 0 | 9:00 | T <sub>1</sub> arrive | 1        | 1            | [T <sub>1</sub> ]                  |
| 1 | 0 | 9:05 | T <sub>2</sub> arrive | 2        | 2            | [T <sub>1</sub> , T <sub>2</sub> ] |
| 2 | 0 | 9:07 | T <sub>1</sub> dep    | 1        | 2            | [T <sub>2</sub> ]                  |
| 2 | 1 | 9:10 | T <sub>3</sub> arrive | 2        | 2            | [T <sub>2</sub> , T <sub>3</sub> ] |
| 3 | 1 | 9:12 | T <sub>2</sub> dep    | 1        | 2            | [T <sub>3</sub> ]                  |
| 3 | 2 | 9:20 | T <sub>3</sub> dep    | 0        | 2            | []                                 |

Maximum platform required = 2

## Code

```
#include <bits/stdc++.h>
using namespace std;
```

```

class Solution {
public:
 int countPlatforms(int n, int arr[], int dep[]) {
 sort(arr, arr + n);
 sort(dep, dep + n);

 int platforms = 1;
 int result = 1;
 int i = 1, j = 0;

 while (i < n && j < n) {
 if (arr[i] <= dep[j]) {
 platforms++;
 i++;
 } else {
 platforms--;
 j++;
 }
 result = max(result, platforms);
 }
 return result;
 }
};

int main() {
 int arr[] = {900, 945, 955, 1100, 1500, 1800};
 int dep[] = {920, 1200, 1130, 1150, 1900, 2000};
 int n = sizeof(arr)/sizeof(arr[0]);
 Solution obj;
 cout << obj.countPlatforms(n, arr, dep);
 return 0;
}

```

### Complexity Analysis

Time Complexity:  $O(N \log N)$

Sorting both arrival and departure arrays takes  $N \log N$  time.

Space Complexity: O(1)  
No extra space is used apart from variables.

# 10. Job Sequencing Problem

You are given N jobs. Each job has an id, a deadline, and a profit.

Each job takes exactly one unit of time, and only one job can be done at a time.

A job gives profit only if it is completed on or before its deadline.

Your task is to find:

1. The maximum number of jobs that can be done
2. The maximum profit that can be earned

Example:

Jobs = {(1,4,20), (2,1,10), (3,1,40), (4,1,30)}

If we select the job with profit 40 first (deadline 1) and then the job with profit 20 (deadline 4), we can do 2 jobs with total profit 60.

---

## Approach:

### Algorithm

We always try to take the job with the highest profit first.

For each job, we try to schedule it as late as possible before its deadline so that earlier time slots remain free for other jobs.

Steps:

- Sort all jobs in decreasing order of profit.
- Find the maximum deadline among all jobs.
- Create an array slot[ ] of size maxDeadline + 1 and initialize all values to -1 (free slot).

- Traverse jobs one by one (highest profit first).
- For the current job, check from its deadline backwards to find a free slot.
- If a free slot is found, assign the job, increase job count, and add profit.
- Continue for all jobs.
- Return total jobs done and total profit.

Example dry run (Example 1):

Sorted by profit → (3,1,40), (4,1,30), (1,4,20), (2,1,10)

- Job 3 goes to slot 1 → profit = 40
  - Job 4 cannot be placed (slot 1 already used)
  - Job 1 goes to slot 4 → profit = 60
  - Job 2 cannot be placed
- Result: 2 jobs, profit = 60

## Code

```
#include <bits/stdc++.h>
using namespace std;

struct Job {
 int id;
 int dead;
 int profit;
};

class Solution {
public:
 static bool comparison(Job a, Job b) {
 return a.profit > b.profit;
 }

 pair<int,int> JobScheduling(Job arr[], int n) {
```

```

 sort(arr, arr + n, comparison);

 int maxi = arr[0].dead;
 for (int i = 1; i < n; i++) {
 maxi = max(maxi, arr[i].dead);
 }

 int slot[maxi + 1];
 for (int i = 0; i <= maxi; i++)
 slot[i] = -1;

 int countJobs = 0, jobProfit = 0;

 for (int i = 0; i < n; i++) {
 for (int j = arr[i].dead; j > 0; j--) {
 if (slot[j] == -1) {
 slot[j] = i;
 countJobs++;
 jobProfit += arr[i].profit;
 break;
 }
 }
 }
 return make_pair(countJobs, jobProfit);
 }
};

int main() {
 int n = 4;
 Job arr[n] = {{1,4,20},{2,1,10},{3,2,40},{4,2,30}};
 Solution ob;
 pair<int,int> ans = ob.JobScheduling(arr, n);
 cout << ans.first << " " << ans.second;
 return 0;
}

```

## Complexity Analysis

Time Complexity:  $O(N \log N) + O(N * M)$

- Sorting jobs takes  $O(N \log N)$
- For each job, we may scan up to  $M$  slots (maximum deadline)

Space Complexity:  $O(M)$

- Slot array of size equal to maximum deadline is used

## 11. Candy

A line of  $N$  kids is standing in a row. Each kid has a rating given in the array `ratings`. Candies must be distributed using the following rules:

- Every child must get at least one candy.
- A child with a higher rating than an adjacent child must get more candies than that neighbor.

Return the minimum number of candies required.

Example:

`ratings = [1, 0, 5]`

Candies distribution = [2, 1, 2]

Total candies = 5

---

### Approach 1: Brute Force

#### Algorithm

We start by giving every child one candy. Then we repeatedly fix violations of the rules by scanning from left to right and from right to left until no changes are needed.

Steps:

- Initialize a candies array with all values as 1.
- Repeat until no updates happen in a full iteration:
  - Traverse from left to right:
    - If current rating is higher than left neighbor and candies are not more, increase candies.
  - Traverse from right to left:
    - If current rating is higher than right neighbor and candies are not more, increase candies.
- Sum all candies and return the result.

Example dry run (ratings = [1,0,5]):

Initial candies = [1,1,1]  
 Right to left fix → candies = [2,1,2]  
 No more violations → total = 5

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int candy(vector<int>& ratings) {
 int n = ratings.size();
 vector<int> candies(n, 1);
 bool updated = true;

 while (updated) {
 updated = false;

 for (int i = 1; i < n; i++) {
 if (ratings[i] > ratings[i - 1] && candies[i] <=
candies[i - 1]) {
 candies[i] = candies[i - 1] + 1;
 updated = true;
 }
 }
 }

 return accumulate(candies.begin(), candies.end(), 0);
 }
}
```

```

 }
 }

 for (int i = n - 2; i >= 0; i--) {
 if (ratings[i] > ratings[i + 1] && candies[i] <=
candies[i + 1]) {
 candies[i] = candies[i + 1] + 1;
 updated = true;
 }
 }
}

return accumulate(candies.begin(), candies.end(), 0);
//candies vector ke saare elements ka sum nikaalo,
//aur starting sum 0 se shuru karo.
}
};

int main() {
 Solution obj;
 vector<int> ratings = {1, 0, 5};
 cout << obj.candy(ratings);
 return 0;
}

```

### **Complexity Analysis**

Time Complexity:  $O(N^2)$

Because multiple full passes may be required and each pass takes  $O(N)$ .

Space Complexity:  $O(N)$

An extra array is used to store candies.

---

## **Approach 2: Better Approach**

### **Algorithm**

Instead of repeated updates, we fix the conditions using exactly two passes.

Steps:

- Initialize a candies array with all values as 1.
- Traverse from left to right:
  - If current rating is higher than left neighbor, give more candies.
- Traverse from right to left:
  - If current rating is higher than right neighbor, update candies using max value.
- Sum all candies and return.

Example dry run (ratings = [1,0,5]):

Left to right → candies = [1,1,2]

Right to left → candies = [2,1,2]

Total = 5

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int candy(vector<int>& ratings) {
 int n = ratings.size();
 vector<int> candies(n, 1);

 for (int i = 1; i < n; i++) {
 if (ratings[i] > ratings[i - 1])
 candies[i] = candies[i - 1] + 1;
 }

 for (int i = n - 2; i >= 0; i--) {
 if (ratings[i] > ratings[i + 1])
 candies[i] = max(candies[i], candies[i + 1] + 1);
 }
 return accumulate(candies.begin(), candies.end(), 0);
 }
}
```

```

};

int main() {
 Solution obj;
 vector<int> ratings = {1,0,5};
 cout << obj.candy(ratings);
 return 0;
}

```

### **Complexity Analysis**

Time Complexity: O(N)

Two linear passes and one sum pass.

Space Complexity: O(N)

Extra candies array is used.

---

## **Approach 3: Optimal Approach**

### **Algorithm**

We treat the ratings as increasing and decreasing slopes.

Each increasing slope and decreasing slope adds candies.

The peak is counted twice, so we subtract the smaller slope length.

Steps:

- Start with candies = n (1 candy for each child).
- Traverse the array:
  - Count increasing slope and add candies.
  - Count decreasing slope and add candies.
  - Subtract  $\min(\text{peak}, \text{valley})$  to remove double count.
- Return total candies.

Example dry run (ratings = [1,0,5]):

- Decreasing slope: 1 → add 1
- Increasing slope: 1 → add 1
- Total = 3 + 2 = 5

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int candy(vector<int>& ratings) {
 int n = ratings.size();
 int candies = n;
 int i = 1;

 while (i < n) {
 if (ratings[i] == ratings[i - 1]) {
 i++;
 continue;
 }

 int peak = 0;
 while (i < n && ratings[i] > ratings[i - 1]) {
 peak++;
 candies += peak;
 i++;
 }

 int valley = 0;
 while (i < n && ratings[i] < ratings[i - 1]) {
 valley++;
 candies += valley;
 i++;
 }

 candies -= min(peak, valley);
 }
 }
}
```

```

 }
 return candies;
 }
};

int main() {
 Solution sol;
 vector<int> ratings = {1,3,6,8,9,5,3};
 cout << sol.candy(ratings);
 return 0;
}

```

### **Complexity Analysis**

Time Complexity:  $O(N)$

Each index is processed at most twice.

Space Complexity:  $O(1)$

Only constant variables are used.

Peak element **dono loops me count ho jaata hai:**

- ek baar increasing me
- ek baar decreasing me

But:

Peak ko **max(peak, valley)** candies chahiye,  
sum nahi.

Istiyeh:

- jo extra add ho gaya → subtract kar dete hain

## **FULL DRY RUN**

## Input

```
ratings = {1, 3, 6, 8, 9, 5, 3}
n = 7
candies = 7 (base)
```

---

## Increasing slope

1 → 3 → 6 → 8 → 9

| ste | peak | candies |
|-----|------|---------|
| p   |      |         |

3>1 1 +1 → 8

6>3 2 +2 → 10

8>6 3 +3 → 13

9>8 4 +4 → 17

peak = 4

---

## Decreasing slope

9 → 5 → 3

| ste | valley | candies |
|-----|--------|---------|
| p   |        |         |

5<9 1 +1 → 18

3<5 2 +2 → 20

valley = 2

---

## Overcount correction

candies -= min(4, 2) = 2

Final:

candies = 18

---



## Final Answer

18

# 12. Shortest Job First (or SJF) CPU Scheduling

You are given a list of job durations, where each value represents the time required to complete a job.

The CPU follows the Shortest Job First (SJF) scheduling policy, which means the job with the smallest duration is executed first.

Your task is to calculate the average waiting time of all jobs.

The waiting time of a job is the total time it waits before it starts execution.

Example:

jobs = [3, 1, 4, 2, 5]

After sorting → [1, 2, 3, 4, 5]

Waiting times → [0, 1, 3, 6, 10]

Average waiting time =  $(0 + 1 + 3 + 6 + 10) / 5 = 4$

Explanation:

Socho bank ki line hai:

- Person A → kaam: 1 min
- Person B → kaam: 10 min
- Person C → kaam: 2 min

X Agar B pehle chala gaya:

- A aur C ko 10 min wait karna padega

✓ Agar A → C → B chale:

- Sabka wait minimum hoga

👉 Chhota kaam pehle = sabka wait kam

Yahi hai SJF.

---

## Approach:

### Algorithm

Shortest Job First works by always executing the job with the minimum duration first.

Steps:

- Sort the jobs array in ascending order.
- Initialize `totalTime` = 0 to track execution time so far.
- Initialize `waitTime` = 0 to store total waiting time of all jobs.
- Traverse the sorted jobs array:
  - Add `totalTime` to `waitTime` (waiting time for current job).
  - Add current job duration to `totalTime`.
- After processing all jobs, divide total waiting time by number of jobs to get average waiting time.

Example dry run (jobs = [3,1,4,2,5]):

Sorted jobs = [1,2,3,4,5]

Job 1 → wait = 0, `totalTime` = 1

Job 2 → wait = 1, totalTime = 3  
Job 3 → wait = 3, totalTime = 6  
Job 4 → wait = 6, totalTime = 10  
Job 5 → wait = 10, totalTime = 15  
Average =  $(0+1+3+6+10)/5 = 4$

## Code

```
#include <bits/stdc++.h>
using namespace std;

class ShortestJobFirst {
public:
 float calculateAverageWaitTime(vector<int>& jobs) {
 sort(jobs.begin(), jobs.end());

 float waitTime = 0;
 int totalTime = 0;
 int n = jobs.size();

 for (int i = 0; i < n; i++) {
 waitTime += totalTime;
 totalTime += jobs[i];
 }
 return waitTime / n;
 }
};

int main() {
 vector<int> jobs = {4, 3, 7, 1, 2};
 ShortestJobFirst sjf;
 cout << sjf.calculateAverageWaitTime(jobs);
 return 0;
}
```

## Complexity Analysis

Time Complexity:  $O(N \log N)$

Sorting the jobs array takes  $O(N \log N)$  time, and calculating waiting time takes  $O(N)$ .

Space Complexity: O(1)

No extra space is used apart from variables; sorting is done in-place.

## 13. Program for Least Recently Used (LRU) Page Replacement Algorithm

We need to design an LRU (Least Recently Used) Cache with a fixed capacity.

The cache supports two operations:

- **get(key):**  
Return the value of the key if it exists in the cache, otherwise return -1.  
Accessing a key makes it the most recently used.
- **put(key, value):**  
Insert or update the key with the given value.  
If the cache exceeds its capacity, remove the least recently used key.

Both operations must work in **O(1)** average time.

Example:

Capacity = 2

put(1,1) → cache = {1}  
put(2,2) → cache = {1,2}  
get(1) → returns 1, cache = {2,1}  
put(3,3) → evict 2, cache = {1,3}

---

### Approach 1

#### Algorithm

To achieve O(1) time for both operations, we combine two data structures:

- **HashMap:**  
Stores key → node mapping for fast lookup.

- **Doubly Linked List:**  
Maintains usage order.

- Most recently used node is near the head.
- Least recently used node is near the tail.

Steps:

- Use dummy head and tail nodes to simplify insert and delete operations.
- On `get(key)`:
  - If key exists, move the node to the front (most recently used) and return its value.
  - If not found, return -1.
- On `put(key, value)`:
  - If key already exists, remove the old node.
  - If cache is full, remove the node just before the tail (least recently used).
  - Insert the new node at the front.
- Always keep HashMap updated with the latest node positions.

Example dry run (capacity = 2):

- `put(1,1) → [1]`
- `put(2,2) → [2,1]`
- `get(1) → move 1 to front → [1,2]`
- `put(3,3) → remove 2 → [3,1]`

## Code

```
#include <bits/stdc++.h>
```

```

using namespace std;

class LRUCache {
public:
 class Node {
public:
 int key, val;
 Node* next;
 Node* prev;
 Node(int k, int v) {
 key = k;
 val = v;
 next = prev = NULL;
 }
 };
 Node* head = new Node(-1, -1);
 Node* tail = new Node(-1, -1);
 int cap;
 unordered_map<int, Node*> mp;

 LRUCache(int capacity) {
 cap = capacity;
 head->next = tail;
 tail->prev = head;
 }

 void addNode(Node* node) {
 Node* temp = head->next;
 node->next = temp;
 node->prev = head;
 head->next = node;
 temp->prev = node;
 }

 void deleteNode(Node* node) {
 Node* prevNode = node->prev;
 Node* nextNode = node->next;

```

```

 prevNode->next = nextNode;
 nextNode->prev = prevNode;
}

int get(int key) {
 if (mp.find(key) != mp.end()) {
 Node* node = mp[key];
 int res = node->val;
 mp.erase(key);
 deleteNode(node);
 addNode(node);
 mp[key] = head->next;
 return res;
 }
 return -1;
}

void put(int key, int value) {
 if (mp.find(key) != mp.end()) {
 Node* node = mp[key];
 mp.erase(key);
 deleteNode(node);
 }

 if (mp.size() == cap) {
 mp.erase(tail->prev->key);
 deleteNode(tail->prev);
 }

 addNode(new Node(key, value));
 mp[key] = head->next;
}
};

int main() {
 LRUCache cache(2);
 cache.put(1,1);
 cache.put(2,2);
}

```

```

 cout << cache.get(1) << endl;
 cache.put(3,3);
 cout << cache.get(2) << endl;
 cache.put(4,4);
 cout << cache.get(1) << endl;
 cout << cache.get(3) << endl;
 cout << cache.get(4) << endl;
 return 0;
}

```

## Complexity Analysis

Time Complexity:

- `get()` →  $O(1)$  average time due to HashMap lookup and constant list operations.
- `put()` →  $O(1)$  average time for insert, delete, and eviction.

Space Complexity:  $O(\text{capacity})$

We store at most capacity nodes in the doubly linked list and HashMap.

# 14. Insert Interval

You are given a list of non-overlapping intervals sorted by their start time.

You are also given a new interval.

Your task is to insert the new interval into the list and merge all overlapping intervals, then return the final list of intervals.

Example:

`intervals = [[1,3],[4,5],[6,7],[8,10]]`

`newInterval = [5,6]`

After insertion and merging, the result becomes:

`[[1,3],[4,7],[8,10]]`

## Approach 1: Naive Approach (Insertion and Merging)

### Algorithm

The idea is simple. We first insert the new interval into the list and then merge all overlapping intervals using the same logic as the “Merge Overlapping Intervals” problem.

Steps:

- Add the new interval to the intervals list.
- Sort the intervals based on starting time.
- Initialize a result list and add the first interval.
- Traverse the remaining intervals one by one.
- If the current interval overlaps with the last interval in result, merge them.
- Otherwise, add the current interval to result.
- Return the merged list.

Example dry run:

```
intervals = [[1,3],[4,5],[6,7],[8,10]]
newInterval = [5,6]
After insertion → [[1,3],[4,5],[6,7],[8,10],[5,6]]
After sorting → [[1,3],[4,5],[5,6],[6,7],[8,10]]
Merging overlapping intervals → [[1,3],[4,7],[8,10]]
```

### Code

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

vector<vector<int>> mergeOverlap(vector<vector<int>>& intervals) {
 sort(intervals.begin(), intervals.end());
 vector<vector<int>> res;
 res.push_back(intervals[0]);

 for (int i = 1; i < intervals.size(); i++) {
```

```

vector<int>& last = res.back();
vector<int>& curr = intervals[i];

if (curr[0] <= last[1])
 last[1] = max(last[1], curr[1]);
else
 res.push_back(curr);
}
return res;
}

vector<vector<int>> insertInterval(vector<vector<int>>& intervals,
 vector<int>& newInterval) {
 intervals.push_back(newInterval);
 return mergeOverlap(intervals);
}

int main() {
 vector<vector<int>> intervals = {{1,3},{4,5},{6,7},{8,10}};
 vector<int> newInterval = {5,6};

 vector<vector<int>> res = insertInterval(intervals, newInterval);
 for (auto interval : res) {
 cout << interval[0] << " " << interval[1] << endl;
 }
 return 0;
}

```

### Complexity Analysis

Time Complexity:  $O(n \log n)$

Sorting the intervals takes  $O(n \log n)$ , and merging takes  $O(n)$ .

Space Complexity:  $O(1)$  (excluding output)

No extra data structure is used apart from the result array.

---

## Approach 2: Expected Approach (Contiguous Interval Merging)

### Algorithm

Since the intervals are already sorted, only a contiguous group of intervals can overlap with the new interval. We directly find and merge only those intervals.

Steps:

- Add all intervals that end before the new interval starts.
- Merge all intervals that overlap with the new interval by expanding its range.
- Add the merged interval to the result.
- Add all remaining intervals after the merged interval.
- Return the result list.

Example dry run:

intervals = [[1,3],[4,5],[6,7],[8,10]]

newInterval = [5,6]

- Add intervals before overlap → [[1,3]]
- Merge overlapping intervals → newInterval becomes [4,7]
- Add merged interval → [[1,3],[4,7]]
- Add remaining intervals → [[1,3],[4,7],[8,10]]

### Code

```
#include <iostream>
#include <vector>
using namespace std;

vector<vector<int>> insertInterval(vector<vector<int>>& intervals,
 vector<int>& newInterval) {
 vector<vector<int>> res;
 int i = 0;
 int n = intervals.size();
```

```

 while (i < n && intervals[i][1] < newInterval[0]) {
 res.push_back(intervals[i]);
 i++;
 }

 while (i < n && intervals[i][0] <= newInterval[1]) {
 newInterval[0] = min(newInterval[0], intervals[i][0]);
 newInterval[1] = max(newInterval[1], intervals[i][1]);
 i++;
 }
 res.push_back(newInterval);

 while (i < n) {
 res.push_back(intervals[i]);
 i++;
 }
 return res;
 }

int main() {
 vector<vector<int>> intervals = {{1,3},{4,5},{6,7},{8,10}};
 vector<int> newInterval = {5,6};

 vector<vector<int>> res = insertInterval(intervals, newInterval);
 for (auto interval : res) {
 cout << interval[0] << " " << interval[1] << endl;
 }
 return 0;
}

```

### Complexity Analysis

Time Complexity: O(n)

Each interval is processed only once.

Space Complexity: O(n)

A new result array is used to store the merged intervals.

# 15. Merge Overlapping Sub-intervals

You are given an array of intervals where each interval is represented as [start, end].

If two intervals overlap, they should be merged into a single interval.

The final result should contain only non-overlapping intervals that cover all given intervals.

Example:

intervals = [[1,3],[2,6],[8,10],[15,18]]

Merged result = [[1,6],[8,10],[15,18]]

---

## Approach 1: Brute-Force Approach

### Algorithm

The idea is to merge intervals by checking overlaps manually after sorting.

Steps:

- Sort all intervals based on their starting time.
- Traverse the intervals one by one.
- Take the current interval as a new merged interval.
- Check the next intervals to see if they overlap with the current one.
- If an interval overlaps, extend the end of the current interval.
- Keep merging until no more overlaps are found.
- Store the merged interval and move to the next non-overlapping interval.

Example dry run:

intervals = [[1,3],[2,6],[8,10],[15,18]]

Sorted → same order

- Start with [1,3], overlaps with [2,6] → merge to [1,6]
- [8,10] does not overlap → keep as is
- [15,18] does not overlap → keep as is

Final answer = [[1,6],[8,10],[15,18]]

### Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 vector<vector<int>> merge(vector<vector<int>>& intervals) {
 sort(intervals.begin(), intervals.end());
 vector<vector<int>> ans;
 int n = intervals.size();

 for (int i = 0; i < n;) {
 int start = intervals[i][0];
 int end = intervals[i][1];
 int j = i + 1;

 while (j < n && intervals[j][0] <= end) {
 end = max(end, intervals[j][1]);
 j++;
 }
 ans.push_back({start, end});
 i = j;
 }
 return ans;
 }
};

int main() {
 Solution sol;
 vector<vector<int>> intervals = {{1,3},{2,6},{8,10},{15,18}};
 vector<vector<int>> result = sol.merge(intervals);

 for (auto interval : result) {
 cout << "[" << interval[0] << "," << interval[1] << "] ";
 }
 return 0;
}
```

}

### Complexity Analysis

Time Complexity:  $O(N^2)$

For each interval, we may scan many future intervals to check overlap.

Space Complexity:  $O(N)$

Extra space is used to store the merged intervals.

---

## Approach 2: Optimal Approach

### Algorithm

Instead of checking all future intervals, we only compare each interval with the last merged one.

Steps:

- Sort intervals based on starting time.
- Create an empty result list.
- Traverse each interval:
  - If result is empty or current interval does not overlap with the last one, add it.
  - If it overlaps, merge by updating the end of the last interval.
- Return the result list.

Example dry run:

intervals = [[1,3],[2,6],[8,10],[15,18]]

- Add [1,3]
- [2,6] overlaps → merge to [1,6]
- [8,10] does not overlap → add
- [15,18] does not overlap → add

Final answer = [[1,6],[8,10],[15,18]]

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 vector<vector<int>> merge(vector<vector<int>>& intervals) {
 sort(intervals.begin(), intervals.end());
 vector<vector<int>> merged;

 for (auto interval : intervals) {
 if (merged.empty() || merged.back()[1] < interval[0]) {
 merged.push_back(interval);
 } else {
 merged.back()[1] = max(merged.back()[1], interval[1]);
 }
 }
 return merged;
 }
};

int main() {
 Solution sol;
 vector<vector<int>> intervals = {{1,3},{2,6},{8,10},{15,18}};
 vector<vector<int>> result = sol.merge(intervals);

 for (auto interval : result) {
 cout << "[" << interval[0] << "," << interval[1] << "] ";
 }
 return 0;
}
```

## Complexity Analysis

Time Complexity:  $O(N \log N)$

Sorting takes  $O(N \log N)$  and merging takes  $O(N)$ .

Space Complexity:  $O(N)$

Extra space is used to store the merged intervals.

# 16. Non-overlapping Intervals

You are given N intervals, where each interval is represented as [start, end].

Two intervals are overlapping if one starts before the other ends.

Your task is to return the **minimum number of intervals to remove** so that the remaining intervals are **non-overlapping**.

Example:

Intervals = [[1,2],[2,3],[3,4],[1,3]]

If we remove [1, 3], the remaining intervals do not overlap.

So, the answer is 1.

---

## Approach 1: Brute Force Approach

### Algorithm

The brute force idea is to try all possible subsets of intervals and check which subsets are non-overlapping. Among all valid (non-overlapping) subsets, we find the one with the maximum number of intervals kept. The number of intervals to remove is the total intervals minus this maximum valid subset size.

Steps:

- Generate all possible subsets of the intervals using bitmasking.
- For each subset:
  - Sort the selected intervals by start time.
  - Check if any two consecutive intervals overlap.
- If the subset is valid (non-overlapping), update the maximum size found.
- Return total intervals - maximum valid subset size.

Example dry run (Intervals = [[1,2],[2,3],[3,4],[1,3]]):

- One valid subset is [[1,2],[2,3],[3,4]] with size 3.
- Total intervals = 4

- Minimum removals =  $4 - 3 = 1$

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int eraseOverlapIntervals(vector<vector<int>>& intervals) {
 int n = intervals.size();
 int maxValid = 0;

 for (int mask = 0; mask < (1 << n); mask++) {
 vector<vector<int>> subset;

 for (int i = 0; i < n; i++) {
 if (mask & (1 << i))
 subset.push_back(intervals[i]);
 }

 sort(subset.begin(), subset.end());

 bool valid = true;
 for (int i = 1; i < subset.size(); i++) {
 if (subset[i][0] < subset[i - 1][1]) {
 valid = false;
 break;
 }
 }

 if (valid)
 maxValid = max(maxValid, (int)subset.size());
 }

 return n - maxValid;
 }
};
```

```

int main() {
 Solution sol;
 vector<vector<int>> intervals = {{1,2},{2,3},{3,4},{1,3}};
 cout << sol.eraseOverlapIntervals(intervals);
 return 0;
}

```

## Complexity Analysis

Time Complexity:  $O(2^n \times N \log N)$

There are  $2^n$  subsets. For each subset, we sort and check overlaps.

Space Complexity:  $O(N)$

Extra space is used to store a subset of intervals.

---

## Approach 2: Optimal Approach (Greedy)

### Algorithm

To keep the maximum number of non-overlapping intervals, we greedily select intervals that finish earliest. This ensures that we leave maximum room for future intervals.

Steps:

- Sort all intervals by their ending time.
- Initialize prevEnd as the end of the first interval.
- Initialize a counter for removals.
- Traverse the remaining intervals:
  - If the current interval starts before prevEnd, it overlaps, so increment removal count.
  - Otherwise, update prevEnd to the current interval's end.
- Return the removal count.

Example dry run (Intervals = [[1,2],[2,3],[3,4],[1,3]]):

After sorting by end → [[1,2],[1,3],[2,3],[3,4]]

- Keep [1,2], prevEnd = 2
- [1,3] overlaps → remove
- [2,3] does not overlap → keep, prevEnd = 3
- [3,4] does not overlap → keep  
Removals = 1

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int eraseOverlapIntervals(vector<vector<int>>& intervals) {
 sort(intervals.begin(), intervals.end(),
 [] (vector<int>& a, vector<int>& b) {
 return a[1] < b[1];
 });

 int count = 0;
 int prevEnd = intervals[0][1];

 for (int i = 1; i < intervals.size(); i++) {
 if (intervals[i][0] < prevEnd) {
 count++;
 } else {
 prevEnd = intervals[i][1];
 }
 }
 return count;
 }
};

int main() {
 Solution sol;
 vector<vector<int>> intervals = {{1,3},{2,4},{3,5},{1,2}};
}
```

```
cout << sol.eraseOverlapIntervals(intervals);
return 0;
}
```

### **Complexity Analysis**

Time Complexity:  $O(N \log N)$

Sorting the intervals takes  $O(N \log N)$  and traversal takes  $O(N)$ .

Space Complexity:  $O(1)$

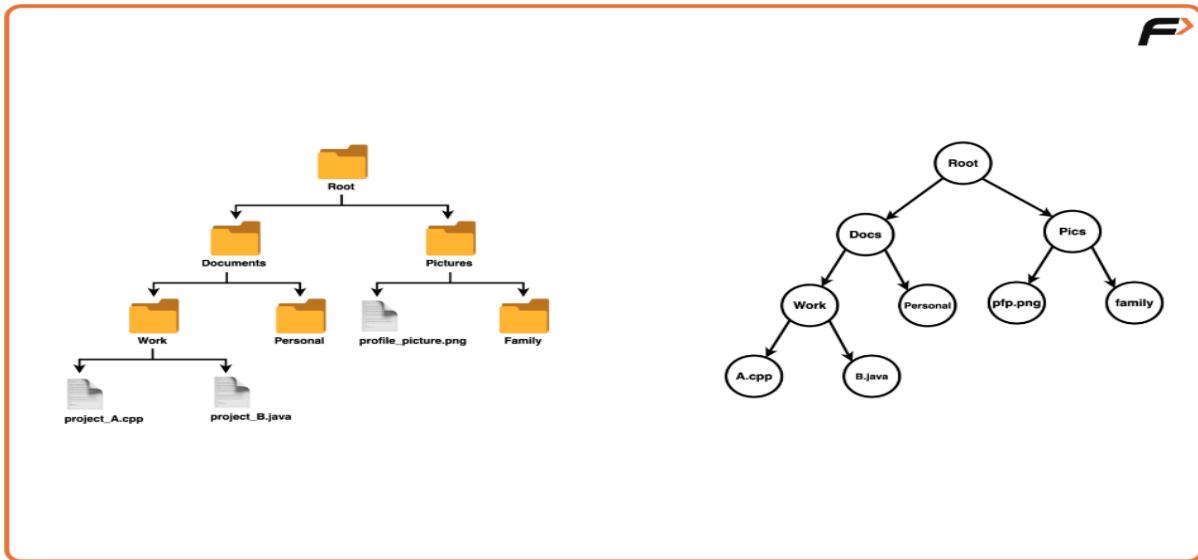
Only constant extra variables are used.

# **Binary Trees**

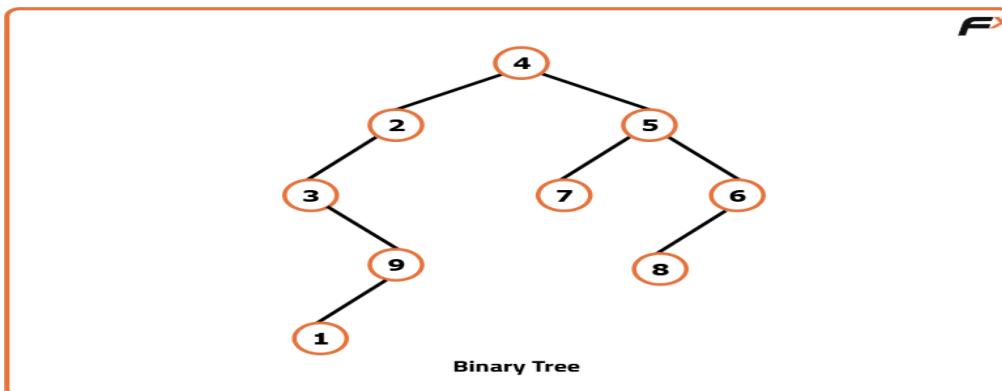
# 1. Introduction to Trees

In the world of data structures and algorithms, understanding binary trees lays the groundwork for hierarchical organisation and efficient data manipulation.

Up until now, we have studied array, linked list, stack and queues which are the fundamental linear data structures. Binary Trees are a different data structure and allow hierarchical organisation and structure of multi-level sequences. This resembles a tree with branching at each node expanding the tree in a non-linear fashion.



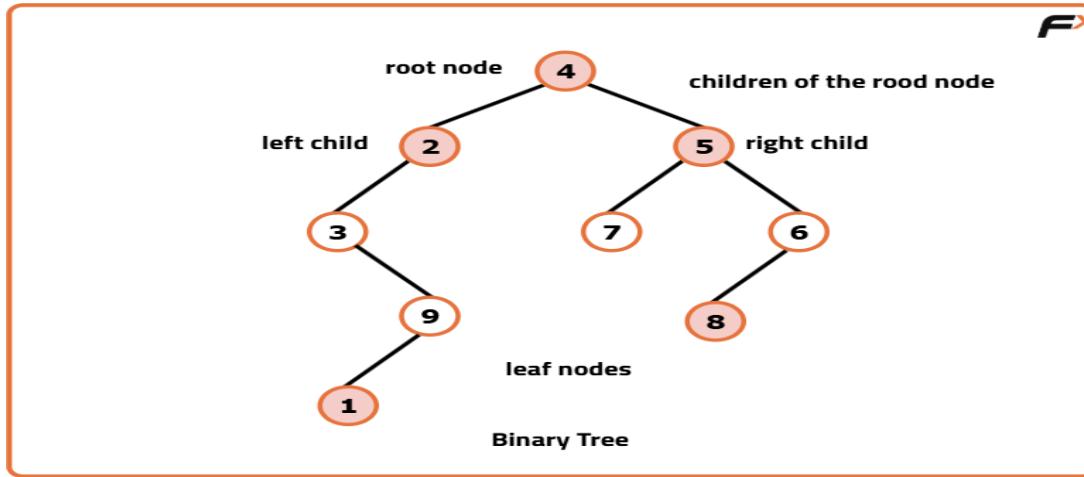
Just as the folders, subfolders and files are hierarchically arranged in your computer's file system, the binary tree has a similar structure with nodes representing folders and their children nodes representing the sub directory or files inside it.



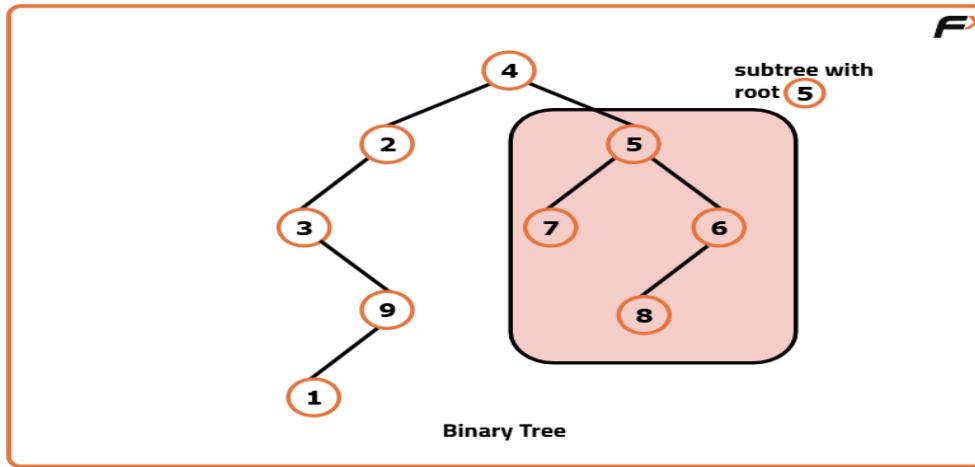
---

**Binary Tree:** Binary tree is a fundamental hierarchical data structure in computer science that comprises nodes arranged in a tree-like structure. It consists of nodes, where each node can have at most two children nodes, known as the left child and the right child.

**Nodes:** Each node in a binary tree contains a piece of data, often referred to as the node's value or key. This node also contains references and pointers to its left and right children so that we can traverse from one node to another in a hierarchical order.

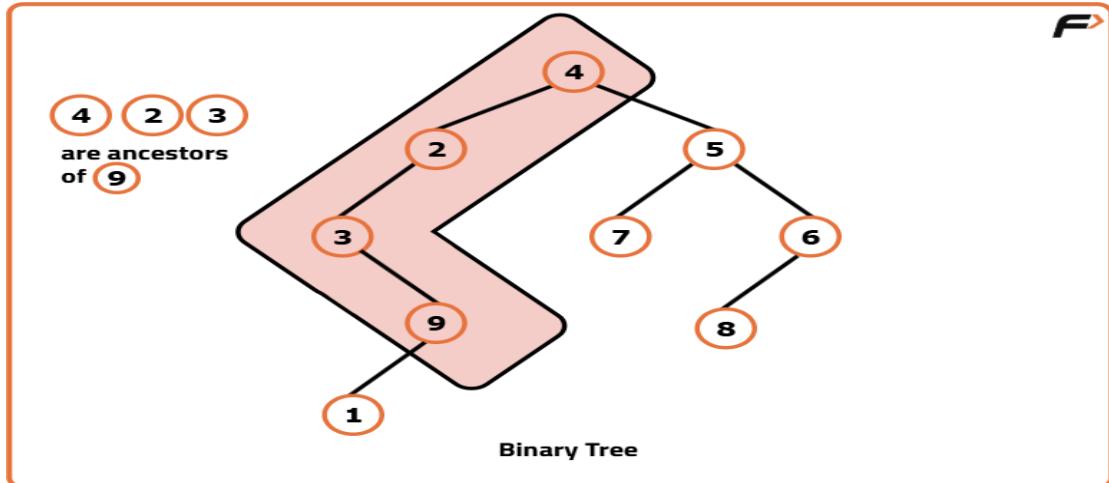


**Root Node:** is the topmost node of a binary tree from which all other nodes stem out. This serves as the entry point for traversing the tree structure.



**Children Nodes** are the nodes directly connected to a parent node. In a binary tree, a parent node can have zero, one or two children nodes, each situated to the left or right of the parent node.

**Leaf Nodes** are the nodes that do not have any children. These nodes lie on the outermost ends of the tree branches and are the terminal points of the traversal.



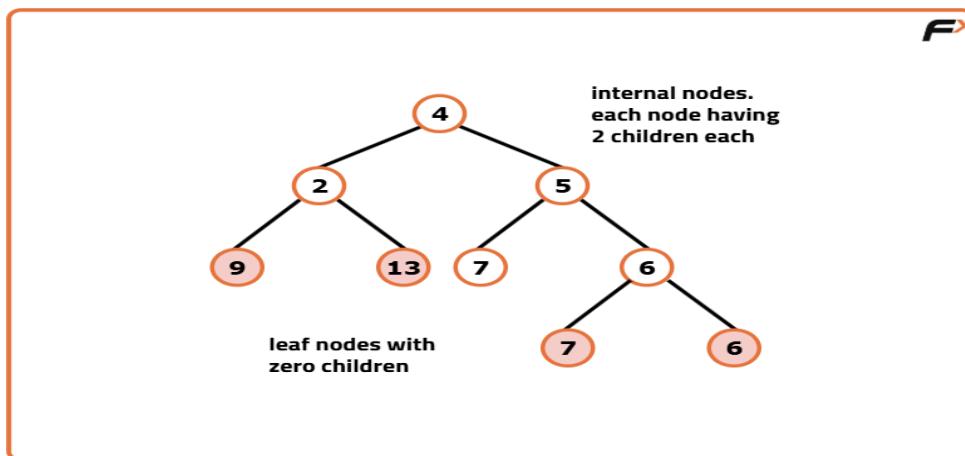
**Ancestors** in a Binary Tree are those nodes that lie on the path from a particular node to the root node. They are the nodes encountered while moving upwards from a specific node through its parent nodes until reaching the root of the tree.

---

## Full Binary Tree:

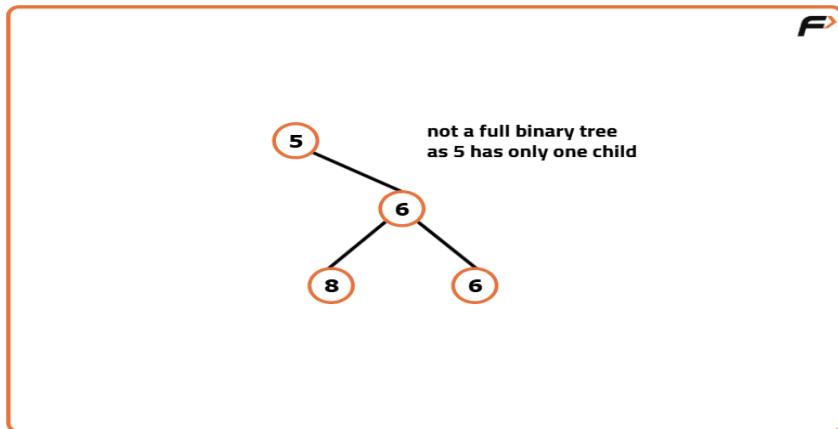
A Full Binary Tree, also known as a Strict Binary Tree, adheres to the structural property where every node has either zero or two children.

No node of this tree has just a single child, all internal nodes have exactly two children or in the case of leaf nodes, no children.



The property that each node has either 2 or 0 children contributes significantly to the tree's balance, making traversal, searching and insertion options more predictable and efficient.

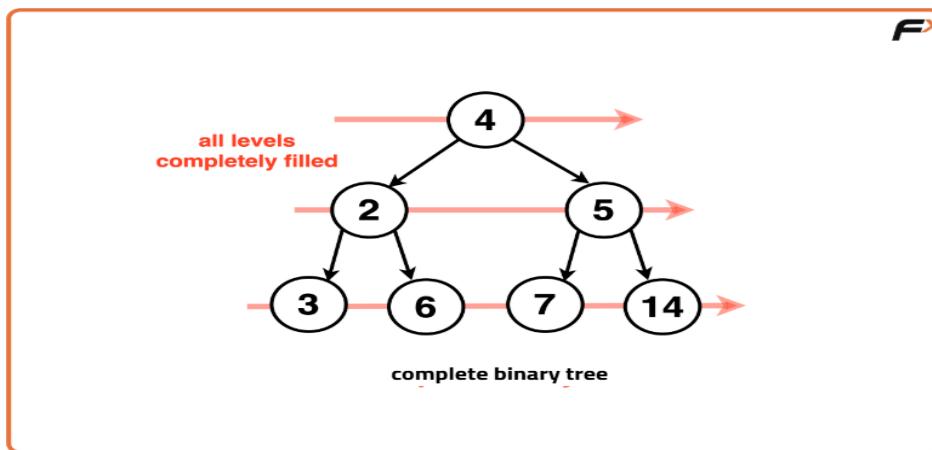
The emphasis that the internal nodes must have exactly two children optimises the tree's space utilisation and makes it more balanced.



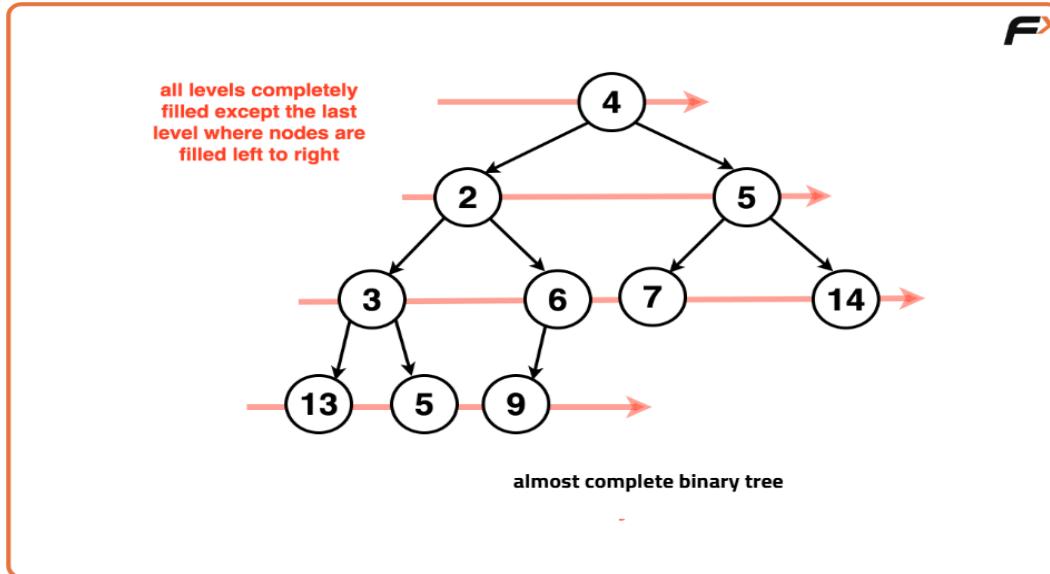
## Complete Binary Tree:

A Complete Binary Tree is a specialised form of Binary Tree where all levels are filled completely except possibly the last level, which is filled from left to right.

All levels of the tree, except possibly the last one, are fully filled. If the last level is not completely filled, it is filled from left to right, ensuring that nodes are positioned as far left as possible.

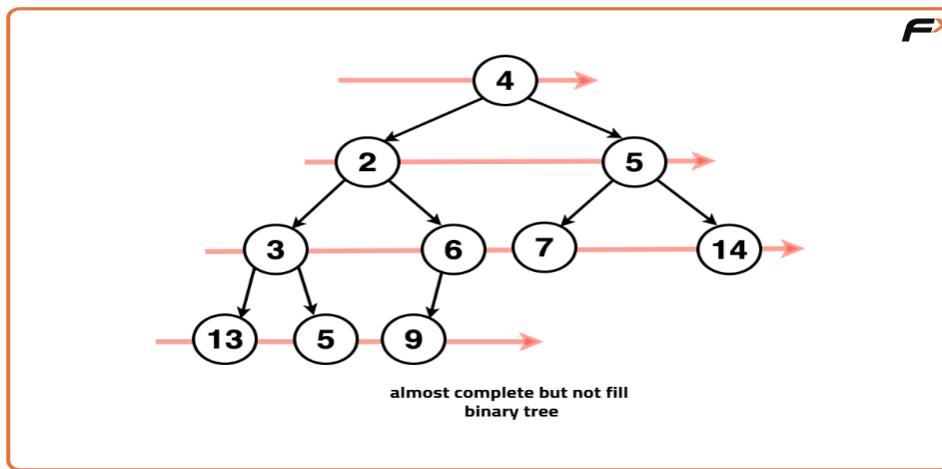


In a complete binary tree, all leaf nodes are in the last level or the second-to-last level, and they are positioned towards the leftmost side.



This structure is particularly useful for storing data in structures like heaps, where efficient access to the top element (root) or certain properties (e.g., maximum or minimum values in a heap) is crucial. The completeness property of a complete binary tree aids in achieving balanced structures, making it easier to implement algorithms and ensuring reasonably consistent performance.

One important aspect to note is that although it might seem similar to a full binary tree, a complete binary tree doesn't require all nodes to have two children; it's about the positioning and arrangement of nodes.



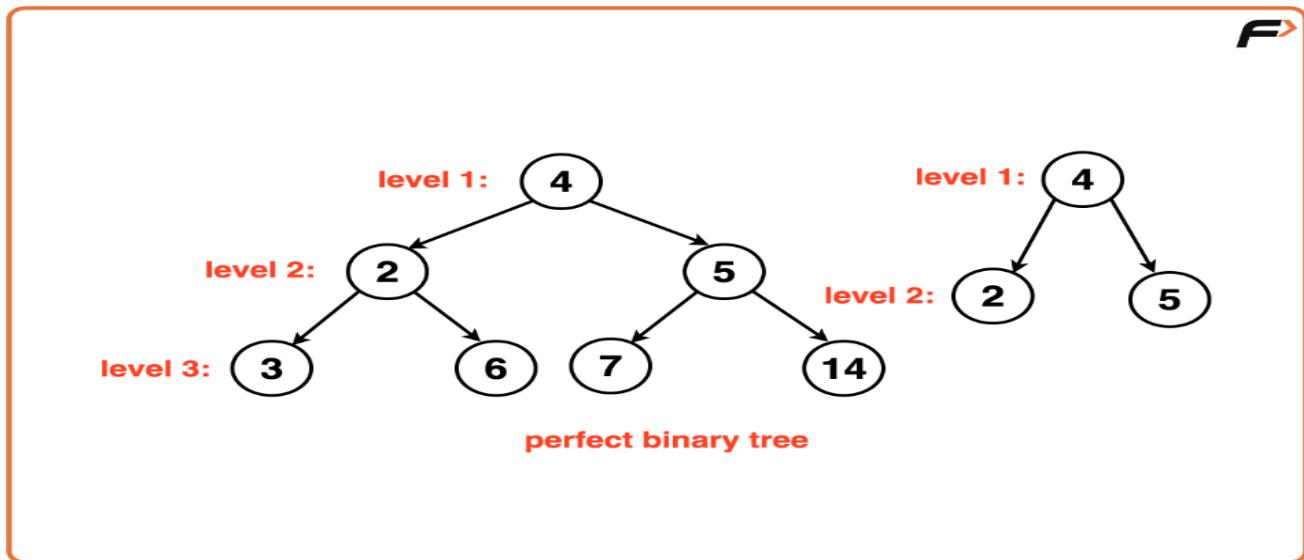
## Perfect Binary Tree:

A Perfect Binary tree is a type of Binary Tree where all leaf nodes are at the same level and the number of leaf nodes is maximised for that level.

Every node in a perfect binary tree has either zero or two children. This means that every internal node (non-leaf node) has exactly two children and all leaf nodes are at the same level.

All levels of this tree are fully filled with nodes including the last level. Perfect Binary Trees have a balanced structure that maximises the number of nodes for a given height, creating a dense structure where the number of nodes doubles as we move down each level of the tree.

Properties of perfect binary trees make them efficient for certain operations like searching and sorting due to their balanced nature. However, achieving and maintaining perfect balance, especially when the number of nodes is not a power of two, might not be feasible in many practical applications.



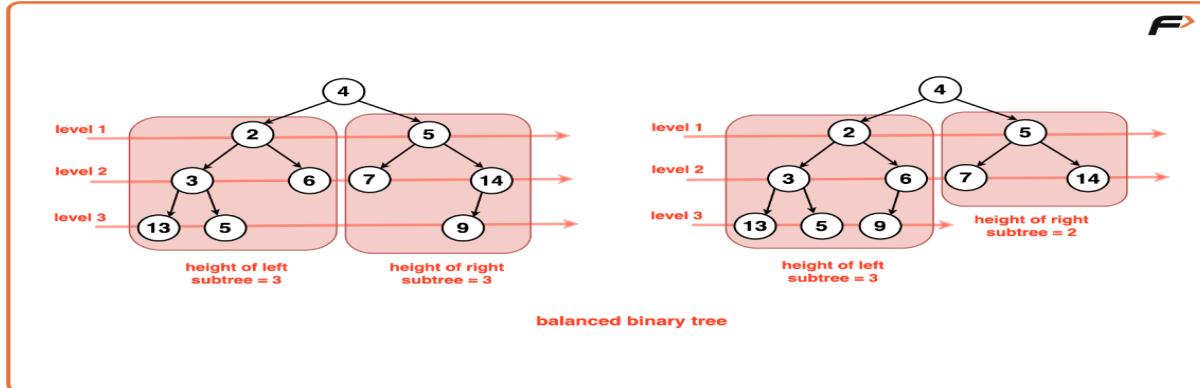
All levels of this tree are fully filled with nodes including the last level. Perfect Binary Trees have a balanced structure that maximises the number of nodes for a given height, creating a dense structure where the number of nodes doubles as we move down each level of the tree.

Properties of perfect binary trees make them efficient for certain operations like searching and sorting due to their balanced nature. However, achieving and maintaining perfect balance, especially when the number of nodes is not a power of two, might not be feasible in many practical applications.

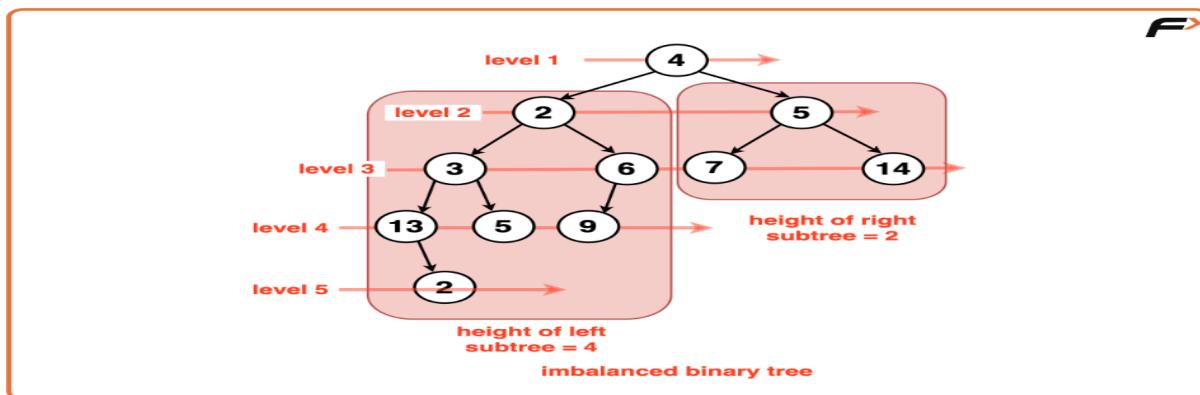
---

## Balanced Binary Tree:

A Balanced Binary tree is a type of Binary Tree where the heights of the two subtrees of any node differ by at most one. This property ensures that the tree remains relatively well-balanced, preventing the tree from becoming highly skewed or degenerate.

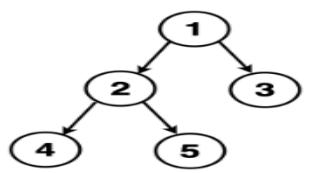


In a balanced binary tree, the height of the tree should be  $\log_2 N$  at maximum, where  $N$  is the number of nodes. This ensures that the tree does not become heavily skewed or imbalanced. The distribution of nodes of both the left and right subtrees remains relatively even throughout the tree.

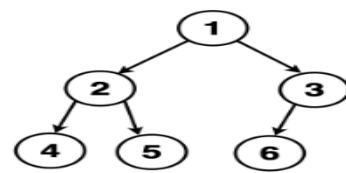


## Degenerate Tree:

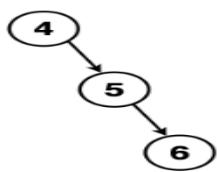
A Degenerate Tree is a Binary Tree where the nodes are arranged in a single path leaning to the right or left. The tree resembles a linked list in its structure where each node points to the next node in a linear fashion.



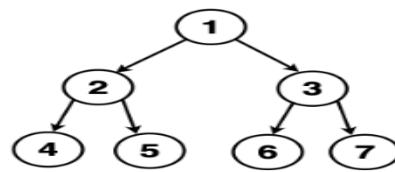
Full Binary Tree



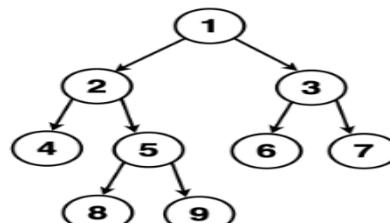
(Almost) Complete Binary Tree



Degenerate Binary Tree



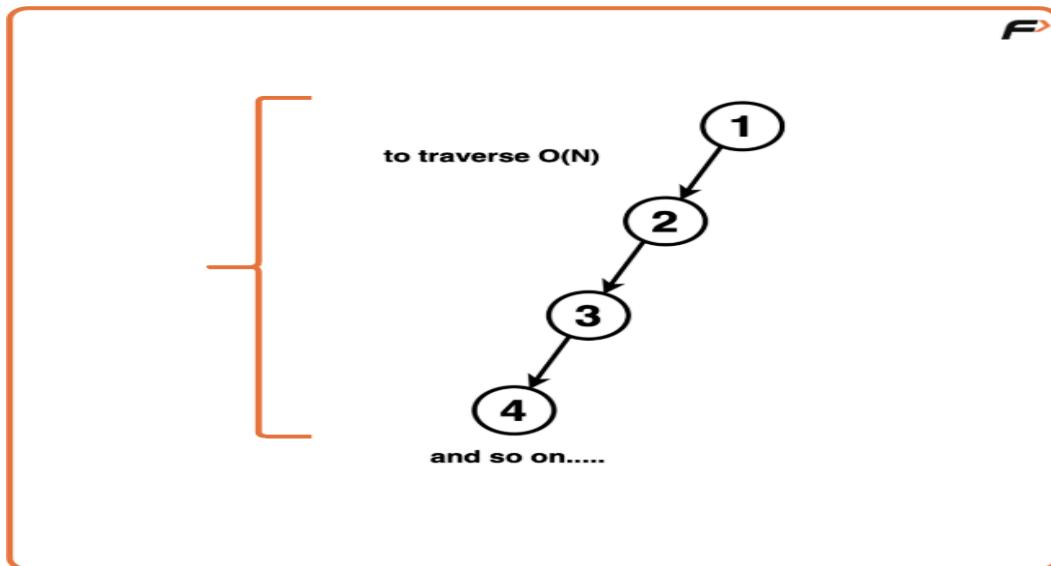
Perfect Binary Tree



Balanced Binary Tree

There's a lot of imbalance in the tree structure due to the sequential arrangement of nodes. This lack of balance results in a tree that resembles a singly linked list rather than a branching structure.

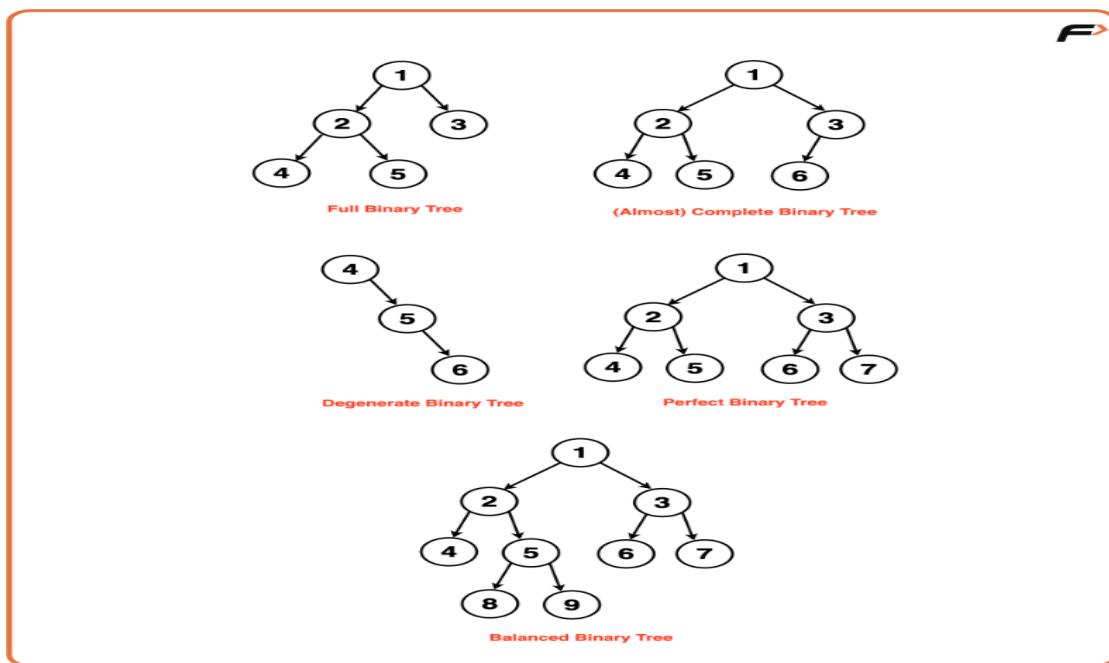
Each level of this tree only has one node. The height of tree reaches 'n' ie. the number of nodes in the tree, resulting in inefficient search operations. Though degenerate trees are not commonly used intentionally due to their inefficient nature for most operations, they may occur inadvertently in scenarios where nodes are inserted in a specific order (e.g., always to the right or left), causing the tree to lose its balanced properties.



Understanding degenerate trees is essential in analysing the worst-case time complexity of algorithms in scenarios where the tree structure degrades to this linear form, as it can help in optimising algorithms for these situations.

---

## In Summary:



In summary, Binary Trees introduce a hierarchical arrangement taking a step ahead of the linear structure we have studied so far.

Binary Trees comprise of nodes, each capable of hosting at most two children hence the predecessor ‘Binary’. These structures mirror the hierarchical organisation seen in file systems.

Full Binary Trees impose the constraint that each node possesses either zero or two children, promoting a well-balanced structure and enhancing predictability in operations like traversal and insertion.

Complete Binary Trees, on the other hand, embrace a specialized form where all levels, save possibly the last, are completely filled. Their design, ensuring nodes are positioned leftmost on the last level, proves valuable for efficient data storage and access, resembling the organized arrangement of folders and files in a computer system.

Perfect Binary Trees take this hierarchical order a step further, showcasing a balanced structure where all leaf nodes align at the same level. Such trees optimize space by filling all levels with nodes, creating a dense structure.

Balanced Binary Trees ensure the difference in heights between subtrees of any node remains minimal, preventing significant skewing or imbalance.

Degenerate Trees represent a case where nodes arrange linearly, akin to a linked list, posing inefficiencies in search operations due to the lack of balance.

## 2. Binary Tree Representation in C++

A binary tree is a hierarchical data structure where each node can have at most two children: a left child and a right child.

In C++, a binary tree is commonly represented using pointers. Each node stores its data and pointers to its left and right child nodes. These pointers connect nodes together and form the tree structure.

A node may point to:

- a left child
- a right child
- or none (leaf node)

## Approach 1

### Algorithm

We represent a binary tree using a Node class.

Each node contains:

- an integer value (data)
- a pointer to the left child
- a pointer to the right child

Steps:

- Define a Node class with data, left pointer, and right pointer.
- Use a constructor to initialize the node value and set child pointers to NULL.
- Create nodes dynamically using new.
- Connect nodes by assigning the left and right pointers of a parent node.

Example explanation:

- Create a root node with value 1.
- Assign its left child as a node with value 2.
- Assign its right child as a node with value 3.
- Assign a left child to node 3 with value 5.

This forms the binary tree structure using pointers.

### Code

```
#include <iostream>
```

```

using namespace std;

// Class representing a Node in the Binary Tree
class Node {
public:
 int data; // Value stored in the node
 Node* left; // Pointer to left child
 Node* right; // Pointer to right child

 // Constructor
 Node(int val) {
 data = val;
 left = NULL;
 right = NULL;
 }
};

// Class to create a Binary Tree
class Solution {
public:
 Node* createBinaryTree() {
 // Create root node
 Node* root = new Node(1);

 // Create left and right children
 root->left = new Node(2);
 root->right = new Node(3);

 // Create further child
 root->right->left = new Node(5);

 return root;
 }
};

// Driver code
int main() {
 Solution sol;

```

```
Node* root = sol.createBinaryTree();
return 0;
}
```

### Complexity Analysis

Time Complexity: O(N)

Creating N nodes takes linear time.

Space Complexity: O(N)

Memory is used to store N nodes of the binary tree.

## 3. Binary Tree Traversal : Inorder Preorder Postorder

Binary Tree traversal means visiting every node of the tree exactly once in a specific order.

Traversals are mainly divided into two types:

- Depth First Search (DFS): Goes deep into the tree before backtracking.
- Breadth First Search (BFS): Traverses the tree level by level.

DFS includes Inorder, Preorder, and Postorder traversals.

BFS includes Level Order traversal.

---

### Approach 1: Inorder Traversal (DFS)

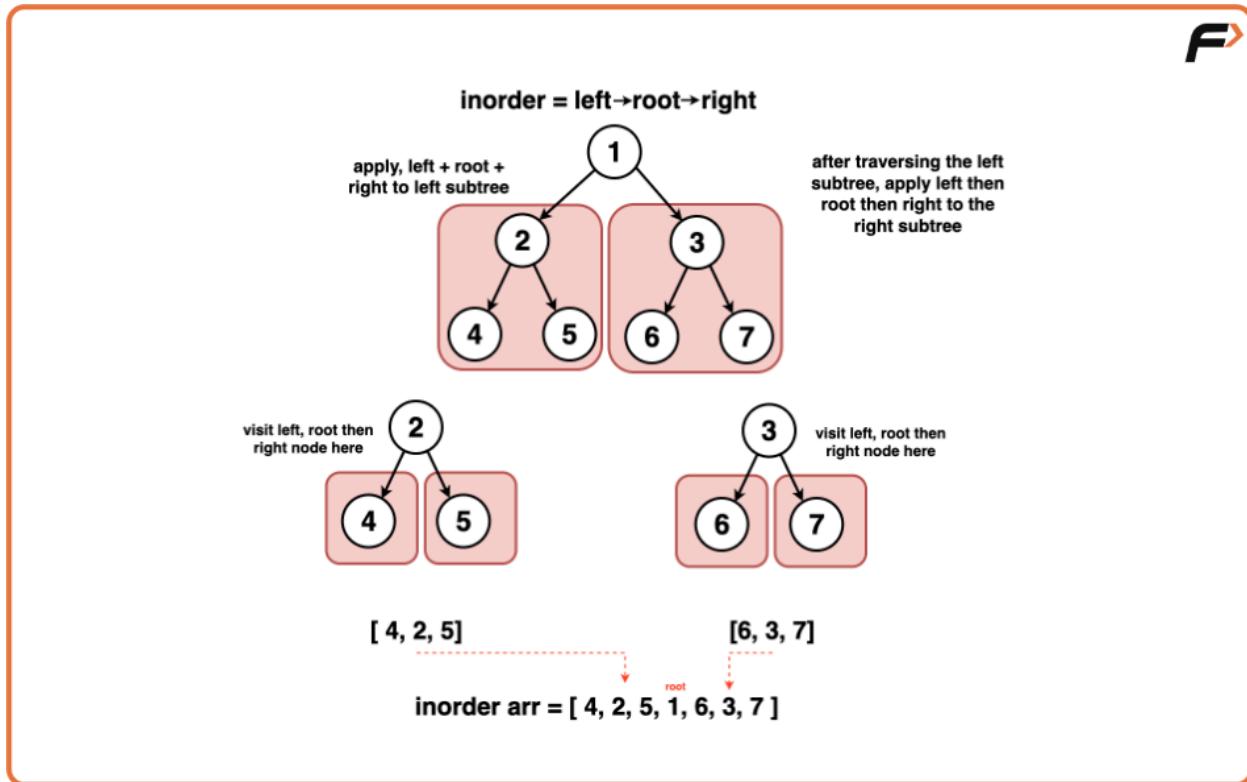
In inorder traversal, nodes are visited in the order:

Left → Root → Right

This means we first visit the left subtree, then the current node, and finally the right subtree.

Example explanation:

For a tree with root 1, left child 2, and right child 3,  
Inorder traversal will visit: 2 → 1 → 3



## Algorithm

- If the current node is NULL, return.
- Recursively traverse the left subtree.
- Visit the current node (print or store its value).
- Recursively traverse the right subtree.

## Code

```
#include <bits/stdc++.h>
using namespace std;
```

```

class Node {
public:
 int data;
 Node* left;
 Node* right;
 Node(int val) {
 data = val;
 left = NULL;
 right = NULL;
 }
};

void inorder(Node* root) {
 if (root == NULL) return;
 inorder(root->left);
 cout << root->data << " ";
 inorder(root->right);
}

int main() {
 Node* root = new Node(1);
 root->left = new Node(2);
 root->right = new Node(3);

 inorder(root);
 return 0;
}

```

### **Complexity Analysis**

Time Complexity:  $O(N)$   
 Each node is visited once.

Space Complexity:  $O(N)$   
 Recursive stack can go up to the height of the tree.

---

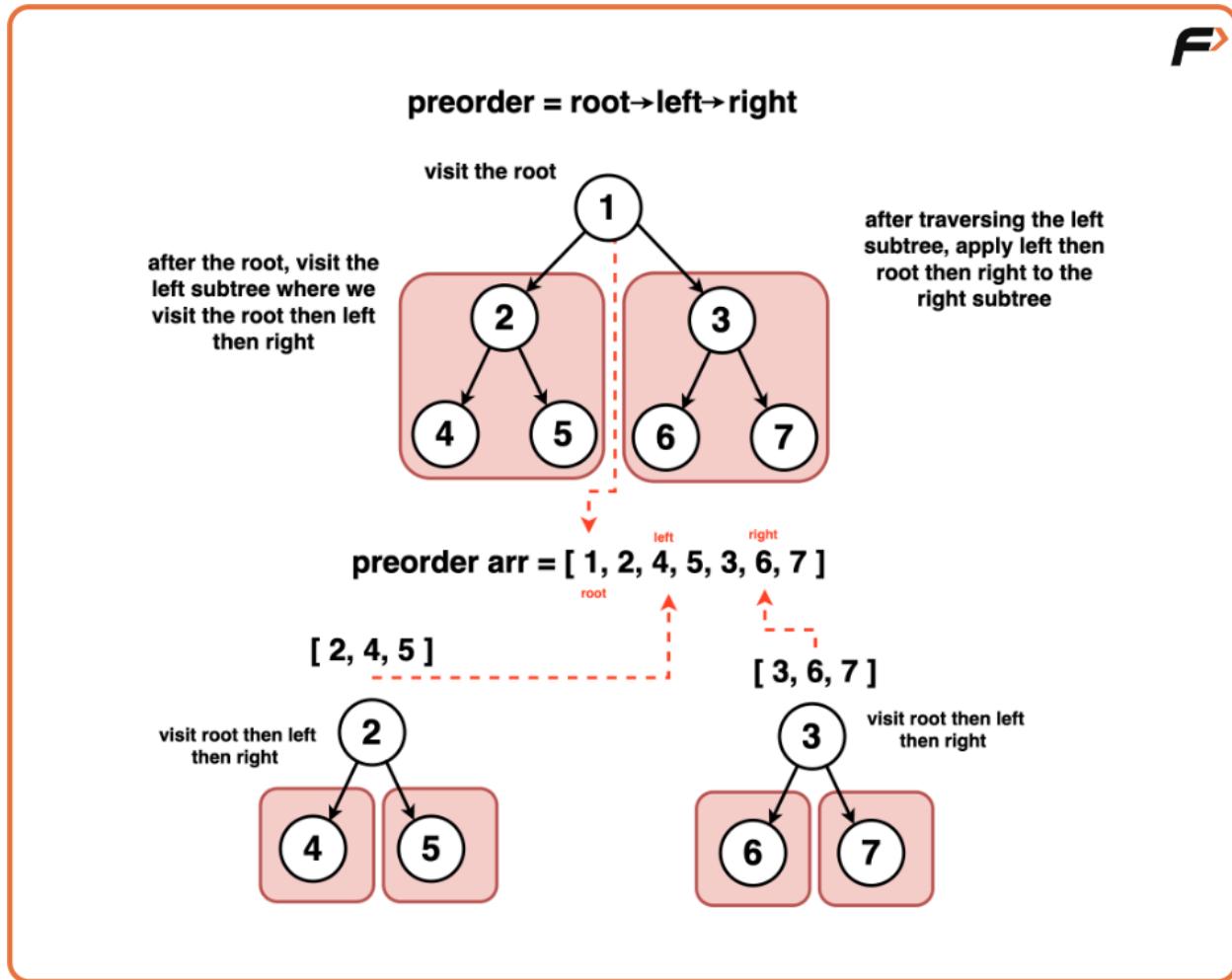
### **Approach 2: Preorder Traversal (DFS)**

In preorder traversal, nodes are visited in the order:  
Root → Left → Right

The current node is visited before its children.

Example explanation:

For the same tree, preorder traversal will visit: 1 → 2 → 3



## Algorithm

- If the current node is NULL, return.
- Visit the current node.
- Recursively traverse the left subtree.
- Recursively traverse the right subtree.

## Code

```
#include <bits/stdc++.h>
using namespace std;

void preorder(Node* root) {
 if (root == NULL) return;
 cout << root->data << " ";
 preorder(root->left);
 preorder(root->right);
}
```

## Complexity Analysis

Time Complexity:  $O(N)$

Each node is visited once.

Space Complexity:  $O(N)$

Space used by recursion stack.

---

## Approach 3: Postorder Traversal (DFS)

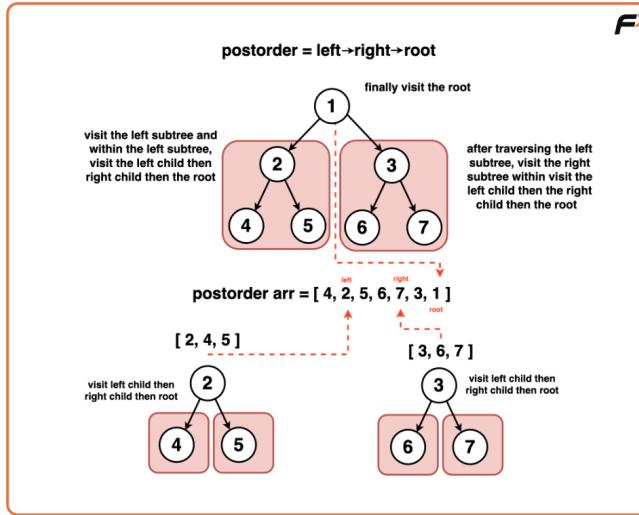
In postorder traversal, nodes are visited in the order:

Left → Right → Root

The current node is visited after both children.

Example explanation:

For the same tree, postorder traversal will visit: 2 → 3 → 1



## Algorithm

- If the current node is NULL, return.
- Recursively traverse the left subtree.
- Recursively traverse the right subtree.
- Visit the current node.

## Code

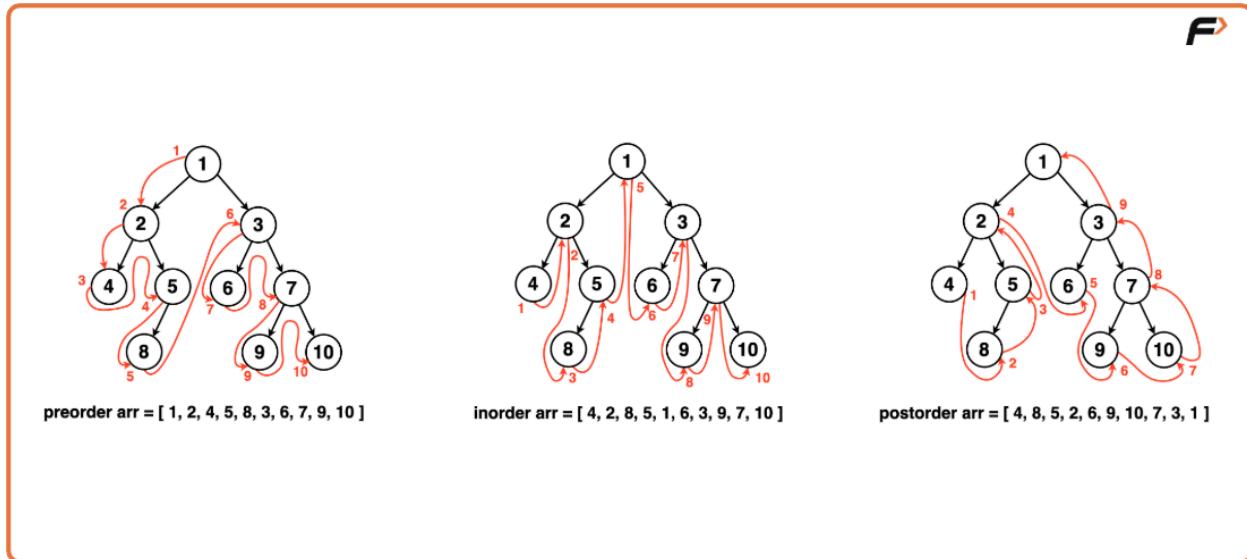
```
#include <bits/stdc++.h>
using namespace std;

void postorder(Node* root) {
 if (root == NULL) return;
 postorder(root->left);
 postorder(root->right);
 cout << root->data << " ";
}
```

## Complexity Analysis

Time Complexity: O(N)  
Each node is visited once.

Space Complexity:  $O(N)$   
Recursive stack space.



## Approach 4: Level Order Traversal (BFS)

Level order traversal visits nodes level by level from left to right.

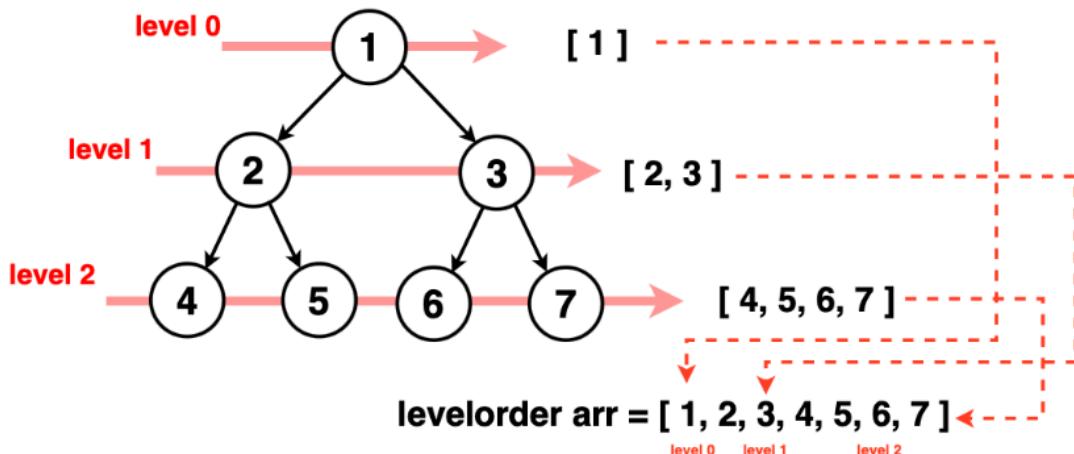
Example explanation:

For the tree, level order traversal will visit:  $1 \rightarrow 2 \rightarrow 3$

### Algorithm

- If root is NULL, return.
- Create a queue and push the root node.
- While the queue is not empty:
  - Pop the front node and visit it.
  - Push its left child if it exists.
  - Push its right child if it exists.

**Level Order = level by level**



## Code

```

#include <bits/stdc++.h>
using namespace std;

void levelOrder(Node* root) {
 if (root == NULL) return;
 queue<Node*> q;
 q.push(root);

 while (!q.empty()) {
 Node* curr = q.front();
 q.pop();
 cout << curr->data << " ";

 if (curr->left) q.push(curr->left);
 if (curr->right) q.push(curr->right);
 }
}

```

### **Complexity Analysis**

Time Complexity:  $O(N)$

Each node is visited once.

Space Complexity:  $O(N)$

Queue can store up to one full level of the tree.

## **4. Now Iterative**

### **Approach 1: Inorder Traversal (Iterative)**

In inorder traversal, we visit the left subtree first, then the root, and then the right subtree.

Using a stack, we keep moving left until we reach NULL, then process nodes while backtracking.

Example explanation:

For tree 1 2 3

Inorder result = 2 1 3

### **Algorithm**

- Create an empty stack.
- Set current node as root.
- While current is not NULL or stack is not empty:
  - Push current node and move to left child.
  - When NULL is reached, pop from stack.
  - Visit the node.
  - Move to right child.

### **Code**

```
#include <bits/stdc++.h>
```

```

using namespace std;

class Node {
public:
 int data;
 Node* left;
 Node* right;
 Node(int val) {
 data = val;
 left = right = NULL;
 }
};

void inorderIterative(Node* root) {
 stack<Node*> st;
 Node* curr = root;

 while (curr != NULL || !st.empty()) {
 while (curr != NULL) {
 st.push(curr);
 curr = curr->left;
 }
 curr = st.top();
 st.pop();
 cout << curr->data << " ";
 curr = curr->right;
 }
}

int main() {
 Node* root = new Node(1);
 root->left = new Node(2);
 root->right = new Node(3);

 inorderIterative(root);
 return 0;
}

```

## **Complexity Analysis**

Time Complexity: O(N)

Each node is pushed and popped once.

Space Complexity: O(N)

Stack space in worst case (skewed tree).

---

## **Approach 2: Preorder Traversal (Iterative)**

In preorder traversal, we visit the root first, then left subtree, then right subtree.

We use a stack and always process the current node before pushing its children.

Example explanation:

For tree 1 2 3

Preorder result = 1 2 3

## **Algorithm**

- If root is NULL, return.
- Push root into stack.
- While stack is not empty:
  - Pop top node and visit it.
  - Push right child first.
  - Push left child next.

## **Code**

```
#include <bits/stdc++.h>
using namespace std;

void preorderIterative(Node* root) {
 if (root == NULL) return;

 stack<Node*> st;
 st.push(root);
```

```

while (!st.empty()) {
 Node* curr = st.top();
 st.pop();
 cout << curr->data << " ";

 if (curr->right) st.push(curr->right);
 if (curr->left) st.push(curr->left);
}
}

```

### **Complexity Analysis**

Time Complexity: O(N)

Every node is visited once.

Space Complexity: O(N)

Stack space for nodes.

## **5. Level Order Traversal / Level Order Traversal in Spiral Form**

You are given a Binary Tree.

Your task is to traverse the tree:

- Level Order Traversal (Normal BFS)
- Level Order Traversal in Spiral (Zig-Zag) Form

Level Order traversal visits nodes **level by level from left to right**.

Spiral Level Order traversal alternates direction at each level:

- Level 0 → left to right
- Level 1 → right to left
- Level 2 → left to right, and so on.

## Approach 1: Level Order Traversal (BFS)

### Algorithm

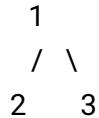
Level Order traversal uses a queue to process nodes level by level.

Steps:

- If root is NULL, return.
- Create a queue and push the root node.
- While the queue is not empty:
  - Take the front node from the queue.
  - Visit (print) the node.
  - Push its left child if it exists.
  - Push its right child if it exists.

Example explanation:

Tree:



Level order traversal → 1 2 3

### Code

```
#include <bits/stdc++.h>
using namespace std;

class Node {
public:
 int data;
 Node* left;
 Node* right;
 Node(int val) {
```

```

 data = val;
 left = right = NULL;
 }
};

void levelOrder(Node* root) {
 if (root == NULL) return;

 queue<Node*> q;
 q.push(root);

 while (!q.empty()) {
 Node* curr = q.front();
 q.pop();
 cout << curr->data << " ";

 if (curr->left) q.push(curr->left);
 if (curr->right) q.push(curr->right);
 }
}

int main() {
 Node* root = new Node(1);
 root->left = new Node(2);
 root->right = new Node(3);

 levelOrder(root);
 return 0;
}

```

### Complexity Analysis

Time Complexity:  $O(N)$

Each node is visited once.

Space Complexity:  $O(N)$

Queue can store nodes of one complete level.

---

## Approach 2: Level Order Traversal in Spiral Form (Zig-Zag)

### Algorithm

Spiral traversal changes direction at every level.

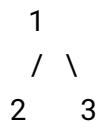
We use a queue for level processing and a flag to control direction.

Steps:

- If root is NULL, return.
- Push root into a queue.
- Use a boolean flag `leftToRight` initially true.
- While queue is not empty:
  - Get the number of nodes at the current level.
  - Create a temporary array of that size.
  - For each node in the level:
    - Pop node from queue.
    - Place its value in correct position based on direction.
    - Push left and right children into the queue.
  - Print the temporary array.
  - Flip the direction flag.

Example explanation:

Tree:



Spiral order traversal → 1 3 2

### Code

```

#include <bits/stdc++.h>
using namespace std;

void spiralLevelOrder(Node* root) {
 if (root == NULL) return;

 queue<Node*> q;
 q.push(root);
 bool leftToRight = true;

 while (!q.empty()) {
 int size = q.size();
 vector<int> level(size);

 for (int i = 0; i < size; i++) {
 Node* curr = q.front();
 q.pop();

 int index = leftToRight ? i : size - 1 - i;
 level[index] = curr->data;

 if (curr->left) q.push(curr->left);
 if (curr->right) q.push(curr->right);
 }

 for (int val : level)
 cout << val << " ";
 }

 leftToRight = !leftToRight;
}
}

```

### Complexity Analysis

Time Complexity: O(N)  
 Each node is processed once.

Space Complexity: O(N)

Queue and temporary vector store nodes of one level.

# 6. Iterative Postorder Traversal of Binary Tree Using 2 Stacks

Given the root of a Binary Tree, we need to perform **postorder traversal** without using recursion.

Postorder traversal follows the order: **Left → Right → Root**.

The task is to return the traversal sequence using **two stacks**.

Example explanation (simple idea):

In recursive postorder, the root is processed after both children.

Using two stacks helps us reverse a modified preorder traversal to achieve postorder.

---

## Approach 1: Using Two Stacks

### Algorithm

We use two stacks to simulate postorder traversal.

Steps:

- If the root is NULL, return an empty list.
- Create two stacks: st1 and st2.
- Push the root node into st1.
- While st1 is not empty:
  - Pop a node from st1.
  - Push this node into st2.
  - Push the left child of the node into st1 if it exists.
  - Push the right child of the node into st1 if it exists.
- After the loop, st2 contains nodes in reverse postorder.

- Pop all nodes from st2 and store their values in the result array.
- Return the result array.

Dry run (simple tree):

```

1
/
2 3

```

- st1 process order: 1 → 2 → 3
- st2 stores: 1, 3, 2
- Final pop from st2 gives: 2 3 1 (postorder)

**Postorder = reverse of (Root → Right → Left)**

### Code

```

#include <bits/stdc++.h>
using namespace std;

struct Node {
 int data;
 Node* left;
 Node* right;
 Node(int val) {
 data = val;
 left = right = NULL;
 }
};

vector<int> postOrder(Node* root) {
 vector<int> postorder;
 if (root == NULL) return postorder;

 stack<Node*> st1, st2;
 st1.push(root);

```

```

 while (!st1.empty()) {
 Node* curr = st1.top();
 st1.pop();
 st2.push(curr);

 if (curr->left)
 st1.push(curr->left);
 if (curr->right)
 st1.push(curr->right);
 } //Root → Right → Left

 while (!st2.empty()) {
 postorder.push_back(st2.top()->data);
 st2.pop();
 } //Left → Right → Root

 return postorder;
 }

int main() {
 Node* root = new Node(1);
 root->left = new Node(2);
 root->right = new Node(3);
 root->left->left = new Node(4);
 root->left->right = new Node(5);

 vector<int> result = postOrder(root);
 for (int x : result) cout << x << " ";
 return 0;
}

```

### Complexity Analysis

Time Complexity: O(N)

Each node is pushed and popped a constant number of times.

Space Complexity: O(N)

Two stacks together can store up to N nodes in the worst case.

## 7. Post-order Traversal of Binary Tree using 1 Stack

You are given the root of a Binary Tree.

Your task is to perform **postorder traversal** using **only one stack** (no recursion, no second stack) and return the traversal sequence.

Postorder traversal follows the order:

**Left → Right → Root**

This approach is more tricky than using two stacks because we must carefully track when a node's right subtree has already been processed.

Stack se recursion simulate karo aur lastVisited se yaad rakho

ki right subtree already process ho chuki hai ya nahi.

### **lastVisited ka role**

Ye batata hai:

“Kya main is node ke right subtree se wapas aa chuka hoon?”

---

### **Approach: Iterative Postorder Traversal Using One Stack**

#### **Algorithm**

The idea is to simulate recursion using one stack and a pointer to track the last visited node.

Steps:

- Create an empty stack.
- Set `curr` to root and `lastVisited` to NULL.
- While `curr` is not NULL or stack is not empty:
  - If `curr` is not NULL:
    - Push `curr` into stack.
    - Move `curr` to its left child.
  - Else:
    - Peek the top node of stack.
    - If the right child exists and has not been visited yet:
      - Move `curr` to the right child.
    - Else:
      - Visit the node (add to result).
      - Mark it as last visited.
      - Pop it from stack.

This ensures that a node is processed only **after** both left and right subtrees are completed.

Example dry run (simple tree):



Traversal order will be: 2 → 3 → 1

## Code

```

#include <bits/stdc++.h>

using namespace std;

struct Node {
 int data;
 Node* left;
 Node* right;
 Node(int val) {
 data = val;
 left = right = NULL;
 }
};

vector<int> postOrder(Node* root) {
 vector<int> result;
 stack<Node*> st;
 Node* curr = root;
 Node* lastVisited = NULL;

 while (curr != NULL || !st.empty()) {
 if (curr != NULL) {
 st.push(curr);
 curr = curr->left;
 } else {
 curr = st.top();
 st.pop();
 curr = curr->right;
 }
 if (curr == NULL && lastVisited == curr) {
 result.push_back(lastVisited->data);
 lastVisited = NULL;
 } else if (curr != NULL) {
 lastVisited = curr;
 }
 }
 return result;
}

```

```

 } else {

 Node* topNode = st.top();

 if (topNode->right != NULL && lastVisited != topNode->right) {

 curr = topNode->right;

 } else {

 result.push_back(topNode->data);

 lastVisited = topNode;

 st.pop();

 }

 }

 return result;
}

int main() {

 Node* root = new Node(1);

 root->left = new Node(2);

 root->right = new Node(3);

 root->left->left = new Node(4);

 root->left->right = new Node(5);

 vector<int> ans = postOrder(root);
}

```

```
 for (int x : ans) cout << x << " ";
 return 0;
}
```

### Complexity Analysis

Time Complexity:  $O(N)$

Each node is pushed and popped from the stack exactly once.

Space Complexity:  $O(N)$

Stack can store up to  $N$  nodes in the worst case (skewed tree).

## 8. Preorder Inorder Postorder Traversals in One Traversal

You are given the root of a Binary Tree.

Your task is to return **preorder**, **inorder**, and **postorder** traversal sequences **by making only one traversal of the tree**.

Normally, each traversal requires a separate pass. Here, we do all three in a single traversal using a stack and a state mechanism.

---

### Approach 1

#### Algorithm

We simulate recursion using a stack that stores a pair of:

- a tree node
- an integer state representing the traversal phase of that node

State meaning:

- 1 → Preorder state
- 2 → Inorder state
- 3 → Postorder state

Steps:

- If the tree is empty, return empty traversals.
- Create three arrays: pre, in, and post.
- Use a stack that stores {node, state}.
- Push the root node with state 1 onto the stack.
- While the stack is not empty:
  - Pop the top element.
  - If state is 1:
    - Add node value to preorder.
    - Change state to 2 and push it back.
    - Push left child with state 1 if it exists.
  - If state is 2:
    - Add node value to inorder.
    - Change state to 3 and push it back.
    - Push right child with state 1 if it exists.
  - If state is 3:
    - Add node value to postorder.
- Return all three traversal arrays.

Example dry run (simple tree):

```
1
 / \
2 3
```

Preorder → 1 2 3

Inorder → 2 1 3

Postorder → 2 3 1

All are produced in one traversal.

(node, state)

| stat | Meaning |
|------|---------|
| e    |         |

1 node pe **first time** aaye

2 left subtree ho chuki

3 left + right dono ho chuke

#### ◆ State ka mapping

| State | Action |
|-------|--------|
|-------|--------|

1 Preorder → left jao

2 Inorder → right jao

3 Postorder → done

## Code

```
#include <bits/stdc++.h>

using namespace std;

// Node structure

struct Node {

 int data;

 Node* left;

 Node* right;

 Node(int val) {

 data = val;

 left = right = NULL;

 }

};

class Solution {

public:

 vector<vector<int>> preInPostTraversal(Node* root) {
```

```

vector<int> pre, in, post;

if (root == NULL)
 return {};

stack<pair<Node*, int>> st;
st.push({root, 1});

while (!st.empty()) {
 auto it = st.top();
 st.pop();

 if (it.second == 1) {
 pre.push_back(it.first->data);
 it.second = 2;
 st.push(it);

 if (it.first->left)
 st.push({it.first->left, 1});
 }
 else if (it.second == 2) {
 in.push_back(it.first->data);
 it.second = 3;
 }
}

```

```

 st.push(it);

 if (it.first->right)
 st.push({it.first->right, 1});

 }

 else {
 post.push_back(it.first->data);
 }

}

return {pre, in, post};
};

};

int main() {

 Node* root = new Node(1);

 root->left = new Node(2);

 root->right = new Node(3);

 root->left->left = new Node(4);

 root->left->right = new Node(5);

 Solution sol;

 vector<vector<int>> res = sol.preInPostTraversal(root);
}

```

```

cout << "Preorder: ";
for (int x : res[0]) cout << x << " ";
cout << endl;

cout << "Inorder: ";
for (int x : res[1]) cout << x << " ";
cout << endl;

cout << "Postorder: ";
for (int x : res[2]) cout << x << " ";
cout << endl;

return 0;
}

```

### **Complexity Analysis**

Time Complexity: O(N)

Each node is processed a constant number of times.

Space Complexity: O(N)

Space is used for the stack and for storing the three traversal arrays.

# 9. Maximum Depth of a Binary Tree

You are given the root of a Binary Tree.

Your task is to find the **height (maximum depth)** of the tree.

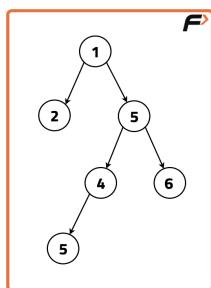
The height of a binary tree is defined as the **number of nodes on the longest path from the root node to any leaf node**.

Example:

Binary Tree: 1 2 5 -1 -1 4 6 5

The longest path is 1 → 5 → 4 → 5

So, the height of the tree is 4.



---

## Approach 1

### Algorithm

This approach uses **recursion** to calculate the height of the tree.

Steps:

- If the current node is NULL, return 0 (empty tree has height 0).
- Recursively calculate the height of the left subtree.
- Recursively calculate the height of the right subtree.
- The height of the current node is  $1 + \max(\text{leftHeight}, \text{rightHeight})$ .
- Return this value.

Example explanation:

For a tree with root 1:

- Height of left subtree = 3
- Height of right subtree = 1
- Height of tree =  $1 + \max(3, 1) = 4$

## Code

```
#include <bits/stdc++.h>
using namespace std;

// Node structure for the binary tree
struct Node {
 int data;
 Node* left;
 Node* right;
 Node(int val) : data(val), left(nullptr), right(nullptr) {}
};

class Solution {
public:
 int maxDepth(Node* root) {
 if (root == NULL) {
 return 0;
 }

 int lh = maxDepth(root->left);
 int rh = maxDepth(root->right);

 return 1 + max(lh, rh);
 }
};

// Driver code
int main() {
 Node* root = new Node(1);
 root->left = new Node(2);
 root->right = new Node(3);
 root->left->left = new Node(4);
```

```

root->left->right = new Node(5);
root->left->right->right = new Node(6);
root->left->right->right->right = new Node(7);

Solution solution;
cout << solution.maxDepth(root);
return 0;
}

```

### **Complexity Analysis**

Time Complexity: O(N)

Each node of the binary tree is visited exactly once.

Space Complexity: O(N)

In the worst case (skewed tree), the recursion stack can go up to N nodes.

## **10. Check if the Binary Tree is Balanced Binary Tree**

You are given a Binary Tree.

Your task is to check whether the tree is **balanced** or not.

A Binary Tree is said to be **balanced** if for **every node**, the absolute difference between the height of its left subtree and right subtree is **not more than 1**.

Example:

Input: [3, 9, 20, null, null, 15, 7]

Output: Yes

Because for every node, the height difference between left and right subtrees is at most 1.

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

## Algorithm

In this approach, we calculate the height of left and right subtrees for every node and check the balance condition.

Steps:

- If the root is NULL, return true (empty tree is balanced).
- Compute height of the left subtree.
- Compute height of the right subtree.
- Check if  $\text{abs}(\text{leftHeight} - \text{rightHeight}) \leq 1$ .
- Recursively check if left subtree is balanced.
- Recursively check if right subtree is balanced.
- If all conditions are satisfied, return true; otherwise return false.

Example explanation:

At each node, height is calculated again and again for subtrees, which causes repeated work.

## Code

```
#include <bits/stdc++.h>
using namespace std;

struct Node {
 int data;
 Node* left;
 Node* right;
 Node(int val) : data(val), left(NULL), right(NULL) {}
};

class Solution {
public:
 bool isBalanced(Node* root) {
 if (root == NULL)
 return true;
```

```

 int leftHeight = getHeight(root->left);
 int rightHeight = getHeight(root->right);

 if (abs(leftHeight - rightHeight) <= 1 &&
 isBalanced(root->left) &&
 isBalanced(root->right))
 return true;

 return false;
 }

 int getHeight(Node* root) {
 if (root == NULL)
 return 0;

 int lh = getHeight(root->left);
 int rh = getHeight(root->right);
 return max(lh, rh) + 1;
 }
};

int main() {
 Node* root = new Node(1);
 root->left = new Node(2);
 root->right = new Node(3);
 root->left->left = new Node(4);
 root->left->right = new Node(5);
 root->left->right->right = new Node(6);
 root->left->right->right->right = new Node(7);

 Solution sol;
 cout << sol.isBalanced(root);
 return 0;
}

```

### Complexity Analysis

Time Complexity:  $O(N^2)$

For every node, height calculation takes  $O(N)$  in the worst case.

Space Complexity:  $O(H)$

Recursive call stack space, where  $H$  is the height of the tree.

---

## Approach 2: Optimal Approach

### Algorithm

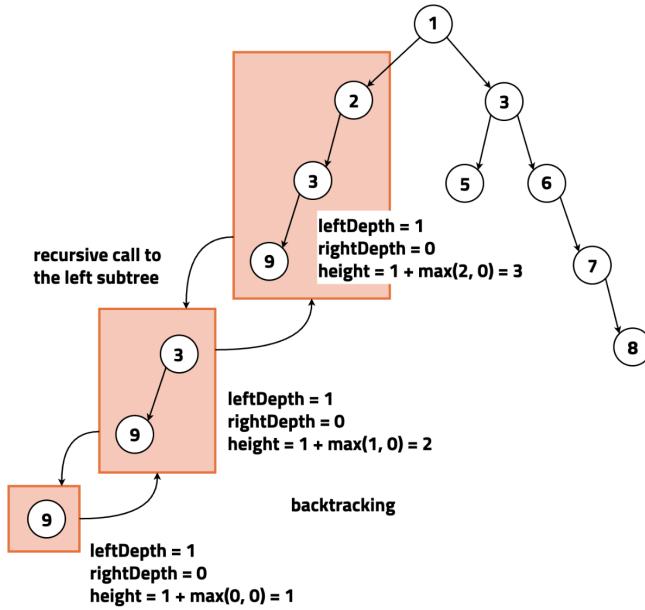
This approach avoids repeated height calculations by combining height computation and balance checking in a **single postorder traversal**.

Steps:

- Traverse the tree using postorder (Left → Right → Root).
- For each node:
  - Get height of left subtree.
  - Get height of right subtree.
- If any subtree is already unbalanced, return -1.
- If  $\text{abs}(\text{leftHeight} - \text{rightHeight}) > 1$ , return -1.
- Otherwise, return  $\max(\text{leftHeight}, \text{rightHeight}) + 1$ .
- If final result is not -1, the tree is balanced.

Example explanation:

The moment an unbalanced node is found, we stop further computation and propagate -1 upward.



## Code

```
#include <bits/stdc++.h>
using namespace std;

struct Node {
 int data;
 Node* left;
 Node* right;
 Node(int val) : data(val), left(NULL), right(NULL) {}
};

class Solution {
public:
 bool isBalanced(Node* root) {
 return dfsHeight(root) != -1;
 }

 int dfsHeight(Node* root) {
```

```

 if (root == NULL)
 return 0;

 int lh = dfsHeight(root->left);
 if (lh == -1)
 return -1;

 int rh = dfsHeight(root->right);
 if (rh == -1)
 return -1;

 if (abs(lh - rh) > 1)
 return -1;

 return max(lh, rh) + 1;
 }
};

int main() {
 Node* root = new Node(1);
 root->left = new Node(2);
 root->right = new Node(3);
 root->left->left = new Node(4);
 root->left->right = new Node(5);
 root->left->right->right = new Node(6);
 root->left->right->right->right = new Node(7);

 Solution sol;
 cout << sol.isBalanced(root);
 return 0;
}

```

### Complexity Analysis

Time Complexity: O(N)

Each node is visited exactly once.

Space Complexity: O(H)

Only recursion stack space is used, where H is the height of the tree.

# 11. Calculate the Diameter of a Binary Tree

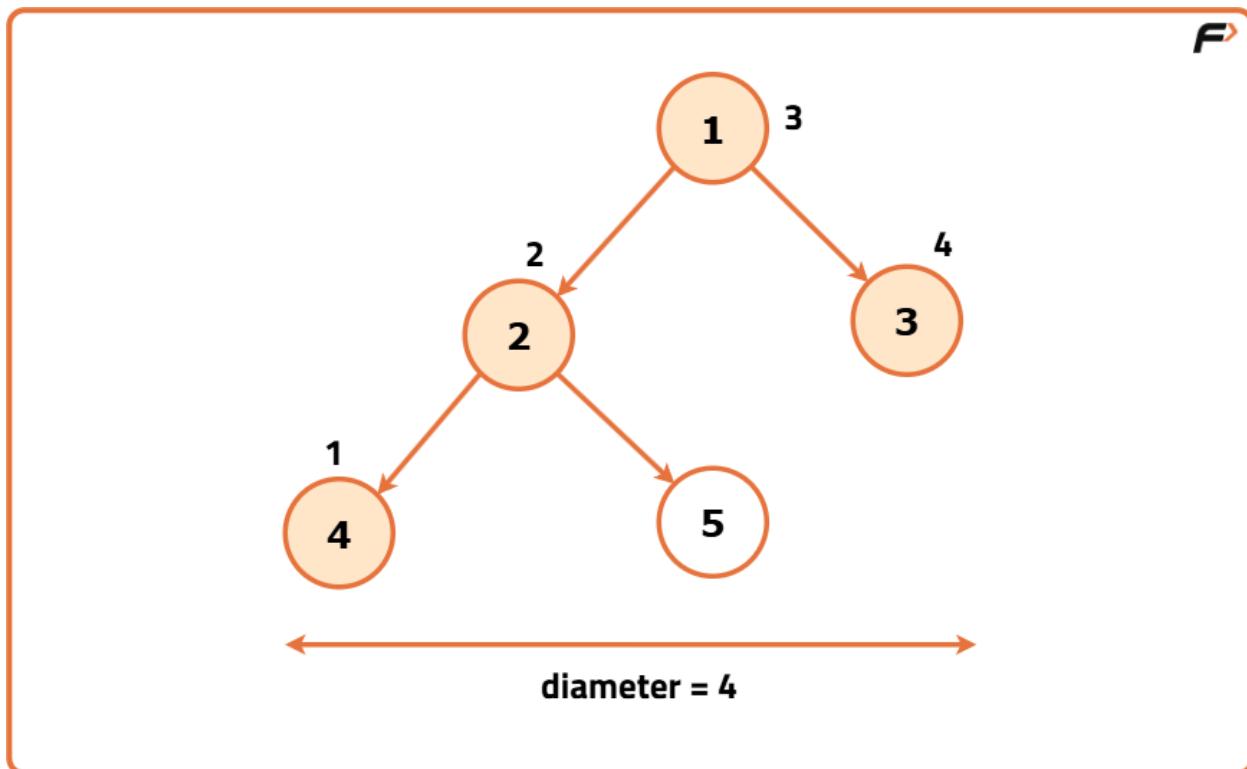
The diameter of a Binary Tree is defined as the **longest distance between any two nodes** in the tree.

This distance is measured as the **number of edges** on the longest path.

The path may or may not pass through the root.

Diameter = tree ke kisi bhi do nodes ke beech ka longest path (edges count)

⚠ Path root se jaana zaroori nahi



## Approach 1: Brute Force Approach

In this approach, we consider **every node as a turning point** of the diameter.

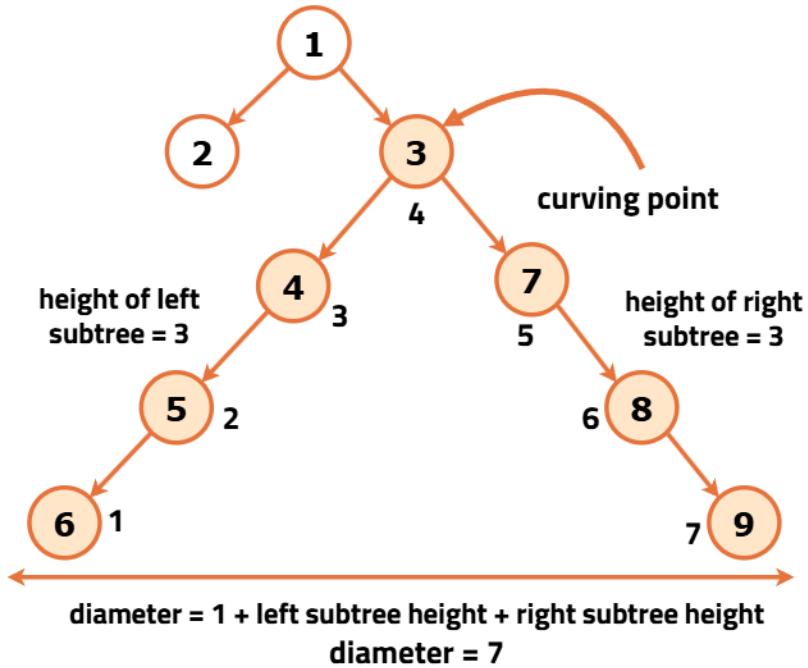
At each node:

- Find the height of the left subtree.
- Find the height of the right subtree.
- The diameter passing through that node is `leftHeight + rightHeight`.

We calculate this for all nodes and keep track of the maximum value.

### Algorithm

- Initialize a variable `diameter` to store the maximum diameter.
- Define a recursive function to calculate height.
- For each node:
  - Recursively compute left subtree height.
  - Recursively compute right subtree height.
  - Update diameter as `max(diameter, leftHeight + rightHeight)`.
- Return the diameter.



## Code

```
#include <bits/stdc++.h>
using namespace std;

struct Node {
 int data;
 Node* left;
 Node* right;
 Node(int val) : data(val), left(NULL), right(NULL) {}
};

class Solution {
public:
 int diameter = 0;

 int height(Node* root) {
 if (root == NULL)
 return 0;
 else
 return max(height(root->left), height(root->right)) + 1;
 }

 void calculateDiameter(Node* root) {
 if (root == NULL)
 return;
 diameter = max(diameter, height(root->left) + height(root->right));
 calculateDiameter(root->left);
 calculateDiameter(root->right);
 }
};
```

```

 return 0;

 int lh = height(root->left);
 int rh = height(root->right);

 diameter = max(diameter, lh + rh);

 return 1 + max(lh, rh);
 }

 int diameterOfBinaryTree(Node* root) {
 height(root);
 return diameter;
 }
};

int main() {
 Node* root = new Node(1);
 root->left = new Node(2);
 root->right = new Node(3);
 root->left->left = new Node(4);
 root->left->right = new Node(5);
 root->left->right->right = new Node(6);
 root->left->right->right->right = new Node(7);

 Solution sol;
 cout << sol.diameterOfBinaryTree(root);
 return 0;
}

```

## Complexity Analysis

- Time Complexity: **O(N<sup>2</sup>)**  
Height is recomputed for each node.
- Space Complexity: **O(H)**  
Due to recursion stack, where H is the height of the tree.

---

## Approach 2: Optimal Approach (Single Traversal)

The brute force approach is inefficient because heights are recalculated many times.

We can optimize this by calculating **height and diameter together** in a single postorder traversal.

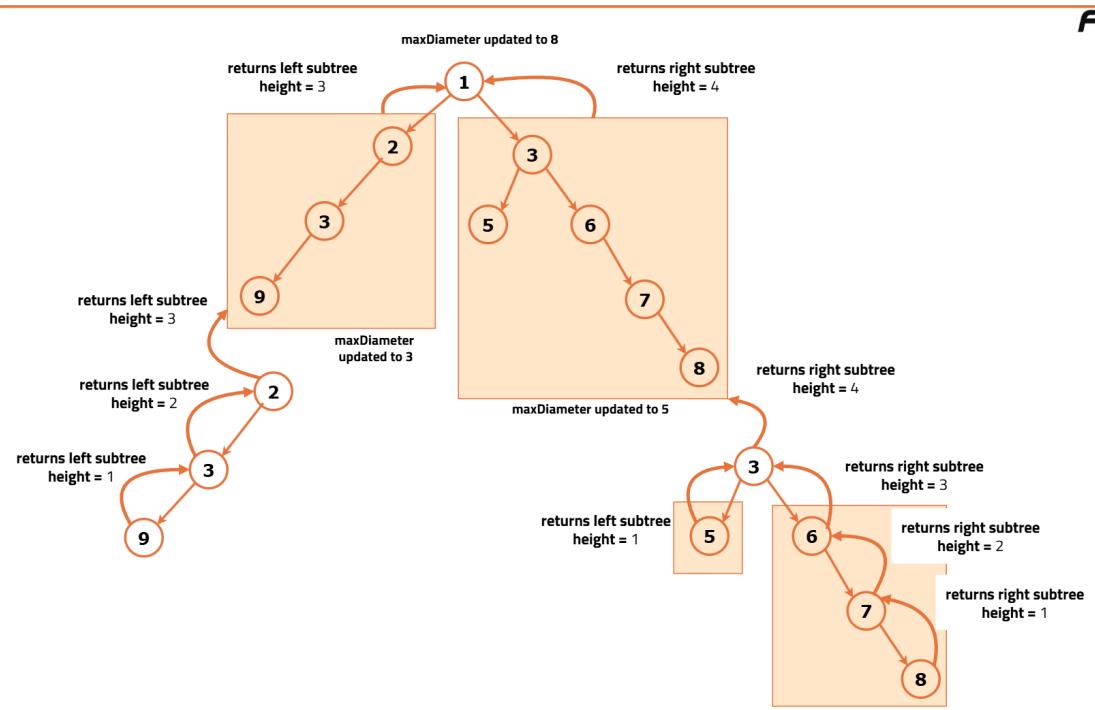
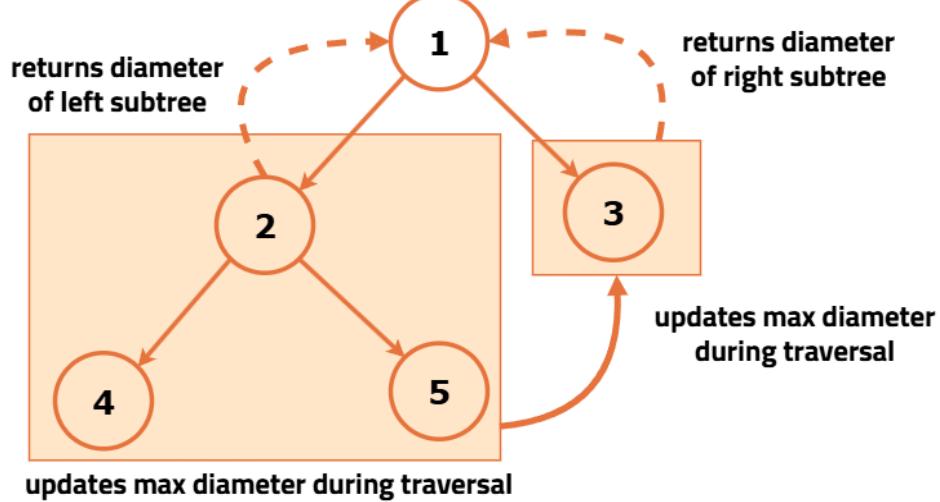
At each node:

- Get left subtree height.
- Get right subtree height.
- Update diameter.
- Return the height to the parent.

This avoids repeated height calculations.

### Algorithm

- Initialize diameter as 0.
- Use a recursive postorder function.
- For each node:
  - Compute left height.
  - Compute right height.
  - Update diameter as  $\max(\text{diameter}, \text{leftHeight} + \text{rightHeight})$ .
  - Return  $1 + \max(\text{leftHeight}, \text{rightHeight})$ .
- After traversal, return diameter.



## Code

```
#include <bits/stdc++.h>
using namespace std;
```

```

struct Node {
 int data;
 Node* left;
 Node* right;
 Node(int val) : data(val), left(NULL), right(NULL) {}
};

class Solution {
public:
 int diameterOfBinaryTree(Node* root) {
 int diameter = 0;
 height(root, diameter);
 return diameter;
 }

 int height(Node* root, int& diameter) {
 if (root == NULL)
 return 0;

 int lh = height(root->left, diameter);
 int rh = height(root->right, diameter);

 diameter = max(diameter, lh + rh);

 return 1 + max(lh, rh);
 }
};

int main() {
 Node* root = new Node(1);
 root->left = new Node(2);
 root->right = new Node(3);
 root->left->left = new Node(4);
 root->left->right = new Node(5);
 root->left->right->right = new Node(6);
 root->left->right->right->right = new Node(7);

 Solution sol;

```

```

cout << sol.diameterOfBinaryTree(root);
return 0;
}

```

### Complexity Analysis

- Time Complexity: **O(N)**  
Each node is visited exactly once.
- Space Complexity: **O(H)**  
Recursion stack space, where H is the height of the tree.

## 12. Maximum Sum Path in Binary Tree

You are given a Binary Tree.

Your task is to find the **maximum sum possible along any path in the tree**.

A path:

- Can start and end at **any node**.
- Must follow parent–child connections.
- Cannot visit the same node more than once.
- Does **not** need to pass through the root.

### Question Explanation

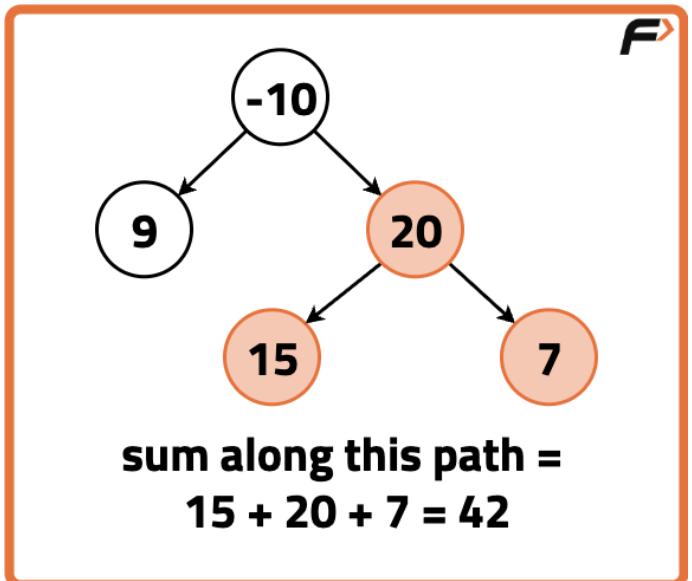
We need to find a path in the binary tree such that the **sum of node values on that path is maximum**.

Example 1:

Binary Tree: -10 9 20 -1 -1 15 7

The path  $15 \rightarrow 20 \rightarrow 7$  gives the maximum sum.

$$\text{Sum} = 15 + 20 + 7 = 42$$

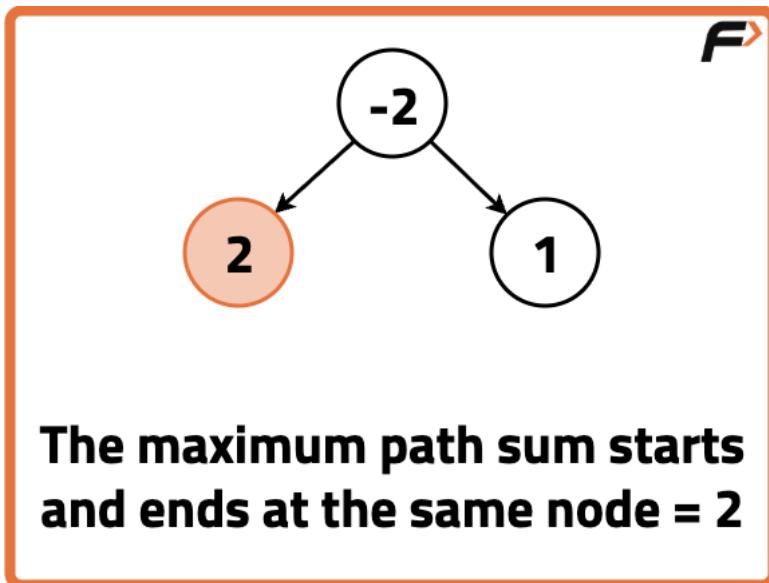


Example 2:

Binary Tree: -2 2 1

The best path is just the node 2.

$$\text{Sum} = 2$$



---

## Approach

## Algorithm

We solve this using recursion and treat **every node as a possible turning point** of the path.

At each node, we calculate two things:

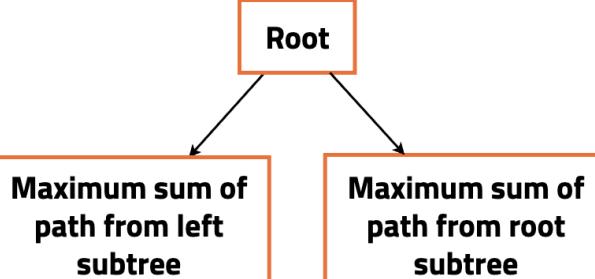
1. **Maximum path sum passing through the node**  
= node value + left subtree contribution + right subtree contribution  
This is used to update the global answer.
2. **Maximum one-sided path sum to return to parent**  
= node value + max(left contribution, right contribution)

Important points:

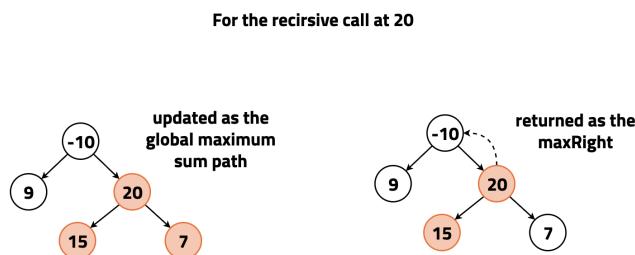
- If a subtree gives a negative sum, we ignore it (use 0 instead).
- We keep a global variable to store the maximum path sum found so far.

Steps:

- If the node is NULL, return 0.
- Recursively compute left and right path sums.
- Ignore negative sums by taking  $\max(0, \text{left/right})$ .
- Update the global maximum using `left + right + node value`.
- Return the best one-sided path to the parent.



**maximum sum of path from node when node is considered a turning point**



maximum sum path considering 20 as the turning point =  $20 + 15 + 7$

this maximum path sum is also compared and updated as the global maximum

$$= \text{current node's value} + \text{maxLeft} + \text{maxRight}$$

we return the maximum sum, considering either the left or right branch, along with the current node's value of the maximum sum up to this node for the recursive function

$$= \text{current node's value} + \max(\text{maxLeft}, \text{maxRight})$$

## Code

```

#include <bits/stdc++.h>
using namespace std;

// Tree Node Definition
struct TreeNode {
 int val;
 TreeNode* left;
 TreeNode* right;
}

```

```

TreeNode(int x) {
 val = x;
 left = NULL;
 right = NULL;
}
};

class Solution {
public:
 int maxPathSum(TreeNode* root) {
 int maxSum = INT_MIN;
 dfs(root, maxSum);
 return maxSum;
 }

 int dfs(TreeNode* node, int &maxSum) {
 if (node == NULL)
 return 0;

 int left = max(0, dfs(node->left, maxSum));
 int right = max(0, dfs(node->right, maxSum));

 maxSum = max(maxSum, left + right + node->val);

 return max(left, right) + node->val;
 }
};

// Driver Code
int main() {
 TreeNode* root = new TreeNode(-10);
 root->left = new TreeNode(9);
 root->right = new TreeNode(20);
 root->right->left = new TreeNode(15);
 root->right->right = new TreeNode(7);

 Solution obj;
 cout << obj.maxPathSum(root);
}

```

```
 return 0;
}
```

---

## Complexity Analysis

### Time Complexity: O(N)

Each node is visited exactly once during DFS traversal.

### Space Complexity: O(H)

Auxiliary space used by recursion stack, where H is the height of the binary tree.

# 13. Check if Two Trees are Identical

Given two Binary Trees, your task is to check whether they are **identical** or not.

Two binary trees are said to be identical if **all the following conditions are true for every corresponding node:**

- The values of the nodes are equal.
  - The left subtrees are identical.
  - The right subtrees are identical.
- 

## Question Explanation

We compare both trees **node by node**.

- If both nodes are NULL, they are identical up to this point.
- If one node is NULL and the other is not, the trees are not identical.
- If both nodes exist but their values are different, the trees are not identical.

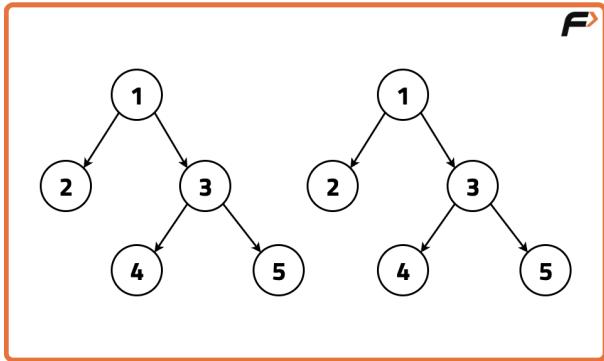
- Otherwise, we recursively compare the left children and the right children.

Example 1:

Tree 1: 1 2 3 -1 -1 4 5

Tree 2: 1 2 3 -1 -1 4 5

Both structure and values match → **True**

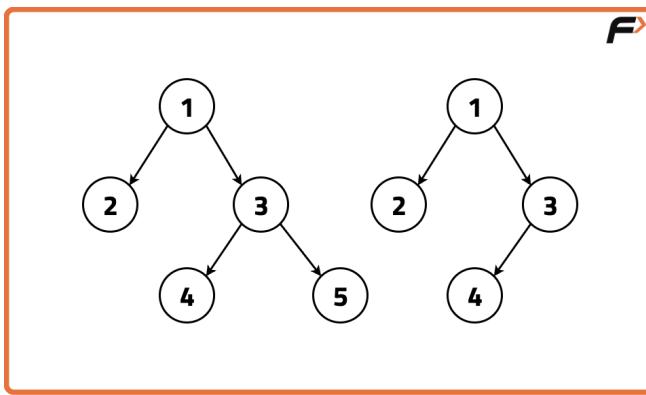


Example 2:

Tree 1: 1 2 3 -1 -1 4 5

Tree 2: 1 2 3 -1 -1 4

Structure differs → **False**



## Approach

### Algorithm

- If both current nodes are NULL, return true.

- If only one node is NULL, return false.
- If values of current nodes are not equal, return false.
- Recursively check:
  - Left subtree of both trees
  - Right subtree of both trees
- If all checks pass, return true.

This ensures both **structure** and **data** are the same.

---

## Code

```
#include <bits/stdc++.h>
using namespace std;

// Node structure for the binary tree
struct Node {
 int data;
 Node* left;
 Node* right;
 Node(int val) : data(val), left(NULL), right(NULL) {}
};

class Solution {
public:
 bool isIdentical(Node* node1, Node* node2) {
 if (node1 == NULL && node2 == NULL)
 return true;

 if (node1 == NULL || node2 == NULL)
 return false;

 if (node1->data != node2->data)
 return false;
 }
};
```

```

 return isIdentical(node1->left, node2->left) &&
 isIdentical(node1->right, node2->right);
 }
};

int main() {
 Node* root1 = new Node(1);
 root1->left = new Node(2);
 root1->right = new Node(3);
 root1->left->left = new Node(4);

 Node* root2 = new Node(1);
 root2->left = new Node(2);
 root2->right = new Node(3);
 root2->left->left = new Node(4);

 Solution sol;
 cout << sol.isIdentical(root1, root2);
 return 0;
}

```

---

## Complexity Analysis

### Time Complexity: $O(N + M)$

Each node of both trees is visited once, where N and M are the number of nodes in the two trees.

### Space Complexity: $O(H)$

Only recursion stack space is used, where H is the height of the tree.

# 14. Zig Zag Traversal Of Binary Tree

Given a Binary Tree, print its **zigzag (spiral) level order traversal**.

In zigzag traversal, the direction of traversal alternates at each level.

- Level 1: Left to Right
- Level 2: Right to Left
- Level 3: Left to Right
- and so on.

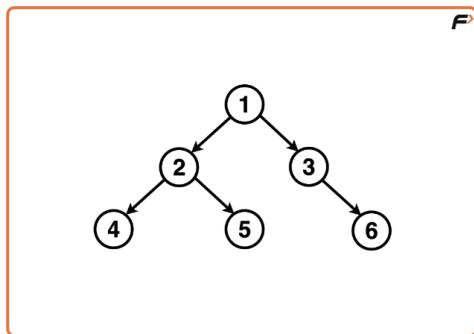
---

## Question Explanation

We traverse the tree **level by level** like normal level order traversal, but the order of visiting nodes changes at each level.

Example 1:

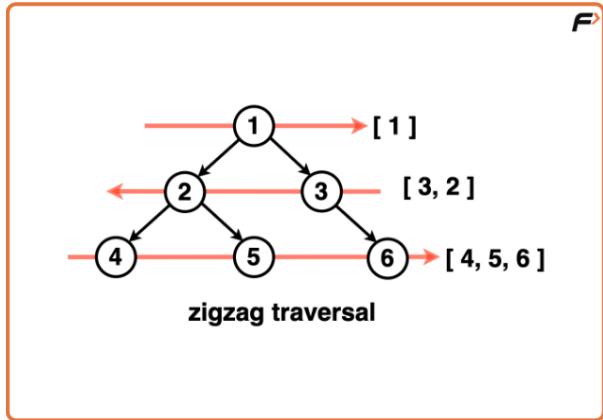
Binary Tree: 1 2 3 4 5 -1 6



Traversal:

- Level 1 → [ 1 ] (left to right)
- Level 2 → [ 3 , 2 ] (right to left)
- Level 3 → [ 4 , 5 , 6 ] (left to right)

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



## Approach 1

### Algorithm

This approach is based on **Level Order Traversal (BFS)** using a queue, with an extra flag to control direction.

Steps:

- If the root is NULL, return an empty result.
- Create a queue and push the root.
- Maintain a boolean flag `leftToRight`, initially set to true.
- While the queue is not empty:
  - Find the number of nodes at the current level.
  - Create a vector `level` of size equal to the number of nodes.
  - For each node at the current level:
    - Pop the node from the queue.
    - Decide the index in `level`:
    - If `leftToRight` is true, `index = i`.

- Else,  $\text{index} = \text{size} - 1 - i$ .
    - Store the node value at the calculated index.
    - Push left and right children into the queue if they exist.
  - Add `level` to the answer.
  - Flip `leftToRight` for the next level.
  - Return the final result.
- 

## Code

```
#include <bits/stdc++.h>
using namespace std;

// Node structure for the binary tree
struct TreeNode {
 int val;
 TreeNode* left;
 TreeNode* right;
 TreeNode(int x) : val(x), left(NULL), right(NULL) {}
};

class Solution {
public:
 vector<vector<int>> zigzagLevelOrder(TreeNode* root) {
 vector<vector<int>> ans;
 if (root == NULL) return ans;

 queue<TreeNode*> q;
 q.push(root);

 bool leftToRight = true;

 while (!q.empty()) {
 int size = q.size();
 vector<int> temp;
 for (int i = 0; i < size; i++) {
 TreeNode* curr = q.front();
 q.pop();
 if (leftToRight) {
 temp.push_back(curr->val);
 } else {
 temp.push_back(curr->val);
 }
 if (curr->left != NULL) q.push(curr->left);
 if (curr->right != NULL) q.push(curr->right);
 }
 ans.push_back(temp);
 leftToRight = !leftToRight;
 }
 return ans;
 }
};
```

```

vector<int> level(size);

for (int i = 0; i < size; i++) {
 TreeNode* node = q.front();
 q.pop();

 int index = leftToRight ? i : size - 1 - i;
 level[index] = node->val;

 if (node->left) q.push(node->left);
 if (node->right) q.push(node->right);
}

ans.push_back(level);
leftToRight = !leftToRight;
}

return ans;
}

};

int main() {
 TreeNode* root = new TreeNode(1);
 root->left = new TreeNode(2);
 root->right = new TreeNode(3);
 root->left->left = new TreeNode(4);
 root->left->right = new TreeNode(5);
 root->right->right = new TreeNode(6);

 Solution sol;
 vector<vector<int>> result = sol.zigzagLevelOrder(root);

 for (auto level : result) {
 cout << "[";
 for (int i = 0; i < level.size(); i++) {
 cout << level[i];
 if (i != level.size() - 1) cout << ", ";
 }
 }
}

```

```
 cout << "] ";
}
return 0;
}
```

---

## Complexity Analysis

### Time Complexity: O(N)

Each node is visited exactly once during the level order traversal.

### Space Complexity: O(N)

Queue can store up to O(N) nodes in the worst case, and the result vector also stores all nodes.

# 15. Boundary Traversal of a Binary Tree

### Problem Statement

Given a Binary Tree, perform the **boundary traversal** of the tree in **anti-clockwise direction**, starting from the root.

Boundary traversal includes:

1. Root node
  2. Left boundary (excluding leaf nodes)
  3. All leaf nodes (from left to right)
  4. Right boundary (excluding leaf nodes, added in reverse order)
- 

### Question Explanation

Boundary traversal means visiting only the **outer boundary nodes** of the tree.

Order of traversal (anti-clockwise):

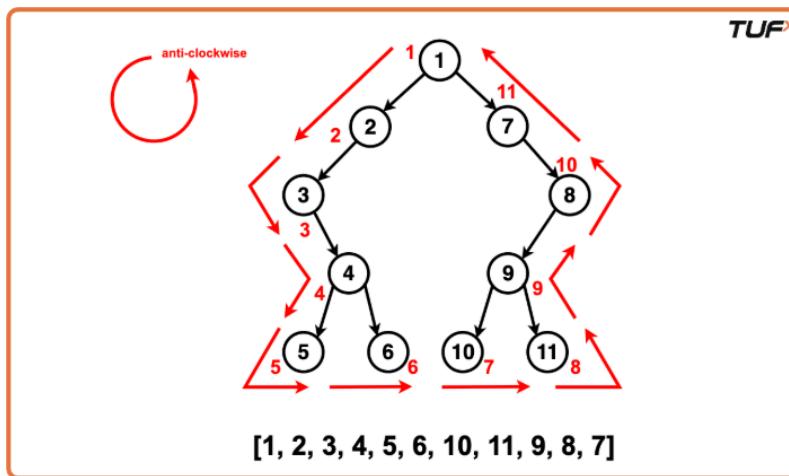
- Start with the root.
- Traverse the left boundary top to bottom (excluding leaves).
- Traverse all leaf nodes from left to right.
- Traverse the right boundary bottom to top (excluding leaves).

Example 1

Binary Tree: 1 2 7 3 -1 -1 8 -1 4 9 -1 5 6 10 11

Boundary Traversal:

[1, 2, 3, 4, 5, 6, 10, 11, 9, 8, 7]



## Approach

### Algorithm

We break the boundary traversal into **three clear parts**.

#### 1. Left Boundary

- Start from root->left.

- Keep moving to left child.
- If left child does not exist, move to right child.
- Add only non-leaf nodes.

## 2. Leaf Nodes

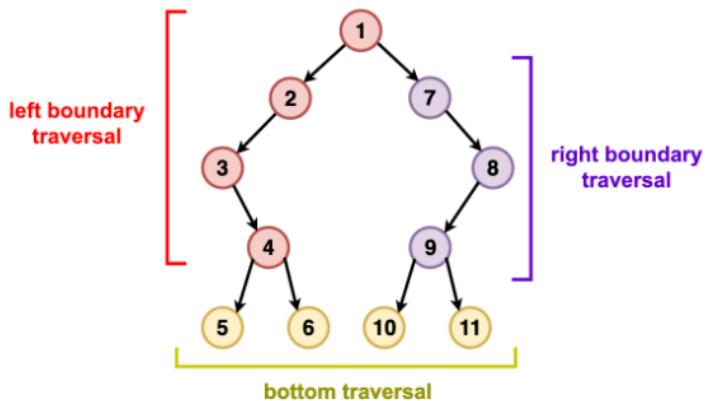
- Traverse the whole tree using recursion.
- Whenever a node is a leaf, add it to the result.

## 3. Right Boundary

- Start from root->right.
- Keep moving to right child.
- If right child does not exist, move to left child.
- Store non-leaf nodes in a temporary array.
- Add them to result in reverse order.

Important:

- Leaf nodes must **not be repeated** in left or right boundary.
- Root should be added only once.



## Code

```

#include <bits/stdc++.h>
using namespace std;

// Node structure
struct Node {
 int data;
 Node* left;
 Node* right;
 Node(int val) : data(val), left(NULL), right(NULL) {}
};

class Solution {
public:
 bool isLeaf(Node* node) {
 return node->left == NULL && node->right == NULL;
 }

 void addLeftBoundary(Node* root, vector<int>& res) {
 Node* curr = root->left;
 while (curr) {
 if (!isLeaf(curr))
 res.push_back(curr->data);
 curr = curr->left;
 }
 }

 void addRightBoundary(Node* root, vector<int>& res) {
 Node* curr = root->right;
 while (curr) {
 if (!isLeaf(curr))
 res.push_back(curr->data);
 curr = curr->right;
 }
 }

 void addBottomBoundary(Node* root, vector<int>& res) {
 Node* curr = root->left;
 while (curr) {
 if (curr->right)
 curr = curr->right;
 else
 curr = curr->right;
 }
 curr = root->right;
 while (curr) {
 if (curr->left)
 curr = curr->left;
 else
 curr = curr->left;
 }
 }

 vector<int> boundaryOfBinaryTree(Node* root) {
 vector<int> res;
 if (root)
 addLeftBoundary(root, res);
 addBottomBoundary(root, res);
 addRightBoundary(root, res);
 return res;
 }
};

```

```

 if (curr->left)
 curr = curr->left;
 else
 curr = curr->right;
 }
}

void addLeaves(Node* root, vector<int>& res) {
 if (root == NULL) return;

 if (isLeaf(root)) {
 res.push_back(root->data);
 return;
 }
 addLeaves(root->left, res);
 addLeaves(root->right, res);
}

void addRightBoundary(Node* root, vector<int>& res) {
 Node* curr = root->right;
 vector<int> temp;
 while (curr) {
 if (!isLeaf(curr))
 temp.push_back(curr->data);
 if (curr->right)
 curr = curr->right;
 else
 curr = curr->left;
 }
 for (int i = temp.size() - 1; i >= 0; i--)
 res.push_back(temp[i]);
}

vector<int> boundaryTraversal(Node* root) {
 vector<int> res;
 if (root == NULL) return res;

 if (!isLeaf(root))

```

```

 res.push_back(root->data);

 addLeftBoundary(root, res);
 addLeaves(root, res);
 addRightBoundary(root, res);

 return res;
 }
};

int main() {
 Node* root = new Node(1);
 root->left = new Node(2);
 root->right = new Node(3);
 root->left->left = new Node(4);
 root->left->right = new Node(5);
 root->right->left = new Node(6);
 root->right->right = new Node(7);

 Solution sol;
 vector<int> ans = sol.boundaryTraversal(root);

 for (int x : ans)
 cout << x << " ";
 return 0;
}

```

---

## Complexity Analysis

### Time Complexity: O(N)

Every node is visited at most once while finding boundaries and leaves.

### Space Complexity: O(N)

Result vector stores boundary nodes.

Recursive stack space can go up to O(H), where H is height of tree.

# 16. Vertical Order Traversal of Binary Tree

## Problem Statement

Given a Binary Tree, return its **Vertical Order Traversal** from the **leftmost vertical level to the rightmost vertical level**.

If multiple nodes lie on the same vertical line, they must be printed in **level order** (top to bottom, left to right).

---

## Question Explanation

In vertical order traversal, we imagine vertical lines passing through the tree.

- Root lies at vertical level 0
- Left child moves to vertical -1
- Right child moves to vertical +1
- Nodes having the same vertical value belong to the same column

If multiple nodes fall in the same vertical column, they must appear in **level order traversal order**.

### Example 1

Binary Tree: 1 2 3 4 10 9 11 -1 5 -1 -1 -1 -1 -1 -1 6

Output:

`[[4], [2, 5], [1, 10, 9, 6], [3], [11]]`

Explanation:

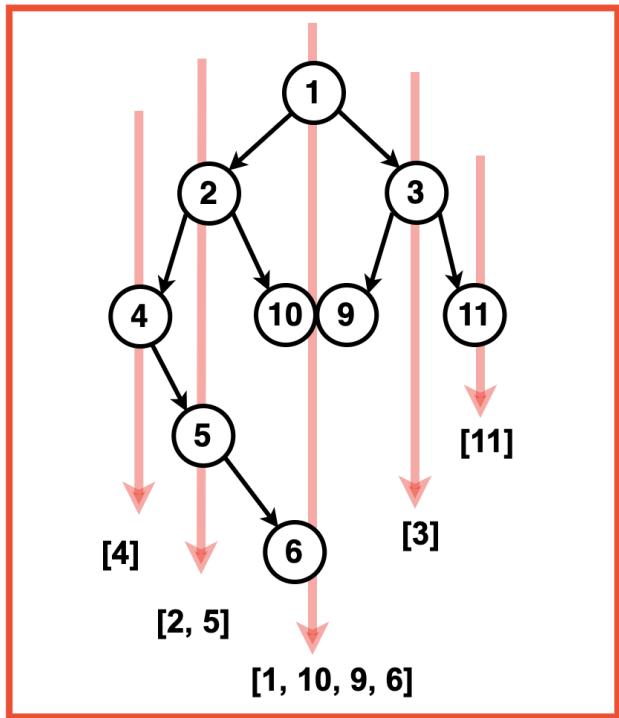
Vertical -2 → [4]

Vertical -1 → [2, 5]

Vertical 0 → [1, 10, 9, 6]

Vertical 1 → [3]

Vertical 2 → [11]



## Approach

### Algorithm

We use **Level Order Traversal (BFS)** and track two coordinates for every node:

- $x \rightarrow$  vertical level
- $y \rightarrow$  tree level (depth)

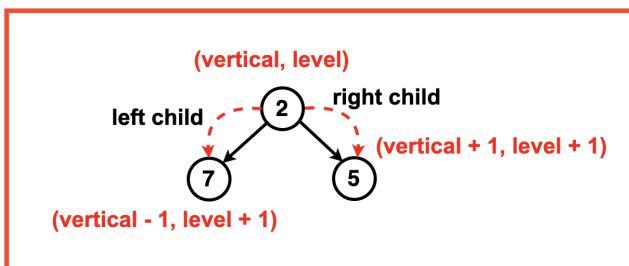
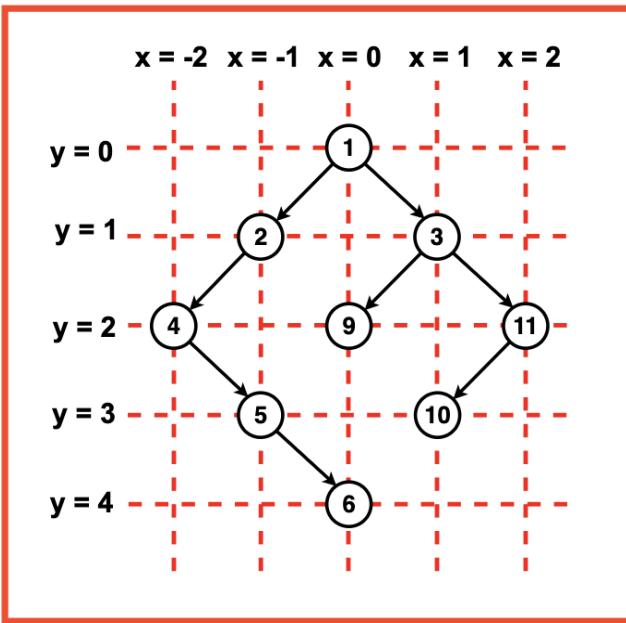
Steps:

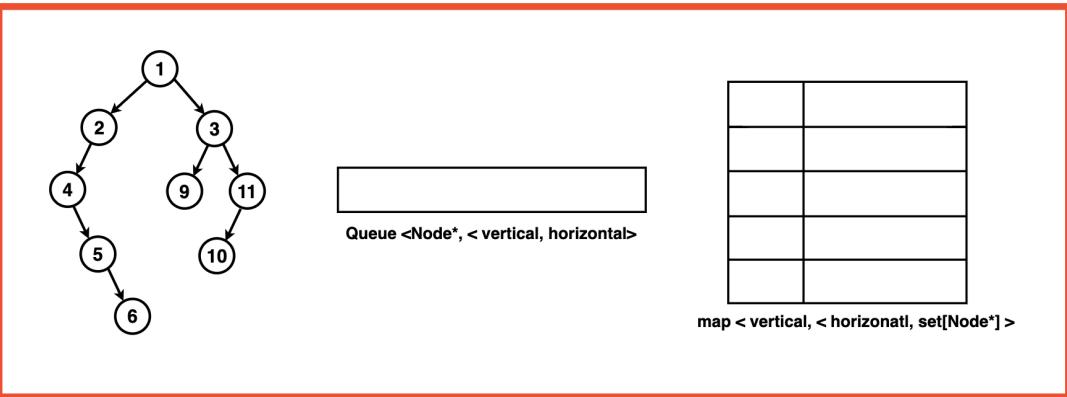
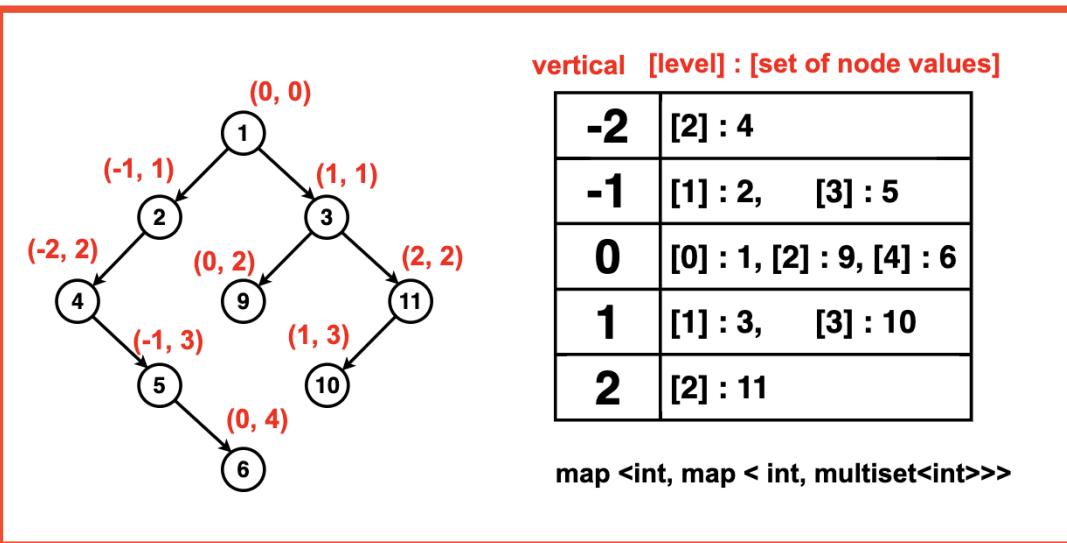
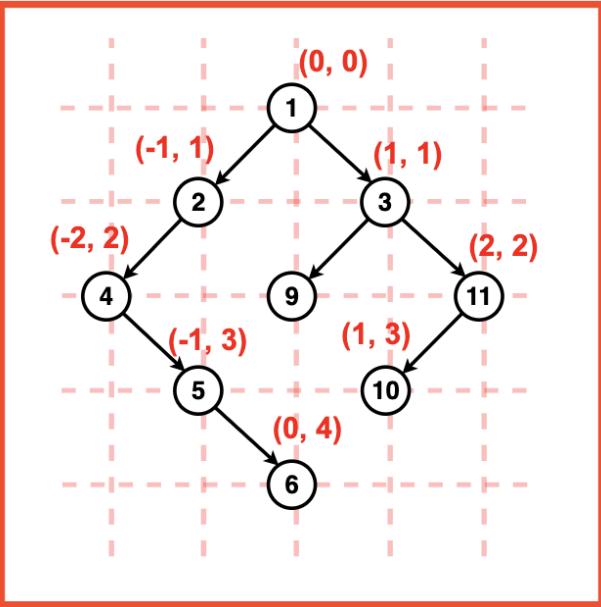
1. Use a map to store nodes based on vertical and level  
 $\text{map<}int, \text{ map<}int, \text{ multiset<}int>>$
2. Use a queue for BFS storing (node,  $x$ ,  $y$ )
3. Start with root at ( $0, 0$ )
4. For every node:

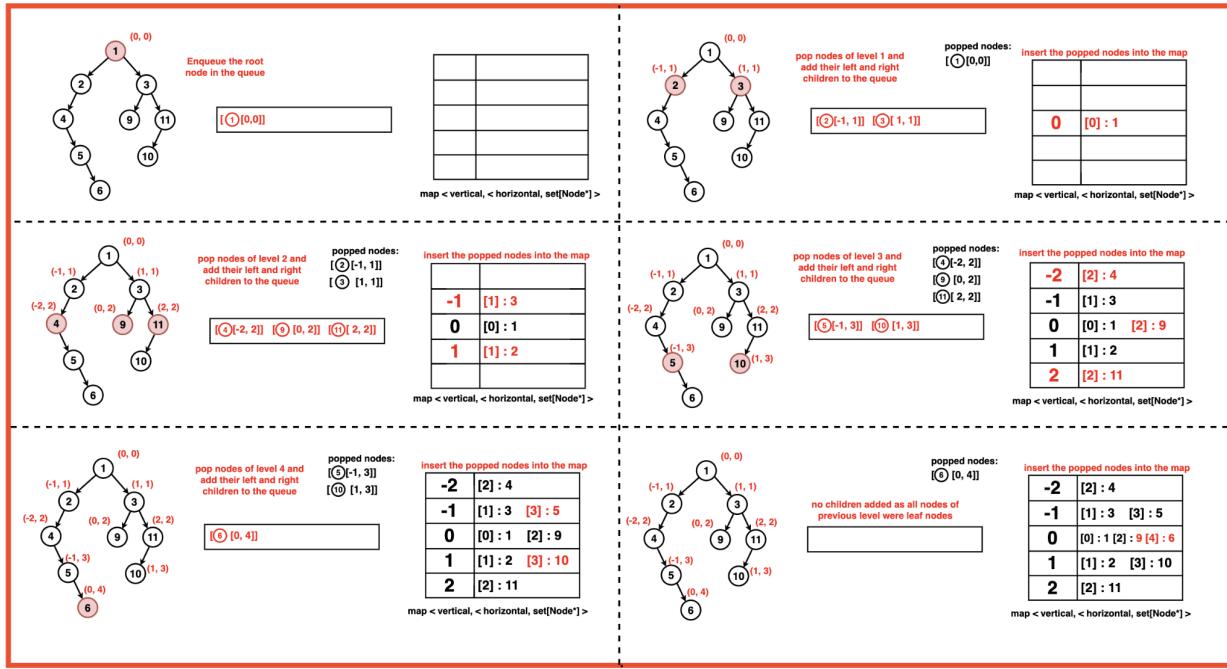
- Insert node value into map at  $(x, y)$
  - Left child  $\rightarrow (x-1, y+1)$
  - Right child  $\rightarrow (x+1, y+1)$
5. After BFS, traverse the map from leftmost vertical to rightmost
6. Collect values level by level into result

This guarantees:

- Left to right vertical order
- Top to bottom level order
- Correct order for overlapping nodes







## Code

```
#include <bits/stdc++.h>
using namespace std;

// Binary Tree Node
struct Node {
 int data;
 Node* left;
 Node* right;
 Node(int val) : data(val), left(NULL), right(NULL) {}
};

class Solution {
public:
 vector<vector<int>> verticalTraversal(Node* root) {
 // vertical -> level -> nodes
 map<int, map<int, multiset<int>> nodes;
 queue<pair<Node*, pair<int, int>> q;

 // root at vertical 0, level 0
 q.push({root, {0, 0}});

```

```

while (!q.empty()) {
 auto temp = q.front();
 q.pop();

 Node* node = temp.first;
 int x = temp.second.first;
 int y = temp.second.second;

 nodes[x][y].insert(node->data);

 if (node->left)
 q.push({node->left, {x - 1, y + 1}});
 if (node->right)
 q.push({node->right, {x + 1, y + 1}});
}

vector<vector<int>> ans;

for (auto vertical : nodes) {
 vector<int> col;
 for (auto level : vertical.second) {
 col.insert(col.end(),
 level.second.begin(),
 level.second.end());
 }
 ans.push_back(col);
}
return ans;
};

int main() {
 Node* root = new Node(1);
 root->left = new Node(2);
 root->right = new Node(3);
 root->left->left = new Node(4);
 root->left->right = new Node(10);
}

```

```

root->right->left = new Node(9);
root->right->right = new Node(11);
root->left->left->right = new Node(5);
root->left->left->right->right = new Node(6);

Solution sol;
vector<vector<int>> res = sol.verticalTraversal(root);

for (auto v : res) {
 cout << "[";
 for (int x : v) cout << x << " ";
 cout << "]";
}
return 0;
}

```

---

### **multiset<int>**

- Values store karta hai
- Automatically sorted
- Duplicate allowed

👉 Ye isliye use hota hai kyunki:

- Same row & column me multiple nodes ho sakte hain
- Unsorted order me output karna hota hai

```

col.insert(col.end()),

level.second.begin(),

level.second.end());

level.second

```

multiset<int>

👉 Isme: same column & same row ke saare node values

already sorted order me hote hain

col.insert(...) ka matlab

col.insert(

    col.end(),         // end me insert karo

    begin iterator,    // kaha se

    end iterator      // kaha tak

);

👉 Matlab:

is row ke saare values ko col vector ke end me daal do

## Complexity Analysis

**Time Complexity:**  $O(N \log N)$

Each node is visited once. Insertions into map and multiset take logarithmic time.

**Space Complexity:**  $O(N)$

Map stores all nodes and queue can hold up to  $N/2$  nodes in worst case.

# 17. Top View of a Binary Tree

## Problem Statement

Given a Binary Tree, return its **Top View**.

The Top View of a Binary Tree contains the nodes that are visible when the tree is viewed from the top.

---

## Question Explanation

To understand the top view, imagine looking at the binary tree from above.

- Nodes are arranged on **vertical lines**
- Each node has a **vertical index**
  - Root → vertical = 0
  - Left child → vertical - 1
  - Right child → vertical + 1
- For every vertical line, **only the topmost node** (the one that appears first when moving level by level) will be visible

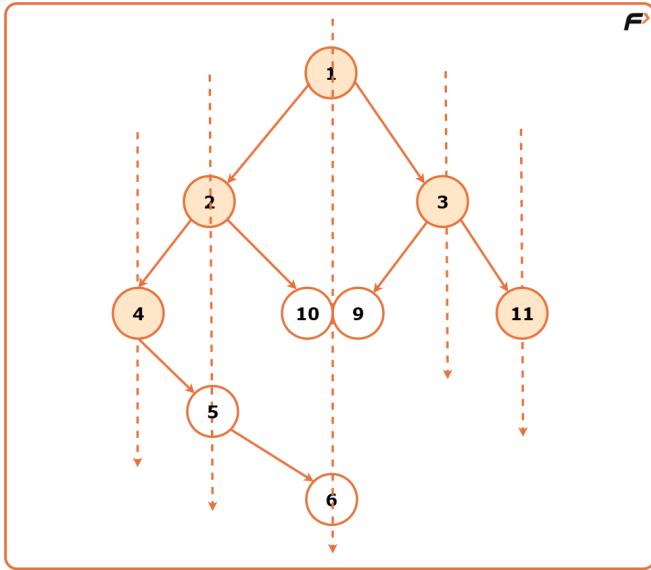
## Example 1

Binary Tree: 1 2 3 4 10 9 11 -1 5 -1 -1 -1 -1 -1 -1 6

Top View Output: [4, 2, 1, 3, 11]

Explanation:

- Vertical -2 → 4
- Vertical -1 → 2
- Vertical 0 → 1
- Vertical +1 → 3
- Vertical +2 → 11



## Approach

We use **Level Order Traversal (BFS)** because it ensures that nodes closer to the root are processed first.

1. If the tree is empty, return an empty array.
2. Create a map `mpp` where:
  - o key → vertical index
  - o value → first node seen at that vertical
3. Use a queue for BFS storing (`node, verticalIndex`).
4. Start by pushing the root with vertical index 0.
5. While the queue is not empty:
  - o Pop the front node.
  - o If its vertical index is **not present** in the map, store the node value.
  - o Push left child with vertical - 1.

- Push right child with vertical + 1.
6. Traverse the map from leftmost to rightmost vertical and store values in the answer array.
  7. Return the answer.
- 

### Code

```
#include <bits/stdc++.h>
using namespace std;

// Node structure for Binary Tree
struct Node {
 int data;
 Node* left;
 Node* right;
 Node(int val) : data(val), left(NULL), right(NULL) {}
};

class Solution {
public:
 vector<int> topView(Node* root) {
 vector<int> ans;
 if (root == NULL) return ans;

 // Map to store first node at each vertical
 map<int, int> mpp;

 // Queue for BFS {node, vertical}
 queue<pair<Node*, int>> q;
 q.push({root, 0});

 while (!q.empty()) {
 auto it = q.front();
 q.pop();

 Node* node = it.first;
 int vertical = it.second;

 if (mpp.find(vertical) == mpp.end())
 mpp[vertical] = node->data;
 else
 mpp[vertical] = max(mpp[vertical], node->data);

 if (node->left)
 q.push({node->left, vertical - 1});
 if (node->right)
 q.push({node->right, vertical + 1});
 }
 for (auto i : mpp)
 ans.push_back(i.second);
 return ans;
 }
};
```

```

 int vertical = it.second;

 // Store only first node of each vertical
 if (mpp.find(vertical) == mpp.end()) {
 mpp[vertical] = node->data;
 }

 if (node->left)
 q.push({node->left, vertical - 1});
 if (node->right)
 q.push({node->right, vertical + 1});
 }

 // Collect result from map (already sorted)
 for (auto it : mpp) {
 ans.push_back(it.second);
 }

 return ans;
}

};

// Driver code
int main() {
 Node* root = new Node(1);
 root->left = new Node(2);
 root->right = new Node(3);
 root->left->left = new Node(4);
 root->left->right = new Node(10);
 root->right->left = new Node(9);
 root->right->right = new Node(11);

 Solution sol;
 vector<int> result = sol.topView(root);

 cout << "Top View: ";
 for (int x : result) cout << x << " ";
 return 0;
}

```

}

---

## Complexity Analysis

**Time Complexity:**  $O(N)$

Each node is visited exactly once during BFS.

**Space Complexity:**  $O(N)$

Queue and map together can store up to  $N$  nodes in the worst case.

# 18. Bottom View of a Binary Tree

Given a Binary Tree, return its **Bottom View**.

The Bottom View of a Binary Tree consists of the nodes that are visible when the tree is viewed from the bottom.

---

## Question Explanation

To find the bottom view, imagine drawing **vertical lines** through the binary tree.

- Each node lies on a vertical line (called vertical index).
- Root node has vertical index 0.
- Left child → vertical index -1.
- Right child → vertical index +1.
- For each vertical index, **the node that appears last in level order traversal** (i.e., the lowest node) will be visible from the bottom.

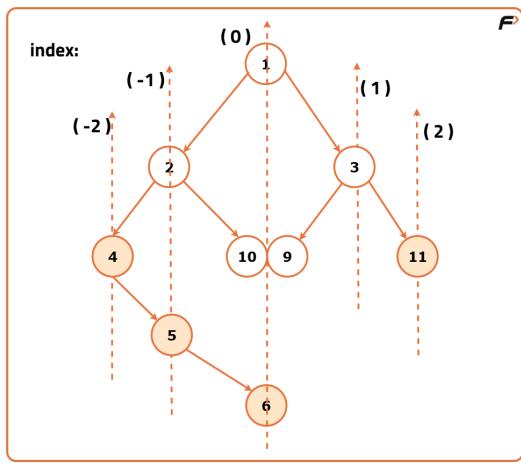
### Example 1

Binary Tree: 1 2 3 4 10 9 11 -1 5 -1 -1 -1 -1 -1 -1 6

Bottom View Output: [4, 5, 6, 3, 11]

Explanation:

- Vertical -2 → 4
- Vertical -1 → 5
- Vertical 0 → 6
- Vertical +1 → 3
- Vertical +2 → 11



## Approach 1 (Optimal Approach)

### Algorithm

1. If the tree is empty, return an empty array.
2. Create a map where:
  - key → vertical index
  - value → node value at that vertical (updated continuously)
3. Use **level order traversal (BFS)** with a queue.
4. Push the root into the queue with vertical index 0.
5. While the queue is not empty:
  - Pop a node and its vertical index.

- Update the map for this vertical index with the current node's value.  
(This ensures the bottom-most node overwrites previous ones.)
  - Push left child with vertical - 1.
  - Push right child with vertical + 1.
6. Traverse the map from leftmost to rightmost vertical index and store values in the result.
7. Return the result vector.
- 

### Code

```
#include <bits/stdc++.h>
using namespace std;

// Node structure for the binary tree
struct Node {
 int data;
 Node* left;
 Node* right;
 Node(int val) : data(val), left(NULL), right(NULL) {}
};

class Solution {
public:
 vector<int> bottomView(Node* root) {
 vector<int> ans;
 if (root == NULL) return ans;

 // Map to store vertical -> bottom node value
 map<int, int> mpp;

 // Queue for BFS: {node, vertical index}
 queue<pair<Node*, int>> q;
 q.push({root, 0});

 while (!q.empty()) {


```

```

 auto it = q.front();
 q.pop();

 Node* node = it.first;
 int vertical = it.second;

 // Always overwrite to keep bottom-most node
 mpp[vertical] = node->data;

 if (node->left)
 q.push({node->left, vertical - 1});
 if (node->right)
 q.push({node->right, vertical + 1});
 }

 // Extract result from map
 for (auto it : mpp) {
 ans.push_back(it.second);
 }

 return ans;
}

};

// Driver code
int main() {
 Node* root = new Node(1);
 root->left = new Node(2);
 root->left->left = new Node(4);
 root->left->right = new Node(10);
 root->left->left->right = new Node(5);
 root->left->left->right->right = new Node(6);
 root->right = new Node(3);
 root->right->left = new Node(9);
 root->right->right = new Node(11);

 Solution sol;
 vector<int> res = sol.bottomView(root);
}

```

```
cout << "Bottom View: ";
for (int x : res) cout << x << " ";
return 0;
}
```

---

## Complexity Analysis

**Time Complexity:**  $O(N)$

Each node is visited exactly once during BFS traversal.

**Space Complexity:**  $O(N)$

The queue and map together can store up to  $N$  nodes in the worst case.

# 19. Left View / Right View of a Binary Tree

---

## Problem Recap

You are standing on **one side of a binary tree**:

- **Left View** → nodes visible from the **left side**
- **Right View** → nodes visible from the **right side**

Return the visible nodes **from top to bottom**.

---

At each level, only one node is visible:

- **Left View** → the **first node** encountered at that level
- **Right View** → the **first node** encountered at that level **when traversing right-first**

So the problem reduces to:

"At each depth, pick the first node seen from the desired direction."

---

## ◆ Optimal Approach

We use **Depth First Search** with level tracking.

### Why DFS works?

- We track the **current level**
  - If this level is visited for the **first time**, we store the node
  - Order of recursion decides **left or right view**
- 

## Left View (DFS)

### Traversal Order

Root → Left → Right

### Logic

- If `res.size() == level` → first node of this level → add it

### Code (Left View)

```
#include <bits/stdc++.h>
using namespace std;
```

```

struct Node {
 int data;
 Node* left;
 Node* right;
 Node(int val) : data(val), left(NULL), right(NULL) {}
};

class Solution {
public:
 void dfsLeft(Node* root, int level, vector<int>& res) {
 if (!root) return;

 // First node at this level
 if (res.size() == level)
 res.push_back(root->data);

 dfsLeft(root->left, level + 1, res);
 dfsLeft(root->right, level + 1, res);
 }

 vector<int> leftView(Node* root) {
 vector<int> res;
 dfsLeft(root, 0, res);
 return res;
 }
};

```

---

## Right View (DFS)

### Traversal Order

Root → Right → Left

### Logic

- Same idea, but traverse **right first**

## Code (Right View)

```
class Solution {
public:
 void dfsRight(Node* root, int level, vector<int>& res) {
 if (!root) return;

 if (res.size() == level)
 res.push_back(root->data);

 dfsRight(root->right, level + 1, res);
 dfsRight(root->left, level + 1, res);
 }

 vector<int> rightView(Node* root) {
 vector<int> res;
 dfsRight(root, 0, res);
 return res;
 }
};
```

---

## ◆ BFS (Level Order) Approach (Also Valid)

### Idea

- Perform level order traversal
- For each level:
  - **Left View** → pick first node
  - **Right View** → pick last node

## BFS Code (Both Views)

```
vector<int> leftView(Node* root) {
 vector<int> res;
 if (!root) return res;

 queue<Node*> q;
 q.push(root);
```

```

while (!q.empty()) {
 int size = q.size();
 for (int i = 0; i < size; i++) {
 Node* node = q.front(); q.pop();

 if (i == 0) res.push_back(node->data); // leftmost

 if (node->left) q.push(node->left);
 if (node->right) q.push(node->right);
 }
}
return res;
}

vector<int> rightView(Node* root) {
 vector<int> res;
 if (!root) return res;

 queue<Node*> q;
 q.push(root);

 while (!q.empty()) {
 int size = q.size();
 for (int i = 0; i < size; i++) {
 Node* node = q.front(); q.pop();

 if (i == size - 1) res.push_back(node->data); // rightmost

 if (node->left) q.push(node->left);
 if (node->right) q.push(node->right);
 }
 }
 return res;
}

```

---



## Complexity Analysis

| Approach | Time | Space |
|----------|------|-------|
|----------|------|-------|

|     |        |        |
|-----|--------|--------|
| DFS | $O(N)$ | $O(H)$ |
|-----|--------|--------|

|     |        |        |
|-----|--------|--------|
| BFS | $O(N)$ | $O(N)$ |
|-----|--------|--------|

- $N \rightarrow$  number of nodes
- $H \rightarrow$  height of tree

## 20. Check for Symmetrical Binary Tree

A Binary Tree is said to be **symmetric** if its left subtree is a mirror image of its right subtree. If we draw a vertical line through the root, the structure and values on both sides should reflect each other exactly.

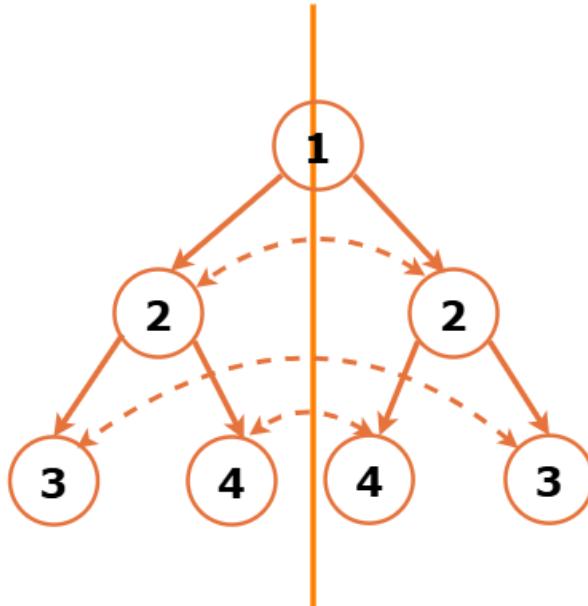
In other words:

- Left child of the left subtree must match the right child of the right subtree.
- Right child of the left subtree must match the left child of the right subtree.
- This condition must hold true for **every node** in the tree.

Example explanation:

For the tree 1 2 2 3 4 4 3

The left subtree [2, 3, 4] is a mirror of the right subtree [2, 4, 3], so the tree is symmetric.



## Approach

We solve this problem using **recursion** by comparing two subtrees in a mirrored manner.

Steps:

1. If the root is NULL, the tree is symmetric.
2. Create a helper function that takes two nodes:
  - o One from the left subtree.
  - o One from the right subtree.
3. Base cases:
  - o If both nodes are NULL, return true.
  - o If only one is NULL, return false.
4. Check:

- Values of both nodes must be equal.
  - Left child of the first node must match the right child of the second node.
  - Right child of the first node must match the left child of the second node.
5. If all checks pass recursively, the tree is symmetric.

This works because we always compare nodes that should mirror each other.

### Code

```
#include <bits/stdc++.h>
using namespace std;

// Node structure for the binary tree
struct Node {
 int data;
 Node* left;
 Node* right;
 Node(int val) : data(val), left(nullptr), right(nullptr) {}
};

class Solution {
private:
 bool isSymmetricUtil(Node* root1, Node* root2) {
 if (root1 == NULL || root2 == NULL)
 return root1 == root2;

 return (root1->data == root2->data)
 && isSymmetricUtil(root1->left, root2->right)
 && isSymmetricUtil(root1->right, root2->left);
 }

public:
 bool isSymmetric(Node* root) {
 if (!root) return true;
 return isSymmetricUtil(root->left, root->right);
 }
};
```

```

int main() {
 Node* root = new Node(1);
 root->left = new Node(2);
 root->right = new Node(2);
 root->left->left = new Node(3);
 root->left->right = new Node(4);
 root->right->left = new Node(4);
 root->right->right = new Node(3);

 Solution sol;
 cout << sol.isSymmetric(root);
 return 0;
}

```

## Complexity Analysis

Time Complexity: **O(N)**

Each node is visited once while comparing mirrored nodes.

Space Complexity: **O(H)**

Extra space is used by the recursion stack, where H is the height of the tree.

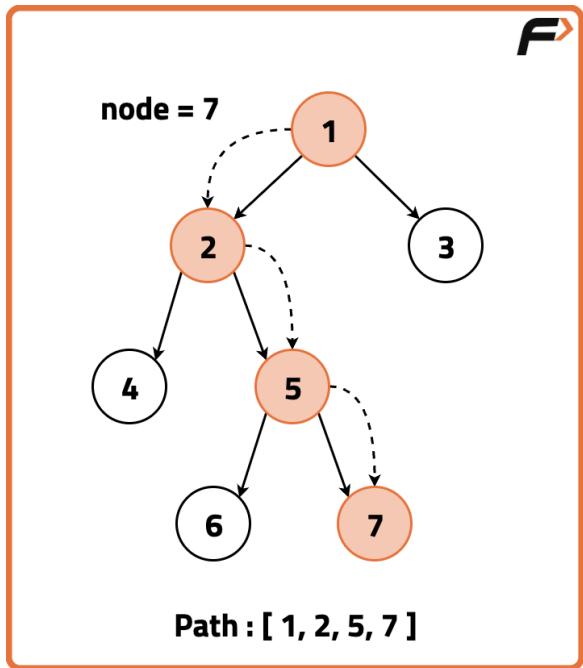
In the worst case (skewed tree), H = N.

# 21. Print Root to Node Path in a Binary Tree

You are given:

- The **root** of a Binary Tree
- A **target node value** (unique and guaranteed to exist)

You need to **return the path from the root to that target node**.



## Key Idea (DFS + Backtracking)

We perform a **Depth First Search (DFS)** and maintain a vector that stores the current path from the root.

At every node:

- Add the node to the path
- If this node is the target → path found
- Otherwise, search in **left** and **right** subtrees
- If the target is not found in either subtree, **backtrack** (remove the node from path)

This ensures that the path vector always represents the correct root-to-current-node path.

## Algorithm

1. Start DFS from the root.

2. Push current node's value into the path vector.
  3. If current node's value == target, return true.
  4. Recursively search left and right subtrees.
  5. If neither subtree contains the target:
    - o Pop current node from path (backtracking)
    - o Return false
  6. Final path vector contains the root → target path.
- 

## C++ Code

```
#include <bits/stdc++.h>
using namespace std;

// Structure for a binary tree node
struct TreeNode {
 int val;
 TreeNode* left;
 TreeNode* right;

 TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

class Solution {
public:
 // Helper DFS function
 bool getPath(TreeNode* root, vector<int>& path, int target) {
 if (root == NULL) return false;

 // Add current node
 path.push_back(root->val);

 // If target found

```

```

 if (root->val == target)
 return true;

 // Search left or right subtree
 if (getPath(root->left, path, target) ||
 getPath(root->right, path, target))
 return true;

 // Backtrack if target not found
 path.pop_back();
 return false;
 }

 // Main function
 vector<int> rootToNodePath(TreeNode* root, int target) {
 vector<int> path;
 getPath(root, path, target);
 return path;
 }
};

int main() {
 // Constructing the tree
 TreeNode* root = new TreeNode(1);
 root->left = new TreeNode(2);
 root->right = new TreeNode(3);
 root->left->left = new TreeNode(4);
 root->left->right = new TreeNode(5);
 root->left->right->right = new TreeNode(7);

 Solution sol;
 int target = 7;

 vector<int> path = sol.rootToNodePath(root, target);

 cout << "Path from root to node " << target << ":" ;
 for (int x : path) cout << x << " ";
 return 0;
}

```

}

---

## Example Walkthrough

For target 7, traversal path becomes:

1 → 2 → 5 → 7

Backtracking ensures that wrong branches are discarded automatically.

---

## Complexity Analysis

- **Time Complexity:**  $O(N)$   
Every node is visited once in the worst case.
- **Space Complexity:**  $O(N)$   
Due to recursion stack + path vector.

# 22. Lowest Common Ancestor (LCA) of a Binary Tree

Given the **root** of a binary tree and **two nodes p and q**, find their **Lowest Common Ancestor (LCA)**.

LCA is the **lowest (deepest) node** in the tree that has **both p and q as descendants**.

A node can be a descendant of itself.

---

## Key Insight (Recursive DFS)

At every node, there are **three possibilities**:

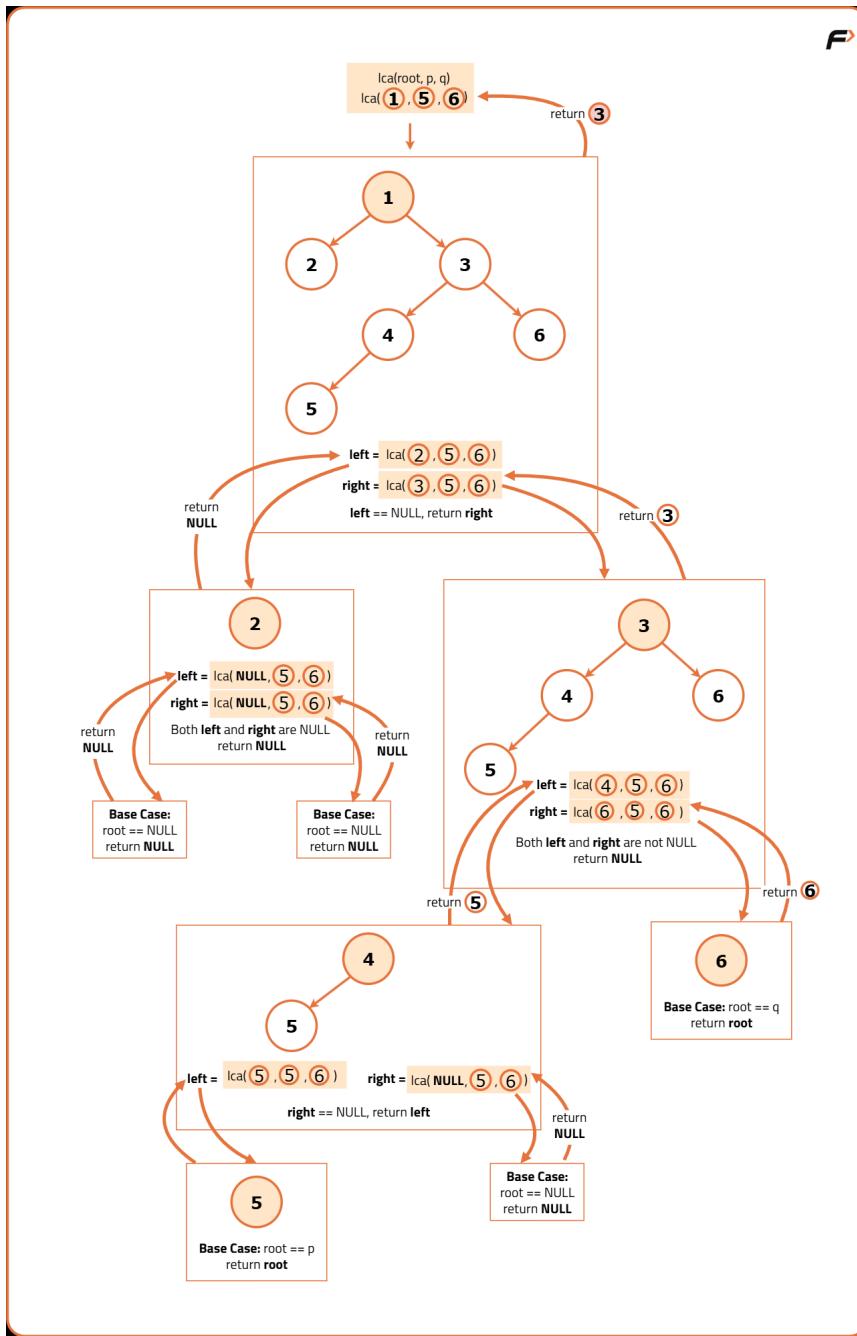
1. p and q lie in **different subtrees** → current node is LCA
2. Both lie in **the left subtree**
3. Both lie in **the right subtree**

We solve this using **postorder traversal**.

---

## Algorithm (Optimal & Clean)

1. **Base Case**
  - If `root == NULL`, return `NULL`
  - If `root == p` or `root == q`, return `root`
2. Recursively find LCA in:
  - Left subtree
  - Right subtree
3. **Decision**
  - If both left and right are non-null → current node is LCA
  - Else return the non-null child



## C++ Code (Standard Interview Version)

```
#include <bits/stdc++.h>
using namespace std;
```

```
struct TreeNode {
 int data;
```

```

TreeNode* left;
TreeNode* right;

TreeNode(int val) : data(val), left(NULL), right(NULL) {}

};

class Solution {
public:
 TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p,
TreeNode* q) {
 // Base case
 if (root == NULL || root == p || root == q) {
 return root;
 }

 // Search left and right subtrees
 TreeNode* left = lowestCommonAncestor(root->left, p, q);
 TreeNode* right = lowestCommonAncestor(root->right, p, q);

 // If p and q are found in different subtrees
 if (left && right) {
 return root;
 }

 // Otherwise return non-null child
 return left ? left : right;
 }
};

// Driver Code
int main() {
 // Construct the tree
 TreeNode* root = new TreeNode(3);
 root->left = new TreeNode(5);
 root->right = new TreeNode(1);
 root->left->left = new TreeNode(6);
 root->left->right = new TreeNode(2);
 root->right->left = new TreeNode(0);
}

```

```

root->right->right = new TreeNode(8);

Solution sol;
TreeNode* p = root->left; // Node 5
TreeNode* q = root->right; // Node 1

TreeNode* lca = sol.lowestCommonAncestor(root, p, q);
cout << "Lowest Common Ancestor: " << lca->data << endl;

return 0;
}

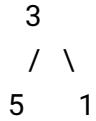
```

---

## Example Walkthrough

### Example 1

p = 5, q = 1

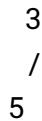


- 5 in left subtree
  - 1 in right subtree
- LCA = 3**

---

### Example 2

p = 5, q = 4



\  
4

- One node is ancestor of the other

LCA = 5

## 23. Maximum Width of a Binary Tree

Given a Binary Tree, you need to find its **maximum width**.

The width of a level is defined as the number of positions between the **leftmost** and **rightmost** non-null nodes at that level, **including null positions in between**, assuming the tree is placed like a complete binary tree.

In simple words, for every level, imagine all nodes placed in their correct positions as in a complete binary tree. The width is the distance between the first and last node at that level. The answer is the **maximum such width among all levels**.

### Example Explanation

Example 1:

Binary Tree: 1 2 3 5 6 -1 9

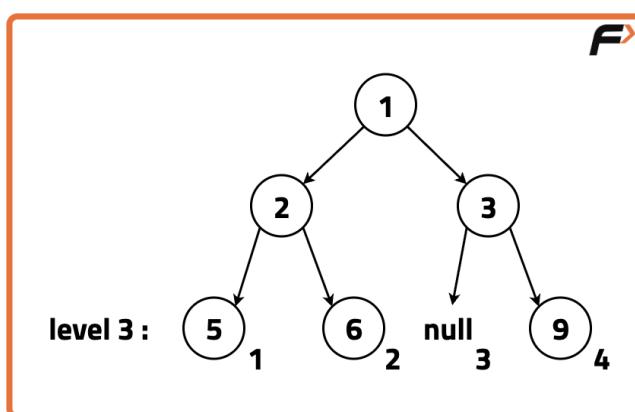
At level 3, the nodes look like:

5 6 null 9

Here, leftmost node is 5 and rightmost node is 9.

Even though there is a null in between, it is counted.

So width = 4, which is the maximum.

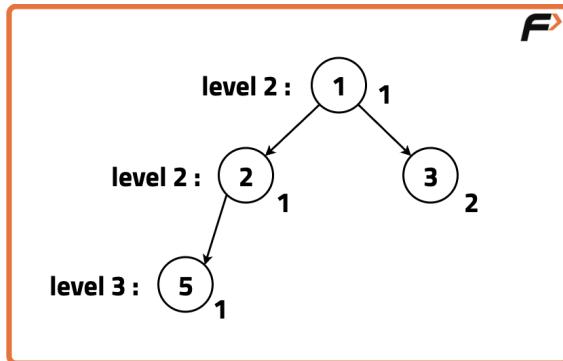


Example 2:

Binary Tree: 1 2 3 5

Level 2 contains nodes {2, 3}

Width = 2, which is the maximum.



## Approach

We use **level order traversal (BFS)** and assign an index to every node as if the tree were a **complete binary tree**.

- Assign index 0 to the root.
- For any node with index  $i$ :
  - Left child gets index  $2*i + 1$
  - Right child gets index  $2*i + 2$
- For each level:
  - Note the index of the **first node** and **last node**
  - Width of that level =  $\text{lastIndex} - \text{firstIndex} + 1$
- Keep updating the maximum width.

To avoid overflow, indices are **normalized** at every level by subtracting the minimum index of that level.

---

## Code (C++)

```
#include <bits/stdc++.h>
using namespace std;

class TreeNode {
public:
 int val;
 TreeNode* left;
 TreeNode* right;
 TreeNode(int x) {
 val = x;
 left = nullptr;
 right = nullptr;
 }
};

class Solution {
public:
 int widthOfBinaryTree(TreeNode* root) {
 if (!root) return 0;

 int maxWidth = 0;
 queue<pair<TreeNode*, int>> q;
 q.push({root, 0});

 while (!q.empty()) {
 int size = q.size();
 int minIndex = q.front().second;
 int first = 0, last = 0;

 for (int i = 0; i < size; i++) {
 TreeNode* node = q.front().first;
 int currIndex = q.front().second - minIndex;
 q.pop();

 if (i == 0) first = currIndex;

 if (node->left)
 q.push({node->left, currIndex + 1});
 if (node->right)
 q.push({node->right, currIndex + 1});
 }

 maxWidth = max(maxWidth, last - first + 1);
 }
 }
};
```

```

 if (i == size - 1) last = currIndex;

 if (node->left)
 q.push({node->left, 2 * currIndex + 1});
 if (node->right)
 q.push({node->right, 2 * currIndex + 2});
 }

 maxWidth = max(maxWidth, last - first + 1);
}
return maxWidth;
}
};

int main() {
 TreeNode* root = new TreeNode(1);
 root->left = new TreeNode(3);
 root->right = new TreeNode(2);
 root->left->left = new TreeNode(5);
 root->left->right = new TreeNode(3);
 root->right->right = new TreeNode(9);

 Solution sol;
 cout << "Maximum width: " << sol.widthOfBinaryTree(root);
 return 0;
}

```

---

## Complexity Analysis

### Time Complexity: O(N)

Each node is visited exactly once during level order traversal.

### Space Complexity: O(N)

The queue can hold up to  $N/2$  nodes in the worst case (last level of the tree).

# 24. Check for Children Sum Property in a Binary Tree

Given a Binary Tree, you need to **convert** the tree so that it follows the **Children Sum Property**.

The Children Sum Property says that for **every node**, the value of the node must be **equal to the sum of values of its left and right children**.

Important rules:

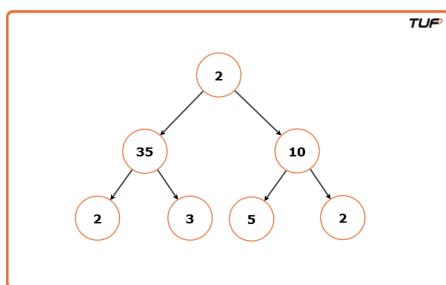
- If a child is missing, its value is considered 0.
- You are **allowed to increase node values**, but **not allowed to decrease** any node value.
- The **structure of the tree must not change**.

---

## Example Explanation

Example 1:

Input Tree values: 2 35 10 2 3 5 2



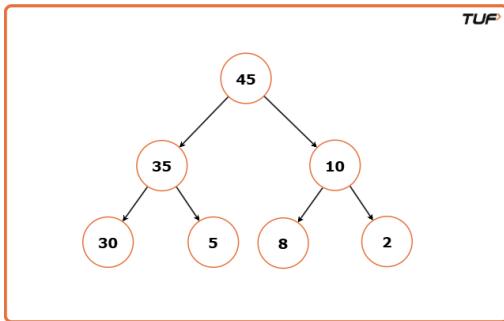
Some parent nodes have values greater than the sum of their children.

Since we cannot decrease parent values, we increase child values so that:

`parent = left child + right child`

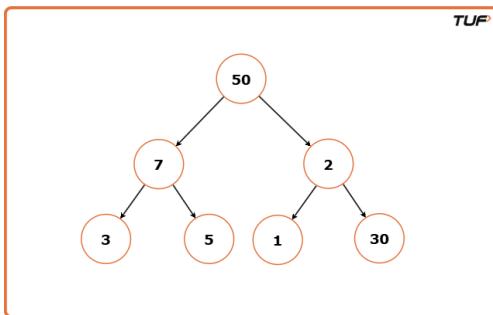
After valid increments, the tree becomes:

45 35 10 30 5 8 2



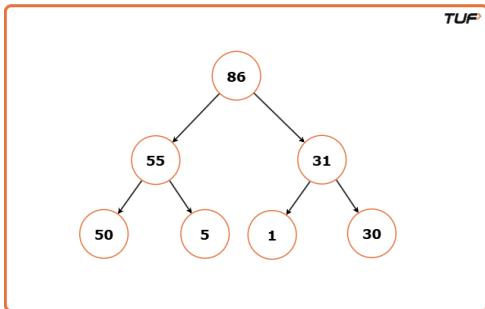
Example 2:

Input Tree values: 50 7 2 3 5 1 30



Here, children sums are less than parent values at some nodes.

So we increase children values recursively until the property is satisfied.



## Approach

- Traverse the tree using recursion.
- For each node:
  - Compute the sum of left and right child values.

- If children sum is **greater than or equal to** node value, update node value.
    - If children sum is **less than** node value, increase one child value to match the parent.
  - Recursively apply the same logic to left and right subtrees.
  - After recursion, update the current node value to the final sum of its children.
  - This ensures:
    - No node value is decreased.
    - Children Sum Property holds for all nodes.
- 

## Code (C++)

```
#include <iostream>
using namespace std;

struct TreeNode {
 int val;
 TreeNode *left;
 TreeNode *right;
 TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

class Solution {
public:
 void changeTree(TreeNode* root) {
 if (root == NULL) {
 return;
 }

 int child = 0;
 if (root->left) {
 child += root->left->val;
 }
 if (root->right) {
```

```

 child += root->right->val;
 }

 if (child >= root->val) {
 root->val = child;
 } else {
 if (root->left) {
 root->left->val = root->val;
 } else if (root->right) {
 root->right->val = root->val;
 }
 }

 changeTree(root->left);
 changeTree(root->right);

 int tot = 0;
 if (root->left) {
 tot += root->left->val;
 }
 if (root->right) {
 tot += root->right->val;
 }

 if (root->left || root->right) {
 root->val = tot;
 }
}

void inorderTraversal(TreeNode* root) {
 if (root == nullptr) return;
 inorderTraversal(root->left);
 cout << root->val << " ";
 inorderTraversal(root->right);
}

int main() {

```

```

TreeNode* root = new TreeNode(3);
root->left = new TreeNode(5);
root->right = new TreeNode(1);
root->left->left = new TreeNode(6);
root->left->right = new TreeNode(2);
root->right->left = new TreeNode(0);
root->right->right = new TreeNode(8);
root->left->right->left = new TreeNode(7);
root->left->right->right = new TreeNode(4);

Solution sol;

cout << "Before: ";
inorderTraversal(root);
cout << endl;

sol.changeTree(root);

cout << "After: ";
inorderTraversal(root);
cout << endl;

return 0;
}

```

---

## Complexity Analysis

### Time Complexity: O(N)

Each node is visited once and processed in constant time.

### Space Complexity: O(N)

Recursive call stack can go up to the height of the tree.

# 25. Print all the Nodes at a Distance of K in a Binary Tree

Given a Binary Tree, a target node, and an integer k, you need to return all nodes whose **distance from the target node is exactly k**.

Distance means the number of edges between two nodes.

Nodes can be above the target (parent side) or below it (children side).

The result can be returned in **any order**.

---

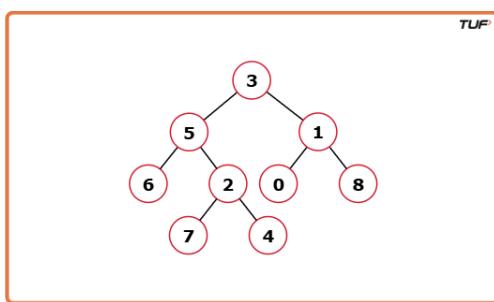
## Example Explanation

Example 1:

Tree: [3, 5, 1, 6, 2, 0, 8, N, N, 7, 4], target = 5, k = 2

From node 5:

- Distance 1 nodes → 6, 2, 3
- Distance 2 nodes → 7, 4, 1

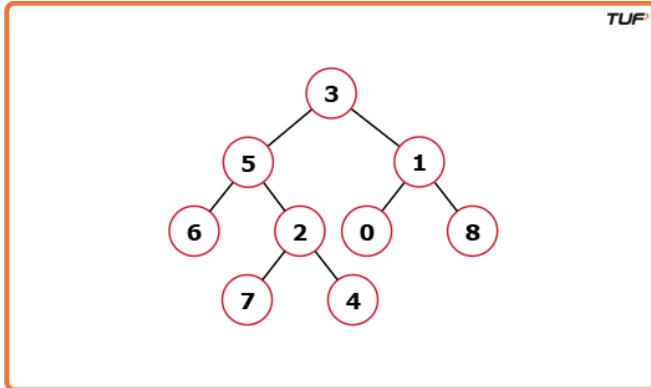


Output: [7, 4, 1]

Example 2:

Same tree, target = 5, k = 3

Distance 3 nodes from 5 → 0, 8



Output: [0, 8]

---

## Approach

- A binary tree only has child pointers, so we cannot move upward directly.
- First, create a **parent mapping** so every node knows its parent.
- This converts the tree into an **undirected graph**.
- Then perform **BFS starting from the target node**.
- Move in three directions from every node:
  - left child
  - right child
  - parent
- Use a visited set to avoid revisiting nodes.
- Stop BFS when distance k is reached.
- All nodes remaining in the queue are at distance k.

Steps:

1. Traverse the tree using BFS and store parent of each node.
  2. Start BFS from the target node.
  3. For each level, increase distance count.
  4. When distance becomes k, collect node values and return.
- 

### Code (C++)

```
#include <bits/stdc++.h>
using namespace std;

struct TreeNode {
 int val;
 TreeNode* left;
 TreeNode* right;
 TreeNode(int x) : val(x), left(NULL), right(NULL) {}
};

class Solution {
public:
 vector<int> distanceK(TreeNode* root, TreeNode* target, int k) {
 if (!root) return {};

 unordered_map<TreeNode*, TreeNode*> parentMap;
 buildParent(root, parentMap);

 queue<TreeNode*> q;
 unordered_set<TreeNode*> visited;

 q.push(target);
 visited.insert(target);

 int level = 0;

 while (!q.empty()) {
 int size = q.size();
```

```

 if (level == k) break;

 level++;

 for (int i = 0; i < size; i++) {
 TreeNode* node = q.front();
 q.pop();

 if (node->left && !visited.count(node->left)) {
 visited.insert(node->left);
 q.push(node->left);
 }

 if (node->right && !visited.count(node->right)) {
 visited.insert(node->right);
 q.push(node->right);
 }

 if (parentMap.count(node) &&
!visited.count(parentMap[node])) {
 visited.insert(parentMap[node]);
 q.push(parentMap[node]);
 }
 }

 vector<int> ans;
 while (!q.empty()) {
 ans.push_back(q.front()->val);
 q.pop();
 }

 return ans;
 }

private:
 void buildParent(TreeNode* root, unordered_map<TreeNode*,
TreeNode*>& parentMap) {

```

```

queue<TreeNode*> q;
q.push(root);

while (!q.empty()) {
 TreeNode* node = q.front();
 q.pop();

 if (node->left) {
 parentMap[node->left] = node;
 q.push(node->left);
 }

 if (node->right) {
 parentMap[node->right] = node;
 q.push(node->right);
 }
}
};

int main() {
 TreeNode* root = new TreeNode(3);
 root->left = new TreeNode(5);
 root->right = new TreeNode(1);
 root->left->left = new TreeNode(6);
 root->left->right = new TreeNode(2);
 root->left->right->left = new TreeNode(7);
 root->left->right->right = new TreeNode(4);
 root->right->left = new TreeNode(0);
 root->right->right = new TreeNode(8);

 Solution sol;
 TreeNode* target = root->left;
 int k = 2;

 vector<int> result = sol.distanceK(root, target, k);

 for (int x : result) cout << x << " ";
}

```

```
 return 0;
}
```

---

## Complexity Analysis

### Time Complexity: O(N)

- One BFS to build parent map.
- One BFS from target node.
- Each node is visited at most once.

### Space Complexity: O(N)

- Parent map stores all nodes.
- Queue and visited set can store up to N nodes in worst case.

# 26. Minimum Time Taken to Burn the Binary Tree from a Node

You are given the root of a Binary Tree and a target node value.

If the target node is set on fire, the fire spreads every second to all directly connected nodes, which are: left child, right child, parent

You have to find the **minimum time required to burn the entire binary tree**.

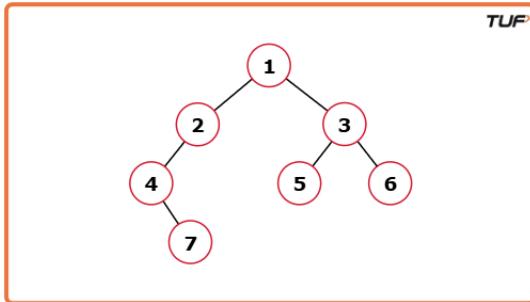
### Example Explanation

Example 1:

Tree: [1, 2, 3, 4, null, 5, 6, null, 7], target = 1

- Second 0: node 1 burns

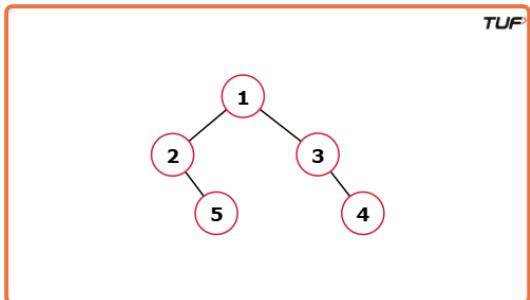
- Second 1: nodes 2 and 3 burn
- Second 2: nodes 4, 5, 6 burn
- Second 3: node 7 burns



Output: 3

Example 2:

Tree: [1, 2, 3, null, 5, null, 4], target = 4



- Second 0: node 4
- Second 1: node 3
- Second 2: node 1
- Second 3: node 2
- Second 4: node 5

Output: 4

## Approach

- In a binary tree, nodes do not have parent pointers.

- But fire can spread **upwards** to parent also.
- So we first convert the binary tree into an **undirected graph**.
- Every node will be connected to:
  - its left child
  - its right child
  - its parent
- After conversion, the problem becomes:
  - starting from the target node, find how many seconds it takes to visit all nodes level by level.

Steps:

1. Traverse the tree and build an adjacency list (graph) using recursion.
  2. Start BFS from the target node.
  3. Use a visited set to avoid revisiting nodes.
  4. Each BFS level represents **1 second of burning**.
  5. Count how many levels are required until all nodes are burned.
- 

### Code (C++)

```
#include <bits/stdc++.h>
using namespace std;

struct TreeNode {
 int val;
 TreeNode* left;
 TreeNode* right;
 TreeNode(int x) : val(x), left(NULL), right(NULL) {}
};
```

```

class Solution {
public:
 int minTime(TreeNode* root, int target) {
 unordered_map<int, vector<int>> graph;
 buildGraph(root, NULL, graph);

 queue<int> q;
 unordered_set<int> visited;

 q.push(target);
 visited.insert(target);

 int time = 0;

 while (!q.empty()) {
 int size = q.size();
 bool spread = false;

 for (int i = 0; i < size; i++) {
 int node = q.front();
 q.pop();

 for (int nbr : graph[node]) {
 if (!visited.count(nbr)) {
 visited.insert(nbr);
 q.push(nbr);
 spread = true;
 }
 }
 }

 if (spread) time++;
 }

 return time;
 }
}

```

```

private:
 void buildGraph(TreeNode* node, TreeNode* parent,
 unordered_map<int, vector<int>>& graph) {
 if (!node) return;

 if (parent) {
 graph[node->val].push_back(parent->val);
 graph[parent->val].push_back(node->val);
 }

 buildGraph(node->left, node, graph);
 buildGraph(node->right, node, graph);
 }
};

int main() {
 TreeNode* root = new TreeNode(1);
 root->left = new TreeNode(2);
 root->right = new TreeNode(3);
 root->left->left = new TreeNode(4);
 root->right->left = new TreeNode(5);
 root->right->right = new TreeNode(6);
 root->left->left->right = new TreeNode(7);

 Solution sol;
 int target = 1;

 cout << "Minimum time to burn the tree: "
 << sol.minTime(root, target);

 return 0;
}

```

---

## Complexity Analysis

**Time Complexity: O(N)**

- One traversal to build the graph.
- One BFS traversal to simulate burning.
- Each node is processed once.

### Space Complexity: O(N)

- Adjacency list stores all nodes.
- Visited set and BFS queue can store up to N nodes.
- Hence linear extra space is used.

## 26. Count Number of Nodes in a Binary Tree

You are given the root of a **Complete Binary Tree**.

Your task is to count and return the total number of nodes in the tree.

A **Complete Binary Tree** is a tree where:

- All levels are completely filled except possibly the last.
- The last level nodes are filled from left to right.

---

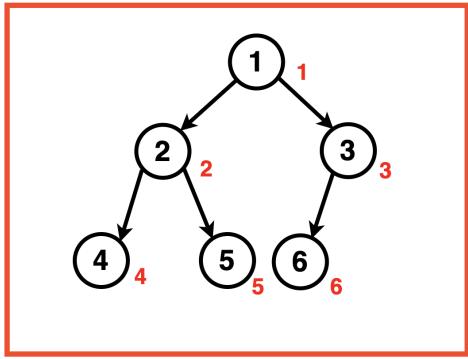
### Example Explanation

Example 1:

Tree: 1 2 3 4 5 6

All nodes are present level by level.

Total nodes = 6.



Example 2:

Tree: 2 4 3 5 9 8 7 1 6

All nodes follow the complete binary tree rule.

Total nodes = 9.

---

## Approach 1: Brute Force Traversal

### Algorithm

- Traverse the entire binary tree.
- Every time we visit a node, increase a counter by 1.
- We can use any traversal (inorder, preorder, postorder).
- Finally, return the counter value.

Steps:

1. Initialize count = 0.
  2. Traverse the tree recursively.
  3. For each non-null node, increment count.
  4. Return count.
-

## Code (C++)

```
#include <bits/stdc++.h>
using namespace std;

struct TreeNode {
 int val;
 TreeNode* left;
 TreeNode* right;
 TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

class Solution {
public:
 void inorder(TreeNode* root, int &count) {
 if (root == NULL) return;
 count++;
 inorder(root->left, count);
 inorder(root->right, count);
 }

 int countNodes(TreeNode* root) {
 int count = 0;
 inorder(root, count);
 return count;
 }
};

int main() {
 TreeNode* root = new TreeNode(1);
 root->left = new TreeNode(2);
 root->right = new TreeNode(3);
 root->left->left = new TreeNode(4);
 root->left->right = new TreeNode(5);
 root->right->left = new TreeNode(6);

 Solution sol;
 cout << sol.countNodes(root);
 return 0;
}
```

}

---

## Complexity Analysis

### Time Complexity: O(N)

Every node is visited once.

### Space Complexity: O(N)

Recursive stack can go up to N in the worst case (skewed tree).

---

## Approach 2: Optimal Using Complete Binary Tree Property

### Algorithm

This approach uses the special property of a **complete binary tree**.

Key idea:

- If the left height and right height of a tree are equal, then the tree is a **perfect binary tree**.

A perfect binary tree with height h has:

$$\text{nodes} = 2^h - 1$$

•

Steps:

1. Find the height of the leftmost path.
2. Find the height of the rightmost path.
3. If both heights are equal:
  - Directly return ( $2^{\text{height}} - 1$ ).

4. Otherwise:

- Recursively count nodes in left and right subtrees.
  - Return `1 + leftCount + rightCount`.
- 

## Code (C++)

```
#include <bits/stdc++.h>
using namespace std;

struct TreeNode {
 int val;
 TreeNode* left;
 TreeNode* right;
 TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

class Solution {
public:
 int countNodes(TreeNode* root) {
 if (root == NULL) return 0;

 int lh = leftHeight(root);
 int rh = rightHeight(root);

 if (lh == rh) {
 return (1 << lh) - 1; //2^h -1
 }

 return 1 + countNodes(root->left) + countNodes(root->right);
 }

 int leftHeight(TreeNode* node) {
 int h = 0;
 while (node) {
 h++;
 }
 }
}
```

```

 node = node->left;
 }
 return h;
}

int rightHeight(TreeNode* node) {
 int h = 0;
 while (node) {
 h++;
 node = node->right;
 }
 return h;
};

int main() {
 TreeNode* root = new TreeNode(1);
 root->left = new TreeNode(2);
 root->right = new TreeNode(3);
 root->left->left = new TreeNode(4);
 root->left->right = new TreeNode(5);
 root->right->left = new TreeNode(6);

 Solution sol;
 cout << sol.countNodes(root);
 return 0;
}

```

---

## Complexity Analysis

### Time Complexity: $O(\log N \times \log N)$

- Height calculation takes  $O(\log N)$ .
- Recursive calls happen at most  $\log N$  times.

### Space Complexity: $O(\log N)$

- Recursion depth equals height of the tree.
- Since the tree is complete, height is  $\log N$ .

## 27. Requirements needed to construct a Unique Binary Tree | Theory

You are given **two traversal types** of a binary tree, and you need to decide **whether these two traversals are sufficient to construct a UNIQUE binary tree**.

Each traversal is represented by a number:

- 1 → Preorder
- 2 → Inorder
- 3 → Postorder

The task is **not to construct the tree**, but only to check **if a unique binary tree is possible** from the given pair of traversals.

---

### Key Idea (Very Important)

A **binary tree can be uniquely constructed only if one of the traversals is INORDER**, combined with either:

- Preorder
- Postorder

If **Inorder traversal is missing**, then the tree **cannot be uniquely determined**.

---

## Why Inorder is Necessary

- **Inorder traversal** gives the exact position of the root between left and right subtrees.
- Preorder tells us the root first.
- Postorder tells us the root last.
- Without inorder, we cannot decide how nodes are split into left and right subtrees.

So:

- Inorder + Preorder → Unique tree
- Inorder + Postorder → Unique tree
- Preorder + Postorder → **Not unique**

---

## Valid Traversal Combinations

| Traversal Pair              | Unique Tree Possible? | Reason                         |
|-----------------------------|-----------------------|--------------------------------|
| Preorder (1) + Inorder (2)  | Yes                   | Inorder gives left/right split |
| Inorder (2) + Preorder (1)  | Yes                   | Same as above                  |
| Postorder (3) + Inorder (2) | Yes                   | Inorder gives structure        |
| Inorder (2) + Postorder (3) | Yes                   | Same as above                  |

Preorder (1) + Postorder (3)      No      Multiple trees possible

Postorder (3) + Preorder (1)      No      Multiple trees possible

---

## Example Explanation

### Example 1

Input: a = 1 (Preorder), b = 2 (Inorder)

Output: true

Explanation:

Preorder gives root order, inorder gives exact left-right separation.

So, a unique binary tree can be constructed.

---

### Example 2

Input: a = 1 (Preorder), b = 3 (Postorder)

Output: false

Explanation:

Without inorder traversal, we cannot uniquely determine the structure.

Multiple binary trees are possible with the same preorder and postorder.

---

## Final Rule to Remember (Interview Ready)

**A unique binary tree can be constructed if and only if one of the traversals is INORDER.**

This logic works in **O(1) time and O(1) space**, since we are only checking traversal types, not building the tree.

# 28. Construct A Binary Tree from Inorder and Preorder Traversal

You are given two arrays:

- **Preorder traversal** of a binary tree
- **Inorder traversal** of the same binary tree

Your task is to **construct the unique binary tree** represented by these two traversals and return its root.

A unique binary tree is always possible when **Preorder + Inorder** traversals are given.

---

## Question Explanation

- **Preorder traversal** visits nodes in this order:  
Root → Left → Right
- **Inorder traversal** visits nodes in this order:  
Left → Root → Right

From preorder, we always know **which node is the root**.

From inorder, we know **how the tree is split into left and right subtrees**.

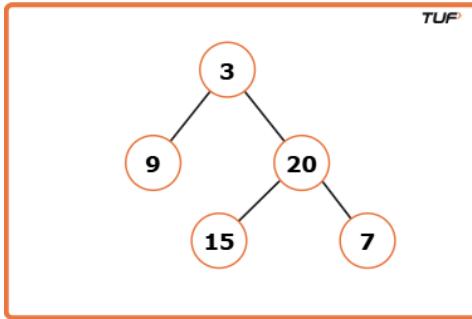
By combining both traversals, we can recursively build the exact tree.

---

## Example Explanation

### Input

```
preorder = [3, 9, 20, 15, 7]
inorder = [9, 3, 15, 20, 7]
```



Steps:

- First element of preorder is 3 → root
- In inorder, 3 splits array into:
  - Left subtree → [ 9 ]
  - Right subtree → [ 15, 20, 7 ]
- Recursively repeat the same steps for left and right parts

This process builds the unique binary tree.

---

## Approach

1. Store the index of each value from inorder traversal in a map for O(1) lookup.
2. Use a recursive function with index ranges instead of slicing arrays.
3. Base Case:
  - If preorder range or inorder range becomes invalid, return NULL.
4. Take the first element of the current preorder range as the root.
5. Find the root's index in inorder using the map.
6. Calculate the number of nodes in the left subtree.

7. Recursively construct:

- Left subtree using left part of inorder
- Right subtree using right part of inorder

8. Return the constructed root.

---

### Code (C++)

```
#include <bits/stdc++.h>

using namespace std;

// TreeNode structure
struct TreeNode {

 int val;

 TreeNode *left;

 TreeNode *right;

 TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}

};

class Solution {

public:

 TreeNode* buildTree(vector<int>& preorder, vector<int>& inorder) {

 map<int,int> inMap;

 for(int i=0;i<inorder.size();i++){


```

```

 inMap[inorder[i]] = i;

 }

 return build(preorder, 0, preorder.size()-1,
 inorder, 0, inorder.size()-1, inMap);
}

private:

 TreeNode* build(vector<int>& preorder, int preStart, int preEnd,
 vector<int>& inorder, int inStart, int inEnd,
 map<int, int>& inMap){

 if(preStart > preEnd || inStart > inEnd) return nullptr;

 TreeNode* root = new TreeNode(preorder[preStart]);

 int inRoot = inMap[root->val];

 int leftSize = inRoot - inStart;

 root->left = build(preorder, preStart+1, preStart+leftSize,
 inorder, inStart, inRoot-1, inMap);

 root->right = build(preorder, preStart+leftSize+1, preEnd,
 inorder, inRoot+1, inEnd, inMap);

 return root;
}
};


```

```

// Inorder print for verification

void printInorder(TreeNode* root){

 if(!root) return;

 printInorder(root->left);

 cout<<root->val<<" ";

 printInorder(root->right);

}

int main(){

 vector<int> preorder = {3,9,20,15,7};

 vector<int> inorder = {9,3,15,20,7};

 Solution sol;

 TreeNode* root = sol.buildTree(preorder,inorder);

 printInorder(root);

 return 0;

}

```

### Time Complexity:

$O(N)$

Each node is created and processed exactly once.

### Space Complexity:

$O(N)$

- Hash map stores  $N$  elements
- Recursion stack can go up to height of tree (worst case  $N$ )

## 29. Construct Binary Tree from Inorder and Postorder Traversal

You are given two arrays:

- **Inorder traversal** of a binary tree
- **Postorder traversal** of the same binary tree

Your task is to **construct the unique binary tree** represented by these two traversals and return its root.

A unique binary tree can always be constructed when **Inorder + Postorder** traversals are given.

---

- **Inorder traversal** follows:  
Left → Root → Right
- **Postorder traversal** follows:  
Left → Right → Root

From **postorder**, the **last element is always the root** of the current tree or subtree.

From **inorder**, once we know the root, we can split the tree into:

- Left subtree (elements before root)

- Right subtree (elements after root)

Using this idea recursively, the entire tree can be constructed.

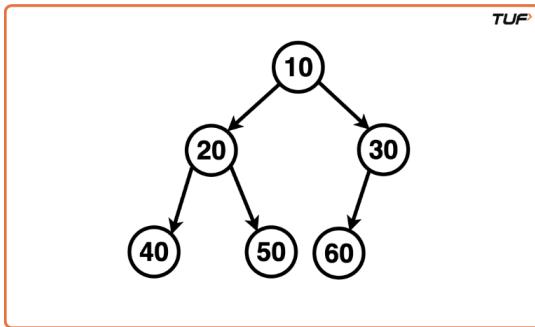
---

## Example Explanation

### Input

Inorder = [40, 20, 50, 10, 60, 30]

Postorder = [40, 50, 20, 60, 30, 10]



Steps:

1. Last element of postorder is 10 → root
2. In inorder, 10 splits array into:
  - Left: [40, 20, 50]
  - Right: [60, 30]
3. Recursively repeat the same steps for left and right parts

This process builds the unique binary tree.

---

## Approach

1. Create a hashmap to store the index of each element in the inorder traversal for fast lookup.
  2. Define a recursive function that works with index ranges instead of slicing arrays.
  3. Base Case:
    - o If inorder or postorder range becomes invalid, return NULL.
  4. Take the **last element of the current postorder range** as the root.
  5. Find the root's index in inorder using the hashmap.
  6. Calculate the size of the left subtree.
  7. Recursively build:
    - o Left subtree using corresponding inorder and postorder ranges
    - o Right subtree using remaining ranges
  8. Return the constructed root.
- 

## Code (C++)

```
#include <bits/stdc++.h>
using namespace std;

// TreeNode structure
struct TreeNode {
 int val;
 TreeNode *left, *right;
 TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

class Solution {
public:
 TreeNode* buildTree(vector<int>& inorder, vector<int>& postorder)
 {
 map<int,int> inMap;
```

```

 for(int i=0;i<inorder.size();i++){
 inMap[inorder[i]] = i;
 }
 return build(inorder,0,inorder.size()-1,
 postorder,0,postorder.size()-1,inMap);
 }

private:
 TreeNode* build(vector<int>& inorder,int inStart,int inEnd,
 vector<int>& postorder,int postStart,int postEnd,
 map<int,int>& inMap){
 if(inStart > inEnd || postStart > postEnd) return nullptr;

 TreeNode* root = new TreeNode(postorder[postEnd]);
 int inRoot = inMap[root->val];
 int leftSize = inRoot - inStart;

 root->left = build(inorder,inStart,inRoot-1,
 postorder,postStart,postStart+leftSize-1,inMap);

 root->right = build(inorder,inRoot+1,inEnd,
 postorder,postStart+leftSize,postEnd-1,inMap);
 return root;
 }
};

// Inorder print to verify tree
void printInorder(TreeNode* root){
 if(!root) return;
 printInorder(root->left);
 cout<<root->val<<" ";
 printInorder(root->right);
}

int main(){
 vector<int> inorder = {40,20,50,10,60,30};

```

```
vector<int> postorder = {40, 50, 20, 60, 30, 10};

Solution sol;
TreeNode* root = sol.buildTree(inorder, postorder);

printInorder(root);
return 0;
}
```

---

## Complexity Analysis

### Time Complexity:

$O(N)$

Each node is created and processed exactly once.

### Space Complexity:

$O(N)$

- Hashmap stores N elements
- Recursion stack can go up to height of tree (worst case N)

# 30. Serialize And Deserialize a Binary Tree

Given a Binary Tree, you need to **serialize** it into a string and then **deserialize** that string back into the original binary tree structure.

- **Serialization** means converting the tree into a string format that can be stored or transmitted.
- **Deserialization** means reconstructing the exact same binary tree from that string.

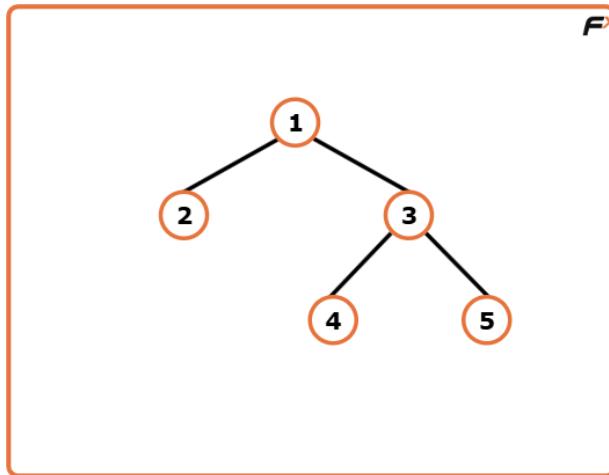
There is no restriction on the format, but after deserialization, the tree structure must remain exactly the same as the original.

---

## Example Explanation

### Input Tree:

1 2 3 -1 -1 4 5



Using **level order traversal**:

- We store node values in order
- Use # to represent NULL nodes
- Separate values using commas

**Serialized String:**

1,2,3,#,#,4,5,#,#,#,

During deserialization, we read this string and rebuild the tree level by level, reconnecting left and right children correctly.

---

## Algorithm

### Serialization (Level Order Traversal)

1. If root is NULL, return empty string.
2. Use a queue for BFS.
3. Push root into the queue.
4. While queue is not empty:
  - o Pop a node.
  - o If node is NULL, append "#, " to string.
  - o Else append node->val + ", " and push its left and right children.
5. Return the final string.

### **Deserialization (Level Order Reconstruction)**

1. If input string is empty, return NULL.
  2. Use stringstream to split values by comma.
  3. First value is the root.
  4. Use a queue to rebuild tree level by level.
  5. For each node:
    - o Read left value → if not #, create left child.
    - o Read right value → if not #, create right child.
  6. Continue until string is fully processed.
  7. Return the root.
- 

### **Code (C++)**

```
#include <bits/stdc++.h>
using namespace std;
```

```

// Definition for a binary tree node
struct TreeNode {
 int val;
 TreeNode* left;
 TreeNode* right;
 TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

class Solution {
public:
 // Serialize the tree using level order traversal
 string serialize(TreeNode* root) {
 if (!root) return "";

 string s = "";
 queue<TreeNode*> q;
 q.push(root);

 while (!q.empty()) {
 TreeNode* node = q.front();
 q.pop();

 if (node == nullptr) {
 s += "#,";

 } else {
 s += to_string(node->val) + ",";
 q.push(node->left);
 q.push(node->right);
 }
 }
 return s;
 }

 // Deserialize the string back to tree
 TreeNode* deserialize(string data) {
 if (data.size() == 0) return nullptr;

```

```

stringstream ss(data);
string str;
getline(ss, str, ',');

TreeNode* root = new TreeNode(stoi(str));
queue<TreeNode*> q;
q.push(root);

while (!q.empty()) {
 TreeNode* node = q.front();
 q.pop();

 getline(ss, str, ',');
 if (str != "#") {
 node->left = new TreeNode(stoi(str));
 q.push(node->left);
 }

 getline(ss, str, ',');
 if (str != "#") {
 node->right = new TreeNode(stoi(str));
 q.push(node->right);
 }
}
return root;
};

// Inorder traversal for checking
void inorder(TreeNode* root) {
 if (!root) return;
 inorder(root->left);
 cout << root->val << " ";
 inorder(root->right);
}

int main() {
 TreeNode* root = new TreeNode(1);

```

```

root->left = new TreeNode(2);
root->right = new TreeNode(3);
root->right->left = new TreeNode(4);
root->right->right = new TreeNode(5);

Solution sol;

string serialized = sol.serialize(root);
cout << "Serialized: " << serialized << endl;

TreeNode* deserialized = sol.deserialize(serialized);
cout << "Tree after deserialization (Inorder): ";
inorder(deserialized);
cout << endl;

return 0;
}

```

---

## Complexity Analysis

**Time Complexity:**  $O(N)$

- Serialization visits each node once.
- Deserialization also processes each node once.

**Space Complexity:**  $O(N)$

- Queue stores nodes during BFS.
- Serialized string also stores all node values including NULL markers.

# 31. Morris Preorder Traversal of a Binary Tree

Given a Binary Tree, you need to perform **Morris Preorder Traversal** and return the preorder sequence of the tree.

Preorder traversal follows the order:

**Root → Left → Right**

The key requirement of Morris Traversal is that it must be done in **O(1) extra space**, which means:

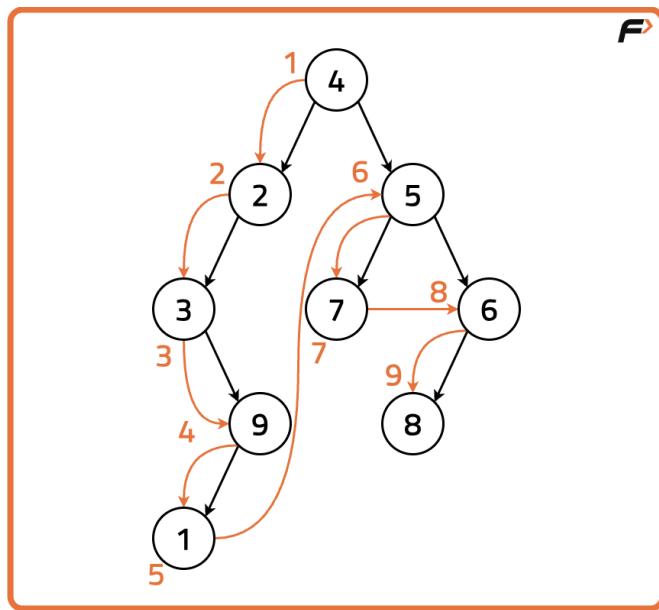
- No recursion
- No stack
- Only temporary modification of tree pointers (which are restored later)

---

## Example Explanation

**Input Tree:**

1 2 3 4 5 6



Using Morris Preorder Traversal:

- We visit the root first
- Then traverse the left subtree
- Finally traverse the right subtree
- Temporary links (threads) are created to avoid recursion and stack usage

**Output:**

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

---

## Algorithm

This algorithm is based on **Morris Inorder Traversal**, with a small change to achieve **Preorder traversal**.

1. Initialise a pointer `cur` pointing to the root.
2. While `cur` is not NULL:
  - If `cur->left` is NULL:
    - Add `cur->val` to the preorder list.
    - Move to `cur->right`.
  - Else:
    - Find the inorder predecessor of `cur` (rightmost node in left subtree).
    - If predecessor's right is NULL:
      - Create a temporary thread from predecessor to `cur`.
      - Move `cur` to its left child.
    - Else:

- Remove the temporary thread.
  - Add `cur->val` to the preorder list.
  - Move `cur` to its right child.
3. Continue until the entire tree is traversed.
  4. Return the preorder list.

This traversal modifies the tree temporarily but restores it before moving ahead.

---

### Code (C++)

```
#include <bits/stdc++.h>
using namespace std;

// TreeNode structure
struct TreeNode {
 int val;
 TreeNode *left;
 TreeNode *right;
 TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

class Solution {
public:
 vector<int> getPreorder(TreeNode* root) {
 vector<int> preorder;
 TreeNode* cur = root;

 while (cur != NULL) {
 if (cur->left == NULL) {
 preorder.push_back(cur->val);
 cur = cur->right;
 } else {
 TreeNode* prev = cur->left;
 while (prev->right && prev->right != cur) {

```

```

 prev = prev->right;
 }

 if (prev->right == NULL) {
 preorder.push_back(cur->val);
 prev->right = cur;
 cur = cur->left;
 } else {
 prev->right = NULL;
 cur = cur->right;
 }
}

return preorder;
}

};

int main() {
 TreeNode* root = new TreeNode(1);
 root->left = new TreeNode(2);
 root->right = new TreeNode(3);
 root->left->left = new TreeNode(4);
 root->left->right = new TreeNode(5);
 root->left->right->right = new TreeNode(6);

 Solution sol;
 vector<int> preorder = sol.getPreorder(root);

 cout << "Binary Tree Morris Preorder Traversal: ";
 for (int x : preorder) cout << x << " ";
 cout << endl;

 return 0;
}

```

---

**Time Complexity:**  $O(2N)$

Each node is visited at most twice:

- Once when creating the temporary thread
- Once when removing the thread

Hence overall time is linear.

**Space Complexity:**  $O(1)$

No recursion or stack is used. Only constant extra pointers are required.

## 32. Morris Inorder Traversal of a Binary Tree

Given a Binary Tree, implement **Morris Inorder Traversal** and return the inorder sequence of the tree.

Inorder traversal follows the order:

**Left → Root → Right**

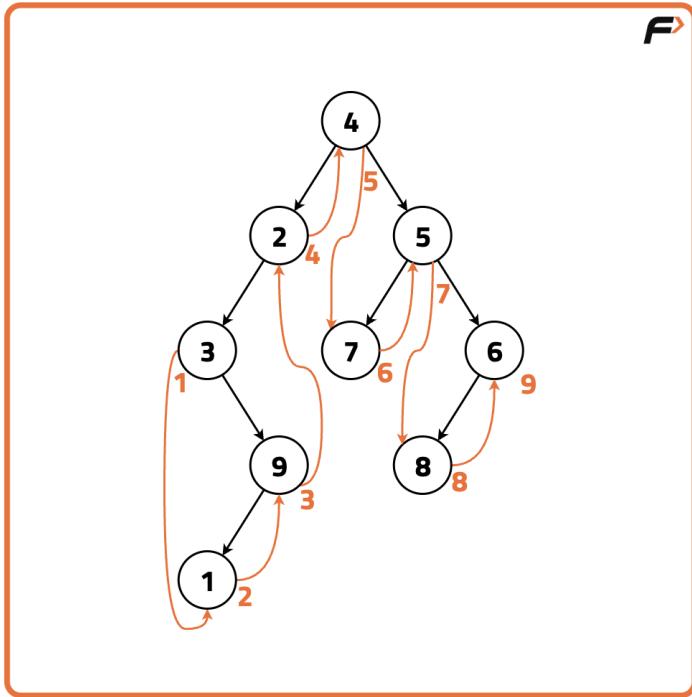
Morris Traversal is special because it performs this traversal using  **$O(1)$  extra space**, which means:

- No recursion
- No stack
- The tree structure is temporarily modified and then restored

---

**Input Tree:**

1 2 3 4 5 6



Inorder traversal visits:

- Left subtree first
- Then the root
- Then the right subtree

#### **Output:**

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

Morris traversal achieves this without storing nodes externally.

---

## **Approach:**

### **Algorithm**

Morris Inorder Traversal uses the concept of an **inorder predecessor**.

1. Initialise a pointer `cur` pointing to the root.

2. While `cur` is not `NULL`:

- **Case 1:** If `cur->left` is `NULL`
  - Add `cur->val` to result.
  - Move `cur` to `cur->right`.
- **Case 2:** If `cur->left` is not `NULL`
  - Find inorder predecessor (rightmost node in left subtree).
  - If predecessor's right is `NULL`:
    - Create a temporary link from predecessor to `cur`.
    - Move `cur` to its left child.
  - Else:
    - Remove the temporary link.
    - Add `cur->val` to result.
    - Move `cur` to its right child.

3. Continue until the entire tree is traversed.

4. Return the inorder sequence.

Temporary links ensure we return back after finishing the left subtree.

---

### Code (C++)

```
#include <bits/stdc++.h>
using namespace std;

// TreeNode structure
struct TreeNode {
 int val;
```

```

TreeNode *left;
TreeNode *right;
TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

class Solution {
public:
 vector<int> getInorder(TreeNode* root) {
 vector<int> inorder;
 TreeNode* cur = root;

 while (cur != NULL) {
 if (cur->left == NULL) {
 inorder.push_back(cur->val);
 cur = cur->right;
 } else {
 TreeNode* prev = cur->left;
 while (prev->right && prev->right != cur) {
 prev = prev->right;
 }

 if (prev->right == NULL) {
 prev->right = cur;
 cur = cur->left;
 } else {
 prev->right = NULL;
 inorder.push_back(cur->val);
 cur = cur->right;
 }
 }
 }
 return inorder;
 }
};

int main() {
 TreeNode* root = new TreeNode(1);
 root->left = new TreeNode(2);

```

```

root->right = new TreeNode(3);
root->left->left = new TreeNode(4);
root->left->right = new TreeNode(5);
root->left->right->right = new TreeNode(6);

Solution sol;
vector<int> inorder = sol.getInorder(root);

cout << "Binary Tree Morris Inorder Traversal: ";
for (int x : inorder) cout << x << " ";
cout << endl;

return 0;
}

```

---

## Complexity Analysis

**Time Complexity:**  $O(2N)$

Each node is visited at most twice:

- Once when creating the temporary thread
- Once when removing it

So overall time is linear.

**Space Complexity:**  $O(1)$

No recursion, no stack, only constant extra pointers are used

# 33. Flatten Binary Tree to Linked List

Given a Binary Tree, convert it into a **linked list in-place** such that the nodes follow the **preorder traversal order** of the tree.

Rules to follow:

- Use the **right pointer** as the next pointer of the linked list.
- Set all **left pointers to NULL**.
- Do not create new nodes.
- Modify the tree **in-place**.

Preorder traversal order is:

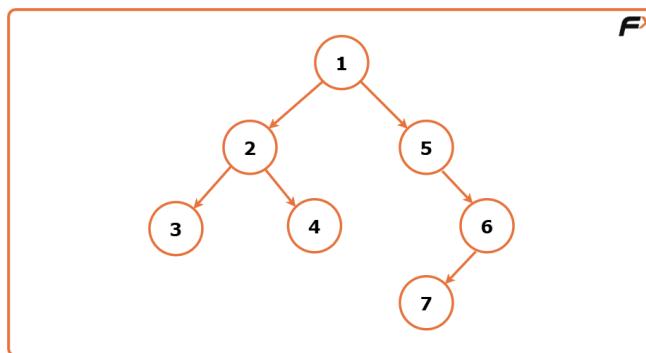
**Root → Left → Right**

---

## Example Explanation

**Input Tree:**

1 2 5 3 4 -1 6 -1 -1 -1 -1 7



Preorder traversal:

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

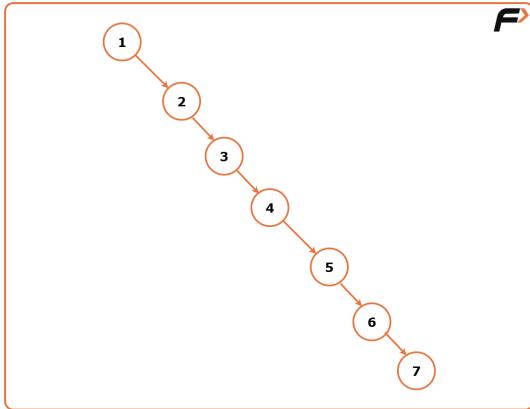
After flattening:

- Each node's left = NULL

- Each node's right points to the next node in preorder

**Output (Right pointers):**

1 → 2 → 3 → 4 → 5 → 6 → 7



## Approach 1: Brute Force (Recursive – Reverse Preorder)

### Algorithm

1. Use a global pointer `prev` to store the previously processed node.
2. Traverse the tree in **reverse preorder**:
  - Right subtree
  - Left subtree
3. For each node:
  - Set `root->right = prev`
  - Set `root->left = NULL`
  - Update `prev = root`
4. This builds the linked list in correct preorder order when recursion unwinds.

---

## Code (C++)

```
#include <bits/stdc++.h>
using namespace std;

struct TreeNode {
 int val;
 TreeNode* left;
 TreeNode* right;
 TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

class Solution {
public:
 TreeNode* prev = nullptr;

 void flatten(TreeNode* root) {
 if (root == nullptr) return;

 flatten(root->right);
 flatten(root->left);

 root->right = prev;
 root->left = nullptr;
 prev = root;
 }
};

void printFlattenTree(TreeNode* root) {
 while (root) {
 cout << root->val << " ";
 root = root->right;
 }
}

int main() {
 TreeNode* root = new TreeNode(1);
```

```

root->left = new TreeNode(2);
root->right = new TreeNode(5);
root->left->left = new TreeNode(3);
root->left->right = new TreeNode(4);
root->right->right = new TreeNode(6);

Solution sol;
sol.flatten(root);

printFlattenTree(root);
return 0;
}

```

---

## Complexity Analysis

- **Time Complexity:**  $O(N)$   
Each node is visited once.
  - **Space Complexity:**  $O(H)$   
Due to recursion stack, where  $H$  is tree height.
- 

## Approach 2: Better Approach (Iterative Using Stack)

### Algorithm

1. Use a stack to simulate preorder traversal.
2. Push root into stack.
3. While stack is not empty:
  - Pop the top node.
  - Push right child (if exists).
  - Push left child (if exists).

- Set current node's right to stack top.
- Set current node's left to NULL.

4. This directly links nodes in preorder order.

---

### Code (C++)

```
#include <bits/stdc++.h>
using namespace std;

struct TreeNode {
 int val;
 TreeNode* left;
 TreeNode* right;
 TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

class Solution {
public:
 void flatten(TreeNode* root) {
 if (root == nullptr) return;

 stack<TreeNode*> st;
 st.push(root);

 while (!st.empty()) {
 TreeNode* cur = st.top();
 st.pop();

 if (cur->right) st.push(cur->right);
 if (cur->left) st.push(cur->left);

 if (!st.empty()) cur->right = st.top();
 cur->left = nullptr;
 }
 }
};
```

```

void printFlattenTree(TreeNode* root) {
 while (root) {
 cout << root->val << " ";
 root = root->right;
 }
}

int main() {
 TreeNode* root = new TreeNode(1);
 root->left = new TreeNode(2);
 root->right = new TreeNode(5);
 root->left->left = new TreeNode(3);
 root->left->right = new TreeNode(4);
 root->right->right = new TreeNode(6);

 Solution sol;
 sol.flatten(root);

 printFlattenTree(root);
 return 0;
}

```

---

## Complexity Analysis

- **Time Complexity:**  $O(N)$   
Each node is processed once.
  - **Space Complexity:**  $O(H)$   
Stack space depends on tree height.
- 

## Approach 3: Optimal Approach (Morris Traversal Style)

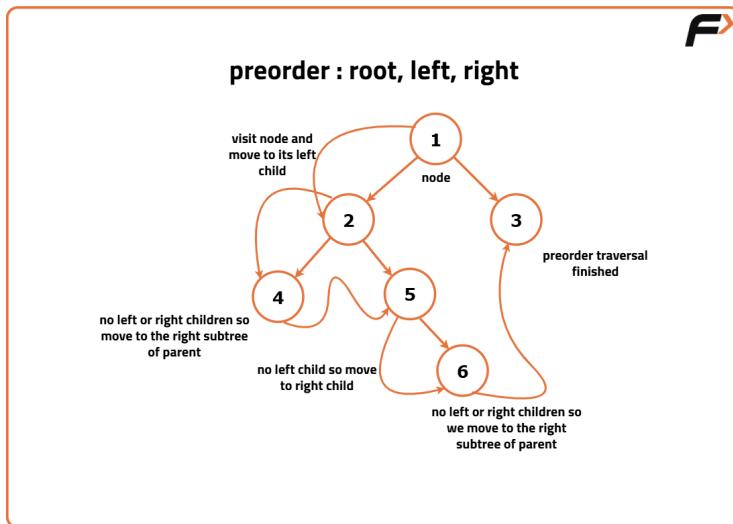
1. Start with `curr = root`.

2. While curr is not NULL:

- If curr->left exists:
  - Find the rightmost node in left subtree.
  - Connect it to curr->right.
  - Move left subtree to right.
  - Set curr->left = NULL.
- Move curr = curr->right.

3. Tree becomes a right-skewed linked list in preorder.

This modifies pointers in-place without recursion or stack.



## Code (C++)

```
#include <bits/stdc++.h>
using namespace std;

struct TreeNode {
```

int val;

TreeNode\* left;

```

TreeNode* right;
TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

class Solution {
public:
 void flatten(TreeNode* root) {
 TreeNode* curr = root;

 while (curr) {
 if (curr->left) {
 TreeNode* pre = curr->left;
 while (pre->right) {
 pre = pre->right;
 }
 pre->right = curr->right;
 curr->right = curr->left;
 curr->left = nullptr;
 }
 curr = curr->right;
 }
 }

 void printFlattenTree(TreeNode* root) {
 while (root) {
 cout << root->val << " ";
 root = root->right;
 }
 }
}

int main() {
 TreeNode* root = new TreeNode(1);
 root->left = new TreeNode(2);
 root->right = new TreeNode(5);
 root->left->left = new TreeNode(3);
 root->left->right = new TreeNode(4);
 root->right->right = new TreeNode(6);
}

```

```
Solution sol;
sol.flatten(root);

printFlattenTree(root);
return 0;
}
```

---

## Complexity Analysis

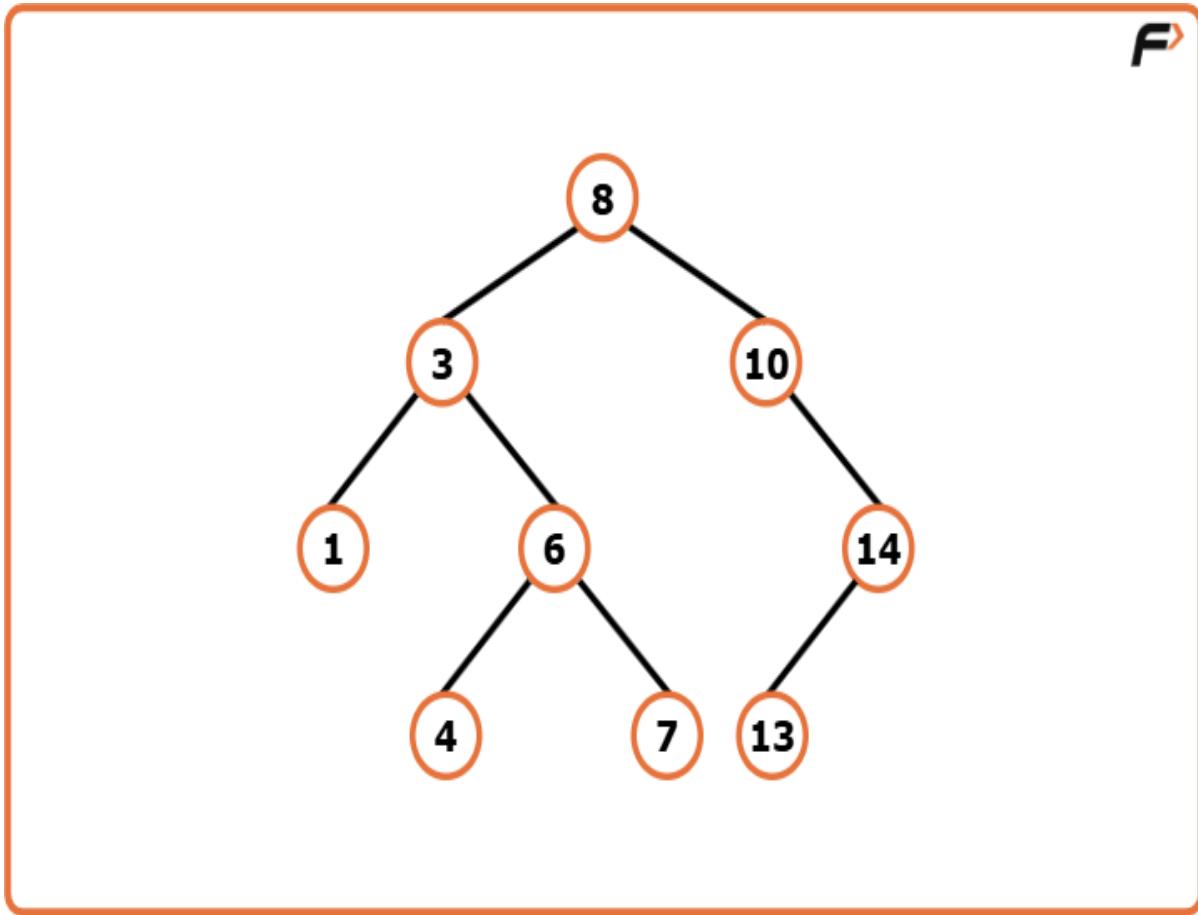
- **Time Complexity:**  $O(2N)$   
Each node may be visited twice while adjusting links.
- **Space Complexity:**  $O(1)$   
No recursion or stack used.

# **Binary Search Tree**

# 1. Introduction to Binary Search Trees

Binary Search Trees, popularly known as BST, are the category of Binary Trees used to optimize the operation of searching an element among the Tree Nodes in a Binary Tree.

Let's understand the dynamics of a Binary Search Tree using an illustration below:

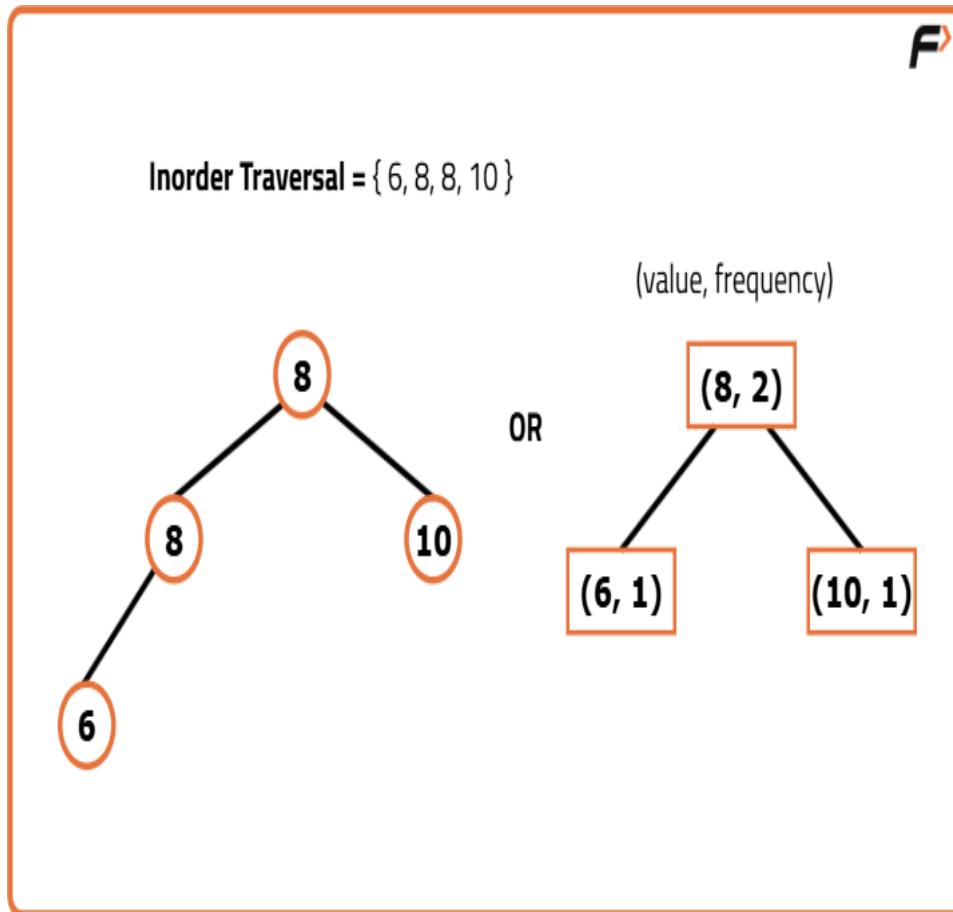


As we can visualize from the above Tree, the right subtree has all the elements greater than the root element and the left subtree has all the elements lesser than the root element. A point to note here is that **every subtree is itself a Binary Search Tree** as it contains a subtree whose root node is lesser than the value of the current node and another subtree whose root node is greater than the value of the current node.

For every given node in a Binary Search Tree if a left or a right child exists then:

Left Child < Node < Right Child

Now, for the general implementation of Binary Search Trees, **duplicate node values are not allowed**. However, in some exceptional cases of Binary Search Tree implementation, we can represent a Binary Search Tree with duplicate node values in the following ways:



## Need for Binary Search Tree

After understanding how a Binary Search Tree is represented, let us now understand why there is a need to use a Binary Search Tree instead of a simple Binary Tree.

So, generally in a BST, the maximum height in almost all cases is kept in order of  $\log(2N)$  where  $N$  = No. of nodes which is in contrast to the plain Binary Tree whose maximum height can reach the order of  $N$  when the tree is **skewed**.

This particularly makes the time of searching for a given node in a Binary Search Tree a lot less than searching in a simple Binary Tree.

For example, if we have to search for an element, we can directly compare it to the root node value of the BST, if the value matches then we have found the element, if say the value is greater than the root node, we say that we now move on to searching that element in the right subtree because the right subtree has all the node values greater than the root value. The search process is terminated for the left subtree as it would only search for those nodes which have values lesser than the root.

Note that, this is very similar to the Binary Search operation which we perform in Arrays to optimize the time taken to search for an element using Linear Search. Hence this category of trees is named **Binary Search Trees**.

## 2. Search in a Binary Search Tree

Given a **Binary Search Tree (BST)** and a key value, you need to **search for the node whose value is equal to the given key**.

If such a node exists, return that node; otherwise, return nullptr.

In a Binary Search Tree:

- Left subtree contains values **smaller** than the node.
- Right subtree contains values **greater** than the node.

---

### Example Explanation

#### Example 1

Input BST: 8 5 12 4 7 10 14

Key = 10

We start from root 8:

- $10 > 8 \rightarrow$  move right
- reach 12
- $10 < 12 \rightarrow$  move left
- reach 10  $\rightarrow$  found

Output: True

### Example 2

Input BST: 4 2 6 1 3 5 7

Key = 3

If traversal ends at `nullptr` without finding the key, the value does not exist.

Output: False

---

## Approach

1. Start from the root node.
2. While the current node is not `NULL`:
  - If the current node's value equals the key, return the current node.
  - If the key is smaller than the current node's value, move to the left child.
  - If the key is greater than the current node's value, move to the right child.
3. If the traversal reaches `NULL`, the key is not present. Return `nullptr`.

This works because the BST property helps us decide the direction at each step.

---

## Code (C++)

```
#include <bits/stdc++.h>

using namespace std;

// Definition of TreeNode

struct TreeNode {

 int val;

 TreeNode* left;

 TreeNode* right;

 TreeNode(int data) {

 val = data;

 left = right = nullptr;

 }

};

class Solution {
```

```

// Function to search for a node with given key in BST

TreeNode* searchBST(TreeNode* root, int target) {

 // Traverse until target is found or root becomes NULL

 while (root != nullptr && root->val != target) {

 // Move left if target is smaller

 if (target < root->val) {

 root = root->left;

 }

 // Move right if target is larger

 else {

 root = root->right;

 }

 }

 // Returns node if found, otherwise nullptr

 return root;

};


```



## Complexity Analysis

- **Time Complexity:**  
 $O(\log N)$  in a balanced BST because each comparison removes half of the tree.  
 $O(N)$  in the worst case for a skewed tree.
- **Space Complexity:**  
 $O(1)$  because the solution is iterative and uses no extra data structures.

# 3. Find Minimum / Maximum in a Binary Search Tree

Given a **Binary Search Tree (BST)**, find:

- the **minimum value node**
- the **maximum value node**

In a BST:

- All values in the **left subtree** are smaller than the root.
- All values in the **right subtree** are greater than the root.

This property makes finding min and max very straightforward.

---

## Question Explanation

You are given the root of a Binary Search Tree.

You need to:

- Return the node with the **minimum value**
- Return the node with the **maximum value**

If the tree is empty, return `nullptr`.

---

## Approach

### To find Minimum value:

1. Start from the root.
2. Keep moving to the **left child** because smaller values are always on the left.
3. When a node has no left child, that node is the **minimum**.

### To find Maximum value:

1. Start from the root.
  2. Keep moving to the **right child** because larger values are always on the right.
  3. When a node has no right child, that node is the **maximum**.
- 

## Code (C++)

```
#include <bits/stdc++.h>
using namespace std;

// Definition of TreeNode
struct TreeNode {
 int val;
 TreeNode* left;
 TreeNode* right;

 TreeNode(int x) {
```

```

 val = x;
 left = right = nullptr;
 }
};

class Solution {
public:
 // Function to find minimum value node in BST
 TreeNode* findMin(TreeNode* root) {
 if (root == nullptr) return nullptr;

 while (root->left != nullptr) {
 root = root->left;
 }
 return root;
 }

 // Function to find maximum value node in BST
 TreeNode* findMax(TreeNode* root) {
 if (root == nullptr) return nullptr;

 while (root->right != nullptr) {
 root = root->right;
 }
 return root;
 }
};

int main() {
 TreeNode* root = new TreeNode(8);
 root->left = new TreeNode(3);
 root->right = new TreeNode(10);
 root->left->left = new TreeNode(1);
 root->left->right = new TreeNode(6);
 root->right->right = new TreeNode(14);

 Solution sol;

```

```

TreeNode* minNode = sol.findMin(root);
TreeNode* maxNode = sol.findMax(root);

if (minNode)
 cout << "Minimum value: " << minNode->val << endl;
if (maxNode)
 cout << "Maximum value: " << maxNode->val << endl;

return 0;
}

```

---

## Complexity Analysis

- **Time Complexity:**  
 $O(H)$  where  $H$  is the height of the tree.
  - Balanced BST  $\rightarrow O(\log N)$
  - Skewed BST  $\rightarrow O(N)$
- **Space Complexity:**  
 $O(1)$  because no recursion or extra data structures are used.

## 4. Ceil in a Binary Search Tree

Given a **Binary Search Tree (BST)** and a key value, you need to find the **ceiling** of that key. The **ceiling** is defined as the **smallest value in the BST that is greater than or equal to the given key**.

If no such value exists, return  $-1$  (or `nullptr` logically).

Because this is a BST, we can use its ordering property to search efficiently.

---

## Question Explanation

In a Binary Search Tree:

- Left subtree contains **smaller values**
- Right subtree contains **larger values**

While traversing the tree:

- If a node value is **equal** to the key, that value itself is the ceiling.
- If a node value is **greater** than the key, it can be a possible ceiling, but we still try to find a smaller valid one on the left.
- If a node value is **smaller** than the key, it cannot be the ceiling, so we move right.

---

## Example Explanation

### Example 1

BST: 10 5 15 2 6

Key = 7

Traversal:

- Start at 10 → greater than 7 → possible ceil = 10 → move left
- At 5 → smaller than 7 → move right
- At 6 → smaller than 7 → move right → null

Final answer = **10**

---

## Algorithm

1. Initialize a variable **ceil** = -1.
2. Start traversing from the root.

3. While the current node is not null:
    - o If node value equals key → update ceil and return.
    - o If key is greater than node value → move to right subtree.
    - o If key is smaller than node value:
      - Update ceil with current node value.
      - Move to left subtree.
  4. Return ceil after traversal ends.
- 

### Code (C++)

```
#include <bits/stdc++.h>
using namespace std;

// Definition of TreeNode
struct TreeNode {
 int val;
 TreeNode* left;
 TreeNode* right;

 TreeNode(int x) {
 val = x;
 left = right = nullptr;
 }
};

class Solution {
public:
 int findCeil(TreeNode* root, int key) {
 int ceil = -1;

 while (root != nullptr) {
 if (root->val == key) {

```

```

 ceil = root->val;
 return ceil;
 }
 if (key > root->val) {
 root = root->right;
 } else {
 ceil = root->val;
 root = root->left;
 }
}
return ceil;
};

// Inorder traversal for verification
void inorder(TreeNode* root) {
 if (!root) return;
 inorder(root->left);
 cout << root->val << " ";
 inorder(root->right);
}

int main() {
 TreeNode* root = new TreeNode(10);
 root->left = new TreeNode(5);
 root->right = new TreeNode(15);
 root->left->left = new TreeNode(2);
 root->left->right = new TreeNode(6);

 Solution sol;
 int key = 7;

 cout << "BST Inorder: ";
 inorder(root);
 cout << endl;

 int result = sol.findCeil(root, key);
 cout << "Ceil of " << key << " is: " << result << endl;
}

```

```
 return 0;
}
```

---

## Complexity Analysis

- **Time Complexity:**  $O(H)$   
Where  $H$  is the height of the BST.
  - Balanced BST  $\rightarrow O(\log N)$
  - Skewed BST  $\rightarrow O(N)$
- **Space Complexity:**  $O(1)$   
No recursion or extra data structures are used.

## 5. Floor in a Binary Search Tree

Given a **Binary Search Tree (BST)** and a key value, find the **floor** of that key.  
The **floor** is defined as the **largest value in the BST that is smaller than or equal to the given key**.  
If such a value does not exist, return -1.

---

### Question Explanation

In a Binary Search Tree:

- Left subtree contains values **smaller** than the current node.

- Right subtree contains values **greater** than the current node.

To find the floor:

- If we find the key itself, that value is the floor.
  - If the key is greater than the current node, the current node is a **possible floor**, and we try to find a larger one on the right.
  - If the key is smaller than the current node, the current node cannot be the floor, so we move left.
- 

## Example Explanation

### Example 1

BST: 10 5 15 2 6

Key = 7

Traversal:

- Start at 10 → 7 < 10 → move left
- At 5 → 7 > 5 → floor = 5 → move right
- At 6 → 7 > 6 → floor = 6 → move right → null

Final answer = **6**

---

## Approach 1

### Algorithm

1. Initialize a variable **floor** = -1.
2. Start traversal from the root.

3. While the current node is not null:
    - If current node value equals key, assign it to `floor` and return.
    - If key is greater than current node value:
      - Update `floor` with current node value.
      - Move to the right subtree.
    - If key is smaller than current node value:
      - Move to the left subtree.
  4. Return `floor` after traversal ends.
- 

### **Code (C++)**

```
#include <bits/stdc++.h>
using namespace std;

// Definition of TreeNode
struct TreeNode {
 int val;
 TreeNode* left;
 TreeNode* right;

 TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

class Solution {
public:
 int floorInBST(TreeNode* root, int key) {
 int floor = -1;

 while (root) {
 if (root->val == key) {
 floor = root->val;
 }
 if (root->val < key)
 root = root->right;
 else
 root = root->left;
 }
 return floor;
 }
};
```

```

 return floor;
 }
 if (key > root->val) {
 floor = root->val;
 root = root->right;
 } else {
 root = root->left;
 }
}
return floor;
};

// Inorder traversal for checking BST
void printInOrder(TreeNode* root) {
 if (!root) return;
 printInOrder(root->left);
 cout << root->val << " ";
 printInOrder(root->right);
}

int main() {
 TreeNode* root = new TreeNode(10);
 root->left = new TreeNode(5);
 root->right = new TreeNode(13);
 root->left->left = new TreeNode(3);
 root->left->right = new TreeNode(6);
 root->left->right->right = new TreeNode(9);

 cout << "BST Inorder: ";
 printInOrder(root);
 cout << endl;

 Solution sol;
 int key = 8;
 int result = sol.floorInBST(root, key);

 if (result != -1)

```

```
 cout << "Floor of " << key << " is: " << result << endl;
 else
 cout << "No floor found" << endl;

 return 0;
}
```

---

## Complexity Analysis

- **Time Complexity:**  $O(H)$

Where H is the height of the BST.

- Balanced BST  $\rightarrow O(\log N)$
- Skewed BST  $\rightarrow O(N)$

- **Space Complexity:**  $O(1)$

The algorithm is iterative and uses constant extra space.

## 6. Insert a given Node in Binary Search Tree

Given a **Binary Search Tree (BST)** and a key value, insert a new node with that key into the BST.

The insertion must follow **BST properties** and **tree structure must remain valid**.

If the BST is empty, the new node becomes the root.

---

## Question Explanation

In a Binary Search Tree:

- Left subtree contains values **smaller** than the node.
- Right subtree contains values **greater** than the node.

To insert a new value:

- Start from the root.
- Compare the key with the current node.
- Move **left** if the key is smaller.
- Move **right** if the key is greater.
- Insert the node at the first NULL position where it fits.

It is guaranteed that the key does **not already exist** in the BST.

---

## Example Explanation

BST: 4 2 6 1 3 5 7

Insert key = 8

Traversal:

- $8 > 4 \rightarrow$  move right
- $8 > 6 \rightarrow$  move right
- $8 > 7 \rightarrow$  move right  $\rightarrow$  NULL

Insert 8 as right child of 7.

---

## Algorithm

1. If root is NULL, create and return a new node.

2. Initialize a pointer `curr` starting from root.
  3. Traverse the BST:
    - o If `key < curr->val`, move to left subtree.
    - o If `key > curr->val`, move to right subtree.
  4. When a NULL position is found:
    - o Insert the new node there.
  5. Return the original root.
- 

### Code (C++)

```
#include <bits/stdc++.h>
using namespace std;

// Definition of TreeNode
struct TreeNode {
 int val;
 TreeNode* left;
 TreeNode* right;

 TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

class Solution {
public:
 TreeNode* insertIntoBST(TreeNode* root, int key) {
 // If tree is empty, new node becomes root
 if (root == nullptr) {
 return new TreeNode(key);
 }

 TreeNode* curr = root;
```

```

 while (true) {
 if (key < curr->val) {
 if (curr->left != nullptr) {
 curr = curr->left;
 } else {
 curr->left = new TreeNode(key);
 break;
 }
 } else {
 if (curr->right != nullptr) {
 curr = curr->right;
 } else {
 curr->right = new TreeNode(key);
 break;
 }
 }
 }
 return root;
 }
};

// Inorder traversal to verify BST
void printInOrder(TreeNode* root) {
 if (!root) return;
 printInOrder(root->left);
 cout << root->val << " ";
 printInOrder(root->right);
}

int main() {
 TreeNode* root = new TreeNode(4);
 root->left = new TreeNode(2);
 root->right = new TreeNode(6);
 root->left->left = new TreeNode(1);
 root->left->right = new TreeNode(3);
 root->right->left = new TreeNode(5);
 root->right->right = new TreeNode(7);
}

```

```

Solution sol;
root = sol.insertIntoBST(root, 8);

cout << "BST after insertion (Inorder): ";
printInOrder(root);
cout << endl;

return 0;
}

```

---

## Complexity Analysis

- **Time Complexity:**  $O(H)$   
Where H is the height of the BST.
  - Balanced BST  $\rightarrow O(\log N)$
  - Skewed BST  $\rightarrow O(N)$
- **Space Complexity:**  $O(1)$   
Iterative approach uses constant extra space.

# 7. Delete a Node in Binary Search Tree

Given a **Binary Search Tree (BST)** and a key value, delete the node having that key from the BST.

After deletion, the BST property must remain valid.

---

## Question Explanation

In a Binary Search Tree:

- Left subtree contains values **smaller** than the node.
- Right subtree contains values **greater** than the node.

While deleting a node, there are **three possible cases**:

1. **Node is a leaf (no children)**  
→ Simply remove the node.
  2. **Node has one child**  
→ Replace the node with its child.
  3. **Node has two children**  
→ Replace the node's value with its **inorder successor** (smallest value in right subtree), (Inorder successor **ya** inorder predecessor — dono valid hote hain)  
→ Then delete that successor node.
- 

## Example Explanation

BST: 5 3 6 2 4 -1 7

Delete key = 3

- Node 3 has two children (2 and 4)
- Inorder successor of 3 is 4
- Replace 3 with 4
- Delete original node 4

Final BST remains valid.

---

## Algorithm

1. If root is NULL, return NULL.
  2. If key < root value, delete from left subtree.
  3. If key > root value, delete from right subtree.
  4. If key == root value:
    - o If left child is NULL, return right child.
    - o If right child is NULL, return left child.
    - o If both children exist:
      - Find inorder successor (minimum in right subtree).
      - Replace root value with successor value.
      - Delete successor from right subtree.
  5. Return the root.
- 

## Code (C++)

```
#include <bits/stdc++.h>
using namespace std;

// Definition of TreeNode
struct TreeNode {
 int val;
 TreeNode* left;
 TreeNode* right;

 TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

class Solution {
public:
 // Function to find minimum value node in BST
```

```

TreeNode* findMin(TreeNode* root) {
 while (root->left != nullptr) {
 root = root->left;
 }
 return root;
}

// Function to delete a node from BST
TreeNode* deleteNode(TreeNode* root, int key) {
 // Base case
 if (root == nullptr) return nullptr;

 // Traverse left subtree
 if (key < root->val) {
 root->left = deleteNode(root->left, key);
 }
 // Traverse right subtree
 else if (key > root->val) {
 root->right = deleteNode(root->right, key);
 }
 // Node found
 else {
 // Case 1: No left child
 if (root->left == nullptr) {
 TreeNode* temp = root->right;
 delete root;
 return temp;
 }
 // Case 2: No right child
 else if (root->right == nullptr) {
 TreeNode* temp = root->left;
 delete root;
 return temp;
 }
 // Case 3: Two children
 TreeNode* temp = findMin(root->right);
 root->val = temp->val;
 root->right = deleteNode(root->right, temp->val);
 }
}

```

```

 }
 return root;
 }
};

// Inorder traversal to verify BST
void printInOrder(TreeNode* root) {
 if (!root) return;
 printInOrder(root->left);
 cout << root->val << " ";
 printInOrder(root->right);
}

int main() {
 TreeNode* root = new TreeNode(5);
 root->left = new TreeNode(3);
 root->right = new TreeNode(6);
 root->left->left = new TreeNode(2);
 root->left->right = new TreeNode(4);
 root->right->right = new TreeNode(7);

 Solution sol;
 root = sol.deleteNode(root, 3);

 cout << "BST after deletion (Inorder): ";
 printInOrder(root);
 cout << endl;

 return 0;
}

```

---

## Complexity Analysis

- **Time Complexity:**  $O(H)$   
Where H is the height of the BST.

- Balanced BST  $\rightarrow O(\log N)$
- Skewed BST  $\rightarrow O(N)$
- **Space Complexity:**  $O(H)$   
Due to recursion stack.
  - Balanced BST  $\rightarrow O(\log N)$
  - Skewed BST  $\rightarrow O(N)$

## 8. Kth Largest / Smallest Element in Binary Search Tree

Given the root of a **Binary Search Tree (BST)** and an integer k, return:

- the **kth smallest** element
- the **kth largest** element

The result should be returned as an array where:

- first value = kth smallest
- second value = kth largest

The indexing is **1-based**.

---

### Question Explanation

In a Binary Search Tree:

- **Inorder traversal** gives nodes in **sorted (ascending) order**
- **Reverse inorder traversal** gives nodes in **descending order**

Using this property, we can find the kth smallest and kth largest elements.

---

## Example Explanation

BST: [3, 1, 4, null, 2], k = 1

Inorder traversal → [1, 2, 3, 4]

- 1st smallest = 1
- 1st largest = 4

Output → [1, 4]

---

## Approach 1: Brute Force

### Algorithm

1. Create a vector to store node values.
2. Perform **inorder traversal** of the BST and store all values.
3. Since inorder traversal is sorted:
  - kth smallest = values[k-1]
  - kth largest = values[n-k]
4. Return both values.

---

## Code (C++)

```
#include <bits/stdc++.h>
using namespace std;

// Tree node structure
struct TreeNode {
 int data;
 TreeNode *left;
 TreeNode *right;
 TreeNode(int val) : data(val), left(nullptr), right(nullptr) {}
};

class Solution {
public:
 void inorder(TreeNode* root, vector<int>& vals) {
 if (!root) return;
 inorder(root->left, vals);
 vals.push_back(root->data);
 inorder(root->right, vals);
 }

 vector<int> kLargestSmallest(TreeNode* root, int k) {
 vector<int> vals;
 inorder(root, vals);
 int kthSmallest = vals[k - 1];
 int kthLargest = vals[vals.size() - k];
 return {kthSmallest, kthLargest};
 }
};

int main() {
 TreeNode* root = new TreeNode(3);
 root->left = new TreeNode(1);
 root->left->right = new TreeNode(2);
 root->right = new TreeNode(4);
```

```

Solution sol;
int k = 1;
vector<int> ans = sol.kLargesSmall(root, k);

cout << "[" << ans[0] << ", " << ans[1] << "]";
return 0;
}

```

---

## Complexity Analysis

- **Time Complexity:**  $O(N)$   
Every node is visited once during inorder traversal.
  - **Space Complexity:**  $O(N)$   
Extra space is used to store all node values.
- 

## Approach 2: Optimal (Without Extra Array)

### Algorithm

#### Kth Smallest

1. Perform **inorder traversal**.
2. Maintain a counter.
3. When counter reaches  $k$ , store the value.

#### Kth Largest

1. Perform **reverse inorder traversal** (Right → Root → Left).
2. Maintain a counter.
3. When counter reaches  $k$ , store the value.

Return both results.

---

### Code (C++)

```
#include <bits/stdc++.h>
using namespace std;

// Tree node structure
struct TreeNode {
 int data;
 TreeNode *left;
 TreeNode *right;
 TreeNode(int val) : data(val), left(nullptr), right(nullptr) {}
};

class Solution {
public:
 int k, ans;

 void inorder(TreeNode* root) {
 if (!root || ans != -1) return;
 inorder(root->left);
 if (--k == 0) {
 ans = root->data;
 return;
 }
 inorder(root->right);
 }

 void reverseInorder(TreeNode* root) {
 if (!root || ans != -1) return;
 reverseInorder(root->right);
 if (--k == 0) {
 ans = root->data;
 return;
 }
 reverseInorder(root->left);
 }
}
```

```

vector<int> kLargesSmall(TreeNode* root, int kVal) {
 k = kVal;
 ans = -1;
 inorder(root);
 int kthSmallest = ans;

 k = kVal;
 ans = -1;
 reverseInorder(root);
 int kthLargest = ans;

 return {kthSmallest, kthLargest};
}

int main() {
 TreeNode* root = new TreeNode(3);
 root->left = new TreeNode(1);
 root->left->right = new TreeNode(2);
 root->right = new TreeNode(4);

 Solution sol;
 int k = 1;
 vector<int> ans = sol.kLargesSmall(root, k);

 cout << "[" << ans[0] << ", " << ans[1] << "]";
 return 0;
}

```

---

## Complexity Analysis

- **Time Complexity:**  $O(N)$   
In worst case, traversal may visit all nodes.
- **Space Complexity:**  $O(H)$   
Due to recursion stack, where  $H$  is the height of the BST.

- Balanced BST  $\rightarrow O(\log N)$
- Skewed BST  $\rightarrow O(N)$

## 9. Check if a Tree is a Binary Search Tree (BST) or just a Binary Tree (BT)

### Problem Explanation

You are given the root of a binary tree.

Your task is to **check whether the given tree follows the Binary Search Tree (BST) property or not.**

A **Binary Search Tree (BST)** has the following property:

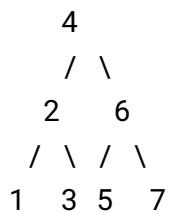
- All values in the **left subtree** of a node are **strictly smaller** than the node's value.
- All values in the **right subtree** of a node are **strictly greater** than the node's value.
- This property must hold **for every node** in the tree.

If the tree satisfies this property, return **true**, otherwise return **false**.

---

### Example Explanation

Example 1 (BST):

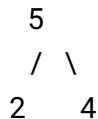


Inorder traversal → 1 2 3 4 5 6 7 (sorted)

So, this tree **is a BST**.

---

Example 2 (Not a BST):



Here, 4 is on the right of 5 but  $4 < 5$ , which violates BST rules.

So, this tree **is NOT a BST**.

---

## Approach 1: Inorder Traversal Check

### Algorithm

This approach is based on a key property of BST:

- **Inorder traversal of a BST always produces a strictly increasing sequence**

Steps:

1. Perform an inorder traversal of the binary tree.
  2. Store the traversal values in a vector.
  3. Check if the vector is **strictly sorted**.
  4. If yes → tree is a BST.
  5. Otherwise → tree is not a BST.
- 

### Code (C++)

```
#include <bits/stdc++.h>
using namespace std;
```

```

// Tree node structure
struct TreeNode {
 int val;
 TreeNode* left;
 TreeNode* right;
 TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

class Solution {
public:
 void inorder(TreeNode* root, vector<int>& arr) {
 if (!root) return;
 inorder(root->left, arr);
 arr.push_back(root->val);
 inorder(root->right, arr);
 }

 bool isBST(TreeNode* root) {
 vector<int> arr;
 inorder(root, arr);

 for (int i = 1; i < arr.size(); i++) {
 if (arr[i] <= arr[i - 1]) {
 return false;
 }
 }
 return true;
 }
};

int main() {
 TreeNode* root = new TreeNode(4);
 root->left = new TreeNode(2);
 root->right = new TreeNode(6);
 root->left->left = new TreeNode(1);
 root->left->right = new TreeNode(3);
}

```

```

Solution sol;
if (sol.isBST(root))
 cout << "Tree is a BST";
else
 cout << "Tree is NOT a BST";

return 0;
}

```

---

## Complexity Analysis

- **Time Complexity:**  $O(N)$   
Each node is visited once during inorder traversal.
  - **Space Complexity:**  $O(N)$   
Extra space is used to store inorder traversal values.
- 

## Approach 2: Range (Min–Max) Validation (Optimal)

### Algorithm

Instead of storing values, we validate each node using a **range**.

Steps:

1. For each node, maintain a valid range (`min, max`).
2. Initially, range is  $(-\infty, +\infty)$ .
3. For every node:
  - Node value must lie strictly between `min` and `max`.
4. For left child:
  - New range  $\rightarrow (\min, \text{current node value})$

5. For right child:

- o New range → (current node value, max)

6. If any node violates the range, return false.

---

## Code (C++)

```
#include <bits/stdc++.h>
using namespace std;

// Tree node structure
struct TreeNode {
 int val;
 TreeNode* left;
 TreeNode* right;
 TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

class Solution {
public:
 bool check(TreeNode* root, long minVal, long maxVal) {
 if (!root) return true;

 if (root->val <= minVal || root->val >= maxVal)
 return false;

 return check(root->left, minVal, root->val) &&
 check(root->right, root->val, maxVal);
 }

 bool isBST(TreeNode* root) {
 return check(root, LONG_MIN, LONG_MAX);
 }
};

int main() {
```

```

TreeNode* root = new TreeNode(5);
root->left = new TreeNode(2);
root->right = new TreeNode(4);

Solution sol;
if (sol.isBST(root))
 cout << "Tree is a BST";
else
 cout << "Tree is NOT a BST";

return 0;
}

```

---

## Complexity Analysis

- **Time Complexity:**  $O(N)$   
Each node is checked once.
- **Space Complexity:**  $O(H)$   
Due to recursion stack, where  $H$  is tree height.
  - Balanced tree  $\rightarrow O(\log N)$
  - Skewed tree  $\rightarrow O(N)$

# 10. Lowest Common Ancestor in a Binary Search Tree

You are given the **root of a Binary Search Tree (BST)** and two nodes  $n_1$  and  $n_2$  that are guaranteed to be present in the tree.

Your task is to find the **Lowest Common Ancestor (LCA)** of these two nodes.

The **Lowest Common Ancestor** of two nodes is the **deepest node (farthest from the root)** that has **both nodes as its descendants**.

A node can also be a descendant of itself.

Because the tree is a **Binary Search Tree**, it follows:

- Left subtree values < root
- Right subtree values > root

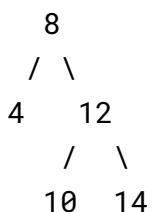
We will use this property to optimize the solution.

---

## Example Explanation

Example 1:

BST :



n1 = 4, n2 = 14

- Node 8 is the first node where one value lies on the left and the other on the right.
- Hence, **LCA = 8**

n1 = 10, n2 = 14

- Both nodes lie in the **right subtree of 8**
- At node 12, one node lies on the left and the other on the right
- Hence, **LCA = 12**

# Approach 1: Naive Approach (Using Binary Tree Method)

## Algorithm

This approach does **not use BST properties** and works for a normal binary tree.

1. If the current node is NULL, return NULL.
  2. If the current node matches n1 or n2, return the current node.
  3. Recursively find LCA in left subtree.
  4. Recursively find LCA in right subtree.
  5. If both left and right recursive calls return non-null values, current node is the LCA.
  6. Otherwise, return the non-null value.
- 

## Code (C++)

```
#include <bits/stdc++.h>
using namespace std;

struct Node {
 int data;
 Node* left;
 Node* right;
 Node(int x) : data(x), left(nullptr), right(nullptr) {}
};

class Solution {
public:
 Node* LCA(Node* root, Node* n1, Node* n2) {
 if (root == nullptr) return nullptr;

 if (root == n1 || root == n2)
 return root;

 Node* leftLCA = LCA(root->left, n1, n2);
 Node* rightLCA = LCA(root->right, n1, n2);

 if (leftLCA != nullptr && rightLCA != nullptr)
 return root;
 else
 return leftLCA ? leftLCA : rightLCA;
 }
};
```

```

 Node* rightLCA = LCA(root->right, n1, n2);

 if (leftLCA && rightLCA)
 return root;

 return (leftLCA != nullptr) ? leftLCA : rightLCA;
 }
};

```

---

## Complexity Analysis

- **Time Complexity:**  $O(N)$   
In the worst case, all nodes are visited.
  - **Space Complexity:**  $O(N)$   
Due to recursion stack in a skewed tree.
- 

## Approach 2: Using BST Properties (Recursive Approach)

### Algorithm

This approach uses the **ordering property of BST**.

1. If root is NULL, return NULL.
  2. If both n1 and n2 are smaller than root, LCA lies in the **left subtree**.
  3. If both n1 and n2 are greater than root, LCA lies in the **right subtree**.
  4. Otherwise, the current root is the **LCA**.
- 

```

#include <bits/stdc++.h>
using namespace std;

struct Node {

```

```

int data;
Node* left;
Node* right;
Node(int x) : data(x), left(nullptr), right(nullptr) {}
};

class Solution {
public:
 Node* LCA(Node* root, Node* n1, Node* n2) {
 if (root == nullptr)
 return nullptr;

 if (root->data > n1->data && root->data > n2->data)
 return LCA(root->left, n1, n2);

 if (root->data < n1->data && root->data < n2->data)
 return LCA(root->right, n1, n2);

 return root;
 }
};

```

- **Time Complexity:**  $O(H)$   
We traverse only one path of the BST, where  $H$  is the height.
  - **Space Complexity:**  $O(H)$   
Due to recursion stack.
- 

## Approach 3: Using BST Properties (Iterative Method)

### Algorithm

This is an optimized version of Approach 2 where recursion is removed.

1. Start from the root.

2. While root is not NULL:

- If both nodes are smaller than root, move left.
- If both nodes are greater than root, move right.
- Otherwise, current node is the LCA.

3. Return the current node.

---

### Code (C++)

```
#include <bits/stdc++.h>
using namespace std;

struct Node {
 int data;
 Node* left;
 Node* right;
 Node(int x) : data(x), left(nullptr), right(nullptr) {}
};

class Solution {
public:
 Node* LCA(Node* root, Node* n1, Node* n2) {
 while (root != nullptr) {
 if (root->data > n1->data && root->data > n2->data)
 root = root->left;
 else if (root->data < n1->data && root->data < n2->data)
 root = root->right;
 else
 return root;
 }
 return nullptr;
 }
};
```

---

## Complexity Analysis

- **Time Complexity:**  $O(H)$   
Only one downward traversal of the BST.
- **Space Complexity:**  $O(1)$   
No recursion or extra data structures used.

# 11. Construct BST from Preorder Traversal

## Question Explanation

You are given the **preorder traversal** of a **Binary Search Tree (BST)**.  
Your task is to **construct the BST** from this preorder traversal.

In a BST:

- All values in the left subtree are **smaller** than the root.
- All values in the right subtree are **greater** than the root.
- Preorder traversal follows the order: **Root → Left → Right**.

The constructed tree must be such that **its preorder traversal is exactly the given array**.

---

## Example Explanation

Input:

`pre = [10, 5, 1, 7, 40, 50]`

- 10 is the first element → root

- Values smaller than 10 (5, 1, 7) go to left subtree
- Values greater than 10 (40, 50) go to right subtree
- Same logic is applied recursively

Inorder traversal of the constructed BST will be:

1 5 7 10 40 50

---

## Approach 1: Naive – One by One Insert

### Algorithm

1. Create an empty BST.
2. Traverse the preorder array from left to right.
3. Insert each element into the BST using normal BST insertion rules.
4. After all insertions, the BST is constructed.

This approach directly uses the property that inserting nodes one by one into a BST will form a valid BST.

---

### Code

```
#include <bits/stdc++.h>
using namespace std;

struct Node {
 int key;
 Node* left;
 Node* right;
 Node(int val) {
 key = val;
 left = nullptr;
 right = nullptr;
 }
}
```

```

 }
};

// Standard BST insert function
Node* insertBST(Node* root, int key) {
 if (root == nullptr)
 return new Node(key);

 if (key < root->key)
 root->left = insertBST(root->left, key);
 else if (key > root->key)
 root->right = insertBST(root->right, key);

 return root;
}

// Construct BST from preorder traversal
Node* construct(vector<int>& pre) {
 Node* root = nullptr;
 for (int key : pre) {
 root = insertBST(root, key);
 }
 return root;
}

// Inorder traversal
void inorder(Node* root) {
 if (root == nullptr) return;
 inorder(root->left);
 cout << root->key << " ";
 inorder(root->right);
}

int main() {
 vector<int> pre = {10, 5, 1, 7, 40, 50};
 Node* root = construct(pre);
 inorder(root);
 return 0;
}

```

}

---

## Complexity Analysis

- **Time Complexity:**  $O(N^2)$   
Each insertion can take  $O(N)$  in the worst case (skewed BST).
  - **Space Complexity:**  $O(N)$   
Due to recursion stack in worst case.
- 

## Approach 2: Better – Find First Greater than Root

### Algorithm

1. The first element of preorder is always the root.
  2. Find the **first element greater than root**.
  3. Elements between (root index + 1) and (i - 1) form the **left subtree**.
  4. Elements from i to end form the **right subtree**.
  5. Recursively apply the same logic to left and right subarrays.
- 

### Code

```
#include <bits/stdc++.h>
using namespace std;

struct Node {
 int data;
 Node* left;
 Node* right;
 Node(int x) {
```

```

 data = x;
 left = nullptr;
 right = nullptr;
 }
};

Node* constructUtil(vector<int>& pre, int low, int high) {
 if (low > high)
 return nullptr;

 Node* root = new Node(pre[low]);
 if (low == high)
 return root;

 int i;
 for (i = low + 1; i <= high; i++) {
 if (pre[i] > root->data)
 break;
 }

 root->left = constructUtil(pre, low + 1, i - 1);
 root->right = constructUtil(pre, i, high);

 return root;
}

Node* construct(vector<int>& pre) {
 return constructUtil(pre, 0, pre.size() - 1);
}

void inorder(Node* root) {
 if (!root) return;
 inorder(root->left);
 cout << root->data << " ";
 inorder(root->right);
}

int main() {

```

```

vector<int> pre = {10, 5, 1, 7, 40, 50};
Node* root = construct(pre);
inorder(root);
return 0;
}

```

---

## Complexity Analysis

- **Time Complexity:**  $O(N^2)$   
For each node, we scan part of the array to find the split index.
  - **Space Complexity:**  $O(N)$   
Due to recursion stack.
- 

## Approach 3: Efficient – Pass Range in Recursion

### Algorithm

This approach uses the **valid range of values** for each subtree.

1. Maintain an index `idx` for preorder traversal.
2. Start with range  $(-\infty, +\infty)$ .
3. If current value is not in range, return NULL.
4. Create node using current value and increment index.
5. Recur for:
  - Left subtree with range  $(\min, \text{root}\rightarrow\text{data})$
  - Right subtree with range  $(\text{root}\rightarrow\text{data}, \max)$
6. Continue until all elements are used.

---

## Code

```
#include <bits/stdc++.h>
using namespace std;

struct Node {
 int data;
 Node* left;
 Node* right;
 Node(int val) {
 data = val;
 left = nullptr;
 right = nullptr;
 }
};

Node* constructUtil(vector<int>& pre, int& idx, int minVal, int maxVal) {
 if (idx >= pre.size())
 return nullptr;

 int key = pre[idx];
 if (key <= minVal || key >= maxVal)
 return nullptr;

 Node* root = new Node(key);
 idx++;

 root->left = constructUtil(pre, idx, minVal, key);
 root->right = constructUtil(pre, idx, key, maxVal);

 return root;
}

Node* constructTree(vector<int>& pre) {
 int idx = 0;
 return constructUtil(pre, idx, INT_MIN, INT_MAX);
```

```

}

void inorder(Node* root) {
 if (!root) return;
 inorder(root->left);
 cout << root->data << " ";
 inorder(root->right);
}

int main() {
 vector<int> pre = {10, 5, 1, 7, 40, 50};
 Node* root = constructTree(pre);
 inorder(root);
 return 0;
}

```

---

## Complexity Analysis

- **Time Complexity:**  $O(N)$   
Each element is processed exactly once.
- **Space Complexity:**  $O(N)$   
Recursion stack in worst case (skewed tree).

# 12. Inorder Successor / Predecessor in Binary Search Tree

You are given a **Binary Search Tree (BST)** and a **key value** representing a node in the tree.  
Your task is to find:

- **Inorder Predecessor:**  
The node that appears **just before** the given node in inorder traversal.

- **Inorder Successor:**

The node that appears **just after** the given node in inorder traversal.

If no predecessor or successor exists, return `nullptr`.

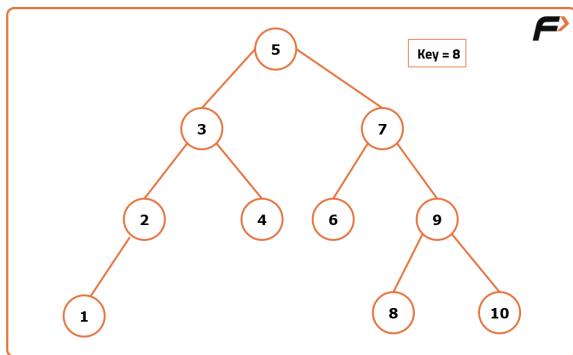
Inorder traversal of BST always gives a **sorted sequence**.

---

## Example Explanation

Input BST (inorder):

1 2 3 4 5 6 7 8 9 10



Key = 8

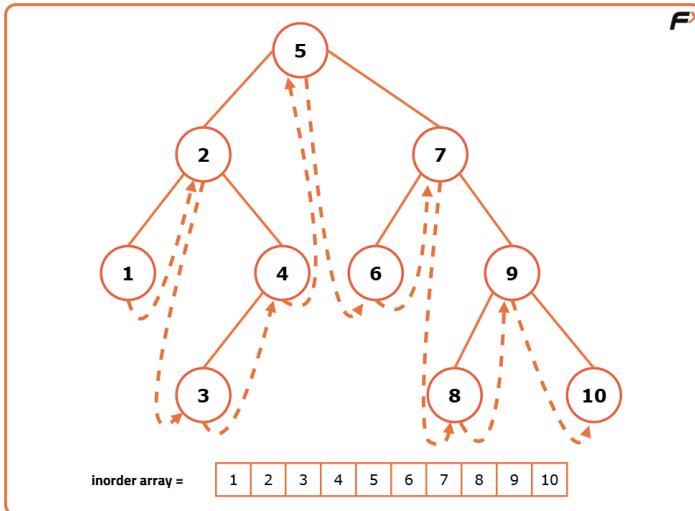
- Predecessor = 7
  - Successor = 9
- 

## Approach 1: Brute Force

### Algorithm

1. Perform **inorder traversal** of the BST and store all values in a vector.
2. Since inorder traversal of BST is sorted:

- Find the index of key.
  - Predecessor = element at index - 1 (if exists).
  - Successor = element at index + 1 (if exists).
3. If index is at boundary, return nullptr accordingly.



## Code

```
#include <bits/stdc++.h>
using namespace std;

struct TreeNode{
 int val;
 TreeNode* left;
 TreeNode* right;
 TreeNode(int x): val(x), left(nullptr), right(nullptr) {}
};

class Solution{
public:
 void inorder(TreeNode* root, vector<int>& arr){
 if(!root) return;
 inorder(root->left, arr);
 arr.push_back(root->val);
 inorder(root->right, arr);
 }
};
```

```

 inorder(root->right, arr);
 }

pair<int,int> findPreSuc(TreeNode* root, int key){
 vector<int> arr;
 inorder(root, arr);

 int pre = -1, suc = -1;
 for(int i=0;i<arr.size();i++){
 if(arr[i] == key){
 if(i > 0) pre = arr[i-1];
 if(i < arr.size()-1) suc = arr[i+1];
 break;
 }
 }
 return {pre, suc};
}

```

---

## Complexity Analysis

- **Time Complexity:**  $O(N)$   
Entire tree is traversed.
  - **Space Complexity:**  $O(N)$   
Inorder array storage.
- 

## Approach 2: Better (Using BST Property – Iterative Traversal)

### Algorithm

#### For Successor

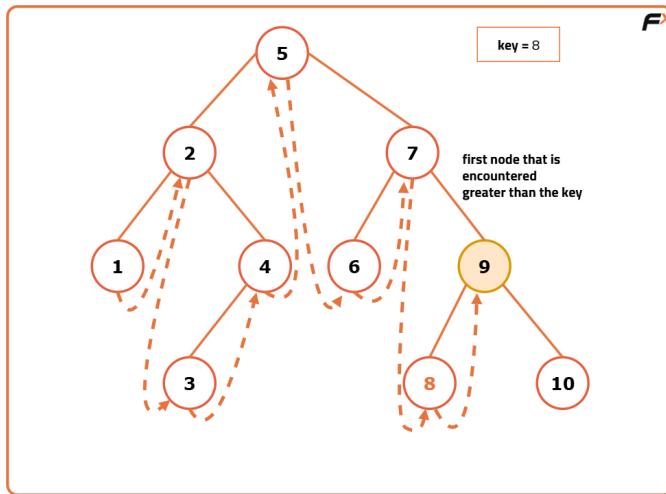
- Start from root.

- If current value > key:
  - Update successor.
  - Move left.
- Else move right.

## For Predecessor

- Start from root.
- If current value < key:
  - Update predecessor.
  - Move right.
- Else move left.

Traversal continues until root becomes null.



## Code

```
#include <bits/stdc++.h>
using namespace std;

struct TreeNode{
 int val;
 TreeNode* left;
 TreeNode* right;
 TreeNode(int x): val(x), left(nullptr), right(nullptr) {}
};

class Solution{
```

```

public:
 pair<TreeNode*, TreeNode*> findPreSuc(TreeNode* root, int key){
 TreeNode* pre = nullptr;
 TreeNode* suc = nullptr;
 TreeNode* cur = root;

 // find predecessor
 while(cur){
 if(cur->val < key){
 pre = cur;
 cur = cur->right;
 } else {
 cur = cur->left;
 }
 }

 cur = root;
 // find successor
 while(cur){
 if(cur->val > key){
 suc = cur;
 cur = cur->left;
 } else {
 cur = cur->right;
 }
 }

 return {pre, suc};
 }
};

```

## Complexity Analysis

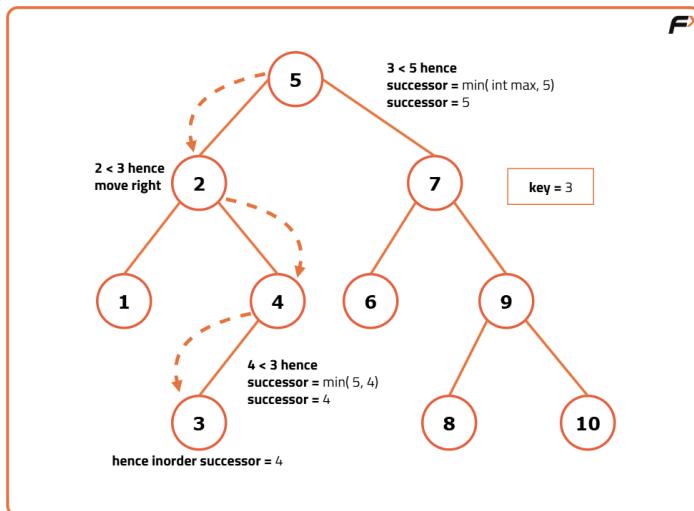
- **Time Complexity:**  $O(H)$   
Only height of BST is traversed.

- **Space Complexity:**  $O(1)$   
No extra data structures used.
- 

## Approach 3: Optimal (BST Property – Single Traversal)

### Algorithm

1. Traverse BST once.
2. If node value equals key:
  - o Stop.
3. If node value > key:
  - o Update successor.
  - o Move left.
4. If node value < key:
  - o Update predecessor.
  - o Move right.
5. Continue until null.



---

### Code

```
#include <bits/stdc++.h>
using namespace std;
```

```

struct TreeNode{
 int val;
 TreeNode* left;
 TreeNode* right;
 TreeNode(int x): val(x), left(nullptr), right(nullptr) {}
};

class Solution{
public:
 pair<TreeNode*,TreeNode*> findPreSuc(TreeNode* root, int key){
 TreeNode* pre = nullptr;
 TreeNode* suc = nullptr;

 while(root){
 if(root->val == key){
 break;
 }
 else if(root->val > key){
 suc = root;
 root = root->left;
 }
 else{
 pre = root;
 root = root->right;
 }
 }
 return {pre, suc};
 }
};

```

---

- **Time Complexity:**  $O(H)$   
Only one downward traversal.
- **Space Complexity:**  $O(1)$   
Constant extra space.

# 13. Merge 2 BSTs

You are given **two Binary Search Trees (BSTs)**.

Your task is to return **all elements from both BSTs in sorted order**.

Remember:

- Inorder traversal of a BST gives elements in **sorted order**.
  - Duplicate values are allowed and should be included.
- 

## Example Explanation

### Input

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

BST 2: [2, 1, 3, null, null, null, 7, 6]

After collecting and sorting all elements:

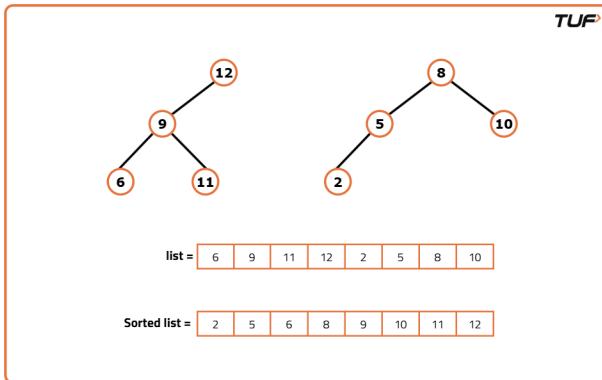
[1, 2, 2, 3, 3, 4, 5, 6, 6, 7]

---

## Approach 1: Brute Force

1. Traverse the first BST and store all its elements in a list.
2. Traverse the second BST and store all its elements in the same list.
3. Sort the combined list.
4. Return the sorted list.

This approach ignores the BST property during traversal and sorts later.



## Code

```
#include <bits/stdc++.h>
using namespace std;

// Structure for BST node
struct Node {
 int data;
 Node* left;
 Node* right;
 Node(int val) : data(val), left(NULL), right(NULL) {}
};

class Solution {
public:
 void traverse(Node* root, vector<int>& arr) {
 if (!root) return;
 traverse(root->left, arr);
 arr.push_back(root->data);
 traverse(root->right, arr);
 }

 vector<int> mergeBSTs(Node* root1, Node* root2) {
 vector<int> arr;
 traverse(root1, arr);
 traverse(root2, arr);
 sort(arr.begin(), arr.end());
 return arr;
 }
}
```

```

};

int main() {
 Node* root1 = new Node(2);
 root1->left = new Node(1);
 root1->right = new Node(4);

 Node* root2 = new Node(3);
 root2->left = new Node(0);
 root2->right = new Node(5);

 Solution sol;
 vector<int> result = sol.mergeBSTs(root1, root2);

 for (int x : result) cout << x << " ";
 return 0;
}

```

## Complexity Analysis

- **Time Complexity:**  $O((n + m) \log(n + m))$   
Sorting dominates the time.
- **Space Complexity:**  $O(n + m)$   
Extra list to store all elements.

## Approach 2: Optimal (Using Inorder Traversal + Merge)

1. Perform **inorder traversal** on the first BST and store elements in arr1.
2. Perform **inorder traversal** on the second BST and store elements in arr2.
  - Both arrays will already be sorted.
3. Merge the two sorted arrays using the **two-pointer technique**.
4. Return the merged sorted array.

This approach fully uses BST properties.

---

## Code

```
#include <bits/stdc++.h>
using namespace std;

// Structure for BST node
struct Node {
 int data;
 Node* left;
 Node* right;
 Node(int val) : data(val), left(NULL), right(NULL) {}
};

class Solution {
public:
 void inorder(Node* root, vector<int>& arr) {
 if (!root) return;
 inorder(root->left, arr);
 arr.push_back(root->data);
 inorder(root->right, arr);
 }

 vector<int> mergeArrays(vector<int>& a, vector<int>& b) {
 vector<int> res;
 int i = 0, j = 0;

 while (i < a.size() && j < b.size()) {
 if (a[i] < b[j]) res.push_back(a[i++]);
 else res.push_back(b[j++]);
 }

 while (i < a.size()) res.push_back(a[i++]);
 while (j < b.size()) res.push_back(b[j++]);

 return res;
 }
}
```

```

vector<int> mergeBSTs(Node* root1, Node* root2) {
 vector<int> arr1, arr2;
 inorder(root1, arr1);
 inorder(root2, arr2);
 return mergeArrays(arr1, arr2);
}

int main() {
 Node* root1 = new Node(3);
 root1->left = new Node(1);
 root1->right = new Node(5);

 Node* root2 = new Node(4);
 root2->left = new Node(2);
 root2->right = new Node(6);

 Solution sol;
 vector<int> result = sol.mergeBSTs(root1, root2);

 for (int x : result) cout << x << " ";
 return 0;
}

```

---

## Complexity Analysis

- **Time Complexity:**  $O(n + m)$   
Each node is visited once, merge is linear.
- **Space Complexity:**  $O(n + m)$   
Two inorder arrays and merged result.

# 14. Two Sum In BST | Check if there exists a pair with Sum K

You are given the **root of a Binary Search Tree (BST)** and an integer **K**.

You need to check whether there exist **two different nodes** in the BST such that the **sum of their values is exactly K**.

Return **true** if such a pair exists, otherwise return **false**.

Because this is a BST, its **inorder traversal gives a sorted sequence**, which can be used to optimize the solution.

---

## Example Explanation

### Example 1

BST: 5 3 6 2 4 -1 7, K = 9

Inorder traversal  $\rightarrow [2, 3, 4, 5, 6, 7]$

Possible pair:  $2 + 7 = 9$

So, output is **true**.

---

## Approach 1: Brute Force (Inorder + Two Pointers)

### Algorithm

1. Perform an **inorder traversal** of the BST and store all node values in a vector.
2. Since inorder traversal of a BST is sorted, apply the **two pointer technique**:
  - o One pointer at the start.
  - o One pointer at the end.
3. While left pointer is smaller than right pointer:

- If  $\text{arr}[\text{left}] + \text{arr}[\text{right}] == K$ , return true.
  - If sum is smaller than K, move left pointer forward.
  - If sum is greater than K, move right pointer backward.
4. If no such pair is found, return false.
- 

## Code

```
#include <bits/stdc++.h>
using namespace std;

struct TreeNode {
 int val;
 TreeNode* left;
 TreeNode* right;
 TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

class Solution {
public:
 bool findTarget(TreeNode* root, int k) {
 vector<int> inorder;
 inorderTraversal(root, inorder);

 int l = 0, r = inorder.size() - 1;
 while (l < r) {
 int sum = inorder[l] + inorder[r];
 if (sum == k) return true;
 else if (sum < k) l++;
 else r--;
 }
 return false;
 }

private:
 void inorderTraversal(TreeNode* root, vector<int>& inorder) {
```

```

 if (!root) return;
 inorderTraversal(root->left, inorder);
 inorder.push_back(root->val);
 inorderTraversal(root->right, inorder);
 }
};

int main() {
 TreeNode* root = new TreeNode(5);
 root->left = new TreeNode(3);
 root->right = new TreeNode(6);
 root->left->left = new TreeNode(2);
 root->left->right = new TreeNode(4);
 root->right->right = new TreeNode(7);

 Solution sol;
 int k = 9;
 cout << sol.findTarget(root, k);
 return 0;
}

```

---

## Complexity Analysis

- **Time Complexity:**  $O(N)$   
Inorder traversal takes  $O(N)$  and two-pointer scan also takes  $O(N)$ .
  - **Space Complexity:**  $O(N)$   
Extra space is used to store inorder traversal.
- 

## Approach 2: Optimal (BST Iterator – Two Pointer Without Extra Array)

### Algorithm

This approach avoids storing the entire inorder traversal.

1. Create two iterators:
  - o One inorder iterator (smallest to largest).
  - o One reverse inorder iterator (largest to smallest).
2. Initialize two values:
  - o  $i$  from the inorder iterator.
  - o  $j$  from the reverse inorder iterator.
3. While  $i < j$ :
  - o If  $i + j == K$ , return true.
  - o If sum is smaller than  $K$ , move inorder iterator forward.
  - o If sum is greater than  $K$ , move reverse inorder iterator backward.
4. If no pair is found, return false.

This simulates the two-pointer technique directly on the BST.

---

## Code

```
#include <bits/stdc++.h>
using namespace std;

struct TreeNode {
 int val;
 TreeNode* left;
 TreeNode* right;
 TreeNode(int x) : val(x), left(NULL), right(NULL) {}
};

// push leftmost path (normal inorder)
void pushLeft(TreeNode* node, stack<TreeNode*>& st) {
 while (node) {

```

```

 st.push(node);
 node = node->left;
 }
}

// push rightmost path (reverse inorder)
void pushRight(TreeNode* node, stack<TreeNode*>& st) {
 while (node) {
 st.push(node);
 node = node->right;
 }
}

bool findTarget(TreeNode* root, int k) {
 if (!root) return false;

 stack<TreeNode*> stL, stR;

 // initialize both stacks
 pushLeft(root, stL); // smallest
 pushRight(root, stR); // largest

 while (!stL.empty() && !stR.empty()) {
 TreeNode* leftNode = stL.top();
 TreeNode* rightNode = stR.top();

 // IMPORTANT: same node use nahi karna
 if (leftNode == rightNode) break;

 int sum = leftNode->val + rightNode->val;

 if (sum == k)
 return true;
 else if (sum < k) {
 // move left pointer forward
 stL.pop();
 pushLeft(leftNode->right, stL);
 } else {

```

```

 // move right pointer backward
 stR.pop();
 pushRight(rightNode->left, stR);
 //pushLeft() function khud NULL handle karta hai
 }
}

return false;
}

int main() {
 TreeNode* root = new TreeNode(7);
 root->left = new TreeNode(3);
 root->right = new TreeNode(15);
 root->right->left = new TreeNode(9);
 root->right->right = new TreeNode(20);

 int k = 22;
 cout << findTarget(root, k);
 return 0;
}

```

---

## Complexity Analysis

- **Time Complexity:**  $O(N)$   
Each node is visited at most once by the iterators.
- **Space Complexity:**  $O(H)$   
Stack space used by iterators, where  $H$  is the height of the BST.

# 15. Recover BST – Two Nodes are Swapped, Correct it

You are given a **Binary Search Tree (BST)** in which **exactly two nodes have been swapped by mistake**.

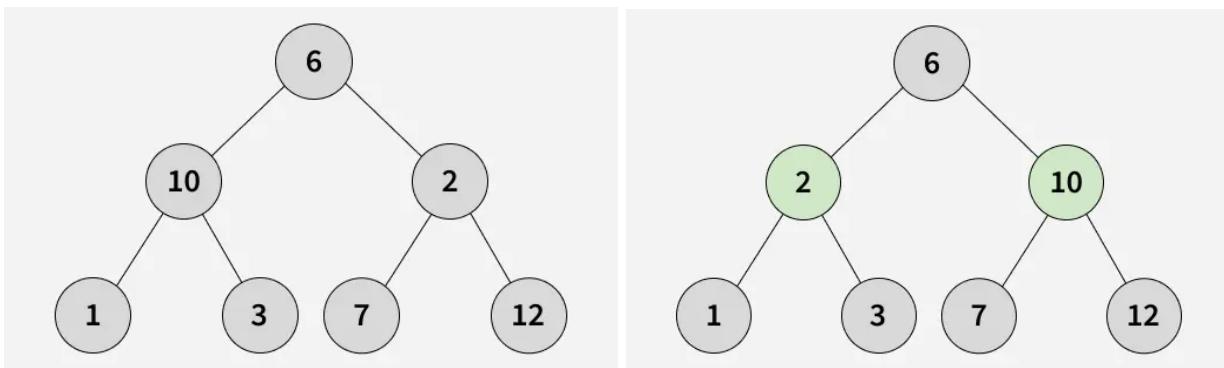
Because of this swap, the tree no longer follows the BST property.

Your task is to **restore the BST** by correcting the swapped nodes **without changing the tree structure**.

A valid BST has the property that its **inorder traversal is always sorted**.

---

## Example Explanation



If inorder traversal of a BST should be:

[1, 2, 3, 6, 7, 10, 12]

But due to swapping, it becomes:

[1, 6, 3, 2, 7, 10, 12]

Here, **two values break the sorted order**, which helps us identify the swapped nodes.

---

## Approach 1: Inorder Traversal + Sorting (Naive)

1. Perform **inorder traversal** of the BST and store all node values in a vector.

2. Since exactly two nodes are swapped, the inorder array will not be sorted.
3. **Sort the inorder array** to get the correct order.
4. Perform inorder traversal again and **replace node values** using the sorted array.
5. The BST property is restored.

This approach fixes values, not structure.

---

## Code

```
#include <bits/stdc++.h>
using namespace std;

struct Node {
 int data;
 Node* left;
 Node* right;
 Node(int x) : data(x), left(nullptr), right(nullptr) {}
};

void findInorder(Node* root, vector<int>& inorder) {
 if (!root) return;
 findInorder(root->left, inorder);
 inorder.push_back(root->data);
 findInorder(root->right, inorder);
}

void correctBSTUtil(Node* root, vector<int>& inorder, int& index) {
 if (!root) return;
 correctBSTUtil(root->left, inorder, index);
 root->data = inorder[index++];
 correctBSTUtil(root->right, inorder, index);
}

void correctBST(Node* root) {
 vector<int> inorder;
```

```
 findInorder(root, inorder);
 sort(inorder.begin(), inorder.end());
 int index = 0;
 correctBSTUtil(root, inorder, index);
}


```

---

## Complexity Analysis

- **Time Complexity:**  $O(N \log N)$   
Inorder traversal is  $O(N)$  and sorting is  $O(N \log N)$ .
  - **Space Complexity:**  $O(N)$   
Extra array is used to store inorder traversal.
- 

## Approach 2: One Inorder Traversal (Expected / Optimal)

### Algorithm

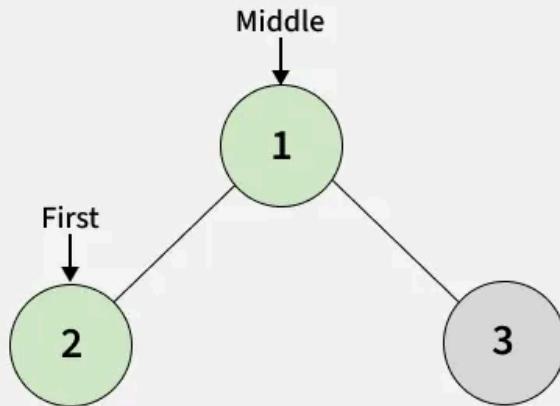
This approach uses the fact that inorder traversal of a BST should be sorted.

There are **two cases**:

1. **Swapped nodes are adjacent** in inorder traversal  
→ Only one violation is found.
2. **Swapped nodes are not adjacent**  
→ Two violations are found.

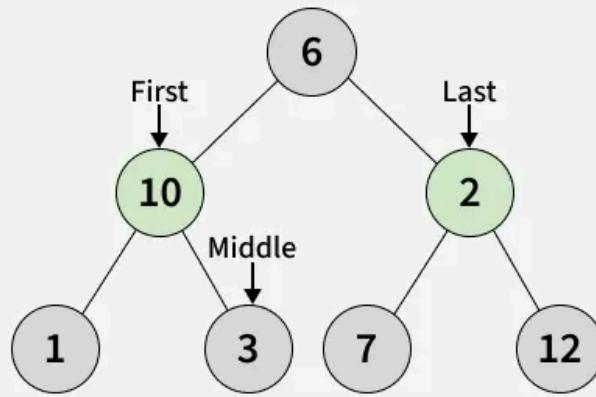
**01**  
Step

One violation means nodes are adjacent so swap first and middle pointer.



**02**  
Step

Two violations means nodes are not adjacent so swap first and last pointer.



Steps:

1. Do an inorder traversal while tracking:
  - `prev` → previously visited node
  - `first, middle, last` → nodes involved in violation
2. Whenever `prev->data > current->data`, a violation is found.
3. On first violation:
  - `first = prev`
  - `middle = current`

- 4. On second violation:
  - o `last = current`
- 5. After traversal:
  - o If `last` exists → swap `first` and `last`
  - o Else → swap `first` and `middle`

```
Node* first; // pehla galat node

Node* middle; // first violation ka current node

Node* last; // second violation ka current node

Node* prev; // inorder me previous node
```

Only one violation:

```
first = 3

middle = 2

last = NULL
```

Final pointers state

```
first = 10

middle = 3

last = 2
```

---

## Code

```
#include <bits/stdc++.h>
using namespace std;
```

```

struct Node {
 int data;
 Node* left;
 Node* right;
 Node(int x) : data(x), left(nullptr), right(nullptr) {}
};

void correctBSTUtil(Node* root, Node*& first, Node*& middle,
 Node*& last, Node*& prev) {
 if (!root) return;

 correctBSTUtil(root->left, first, middle, last, prev);

 if (prev && root->data < prev->data) {
 if (!first) {
 first = prev;
 middle = root;
 } else {
 last = root;
 }
 }

 prev = root;

 correctBSTUtil(root->right, first, middle, last, prev);
}

void correctBST(Node* root) {
 Node *first = nullptr, *middle = nullptr;
 Node *last = nullptr, *prev = nullptr;

 correctBSTUtil(root, first, middle, last, prev);

 if (first && last)
 swap(first->data, last->data);
 else if (first && middle)
 swap(first->data, middle->data);
}

```

---

## Complexity Analysis

- **Time Complexity:**  $O(N)$   
Single inorder traversal of the tree.
- **Space Complexity:**  $O(H)$   
Recursive stack space, where  $H$  is the height of the BST.

# 16. Largest BST in Binary Tree

Given the root of a **Binary Tree**, return the **size of the largest subtree** which is also a **Binary Search Tree (BST)**.

A subtree is considered a BST if:

- Left subtree contains values **less than** root
- Right subtree contains values **greater than** root
- Both left and right subtrees are BSTs

---

## Brute Force Approach

### Algorithm

1. For every node in the binary tree, consider it as a potential root of a BST.
2. Check whether the subtree rooted at that node is a valid BST using range constraints.
3. If valid, compute the size of that subtree.
4. Maintain a global maximum size.

5. Return the maximum size found.

This approach repeatedly validates subtrees, leading to inefficiency.

---

### Code (Brute Force)

```
#include <bits/stdc++.h>
using namespace std;

struct TreeNode {
 int data;
 TreeNode* left;
 TreeNode* right;
 TreeNode(int val) : data(val), left(nullptr), right(nullptr) {}
};

struct Info {
 int size;
 bool isBST;
 int minVal;
 int maxVal;
};

class Solution {
public:
 Info solve(TreeNode* node) {
 // Base case
 if (!node) {
 return {0, true, INT_MAX, INT_MIN};
 }

 Info left = solve(node->left);
 Info right = solve(node->right);

 // If current subtree is BST
 if (left.isBST && right.isBST &&
 left.maxVal < node->data &&
```

```

 node->data < right.minVal) {

 return {
 left.size + right.size + 1, // size
 true, // isBST
 min(node->data, left.minVal), // min
 max(node->data, right.maxVal) // max
 };
}

// If not BST
return {
 max(left.size, right.size),
 false,
 INT_MIN,
 INT_MAX
};
}

int largestBST(TreeNode* root) {
 return solve(root).size;
}
};

```

---

## Complexity Analysis

- **Time Complexity:**  $O(N^2)$   
Each node may validate an entire subtree.
  - **Space Complexity:**  $O(H)$   
Recursive stack where  $H$  is height of tree.
-

# Optimal Approach (Postorder DP)

For each node, we only need **information from left and right subtrees**:

- Minimum value
- Maximum value
- Size of largest BST

Using **postorder traversal**, we compute everything bottom-up in **one pass**.

---

## Algorithm

For every node, maintain:

- `minNode` → minimum value in subtree
- `maxNode` → maximum value in subtree
- `maxSize` → size of largest BST in subtree

Steps:

1. Recursively compute values for left and right subtrees.
  2. If `left.maxNode < root.data < right.minNode`, subtree is BST:
    - Update `size = left.size + right.size + 1`
  3. Else:
    - Mark subtree invalid using extreme values
    - Carry forward max size from children
  4. Return the maximum size found.
-

## Code (Optimal)

```
#include <bits/stdc++.h>
using namespace std;

struct TreeNode {
 int data;
 TreeNode* left;
 TreeNode* right;
 TreeNode(int val) : data(val), left(nullptr), right(nullptr) {}
};

class Solution {
public:
 struct NodeValue {
 int minNode, maxNode, maxSize;
 NodeValue(int minNode, int maxNode, int maxSize)
 : minNode(minNode), maxNode(maxNode), maxSize(maxSize) {}
 };

 NodeValue helper(TreeNode* node) {
 if (!node) {
 return NodeValue(INT_MAX, INT_MIN, 0);
 }

 NodeValue left = helper(node->left);
 NodeValue right = helper(node->right);

 if (left.maxNode < node->data && node->data < right.minNode) {
 return NodeValue(
 min(node->data, left.minNode),
 max(node->data, right.maxNode),
 left.maxSize + right.maxSize + 1
);
 }

 return NodeValue(INT_MIN, INT_MAX, max(left.maxSize,
right.maxSize));
 }
}
```

```
int largestBST(TreeNode* root) {
 return helper(root).maxSize;
}
};
```

---

## Complexity Analysis

- **Time Complexity:**  $O(N)$   
Each node is processed exactly once.
- **Space Complexity:**  $O(H)$   
Recursive stack space (height of tree).

# **Graphs**

# 1. Introduction to Graph

## What is a graph data structure?

There are two types of data structures

- Linear
- Non - linear

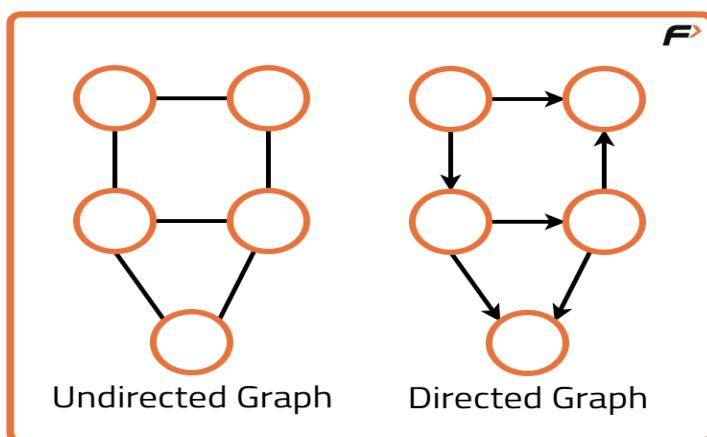
We are aware of linear data structures such as arrays, stacks, queues, and linked lists. They are called linear because data elements are arranged in a linear or sequential manner.

The only non-linear data structure that we've seen so far is Tree. In fact, a tree is a special type of graph with some restrictions. Graphs are data structures that have a wide-ranging application in real life. These include analysis of electrical circuits, finding the shortest routes between two places, building navigation systems like Google Maps, even social media using graphs to store data about each user, etc. To understand and use the graph data structure, let's get familiar with the definitions and terms associated with graphs.

---

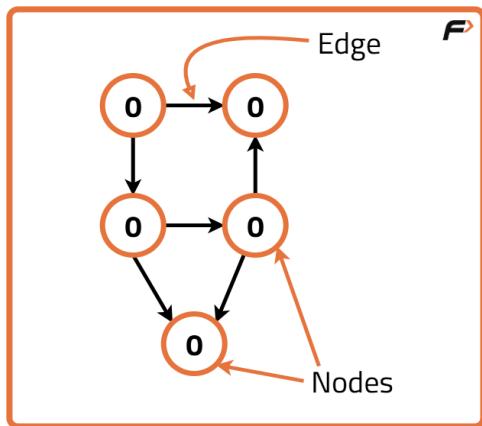
## Definitions and Terminology

A graph is a non-linear data structure consisting of nodes that have data and are connected to other nodes through edges.



**Nodes** are circles represented by numbers. Nodes are also referred to as vertices. They store the data. The numbering of the nodes can be done in any order, no specific order needs to be followed.

In the following example, the number of nodes or vertices = 5

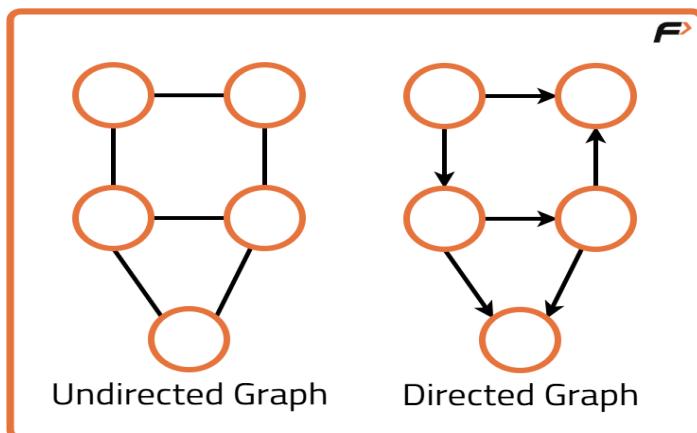


Two nodes are connected by a horizontal line called **Edge**. Edge can be directed or undirected. Basically, pairs of vertices are called edges.

In the above example, the edge can go from 1 to 4 or from 4 to 1, i.e. a bidirectional edge can be in both directions, hence called an **undirected edge**. Thus, the pairs (1,4) and (4,1) represent the same edge.

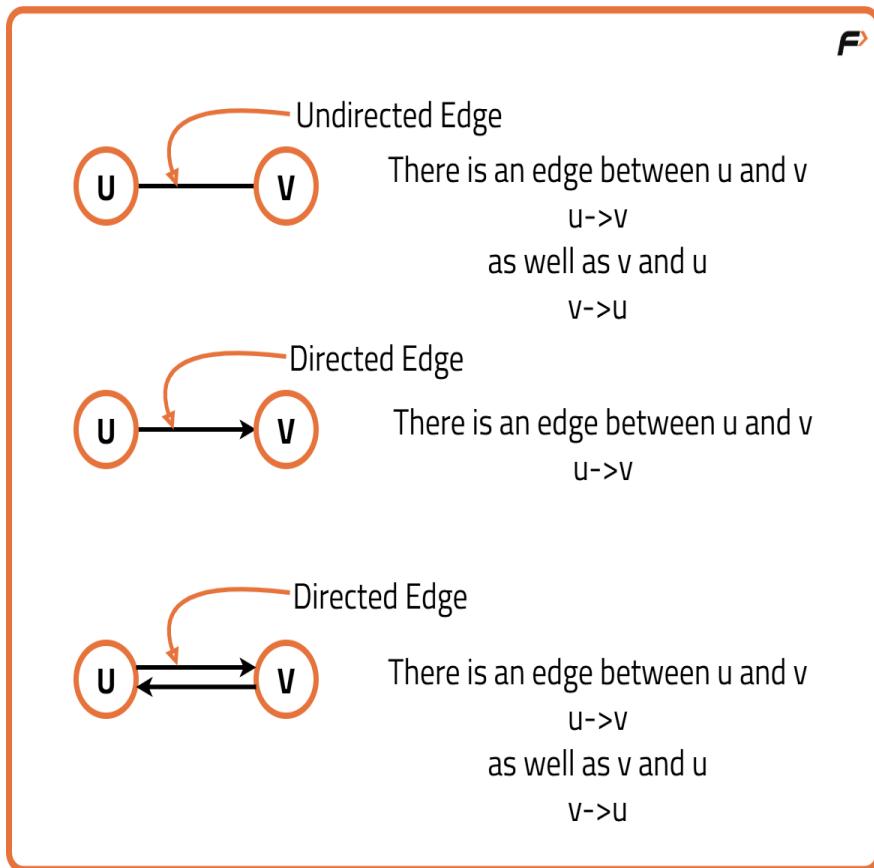
---

## Types of Graphs



- An undirected graph is a graph where edges are bidirectional, with no direction associated with them, i.e., there will be an undirected edge. In an undirected graph, the pair of vertices representing any edge is unordered. Thus, the pairs  $(u, v)$  and  $(v, u)$  represent the same edge.
- A directed graph is a graph where all the edges are directed from one vertex to another, i.e., there will be a directed edge. It contains an ordered pair of vertices. It implies each edge is represented by a directed pair  $\langle u, v \rangle$ . Therefore,  $\langle u, v \rangle$  and  $\langle v, u \rangle$  represent two different edges.

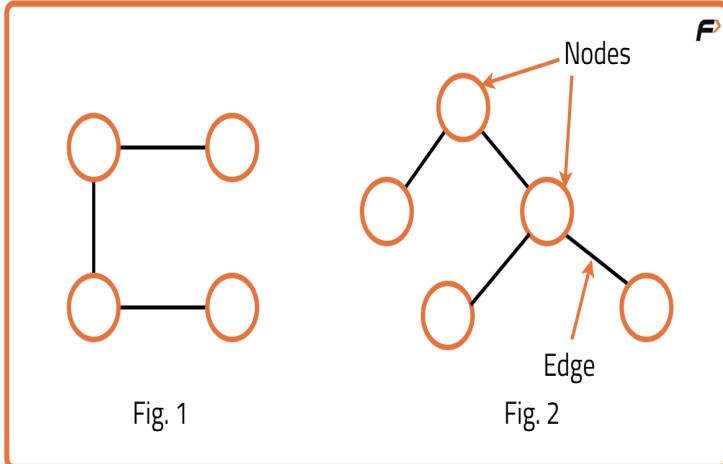
There can be multi-directed edges, hence bidirectional edges, as shown in the example below.



# Structure of Graph

Does every graph have a cycle?

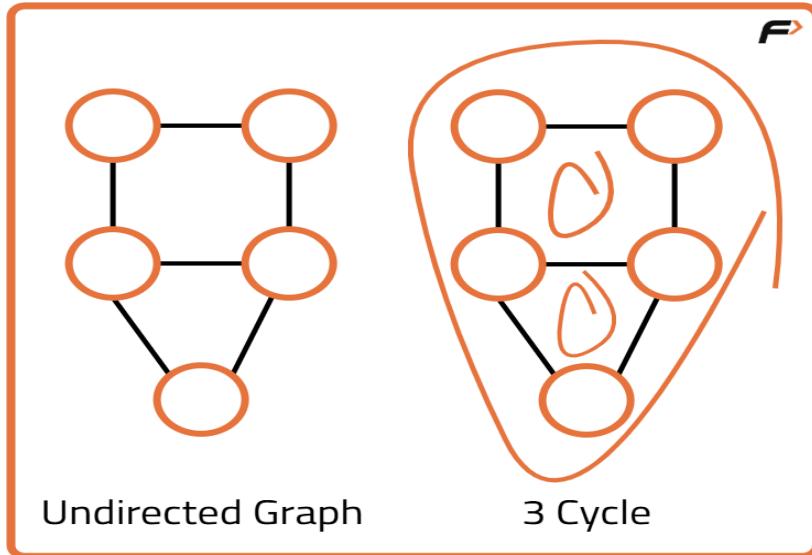
The answer is No! Let us consider the following examples to understand this.



**Fig. 1** does not form a cycle but still, it is a graph.

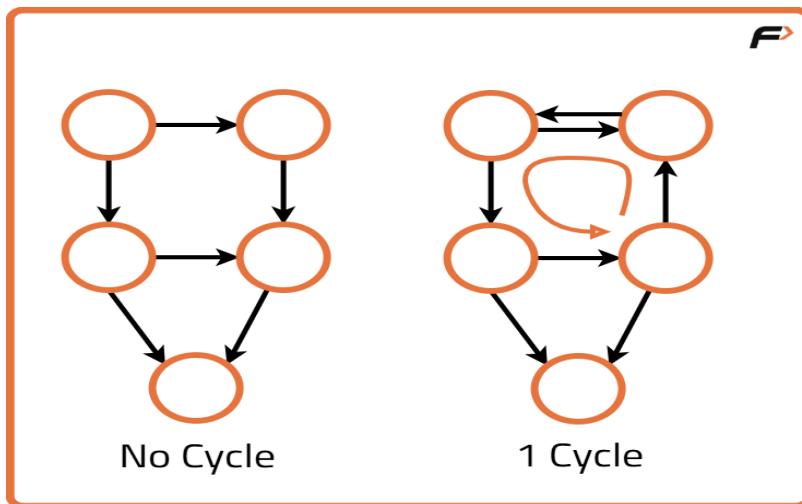
**Fig. 2** is an example of a binary tree. It can also be called a graph because it follows all the rules. We've nodes and edges, and this is the minimal condition to be called a graph.

So a graph does not necessarily mean to be an enclosed structure, it can be an open structure as well. A graph is said to have a cycle if it starts from a node and ends at the same node. There can be multiple cycles in a graph.



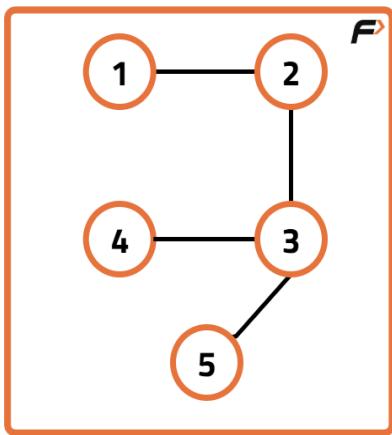
If there is at least one cycle present in the graph then it is called an **Undirected Cyclic Graph**.

In the following examples of directed graphs, the first directed graph is not cyclic as we can't start from a node and end at the same node. Hence it is called **Directed Acyclic Graph**, commonly called **DAG**.



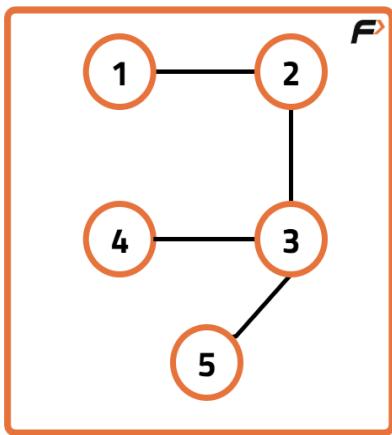
If we just add an edge to the directed graph, then at least one cycle is present in the graph, hence it becomes **Directed Cyclic Graph**.

## Path in a Graph



The path contains a lot of nodes and each of them is reachable.

Consider the given graph,



1 2 3 5 is a path.

1 2 3 2 1 is not a path, because a node can't appear twice in a path.

1 3 5 is not a path, as adjacent nodes must have an edge and there is no edge between 1 and 3.

---

## Degree of Graph

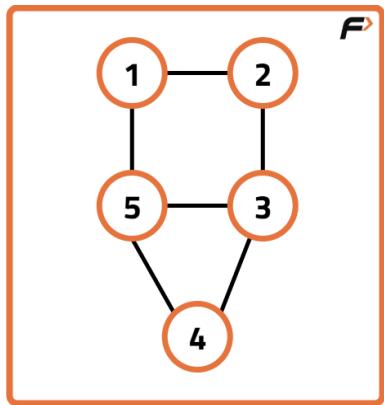
It is the number of edges that go inside or outside that node.

For **undirected graphs**, the degree is the number of edges attached to a node.

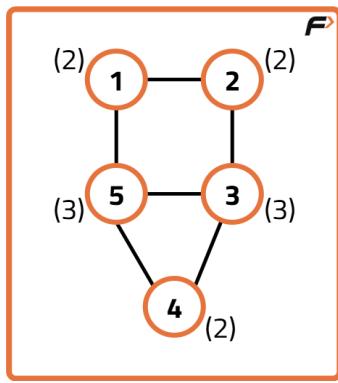
Example,

$$D(3) = 3$$

$$D(4) = 2$$

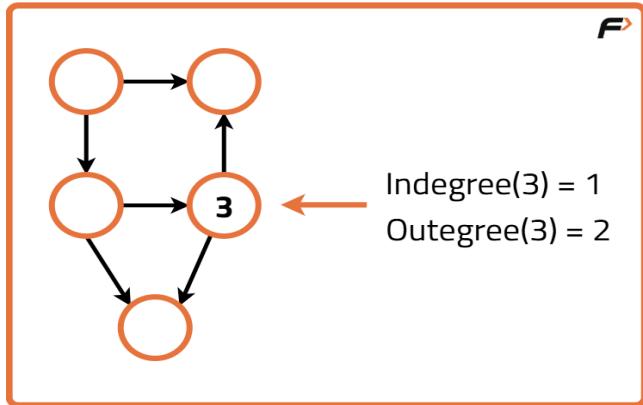


**Property:** It states that the total degree of a graph is equal to twice the number of edges. This is because every edge is associated/ connected to two nodes.



$$\text{Total Degree of a graph} = 2 \times E$$

$$\text{Example, } (2+2+3+2+3) = 2 \times 6 \Rightarrow 12 = 12$$

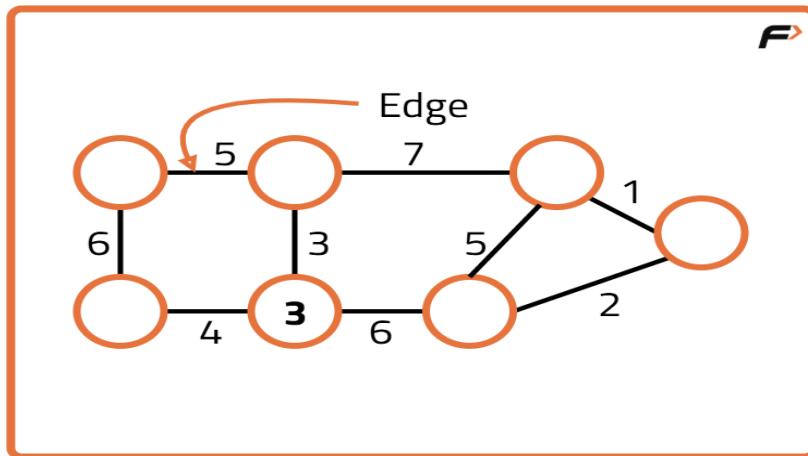


For **directed graphs**, we've Indegree and Outdegree. The **indegree** of a node is the number of incoming edges. The **outdegree** of a node is the number of outgoing edges.

---

## Edge Weight

A graph may have weights assigned on its edges. It is often referred to as the cost of the edge.



If weights are not assigned then we assume the unit weight, i.e, 1. In applications, weight may be a measure of the cost of a route. For example, if vertices A and B represent towns in a road network, then weight on edge AB may represent the cost of moving from A to B, or vice versa.

## 2. Graph Representation in C++

A graph is made of **nodes (vertices)** and **edges** connecting them.  
The question gives whether the graph is **directed or undirected**.

Input format:

- First line: two integers n and m
  - n = number of nodes
  - m = number of edges
- Next m lines: two integers u and v representing an edge between nodes u and v

There is **no fixed limit on the number of edges**. If more edges are added, m increases.

Nodes can be **1-based indexed**, so we usually create storage of size n+1.

The two most common ways to store a graph are:

1. Adjacency Matrix
  2. Adjacency List
- 

### Approach 1: Adjacency Matrix

An adjacency matrix is a **2D array of size n x n**.

If there is an edge between node i and node j, then:

- $\text{adj}[i][j] = 1$
- Otherwise,  $\text{adj}[i][j] = 0$

For an **undirected graph**, if there is an edge between u and v, then both  $(u, v)$  and  $(v, u)$  are marked.

### Example Explanation

Input:

```
5 6
1 2
1 3
2 4
3 4
3 5
4 5
```

Here:

- Number of nodes = 5
- Number of edges = 6

If there is an edge between 1 and 2:

- $\text{adj}[1][2] = 1$
- $\text{adj}[2][1] = 1$

If there is **no edge** between 5 and 1:

- $\text{adj}[5][1] = 0$

This matrix helps quickly check whether two nodes are directly connected.

|   | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 |   |   |   |   |   |   |
| 1 |   |   | 1 | 1 |   |   |
| 2 |   | 1 |   |   | 1 |   |
| 3 |   | 1 |   |   | 1 | 1 |
| 4 |   |   | 1 | 1 |   | 1 |
| 5 |   |   |   | 1 | 1 |   |

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

## Algorithm

1. Read n and m
2. Create a 2D array adj of size (n+1) x (n+1)
3. For each edge (u, v):
  - o Mark adj[u][v] = 1
  - o Mark adj[v][u] = 1 (only for undirected graph)

## Code

```
#include <bits/stdc++.h>
using namespace std;

int main()
{
 int n, m;
 cin >> n >> m;
 int adj[n+1][n+1];
 for(int i = 1; i <= n; i++)
 for(int j = 1; j <= n; j++)
```

```

 adj[i][j] = 0;
for(int i = 0; i < m; i++)
{
 int u, v;
 cin >> u >> v;
 adj[u][v] = 1;
 adj[v][u] = 1; // remove this line for directed graph
}
return 0;
}

```

## Complexity Analysis

- **Space Complexity:**  $O(n \times n)$   
Reason: A full matrix of size  $n \times n$  is stored.
  - **Time Complexity:**  $O(m)$   
Reason: Each edge is processed once.
- 

## Approach 2: Adjacency List (Undirected Graph)

In this method, for every node we store a **list of its adjacent nodes**.  
This uses much less space compared to adjacency matrix.

Each edge in an undirected graph is stored **twice**:

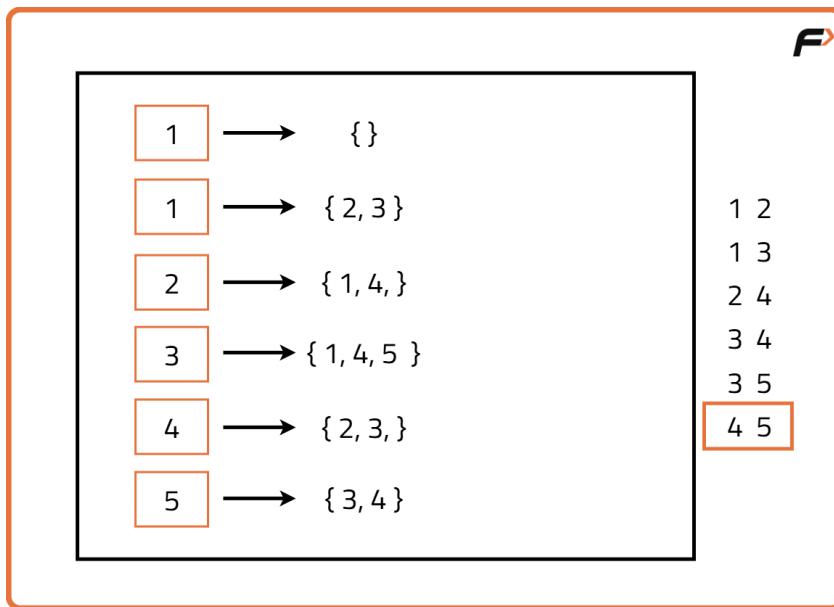
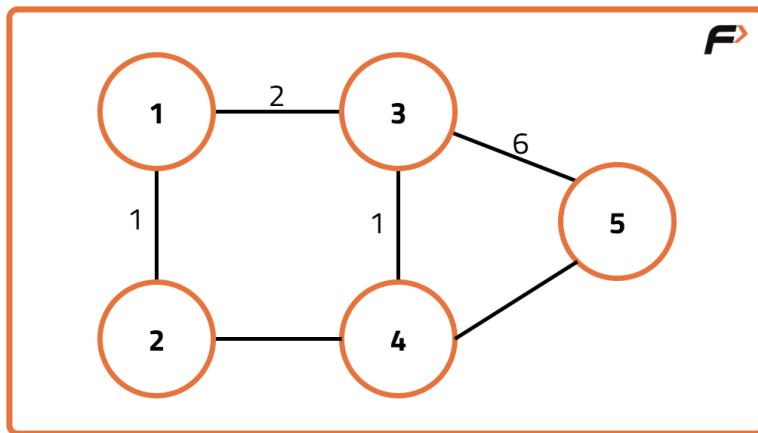
- $v$  is stored in the list of  $u$
- $u$  is stored in the list of  $v$

## Example Explanation

From the same graph:

- Node 4 is connected to 2, 3, and 5

- So  $\text{adj}[4] = \{2, 3, 5\}$



## Algorithm

1. Read n and m
2. Create an array of vectors adj of size n+1
3. For each edge (u, v):
  - o Add v to adj[u]

- Add u to adj[v]

## Code

```
#include <bits/stdc++.h>
using namespace std;

int main()
{
 int n, m;
 cin >> n >> m;
 vector<int> adj[n+1]; //Array of vectors
 for(int i = 0; i < m; i++)
 {
 int u, v;
 cin >> u >> v;
 adj[u].push_back(v);
 adj[v].push_back(u);
 }
 return 0;
}
```

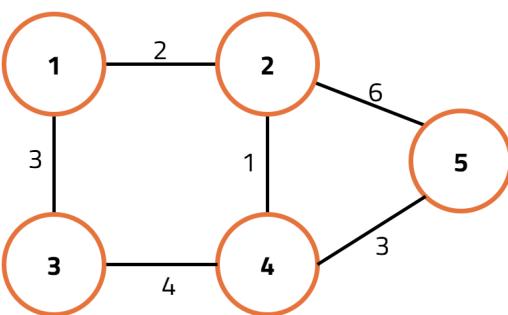
## Complexity Analysis

- **Space Complexity:**  $O(2E)$   
Reason: Each edge is stored twice in an undirected graph.
  - **Time Complexity:**  $O(m)$   
Reason: Each edge insertion is done once.
- 

## Approach 3: Adjacency List (Directed Graph)

In a directed graph, an edge  $(u, v)$  means  $u \rightarrow v$  only.  
So we store v only in the list of u.

Each edge is stored **only once**.



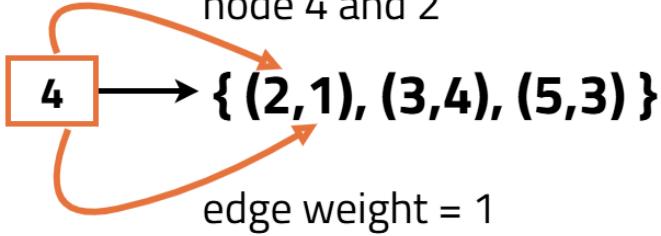
### Undirected Graph

```
int u, v, wt;
cin >> u >> v >> wt;
adj[u][v] = wt;
adj[v][u] = wt;
```

### Directed Graph

```
int u, v, wt;
cin >> u >> v >> wt;
adj[u][v] = wt;
```

edge between  
node 4 and 2



## Algorithm

1. Read n and m
2. Create an array of vectors adj of size n+1
3. For each edge (u, v):

- Add v to adj[u]

## Code

```
#include <bits/stdc++.h>
using namespace std;

int main()
{
 int n, m;
 cin >> n >> m;
 vector<int> adj[n+1];
 for(int i = 0; i < m; i++)
 {
 int u, v;
 cin >> u >> v;
 adj[u].push_back(v);
 }
 return 0;
}
```

## Complexity Analysis

- **Space Complexity:**  $O(E)$   
Reason: Each directed edge is stored only once.
  - **Time Complexity:**  $O(m)$   
Reason: Each edge is processed once.
- 

## Approach 4: Weighted Graph Representation (Adjacency List)

In weighted graphs, every edge has a **weight**.  
So instead of storing only the node, we store a **pair**:

- (neighbor, weight)

## Algorithm

1. Read n and m
2. Create an array of vectors of pairs
3. For each edge (u, v, wt):
  - Store (v, wt) in adj[u]
  - Also store (u, wt) for undirected graph

## Code

```
#include <bits/stdc++.h>
using namespace std;

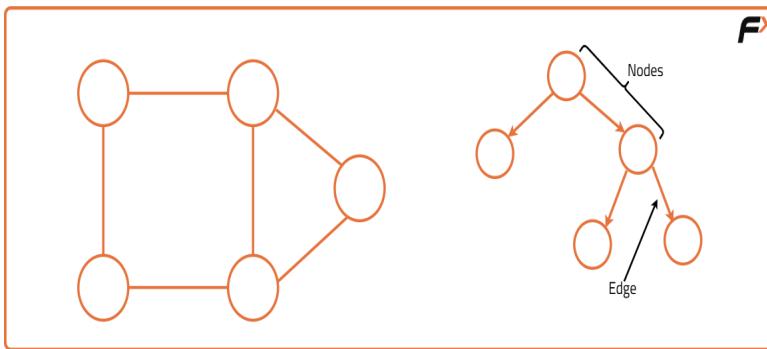
int main()
{
 int n, m;
 cin >> n >> m;
 vector<pair<int,int>> adj[n+1];
 for(int i = 0; i < m; i++)
 {
 int u, v, wt;
 cin >> u >> v >> wt;
 adj[u].push_back({v, wt});
 adj[v].push_back({u, wt}); // remove for directed weighted
graph
 }
 return 0;
}
```

- **Space Complexity:** O(2E) for undirected, O(E) for directed  
Reason: Each edge stores a pair (node, weight).

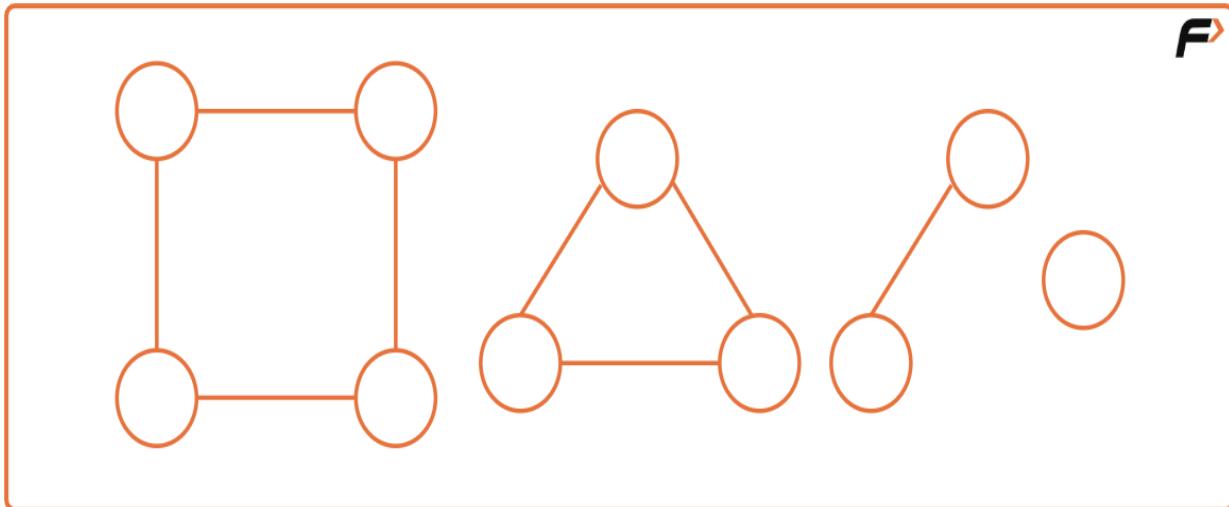
- **Time Complexity:**  $O(m)$   
Reason: Each edge is inserted once.

### 3. Connected Components in Graphs

So far we've seen different types of graphs. Graphs can be connected or can be like a binary tree (as we know all trees are graphs with some restrictions) as shown in the following figure.



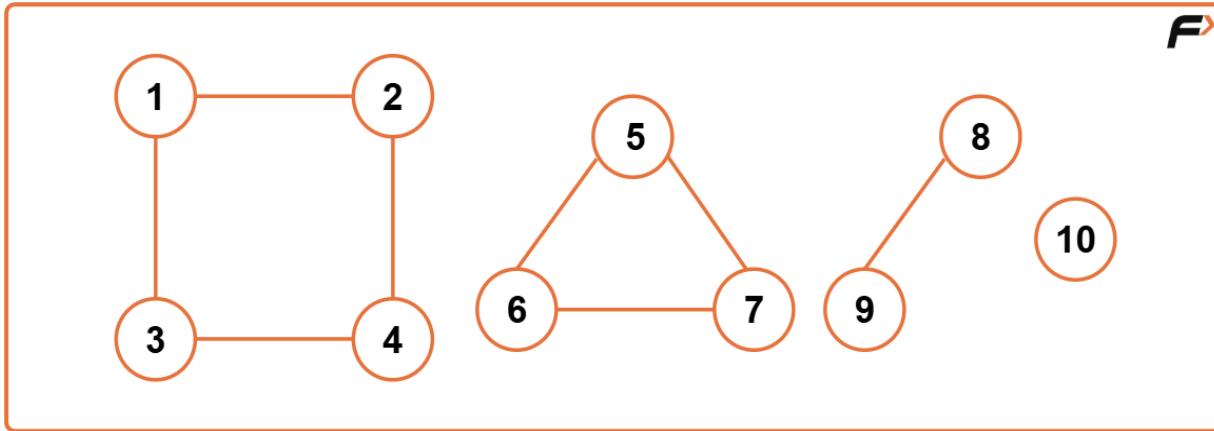
But what would you call the following figure?



The most common answer would be these are 4 different graphs as they are not connected.

But is it possible to call them a single graph? To answer this, let us consider the question given:

Given an undirected graph with 10 nodes and 8 edges. The edges are (1,2), (1,3), (2,4), (4,3), (5,6), (5,7), (6,7), (8,9) .The graph that can be formed with the given information is as follows:



Apparently, it's a graph, which is in 4 pieces, the last one being a single node. In this case, we can say, the graph has been broken down into 4 different **connected components**. So next time if you see two different parts of a graph and they are not connected, then do not say that it cannot be a single graph. In the above example, they can be 4 different graphs but according to the given question and the input, we can call them parts of a single graph.

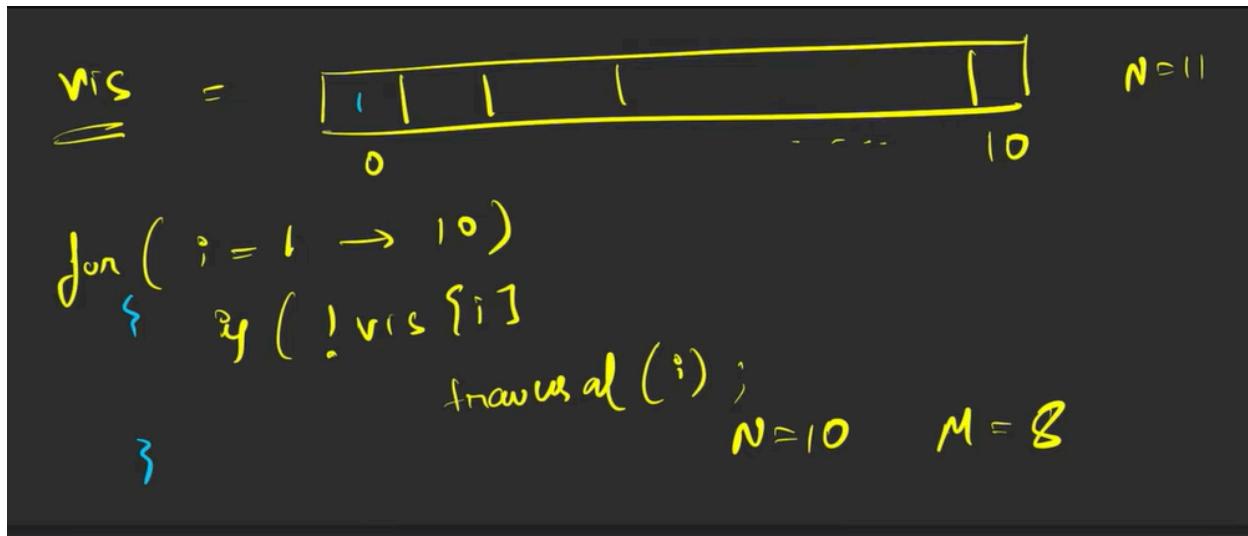
## Graph Traversal

In the upcoming topics, we'll be learning about a lot of algorithms. Now, assume a traversal algorithm. Any traversal algorithm will always use a **visited array**.

For the same example, we will create an array of size 11 ( $n+1$ ) starting with the zeroth index. Initialize this visited array to zero, indicating that all the nodes are unvisited. Then follow the following algorithm. If a node is not visited, then call the traversal algorithm.

### **Why can't we just call traversal(1)?**

We cannot just call `traversal(node)` because a graph can have multiple components and traversal algorithms are designed in such a way that they will traverse the entire connected portion of the graph. For example, `traversal(1)` will traverse only the connected nodes, i.e., nodes 2, 3, and 4, but not the connected components.



## 4. Breadth First Search (BFS): Level Order Traversal

Given an **undirected graph**, return a vector containing all the nodes traversed using **Breadth First Search (BFS)**.

BFS starts from a starting node and explores all its neighbors first before moving to the next level of nodes.

The traversal is done **level by level** using a **queue**.

The graph is represented using an **adjacency list**, and nodes are assumed to be **0-based indexed**.

The traversal always starts from node 0.

### Algorithm

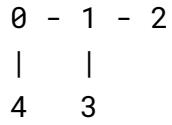
1. Create a **visited** array of size V and initialize all values to 0.
2. Create a queue data structure.

3. Mark the starting node 0 as visited and push it into the queue.
4. While the queue is not empty:
  - o Take the front element of the queue and remove it.
  - o Add this node to the BFS result vector.
  - o Traverse all adjacent nodes of this node.
  - o If an adjacent node is not visited:
    - Mark it as visited.
    - Push it into the queue.
5. Continue until the queue becomes empty.
6. Return the BFS traversal vector.

This ensures nodes are visited in **breadthwise (level order)** manner.

## Example Explanation

For the graph:

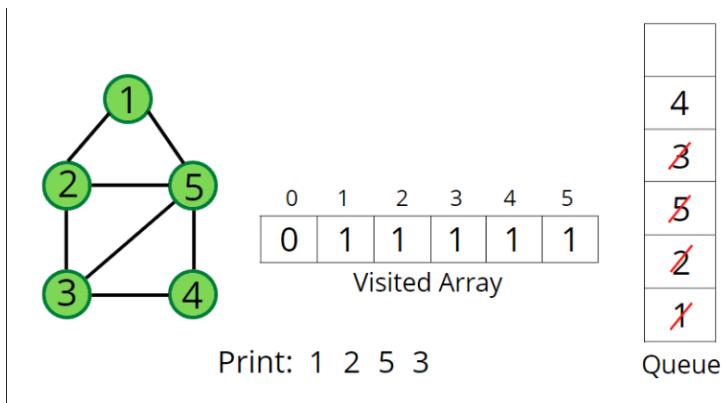
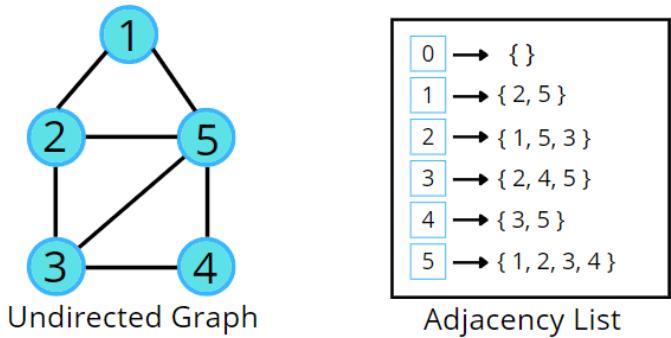


Starting from node 0:

- Visit 0, push its neighbors 1 and 4
- Visit 1, push its neighbors 2 and 3
- Visit 4, then 2, then 3

Output order:

0 1 4 2 3



## Code

```

#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 vector<int> bfsOfGraph(int V, vector<int> adj[]) {
 int vis[V] = {0};
 queue<int> q;
 vector<int> bfs;
 vis[0] = 1;
 q.push(0);
 while(!q.empty()) {
 int node = q.front();
 q.pop();
 bfs.push_back(node);
 for(auto it : adj[node]) {
 if(!vis[it]) {

```

```

 vis[it] = 1;
 q.push(it);
 }
}
return bfs;
}

void addEdge(vector<int> adj[], int u, int v) {
 adj[u].push_back(v);
 adj[v].push_back(u);
}

void printAns(vector<int> &ans) {
 for(int i = 0; i < ans.size(); i++)
 cout << ans[i] << " ";
}

int main() {
 vector<int> adj[6];
 addEdge(adj, 0, 1);
 addEdge(adj, 1, 2);
 addEdge(adj, 1, 3);
 addEdge(adj, 0, 4);
 Solution obj;
 vector<int> ans = obj.bfsOfGraph(5, adj);
 printAns(ans);
 return 0;
}

```

## Complexity Analysis

- **Time Complexity:**  $O(N) + O(2E)$

Reason:  $O(N)$  for visiting all nodes and  $O(2E)$  for traversing adjacency lists in an undirected graph.

- **Space Complexity:**  $O(N)$   
Reason: Space used by visited array, queue, and result vector.

## ✓ Updated BFS for disconnected graph

```
vector<int> bfsOfGraph(int V, vector<int> adj[]) {
 vector<int> bfs;
 vector<int> vis(V, 0);
 queue<int> q;

 for(int i = 0; i < V; i++) {
 if(!vis[i]) {
 vis[i] = 1;
 q.push(i);

 while(!q.empty()) {
 int node = q.front();
 q.pop();
 bfs.push_back(node);

 for(auto it : adj[node]) {
 if(!vis[it]) {
 vis[it] = 1;
 q.push(it);
 }
 }
 }
 }
 }
 return bfs;
}
```

## 5. Depth First Search (DFS)

Given an **undirected graph**, return a vector of all nodes by traversing the graph using **Depth First Search (DFS)**.

DFS explores the graph by going **as deep as possible** along one path before backtracking and

exploring other paths.

The graph is represented using an **adjacency list**, and traversal starts from a chosen node.

---

## Approach

### Algorithm

1. Use a visited array to keep track of visited nodes.
2. Start DFS from a given node.
3. Mark the current node as visited and store it in the result vector.
4. Traverse all adjacent nodes of the current node using the adjacency list.
5. For each adjacent node that is not visited:
  - o Recursively apply DFS on that node.
6. When a node has no unvisited neighbors, backtrack to the previous node.
7. Repeat until all reachable nodes are visited.

DFS uses **recursion and backtracking** to explore the graph fully.

### Example Explanation

Example:

```
V = 5
adj = [[1,2], [0,3], [0,4], [1], [2]]
```

Start DFS from node 0:

- Visit 0
- Go to 1

- Go to 3 (no more neighbors, backtrack)
- Back to 0, go to 2
- Go to 4

DFS traversal:

0 1 3 2 4

This order shows how DFS goes deep first, then backtracks.

---

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 void dfs(int v, vector<int> adj[], vector<int>& visited,
vector<int>& result) {
 visited[v] = 1;
 result.push_back(v);
 for(int u : adj[v]) {
 if(!visited[u]) {
 dfs(u, adj, visited, result);
 }
 }
 }
};

int main() {
 int V = 5;
 vector<int> adj[V];
 adj[0] = {1, 2};
 adj[1] = {0, 3};
 adj[2] = {0, 4};
```

```

adj[3] = {1};
adj[4] = {2};
vector<int> visited(V, 0);
vector<int> result;
Solution sol;
sol.dfs(0, adj, visited, result);
for(int x : result) cout << x << " ";
return 0;
}

```

## Complexity Analysis

- **Time Complexity:**  $O(V + E)$

Reason: Each vertex is visited once and each edge is checked once through the adjacency list.

- **Space Complexity:**  $O(V)$

Reason: Extra space is used by the visited array and the recursion stack.

## If Node fix na ho to:

```

vector<int> dfsOfGraph(int V, vector<int> adj[]) {
 vector<int> visited(V, 0);
 vector<int> result;

 for(int i = 0; i < V; i++) {
 if(!visited[i]) {
 dfs(i, adj, visited, result);
 }
 }
 return result;
}

```

# 6. Number of Provinces

Given an **undirected graph** with  $V$  vertices, find the number of **provinces**.

Two vertices  $u$  and  $v$  belong to the same province if there exists a **path** between them, directly or indirectly.

The graph is given as an **adjacency matrix**  $\text{adj}$  of size  $V \times V$ :

- $\text{adj}[i][j] = 1$  means city  $i$  and city  $j$  are directly connected
- $\text{adj}[i][j] = 0$  means there is no direct connection

A province is simply a **connected component** of the graph.

---

## Algorithm

A province contains all cities that can be reached from one city using traversal.

1. Create a **visited** array to track visited cities.
2. Convert the adjacency matrix into an adjacency list for easy traversal.
3. Loop through all cities from 0 to  $V-1$ .
4. If a city is not visited:
  - It means we found a new province.
  - Increase the province counter.
  - Run DFS from that city to mark all cities in this province as visited.
5. The number of times DFS is started gives the number of provinces.

DFS ensures that all directly or indirectly connected cities are visited in one call.

## Example Explanation

Example 1:

```
adj = [
[1,0,0,1],
[0,1,1,0],
[0,1,1,0],
[1,0,0,1]
]
```

- Cities 1 and 4 are connected → one province
- Cities 2 and 3 are connected → another province
- No connection between these groups

Answer = 2

Example 2:

```
adj = [
[1,0,1],
[0,1,0],
[1,0,1]
]
```

- Cities 1 and 3 are connected
- City 2 is isolated

Answer = 2

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
private:
 void dfs(int node, vector<int> adjList[], int visited[]) {
 visited[node] = 1;
```

```

 for(auto it : adjList[node]) {
 if(!visited[it]) {
 dfs(it, adjList, visited);
 }
 }
 }

public:
 int numProvinces(vector<vector<int>> adj, int V) {
 vector<int> adjList[V];
 for(int i = 0; i < V; i++) {
 for(int j = 0; j < V; j++) {
 if(adj[i][j] == 1 && i != j) {
 adjList[i].push_back(j);
 adjList[j].push_back(i);
 }
 }
 }
 }

 int visited[V] = {0};
 int count = 0;

 for(int i = 0; i < V; i++) {
 if(!visited[i]) {
 count++;
 dfs(i, adjList, visited);
 }
 }
 return count;
}

int main() {
 vector<vector<int>> adj = {
 {1,0,1},
 {0,1,0},
 {1,0,1}
 };
}

```

```

int V = 3;
Solution obj;
cout << obj.numProvinces(adj, V);
return 0;
}

```

## Complexity Analysis

- **Time Complexity:**  $O(V + E)$   
Reason: Each vertex is visited once, and all its edges are traversed during DFS.
- **Space Complexity:**  $O(V)$   
Reason: Space used for visited array and recursion stack.

# 7. Connected Components

Given an **undirected graph** with  $V$  vertices numbered from 0 to  $V-1$  and a list of  $E$  edges, find the **number of connected components** in the graph.

Two vertices belong to the same connected component if there exists a **path** between them.

A connected component is a group of vertices where:

- Every vertex can reach every other vertex in the group.
- No vertex in the group is connected to a vertex outside the group.

## Approach

To count connected components, we need to make sure every vertex is visited exactly once and grouped correctly.

1. Build an **adjacency list** from the given edge list.
2. Create a **visited** array of size  $V$  and initialize all values to 0.
3. Initialize a counter **components** = 0.
4. Loop through all vertices from 0 to  $V-1$ :
  - o If a vertex is not visited:
    - This means we found a new connected component.
    - Increment the component counter.
    - Perform **BFS** starting from this vertex.
    - During BFS, mark all reachable vertices as visited.
5. After all vertices are processed, return the component count.

Each BFS traversal covers exactly one connected component.

## Example Explanation

Example 1:

```
V = 4
edges = [[0,1],[1,2]]
```

- Vertices {0, 1, 2} are connected → one component
- Vertex {3} is isolated → another component

Answer = 2

Example 2:

```
V = 7
edges = [[0,1],[1,2],[2,3],[4,5]]
```

- $\{0, 1, 2, 3\}$  form one component
- $\{4, 5\}$  form another component
- $\{6\}$  is isolated

Answer = 3

---

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int countComponents(int V, vector<vector<int>>& edges) {
 vector<vector<int>> adj(V);
 for(auto &e : edges) {
 adj[e[0]].push_back(e[1]);
 adj[e[1]].push_back(e[0]);
 }

 vector<int> visited(V, 0);
 int components = 0;

 for(int i = 0; i < V; i++) {
 if(!visited[i]) {
 components++;
 queue<int> q;
 q.push(i);
 visited[i] = 1;
 while(!q.empty()) {
 int node = q.front();
 q.pop();
 for(auto nbr : adj[node]) {
 if(!visited[nbr]) {
 visited[nbr] = 1;
 }
 }
 }
 }
 }
 }
};
```

```

 q.push_nbr();
 }
}
}
return components;
}
};

int main() {
int V = 5;
vector<vector<int>> edges = {{0,1},{1,2},{3,4}};
Solution sol;
cout << sol.countComponents(V, edges);
return 0;
}

```

## Complexity Analysis

- **Time Complexity:**  $O(V + E)$   
Reason: Each vertex is visited once and each edge is checked at most twice.
- **Space Complexity:**  $O(V + E)$   
Reason: Space used for adjacency list, visited array, and queue.

# 8. Rotten Oranges : Min time to rot all oranges : BFS

Given an  $n \times m$  grid where:

- 2 represents a **rotten orange**

- 1 represents a **fresh orange**
- 0 represents an **empty cell**

Every minute, a fresh orange becomes rotten if it is **adjacent (up, down, left, right)** to a rotten orange.

Return the **minimum number of minutes** required so that **no fresh orange remains**.

If it is **not possible** to rot all oranges, return -1.

---

## Approach

### Algorithm

The problem is solved using **Breadth First Search (BFS)** because rotting happens **level by level (minute by minute)**.

1. Create a **queue** to store positions of all initially rotten oranges.
2. Traverse the grid:
  - Count total number of oranges (tot) (fresh + rotten).
  - Push all rotten oranges into the queue.
3. Maintain:
  - cnt → number of oranges that become rotten during BFS
  - days → total minutes taken
4. Perform BFS:
  - For each level (one minute), process all currently rotten oranges.
  - For every rotten orange, check its 4 directions.
  - If a fresh orange is found, mark it rotten and push it into the queue.
  - Increase the rotten count.

5. After processing one level, if the queue is not empty, increment days.

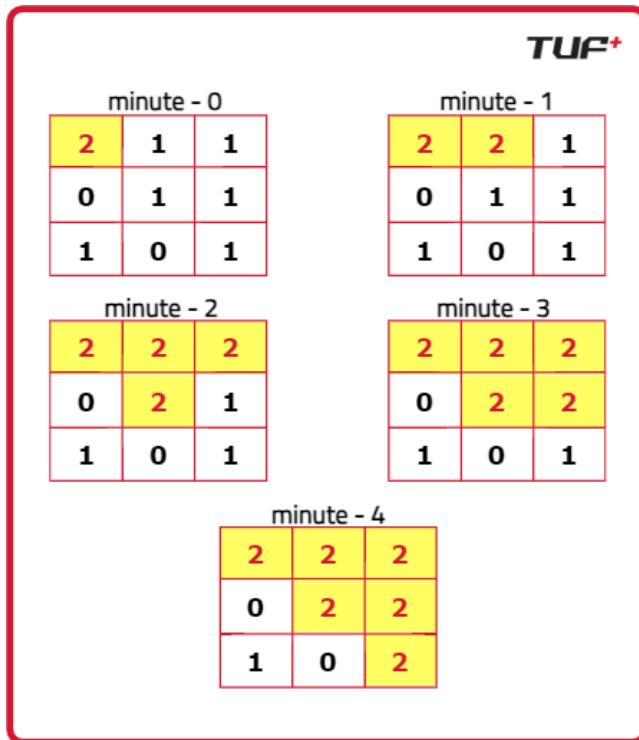
6. After BFS ends:

- If `cnt == tot`, return days
- Otherwise, return -1 (some fresh oranges are unreachable)

## Example Explanation

Example 1:

```
grid = [
 [2,1,1],
 [0,1,1],
 [1,0,1]
]
```



Orange at position (2, 0) can never be reached by any rotten orange.

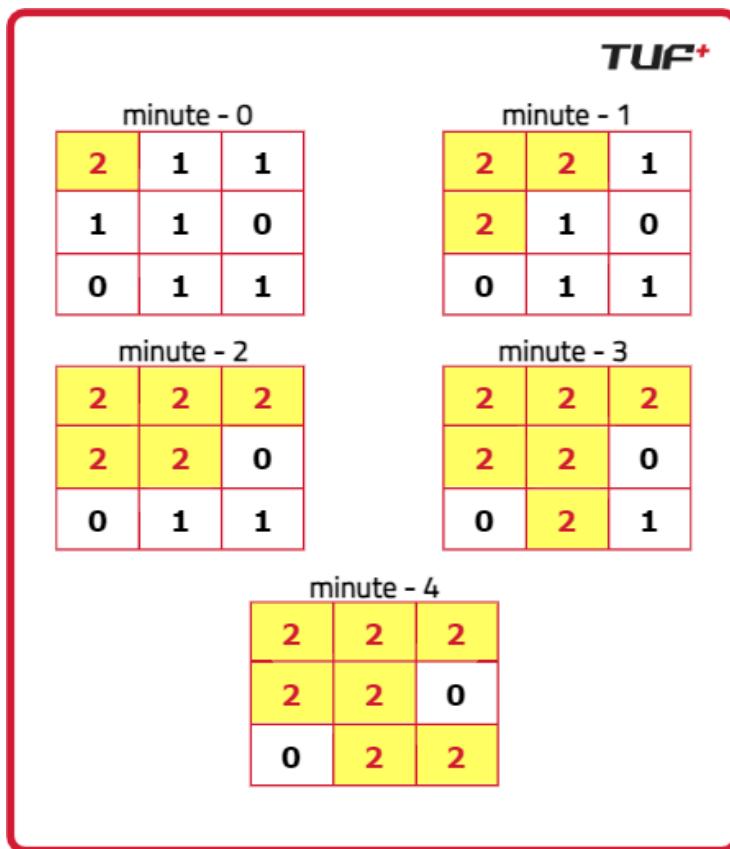
So answer is -1.

Example 2:

```
grid = [
 [2,1,1],
 [1,1,0],
 [0,1,1]
]
```

All oranges can be rotten level by level.

Total time required = 4.



## Code

```
#include <bits/stdc++.h>
using namespace std;
```

```

int orangesRotting(vector<vector<int>>& grid) {
 if(grid.empty()) return 0;

 int m = grid.size();
 int n = grid[0].size();
 int days = 0;
 int tot = 0;
 int cnt = 0;

 queue<pair<int,int>> rotten;

 for(int i = 0; i < m; i++) {
 for(int j = 0; j < n; j++) {
 if(grid[i][j] != 0) tot++;
 if(grid[i][j] == 2)
 rotten.push({i,j});
 }
 }

 int dx[4] = {0,0,1,-1};
 int dy[4] = {1,-1,0,0};

 while(!rotten.empty()) {
 int k = rotten.size();
 cnt += k;
 while(k--) {
 int x = rotten.front().first;
 int y = rotten.front().second;
 rotten.pop();
 for(int i = 0; i < 4; i++) {
 int nx = x + dx[i];
 int ny = y + dy[i];
 if(nx < 0 || ny < 0 || nx >= m || ny >= n ||
grid[nx][ny] != 1)
 continue;
 grid[nx][ny] = 2;
 rotten.push({nx, ny});
 }
 }
 }

 return days;
}

```

```

 }
 if(!rotten.empty())
 days++;
 }
 return tot == cnt ? days : -1;
}

int main() {
 vector<vector<int>> grid = {
 {2,1,1},
 {1,1,0},
 {0,1,1}
 };
 cout << orangesRotting(grid);
 return 0;
}

```

## Complexity Analysis

- **Time Complexity:**  $O(n \times m)$   
Reason: Each cell is processed once and each time we check 4 directions.
- **Space Complexity:**  $O(n \times m)$   
Reason: In the worst case, all oranges are stored in the queue at once.

## 9. Flood Fill Algorithm - Graphs

An image is represented as a **2D grid of integers**, where each integer represents a pixel color. Given a starting pixel (`sr, sc`) and a `newColor`, perform **flood fill** such that all pixels **connected in 4 directions** (up, right, down, left) having the **same initial color** as the starting pixel are changed to `newColor`.

---

# Approach

## Algorithm

1. Store the **initial color** of the starting pixel ( $sr$ ,  $sc$ ).
2. Create a **copy of the image** so the original image is not modified directly.
3. Use **DFS traversal** starting from ( $sr$ ,  $sc$ ).
4. At each pixel:
  - o Change its color to  $newColor$ .
  - o Explore its 4 neighbors (up, right, down, left).
5. For each neighbor:
  - o Check if it is inside the grid.
  - o Check if it has the **same initial color**.
  - o Check if it is **not already colored** with  $newColor$ .
6. Recursively apply DFS on valid neighbors.
7. Continue until all connected pixels with the initial color are filled.
8. Return the updated image.

## Example Explanation

For the image:

```
1 1 1
1 1 0
1 0 1
```

Starting at (1,1) with  $newColor = 2$ :

- Initial color = 1
- All connected 1s reachable from (1, 1) are changed to 2
- Cells with 0 block the fill

Result:

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

---

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
private:
 void dfs(int row, int col, vector<vector<int>>& ans,
 vector<vector<int>>& image, int newColor,
 int delRow[], int delCol[], int iniColor) {
 ans[row][col] = newColor;
 int n = image.size();
 int m = image[0].size();
 for(int i = 0; i < 4; i++) {
 int nrow = row + delRow[i];
 int ncol = col + delCol[i];
 if(nrow >= 0 && nrow < n && ncol >= 0 && ncol < m &&
 image[nrow][ncol] == iniColor &&
 ans[nrow][ncol] != newColor) {
 dfs(nrow, ncol, ans, image, newColor, delRow, delCol,
 iniColor);
 }
 }
 }
public:
```

```

vector<vector<int>> floodFill(vector<vector<int>>& image,
 int sr,int sc,int newColor) {
 int iniColor = image[sr][sc];
 vector<vector<int>> ans = image;
 int delRow[] = {-1,0,1,0};
 int delCol[] = {0,1,0,-1};
 dfs(sr, sc, ans, image, newColor, delRow, delCol, iniColor);
 return ans;
}

int main() {
 vector<vector<int>> image = {
 {1,1,1},
 {1,1,0},
 {1,0,1}
 };
 Solution obj;
 vector<vector<int>> ans = obj.floodFill(image, 1, 1, 2);
 for(auto i : ans) {
 for(auto j : i)
 cout << j << " ";
 cout << "\n";
 }
 return 0;
}

```

## Complexity Analysis

- **Time Complexity:**  $O(N \times M)$

Reason: In the worst case, DFS visits every cell once and checks 4 directions.

- **Space Complexity:**  $O(N \times M)$

Reason: Extra space for the copied image and recursion stack in the worst case.

# 10. Detect Cycle in an Undirected Graph (using BFS)

Given an **undirected graph** with  $V$  vertices and  $E$  edges, check whether the graph contains **any cycle**.

A cycle exists if we can start from a node and come back to the **same node through a different path**.

---

## Algorithm

We use **Breadth First Search (BFS)** with **parent tracking**.

1. Create a **visited** array of size  $V$  and initialize all values to 0.
2. Since the graph can have multiple connected components, loop through all vertices.
3. For every unvisited vertex, start a BFS:
  - o Push a pair (**node**, **parent**) into the queue.
  - o Mark the node as visited.
4. While the queue is not empty:
  - o Pop the front element (**node**, **parent**).
  - o Traverse all adjacent nodes of **node**.
  - o If an adjacent node is **not visited**:
    - Mark it visited.
    - Push (**adjacentNode**, **node**) into the queue.
  - o Else if the adjacent node is visited **and it is not the parent**:
    - A cycle is detected, return true.

5. If BFS finishes for all components and no cycle is found, return false.

The parent is required to avoid detecting the same edge in reverse direction as a cycle.

## Example Explanation

Example:

Edges: 4 → 5 → 6 → 4

- Start BFS from node 4
- Visit 5, then 6
- From 6, we again reach 4 which is already visited and not the parent
- This confirms the presence of a cycle

If no such situation occurs, then the graph has no cycle.

If you visit a vertex that's already visited AND it's not your parent → cycle!

---

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
private:
 bool detect(int src, vector<int> adj[], int vis[]) {
 vis[src] = 1;
 queue<pair<int,int>> q;
 q.push({src, -1}); // (node, parent)
 while(!q.empty()) {
 int node = q.front().first;
 int parent = q.front().second;
 q.pop();
 for (int i : adj[node]) {
 if (vis[i] == 0) {
 vis[i] = 1;
 q.push({i, node});
 } else if (parent != i) {
 return true;
 }
 }
 }
 return false;
 }
}
```

```

 q.pop();
 for(auto adjacentNode : adj[node]) {
 if(!vis[adjacentNode]) {
 vis[adjacentNode] = 1;
 q.push({adjacentNode, node});
 }
 else if(adjacentNode != parent) {
 return true;
 }
 }
 }
 return false;
}

public:
 bool isCycle(int V, vector<int> adj[]) {
 int vis[V] = {0};
 for(int i = 0; i < V; i++) {
 if(!vis[i]) {
 if(detect(i, adj, vis))
 return true;
 }
 }
 return false;
 }
};

int main() {
 vector<int> adj[4] = {{}, {2}, {1, 3}, {2}};
 Solution obj;
 bool ans = obj.isCycle(4, adj);
 if(ans) cout << "1";
 else cout << "0";
 return 0;
}

```

## Complexity Analysis

- **Time Complexity:**  $O(N + 2E) + O(N)$   
Reason: BFS traverses all nodes and edges, and an extra loop handles disconnected components.
- **Space Complexity:**  $O(N)$   
Reason: Space used by visited array and queue.

# 11. Detect Cycle in an Undirected Graph (using DFS)

Given an **undirected graph** with  $V$  vertices and  $E$  edges, check whether the graph contains **any cycle** using **Depth First Search (DFS)**.

In an undirected graph, a cycle exists if during traversal we reach a **visited node that is not the parent** of the current node.

---

## Approach

### Algorithm

1. Create a **visited** array to mark visited vertices.
2. Since the graph can have **multiple connected components**, iterate over all vertices.
3. For every unvisited vertex:
  - Start a DFS with its parent as -1.
4. In DFS:
  - Mark the current node as visited.

- Traverse all its adjacent nodes.
  - If an adjacent node is not visited, recursively call DFS with current node as parent.
  - If an adjacent node is already visited and is **not the parent**, a cycle exists.
5. If DFS finishes without detecting such a case, no cycle exists.
  6. If any DFS call detects a cycle, return true. Otherwise, return false.

## Example Explanation

Example:

4 → 5 → 6 → 4

- Start DFS from node 4
  - Visit 5, then 6
  - From 6, we reach 4 again
  - 4 is visited and is not the parent of 6
  - Hence, a cycle is detected
- 

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 bool dfs(int node, int parent, vector<int> adj[], vector<int>& visited) {
 visited[node] = 1;
 for(int neighbor : adj[node]) {
 if(visited[neighbor] == 1 && neighbor != parent)
 return true;
 else if(visited[neighbor] == 0)
 if(dfs(neighbor, node, adj, visited))
 return true;
 }
 return false;
 }
};
```

```

 if(!visited[neighbor]) {
 if(dfs(neighbor, node, adj, visited))
 return true;
 }
 else if(neighbor != parent) {
 return true;
 }
 }
 return false;
}

bool isCycle(int V, vector<int> adj[]) {
 vector<int> visited(V, 0);
 for(int i = 0; i < V; i++) {
 if(!visited[i]) {
 if(dfs(i, -1, adj, visited)) // -1 is parent
 return true;
 }
 }
 return false;
}
};

int main() {
 int V = 5;
 vector<int> adj[V];
 adj[0].push_back(1);
 adj[1].push_back(0);
 adj[1].push_back(2);
 adj[2].push_back(1);
 adj[2].push_back(3);
 adj[3].push_back(2);
 adj[3].push_back(4);
 adj[4].push_back(3);
 adj[4].push_back(1);

 Solution sol;
 if(sol.isCycle(V, adj))

```

```

 cout << "Cycle detected";
else
 cout << "No cycle found";
return 0;
}

```

## Complexity Analysis

- **Time Complexity:**  $O(V + E)$   
Reason: Each vertex is visited once and each edge is explored during DFS.
- **Space Complexity:**  $O(V + E)$   
Reason: Space used for adjacency list, visited array, and recursion stack.

# 12. Distance of Nearest Cell having 1

Given a binary grid of size  $N \times M$ , find the **distance of the nearest cell having value 1 for every cell** in the grid.

Distance is measured using **4-directional movement** (up, right, down, left).

---

## Approach

### Algorithm

We use **Breadth First Search (BFS)** because BFS naturally expands **level by level**, which directly represents distance.

1. Create:
  - visited matrix to mark visited cells

- dist matrix to store distance of nearest 1
  - Queue to store ((row, col), steps)
2. Traverse the grid:
- If a cell contains 1, push it into the queue with distance 0
  - Mark it as visited
3. Start BFS:
- Pop a cell from the queue
  - Store its distance in the dist matrix
  - Explore its 4 neighbors
4. For every valid, unvisited neighbor:
- Mark it visited
  - Push it into the queue with steps + 1
5. Continue until the queue becomes empty.
6. Return the distance matrix.

All 1 cells act as **multiple BFS sources**, ensuring the shortest distance is found.

## Example Explanation

Input:

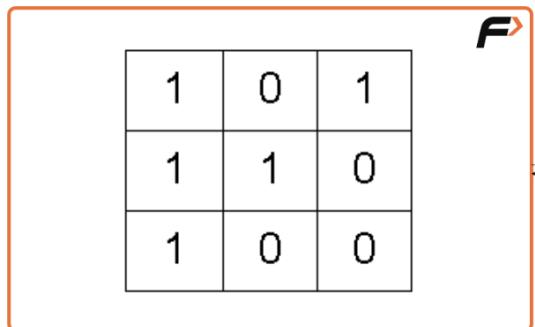
```
0 1 1 0
1 1 0 0
0 0 1 1
```

- All cells containing 1 are pushed into the queue with distance 0

- BFS spreads to nearby 0 cells level by level
- Each 0 cell gets the distance of the nearest 1

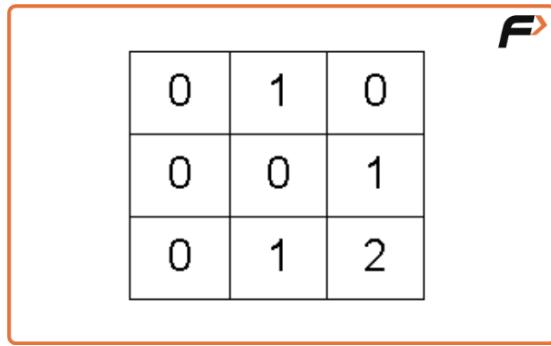
Ex 2:

Input



|   |   |   |
|---|---|---|
| 1 | 0 | 1 |
| 1 | 1 | 0 |
| 1 | 0 | 0 |

Output



|   |   |   |
|---|---|---|
| 0 | 1 | 0 |
| 0 | 0 | 1 |
| 0 | 1 | 2 |

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 vector<vector<int>> nearest(vector<vector<int>> grid) {
 int n = grid.size();
 int m = grid[0].size();
 vector<vector<int>> vis(n, vector<int>(m, 0));
 queue<pair<int, int>> q;
 for (int i = 0; i < n; i++) {
 for (int j = 0; j < m; j++) {
 if (grid[i][j] == 1) {
 q.push({i, j});
 vis[i][j] = 1;
 }
 }
 }
 while (!q.empty()) {
 pair<int, int> p = q.front();
 q.pop();
 int x = p.first;
 int y = p.second;
 if (x - 1 >= 0 && vis[x - 1][y] == 0) {
 vis[x - 1][y] = vis[x][y] + 1;
 q.push({x - 1, y});
 }
 if (x + 1 < n && vis[x + 1][y] == 0) {
 vis[x + 1][y] = vis[x][y] + 1;
 q.push({x + 1, y});
 }
 if (y - 1 >= 0 && vis[x][y - 1] == 0) {
 vis[x][y - 1] = vis[x][y] + 1;
 q.push({x, y - 1});
 }
 if (y + 1 < m && vis[x][y + 1] == 0) {
 vis[x][y + 1] = vis[x][y] + 1;
 q.push({x, y + 1});
 }
 }
 return vis;
 }
};
```

```

vector<vector<int>> dist(n, vector<int>(m, 0));
queue<pair<pair<int,int>, int>> q;

for(int i = 0; i < n; i++) {
 for(int j = 0; j < m; j++) {
 if(grid[i][j] == 1) {
 q.push({{i, j}, 0});
 vis[i][j] = 1;
 }
 }
}

int delrow[] = {-1, 0, 1, 0};
int delcol[] = {0, 1, 0, -1};

while(!q.empty()) {
 int row = q.front().first.first;
 int col = q.front().first.second;
 int steps = q.front().second;
 q.pop();
 dist[row][col] = steps;

 for(int i = 0; i < 4; i++) {
 int nrow = row + delrow[i];
 int ncol = col + delcol[i];
 if(nrow >= 0 && nrow < n && ncol >= 0 && ncol < m &&
 vis[nrow][ncol] == 0) {
 vis[nrow][ncol] = 1;
 q.push({{nrow, ncol}, steps + 1});
 }
 }
}
return dist;
}

int main() {
 vector<vector<int>> grid = {

```

```

 {0,1,1,0},
 {1,1,0,0},
 {0,0,1,1}
};

Solution obj;
vector<vector<int>> ans = obj.nearest(grid);
for(auto i : ans) {
 for(auto j : i)
 cout << j << " ";
 cout << "\n";
}
return 0;
}

```

## Complexity Analysis

- **Time Complexity:**  $O(N \times M)$   
Reason: Each cell is visited once and each visit checks 4 neighbors.
- **Space Complexity:**  $O(N \times M)$   
Reason: Space used for visited matrix, distance matrix, and queue.

# 13. Surrounded Regions | Replace O's with X's

Given a matrix `mat` of size  $N \times M$  where each cell contains either '0' or 'X'.

Replace all '0' with 'X' **only if the '0' is completely surrounded by 'X' from all 4 directions** (up, down, left, right).

An '0' (or group of '0') **connected to the boundary** of the matrix can **never** be surrounded, so it must **remain '0'**.

---

## Example

### Example 1

Input:

```
X X X X
X 0 X X
X 0 0 X
X 0 X X
X X 0 0
```

Explanation:

- The '0' at the bottom-right is connected to the boundary → cannot be replaced.
- The group of '0' in the middle is fully surrounded by 'X'.

Output:

```
X X X X
X X X X
X X X X
X X X X
X X 0 0
```

This example shows why **boundary-connected '0'** must be protected first.

---

## Approach

### Algorithm

1. Any '0' connected to the **boundary** is **safe** and cannot be replaced.

2. Start DFS from all boundary cells that contain '0'.
3. Mark all '0' cells reachable from the boundary as **visited**.
4. Traverse the entire matrix:
  - o If a cell is '0' and **not visited**, replace it with 'X'.
  - o Visited '0' remain unchanged.

This ensures only **fully surrounded regions** are replaced.

---

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
private:
 void dfs(int row, int col, vector<vector<int>>& vis,
 vector<vector<char>>& mat, int delrow[], int delcol[]) {
 vis[row][col] = 1;
 int n = mat.size(), m = mat[0].size();
 for(int k = 0; k < 4; k++) {
 int nrow = row + delrow[k];
 int ncol = col + delcol[k];
 if(nrow >= 0 && nrow < n && ncol >= 0 && ncol < m &&
 !vis[nrow][ncol] && mat[nrow][ncol] == '0') {
 dfs(nrow, ncol, vis, mat, delrow, delcol);
 }
 }
 }

public:
 vector<vector<char>> fill(int n, int m, vector<vector<char>> mat) {
 vector<vector<int>> vis(n, vector<int>(m, 0));
 int delrow[4] = {-1, 0, 1, 0};
 int delcol[4] = {0, 1, 0, -1};
 }
}
```

```

// first and last row
for(int j = 0; j < m; j++) {
 if(mat[0][j] == '0' && !vis[0][j])
 dfs(0, j, vis, mat, delrow, delcol);
 if(mat[n-1][j] == '0' && !vis[n-1][j])
 dfs(n-1, j, vis, mat, delrow, delcol);
}

// first and last column
for(int i = 0; i < n; i++) {
 if(mat[i][0] == '0' && !vis[i][0])
 dfs(i, 0, vis, mat, delrow, delcol);
 if(mat[i][m-1] == '0' && !vis[i][m-1])
 dfs(i, m-1, vis, mat, delrow, delcol);
}

// replace surrounded 0's
for(int i = 0; i < n; i++) {
 for(int j = 0; j < m; j++) {
 if(mat[i][j] == '0' && !vis[i][j])
 mat[i][j] = 'X';
 }
}
return mat;
}
};

int main() {
vector<vector<char>> mat{
 {'X', 'X', 'X', 'X'},
 {'X', '0', 'X', 'X'},
 {'X', '0', '0', 'X'},
 {'X', '0', 'X', 'X'},
 {'X', 'X', '0', '0'}
};
Solution ob;

```

```

 vector<vector<char>> ans = ob.fill(mat.size(), mat[0].size(),
mat);
 for(auto i : ans) {
 for(auto j : i) cout << j << " ";
 cout << "\n";
 }
 return 0;
 }

```

---

## Complexity Analysis

- **Time Complexity:**  $O(N \times M)$   
Reason: Each cell is visited at most once in DFS and once in final traversal.
- **Space Complexity:**  $O(N \times M)$   
Reason: Visited matrix and recursion stack in the worst case.

# 14. Number of Enclaves

You are given an  $N \times M$  binary grid:

- 0 represents **sea**
- 1 represents **land**

A move is allowed **only between adjacent land cells (up, down, left, right)** or you can walk **out of the boundary** of the grid.

A **land cell is called an enclave if it is not possible to walk off the boundary starting from that cell**, no matter how many moves you make.

Your task is to **count the number of such land cells**.

---

## Example

### Example 1

Input:

```
0 0 0 0
1 0 1 0
0 1 1 0
0 0 0 0
```

Explanation:

- Land cells connected to the boundary are **not enclaves**
- The land cells (1, 2), (2, 1), (2, 2) are **completely surrounded**
- From these cells, we cannot reach the boundary

Output:

3

|   |   |   |   |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 |
| 0 | 0 | 0 | 0 |

This example shows that **only land cells not connected to the boundary are counted**.

---

# Approach

## Algorithm

The key observation is:

- **Any land cell connected to the boundary cannot be an enclave**

Steps:

1. Create a **visited** matrix to track visited land cells.
2. Traverse all **boundary cells**:
  - If a boundary cell is land (1), mark it visited and push it into a queue.
3. Perform **BFS** from these boundary land cells:
  - Move in 4 directions
  - Mark all reachable land cells as visited
4. After BFS:
  - Traverse the grid
  - Count land cells (1) that are **not visited**
5. Return the count

Unvisited land cells are exactly the **enclaves**.

---

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
```

```

public:
 int numberOfEnclaves(vector<vector<int>>& grid) {
 int n = grid.size();
 int m = grid[0].size();

 vector<vector<int>> vis(n, vector<int>(m, 0));
 queue<pair<int,int>> q;

 // push all boundary land cells
 for(int i = 0; i < n; i++) {
 for(int j = 0; j < m; j++) {
 if(i == 0 || j == 0 || i == n-1 || j == m-1) {
 if(grid[i][j] == 1) {
 vis[i][j] = 1;
 q.push({i, j});
 }
 }
 }
 }

 int delrow[] = {-1, 0, 1, 0};
 int delcol[] = {0, 1, 0, -1};

 // BFS from boundary land cells
 while(!q.empty()) {
 auto it = q.front();
 q.pop();
 int row = it.first;
 int col = it.second;

 for(int k = 0; k < 4; k++) {
 int nrow = row + delrow[k];
 int ncol = col + delcol[k];

 if(nrow >= 0 && nrow < n && ncol >= 0 && ncol < m &&
 !vis[nrow][ncol] && grid[nrow][ncol] == 1) {
 vis[nrow][ncol] = 1;
 q.push({nrow, ncol});
 }
 }
 }
 }
}

```

```

 }
 }
}

// count enclaves
int cnt = 0;
for(int i = 0; i < n; i++) {
 for(int j = 0; j < m; j++) {
 if(grid[i][j] == 1 && vis[i][j] == 0)
 cnt++;
 }
}
return cnt;
}

int main() {
 vector<vector<int>> grid{
 {0,0,0,0},
 {1,0,1,0},
 {0,1,1,0},
 {0,0,0,0}
 };
 Solution obj;
 cout << obj.numberofEnclaves(grid);
 return 0;
}

```

---

## Complexity Analysis

- **Time Complexity:**  $O(N \times M)$   
Reason: Each cell is visited at most once during BFS and counting.
- **Space Complexity:**  $O(N \times M)$   
Reason: Visited matrix and BFS queue can store all cells in the worst case.

# 15. Word Ladder - I : G-29

Given two **distinct words** startWord and targetWord, and a list wordList containing **unique words of equal length**, find the **length of the shortest transformation sequence** from startWord to targetWord.

Rules for transformation:

- Only **one character** can be changed at a time.
  - Each transformed word **must exist in wordList**, including targetWord.
  - All words contain **only lowercase letters**.
  - startWord may or may not be present in wordList.
  - If no such transformation is possible, return 0.
- 

## Example

### Example 1

Input:

```
wordList = {"des", "der", "dfr", "dgt", "dfs"}
startWord = "der"
targetWord = "dfs"
```

Explanation:

"der" → "dfr" → "dfs"

- Change e to f → "dfr"
- Change r to s → "dfs"

All intermediate words exist in `wordList`.

Output:

3

---

## Example 2

Input:

```
wordList = {"geek", "gefk"}
startWord = "gedk"
targetWord = "geek"
```

Explanation:

"gedk" → "geek"

- Change d to e

Output:

2

These examples show that we must find the **minimum number of valid transformations**.

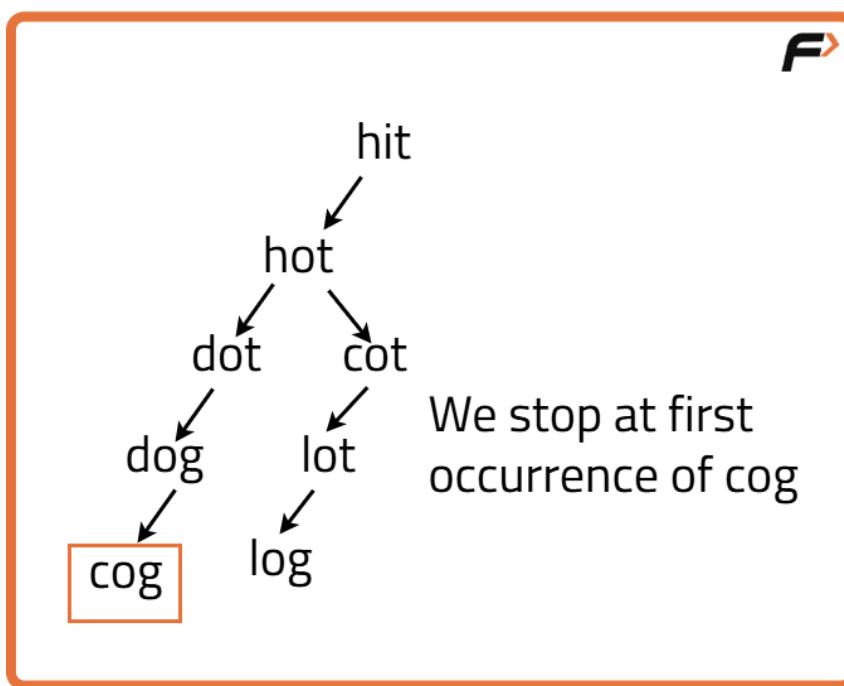
---

## Approach

This problem can be solved using **Breadth First Search (BFS)** because BFS always finds the **shortest path**.

1. Treat each word as a **node**.
2. Two words are connected if they differ by **exactly one character**.

3. Push {startWord, 1} into a queue where 1 is the starting step.
4. Store all words from wordList in an unordered\_set for fast lookup.
5. While the queue is not empty:
  - o Pop the front word and its current step count.
  - o If the word equals targetWord, return the step count.
  - o For each character in the word:
    - Try replacing it with all characters from 'a' to 'z'.
    - If the new word exists in the set:
      - Remove it from the set (to avoid revisiting).
      - Push it into the queue with steps + 1.
6. If BFS ends without reaching targetWord, return 0.



## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int wordLadderLength(string startWord, string targetWord,
vector<string>& wordList) {
 queue<pair<string,int>> q;
 q.push({startWord, 1});

 unordered_set<string> st(wordList.begin(), wordList.end());
 st.erase(startWord);

 while(!q.empty()) {
 string word = q.front().first;
 int steps = q.front().second;
 q.pop();

 if(word == targetWord) return steps;

 for(int i = 0; i < word.size(); i++) {
 char original = word[i];
 for(char ch = 'a'; ch <= 'z'; ch++) {
 word[i] = ch;
 if(st.find(word) != st.end()) {
 st.erase(word);
 q.push({word, steps + 1});
 }
 }
 word[i] = original;
 }
 }
 return 0;
 }
};

int main() {
```

```

vector<string> wordList = {"des", "der", "dfr", "dgt", "dfs"};
string startWord = "der";
string targetWord = "dfs";

Solution obj;
cout << obj.wordLadderLength(startWord, targetWord, wordList);
return 0;
}

```

---

## Complexity Analysis

- **Time Complexity:**  $O(N \times L \times 26)$   
Reason: For each word, we try changing all L characters to 26 possible letters.
- **Space Complexity:**  $O(N \times L)$   
Reason: Space used by the unordered\_set and BFS queue.

# 16. G-30 : Word Ladder-II

Given two **distinct words** `startWord` and `targetWord`, and a list `wordList` containing **unique words of equal length**, find **all the shortest transformation sequences** from `startWord` to `targetWord`.

Rules:

- Only **one letter** can be changed at a time.
- Each transformed word **must exist in wordList**, including `targetWord`.
- All words contain **only lowercase letters**.
- `startWord` may or may not be present in `wordList`.

- If no valid transformation exists, return an **empty list**.

**Ek BFS level ke andar words delete nahi karte  
Level complete hone ke baad delete karte hain**

---

## Examples

### Example 1

Input:

```
startWord = "der"
targetWord = "dfs"
wordList = {"des", "der", "dfr", "dgt", "dfs"}
```

Explanation:

Shortest transformation length = **3**

All shortest sequences are:

```
der → des → dfs
der → dfr → dfs
```

So the output is:

```
[["der", "dfr", "dfs"], ["der", "des", "dfs"]]
```

This example shows that **multiple shortest paths** can exist.

---

### Example 2

Input:

```
startWord = "gedk"
targetWord = "geek"
wordList = {"geek", "gefk"}
```

Explanation:

gedk → geek

Only one shortest sequence exists.

Output:

[ ["gedk", "geek"] ]

---

## Algorithm

This problem is an extension of **Word Ladder I**.

Instead of finding only the length, we must find **all shortest transformation sequences**.

We use **BFS with paths**:

1. Insert all words from wordList into an unordered\_set for fast lookup.
2. Use a **queue of vector<string>**, where each vector represents a transformation sequence.
3. Push {startWord} as the initial sequence into the queue.
4. Maintain:
  - level → current BFS depth
  - usedOnLevel → words used at the current level
5. While the queue is not empty:
  - Pop a sequence.
  - If the sequence length increases, remove all words used in the previous level from the set.
  - Take the **last word** of the sequence.

- If it equals `targetWord`:
  - Store it if it is the first found sequence
  - Or if its length equals already found shortest sequences
- Try changing each character of the word from 'a' to 'z':
  - If the new word exists in the set:
    - Add it to the sequence
    - Push the new sequence into the queue
    - Mark the word as used on this level

6. After BFS ends, return all stored sequences.

BFS guarantees that we only collect **shortest sequences**.

---

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 vector<vector<string>> findSequences(string beginWord, string endWord,
 vector<string>& wordList) {
 unordered_set<string> st(wordList.begin(), wordList.end());
 queue<vector<string>> q;
 q.push({beginWord});

 vector<string> usedOnLevel;
 usedOnLevel.push_back(beginWord);
 int level = 0;

 vector<vector<string>> ans;
```

```

while(!q.empty()) {
 vector<string> vec = q.front();
 q.pop();

 if(vec.size() > level) {
 level++;
 for(auto it : usedOnLevel)
 st.erase(it);
 usedOnLevel.clear();
 }

 string word = vec.back();

 if(word == endWord) {
 if(ans.size() == 0)
 ans.push_back(vec);
 else if(ans[0].size() == vec.size())
 ans.push_back(vec);
 }

 for(int i = 0; i < word.size(); i++) {
 char original = word[i];
 for(char c = 'a'; c <= 'z'; c++) {
 word[i] = c;
 if(st.count(word)) {
 vec.push_back(word);
 q.push(vec);
 usedOnLevel.push_back(word);
 vec.pop_back();
 }
 }
 word[i] = original;
 }
}

return ans;
}
};

```

```

bool comp(vector<string> a, vector<string> b) {
 string x = "", y = "";
 for(string i : a) x += i;
 for(string i : b) y += i;
 return x < y;
}

int main() {
 vector<string> wordList = {"des", "der", "dfr", "dgt", "dfs"};
 string startWord = "der", targetWord = "dfs";

 Solution obj;
 vector<vector<string>> ans = obj.findSequences(startWord,
targetWord, wordList);

 if(ans.size() == 0)
 cout << -1 << endl;
 else {
 sort(ans.begin(), ans.end(), comp);
 for(auto seq : ans) {
 for(auto word : seq)
 cout << word << " ";
 cout << endl;
 }
 }
 return 0;
}

```

---

## Complexity Analysis

- **Time Complexity:**

$$O(N \times L \times 26 + S \times L)$$

Reason:

- For each word, we try  $L \times 26$  transformations

- S shortest sequences of length L are stored

- **Space Complexity:**

$$O(N \times L + S \times L)$$

Reason:

- Queue stores full paths
- Set stores unused words
- Result stores all shortest sequences

## 17. Number of Islands

Given a grid of size  $N \times M$  consisting of:

- '1' → Land
- '0' → Water

Find the **number of islands**.

An island is formed by connecting **adjacent land cells ('1')** in **all 8 directions**:

- Up, Down, Left, Right
- All 4 diagonals

Water ('0') separates islands.

---

# **Example**

## **Example 1**

Input:

```
1 1 0 0 0
1 1 0 0 0
0 0 1 0 0
0 0 0 1 1
```

Explanation:

- Top-left block of 1s → Island 1
- Single 1 in middle → Island 2
- Bottom-right connected 1s → Island 3

Output:

3

---

## **Example 2**

Input:

```
1 1 1
1 1 1
1 1 1
```

Explanation:

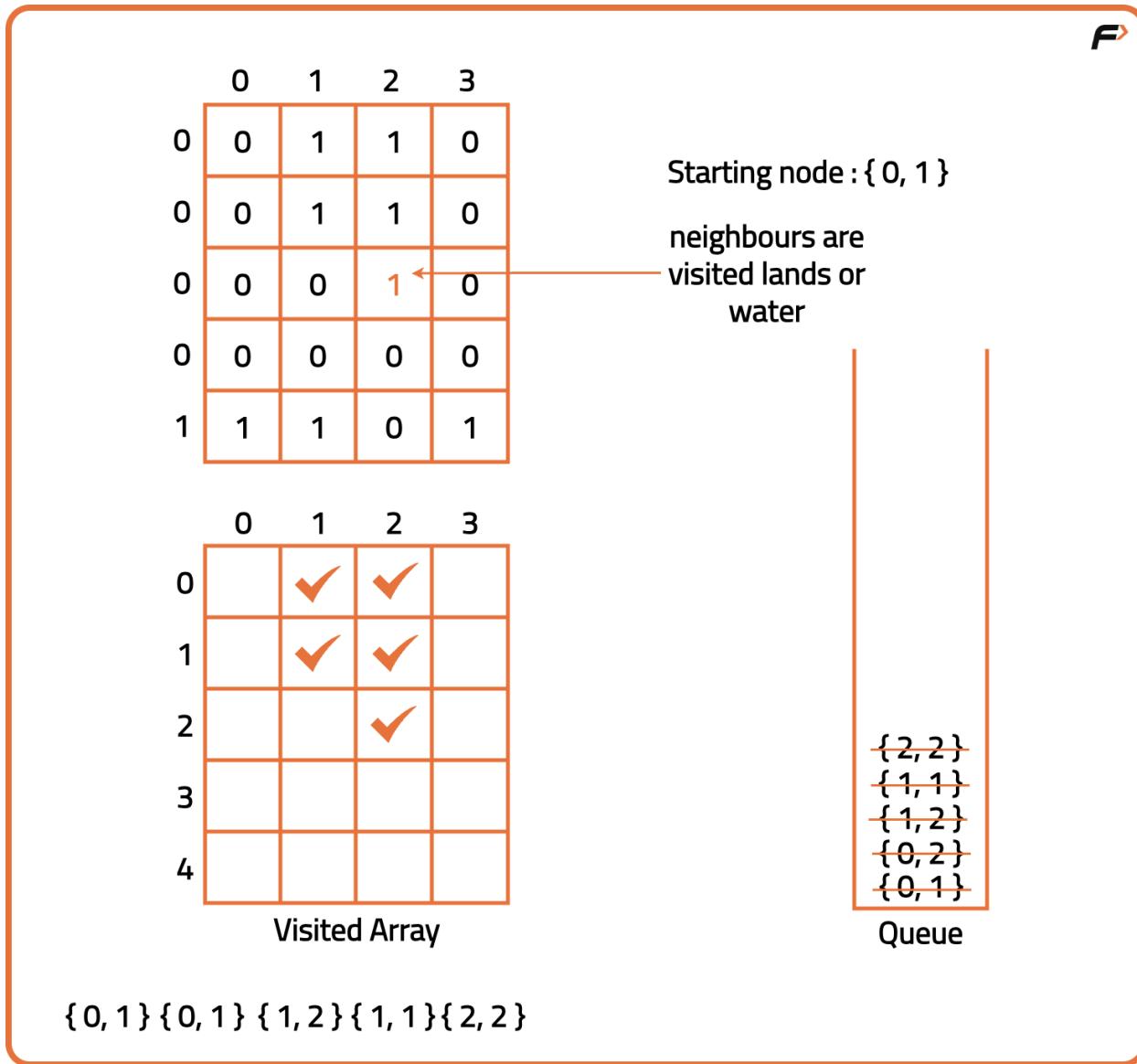
- All land cells are connected (including diagonals)

Output:

1

These examples show that **diagonal connections also matter**.

## Approach



## Algorithm

An island is a **connected component** of land cells ('1') considering **8-directional connectivity**.

We use **Breadth First Search (BFS)**.

1. Create a **visited** matrix to mark visited cells.
  2. Traverse the entire grid.
  3. Whenever an **unvisited land cell ('1')** is found:
    - o Increment island count.
    - o Start BFS from that cell.
  4. BFS will:
    - o Visit all land cells connected in 8 directions.
    - o Mark them as visited.
  5. Continue scanning the grid.
  6. The number of BFS calls equals the number of islands.
- 

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 void bfs(int row, int col, vector<vector<int>>& vis,
 vector<vector<char>>& grid) {

 queue<pair<int,int>> q;
 q.push({row, col});
 vis[row][col] = 1;

 int delRow[] = {-1,-1,-1,0,1,1,1,0};
 int delCol[] = {-1,0,1,1,1,0,-1,-1};
```

```

 while(!q.empty()) {
 int r = q.front().first;
 int c = q.front().second;
 q.pop();

 for(int i = 0; i < 8; i++) {
 int nrow = r + delRow[i];
 int ncol = c + delCol[i];

 if(nrow >= 0 && nrow < grid.size() &&
 ncol >= 0 && ncol < grid[0].size() &&
 !vis[nrow][ncol] &&
 grid[nrow][ncol] == '1') {

 vis[nrow][ncol] = 1;
 q.push({nrow, ncol});
 }
 }
 }

 int numIslands(vector<vector<char>>& grid) {
 int n = grid.size();
 int m = grid[0].size();
 vector<vector<int>> vis(n, vector<int>(m, 0));
 int count = 0;

 for(int i = 0; i < n; i++) {
 for(int j = 0; j < m; j++) {
 if(!vis[i][j] && grid[i][j] == '1') {
 count++;
 bfs(i, j, vis, grid);
 }
 }
 }
 return count;
 }
 };
}

```

```

int main() {
 vector<vector<char>> grid = {
 {'1', '1', '0', '0', '0'},
 {'1', '1', '0', '0', '0'},
 {'0', '0', '1', '0', '0'},
 {'0', '0', '0', '1', '1'}
 };

 Solution obj;
 cout << obj.numIslands(grid);
 return 0;
}

```

---

## Complexity Analysis

- **Time Complexity:**  $O(N \times M)$   
Reason: Every cell is visited at most once.
- **Space Complexity:**  $O(N \times M)$   
Reason: Visited matrix and BFS queue in the worst case.

# 18. Bipartite Graph | DFS Implementation

Given an **undirected graph** with  $V$  vertices (0-based indexing) represented using an **adjacency list**, check whether the graph is **bipartite**.

A graph is called **bipartite** if we can color all vertices using **only two colors** such that **no two adjacent vertices have the same color**.

---

## Example 1

Input:

```
0 --- 1
| |
2 --- 3
```

Explanation:

- We can color the graph using 2 colors:
  - Color nodes  $\{0, 2\}$  with color 0
  - Color nodes  $\{1, 3\}$  with color 1
- No adjacent nodes have the same color

Output:

```
1
```

---

## Example 2

Input:

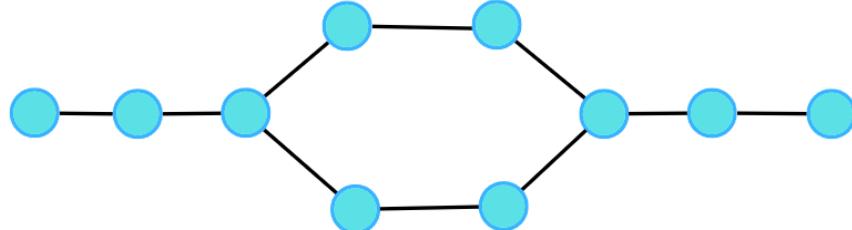
```
0 --- 2
| \ |
3 --- 1
```

Explanation:

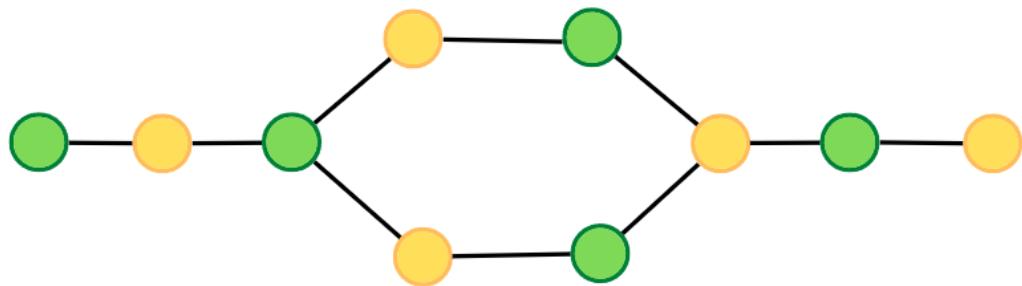
- This graph contains an **odd-length cycle**
- While coloring, two adjacent nodes are forced to have the same color
- Hence, it cannot be bipartite

Output:

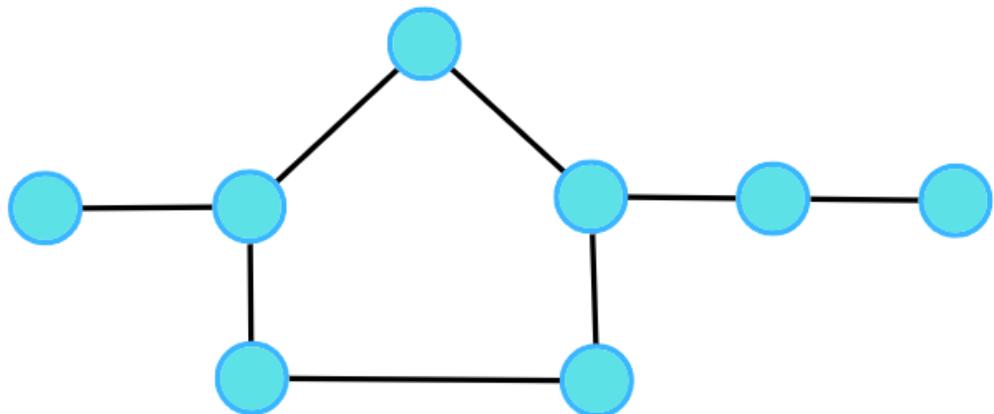
0



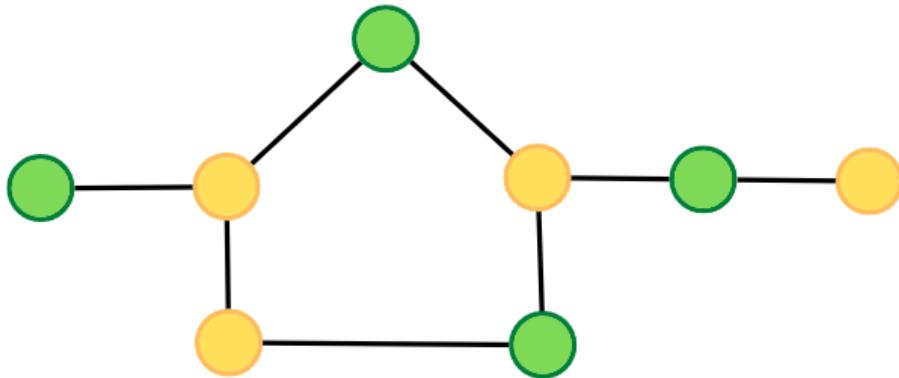
Output 1



Input



Output 0



These examples show that **graphs with odd cycles are not bipartite**, while graphs without odd cycles can be bipartite.

---

## Approach

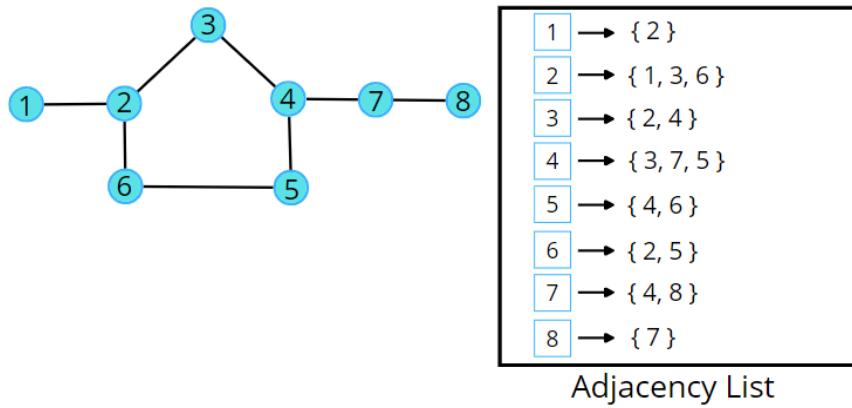
### Algorithm

We use **Depth First Search (DFS)** with coloring.

1. Create a color array of size  $V$ , initialize all values to  $-1$  (uncolored).
2. Since the graph can have **multiple connected components**, iterate over all vertices.
3. For every uncolored vertex:
  - o Start DFS and assign it color  $0$ .
4. In DFS:
  - o Assign the current node the given color.
  - o Traverse all adjacent nodes:
    - If an adjacent node is uncolored, assign it the **opposite color** and continue DFS.
    - If an adjacent node is already colored with the **same color**, return false.
5. If DFS completes successfully for all components, return true.

DFS ensures we try to color the graph consistently. Any conflict means the graph is **not bipartite**.

---



## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
private:
 bool dfs(int node, int col, int color[], vector<int> adj[]) {
 color[node] = col;
 for(auto it : adj[node]) {
 if(color[it] == -1) {
 if(!dfs(it, !col, color, adj))
 return false;
 }
 else if(color[it] == col) {
 return false;
 }
 }
 return true;
 }
}
```

```

public:
 bool isBipartite(int V, vector<int> adj[]) {
 int color[V];
 for(int i = 0; i < V; i++) color[i] = -1;

 for(int i = 0; i < V; i++) {
 if(color[i] == -1) {
 if(!dfs(i, 0, color, adj))
 return false;
 }
 }
 return true;
 }

void addEdge(vector<int> adj[], int u, int v) {
 adj[u].push_back(v);
 adj[v].push_back(u);
}

int main() {
 vector<int> adj[4];
 addEdge(adj, 0, 2);
 addEdge(adj, 0, 3);
 addEdge(adj, 2, 3);
 addEdge(adj, 3, 1);

 Solution obj;
 bool ans = obj.isBipartite(4, adj);
 if(ans) cout << "1\n";
 else cout << "0\n";
 return 0;
}

```

---

## Complexity Analysis

- **Time Complexity:**  $O(V + 2E)$   
Reason: Each vertex and each edge is visited once during DFS.
- **Space Complexity:**  $O(V)$   
Reason: Space used by color array and recursion stack.

## 19. Detect cycle in a directed graph (using DFS) : G-19

Given a **directed graph** with V vertices and E edges, check whether the graph contains **any cycle** or not.

In a directed graph, a cycle exists **only if during traversal we visit a node again on the same DFS path.**

---

### Example 1

Input:

8 → 9 → 10 → 8

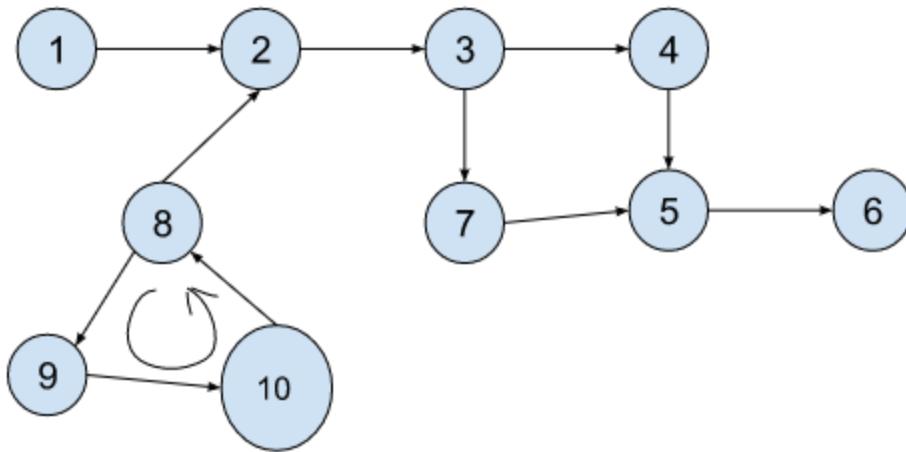
Explanation:

- Starting from node 8, we reach 9, then 10, and again come back to 8
- The node 8 is revisited on the **same path**

Output:

true

Example:



$N = 10, E = 11$

**Output:** true

**Explanation:**  $8 \rightarrow 9 \rightarrow 10$  is a cycle.

---

## Example 2

Input:

$1 \rightarrow 2 \rightarrow 3 \rightarrow 4$

Explanation:

- No node is revisited on the same DFS path
- The graph has no cycle

Output:

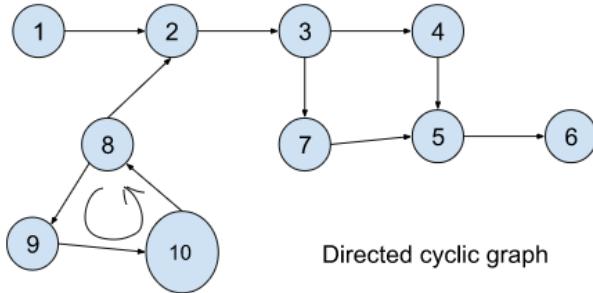
false

These examples show that **revisiting a node via a different path is not a cycle**, but revisiting it **on the same path is**.

---

# Approach

## Algorithm



To detect a cycle in a **directed graph**, we use **DFS with two visited arrays**.

1. Create two arrays:
  - `vis[ ]` → marks nodes that are completely visited
  - `pathVis[ ]` → marks nodes visited in the **current DFS path**
2. Traverse the graph component-wise using DFS.
3. In DFS for a node:
  - Mark it as visited in both `vis` and `pathVis`.
4. For every adjacent node:
  - **Case 1:** If the node is not visited, call DFS recursively.
  - **Case 2:** If the node is visited **and also marked in `pathVis`**, a cycle exists → return true.
  - **Case 3:** If the node is visited but not in `pathVis`, ignore it.
5. After exploring all neighbors:
  - Unmark the node from `pathVis` (backtracking).
6. If no cycle is found in any DFS call, return false.

The key idea is:

👉 Cycle exists only if a node is revisited on the same DFS path.

---

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
private:
 bool dfsCheck(int node, vector<int> adj[], int vis[], int
pathVis[]) {
 vis[node] = 1;
 pathVis[node] = 1;

 for(auto it : adj[node]) {
 if(!vis[it]) {
 if(dfsCheck(it, adj, vis, pathVis))
 return true;
 }
 else if(pathVis[it]) {
 return true;
 }
 }

 pathVis[node] = 0;
 return false;
 }

public:
 bool isCyclic(int V, vector<int> adj[]) {
 int vis[V] = {0};
 int pathVis[V] = {0};

 for(int i = 0; i < V; i++) {
 if(!vis[i]) {
 if(dfsCheck(i, adj, vis, pathVis))
 return true;
 }
 }
 }
}
```

```

 }
 }
 return false;
}
};

int main() {
 vector<int> adj[11] = {
 {}, {2}, {3}, {4,7}, {5}, {6}, {}, {5}, {9}, {10}, {8}
 };
 int V = 11;

 Solution obj;
 bool ans = obj.isCyclic(V, adj);

 if(ans) cout << "True\n";
 else cout << "False\n";

 return 0;
}

```

---

## Complexity Analysis

- **Time Complexity:**  $O(V + E)$   
Reason: Each node and edge is visited once during DFS.
- **Space Complexity:**  $O(V)$   
Reason: Two visited arrays and recursion stack space.

# 20. Topological Sort Algorithm | DFS : G-21

Given a **Directed Acyclic Graph (DAG)** with  $V$  vertices numbered from 0 to  $V-1$ , represented using an **adjacency list**, find **any valid topological ordering** of the graph.

In **topological sorting**, for every directed edge  $u \rightarrow v$ , node  $u$  must appear **before** node  $v$  in the ordering.

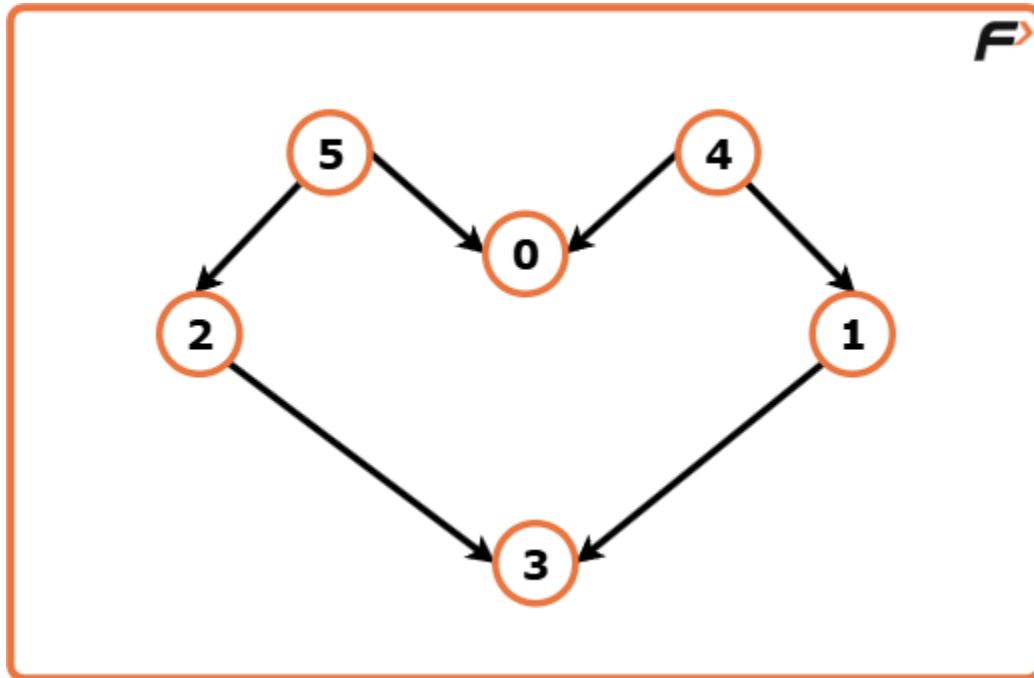
---

## Examples

### Example 1

Input:

$V = 6$   
 $\text{adj} = [[], [], [3], [1], [0,1], [0,2]]$



One valid Output:

[5, 4, 2, 3, 1, 0]

Explanation:

The ordering satisfies all edge conditions:

- $5 \rightarrow 0 \rightarrow 5$  comes before 0
- $4 \rightarrow 0 \rightarrow 4$  comes before 0
- $5 \rightarrow 2 \rightarrow 5$  comes before 2
- $2 \rightarrow 3 \rightarrow 2$  comes before 3
- $3 \rightarrow 1 \rightarrow 3$  comes before 1
- $4 \rightarrow 1 \rightarrow 4$  comes before 1

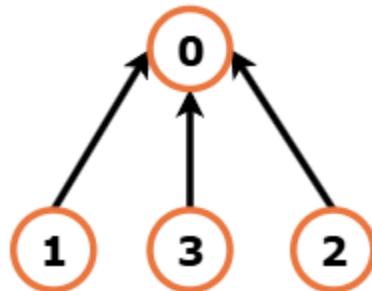
Multiple answers are possible for a DAG.

---

## Example 2

Input:

```
V = 4
adj = [[], [0], [0], [0]]
```



Output:

[3, 2, 1, 0]

Explanation:

- 1 → 0, 2 → 0, 3 → 0
- All nodes {1, 2, 3} must appear before 0

---

## Approach 1: Using DFS (Depth First Search)

### Algorithm

Topological sorting using DFS works on the idea of **finishing times**.

1. Create a visited array of size V, initialized to 0.
2. Create a stack to store nodes.

3. For every vertex:

- If it is not visited, call DFS on it.

4. In DFS:

- Mark the current node as visited.
- For every adjacent node:
  - If not visited, call DFS recursively.
- After visiting all adjacent nodes, push the current node into the stack.

5. After DFS is done for all vertices:

- Pop elements from the stack to get the topological order.

Pushing a node after exploring all its dependencies ensures correct ordering.

---

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 void dfs(int node, vector<int> adj[], vector<int>& vis,
 stack<int>& st) {
 vis[node] = 1;
 for(auto it : adj[node]) {
 if(!vis[it]) {
 dfs(it, adj, vis, st);
 }
 }
 st.push(node);
 }

 vector<int> topoSort(int V, vector<int> adj[]) {
```

```

vector<int> vis(V, 0);
stack<int> st;

for(int i = 0; i < V; i++) {
 if(!vis[i]) {
 dfs(i, adj, vis, st);
 }
}

vector<int> ans;
while(!st.empty()) {
 ans.push_back(st.top());
 st.pop();
}
return ans;
}

int main() {
 int V = 6;
 vector<int> adj[V];
 adj[5].push_back(0);
 adj[5].push_back(2);
 adj[4].push_back(0);
 adj[4].push_back(1);
 adj[2].push_back(3);
 adj[3].push_back(1);

 Solution obj;
 vector<int> res = obj.topoSort(V, adj);

 for(auto it : res) cout << it << " ";
 return 0;
}

```

- **Time Complexity:**  $O(V + E)$

Reason: Each vertex and each edge is visited exactly once during DFS.

- **Space Complexity:**  $O(V + E)$

Reason: Adjacency list, visited array, recursion stack, and result stack.

---

## Approach 2: Using BFS (Kahn's Algorithm)

### Algorithm

This approach is based on **in-degrees** of vertices.

1. Create an `indegree` array of size  $V$ , initialized to 0.
2. For every edge  $u \rightarrow v$ , increment `indegree[v]`.
3. Push all vertices with `indegree = 0` into a queue.
4. While the queue is not empty:
  - Pop a node and add it to the result.
  - For all its adjacent nodes:
    - Decrease their in-degree by 1.
    - If in-degree becomes 0, push into the queue.
5. The order in which nodes are removed from the queue gives a topological sort.

This works because nodes with zero in-degree have no dependencies.

---

### Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 vector<int> topologicalSort(int V, vector<int> adj[]) {
```

```

vector<int> indegree(V, 0);

for(int i = 0; i < V; i++) {
 for(auto it : adj[i]) {
 indegree[it]++;
 }
}

queue<int> q;
for(int i = 0; i < V; i++) {
 if(indegree[i] == 0)
 q.push(i);
}

vector<int> topo;
while(!q.empty()) {
 int node = q.front();
 q.pop();
 topo.push_back(node);

 for(auto it : adj[node]) {
 indegree[it]--;
 if(indegree[it] == 0)
 q.push(it);
 }
}
return topo;
};

int main() {
 int V = 6;
 vector<int> adj[V];
 adj[5].push_back(0);
 adj[5].push_back(2);
 adj[4].push_back(0);
 adj[4].push_back(1);
 adj[2].push_back(3);
}

```

```

adj[3].push_back(1);

Solution obj;
vector<int> ans = obj.topologicalSort(V, adj);

for(auto it : ans) cout << it << " ";
return 0;
}

```

## Complexity Analysis

- **Time Complexity:**  $O(V + E)$   
Reason: Each vertex is processed once and each edge reduces indegree once.
- **Space Complexity:**  $O(V + E)$   
Reason: Adjacency list, indegree array, queue, and result vector.

# 21. Kahn's Algorithm | Topological Sort Algorithm | BFS : G-22

Given a **Directed Acyclic Graph (DAG)** with V vertices and E edges, find **any valid topological sorting** of the graph.

In topological sorting, for every directed edge  $u \rightarrow v$ , node u must always appear **before** node v in the ordering.

---

## Examples

### Example 1

Input:

V = 6

```
adj = [[], [], [3], [1], [0,1], [0,2]]
```

Explanation:

Edges present:

```
5 → 0
5 → 2
4 → 0
4 → 1
2 → 3
3 → 1
```

One valid topological ordering:

```
[5, 4, 2, 3, 1, 0]
```

Another valid ordering:

```
[4, 5, 2, 3, 1, 0]
```

Multiple answers are possible because the graph is a DAG.

---

## Example 2

Input:

```
V = 4
adj = [[], [0], [0], [0]]
```

Explanation:

Edges present:

```
1 → 0
2 → 0
3 → 0
```

All nodes  $\{1, 2, 3\}$  must come before 0.

One valid output:

[3, 2, 1, 0]

---

## Approach

### Algorithm (Kahn's Algorithm – BFS)

Kahn's Algorithm uses **Breadth First Search (BFS)** and **in-degree counting**.

1. Create an array `inDegree[ ]` to store the number of incoming edges for each node.
2. Traverse the adjacency list and compute in-degrees of all nodes.
3. Push all nodes with `inDegree = 0` into a queue (these nodes have no dependencies).
4. While the queue is not empty:
  - o Pop a node from the queue and add it to the answer.
  - o For every adjacent node:
    - Decrease its in-degree by 1.
    - If in-degree becomes 0, push it into the queue.
5. The order in which nodes are removed from the queue forms a valid topological sort.

This works because a node can only be processed **after all its parent nodes are processed**.

---

## Code

```
#include <bits/stdc++.h>
using namespace std;
```

```

class Solution {
public:
 vector<int> topoSort(int V, vector<int> adj[]) {
 vector<int> ans;
 vector<int> inDegree(V, 0);

 // Calculate in-degree of each node
 for(int i = 0; i < V; i++) {
 for(auto it : adj[i]) {
 inDegree[it]++;
 }
 }

 // Queue for BFS
 queue<int> q;

 // Push nodes with in-degree 0
 for(int i = 0; i < V; i++) {
 if(inDegree[i] == 0)
 q.push(i);
 }

 // BFS
 while(!q.empty()) {
 int node = q.front();
 q.pop();
 ans.push_back(node);

 for(auto it : adj[node]) {
 inDegree[it]--;
 if(inDegree[it] == 0)
 q.push(it);
 }
 }

 return ans;
 }
};

```

```

int main() {
 int V = 6;
 vector<int> adj[V] = {
 {},
 {},
 {3},
 {1},
 {0,1},
 {0,2}
 };

 Solution obj;
 vector<int> ans = obj.topoSort(V, adj);

 for(int x : ans)
 cout << x << " ";

 return 0;
}

```

---

## Complexity Analysis

**Time Complexity:**  $O(V + E)$

Reason:

- Each vertex is processed once
- Each edge is processed once while reducing in-degrees

**Space Complexity:**  $O(V)$

Reason:

- In-degree array and queue can store up to  $V$  nodes

# 22. Detect a Cycle in a Directed Graph

Given a **directed graph** with  $V$  vertices labeled from 0 to  $V-1$ , represented using an **adjacency list**, determine whether the graph contains **any cycle**.

A cycle exists if we can start from a node and come back to the same node by following the **direction of edges**.

---

## Examples

### Example 1

Input:

```
V = 6
adj = [[1], [2, 5], [3], [4], [1], []]
```

Explanation:

$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 1$

We can start from node 1 and return back to 1 following directed edges.

So, a **cycle exists**.

Output:

True

---

### Example 2

Input:

```
V = 4
adj = [[1,2], [2], [], [0,2]]
```

Explanation:

- No path forms a closed loop following edge directions.
- Every path ends without coming back to the starting node.

Output:

False

These examples show that **not all directed graphs have cycles**, and direction of edges matters.

---

## Approach

### Algorithm (Using BFS – Kahn's Algorithm)

A cycle in a directed graph can be detected using **Kahn's Algorithm**, which is based on **Topological Sorting**.

Key idea:

- A **Directed Acyclic Graph (DAG)** always has a topological ordering containing **all vertices**.
- If a graph contains a cycle, **some vertices will never get in-degree 0**, so they will never be processed.

Steps:

1. Create an **indegree array** to store the number of incoming edges for each vertex.
2. Traverse the adjacency list and compute indegree of every vertex.

3. Push all vertices with `indegree = 0` into a queue.
  4. Initialize a counter `count = 0` to count processed nodes.
  5. While the queue is not empty:
    - o Pop a node from the queue.
    - o Increment count.
    - o For each neighbor of this node:
      - Decrease its indegree by 1.
      - If indegree becomes 0, push it into the queue.
  6. After BFS:
    - o If `count == V`, all nodes were processed → **no cycle**.
    - o If `count < V`, some nodes were not processed → **cycle exists**.
- 

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 bool isCyclic(int V, vector<vector<int>>& adj) {
 vector<int> indegree(V, 0);

 // Calculate indegree of each vertex
 for(int i = 0; i < V; i++) {
 for(auto it : adj[i]) {
 indegree[it]++;
 }
 }
 }
}
```

```

queue<int> q;

// Push all vertices with indegree 0
for(int i = 0; i < V; i++) {
 if(indegree[i] == 0)
 q.push(i);
}

int count = 0;

// BFS using Kahn's Algorithm
while(!q.empty()) {
 int node = q.front();
 q.pop();
 count++;

 for(auto it : adj[node]) {
 indegree[it]--;
 if(indegree[it] == 0)
 q.push(it);
 }
}

// If all nodes are not processed, cycle exists
return count != V;
}

};

int main() {
 int V = 6;
 vector<vector<int>> adj = {
 {1}, {2, 5}, {3}, {4}, {1}, {}
 };

 Solution obj;
 if(obj.isCyclic(V, adj))
 cout << "True";
 else

```

```
 cout << "False";

 return 0;
}
```

---

## Complexity Analysis

**Time Complexity:**  $O(V + E)$

Reason:

- Each vertex is processed once.
- Each edge is considered once while updating indegrees.

**Space Complexity:**  $O(V + E)$

Reason:

- Adjacency list stores edges.
- Indegree array and queue take up to  $O(V)$  space.

# 23. Course Schedule I and II | Pre-requisite Tasks | Topological Sort : G-24

There are tasks (or courses) labeled from 0 to N-1.

Some tasks depend on other tasks and must be completed **after** their prerequisites.

A prerequisite pair [ a, b ] means:

- **Task b must be completed before task a**

These problems are solved using **Topological Sort on a Directed Graph**.

---

## Examples

### Example 1 (Possible)

Input:

```
N = 4
P = 3
prerequisites = {{1,0}, {2,1}, {3,2}}
```

Explanation:

$0 \rightarrow 1 \rightarrow 2 \rightarrow 3$

One valid order:

3 2 1 0

All tasks can be completed.

Output:

Yes

---

### Example 2 (Not Possible)

Input:

```
N = 4
P = 4
prerequisites = {{1,2}, {4,3}, {2,4}, {4,1}}
```

Explanation:

$1 \rightarrow 2 \rightarrow 4 \rightarrow 1$  (cycle)

Because of the cycle, tasks depend on each other circularly.

Output:

No

---

## Course Schedule I

(Check if it is possible to finish all tasks)

### Approach 1

#### Algorithm

The problem reduces to **cycle detection in a directed graph**.

If the graph has a cycle, tasks cannot be finished.

If the graph is a **DAG**, tasks can be finished.

We use **Kahn's Algorithm (BFS Topological Sort)**:

1. Treat each task as a node.
2. Create a directed edge  $b \rightarrow a$  for prerequisite  $[a, b]$ .
3. Compute inDegree for every task.
4. Push all tasks with inDegree = 0 into a queue.
5. Process the queue:
  - o Remove a task.
  - o Reduce inDegree of its neighbors.
  - o Push neighbors whose inDegree becomes 0.
6. Count processed tasks.
7. If processed count == N  $\rightarrow$  possible, else not possible.

---

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 bool canFinish(int numCourses, vector<vector<int>>& prerequisites)
{
 vector<vector<int>> adj(numCourses);
 vector<int> inDegree(numCourses, 0);

 for(auto &p : prerequisites) {
 int a = p[0], b = p[1];
 adj[b].push_back(a);
 inDegree[a]++;
 }

 queue<int> q;
 for(int i = 0; i < numCourses; i++) {
 if(inDegree[i] == 0) q.push(i);
 }

 int cnt = 0;
 while(!q.empty()) {
 int node = q.front(); q.pop();
 cnt++;

 for(int nei : adj[node]) {
 inDegree[nei]--;
 if(inDegree[nei] == 0)
 q.push(nei);
 }
 }

 return cnt == numCourses;
}
```

```

};

int main() {
 Solution sol;
 vector<vector<int>> prerequisites = {{1,0}, {0,1}};
 int numCourses = 2;
 cout << (sol.canFinish(numCourses, prerequisites) ? "true" :
"false");
 return 0;
}

```

---

## Complexity Analysis

- **Time Complexity:**  $O(V + E)$   
Each task and dependency is processed once.
  - **Space Complexity:**  $O(V + E)$   
Adjacency list, inDegree array, queue.
- 

## Course Schedule II

(Return one valid order of tasks)

### Approach 2

#### Algorithm

This is an extension of Course Schedule I.

Instead of only checking feasibility, we **store the topological order**.

Steps are the same as Kahn's Algorithm:

1. Build graph and inDegree array.
2. Push all nodes with inDegree = 0 into queue.

3. While queue is not empty:
    - o Pop node and add it to order.
    - o Reduce inDegree of neighbors.
  4. If order size == N, return order.
  5. Otherwise, return empty array (cycle exists).
- 

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 vector<int> findOrder(int numCourses, vector<vector<int>>& prerequisites) {
 vector<vector<int>> adj(numCourses);
 vector<int> inDegree(numCourses, 0);

 for(auto &p : prerequisites) {
 int a = p[0], b = p[1];
 adj[b].push_back(a);
 inDegree[a]++;
 }

 queue<int> q;
 for(int i = 0; i < numCourses; i++) {
 if(inDegree[i] == 0) q.push(i);
 }

 vector<int> order;
 while(!q.empty()) {
 int node = q.front(); q.pop();
 order.push_back(node);

```

```

 for(int nei : adj[node]) {
 inDegree[nei]--;
 if(inDegree[nei] == 0)
 q.push(nei);
 }
 }

 if(order.size() == numCourses)
 return order;
 return {};
}
};

int main() {
 Solution sol;
 vector<vector<int>> prerequisites = {{1,0},{2,0},{3,1},{3,2}};
 int numCourses = 4;

 vector<int> ans = sol.findOrder(numCourses, prerequisites);
 for(int x : ans) cout << x << " ";
 return 0;
}

```

---

## Complexity Analysis

- **Time Complexity:**  $O(V + E)$
- **Space Complexity:**  $O(V + E)$

# 24. Find Eventual Safe States – BFS – Topological Sort : G-25

Given a **directed graph** with  $V$  vertices labeled from 0 to  $V-1$ , represented using an **adjacency list**, find all **eventual safe nodes**.

- A **terminal node** is a node with **no outgoing edges**.
  - A **safe node** is a node from which **every possible path** eventually leads to a terminal node.
  - If a node is part of a **cycle** or can reach a cycle, it is **not safe**.
  - Return all safe nodes in **ascending order**.
- 

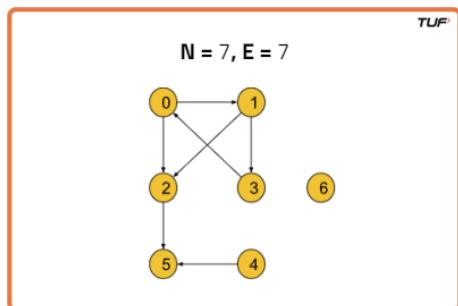
## Examples

### Example 1

Input:

$N = 7$

Edges = 7



Explanation (conceptual):

- There is a cycle:  $0 \rightarrow 1 \rightarrow 3 \rightarrow 0$

- Node 7 leads into this cycle
- Nodes 0, 1, 3, 7 are **not safe**
- Nodes 2, 4, 5, 6 eventually lead to terminal nodes

Output:

{2, 4, 5, 6}

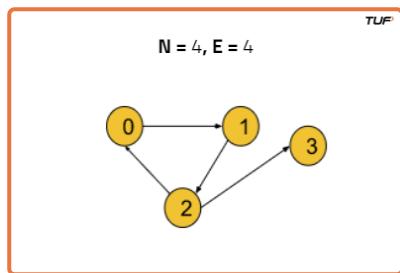
---

## Example 2

Input:

$N = 4$

Edges = 4



Explanation:

- Only node 3 has no outgoing edges
- All other nodes either form or lead into a cycle

Output:

{3}

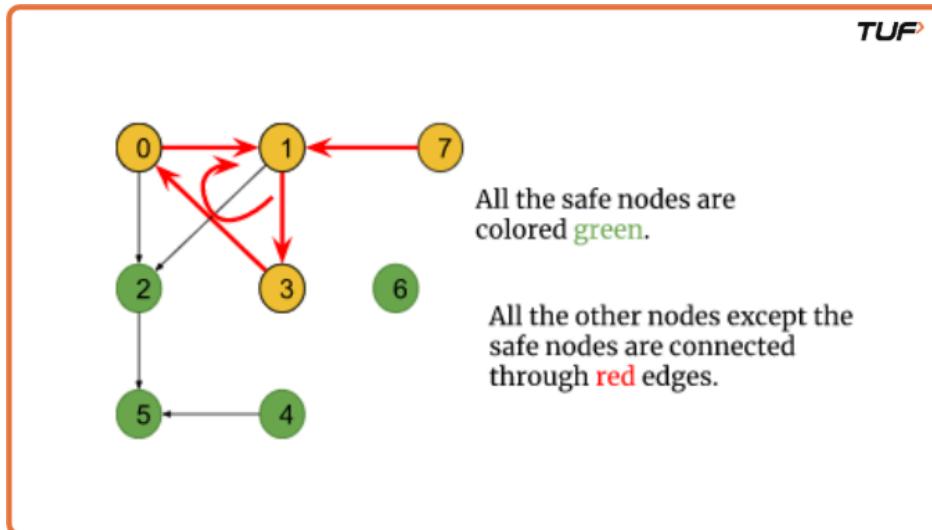
These examples show that **nodes involved in cycles or leading to cycles are unsafe**.

---

# Approach

## Algorithm (BFS + Topological Sort)

The idea is to **eliminate all nodes that are part of cycles or lead to cycles**.



Key observations:

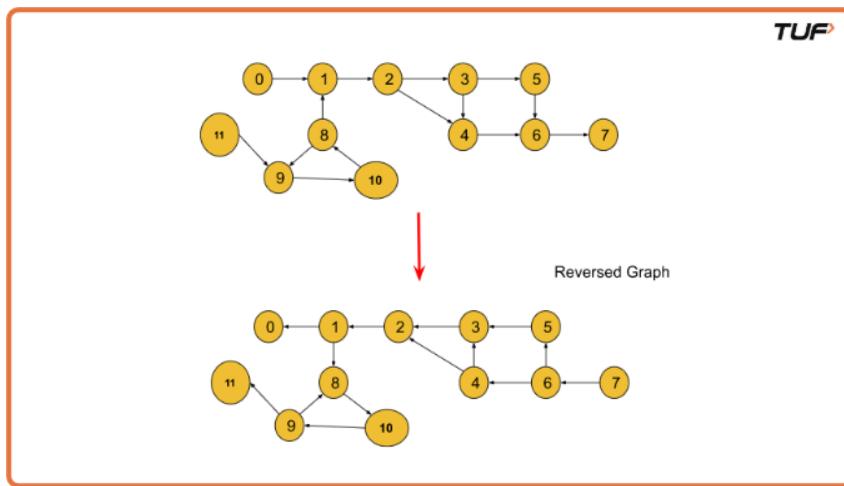
- Terminal nodes (outdegree = 0) are always **safe**.
- If a node points only to safe nodes, it is also safe.
- Nodes in cycles will never become terminal.

To use **topological sort logic**, we do the following:

1. **Reverse the graph:**
  - Original edge:  $u \rightarrow v$
  - Reversed edge:  $v \rightarrow u$
2. In the reversed graph:
  - Terminal nodes become nodes with **indegree = 0**.
3. Create an **indegree[ ]** array based on the reversed graph.

4. Push all nodes with **indegree = 0** into a queue.
5. Perform BFS:
  - o Pop a node → it is safe.
  - o For each neighbor in the reversed graph:
    - Decrease its indegree.
    - If indegree becomes 0, push it into the queue.
6. All nodes processed by BFS are **safe nodes**.
7. Sort the result as required.

Nodes that are part of cycles will **never reach indegree 0**, so they are excluded automatically.



## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 vector<int> eventualSafeNodes(int V, vector<int> adj[]) {
 vector<int> adjRev[V];
 for (int i = 0; i < V; i++) {
 for (int j : adj[i]) {
 adjRev[j].push_back(i);
 }
 }
 queue<int> q;
 for (int i = 0; i < V; i++) {
 if (adj[i].size() == 0) {
 q.push(i);
 }
 }
 while (!q.empty()) {
 int node = q.front();
 q.pop();
 for (int neighbor : adjRev[node]) {
 adjRev[neighbor].erase(
 find(adjRev[neighbor].begin(), adjRev[neighbor].end(), node));
 if (adjRev[neighbor].size() == 0) {
 q.push(neighbor);
 }
 }
 }
 vector<int> result;
 for (int i = 0; i < V; i++) {
 if (adjRev[i].size() == 0) {
 result.push_back(i);
 }
 }
 return result;
 }
};
```

```

vector<int> indegree(V, 0);

// Reverse the graph and compute indegree
for(int i = 0; i < V; i++) {
 for(auto it : adj[i]) {
 adjRev[it].push_back(i);
 indegree[i]++;
 }
}

queue<int> q;
vector<int> safeNodes;

// Push all nodes with indegree 0
for(int i = 0; i < V; i++) {
 if(indegree[i] == 0)
 q.push(i);
}

// BFS traversal
while(!q.empty()) {
 int node = q.front();
 q.pop();
 safeNodes.push_back(node);

 for(auto it : adjRev[node]) {
 indegree[it]--;
 if(indegree[it] == 0)
 q.push(it);
 }
}

sort(safeNodes.begin(), safeNodes.end());
return safeNodes;
}

int main() {

```

```

vector<int> adj[12] = {
 {1}, {2}, {3,4}, {4,5}, {6}, {6}, {7}, {},
 {1,9}, {10}, {8}, {9}
};

int V = 12;

Solution obj;
vector<int> ans = obj.eventualSafeNodes(V, adj);

for(int x : ans)
 cout << x << " ";
return 0;
}

```

---

## Time Complexity

$$O(V + E) + O(N \log N)$$

- $O(V + E)$  for BFS traversal
- $O(N \log N)$  for sorting safe nodes

## Space Complexity

$$O(V + E)$$

- Reversed adjacency list
- Indegree array
- Queue for BFS

# 25. Alien Dictionary – Topological Sort : G-26

You are given a **sorted dictionary of an alien language**.

Your task is to **determine the order of characters** in that alien language.

- The dictionary contains N words.
- Each word is made of characters from an alien language.
- There are exactly K unique characters (assumed to be the first K lowercase letters).
- The dictionary is already sorted according to the alien language rules.

You need to return **one valid ordering of the characters**.

---

## Example 1

Input:

```
N = 5, K = 4
dict = {"baa", "abcd", "abca", "cab", "cad"}
```

Explanation (compare consecutive words):

- "baa" vs "abcd" → b comes before a
- "abcd" vs "abca" → d comes before a
- "abca" vs "cab" → a comes before c
- "cab" vs "cad" → b comes before d

So one valid alien character order is:

b d a c

---

## Example 2

Input:

```
N = 3, K = 3
dict = {"caa", "aaa", "aab"}
```

Explanation:

- "caa" vs "aaa" → c comes before a
- "aaa" vs "aab" → a comes before b

So one valid order is:

c a b

These examples show that **character order is derived by comparing adjacent words.**

---

## Approach

### Algorithm

This problem can be solved using **Topological Sort (BFS / Kahn's Algorithm)**.

#### Step 1: Build the Graph

- Treat each character as a node.
- Compare every pair of **adjacent words**.
- For the first position where characters differ:
  - If  $s1[i] \neq s2[i]$ , then add a directed edge  
 $s1[i] \rightarrow s2[i]$

- Break, because only the first mismatch matters.

### **Step 2: Topological Sort (BFS)**

- Create an `indegree` array for all  $K$  characters.
- Push all characters with `indegree = 0` into a queue.
- While the queue is not empty:
  - Pop a character and add it to the result.
  - Reduce `indegree` of its neighbors.
  - If any neighbor's `indegree` becomes 0, push it into the queue.

### **Step 3: Final Answer**

- Convert numeric nodes back to characters.
- Return the ordering as a string.

This works because the character dependency graph is a **Directed Acyclic Graph (DAG)**.

---

## **Code**

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
private:
 vector<int> topoSort(int V, vector<int> adj[]) {
 vector<int> indegree(V, 0);

 for(int i = 0; i < V; i++) {
 for(auto it : adj[i]) {
 indegree[it]++;
 }
 }
 }
}
```

```

 }

 queue<int> q;
 for(int i = 0; i < V; i++) {
 if(indegree[i] == 0)
 q.push(i);
 }

 vector<int> topo;
 while(!q.empty()) {
 int node = q.front();
 q.pop();
 topo.push_back(node);

 for(auto it : adj[node]) {
 indegree[it]--;
 if(indegree[it] == 0)
 q.push(it);
 }
 }
 return topo;
}

public:
 string findOrder(string dict[], int N, int K) {
 vector<int> adj[K];

 for(int i = 0; i < N - 1; i++) {
 string s1 = dict[i];
 string s2 = dict[i + 1];
 int len = min(s1.size(), s2.size());

 for(int j = 0; j < len; j++) {
 if(s1[j] != s2[j]) {
 adj[s1[j] - 'a'].push_back(s2[j] - 'a');
 break;
 }
 }
 }
 }
}

```

```

 }

 vector<int> topo = topoSort(K, adj);

 string ans = "";
 for(auto it : topo) {
 ans += char(it + 'a');
 }
 return ans;
}

};

int main() {
 int N = 5, K = 4;
 string dict[] = {"baa", "abcd", "abca", "cab", "cad"};

 Solution obj;
 string res = obj.findOrder(dict, N, K);

 for(char c : res)
 cout << c << " ";
 return 0;
}

```

---

## Time Complexity

$$O(N * L) + O(K + E)$$

- $N * L$  for comparing adjacent words (up to first mismatch)
- $K + E$  for topological sort

## Space Complexity

$$O(K)$$

- Adjacency list
- Indegree array
- Queue
- Result array

## 26. Shortest Path in Undirected Graph with Unit Distance : G-28

---

### Problem Statement

Given an **undirected graph with unit weight edges**, find the **shortest distance from the source node (0) to all other nodes**.

- If a node is **unreachable**, return -1 for that node.
  - All edges have weight = 1.
- 

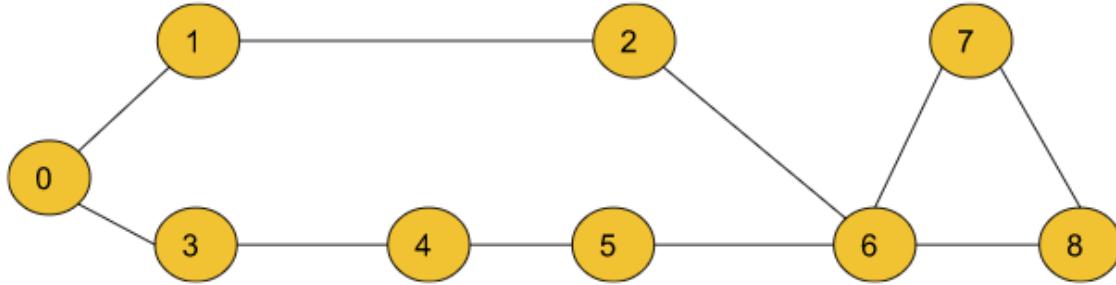
### Examples (Important)

#### Example 1

##### Input

n = 9, m = 10

```
edges = [[0,1],[0,3],[3,4],[4,5],[5,6],
 [1,2],[2,6],[6,7],[7,8],[6,8]]
src = 0
```



## Output

```
0 1 2 1 2 3 3 4 4
```

## Explanation

- Distance from  $0 \rightarrow 0 = 0$
- Distance from  $0 \rightarrow 1 = 1$
- Distance from  $0 \rightarrow 3 = 1$
- Distance from  $0 \rightarrow 2 = 2$  ( $0 \rightarrow 1 \rightarrow 2$ )
- Distance from  $0 \rightarrow 6 = 3$  ( $0 \rightarrow 1 \rightarrow 2 \rightarrow 6$ )
- Distance from  $0 \rightarrow 8 = 4$

This array represents the **shortest number of edges** needed to reach each node from source 0.

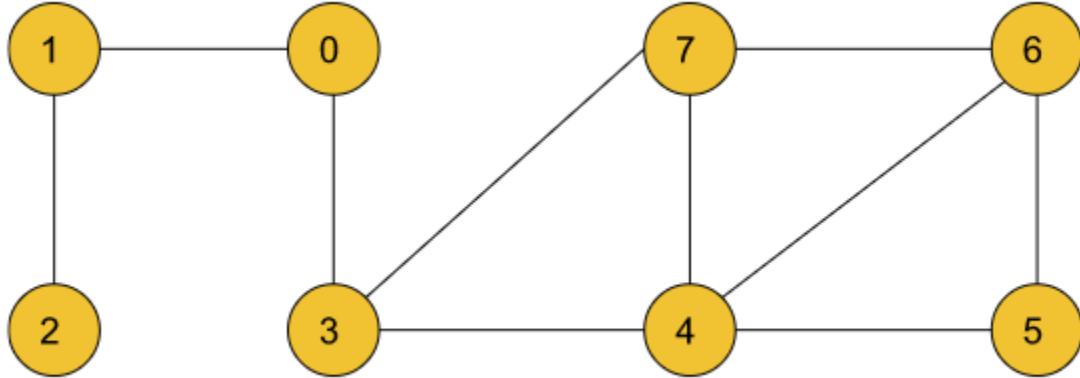
---

## Example 2

### Input

```
n = 8, m = 10
```

```
edges = [[1,0],[2,1],[0,3],[3,7],[3,4],
 [7,4],[7,6],[4,5],[4,6],[6,5]]
src = 0
```



## Output

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

## Explanation

- Nodes closer to source appear with smaller distances.
- If a node were unreachable, it would be -1.

---

## Approach

### Why BFS?

- The graph is **undirected**
- All edges have **unit weight (1)**
- **Breadth First Search (BFS)** always explores nodes level by level  
→ guarantees the **shortest path in terms of edges**

So **BFS** is optimal here (no need for Dijkstra).

---

## Algorithm

1. Convert edge list into an **adjacency list**
2. Create a `dist[ ]` array initialized with a large value (`1e9`)
3. Set `dist[src] = 0`
4. Push `src` into a queue
5. Perform BFS:
  - o For each popped node, explore its neighbors
  - o If `dist[node] + 1 < dist[neighbor]`
    - Update distance
    - Push neighbor into queue
6. After BFS:
  - o If distance is still `1e9`, mark it as `-1`

---

## Code (C++)

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 vector<int> shortestPath(vector<vector<int>>& edges, int N, int M,
 int src) {
```

```

// Adjacency list
vector<int> adj[N];
for (auto it : edges) {
 adj[it[0]].push_back(it[1]);
 adj[it[1]].push_back(it[0]);
}

// Distance array
vector<int> dist(N, 1e9);
dist[src] = 0;

// BFS queue
queue<int> q;
q.push(src);

while (!q.empty()) {
 int node = q.front();
 q.pop();

 for (auto it : adj[node]) {
 if (dist[node] + 1 < dist[it]) {
 dist[it] = dist[node] + 1;
 q.push(it);
 }
 }
}

// Convert unreachable nodes to -1
for (int i = 0; i < N; i++) {
 if (dist[i] == 1e9)
 dist[i] = -1;
}

return dist;
};

int main() {

```

```

int N = 9, M = 10;
vector<vector<int>> edges = {
 {0,1},{0,3},{3,4},{4,5},{5,6},
 {1,2},{2,6},{6,7},{7,8},{6,8}
};

Solution obj;
vector<int> ans = obj.shortestPath(edges, N, M, 0);

for (int x : ans)
 cout << x << " ";

return 0;
}

```

---

## Complexity Analysis

### Time Complexity

$O(N + M)$

- BFS visits every node once  $\rightarrow O(N)$
- Traverses every edge once  $\rightarrow O(M)$

### Space Complexity

$O(N + M)$

- Adjacency list
- Distance array
- BFS queue

# 27. Shortest Path in Directed Acyclic Graph (DAG) using Topological Sort :

## G-27

---

You are given a **Directed Acyclic Graph (DAG)** with

- **N vertices** numbered from 0 to N-1

**M directed edges**, where each edge is of the form

$u \rightarrow v$  with weight  $w$

- 

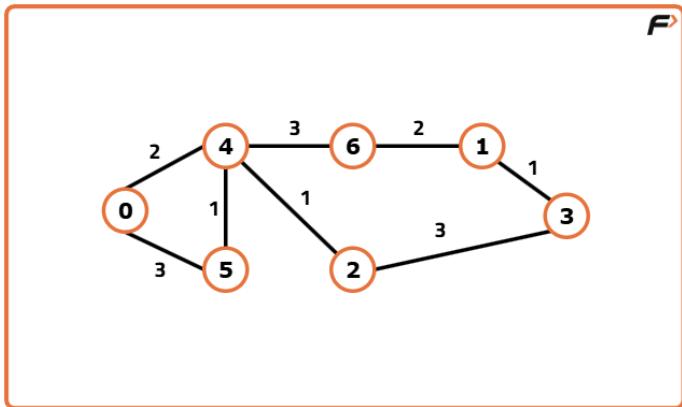
You are required to find the **shortest distance from the source vertex 0 to all other vertices**.

- If a vertex is **not reachable** from the source, return -1 for that vertex.
- 

### Example 1

#### Input

```
n = 7, m = 8
Edges = [
 [0,4,2], [0,5,3], [5,4,1], [4,6,3],
 [4,2,1], [6,1,2], [2,3,3], [1,3,1]
]
```



## Output

0 7 3 6 2 3 5

## Explanation

| <b>Node</b> | <b>Shortest Distance from 0</b> | <b>Path Example</b> |
|-------------|---------------------------------|---------------------|
| <b>0</b>    |                                 |                     |
| 0           | 0                               | source              |
| 4           | 2                               | 0 → 4               |
| 5           | 3                               | 0 → 5               |
| 2           | 3                               | 0 → 4 → 2           |
| 6           | 5                               | 0 → 4 → 6           |
| 1           | 7                               | 0 → 4 → 6 → 1       |
| 3           | 6                               | 0 → 4 → 2 → 3       |

All distances are computed using **directed edges only**.

---

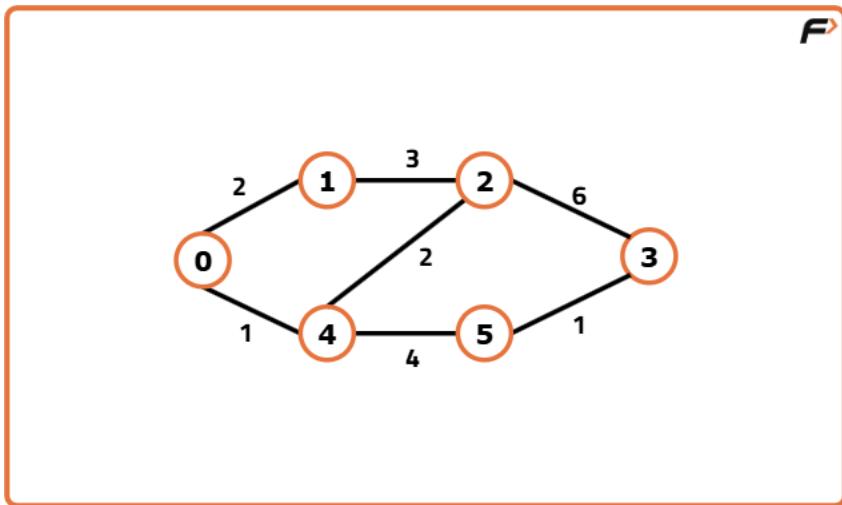
## Example 2

### Input

```

n = 6, m = 7
edges = [
 [0,1,2], [0,4,1], [4,5,4],
 [4,2,2], [1,2,3], [2,3,6], [5,3,1]
]

```



## Output

0 2 3 6 1 5

## Explanation

| Node | Shortest Distance |
|------|-------------------|
| 0    | 0                 |
| 1    | 2                 |
| 4    | 1                 |
| 2    | 3                 |
| 5    | 5                 |
| 3    | 6                 |

## Why Topological Sort Works Here

- The graph is a **DAG** → no cycles
- In a DAG, **topological order guarantees** that when we process a node, all nodes that can reach it have already been processed.
- This allows us to **relax edges only once**, unlike Dijkstra.

👉 Dijkstra is required only when cycles may exist.

---

## Approach

### Key Idea

If we already know the shortest distances to all nodes that come **before** a node in topological order, we can easily compute the shortest distance for the current node.

---

## Algorithm

### Build adjacency list

Store graph as:

```
adj[u] = { (v, weight) }
```

- 1.
2. **Topological Sort using DFS**

- Maintain a `visited[]` array
- Push nodes into a stack after DFS completion

### Initialize distance array

```
dist[i] = INF
dist[0] = 0
```

- 3.
4. **Process nodes in topological order**

- Pop nodes from stack

For each edge  $u \rightarrow v$  ( $wt$ ):

```
if dist[u] + wt < dist[v]
 dist[v] = dist[u] + wt
```

○

### 5. Convert unreachable nodes

- Replace INF with -1

## C++ Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
private:
 // DFS for Topological Sort
 void topoSort(int node, vector<pair<int,int>> adj[],
 vector<int>& vis, stack<int>& st) {

 vis[node] = 1;

 for (auto it : adj[node]) {
 int v = it.first;
 if (!vis[v]) {
 topoSort(v, adj, vis, st);
 }
 }
 st.push(node);
 }

public:
 vector<int> shortestPath(int N, int M, vector<vector<int>>& edges)
{
```

```

// Adjacency list
vector<pair<int,int>> adj[N];
for (auto &e : edges) {
 adj[e[0]].push_back({e[1], e[2]});
}

// Topological sort
vector<int> vis(N, 0);
stack<int> st;
for (int i = 0; i < N; i++) {
 if (!vis[i]) {
 topoSort(i, adj, vis, st);
 }
}

// Distance array
vector<int> dist(N, 1e9);
dist[0] = 0;

// Relax edges in topo order
while (!st.empty()) {
 int node = st.top();
 st.pop();

 if (dist[node] != 1e9) {
 for (auto it : adj[node]) {
 int v = it.first;
 int wt = it.second;
 if (dist[node] + wt < dist[v]) {
 dist[v] = dist[node] + wt;
 }
 }
 }
}

// Mark unreachable nodes
for (int i = 0; i < N; i++) {

```

```

 if (dist[i] == 1e9)
 dist[i] = -1;
 }

 return dist;
}
};

int main() {
 int N = 6, M = 7;
 vector<vector<int>> edges = {
 {0,1,2}, {0,4,1}, {4,5,4},
 {4,2,2}, {1,2,3}, {2,3,6}, {5,3,1}
 };

 Solution obj;
 vector<int> ans = obj.shortestPath(N, M, edges);

 for (int x : ans) cout << x << " ";
 return 0;
}

```

---

## Complexity Analysis

### Time Complexity

$O(N + M)$

- Topological sort  $\rightarrow O(N + M)$
- Edge relaxation  $\rightarrow O(N + M)$

### Space Complexity

$O(N + M)$

- Adjacency list
- Stack for topo sort
- Distance & visited arrays

## 28. Dijkstra's Algorithm (Using Set) : G-33

---

### Problem Statement

You are given a **weighted, undirected, and connected graph** with

- **V vertices**
- An **adjacency list** adj, where each entry contains {neighbor, weight}
- A **source vertex S**

Your task is to find the **shortest distance from the source vertex S to every other vertex**.

If a vertex is unreachable (not possible here since graph is connected), its distance would remain infinity.

#### Note:

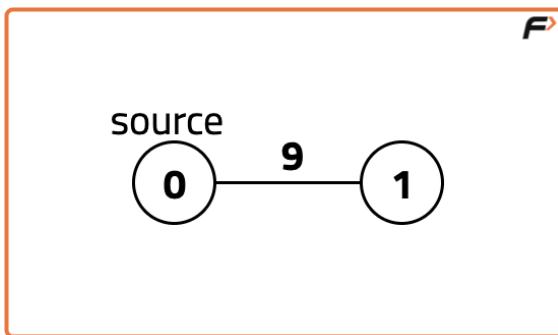
- The graph **does not contain negative weight edges**
- Dijkstra's Algorithm **fails** if negative weights exist

---

## Example 1

### Input

```
V = 2
adj = [
 {{1, 9}},
 {{0, 9}}
]
S = 0
```



### Output

```
0 9
```

### Explanation

- Distance from node 0 to itself = 0
- Distance from node 0 to node 1 = 9

---

## Example 2

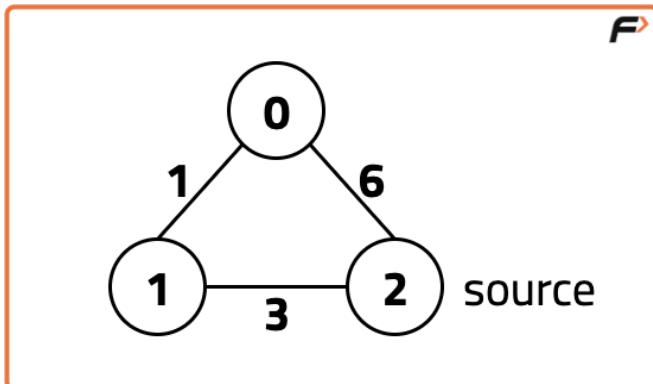
### Input

```
V = 3
adj = {
 {{1,1}, {2,6}},
```

```

 {{2,3}, {0,1}},
 {{1,3}, {0,6}}
}
s = 2

```



### Output

4 3 0

### Explanation

Shortest paths from source 2:

| Destinatio<br>n | Path      | Distanc<br>e |
|-----------------|-----------|--------------|
| 2               | 2         | 0            |
| 1               | 2 → 1     | 3            |
| 0               | 2 → 1 → 0 | 4            |

Direct path  $2 \rightarrow 0$  has weight 6, but  $2 \rightarrow 1 \rightarrow 0 = 3 + 1 = 4$ , which is shorter.

---

## Intuition

Dijkstra's Algorithm is a **greedy algorithm**:

- Always pick the node with the **smallest known distance**

- Try to improve (relax) distances of its neighbors
- Once a node is picked with minimum distance, that distance is **final**

Using a **set**:

- Keeps {distance, node} pairs in **sorted order**
  - Allows **deleting outdated entries**, unlike priority queue
- 

## Approach

### Key Data Structures

- **Distance Array**  
 $\text{dist}[i] \rightarrow$  shortest distance from source to node *i*
  - **Set {distance, node}**  
Always gives the node with the smallest distance
- 

1. Initialize:

- $\text{dist}[i] = \text{INF}$  for all nodes
- $\text{dist}[S] = 0$
- Insert  $\{0, S\}$  into the set

2. While the set is not empty:

- Extract the node with **minimum distance**
- For each adjacent node:
  - If  $\text{currentDist} + \text{edgeWeight} < \text{dist}[\text{adjNode}]$

- Remove old {`dist[adjNode]`, `adjNode`} from set (if exists)
  - Update `dist[adjNode]`
  - Insert updated pair into set
3. Return the distance array
- 

## C++ Code (Using Set)

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 vector<int> dijkstra(int V, vector<vector<int>> adj[], int S) {

 // Set stores {distance, node}
 set<pair<int,int>> st;

 // Distance array
 vector<int> dist(V, 1e9);

 // Source initialization
 dist[S] = 0;
 st.insert({0, S});

 while (!st.empty()) {
 auto it = *st.begin();
 st.erase(it);

 int node = it.second;
 int currDist = it.first;

 // Traverse neighbors
 for (auto nbr : adj[node]) {
 int adjNode = nbr[0];
 int newDist = currDist + nbr[1];

 if (newDist < dist[adjNode]) {
 dist[adjNode] = newDist;
 st.insert({newDist, adjNode});
 }
 }
 }

 return dist;
 }
}
```

```

 int wt = nbr[1];

 if (currDist + wt < dist[adjNode]) {

 // Remove old entry if exists
 if (dist[adjNode] != 1e9)
 st.erase({dist[adjNode], adjNode});

 // Update distance
 dist[adjNode] = currDist + wt;
 st.insert({dist[adjNode], adjNode});
 }
 }

 return dist;
}
};

int main() {
 int V = 3, S = 2;
 vector<vector<int>> adj[V];

 adj[0].push_back({1,1});
 adj[0].push_back({2,6});
 adj[1].push_back({2,3});
 adj[1].push_back({0,1});
 adj[2].push_back({1,3});
 adj[2].push_back({0,6});

 Solution obj;
 vector<int> ans = obj.dijkstra(V, adj, S);

 for (int x : ans) cout << x << " ";
 return 0;
}

```

---

## Complexity Analysis

### Time Complexity

$O(E \log V)$

- Each edge may cause insertion/deletion in set  $\rightarrow \log V$

### Space Complexity

$O(V + E)$

- Distance array + adjacency list + set

## 29. Dijkstra's Algorithm – Using Priority Queue : G-32

---

### Problem Statement

You are given a **weighted, undirected, and connected graph** with

- **V vertices**
- **E edges**
- **A source vertex S**

Your task is to find the **shortest distance from the source vertex S to all other vertices**.

**Note:**

- The graph does **not** contain negative weight edges or cycles
  - If a node is unreachable, its distance would be  $-1$  (not applicable here since graph is connected)
- 

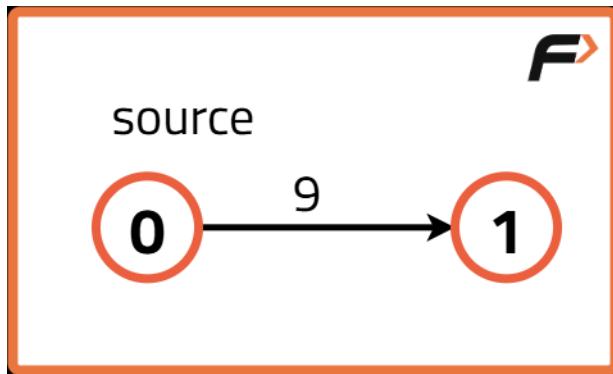
## Example 1

### Input

$V = 2, E = 1$

Edges:  $0 \rightarrow 1$

$S = 0$



### Output

$[0, 9]$

### Explanation

- Distance from source 0 to itself = 0
  - Distance from 0 to 1 = 9
- 

## Example 2

### Input

$V = 3, E = 3$

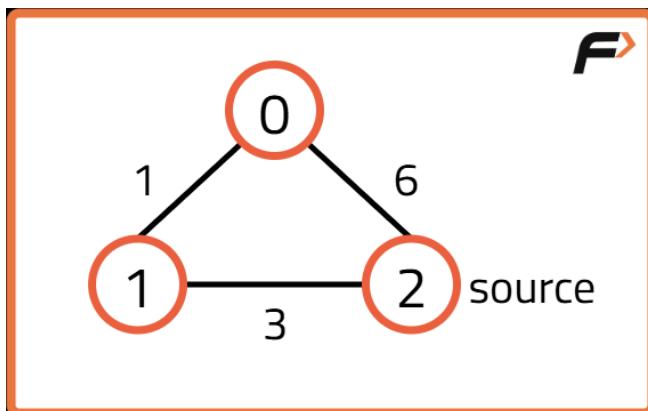
Edges:

2 --1--> 1

1 --3--> 0

2 --6--> 0

$S = 2$



## Output

[4, 3, 0]

## Explanation

Shortest paths from source 2:

| Node | Path      | Distance    |
|------|-----------|-------------|
| 2    | 2         | 0           |
| 1    | 2 → 1     | 1           |
| 0    | 2 → 1 → 0 | $1 + 3 = 4$ |

Direct path  $2 \rightarrow 0$  has weight 6, which is **not optimal**.

---

## Intuition

Dijkstra's Algorithm is a **greedy algorithm**:

- Always expand the **closest unprocessed node**
- Once a node is popped with minimum distance, that distance is **final**
- Works only when all edge weights are **non-negative**

Using a **Priority Queue (Min-Heap)** helps us:

- Efficiently extract the node with the **minimum distance**
  - Avoid scanning all vertices repeatedly
- 

## Algorithm

1. Create a `dist[ ]` array initialized with a very large value (INF)
2. Set `dist[source] = 0`
3. Use a **min-heap priority queue** storing {distance, node}
4. Push {0, source} into the queue
5. While the queue is not empty:
  - Pop the node with the smallest distance
  - If this distance is **greater than the stored distance**, skip it
  - For each adjacent node:
    - If `dist[curr] + weight < dist[adj]`
    - Update `dist[adj]`
    - Push `{dist[adj], adj}` into the queue
6. After completion, `dist[ ]` contains shortest distances from the source

---

## C++ Code (Priority Queue – Min Heap)

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 vector<int> dijkstra(int V, vector<vector<pair<int,int>>>& adj,
 int src) {

 // Distance array
 vector<int> dist(V, 1e9);

 // Min-heap {distance, node}
 priority_queue<
 pair<int,int>,
 vector<pair<int,int>>,
 greater<pair<int,int>>
 > pq;

 // Initialize source
 dist[src] = 0;
 pq.push({0, src});

 while (!pq.empty()) {
 auto top = pq.top();
 pq.pop();

 int node = top.second;
 int currDist = top.first;

 // Skip outdated entry
 if (currDist > dist[node]) continue;

 // Relax neighbors
 for (auto it : adj[node]) {
 int next = it.first;
```

```

 int wt = it.second;

 if (dist[node] + wt < dist[next]) {
 dist[next] = dist[node] + wt;
 pq.push({dist[next], next});
 }
 }
}

return dist;
}
};

int main() {
 int V = 5;
 vector<vector<pair<int,int>>> adj(V);

 adj[0].push_back({1, 2});
 adj[0].push_back({2, 4});
 adj[1].push_back({2, 1});
 adj[1].push_back({3, 7});
 adj[2].push_back({4, 3});
 adj[3].push_back({4, 2});

 Solution obj;
 vector<int> dist = obj.dijkstra(V, adj, 0);

 for (int i = 0; i < V; i++) {
 cout << dist[i] << " ";
 }
}

```

---

## Complexity Analysis

### Time Complexity

$$O((V + E) \log V)$$

- Each edge relaxation may push into heap  $\rightarrow \log V$

## Space Complexity

$O(V + E)$

- Distance array + adjacency list + priority queue

# 30. G-36: Shortest Distance in a Binary Maze

Given an  $n \times m$  binary grid where each cell is either 0 or 1. You are given a source cell and a destination cell. You can move only through cells having value 1. Movement is allowed only in four directions: up, down, left, and right.

Your task is to find the shortest distance from the source to the destination. If no path exists, return -1.

If the source and destination are the same cell, the distance is 0.

Example:

Grid

```
1 1 1 1
1 1 0 1
1 1 1 1
1 1 0 0
1 0 0 1
```

Source = (0,1), Destination = (2,2)

Shortest path is  $(0,1) \rightarrow (1,1) \rightarrow (2,1) \rightarrow (2,2)$

Total steps = 3

## Approach

### Algorithm

We use Breadth First Search because each move has equal cost and BFS always gives the shortest path.

1. If source and destination are the same, return 0.
2. Create a queue that stores {distance, {row, col}}.
3. Create a distance matrix and initialize all values to a very large number.
4. Set the source distance as 0 and push it into the queue.
5. Define direction arrays for up, right, down, and left.
6. While the queue is not empty:
  - o Pop the front element.
  - o For each of the 4 directions:
    - Check boundary conditions.
    - Check if the cell value is 1.
    - Check if the new distance is smaller than the stored distance.
    - Update distance and push the cell into the queue.
    - If destination is reached, return the distance.
7. If BFS ends and destination is not reached, return -1.

### Code

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
```

```

public:

 int shortestPath(vector<vector<int>> &grid, pair<int,int> source,
pair<int,int> destination) {

 if(source.first==destination.first &&
source.second==destination.second)

 return 0;

 int n=grid.size();

 int m=grid[0].size();

 vector<vector<int>> dist(n,vector<int>(m,1e9));

 queue<pair<int,pair<int,int>>> q;

 dist[source.first][source.second]=0;

 q.push({0,{source.first,source.second}});

 int dr[]={-1,0,1,0};

 int dc[]={0,1,0,-1};

 while(!q.empty()){

 auto it=q.front();

 q.pop();

 int dis=it.first;

 int r=it.second.first;

 int c=it.second.second;

 for(int i=0;i<4;i++){

 int nr=r+dr[i];

 int nc=c+dc[i];

 if(nr<0||nr>n-1||nc<0||nc>m-1||dist[nr][nc]<dis)
 continue;

 dist[nr][nc]=dis+1;

 q.push({dis+1,{nr,nc}});
 }
 }
 }
}

```

```

int r=it.second.first;

int c=it.second.second;

for(int i=0;i<4;i++){

 int newr=r+dr[i];

 int newc=c+dc[i];

 if(newr>=0 && newr<n && newc>=0 && newc<m &&

 grid[newr][newc]==1 && dis+1<dist[newr][newc]){

 dist[newr][newc]=dis+1;

 if(newr==destination.first &&
newc==destination.second)

 return dis+1;

q.push({dis+1,{newr,newc}});

 }

}

return -1;

};


```

```

int main(){

 vector<vector<int>> grid={{1,1,1,1},
 {1,1,0,1},
 {1,1,1,1},
 {1,1,0,0},
 {1,0,0,1}};

 pair<int,int> source={0,1};
 pair<int,int> destination={2,2};

 Solution obj;

 cout<<obj.shortestPath(grid,source,destination);

 return 0;
}

```

## Complexity Analysis

Time Complexity: **O(N × M)**

Each cell is visited at most once, and for every cell we check 4 directions.

Space Complexity: **O(N × M)**

Distance matrix and queue together take space proportional to the number of cells.

# 31. G-37: Path With Minimum Effort

You are given a 2D grid heights where each cell represents the height at that position. You start from the top-left cell  $(0, 0)$  and want to reach the bottom-right cell  $(n-1, m-1)$ .

You can move only in four directions: up, down, left, and right.

The effort of a path is defined as the **maximum absolute difference in heights between any two consecutive cells** in that path.

Your goal is to find a path such that this effort is minimum.

Example:

For heights =

1 2 2

3 8 2

5 3 5

One path is  $1 \rightarrow 3 \rightarrow 5 \rightarrow 3 \rightarrow 5$ .

The maximum height difference here is 2, which is smaller than other possible paths.

So the answer is 2.

---

## Approach

### Algorithm

This problem is solved using Dijkstra's algorithm because we want to minimize the maximum edge cost along a path.

1. Use a priority queue that stores {effort, {row, col}}.
2. Create a distance matrix and initialize all values with a large number.
3. Set distance of source cell  $(0, 0)$  as 0 and push it into the priority queue.
4. While the priority queue is not empty:
  - o Pop the cell with the minimum effort.
  - o If the destination cell is reached, return the effort.
  - o For all 4 directions:

- Check if the new cell is inside the grid.
  - Compute new effort as  
 $\max(\text{current effort}, \text{absolute height difference})$
  - If this effort is smaller than the stored value, update it and push into the queue.
5. If destination is not reached, return 0.

This works because the priority queue always processes the path with the least current effort first.

### Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int MinimumEffort(vector<vector<int>> &heights) {
 int n = heights.size();
 int m = heights[0].size();

 priority_queue<pair<int, pair<int, int>>,
 vector<pair<int, pair<int, int>>>,
 greater<pair<int, pair<int, int>>> pq;

 vector<vector<int>> dist(n, vector<int>(m, 1e9));
 dist[0][0]=0;
 pq.push({0, {0, 0}});

 int dr[]={-1, 0, 1, 0};
 int dc[]={0, 1, 0, -1};

 while(!pq.empty()){
 auto it=pq.top();
 pq.pop();
 int diff=it.first;
 int r=it.second.first;
```

```

 int c=it.second.second;

 if(r==n-1 && c==m-1)
 return diff;

 for(int i=0;i<4;i++){
 int newr=r+dr[i];
 int newc=c+dc[i];

 if(newr>=0 && newr<n && newc>=0 && newc<m){
 int
newEffort=max(abs(heights[r][c]-heights[newr][newc]),diff);
 if(newEffort<dist[newr][newc]){
 dist[newr][newc]=newEffort;
 pq.push({newEffort,{newr,newc}});
 }
 }
 }
 return 0;
 }
};

int main(){
 vector<vector<int>> heights={{1,2,2},{3,8,2},{5,3,5}};
 Solution obj;
 cout<<obj.MinimumEffort(heights);
 return 0;
}

```

## Complexity Analysis

**Time Complexity:**  $O(N * M * \log(N * M))$

Each cell can be inserted into the priority queue, and each insertion or deletion takes  $\log(N*M)$  time.

**Space Complexity:**  $O(N * M)$

Used for the distance matrix and the priority queue in the worst case.

# 32. G-38: Cheapest Flights Within K Stops

You are given  $n$  cities connected by flights. Each flight is represented as [from, to, price].

You are also given a source city  $\text{src}$ , a destination city  $\text{dst}$ , and an integer  $k$  which represents the maximum number of stops allowed.

You need to find the **minimum cost** to travel from  $\text{src}$  to  $\text{dst}$  using **at most  $k$  stops**.

If no such route exists, return -1.

Example:

Cities = 4

Flights = [0→1 (100), 1→3 (600), 1→2 (100), 2→3 (200)]

$\text{src} = 0$ ,  $\text{dst} = 3$ ,  $k = 1$

Path  $0 \rightarrow 1 \rightarrow 3$  has cost 700 and uses 1 stop, so it is valid.

Path  $0 \rightarrow 1 \rightarrow 2 \rightarrow 3$  is cheaper but uses 2 stops, so it is invalid.

Answer = 700.

---

## Approach

### Algorithm

This approach uses **BFS with stop control**.

1. Create an adjacency list from the flights array.
2. Use a queue that stores {stops, {node, cost}}.
3. Create a distance array initialized with a very large value.
4. Push the source node with 0 stops and 0 cost into the queue.
5. While the queue is not empty:
  - o Pop the front element.
  - o If the number of stops is greater than  $k$ , skip it.

- Traverse all adjacent nodes.
- If a cheaper cost is found and stops are within limit:
  - Update the distance.
  - Push the adjacent node with incremented stop count.

6. After traversal:

- If destination cost is still infinite, return -1.
- Otherwise return the stored minimum cost.

This works because BFS ensures we explore paths level by level based on stops.

**Code**

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int CheapestFLight(int n, vector<vector<int>> &flights,
 int src, int dst, int K) {

 vector<pair<int,int>> adj[n];
 for(auto it: flights){
 adj[it[0]].push_back({it[1], it[2]});
 }

 queue<pair<int,pair<int,int>>> q;
 q.push({0,{src,0}});

 vector<int> dist(n,1e9);
 dist[src]=0;

 while(!q.empty()){
 auto it=q.front();
 q.pop();
 int stops=it.first;
 int node=it.second.first;
 int cost=it.second.second;
 if(stops>K)
 continue;
 for(auto it: adj[node]){
 int adjNode=it.first;
 int adjCost=it.second;
 if(cost+adjCost<dist[adjNode] && stops<=K)
 dist[adjNode]=cost+adjCost;
 }
 }
 if(dist[dst]==1e9)
 return -1;
 else
 return dist[dst];
 }
}
```

```

 int node=it.second.first;
 int cost=it.second.second;

 if(stops>K) continue;

 for(auto iter: adj[node]){
 int adjNode=iter.first;
 int edW=iter.second;

 if(cost+edW<dist[adjNode] && stops<=K){
 dist[adjNode]=cost+edW;
 q.push({stops+1, {adjNode, cost+edW}});
 }
 }
 }

 if(dist[dst]==1e9) return -1;
 return dist[dst];
}

int main(){
 int n=4, src=0, dst=3, K=1;
 vector<vector<int>>
flights={{0,1,100},{1,2,100},{2,0,100},{1,3,600},{2,3,200}};
 Solution obj;
 cout<<obj.CheapestFlight(n,flights,src,dst,K);
 return 0;
}

```

## Complexity Analysis

**Time Complexity:**  $O(E)$

Each flight edge is processed at most once during BFS traversal.

**Space Complexity:**  $O(E + V)$

Adjacency list stores all flights, and the distance array stores minimum cost for each city.

# 33. Network Delay Time

You are given a directed weighted graph with  $n$  nodes numbered from 1 to  $n$ . Each edge  $(u, v, w)$  means a signal can go from node  $u$  to node  $v$  in  $w$  units of time.

A signal starts from node  $k$  at time 0. Whenever a node receives the signal, it immediately forwards it to all its outgoing neighbors.

You need to find the **minimum time** required for **all nodes** to receive the signal.

If any node cannot be reached from  $k$ , return -1.

Example:

times = [[2,1,1],[2,3,1],[3,4,1]],  $n = 4$ ,  $k = 2$

From node 2:

- Node 1 receives signal in 1 unit
- Node 4 receives signal via  $2 \rightarrow 3 \rightarrow 4$  in 2 units

Answer = 2

---

## Approach

### Algorithm

This problem is a shortest path problem on a directed graph with non-negative weights, so we use **Dijkstra's Algorithm**.

1. Create an adjacency list where each node stores its outgoing edges and weights.
2. Create a distance array initialized to infinity for all nodes.
3. Set distance of source node  $k$  to 0.
4. Use a min-heap (priority queue) storing {time, node}.
5. Push  $\{0, k\}$  into the priority queue.
6. While the priority queue is not empty:
  - Pop the node with the smallest current time.

- For each neighbor:
    - If going through the current node gives a smaller time, update the distance and push it into the queue.
7. After processing all reachable nodes:
- Find the maximum value in the distance array.
  - If any node is still unreachable (infinity), return -1.
  - Otherwise, return the maximum time.

Dijkstra works here because all edge weights are non-negative and we want minimum arrival times.

### Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int networkDelayTime(vector<vector<int>>& times, int n, int k) {
 vector<vector<pair<int,int>>> adj(n+1);
 for(auto &t: times){
 adj[t[0]].push_back({t[1], t[2]});
 }

 priority_queue<pair<int,int>, vector<pair<int,int>>, greater<>> pq;
 vector<int> dist(n+1, INT_MAX);

 dist[k]=0;
 pq.push({0,k});

 while(!pq.empty()){
 auto it=pq.top();
 pq.pop();
 int time=it.first;
 int node=it.second;

```

```

 for(auto &nbr: adj[node]){
 int v=nbr.first;
 int wt=nbr.second;
 if(dist[v]>time+wt){
 dist[v]=time+wt;
 pq.push({dist[v],v});
 }
 }
 }

 int ans=*max_element(dist.begin()+1, dist.end());
 if(ans==INT_MAX) return -1;
 return ans;
}

int main(){
 Solution sol;
 vector<vector<int>> times={{2,1,1},{2,3,1},{3,4,1}};
 int n=4, k=2;
 cout<<sol.networkDelayTime(times,n,k);
 return 0;
}

```

## Complexity Analysis

**Time Complexity:**  $O((V + E) \log V)$

Each edge relaxation and priority queue operation takes  $\log V$  time.

**Space Complexity:**  $O(V + E)$

Adjacency list stores all edges, and the distance array plus priority queue store up to  $V$  nodes.

# 34. G-40: Number of Ways to Arrive at Destination

You are given a city with  $n$  intersections numbered from 0 to  $n-1$ . The intersections are connected by **bi-directional roads**, and each road has a travel time.

You start at intersection 0 and want to reach intersection  $n-1$ .

Your task is to find **how many different ways** you can reach the destination **using the shortest possible time**.

Since the number of ways can be large, return the answer modulo  $10^9 + 7$ .

Example:

If the shortest time to reach the destination is 7 minutes, and there are 4 different paths that take exactly 7 minutes, then the answer is 4.

---

## Approach

### Algorithm

This problem is solved using **Dijkstra's Algorithm with path counting**.

1. Create an adjacency list to store roads between intersections.
2. Use a priority queue (min-heap) that stores {distance, node}.
3. Create a distance array  $\text{dist}[ ]$  initialized with infinity.
4. Create a  $\text{ways}[ ]$  array to store the number of shortest paths to each node.
5. Set  $\text{dist}[0] = 0$  and  $\text{ways}[0] = 1$  because there is exactly one way to start.
6. Push  $\{0, 0\}$  into the priority queue.
7. While the priority queue is not empty:
  - o Pop the node with the smallest distance.
  - o For every adjacent node:

- If a **shorter distance** is found:
    - Update the distance.
    - Copy the number of ways from the current node.
  - If the **same shortest distance** is found:
    - Add the number of ways from the current node.
8. At the end, return  $\text{ways}[n-1] \% (10^9 + 7)$ .

This works because Dijkstra ensures we process nodes in increasing order of shortest distance, and we count all paths that achieve that same minimum distance.

---

### Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int countPaths(int n, vector<vector<int>> &roads) {

 vector<pair<int,int>> adj[n];
 for(auto it: roads){
 adj[it[0]].push_back({it[1], it[2]});
 adj[it[1]].push_back({it[0], it[2]});
 }

 priority_queue<pair<long long,int>,
 vector<pair<long long,int>>,
 greater<pair<long long,int>>> pq;

 vector<long long> dist(n, LLONG_MAX);
 vector<int> ways(n, 0);

 dist[0]=0;
 ways[0]=1;
```

```

pq.push({0,0});

int mod=1e9+7;

while(!pq.empty()){
 auto it=pq.top();
 pq.pop();
 long long dis=it.first;
 int node=it.second;

 for(auto nbr: adj[node]){
 int adjNode=nbr.first;
 int edW=nbr.second;

 if(dis+edW<dist[adjNode]){
 dist[adjNode]=dis+edW;
 ways[adjNode]=ways[node];
 pq.push({dist[adjNode],adjNode});
 }
 else if(dis+edW==dist[adjNode]){
 ways[adjNode]=(ways[adjNode]+ways[node])%mod;
 }
 }
 return ways[n-1]%mod;
}

int main(){
 int n=7;
 vector<vector<int>>
roads={{0,6,7},{0,1,2},{1,2,3},{1,3,3},{6,3,3},
{3,5,1},{6,5,1},{2,5,1},{0,4,5},{4,6,2}};

 Solution obj;
 cout<<obj.countPaths(n,roads);
 return 0;
}

```

---

## Complexity Analysis

**Time Complexity:**  $O(E \log V)$

Dijkstra's algorithm processes each edge with priority queue operations.

**Space Complexity:**  $O(V + E)$

Used for adjacency list, distance array, ways array, and priority queue.

# 35. G-39: Minimum Multiplications to Reach End

You are given a starting number `start`, a target number `end`, and an array `arr` of numbers.

At each step, you can multiply the current number with **any element of the array**, and then take modulo 100000.

Your task is to find the **minimum number of multiplications** needed to reach end starting from `start`.

If it is not possible to reach end, return -1.

Example:

`arr = {2,5,7}`, `start = 3`, `end = 30`

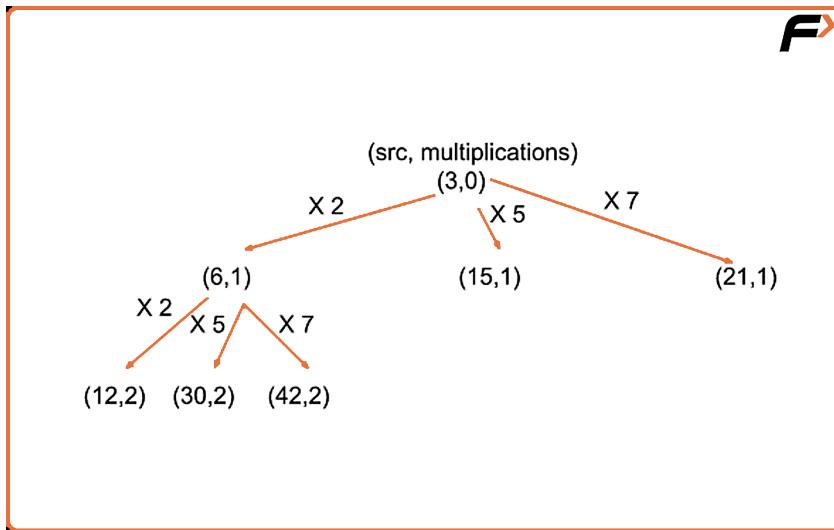
`3 → 6 → 30`

Minimum steps = 2

---

## Approach

### Algorithm



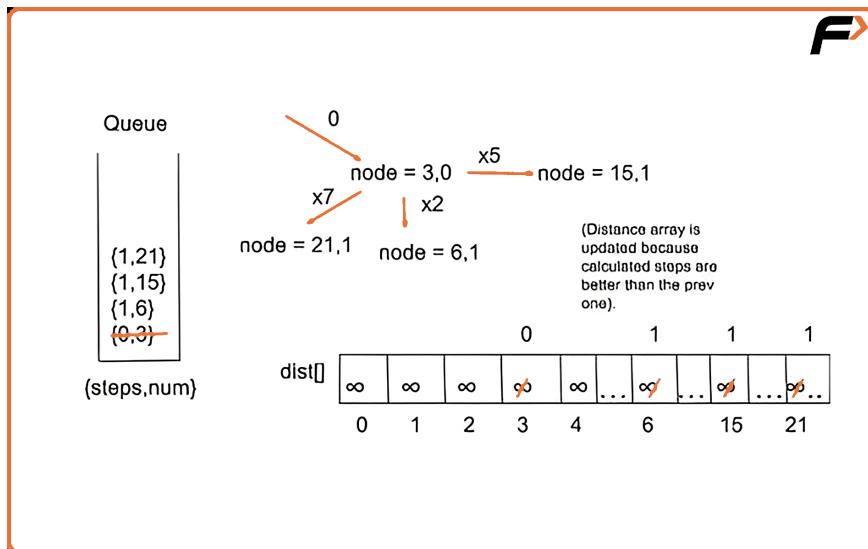
This problem can be treated as a **shortest path problem**.

- Each number (0 to 99999) is considered as a node.
- From a number  $x$ , you can go to  $(x * \text{arr}[i]) \% 100000$  in one step.
- Since every operation has equal cost (1 step), **BFS** is used.

Steps:

1. Create a queue that stores {number, steps}.
2. Create a distance array dist of size 100000, initialized with a very large value.
3. Set  $\text{dist}[\text{start}] = 0$  and push {start, 0} into the queue.
4. While the queue is not empty:
  - Pop the front element.
  - For each value in arr:
    - Compute  $\text{newNum} = (\text{currentNum} * \text{arr}[i]) \% 100000$ .
    - If  $\text{steps} + 1 < \text{dist}[\text{newNum}]$ , update it and push into the queue.
    - If  $\text{newNum} == \text{end}$ , return  $\text{steps} + 1$ .

5. If BFS finishes and end is not reached, return -1.



This guarantees minimum steps because BFS explores level by level.

---

### Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int minimumMultiplications(vector<int> &arr, int start, int end) {

 queue<pair<int,int>> q;
 q.push({start,0});

 vector<int> dist(100000,1e9);
 dist[start]=0;

 int mod=100000;

 while(!q.empty()){
 int node=q.front().first;
 int steps=q.front().second;

```

```

q.pop();

for(auto it: arr){
 int num=(node*it)%mod;

 if(steps+1<dist[num]){
 dist[num]=steps+1;

 if(num==end)
 return steps+1;

 q.push({num, steps+1});
 }
}

return -1;
};

int main(){
 vector<int> arr={2, 5, 7};
 int start=3, end=30;

 Solution obj;
 cout<<obj.minimumMultiplications(arr, start, end);
 return 0;
}

```

---

## Complexity Analysis

**Time Complexity:**  $O(100000 * N)$

Each number from 0 to 99999 can be visited, and for each we try all N multipliers.

**Space Complexity:**  $O(100000)$

Distance array stores the minimum steps for each possible number, and the queue can hold at most these many states.

# 36. Bellman Ford Algorithm: G-41

You are given a **directed weighted graph** with  $V$  vertices and a list of edges. Each edge is of the form  $[u, v, w]$ , meaning there is a directed edge from  $u$  to  $v$  with weight  $w$ .

You are also given a source vertex  $S$ .

Your task is to find the **shortest distance from the source  $S$  to all other vertices**.

If the graph contains a **negative weight cycle**, return an array containing only  $-1$ .

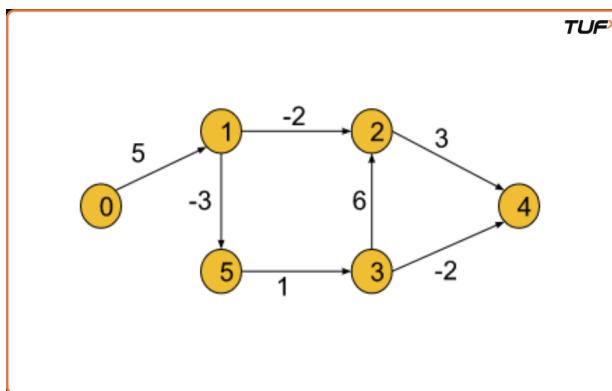
Example:

$V = 6$

Edges =

$[3,2,6], [5,3,1], [0,1,5], [1,5,-3], [1,2,-2], [3,4,-2], [2,4,3]$

$S = 0$



Output:

0 5 3 3 1 2

This means the shortest distances from node 0 to all other nodes are as shown.

---

## Approach

### Algorithm

Bellman-Ford algorithm is used to find shortest paths from a single source even when **negative edge weights** are present.

1. Create a distance array  $dist$  of size  $V$ .

2. Initialize all distances as infinity except the source S, which is set to 0.
3. Repeat the following process  $V-1$  times:
  - For every edge  $(u, v, wt)$ :
    - If  $dist[u]$  is not infinity and  $dist[u] + wt < dist[v]$ , update  $dist[v]$ .
4. After  $V-1$  relaxations, perform one more iteration over all edges:
  - If any distance can still be reduced, a **negative cycle exists**.
  - In that case, return  $\{-1\}$ .
5. If no negative cycle is found, return the distance array.

### Why $V-1$ times?

The shortest path between two vertices can have at most  $V-1$  edges. More relaxations are only needed to detect negative cycles.

---

### Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 vector<int> bellman_ford(int V, vector<vector<int>>& edges, int S)
 {

 vector<int> dist(V, 1e8);
 dist[S] = 0;

 // Relax all edges $V-1$ times
 for(int i = 0; i < V - 1; i++){
 for(auto it : edges){
 int u = it[0];
 int v = it[1];
 if(dist[u] != 1e8 && dist[u] + edges[u][v] < dist[v])
 dist[v] = dist[u] + edges[u][v];
 }
 }
 }
};
```

```

 int wt = it[2];

 if(dist[u] != 1e8 && dist[u] + wt < dist[v]){
 dist[v] = dist[u] + wt;
 }
 }
}

// Check for negative cycle
for(auto it : edges){
 int u = it[0];
 int v = it[1];
 int wt = it[2];

 if(dist[u] != 1e8 && dist[u] + wt < dist[v]){
 return {-1};
 }
}

return dist;
}
};

int main() {
 int V = 6;
 vector<vector<int>> edges = {
 {3,2,6},{5,3,1},{0,1,5},
 {1,5,-3},{1,2,-2},{3,4,-2},{2,4,3}
 };

 int S = 0;
 Solution obj;
 vector<int> ans = obj.bellman_ford(V, edges, S);

 for(int x : ans){
 cout << x << " ";
 }
 return 0;
}

```

}

---

## Complexity Analysis

**Time Complexity:**  $O(V * E)$

Each edge is relaxed  $V-1$  times, plus one extra pass for cycle detection.

**Space Complexity:**  $O(V)$

Only the distance array of size  $V$  is used.

# 37. Floyd Warshall Algorithm: G-42

You are given a directed weighted graph with  $V$  vertices numbered from 0 to  $V-1$ .

The graph is represented using an adjacency matrix `matrix`, where `matrix[i][j]` is the weight of the edge from vertex  $i$  to vertex  $j$ .

If `matrix[i][j] = -1`, it means there is no direct edge between  $i$  and  $j$ .

Your task is to find the **shortest distance between every pair of vertices**.

If a vertex is unreachable from another vertex, the value should remain  $-1$ .

Example:

Input matrix:

0 2 -1 -1

1 0 3 -1

-1 -1 0 1

3 5 4 0

After applying Floyd Warshall, the shortest distance from vertex 0 to 2 becomes 5, and from 1 to 3 becomes 4.

---

## Approach

### Algorithm

Floyd Warshall is an **all-pairs shortest path algorithm**.

1. Treat the given adjacency matrix as the distance matrix.
2. Use three nested loops:
  - Outer loop picks an intermediate node k.
  - Inner loops try to improve the path from i to j using node k.
3. For every pair (i, j):
  - If there is no path from i to k or k to j, skip.
  - If there is no direct edge from i to j, update it using  $i \rightarrow k \rightarrow j$ .
  - Otherwise, take the minimum of the current distance and the new distance via k.
4. After processing all intermediate nodes, the matrix contains the shortest distances.

This works because every shortest path can be built by gradually allowing more intermediate vertices.

---

### Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 void shortest_distance(vector<vector<int>> &matrix){
 int n = matrix.size();

 for(int k = 0; k < n; k++){
 for(int i = 0; i < n; i++){
 for(int j = 0; j < n; j++){
 if(matrix[i][k] == -1 || matrix[k][j] == -1)
 continue;

 if(matrix[i][j] == -1){
```

```

 matrix[i][j] = matrix[i][k] + matrix[k][j];
 }
 else{
 matrix[i][j] = min(matrix[i][j],
 matrix[i][k] +
matrix[k][j]);
 }
}
}
}
}
}

int main(){
vector<vector<int>> matrix = {
 {0, 2, -1, -1},
 {1, 0, 3, -1},
 {-1, -1, 0, 1},
 {3, 5, 4, 0}
};

Solution sol;
sol.shortest_distance(matrix);

for(int i = 0; i < matrix.size(); i++){
 for(int j = 0; j < matrix.size(); j++){
 cout << matrix[i][j] << " ";
 }
 cout << endl;
}
return 0;
}

```

---

## Complexity Analysis

**Time Complexity:**  $O(V^3)$

Three nested loops over all vertices.

**Space Complexity:**  $O(V^2)$

The adjacency matrix itself stores distances for all vertex pairs.

## 38. Find the City With the Smallest Number of Neighbours at a Threshold Distance: G-43

You are given  $n$  cities numbered from 0 to  $n-1$ . The cities are connected by **bidirectional weighted edges**.

Each edge  $[u, v, w]$  means the distance between city  $u$  and city  $v$  is  $w$ .

You are also given an integer `distanceThreshold`.

For every city, count how many cities (including itself) are reachable with **shortest distance  $\leq$  distanceThreshold**.

You must return the city that has the **smallest number of reachable cities**.

If multiple cities have the same minimum count, return the city with the **largest city number**.

Example:

If city 0 and city 3 both can reach only 2 cities within the threshold, the answer is city 3.

---

### Approach

#### Algorithm

This problem is solved using the **Floyd Warshall Algorithm** to compute all-pairs shortest paths.

1. Create a distance matrix `dist` of size  $n \times n$ .
2. Initialize all distances as `INT_MAX`.
3. For every edge  $[u, v, w]$ :

- Set  $\text{dist}[u][v] = w$
  - Set  $\text{dist}[v][u] = w$  (because the graph is bidirectional)
4. Set  $\text{dist}[i][i] = 0$  for all cities.
5. Apply Floyd Warshall:
- For every intermediate city k
  - For every pair (i, j)
  - Update  

$$\text{dist}[i][j] = \min(\text{dist}[i][j], \text{dist}[i][k] + \text{dist}[k][j])$$
6. For each city:
- Count how many cities have distance  $\leq \text{distanceThreshold}$ .
7. Keep track of the minimum count.
- If the count is smaller or equal, update the answer city.
  - Equal is allowed because we want the **largest city number** in case of a tie.
8. Return the city number.
- 

### Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int findCity(int n, int m, vector<vector<int>>& edges, int
distanceThreshold) {

 vector<vector<int>> dist(n, vector<int>(n, INT_MAX));

```

```

 for(auto it : edges){
 dist[it[0]][it[1]] = it[2];
 dist[it[1]][it[0]] = it[2];
 }

 for(int i = 0; i < n; i++)
 dist[i][i] = 0;

 for(int k = 0; k < n; k++){
 for(int i = 0; i < n; i++){
 for(int j = 0; j < n; j++){
 if(dist[i][k] == INT_MAX || dist[k][j] == INT_MAX)
 continue;
 dist[i][j] = min(dist[i][j], dist[i][k] +
dist[k][j]);
 }
 }
 }

 int cntCity = n;
 int cityNo = -1;

 for(int city = 0; city < n; city++){
 int cnt = 0;
 for(int adjCity = 0; adjCity < n; adjCity++){
 if(dist[city][adjCity] <= distanceThreshold)
 cnt++;
 }
 if(cnt <= cntCity){
 cntCity = cnt;
 cityNo = city;
 }
 }
 return cityNo;
 }
};

int main(){

```

```

int n = 4, m = 4;
vector<vector<int>> edges = {
 {0,1,3},{1,2,1},{1,3,4},{2,3,1}
};
int distanceThreshold = 4;

Solution obj;
cout << obj.findCity(n, m, edges, distanceThreshold);
return 0;
}

```

---

### **Complexity Analysis**

**Time Complexity:**  $O(V^3)$

Three nested loops are used in Floyd Warshall, where V is the number of cities.

**Space Complexity:**  $O(V^2)$

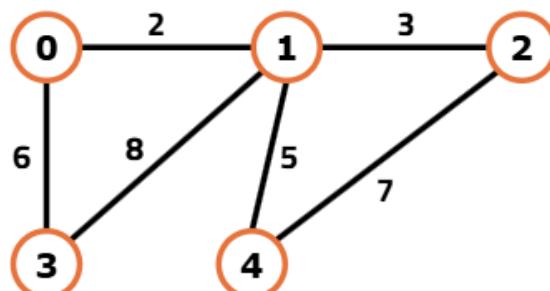
The distance matrix of size  $V \times V$  is used to store shortest paths.

## **39. Minimum Spanning Tree - Theory: G-44**

### **Spanning Tree:**

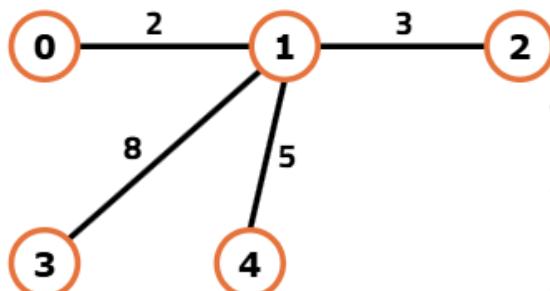
**A spanning tree is a tree in which we have N nodes(i.e. All the nodes present in the original graph) and N-1 edges and all nodes are reachable from each other.**

Let's understand this using an example. Assume we are given an undirected weighted graph with N nodes and M edges. Here in this example, we have taken N as 5 and M as 6.



Given  
Undirected  
Weighted graph

For the above graph, if we try to draw a spanning tree, the following illustration will be one:

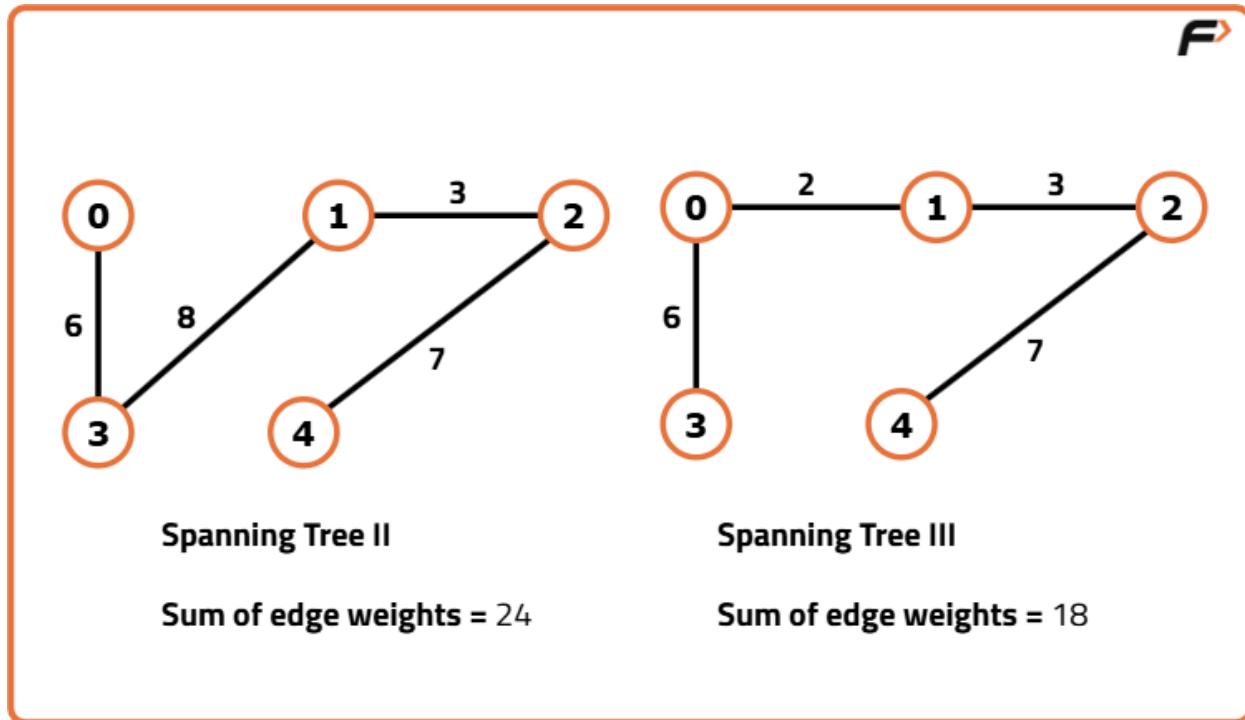


**There are 5 nodes and 4 edges, and all nodes are reachable from each other. So, this is definitely a spanning tree.**

**Spanning Tree I**

**Sum of edge weights = 18**

We can draw more spanning trees for the given graph. Two of them are like the following:



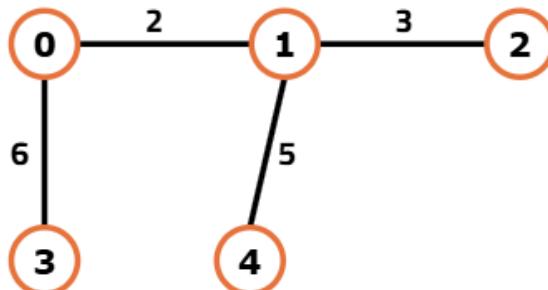
**Note:** Point to remember is that a graph may have more than one spanning trees.

All the above spanning trees contain some edge weights. For each of them, if we add the edge weights we can get the sum for that particular tree. Now, let's try to figure out the minimum spanning tree:

## Minimum Spanning Tree:

**Among all possible spanning trees of a graph, the minimum spanning tree is the one for which the sum of all the edge weights is the minimum.**

Let's understand the definition using the given graph drawn above. Until now, for the given graph we have drawn three spanning trees with the sum of edge weights 18, 24, and 18. If we can draw all possible spanning trees, we will find that the following spanning tree with the minimum sum of edge weights 16 is the **minimum spanning tree** for the given graph:

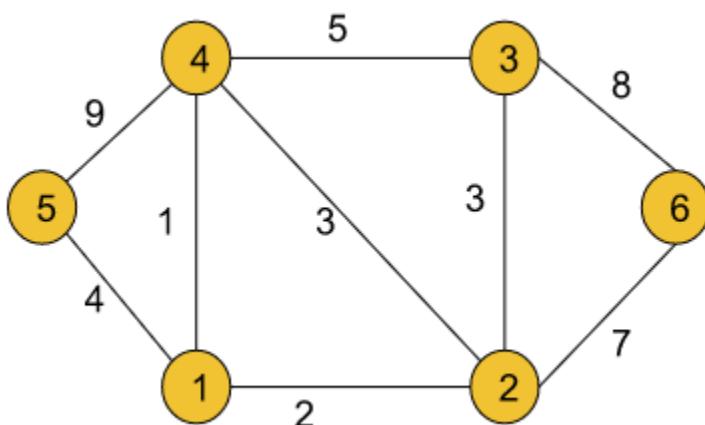


**Minimum spanning tree**  
**Sum of edge weights = 16**

**Note:** There may exist multiple minimum spanning trees for a graph like a graph may have multiple spanning trees.

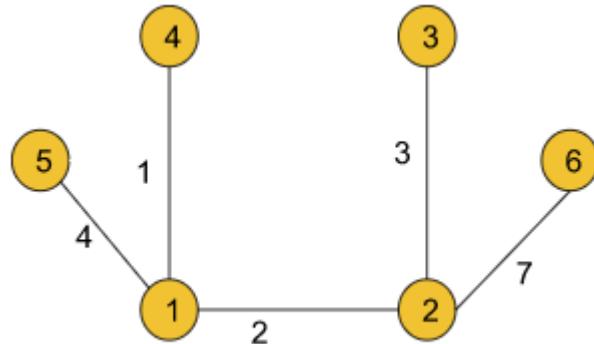
### Practice Problem

Now, in case you have understood the concepts well, you can try to figure out the minimum spanning tree for the following undirected weighted graph:



Answer: Sum of edge

weights = 17The MST of the given Graph will look like



this:

There are a couple of algorithms that help us to find the minimum spanning tree of a graph. One is **Prim's algorithm** and the other is **Kruskal's algorithm**. We will be discussing all of them in the upcoming articles.

## 40. Prim's Algorithm – Minimum Spanning Tree: G-45

You are given a **weighted, undirected, and connected graph** with V vertices and E edges.

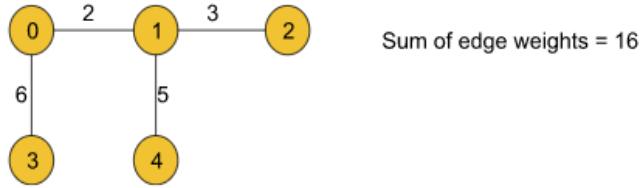
Your task is to find the **sum of the weights of the edges in the Minimum Spanning Tree (MST)**.

Sometimes, you may also be asked to store the MST edges, but here the focus is on the **sum of weights**.

A **Minimum Spanning Tree** connects all vertices using exactly  $V-1$  edges such that the total edge weight is minimum and no cycles are formed.

Example:

If the MST edges are  $(0, 1)$ ,  $(0, 3)$ ,  $(1, 2)$ ,  $(1, 4)$  and their weights add up to 16, then the answer is 16.



## Approach

### Algorithm

Prim's Algorithm builds the MST **greedily**, starting from any node and always choosing the minimum weight edge that connects a visited node to an unvisited node.

Steps:

1. Use a **visited array** to mark nodes that are already part of the MST.
2. Use a **priority queue (min-heap)** that stores pairs {edgeWeight, node}.
3. Start from node 0 by pushing {0, 0} into the priority queue.
4. Initialize sum = 0 to store the total weight of the MST.
5. While the priority queue is not empty:
  - o Pop the element with the minimum edge weight.
  - o If the node is already visited, skip it.
  - o Otherwise:
    - Mark the node as visited.
    - Add the edge weight to sum.
    - Push all adjacent unvisited nodes into the priority queue with their edge weights.
6. When all nodes are visited, sum contains the total weight of the MST.

This works because at every step we **greedily pick the smallest edge** that expands the current tree without forming a cycle.

---

### Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int spanningTree(int V, vector<vector<int>> adj[]) {

 priority_queue<pair<int,int>,
 vector<pair<int,int>>,
 greater<pair<int,int>>> pq;

 vector<int> vis(V,0);

 pq.push({0,0}); // {weight, node}
 int sum = 0;

 while(!pq.empty()){
 auto it = pq.top();
 pq.pop();
 int wt = it.first;
 int node = it.second;

 if(vis[node]) continue;

 vis[node] = 1;
 sum += wt;

 for(auto x : adj[node]){
 int adjNode = x[0];
 int edW = x[1];
 if(!vis[adjNode]){
 pq.push({edW, adjNode});
 }
 }
 }
 }
}
```

```

 }
 }
 return sum;
}
};

int main(){
 int V = 5;
 vector<vector<int>> edges = {
 {0,1,2},{0,2,1},{1,2,1},
 {2,3,2},{3,4,1},{4,2,2}
 };

 vector<vector<int>> adj[V];
 for(auto it : edges){
 adj[it[0]].push_back({it[1], it[2]});
 adj[it[1]].push_back({it[0], it[2]});
 }

 Solution obj;
 cout << obj.spanningTree(V, adj);
 return 0;
}

```

---

## Complexity Analysis

**Time Complexity:**  $O(E \log E)$

Each edge can be pushed into the priority queue, and push/pop operations take  $\log E$  time.

**Space Complexity:**  $O(E + V)$

$O(E)$  for the priority queue and adjacency list, and  $O(V)$  for the visited array.

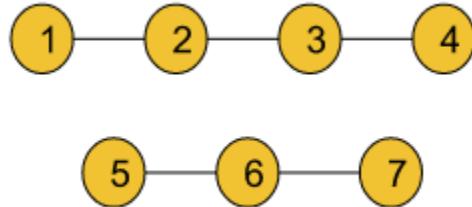
# 41. Disjoint Set | Union by Rank | Union by Size | Path Compression: G-46

A **Disjoint Set** (also called Union-Find) is a data structure used to manage a collection of **non-overlapping sets**.

It is mainly used to answer questions like:

“Do node u and node v belong to the same connected component?”

This is very useful in **graphs that change over time (dynamic graphs)**, where edges are added step by step and we need fast connectivity checks.



---

## Why Disjoint Set is needed

Suppose we are given an undirected graph and asked whether node 1 and node 5 belong to the same component.

- Using DFS or BFS:  $O(N + E)$  time for every query.
- Using Disjoint Set: **almost constant time** per query.

That is why Disjoint Set is preferred for **multiple connectivity queries**.

---

## Dynamic Graph Concept

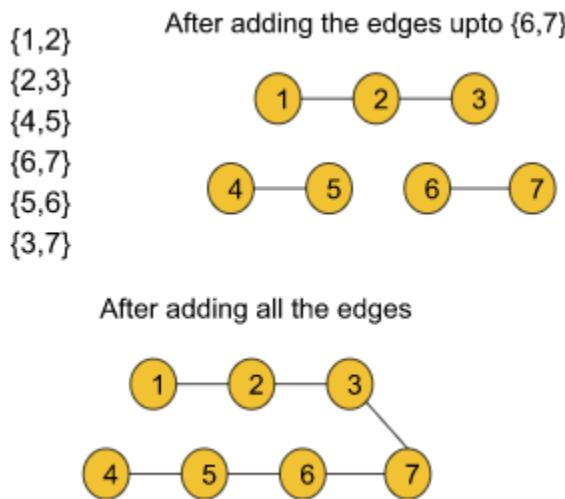
A dynamic graph is a graph where edges are added over time.

Example edge sequence:

$\{(1,2), (2,3), (4,5), (6,7), (5,6), (3,7)\}$

- After adding first 4 edges → node 1 and node 4 are in different components.
- After adding all edges → node 1 and node 4 become part of the same component.

Disjoint Set can answer such queries efficiently **after every update**.



---

## Operations in Disjoint Set

1. **findUPar(node)**  
Finds the **ultimate parent (root)** of a node.
2. **Union(u, v)**  
Merges the sets containing u and v.

Optimizations:

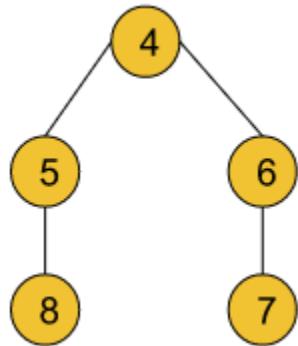
- Union by Rank
- Union by Size
- Path Compression

---

## Important Terminology

### Ultimate Parent

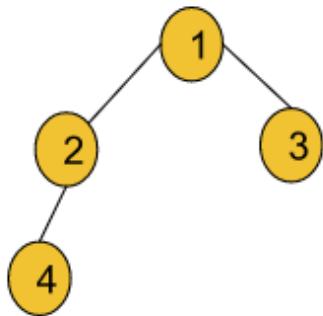
The topmost parent (root) of a component.



In this graph, the parent of 8 is 5 but the ultimate parent of 8 is 4

### Rank

An approximate measure of the height of the tree.



Here the rank of node 1 is 2 as the distance between node 1 and the furthest leaf node 4 is 2.

### Size

Number of nodes in a connected component.

---

## Union by Rank

In union by rank:

- Each node has a rank value.
- While merging two components, attach the component with **smaller rank** under the one with **larger rank**.
- If ranks are equal, attach any one and increase the rank by 1.

This helps in keeping the tree shallow.

---

## Why Ultimate Parent Matters

Two nodes may have different immediate parents but still belong to the same component.

Example:

- Node 5 → parent 6
- Node 7 → parent 6

Immediate parents differ, but **ultimate parent is same**, so they are connected.

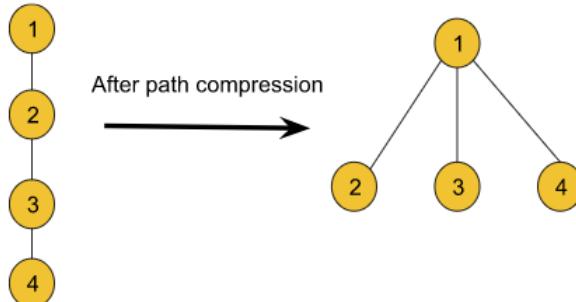
Hence, comparisons are always done using **ultimate parents**.

---

## Path Compression

While finding the ultimate parent:

- We directly connect every node in the path to the ultimate parent.
- This flattens the structure.

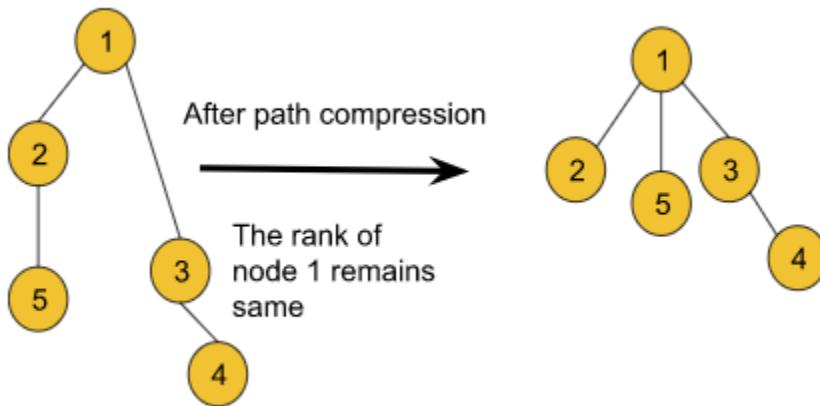


Before path compression:

- Finding parent may take  $O(\log N)$  time.

After path compression:

- Finding parent takes **almost constant time**.



Important:

- **Rank should not be updated during path compression** because it may get distorted.

Given

{1,2}

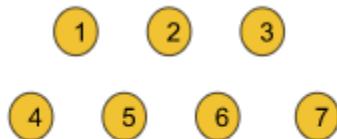
{2,3}

{4,5}

{6,7}

{5,6}

Initial configuration:



1 2 3 4 5 6 7

rank array

|   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|

parent array

|   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|

## Union by Rank Code (with Path Compression)

```
#include <bits/stdc++.h>
using namespace std;

class DisjointSet {
 vector<int> rank, parent;
public:
 DisjointSet(int n) {
 rank.resize(n+1, 0);
 parent.resize(n+1);
 for(int i=0;i<=n;i++)
 parent[i]=i;
 }

 int findUPar(int node){
 if(node==parent[node])
 return node;
 return parent[node]=findUPar(parent[node]);
 }

 void unionByRank(int u,int v){
 int pu=findUPar(u);
 int pv=findUPar(v);
 if(pu==pv) return;

 if(rank[pu]<rank[pv]){
 parent[pu]=pv;
 }
 else if(rank[pv]<rank[pu]){
 parent[pv]=pu;
 }
 else{
 parent[pv]=pu;
 rank[pu]++;
 }
 }
};
```

```

int main(){
 DisjointSet ds(7);
 ds.unionByRank(1, 2);
 ds.unionByRank(2, 3);
 ds.unionByRank(4, 5);
 ds.unionByRank(6, 7);
 ds.unionByRank(5, 6);

 if(ds.findUPar(3)==ds.findUPar(7))
 cout<<"Same\n";
 else
 cout<<"Not same\n";

 ds.unionByRank(3, 7);

 if(ds.findUPar(3)==ds.findUPar(7))
 cout<<"Same\n";
 else
 cout<<"Not same\n";
}

```

---

## Union by Size

Union by size is similar to union by rank, but instead of rank:

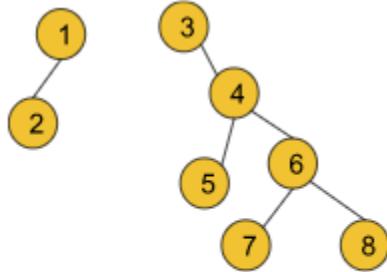
- We store the **size of each component**.
- Attach the smaller component under the larger one.
- Update the size accordingly.

This method is more intuitive because **size does not get distorted** after path compression.

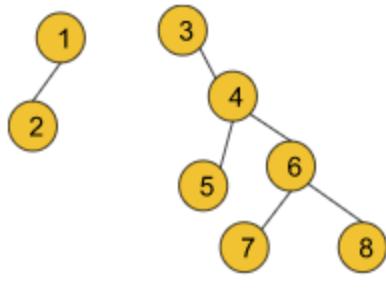
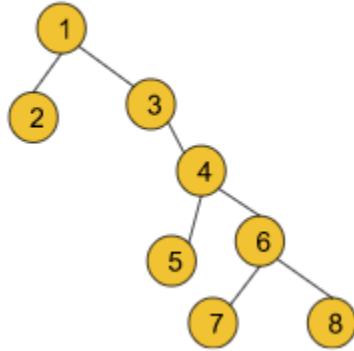
---

## Union by Size Algorithm

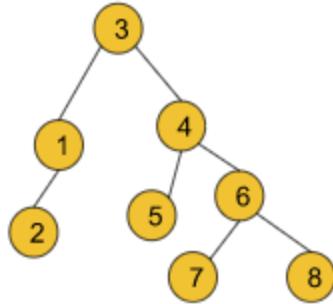
1. Find ultimate parents of  $u$  and  $v$ .
2. Compare  $\text{size}[pu]$  and  $\text{size}[pv]$ .
3. Attach smaller size component to larger size component.
4. Update the size.



Case 1: Connecting larger to smaller



Case 2: Connecting smaller to larger



Given

{1,2}

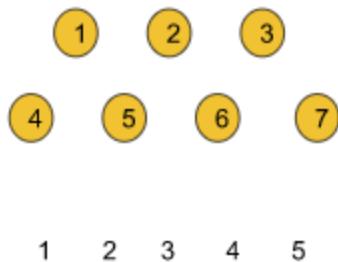
{2,3}

{4,5}

{6,7}

{5,6}

Initial configuration:



size array

|   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 |
|---|---|---|---|---|---|---|

parent array

|   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|

Given

Final configuration:

Union(1,2)

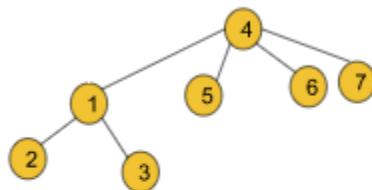
Union(2,3)

Union(4,5)

Union(6,7)

Union(5,6)

Union(3,7)



size array

|   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|
| 3 | 1 | 1 | 7 | 1 | 2 | 1 |
|---|---|---|---|---|---|---|

parent array

|   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 4 | 4 | 4 | 6 |
|---|---|---|---|---|---|---|

## Union by Size Code

```
#include <bits/stdc++.h>
using namespace std;

class DisjointSet {
 vector<int> parent, size;
public:
 DisjointSet(int n){
 parent.resize(n+1);
 size.resize(n+1, 1);
 for(int i=0;i<=n;i++)
 parent[i]=i;
```

```

}

int findUPar(int node){
 if(node==parent[node])
 return node;
 return parent[node]=findUPar(parent[node]);
}

void unionBySize(int u,int v){
 int pu=findUPar(u);
 int pv=findUPar(v);
 if(pu==pv) return;

 if(size[pu]<size[pv]){
 parent[pu]=pv;
 size[pv]+=size[pu];
 }
 else{
 parent[pv]=pu;
 size[pu]+=size[pv];
 }
}
};

int main(){
 DisjointSet ds(7);
 ds.unionBySize(1,2);
 ds.unionBySize(2,3);
 ds.unionBySize(4,5);
 ds.unionBySize(6,7);
 ds.unionBySize(5,6);

 if(ds.findUPar(3)==ds.findUPar(7))
 cout<<"Same\n";
 else
 cout<<"Not same\n";

 ds.unionBySize(3,7);
}

```

```

if(ds.findUPar(3)==ds.findUPar(7))
 cout<<"Same\n";
else
 cout<<"Not same\n";
}

```

---

## Time Complexity

- **findUPar + union operations:**  $O(\alpha(N))$
- $\alpha(N)$  (inverse Ackermann function) is very small
- Practically treated as **constant time**

# 42. Kruskal's Algorithm – Minimum Spanning Tree: G-47

You are given a **weighted, undirected, and connected graph** with V vertices and E edges. Your task is to find the **sum of weights of the edges in the Minimum Spanning Tree (MST)**.

A Minimum Spanning Tree connects all vertices using exactly  $V-1$  edges, has no cycles, and the total edge weight is minimum.

Example:

For edges [0-1(1), 1-2(2), 2-3(3), 0-3(4)], the MST uses edges with weights 1, 2, 3, so the answer is 6.

---

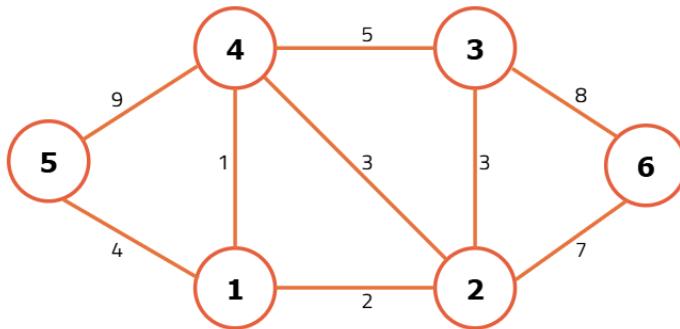
## Approach

## Algorithm

Kruskal's Algorithm builds the MST by **always choosing the smallest available edge** that does not form a cycle.

1. Store all edges in the form (weight, u, v).
2. Sort all edges in **ascending order of weight**.
3. Create a **Disjoint Set (Union-Find)** to track connected components.
4. Iterate through the sorted edges:
  - o If u and v belong to different components:
    - Add the edge weight to the MST sum.
    - Union the components of u and v.
  - o Otherwise, skip the edge (it forms a cycle).
5. After processing all edges, the accumulated sum is the MST weight.

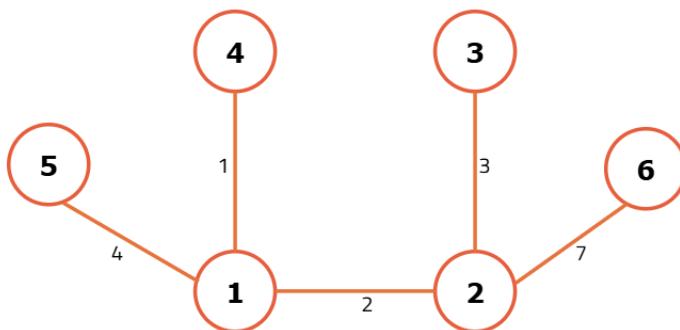
Cycle detection is efficiently handled using the Disjoint Set data structure.



Sorted edges according to weights:  
(wt u v)

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

The minimum spanning tree with  
sum of edge weights = 17



## Code

```
#include <bits/stdc++.h>
using namespace std;

class DisjointSet {
 vector<int> parent, size;
public:
 DisjointSet(int n) {
 parent.resize(n+1);
 size.resize(n+1, 1);
 for(int i=0;i<=n;i++) parent[i]=i;
 }

 int findUPar(int node) {
 if(node==parent[node]) return node;
```

```

 return parent[node]=findUPar(parent[node]);
 }

void unionBySize(int u,int v) {
 int pu=findUPar(u);
 int pv=findUPar(v);
 if(pu==pv) return;
 if(size[pu]<size[pv]) {
 parent[pu]=pv;
 size[pv]+=size[pu];
 } else {
 parent[pv]=pu;
 size[pu]+=size[pv];
 }
}
};

class Solution {
public:
 int spanningTree(int V, vector<vector<int>> adj[]) {
 vector<pair<int,pair<int,int>>> edges;

 for(int i=0;i<V;i++){
 for(auto it: adj[i]){
 edges.push_back({it[1],{i,it[0]}});
 }
 }

 sort(edges.begin(),edges.end());

 DisjointSet ds(V);
 int sum=0;

 for(auto it: edges){
 int wt=it.first;
 int u=it.second.first;
 int v=it.second.second;

```

```

 if(ds.findUPar(u)!=ds.findUPar(v)){
 sum+=wt;
 ds.unionBySize(u,v);
 }
 }
 return sum;
}

int main(){
 int V=4;
 vector<vector<int>> edges={{0,1,1},{1,2,2},{2,3,3},{0,3,4}};
 vector<vector<int>> adj[4];

 for(auto it: edges){
 adj[it[0]].push_back({it[1],it[2]});
 adj[it[1]].push_back({it[0],it[2]});
 }

 Solution obj;
 cout<<obj.spanningTree(V,adj);
 return 0;
}

```

---

## Complexity Analysis

### Time Complexity:

$O(E \log E)$

Sorting the edges dominates. Disjoint set operations are almost constant time.

### Space Complexity:

$O(E + V)$

Edges array uses  $O(E)$  space, and Disjoint Set uses  $O(V)$  space.

# 43. Number of Operations to Make Network Connected – DSU: G-49

You are given a graph with  $n$  vertices and  $m$  edges.

In one operation, you can **remove any existing edge** and **add it between any two vertices**.

Your task is to find the **minimum number of operations** required to make the graph **connected**.

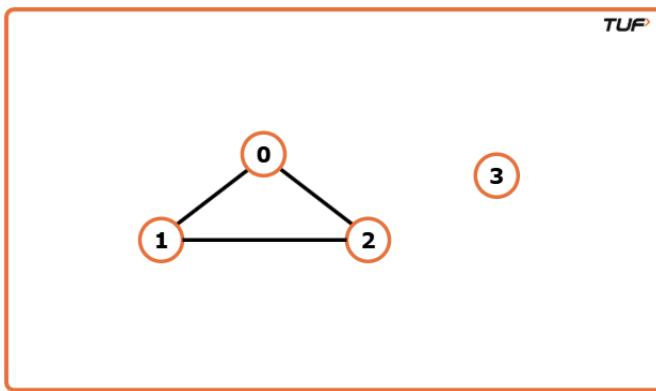
If it is not possible to make the graph connected, return -1.

A graph is connected if there is a path between every pair of vertices.

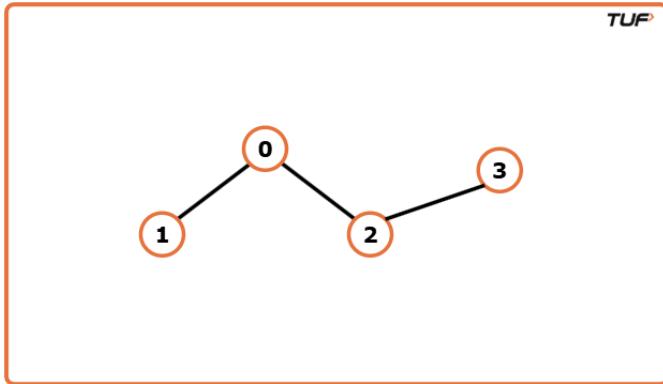
Example:

If the graph has multiple disconnected components, we need to connect those components using available extra edges.

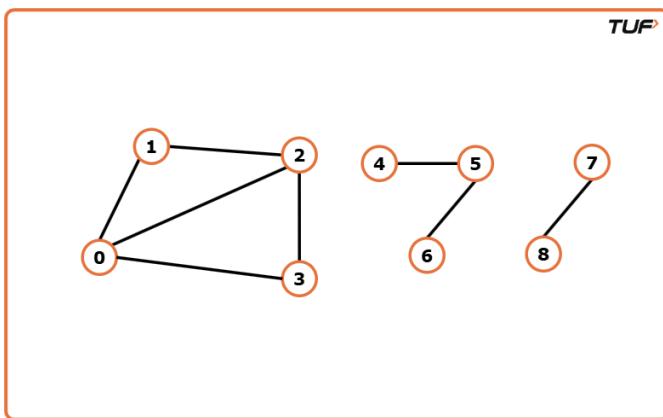
Input:



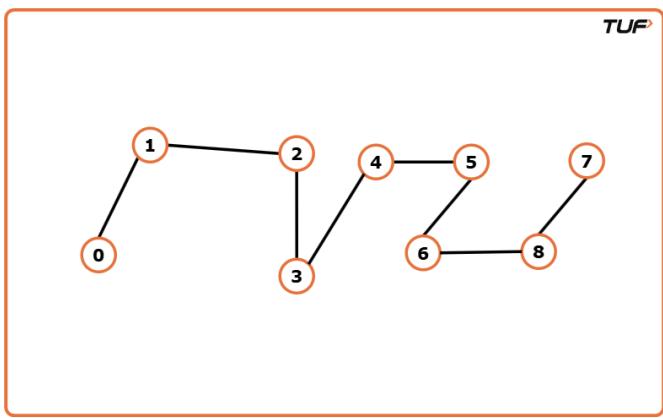
Output: 1



Input



Output: 1



We need a minimum of 2 operations to make the two components connected. We can remove the edge (0,2) and add the edge between node 3 and node 4 and we can remove the edge (0,3) and add it between nodes 6 and 8 like the following:

---

## Approach

### Algorithm

To make a graph connected:

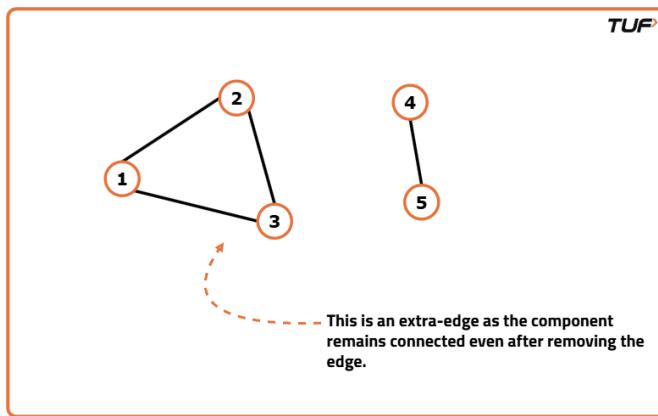
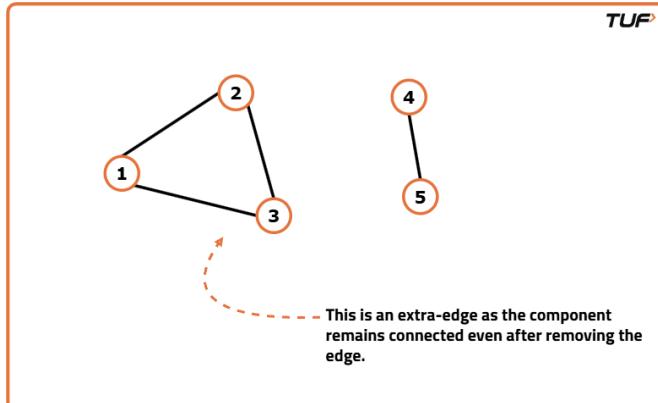
- If a graph has components connected components, then we need at least components - 1 edges to connect them.
- We are not allowed to add new edges freely; we can only **reuse extra (redundant) edges**.

Steps:

1. Initialize a **Disjoint Set Union (DSU)** to track connected components.
2. Traverse all edges:
  - If an edge connects two nodes already in the same component, it is an **extra edge**.
  - Otherwise, union the two components.
3. After processing all edges, count the number of connected components.
4. To connect all components, we need (components - 1) edges.
5. If the number of extra edges is **at least** (components - 1), return (components - 1).
6. Otherwise, return -1.

Important check:

- If total edges < n - 1, it is **impossible** to connect the graph.



## Code

```
#include <bits/stdc++.h>
using namespace std;

class DSU {
public:
 vector<int> parent, rank;

 DSU(int n) {
 parent.resize(n);
 rank.resize(n, 0);
 for(int i = 0; i < n; i++)
 parent[i] = i;
 }

 int find(int x) {
```

```

 if(parent[x] == x)
 return x;
 return parent[x] = find(parent[x]);
 }

void unite(int x, int y) {
 int px = find(x);
 int py = find(y);

 if(px == py) return;

 if(rank[px] < rank[py])
 parent[px] = py;
 else if(rank[px] > rank[py])
 parent[py] = px;
 else {
 parent[py] = px;
 rank[px]++;
 }
}

class Solution {
public:
 int makeConnected(int n, vector<vector<int>>& connections) {

 if(connections.size() < n - 1)
 return -1;

 DSU dsu(n);

 for(auto &edge : connections){
 dsu.unite(edge[0], edge[1]);
 }

 unordered_set<int> components;
 for(int i = 0; i < n; i++){
 components.insert(dsu.find(i));
 }
 }
}

```

```

 }

 return components.size() - 1;
}
};

int main() {
 int n = 6;
 vector<vector<int>> connections = {{0,1},{0,2},{0,3},{1,4}};
 Solution obj;
 cout << obj.makeConnected(n, connections);
 return 0;
}

```

---

### Complexity Analysis

**Time Complexity:**  $O(N + M \times \alpha(N))$

Each DSU operation is almost constant time, and we process all nodes and edges once.

**Space Complexity:**  $O(N)$

Used for DSU parent and rank arrays.

## 44. Most Stones Removed with Same Row or Column – DSU: G-53

You are given  $n$  stones placed on a 2D plane. Each stone is placed at a unique coordinate  $(x, y)$ .

A stone can be removed **only if there exists another stone in the same row or the same column that is not removed**.

Your task is to find the **maximum number of stones** that can be removed following this rule.

Key idea:

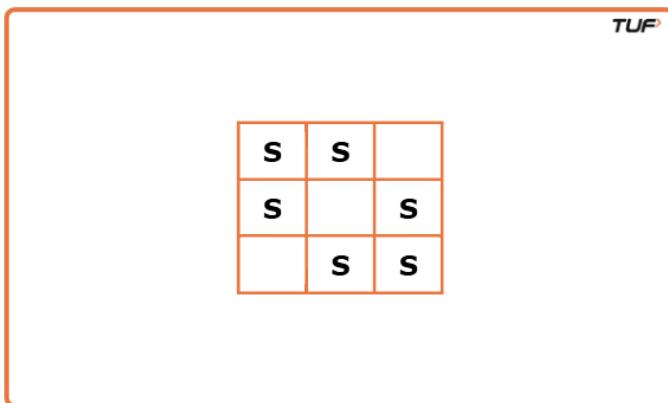
From a group of stones that are connected by same rows or columns, **we must leave at least one stone**, and all others can be removed.

---

## Explanation with example

Example:

```
stones = [[0,0],[0,1],[1,0],[1,2],[2,1],[2,2]]
```



All stones are connected through rows or columns, forming **one single group**.

From one group of size 6, we can remove  $6 - 1 = 5$  stones.

Another example may form **multiple groups**.

From each group, we can remove (group size - 1) stones.

So overall:

**Maximum stones removed = total stones – number of connected groups**

---

## Approach

### Algorithm

We use **Disjoint Set Union (DSU)** to group stones that lie in the same row or same column.

1. Treat each **row index** and each **column index** as separate DSU nodes.
2. To avoid collision between row and column indices, add an offset to column indices.

3. For every stone ( $x, y$ ):
    - o Union  $x$  (row) with  $y + \text{offset}$  (column).
  4. After all unions:
    - o Count how many **unique connected components** exist (based on ultimate parents of rows).
  5. Since one stone must remain in each component:
    - o  $\text{answer} = \text{total stones} - \text{number of components}$
- 

## Code

```
#include <bits/stdc++.h>
using namespace std;

class DSU {
public:
 unordered_map<int,int> parent;

 int find(int x){
 if(parent.find(x)==parent.end())
 parent[x]=x;
 if(parent[x]==x) return x;
 return parent[x]=find(parent[x]);
 }

 void unite(int x,int y){
 int px=find(x);
 int py=find(y);
 if(px!=py)
 parent[px]=py;
 }
};

class Solution {
public:
```

```

int removeStones(vector<vector<int>>& stones) {
 DSU dsu;
 int offset = 10001;

 for(auto &stone: stones){
 dsu.unite(stone[0], stone[1] + offset);
 }

 unordered_set<int> components;
 for(auto &stone: stones){
 components.insert(dsu.find(stone[0]));
 }

 return stones.size() - components.size();
}

};

int main(){
 vector<vector<int>> stones = {
 {0,0},{0,1},{1,0},{1,2},{2,1},{2,2}
 };

 Solution obj;
 cout << obj.removeStones(stones);
 return 0;
}

```

---

## Complexity Analysis

### Time Complexity:

$O(N \times \alpha(N))$

Each union and find operation is almost constant time.

### Space Complexity:

$O(N)$

DSU stores parents for rows and columns involved.

# 45. Accounts Merge – DSU: G-50

You are given multiple user accounts.

Each account contains a **user name** followed by one or more **email addresses**.

Two accounts belong to the **same person** if they share **at least one common email**.

Your task is to **merge such accounts** and return the result where:

- Each merged account starts with the user's name
  - Followed by **sorted, unique emails**
  - Output order does not matter
- 

## Example Explanation

Example 1:

Two accounts of "John" share the email `johnsmith@mail.com`, so they belong to the same person and must be merged.

Accounts of "Mary" and another "John" do not share emails with\_ud, so they stay separate.

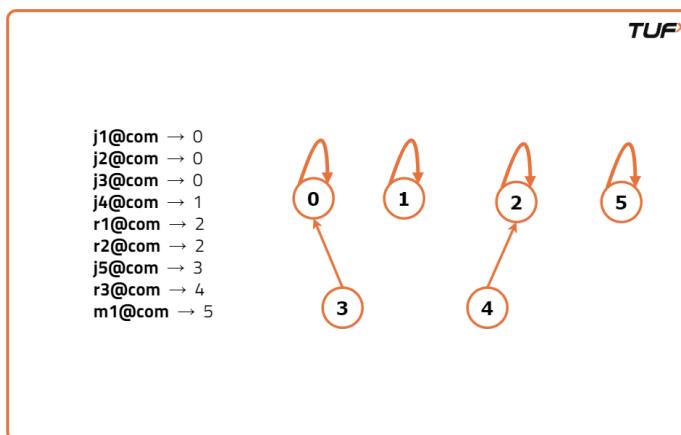
---

## Algorithm

We treat **each account index as a node** and use **Disjoint Set Union (DSU)** to merge accounts that share emails.

1. Create a Disjoint Set for n accounts.
2. Create a map `email → account index`.
3. Traverse each account:
  - For every email:
    - If the email is not in the map, store `email → current index`

- If it already exists, union the current account index with the stored index
4. After all unions:
    - Traverse all emails again
    - Find the **ultimate parent account** for each email
    - Group emails under their ultimate parent  5. For each group:
    - Sort emails
    - Add the account name at the front  6. Store all merged accounts in the result.



## Code

```

#include <bits/stdc++.h>
using namespace std;

class DisjointSet {
 vector<int> parent, size;
public:
 DisjointSet(int n) {
 parent.resize(n);
 size.resize(n, 1);
 }
}

```

```

 for(int i=0;i<n;i++) parent[i]=i;
 }

 int findUPar(int node) {
 if(node==parent[node]) return node;
 return parent[node]=findUPar(parent[node]);
 }

 void unionBySize(int u,int v) {
 int pu=findUPar(u);
 int pv=findUPar(v);
 if(pu==pv) return;
 if(size[pu]<size[pv]) {
 parent[pu]=pv;
 size[pv]+=size[pu];
 } else {
 parent[pv]=pu;
 size[pu]+=size[pv];
 }
 }
};

class Solution {
public:
 vector<vector<string>> accountsMerge(vector<vector<string>>&
details) {
 int n=details.size();
 DisjointSet ds(n);
 unordered_map<string,int> mailToNode;

 for(int i=0;i<n;i++){
 for(int j=1;j<details[i].size();j++){
 string mail=details[i][j];
 if(mailToNode.find(mail)==mailToNode.end())
 mailToNode[mail]=i;
 else
 ds.unionBySize(i,mailToNode[mail]);
 }
 }
 }
};

```

```

 }

 vector<vector<string>> mergedMail(n);
 for(auto it: mailToNode){
 string mail=it.first;
 int node=ds.findUPar(it.second);
 mergedMail[node].push_back(mail);
 }

 vector<vector<string>> ans;
 for(int i=0;i<n;i++){
 if(mergedMail[i].empty()) continue;
 sort(mergedMail[i].begin(), mergedMail[i].end());
 vector<string> temp;
 temp.push_back(details[i][0]);
 for(auto &mail: mergedMail[i])
 temp.push_back(mail);
 ans.push_back(temp);
 }
 return ans;
}

};

int main(){
 vector<vector<string>> accounts={
 {"John", "j1@com", "j2@com", "j3@com"},
 {"John", "j4@com"},
 {"Raj", "r1@com", "r2@com"},
 {"John", "j1@com", "j5@com"},
 {"Raj", "r2@com", "r3@com"},
 {"Mary", "m1@com"}
 };
}

Solution obj;
auto res=obj.accountsMerge(accounts);

for(auto &acc: res){
 for(auto &s: acc) cout<<s<<" ";
}

```

```
 cout<<endl;
}
return 0;
}
```

---

## Complexity Analysis

### Time Complexity:

$$O(N + E) + O(E \log E)$$

- Traversing emails and unions:  $O(N + E)$
- Sorting emails for each merged account

### Space Complexity:

$$O(N + E)$$

- DSU arrays
- Email map
- Merged email storage

# 46. Number of Islands – II (Online Queries) – DSU: G-51

You are given a grid of size  $n \times m$  which is initially filled with water (0).

You are also given  $k$  operations. Each operation turns a water cell into land (1).

After **each operation**, you need to return the **current number of islands**.

An **island** is a group of land cells (1) connected **vertically or horizontally**.

Important points:

- Initially, there are **no islands**.
  - The same cell can appear multiple times in the operations.
  - You must return an answer **after every operation**.
- 

## Example Explanation

Example 1:

$n = 4, m = 5$

Operations:  $\{(1, 1), (0, 1), (3, 3), (3, 4)\}$

After each operation, islands count becomes:

1 1 2 2

This happens because new land may either:

- Create a new island
  - Or merge with existing nearby islands
- 

## Approach

This is an **online connectivity problem**, best solved using **Disjoint Set Union (DSU)**.

1. Treat each cell ( $\text{row}, \text{col}$ ) as a node using mapping:  
 $\text{node} = \text{row} * m + \text{col}$
2. Maintain a visited matrix to mark land cells.
3. Maintain a DSU for  $n * m$  possible cells.
4. Keep a counter  $\text{cnt}$  for number of islands.
5. For each operation ( $\text{row}, \text{col}$ ):

- If the cell is already land, push current cnt and continue.
  - Otherwise:
    - Mark it as land.
    - Increase cnt by 1.
    - Check its 4 neighbors (up, right, down, left).
    - If any neighbor is land and belongs to a different DSU component:
      - Union them.
      - Decrease cnt by 1 (two islands merged).
  - Push cnt into the answer array.
6. Return the answer array.

This works because:

- Each new land starts as a new island.
- DSU efficiently merges connected lands.
- Island count updates dynamically.

## Code

```
#include <bits/stdc++.h>
using namespace std;

class DisjointSet {
 vector<int> parent, size;
public:
 DisjointSet(int n) {
 parent.resize(n);
 size.resize(n, 1);
```

```

 for(int i=0;i<n;i++) parent[i]=i;
 }

int findUPar(int node){
 if(node==parent[node]) return node;
 return parent[node]=findUPar(parent[node]);
}

void unionBySize(int u,int v){
 int pu=findUPar(u);
 int pv=findUPar(v);
 if(pu==pv) return;
 if(size[pu]<size[pv]){
 parent[pu]=pv;
 size[pv]+=size[pu];
 } else {
 parent[pv]=pu;
 size[pu]+=size[pv];
 }
}
};

class Solution {
 bool isValid(int r,int c,int n,int m){
 return r>=0 && r<n && c>=0 && c<m;
 }
public:
 vector<int> num0fIslands(int n, int m, vector<vector<int>>&
operators) {

 DisjointSet ds(n*m);
 vector<vector<int>> vis(n,vector<int>(m,0));
 vector<int> ans;
 int cnt=0;

 int dr[]={-1,0,1,0};
 int dc[]={0,1,0,-1};

```

```

 for(auto &op: operators){
 int r=op[0], c=op[1];

 if(vis[r][c]){
 ans.push_back(cnt);
 continue;
 }

 vis[r][c]=1;
 cnt++;

 int node=r*m+c;

 for(int i=0;i<4;i++){
 int nr=r+dr[i];
 int nc=c+dc[i];
 if(isValid(nr,nc,n,m) && vis[nr][nc]){
 int adjNode=nr*m+nc;
 if(ds.findUPar(node)!=ds.findUPar(adjNode)){
 cnt--;
 ds.unionBySize(node,adjNode);
 }
 }
 ans.push_back(cnt);
 }
 return ans;
 }
 };

int main(){
 int n=4,m=5;
 vector<vector<int>> ops={{0,0},{0,0},{1,1},{1,0},{0,1},
{0,3},{1,3},{0,4},{3,2},{2,2},{1,2},{0,2}};

 Solution obj;
 vector<int> res=obj.numOfIslands(n,m,ops);
 for(int x:res) cout<<x<<" ";
}

```

```
 return 0;
}
```

---

## Complexity Analysis

### Time Complexity:

$O(K \times \alpha(N \times M))$

Each operation involves constant DSU operations.

### Space Complexity:

$O(N \times M)$

For visited matrix and DSU arrays.

# 47. Making a Large Island – DSU: G-52

You are given an  $n \times n$  binary grid where 1 represents land and 0 represents water.

An island is a group of 1s connected **horizontally or vertically**.

You are allowed to change **at most one 0** into 1.

Your task is to return the **maximum possible size of an island** after this operation.

---

## Question Explanation

Initially, the grid may contain multiple islands or none at all.

When we change a 0 to 1, it may:

- Create a new island of size 1
- Connect multiple nearby islands into one large island

The goal is to choose the best 0 to flip so that the resulting island size is maximum.

If the grid already contains only 1s, then no change is needed and the answer is simply the total number of cells.

---

## Example Explanation

Example 1

```
grid = [[1, 0], [0, 1]]
```

There are two separate islands of size 1.

If we change any one 0 to 1, it connects both islands.

Resulting island size = 3.

Example 2

```
grid = [[1, 1], [1, 1]]
```

All cells are already connected.

Largest island size = 4.

---

## Approach

### Algorithm

We use **Disjoint Set Union (DSU)** to efficiently manage island connections and sizes.

1. Treat each cell ( $i, j$ ) as a node using:  
$$\text{node} = i * n + j$$
2. Create a DSU of size  $n * n$ .
3. Traverse the grid and union all adjacent land cells (1s) to form initial islands.
4. Now, for every 0 cell:
  - o Look at its 4 neighbors.

- Collect **unique island parents** of neighboring 1s using a set (to avoid double counting).
  - Sum the sizes of these unique islands.
  - Add 1 for the flipped cell.
  - Update the maximum island size.
5. Handle the edge case where the grid has no 0:
- The answer is the size of the largest existing island.
6. Return the maximum size found.
- 

## Code

```
#include <bits/stdc++.h>
using namespace std;

class DisjointSet {
public:
 vector<int> parent, size;

 DisjointSet(int n) {
 parent.resize(n);
 size.resize(n, 1);
 for(int i = 0; i < n; i++)
 parent[i] = i;
 }

 int findUPar(int node) {
 if(node == parent[node]) return node;
 return parent[node] = findUPar(parent[node]);
 }

 void unionBySize(int u, int v) {
 int pu = findUPar(u);

```

```

 int pv = findUPar(v);
 if(pu == pv) return;

 if(size[pu] < size[pv]) {
 parent[pu] = pv;
 size[pv] += size[pu];
 } else {
 parent[pv] = pu;
 size[pu] += size[pv];
 }
 }
};

class Solution {
 bool isValid(int r, int c, int n) {
 return r >= 0 && r < n && c >= 0 && c < n;
 }

public:
 int largestIsland(vector<vector<int>>& grid) {
 int n = grid.size();
 DisjointSet ds(n * n);

 int dr[4] = {-1, 0, 1, 0};
 int dc[4] = {0, 1, 0, -1};

 // Step 1: Union existing islands
 for(int i = 0; i < n; i++) {
 for(int j = 0; j < n; j++) {
 if(grid[i][j] == 0) continue;
 for(int d = 0; d < 4; d++) {
 int ni = i + dr[d];
 int nj = j + dc[d];
 if(isValid(ni, nj, n) && grid[ni][nj] == 1) {
 int node1 = i * n + j;
 int node2 = ni * n + nj;
 ds.unionBySize(node1, node2);
 }
 }
 }
 }
 }
};

```

```

 }
 }
}

int ans = 0;

// Step 2: Try converting each 0 to 1
for(int i = 0; i < n; i++) {
 for(int j = 0; j < n; j++) {
 if(grid[i][j] == 1) continue;

 set<int> components;
 for(int d = 0; d < 4; d++) {
 int ni = i + dr[d];
 int nj = j + dc[d];
 if(isValid(ni, nj, n) && grid[ni][nj] == 1) {
 int node = ni * n + nj;
 components.insert(ds.findUPar(node));
 }
 }
 int totalSize = 1;
 for(auto comp : components) {
 totalSize += ds.size[comp];
 }
 ans = max(ans, totalSize);
 }
}

// Edge case: all 1s
for(int i = 0; i < n * n; i++) {
 ans = max(ans, ds.size[ds.findUPar(i)]);
}

return ans;
}
};


```

```
int main() {
 vector<vector<int>> grid = {{1,0},{0,1}};
 Solution obj;
 cout << obj.largestIsland(grid);
 return 0;
}
```

---

## Complexity Analysis

### Time Complexity:

$O(n^2)$

We traverse the grid a constant number of times and DSU operations are nearly constant.

### Space Complexity:

$O(n^2)$

DSU stores parent and size for all  $n^2$  cells.

# 48. Swim in Rising Water

You are given an  $n \times n$  grid where  $\text{grid}[i][j]$  represents the elevation of a cell.

Rain starts at time  $t = 0$ , and at time  $t$ , water level is  $t$  everywhere.

You start from the top-left cell  $(0, 0)$  and want to reach the bottom-right cell  $(n-1, n-1)$ .

You can move **up, down, left, right** between adjacent cells **only if both cells have elevation  $\leq t$** .

Your task is to return the **minimum time  $t$**  at which it is possible to reach the destination.

---

### Example 1

$\text{grid} = [[0,2],[1,3]]$

- At  $t = 0$ , only cell  $(0, 0)$  is accessible.
- At  $t = 1$ , cell  $(1, 0)$  becomes accessible.
- At  $t = 2$ , cell  $(0, 1)$  becomes accessible.
- At  $t = 3$ , cell  $(1, 1)$  becomes accessible.

So the minimum time required is 3.

---

## Approach

### Algorithm

This problem can be seen as finding a path from start to end such that the **maximum elevation on the path is minimized**.

We simulate the rising water using a **min-heap (priority queue)**:

1. Start from cell  $(0, 0)$  and push it into a min-heap.
2. The heap always gives the cell with the **minimum required water level**.
3. Keep track of visited cells to avoid revisiting.
4. For each cell popped from the heap:
  - The current time needed is the maximum elevation seen so far on that path.
5. Push all valid, unvisited neighbors into the heap with:
  - `max(current_time, neighbor_elevation)`
6. As soon as we reach  $(n-1, n-1)$ , return the current time.

---

The first time we reach the destination is guaranteed to be the minimum possible time.

---

## Code

```

#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int swimInWater(vector<vector<int>>& grid) {
 int n = grid.size();

 // Min-heap storing {time, row, col}
 priority_queue<vector<int>, vector<vector<int>>,
 greater<vector<int>> pq;

 vector<vector<int>> vis(n, vector<int>(n, 0));

 // Start from (0,0)
 pq.push({grid[0][0], 0, 0});
 vis[0][0] = 1;

 int dr[4] = {0, 1, 0, -1};
 int dc[4] = {1, 0, -1, 0};

 while (!pq.empty()) {
 auto cur = pq.top();
 pq.pop();

 int time = cur[0];
 int r = cur[1];
 int c = cur[2];

 // Destination reached
 if (r == n - 1 && c == n - 1)
 return time;

 // Explore neighbors
 for (int i = 0; i < 4; i++) {
 int nr = r + dr[i];
 int nc = c + dc[i];

```

```

 if (nr >= 0 && nc >= 0 && nr < n && nc < n &&
!vis[nr][nc]) {
 vis[nr][nc] = 1;
 pq.push({max(time, grid[nr][nc]), nr, nc});
 }
 }
 return -1;
}
};

int main() {
 vector<vector<int>> grid = {{0,2},{1,3}};
 Solution obj;
 cout << obj.swimInWater(grid);
 return 0;
}

```

---

## Complexity Analysis

### Time Complexity:

$O(n^2 \log n^2)$

Each cell is pushed into the priority queue once, and heap operations take logarithmic time.

### Space Complexity:

$O(n^2)$

Used for the visited matrix and the priority queue.

# 49. Bridges in Graph – Using Tarjan's Algorithm (Time In & Low Time): G-55

You are given an undirected graph with n nodes (servers) numbered from 0 to n-1.

Each connection [a, b] represents a bidirectional link between server a and server b.

A **critical connection (bridge)** is an edge such that **removing it disconnects the graph**, meaning some servers can no longer reach others.

Your task is to return **all such critical connections**.

---

## Question Explanation

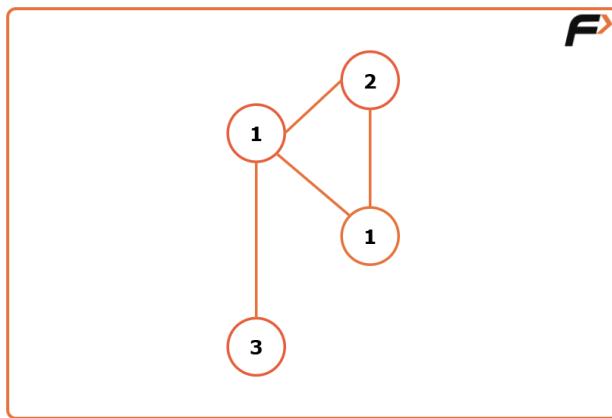
In a connected network, some edges are very important.

If removing an edge increases the number of connected components, that edge is called a **bridge**.

For example, in

```
connections = [[0,1],[1,2],[2,0],[1,3]]
```

- The edge [1, 3] is the only way to reach node 3
- Removing it disconnects node 3 from the rest  
So [1, 3] is a bridge.



## Approach

### Algorithm (Tarjan's Algorithm using DFS)

We use **DFS with time of insertion (tin) and lowest reachable time (low)**.

Definitions:

- `tin[node]`: the time when the node is first visited
- `low[node]`: the minimum `tin` reachable from that node (including back edges)

Steps:

1. Build the adjacency list from the given connections.
2. Maintain:
  - `vis[]` to mark visited nodes
  - `tin[]` for discovery time
  - `low[]` for lowest reachable time
  - a global timer
3. Start DFS from node 0 (if graph is disconnected, DFS should be started from every unvisited node).
4. For each edge (`node → adjNode`):
  - If `adjNode` is the parent, skip it.
  - If `adjNode` is unvisited:
    - DFS on it
    - Update `low[node] = min(low[node], low[adjNode])`
    - If `low[adjNode] > tin[node]`, then this edge is a **bridge**

- If adjNode is already visited (back edge):
    - Update  $\text{low}[\text{node}] = \min(\text{low}[\text{node}], \text{tin}[\text{adjNode}])$
5. Store all such edges that satisfy the bridge condition.
- 

## Example Explanation

For

```
connections = [[0,1],[1,2],[2,0],[1,3]]
```

- Nodes 0, 1, 2 form a cycle → no bridge inside the cycle
  - Node 3 is connected only via 1
  - $\text{low}[3] > \text{tin}[1] \rightarrow$  edge [1, 3] is a bridge
- 

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
private:
 int timer = 1;

 void dfs(int node, int parent, vector<int> &vis,
 vector<int> adj[], int tin[], int low[],
 vector<vector<int>> &bridges) {

 vis[node] = 1;
 tin[node] = low[node] = timer;
 timer++;

 for (auto it : adj[node]) {
 if (it == parent) continue;

 if (vis[it] == 0) {
 dfs(it, node, vis, adj, tin, low, bridges);
 low[node] = min(low[node], low[it]);
 if (low[it] > tin[node])
 bridges.push_back({node, it});
 } else if (vis[it] == 1) {
 low[node] = min(low[node], tin[it]);
 }
 }
 }
};
```

```

 if (!vis[it]) {
 dfs(it, node, vis, adj, tin, low, bridges);

 low[node] = min(low[node], low[it]);

 if (low[it] > tin[node]) {
 bridges.push_back({node, it});
 }
 } else {
 low[node] = min(low[node], tin[it]);
 }
 }

public:
vector<vector<int>> criticalConnections(int n,
 vector<vector<int>>& connections) {

 vector<int> adj[n];
 for (auto &it : connections) {
 adj[it[0]].push_back(it[1]);
 adj[it[1]].push_back(it[0]);
 }

 vector<int> vis(n, 0);
 int tin[n], low[n];
 vector<vector<int>> bridges;

 dfs(0, -1, vis, adj, tin, low, bridges);

 return bridges;
}

int main() {
 int n = 4;
 vector<vector<int>> connections = {

```

```

{0,1},{1,2},{2,0},{1,3}
};

Solution obj;
vector<vector<int>> ans = obj.criticalConnections(n, connections);

for (auto &e : ans) {
 cout << "[" << e[0] << "," << e[1] << "] ";
}
return 0;
}

```

---

## Complexity Analysis

### Time Complexity:

$O(V + E)$

Each vertex and edge is visited once during DFS.

### Space Complexity:

$O(V + E)$

Adjacency list plus arrays vis, tin, and low.

# 50. Articulation Point in Graph: G-56

You are given an undirected graph.

Your task is to find all **articulation points (cut vertices)**.

An articulation point is a vertex such that **removing it (along with its edges)** increases the number of connected components in the graph.

If no articulation point exists, return -1.

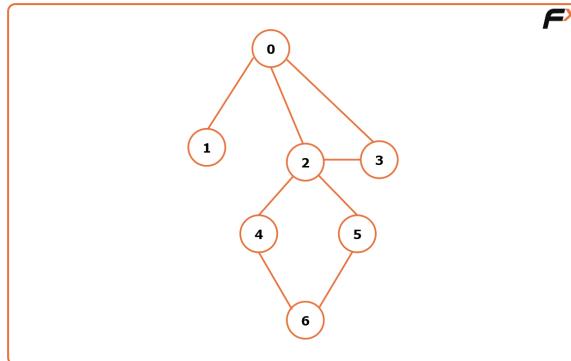
## Question Explanation

In a graph, some vertices are very important for connectivity.

If removing a vertex breaks the graph into two or more disconnected parts, that vertex is called an **articulation point**.

For example:

- If removing node 0 disconnects the graph, then 0 is an articulation point.



- Removing node 0
- Removing node 2

- If removing node 2 also disconnects the graph, then 2 is also an articulation point.

---

## Approach

### Algorithm

We use **Tarjan's Algorithm** with **DFS**, using two arrays:

- `tin[node]` → time when the node is first visited
- `low[node]` → lowest discovery time reachable from that node

Steps:

1. Build the adjacency list of the graph.
2. Initialize:
  - o timer to track discovery time
  - o vis[ ] to mark visited nodes
  - o tin[ ] and low[ ]
3. Run DFS for every unvisited node (graph may be disconnected).
4. During DFS:
  - o Set  $\text{tin}[\text{node}] = \text{low}[\text{node}] = \text{timer}++$
  - o Maintain a child count for the current node
5. For each adjacent node:
  - o If it is the parent, skip it
  - o If unvisited:
    - DFS on it
    - Update  $\text{low}[\text{node}] = \min(\text{low}[\text{node}], \text{low}[\text{child}])$
    - If  $\text{low}[\text{child}] \geq \text{tin}[\text{node}]$  and node is **not root**, mark node as articulation point
    - Increase child count
  - o If already visited:
    - Update  $\text{low}[\text{node}] = \min(\text{low}[\text{node}], \text{tin}[\text{adjNode}])$
6. Special case for root:

- If root has more than one child, it is an articulation point
7. Collect all marked nodes as articulation points.
  8. If none found, return -1.
- 

## Example Explanation

If a node has a child subtree that **cannot reach any ancestor** of the node using a back edge, then removing the node will disconnect that subtree. Hence, the node is an articulation point.

---

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
private:
 int timer = 1;

 void dfs(int node, int parent, vector<int> &vis,
 int tin[], int low[], vector<int> &mark,
 vector<int> adj[]) {

 vis[node] = 1;
 tin[node] = low[node] = timer++;
 int child = 0;

 for (auto it : adj[node]) {
 if (it == parent) continue;

 if (!vis[it]) {
 dfs(it, node, vis, tin, low, mark, adj);
 low[node] = min(low[node], low[it]);

 if (low[it] >= tin[node] && parent != -1) {

```

```

 mark[node] = 1;
 }
 child++;
}
else {
 low[node] = min(low[node], tin[it]);
}
}

if (parent == -1 && child > 1) {
 mark[node] = 1;
}
}

public:
vector<int> articulationPoints(int n, vector<int> adj[]) {

 vector<int> vis(n, 0), mark(n, 0);
 int tin[n], low[n];

 for (int i = 0; i < n; i++) {
 if (!vis[i]) {
 dfs(i, -1, vis, tin, low, mark, adj);
 }
 }

 vector<int> ans;
 for (int i = 0; i < n; i++) {
 if (mark[i]) ans.push_back(i);
 }

 if (ans.size() == 0) return {-1};
 return ans;
}
};

int main() {
 int n = 5;

```

```

vector<vector<int>> edges = {
 {0,1}, {1,4}, {2,4}, {2,3}, {3,4}
};

vector<int> adj[n];
for (auto e : edges) {
 adj[e[0]].push_back(e[1]);
 adj[e[1]].push_back(e[0]);
}

Solution sol;
vector<int> res = sol.articulationPoints(n, adj);
for (int x : res) cout << x << " ";
return 0;
}

```

---

## Complexity Analysis

### Time Complexity:

$O(V + E)$

DFS visits every vertex and edge once.

### Space Complexity:

$O(V)$

Used for vis, tin, low, and recursion stack.

# 51. Strongly Connected Components – Kosaraju's Algorithm (G-54)

---

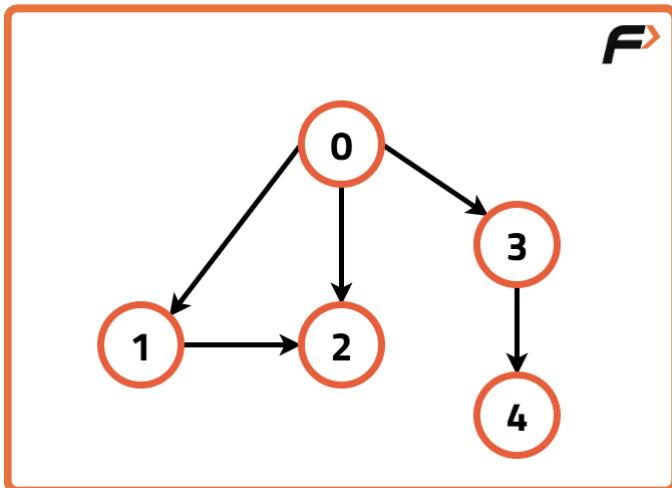
You are given a **directed graph** with  $V$  vertices (0 to  $V-1$ ) and  $E$  edges.

Your task is to **find the number of Strongly Connected Components (SCCs)**.

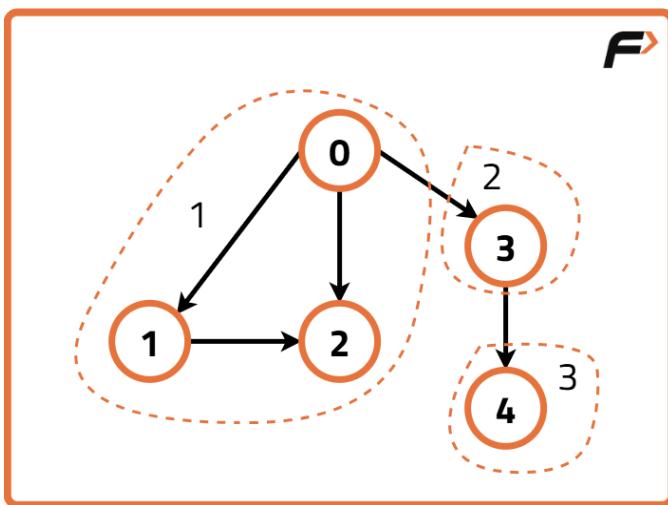
### Strongly Connected Component (SCC):

A group of vertices where **every vertex is reachable from every other vertex** in the same group.

Input:



Result: 3



---

### Key Idea (Intuition)

In a directed graph:

- Reachability is **direction dependent**

- Cycles form the backbone of SCCs

**Kosaraju's Algorithm** works by exploiting two observations:

1. Nodes that finish **later** in DFS are more “independent”
  2. Reversing edges preserves SCC structure but changes reachability order
- 

## Algorithm (Kosaraju's – 3 Steps)

### Step 1: DFS + Stack (Finishing Time Order)

- Perform DFS on the **original graph**
- Push nodes into a stack **after DFS completes** (based on finishing time)

### Step 2: Transpose the Graph

- Reverse all edges  
If there is an edge  $u \rightarrow v$ , make it  $v \rightarrow u$

### Step 3: DFS on Transposed Graph

- Pop nodes from the stack
- For each unvisited node, perform DFS in the **transposed graph**
- Each DFS traversal gives **one SCC**

The **number of DFS calls in step 3 = number of SCCs**

---

## Why This Works

- The first DFS orders nodes by how “deeply connected” they are
- The transpose graph ensures we only visit nodes inside the same SCC

- Processing nodes in decreasing finish time avoids cross-SCC traversal
- 

## Implementation (C++)

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
private:
 // DFS to fill stack by finishing time
 void dfs(int node, vector<int> &vis,
 vector<int> adj[], stack<int> &st) {

 vis[node] = 1;
 for (auto it : adj[node]) {
 if (!vis[it]) {
 dfs(it, vis, adj, st);
 }
 }
 // push after finishing DFS
 st.push(node);
 }

 // DFS on transposed graph
 void dfsTranspose(int node, vector<int> &vis,
 vector<int> adjT[]) {

 vis[node] = 1;
 for (auto it : adjT[node]) {
 if (!vis[it]) {
 dfsTranspose(it, vis, adjT);
 }
 }
 }

public:
 int kosaraju(int V, vector<int> adj[]) {
```

```

vector<int> vis(V, 0);
stack<int> st;

// Step 1: DFS on original graph
for (int i = 0; i < V; i++) {
 if (!vis[i]) {
 dfs(i, vis, adj, st);
 }
}

// Step 2: Transpose graph
vector<int> adjT[V];
for (int i = 0; i < V; i++) {
 vis[i] = 0;
 for (auto it : adj[i]) {
 adjT[it].push_back(i);
 }
}

// Step 3: DFS using stack order
int scc = 0;
while (!st.empty()) {
 int node = st.top();
 st.pop();

 if (!vis[node]) {
 scc++;
 dfsTranspose(node, vis, adjT);
 }
}
return scc;
};

int main() {
 int V = 5;
 vector<vector<int>> edges = {

```

```

 {1,0}, {0,2}, {2,1}, {0,3}, {3,4}
};

vector<int> adj[V];
for (auto e : edges) {
 adj[e[0]].push_back(e[1]);
}

Solution sol;
cout << "Number of SCCs: "
 << sol.kosaraju(V, adj) << endl;
return 0;
}

```

---

## Complexity Analysis

### Metric Complexity

**Time**  $O(V + E)$

**Space**  $O(V + E)$

- Each edge and vertex is processed at most twice
- Stack + transpose graph storage included

**DP**

# Dynamic Programming Introduction

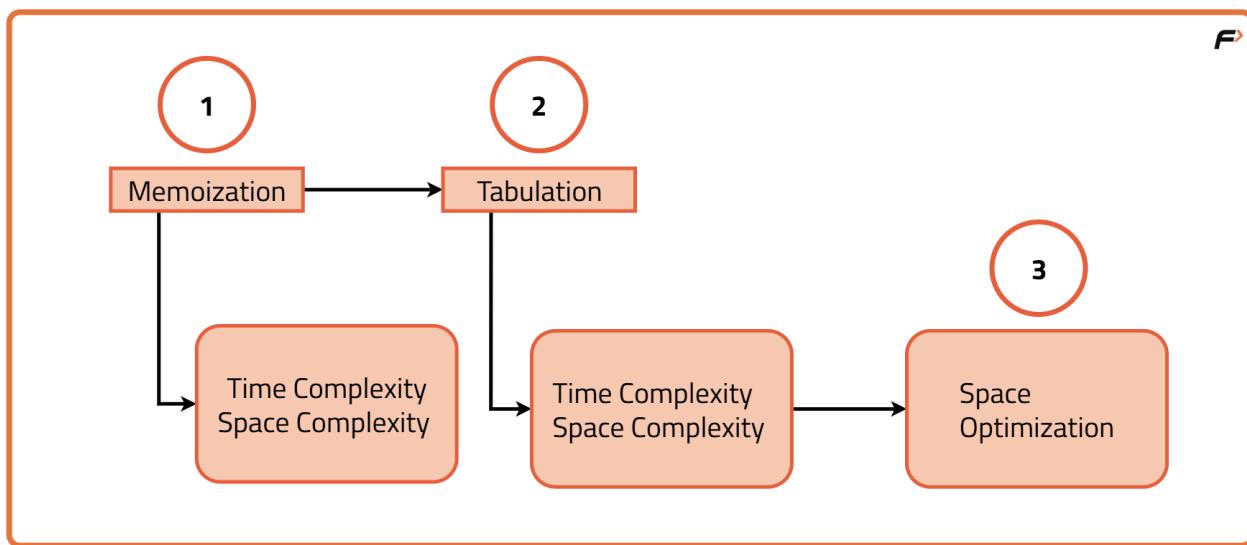
## Problem Statement: Introduction To Dynamic Programming

Dynamic Programming can be described as storing answers to various sub-problems to be used later whenever required to solve the main problem.

The two common dynamic programming approaches are:

- Memoization: Known as the “top-down” dynamic programming, usually the problem is solved in the direction of the main problem to the base cases.
- Tabulation: Known as the “bottom-up ” dynamic programming, usually the problem is solved in the direction of solving the base cases to the main problem

**Note:** The base case does not always mean smaller input. It depends upon the implementation of the algorithm.



We will be using the example of Fibonacci numbers here. The following series is called the Fibonacci series:

0,1,1,2,3,5,8,13,21,...

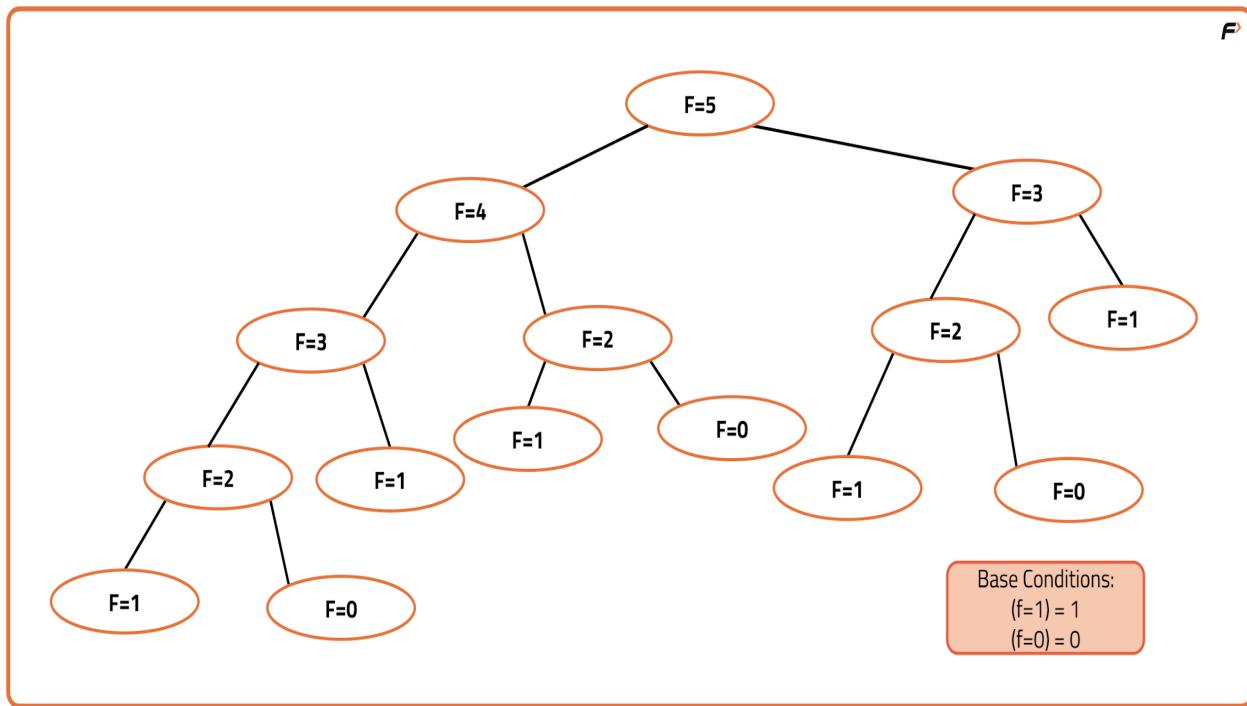
We need to find the  $n$ th Fibonacci number, where  $n$  is based on a 0-based index. Every  $i$ th number of the series is equal to the sum of  $(i-1)$ th and  $(i-2)$ th number where the first and second number is given as 0 and 1 respectively.

## Part - 1: Memoizaton

As every number is equal to the sum of the previous two terms, the recurrence relation can be written as:

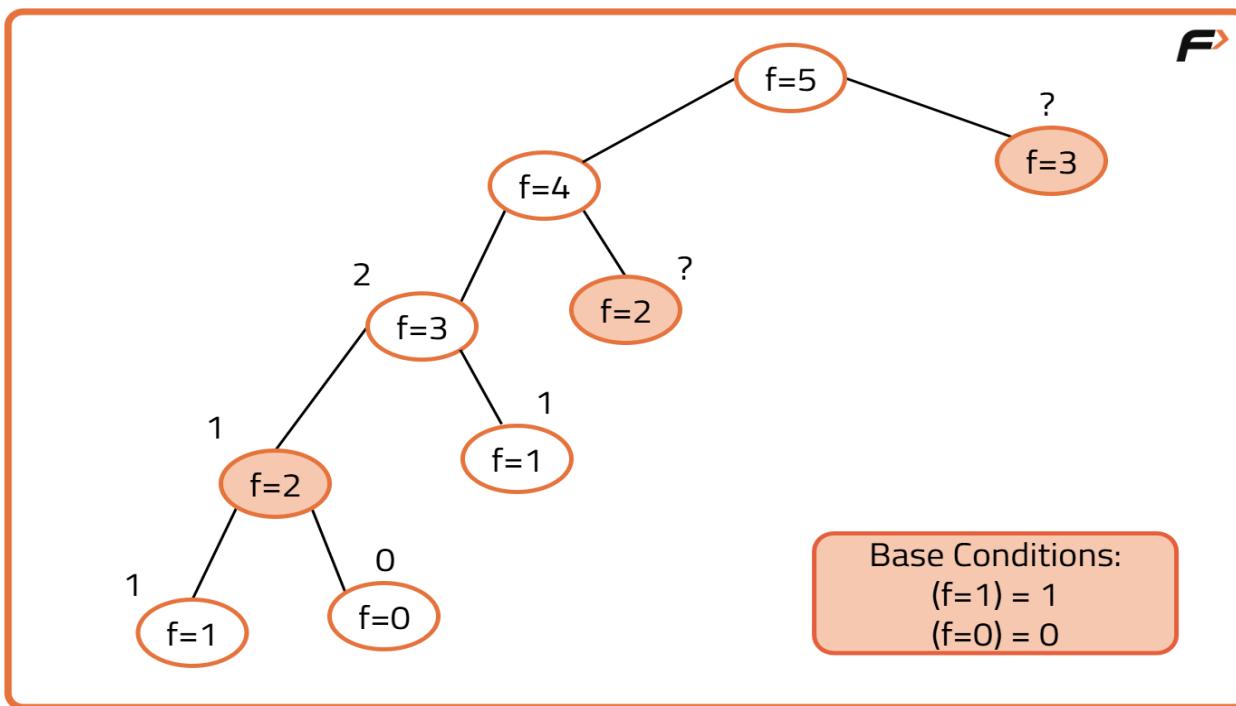
$$f(n) = f(n-1) + f(n-2)$$

If we draw the recursive tree for  $n=5$ , it will be:



If there are two recursive calls inside a function, the program will run the first call, finish its execution and then run the second call. Due to this reason, each and every call in the recursive tree will be executed. This gives the recursive code its exponential time complexity. If we can store the values of sub-problems in the first time, then we can simply find its value in constant time whenever we need

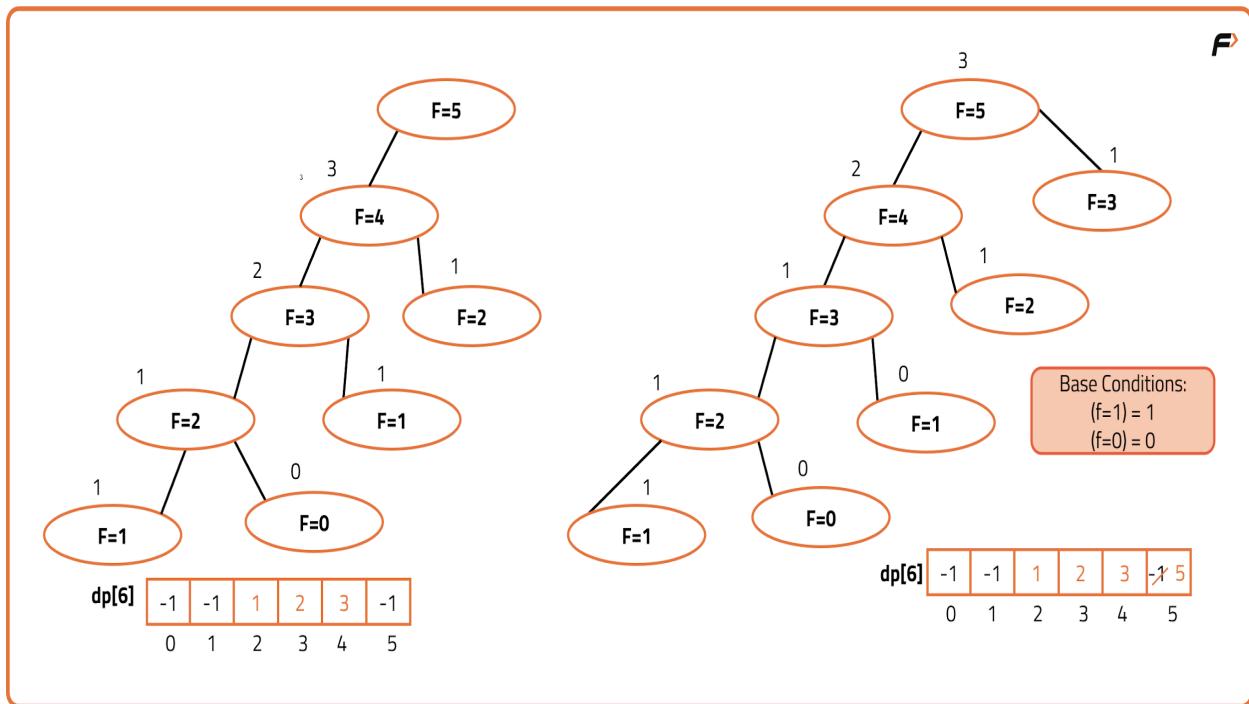
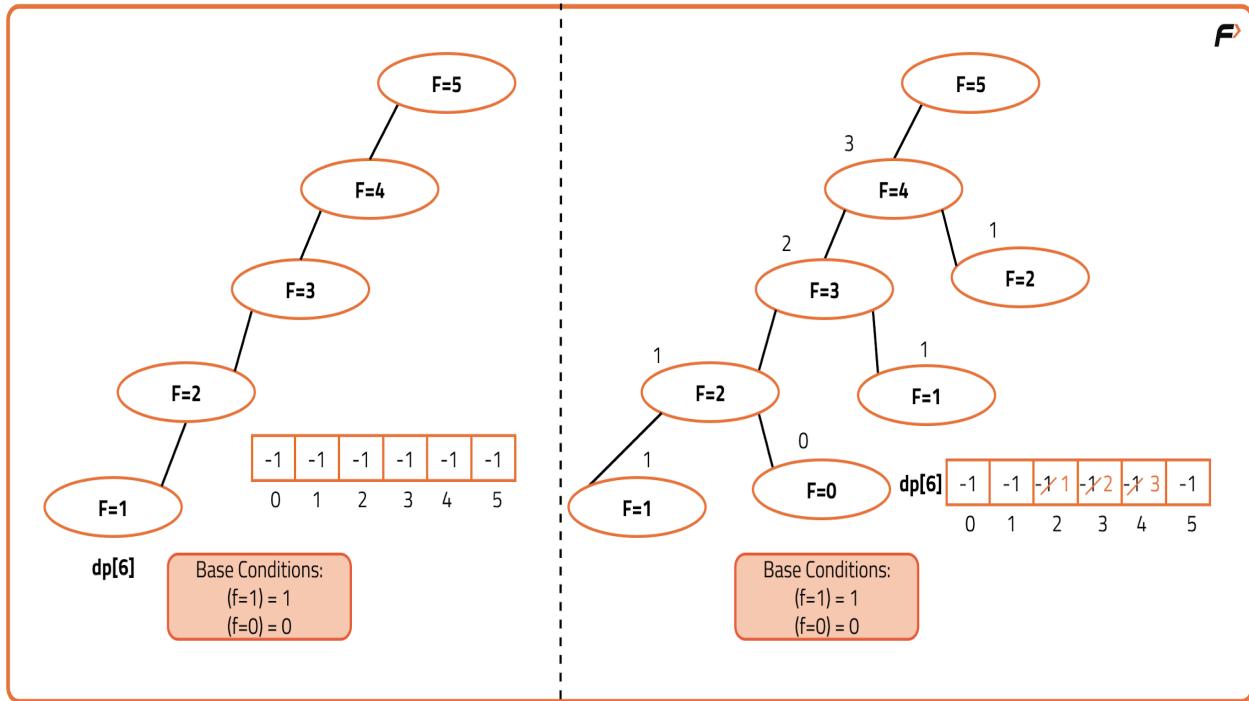
it. This technique is called **Memoization**. The cases which are solved again and again in recursion are called overlapping sub-problems.



### **Steps to memoize a recursive solution:**

Any recursive solution to a problem can be memoized using these three steps:

- Create a  $dp[n+1]$  array initialized to -1.
  - Whenever we want to find the answer of a particular value (say  $n$ ), we first check whether the answer is already calculated using the  $dp$  array. If yes, simply return the value from the  $dp$  array.
  - If not, then we are finding the answer for the given value for the first time, we will use the recursive relation as usual but before returning from the function, we will store the solution in our  $dp$  array.



```
#include <bits/stdc++.h>
using namespace std;
```

```
class Solution {
```

```

public:
 // Function to calculate Fibonacci using memoization
 int fib(int n, vector<int>& dp) {
 // If base case return n
 if (n <= 1) return n;

 // If already computed, return stored value
 if (dp[n] != -1) return dp[n];

 // Otherwise compute and store
 dp[n] = fib(n - 1, dp) + fib(n - 2, dp);
 return dp[n];
 }

};

int main() {
 int n = 10;
 vector<int> dp(n + 1, -1);
 Solution sol;
 cout << sol.fib(n, dp);
 return 0;
}

```

## Approach 2: Tabulation

Tabulation is a bottom-up approach where we **start from base cases** and build the solution iteratively. This avoids recursion and extra stack space.

### Algorithm

1. Create a dp array of size  $n+1$ .
2. Initialize  $dp[0] = 0$  and  $dp[1] = 1$ .
3. Use a loop from 2 to  $n$ .
4. Set  $dp[i] = dp[i-1] + dp[i-2]$ .
5. Return  $dp[n]$ .

### Code

```

#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int fib(int n) {
 if (n <= 1) return n;
 vector<int> dp(n + 1, 0);
 dp[0] = 0;
 dp[1] = 1;
 for (int i = 2; i <= n; i++) {
 dp[i] = dp[i - 1] + dp[i - 2];
 }
 return dp[n];
 }
};

int main() {
 int n = 10;
 Solution sol;
 cout << sol.fib(n);
 return 0;
}

```

## Complexity Analysis

- **Time Complexity:**  $O(n)$   
Reason: Single loop from 2 to n.
- **Space Complexity:**  $O(n)$   
Reason: dp array of size  $n+1$ .

## Approach 3: Space Optimization

From the relation:

$$dp[i] = dp[i-1] + dp[i-2]$$

We observe that we only need the **previous two values**, not the entire dp array.

## Algorithm

1. Handle base cases  $n = 0$  and  $n = 1$ .
2. Use two variables:
  - o `prev2` for  $\text{fib}(i-2)$
  - o `prev` for  $\text{fib}(i-1)$
3. For each iteration, compute  $\text{curr} = \text{prev} + \text{prev2}$ .
4. Update  $\text{prev2} = \text{prev}$  and  $\text{prev} = \text{curr}$ .
5. After the loop, return `prev`.

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int fib(int n) {
 if (n == 0) return 0;
 if (n == 1) return 1;

 int prev2 = 0;
 int prev = 1;
 int curr;

 for (int i = 2; i <= n; i++) {
 curr = prev + prev2;
 prev2 = prev;
 prev = curr;
 }
 return prev;
 }
}
```

```

};

int main() {
 Solution s;
 int n = 10;
 cout << s.fib(n);
 return 0;
}

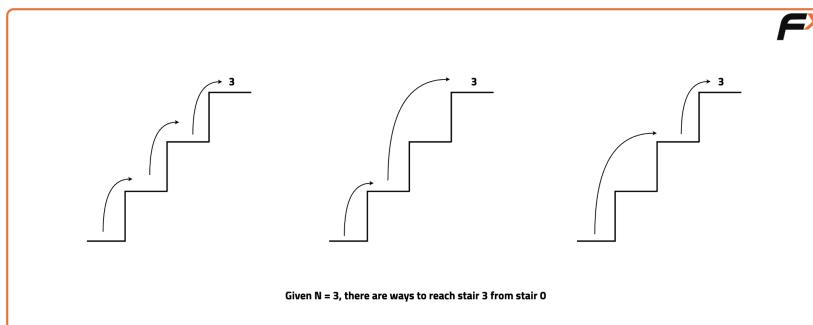
```

## Complexity Analysis

- **Time Complexity:**  $O(n)$   
Reason: Single loop from 2 to n.
- **Space Complexity:**  $O(1)$   
Reason: Only constant extra variables are used.

# Dynamic Programming: Climbing Stairs

You start at stair **0** and want to reach stair **n**.  
At each move, you can climb **1 step** or **2 steps**.  
Return the **total number of distinct ways** to reach stair n.



---

## Key Insight

This problem is identical to a **Fibonacci-style recurrence**.

To reach stair n:

- You must come from n-1 (1 step), or
- From n-2 (2 steps)

So,

$$\text{ways}(n) = \text{ways}(n-1) + \text{ways}(n-2)$$

---

## Base Cases

- $\text{ways}(0) = 1 \rightarrow$  one way (do nothing)
- $\text{ways}(1) = 1 \rightarrow$  one way (1 step)

---

## Approach 1: Tabulation (Bottom-Up DP)

### Algorithm

1. Create a dp array of size n+1

2. Initialize:

- $\text{dp}[0] = 1$

- $\text{dp}[1] = 1$

For  $i = 2 \rightarrow n$ :

$dp[i] = dp[i-1] + dp[i-2]$

3.

4. Return  $dp[n]$

## Code

```
#include <bits/stdc++.h>
using namespace std;

int main() {
 int n = 3;

 vector<int> dp(n + 1, -1);

 dp[0] = 1;
 dp[1] = 1;

 for (int i = 2; i <= n; i++) {
 dp[i] = dp[i - 1] + dp[i - 2];
 }

 cout << dp[n];
 return 0;
}
```

## Complexity

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

---

## Approach 2: Space Optimized DP (Recommended)

We only need the **previous two values**, not the full array.

## Algorithm

1. Maintain two variables:

- o  $\text{prev2} = \text{ways}(i-2)$
- o  $\text{prev1} = \text{ways}(i-1)$

For each step:

$\text{curr} = \text{prev1} + \text{prev2}$

- 2.
3. Shift values forward
4. Return final value

## Code

```
#include <bits/stdc++.h>
using namespace std;

int climbStairs(int n) {
 if (n <= 1) return 1;

 int prev2 = 1; // ways(0)
 int prev1 = 1; // ways(1)

 for (int i = 2; i <= n; i++) {
 int curr = prev1 + prev2;
 prev2 = prev1;
 prev1 = curr;
 }
 return prev1;
}

int main() {
 int n = 3;
```

```
 cout << climbStairs(n);
 return 0;
}
```

## Complexity

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

# Dynamic Programming: Frog Jump (DP-3)

## Problem Summary

A frog starts at stair 0 and wants to reach stair  $n-1$ .

From stair  $i$ , it can jump to:

- $i + 1$
- $i + 2$

Energy cost of a jump from  $i \rightarrow j$ :

$\text{abs}(\text{height}[i] - \text{height}[j])$

Return the **minimum total energy** required to reach stair  $n-1$ .

---

Let

$dp[i]$  = minimum energy required to reach stair  $i$ .

From stair  $i$ , the frog could have come from:

- $i-1$
- $i-2$

So the recurrence is:

```
dp[i] = min(
 dp[i-1] + abs(height[i] - height[i-1]),
 dp[i-2] + abs(height[i] - height[i-2]))
)
```

## Base Case

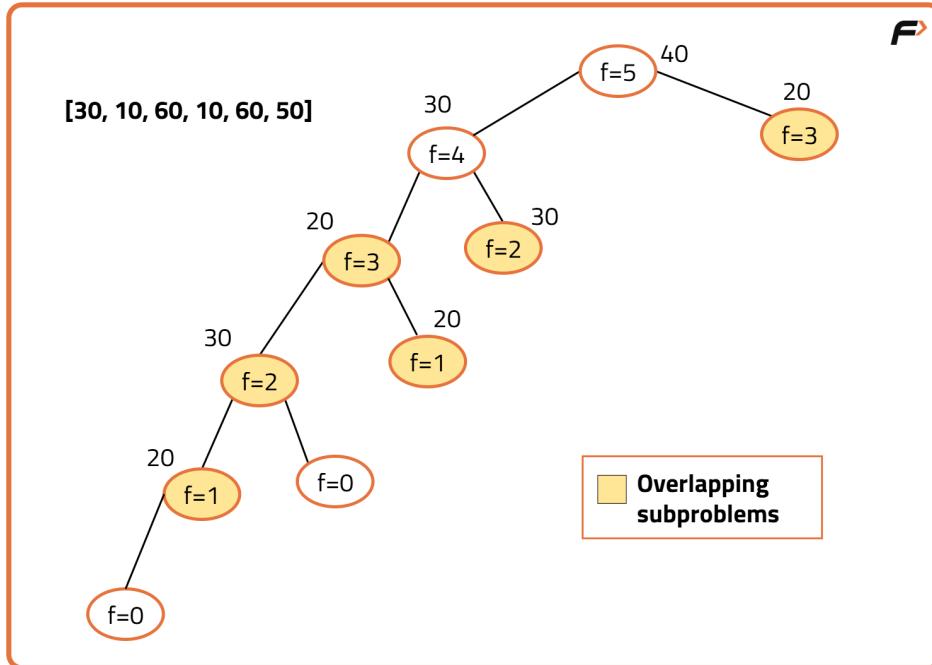
$dp[0] = 0$

---

## 1 Memoization (Top-Down)

### Algorithm

- Define a recursive function  $solve(i)$
- Store already computed results in  $dp[ ]$
- Avoid recomputation of overlapping subproblems



## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int solve(int i, const vector<int>& height, vector<int>& dp) {
 if (i == 0) return 0;
 if (dp[i] != -1) return dp[i];

 int jumpOne = solve(i - 1, height, dp)
 + abs(height[i] - height[i - 1]);

 int jumpTwo = INT_MAX;
 if (i > 1) {
 jumpTwo = solve(i - 2, height, dp)
 + abs(height[i] - height[i - 2]);
 }

 return dp[i] = min(jumpOne, jumpTwo);
 }
}
```

```

int frogJump(vector<int>& height) {
 int n = height.size();
 vector<int> dp(n, -1);
 return solve(n - 1, height, dp);
}
};

```

## Complexity

- **Time:**  $O(n)$
  - **Space:**  $O(n)$  (DP + recursion stack)
- 

## 2 Tabulation (Bottom-Up)

### Algorithm

- Build the solution iteratively
- Each state depends only on previous two states

### Code

```

#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int frogJump(const vector<int>& height) {
 int n = height.size();
 vector<int> dp(n, INT_MAX);

 dp[0] = 0;

 for (int i = 1; i < n; i++) {

```

```

 int jumpOne = dp[i - 1] + abs(height[i] - height[i - 1]);
 int jumpTwo = INT_MAX;

 if (i > 1) {
 jumpTwo = dp[i - 2] + abs(height[i] - height[i - 2]);
 }

 dp[i] = min(jumpOne, jumpTwo);
 }
 return dp[n - 1];
}
};

```

## Complexity

- **Time:**  $O(n)$
  - **Space:**  $O(n)$
- 

## ③ Space Optimized DP Key Observation

To compute  $dp[i]$ , we only need:

- $dp[i-1]$
- $dp[i-2]$

So we can remove the DP array.

## Code

```

#include <bits/stdc++.h>
using namespace std;

class Solution {
public:

```

```

int frogJump(const vector<int>& height) {
 int n = height.size();
 if (n == 1) return 0;

 int prev = 0; // dp[i-1]
 int prev2 = 0; // dp[i-2]

 for (int i = 1; i < n; i++) {
 int jumpOne = prev + abs(height[i] - height[i - 1]);
 int jumpTwo = INT_MAX;

 if (i > 1) {
 jumpTwo = prev2 + abs(height[i] - height[i - 2]);
 }

 int cur = min(jumpOne, jumpTwo);
 prev2 = prev;
 prev = cur;
 }
 return prev;
}

```

## Complexity

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

# Dynamic Programming: Frog Jump with k Distances (DP 4)

A frog wants to reach the last step of a staircase. There are  $n$  steps, and each step has a height given in an array `heights`. The frog starts at index 0 and wants to reach index  $n-1$ .

From any step  $i$ , the frog can jump to any step between  $i+1$  and  $i+k$  (as long as that step exists).

The energy required for a jump from step  $i$  to step  $j$  is  $\text{abs}(\text{heights}[i] - \text{heights}[j])$ .

The task is to find the **minimum total energy** needed to reach the last step.

Example explanation:

For `heights` = [10, 5, 20, 0, 15] and  $k = 2$

The frog jumps:

- Step 0 → Step 2, cost =  $|10 - 20| = 10$
- Step 2 → Step 4, cost =  $|20 - 15| = 5$   
Total energy = 15

---

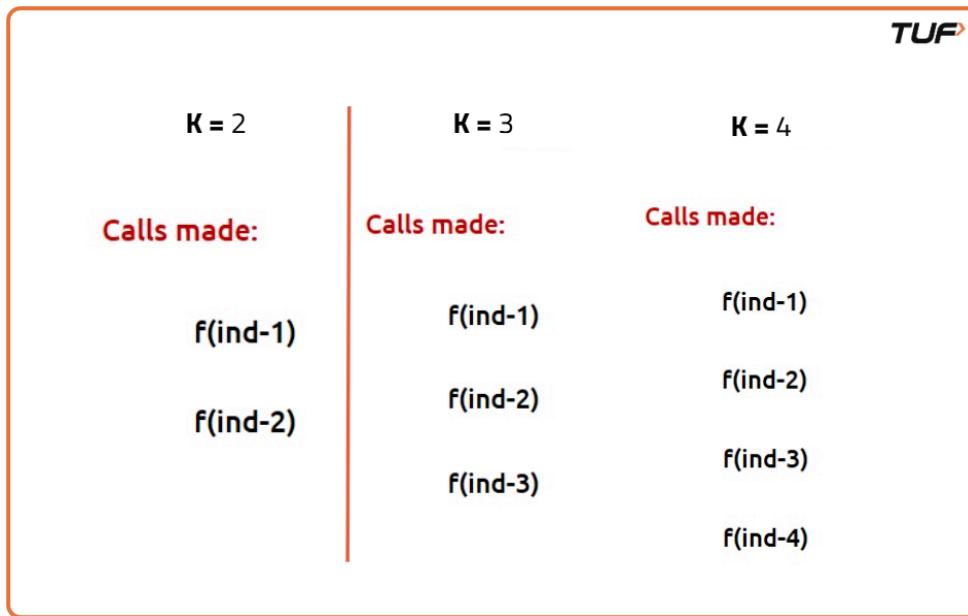
## Approach 1: Memoization Approach

### Algorithm

We use recursion with memoization to avoid recalculating the same states again and again.

- Define a function `solveUtil(ind)` that returns the minimum energy required to reach step `ind`.
- Base case:  
If `ind == 0`, the cost is 0 because the frog starts there.
- If the answer for `ind` is already stored in the `dp` array, return it.
- Otherwise, try all jumps from 1 to  $k$ :

- If  $\text{ind} - j \geq 0$ , calculate the cost of jumping from  $\text{ind}-j$  to  $\text{ind}$ .
- Take the minimum of all possible jump costs.
- Store the result in  $\text{dp}[\text{ind}]$  and return it.
- The final answer is  $\text{solveUtil}(n-1)$ .



This ensures we explore all valid paths but compute each state only once.

### Code

```
#include <bits/stdc++.h>
using namespace std;

// Function to find the minimum cost to reach index 'ind' using at most 'k' jumps
int solveUtil(int ind, vector<int>& height, vector<int>& dp, int k) {
 if (ind == 0) return 0;
 if (dp[ind] != -1) return dp[ind];
 int minSteps = INT_MAX;
 for (int j = 1; j <= k; j++) {
 if (ind - j >= 0) {
 int jump = solveUtil(ind - j, height, dp, k)
 + abs(height[ind] - height[ind - j]);
 minSteps = min(minSteps, jump);
 }
 }
 dp[ind] = minSteps;
 return dp[ind];
}
```

```

 mmSteps = min(mmSteps, jump);
 }
}

return dp[ind] = mmSteps;
}

int solve(int n, vector<int>& height, int k) {
 vector<int> dp(n, -1);
 return solveUtil(n - 1, height, dp, k);
}

int main() {
 vector<int> height{30, 10, 60, 10, 60, 50};
 int n = height.size();
 int k = 2;
 cout << solve(n, height, k) << endl;
 return 0;
}

```

## Complexity Analysis

- Time Complexity:  $O(n * k)$   
For each index, we try up to  $k$  jumps, and each state is computed only once due to memoization.
  - Space Complexity:  $O(n)$   
The dp array stores results for all  $n$  indices. The recursion stack also goes up to  $n$  in the worst case.
- 

## Approach 2: Tabulation Approach

### Algorithm

This approach solves the problem using bottom-up dynamic programming.

- Create a dp array where  $dp[i]$  represents the minimum energy required to reach step  $i$ .

- Initialize  $dp[0] = 0$  because no energy is needed at the start.
- For each step  $i$  from 1 to  $n-1$ :
  - Try all jumps from 1 to  $k$ .
  - If  $i - j \geq 0$ , compute:  

$$dp[i - j] + \text{abs}(\text{height}[i] - \text{height}[i - j])$$
  - Store the minimum of all such values in  $dp[i]$ .
- The answer is stored in  $dp[n-1]$ .

This builds the solution step by step without recursion.

### Code

```
#include <bits/stdc++.h>
using namespace std;

// Function to compute the minimum cost to reach the end using at most
// 'k' jumps
int solveUtil(int n, vector<int>& height, vector<int>& dp, int k) {
 dp[0] = 0;
 for (int i = 1; i < n; i++) {
 int mmSteps = INT_MAX;
 for (int j = 1; j <= k; j++) {
 if (i - j >= 0) {
 int jump = dp[i - j] + abs(height[i] - height[i - j]);
 mmSteps = min(mmSteps, jump);
 }
 }
 dp[i] = mmSteps;
 }
 return dp[n - 1];
}

int solve(int n, vector<int>& height, int k) {
 vector<int> dp(n, -1);
 return solveUtil(n, height, dp, k);
}
```

```

 }

int main() {
 vector<int> height{30, 10, 60, 10, 60, 50};
 int n = height.size();
 int k = 2;
 cout << solve(n, height, k) << endl;
 return 0;
}

```

### Complexity Analysis

- Time Complexity:  $O(n * k)$   
For each of the  $n$  steps, we check up to  $k$  possible jumps.
- Space Complexity:  $O(n)$   
The dp array stores the minimum cost for each step.

## Maximum sum of non-adjacent elements (DP 5)

Given an array of  $N$  positive integers, we need to find the maximum sum of a subsequence such that no two chosen elements are adjacent in the original array.

A subsequence means we can delete some elements, but the remaining elements must stay in the same order.

Example explanation:

For nums = [2, 1, 4, 9]

If we choose 2 and 9, they are not adjacent and their sum is 11, which is the maximum possible.

---

## Approach 1: Memoization

### Algorithm

We use recursion with the pick / not-pick idea.

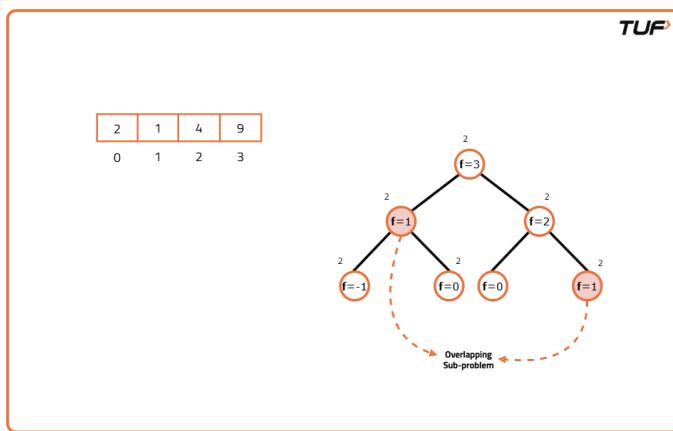
At every index  $i$ , we have two choices:

- Pick the current element  $\text{arr}[i]$ . If we pick it, we cannot pick the adjacent element, so we move to index  $i-2$ .
- Do not pick the current element. Then we move to index  $i-1$ .

We define a recursive function that returns the maximum sum we can get up to index  $i$ .

- If  $i < 0$ , return 0 because no elements are left.
- If  $i == 0$ , return  $\text{arr}[0]$  because only one element is available.
- If the answer for index  $i$  is already computed, return it from the dp array.
- Otherwise, compute both choices (pick and not pick), take the maximum, store it in  $\text{dp}[i]$ , and return it.

The final answer is the value computed for the last index.



### Code

```
#include <bits/stdc++.h>
using namespace std;
```

```

class Solution {
public:
 int solve(vector<int>& arr, int i, vector<int>& dp) {
 if (i < 0) return 0;
 if (i == 0) return arr[0];
 if (dp[i] != -1) return dp[i];
 int pick = arr[i] + solve(arr, i - 2, dp);
 int notPick = solve(arr, i - 1, dp);
 return dp[i] = max(pick, notPick);
 }

 int maximumNonAdjacentSum(vector<int>& arr) {
 int n = arr.size();
 vector<int> dp(n, -1);
 return solve(arr, n - 1, dp);
 }
};

int main() {
 vector<int> arr = {2, 1, 4, 9};
 Solution obj;
 cout << obj.maximumNonAdjacentSum(arr);
 return 0;
}

```

## Complexity Analysis

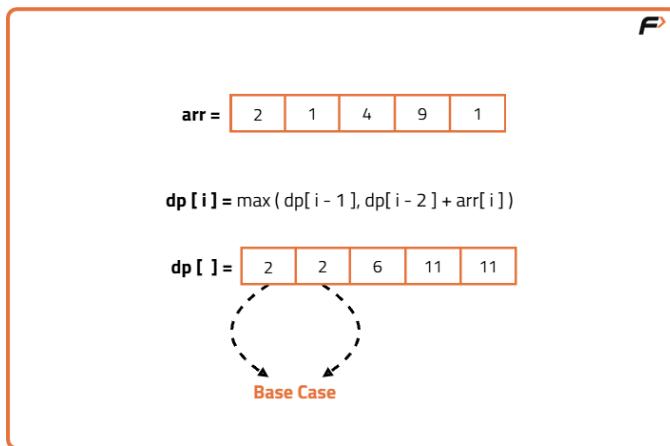
- Time Complexity:  $O(N)$   
Each index is solved once and stored in the dp array.
  - Space Complexity:  $O(N + N)$   
One  $O(N)$  for the dp array and one  $O(N)$  for the recursion stack.
- 

## Approach 2: Tabulation

### Algorithm

This is a bottom-up dynamic programming approach.

- Create a dp array where  $dp[i]$  stores the maximum sum possible up to index  $i$ .
- $dp[0] = arr[0]$  because only one element is available.
- $dp[1] = \max(arr[0], arr[1])$  because we can pick only one of them.
- For every index  $i$  from 2 to  $n-1$ :
  - If we pick  $arr[i]$ , the sum becomes  $arr[i] + dp[i-2]$ .
  - If we do not pick it, the sum is  $dp[i-1]$ .
  - Store the maximum of these two in  $dp[i]$ .
- The answer is  $dp[n-1]$ .



## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int maximumNonAdjacentSum(vector<int>& arr) {
 int n = arr.size();
 if (n == 1) return arr[0];
 vector<int> dp(n);
```

```

 dp[0] = arr[0];
 dp[1] = max(arr[0], arr[1]);
 for (int i = 2; i < n; i++) {
 dp[i] = max(arr[i] + dp[i - 2], dp[i - 1]);
 }
 return dp[n - 1];
 }

};

int main() {
 vector<int> arr = {2, 1, 4, 9};
 Solution obj;
 cout << obj.maximumNonAdjacentSum(arr);
 return 0;
}

```

## Complexity Analysis

- Time Complexity:  $O(N)$   
Each element is processed once.
  - Space Complexity:  $O(N)$   
Extra space is used for the dp array.
- 

## Approach 3: Space Optimization

### Algorithm

In the tabulation approach, we only need the results of the previous two indices to compute the current answer. So instead of using a full dp array, we use two variables.

- prev stores the answer up to the previous index.
- prev2 stores the answer up to two indices back.
- For each element:

- Include current element:  $\text{nums}[i] + \text{prev2}$
  - Exclude current element:  $\text{prev}$
  - Take the maximum of both and store it in a temporary variable.
  - Update  $\text{prev2}$  and  $\text{prev}$ .
- The final answer is stored in  $\text{prev}$ .

### Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int maxSum(vector<int>& nums) {
 if (nums.empty()) return 0;
 int prev2 = 0;
 int prev = nums[0];
 for (int i = 1; i < nums.size(); i++) {
 int include = nums[i] + prev2;
 int exclude = prev;
 int curr = max(include, exclude);
 prev2 = prev;
 prev = curr;
 }
 return prev;
 }
};

int main() {
 vector<int> arr = {3, 2, 5, 10, 7};
 Solution obj;
 cout << obj.maxSum(arr) << endl;
 return 0;
}
```

## Complexity Analysis

- Time Complexity:  $O(N)$   
Each element is processed once.
- Space Complexity:  $O(1)$   
Only constant extra space is used.

# Dynamic Programming: House Robber (DP 6)

A thief wants to rob houses arranged in a circular street. Each house has some money stored in it, given in an array Arr.

The security system is active such that if two adjacent houses are robbed, the police will be notified.

Because the houses are in a **circle**, the first and the last house are also adjacent.

The task is to find the **maximum amount of money** the thief can rob without robbing any two adjacent houses.

Example explanation:

For money = [2, 1, 4, 9]

The thief cannot rob both house 0 and house 3 together because they are adjacent in a circular arrangement.

The maximum loot is obtained by robbing houses with money 1 and 9, giving total 10.

---

## Approach

### Algorithm

This problem is a variation of the linear House Robber problem.

Because the houses are circular:

- We cannot pick both the first and the last house together.

So we split the problem into **two linear subproblems**:

1. Exclude the first house and consider houses from index 1 to n-1
2. Exclude the last house and consider houses from index 0 to n-2

For both cases, we solve the **linear House Robber problem** and take the maximum of the two results.

For the linear solution:

- prev stores the maximum money robbed till the previous house
- prev2 stores the maximum money robbed till the house before the previous one
- At each house, we decide:
  - Pick the current house and add prev2
  - Or skip the current house and take prev
- We update prev and prev2 accordingly

This ensures no two adjacent houses are robbed.

### Code

```
#include <bits/stdc++.h>
using namespace std;

// Function to solve the linear house robber problem
long long int solve(vector<int>& arr) {
 int n = arr.size();
 if (n == 1) return arr[0];
 long long int prev = arr[0];
 long long int prev2 = 0;
```

```

 for (int i = 1; i < n; i++) {
 long long int pick = arr[i];
 if (i > 1) pick += prev2;
 long long int nonPick = prev;
 long long int cur_i = max(pick, nonPick);
 prev2 = prev;
 prev = cur_i;
 }
 return prev;
 }

 // Function to solve the circular house robber problem
 long long int robStreet(int n, vector<int> &arr) {
 if (n == 0) return 0;
 if (n == 1) return arr[0];
 vector<int> arr1, arr2;
 for (int i = 0; i < n; i++) {
 if (i != 0) arr1.push_back(arr[i]);
 if (i != n - 1) arr2.push_back(arr[i]);
 }
 long long int ans1 = solve(arr1);
 long long int ans2 = solve(arr2);
 return max(ans1, ans2);
 }

 int main() {
 vector<int> arr{1, 5, 1, 2, 6};
 int n = arr.size();
 cout << robStreet(n, arr);
 return 0;
 }
}

```

## Complexity Analysis

- Time Complexity:  $O(n)$   
We traverse the houses twice (excluding first and excluding last), each in linear time.
- Space Complexity:  $O(1)$   
Only constant extra space is used for variables like `prev` and `prev2`.

# Dynamic Programming: Ninja's Training (DP 7)

A ninja is training for n days. Each day, he can perform **one of three activities**:

- Running
- Fighting practice
- Stealth training

The points gained for each activity on each day are given in a 2D matrix `points`, where `points[day][activity]` represents the points earned.

The rule is that **the ninja cannot perform the same activity on two consecutive days**. We need to find the **maximum total points** the ninja can earn over all days.

Example explanation:

For

`[[10, 40, 70], [20, 50, 80], [30, 60, 90]]`

One optimal choice is:

- Day 1: Fighting practice = 70
- Day 2: Stealth training = 50
- Day 3: Fighting practice = 90

Total = 210, which is the maximum possible.

---

## Approach 1: Memoization Approach

### Algorithm

We use recursion with memoization.

- Define a function  $f(\text{day}, \text{last})$  that returns the maximum points from day 0 to day, where  $\text{last}$  is the activity done on the previous day.
- $\text{last}$  can be 0, 1, 2 (activities) or 3 (no activity restriction).
- Base case:
  - If  $\text{day} == 0$ , choose the maximum activity on day 0 that is not equal to  $\text{last}$ .
- For other days:
  - Try all activities except  $\text{last}$ .
  - Add current day's points to the result of  $f(\text{day}-1, \text{current\_activity})$ .
  - Take the maximum.
- Store results in a  $\text{dp}[\text{day}][\text{last}]$  table to avoid recomputation.
- Final answer is  $f(n-1, 3)$ .

### Code

```
#include <bits/stdc++.h>
using namespace std;

int f(int day, int last, vector<vector<int>> &points,
vector<vector<int>> &dp) {
 if (dp[day][last] != -1) return dp[day][last];
 if (day == 0) {
 int maxi = 0;
 for (int i = 0; i <= 2; i++) {
 if (i != last)
 maxi = max(maxi, points[0][i]);
 }
 return dp[day][last] = maxi;
 }
 int maxi = 0;
 for (int i = 0; i <= 2; i++) {
 if (i != last) {
```

```

 int activity = points[day][i] + f(day - 1, i, points, dp);
 maxi = max(maxi, activity);
 }
}

return dp[day][last] = maxi;
}

int ninjaTraining(int n, vector<vector<int>> &points) {
 vector<vector<int>> dp(n, vector<int>(4, -1));
 return f(n - 1, 3, points, dp);
}

int main() {
 vector<vector<int>> points = {{10, 40, 70},
 {20, 50, 80},
 {30, 60, 90}};
 int n = points.size();
 cout << ninjaTraining(n, points);
}

```

## Complexity Analysis

- Time Complexity:  $O(n \times 4 \times 3) = O(n)$   
For each day, we try 4 possible last values and 3 activities.
  - Space Complexity:  $O(n \times 4) + O(n)$   
 $O(n \times 4)$  for the DP table and  $O(n)$  for recursion stack.
- 

## Approach 2: Tabulation Approach

### Algorithm

This is a bottom-up DP solution.

- Create a DP table  $dp[day][last]$ .

- On day 0, initialize  $dp[0][last]$  with the best activity excluding last.
- For each next day:
  - For every possible last, try all activities not equal to last.
  - Update  $dp[day][last]$  with the maximum achievable points.
- The final answer is stored in  $dp[n-1][3]$ .

### Code

```
#include <bits/stdc++.h>
using namespace std;

int ninjaTraining(int n, vector<vector<int>>& points) {
 vector<vector<int>> dp(n, vector<int>(4, 0));

 dp[0][0] = max(points[0][1], points[0][2]);
 dp[0][1] = max(points[0][0], points[0][2]);
 dp[0][2] = max(points[0][0], points[0][1]);
 dp[0][3] = max(points[0][0], max(points[0][1], points[0][2]));

 for (int day = 1; day < n; day++) {
 for (int last = 0; last < 4; last++) {
 dp[day][last] = 0;
 for (int task = 0; task <= 2; task++) {
 if (task != last) {
 int activity = points[day][task] + dp[day - 1][task];
 dp[day][last] = max(dp[day][last], activity);
 }
 }
 }
 }
 return dp[n - 1][3];
}

int main() {
 vector<vector<int>> points = {{10, 40, 70},
 {20, 50, 80},
```

```

 {30, 60, 90}};

int n = points.size();
cout << ninjaTraining(n, points);
}

```

## Complexity Analysis

- Time Complexity:  $O(n \times 4 \times 3) = O(n)$
  - Space Complexity:  $O(n \times 4)$   
DP table stores results for all days and last activities.
- 

## Approach 3: Space Optimization Approach

### Algorithm

We observe that to compute the current day, we only need results from the previous day.

- Use two arrays of size 4:
  - prev for previous day
  - temp for current day
- Initialize prev for day 0.
- For each day:
  - Compute temp[last] by checking all task != last.
  - Assign prev = temp.
- Final answer is prev[3].

### Code

```
#include <bits/stdc++.h>
using namespace std;
```

```

int ninjaTraining(int n, vector<vector<int>>& points) {
 vector<int> prev(4, 0);

 prev[0] = max(points[0][1], points[0][2]);
 prev[1] = max(points[0][0], points[0][2]);
 prev[2] = max(points[0][0], points[0][1]);
 prev[3] = max(points[0][0], max(points[0][1], points[0][2]));

 for (int day = 1; day < n; day++) {
 vector<int> temp(4, 0);
 for (int last = 0; last < 4; last++) {
 for (int task = 0; task <= 2; task++) {
 if (task != last) {
 temp[last] = max(temp[last], points[day][task] +
prev[task]);
 }
 }
 }
 prev = temp;
 }
 return prev[3];
}

int main() {
 vector<vector<int>> points = {{10, 40, 70},
 {20, 50, 80},
 {30, 60, 90}};
 int n = points.size();
 cout << ninjaTraining(n, points);
}

```

## Complexity Analysis

- Time Complexity:  $O(n \times 4 \times 3) = O(n)$
- Space Complexity:  $O(1)$   
Only constant-sized arrays are used.

# Grid Unique Paths : DP on Grids (DP 8)

You are given a grid with  $m$  rows and  $n$  columns. You start from the top-left cell  $(0, 0)$  and want to reach the bottom-right cell  $(m-1, n-1)$ .

From any cell, you are allowed to move **only in two directions**:

- Right
- Down

You need to return the **total number of unique paths** from the start cell to the destination cell.

Example explanation:

For  $m = 3$ ,  $n = 2$

You need to make exactly 1 right move and 2 down moves.

All different orders of these moves give 3 unique paths.

---

## Approach 1: Memoization Approach

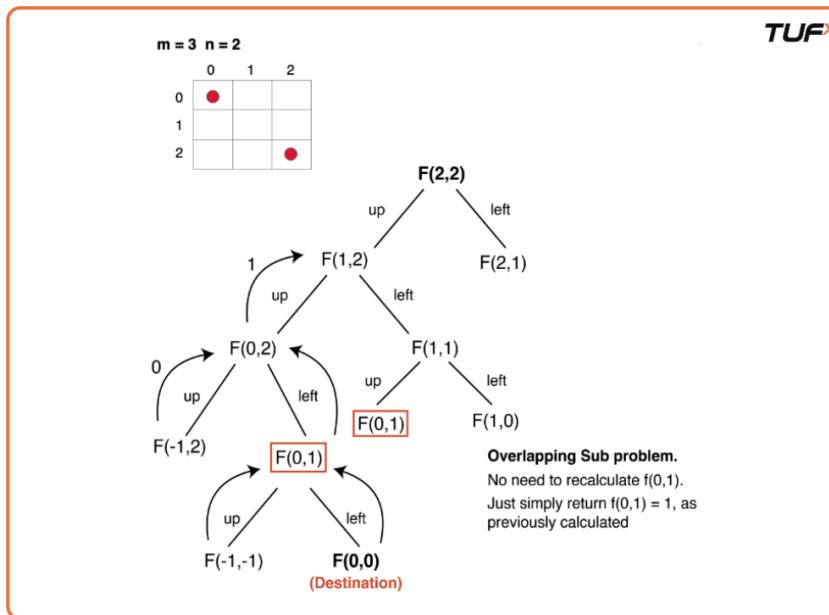
### Algorithm

We use recursion with memoization.

- Define a recursive function  $\text{func}(i, j)$  that returns the number of ways to reach cell  $(i, j)$  from  $(0, 0)$ .
- Base cases:
  - If  $i == 0$  and  $j == 0$ , there is exactly 1 way (starting point).
  - If  $i < 0$  or  $j < 0$ , return 0 because it is out of bounds.
- If the value for  $(i, j)$  is already stored in  $\text{dp}$ , return it.
- Otherwise:

- Ways from top =  $\text{func}(i-1, j)$
- Ways from left =  $\text{func}(i, j-1)$
- Total ways = top + left
- Store the result in  $\text{dp}[i][j]$ .

The final answer is  $\text{func}(m-1, n-1)$ .



## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
private:
 int func(int i, int j, vector<vector<int>>& dp){
 if (i == 0 && j == 0) return 1;
 if (i < 0 || j < 0) return 0;
 if (dp[i][j] != -1) return dp[i][j];
 int up = func(i - 1, j, dp);
 int left = func(i, j - 1, dp);
 return dp[i][j] = up + left;
 }
}
```

```

public:
 int uniquePaths(int m, int n) {
 vector<vector<int>> dp(m, vector<int>(n, -1));
 return func(m - 1, n - 1, dp);
 }
};

int main() {
 int m = 3, n = 2;
 Solution sol;
 cout << sol.uniquePaths(m, n);
 return 0;
}

```

## Complexity Analysis

- Time Complexity:  $O(m * n)$   
Each cell  $(i, j)$  is computed only once due to memoization.
  - Space Complexity:  $O(m * n)$   
DP table stores results for all grid cells.
- 

## Approach 2: Tabulation Approach

### Algorithm

This is a bottom-up dynamic programming solution.

- Create a dp array of size  $m \times n$ .
- $dp[i][j]$  represents the number of ways to reach cell  $(i, j)$ .
- Base case:
  - $dp[0][0] = 1$
- For each cell  $(i, j)$ :

- Ways from top =  $dp[i-1][j]$  if  $i > 0$
- Ways from left =  $dp[i][j-1]$  if  $j > 0$
- $dp[i][j] = \text{up} + \text{left}$
- The final answer is stored in  $dp[m-1][n-1]$ .

### Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
private:
 int func(int m, int n, vector<vector<int>>& dp) {
 for (int i = 0; i < m; i++) {
 for (int j = 0; j < n; j++) {
 if (i == 0 && j == 0) {
 dp[i][j] = 1;
 continue;
 }
 int up = 0, left = 0;
 if (i > 0) up = dp[i - 1][j];
 if (j > 0) left = dp[i][j - 1];
 dp[i][j] = up + left;
 }
 }
 return dp[m - 1][n - 1];
 }
public:
 int uniquePaths(int m, int n) {
 vector<vector<int>> dp(m, vector<int>(n, -1));
 return func(m, n, dp);
 }
};

int main() {
 int m = 3, n = 2;
```

```

 Solution sol;
 cout << sol.uniquePaths(m, n);
 return 0;
}

```

## Complexity Analysis

- Time Complexity:  $O(m * n)$   
Each grid cell is processed once.
  - Space Complexity:  $O(m * n)$   
A 2D DP table is used.
- 

## Approach 3: Space Optimized Approach

### Algorithm

From the relation  $dp[i][j] = dp[i-1][j] + dp[i][j-1]$ , we observe:

- To compute the current row, we only need the previous row and the current row.

So we use:

- prev array → previous row
- temp array → current row

Steps:

- Initialize prev with zeros.
- For each row:
  - Build temp using values from prev (up) and temp itself (left).
  - Assign  $prev = temp$ .

- Final answer is  $\text{prev}[n-1]$ .

**TUF**

| $\text{prev}[n]$ | 0 | 0 | 0   | 0       |
|------------------|---|---|-----|---------|
| $i \setminus j$  | 0 | 1 | ... | $n - 1$ |
| 0                | 1 | 1 | 1   | 1       |
| 1                | 1 | 1 |     |         |
| :                |   |   |     |         |
| $m - 1$          |   |   |     |         |

$\text{dp}[i][j] = \text{dp}[1-i][j] + \text{dp}[1-j][i]$

This row's cells only need row prev's values and current row's previous values.

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
private:
 int func(int m, int n){
 vector<int> prev(n, 0);
 for (int i = 0; i < m; i++) {
 vector<int> temp(n, 0);
 for (int j = 0; j < n; j++) {
 if (i == 0 && j == 0) {
 temp[j] = 1;
 continue;
 }
 int up = 0, left = 0;
 if (i > 0) up = prev[j];
 if (j > 0) left = temp[j - 1];
 temp[j] = up + left;
 }
 prev = temp;
 }
 return prev[n - 1];
 }
}
```

```

 }
public:
 int uniquePaths(int m, int n) {
 return func(m, n);
 }
};

int main() {
 int m = 3, n = 2;
 Solution sol;
 cout << sol.uniquePaths(m, n);
 return 0;
}

```

### Complexity Analysis

- Time Complexity:  $O(m * n)$   
Each cell is still processed once.
- Space Complexity:  $O(n)$   
Only two 1D arrays of size n are used.

## Grid Unique Paths 2 (DP 9)

You are given an  $m \times n$  grid where each cell contains either 0 or 1.

0 means the cell is free, and 1 means the cell is blocked.

You start from the top-left cell  $(0, 0)$  and want to reach the bottom-right cell  $(m-1, n-1)$ .

You can move **only right or down**.

If a cell is blocked, you cannot pass through it.

Return the **total number of unique paths** from start to destination.

Example explanation:

For

$[[0, 0, 0], [0, 1, 0], [0, 0, 0]]$

The middle cell is blocked, so only two valid paths exist that go around it.

---

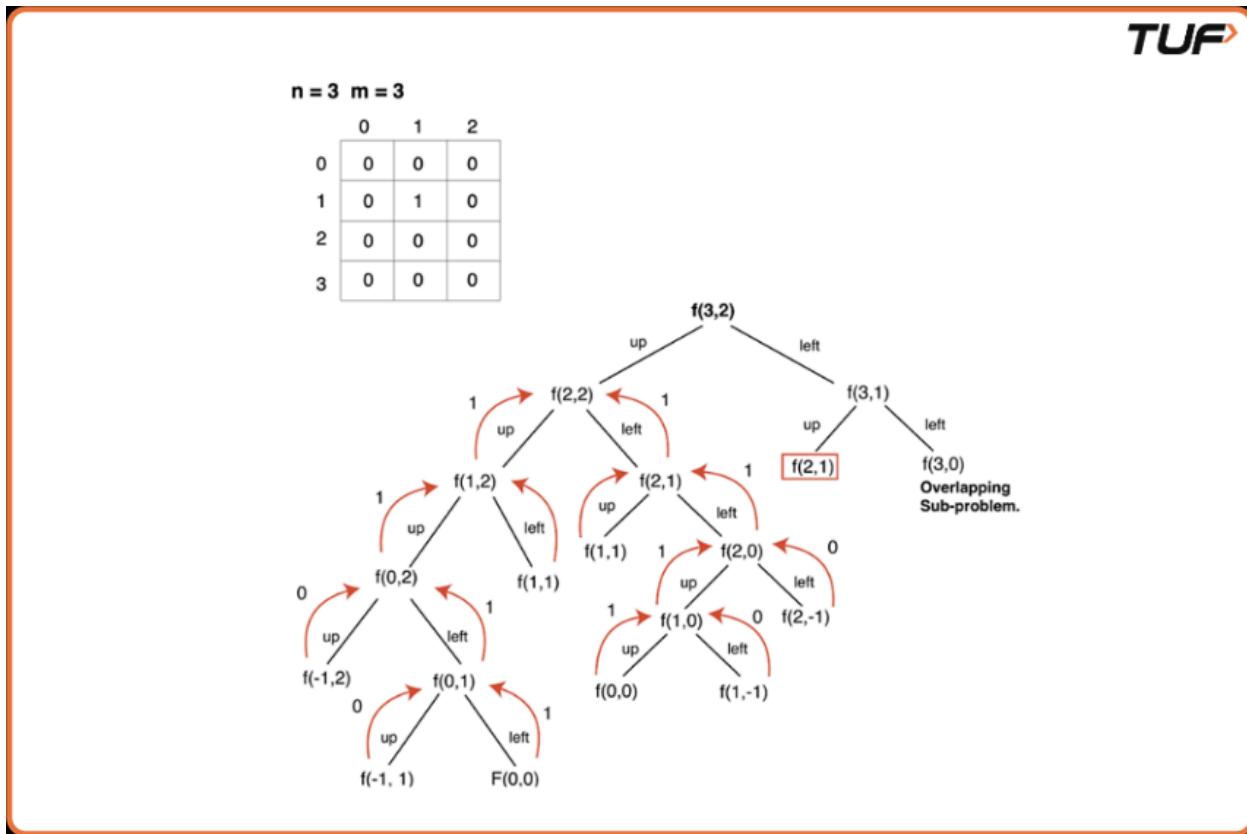
## Approach 1: Memoization Approach

### Algorithm

We use recursion with memoization.

- Define a function `func(i, j)` that returns the number of ways to reach cell  $(i, j)$  from  $(0, 0)$ .
- Base cases:
  - If  $i < 0$  or  $j < 0$ , return 0 (out of bounds).
  - If `matrix[i][j] == 1`, return 0 (blocked cell).
  - If  $i == 0$  and  $j == 0$ , return 1 (starting point).
- If `dp[i][j]` is already computed, return it.
- Otherwise:
  - Ways from top = `func(i-1, j)`
  - Ways from left = `func(i, j-1)`
  - Store and return `dp[i][j] = up + left.`

The final answer is `func(m-1, n-1)`.



## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
private:
 int func(int i, int j, vector<vector<int>>& matrix,
vector<vector<int>>& dp) {
 if (i < 0 || j < 0 || matrix[i][j] == 1) return 0;
 if (i == 0 && j == 0) return 1;
 if (dp[i][j] != -1) return dp[i][j];
 int up = func(i - 1, j, matrix, dp);
 int left = func(i, j - 1, matrix, dp);
 return dp[i][j] = up + left;
 }
public:
 int uniquePathsWithObstacles(vector<vector<int>>& matrix) {
 int m = matrix.size();
 int n = matrix[0].size();
```

```

 vector<vector<int>> dp(m, vector<int>(n, -1));
 return func(m - 1, n - 1, matrix, dp);
 }
};

int main() {
 vector<vector<int>> maze{{0, 0, 0}, {0, 1, 0}, {0, 0, 0}};
 Solution sol;
 cout << sol.uniquePathsWithObstacles(maze);
 return 0;
}

```

## Complexity Analysis

- Time Complexity:  $O(m * n)$   
Each cell is computed once due to memoization.
  - Space Complexity:  $O(m * n) + O(m + n)$   
DP table plus recursion stack depth.
- 

## Approach 2: Tabulation Approach

### Algorithm

This is a bottom-up DP approach.

- Create a DP table  $dp[m][n]$  where  $dp[i][j]$  stores number of ways to reach  $(i, j)$ .
- If a cell is blocked ( $matrix[i][j] == 1$ ), set  $dp[i][j] = 0$ .
- Base case:
  - If  $(i, j) == (0, 0)$  and not blocked,  $dp[0][0] = 1$ .
- For every cell:
  - Ways from top =  $dp[i-1][j]$  if  $i > 0$

- Ways from left =  $dp[i][j-1]$  if  $j > 0$
- $dp[i][j] = up + left$
- Final answer is  $dp[m-1][n-1]$ .

### Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
private:
 int func(int m, int n, vector<vector<int>>& matrix,
vector<vector<int>>& dp) {
 for (int i = 0; i < m; i++) {
 for (int j = 0; j < n; j++) {
 if (matrix[i][j] == 1) {
 dp[i][j] = 0;
 continue;
 }
 if (i == 0 && j == 0) {
 dp[i][j] = 1;
 continue;
 }
 int up = (i > 0) ? dp[i - 1][j] : 0;
 int left = (j > 0) ? dp[i][j - 1] : 0;
 dp[i][j] = up + left;
 }
 }
 return dp[m - 1][n - 1];
 }
public:
 int uniquePathsWithObstacles(vector<vector<int>>& matrix) {
 int m = matrix.size();
 int n = matrix[0].size();
 vector<vector<int>> dp(m, vector<int>(n, 0));
 return func(m, n, matrix, dp);
 }
}
```

```

};

int main() {
 vector<vector<int>> maze{{0,0,0},{0,1,0},{0,0,0}};
 Solution sol;
 cout << sol.uniquePathsWithObstacles(maze);
 return 0;
}

```

## Complexity Analysis

- Time Complexity:  $O(m * n)$   
Each grid cell is processed once.
  - Space Complexity:  $O(m * n)$   
A 2D DP table is used.
- 

## Approach 3: Space Optimized Approach

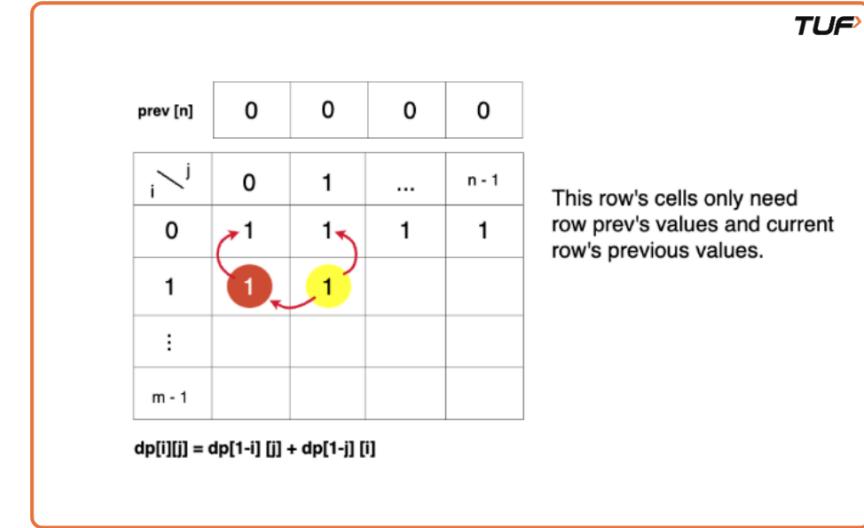
### Algorithm

From the recurrence

$dp[i][j] = dp[i-1][j] + dp[i][j-1]$ ,  
we only need the previous row and current row.

- Use two 1D arrays:
  - $prev \rightarrow$  previous row
  - $curr \rightarrow$  current row
- If a cell is blocked, set  $curr[j] = 0$ .
- Otherwise:
  - $up = prev[j]$

- `left = curr[j-1]`
  - `curr[j] = up + left`
- After each row, assign `prev = curr`.
- Final answer is `prev[n-1]`.



## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
private:
 int func(int m, int n, vector<vector<int>>& matrix) {
 vector<int> prev(n, 0), curr(n, 0);
 for (int i = 0; i < m; i++) {
 for (int j = 0; j < n; j++) {
 if (matrix[i][j] == 1) {
 curr[j] = 0;
 continue;
 }
 if (i == 0 && j == 0) {
 curr[j] = 1;
 continue;
 }
 }
 }
 return curr[n-1];
 }
};
```

```

 int up = (i > 0) ? prev[j] : 0;
 int left = (j > 0) ? curr[j - 1] : 0;
 curr[j] = up + left;
 }
 prev = curr;
}
return prev[n - 1];
}

public:
int uniquePathsWithObstacles(vector<vector<int>>& matrix) {
 int m = matrix.size();
 int n = matrix[0].size();
 return func(m, n, matrix);
}
};

int main() {
 vector<vector<int>> maze{{0,0,0},{0,1,0},{0,0,0}};
 Solution sol;
 cout << sol.uniquePathsWithObstacles(maze);
 return 0;
}

```

## Complexity Analysis

- Time Complexity:  $O(m * n)$   
Each cell is processed once.
- Space Complexity:  $O(n)$   
Only two 1D arrays of size  $n$  are used.

# Minimum Path Sum In a Grid (DP 10)

You are given an  $m \times n$  grid filled with **non-negative numbers**.

You start from the **top-left cell (0,0)** and want to reach the **bottom-right cell ( $m-1, n-1$ )**.

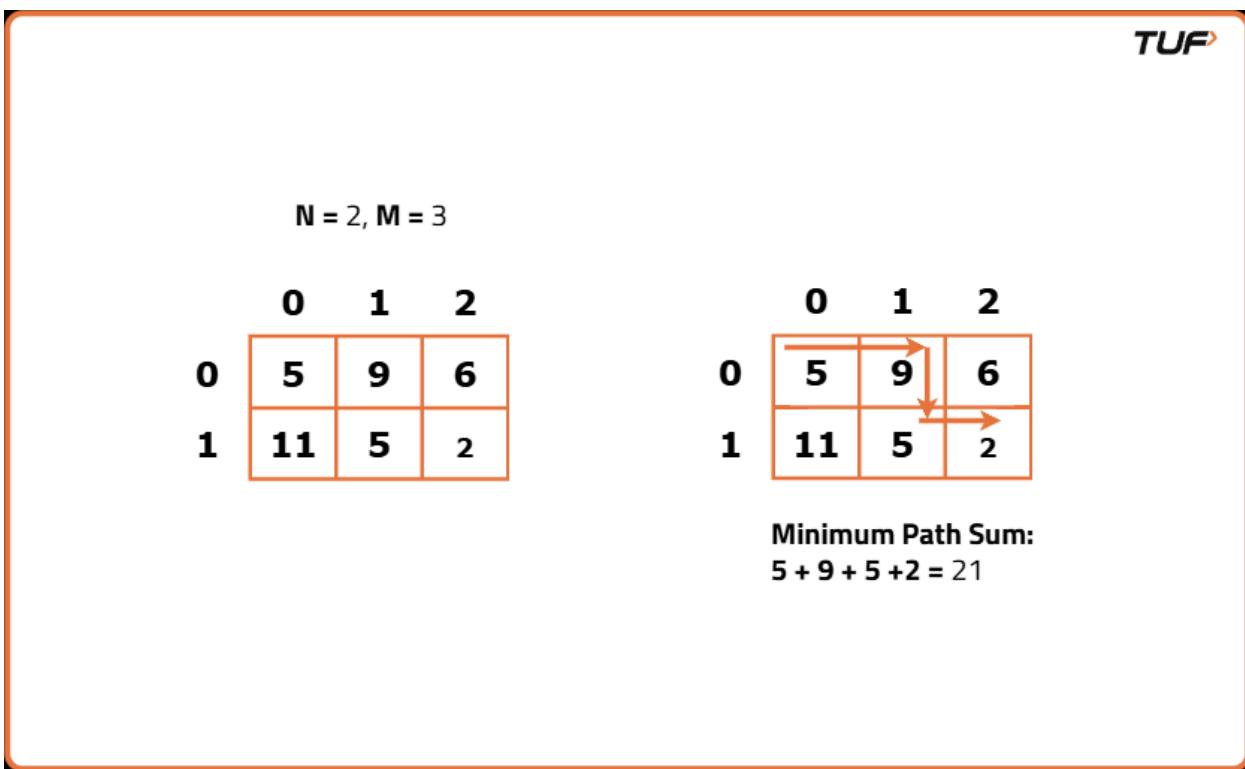
You are allowed to move **only right or down** at any point.

Your task is to find the **minimum possible sum** of numbers along any valid path.

Example explanation:

For  $\begin{bmatrix} 5, 9, 6 \\ 11, 5, 2 \end{bmatrix}$

The path  $5 \rightarrow 9 \rightarrow 5 \rightarrow 2$  gives sum 21, which is minimum.



## Approach 1: Memoization Approach

### Algorithm

This problem is similar to Grid Unique Paths, but instead of counting paths, we **minimize cost**.

Why greedy does not work:

Choosing the locally smallest value can lead to a path with a larger cost later. Hence, we must explore all valid paths using DP.

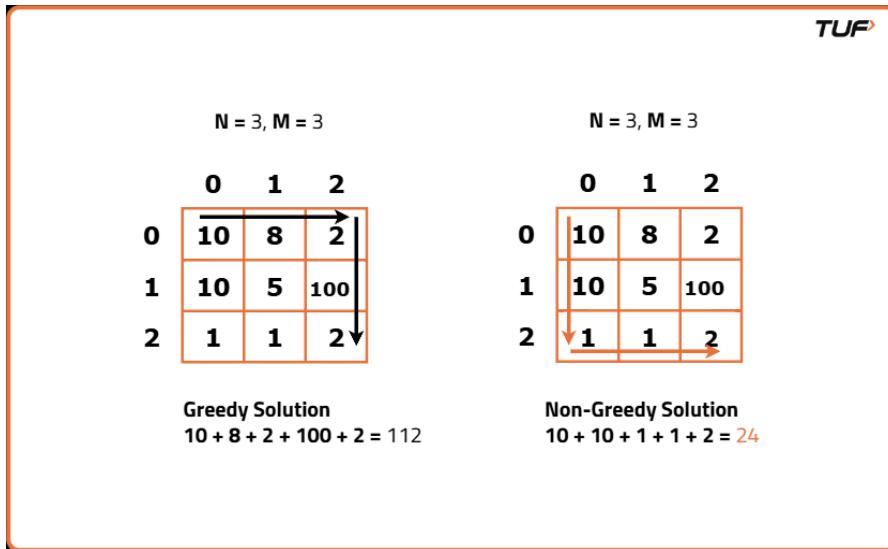
TUF

**N = 3, M = 3**

|   | 0  | 1 | 2   |
|---|----|---|-----|
| 0 | 10 | 8 | 2   |
| 1 | 10 | 5 | 100 |
| 2 | 1  | 1 | 2   |

Steps:

- Define a recursive function `minPath(i, j)` that returns the minimum path sum to reach cell  $(i, j)$ .
- Base cases:
  - If  $(i, j) == (0, 0)$ , return `grid[0][0]`.
  - If  $i < 0$  or  $j < 0$ , return a very large value (`1e9`) to reject invalid paths.
- If `dp[i][j]` is already computed, return it.
- Otherwise:
  - Cost from up = `grid[i][j] + minPath(i-1, j)`
  - Cost from left = `grid[i][j] + minPath(i, j-1)`
  - Store and return the minimum of the two.
- Final answer is `minPath(n-1, m-1)`.



## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int minPath(int i, int j, vector<vector<int>> &grid,
 vector<vector<int>> &dp) {
 if (i == 0 && j == 0) return grid[0][0];
 if (i < 0 || j < 0) return 1e9;
 if (dp[i][j] != -1) return dp[i][j];
 int up = grid[i][j] + minPath(i - 1, j, grid, dp);
 int left = grid[i][j] + minPath(i, j - 1, grid, dp);
 return dp[i][j] = min(up, left);
 }

 int minPathSum(vector<vector<int>> &grid) {
 int n = grid.size();
 int m = grid[0].size();
 vector<vector<int>> dp(n, vector<int>(m, -1));
 return minPath(n - 1, m - 1, grid, dp);
 }
};
```

```

int main() {
 vector<vector<int>> grid{{5,9,6},{11,5,2}};
 Solution obj;
 cout << obj.minPathSum(grid);
 return 0;
}

```

## Complexity Analysis

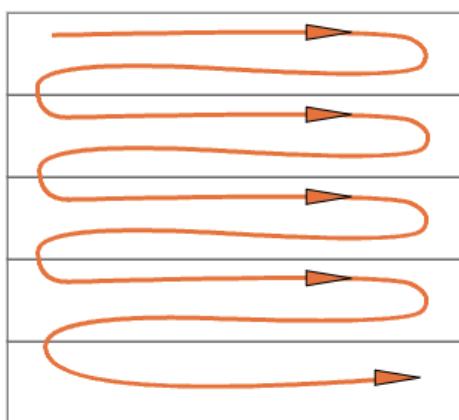
- Time Complexity:  $O(N \times M)$   
Each cell is computed once due to memoization.
  - Space Complexity:  $O(N \times M) + O(N + M)$   
DP table plus recursion stack.
- 

## Approach 2: Tabulation Approach

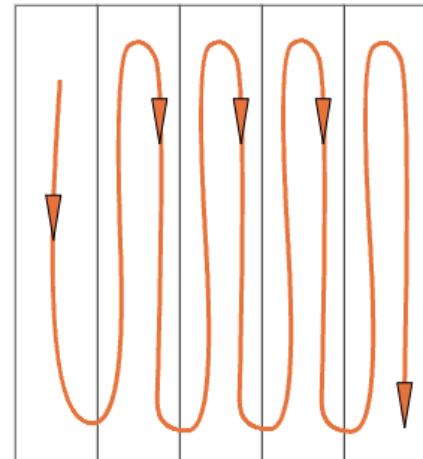
### Algorithm

This is a bottom-up DP solution.

- Create a  $dp[n][m]$  table.
- Base case:  $dp[0][0] = grid[0][0]$ .
- Traverse the grid row by row:
  - For each cell  $(i, j)$ :
    - Cost from up =  $dp[i-1][j]$  if  $i > 0$  else large value
    - Cost from left =  $dp[i][j-1]$  if  $j > 0$  else large value
    - $dp[i][j] = grid[i][j] + \min(\text{up}, \text{left})$
- Final answer is  $dp[n-1][m-1]$ .



from (0, 0) to (n-1, m-1)



from (0, 0) to (n-1, m-1)

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int minPathSum(vector<vector<int>> &matrix) {
 int n = matrix.size();
 int m = matrix[0].size();
 vector<vector<int>> dp(n, vector<int>(m, 0));

 for (int i = 0; i < n; i++) {
 for (int j = 0; j < m; j++) {
 if (i == 0 && j == 0)
 dp[i][j] = matrix[i][j];
 else {
 int up = matrix[i][j] + (i > 0 ? dp[i - 1][j] : 1e9);
 int left = matrix[i][j] + (j > 0 ? dp[i][j - 1] : 1e9);
 dp[i][j] = min(up, left);
 }
 }
 }
 return dp[n - 1][m - 1];
 }
};
```

```

 dp[i][j] = min(up, left);
 }
}
return dp[n - 1][m - 1];
}
};

int main() {
 vector<vector<int>> matrix{{5, 9, 6}, {11, 5, 2}};
 Solution obj;
 cout << obj.minPathSum(matrix);
 return 0;
}

```

## Complexity Analysis

- Time Complexity:  $O(N*M)$
  - Space Complexity:  $O(N*M)$
- 

## Approach 3: Space Optimization Approach

### Algorithm

From the recurrence:

$$dp[i][j] = grid[i][j] + \min(dp[i-1][j], dp[i][j-1])$$

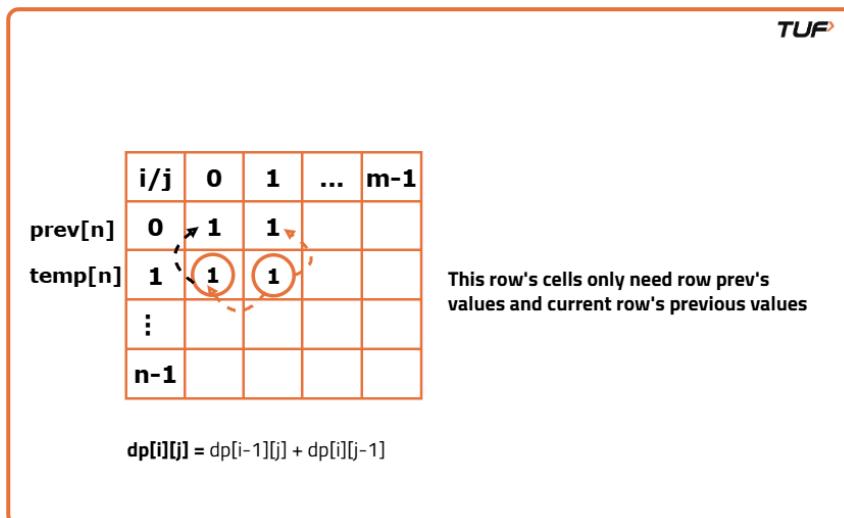
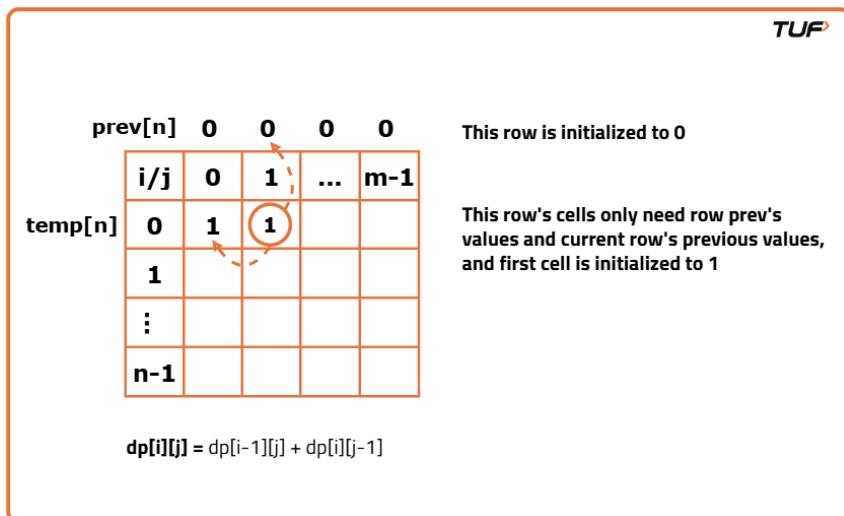
We only need:

- Previous row
- Current row

Steps:

- Use a 1D array prev for previous row.

- For each row, build a temp array for current row.
- After finishing a row, assign `prev = temp`.
- Final answer is `prev[m-1]`.



## Code

```
#include <bits/stdc++.h>
using namespace std;

class solution {
public:
 int minSumPath(int n, int m, vector<vector<int>> &matrix) {
```

```

vector<int> prev(m, 0);
for (int i = 0; i < n; i++) {
 vector<int> temp(m, 0);
 for (int j = 0; j < m; j++) {
 if (i == 0 && j == 0)
 temp[j] = matrix[i][j];
 else {
 int up = matrix[i][j] + (i > 0 ? prev[j] : 1e9);
 int left = matrix[i][j] + (j > 0 ? temp[j - 1] :
1e9);
 temp[j] = min(up, left);
 }
 }
 prev = temp;
}
return prev[m - 1];
};

int main() {
 vector<vector<int>> matrix{{5, 9, 6}, {11, 5, 2}};
 int n = matrix.size();
 int m = matrix[0].size();
 solution obj;
 cout << obj.minSumPath(n, m, matrix);
 return 0;
}

```

## Complexity Analysis

- Time Complexity:  $O(N \times M)$
- Space Complexity:  $O(M)$   
Only one row is stored at a time.

# Minimum Path Sum in Triangular Grid

## (DP 11)

You are given a triangular grid with  $n$  rows.

The first row has 1 element, the second row has 2 elements, and so on.

You start from the **top element (0,0)** and move to the **last row**.

From a cell  $(i, j)$ , you are allowed to move only to:

- the **bottom cell**  $(i+1, j)$
- the **bottom-right cell**  $(i+1, j+1)$

You need to return the **minimum possible path sum** from the top to any cell in the last row.

Example explanation:

For  $[[1], [1, 2], [1, 2, 4]]$

One minimum path is  $1 \rightarrow 1 \rightarrow 1$ , sum = 3.

---

### Approach 1: Memoization Approach

#### Algorithm

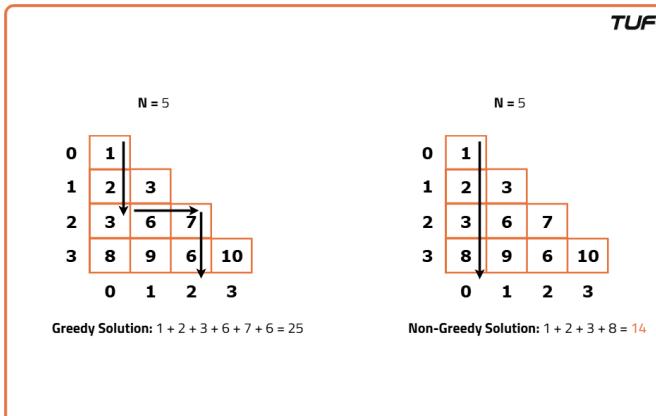
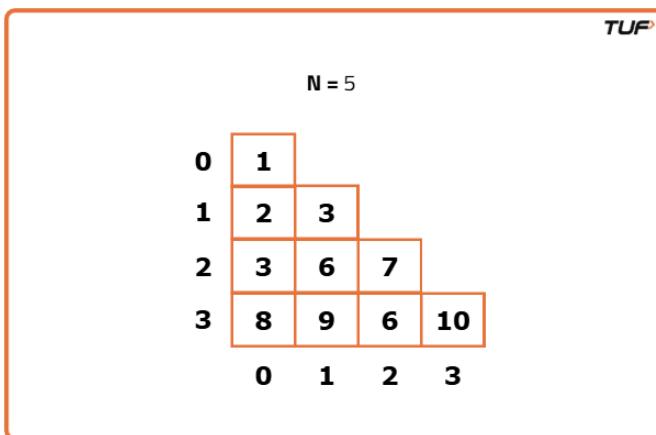
This problem is a variation of the minimum path sum in a grid, but the grid is triangular.

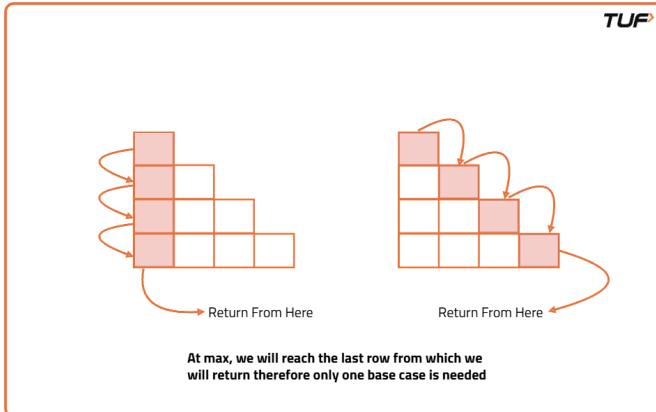
A greedy approach fails because choosing the locally smaller value may lead to a larger cost later.

We use recursion with memoization.

- Define a recursive function `solve(i, j)` that returns the minimum path sum starting from cell  $(i, j)$  to the bottom.
- Base case:
  - If  $i == n-1$  (last row), return `triangle[i][j]`.

- If the value is already stored in  $dp[i][j]$ , return it.
- Otherwise:
  - Move **down**:  $triangle[i][j] + solve(i+1, j)$
  - Move **diagonal**:  $triangle[i][j] + solve(i+1, j+1)$
  - Take the minimum of both.
- Store the result in  $dp$  to avoid recomputation.
- Final answer is  $solve(0, 0)$ .





## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int solve(int i, int j, vector<vector<int>> &triangle, int n,
 vector<vector<int>> &dp) {
 if (dp[i][j] != -1) return dp[i][j];
 if (i == n - 1) return triangle[i][j];
 int down = triangle[i][j] + solve(i + 1, j, triangle, n, dp);
 int diag = triangle[i][j] + solve(i + 1, j + 1, triangle, n,
 dp);
 return dp[i][j] = min(down, diag);
 }

 int minimumPathSum(vector<vector<int>> &triangle) {
 int n = triangle.size();
 vector<vector<int>> dp(n, vector<int>(n, -1));
 return solve(0, 0, triangle, n, dp);
 }
};

int main() {
 vector<vector<int>> triangle{{1}, {2, 3}, {3, 6, 7}, {8, 9, 6, 10}};
 Solution obj;
 cout << obj.minimumPathSum(triangle);
 return 0;
}
```

}

## Complexity Analysis

- Time Complexity:  $O(N^2)$   
Every cell in the triangle is computed once.
  - Space Complexity:  $O(N^2) + O(N)$   
DP table plus recursion stack depth.
- 

## Approach 2: Tabulation Approach

### Algorithm

This is a bottom-up DP approach.

- Create a dp table with the same shape as the triangle.
- Initialize the **last row** of dp with the values from the triangle, since there is no choice after that.
- Start from the second-last row and move upwards:
  - For each cell  $(i, j)$ :
    - $dp[i][j] = triangle[i][j] + \min(dp[i+1][j], dp[i+1][j+1])$
- The answer will be stored at  $dp[0][0]$ .

### Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int minimumPathSum(vector<vector<int>> &triangle, int n) {
 vector<vector<int>> dp(n, vector<int>(n, 0));
 dp[n-1] = triangle[n-1];
 for (int i = n - 2; i >= 0; i--) {
 for (int j = 0; j <= i; j++) {
 if (j == i)
 dp[i][j] = triangle[i][j] + dp[i+1][j];
 else
 dp[i][j] = triangle[i][j] + min(dp[i+1][j], dp[i+1][j+1]);
 }
 }
 return dp[0][0];
 }
};
```

```

 for (int j = 0; j < n; j++)
 dp[n - 1][j] = triangle[n - 1][j];

 for (int i = n - 2; i >= 0; i--) {
 for (int j = i; j >= 0; j--) {
 int down = triangle[i][j] + dp[i + 1][j];
 int diag = triangle[i][j] + dp[i + 1][j + 1];
 dp[i][j] = min(down, diag);
 }
 }
 return dp[0][0];
 }

int main() {
 vector<vector<int>> triangle{{1}, {2,3}, {3,6,7}, {8,9,6,10}};
 int n = triangle.size();
 Solution obj;
 cout << obj.minimumPathSum(triangle, n);
 return 0;
}

```

## Complexity Analysis

- Time Complexity:  $O(N^2)$   
Entire triangular grid is processed once.
  - Space Complexity:  $O(N^2)$   
DP table stores results for all cells.
- 

## Approach 3: Space Optimization Approach

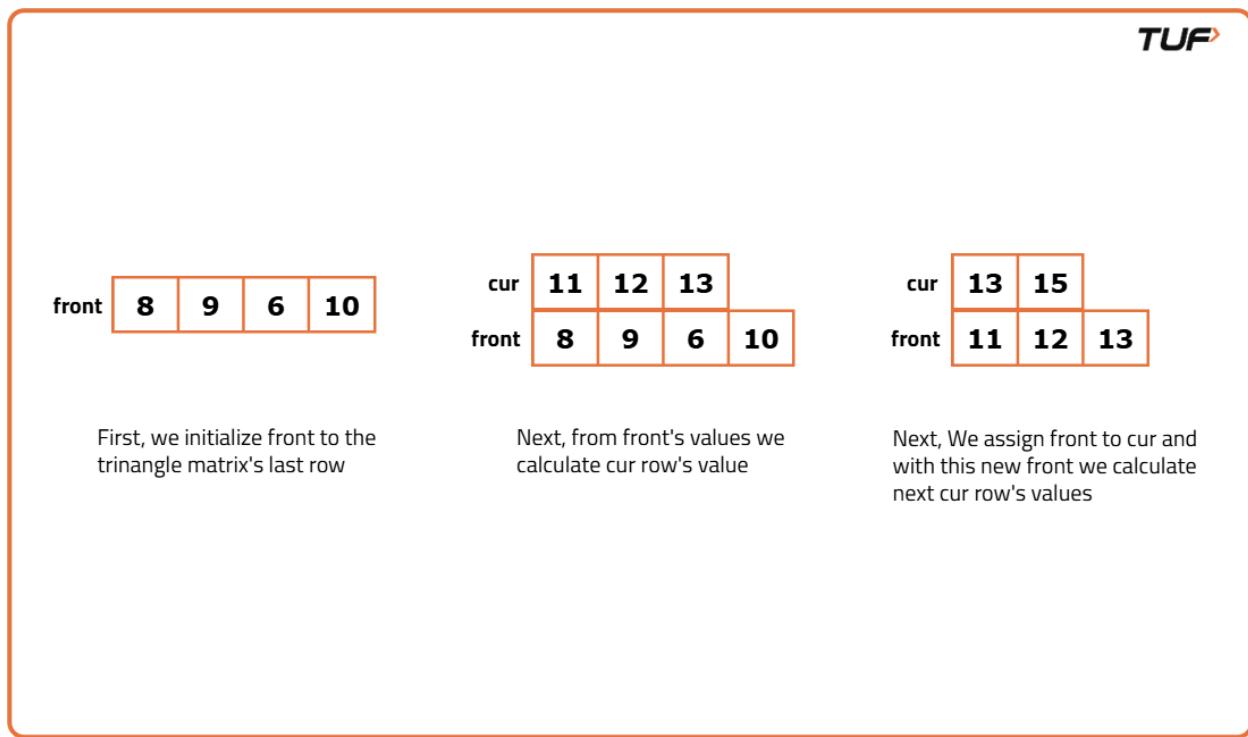
### Algorithm

From the recurrence relation:

$$dp[i][j] = triangle[i][j] + \min(dp[i+1][j], dp[i+1][j+1])$$

We only need the **next row** to compute the current row.

- Use a 1D array `front` initialized with the last row of the triangle.
- Use another array `cur` for the current row.
- Traverse from bottom to top:
  - For each cell, compute minimum using `front[j]` and `front[j+1]`.
- After each row, assign `front = cur`.
- Final answer is `front[0]`.



## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int minimumPathSum(vector<vector<int>> &triangle, int n) {
 vector<int> front(n, 0), cur(n, 0);
 for (int j = 0; j < n; j++)
 front[j] = triangle[n - 1][j];
 for (int i = n - 2; i >= 0; i--) {
 for (int j = 0; j < n; j++) {
 cur[j] = triangle[i][j] + min(front[j], front[j + 1]);
 }
 front = cur;
 }
 return front[0];
 }
};
```

```

 for (int i = n - 2; i >= 0; i--) {
 for (int j = i; j >= 0; j--) {
 int down = triangle[i][j] + front[j];
 int diag = triangle[i][j] + front[j + 1];
 cur[j] = min(down, diag);
 }
 front = cur;
 }
 return front[0];
 }

int main() {
 vector<vector<int>> triangle{{1}, {2,3}, {3,6,7}, {8,9,6,10}};
 int n = triangle.size();
 Solution obj;
 cout << obj.minimumPathSum(triangle, n);
 return 0;
}

```

## Complexity Analysis

- Time Complexity:  $O(N^2)$   
Every element in the triangle is processed once.
- Space Complexity:  $O(N)$   
Only one row is stored at any time.

# Minimum / Maximum Falling Path Sum (DP 12)

You are given a 2D grid matrix with integer values.

You can start from **any cell in the first row** and must reach **any cell in the last row**.

From a cell  $(i, j)$ , you are allowed to move to:

- bottom  $(i+1, j)$
- bottom-left  $(i+1, j-1)$
- bottom-right  $(i+1, j+1)$

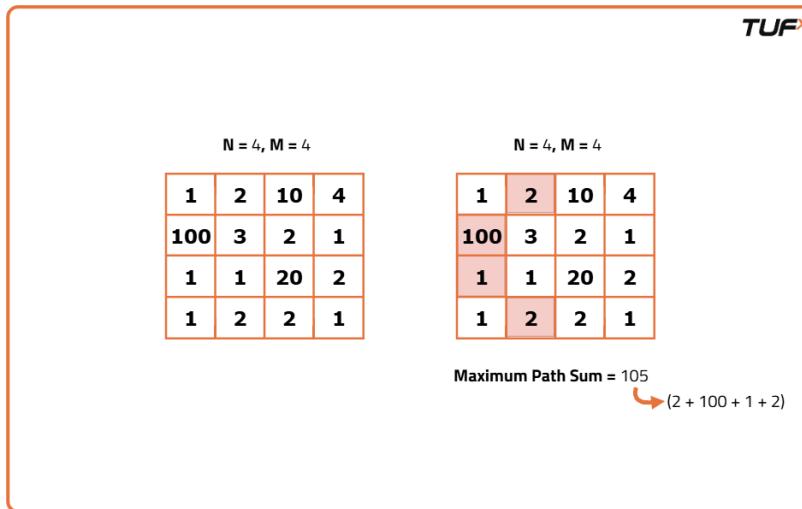
You need to return the **minimum falling path sum** (same logic applies for maximum by replacing min with max).

Example explanation:

For

$\begin{bmatrix} [1, 4, 3, 1], [2, 3, -1, -1], [1, 1, -1, 8] \end{bmatrix}$

One valid minimum path is  $4 \rightarrow -1 \rightarrow 8$ , sum = 11.



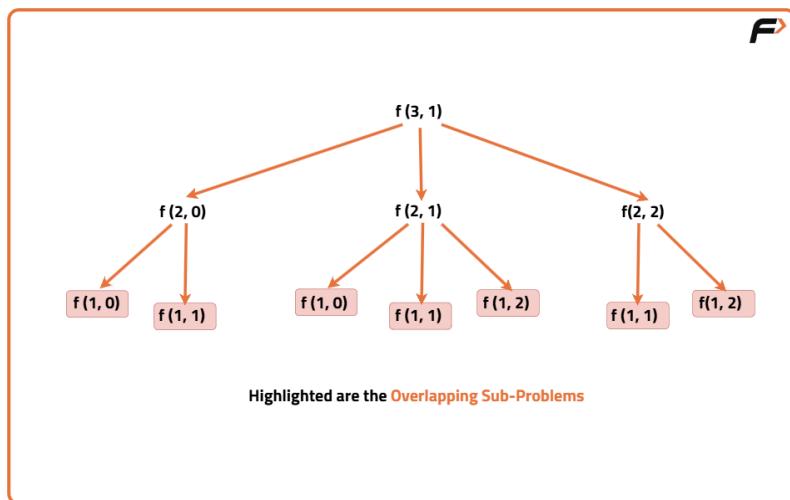
## Approach 1: Memoization

### Algorithm

A greedy approach does not work because choosing a locally smaller value may lead to a worse path later.

We use recursion with memoization.

- Define a recursive function  $\text{dfs}(\text{row}, \text{col})$  that returns the minimum falling path sum starting from  $(\text{row}, \text{col})$ .
- Base cases:
  - If  $\text{col} < 0$  or  $\text{col} \geq m$ , return a very large value ( $1e9$ ) to reject invalid paths.
  - If  $\text{row} == n-1$  (last row), return  $\text{matrix}[\text{row}][\text{col}]$ .
- If  $\text{dp}[\text{row}][\text{col}]$  is already computed, return it.
- Otherwise:
  - Try all three possible moves: down, down-left, down-right.
  - Add the current cell value to the minimum of these.
- Since the path can start from any column in the first row, compute the answer for all columns in row 0 and take the minimum.



## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int dfs(int row, int col, vector<vector<int>>& matrix,
vector<vector<int>>& dp) {
 int n = matrix.size();
 int m = matrix[0].size();
 if (col < 0 || col >= m) return 1e9;
 if (row == n - 1) return matrix[row][col];
 if (dp[row][col] != -1) return dp[row][col];
 int down = dfs(row + 1, col, matrix, dp);
 int downLeft = dfs(row + 1, col - 1, matrix, dp);
 int downRight = dfs(row + 1, col + 1, matrix, dp);
 return dp[row][col] =
 matrix[row][col] + min({down, downLeft, downRight});
 }

 int minFallingPathSum(vector<vector<int>>& matrix) {
 int n = matrix.size();
 int m = matrix[0].size();
 vector<vector<int>> dp(n, vector<int>(m, -1));
 int ans = 1e9;
 for (int col = 0; col < m; col++) {
 ans = min(ans, dfs(0, col, matrix, dp));
 }
 return ans;
 }
};

int main() {
 vector<vector<int>> matrix{
 {1,4,3,1},
 {2,3,-1,-1},
 {1,1,-1,8}
 };
}
```

```

 Solution obj;
 cout << obj.minFallingPathSum(matrix);
 return 0;
}

```

## Complexity Analysis

- Time Complexity:  $O(N \times M)$   
Each cell (row, col) is computed once.
  - Space Complexity:  $O(N \times M) + O(N)$   
DP table plus recursion stack depth.
- 

## Approach 2: Tabulation

### Algorithm

This is a bottom-up DP approach.

- Create a dp array of size  $n \times m$ .
- Initialize the **last row** of dp with the values from the matrix.
- Move from the second-last row upwards:
  - For each cell (row, col):
    - Take minimum of:
      - $dp[row+1][col]$
      - $dp[row+1][col-1]$  if valid
      - $dp[row+1][col+1]$  if valid
    - Add current cell value.
- After filling, the answer is the minimum value in  $dp[0]$ .

F

|     |   |    |   |
|-----|---|----|---|
| 1   | 2 | 10 | 4 |
| 100 | 3 | 2  | 1 |
| 1   | 1 | 20 | 2 |
| 1   | 2 | 2  | 1 |

↳  $dp[i][j] = \text{max sum to reach}(i, j)$

|     |     |     |    |
|-----|-----|-----|----|
| 1   | 2   | 10  | 4  |
| 102 | 13  | 12  | 11 |
| 103 | 103 | 33  | 14 |
| 104 | 105 | 105 | 34 |

Maximum path sum = 105

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int minFallingPathSum(vector<vector<int>>& matrix) {
 int n = matrix.size();
 int m = matrix[0].size();
 vector<vector<int>> dp(n, vector<int>(m, 0));

 for (int col = 0; col < m; col++)
 dp[n - 1][col] = matrix[n - 1][col];

 for (int row = n - 2; row >= 0; row--) {
 for (int col = 0; col < m; col++) {
 int down = dp[row + 1][col];
 int downLeft = (col > 0) ? dp[row + 1][col - 1] : 1e9;
 int downRight = (col < m - 1) ? dp[row + 1][col + 1] : 1e9;
 dp[row][col] =
 matrix[row][col] + min({down, downLeft,
 downRight});
 }
 }
 }
};
```

```

 }
 }

 return *min_element(dp[0].begin(), dp[0].end());
}
};

int main() {
 vector<vector<int>> matrix{
 {1, 4, 3, 1},
 {2, 3, -1, -1},
 {1, 1, -1, 8}
 };
 Solution obj;
 cout << obj.minFallingPathSum(matrix);
 return 0;
}

```

## Complexity Analysis

- Time Complexity:  $O(N*M)$   
Entire grid is processed once.
  - Space Complexity:  $O(N*M)$   
DP table stores results for all cells.
- 

## Approach 3: Space Optimization

### Algorithm

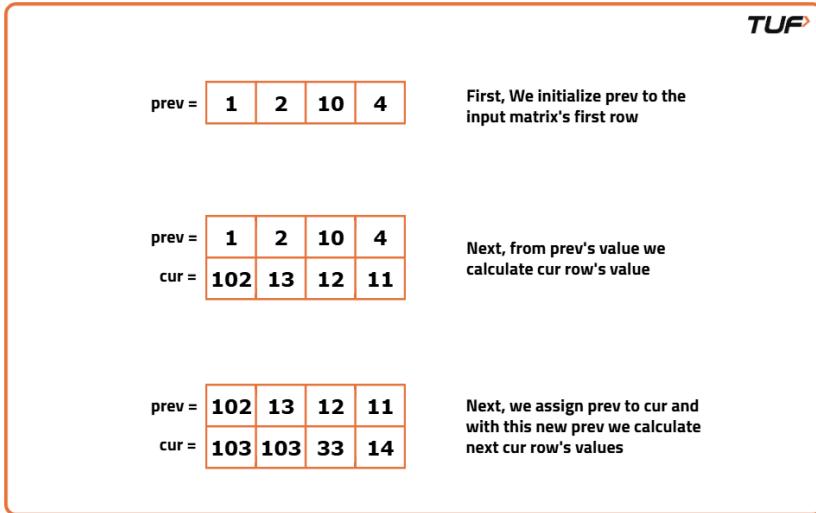
From the relation:

$$dp[\text{row}][\text{col}] = \text{matrix}[\text{row}][\text{col}] + \min(dp[\text{row}+1][\text{col}], \\ dp[\text{row}+1][\text{col}-1], dp[\text{row}+1][\text{col}+1])$$

We only need the **next row** to compute the current row.

- Use a 1D array dp initialized with the last row of the matrix.

- For each row from bottom to top:
  - Build a curr array using values from dp.
  - Replace dp with curr.
- Final answer is the minimum value in dp.



## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int minFallingPathSum(vector<vector<int>>& matrix) {
 int n = matrix.size();
 int m = matrix[0].size();
 vector<int> dp = matrix[n - 1];

 for (int row = n - 2; row >= 0; row--) {
 vector<int> curr(m, 0);
 for (int col = 0; col < m; col++) {
 int down = dp[col];
 int downLeft = (col > 0) ? dp[col - 1] : 1e9;
 int downRight = (col < m - 1) ? dp[col + 1] : 1e9;
 curr[col] =

```

```

 matrix[row][col] + min({down, downLeft,
downRight});
 }
 dp = curr;
}

return *min_element(dp.begin(), dp.end());
}
};

int main() {
vector<vector<int>> matrix{
{1,4,3,1},
{2,3,-1,-1},
{1,1,-1,8}
};
Solution obj;
cout << obj.minFallingPathSum(matrix);
return 0;
}

```

## Complexity Analysis

- Time Complexity:  $O(N \times M)$   
Each cell is processed once.
- Space Complexity:  $O(M)$   
Only one row is stored at a time.

# 3-D DP : Ninja and his Friends (DP-13)

We are given a grid of size  $N \times M$  where each cell contains some chocolates.

Alice starts at cell  $(0, 0)$  and Bob starts at cell  $(0, M-1)$ .

Both Alice and Bob move **row by row downward**. From a cell  $(i, j)$ , they can move to:

- bottom  $(i+1, j)$
- bottom-left  $(i+1, j-1)$
- bottom-right  $(i+1, j+1)$

When they visit a cell, they collect all chocolates from that cell.

If both of them land on the **same cell**, chocolates are counted **only once**.

They cannot move outside the grid.

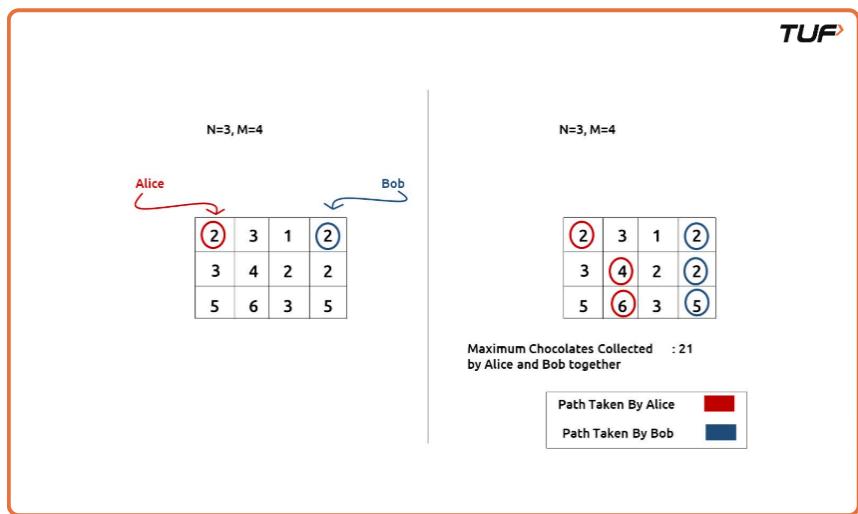
We need to return the **maximum total chocolates** Alice and Bob can collect together.

Example explanation:

For

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

One optimal path allows Alice and Bob to collect a total of 21 chocolates



## Approach 1: Memoization Approach

### Algorithm

This is a 3-dimensional DP problem because the state depends on:

- current row  $i$
- Alice's column  $j_1$
- Bob's column  $j_2$

Greedy does not work because taking the maximum chocolates at the current step may block better choices later.

We use recursion with memoization.

- Define  $\text{solve}(i, j_1, j_2)$  = maximum chocolates collectible starting from row  $i$ , where Alice is at column  $j_1$  and Bob at column  $j_2$ .
- If  $j_1$  or  $j_2$  goes out of bounds, return a very small value ( $-1e9$ ) to mark an invalid path.
- Base case:
  - If  $i == n-1$  (last row):
    - If  $j_1 == j_2$ , return  $\text{grid}[i][j_1]$
    - Else return  $\text{grid}[i][j_1] + \text{grid}[i][j_2]$
- If the value is already stored in  $\text{dp}$ , return it.
- Otherwise:
  - From  $(i, j_1, j_2)$ , try all  $3 \times 3 = 9$  combinations of moves for Alice and Bob.
  - For each combination, move to the next row and take the maximum result.
- Store and return the result.

The final answer is `solve(0, 0, M-1)`.

### Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int solve(int i, int j1, int j2, int n, int m,
 vector<vector<int>>& grid,
 vector<vector<vector<int>>& dp) {
 if (j1 < 0 || j1 >= m || j2 < 0 || j2 >= m)
 return -1e9;
 if (i == n - 1) {
 if (j1 == j2) return grid[i][j1];
 return grid[i][j1] + grid[i][j2];
 }
 if (dp[i][j1][j2] != -1) return dp[i][j1][j2];
 int maxi = -1e9;
 int curr = (j1 == j2) ? grid[i][j1]
 : grid[i][j1] + grid[i][j2];
 for (int dj1 = -1; dj1 <= 1; dj1++) {
 for (int dj2 = -1; dj2 <= 1; dj2++) {
 maxi = max(maxi,
 curr + solve(i + 1, j1 + dj1, j2 + dj2,
 n, m, grid, dp));
 }
 }
 return dp[i][j1][j2] = maxi;
 }

 int maximumChocolates(int n, int m, vector<vector<int>>& grid) {
 vector<vector<vector<int>>> dp(
 n, vector<vector<int>>(m, vector<int>(m, -1)));
 return solve(0, 0, m - 1, n, m, grid, dp);
 }
};

int main() {
```

```

vector<vector<int>> grid = {
 {2, 3, 1, 2},
 {3, 4, 2, 2},
 {5, 6, 3, 5}
};

Solution obj;
cout << obj.maximumChocolates(grid.size(), grid[0].size(), grid);
return 0;
}

```

## Complexity Analysis

- Time Complexity:  $O(N * M * M * 9)$   
There are  $N * M * M$  states, and for each we try 9 move combinations.
  - Space Complexity:  $O(N * M * M) + O(N)$   
DP table plus recursion stack.
- 

## Approach 2: Tabulation Approach

### Algorithm

This is the bottom-up version of the memoized solution.

- Create a 3D DP table  $dp[i][j1][j2]$ .
- Base case:
  - Fill the last row ( $i = n-1$ ) directly using grid values.
- Move from row  $n-2$  up to  $0$ :
  - For each  $(j1, j2)$  pair, try all 9 move combinations.
  - Take the maximum possible chocolates.
- The final answer is stored at  $dp[0][0][m-1]$ .

## Understanding the 3D DP Grid

| dp[1][1][2] |   |   |   |   |
|-------------|---|---|---|---|
|             | 0 | 1 | 2 | 3 |
| 0           | 2 | 3 | 1 | 2 |
| 1           | 3 | 4 | 2 | 2 |
| 2           | 5 | 6 | 3 | 5 |

dp[1][1][2] gives the Maximum chocolates collected when Alice is at (1,1) and Bob is at (1,2).

| dp[2][1][2] |   |   |   |   |
|-------------|---|---|---|---|
|             | 0 | 1 | 2 | 3 |
| 0           | 2 | 3 | 1 | 2 |
| 1           | 3 | 4 | 2 | 2 |
| 2           | 5 | 6 | 3 | 5 |

dp[2][1][2] gives the Maximum chocolates collected when Alice is at (2,1) and Bob is at (2,2).

dp[2][1][1]

|   | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 2 | 3 | 1 | 2 |
| 1 | 3 | 4 | 2 | 2 |
| 2 | 5 | 6 | 3 | 5 |

dp[2][1][1] gives the Maximum chocolates collected when Alice is at (2,1) and Bob is at (2,1).

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int maximumChocolates(int n, int m, vector<vector<int>>& grid) {
 vector<vector<vector<int>>> dp(
 n, vector<vector<int>>(m, vector<int>(m, 0)));

 for (int j1 = 0; j1 < m; j1++) {
 for (int j2 = 0; j2 < m; j2++) {
 if (j1 == j2) dp[n-1][j1][j2] = grid[n-1][j1];
 else dp[n-1][j1][j2] = grid[n-1][j1] + grid[n-1][j2];
 }
 }

 for (int i = n - 2; i >= 0; i--) {
 for (int j1 = 0; j1 < m; j1++) {
 for (int j2 = 0; j2 < m; j2++) {
 int maxi = -1e9;
 int curr = (j1 == j2) ? grid[i][j1]
 : grid[i][j1] + grid[i][j2];
 for (int dj1 = -1; dj1 <= 1; dj1++) {
 for (int dj2 = -1; dj2 <= 1; dj2++) {
 if (j1 + dj1 < 0 || j1 + dj1 >= m || j2 + dj2 < 0 || j2 + dj2 >= m)
 continue;
 maxi = max(maxi, dp[i+1][j1+dj1][j2+dj2]);
 }
 }
 dp[i][j1][j2] = maxi;
 }
 }
 }
 }
}
```

```

 int nj1 = j1 + dj1;
 int nj2 = j2 + dj2;
 if (nj1 >= 0 && nj1 < m &&
 nj2 >= 0 && nj2 < m) {
 maxi = max(maxi,
 curr + dp[i+1][nj1][nj2]);
 }
 }
 dp[i][j1][j2] = maxi;
}
}
return dp[0][0][m-1];
}

int main() {
 vector<vector<int>> grid = {
 {2, 3, 1, 2},
 {3, 4, 2, 2},
 {5, 6, 3, 5}
 };
 Solution obj;
 cout << obj.maximumChocolates(grid.size(), grid[0].size(), grid);
 return 0;
}

```

## Complexity Analysis

- Time Complexity:  $O(N * M * M * 9)$
- Space Complexity:  $O(N * M * M)$

## Approach 3: Space Optimization Approach

## Algorithm

In tabulation, each row depends only on the **next row**.

- Instead of a full 3D DP table, use:
  - $\text{next}[m][m]$  → stores values for row  $i+1$
  - $\text{curr}[m][m]$  → stores values for row  $i$
- Initialize next using the last row of the grid.
- Move upward row by row, filling curr from next.
- After each row, assign  $\text{next} = \text{curr}$ .
- Final answer is  $\text{next}[0][m-1]$ .

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int maximumChocolates(int n, int m, vector<vector<int>>& grid) {
 vector<vector<int>> next(m, vector<int>(m, 0));
 vector<vector<int>> curr(m, vector<int>(m, 0));

 for (int j1 = 0; j1 < m; j1++) {
 for (int j2 = 0; j2 < m; j2++) {
 if (j1 == j2) next[j1][j2] = grid[n-1][j1];
 else next[j1][j2] = grid[n-1][j1] + grid[n-1][j2];
 }
 }

 for (int i = n - 2; i >= 0; i--) {
 for (int j1 = 0; j1 < m; j1++) {
 for (int j2 = 0; j2 < m; j2++) {
 int maxi = -1e9;
```

```

 int currVal = (j1 == j2) ? grid[i][j1]
 : grid[i][j1] +
grid[i][j2];
 for (int dj1 = -1; dj1 <= 1; dj1++) {
 for (int dj2 = -1; dj2 <= 1; dj2++) {
 int nj1 = j1 + dj1;
 int nj2 = j2 + dj2;
 if (nj1 >= 0 && nj1 < m &&
 nj2 >= 0 && nj2 < m) {
 maxi = max(maxi,
 currVal + next[nj1][nj2]);
 }
 }
 }
 curr[j1][j2] = maxi;
 }
}
next = curr;
}
return next[0][m-1];
}
};

int main() {
 vector<vector<int>> grid = {
 {2, 3, 1, 2},
 {3, 4, 2, 2},
 {5, 6, 3, 5}
 };
 Solution obj;
 cout << obj.maximumChocolates(grid.size(), grid[0].size(), grid);
 return 0;
}

```

## Complexity Analysis

- Time Complexity:  $O(N * M * M * 9)$

- Space Complexity:  $O(M * M)$

# Subset Sum Equal to Target (DP-14)

We are given an array ARR of N positive integers and an integer K.

We need to check whether there exists **any subset (subsequence)** of the array whose sum is exactly equal to K.

A subset can be contiguous or non-contiguous, but the order of elements must remain the same as in the original array.

Example explanation:

For ARR = [4, 3, 5, 2] and K = 6

Subset [4, 2] gives sum 6, so the answer is true.

---

## Approach 1: Memoization Approach

### Algorithm

Greedy does not work here because we are not optimizing step by step; we just need to check existence.

We use recursion with memoization using the **pick / not pick** technique.

Define a function  $f(ind, target)$  that returns true if a subset with sum target can be formed using elements from index 0 to ind.

Base cases:

- If  $target == 0$ , we already formed the required sum, return true.
- If  $ind == 0$ , return true if  $arr[0] == target$ , else false.

At each index, we have two choices:

- **Not pick** the current element →  $f(ind-1, \ target)$
- **Pick** the current element (only if  $arr[ind] \leq target$ ) →  $f(ind-1, \ target - arr[ind])$

If either choice returns true, the answer is true.

To avoid recomputation, store results in a DP table  $dp[ind][target]$ .

### Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 bool subsetSumUtil(int ind, int target, vector<int>& arr,
vector<vector<int>>& dp) {
 if (target == 0) return true;
 if (ind == 0) return arr[0] == target;
 if (dp[ind][target] != -1) return dp[ind][target];

 bool notTaken = subsetSumUtil(ind - 1, target, arr, dp);
 bool taken = false;
 if (arr[ind] <= target)
 taken = subsetSumUtil(ind - 1, target - arr[ind], arr,
dp);

 return dp[ind][target] = notTaken || taken;
 }

 bool subsetSumToK(int n, int k, vector<int>& arr) {
 vector<vector<int>> dp(n, vector<int>(k + 1, -1));
 return subsetSumUtil(n - 1, k, arr, dp);
 }
};

int main() {
 vector<int> arr = {1, 2, 3, 4};
```

```

int k = 4;
Solution sol;
cout << sol.subsetSumToK(arr.size(), k, arr);
return 0;
}

```

## Complexity Analysis

- Time Complexity:  $O(N \cdot K)$   
There are  $N \cdot K$  states, each solved once.
  - Space Complexity:  $O(N \cdot K) + O(N)$   
DP table plus recursion stack.
- 

## Approach 2: Tabulation Approach

### Algorithm

We convert the recursive solution into bottom-up DP.

Let  $dp[i][t]$  be true if a subset with sum  $t$  can be formed using elements from index 0 to  $i$ .

Initialization:

- For all  $i$ ,  $dp[i][0] = \text{true}$  (sum 0 is always possible).
- If  $arr[0] \leq K$ , set  $dp[0][arr[0]] = \text{true}$ .

Transition:

- $\text{notTaken} = dp[i-1][t]$
- $\text{taken} = dp[i-1][t - arr[i]]$  if  $arr[i] \leq t$
- $dp[i][t] = \text{notTaken} \text{ || taken}$

Final answer is  $dp[n-1][K]$ .

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 bool subsetSumToK(int n, int k, vector<int>& arr) {
 vector<vector<bool>> dp(n, vector<bool>(k + 1, false));

 for (int i = 0; i < n; i++)
 dp[i][0] = true;

 if (arr[0] <= k)
 dp[0][arr[0]] = true;

 for (int i = 1; i < n; i++) {
 for (int target = 1; target <= k; target++) {
 bool notTaken = dp[i - 1][target];
 bool taken = false;
 if (arr[i] <= target)
 taken = dp[i - 1][target - arr[i]];
 dp[i][target] = notTaken || taken;
 }
 }
 return dp[n - 1][k];
 }
};

int main() {
 vector<int> arr = {1, 2, 3, 4};
 int k = 4;
 Solution sol;
 cout << sol.subsetSumToK(arr.size(), k, arr);
 return 0;
}
```

## Complexity Analysis

- Time Complexity:  $O(N*K)$   
Two nested loops.
  - Space Complexity:  $O(N*K)$   
2D DP table, no recursion stack.
- 

## Approach 3: Space Optimization Approach

### Algorithm

From the relation:

$$dp[i][t] = dp[i-1][t] \cup dp[i-1][t - arr[i]]$$

We only need the **previous row**, so we use a 1D array.

Steps:

- Use a boolean array `prev` of size  $K+1$ .
- Initialize `prev[0] = true`.
- If  $arr[0] \leq K$ , set `prev[arr[0]] = true`.
- For each new element, create `cur` array:
  - `cur[0] = true`
  - Update `cur[target]` using values from `prev`.
- Move `cur` into `prev`.

Final answer is `prev[K]`.



Eg: [4, 2, 3, 7, ..., 23]  
Target = 11

arr[0] = 4

target →

d[0][4]

| target IND | 0    | 1     | 2     | 3     | 4     | ...   | k     |
|------------|------|-------|-------|-------|-------|-------|-------|
| 0          | true | false | false | false | true  | false | false |
| 1          | true | false | false | false | false | false | false |
| ⋮          | true | false | false | false | false | false | false |
| N - 1      | true | false | false | false | false | false | false |

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 bool subsetSumToK(int n, int k, vector<int>& arr) {
 vector<bool> prev(k + 1, false);
 prev[0] = true;

 if (arr[0] <= k)
 prev[arr[0]] = true;

 for (int i = 1; i < n; i++) {
 vector<bool> cur(k + 1, false);
 cur[0] = true;
 for (int target = 1; target <= k; target++) {
 bool notTaken = prev[target];
 bool taken = false;
 if (arr[i] <= target)
 taken = prev[target - arr[i]];
 cur[target] = notTaken || taken;
 }
 }
 return prev[k];
 }
}
```

```

 prev = cur;
 }
 return prev[k];
}
};

int main() {
 vector<int> arr = {1, 2, 3, 4};
 int k = 4;
 Solution sol;
 cout << sol.subsetSumToK(arr.size(), k, arr);
 return 0;
}

```

### Complexity Analysis

- Time Complexity:  $O(N*K)$   
Each element processes all target values.
- Space Complexity:  $O(K)$   
Only one DP row is stored.

## Partition Equal Subset Sum (DP-15)

We are given an array `arr` of  $n$  integers.

We need to check whether the array can be divided into **two subsets** such that the **sum of elements in both subsets is equal**.

If such a partition exists, return `true`, otherwise return `false`.

Explanation idea:

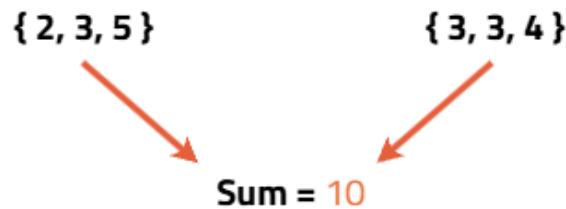
If the total sum of the array is  $S$ , then each subset must have sum  $S/2$ .

So the problem reduces to checking whether there exists a **subset with sum =  $S/2$** .

arr = 

|   |   |   |   |   |   |
|---|---|---|---|---|---|
| 2 | 3 | 3 | 3 | 4 | 5 |
|---|---|---|---|---|---|

We can partition the array in two subsequences



## Memoization Approach

### Algorithm

This problem is a direct modification of **Subset Sum Equal to Target**.

1. Calculate the total sum of the array.
2. If the total sum is **odd**, it is impossible to split it into two equal subsets, so return false.
3. Otherwise, set **target** = **totalSum / 2**.
4. Now check whether there exists a subset with sum equal to **target**.

Define a recursive function **f(ind, target)**:

- It returns true if a subset with sum **target** can be formed using elements from index 0 to **ind**.

Base cases:

- If  $\text{target} == 0$ , return true (subset found).
- If  $\text{ind} == 0$ , return true if  $\text{arr}[0] == \text{target}$ , else false.

Recursive choices:

- **Not take** the current element  $\rightarrow f(\text{ind}-1, \text{target})$
- **Take** the current element (only if  $\text{arr}[\text{ind}] \leq \text{target}$ )  $\rightarrow f(\text{ind}-1, \text{target}-\text{arr}[\text{ind}])$

Return true if any of the above choices gives true.

To avoid recomputation, store results in a DP table  $dp[\text{ind}][\text{target}]$ .

### Code

```
#include <bits/stdc++.h>
using namespace std;

bool subsetSumUtil(int ind, int target, vector<int>& arr,
vector<vector<int>>& dp) {
 if (target == 0) return true;
 if (ind == 0) return arr[0] == target;
 if (dp[ind][target] != -1) return dp[ind][target];

 bool notTaken = subsetSumUtil(ind - 1, target, arr, dp);
 bool taken = false;
 if (arr[ind] <= target)
 taken = subsetSumUtil(ind - 1, target - arr[ind], arr, dp);

 return dp[ind][target] = notTaken || taken;
}

bool canPartition(int n, vector<int>& arr) {
 int totalSum = 0;
 for (int i = 0; i < n; i++) totalSum += arr[i];
 if (totalSum % 2 != 0) return false;
```

```

 int target = totalSum / 2;
 vector<vector<int>> dp(n, vector<int>(target + 1, -1));
 return subsetSumUtil(n - 1, target, arr, dp);
 }

int main() {
 vector<int> arr = {2, 3, 3, 3, 4, 5};
 int n = arr.size();

 if (canPartition(n, arr))
 cout << "The Array can be partitioned into two equal subsets";
 else
 cout << "The Array cannot be partitioned into two equal
subsets";

 return 0;
}

```

## Complexity Analysis

- Time Complexity:  $O(N*K)$   
There are  $N*K$  states ( $N$  = number of elements,  $K$  = target sum).
  - Space Complexity:  $O(N*K) + O(N)$   
DP table plus recursion stack.
- 

## Tabulation Approach

### Algorithm

This is the bottom-up version of the memoization solution.

1. Compute total sum. If it is odd, return false.
2. Set  $target = totalSum / 2$ .

3. Create a DP table  $dp[n][target+1]$  where  
 $dp[i][t]$  is true if a subset with sum  $t$  can be formed using elements from 0 to  $i$ .

Initialization:

- For all  $i$ , set  $dp[i][0] = \text{true}$  (sum 0 is always possible).
- If  $\text{arr}[0] \leq \text{target}$ , set  $dp[0][\text{arr}[0]] = \text{true}$ .

Transition:

- $\text{notTaken} = dp[i-1][t]$
- $\text{taken} = dp[i-1][t-\text{arr}[i]]$  if  $\text{arr}[i] \leq t$
- $dp[i][t] = \text{notTaken} \text{ || taken}$

Final answer is  $dp[n-1][\text{target}]$ .

### Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 bool canPartition(int n, vector<int>& arr) {
 int totalSum = 0;
 for (int x : arr) totalSum += x;
 if (totalSum % 2 != 0) return false;

 int target = totalSum / 2;
 vector<vector<bool>> dp(n, vector<bool>(target + 1, false));

 for (int i = 0; i < n; i++)
 dp[i][0] = true;

 if (arr[0] <= target)
 dp[0][arr[0]] = true;
```

```

 for (int i = 1; i < n; i++) {
 for (int t = 1; t <= target; t++) {
 bool notTaken = dp[i - 1][t];
 bool taken = false;
 if (arr[i] <= t)
 taken = dp[i - 1][t - arr[i]];
 dp[i][t] = notTaken || taken;
 }
 }
 return dp[n - 1][target];
 }
};

int main() {
 vector<int> arr = {2, 3, 3, 3, 4, 5};
 Solution obj;
 cout << obj.canPartition(arr.size(), arr);
 return 0;
}

```

## Complexity Analysis

- Time Complexity:  $O(N*K)$
- Space Complexity:  $O(N*K)$

## Space Optimization Approach

### Algorithm

From the relation

$$dp[i][t] = dp[i-1][t] \ || \ dp[i-1][t-arr[i]]$$

We only need the **previous row**, so we can optimize space to 1D.

Steps:

1. Compute total sum, return false if odd.
2. Set target = totalSum / 2.
3. Use a 1D array prev[target+1].
4. Initialize:
  - o prev[0] = true
  - o If arr[0] <= target, set prev[arr[0]] = true
5. For each new element, build a new cur array using prev.
6. Move cur into prev.

Final answer is prev[target].

### Code

```
#include <bits/stdc++.h>
using namespace std;

bool canPartition(int n, vector<int>& arr) {
 int totalSum = 0;
 for (int x : arr) totalSum += x;
 if (totalSum % 2 != 0) return false;

 int target = totalSum / 2;
 vector<bool> prev(target + 1, false);
 prev[0] = true;

 if (arr[0] <= target)
 prev[arr[0]] = true;

 for (int i = 1; i < n; i++) {
 vector<bool> cur(target + 1, false);
 cur[0] = true;
 for (int t = 1; t <= target; t++) {
 bool notTaken = prev[t];
 bool taken = prev[t - arr[i]];
 cur[t] = notTaken || taken;
 }
 prev = cur;
 }
}
```

```

 bool taken = false;
 if (arr[i] <= t)
 taken = prev[t - arr[i]];
 cur[t] = notTaken || taken;
 }
 prev = cur;
}
return prev[target];
}

int main() {
 vector<int> arr = {2, 3, 3, 3, 4, 5};
 int n = arr.size();

 if (canPartition(n, arr))
 cout << "The Array can be partitioned into two equal subsets";
 else
 cout << "The Array cannot be partitioned into two equal
subsets";

 return 0;
}

```

## Complexity Analysis

- Time Complexity:  $O(N*K)$
- Space Complexity:  $O(K)$

# Partition Set Into 2 Subsets With Minimum Absolute Sum Difference (DP-16)

We are given an array `arr` of  $n$  integers.

We need to partition the array into two subsets such that the **absolute difference between their sums is minimum**.

Key idea:

If the total sum of the array is  $S$  and one subset has sum  $s_1$ , then the other subset has sum  $S - s_1$ .

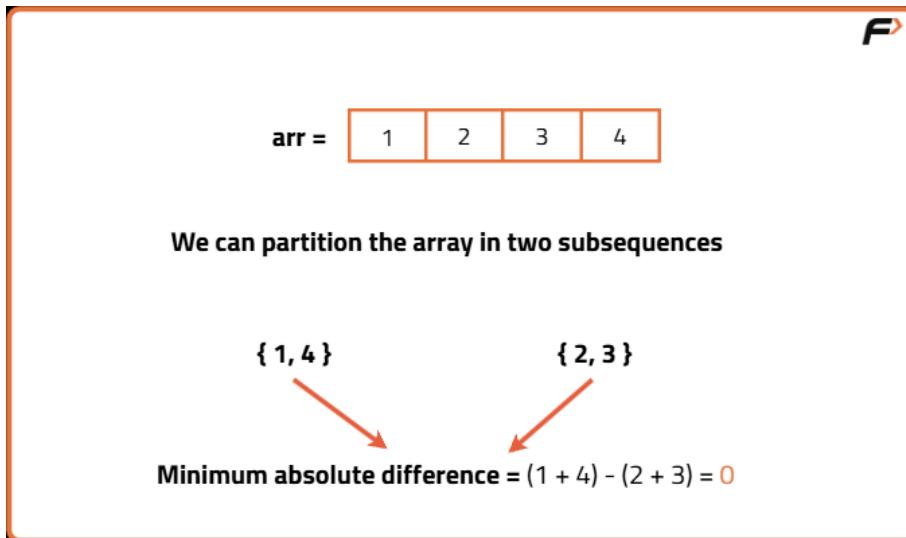
The absolute difference will be  $| (S - s_1) - s_1 | = |S - 2s_1|$ .

So, we only need to find all possible subset sums  $s_1$  and choose the one that minimizes this difference.

Example explanation:

For `arr` = [1, 2, 3, 4], total sum = 10.

Possible subset sum  $s_1$  = 5 gives difference  $|10 - 2*5| = 0$ , which is minimum.



## Approach 1: Memoization

This problem is a variation of **Subset Sum Equal to Target**.

- First calculate the total sum totSum.
- A subset sum can range from 0 to totSum.
- Use a recursive function  $f(ind, target)$  that returns true if a subset with sum target can be formed using elements from index 0 to ind.

Base cases:

- If  $target == 0$ , return true.
- If  $ind == 0$ , return true if  $arr[0] == target$ , else false.

Choices:

- Do not take the current element  $\rightarrow f(ind-1, target)$
- Take the current element (only if  $arr[ind] \leq target$ )  $\rightarrow f(ind-1, target - arr[ind])$

Store results in a DP table to avoid recomputation.

After filling the DP table, iterate over all possible subset sums s1 that are achievable and compute  
 $abs(totSum - 2*s1)$ .  
Return the minimum value.

### Code

```
#include <bits/stdc++.h>
using namespace std;

bool subsetSumUtil(int ind, int target, vector<int>& arr,
vector<vector<int>>& dp) {
 if (target == 0) return true;
 if (ind == 0) return arr[0] == target;
 if (dp[ind][target] != -1) return dp[ind][target];

 bool notTaken = subsetSumUtil(ind - 1, target, arr, dp);
 bool taken = false;
```

```

 if (arr[ind] <= target)
 taken = subsetSumUtil(ind - 1, target - arr[ind], arr, dp);

 return dp[ind][target] = notTaken || taken;
 }

int minSubsetSumDifference(vector<int>& arr, int n) {
 int totSum = 0;
 for (int x : arr) totSum += x;

 vector<vector<int>> dp(n, vector<int>(totSum + 1, -1));

 for (int s = 0; s <= totSum; s++) {
 subsetSumUtil(n - 1, s, arr, dp);
 }

 int mini = INT_MAX;
 for (int s = 0; s <= totSum; s++) {
 if (dp[n - 1][s]) {
 int diff = abs(totSum - 2 * s);
 mini = min(mini, diff);
 }
 }
 return mini;
}

int main() {
 vector<int> arr = {1, 2, 3, 4};
 cout << minSubsetSumDifference(arr, arr.size());
 return 0;
}

```

## Complexity Analysis

- Time Complexity:  $O(N*K)$   
 $N$  is number of elements,  $K$  is total sum.

- Space Complexity:  $O(N*K) + O(N)$   
DP table plus recursion stack.
- 

## Approach 2: Tabulation

### Algorithm

Convert the memoization solution into bottom-up DP.

- Create a DP table  $dp[n][totSum+1]$  where  
 $dp[i][s] = \text{true}$  if a subset with sum  $s$  can be formed using elements  $0..i$ .
- Initialization:
  - For all  $i$ ,  $dp[i][0] = \text{true}$ .
  - If  $arr[0] \leq totSum$ , set  $dp[0][arr[0]] = \text{true}$ .
- Fill the table using:
  - $\text{notTaken} = dp[i-1][s]$
  - $\text{taken} = dp[i-1][s-arr[i]]$  if  $arr[i] \leq s$
- After filling, check all  $s$  where  $dp[n-1][s]$  is true and compute the minimum difference.

### Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int minSubsetSumDifference(vector<int>& arr, int n) {
 int totSum = 0;
 for (int x : arr) totSum += x;

 vector<vector<bool>> dp(n, vector<bool>(totSum + 1, false));
```

```

 for (int i = 0; i < n; i++)
 dp[i][0] = true;

 if (arr[0] <= totSum)
 dp[0][arr[0]] = true;

 for (int i = 1; i < n; i++) {
 for (int s = 1; s <= totSum; s++) {
 bool notTaken = dp[i - 1][s];
 bool taken = false;
 if (arr[i] <= s)
 taken = dp[i - 1][s - arr[i]];
 dp[i][s] = notTaken || taken;
 }
 }

 int mini = INT_MAX;
 for (int s = 0; s <= totSum; s++) {
 if (dp[n - 1][s]) {
 int diff = abs(totSum - 2 * s);
 mini = min(mini, diff);
 }
 }
 return mini;
 }
};

int main() {
 vector<int> arr = {1, 2, 3, 4};
 Solution sol;
 cout << sol.minSubsetSumDifference(arr, arr.size());
 return 0;
}

```

## Complexity Analysis

- Time Complexity:  $O(N*K)$

- Space Complexity:  $O(N*K)$
- 

## Approach 3: Space Optimization

### Algorithm

From the relation

$dp[i][s] = dp[i-1][s] \text{ || } dp[i-1][s-arr[i]]$ ,  
we only need the previous row.

- Use a 1D boolean array prev.
- Initialize  $\text{prev}[0] = \text{true}$ .
- If  $\text{arr}[0] \leq \text{totSum}$ , set  $\text{prev}[\text{arr}[0]] = \text{true}$ .
- For each element, build a new cur array using prev.
- After processing all elements, check all achievable sums and compute the minimum difference.

### Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int minSubsetSumDifference(vector<int>& arr, int n) {
 int totSum = 0;
 for (int x : arr) totSum += x;

 vector<bool> prev(totSum + 1, false);
 prev[0] = true;
 if (arr[0] <= totSum)
 prev[arr[0]] = true;

 for (int i = 1; i < n; i++) {
```

```

 vector<bool> cur(totSum + 1, false);
 cur[0] = true;
 for (int s = 1; s <= totSum; s++) {
 bool notTaken = prev[s];
 bool taken = false;
 if (arr[i] <= s)
 taken = prev[s - arr[i]];
 cur[s] = notTaken || taken;
 }
 prev = cur;
 }

 int mini = INT_MAX;
 for (int s = 0; s <= totSum; s++) {
 if (prev[s]) {
 int diff = abs(totSum - 2 * s);
 mini = min(mini, diff);
 }
 }
 return mini;
}

int main() {
 vector<int> arr = {1, 2, 3, 4};
 Solution sol;
 cout << sol.minSubsetSumDifference(arr, arr.size());
 return 0;
}

```

## Complexity Analysis

- Time Complexity:  $O(N*K)$
- Space Complexity:  $O(K)$

# Count Subsets with Sum K (DP-17)

We are given an array  $\text{arr}$  of  $n$  integers and an integer  $K$ .

We need to **count the number of subsets** whose sum is exactly  $K$ .

A subset can be **non-contiguous**, but elements must follow the original order.

---

For every element, we have **two choices**:

1. **Include** the element → reduce target by  $\text{arr}[i]$
2. **Exclude** the element → target remains the same

This naturally leads to a **pick / not-pick DP pattern**.

Arr

|   |   |   |   |
|---|---|---|---|
| 1 | 2 | 2 | 3 |
|---|---|---|---|

$K = 3$



**We can have the following unique subsets with target Sum of 3**

[ 1, 2 ]  
Arr[0] Arr[1]

[ 1, 2 ]  
Arr[0] Arr[2]

[ 3 ]  
Arr[3]

**Total Count of subsets : 3**

# Memoization Approach

- Define  $dp[ind][target]$  = number of subsets using elements  $0..ind$  with sum = target

Recursive relation:

```
dp[ind][target] =
 dp[ind-1][target] // not pick
+ dp[ind-1][target - arr[ind]] // pick (if arr[ind] <= target)
●
```

## Base Cases

- If  $target == 0 \rightarrow$  return 1 (empty subset)
- If  $ind == 0$ :
  - return 1 if  $arr[0] == target$
  - else 0

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int solve(int ind, int target, vector<int>& arr,
vector<vector<int>>& dp) {
 if (target == 0) return 1;
 if (ind == 0) return (arr[0] == target);

 if (dp[ind][target] != -1)
 return dp[ind][target];

 int notTake = solve(ind - 1, target, arr, dp);
 int take = solve(ind - 1, target - arr[ind], arr, dp);
 dp[ind][target] = take + notTake;
 return dp[ind][target];
 }
};
```

```

 int take = 0;
 if (arr[ind] <= target)
 take = solve(ind - 1, target - arr[ind], arr, dp);

 return dp[ind][target] = take + notTake;
 }

 int countSubsets(vector<int>& arr, int K) {
 int n = arr.size();
 vector<vector<int>> dp(n, vector<int>(K + 1, -1));
 return solve(n - 1, K, arr, dp);
 }
};

int main() {
 vector<int> arr = {1, 2, 2, 3};
 int K = 3;
 Solution obj;
 cout << obj.countSubsets(arr, K);
 return 0;
}

```

## Complexity

- **Time:**  $O(N * K)$
  - **Space:**  $O(N * K) + \text{recursion stack}$
- 

## Tabulation Approach

### Algorithm

- $dp[i][t] = \text{number of subsets using elements } 0..i \text{ with sum } t$
- Initialize:

- $dp[i][0] = 1$  for all  $i$
- $dp[0][arr[0]] = 1$  (if  $arr[0] \leq K$ )
- Fill table using the same include / exclude logic

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int countSubsets(vector<int>& arr, int K) {
 int n = arr.size();
 vector<vector<int>> dp(n, vector<int>(K + 1, 0));

 for (int i = 0; i < n; i++)
 dp[i][0] = 1;

 if (arr[0] <= K)
 dp[0][arr[0]] = 1;

 for (int i = 1; i < n; i++) {
 for (int t = 0; t <= K; t++) {
 int notTake = dp[i - 1][t];
 int take = 0;
 if (arr[i] <= t)
 take = dp[i - 1][t - arr[i]];
 dp[i][t] = take + notTake;
 }
 }
 return dp[n - 1][K];
 }
};

int main() {
 vector<int> arr = {1, 2, 2, 3};
 int K = 3;
```

```

 Solution obj;
 cout << obj.countSubsets(arr, K);
 return 0;
}

```

## Complexity

- **Time:**  $O(N * K)$
  - **Space:**  $O(N * K)$
- 

## Space Optimized Approach

### Algorithm

We observe:

$dp[i][t]$  depends only on  $dp[i-1][t]$  and  $dp[i-1][t - arr[i]]$

So we only need **one 1D array**.

- $dp[t] = \text{number of subsets with sum } t$
- Traverse  $t$  **backwards** to avoid overwriting needed values

### Code

```

#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int countSubsets(vector<int>& arr, int K) {
 vector<int> dp(K + 1, 0);
 dp[0] = 1;

```

```

 if (arr[0] <= K)
 dp[arr[0]]++;

 for (int i = 1; i < arr.size(); i++) {
 for (int t = K; t >= arr[i]; t--) {
 dp[t] += dp[t - arr[i]];
 }
 }
 return dp[K];
 }

};

int main() {
 vector<int> arr = {1, 2, 2, 3};
 int K = 3;
 Solution obj;
 cout << obj.countSubsets(arr, K);
 return 0;
}

```

## Complexity

- **Time:**  $O(N * K)$
- **Space:**  $O(K)$

# Count Partitions with Given Difference (DP-18)

Given an array of **N positive integers** and an integer **D**, we need to count the number of ways to partition the array into two subsets **S1** and **S2** such that  **$S1 - S2 = D$  and  $S1 \geq S2$ .**

The order of elements inside subsets does not matter, but each element must belong to exactly one subset.

---

## Example Explanation

### Example 1

Input:  $\text{arr} = [1, 1, 2, 3]$ ,  $D = 1$

Total sum =  $1 + 1 + 2 + 3 = 7$

Let sum of S1 =  $x$  and sum of S2 =  $y$

We know:

- $x - y = 1$
- $x + y = 7$

Adding both equations:

$$2x = 8 \rightarrow x = 4$$

So the problem becomes:

**Count subsets with sum = 4**

Valid subsets with sum 4:

- $[1, 1, 2]$
- $[1, 3]$  (using first 1)
- $[1, 3]$  (using second 1)

So, answer = 3

---

## Approach

1. Let totalSum be the sum of all elements in the array.

2. Let subset sums be S1 and S2.
  3. From the problem:
    - o  $S1 - S2 = D$
    - o  $S1 + S2 = \text{totalSum}$
  4. Adding both equations:
    - o  $2 \times S1 = \text{totalSum} + D$
    - o  $S1 = (\text{totalSum} + D) / 2$
  5. If  $(\text{totalSum} + D)$  is odd, partition is impossible → return 0.
  6. If  $D > \text{totalSum}$ , partition is impossible → return 0.
  7. Now the problem reduces to:
    - o **Count the number of subsets with sum = target**
    - o where  $\text{target} = (\text{totalSum} + D) / 2$
  8. Use Dynamic Programming to count subsets with given sum.
- 

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int countPartitions(vector<int>& arr, int d) {
 int totalSum = accumulate(arr.begin(), arr.end(), 0);

 // Invalid cases
 if (d > totalSum || (totalSum + d) % 2 != 0)
 return 0;
```

```

int target = (totalSum + d) / 2;
int n = arr.size();

vector<int> dp(target + 1, 0);

// Base case: one way to make sum 0 (empty subset)
dp[0] = 1;

// First element handling
if (arr[0] <= target)
 dp[arr[0]] += 1;

// Fill DP array
for (int i = 1; i < n; i++) {
 vector<int> curr(target + 1, 0);
 curr[0] = 1;

 for (int sum = 0; sum <= target; sum++) {
 int notTake = dp[sum];
 int take = 0;
 if (arr[i] <= sum)
 take = dp[sum - arr[i]];
 curr[sum] = take + notTake;
 }
 dp = curr;
}

return dp[target];
}

};

int main() {
 Solution sol;
 vector<int> arr = {1, 1, 2, 3};
 int d = 1;
 cout << sol.countPartitions(arr, d) << endl;
 return 0;
}

```

---

## Complexity Analysis

### Time Complexity:

$O(N \times K)$

Where N is the number of elements and  $K = (\text{totalSum} + D) / 2$ .

Each DP state is computed once.

### Space Complexity:

$O(K)$

We use a single 1D DP array of size target + 1.

# Assign Cookies

A teacher wants to distribute cookies to students.

Each student can get **at most one cookie**.

- $\text{student}[i]$  = minimum cookie size required by the ith student
- $\text{cookie}[j]$  = size of the jth cookie
- A cookie can be assigned if  $\text{cookie}[j] \geq \text{student}[i]$

The task is to **maximize the number of students who get a cookie**.

---

## Example Explanation

### Example 1

Student = [1, 2, 3]

Cookie = [1, 1]

Sort both arrays:

Student → [1, 2, 3]

Cookie → [1, 1]

- Cookie 1 → satisfies student 1
- Next cookie 1 → cannot satisfy student 2

So only **1 student** is satisfied.

---

### **Example 2**

Student = [1, 2]

Cookie = [1, 2, 3]

Sorted arrays remain the same.

- Cookie 1 → student 1
- Cookie 2 → student 2

So **2 students** are satisfied.

---

## **Approach 1: Memoization**

### **Algorithm**

- Sort both arrays.
- Use recursion to try all possibilities.
- State: (studentIndex, cookieIndex)
- Choices:
  - Skip current cookie.
  - If cookie can satisfy student, assign it.

- Store results in a 2D DP table to avoid recomputation.
- Stop when all students or cookies are processed.

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int findContentChildren(vector<int>& student, vector<int>& cookie)
 {
 sort(student.begin(), student.end());
 sort(cookie.begin(), cookie.end());

 int n = student.size(), m = cookie.size();
 vector<vector<int>> dp(n, vector<int>(m, -1));

 return solve(0, 0, student, cookie, dp);
 }

private:
 int solve(int i, int j, vector<int>& student, vector<int>& cookie,
 vector<vector<int>>& dp) {
 if (i == student.size() || j == cookie.size())
 return 0;

 if (dp[i][j] != -1)
 return dp[i][j];

 int skip = solve(i, j + 1, student, cookie, dp);
 int take = 0;

 if (cookie[j] >= student[i])
 take = 1 + solve(i + 1, j + 1, student, cookie, dp);

 return dp[i][j] = max(skip, take);
 }
}
```

```

};

int main() {
 vector<int> student = {1, 2, 3};
 vector<int> cookie = {1, 1};

 Solution obj;
 cout << obj.findContentChildren(student, cookie);
 return 0;
}

```

## Complexity Analysis

- **Time Complexity:**  $O(n \times m)$   
Every student–cookie pair is evaluated once.
  - **Space Complexity:**  $O(n \times m) + O(n + m)$   
DP table + recursion stack.
- 

## Approach 2: Tabulation

### Algorithm

- Sort both arrays.
- Create a DP table where  $dp[i][j]$  represents the maximum students satisfied using students from  $i$  and cookies from  $j$ .
- Fill the table bottom-up.
- At each cell:
  - Skip cookie.
  - Assign cookie if possible.
- Final answer is at  $dp[0][0]$ .

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int findContentChildren(vector<int>& student, vector<int>& cookie)
{
 sort(student.begin(), student.end());
 sort(cookie.begin(), cookie.end());

 int n = student.size(), m = cookie.size();
 vector<vector<int>> dp(n + 1, vector<int>(m + 1, 0));

 for (int i = n - 1; i >= 0; i--) {
 for (int j = m - 1; j >= 0; j--) {
 int skip = dp[i][j + 1];
 int take = 0;
 if (cookie[j] >= student[i])
 take = 1 + dp[i + 1][j + 1];
 dp[i][j] = max(skip, take);
 }
 }
 return dp[0][0];
}

int main() {
 vector<int> student = {1, 2};
 vector<int> cookie = {1, 2, 3};

 Solution obj;
 cout << obj.findContentChildren(student, cookie);
 return 0;
}
```

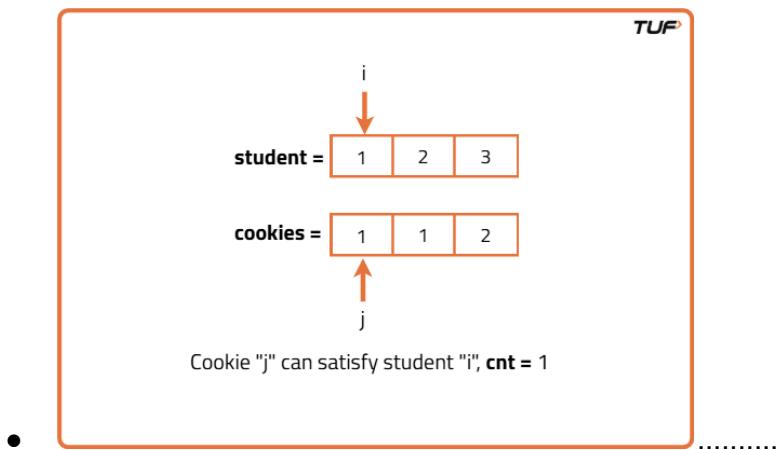
## Complexity Analysis

- **Time Complexity:**  $O(n \times m)$
  - **Space Complexity:**  $O(n \times m)$
- 

## Approach 3: Optimal (Greedy)

### Algorithm

- Sort both arrays.
- Use two pointers:
  - One for students
  - One for cookies
- If the current cookie satisfies the current student:
  - Assign it and move both pointers.
- Otherwise:
  - Skip the cookie.
- Count how many students are satisfied.



This works because we always give the **smallest possible cookie** to the **least greedy student**, leaving bigger cookies for others.

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int findContentChildren(vector<int>& student, vector<int>& cookie)
{
 sort(student.begin(), student.end());
 sort(cookie.begin(), cookie.end());

 int i = 0, j = 0;

 while (i < student.size() && j < cookie.size()) {
 if (cookie[j] >= student[i]) {
 i++;
 }
 j++;
 }
 return i;
}

int main() {
 vector<int> student = {1, 2, 3};
 vector<int> cookie = {1, 1};

 Solution obj;
 cout << obj.findContentChildren(student, cookie);
 return 0;
}
```

## Complexity Analysis

- **Time Complexity:**  $O(n \log n + m \log m)$   
Sorting both arrays.

- **Space Complexity:** O(1)  
No extra space used.

# Minimum Coins (DP-20)

## Problem Statement

Given an array `coins[ ]` of different denominations and an integer `amount`, return the **minimum number of coins** required to make that amount.

You have **infinite supply** of each coin.

If it's not possible, return **-1**.

---

## Example 1

Input: `coins = [1, 2, 5]`, `amount = 11`

Output: 3

## Explanation (step-by-step):

- Try largest useful coins first conceptually:
    - $11 = 5 + 5 + 1 \rightarrow \text{3 coins}$
  - No combination uses fewer than 3 coins.
  - Hence answer = **3**
- 

## Example 2

Input: `coins = [2, 5]`, `amount = 3`

Output: -1

## Explanation:

- Possible sums using 2 and 5: 2, 4, 5, 6, 7, ...
  - We can **never form 3**
  - Hence answer = -1
- 

## Why Greedy Fails

Greedy = always pick the largest coin first.

### Counter Example

```
coins = [1, 5, 6, 9], amount = 11
```

Greedy picks:

- $9 + 1 + 1 \rightarrow 3 \text{ coins}$

Optimal:

- $5 + 6 \rightarrow 2 \text{ coins}$

So greedy does **not** guarantee minimum coins.

---

## Key DP Idea

At every step:

- Either **pick** a coin (stay on same index, reduce amount)
- Or **don't pick** it (try other coins)

We want the **minimum count**, not the number of ways.

---

# Memoization (Top-Down DP)

## State

$dp[x]$  = minimum coins needed to make amount  $x$

## Transitions

$dp[x] = \min(1 + dp[x - \text{coin}])$  for all  $\text{coin} \leq x$

## Base Cases

- $dp[0] = 0$
  - If amount < 0 → invalid
  - If no solution → return -1
- 

## Memoization Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int coinChange(vector<int>& coins, int amount) {
 vector<int> dp(amount + 1, -2); // -2 = not computed
 return solve(coins, amount, dp);
 }

private:
 int solve(vector<int>& coins, int rem, vector<int>& dp) {
 if (rem == 0) return 0;
 if (rem < 0) return -1;
 if (dp[rem] != -2) return dp[rem];

 int mini = INT_MAX;
 for (int coin : coins) {
 if (rem - coin >= 0) {
 int subproblem = solve(coins, rem - coin, dp);
 if (subproblem != -1) {
 mini = min(mini, subproblem + 1);
 }
 }
 }
 dp[rem] = mini;
 return mini;
 }
}
```

```

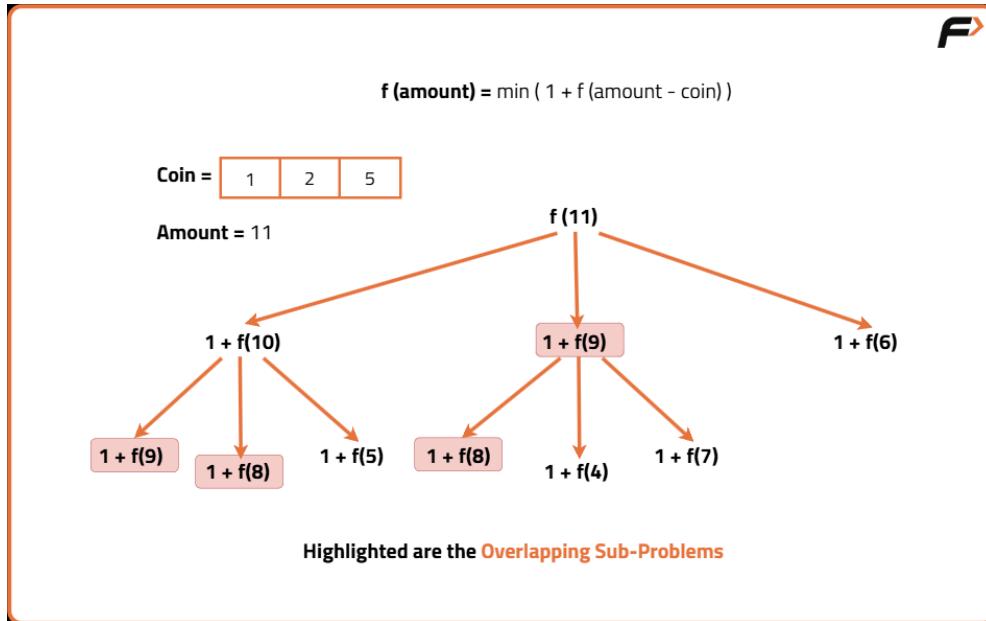
 int res = solve(coins, rem - coin, dp);
 if (res >= 0)
 mini = min(mini, 1 + res);
 }

 dp[rem] = (mini == INT_MAX) ? -1 : mini;
 return dp[rem];
}
};

```

## Complexity

- **Time:**  $O(N \times \text{amount})$
- **Space:**  $O(\text{amount}) + \text{recursion stack}$



## Tabulation (Bottom-Up DP)

$\text{dp}[x]$  = minimum coins needed to make amount  $x$

### Initialization

- $dp[0] = 0$
- All others =  $\infty$

## Iteration

For each amount from  $1 \rightarrow \text{amount}$ , try all coins.

---

### Tabulation Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int coinChange(vector<int>& coins, int amount) {
 vector<int> dp(amount + 1, INT_MAX);
 dp[0] = 0;

 for (int i = 1; i <= amount; i++) {
 for (int coin : coins) {
 if (i - coin >= 0 && dp[i - coin] != INT_MAX) {
 dp[i] = min(dp[i], 1 + dp[i - coin]);
 }
 }
 }

 return dp[amount] == INT_MAX ? -1 : dp[amount];
 }
};
```

Coins = 

|   |   |   |
|---|---|---|
| 1 | 2 | 5 |
|---|---|---|

 Amount = 15

Initially our DP array is

|   |          |          |          |          |          |          |          |          |          |          |          |          |
|---|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|
| 0 | $\infty$ |
| 0 | 1        | 2        | 3        | 4        | 5        | 6        | 7        | 8        | 9        | 10       | 11       |          |

Using relation,  $dp[i] = \min(1 + dp[i - \text{coin}])$

|   |   |   |   |   |   |   |   |   |   |    |    |
|---|---|---|---|---|---|---|---|---|---|----|----|
| 0 | 1 | 1 | 2 | 2 | 1 | 2 | 2 | 3 | 3 | 2  | 3  |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |

### Example Dry Run (coins = [1,2,5], amount = 5)

Amount dp

|   |   |
|---|---|
| 0 | 0 |
| 1 | 1 |
| 2 | 1 |
| 3 | 2 |
| 4 | 2 |
| 5 | 1 |

### Complexity

- Time:  $O(N \times \text{amount})$
- Space:  $O(\text{amount})$

## Space Optimized DP (Unbounded Knapsack Style)

We can reuse the **same row** because coins can be used unlimited times.

---

### Space Optimized Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int minimumElements(vector<int>& coins, int T) {
 int n = coins.size();
 vector<int> dp(T + 1, 1e9);
 dp[0] = 0;

 for (int coin : coins) {
 for (int t = coin; t <= T; t++) {
 dp[t] = min(dp[t], 1 + dp[t - coin]);
 }
 }

 return dp[T] >= 1e9 ? -1 : dp[T];
 }
};
```

### Complexity

- **Time:**  $O(N \times T)$
- **Space:**  $O(T)$

# Target Sum (DP - 21)

We are given an array ARR of size N and an integer Target. We must place either a + or - sign in front of every element of the array. After placing signs in front of all elements, we add them together. Our task is to count the total number of different ways in which the final sum becomes exactly equal to Target.

Example 1:

nums = [1,1,1,1,1], target = 3

Possible valid expressions:

$$-1 + 1 + 1 + 1 + 1 = 3$$

$$+1 - 1 + 1 + 1 + 1 = 3$$

$$+1 + 1 - 1 + 1 + 1 = 3$$

$$+1 + 1 + 1 - 1 + 1 = 3$$

$$+1 + 1 + 1 + 1 - 1 = 3$$

So total ways = 5.

Example 2:

nums = [1], target = 1

Only one way:  $+1 = 1$ , so answer = 1.

---

## Approach 1: Memoization

### Algorithm

Trying all + and - combinations directly is exponential. Instead, we reduce the problem to a known DP problem.

Let:

- totSum = sum of all array elements
- Elements with + sign form subset S1
- Elements with - sign form subset S2

We know:

$$S1 - S2 = \text{target}$$

$$S1 + S2 = \text{totSum}$$

Solving these equations:

$$S2 = (\text{totSum} - \text{target}) / 2$$

So the problem becomes:

**Count number of subsets whose sum is  $(\text{totSum} - \text{target}) / 2$**

Edge cases:

- If  $\text{totSum} < \text{target}$ , answer is 0
- If  $(\text{totSum} - \text{target})$  is odd, answer is 0

Now we count subsets with given sum using recursion + memoization.

Recursive definition  $f(\text{ind}, \text{target})$ :

- Number of ways to form target using elements from index 0 to  $\text{ind}$ .

Base cases:

- If  $\text{ind} == 0$ :
  - If  $\text{target} == 0$  and  $\text{arr}[0] == 0$ , return 2 (pick or not pick)
  - If  $\text{target} == 0$  or  $\text{target} == \text{arr}[0]$ , return 1
  - Else return 0

Choices:

- Not pick current element  $\rightarrow f(\text{ind}-1, \text{target})$
- Pick current element (if possible)  $\rightarrow f(\text{ind}-1, \text{target} - \text{arr}[\text{ind}])$

Store results in  $dp[\text{ind}][\text{target}]$ .

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int findTargetSumWays(vector<int>& nums, int target) {
 int totalSum = accumulate(nums.begin(), nums.end(), 0);
 if ((totalSum - target) < 0 || (totalSum - target) % 2 != 0)
 return 0;

 int subsetSum = (totalSum - target) / 2;
 vector<vector<int>> dp(nums.size(), vector<int>(subsetSum + 1,
-1));
 return countSubsets(nums, nums.size() - 1, subsetSum, dp);
 }

 int countSubsets(vector<int>& nums, int ind, int target,
vector<vector<int>>& dp) {
 if (ind == 0) {
 if (target == 0 && nums[0] == 0) return 2;
 if (target == 0 || target == nums[0]) return 1;
 return 0;
 }
 if (dp[ind][target] != -1) return dp[ind][target];

 int notPick = countSubsets(nums, ind - 1, target, dp);
 int pick = 0;
 if (nums[ind] <= target)
 pick = countSubsets(nums, ind - 1, target - nums[ind],
dp);

 return dp[ind][target] = pick + notPick;
 }
};

int main() {
 vector<int> nums = {1,1,1,1,1};
```

```

int target = 3;
Solution sol;
cout << sol.findTargetSumWays(nums, target);
return 0;
}

```

## Complexity Analysis

Time Complexity:  $O(N*K)$

There are  $N*K$  states and each is computed once.

Space Complexity:  $O(N*K) + O(N)$

$N*K$  for DP table and  $O(N)$  recursion stack.

---

## Approach 2: Tabulation

### Algorithm

We convert the memoization solution into bottom-up DP.

Let  $\text{newTarget} = (\text{totalSum} + \text{target}) / 2$ .

Create a DP table:

$dp[i][j] = \text{number of ways to form sum } j \text{ using first } i \text{ elements.}$

Initialization:

- $dp[i][0] = 1$  for all  $i$  (empty subset gives sum 0)

Transition:

- Not pick:  $dp[i-1][j]$
- Pick (if possible):  $dp[i-1][j - \text{nums}[i-1]]$

Final answer:  $dp[n][\text{newTarget}]$ .



Eg: [4, 2, 3, 7, ..., 23]  
Target = 11

target →

| target /ind | 0 | 1 | 2 | 3 | 4 | ... | k |
|-------------|---|---|---|---|---|-----|---|
| 0           | 0 | 0 | 0 | 0 | 0 | 0   | 0 |
| 1           | 0 | 0 | 0 | 0 | 0 | 0   | 0 |
| :           | 0 | 0 | 0 | 0 | 0 | 0   | 0 |
| N - 1       | 0 | 0 | 0 | 0 | 0 | 0   | 0 |

target →

| target /ind | 0 | 1     | 2     | 3     | 4     | ...   | k     |
|-------------|---|-------|-------|-------|-------|-------|-------|
| 0           | 1 | false | false | false | false | false | false |
| 1           | 1 | false | false | false | false | false | false |
| :           | 1 | false | false | false | false | false | false |
| N - 1       | 1 | false | false | false | false | false | false |



Eg: [4, 2, 3, 7, ..., 23]

Target = 11

arr[0] = 4

target →

| target IND | 0 | 1 | 2 | 3 | 4 | ... | k |
|------------|---|---|---|---|---|-----|---|
| 0          | 1 | 0 | 0 | 0 | 1 | 0   | 0 |
| 1          | 1 | 0 | 0 | 0 | 0 | 0   | 0 |
| :          | 1 | 0 | 0 | 0 | 0 | 0   | 0 |
| N - 1      | 1 | 0 | 0 | 0 | 0 | 0   | 0 |

Ind  
↓

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int findTargetSumWays(vector<int>& nums, int target) {
 int n = nums.size();
 int totalSum = accumulate(nums.begin(), nums.end(), 0);
 if ((totalSum + target) % 2 != 0 || abs(target) > totalSum)
 return 0;

 int newTarget = (totalSum + target) / 2;
 vector<vector<int>> dp(n + 1, vector<int>(newTarget + 1, 0));

 for (int i = 0; i <= n; i++) dp[i][0] = 1;

 for (int i = 1; i <= n; i++) {
 for (int j = 0; j <= newTarget; j++) {
 dp[i][j] = dp[i - 1][j];
 if (j - nums[i] >= 0)
 dp[i][j] += dp[i - 1][j - nums[i]];
 }
 }
 return dp[n][newTarget];
 }
};
```

```

 if (nums[i - 1] <= j)
 dp[i][j] += dp[i - 1][j - nums[i - 1]];
 }
}
return dp[n][newTarget];
}

int main() {
 vector<int> nums = {1,1,1,1,1};
 int target = 3;
 Solution sol;
 cout << sol.findTargetSumWays(nums, target);
 return 0;
}

```

## Complexity Analysis

Time Complexity:  $O(N*K)$

Two nested loops over elements and target.

Space Complexity:  $O(N*K)$

Full DP table is used, recursion stack removed.

---

## Approach 3: Space Optimization

### Algorithm

From the relation:

$$dp[ind][target] = dp[ind-1][target] + dp[ind-1][target - arr[ind]]$$

We only need the previous row. So we use a 1D DP array.

Steps:

- Compute  $newTarget = (totalSum + target) / 2$
- Initialize  $dp[0] = 1$

- For each number, update dp from right to left

Final answer is `dp[newTarget]`.

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int findTargetSumWays(vector<int>& nums, int target) {
 int total = accumulate(nums.begin(), nums.end(), 0);
 if ((total + target) % 2 != 0 || abs(target) > total)
 return 0;

 int newTarget = (total + target) / 2;
 vector<int> dp(newTarget + 1, 0);
 dp[0] = 1;

 for (int num : nums) {
 for (int j = newTarget; j >= num; j--) {
 dp[j] += dp[j - num];
 }
 }
 return dp[newTarget];
 }
};

int main() {
 vector<int> nums = {1,1,1,1,1};
 int target = 3;
 Solution sol;
 cout << sol.findTargetSumWays(nums, target);
 return 0;
}
```

## Complexity Analysis

Time Complexity:  $O(N \cdot K)$

Each element updates all possible target sums once.

Space Complexity:  $O(K)$

Only one DP array of size  $K+1$  is used.

## Coin Change 2 (DP - 22)

We are given an array  $\text{Arr}$  of  $N$  **distinct coin denominations** and an integer **Target**. We have an **infinite supply** of each coin. Our task is to **count the total number of different ways** to form the target sum using these coins.

Order of coins does **not** matter.

Example 1:

coins = [2,4,10], amount = 10

Possible combinations:

$$10 = 10$$

$$10 = 4 + 4 + 2$$

$$10 = 4 + 2 + 2 + 2$$

$$10 = 2 + 2 + 2 + 2 + 2$$

So output = 4

Example 2:

coins = [5], amount = 5

Only one way:

$$5 = 5$$

So output = 1

---

# Approach 1: Memoization

## Algorithm

This is a classic **unbounded knapsack / coin change counting** problem.

We define a recursive function

`countWaysToMakeChangeUtil(ind, T)`

which returns the number of ways to form sum  $T$  using coins from index 0 to  $ind$ .

Base case:

- If  $ind == 0$ , we can only use the first coin.
- If  $T$  is divisible by  $arr[0]$ , there is exactly **1 way**.
- Otherwise, there are **0 ways**.

At each index, we have two choices:

- **Not Taken:** Do not use current coin → move to  $ind-1$
- **Taken:** Use current coin → stay at  $ind$  and reduce  $T$  by  $arr[ind]$

We store results in a DP table  $dp[ind][T]$  to avoid recomputation.

Final answer is `countWaysToMakeChange(n-1, target)`.

## Code

```
#include <bits/stdc++.h>
using namespace std;

long countWaysToMakeChangeUtil(vector<int>& arr, int ind, int T,
vector<vector<long>>& dp) {
 if (ind == 0) {
 return (T % arr[0] == 0);
 }
 if (dp[ind][T] != -1)
 return dp[ind][T];
 else
 dp[ind][T] = countWaysToMakeChangeUtil(arr, ind-1, T);
 for (int i=1; i<arr.size(); i++) {
 if (arr[i] <= T)
 dp[ind][T] += countWaysToMakeChangeUtil(arr, ind, T-arr[i]);
 }
}
```

```

long notTaken = countWaysToMakeChangeUtil(arr, ind - 1, T, dp);
long taken = 0;
if (arr[ind] <= T)
 taken = countWaysToMakeChangeUtil(arr, ind, T - arr[ind], dp);

return dp[ind][T] = notTaken + taken;
}

long countWaysToMakeChange(vector<int>& arr, int n, int T) {
 vector<vector<long>> dp(n, vector<long>(T + 1, -1));
 return countWaysToMakeChangeUtil(arr, n - 1, T, dp);
}

int main() {
 vector<int> arr = {1, 2, 3};
 int target = 4;
 int n = arr.size();
 cout << countWaysToMakeChange(arr, n, target);
 return 0;
}

```

## Complexity Analysis

Time Complexity: **O(N × T)**

Each state (ind, T) is computed once.

Space Complexity: **O(N × T) + O(T)**

DP table takes N\*T space and recursion stack can go up to T in worst case.

---

## Approach 2: Tabulation

### Algorithm

We convert the memoization approach into an iterative DP solution.

Create a DP table:

$dp[ind][target]$  = number of ways to form target using coins from 0 to ind.

Initialization:

- For  $ind = 0$ , if target is divisible by  $arr[0]$ , set  $dp[0][target] = 1$ .

Transition:

- **Not Taken:**  $dp[ind-1][target]$
- **Taken:**  $dp[ind][target - arr[ind]]$  (same row because coins are unlimited)

Final answer is  $dp[n-1][T]$ .

## Code

```
#include <bits/stdc++.h>
using namespace std;

long countWaysToMakeChange(vector<int>& arr, int n, int T) {
 vector<vector<long>> dp(n, vector<long>(T + 1, 0));

 for (int i = 0; i <= T; i++) {
 if (i % arr[0] == 0)
 dp[0][i] = 1;
 }

 for (int ind = 1; ind < n; ind++) {
 for (int target = 0; target <= T; target++) {
 long notTaken = dp[ind - 1][target];
 long taken = 0;
 if (arr[ind] <= target)
 taken = dp[ind][target - arr[ind]];
 dp[ind][target] = notTaken + taken;
 }
 }
 return dp[n - 1][T];
}
```

```

int main() {
 vector<int> arr = {1, 2, 3};
 int target = 4;
 int n = arr.size();
 cout << countWaysToMakeChange(arr, n, target);
 return 0;
}

```

## Complexity Analysis

Time Complexity: **O(N × T)**

Two nested loops over coins and target.

Space Complexity: **O(N × T)**

DP table of size N\*T. No recursion stack.

---

## Approach 3: Space Optimization

### Algorithm

From the relation:

$$dp[ind][target] = dp[ind-1][target] + dp[ind][target - arr[ind]]$$

We observe:

- We only need the **previous row** and the **current row**.

Steps:

- Initialize a 1D array prev for the first coin.
- For each next coin, build a cur array using prev and itself.
- After processing a coin, assign cur to prev.

Final answer is `prev[T]`.

## Code

```
#include <bits/stdc++.h>
using namespace std;

long countWaysToMakeChange(vector<int>& arr, int n, int T) {
 vector<long> prev(T + 1, 0);

 for (int i = 0; i <= T; i++) {
 if (i % arr[0] == 0)
 prev[i] = 1;
 }

 for (int ind = 1; ind < n; ind++) {
 vector<long> cur(T + 1, 0);
 for (int target = 0; target <= T; target++) {
 long notTaken = prev[target];
 long taken = 0;
 if (arr[ind] <= target)
 taken = cur[target - arr[ind]];
 cur[target] = notTaken + taken;
 }
 prev = cur;
 }
 return prev[T];
}

int main() {
 vector<int> arr = {1, 2, 3};
 int target = 4;
 int n = arr.size();
 cout << countWaysToMakeChange(arr, n, target);
 return 0;
}
```

## Complexity Analysis

Time Complexity: **O(N × T)**

Each coin processes all target values once.

Space Complexity: **O(T)**

Only two 1D arrays of size  $T+1$  are used.

## Unbounded Knapsack (DP - 23)

A thief wants to rob a store and has a bag with capacity  $W$ . The store has  $n$  items and each item is available in **infinite quantity**.

The weight of items is given in array  $wt[ ]$  and their corresponding values are given in array  $val[ ]$ .

The thief can either take an item completely or not take it at all (no fractions allowed).

An item can be taken **any number of times**.

We need to find the **maximum total value** the thief can steal without exceeding the bag capacity.

Example 1:

$n = 3, W = 8$

$wt = [2,4,6], val = [5,11,13]$

Best choice is taking item of weight 4 twice:

$4 + 4 = 8$ , value =  $11 + 11 = 22$

Output = 22

Example 2:

$n = 2, W = 3$

$wt = [2,1], val = [4,2]$

We can take item of weight 1 three times:

$1 + 1 + 1 = 3$ , value =  $2 + 2 + 2 = 6$

Output = 6

Greedy does not work because choosing the item with highest value or best value/weight ratio locally may block better combinations later.

# Approach 1: Memoization

## Algorithm

We solve this using recursion with memoization.

Define a function  $f(ind, W)$  which returns the **maximum value** we can get using items from index 0 to  $ind$  with remaining capacity  $W$ .

Base case:

- If  $ind == 0$ , we only have the first item.
- Since items are unlimited, we take it  $(W / wt[0])$  times.
- Value =  $(W / wt[0]) * val[0]$ .

Choices at index  $ind$ :

- **Not Take:** skip current item  $\rightarrow f(ind-1, W)$
- **Take:** take current item and stay at same index (unbounded)  
 $\rightarrow val[ind] + f(ind, W - wt[ind])$  (only if  $wt[ind] \leq W$ )

We take the maximum of both choices.

To avoid recomputation, store results in  $dp[ind][W]$ .



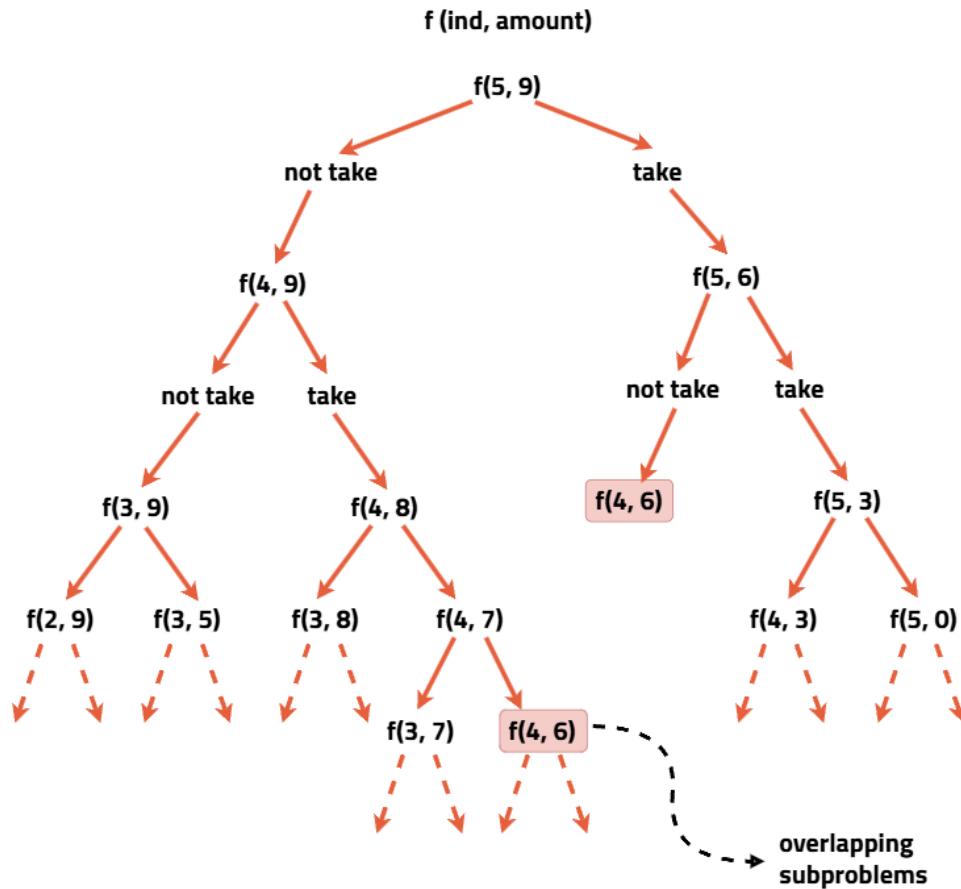
**wt =**

|   |   |   |   |   |   |
|---|---|---|---|---|---|
| 5 | 6 | 2 | 4 | 1 | 3 |
|---|---|---|---|---|---|

**val =**

|    |    |    |    |    |    |
|----|----|----|----|----|----|
| 10 | 20 | 30 | 40 | 50 | 60 |
|----|----|----|----|----|----|

**amount = 9**



## Code

```

#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int knapsackUtil(vector<int>& wt, vector<int>& val, int ind, int W, vector<vector<int>>& dp) {
 if (ind == 0) {
 return (W / wt[0]) * val[0];
 }
 if (dp[ind][W] != -1)
 return dp[ind][W];
 int notTake = knapsackUtil(wt, val, ind - 1, W, dp);
 int take = val[ind] + knapsackUtil(wt, val, ind - 1, W - wt[ind], dp);
 dp[ind][W] = max(notTake, take);
 return dp[ind][W];
 }
};

```

```

 }

 if (dp[ind][W] != -1)
 return dp[ind][W];

 int notTaken = knapsackUtil(wt, val, ind - 1, W, dp);
 int taken = INT_MIN;
 if (wt[ind] <= W)
 taken = val[ind] + knapsackUtil(wt, val, ind, W - wt[ind],
dp);

 return dp[ind][W] = max(notTaken, taken);
}

int unboundedKnapsack(int n, int W, vector<int>& val, vector<int>&
wt) {
 vector<vector<int>> dp(n, vector<int>(W + 1, -1));
 return knapsackUtil(wt, val, n - 1, W, dp);
}
};

int main() {
 vector<int> wt = {2, 4, 6};
 vector<int> val = {5, 11, 13};
 int W = 10;
 int n = wt.size();

 Solution obj;
 cout << obj.unboundedKnapsack(n, W, val, wt);
 return 0;
}

```

## Complexity Analysis

Time Complexity: **O(N \* W)**

Each state ( $ind, W$ ) is solved once.

Space Complexity: **O(N \* W) + O(N)**

DP table uses  $N*W$  space and recursion stack uses  $O(N)$ .

---

## Approach 2: Tabulation

### Algorithm

We convert the recursive solution into bottom-up DP.

Create a DP table:

$dp[ind][cap]$  = maximum value using items  $0..ind$  with capacity  $cap$ .

Initialization:

- For  $ind = 0$ , we can take item 0 multiple times:  
 $dp[0][cap] = (cap / wt[0]) * val[0]$

Transition:

- **Not Take:**  $dp[ind-1][cap]$
- **Take:**  $val[ind] + dp[ind][cap - wt[ind]]$  (same row because unlimited)

Final answer is  $dp[n-1][W]$ .

### Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int unboundedKnapsack(int n, int W, vector<int>& val, vector<int>& wt) {
 vector<vector<int>> dp(n, vector<int>(W + 1, 0));

 for (int cap = wt[0]; cap <= W; cap++) {
 dp[0][cap] = (cap / wt[0]) * val[0];
 }

 for (int ind = 1; ind < n; ind++) {
 for (int cap = 0; cap <= W; cap++) {
 if (cap <= wt[ind]) {
 dp[ind][cap] = max(dp[ind-1][cap], val[ind] + dp[ind][cap - wt[ind]]);
 } else {
 dp[ind][cap] = dp[ind-1][cap];
 }
 }
 }
 }
}
```

```

 for (int cap = 0; cap <= W; cap++) {
 int notTaken = dp[ind - 1][cap];
 int taken = INT_MIN;
 if (wt[ind] <= cap)
 taken = val[ind] + dp[ind][cap - wt[ind]];
 dp[ind][cap] = max(notTaken, taken);
 }
 }
 return dp[n - 1][W];
}

int main() {
 vector<int> wt = {2, 4, 6};
 vector<int> val = {5, 11, 13};
 int W = 10;
 int n = wt.size();

 Solution obj;
 cout << obj.unboundedKnapsack(n, W, val, wt);
 return 0;
}

```

## Complexity Analysis

Time Complexity: **O(N \* W)**

Two nested loops over items and capacity.

Space Complexity: **O(N \* W)**

DP table of size N \* W, recursion stack eliminated.

---

## Approach 3: Space Optimization

### Algorithm

From the relation:

$dp[ind][cap] = \max(dp[ind-1][cap], val[ind] + dp[ind][cap - wt[ind]])$

We notice:

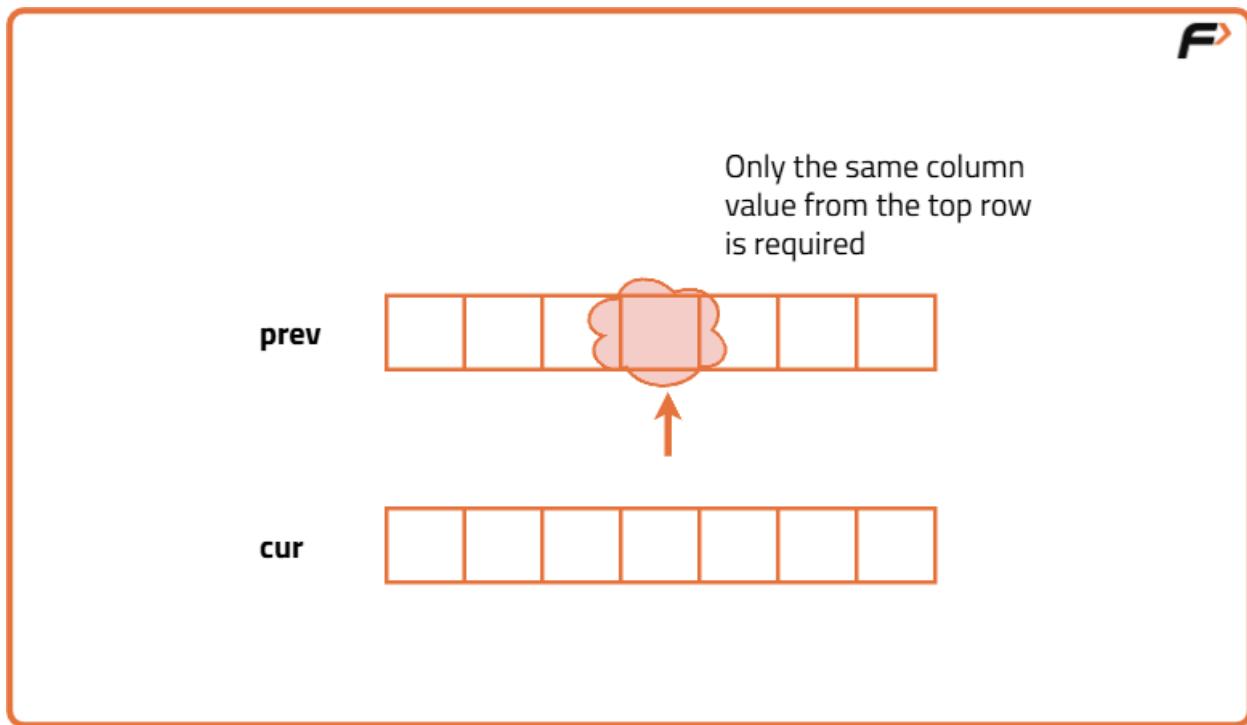
- $dp[ind-1][cap] \rightarrow$  previous value
- $dp[ind][cap - wt[ind]] \rightarrow$  current row value

So we can use **only one 1D array**.

Steps:

- Initialize the array for the first item.
- For each next item, update the same array from left to right.
- This works because unlimited items allow reuse of current row values.

Final answer is stored in  $cur[W]$ .



## Code

```
#include <bits/stdc++.h>
using namespace std;
```

```

class Solution {
public:
 int unboundedKnapsack(int n, int W, vector<int>& val, vector<int>& wt) {
 vector<int> cur(W + 1, 0);

 for (int cap = wt[0]; cap <= W; cap++) {
 cur[cap] = (cap / wt[0]) * val[0];
 }

 for (int ind = 1; ind < n; ind++) {
 for (int cap = 0; cap <= W; cap++) {
 int notTaken = cur[cap];
 int taken = INT_MIN;
 if (wt[ind] <= cap)
 taken = val[ind] + cur[cap - wt[ind]];
 cur[cap] = max(notTaken, taken);
 }
 }
 return cur[W];
 }
};

int main() {
 vector<int> wt = {2, 4, 6};
 vector<int> val = {5, 11, 13};
 int W = 10;
 int n = wt.size();

 Solution obj;
 cout << obj.unboundedKnapsack(n, W, val, wt);
 return 0;
}

```

## Complexity Analysis

Time Complexity: **O(N \* W)**

Each item updates all capacities once.

Space Complexity: **O(W)**

Only one array of size  $W+1$  is used.

## Rod Cutting Problem (DP - 24)

We are given a rod of length  $N$  inches and an array  $\text{price}[]$ , where  $\text{price}[i]$  represents the value of a rod piece of length ( $i + 1$ ) inches.

We can cut the rod into any number of pieces (or not cut it at all) and sell the pieces.

Our goal is to **maximize the total value** obtained by cutting and selling the rod.

This problem is an **unbounded knapsack** problem because:

- Each rod length can be used multiple times.
- We want to maximize value under a length constraint.

Example 1:

$\text{price} = [1, 6, 8, 9, 10, 19, 7, 20]$ ,  $N = 8$

Best cut: lengths 2 and 6 → value =  $6 + 19 = 25$

Example 2:

$\text{price} = [1, 5, 8, 9]$ ,  $N = 4$

Best cut:  $2 + 2 \rightarrow \text{value} = 5 + 5 = 10$

Greedy does not work because choosing the locally best price may block better future combinations.

---

## Approach 1: Memoization

### Algorithm

We use recursion with memoization.

Define a function  $f(ind, len)$  that returns the **maximum value** we can obtain using rod lengths from index 0 to  $ind$  to form total length  $len$ .

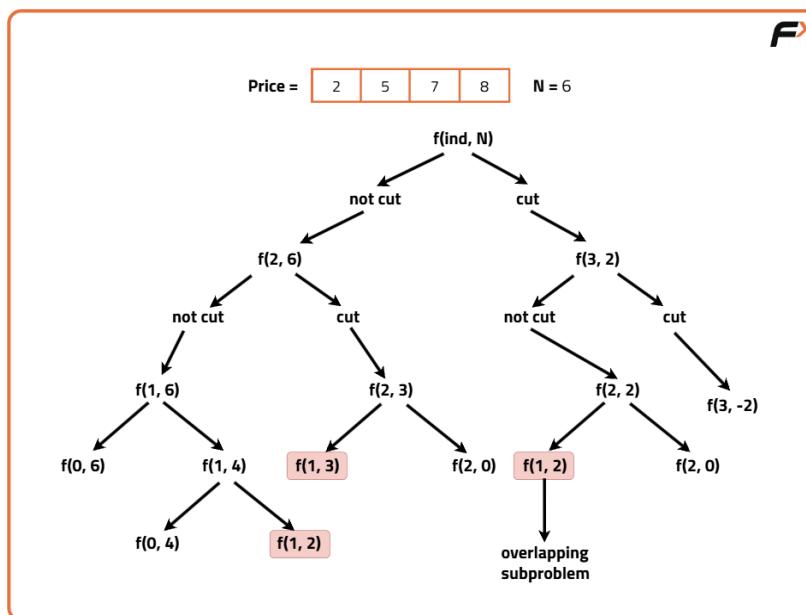
Base case:

- If  $ind == 0$ , only rod of length 1 is available.
- We can cut it  $len$  times.
- Value =  $len * price[0]$ .

Choices at index  $ind$ :

- **Not Take**: do not use current rod length  
 $\rightarrow f(ind-1, len)$
- **Take**: use current rod length ( $ind+1$ ) and stay at same index  
 $\rightarrow price[ind] + f(ind, len - (ind+1))$   
 (only if  $(ind+1) \leq len$ )

We store results in  $dp[ind][len]$  to avoid recomputation.



## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int rodCutting(vector<int> price, int n) {
 vector<vector<int>> dp(n, vector<int>(n + 1, 0));

 for(int length = 0; length <= n; length++) {
 dp[0][length] = length * price[0];
 }

 for(int i = 1; i < n; i++) {
 for(int length = 0; length <= n; length++) {
 int notTake = dp[i - 1][length];
 int take = INT_MIN;
 int rodLength = i + 1;

 if(rodLength <= length) {
 take = price[i] + dp[i][length - rodLength];
 }
 dp[i][length] = max(take, notTake);
 }
 }
 return dp[n - 1][n];
 }
};

int main() {
 int n = 8;
 vector<int> price = {1, 5, 8, 9, 10, 17, 17, 20};
 Solution obj;
 cout << obj.rodCutting(price, n);
 return 0;
}
```

## Complexity Analysis

Time Complexity:  $O(N \times N)$

Each state (`ind, length`) is computed once.

Space Complexity:  $O(N \times N)$

We use a 2D DP table.

---

## Approach 2: Tabulation

### Algorithm

This is the bottom-up version of the memoization approach.

We create a DP table:

`dp[ind][len]` = maximum value using rod lengths up to `ind` for total length `len`.

Initialization:

- For `ind = 0`,  
`dp[0][len] = len * price[0]` because rod length 1 can be used repeatedly.

Transition:

- **Not Take:** `dp[ind-1][len]`
- **Take:** `price[ind] + dp[ind][len - (ind+1)]`

Final answer is `dp[n-1][n]`.

### Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int rodCutting(vector<int>& price, int n) {
 vector<vector<int>> dp(n, vector<int>(n + 1, 0));
```

```

 for(int length = 0; length <= n; length++){
 dp[0][length] = price[0] * length;
 }

 for(int ind = 1; ind < n; ind++) {
 for(int length = 1; length <= n; length++) {
 int notTaken = dp[ind - 1][length];
 int taken = INT_MIN;
 int rodLength = ind + 1;

 if(rodLength <= length) {
 taken = price[ind] + dp[ind][length - rodLength];
 }
 dp[ind][length] = max(notTaken, taken);
 }
 }
 return dp[n - 1][n];
 }
};

int main() {
 vector<int> price = {2,4,6,8};
 int n = price.size();
 Solution sol;
 cout << sol.rodCutting(price, n);
 return 0;
}

```

## Complexity Analysis

Time Complexity: **O(N × N)**

Two nested loops over rod lengths.

Space Complexity: **O(N × N)**

We store the full DP table.

---

# Approach 3: Space Optimization

## Algorithm

From the relation:

$$dp[ind][len] = \max(dp[ind-1][len], \\ price[ind] + dp[ind][len - (ind+1)])$$

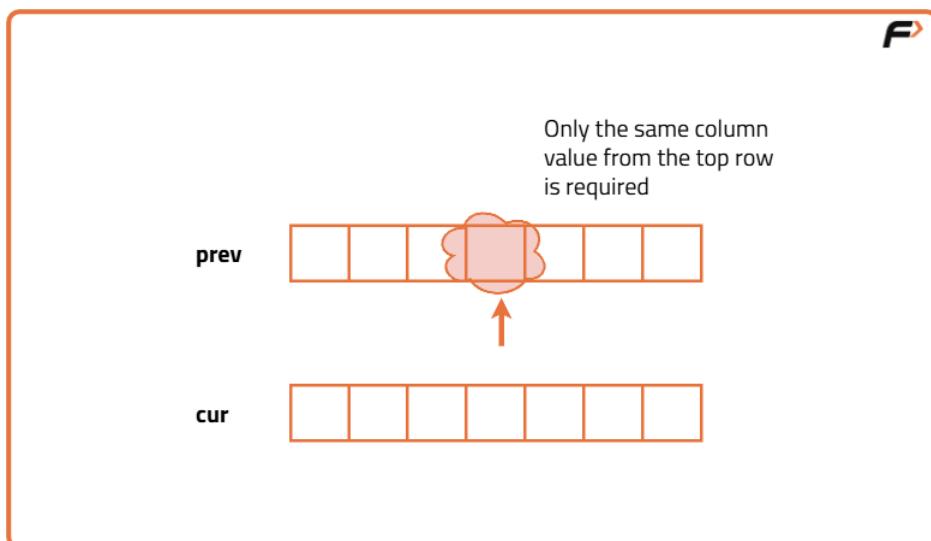
We observe:

- We only need the previous row and current row.
- So we can reduce space to **1D DP**.

Steps:

- Initialize a 1D array for the first rod length.
- Update the same array for each rod length.
- Since it is unbounded, we can reuse current row values.

Final answer will be stored in `prev[n]`.



## Code

```

#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int rodCutting(vector<int>& price, int n) {
 vector<int> prev(n + 1, 0), cur(n + 1, 0);

 for(int length = 0; length <= n; length++){
 prev[length] = price[0] * length;
 }

 for(int ind = 1; ind < n; ind++) {
 for(int length = 1; length <= n; length++) {
 int notTaken = prev[length];
 int taken = INT_MIN;
 int rodLength = ind + 1;

 if(rodLength <= length) {
 taken = price[ind] + cur[length - rodLength];
 }
 cur[length] = max(notTaken, taken);
 }
 prev = cur;
 }
 return prev[n];
 }
};

int main() {
 vector<int> price = {2,4,6,8};
 int n = price.size();
 Solution sol;
 cout << sol.rodCutting(price, n);
 return 0;
}

```

## Complexity Analysis

Time Complexity: **O(N × N)**

Each rod length processes all possible total lengths.

Space Complexity: **O(N)**

Only two 1D arrays of size N+1 are used.

# Longest Common Subsequence (DP - 25)

Given two strings str1 and str2, we need to find the **length of the longest common subsequence** present in both strings.

A subsequence keeps the relative order of characters but does not need to be contiguous.

Example 1:

str1 = "bdefg", str2 = "bfg"

The longest common subsequence is " b f g ", so the answer is 3.

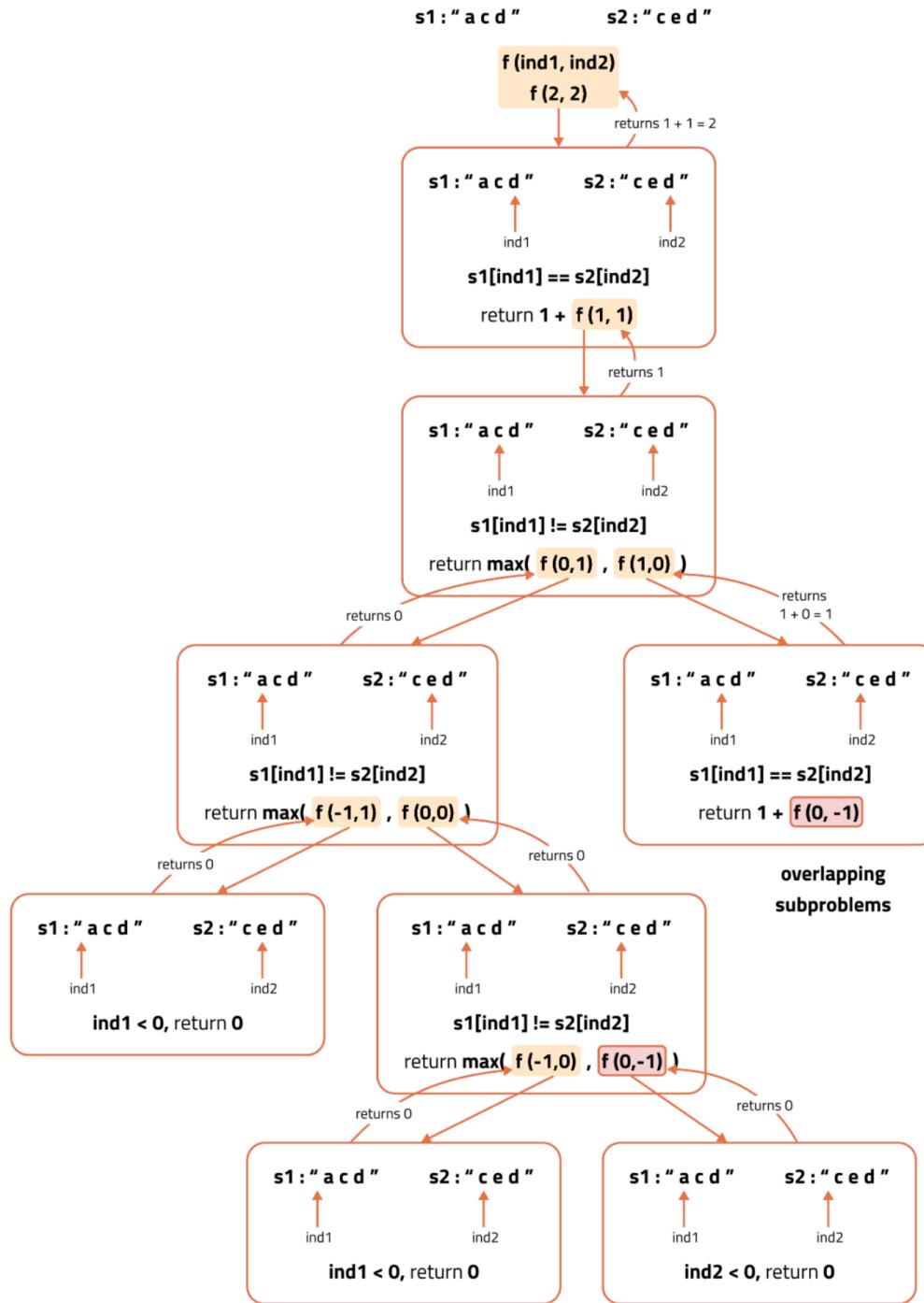
Example 2:

str1 = "mnop", str2 = "mnq"

The longest common subsequence is " m n ", so the answer is 2.

---

## Approach 1: Memoization



## Algorithm

We solve the problem using recursion with memoization.

We define a recursive function

$\text{func(ind1, ind2)}$  → length of LCS considering  $\text{str1}[0..ind1]$  and  $\text{str2}[0..ind2]$ .

Base cases:

- If  $\text{ind1} < 0$  or  $\text{ind2} < 0$ , no characters are left, so return 0.

Recursive cases:

- If  $\text{str1[ind1]} == \text{str2[ind2]}$ , this character is part of LCS, so  
 $1 + \text{func(ind1-1, ind2-1)}$
- Otherwise, we try both possibilities and take maximum:
  - $\text{func(ind1, ind2-1)}$
  - $\text{func(ind1-1, ind2)}$

Since the same  $(\text{ind1}, \text{ind2})$  states repeat, we store results in a DP table  $\text{dp}[\text{ind1}][\text{ind2}]$ .

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution{
private:
 int func(string& s1, string& s2, int ind1, int ind2,
vector<vector<int>>& dp) {
 if (ind1 < 0 || ind2 < 0)
 return 0;

 if (dp[ind1][ind2] != -1)
 return dp[ind1][ind2];

 if (s1[ind1] == s2[ind2])
 return dp[ind1][ind2] = 1 + func(s1, s2, ind1 - 1, ind2 -
1, dp);
 }
}
```

```

 return dp[ind1][ind2] = max(
 func(s1, s2, ind1, ind2 - 1, dp),
 func(s1, s2, ind1 - 1, ind2, dp)
);
 }

public:
 int lcs(string str1, string str2) {
 int n = str1.size();
 int m = str2.size();
 vector<vector<int>> dp(n, vector<int>(m, -1));
 return func(str1, str2, n - 1, m - 1, dp);
 }
};

int main() {
 string s1 = "acd";
 string s2 = "ced";
 Solution sol;
 cout << sol.lcs(s1, s2);
 return 0;
}

```

## Complexity Analysis

Time Complexity:  $O(n \times m)$

There are  $n*m$  unique ( $ind1, ind2$ ) states, each solved once.

Space Complexity:  $O(n \times m) + O(n + m)$

$O(n*m)$  for DP table and  $O(n+m)$  for recursion stack.

---

## Approach 2: Tabulation

### Algorithm

We convert the recursive solution into a bottom-up DP approach.

We create a DP table  $dp[n+1][m+1]$  where  
 $dp[i][j] = \text{LCS length of } str1[0..i-1] \text{ and } str2[0..j-1]$ .

Index shifting is done to handle base cases cleanly.

Base initialization:

- $dp[i][0] = 0$  for all  $i$
- $dp[0][j] = 0$  for all  $j$

Transition:

- If characters match:  
 $dp[i][j] = 1 + dp[i-1][j-1]$
- Else:  
 $dp[i][j] = \max(dp[i-1][j], dp[i][j-1])$

Final answer is  $dp[n][m]$ .

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution{
public:
 int lcs(string str1, string str2) {
 int n = str1.size();
 int m = str2.size();
 vector<vector<int>> dp(n + 1, vector<int>(m + 1, 0));

 for (int i = 1; i <= n; i++) {
 for (int j = 1; j <= m; j++) {
 if (str1[i - 1] == str2[j - 1])
 dp[i][j] = 1 + dp[i - 1][j - 1];
 else
 dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
 }
 }
 }
};
```

```

 }
 return dp[n][m];
 }
};

int main() {
 string s1 = "acd";
 string s2 = "ced";
 Solution sol;
 cout << sol.lcs(s1, s2);
 return 0;
}

```

## Complexity Analysis

Time Complexity:  $O(n \times m)$

We fill each cell of the DP table once.

Space Complexity:  $O(n \times m)$

A 2D DP table of size  $(n+1) \times (m+1)$  is used.

---

## Approach 3: Space Optimization

### Algorithm

From the tabulation relation, each row depends only on:

- The previous row
- The current row (left value)

So we only keep two 1D arrays:

- prev → previous row
- cur → current row

Steps:

- Initialize both arrays with 0.
- For each character of str1, compute cur using prev.
- After each row, assign prev = cur.

Final answer is stored in prev[m].

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution{
public:
 int lcs(string str1, string str2) {
 int n = str1.size();
 int m = str2.size();
 vector<int> prev(m + 1, 0), cur(m + 1, 0);

 for (int i = 1; i <= n; i++) {
 for (int j = 1; j <= m; j++) {
 if (str1[i - 1] == str2[j - 1])
 cur[j] = 1 + prev[j - 1];
 else
 cur[j] = max(prev[j], cur[j - 1]);
 }
 prev = cur;
 }
 return prev[m];
 }
};

int main() {
 string s1 = "acd";
 string s2 = "ced";
 Solution sol;
 cout << sol.lcs(s1, s2);
```

```
 return 0;
}
```

## Complexity Analysis

Time Complexity:  $O(n \times m)$

We still process all character pairs once.

Space Complexity:  $O(m)$

Only two arrays of size  $m+1$  are used instead of a full 2D table.

# Print Longest Common Subsequence (DP - 26)

Given two strings str1 and str2, we need to **print the longest common subsequence (LCS)** between them.

A subsequence maintains the **relative order** of characters but does not need to be contiguous.

This problem is an extension of **Longest Common Subsequence (DP - 25)** where instead of only finding the length, we also **reconstruct and print the actual subsequence**.

Example 1:

str1 = "abcd", str2 = "bdef"

The LCS is "bd".

Example 2:

str1 = "apple", str2 = "waffle"

The LCS is "ale".

---

## Approach

### Algorithm

To print the LCS, we first build the **DP table for LCS length**, then **trace back** from the table to construct the subsequence.

Steps:

1. Let  $n = \text{length of str1}$ ,  $m = \text{length of str2}$ .
2. Create a DP table  $dp[n+1][m+1]$  where  
 $dp[i][j]$  stores the length of LCS of  $\text{str1}[0..i-1]$  and  $\text{str2}[0..j-1]$ .
3. Fill the DP table:
  - o If  $\text{str1}[i-1] == \text{str2}[j-1]$ ,  
 $dp[i][j] = 1 + dp[i-1][j-1]$
  - o Else,  
 $dp[i][j] = \max(dp[i-1][j], dp[i][j-1])$
4. After filling the table, start from  $dp[n][m]$  and reconstruct the LCS:
  - o If characters match, add the character to answer and move diagonally ( $i-1$ ,  $j-1$ )
  - o Else, move in the direction of the larger DP value:
    - If  $dp[i-1][j] > dp[i][j-1]$ , move up
    - Otherwise, move left
5. Continue until  $i == 0$  or  $j == 0$ .
6. The LCS is built in reverse order, so reverse it before returning.

Example dry run ( $\text{str1} = \text{"abcd"}$ ,  $\text{str2} = \text{"bdef"}$ ):

- DP table gives LCS length = 2
- Tracing back picks 'd' then 'b'
- Reverse → "bd"

**S1 : "abcde"****S2 : "bdgek"**

|   | 0 | 1   | 2   | 3   | 4   | 5   |
|---|---|-----|-----|-----|-----|-----|
| 0 | 0 | 0   | 0   | 0   | 0   | 0   |
| 1 | 0 | 'b' | 'd' | 'g' | 'e' | 'k' |
| 2 | 0 | 'b' | 1   | 1   | 1   | 1   |
| 3 | 0 | 'c' | 1   | 1   | 1   | 1   |
| 4 | 0 | 'd' | 1   | 2   | 2   | 2   |
| 5 | 0 | 'e' | 1   | 2   | 2   | 3   |

**S1[1] == S2[0]**

|               |   |   |   |
|---------------|---|---|---|
| <b>str :</b>  | b | d | e |
| <b>size :</b> | 0 | 1 | 2 |

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 string longestCommonSubsequence(string &str1, string &str2) {
 int n = str1.size();
 int m = str2.size();

 vector<vector<int>> dp(n + 1, vector<int>(m + 1, 0));

```

```

 for (int i = 1; i <= n; i++) {
 for (int j = 1; j <= m; j++) {
 if (str1[i - 1] == str2[j - 1])
 dp[i][j] = 1 + dp[i - 1][j - 1];
 else
 dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
 }
 }

 int i = n, j = m;
 string lcs = "";

 while (i > 0 && j > 0) {
 if (str1[i - 1] == str2[j - 1]) {
 lcs += str1[i - 1];
 i--;
 j--;
 } else if (dp[i - 1][j] > dp[i][j - 1]) {
 i--;
 } else {
 j--;
 }
 }

 reverse(lcs.begin(), lcs.end());
 return lcs;
 }
};

int main() {
 string str1 = "abcd";
 string str2 = "bdef";

 Solution sol;
 cout << sol.longestCommonSubsequence(str1, str2);
 return 0;
}

```

## Complexity Analysis

Time Complexity:  $O(N \times M) + O(N + M)$

- $O(N \times M)$  to build the DP table
- $O(N + M)$  to reconstruct the LCS by tracing back

Space Complexity:  $O(N \times M)$

We use a 2D DP table of size  $(n+1) \times (m+1)$  to store LCS lengths.

# Longest Common Substring (DP - 27)

Given two strings str1 and str2, we need to find the **length of the longest common substring** between them.

A substring is a **contiguous** sequence of characters, meaning characters must appear continuously without gaps.

Example 1:

str1 = "abcde", str2 = "abfce"

The longest common substring is "ab", so the answer is 2.

Example 2:

str1 = "abcdxyz", str2 = "xyzabcd"

The longest common substring is "abcd", so the answer is 4.

The key difference from LCS is that here **continuity is mandatory**. If characters do not match, the substring breaks immediately.

---

## Approach 1: Tabulation

### Algorithm

We use dynamic programming to build a table where each cell stores the length of the longest common substring **ending at those indices**.

Steps:

1. Let  $n$  be the length of  $\text{str1}$  and  $m$  be the length of  $\text{str2}$ .
2. Create a DP table  $\text{dp}[n+1][m+1]$  initialized with 0.
3. Traverse both strings:
  - o If  $\text{str1}[i-1] == \text{str2}[j-1]$ , it means the substring can be extended, so  
 $\text{dp}[i][j] = 1 + \text{dp}[i-1][j-1]$
  - o If they do not match, continuity breaks, so  
 $\text{dp}[i][j] = 0$
4. Keep track of the **maximum value** found in the DP table, because the longest common substring may end anywhere, not necessarily at the last index.
5. Return this maximum value as the answer.

Example dry run ( $\text{str1} = \text{"abcde"}$ ,  $\text{str2} = \text{"abfce"}$ ):

- Matching "a" → length 1
- Matching "b" → length 2
- Mismatch at next characters resets length to 0  
Maximum found = 2

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int longestCommonSubstr(string str1, string str2) {
 int n = str1.size();
 int m = str2.size();
```

```

vector<vector<int>> dp(n + 1, vector<int>(m + 1, 0));
int ans = 0;

for (int i = 1; i <= n; i++) {
 for (int j = 1; j <= m; j++) {
 if (str1[i - 1] == str2[j - 1]) {
 dp[i][j] = 1 + dp[i - 1][j - 1];
 ans = max(ans, dp[i][j]);
 } else {
 dp[i][j] = 0;
 }
 }
}
return ans;
}

int main() {
 string s1 = "abcde";
 string s2 = "abfce";

 Solution sol;
 cout << sol.longestCommonSubstr(s1, s2);
 return 0;
}

```

## Complexity Analysis

Time Complexity: **O(n × m)**

Every pair of characters from both strings is compared once.

Space Complexity: **O(n × m)**

A 2D DP table of size  $(n+1) \times (m+1)$  is used.

---

## Approach 2: Space Optimized

## Algorithm

From the tabulation relation:

$$dp[i][j] = 1 + dp[i-1][j-1]$$

We observe that:

- To compute the current row, we only need the **previous row**.
- So instead of a full 2D table, we can use two 1D arrays:
  - `prev` → previous row
  - `cur` → current row

Steps:

1. Initialize two arrays `prev` and `cur` of size  $m+1$  with 0.
2. Traverse the strings:
  - If characters match,  
 $cur[j] = 1 + prev[j-1]$
  - Else,  
 $cur[j] = 0$
3. Update the maximum length found.
4. After each row, assign `prev = cur`.
5. Return the maximum length.

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int longestCommonSubstr(string str1, string str2) {
```

```

int n = str1.size();
int m = str2.size();

vector<int> prev(m + 1, 0), cur(m + 1, 0);
int ans = 0;

for (int i = 1; i <= n; i++) {
 for (int j = 1; j <= m; j++) {
 if (str1[i - 1] == str2[j - 1]) {
 cur[j] = 1 + prev[j - 1];
 ans = max(ans, cur[j]);
 } else {
 cur[j] = 0;
 }
 }
 prev = cur;
}
return ans;
}

int main() {
 string s1 = "abcdxyz";
 string s2 = "xyzabcd";

 Solution sol;
 cout << sol.longestCommonSubstr(s1, s2);
 return 0;
}

```

## Complexity Analysis

Time Complexity: **O(n × m)**

All character pairs are still processed once.

Space Complexity: **O(m)**

Only two 1D arrays of size  $m+1$  are used instead of a full 2D DP table.

# Longest Palindromic Subsequence (DP - 28)

Given a string  $s$ , we need to find the **length of the longest palindromic subsequence** present in it.

A palindrome reads the same from left to right and right to left.

A subsequence is formed by deleting some or no characters without changing the order of the remaining characters.

Example 1:

$s = "eeeme"$

The longest palindromic subsequence is "eeee", so the answer is 4.

Example 2:

$s = "annb"$

The longest palindromic subsequence is "nn", so the answer is 2.

---

## Approach 1: Tabulation

### Algorithm

A palindrome remains the same when reversed.

So, the **Longest Palindromic Subsequence (LPS)** of a string is the same as the **Longest Common Subsequence (LCS)** between the string and its reverse.

Steps:

1. Take the original string  $s$ .
2. Create another string  $t$  which is the reverse of  $s$ .
3. Now find the **LCS length** between  $s$  and  $t$ .
4. The obtained LCS length is the length of the longest palindromic subsequence.

DP construction:

- Create a DP table  $dp[n+1][n+1]$  where  
 $dp[i][j] = \text{LCS length of } s[0..i-1] \text{ and } t[0..j-1]$
- If characters match:  
 $dp[i][j] = 1 + dp[i-1][j-1]$
- Else:  
 $dp[i][j] = \max(dp[i-1][j], dp[i][j-1])$

Finally, return  $dp[n][n]$ .

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution{
public:
 int longestPalinSubseq(string s){
 string t = s;
 reverse(t.begin(), t.end());

 int n = s.size();
 vector<vector<int>> dp(n + 1, vector<int>(n + 1, 0));

 for(int i = 1; i <= n; i++){
 for(int j = 1; j <= n; j++){
 if(s[i - 1] == t[j - 1])
 dp[i][j] = 1 + dp[i - 1][j - 1];
 else
 dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
 }
 }
 return dp[n][n];
 }
};
```

```

int main(){
 string s = "bbabcbcab";
 Solution sol;
 cout << sol.longestPalinSubseq(s);
 return 0;
}

```

## Complexity Analysis

Time Complexity:  **$O(N \times N)$**

We fill a DP table of size  $N \times N$ .

Space Complexity:  **$O(N \times N)$**

A 2D DP table is used to store LCS values.

---

## Approach 2: Space Optimized

### Algorithm

In the tabulation approach, we observe that:

- To compute the current row, we only need the **previous row**.
- So instead of a full 2D DP table, we use **two 1D arrays**.

Steps:

1. Reverse the string  $s$  to get  $t$ .
2. Use two arrays:
  - $\text{prev} \rightarrow \text{represents } dp[i-1][*]$
  - $\text{cur} \rightarrow \text{represents } dp[i][*]$
3. Apply the same LCS logic as before.
4. After each row, assign  $\text{prev} = \text{cur}$ .

5. The final answer will be in `prev[n]`.



`str : "bbabcbcab"`

`rev(str) : "bacbcbabb"`

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution{
private:
 int lcs(string s1, string s2){
 int n = s1.size();
 int m = s2.size();

 vector<int> prev(m + 1, 0), cur(m + 1, 0);

 for(int i = 1; i <= n; i++){
 for(int j = 1; j <= m; j++){
 if(s1[i - 1] == s2[j - 1])
 cur[j] = 1 + prev[j - 1];
 else
 cur[j] = max(prev[j], cur[j - 1]);
 }
 prev = cur;
 }
 return prev[m];
 }

public:
```

```

int longestPalinSubseq(string s){
 string t = s;
 reverse(t.begin(), t.end());
 return lcs(s, t);
}

int main(){
 string s = "bbabcabcab";
 Solution sol;
 cout << sol.longestPalinSubseq(s);
 return 0;
}

```

## Complexity Analysis

Time Complexity: **O(N × N)**

All character pairs are processed once.

Space Complexity: **O(N)**

Only two 1D arrays of size N+1 are used instead of a 2D table.

# Minimum Insertions to Make String Palindrome (DP - 29)

Given a string s, we need to find the **minimum number of insertions** required to make the string a palindrome.

A palindrome reads the same forward and backward.

We are allowed to insert characters at **any position** in the string.

Example 1:

s = "abcaa"

We can insert "c" and "b" to form "abcacba", which is a palindrome.

So the answer is 2.

Example 2:

s = "ba"

Insert "a" at the beginning to form "aba".

So the answer is 1.

---

## Approach 1: Tabulation

### Algorithm

To make a string a palindrome with minimum insertions, we use the concept of **Longest Palindromic Subsequence (LPS)**.

Key idea:

- If we keep the longest palindromic subsequence intact, the remaining characters must be inserted to make the string a palindrome.
- Therefore:  
**Minimum insertions = length of string – length of LPS**

How to find LPS:

- The Longest Palindromic Subsequence of a string is equal to the **Longest Common Subsequence (LCS)** between the string and its reverse.

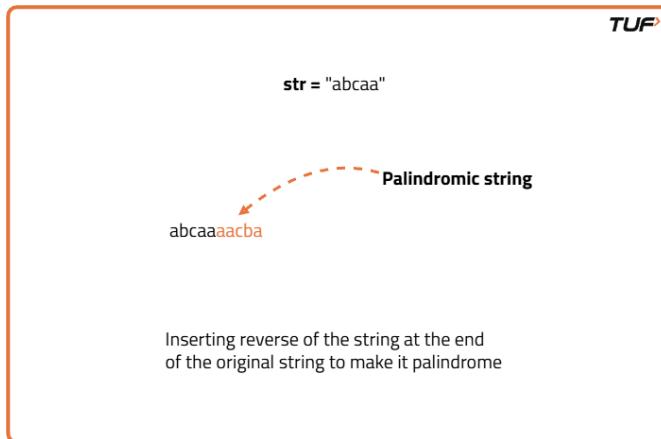
Steps:

1. Take the string s.
2. Reverse it to get rev.
3. Find LCS of s and rev using DP.
4. Let k be the length of LPS.

5. Answer =  $n - k$ , where  $n$  is the length of  $s$ .

Example dry run for "abcaa":

- Reverse = "aacba"
- LPS length = 3 ("aaa")
- Minimum insertions =  $5 - 3 = 2$



```
str = "abcaa"
```

Keep "aaa" intact as it is itself palindromic

Adding Reverse of the remaining characters to the String

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int lcs(string s1, string s2) {
 int n = s1.size();
 int m = s2.size();
 vector<vector<int>> dp(n + 1, vector<int>(m + 1, 0));

 for (int i = 1; i <= n; i++) {
 for (int j = 1; j <= m; j++) {
 if (s1[i - 1] == s2[j - 1])
 dp[i][j] = 1 + dp[i - 1][j - 1];
 else
 dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
 }
 }
 return dp[n][m];
 }

 int longestPalindromeSubsequence(string s) {
 string t = s;
 reverse(t.begin(), t.end());
 return lcs(s, t);
 }
}
```

```

 }

 int minInsertion(string s) {
 int n = s.size();
 int k = longestPalindromeSubsequence(s);
 return n - k;
 }
};

int main() {
 Solution obj;
 string s = "abcaa";
 cout << obj.minInsertion(s);
 return 0;
}

```

## Complexity Analysis

Time Complexity:  **$O(N \times N)$**

We compute LCS using a DP table of size  $N \times N$ .

Space Complexity:  **$O(N \times N)$**

A 2D DP array is used to store LCS values.

---

## Approach 2: Space Optimized

### Algorithm

In the tabulation approach, we only use:

- the previous DP row
- the current DP row

So instead of a full 2D DP array, we use **two 1D arrays**.

Steps:

1. Reverse the string to get rev.
2. Use two arrays prev and cur of size N+1.
3. Compute LCS row by row.
4. After each row, assign prev = cur.
5. LPS length = final LCS value.
6. Minimum insertions = n - LPS.

This approach reduces space while keeping the same logic.

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int lcs(string s1, string s2) {
 int n = s1.size();
 int m = s2.size();
 vector<int> prev(m + 1, 0), cur(m + 1, 0);

 for (int i = 1; i <= n; i++) {
 for (int j = 1; j <= m; j++) {
 if (s1[i - 1] == s2[j - 1])
 cur[j] = 1 + prev[j - 1];
 else
 cur[j] = max(prev[j], cur[j - 1]);
 }
 prev = cur;
 }
 return prev[m];
 }

 int longestPalindromeSubsequence(string s) {
 string t = s;
```

```

 reverse(t.begin(), t.end());
 return lcs(s, t);
 }

 int minInsertion(string s) {
 int n = s.size();
 int k = longestPalindromeSubsequence(s);
 return n - k;
 }
};

int main() {
 Solution sol;
 string s = "abcaa";
 cout << sol.minInsertion(s);
 return 0;
}

```

## Complexity Analysis

Time Complexity:  $O(N \times N)$

All character pairs are processed once while computing LCS.

Space Complexity:  $O(N)$

Only two 1D arrays of size  $N+1$  are used instead of a 2D DP table.

# Minimum Insertions/Deletions to Convert String (DP - 30)

We are given two strings `str1` and `str2`.

We are allowed only **two operations on str1**:

1. Delete any character
2. Insert any character

Our task is to find the **minimum number of operations** required to convert `str1` into `str2`.

Pre-requisite: Longest Common Subsequence

Example 1:

str1 = "kitten", str2 = "sitting"

To convert:

- delete 'k', insert 's'
- delete 'e', insert 'i'
- insert 'g'

Total operations = 5

Example 2:

str1 = "flaw", str2 = "lawn"

Delete 'f', insert 'n'

Total operations = 2

## Approach 1: Tabulation

### Algorithm

Instead of directly minimizing operations, we first think how to **maximize what we can keep unchanged.**

The characters that can remain unchanged in both strings (in the same order) form the **Longest Common Subsequence (LCS)**.

Let:

- n = length of str1
- m = length of str2
- k = length of LCS of str1 and str2

Steps:

1. Keep the k LCS characters intact.
2. Delete the remaining characters from str1: (n - k) deletions.
3. Insert the remaining characters of str2: (m - k) insertions.

So,

$$\text{Minimum operations} = (n - k) + (m - k)$$

To compute k, we use DP to find LCS.

DP definition:

- $dp[i][j] = \text{length of LCS of str1[0..i-1] and str2[0..j-1]}$

Transition:

- If characters match:  
 $dp[i][j] = 1 + dp[i-1][j-1]$
- Else:  
 $dp[i][j] = \max(dp[i-1][j], dp[i][j-1])$

Final LCS length =  $dp[n][m]$ .



```
str1 = "abcd"
```

**Delete all characters from str1 = ~~abcd~~**      4 operations

**Insert all characters of str2 to str1 = anc**      3 operations

**Number of Insertions required =  $7(4 + 3)$**

**str1 = "abcd"      str2 = "anc"**

**Longest common subsequence = "ac"**

**Keeping lcs intact and deleting all other characters from str1 = abcd**      2 operations

**str1 = "abcd"      str2 = "anc"**

**Longest common subsequence = "ac"**

**Keeping lcs intact and deleting all other characters from str1 = abcd**      2 operations

**Inserting the remaining characters in str1 = anc**      1 operations

**Minimum operations required = 3(2 + 1)**

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int minOperations(string s1, string s2) {
 int n = s1.length();
 int m = s2.length();

 vector<vector<int>> dp(n + 1, vector<int>(m + 1, 0));

 for (int i = 1; i <= n; i++) {
 for (int j = 1; j <= m; j++) {
 if (s1[i - 1] == s2[j - 1])
 dp[i][j] = dp[i - 1][j - 1] + 1;
 else
 dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
 }
 }

 return n + m - 2 * dp[n][m];
 }
};
```

```

 dp[i][j] = 1 + dp[i - 1][j - 1];
 else
 dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
 }
}

int lcs = dp[n][m];
return (n - lcs) + (m - lcs);
}

int main() {
 Solution sol;
 string s1 = "heap", s2 = "pea";
 cout << sol.minOperations(s1, s2);
 return 0;
}

```

## Complexity Analysis

Time Complexity: **O(N × M)**

We fill the entire DP table once.

Space Complexity: **O(N × M)**

A 2D DP table of size  $(n+1) \times (m+1)$  is used.

---

## Approach 2: Space Optimized

### Algorithm

In the tabulation approach, to compute the current DP row, we only need:

- the previous row
- the current row itself

So we can optimize space by using **two 1D arrays**:

- `prev` → represents  $dp[i-1][*]$
- `cur` → represents  $dp[i][*]$

Steps:

1. Initialize `prev` and `cur` arrays of size  $m+1$ .
2. Compute LCS row by row using the same transition.
3. After each row, assign `prev = cur`.
4. LCS length will be in `prev[m]`.
5. Answer =  $(n - lcs) + (m - lcs)$ .

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int minOperations(string s1, string s2) {
 int n = s1.size(), m = s2.size();

 vector<int> prev(m + 1, 0), cur(m + 1, 0);

 for (int i = 1; i <= n; i++) {
 for (int j = 1; j <= m; j++) {
 if (s1[i - 1] == s2[j - 1])
 cur[j] = 1 + prev[j - 1];
 else
 cur[j] = max(prev[j], cur[j - 1]);
 }
 prev = cur;
 }

 int lcs = prev[m];
 }
}
```

```

 return (n - lcs) + (m - lcs);
 }
};

int main() {
 Solution obj;
 string s1 = "heap", s2 = "pea";
 cout << obj.minOperations(s1, s2);
 return 0;
}

```

## Complexity Analysis

Time Complexity: **O(N × M)**

Each character pair is processed once.

Space Complexity: **O(M)**

Only two 1D arrays of size m+1 are used.

# Shortest Common Supersequence (DP - 31)

We are given two strings S1 and S2.

We need to return their **shortest common supersequence**.

A supersequence is a string that contains **both S1 and S2 as subsequences**, meaning both strings should appear in the same relative order, but not necessarily contiguously.

Example 1:

str1 = "mno", str2 = "nop"

The shortest common supersequence is "mnop".

Example 2:

str1 = "dynamic", str2 = "program"

The shortest common supersequence is "dynprogramic".

---

## Approach

### Algorithm

Pre-requisite: Longest Common Subsequence, Print Longest Common Subsequence

If we simply concatenate both strings, we will always get a supersequence, but it may not be the shortest.

To minimize the length, we should **avoid repeating common characters that appear in the same order** in both strings.

These common characters are exactly the characters of the **Longest Common Subsequence (LCS)**.

Key ideas:

- Characters of the LCS should appear **only once** in the final string.
- All non-LCS characters from both strings should be placed around the LCS while maintaining order.
- Length of Shortest Common Supersequence =  $n + m - k$   
where  $n = \text{len}(S1)$ ,  $m = \text{len}(S2)$ ,  $k = \text{length of LCS}$ .

Steps to build the string:

1. Build the DP table for LCS of S1 and S2.
2. Start from  $dp[n][m]$  and move backwards.
3. If  $S1[i-1] == S2[j-1]$ , this character is part of LCS:
  - Add it once to the answer.
  - Move diagonally ( $i--$ ,  $j--$ ).

4. If characters do not match:
  - Move towards the larger of  $dp[i-1][j]$  or  $dp[i][j-1]$ .
  - Add the corresponding character to the answer.
5. After one string is exhausted, add remaining characters of the other string.
6. Reverse the built string to get the final answer.

The diagram shows a 6x7 grid representing a dynamic programming table for the shortest common supersequence problem. The rows are indexed from 0 to 5, and the columns are indexed from 0 to 6. The grid contains the following data:

|   | 0 | 1   | 2   | 3   | 4   | 5   |
|---|---|-----|-----|-----|-----|-----|
| 0 | 0 | 'g' | 'r' | 'o' | 'o' | 't' |
| 1 | 0 | 'b' | 0   | 0   | 0   | 0   |
| 2 | 0 | 'r' | 0   | 1   | 1   | 1   |
| 3 | 0 | 'u' | 0   | 1   | 1   | 1   |
| 4 | 0 | 't' | 0   | 1   | 1   | 1   |
| 5 | 0 | 'e' | 0   | 1   | 1   | 1   |

Arrows indicate the path from (0,0) to (5,5) through cells containing 'b', 'r', 'u', and 't'. A note on the right says "break while loop and add remaining characters to ans String". Below the grid, the string "ans = etoourgb" is shown.

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 string shortestSupersequence(string s1, string s2) {
 int n = s1.size();
 int m = s2.size();

 vector<vector<int>> dp(n + 1, vector<int>(m + 1, 0));

 // Build LCS table
 for (int i = 1; i <= n; i++) {
 for (int j = 1; j <= m; j++) {
 if (s1[i - 1] == s2[j - 1])
 dp[i][j] = dp[i - 1][j - 1] + 1;
 else
 dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
 }
 }

 string ans;
 int i = n, j = m;
 while (i > 0 && j > 0) {
 if (s1[i - 1] == s2[j - 1]) {
 ans += s1[i - 1];
 i--;
 j--;
 } else if (dp[i - 1][j] > dp[i][j - 1]) {
 ans += s1[i - 1];
 i--;
 } else {
 ans += s2[j - 1];
 j--;
 }
 }

 while (i > 0)
 ans += s1[i - 1];
 while (j > 0)
 ans += s2[j - 1];

 reverse(ans.begin(), ans.end());
 return ans;
 }
};
```

```

 for (int j = 1; j <= m; j++) {
 if (s1[i - 1] == s2[j - 1])
 dp[i][j] = 1 + dp[i - 1][j - 1];
 else
 dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
 }
 }

 // Backtrack to build SCS
 int i = n, j = m;
 string ans = "";

 while (i > 0 && j > 0) {
 if (s1[i - 1] == s2[j - 1]) {
 ans += s1[i - 1];
 i--;
 j--;
 } else if (dp[i - 1][j] > dp[i][j - 1]) {
 ans += s1[i - 1];
 i--;
 } else {
 ans += s2[j - 1];
 j--;
 }
 }

 // Add remaining characters
 while (i > 0) {
 ans += s1[i - 1];
 i--;
 }
 while (j > 0) {
 ans += s2[j - 1];
 j--;
 }

 reverse(ans.begin(), ans.end());
 return ans;
}

```

```

 }
};

int main() {
 Solution obj;
 string s1 = "mno";
 string s2 = "nop";
 cout << obj.shortestSupersequence(s1, s2);
 return 0;
}

```

## Complexity Analysis

Time Complexity: **O(N × M)**

We build the LCS DP table in  $O(N \times M)$  time and backtracking takes  $O(N + M)$ .

Space Complexity: **O(N × M)**

A 2D DP table of size  $(n+1) \times (m+1)$  is used.

# Distinct Subsequences (DP - 32)

Given two strings  $s$  and  $t$ , we need to count how many **distinct subsequences** of  $s$  are equal to  $t$ .

A subsequence is formed by deleting zero or more characters from  $s$  without changing the order of remaining characters.

Example 1:

$s = "axbxax"$ ,  $t = "axa"$

There are 2 distinct ways to form "axa" from  $s$ .

Example 2:

$s = "babgbag"$ ,  $t = "bag"$

There are 5 distinct subsequences of  $s$  that equal "bag".

The task is to count all possible valid ways.

---

## Approach 1: Memoization

### Algorithm

We use recursion with memoization.

Define a recursive function  $f(i, j)$ :

- $i \rightarrow$  current index in  $s$
- $j \rightarrow$  current index in  $t$
- It returns the number of ways to form  $t[j:]$  from  $s[i:]$

Base cases:

- If  $j == t.length()$ , we have successfully formed  $t$ , so return 1.
- If  $i == s.length()$  and  $j < t.length()$ ,  $s$  is exhausted, so return 0.

Recursive cases:

- If  $s[i] == t[j]$ , we have two choices:
  1. Take this character  $\rightarrow f(i+1, j+1)$
  2. Skip this character  $\rightarrow f(i+1, j)$   
Add both results.
- If  $s[i] != t[j]$ , we can only skip  $s[i] \rightarrow f(i+1, j)$ .

To avoid recomputation, store results in a 2D DP array  $dp[i][j]$ .

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int helper(int i, int j, string &s, string &t, vector<vector<int>>
&dp) {
 if (j == t.size()) return 1;
 if (i == s.size()) return 0;
 if (dp[i][j] != -1) return dp[i][j];

 if (s[i] == t[j]) {
 int take = helper(i + 1, j + 1, s, t, dp);
 int notTake = helper(i + 1, j, s, t, dp);
 return dp[i][j] = take + notTake;
 } else {
 return dp[i][j] = helper(i + 1, j, s, t, dp);
 }
 }

 int numDistinct(string s, string t) {
 vector<vector<int>> dp(s.size(), vector<int>(t.size(), -1));
 return helper(0, 0, s, t, dp);
 }
};

int main() {
 Solution sol;
 string s = "babgbag";
 string t = "bag";
 cout << sol.numDistinct(s, t);
 return 0;
}
```

## Complexity Analysis

Time Complexity: **O(N × M)**

Each (i, j) state is computed once.

Space Complexity: **O(N × M) + O(N)**

DP table plus recursion stack space.

---

## Approach 2: Tabulation

### Algorithm

We convert the memoized solution into bottom-up DP.

Define:

- $dp[i][j] = \text{number of ways to form } t[j:] \text{ from } s[i:]$

Initialization:

- $dp[i][m] = 1$  for all  $i$  because empty  $t$  can always be formed.
- $dp[n][j] = 0$  for  $j < m$  because non-empty  $t$  cannot be formed from empty  $s$ .

Transition:

- If  $s[i] == t[j]$ :  
 $dp[i][j] = dp[i+1][j+1] + dp[i+1][j]$
- Else:  
 $dp[i][j] = dp[i+1][j]$

Answer is  $dp[0][0]$ .

### Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
```

```

public:
 int numDistinct(string s, string t) {
 int n = s.size();
 int m = t.size();

 vector<vector<long long>> dp(n + 1, vector<long long>(m + 1,
0));

 for (int i = 0; i <= n; i++) {
 dp[i][m] = 1;
 }

 for (int i = n - 1; i >= 0; i--) {
 for (int j = m - 1; j >= 0; j--) {
 if (s[i] == t[j])
 dp[i][j] = dp[i + 1][j + 1] + dp[i + 1][j];
 else
 dp[i][j] = dp[i + 1][j];
 }
 }
 return dp[0][0];
 }

int main() {
 Solution sol;
 string s = "babgbag";
 string t = "bag";
 cout << sol.numDistinct(s, t);
 return 0;
}

```

## Complexity Analysis

Time Complexity: **O(N × M)**  
 Each DP state is filled once.

Space Complexity: **O(N × M)**  
 A full 2D DP table is used.

---

## Approach 3: Space Optimization

### Algorithm

In tabulation, each row depends only on the next row.  
So we use a single 1D DP array of size  $m+1$ .

Steps:

1. Initialize  $dp[m] = 1$  (empty  $t$  case).
2. Traverse  $s$  from end to start.
3. For each character of  $s$ , update  $dp$  from right to left:
  - o If  $s[i] == t[j]$ :  
 $dp[j] = dp[j] + dp[j+1]$
  - o Else:  
 $dp[j]$  remains unchanged.
4. Final answer is  $dp[0]$ .

### Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int numDistinct(string s, string t) {
 int n = s.size();
 int m = t.size();

 vector<long long> dp(m + 1, 0);
 dp[m] = 1;

 for (int i = n - 1; i >= 0; i--) {
```

```

 for (int j = m - 1; j >= 0; j--) {
 if (s[i] == t[j]) {
 dp[j] = dp[j] + dp[j + 1];
 }
 }
 }
 return (int)dp[0];
}

int main() {
 Solution sol;
 string s = "rabbbit";
 string t = "rabbit";
 cout << sol.numDistinct(s, t);
 return 0;
}

```

## Complexity Analysis

Time Complexity: **O(N × M)**

All character pairs are processed once.

Space Complexity: **O(M)**

Only one 1D DP array of size  $m+1$  is used.

# Edit Distance (DP - 33)

We are given two strings S1 and S2.

We need to convert S1 into S2 using the **minimum number of operations**.

Allowed operations:

1. Delete a character
2. Insert a character
3. Replace a character

We must return the minimum operations required.

Example 1:

start = "planet", target = "plan"

We delete 'e' and 't'.

Answer = 2

Example 2:

start = "abcdefg", target = "azced"

Operations:

- replace b → z
- delete d
- delete f
- replace g → d

Answer = 4

---

## Approach 1: Brute Force + Memoization

### Algorithm

We solve the problem using recursion with memoization.

Define a function  $f(i, j)$ :

- $i \rightarrow$  index in S1
- $j \rightarrow$  index in S2
- It returns minimum operations to convert  $S1[0..i]$  to  $S2[0..j]$

Base cases:

- If  $i < 0$ , all remaining characters of S2 must be inserted → return  $j + 1$
- If  $j < 0$ , all remaining characters of S1 must be deleted → return  $i + 1$

Recursive cases:

- If  $S1[i] == S2[j]$ , no operation needed  
 $f(i, j) = f(i-1, j-1)$
- Else, we try all three operations:
  - Replace →  $1 + f(i-1, j-1)$
  - Delete →  $1 + f(i-1, j)$

- Insert  $\rightarrow 1 + f(i, j-1)$   
Take the minimum of these three.

Since there are overlapping subproblems, we store results in a DP table.

## Code

```
#include <bits/stdc++.h>
using namespace std;

int editDistanceUtil(string& s1, string& s2, int i, int j, vector<vector<int>>& dp) {
 if (i < 0) return j + 1;
 if (j < 0) return i + 1;

 if (dp[i][j] != -1) return dp[i][j];

 if (s1[i] == s2[j]) {
 return dp[i][j] = editDistanceUtil(s1, s2, i - 1, j - 1, dp);
 }

 int replaceOp = 1 + editDistanceUtil(s1, s2, i - 1, j - 1, dp);
 int deleteOp = 1 + editDistanceUtil(s1, s2, i - 1, j, dp);
 int insertOp = 1 + editDistanceUtil(s1, s2, i, j - 1, dp);

 return dp[i][j] = min(replaceOp, min(deleteOp, insertOp));
}

int editDistance(string& s1, string& s2) {
 int n = s1.size();
 int m = s2.size();
 vector<vector<int>> dp(n, vector<int>(m, -1));
 return editDistanceUtil(s1, s2, n - 1, m - 1, dp);
}

int main() {
 string s1 = "horse";
 string s2 = "ros";
 cout << editDistance(s1, s2);
 return 0;
}
```

## Complexity Analysis

Time Complexity: **O(N × M)**

Each state ( $i, j$ ) is computed once.

Space Complexity: **O(N × M) + O(N + M)**

DP table plus recursion stack space.

---

## Approach 2: Tabulation (Bottom-Up)

### Algorithm

In memoization, we used negative indices, which are not allowed in arrays.  
So we shift indices by 1.

Define:

- $dp[i][j] = \text{minimum operations to convert } S1[0..i-1] \text{ to } S2[0..j-1]$

Base cases:

- $dp[i][0] = i \rightarrow \text{delete all characters from } S1$
- $dp[0][j] = j \rightarrow \text{insert all characters into } S1$

Transitions:

- If  $S1[i-1] == S2[j-1]$   
 $dp[i][j] = dp[i-1][j-1]$
- Else  
 $dp[i][j] = 1 + \min(dp[i-1][j-1], dp[i-1][j], dp[i][j-1])$

Answer is  $dp[n][m]$ .

### Code

```
#include <bits/stdc++.h>
using namespace std;

int editDistance(string& s1, string& s2) {
 int n = s1.size();
 int m = s2.size();

 vector<vector<int>> dp(n + 1, vector<int>(m + 1, 0));
```

```

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

for (int i = 1; i <= n; i++) {
 for (int j = 1; j <= m; j++) {
 if (s1[i - 1] == s2[j - 1]) {
 dp[i][j] = dp[i - 1][j - 1];
 } else {
 dp[i][j] = 1 + min(dp[i - 1][j - 1],
 min(dp[i - 1][j], dp[i][j - 1]));
 }
 }
}
return dp[n][m];
}

int main() {
 string s1 = "horse";
 string s2 = "ros";
 cout << editDistance(s1, s2);
 return 0;
}

```

## Complexity Analysis

Time Complexity: **O(N × M)**

Two nested loops over both strings.

Space Complexity: **O(N × M)**

2D DP table is used.

---

## Approach 3: Space Optimized

### Algorithm

From the DP relation:

$dp[i][j]$  depends only on:

- $dp[i-1][j-1]$
- $dp[i-1][j]$

- $dp[i][j-1]$

So we only need the **previous row**.

Steps:

1. Use two arrays: prev and cur
2. Initialize  $prev[j] = j$
3. For each row  $i$ :
  - o set  $cur[0] = i$
  - o compute remaining columns
4. After each row, assign  $prev = cur$
5. Final answer is  $prev[m]$

## Code

```
#include <bits/stdc++.h>
using namespace std;

int editDistance(string& s1, string& s2) {
 int n = s1.size();
 int m = s2.size();

 vector<int> prev(m + 1, 0), cur(m + 1, 0);

 for (int j = 0; j <= m; j++) prev[j] = j;

 for (int i = 1; i <= n; i++) {
 cur[0] = i;
 for (int j = 1; j <= m; j++) {
 if (s1[i - 1] == s2[j - 1]) {
 cur[j] = prev[j - 1];
 } else {
 cur[j] = 1 + min(prev[j - 1],
 min(prev[j], cur[j - 1]));
 }
 }
 prev = cur;
 }
 return prev[m];
}

int main() {
 string s1 = "horse";
}
```

```

string s2 = "ros";
cout << editDistance(s1, s2);
return 0;
}

```

## Complexity Analysis

Time Complexity: **O(N × M)**

Same transitions as tabulation.

Space Complexity: **O(M)**

Only two rows of size M+1 are stored.

# Wildcard Matching (DP - 34)

We are given two strings S1 and S2.

S1 is a **pattern string** that can contain special characters:

- '?' matches **exactly one** character of S2
- '\*' matches **any sequence** of characters of S2 (including empty)

We need to check whether the pattern S1 **matches completely** with string S2.

Example 1:

S1 = "ab\*cd", S2 = "abdefcd"

\* matches "def", rest characters match → **true**

Example 2:

S1 = "\*a\*b", S2 = "aaab"

First \* matches "aa", second \* matches empty → **true**

## Approach 1: Memoization (Top-Down DP)

## Algorithm

We solve this using recursion with memoization.

Define a function  $f(i, j)$ :

- $i \rightarrow$  index in pattern  $S_1$
- $j \rightarrow$  index in text  $S_2$
- It returns whether  $S_1[0..i]$  matches  $S_2[0..j]$

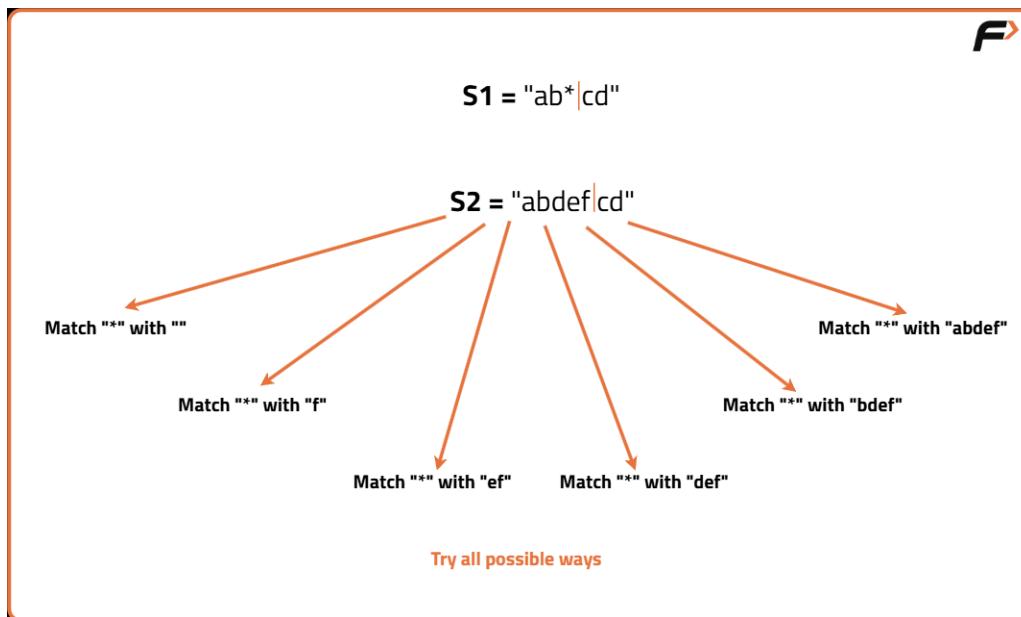
Base cases:

- If  $i < 0$  and  $j < 0 \rightarrow$  both strings finished  $\rightarrow$  true
- If  $i < 0$  and  $j \geq 0 \rightarrow$  pattern finished but text left  $\rightarrow$  false
- If  $j < 0$  and  $i \geq 0 \rightarrow$  text finished, pattern must be all '\*'

Recursive cases:

- If  $S_1[i] == S_2[j]$  or  $S_1[i] == ?$   
 $\rightarrow$  move both pointers:  $f(i-1, j-1)$
- If  $S_1[i] == ^*$ 
  - match empty sequence  $\rightarrow f(i-1, j)$
  - match one or more characters  $\rightarrow f(i, j-1)$
- Else  $\rightarrow$  characters do not match  $\rightarrow$  false

We store results in  $dp[i][j]$  to avoid recomputation.



## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 bool isAllStars(string &s, int i) {
 for (int k = 0; k <= i; k++) {
 if (s[k] != '*') return false;
 }
 return true;
 }

 bool solve(int i, int j, string &s1, string &s2, vector<vector<int>> &dp) {
 if (i < 0 && j < 0) return true;
 if (i < 0 && j >= 0) return false;
 if (j < 0 && i >= 0) return isAllStars(s1, i);

 if (dp[i][j] != -1) return dp[i][j];

 if (s1[i] == s2[j] || s1[i] == '?')
 return dp[i][j] = solve(i - 1, j - 1, s1, s2, dp);

 if (s1[i] == '*')
 return dp[i][j] = solve(i - 1, j, s1, s2, dp) ||
 solve(i, j - 1, s1, s2, dp);

 return dp[i][j] = false;
 }

 bool wildcardMatching(string &s1, string &s2) {
 int n = s1.size(), m = s2.size();
 vector<vector<int>> dp(n, vector<int>(m, -1));
 return solve(n - 1, m - 1, s1, s2, dp);
 }
};

int main() {
 string s1 = "ab*cd";
 string s2 = "abdefcd";
 Solution sol;
 cout << sol.wildcardMatching(s1, s2);
 return 0;
}
```

## Complexity Analysis

Time Complexity:  $O(N \times M)$

Each  $(i, j)$  state is computed once.

Space Complexity:  $O(N \times M) + O(N + M)$

DP table plus recursion stack.

---

## Approach 2: Tabulation (Bottom-Up DP)

### Algorithm

To remove recursion, we shift indices by 1.

Define:

$dp[i][j] = \text{true if } S1[0..i-1] \text{ matches } S2[0..j-1]$

Base cases:

- $dp[0][0] = \text{true}$
- $dp[0][j] = \text{false}$  (empty pattern cannot match non-empty text)
- $dp[i][0] = \text{true only if first } i \text{ characters of pattern are all } '*'$

Transitions:

- If  $S1[i-1] == S2[j-1]$  or  $S1[i-1] == '?'$   
 $dp[i][j] = dp[i-1][j-1]$
- If  $S1[i-1] == '*'$   
 $dp[i][j] = dp[i-1][j] \text{ || } dp[i][j-1]$
- Else  
 $dp[i][j] = \text{false}$

Final answer is  $dp[n][m]$ .

**Recursive code indexes:** -1, 0, 1, ..., n  
**Shifted indexes:** 0, 1, ..., n+1

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 bool isAllStars(string &s, int i) {
 for (int k = 1; k <= i; k++) {
 if (s[k - 1] != '*') return false;
 }
 return true;
 }

 bool wildcardMatching(string &s1, string &s2) {
 int n = s1.size(), m = s2.size();
 vector<vector<bool>> dp(n + 1, vector<bool>(m + 1, false));

 dp[0][0] = true;

 for (int i = 1; i <= n; i++)
 dp[i][0] = isAllStars(s1, i);

 for (int i = 1; i <= n; i++) {
 for (int j = 1; j <= m; j++) {
 if (s1[i - 1] == s2[j - 1] || s1[i - 1] == '?')
 dp[i][j] = dp[i - 1][j - 1];
 else if (s1[i - 1] == '*')
 dp[i][j] = dp[i - 1][j] || dp[i][j - 1];
 else
 dp[i][j] = false;
 }
 }
 }
}
```

```

 }
 return dp[n][m];
 }
};

int main() {
 string s1 = "ab*cd";
 string s2 = "abdefcd";
 Solution sol;
 cout << sol.wildcardMatching(s1, s2);
 return 0;
}

```

## Complexity Analysis

Time Complexity: **O(N × M)**

Two nested loops over pattern and text.

Space Complexity: **O(N × M)**

2D DP table.

---

## Approach 3: Space Optimization

### Algorithm

From the DP relation:

$dp[i][j]$  depends on:

- $dp[i-1][j-1]$
- $dp[i-1][j]$
- $dp[i][j-1]$

So we only need:

- previous row (prev)
- current row (cur)

Steps:

1. Initialize  $prev[0] = true$
2. For each pattern index  $i$ :

- o `cur[0] = isAllStars(s1, i)`
  - o fill remaining columns
3. After each row, do `prev = cur`
4. Answer is `prev[m]`

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 bool isAllStars(string &s, int i) {
 for (int k = 1; k <= i; k++) {
 if (s[k - 1] != '*') return false;
 }
 return true;
 }

 bool wildcardMatching(string &s1, string &s2) {
 int n = s1.size(), m = s2.size();
 vector<bool> prev(m + 1, false), cur(m + 1, false);

 prev[0] = true;

 for (int i = 1; i <= n; i++) {
 cur[0] = isAllStars(s1, i);
 for (int j = 1; j <= m; j++) {
 if (s1[i - 1] == s2[j - 1] || s1[i - 1] == '?')
 cur[j] = prev[j - 1];
 else if (s1[i - 1] == '*')
 cur[j] = prev[j] || cur[j - 1];
 else
 cur[j] = false;
 }
 prev = cur;
 }
 return prev[m];
 }
};

int main() {
 string s1 = "ab*cd";
 string s2 = "abdefcd";
```

```

Solution sol;
cout << sol.wildcardMatching(s1, s2);
return 0;
}

```

## Complexity Analysis

Time Complexity: **O(N × M)**

Same transitions as tabulation.

Space Complexity: **O(M)**

Only two rows of DP are stored.

# Stock Buy and Sell (DP-35)

We are given an array  $\text{Arr}[ ]$  of length  $n$  where  $\text{Arr}[i]$  represents the stock price on day  $i$ .

We are allowed to **buy and sell the stock only once**.

The selling day must be **the same day or after the buying day**.

Our task is to find the **maximum profit** possible.

If no profit is possible, return 0.

Example 1:

$\text{prices} = [7, 1, 5, 3, 6, 4]$

Buy at price 1 (day 2) and sell at price 6 (day 5).

Profit = 6 - 1 = 5.

Example 2:

$\text{prices} = [7, 6, 4, 3, 1]$

Prices keep decreasing, so no transaction is done.

Profit = 0.

# Approach 1: Brute Force

## Algorithm

We try **all possible buy and sell day pairs** and calculate profit for each valid pair.

Steps:

- Initialize `maxProfit = 0`
- Loop day `i` from `0` to `n-2` as buying day
- Loop day `j` from `i+1` to `n-1` as selling day
- Calculate `profit = prices[j] - prices[i]`
- Update `maxProfit` if profit is greater
- Return `maxProfit`

This checks every possible transaction.

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int maxProfit(vector<int>& prices) {
 int maxProfit = 0;

 for (int i = 0; i < prices.size(); i++) {
 for (int j = i + 1; j < prices.size(); j++) {
 int profit = prices[j] - prices[i];
 maxProfit = max(maxProfit, profit);
 }
 }
 return maxProfit;
 }
}
```

```

};

int main() {
 Solution obj;
 vector<int> prices = {7, 1, 5, 3, 6, 4};
 cout << obj.maxProfit(prices) << endl;
 return 0;
}

```

## Complexity Analysis

- **Time Complexity:**  $O(n^2)$   
Two nested loops check all buy-sell pairs.
  - **Space Complexity:**  $O(1)$   
Only variables are used, no extra data structures.
- 

## Approach 2: Optimal Approach

### Algorithm

Instead of checking all pairs, we track the **minimum price seen so far**.

Idea:

- If we sell on day *i*, the best buy day is the day with the **minimum price before day i**
- Keep updating the minimum price
- At each day, calculate profit if sold today
- Track the maximum profit

Steps:

- Initialize `minPrice = INT_MAX`

- Initialize `maxProfit = 0`
- Traverse prices from left to right
- Update `minPrice` if current price is smaller
- Calculate `profit = prices[i] - minPrice`
- Update `maxProfit`
- Return `maxProfit`

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int maxProfit(vector<int>& prices) {
 int minPrice = INT_MAX;
 int maxProfit = 0;

 for (int i = 0; i < prices.size(); i++) {
 if (prices[i] < minPrice)
 minPrice = prices[i];

 int profit = prices[i] - minPrice;
 if (profit > maxProfit)
 maxProfit = profit;
 }
 return maxProfit;
 }
};

int main() {
 Solution obj;
 vector<int> prices = {7, 1, 5, 3, 6, 4};
 cout << "Maximum Profit: " << obj.maxProfit(prices) << endl;
}
```

```
 return 0;
}
```

## Complexity Analysis

- **Time Complexity:**  $O(n)$   
Single pass through the array.
- **Space Complexity:**  $O(1)$   
Only constant extra variables are used.

# Buy and Sell Stock – III (DP-37)

We are given an array  $\text{Arr}[ ]$  of length  $n$  where  $\text{Arr}[i]$  represents the stock price on day  $i$ .

Rules:

- We can buy and sell the stock multiple times.
- To sell, we must have bought the stock before.
- We cannot hold more than one stock at a time.
- We can do **at most 2 transactions** (one transaction = one buy + one sell).

We need to find the **maximum profit**.

|                    |                                    |
|--------------------|------------------------------------|
| <b>Allowed</b>     | B S B S<br>Arr: [7, 1, 5, 3, 6, 4] |
| <b>Not Allowed</b> | S B<br>Arr: [7, 1, 5, 3, 6, 4]     |
| <b>Not Allowed</b> | B B<br>Arr: [7, 1, 5, 3, 6, 4]     |
| <b>Not Allowed</b> | B S S<br>Arr: [7, 1, 5, 3, 6, 4]   |

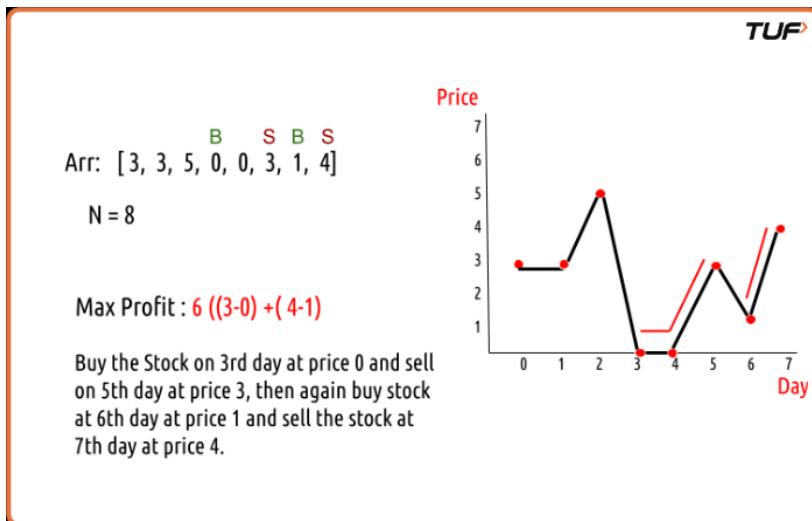
Example:

$$\text{Arr} = \{3, 3, 5, 0, 0, 3, 1, 4\}$$

$$\text{Maximum profit} = (3 - 0) + (4 - 1) = 6$$

Explanation:

Buy at price 0, sell at 3, then buy at 1, sell at 4.



## Approach 1: Memoization Approach

We solve this using recursion with memoization.

On each day, we decide:

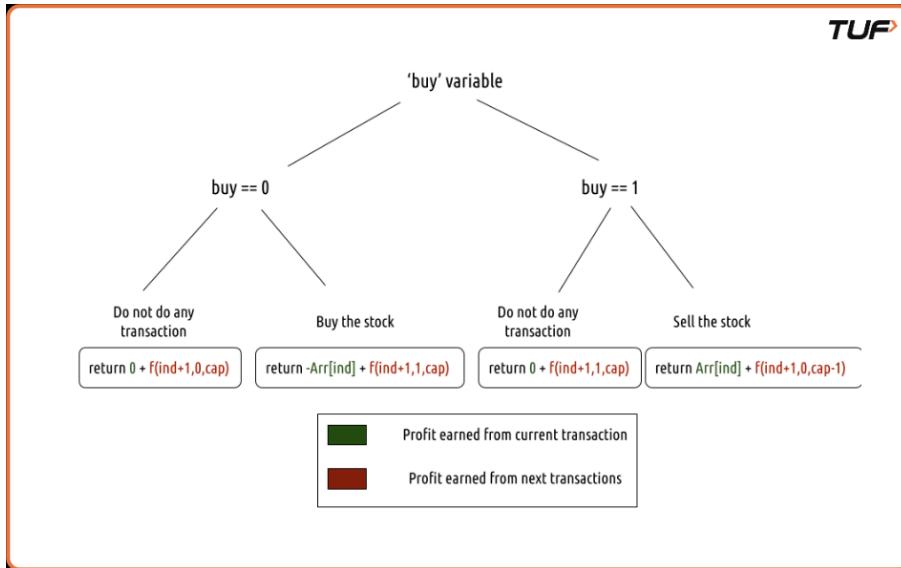
- Whether we can **buy** or must **sell**
- How many transactions are still left

State is defined by:

- `ind` → current day
- `buy` → 0 means we can buy, 1 means we can sell
- `cap` → remaining transactions

## Algorithm

- If we reach the end of days or `cap == 0`, profit is 0
- If `buy == 0`:
  - Skip the day
  - Buy the stock
- If `buy == 1`:
  - Skip the day
  - Sell the stock and reduce transaction count
- Take the maximum of choices
- Store results in a 3D DP array to avoid recomputation



## Code

```

#include <bits/stdc++.h>
using namespace std;

class StockProfit {
public:
 int getAns(vector<int>& Arr, int n, int ind, int buy, int cap,
 vector<vector<vector<int>>& dp) {

 if (ind == n || cap == 0)
 return 0;

 if (dp[ind][buy][cap] != -1)
 return dp[ind][buy][cap];

 int profit;
 if (buy == 0) {
 profit = max(
 getAns(Arr, n, ind + 1, 0, cap, dp),
 -Arr[ind] + getAns(Arr, n, ind + 1, 1, cap, dp)
);
 } else {
 profit = max(
 getAns(Arr, n, ind + 1, 1, cap, dp),

```

```

 Arr[ind] + getAns(Arr, n, ind + 1, 0, cap - 1, dp)
);
}
return dp[ind][buy][cap] = profit;
}

int maxProfit(vector<int>& prices, int n) {
 vector<vector<vector<int>>> dp(
 n, vector<vector<int>>(2, vector<int>(3, -1))
);
 return getAns(prices, n, 0, 0, 2, dp);
}
};

int main() {
 vector<int> prices = {3,3,5,0,0,3,1,4};
 StockProfit obj;
 cout << "The maximum profit that can be generated is "
 << obj.maxProfit(prices, prices.size());
 return 0;
}

```

## Complexity Analysis

- **Time Complexity:**  $O(N * 2 * 3)$   
Each state (day, buy, cap) is computed once.
  - **Space Complexity:**  $O(N * 2 * 3) + O(N)$   
DP table plus recursion stack.
- 

## Approach 2: Tabulation Approach

We convert the memoization into a bottom-up DP.

- Create a DP table  $dp[ind][buy][cap]$

- Base cases are already 0 when:
  - $\text{ind} == n$
  - $\text{cap} == 0$
- Fill table from last day to first day
- For each state, apply the same buy/sell logic
- Final answer is  $\text{dp}[0][0][2]$

## Code

```
#include <bits/stdc++.h>
using namespace std;

class StockProfit {
public:
 int maxProfit(vector<int>& Arr, int n) {
 vector<vector<vector<int>>> dp(
 n + 1, vector<vector<int>>(2, vector<int>(3, 0)))
 ;

 for (int ind = n - 1; ind >= 0; ind--) {
 for (int buy = 0; buy <= 1; buy++) {
 for (int cap = 1; cap <= 2; cap++) {
 if (buy == 0) {
 dp[ind][buy][cap] = max(
 dp[ind + 1][0][cap],
 -Arr[ind] + dp[ind + 1][1][cap]
);
 } else {
 dp[ind][buy][cap] = max(
 dp[ind + 1][1][cap],
 Arr[ind] + dp[ind + 1][0][cap - 1]
);
 }
 }
 }
 }
 }
}
```

```

 }
 }
 return dp[0][0][2];
}
};

int main() {
 vector<int> prices = {3,3,5,0,0,3,1,4};
 StockProfit obj;
 cout << "The maximum profit that can be generated is "
 << obj.maxProfit(prices, prices.size());
 return 0;
}

```

## Complexity Analysis

- **Time Complexity:**  $O(N * 2 * 3)$
  - **Space Complexity:**  $O(N * 2 * 3)$
- 

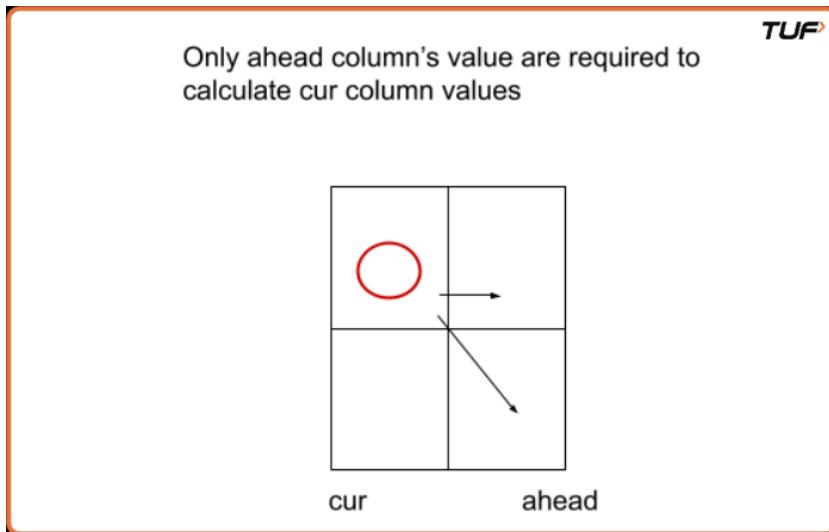
## Approach 3: Space Optimization Approach

We observe that only the **next day** values are needed.

### Algorithm

- Use two 2D arrays:
  - ahead → represents  $dp[ind+1]$
  - cur → represents  $dp[ind]$
- Update cur using ahead
- After each day, copy cur to ahead

- Final answer is ahead[0][2]



## Code

```
#include <bits/stdc++.h>
using namespace std;

class StockProfit {
public:
 int maxProfit(vector<int>& Arr, int n) {
 vector<vector<int>> ahead(2, vector<int>(3, 0));
 vector<vector<int>> cur(2, vector<int>(3, 0));

 for (int ind = n - 1; ind >= 0; ind--) {
 for (int buy = 0; buy <= 1; buy++) {
 for (int cap = 1; cap <= 2; cap++) {
 if (buy == 0) {
 cur[buy][cap] = max(
 ahead[0][cap],
 -Arr[ind] + ahead[1][cap]
);
 } else {
 cur[buy][cap] = max(
 ahead[1][cap],
 Arr[ind] + ahead[0][cap - 1]
);
 }
 }
 }
 }
 return cur[1][2];
 }
};
```

```

 }
 }
}
ahead = cur;
}
return ahead[0][2];
}
};

int main() {
 vector<int> prices = {3,3,5,0,0,3,1,4};
 StockProfit obj;
 cout << "The maximum profit that can be generated is "
 << obj.maxProfit(prices, prices.size());
 return 0;
}

```

## Complexity Analysis

- **Time Complexity:**  $O(N * 2 * 3)$
- **Space Complexity:**  $O(1)$   
Only two  $2 \times 3$  arrays are used.

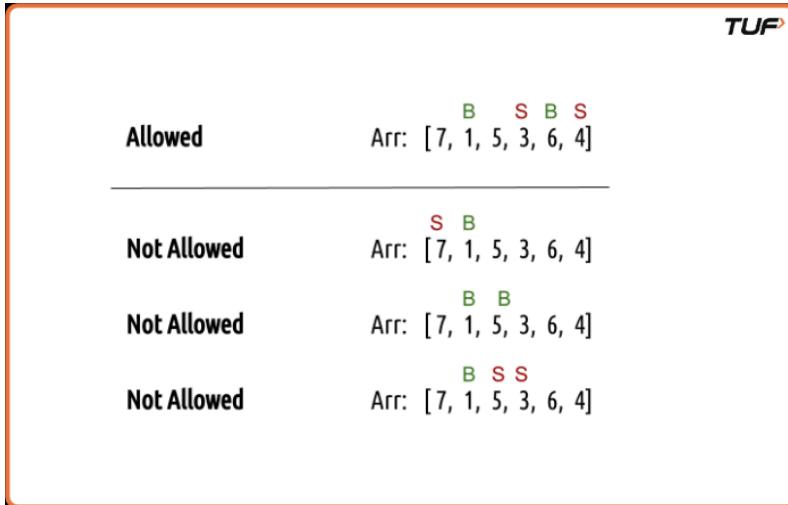
# Buy and Sell Stock – IV (DP-38)

We are given stock prices for n days and allowed to perform **at most K transactions**.

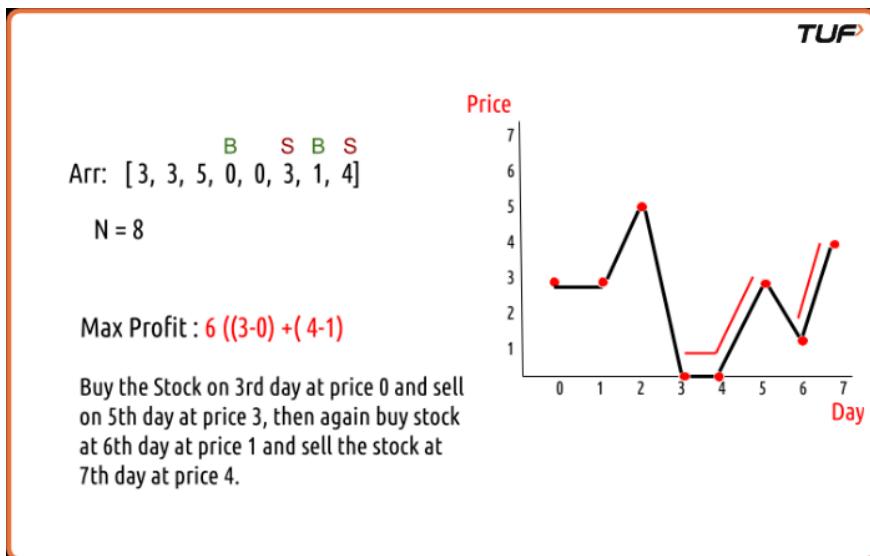
### Rules recap

- One transaction = **buy + sell**

- You must sell before buying again
- You cannot sell before buying
- Goal: **maximize total profit**



This is a **generalization of Stock-III** (where K = 2).



At any day, your decision depends on **three things**:

1. day (ind)
2. buy → are you allowed to buy (0) or must sell (1)
3. cap → how many transactions are still left

So the DP state is:

$dp[ind][buy][cap]$

Meaning:

Maximum profit from day  $ind$  onward,  
given whether we can buy or sell,  
and  $cap$  transactions remaining.

---

## Transitions

### If $buy == 0$ (we can buy)

- Skip the day  $\rightarrow dp[ind+1][0][cap]$
- Buy stock  $\rightarrow -price[ind] + dp[ind+1][1][cap]$

```
dp[ind][0][cap] = max(
 dp[ind+1][0][cap],
 -Arr[ind] + dp[ind+1][1][cap]
)
```

### If $buy == 1$ (we can sell)

- Skip the day  $\rightarrow dp[ind+1][1][cap]$
- Sell stock  $\rightarrow +price[ind] + dp[ind+1][0][cap-1]$

```
dp[ind][1][cap] = max(
 dp[ind+1][1][cap],
 Arr[ind] + dp[ind+1][0][cap-1]
)
```

---

## Base Cases

- If  $\text{ind} == n \rightarrow$  no days left  $\rightarrow$  profit 0
  - If  $\text{cap} == 0 \rightarrow$  no transactions left  $\rightarrow$  profit 0
- 

## 1 Memoization (Top-Down)

### Code

```
#include <bits/stdc++.h>
using namespace std;

class StockBuySell {
public:
 int solve(int ind, int buy, int cap,
 vector<int>& arr,
 vector<vector<vector<int>>>& dp) {

 if (ind == arr.size() || cap == 0)
 return 0;

 if (dp[ind][buy][cap] != -1)
 return dp[ind][buy][cap];

 int profit;
 if (buy == 0) {
 profit = max(
 solve(ind + 1, 0, cap, arr, dp),
 -arr[ind] + solve(ind + 1, 1, cap, arr, dp)
);
 } else {
 profit = max(
 solve(ind + 1, 1, cap, arr, dp),
```

```

 arr[ind] + solve(ind + 1, 0, cap - 1, arr, dp)
);
}

return dp[ind][buy][cap] = profit;
}

int maximumProfit(vector<int>& prices, int k) {
 int n = prices.size();
 vector<vector<vector<int>>> dp(
 n, vector<vector<int>>(2, vector<int>(k + 1, -1))
);
 return solve(0, 0, k, prices, dp);
}
};

int main() {
 vector<int> prices = {3,3,5,0,0,3,1,4};
 int k = 2;

 StockBuySell obj;
 cout << obj.maximumProfit(prices, k);
 return 0;
}

```

## Complexity

- **Time:**  $O(N \times 2 \times K)$
  - **Space:**  $O(N \times 2 \times K) + \text{recursion stack}$
- 

## 2 Tabulation (Bottom-Up)

### Idea

Convert recursion into loops:

- Day from  $n-1 \rightarrow 0$
- buy in  $\{0, 1\}$
- cap from  $1 \rightarrow K$

## Code

```
#include <bits/stdc++.h>
using namespace std;

class StockBuySell {
public:
 int maximumProfit(vector<int>& arr, int k) {
 int n = arr.size();

 vector<vector<vector<int>>> dp(
 n + 1, vector<vector<int>>(2, vector<int>(k + 1, 0))
);

 for (int ind = n - 1; ind >= 0; ind--) {
 for (int buy = 0; buy <= 1; buy++) {
 for (int cap = 1; cap <= k; cap++) {
 if (buy == 0) {
 dp[ind][buy][cap] = max(
 dp[ind + 1][0][cap],
 -arr[ind] + dp[ind + 1][1][cap]
);
 } else {
 dp[ind][buy][cap] = max(
 dp[ind + 1][1][cap],
 arr[ind] + dp[ind + 1][0][cap - 1]
);
 }
 }
 }
 }
 }
}
```

```

 return dp[0][0][k];
 }
};

int main() {
 vector<int> prices = {3,3,5,0,0,3,1,4};
 int k = 2;

 StockBuySell obj;
 cout << obj.maximumProfit(prices, k);
 return 0;
}

```

## Complexity

- **Time:**  $O(N \times 2 \times K)$
  - **Space:**  $O(N \times 2 \times K)$
- 

## 3 Space Optimization (Best)

### Observation

$dp[ind]$  depends only on  $dp[ind+1]$ .

So we only need **two  $2 \times (K+1)$  arrays**:

- ahead  $\rightarrow$  day  $ind+1$
  - cur  $\rightarrow$  day  $ind$
- 

### Code

```
#include <bits/stdc++.h>
```

```

using namespace std;

class StockBuySell {
public:
 int maximumProfit(vector<int>& arr, int k) {
 int n = arr.size();

 vector<vector<int>> ahead(2, vector<int>(k + 1, 0));
 vector<vector<int>> cur(2, vector<int>(k + 1, 0));

 for (int ind = n - 1; ind >= 0; ind--) {
 for (int buy = 0; buy <= 1; buy++) {
 for (int cap = 1; cap <= k; cap++) {
 if (buy == 0) {
 cur[buy][cap] = max(
 ahead[0][cap],
 -arr[ind] + ahead[1][cap]
);
 } else {
 cur[buy][cap] = max(
 ahead[1][cap],
 arr[ind] + ahead[0][cap - 1]
);
 }
 }
 }
 ahead = cur;
 }
 return ahead[0][k];
 }
};

int main() {
 vector<int> prices = {3,3,5,0,0,3,1,4};
 int k = 2;

 StockBuySell obj;
 cout << obj.maximumProfit(prices, k);
}

```

```

 return 0;
}

```

## Complexity

- **Time:**  $O(N \times 2 \times K)$
- **Space:**  $O(2 \times K) \rightarrow O(K)$

# Buy and Sell Stocks With Cooldown | (DP - 39)

We are given an array Arr[] where Arr[i] represents the stock price on day i.

We can buy and sell the stock multiple times but with some rules.

We must buy before selling. After selling, we cannot buy on the very next day because of the cooldown rule.

Our task is to find the maximum profit that can be earned.

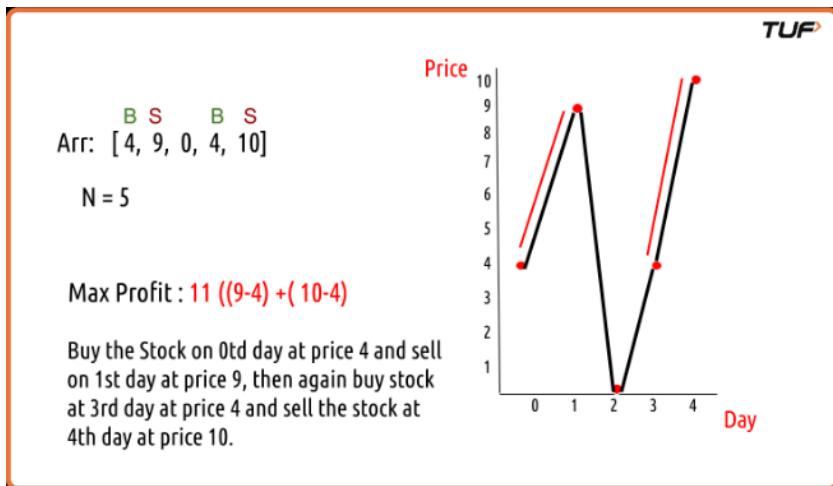
|                    |                                           |                      |
|--------------------|-------------------------------------------|----------------------|
| <b>Allowed</b>     | <b>B S B S</b><br>Arr: [7, 1, 5, 3, 6, 4] | <b>TUF</b>           |
| <b>Not Allowed</b> | <b>S B</b><br>Arr: [7, 1, 5, 3, 6, 4]     |                      |
| <b>Not Allowed</b> | <b>B B</b><br>Arr: [7, 1, 5, 3, 6, 4]     |                      |
| <b>Not Allowed</b> | <b>B S S</b><br>Arr: [7, 1, 5, 3, 6, 4]   |                      |
| <b>Not Allowed</b> | <b>B S B S</b><br>Arr: [7, 1, 5, 3, 6, 4] | [Bought on cooldown] |

Example:

Arr = [4, 9, 0, 4, 10]

If we buy on day 0 (price 4) and sell on day 1 (price 9), profit = 5.

Due to cooldown, we cannot buy on day 2.  
 Then we buy on day 3 (price 4) and sell on day 4 (price 10), profit = 6.  
 Total profit = 11.



## Approach 1: Memoization

### Algorithm

We process the array day by day using recursion.  
 At each day, we track whether we are allowed to buy or sell.

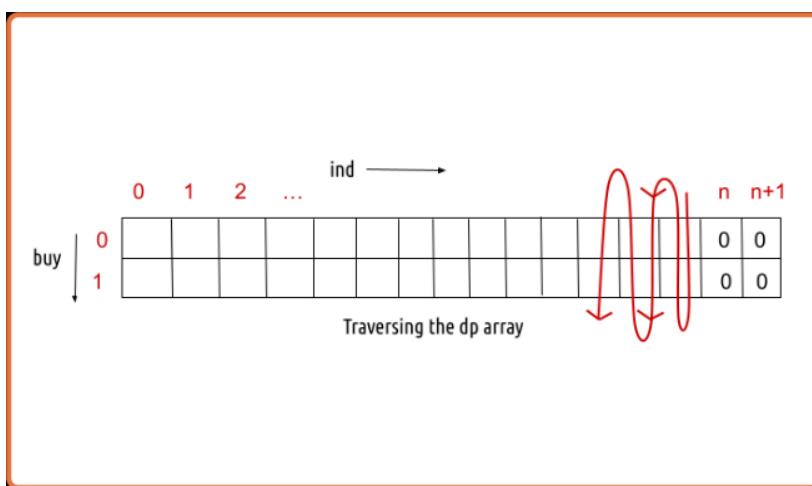
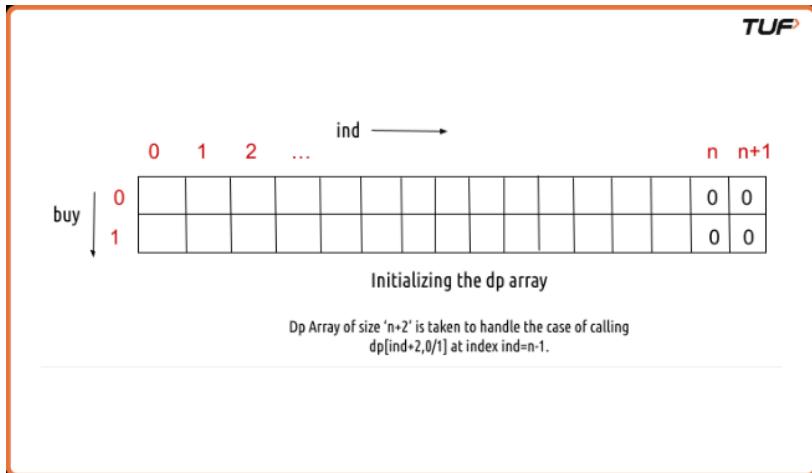
If buying is allowed:

- Option 1: Skip the day.
- Option 2: Buy the stock and move to sell state.

If selling is allowed:

- Option 1: Skip the day.
- Option 2: Sell the stock and move two days ahead because of cooldown.

We use a DP table  $dp[ind][buy]$  to store results so the same state is not recalculated.



## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution{
public:
 int getAns(vector<int> Arr,int ind,int buy,int n,vector<vector<int>> &dp){
 if(ind>=n) return 0;
 if(dp[ind][buy]!=-1) return dp[ind][buy];
 int profit;
 if(buy==0){
 profit=max(getAns(Arr,ind+1,0,n,dp),
 -Arr[ind]+getAns(Arr,ind+1,1,n,dp));
 }
 else{
 profit=-Arr[ind]+getAns(Arr,ind+1,0,n,dp);
 }
 dp[ind][buy]=profit;
 return profit;
 }
};
```

```

 if(buy==1){
 profit=max(getAns(Arr,ind+1,1,n,dp),
 Arr[ind]+getAns(Arr,ind+2,0,n,dp));
 }
 return dp[ind][buy]=profit;
 }

int stockProfit(vector<int> &Arr){
 int n=Arr.size();
 vector<vector<int>> dp(n,vector<int>(2,-1));
 return getAns(Arr,0,0,n,dp);
}
};

int main(){
 vector<int> prices={4,9,0,4,10};
 Solution obj;
 cout<<obj.stockProfit(prices);
 return 0;
}

```

## Complexity Analysis

Time Complexity:  $O(N^2)$

There are  $N$  days and 2 states (buy or sell). Each state is computed once.

Space Complexity:  $O(N^2) + O(N)$

*DP table uses  $N^2$  space and recursion stack can go up to  $N$ .*

---

## Approach 2: Tabulation

### Algorithm

We convert recursion into iteration.

We create a DP table  $dp[ind][buy]$  where  $ind$  goes from last day to first.

Base case: after the last day, profit is 0.  
We fill the table backwards using the same transitions as memoization.  
The final answer is  $dp[0][0]$ .

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution{
public:
int stockProfit(vector<int> &Arr){
 int n=Arr.size();
 vector<vector<int>> dp(n+2,vector<int>(2,0));
 for(int ind=n-1;ind>=0;ind--){
 for(int buy=0;buy<=1;buy++){
 int profit;
 if(buy==0){
 profit=max(dp[ind+1][0],
 -Arr[ind]+dp[ind+1][1]);
 }
 if(buy==1){
 profit=max(dp[ind+1][1],
 Arr[ind]+dp[ind+2][0]);
 }
 dp[ind][buy]=profit;
 }
 }
 return dp[0][0];
}
};

int main(){
 vector<int> prices={4, 9, 0, 4, 10};
 Solution obj;
 cout<<obj.stockProfit(prices);
 return 0;
}
```

## Complexity Analysis

Time Complexity:  $O(N^2)$

Two nested loops over days and buy/sell states.

Space Complexity:  $O(N^2)$

Only the DP table is used, no recursion stack.

---

## Approach 3: Space Optimization

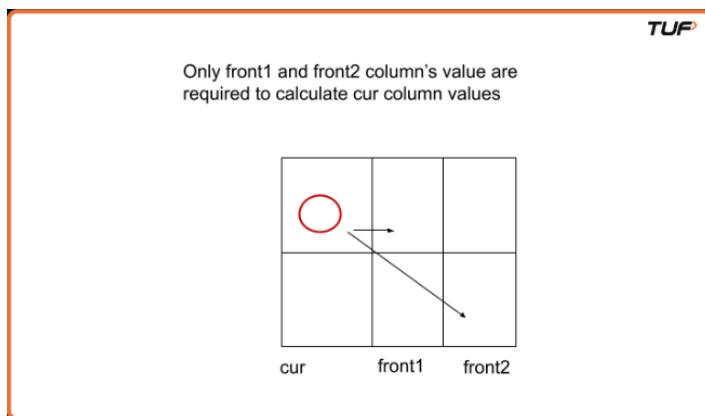
### Algorithm

For each day, we only need results from the next one or two days.

So instead of a full DP table, we keep only three arrays:

- cur for current day
- front1 for next day
- front2 for day after next

We iterate from last day to first and update these arrays.



### Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution{
```

```

public:
int stockProfit(vector<int> &Arr){
 int n=Arr.size();
 vector<int> cur(2,0),front1(2,0),front2(2,0);
 for(int ind=n-1;ind>=0;ind--){
 for(int buy=0;buy<=1;buy++){
 int profit;
 if(buy==0){
 profit=max(front1[0],
 -Arr[ind]+front1[1]);
 }
 if(buy==1){
 profit=max(front1[1],
 Arr[ind]+front2[0]);
 }
 cur[buy]=profit;
 }
 front2=front1;
 front1=cur;
 }
 return cur[0];
}

int main(){
 vector<int> prices={4, 9, 0, 4, 10};
 Solution obj;
 cout<<obj.stockProfit(prices);
 return 0;
}

```

## Complexity Analysis

Time Complexity:  $O(N^2)$

We process each day with two states.

Space Complexity:  $O(1)$

Only three arrays of size 2 are used.

# Buy and Sell Stocks With Transaction Fees | (DP - 40)

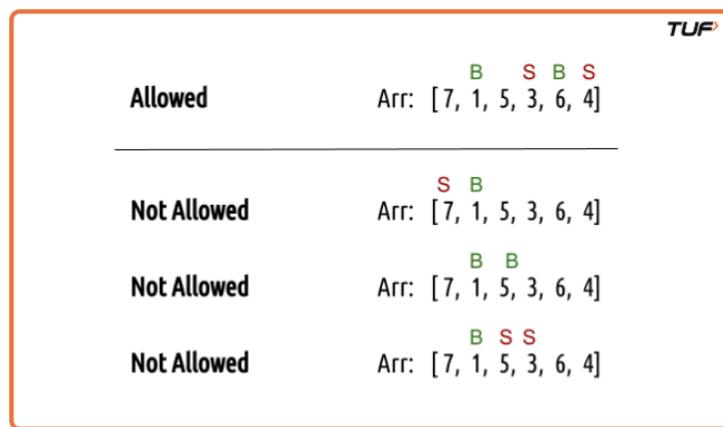
We are given an array Arr[] where Arr[i] represents the stock price on day i.

We can buy and sell stocks multiple times, but we must follow these rules.

We must buy before selling. After selling, we are allowed to buy again.

Each complete transaction (buy + sell) has a fixed transaction fee that is deducted when we sell.

Our goal is to find the maximum profit possible.



Example:

Arr = [1,3,2,8,4,9], Fee = 2

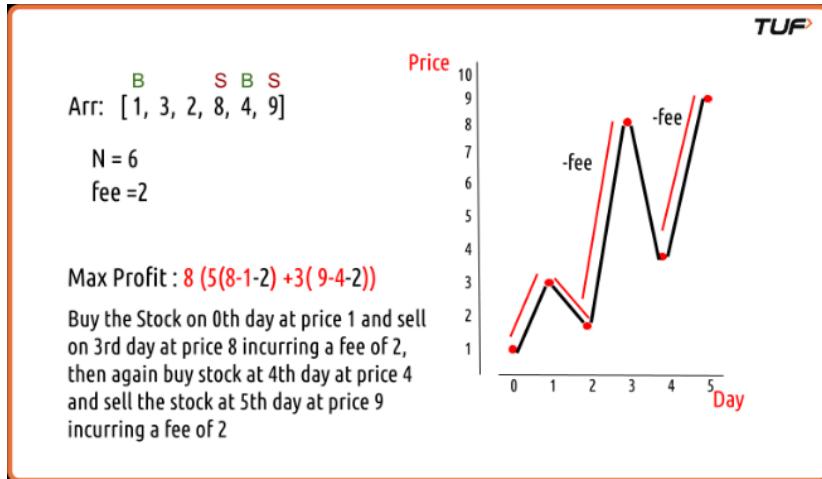
Buy on day 0 at price 1 and sell on day 3 at price 8.

Profit =  $8 - 1 - 2 = 5$

Then buy on day 4 at price 4 and sell on day 5 at price 9.

Profit =  $9 - 4 - 2 = 3$

Total profit = 8



## Approach 1: Memoization

### Algorithm

We solve the problem using recursion and dynamic programming.

At each day, we decide based on whether we are allowed to buy or need to sell.

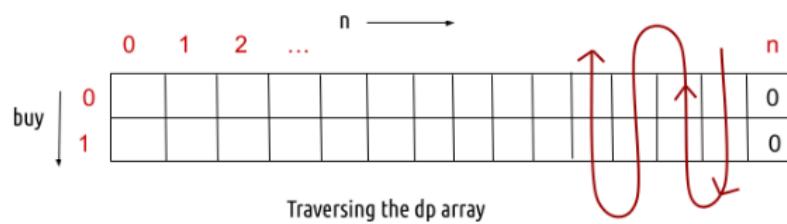
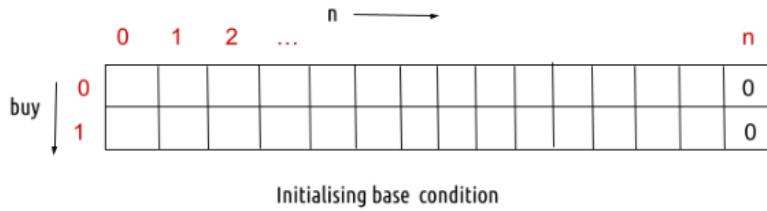
If  $\text{buy} = 0$  (allowed to buy):

- Skip the day and move to the next day.
- Buy the stock today and move to sell state.

If  $\text{buy} = 1$  (allowed to sell):

- Skip the day and continue holding.
- Sell the stock today, subtract the transaction fee, and move to buy state.

We store results in a DP table  $\text{dp}[\text{ind}][\text{buy}]$  to avoid recomputation.



## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution{
public:
 int getAns(vector<int> &Arr, int ind, int buy, int n, int fee, vector<vector<int>> &dp){
 if(ind==n) return 0;
 if(dp[ind][buy]!=-1) return dp[ind][buy];
 int profit;
 if(buy==0){
 profit = max(-Arr[ind] + getAns(
```

```

 profit=max(getAns(Arr,ind+1,0,n,fee,dp),
 -Arr[ind]+getAns(Arr,ind+1,1,n,fee,dp));
 }
 if(buy==1){
 profit=max(getAns(Arr,ind+1,1,n,fee,dp),
 Arr[ind]-fee+getAns(Arr,ind+1,0,n,fee,dp));
 }
 return dp[ind][buy]=profit;
}

int maximumProfit(int n,int fee,vector<int> &Arr){
 vector<vector<int>> dp(n,vector<int>(2,-1));
 return getAns(Arr,0,0,n,fee,dp);
}
;

int main(){
 vector<int> prices={1,3,2,8,4,9};
 int fee=2;
 Solution obj;
 cout<<obj.maximumProfit(prices.size(),fee,prices);
 return 0;
}

```

## Complexity Analysis

Time Complexity:  $O(N^2)$

There are  $N$  days and 2 states (buy or sell). Each state is computed once.

Space Complexity:  $O(N^2) + O(N)$

*DP table* uses  $N^2$  space and recursion stack takes up to  $N$ .

---

## Approach 2: Tabulation

### Algorithm

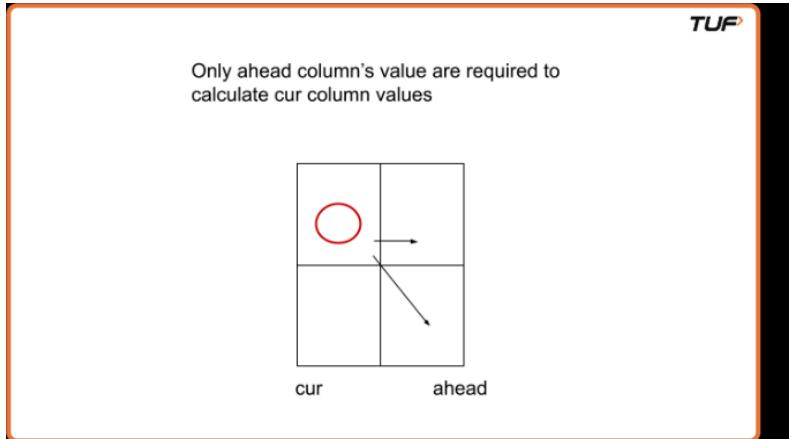
We convert recursion into an iterative DP approach.

Create a 2D DP table  $dp[ind][buy]$ .

Base case: when  $ind = n$ , profit is 0.

Fill the table from the last day to the first using the same transitions as memoization.

The answer is  $dp[0][0]$ , meaning starting on day 0 with permission to buy.



## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution{
public:
int maximumProfit(int n,int fee,vector<int> &Arr){
 vector<vector<int>> dp(n+1,vector<int>(2,0));
 for(int ind=n-1;ind>=0;ind--){
 for(int buy=0;buy<=1;buy++){
 int profit;
 if(buy==0){
 profit=max(dp[ind+1][0],
 -Arr[ind]+dp[ind+1][1]);
 }
 if(buy==1){
 profit=max(dp[ind+1][1],
 Arr[ind]-fee+dp[ind+1][0]);
 }
 dp[ind][buy]=profit;
 }
 }
 return dp[0][0];
}
```

```

 }
};

int main(){
 vector<int> prices={1,3,2,8,4,9};
 int fee=2;
 Solution obj;
 cout<<obj.maximumProfit(prices.size(),fee,prices);
 return 0;
}

```

## Complexity Analysis

Time Complexity:  $O(N^2)$

Two nested loops over days and buy/sell states.

Space Complexity:  $O(N^2)$

Only the DP table is used, no recursion stack.

---

## Approach 3: Space Optimization

### Algorithm

For each day, we only need the results from the next day.

So instead of a full DP table, we use two arrays:

- ahead: stores next day results
- cur: stores current day results

We iterate from last day to first and update values accordingly.

The final answer is cur[0].

### Code

```

#include <bits/stdc++.h>
using namespace std;

class Solution{

```

```

public:
int maximumProfit(int n,int fee,vector<int> &Arr){
 vector<long> ahead(2,0),cur(2,0);
 for(int ind=n-1;ind>=0;ind--){
 for(int buy=0;buy<=1;buy++){
 long profit;
 if(buy==0){
 profit=max(ahead[0],
 -Arr[ind]+ahead[1]);
 }
 if(buy==1){
 profit=max(ahead[1],
 Arr[ind]-fee+ahead[0]);
 }
 cur[buy]=profit;
 }
 ahead=cur;
 }
 return cur[0];
}

int main(){
 vector<int> prices={1,3,2,8,4,9};
 int fee=2;
 Solution obj;
 cout<<obj.maximumProfit(prices.size(),fee,prices);
 return 0;
}

```

## Complexity Analysis

Time Complexity:  $O(N^2)$   
 Each day processes two states.

Space Complexity:  $O(1)$   
 Only two arrays of size 2 are used.

# Longest Increasing Subsequence |

## (DP - 41)

We are given an integer array `nums`.

We need to find the length of the longest subsequence such that the elements are in strictly increasing order.

A subsequence is formed by deleting some elements without changing the order of the remaining elements.

Example:

`nums` = [10, 9, 2, 5, 3, 7, 101, 18]

One longest increasing subsequence is [2, 3, 7, 101].

Its length is 4, which is the answer.

---

## Approach

### Algorithm

We use recursion with memoization to try all possible subsequences.

We define a function that works with two variables:

- `ind`: current index we are at
- `prev_ind`: index of the previously taken element in the subsequence  
If no element has been taken yet, `prev_ind` = -1.

At every index, we have two choices:

- Do not take the current element and move to the next index.
- Take the current element if it is strictly greater than the previously taken element.

To store overlapping subproblems, we use a 2D DP array:

- `dp[ind][prev_ind + 1]`  
We add 1 to `prev_ind` because `prev_ind` can be -1, and array indices cannot be negative.

Base case:

- If we are at the last index, we check if we can take it or not and return accordingly.

The answer is the maximum length obtained by either taking or not taking the current element.

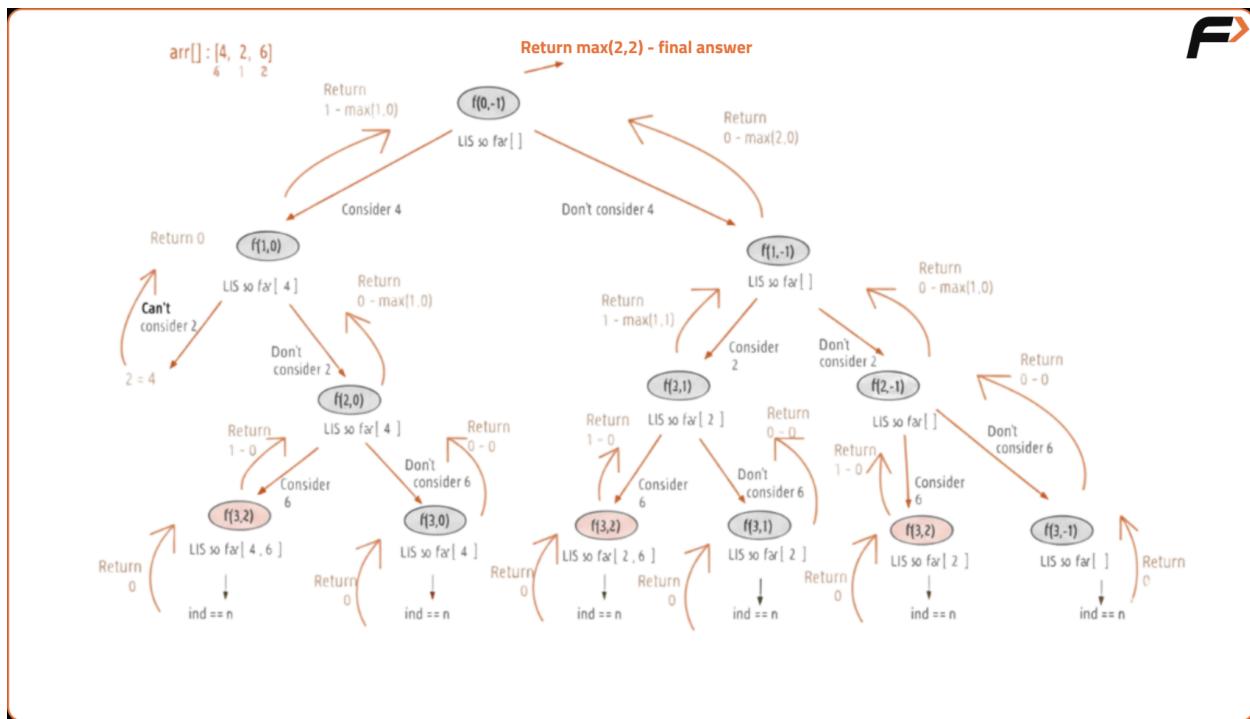
Example dry run (`nums = [10, 9, 2, 5, 3, 7, 101, 18]`):

We start at index 0 with `prev_ind = -1`.

We try skipping 10 and also try taking 10.

This process continues for every index, and the DP table ensures we do not recompute states.

The maximum length found is 4.



## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
private:
 int func(int i,int prevInd,vector<int> &arr,vector<vector<int>> &dp){
 if(i==arr.size()-1){
 if(prevInd==-1 || arr[prevInd]<arr[i]) return 1;
 return 0;
 }
 if(dp[i][prevInd+1]!=-1) return dp[i][prevInd+1];
 int notTake=func(i+1,prevInd,arr,dp);
 int take=0;
 if(prevInd==-1 || arr[i]>arr[prevInd])
 take=1+func(i+1,i,arr,dp);
 return dp[i][prevInd+1]=max(take,notTake);
 }
}
```

```

public:
 int LIS(vector<int>& nums){
 int n=nums.size();
 vector<vector<int>> dp(n,vector<int>(n+1,-1));
 return func(0,-1,nums,dp);
 }
};

int main(){
 vector<int> nums={10,9,2,5,3,7,101,18};
 Solution sol;
 cout<<"The length of the LIS for the given array is: "<<sol.LIS(nums);
 return 0;
}

```

## Complexity Analysis

Time Complexity:  $O(n^2)$

There are  $n$  possible values for  $ind$  and  $n$  possible values for  $prev\_ind$ . Each state is computed once.

Space Complexity:  $O(n^2)$

We use a 2D DP array of size  $n \times (n+1)$ .

The recursion stack can go up to  $O(n)$ , but it is dominated by the DP array.

# Printing Longest Increasing Subsequence | (DP - 42)

We are given an array  $arr$  of size  $n$ .

Our task is to return the **Longest Increasing Subsequence (LIS)** such that:

- The subsequence is strictly increasing.

- If multiple LIS are possible, we return the one that is **index-wise lexicographically smallest**, meaning the elements are chosen as early as possible from the original array.

A subsequence is formed by deleting elements without changing the order.

Example:

$\text{arr} = [1,3,2,4,6,5]$

Possible LIS are  $[1,3,4,6]$  and  $[1,2,4,6]$ .

Since 3 appears earlier than 2 in the array,  $[1,3,4,6]$  is chosen.

---

## Approach

### Algorithm

We use Dynamic Programming to first compute the length of LIS ending at each index.

Steps:

1. Create a DP array  $\text{dp}[]$  where  $\text{dp}[i]$  stores the length of LIS ending at index  $i$ .
2. Create another array  $\text{prev}[]$  to store the previous index of the element used in LIS for reconstruction.
3. Initialize  $\text{dp}[i] = 1$  for all  $i$  because every element alone is an LIS of length 1.
4. For every index  $i$ , check all previous indices  $j < i$ :
  - If  $\text{arr}[j] < \text{arr}[i]$  and  $\text{dp}[j] + 1 > \text{dp}[i]$ , update  $\text{dp}[i]$  and set  $\text{prev}[i] = j$ .
  - This ensures we always pick the earliest valid sequence, giving lexicographically smallest LIS.
5. Find the index with the maximum value in  $\text{dp}[]$ .
6. Reconstruct the LIS by following  $\text{prev}[]$  pointers backward.
7. Reverse the collected sequence to get the correct LIS order.

Example dry run ( $\text{arr} = [10,22,9,33,21,50,41,60,80]$ ):

The dp array identifies LIS length as 6.

Using prev[], we trace back elements like 80 → 60 → 50 → 33 → 22 → 10.  
After reversing, we get [10,22,33,50,60,80].

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 vector<int> longestIncreasingSubsequence(vector<int> &nums) {
 int n = nums.size();
 vector<int> dp(n, 1), prev(n, -1);

 for(int i=1;i<n;i++){
 for(int j=0;j<i;j++){
 if(nums[j]<nums[i] && dp[j]+1>dp[i]){
 dp[i]=dp[j]+1;
 prev[i]=j;
 }
 }
 }

 int maxLen=0,lastIndex=0;
 for(int i=0;i<n;i++){
 if(dp[i]>maxLen){
 maxLen=dp[i];
 lastIndex=i;
 }
 }

 vector<int> lis;
 while(lastIndex!=-1){
 lis.push_back(nums[lastIndex]);
 lastIndex=prev[lastIndex];
 }

 reverse(lis.begin(),lis.end());
 return lis;
 }
}
```

```

};

int main(){
 vector<int> nums={10,22,9,33,21,50,41,60,80};
 Solution sol;
 vector<int> lis=sol.longestIncreasingSubsequence(nums);
 for(int x:lis) cout<<x<<" ";
 return 0;
}

```

## Complexity Analysis

Time Complexity:  $O(N^2)$

Two nested loops are used to compute the dp and prev arrays.

Space Complexity:  $O(N)$

We use dp[] and prev[] arrays of size N to store LIS length and reconstruction path.

# Longest Increasing Subsequence | Binary Search | (DP - 43)

We are given an integer array nums.

Our task is to find the length of the longest subsequence such that all elements are in strictly increasing order.

A subsequence is formed by removing some elements without changing the order of the remaining elements.

Example:

nums = [10, 9, 2, 5, 3, 7, 101, 18]

One longest increasing subsequence is [2, 3, 7, 101].

The length of this subsequence is 4.

# Approach

## Algorithm

We solve this problem using a greedy approach combined with binary search.

We maintain a temporary array temp.

temp does not store the actual LIS, but it stores the smallest possible ending value of an increasing subsequence of a given length.

Steps:

1. Initialize an empty array temp.
2. Insert the first element of nums into temp.
3. Traverse the array from the second element onward:
  - o If the current element is greater than the last element of temp, append it to temp.
  - o Otherwise, find the index of the smallest element in temp that is greater than or equal to the current element using binary search (`lower_bound`), and replace that element.
4. At the end, the size of temp gives the length of the LIS.

Dry run example:

nums = [1, 5, 7, 2, 3]

- Start: temp = [1]
- $5 > 1 \rightarrow$  temp = [1, 5]
- $7 > 5 \rightarrow$  temp = [1, 5, 7]
- $2 \leq 7 \rightarrow$  replace 5  $\rightarrow$  temp = [1, 2, 7]
- $3 \leq 7 \rightarrow$  replace 7  $\rightarrow$  temp = [1, 2, 3]

Length of temp = 3, which is the LIS length.

Binary search helps keep temp minimal, allowing better chances to build longer subsequences later.

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int LIS(vector<int>& nums) {
 int n = nums.size();
 vector<int> temp;
```

```

temp.push_back(nums[0]);
for(int i=1;i<n;i++){
 if(nums[i]>temp.back()){
 temp.push_back(nums[i]);
 } else {
 int ind = lower_bound(temp.begin(), temp.end(), nums[i]) - temp.begin();
 temp[ind] = nums[i];
 }
}
return temp.size();
};

int main(){
vector<int> nums={10,9,2,5,3,7,101,18};
Solution sol;
cout<<"The length of the LIS for the given array is: "<<sol.LIS(nums);
return 0;
}

```

## Complexity Analysis

Time Complexity:  $O(N \log N)$

For each element, we perform a binary search on the temp array, which takes  $\log N$  time.

Space Complexity:  $O(N)$

The temp array can grow up to size  $N$  in the worst case.

# Longest Divisible Subset | (DP - 44)

We are given an array  $\text{nums}$  of positive integers.

We need to find the largest subset such that for every pair  $(a, b)$  in the subset, either  $a \% b == 0$  or  $b \% a == 0$  holds true.

If multiple valid subsets exist, we can return any one of them.

Example:

nums = [3, 5, 10, 20]

One valid largest divisible subset is [5, 10, 20] because 10 is divisible by 5 and 20 is divisible by 10.

---

## Approach

### Algorithm

We solve this problem using Dynamic Programming with reconstruction.

1. First, sort the array in ascending order.  
This helps because if  $\text{nums}[i]$  is divisible by  $\text{nums}[j]$  and  $i > j$ , then  $\text{nums}[i]$  can extend the divisible subset ending at  $j$ .
2. Create two arrays:
  - o  $\text{dp}[i]$ : length of the longest divisible subset ending at index  $i$ .
  - o  $\text{parent}[i]$ : previous index used to form the subset ending at  $i$ .
3. Initialize:
  - o  $\text{dp}[i] = 1$  for all  $i$ , since each element alone is a valid subset.
  - o  $\text{parent}[i] = i$  initially.
4. For each index  $i$ , check all previous indices  $\text{prevInd} < i$ :
  - o If  $\text{nums}[i] \% \text{nums}[\text{prevInd}] == 0$  and  $\text{dp}[\text{prevInd}] + 1 > \text{dp}[i]$ , update  $\text{dp}[i]$  and set  $\text{parent}[i] = \text{prevInd}$ .
5. While filling  $\text{dp}$ , keep track of the index where the maximum subset length ends ( $\text{lastIndex}$ ).
6. Backtrack from  $\text{lastIndex}$  using the  $\text{parent}$  array to collect the elements of the subset.

Example dry run (nums = [16, 8, 2, 4, 32]):

After sorting → [2, 4, 8, 16, 32]

Each element is divisible by the previous one, so  $\text{dp}$  becomes [1,2,3,4,5].

Backtracking gives the subset [2, 4, 8, 16, 32].

### Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 vector<int> largestDivisibleSubset(vector<int> nums) {
 int n = nums.size();
 sort(nums.begin(), nums.end());
```

```

vector<int> dp(n,1), parent(n);
int maxLen = 0, lastIndex = 0;

for(int i=0;i<n;i++){
 parent[i] = i;
 for(int prevInd=0;prevInd<i;prevInd++){
 if(nums[i] % nums[prevInd] == 0 && dp[prevInd] + 1 > dp[i]){
 dp[i] = dp[prevInd] + 1;
 parent[i] = prevInd;
 }
 }
 if(dp[i] > maxLen){
 maxLen = dp[i];
 lastIndex = i;
 }
}

vector<int> ans;
int i = lastIndex;
while(parent[i] != i){
 ans.push_back(nums[i]);
 i = parent[i];
}
ans.push_back(nums[i]);

return ans;
};

int main(){
 vector<int> nums = {3,5,10,20};
 Solution sol;
 vector<int> ans = sol.largestDivisibleSubset(nums);
 for(int x: ans) cout<<x<<" ";
 return 0;
}

```

## Complexity Analysis

Time Complexity:  $O(n^2)$

We use two nested loops to fill the dp and parent arrays.

Space Complexity:  $O(n)$

We use dp and parent arrays of size  $n$  to store subset length and reconstruction path.

# Longest String Chain | (DP - 45)

We are given an array of strings `words[]`.

We need to find the length of the longest string chain.

A string chain follows these rules:

- Each next word is formed by inserting exactly one character into the previous word.
- The order of characters must remain the same.
- The first word can be any word from the array.

Example:

`words = ["a", "ab", "abc", "abcd", "abcde"]`

Each word is formed by adding one character to the previous word.

So the longest chain length is 5.

---

## Approach

### Algorithm

We use Dynamic Programming to solve this problem.

1. Sort all words based on their length in ascending order.

This ensures that when we process a word, all possible shorter predecessor words are already processed.

2. Create a DP array  $dp[]$  where:

- $dp[i]$  stores the length of the longest string chain ending at  $\text{words}[i]$ .
- Initially,  $dp[i] = 1$  because every word alone forms a chain of length 1.

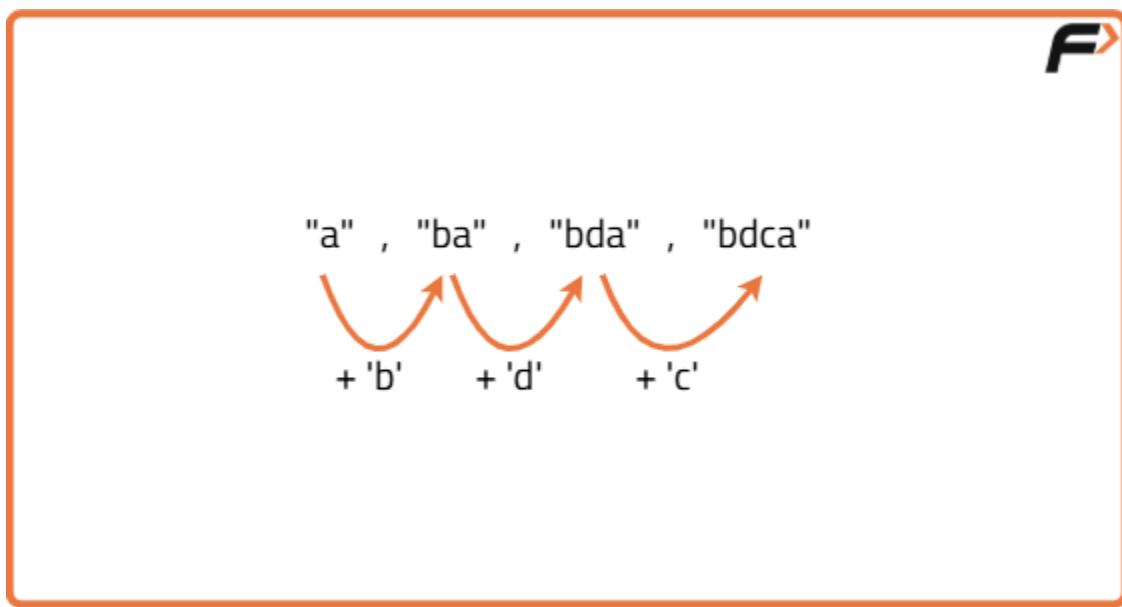
3. For each word at index  $i$ , check all previous words  $j < i$ :

- If  $\text{words}[j]$  can form  $\text{words}[i]$  by inserting exactly one character, then  $\text{words}[i]$  can extend the chain ending at  $j$ .
- If  $dp[j] + 1 > dp[i]$ , update  $dp[i]$ .

4. To check if a word  $t$  is a valid predecessor of  $s$ :

- Length of  $s$  must be exactly one more than  $t$ .
- Traverse both strings using two pointers and allow exactly one extra character in  $s$ .

5. Keep track of the maximum value in  $dp[]$ , which is the length of the longest string chain.



Example dry run ( $\text{words} = ["d", "do", "dot", "dots"]$ ):

After sorting  $\rightarrow ["d", "do", "dot", "dots"]$

Each next word is formed by adding one character.

$dp$  becomes [1,2,3,4].

Maximum chain length = 4.

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int longestStringChain(vector<string>& words) {
 int n = words.size();
 sort(words.begin(), words.end(), compare);
 vector<int> dp(n, 1);
 int maxLen = 0;

 for(int i=0;i<n;i++){
 for(int j=0;j<i;j++){
 if(checkPossible(words[i],words[j]) && dp[i]<dp[j]+1){
 dp[i]=dp[j]+1;
 }
 }
 if(dp[i]>maxLen) maxLen=dp[i];
 }
 return maxLen;
 }

private:
 static bool compare(string &s,string &t){
 return s.size()<t.size();
 }

 bool checkPossible(string &s,string &t){
 if(s.size()!=t.size()+1) return false;
 int i=0,j=0;
 while(i<s.size()){
 if(j<t.size() && s[i]==t[j]){
 i++;j++;
 } else {
 i++;
 }
 }
 }
}
```

```

 return j==t.size();
 }
};

int main(){
 vector<string> words={"a", "ab", "abc", "abcd", "abcde"};
 Solution sol;
 cout<<"The length of the Longest String Chain is:
"<<sol.longestStringChain(words);
 return 0;
}

```

## Complexity Analysis

Time Complexity:  $O(n^2 * m)$

For each word, we compare it with all previous words, and each comparison takes up to  $m$  time where  $m$  is the average word length.

Space Complexity:  $O(n)$

We use a DP array of size  $n$  to store the longest chain length for each word.

# Longest Bitonic Subsequence | (DP - 46)

We are given an array arr of size  $n$ .

We need to find the length of the longest **bitonic subsequence**.

A bitonic subsequence is a sequence that:

- First strictly increases
- Then strictly decreases

The elements do not need to be contiguous, but their order must be preserved.

Example:

arr = [5,1,4,2,3,6,8,7]

One longest bitonic subsequence is [1,2,3,6,8,7].

Its length is 6.

arr []: 

|   |    |   |    |   |   |   |   |
|---|----|---|----|---|---|---|---|
| 1 | 11 | 2 | 10 | 4 | 5 | 2 | 1 |
|---|----|---|----|---|---|---|---|

dp1 []: 

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 2 | 3 | 3 | 4 | 2 | 1 |
|---|---|---|---|---|---|---|---|

dp[i] gives the length of the LIS from index 0 to index i  
Here, dp[5] gives us length of LIS (4)

arr []: 

|   |    |   |    |   |   |   |   |
|---|----|---|----|---|---|---|---|
| 1 | 11 | 2 | 10 | 4 | 5 | 2 | 1 |
|---|----|---|----|---|---|---|---|

Index i

i = 3, arr[i] = 10

LIS from index=0 to index= i

i = 3, arr[i] = 10

LDS from index=i to index= n-1  
or  
LIS from index=n-1 to index= i

dp1 gives the LIS from index 0 to index i

arr []: 

|   |    |   |    |   |   |   |   |
|---|----|---|----|---|---|---|---|
| 1 | 11 | 2 | 10 | 4 | 5 | 2 | 1 |
|---|----|---|----|---|---|---|---|

dp1 []: 

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 2 | 3 | 3 | 4 | 2 | 1 |
|---|---|---|---|---|---|---|---|

dp2 gives the LIS from index n-1 to index i

arr []: 

|   |    |   |    |   |   |   |   |
|---|----|---|----|---|---|---|---|
| 1 | 11 | 2 | 10 | 4 | 5 | 2 | 1 |
|---|----|---|----|---|---|---|---|

dp2 []: 

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 1 | 5 | 2 | 4 | 3 | 3 | 2 | 1 |
|---|---|---|---|---|---|---|---|

ans[i] = dp1[i] + dp2[i] - 1

arr []: 

|   |    |   |    |   |   |   |   |
|---|----|---|----|---|---|---|---|
| 1 | 11 | 2 | 10 | 4 | 5 | 2 | 1 |
|---|----|---|----|---|---|---|---|

dp1 []: 

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 2 | 3 | 3 | 4 | 2 | 1 |
|---|---|---|---|---|---|---|---|

dp2 []: 

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 1 | 5 | 2 | 4 | 3 | 3 | 2 | 1 |
|---|---|---|---|---|---|---|---|

ans []: 

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 1 | 6 | 3 | 6 | 5 | 6 | 3 | 1 |
|---|---|---|---|---|---|---|---|

---

# Approach

## Algorithm

We solve this problem using Dynamic Programming by combining two sequences.

### 1. Longest Increasing Subsequence (LIS)

For every index i, compute the length of the longest increasing subsequence ending at i.

- o Initialize LIS\_dp[i] = 1 for all i.
- o For each i, check all previous indices prev < i.
- o If arr[prev] < arr[i], update LIS\_dp[i] = max(LIS\_dp[i], LIS\_dp[prev] + 1).

### 2. Longest Decreasing Subsequence (LDS)

For every index i, compute the length of the longest decreasing subsequence starting at i.

- o Initialize LDS\_dp[i] = 1 for all i.
- o Traverse from right to left.
- o For each i, check all next indices prev > i.
- o If arr[prev] < arr[i], update LDS\_dp[i] = max(LDS\_dp[i], LDS\_dp[prev] + 1).

### 3. Combine LIS and LDS

For each index i, the length of the bitonic subsequence with peak at i is:

$$\text{LIS}_{\text{dp}}[i] + \text{LDS}_{\text{dp}}[i] - 1$$

We subtract 1 because arr[i] is counted twice.

### 4. Track the maximum value obtained from all indices.

Example dry run (arr = [10,20,30,40,50,40,30,20]):

LIS\_dp = [1,2,3,4,5,4,3,2]

LDS\_dp = [1,2,3,4,5,4,3,2]

Maximum bitonic length =  $5 + 5 - 1 = 9$ ?

But correct alignment gives full array length = 8.

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int LongestBitonicSequence(vector<int>& arr) {
 int n = arr.size();
 vector<int> LIS_dp(n,1), LDS_dp(n,1);
 int maxLen = 0;
```

```

// LIS calculation
for(int i=0;i<n;i++){
 for(int prev=0;prev<i;prev++){
 if(arr[prev] < arr[i] && LIS_dp[i] < LIS_dp[prev] + 1){
 LIS_dp[i] = LIS_dp[prev] + 1;
 }
 }
}

// LDS calculation
for(int i=n-1;i>=0;i--){
 for(int prev=n-1;prev>i;prev--){
 if(arr[prev] < arr[i] && LDS_dp[i] < LDS_dp[prev] + 1){
 LDS_dp[i] = LDS_dp[prev] + 1;
 }
 }
 maxLen = max(maxLen, LIS_dp[i] + LDS_dp[i] - 1);
}
}

return maxLen;
}
};

int main(){
vector<int> arr = {5,1,4,2,3,6,8,7};
Solution sol;
cout << "The length of the Longest Bitonic Sequence is: "
 << sol.LongestBitonicSequence(arr);
return 0;
}

```

## Complexity Analysis

Time Complexity:  $O(n^2)$

We use two nested loops to compute LIS and LDS.

Space Complexity:  $O(n)$

We use two arrays of size n to store LIS and LDS values.

# Number of Longest Increasing Subsequences | (DP - 47)

We are given an integer array `nums`.

Our task is to find how many **Longest Increasing Subsequences (LIS)** exist in the array.

A Longest Increasing Subsequence is a subsequence where:

- Elements are strictly increasing.
- The length of the subsequence is maximum possible.

Example:

`nums` = [1,3,5,4,7]

The LIS length is 4.

Two LIS exist: [1,3,4,7] and [1,3,5,7].

So the answer is 2.

F

Array : [ 1, 3, 5, 4, 7 ]

The length of the longest increasing subsequence is 5

The count of the longest increasing subsequence with length 4 is 2

( Answer )

Longest Increasing Subsequence(s) :

[ 1, 3, 4, 7 ]

[ 1, 3, 4, 7 ]

---

## Approach

### Algorithm

In the normal LIS problem, we only track the length of LIS ending at each index. Here, we also need to count how many such LIS end at each index.

Steps:

1. Create two arrays:
    - o  $dp[i]$ : length of LIS ending at index  $i$ .
    - o  $ct[i]$ : number of LIS of length  $dp[i]$  ending at index  $i$ .
  2. Initialize:
    - o  $dp[i] = 1$  for all  $i$ , because each element alone forms an LIS of length 1.
    - o  $ct[i] = 1$  for all  $i$ , because there is one such subsequence.
  3. For each index  $i$ , check all previous indices  $j < i$ :
    - o If  $\text{nums}[j] < \text{nums}[i]$  and  $dp[j] + 1 > dp[i]$ :
      - Update  $dp[i] = dp[j] + 1$ .
      - Set  $ct[i] = ct[j]$  because we found a longer LIS.
    - o Else if  $\text{nums}[j] < \text{nums}[i]$  and  $dp[j] + 1 == dp[i]$ :
      - Add  $ct[j]$  to  $ct[i]$ , because another LIS of same length is found.
  4. Keep track of the maximum LIS length found.
  5. At the end, sum  $ct[i]$  for all indices where  $dp[i]$  equals the maximum length.

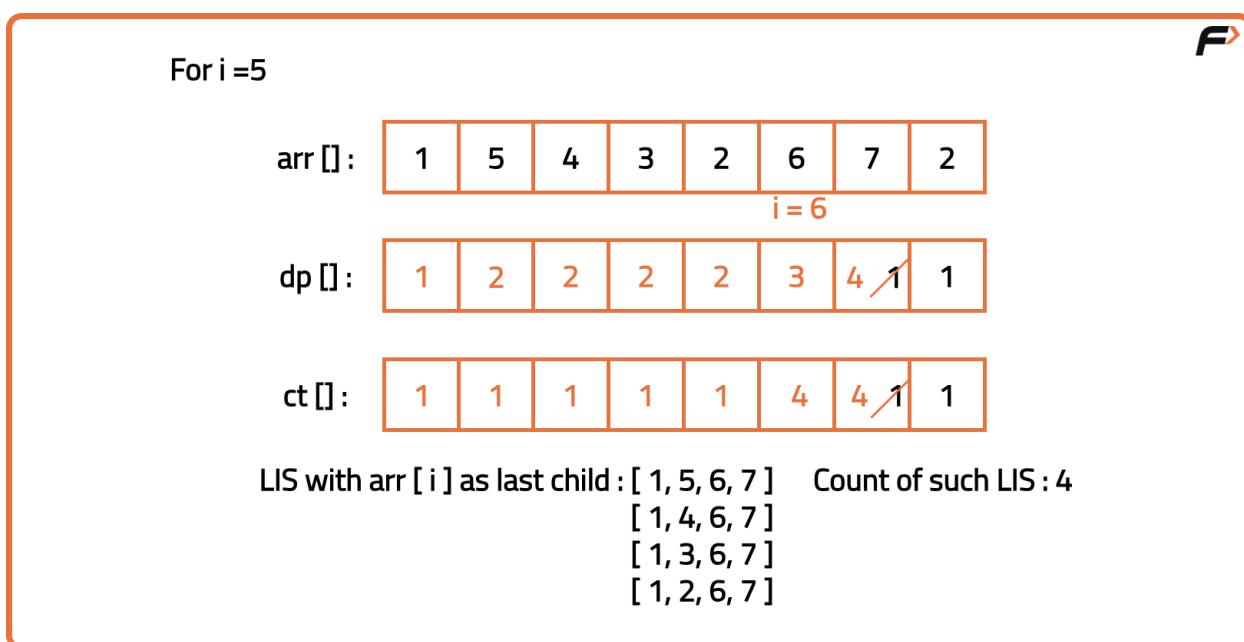
Example dry run (nums = [1,3,5,4,7]):

`dp = [1,2,3,3,4]`

ct = [1,1,1,1,2]

Maximum LIS length = 4

Total count = 2



## Code

```
#include <bits/stdc++.h>
using namespace std;
```

```
class Solution {
```

```

public:
 int findNumberOfLIS(vector<int>& arr) {
 int n = arr.size();
 vector<int> dp(n, 1), ct(n, 1);
 int maxi = 1;

 for(int i=0;i<n;i++){
 for(int j=0;j<i;j++){
 if(arr[j] < arr[i] && dp[j] + 1 > dp[i]){
 dp[i] = dp[j] + 1;
 ct[i] = ct[j];
 }
 else if(arr[j] < arr[i] && dp[j] + 1 == dp[i]){
 ct[i] += ct[j];
 }
 }
 maxi = max(maxi, dp[i]);
 }

 int countLIS = 0;
 for(int i=0;i<n;i++){
 if(dp[i] == maxi){
 countLIS += ct[i];
 }
 }
 return countLIS;
 }
};

int main(){
 vector<int> arr = {1,5,4,3,2,6,7,2};
 Solution sol;
 cout << "The count of Longest Increasing Subsequences (LIS) is "
 << sol.findNumberOfLIS(arr);
 return 0;
}

```

## Complexity Analysis

Time Complexity:  $O(N^2)$

For each element, we compare it with all previous elements.

Space Complexity: O(N)

We use two arrays of size N to store LIS length and count.

# Matrix Chain Multiplication | (DP - 48)

We are given an array arr of size n+1.

This array represents n matrices such that:

- Matrix A1 has dimension arr[0] x arr[1]
- Matrix A2 has dimension arr[1] x arr[2]
- ...
- Matrix An has dimension arr[n-1] x arr[n]

Our task is to find the **minimum number of scalar multiplications** needed to multiply all matrices together.

The order of multiplication matters because matrix multiplication cost depends on dimensions.

Example:

arr = [40, 20, 30, 10, 30]

Best order is ((A1 × (A2 × A3)) × A4)

Minimum cost = 26000

---

## Approach 1: Recursive Approach

### Algorithm

This is a classic **partition DP** problem.

1. We define a function  $f(i, j)$  that returns the minimum cost to multiply matrices from  $A_i$  to  $A_j$ .
2. **Base case:**
  - o If  $i == j$ , there is only one matrix, so no multiplication is needed. Return 0.
3. **Try all possible partitions:**
  - o For  $k$  from  $i$  to  $j-1$ , split the chain into:

- $f(i, k)$
- $f(k+1, j)$

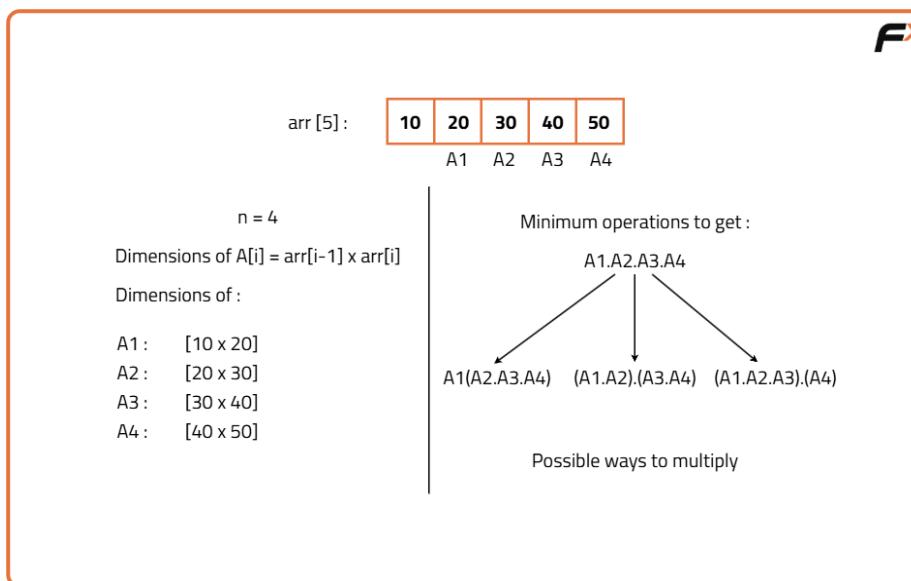
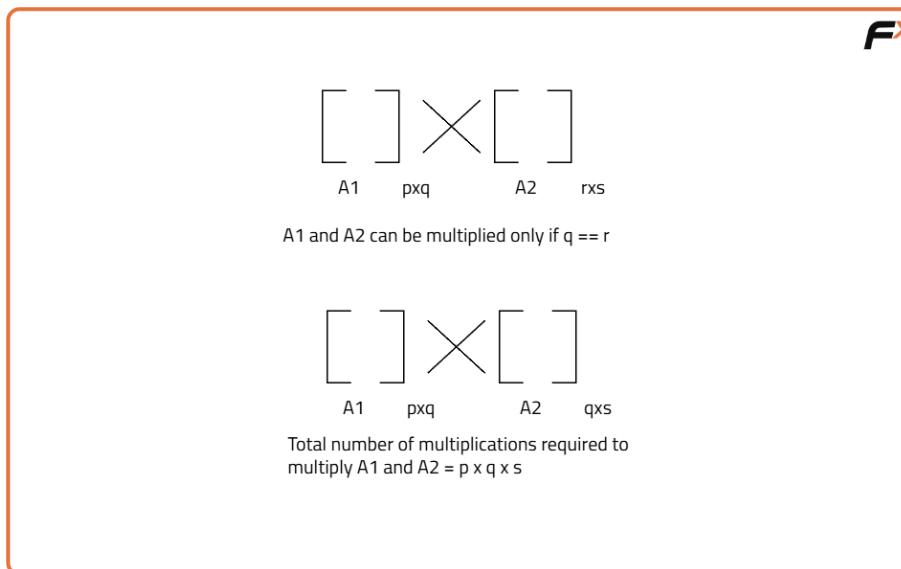
#### 4. Cost calculation:

- Cost =  $f(i, k) + f(k+1, j) + (\text{arr}[i-1] * \text{arr}[k] * \text{arr}[j])$

5. Return the minimum cost among all partitions.

Example dry run ( $\text{arr} = [40, 20, 30, 10, 30]$ ):

We try all possible  $k$  splits and compute cost for each parenthesization.  
The minimum among them is returned.



## Code

```
#include <bits/stdc++.h>
```

```

using namespace std;

class Solution {
public:
 int matrixChainOrder(vector<int>& arr, int i, int j) {
 if(i == j) return 0;
 int minCost = INT_MAX;
 for(int k = i; k < j; k++) {
 int cost1 = matrixChainOrder(arr, i, k);
 int cost2 = matrixChainOrder(arr, k + 1, j);
 int costMultiply = arr[i - 1] * arr[k] * arr[j];
 int total = cost1 + cost2 + costMultiply;
 minCost = min(minCost, total);
 }
 return minCost;
 }
};

int main() {
 vector<int> arr = {40, 20, 30, 10, 30};
 Solution sol;
 cout << "Minimum number of multiplications is: "
 << sol.matrixChainOrder(arr, 1, arr.size() - 1);
 return 0;
}

```

## Complexity Analysis

Time Complexity:  $O(2^n)$   
 Every possible parenthesization is explored recursively.

Space Complexity:  $O(n)$   
 Maximum recursion depth is n.

---

## Approach 2: Memoization (Top-Down DP)

### Algorithm

The recursive approach has overlapping subproblems, so we store results.

1. Create a 2D dp array where  $dp[i][j]$  stores the minimum cost to multiply matrices from  $i$  to  $j$ .
2. Initialize dp with -1.
3. Before computing  $f(i, j)$ , check:
  - o If  $dp[i][j]$  is already computed, return it.
4. Otherwise, compute using the same recursive logic and store the result in  $dp[i][j]$ .
5. Final answer is  $dp[1][n-1]$ .

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int solve(vector<int>& arr, int i, int j, vector<vector<int>>& dp) {
 if(i == j) return 0;
 if(dp[i][j] != -1) return dp[i][j];
 int minCost = INT_MAX;
 for(int k = i; k < j; k++) {
 int cost1 = solve(arr, i, k, dp);
 int cost2 = solve(arr, k + 1, j, dp);
 int costMultiply = arr[i - 1] * arr[k] * arr[j];
 int total = cost1 + cost2 + costMultiply;
 minCost = min(minCost, total);
 }
 return dp[i][j] = minCost;
 }

 int matrixChainOrder(vector<int>& arr) {
 int n = arr.size();
 vector<vector<int>> dp(n, vector<int>(n, -1));
 return solve(arr, 1, n - 1, dp);
 }
};

int main() {
 vector<int> arr = {40, 20, 30, 10, 30};
 Solution sol;
 cout << "Minimum number of multiplications is: "
 << sol.matrixChainOrder(arr);
 return 0;
}
```

## Complexity Analysis

Time Complexity:  $O(n^3)$

There are  $O(n^2)$  states  $(i, j)$  and for each we try  $O(n)$  partitions.

Space Complexity:  $O(n^2)$

We use a 2D dp table of size  $n \times n$  and recursion stack of  $O(n)$ .

# Matrix Chain Multiplication | Tabulation Method | (DP - 49)

We are given an array `nums` of size  $n$ .

This array represents a chain of matrices such that:

- Matrix A1 has dimension  $\text{nums}[0] \times \text{nums}[1]$
- Matrix A2 has dimension  $\text{nums}[1] \times \text{nums}[2]$
- ...
- Matrix A( $n-1$ ) has dimension  $\text{nums}[n-2] \times \text{nums}[n-1]$

Our task is to find the **minimum number of scalar multiplications** needed to multiply the entire chain of matrices by placing parentheses optimally.

Example:

`nums = [10, 15, 20, 25]`

Two ways:

- $A1 \times (A2 \times A3) \rightarrow 10 \times 15 \times 25 + 15 \times 20 \times 25 = 11250$
  - $(A1 \times A2) \times A3 \rightarrow 10 \times 15 \times 20 + 10 \times 20 \times 25 = 8000$
- Minimum cost = 8000

---

## Approach

We solve this using **tabulation (bottom-up DP)**.

1. Create a 2D DP array dp where:
  - o  $dp[i][j]$  represents the minimum cost to multiply matrices from  $A_i$  to  $A_j$ .
2. Initialize:
  - o  $dp[i][i] = 0$  because a single matrix needs no multiplication.
3. Fill the table by increasing chain length:
  - o Start from chains of length 2 up to  $n-1$ .
4. For each pair  $(i, j)$ , try all possible split points k:
  - o Cost =  $dp[i][k] + dp[k+1][j] + \text{nums}[i-1] \times \text{nums}[k] \times \text{nums}[j]$
5. Store the minimum cost for  $dp[i][j]$ .
6. Final answer is stored in  $dp[1][n-1]$ .

Example dry run ( $\text{nums} = [4, 2, 3]$ ):

Only one multiplication:

Cost =  $4 \times 2 \times 3 = 24$

So  $dp[1][2] = 24$ .

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int matrixMultiplication(vector<int>& nums) {
 int n = nums.size();
 vector<vector<int>> dp(n, vector<int>(n, INT_MAX));

 // Single matrix cost is zero
 for(int i = 1; i < n; i++) {
 dp[i][i] = 0;
 }

 // Chain length from 2 to n-1
 for(int len = 2; len < n; len++) {
 for(int i = 1; i <= n - len; i++) {
 int j = i + len - 1;
 for(int k = i; k < j; k++) {
 int cost = dp[i][k] + dp[k+1][j]
 + nums[i-1] * nums[k] * nums[j];
 dp[i][j] = min(dp[i][j], cost);
 }
 }
 }
 }
}
```

```

 return dp[1][n-1];
 }
};

int main() {
 vector<int> nums = {10, 15, 20, 25};
 Solution sol;
 cout << sol.matrixMultiplication(nums);
 return 0;
}

```

## Complexity Analysis

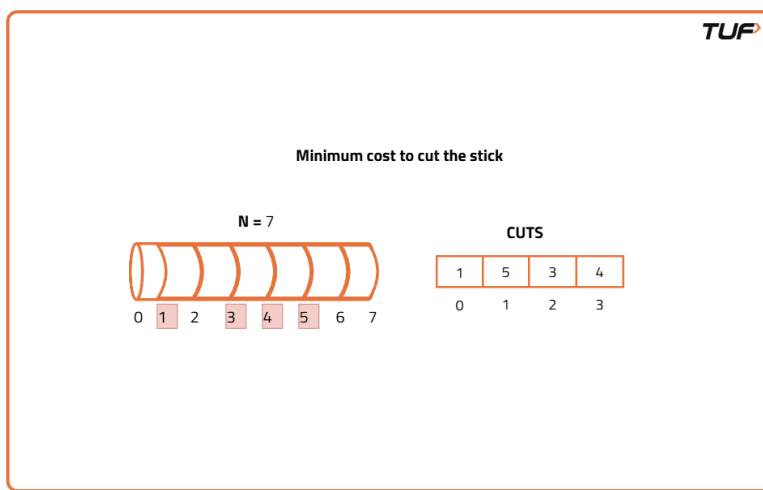
Time Complexity:  $O(N^3)$

We use three nested loops: chain length, start index, and split point.

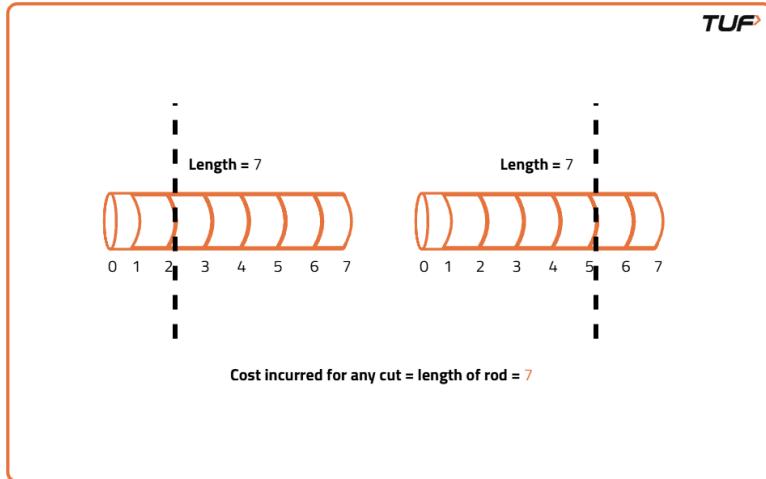
Space Complexity:  $O(N^2)$

A 2D DP table is used to store minimum multiplication costs.

## Minimum Cost to Cut the Stick | (DP - 50)



We are given a stick of length  $n$  and an array  $\text{cuts}$  of size  $c$ .  
 Each value in  $\text{cuts}$  represents a position where the stick must be cut.  
 Every cut costs the **current length of the stick segment** being cut.

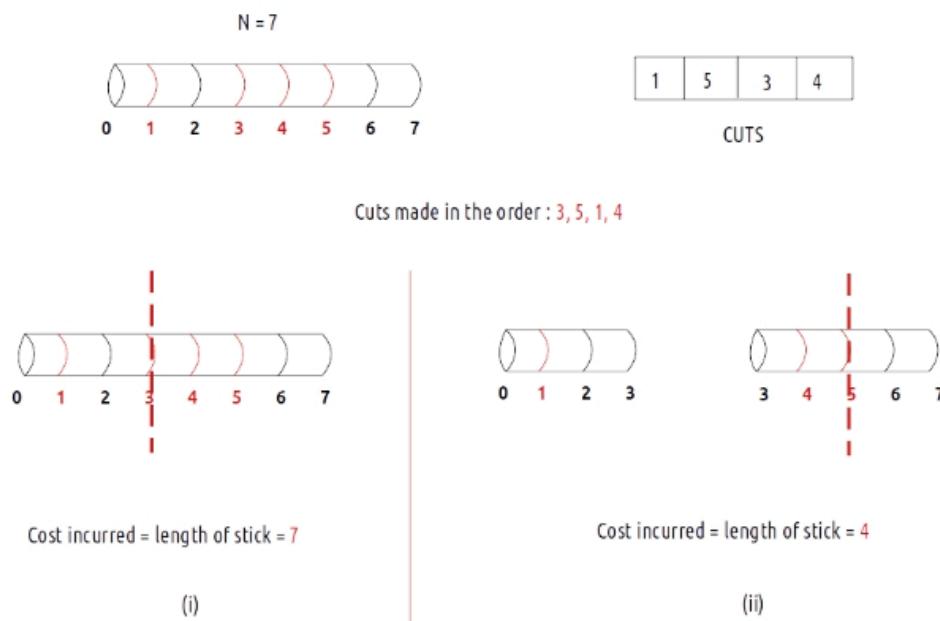


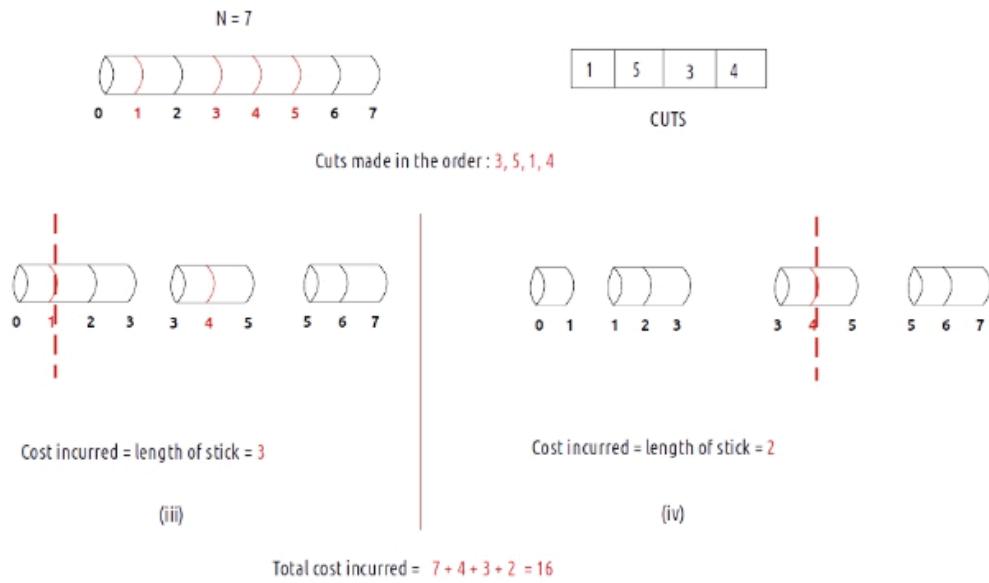
We must perform **all cuts**, but we can choose the order.  
The goal is to find the **minimum total cost**.

Example:

$n = 7$ , cuts = [1,3,4,5]

One optimal order gives total cost = 16.





## Approach 1: Recursion

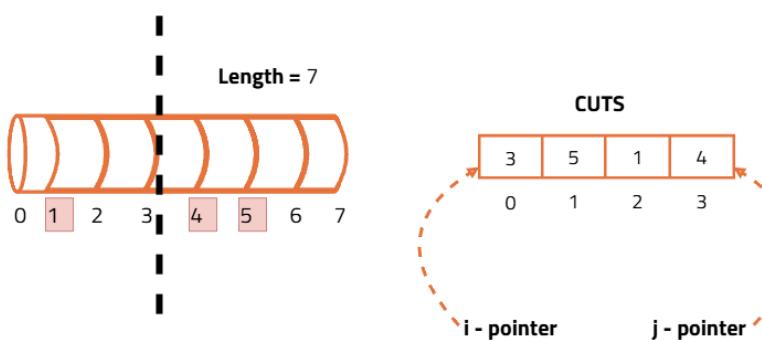
### Algorithm

This problem depends on the **order of cuts**, so we use **Partition DP**.

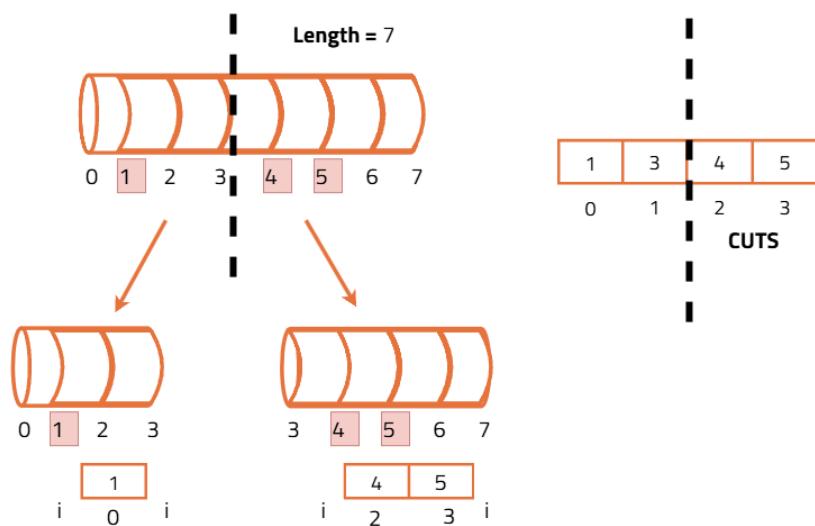
Steps:

1. First sort the cuts array.
2. Add 0 at the start and n at the end to represent stick boundaries.
3. Define a recursive function  $f(i, j)$ :
  - o It returns the minimum cost to cut between cuts[i] and cuts[j].
4. Base case:
  - o If  $i > j$ , there are no cuts to make, so cost = 0.
5. For every possible cut index ind from i to j:
  - o Cost of current cut =  $\text{cuts}[j+1] - \text{cuts}[i-1]$
  - o Add cost of left part:  $f(i, \text{ind}-1)$
  - o Add cost of right part:  $f(\text{ind}+1, j)$
6. Return the minimum cost among all choices.

Minimum cost to cut the stick



Minimum cost to cut the stick



```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int solve(int i,int j,vector<int> &cuts){
```

```

if(i>j) return 0;
int mini=INT_MAX;
for(int ind=i;ind<=j;ind++){
 int cost = cuts[j+1]-cuts[i-1]
 + solve(i,ind-1,cuts)
 + solve(ind+1,j,cuts);
 mini=min(mini,cost);
}
return mini;
}

int minimumCost(int n,int c,vector<int> &cuts){
 cuts.push_back(n);
 cuts.insert(cuts.begin(),0);
 sort(cuts.begin(),cuts.end());
 return solve(1,c,cuts);
}
};

int main(){
 vector<int> cuts={3,5,1,4};
 int n=7;
 Solution sol;
 cout<<sol.minimumCost(n,cuts.size(),cuts);
 return 0;
}

```

## Complexity Analysis

Time Complexity:  $O(2^C)$

All possible cut orders are tried recursively.

Space Complexity:  $O(C)$

Recursive stack depth.

---

## Approach 2: Memoization

### Algorithm

The recursive solution has overlapping subproblems.

We store results in a DP table to avoid recomputation.

Steps:

1. Use a 2D dp array where  $dp[i][j]$  stores minimum cost for cuts i to j.
2. Initialize dp with -1.
3. Before computing  $f(i,j)$ , check  $dp[i][j]$ .
4. If already computed, return it.
5. Otherwise compute using recursion and store in dp.

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int solve(int i,int j,vector<int> &cuts,vector<vector<int>> &dp){
 if(i>j) return 0;
 if(dp[i][j]!=-1) return dp[i][j];
 int mini=INT_MAX;
 for(int ind=i;ind<=j;ind++){
 int cost = cuts[j+1]-cuts[i-1]
 + solve(i,ind-1,cuts,dp)
 + solve(ind+1,j,cuts,dp);
 mini=min(mini,cost);
 }
 return dp[i][j]=mini;
 }

 int minimumCost(int n,int c,vector<int> &cuts){
 cuts.push_back(n);
 cuts.insert(cuts.begin(),0);
 sort(cuts.begin(),cuts.end());
 vector<vector<int>> dp(c+1,vector<int>(c+1,-1));
 return solve(1,c,cuts,dp);
 }
};

int main(){
 vector<int> cuts={3,5,1,4};
 int n=7;
 Solution sol;
 cout<<sol.minimumCost(n,cuts.size(),cuts);
 return 0;
}
```

## Complexity Analysis

Time Complexity:  $O(C^3)$

There are  $C^2$  states and for each we try  $C$  cuts.

Space Complexity:  $O(C^2)$

DP table + recursion stack.

---

## Approach 3: Tabulation

### Algorithm

We convert memoization into bottom-up DP.

Steps:

1. Sort cuts and add 0 and n.
2. Create dp table initialized with 0.
3.  $dp[i][j]$  represents minimum cost for cuts i to j.
4. Fill dp by increasing segment length:
  - o i moves from c to 1
  - o j moves from i to c
5. For every possible cut ind between i and j:
  - o  $dp[i][j] = \min(dp[i][j],$   
 $cuts[j+1]-cuts[i-1] + dp[i][ind-1] + dp[ind+1][j])$
6. Final answer is  $dp[1][c]$ .

For Subsequence length

$$dp[i][j] = \min ( cuts[j] - cuts[i] + dp[i][k] + dp[k][j] )$$

For all k in ( i + 1 ) to ( j - 1 )

|        | 0 | 1 | 2 | 3 | 4  | 5  |
|--------|---|---|---|---|----|----|
| dp = 0 | - | 0 | 3 | 7 | 10 | 16 |
| 1      | - | 0 | 3 | 6 | 12 |    |
| 2      |   | - | 0 | 2 | 6  |    |
| 3      |   |   | - | 0 | 3  |    |
| 4      |   |   |   | - | 0  |    |
| 5      |   |   |   |   | -  |    |

return:

$$dp[0][5] = 16$$

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int minimumCost(int n,int c,vector<int> &cuts){
 cuts.push_back(n);
 cuts.insert(cuts.begin(),0);
 sort(cuts.begin(),cuts.end());

 vector<vector<int>> dp(c+2,vector<int>(c+2,0));

 for(int i=c;i>=1;i--){
 for(int j=i;j<=c;j++){
 int mini=INT_MAX;
 for(int ind=i;ind<=j;ind++){
 int cost = cuts[j+1]-cuts[i-1]
 + dp[i][ind-1]
 + dp[ind+1][j];
 mini=min(mini,cost);
 }
 dp[i][j]=mini;
 }
 }
 return dp[1][c];
 }
}
```

```

 }
};

int main(){
 vector<int> cuts={3,5,1,4};
 int n=7;
 Solution sol;
 cout<<sol.minimumCost(n,cuts.size(),cuts);
 return 0;
}

```

## Complexity Analysis

Time Complexity:  $O(C^3)$   
 Three nested loops over cuts.

Space Complexity:  $O(C^2)$   
 2D DP table used.

# Burst Balloons | Partition DP | (DP - 51)

We are given an array  $arr[]$  of size  $n$  where each element represents a balloon with a number on it.

We need to burst all the balloons one by one.

If we burst the  $i$ -th balloon, the coins gained are:  
 $arr[i-1] * arr[i] * arr[i+1]$

If  $i-1$  or  $i+1$  goes out of bounds, we treat that value as 1.

Our task is to find the **maximum coins** we can collect by bursting the balloons in an optimal order.

Example:

arr = [3,1,5,8]

One optimal order gives total coins = 167.

---

## Approach 1: Recursive Approach

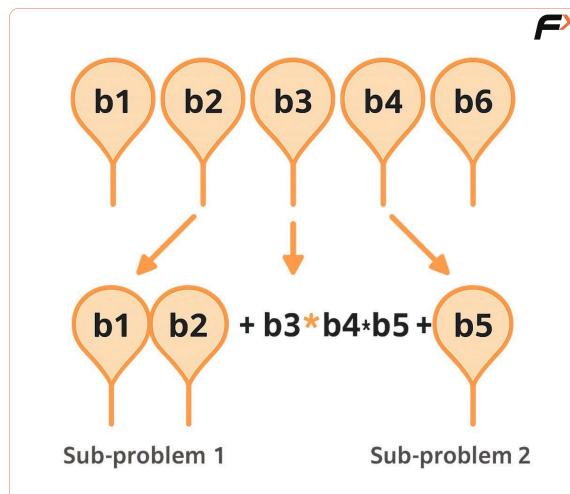
### Algorithm

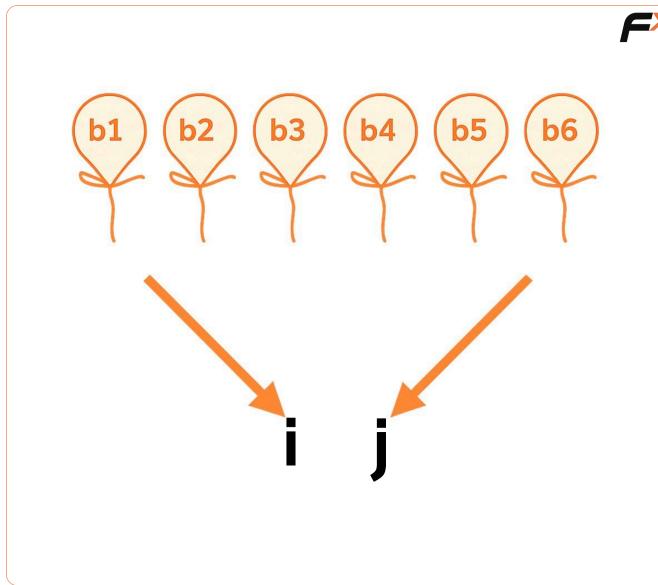
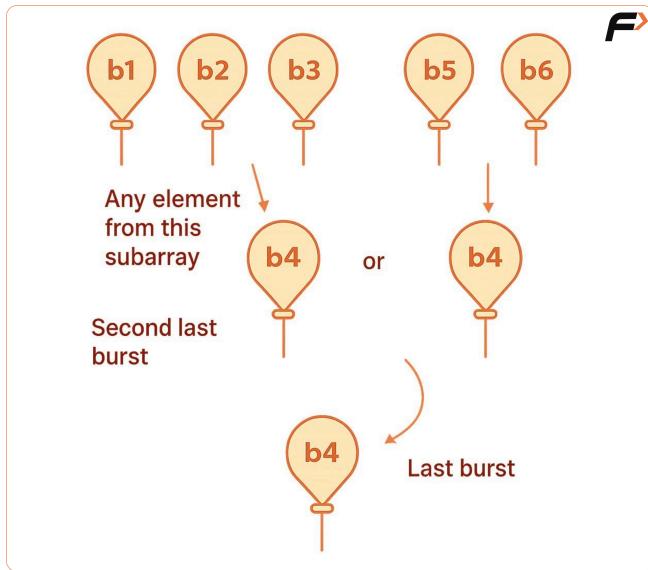
The order of bursting balloons affects the total coins.

Instead of choosing the first balloon to burst, we think in reverse and choose the **last balloon to burst**.

Steps:

1. Add 1 at the beginning and end of the array to handle boundary conditions.
2. Define a recursive function  $f(i, j)$  which returns the maximum coins obtainable by bursting balloons from index  $i$  to  $j$ .
3. Base case:
  - o If  $i > j$ , no balloons left, return 0.
4. Try every balloon  $k$  between  $i$  and  $j$  as the **last balloon to burst**.
5. Coins gained:
  - o  $\text{nums}[i-1] * \text{nums}[k] * \text{nums}[j+1]$
6. Add coins from left part  $f(i, k-1)$  and right part  $f(k+1, j)$ .
7. Return the maximum value among all choices.





## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int solve(int i,int j,vector<int> &nums){
 if(i>j) return 0;
 int maxi=INT_MIN;
 for(int k=i;k<=j;k++){
 int coins = nums[i-1]*nums[k]*nums[j+1]
 + solve(i,k-1,nums)
 }
 }
}
```

```

 + solve(k+1,j,nums);
 maxi = max(maxi, coins);
}
return maxi;
}

int maxCoins(vector<int> &nums){
 int n=nums.size();
 nums.insert(nums.begin(),1);
 nums.push_back(1);
 return solve(1,n,nums);
}
};

int main(){
 vector<int> nums={3,1,5,8};
 Solution sol;
 cout<<"Maximum coins obtained: "<<sol.maxCoins(nums);
 return 0;
}

```

## Complexity Analysis

Time Complexity: Exponential

All possible bursting orders are explored recursively.

Space Complexity: O(N)

Due to recursion stack.

---

## Approach 2: Memoization

### Algorithm

The recursive solution has overlapping subproblems, so we store results in a DP table.

Steps:

1. Create a 2D dp array where  $dp[i][j]$  stores maximum coins for range i to j.
2. Initialize dp with -1.
3. Before solving  $f(i, j)$ , check if  $dp[i][j]$  is already computed.
4. If yes, return it.

- Otherwise, compute using recursion and store the result in dp.

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int solve(int i,int j,vector<int> &nums,vector<vector<int>> &dp){
 if(i>j) return 0;
 if(dp[i][j]!=-1) return dp[i][j];
 int maxi=INT_MIN;
 for(int k=i;k<=j;k++){
 int coins = nums[i-1]*nums[k]*nums[j+1]
 + solve(i,k-1,nums,dp)
 + solve(k+1,j,nums,dp);
 maxi = max(maxi, coins);
 }
 return dp[i][j]=maxi;
 }

 int maxCoins(vector<int> &nums){
 int n=nums.size();
 nums.insert(nums.begin(),1);
 nums.push_back(1);
 vector<vector<int>> dp(n+2,vector<int>(n+2,-1));
 return solve(1,n,nums,dp);
 }
};

int main(){
 vector<int> nums={3,1,5,8};
 Solution sol;
 cout<<"Maximum coins obtained: "<<sol.maxCoins(nums);
 return 0;
}
```

## Complexity Analysis

Time Complexity:  $O(N^3)$

There are  $N^2$  states and for each state we try  $N$  partitions.

Space Complexity:  $O(N^2) + O(N)$   
DP table plus recursion stack.

---

## Approach 3: Tabulation

### Algorithm

We convert the memoized solution into bottom-up DP.

Steps:

1. Add 1 at the beginning and end of the array.
2. Create a 2D dp array initialized with 0.
3.  $dp[i][j]$  represents maximum coins for bursting balloons from  $i$  to  $j$ .
4. Fill dp table such that smaller ranges are solved first:
  - o  $i$  goes from  $n$  to 1
  - o  $j$  goes from 1 to  $n$
5. For each  $(i, j)$ , try all possible last balloons index between  $i$  and  $j$ .
6. Update  $dp[i][j]$  with the maximum value.
7. Final answer is  $dp[1][n]$ .

### Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int maxCoins(vector<int> &nums){
 int n=nums.size();
 nums.insert(nums.begin(),1);
 nums.push_back(1);
 vector<vector<int>> dp(n+2,vector<int>(n+2,0));

 for(int i=n;i>=1;i--){
 for(int j=1;j<=n;j++){
 if(i>j) continue;
 int maxi=INT_MIN;
 for(int ind=i;ind<=j;ind++){
 int coins = nums[i-1]*nums[ind]*nums[j+1]
 + dp[i][ind-1]
 + dp[ind+1][j];
 maxi = max(maxi, coins);
 }
 }
 }
 return dp[1][n];
 }
};
```

```

 }
 dp[i][j]=maxi;
 }
}
return dp[1][n];
}

int main(){
 vector<int> nums={3,1,5,8};
 Solution sol;
 cout<<"Maximum coins obtained: "<<sol.maxCoins(nums);
 return 0;
}

```

## Complexity Analysis

Time Complexity:  $O(N^3)$

Three nested loops over ranges and partitions.

Space Complexity:  $O(N^2)$

2D DP table is used.

# Evaluate Boolean Expression to True | Partition DP | (DP - 52)

We are given a boolean expression string consisting of:

- Operands: T (true) and F (false)
- Operators: & (AND), | (OR), ^ (XOR)

We need to count the number of ways to parenthesize the expression such that the final result evaluates to **True**.

The answer should be returned modulo  **$10^9 + 7$** .

Example:

expression = "F | T^F"

There are 2 different ways to parenthesize the expression so that it evaluates to True.

---

## Approach 1: Brute Force (Recursive Partition DP)

### Algorithm

This is a classic **partition DP** problem where the result depends on how we split the expression.

We define a recursive function:

$f(i, j, \text{isTrue})$

It returns the number of ways to evaluate the substring from index  $i$  to  $j$  such that:

- $\text{isTrue} = 1 \rightarrow$  expression evaluates to True
- $\text{isTrue} = 0 \rightarrow$  expression evaluates to False

Base cases:

- If  $i > j$ , return 0 (invalid expression).
- If  $i == j$ , check if the single character matches the required boolean value.

Recursive step:

- Try all operators between  $i$  and  $j$ .
- Split the expression into left and right parts.
- Recursively compute:
  - $\text{leftTrue}, \text{leftFalse}$
  - $\text{rightTrue}, \text{rightFalse}$
- Combine results based on the operator:
  - $\&$  → True only if both sides are True
  - $|$  → True if any side is True
  - $^$  → True if exactly one side is True
- Sum all valid combinations.

### Code

```
#include <bits/stdc++.h>
using namespace std;

const int mod = 1000000007;
```

```

int solve(int i,int j,int isTrue,string &exp){
 if(i>j) return 0;
 if(i==j){
 if(isTrue) return exp[i]=='T';
 else return exp[i]=='F';
 }
 long long ways=0;
 for(int k=i+1;k<=j-1;k+=2){
 long long IT=solve(i,k-1,1,exp);
 long long IF=solve(i,k-1,0,exp);
 long long rT=solve(k+1,j,1,exp);
 long long rF=solve(k+1,j,0,exp);

 if(exp[k]=='&'){
 if(isTrue) ways=(ways+IT*rT)%mod;
 else ways=(ways+IF*rT+IT*rF+IF*rF)%mod;
 }
 else if(exp[k]=='|'){
 if(isTrue) ways=(ways+IT*rT+IT*rF+IF*rT)%mod;
 else ways=(ways+IF*rF)%mod;
 }
 else{
 if(isTrue) ways=(ways+IT*rF+IF*rT)%mod;
 else ways=(ways+IT*rT+IF*rF)%mod;
 }
 }
 return ways;
}

int evaluateExp(string &exp){
 return solve(0,exp.size()-1,1,exp);
}

int main(){
 string exp="F|T^F";
 cout<<evaluateExp(exp);
 return 0;
}

```

## Complexity Analysis

Time Complexity: Exponential  
 All possible parenthesizations are explored.

Space Complexity: O(N)  
Recursion stack depth.

---

## Approach 2: Memoization (Top-Down DP)

### Algorithm

The brute force solution has overlapping subproblems.

We use a 3D DP array:

`dp[i][j][isTrue]`

It stores the number of ways to evaluate substring  $i$  to  $j$  as True or False.

Before solving any subproblem, we check if it is already computed.

### Code

```
#include <bits/stdc++.h>
using namespace std;

const int mod = 1000000007;

int solve(int i,int j,int isTrue,string &exp,vector<vector<vector<int>>> &dp){
 if(i>j) return 0;
 if(i==j){
 if(isTrue) return exp[i]=='T';
 else return exp[i]=='F';
 }
 if(dp[i][j][isTrue]!=-1) return dp[i][j][isTrue];

 long long ways=0;
 for(int k=i+1;k<=j-1;k+=2){
 long long IT=solve(i,k-1,1,exp,dp);
 long long IF=solve(i,k-1,0,exp,dp);
 long long rT=solve(k+1,j,1,exp,dp);
 long long rF=solve(k+1,j,0,exp,dp);

 if(exp[k]=='&'){
 if(isTrue) ways=(ways+IT*rT)%mod;
 else ways=(ways+IF*rT+IT*rF+IF*rF)%mod;
 }
 else if(exp[k]=='|'){
 if(isTrue) ways=(ways+IT+IF)%mod;
 else ways=(ways+(IT*rT+IF*rF)-(IT*rF+IF*rT))%mod;
 }
 }
 dp[i][j][isTrue]=ways;
 return ways;
}
```

```

 if(isTrue) ways=(ways+IT*rT+IT*rF+IF*rT)%mod;
 else ways=(ways+IF*rF)%mod;
 }
 else{
 if(isTrue) ways=(ways+IT*rF+IF*rT)%mod;
 else ways=(ways+IT*rT+IF*rF)%mod;
 }
}
return dp[i][j][isTrue]=ways;
}

int evaluateExp(string &exp){
 int n=exp.size();
 vector<vector<vector<int>>> dp(n,vector<vector<int>>(n,vector<int>(2,-1)));
 return solve(0,n-1,1,exp,dp);
}

int main(){
 string exp="F|T^F";
 cout<<evaluateExp(exp);
 return 0;
}

```

## Complexity Analysis

Time Complexity:  $O(N^3)$

There are  $O(N^2 \times 2)$  states, and for each we try  $O(N)$  partitions.

Space Complexity:  $O(N^2) + O(N)$

DP table and recursion stack.

---

## Approach 3: Tabulation (Bottom-Up DP)

### Algorithm

We convert memoization into bottom-up DP.

Steps:

- Create a 3D DP table  $dp[i][j][\text{isTrue}]$ .
- Base case: when  $i == j$ , directly evaluate T or F.

- Fill DP table for increasing substring lengths.
- Use the same operator combination logic as recursion.
- Final answer is  $dp[0][n-1][1]$ .

## Code

```
#include <bits/stdc++.h>
using namespace std;

const int mod = 1000000007;

int evaluateExp(string &exp){
 int n=exp.size();
 vector<vector<vector<long long>>> dp(n, vector<vector<long long>>(n, vector<long long>(2,0)));

 for(int i=0;i<n;i++){
 dp[i][i][1]=(exp[i]=='T');
 dp[i][i][0]=(exp[i]=='F');
 }

 for(int len=3;len<=n;len+=2){
 for(int i=0;i+len-1<n;i++){
 int j=i+len-1;
 for(int k=i+1;k<=j-1;k+=2){
 for(int isTrue=0;isTrue<=1;isTrue++){
 long long IT=dp[i][k-1][1];
 long long IF=dp[i][k-1][0];
 long long rT=dp[k+1][j][1];
 long long rF=dp[k+1][j][0];

 if(exp[k]=='&'){
 if(isTrue) dp[i][j][1]=(dp[i][j][1]+IT*rT)%mod;
 else dp[i][j][0]=(dp[i][j][0]+IF*rT+IT*rF+IF*rF)%mod;
 }
 else if(exp[k]=='|'){
 if(isTrue) dp[i][j][1]=(dp[i][j][1]+IT*rT+IT*rF+IF*rT)%mod;
 else dp[i][j][0]=(dp[i][j][0]+IF*rF)%mod;
 }
 else{
 if(isTrue) dp[i][j][1]=(dp[i][j][1]+IT*rF+IF*rT)%mod;
 else dp[i][j][0]=(dp[i][j][0]+IT*rT+IF*rF)%mod;
 }
 }
 }
 }
 }
}
```

```

 }
 }
 return dp[0][n-1][1];
}

int main(){
 string exp="F|T^F";
 cout<<evaluateExp(exp);
 return 0;
}

```

## Complexity Analysis

Time Complexity:  $O(N^3)$

Triple nested loops over substring length, partition, and boolean state.

Space Complexity:  $O(N^2)$

3D DP table used.

# Palindrome Partitioning – II | Front Partition | (DP - 53)

We are given a string  $s$ .

We need to partition the string such that **every substring is a palindrome**.

Our task is to return the **minimum number of cuts** needed to achieve this.

A cut means dividing the string into two parts.

Example:

$s = "aab"$

One valid partition is: aa | b

Only 1 cut is needed, so the answer is 1.

# Approach 1: Memoization (Front Partition)

## Algorithm

This problem is solved using the **front partition technique**.

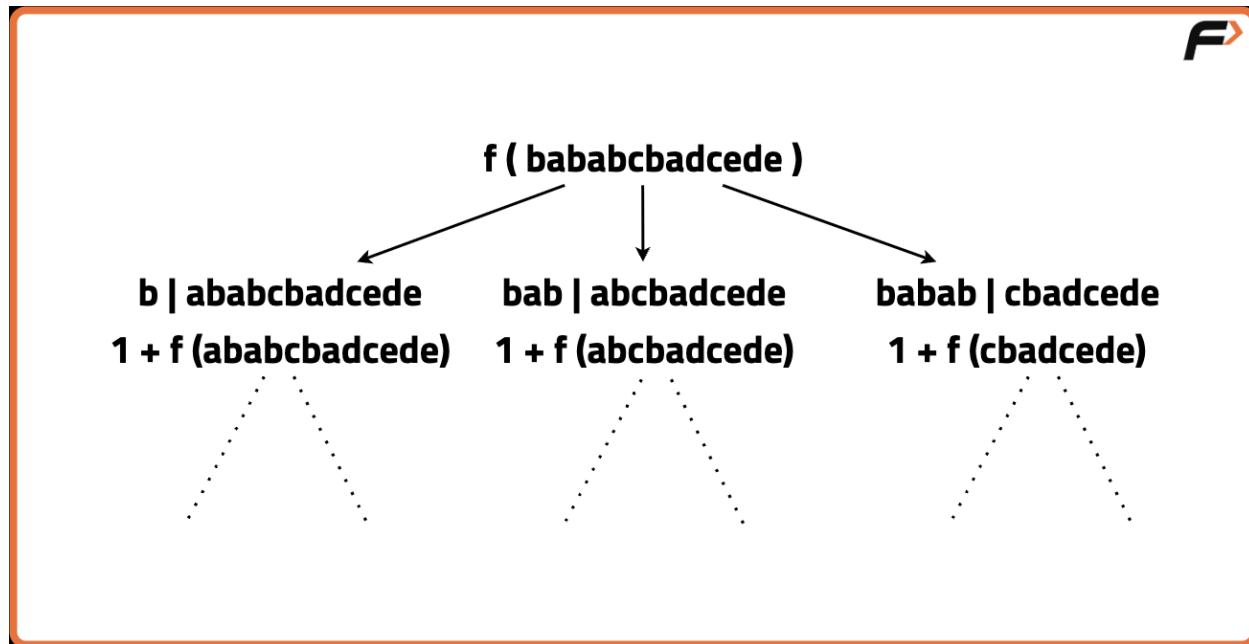
We start partitioning the string from the first index and move forward.

Steps:

1. Define a function  $f(\text{start})$  which returns the minimum cuts needed for substring  $s[\text{start} \dots \text{end}]$ .
2. Base case:
  - o If  $\text{start}$  reaches the end of the string, return 0.
  - o If the substring  $s[\text{start} \dots \text{end}]$  itself is a palindrome, return 0.
3. From index  $\text{start}$ , try all possible ending indices  $\text{end}$ .
4. If substring  $s[\text{start} \dots \text{end}]$  is a palindrome:
  - o Make 1 cut and recursively solve for  $\text{end} + 1$ .
5. Take the minimum among all possible cuts.
6. Use a memo array  $dp[\text{start}]$  to store results and avoid recomputation.

Example dry run ( $s = "aab"$ ):

- From index 0:
  - o "a" is palindrome  $\rightarrow 1 + f(1)$
  - o "aa" is palindrome  $\rightarrow 1 + f(2)$
- Best choice gives minimum cuts = 1.



## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
private:
 bool isPalindrome(const string& s,int i,int j){
 while(i<j){
 if(s[i]!=s[j]) return false;
 i++; j--;
 }
 return true;
 }

 int solve(const string& s,int start,vector<int>& dp){
 int n=s.size();
 if(start==n) return 0;
 if(isPalindrome(s,start,n-1)) return 0;
 if(dp[start]!=-1) return dp[start];

 int mini=INT_MAX;
 for(int end=start;end<n;end++){
 if(isPalindrome(s,start,end)){
 int cuts=1+solve(s,end+1,dp);
 mini=min(mini,cuts);
 }
 }
 dp[start]=mini;
 return mini;
 }
}
```

```

 }
 return dp[start]=mini;
 }

public:
 int minCut(string s){
 int n=s.size();
 vector<int> dp(n,-1);
 return solve(s,0,dp);
 }
};

int main(){
 string s="aab";
 Solution sol;
 cout<<"Minimum cuts needed: "<<sol.minCut(s);
 return 0;
}

```

## Complexity Analysis

Time Complexity:  $O(N^2)$

There are N states, and for each state we check palindrome substrings.

Space Complexity:  $O(N) + O(N)$

DP array plus recursion stack.

---

## Approach 2: Tabulation (Bottom-Up)

### Algorithm

We convert the memoization approach into a bottom-up DP.

Steps:

1. Define a DP array  $dp[i]$  where:
  - o  $dp[i] = \text{minimum cuts needed for substring } s[i \dots \text{end}]$ .
2. Base case:
  - o  $dp[n] = -1$  (no cut needed after the last character).
3. Fill DP from right to left.
4. For each index  $i$ , try all substrings  $s[i \dots j]$ .

5. If  $s[i \dots j]$  is a palindrome:
  - o  $dp[i] = \min(dp[i], 1 + dp[j+1])$
6. Final answer is  $dp[0]$ .

Example dry run ( $s = "aab"$ ):

- $dp[3] = -1$
- $dp[2] = 0 ("b")$
- $dp[1] = 1 ("a|b")$
- $dp[0] = 1 ("aa|b")$

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
private:
 bool isPalindrome(const string& s, int i, int j){
 while(i < j){
 if(s[i] != s[j]) return false;
 i++; j--;
 }
 return true;
 }

public:
 int minCut(string s){
 int n = s.size();
 vector<int> dp(n + 1, 0);
 dp[n] = -1;

 for(int i = n - 1; i >= 0; i--){
 int mini = INT_MAX;
 for(int j = i; j < n; j++){
 if(isPalindrome(s, i, j)){
 mini = min(mini, 1 + dp[j + 1]);
 }
 }
 dp[i] = mini;
 }
 return dp[0];
 }
};
```

```

int main(){
 string s="aab";
 Solution sol;
 cout<<"Minimum cuts needed: "<<sol.minCut(s);
 return 0;
}

```

## Complexity Analysis

Time Complexity:  $O(N^2)$

For each index, all substrings are checked.

Space Complexity:  $O(N)$

Only a DP array is used.

# Partition Array for Maximum Sum | Front Partition | (DP - 54)

We are given an integer array  $\text{arr}$  of length  $n$  and an integer  $k$ .

We can partition the array into one or more **contiguous sub-arrays**, where each sub-array length is between 1 and  $k$ .

After making a partition:

- Every element in a sub-array is replaced by the **maximum value** of that sub-array.
- The final array is only used to calculate the total sum.

Our task is to return the **maximum possible sum** after performing exactly one such partition-and-replace operation.

Example:

$\text{arr} = [1, 15, 7, 9, 2, 5, 10]$ ,  $k = 3$

One optimal partition is:  $[1, 15, 7 | 9 | 2, 5, 10]$

After replacement: [15,15,15,9,10,10,10]  
Sum = 84

---

## Approach 1: Memoization (Front Partition)

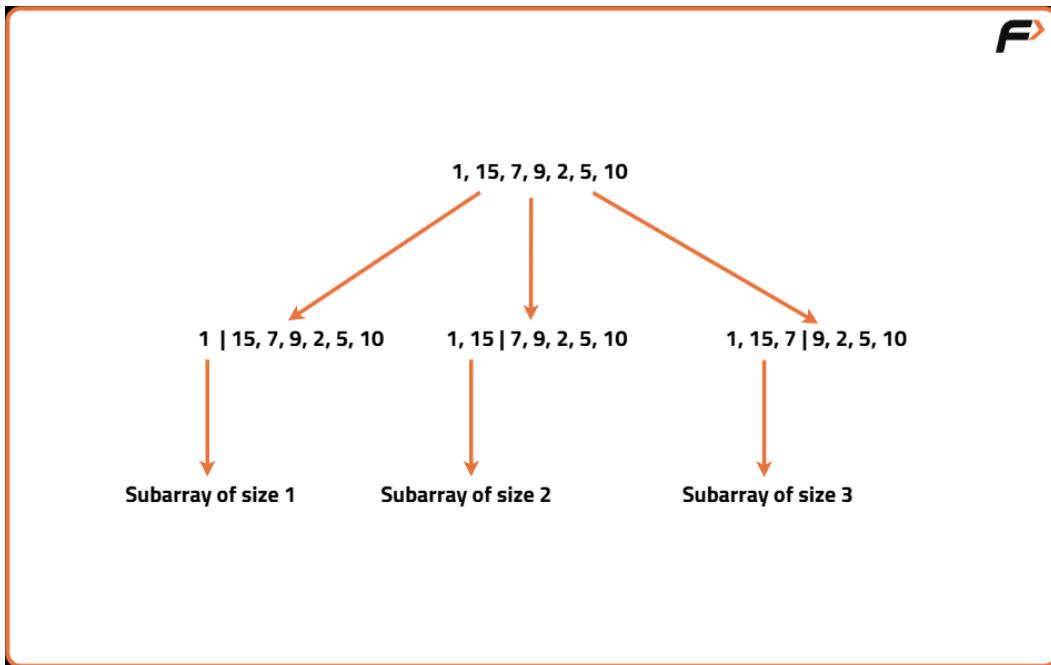
### Algorithm

This is a **front partition DP** problem where we decide partitions starting from the first index.

1. Define a recursive function  $f(\text{start})$  that returns the maximum sum obtainable from index  $\text{start}$  to the end.
2. Base case:
  - o If  $\text{start} == n$ , return 0 (no elements left).
3. Use a DP array  $dp[\text{start}]$  to store the result for each starting index.
4. From index  $\text{start}$ , try all sub-arrays of length 1 to  $k$  (without crossing array bounds).
5. While extending the sub-array, keep track of the maximum element.
6. For each valid length:
  - o Current contribution =  $\text{maxElement} * \text{length}$
  - o Total sum = current contribution +  $f(\text{start} + \text{length})$
7. Take the maximum among all choices and store it in  $dp[\text{start}]$ .

Example dry run ( $\text{arr} = [2,2,2,2]$ ,  $k = 2$ ):

- From index 0:
  - o Take [2] →  $2 * 1 + f(1)$
  - o Take [2,2] →  $2 * 2 + f(2)$
- Best choice gives total sum = 8.



## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
private:
 int solve(vector<int>& arr,int k,int start,vector<int>& dp){
 int n = arr.size();
 if(start == n) return 0;
 if(dp[start] != -1) return dp[start];

 int maxSum = 0;
 int maxElem = 0;

 for(int len = 1; len <= k && start + len <= n; len++){
 maxElem = max(maxElem, arr[start + len - 1]);
 int cur = maxElem * len + solve(arr, k, start + len, dp);
 maxSum = max(maxSum, cur);
 }
 return dp[start] = maxSum;
 }

public:
 int maxSumAfterPartitioning(vector<int>& arr,int k){
 int n = arr.size();
```

```

 vector<int> dp(n, -1);
 return solve(arr, k, 0, dp);
 }
};

int main(){
 vector<int> arr = {1,15,7,9,2,5,10};
 int k = 3;
 Solution sol;
 cout << sol.maxSumAfterPartitioning(arr, k);
 return 0;
}

```

## Complexity Analysis

Time Complexity:  $O(N*K)$

There are N states, and for each state we try up to K partition lengths.

Space Complexity:  $O(N) + O(N)$

DP array plus recursion stack.

---

## Approach 2: Tabulation (Bottom-Up)

### Algorithm

We convert the memoization solution into an iterative DP.

1. Create a DP array dp of size n+1 where:
  - o  $dp[i]$  = maximum sum obtainable from index i to the end.
2. Base case:
  - o  $dp[n] = 0$
3. Fill dp from right to left (from n-1 to 0).
4. For each index i:
  - o Try all sub-arrays of length 1 to k.
  - o Track the maximum element in the current sub-array.
  - o Update  $dp[i] = \max(dp[i], \maxElem * length + dp[i+length])$
5. Final answer is  $dp[0]$ .

### Code

```

#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int maxSumAfterPartitioning(vector<int>& arr,int k){
 int n = arr.size();
 vector<int> dp(n+1, 0);

 for(int i = n-1; i >= 0; i--){
 int maxElem = 0;
 int maxSum = 0;

 for(int len = 1; len <= k && i + len <= n; len++){
 maxElem = max(maxElem, arr[i + len - 1]);
 maxSum = max(maxSum, maxElem * len + dp[i + len]);
 }
 dp[i] = maxSum;
 }
 return dp[0];
 }
};

int main(){
 vector<int> arr = {1,15,7,9,2,5,10};
 int k = 3;
 Solution sol;
 cout << sol.maxSumAfterPartitioning(arr, k);
 return 0;
}

```

## Complexity Analysis

Time Complexity:  $O(N*K)$

For each index, we check up to K partition lengths.

Space Complexity:  $O(N)$

Only the DP array is used.

# Maximum Rectangle Area with all 1's

## | DP on Rectangles | (DP - 55)

We are given a binary matrix of size  $m \times n$  containing only 0's and 1's.

We need to find the **largest rectangular area** that contains only 1's and return its area.

A rectangle must be formed using **contiguous rows and contiguous columns**.

Example:

```
matrix =
[[1,0,1,0,0],
 [1,0,1,1,1],
 [1,1,1,1,1],
 [1,0,0,1,0]]
```

|   |   |   |   |   |
|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 |

The maximum rectangle of only 1's has area = 6.

|   |   |   |   |   |
|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 |

# Approach

## Algorithm

This problem is solved by **converting each row of the matrix into a histogram** and then applying the **Largest Rectangle in Histogram** logic.

Steps:

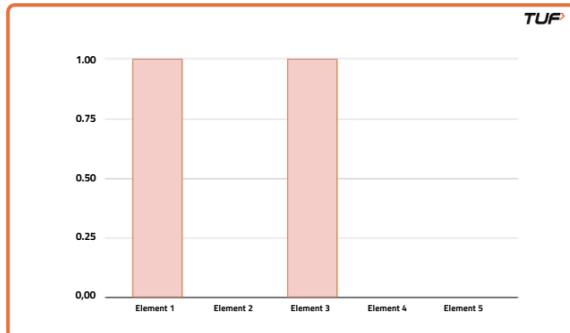
1. Create an array `height[ ]` of size equal to the number of columns.
2. Traverse the matrix row by row.
3. For each cell:
  - o If the value is 1, increase the height of that column by 1.
  - o If the value is 0, reset the height of that column to 0.
4. After processing a row, the `height[ ]` array represents a histogram.
5. Compute the **largest rectangle area in this histogram** using a stack-based approach.
6. Keep track of the maximum area across all rows.
7. The final maximum is the answer.

Example dry run (for first two rows):

- Row 1 → `height = [1,0,1,0,0]`
- Row 2 → `height = [2,0,2,1,1]`

Each of these histograms is processed to get the largest rectangle area.

|   |   |   |   |   |
|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 |



|   |   |   |   |   |
|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 |

## Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 int largestRectangleArea(vector<int>& heights) {
 stack<int> st;
 int maxArea = 0;

 heights.push_back(0); // sentinel

 for(int i = 0; i < heights.size(); i++) {
 while(!st.empty() && heights[i] < heights[st.top()]) {
 int h = heights[st.top()];
 st.pop();
 int w = st.empty() ? i : i - st.top() - 1;
 maxArea = max(maxArea, h * w);
 }
 st.push(i);
 }
 return maxArea;
 }

 int maximalRectangle(vector<vector<char>>& matrix) {
 if(matrix.empty()) return 0;

 int m = matrix[0].size();
 vector<int> height(m, 0);
 int maxArea = 0;
```

```

for(auto &row : matrix) {
 for(int i = 0; i < m; i++) {
 if(row[i] == '1') height[i]++;
 else height[i] = 0;
 }
 maxArea = max(maxArea, largestRectangleArea(height));
}
return maxArea;
};

int main() {
 vector<vector<char>> matrix = {
 {'1','0','1','0','0'},
 {'1','0','1','1','1'},
 {'1','1','1','1','1'},
 {'1','0','0','1','0'}
 };

 Solution sol;
 cout << sol.maximalRectangle(matrix);
 return 0;
}

```

---

Time Complexity:  $O(N * M)$

For each row ( $N$ ), we process the histogram in  $O(M)$ .

Space Complexity:  $O(M)$

Used for the height array and stack.

# Count Square Submatrices with All 1s | DP on Rectangles | (DP - 56)

We are given a binary matrix of size  $n \times m$  that contains only 0s and 1s.

Our task is to count how many **square submatrices** exist such that **all elements inside the square are 1**.

A square submatrix means:

- Same number of rows and columns
- All values inside are 1

Example:

```
matrix =
[[0,1,1,1],
 [1,1,1,1],
 [0,1,1,1]]
```

There are:

- 10 squares of size 1
  - 4 squares of size 2
  - 1 square of size 3
- Total = 15

---

## Approach

### Algorithm

Instead of checking every possible square (which is slow), we use Dynamic Programming.

We create a DP matrix where:

- $dp[i][j]$  = size of the **largest square of all 1s** that ends at cell  $(i, j)$  as the bottom-right corner.

Steps:

1. Create a DP matrix of the same size as the input matrix.
  2. First row and first column:
    - $dp[i][j] = \text{matrix}[i][j]$
    - Because a square larger than size 1 cannot be formed there.
  3. For remaining cells:
    - If  $\text{matrix}[i][j] == 0 \rightarrow dp[i][j] = 0$
    - If  $\text{matrix}[i][j] == 1 \rightarrow$   
 $dp[i][j] = 1 + \min($   
 $dp[i-1][j], // \text{top}$   
 $dp[i][j-1], // \text{left}$   
 $dp[i-1][j-1] // \text{top-left}$   
 $)$
  4. Why this works:
    - A square of size  $k$  can only be formed if all three neighboring squares can form at least size  $k-1$ .
  5. Every  $dp[i][j]$  contributes  $dp[i][j]$  squares:
    - If  $dp[i][j] = 3$ , it means there are squares of size 1, 2, and 3 ending at that cell.
  6. Sum all values of the DP matrix to get the answer.
- 

## Code

```
#include <bits/stdc++.h>
using namespace std;

int countSquares(int n, int m, vector<vector<int>> &arr) {
```

```

vector<vector<int>> dp(n, vector<int>(m, 0));

// First row
for(int j = 0; j < m; j++)
 dp[0][j] = arr[0][j];

// First column
for(int i = 0; i < n; i++)
 dp[i][0] = arr[i][0];

// Fill DP table
for(int i = 1; i < n; i++) {
 for(int j = 1; j < m; j++) {
 if(arr[i][j] == 0)
 dp[i][j] = 0;
 else
 dp[i][j] = 1 + min(dp[i-1][j],
 min(dp[i][j-1], dp[i-1][j-1]));
 }
}

// Count total squares
int total = 0;
for(int i = 0; i < n; i++) {
 for(int j = 0; j < m; j++) {
 total += dp[i][j];
 }
}

return total;
}

int main() {
 vector<vector<int>> arr = {
 {0,1,1,1},
 {1,1,1,1},
 {0,1,1,1}
 };
}

```

```
int n = 3, m = 4;
cout << "The number of squares: " << countSquares(n, m, arr);
return 0;
}
```

---

## Complexity Analysis

Time Complexity:  $O(n \times m)$

Each cell is processed exactly once.

Space Complexity:  $O(n \times m)$

A DP table of the same size as the matrix is used.

# **Tries**

# 1. Implement Trie – 1

We need to implement a Trie (Prefix Tree) that supports the following operations:

- **insert(word):** Insert a word into the Trie.
- **search(word):** Return true if the exact word exists in the Trie.
- **startsWith(prefix):** Return true if there is any word in the Trie that starts with the given prefix.

A Trie stores characters level by level, where each node represents a character. This helps in fast prefix-based searching.

Example:

If we insert "apple" and "app":

- Searching "apple" → true
  - Searching "app" → true
  - Searching "ap" → false
  - startsWith "ap" → true
- 

## Approach

### Algorithm

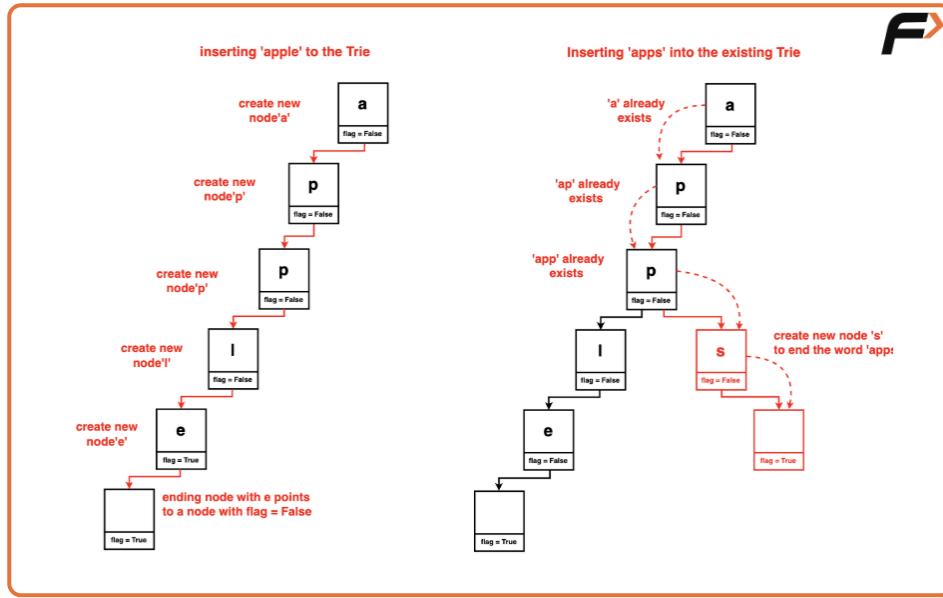
A Trie is made of nodes. Each node contains:

- An array of size 26 to store links for characters 'a' to 'z'
- A boolean flag to mark whether a word ends at this node

### Insert operation

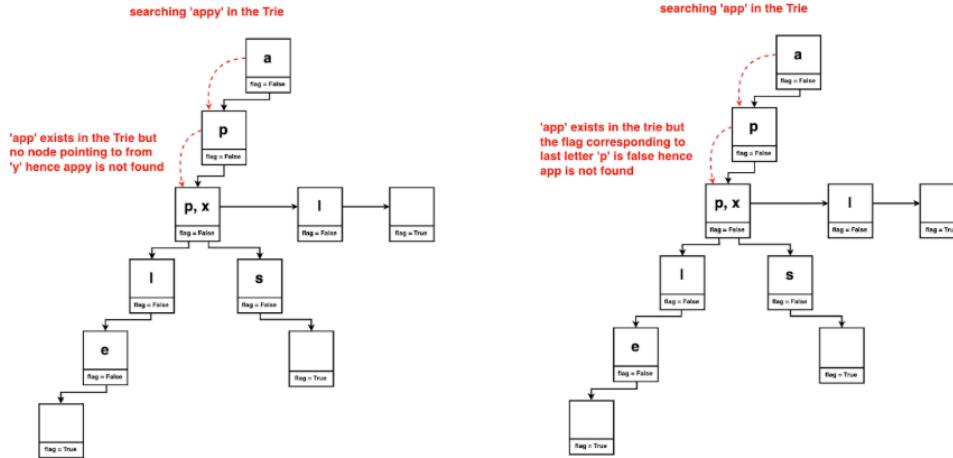
1. Start from the root node.

2. For each character in the word:
  - If the child node for that character does not exist, create it.
  - Move to the child node.
3. After inserting all characters, mark the last node as end of word.



## Search operation

1. Start from the root node.
2. For each character in the word:
  - If the child node does not exist, return false.
  - Move to the child node.
3. After all characters are processed, check if the current node is marked as end of word.

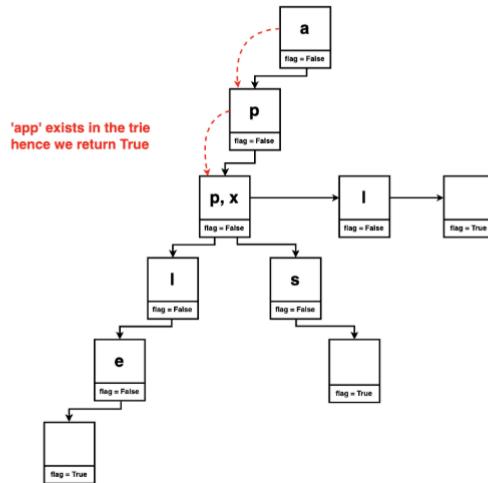


## Prefix check (startsWith)

1. Start from the root node.
2. For each character in the prefix:
  - o If the child node does not exist, return false.
  - o Move to the child node.
3. If all characters are found, return true.



searching if prefix 'app' exists in the Trie



## Code

```
#include <bits/stdc++.h>
using namespace std;

// Node structure for Trie
struct Node {
 Node* links[26] = {nullptr};
 bool flag = false;

 bool containsKey(char ch) {
 return links[ch - 'a'] != nullptr;
 }

 void put(char ch, Node* node) {
 links[ch - 'a'] = node;
 }

 Node* get(char ch) {
 return links[ch - 'a'];
 }
}
```

```

void setEnd() {
 flag = true;
}

bool isEnd() {
 return flag;
}
};

// Trie class
class Trie {
private:
 Node* root;

public:
 Trie() {
 root = new Node();
 }

 void insert(string word) {
 Node* node = root;
 for(char ch : word) {
 if(!node->containsKey(ch)) {
 node->put(ch, new Node());
 }
 node = node->get(ch);
 }
 node->setEnd();
 }

 bool search(string word) {
 Node* node = root;
 for(char ch : word) {
 if(!node->containsKey(ch)) {
 return false;
 }
 node = node->get(ch);
 }
 }
}

```

```

 return node->isEnd();
 }

bool startsWith(string prefix) {
 Node* node = root;
 for(char ch : prefix) {
 if(!node->containsKey(ch)) {
 return false;
 }
 node = node->get(ch);
 }
 return true;
}

int main() {
 Trie trie;
 trie.insert("apple");
 cout << trie.search("apple") << endl; // true
 cout << trie.search("app") << endl; // false
 cout << trie.startsWith("app") << endl; // true
 trie.insert("app");
 cout << trie.search("app") << endl; // true
 return 0;
}

```

---

## Complexity Analysis

### Time Complexity

- Insert:  $O(L)$ , where L is the length of the word
- Search:  $O(L)$
- startsWith:  $O(L)$

This is because we process one character at a time.

## Space Complexity

- $O(N)$ , where  $N$  is the total number of characters stored in the Trie  
Each character may create a new node in the Trie.

# 2. Implement Trie – II (with frequency counts)

This is the **extended Trie** version where, unlike Trie-I, we must **count occurrences** of words and prefixes and also **erase** words correctly.

---

## What's new compared to Trie-I?

Each Trie node now maintains **two counters**:

1. **cntEndWith** →  
How many words **end exactly at this node**
2. **cntPrefix** →  
How many words **pass through this node** (i.e., have this prefix)

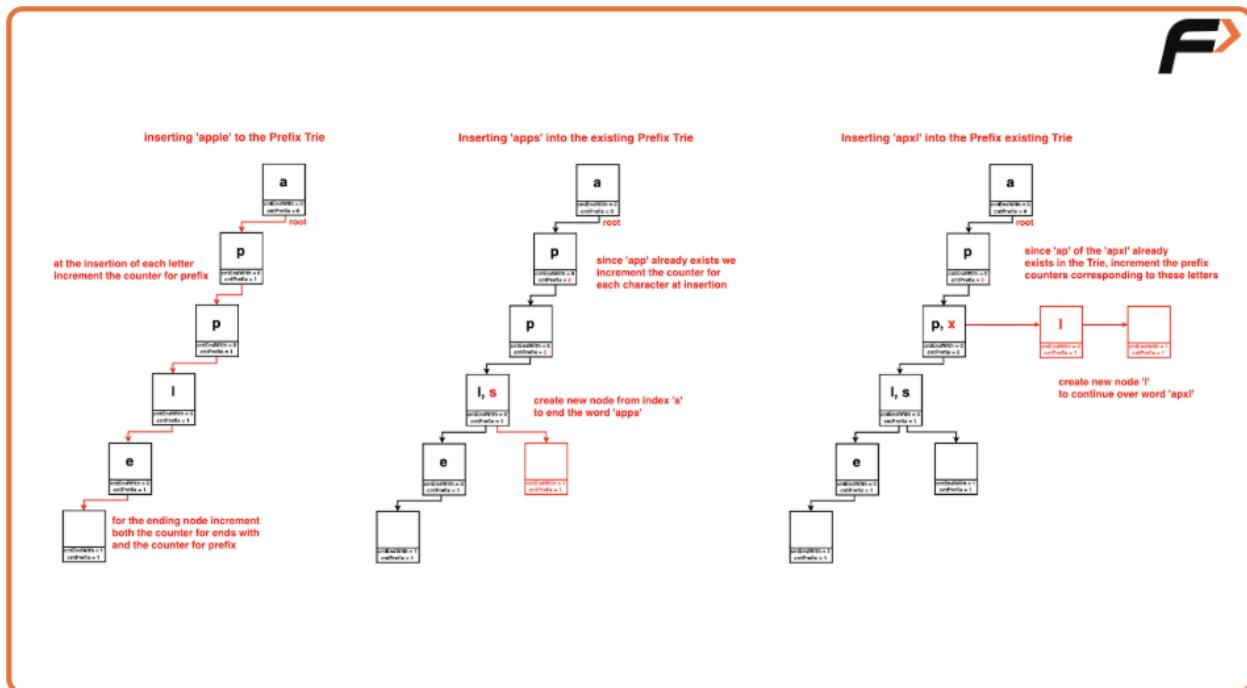
These counters allow us to:

- Count duplicate insertions
  - Count how many words share a prefix
  - Erase words safely without deleting unrelated prefixes
-

# Core Operations Explained

## 1 Insert(word)

- Traverse character by character
- Create nodes if missing
- **Increment cntPrefix for every visited node**
- At the last node, **increment cntEndWith**

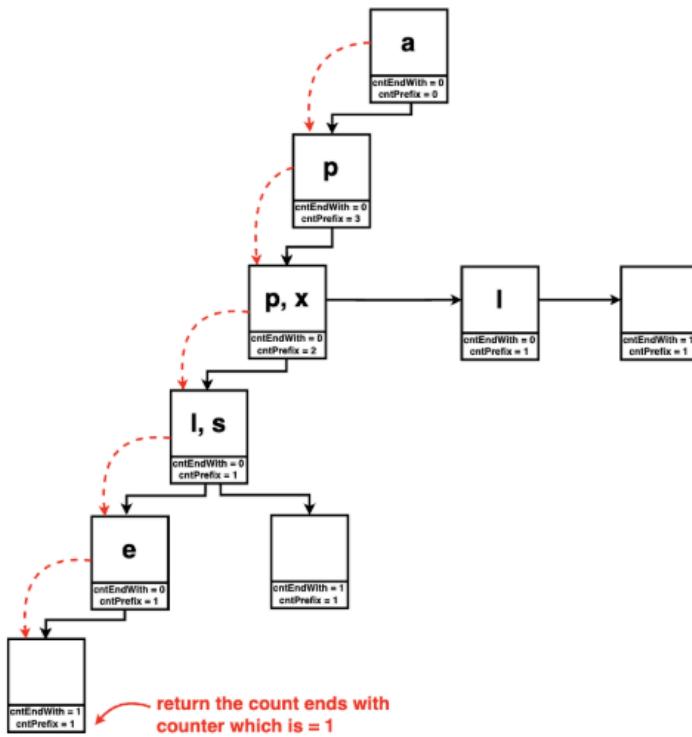


## 2 countWordsEqualTo(word)

- Traverse the word
- If path breaks → return 0
- Otherwise → return cntEndWith of last node



Count number of words equal to 'apple'

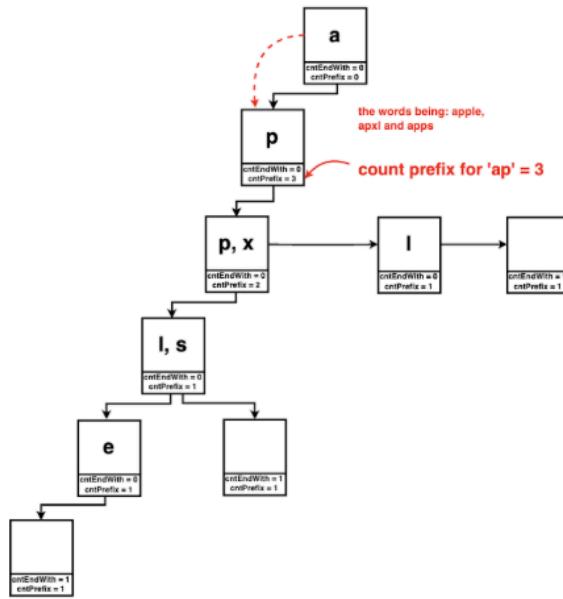


### 3 countWordsStartingWith(prefix)

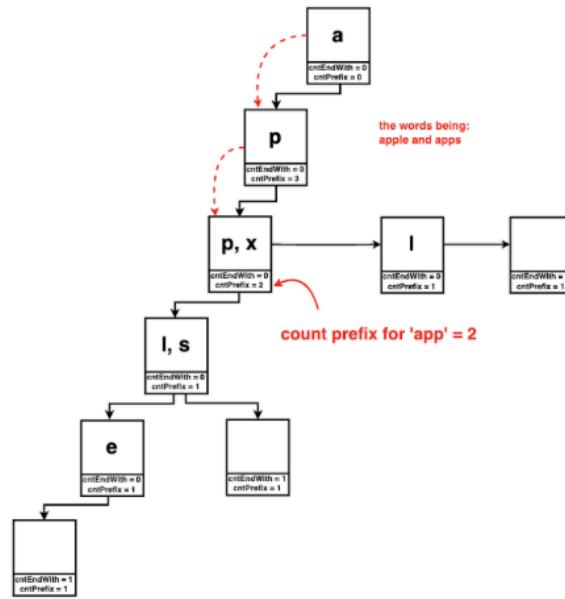
- Traverse the prefix
- If path breaks → return 0
- Otherwise → return cntPrefix of last node



Count number of words starting with 'ap'



Count number of words starting with 'app'



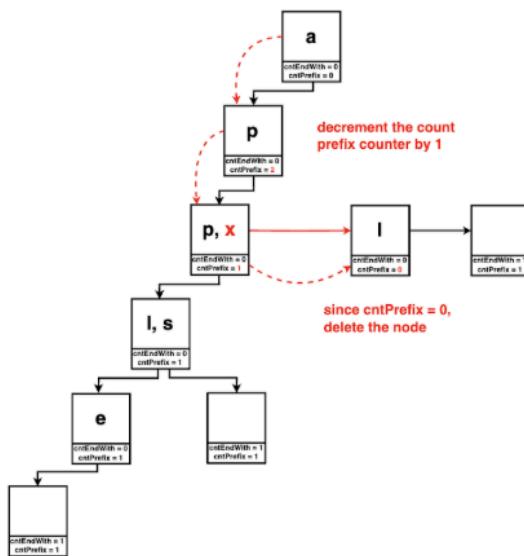
#### 4] erase(word)

- Traverse the word
- **Decrement cntPrefix at every node**
- At the last node, **decrement cntEndWith**

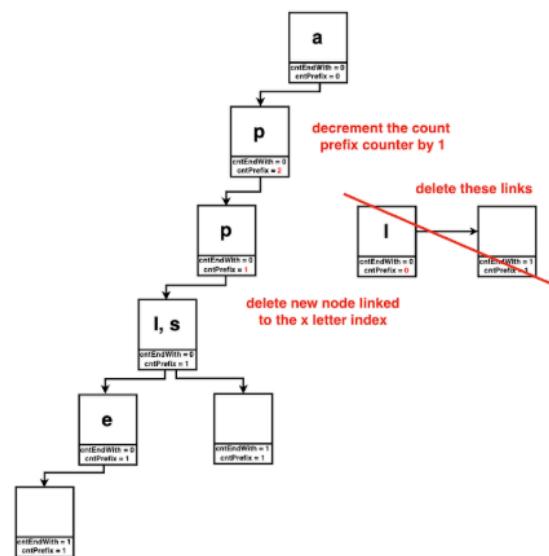
We do **not delete nodes**, because other words may still depend on them.



Erase the word 'apxl'



Erase the word 'apxl'




---

```
#include <bits/stdc++.h>
using namespace std;

struct Node {
 Node* links[26] = {nullptr};
 int cntEndWith = 0; // words ending here
 int cntPrefix = 0; // words passing through

 bool containsKey(char ch) {
 return links[ch - 'a'] != nullptr;
 }

 Node* get(char ch) {
 return links[ch - 'a'];
 }

 void put(char ch, Node* node) {
 links[ch - 'a'] = node;
 }
}
```

```

};

class Trie {
private:
 Node* root;

public:
 Trie() {
 root = new Node();
 }

 void insert(string word) {
 Node* node = root;
 for (char ch : word) {
 if (!node->containsKey(ch)) {
 node->put(ch, new Node());
 }
 node = node->get(ch);
 node->cntPrefix++;
 }
 node->cntEndWith++;
 }

 int countWordsEqualTo(string word) {
 Node* node = root;
 for (char ch : word) {
 if (!node->containsKey(ch)) return 0;
 node = node->get(ch);
 }
 return node->cntEndWith;
 }

 int countWordsStartingWith(string prefix) {
 Node* node = root;
 for (char ch : prefix) {
 if (!node->containsKey(ch)) return 0;
 node = node->get(ch);
 }
 }
}

```

```

 return node->cntPrefix;
 }

void erase(string word) {
 Node* node = root;
 for (char ch : word) {
 if (!node->containsKey(ch)) return;
 node = node->get(ch);
 node->cntPrefix--;
 }
 node->cntEndWith--;
}
};

int main() {
 Trie trie;
 trie.insert("apple");
 trie.insert("apple");

 cout << trie.countWordsEqualTo("apple") << endl; // 2
 cout << trie.countWordsStartingWith("app") << endl; // 2

 trie.erase("apple");

 cout << trie.countWordsEqualTo("apple") << endl; // 1
 cout << trie.countWordsStartingWith("app") << endl; // 1
}

```

---

## Time & Space Complexity

### ⌚ Time Complexity

For all operations:

- $O(L)$  where  $L$  = length of the word or prefix

## Space Complexity

- **O(total characters inserted)**  
Worst case: every character creates a new Trie node

# 3. Longest Valid Word with All Prefixes

Given an array of strings, you have to find the **longest string** such that **all of its prefixes are also present** in the array.

If more than one string has the same maximum length, return the **lexicographically smallest** one.

A prefix means a substring starting from the first character. The word itself is also considered a prefix.

### Example

arr = ["ab", "a", "abc", "abd"]

Both "abc" and "abd" are valid because their prefixes exist.

"abc" → "a", "ab", "abc"

"abd" → "a", "ab", "abd"

Both have same length, so lexicographically smaller "abc" is the answer.

---

### Approach 1

Using Sorting and Binary Search

### Algorithm

1. Sort the array lexicographically so binary search can be used.
2. For each word in the array:
  - o Build its prefixes one by one.

- For every prefix, check if it exists in the array using binary search.
3. If all prefixes exist, the word is valid.
  4. Update the result if:
    - The word is longer than current result, or
    - Length is same but word is lexicographically smaller.

### Code

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

string longestString(vector<string>& arr) {
 sort(arr.begin(), arr.end());
 string result;

 for (string& word : arr) {
 bool isValid = true;
 string prefix;

 for (char ch : word) {
 prefix += ch;
 if (!binary_search(arr.begin(), arr.end(), prefix)) {
 isValid = false;
 break;
 }
 }

 if (isValid &&
 (word.size() > result.size() ||
 (word.size() == result.size() && word < result))) {
 result = word;
 }
 }
 return result;
}

int main() {
 vector<string> arr = {"ab", "a", "abc", "abd"};
 cout << longestString(arr);
 return 0;
}
```

## Complexity Analysis

- **Time Complexity:**  $O(n * k^2 * \log n)$   
For each word ( $n$ ), we generate  $k$  prefixes, and each binary search takes  $O(k \log n)$  due to string comparison.
  - **Space Complexity:**  $O(1)$   
No extra data structure used apart from variables.
- 

## Approach 2

Rabin-Karp Based Prefix Validation

### Algorithm

1. Use double hashing to reduce collision chances.
2. Compute hash pairs for all words and store them in a set.
3. For each word:
  - o Generate prefix hashes incrementally.
  - o Check if each prefix hash exists in the set.
4. If all prefixes exist, update the result using length and lexicographical order.

### Code

```
#include <iostream>
#include <vector>
#include <set>
using namespace std;

const int base1 = 31, mod1 = 1e9 + 7;
const int base2 = 37, mod2 = 1e9 + 9;

vector<int> computeHash(string& s) {
 int h1 = 0, h2 = 0, p1 = 1, p2 = 1;
 for (char ch : s) {
 int val = ch - 'a' + 1;
 h1 = (h1 + val * 1LL * p1) % mod1;
 h2 = (h2 + val * 1LL * p2) % mod2;
 p1 = (p1 * 1LL * base1) % mod1;
 p2 = (p2 * 1LL * base2) % mod2;
 }
 return {h1,h2};
}

bool allPrefixExist(string& word, set<vector<int>>& hashSet) {
```

```

int h1 = 0, h2 = 0, p1 = 1, p2 = 1;
for (char ch : word) {
 int val = ch - 'a' + 1;
 h1 = (h1 + val * 1LL * p1) % mod1;
 h2 = (h2 + val * 1LL * p2) % mod2;
 if (hashSet.find({h1,h2}) == hashSet.end())
 return false;
 p1 = (p1 * 1LL * base1) % mod1;
 p2 = (p2 * 1LL * base2) % mod2;
}
return true;
}

string longestString(vector<string>& arr) {
set<vector<int>> hashSet;
for (string& word : arr)
 hashSet.insert(computeHash(word));

string result;
for (string& word : arr) {
 if (allPrefixExist(word, hashSet)) {
 if (word.size() > result.size() ||
 (word.size() == result.size() && word < result)) {
 result = word;
 }
 }
}
return result;
}

int main() {
vector<string> arr = {"ab","a","abc","abd"};
cout << longestString(arr);
return 0;
}

```

## Complexity Analysis

- **Time Complexity:**  $O(n * k * \log n)$   
Each prefix check involves set lookup.
- **Space Complexity:**  $O(n)$   
Hash values of all words are stored.

## Approach 3

Trie-Based Prefix Validation

### Algorithm

1. Create a Trie node with 26 children and an `isEnd` flag.
2. Insert all words into the Trie.
3. For each word:
  - o Traverse the Trie character by character.
  - o At every step, ensure the node exists and `isEnd` is true.
4. Update the result using length and lexicographical comparison.

### Code

```
#include <iostream>
#include <vector>
using namespace std;

struct TrieNode {
 TrieNode* children[26];
 bool isEnd;
 TrieNode() {
 isEnd = false;
 for (int i = 0; i < 26; i++) children[i] = nullptr;
 }
};

class Trie {
public:
 TrieNode* root;
 Trie() {
 root = new TrieNode();
 }

 void insert(string& word) {
 TrieNode* node = root;
 for (char ch : word) {
 int idx = ch - 'a';
 if (!node->children[idx])
 node->children[idx] = new TrieNode();
 node = node->children[idx];
 }
 node->isEnd = true;
 }
}
```

```

bool allPrefixesExist(string& word) {
 TrieNode* node = root;
 for (char ch : word) {
 int idx = ch - 'a';
 node = node->children[idx];
 if (!node || !node->isEnd)
 return false;
 }
 return true;
};

string longestString(vector<string>& arr) {
 Trie trie;
 for (string& word : arr)
 trie.insert(word);

 string result;
 for (string& word : arr) {
 if (trie.allPrefixesExist(word)) {
 if (word.size() > result.size() ||
 (word.size() == result.size() && word < result)) {
 result = word;
 }
 }
 }
 return result;
}

int main() {
 vector<string> arr = {"ab", "a", "abc", "abd"};
 cout << longestString(arr);
 return 0;
}

```

## Complexity Analysis

- **Time Complexity:**  $O(n * k)$   
Trie insertion and prefix checking both depend on word length.
- **Space Complexity:**  $O(n * k)$   
Trie stores all characters of all words.

# 4. Number of Distinct Substrings in a String Using Trie

Given a string S, find the **number of distinct substrings**, including the **empty substring**. A substring is formed by removing zero or more characters from the **start and end** of the string.

Two substrings are different if their lengths are different or at least one character differs at the same index.

## Example

s = "aba"

Distinct substrings are: "a", "ab", "aba", "b", "ba", ""

Output = 6

s = "abc"

Distinct substrings are: "a", "ab", "abc", "b", "bc", "c", ""

Output = 7

---

## Approach 1

Brute-Force Using Set

### Algorithm

1. Create an empty set to store distinct substrings.
2. Fix a starting index i from 0 to n-1.
3. From index i, build substrings by extending till the end using index j.
4. Insert each generated substring into the set.
5. After all substrings are inserted, count the size of the set.
6. Add 1 to include the empty substring.

### Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
 set<string> countDistinctSubstrings(const string& s) {
 set<string> st;
 int n = s.length();
```

```

for (int i = 0; i < n; i++) {
 string str = "";
 for (int j = i; j < n; j++) {
 str += s[j];
 st.insert(str);
 }
}
return st;
}

int main() {
 string s = "striver";
 Solution sol;

 set<string> substrings = sol.countDistinctSubstrings(s);
 int count = substrings.size();

 cout << "Number of distinct substrings: " << count + 1 << endl;
 return 0;
}

```

## Complexity Analysis

- **Time Complexity:**  $O(N^2)$   
All possible substrings are generated using two loops.
  - **Space Complexity:**  $O(N^2)$   
The set can store up to  $N^2$  substrings in the worst case.
- 

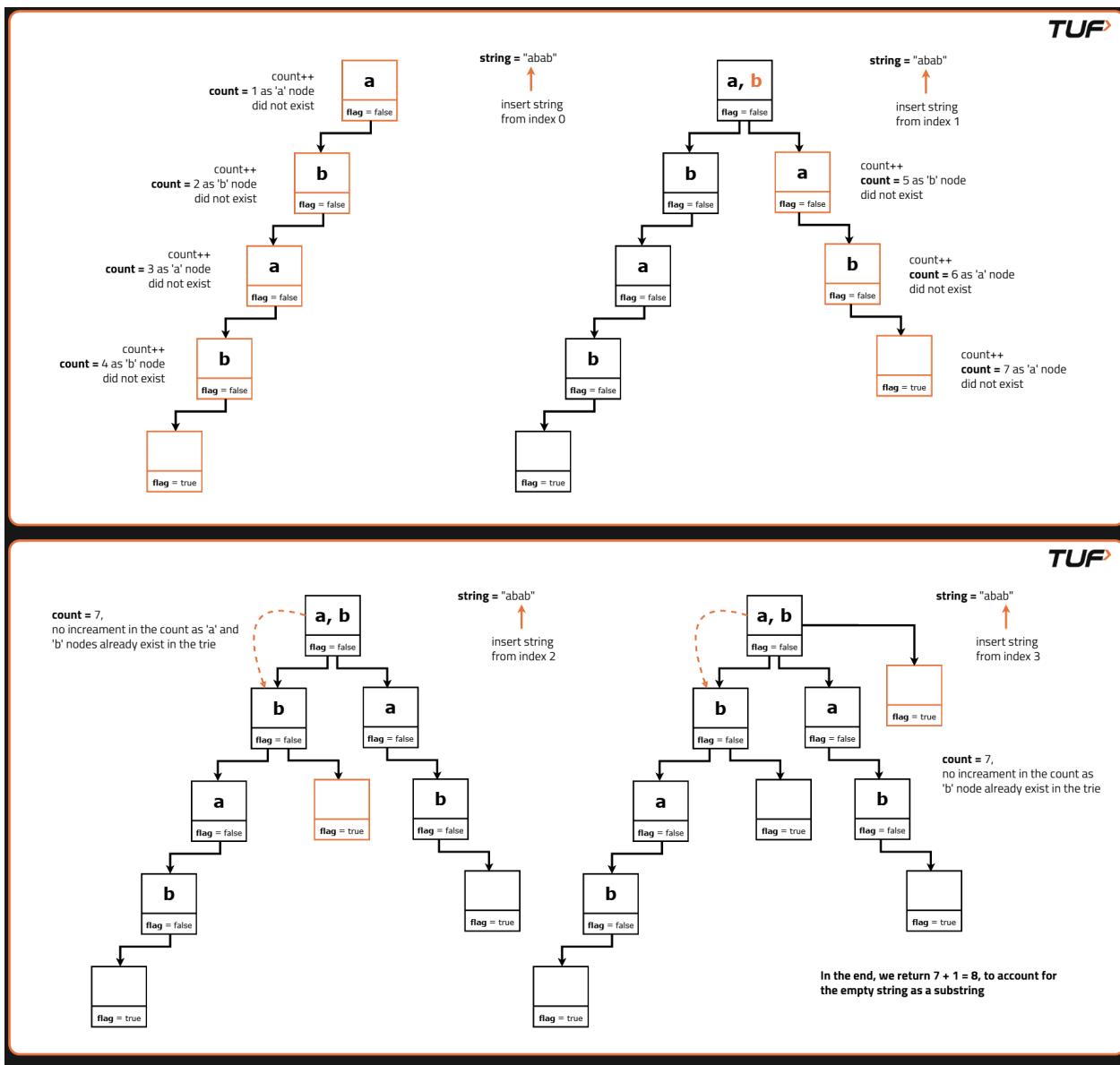
## Approach 2

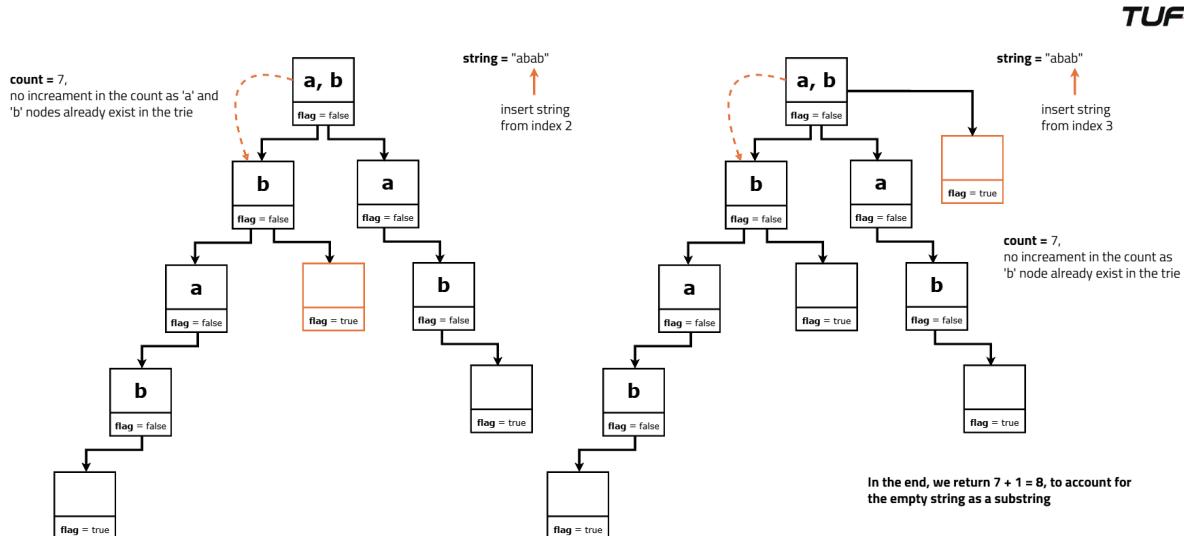
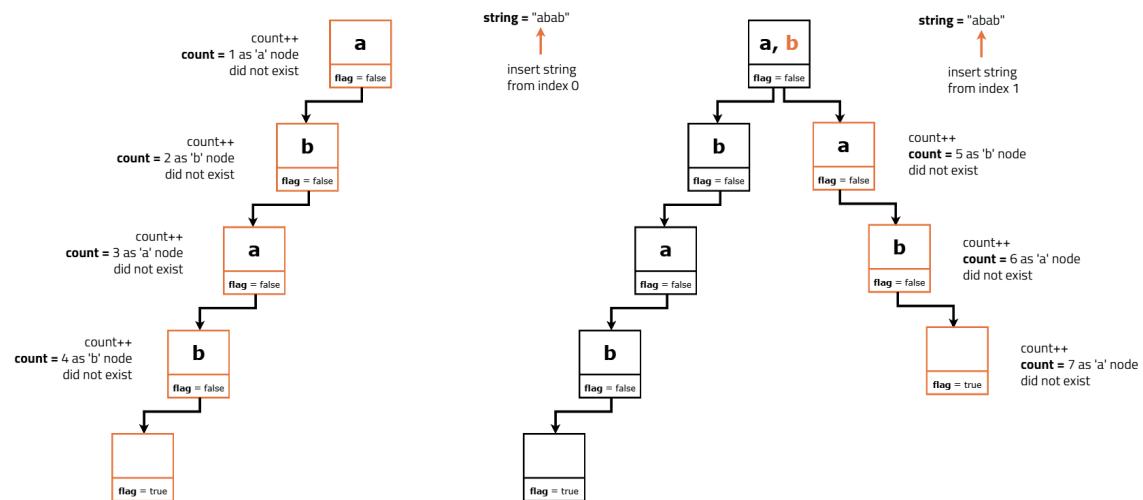
Trie-Based Optimal Approach

### Algorithm

1. Create a Trie with a root node.
2. Initialize a counter to count distinct substrings.
3. For each starting index  $i$ :
  - Start from the root node.
  - For every character  $j$  from  $i$  to end:

- If the character path does not exist in the Trie, create it and increment the counter.
  - Move to the next Trie node.
4. Each new Trie node represents a **new distinct substring**.
5. Add 1 at the end to count the empty substring.





## Code

```
#include <bits/stdc++.h>
using namespace std;

struct Node {
 Node* links[26];
 bool flag = false;

 bool containsKey(char ch) {
 return links[ch - 'a'] != NULL;
 }
}
```

```

Node* get(char ch) {
 return links[ch - 'a'];
}

void put(char ch, Node* node) {
 links[ch - 'a'] = node;
}
};

class Solution {
public:
 int countDistinctSubstrings(string &s) {
 Node* root = new Node();
 int cnt = 0;
 int n = s.size();

 for (int i = 0; i < n; i++) {
 Node* node = root;
 for (int j = i; j < n; j++) {
 if (!node->containsKey(s[j])) {
 node->put(s[j], new Node());
 cnt++;
 }
 node = node->get(s[j]);
 }
 }
 return cnt + 1;
 }
};

int main() {
 string s = "striver";
 Solution sol;
 cout << "Number of distinct substrings: " << sol.countDistinctSubstrings(s) << endl;
 return 0;
}

```

## Complexity Analysis

- **Time Complexity:**  $O(N^2)$   
All substrings are processed once, and Trie insertion is  $O(1)$  per character.
- **Space Complexity:**  $O(N^2)$   
Trie stores nodes for all unique substrings.

# 5. Bit PreRequisites for TRIE Problems

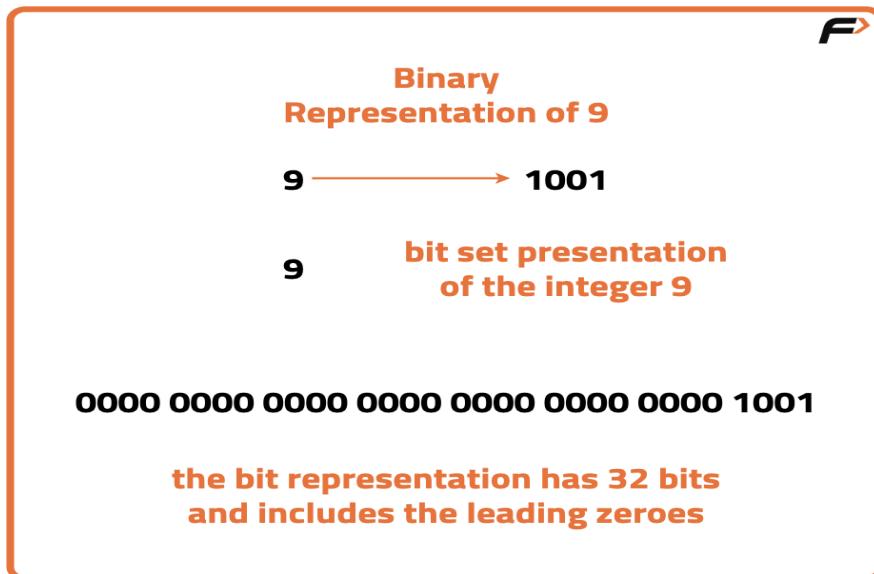
Up until now we have looked at Trie and the various functions that can be performed on it like searching, insertion and finding prefix. An important application of bitwise operations in tries is XOR (exclusive OR). XOR is commonly used in tries to efficiently perform operations like bitwise toggling or finding the XOR of elements. This operation is particularly useful in scenarios where we need to compare or manipulate binary representations of data.

---

## Basics of Bit Manipulation

### Binary Representation

When representing the integer 9 in binary, it's represented as "1001". However, in Trie data structures, each bit position represents a decision point in the trie, so the entire bit set is stored ie. "0000 ... 1001", including leading zeros.

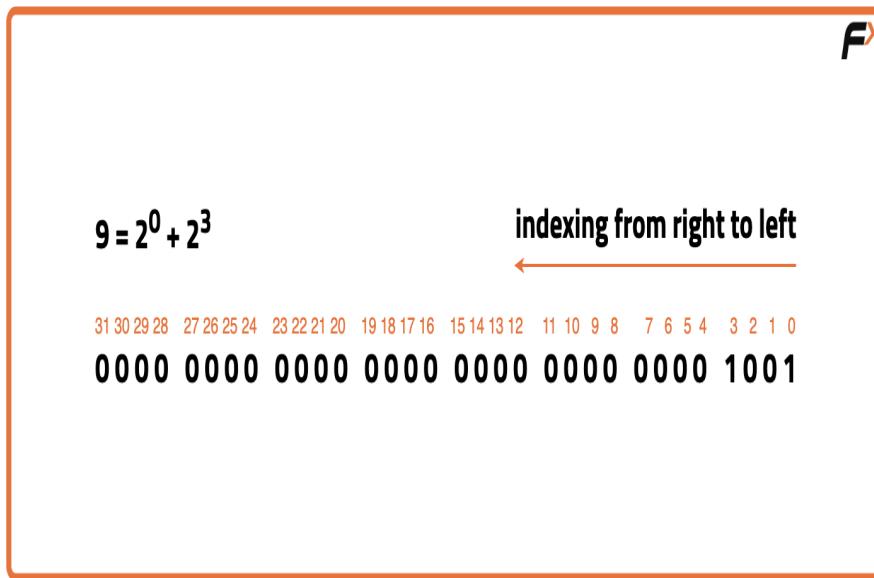


Therefore, in a trie, the binary representation of 9 would indeed be stored as "0000 .... 1001" comprising of a total of 32 bits, where each bit position represents a node in the trie, regardless of the leading zeroes.

---

## Indexing in Binary Representation

In binary representation, indexing typically starts from the rightmost bit, also known as the least significant bit (LSB). This means that the rightmost bit has an index of 0, and the index increases as you move towards the left, with the leftmost bit having the highest index. This convention is consistent with the positional numbering system in binary, where each bit represents a power of 2, starting from  $2^0$  (1) on the right and increasing by powers of 2 as you move left. For example, let's consider the binary representation of the integer 9: "0000 .... 1001". In this representation:



The right bit (the first bit from the right) has an index of 0. The second rightmost bit has an index of 1. This pattern continues, with each subsequent bit to the right having an index one higher than the previous bit. This indexing convention is essential for various bitwise operations, including those used in tries, where the position of each bit matters for traversal and manipulation of the data structure.

---

## XOR Operation

The XOR (exclusive OR) operation is a fundamental bitwise operation that compares corresponding bits of two operands. The result of the XOR operation is 1 if the bits are different and 0 if they are the same.

Here's how the XOR operation works:

- If both bits are 0, the result is 0.
- If both bits are 1, the result is 0.
- If one bit is 0 and the other is 1, the result is 1.

In other words, the XOR operation yields a 1 only if the bits being compared are different; otherwise, it yields a 0.

Here's the XOR truth table:

| Input 1 | Input 2 | Output |
|---------|---------|--------|
| 0       | 0       | 0      |
| 0       | 1       | 1      |
| 1       | 0       | 1      |
| 1       | 1       | 0      |

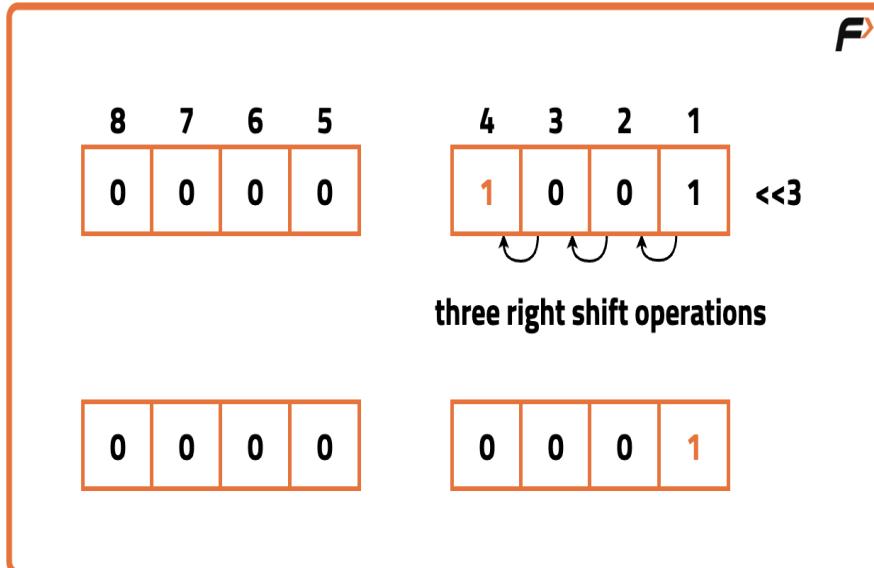
Important Points to Note are:

- If there are an even number of 1s (e.g., 0, 2, 4...), the XOR operation will result in 0. This is because for every 1, there must be another 1 paired with it to produce a 0 output (e.g.,  $1 \text{ XOR } 1 = 0$ ).
- If there are an odd number of 1s (e.g., 1, 3, 5...), the XOR operation will result in 1. This is because there will always be one unpaired 1 left, resulting in a 1 output (e.g.,  $1 \text{ XOR } 1 \text{ XOR } 1 = 1$ ).

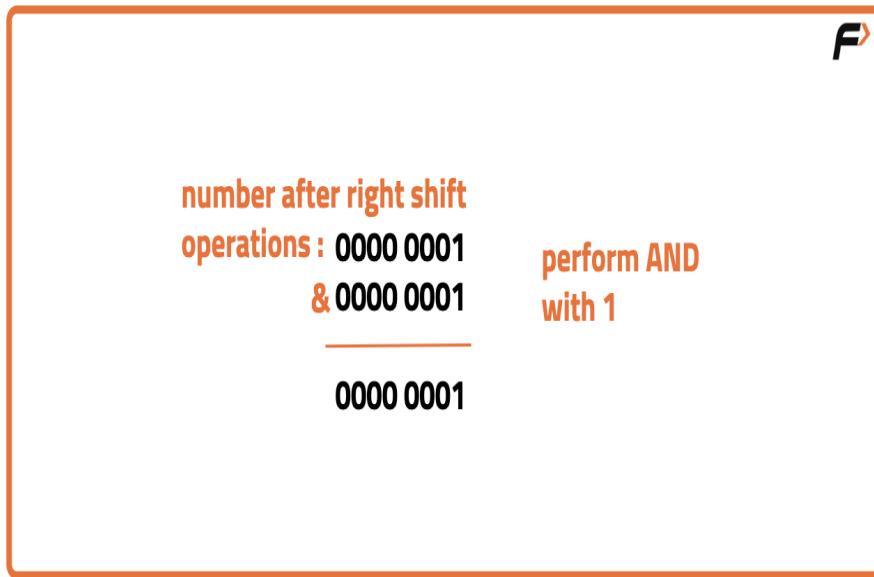
---

### How to Check if a Bit is set or not

To check if a particular bit is set (i.e., has a value of 1) or not in a binary number, we use a bitwise AND operation.



- Right shift the number by the index of the bit you want to check.
- Perform a bitwise AND operation with 1.
- Check if the result is equal to 1.



### How to Turn On a Bit

To "turn on" a bit means to set it to the value of 1 within a binary number. This operation involves modifying the binary representation of a number such that the targeted bit becomes 1 while leaving other bits unchanged. Turning on a bit is often done using bitwise OR operation.



|   |   |   |   |
|---|---|---|---|
| 7 | 6 | 5 | 4 |
| 0 | 0 | 0 | 0 |

|   |   |   |   |
|---|---|---|---|
| 3 | 2 | 1 | 0 |
| 1 | 0 | 0 | 1 |

Example : Turn on bit at index 2                     $1 \ll 2$

Left shift 1 by the index of  
the bit to turn on                                      0000 0100

- Create a bitmask with the bit you want to turn on set to 1 and all other bits set to 0.
- Perform a bitwise OR operation between the original number and the bitmask.



Original Number : 0000 1001 perform OR to  
bitmask : || 0000 0100 turn on index 2  
\_\_\_\_\_  
0000 1101

|   |   |   |   |
|---|---|---|---|
| 7 | 6 | 5 | 4 |
| 0 | 0 | 0 | 0 |

|   |   |   |   |
|---|---|---|---|
| 3 | 2 | 1 | 0 |
| 1 | 1 | 0 | 1 |

bit at index 2 has been turned on

# 6. Maximum XOR of Two Numbers in an Array

Given an integer array `nums`, you have to find the **maximum XOR value** that can be obtained using `nums[i] XOR nums[j]` where  $0 \leq i \leq j < n$ .

XOR (exclusive OR) gives 1 when the two bits are different and 0 when they are the same. To maximize XOR, we try to make as many bit positions different as possible between two numbers.

## Example

`nums = [3, 9, 10, 5, 1]`

Binary:

- $10 \rightarrow 1010$
  - $5 \rightarrow 0101$
- $\text{XOR} \rightarrow 1111 = 15$
- 

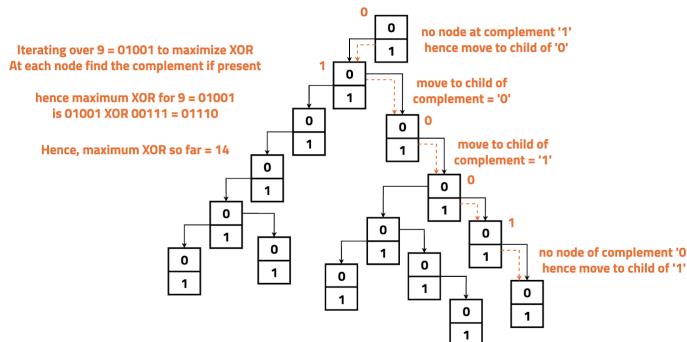
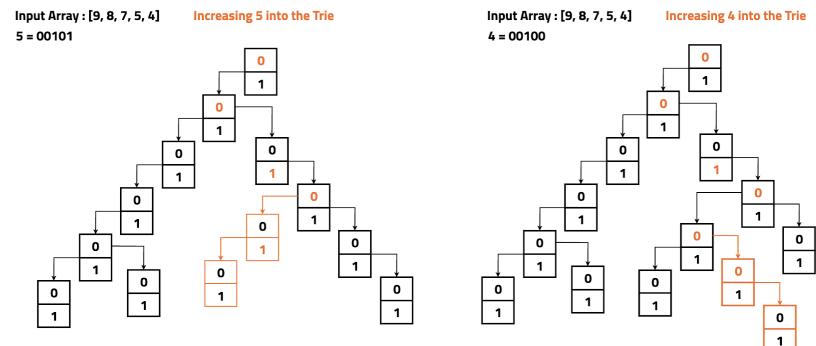
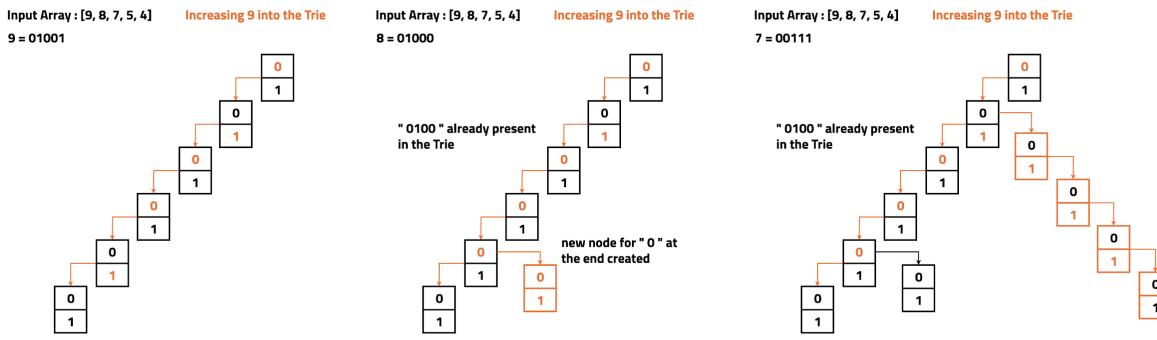
## Approach

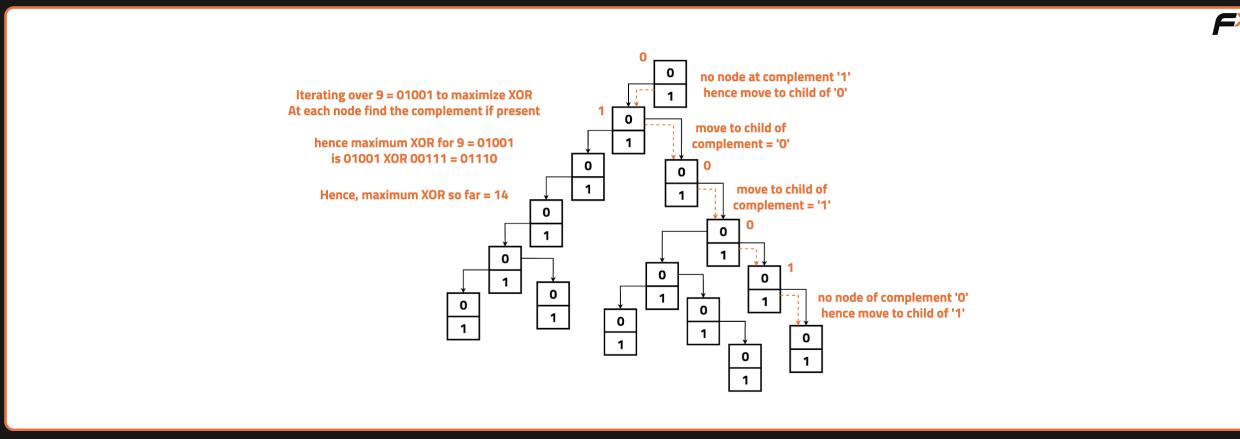
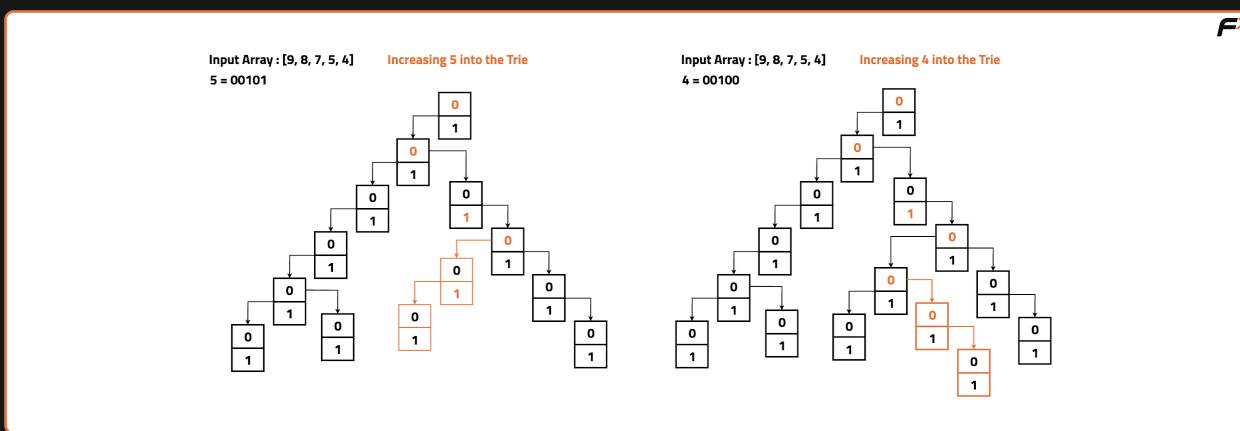
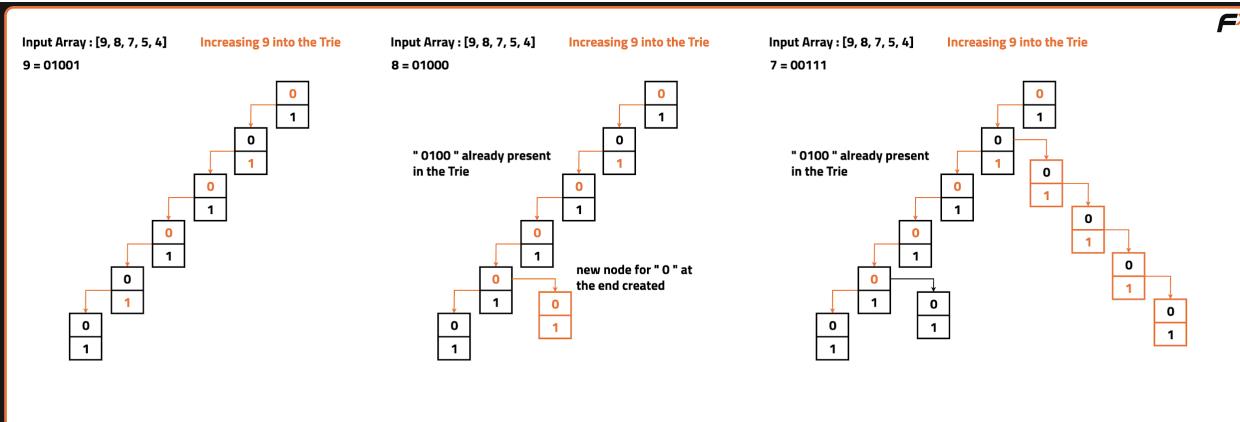
Using Trie (Binary Trie)

## Algorithm

1. Create a Trie where each node has two links: 0 and 1.
2. Insert every number into the Trie by processing its bits from **most significant bit (31)** to **least significant bit (0)**.
3. To find maximum XOR for a number:
  - Traverse the Trie again from bit 31 to 0.
  - For each bit, try to move to the **opposite bit** ( $1 - \text{bit}$ ).
  - If the opposite bit exists, set the corresponding bit in XOR result.
  - Otherwise, move to the same bit.
4. Repeat this for all numbers and keep track of the maximum XOR value.

This works because choosing opposite bits at higher positions gives a larger XOR value.





```
#include <bits/stdc++.h>
```

```
using namespace std;
```

```
class Node {
```

```

public:
 Node* links[2];

 bool containsKey(int bit) {
 return links[bit] != NULL;
 }

 Node* get(int bit) {
 return links[bit];
 }

 void put(int bit, Node* node) {
 links[bit] = node;
 }
};

class Solution {
public:
 Node* root;

 Solution() {
 root = new Node();
 }
}

```

```

void insert(int num) {

 Node* node = root;

 for (int i = 31; i >= 0; i--) {

 int bit = (num >> i) & 1;

 if (!node->containsKey(bit)) {

 node->put(bit, new Node());

 }

 node = node->get(bit);

 }

}

```

```

int getMaxXOR(int num) {

 Node* node = root;

 int maxXor = 0;

 for (int i = 31; i >= 0; i--) {

 int bit = (num >> i) & 1;

 if (node->containsKey(1 - bit)) {

 maxXor |= (1 << i);

 node = node->get(1 - bit);

 } else {

 node = node->get(bit);

 }

 }

}

```

```

 return maxXor;

 }

int findMaximumXOR(vector<int>& nums) {
 for (int num : nums) {
 insert(num);
 }

 int maxResult = 0;
 for (int num : nums) {
 maxResult = max(maxResult, getMaxXOR(num));
 }
 return maxResult;
}

};

int main() {
 vector<int> nums = {3, 9, 10, 5, 1};
 Solution sol;
 cout << sol.findMaximumXOR(nums);
 return 0;
}

```

- **Time Complexity:**  $O(N)$   
Each number is inserted and queried in the Trie using 32 bits, which is constant.
- **Space Complexity:**  $O(N)$   
Trie stores up to 32 nodes per number in the worst case.

# 7. Maximum Xor Queries | Trie

You are given an array of **non-negative integers** and a list of queries.  
Each query is of the form **[Xi, Ai]**.

For each query, you have to find the **maximum value of (Xi XOR num)** such that:

- num is taken from the array
- num  $\leq$  Ai

If **no number in the array is  $\leq$  Ai**, return **-1** for that query.

## Example

Array = [3, 10, 5, 25, 2, 8]

Queries = [(0,1), (1,2), (3,3)]

For (0,1): no element  $\leq 1 \rightarrow$  answer = -1

For (1,2): only element  $\leq 2$  is 2  $\rightarrow 1 \text{ XOR } 2 = 3$

For (3,3): elements  $\leq 3$  are 2 and 3  $\rightarrow \max(3 \text{ XOR } 2, 3 \text{ XOR } 3) = 1$

---

## Approach

Trie + Offline Query Processing

## Algorithm

1. XOR gives maximum value when bits are different.
2. To maximize XOR efficiently, we use a **binary Trie** where:
  - o Each node has two links: 0 and 1
  - o Each path represents the binary form of a number
3. Instead of inserting all array elements at once:
  - o Sort the array
  - o Sort queries by Ai
4. Process queries one by one:
  - o Insert all array elements  $\leq$  **current Ai** into the Trie
  - o If Trie is empty, answer is -1
  - o Otherwise, traverse the Trie to find maximum XOR for Xi
5. Store answers using original query indices.

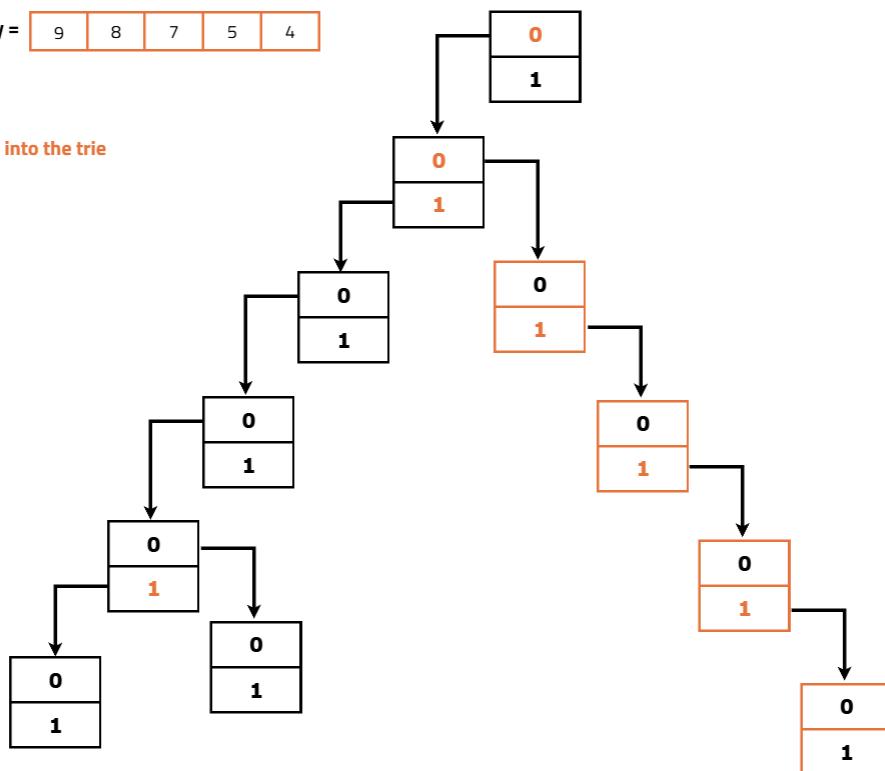
This avoids unnecessary insertions and keeps operations efficient.



Input Array = [ 9 | 8 | 7 | 5 | 4 ]

7 = 00111

Inserting 7 into the trie



```
#include <bits/stdc++.h>
```

```
using namespace std;
```

```
// Trie node
```

```
struct Node {
```

```
 Node* links[2];
```

```
 bool containsKey(int bit) {
```

```
 return links[bit] != NULL;
```

```
}
```

```

Node* get(int bit) {
 return links[bit];
}

void put(int bit, Node* node) {
 links[bit] = node;
}

// Trie class

class Trie {
 Node* root;

public:
 Trie() {
 root = new Node();
 }

 void insert(int num) {
 Node* node = root;
 for (int i = 31; i >= 0; i--) {
 int bit = (num >> i) & 1;
 if (!node->containsKey(bit)) {

```

```

 node->put(bit, new Node());
 }

 node = node->get(bit);

}

}

int findMax(int num) {

 Node* node = root;

 int maxXor = 0;

 for (int i = 31; i >= 0; i--) {

 int bit = (num >> i) & 1;

 if (node->containsKey(1 - bit)) {

 maxXor |= (1 << i);

 node = node->get(1 - bit);

 } else {

 node = node->get(bit);

 }

 }

 return maxXor;
}

};

class Solution {

```

```

public:

vector<int> maxXorQueries(vector<int>& arr,
 vector<vector<int>>& queries) {

 vector<int> ans(queries.size(), 0);
 vector<pair<int, pair<int,int>>> offlineQueries;

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

 int idx = 0;
 for (auto& q : queries) {
 offlineQueries.push_back({q[1], {q[0], idx++}});
 }

 sort(offlineQueries.begin(), offlineQueries.end());

 Trie trie;
 int i = 0, n = arr.size();

 for (auto& q : offlineQueries) {
 int m = q.first;
 int x = q.second.first;
 int index = q.second.second;

```

```

 while (i < n && arr[i] <= m) {
 trie.insert(arr[i]);
 i++;
 }

 if (i == 0)
 ans[index] = -1;
 else
 ans[index] = trie.findMax(x);
 }

 return ans;
}

};

int main() {
 vector<int> arr = {3, 10, 5, 25, 2, 8};
 vector<vector<int>> queries = {{0,1}, {1,2}, {3,3}};

 Solution obj;
 vector<int> result = obj.maxXorQueries(arr, queries);

 for (int x : result)
 cout << x << " ";
 return 0;
}

```

}

## Complexity Analysis

- **Time Complexity:**

$$O(32 \cdot N + Q \cdot \log Q + 32 \cdot Q)$$

- 32 bits for each insertion and query
- Sorting queries costs  $Q \cdot \log Q$

- **Space Complexity:**

$$O(32 \cdot N + Q)$$

- Trie stores bits of array elements
- Extra space for offline queries and answers