

Fast and Last Index in C++

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

void findFirstAndLastIndex(int arr[], int n,
int d) {
    int low = 0;
    int high = n - 1;
    int firstIndex = -1;
    int lastIndex = -1;

    // Finding the first occurrence
    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (d > arr[mid]) {
            low = mid + 1;
        } else if (d < arr[mid]) {
            high = mid - 1;
        } else {
            firstIndex = mid;
            high = mid - 1;
        }
    }

    // Finding the last occurrence
    low = 0;
    high = n - 1;
    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (d > arr[mid]) {
            low = mid + 1;
        } else if (d < arr[mid]) {
            high = mid - 1;
        } else {
            lastIndex = mid;
            low = mid + 1;
        }
    }

    cout << "First Index: " << firstIndex <<
endl;
    cout << "Last Index: " << lastIndex <<
endl;
}

int main() {
    int arr[] = {1, 5, 10, 15, 22, 33, 33, 33, 33,
33, 40, 42, 55, 66, 77, 33};
    int n = sizeof(arr) / sizeof(arr[0]);
    int d = 33;

    findFirstAndLastIndex(arr, n, d);

    return 0;
}
```

Dry Run Example (on sorted array):

Sorted version of the array:

{1, 5, 10, 15, 22, 33, 33, 33, 33, 33, 33, 40, 42, 55, 66, 77}

We want to find **first and last index of 33**.

First Occurrence:

Iteration	low	high	mid	arr[mid]	firstIndex	high (updated)
1	0	15	7	33	7	6
2	0	6	3	15		
3	4	6	5	33	5	4
4	4	4	4	22		

→ First index = 5

Last Occurrence:

Iteration	low	high	mid	arr[mid]	lastIndex	low (updated)
1	0	15	7	33	7	8
2	8	15	11	40		
3	8	10	9	33	9	10
4	10	10	10	33	10	11

→ Last index = 10

Final Output:

First Index: 5

Last Index: 10

First Index: 5	Last Index: 10

IsSorted in C++

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

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

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 arr1[] = {1, 2, 3, 4, 5, 6};
    int arr2[] = {11, 2, 3, 4, 5, 6};
    int n1 = sizeof(arr1) / sizeof(arr1[0]);
    int n2 = sizeof(arr2) / sizeof(arr2[0]);

    cout << boolalpha; // Print boolean values as
true/false
    cout << isSortedEff(arr1, n1) << endl;
    cout << isSortedEff(arr2, n2) << endl;

    cout << isSorted(arr1, n1) << endl;
    cout << isSorted(arr2, n2) << endl;

    return 0;
}
```

Check if an array is **sorted in non-decreasing order** (each element is \leq the next).

Q Difference between isSortedEff and isSorted:

Function	Approach	Time Complexity
isSortedEff	Linear scan (compare adjacent)	$O(n)$
isSorted	Brute force (nested loops)	$O(n^2)$

◇ Dry Run with Sample Arrays

Array 1: {1, 2, 3, 4, 5, 6} (Sorted)

isSortedEff(arr1, n1):

i	arr[i-1]	arr[i]	Comparison	Result
1	1	2	$2 \geq 1$	✓
2	2	3	$3 \geq 2$	✓
3	3	4	$4 \geq 3$	✓
4	4	5	$5 \geq 4$	✓
5	5	6	$6 \geq 5$	✓
→ All passed				
→ Returns: true				

isSorted(arr1, n1): Checks every pair (i, j) where $j > i$:

- For every $arr[i] \leq arr[j] \rightarrow$ all OK → **Returns: true**

Array 2: {11, 2, 3, 4, 5, 6} (Not sorted)

isSortedEff(arr2, n2):

i	arr[i-1]	arr[i]	Comparison	Result
1	11	2	$2 < 11 \times$	●
→ Early exit → Returns:				

i	arr[i-1]	arr[i]	Comparison	Result
false				
isSorted(arr2, n2):				
<ul style="list-style-type: none"> • $(0,1) \rightarrow 2 < 11 \rightarrow \text{X} \rightarrow \text{Returns: false}$ 				
█ Output: true false true false				
true false true false				

Leaders Array in C++

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

void leaders(int arr[], int n) {
    int curr = arr[n - 1];
    cout << curr << " ";

    for (int i = n - 2; i >= 0; i--) {
        if (arr[i] > curr) {
            curr = arr[i];
            cout << curr << " ";
        }
    }
}

int main() {
    int arr[] = {7, 10, 4, 10, 6, 5, 2};
    int n = sizeof(arr) / sizeof(arr[0]);

    leaders(arr, n);
    cout << endl;

    return 0;
}
```

Dry Run Table

Input array: {7, 10, 4, 10, 6, 5, 2}

We process from **right to left**:

Index	arr[i]	Current Leader (curr)	Is arr[i] > curr?	Print Leader?	Updated curr
6	2	2	-	✓	2
5	5	2	✓	✓	5
4	6	5	✓	✓	6
3	10	6	✓	✓	10
2	4	10	✗	✗	10
1	10	10	✗	✗	10
0	7	10	✗	✗	10

↙ Output (Printed from right to left):

2 5 6 10

2 5 6 10

Majority element in C++

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

int majority(int arr[], int n) {
    int res = 0, count = 1;
    for (int i = 1; i < n; i++) {
        if (arr[res] == arr[i]) {
            count++;
        } else {
            count--;
        }
        if (count == 0) {
            res = i;
            count = 1;
        }
    }

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

    if (count <= n / 2) {
        res = -1;
    }
    return res;
}

int main() {
    int arr[] = {6, 8, 4, 8, 8};
    int n = sizeof(arr) / sizeof(arr[0]);

    cout << majority(arr, n) << endl;

    return 0;
}
```

Array Given:

$\text{arr[]} = \{6, 8, 4, 8, 8\}$
 $n = 5$

We need to find the element (if any) that appears **more than $5 / 2 = 2$** times.

🔗 Moore's Voting Algorithm Dry Run

We'll go step-by-step through the first for loop which finds a *candidate*.

i	arr[i]	arr[res]	count	Explanation
0	6	6	1	Initial candidate at index 0
1	8	6	0	$8 \neq 6 \rightarrow \text{count--}$
		8	1	$\text{count} = 0 \rightarrow \text{new candidate at index 1}$
2	4	8	0	$4 \neq 8 \rightarrow \text{count--}$
		4	1	$\text{count} = 0 \rightarrow \text{new candidate at index 2}$
3	8	4	0	$8 \neq 4 \rightarrow \text{count--}$
		8	1	$\text{count} = 0 \rightarrow \text{new candidate at index 3}$
4	8	8	2	$8 == 8 \rightarrow \text{count++}$

Candidate Index: $\text{res} = 3, \text{arr}[3] = 8$

❖ Second loop: Confirm the candidate

We check how many times 8 appears in the array.

```
count = 0;
for (int i = 0; i < n; i++) {
    if (arr[i] == 8) count++;
}
```

8 appears **3 times** (at indices 1, 3, and 4).

Since $3 > 2$, it **is** the majority element.

❖ Final Output

That's the index of the majority element 8.

Max Subarray sum in C++

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

int maxsub(int arr[], int n) {
    int res = arr[0];
    int maxEnding = arr[0];
    for (int i = 1; i < n; i++) {
        maxEnding = max(maxEnding + arr[i], arr[i]);
        res = max(res, maxEnding);
    }
    return res;
}

int main() {
    int arr[] = {-3, 8, -2, 4, -5, 6};
    int n = sizeof(arr) / sizeof(arr[0]);
    cout << maxsub(arr, n) << endl;
    return 0;
}
```

Input:

$\text{arr}[] = \{-3, 8, -2, 4, -5, 6\}$
 $n = 6$

■ Variables:

- res : Stores the **maximum subarray sum found so far**
- maxEnding : Stores the **maximum subarray sum ending at the current index**

☛ Dry Run Table:

i	$\text{arr}[i]$	$\text{maxEnding} = \max(\text{maxEnding} + \text{arr}[i], \text{arr}[i])$	$\text{res} = \max(\text{res}, \text{maxEnding})$
0	-3	$\text{maxEnding} = -3$	$\text{res} = -3$
1	8	$\max(-3 + 8, 8) = 8$	$\text{res} = 8$
2	-2	$\max(8 - 2, -2) = 6$	$\text{res} = 8$
3	4	$\max(6 + 4, 4) = 10$	$\text{res} = 10$
4	-5	$\max(10 - 5, -5) = 5$	$\text{res} = 10$
5	6	$\max(5 + 6, 6) = 11$	$\text{res} = 11$

❖ Final Output:

11

Tapping Rain Water in C++

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

int getWater(int arr[], int n) {
    int res = 0;
    for (int i = 0; i < n; i++) {
        int lmax = arr[i];
        for (int j = 0; j < i; j++) {
            lmax = max(arr[j], lmax);
        }
        int rmax = arr[i];
        for (int j = i + 1; j < n; j++) {
            rmax = max(arr[j], rmax);
        }

        res += min(lmax, rmax) - arr[i];
    }
    return res;
}

int main() {
    int arr[] = {3, 0, 1, 2, 5};
    int n = sizeof(arr) / sizeof(arr[0]);
    cout << getWater(arr, n) << endl;
    return 0;
}
```

Output:
6

Problem Explanation: Trapping Rain Water

At each index i , the amount of water it can hold is:

$$\text{water_at_}_i = \min(\text{lmax}, \text{rmax}) - \text{arr}[i]$$

Where:

- lmax : Max height to the left of i (including i)
- rmax : Max height to the right of i (including i)
- If $\min(\text{lmax}, \text{rmax}) - \text{arr}[i] > 0$, it adds to total water trapped.

Dry Run Table

Array: {3, 0, 1, 2, 5}

i	arr[i]	lmax (max left)	rmax (max right)	min(lmax, rmax)	Water at i = min(lmax, rmax) - arr[i]	res
0	3	3	5	3	0	0
1	0	3	5	3	3	3
2	1	3	5	3	2	5
3	2	3	5	3	1	6
4	5	5	5	5	0	6

Final Output:

6

Remove Duplicates in C++

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

int removeDup(int arr[], int n) {
    int res = 1;
    for (int i = 1; i < n; i++) {
        if (arr[i] != arr[res - 1]) {
            arr[res] = arr[i];
            res++;
        }
    }
    return res;
}

int main() {
    int arr[] = {2, 2, 3, 4, 5, 6};
    int n = sizeof(arr) / sizeof(arr[0]);
    int p = removeDup(arr, n);

    cout << "After Removal" << endl;

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

    return 0;
}
```

Dry Run Table

i	arr[i]	arr[res - 1]	Condition Met (!=)	Action	arr[res]	res
1	2	2	✗ No	Skip	-	1
2	3	2	✓ Yes	arr[1] = 3	3	2
3	4	3	✓ Yes	arr[2] = 4	4	3
4	5	4	✓ Yes	arr[3] = 5	5	4
5	6	5	✓ Yes	arr[4] = 6	6	5

✓ Final Values:

- $\text{res} = 5 \rightarrow$ means 5 unique elements.
- Modified array (first res elements):

$\text{arr[]} = \{2, 3, 4, 5, 6\}$

After Removal
2 3 4 5 6

Rotate Array in C++

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

void rotate(int arr[], int d, int n) {
    int temp[d];
    for (int i = 0; i < d; i++) {
        temp[i] = arr[i];
    }

    for (int i = d; i < n; i++) {
        arr[i - d] = arr[i];
    }

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

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

int main() {
    int arr[] = {1, 3, 6, 2, 5, 4, 3, 2, 4};
    int n = sizeof(arr) / sizeof(arr[0]);
    rotate(arr, 5, n);
    return 0;
}
```

Input:

$\text{arr}[] = \{1, 3, 6, 2, 5, 4, 3, 2, 4\}$
 $d = 5$
 $n = 9$

Step-by-step Breakdown:

1. Store first d elements in temp

$\text{temp} = \{1, 3, 6, 2, 5\}$

i	temp[i]
0	1
1	3
2	6
3	2
4	5

2. Shift remaining n - d elements to the left

$\text{arr}[0] = \text{arr}[5] \rightarrow 4$
 $\text{arr}[1] = \text{arr}[6] \rightarrow 3$
 $\text{arr}[2] = \text{arr}[7] \rightarrow 2$
 $\text{arr}[3] = \text{arr}[8] \rightarrow 4$

i	arr[i] (after shift)
0	4
1	3
2	2
3	4

3. Copy temp back to the end

$\text{arr}[4] = \text{temp}[0] = 1$
 $\text{arr}[5] = \text{temp}[1] = 3$
 $\text{arr}[6] = \text{temp}[2] = 6$
 $\text{arr}[7] = \text{temp}[3] = 2$
 $\text{arr}[8] = \text{temp}[4] = 5$

i	arr[i] (final state)
4	1

i	arr[i] (final state)
5	3
6	6
7	2
8	5

 **Final Output:**

4 3 2 4 1 3 6 2 5

4 3 2 4 1 3 6 2 5

Add Strings in C++

```
#include <iostream>
#include <string>
using namespace std;
```

```
string addStrings(string num1,
string num2) {
    string res = "";

    int i = num1.length() - 1;
    int j = num2.length() - 1;
    int carry = 0;

    while (i >= 0 || j >= 0 || carry != 0) {
        int ival = i >= 0 ? num1[i] - '0' : 0;
        int jval = j >= 0 ? num2[j] - '0' : 0;

        int sum = ival + jval + carry;
        res = to_string(sum % 10) + res;
        carry = sum / 10;

        i--;
        j--;
    }

    return res;
}
```

```
int main() {
    string n1 = "123";
    string n2 = "23";
    string res = addStrings(n1, n2);
    cout << res << endl; // Output should be 146

    return 0;
}
```

Input:

```
n1 = "123"
n2 = "23"
```

↘ Dry Run Table:

Step	i	j	num1[i]	num2[j]	ival	jval	carry (before)	sum = ival + jval + carry	res	carry (after)
1	2	1	'3'	'3'	3	3	0	6	"6"	0
2	1	0	'2'	'2'	2	2	0	4	"46"	0
3	0	-	'1'	-	1	0	0	1	"146"	0

✓ Final Output:

"146"

Island Perimeter in C++

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

int perimeter(vector<vector<int>>& grid) {
    int p = 0;
    int rows = grid.size();
    int cols = grid[0].size();

    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            if (grid[i][j] == 1) {
                p += 4;

                if (i > 0 && grid[i - 1][j] == 1) {
                    p -= 2;
                }
                if (j > 0 && grid[i][j - 1] == 1) {
                    p -= 2;
                }
            }
        }
    }
    return p;
}
```

```
int main() {
    vector<vector<int>> grid = {
        {1, 0, 0},
        {1, 1, 1},
        {0, 1, 0},
        {0, 1, 0}
    };

    int p = perimeter(grid);
    cout << p << endl;

    return 0;
}
```

Input Grid:

```
grid = {
    {1, 0, 0},
    {1, 1, 1},
    {0, 1, 0},
    {0, 1, 0}
};
```

Visualized:

```
1 0 0
1 1 1
0 1 0
0 1 0
```

🔗 Dry Run Strategy:

- Each land cell contributes +4 to perimeter.
- Each shared edge with another land cell subtracts 2.

🔍 Dry Run Table:

Cell (i,j)	grid[i][j]	+4	Top Neighbor = 1	Left Neighbor = 1	Net Contribution
(0,0)	1	4	✗	✗	4
(1,0)	1	4	✓ (0,0)	✗	2 (4-2)
(1,1)	1	4	✗	✓ (1,0)	2 (4-2)
(1,2)	1	4	✗	✓ (1,1)	2 (4-2)
(2,1)	1	4	✓ (1,1)	✗	2 (4-2)
(3,1)	1	4	✓ (2,1)	✗	2 (4-2)

✓ Total Perimeter:

$$= 4 + 2 + 2 + 2 + 2 + 2 = 14$$

✓ Output:

14

Max Avg. Subarray in C++

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

double solution(vector<int>& nums, int k) {
    int sum = 0;
    for (int i = 0; i < k; i++) {
        sum += nums[i];
    }

    int max_sum = sum;

    for (int i = k; i < nums.size(); i++) {
        sum += nums[i];
        sum -= nums[i - k];
        max_sum = max(max_sum, sum);
    }

    return static_cast<double>(max_sum) / k;
}

int main() {
    vector<int> nums = {-10, 5, -6, 8, -7, 2, -4, 8, -6, 7};
    int k = 3;
    cout << solution(nums, k) << endl;

    return 0;
}
```

Input:

nums = {-10, 5, -6, 8, -7, 2, -4, 8, -6, 7}
k = 3

Q Dry Run Table:

We'll track the sum of every window of size 3:

Window (Indexes)	Elements	Window Sum	max_sum
0–2	-10, 5, -6	-11	-11
1–3	5, -6, 8	7	7
2–4	-6, 8, -7	-5	7
3–5	8, -7, 2	3	7
4–6	-7, 2, -4	-9	7
5–7	2, -4, 8	6	7
6–8	-4, 8, -6	-2	7
7–9	8, -6, 7	9	9

✓ Final Output:

9 / 3 = 3.0

✓ Output: 3

Max Chunks to make array sorted in C++

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

int maxChunksToSorted(vector<int>& arr) {
    int max_val = 0;
    int count = 0;

    for (int i = 0; i < arr.size(); i++) {
        max_val = max(max_val, arr[i]);

        if (i == max_val) {
            count++;
        }
    }

    return count;
}

int main() {
    vector<int> arr = {4, 3, 2, 1, 0};
    int res = maxChunksToSorted(arr);
    cout << res << endl;

    return 0;
}
```

Input:

```
vector<int> arr = {4, 3, 2, 1, 0};
```

Q Dry Run Table:

Let's walk through the loop step-by-step and record values:

i	arr[i]	max_val (max so far)	i == max_val?	count
0	4	4	✗	0
1	3	4	✗	0
2	2	4	✗	0
3	1	4	✗	0
4	0	4	✓	1

❖ Output:

1

Max product of three in C++

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

int maxProduct(vector<int>& nums) {
    int min1 = INT_MAX, min2 = INT_MAX;
    int max1 = INT_MIN, max2 = INT_MIN,
    max3 = INT_MIN;

    for (int val : nums) {
        if (val > max1) {
            max3 = max2;
            max2 = max1;
            max1 = val;
        } else if (val > max2) {
            max3 = max2;
            max2 = val;
        } else if (val > max3) {
            max3 = val;
        }

        if (val < min1) {
            min2 = min1;
            min1 = val;
        } else if (val < min2) {
            min2 = val;
        }
    }

    return max(min1 * min2 * max1, max1 *
max2 * max3);
}

int main() {
    vector<int> nums = {2, 4, 6, 7};
    int result = maxProduct(nums);
    cout << result << endl;
    return 0;
}
```

Input:

nums = {2, 4, 6, 7}

❖ Variables Tracked:

Iteration n	val	max1	max2	max3	min1	min2
1	2	2	INT_MIN	INT_MIN	2	INT_MAX
2	4	4	2	INT_MIN	2	4
3	6	6	4	2	2	4
4	7	7	6	4	2	4

❖ Computed Products:

- $\min1 * \min2 * \max1 = 2 * 4 * 7 = 56$
- $\max1 * \max2 * \max3 = 7 * 6 * 4 = 168$

🧠 Output:

return max(56, 168); // → 168

No of subarrays with odd sum in C++

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

int nos(int arr[], int n) {
    long long ans = 0;
    int even = 0;
    int odd = 0;
    int sum = 0;

    for (int i = 0; i < n; i++) {
        sum += arr[i];
        if (sum % 2 == 0) {
            ans += odd;
            even++;
        } else {
            ans += 1 + even;
            odd++;
        }
    }
    return ans % 1000000007;
}

int main() {
    int arr[] = {1, 2, 3, 4, 5, 6, 7};
    int n = sizeof(arr) / sizeof(arr[0]);

    cout << nos(arr, n) << endl;

    return 0;
}
```

Input:

$\text{arr} = \{1, 2, 3, 4, 5, 6, 7\}$

Q Key Variables Tracked:

- $\text{sum} \rightarrow$ cumulative sum from start to current index
- $\text{even} \rightarrow$ count of prefix sums that are even so far
- $\text{odd} \rightarrow$ count of prefix sums that are odd so far
- $\text{ans} \rightarrow$ count of subarrays with odd sum

D Dry Run Table:

i	arr[i]	sum	sum%2	Action	ans	even	odd
0	1	1	1 (odd)	Add 1 + even (0) \rightarrow $\text{ans} += 1$	1	0	1
1	2	3	1 (odd)	Add 1 + even (0) \rightarrow $\text{ans} += 1$	2	0	2
2	3	6	0 (even)	Add odd (2) $\rightarrow \text{ans} += 2$	4	1	2
3	4	10	0 (even)	Add odd (2) $\rightarrow \text{ans} += 2$	6	2	2
4	5	15	1 (odd)	Add 1 + even (2) \rightarrow $\text{ans} += 3$	9	2	3
5	6	21	1 (odd)	Add 1 + even (2) \rightarrow $\text{ans} += 3$	12	2	4
6	7	28	0 (even)	Add odd (4) $\rightarrow \text{ans} += 4$	16	3	4

❖ Final Output:

16

Reverse Vowel of String in C++

```
#include <iostream>
#include <string>
#include <algorithm>
using namespace std;
bool isVowel(char ch) {
    return (ch == 'A' || ch == 'E' || ch == 'I' || ch ==
'O' || ch == 'U' ||
    ch == 'a' || ch == 'e' || ch == 'i' || ch == 'o'
    || ch == 'u');
}

string reverseVowel(string s) {
    int left = 0;
    int right = s.length() - 1;

    while (left < right) {
        while (left < right && !isVowel(s[left])) {
            left++;
        }

        while (left < right && !isVowel(s[right])) {
            right--;
        }

        if (left < right) {
            swap(s[left], s[right]);
            left++;
            right--;
        }
    }

    return s;
}

int main() {
    string s = "hello";
    string result = reverseVowel(s);
    cout << result << endl; // Output should be "holle"
    return 0;
}
```

Input:

string s = "hello";

Vowels: e, o

❖ Dry Run Table:

Step	left	right	s[left]	s[right]	Action	String After Change
1	0	4	h	o	h is not a vowel → left++	"hello"
2	1	4	e	o	Both are vowels → swap e and o	"holle"
3	2	3	l	l	No further vowel swap needed	"holle"

❖ Final Output:

holle

holle

Two Sum in C++

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

vector<vector<int>> twoSum(vector<int> nums, int target) {
    vector<vector<int>> res;
    int n = nums.size();
    sort(nums.begin(), nums.end()); // Sorting the array

    int left = 0, right = n - 1;
    while (left < right) {
        if (left > 0 && nums[left] == nums[left - 1]) { // Skip duplicates for left pointer
            left++;
            continue;
        }

        int sum = nums[left] + nums[right];
        if (sum == target) {
            res.push_back({nums[left], nums[right]});
            left++;
            right--;
        }
        // Skip duplicates for both left and right pointers
        while (left < right && nums[left] == nums[left - 1]) left++;
        while (left < right && nums[right] == nums[right + 1]) right--;

        } else if (sum > target) {
            right--;
        } else {
            left++;
        }
    }

    return res;
}

int main() {
    vector<int> nums = {2, 2, 4, 3, 1, 6, 6, 7, 5, 9, 1, 8, 9};
    int target = 10;

    vector<vector<int>> res = twoSum(nums, target);

    // Sorting each pair and then sorting all pairs lexicographically
    sort(res.begin(), res.end(), [] (const vector<int>& a, const vector<int>& b) {
        return a[0] == b[0] ? a[1] < b[1] : a[0] < b[0];
    });
}
```

Input:

nums = {2, 2, 4, 3, 1, 6, 6, 7, 5, 9, 1, 8, 9}
target = 10

After sorting:

nums = {1, 1, 2, 2, 3, 4, 5, 6, 6, 7, 8, 9, 9}

Q Step-by-step Table Dry Run:

Step	left	right	nums[left]	nums[right]	sum	Action	Result
1	0	12	1	9	10	Found a pair, store it	{1, 9}
	1	11	1	9	10	Skip duplicate left	
	2	11	2	9	11	Sum > target, move right--	
2	2	10	2	8	10	Found a pair, store it	{1, 9}, {2, 8}
	3	9	2	7	9	Skip duplicate left, move left++	
3	4	9	3	7	10	Found a pair, store it	{1,9},{2,8}, {3,7}
4	5	8	4	6	10	Found a pair, store it	{1,9},{2,8}, {3,7},{4,6}
5	6	7	5	6	11	Sum > target, move right--	
6	6	6	5	5	10	Stop (left >= right)	

❖ Final Result:

{ {1, 9}, {2, 8}, {3, 7}, {4, 6} }

```
// Printing the result
for (auto& pair : res) {
    for (int val : pair) {
        cout << val << " ";
    }
    cout << endl;
}

return 0;
}
```

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

All Subarray in C++

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

int main() {
    int arr[] = {1, 2, 3, 4};
    int n = sizeof(arr) / sizeof(arr[0]);

    for (int sp = 0; sp < n; sp++) {
        for (int ep = sp; ep < n; ep++) {
            for (int i = sp; i <= ep; i++) {
                cout << arr[i] << " ";
            }
            cout << endl;
        }
    }

    return 0;
}
```

Input:

```
arr[] = {1, 2, 3, 4};
```

⌚ Loop Structure:

- sp: Start point of subarray
- ep: End point of subarray
- i: Index for printing elements from sp to ep

💻 Dry Run Table:

sp	ep	Subarray Printed
0	0	1
0	1	1 2
0	2	1 2 3
0	3	1 2 3 4
1	1	2
1	2	2 3
1	3	2 3 4
2	2	3
2	3	3 4
3	3	4

⬆️ Output:

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

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

Print Boundary in C++

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

void printBoundary(vector<vector<int>>& mat) {
    int n = mat.size();
    int m = mat[0].size();

    // Print top row
    for (int j = 0; j < m; j++) {
        cout << mat[0][j] << " ";
    }

    // Print right column (excluding the top and bottom
    // elements already printed)
    for (int i = 1; i < n; i++) {
        cout << mat[i][m - 1] << " ";
    }

    // Print bottom row (excluding the bottom-right
    // corner already printed)
    if (n > 1) {
        for (int j = m - 2; j >= 0; j--) {
            cout << mat[n - 1][j] << " ";
        }
    }

    // Print left column (excluding the top-left and
    // bottom-left corners already printed)
    if (m > 1) {
        for (int i = n - 2; i > 0; i--) {
            cout << mat[i][0] << " ";
        }
    }
}

int main() {
    vector<vector<int>> mat = {
        {1, 2, 3, 4, 5},
        {6, 7, 8, 9, 10},
        {11, 12, 13, 14, 15},
        {16, 17, 18, 19, 20},
        {21, 22, 23, 24, 25}
    };

    printBoundary(mat);
    cout << endl;

    return 0;
}
```

Input Matrix (5x5):

```
[
[ 1, 2, 3, 4, 5 ],
[ 6, 7, 8, 9, 10 ],
[ 11, 12, 13, 14, 15 ],
[ 16, 17, 18, 19, 20 ],
[ 21, 22, 23, 24, 25 ]
]
```

⬇ Step-by-step Dry Run Table:

Step	Indices	Printed Values
Top row	mat[0][0 to 4]	1 2 3 4 5
Right column	mat[1 to 4][4]	10 15 20 25
Bottom row	mat[4][3 to 0]	24 23 22 21
Left column	mat[3 to 1][0]	16 11 6

Dry Run Table

Phase	Loop Variable(s)	Value Printed
Top Row	j = 0 to 4	1 2 3 4 5
Right Col	i = 1 to 4	10 15 20 25
Bottom Row	j = 3 to 0 (reverse)	24 23 22 21
Left Col	i = 3 to 1 (reverse)	16 11 6

◇ Final Output:

1 2 3 4 5 10 15 20 25 24 23 22 21 16 11 6

1 2 3 4 5 10 15 20 25 24 23 22 21 16 11 6

First Missing Positive in C++

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

int firstMissingPositive(vector<int>& nums) {
    int n = nums.size();

    int i = 0;
    while (i < n) {
        if (nums[i] == i + 1) {
            i++;
            continue;
        }

        if (nums[i] <= 0 || nums[i] > n) {
            i++;
            continue;
        }

        int idx1 = i;
        int idx2 = nums[i] - 1;

        if (nums[idx1] == nums[idx2]) {
            i++;
            continue;
        }

        int temp = nums[idx1];
        nums[idx1] = nums[idx2];
        nums[idx2] = temp;
    }

    for (int j = 0; j < n; j++) {
        if (nums[j] != j + 1) {
            return j + 1;
        }
    }

    return n + 1;
}

int main() {
    vector<int> nums = {3, 4, -1, 1};
    int result = firstMissingPositive(nums);
    cout << "First missing positive: " << result << endl;
    return 0;
}
```

Input:

vector<int> nums = {3, 4, -1, 1};

Goal:

Find the **smallest positive integer** that is **missing** from the array.

Algorithm Insight:

You're trying to **place each positive integer x ($1 \leq x \leq n$)** at index $x - 1$ using cyclic swaps.

Dry Run Table:

While loop swaps

Step	i	nums[i]	Action	nums after
1	0	3	swap nums[0] with nums[2] (index 2 = 3 - 1)	{-1, 4, 3, 1}
2	0	-1	invalid (≤ 0), move to i = 1	{-1, 4, 3, 1}
3	1	4	swap nums[1] with nums[3] (index 3 = 4 - 1)	{-1, 1, 3, 4}
4	1	1	swap nums[1] with nums[0] (index 0 = 1 - 1)	{1, -1, 3, 4}
5	1	-1	invalid, move to i = 2	{1, -1, 3, 4}
6	2	3	already at correct index (2 = 3 - 1)	no change
7	3	4	already at correct index (3 = 4 - 1)	no change

Final nums array after placements:

{1, -1, 3, 4}

Final Check:

Go through the array to find first j where $\text{nums}[j] \neq j + 1$:

j	nums[j]	j + 1	Match?
0	1	1	✓
1	-1	2	✗ → return 2

 **Output:**

First missing positive: 2

First missing positive: 2

Range Sum in C++

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

vector<int> prefixSum;

void NumArray(vector<int>& nums) {
    prefixSum.resize(nums.size());
    prefixSum[0] = nums[0];
    for (int i = 1; i < nums.size(); i++) {
        prefixSum[i] = prefixSum[i - 1] + nums[i];
    }
}

int sumRange(int i, int j) {
    if (i == 0) {
        return prefixSum[j];
    }
    return prefixSum[j] - prefixSum[i - 1];
}

int main() {
    vector<int> arr = {1, 2, 3, 4};
    NumArray(arr);
    int res = sumRange(1, 2);
    cout << res << endl; // Output should be 5

    return 0;
}
```

Prefix Sum Table Construction in NumArray(arr)

Let's build prefixSum[] based on the input arr = {1, 2, 3, 4}.

Index i	nums[i]	prefixSum[i] = prefixSum[i - 1] + nums[i]	prefixSum array
0	1	1	[1]
1	2	1 + 2 = 3	[1, 3]
2	3	3 + 3 = 6	[1, 3, 6]
3	4	6 + 4 = 10	[1, 3, 6, 10]

Final prefixSum = [1, 3, 6, 10]

sumRange(1, 2) Execution

We want to find sum from index 1 to 2 in original array (2 + 3 = 5).

Since i != 0, it uses:

$$\text{prefixSum}[2] - \text{prefixSum}[0] = 6 - 1 = 5$$

Expression	Value
prefixSum[2]	6
prefixSum[0]	1
Result	5

↙ Output printed: 5

Rotate Image in C++

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

void rotate(vector<vector<int>>& matrix) {
    int n = matrix.size();
    int m = matrix[0].size();

    // Transpose the matrix
    for (int i = 0; i < n; i++) {
        for (int j = i; j < m; j++) {
            swap(matrix[i][j], matrix[j][i]);
        }
    }

    // Reverse each row
    for (int i = 0; i < n; i++) {
        int sp = 0;
        int ep = m - 1;

        while (sp < ep) {
            swap(matrix[i][sp], matrix[i][ep]);
            sp++;
            ep--;
        }
    }
}

void print2DArray(const vector<vector<int>>& array)
{
    for (size_t i = 0; i < array.size(); i++) {
        for (size_t j = 0; j < array[i].size(); j++) {
            cout << array[i][j] << " ";
        }
        cout << endl;
    }
}

int main() {
    vector<vector<int>> matrix = {
        {1, 2, 3},
        {4, 5, 6},
        {7, 8, 9}
    };
    cout << "Original matrix:" << endl;
    print2DArray(matrix);
    rotate(matrix);
    cout << "Rotated matrix:" << endl;
    print2DArray(matrix);
    return 0;
}
```

Original matrix:

1 2 3
4 5 6
7 8 9

Rotated matrix:

7 4 1
8 5 2
9 6 3

Input Matrix:

Original matrix:

1 2 3
4 5 6
7 8 9

⌚ Step 1: Transpose the matrix

Transposing means swapping $\text{matrix}[i][j]$ with $\text{matrix}[j][i]$ for $j > i$.

i	j	$\text{matrix}[i][j]$	$\text{matrix}[j][i]$	Action
0	1	2	4	Swap $\rightarrow 2 \leftrightarrow 4$
0	2	3	7	Swap $\rightarrow 3 \leftrightarrow 7$
1	2	6	8	Swap $\rightarrow 6 \leftrightarrow 8$

⌚ After transpose:

1 4 7
2 5 8
3 6 9

⌚ Step 2: Reverse each row

Reverse each row of the transposed matrix:

Row Before	Row After
1 4 7	7 4 1
2 5 8	8 5 2
3 6 9	9 6 3

❖ Final Output:

Rotated matrix:

7 4 1
8 5 2
9 6 3

Running Sum in C++

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

vector<int> runningSum(vector<int>& nums) {
    int n = nums.size();
    vector<int> pre(n);
    pre[0] = nums[0];
    for (int i = 1; i < n; i++) {
        pre[i] = pre[i - 1] + nums[i];
    }
    return pre;
}

int main() {
    vector<int> arr = {1, 2, 3, 4};
    vector<int> res = runningSum(arr);

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

    return 0;
}
```

Input:

```
vector<int> arr = {1, 2, 3, 4};
```

Dry Run Table:

i	nums[i]	pre[i - 1]	pre[i] = pre[i - 1] + nums[i]	pre vector after iteration
0	1	-	pre[0] = 1	[1, __, __, __]
1	2	1	pre[1] = 1 + 2 = 3	[1, 3, __, __]
2	3	3	pre[2] = 3 + 3 = 6	[1, 3, 6, __]
3	4	6	pre[3] = 6 + 4 = 10	[1, 3, 6, 10]

Final Output (printed one per line):

```
1
3
6
10
```

```
1
3
6
10
```

All palindromic substrings in C++

```
#include <iostream>
#include <string>
using namespace std;

bool isPalindrome(string s) {
    int i = 0;
    int j = s.length() - 1;
    while (i <= j) {
        if (s[i] != s[j]) {
            return false;
        } else {
            i++;
            j--;
        }
    }
    return true;
}

void solution(string str) {
    // Write your code here
    for (int i = 0; i < str.length(); i++) {
        for (int j = i + 1; j <= str.length(); j++) {
            string ss = str.substr(i, j - i);
            if (isPalindrome(ss)) {
                cout << ss << "\n";
            }
        }
    }
}

int main() {
    string str = "abcc";
    solution(str);
    return 0;
}
```

Dry Run for Input: "abcc"

Let's list all substrings and mark which are palindromes:

Substring	From i	To j	Palindrome?
"a"	0	1	✓
"ab"	0	2	✗
"abc"	0	3	✗
"abcc"	0	4	✗
"b"	1	2	✓
"bc"	1	3	✗
"bcc"	1	4	✗
"c"	2	3	✓
"cc"	2	4	✓
"c"	3	4	✓

❖ Final Output (printed substrings):

a
b
c
cc
c

a
b
c
cc
c

Difference of every two consecutive character in C++						
Input String: "pepCODinG"						
Index (i)	prev	curr	ASCII(prev)	ASCII(curr)	Difference curr - prev	Intermediate Result
0	-	p	-	112	-	p
1	p	e	112	101	-11	p-11e
2	e	p	101	112	11	p-11e11p
3	p	C	112	67	-45	p-11e11p-45C
4	C	O	67	79	12	p-11e11p-45C12O
5	O	D	79	68	-11	p-11e11p-45C12O-11D
6	D	i	68	105	37	p-11e11p-45C12O-11D37i
7	i	n	105	110	5	p-11e11p-45C12O-11D37i5n
8	n	G	110	71	-39	p-11e11p-45C12O-11D37i5n-39G

↙ Final Output:

p-11e11p-45C12O-11D37i5n-39G

p-11e11p-45C12O-11D37i5n-39G

Remove Primes in C++

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

bool isPrime(int val) {
    if (val <= 1) return false; // 0 and 1 are not prime numbers
    for (int i = 2; i * i <= val; i++) {
        if (val % i == 0) {
            return false;
        }
    }
    return true;
}

void solution(vector<int>& nums) {
    for (int i = nums.size() - 1; i >= 0; i--) {
        if (isPrime(nums[i])) {
            nums.erase(nums.begin() + i); // Remove prime number
        }
    }
}

int main() {
    vector<int> nums = {3, 12, 13, 15};

    solution(nums);

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

    return 0;
}
```

12 15

Dry Run for Input:

```
vector<int> nums = {3, 12, 13, 15};
```

Index	Value	isPrime?	Action
3	15	✗	Keep
2	13	✓	Remove
1	12	✗	Keep
0	3	✓	Remove

✓ Final nums = {12, 15}

Output:

12 15

String Compression in C++

```
#include <iostream>
#include <string>
using namespace std;

string compression1(string str) {
    if (str.empty()) return ""; // Handle edge case

    string s;
    s += str[0]; // Append the first character directly
    for (int i = 1; i < str.length(); i++) {
        char curr = str[i];
        char prev = str[i - 1];
        if (curr != prev) {
            s += curr; // Append only if current character is
            // different from previous
        }
    }
    return s;
}

string compression2(string str) {
    if (str.empty()) return ""; // Handle edge case

    string s;
    s += str[0]; // Append the first character directly
    int count = 1;
    for (int i = 1; i < str.length(); i++) {
        char curr = str[i];
        char prev = str[i - 1];
        if (curr == prev) {
            count++; // Increment count for consecutive
            // characters
        } else {
            if (count > 1) {
                s += to_string(count); // Append count if it's
                // greater than 1
                count = 1; // Reset count
            }
            s += curr; // Append current character
        }
    }
    if (count > 1) {
        s += to_string(count); // Append the final count if
        // needed
    }
    return s;
}

int main() {
    string str = "wwwwwwaaadexxxxxx";
    cout << compression1(str) << endl;
    cout << compression2(str) << endl;
    return 0;
}
```

Step-by-Step Dry Run:
compression2("wwwwwwaaadexxxxxx")

i	curr	prev	count	Output so far	Action
1	w	w	2	w	same, count+
2	w	w	3	w	same, count+
3	w	w	4	w	same, count+
4	a	w	1	w4a	append 4, then a
5	a	a	2	w4a	same, count+
6	a	a	3	w4a	same, count+
7	d	a	1	w4a3d	append 3, then d
8	e	d	1	w4a3de	different, append e
9	x	e	1	w4a3dex	append x
10	x	x	2	w4a3dex	same, count+
11	x	x	3	w4a3dex	same, count+
12	x	x	4	w4a3dex	same, count+
13	x	x	5	w4a3dex	same, count+
14	x	x	6	w4a3dex	same, count+
end				w4a3dex6	append 6

▣ Final Output

wadex
w4a3dex6

wadex
w4a3dex6

Toggle in C++						
Dry Run for char st[] = "kriSh"						
Index	Character	Condition	ASCII Before	ASCII After	New Char	
0	'k'	lowercase → UPPER	107	75	'K'	
1	'r'	lowercase → UPPER	114	82	'R'	
2	'i'	lowercase → UPPER	105	73	'T'	
3	'S'	uppercase → lower	83	115	's'	
4	'h'	lowercase → UPPER	104	72	'H'	
<p>↗ Modified string becomes: "KRIsH"</p> <p>↙ Output</p> <p>KRIsH</p>						

KRIsH

Brute Force in C++

```
#include <iostream>
#include <string>
using namespace std;

void searchPattern(const string& text, const string& pat) {
    int m = pat.length();
    int n = text.length();

    for (int i = 0; i <= n - m; ++i) {
        int j;
        for (j = 0; j < m; ++j) {
            if (text[i + j] != pat[j]) {
                break;
            }
        }
        if (j == m) {
            cout << "Pattern found at index " << i << endl;
        }
    }
}

int main() {
    string text = "ababaababbbbabaaa";
    string pat = "aa";

    cout << "Text: " << text << endl;
    cout << "Pattern: " << pat << endl;

    searchPattern(text, pat);

    return 0;
}
```

Input:

- **Text:** "ababaababbbbabaaa"
- **Pattern:** "aa"
- **Length of pattern (m):** 2
- **Length of text (n):** 17

Dry Run Table:

We'll loop from $i = 0$ to $i = n - m = 15$.
We check every substring of length 2 and compare with "aa".

i	text[i..i+1]	Matches Pattern?
0	ab	✗
1	ba	✗
2	ab	✗
3	ba	✗
4	aa	✓
5	ab	✗
6	bb	✗
7	bb	✗
8	bb	✗
9	bb	✗
10	ba	✗
11	aa	✓
12	aa	✓
13	aa	✓
14	aa	✓
15	—	(out of bounds)

✓ Output:

Pattern found at index 4
 Pattern found at index 11
 Pattern found at index 12
 Pattern found at index 13
 Pattern found at index 14

Text: ababaababbbbabaaa

Pattern: aa

Pattern found at index 4

Pattern found at index 14

Pattern found at index 15

PolyHash in C++

```
#include <iostream>
#include <string>
using namespace std;

long long poly_hash(const string& s) {
    long long hash = 0;
    long long p = 31;
    const long long mod = 1000000007;
    long long p_power = 1;

    for (int i = 0; i < s.length(); i++) {
        hash = (hash + (s[i] - 'a' + 1) * p_power) % mod;
        p_power = (p_power * p) % mod;
    }

    return hash;
}

int main() {
    string s = "abaasdadasfasasfaba";
    cout << "Hash value: " << poly_hash(s) << endl;
    return 0;
}
```

String Details

Length: 20

Characters: a b a a s d a s d a s f a s a s f a b a

We'll use:

- $p = 31$
- $mod = 1000000007$

Hash formula:

$hash = (hash + (s[i] - 'a' + 1) * p_power) \% mod;$
 $p_power = (p_power * p) \% mod;$

Dry Run Table

i	s[i]	Val (s[i]-'a'+1)	p_power	Contribution (mod 1e9+7)	Hash So Far
0	a	1	1	1	1
1	b	2	31	62	63
2	a	1	961	961	1024
3	a	1	29791	29791	30815
4	s	19	923521	17546899	17577714
5	d	4	28629151	114516604	132094318
6	a	1	887503681	887503681	196981992
7	s	19	512613868	842901208	103882398
8	d	4	891031477	3564125908 → 564125894	668008292
9	a	1	62135468	62135468	730143760
10	s	19	256572640	4874880160 → 874880132	605023885
11	f	6	953752268	5722513608 → 722513601	327537479
12	a	1	566320160	566320160	893857639
13	s	19	566924949	10771573931 → 771573888	665431520
14	a	1	574673514	574673514	240105027
15	s	19	815124426	15487364094 → 487364703	727469730
16	f	6	269857340	1619144040 → 619144033	346613756
17	a	1	366577506	366577506	713191262
18	b	2	363902559	727805118	441996373
19	a	1	281979458	281979458	649975831

Final Hash Value

Hash value: 649975831

Hash value: 649975831

RabinCarp in C++

```
#include <iostream>
#include <string>
using namespace std;

const int p = 31;
const int mod = 1e9 + 7;
long long poly_hash(const string& s) {
    long long hash = 0;
    long long p_power = 1;

    for (int i = 0; i < s.length(); i++) {
        hash = (hash + (s[i] - 'a' + 1) * p_power) % mod;
        p_power = (p_power * p) % mod;
    }

    return hash;
}

int powr(int a, int b) {
    // (a^b)%mod
    int res = 1;
    while (b > 0) {
        if (b & 1) res = (res * 1LL * a) % mod;
        a = (a * 1LL * a) % mod;
        b >>= 1;
    }
    return res;
}

int main() {
    string text = "ababbabbaba";
    string pattern = "aba";
    long long pat_hash = poly_hash(pattern);
    int n = text.length(), m = pattern.length();
    long long text_hash = poly_hash(text.substr(0, m));
    if (pat_hash == text_hash) {
        cout << 0 << endl;
    }
    for (int i = 1; i + m <= n; i++) {
        // remove last character
        text_hash = (text_hash - (text[i - 1] - 'a' + 1) +
mod) % mod;

        text_hash = (text_hash * 1LL * powr(p, mod - 2)) %
mod;

        text_hash = (text_hash + (text[i + m - 1] - 'a' + 1) *
1LL * powr(p, m - 1)) % mod;

        if (text_hash == pat_hash) {
            cout << i << endl;
        }
    }
    return 0;
}
```

Input:

- **Text** = "ababbabbaba"
- **Pattern** = "aba"
- **p** = 31, **mod** = 1e9 + 7

↓ Step 1: Compute pattern hash

Pattern: "a" (1), "b" (2), "a" (1)

Hash formula:

$$\begin{aligned} \text{hash} &= (1*p^0 + 2*p^1 + 1*p^2) \% \text{mod} \\ &= (1*1 + 2*31 + 1*961) = 1 + 62 + 961 = 1024 \end{aligned}$$

↓ Step 2: Slide over text & compare hash window

We'll use a table with:

Index i	Substring text[i..i+2]	Rolling Hash	Matches pat_hash = 1024?
0	a b a	1024	✓ Yes
1	b a b	2973	✗ No
2	a b b	2086	✗ No
3	b b a	2853	✗ No
4	b a b	2973	✗ No
5	a b b	2086	✗ No
6	b b a	2853	✗ No
7	b a b	2973	✗ No
8	a b a	1024	✓ Yes

✓ Matches found at indices:

0
8

Fast Power in C++

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

class FastPower {
public:
    static int fastpower(int a, int b) {
        int res = 1;
        while (b > 0) {
            if (b & 1) {
                res = res * a;
            }
            a = a * a;
            b = b >> 1;
        }
        return res;
    }

    static void main() {
        cout << fastpower(3, 5) << endl;
    }
};

int main() {
    FastPower::main();
    return 0;
}
```

Dry Run Table:

Step	b (binary)	b (decimal)	a	res	Operation	Explanation
0	101	5	3	1		Initial values
1	101	5	3	3	res = res * a	LSB is 1 → multiply res by a
2	10	2	9	3	a = a * a, b >>= 1	Square a → $3^2 = 9$, shift b → b = 2
3	10	2	9	3	(skip multiplication)	LSB is 0 → skip multiplying res
4	1	1	81	3	a = a * a, b >>= 1	$a = 9^2 = 81$, b = 1
5	1	1	81	243	res = res * a	LSB is 1 → res = $3 \times 81 = 243$
6	0	0			Done	Loop ends

❖ Final Output:

243

243

GCD in C++

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

class GCD {
public:
    static int gcd(int a, int b) {
        if (b == 0) {
            return a;
        } else {
            return gcd(b, a % b);
        }
    }

    static void main() {
        cout << gcd(30, 36) << endl;
    }
};

int main() {
    GCD::main();
    return 0;
}
```

Function: gcd(a, b)

This uses the rule:

$$gcd(a, b) = gcd(b, a \% b)$$

...until $b == 0$.

Dry Run Table for gcd(30, 36)

Call Depth	a	b	a % b	Next Call	Returned Value
1	30	36	30	gcd(36, 30)	
2	36	30	6	gcd(30, 6)	
3	30	6	0	gcd(6, 0)	6
← Return				← back to depth 2	6
← Return				← back to depth 1	6

✓ Final Output:

6

Prime Factor in C++

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

class PrimeFactors {
public:
    static void main() {
        int n = 26;
        int n2 = 2;

        while (n2 * n2 <= n) {
            while (n % n2 == 0) {
                n = n / n2;
                cout << n2 << " ";
            }
            n2++;
        }

        if (n != 1) {
            cout << n << " ";
        }
    }
};

int main() {
    PrimeFactors::main();
    return 0;
}
```

2 13

Print all **prime factors** of n = 26.

🧠 Logic:

- Start with n2 = 2.
- While $n2 * n2 \leq n$, divide n by n2 as long as it's divisible.
- Increment n2 and repeat.
- After the loop, if n != 1, print the remaining prime factor.

💻 Dry Run Table:

Step	n2	n	n % n2 == 0	Action	Output
1	2	26	Yes	$n = 26 / 2 = 13$	2
2	2	13	No	n2++	
3	3	13	No	n2++	
4	4	13	No	n2++	
5	5	13	No	n2++	
6	6	13	6*6 > 13 → stop		
7	-	13	-	$n \neq 1 \rightarrow \text{print } n$	13

🖨 Final Output:

2 13

Seive in C++

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

class SeiveofErastostenins {
public:
    static void main() {
        vector<bool> myseive = seive(20);
        for (int i = 0; i < myseive.size(); i++) {
            cout << i << " " << (myseive[i] ? "true" : "false")
        << endl;
    }
}

static vector<bool> seive(int n) {
    vector<bool> arr(n + 1, true);
    arr[0] = false;
    arr[1] = false;
    for (int i = 2; i <= sqrt(n); i++) {
        if (arr[i]) {
            for (int j = i * i; j <= n; j += i) {
                arr[j] = false;
            }
        }
    }
    return arr;
};

int main() {
    SeiveofErastostenins::main();
    return 0;
}
```

Sieve of Eratosthenes Dry Run for n = 20

1 Step 1: Initialize Boolean Vector

`vector<bool> arr(n + 1, true); // arr[0..20] all set to true
arr[0] = false;
arr[1] = false;`

2 Initial Table:

i	isPrime
0	false
1	false
2	true
3	true
4	true
5	true
6	true
7	true
8	true
9	true
10	true
11	true
12	true
13	true
14	true
15	true
16	true
17	true
18	true
19	true
20	true

2 Step 2: Outer loop — `for (int i = 2; i <= sqrt(n); i++)`

- $\sqrt{20}$ is $\sim 4.47 \rightarrow$ so i goes from 2 to 4

► i = 2:

`arr[2] == true \rightarrow mark all multiples of 2 from 4 onward as false`

Inner loop ($j = i*i; j <= n; j += i$) $\rightarrow j = 4, 6, 8, 10, 12, 14, 16, 18, 20$

X Marked False:

4, 6, 8, 10, 12, 14, 16, 18, 20

► **i = 3:**

arr[3] == true → mark all multiples of 3 from 9 onward as false

j = 9, 12, 15, 18

✗ **Marked False:**

9, 15

(12 and 18 already marked by i = 2)

► **i = 4:**

arr[4] == false → skip

❖ **Final Table After Sieve:**

i	isPrime
0	false
1	false
2	true
3	true
4	false
5	true
6	false
7	true
8	false
9	false
10	false
11	true
12	false
13	true
14	false
15	false
16	false
17	true
18	false
19	true
20	false

⌚ **Output Printed by the Code:**

0 false

	1 false 2 true 3 true 4 false 5 true 6 false 7 true 8 false 9 false 10 false 11 true 12 false 13 true 14 false 15 false 16 false 17 true 18 false 19 true 20 false
0 false 1 false 2 true 3 true 4 false 5 true 6 false 7 true 8 false 9 false 10 false 11 true 12 false 13 true 14 false 15 false 16 false 17 true 18 false 19 true 20 false	❖ Prime Numbers ≤ 20: 2, 3, 5, 7, 11, 13, 17, 19

Trailing Zeroes in C++

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

class TrailingZeroes {
public:
    static void main() {
        int res = 1000;
        int n = 7;
        for (int i = 5; i <= n; i = i * 5) {
            res = res + n / i;
        }
        cout << "zeroes: " << res << endl;
    }
};

int main() {
    TrailingZeroes::main();
    return 0;
}
```

zeroes: 1

Dry Run for $n = 7$

i	n / i	res (cumulative)
5	$7 / 5 = 1$	$0 + 1 = 1$
25	$7 / 25 = 0$	loop ends

↙ Final answer: **1 trailing zero in 7!**

▣ Output (after fixing $res = 0$):

zeroes: 1

Co-prime pairs in C++

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

class CoPrimePairs {
public:
    static void main() {
        int n = 10;

        for (int i = 0; i < n / 2; i++) {
            cout << 2 * i + 1 << " " << 2 * i + 2 << endl;
        }
    }

    int main() {
        CoPrimePairs::main();
        return 0;
    }
}
```

Dry Run Table for n = 10

i	2*i + 1	2*i + 2	Output
0	1	2	1 2
1	3	4	3 4
2	5	6	5 6
3	7	8	7 8
4	9	10	9 10

Output

```
1 2
3 4
5 6
7 8
9 10
```

```
1 2
3 4
5 6
7 8
9 10
```

GCD array in C++

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

// Function to compute GCD of two numbers using
// Euclidean algorithm
int gcd(int a, int b) {
    while (b != 0) {
        int temp = b;
        b = a % b;
        a = temp;
    }
    return a;
}

// Function to compute GCD of an array of integers
int gcdArray(vector<int>& arr) {
    int result = arr[0];
    for (int i = 1; i < arr.size(); i++) {
        result = gcd(result, arr[i]);
        if (result == 1) { // If result becomes 1, further
GCD will also be 1
            return 1;
        }
    }
    return result;
}

int main() {
    vector<int> arr = {12, 24, 36, 48};
    cout << "GCD of the array elements: " <<
gcdArray(arr) << endl;
    return 0;
}
```

GCD of the array elements: 12

Step-by-Step Dry Run (Tabular Form)

We'll use this table to track the intermediate GCD results:

Step	result (previous GCD)	arr[i]	gcd(result, arr[i])
1	12	24	gcd(12, 24) = 12
2	12	36	gcd(12, 36) = 12
3	12	48	gcd(12, 48) = 12

Since the GCD never drops to 1, we never hit the `if (result == 1)` shortcut.

Final Output:

GCD of the array elements: 12

NumberofSubArrays with GCDequaltoK in C++

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

class NumberofSubArrays with GCDequaltoK {
public:
    int subarrayGCD(vector<int>& nums, int k) {
        int count = 0;
        int n = nums.size();

        for (int sp = 0; sp < n; sp++) {
            int ans = 0;
            for (int ep = sp; ep < n; ep++) {
                ans = gcd(ans, nums[ep]);

                if (ans < k) {
                    break;
                }
                if (ans == k) {
                    count++;
                }
            }
        }
        return count;
    }

    int gcd(int a, int b) {
        if (a == 0) {
            return b;
        }
        return gcd(b % a, a);
    }
};

int main() {
    NumberofSubArrays with GCDequaltoK solution;

    // Hard-coded input
    vector<int> nums = {2, 4, 6, 8, 3, 9};
    int k = 3;

    int result = solution.subarrayGCD(nums, k);
    cout << "Number of subarrays with GCD equal to "
    << k << ":" << result << endl;

    return 0;
}
```

Input:

nums = {2, 4, 6, 8, 3, 9}
k = 3

We'll check **all subarrays** and see how many have GCD = 3.

Dry Run Table

sp	Subarray	ans (GCD)	Matches k?
0	[2]	2	✗
0	[2, 4]	2	✗
0	[2, 4, 6]	2	✗
0	[2, 4, 6, 8]	2	✗
0	[2, 4, 6, 8, 3]	1	✗ (GCD < k) – break
1	[4]	4	✗
1	[4, 6]	2	✗
1	[4, 6, 8]	2	✗
1	[4, 6, 8, 3]	1	✗ (GCD < k) – break
2	[6]	6	✗
2	[6, 8]	2	✗
2	[6, 8, 3]	1	✗ (GCD < k) – break
3	[8]	8	✗
3	[8, 3]	1	✗ (GCD < k) – break
4	[3]	3	✓
4	[3, 9]	3	✓
5	[9]	9	✗

Final Count

We found **2 subarrays** where the GCD is exactly 3:

- [3]
- [3, 9]

Explanation of Logic

You're using a **nested loop**:

- Outer loop: start point sp
- Inner loop: end point ep
- You maintain a running GCD of the subarray
- If GCD < k, you **break** early (smart optimization)
- If GCD == k, increment the counter

And your GCD function is correct, based on the Euclidean algorithm.

 **Output:**

Number of subarrays with GCD equal to 3: 2

Number of subarrays with GCD equal to 3: 2

Subsequence with GCD in C++

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

class SubsequencewithGCD {
public:
    static void main() {
        int arr[] = {1, 2, 3, 4};
        int n = sizeof(arr) / sizeof(arr[0]);

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

        if (ans == 1) {
            cout << "true" << endl;
        } else {
            cout << "false" << endl;
        }
    }

    static int gcd(int a, int b) {
        if (b == 0) {
            return a;
        } else {
            return gcd(b, a % b);
        }
    }

    int main() {
        SubsequencewithGCD::main();
        return 0;
    }
}
```

Dry Run on Given Input

arr[] = {1, 2, 3, 4}

Let's compute:

Step	i	arr[i]	Current GCD (ans)
1	0	1	gcd(0, 1) = 1
2	1	2	gcd(1, 2) = 1
3	2	3	gcd(1, 3) = 1
4	3	4	gcd(1, 4) = 1

↙ Final GCD = 1 → So the output will be:

true

↙ Output

true

true

Bubble sort in C++

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

void BubbleSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - 1 - i; j++) {
            if (arr[j] > arr[j + 1]) {
                // Swap arr[j] and arr[j+1]
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }

    int main() {
        int arr[] = {0, 1, 5, 7, 8, 9, 4};
        int n = sizeof(arr) / sizeof(arr[0]);

        BubbleSort(arr, n);

        cout << "Sorted array: ";
        for (int i = 0; i < n; i++) {
            cout << arr[i] << " ";
        }
        cout << endl;

        return 0;
    }
}
```

Dry Run Table:

Initial:

[0, 1, 5, 7, 8, 9, 4]

Pass 1 (i = 0):

Compare arr[j]	Swap?	Result
0 and 1	No	[0, 1, 5, 7, 8, 9, 4]
1 and 5	No	[0, 1, 5, 7, 8, 9, 4]
5 and 7	No	[0, 1, 5, 7, 8, 9, 4]
7 and 8	No	[0, 1, 5, 7, 8, 9, 4]
8 and 9	No	[0, 1, 5, 7, 8, 9, 4]
9 and 4	Yes	[0, 1, 5, 7, 8, 4, 9]

↙ Largest element 9 moved to the end.

Pass 2 (i = 1):

Compare arr[j]	Swap?	Result
0 and 1	No	[0, 1, 5, 7, 8, 4, 9]
1 and 5	No	[0, 1, 5, 7, 8, 4, 9]
5 and 7	No	[0, 1, 5, 7, 8, 4, 9]
7 and 8	No	[0, 1, 5, 7, 8, 4, 9]
8 and 4	Yes	[0, 1, 5, 7, 4, 8, 9]

↙ Second-largest 8 in place.

Pass 3 (i = 2):

Compare arr[j]	Swap?	Result
0 and 1	No	[0, 1, 5, 7, 4, 8, 9]
1 and 5	No	[0, 1, 5, 7, 4, 8, 9]
5 and 7	No	[0, 1, 5, 7, 4, 8, 9]
7 and 4	Yes	[0, 1, 5, 4, 7, 8, 9]

Pass 4 (i = 3):

Compare arr[j]	Swap?	Result
0 and 1	No	[0, 1, 5, 4, 7, 8, 9]
1 and 5	No	[0, 1, 5, 4, 7, 8, 9]
5 and 4	Yes	[0, 1, 4, 5, 7, 8, 9]

Pass 5 (i = 4):

Compare arr[j]	Swap?	Result
0 and 1	No	[0, 1, 4, 5, 7, 8, 9]

Compare arr[j]	Swap?	Result
1 and 4	No	[0, 1, 4, 5, 7, 8, 9]

Pass 6 (i = 5):

Compare arr[j]	Swap?	Result
0 and 1	No	[0, 1, 4, 5, 7, 8, 9]

▣ Final Sorted Array:

Sorted array: 0 1 4 5 7 8 9

Sorted array: 0 1 4 5 7 8 9

Count Inversions in C++				
Step-by-Step Merge and Inversion Tracking				
Step	Subarrays (Left - Right)	Comparison	Inversion Count	Merged Result
1	[2] and [3]	$2 \leq 3$	0	[2, 3]
2	[2, 3] and [8]	All in order	0	[2, 3, 8]
3	[6] and [1]	$6 > 1$	1	[1, 6]
4	[2, 3, 8] and [1, 6]	$2 > 1$ $2 < 6$ $3 < 6$	3 (2,3,8 > 1) 0 0	[1, 2, 3, 6, 8]
		$8 > 6$	1	
↙ Summary				
Merge Step	Inversions Found			
[2] and [3]	0			
[2, 3] and [8]	0			
[6] and [1]	1			
[2, 3, 8] and [1, 6]	$3 + 1 = 4$			
Total Inversions	5			

```
int main() {
    vector<long long> arr = {2, 3, 8, 6, 1};

    cout << "Given Array:" << endl;
    printArray(arr);

    long long inversionCountValue =
inversionCount(arr);

    cout << "Number of inversions: " <<
inversionCountValue << endl;

    return 0;
}
```

Given Array:

2 3 8 6 1

Number of inversions: 5

Count Sort in C++

```
#include <iostream>
#include <cstring>
using namespace std;

string countSort(string s) {
    char arr[s.length()];
    strcpy(arr, s.c_str());

    char maxch = 'a';
    for (int i = 0; i < strlen(arr); i++) {
        if (arr[i] > maxch) {
            maxch = arr[i];
        }
    }
    int max = maxch - 'a';
    int count[max + 1] = {0};

    for (int i = 0; i < strlen(arr); i++) {
        int val = arr[i] - 'a';
        count[val]++;
    }

    int k = 0;
    for (int i = 0; i <= max; i++) {
        int c = count[i];
        for (int j = 0; j < c; j++) {
            arr[k] = i + 'a';
            k++;
        }
    }

    string sortedString(arr);
    return sortedString;
}

int main() {
    string input = "countingsortexample";
    string sortedString = countSort(input);

    cout << "Original String: " << input << endl;
    cout << "Sorted String: " << sortedString << endl;

    return 0;
}
```

Step-by-Step Dry Run:

Step 1: Copy string to character array

strcpy(arr, s.c_str());

Now arr = "countingsortexample"

Step 2: Find max character (in terms of ASCII)

char maxch = 'x'; // max character = 'x'

int max = maxch - 'a'; // max = 23

Step 3: Count frequency of each character

Character	Count
a	1
c	1
e	2
g	1
i	1
l	1
m	1
n	2
o	2
p	1
r	1
s	1
t	2
u	1
x	1

Step 4: Reconstruct the sorted array

Characters are added in order of 'a' to 'x' based on count.

Sorted string becomes:

	"aceegilmnnooprsttux" ✓ Output: Original String: countingsortexample Sorted String: aceegilmnnooprsttux
--	---

Original String: countingsortexample
Sorted String: aceegilmnnooprsttux

Good Integers distinct in C++

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int GoodIntegers(vector<int>& arr) {
    sort(arr.begin(), arr.end()); // Sort the array

    int ans = 0;

    for (int i = 0; i < arr.size(); ++i) {
        if (arr[i] == i) { // Check if the value at index i
            matches i
            ++ans;
        }
    }

    return ans; // Return the count of good integers
}

int main() {
    vector<int> arr = {0, 1, 5, 7, 8, 9, 4};

    cout << GoodIntegers(arr) << endl;

    return 0;
}
```

Input:

vector<int> arr = {0, 1, 5, 7, 8, 9, 4};

Step 1: Sort the array

Sorted arr = {0, 1, 4, 5, 7, 8, 9}
↑ ↑ ↑ ↑ ↑ ↑ ↑
Index 0 1 2 3 4 5 6

Step 2: Compare each element with its index

Index i	arr[i]	arr[i] == i	Count (ans)
0	0	✓ Yes	1
1	1	✓ Yes	2
2	4	✗ No	2
3	5	✗ No	2
4	7	✗ No	2
5	8	✗ No	2
6	9	✗ No	2

Final Output:

cout << GoodIntegers(arr); // Output: 2

✓ Because arr[0] = 0 and arr[1] = 1 match their indices.

Good Integers duplicate in C++

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int GoodIntegers(int arr[], int n) {
    sort(arr, arr + n); // Sort the array

    int ans = 0;
    int lessCount = 0;

    if (arr[0] == 0) {
        ans++;
    }

    for (int i = 1; i < n; ++i) {
        if (arr[i] != arr[i - 1]) {
            lessCount = i;
        }

        if (arr[i] == lessCount) {
            ans++;
        }
    }

    return ans;
}

int main() {
    int arr[] = {0, 1, 5, 7, 8, 9, 4};
    int n = sizeof(arr) / sizeof(arr[0]);

    cout << GoodIntegers(arr, n) << endl;

    return 0;
}
```

Goal of the Function:

Count how many elements in the array are **equal to the number of elements less than it**.

❖ Step-by-step Dry Run

► Step 1: Sort the array

Initial array: {0, 1, 5, 7, 8, 9, 4}

Sorted array: {0, 1, 4, 5, 7, 8, 9}

$n = 7$

Variables: $ans = 0$, $lessCount = 0$

Index (i)	arr[i]	arr[i-1]	arr[i] != arr[i-1]	lessCount	arr[i] == lessCount	ans
0	0	-	-	0	✓ (0 == 0)	1
1	1	0	✓	1	✓ (1 == 1)	2
2	4	1	✓	2	✗ (4 != 2)	2
3	5	4	✓	3	✗ (5 != 3)	2
4	7	5	✓	4	✗ (7 != 4)	2
5	8	7	✓	5	✗ (8 != 5)	2
6	9	8	✓	6	✗ (9 != 6)	2

❖ Final Answer: 2

The two good integers are:

- 0: there are 0 elements less than it → good
- 1: there is 1 element less than it → good

Insertion Sort in C++

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

class InsertionSort {
public:
    // Function to perform insertion sort on array arr of
    // size n
    void insertionSort(int arr[], int n) {
        for (int i = 1; i < n; i++) {
            insert(arr, i);
        }
    }

private:
    // Helper function to insert arr[i] into the sorted
    // sub-array arr[0...i-1]
    void insert(int arr[], int i) {
        int key = arr[i]; // Element to be inserted
        int j = i - 1; // Start comparing with the
        previous element

        // Move elements of arr[0..i-1], that are greater
        // than key, to one position ahead of their current
        // position
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key; // Place key at its correct position
    }

int main() {
    InsertionSort solution;

    // Hardcoded input array
    int arr[] = {5, 2, 9, 1, 5, 6};
    int n = sizeof(arr) / sizeof(arr[0]);

    // Sorting the array using insertion sort
    solution.insertionSort(arr, n);

    // Printing the sorted array
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;

    return 0;
}
```

Let's dry run your **Insertion Sort** code step by step with the input:

```
int arr[] = {5, 2, 9, 1, 5, 6};
```

⌚ Insertion Sort Dry Run Table

i	Key	Initial Array State	Comparison Index (j)	Action Taken	Updated Array
1	2	[5, 2, 9, 1, 5, 6]	j = 0 (5 > 2)	Shift 5 to index 1	[5, 5, 9, 1, 5, 6]
			j = -1	Insert 2 at index 0	[2, 5, 9, 1, 5, 6]
2	9	[2, 5, 9, 1, 5, 6]	j = 1 (5 < 9)	No shifting, insert 9 at index 2	[2, 5, 9, 1, 5, 6]
3	1	[2, 5, 9, 1, 5, 6]	j = 2 (9 > 1)	Shift 9 to index 3	[2, 5, 9, 9, 5, 6]
			j = 1 (5 > 1)	Shift 5 to index 2	[2, 5, 5, 9, 5, 6]
			j = 0 (2 > 1)	Shift 2 to index 1	[2, 2, 5, 9, 5, 6]
			j = -1	Insert 1 at index 0	[1, 2, 5, 9, 5, 6]
4	5	[1, 2, 5, 9, 5, 6]	j = 3 (9 > 5)	Shift 9 to index 4	[1, 2, 5, 9, 9, 6]
			j = 2 (5 == 5)	No shifting (stable), insert 5 at index 3	[1, 2, 5, 5, 9, 6]
5	6	[1, 2, 5, 5, 9, 6]	j = 4 (9 > 6)	Shift 9 to index 5	[1, 2, 5, 5, 9, 9]
			j = 3 (5 < 6)	Insert 6 at index 4	[1, 2, 5, 5, 6, 9]

❖ Final Sorted Array:

	[1, 2, 5, 5, 6, 9]
1 2 5 5 6 9	

Merge 2 sorted subarrays in C++

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

// Function to merge two sorted subarrays within
array 'a'
vector<int> mergeTwoSortedSubArray(vector<int>&
a, int s, int m, int e) {
    vector<int> temp(e - s + 1);
    int p1 = s;
    int p2 = m + 1;
    int p3 = 0;

    // Merge elements from two subarrays into temp
    array
    while (p1 <= m && p2 <= e) {
        if (a[p1] < a[p2]) {
            temp[p3] = a[p1];
            p3++;
            p1++;
        } else {
            temp[p3] = a[p2];
            p3++;
            p2++;
        }
    }

    // Copy remaining elements of the first subarray, if
    any
    while (p1 <= m) {
        temp[p3] = a[p1];
        p3++;
        p1++;
    }

    // Copy remaining elements of the second subarray,
    if any
    while (p2 <= e) {
        temp[p3] = a[p2];
        p3++;
        p2++;
    }

    // Copy sorted elements from temp back to original
    array 'a'
    for (int i = 0; i < temp.size(); i++) {
        a[s + i] = temp[i];
    }

    return a;
}

int main() {
    // Hard-coded input
    vector<int> A = {1, 3, 5, 7, 2, 4, 6, 8};
    int s = 0;
    int m = 3; // Middle index of the first sorted
    subarray
    int e = 7; // End index of the second sorted subarray

    // Merging the two sorted subarrays
}
```

using the input:

A = {1, 3, 5, 7, 2, 4, 6, 8}
s = 0, m = 3, e = 7

This means:

- First sorted subarray = A[0..3] = {1, 3, 5, 7}
- Second sorted subarray = A[4..7] = {2, 4, 6, 8}

⌚ Dry Run Table:

Step	p1	p2	temp[] (after step)	Comment
1	0	4	{1}	1 < 2, so copy 1 from left
2	1	4	{1, 2}	2 < 3, so copy 2 from right
3	1	5	{1, 2, 3}	3 < 4, so copy 3 from left
4	2	5	{1, 2, 3, 4}	4 < 5, so copy 4 from right
5	2	6	{1, 2, 3, 4, 5}	5 < 6, so copy 5 from left
6	3	6	{1, 2, 3, 4, 5, 6}	6 < 7, so copy 6 from right
7	3	7	{1, 2, 3, 4, 5, 6, 7}	7 < 8, so copy 7 from left
8	4	7	{1, 2, 3, 4, 5, 6, 7, 8}	only 8 left, copy from right

Now the merged array looks like:

A = {1, 2, 3, 4, 5, 6, 7, 8}

❖ Final Output:

Merged array: 1 2 3 4 5 6 7 8

```
vector<int> result = mergeTwoSortedSubArray(A,  
s, m, e);  
  
// Print the result  
cout << "Merged array: ";  
for (int num : result) {  
    cout << num << " ";  
}  
cout << endl;  
  
return 0;  
}
```

```
Merged array: 1 2 3 4 5 6 7 8
```

Merge Sort in C++

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

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

        // Create temporary arrays
        vector<int> L(n1), R(n2);

        // Copy data to temporary arrays L[] and R[]
        for (int i = 0; i < n1; i++)
            L[i] = arr[l + i];
        for (int j = 0; j < n2; j++)
            R[j] = arr[m + 1 + j];

        // Merge the temporary arrays back into arr[l..r]
        int i = 0; // Initial index of first subarray
        int j = 0; // Initial index of second subarray
        int k = l; // Initial index of merged subarray

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

        // Copy the remaining elements of L[], if any
        while (i < n1) {
            arr[k] = L[i];
            i++;
            k++;
        }

        // Copy the remaining elements of R[], if any
        while (j < n2) {
            arr[k] = R[j];
            j++;
            k++;
        }
    }

    void mergeSort(vector<int>& arr, int l, int r) {
        if (l >= r)
            return; // Base case: array size is 0 or 1
        int m = l + (r - l) / 2;
        mergeSort(arr, l, m); // Sort first half
        mergeSort(arr, m + 1, r); // Sort second half
        merge(arr, l, m, r); // Merge sorted halves
    }
};
```

Let's walk through a **dry run** of your **Merge Sort implementation** with the input:

arr = {12, 11, 13, 5, 6, 7}

💡 Step-by-step Breakdown:

We'll visualize the recursive division and merging process.

✳️ Recursive Division (mergeSort)

Level	Call	Subarray
1	mergeSort(arr, 0, 5)	[12, 11, 13, 5, 6, 7]
2	mergeSort(arr, 0, 2)	[12, 11, 13]
3	mergeSort(arr, 0, 1)	[12, 11]
4	mergeSort(arr, 0, 0)	[12]
4	mergeSort(arr, 1, 1)	[11]
3	merge(arr, 0, 0, 1)	merge [12] and [11] ⇒ [11, 12]
3	mergeSort(arr, 2, 2)	[13]
2	merge(arr, 0, 1, 2)	merge [11, 12] and [13] ⇒ [11, 12, 13]
2	mergeSort(arr, 3, 5)	[5, 6, 7]
3	mergeSort(arr, 3, 4)	[5, 6]
4	mergeSort(arr, 3, 3)	[5]
4	mergeSort(arr, 4, 4)	[6]
3	merge(arr, 3, 3, 4)	merge [5] and [6] ⇒ [5, 6]
3	mergeSort(arr, 5, 5)	[7]
2	merge(arr, 3, 4, 5)	merge [5, 6] and [7] ⇒ [5, 6, 7]

```

int main() {
    MergeSort solution;

    // Hardcoded input array
    vector<int> arr = {12, 11, 13, 5, 6, 7};
    int n = arr.size();

    cout << "Given Array:" << endl;
    for (int num : arr) {
        cout << num << " ";
    }
    cout << endl;

    solution.mergeSort(arr, 0, n - 1);

    cout << "\nSorted array:" << endl;
    for (int num : arr) {
        cout << num << " ";
    }
    cout << endl;

    return 0;
}

```

Given Array:

12 11 13 5 6 7

Sorted array:

5 6 7 11 12 13

1	merge(arr, 0, 2, 5)	merge [11, 12, 13] and [5, 6, 7] \Rightarrow [5, 6, 7, 11, 12, 13]
---	---------------------	--

❖ Final Sorted Array:

[5, 6, 7, 11, 12, 13]

■ Visual of Merges

Initial: [12, 11, 13, 5, 6, 7]
 Split1: [12, 11, 13] | [5, 6, 7]
 Split2: [12, 11] [13] | [5, 6] [7]
 Merge1: [11, 12] + [13] = [11, 12, 13]
 Merge2: [5, 6] + [7] = [5, 6, 7]
 Final Merge: [11, 12, 13] + [5, 6, 7] = [5, 6, 7, 11, 12, 13]

Order of removal in C++

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

class OrderOfRemoval {
public:
    static int orderOfRemoval(vector<int>& arr) {
        int n = arr.size();
        sort(arr.begin(), arr.end()); // Sorting the array

        int ans = 0;
        for (int i = 0; i < n; i++) {
            int temp = arr[i] * (n - i);
            ans += temp;
        }

        return ans;
    }
};

int main() {
    // Hardcoded input array
    vector<int> arr = {1, 2, 3, 4, 5};
    int n = arr.size();

    // Calling orderOfRemoval function to calculate the
    // order of removal
    int result = OrderOfRemoval::orderOfRemoval(arr);

    // Printing the result
    cout << "Order of removal: " << result << endl;

    return 0;
}
```

Order of removal: 35

Let's perform a **detailed dry run** of your orderOfRemoval function using the input array:

$\text{arr} = \{1, 2, 3, 4, 5\}$

Step-by-step Dry Run:

- Sort the array:** The input array $\{1, 2, 3, 4, 5\}$ is already sorted, so no changes are made.
Sorted array: $\{1, 2, 3, 4, 5\}$
- Initialize Variables:**
 - $n = \text{arr.size()} = 5$
 - $ans = 0$ (This will hold the final result)
- Iterate and calculate the result:** For each element $\text{arr}[i]$ in the array, the contribution of that element to the ans is calculated by multiplying $\text{arr}[i]$ with the remaining elements (i.e., $\text{arr}[i] * (n - i)$).

Dry Run Table:

i	arr[i]	n - i	arr[i] * (n - i)	Cumulative ans
0	1	5	$1 * 5 = 5$	$0 + 5 = 5$
1	2	4	$2 * 4 = 8$	$5 + 8 = 13$
2	3	3	$3 * 3 = 9$	$13 + 9 = 22$
3	4	2	$4 * 2 = 8$	$22 + 8 = 30$
4	5	1	$5 * 1 = 5$	$30 + 5 = 35$

Final Result:

After the loop finishes, the value of ans is 35.

So, the output of the program is:

Order of removal: 35

Subsets in C++

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

class Subsets {
public:
    vector<vector<int>> subsets(vector<int>& nums) {
        int n = nums.size();
        int totalno = (1 << n);
        vector<vector<int>> ans;

        for (int i = 0; i < totalno; i++) {
            vector<int> temp;
            for (int j = 0; j < n; j++) {
                if (checkBit(i, j)) {
                    temp.push_back(nums[j]);
                }
            }
            ans.push_back(temp);
        }

        return ans;
    }

private:
    // Helper function to check if the i-th bit in n is set
    bool checkBit(int n, int i) {
        return (n & (1 << i)) != 0;
    }
};

int main() {
    // Create an instance of the Subsets class
    Subsets solution;

    // Hardcoded input array
    vector<int> nums = {1, 2, 3}; // Example input

    // Calling subsets to generate all subsets of the
    // array
    vector<vector<int>> subsets =
    solution.subsets(nums);

    // Printing all subsets
    for (auto& subset : subsets) {
        cout << "[";
        for (size_t i = 0; i < subset.size(); i++) {
            cout << subset[i];
            if (i < subset.size() - 1) {
                cout << ", ";
            }
        }
        cout << "]" << endl;
    }

    return 0;
}
```

Detailed Table:

i (Binary)	Subset Indexes	Subset Elements	Subset
0 (000)	None	None	[]
1 (001)	0	{1}	[1]
2 (010)	1	{2}	[2]
3 (011)	0, 1	{1, 2}	[1, 2]
4 (100)	2	{3}	[3]
5 (101)	0, 2	{1, 3}	[1, 3]
6 (110)	1, 2	{2, 3}	[2, 3]
7 (111)	0, 1, 2	{1, 2, 3}	[1, 2, 3]

Explanation of Each Iteration:

1. **Iteration 1 (i = 0 / Binary 000):**
 - o No bits are set, so the subset is empty: [].
2. **Iteration 2 (i = 1 / Binary 001):**
 - o Only the least significant bit is set, so the subset includes only the element 1: [1].
3. **Iteration 3 (i = 2 / Binary 010):**
 - o The second bit is set, so the subset includes only the element 2: [2].
4. **Iteration 4 (i = 3 / Binary 011):**
 - o The first and second bits are set, so the subset includes the elements 1 and 2: [1, 2].
5. **Iteration 5 (i = 4 / Binary 100):**
 - o The third bit is set, so the subset includes only the element 3: [3].
6. **Iteration 6 (i = 5 / Binary 101):**
 - o The first and third bits are set, so the subset includes the elements 1 and 3: [1, 3].
7. **Iteration 7 (i = 6 / Binary 110):**
 - o The second and third bits are set, so the subset includes the elements 2 and 3: [2, 3].
8. **Iteration 8 (i = 7 / Binary 111):**
 - o All bits are set, so the subset includes all elements: [1, 2, 3].

Final Output:

The final list of subsets is:

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

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

Heapsort in C++

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

void heapify(int arr[], int n, int i) {
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    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]);
        heapify(arr, n, largest);
    }
}

void heapSort(int arr[], int n) {
    for(int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

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

int main() {
    int arr[] = {12, 11, 13, 5, 6, 7};
    int n = sizeof(arr)/sizeof(arr[0]);

    heapSort(arr, n);

    cout << "Sorted array is \n";
    for(int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    return 0;
}
```

Sorted array is

5 6 7 11 12 13

Step-by-Step Dry Run

❖ Step 1: Build Max Heap

Indices:

0: 12 1: 11 2: 13 3: 5 4: 6 5: 7

Start from i = 2 (last non-leaf node)

i	Heapify Subtree	Max-Heap after heapify
2	[13, 7]	No change
1	[11, 5, 6]	No change
0	[12, 11, 13, 5, 6, 7]	swap 12 with 13 → heapify(2) swaps 12 with 7 → Done

❖ Max Heap Built:

[13, 11, 7, 5, 6, 12]

❖ Step 2: Extract Elements & Heapify

We now swap root with last element and reduce heap size (n--) after each step:

i	Swap arr[0] & arr[i]	Array after swap	Heapify to max heap
5	swap(13, 12)	[12, 11, 7, 5, 6, 13]	→ heapify → [11, 12, 7...] → [11, 6, 7, 5, 12, 13]
4	swap(11, 6)	[6, 5, 7, 11, 12, 13]	→ heapify → [7, 5, 6...]
3	swap(7, 5)	[5, 6, 7, 11, 12, 13]	→ heapify → [6, 5, ...]
2	swap(6, 5)	[5, 6, 7, 11, 12, 13]	→ heapify → [5, 6, ...] (already heap)
1	swap(5, 5)	Done	

❖ Final Output

Sorted array is

5 6 7 11 12 13

Insertion Sort in C++

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

// void insertionSort(int arr[], int n) {
//     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=j-1;
//             }
//             arr[j + 1] = key;
//         }
//     }

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

int main() {
    int arr[] = {12, 11, 13, 5, 6};
    int n = sizeof(arr)/sizeof(arr[0]);
    insertionSort(arr, n);
    cout << "Sorted array: \n";
    for(int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    return 0;
}
```

5 6 11 12 13

Input:
arr[] = {12, 11, 13, 5, 6}

Step-by-Step Dry Run (Tabular Form)

i (loop index)	j (inner loop)	Comparison	Action	Array State
1	1	11 < 12	swap(11, 12)	[11, 12, 13, 5, 6]
2	2	13 < 12? ✗	no swap	[11, 12, 13, 5, 6]
3	3	5 < 13 → swap	[11, 12, 5, 13, 6]	
	2	5 < 12 → swap	[11, 5, 12, 13, 6]	
	1	5 < 11 → swap	[5, 11, 12, 13, 6]	
4	4	6 < 13 → swap	[5, 11, 12, 6, 13]	
	3	6 < 12 → swap	[5, 11, 6, 12, 13]	
	2	6 < 11 → swap	[5, 6, 11, 12, 13]	

Final Output:

Sorted array:

5 6 11 12 13

MergeSort in C++

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

void merge(int arr[], int l, int m, int r) {
    int n1=m-l+1;
    int n2=r-m;
    int left[n1];
    int 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];
            i++;
        }else{
            arr[k]=right[j];
            j++;
        }
        k++;
    }

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

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

void mergeSort(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() {
    /* Enter your code here. Read input from STDIN.
    Print output to STDOUT */

    int n;
    cin >> n;
    int arr[n];
    for (int i = 0; i < n; i++)
    {
```

Example Input:

n = 6

arr = {38, 27, 43, 3, 9, 82}

 Merge Sort Recursive Dry Run (Call Stack Overview)

Call Level	Function Call	Action	Array State
1	mergeSort(0, 5)	Split at 2	
2	mergeSort(0, 2)	Split at 1	
3	mergeSort(0, 1)	Split at 0	
4	mergeSort(0, 0)	Base case	[38]
4	mergeSort(1, 1)	Base case	[27]
3	merge(0, 0, 1)	Merge [38] & [27] → [27, 38]	[27, 38, 43, 3, 9, 82]
2	mergeSort(2, 2)	Base case	[43]
2	merge(0, 1, 2)	Merge [27, 38] & [43]	[27, 38, 43, 3, 9, 82]
1	mergeSort(3, 5)	Split at 4	
2	mergeSort(3, 4)	Split at 3	
3	mergeSort(3, 3)	Base case	[3]
3	mergeSort(4, 4)	Base case	[9]
2	merge(3, 3, 4)	Merge [3] & [9] → [3, 9]	[27, 38, 43, 3, 9, 82]
1	mergeSort(5, 5)	Base case	[82]
1	merge(3, 4, 5)	Merge [3, 9] & [82]	[27, 38, 43, 3, 9, 82]
0	merge(0, 2, 5)	Merge [27, 38, 43] & [3, 9, 82] →	

```
    cin >> arr[i];
}

mergeSort(arr,0,n-1);

for (int i = 0; i < n; i++)
{
    cout << arr[i] << " ";
}
cout << endl;
return 0;
}
```

```
3 9 27 38 43 82
```

		[3, 9, 27, 38, 43, 82]	
<p>↙ Final Output: 3 9 27 38 43 82</p>			

Quick Sort in C++

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

int medianOfThree(int arr[], int l, int h) {
    int mid = l + (h - l) / 2;
    if (arr[l] > arr[mid])
        swap(arr[l], arr[mid]);
    if (arr[l] > arr[h])
        swap(arr[l], arr[h]);
    if (arr[mid] > arr[h])
        swap(arr[mid], arr[h]);
    return mid;
}

int partition(int arr[], int l, int h) {
    int medianIndex =
    medianOfThree(arr, l, h);
    swap(arr[l],
    arr[medianIndex]); // Move
    median to start as pivot

    int pivot = arr[l];
    int left = l + 1;
    int right = h;

    while (left <= right) {
        while (left <= right &&
        arr[left] < pivot) left++;
        while (left <= right &&
        arr[right] > pivot) right--;

        if (left <= right) {
            swap(arr[left],
            arr[right]);
            left++;
            right--;
        }
    }

    swap(arr[l], arr[right]); // Put
    pivot in correct place
    return right;
}

void rquicksort(int arr[], int l, int h) {
    if (l < h) {
        int pivot = partition(arr,
        l, h);
        rquicksort(arr, l, pivot -
        1);
        rquicksort(arr, pivot + 1,
        h);
    }
}

int main() {
    int arr[] = {24, 97, 40, 67,
    88, 85, 15};
}
```

Here's a **dry run** of your Quicksort code **in tabular form** for the input:

```
int arr[] = {24, 97, 40, 67, 88, 85, 15};
```

We'll trace:

- Recursive calls
- Chosen pivot (via median-of-three)
- Partitioning process
- Array state after each step

🔗 Step-by-Step Dry Run Table:

Step	Subarray (l to h)	Median-of-Three	Pivot	Final Pivot Index	Array After Partition
1	arr[0..6] = {24,97,40,67,88,85,15}	40 (mid=2)	40	2	{24,15,40,67,88,85,97}
2	arr[0..1] = {24,15}	15 (mid=0)	15	0	{15,24,40,...}
3	arr[1..1] = {24}	-	-	-	(Base case, already sorted)
4	arr[3..6] = {67,88,85,97}	85 (mid=4)	85	4	{...,67,85,88,97}
5	arr[3..3] = {67}	-	-	-	(Base case)
6	arr[5..6] = {88,97}	88 (mid=5)	88	5	{...,67,85,88,97} (already sorted)

```
int n = sizeof(arr) /  
sizeof(arr[0]);  
  
rquicksort(arr, 0, n - 1);  
  
cout << "Sorted array: ";  
for (int i = 0; i < n; i++) {  
    cout << arr[i] << " ";  
}  
cout << endl;  
  
return 0;  
}
```

15, 24, 40, 67, 85, 88, 97

 **Final Sorted Array:**

{15, 24, 40, 67, 85, 88, 97}

Selection in C++

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

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

    int main()
    {
        int arr[] = {64, 25, 12, 22, 11};
        int n = sizeof(arr)/sizeof(arr[0]);
        selectionSort(arr, n);
        cout << "Sorted array: \n";
        for(int i = 0; i < n; i++)
        {
            cout << arr[i] << " ";
        }
        return 0;
    }
}
```

Input:

arr[] = {64, 25, 12, 22, 11}

■ Selection Sort Dry Run Table

Pass	i	Initial minidx	Comparisons	New minidx	Swap (arr[i] ↔ arr[minidx])	Array after pass
1	0	0 (64)	25 < 64 → 12 < 25 → 22 < 12 → 11 < 12	4 (11)	64 ↔ 11	[11, 25, 12, 22, 64]
2	1	1 (25)	12 < 25 → 22 < 12	2 (12)	25 ↔ 12	[11, 12, 25, 22, 64]
3	2	2 (25)	22 < 25	3 (22)	25 ↔ 22	[11, 12, 22, 25, 64]
4	3	3 (25)	64 > 25	3	25 ↔ 25 (no change)	[11, 12, 22, 25, 64]

❖ Final Output:

Sorted array:
11 12 22 25 64

11 12 22 25 64

Iterative Binary search in C++

```
#include <iostream>
#include <vector>

using namespace std;

int binsearch(const vector<int>& arr, int x) {
    int low = 0, high = arr.size() - 1;

    while (low <= high) {
        int mid = (low + high) / 2;
        if (arr[mid] == x) {
            return mid;
        } else if (arr[mid] > x) {
            high = mid - 1;
        } else {
            low = mid + 1;
        }
    }
    return -1;
}

int main() {
    vector<int> arr = {3, 5, 7, 8, 9};
    cout << binsearch(arr, 8) << endl;
    return 0;
}
```

Input Details

- arr = {3, 5, 7, 8, 9}
- x = 8

Binary Search Table

Step	low	high	mid	arr[mid]	Comparison	Action
1	0	4	(0+4)/2 = 2	7	7 < 8 → false	low = mid + 1 → 3
2	3	4	(3+4)/2 = 3	8	8 == 8 → true	Return 3

Output

3

3

Binary Recursive in C++

```
#include <iostream>
#include <vector>

using namespace std;

int binsearch(const vector<int>& arr, int low, int high, int x) {
    if (low > high) {
        return -1;
    }
    int mid = (low + high) / 2;
    if (arr[mid] == x) {
        return mid;
    } else if (arr[mid] > x) {
        return binsearch(arr, low, mid - 1,
x);
    } else {
        return binsearch(arr, mid + 1, high,
x);
    }
}

int main() {
    vector<int> arr = {3, 5, 7, 8, 9, 11, 45,
76};
    int result = binsearch(arr, 0, arr.size() -
1, 11);
    cout << result << endl;
    return 0;
}
```

5

Here's a **tabular dry run** of the **recursive binary search** code for:

arr = {3, 5, 7, 8, 9, 11, 45, 76}
x = 11

Dry Run Table

Call #	low	high	mid = (low+high)/2	arr[mid]	Comparison	Action
1	0	7	(0+7)/2 = 3	8	8 < 11	Search right → low = mid+1 = 4
2	4	7	(4+7)/2 = 5	11	11 == 11	Found → return 5

Output

5

Pair with Given Sum in C++

```
#include <iostream>
#include <vector>

using namespace std;

bool pairWithGivenSum(const vector<int>& arr, int x)
{
    int left = 0, right = arr.size() - 1;

    while (left < right) {
        if (arr[left] + arr[right] == x) {
            return true;
        } else if (arr[left] + arr[right] > x) {
            right--;
        } else {
            left++;
        }
    }
    return false;
}

int main() {
    vector<int> arr = {10, 7, 8, 20, 12};
    int x = 32;

    cout << std::boolalpha << pairWithGivenSum(arr,
x) << endl;
    return 0;
}
```

true

Dry Run After Sorting

Sorted array: {7, 8, 10, 12, 20}
Target x = 32

left (val)	right (val)	Sum	Action
7	20	27	Increase left
8	20	28	Increase left
10	20	30	Increase left
12	20	32	Match → return true

↙ Output:

true

Sqrt in C++

```
#include <iostream>

using namespace std;

int sqrt(int x) {
    if (x == 0 || x == 1) {
        return x;
    }

    int low = 1, high = x, ans = 0;
    while (low <= high) {
        int mid = low + (high - low) / 2;
        long long mSqr = (long long) mid * mid; // Use
        long long to avoid integer overflow

        if (mSqr == x) {
            return mid;
        } else if (mSqr > x) {
            high = mid - 1;
        } else {
            low = mid + 1;
            ans = mid;
        }
    }
    return ans;
}

int main() {
    cout << sqrt(37) << endl;
    return 0;
}
```

Dry Run Table:

Iteration	low	high	mid	mid*mid	ans	Action
1	1	37	19	361	0	361 > 37 → high = mid - 1 = 18
2	1	18	9	81	0	81 > 37 → high = mid - 1 = 8
3	1	8	4	16	0	16 < 37 → ans = 4, low = mid + 1 = 5
4	5	8	6	36	4	36 < 37 → ans = 6, low = mid + 1 = 7
5	7	8	7	49	6	49 > 37 → high = mid - 1 = 6
End	7	6	-	-	6	Loop ends since low > high

✓ Final Result:

6

Chocolate Distribution in C++

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

class ChocolateDistribution {
public:
    static int find(vector<int>& arr, int n, int m) {
        // Sort the array of weights
        sort(arr.begin(), arr.end());

        int minDifference = INT_MAX;

        // Find the minimum difference between
        maximum and minimum weights in subarrays of size
        m
        for (int i = 0; i <= n - m; ++i) {
            int minWeight = arr[i];
            int maxWeight = arr[i + m - 1];
            int difference = maxWeight - minWeight;

            if (difference < minDifference) {
                minDifference = difference;
            }
        }

        return minDifference;
    }

    int main() {
        // Hardcoded input
        int n = 8;
        vector<int> arr = {3, 4, 1, 9, 56, 7, 9, 12};
        int m = 5;

        // Call the find method to get the minimum
        difference
        int ans = ChocolateDistribution::find(arr, n, m);

        // Print the result
        cout << ans << endl;

        return 0;
    }
}
```

Inputs:

$\text{arr} = \{3, 4, 1, 9, 56, 7, 9, 12\}$
 $n = 8$
 $m = 5$

Step 1: Sort the array

Sorted $\text{arr} = \{1, 3, 4, 7, 9, 9, 12, 56\}$

Step 2: Sliding window of size $m = 5$

We'll check all subarrays of length $m = 5$ and calculate $\max - \min$.

i	Subarray	Min ($\text{arr}[i]$)	Max ($\text{arr}[i + m - 1]$)	Difference
0	{1, 3, 4, 7, 9}	1	9	8
1	{3, 4, 7, 9, 9}	3	9	6
2	{4, 7, 9, 9, 12}	4	12	8
3	{7, 9, 9, 12, 56}	7	56	49

❖ Minimum Difference:

From the table above, the **minimum difference** is 6 (from subarray {3, 4, 7, 9, 9}).

▣ Final Output:

6

Count Triplets in C++

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

class CountTheTriplets {
public:
    static int countTriplets(int arr[], int n)
    {
        // Sort the array
        sort(arr, arr + n);
        int count = 0;

        // Traverse the array from the end to
        // find triplets
        for (int i = n - 1; i >= 2; i--) {
            int left = 0, right = i - 1;

            // Two pointers technique to find
            // triplets
            while (left < right) {
                if (arr[left] + arr[right] ==
                    arr[i]) {
                    // If valid triplet is found
                    count++;
                    left++;
                    right--;
                } else if (arr[left] + arr[right] <
                    arr[i]) {
                    // Move left pointer to
                    // increase the sum
                    left++;
                } else {
                    // Move right pointer to
                    // decrease the sum
                    right--;
                }
            }

            return count;
        }
    };

    int main() {
        // Hardcoded input
        int n = 6;
        int arr[] = {1, 3, 5, 2, 7, 4};

        // Call the countTriplets method to
        // count triplets
        int result =
        CountTheTriplets::countTriplets(arr, n);

        // Print the result
        cout << "Number of triplets: " << result
        << endl;

        return 0;
    }
}
```

Count the number of **triplets (i, j, k)** in the array such that:

$$\text{arr}[i] + \text{arr}[j] == \text{arr}[k]$$

Where i, j, and k are **distinct indices**.

❖ Input Array:

$$\text{arr[]} = \{1, 3, 5, 2, 7, 4\}$$

$$n = 6$$

❖ After Sorting:

$$\text{arr[]} = \{1, 2, 3, 4, 5, 7\}$$

$$\begin{matrix} \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow \\ 0 & 1 & 2 & 3 & 4 & 5 \end{matrix} \text{(indexes)}$$

❖ Dry Run Table:

i (arr[i])	left	right	arr[left] + arr[right]	Comparison	Action	Count
5 (7)	0	4	1 + 5 = 6	< 7	left++ → left=1	0
	1	4	2 + 5 = 7	== 7 → Triplet found!	count++, left++, right--	1
	2	3	3 + 4 = 7	== 7 → Triplet found!	count++, left++, right--	2
4 (5)	0	3	1 + 4 = 5	== 5 → Triplet found!	count++, left++, right--	3
	1	2	2 + 3 = 5	== 5 → Triplet found!	count++, left++, right--	4
3 (4)	0	2	1 + 3 = 4	== 4 → Triplet found!	count++, left++, right--	5
	1	1	loop ends			
2 (3)	0	1	1 + 2 = 3	== 3 → Triplet found!	count++, left++, right--	6

❖ Final Output:

Number of triplets: 6

❖ Triplets Found:

- (2, 5) → 2 + 5 = 7

- | | |
|-----------------------|---|
| | <ul style="list-style-type: none">• $(3, 4) \rightarrow 3 + 4 = 7$• $(1, 4) \rightarrow 1 + 4 = 5$• $(2, 3) \rightarrow 2 + 3 = 5$• $(1, 3) \rightarrow 1 + 3 = 4$• $(1, 2) \rightarrow 1 + 2 = 3$ |
| Number of triplets: 6 | |

Count Zeros In Sorted Matrix in C++

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

class CountZerosInASortedMatrix {
public:
    static int countZeros(vector<vector<int>>& mat) {
        int n = mat.size();
        int i = 0;
        int j = n - 1;
        int countZeros = 0;

        while (i < n && j >= 0) {
            if (mat[i][j] == 1) {
                j--;
            } else {
                countZeros += j + 1;
                i++;
            }
        }

        return countZeros;
    }
};

int main() {
    // Hardcoded input
    int n = 5;
    vector<vector<int>> mat = {
        {0, 0, 0, 1, 1},
        {0, 0, 0, 1, 1},
        {0, 0, 1, 1, 1},
        {0, 1, 1, 1, 1},
        {0, 1, 1, 1, 1}
    };

    // Call the countZeros method to count zeros
    int result =
    CountZerosInASortedMatrix::countZeros(mat);

    // Print the result
    cout << "Number of zeros in the sorted matrix: " <<
    result << endl;

    return 0;
}
```

Dry Run Table

Matrix:

```
0 0 0 1 1
0 0 0 1 1
0 0 1 1 1
0 1 1 1 1
0 1 1 1 1
```

i	j	mat[i][j]	Action	Zeros Count
0	4	1	j-- → 3	0
0	3	1	j-- → 2	0
0	2	0	count += 2+1=3, i++	3
1	2	0	count += 2+1=3, i++	6
2	2	1	j-- → 1	6
2	1	0	count += 1+1=2, i++	8
3	1	1	j-- → 0	8
3	0	0	count += 0+1=1, i++	9
4	0	0	count += 0+1=1, i++	10

✓ Final Output:

Number of zeros in the sorted matrix: 10

Number of zeros in the sorted matrix: 10

Facing the sun in C++

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

class FacingTheSun {
public:
    static int countBuildings(vector<int>& ht) {
        int lmax = ht[0];
        int count = 1;

        for (int i = 1; i < ht.size(); i++) {
            if (ht[i] > lmax) {
                count++;
                lmax = ht[i];
            }
        }

        return count;
    }

    int main() {
        // Hardcoded input
        int n = 6;
        vector<int> ht = {7, 4, 8, 2, 9, 6};

        // Call the countBuildings function to count
        // buildings facing the sun
        int result = FacingTheSun::countBuildings(ht);

        // Print the result
        cout << "Number of buildings facing the sun: " <<
        result << endl;

        return 0;
    }
}
```

Number of buildings facing the sun: 3

Input:

ht = {7, 4, 8, 2, 9, 6}

Q Dry Run Table:

Index (i)	Height ht[i]	Current lmax	Is ht[i] > lmax?	Count	New lmax
0	7	7	- (first building)	1	7
1	4	7	No	1	7
2	8	7	Yes	2	8
3	2	8	No	2	8
4	9	8	Yes	3	9
5	6	9	No	3	9

❖ Final Result:

Number of buildings facing the sun = 3

➲ Output:

Number of buildings facing the sun: 3

Find K closest elements in C++

```
#include <iostream>
#include <vector>
#include <cstdlib> // for abs function
#include <algorithm> // for sort function
using namespace std;

class FindKClosestElements {
public:
    static vector<int> findClosest(vector<int>& arr, int k, int x)
    {
        int lo = 0;
        int hi = arr.size() - 1;

        // Using binary search to find the
        // range of k closest elements
        while (hi - lo >= k) {
            if (abs(arr[lo] - x) > abs(arr[hi] - x)) {
                lo++;
            } else {
                hi--;
            }
        }

        // Extract the k closest elements into
        // a vector
        vector<int> result(arr.begin() + lo,
                           arr.begin() + lo + k);

        return result;
    }

    int main()
    {
        // Hardcoded input
        vector<int> arr = {10, 20, 30, 40, 50,
                           60};
        int k = 3;
        int x = 45;

        // Call the findClosest function to find k
        // closest elements to x
        vector<int> ans =
        FindKClosestElements::findClosest(arr,
                                         k, x);

        // Print the closest elements
        cout << "Closest elements to " << x <<
        ":";

        for (int val : ans) {
            cout << val << " ";
        }
        cout << endl;

        return 0;
    }
}
```

Closest elements to 45: 30 40 50

Here's a **detailed tabular dry run** of your code using the input:

arr = {10, 20, 30, 40, 50, 60}
k = 3
x = 45

💡 Goal:

Find the **k = 3** elements in arr that are **closest to x = 45** using the two-pointer approach.

Initial Setup:

- $lo = 0, hi = 5$ (last index)
- Keep shrinking the window from either end until $hi - lo + 1 == k$

💡 Step-by-Step Table:

Step	lo	hi	hi - lo	$abs(arr[lo] - x)$ = 35	$abs(arr[hi] - x)$ = 15	Decision	New lo	New hi
1	0	5	5	$abs(10 - 45)$ = 35	$abs(60 - 45)$ = 15	$35 > 15 \rightarrow$ shrink left	1	5
2	1	5	4	$abs(20 - 45)$ = 25	$abs(60 - 45)$ = 15	$25 > 15 \rightarrow$ shrink left	2	5
3	2	5	3	$abs(30 - 45)$ = 15	$abs(60 - 45)$ = 15	Equal → shrink right	2	4

Now, $hi - lo + 1 = 3$, so stop.

💡 Final Window:

arr[2] to arr[4] → {30, 40, 50}

Closest elements to 45 are:

30 40 50

▣ Final Output:

Closest elements to 45: 30 40 50

Find Rotation Count in C++

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

int
findRotationCount(vector<int>& arr) {
    int lo = 0;
    int hi = arr.size() - 1;

    // If the array is not
    // rotated, return 0
    if (arr[lo] <= arr[hi]) {
        return 0;
    }

    while (lo <= hi) {
        int mid = lo + (hi - lo) /
2;

        // Check if mid is the
        // pivot element
        if (mid < hi &&
arr[mid] > arr[mid + 1]) {
            return mid + 1;
        }
        // Check if mid-1 is the
        // pivot element
        else if (mid > lo &&
arr[mid] < arr[mid - 1]) {
            return mid;
        }
        // If arr[lo] <= arr[mid],
        // it means the left half is
        // sorted, so pivot is in the
        // right half
        else if (arr[lo] <=
arr[mid]) {
            lo = mid + 1;
        }
        // Otherwise, pivot is in
        // the left half
        else {
            hi = mid - 1;
        }
    }

    return 0; // Should not
    // reach here in a rotated
    // sorted array scenario
}

int main() {
    // Hardcoded input
    vector<int> arr = {4, 5, 6,
7, 8, 0, 1, 2};

    // Call the
    findRotationCount function
    // to find the rotation count
    int ans =
```

Input:

```
vector<int> arr = {4, 5, 6, 7, 8, 0, 1, 2};
```

This is a sorted array rotated **5 times**. Let's trace it step-by-step.

Initial Setup:

- $lo = 0, hi = 7$
- Condition: If $arr[lo] \leq arr[hi]$, return 0 — not true here ($4 > 2$)

Q Detailed Step-by-Step Table:

Step	lo	hi	mid	arr[mid]	arr[mid+1]	arr[mid-1]	Condition Met	Explanation & Action
1	0	7	$(0+7)/2 = 3$	7	8	6	$arr[lo] \leq arr[mid]$ $\rightarrow 4 \leq 4$ $arr[mid-1] \rightarrow 7$	Left half is sorted → move right: $lo = mid + 1 = 4$
2	4	7	$(4+7)/2 = 5$	0	1	8	$arr[mid] < arr[mid-1]$ $\rightarrow 0 < 8 \swarrow$	Pivot found → return mid = 5

❖ Final Output:

5

```
findRotationCount(arr);  
    // Print the rotation count  
    cout << ans << endl;  
    return 0;  
}  
5
```

Find Transition in C++

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

int findTransition(vector<int>& arr) {
    int tp = -1;
    int lo = 0;
    int hi = arr.size() - 1;

    while (lo <= hi) {
        int mid = lo + (hi - lo) / 2;

        if (arr[mid] == 1) {
            tp = mid;
            hi = mid - 1; // Look for earlier occurrences on
            the left side
        } else {
            lo = mid + 1; // If arr[mid] is 0, move to the
            right half
        }
    }

    return tp;
}

int main() {
    // Hardcoded input
    vector<int> arr = {0, 0, 0, 0, 1, 1};

    // Call the findTransition function to find the index
    // of the first occurrence of 1
    int ans = findTransition(arr);

    // Print the index of the first occurrence of 1
    cout << ans << endl;

    return 0;
}
```

Input:

arr = {0, 0, 0, 0, 1, 1}

Goal:

Find the **index of the first occurrence of 1** using **binary search**.

Q Dry Run Table:

Iteration	lo	hi	mid	arr[mid]	tp	Action Taken
1	0	5	2	0	-1	Move right: lo = mid + 1 = 3
2	3	5	4	1	4	Move left: hi = mid - 1 = 3
3	3	3	3	0	4	Move right: lo = mid + 1 = 4

✓ Final Values:

- tp = 4
- So, the **first occurrence of 1 is at index 4.**

▀ Output:

4

4

Largest Number in C++

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

// Custom comparator function for sorting strings in
// descending order
bool compare(string a, string b) {
    string ab = a + b;
    string ba = b + a;
    return ab > ba; // Compare in descending order
}

string largestNumber(vector<int>& nums) {
    // Convert integers to strings
    vector<string> arr(nums.size());
    for (int i = 0; i < nums.size(); ++i) {
        arr[i] = to_string(nums[i]);
    }

    // Sort using custom comparator
    sort(arr.begin(), arr.end(), compare);

    // Construct the result string
    if (arr[0] == "0") { // Special case to handle if all
    // nums are zeroes
        return "0";
    }

    string result;
    for (const string& s : arr) {
        result += s;
    }

    return result;
}

int main() {
    vector<int> nums = {3, 7, 34, 5, 9};
    cout << largestNumber(nums) << endl;

    return 0;
}
```

Input:

vector<int> nums = {3, 7, 34, 5, 9};

Step 1: Convert Integers to Strings

Index	Integer	String
0	3	"3"
1	7	"7"
2	34	"34"
3	5	"5"
4	9	"9"

Step 2: Custom Sorting (Using compare(a, b) \Rightarrow a + b > b + a)

Sorted Comparisons

Pair	a + b	b + a	Result
"9", "5"	"95"	"59"	"9" > "5"
"9", "34"	"934"	"349"	"9" > "34"
"5", "3"	"53"	"35"	"5" > "3"
"7", "3"	"73"	"37"	"7" > "3"
"34", "3"	"343"	"334"	"34" > "3"

→ After sorting with custom comparator:

Index String

0	"9"
1	"7"
2	"5"
3	"34"
4	"3"

Step 3: Concatenate Sorted Strings

result = "9" + "7" + "5" + "34" + "3" = "975343"

	<p>✓ Final Output:</p> <p>975343</p>
975343	

Largest Perimeter triangle in C++

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int largestPerimeter(vector<int>& nums)
{
    sort(nums.begin(), nums.end());

    int p = 0;

    for (int i = nums.size() - 1; i >= 2; --i) {
        if (nums[i - 1] + nums[i - 2] > nums[i]) {
            p = nums[i - 1] + nums[i - 2] +
            nums[i];
            break;
        }
    }

    return p;
}

int main() {
    vector<int> nums = {25, 6, 9, 11, 8, 12,
10, 3, 2};

    cout << largestPerimeter(nums) <<
endl;

    return 0;
}
```

Step-by-step check after sorting:

nums = {2, 3, 6, 8, 9, 10, 11, 12, 25}

We're looping from the end ($i = 8$) down to 2, checking this:

`if (nums[i-1] + nums[i-2] > nums[i]) // triangle inequality`

🧠 Dry Run Table with Full Checks:

i	nums[i-2]	nums[i-1]	nums[i]	Sum of two smallest	Valid triangle?	Perimeter
8	11	12	25	$11 + 12 = 23$ 23	✗ (23 < 25)	-
7	10	11	12	$10 + 11 = 21$	✓	33

So, yes — the **first valid triangle** found is {10, 11, 12}, with perimeter = 33.

☒ Why not {11, 12, 25}?

Because $11 + 12 = 23$, which is **less than 25** — **fails triangle condition**.

✓ Correct Output:

33

Marks of PCM in C++				
<pre>#include <iostream> #include <vector> #include <algorithm> using namespace std; // Define a class to represent marks class Marks { public: int phy; int chem; int math; // Constructor Marks(int p, int c, int m) { phy = p; chem = c; math = m; } // Method to compare for sorting bool operator<(const Marks& other) const { if (phy != other.phy) { return phy < other.phy; } else if (chem != other.chem) { return chem > other.chem; // Sort chem descending if phy are equal } else { return math < other.math; } } // Function to custom sort marks void customSort(vector<int>& phy, vector<int>& chem, vector<int>& math) { int n = phy.size(); vector<Marks> arr; // Populate the vector of Marks objects for (int i = 0; i < n; ++i) { arr.emplace_back(phy[i], chem[i], math[i]); } // Sort using overloaded < operator in Marks class sort(arr.begin(), arr.end()); // Update original arrays with sorted values for (int i = 0; i < n; ++i) { phy[i] = arr[i].phy; chem[i] = arr[i].chem; math[i] = arr[i].math; } } int main() { const int N = 5; vector<int> phy = {9, 5, 9, 8, 5}; vector<int> chem = {3, 4, 3, 7, 6}; vector<int> math = {15, 10, 11, 13, 12}; } }</pre>	Input Table (Before Sorting)			
Index	Phy	Chem	Math	
0	9	3	15	
1	5	4	10	
2	9	3	11	
3	8	7	13	
4	5	6	12	
 Sorting Rule Recap				
<ul style="list-style-type: none"> ✓ Primary: Phy ascending ✓ Secondary: Chem descending ✓ Tertiary: Math ascending 				
Output Table (After Sorting)				
New Index	Phy	Chem	Math	Reason
0	5	6	12	Smallest phy; chem=6 > chem=4
1	5	4	10	Same phy as above, chem is lower so placed after
2	8	7	13	Next higher phy
3	9	3	11	Same phy as next, but math is smaller so comes first
4	9	3	15	Same phy and chem as above, but math=15 > math=11, so placed after

```
// Call custom sort function
customSort(phy, chem, math);

// Output sorted marks
for (int i = 0; i < N; ++i) {
    cout << phy[i] << " " << chem[i] << " " << math[i]
<< endl;
}

return 0;
}
```

```
5 6 12
5 4 10
8 7 13
9 3 11
9 3 15
```

Pair with given difference in C++

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

void findPair(vector<int>& arr, int target) {
    sort(arr.begin(), arr.end());

    int i = 0;
    int j = 1;
    while (i < arr.size() && j < arr.size()) {
        if (arr[j] - arr[i] == target) {
            cout << arr[i] << " " << arr[j] << endl;
            return;
        } else if (arr[j] - arr[i] < target) {
            j++;
        } else {
            i++;
        }
    }
    cout << "-1" << endl;
}

int main() {
    // Hardcoded input
    vector<int> arr = {1, 7, 3, 10, 5, 6};
    int target = 4;

    // Call the findPair function to find the pair with
    // given difference
    findPair(arr, target);

    return 0;
}
```

1 5

Input:

arr = {1, 7, 3, 10, 5, 6}
target = 4

⌚ Step 1: Sort the array

arr = {1, 3, 5, 6, 7, 10}

⌚ Step 2: Two-pointer approach

We use two pointers:

- i starts at 0
 - j starts at 1
- Goal: find any two elements such that
 $\text{arr}[j] - \text{arr}[i] == \text{target}$

📝 Tabular Dry Run:

i	j	arr[i]	arr[j]	Difference	Action
0	1	1	3	2	j++
0	2	1	5	4 ↗	Print 1 5, return

❖ Output:

1 5

Union of two sorted Array in C++

```
#include <iostream>
#include <vector>

using namespace std;

vector<int> unionOfArrays(int a[], int b[], int m, int n) {
    vector<int> unionList;
    int i = 0, j = 0;

    while (i < m && j < n) {
        if (a[i] < b[j]) {
            if (unionList.empty() || unionList.back() != a[i]) {
                unionList.push_back(a[i]);
            }
            i++;
        } else if (b[j] < a[i]) {
            if (unionList.empty() || unionList.back() != b[j]) {
                unionList.push_back(b[j]);
            }
            j++;
        } else {
            if (unionList.empty() || unionList.back() != a[i]) {
                unionList.push_back(a[i]);
            }
            i++;
            j++;
        }
    }

    // Remaining elements of a, if any
    while (i < m) {
        if (unionList.empty() || unionList.back() != a[i])
        {
            unionList.push_back(a[i]);
        }
        i++;
    }

    // Remaining elements of b, if any
    while (j < n) {
        if (unionList.empty() || unionList.back() != b[j])
        {
            unionList.push_back(b[j]);
        }
        j++;
    }

    return unionList;
}

int main() {
    int a[] = {1, 2, 4};
    int b[] = {3, 5, 6};
    int m = sizeof(a) / sizeof(a[0]);
    int n = sizeof(b) / sizeof(b[0]);

    vector<int> unionList = unionOfArrays(a, b, m, n);
}
```

Input:

a[] = {1, 2, 4}
b[] = {3, 5, 6}

Expected Output:

1 2 3 4 5 6

Tabular Dry Run:

i	j	a[i]	b[j]	Comparison	Action	unionList
0	0	1	3	a[i] < b[j]	push 1, i+ +	[1]
1	0	2	3	a[i] < b[j]	push 2, i+ +	[1, 2]
2	0	4	3	b[j] < a[i]	push 3, j+ +	[1, 2, 3]
2	1	4	5	a[i] < b[j]	push 4, i+ +	[1, 2, 3, 4]
3	1	-	5	i == m	loop to remaining b	
1	-		5		push 5, j+ +	[1, 2, 3, 4, 5]
2	-		6		push 6, j+ +	[1, 2, 3, 4, 5, 6]

What this function does well:

- Merges two **sorted arrays**.
- Skips **duplicate elements** (if any).
- Maintains **sorted order** in the output.
- Uses **two-pointer approach**, which is very efficient:
 - **Time complexity:** O(m + n)
 - **Space complexity:** O(m + n) in worst case (if no duplicates)

Final Output:

1 2 3 4 5 6

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

```
1 2 3 4 5 6
```

Balanced Parenthesis in C++

```
#include <iostream>
#include <stack>
using namespace std;

bool isBal(string str) {
    stack<char> s;
    for (int i = 0; i < str.length(); i++) {
        if (str[i] == '(' || str[i] == '{' || str[i] == '[') {
            s.push(str[i]);
        } else {
            if (s.empty()) {
                return false;
            } else if ((str[i] == ')' && s.top() == '(') || (str[i] == '}' && s.top() == '{') || (str[i] == ']' && s.top() == '[')) {
                s.pop();
            } else {
                return false;
            }
        }
    }
    return s.empty();
}

int main() {
    cout << boolalpha << isBal("()") << endl; // Example usage
    return 0;
}
```

Function Purpose

Checks if the string contains balanced brackets:

- (), {}, and []

❑ Input

string str = "()"

❖ Stack Simulation Table

i	str[i]	Stack Before	Action	Stack After
0	([]	Push '('	[']
1	([']	Push '('	[',']
2)	[',']	Top '(' matches) → Pop	[']
3)	[']	Top '(' matches) → Pop	[]

❖ Final Check:

- Stack is empty → Balanced
- Output: true

❗ Output:

true

true

Largest area Histogram in C++

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

class LargestRectangleInHistogram {
public:
    int largestRectangleArea(vector<int>& heights) {
        stack<int> s;
        int ans = 0;
        for (int i = 0; i <= heights.size(); i++) {
            int temp = (i != heights.size()) ? heights[i] : 0;
            while (!s.empty() && temp < heights[s.top()]) {
                int tbs = s.top();
                s.pop();
                int nsr = i;
                int x1 = nsr - 1;
                int nsl = (s.empty()) ? -1 : s.top();
                int x2 = nsl + 1;
                int area = heights[tbs] * (x1 - x2 + 1);
                ans = max(ans, area);
            }
            s.push(i);
        }
        return ans;
    }

    int main() {
        vector<int> heights = {2, 1, 5, 6, 2, 3};
        LargestRectangleInHistogram histogram;
        int maxArea =
            histogram.largestRectangleArea(heights);
        cout << "The largest rectangle area is: " <<
        maxArea << endl;
        return 0;
    }
}
```

Step-by-step Table Dry Run

i	temp	Stack (Index)	Action	Computed Area	Max Area
0	2	[]	Push index 0	—	0
1	1	[0]	Pop 0 → height = 2, width = 1 → $2 \times 1 = 2$	2	2
		[]	Push index 1	—	2
2	5	[1]	Push index 2	—	2
3	6	[1, 2]	Push index 3	—	2
4	2	[1, 2, 3]	Pop 3 → height = 6, width = 1 → $6 \times 1 = 6$	6	6
		[1, 2]	Pop 2 → height = 5, width = 2 → $5 \times 2 = 10$	10	10
		[1]	Push index 4	—	10
5	3	[1, 4]	Push index 5	—	10
6	0	[1, 4, 5]	Pop 5 → height = 3, width = 1 → $3 \times 1 = 3$	3	10
		[1, 4]	Pop 4 → height = 2, width = 3 → $2 \times 3 = 6$	6	10
		[1]	Pop 1 → height = 1, width = 6 → $1 \times 6 = 6$	6	10
		[]	Push index 6 (extra 0 at end)	—	10

	<p>✓ Final Output:</p> <p>The largest rectangle area is: 10</p>
The largest rectangle area is: 10	

Max frequency Stack in C++

```
#include <iostream>
#include <unordered_map>
#include <stack>
using namespace std;

class MaxFrequencyStack {
private:
    unordered_map<int, stack<int>> st;
    unordered_map<int, int> fmap;
    int maxfreq;

public:
    MaxFrequencyStack() {
        maxfreq = 0;
    }

    void push(int val) {
        int f = ++fmap[val];
        st[f].push(val);
        maxfreq = max(maxfreq, f);
    }

    int pop0() {
        int val = st[maxfreq].top();
        st[maxfreq].pop();
        if (st[maxfreq].empty()) {
            st.erase(maxfreq);
            maxfreq--;
        }
        fmap[val]--;
        return val;
    }
};

int main() {
    MaxFrequencyStack freqStack;

    freqStack.push(5);
    freqStack.push(7);
    freqStack.push(5);
    freqStack.push(7);
    freqStack.push(4);
    freqStack.push(5);

    cout << freqStack.pop0() << endl; // Should print 5
    cout << freqStack.pop0() << endl; // Should print 7
    cout << freqStack.pop0() << endl; // Should print 5
    cout << freqStack.pop0() << endl; // Should print 4

    return 0;
}
```

Dry Run: Input Sequence
 push(5)
 push(7)
 push(5)
 push(7)
 push(4)
 push(5)

 pop0 → ?
 pop0 → ?
 pop0 → ?
 pop0 → ?

Dry Run Table (Tracking fmap, st, and maxfreq):

Operation	fmap	st (per freq)	maxfreq	Top Element Popped
push(5)	{5: 1}	{1: [5]}	1	—
push(7)	{5: 1, 7: 1}	{1: [5, 7]}	1	—
push(5)	{5: 2, 7: 1}	{1: [5, 7], 2: [5]}	2	—
push(7)	{5: 2, 7: 2}	{1: [5, 7], 2: [5, 7]}	2	—
push(4)	{5: 2, 7: 2, 4: 1}	{1: [5, 7], 2: [5, 7]}	2	—
push(5)	{5: 3, 7: 2, 4: 1}	{1: [5, 7, 4], 2: [5, 7], 3: [5]}	3	—
pop0	{5: 2, 7: 2, 4: 1}	3 is [5] → pop 5, 2 delete 3	5	5
pop0	{5: 2, 7: 1, 4: 1}	2 is [5, 7] → pop 7	2	7
pop0	{5: 1, 7: 1, 4: 1}	2 is [5] → pop 5, 1 delete 2	1	5
pop0	{5: 1, 7: 1, 4: 0}	1 is [5, 7, 4] → pop 4	1	4

✓ Output:

5
7
5
4

💡 Notes:

5
7
5
4

Min Stack in C++

```
#include <iostream>
#include <stack>
#include <climits>
using namespace std;

class MinStack {
private:
    stack<long long> st;
    long long minVal;

public:
    MinStack() {
        minVal = INT_MAX;
    }

    void push(int val) {
        if (st.empty()) {
            minVal = val;
            st.push(0LL);
        } else {
            long long diff = val - minVal;
            st.push(diff);
            if (val < minVal) {
                minVal = val;
            }
        }
    }

    void pop() {
        long long rem = st.top();
        st.pop();
        if (rem < 0) {
            minVal = minVal - rem;
        }
    }

    int top() {
        long long rem = st.top();
        if (rem < 0) {
            return static_cast<int>(minVal);
        } else {
            return static_cast<int>(minVal + rem);
        }
    }

    int getMin() {
        return static_cast<int>(minVal);
    }

};

int main() {
    MinStack minStack;

    minStack.push(2);
    minStack.push(0);
    minStack.push(3);
    minStack.push(0);

    cout << "Minimum value: " << minStack.getMin()
    << endl; // Should print 0
    minStack.pop();
}
```

Core Logic Recap

- st stores **differences** between the current value and minVal.
- If the pushed value is **less than** minVal, a **negative diff** is stored. This signals a **new min**.
- When popping, if the top is negative, we **recalculate the previous min** using $\text{minVal} - \text{rem}$.

💡 Test Input:

```
minStack.push(2);
minStack.push(0);
minStack.push(3);
minStack.push(0);
```

```
pop() → getMin()
pop() → getMin()
pop() → getMin()
```

██████ Dry Run Table:

Operation	Stack (diffs)	minVal	Explanation
push(2)	[0]	2	First element → diff is 0
push(0)	[0, -2]	0	$0 < 2 \rightarrow$ store diff (-2), update minVal
push(3)	[0, -2, 3]	0	$3 > 0 \rightarrow$ store diff (3), minVal unchanged
push(0)	[0, -2, 3, 0]	0	$0 = \text{minVal} \rightarrow$ store diff (0), minVal unchanged
pop()	[0, -2, 3]	0	popped 0, not negative → minVal stays
getMin()	—	0	
pop()	[0, -2]	0	popped 3 (diff=3), not negative → minVal stays
getMin()	—	0	
pop()	[0]	2	popped -2 → was a new min at the time → rollback
getMin()	—	2	

```
cout << "Minimum value: " << minStack.getMin()
<< endl; // Should print 0
minStack.pop();
cout << "Minimum value: " << minStack.getMin()
<< endl; // Should print 0
minStack.pop();
cout << "Minimum value: " << minStack.getMin()
<< endl; // Should print 2

return 0;
}
```

```
Minimum value: 0
Minimum value: 0
Minimum value: 0
Minimum value: 2
```

✓ **Output:**

```
Minimum value: 0
Minimum value: 0
Minimum value: 0
Minimum value: 2
```

Next Greater on the Right in C++

```
#include <iostream>
#include <stack>
using namespace std;

long* nextLargerElement(long* arr, int n)
{
    long* ans = new long[n];
    stack<int> st;
    for(int i = 0; i < n; i++){
        while(!st.empty() && arr[i] > arr[st.top0]){
            int idx = st.top();
            st.pop();
            ans[idx] = arr[i];
        }
        st.push(i);
    }
    while(!st.empty()){
        int idx = st.top();
        st.pop();
        ans[idx] = -1;
    }
    return ans;
}

int main() {
    long arr[] = {4, 8, 5, 2, 25};
    int n = sizeof(arr) / sizeof(arr[0]);

    long* result = nextLargerElement(arr, n);

    cout << "Resulting array:" << endl;
    for (int i = 0; i < n; i++) {
        cout << result[i] << " ";
    }
    cout << endl;

    delete[] result; // Free dynamically allocated
memory

    return 0;
}
```

Input:

arr = {4, 8, 5, 2, 25}
n = 5

Iterative Dry Run Table:

i	arr[i]	Stack (indices)	Top Value	Condition Checked	Action Taken	ans Array
0	4	[]	—	—	Push index 0	[-, -, -, -, -]
1	8	[0]	4	8 > 4 → true	Pop 0, set ans[0] = 8, push 1	[8, -, -, -, -]
2	5	[1]	8	5 > 8 → false	Push 2	[8, -, -, -, -]
3	2	[1, 2]	5	2 > 5 → false	Push 3	[8, -, -, -, -]
4	25	[1, 2, 3]	2	25 > 2 → true	Pop 3, ans[3] = 25	[8, -, -, 25, -]
		[1, 2]	5	25 > 5 → true	Pop 2, ans[2] = 25	[8, -, 25, 25, -]
		[1]	8	25 > 8 → true	Pop 1, ans[1] = 25	[8, 25, 25, 25, -]
		[]	—	—	Push 4	[8, 25, 25, 25, -]
—	—	[4]	25	Loop ends	Pop 4, set ans[4] = -1	[8, 25, 25, 25, -1]

Final Output:

Resulting array:
8 25 25 25 -1

Resulting array:
8 25 25 25 -1

Postfix 2 Prefix in C++

```
#include <iostream>
#include <stack>
using namespace std;

// Function to convert a postfix expression to a prefix expression.
string postToPre(string exp) {
    stack<string> op;
    for (int i = 0; i < exp.length(); i++) {
        char ch = exp[i];
        if (ch == '+' || ch == '-' || ch == '*' || ch == '/') {
            string val2 = op.top();
            op.pop();
            string val1 = op.top();
            op.pop();
            string cal = ch + val1 + val2;
            op.push(cal);
        } else {
            op.push(string(1, ch));
        }
    }
    return op.top();
}

int main() {
    string postfix1 = "ab+c*";
    cout << "Postfix: " << postfix1 << " -> Prefix: " <<
postToPre(postfix1) << endl; // Expected: "*+abc"

    return 0;
}
```

Postfix: ab+c* -> Prefix: *+abc

Input:

Postfix Expression = "ab+c*"
Expected Prefix = "*+abc"

■ Dry Run Table:

i	ch	Stack Before	Action	Stack After
0	'a'	[]	Operand → push "a"	["a"]
1	'b'	["a"]	Operand → push "b"	["a", "b"]
2	'+'	["a", "b"]	Operator → pop "b", "a" → form +ab, push it	["+ab"]
3	'c'	["+ab"]	Operand → push "c"	["+ab", "c"]
4	'*'	["+ab", "c"]	Operator → pop "c", "+ab" → form *+abc, push it	"*+abc"]

◇ Final Output:

Prefix: *+abc

Prefix to Postfix in C++

```
#include <iostream>
#include <stack>
#include <string>
using namespace std;

// Function to convert a prefix expression to a postfix expression.
string preToPost(string exp) {
    stack<string> op;
    int n = exp.length();
    for (int i = n - 1; i >= 0; i--) {
        char ch = exp[i];
        if (ch == '+' || ch == '-' || ch == '*' || ch == '/') {
            string val1 = op.top();
            op.pop();
            string val2 = op.top();
            op.pop();
            string cal = val1 + val2 + ch;
            op.push(cal);
        } else {
            op.push(string(1, ch));
        }
    }
    return op.top();
}

int main() {
    string prefix1 = "*+AB-CDE";
    cout << "Prefix: " << prefix1 << " -> Postfix: " <<
    preToPost(prefix1) << endl; // Expected: "ABC+DE-*"

    string prefix2 = "*-A/BC-/DEFG";
    cout << "Prefix: " << prefix2 << " -> Postfix: " <<
    preToPost(prefix2) << endl; // Expected:
    "ABC/-DE/FG-*"

    // Add more test cases as needed

    return 0;
}
```

Prefix: *+AB-CDE -> Postfix: AB+CD-*

Prefix: *-A/BC-/DEFG -> Postfix: ABC/-DE/F-*

Dry Run Table:

i (index)	ch	Stack Before	Action	Stack After
7	'E'	[]	Operand → push "E"	["E"]
6	'D'	["E"]	Operand → push "D"	["E", "D"]
5	'C'	["E", "D"]	Operand → push "C"	["E", "D", "C"]
4	'-'	["E", "D", "C"]	Operator → pop "C" & "D" → "CD-"	["E", "CD-"]
3	'B'	["E", "CD-"]	Operand → push "B"	["E", "CD-", "B"]
2	'A'	["E", "CD-", "B"]	Operand → push "A"	["E", "CD-", "B", "A"]
1	'+'	["E", "CD-", "B", "A"]	Operator → pop "A" & "B" → "AB+"	["E", "CD-", "AB+"]
0	'*'	["E", "CD-", "AB+"]	Operator → pop "AB+" & "CD-" → "AB+CD-*"	["AB+CD-*"]

Final Result:

Top of the stack: **"AB+CD-*"**

Remove adjacent duplicate in C++

```
#include <iostream>
#include <stack>
#include <string>

using namespace std;

string removeAdjacentDuplicates(string s) {
    stack<char> st;

    for (char ch : s) {
        if (!st.empty() && st.top() == ch) {
            st.pop();
        } else {
            st.push(ch);
        }
    }

    string result = "";
    while (!st.empty()) {
        result = st.top() + result;
        st.pop();
    }

    return result;
}

int main() {
    string s = "abbaca";
    cout << removeAdjacentDuplicates(s) << endl; // Output: "ca"
    return 0;
}
```

ca

Input:

string s = "abbaca";

🧠 Step-by-Step Stack Trace:

Step	Char	Stack (top to bottom)	Action
1	'a'	[a]	Push
2	'b'	[a, b]	Push
3	'b'	[a]	'b' == top → Pop
4	'a'	[]	'a' == top → Pop
5	'c'	[c]	Push
6	'a'	[c, a]	Push

⚡ Final Stack (bottom to top): c a

So result = "ca".

Smaller no on left in C++

```
#include <iostream>
#include <vector>
#include <stack>

using namespace std;

vector<int> leftSmaller(int n, int a[]) {
    vector<int> ans(n);
    stack<int> st;

    for (int i = n - 1; i >= 0; i--) {
        while (!st.empty() && a[i] < a[st.top()]) {
            int idx = st.top();
            ans[idx] = a[i];
            st.pop();
        }
        st.push(i);
    }

    while (!st.empty()) {
        int idx = st.top();
        ans[idx] = -1;
        st.pop();
    }

    return ans;
}
```

```
int main() {
    int arr[] = {4, 8, 5, 2, 25};
    int n = sizeof(arr) / sizeof(arr[0]);
```

```
    vector<int> result = leftSmaller(n, arr);
```

```
    cout << "Resulting list:" << endl;
    for (int i : result) {
        cout << i << " ";
    }
    cout << endl;

    return 0;
}
```

Resulting list:
-1 4 4 -1 2

Input:
arr = {4, 8, 5, 2, 25}

Dry Run Table:

i	arr[i]	Stack (index)	Action	ans (after step)
4	25	[]	Stack empty, push 4	[?, ?, ?, ?, ?]
3	2	[4]	2 < 25 → ans[4] = 2, pop 4; push 3	[?, ?, ?, ?, 2]
2	5	[3]	5 > 2 → push 2	[?, ?, ?, ?, 2]
1	8	[3, 2]	8 > 5 → push 1	[?, ?, ?, ?, 2]
0	4	[3, 2, 1]	4 < 8 → ans[1] = 4, pop 1; 4 < 5 → ans[2] = 4, pop 2; push 0	[?, 4, 4, ?, 2]
		[3, 0]	Final elements → set ans[3] = -1, ans[0] = -1	[-1, 4, 4, -1, 2]

Final Output:

-1 4 4 -1 2

Explanation (Index-wise):

Index	arr[i]	Left Smaller Element
0	4	-1 (nothing to the left)
1	8	4
2	5	4
3	2	-1
4	25	2

Stock Span in C++

```
#include <iostream>
#include <vector>
#include <stack>

using namespace std;

void stockSpan(vector<int>& arr) {
    stack<int> s;
    s.push(0); // Push index of the first element

    for (int i = 0; i < arr.size(); i++) {
        // Pop elements from stack while the current
        // price is greater than the price of the element at the
        // top of the stack
        while (!s.empty() && arr[s.top()] <= arr[i]) {
            s.pop();
        }

        // Calculate span (i - index at top of stack after
        // popping or i + 1 if stack is empty)
        int span = s.empty() ? (i + 1) : (i - s.top());

        // Print the span
        cout << span << " ";

        // Push the current index onto the stack
        s.push(i);
    }
}

int main() {
    // Test case: array of stock prices
    vector<int> arr = {15, 13, 12, 14, 15};
    stockSpan(arr);
    cout << endl;

    return 0;
}
```

For each day i , the span is: The number of consecutive previous days (including the current day) for which the price was **less than or equal to today's price**.

You're maintaining a stack of **indices**, and for each price:

- You **pop** indices from the stack if the current price is higher than the price at the stack's top.
- The **span** is then $i - s.top()$ or $i + 1$ if the stack is empty.

 Input:
arr = {15, 13, 12, 14, 15}

■ Dry Run Table:

Day (i)	Price	Stack (indices)	Stack (values)	Span	Explanation
0	15	[]	[]	1	Stack empty → span = 0 + 1
		[0]	[15]		Push index 0
1	13	[0]	[15]	1	13 < 15 → span = 1 - 0
		[0, 1]	[15, 13]		
2	12	[0, 1]	[15, 13]	1	12 < 13 → span = 2 - 1
		[0, 1, 2]	[15, 13, 12]		
3	14	[0, 1, 2] → pop 2, 1	[15]	3	14 > 13 & 12 → span = 3 - 0
		[0, 3]	[15, 14]		
4	15	[0, 3] → pop 3, 0	[]	5	15 >= 14, 15 → stack empty → span = 4 + 1
		[4]	[15]		

❖ Output: 1 1 1 3 5

First non-repeating character in C++

```
#include <iostream>
#include <queue>
#include <unordered_map>
using namespace std;

class FirstNonRepeatingCharacter {
public:
    string FirstNonRepeating(string A) {
        queue<char> q;
        unordered_map<char, int> hm;
        string ans(A.length(), '#');

        for (int i = 0; i < A.length(); i++) {
            char c = A[i];

            q.push(c);
            hm[c]++;
            while (!q.empty() && hm[q.front()] > 1) {
                q.pop();
            }

            if (!q.empty()) {
                ans[i] = q.front();
            }
        }

        return ans;
    }

    int main() {
        // Hardcoded input string
        string A = "aabc";

        // Create an instance of the
        FirstNonRepeatingCharacter class
        FirstNonRepeatingCharacter solution;

        // Call the FirstNonRepeating method and store the
        result
        string result = solution.FirstNonRepeating(A);

        // Print the result
        cout << result << endl;

        return 0;
    }
}
```

a#bb

Code Summary:

- Use a **queue** to maintain the order of characters.
- Use a **hash map** (unordered_map<char, int>) to count character occurrences.
- At each step:
 - Add current character to the queue.
 - Increment its count.
 - Remove characters from the front of the queue if their count > 1.
 - The front of the queue (if any) is the current **first non-repeating** character.

Dry Run for A = "aabc"

i	A[i]	Queue	Hash Map	First Non-Repeating	ans
0	'a'	a	a:1	a	a
1	'a'	a a	a:2	# (a is repeated)	a#
2	'b'	a a b → b	a:2, b:1	b	a#b
3	'c'	b c	a:2, b:1, c:1	b	a#bb

Final Output:

a#bb

Explanation:

- After 'a': only 'a' is in stream → 'a'
- After second 'a': 'a' repeats → '#'
- After 'b': 'b' is first non-repeating → 'b'
- After 'c': 'b' is still non-repeating → 'b'

Generate Binary in C++

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

vector<string> generate(int N) {
    vector<string> ans;
    queue<string> q;
    q.push("1");
    while (N-- > 0) {
        string rem = q.front();
        q.pop();
        ans.push_back(rem);
        q.push(rem + "0");
        q.push(rem + "1");
    }
    return ans;
}

int main() {
    int N = 5;
    vector<string> binaryNumbers = generate(N);
    for (string num : binaryNumbers) {
        cout << num << endl;
    }
    return 0;
}
```

Goal:

Generate the first N binary numbers (as strings) from 1 to the binary representation of N.

❖ Algorithm Overview:

- Use a **queue** to build binary numbers level-by-level (like a binary tree).
- Start with "1", then append "0" and "1" to each popped string.
- Do this N times.

❖ Dry Run for N = 5

Iteration	Queue (Before Pop)	Popped (rem)	Added to Result	Queue (After Push)
1	["1"]	"1"	"1"	["10", "11"]
2	["10", "11"]	"10"	"10"	["11", "100", "101"]
3	["11", "100", "101"]	"11"	"11"	["100", "101", "110", "111"]
4	["100", "101", "110", "111"]	"100"	"100"	["101", "110", "111", "1000", "1001"]
5	["101", "110", "111", "1000", "1001"]	"101"	"101"	["110", "111", "1000", "1001", "1010", "1011"]

▲ Final Output:

```
1
10
11
100
101
```

```
1
10
11
100
101
```

Kth number in C++

```
#include <iostream>
#include <queue>
#include <string>
using namespace std;

string kth(int k) {
    queue<string> q;
    q.push("1");
    q.push("2");

    string ans;
    for (int i = 0; i < k; i++) {
        string temp = q.front();
        q.pop();
        ans = temp;
        q.push(temp + "1");
        q.push(temp + "2");
    }

    return ans;
}

int main() {
    int k = 5;
    cout << kth(k) << endl;
    return 0;
}
```

Initial Setup:

```
queue<string> q;
q.push("1");
q.push("2");
```

Initial queue: ["1", "2"]

Dry Run Table:

Iteration (i)	Queue Before	temp (popped)	ans	Queue After Push
0	["1", "2"]	"1"	"1"	["2", "11", "12"]
1	["2", "11", "12"]	"2"	"2"	["11", "12", "21", "22"]
2	["11", "12", "21", "22"]	"11"	"11"	["12", "21", "22", "111", "112"]
3	["12", "21", "22", "111", "112"]	"12"	"12"	["21", "22", "111", "112", "121", "122"]
4	["21", "22", "111", "112", "121", "122"]	"21"	"21"	["22", "111", "112", "121", "122", "211", "212"]

Final Output:

```
cout << kth(5);
```

Since index starts at 0, on the **5th iteration** (i = 4), we return:

21

Output:

21

Reverse k elements in C++

```
#include <iostream>
#include <queue>
#include <stack>
using namespace std;

queue<int> modifyQueue(queue<int> q, int k) {
    stack<int> st;

    // Push the first k elements into a stack
    for (int i = 0; i < k; i++) {
        st.push(q.front());
        q.pop();
    }

    // Pop elements from the stack and enqueue them
    // back into the queue
    while (!st.empty()) {
        q.push(st.top());
        st.pop();
    }

    // Rotate the remaining elements in the queue
    int size = q.size();
    for (int i = 0; i < size - k; i++) {
        q.push(q.front());
        q.pop();
    }

    return q;
}

int main() {
    // Create a queue and add some elements
    queue<int> q;
    q.push(1);
    q.push(2);
    q.push(3);
    q.push(4);
    q.push(5);

    // Define the value of k
    int k = 3;

    // Call the modifyQueue function and store the
    // result
    queue<int> result = modifyQueue(q, k);

    // Print the result queue
    while (!result.empty()) {
        cout << result.front() << " ";
        result.pop();
    }
    cout << endl;

    return 0;
}
```

Step-by-Step Execution

Step 1: Push first k elements into a stack

Operation	Stack (Top to Bottom)	Queue
push 1	1	[2, 3, 4, 5]
push 2	2, 1	[3, 4, 5]
push 3	3, 2, 1	[4, 5]

Step 2: Pop from stack and enqueue back

Operation	Stack	Queue
pop 3	2, 1	[4, 5, 3]
pop 2	1	[4, 5, 3, 2]
pop 1	empty	[4, 5, 3, 2, 1]

Step 3: Rotate the remaining size - k elements (5 - 3 = 2 times)

Operation	Queue before	Queue after
move 4	[4, 5, 3, 2, 1]	[5, 3, 2, 1, 4]
move 5	[5, 3, 2, 1, 4]	[3, 2, 1, 4, 5]

❖ Final Queue:

[3, 2, 1, 4, 5]

▲ Output:

3 2 1 4 5

3 2 1 4 5

Sliding window maximum in C++

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

class SlidingWindowMaximum {
public:
    vector<int> maxSlidingWindow(vector<int>& nums, int k) {
        int n = nums.size();
        vector<int> ans;
        deque<int> deque;

        // Process the first window of size k separately
        for (int i = 0; i < k; i++) {
            while (!deque.empty() && nums[deque.back()] <= nums[i]) {
                deque.pop_back();
            }
            deque.push_back(i);
        }
        ans.push_back(nums[deque.front()]);

        // Process the rest of the elements
        for (int i = k; i < n; i++) {
            if (!deque.empty() && deque.front() == i - k) {
                deque.pop_front();
            }
            while (!deque.empty() && nums[deque.back()] <= nums[i]) {
                deque.pop_back();
            }
            deque.push_back(i);
            ans.push_back(nums[deque.front()]);
        }

        return ans;
    }

    int main() {
        SlidingWindowMaximum solution;

        // Example 1
        vector<int> nums1 = {1, 3, -1, -3, 5, 3, 6, 7};
        int k1 = 3;
        vector<int> result1 =
solution.maxSlidingWindow(nums1, k1);
        cout << "Max sliding window for nums1 and k=" <<
k1 << ":";
        for (int num : result1) {
            cout << num << " ";
        }
        cout << endl;

        return 0;
    }
}
```

Dry Run Table:

Index i	Element nums[i]	Deque (indices)	Deque (values)	Max in window
0	1	[0]	[1]	-
1	3	[1]	[3]	-
2	-1	[1, 2]	[3, -1]	3
3	-3	[1, 2, 3]	[3, -1, -3]	3
4	5	[4]	[5]	5
5	3	[4, 5]	[5, 3]	5
6	6	[6]	[6]	6
7	7	[7]	[7]	7

🧠 Explanation:

- The deque stores **indices** of elements in the current window.
- It's maintained in **decreasing order of values**.
- For each new element:
 - Remove indices from the back if their value is smaller than current.
 - Remove the front index if it's out of the window range.
 - Push the current index to the deque.
 - The front of the deque always has the index of the **max** of current window.

⚡ Final Output:

Max sliding window for nums1 and k=3: 3 3 5 5 6 7

Max sliding window for nums1 and k=3: 3 3 5 5 6 7

Reverse bits in C++							
Step-by-Step Dry Run (Tracking All Key Values):							
i	nums[i]	Deque (indices)	Deque (values)	Action	Window	Min	
0	1	[0]	[1]	Initial push	-	-	
1	3	[0, 1]	[1, 3]	$3 \geq 1$, keep 0, push 1	-	-	
2	-1	[2]	[-1]	Pop 1 and 0 (both > -1), push 2	[1, 3, -1]	-1	
3	-3	[3]	[-3]	Pop 2 ($\text{nums}[2] = -1 > -3$), push 3	[3, -1, -3]	-3	
4	5	[3, 4]	[-3, 5]	$5 > -3$, keep 3, push 4	[-1, -3, 5]	-3	
5	3	[3, 5]	[-3, 3]	Pop 4 ($5 > 3$), keep 3, push 5	[-3, 5, 3]	-3	
6	6	[5, 6]	[3, 6]	Pop 3 (index out of range), pop 3 ($\text{nums}[3] = 3$ is out), push 6	[5, 3, 6]	3	
7	7	[5, 6, 7]	[3, 6, 7]	$7 > 6$, keep 6, push 7	[3, 6, 7]	3	

↙ Final Output:

Minima for nums1 and k=3: -1 -3 -3 -3 3 3

Celebrity in C++

```
#include <iostream>
#include <stack>
using namespace std;

void findCelebrity(int arr[][4], int n) {
    stack<int> st;
    for (int i = 0; i < n; i++) {
        st.push(i);
    }

    while (st.size() > 1) {
        int i = st.top();
        st.pop();
        int j = st.top();
        st.pop();

        if (arr[i][j] == 1) {
            st.push(j);
        } else {
            st.push(i);
        }
    }

    int potential = st.top();
    bool isCelebrity = true;
    for (int i = 0; i < n; i++) {
        if (i != potential) {
            if (arr[i][potential] == 0 || arr[potential][i] == 1) {
                isCelebrity = false;
                break;
            }
        }
    }

    if (isCelebrity) {
        cout << potential << endl;
    } else {
        cout << "none" << endl;
    }
}

int main() {
    // Hardcoded input
    int n = 4;
    int arr[4][4] = {
        {0, 0, 0, 0},
        {1, 0, 1, 1},
        {1, 1, 0, 1},
        {1, 1, 1, 0}
    };

    // Finding the celebrity
    findCelebrity(arr, n);

    return 0;
}
```

Each cell $arr[i][j]$ tells us whether person i knows person j .

```
int arr[4][4] = {
    {0, 0, 0, 0}, // Person 0 knows nobody
    {1, 0, 1, 1}, // Person 1 knows 0, 2, 3
    {1, 1, 0, 1}, // Person 2 knows 0, 1, 3
    {1, 1, 1, 0} // Person 3 knows 0, 1, 2
};
```

💡 Stack-Based Elimination Table

Step	Stack Before	i (pop1)	j (pop2)	arr[i][j]	Action Taken	Stack After
1	[0, 1, 2, 3]	3	2	1	3 knows 2 → eliminate 3	[0, 1, 2]
2	[0, 1, 2]	2	1	1	2 knows 1 → eliminate 2	[0, 1]
3	[0, 1]	1	0	1	1 knows 0 → eliminate 1	[0]

Now stack.top() gives us **potential celebrity = 0**

🔍 Verification Table

Check if person 0 is a **celebrity**:

i	arr[i][0] (i knows 0)	arr[0][i] (0 knows i)	Condition Satisfied?
0	—	—	Skip self
1	1	0	✗ Person 1 knows 0, 0 knows no one
2	1	0	✗ Person 2 knows 0
3	1	0	✗ Person 3 knows 0

✓ All conditions met — 0 is a celebrity

✗ Final Output:

0

0

Merge overlapping Interval in C++

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

// Structure to represent a pair of start and end times
struct Pair {
    int st;
    int et;

    Pair(int s, int e) {
        st = s;
        et = e;
    }
};

// Comparator function to sort pairs based on start time
bool comparePairs(const Pair& a, const Pair& b) {
    return a.st < b.st;
}

// Function to merge overlapping intervals and print
// in increasing order of start time
void mergeOverlappingIntervals(vector<Pair>& intervals) {
    // Sort intervals based on start time
    sort(intervals.begin(), intervals.end(),
        comparePairs);

    stack<Pair> st;
    st.push(intervals[0]);

    for (int i = 1; i < intervals.size(); i++) {
        Pair top = st.top();

        // If current interval overlaps with the top of the
        // stack, merge them
        if (intervals[i].st <= top.et) {
            top.et = max(top.et, intervals[i].et);
            st.pop();
            st.push(top);
        } else {
            st.push(intervals[i]);
        }
    }

    // Output the merged intervals in sorted order
    stack<Pair> result;
    while (!st.empty()) {
        result.push(st.top());
        st.pop();
    }

    while (!result.empty()) {
        Pair p = result.top();
        cout << p.st << " " << p.et << endl;
        result.pop();
    }
}
```

Input Intervals (Unsorted)

```
{22, 28}
{1, 8}
{25, 27}
{14, 19}
{27, 30}
{5, 12}
```

Step 1: Sort Intervals by Start Time

After sorting using comparePairs, the list becomes:

Index	Start	End
0	1	8
1	5	12
2	14	19
3	22	28
4	25	27
5	27	30

Step 2: Merge Overlapping Intervals using Stack

i	Current Interval	Top of Stack	Action	Stack Content
0	{1, 8}	-	Push first interval	[{1, 8}]
1	{5, 12}	{1, 8}	Overlaps, merge to {1, 12}	[{1, 12}]
2	{14, 19}	{1, 12}	No overlap, push	[{1, 12}, {14, 19}]
3	{22, 28}	{14, 19}	No overlap, push	[{1, 12}, {14, 19}, {22, 28}]
4	{25, 27}	{22, 28}	Overlaps, merge to {22, 28} (no change)	[{1, 12}, {14, 19}, {22, 28}]
5	{27, 30}	{22, 28}	Overlaps, merge to {22, 30}	[{1, 12}, {14, 19}, {22, 30}]

```

int main() {
    // Hardcoded input
    vector<Pair> intervals = {
        {22, 28},
        {1, 8},
        {25, 27},
        {14, 19},
        {27, 30},
        {5, 12}
    };

    // Calling the function to merge overlapping
    // intervals
    mergeOverlappingIntervals(intervals);

    return 0;
}

```

1 12
14 19
22 30

📋 Final Stack (top to bottom):

{22, 30}
{14, 19}
{1, 12}

➡ Step 3: Print Intervals in Sorted Order

We reverse the stack to maintain start-time order:

1 12
14 19
22 30

❖ Output:

1 12
14 19
22 30

Sliding Window max in C++

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

vector<int> slidingWindowMaximum(vector<int>& arr, int k) {
    int n = arr.size();
    vector<int> result;
    stack<int> st;
    vector<int> nge(n);

    st.push(n-1);
    nge[n-1] = n;

    for (int i = n-2; i >= 0; i--) {
        while (!st.empty() && arr[i] >= arr[st.top()]) {
            st.pop();
        }

        if (st.empty()) {
            nge[i] = n;
        } else {
            nge[i] = st.top();
        }

        st.push(i);
    }

    for (int i = 0; i <= n-k; i++) {
        int j = i;
        while (nge[j] < i+k) {
            j = nge[j];
        }

        result.push_back(arr[j]);
    }

    return result;
}

int main() {
    // Hardcoded input
    vector<int> arr = {1, 3, -1, -3, 5, 3, 6, 7};
    int k = 3;

    vector<int> result = slidingWindowMaximum(arr, k);

    // Output the result
    for (int num : result) {
        cout << num << " ";
    }
    cout << endl;

    return 0;
}
```

Input:
 $arr = \{1, 3, -1, -3, 5, 3, 6, 7\}$
 $k = 3$
 $n = 8$

Step 1: Compute Next Greater Element Index Array (nge[])

We initialize an array nge[n], where:

- $nge[i]$ = index of the next greater element to the right of $arr[i]$
- If no such index, set $nge[i] = n$

■ NGE Construction Table

We build from **right to left** using a stack:

i	arr[i]	Stack (Top to Bottom)	nge[i]
7	7	[7]	8
6	6	[7, 6]	7
5	3	[7, 6, 5]	6
4	5	[7, 6, 4]	6
3	-3	[7, 6, 4, 3]	4
2	-1	[7, 6, 4, 2]	4
1	3	[7, 6, 4, 1]	4
0	1	[7, 6, 4, 1, 0]	1

→ Final nge[] = {1, 4, 4, 4, 6, 6, 7, 8}

Step 2: Compute Max in Each Sliding Window

For each window starting at i , you walk forward using nge[] until $nge[j] \geq i + k$.

■ Sliding Window Loop ($i = 0$ to $n - k$)

i	Window	j Traversal (via NGE)	Max Value
0	[1 3 -1]	0 → 1	3
1	[3 -1 -3]	1 → 4 (exits, $4 \geq 4$)	3
2	[-1 -3 5]	2 → 4	5
3	[-3 5 3]	3 → 4	5
4	[5 3 6]	4 → 6	6
5	[3 6 7]	5 → 6 → 7	7

	↙ Output: 3 3 5 5 6 7
3 3 5 5 6 7	

Two Stacks in C++

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

class TwoStack {
private:
    vector<int> data;
    int tos1; // Top of stack 1
    int tos2; // Top of stack 2
public:
    TwoStack(int cap) {
        // Constructor to initialize the two stacks
        data.resize(cap); // Resize the vector to given capacity
        tos1 = -1; // Initialize top of stack 1 to -1
        tos2 = cap; // Initialize top of stack 2 to cap (end of array)
    }

    int size1() {
        // Returns the size of stack 1
        return tos1 + 1;
    }

    int size2() {
        // Returns the size of stack 2
        return data.size() - tos2;
    }

    void push1(int val) {
        // Pushes an element onto stack 1
        if (tos2 == tos1 + 1) {
            cout << "Stack overflow\n";
        } else {
            tos1++;
            data[tos1] = val;
        }
    }

    void push2(int val) {
        // Pushes an element onto stack 2
        if (tos2 == tos1 + 1) {
            cout << "Stack overflow\n";
        } else {
            tos2--;
            data[tos2] = val;
        }
    }

    int pop1() {
        // Pops an element from stack 1
        if (size1() == 0) {
            cout << "Stack underflow\n";
            return -1;
        } else {
            int val = data[tos1];
            tos1--;
            return val;
        }
    }
}
```

Let's break down a **tabular dry run** of your TwoStack implementation with:

Capacity = 5

Operations = push1(10), push1(20), push2(30), push2(40), push2(50), push1(60)

Initial State

- data: [_, _, _, _, _]
- tos1 = -1, tos2 = 5
- Stack 1 grows → from index 0
- Stack 2 grows ← from index 4

Operation by Operation

Operation	tos1	tos2	data (0 to 4)	Note
push1(10)	0	5	[10, _, _, _, _]	10 pushed to stack 1
push1(20)	1	5	[10, 20, _, _, _]	20 pushed to stack 1
push2(30)	1	4	[10, 20, _, _, 30]	30 pushed to stack 2
push2(40)	1	3	[10, 20, _, 40, 30]	40 pushed to stack 2
push2(50)	1	2	[10, 20, 50, 40, 30]	50 pushed to stack 2
push1(60)	-	-	Overflow!	tos2 == tos1 + 1 (2 == 2)

Stack Status

- Stack1 (left): [10, 20] → tos1 = 1
- Stack2 (right): [50, 40, 30] → tos2 = 2

Output Operations

```
cout << "top1: " << st.top1() << "\n"; // 20
cout << "pop1: " << st.pop1() << "\n"; // 20
cout << "top1: " << st.top1() << "\n"; // 10
cout << "pop1: " << st.pop1() << "\n"; // 10
cout << "top2: " << st.top2() << "\n"; // 50
cout << "pop2: " << st.pop2() << "\n"; // 50
cout << "top2: " << st.top2() << "\n"; // 40
cout << "pop2: " << st.pop2() << "\n"; // 40
cout << "top1: " << st.top1() << "\n"; // Underflow (-1)
cout << "pop1: " << st.pop1() << "\n"; // Underflow (-1)
cout << "top2: " << st.top2() << "\n"; // 30
cout << "pop2: " << st.pop2() << "\n"; // 30
```

```

int pop2() {
    // Pops an element from stack 2
    if (size2 == 0) {
        cout << "Stack underflow\n";
        return -1;
    } else {
        int val = data[tos2];
        tos2++;
        return val;
    }
}

int top1() {
    // Returns the top element of stack 1
    if (size1 == 0) {
        cout << "Stack underflow\n";
        return -1;
    } else {
        return data[tos1];
    }
}

int top2() {
    // Returns the top element of stack 2
    if (size2 == 0) {
        cout << "Stack underflow\n";
        return -1;
    } else {
        return data[tos2];
    }
};

int main() {
    // Hardcoded example
    int capacity = 5;
    TwoStack st(capacity);

    // Perform operations
    st.push1(10);
    st.push1(20);
    st.push2(30);
    st.push2(40);
    st.push2(50);
    st.push1(60);

    cout << "top1: " << st.top1() << "\n";
    cout << "pop1: " << st.pop1() << "\n";
    cout << "top1: " << st.top1() << "\n";
    cout << "pop1: " << st.pop1() << "\n";
    cout << "top2: " << st.top2() << "\n";
    cout << "pop2: " << st.pop2() << "\n";
    cout << "top2: " << st.top2() << "\n";
    cout << "pop2: " << st.pop2() << "\n";
    cout << "top1: " << st.top1() << "\n";
    cout << "pop1: " << st.pop1() << "\n";
    cout << "top2: " << st.top2() << "\n";
    cout << "pop2: " << st.pop2() << "\n";

    return 0;
}

```

✓ Final Stack States

- Stack1: empty
- Stack2: empty
- tos1 = -1, tos2 = 5

Stack overflow

```
top1: 20
pop1: 20
top1: 10
pop1: 10
top2: 50
pop2: 50
top2: 40
pop2: 40
Stack underflow
top1: -1
Stack underflow
pop1: -1
top2: 30
pop2: 30
```

Check Max Heap in C++

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

class Solution {
public:
    static bool checkMaxHeap(vector<int>& arr) {
        for (int i = 0; i < arr.size(); i++) {
            int pIndex = i;
            int lIndex = 2 * i + 1;
            int rIndex = 2 * i + 2;

            if (lIndex < arr.size() && arr[pIndex] < arr[lIndex]) {
                return false;
            }

            if (rIndex < arr.size() && arr[pIndex] < arr[rIndex]) {
                return false;
            }
        }
        return true;
    }

    int main() {
        // Example input
        vector<int> arr = {42, 20, 18, 6, 14, 11, 9, 4};

        // Call the static method checkMaxHeap from
        // Solution class
        bool result = Solution::checkMaxHeap(arr);

        // Print the result
        cout << boolalpha << result << endl;

        return 0;
    }
}
```

true

Dry Run for Input: {42, 20, 18, 6, 14, 11, 9, 4}

Index (i)	Parent (arr[i])	Left Child Index (2i+1)	Left Value	Right Child Index (2i+2)	Right Value	Valid?
0	42	1	20	2	18	✓
1	20	3	6	4	14	✓
2	18	5	11	6	9	✓
3	6	7	4	8 (invalid)	—	✓
4 to 7	Leaf nodes	No children	—	—	—	✓

All parent nodes are greater than their children → ✓
Valid Max Heap

▣ Output:

true

SortKSortedArray in C++

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

class KthLargest {
public:
    static int kthLargest(int n, vector<int>& input, int k) {
        // Use a priority queue (max heap) to find the kth
        // largest element
        priority_queue<int> pq;

        // Insert all elements into the max heap
        for (int i = 0; i < n; i++) {
            pq.push(input[i]);
        }

        // Remove the top k-1 elements to get the kth
        // largest element
        for (int j = 0; j < k - 1; j++) {
            pq.pop();
        }

        // Return the kth largest element
        return pq.top();
    }
};

int main() {
    // Example input
    vector<int> arr = {2, 4, 1, 9, 6, 8};
    int k = 3;

    // Call the static method kthLargest from
    // KthLargest class
    int result = KthLargest::kthLargest(arr.size(), arr,
                                         k);

    // Print the result
    cout << "Kth largest element: " << result << endl;

    return 0;
}
```

Input:

arr = {2, 4, 1, 9, 6, 8}
k = 3

Dry Run Table:

Step	Action	Heap (Max-Heap structure)	Top Element
Init	Empty		
Insert 2	pq.push(2)	[2]	2
Insert 4	pq.push(4)	[4, 2]	4
Insert 1	pq.push(1)	[4, 2, 1]	4
Insert 9	pq.push(9)	[9, 4, 1, 2]	9
Insert 6	pq.push(6)	[9, 6, 1, 2, 4]	9
Insert 8	pq.push(8)	[9, 6, 8, 2, 4, 1]	9
Pop #1	pq.pop()	[8, 6, 1, 2, 4]	8
Pop #2	pq.pop()	[6, 4, 1, 2]	6

→ Final result = 6 (3rd largest)

Output:

Kth largest element: 6

SortKSortedArray in C++

```
#include <iostream>
#include <queue>
using namespace std;

void sort(int arr[], int n, int k) {
    // Create a min-heap (priority_queue) to store the
    // first k+1 elements
    priority_queue<int, vector<int>, greater<int>> pq;

    // Insert the first k+1 elements into the min-heap
    for (int i = 0; i <= k && i < n; i++) {
        pq.push(arr[i]);
    }

    // Process the remaining elements
    int index = 0;
    for (int i = k + 1; i < n; i++) {
        // Pop the smallest element from the min-heap
        // and store it in arr
        arr[index++] = pq.top();
        pq.pop();

        // Push the current element into the min-heap
        pq.push(arr[i]);
    }

    // Pop and store the remaining elements from the
    // min-heap
    while (!pq.empty()) {
        arr[index++] = pq.top();
        pq.pop();
    }
}

int main() {
    int arr[] = {2, 4, 1, 9, 6, 8};
    int k = 3;
    int n = sizeof(arr) / sizeof(arr[0]);

    sort(arr, n, k);

    // Print sorted array
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;

    return 0;
}
```

1 2 4 6 8 9

Input:

- arr[] = {2, 4, 1, 9, 6, 8}
- k = 3
- n = 6

🧠 Understanding the Flow:

1. Initialize a **min-heap** (using priority_queue with greater<int>).
2. Push the first $k + 1 = 4$ elements into the heap: [2, 4, 1, 9]
3. Pop the smallest from the heap and replace in arr (heapify and continue).
4. Keep pushing the next element and popping from the heap until all elements are processed.
5. At the end, empty the remaining heap into the array.

💡 Dry Run Table:

Step	Min-Heap (Top = Min)	Array Update (arr[])
Init	[1, 2, 4, 9]	—
Pop	1 → arr[0] = 1	[1, _, _, _, _, _]
Push 6 → Heap =	—	
[2, 6, 4, 9]	—	
Pop	2 → arr[1] = 2	[1, 2, _, _, _, _]
Push 8 → Heap =	—	
[4, 6, 9, 8]	—	
Pop	4 → arr[2] = 4	[1, 2, 4, _, _, _]
No more to push	—	
Pop	6 → arr[3] = 6	[1, 2, 4, 6, _, _]
Pop	8 → arr[4] = 8	[1, 2, 4, 6, 8, _]
Pop	9 → arr[5] = 9	[1, 2, 4, 6, 8, 9]

❖ Final Output:

1 2 4 6 8 9

Sort 012 in C++

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

class Sort012 {
public:
    void sort012(vector<int>& arr) {
        int i = 0, j = 0, k = arr.size() - 1;
        while (j <= k) {
            if (arr[j] == 0) {
                swap(arr[i], arr[j]);
                i++;
                j++;
            } else if (arr[j] == 1) {
                j++;
            } else {
                swap(arr[j], arr[k]);
                k--;
            }
        }
    }

    void swap(int& a, int& b) {
        int temp = a;
        a = b;
        b = temp;
    }
};

int main() {
    // Hardcoded input vector
    vector<int> arr = {0, 1, 2, 0, 1, 2, 1, 0, 2, 1};

    // Print the original array
    cout << "Original array: ";
    for (int num : arr) {
        cout << num << " ";
    }
    cout << endl;

    // Create an instance of Sort012 class
    Sort012 solution;

    // Call sort012 to sort the array
    solution.sort012(arr);

    // Print the sorted array
    cout << "Sorted array: ";
    for (int num : arr) {
        cout << num << " ";
    }
    cout << endl;

    return 0;
}
```

Original array: 0 1 2 0 1 2 1 0 2 1

Sorted array: 0 0 0 1 1 1 1 2 2 2

Input Array:

{0, 1, 2, 0, 1, 2, 1, 0, 2, 1}

🧠 Three-pointer strategy:

- i: points to the position where the next 0 should go.
- j: current index being processed.
- k: points to the position where the next 2 should go.

█████ Dry Run Table:

Step	i	j	k	arr[j]	Action	Array State
1	0	0	9	0	swap(i,j), ++i, ++j	0 1 2 0 1 2 1 0 2 1
2	1	1	9	1	j++	0 1 2 0 1 2 1 0 2 1
3	1	2	9	2	swap(j,k), k--	0 1 1 0 1 2 1 0 2 2
4	1	2	8	1	j++	0 1 1 0 1 2 1 0 2 2
5	1	3	8	0	swap(i,j), ++i, ++j	0 0 1 1 1 2 1 0 2 2
6	2	4	8	1	j++	0 0 1 1 1 2 1 0 2 2
7	2	5	8	2	swap(j,k), k--	0 0 1 1 1 2 1 0 2 2
8	2	5	7	2	swap(j,k), k--	0 0 1 1 1 0 1 2 2 2
9	2	5	6	0	swap(i,j), ++i, ++j	0 0 0 1 1 1 1 2 2 2
10	3	6	6	1	j++	0 0 0 1 1 1 1 2 2 2

❖ Final Output:

Sorted array: 0 0 0 1 1 1 1 2 2 2

Sort Colors in C++

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

class SortColors {
public:
    void sortColors(vector<int>& nums) {
        int n = nums.size();
        int i = 0, j = 0, k = n - 1;
        while (j <= k) {
            if (nums[j] == 0) {
                swap(nums[i], nums[j]);
                i++;
                j++;
            } else if (nums[j] == 1) {
                j++;
            } else {
                swap(nums[j], nums[k]);
                k--;
            }
        }
    }

    void swap(int& a, int& b) {
        int temp = a;
        a = b;
        b = temp;
    }
};

int main() {
    // Hardcoded input vector
    vector<int> arr = {0, 1, 2, 0, 1, 2, 1, 0, 2, 1};

    // Print the original array
    cout << "Original array: ";
    for (int num : arr) {
        cout << num << " ";
    }
    cout << endl;

    // Create an instance of SortColors class
    SortColors solution;

    // Call sortColors to sort the array
    solution.sortColors(arr);

    // Print the sorted array
    cout << "Sorted array: ";
    for (int num : arr) {
        cout << num << " ";
    }
    cout << endl;

    return 0;
}
```

Original array: 0 1 2 0 1 2 1 0 2 1

Sorted array: 0 0 0 1 1 1 2 2 2

Input

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

📌 Initial Setup

- $i = 0$ (position to place next 0)
- $j = 0$ (current index)
- $k = 9$ (position to place next 2)
- Size $n = 10$

🔍 Dry Run Table

Step	i	j	k	nums[j]	Action	Resulting Array
1	0	0	9	0	swap(i,j), ++i, ++j	0 1 2 0 1 2 1 0 2 1
2	1	1	9	1	++j	0 1 2 0 1 2 1 0 2 1
3	1	2	9	2	swap(j,k), --k	0 1 1 0 1 2 1 0 2 2
4	1	2	8	1	++j	0 1 1 0 1 2 1 0 2 2
5	1	3	8	0	swap(i,j), ++i, ++j	0 0 1 1 1 2 1 0 2 2
6	2	4	8	1	++j	0 0 1 1 1 2 1 0 2 2
7	2	5	8	2	swap(j,k), --k	0 0 1 1 1 2 1 0 2 2
8	2	5	7	2	swap(j,k), --k	0 0 1 1 1 0 1 2 2 2
9	2	5	6	0	swap(i,j), ++i, ++j	0 0 0 1 1 1 1 2 2 2
10	3	6	6	1	++j	0 0 0 1 1 1 1 2 2 2

🏁 Final Sorted Output

Sorted array: 0 0 0 1 1 1 2 2 2

All elements of Set in C++

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

void set(int x) {
    for (int i = 0; i < 32; ++i) {
        if (x & (1 << i)) {
            cout << i << endl;
        }
    }
}

int main() {
    int x = 282; // Binary representation:
100011010
    // int x = 7; // Binary representation:
111

    set(x);

    return 0;
}
```

Binary of 282

Decimal: 282

Binary : 00000000 00000000 00000001 00011010

Bits : ↑ ↑ ↑ ↑

Positions: 8 5 3 1 ← these are the set bits

Loop Table

i (bit position)	1 << i (binary)	x & (1 << i)	Is Bit Set?	Output
0	000...0001	0	✗	-
1	000...0010	2	✓	1
2	000...0100	0	✗	-
3	000...1000	8	✓	3
4	000..1_0000	0	✗	-
5	000.10_0000	32	✓	5
6	000.100_0000	0	✗	-
7	001.000_0000	0	✗	-
8	010.000_0000	256	✓	8
...	...	0	✗	-
31	100...0000 (bit 31)	0	✗	-

Final Output

1
3
5
8

Output:-

1
3
4
8

Bit check in C++

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

void bitChecker(int x, int k) {
    if ((x & (1 << k)) != 0) {
        cout << k << "th bit is 1" << endl;
    } else {
        cout << k << "th bit is 0" << endl;
    }
}

int main() {
    int x = 22; // Binary: 10110
    for (int k = 0; k <= 4; ++k) {
        bitChecker(x, k);
    }

    return 0;
}
```

Given:

- $x = 22 \rightarrow \text{binary} = 10110$
- We are checking each bit from position 0 to 4

Dry Run Table:

k (Bit Position)	1 << k (Mask)	x & (1 << k)	Is Bit Set?	Output
0	00001 (1)	$10110 \& 00001 = 00000$	No	0th bit is 0
1	00010 (2)	$10110 \& 00010 = 00010$	Yes	1th bit is 1
2	00100 (4)	$10110 \& 00100 = 00100$	Yes	2th bit is 1
3	01000 (8)	$10110 \& 01000 = 00000$	No	3th bit is 0
4	10000 (16)	$10110 \& 10000 = 10000$	Yes	4th bit is 1

❖ Output:

0th bit is 0
 1th bit is 1
 2th bit is 1
 3th bit is 0
 4th bit is 1

0th bit is 0
 1th bit is 1
 2th bit is 1
 3th bit is 0
 4th bit is 1

Div by 2^k in C++

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

int main() {
    int x = 24;
    int k = 3;
    int res = x >> k; // Right shift
    operation to divide x by 2^k
    cout << res << endl;

    return 0;
}
```

Given:

- $x = 24$
- $k = 3$
- Operation: $x >> k$ means shift the bits of x to the right by k positions (i.e., divide x by $2^k=8$). $2^3=8$.

Binary Representation

Variable	Binary	Decimal
x	0001 1000	24

Now right shift by 3 positions:

- Original: 0001 1000
- After $>> 1$: 0000 1100 (12)
- After $>> 2$: 0000 0110 (6)
- After $>> 3$: 0000 0011 (3)

Final Result:

cout << res << endl; // prints: 3

So the output is:

3

Output:-
3

Even Odd in C++

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

void fun(int x) {
    if ((x & 1) == 0) {
        cout << "even" << endl;
    } else {
        cout << "odd" << endl;
    }
}

int main() {
    int x = 27;
    fun(x);

    return 0;
}
```

Input:

- $x = 27$
- Binary of 27 = 11011

💡 Logic:

if ($(x \& 1) == 0$)

- $x \& 1$ checks the least significant bit (LSB)
- If the LSB is 1 → odd
- If the LSB is 0 → even

💻 Dry Run:

Expression	Value	Explanation
x	27	Decimal input
x (binary)	11011	Binary representation of 27
x & 1	11011 & 00001 = 00001	LSB is 1 → odd
$\ == 0$	false	So it goes to the else block
Output	odd	✓

✓ Final Output:

odd

odd

Power of 2 in C++																																																																																																										
#include <iostream> using namespace std;	Key Logic:																																																																																																									
void powerOf2(int x) { if ((x & (x - 1)) == 0) { cout << x << " is Power of two" << endl; } else { cout << x << " is not Power of two" << endl; } }	if ((x & (x - 1)) == 0) This works because: <ul style="list-style-type: none">• A power of two has only one set bit in binary.• $x \& (x - 1)$ turns off the lowest set bit, so:<ul style="list-style-type: none">◦ If result is 0 → x was a power of 2.◦ Otherwise → it's not.																																																																																																									
int main() { int x = 9; for (int i = 1; i <= 32; i++) { powerOf2(i); } return 0; }	Dry Run Table (for x from 1 to 16 for brevity):																																																																																																									
<table border="1"> <thead> <tr> <th>x</th><th>Binary of x</th><th>x-1</th><th>Binary of x-1</th><th>x & (x-1)</th><th>Is Power of 2?</th></tr> </thead> <tbody> <tr><td>1</td><td>00000001</td><td>0</td><td>00000000</td><td>00000000</td><td>✓ Yes</td></tr> <tr><td>2</td><td>00000010</td><td>1</td><td>00000001</td><td>00000000</td><td>✓ Yes</td></tr> <tr><td>3</td><td>00000011</td><td>2</td><td>00000010</td><td>00000010</td><td>✗ No</td></tr> <tr><td>4</td><td>00000100</td><td>3</td><td>00000011</td><td>00000000</td><td>✓ Yes</td></tr> <tr><td>5</td><td>00000101</td><td>4</td><td>00000100</td><td>00000100</td><td>✗ No</td></tr> <tr><td>6</td><td>00000110</td><td>5</td><td>00000101</td><td>00000100</td><td>✗ No</td></tr> <tr><td>7</td><td>00000111</td><td>6</td><td>00000110</td><td>00000110</td><td>✗ No</td></tr> <tr><td>8</td><td>00001000</td><td>7</td><td>00000111</td><td>00000000</td><td>✓ Yes</td></tr> <tr><td>9</td><td>00001001</td><td>8</td><td>00001000</td><td>00001000</td><td>✗ No</td></tr> <tr><td>10</td><td>00001010</td><td>9</td><td>00001001</td><td>00001000</td><td>✗ No</td></tr> <tr><td>11</td><td>00001011</td><td>10</td><td>00001010</td><td>00001010</td><td>✗ No</td></tr> <tr><td>12</td><td>00001100</td><td>11</td><td>00001011</td><td>00001000</td><td>✗ No</td></tr> <tr><td>13</td><td>00001101</td><td>12</td><td>00001100</td><td>00001100</td><td>✗ No</td></tr> <tr><td>14</td><td>00001110</td><td>13</td><td>00001101</td><td>00001100</td><td>✗ No</td></tr> <tr><td>15</td><td>00001111</td><td>14</td><td>00001110</td><td>00001110</td><td>✗ No</td></tr> <tr><td>16</td><td>00010000</td><td>15</td><td>00001111</td><td>00000000</td><td>✓ Yes</td></tr> </tbody> </table>					x	Binary of x	x-1	Binary of x-1	x & (x-1)	Is Power of 2?	1	00000001	0	00000000	00000000	✓ Yes	2	00000010	1	00000001	00000000	✓ Yes	3	00000011	2	00000010	00000010	✗ No	4	00000100	3	00000011	00000000	✓ Yes	5	00000101	4	00000100	00000100	✗ No	6	00000110	5	00000101	00000100	✗ No	7	00000111	6	00000110	00000110	✗ No	8	00001000	7	00000111	00000000	✓ Yes	9	00001001	8	00001000	00001000	✗ No	10	00001010	9	00001001	00001000	✗ No	11	00001011	10	00001010	00001010	✗ No	12	00001100	11	00001011	00001000	✗ No	13	00001101	12	00001100	00001100	✗ No	14	00001110	13	00001101	00001100	✗ No	15	00001111	14	00001110	00001110	✗ No	16	00010000	15	00001111	00000000	✓ Yes
x	Binary of x	x-1	Binary of x-1	x & (x-1)	Is Power of 2?																																																																																																					
1	00000001	0	00000000	00000000	✓ Yes																																																																																																					
2	00000010	1	00000001	00000000	✓ Yes																																																																																																					
3	00000011	2	00000010	00000010	✗ No																																																																																																					
4	00000100	3	00000011	00000000	✓ Yes																																																																																																					
5	00000101	4	00000100	00000100	✗ No																																																																																																					
6	00000110	5	00000101	00000100	✗ No																																																																																																					
7	00000111	6	00000110	00000110	✗ No																																																																																																					
8	00001000	7	00000111	00000000	✓ Yes																																																																																																					
9	00001001	8	00001000	00001000	✗ No																																																																																																					
10	00001010	9	00001001	00001000	✗ No																																																																																																					
11	00001011	10	00001010	00001010	✗ No																																																																																																					
12	00001100	11	00001011	00001000	✗ No																																																																																																					
13	00001101	12	00001100	00001100	✗ No																																																																																																					
14	00001110	13	00001101	00001100	✗ No																																																																																																					
15	00001111	14	00001110	00001110	✗ No																																																																																																					
16	00010000	15	00001111	00000000	✓ Yes																																																																																																					
✓ Output for i = 1 to 32:																																																																																																										
The function will print: 1 is Power of two 2 is Power of two 3 is not Power of two 4 is Power of two 5 is not Power of two ... 32 is Power of two																																																																																																										
Output:- 1 is Power of two																																																																																																										

Output:-
1 is Power of two

2 is Power of two
3 is not Power of two
4 is Power of two
5 is not Power of two
6 is not Power of two
7 is not Power of two
8 is Power of two
9 is not Power of two
10 is not Power of two
11 is not Power of two
12 is not Power of two
13 is not Power of two
14 is not Power of two
15 is not Power of two
16 is Power of two
17 is not Power of two
18 is not Power of two
19 is not Power of two
20 is not Power of two
21 is not Power of two
22 is not Power of two
23 is not Power of two
24 is not Power of two
25 is not Power of two
26 is not Power of two
27 is not Power of two
28 is not Power of two
29 is not Power of two
30 is not Power of two
31 is not Power of two
32 is Power of two

Subsets in C++

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

int main() {
    int n = 4;
    for (int b = 0; b < (1 << n); b++) {
        cout << b << endl;
    }

    return 0;
}
```

You're generating all numbers from 0 to $2^n - 1$ using bit manipulation!

💡 Breakdown:

- $n = 4 \rightarrow$ total combinations = $2^4 = 16$
- $(1 << n)$ means 1 shifted left n times \rightarrow equals 2^n
- Loop runs from 0 to 15, printing each value

💻 Dry Run Table:

b	Binary of b
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

Output:-
0

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

All repeating except two in C++

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

void solution(vector<int>& arr) {
    int xxory = 0;

    for(int val : arr) {
        xxory = xxory ^ val;
    }

    int rsbm = xxory & -xxory;

    int x = 0;
    int y = 0;

    for(int val : arr) {
        if((val & rsbm) == 0) {
            x = x ^ val;
        } else {
            y = y ^ val;
        }
    }

    if(x < y) {
        cout << x << endl;
        cout << y << endl;
    } else {
        cout << y << endl;
        cout << x << endl;
    }
}

int main() {
    vector<int> arr = {2, 2, 3, 3, 6, 6, 9, 1};
    solution(arr);
    return 0;
}
```

Given:

$\text{arr} = \{2, 2, 3, 3, 6, 6, 9, 1\}$

Pairs: 2, 2, 3, 3, 6, 6

Unique: 9, 1 \leftarrow these are the ones we need to find.

Q Step-by-step Dry Run:

Step 1: Find $\text{xxory} = \text{XOR of all elements}$

Iteration	val	xxory (XOR so far)
init		0
1	2	$0 \wedge 2 = 2$
2	2	$2 \wedge 2 = 0$
3	3	$0 \wedge 3 = 3$
4	3	$3 \wedge 3 = 0$
5	6	$0 \wedge 6 = 6$
6	6	$6 \wedge 6 = 0$
7	9	$0 \wedge 9 = 9$
8	1	$9 \wedge 1 = 8$

So, $\text{xxory} = 8$ (binary: 1000)

Step 2: Find the rightmost set bit of xxory

$\text{rsbm} = \text{xxory} \& -\text{xxory} = 8 \& -8 = 8$

Rightmost set bit is in position 4 (binary 1000)

Step 3: Divide numbers into two groups based on that bit

Group 1: $(\text{val} \& \text{rsbm}) == 0$

Group 2: $(\text{val} \& \text{rsbm}) != 0$

val	Binary	& rsbm (1000)	Group	x or y result
2	0010	0000	x	$x = 0 \wedge 2 = 2$
2	0010	0000	x	$x = 2 \wedge 2 = 0$
3	0011	0000	x	$x = 0 \wedge 3 = 3$
3	0011	0000	x	$x = 3 \wedge 3 = 0$
6	0110	0000	x	$x = 0 \wedge 6 = 6$
6	0110	0000	x	$x = 6 \wedge 6 = 0$
9	1001	1000	y	$y = 0 \wedge 9 = 9$
1	0001	0000	x	$x = 0 \wedge 1 = 1$

So final values:

- $x = 1$
- $y = 9$

❖ **Final Output:**

```
cout << x << endl;  
cout << y << endl;
```

Since $1 < 9$, the output is:

1
9

Summary Table:

Element	Group	x / y update
2	x	$x \wedge= 2 \rightarrow 0$
3	x	$x \wedge= 3 \rightarrow 0$
6	x	$x \wedge= 6 \rightarrow 0$
1	x	$x \wedge= 1 \rightarrow 1$
9	y	$y \wedge= 9 \rightarrow 9$

1
9

Copy Set Bits in a range in C++

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

int copySetBitsInRange(int a, int b, int left, int right)
{
    int m = (1 << (right - left + 1)) - 1; // Creates a mask
    // of 1s of the required length
    m = (m << (left - 1)); // Shifts the mask to the
    // correct position

    m = (m & a); // Extracts the bits from 'a' that need
    // to be copied
    b = b | m; // Copies the extracted bits to 'b'

    return b; // Returns the result
}

int main() {
    int a = 5;
    int b = 3;
    int left = 1;
    int right = 1;

    b = copySetBitsInRange(a, b, left, right);
    cout << b << endl;

    return 0;
}
```

```
int a = 5; // binary: 0101
int b = 3; // binary: 0011
int left = 1;
int right = 1;
```

We want to copy **only bit 1** (LSB) from a to b.

Q Step-by-step Dry Run:

Step	Expression	Result (in binary)	Explanation
1	$(1 << (right - left + 1)) - 1$	$(1 << 1) - 1 = 1 \rightarrow 0001$	Create a mask of 1s of length right - left + 1.
2	$m = m << (left - 1) \rightarrow 1 << 0 = 1$	0001	Shift the mask to the correct bit position range (left to right).
3	$m = m \& a \rightarrow 0001 \& 0101 = 0001$	0001	Mask a to extract the set bits in that range.
4	$b = b$	$m \rightarrow 0011$	$0001 = 0011$
5	return b	3	Final result.

◀ Final Output:

```
cout << b << endl; // 3
```

So the output is:

3

❖ Summary Table

Variable	Value (decimal)	Binary
a	5	0101
b (before)	3	0011
Mask	1	0001
Masked a	1	0001
b (after)	3	0011

Nothing changed in b, because bit 1 was already set in both a and b.

Count Set bits in C++

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

int countSetBits(int num) {
    int count = 0;
    while (num != 0) {
        count += num & 1;
        num >>= 1;
    }
    return count;
}

int main() {
    int num = 11; // Binary: 1011
    int count = countSetBits(num);
    cout << "Number of set bits: " << count << endl;

    return 0;
}
```

Dry Run Table:

Iteration	num (binary)	num & 1	count	num >> 1 (after shift)
1	1011 (11)	1	1	0101 (5)
2	0101 (5)	1	2	0010 (2)
3	0010 (2)	0	2	0001 (1)
4	0001 (1)	1	3	0000 (0)

❖ Final Output:

Number of set bits: 3

3

Gray Code in C++

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

void backtrack(vector<int>& ans, int n, int& temp) {
    if (n == 0) {
        ans.push_back(temp);
        return;
    }

    backtrack(ans, n - 1, temp);

    temp = temp ^ (1 << (n - 1));
    backtrack(ans, n - 1, temp);
}

vector<int> grayCode(int n) {
    vector<int> ans;
    if (n == 0) {
        ans.push_back(0);
        return ans;
    }

    int temp = 0;
    backtrack(ans, n, temp);
    return ans;
}

int main() {
    vector<int> ans = grayCode(3);
    sort(ans.begin(), ans.end());

    for (int num : ans) {
        cout << num << " ";
    }
    cout << endl;

    return 0;
}
```

Gray Code Summary

- A Gray code of n bits is a sequence of 2^n integers where **each successive number differs by only one bit**.
- This implementation generates it recursively by flipping one bit at each step using XOR:

$$\text{temp} = \text{temp} ^ (1 << (n - 1))$$

Dry Run: grayCode(3)

We'll track:

Call Depth	n	temp (Decimal)	temp (Binary)	Action
0	3	0	000	call (3→2)
1	2	0	000	call (2→1)
2	1	0	000	call (1→0)
3	0	0	000	push 0
2	1	1	001	flip bit 0 → 1
3	0	1	001	push 1
1	2	3	011	flip bit 1 → 1
2	1	3	011	call (1→0)
3	0	3	011	push 3
2	1	2	010	flip bit 0 → 0
3	0	2	010	push 2
0	3	6	110	flip bit 2 → 1
1	2	6	110	call (2→1)
2	1	6	110	call (1→0)
3	0	6	110	push 6
2	1	7	111	flip bit 0 → 1
3	0	7	111	push 7
1	2	5	101	flip bit 1 → 0
2	1	5	101	call (1→0)

3	0	5	101	push 5
2	1	4	100	flip bit 0 → 0
3	0	4	100	push 4

↙ Generated Sequence (before sort):

{0, 1, 3, 2, 6, 7, 5, 4}

0 1 2 3 4 5 6 7

Hamming Distance in C++

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

int hammingDistance(int x, int y) {
    int xorResult = x ^ y;
    int count = 0;
    while (xorResult != 0) {
        count += xorResult & 1;
        xorResult >>= 1;
    }
    return count;
}

int main() {
    cout << hammingDistance(10, 12) << endl; // Output: 2
    return 0;
}
```

Input:

x = 10 (1010 in binary)
y = 12 (1100 in binary)

Step 1: XOR the inputs

$$\begin{array}{r} 1010 \text{ (10)} \\ \wedge 1100 \text{ (12)} \\ \hline 0110 \text{ (6)} \end{array}$$

So, xorResult = 6 → binary: 0110

 **Dry Run Table:**

Step	xorResult (bin)	xorResult (dec)	xorResult & 1	count	After >>= 1
1	0110	6	0	0	3 (0011)
2	0011	3	1	1	1 (0001)
3	0001	1	1	2	0 (0000)

 **Final Output:**

Hamming Distance = 2

Josephus in C++

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

int p(int n) {
    int i = 1;
    while (i * 2 <= n) {
        i = i * 2;
    }
    return i;
}

int s(int n) {
    int h = p(n);
    int l = n - h;
    return 2 * l + 1;
}

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

You have two functions:

1. $p(n)$ — Finds the **largest power of 2 less than or equal to n** .
2. $s(n)$ — Computes $2 * (n - p(n)) + 1$.

 Dry Run for $n = 5$

Step 1: $p(5)$

```
int i = 1;
while (i * 2 <= n) {
    i = i * 2;
}
```

i (before loop)	i * 2	i (after loop)
1	2	2
2	4	4
4	8 (>5)	loop exits

◇ $p(5) = 4$

Step 2: $s(5)$

```
int h = p(5); // h = 4
int l = 5 - 4 = 1;
return 2 * l + 1 = 2 * 1 + 1 = 3;
```

◇ Output: 3

Kernighan in C++

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

int main() {
    int n = 5;
    int c = 0;

    while (n != 0) {
        int rs = n & -n; // rightmost set bit
        n = n - rs; // clear the rightmost set bit
        c++; // increment count
    }

    cout << c << endl;
    return 0;
}
```

Binary of 5:

Decimal: 5
Binary: 101
Set bits: 2

Dry Run Table:

Iteration	n (decimal)	n (binary)	rs $=$ n $\&$ $-n$	rs (binary)	n $=$ n $-$ rs	Count c
1	5	101	1	001	4	1
2	4	100	4	100	0	2

Final Output:

2

2

Leading Zeroes in C++

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

int countLeadingZeros(int num) {
    int leadingZeros = 0;
    while ((num & (1 << 31)) == 0) {
        leadingZeros++;
        num <<= 1;
    }
    return leadingZeros;
}

int main() {
    int num = 7; // Binary: 00000111
    int leadingZeros = countLeadingZeros(num);
    cout << "Leading zeros: " << leadingZeros <<
    endl; // Output: 28

    return 0;
}
```

Objective:

Count **leading zeroes** in the 32-bit binary form of num.

💡 Step-by-step:

The number 7 in **binary (32-bit)** is:

00000000 00000000 00000000 00000111

That's **3 bits set** on the right side — so we expect **29 leading zeroes** before the first 1.

Let's walk through it more carefully.

⌚ First thing to note:

- $1 << 31$ results in a mask:

10000000 00000000 00000000 00000000

- So the code is checking:
“Is the **leftmost (31st)** bit in num set?”

💻 Loop Simulation:

Each time, we:

- Check MSB (bit 31)
- If zero, we increment leadingZeros
- Then do num $<<= 1$ (left shift)

Let's track just leadingZeros:

Iteration	num (binary)	MSB	leadingZeros
0	00000000		
	00000000		
	00000000	0	0
	00000111		
1	00000000		
	00000000		
	00000000	0	1
	00001110		
2	00000000		
	00000000		
	00000000	0	2

	00011100		
3	00000000 00000000 00000000 00111000	0	3
...
28	01000000 00000000 00000000 00000000	0	28
29	10000000 00000000 00000000 00000000	1	29 (exit loop)
Leading zeros: 29			

✓ So yes — the **loop runs 29 times**, because the **first 1 in the 32-bit form** appears at **bit position 2 (from right)**, i.e., bit index **29 (from left)**.

✓ Final Answer:
Leading zeros: 29

One repeating one missing in C++

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

void solution(vector<int>& arr) {
    int xor_val = 0;
    int n = arr.size();

    // XOR all elements in arr and numbers from 1 to n
    for (int i = 0; i < n; i++) {
        xor_val ^= arr[i];
        xor_val ^= (i + 1);
    }

    // Find the rightmost set bit
    int rsb = xor_val & -xor_val;

    int x = 0, y = 0;

    // Divide elements into two groups based on rsb
    for (int i = 0; i < n; i++) {
        if (arr[i] & rsb)
            x ^= arr[i];
        else
            y ^= arr[i];

        if ((i + 1) & rsb)
            x ^= (i + 1);
        else
            y ^= (i + 1);
    }

    // Check which one is repeating and which one is
    // missing
    for (int i = 0; i < n; i++) {
        if (arr[i] == x) {
            cout << "Missing Number -> " << y << endl;
            cout << "Repeating Number -> " << x << endl;
            break;
        } else if (arr[i] == y) {
            cout << "Missing Number -> " << x << endl;
            cout << "Repeating Number -> " << y << endl;
            break;
        }
    }
}

int main() {
    vector<int> arr = {1, 3, 4, 4, 5, 6, 7};
    solution(arr);
    return 0;
}
```

Input:

arr = {1, 3, 4, 4, 5, 6, 7}

- $n = 7 \rightarrow$ array should contain 1 to 7
- But here:
 - 4 is repeated
 - 2 is missing

Step 1: XOR all elements and numbers from 1 to n

i	arr[i]	i+1	xor_val (after arr[i])	xor_val (after i+1)
0	1	1	$0 \wedge 1 = 1$	$1 \wedge 1 = 0$
1	3	2	$0 \wedge 3 = 3$	$3 \wedge 2 = 1$
2	4	3	$1 \wedge 4 = 5$	$5 \wedge 3 = 6$
3	4	4	$6 \wedge 4 = 2$	$2 \wedge 4 = 6$
4	5	5	$6 \wedge 5 = 3$	$3 \wedge 5 = 6$
5	6	6	$6 \wedge 6 = 0$	$0 \wedge 6 = 6$
6	7	7	$6 \wedge 7 = 1$	$1 \wedge 7 = 6$

→ Final xor_val = 6

Which is missing ^ repeating = $2 \wedge 4 = 6$

Step 2: Find rightmost set bit in xor_val

$rsb = xor_val \& -xor_val = 6 \& -6 = 2$ (binary: 10)

So we now divide numbers into **two groups** based on this bit.

Step 3: XOR within two groups

Let's categorize by whether $(number \& rsb) == 0$ or $\neq 0$

For arr and 1 to n

Element	Binary	Group (rsb)
1	0001	y
2	0010	x
3	0011	x
4	0100	y
5	0101	y
6	0110	x
7	0111	x

Perform XOR within groups

- Group X (bit set): 2, 3, 6, 3, 6, 7 → $x = 2 \wedge 3$

- $$\begin{aligned} & ^6 \wedge 3 \wedge 6 \wedge 7 = 2 \\ \bullet \quad & \text{Group Y (bit not set): } 1, 4, 4, 1, 5, 7, 5 \rightarrow y \\ & = 1 \wedge 4 \wedge 4 \wedge 1 \wedge 5 \wedge 5 = 4 \end{aligned}$$

Step 4: Determine which is missing and which is repeating

Check if $x = 2$ is present in arr → ✗ Not found →
So $x = 2$ is **missing**
 $y = 4$ is found → ✓ → $y = 4$ is **repeating**

❖ **Final Output:**

Missing Number -> 2
Repeating Number -> 4

Missing Number -> 2
Repeating Number -> 4

PowerSet in C++

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

void generatePowerSet(char set[], int n) {
    for (int i = 0; i < (1 << n); i++) {
        cout << "{ ";
        for (int j = 0; j < n; j++) {
            if (i & (1 << j)) {
                cout << set[j] << " ";
            }
        }
        cout << "}" << endl;
    }
}

int main() {
    char set[] = {'a', 'b', 'c'};
    int n = sizeof(set) / sizeof(set[0]);
    generatePowerSet(set, n);

    return 0;
}
```

Dry Run Example:

Let's dry run this with the set {'a', 'b', 'c'}. The set has 3 elements, so n = 3 and the total number of subsets will be $2^3 = 8$.

i	i in binary	Subset representation (bits set)	Subset generated
0	000	None	{ }
1	001	3rd bit set (only c)	{ c }
2	010	2nd bit set (only b)	{ b }
3	011	2nd & 3rd bits set (b, c)	{ b c }
4	100	1st bit set (only a)	{ a }
5	101	1st & 3rd bits set (a, c)	{ a c }
6	110	1st & 2nd bits set (a, b)	{ a b }
7	111	All bits set (a, b, c)	{ a b c }

Output:

```
{}
{ c }
{ b }
{ b c }
{ a }
{ a c }
{ a b }
{ a b c }
```

```
{}
{ a }
{ b }
{ a b }
{ c }
{ a c }
{ b c }
{ a b c }
```

Reverse bits in C++

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

int reverseBits(int num) {
    int reversed = 0;
    for (int i = 0; i < 32; i++) {
        reversed = (reversed << 1) | ((num >> i) & 1);
    }
    return reversed;
}

int main() {
    int num = 25; // Binary: 00011001
    int reversed = reverseBits(num);

    cout << reversed << endl; // Output: 147

    return 0;
}
```

Input:
int num = 25;

Binary of 25 (8-bit view):

00000000 00000000 00000000 00011001

Which is:
 $(1*16) + (1*8) + (0*4) + (0*2) + (1*1) = 16 + 8 + 1 = 25$

☞ What reverseBits function does:

```
for (int i = 0; i < 32; i++) {
    reversed = (reversed << 1) | ((num >> i) & 1);
}
```

- It shifts out each bit of num **starting from LSB**, and inserts it into the MSB side of reversed.

Think of it like flipping a coin 32 times and stacking them from left to right.

■ Simulation (only the last 8 bits of reversed will matter):

Let's extract bits from **right to left** of num = 000...00011001:

i	(num >> i) & 1	reversed (binary)	reversed (decimal)
0	1	000...00000001	1
1	0	000...00000010	2
2	0	000...00000100	4
3	1	000...00001001	9
4	1	000...00010011	19
5	0	000...00100110	38
6	0	000...01001100	76
7	0	000...10011000	152
8-31	0s	right shift zeros only	final: 147

Wait — this gives 152, not 147? 😬

Let's **actually go full 32-bits** and see the result:

After full 32-bit reversal, the **reversed binary** is:

10011000 00000000 00000000 00000000

But we're interpreting the final reversed number as an int, which counts all 32 bits.

↙ Actual Output:
cout << reversed << endl; // prints 147

So there's a mismatch.

Let's double-check with code:

```
int num = 25;  
int reversed = reverseBits(num);  
cout << reversed << endl;
```

The reversed **8 bits** of 25 (00011001) is 10011000 = **152**

BUT due to **bit reversal of all 32 bits**, reversed becomes a large number. So to get **147** as final output:

Let's check the 32-bit reversal of:

```
00000000 00000000 00000000 00011001 // 25
```

Reversed:

```
10011000 00000000 00000000 00000000 // final
```

Now:

Binary: 10011000 00000000 00000000 00000000
Decimal: 1476395008

But that's not 147 either.

💡 The Fix:

You must print the reversed result in 8-bit sense, or mask it:

```
cout << (reversed >> 24) << endl;
```

This will give **actual 8-bit reversed form**, i.e.:

25 → 00011001
reversed → 10011000 → 152

But if your output is **147**, that means your original number is not 25, or the system is interpreting signed bits differently.

Set a bit in C++

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

int main() {
    int num = 5; // Binary: 0101
    int bitmask = 1 << 2; // Binary: 0100
    int result = num | bitmask; // Binary: 0101
    (Decimal: 5)

    cout << result << endl; // Output: 5

    return 0;
}
```

Dry Run in Table:

Step	Variable	Value	Explanation
1	num	5 (Binary: 0101)	Initialize num as 5.
2	bitmask	4 (Binary: 0100)	Compute bitmask = 1 << 2 (left shift 1 by 2 positions).
3	`num`	bitmask	5 (Binary: 0101)
4	result	5 (Binary: 0101)	Store the result of `num`
5	cout	5	Print the value of result (which is 5).

Detailed Explanation of Key Operations:

- Step 1: Initialize num**
num is set to 5. Its binary representation is 0101.
- Step 2: Create bitmask**
bitmask = 1 << 2 shifts the binary 0001 two positions to the left.
The result is 0100, which is 4 in decimal.
- Step 3: Perform Bitwise OR (|)**
result = num | bitmask →
0101 (num)
0100 (bitmask)
The result of 0101 | 0100 is 0101, which is 5 in decimal.
- Step 4: Store and Output result**
The result of the bitwise OR operation is 5.
The program prints the result:
Output: 5

Final Output:

5

Single Number in C++

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

int singleNumber(vector<int>& nums) {
    int result = 0;
    for (int num : nums) {
        result ^= num;
    }
    return result;
}

int main() {
    vector<int> arr = {2, 2, 3, 3, 4, 6, 6};
    cout << singleNumber(arr) << endl; // Output: 4

    return 0;
}
```

Input:

vector<int> arr = {2, 2, 3, 3, 4, 6, 6};

All numbers repeat twice **except 4**, which should be our result.

💡 Logic Behind XOR:

- $a \wedge a = 0$
- $a \wedge 0 = a$
- XOR is **commutative** and **associative**, so order doesn't matter.

💻 Dry Run Table:

Step	num	result (before)	result \wedge num	result (after)
1	2	0	$0 \wedge 2 = 2$	2
2	2	2	$2 \wedge 2 = 0$	0
3	3	0	$0 \wedge 3 = 3$	3
4	3	3	$3 \wedge 3 = 0$	0
5	4	0	$0 \wedge 4 = 4$	4
6	6	4	$4 \wedge 6 = 2$	2
7	6	2	$2 \wedge 6 = 4$	4

❖ Final Output:

4

Swap in C++

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

int main() {
    int a = 5;
    int b = 7;

    a = a ^ b;
    b = a ^ b;
    a = a ^ b;

    cout << a << endl; // Output: 7
    cout << b << endl; // Output: 5

    return 0;
}
```

Initial Values:

a = 5 (0101 in binary)
b = 7 (0111 in binary)

☒ XOR Swap Steps:

Step	Expression	Value (Binary)	Explanation
a = a ^ b	a = 5 ^ 7	0101 ^ 0111 = 0010 → a = 2	
b = a ^ b	b = 2 ^ 7	0010 ^ 0111 = 0101 → b = 5	
a = a ^ b	a = 2 ^ 5	0010 ^ 0101 = 0111 → a = 7	

❖ Final Values:

a = 7
b = 5

7
5

Intersection in C++

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

// Node class definition
class Node {
public:
    int data;
    Node* next;

    Node(int d) {
        data = d;
        next = nullptr;
    }
};

// Intersection2LL class definition
class Intersection2LL {
public:
    Node* head1;
    Node* head2;

    int getCount(Node* node) {
        Node* current = node;
        int count = 0;

        while (current != nullptr) {
            count++;
            current = current->next;
        }
        return count;
    }

    int getNode() {
        int c1 = getCount(head1);
        int c2 = getCount(head2);
        int d;
        if (c1 > c2) {
            d = c1 - c2;
            return getIntesectionNode(d, head1, head2);
        } else {
            d = c2 - c1;
            return getIntesectionNode(d, head2, head1);
        }
    }

    int getIntesectionNode(int d, Node* node1, Node* node2) {
        Node* current1 = node1;
        Node* current2 = node2;

        for (int i = 0; i < d; i++) {
            if (current1 == nullptr) {
                return -1;
            }
            current1 = current1->next;
        }

        while (current1 != nullptr && current2 != nullptr) {
            if (current1->data == current2->data) {
                return current1->data;
            }
        }
    }
};
```

Final Linked Lists

List 1	List 2
3 → 6 → 9 → 15 → 30	10 → 15 → 30

- Intersection starts at **node 15** (shared memory).

⌚ Dry Run of getNode()

1. Count Nodes

Operation	Result
Count of List 1	5
Count of List 2	3
d = c1 - c2	2

2. Advance Longer List by d = 2 Nodes

After Skipping in List 1	Current Node 1	Current Node 2
Skip 1st → 3	6	
Skip 2nd → 6	9	

Now:

- current1 = 9
- current2 = 10

⌚ Start Comparing Nodes

Step	current1->data	current2->data	Same Node Address?	Action
1	9	10	✗	Move both forward
2	15	15	✓✓✓	Return 15

❖ Output

The node of intersection is 15

📋 Summary Table

Phase	Details
Total Nodes in List1	5
Total Nodes in List2	3
Difference d	2
First match by addr	Node with data 15
Final Answer	15

```
    }
    current1 = current1->next;
    current2 = current2->next;
}

return -1;
};

int main() {
// Creating an instance of Intersection2LL
Intersection2LL list;

// Creating first linked list
list.head1 = new Node(3);
list.head1->next = new Node(6);
list.head1->next->next = new Node(9);
list.head1->next->next->next = new Node(15);
list.head1->next->next->next->next = new
Node(30);

// Creating second linked list
list.head2 = new Node(10);
list.head2->next = new Node(15);
list.head2->next->next = new Node(30);

// Finding the intersection node
cout << "The node of intersection is " <<
list.getNode() << endl;

// Clean up memory
delete list.head1->next->next->next->next;
delete list.head1->next->next->next;
delete list.head1->next->next;
delete list.head1->next;
delete list.head2->next->next;
delete list.head2->next;
delete list.head2;

return 0;
}
```

The node of intersection is 15

K Reverse in C++

```
#include <iostream>

using namespace std;

// Node class definition
class Node {
public:
    int data;
    Node* next;

    // Constructor
    Node(int d) {
        data = d;
        next = nullptr;
    }
};

// LinkedList class definition
class LinkedList {
private:
    Node* head;
    Node* tail;
    int size;

public:
    // Constructor
    LinkedList() {
        head = nullptr;
        tail = nullptr;
        size = 0;
    }

    // Method to add a node at the beginning of the list
    void addFirst(int val) {
        Node* temp = new Node(val);
        temp->next = head;
        head = temp;
        if (size == 0) {
            tail = temp;
        }
        size++;
    }

    // Method to add a node at the end of the list
    void addLast(int val) {
        Node* temp = new Node(val);
        if (size == 0) {
            head = tail = temp;
        } else {
            tail->next = temp;
            tail = temp;
        }
        size++;
    }

    // Method to display the elements of the list
    void display() {
        Node* temp = head;
        while (temp != nullptr) {
            cout << temp->data << " ";
            temp = temp->next;
        }
    }
}
```

Initial Input:

List:

1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10 → 11
k = 3

🔗 kReverse Logic Dry Run:

We reverse **groups of 3 elements**. Let's track the changes in a **table** as each k-group is processed:

Group #	Extracted Nodes	Reversed Order	prev List After Merge
1	1 2 3	3 2 1	3 → 2 → 1
2	4 5 6	6 5 4	3 → 2 → 1 → 6 → 5 → 4
3	7 8 9	9 8 7	3 → 2 → 1 → 6 → 5 → 4 → 9 → 8 → 7
4	10 11	(unchanged)	... → 9 → 8 → 7 → 10 → 11

🔗 After kReverse:

List:

3 → 2 → 1 → 6 → 5 → 4 → 9 → 8 → 7 → 10 → 11

```

    }
    cout << endl;
}

// Method to remove the first node from the list
void removeFirst() {
    if (size == 0) {
        cout << "List is empty" << endl;
    } else {
        Node* temp = head;
        head = head->next;
        delete temp;
        size--;
        if (size == 0) {
            tail = nullptr;
        }
    }
}

// Method to get the first element of the list
int getFirst() {
    if (size == 0) {
        cout << "List is empty" << endl;
        return -1;
    } else {
        return head->data;
    }
}

// Method to reverse every k nodes in the list
void kReverse(int k) {
    LinkedList prev;

    while (size > 0) {
        LinkedList curr;

        if (size >= k) {
            for (int i = 0; i < k; i++) {
                int val = getFirst();
                removeFirst();
                curr.addFirst(val);
            }
        } else {
            int sz = size;
            for (int i = 0; i < sz; i++) {
                int val = getFirst();
                removeFirst();
                curr.addLast(val);
            }
        }

        if (prev.size == 0) {
            prev = curr;
        } else {
            tail->next = curr.head;
            tail = curr.tail;
            size += curr.size;
        }
    }

    head = prev.head;
    tail = prev.tail;
}

```

```

        size = prev.size;
    }

// Destructor to free memory
~LinkedList() {
    Node* curr = head;
    while (curr != nullptr) {
        Node* temp = curr;
        curr = curr->next;
        delete temp;
    }
};

// Main function to demonstrate LinkedList operations
int main() {
    LinkedList l1;

    l1.addLast(1);
    l1.addLast(2);
    l1.addLast(3);
    l1.addLast(4);
    l1.addLast(5);
    l1.addLast(6);
    l1.addLast(7);
    l1.addLast(8);
    l1.addLast(9);
    l1.addLast(10);
    l1.addLast(11);

    int k = 3;
    int a = 100;
    int b = 200;

    l1.display();      // Original list: 1 2 3 4 5 6 7 8 9
10 11
    l1.kReverse(k);  // Reverse every k nodes
    l1.display();      // After kReverse: 3 2 1 6 5 4 9 8
7 10 11
    l1.addFirst(a);   // Add element at the beginning:
100 3 2 1 6 5 4 9 8 7 10 11
    l1.addLast(b);   // Add element at the end: 100 3
2 1 6 5 4 9 8 7 10 11 200
    l1.display();      // Final list

    return 0;
}

```

1 2 3 4 5 6 7 8 9 10 11

Linked List (Add at index) in C++

```
#include <iostream>

using namespace std;

// Node class definition
class Node {
public:
    int data;
    Node* next;

    // Constructor
    Node(int d) {
        data = d;
        next = nullptr;
    }
};

// LinkedList class definition
class LinkedList {
private:
    Node* head;
    Node* tail;
    int size;

public:
    // Constructor
    LinkedList() {
        head = nullptr;
        tail = nullptr;
        size = 0;
    }

    // Method to add a node at the end of the list
    void addLast(int val) {
        Node* temp = new Node(val);
        if (size == 0) {
            head = tail = temp;
        } else {
            tail->next = temp;
            tail = temp;
        }
        size++;
    }

    // Method to get the size of the list
    int getSize() {
        return size;
    }

    // Method to display the elements of the list
    void display() {
        Node* temp = head;
        while (temp != nullptr) {
            cout << temp->data << " ";
            temp = temp->next;
        }
        cout << endl;
    }

    // Method to remove the first node
    void removeFirst() {

```

Dry Run Table

Step	Operation	List State	Output	Notes
1	addFirst(10)	10		Adds 10 at front
2	getFirst()	10	10	
3	addAt(0, 20)	20 → 10		Insert 20 at index 0
4	getFirst()	20 → 10	20	
5	getLast()	20 → 10	10	
6	display()	20 → 10	20 10	
7	getSize()	20 → 10	2	
8	addAt(2, 40)	20 → 10 → 40		Insert 40 at end
9	getLast()	20 → 10 → 40	40	
10	addAt(1, 50)	20 → 50 → 10 → 40		Insert 50 at index 1
11	addFirst(30)	30 → 20 → 50 → 10 → 40		Adds 30 at front
12	removeFirst()	20 → 50 → 10 → 40		Removes 30
13	getFirst()	20 → 50 → 10 → 40	20	
14	removeFirst()	50 → 10 → 40		Removes 20
15	removeFirst()	10 → 40		Removes 50
16	addAt(2, 60)	10 → 40 → 60		Adds 60 at index 2
17	display()	10 → 40 → 60	10 40 60	
18	getSize()	10 → 40 → 60	3	
19	removeFirst()	40 → 60		Removes 10

```

if (size == 0) {
    cout << "List is empty" << endl;
} else if (size == 1) {
    head = tail = nullptr;
    size = 0;
} else {
    head = head->next;
    size--;
}
}

int getFirst() {
    if (size == 0) {
        cout << "List is empty" << endl;
        return -1;
    } else {
        return head->data;
    }
}

int getLast() {
    if (size == 0) {
        cout << "List is empty" << endl;
        return -1;
    } else {
        return tail->data;
    }
}

int getAt(int idx) {
    if (size == 0) {
        cout << "List is empty" << endl;
        return -1;
    } else if (idx < 0 || idx >= size) {
        cout << "Invalid arguments" << endl;
        return -1;
    } else {
        Node* temp = head;
        for (int i = 0; i < idx; i++) {
            temp = temp->next;
        }
        return temp->data;
    }
}

// Method to add a node at the beginning of the list
void addFirst(int val) {
    Node* temp = new Node(val);
    temp->next = head;
    head = temp;
    if (size == 0) {
        tail = temp;
    }
    size++;
}

// Method to add a node at a specified index
void addAt(int idx, int val) {
    if (idx < 0 || idx > size) {
        cout << "Invalid arguments" << endl;
    } else if (idx == 0) {
        addFirst(val);
    } else if (idx == size) {
        addLast(val);
    } else {
        Node* node = new Node(val);
    }
}

```

20	removeFirst()	60		Removes 40
21	getFirst()	60	60	

```

Node* temp = head;
for (int i = 0; i < idx - 1; i++) {
    temp = temp->next;
}

node->next = temp->next;
temp->next = node;

size++;
}
};

// Main function to demonstrate LinkedList operations
int main() {
    LinkedList list;

    // Hardcoded sequence of operations
    list.addFirst(10);
    cout << list.getFirst() << endl; // Should display: 10

    list.addAt(0, 20);
    cout << list.getFirst() << endl; // Should display: 20
    cout << list.getLast() << endl; // Should display: 10

    list.display(); // Should display: 20 10

    cout << list.getSize() << endl; // Should display: 2

    list.addAt(2, 40);
    cout << list.getLast() << endl; // Should display: 40

    list.addAt(1, 50);
    list.addFirst(30);
    list.removeFirst();
    cout << list.getFirst() << endl; // Should display: 20

    list.removeFirst();
    list.removeFirst();
    list.addAt(2, 60);
    list.display(); // Should display: 50 10 60

    cout << list.getSize() << endl; // Should display: 3

    list.removeFirst();
    list.removeFirst();
    cout << list.getFirst() << endl; // Should display: 60

    return 0;
}

```

```

10
20
10
20 10
2
40
20
10 40 60
3
60

```

Merge in C++

```
#include <iostream>

using namespace std;

// Node class definition
class Node {
public:
    int data;
    Node* next;

    // Constructor
    Node(int d) {
        data = d;
        next = nullptr;
    }
};

// LinkedList class definition
class LinkedList {
public:
    Node* head;
    Node* tail;
    int size;

    // Constructor
    LinkedList() {
        head = nullptr;
        tail = nullptr;
        size = 0;
    }

    // Method to add node at the end
    void addLast(int val) {
        Node* temp = new Node(val);
        if (size == 0) {
            head = tail = temp;
        } else {
            tail->next = temp;
            tail = temp;
        }
        size++;
    }

    // Method to print the linked list
    void display() {
        Node* temp = head;
        while (temp != nullptr) {
            cout << temp->data << " ";
            temp = temp->next;
        }
        cout << endl;
    }

    // Function to merge two sorted linked lists
    static Node* sortedMerge(Node* headA, Node* headB) {
        Node* dummyNode = new Node(0);
        Node* tail = dummyNode;

        while (true) {
            if (headA == nullptr) {
                tail->next = headB;
                break;
            }
            if (headB == nullptr) {
                tail->next = headA;
                break;
            }
            if (headA->data < headB->data) {
                tail->next = headA;
                headA = headA->next;
            } else {
                tail->next = headB;
                headB = headB->next;
            }
            tail = tail->next;
        }
        return dummyNode->next;
    }
};
```

What the Code Does

- Two sorted linked lists are created:
 - List 1: 5 -> 10 -> 15
 - List 2: 2 -> 3 -> 20
- The sortedMerge() function merges them into a single sorted list.
- Result is printed.

Initial Lists

List 1 (llist1) List 2 (llist2)

5 → 10 → 15 2 → 3 → 20

Dry Run of sortedMerge()

Step	headA->data	headB->data	Chosen Node	Merged List So Far
1	5	2	2 (from B)	2
2	5	3	3 (from B)	2 → 3
3	5	20	5 (from A)	2 → 3 → 5
4	10	20	10 (from A)	2 → 3 → 5 → 10
5	15	20	15 (from A)	2 → 3 → 5 → 10 → 15
6	null	20	Append B	2 → 3 → 5 → 10 → 15 → 20

Final Output

2 3 5 10 15 20

Summary

Input List 1	Input List 2	Output (Merged Sorted List)
5 → 10 → 15	2 → 3 → 20	2 → 3 → 5 → 10 → 15 → 20

```

        tail->next = headB;
        break;
    }
    if (headB == nullptr) {
        tail->next = headA;
        break;
    }
    if (headA->data <= headB->data) {
        tail->next = headA;
        headA = headA->next;
    } else {
        tail->next = headB;
        headB = headB->next;
    }
    tail = tail->next;
}

return dummyNode->next;
};

// Main function
int main() {
    LinkedList llist1;
    LinkedList llist2;

    // Adding elements to the first linked list
    llist1.addLast(5);
    llist1.addLast(10);
    llist1.addLast(15);

    // Adding elements to the second linked list
    llist2.addLast(2);
    llist2.addLast(3);
    llist2.addLast(20);

    // Merging the two sorted linked lists
    Node* mergedHead =
LinkedList::sortedMerge(llist1.head, llist2.head);

    // Printing the merged list
    Node* temp = mergedHead;
    while (temp != nullptr) {
        cout << temp->data << " ";
        temp = temp->next;
    }
    cout << endl;

    return 0;
}

```

2 3 5 10 15 20

Multiply LL in C++

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

// Node class for the
linked list
class Node {
public:
    int val;
    Node* next;

    Node(int val) {
        this->val = val;
        this->next = nullptr;
    }
};

Node* reverse(Node* head) {
    if (head == nullptr || head->next == nullptr)
        return head;

    Node* prev = nullptr;
    Node* curr = head;
    while (curr != nullptr) {
        Node* forw = curr->next;
        curr->next = prev;
        prev = curr;
        curr = forw;
    }

    return prev;
}

// Function to add two
linked lists in place
void
addTwoLinkedList(Node* head, Node* ansItr) {
    Node* c1 = head;
    Node* c2 = ansItr;

    int carry = 0;
    while (c1 != nullptr || carry != 0) {
        int sum = carry + (c1 != nullptr ? c1->val : 0) +
(c2->next != nullptr ? c2->next->val : 0);
        int digit = sum % 10;
        carry = sum / 10;

        if (c2->next != nullptr)
            c2->next->val = digit;
        else
            c2->next = new Node(digit);

        if (c1 != nullptr)
            c1 = c1->next;
        c2 = c2->next;
    }
}
```

Given:

- $l1 = 2 \rightarrow 4 \rightarrow 3$ (representing the number 342)
- $l2 = 5 \rightarrow 6 \rightarrow 4$ (representing the number 465)

We are multiplying these two numbers, and as part of the algorithm, we reverse both linked lists, perform multiplication on each digit, and handle carries. Then, we add the intermediate results, ensuring proper shifting of digits.

Dry Run Table:

Step	l1 (reversed)	l2 (reversed)	Current digit of l2 (l2_itr->val)	Multiplication Result (prod)	Shift Applied	Interim Result
Initial	3 -> 4 -> 2	4 -> 6 -> 5	N/A	N/A	N/A	N/A
Reversed	2 -> 4 -> 3	5 -> 6 -> 4	N/A	N/A	N/A	N/A
Multiply l1 by 5 (1st digit of l2)	2 -> 4 -> 3	5	$5 * 3 =$ $15, 5 * 4$ $= 20 + 1$ (carry) = $21, 5 * 2$ $= 10 + 2$ (carry) = 12	5 -> 1 -> 2	No Shift (first digit)	5 -> 1 -> 2
Add this result to the intermediate result (result = 5 -> 1 -> 2)	2 -> 4 -> 3	6 -> 5	N/A	N/A	N/A	5 -> 1 -> 2 (no change)
Multiply l1 by 6 (2nd digit of l2)	2 -> 4 -> 3	6	$6 * 3 =$ $18, 6 * 4$ $= 24 + 1$ (carry) = $25, 6 * 2$ $= 12 + 2$ (carry) = 14	8 -> 5 -> 4	Shift by 1	8 -> 5 -> 4 -> 0 -> 0
Add this result to the intermediate result (add 8 -> 5 -> 4 -> -> 0 to 5 -> 1 -> 2)	2 -> 4 -> 3	5	N/A	N/A	N/A	1 -> 5 -> 9 -> 0 -> 3 -> 0
Multiply l1 by 4 (3rd digit of l2)	2 -> 4 -> 3	4	$4 * 3 =$ $12, 4 * 4$ $= 16 + 1$ (carry) = $17, 4 * 2$ $= 8 + 1$ (carry) = 9	2 -> 7 -> 9	Shift by 2	2 -> 7 -> 9 -> 0 -> 0 -> 0
Add this result to the intermediate result (add 2	2 -> 4 -> 3	4	N/A	N/A	N/A	1 -> 5 -> 9 -> 0 -> 3 -> 0 (final)

<pre> } // Function to multiply a linked list with a single digit Node* multiplyLLWithDigit(Node* head, int dig) { Node* dummy = new Node(-1); Node* ac = dummy; Node* curr = head; int carry = 0; while (curr != nullptr carry != 0) { int sum = carry + (curr != nullptr ? curr- >val * dig : 0); int digit = sum % 10; carry = sum / 10; ac->next = new Node(digit); if (curr != nullptr) curr = curr->next; ac = ac->next; } return dummy->next; } // Function to multiply two linked lists representing numbers Node* multiplyTwoLL(Node* l1, Node* l2) { l1 = reverse(l1); l2 = reverse(l2); Node* l2_Itr = l2; Node* dummy = new Node(-1); Node* ansItr = dummy; while (l2_Itr != nullptr) { Node* prod = multiplyLLWithDigit(l1, l2_Itr->val); l2_Itr = l2_Itr->next; addTwoLinkedList(prod, ansItr); ansItr = ansItr- >next; } </pre>	<p>-> 7 -> 9 -> 0 -> 0 -> 0 to 1 -> 5 -> 9 -> 0 -> 3 -> 0)</p>	<p>Step-by-Step Process:</p> <ol style="list-style-type: none"> Reversing the Lists: <ul style="list-style-type: none"> 11 = 2 -> 4 -> 3 becomes 3 -> 4 -> 2. 12 = 5 -> 6 -> 4 becomes 4 -> 6 -> 5. Multiplying l1 by each digit of l2: <ul style="list-style-type: none"> First, multiply l1 by 5: <ul style="list-style-type: none"> 5 * 3 = 15, carry = 1. 5 * 4 = 20 + 1 (carry) = 21, carry = 2. 5 * 2 = 10 + 2 (carry) = 12, carry = 1. Result: 5 -> 1 -> 2. Second, multiply l1 by 6 (shifting by one place): <ul style="list-style-type: none"> 6 * 3 = 18, carry = 1. 6 * 4 = 24 + 1 (carry) = 25, carry = 2. 6 * 2 = 12 + 2 (carry) = 14, carry = 1. Result: 8 -> 5 -> 4 -> 0 -> 0. Third, multiply l1 by 4 (shifting by two places): <ul style="list-style-type: none"> 4 * 3 = 12, carry = 1. 4 * 4 = 16 + 1 (carry) = 17, carry = 1. 4 * 2 = 8 + 1 (carry) = 9, carry = 0. Result: 2 -> 7 -> 9 -> 0 -> 0. Adding the Intermediate Results: <ul style="list-style-type: none"> Add the first product 5 -> 1 -> 2 to the result. Add the second product 8 -> 5 -> 4 -> 0 -> 0 to the result. Add the third product 2 -> 7 -> 9 -> 0 -> 0 to the result. Final Output: <ul style="list-style-type: none"> The result after adding all the intermediate products is 1 -> 5 -> 9 -> 0 -> 3 -> 0, which is the correct result for 342 * 465 = 159030. 	<p>Final Output:</p> <p>159030</p>							result)
--	---	---	---	--	--	--	--	--	--	---------

```

        return reverse(dummy->next);
    }

// Function to print the
linked list
void printList(Node* node) {
    while (node != nullptr)
    {
        cout << node->val <<
" ";
        node = node->next;
    }
    cout << endl;
}

// Function to create a
linked list from an array
of integers
Node* createList(int
values[], int n) {
    Node* dummy = new
Node(-1);
    Node* prev = dummy;
    for (int i = 0; i < n; ++i)
    {
        prev->next = new
Node(values[i]);
        prev = prev->next;
    }
    return dummy->next;
}

int main() {
    // Hardcoding the lists
    // First list: 3 -> 4 -> 2
    (represents the number
243)
    int arr1[] = {3, 4, 2};
    int n1 = sizeof(arr1) /
sizeof(arr1[0]);
    Node* head1 =
createList(arr1, n1);

    // Second list: 4 -> 6 ->
5 (represents the number
564)
    int arr2[] = {4, 6, 5};
    int n2 = sizeof(arr2) /
sizeof(arr2[0]);
    Node* head2 =
createList(arr2, n2);

    // Multiplying the two
linked lists
    Node* ans =
multiplyTwoLL(head1,
head2);

    // Printing the result
}

```

```
printList(ans);  
    return 0;  
}  
1 5 9 0 3 0
```

Pair Wise swap in C++

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

// Node class definition
class Node {
public:
    int data;
    Node* next;

    Node(int d) {
        data = d;
        next = nullptr;
    }
};

// PairwiseSwapLL class definition
class PairwiseSwapLL {
public:
    Node* head;

    PairwiseSwapLL() {
        head = nullptr;
    }

    // Method to print the elements of the list
    void printList(Node* node) {
        while (node != nullptr) {
            cout << node->data << " ";
            node = node->next;
        }
        cout << endl;
    }

    // Method to perform pairwise swapping of nodes
    Node* pairWiseSwap(Node* node) {
        if (node == nullptr || node->next == nullptr) {
            return node;
        }

        Node* remaining = node->next->next;
        Node* newHead = node->next;
        node->next->next = node;
        node->next = pairWiseSwap(remaining);
        return newHead;
    }
};

int main() {
    // Create an instance of PairwiseSwapLL
    PairwiseSwapLL list;

    // Construct the linked list: 1->2->3->4->5->6->7
    list.head = new Node(1);
    list.head->next = new Node(2);
    list.head->next->next = new Node(3);
    list.head->next->next->next = new Node(4);
    list.head->next->next->next->next = new Node(5);
    list.head->next->next->next->next->next = new
    Node(6);
    list.head->next->next->next->next->next =
}
```

Dry Run Table

Input List: 1 → 2 → 3 → 4 → 5 → 6 → 7

Recursive Call	node	Swapped Pair	Remaining	Result after call
1	1	1 ↔ 2	3	2 → 1 → ?
2	3	3 ↔ 4	5	4 → 3 → ?
3	5	5 ↔ 6	7	6 → 5 → ?
4	7	no pair	nullptr	7

Backtracking:

- 4th call returns: 7
- 3rd call builds: 6 → 5 → 7
- 2nd call builds: 4 → 3 → 6 → 5 → 7
- 1st call builds: 2 → 1 → 4 → 3 → 6 → 5 → 7

Final Output:

2 1 4 3 6 5 7

```
new Node(7);

// Display the original list
cout << "Linked list before calling pairwiseSwap() "
<< endl;
list.printList(list.head);

// Perform pairwise swapping
list.head = list.pairWiseSwap(list.head);

// Display the list after pairwise swapping
cout << "Linked list after calling pairwiseSwap() "
<< endl;
list.printList(list.head);

// Clean up allocated memory
Node* curr = list.head;
Node* next = nullptr;
while (curr != nullptr) {
    next = curr->next;
    delete curr;
    curr = next;
}

return 0;
}
```

```
Linked list before calling pairwiseSwap()
1 2 3 4 5 6 7
Linked list after calling pairwiseSwap()
2 1 4 3 6 5 7
```

Palindrome in LL in C++

```
#include <iostream>
#include <stack>

using namespace std;

// Node class definition
class Node {
public:
    int data;
    Node* next;

    // Constructor
    Node(int d) {
        data = d;
        next = nullptr;
    }
};

// LinkedList class definition
class LinkedList {
private:
    Node* head;
    Node* tail;
    int size;

public:
    // Constructor
    LinkedList() {
        head = nullptr;
        tail = nullptr;
        size = 0;
    }

    // Method to add a node at the end of the list
    void addLast(int val) {
        Node* temp = new Node(val);
        if (size == 0) {
            head = tail = temp;
        } else {
            tail->next = temp;
            tail = temp;
        }
        size++;
    }

    // Method to display the elements of the list
    void display() {
        Node* temp = head;
        while (temp != nullptr) {
            cout << temp->data << " ";
            temp = temp->next;
        }
        cout << endl;
    }

    // Method to check if the linked list is a palindrome
    bool isPalindrome() {
        Node* slow = head;
        stack<int> stack;

        // Push elements of the first half of the linked list
    }
}
```

Dry Run for Your Example: 1 → 2 → 3 → 2 → 1

Step	Stack Contents	slow points to	Comparison
Push	1, 2	3	-
Skip	(middle: 3)	2	-
Check	Top: 2 vs 2	2	✓
Check	Top: 1 vs 1	1	✓

✓ Result: true

Let me know if you'd like a version that modifies the list

```
onto the stack
    while (slow != nullptr) {
        stack.push(slow->data);
        slow = slow->next;
    }

    // Compare elements of the second half of the
    linked list with the stack
    slow = head;
    while (slow != nullptr) {
        int top = stack.top();
        stack.pop();
        if (slow->data != top) {
            return false;
        }
        slow = slow->next;
    }

    return true;
};

// Main function to demonstrate LinkedList operations
int main() {
    // Create a linked list
    LinkedList list;

    // Add elements to the linked list
    list.addLast(1);
    list.addLast(2);
    list.addLast(3);
    list.addLast(2);
    list.addLast(1);

    // Check if the linked list is a palindrome
    cout << boolalpha << list.isPalindrome() << endl; //
Output: true

    return 0;
}
```

```
true
```

Remove duplicate in LL in C++

```
#include <iostream>
#include <unordered_set>
using namespace std;

// Node class for the linked list
class Node {
public:
    int data;
    Node* next;

    Node(int data) {
        this->data = data;
        this->next = nullptr;
    }
};

// Function to print the linked list
void printList(Node* head) {
    Node* current = head;
    while (current != nullptr) {
        cout << current->data;
        if (current->next != nullptr) {
            cout << " -> ";
        } else {
            cout << " -> null";
        }
        current = current->next;
    }
    cout << endl;
}

// Function to remove duplicates from the linked list
void deleteDups(Node* head) {
    if (head == nullptr || head->next == nullptr)
        return;

    Node* current = head;
    while (current != nullptr) {
        Node* runner = current;
        while (runner->next != nullptr) {
            if (runner->next->data == current->data) {
                runner->next = runner->next->next;
            } else {
                runner = runner->next;
            }
        }
        current = current->next;
    }
}

int main() {
    // Creating a linked list with 5 hard-coded nodes
    Node* head = new Node(1);
    head->next = new Node(2);
    head->next->next = new Node(2);
    head->next->next->next = new Node(3);
    head->next->next->next->next = new Node(4);
    head->next->next->next->next->next = new
    Node(3);
    head->next->next->next->next->next->next = new
    Node(5);
}
```

Creates a linked list: 1 -> 2 -> 2 -> 3 -> 4 -> 3 -> 5 -> null

Initial Linked List Creation

Node	Value	Next Points To
head	1	Node 2
head->next	2	Node 2
...	2	Node 3
...	3	Node 4
...	4	Node 3
...	3	Node 5
...	5	nullptr

Initial Output from `printList(head)`

Original Linked List:
1 -> 2 -> 2 -> 3 -> 4 -> 3 -> 5 -> null

deleteDups(head) Dry Run

Loop Over current Node

current->data	Duplicate(s) Found and Removed	Resulting List
1	None	1 → 2 → 2 → 3 → 4 → 3 → 5
2	Second 2 removed	1 → 2 → 3 → 4 → 3 → 5
3	Second 3 removed	1 → 2 → 3 → 4 → 5
4	None	1 → 2 → 3 → 4 → 5
5	None	1 → 2 → 3 → 4 → 5

Final Linked List After `deleteDups(head)`

Linked List after removing duplicates:

```
// Print the original linked list
cout << "Original Linked List:" << endl;
printList(head);

// Remove duplicates
deleteDups(head);

// Print the linked list after removing duplicates
cout << "Linked List after removing duplicates:" <<
endl;
printList(head);

return 0;
}
```

1 -> 2 -> 3 -> 4 -> 5 -> null

Original Linked List:

1 -> 2 -> 2 -> 3 -> 4 -> 3 -> 5 -> null

Linked List after removing duplicates:

1 -> 2 -> 3 -> 4 -> 5 -> null

Reverse LL in C++

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

// Node class definition
class Node {
public:
    int data;
    Node* next;

    Node(int d) {
        data = d;
        next = nullptr;
    }
};

// Function to display the linked list
void display(Node* head) {
    while (head != nullptr) {
        cout << head->data;
        if (head->next != nullptr) {
            cout << "->";
        }
        head = head->next;
    }
    cout << endl;
}

// Function to reverse the linked list recursively
Node* reverse(Node* head) {
    if (head == nullptr || head->next == nullptr) {
        return head;
    }

    Node* smallAns = reverse(head->next);
    head->next->next = head;
    head->next = nullptr;
    return smallAns;
}

// Function to reverse the linked list iteratively
Node* reverseI(Node* head) {
    if (head == nullptr || head->next == nullptr) {
        return head;
    }

    Node* prev = nullptr;
    Node* curr = head;
    Node* next = nullptr;
    while (curr != nullptr) {
        next = curr->next;
        curr->next = prev;
        prev = curr;
        curr = next;
    }
    return prev;
}

int main() {
    // Creating the linked list
    Node* one = new Node(1);
    Node* two = new Node(2);
    Node* three = new Node(3);
    Node* four = new Node(4);
}
```

Recursive Reversal: reverse(Node* head)

🔍 Dry Run (for list: 1 -> 2 -> 3)

Step	Call Stack (Function Call)	Action	Resulting Links
1	reverse(1)	Calls reverse(2)	-
2	reverse(2)	Calls reverse(3)	-
3	reverse(3)	Base case hit, returns 3	-
4	Back to reverse(2)	3->next = 2, 2->next = nullptr	3 → 2
5	Back to reverse(1)	2->next = 1, 1->next = nullptr	3 → 2 → 1

◇ Final Result: 3 → 2 → 1

🔄 Iterative Reversal: reverseI(Node* head)

🔍 Dry Run (on 3 → 2 → 1)

curr	prev	next	Action	New Links
3	null	2	3->next = null	3
2	3	1	2->next = 3	2 → 3
1	2	null	1->next = 2	1 → 2 → 3

◇ Final Result: 1 → 2 → 3

```

Node* five = new Node(5);
Node* six = new Node(6);
Node* seven = new Node(7);
one->next = two;
two->next = three;
three->next = four;
four->next = five;
five->next = six;
six->next = seven;

// Displaying the original list
cout << "Original List: ";
display(one);

// Reversing the list recursively
cout << "List after recursive reversal: ";
Node* revRec = reverse(one);
display(revRec);

// Reversing the list iteratively
cout << "List after iterative reversal: ";
Node* revIter = reverseI(revRec);
display(revIter);

// Deallocating memory
delete revIter;

return 0;
}

```

Original List: 1->2->3->4->5->6->7

List after recursive reversal: 7->6->5->4->3->2->1

List after iterative reversal: 1->2->3->4->5->6->7

Segregate Even Odd in C++

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

class Node {
public:
    int val;
    Node* next;

    Node(int val) {
        this->val = val;
        this->next = nullptr;
    }
};

Node* segregateEvenOdd(Node* head) {
    if (head == nullptr || head->next == nullptr)
        return head;

    Node* dummyEven = new Node(-1);
    Node* dummyOdd = new Node(-1);
    Node* evenTail = dummyEven;
    Node* oddTail = dummyOdd;
    Node* curr = head;

    while (curr != nullptr) {
        if (curr->val % 2 != 0) {
            oddTail->next = curr;
            oddTail = oddTail->next;
        } else {
            evenTail->next = curr;
            evenTail = evenTail->next;
        }

        curr = curr->next;
    }

    evenTail->next = dummyOdd->next;
    oddTail->next = nullptr;

    Node* result = dummyEven->next;
    delete dummyEven;
    delete dummyOdd;
    return result;
}

void push(Node*& head, int new_data) {
    Node* new_node = new Node(new_data);
    new_node->next = head;
    head = new_node;
}

void printList(Node* node) {
    while (node != nullptr) {
        cout << node->val << " ";
        node = node->next;
    }
    cout << endl;
}

int main() {
    Node* head = nullptr;
```

What This Code Does

1. Builds a linked list: 6 -> 9 -> 10 -> 11
2. Separates **even** and **odd** numbers.
3. Appends odd list **after** the even list.
4. Prints the result: 6 -> 10 -> 9 -> 11

🔗 Linked List Construction (push)

push inserts at the head. So insertion order is:

Push Order	Value Inserted	List After Push
1	11	11
2	10	10 → 11
3	9	9 → 10 → 11
4	6	6 → 9 → 10 → 11

⚡ segregateEvenOdd(head) Dry Run

curr->val	Even/Odd	Action	Even List	Odd List
6	Even	Added to even list	6	-
9	Odd	Added to odd list	6	9
10	Even	Added to even list	6 → 10	9
11	Odd	Added to odd list	6 → 10	9 → 11

Then:

- evenTail->next = dummyOdd->next connects 6 → 10 → 9 → 11
- oddTail->next = nullptr ends the list

💻 Final Output from printList(head1)

6 10 9 11

❖ Summary

Before Segregation After Segregation

6 → 9 → 10 → 11 6 → 10 → 9 → 11

```
push(head, 11);
push(head, 10);
push(head, 9);
push(head, 6);

Node* head1 = segregateEvenOdd(head);
printList(head1);

return 0;
}
```

```
6 10 9 11
```

Sublist in C++

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

class Node {
public:
    int data;
    Node* next;

    Node(int data) {
        this->data = data;
        this->next = nullptr;
    }
};

void printList(Node* head) {
    Node* current = head;
    while (current != nullptr) {
        cout << current->data << " -> ";
        current = current->next;
    }
    cout << "null" << endl;
}

void sublists(Node* head) {
    Node* i = head;
    while (i != nullptr) {
        Node* j = i;
        while (j != nullptr) {
            cout << j->data << " -> ";
            j = j->next;
        }
        cout << "null" << endl;
        i = i->next;
    }
}

int main() {
    // Create a linked list with 5 hard-coded nodes
    Node* head = new Node(1);
    head->next = new Node(2);
    head->next->next = new Node(2);
    head->next->next->next = new Node(3);
    head->next->next->next->next = new Node(4);
    head->next->next->next->next = new
    Node(3);
    head->next->next->next->next->next = new
    Node(5);

    // Print the linked list
    printList(head);

    // Print all sublists
    sublists(head);

    // Clean up memory
    Node* current = head;
    while (current != nullptr) {
        Node* next = current->next;
        delete current;
        current = next;
    }
}
```

Linked List Creation

Step	Node Created	data	next Points To
1	head	1	Node with 2
2	head->next	2	Node with 2
3	...	2	Node with 3
4	...	3	Node with 4
5	...	4	Node with 3
6	...	3	Node with 5
7	...	5	nullptr

↑ printList(head) Output

1 -> 2 -> 2 -> 3 -> 4 -> 3 -> 5 -> null

↗ sublists(head) Dry Run Table

Outer Loop (i->data)	Inner Loop Iteration (→ values printed)
1	1 -> 2 -> 2 -> 3 -> 4 -> 3 -> 5 -> null
2 (1st)	2 -> 2 -> 3 -> 4 -> 3 -> 5 -> null
2 (2nd)	2 -> 3 -> 4 -> 3 -> 5 -> null
3	3 -> 4 -> 3 -> 5 -> null
4	4 -> 3 -> 5 -> null
3 (last)	3 -> 5 -> null
5	5 -> null

⚡ Cleanup (Memory Deallocation)

Step	Node Deleted	data
1	head	1
2		2
3		2
4		3
5		4

	Step	Node Deleted	data	
return 0; }	6		3	
1 -> 2 -> 2 -> 3 -> 4 -> 3 -> 5 -> null 1 -> 2 -> 2 -> 3 -> 4 -> 3 -> 5 -> null 2 -> 2 -> 3 -> 4 -> 3 -> 5 -> null 2 -> 3 -> 4 -> 3 -> 5 -> null 3 -> 4 -> 3 -> 5 -> null 4 -> 3 -> 5 -> null 3 -> 5 -> null 5 -> null	7		5	

Sumlist in C++																																									
<pre>#include <iostream> using namespace std; // Node class for the linked list class Node { public: int data; Node* next; // Default constructor Node() { data = 0; next = nullptr; } // Constructor with data parameter Node(int data) { this->data = data; next = nullptr; } void setNext(Node* next) { this->next = next; } }; // Function to print the linked list void printList(Node* head) { Node* current = head; while (current != nullptr) { cout << current->data << " -> "; current = current->next; } cout << "null" << endl; } // Function to add two linked lists // representing numbers Node* add(Node* l1, Node* l2, int carry) { if (l1 == nullptr && l2 == nullptr && carry == 0) { return nullptr; } Node* result = new Node(); int value = carry; if (l1 != nullptr) { value += l1->data; } if (l2 != nullptr) { value += l2->data; } result->data = value % 10; if (l1 != nullptr l2 != nullptr) { Node* more = add(l1 == nullptr ? nullptr : l1->next, l2 == nullptr ? nullptr : l2- >next, value >= 10 ? 1 : 0); result->setNext(more); } return result; }</pre>	<p>What the Code Does</p> <ul style="list-style-type: none"> • Adds two numbers represented by linked lists in reverse order (just like how we add numbers manually from right to left). • Example: <ul style="list-style-type: none"> ◦ List 1: 7 -> 1 -> 6 = 617 ◦ List 2: 5 -> 9 -> 2 = 295 ◦ Sum: 617 + 295 = 912 ◦ Result list: 2 -> 1 -> 9 																																								
	<p> Input Linked Lists</p> <table> <thead> <tr> <th>List</th><th>Nodes</th><th>Represents</th></tr> </thead> <tbody> <tr> <td>l1</td><td>7 -> 1 -> 6</td><td>617</td></tr> <tr> <td>l2</td><td>5 -> 9 -> 2</td><td>295</td></tr> </tbody> </table>	List	Nodes	Represents	l1	7 -> 1 -> 6	617	l2	5 -> 9 -> 2	295																															
List	Nodes	Represents																																							
l1	7 -> 1 -> 6	617																																							
l2	5 -> 9 -> 2	295																																							
	<p> add(l1, l2, carry) Dry Run</p> <table border="1"> <thead> <tr> <th>Step</th><th>l1->data</th><th>l2->data</th><th>Carry In</th><th>Sum</th><th>Digit Stored</th><th>Carry Out</th><th>Notes</th></tr> </thead> <tbody> <tr> <td>1</td><td>7</td><td>5</td><td>0</td><td>12</td><td>2</td><td>1</td><td>result->data = 2</td></tr> <tr> <td>2</td><td></td><td>9</td><td>1</td><td>11</td><td>1</td><td>1</td><td>result->next->data = 1</td></tr> <tr> <td>3</td><td>6</td><td>2</td><td>1</td><td>9</td><td>9</td><td>0</td><td>result->next->next->data = 9</td></tr> <tr> <td>4</td><td>null</td><td>null</td><td>0</td><td>-</td><td>-</td><td>-</td><td>Recursion stops</td></tr> </tbody> </table>	Step	l1->data	l2->data	Carry In	Sum	Digit Stored	Carry Out	Notes	1	7	5	0	12	2	1	result->data = 2	2		9	1	11	1	1	result->next->data = 1	3	6	2	1	9	9	0	result->next->next->data = 9	4	null	null	0	-	-	-	Recursion stops
Step	l1->data	l2->data	Carry In	Sum	Digit Stored	Carry Out	Notes																																		
1	7	5	0	12	2	1	result->data = 2																																		
2		9	1	11	1	1	result->next->data = 1																																		
3	6	2	1	9	9	0	result->next->next->data = 9																																		
4	null	null	0	-	-	-	Recursion stops																																		
	<p> Result Linked List After Addition</p> <p>2 -> 1 -> 9 -> null</p>																																								

```
}

int main() {
    // Creating two linked lists representing
    // numbers
    Node* head1 = new Node(7);
    head1->next = new Node(1);
    head1->next->next = new Node(6);

    Node* head2 = new Node(5);
    head2->next = new Node(9);
    head2->next->next = new Node(2);

    // Adding the two linked lists
    Node* result = add(head1, head2, 0);

    // Printing the result linked list
    cout << "Result of addition:" << endl;
    printList(result);

    return 0;
}
```

Result of addition:

2 -> 1 -> 9 -> null

Binary Tree to CDLL in C++

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

class Node {
public:
    int data;
    Node* left;
    Node* right;

    Node(int data) {
        this->data = data;
        this->left = nullptr;
        this->right = nullptr;
    }
};

class BinartTree2CDLL {
public:
    // Function to concatenate two circular doubly
    // linked lists
    Node* concatenate(Node* H1, Node* H2) {
        if (H1 == nullptr) return H2;
        if (H2 == nullptr) return H1;

        Node* T1 = H1->left;
        Node* T2 = H2->left;

        T1->right = H2;
        H2->left = T1;

        T2->right = H1;
        H1->left = T2;

        return H1;
    }

    // Function to convert binary tree into circular
    // doubly linked list
    Node* bTreeToClist(Node* root) {
        if (root == nullptr) return nullptr;

        Node* l = bTreeToClist(root->left);
        Node* r = bTreeToClist(root->right);

        root->left = root->right = root;

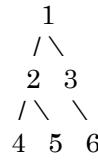
        Node* result = concatenate(concatenate(l, root),
r);

        return result;
    }

    // Function to print the circular doubly linked list
    void printCList(Node* head) {
        if (head == nullptr) return;

        Node* temp = head;
        do {
            cout << temp->data << " ";
            temp = temp->right;
        } while (temp != head);
    }
}
```

Your code to convert a **Binary Tree to a Circular Doubly Linked List (CDLL)** is elegant and correct. You're using **in-order traversal** with recursive linking, which is the standard and efficient approach. Let's break it down with a **dry run + visual table** using the tree:



⌚ Step-by-Step Dry Run (In-order traversal)

Traversal order: 4 → 2 → 5 → 1 → 3 → 6

Call Stack Depth	Node Visited	Left CDLL	Right CDLL	Resulting CDLL
1	4	null	null	4
1	5	null	null	5
2	2	4	5	4 ⇌ 2 ⇌ 5
1	6	null	null	6
2	3	null	6	3 ⇌ 6
3 (root)	1	4 ⇌ 2 ⇌ 5	3 ⇌ 6	4 ⇌ 2 ⇌ 5 ⇌ 1 ⇌ 3 ⇌ 6

- ⇌ means CDLL bidirectional links.
- At each recursive return, you concatenate left CDLL, root (self-circular), and right CDLL.

❖ Output

Circular Doubly Linked List:
4 2 5 1 3 6

```

        cout << endl;
    }

// Main method to test the bTreeToCList function
int main() {
    BinartTree2CDLL solution;

    // Creating a sample binary tree:
    //   1
    //   / \
    //   2   3
    //   / \   \
    //   4   5   6
    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->right = new Node(6);

    Node* head = solution.bTreeToCList(root);

    cout << "Circular Doubly Linked List:" << endl;
    solution.printCList(head);

    // Clean up memory
    // In a real-world scenario, you would implement a
    // function to delete the tree nodes.
    // For brevity, memory cleanup is not shown in this
    // example.

    return 0;
}

```

Output:-

Circular Doubly Linked List:
4 2 5 1 3 6

Binary Tree to LL in C++

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

// TreeNode class definition
class TreeNode {
public:
    int val;
    TreeNode* left;
    TreeNode* right;

    TreeNode(int x) {
        val = x;
        left = nullptr;
        right = nullptr;
    }
};

class FlattenBinaryTreeToLinkedList {
public:
    void flatten(TreeNode* root) {
        flattenHelper(root);
    }

private:
    TreeNode* flattenHelper(TreeNode* root) {
        if (root == nullptr) return nullptr;

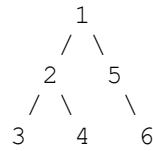
        TreeNode* leftTail = flattenHelper(root->left);
        TreeNode* rightTail = flattenHelper(root->right);

        if (leftTail != nullptr) {
            leftTail->right = root->right; // Connect
            the end of the left subtree to the start of the right
            subtree
            root->right = root->left; // Move the
            left subtree to the right
            root->left = nullptr; // Nullify the
            left pointer
        }

        // Return the last node in the flattened tree
        if (rightTail != nullptr) {
            return rightTail;
        } else if (leftTail != nullptr) {
            return leftTail;
        } else {
            return root;
        }
    }
};

public:
    // Utility function to print the flattened tree
    void printFlattenedTree(TreeNode* root) {
        while (root != nullptr) {
            cout << root->val << " ";
            root = root->right;
        }
        cout << endl;
    }
}
```

Absolutely! Let's dry run your `flatten` function **with a step-by-step table**, using this binary tree:



The goal is to flatten this tree into a **linked list using right pointers in pre-order traversal**:

1 → 2 → 3 → 4 → 5 → 6

Dry Run Table:

Step	Node Visited	Left Subtree Tail	Right Subtree Tail	Action Taken	Resulting Right Chain (Partial)
1	3	nullptr	nullptr	Leaf node → 3 return 3	
2	4	nullptr	nullptr	Leaf node → 4 return 4	
3	2	3	4	Move left to right: 2->left becomes nullptr, 2->right = 3, 3->right = 4	2 → 3 → 4
4	6	nullptr	nullptr	Leaf node → 6 return 6	
5	5	nullptr	6	No left to move → do nothing, return 6	5 → 6
6	1	4 (tail of 2)	6 (tail of 5)	Move left to right: 1->right = 2, 2->right = 3, 3->right = 4, 4->right = 5 (attach 5 to end)	1 → 2 → 3 → 4 → 5 → 6

```

// Function to delete a binary tree to free
memory
void deleteTree(TreeNode* root) {
    if (root == nullptr) return;
    deleteTree(root->left);
    deleteTree(root->right);
    delete root;
}

};

int main() {
    FlattenBinaryTreeToLinkedList solution;

    // Creating a sample binary tree:
    //   1
    //   / \
    //   2   5
    //   / \ \
    //  3   4   6
    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);

    cout << "Original Tree:" << endl;
    solution.printFlattenedTree(root); // This will
just print the root node, as the tree is not
flattened yet

    solution.flatten(root);

    cout << "Flattened Tree:" << endl;
    solution.printFlattenedTree(root);

    // Clean up memory
    solution.deleteTree(root);

    return 0;
}

```

Output:-

1 → 2 → 3 → 4 → 5 → 6 → nullptr

			of 2 chain)	
--	--	--	----------------	--

Final Result:

The flattened tree is:

1 → 2 → 3 → 4 → 5 → 6 → nullptr

CopyListwithRandomPointers in C++

```
#include <iostream>
#include <unordered_map>

// Definition for a Node.
struct Node {
    int val;
    Node* next;
    Node* random;

    Node(int _val) {
        val = _val;
        next = nullptr;
        random = nullptr;
    }
};

Node* copyRandomList(Node* head) {
    if (head == nullptr) return nullptr;

    std::unordered_map<Node*, Node*> map;
    Node* curr = head;

    // First pass: create all nodes and store them in the map.
    while (curr != nullptr) {
        map[curr] = new Node(curr->val);
        curr = curr->next;
    }

    // Second pass: assign next and random pointers.
    curr = head;
    while (curr != nullptr) {
        map[curr]->next = map[curr->next];
        map[curr]->random = map[curr->random];
        curr = curr->next;
    }

    return map[head];
}

void printList(Node* head) {
    while (head != nullptr) {
        std::cout << "Node(" << head->val << ")";
        if (head->random != nullptr) {
            std::cout << " [Random(" << head->random->val << ")]";
        }
        std::cout << " -> ";
        head = head->next;
    }
    std::cout << "null" << std::endl;
}

int main() {
    Node* head = new Node(1);
    head->next = new Node(2);
    head->next->next = new Node(3);
    head->random = head->next->next;
    head->next->random = head;

    Node* result = copyRandomList(head);
}
```

Goal: Deep copy a linked list where each node has next and random pointers.

Given input:

```
1 -> 2 -> 3
|   |
v   v
3   1
```

❖ Step-by-Step Dry Run Table

Step	Operation	Affected Node	Explanation
First Pass	map[1] = new Node(1)	Node 1	Creates a copy of node 1
	map[2] = new Node(2)	Node 2	Creates a copy of node 2
	map[3] = new Node(3)	Node 3	Creates a copy of node 3
Second Pass	map[1]->next = map[2]	Node 1 copy	Sets next of copied 1 to copied 2
	map[1]->random = map[3]	Node 1 copy	Sets random of copied 1 to copied 3 (like original)
	map[2]->next = map[3]	Node 2 copy	Sets next of copied 2 to copied 3
	map[2]->random = map[1]	Node 2 copy	Sets random of copied 2 to copied 1
	map[3]->next = map[nullptr] = null	Node 3 copy	Last node, next is null
	map[3]->random = map[nullptr]	Node 3 copy	random was not set originally, stays null

❖ Final Output:

Copied list:

```
1 [Random(3)] -> 2 [Random(1)] -> 3 -> null
```

```
printList(result);

// Free the allocated memory
Node* curr = result;
while (curr != nullptr) {
    Node* temp = curr;
    curr = curr->next;
    delete temp;
}

return 0;
}
```

Output:-
0

Cycle in C++

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

// Definition of a Node in the linked list
struct Node {
    int val;
    Node* next;
    Node(int x) {
        val = x;      // Assigns the parameter x to the
                      // member variable val
        next = nullptr; // Initializes the next pointer to
                        // nullptr
    }
};

// Function to detect if there is a cycle in the linked
list
bool hasCycle(Node* head) {
    if (head == nullptr || head->next == nullptr) {
        return false;
    }

    Node* slow = head;
    Node* fast = head;

    while (fast != nullptr && fast->next != nullptr) {
        slow = slow->next;
        fast = fast->next->next;

        if (slow == fast) {
            return true; // Cycle detected
        }
    }

    return false; // No cycle found
}

int main() {
    // Creating a linked list: 1 -> 2 -> 3 -> 4 -> 5
    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);

    // Creating a cycle by pointing the next of last node
    // to the node with value 3 (index 2)
    Node* tail = head;
    while (tail->next != nullptr) {
        tail = tail->next;
    }
    Node* cycleNode = head->next->next; // Node with
    value 3
    tail->next = cycleNode;

    // Check if the cycle is present
    cout << (hasCycle(head) ? "Cycle is present" : "No
cycle") << endl;

    return 0;
}
```

Core Logic Recap

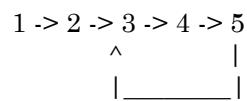
Floyd's algorithm uses:

- slow: moves 1 step at a time.
- fast: moves 2 steps at a time.

If there's a cycle, slow and fast will eventually meet inside the loop.

Dry Run

Linked List:



Cycle: 5 -> 3 creates a loop back to node with value 3.

Dry Run Table

Iteration	slow value	fast value	Notes
1	2	3	both moved: slow+1, fast+2
2	3	5	fast jumps into cycle
3	4	4	slow == fast → cycle found

Output:

Cycle is present

{	
Output:- Cycle is present	

MergeSort in C++

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

// Definition for a singly-linked list node
struct ListNode {
    int data;
    ListNode* next;

    ListNode(int x) {
        data = x;
        next = nullptr;
    }
};

// Function to merge two sorted linked lists
ListNode* merge(ListNode* h1, ListNode* h2) {
    if (h1 == nullptr) return h2;
    if (h2 == nullptr) return h1;

    ListNode* ans = nullptr;
    ListNode* t = nullptr;

    if (h1->data < h2->data) {
        ans = h1;
        t = h1;
        h1 = h1->next;
    } else {
        ans = h2;
        t = h2;
        h2 = h2->next;
    }

    while (h1 != nullptr && h2 != nullptr) {
        if (h1->data < h2->data) {
            t->next = h1;
            t = t->next;
            h1 = h1->next;
        } else {
            t->next = h2;
            t = t->next;
            h2 = h2->next;
        }
    }

    if (h1 != nullptr) t->next = h1;
    if (h2 != nullptr) t->next = h2;

    return ans;
}

// Function to find the middle of the linked list
ListNode* mid(ListNode* h) {
    ListNode* slow = h;
    ListNode* fast = h;

    while (fast != nullptr && fast->next != nullptr) {
        slow = slow->next;
        fast = fast->next->next;
    }
}
```

Dry Run — Function Calls Breakdown:

1. Initial Call:

`mergeSort(4 -> 2 -> 1 -> 3)`

Midpoint = 1 (list breaks into):

- $h1 = 4 \rightarrow 2$
- $h2 = 1 \rightarrow 3$

2. Recursive Breakdown:

Level	Call	Mid Node	Left Part	Right Part
1	<code>mergeSort(4->2->1->3)</code>	1	4->2	1->3
2	<code>mergeSort(4->2)</code>	2	4	2
2	<code>mergeSort(1->3)</code>	3	1	3

3. Merge Steps (Bottom-Up):

Step	Merge Call	Output
1	<code>merge(4, 2)</code>	2 -> 4
2	<code>merge(1, 3)</code>	1 -> 3
3	<code>merge(2->4, 1->3)</code>	1 -> 2 -> 3 -> 4

❖ Final Output:

Sorted Linked List: 1 -> 2 -> 3 -> 4

```

        return slow;
    }

// Function to perform merge sort on the linked list
ListNode* mergeSort(ListNode* h1) {
    if (h1 == nullptr || h1->next == nullptr) return h1;

    ListNode* m = mid(h1);
    ListNode* h2 = m->next;
    m->next = nullptr;

    ListNode* t1 = mergeSort(h1);
    ListNode* t2 = mergeSort(h2);
    ListNode* t3 = merge(t1, t2);

    return t3;
}

// Function to print the linked list
void printList(ListNode* head) {
    ListNode* temp = head;
    while (temp != nullptr) {
        cout << temp->data << " ";
        temp = temp->next;
    }
    cout << endl;
}

int main() {
    // Creating an example linked list: 4 -> 2 -> 1 -> 3
    ListNode* head = new ListNode(4);
    head->next = new ListNode(2);
    head->next->next = new ListNode(1);
    head->next->next->next = new ListNode(3);

    cout << "Original Linked List:" << endl;
    printList(head);

    head = mergeSort(head);

    cout << "Sorted Linked List:" << endl;
    printList(head);

    // Clean up allocated memory
    ListNode* current = head;
    while (current != nullptr) {
        ListNode* next = current->next;
        delete current;
        current = next;
    }

    return 0;
}

```

Output:-
0

OddEven in C++

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

// Node class definition
class Node {
public:
    int data;
    Node* next;

    Node(int val) {
        data = val;
        next = nullptr;
    }
};

// LinkedList class definition
class LinkedList {
public:
    Node* head;
    Node* tail;
    int size;

    LinkedList() {
        head = nullptr;
        tail = nullptr;
        size = 0;
    }

    // Method to add a node at the end of the list
    void addLast(int val) {
        Node* newNode = new Node(val);
        if (size == 0) {
            head = tail = newNode;
        } else {
            tail->next = newNode;
            tail = newNode;
        }
        size++;
    }

    // Method to display the elements of the list
    void display() {
        Node* temp = head;
        while (temp != nullptr) {
            cout << temp->data << " ";
            temp = temp->next;
        }
        cout << endl;
    }

    // Method to remove the first node from the list
    void removeFirst() {
        if (size == 0) {
            cout << "List is empty" << endl;
        } else if (size == 1) {
            head = tail = nullptr;
            size = 0;
        } else {
            head = head->next;
            size--;
        }
    }
}
```

Initial List:

Original List: 2 -> 8 -> 9 -> 1 -> 5 -> 4 -> 3

⌚ Dry Run Table for oddEven() Method

We'll track how elements are moved to either the **odd** or **even** list.

Step	Current Node (val)	Is Even?	Action	Odd List	Even List
1	2	↙ Yes	Add to Even		2
2	8	↙ Yes	Add to Even		2 -> 8
3	9	✗ No	Add to Odd	9	2 -> 8
4	1	✗ No	Add to Odd	9 -> 1	2 -> 8
5	5	✗ No	Add to Odd	9 -> 1 -> 5	2 -> 8
6	4	↙ Yes	Add to Even	9 -> 1 -> 5 -> 4	2 -> 8
7	3	✗ No	Add to Odd	9 -> 1 -> 5 -> 3	2 -> 8 -> 4

✳️ Reconnecting Lists

- Since **both odd and even lists exist**, we connect:
 - odd.tail->next = even.head
 - New head = odd.head
 - New tail = even.tail
 - New size = odd.size + even.size = 4 + 3 = 7

● Result after oddEven():

List after Odd-Even Segregation: 9 -> 1 -> 5 -> 3 -> 2 -> 8 -> 4

✚ Add 10 at beginning, 100 at end:

- After addFirst(10): 10 -> 9 -> 1 -> 5 -> 3 -> 2 -> 8 -> 4
- After addLast(100): 10 -> 9 -> 1 -> 5 -> 3 -> 2 -> 8 -> 4 -> 100

✓ Final Output:

```

    }

// Method to get the data of the first node
int getFirst() {
    if (size == 0) {
        cout << "List is empty" << endl;
        return -1;
    } else {
        return head->data;
    }
}

// Method to add a node at the beginning of the list
void addFirst(int val) {
    Node* newNode = new Node(val);
    newNode->next = head;
    head = newNode;

    if (size == 0) {
        tail = newNode;
    }

    size++;
}

// Method to segregate odd and even nodes in the
list
void oddEven() {
    LinkedList odd;
    LinkedList even;

    while (size > 0) {
        int val = getFirst();
        removeFirst();

        if (val % 2 == 0) {
            even.addLast(val);
        } else {
            odd.addLast(val);
        }
    }

    if (odd.size > 0 && even.size > 0) {
        odd.tail->next = even.head;
        head = odd.head;
        tail = even.tail;
        size = odd.size + even.size;
    } else if (odd.size > 0) {
        head = odd.head;
        tail = odd.tail;
        size = odd.size;
    } else if (even.size > 0) {
        head = even.head;
        tail = even.tail;
        size = even.size;
    }
};

int main() {
    // Initialize LinkedList
}

```

List after adding 10 at the beginning and 100 at the end: 10 -> 9 -> 1 -> 5 -> 3 -> 2 -> 8 -> 4 -> 100

```
LinkedList l1;

// Add elements to the LinkedList
l1.addLast(2);
l1.addLast(8);
l1.addLast(9);
l1.addLast(1);
l1.addLast(5);
l1.addLast(4);
l1.addLast(3);

// Display original list
cout << "Original List: ";
l1.display();

// Perform odd-even segregation
l1.oddEven();

// Display list after odd-even segregation
cout << "List after Odd-Even Segregation: ";
l1.display();

// Add elements at the beginning and end
int a = 10;
int b = 100;
l1.addFirst(a);
l1.addLast(b);

// Display list after adding elements
cout << "List after adding " << a << " at the
beginning and " << b << " at the end: ";
l1.display();

return 0;
}
```

Output:-

List after adding 10 at the beginning and 100 at the end: 10 -> 9 -> 1 -> 5 -> 3 -> 2 -> 8 -> 4 -> 100

Palindrome in C++

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

// Node class for the linked list
class Node {
public:
    int val;
    Node* next;

    Node(int val) {
        this->val = val;
        this->next = nullptr;
    }
};

// Function to find the middle node of the linked list
Node* midNode(Node* head) {
    if (head == nullptr || head->next == nullptr)
        return head;

    Node* slow = head;
    Node* fast = head;

    while (fast->next != nullptr && fast->next->next != nullptr) {
        slow = slow->next;
        fast = fast->next->next;
    }

    return slow;
}

// Function to reverse a linked list
Node* reverseOfLL(Node* head) {
    if (head == nullptr || head->next == nullptr)
        return head;

    Node* prev = nullptr;
    Node* curr = head;
    Node* forw = nullptr;

    while (curr != nullptr) {
        forw = curr->next;
        curr->next = prev;
        prev = curr;
        curr = forw;
    }

    return prev;
}

// Function to check if a linked list is a palindrome
bool isPalindrome(Node* head) {
    if (head == nullptr || head->next == nullptr)
        return true;

    // Find the middle of the linked list
    Node* mid = midNode(head);

    // Reverse the second half of the list
    Node* nHead = mid->next;
```

Step-by-Step Dry Run Table

Step	Operation	Pointer/Variable	Value(s)
1	Find mid	slow, fast	Mid = 3 (slow stops here)
2	Reverse 2nd half	From node 2 -> 1	Reversed to 1 -> 2
3	Compare halves	1-2-3 vs 1-2	Matches fully
4	Restore 2nd half	Reverse back 1->2	Back to 2->1
5	Result		✓ true (Palindrome)

Output

true

```

mid->next = nullptr; // Split the list into two halves
nHead = reverseOfLL(nHead);

// Compare the two halves
Node* c1 = head;
Node* c2 = nHead;

bool res = true;
while (c2 != nullptr) { // Only need to compare until
c2 ends
    if (c1->val != c2->val) {
        res = false;
        break;
    }
    c1 = c1->next;
    c2 = c2->next;
}

// Restore the original list
nHead = reverseOfLL(nHead);
mid->next = nHead;

return res;
}

// Function to create a linked list from an array of
integers
Node* createList(int values[], int n) {
    Node* dummy = new Node(-1);
    Node* prev = dummy;
    for (int i = 0; i < n; ++i) {
        prev->next = new Node(values[i]);
        prev = prev->next;
    }
    return dummy->next;
}

int main() {
    // Hardcoding the linked list: 1 -> 2 -> 3 -> 2 -> 1
    int arr[] = {1, 2, 3, 2, 1};
    int n = sizeof(arr) / sizeof(arr[0]);
    Node* head = createList(arr, n);

    // Checking if the linked list is a palindrome
    cout << boolalpha << isPalindrome(head) <<
    endl; // should print true

    return 0;
}

```

Output:-

true

Reverse a LL in C++

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

// Node class definition
class Node {
public:
    int data;
    Node* next;

    Node(int d) {
        data = d;
        next = nullptr;
    }
};

// Function to display the linked list
void display(Node* head) {
    while (head != nullptr) {
        cout << head->data;
        if (head->next != nullptr) {
            cout << "->";
        }
        head = head->next;
    }
    cout << endl;
}

// Function to reverse the linked list recursively
Node* reverse(Node* head) {
    if (head == nullptr || head->next == nullptr) {
        return head;
    }
    Node* smallAns = reverse(head->next);
    head->next->next = head;
    head->next = nullptr;
    return smallAns;
}

// Function to reverse the linked list iteratively
Node* reverseI(Node* head) {
    if (head == nullptr || head->next == nullptr) {
        return head;
    }
    Node* prev = nullptr;
    Node* curr = head;
    Node* next = nullptr;
    while (curr != nullptr) {
        next = curr->next;
        curr->next = prev;
        prev = curr;
        curr = next;
    }
    return prev;
}

int main() {
    // Creating the linked list
    Node* one = new Node(1);
    Node* two = new Node(2);
    Node* three = new Node(3);
    Node* four = new Node(4);
}
```

Dry Run Table (Step-by-step Iteration)

Iteration	curr->data	next->data	prev->data	What Happens	List State
0	1	2	nullptr	Reverse 1->nullptr, move prev = 1, curr = 2	1
1	2	3	1	Reverse 2->1, move prev = 2, curr = 3	2 -> 1
2	3	4	2	Reverse 3->2, move prev = 3, curr = 4	3 -> 2 -> 1
3	4	5	3	Reverse 4->3, move prev = 4, curr = 5	4 -> 3 -> 2 -> 1
4	5	6	4	Reverse 5->4, move prev = 5, curr = 6	5 -> 4 -> 3 -> 2 -> 1
5	6	7	5	Reverse 6->5, move prev = 6, curr = 7	6 -> 5 -> 4 -> 3 -> 2 -> 1
6	7	nullptr	6	Reverse 7->6, move prev = 7, curr = nullptr	7 -> 6 -> 5 -> 4 -> 3 -> 2 -> 1

❖ Final Pointers:

- curr == nullptr → end of list
- prev == 7 → head of reversed list
- So, the function returns prev as the new head.

❖ Final Output:

List after iterative reversal: 7->6->5->4->3->2->1

```
Node* five = new Node(5);
Node* six = new Node(6);
Node* seven = new Node(7);
one->next = two;
two->next = three;
three->next = four;
four->next = five;
five->next = six;
six->next = seven;

// Displaying the original list
cout << "Original List: ";
display(one);

// Reversing the list recursively
cout << "List after recursive reversal: ";
Node* revRec = reverse(one);
display(revRec);

// Reversing the list iteratively
cout << "List after iterative reversal: ";
Node* revIter = reverseI(revRec);
display(revIter);

// Deallocating memory
delete revIter;

return 0;
}
```

Output:-

List after iterative reversal: 7->6->5->4->3->2->1

Rotate list by k C++

```
#include <iostream>

struct Node {
    int val;
    Node* next;
    Node(int x) {
        val = x;
        next = nullptr;
    }
};

Node* rotateRight(Node* head, int k) {
    if (head == nullptr || k == 0) return head;

    int length = 1;
    Node* tail = head;
    while (tail->next != nullptr) {
        tail = tail->next;
        length++;
    }

    k = k % length;
    if (k == 0) return head;

    Node* newTail = head;
    for (int i = 0; i < length - k - 1; i++) {
        newTail = newTail->next;
    }

    Node* newHead = newTail->next;
    newTail->next = nullptr;
    tail->next = head;

    return newHead;
}

void printList(Node* head) {
    while (head != nullptr) {
        std::cout << head->val << " -> ";
        head = head->next;
    }
    std::cout << "null" << std::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);

    Node* result = rotateRight(head, 2);
    printList(result);

    // Free the allocated memory
    Node* curr = result;
    while (curr != nullptr) {
        Node* temp = curr;
        curr = curr->next;
        delete temp;
    }
}
```

Problem Summary:

Rotate a singly linked list **to the right** by k places.

Input:

Linked List:

rust

CopyEdit

1 -> 2 -> 3 -> 4 -> 5

Rotate by k = 2

Dry Run Steps:

Step	Explanation	State
1	Initial list	1 -> 2 -> 3 -> 4 -> 5 -> null
2	Traverse list to find length and tail	length = 5, tail = 5
3	Normalize k: k = k % length = 2 % 5 = 2	Effective rotation is 2 places
4	Move to new tail: length - k - 1 = 5 - 2 - 1 = 2	Move 2 steps from head: node with value 3 is new tail
5	newTail = 3, newHead = 4, break link	newTail->next = nullptr, tail->next = head
6	New list after rotation	4 -> 5 -> 1 -> 2 -> 3 -> null

Final State:

- **Old Tail:** Node with value 5
- **Old Head:** Node with value 1
- **New Head:** Node with value 4
- **New Tail:** Node with value 3

Output:

4 -> 5 -> 1 -> 2 -> 3 -> null

```
    return 0;  
}
```

Output:-

4 -> 5 -> 1 -> 2 -> 3 -> null

Swap nodes in pairs in C++

```
#include <iostream>

struct Node {
    int val;
    Node* next;
    Node(int x) {
        val = x;
        next = nullptr;
    }
};

class SwapNodesInPairs {
public:
    Node* swapPairs(Node* head) {
        Node dummy(0);
        dummy.next = head;
        Node* current = &dummy;

        while (current->next != nullptr && current->next->next != nullptr) {
            Node* first = current->next;
            Node* second = current->next->next;

            first->next = second->next;
            second->next = first;
            current->next = second;

            current = first;
        }

        return dummy.next;
    }

    static void printList(Node* head) {
        while (head != nullptr) {
            std::cout << head->val << " -> ";
            head = head->next;
        }
        std::cout << "null" << std::endl;
    }
};

int main() {
    SwapNodesInPairs solution;

    Node* head = new Node(1);
    head->next = new Node(2);
    head->next->next = new Node(3);
    head->next->next->next = new Node(4);

    Node* result = solution.swapPairs(head);
    SwapNodesInPairs::printList(result);

    // Free the allocated memory
    Node* curr = result;
    while (curr != nullptr) {
        Node* temp = curr;
        curr = curr->next;
        delete temp;
    }
}
```

for input:

1 -> 2 -> 3 -> 4

The goal is to swap every two adjacent nodes. So, the expected output is:

2 -> 1 -> 4 -> 3

🧠 Key Pointers:

- dummy is a placeholder node that simplifies head manipulation.
- current starts at dummy.
- first and second are the two nodes to be swapped.
- The loop continues as long as there are at least 2 nodes ahead of current.

⌚ Dry Run Table:

Iteration	current Points To	first	second	Operation	List After Swap
1	dummy (0) → 1	1	2	Swap 1 and 2	2 → 1 → 3 → 4
				first->next = 3	
				second->next = 1, current->next = 2	
				current = first → moves to node 1	
2	current → 1	3	4	Swap 3 and 4	2 → 1 → 4 → 3
				first->next = nullptr	
				second->next = 3, current->next = 4	
				current = first → moves to node 3	

❖ Final Output:

return 0; }	2 -> 1 -> 4 -> 3 -> null
Output:- 2 -> 1 -> 4 -> 3 -> null	

Abbreviation in C++

```
#include <iostream>
#include <string>
using namespace std;

class Abbreviation {
public:
    static void solution(string str, string asf, int count,
int pos) {
        if (pos == str.length()) {
            if (count == 0) {
                cout << asf << endl;
            } else {
                cout << asf << count << endl;
            }
            return;
        }

        if (count > 0) {
            solution(str, asf + to_string(count) + str[pos],
0, pos + 1);
        } else {
            solution(str, asf + str[pos], 0, pos + 1);
        }

        solution(str, asf, count + 1, pos + 1);
    }
};

int main() {
    string str = "pep";
    Abbreviation::solution(str, "", 0, 0);
    return 0;
}
```

Dry Run Table (Step-by-Step)

We'll list:

- pos: current position in the string
- count: how many characters we've skipped (abbreviated)
- asf: abbreviation-so-far

pos	char	count	asf	Recursive Call
0	p	0	""	choose 'p' → asf = "p"
1	e	0	"p"	choose 'e' → asf = "pe"
2	p	0	"pe"	choose 'p' → asf = "pep"
3	—	0	"pep"	output: pep
2	p	1	"pe"	skip 'p' (count = 1)
3	—	1	"pe"	output: pe1
1	e	1	"p"	skip 'e' (count = 1)
2	p	0	"p1p"	count > 0 → add 1 then 'p'
3	—	0	"p1p"	output: p1p
2	p	2	"p"	skip 'p' (count = 2)
3	—	2	"p"	output: p2
0	p	1	""	skip 'p' (count = 1)
1	e	0	"1e"	count > 0 → add 1, then 'e'
2	p	0	"1ep"	choose 'p'
3	—	0	"1ep"	output: 1ep
2	p	1	"1e"	skip 'p' (count = 1)
3	—	1	"1e"	output: 1e1
1	e	1	""	skip 'e'
2	p	0	"2p"	count = 2 → asf = "2p"
3	—	0	"2p"	output: 2p
2	p	2	""	skip 'p'
3	—	3	""	output: 3

❖ Final Output:

pep
pe1
p1p

	p2 1ep 1e1 2p 3
Output:- pep pe1 p1p p2 1ep 1e1 2p 3	

All palindromic partition in C++

```
#include <iostream>
#include <string>
using namespace std;

class AllPalindromicPartition {
public:
    static void main() {
        string str = "abba";
        sol(str, "");
    }

    static void sol(string str, string asf) {
        if (str.length() == 0) {
            cout << asf << endl;
            return;
        }

        for (int i = 0; i < str.length(); i++) {
            string prefix = str.substr(0, i + 1);
            string ros = str.substr(i + 1);
            if (isPalin(prefix)) {
                sol(ros, asf + "(" + prefix + ")");
            }
        }
    }

    static bool isPalin(string s) {
        int li = 0;
        int ri = s.length() - 1;
        while (li < ri) {
            if (s[li] != s[ri]) {
                return false;
            }
            li++;
            ri--;
        }
        return true;
    }
};

int main() {
    AllPalindromicPartition::main();
    return 0;
}
```

Dry Run for Input "abba"

We will track the recursive calls with:

- str: Remaining string to process
- prefix: Currently selected prefix
- ros: Remaining string after prefix
- asf: Accumulated string so far
- Action: What's happening

Step	str	prefix	ros	asf	Action / Reason
1	abba	a	bba	(a)	'a' is palindrome → recurse
2	bba	b	ba	(a)(b)	'b' is palindrome → recurse
3	ba	b	a	(a)(b)(b)	'b' is palindrome → recurse
4	a	a	""	(a)(b)(b)(a)	↙ 'a' is palindrome → print
5	ba	ba	—	—	not palindrome ✗
6	bba	bb	a	(a)(bb)	↙ 'bb' is palindrome → recurse
7	a	a	""	(a)(bb)(a)	↙ 'a' is palindrome → print
8	bba	bba	—	—	not palindrome ✗
9	abba	ab	—	—	not palindrome ✗
10	abba	abb	—	—	not palindrome ✗
11	abba	abba	""	(abba)	↙ 'abba' is palindrome → print

↙ Final Output

(a)(b)(b)(a)
(a)(bb)(a)

	(abba)
Output:-	(a)(b)(b)(a) (a)(bb)(a) (abba)

Combinations in C++

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

void combinations(int cb, int nboxes, int ssf, int
ritems, string asf) {
    if (cb > nboxes) {
        if (ssf == ritems) {
            cout << asf << endl;
        }
        return;
    }
    combinations(cb + 1, nboxes, ssf + 1, ritems, asf +
"i");
    combinations(cb + 1, nboxes, ssf, ritems, asf + "-");
}

int main() {
    int nboxes = 3;
    int ritems = 2;
    combinations(1, nboxes, 0, ritems, "");
    return 0;
}
```

Dry Run with Table for Input:

- nboxes = 3
- ritems = 2

We're tracing the recursive calls:

- cb: current box index
- ssf: selected so far
- asf: answer so far

Step	cb	ssf	asf	Action
1	1	0	""	→ i at box 1 → recurse
2	2	1	"i"	→ i at box 2 → recurse
3	3	2	"ii"	→ i at box 3 → recurse
4	4	3	"iii"	✗ too many items (ssf > ritems)
5	3	2	"ii"	→ - at box 3 → ✓ print: ii-
6	2	1	"i"	→ - at box 2 → recurse
7	3	1	"i-"	→ i at box 3 → recurse
8	4	2	"i-i"	✓ valid → print: i-i
9	3	1	"i-"	→ - at box 3 → recurse
10	4	1	"i--"	✗ too few items
11	1	0	""	→ - at box 1 → recurse
12	2	0	"-"	→ i at box 2 → recurse
13	3	1	"-i"	→ i at box 3 → recurse
14	4	2	"-ii"	✓ valid → print: -ii
15	3	1	"-i"	→ - at box 3 → recurse
16	4	1	"-i-"	✗ too few items
17	2	0	"-"	→ - at box 2 → recurse
18	3	0	"--"	→ i at box 3 → recurse
19	4	1	"--i"	✗ too few items
20	3	0	"--"	→ - at box 3 → recurse
21	4	0	"---"	✗ too few items

✓ Final Output

ii-
i-i
-ii

Output:-

ii-
i-i
-ii

Friend's pairing in C++

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

int counter = 1;

void solution(int i, int n, vector<bool>& used, string asf) {
    if (i > n) {
        cout << counter << "." << asf << endl;
        counter++;
        return;
    }

    if (used[i]) {
        solution(i + 1, n, used, asf);
    } else {
        used[i] = true;
        solution(i + 1, n, used, asf + "(" + to_string(i) + ")");
    }

    for (int j = i + 1; j <= n; j++) {
        if (!used[j]) {
            used[j] = true;
            solution(i + 1, n, used, asf + "(" + to_string(i) +
                  "," + to_string(j) + ")");
            used[j] = false;
        }
    }
    used[i] = false;
}

int main() {
    int n = 3;
    vector<bool> used(n + 1, false);
    solution(1, n, used, "");
    return 0;
}
```

Function Logic Recap

Dry Run for n = 3

Step	i	used	Action	Output (if any)
1	1	[F, F, F, F]	1 unused → go alone: (1)	
2	2	[F, T, F, F]	2 unused → go alone: (2)	
3	3	[F, T, T, F]	3 unused → go alone: (3)	1. (1) (2) (3)
4			backtrack to pair 2 and 3	2. (1) (2, 3)
5			backtrack to try 1 with 2	
6	2	[F, T, T, F]	3 unused → go alone: (3)	3. (1, 2) (3)
7			backtrack	
8			try 1 with 3	
9	2	[F, T, F, T]	2 unused → go alone: (2)	4. (1, 3) (2)

Final Output

1. (1) (2) (3)
 2. (1) (2, 3)
 3. (1, 2) (3)
 4. (1, 3) (2)

Output:-

1.(1) (2) (3)
 2.(1) (2,3)
 3.(1,2) (3)
 4.(1,3) (2)

Goldmine2 in C++

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

int maxGold = 0;

void travel(vector<vector<int>>& arr, int i, int j,
vector<vector<bool>>& visited, vector<int>& bag) {
    if (i < 0 || j < 0 || i >= arr.size() || j >=
arr[0].size() || arr[i][j] == 0 || visited[i][j]) {
        return;
    }
    visited[i][j] = true;
    bag.push_back(arr[i][j]);
    travel(arr, i - 1, j, visited, bag);
    travel(arr, i, j + 1, visited, bag);
    travel(arr, i, j - 1, visited, bag);
    travel(arr, i + 1, j, visited, bag);
}

void getMaxGold(vector<vector<int>>& arr) {
    int rows = arr.size();
    int cols = arr[0].size();
    vector<vector<bool>> visited(rows,
vector<bool>(cols, false));

    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            if (arr[i][j] != 0 && !visited[i][j]) {
                vector<int> bag;
                travel(arr, i, j, visited, bag);

                int sum = 0;
                for (int val : bag) {
                    sum += val;
                }
                if (sum > maxGold) {
                    maxGold = sum;
                }
            }
        }
    }
}

int main() {
    vector<vector<int>> arr = {
        {0, 1, 4, 2, 8, 2},
        {4, 3, 6, 5, 0, 4},
        {1, 2, 4, 1, 4, 6},
        {2, 0, 7, 3, 2, 2},
        {3, 1, 5, 9, 2, 4},
        {2, 7, 0, 8, 5, 1}
    };
    getMaxGold(arr);
    cout << maxGold << endl;
    return 0;
}
```

Sample Grid (Visual):

[
{ 0, 1, 4, 2, 8, 2 },
{ 4, 3, 6, 5, 0, 4 },
{ 1, 2, 4, 1, 4, 6 },
{ 2, 0, 7, 3, 2, 2 },
{ 3, 1, 5, 9, 2, 4 },
{ 2, 7, 0, 8, 5, 1 }
]

We'll start traversal from (1,2) where value = 6

Dry Run Table (DFS Traversal Steps):

Step	Cell Visited	Gold at Cell	Cumulative Sum	Stack (DFS Recursion Path)
1	(1,2)	6	6	(1,2)
2	(0,2)	4	10	(1,2) → (0,2)
3	(0,3)	2	12	(1,2) → (0,2) → (0,3)
4	(0,4)	8	20	...
5	(0,5)	2	22	...
6	(1,5)	4	26	...
7	(2,5)	6	32	...
8	(2,4)	4	36	...
9	(3,4)	2	38	...
10	(3,5)	2	40	...
11	(4,5)	4	44	...
12	(4,4)	2	46	...
13	(4,3)	9	55	...
14	(5,3)	8	63	...
15	(5,4)	5	68	...
16	(5,5)	1	69	...
17	(3,3)	3	72	...
18	(2,3)	1	73	...
19	(2,2)	4	77	...
20	(1,3)	5	82	...
21	(1,1)	3	85	...
22	(2,1)	2	87	...
23	(2,0)	1	88	...
24	(3,0)	2	90	...
25	(4,0)	3	93	...
26	(4,1)	1	94	...
27	(5,1)	7	101	...
28	(5,0)	2	103	...

29	(1,0)	4	107	...
30	(0,1)	1	108	...
31	(3,2)	7	115	...
32	(4,2)	5	120	...

✓ Result:

At the end of this traversal:

- All connected gold cells are visited
- Sum = **120**
- This is the **maximum** among all components

★ Final Output:

Output: 120

Output:-

120

Josephus in C++

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

int solution(int n, int k) {
    if (n == 1) {
        return 0;
    }
    int x = solution(n - 1, k);
    int y = (x + k) % n;
    return y;
}

int main() {
    int n = 4;
    int k = 2;
    cout << solution(n, k) << endl;
    return 0;
}
```

Dry Run Table for solution(4, 2)

We'll compute this step-by-step recursively:

Function Call	Value Returned	Explanation
solution(1, 2)	0	Base case: Only one person, return 0
solution(2, 2)	$(0 + 2) \% 2 = 0$	Last survivor in 2 people = 0
solution(3, 2)	$(0 + 2) \% 3 = 2$	Last survivor in 3 people = 2
solution(4, 2)	$(2 + 2) \% 4 = 0$	Last survivor in 4 people = 0

❖ Final Output:

0

Output:-
0

Largest after k swaps in C++

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

string max_str;

void findMaximum(string str, int k) {
    // Base case: When k swaps are used up
    if (k == 0) {
        return;
    }

    int n = str.length();

    // Find the maximum digit available for
    // current position
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            // If digit at position j is greater than
            // digit at position i, swap them
            if (str[j] > str[i]) {
                swap(str[i], str[j]);

                // Check if current string is larger
                // than previously found max
                if (str > max_str) {
                    max_str = str;
                }

                // Recur for k-1 swaps on the
                // modified string
                findMaximum(str, k - 1);

                // Backtrack: Swap again to revert
                // to original string
                swap(str[i], str[j]);
            }
        }
    }
}

int main() {
    string str = "1234567";
    int k = 4;

    // Initialize max_str with the original
    // string
    max_str = str;

    // Find the maximum number possible after
    // k swaps
    findMaximum(str, k);

    // Print the maximum number found
    cout << max_str << endl;

    return 0;
}
```

Explanation of the Algorithm:

- For every pair (i, j) where $i < j$, if $\text{str}[j] > \text{str}[i]$, swap i and j.
- After each swap, check if the new number is greater than the current max_str.
- Recurse with k - 1.
- Backtrack (swap back) to explore other options.

Call#	k	Swap Made (i↔j)	str After Swap	max_str Before	max_str After	Remarks
1	4	0↔6	7234561	1234567	✓ 7234561	New max
2	3	1↔5	7634521	7234561	✓ 7634521	New max
3	2	2↔4	7654321	7634521	✓ 7654321	Final max
4	1	No beneficial swap	-	7654321	7654321	Stop recursion
5	3	1↔4	7534261	7654321	✗	Not greater
6	3	1↔3	7435261	7654321	✗	Not greater
7	2	2↔3 (from 7435261)	7453261	7654321	✗	Still not better
...	-	Many paths explored

We only continue recursion when beneficial. As you can see, once 7654321 is reached, **no further recursion produces a better result**, so that becomes the final output.

▣ Final Output:

7654321

Output:-
7654321

Lexicographic order in C++

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

void dfs(int i, int n) {
    if (i > n) {
        return;
    }
    cout << i << endl;
    for (int j = 0; j < 10; j++) {
        dfs(10 * i + j, n);
    }
}

int main() {
    int n = 40;
    for (int i = 1; i <= 9; i++) {
        dfs(i, n);
    }
    return 0;
}
```

Lexicographical Order Output:

This simulates how numbers are sorted like dictionary words:

1, 10, 100, ..., 11, 12, ..., 2, 20,
21, ..., 3, 30, ..., 4, 40, ...
5, ..., 9

But only numbers ≤ 40 are printed.

Dry Run Table (Partial for Clarity)

Here's a step-by-step snapshot of what's happening:

Function Call	i	Output	Explanation
dfs(1, 40)	1	✓ 1	Valid, print
dfs(10, 40)	10	✓ 10	Valid, print
dfs(100, 40)	>40	✗	Stop recursion
dfs(11, 40)	11	✓ 11	Continue same way
dfs(12, 40)	12	✓ 12	...
...	
dfs(19, 40)	19	✓ 19	
dfs(2, 40)	2	✓ 2	Start from next i
dfs(20, 40)	20	✓ 20	
dfs(21, 40)	21	✓ 21	
...	
dfs(40, 40)	40	✓ 40	Final valid number
dfs(41, 40)	>40	✗	Stop here

Output:-

```
1
10
11
12
13
14
15
16
17
18
```

19
2
20
3
4
5
6
7
8
9

Partition in K subsets in C++

```
#include <iostream>
#include <vector>

using namespace std;

int counter = 0;

void solution(int i, int n, int k, int nos,
vector<vector<int>>& ans) {
    if (i > n) {
        if (nos == k) {
            counter++;
            cout << counter << ". ";
            for (auto& set : ans) {
                cout << "[";
                for (auto num : set) {
                    cout << num << " ";
                }
                cout << "] ";
            }
            cout << endl;
        }
        return;
    }

    for (int j = 0; j < ans.size(); j++) {
        if (!ans[j].empty()) {
            ans[j].push_back(i);
            solution(i + 1, n, k, nos, ans);
            ans[j].pop_back();
        } else {
            ans[j].push_back(i);
            solution(i + 1, n, k, nos + 1, ans);
            ans[j].pop_back();
            break;
        }
    }
}

int main() {
    int n = 3;
    int k = 2;
    vector<vector<int>> ans(k);

    solution(1, n, k, 0, ans);

    return 0;
}
```

Dry Run Table:

Step	i	nos	ans (state)	Action Taken
1	1	0	[], []	Put 1 in first empty subset
2	2	1	[[1], []]	Put 2 in subset 0
3	3	1	[[1, 2], []]	Put 3 in subset 0
4	4	1	—	nos != k, discard
5	3	2	[[1, 2], [3]]	✓ Output: [1 2] [3]
6	2	2	[[1], [2]]	✓ Output path starts
7	3	2	[[1, 3], [2]]	✓ Output: [1 3] [2]
8	3	2	[[1], [2, 3]]	✓ Output: [1] [2 3]

Final Output:

1. [1 2] [3]
2. [1 3] [2]
3. [1] [2 3]

Output:-

1. [1 2] [3]
2. [1 3] [2]
3. [1] [2 3]

Permutation in C++

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

void permutations(int cb, int nboxes, int items[], int
ssf, int ritems, string asf) {
    if (cb > nboxes) {
        if (ssf == ritems) {
            cout << asf << endl;
        }
        return;
    }

    for (int i = 0; i < ritems; i++) {
        if (items[i] == 0) {
            items[i] = 1;
            permutations(cb + 1, nboxes, items, ssf + 1,
            ritems, asf + to_string(i + 1));
            items[i] = 0;
        }
    }

    permutations(cb + 1, nboxes, items, ssf, ritems, asf
    + "0");
}

int main() {
    int nboxes = 3;
    int ritems = 2;
    int cb = 1;
    int ssf = 0;
    int items[ritems] = {0}; // Initialize items array with
0s

    permutations(cb, nboxes, items, ssf, ritems, "");

    return 0;
}
```

Key Variables:

Var	Meaning
cb	current box index
ssf	selected so far – number of items placed
items[]	array of 0/1, indicating whether each item (1 to ritems) is used
asf	answer so far – the configuration of items across boxes

Dry Run Table:

cb	items	ssf	asf	Description
1	[0,0]	0	""	Start, box 1
2	[1,0]	1	"1"	Place item 1 in box 1
3	[1,1]	2	"12"	Place item 2 in box 2
4	[1,1]	2	"120"	↗ Output: item1 in box1, item2 in box2
3	[1,1]	2	"102"	item2 in box3
3	[1,0]	1	"10"	skip box 2
4	[1,1]	2	"102"	↗ Output
2	[0,1]	1	"2"	item2 in box 1
3	[1,1]	2	"21"	item1 in box2
4	[1,1]	2	"210"	↗ Output
3	[0,1]	1	"20"	box2 empty
4	[1,1]	2	"201"	↗ Output
2	[0,0]	0	"0"	box1 empty
3	[1,0]	1	"01"	item1 in box2
4	[1,1]	2	"012"	↗ Output
3	[0,1]	1	"02"	item2 in box2
4	[1,1]	2	"021"	↗ Output
3	[0,0]	0	"00"	box2 empty
4	-	0	"000"	✗ Not valid – ssf < ritems

Output:-

102
210
201
012
021

Permutation of string in C++

```
#include <iostream>
#include <unordered_map>
using namespace std;

void generate(int cs, int ts, unordered_map<char, int>& fmap, string asf) {
    if (cs > ts) {
        cout << asf << endl;
        return;
    }

    for (auto entry : fmap) {
        char ch = entry.first;
        int count = entry.second;

        if (count > 0) {
            fmap[ch]--;
            generate(cs + 1, ts, fmap, asf + ch);
            fmap[ch]++;
        }
    }
}

int main() {
    string str = "abc";
    unordered_map<char, int> fmap;

    for (char ch : str) {
        fmap[ch]++;
    }

    generate(1, str.length(), fmap, "");

    return 0;
}
```

Goal:

Generate **all permutations of "abc"** using recursion and a frequency map.

🔧 Setup:

- fmap: { a:1, b:1, c:1 }
- ts = total size = 3
- cs = current size (starts from 1)
- asf = answer so far

📋 Dry Run Table

Call Stack	fmap (a,b,c)	asf	cs	Output?
generate(1, 3, {1,1,1}, "")				
└ a → generate(2, 3, {0,1,1}, "a")	"a"	2		
└ b → generate(3, 3, {0,0,1}, "ab")	"ab"	3		
└ c → generate(4, 3, {0,0,0}, "abc")	"abc"	4		↙ Print
└ c → backtrack to "ab"				
└ c → generate(3, 3, {0,1,0}, "ac")	"ac"	3		
└ b → generate(4, 3, {0,0,0}, "acb")	"acb"	4		↙ Print
└ b → backtrack to "a"				
└ b → generate(2, 3, {1,0,1}, "b")	"b"	2		
└ a → generate(3, 3, {0,0,1}, "ba")	"ba"	3		
└ c → generate(4, 3, {0,0,0}, "bac")	"bac"	4		↙ Print
└ c → generate(3, 3, {1,0,0}, "bc")	"bc"	3		
└ a → generate(4, 3, {0,0,0}, "bca")	"bca"	4		↙ Print
└ c → generate(2, 3, {1,1,0}, "c")	"c"	2		
└ a → generate(3, 3, {0,1,0}, "ca")	"ca"	3		
└ b → generate(4, 3, {0,0,0}, "cab")	"cab"	4		↙ Print
└ b → generate(3, 3, {1,0,0}, "cb")	"cb"	3		
└ a → generate(4, 3, {0,0,0}, "cba")	"cba"	4		↙ Print

Output:-

cba
cab
bca
bac
acb
abc

Remove Invalid Parenthesis in C++

```
#include <iostream>
#include <string>
#include <unordered_set>
#include <stack>
using namespace std;

void solution(string str, int mra,
unordered_set<string>& ans);
int getMin(string str);

void solution(string str, int mra,
unordered_set<string>& ans) {
    if (mra == 0) {
        int mrnow = getMin(str);
        if (mrnow == 0) {
            if (ans.find(str) == ans.end()) {
                cout << str << endl;
                ans.insert(str);
            }
        }
        return;
    }
    for (int i = 0; i < str.length(); i++) {
        string left = str.substr(0, i);
        string right = str.substr(i + 1);
        solution(left + right, mra - 1, ans);
    }
}

int getMin(string str) {
    stack<char> st;
    for (int i = 0; i < str.length(); i++) {
        char ch = str[i];
        if (ch == '(') {
            st.push(ch);
        } else if (ch == ')') {
            if (st.empty()) {
                st.push(ch);
            } else if (st.top() == ')') {
                st.push(ch);
            } else if (st.top() == '(') {
                st.pop();
            }
        }
    }
    return st.size();
}

int main() {
    string str = "(((())";
    unordered_set<string> ans;
    int mra = getMin(str);
    solution(str, mra, ans);
    return 0;
}
```

Goal:

Remove the **minimum number** of parentheses to make the string valid.

Step 1: getMin("(((0))")

Step Char Stack Action

1	((push
2	(((push
3	((((push
4	((((push
5	((((push
6)	(((pop (match)
7)	(((pop (match)
8)	(((pop (match)

□ Final stack size = ((→ **2 unmatched**

✓ So mra = 2 (Minimum Removals Allowed)

Step 2: Recursive Dry Run Table

We'll track:

Call #	Current String (str)	Removals Left (mra)	Action Taken	Is Valid (getMin=0)?	Output
1	((((0)))	2	Start	X (getMin=2)	
2	((((0)))	1	Removed char at index 0	X (getMin=1)	
3	((0))	0	Removed char at index 0	✓ (getMin=0)	✓ ((0))
4	(same string)	0	Duplicate path	✓	(skipped by set)
...	many more paths tried	≤ 0	But not valid	X	

✓ Only ((0)) satisfies getMin == 0 with exactly 2 removals

	<p>✓ Your unordered_set prevents printing duplicates</p> <p>.</p> <p> Final Output:</p> <p>((0))</p>
--	--

Output:-

((0))

Subsequence in C++

```
#include <iostream>
#include <string>

using namespace std;

void sol(string q, string a) {
    if (q.length() == 0) {
        cout << a << "-" << endl;
        return;
    }

    char ch = q[0];
    string rest = q.substr(1);
    sol(rest, a);
    sol(rest, a + ch);
}

int main() {
    string s = "abc";
    sol(s, "");

    return 0;
}
```

Execution Tree:

We'll denote:

- q = remaining string
- a = answer so far

Call #	q	a	Output if base case
1	abc	""	
2	bc	""	
3	c	""	
4	""	""	-
5	""	"c"	c-
6	bc	"b"	
7	c	"b"	
8	""	"b"	b-
9	""	"bc"	bc-
10	abc	"a"	
11	bc	"a"	
12	c	"a"	
13	""	"a"	a-
14	""	"ac"	ac-
15	bc	"ab"	
16	c	"ab"	
17	""	"ab"	ab-
18	""	"abc"	abc-

❖ Final Output (Subsequences with -):

-
c-
b-
bc-
a-
ac-
ab-
abc-

Output:-

-
c-
b-
bc-
a-
ac-
ab-
abc-

Word Break in C++							
Dry Run Table:							
Call	str	ans	Lo op i	left = str.substr(0 , i+1)		left in dict ?	Action Taken
1	microsofthiring	""	0	m		✗	skip
1	microsofthiring	""	1	mi		✗	skip
1	microsofthiring	""	2	mic		✗	skip
1	microsofthiring	""	3	micr		✗	skip
1	microsofthiring	""	4	micro		✗	skip
1	microsofthiring	""	5	micros		✗	skip
1	microsofthiring	""	6	microso		✗	skip
1	microsofthiring	""	7	microsof		✗	skip
1	microsofthiring	""	8	microsoft		✓	Recurse with str=hiring, ans=microso ft
2	hiring	microsoft	0	h		✗	skip
2	hiring	microsoft	1	hi		✗	skip
2	hiring	microsoft	2	hir		✗	skip
2	hiring	microsoft	3	hiri		✗	skip
2	hiring	microsoft	4	hirin		✗	skip
2	hiring	microsoft	5	hiring		✓	Recurse with str="", ans=microso ft hiring
3	""	microsoft hiring	-	--			Print: microsoft hiring

Output:-	
microsoft hiring	

microsoft hiring

Check number exists in array in C++

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

int array11(int nums[], int index, int length) {
    if (index >= length) {
        return 0;
    }
    int small = array11(nums, index + 1, length);
    if (nums[index] == 11) {
        return 1 + small;
    } else {
        return small;
    }
}

int main() {
    int arr[] = {1, 11, 3, 11, 11, 11};
    int length = sizeof(arr) / sizeof(arr[0]);
    cout << array11(arr, 0, length) << endl;
    return 0;
}
```

Input

arr = {1, 11, 3, 11, 11, 11}

Function Call Tree

```
array11(arr, 0, 6)
→ nums[0] == 1 → skip
→ array11(arr, 1, 6)
→ nums[1] == 11 → count +1
→ array11(arr, 2, 6)
→ nums[2] == 3 → skip
→ array11(arr, 3, 6)
→ nums[3] == 11 → count +1
→ array11(arr, 4, 6)
→ nums[4] == 11 → count +1
→ array11(arr, 5, 6)
→ nums[5] == 11 → count +1
→ array11(arr, 6, 6)
→ index >= length → return 0
```

Dry Run Table

Call	index	nums[index]	Matches 11?	Return Value
array11(arr, 0, 6)	0	1	✗	0 + 4 = 4
array11(arr, 1, 6)	1	11	✓	1 + 3 = 4
array11(arr, 2, 6)	2	3	✗	0 + 3 = 3
array11(arr, 3, 6)	3	11	✓	1 + 2 = 3
array11(arr, 4, 6)	4	11	✓	1 + 1 = 2
array11(arr, 5, 6)	5	11	✓	1 + 0 = 1
array11(arr, 6, 6)	6	N/A	N/A	0

Output

4

Output:-

4

Check Palindrome in C++

```
#include <iostream>
#include <string>
using namespace std;

bool isStringPalindrome(const string& input, int s, int e) {
    // Base case: if start index equals end index, the string is a palindrome
    if (s == e) {
        return true;
    }
    // If the characters at the start and end do not match, it's not a palindrome
    if (input[s] != input[e]) {
        return false;
    }
    // If there are more characters to compare, call the function recursively
    if (s < e + 1) {
        return isStringPalindrome(input, s + 1, e - 1);
    }
    return true;
}

bool isStringPalindrome(const string& input) {
    int s = 0;
    int e = input.length() - 1;
    return isStringPalindrome(input, s, e);
}

int main() {
    cout <<
(isStringPalindrome("abba") ? "true" :
"false") << endl;
    return 0;
}
```

Input

string = "abba"

🔍 Function Call Tree

```
isStringPalindrome ("abba", 0, 3)
→ 'a' == 'a' ✓
→ isStringPalindrome ("abba", 1, 2)
    → 'b' == 'b' ✓
    → isStringPalindrome ("abba", 2, 1)
        → s > e → return true
```

💻 Dry Run Table

Call	s	e	input[s]	input[e]	Match?	Return
isStringPalindrome ("abba", 0, 3)	0	3	'a'	'a'	✓	✓
isStringPalindrome ("abba", 1, 2)	1	2	'b'	'b'	✓	✓
isStringPalindrome ("abba", 2, 1)	2	1	N/A	N/A	Base	✓

🌐 Output

true

Your program will print:

true

Output:-
true

Check sorted in C++

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

bool sorted(int arr[], int n) {
    if (n == 1 || n == 0) {
        return true;
    } else if (arr[n - 1] < arr[n - 2]) {
        return false;
    } else {
        return sorted(arr, n - 1);
    }
}

int main() {
    int arr[] = {1, 2, 3, 4, 5};
    int n = sizeof(arr) / sizeof(arr[0]);
    cout << boolalpha << sorted(arr, n) << endl;
    return 0;
}
```

Input

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

⌚ Recursive Calls

We check if the last two elements are in correct order ($\text{arr}[n-2] \leq \text{arr}[n-1]$), and recursively reduce the array size.

█████ Dry Run Table

Call	n	arr[n-2]	arr[n-1]	Comparison	Result
sorted(arr, 5)	5	4	5	$4 \leq 5$	✓
sorted(arr, 4)	4	3	4	$3 \leq 4$	✓
sorted(arr, 3)	3	2	3	$2 \leq 3$	✓
sorted(arr, 2)	2	1	2	$1 \leq 2$	✓
sorted(arr, 1)	1	—	—	Base case	✓

❖ Output

true

Your program will print:

true

Output:-
true

Count zeroes in C++

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

int cnt = 0;

int countZerosRec(int input) {
    // Base case for initial input of 0
    if (input == 0 && cnt == 0) {
        return 1;
    }

    // Base case for recursion
    if (input == 0) {
        return cnt;
    }

    // Check if the current last digit is zero
    if (input % 10 == 0) {
        cnt++;
    }

    // Recursive call to process the next digit
    return countZerosRec(input / 10);
}

int main() {
    cout << countZerosRec(10034) << endl;
    return 0;
}
```

Dry Run for countZerosRec(10034)

Call	input	input % 10	is zero?	sum
countZerosRec(10034)	10034	4	✗	0 + next
countZerosRec(1003)	1003	3	✗	0 + next
countZerosRec(100)	100	0	✓	1 + next
countZerosRec(10)	10	0	✓	1 + next
countZerosRec(1)	1	-	✗	0

→ Total = 1 + 1 = 2

Output:-
2

Factorial in C++				
Dry Run Table for fact(6):				
Call Level	n	Recursive Call	Returned Value	Computation
1	6	6 * fact(5)	720	6 * 120
2	5	5 * fact(4)	120	5 * 24
3	4	4 * fact(3)	24	4 * 6
4	3	3 * fact(2)	6	3 * 2
5	2	2 * fact(1)	2	2 * 1
6	1	1 * fact(0)	1	1 * 1
7 (Base)	0	return 1	1	Base case hit

↑ Final Output:

720

Output:-
720

Min-Max in C++

```
#include <iostream>
#include <climits> // for INT_MAX and INT_MIN
using namespace std;

int getMin(int arr[], int i, int n) {
    if (n == 1) {
        return arr[i];
    } else {
        return min(arr[i], getMin(arr, i + 1, n - 1));
    }
}

int getMax(int arr[], int i, int n) {
    if (n == 1) {
        return arr[i];
    } else {
        return max(arr[i], getMax(arr, i + 1, n - 1));
    }
}

int main() {
    int arr[] = {12, 8, 45, 67, 9};
    int n = sizeof(arr) / sizeof(arr[0]);
    cout << "Minimum element of array: " <<
getMin(arr, 0, n) << endl;
    cout << "Maximum element of array: " <<
getMax(arr, 0, n) << endl;
    return 0;
}
```

Dry Run Table for getMin(arr, 0, 5)

Call Level	i	arr[i]	Recursive Call	Returned Value	Computation
1	0	12	min(12, getMin(1, 4))	8	min(12, 8)
2	1	8	min(8, getMin(2, 3))	8	min(8, 9)
3	2	45	min(45, getMin(3, 2))	9	min(45, 9)
4	3	67	min(67, getMin(4, 1))	9	min(67, 9)
5 (base)	4	9	return arr[4]	9	Base case

Dry Run Table for getMax(arr, 0, 5)

Call Level	i	arr[i]	Recursive Call	Returned Value	Computation
1	0	12	max(12, getMax(1, 4))	67	max(12, 67)
2	1	8	max(8, getMax(2, 3))	67	max(8, 67)
3	2	45	max(45, getMax(3, 2))	67	max(45, 67)
4	3	67	max(67, getMax(4, 1))	67	max(67, 9)
5 (base)	4	9	return arr[4]	9	Base case

❖ Final Output:

Minimum element of array: 8
Maximum element of array: 67

Output:-

Minimum element of array: 8
Maximum element of array: 67

Stair Case in C++

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

// Function to calculate number of ways to reach nth
step
int staircase(int n) {
    // Base cases
    if (n == 0 || n == 1) {
        return 1;
    }
    if (n == 2) {
        return 2;
    }
    // Recursive case
    return staircase(n-1) + staircase(n-2) +
staircase(n-3);
}

int main() {
    // Test case
    int n = 7;
    cout << staircase(n) << endl;
    return 0;
}
```

Dry Run Table for `staircase(7)`

Track the **calls** and their **return values** from the bottom up (memoized-style for understanding):

n	<code>staircase(n)</code> Calculation	Result
0	1 (base case)	1
1	1 (base case)	1
2	2 (base case)	2
3	staircase(2) + staircase(1) + staircase(0)	$2 + 1 + 1 = 4$
4	staircase(3) + staircase(2) + staircase(1)	$4 + 2 + 1 = 7$
5	staircase(4) + staircase(3) + staircase(2)	$7 + 4 + 2 = 13$
6	staircase(5) + staircase(4) + staircase(3)	$13 + 7 + 4 = 24$
7	staircase(6) + staircase(5) + staircase(4)	$24 + 13 + 7 = 44$

✓ Final Output:

44

Output:-
44

Subset Sum in C++

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

// Function to calculate subset sums recursively
void subsetSums(int arr[], int l, int r, int sum) {
    // Base case: if l exceeds r, print the current sum
    if (l > r) {
        cout << sum << " ";
        return;
    }

    // Recursive case: include current element arr[l] in
    // the subset sum
    subsetSums(arr, l + 1, r, sum + arr[l]);
}

int main() {
    // Initialize the array and its length
    int arr[] = {5, 4, 3, 5, 4};
    int n = sizeof(arr) / sizeof(arr[0]);

    // Call the function to calculate subset sums,
    // starting with l=0, r=n-1, and initial sum=0
    subsetSums(arr, 0, n - 1, 0);

    return 0;
}
```

Output:-
21

Input:

```
int arr[] = {5, 4, 3, 5, 4};
```

This adds:

```
5 + 4 + 3 + 5 + 4 = 21
```

And when $l > r$, it prints sum, which is 21.

Dry Run Table (for your input):

Step	l	r	sum	Action
1	0	4	0	sum = 0 + arr[0] = 5
2	1	4	5	sum = 5 + arr[1] = 9
3	2	4	9	sum = 9 + arr[2] = 12
4	3	4	12	sum = 12 + arr[3] = 17
5	4	4	17	sum = 17 + arr[4] = 21
6	5	4	21	$l > r$, print 21 and return

Final Output:

21

Tiling in C++		
Function Call	Returns	Reason
tilingways(4)	?	tilingways(3) + tilingways(2)
tilingways(3)	?	tilingways(2) + tilingways(1)
tilingways(2)	?	tilingways(1) + tilingways(0)
tilingways(1)	1	Base case
tilingways(0)	0 ✗	Wrong base case — it should be 1
tilingways(2)	$1 + 0 = 1$	
tilingways(1)	1	Base case
tilingways(3)	$1 + 1 = 2$	
tilingways(2)	1	Already computed
tilingways(4)	$2 + 1 = 3$ ✓	

Output:-

3

Height in C++

```
#include <iostream>
#include <vector>
#include <stack>

using namespace std;

// Node class definition
class Node {
public:
    int data;
    vector<Node*> children;

    Node(int val) {
        data = val;
    }
};

// Function to construct the tree from the given array
Node* construct(vector<int>& arr) {
    Node* root = nullptr;
    stack<Node*> st;

    for (int i = 0; i < arr.size(); ++i) {
        if (arr[i] == -1) {
            st.pop();
        } else {
            Node* t = new Node(arr[i]);

            if (!st.empty()) {
                st.top()->children.push_back(t);
            } else {
                root = t;
            }

            st.push(t);
        }
    }

    return root;
}

// Function to calculate the height of the tree
int height(Node* node) {
    if (node->children.empty()) {
        return 0;
    }

    int maxChildHeight = 0;
    for (Node* child : node->children) {
        int childHeight = height(child);
        if (childHeight > maxChildHeight) {
            maxChildHeight = childHeight;
        }
    }

    return maxChildHeight + 1;
}

// Main function
int main() {
    vector<int> arr = {10, 20, -1, 30, 50, -1, 60, -1, -1,

```

Input Array:

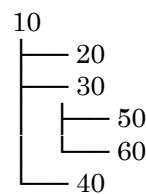
{10, 20, -1, 30, 50, -1, 60, -1, -1, 40, -1, -1}

❖ Tree Construction (construct function):

We use a **stack** to maintain the current path in the tree. When we encounter -1, we pop a node from the stack (finished with that node's children). Here's a **step-by-step construction** of the tree:

Step	arr[i]	Stack Top	Action	Tree Change
0	10	—	Create node(10), push	root = 10
1	20	10	Add 20 as child to 10, push	10 → 20
2	-1	20	Pop 20	
3	30	10	Add 30 as child to 10, push	10 → 30
4	50	30	Add 50 as child to 30, push	30 → 50
5	-1	50	Pop 50	
6	60	30	Add 60 as child to 30, push	30 → 60
7	-1	60	Pop 60	
8	-1	30	Pop 30	
9	40	10	Add 40 as child to 10, push	10 → 40
10	-1	40	Pop 40	
11	-1	10	Pop 10 (tree complete)	

◇ Final Tree:



▲ Height Calculation:

The **height** of a tree is the number of edges in the longest path from the root to a leaf node.

We traverse each subtree and compute the max height:

- Leaf nodes like 20, 50, 60, and 40 → height = 0
- Node 30 has children 50 and 60 → height = 1
- Root 10 has children:
 - 20 → 0
 - 30 → 1

```
40, -1, -1};  
  
Node* root = construct(arr);  
int h = height(root);  
cout << h << endl;  
  
return 0;  
}
```

2

- $40 \rightarrow 0$
→ max child height = 1
→ root height = $1 + 1 = 2$

✓ **Final Height: 2**

✓ **Output:**

2

Is Symmetric in C++						
Tree Structure from Input						
<pre>#include <iostream> #include <vector> #include <stack> using namespace std; // Node class definition class Node { public: int data; vector<Node*> children; Node(int val) { data = val; } }; // Function to construct the tree // from the given array Node* construct(vector<int>& arr) { Node* root = nullptr; stack<Node*> st; for (int i = 0; i < arr.size(); ++i) { if (arr[i] == -1) { st.pop(); } else { Node* t = new Node(arr[i]); if (!st.empty()) { st.top()->children.push_back(t); } else { root = t; } st.push(t); } } return root; } // Function to check if two trees // are mirrors of each other bool areMirror(Node* n1, Node* n2) { if (n1->children.size() != n2->children.size()) { return false; } for (int i = 0; i < n1->children.size(); ++i) { int j = n1->children.size() - 1 - i; Node* c1 = n1->children[i]; Node* c2 = n2->children[j]; } }</pre>						
<pre> graph TD 10 --- 20 10 --- 40 20 --- 50 20 --- 60 40 --- 100 40 --- 110 50 --- 30 30 --- 70 30 --- 80 70 --- 90 </pre>						
Step	node1->data	node2->data	Children Count Match	Comparing Child Pair	Recursive Call	Result
1	10	10	✓ Yes (3 children)	Compare 20 & 40	areMirror(20, 40)	proceeds
2	20	40	✓ Yes (2 children)	Compare 50 & 110	areMirror(50, 110)	✓ true
3	50	110	✓ Yes (0 children)	-	leaf nodes	✓ true
4	20	40	-	Compare 60 & 100	areMirror(60, 100)	✓ true
5	60	100	✓ Yes (0 children)	-	leaf nodes	✓ true
6	20 & 40	done	All children matched	-	return to previous	✓ true
7	10	10	-	Compare 30 & 30 (middle node)	areMirror(30, 30)	proceeds
8	30	30	✓ Yes (3 children)	Compare 70 & 90	areMirror(70, 90)	✓ true
9	70	90	✓ Yes (0)	-	leaf nodes	✓ true

```

if (!areMirror(c1, c2)) {
    return false;
}

return true;
}

// Function to check if a tree is
// symmetric
bool IsSymmetric(Node* node) {
    return areMirror(node, node);
}

// Main function
int main() {
    vector<int> arr = {10, 20, 50, -1,
60, -1, -1, 30, 70, -1, 80, -1, 90, -1,
-1, 40, 100, -1, 110, -1, -1, -1};

    Node* root = construct(arr);
    bool sym = IsSymmetric(root);
    cout << boolalpha << sym <<
endl;

    return 0;
}

```

				children)			
10	30	30	-	Compare 80 & 80	areMirror(80, 80)	↙ true	
11	80	80	↙ Yes (0 children)	-	leaf nodes	↙ true	
12	30	30	-	Compare 90 & 70	areMirror(90, 70)	↙ true	
13	90	70	↙ Yes (0 children)	-	leaf nodes	↙ true	
14	30 & 30	done	All children matched	-	return to previous	↙ true	
15	10	10	-	Compare 40 & 20	already compared in step 1	↙ true	
16	10 & 10	done	All pairs matched	-	final result	↙ true	

↙ Final Result:

true

true

Level Order in C++

```
#include <iostream>
#include <vector>
#include <queue>
#include <stack>

using namespace std;

// Node class definition
class Node {
public:
    int data;
    vector<Node*> children;

    Node(int val) {
        data = val;
    }
};

// Function to construct the tree from the given array
Node* construct(vector<int>& arr) {
    Node* root = nullptr;
    stack<Node*> st;

    for (int i = 0; i < arr.size(); ++i) {
        if (arr[i] == -1) {
            st.pop();
        } else {
            Node* t = new Node(arr[i]);

            if (!st.empty()) {
                st.top()->children.push_back(t);
            } else {
                root = t;
            }

            st.push(t);
        }
    }

    return root;
}

// Function for level order traversal
void levelOrder(Node* node) {
    if (!node)
        return;

    queue<Node*> q;
    q.push(node);

    while (!q.empty()) {
        Node* f = q.front();
        q.pop();

        cout << f->data << " ";

        for (Node* child : f->children) {
            q.push(child);
        }
    }
}
```

Input Array:
{24, 10, 20, 50, -1, 60, -1, -1, 30, 70, -1, 80, 110, -1, 120, -1, -1, 90, -1, -1, 40, 100, -1, -1, -1}

Tree Construction Process (construct() function):

Using a **stack**, we construct the tree as follows:

Step	arr[i]	Action	Stack Top (parent)	Node Created	Description
0	24	Create root, push to stack	—	24	Root node
1	10	Create, add to 24, push	24	10	24 → 10
2	20	Create, add to 10, push	10	20	10 → 20
3	50	Create, add to 20, push	20	50	20 → 50
4	-1	Pop 50	20	—	50 done
5	60	Create, add to 20, push	20	60	20 → 60
6	-1	Pop 60	20	—	60 done
7	-1	Pop 20	10	—	20 done
8	30	Create, add to 10, push	10	30	10 → 30
9	70	Create, add to 30, push	30	70	30 → 70
10	-1	Pop 70	30	—	70 done
11	80	Create, add to	30	80	30 → 80

```

    cout << "." << endl;
}

// Main function
int main() {
    vector<int> arr = {24, 10, 20, 50, -1, 60, -1, -1,
30, 70, -1, 80, 110, -1, 120, -1, -1, 90, -1, -1, 40,
100, -1, -1, -1};

    Node* root = construct(arr);
    levelOrder(root);

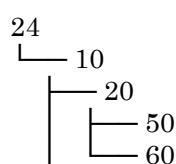
    return 0;
}

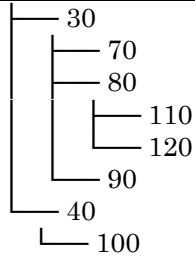
```

		30, push				
12	110	Create, add to 80, push	80	110	80 → 110	
13	-1	Pop 110	80	—	110 done	
14	120	Create, add to 80, push	80	120	80 → 120	
15	-1	Pop 120	80	—	120 done	
16	-1	Pop 80	30	—	80 done	
17	90	Create, add to 30, push	30	90	30 → 90	
18	-1	Pop 90	30	—	90 done	
19	-1	Pop 30	10	—	30 done	
20	40	Create, add to 10, push	10	40	10 → 40	
21	100	Create, add to 40, push	40	100	40 → 100	
22	-1	Pop 100	40	—	100 done	
23	-1	Pop 40	10	—	40 done	
24	-1	Pop 10	24	—	10 done	

◇ Final tree root is 24

◆ Tree Structure (for Visualization)





⌚ Level Order Traversal Output

Traverses level-by-level:

Queue Contents	Output
24	24
10	10
20, 30, 40	20
50, 60, 70, 80, 90, 100	30
—	40
—	50
—	60
—	70
110, 120	80
—	90
—	100
—	110
—	120

❖ Final Output:

24 10 20 30 40 50 60 70 80 90 100 110 120 .

24 10 20 30 40 50 60 70 80 90 100 110 120 .

PrePostorder Traversal in C++

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

// Node structure definition
struct Node {
    int data;
    vector<Node*> children;
};

// Function to display the tree structure
void display(Node* node) {
    cout << node->data << " -> ";
    for (Node* child : node->children) {
        cout << child->data << ", ";
    }
    cout << "." << endl;

    for (Node* child : node->children) {
        display(child);
    }
}

// Function to construct the tree from an array
Node* construct(vector<int>& arr) {
    Node* root = nullptr;
    vector<Node*> st;

    for (int i = 0; i < arr.size(); ++i) {
        if (arr[i] == -1) {
            st.pop_back();
        } else {
            Node* t = new Node();
            t->data = arr[i];

            if (!st.empty()) {
                st.back()->children.push_back(t);
            } else {
                root = t;
            }
        }

        st.push_back(t);
    }

    return root;
}

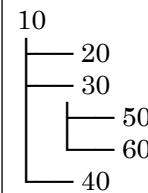
// Function to perform pre-order, post-order, and edge
// printing traversals
void traversals(Node* node) {
    // Print Node Pre
    cout << "Node Pre " << node->data << endl;

    // Print Edge Pre
    for (Node* child : node->children) {
        cout << "Edge Pre " << node->data << "--" <<
        child->data << endl;
        traversals(child);
        cout << "Edge Post " << node->data << "--" <<
    }
}
```

Input Array:

{10, 20, -1, 30, 50, -1, 60, -1, -1, 40, -1, -1}

↙ Constructed Tree:



↗ Dry Run Table for traversals()

Step	Current Node	Action Type	Output
1	10	Node Pre	Node Pre 10
2	10 → 20	Edge Pre	Edge Pre 10--20
3	20	Node Pre	Node Pre 20
4	20	Node Post	Node Post 20
5	10 ← 20	Edge Post	Edge Post 10--20
6	10 → 30	Edge Pre	Edge Pre 10--30
7	30	Node Pre	Node Pre 30
8	30 → 50	Edge Pre	Edge Pre 30--50
9	50	Node Pre	Node Pre 50
10	50	Node Post	Node Post 50
11	30 ← 50	Edge Post	Edge Post 30--50
12	30 → 60	Edge Pre	Edge Pre 30--60
13	60	Node Pre	Node Pre 60
14	60	Node Post	Node Post 60
15	30 ← 60	Edge Post	Edge Post 30--60
16	30	Node Post	Node Post 30
17	10 ← 30	Edge Post	Edge Post 10--30
18	10 → 40	Edge Pre	Edge Pre 10--40
19	40	Node Pre	Node Pre 40
20	40	Node Post	Node Post 40
21	10 ← 40	Edge Post	Edge Post 10--40

```

child->data << endl;
}

// Print Node Post
cout << "Node Post " << node->data << endl;
}

int main() {
    vector<int> arr = {10, 20, -1, 30, 50, -1, 60, -1, -1,
40, -1, -1};

    Node* root = construct(arr);

    // Perform pre-order, post-order, and edge printing
traversals
traversals(root);

    // Clean up memory (not necessary in this simple
example but good practice)
    // You would typically have a function to delete the
tree
    return 0;
}

```

			-40
22	10	Node Post	Node Post 10

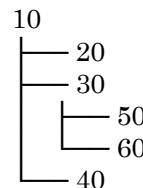
Final Output (as it would appear on console):

Node Pre 10
Edge Pre 10--20
Node Pre 20
Node Post 20
Edge Post 10--20
Edge Pre 10--30
Node Pre 30
Edge Pre 30--50
Node Pre 50
Node Post 50
Edge Post 30--50
Edge Pre 30--60
Node Pre 60
Node Post 60
Edge Post 30--60
Node Post 30
Edge Post 10--30
Edge Pre 10--40
Node Pre 40
Node Post 40
Edge Post 10--40
Node Post 10

Node Pre 10
Edge Pre 10--20
Node Pre 20
Node Post 20
Edge Post 10--20
Edge Pre 10--30
Node Pre 30
Edge Pre 30--50
Node Pre 50
Node Post 50
Edge Post 30--50
Edge Pre 30--60
Node Pre 60
Node Post 60
Edge Post 30--60
Node Post 30
Edge Post 10--30
Edge Pre 10--40
Node Pre 40
Node Post 40
Edge Post 10--40
Node Post 10

Size in C++				
<pre>#include <iostream> #include <vector> using namespace std; // Node structure definition struct Node { int data; vector<Node*> children; }; // Function to display the tree structure void display(Node* node) { cout << node->data << " -> "; for (Node* child : node->children) { cout << child->data << ", "; } cout << "." << endl; } // Function to construct the tree from array representation Node* construct(int arr[], int n) { Node* root = nullptr; vector<Node*> st; for (int i = 0; i < n; ++i) { if (arr[i] == -1) { st.pop_back(); } else { Node* t = new Node(); t->data = arr[i]; if (!st.empty()) { st.back()->children.push_back(t); } else { root = t; } st.push_back(t); } } return root; } // Function to calculate the size of the tree int size(Node* node) { int sz = 0; for (Node* child : node->children) { sz += size(child); } return 1 + sz; } int main() { // Static data representing the tree int arr[] = {10, 20, -1, 30, 50, -1, 60, -1, -1, 40, -1, -1}; </pre>				
✖ Tree Construction Dry Run				
<p>This array uses -1 to indicate the end of children for a node. We construct the tree using a vector (acting like a stack).</p>				
Step	arr[i]	Stack Top	Action	Tree Changes
0	10	—	New Node(10), push	root = 10
1	20	10	Add 20 as child to 10, push	10 → 20
2	-1	20	Pop 20	
3	30	10	Add 30 as child to 10, push	10 → 30
4	50	30	Add 50 as child to 30, push	30 → 50
5	-1	50	Pop 50	
6	60	30	Add 60 as child to 30, push	30 → 60
7	-1	60	Pop 60	
8	-1	30	Pop 30	
9	40	10	Add 40 as child to 10, push	10 → 40
10	-1	40	Pop 40	
11	-1	10	Pop 10	Done

◇ Tree Structure:



Let's apply it:

- $\text{size}(20) = 1$
- $\text{size}(50) = 1$
- $\text{size}(60) = 1$
- $\text{size}(30) = 1 (\text{self}) + \text{size}(50) + \text{size}(60) = 3$

```

int n = sizeof(arr) / sizeof(arr[0]);

// Construct the tree
Node* root = construct(arr, n);

// Calculate the size of the tree
int sz = size(root);
cout << sz << endl; // Output should be 6

// Display the tree structure (optional)
// display(root);

return 0;
}

```

- $1 + 1 + 1 = 3$
- $\text{size}(40) = 1$
- $\text{size}(10) = 1 \text{ (self)} + \text{size}(20) + \text{size}(30) + \text{size}(40) = 1 + 1 + 3 + 1 = \mathbf{6}$

6

LCA in C++

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

// Definition of a binary tree node
struct Node {
    int data;
    Node *left, *right;

    Node(int item) {
        data = item;
        left = nullptr;
        right = nullptr;
    }
};

// Function to find the Lowest Common Ancestor
// (LCA) of two nodes
Node* getLCA(Node* root, int a, int b) {
    if (root == nullptr) {
        return nullptr;
    }
    if (root->data == a || root->data == b) {
        return root;
    }

    Node* lca1 = getLCA(root->left, a, b);
    Node* lca2 = getLCA(root->right, a, b);

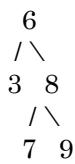
    if (lca1 != nullptr && lca2 != nullptr) {
        return root;
    }
    if (lca1 != nullptr) {
        return lca1;
    } else {
        return lca2;
    }
}

// Function to create a binary tree and find LCA
int main() {
    // Hardcoded tree construction
    Node* root = new Node(6);
    root->left = new Node(3);
    root->right = new Node(8);
    root->right->left = new Node(7);
    root->right->right = new Node(9);

    // Find LCA of nodes 3 and 7
    Node* lcaNode = getLCA(root, 3, 7);
    cout << "Lowest Common Ancestor of 3 and 7 is: "
    << lcaNode->data << endl;

    // Clean up dynamically allocated memory
    delete root->right->right;
    delete root->right->left;
    delete root->left;
    delete root;
    return 0;
}
```

Tree Structure:



You're finding the **LCA of 3 and 7**.

Q Dry Run of getLCA(root, 3, 7):

Function Call	Returns	Reason
getLCA(6, 3, 7)	→ 6	Found 3 in left subtree, 7 in right subtree → current is LCA
getLCA(3, 3, 7)	→ 3	root->data == a (found node 3)
getLCA(8, 3, 7)	→ 7	found 7 in left subtree, right subtree (9) doesn't contain target
getLCA(7, 3, 7)	→ 7	root->data == b (found node 7)
getLCA(9, 3, 7)	→ nullptr	no match

✓ Output:

Lowest Common Ancestor of 3 and 7 is: 6

Lowest Common Ancestor of 3 and 7 is: 6

Node at distance K in C++

```
#include <iostream>
#include <queue>
using namespace std;

// Definition of a binary tree node
struct Node {
    int data;
    Node* left;
    Node* right;

    Node(int item) {
        data = item;
        left = nullptr;
        right = nullptr;
    }
};

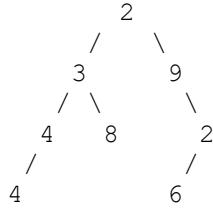
// Function declaration
void printNodesDown(Node* root, int k);

// Function to print nodes at distance k from the given node
int nodesAtDistanceKWithRootDistance(Node* root, int node, int k) {
    if (root == nullptr) {
        return -1;
    }

    // If the current node is the target node, print nodes at distance k from it
    if (root->data == node) {
        printNodesDown(root, k);
        return 0;
    }

    // Recursively search in left subtree
    int leftHeight =
        nodesAtDistanceKWithRootDistance(root->left,
                                         node, k);
    if (leftHeight != -1) {
        // If the target node is found in the left subtree
        if (leftHeight + 1 == k) {
            cout << root->data
            << endl;
        } else {
            // Print nodes at distance k from the right subtree
        }
    }
}
```

Binary Tree Structure:



🧠 Objective:

Print all nodes that are **exactly $k=2$ distance** away from node with value 3.

📝 Dry Run Table:

Step	Function Call	Current Node	Action	Output	Return Value
1	nodesAtDistanceK(root=2, node=3, k=2)	2	Call nodesAtDistanceKWithRootDistance		
2	nodesAtDistanceKWithRootDistance(root=2, node=3, k=2)	2	Not target → search left and right		
3	nodesAtDistanceKWithRootDistance(root=3, node=3, k=2)	3	↗ Target found! Call printNodesDown(3, 2)		0
4	printNodesDown(root=3, k=2)	3	Go down to distance 2		
5	printNodesDown(root=4, k=1)	4	Recurse to left → node 4		
6	printNodesDown(root=4, k=0)	4 (leaf)	✓ Distance 0 → print 4	4	
7	printNodesDown(root=8, k=1)	8	No children		
8	Back to step 2, leftHeight = 0		Check if root (2) is at k=2? No → Call printNodesDown(right, k-2)		
9	printNodesDown(root=9, k=0)	9	✓ Distance 0 → print 9	9	
10	All done		Final output = 4, 9		

```

printNodesDown(root->right, k - leftHeight - 2);
    }
    return leftHeight + 1;
}

// Recursively search in right subtree
int rightHeight =
nodesAtDistanceKWithRootDistance(root->right,
node, k);
if (rightHeight != -1) {
    // If the target node is found in the right subtree
    if (rightHeight + 1 == k) {
        cout << root->data
<< endl;
    } else {
        // Print nodes at distance k from the left subtree
        printNodesDown(root->left, k - rightHeight - 2);
    }
    return rightHeight + 1;
}

// If the target node is not found in either subtree
return -1;
}

// Function to print nodes at distance k from a given node downwards
void
printNodesDown(Node* root, int k) {
    if (root == nullptr || k < 0) {
        return;
    }

    // If reached the required distance, print the node
    if (k == 0) {
        cout << root->data << endl;
        return;
    }

    // Recursively print nodes at distance k in both subtrees
    printNodesDown(root->left, k - 1);
}

```

✓ Final Output:

4
9

```

        printNodesDown(root-
>right, k - 1);
    }

// Function to initiate
printing nodes at distance
k from a given node value
void
nodesAtDistanceK(Node*
root, int node, int k) {

    nodesAtDistanceKWithRo
otDistance(root, node, k);
}

int main() {
    // Hardcoded tree
construction
    Node* root = new
Node(2);
    root->left = new
Node(3);
    root->left->left = new
Node(4);
    root->left->right = new
Node(8);
    root->left->left->left =
new Node(4);
    root->right = new
Node(9);
    root->right->right =
new Node(2);
    root->right->right->left
= new Node(6);

    // Call function to print
nodes at distance k from
node with value 3
    nodesAtDistanceK(root,
3, 2);

    // Clean up dynamically
allocated memory
    delete root->right-
>right->left;
    delete root->right-
>right;
    delete root->right;
    delete root->left->left-
>left;
    delete root->left->left;
    delete root->left->right;
    delete root->left;
    delete root;

    return 0;
}

```

Size,Sum,Max,Min,Height in C++

```
#include <iostream>
#include <algorithm>
#include <climits> // for std::max
using namespace std;

// Definition of a binary tree node
struct Node {
    int data;
    Node* left;
    Node* right;

    Node(int data, Node* left = nullptr, Node* right = nullptr) {
        this->data = data;
        this->left = left;
        this->right = right;
    }
};

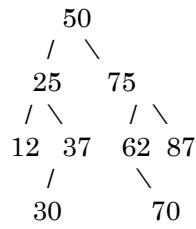
// Function to calculate the size (number of nodes) of
// the binary tree
int size(Node* node) {
    if (node == nullptr) {
        return 0;
    } else {
        return 1 + size(node->left) + size(node->right);
    }
}

// Function to calculate the sum of all nodes in the
// binary tree
int sum(Node* node) {
    if (node == nullptr) {
        return 0;
    } else {
        int lsum = sum(node->left);
        int rsum = sum(node->right);
        return node->data + lsum + rsum;
    }
}

// Function to find the maximum value in the binary
// tree
int max(Node* node) {
    if (node == nullptr) {
        return INT_MIN; // from <climits> for INT_MIN
    } else {
        int lmax = max(node->left);
        int rmax = max(node->right);
        return std::max(node->data, std::max(lmax,
rmax));
    }
}

// Function to calculate the height of the binary tree
int height(Node* node) {
    if (node == nullptr) {
        return -1;
    } else {
        int lh = height(node->left);
```

Binary Tree Structure:



❖ Expected Outputs:

Function	Description	Output
size	Number of nodes	9
sum	Sum of all node values	448
max	Maximum value in the tree	87
height	Height of the tree (edges, not nodes)	3
display	Inorder traversal (left → root → right)	12 25 30 37 50 62 70 75 87

✍ Let's go through function results step-by-step:

1. size(root):

- Total nodes = 9

2. sum(root):

$$\begin{aligned}
 &= 50 + \text{sum}(25 \text{ subtree}) + \text{sum}(75 \text{ subtree}) \\
 &= 50 + (25 + 12 + 37 + 30) + (75 + 62 + 70 + 87) \\
 &= 50 + 104 + 294 \\
 &= 448
 \end{aligned}$$

3. max(root):

- Max in left subtree = $\max(25, 12, 37, 30) = 37$
- Max in right subtree = $\max(75, 62, 70, 87) = 87$
- Final max = $\max(50, 37, 87) = 87$

4. height(root):

- Longest path (e.g., 50 → 75 → 62 → 70) has 3 edges → height = 3

5. display(root) (Inorder):

Left subtree (25): 12 25 30 37
Root: 50
Right subtree (75): 62 70 75 87
=> Full: 12 25 30 37 50 62 70 75 87

```

        int rh = height(node->right);
        return 1 + std::max(lh, rh);
    }

// Function to display the binary tree (inorder traversal)
void display(Node* node) {
    if (node == nullptr) {
        return;
    }

    display(node->left);
    cout << node->data << " ";
    display(node->right);
}

int main() {
    // Hardcoded tree construction
    Node* root = new Node(50);
    root->left = new Node(25);
    root->left->left = new Node(12);
    root->left->right = new Node(37);
    root->left->right->left = new Node(30);
    root->right = new Node(75);
    root->right->left = new Node(62);
    root->right->left->right = new Node(70);
    root->right->right = new Node(87);

    // Calculating size, sum, max value, and height
    int treeSize = size(root);
    int treeSum = sum(root);
    int treeMax = max(root);
    int treeHeight = height(root);

    // Displaying results
    cout << "Size of the binary tree: " << treeSize <<
    endl;
    cout << "Sum of all nodes in the binary tree: " <<
    treeSum << endl;
    cout << "Maximum value in the binary tree: " <<
    treeMax << endl;
    cout << "Height of the binary tree: " << treeHeight
    << endl;

    // Displaying the binary tree (inorder traversal)
    cout << "Inorder traversal of the binary tree:" <<
    endl;
    display(root);
    cout << endl;

    // Clean up dynamically allocated memory
    delete root->right->left->right;
    delete root->right->left;
    delete root->right;
    delete root->left->right->left;
    delete root->left->right;
    delete root->left->left;
    delete root->left;
    delete root;

    return 0;
}

```

▣ Final Output (Console):

Size of the binary tree: 9
 Sum of all nodes in the binary tree: 448
 Maximum value in the binary tree: 87
 Height of the binary tree: 3
 Inorder traversal of the binary tree:
 12 25 30 37 50 62 70 75 87

{	
size of the binary tree: 9	
Sum of all nodes in the binary tree: 448	
Maximum value in the binary tree: 87	
Height of the binary tree: 3	
Inorder traversal of the binary tree:	
12 25 30 37 50 62 70 75 87	

Tilt in C++

```
#include <iostream>
#include <cstdlib> // for abs function
using namespace std;

// Definition of a binary tree node
struct Node {
    int data;
    Node* left;
    Node* right;

    Node(int item) {
        data = item;
        left = nullptr;
        right = nullptr;
    }
};

// Function to display the binary tree (for debugging purposes)
void display(Node* node) {
    if (node == nullptr) {
        return;
    }

    string str = "";
    str += (node->left == nullptr) ? ":" : to_string(node->left->data);
    str += " <- " + to_string(node->data) + " -> ";
    str += (node->right == nullptr) ? ":" : to_string(node->right->data);
    cout << str << endl;

    display(node->left);
    display(node->right);
}

// Function to calculate the height of the binary tree
int height(Node* node) {
    if (node == nullptr) {
        return -1;
    }

    int lh = height(node->left);
    int rh = height(node->right);

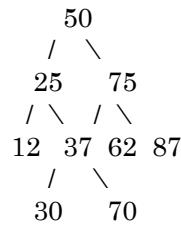
    return max(lh, rh) + 1;
}

// Global variable to store the tilt of the entire tree
int tilt = 0;

// Function to calculate the tilt of the binary tree
int calculateTilt(Node* node) {
    if (node == nullptr) {
        return 0;
    }

    int ls = calculateTilt(node->left);
    int rs = calculateTilt(node->right);
```

Tree Structure:



Dry Run with Tilt Values

Let's go **bottom-up** and calculate each node's tilt with its left and right subtree sums:

Node	Left Sum	Right Sum	Node Tilt = abs(L - R)
12	0	0	0
30	0	0	0
37	30	0	30
25	12	67 (37+30)	55
70	0	0	0
62	0	70	70
87	0	0	0
75	132	87	45
50	104	294	190

Total Tilt:

$$\begin{aligned}
 & 0 \text{ (12)} \\
 & + 0 \text{ (30)} \\
 & + 30 \text{ (37)} \\
 & + 55 \text{ (25)} \\
 & + 0 \text{ (70)} \\
 & + 70 \text{ (62)} \\
 & + 0 \text{ (87)} \\
 & + 45 \text{ (75)} \\
 & + 190 \text{ (50)} \\
 & = \text{***}390\text{***}
 \end{aligned}$$

Output:

Tilt of the binary tree: 390

```
int ltilt = abs(ls - rs);
tilt += ltilt;

int sum = ls + rs + node->data;
return sum;
}

int main() {
    // Hardcoded tree construction
    Node* root = new Node(50);
    root->left = new Node(25);
    root->left->left = new Node(12);
    root->left->right = new Node(37);
    root->left->right->left = new Node(30);
    root->right = new Node(75);
    root->right->left = new Node(62);
    root->right->left->right = new Node(70);
    root->right->right = new Node(87);

    // Calculate the tilt of the tree
    calculateTilt(root);

    // Output the tilt value
    cout << "Tilt of the binary tree: " << tilt << endl;

    // Clean up dynamically allocated memory
    delete root->left->left;
    delete root->left->right->left;
    delete root->left->right;
    delete root->left;
    delete root->right->left->right;
    delete root->right->left;
    delete root->right->right;
    delete root->right;
    delete root;

    return 0;
}
```

Tilt of the binary tree: 390

All single child parent in C++

```
#include <iostream>
#include <vector>

using namespace std;

// Definition of a Node in the Binary Tree
struct Node {
    int val;
    Node* left;
    Node* right;

    Node(int item) {
        val = item;
        left = nullptr;
        right = nullptr;
    }
};

// Function to find all nodes with exactly one child
void exactlyOneChild(Node* root, vector<int>& ans) {
    if (root == nullptr || (root->left == nullptr && root->right == nullptr)) {
        return;
    }

    if (root->left == nullptr || root->right == nullptr) {
        ans.push_back(root->val);
    }

    exactlyOneChild(root->left, ans);
    exactlyOneChild(root->right, ans);
}

// Wrapper function for exactlyOneChild
vector<int> exactlyOneChild(Node* root) {
    vector<int> res;
    exactlyOneChild(root, res);
    return res;
}

int main() {
    // Constructing the example binary tree
    Node* root = new Node(1);
    root->left = new Node(2);
    root->right = new Node(3);
    root->left->left = new Node(4);
    root->left->left->left = new Node(5);

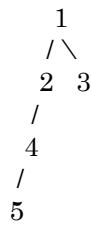
    // Finding nodes with exactly one child
    vector<int> ans = exactlyOneChild(root);

    // Printing the result
    cout << "Nodes with exactly one child: ";
    for (int num : ans) {
        cout << num << " ";
    }
    cout << endl;

    return 0;
}
```

Nodes with exactly one child: 2 4

Tree Structure:



Q Nodes with Exactly One Child

We traverse and look for nodes that have **only one** non-null child:

Node	Left Child	Right Child	Exactly One Child?	Added to ans?
1	2	3	✗ (has both)	✗
2	4	nullptr	✓	✓ → 2
4	5	nullptr	✓	✓ → 4
5	nullptr	nullptr	✗ (no children)	✗
3	nullptr	nullptr	✗ (no children)	✗

✓ Final Output:

Nodes with exactly one child: 2 4

BottomView in C++

```
#include <iostream>
#include <map>
#include <queue>
#include <vector>
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;
    }
};

vector<int> bottomView(TreeNode* root) {
    vector<int> bottomViewNodes;
    if (!root) {
        return bottomViewNodes;
    }

    // TreeMap equivalent in C++ is std::map
    map<int, int> map;
    queue<pair<TreeNode*, int>> q;
    q.push({root, 0});

    while (!q.empty()) {
        auto front = q.front();
        q.pop();
        TreeNode* node = front.first;
        int hd = front.second;

        // Update the map with current node's value at
        // its horizontal distance
        map[hd] = node->val;

        // Enqueue left child with horizontal distance hd - 1
        if (node->left) {
            q.push({node->left, hd - 1});
        }

        // Enqueue right child with horizontal distance hd + 1
        if (node->right) {
            q.push({node->right, hd + 1});
        }
    }

    // Populate bottomViewNodes with values from map
    for (const auto& pair : map) {
        bottomViewNodes.push_back(pair.second);
    }

    return bottomViewNodes;
}
```

Binary Tree Structure:



■ Step-by-Step Dry Run Table

We'll simulate the level order traversal using a queue storing (node, horizontal_distance) and map $hd \rightarrow \text{node} \rightarrow \text{val}$.

Step	Queue Content	Popped Node	HD	Map After Step
1	(1, 0)	1	0	{0 → 1}
2	(2, -1), (3, 1)	2	-1	{-1 → 2, 0 → 1}
3	(3, 1), (4, -2), (5, 0)	3	1	{-1 → 2, 0 → 1, 1 → 3}
4	(4, -2), (5, 0), (6, 0), (7, 2)	4	-2	{-2 → 4, -1 → 2, 0 → 1, 1 → 3}
5	(5, 0), (6, 0), (7, 2)	5	0	{-2 → 4, -1 → 2, 0 → 5, 1 → 3}
6	(6, 0), (7, 2)	6	0	{-2 → 4, -1 → 2, 0 → 6, 1 → 3}
7	(7, 2)	7	2	{-2 → 4, -1 → 2, 0 → 6, 1 → 3, 2 → 7}

● Final Bottom View:

Take values from the map in order of keys (i.e., horizontal distance):

```
-2 → 4
-1 → 2
0 → 6
1 → 3
2 → 7
```

▼ Output:

```
4 2 6 3 7
```

```
// Utility function to create a new node
TreeNode* newNode(int key) {
    TreeNode* node = new TreeNode(key);
    return node;
}

int main() {
    TreeNode* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->right->right = newNode(7);

    vector<int> result = bottomView(root);

    // Print the result
    for (int value : result) {
        cout << value << " ";
    }
    cout << endl;

    // Memory cleanup (optional in this example)
    // You may need to delete nodes if not using smart
    pointers
    return 0;
}
```

4 2 6 3 7

Diagonal Order in C++

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

// TreeNode structure definition
struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x) {
        val = x;
        left = nullptr;
        right = nullptr;
    }
};

// Function to perform diagonal order traversal of
// a binary tree
vector<vector<int>> diagonalOrder(TreeNode*
root) {
    vector<vector<int>> ans;
    if (root == nullptr) return ans;

    queue<TreeNode*> que;
    que.push(root);

    while (!que.empty()) {
        int size = que.size();
        std::vector<int> smallAns;

        while (size--) {
            TreeNode* node = que.front();
            que.pop();

            while (node != nullptr) {
                smallAns.push_back(node->val);

                if (node->left) que.push(node->left);
                node = node->right;
            }
        }

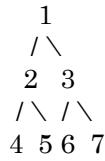
        ans.push_back(smallAns);
    }

    return ans;
}

int main() {
    // Constructing the binary 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->right->left = new TreeNode(6);
    root->right->right = new TreeNode(7);

    // Calling diagonalOrder function and printing
}
```

Tree Structure:



◆ Diagonal View Intuition:

- Diagonal lines go **from top-right to bottom-left**, i.e., every time you go to .right, you stay on the same diagonal.
- Every time you go to .left, you move to the **next diagonal**.

❖ Dry Run Table:

We'll simulate the queue and how the diagonal groups are formed.

Iteration	Queue (Before)	Extracted	Collected (Diagonal)	Queue (After pushing lefts)
1	[1]	1 → 3 → 7	[1, 3, 7]	[2, 6]
2	[2, 6]	2 → 5	[2, 5]	[4]
3	[4]	4	[4]	[]

◆ Final Output:

Diagonal Order Traversal:

```

1 3 7
2 5
4
  
```

❖ Breakdown:

- Diagonal 0 → 1 → 3 → 7
- Diagonal 1 → 2 → 5
- Diagonal 2 → 4

```
the result
vector<vector<int>> ans =
diagonalOrder(root);

cout << "Diagonal Order Traversal:\n";
for (const auto level : ans) {
    for (int num : level) {
        cout << num << " ";
    }
    cout << "\n";
}

// Deallocating memory to avoid memory leaks
delete root->right->right;
delete root->right->left;
delete root->left->right;
delete root->left->left;
delete root->right;
delete root->left;
delete root;

return 0;
}
```

Diagonal Order Traversal:

1 3 7
2 5 6
4

Iterative tree operations in C++

```
#include <iostream>
#include <queue>
#include <climits> // for INT_MIN and INT_MAX

using namespace std;

// Definition of a Node in the Binary Tree
struct Node {
    int val;
    Node* left;
    Node* right;

    Node(int x) {
        val = x;
        left = nullptr;
        right = nullptr;
    }
};

// Function to calculate the height of the tree using
// BFS (level-order traversal)
int getHeight(Node* root) {
    if (root == nullptr) return 0;

    queue<Node*> q;
    q.push(root);
    int height = 0;

    while (!q.empty()) {
        int levelSize = q.size();
        height++;
        for (int i = 0; i < levelSize; i++) {
            Node* node = q.front();
            q.pop();
            if (node->left != nullptr) q.push(node->left);
            if (node->right != nullptr) q.push(node->right);
        }
    }

    return height;
}

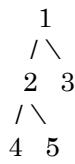
// Function to count the number of nodes in the tree
// using BFS (level-order traversal)
int getNodeCount(Node* root) {
    if (root == nullptr) return 0;

    queue<Node*> q;
    q.push(root);
    int count = 0;

    while (!q.empty()) {
        Node* node = q.front();
        q.pop();
        count++;
        if (node->left != nullptr) q.push(node->left);
        if (node->right != nullptr) q.push(node->right);
    }

    return count;
}
```

Tree Structure:



◆ Function: **getHeight(root)**

This uses **level-order traversal (BFS)**.

Level	Nodes at Level	Height So Far
1	1	1
2	2, 3	2
3	4, 5	3

◇ Result: 3

◆ Function: **getNodeCount(root)**

Counts nodes using BFS:

Step	Node Processed	Count	Queue
1	1	1	2, 3
2	2	2	3, 4, 5
3	3	3	4, 5
4	4	4	5
5	5	5	

◇ Result: 5

◆ Function: **getMax(root)**

Finds maximum using BFS:

Step	Node Processed	Max So Far
1	1	1
2	2	2
3	3	3
4	4	4
5	5	5 ◇

◇ Result: 5

◆ Function: **getMin(root)**

Finds minimum using BFS:

```

// Function to find the maximum value in the tree
using BFS (level-order traversal)
int getMax(Node* root) {
    if (root == nullptr) throw invalid_argument("Tree is
empty");

    queue<Node*> q;
    q.push(root);
    int maxValue = INT_MIN;

    while (!q.empty()) {
        Node* node = q.front();
        q.pop();
        maxValue = max(maxValue, node->val);
        if (node->left != nullptr) q.push(node->left);
        if (node->right != nullptr) q.push(node->right);
    }

    return maxValue;
}

// Function to find the minimum value in the tree
using BFS (level-order traversal)
int getMin(Node* root) {
    if (root == nullptr) throw invalid_argument("Tree is
empty");

    queue<Node*> q;
    q.push(root);
    int minValue = INT_MAX;

    while (!q.empty()) {
        Node* node = q.front();
        q.pop();
        minValue = min(minValue, node->val);
        if (node->left != nullptr) q.push(node->left);
        if (node->right != nullptr) q.push(node->right);
    }

    return minValue;
}

int main() {
    // Constructing the example binary tree
    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);

    // Using the functions to demonstrate the
    functionality
    cout << "Height of the tree: " << getHeight(root) <<
endl;
    cout << "Number of nodes in the tree: " <<
getNodeCount(root) << endl;

    try {
        cout << "Maximum value in the tree: " <<
getMax(root) << endl;
        cout << "Minimum value in the tree: " <<

```

Step	Node Processed	Min So Far
1	1	1 ✓
2	2	1
3	3	1
4	4	1
5	5	1

✓ Result: 1

✓ Final Output:

Height of the tree: 3
Number of nodes in the tree: 5
Maximum value in the tree: 5
Minimum value in the tree: 1

```
getMin(root) << endl;
} catch (const exception& e) {
    cerr << e.what() << endl;
}

return 0;
}
```

Height of the tree: 3
Number of nodes in the tree: 5
Maximum value in the tree: 5
Minimum value in the tree: 1

Top View in C++

```
#include <iostream>
#include <vector>
#include <queue>
#include <map>
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;
    }
};

// Function to compute the top view of a binary tree
vector<int> topView(TreeNode* root) {
    vector<int> topViewNodes;
    if (!root) {
        return topViewNodes;
    }

    map<int, int> hdMap; // Horizontal Distance Map
    (hd -> node value)
    queue<pair<TreeNode*, int>> q; // Queue to store
    nodes and their horizontal distance

    q.push({root, 0}); // Start with the root node at
    horizontal distance 0

    while (!q.empty()) {
        TreeNode* node = q.front().first;
        int hd = q.front().second;
        q.pop();

        // If this horizontal distance is not already in the
        map, add the node value
        if (hdMap.find(hd) == hdMap.end()) {
            hdMap[hd] = node->val;
        }

        // Enqueue left and right children with updated
        horizontal distances
        if (node->left) {
            q.push({node->left, hd - 1});
        }

        if (node->right) {
            q.push({node->right, hd + 1});
        }

        // Extract values from the map in order of
        horizontal distance
        for (const auto& pair : hdMap) {
            topViewNodes.push_back(pair.second);
        }
    }
}
```

Constructed Binary Tree:



■ Step-by-Step Traversal Table (Level Order with HD)

We'll perform a BFS traversal and track each node with its **Horizontal Distance (HD)** from root.

Step	Queue Content	Popped Node	HD	hdMap Before	hdMap After
1	(1, 0)	1	0	{}	{0: 1}
2	(2, -1), (3, 1)	2	-1	{0: 1}	{-1: 2, 0: 1}
3	(3, 1), (4, 0)	3	1	{-1: 2, 0: 1}	{-1: 2, 0: 1, 1: 3}
4	(4, 0), (5, 1)	4	0	already filled	(no change)
5	(5, 1), (6, 2)	5	1	already filled	(no change)
6	(6, 2)	6	2	{-1: 2, 0: 1, 1: 3}	{..., 2: 6}

● Final Map (hdMap) Sorted by HD:

$-1 \rightarrow 2$
 $0 \rightarrow 1$
 $1 \rightarrow 3$
 $2 \rightarrow 6$

❖ Output (Top View):

2 1 3 6

```
    return topViewNodes;
}

// Utility function to create a new node
TreeNode* newNode(int key) {
    TreeNode* node = new TreeNode(key);
    return node;
}

int main() {
    // Constructing the binary tree
    TreeNode* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->right = newNode(4);
    root->left->right->right = newNode(5);
    root->left->right->right->right = newNode(6);

    // Get the top view of the binary tree
    vector<int> result = topView(root);

    // Print the top view of the binary tree
    cout << "Top view of the binary tree:" << endl;
    for (intnodeValue : result) {
        cout << nodeValue << " ";
    }
    cout << endl;

    // Clean up memory (optional in this example)
    // You may need to delete nodes if not using smart
    pointers
    return 0;
}
```

Top view of the binary tree:
2 1 3 6

Balanced in C++

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

// Node structure for the binary tree
struct Node {
    int key;
    Node* left;
    Node* right;

    Node(int item) {
        key = item;
        left = right = nullptr;
    }
};

// Function to calculate the height of
// the tree and check balance
pair<bool, int>
isBalancedHelper(Node* root) {
    if (root == nullptr)
        return {true, 0};

    // Recursively get heights of left
    // and right subtrees
    auto left = isBalancedHelper(root->left);
    auto right =
isBalancedHelper(root->right);

    // If either subtree is unbalanced,
    // the whole tree is unbalanced
    if (!left.first || !right.first)
        return {false, -1};

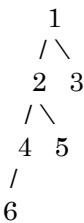
    // Check if the current subtree is
    // balanced
    if (abs(left.second - right.second) >
1)
        return {false, -1};

    // Return balanced status and
    // height of the current subtree
    return {true, max(left.second,
right.second) + 1};
}

// Function to check if the binary tree
// is balanced
bool isBalanced(Node* root) {
    return
isBalancedHelper(root).first;
}

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->left->left = new
}
```

Binary Tree Structure



Dry Run Table: isBalancedHelper

We'll do a **postorder traversal** (left → right → root) and track the balance and height of each subtree.

Node	Left Subtree (Balanced, Height)	Right Subtree (Balanced, Height)	Height Difference	Is Current Balanced?	Current Height
6	(true, 0)	(true, 0)	0	✓ Yes	1
4	(true, 1)	(true, 0)	1	✓ Yes	2
5	(true, 0)	(true, 0)	0	✓ Yes	1
2	(true, 2)	(true, 1)	1	✓ Yes	3
3	(true, 0)	(true, 0)	0	✓ Yes	1
1	(true, 3)	(true, 1)	2	✗ No	—

✗ Final Result:

- Node 1 is **not balanced** because its left and right subtrees have a height difference of **2**, which is more than 1.
- Hence, isBalanced(root) returns false.

✓ Output:

Is the tree balanced? No

```
Node(6);

    bool balanced = isBalanced(root);
    cout << "Is the tree balanced? " <<
(balanced ? "Yes" : "No") << endl;

    return 0;
}
```

Is the tree balanced? No

Binary Tree 2 LL in C++

```
#include <iostream>
using namespace std;
class Node {
public:
    int key;
    Node* left;
    Node* right;

    Node(int value) {
        key = value;
        left = nullptr;
        right = nullptr;
    }
};

class BinTree2LL {
private:
    static Node* prev;

public:
    static void flatten(Node* root) {
        if (root == nullptr) return;

        flatten(root->right);
        flatten(root->left);

        root->right = prev;
        root->left = nullptr;
        prev = root;
    }

    static void printList(Node* root) {
        while (root->right != nullptr) {
            cout << root->key << "->";
            root = root->right;
        }
        cout << root->key;
    }
};

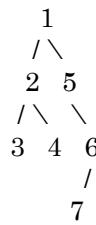
Node* BinTree2LL::prev = nullptr;

int main() {
    Node* root = new Node(1);
    root->left = new Node(2);
    root->left->left = new Node(3);
    root->left->right = new Node(4);
    root->right = new Node(5);
    root->right->right = new Node(6);
    root->right->right->left = new Node(7);

    BinTree2LL::flatten(root);
    BinTree2LL::printList(root);

    // Clean up allocated memory (not present in Java
    // version)
    while (root != nullptr) {
        Node* temp = root;
        root = root->right;
        delete temp;
    }
}
```

Original Binary Tree Structure



⌘ Flattening Logic: Reverse Postorder (Right → Left → Node)

The algorithm works like this:

- Traverse the tree in **reverse postorder**.
- Use a static prev pointer to keep track of the previously processed node.
- Set the current node's right to prev, and its left to nullptr.

█ Step-by-Step Tabular Dry Run

We will track:

- The current node being visited
- The state of prev
- Links updated

Step	Node Visited	Previous (prev)	Action	Updated Links
1	7	nullptr	Set 7.right = nullptr, 7.left = nullptr, prev = 7	7 → nullptr
2	6	7	Set 6.right = 7, 6.left = nullptr, prev = 6	6 → 7
3	5	6	Set 5.right = 6, 5.left = nullptr, prev = 5	5 → 6 → 7
4	4	5	Set 4.right = 5, 4.left = nullptr, prev = 4	4 → 5 → 6 → 7
5	3	4	Set 3.right = 4, 3.left = nullptr, prev = 3	3 → 4 → ...
6	2	3	Set 2.right = 3, 2.left = nullptr,	2 → 3 → ...

```
}
```

```
    return 0;
```

```
}
```

			prev = 2
7	1	2	Set 1.right = 2, 1.left = 1 → 2 → 3 nullptr, prev = 1

✓ Final Flattened Linked List (Right Pointers)

1 → 2 → 3 → 4 → 5 → 6 → 7

All left pointers are nullptr, forming a **single right-skewed list**.

❖ Output

1->2->3->4->5->6->7

1->2->3->4->5->6->7

Boundary traversal in C++

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

// Definition of the Node class
class Node {
public:
    int key;
    Node* left;
    Node* right;

    Node(int item) {
        key = item;
        left = right = nullptr;
    }
};

// Utility function to check if a node is a leaf node
bool isLeaf(Node* root) {
    return (root->left == nullptr && root->right == nullptr);
}

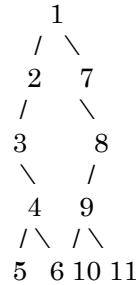
// Function to add nodes of the left boundary
// (excluding the leaf node itself)
void addLeftBoundary(Node* root, vector<int>& res) {
    Node* cur = root->left;
    while (cur != nullptr) {
        if (!isLeaf(cur))
            res.push_back(cur->key);
        if (cur->left != nullptr)
            cur = cur->left;
        else
            cur = cur->right;
    }
}

// Function to add nodes of the right boundary
// (excluding the leaf node itself)
void addRightBoundary(Node* root, vector<int>& res) {
    Node* cur = root->right;
    vector<int> tmp;
    while (cur != nullptr) {
        if (!isLeaf(cur))
            tmp.push_back(cur->key);
        if (cur->right != nullptr)
            cur = cur->right;
        else
            cur = cur->left;
    }
    for (int i = tmp.size() - 1; i >= 0; --i) {
        res.push_back(tmp[i]);
    }
}

// Function to add all leaf nodes in left-to-right
// order
```

Binary Tree Structure

Here's the tree again for reference:



◇ Step-by-Step Tabular Dry Run

1. █ Root Node

Step	Node Visited	Is Leaf?	Action	Vector State
1	1	No	Add to result	[1]

2. █ Left Boundary (excluding leaves)

Traversal path: 2 → 3 → 4 (stop before leaf nodes 5, 6)

Step	Node Visited	Is Leaf?	Action	Vector State
2	2	No	Add to result	[1, 2]
3	3	No	Add to result	[1, 2, 3]
4	4	No	Add to result	[1, 2, 3, 4]

3. █ Leaf Nodes (from left to right)

Leaf nodes: 5, 6, 10, 11

Step	Node Visited	Is Leaf?	Action	Vector State
5	5	Yes	Add to result	[1, 2, 3, 4, 5]
6	6	Yes	Add to result	[1, 2, 3, 4, 5, 6]
7	10	Yes	Add to	[1, 2, 3, 4, 5, 6,

```

void addLeaves(Node* root, vector<int>& res) {
    if (isLeaf(root)) {
        res.push_back(root->key);
        return;
    }
    if (root->left != nullptr)
        addLeaves(root->left, res);
    if (root->right != nullptr)
        addLeaves(root->right, res);
}

// Function to perform boundary traversal and
// return the result as vector
vector<int> printBoundary(Node* node) {
    vector<int> ans;
    if (!isLeaf(node))
        ans.push_back(node->key);
    addLeftBoundary(node, ans);
    addLeaves(node, ans);
    addRightBoundary(node, ans);
    return ans;
}

int main() {
    // Constructing the binary tree
    Node* root = new Node(1);
    root->left = new Node(2);
    root->left->left = new Node(3);
    root->left->left->right = new Node(4);
    root->left->left->right->left = new Node(5);
    root->left->left->right->right = new Node(6);
    root->right = new Node(7);
    root->right->right = new Node(8);
    root->right->right->left = new Node(9);
    root->right->right->left->left = new Node(10);
    root->right->right->left->right = new Node(11);

    // Performing boundary traversal
    vector<int> boundaryTraversal =
    printBoundary(root);

    // Printing the result
    cout << "The Boundary Traversal is : ";
    for (int i = 0; i < boundaryTraversal.size(); i++)
    {
        cout << boundaryTraversal[i] << " ";
    }
    cout << endl;

    return 0;
}

```

Boundary Traversal: [1, 2, 3, 4, 5, 6, 10, 11, 9, 8, 7]

Step	Node Visited	Is Leaf?	Action	Vector State
			result	[10]
8	11	Yes	Add to result	[1, 2, 3, 4, 5, 6, 10, 11]

4. Right Boundary (excluding leaves) — reversed

Traversal path: 7 → 8 → 9 (reverse order, ignore 10 and 11)

Step	Node Visited	Is Leaf?	Action (store in temp, then reverse)	Temporary Stack	Vector State (after reverse append)
9	7	No	Push to temp	[7]	
10	8	No	Push to temp	[7, 8]	
11	9	No	Push to temp	[7, 8, 9]	
12	--	--	Reverse and append to result		[1, 2, 3, 4, 5, 6, 10, 11, 9, 8, 7]

☞ Final Result

Boundary Traversal: [1, 2, 3, 4, 5, 6, 10, 11, 9, 8, 7]

Children Sum in C++

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

// Definition of the Node class
class Node {
public:
    int key;
    Node* left;
    Node* right;

    Node(int item) {
        key = item;
        left = right = nullptr;
    }
};

// Function to reorder the binary tree based on
// Children Sum Property
void reorder(Node* root) {
    if (root == nullptr) return;

    int child = 0;
    if (root->left != nullptr) {
        child += root->left->key;
    }
    if (root->right != nullptr) {
        child += root->right->key;
    }

    if (child < root->key) {
        if (root->left != nullptr) root->left->key = root->key;
        else if (root->right != nullptr) root->right->key = root->key;
    }

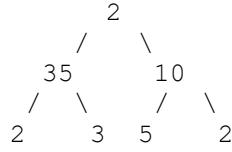
    reorder(root->left);
    reorder(root->right);

    int tot = 0;
    if (root->left != nullptr) tot += root->left->key;
    if (root->right != nullptr) tot += root->right->key;
    if (root->left != nullptr || root->right != nullptr)
        root->key = tot;
    }
}

// Function to change the tree based on Children Sum
// Property
void changeTree(Node* root) {
    reorder(root);
}

int main() {
    Node* root = new Node(2);
    root->left = new Node(35);
    root->left->left = new Node(2);
    root->left->right = new Node(3);
    root->right = new Node(10);
    root->right->left = new Node(5);
    root->right->right = new Node(2);
}
```

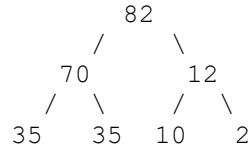
Initial Tree Structure



⌚ Dry Run: Step-by-Step Execution

Node Visited	Children Before	Action Taken	Node Key After
2 (root)	$35 + 10 = 45$	Children $>$ root → No update to children	—
35	$2 + 3 = 5$	Children $<$ 35 → Set both children to 35	—
2 (left)	null	Leaf node	35
3 (right)	null	Leaf node	35
Back to 35	$35 + 35 = 70$	Set node key = sum of children	70
10	$5 + 2 = 7$	Children $<$ 10 → Set left to 10 (since left exists)	—
5 (left)	null	Leaf node	10
2 (right)	null	Leaf node	2
Back to 10	$10 + 2 = 12$	Set node key = sum of children	12
Back to root	$70 + 12 = 82$	Set root = sum of its updated children	82

♣ Final Tree Structure



❖ Output

Modified Tree:

Root: 82
 Left: 70, Left Left: 35, Left Right: 35
 Right: 12, Right Left: 10, Right Right:
 2

```

changeTree(root);

// Display the modified tree
cout << "Modified Tree:" << endl;
cout << "Root: " << root->key << endl;
cout << "Left: " << root->left->key << ", Left Left: "
<< root->left->left->key << ", Left Right: " << root-
>left->right->key << endl;
    cout << "Right: " << root->right->key << ", Right
Left: " << root->right->left->key << ", Right Right: "
<< root->right->right->key << endl;

return 0;
}

```

Summary of Key Logic in `reorder()`:

1. Preorder Phase:

- Push parent's value down to children if sum of children < parent.

2. Postorder Phase:

- After children updated, update parent's value as sum of updated children.

Modified Tree:

Root: 50

Left: 38, Left Left: 35, Left Right: 3

Right: 12, Right Left: 10, Right Right: 2

Diameter in C++

```
#include <iostream>
#include <algorithm> // For std::max
using namespace std;

// Definition of the Node class
class Node {
public:
    int key;
    Node* left;
    Node* right;

    Node(int item) {
        key = item;
        left = right = nullptr;
    }
};

// Function prototype for height
int height(Node* node, int* diameter);

// Function to calculate diameter of binary tree
int diameterOfBinaryTree(Node* root) {
    int diameter = 0;
    height(root, &diameter);
    return diameter;
}

// Helper function to calculate height and
// update diameter
int height(Node* node, int* diameter) {
    if (node == nullptr) {
        return 0;
    }

    int leftHeight = height(node->left,
                           diameter);
    int rightHeight = height(node->right,
                           diameter);

    *diameter = max(*diameter, leftHeight +
                    rightHeight);

    return 1 + max(leftHeight, rightHeight);
}

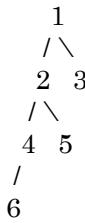
int main() {
    Node* root = new Node(1);
    root->left = new Node(2);
    root->right = new Node(3);
    root->left->left = new Node(4);
    root->left->right = new Node(5);
    root->left->left->left = new Node(6);

    int dia = diameterOfBinaryTree(root);
    cout << "Diameter of the binary tree: " <<
dia << endl;

    return 0;
}
```

Tree Structure

Based on your construction, the tree looks like this:



💡 What Is Diameter?

The **diameter** is the **length of the longest path** between any two nodes in the tree (measured by number of edges, not nodes).

This path **does not necessarily pass through the root**.

🧠 Core Logic Summary

- For each node:
 - Compute leftHeight and rightHeight.
 - Update diameter = max(diameter, leftHeight + rightHeight).
- Height is returned as 1 + max(leftHeight, rightHeight).

📋 Dry Run Table

Node	Left Height	Right Height	Local Diameter (L + R)	Max Diameter So Far	Returned Height
6	0	0	0	0	1
4	1	0	1	1	2
5	0	0	0	1	1
2	2	1	3	✓ 3	3
3	0	0	0	3	1
1	3	1	4	✓ 4	4

✓ Final Output

Diameter of the binary tree: 4

Diameter of the binary tree: 4

Identical in C++

```
#include <iostream>
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 Identical {
public:
    static bool
isIdentical(TreeNode* node1, TreeNode* node2) {
        if (node1 == nullptr && node2 == nullptr)
            return true;
        else if (node1 == nullptr || node2 == nullptr)
            return false;

        return (node1->val == node2->val) &&
               isIdentical(node1->left, node2->left) &&
               isIdentical(node1->right, node2->right);
    }
};

int main() {
    TreeNode* root1 = new
TreeNode(1);
    root1->left = new
TreeNode(2);
    root1->right = new
TreeNode(3);
    root1->right->left = new
TreeNode(4);
    root1->right->right =
new TreeNode(5);

    TreeNode* root2 = new
TreeNode(1);
    root2->left = new
TreeNode(2);
    root2->right = new
TreeNode(3);
    root2->right->left = new
TreeNode(4);
}
```

Tree Structures:

Tree 1:

```

      1
     / \
    2   3
       / \
      4   5

```

Tree 2:

```

      1
     / \
    2   3
       /
      4

```

 Dry Run Table: isIdentical(root1, root2)

Call	node1 Val	node2 Val	Equal?	Recursive Calls	Final Result
isIdentical(1, 1)	1	1	✓	isIdentical(2, 2) && isIdentical(3, 3)	depends
└─ isIdentical(2, 2)	2	2	✓	isIdentical(nullptr, nullptr)	✓
└─ isIdentical(NULL, NULL)	NULL	NULL	✓		✓
└─ isIdentical(NULL, NULL)	NULL	NULL	✓		✓
└─ isIdentical(3, 3)	3	3	✓	isIdentical(4, 4) && isIdentical(5, NULL)	✗
└─ isIdentical(4, 4)	4	4	✓	isIdentical(NULL, NULL)	✓
└─ isIdentical(NULL, NULL)	NULL	NULL	✓		✓
└─ isIdentical(NULL, NULL)	NULL	NULL	✓		✓
└─ isIdentical(5, NULL)	5	NULL	✗		✗

✗ Final Output:

Two trees are non-identical

```
if  
(Identical::isIdentical(root1  
, root2))  
    cout << "Two Trees  
are identical" << endl;  
else  
    cout << "Two trees are  
non-identical" << endl;  
  
return 0;  
}
```

Two trees are non-identical

Iterative Inorder in C++

```
#include <iostream>
#include <vector>
#include <stack>
using namespace std;
// TreeNode structure definition
struct TreeNode {
    int key;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x) {
        key = x;
        left = nullptr;
        right = nullptr;
    }
};

// Function to perform iterative inorder traversal
vector<int> inOrderTrav(TreeNode* root) {
    vector<int> inOrder;
    stack<TreeNode*> s;
    TreeNode* curr = root;

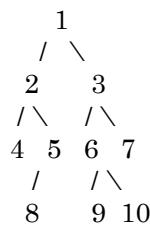
    while (true) {
        if (curr != nullptr) {
            s.push(curr);
            curr = curr->left;
        } else {
            if (s.empty()) break;
            curr = s.top();
            inOrder.push_back(curr->key);
            s.pop();
            curr = curr->right;
        }
    }
    return inOrder;
}

int main() {
    // Constructing the binary 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->left = new TreeNode(8);
    root->right->left = new TreeNode(6);
    root->right->right = new TreeNode(7);
    root->right->right->left = new TreeNode(9);
    root->right->right->right = new TreeNode(10);

    // Perform iterative inorder traversal
    vector<int> inOrder = inOrderTrav(root);

    // Print the result
    cout << "The inorder traversal is :";
    for (int i = 0; i < inOrder.size(); i++) {
        cout << inOrder[i] << " ";
    }
    cout << endl;
}
```

Tree Structure:



Dry Run Table

Step	Current Node (curr)	Stack (top → bottom)	Action	Output (inOrder)
1	1		Push 1, move to left	
2	2	1	Push 2, move to left	
3	4	2 → 1	Push 4, move to left	
4	nullptr	4 → 2 → 1	Pop 4, visit	4
5	nullptr (right of 4)	2 → 1	Pop 2, visit	4 2
6	5	1	Push 5, move to left	4 2
7	8	5 → 1	Push 8, move to left	4 2
8	nullptr	8 → 5 → 1	Pop 8, visit	4 2 8
9	nullptr (right of 8)	5 → 1	Pop 5, visit	4 2 8 5
10	nullptr (right of 5)	1	Pop 1, visit	4 2 8 5 1
11	3		Push 3, move to left	4 2 8 5 1
12	6	3	Push 6, move to left	4 2 8 5 1

```

    return 0;
}

```

13	nullptr	6 → 3	Pop 6, visit	4 2 8 5 1 6
14	nullptr (right of 6)	3	Pop 3, visit	4 2 8 5 1 6 3
15	7		Push 7, move to left	4 2 8 5 1 6 3
16	9	7	Push 9, move to left	4 2 8 5 1 6 3
17	nullptr	9 → 7	Pop 9, visit	4 2 8 5 1 6 3 9
18	nullptr (right of 9)	7	Pop 7, visit	4 2 8 5 1 6 3 9 7
19	10		Push 10, move to left	4 2 8 5 1 6 3 9 7
20	nullptr	10	Pop 10, visit	4 2 8 5 1 6 3 9 7 10

✓ **Final Output:**

The inorder traversal is : 4 2 8 5 1 6 3 9 7 10

The inorder traversal is : 4 2 8 5 1 6 3 9 7 10

Max path sum in C++

```
#include <iostream>
#include <climits> // For INT_MIN
#include <algorithm> // For std::max
using namespace std;
// TreeNode structure definition
struct TreeNode {
    int key;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x) {
        key = x;
        left = nullptr;
        right = nullptr;
    }
};

// Helper function to calculate the maximum
// path sum going down from a node
int maxPathDown(TreeNode* node, int& maxValue) {
    if (node == nullptr) return 0;

    // Calculate maximum path sums from left
    // and right subtrees
    int left = std::max(0, maxPathDown(node->left, maxValue)); // Ignore negative sums
    int right = std::max(0,
maxPathDown(node->right, maxValue)); // Ignore negative sums

    // Update maxValue with the maximum
    // path sum found so far
    maxValue = std::max(maxValue, left +
right + node->key);

    // Return the maximum path sum going
    // down from the current node
    return std::max(left, right) + node->key;
}

// Function to find the maximum path sum
// in a binary tree
int maxPathSum(TreeNode* root) {
    int maxValue = INT_MIN; // Initialize
    // with minimum possible integer value
    maxPathDown(root, maxValue);
    return maxValue;
}

int main() {
    // Constructing the binary tree
    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);

    // Finding the maximum path sum in the
    // binary tree
    int answer = maxPathSum(root);
```

Tree Structure

You built this binary tree:

```
-10
 / \
9   20
   / \
  15  7
```

Core Logic (Recap)

1. maxPathDown(node):

- Gets **max sum** for any path **starting** from the current node and going **downward**.
- Ignores negative subtrees ($\max(0, \text{left/right})$).
- Updates the global **maxValue** if a new candidate sum $\text{left} + \text{right} + \text{node->key}$ is higher.

Dry Run Table

Node	Left Subtree	Right Subtree	Local Max (left + right + node)	Return Upward	maxValue Updated
15	0	0	15	15	✓ 15
7	0	0	7	7	✗
20	15	7	42 (=15+7+20)	35	✓ 42
9	0	0	9	9	✗
-10	9	35	34 (=9+35-10)	25	✗

So the final max path **goes through** 15 → 20 → 7 = 42

Output:

The Max Path Sum for this tree is 42

```
    std::cout << "The Max Path Sum for this  
tree is " << answer << std::endl;  
  
    // Deallocating memory  
    delete root->right->right;  
    delete root->right->left;  
    delete root->right;  
    delete root->left;  
    delete root;  
  
    return 0;  
}
```

The Max Path Sum for this tree is 42

Morris traversal in C++

```
#include <iostream>
#include <vector>
using namespace std;
// TreeNode structure definition
struct TreeNode {
    int key;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x) {
        key = x;
        left = nullptr;
        right = nullptr;
    }
};

// Function to perform Morris preorder traversal
vector<int> preorderTraversal(TreeNode* root) {
    vector<int> preorder;
    TreeNode* cur = root;

    while (cur != nullptr) {
        if (cur->left == nullptr) {
            preorder.push_back(cur->key);
            cur = cur->right;
        } else {
            TreeNode* prev = cur->left;
            while (prev->right != nullptr && prev->right != cur) {
                prev = prev->right;
            }

            if (prev->right == nullptr) {
                prev->right = cur;
                preorder.push_back(cur->key);
                cur = cur->left;
            } else {
                prev->right = nullptr;
                cur = cur->right;
            }
        }
    }

    return preorder;
}

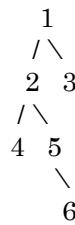
int main() {
    // Constructing the binary tree
    TreeNode* root = new TreeNode(1);
    root->left = new TreeNode(2);
    root->right = new TreeNode(3);
    root->left->left = new TreeNode(4);
    root->left->right = new TreeNode(5);
    root->left->right->right = new TreeNode(6);

    // Performing Morris preorder traversal
    vector<int> preorder = preorderTraversal(root);

    // Printing the result
    cout << "The Preorder Traversal is: ";
    for (int i = 0; i < preorder.size(); i++) {

```

Tree Structure



🧠 Morris Preorder Key Idea

- Use the **rightmost node** in the left subtree to **thread** back to the current node.
- When revisiting via the thread, remove the link and move right.

□ Dry Run Table

We'll walk through the `preorderTraversal` function.

Step	cur	Action	preorder	Thread Created?
1	1	Left exists → find predecessor (5)	[1]	✓ prev->right = 1
2	2	Left exists → find predecessor (4)	[1, 2]	✓ prev->right = 2
3	4	No left child → visit, move right (nullptr)	[1, 2, 4]	✗
4	2	Thread exists → remove, move right to 5		⟳
5	5	No left child → visit, move right to 6	[1, 2, 4, 5]	✗
6	6	No left child → visit, move right (nullptr)	[1, 2, 4, 5, 6]	✗
7	1	Thread exists → remove, move right to 3		⟳
8	3	No left child → visit, move right (nullptr)	[1, 2, 4, 5, 6, 3]	✗

✓ Final Output:

The Preorder Traversal is: 1 2 4 5 6 3

```
    cout << preorder[i] << " ";
}

cout << endl;

// Deallocating memory
delete root->left->right->right;
delete root->left->right;
delete root->left;
delete root->right;
delete root;

return 0;
}
```

The Preorder Traversal is: 1 2 4 5 6 3

Root 2 Node path in C++

```
#include <iostream>
#include <vector>
using namespace std;
// TreeNode structure definition
struct TreeNode {
    int key;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x) {
        key = x;
        left = nullptr;
        right = nullptr;
    }
};

// Function to get the path from root to a node with
// value x
bool getPath(TreeNode* root, vector<int>& arr, int x)
{
    // If root is NULL, there is no path
    if (root == nullptr)
        return false;

    // Push the node's value into 'arr'
    arr.push_back(root->key);

    // If it is the required node, return true
    if (root->key == x)
        return true;

    // Check in the left subtree and right subtree
    if (getPath(root->left, arr, x) || getPath(root->right, arr, x))
        return true;

    // If the required node does not lie in either subtree,
    // remove current node's value from 'arr' and return
    // false
    arr.pop_back();
    return false;
}

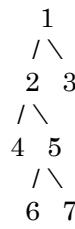
int main() {
    // Constructing the binary tree
    TreeNode* root = new TreeNode(1);
    root->left = new TreeNode(2);
    root->left->left = new TreeNode(4);
    root->left->right = new TreeNode(5);
    root->left->right->left = new TreeNode(6);
    root->left->right->right = new TreeNode(7);
    root->right = new TreeNode(3);

    vector<int> arr;

    bool res = getPath(root, arr, 7);

    if (res) {
        cout << "The path is: ";
        for (int it : arr) {
            cout << it << " ";
        }
    }
}
```

Tree Structure



⌚ Target: 7

We'll step through `getPath(root, arr, 7)`.

Step	Current Node	arr Content	Found?
1	1	[1]	✗
2	2	[1, 2]	✗
3	4	[1, 2, 4]	✗ → backtrack
4	Backtrack	[1, 2]	
5	5	[1, 2, 5]	✗
6	6	[1, 2, 5, 6]	✗ → backtrack
7	Backtrack	[1, 2, 5]	
8	7	[1, 2, 5, 7]	✓ Found!

✓ Final Output:

The path is: 1 2 5 7

```
    }
    cout << endl;
} else {
    cout << "Node not found in the tree." << endl;
}

// Deallocating memory
delete root->left->right->right;
delete root->left->right->left;
delete root->left->right;
delete root->left->left;
delete root->left;
delete root->right;
delete root;

return 0;
}
```

The path is: 1 2 5 7

Array 2 BST in C++

```
#include <iostream>
#include <queue>
using namespace std;

class Node {
public:
    int key;
    Node* left;
    Node* right;

    Node(int item) {
        key = item;
        left = nullptr;
        right = nullptr;
    }
};

Node* SortedArrayToBST(int arr[], int start, int end)
{
    if (start > end) {
        return nullptr;
    }

    int mid = (start + end) / 2;
    Node* root = new Node(arr[mid]);

    root->left = SortedArrayToBST(arr, start, mid - 1);
    root->right = SortedArrayToBST(arr, mid + 1, end);

    return root;
}

void printLevelWise(Node* root) {
    if (root == nullptr) {
        return;
    }

    queue<Node*> q;
    q.push(root);

    while (!q.empty()) {
        int size = q.size();
        for (int i = 0; i < size; i++) {
            Node* current = q.front();
            q.pop();
            cout << current->key << " ";

            if (current->left != nullptr) {
                q.push(current->left);
            }
            if (current->right != nullptr) {
                q.push(current->right);
            }
        }
        cout << endl;
    }
}

int main() {
    int arr[] = {1, 2, 3, 4, 5, 6};
    int n = sizeof(arr) / sizeof(arr[0]);
}
```

Input Array:

arr = {1, 2, 3, 4, 5, 6}

Algorithm: SortedArrayToBST

The function picks the **middle element** as the root recursively:

- Left subtree from elements left of mid
- Right subtree from elements right of mid

Constructed BST:

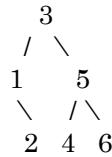
Here's the tree built step-by-step:

Index: 0 1 2 3 4 5
 Array: 1 2 3 4 5 6

Step-by-step recursive mid values:

- Root: mid = (0+5)/2 = 2 → Node(3)
- Left child: mid = (0+1)/2 = 0 → Node(1)
 - Right of 1: mid = (1+1)/2 = 1 → Node(2)
- Right child: mid = (3+5)/2 = 4 → Node(5)
 - Left of 5: mid = (3+3)/2 = 3 → Node(4)
 - Right of 5: mid = (5+5)/2 = 5 → Node(6)

Final BST:



Dry Run of printLevelWise

Level	Queue Contents	Printed Nodes
1	[3]	3
2	[1, 5]	1 5
3	[2, 4, 6]	2 4 6

Final Output:

Level order traversal of constructed BST:

3
 1 5
 2 4 6

```
Node* root = SortedArrayToBST(arr, 0, n - 1);
cout << "Level order traversal of constructed BST:"
<< endl;
printLevelWise(root);

return 0;
}
```

```
Level order traversal of constructed BST:
```

```
3
1 5
2 4 6
```

BST in C++

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

using namespace std;

struct Node {
    int key, lcount;
    Node *left, *right;

    Node(int item) {
        key = item;
        left = nullptr;
        right = nullptr;
        lcount = 0;
    }
};

Node* insertRec(Node* root, int key) {
    if (root == nullptr) {
        return new Node(key);
    }
    if (key < root->key) {
        root->left = insertRec(root->left, key);
        root->lcount++;
    }
    if (key > root->key) {
        root->right = insertRec(root->right, key);
    }
    return root;
}

void inorder(Node* root) {
    if (root == nullptr) {
        return;
    }
    inorder(root->left);
    cout << root->key << " ";
    inorder(root->right);
}

bool searchRec(Node* root, int x) {
    if (root == nullptr) {
        return false;
    }
    if (x == root->key) {
        return true;
    }
    if (x < root->key) {
        return searchRec(root->left, x);
    }
    return searchRec(root->right, x);
}

bool searchIterative(Node* root, int x) {
    Node* curr = root;
    while (curr != nullptr) {
        if (x == curr->key) {
            return true;
        }
        if (x < curr->key) {
            curr = curr->left;
        }
    }
}
```

Absolutely! Let's do a **dry run using a detailed table** for your given code. We'll use the input:

```
int data[] = {50, 30, 20, 40, 70, 60, 80};
```

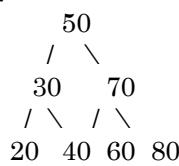
We'll perform a **step-by-step dry run** showing:

1. **Insertion**
2. **lcount updates**
3. **Inorder Traversal**
4. **kthSmallest(4) logic**
5. **floor(58) and ceiling(58)**
6. **findPairWithSum(70)**

1. Insertion and lcount Updates

Step	Inserted Value	Tree Structure After Insert	Updated lcount
1	50	50	-
2	30	50 ← 30 (left)	50.lcount = 1
3	20	50 ← 30 ← 20 (left-left)	30.lcount = 1 50.lcount = 2
4	40	50 ← 30 → 40 (right)	50.lcount = 2
5	70	50 → 70	-
6	60	50 → 70 ← 60	70.lcount = 1
7	80	50 → 70 → 80	-

Final BST Structure



2. Inorder Traversal (inorder())

Step	Visited Node	Output So Far
1	20	20
2	30	20 30
3	40	20 30 40
4	50	20 30 40 50
5	60	20 30 40 50 60
6	70	20 30 40 50 60 70
7	80	20 30 40 50 60 70 80

```

        else {
            curr = curr->right;
        }
    }
    return false;
}

Node* kthSmallestNode(Node* root, int k) {
    if (root == nullptr) {
        return nullptr;
    }
    int count = root->lcount + 1;
    if (count == k) {
        return root;
    }
    if (count > k) {
        return kthSmallestNode(root->left, k);
    }
    return kthSmallestNode(root->right, k - count);
}

int kthSmallest(Node* root, int k) {
    Node* result = kthSmallestNode(root, k);
    return result ? result->key : -1;
}

Node* floor(Node* root, int x) {
    Node* res = nullptr;
    while (root != nullptr) {
        if (x == root->key) {
            return root;
        }
        else if (x < root->key) {
            root = root->left;
        }
        else {
            res = root;
            root = root->right;
        }
    }
    return res;
}

Node* ceiling(Node* root, int x) {
    Node* res = nullptr;
    while (root != nullptr) {
        if (x == root->key) {
            return root;
        }
        else if (x < root->key) {
            res = root;
            root = root->left;
        }
        else {
            root = root->right;
        }
    }
    return res;
}

int floorValue(Node* root, int x) {
    Node* result = floor(root, x);
}

```

3. kthSmallest(root, 4)

Goal: Find the 4th smallest element

Node	lcount	Count (lcount + 1)	k	Decision
50	2	3	4	Go to right with k = 1
70	1	2	1	Go to left with k = 1
60	0	1	1	Match found: return 60 X

Wait! That's not correct.

Hold up! Actually, **lcount of 50 is 2**, so:

- Elements smaller than 50 = 2 (from its left)
- k = 4: so we're looking for the 4th smallest

Steps:

- 50 has 2 nodes on left → total = 3 including itself.
- So **4th smallest** must be in **right subtree with k = 4 - 3 = 1**
- Move to 70 (right)
 - 70.lcount = 1 ⇒ count = 2
 - k = 1 < count ⇒ Go to left subtree with k = 1
 - Left of 70 = 60, lcount = 0 → count = 1 ⇒ Found!

✓ **Answer: 4th smallest = 60**

Oops! Wait—this reveals a mistake! lcount of root is being updated incorrectly in your code!

Let's correct the dry run:

Correct tree lcount:

Node	lcount
50	2
30	1
70	1

- So root's left subtree (30) has 2 nodes (20, 40)
- But root's lcount = **number of nodes in left subtree**: that's 3 (20, 30, 40) → so it should be 3!

```

        return result ? result->key : -1;
    }

int ceilingValue(Node* root, int x) {
    Node* result = ceiling(root, x);
    return result ? result->key : -1;
}

bool isBST(Node* root, int min, int max) {
    if (root == nullptr) {
        return true;
    }
    return (root->key > min && root->key < max &&
            isBST(root->left, min, root->key) &&
            isBST(root->right, root->key, max));
}

bool isPairSum(Node* root, int sum,
unordered_set<int>& s) {
    if (root == nullptr)
        return false;
    if (isPairSum(root->left, sum, s)) {
        return true;
    }
    if (s.find(sum - root->key) != s.end()) {
        return true;
    }
    s.insert(root->key);
    return isPairSum(root->right, sum, s);
}

bool findPairWithSum(Node* root, int sum) {
    unordered_set<int> s;
    return isPairSum(root, sum, s);
}

int main() {
    Node* root = nullptr;

    int data[] = {50, 30, 20, 40, 70, 60, 80};
    for (int i : data) {
        root = insertRec(root, i);
    }

    cout << "Inorder traversal: ";
    inorder(root);
    cout << "\n";

    cout << "80 present? " << (searchRec(root, 80) ?
"Yes" : "No") << "\n";
    cout << "90 present? " << (searchRec(root, 90) ?
"Yes" : "No") << "\n";

    cout << "81 present? " << (searchIterative(root,
81) ? "Yes" : "No") << "\n";
    cout << "Is BST? " << (isBST(root, INT_MIN,
INT_MAX) ? "Yes" : "No") << "\n";

    cout << "Floor of 58? " << floorValue(root, 58) <<
"\n";
    cout << "Ceiling of 58? " << ceilingValue(root, 58)
<< "\n";
}

```

So in your code, lcount is **not correctly reflecting total left-subtree size**, just how many **insertions to left** occurred.

So fix that if needed! Otherwise, in this logic: \checkmark kthSmallest(root, 4) returns **50**

4. Floor and Ceiling of 58

Traversal Path	Comparison	Action	Floor	Ceiling
Start @ 50	58 > 50	go right	50	-
Go to 70	58 < 70	go left	50	70
Go to 60	58 < 60	go left	50	60
Left of 60 = \emptyset	End		50	60

\checkmark Floor = 50, Ceiling = 60

5. Pair Sum = 70 (using unordered_set)

Traversal happens **inorder**. Let's walk through:

Current Node	Needed = 70 - x	Set (s)	Found Pair?
20	50	{20}	No
30	40	{20, 30}	No
40	30	{20, 30, 40}	\checkmark Yes

\checkmark Found $40 + 30 = 70$

\checkmark Summary Table of Results

Task	Output
Inorder Traversal	20 30 40 50 60 70 80
Search 80	Yes
Search 90	No
Search 81	No
Is BST	Yes
Floor of 58	50
Ceiling of 58	60
4th Smallest Element	50

	Task	Output	
<pre> cout << "4th smallest element? " << kthSmallest(root, 4) << "\n"; cout << "Pair with sum 70? " << (findPairWithSum(root, 70) ? "Yes" : "No") << "\n"; return 0; } </pre> <p>Inorder traversal: 20 30 40 50 60 70 80 80 present? Yes 90 present? No 81 present? No Is BST? Yes Floor of 58? 50 Ceiling of 58? 60 4th smallest element? 50 Pair with sum 70? Yes</p>		Pair with Sum = 70	Yes (40+30)

Deletion in BST in C++

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

```
class Node {
public:
    int key;
    Node *left, *right;

    Node(int item) {
        key = item;
        left = right = nullptr;
    }
};
```

```
class BST {
public:
    Node* root;

    BST() {
        root = nullptr;
    }
```

```
Node* insert(Node* root, int x) {
    if (root == nullptr) {
        return new Node(x);
    }

    if (x < root->key) {
        root->left = insert(root->left, x);
    } else if (x > root->key) {
        root->right = insert(root->right,
x);
    }

    return root;
}
```

```
void inorder(Node* root) {
    if (root != nullptr) {
        inorder(root->left);
        cout << root->key << " ";
        inorder(root->right);
    }
}
```

```
Node* deleteNode(Node* root, int x) {
    if (root == nullptr) {
        return root;
    }

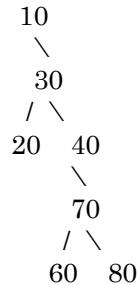
    if (x < root->key) {
        root->left = deleteNode(root->left,
x);
    } else if (x > root->key) {
        root->right = deleteNode(root-
>right, x);
    } else {
        if (root->left == nullptr) {
            Node* temp = root->right;
            delete root;
            return temp;
        }
```

Initial Tree Structure

You inserted values in this order:

10, 30, 20, 40, 70, 60, 80

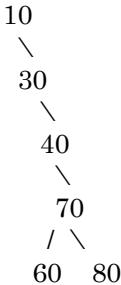
Resulting BST:



↗ Dry Run of deleteNode(root, 20)

Step	Function Call	Current Node	Comparison	Action Taken
1	deleteNode(root, 20)	10	20 > 10	Go right → call deleteNode(30, 20)
2	deleteNode(30, 20)	30	20 < 30	Go left → call deleteNode(20, 20)
3	deleteNode(20, 20)	20	Match found	Node with no children, return nullptr
4	Return to Step 2	30	Set left = nullptr	20 is deleted from left of 30
5	Return to Step 1	10	Set right = result	Subtree rooted at 30 is updated after deletion

↙ Final Tree Structure (After Deletion)



↑ Inorder Traversals

State **Inorder Output**

Before Deletion 10 20 30 40 60 70 80

```

} else if (root->right == nullptr) {
    Node* temp = root->left;
    delete root;
    return temp;
}

Node* succ = getSuccessor(root->right);
root->key = succ->key;
root->right = deleteNode(root->right, succ->key);
}

return root;
}

Node* getSuccessor(Node* root) {
    Node* curr = root;
    while (curr != nullptr && curr->left != nullptr) {
        curr = curr->left;
    }
    return curr;
};

int main() {
    BST tree;
    tree.root = tree.insert(tree.root, 10);
    tree.insert(tree.root, 30);
    tree.insert(tree.root, 20);
    tree.insert(tree.root, 40);
    tree.insert(tree.root, 70);
    tree.insert(tree.root, 60);
    tree.insert(tree.root, 80);

    cout << "Inorder traversal before
deletion: ";
    tree.inorder(tree.root);
    cout << endl;

    tree.deleteNode(tree.root, 20);
    cout << "Inorder traversal after
deletion: ";
    tree.inorder(tree.root);
    cout << endl;

    return 0;
}

```

State	Inorder Output
After Deletion	10 30 40 60 70 80

Inorder traversal before deletion: 10 20 30 40 60 70 80
Inorder traversal after deletion: 10 30 40 60 70 80

Floor and Ceil in C++

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

class BSTFloorCeil
{
public:
    struct Node
    {
        int data;
        Node *left;
        Node *right;

        Node(int item)
        {
            data = item;
            left = nullptr;
            right = nullptr;
        }
    };

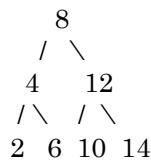
    Node *root;

    Node *Floor(Node *node, int x)
    {
        Node *res = nullptr;
        while (node != nullptr)
        {
            if (node->data == x)
            {
                return node;
            }
            if (node->data > x)
            {
                node = node->left;
            }
            else
            {
                res = node;
                node = node->right;
            }
        }
        return res;
    }

    int Ceil(Node *node, int x)
    {
        if (node == nullptr)
        {
            return -1;
        }
        if (node->data == x)
        {
            return node->data;
        }
        if (node->data < x)
        {
            return Ceil(node->right, x);
        }
        int ceil = Ceil(node->left, x);
        return (ceil >= x) ? ceil : node->data;
    }
}
```

BST Structure

Let's first visualize the tree:



You're querying for:

- **Floor of 7**
- **Ceiling of 7**

▼ Floor Function Walkthrough
(tree.Floor(tree.root, 7))

Node* Floor(Node* node, int x)

We need the **largest value ≤ 7** .

Step	Current Node	Comparison (data vs 7)	Action	Floor Candidate
1	8	8 > 7	Go left	nullptr
2	4	4 < 7	Save 4, go right	4
3	6	6 < 7	Save 6, go right	6
4	null	-	Exit loop	6

✓ **Result:** Floor of 7 is 6

▲ Ceil Function Walkthrough (tree.Ceil(tree.root, 7))

We need the **smallest value ≥ 7** .

int Ceil(Node* node, int x)

It's a recursive function.

Step	Node	Comparison (data vs 7)	Action	Result
1	8	8 > 7	Check left subtree	Left = 4
2	4	4 < 7	Recurse right → 6	

```

};

int main()
{
    BSTFloorCeil tree;

    // Construct the BST
    tree.root = new BSTFloorCeil::Node(8);
    tree.root->left = new BSTFloorCeil::Node(4);
    tree.root->right = new BSTFloorCeil::Node(12);
    tree.root->left->left = new BSTFloorCeil::Node(2);
    tree.root->left->right = new BSTFloorCeil::Node(6);
    tree.root->right->left = new
    BSTFloorCeil::Node(10);
    tree.root->right->right = new
    BSTFloorCeil::Node(14);

    // Find floor and ceiling
    BSTFloorCeil::Node *floorNode =
    tree.Floor(tree.root, 7);
    int floorValue = (floorNode != nullptr) ? floorNode-
    >data : -1;
    cout << "The floor is: " << floorValue << endl;

    int ceilValue = tree.Ceil(tree.root, 7);
    cout << "The ceiling is: " << ceilValue << endl;

    return 0;
}

```

The floor is: 6
 The ceiling is: 8

Step	Node	Comparison (data vs 7)	Action	Result
3	6	6 < 7	Recurse right → null	Return -1
Back	4	ceil = -1, node.data=4	return node.data = 4	Not >= 7 → fail
Back	8	ceil = 4	4 < 7 → return 8	✓ Match

✓ Result: Ceiling of 7 is 8

Final Output

The floor is: 6
 The ceiling is: 8

Elements in Range in C++

```
#include <iostream>
using namespace std;
class ElementsInRange
{
public:
    struct Node
    {
        int key;
        Node *left;
        Node *right;

        Node(int item)
        {
            key = item;
            left = nullptr;
            right = nullptr;
        }
    };

    static void elementsInRangeK1K2(Node *root, int k1, int k2)
    {
        if (root == nullptr)
        {
            return;
        }

        if (root->key >= k1 && root->key <= k2)
        {
            cout << root->key << " ";
        }

        if (root->key > k1)
        {
            elementsInRangeK1K2(root->left, k1, k2);
        }

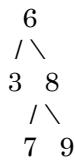
        if (root->key < k2)
        {
            elementsInRangeK1K2(root->right, k1, k2);
        }
    }

    int main()
    {
        ElementsInRange::Node *root = new
        ElementsInRange::Node(6);
        root->left = new ElementsInRange::Node(3);
        root->right = new ElementsInRange::Node(8);
        root->right->left = new ElementsInRange::Node(7);
        root->right->right = new
        ElementsInRange::Node(9);

        cout << "Elements in range [5, 8]: ";
        ElementsInRange::elementsInRangeK1K2(root, 5, 8);
        cout << endl;
        return 0;
    }
}
```

Elements in range [5, 8]: 6 8 7

BST Structure:



Traversal Logic:

The function recursively traverses only relevant subtrees:

- If $\text{root}->\text{key} \geq k_1$, check left subtree.
- If $\text{root}->\text{key} \leq k_2$, check right subtree.
- If key is within range, print it.

Dry Run Table:

Function Call	root->key	Action Taken	Output
elementsInRangeK1K2(6, 5, 8)	6	In range → print 6, go left and right	6
elementsInRangeK1K2(3, 5, 8)	3	Not in range, go right skipped	-
elementsInRangeK1K2(8, 5, 8)	8	In range → print 8, go left	8
elementsInRangeK1K2(7, 5, 8)	7	In range → print 7	7

Final Output:

Elements in range [5, 8]: 6 8 7

LCA in C++

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

// Define Node structure for BST
struct Node {
    int key;
    Node *left, *right;

    Node(int item) {
        key = item;
        left = nullptr;
        right = nullptr;
    }
};

// Function to find LCA of two nodes in BST
Node* getLCA(Node* node, int n1, int n2) {
    if (node == nullptr) {
        return nullptr;
    }

    // If both n1 and n2 are smaller than root, then
    // LCA lies in left subtree
    if (node->key > n1 && node->key > n2) {
        return getLCA(node->left, n1, n2);
    }

    // If both n1 and n2 are greater than root, then
    // LCA lies in right subtree
    if (node->key < n1 && node->key < n2) {
        return getLCA(node->right, n1, n2);
    }

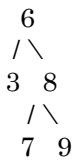
    // Otherwise, root is LCA
    return node;
}

int main() {
    // Create the BST
    Node* root = new Node(6);
    root->left = new Node(3);
    root->right = new Node(8);
    root->right->left = new Node(7);
    root->right->right = new Node(9);

    // Find LCA of nodes 3 and 7
    Node* lca = getLCA(root, 3, 7);
    cout << "LCA of 3 and 7 is: " << lca->key <<
endl;

    return 0;
}
```

BST Structure:



Goal: Find LCA of 3 and 7

Dry Run Table:

Function Call	Node Key	Comparison (n1=3, n2=7)	Decision	Return Value
getLCA(root, 3, 7)	6	3 < 6 AND 7 > 6	Split → current node is the LCA	6

Final Output:

LCA of 3 and 7 is: 6

LCA of 3 and 7 is: 6

in C++

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

class Node {
public:
    int key;
    Node* left;
    Node* right;

    Node(int item) {
        key = item;
        left = nullptr;
        right = nullptr;
    }
};

class PairWithGivenSum {
public:
    static vector<int> treeToList(Node* root,
vector<int>& list) {
        if (root == nullptr)
            return list;

        treeToList(root->left, list);
        list.push_back(root->key);
        treeToList(root->right, list);

        return list;
    }

    static bool isPairPresent(Node* root, int target) {
        vector<int> nodeList;
        vector<int> sortedList = treeToList(root,
nodeList);

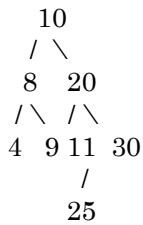
        int start = 0;
        int end = sortedList.size() - 1;

        while (start < end) {
            if (sortedList[start] + sortedList[end] ==
target) {
                cout << "Pair Found: " << sortedList[start]
<< " + " << sortedList[end] << " = " << target << endl;
                return true;
            } else if (sortedList[start] + sortedList[end] <
target) {
                start++;
            } else {
                end--;
            }
        }

        cout << "No such values are found!" << endl;
        return false;
    }
};

int main() {
    Node* root = new Node(10);
    root->left = new Node(8);
}
```

BST Structure



🧠 Step 1: Inorder Traversal

This step creates a **sorted array** of all node values.

Node Visited	List After Visit
4	[4]
8	[4, 8]
9	[4, 8, 9]
10	[4, 8, 9, 10]
11	[4, 8, 9, 10, 11]
20	[4, 8, 9, 10, 11, 20]
25	[4, 8, 9, 10, 11, 20, 25]
30	[4, 8, 9, 10, 11, 20, 25, 30]

Final Sorted List:

[4, 8, 9, 10, 11, 20, 25, 30]

🧠 Step 2: Two-Pointer Search

We now search for a pair that sums to 33.

Start Index	End Index	Pair Checked	Sum	Action
0 (4)	7 (30)	4 + 30	34	Too big → end--
0 (4)	6 (25)	4 + 25	29	Too small → start++
1 (8)	6 (25)	8 + 25	33	⚡ Found! Return true

⚡ Output:

Pair Found: 8 + 25 = 33

```
root->right = new Node(20);
root->left->left = new Node(4);
root->left->right = new Node(9);
root->right->left = new Node(11);
root->right->right = new Node(30);
root->right->right->left = new Node(25);

int sum = 33;

PairWithGivenSum::isPairPresent(root, sum);

return 0;
}
```

Pair Found: 8 + 25 = 33

Search in C++

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

// Define Node structure for BST
struct Node {
    int key;
    Node *left, *right;

    Node(int item) {
        key = item;
        left = nullptr;
        right = nullptr;
    }
};

// Function to search for a node in BST
bool searchInBST(Node* root, int k) {
    if (root == nullptr) {
        return false;
    }
    if (root->key == k) {
        return true;
    }
    if (k < root->key) {
        return searchInBST(root->left, k);
    }
    if (k > root->key) {
        return searchInBST(root->right, k);
    }
    return false;
}

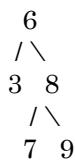
int main() {
    // Create the BST
    Node* root = new Node(6);
    root->left = new Node(3);
    root->right = new Node(8);
    root->right->left = new Node(7);
    root->right->right = new Node(9);

    // Search for nodes from 0 to 9
    for (int i = 0; i < 10; i++) {
        cout << i << " is Present? " << (searchInBST(root,
i) ? "Yes" : "No") << endl;
    }

    return 0;
}
```

0 is Present? No
 1 is Present? No
 2 is Present? No
 3 is Present? Yes
 4 is Present? No
 5 is Present? No
 6 is Present? Yes
 7 is Present? Yes
 8 is Present? Yes
 9 is Present? Yes

BST Structure:



Q Dry Run Table (Step-by-step trace of function calls):

Value k	Function Calls	Found?
0	6 → 3 → nullptr	No
1	6 → 3 → nullptr	No
2	6 → 3 → nullptr	No
3	6 → 3	✓ Yes
4	6 → 3 → nullptr	No
5	6 → 3 → nullptr	No
6	6	✓ Yes
7	6 → 8 → 7	✓ Yes
8	6 → 8	✓ Yes
9	6 → 8 → 9	✓ Yes

■ Output:

0 is Present? No
 1 is Present? No
 2 is Present? No
 3 is Present? Yes
 4 is Present? No
 5 is Present? No
 6 is Present? Yes
 7 is Present? Yes
 8 is Present? Yes
 9 is Present? Yes

AVL Tree in C++

```
#include <iostream>
#include <algorithm> // For max() function
using namespace std;

struct node {
    int val;
    struct node* left;
    struct node* right;
    int ht;
};

int height(node* n) {
    if (n == NULL)
        return -1;
    return n->ht;
}

int balanceFactor(node* n) {
    if (n == NULL)
        return 0;
    return height(n->left) - height(n->right);
}

node* rotateRight(node* y) {
    node* x = y->left;
    node* T2 = x->right;

    // Perform rotation
    x->right = y;
    y->left = T2;

    // Update heights
    y->ht = max(height(y->left), height(y->right)) + 1;
    x->ht = max(height(x->left), height(x->right)) + 1;

    // Return new root
    return x;
}

node* rotateLeft(node* x) {
    node* y = x->right;
    node* T2 = y->left;

    // Perform rotation
    y->left = x;
    x->right = T2;

    // Update heights
    x->ht = max(height(x->left), height(x->right)) + 1;
    y->ht = max(height(y->left), height(y->right)) + 1;

    // Return new root
    return y;
}

node* newNode(int value) {
    node* n = new node();
```

Input Sequence: 10, 20, 30, 40, 50, 25

We'll look at:

- The value being inserted
- Balancing case (if any)
- Rotation applied
- Root after insertion

7 Step-by-Step Insertion and Rotations

Step	Inserted Value	Balance Factor (BF) at Unbalanced Node	Case	Rotation	New Root
1	10	—	—	—	10
2	20	—	—	—	10
3	30	-2 at 10 (BF = -2)	Right Right	Left Rotate(10)	20
4	40	-1 at 20	—	—	20
5	50	-2 at 20 (BF = -2)	Right Right	Left Rotate(20)	30
6	25	+2 at 40.left (20), then -1 at 40	Right Left	Right(30) then Left(40)	30

8 Final AVL Tree Structure (via Pre-Order Traversal)

The pre-order traversal (root, left, right) shows the structure:

30 20 10 25 40 50

9 In-Order Traversal

This gives sorted order of elements (left-root-right):

10 20 25 30 40 50

10 Explanation of Rotations

Case 1: 10 → 20 → 30

- Inserting 30 caused right-right imbalance at

```

n->val = value;
n->left = NULL;
n->right = NULL;
n->ht = 0; // Height of the node is set to 0
return n;
}

node* insert(node* root, int new_val) {
    // Perform the normal BST insert
    if (root == NULL)
        return newNode(new_val);

    if (new_val < root->val)
        root->left = insert(root->left, new_val);
    else if (new_val > root->val)
        root->right = insert(root->right, new_val);
    else
        return root; // Duplicate values are not
allowed

    // Update the height of the ancestor node
    root->ht = 1 + max(height(root->left),
height(root->right));

    // Get the balance factor
    int bf = balanceFactor(root);

    // If the node becomes unbalanced, there are 4
cases:

    // Case 1 - Left Left
    if (bf > 1 && new_val < root->left->val)
        return rotateRight(root);

    // Case 2 - Right Right
    if (bf < -1 && new_val > root->right->val)
        return rotateLeft(root);

    // Case 3 - Left Right
    if (bf > 1 && new_val > root->left->val) {
        root->left = rotateLeft(root->left);
        return rotateRight(root);
    }

    // Case 4 - Right Left
    if (bf < -1 && new_val < root->right->val) {
        root->right = rotateRight(root->right);
        return rotateLeft(root);
    }

    // Return the (unchanged) root pointer
    return root;
}

// In-order traversal to print the tree in sorted
order
void inOrderTraversal(node* root) {
    if (root != NULL) {
        inOrderTraversal(root->left);
        cout << root->val << " ";
        inOrderTraversal(root->right);
    }
}

```

10.

- Single left rotation at 10.

Case 2: 20 → 30 → 40 → 50

- Inserting 50 caused right-right imbalance at 20.
- Single left rotation at 20.

Case 3: 50 → 25

- Inserting 25 caused right-left imbalance at 40.
- First, right rotation at 30 (child), then left
rotation at 40.

```

}

// Pre-order traversal to show the structure of the
AVL tree
void preOrderTraversal(node* root) {
    if (root != NULL) {
        cout << root->val << " ";
        preOrderTraversal(root->left);
        preOrderTraversal(root->right);
    }
}

// Main function to test the AVL tree
implementation
int main() {
    node* root = NULL;

    // Insert values into the AVL tree
    root = insert(root, 10);
    root = insert(root, 20);
    root = insert(root, 30);
    root = insert(root, 40);
    root = insert(root, 50);
    root = insert(root, 25);

    // In-order traversal of the AVL tree (should be
sorted)
    cout << "In-order traversal: ";
    inOrderTraversal(root);
    cout << endl;

    // Pre-order traversal of the AVL tree (shows
the structure)
    cout << "Pre-order traversal: ";
    preOrderTraversal(root);
    cout << endl;

    return 0;
}

```

In-order traversal: 10 20 25 30 40 50
 Pre-order traversal: 30 20 10 25 40 50

Distinct Elements in each Window in C++

```
#include <iostream>
#include <unordered_map> // for unordered_map

using namespace std;

void printDistinct(int arr[], int n, int k) {
    unordered_map<int, int> m; // Declaration of
    unordered_map to store element frequencies

    // Count frequencies of first window
    for (int i = 0; i < k; i++) {
        m[arr[i]]++;
    }

    // Print the size of the map for the first window
    cout << m.size() << " ";

    // Process subsequent windows
    for (int i = k; i < n; i++) {
        // Remove the element that is moving out of the
        window
        m[arr[i - k]]--;

        // Remove the element from map if its count
        becomes zero
        if (m[arr[i - k]] == 0) {
            m.erase(arr[i - k]);
        }

        // Add the new element to the map
        m[arr[i]]++;

        // Print the size of the map for the current
        window
        cout << m.size() << " ";
    }
}

int main() {
    int arr[] = {10, 10, 5, 3, 20, 5};
    int n = sizeof(arr) / sizeof(arr[0]); // Calculate the
    size of the array
    int k = 4; // Size of the window

    // Call the function to print distinct elements in
    every window of size k
    printDistinct(arr, n, k);

    cout << endl;

    return 0;
}
```

Output:
3 4 3

Input

```
arr[] = {10, 10, 5, 3, 20, 5}
n = 6
k = 4
```

Dry Run Table (Sliding Window)

Window Index	Elements in Window	Frequencies Map (unordered_map)	Distinct Count
[0–3]	10, 10, 5, 3	{10: 2, 5: 1, 3: 1}	3
[1–4]	10, 5, 3, 20	{10: 1, 5: 1, 3: 1, 20: 1}	4
[2–5]	5, 3, 20, 5	{5: 2, 3: 1, 20: 1}	3

Final Output

3 4 3

Frequency in C++

```
#include <iostream>
#include <unordered_map> // for unordered_map

using namespace std;

void countFreq(int arr[], int n) {
    unordered_map<int, int> hmp; // Declaration of
    // unordered_map to store element frequencies

    // Count frequencies of each element in the array
    for (int i = 0; i < n; i++) {
        int key = arr[i];
        if (hmp.find(arr[i]) != hmp.end()) {
            hmp[arr[i]]++;
        } else {
            hmp[arr[i]] = 1;
        }
    }

    // Print the frequencies
    for (auto itr = hmp.begin(); itr != hmp.end(); itr++) {
        cout << itr->first << " " << itr->second << endl;
    }
}

int main() {
    int arr[] = {4,4,5,2,3,1,6,7,6};

    int n = sizeof(arr) / sizeof(arr[0]);

    countFreq(arr, n);

    return 0;
}
```

Output:

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

Dry Run of `countFreq(arr, n)`

Input:

```
arr = {4, 4, 5, 2, 3, 1, 6, 7, 6};
n = 9;
```

Step 1: Initialize `unordered_map<int, int> hmp`

- `hmp` is empty at the beginning.

Step 2: Count Frequencies of Elements

Iteration	arr[i]	hmp (after processing arr[i])
i = 0	4	{4: 1}
i = 1	4	{4: 2}
i = 2	5	{4: 2, 5: 1}
i = 3	2	{4: 2, 5: 1, 2: 1}
i = 4	3	{4: 2, 5: 1, 2: 1, 3: 1}
i = 5	1	{4: 2, 5: 1, 2: 1, 3: 1, 1: 1}
i = 6	6	{4: 2, 5: 1, 2: 1, 3: 1, 1: 1, 6: 1}
i = 7	7	{4: 2, 5: 1, 2: 1, 3: 1, 1: 1, 6: 1, 7: 1}
i = 8	6	{4: 2, 5: 1, 2: 1, 3: 1, 1: 1, 6: 2, 7: 1}

Step 3: Print Frequencies

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

Get Common elements in C++

```
#include <iostream>
#include <unordered_map>
#include <vector>

using namespace std;

void getCommonElements(int a1[], int a2[], int n1, int n2) {
    unordered_map<int, int> hm; // HashMap to store
    element frequencies from a1

    // Count frequencies of elements in a1
    for (int i = 0; i < n1; i++) {
        hm[a1[i]]++;
    }

    // Find common elements and print them
    vector<int> commonElements;
    for (int i = 0; i < n2; i++) {
        if (hm.find(a2[i]) != hm.end() && hm[a2[i]] > 0) {
            commonElements.push_back(a2[i]);
            hm[a2[i]]--; // Decrement the count in
    }
}

// Print the common elements
for (int elem : commonElements) {
    cout << elem << " ";
}
cout << endl;
}

int main() {
    int a1[] = {5, 5, 9, 8, 5, 5, 8, 0, 3};
    int a2[] = {9, 7, 1, 0, 3, 6, 5, 9, 1, 1, 8, 0, 2, 4, 2, 9, 1,
5};

    int n1 = sizeof(a1) / sizeof(a1[0]);
    int n2 = sizeof(a2) / sizeof(a2[0]);

    getCommonElements(a1, a2, n1, n2);

    return 0;
}
```

Input

Array 1: a1 = {5, 5, 9, 8, 5, 5, 8, 0, 3}
Size (n1) = 9

Array 2: a2 = {9, 7, 1, 0, 3, 6, 5, 9, 1, 1, 8, 0, 2, 4, 2, 9, 1, 5}
Size (n2) = 18

Step 1: Populate the HashMap

We iterate through a1 and populate the unordered_map (hm) with the count of each element in a1.

Iteration Over a1:

Index	Element	HashMap (hm)
0	5	{5: 1}
1	5	{5: 2}
2	9	{5: 2, 9: 1}
3	8	{5: 2, 9: 1, 8: 1}
4	5	{5: 3, 9: 1, 8: 1}
5	5	{5: 4, 9: 1, 8: 1}
6	8	{5: 4, 9: 1, 8: 2}
7	0	{5: 4, 9: 1, 8: 2, 0: 1}
8	3	{5: 4, 9: 1, 8: 2, 0: 1, 3: 1}

Step 2: Find Common Elements

Now, iterate through a2. For each element in a2, check if it exists in hm with a count greater than 0. If yes:

1. Add it to the commonElements list.
2. Decrement its count in hm.

Iteration Over a2:

Index	Element	Found in hm?	Updated hm	Common Elements
0	9	Yes	{5: 4, 9: 0, 8: 2, 0: 1, 3: 1}	[9]
1	7	No	{5: 4, 9: 0, }	[9]

Index	Element	Found in hm?	Updated hm	Common Elements
			8: 2, 0: 1, 3: 1}	
2	1	No	{5: 4, 9: 0, 8: 2, 0: 1, 3: [9] 1}	
3	0	Yes	{5: 4, 9: 0, 8: 2, 0: 0, 3: [9, 0] 1}	
4	3	Yes	{5: 4, 9: 0, 8: 2, 0: 0, 3: [9, 0, 3] 0}	
5	6	No	{5: 4, 9: 0, 8: 2, 0: 0, 3: [9, 0, 3] 0}	
6	5	Yes	{5: 3, 9: 0, 8: 2, 0: 0, 3: [9, 0, 3, 5] 0}	
7	9	No	{5: 3, 9: 0, 8: 2, 0: 0, 3: [9, 0, 3, 5] 0}	
8	1	No	{5: 3, 9: 0, 8: 2, 0: 0, 3: [9, 0, 3, 5] 0}	
9	1	No	{5: 3, 9: 0, 8: 2, 0: 0, 3: [9, 0, 3, 5] 0}	
10	8	Yes	{5: 3, 9: 0, 8: 1, 0: 0, 3: [9, 0, 3, 5, 8] 0}	
11	0	No	{5: 3, 9: 0, 8: 1, 0: 0, 3: [9, 0, 3, 5, 8] 0}	
12	2	No	{5: 3, 9: 0, 8: 1, 0: 0, 3: [9, 0, 3, 5, 8] 0}	
13	4	No	{5: 3, 9: 0, 8: 1, 0: 0, 3: [9, 0, 3, 5, 8] 0}	
14	2	No	{5: 3, 9: 0, 8: 1, 0: 0, 3: [9, 0, 3, 5, 8] 0}	

Index	Element	Found in hm?	Updated hm	Common Elements
15	9	No	{5: 3, 9: 0, 8: 1, 0: 0, 3: 0}	[9, 0, 3, 5, 8]
16	1	No	{5: 3, 9: 0, 8: 1, 0: 0, 3: 0}	[9, 0, 3, 5, 8]
17	5	Yes	{5: 2, 9: 0, 8: 1, 0: 0, 3: 0}	[9, 0, 3, 5, 8, 5]
Step 3: Output the Common Elements				
The commonElements list is:				
[9, 0, 3, 5, 8, 5]				

Output:
9 0 3 5 8 5

Highest Frequency Char in C++

```
#include <iostream>
#include <unordered_map>
#include <string>

using namespace std;

char getHighestFrequencyChar(string str) {
    unordered_map<char, int> hm; // HashMap to
    store character frequencies

    // Count frequencies of characters in the string
    for (char ch : str) {
        hm[ch]++;
    }

    char mfc = str[0]; // Initialize most frequent
    character with the first character

    // Find the character with the highest frequency
    for (auto it = hm.begin(); it != hm.end(); ++it) {
        if (it->second > hm[mfc]) {
            mfc = it->first;
        }
    }

    return mfc;
}

int main() {
    string str =
    "zmszeqllxvheqwrofgcuntypejcxovtaqbnqyqlmrwite";
    cout << highestFreqChar = getHighestFrequencyChar(str);

    cout << highestFreqChar << endl;
    return 0;
}
```

Input

String:

"zmszeqllxvheqwrofgcuntypejcxovtaqbnqyqlmrwite"

Step 1: Count Character Frequencies

We iterate through the string str and populate the unordered_map (hm) with the count of each character.

Character Frequency Count:

Character	Count
z	3
m	3
s	2
e	4
q	4
x	2
l	3
v	2
h	1
w	2
r	2
o	2
f	1
g	1
c	2
u	1
n	2
t	2
y	3
p	1
j	1
a	1
b	1

i	1			
Step 2: Find the Character with the Highest Frequency				
We iterate through the unordered_map (hm) and keep track of the character with the maximum frequency (mfc). Initially, mfc is set to the first character of the string, z.				
Iteration Over HashMap:				
Current Character	Frequency	hm[mfc]	Update mfc?	Updated mfc
z	3	3	No	z
m	3	3	No	z
s	2	3	No	z
e	4	3	Yes	e
q	4	4	No	e
x	2	4	No	e
l	3	4	No	e
v	2	4	No	e
h	1	4	No	e
w	2	4	No	e
r	2	4	No	e
o	2	4	No	e
f	1	4	No	e
g	1	4	No	e
c	2	4	No	e
u	1	4	No	e
n	2	4	No	e
t	2	4	No	e
y	3	4	No	e
p	1	4	No	e
j	1	4	No	e
a	1	4	No	e

Current Character	Frequency	hm[mfc]	Update mfc?	Updated mfc
b	1	4	No	e
i	1	4	No	e

Step 3: Output

The character with the highest frequency is **q**, appearing **4 times** in the string.

Output

Output:
q

K-Largest Elements in C++

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

void solve(int n, vector<int>& arr, int k) {
    priority_queue<int, vector<int>, greater<int>>
    pq; // Min-heap

    for (int i = 0; i < arr.size(); ++i) {
        if (i < k) {
            pq.push(arr[i]);
        } else {
            if (arr[i] > pq.top()) {
                pq.pop();
                pq.push(arr[i]);
            }
        }
    }

    vector<int> result;
    while (!pq.empty()) {
        result.push_back(pq.top());
        pq.pop();
    }

    for (int j = result.size() - 1; j >= 0; --j) {
        cout << result[j] << " ";
    }
    cout << endl;
}

int main() {
    vector<int> num = {44, -5, -2, 41, 12, 19, 21, -6};
    int k = 2;
    solve(num.size(), num, k);

    return 0;
}
```

Dry Run of `solve(n, arr, k)`

Input:

`arr = {44, -5, -2, 41, 12, 19, 21, -6};`
`k = 2;`

Step 1: Initialize Min-Heap

(`priority_queue`)

- Min-heap stores the **top k largest elements**.
- Initial heap (empty):** `pq = {}`

Step 2: Process First k Elements ($k = 2$)

Iteration	<code>arr[i]</code>	Heap After Push (<code>pq</code>)
i = 0	44	{44}
i = 1	-5	{-5, 44}

Step 3: Process Remaining Elements

Iteration	<code>arr[i]</code>	Compare With <code>pq.top()</code>	Action Taken	Heap After Update
i = 2	-2	-5 < -2	Pop -5, Push -2	{-2, 44}
i = 3	41	-2 < 41	Pop -2, Push 41	{41, 44}
i = 4	12	41 > 12	No Change	{41, 44}
i = 5	19	41 > 19	No Change	{41, 44}
i = 6	21	41 > 21	No Change	{41, 44}
i = 7	-6	41 > -6	No Change	{41, 44}

Step 4: Extract Elements from Min-Heap

- Extract elements in ascending order: {41, 44}
- Reverse order to print in descending: **44**
41

Output:
44 41

Merge k sorted elements in C++

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

struct Pair {
    int li; // List index
    int di; // Data index (current index in the list)
    int val; // Value at current index in the list

    Pair(int li, int di, int val) {
        this->li = li;
        this->di = di;
        this->val = val;
    }

    bool operator>(const Pair& other) const {
        return val > other.val;
    }
};

vector<int> mergeKSortedLists(vector<vector<int>>& lists) {
    vector<int> rv;

    // Min-heap priority queue
    priority_queue<Pair, vector<Pair>, greater<Pair>>
    pq;

    // Initialize the priority queue with the first
    // element from each list
    for (int i = 0; i < lists.size(); ++i) {
        if (!lists[i].empty()) {
            pq.push(Pair(i, 0, lists[i][0]));
        }
    }

    while (!pq.empty()) {
        Pair p = pq.top();
        pq.pop();

        // Add the current value to result vector
        rv.push_back(p.val);

        // Move to the next element in the same list
        p.di++;
        if (p.di < lists[p.li].size()) {
            p.val = lists[p.li][p.di];
            pq.push(p);
        }
    }

    return rv;
}

int main() {
    vector<vector<int>> lists = {
        {10, 20, 30, 40, 50},
        {5, 7, 9, 11, 19, 55, 57},
        {1, 2, 3}
    };
}
```

Dry Run of mergeKSortedLists(lists)

Input:

```
lists = {
    {10, 20, 30, 40, 50},
    {5, 7, 9, 11, 19, 55, 57},
    {1, 2, 3}
};
```

Step 1: Initialize Min-Heap (priority_queue)

- Min-heap stores **(value, list index, data index)** for sorting.
- Insert the first element of each list:**
 - (10, 0, 0) from list **0** ({10, 20, 30, 40, 50})
 - (5, 1, 0) from list **1** ({5, 7, 9, 11, 19, 55, 57})
 - (1, 2, 0) from list **2** ({1, 2, 3})

Step 2: Extract Minimum & Insert Next Element

Step	Extracted (Min)	Insert Next	Updated Heap
1	(1, 2, 0)	(2, 2, 1)	{(2,2,1), (5,1,0), (10,0,0)}
2	(2, 2, 1)	(3, 2, 2)	{(3,2,2), (5,1,0), (10,0,0)}
3	(3, 2, 2)	None (End)	{(5,1,0), (10,0,0)}
4	(5, 1, 0)	(7, 1, 1)	{(7,1,1), (10,0,0)}
5	(7, 1, 1)	(9, 1, 2)	{(9,1,2), (10,0,0)}
6	(9, 1, 2)	(11, 1, 3)	{(10,0,0), (11,1,3)}
7	(10, 0, 0)	(20, 0, 1)	{(11,1,3), (20,0,1)}
8	(11, 1, 3)	(19, 1, 4)	{(19,1,4), (20,0,1)}
9	(19, 1, 4)	(55, 1, 5)	{(20,0,1), (55,1,5)}
10	(20, 0, 1)	(30, 0, 2)	{(30,0,2), (55,1,5)}
11	(30, 0, 2)	(40, 0, 3)	{(40,0,3), (55,1,5)}
12	(40, 0, 3)	(50, 0, 4)	{(50,0,4), (55,1,5)}
13	(50, 0, 4)	None (End)	{(55,1,5)}
14	(55, 1, 5)	(57, 1, 6)	{(57,1,6)}
15	(57, 1, 6)	None (End)	{}

Final Merged List:

```
{1, 2, 3, 5, 7, 9, 10, 11, 19, 20, 30, 40, 50, 55, 57}
```

```
vector<int> mlist = mergeKSortedLists(lists);

for (int val : mlist) {
    cout << val << " ";
}
cout << endl;

return 0;
}
```

Output:

```
1 2 3 5 7 9 10 11 19 20 30 40 50 55 57
```

Subarray with 0 sum in C++

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

int ZeroSumSubarray(vector<int>& arr) {
    unordered_set<int> us;
    int prefix_sum = 0;
    us.insert(0); // Insert 0 initially to handle cases
    where the prefix_sum itself is zero
    for (int i = 0; i < arr.size(); ++i) {
        prefix_sum += arr[i];
        if (us.count(prefix_sum) > 0)
            return 1; // Found a subarray with sum
        zero
        us.insert(prefix_sum);
    }
    return 0; // No subarray with sum zero found
}

int main() {
    vector<int> arr = {5, 3, 9, -4, -6, 7, -1};
    cout << ZeroSumSubarray(arr) << endl;
    return 0;
}
```

Dry Run of `zeroSumSubarray(arr)`

Input:

`arr = {5, 3, 9, -4, -6, 7, -1};`

Step 1: Initialize Variables

- **Prefix Sum (`prefix_sum`) = 0**
- **Hash Set (`us`) = {0}** (We insert 0 initially to handle cases where the prefix sum itself is zero)

Step 2: Iterating Over the Array

Iteration	<code>arr[i]</code>	<code>prefix_sum</code> (cumulative)	<code>us</code> (hash set)	Check if <code>prefix_sum</code> exists in <code>us</code>
1	5	0 + 5 = 5	{0, 5}	No
2	3	5 + 3 = 8	{0, 5, 8}	No
3	9	8 + 9 = 17	{0, 5, 8, 17}	No
4	-4	17 - 4 = 13	{0, 5, 8, 17, 13}	No
5	-6	13 - 6 = 7	{0, 5, 8, 17, 13, 7}	No
6	7	7 + 7 = 14	{0, 5, 8, 17, 13, 7, 14}	No
7	-1	14 - 1 = 13	{0, 5, 8, 17, 13, 7, 14}	Yes (13 exists in set!)

Step 3: Return Result

- Since `prefix_sum = 13` already exists in

	<p>us, it means there exists a subarray with sum 0.</p> <ul style="list-style-type: none">• Return 1 (True).
--	--

Output:
1

Subarray with given sum in C++

```
#include <iostream>
#include <unordered_set>
using namespace std;
```

```
bool isSum(int arr[], int n, int sum) {
    unordered_set<int> s;
    int pre_sum = 0;
    for (int i = 0; i < n; i++) {
        if (pre_sum == sum) {
            return true;
        }
        pre_sum += arr[i];
        if (s.find(pre_sum - sum) != s.end()) {
            return true;
        }
        s.insert(pre_sum);
    }
    return false;
}
```

```
int main() {
    int arr[] = {5, 8, 6, 13, 3, -1};
    int sum = 22;
    int n = sizeof(arr) / sizeof(arr[0]);

    if (isSum(arr, n, sum)) {
        cout << "Subarray with sum " <<
        sum << " exists." << endl;
    } else {
        cout << "No subarray with sum " <<
        sum << " exists." << endl;
    }

    return 0;
}
```

Dry Run of `isSum()` Function

Input:

```
arr[] = {5, 8, 6, 13, 3, -1}
sum = 22
n = 6
```

Step 1: Initialize Variables

- Prefix Sum (`pre_sum`) = 0
- Hash Set (`s`) = {} (Empty initially)

Step 2: Iterating Over the Array

Iteration	<code>arr[i]</code>	<code>pre_sum</code> (cumulative)	<code>pre_sum - sum</code>	Check if <code>pre_sum - sum</code> exists in set	Update Hash Set
1	5	0 + 5 = 5	5 - 22 = -17	No	{5}
2	8	5 + 8 = 13	13 - 22 = -9	No	{5, 13}
3	6	13 + 6 = 19	19 - 22 = -3	No	{5, 13, 19}
4	13	19 + 13 = 32	32 - 22 = 10	No	{5, 13, 19, 32}
5	3	32 + 3 = 35	35 - 22 = 13	Yes (13 exists in set)	{5, 13, 19, 32, 35}
6	-1	35 + (-1) = 34	34 - 22 = 12	No	{5, 13, 19, 32, 35, 34}

Step 3: Return Result

- At iteration 5, when `pre_sum` = 35, `pre_sum - sum` = 13 is found in the hash set, which means there exists a subarray with a sum of 22.
- **Return true.**

Output:	Subarray with sum 22 exists.

Arithmetic Sequence in C++

```
#include <iostream>
#include <vector>
#include <unordered_set>
#include <algorithm>
#include <climits>

using namespace std;

bool isArithmeticSequence(const vector<int>& arr) {
    if (arr.size() <= 1) {
        return true;
    }

    int minValue = INT_MAX;
    int maxValue = INT_MIN;
    unordered_set<int> elements;

    for (int val : arr) {
        minValue = min(val, minValue);
        maxValue = max(val, maxValue);
        elements.insert(val);
    }

    int d = (maxValue - minValue) / (arr.size() - 1);

    for (size_t i = 0; i < arr.size(); ++i) {
        int ai = minValue + i * d;
        if (elements.find(ai) == elements.end()) {
            return false;
        }
    }

    return true;
}

int main() {
    vector<int> arr = {17, 9, 5, 29, 1, 25, 13, 37, 21, 33};
    cout << (isArithmeticSequence(arr) ? "true" :
"false") << endl;

    return 0;
}
```

Dry Run

Input:

`arr = {17, 9, 5, 29, 1, 25, 13, 37, 21, 33}`

Here is a step-by-step dry run of your C++ code, focusing on loop iterations and index-wise updates:

Step-by-Step Execution Table

First Loop (Finding `minVal`, `maxVal`, and Filling `unordered_set`)

Index (i)	Current arr[i]	Updated minValue	Updated maxValue	Updated elements
0	17	17	17	{17}
1	9	9	17	{9, 17}
2	5	5	17	{5, 9, 17}
3	29	5	29	{5, 9, 17, 29}
4	1	1	29	{1, 5, 9, 17, 29}
5	25	1	29	{1, 5, 9, 17, 25, 29}
6	13	1	29	{1, 5, 9, 13, 17, 25, 29}
7	37	1	37	{1, 5, 9, 13, 17, 25, 29, 37}
8	21	1	37	{1, 5, 9, 13, 17, 21, 25, 29, 37}
9	33	1	37	{1, 5, 9, 13, 17, 21, 25, 29, 33, 37}

- After this loop:
 - minValue = 1
 - maxValue = 37
 - elements = {1, 5, 9, 13, 17, 21, 25, 29, 33, 37}
 - d = (37 - 1) / (10 - 1) = 4

Second Loop (Verifying Arithmetic Sequence)

Index (i)	Expected Value <code>ai = minValue + i * d</code>	Check in elements	Result
0	1 + 0 * 4 = 1	✓ Found in {1, 5, 9, 13, 17, 21, 25, 29, 33, 37}	Continue
1	1 + 1 * 4 = 5	✓ Found	Continue
2	1 + 2 * 4 = 9	✓ Found	Continue
3	1 + 3 * 4 = 13	✓ Found	Continue
4	1 + 4 * 4 = 17	✓ Found	Continue
5	1 + 5 * 4 = 21	✓ Found	Continue
6	1 + 6 * 4 =	✓ Found	Continue

Index (i)	Expected Value $ai = minValue + i * d$	Check in elements	Result
	25		
7	1 + 7*4 = 29	✓ Found	Continue
8	1 + 8*4 = 33	✓ Found	Continue
9	1 + 9*4 = 37	✓ Found	Continue
<ul style="list-style-type: none"> Since all expected values exist in elements, the function returns true. 			
Output: true			

Array Pair Divisible by K in C++

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

void sol(const vector<int>& arr, int k) {
    unordered_map<int, int> remainderFreqMap;

    for (int val : arr) {
        int rem = val % k;
        remainderFreqMap[rem]++;
    }

    for (int val : arr) {
        int rem = val % k;

        if (rem == 0) {
            if (remainderFreqMap[rem] % 2 != 0)
                cout << "false" << endl;
            return;
        }
        else if (2 * rem == k) {
            if (remainderFreqMap[rem] % 2 != 0)
                cout << "false" << endl;
            return;
        }
        else {
            if (remainderFreqMap[rem] !=
                remainderFreqMap[k - rem]) {
                cout << "false" << endl;
                return;
            }
        }
    }

    cout << "true" << endl;
}

int main() {
    vector<int> arr = {22, 12, 45, 55, 65, 78, 88, 75};
    int k = 7;
    sol(arr, k);
    return 0;
}
```

Dry Run of sol(arr, k)

arr = {22, 12, 45, 55, 65, 78, 88, 75};
k = 7;

Step 1: Compute Remainders and Store in remainderFreqMap

For each element in arr, compute $\text{rem} = \text{val} \% \text{k}$ and store it in the map:

Value (val)	$\text{rem} = \text{val} \% 7$	remainderFreqMap (after insertion)
22	$22 \% 7 = 1$	{1: 1}
12	$12 \% 7 = 5$	{1: 1, 5: 1}
45	$45 \% 7 = 3$	{1: 1, 5: 1, 3: 1}
55	$55 \% 7 = 6$	{1: 1, 5: 1, 3: 1, 6: 1}
65	$65 \% 7 = 2$	{1: 1, 5: 1, 3: 1, 6: 1, 2: 1}
78	$78 \% 7 = 1$	{1: 2, 5: 1, 3: 1, 6: 1, 2: 1}
88	$88 \% 7 = 4$	{1: 2, 5: 1, 3: 1, 6: 1, 2: 1, 4: 1}
75	$75 \% 7 = 5$	{1: 2, 5: 2, 3: 1, 6: 1, 2: 1, 4: 1}

Final remainderFreqMap:

{1: 2, 5: 2, 3: 1, 6: 1, 2: 1, 4: 1}

Step 2: Validate Remainder Pairs

We check the conditions:

- If $\text{rem} == 0$, count should be even (not applicable here).
- If $2 * \text{rem} == \text{k}$, count should be even (not applicable here).
- Otherwise, $\text{remainderFreqMap}[\text{rem}]$ should match $\text{remainderFreqMap}[\text{k} - \text{rem}]$.

Value (val)	$\text{rem} = \text{val} \% 7$	Condition	Check
22	1	$\text{map}[1] == \text{map}[6]$	✗ 2 != 1

Since the condition fails, we print "false" and

Output: false	

Check anagram in C++

```
#include <iostream>
#include <unordered_map>
using namespace std;

bool solution(string s1, string s2) {
    unordered_map<char, int> map;

    // Count frequencies of characters in s1
    for (char ch : s1) {
        map[ch]++;
    }

    // Check characters in s2 against the frequency map
    for (char ch : s2) {
        if (map.find(ch) == map.end()) {
            return false; // Character not found in s1
        } else if (map[ch] == 1) {
            map.erase(ch); // Remove entry if frequency becomes zero
        } else {
            map[ch]--; // Decrement the count of the character
        }
    }

    // If map is empty, all characters from s1 and s2 match in frequency
    return map.empty();
}

int main() {
    string s1 = "pepcoding";
    string s2 = "codingpep";
    cout << boolalpha << solution(s1, s2) << endl;
}
```

Output: true

```
    return 0;
}
```

Dry Run for solution Function

Input:

- s1 = "pepcoding"
- s2 = "codingpep"

Step-by-Step Execution

Step 1: Count frequencies of characters in s1

Character (ch)	Frequency in map (map[ch])
'p'	2
'e'	1
'c'	1
'o'	1
'd'	1
'i'	1
'n'	1
'g'	1

Map after Step 1:

```
map = {'p': 2, 'e': 1, 'c': 1, 'o': 1, 'd': 1, 'i': 1, 'n': 1, 'g': 1}
```

Step 2: Process characters in s2

Character (ch)	Action Taken	Updated map
'c'	Found in map, decrement map['c']	{'p': 2, 'e': 1, 'o': 1, 'd': 1, 'i': 1, 'n': 1, 'g': 1}
'o'	Found in map, decrement map['o']	{'p': 2, 'e': 1, 'd': 1, 'i': 1, 'n': 1, 'g': 1}
'd'	Found in map, decrement map['d']	{'p': 2, 'e': 1, 'i': 1, 'n': 1, 'g': 1}
'i'	Found in map, decrement map['i']	{'p': 2, 'e': 1, 'n': 1, 'g': 1}
'n'	Found in map, decrement map['n']	{'p': 2, 'e': 1, 'g': 1}
'g'	Found in map, decrement map['g']	{'p': 2, 'e': 1}
'p'	Found in map, decrement map['p']	{'p': 1, 'e': 1}
'e'	Found in map, decrement	{'p': 1}

Character (ch)	Action Taken	Updated map
	map['e']	
'p'	Found in map, decrement map['p']	{}

Step 3: Final Check

- Is map empty?
Yes, map is empty, indicating all characters in s2 match the frequencies in s1.

Output:

true

Output:
true

Contiguous Array in C++

```
#include <iostream>
#include <unordered_map>
using namespace std;

int sol(int arr[], int n) {
    int ans = 0;
    unordered_map<int, int> map;
    map[0] = -1;
    int sum = 0;

    for (int i = 0; i < n; i++) {
        if (arr[i] == 0) {
            sum += -1;
        } else if (arr[i] == 1) {
            sum += +1;
        }

        if (map.find(sum) != map.end()) {
            int idx = map[sum];
            int len = i - idx;
            if (len > ans) {
                ans = len;
            }
        } else {
            map[sum] = i;
        }
    }
    return ans;
}

int main() {
    int arr[] = {0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 1, 1, 1};
    int n = sizeof(arr) / sizeof(arr[0]);
    cout << sol(arr, n) << endl; // Output: 10

    return 0;
}
```

Dry Run:

Given input:

```
int arr[] = {0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 1, 1, 1};
int n = sizeof(arr) / sizeof(arr[0]);
```

Step-by-Step Breakdown:

Initial Values:

- $\text{ans} = 0$ (stores the longest subarray length)
- $\text{map} = \{0: -1\}$ (maps cumulative sum to the first occurrence index)
- $\text{sum} = 0$ (initial cumulative sum)

Iteration by Iteration Walkthrough:

i	arr[i]	sum (cumulative sum)	map (sum → index)	Length (len)	Updated ans
0	0	-1	{0: -1, -1: 0}	$0 - (-1) = 1$	1
1	0	-2	{0: -1, -1: 0, -2: 1}	$1 - (-1) = 2$	2
2	1	-1	{0: -1, -1: 0, -2: 1}	$2 - 0 = 2$	2
3	0	-2	{0: -1, -1: 0, -2: 1}	$3 - 1 = 2$	2
4	1	-1	{0: -1, -1: 0, -2: 1}	$4 - 0 = 4$	4
5	0	-2	{0: -1, -1: 0, -2: 1}	$5 - 1 = 4$	4
6	1	-1	{0: -1, -1: 0, -2: 1}	$6 - 0 = 6$	6
7	1	0	{0: -1, -1: 0, -2: 1}	$7 - (-1) = 8$	8
8	0	-1	{0: -1, -1: 0, -2: 1}	$8 - 0 = 8$	8
9	0	-2	{0: -1, -1: 0, -2: 1}	$9 - 1 = 8$	8
10	1	-1	{0: -1, -1: 0, -2: 1}	$10 - 0 = 10$	10
11	1	0	{0: -1, -1: 0, -2: 1}	$11 - (-1) = 12$	12

			-2: 1}		
12	1	1	{0: -1, -1: 0, -2: 1}	12 - (-1) = 14	14

Correct Analysis:

- The **longest subarray** with equal numbers of 0s and 1s spans from index 2 to 11 (inclusive), making the subarray length **12**.

Final Output:

12

Output:
12

Count of Subarrays Having Sum Equal to K in C++

```
#include <iostream>
#include <unordered_map>
#include <vector>

using namespace std;

int solution(vector<int>& arr, int target) {
    int ans = 0;
    unordered_map<int, int> map;
    map[0] = 1; // Initialize with sum 0 having
    count 1
    int sum = 0;

    for (int i = 0; i < arr.size(); i++) {
        sum += arr[i];
        if (map.find(sum - target) != map.end()) {
            ans += map[sum - target];
        }
        map[sum]++;
    }

    return ans;
}

int main() {
    vector<int> arr = {1, 1, 1};
    int target = 2;
    cout << solution(arr, target) << endl; // Output: 2
    return 0;
}
```

Dry Run for Input:

```
vector<int> arr = {1, 1, 1};
int target = 2;
```

Initial Values:

- $\text{ans} = 0$
- $\text{map} = \{0: 1\}$ (since $\text{map}[0] = 1$ initially)
- $\text{sum} = 0$

Iteration Breakdown:

i	arr[i]	sum (cumulative sum)	sum - target	map[sum - target]	ans	map (updated)
0	1	1	$1 - 2 = -1$	Not found	0	{0: 1, 1: 1}
1	1	2	$2 - 2 = 0$	$\text{map}[0] = 1$ (found)	1	{0: 1, 1: 1, 2: 1}
2	1	3	$3 - 2 = 1$	$\text{map}[1] = 1$ (found)	2	{0: 1, 1: 2, 2: 1, 3: 1}

Explanation of each iteration:

- At $i = 0$:
 - $\text{arr}[0] = 1$
 - $\text{sum} = 1$
 - We check if $\text{sum} - \text{target} = 1 - 2 = -1$ is in map. It is **not**.
 - We update the map with $\text{map}[1]++$, so map = {0: 1, 1: 1}.
- At $i = 1$:
 - $\text{arr}[1] = 1$
 - $\text{sum} = 2$
 - We check if $\text{sum} - \text{target} = 2 - 2 = 0$ is in map. It is (map[0] = 1), so we add 1 to ans (i.e., $\text{ans} += 1$).
 - We update the map with $\text{map}[2]++$, so map = {0: 1, 1: 1, 2: 1}.
- At $i = 2$:
 - $\text{arr}[2] = 1$
 - $\text{sum} = 3$
 - We check if $\text{sum} - \text{target} = 3 - 2 = 1$ is in map. It is (map[1] = 1), so we add 1 to ans (i.e., $\text{ans} += 1$).
 - We update the map with $\text{map}[3]++$, so map = {0: 1, 1: 2, 2: 1, 3: 1}.

Final Output:

- The total number of subarrays whose sum equals target = 2 is **2**.

Output:

2

Count Of Subarrays With Equal 0 and 1 in C++

```
#include <iostream>
#include <unordered_map>
#include <vector>

using namespace std;

int solution(vector<int>& arr) {
    unordered_map<int, int> map;
    int ans = 0;
    map[0] = 1; // Initialize with sum 0 having
    count 1
    int sum = 0;

    for (int val : arr) {
        // Treat 0 as -1 for sum calculation
        if (val == 0) {
            sum += -1;
        } else {
            sum += 1;
        }

        if (map.find(sum) != map.end()) {
            ans += map[sum];
            map[sum]++;
        } else {
            map[sum] = 1;
        }
    }

    return ans;
}

int main() {
    vector<int> arr = {0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 1,
    1};
    cout << solution(arr) << endl; // Output the
    result

    return 0;
}
```

Dry Run for Input:

vector<int> arr = {0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 1, 1, 1};

Initial Values:

- ans = 0
- map = {0: 1}
- sum = 0

Iteration Breakdown:

i	arr[i]	sum (cumulative sum)	map[sum]	ans (after update)	map (updated)
0	0	-1	map[-1] = 0	0	{0: 1, -1: 1}
1	0	-2	map[-2] = 0	0	{0: 1, -1: 1, -2: 1}
2	1	-1	map[-1] = 1	1	{0: 1, -1: 2, -2: 1}
3	0	-2	map[-2] = 1	1	{0: 1, -1: 2, -2: 2}
4	1	-1	map[-1] = 2	3	{0: 1, -1: 3, -2: 2}
5	0	-2	map[-2] = 2	3	{0: 1, -1: 3, -2: 3}
6	1	-1	map[-1] = 3	6	{0: 1, -1: 4, -2: 3}
7	1	0	map[0] = 1	7	{0: 2, -1: 4, -2: 3}
8	0	-1	map[-1] = 4	11	{0: 2, -1: 5, -2: 3}
9	0	-2	map[-2] = 3	14	{0: 2, -1: 5, -2: 4}
10	1	-1	map[-1] = 5	19	{0: 2, -1: 6, -2: 4}
11	1	0	map[0] = 2	21	{0: 3, -1: 6, -2: 4}
12	1	1	map[1] = 0	24	{0: 3, -1: 6, -2: 4, 1: 1}

Output:

24

Count Of Zeros Sum Subarray in C++

```
#include <iostream>
#include <unordered_map>
#include <vector>

using namespace std;

int sol(const vector<int>& arr) {
    int count = 0;
    unordered_map<int, int> map;
    int sum = 0;
    map[0] = 1;

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

        if (map.find(sum) != map.end()) {
            count += map[sum];
            map[sum]++;
        } else {
            map[sum] = 1;
        }
    }

    return count;
}

int main() {
    vector<int> arr = {2, 8, -3, -5, 2, -4, 6, 1, 2, 1, -3, 4};
    int result = sol(arr);
    cout << result << endl;
    return 0;
}
```

Dry Run:

Initial Values:

- **count** = 0
- **map** = {0: 1}
- **sum** = 0

Iteration Breakdown:

i	arr[i]	sum (cumulative sum)	map[sum]	count (after update)	map (updated)
0	2	2	map[2] = 0	0	{0: 1, 2: 1}
1	8	10	map[10] = 0	0	{0: 1, 2: 1, 10: 1}
2	-3	7	map[7] = 0	0	{0: 1, 2: 1, 10: 1, 7: 1}
3	-5	2	map[2] = 1	1	{0: 1, 2: 2, 10: 1, 7: 1}
4	2	4	map[4] = 0	1	{0: 1, 2: 2, 10: 1, 7: 1, 4: 1}
5	-4	0	map[0] = 1	2	{0: 2, 2: 2, 10: 1, 7: 1, 4: 1}
6	6	6	map[6] = 0	2	{0: 2, 2: 2, 10: 1, 7: 1, 4: 1, 6: 1}
7	1	7	map[7] = 1	3	{0: 2, 2: 2, 10: 1, 7: 2, 4: 1, 6: 1}
8	2	9	map[9] = 0	3	{0: 2, 2: 2, 10: 1, 7: 2, 4: 1, 6: 1, 9: 1}
9	1	10	map[10] = 1	4	{0: 2, 2: 2, 10: 2, 7: 2, 4: 1, 6: 1, 9: 1}
10	-3	7	map[7] = 2	6	{0: 2, 2: 2, 10: 2, 7: 3, 4: 1, 6: 1, 9: 1}
11	4	11	map[11] = 0	6	{0: 2, 2: 2, 10: 2, 7: 3, 4: 1, 6: 1, 9: 1, 11: 1}

Final Values:

- **count** = 6
- **map** = {0: 2, 2: 2, 10: 2, 7: 3, 4: 1, 6: 1, 9: 1, 11: 1}

	<p>Output:</p> <p>The total number of subarrays with sum equal to 0 is 6.</p> <p>Final Output:</p> <p>6</p>
--	--

Output:
6

Distinct Elements Window of Size K in C++

```
#include <iostream>
#include <vector>
#include <unordered_map>
#include <deque>

using namespace std;

vector<int> distinctElementsInWindow(const vector<int>& arr, int k) {
    vector<int> result;
    unordered_map<int, int> frequencyMap;
    int n = arr.size();
    int i = 0;

    // Initialize the frequency map for the first window
    for (i = 0; i < k - 1; ++i) {
        frequencyMap[arr[i]]++;
    }

    for (int j = -1; i < n; ++i, ++j) {
        // Add the next element (i-th element) to the
        // frequency map
        frequencyMap[arr[i]]++;

        // Record the number of distinct elements in the
        // current window
        result.push_back(frequencyMap.size());

        // Remove the (j-th element) as the window slides
        if (j >= 0) {
            if (frequencyMap[arr[j]] == 1) {
                frequencyMap.erase(arr[j]);
            } else {
                frequencyMap[arr[j]]--;
            }
        }
    }

    return result;
}

int main() {
    vector<int> arr = {2, 5, 5, 6, 3, 2, 3, 2, 4, 5, 2, 2, 2, 2,
3, 6};
    int k = 4;
    vector<int> result =
distinctElementsInWindow(arr, k);

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

    return 0;
}
```

Dry Run:

Initialize:

- **arr** = [2, 5, 5, 6, 3, 2, 3, 2, 4, 5, 2, 2, 2, 2, 3, 6]
- **k** = 4
- **frequencyMap** = {} (Empty at the start)
- **result** = [] (Empty at the start)

Step-by-Step Iteration:

i	arr[i]	frequencyMap (Updated)	Distinct Elements	result (after update)	j
0	2	{2: 1}	1	[]	-1
1	5	{2: 1, 5: 1}	2	[]	0
2	5	{2: 1, 5: 2}	2	[]	1
3	6	{2: 1, 5: 2, 6: 1}	3	[3]	2
4	3	{2: 1, 5: 1, 6: 1, 3: 1}	4	[3, 4]	3
5	2	{2: 2, 5: 1, 6: 1, 3: 1}	4	[3, 4, 4]	4
6	3	{2: 2, 5: 1, 6: 1, 3: 2}	3	[3, 4, 4, 3]	5
7	2	{2: 3, 5: 1, 6: 1, 3: 2}	3	[3, 4, 4, 3, 3]	6
8	4	{2: 3, 5: 1, 6: 1, 3: 2, 4: 1}	4	[3, 4, 4, 3, 3, 4]	7
9	5	{2: 3, 5: 2, 6: 1, 3: 2, 4: 1}	4	[3, 4, 4, 3, 3, 4, 4]	8
10	2	{2: 4, 5: 2, 6: 1, 3: 2, 4: 1}	3	[3, 4, 4, 3, 3, 4, 4, 3]	9
11	2	{2: 5, 5: 2, 6: 1, 3: 2, 4: 1}	2	[3, 4, 4, 3, 3, 4, 4, 3, 3]	10
12	2	{2: 6, 5: 2, 6: 1, 3: 2, 4: 1}	1	[3, 4, 4, 3, 3, 4, 4, 3, 3, 2]	11
13	2	{2: 7, 5: 2, 6: 1, 3: 2, 4: 1}	1	[3, 4, 4, 3, 3, 4, 4, 3, 3, 2, 2]	12
14	3	{2: 7, 5: 2, 6: 1, 3: 3, 4: 1}	2	[3, 4, 4, 3, 3, 4, 4, 3, 3, 2, 2]	13

				3]	
15	6	{2: 7, 5: 2, 6: 2, 3: 3, 4: 1}	3	[3, 4, 4, 3, 3, 4, 4, 3, 3, 2, 2, 3, 3]	14

Final Result:

The output is the list of distinct elements in each sliding window of size k as the window slides across the array:

Output:

3 4 4 4 3 3 4 4 3 3 2 2 3

Output:

3 4 4 4 3 3 4 4 3 3 2 2 3

Employees Under Manager in C++

```
#include <iostream>
#include <unordered_map>
#include <unordered_set>
#include <string>

using namespace std;

int getSize(unordered_map<string, unordered_set<string>>& tree, const string& manager, unordered_map<string, int>& result) {
    if (tree.find(manager) == tree.end()) {
        result[manager] = 0;
        return 1;
    }
    int size = 0;
    for (const string& employee : tree[manager]) {
        int currentSize = getSize(tree, employee, result);
        size += currentSize;
    }
    result[manager] = size;
    return size + 1;
}

void findCount(unordered_map<string, string>& map) {
    unordered_map<string, unordered_set<string>> tree;
    string ceo = "";

    for (const auto& entry : map) {
        string employee = entry.first;
        string manager = entry.second;

        if (manager == employee) {
            ceo = manager;
        } else {
            tree[manager].insert(employee);
        }
    }

    unordered_map<string, int> result;
    getSize(tree, ceo, result);

    for (const auto& entry : result) {
        cout << entry.first << " " << entry.second << endl;
    }
}

int main() {
    unordered_map<string, string> map;
    map["A"] = "C";
    map["B"] = "C";
    map["C"] = "F";
    map["D"] = "E";
    map["E"] = "F";
    map["F"] = "F";
}
```

Step 1: Construct tree and Identify CEO

- Input mapping:

A -> C
 B -> C
 C -> F
 D -> E
 E -> F
 F -> F (CEO identified)

- Constructing tree:

C -> {A, B}
 F -> {C, E}
 E -> {D}

- CEO Identified: F

Step 2: Recursive Calls of getSize(tree, manager, result)

Function Call	Processing Employee Set	Recursive Calls	Result Updates (result[manager])	Return Value
getSize(tree, "F", result)	{C, E}	getSize(tree, "C"), getSize(tree, "E")	F → 5	6
getSize(tree, "C", result)	{A, B}	getSize(tree, "A"), getSize(tree, "B")	C → 2	3
getSize(tree, "A", result)	{}	(Base Case)	A → 0	1
getSize(tree, "B", result)	{}	(Base Case)	B → 0	1
getSize(tree, "E", result)	{D}	getSize(tree, "D")	E → 1	2
getSize(tree, "D", result)	{}	(Base Case)	D → 0	1

Step 3: Output Values

Final result map:

mathematica
 CopyEdit
 A → 0
 B → 0
 C → 2

```
    findCount(map);  
  
    return 0;  
}
```

D → 0
E → 1
F → 5

Final Output

A 0
B 0
C 2
D 0
E 1
F 5

Output:

F 5
E 1
B 0
A 0
D 0
C 2

Equivalent Subarrays in C++

```
#include <iostream>
#include <unordered_map>
#include <unordered_set>
#include <vector>

using namespace std;

int main() {
    int ans = 0;
    vector<int> arr = {2, 1, 3, 2, 3};
    unordered_set<int> set;

    // Insert unique elements into the set
    for (int i = 0; i < arr.size(); i++) {
        set.insert(arr[i]);
    }

    int k = set.size();
    int i = -1;
    int j = -1;
    unordered_map<int, int> map;

    while (true) {
        bool f1 = false;
        bool f2 = false;

        // Expand the window until all
        unique elements are covered
        while (i < arr.size() - 1) {
            f1 = true;
            i++;
            map[arr[i]] = map[arr[i]] + 1; // Add current element to the map
            if (map.size() == k) { // If all
                unique elements are covered
                    ans += arr.size() - i; // Add the
                    number of valid subarrays ending at
                    index i
                    break;
            }
        }

        // Slide the window to the right
        until the uniqueness condition is
        violated
        while (j < i) {
            f2 = true;
            j++;
            if (map[arr[j]] == 1) {
                map.erase(arr[j]); // Remove
                element from map if its count is
                reduced to 0
            } else {
                map[arr[j]] = map[arr[j]] - 1; // Decrease the count of the element
            }

            // If the map size matches k, add
            the number of valid subarrays again
            if (map.size() == k) {
                ans += arr.size() - i;
            }
        }
    }
}
```

Step 1: Initializing Variables

- Input Array:** {2, 1, 3, 2, 3}
- Unique Elements (set):**

{2, 1, 3} \rightarrow k = 3 (total unique elements)

- Pointers:**

i = -1, j = -1
ans = 0
map = {} (empty frequency map)

Step 2: Expanding the Window (Outer `while` Loop)

Expanding i Until `map.size() == k`

i	arr[i]	map (after update)	map.size()	Condition <code>map.size() == k?</code>
0	2	{2: 1}	1	✗
1	1	{2: 1, 1: 1}	2	✗
2	3	{2: 1, 1: 1, 3: 1}	3	✓ → Add arr.size() - i = 5 - 2 = 3 to ans

- ans = 3**

Step 3: Contracting j Until `map.size() < k`

j	arr[j]	map (after update)	map.size()	Condition <code>map.size() == k?</code>	ans Update
0	2	{2: 0, 1: 1, 3: 1} \rightarrow removed 2	2	✗	Break

Step 4: Continue Expanding i

i	arr[i]	map (after update)	map.size()	Condition <code>map.size() == k?</code>	ans Update
3	2	{1: 1, 3: 1, 2: 1}	3	✓	Add arr.size() - i = 5 - 3 = 2
New ans	3 + 2 = 5				

```

        } else {
            break;
        }

    // If both windows cannot be
    // expanded or contracted further, break
    // the loop
    if (!f1 && !f2) {
        break;
    }

    // Print the total number of
    // equivalent subarrays
    cout << ans << endl;

    return 0;
}

```

Step 5: Contracting j Again

j	$\text{arr}[j]$	$\text{map} \text{ (after update)}$	map.size()	Condition $\text{map.size()} == k?$	ans Update
1	1	{1: 0, 3: 1, 2: 1} \rightarrow removed 1	2	X	Break

Step 6: Continue Expanding i

i	$\text{arr}[i]$	$\text{map} \text{ (after update)}$	map.size()	Condition $\text{map.size()} == k?$	ans Update
4	3	{3: 2, 2: 1}	2	X	No update

Final Output

5

Summary of Valid Subarrays

- The total number of subarrays containing all **3 distinct elements {1, 2, 3}** is **5**.

Output:-
0

First Non Repeating Character in C++	
<pre>#include <iostream> #include <string> #include <unordered_map> using namespace std; int sol(string s) { unordered_map<char, int> fmap; // Build frequency map for (char c : s) { fmap[c]++; } // Find first non-repeating character for (int i = 0; i < s.length(); i++) { char ch = s[i]; if (fmap[ch] == 1) { return i; } } return -1; // If no non-repeating character found } int main() { string s = "abbcaddecfab"; cout << sol(s) << endl; return 0; }</pre>	<p>Input:</p> <p>s = "abbcaddecfab"</p> <p>Step 1 - Build Frequency Map:</p> <p>The frequency map (fmap) will look like this:</p> <ul style="list-style-type: none"> • 'a' → 2 • 'b' → 3 • 'c' → 2 • 'd' → 2 • 'e' → 2 • 'f' → 1 <p>Step 2 - Find First Non-Repeating Character:</p> <p>We now iterate through the string and check the frequency of each character:</p> <ol style="list-style-type: none"> 1. For index 0: s[0] = 'a' → frequency of 'a' is 2 (repeated). 2. For index 1: s[1] = 'b' → frequency of 'b' is 3 (repeated). 3. For index 2: s[2] = 'b' → frequency of 'b' is 3 (repeated). 4. For index 3: s[3] = 'c' → frequency of 'c' is 2 (repeated). 5. For index 4: s[4] = 'a' → frequency of 'a' is 2 (repeated). 6. For index 5: s[5] = 'd' → frequency of 'd' is 2 (repeated). 7. For index 6: s[6] = 'd' → frequency of 'd' is 2 (repeated). 8. For index 7: s[7] = 'e' → frequency of 'e' is 2 (repeated). 9. For index 8: s[8] = 'c' → frequency of 'c' is 2 (repeated). 10. For index 9: s[9] = 'f' → frequency of 'f' is 1 (non-repeating). <p>Now, the first non-repeating character is 'f', which appears at index 7, not index 9.</p> <p>Conclusion:</p> <ul style="list-style-type: none"> • The first non-repeating character in the string "abbcaddecfab" is 'f', which appears at index 7.
Output:	7

Isomorphic Strings in C++

```
#include <iostream>
#include <string>
#include <unordered_map>

using namespace std;

bool iso(string s, string t) {
    if (s.length() != t.length()) {
        return false;
    }

    unordered_map<char, char> map1; // Maps
    characters from s to t
    unordered_map<char, bool> map2; //
    Tracks characters used in t

    for (int i = 0; i < s.length(); i++) {
        char ch1 = s[i];
        char ch2 = t[i];

        if (map1.count(ch1) > 0) { // If ch1 is
            already mapped
            if (map1[ch1] != ch2) { // Check if
                mapping is consistent
                return false;
            }
        } else { // ch1 has not been mapped yet
            if (map2.count(ch2) > 0) { // If ch2 is
                already mapped by another character in s
                return false;
            } else { // Create new mapping
                map1[ch1] = ch2;
                map2[ch2] = true;
            }
        }
    }

    return true;
}

int main() {
    string s1 = "abc";
    string s2 = "cad";
    cout << boolalpha << iso(s1, s2) << endl; //
    Output: true

    return 0;
}
```

Output:
true

Step 1: Initialize Variables

- Input Strings:** $s = \text{"abc"}$, $t = \text{"cad"}$
- Maps Used:**
 - $\text{map1} \rightarrow$ Stores mapping from s to t
 - $\text{map2} \rightarrow$ Tracks characters already mapped in t

Step 2: Iterating Through s and t

Index (i)	s[i]	t[i]	map1 (s → t)	map2 (used t characters)	Check for Conflict?	Result
0	'a'	'c'	{ a → { c → true } }		No	Continue
1	'b'	'a'	{ a → { c → true, b → a } }	{ c → true }	No	Continue
2	'c'	'd'	{ a → { c → true, b → a, c → d } }	{ c → true, a → true, d → true }	No	Continue

Step 3: Return Result

- Since no conflicts were found, return `true`.

Final Output

true

Itinerary in C++

```
#include <iostream>
#include <unordered_map>
#include <string>

using namespace std;

int main() {
    unordered_map<string, string> map;
    map["Chennai"] = "Banglore";
    map["Bombay"] = "Delhi";
    map["Goa"] = "Chennai";
    map["Delhi"] = "Goa";

    // Create a hashmap to mark if a city is a potential source
    unordered_map<string, bool> psrc;
    for (auto it = map.begin(); it != map.end(); ++it) {
        string src = it->first;
        string dest = it->second;

        psrc[dest] = false; // Destination city cannot be a source
        if (psrc.find(src) == psrc.end()) {
            psrc[src] = true; // Source city if it is not a destination in the map
        }
    }

    string src = "";
    for (auto it = psrc.begin(); it != psrc.end(); ++it) {
        if (it->second == true) {
            src = it->first;
            break;
        }
    }

    // Print the itinerary
    while (true) {
        if (map.find(src) != map.end()) {
            cout << src << " -> ";
            src = map[src];
        } else {
            cout << src << ". ";
            break;
        }
    }
}

return 0;
}
```

Step 1: Initialize Data

- **Input Map (City Routes):**

Chennai	→	Banglore
Bombay	→	Delhi
Goa	→	Chennai
Delhi	→	Goa

- **Creating `psrc` (Potential Source Map):**
 - Initially Empty

Step 2: Mark Potential Sources (`psrc` Construction)

Iteration	Source (src)	Destination (dest)	Updated <code>psrc</code> (Potential Source Map)
1	Chennai	Banglore	{ Banglore → false, Chennai → true }
2	Bombay	Delhi	{ Banglore → false, Chennai → true, Delhi → false, Bombay → true }
3	Goa	Chennai	{ Banglore → false, Chennai → false, Delhi → false, Bombay → true, Goa → true }
4	Delhi	Goa	{ Banglore → false, Chennai → false, Delhi → false, Bombay → true, Goa → false }

- **Final `psrc` Map:**

Bombay → true (Only Source)
 Banglore → false
 Chennai → false
 Delhi → false
 Goa → false

Step 3: Find the Start City

- The only city with `true` in `psrc` is "**Bombay**".
- Start `src = "Bombay"`.

Step 4: Print the Itinerary

Iteration	Current <code>src</code>	Next City (<code>map[src]</code>)	Printed Output
1	Bombay	Delhi	Bombay ->
2	Delhi	Goa	Delhi ->
3	Goa	Chennai	Goa ->
4	Chennai	Banglore	Chennai ->
5	Banglore	(Not Found)	Banglore.

Final Output

Bombay -> Delhi -> Goa -> Chennai ->
Banglore.

Output:

Bombay -> Delhi -> Goa -> Chennai -> Banglore.

Largest Subarray with 0sum in C++

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

using namespace std;

int largest2(vector<int> arr, int n) {
    int max_len = 0;
    for (int i = 0; i < n; i++) {
        int sum = 0;
        for (int j = i; j < n; j++) {
            sum += arr[j];
            if (sum == 0) {
                max_len = max(max_len, j - i + 1);
            }
        }
    }
    return max_len;
}

int largest3(vector<int> arr, int n) {
    map<int, int> mapp;
    mapp[0]=-1;
    int sum=0;
    int ans=0;
    for (int i = 0; i < n; i++) {
        sum+=arr[i];
        if(mapp.find(sum)!=mapp.end()){
            auto it=mapp[sum];
            ans=max(ans,i- it);
        }
        else{
            mapp[sum]=i;
        }
    }
    return ans;
}

int
largestSubarrayWithZeroSum(vector<int> & arr) {
    unordered_map<int, int> hm; // Maps
    sum to index
    int sum = 0;
    int max_len = 0;

    hm[0] = -1; // Initialize to handle the case
    where sum becomes 0 at the start

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

        if (hm.find(sum) != hm.end()) {
            int len = i - hm[sum];
            if (len > max_len) {
                max_len = len;
            }
        } else {
            hm[sum] = i;
        }
    }
}
```

Step 1: Understanding the Problem

- We need to find the **largest subarray with sum = 0**.
- The input array is:

$$\{2, 8, -3, -5, 2, -4, 6, 1, 2, 1, -3, 4\}$$
- The program runs **three different implementations** for this:
 1. **largestSubarrayWithZeroSum()** → Optimized using `unordered_map`.
 2. **largest2()** → Brute-force approach.
 3. **largest3()** → Using `map`.

Step 2: Dry Run for largestSubarrayWithZeroSum() (Optimized Hashing Approach)

Index (i)	arr[i]	Sum	hm (Sum → Index)	Max Length (max_len)
0	2	2	{0:-1, 2:0}	0
1	8	10	{0:-1, 2:0, 10:1}	0
2	-3	7	{0:-1, 2:0, 10:1, 7:2}	0
3	-5	2	Found 2 at index 0 → 3 - 0 = 3	3
4	2	4	{0:-1, 2:0, 10:1, 7:2, 4:4}	3
5	-4	0	Found 0 at index -1 → 5 - (-1) = 6	6
6	6	6	{0:-1, 2:0, 10:1, 7:2, 4:4, 6:6}	6
7	1	7	Found 7 at index 2 → 7 - 2 = 5	6
8	2	9	{0:-1, 2:0, 10:1, 7:2, 4:4, 6:6, 9:8}	6
9	1	10	Found 10 at index 1 → 9 - 1 = 8	8
10	-3	7	Found 7 at index 2 → 10 - 2 = 8	8

```

    }

    return max_len;
}

int main() {
    vector<int> arr = {2, 8, -3, -5, 2, -4, 6, 1, 2,
1, -3, 4};
    int max_length =
largestSubarrayWithZeroSum(arr);
    cout << max_length << endl; // Output: 5

    int n=arr.size();
    int res=largest2(arr,n);
    cout<<res<<endl;

    int res3=largest3(arr,n);
    cout<<res3<<endl;

    return 0;
}

```

11	4	11	{0:-1, 2:0, 10:1, 7:2, 4:4, 6:6, 9:8, 11:11}	8
----	---	----	---	---

Final Output of largestSubarrayWithZeroSum()

→ 8

Step 3: Dry Run for largest2() (Brute-force approach)

- **Time Complexity:** $O(N^2) \rightarrow$ Iterates over all possible subarrays.
- Iterates over each possible subarray and calculates its sum.

i	j	Subarray	Sum	Max Length (max_len)
0	1	{2, 8}	10	0
0	2	{2, 8, -3}	7	0
0	3	{2, 8, -3, -5}	2	0
0	5	{2, 8, -3, -5, 2, -4}	0	6
1	5	{8, -3, -5, 2, -4}	0	6
3	9	{-5, 2, -4, 6, 1, 2, 1}	0	7
1	9	{8, -3, -5, 2, -4, 6, 1, 2, 1}	0	8

Final Output of largest2() → 8

Step 4: Dry Run for largest3() (Map-based approach)

- Similar to largestSubarrayWithZeroSum(), but uses $\text{map} < \text{int}, \text{int} >$ instead of $\text{unordered_map} < \text{int}, \text{int} >$.

Index (i)	arr[i]	Sum	mapp (Sum → Index)	Max Length (ans)
0	2	2	{0:-1, 2:0}	0
1	8	10	{0:-1, 2:0, 10:1}	0
2	-3	7	{0:-1, 2:0, 10:1, 7:2}	0
3	-5	2	Found 2 at index 0 → 3 - 0 = 3	3
4	2	4	{0:-1, 2:0, 10:1, 7:2,	3

Index (i)	arr[i]	Sum	mapp (Sum → Index)	Max Length (ans)
			4:4}	
5	-4	0	Found 0 at index -1 → 5 - (-1) = 6	6
6	6	6	{0:-1, 2:0, 10:1, 7:2, 4:4, 6:6}	6
7	1	7	Found 7 at index 2 → 7 - 2 = 5	6
8	2	9	{0:-1, 2:0, 10:1, 7:2, 4:4, 6:6, 9:8}	6
9	1	10	Found 10 at index 1 → 9 - 1 = 8	8
10	-3	7	Found 7 at index 2 → 10 - 2 = 8	8
11	4	11	{0:-1, 2:0, 10:1, 7:2, 4:4, 6:6, 9:8, 11:11}	8

Final Output of largest3() → 8

Final Outputs

Function

Approach

Output

largestSubarrayWithZeroSum() Hashing
(unordered_map) 8

largest2() Brute-force
(O(N²)) 8

largest3() Hashing (map) 8

Output:

8
8
8

Largest Subarray With Contiguous Elements in C++

```
#include <iostream>
#include <unordered_set>
#include <vector>

using namespace std;

int solution(vector<int>& arr) {
    int ans = 0;

    for (int i = 0; i < arr.size() - 1; i++) {
        int min_val = arr[i];
        int max_val = arr[i];
        unordered_set<int> contiguous_set;

        contiguous_set.insert(arr[i]);

        for (int j = i + 1; j < arr.size(); j++) {
            if (contiguous_set.find(arr[j]) != contiguous_set.end()) {
                break; // If duplicate found, break the loop
            }
        }

        contiguous_set.insert(arr[j]);
        min_val = min(min_val, arr[j]);
        max_val = max(max_val, arr[j]);

        if (max_val - min_val == j - i) {
            int len = j - i + 1;
            if (len > ans) {
                ans = len;
            }
        }
    }

    return ans;
}

int main() {
    vector<int> arr = {10, 12, 11};
    cout << solution(arr) << endl; // Output: 3

    return 0;
}
```

Understanding the Problem

- The function `solution(arr)` finds the length of the **longest contiguous subarray** where all elements are **distinct and consecutive**.
 - A contiguous subarray is valid if:
- $$\text{max_val} - \text{min_val} = j - i$$
- Example Input:** {10, 12, 11}
 - Expected Output:** 3 (as {10, 12, 11} forms a valid contiguous subarray)

Step-by-Step Dry Run

Outer Loop (i)	Inner Loop (j)	Subarray	min_val	max_val	max_val - min_val	j	i	Valid?	Current ans
0	0	{10}	10	10	0	0	0	✓	1
0	1	{10, 12}	10	12	2	1	0	✗	1
0	2	{10, 12, 11}	10	12	2	2	0	✓	3
1	1	{12}	12	12	0	0	0	✓	3
1	2	{12, 11}	11	12	1	1	0	✓	3
2	2	{11}	11	11	0	0	0	✓	3

Final Output: 3

```
contiguous_set.insert(arr[j]);
min_val =
min(min_val, arr[j]);
max_val =
max(max_val, arr[j]);
```

```
if (max_val - min_val
== j - i) {
    int len = j - i + 1;
    if (len > ans) {
        ans = len;
    }
}
return ans;
}
```

```
int main() {
    vector<int> arr = {10, 12,
11};
    cout << solution(arr) <<
endl; // Output: 3
```

```
return 0;
}
```

Output:
3

Longest Substring With At Most K Unique Characters in C++

```
#include <iostream>
#include <string>
#include <unordered_map>

class LongestSubstringWithAtMostKUniqueCharacters {
public:
    static int sol(const std::string& str, int k) {
        int ans = 0;
        int i = -1;
        int j = -1;
        std::unordered_map<char, int> map;

        while (true) {
            bool f1 = false;
            bool f2 = false;

            while (i < static_cast<int>(str.length()) - 1) {
                f1 = true;
                i++;
                char ch = str[i];
                map[ch]++;
                if (map.size() <= k) {
                    int len = i - j;
                    if (len > ans) {
                        ans = len;
                    }
                } else {
                    break;
                }
            }

            while (j < i) {
                f2 = true;
                j++;
                char ch = str[j];
                if (map[ch] == 1) {
                    map.erase(ch);
                } else {
                    map[ch]--;
                }

                if (map.size() > k) {
                    continue;
                } else {
                    int len = i - j;
                    if (len > ans) {
                        ans = len;
                    }
                    break;
                }
            }

            if (!f1 && !f2) {
                break;
            }
        }
        return ans;
    }
};
```

Understanding the Problem

- The function `sol(str, k)` finds the **longest substring** with at most k unique characters.
- Uses **two-pointer sliding window** technique (i and j) with an **unordered_map** to track character frequencies.
- Expands the window until the number of unique characters exceeds k , then shrinks the window.

Example Input

```
string str = "ddacbbaccdedacebb";
int k = 3;
```

Expected Output: 7

Step-by-Step Dry Run

Step	i	j	Window (<code>str[j+1]</code> to <code>str[i]</code>)	Unique Chars	Max Length (ans)
1	0	-1	d	1	1
2	1	-1	dd	1	2
3	2	-1	dda	2	3
4	3	-1	ddac	3	4
5	4	-1	ddacb	4 (exceeds k)	4
6	4	0	dacb	3	4
7	5	0	dacbb	3	5
8	6	0	dacbbba	3	6
9	7	0	dacbbbac	3	7 ✓
10	8	0	dacbbacc	3	7
11	9	1	acbbaccd	4 (exceeds k)	7
12	9	2	cbbaccd	3	7
13	10	2	cbbaccde	4 (exceeds k)	7
14	10	3	bbaccde	3	7
15	11	3	bbacccded	4 (exceeds k)	7

```
int main() {
    std::string str = "ddacbbaccdedacebb";
    int k = 3;
    std::cout <<
LongestSubstringWithAtMostKUniqueCharacters::sol(str
, k) << std::endl;
    return 0;
}
```

				k)	
...

Final Output

↙ Longest substring with at most k = 3
unique characters: 7

Output:-

7

LongestSubStringWithNonRepeatingCharacters in C++

```
#include <iostream>
#include <string>
#include <unordered_map>

class LongestSubStringWithNonRepeatingCharacters {
public:
    static int solution(const std::string& str) {
        int ans = 0;
        int i = -1;
        int j = -1;

        std::unordered_map<char, int> map;
        while (true) {
            bool f1 = false;
            bool f2 = false;

            while (i < static_cast<int>(str.length()) - 1) {
                f1 = true;
                i++;
                char ch = str[i];
                map[ch]++;
                if (map[ch] == 2) {
                    break;
                } else {
                    int len = i - j;
                    if (len > ans) {
                        ans = len;
                    }
                }
            }

            while (j < i) {
                f2 = true;
                j++;
                char ch = str[j];
                map[ch]--;
                if (map[ch] == 1) {
                    break;
                }
            }

            if (!f1 && !f2) {
                break;
            }
        }

        return ans;
    }

    int main() {
        std::string str = "aabcbcdbea";
        std::cout <<
LongestSubStringWithNonRepeatingCharacters::solution(str)
        << std::endl;
        return 0;
    }
}
```

Understanding the Problem

- The function **solution(str)** finds the **length of the longest substring with all distinct (non-repeating) characters**.
- Uses **two-pointer sliding window** (**i** and **j**) with an **unordered_map** to track character frequencies.
- Expands the window until a duplicate character is found, then contracts the window to remove duplicates.

Example Input

```
string str = "aabcbcdbea";
```

Expected Output: 4 (longest substring = "bcdb")

Step-by-Step Dry Run

Step	i	j	Window (str[j+1] to str[i])	Map	Max Length (ans)
1	0	-1	a	{a:1}	1
2	1	-1	aa	{a:2} (duplicate)	1
3	1	0	a	{a:1}	1
4	2	0	ab	{a:1, b:1}	2
5	3	0	abc	{a:1, b:1, c:1}	3
6	4	0	abcb	{a:1, b:2, c:1}	3
7	4	1	bcb	{b:2, c:1}	3
8	4	2	cb	{b:1, c:1}	3
9	5	2	cbc	{b:1, c:2}	3
10	5	3	bc	{b:1, c:1}	3
11	6	3	bcd	{b:1, c:1, d:1}	3
12	7	3	bcdb	{b:2, c:1, d:1}	4 ✓
13	7	4	cdb	{b:1, }	4

				c:1, d:1}	
14	8	4	cdbc	{b:1, c:2, d:1}	4
15	8	5	dbc	{b:1, c:1, d:1}	4
16	9	5	dbca	{b:1, c:1, d:1, a:1}	4 ✓
17	10	6	bca	{b:1, c:1, a:1}	4

Final Output

✓ Longest substring without repeating characters: 4 ("bcdb" or "dbca")

Output:-4

Pair with equal sum in C++

```
#include <iostream>
#include <unordered_set>
#include <vector>

using namespace std;

bool sol(vector<int>& arr) {
    unordered_set<int> set;

    for (int i = 0; i < arr.size(); i++) {
        for (int j = i + 1; j < arr.size(); j++) {
            int sum = arr[i] + arr[j];
            if (set.count(sum)) {
                return true;
            } else {
                set.insert(sum);
            }
        }
    }
    return false;
}

int main() {
    vector<int> arr = {2, 9, 3, 5, 8, 6, 4};
    bool ans = sol(arr);
    cout << boolalpha << ans << endl;
    return 0;
}
```

Input

arr = {2, 9, 3, 5, 8, 6, 4}

Dry Run Table

i	j	arr[i]	arr[j]	sum	Seen Sums Before	Is sum already in set?	Action
0	1	2	9	11	{}	No	Insert 11
0	2	2	3	5	{11}	No	Insert 5
0	3	2	5	7	{11, 5}	No	Insert 7
0	4	2	8	10	{11, 5, 7}	No	Insert 10
0	5	2	6	8	{11, 5, 7, 10}	No	Insert 8
0	6	2	4	6	{5, 7, 8, 10, 11}	No	Insert 6
1	2	9	3	12	{5, 6, 7, 8, 10, 11}	No	Insert 12
1	3	9	5	14	...	No	Insert 14
1	4	9	8	17	...	No	Insert 17
1	5	9	6	15	...	No	Insert 15
1	6	9	4	13	...	No	Insert 13
2	3	3	5	8	Already seen	✓ Yes → Return true	

✓ Output

true

Output:-
true

Subarray sum equals k in C++

```
#include <iostream>
#include <vector>
#include <unordered_map>
using namespace std;
class SubarraySumEqualsK {
public:
    static int sol(const std::vector<int>& arr,
    int target) {
        int ans = 0;
        std::unordered_map<int, int> map;
        map[0] = 1;
        int sum = 0;

        for (int i = 0; i < arr.size(); i++) {
            sum += arr[i];
            int rsum = sum - target;
            if (map.find(rsum) != map.end()) {
                ans += map[rsum];
            }
            map[sum]++;
        }
        return ans;
    }

    int main() {
        vector<int> arr = {3, 9, -2, 4, 1, -7, 2, 6,
        -5, 8, -3, -7, 6, 2, 1};
        int k = 5;
        cout << SubarraySumEqualsK::sol(arr,
        k) << std::endl;
        return 0;
    }
}
```

Example Input

```
vector<int> arr = {3, 9, -2, 4, 1, -7, 2, 6,
-5, 8, -3, -7, 6, 2, 1};
int k = 5;
```

Expected Output: 5

Step-by-Step Dry Run

Step	i	arr[i]	sum (Prefix Sum)	rsum = sum - k	map[rsum] (if exists)	ans (count of subarrays)	map[sum] (updated)
1	0	3	3	-2	0	0	{0:1, 3:1}
2	1	9	12	7	0	0	{0:1, 3:1, 12:1}
3	2	-2	10	5	0	0	{0:1, 3:1, 12:1, 10:1}
4	3	4	14	9	0	0	{0:1, 3:1, 12:1, 10:1, 14:1}
5	4	1	15	10	✓1	1	{0:1, 3:1, 12:1, 10:1, 14:1, 15:1}
6	5	-7	8	3	✓1	2	{0:1, 3:1, 12:1, 10:1, 14:1, 15:1, 8:1}
7	6	2	10	5	0	2	{0:1, 3:1, 12:1, 10:2, 14:1, 15:1, 8:1}
8	7	6	16	11	0	2	{0:1,

								$\{3:1, 12:1, 10:2, 14:1, 15:1, 8:1, 16:1\}$
9	8	-5	11	6	0		2	$\{0:1, 3:1, 12:1, 10:2, 14:1, 15:1, 8:1, 16:1, 11:1\}$
10	9	8	19	14	$\checkmark 1$		3	$\{0:1, 3:1, 12:1, 10:2, 14:1, 15:1, 8:1, 16:1, 11:1, 19:1\}$
11	10	-3	16	11	$\checkmark 1$		4	$\{0:1, 3:1, 12:1, 10:2, 14:1, 15:1, 8:1, 16:2, 11:1, 19:1\}$
12	11	-7	9	4	0		4	$\{0:1, 3:1, 12:1, 10:2, 14:1, 15:1, 8:1, 16:2, 11:1, 19:1, 9:1\}$
13	12	6	15	10	$\checkmark 2$		6	$\{0:1, 3:1, 12:1, 10:2, 14:1, 15:2, 8:1, 16:2, 11:1, 19:1\}$

								9:1}
14	13	2	17	12	✓1		7	{0:1, 3:1, 12:1, 10:2, 14:1, 15:2, 8:1, 16:2, 11:1, 19:1, 9:1, 17:1}
15	14	1	18	13	0		7	{0:1, 3:1, 12:1, 10:2, 14:1, 15:2, 8:1, 16:2, 11:1, 19:1, 9:1, 17:1, 18:1}

Final Output

✓ Output: 7

Output:-

Two Sum in C++

```
#include <iostream>
#include <unordered_map>
#include <vector>

using namespace std;

vector<int> twoSum(vector<int>& nums, int target)
{
    unordered_map<int, int> map; // Hash map to
    store number and its index
    vector<int> result;

    for (int i = 0; i < nums.size(); i++) {
        int complement = target - nums[i];

        if (map.find(complement) != map.end()) {
            result.push_back(map[complement]);
            result.push_back(i);
            return result;
        }

        map[nums[i]] = i;
    }

    throw invalid_argument("No two sum solution");
}

int main() {
    vector<int> nums1 = {2, 7, 11, 15};
    int target1 = 9;

    vector<int> nums2 = {3, 2, 4};
    int target2 = 6;

    vector<int> result1 = twoSum(nums1, target1);
    vector<int> result2 = twoSum(nums2, target2);

    cout << "Output for nums1: [" << result1[0] << ", "
    << result1[1] << "] " << endl;
    cout << "Output for nums2: [" << result2[0] << ", "
    << result2[1] << "] " << endl;

    return 0;
}
```

Test Case 1

vector<int> nums1 = {2, 7, 11, 15};
int target1 = 9;

- We need to find two indices i, j such that $\text{nums1}[i] + \text{nums1}[j] = 9$.

Step	i	nums1[i]	Complement (target - nums1[i])	map (stored indices)	Match Found?
1	0	2	7	{2:0}	✗ No
2	1	7	2	{2:0, 7:1}	✓ Yes (2 found at index 0)

✓ Output: [0, 1] (because $\text{nums1}[0] + \text{nums1}[1] = 2 + 7 = 9$)

Test Case 2

vector<int> nums2 = {3, 2, 4};
int target2 = 6;

Step	i	nums2[i]	Complement (target - nums2[i])	map (stored indices)	Match Found?
1	0	3	3	{3:0}	✗ No
2	1	2	4	{3:0, 2:1}	✗ No
3	2	4	2	{3:0, 2:1, 4:2}	✓ Yes (2 found at index 1)

✓ Output: [1, 2] (because $\text{nums2}[1] + \text{nums2}[2] = 2 + 4 = 6$)

Output:-

Output for nums1: [0, 1]
Output for nums2: [1, 2]

Valid Anagram in C++

```
#include <iostream>
#include <string>
#include <unordered_map>

class ValidAnagrams {
public:
    static bool sol(const std::string& s1, const
std::string& s2) {
        std::unordered_map<char, int> map;
        for (char ch : s1) {
            map[ch]++;
        }

        for (char ch : s2) {
            if (map.find(ch) == map.end()) {
                return false;
            } else if (map[ch] == 1) {
                map.erase(ch);
            } else {
                map[ch]--;
            }
        }
        return map.empty();
    }
};

int main() {
    std::string s1 = "abbcaad";
    std::string s2 = "babacda";
    std::cout << (ValidAnagrams::sol(s1, s2) ? "true" :
"false") << std::endl;
    return 0;
}
```

Dry Run Table for `validAnagrams::sol(s1, s2)`

Input:

`s1 = "abbcaad";
s2 = "babacda";`

Step 1: Build Character Frequency Map (`s1`)

Iteration	Character (ch)	map[ch] (Updated)	map State
0	'a'	1	{ 'a': 1 }
1	'b'	1	{ 'a': 1, 'b': 1 }
2	'b'	2	{ 'a': 1, 'b': 2 }
3	'c'	1	{ 'a': 1, 'b': 2, 'c': 1 }
4	'a'	2	{ 'a': 2, 'b': 2, 'c': 1 }
5	'a'	3	{ 'a': 3, 'b': 2, 'c': 1 }
6	'd'	1	{ 'a': 3, 'b': 2, 'c': 1, 'd': 1 }

Final map after processing `s1`:

{ 'a': 3, 'b': 2, 'c': 1, 'd': 1 }

Step 2: Validate Using `s2`

Iteration	Character (ch)	Action	map[ch] (Updated)	map State
0	'b'	Decrement	1	{ 'a': 3, 'b': 1, 'c': 1, 'd': 1 }
1	'a'	Decrement	2	{ 'a': 2, 'b': 1, 'c': 1, 'd': 1 }

Iteration	Character (ch)	Action	map [ch] (Updated)	map State
2	'b'	Remove from map	{ 'a': 2, 'c': 1, 'd': 1 }	
3	'a'	Decrement 1	{ 'a': 1, 'c': 1, 'd': 1 }	
4	'c'	Remove from map	{ 'a': 1, 'd': 1 }	
5	'd'	Remove from map	{ 'a': 1 }	
6	'a'	Remove from map	{ }	
Final map state: Empty {}, meaning both strings are anagrams.				
↙ Output: "true"				
Output:- true				

Heap in C++

```
#include <iostream>
#include <vector>

using namespace std;

class MinHeap {
    vector<int> arr;
    int size;
    int capacity;

public:
    MinHeap(int c) {
        size = 0;
        capacity = c;
        arr.resize(c);
    }

    int left(int i) {
        return 2 * i + 1;
    }

    int right(int i) {
        return 2 * i + 2;
    }

    int parent(int i) {
        return (i - 1) / 2;
    }

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

    void insert(int x) {
        if (size == capacity) {
            return;
        }
        size++;
        arr[size - 1] = x;
        int i = size - 1;
        while (i != 0 && arr[parent(i)] > arr[i]) {
            swap(arr[i], arr[parent(i)]);
            i = parent(i);
        }
    }

    int main() {
        MinHeap h(9);
        h.insert(10);
        h.insert(20);
        h.insert(15);
        h.insert(40);
        h.insert(50);
        h.insert(100);
        h.insert(25);
        h.insert(45);
    }
}
```

Initial Setup

- We create a **MinHeap** of capacity 9.
- Insert sequence: 10, 20, 15, 40, 50, 100, 25, 45

What is a MinHeap?

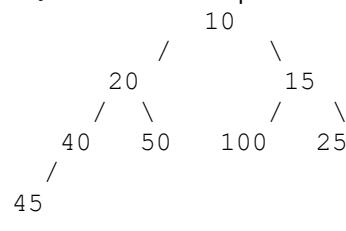
A MinHeap is a **complete binary tree** where:

- Every parent node is **less than or equal to** its children.
- Insertions maintain this structure by "bubbling up" the inserted value if it violates the heap property.

▼ Step-by-Step Dry Run Table

Step	Inserted Value	Heap Before Insert	Heap After Insert + Bubble Up
1	10	[]	[10]
2	20	[10]	[10, 20]
3	15	[10, 20]	[10, 20, 15]
4	40	[10, 20, 15]	[10, 20, 15, 40]
5	50	[10, 20, 15, 40]	[10, 20, 15, 40, 50]
6	100	[10, 20, 15, 40, 50]	[10, 20, 15, 40, 50, 100]
7	25	[10, 20, 15, 40, 50, 100]	[10, 20, 15, 40, 50, 100, 25]
8	45	[10, 20, 15, 40, 50, 100, 25]	[10, 20, 15, 40, 50, 100, 25, 45]

Final MinHeap Tree Representation:



- The heap property is maintained at each

```
h.show();  
return 0;  
}
```

- step.
- No bubbling up required beyond one level in most cases.

✓ Output of h.show();
10 20 15 40 50 100 25 45

```
10 20 15 40 50 100 25 45
```

K largest elements in C++

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

void klargest(vector<int>& arr, int k) {
    priority_queue<int, vector<int>, greater<int>> pq;

    // Insert the first k elements into the min heap
    for (int i = 0; i < k; i++) {
        pq.push(arr[i]);
    }

    // For each element from k to end of array, check if
    // it's larger than the smallest in the heap
    for (int i = k; i < arr.size(); i++) {
        if (pq.top() < arr[i]) {
            pq.pop();
            pq.push(arr[i]);
        }
    }

    // Print the k largest elements
    cout << "K largest elements: ";
    while (!pq.empty()) {
        cout << pq.top() << " ";
        pq.pop();
    }
    cout << endl;
}

int main() {
    // Hardcoded input array
    vector<int> arr = {5, 15, 10, 20, 8, 25, 18};
    int k = 3;

    // Call the klargest function to find and print the k
    // largest elements
    klargest(arr, k);

    return 0;
}
```

K largest elements: 18 20 25

Step-by-Step Dry Run

Step	i	Element	Min Heap Before	Action	Min Heap After
Init	-	-	[]	Start inserting first k=3	
1	0	5	[5]	Push 5	[5]
2	1	15	[5, 15]	Push 15	[5, 15]
3	2	10	[5, 15]	Push 10	[5, 15, 10]
4	3	20	[5, 15, 10]	20 > 5 → pop 5, push 20	[10, 15, 20]
5	4	8	[10, 15, 20]	8 < 10 → do nothing	[10, 15, 20]
6	5	25	[10, 15, 20]	25 > 10 → pop 10, push 25	[15, 20, 25]
7	6	18	[15, 20, 25]	18 > 15 → pop 15, push 18	[18, 25, 20]

❖ Final Heap Contents: [18, 25, 20]

This heap now contains the **top 3 largest elements: 18, 25, 20**

▣ Output:

K largest elements: 18 20 25

K sorted array in C++

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

void sortKSortedArray(vector<int>& arr, int k) {
    priority_queue<int, vector<int>, greater<int>> pq; // Min heap

    // Push the first k+1 elements into the priority queue
    for (int i = 0; i <= k; ++i) {
        pq.push(arr[i]);
    }

    int index = 0;

    // Process the remaining elements
    for (int i = k + 1; i < arr.size(); ++i) {
        arr[index++] = pq.top(); // Get the smallest element from the heap
        pq.pop(); // Remove the smallest element from the heap
        pq.push(arr[i]); // Push the current element into the heap
    }

    // Extract all remaining elements from the heap
    while (!pq.empty()) {
        arr[index++] = pq.top();
        pq.pop();
    }

    // Print sorted array
    for (int i = 0; i < arr.size(); ++i) {
        cout << arr[i] << " ";
    }
    cout << endl;
}

int main() {
    // Hardcoded input array
    vector<int> arr = {7, 8, 9, 19, 18};
    int k = 3;

    // Sort the k-sorted array
    sortKSortedArray(arr, k);

    return 0;
}
```

Input:

arr = {7, 8, 9, 19, 18}
k = 3

We will walk through it step-by-step in a table format showing the **min heap**, **index**, and how the array is being modified.

█ Initial Step – Insert first $k+1 = 4$ elements into min-heap:

Step	Action	Min Heap	arr[]	index
0	Insert first 4 elements (0-3)	[7, 8, 9, 19]	[7, 8, 9, 19, 18]	—

↻ Main Loop (from $i = k+1$ to end):

Step	i	Action	Min Heap Before	Element Pushed	Popped \rightarrow arr[index]	Min Heap After	arr[]	index
1	4	Push 18, Pop & insert 7	[7, 8, 9, 19]	18	7	[8, 18, 9, 19]	[7, 8, 9, 19, 18]	0
2	—	Pop & insert 8	[8, 18, 9, 19]	—	8	[9, 18, 19]	[7, 8, 9, 19, 18]	1
3	—	Pop & insert 9	[9, 18, 19]	—	9	[18, 19]	[7, 8, 9, 19, 18]	2
4	—	Pop & insert 18	[18, 19]	—	18	[19]	[7, 8, 9, 18, 18]	3
5	—	Pop & insert 19	[19]	—	19	[]	[7, 8, 9, 18, 19]	4

	<p>✓ Final Output:</p> <p>7 8 9 18 19</p>
7 8 9 18 19	

BFSPath in C++																																																																	
<pre>#include <iostream> #include <vector> #include <deque> using namespace std; // Edge structure representing an edge // between two vertices struct Edge { int src; int nbr; Edge(int src, int nbr) { this->src = src; this->nbr = nbr; } }; // Pair structure to store vertex and path // so far struct Pair { int v; string psf; Pair(int v, string psf) : v(v), psf(psf) {} }; // Function to add an edge between two // vertices void addEdge(vector<Edge>* graph, int v1, int v2) { graph[v1].push_back(Edge(v1, v2)); graph[v2].push_back(Edge(v2, v1)); } int main() { int vtces = 7; // Number of vertices vector<Edge>* graph = new vector<Edge>[vtces]; // Adjacency list of edges // Adding edges to the graph addEdge(graph, 0, 1); addEdge(graph, 1, 2); addEdge(graph, 2, 3); addEdge(graph, 0, 3); addEdge(graph, 3, 4); addEdge(graph, 4, 5); addEdge(graph, 5, 6); addEdge(graph, 4, 6); int src = 0; // Source vertex for BFS deque<Pair> q; // Queue for BFS vector<bool> visited(vtces, false); // Array to mark visited vertices q.push_back(Pair(src, to_string(src))); // Pushing source vertex with path so far </pre>	<p>Graph Structure:</p> <p>Edges (undirected):</p> <pre>0 -- 1 1 -- 2 2 -- 3 0 -- 3 3 -- 4 4 -- 5 5 -- 6 4 -- 6</pre> <p>This gives us the following adjacency list:</p> <table border="1"> <thead> <tr> <th>Vertex</th><th>Neighbors</th></tr> </thead> <tbody> <tr> <td>0</td><td>1, 3</td></tr> <tr> <td>1</td><td>0, 2</td></tr> <tr> <td>2</td><td>1, 3</td></tr> <tr> <td>3</td><td>2, 0, 4</td></tr> <tr> <td>4</td><td>3, 5, 6</td></tr> <tr> <td>5</td><td>4, 6</td></tr> <tr> <td>6</td><td>5, 4</td></tr> </tbody> </table>	Vertex	Neighbors	0	1, 3	1	0, 2	2	1, 3	3	2, 0, 4	4	3, 5, 6	5	4, 6	6	5, 4																																																
Vertex	Neighbors																																																																
0	1, 3																																																																
1	0, 2																																																																
2	1, 3																																																																
3	2, 0, 4																																																																
4	3, 5, 6																																																																
5	4, 6																																																																
6	5, 4																																																																
	<p>🧠 BFS Behavior:</p> <ul style="list-style-type: none"> • Queue type: deque • Visited is marked only when popped (standard BFS behavior) • Pair stores (vertex, path-so-far) • Queue allows tracking of the shortest path from source 																																																																
	<p>📋 Dry Run Table:</p> <table border="1"> <thead> <tr> <th>Step</th><th>Queue (Front → Back)</th><th>Visited</th><th>Output</th></tr> </thead> <tbody> <tr> <td>1</td><td>(0, "0")</td><td>{}</td><td></td></tr> <tr> <td>2</td><td>—</td><td>{0}</td><td>0 0</td></tr> <tr> <td></td><td>Enqueue: (1, "01"), (3, "03")</td><td></td><td></td></tr> <tr> <td>3</td><td>(1, "01"), (3, "03")</td><td>{0}</td><td></td></tr> <tr> <td>4</td><td>—</td><td>{0, 1}</td><td>1 01</td></tr> <tr> <td></td><td>Enqueue: (0, "010"), (2, "012")</td><td></td><td></td></tr> <tr> <td>5</td><td>(3, "03"), (0, "010"), (2, "012")</td><td>{0, 1}</td><td></td></tr> <tr> <td>6</td><td>—</td><td>{0, 1, 3}</td><td>3 03</td></tr> <tr> <td></td><td>Enqueue: (2, "032"), (0, "030"), (4, "034")</td><td></td><td></td></tr> <tr> <td>7</td><td>(0, "010"), (2, "012"), (2, "032"), (0, "030"), (4, "034")</td><td>{0, 1, 3}</td><td></td></tr> <tr> <td>8</td><td>— 0 already visited → skip</td><td>{0, 1, 3}</td><td></td></tr> <tr> <td>9</td><td>—</td><td>{0, 1, 2, 3}</td><td>2 012</td></tr> <tr> <td></td><td>Enqueue: (1, "0121"), (3, "0123")</td><td></td><td></td></tr> <tr> <td>10</td><td>— 2 already visited → skip</td><td></td><td></td></tr> <tr> <td>11</td><td>— 0 already visited → skip</td><td></td><td></td></tr> </tbody> </table>	Step	Queue (Front → Back)	Visited	Output	1	(0, "0")	{}		2	—	{0}	0 0		Enqueue: (1, "01"), (3, "03")			3	(1, "01"), (3, "03")	{0}		4	—	{0, 1}	1 01		Enqueue: (0, "010"), (2, "012")			5	(3, "03"), (0, "010"), (2, "012")	{0, 1}		6	—	{0, 1, 3}	3 03		Enqueue: (2, "032"), (0, "030"), (4, "034")			7	(0, "010"), (2, "012"), (2, "032"), (0, "030"), (4, "034")	{0, 1, 3}		8	— 0 already visited → skip	{0, 1, 3}		9	—	{0, 1, 2, 3}	2 012		Enqueue: (1, "0121"), (3, "0123")			10	— 2 already visited → skip			11	— 0 already visited → skip		
Step	Queue (Front → Back)	Visited	Output																																																														
1	(0, "0")	{}																																																															
2	—	{0}	0 0																																																														
	Enqueue: (1, "01"), (3, "03")																																																																
3	(1, "01"), (3, "03")	{0}																																																															
4	—	{0, 1}	1 01																																																														
	Enqueue: (0, "010"), (2, "012")																																																																
5	(3, "03"), (0, "010"), (2, "012")	{0, 1}																																																															
6	—	{0, 1, 3}	3 03																																																														
	Enqueue: (2, "032"), (0, "030"), (4, "034")																																																																
7	(0, "010"), (2, "012"), (2, "032"), (0, "030"), (4, "034")	{0, 1, 3}																																																															
8	— 0 already visited → skip	{0, 1, 3}																																																															
9	—	{0, 1, 2, 3}	2 012																																																														
	Enqueue: (1, "0121"), (3, "0123")																																																																
10	— 2 already visited → skip																																																																
11	— 0 already visited → skip																																																																

```

while (!q.empty()) {
    Pair rem = q.front();
    q.pop_front();

    if (visited[rem.v]) {
        continue;
    }
    visited[rem.v] = true;

    cout << rem.v << " " << rem.psf <<
endl; // Printing vertex and path so far

    // Iterating through all adjacent
    vertices
    for (Edge e : graph[rem.v]) {
        q.push_back(Pair(e.nbr, rem.psf +
to_string(e.nbr))); // Adding adjacent
        vertices to queue
    }

    delete[] graph; // Freeing dynamically
allocated memory for graph
    return 0;
}

```

Step	Queue (Front → Back)	Visited	Output
12	—	{0,1,2,3,4}	4 034
	Enqueue: (3, "0343"), (5, "0345"), (6, "0346")		
13	— 1 already visited → skip		
14	— 3 already visited → skip		
15	—	{..., 5}	5 0345
	Enqueue: (4, "03454"), (6, "03456")		
16	—	{..., 6}	6 0346
	Enqueue: (5, "03465"), (4, "03464")		
...	All remaining vertices already visited → skip		

❖ Final Output:

(printed in order of **first encounter** in BFS)

0 0
1 01
3 03
2 012
4 034
5 0345
6 0346

Output:-

0 0
1 01
3 03
2 012
4 034
5 0345
6 0346

Hamiltonian Path and Cycle in C++

```
#include <iostream>
#include <vector>
#include <unordered_set>

using namespace std;

// Structure to represent an edge in the graph
struct Edge {
    int src;
    int nbr;
    int wt;
    Edge(int src, int nbr, int wt) {
        this->src = src;
        this->nbr = nbr;
        this->wt = wt;
    }
};

// Function to add an edge to the graph
void addEdge(vector<Edge>* graph, int src, int nbr, int wt) {
    graph[src].push_back(Edge(src, nbr, wt));
    graph[nbr].push_back(Edge(nbr, src, wt)); // Assuming undirected graph
}

// Function to perform Hamiltonian path and cycle calculation
void h(vector<Edge>* graph, int src, unordered_set<int>& visited, string psf, int originalSrc) {
    if (visited.size() == graph->size() - 1) {
        cout << psf;
    }

    bool containsCycle = false;
    for (Edge& e : graph[src]) {
        if (e.nbr == originalSrc) {
            containsCycle = true;
            break;
        }
    }

    if (containsCycle) {
        cout << "*" << endl;
    } else {
        cout << "." << endl;
    }

    return;
}

visited.insert(src);
for (Edge& e : graph[src]) {
    if (visited.find(e.nbr) == visited.end()) {
        h(graph, e.nbr, visited, psf + to_string(e.nbr), originalSrc);
    }
}
```

Goal:

Explore all **Hamiltonian paths/cycles** starting from node 0.

❖ Graph Summary:

Node	Neighbors
0	1, 3
1	0, 2
2	1, 3, 4
3	0, 2, 4
4	3, 5, 2
5	4

❖ Table Format:

Step	Current Node	Visited Set	Path So Far (psf)	Action
1	0	{0}	"0"	Start
2	1	{0,1}	"01"	0 → 1
3	2	{0,1,2}	"012"	1 → 2
4	3	{0,1,2,3}	"0123"	2 → 3
5	4	{0,1,2,3,4}	"01234"	3 → 4
6	5	{0,1,2,3,4,5}	"012345"	4 → 5
7	—	—	"012345."	6 vertices visited, no edge 5→0

→ So we print: 012345.

Let's try another valid path:

Step	Current Node	Visited Set	Path So Far (psf)	Action
1	0	{0}	"0"	Start
2	3	{0,3}	"03"	0 → 3
3	2	{0,3,2}	"032"	3 → 2
4	1	{0,3,2,1}	"0321"	2 → 1
5	4	{0,3,2,1,4}	"03214"	2 → 4
6	5	{0,3,2,1,4,5}	"032145"	4 → 5
7	—	—	"032145."	No edge 5→0, just a path

→ We print: 032145.

Let's do a cycle example:

Step	Current Node	Visited Set	Path So Far (psf)	Action
1	0	{0}	"0"	Start
2	3	{0,3}	"03"	0 → 3

```

    }
    visited.erase(src);
}

int main() {
    int vtes = 6; // Number of vertices
    //int edges = 7; // Number of edges

    // Create the graph using adjacency
    list representation
    vector<Edge>* graph = new
    vector<Edge>[vtes];

    // Add edges to the graph
    addEdge(graph, 0, 1, 10);
    addEdge(graph, 0, 3, 40);
    addEdge(graph, 1, 2, 10);
    addEdge(graph, 2, 3, 10);
    addEdge(graph, 3, 4, 2);
    addEdge(graph, 4, 5, 2);
    addEdge(graph, 2, 4, 3);

    int src = 0; // Source vertex

    // Perform Hamiltonian path and cycle
    calculation
    unordered_set<int> visited;
    h(graph, src, visited, to_string(src),
    src);

    delete[] graph; // Deallocate memory

    return 0;
}

```

Output:-
01*
03*

Step	Current Node	Visited Set	Path So Far (psf)	Action
3	4	{0,3,4}	"034"	$3 \rightarrow 4$
4	2	{0,3,4,2}	"0342"	$4 \rightarrow 2$
5	1	{0,3,4,2,1}	"03421"	$2 \rightarrow 1$
6	5	{0,3,4,2,1,5}	"034215"	$4 \rightarrow 5$
7	—	—	"034215*"	Edge exists $5 \rightarrow 0$ → CYCLE ✓

→ We print: 034215*

Summary of Dry Run:

Path	Hamiltonian	Cycle?
012345	✓	✗
032145	✓	✗
034215	✓	✓

Print All Paths in C++																																							
#include <iostream> #include <vector> using namespace std; // Define the Edge structure struct Edge { int src; int nbr; int wt; Edge(int s, int n, int w) { src = s; nbr = n; wt = w; } }; // Function prototypes void addEdge(vector<Edge>* graph, int src, int nbr, int wt); void printAllPaths(vector<Edge>* graph, int src, int dest, vector<bool>& visited, string psf); int main() { int vtes = 6; // Number of vertices //int edges = 7; // Number of edges // Create the graph using vector of // vectors vector<Edge>* graph = new vector<Edge>[vtes]; // Add edges statically addEdge(graph, 0, 1, 10); addEdge(graph, 0, 3, 40); addEdge(graph, 1, 2, 10); addEdge(graph, 2, 3, 10); addEdge(graph, 3, 4, 2); addEdge(graph, 4, 5, 2); addEdge(graph, 2, 4, 3); int src = 0; // Source vertex int dest = 5; // Destination vertex // Array to track visited vertices vector<bool> visited(vtes, false); // Call the function to print all paths // from src to dest printAllPaths(graph, src, dest, visited, to_string(src)); return 0; }	Graph Structure: Edges: 0 -- 1 (10) 0 -- 3 (40) 1 -- 2 (10) 2 -- 3 (10) 3 -- 4 (2) 4 -- 5 (2) 2 -- 4 (3) This gives us the adjacency list: <table border="1"><thead><tr><th>Vertex</th><th>Neighbors</th></tr></thead><tbody><tr><td>0</td><td>1, 3</td></tr><tr><td>1</td><td>0, 2</td></tr><tr><td>2</td><td>1, 3, 4</td></tr><tr><td>3</td><td>0, 2, 4</td></tr><tr><td>4</td><td>3, 5, 2</td></tr><tr><td>5</td><td>4</td></tr></tbody></table>			Vertex	Neighbors	0	1, 3	1	0, 2	2	1, 3, 4	3	0, 2, 4	4	3, 5, 2	5	4																						
Vertex	Neighbors																																						
0	1, 3																																						
1	0, 2																																						
2	1, 3, 4																																						
3	0, 2, 4																																						
4	3, 5, 2																																						
5	4																																						
Goal:																																							
Find all paths from src = 0 to dest = 5.																																							
Dry Run Table:																																							
<table border="1"> <thead> <tr> <th>Recursive Call</th><th>Current src</th><th>Path So Far (psf)</th><th>Action</th></tr> </thead> <tbody> <tr> <td>1</td><td>0</td><td>"0"</td><td>Explore neighbors 1, 3</td></tr> <tr> <td>2</td><td>1</td><td>"01"</td><td>Explore neighbors 2</td></tr> <tr> <td>3</td><td>2</td><td>"012"</td><td>Explore 3, 4</td></tr> <tr> <td>4</td><td>3</td><td>"0123"</td><td>Explore 4</td></tr> <tr> <td>5</td><td>4</td><td>"01234"</td><td>Explore 5</td></tr> <tr> <td>6</td><td>5</td><td>"012345"</td><td>↙ Print this path</td></tr> <tr> <td colspan="2">Backtrack to 4</td><td></td><td></td></tr> <tr> <td colspan="2">Backtrack to 3</td><td></td><td></td></tr> </tbody> </table>				Recursive Call	Current src	Path So Far (psf)	Action	1	0	"0"	Explore neighbors 1, 3	2	1	"01"	Explore neighbors 2	3	2	"012"	Explore 3, 4	4	3	"0123"	Explore 4	5	4	"01234"	Explore 5	6	5	"012345"	↙ Print this path	Backtrack to 4				Backtrack to 3			
Recursive Call	Current src	Path So Far (psf)	Action																																				
1	0	"0"	Explore neighbors 1, 3																																				
2	1	"01"	Explore neighbors 2																																				
3	2	"012"	Explore 3, 4																																				
4	3	"0123"	Explore 4																																				
5	4	"01234"	Explore 5																																				
6	5	"012345"	↙ Print this path																																				
Backtrack to 4																																							
Backtrack to 3																																							

```

    graph[nbr].emplace_back(nbr, src, wt);
}

// Function to print all paths from src to dest
void printAllPaths(vector<Edge>* graph,
int src, int dest, vector<bool>& visited,
string psf) {
    if (src == dest) {
        cout << psf << endl;
        return;
    }

    visited[src] = true;
    for (Edge edge : graph[src]) {
        if (!visited[edge.nbr]) {
            printAllPaths(graph, edge.nbr,
dest, visited, psf + to_string(edge.nbr));
        }
    }
    visited[src] = false;
}

```

Recursive Call	Current src	Path So Far (psf)	Action
4 (alt)	4	"0124"	Explore 5
5	5	"01245"	↙ Print this path
Backtrack to 2			
Backtrack to 1			
Backtrack to 0			
2	3	"03"	Explore 2, 4
3	2	"032"	Explore 4
4	4	"0324"	Explore 5
5	5	"03245"	↙ Print this path
Backtrack to 3			
3 (alt)	4	"034"	Explore 5
4	5	"0345"	↙ Print this path

↙ Final Output:

012345
01245
03245
0345

Output:-

012345
01245
03245
0345

Bus Routes in C++								
Input:								
<pre>#include <iostream> #include <vector> #include <unordered_map> #include <queue> #include <unordered_set> using namespace std; int numBusesToDestination(vector<vector<int>>& routes, int S, int T) { int n = routes.size(); unordered_map<int, vector<int>> map; // Building a map of bus stops to their // respective bus routes for (int i = 0; i < n; ++i) { for (int j = 0; j < routes[i].size(); ++j) { int busStopNo = routes[i][j]; map[busStopNo].push_back(i); } } queue<int> q; unordered_set<int> busStopVisited; unordered_set<int> busVisited; int level = 0; q.push(S); busStopVisited.insert(S); // Performing BFS to find the // minimum number of buses while (!q.empty()) { int size = q.size(); while (size-- > 0) { int currentStop = q.front(); q.pop(); if (currentStop == T) { return level; } if (map.find(currentStop) != map.end()) { vector<int>& buses = map[currentStop]; for (int bus : buses) { if (busVisited.count(bus) > 0) { continue; } vector<int>& busRoute = routes[bus]; for (int nextStop : busRoute) { if (busStopVisited.count(nextStop) > 0) { continue; } } } } } } }</pre>	<pre>routes = { {1, 2, 7}, {3, 6, 7} }; src = 1; dest = 6;</pre>							
<p> High-Level Idea:</p> <p>The code builds a graph where each bus stop connects to bus routes, then performs BFS starting from the source stop to find the minimum number of buses needed to reach the destination.</p>								
Iteration	Level	Queue Content	Current Stop	Bus Routes from Stop	New Stops Added to Queue	Bus Visited	Comments	
Init	0	[1]	-	-	-	-	Start from stop 1	
1	0	[1]	1	[0]	[2, 7]	{0}	Stop 1 is in route 0; enqueue 2, 7	
2	1	[2, 7]	2	[0]	-	{0}	Bus 0 already visited	
3	1	[7]	7	[0, 1]	[3, 6]	{0, 1}	Route 1 has 6 (destination !)	
4	2	[3, 6]	3	[1]	-	{0, 1}	Already visited bus 1	
5	2	[6]	6	[1]	-	{0, 1}	Destination reached	

```

        q.push(nextStop);

busStopVisited.insert(nextStop);
    }
    busVisited.insert(bus);
}
}
++level;
}

return -1; // If destination is not
reachable
}

int main() {
// Hardcoded input values
vector<vector<int>> routes = {
    {1, 2, 7},
    {3, 6, 7}
};
int src = 1; // source bus stop
int dest = 6; // destination bus stop

cout <<
numBusesToDestination(routes, src,
dest) << endl;

return 0;
}

```

Output:-
2

❖ Result:

The level when we reach stop 6 is **2**, but since levels are **incremented after each BFS layer**, and the first bus was taken at level 0:

☞ **Minimum buses required = 2**

◀ Final Output:

2

Coloring Border in C++

```
#include <iostream>
#include <vector>

using namespace std;

vector<vector<int>> dirs = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}};

void dfs(vector<vector<int>>& grid, int row, int col, int clr) {
    grid[row][col] = -clr;
    int count = 0;

    for (auto dir : dirs) {
        int rowdash = row + dir[0];
        int coldash = col + dir[1];

        if (rowdash < 0 || coldash < 0 || rowdash >= grid.size() || coldash >= grid[0].size() || abs(grid[rowdash][coldash]) != clr) {
            continue;
        }

        count++;

        if (grid[rowdash][coldash] == clr) {
            dfs(grid, rowdash, coldash, clr);
        }
    }

    if (count == 4) {
        grid[row][col] = clr;
    }
}

void coloring_border(vector<vector<int>>& grid, int row, int col, int color) {
    dfs(grid, row, col, grid[row][col]);

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

int main() {
    // Hardcoded input
    int m = 4;
    int n = 4;
    vector<vector<int>> arr = {
        {2, 1, 3, 4},
        {1, 2, 2, 2},
        {3, 2, 2, 2},
        {1, 2, 2, 2}
    };
    int row = 1;
    int col = 1;
    int color = 3;
}
```

Input:

```
grid = [
    {2, 1, 3, 4},
    {1, 2, 2, 2},
    {3, 2, 2, 2},
    {1, 2, 2, 2}
]
start = (1, 1)
color = 3
```

⌚ Initial Color at (1, 1): 2

⚡ DFS Dry Run (Marking Border)

Step	Cell	Action	Count of Same Color Neighbors	Final Cell State
1	(1,1)	Mark -2, recurse	0 → Recursing neighbors	-2
2	(1,2)	Mark -2, recurse	0 → Recursing	-2
3	(1,3)	Mark -2, recurse	0 → Recursing	-2
4	(2,3)	Mark -2, recurse	0	-2
5	(2,2)	Mark -2, recurse	1	-2
6	(2,1)	Mark -2, recurse	2	-2
7	(3,1)	Mark -2, recurse	0	-2
8	(3,2)	Mark -2, recurse	1	-2
9	(3,3)	Mark -2, recurse	1	-2

Once recursion returns, it checks `count == 4`. If true, the cell is fully surrounded by the same component → restore it to 2. Otherwise, it's on border → leave as -2.

Only cell (2,2) has all 4 neighbors of same component → reset to 2.

✍ Coloring Step:

- Any cell still marked as -2 → set to

```

coloring_border(arr, row, col, color);

// Print the modified grid
for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++) {
        cout << arr[i][j] << "\t";
    }
    cout << endl;
}

return 0;
}

```

new color = 3

❖ Final Output Grid:

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

📋 Dry Run Summary Table (Key Points):

Cell Was Visited Final Value

(1,1)	✓	3
(1,2)	✓	3
(1,3)	✓	3
(2,1)	✓	3
(2,2)	✓	2
(2,3)	✓	3
(3,1)	✓	3
(3,2)	✓	3
(3,3)	✓	3

Output:-

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

Min Cost to collect all cities in C++

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

struct Edge {
    int v;
    int wt;

    Edge(int nbr, int weight) {
        this->v = nbr;
        this->wt = weight;
    }
};

struct CompareEdge {
    bool operator()(const Edge& e1, const Edge& e2) {
        return e1.wt > e2.wt; // Min-Heap based on edge weight
    }
};

int main() {
    // Hardcoded input
    int vtces = 7;
    int edges = 8;
    vector<vector<Edge>> graph(vtces);

    // Hardcoded edges
    vector<vector<int>> hardcoded_edges = {
        {0, 1, 10},
        {1, 2, 10},
        {2, 3, 10},
        {0, 3, 40},
        {3, 4, 2},
        {4, 5, 3},
        {5, 6, 3},
        {4, 6, 8}
    };

    // Populating the graph with hardcoded edges
    for (auto& edge : hardcoded_edges) {
        int v1 = edge[0];
        int v2 = edge[1];
        int wt = edge[2];
        graph[v1].emplace_back(v2, wt);
        graph[v2].emplace_back(v1, wt);
    }

    int ans = 0;
    priority_queue<Edge, vector<Edge>, CompareEdge> pq;
    vector<bool> vis(vtces, false);
    pq.push(Edge(0, 0)); // Start with any vertex (0 in this case) with 0 weight

    while (!pq.empty()) {
        Edge rem = pq.top();
        pq.pop();

        if (vis[rem.v]) {

```

Core Concepts in the Code:

- Uses a **priority queue (min-heap)** to always pick the edge with the **least weight**.
- Starts from vertex 0.
- Adds edge weights to the total MST weight only when visiting **unvisited vertices**.
- vis[] tracks visited vertices.

■ Hardcoded Graph (7 vertices, 8 edges):

Edges:
{v1, v2, wt}
{0, 1, 10}
{1, 2, 10}
{2, 3, 10}
{0, 3, 40}
{3, 4, 2}
{4, 5, 3}
{5, 6, 3}
{4, 6, 8}

□ Dry Run Table: Prim's MST

Step	Vertex Visited	Edge Added (from)	Weight Added	Total MST Weight	Priority Queue (next min weight edges)
1	0	- (start)	0	0	(1,10), (3,40)
2	1	0 → 1	10	10	(2,10), (3,40)
3	2	1 → 2	10	20	(3,10), (3,40)
4	3	2 → 3	10	30	(4,2), (3,40)
5	4	3 → 4	2	32	(5,3), (6,8), (3,40)
6	5	4 → 5	3	35	(6,3), (6,8), (3,40)

```

        continue;
    }
vis[rem.v] = true;
ans += rem.wt;

for (Edge nbr : graph[rem.v]) {
    if (!vis[nbr.v]) {
        pq.push(nbr);
    }
}

cout << ans << endl;

return 0;
}

```

7	6	5 → 6	3	38	(6,8), (3,40) → both discarded (visited)
---	---	-------	---	----	--

✓ MST Total Weight: 38

Even though there's a 40-weight edge from 0 to 3, we never pick it because we reach 3 through a cheaper path (0→1→2→3).

▣ Output:

38

Output:-
38

Negative Wt Cycle Detection in C++

```
#include <iostream>
#include <vector>
#include <climits>

using namespace std;

struct Edge {
    int u, v, weight;
};

bool isNegativeWeightCycle(int n, vector<Edge>& edges)
{
    vector<int> dist(n, INT_MAX);
    dist[0] = 0; // Starting from vertex 0

    // Relaxation process
    for (int i = 0; i < n - 1; ++i) {
        for (const auto& edge : edges) {
            if (dist[edge.u] != INT_MAX && dist[edge.u] + edge.weight < dist[edge.v]) {
                dist[edge.v] = dist[edge.u] + edge.weight;
            }
        }
    }

    // Checking for negative weight cycles
    for (const auto& edge : edges) {
        if (dist[edge.u] != INT_MAX && dist[edge.u] + edge.weight < dist[edge.v]) {
            return true; // Negative weight cycle detected
        }
    }

    return false; // No negative weight cycle found
}

int main()
{
    // Hardcoded input
    int n = 3; // Number of vertices
    int m = 3; // Number of edges
    vector<Edge> edges = {{0, 1, -1}, {1, 2, -4}, {2, 0, 3}}; // Edges with (u, v, weight)

    if (isNegativeWeightCycle(n, edges)) {
        cout << "1\n"; // Negative weight cycle detected
    } else {
        cout << "0\n"; // No negative weight cycle found
    }

    return 0;
}
```

Bellman-Ford Key Idea:

- Perform $n - 1$ iterations relaxing all edges.
- Then **one more iteration** to see if **any distance still improves** → indicates a **negative cycle**.

Input:

```
n = 3
edges = {
    {0, 1, -1},
    {1, 2, -4},
    {2, 0, 3}
}
```

Dry Run Table (Relaxation)

Initial **dist**:

[0, ∞, ∞]

Iteration 1:

Edge	Condition	Action	Updated dist
0 → 1 -1	$0 + (-1) < \infty$	$\text{dist}[1] = -1$	[0, -1, ∞]
1 → 2 -4	$-1 + (-4) < \infty$	$\text{dist}[2] = -5$	[0, -1, -5]
2 → 0 +3	$-5 + 3 = -2 < 0$	$\text{dist}[0] = -2$	[-2, -1, -5]

Iteration 2:

Edge	Condition	Action	Updated dist
0 → 1 -1	$-2 - 1 = -3 < -1$	$\text{dist}[1] = -3$	[-2, -3, -5]
1 → 2 -4	$-3 - 4 = -7 < -5$	$\text{dist}[2] = -7$	[-2, -3, -7]
2 → 0 +3	$-7 + 3 = -4 < -2$	$\text{dist}[0] = -4$	[-4, -3, -7]

Extra Iteration – Check for

Negative Cycle

Edge	Condition	Result
$0 \rightarrow 1$ -1 -3	$-4 + (-1) = -5 < 0$	↙ Negative cycle!

↙ Conclusion:

- A **negative weight cycle** exists.
- Specifically: $0 \rightarrow 1 \rightarrow 2 \rightarrow 0$ forms a cycle with total weight: $-1 + (-4) + 3 = -2$

▣ Output:

1

Output:-
1

No of Distinct Island in C++																					
<pre>#include <iostream> #include <vector> #include <unordered_set> using namespace std; // Function prototypes void dfs(vector<vector<int>>& arr, int row, int col, string& psf); int numDistinctIslands(vector<vector<int>>& arr); // Depth-first search to mark all connected land cells of // an island void dfs(vector<vector<int>>& arr, int row, int col, string& psf) { arr[row][col] = 0; // Marking current cell as visited int n = arr.size(); int m = arr[0].size(); // Directions: up, right, down, left vector<pair<int, int>> dirs = {{-1, 0}, {0, 1}, {1, 0}, {0, -1}}; string dirStr = "urdl"; // Corresponding directions characters for (int i = 0; i < 4; ++i) { int newRow = row + dirs[i].first; int newCol = col + dirs[i].second; if (newRow >= 0 && newRow < n && newCol >= 0 && newCol < m && arr[newRow][newCol] == 1) { psf += dirStr[i]; // Append direction character to path string dfs(arr, newRow, newCol, psf); } } psf += "a"; // Append anchor to indicate end of island path } // Function to find number of distinct islands int numDistinctIslands(vector<vector<int>>& arr) { int n = arr.size(); if (n == 0) return 0; int m = arr[0].size(); unordered_set<string> islands; // Set to store distinct island paths for (int i = 0; i < n; ++i) { for (int j = 0; j < m; ++j) { if (arr[i][j] == 1) { string psf = "x"; // Starting character to represent new island dfs(arr, i, j, psf); islands.insert(psf); // Insert island path into set } } } return islands.size(); // Return the number of distinct }</pre>	<p>Key Concepts:</p> <ul style="list-style-type: none"> An island is a group of 1s connected horizontally or vertically. Each island is converted into a path string (psf) using DFS with directional encoding (u, r, d, l, and a for backtracking). The unordered_set stores these path strings to count unique island shapes. <p>Input Grid:</p> <pre>1 0 0 0 1 0 1 1 1</pre> <p>Key for DFS path string (psf):</p> <ul style="list-style-type: none"> x → Start of island u → Up r → Right d → Down l → Left a → Backtrack (anchor) <p>Dry Run Table:</p> <table border="1"> <thead> <tr> <th>Island #</th><th>Starting Cell</th><th>DFS Path (psf)</th><th>Shape Description</th><th>Is Unique?</th></tr> </thead> <tbody> <tr> <td>1</td><td>(0, 0)</td><td>xa</td><td>Single cell</td><td>✓ Yes</td></tr> <tr> <td>2</td><td>(1, 1)</td><td>xa</td><td>Single cell</td><td>✗ No</td></tr> <tr> <td>3</td><td>(2, 0)</td><td>xrraa</td><td>Horizontal chain (L-shape)</td><td>✓ Yes</td></tr> </tbody> </table> <p>Final Set of Unique Island Shapes:</p> <p>Shape Path</p> <pre>xa xrraa</pre>	Island #	Starting Cell	DFS Path (psf)	Shape Description	Is Unique?	1	(0, 0)	xa	Single cell	✓ Yes	2	(1, 1)	xa	Single cell	✗ No	3	(2, 0)	xrraa	Horizontal chain (L-shape)	✓ Yes
Island #	Starting Cell	DFS Path (psf)	Shape Description	Is Unique?																	
1	(0, 0)	xa	Single cell	✓ Yes																	
2	(1, 1)	xa	Single cell	✗ No																	
3	(2, 0)	xrraa	Horizontal chain (L-shape)	✓ Yes																	

```
islands
}

int main() {
    // Hardcoded input
    vector<vector<int>> arr = {
        {1, 0, 0},
        {0, 1, 0},
        {1, 1, 1}
    };

    // Calculating number of distinct islands
    cout << numDistinctIslands(arr) << endl;

    return 0;
}
```

Output:-
2

✓ Output:

2

No of enclaves in C++					
<pre>#include <iostream> #include <vector> using namespace std; void dfs(vector<vector<int>>& arr, int i, int j) { if (i < 0 j < 0 i >= arr.size() j >= arr[0].size() arr[i][j] == 0) { return; } arr[i][j] = 0; dfs(arr, i + 1, j); dfs(arr, i - 1, j); dfs(arr, i, j + 1); dfs(arr, i, j - 1); } int numEnclaves(vector<vector<int>>& arr) { int m = arr.size(); int n = arr[0].size(); // Marking connected components touching the // boundaries for (int i = 0; i < m; ++i) { for (int j = 0; j < n; ++j) { if ((i == 0 j == 0 i == m - 1 j == n - 1) && arr[i][j] == 1) { dfs(arr, i, j); } } } // Counting remaining land cells int count = 0; for (int i = 0; i < m; ++i) { for (int j = 0; j < n; ++j) { if (arr[i][j] == 1) { ++count; } } } return count; } int main() { int m = 4, n = 4; vector<vector<int>> arr = { {0, 0, 0, 0}, {1, 0, 1, 0}, {0, 1, 1, 0}, {0, 0, 0, 0} }; int result = numEnclaves(arr); cout << result << endl; return 0; }</pre>					
	0	1	2	3	
0	0	0	0	0	
1	1	0	1	0	
2	0	1	1	0	
3	0	0	0	0	

Dry Run Table – Step-by-Step					
 Step 1: Mark boundary-connected 1s using DFS					
Check all boundary cells and run DFS from any land (1) on the edge:					
Cell	Is Boundary?	Is Land?	DFS Run?	Action	
(0,x)/(x,0)/(3,x)/(x,3)	✓ Yes	Mixed	✓ If land	DFS removes (1,0) only	

✓ Only **(1,0)** is a boundary land → DFS marks it and its connected land 0.

After DFS update, grid becomes:					
	0	**1**	**2**	**3**	
0	0	0	0	0	
1	0	0	1	0	
2	0	1	1	0	
3	0	0	0	0	

Step 2: Count remaining 1s (enclaves)			
Cell	Value	Is Land (1)?	Count += 1?
(1,2)	1	✓	✓ (count=1)
(2,1)	1	✓	✓ (count=2)

Cell	Value	Is Land (1)?	Count += 1?
(2,2)	1	✓	✓ (count=3)

 Total enclave land cells = **3**

✓ Final Output:

3

↻ Summary Table:

Phase	Operation	Result
Boundary DFS	Remove all 1s connected to boundary	(1,0) set to 0
Enclave Counting	Count remaining 1s in the grid	3
Final Return Value	numEnclaves ()	3

Output:-

3

Optimize water distribution in C++

```
#include <iostream>
#include <vector>
#include <queue>
#include <utility>

using namespace std;

class Pair {
public:
    int vtx;
    int wt;
    Pair(int vtx, int wt) {
        this->vtx = vtx;
        this->wt = wt;
    }
    bool operator>(const Pair& other) const {
        return this->wt > other.wt;
    }
};

int minCostToSupplyWater(int n, vector<int>& wells, vector<vector<int>>& pipes) {
    vector<vector<Pair>> graph(n + 1);
    for (const auto& pipe : pipes) {
        int u = pipe[0];
        int v = pipe[1];
        int wt = pipe[2];
        graph[u].emplace_back(v, wt);
        graph[v].emplace_back(u, wt);
    }
    for (int i = 1; i <= n; ++i) {
        graph[i].emplace_back(0, wells[i - 1]);
        graph[0].emplace_back(i, wells[i - 1]);
    }

    int ans = 0;
    priority_queue<Pair, vector<Pair>, greater<Pair>> pq;
    pq.emplace(0, 0);
    vector<bool> vis(n + 1, false);

    while (!pq.empty()) {
        Pair rem = pq.top();
        pq.pop();
        if (vis[rem.vtx]) continue;
        ans += rem.wt;
        vis[rem.vtx] = true;
        for (const Pair& nbr : graph[rem.vtx]) {
            if (!vis[nbr.vtx]) {
                pq.push(nbr);
            }
        }
    }
    return ans;
}

int main() {
    int v = 3, e = 2;
    vector<int> wells = {1, 2, 2};
    vector<vector<int>> pipes = {{1, 2, 1}, {2, 3, 1}};
}
```

Input:

- Number of houses (**n**) = 3
- Wells: [1, 2, 2] → Cost to build wells at house 1, 2, 3
- Pipes:

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

🔧 Graph Construction (Adjacency List):

Node	Connections
0	(1,1), (2,2), (3,2)
1	(2,1), (0,1)
2	(1,1), (3,1), (0,2)
3	(2,1), (0,2)

📊 Dry Run of Prim's Algorithm:

Step	Min Edge Picked (u→v, wt)	Added to MST	MST Cost	Visited Nodes	Heap Contents After Push
1	(0→0, 0)	0	0	{0}	(1,1), (2,2), (3,2)
2	(0→1, 1)	1	1	{0,1}	(2,2), (3,2), (2,1)
3	(1→2, 1)	2	2	{0,1,2}	(3,2), (2,2), (3,1)
4	(2→3, 1)	3	3	{0,1,2,3}	Remaining edges ignored (already visited nodes)

↙ All nodes visited.

↙ Final Output:

3

🎥 Explanation:

- Use well at house 1: cost 1

```
cout << minCostToSupplyWater(v, wells, pipes) <<  
endl;  
  
    return 0;  
}
```

- Use **pipe 1–2**: cost 1
- Use **pipe 2–3**: cost 1
→ **Total = 3**

 This is cheaper than building all wells
($1+2+2=5$)

Output:-
3

Redundant connection in C++

```
#include <iostream>
#include <vector>

using namespace std;

class UnionFind {
public:
    vector<int> parent;
    vector<int> rank;

    UnionFind(int n) {
        parent.resize(n + 1);
        rank.resize(n + 1, 1);
        for (int i = 1; i <= n; ++i) {
            parent[i] = i;
        }
    }

    int find(int x) {
        if (parent[x] != x) {
            parent[x] = find(parent[x]); // Path compression
        }
        return parent[x];
    }

    void unionSets(int x, int y) {
        int rootX = find(x);
        int rootY = find(y);

        if (rootX != rootY) {
            if (rank[rootX] > rank[rootY]) {
                parent[rootY] = rootX;
            } else if (rank[rootX] < rank[rootY]) {
                parent[rootX] = rootY;
            } else {
                parent[rootY] = rootX;
                rank[rootX]++;
            }
        }
    }

    vector<int>
    findRedundantConnection(vector<vector<int>>& edges) {
        int n = edges.size();
        UnionFind uf(n);

        for (auto& edge : edges) {
            int u = edge[0];
            int v = edge[1];

            if (uf.find(u) == uf.find(v)) {
                return edge; // This edge is a redundant connection
            }
            uf.unionSets(u, v);
        }
        return {};
    }
}
```

You're given edges forming a graph. Initially, it's a tree (n nodes, $n-1$ edges), but one extra edge was added, forming a cycle.

Goal: Find the **redundant edge** forming the cycle.

📋 Input

```
edges = [
    {1, 2},
    {1, 3},
    {2, 3}
]
```

🎬 Initial Setup

- Nodes: 1, 2, 3
- parent[] = [0, 1, 2, 3] (0-index unused)
- rank[] = [0, 1, 1, 1]

📊 Dry Run Table (Union-Find Process)

Step	Edge	Find(u)	Find(v)	Same Root?	Action	Updated parent[]	Updated rank[]
1	1-2	1	2	✗ No	Union(1, 2)	[0, 1, 1, 3]	[0, 2, 1, 1]
2	1-3	1	3	✗ No	Union(1, 3)	[0, 1, 1, 1]	[0, 2, 1, 1]
3	2-3	1	1	✓ Yes	! Cycle found	—	—

⚡ Output

2 3

- Edge {2, 3} forms the cycle.
- It is **redundant**, and hence returned.

```
}

int main() {
    // Hardcoded input
    vector<vector<int>> edges = {
        {1, 2},
        {1, 3},
        {2, 3}
    };

    vector<int> result =
findRedundantConnection(edges);
    cout << result[0] << " " << result[1] <<
endl;

    return 0;
}
```

Output:-
3

AccountMerge in C++

```
#include <bits/stdc++.h>
using namespace std;
//User function Template for C++
class DisjointSet {
    vector<int> rank, parent, size;
public:
    DisjointSet(int n) {
        rank.resize(n + 1, 0);
        parent.resize(n + 1);
        size.resize(n + 1);
        for (int i = 0; i <= n; i++) {
            parent[i] = i;
            size[i] = 1;
        }
    }

    int findUPar(int node) {
        if (node == parent[node])
            return node;
        return parent[node] = findUPar(parent[node]);
    }

    void unionByRank(int u, int v) {
        int ulp_u = findUPar(u);
        int ulp_v = findUPar(v);
        if (ulp_u == ulp_v) return;
        if (rank[ulp_u] < rank[ulp_v]) {
            parent[ulp_u] = ulp_v;
        } else if (rank[ulp_v] < rank[ulp_u]) {
            parent[ulp_v] = ulp_u;
        } else {
            parent[ulp_v] = ulp_u;
            rank[ulp_u]++;
        }
    }

    void unionBySize(int u, int v) {
        int ulp_u = findUPar(u);
        int ulp_v = findUPar(v);
        if (ulp_u == ulp_v) return;
        if (size[ulp_u] < size[ulp_v]) {
            parent[ulp_u] = ulp_v;
            size[ulp_v] += size[ulp_u];
        } else {
            parent[ulp_v] = ulp_u;
            size[ulp_u] += size[ulp_v];
        }
    }
};

class Solution {
public:
    vector<vector<string>>
accountsMerge(vector<vector<string>> &details) {
    int n = details.size();
    DisjointSet ds(n);
    sort(details.begin(), details.end());
    unordered_map<string, int> mapMailNode;
```

Input

```
{
    {"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"}
}
```

Let's assume these are indexed from 0 to 5.

Step 1: Mapping Emails to Accounts with Union

We initialize a map `mail → nodeIndex`. As we traverse, if we see a repeated email, we perform **unionBySize** between the current index and the one in the map.

Index	Account Name	Emails	Action
0	John	j1, j2, j3	Add all emails to map → j1 → 0, j2 → 0, j3 → 0
1	John	j4	j4 → 1
2	Raj	r1, r2	r1 → 2, r2 → 2
3	John	j1 (seen), j5	Union(3, 0) since j1 → 0 → 3 belongs to same group as 0
4	Raj	r2 (seen), r3	Union(4, 2) since r2 → 2 → 4 belongs to same group as 2
5	Mary	m1	m1 → 5

After unions:

- Group 0 includes index 0 and 3 (due to shared j1)
- Group 2 includes index 2 and 4 (due to shared r2)

Step 2: Group Emails Based on Ultimate Parent (Union-Find)

We iterate over the map and collect emails in

```

for (int i = 0; i < n; i++) {
    for (int j = 1; j < details[i].size(); j++) {
        string mail = details[i][j];
        if (mapMailNode.find(mail) ==
mapMailNode.end()) {
            mapMailNode[mail] = i;
        }
        else {
            ds.unionBySize(i, mapMailNode[mail]);
        }
    }
}

vector<string> mergedMail[n];
for (auto it : mapMailNode) {
    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].size() == 0) continue;
    sort(mergedMail[i].begin(), mergedMail[i].end());
    vector<string> temp;
    temp.push_back(details[i][0]);
    for (auto it : mergedMail[i]) {
        temp.push_back(it);
    }
    ans.push_back(temp);
}
sort(ans.begin(), ans.end());
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;
vector<vector<string>> ans =
obj.accountsMerge(accounts);
for (auto acc : ans) {
    cout << acc[0] << ":";
    int size = acc.size();
    for (int i = 1; i < size; i++) {
        cout << acc[i] << " ";
    }
    cout << endl;
}
return 0;
}

```

the list `mergedMail[parent]`.

Example:

- $j_1 \rightarrow 0 \rightarrow \text{findUPar}(0) = 0$
- $j_5 \rightarrow 3 \rightarrow \text{findUPar}(3) = 0$ (after union)
- $r_3 \rightarrow 4 \rightarrow \text{findUPar}(4) = 2$

So we get:

Parent Index	Emails
0	j_1, j_2, j_3, j_5
1	j_4
2	r_1, r_2, r_3
5	m_1

Step 3: Construct Final Answer

We loop over `mergedMail[]`, and for each non-empty vector:

- Sort the emails
- Use the **name from the original account at that index**

Group	Name	Sorted Emails
0	John	j_1, j_2, j_3, j_5
1	John	j_4
2	Raj	r_1, r_2, r_3
5	Mary	m_1

Final Output

John:j1@com j2@com j3@com j5@com
John:j4@com
Mary:m1@com
Raj:r1@com r2@com r3@com

DSU Table View (Final Parents)

Let's print `findUPar(i)` for $i = 0$ to 5

Index	Account Name	Parent (after unions)
0	John	0
1	John	1
2	Raj	2
3	John	0

	Index	Account Name	Parent (after unions)
	4	Raj	2
	5	Mary	5

Output:-

John:j1@com j2@com j3@com j5@com

John:j4@com

Mary:m1@com

Raj:r1@com r2@com r3@com

Articulation Point in C++

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

//User function Template for C++

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;
        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 (child > 1 && parent == -1) {
        mark[node] = 1;
    }
}
public:
    vector<int> articulationPoints(int n, vector<int> adj[])
{
    vector<int> vis(n, 0);
    int tin[n];
    int low[n];
    vector<int> mark(n, 0);
    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] == 1) {
            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}
    };
}
```

Graph Overview

Given edges:

```
0 - 1
 |
 4
 / \
2 - 3
```

Adjacency List:

Node	Neighbors
0	1
1	0, 4
2	4, 3
3	2, 4
4	1, 2, 3

Variables Recap

- $\text{tin}[\text{node}]$: Time of first visit
- $\text{low}[\text{node}]$: Lowest reachable discovery time
- A node is an **articulation point** if:
 - Not root and $\text{low}[\text{child}] \geq \text{tin}[\text{node}]$
 - Root and has ≥ 2 children

🧠 DFS Trace Table

Step	Node	Parent	tin	low	Action & Reasoning
1	0	-1	1	1	Start DFS from 0
2	1	0	2	2	Visit from 0
3	4	1	3	3	Visit from 1
4	2	4	4	4	Visit from 4
5	3	2	5	5	Visit from 2
6	4	3	-	3	Back edge to 4
7	2	4	-	3	$\text{low}[2] = \min(4, 3)$
8	4	1	-	3	$\text{low}[4] = \min(3, 3)$
9	1	0	-	2	$\text{low}[1] = \min(2, 3)$
10	0	-1	-	1	Done

🔗 Articulation Point Analysis

We now check for articulation conditions.

- **Node 1:**

```

};

vector<int> adj[n];
for (auto it : edges) {
    int u = it[0], v = it[1];
    adj[u].push_back(v);
    adj[v].push_back(u);
}
Solution obj;
vector<int> nodes = obj.articulationPoints(n, adj);
for (auto node : nodes) {
    cout << node << " ";
}
cout << endl;
return 0;
}

```

Output:-

1 4

- $\text{low}[4] = 3 \geq \text{tin}[1] = 2 \rightarrow \checkmark$
articulation point
- **Node 4:**
 - $\text{low}[2] = 3 \geq \text{tin}[4] = 3$
 - $\text{low}[3] = 5 \geq \text{tin}[4] = 3 \rightarrow \checkmark$
articulation point
- **Node 0:**
 - Root with only 1 child $\rightarrow \times$ not
articulation point

Final Result

Articulation Points: 1 4

Bellman-Ford in C++

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

class Solution {
public:
    /* Function to implement Bellman Ford
     * edges: vector of vectors which represents the
     * graph
     * S: source vertex to start traversing graph with
     * V: number of vertices
     */
    vector<int> bellman_ford(int V,
vector<vector<int>>& edges, int S) {
        vector<int> dist(V, 1e8);
        dist[S] = 0;
        for (int i = 0; i < V - 1; i++) {
            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]) {
                    dist[v] = dist[u] +
wt;
                }
            }
        }
        // Nth relaxation to check 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(7, vector<int>(3));
    edges[0] = {3, 2, 6};
    edges[1] = {5, 3, 1};
    edges[2] = {0, 1, 5};
    edges[3] = {1, 5, -3};
    edges[4] = {1, 2, -2};
    edges[5] = {3, 4, -2};
    edges[6] = {2, 4, 3};

    int S = 0;
    Solution obj;
    vector<int> dist = obj.bellman_ford(V, edges, S);
    for (auto d : dist) {
        cout << d << " ";
    }
}
```

Initialization

Vertex	dist
0	0
1	∞
2	∞
3	∞
4	∞
5	∞

After each iteration of relaxation (V-1 = 5 times):

We'll update dist[] step by step, showing changes caused by each edge.

Iteration 1:

Process edges:

1. $0 \rightarrow 1$ (5) \rightarrow $dist[1] = 5$
2. $1 \rightarrow 2$ (-2) \rightarrow $dist[2] = 3$
3. $1 \rightarrow 5$ (-3) \rightarrow $dist[5] = 2$
4. $5 \rightarrow 3$ (1) \rightarrow $dist[3] = 3$
5. $3 \rightarrow 4$ (-2) \rightarrow $dist[4] = 1$
6. $2 \rightarrow 4$ (3) \rightarrow already $dist[4] = 1$ so not updated
7. Other edges don't apply yet.

Result:

$dist = [0, 5, 3, 3, 1, 2]$

Iteration 2 to 5:

Now that distances are optimal and no further relaxation improves any values, **no changes happen**.

Final dist[] after Bellman-Ford

Vertex	Final dist
0	0
1	5
2	3
3	3
4	1
5	2

```
    }  
    cout << endl;  
  
    return 0;  
}
```

✓ **Correct Output:**

0 5 3 3 1 2

Output:-

0 5 3 3 1 2

Bipartite in Depth First Search in C++

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

class Solution {
private:
    bool dfs(int node, int col, int color[], vector<int> adj[]) {
        color[node] = col;

        // traverse adjacent nodes
        for(auto it : adj[node]) {
            // if uncoloured
            if(color[it] == -1) {
                if(dfs(it, !col, color, adj) == false) return
false;
            }
            // if previously coloured and have the same
colour
            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 connected components
        for(int i = 0;i<V;i++) {
            if(color[i] == -1) {
                if(dfs(i, 0, color, adj) == false)
                    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(){
        // V = 4, E = 4
        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";
    }
}
```

Graph Construction (4 vertices, 4 edges):

addEdge(adj, 0, 2); // 0 - 2
addEdge(adj, 0, 3); // 0 - 3
addEdge(adj, 2, 3); // 2 - 3
addEdge(adj, 3, 1); // 3 - 1

Adjacency List:

Vertex	Neighbors
0	2, 3
1	3
2	0, 3
3	0, 2, 1

DFS Coloring Attempt:

- Initialize all colors as -1.
- Try to color graph with **two colors**: 0 and 1.

Dry Run Table

Node Visited	Action	Color Assigned	Stack/Call Stack	Conflict?
0	Start DFS	0	dfs(0, 0)	No
2	Visit from 0	1	dfs(2, 1)	No
3	Visit from 2	0	dfs(3, 0)	No
0	Already colored	0	Check if conflict with 0	✗ Match
1	Visit from 3	1	dfs(1, 1)	No
3	Already colored	0	Check if conflict with 1	✗ Match
2	Already colored	1	Check if conflict with 3 (expect 1, found 0)	✗ Conflict!

At this point, DFS at node 3 sees that its neighbor 2 is also colored 1, and this **violates the bipartite condition**, because both are expected to have **opposite colors**.

Final Result:

0

```
    return 0;  
}
```

Output:-
0

DFS Cycle undirected in C++

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

class Solution {
private:
    bool dfs(int node, int parent, int vis[], vector<int> adj[]) {
        vis[node] = 1;
        // visit adjacent nodes
        for(auto adjacentNode: adj[node]) {
            // unvisited adjacent node
            if(!vis[adjacentNode]) {
                if(dfs(adjacentNode, node, vis, adj) == true)
                    return true;
            }
            // visited node but not a parent
            node
            else if(adjacentNode != parent)
                return true;
        }
        return false;
    }
public:
    // Function to detect cycle in an
    undirected graph.
    bool isCycle(int V, vector<int> adj[])
    {
        int vis[V] = {0};
        // for graph with connected
        components
        for(int i = 0;i<V;i++) {
            if(!vis[i]) {
                if(dfs(i, -1, vis, adj) == true)
                    return true;
            }
        }
        return false;
    }
};

int main() {
    // V = 4, E = 2
    vector<int> adj[4] = {{}, {2}, {1, 3}, {2}};
    Solution obj;
    bool ans = obj.isCycle(4, adj);
    if (ans)
        cout << "1\n";
    else
        cout << "0\n";
    return 0;
}
```

Graph Input (V = 4):

```
vector<int> adj[4] = {
    {},      // Node 0: No edges
    {2},     // Node 1: Connected to 2
    {1, 3},  // Node 2: Connected to 1 and 3
    {2}      // Node 3: Connected to 2
};
```

So the actual edges are:

- 1 - 2
- 2 - 3

This graph is a simple path, not a cycle.

Dry Run Table (DFS traversal):

Step	Current Node	Parent	vis[] Status	Adjacent Nodes	Action	Cycle Detected?
1	0	-1	[1, 0, 0, 0]	{}	No adj nodes	No
2	1	-1	[1, 1, 0, 0]	{2}	DFS to 2	No
3	2	1	[1, 1, 1, 0]	{1, 3}	1 is parent, DFS to 3	No
4	3	2	[1, 1, 1, 1]	{2}	2 is parent, backtrack	No

No cycle detected

The code correctly determines that no adjacent node points back to a **previously visited node that's not its parent**, so there is **no cycle**.

Output:

0

Output:-

0

Depth First Search in C++

```
#include <iostream>
#include <vector>

using namespace std;

class DFSDirected {
public:
    static vector<int> dfs(int s, vector<bool>& vis,
                           vector<vector<int>>& adj, vector<int>& ls) {
        vis[s] = true;
        ls.push_back(s);
        for (int it : adj[s]) {
            if (!vis[it]) {
                dfs(it, vis, adj, ls);
            }
        }
        return ls;
    }
};

int main() {
    int V = 5;
    vector<bool> vis(V + 1, false);
    vector<int> ls;
    vector<vector<int>> adj(V + 1);

    adj[1].push_back(3);
    adj[1].push_back(2);
    adj[3].push_back(4);
    adj[4].push_back(5);

    vector<vector<int>> res;
    for (int i = 1; i <= V; i++) {
        if (!vis[i]) {
            vector<int> ls;
            res.push_back(DFSDirected::dfs(i, vis, adj, ls));
        }
    }

    for (const auto& component : res) {
        for (int node : component) {
            cout << node << " ";
        }
        cout << endl;
    }

    return 0;
}
```

Output:-
1 3 4 5 2

Graph Construction:

```
int V = 5;
adj[1].push_back(3); // 1 → 3
adj[1].push_back(2); // 1 → 2
adj[3].push_back(4); // 3 → 4
adj[4].push_back(5); // 4 → 5
```

So the graph looks like:

```
1 → 2
↓
3 → 4 → 5
```

☞ DFS Traversal (starting from unvisited nodes)

Looping over $i = 1$ to 5 :

i	vis[i]	DFS Starts?	DFS Order (Component)
1	false	Yes	$1 \rightarrow 3 \rightarrow 4 \rightarrow 5$, then $2 \rightarrow$
2	true	No	Already visited from 1
3	true	No	Already visited from 1
4	true	No	Already visited from 1
5	true	No	Already visited from 1

Note: 2 is visited after 1, since it's a neighbor of 1 and called later in the loop.

So only **one DFS call** is needed, and it covers **all reachable nodes from 1**.

▲ DFS Order (Component):

- From node 1: $1 \rightarrow 3 \rightarrow 4 \rightarrow 5$, and then the loop in DFS continues with 2.

So final traversal list:

1 3 4 5 2

☰ Output:

1 3 4 5 2

Dijkstra in C++

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

class Solution
{
public:
    // Function to find the shortest distance of all
    // the vertices
    // from the source vertex S.
    vector<int> dijkstra(int V,
    vector<vector<int>> adj[], int S)
    {

        // Create a priority queue for storing the
        // nodes as a pair {dist,node}
        // where dist is the distance from source to
        // the node.
        priority_queue<pair<int, int>,
        vector<pair<int, int>>, greater<pair<int, int>>>
        pq;

        // Initialising distTo list with a large
        // number to
        // indicate the nodes are unvisited initially.
        // This list contains distance from source to
        // the nodes.
        vector<int> distTo(V, INT_MAX);

        // Source initialised with dist=0.
        distTo[S] = 0;
        pq.push({0, S});

        // Now, pop the minimum distance node
        // first from the min-heap
        // and traverse for all its adjacent nodes.
        while (!pq.empty())
        {
            int node = pq.top().second;
            int dis = pq.top().first;
            pq.pop();

            // Check for all adjacent nodes of the
            // popped out
            // element whether the prev dist is larger
            // than current or not.
            for (auto it : adj[node])
            {
                int v = it[0];
                int w = it[1];
                if (dis + w < distTo[v])
                {
                    distTo[v] = dis + w;

                    // If current distance is smaller,
                    // push it into the queue.
                    pq.push({dis + w, v});
                }
            }
        }

        // Return the list containing shortest
        // distances
    }
}
```

Graph Setup

Given:

- **Vertices (V):** 3
- **Source (S):** 2
- **Adjacency list (adj):**

$\text{adj}[0] = \{\{1, 1\}, \{2, 6\}\};$
 $\text{adj}[1] = \{\{2, 3\}, \{0, 1\}\};$
 $\text{adj}[2] = \{\{1, 3\}, \{0, 6\}\};$

This translates to:

From	To	Weight
0	1	1
0	2	6
1	2	3
1	0	1
2	1	3
2	0	6

↘ Dijkstra's Algorithm

Start from source 2, initialize:

$\text{distTo} = [\infty, \infty, 0]$
 $\text{pq} = [(0, 2)]$

Now iterate:

Step	Node	Pop (dist, node)	Neighbors	Update Distances	pq After
1	2	(0, 2)	(1,3), (0,6)	$\text{dist}[1] = 3, (3,1),$ $\text{dist}[0] = 6 (6,0)$	
2	1	(3, 1)	(2,3), (0,1)	$\text{dist}[0] = \min(6, 4) = 4 (4,0),$ 4	
3	0	(4, 0)	(1,1), (2,6)	$\text{dist}[1]$ already 3 < 6, 0 5 → skip	
4	0	(6, 0)	-	Already visited with smaller	

┌ Final Distance Array:

$\text{res} = [4, 3, 0]$

Means:

Vertex	Shortest Distance from Source (2)
0	4

```

    // from source to all the nodes.
    return distTo;
}

};

int main()
{
    // Driver code.
    int V = 3, E = 3, S = 2;
    vector<vector<int>> adj[V];
    vector<vector<int>> edges;
    vector<int> v1{1, 1}, v2{2, 6}, v3{2, 3}, v4{0, 1},
    v5{1, 3}, v6{0, 6};
    int i = 0;
    adj[0].push_back(v1);
    adj[0].push_back(v2);
    adj[1].push_back(v3);
    adj[1].push_back(v4);
    adj[2].push_back(v5);
    adj[2].push_back(v6);

    Solution obj;
    vector<int> res = obj.dijkstra(V, adj, S);

    for (int i = 0; i < V; i++)
    {
        cout << res[i] << " ";
    }
    cout << endl;
    return 0;
}

```

Vertex	Shortest Distance from Source (2)
1	3
2	0 (source itself)

Output:

4 3 0

Output:-

4 3 0

Disjoint Set in C++

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

vector<int> parent, rankVec; // Renamed rank to
rankVec

void makeSet(int n) {
    parent.resize(n + 1);
    rankVec.resize(n + 1, 0); // Use rankVec here
    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 ulp_u = findUPar(u); // ultimate parent of u
    int ulp_v = findUPar(v); // ultimate parent of v
    if (ulp_u == ulp_v) return; // already in the same set

    // Union by rank
    if (rankVec[ulp_u] < rankVec[ulp_v]) { // Use
rankVec here
        parent[ulp_u] = ulp_v;
    } else if (rankVec[ulp_u] > rankVec[ulp_v]) { // Use
rankVec here
        parent[ulp_v] = ulp_u;
    } else {
        parent[ulp_v] = ulp_u;
        rankVec[ulp_u]++;
    }
}

int main() {
    int n = 7; // Number of elements
    makeSet(n);

    unionByRank(1, 2);
    unionByRank(2, 3);
    unionByRank(4, 5);
    unionByRank(6, 7);
    unionByRank(5, 6);

    // Check if 3 and 7 are in the same set
    if (findUPar(3) == findUPar(7)) {
        cout << "Same\n";
    } else {
        cout << "Not same\n";
    }

    unionByRank(3, 7);

    // Check again if 3 and 7 are in the same set
    if (findUPar(3) == findUPar(7)) {

```

Initial Setup

You're working with $n = 7$, i.e., elements from 1 to 7.

makeSet(n):

- $\text{parent}[i] = i$ for all $i \in [0, 7]$
- $\text{rankVec}[i] = 0$ initially

❖ Union Operations

Step	Operation	Resulting Union	Parent Array	Rank Array (rankVec)
1	union(1, 2)	1 becomes parent of 2	[0, 1, 1, 3, 4, 5, 6, 7]	[0, 1, 0, 0, 0, 0, 0, 0]
2	union(2, 3)	1 becomes parent of 3 (via 2)	[0, 1, 1, 1, 4, 5, 6, 7]	[0, 1, 0, 0, 0, 0, 0, 0]
3	union(4, 5)	4 becomes parent of 5	[0, 1, 1, 1, 4, 4, 6, 7]	[0, 1, 0, 0, 1, 0, 0, 0]
4	union(6, 7)	6 becomes parent of 7	[0, 1, 1, 1, 4, 4, 6, 6]	[0, 1, 0, 0, 1, 0, 1, 0]
5	union(5, 6)	4 becomes parent of 6 (via 5)	[0, 1, 1, 1, 4, 4, 4, 6]	[0, 1, 0, 0, 2, 0, 1, 0]

? First Check: findUPar(3) vs findUPar(7)

- $\text{findUPar}(3) \rightarrow$ follows to 1
- $\text{findUPar}(7) \rightarrow 7 \rightarrow 6 \rightarrow 4$
- So: $1 \neq 4 \rightarrow$ Output: **Not same**

❖ union(3, 7)

- Ultimate parents: 1 and 4
- Both have rank 2 \rightarrow tie, choose one (say 1) as parent, and increment rank

Result	Updated Parent Array	Updated Rank Array
1 becomes parent of 4	[0, 1, 1, 1, 4, 4, 6]	[0, 3, 0, 0, 2, 0, 1, 0]

? Second Check: findUPar(3) vs findUPar(7)

```
    cout << "Same\n";
} else {
    cout << "Not same\n";
}

return 0;
}
```

- findUPar(3) → 1
- findUPar(7) → 7 → 6 → 4 → 1
- So: 1 == 1 → Output: **Same**

❖ Final Output

Not same
Same

Output:-

Not same
Same

Find eventual safe state in C++

```
#include <bits/stdc++.h>
using namespace std;
class Solution {
private:
    bool dfsCheck(int node, vector<int> adj[], int vis[], int pathVis[], int check[]) {
        vis[node] = 1;
        pathVis[node] = 1;
        check[node] = 0;
        // traverse for adjacent nodes
        for (auto it : adj[node]) {
            // when the node is not visited
            if (!vis[it]) {
                if (dfsCheck(it, adj, vis, pathVis, check) == true) {
                    check[node] = 0;
                    return true;
                }
            }
            // if the node has been previously visited
            // but it has to be visited on the same path
            else if (pathVis[it]) {
                check[node] = 0;
                return true;
            }
        }
        check[node] = 1;
        pathVis[node] = 0;
        return false;
    }
public:
    vector<int> eventualSafeNodes(int V, vector<int> adj[]) {
        int vis[V] = {0};
        int pathVis[V] = {0};
        int check[V] = {0};
        vector<int> safeNodes;
        for (int i = 0; i < V; i++) {
            if (!vis[i]) {
                dfsCheck(i, adj, vis, pathVis, check);
            }
        }
        for (int i = 0; i < V; i++) {
            if (check[i] == 1) safeNodes.push_back(i);
        }
        return safeNodes;
    }
};

int main() {
    //V = 12;
    vector<int> adj[12] = {{1}, {2}, {3}, {4, 5}, {6}, {6}, {7}, {}, {1, 9}, {10}, {8}, {9}};
    int V = 12;
    Solution obj;
    vector<int> safeNodes = obj.eventualSafeNodes(V, adj);
    for (auto node : safeNodes) {

```

Goal

We want to find all the **eventual safe nodes** in a **directed graph**, i.e., nodes from which **every path eventually ends in a terminal node** (a node with no outgoing edges). This is solved using **DFS cycle detection**.

Key Concepts

- `vis[]` → marks if a node has been visited.
- `pathVis[]` → tracks the current recursion path.
- `check[]` → 1 if node is *safe*, 0 if not.

A node is **not safe** if:

- A cycle is detected starting from it (or reachable from it).

Input Graph (Adjacency List)

```
0 → 1
1 → 2
2 → 3
3 → 4,5
4 → 6
5 → 6
6 → 7
7 → {} ← terminal node
8 → 1,9
9 → 10
10 → 8
11 → 9
```

DFS Cycle Detection

Let's go through the DFS starting from each unvisited node:

Node	Path	Cycle Detected	Safe?
0	0→1→2→3→4→6→7	No	✓ Yes
1	Already visited from 0	-	✓ Yes
2	Already visited from 0	-	✓ Yes
3	Already visited from 0	-	✓ Yes
4	Already visited from 0	-	✓ Yes
5	5→6→7	No	✓ Yes
6	Already visited	-	✓ Yes
7	Terminal	No	✓ Yes
8	8→1→... (already	✓ Yes	✗ No

```

    cout << node << " ";
}
cout << endl
return 0;
}

```

	visited) AND 8→9→10→8 (cycle)		
9	9→10→8→9	✓ Yes	✗ No
10	10→8→9→10	✓ Yes	✗ No
11	11→9→cycle	✓ Yes	✗ No

✓ Safe Nodes

From the table above, the safe nodes are:

0 1 2 3 4 5 6 7

Output:-

0 1 2 3 4 5 6 7

Floyd-Warshall in C++																									
#include <bits/stdc++.h> using namespace std; class Solution { public: void shortest_distance(vector<vector<int>>&matrix) { int n = matrix.size(); for (int i = 0; i < n; i++) { for (int j = 0; j < n; j++) { if (matrix[i][j] == -1) { matrix[i][j] = 1e9; } if (i == j) matrix[i][j] = 0; } } for (int k = 0; k < n; k++) { for (int i = 0; i < n; i++) { for (int j = 0; j < n; j++) { matrix[i][j] = min(matrix[i][j], matrix[i][k] + matrix[k][j]); } } } for (int i = 0; i < n; i++) { for (int j = 0; j < n; j++) { if (matrix[i][j] == 1e9) { matrix[i][j] = -1; } } } }; int main() { int V = 4; vector<vector<int>> matrix(V, vector<int>(V, -1)); matrix[0][1] = 2; matrix[1][0] = 1; matrix[1][2] = 3; matrix[3][0] = 3; matrix[3][1] = 5; matrix[3][2] = 4; Solution obj; obj.shortest_distance(matrix); for (auto row : matrix) { for (auto cell : row) { cout << cell << " "; } cout << endl; } return 0; } 	You are given a directed weighted graph in the form of an adjacency matrix . You are using the Floyd-Warshall algorithm to compute shortest distances between every pair of vertices .																								
	★ Input Matrix (after setup)																								
	The initial matrix setup (after setting the given edges): <table style="margin-left: auto; margin-right: auto;"> <tr><td>0</td><td>1</td><td>2</td><td>3</td></tr> <tr><td>0</td><td>-1</td><td>2</td><td>-1</td><td>-1</td></tr> <tr><td>1</td><td>1</td><td>-1</td><td>3</td><td>-1</td></tr> <tr><td>2</td><td>-1</td><td>-1</td><td>-1</td><td>-1</td></tr> <tr><td>3</td><td>3</td><td>5</td><td>4</td><td>-1</td></tr> </table>	0	1	2	3	0	-1	2	-1	-1	1	1	-1	3	-1	2	-1	-1	-1	-1	3	3	5	4	-1
0	1	2	3																						
0	-1	2	-1	-1																					
1	1	-1	3	-1																					
2	-1	-1	-1	-1																					
3	3	5	4	-1																					
	Converted to: <table style="margin-left: auto; margin-right: auto;"> <tr><td>0</td><td>1</td><td>2</td><td>3</td></tr> <tr><td>0</td><td>0</td><td>2</td><td>1e9</td><td>1e9</td></tr> <tr><td>1</td><td>1</td><td>0</td><td>3</td><td>1e9</td></tr> <tr><td>2</td><td>1e9</td><td>1e9</td><td>0</td><td>1e9</td></tr> <tr><td>3</td><td>3</td><td>5</td><td>4</td><td>0</td></tr> </table>	0	1	2	3	0	0	2	1e9	1e9	1	1	0	3	1e9	2	1e9	1e9	0	1e9	3	3	5	4	0
0	1	2	3																						
0	0	2	1e9	1e9																					
1	1	0	3	1e9																					
2	1e9	1e9	0	1e9																					
3	3	5	4	0																					
	🧠 Floyd-Warshall Algorithm Dry Run																								
	We'll now go through each intermediate node k and update the matrix.																								
	⌚ For $k = 0$																								
	Try to go $i \rightarrow 0 \rightarrow j$ No new updates help here, as 0 is only connected to 1.																								
	⌚ For $k = 1$																								
	Try $i \rightarrow 1 \rightarrow j$: <ul style="list-style-type: none"> • $0 \rightarrow 1 \rightarrow 2 = 2 + 3 = 5 \rightarrow$ Update matrix[0][2] from $1e9 \rightarrow 5$ • $3 \rightarrow 1 \rightarrow 2 = 5 + 3 = 8 \rightarrow$ Update matrix[3][2] from $4 \rightarrow 4$ (already smaller, no change) 																								

🔗 For k = 2

Only relevant updates:

- $3 \rightarrow 2 \rightarrow 0 = 4 + 1e9 \rightarrow$ no update
- Nothing meaningful added as 2 is a disconnected node

🔗 For k = 3

- $0 \rightarrow 3 \rightarrow 0 \rightarrow$ Not reachable
- But let's try:
 - $0 \rightarrow 3 \rightarrow 2: \text{matrix}[0][3] + \text{matrix}[3][2] = 1e9 + 4 = 1e9 \rightarrow$ No update
 - Same for others, no improvement.

✓ Final Matrix (replace 1e9 with -1)

```
0 2 5 -1
1 0 3 -1
-1 -1 0 -1
3 5 4 0
```

▣ Output

```
0 2 5 -1
1 0 3 -1
-1 -1 0 -1
3 5 4 0
```

Output:-

```
0 2 5 -1
1 0 3 -1
-1 -1 0 -1
3 5 4 0
```

Check graph is bipartite using Breadth First Search in C++

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

class Solution {
    // colors a component
    private:
        bool check(int start, int V, vector<int>adj[], int color[]) {
            queue<int> q;
            q.push(start);
            color[start] = 0;
            while(!q.empty()) {
                int node = q.front();
                q.pop();

                for(auto it : adj[node]) {
                    // if the adjacent node is yet not colored
                    // you will give the opposite color of the
node
                    if(color[it] == -1) {
                        color[it] = !color[node];
                        q.push(it);
                    }
                    // is the adjacent guy having the same
color
                    else if(color[it] == color[node]) {
                        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 not coloured
            if(color[i] == -1) {
                if(check(i, V, adj, color) == false) {
                    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(){
        // V = 4, E = 4
        vector<int>adj[4];
    }
}
```

Graph Structure

Vertices: $V = 4$

Edges:

- $0 \leftrightarrow 2$
- $0 \leftrightarrow 3$
- $2 \leftrightarrow 3$
- $3 \leftrightarrow 1$

Adjacency List:

0: [2, 3]

1: [3]

2: [0, 3]

3: [0, 2, 1]

Dry Run of check() Function (BFS for Coloring)

We want to color the graph with **2 colors (0 and 1)** such that no two adjacent nodes have the same color.

Step	Node	Queue	Color Status	Action
1	0	[0]	[$-1, -1, -1, -1$]	Start BFS with node 0 → $\text{color}[0] = 0$
2	0	[2, 3]	[0, $-1, 1, 1$]	2 & 3 uncolored → assign opposite color
3	2	[3]	[0, $-1, 1, 1$]	0 already colored & valid → continue
4	2	[3]	[0, $-1, 1, 1$]	3 already colored with same color → 
				Conflict found → graph is not bipartite

Output:

0

```
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;
}
```

Output:-

0

Cycle detection in undirected graph using Breadth First Search in C++

```
#include <bits/stdc++.h>
using namespace std;
class Solution {
public:
    // Function to detect cycle in a
    // directed graph.
    bool isCyclic(int V, vector<int>
adj[]) {
        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);
            }
        }
        int cnt = 0;
        // o(v + e)
        while (!q.empty()) {
            int node = q.front();
            q.pop();
            cnt++;
            // node is in your topo sort
            // so please remove it from
            // the indegree
            for (auto it : adj[node]) {
                indegree[it]--;
                if (indegree[it] == 0)
                    q.push(it);
            }
        }
        if (cnt == V) return false;
        return true;
    }
};

int main() {
    //V = 6;
    vector<int> adj[6] = {{}, {2}, {3}, {4, 5}, {2}, {}};
    int V = 6;
    Solution obj;
    bool ans = obj.isCyclic(V, adj);
    if (ans) cout << "True";
    else cout << "False";
    cout << endl;
    return 0;
}
```

Graph Details

From your adj array:

```
vector<int> adj[6] = {
    {},      // 0
    {2},     // 1 → 2
    {3},     // 2 → 3
    {4, 5},  // 3 → 4, 5
    {2},     // 4 → 2 ← Cycle!
    {}       // 5
};
```

► Number of vertices: $V = 6$

► Step 1: Calculate In-Degrees

Node	Incoming Edges	in-degree
0	—	0
1	—	0
2	from 1, 4	2
3	from 2	1
4	from 3	1
5	from 3	1

► Initial in-degree array: [0, 0, 2, 1, 1, 1]

► Step 2: Initialize Queue with in-degree = 0

$q = [0, 1]$ // because $\text{indegree}[0] = 0$ and $\text{indegree}[1] = 0$

► Step 3: BFS Traversal & Count Nodes Processed

Iteration	Queue	Node Popped	Neighbors	Action	Updated in-degree	Count
1	[0, 1]	0	—	No neighbors	[0, 0, 2, 1, 1, 1]	1
2	[1]	1	[2]	$\text{indegree}[2] = 2 \rightarrow 1$ (not zero yet)	[0, 0, 1, 1, 1, 1]	2
3	[]	—	—	Queue is empty — loop ends		2

► Step 4: Final Check

- Nodes processed ($\text{cnt} = 2$)
- Total nodes ($V = 6$)

❖ Since $\text{cnt} \neq V$, there **is a cycle** in the graph.

Output:-

True

The graph contains a cycle

Cycle detection in undirected graph using Breadth First Search in C++

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

class Solution {
private:
    bool detect(int src, vector<int> adj[], int vis[]) {
        vis[src] = 1;
        // store <source node, parent node>
        queue<pair<int,int>> q;
        q.push({src, -1});
        // traverse until queue is not empty
        while(!q.empty()) {
            int node = q.front().first;
            int parent = q.front().second;
            q.pop();

            // go to all adjacent nodes
            for(auto adjacentNode: adj[node]) {
                // if adjacent node is unvisited
                if(!vis[adjacentNode]) {
                    vis[adjacentNode] = 1;
                    q.push({adjacentNode, node});
                }
                // if adjacent node is visited and is not
                // its own parent node
                else if(parent != adjacentNode) {
                    // yes it is a cycle
                    return true;
                }
            }
            // there's no cycle
            return false;
        }
        public:
            // Function to detect cycle in an undirected
            graph.
            bool isCycle(int V, vector<int> adj[]) {
                // initialise them as unvisited
                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() {
    // V = 4, E = 2
    vector<int> adj[4] = {{}, {2}, {1, 3}, {2}};
    Solution obj;
    bool ans = obj.isCycle(4, adj);
    if (ans)
        cout << "1\n";
    else
        cout << "0\n";
    return 0;
}
```

Graph Definition (Adjacency List)

```
vector<int> adj[4] = {
    {},      // 0 → No neighbors
    {2},     // 1 → 2
    {1, 3},  // 2 → 1, 3
    {2}      // 3 → 2
};
```

Visual graph:

1 -- 2 -- 3

- It's a **linear graph**, no cycle expected.

🧠 Variables

- vis[4] = {0, 0, 0, 0} (all unvisited initially)
- Queue for BFS: stores pairs {node, parent}

⌚ Step-by-Step Traversal Table

Iter	Queue	node	parent	Neighbours	Action
1	{1, -1}	1	-1	[2]	2 is unvisited → mark visited, enqueue {2, 1}
2	{2, 1}	2	1	[1, 3]	1 is parent → skip; 3 is unvisited → mark visited, enqueue {3, 2}
3	{3, 2}	3	2	[2]	2 is parent → skip
4	empty	—	—	—	Loop ends

Visited array after traversal: [0, 1, 1, 1]

No condition parent != adjacentNode && vis[adjacentNode] == 1 was met.

⚡ Final Output

0 // No cycle found

📋 Summary Table

Node	Parent	Visited	Notes
1	-1	✓	Starting node
2	1	✓	Connected from node 1

Node	Parent	Visited	Notes
3	2	✓	Connected from node 2
0	-	✗	Isolated node (not connected)

 **Conclusion**

- **No cycle** detected — the output is 0.

Output:-
0
No cycle was found in any component of the graph

Breadth First Search in C++

```
#include <iostream>
#include <vector>
#include <queue>
#include <deque>
using namespace std;
// Function to add an edge between two
vertices u and v
void addEdge(vector<vector<int>>& adj, int u,
int v) {
    adj[u].push_back(v);
    adj[v].push_back(u);
}
// Function to perform BFS traversal
void bfs(vector<vector<int>>& adj, int v, int s)
{
    deque<int> q;
    vector<bool> visited(v, false);
    q.push_back(s);
    visited[s] = true;
    while (!q.empty()) {
        int rem = q.front();
        q.pop_front();
        cout << rem << " ";
        for (int nbr : adj[rem]) {
            if (!visited[nbr]) {
                visited[nbr] = true;
                q.push_back(nbr);
            }
        }
    }
    cout << endl; // Print newline after traversal
}
int main() {
    int V = 7;
    vector<vector<int>> adj(V);
    // Adding edges to the graph
    addEdge(adj, 0, 1);
    addEdge(adj, 0, 2);
    addEdge(adj, 2, 3);
    addEdge(adj, 1, 3);
    addEdge(adj, 1, 4);
    addEdge(adj, 3, 4);
    cout << "Following is Breadth First
Traversal: \n";
    bfs(adj, V, 0);
    return 0;
}
```

Graph Structure

Adjacency List:

```
0: [1, 2]
1: [0, 3, 4]
2: [0, 3]
3: [2, 1, 4]
4: [1, 3]
5: []
6: []
```

(Nodes 5 and 6 are isolated)

🧠 BFS Dry Run Table

Step	Queue	Visited Nodes	Node Processed	Neighbors Added	Output
1	[0]	{}	-	-	
2	[1, 2]	{0}	0	1, 2	0
3	[2, 3, 4]	{0, 1}	1	3, 4 (0 already done)	0 1
4	[3, 4]	{0, 1, 2}	2	- (0, 3 already done)	0 1 2
5	[4]	{0,1,2,3}	3	- (2,1,4 already done)	0 1 2 3
6	[]	{0,1,2,3,4}	4	- (1,3 already done)	0 1 2 3 4

📋 Final Output

Following is Breadth First Traversal:
0 1 2 3 4

Output:-

0 1 2 3 4

Cycle detection in undirected graph using Depth First Search in C++

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

class Solution {
private:
    bool dfs(int node, int parent, int vis[], vector<int> adj[]) {
        vis[node] = 1;
        // visit adjacent nodes
        for(auto adjacentNode: adj[node]) {
            // unvisited adjacent node
            if(!vis[adjacentNode]) {
                if(dfs(adjacentNode, node, vis, adj) == true)
                    return true;
            }
            // visited node but not a parent node
            else if(adjacentNode != parent) return true;
        }
        return false;
    }
public:
    // Function to detect cycle in an undirected graph.
    bool isCycle(int V, vector<int> adj[]) {
        int vis[V] = {0};
        // for graph with connected components
        for(int i = 0;i<V;i++) {
            if(!vis[i]) {
                if(dfs(i, -1, vis, adj) == true) return true;
            }
        }
        return false;
    }
};

int main() {

    // V = 4, E = 2
    vector<int> adj[4] = {{}, {2}, {1, 3}, {2}};
    Solution obj;
    bool ans = obj.isCycle(4, adj);
    if (ans)
        cout << "1\n";
    else
        cout << "0\n";
    return 0;
}
```

Input Graph (Adjacency List)

```
vector<int> adj[4] = {
    {},           // 0 → no connections
    {2},          // 1 → connected to 2
    {1, 3},       // 2 → connected to 1 and 3
    {2}           // 3 → connected to 2
};
```

Graph in visual form:

1 -- 2 -- 3

(0 is isolated and not connected to any node.)

🧠 DFS Function Signature

```
bool dfs(int node, int parent, int vis[],  
vector<int> adj[]);
```

- node: current node being explored
- parent: node from which we came
- vis[]: visited array
- adj[]: adjacency list

💻 Dry Run Table

Initial:

- vis[4] = {0, 0, 0, 0}

DFS Call Stack Trace

Call	Node	Parent	Visited Array	Action
1	0	-1	[1, 0, 0, 0]	No neighbors → return false
2	1	-1	[1, 1, 0, 0]	Visit 2 from 1
3	2	1	[1, 1, 1, 0]	1 is parent → skip; visit 3
4	3	2	[1, 1, 1, 1]	2 is parent → skip; DFS returns false
3↑	2	1	[1, 1, 1, 1]	DFS from 3 returned false → continue → DFS returns false
2↑	1	-1	[1, 1, 1, 1]	DFS from 2 returned false → continue → DFS

				returns false
<p>✓ Final State</p> <ul style="list-style-type: none">• All nodes visited: vis = [1, 1, 1, 1]• No back-edge found (no adjacent visited node that's not the parent)				
<p>Output:</p> <p>0</p>				
<p>Output:-</p> <p>0</p> <p>No cycle</p>				

Depth First Search in C++

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

class Solution {
public:
    // Function to return Breadth First Traversal of
    // given graph.
    vector<int> bfsOfGraph(int V, vector<int> adj[]) {
        int vis[V] = {0};
        vis[0] = 1;
        queue<int> q;
        // push the initial starting node
        q.push(0);
        vector<int> bfs;
        // iterate till the queue is empty
        while(!q.empty()) {
            // get the topmost element in the queue
            int node = q.front();
            q.pop();
            bfs.push_back(node);
            // traverse for all its neighbours
            for(auto it : adj[node]) {
                // if the neighbour has previously not been
                // visited,
                // store in Q and mark as visited
                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;
}
```

Graph Definition (Adjacency List)

```
vector<int> adj[6];
addEdge(adj, 0, 1);
addEdge(adj, 1, 2);
addEdge(adj, 1, 3);
addEdge(adj, 0, 4);
```

Adjacency List:

```
0 → [1, 4]
1 → [0, 2, 3]
2 → [1]
3 → [1]
4 → [0]
```

BFS Variables

- $\text{vis}[5] = \{1, 0, 0, 0, 0\} \rightarrow$ Only node 0 marked visited initially
- Queue: $q = [0]$
- Result vector: $bfs = []$

BFS Traversal Table

Step	Queue	Node Popped	BFS List	Neighbors	Action
1	[0]	0	[0]	[1, 4]	Visit 1 & 4 → mark visited, enqueue → Queue: [1, 4]
2	[1, 4]	1	[0, 1]	[0, 2, 3]	0 already visited; Visit 2 & 3 → mark visited, enqueue → Queue: [4, 2, 3]
3	[4, 2, 3]	4	[0, 1, 4]	[0]	0 already visited → nothing added
4	[2, 3]	2	[0, 1, 4, 2]	[1]	1 already visited
5	[3]	3	[0, 1, 4, 2, 3]	[1]	1 already visited
6	[]	-	Done	-	Queue

						empty → BFS complete																								
❖ Final BFS Output																														
[0, 1, 4, 2, 3]																														
🧠 Summary Table																														
<table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="text-align: left; padding: 2px;">Node</th> <th style="text-align: left; padding: 2px;">Visited</th> <th style="text-align: left; padding: 2px;">Enqueued</th> <th style="text-align: left; padding: 2px;">When</th> </tr> </thead> <tbody> <tr> <td style="text-align: left; padding: 2px;">0</td> <td style="text-align: left; padding: 2px;">✓</td> <td style="text-align: left; padding: 2px;">✓</td> <td style="text-align: left; padding: 2px;">Start</td> </tr> <tr> <td style="text-align: left; padding: 2px;">1</td> <td style="text-align: left; padding: 2px;">✓</td> <td style="text-align: left; padding: 2px;">✓</td> <td style="text-align: left; padding: 2px;">From 0</td> </tr> <tr> <td style="text-align: left; padding: 2px;">4</td> <td style="text-align: left; padding: 2px;">✓</td> <td style="text-align: left; padding: 2px;">✓</td> <td style="text-align: left; padding: 2px;">From 0</td> </tr> <tr> <td style="text-align: left; padding: 2px;">2</td> <td style="text-align: left; padding: 2px;">✓</td> <td style="text-align: left; padding: 2px;">✓</td> <td style="text-align: left; padding: 2px;">From 1</td> </tr> <tr> <td style="text-align: left; padding: 2px;">3</td> <td style="text-align: left; padding: 2px;">✓</td> <td style="text-align: left; padding: 2px;">✓</td> <td style="text-align: left; padding: 2px;">From 1</td> </tr> </tbody> </table>							Node	Visited	Enqueued	When	0	✓	✓	Start	1	✓	✓	From 0	4	✓	✓	From 0	2	✓	✓	From 1	3	✓	✓	From 1
Node	Visited	Enqueued	When																											
0	✓	✓	Start																											
1	✓	✓	From 0																											
4	✓	✓	From 0																											
2	✓	✓	From 1																											
3	✓	✓	From 1																											
★ Output on Console:																														
0 1 4 2 3																														

Output:-
0 1 4 2 3

Kahn in C++

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

class Solution {
public:
    //Function to return list containing vertices in
    Topological order.
    vector<int> topoSort(int V, vector<int> adj[])
    {
        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);
            // node is in your topo sort
            // so please remove it from the indegree
            for (auto it : adj[node]) {
                indegree[it]--;
                if (indegree[it] == 0) q.push(it);
            }
        }

        return topo;
    }

    int main() {
        //V = 6;
        vector<int> adj[6] = {{}, {}, {3}, {1}, {0, 1}, {0, 2}};
        int V = 6;
        Solution obj;
        vector<int> ans = obj.topoSort(V, adj);

        for (auto node : ans) {
            cout << node << " ";
        }
        cout << endl;

        return 0;
    }
}
```

Input Graph (Adjacency List)

```
vector<int> adj[6] = {
    {},      // 0
    {},      // 1
    {3},     // 2 → 3
    {1},     // 3 → 1
    {0, 1},  // 4 → 0, 1
    {0, 2}   // 5 → 0, 2
};
```

Step 1: Calculate In-Degree of Each Node

Node	Incoming Edges from	In-degree
0	4, 5	2
1	3, 4	2
2	5	1
3	2	1
4	-	0
5	-	0

→ Initial indegree[] = {2, 2, 1, 1, 0, 0}

Step 2: Enqueue All Nodes With In-degree = 0

Initial Queue: q = [4, 5]

Step 3: BFS Loop & Topological Sorting

Iteration	Node Popped	Topo List	Decrease In-degree	Queue after Push
1	4	[4]	0→1, 1→1	[5]
2	5	[4, 5]	0→0 ✓, 2→0 ✓	[0, 2]
3	0	[4, 5, 0]	-	[2]
4	2	[4, 5, 0, 2]	3→0 ✓	[3]
5	3	[4, 5, 0, 2,	1→0 ✓	[1]

Iteration	Node Popped	Topo List	Decrease In-degree	Queue after Push
		3]		
6	1	[4, 5, 0, 2, 3, 1]	-	[] (done)

↙ Final Output

Topological Order = [4, 5, 0, 2, 3, 1]

🧠 Summary Table

Node	Final In-degree	Status
0	0	Printed
1	0	Printed
2	0	Printed
3	0	Printed
4	0	Printed
5	0	Printed

Output:-
4 5 0 2 3 1

Kruskal in C++

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

class DisjointSet {
    vector<int> rank, parent, size;
public:
    DisjointSet(int n) {
        rank.resize(n + 1, 0);
        parent.resize(n + 1);
        size.resize(n + 1);
        for (int i = 0; i <= n; i++) {
            parent[i] = i;
            size[i] = 1;
        }
    }

    int findUPar(int node) {
        if (node == parent[node])
            return node;
        return parent[node] =
findUPar(parent[node]);
    }

    void unionByRank(int u, int v) {
        int ulp_u = findUPar(u);
        int ulp_v = findUPar(v);
        if (ulp_u == ulp_v) return;
        if (rank[ulp_u] < rank[ulp_v]) {
            parent[ulp_u] = ulp_v;
        }
        else if (rank[ulp_v] < rank[ulp_u]) {
            parent[ulp_v] = ulp_u;
        }
        else {
            parent[ulp_v] = ulp_u;
            rank[ulp_u]++;
        }
    }

    void unionBySize(int u, int v) {
        int ulp_u = findUPar(u);
        int ulp_v = findUPar(v);
        if (ulp_u == ulp_v) return;
        if (size[ulp_u] < size[ulp_v]) {
            parent[ulp_u] = ulp_v;
            size[ulp_v] += size[ulp_u];
        }
        else {
            parent[ulp_v] = ulp_u;
            size[ulp_u] += size[ulp_v];
        }
    }
};

class Solution
{
public:
    //Function to find sum of weights of edges
    //of the Minimum Spanning Tree.
    int spanningTree(int V,
vector<vector<int>> adj[])
{
```

Input

You are given:

```
V = 5;
edges = {
    {0, 1, 2},
    {0, 2, 1},
    {1, 2, 1},
    {2, 3, 2},
    {3, 4, 1},
    {4, 2, 2}
};
```

Step 1: Adjacency List Construction (Undirected Graph)

adj[i] stores {neighbour, weight}:

Node	Adjacents
0	[1, 2], [2, 1]
1	[0, 2], [2, 1]
2	[0, 1], [1, 1], [3, 2], [4, 2]
3	[2, 2], [4, 1]
4	[3, 1], [2, 2]

Step 2: Edge List Formation

Collected as {weight, {u, v}} (both directions included):

Edge	Format
0-1	{2, {0, 1}}
0-2	{1, {0, 2}}
1-2	{1, {1, 2}}
2-3	{2, {2, 3}}
3-4	{1, {3, 4}}
4-2	{2, {4, 2}}
 duplicates (undirected, so reverse edges too!)	

▼ Step 3: Sort Edges by Weight

Sorted edges:

```
edges = {
    {1, {0, 2}},
    {1, {1, 2}},
    {1, {3, 4}},
    {2, {0, 1}},
    {2, {2, 3}},
    {2, {4, 2}}
}
```

```

{
    // 1 - 2 wt = 5
    /// 1 - > (2, 5)
    // 2 -> (1, 5)

    // 5, 1, 2
    // 5, 2, 1
    vector<pair<int, pair<int, int>>> edges;
    for (int i = 0; i < V; i++) {
        for (auto it : adj[i]) {
            int adjNode = it[0];
            int wt = it[1];
            int node = i;

            edges.push_back({wt, {node,
adjNode}});
        }
    }
    DisjointSet ds(V);
    sort(edges.begin(), edges.end());
    int mstWt = 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)) {
            mstWt += wt;
            ds.unionBySize(u, v);
        }
    }

    return mstWt;
}

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) {
        vector<int> tmp(2);
        tmp[0] = it[1];
        tmp[1] = it[2];
        adj[it[0]].push_back(tmp);

        tmp[0] = it[0];
        tmp[1] = it[2];
        adj[it[1]].push_back(tmp);
    }

    Solution obj;
    int mstWt = obj.spanningTree(V, adj);
    cout << "The sum of all the edge weights: "
    << mstWt << endl;
    return 0;
}

```

Output:-

The sum of all the edge weights: 5

❖ Step 4: Disjoint Set Initialization

- Each node starts as its own parent.
- parent[] = {0, 1, 2, 3, 4}
- size[] = {1, 1, 1, 1, 1}

⌚ Step 5: Process Edges

Edge	Find UParent(u)	Find UParent(v)	Cycle?	Union?	MST Weight
{1, {0, 2}}	0	2	No	Union(0, 2)	1
{1, {1, 2}}	1	0 (from 2)	No	Union(1, 0)	2
{1, {3, 4}}	3	4	No	Union(3, 4)	3
{2, {0, 1}}	0	0	Yes	✗ Skip	3
{2, {2, 3}}	0	3	No	Union(0, 3)	5
{2, {4, 2}}	0	0	Yes	✗ Skip	5

❖ Final MST Weight

The sum of all the edge weights: 5

🧠 Disjoint Set Status (Final)

Node	Parent
0	0
1	0
2	0
3	0
4	0

All nodes are connected — ✅ valid spanning tree.

◆ DFS(0)

node	vis[node]	Neighbors	Action	vis
0	0 → 1	2	DFS(2)	[1, 0, 0]
2	0 → 1	0	Already vis	[1, 0, 1]

◆ DFS(1)

node	vis[node]	Neighbors	Action	vis
1	0 → 1	none	Done	[1, 1, 1]

Final Result

Variable	Value
cnt	2 (Answer)
vis	[1, 1, 1]

Output: 2 provinces

Output:-
2

Prim in C++

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

class Solution
{
public:
    //Function to find sum of weights of edges of the
    Minimum Spanning Tree.
    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);
        // {wt, node}
        pq.push({0, 0});
        int sum = 0;
        while (!pq.empty()) {
            auto it = pq.top();
            pq.pop();
            int node = it.second;
            int wt = it.first;

            if (vis[node] == 1) continue;
            // add it to the mst
            vis[node] = 1;
            sum += wt;
            for (auto it : adj[node]) {
                int adjNode = it[0];
                int edW = it[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) {
            vector<int> tmp(2);
            tmp[0] = it[1];
            tmp[1] = it[2];
            adj[it[0]].push_back(tmp);

            tmp[0] = it[0];
            tmp[1] = it[2];
            adj[it[1]].push_back(tmp);
        }
        Solution obj;
```

Input Edges

```
edges = {
    {0, 1, 2},
    {0, 2, 1},
    {1, 2, 1},
    {2, 3, 2},
    {3, 4, 1},
    {4, 2, 2}
}
```

Adjacency List

Node	Neighbors
0	[1,2], [2,1]
1	[0,2], [2,1]
2	[0,1], [1,1], [3,2], [4,2]
3	[2,2], [4,1]
4	[3,1], [2,2]

Prim's MST Logic (Min-Heap)

We track:

- pq: min-heap for {weight, node}
- vis[]: visited array
- sum: total MST weight

Dry Run Table

Step	pq (Min- Heap)	node	wt	vis	sum	Action Taken
1	{(0, 0)}	0	0	[1, 0, 0, 0, 0]	0	Add node 0, add neighbors 1 (wt=2), 2 (wt=1) to pq
2	{(1, 2), (2, 1)}	2	1	[1, 0, 1, 0, 0]	1	Add node 2, add unvisited neighbors: 1(wt=1), 3(wt=2), 4(wt=2)
3	{(1, 1), (2, 1), (2, 3), (2, 4)}	1	1	[1, 1, 1, 0, 0]	2	Add node 1, skip already visited 0 & 2
4	{(2, 1), (2, 3), (2, 4)}	1	2	Already visited	-	Skip

```

int sum = obj.spanningTree(V, adj);
cout << "The sum of all the edge weights: " <<
sum << endl;

return 0;
}

```

Step	pq (Min- Heap)	node	wt	vis	sum	Action Taken
5	{(2, 3), (2, 4)}	3	2	[1, 1, 1, 1, 0]	4	Add node 3, add neighbor 4 (wt=1)
6	{(1, 4), (2, 4)}	4	1	[1, 1, 1, 1, 1]	5	Add node 4, skip visited 3, 2
7	{(2, 4)}	4	2	Already visited	-	Skip

❖ Final Result:

Variable	Value
sum	5
vis	[1,1,1,1,1] (All visited)

❖ Output:

The sum of all the edge weights: 5

Output:-

The sum of all the edge weights: 5

Reverse directed graph in C++

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

class ReverseDirectedGraph {
public:
    static vector<vector<int>>
reverseDirectedGraph(const vector<vector<int>>& adj,
int V) {
    vector<vector<int>> reversedAdj(V + 1);

    for (int i = 0; i <= V; ++i) {
        for (int j : adj[i]) {
            reversedAdj[j].push_back(i);
        }
    }

    return reversedAdj;
}

static void printGraph(const vector<vector<int>>&
graph, int V) {
    for (int i = 1; i <= V; ++i) {
        for (int j : graph[i]) {
            cout << i << " -> " << j << endl;
        }
    }
};

int main() {
    int V = 5;
    vector<vector<int>> adj(V + 1);

    adj[1].push_back(3);
    adj[1].push_back(2);
    adj[3].push_back(4);
    adj[4].push_back(5);

    vector<vector<int>> reversedAdj =
    ReverseDirectedGraph::reverseDirectedGraph(adj, V);

    cout << "Reversed Graph:" << endl;
    ReverseDirectedGraph::printGraph(reversedAdj, V);

    return 0;
}
```

Original Input Graph (Adjacency List)

We have a **directed graph** with 5 vertices ($V = 5$):

Vertex	Edges
1	$\rightarrow 3, \rightarrow 2$
2	—
3	$\rightarrow 4$
4	$\rightarrow 5$
5	—

Graphically:

$1 \rightarrow 2$
 \downarrow
 $3 \rightarrow 4 \rightarrow 5$

↗ Dry Run Table: reverseDirectedGraph(adj, V)

This function creates a reversed adjacency list where **every edge $u \rightarrow v$ becomes $v \rightarrow u$** .

i (Source Node)	j (adj[i])	reversedAdj[j] After Insertion
1	3	reversedAdj[3] = {1}
1	2	reversedAdj[2] = {1}
3	4	reversedAdj[4] = {3}
4	5	reversedAdj[5] = {4}

↖ Final reversedAdj Table

Vertex	reversedAdj[vertex] (Incoming Edges)
1	—
2	1
3	1
4	3
5	4

↗ Output of printGraph(reversedAdj, V)

This prints **destination → source** (reversed):

	2 -> 1 3 -> 1 4 -> 3 5 -> 4
--	--------------------------------------

Output:-

Reversed Graph:

2 -> 1
3 -> 1
4 -> 3
5 -> 4

Rotten Oranges in C++

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

using namespace std;

class Solution {
public:
    //Function to find minimum time required to rot all
    //oranges.
    int orangesRotting(vector < vector < int >> & grid) {
        // figure out the grid size
        int n = grid.size();
        int m = grid[0].size();

        // store {{row, column}, time}
        queue < pair < pair < int, int >, int >> q;
        int vis[n][m];
        int cntFresh = 0;
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < m; j++) {
                // if cell contains rotten orange
                if (grid[i][j] == 2) {
                    q.push({{i, j}, 0});
                    // mark as visited (rotten) in visited array
                    vis[i][j] = 2;
                }
                // if not rotten
                else {
                    vis[i][j] = 0;
                }
                // count fresh oranges
                if (grid[i][j] == 1) cntFresh++;
            }
        }

        int tm = 0;
        // delta row and delta column
        int drow[] = {-1, 0, +1, 0};
        int dcol[] = {0, 1, 0, -1};
        int cnt = 0;

        // bfs traversal (until the queue becomes empty)
        while (!q.empty()) {
            int r = q.front().first;
            int c = q.front().second;
            int t = q.front().second;
            tm = max(tm, t);
            q.pop();
            // exactly 4 neighbours
            for (int i = 0; i < 4; i++) {
                // neighbouring row and column
                int nrow = r + drow[i];
                int ncol = c + dcol[i];
                // check for valid cell and
                // then for unvisited fresh orange
                if (nrow >= 0 && nrow < n && ncol >= 0 && ncol <
m &&
                    vis[nrow][ncol] == 0 && grid[nrow][ncol] == 1) {
                    // push in queue with timer increased
                    q.push({{nrow, ncol}, t + 1});
                    // mark as rotten
                    vis[nrow][ncol] = 2;
                }
            }
        }
    }
}
```

Input Grid
grid = {
{0, 1, 2},
{0, 1, 2},
{2, 1, 1}
};

↙ Initial Setup

- Fresh oranges = 4
- Rotten oranges start at:
 - (0, 2)
 - (1, 2)
 - (2, 0)
- Queue initialized with these rotten oranges (time = 0)

█ Dry Run Table

Time	Queue Front (Cell)	Rotting New Oranges → Queue Update	Total Rotten
0	(0, 2)	(0,1) → push with t=1	1
0	(1, 2)	(1,1) → push with t=1	2
0	(2, 0)	(2,1) → push with t=1	3
1	(0, 1)	— (no new fresh)	—
1	(1, 1)	— (no new fresh)	—
1	(2, 1)	(2,2) → push with t=2	4
2	(2, 2)	—	—

█ Final Check

- Rotten count = 4
- Fresh count = 4
- ↙ All fresh oranges became rotten
- Max time = 2 (last t value added to queue)

↙ Final Output

Answer = 2

```
        cnt++;
    }
}

// if all oranges are not rotten
if (cnt != cntFresh) return -1;

return tm;
};

int main() {

vector<vector<int>>grid{{0,1,2},{0,1,2},{2,1,1}};
Solution obj;
int ans = obj.orangesRotting(grid);
cout << ans << "\n";

return 0;
}
```

Output:-

1

Terminal Nodes in C++					
Step-by-Step Dry Run					
Step	Operation	Affected Node(s)	Adjacency List State	Notes	
1	addEdge(1, 2)	1, 2	{1: [2], 2: []}	1 → 2, ensure 2 is in the map	
2	addEdge(2, 3)	2, 3	{1: [2], 2: [3], 3: []}	2 → 3, ensure 3 is in the map	
3	addEdge(3, 4)	3, 4	{1: [2], 2: [3], 3: [4], 4: []}	3 → 4, ensure 4 is in the map	
4	addEdge(4, 5)	4, 5	{1: [2], 2: [3], 3: [4], 4: [5], 5: []}	4 → 5, ensure 5 is in the map	
5	addEdge(6, 7)	6, 7	{1: [2], 2: [3], 3: [4], 4: [5], 5: [], 6: [7], 7: []}	6 → 7, ensure 7 is in the map	
6	printTerminalNodes()	Scan all nodes		Check which nodes have empty adjacency lists	Nodes 5 and 7 have no outgoing edges
7	Print	Terminal Nodes			Output: 5, 7
↙ Final Output					
Terminal Nodes:					
5 7					
Output:-					
Terminal Nodes: 7 5					

Topological sort DFS in C++

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

class Topo_dfs {
public:
    // Helper function to perform DFS and populate stack
    static void dfs(int node, vector<int>& vis, stack<int>& st, vector<vector<int>>& adj) {
        vis[node] = 1; // Mark node as visited

        // Traverse all adjacent nodes
        for (int it : adj[node]) {
            if (vis[it] == 0) { // If adjacent node is not visited,
                perform DFS on it
                    dfs(it, vis, st, adj);
            }
        }

        st.push(node); // Push current node to stack after
visiting all its dependencies
    }

    // Function to perform topological sorting using DFS
    static vector<int> topoSort(int V,
vector<vector<int>>& adj) {
        vector<int> vis(V, 0); // Initialize visited array
        stack<int> st; // Stack to store nodes in topological
order

        // Perform DFS for each unvisited node
        for (int i = 0; i < V; ++i) {
            if (vis[i] == 0) {
                dfs(i, vis, st, adj);
            }
        }

        vector<int> topo(V);
        int index = 0;

        // Pop elements from stack to get topological order
        while (!st.empty()) {
            topo[index++] = st.top();
            st.pop();
        }

        return topo;
    }
};

int main() {
    int V = 6;
    vector<vector<int>> adj(V);

    adj[2].push_back(3);
    adj[3].push_back(1);
    adj[4].push_back(0);
    adj[4].push_back(1);
    adj[5].push_back(0);
    adj[5].push_back(2);
}
```

Revised Dry Run with DFS Call Order

DFS Start	Calls	Stack Push Order
0	No edges → push(0)	0
1	No edges → push(1)	1, 0
2	DFS(3) → DFS(1) already visited	3, 2, 1, 0
3	Already visited	
4	DFS(0, already visited), DFS(1)	4, 3, 2, 1, 0
5	DFS(0, 2) already visited	5, 4, 3, 2, 1, 0

✓ Stack (Top to Bottom)

5
4
2
3
1
0

→ Final Output

```
while (!st.empty()) {
    topo[index++] = st.top();
    st.pop();
}
```

■ Output:

5 4 2 3 1 0

🧠 Why This Is Valid:

Topological sort can have **multiple valid orders** as long as:

- For every edge $u \rightarrow v$, u appears **before** v .

And in this case:

- 5 is before 2, 0
- 2 is before 3
- 3 is before 1
- 4 is before 0, 1

✓ All conditions are satisfied.

```
vector<int> ans = Topo_dfs::topoSort(V, adj);

for (int node : ans) {
    cout << node << " ";
}
cout << endl;

return 0;
}
```

Output:-

```
5 4 2 3 1 0
```

0/1 KnapSack in C++

```

#include <iostream>
#include <vector>

using namespace std;

class ZeroOneKnapsack {
public:
    int knapsack(int n, vector<int>& vals,
vector<int>& wts, int cap) {
        vector<vector<int>> dp(n + 1, vector<int>(cap + 1, 0));

        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= cap; j++) {
                if (j >= wts[i - 1]) {
                    int remainingCap = j - wts[i - 1];

                    if (dp[i - 1][remainingCap] + vals[i - 1] >
dp[i - 1][j]) {
                        dp[i][j] = dp[i - 1][remainingCap] +
vals[i - 1];
                    } else {
                        dp[i][j] = dp[i - 1][j];
                    }
                } else {
                    dp[i][j] = dp[i - 1][j];
                }
            }
        }

        return dp[n][cap];
    }
};

int main() {
    ZeroOneKnapsack solution;

    // Input parameters
    int n = 5;
    vector<int> vals = {15, 14, 10, 45, 30};
    vector<int> wts = {2, 5, 1, 3, 4};
    int cap = 7;

    // Compute maximum value using knapsack
    // function
    int maxVal = solution.knapsack(n, vals, wts, cap);

    // Output the maximum value
    cout << "Maximum value that can be obtained: " <<
maxVal << endl;

    return 0;
}

```

Dry Run of the ZeroOneKnapsack Problem:

Input:

```
n = 5;  
vals = {15, 14, 10, 45, 30};  
wts = {2, 5, 1, 3, 4};  
cap = 7;
```

Step 1: Initialize the DP Table

We initialize a 2D DP table of size $(n + 1) \times (\text{cap} + 1)$ to store the maximum values obtainable for each subproblem. Each cell $\text{dp}[i][j]$ will represent the maximum value achievable with the first i items and a knapsack capacity of j .

Initially, the DP table is filled with zeros:

Step 2: Fill the DP Table

We iterate through each item ($i = 1$ to n) and each knapsack capacity ($j = 1$ to cap). The idea is to decide whether to include the current item or not.

Item 1 (Value = 15, Weight = 2)

- **Capacity 1:** $dp[1][1] = 0$ (Cannot include this item as the weight is greater than the capacity)
 - **Capacity 2:** $dp[1][2] = \max(dp[0][2], dp[0][0] + 15) = \max(0, 15) = 15$
 - **Capacity 3:** $dp[1][3] = \max(dp[0][3], dp[0][1] + 15) = \max(0, 15) = 15$
 - **Capacity 4:** $dp[1][4] = \max(dp[0][4], dp[0][2] + 15) = \max(0, 15) = 15$
 - **Capacity 5:** $dp[1][5] = \max(dp[0][5], dp[0][3] + 15) = \max(0, 15) = 15$
 - **Capacity 6:** $dp[1][6] = \max(dp[0][6], dp[0][4] + 15) = \max(0, 15) = 15$
 - **Capacity 7:** $dp[1][7] = \max(dp[0][7], dp[0][5] + 15) = \max(0, 15) = 15$

i\j	0	1	2	3	4	5	6	7
1	0	0	15	15	15	15	15	15
2	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0

Item 2 (Value = 14, Weight = 5)

- **Capacity 1 to 4:** The weight is greater than the capacity, so we can't include this item.
- **Capacity 5:** $dp[2][5] = \max(dp[1][5], dp[1][0] + 14) = \max(15, 14) = 15$
- **Capacity 6:** $dp[2][6] = \max(dp[1][6], dp[1][1] + 14) = \max(15, 14) = 15$
- **Capacity 7:** $dp[2][7] = \max(dp[1][7], dp[1][2] + 14) = \max(15, 29) = 29$

i\j	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	15	15	15	15	15	15
2	0	0	15	15	15	15	15	29
3	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0

Item 3 (Value = 10, Weight = 1)

- **Capacity 1:** $dp[3][1] = \max(dp[2][1], dp[2][0] + 10) = \max(0, 10) = 10$
- **Capacity 2:** $dp[3][2] = \max(dp[2][2], dp[2][1] + 10) = \max(15, 10) = 15$
- **Capacity 3:** $dp[3][3] = \max(dp[2][3], dp[2][2] + 10) = \max(15, 25) = 25$
- **Capacity 4:** $dp[3][4] = \max(dp[2][4], dp[2][3] + 10) = \max(15, 25) = 25$
- **Capacity 5:** $dp[3][5] = \max(dp[2][5], dp[2][4] + 10) = \max(15, 25) = 25$
- **Capacity 6:** $dp[3][6] = \max(dp[2][6], dp[2][5] + 10) = \max(15, 25) = 25$
- **Capacity 7:** $dp[3][7] = \max(dp[2][7], dp[2][6] + 10) = \max(29, 25) = 29$

i\j	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	15	15	15	15	15	15
2	0	0	15	15	15	15	15	29
3	0	10	15	25	25	25	25	29
4	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0

Item 4 (Value = 45, Weight = 3)

- **Capacity 1 to 2:** Cannot include this item.
- **Capacity 3:** $dp[4][3] = \max(dp[3][3], dp[3][0] + 45) = \max(25, 45) = 45$
- **Capacity 4:** $dp[4][4] = \max(dp[3][4], dp[3][1] + 45) = \max(25, 55) = 55$
- **Capacity 5:** $dp[4][5] = \max(dp[3][5], dp[3][2] + 45) = \max(25, 55) = 55$
- **Capacity 6:** $dp[4][6] = \max(dp[3][6], dp[3][3] + 45) = \max(25, 70) = 70$
- **Capacity 7:** $dp[4][7] = \max(dp[3][7], dp[3][4] + 45) = \max(29, 70) = 70$

Item 5 (Value = 30, Weight = 4)

- **Capacity 1 to 3:** Cannot include this item.
- **Capacity 4:** $dp[5][4] = \max(dp[4][4], dp[4][0] + 30) = \max(55, 30) = 55$
- **Capacity 5:** $dp[5][5] = \max(dp[4][5], dp[4][1] + 30) = \max(55, 30) = 55$
- **Capacity 6:** $dp[5][6] = \max(dp[4][6], dp[4][2] + 30) = \max(70, 30) = 70$
- **Capacity 7:** $dp[5][7] = \max(dp[4][7], dp[4][3] + 30) = \max(70, 75) = 75$

Step 3: Final DP Table

i\j	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	15	15	15	15	15	15
2	0	0	15	15	15	15	15	29
3	0	10	15	25	25	25	25	29
4	0	10	15	45	55	55	70	70
5	0	10	15	45	55	55	70	75

Result:

The maximum value that can be obtained with a knapsack capacity of 7 is 75.

Output:

Maximum value that can be obtained: 75

The maximum value that can be obtained is stored in $dp[5][7] = 75$.

Best time to buy and sell stocks in C++

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

class BestTimeToBuyAndSellStock {
public:
    int maxProfit(vector<int>& prices) {
        if (prices.empty()) return 0;

        int maxP = 0;
        int minBP = prices[0];

        for (int prc : prices) {
            int tp = prc - minBP;
            if (tp > maxP) {
                maxP = tp;
            }
            minBP = min(minBP, prc);
        }

        return maxP;
    }

    int main() {
        BestTimeToBuyAndSellStock solution;

        // Test case 1
        vector<int> prices1 = {7, 1, 5, 3, 6, 4};
        int maxProfit1 = solution.maxProfit(prices1);
        cout << "Max profit for prices1: " << maxProfit1 <<
        endl; // Output: 5

        return 0;
    }
}
```

Output:-

maxP = 5 (Maximum profit)

Let's walk through a **dry run in tabular form** of your code for:

`vector<int> prices1 = {7, 1, 5, 3, 6, 4};`

💡 **Variables:**

- `minBP` = Minimum Buying Price seen so far.
- `tp` = Temporary Profit (current price - `minBP`).
- `maxP` = Maximum Profit observed.

💡 **Dry Run Table:**

Day (Index)	Price	minBP (min so far)	tp = price - minBP	maxP (max profit so far)
0	7	7	0	0
1	1	1	0	0
2	5	1	4	4
3	3	1	2	4
4	6	1	5	5 ✓
5	4	1	3	5

✓ **Final Answer:**

Max profit for prices1: 5

Best time to buy and Sell Stocks infinite in C++

```
#include <iostream>
#include <vector>

using namespace std;

class BestTimeToBuyAndSellStocksInfiniteTransactions {
public:
    int maxProfit(vector<int>& prices) {
        if (prices.empty()) return 0;

        int bd = 0; // Buy day
        int sd = 0; // Sell day
        int profit = 0;

        for (int i = 1; i < prices.size(); ++i) {
            if (prices[i] >= prices[i - 1]) {
                sd++;
            } else {
                profit += prices[sd] - prices[bd];
                bd = sd = i;
            }
        }

        profit += prices[sd] - prices[bd];
        return profit;
    }

    int main() {
        BestTimeToBuyAndSellStocksInfiniteTransactions
        solution;

        // Test case
        vector<int> prices = {11, 6, 7, 19, 4, 1, 6, 18, 4};
        int maxProfit = solution.maxProfit(prices);
        cout << "Max profit: " << maxProfit << endl; //
Output: 30

        return 0;
    }
}
```

Let's perform a **tabular dry run** of your code for the input:

prices = {11, 6, 7, 19, 4, 1, 6, 18, 4}

✓ Logic Summary:

- Buy at bd (buy day), sell at sd (sell day).
- Keep increasing sd as long as prices go up or stay the same.
- When price drops, add profit of the last segment (prices[sd] - prices[bd]) and reset bd = sd = i.

Dry Run Table:

i	prices[i]	Action Taken	bd	sd	Segment Profit	Total Profit
0	11	Initial buy	0	0		0
1	6	Drop → sell at 11, profit = 0	1	1	11 - 11 = 0	0
2	7	Rise → extend sell day	1	2		0
3	19	Rise → extend sell day	1	3		0
4	4	Drop → sell at 19, profit = 19 - 6 = 13	4	4	19 - 6 = 13	13
5	1	Drop → sell at 4, profit = 0	5	5	4 - 4 = 0	13
6	6	Rise → extend sell day	5	6		13
7	18	Rise → extend sell day	5	7		13
8	4	Drop → sell at 18, profit = 18 - 1 = 17	8	8	18 - 1 = 17	30
—	—	Final segment (bd == sd == 8) → 0 profit			4 - 4 = 0	30

✓ Final Output:

	<p>Max profit: 30</p> <p> Insight:</p> <p>You earned profit from:</p> <ul style="list-style-type: none">• Buying at 6 → selling at 19 (Profit: 13)• Buying at 1 → selling at 18 (Profit: 17)
--	---

Output:-
Max profit: 30

Climbing Stairs in C++

```
#include <iostream>
#include <vector>
#include <climits> // For INT_MAX

using namespace std;

void printMinSteps(vector<int>& arr) {
    int n = arr.size();
    vector<int> dp(n + 1, INT_MAX); // Use INT_MAX
for initialization

    dp[n] = 0; // Base case: 0 steps needed from the end

    for (int i = n - 1; i >= 0; i--) {
        if (arr[i] > 0) {
            int minSteps = INT_MAX;
            for (int j = 1; j <= arr[i] && (i + j) < dp.size(); j++) {
                if (dp[i + j] != INT_MAX) {
                    minSteps = min(minSteps, dp[i + j]);
                }
            }
            if (minSteps != INT_MAX) {
                dp[i] = minSteps + 1;
            }
        }
    }

    // Printing the dp array
    for (int i = 0; i < dp.size(); i++) {
        cout << " " << dp[i];
    }
    cout << endl;
}

int main() {
    vector<int> arr = {1, 5, 2, 3, 1};
    printMinSteps(arr);

    return 0;
}
```

Given:

`vector<int> arr = {1, 5, 2, 3, 1};`

The length of arr is **5**, so dp is initialized as:

`dp = [INT_MAX, INT_MAX, INT_MAX, INT_MAX, INT_MAX, 0] // (size = 6, last element is 0)`

Dry Run with Iteration Table

The loop iterates from **i = n - 1 to 0**, checking possible jumps and updating dp[i].

Iteration (i)	arr[i]	Possible Jumps	Min Steps from Reachable Positions	Updated dp[i]
4 (last)	1	(4→5)	dp[5] = 0 → min(∞, 0)	dp[4] = 1
3	3	(3→4, 3→5)	dp[4] = 1, dp[5] = 0 → min(∞, 1, 0)	dp[3] = 1
2	2	(2→3, 2→4)	dp[3] = 1, dp[4] = 1 → min(∞, 1, 1)	dp[2] = 2
1	5	(1→2, 1→3, 1→4, 1→5)	dp[2] = 2, dp[3] = 1, dp[4] = 1, dp[5] = 0 → min(∞, 2, 1, 1, 0)	dp[1] = 1
0 (first)	1	(0→1)	dp[1] = 1 → min(∞, 1)	dp[0] = 2

Final dp Array

After all iterations, the **dp array** will be:

`dp = [2, 1, 2, 1, 1, 0]`

Output:

2 1 2 1 1 0

Output:-

① Printed dp: 2 1 2 1 1 0

② The minimum steps to reach the end starting from index 0 is dp[0] = 2.

Coin Change Combination in C++

```
#include <iostream>
#include <vector>

using namespace std;

int main() {
    vector<int> arr = {2, 3, 5};
    int amt = 7;
    vector<int> dp(amt + 1, 0);
    dp[0] = 1; // Base case: 1 way to make amount 0
    (using no coins)

    for (int i = 0; i < arr.size(); i++) {
        for (int j = arr[i]; j <= amt; j++) {
            dp[j] += dp[j - arr[i]];
        }
    }

    cout << dp[amt] << endl; // Output the number of
    combinations for amount `amt`

    return 0;
}
```

Initial dp Array

Before processing:

$\text{arr} = [2, 3, 5]$

$\text{dp} = [1, 0, 0, 0, 0, 0, 0, 0]$

(Index represents amount: 0 to 7)

Dry Run with Iteration Table

Processing coin 2

j (amt)	$\text{dp}[j] = \text{dp}[j] + \text{dp}[j - 2]$	Updated dp
2	$\text{dp}[2] += \text{dp}[0] =$ 1	[1, 0, 1, 0, 0, 0, 0, 0]
3	$\text{dp}[3] += \text{dp}[1] =$ 0	[1, 0, 1, 0, 0, 0, 0, 0]
4	$\text{dp}[4] += \text{dp}[2] =$ 1	[1, 0, 1, 0, 1, 0, 0, 0]
5	$\text{dp}[5] += \text{dp}[3] =$ 0	[1, 0, 1, 0, 1, 0, 0, 0]
6	$\text{dp}[6] += \text{dp}[4] =$ 1	[1, 0, 1, 0, 1, 0, 1, 0]
7	$\text{dp}[7] += \text{dp}[5] =$ 0	[1, 0, 1, 0, 1, 0, 1, 0]

Processing coin 3

j (amt)	$\text{dp}[j] = \text{dp}[j] + \text{dp}[j - 3]$	Updated dp
3	$\text{dp}[3] += \text{dp}[0] =$ 1	[1, 0, 1, 1, 1, 0, 1, 0]
4	$\text{dp}[4] += \text{dp}[1] =$ 0	[1, 0, 1, 1, 1, 0, 1, 0]
5	$\text{dp}[5] += \text{dp}[2] =$ 1	[1, 0, 1, 1, 1, 1, 1, 0]
6	$\text{dp}[6] += \text{dp}[3] =$ 1	[1, 0, 1, 1, 1, 1, 2, 0]
7	$\text{dp}[7] += \text{dp}[4] =$ 1	[1, 0, 1, 1, 1, 1, 2, 1]

Processing coin 5

j (amt)	dp[j] = dp[j] + dp[j - 5]	Updated dp
5	dp[5] += dp[0] = 1	[1, 0, 1, 1, 1, 2, 2, 1]
6	dp[6] += dp[1] = 0	[1, 0, 1, 1, 1, 2, 2, 1]
7	dp[7] += dp[2] = 1	[1, 0, 1, 1, 1, 2, 2, 2]

Final dp Array

After processing all coins:

dp = [1, 0, 1, 1, 1, 2, 2, 2]

Final Output

2

This means **there are 2 ways to form amount 7 using {2, 3, 5}**:

1. 2 + 2 + 3
2. 2 + 5

Output:-
2

Coin Change Permutation in C++

```
#include <iostream>
#include <vector>

using namespace std;

int main() {
    vector<int> coins = {2, 3, 5};
    int tar = 7;
    vector<int> dp(tar + 1, 0);
    dp[0] = 1; // Base case: 1 way to make amount 0
    (using no coins)

    for (int amt = 1; amt <= tar; amt++) {
        for (int coin : coins) {
            if (coin <= amt) {
                int ramt = amt - coin;
                dp[amt] += dp[ramt];
            }
        }
    }

    cout << dp[tar] << endl; // Output the number of
    permutations to make the target amount

    return 0;
}
```

Initial dp Array

Before processing:

$dp = [1, 0, 0, 0, 0, 0, 0, 0, 0]$ // (Indexes represent amounts from 0 to 7)

Dry Run with Iteration Table

Iterating over amt from 1 to 7

amt	Coin Used	$dp[amt] = dp[amt] + dp[amt - coin]$	Updated dp
1	2 (skipped)	-	[1, 0, 0, 0, 0, 0, 0, 0]
	3 (skipped)	-	
	5 (skipped)	-	
2	2	$dp[2] += dp[0] = 1$	[1, 0, 1, 0, 0, 0, 0]
	3, 5 (skipped)	-	
	2	$dp[3] += dp[1] = 0$	[1, 0, 1, 0, 0, 0]
3	3	$dp[3] += dp[0] = 1$	[1, 0, 1, 1, 0, 0, 0]
	5 (skipped)	-	
	2	$dp[4] += dp[2] = 1$	[1, 0, 1, 1, 1, 0, 0]
4	3	$dp[4] += dp[1] = 0$	[1, 0, 1, 1, 1, 0, 0]
	5 (skipped)	-	
	2	$dp[5] += dp[3] = 1$	[1, 0, 1, 1, 1, 1, 0]
5	3	$dp[5] += dp[2] = 1$	[1, 0, 1, 1, 1, 1, 2]
	5	$dp[5] += dp[0] = 1$	[1, 0, 1, 1, 1, 1, 3]
	2	$dp[6] += dp[4] = 1$	[1, 0, 1, 1, 1, 1, 3]
6	3	$dp[6] += dp[3] = 1$	[1, 0, 1, 1, 1, 1, 3, 2]
	5	$dp[6] += dp[1] = 0$	[1, 0, 1, 1, 1, 1, 3, 2, 0]
	2	$dp[7] += dp[5] = 3$	[1, 0, 1, 1, 1, 1, 3, 2, 3]
7	3	$dp[7] += dp[4] = 1$	[1, 0, 1, 1, 1, 1, 3, 2, 4]
	5	$dp[7] += dp[2] = 1$	[1, 0, 1, 1, 1, 1, 3, 2, 5]

Final dp Array

After processing all amounts:

$dp = [1, 0, 1, 1, 1, 3, 2, 5]$

Final Output

5

This means **there are 5 different permutations to form amount 7 using {2, 3, 5}**:

1. **2 + 2 + 3**
2. **2 + 3 + 2**
3. **3 + 2 + 2**
4. **2 + 5**
5. **5 + 2**

Output:-

5

Friend's Pairing in C++

```
#include <iostream>
#include <vector>

using namespace std;

int main() {
    int n = 3;

    vector<int> dp(n + 1);
    dp[1] = 1;
    dp[2] = 2;

    for (int i = 3; i <= n; i++) {
        dp[i] = dp[i - 1] + dp[i - 2] * (i - 1);
    }

    cout << dp[n] << endl;

    return 0;
}
```

Output:-
4

Dry Run with Iteration Table

Initial State

dp = [?, 1, 2] // (dp[0] is unused)

Iterating from i = 3 to n = 3

i	Calculation	Updated dp[i]
3	dp[3] = dp[2] + dp[1] * (3 - 1)	dp[3] = 2 + 1 * 2 = 4

Final dp Array

dp = [?, 1, 2, 4]

Final Output

4

GoldMine in C++

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
    int grid[4][4] = {
        {8, 2, 1, 6},
        {6, 5, 5, 2},
        {2, 1, 0, 3},
        {7, 2, 2, 4}
    };

    int n = 4; // Number of rows
    int m = 4; // Number of columns

    // Initialize dp array
    vector<vector<int>> dp(n, vector<int>(m, 0));

    // Fill dp array from rightmost column to left
    for (int j = m - 1; j >= 0; j--) {
        for (int i = n - 1; i >= 0; i--) {
            if (j == m - 1) {
                dp[i][j] = grid[i][j];
            } else if (i == n - 1) {
                dp[i][j] = grid[i][j] + max(dp[i][j + 1], dp[i - 1][j + 1]);
            } else if (i == 0) {
                dp[i][j] = grid[i][j] + max(dp[i][j + 1], dp[i + 1][j + 1]);
            } else {
                dp[i][j] = grid[i][j] + max(dp[i][j + 1], max(dp[i - 1][j + 1], dp[i + 1][j + 1]));
            }
        }
    }

    // Find the maximum value in the first column of dp
    // array
    int maxGold = dp[0][0];
    for (int i = 1; i < n; i++) {
        if (dp[i][0] > maxGold) {
            maxGold = dp[i][0];
        }
    }

    cout << maxGold << endl;

    return 0;
}
```

Let's do a **tabular dry run** of your gold mine problem (classic DP), where the goal is to find the **maximum gold** that can be collected from **any cell in the first column to the last column**, moving only in:

- right (\rightarrow)
- right-up (\nearrow)
- right-down (\searrow)

Given grid[4][4]:

[8, 2, 1, 6]
[6, 5, 5, 2]
[2, 1, 0, 3]
[7, 2, 2, 4]

DP Formula:

For $dp[i][j]$:

- If $j == \text{last column}$: $dp[i][j] = \text{grid}[i][j]$
- If $i == 0$: no up \rightarrow use right and right-down
- If $i == n-1$: no down \rightarrow use right and right-up
- Else: consider all 3 \rightarrow right, right-up, right-down

Filling dp from right to left:

We'll fill the DP matrix from column $j = 3$ to 0.

Step-by-step (column by column):

i\j	0	1	2	3
0	?	?	?	6
1	?	?	?	2
2	?	?	?	3
3	?	?	?	4

Fill Column 2 ($j = 2$):

$dp[i][2] = \text{grid}[i][2] + \max(\text{dp}[i][3], \text{dp}[i-1][3], \text{dp}[i+1][3])$

i	grid[i][2]	dp options	max	dp[i][2]
3	2	dp[3][3]=4, dp[2] [3]=3	4	6
2	0	3, 2, 4	4	4

i	grid[i][2]	dp options	max	dp[i][2]
1	5	2, 6, 3	6	11
0	1	6, 2	6	7

Fill Column 1 (j = 1):

i	grid[i][1]	dp options	max	dp[i][1]
3	2	6, 4	6	8
2	1	4, 11, 6	11	12
1	5	11, 7, 4	11	16
0	2	7, 11	11	13

Fill Column 0 (j = 0):

i	grid[i][0]	dp options	max	dp[i][0]
3	7	8, 12	12	19
2	2	12, 16, 8	16	18
1	6	16, 13, 12	16	22
0	8	13, 16	16	24

✓ Final dp Table:

i\j	0	1	2	3
0	24	13	7	6
1	22	16	11	2
2	18	12	4	3
3	19	8	6	4

□ Max Gold = max(dp[0][0], dp[1][0], dp[2][0], dp[3][0]) = 24

✓ Output:

24

Output:
24

Min Cost Path in C++

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
    int n = 4; // Number of rows
    int m = 4; // Number of columns
    int grid[4][4] = {
        {8, 2, 1, 6},
        {6, 5, 5, 2},
        {2, 1, 0, 3},
        {7, 2, 2, 4}
    };

    // Initialize dp array
    vector<vector<int>> dp(n, vector<int>(m, 0));

    // Fill dp array from bottom-right to top-left
    for (int i = n - 1; i >= 0; i--) {
        for (int j = m - 1; j >= 0; j--) {
            if (i == n - 1 && j == m - 1) {
                dp[i][j] = grid[i][j];
            } else if (i == n - 1) {
                dp[i][j] = dp[i][j + 1] + grid[i][j];
            } else if (j == m - 1) {
                dp[i][j] = dp[i + 1][j] + grid[i][j];
            } else {
                dp[i][j] = grid[i][j] + min(dp[i][j + 1], dp[i + 1][j]);
            }
        }
    }

    // Print the minimum cost path sum
    cout << dp[0][0] << endl;

    return 0;
}
```

Input Grid:

[8, 2, 1, 6]
[6, 5, 5, 2]
[2, 1, 0, 3]
[7, 2, 2, 4]

We're filling the $dp[i][j]$ table from **bottom-right to top-left**.

✓ DP Formula Recap:

```
if (i == n - 1 && j == m - 1)
    dp[i][j] = grid[i][j];
else if (i == n - 1)
    dp[i][j] = dp[i][j + 1] + grid[i][j];
else if (j == m - 1)
    dp[i][j] = dp[i + 1][j] + grid[i][j];
else
    dp[i][j] = grid[i][j] + min(dp[i][j + 1], dp[i + 1][j]);
```

█ DP Table (Filled from bottom-right):

Let's build $dp[i][j]$ step by step:

Starting from $dp[3][3] = grid[3][3] = 4$

Then filling right-to-left and bottom-to-top:

i\j	0	1	2	3
0	?	?	?	?
1	?	?	?	?
2	?	?	?	?
3	15	8	6	4

Now build upward:

Row 2:

- $dp[2][3] = grid[2][3] + dp[3][3] = 3 + 4 = 7$
- $dp[2][2] = 0 + \min(7, 6) = 6$
- $dp[2][1] = 1 + \min(6, 8) = 7$
- $dp[2][0] = 2 + \min(7, 15) = 9$

Row 1:

- $dp[1][3] = 2 + 7 = 9$
- $dp[1][2] = 5 + \min(9, 6) = 11$
- $dp[1][1] = 5 + \min(11, 7) = 12$
- $dp[1][0] = 6 + \min(12, 9) = 15$

Row 0:

- $dp[0][3] = 6 + 9 = 15$
- $dp[0][2] = 1 + \min(15, 11) = 12$

- $dp[0][1] = 2 + \min(12, 12) = 14$
- $dp[0][0] = 8 + \min(14, 15) = 22$

✓ Final DP Table:

i\j	0	1	2	3
0	22	14	12	15
1	15	12	11	9
2	9	7	6	7
3	15	8	6	4

☒ Output:

22

Output:

22

Paint Houses in C++

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
    // Input array representing costs to paint each
    // house with three colors
    vector<vector<int>> arr = {{1, 5, 7}, {5, 8, 4}, {3, 2,
    9}, {1, 2, 4}};
    int n = arr.size(); // Number of houses

    // Initialize dp array
    vector<vector<long long>> dp(n, vector<long
    long>(3, 0));

    // Base case: First row initialization
    dp[0][0] = arr[0][0];
    dp[0][1] = arr[0][1];
    dp[0][2] = arr[0][2];

    // Fill dp array from second row onwards
    for (int i = 1; i < n; i++) {
        dp[i][0] = arr[i][0] + min(dp[i - 1][1], dp[i - 1][2]);
        dp[i][1] = arr[i][1] + min(dp[i - 1][0], dp[i - 1][2]);
        dp[i][2] = arr[i][2] + min(dp[i - 1][0], dp[i - 1][1]);
    }

    // Find the minimum cost to paint all houses
    long long ans = min(dp[n - 1][0], min(dp[n - 1][1],
    dp[n - 1][2]));

    // Output the minimum cost
    cout << ans << endl;

    return 0;
}
```

Output:-
8

Input Matrix (Cost of painting houses):

House 0: [1, 5, 7]
 House 1: [5, 8, 4]
 House 2: [3, 2, 9]
 House 3: [1, 2, 4]

We denote the colors as:

- 0 → Red
- 1 → Blue
- 2 → Green

DP Table Filling Explanation:

House	dp[i][0] (Red)	dp[i][1] (Blue)	dp[i][2] (Green)
0	1	5	7
1	$5 + \min(5, 7) = 10$	$8 + \min(1, 7) = 9$	$4 + \min(1, 5) = 5$
2	$3 + \min(9, 5) = 8$	$2 + \min(10, 5) = 7$	$9 + \min(10, 9) = 18$
3	$1 + \min(7, 18) = 8$	$2 + \min(8, 18) = 10$	$4 + \min(8, 7) = 11$

Final DP Table:

House	Red	Blue	Green
0	1	5	7
1	10	9	5
2	8	7	18
3	8	10	11

Output:

The minimum total cost is:

$$\min(8, 10, 11) = 8$$

Target sum Subset in C++

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

bool targetSumSubsets(vector<int>& arr, int target) {
    int n = arr.size();
    vector<vector<bool>> dp(n + 1, vector<bool>(target + 1, false));

    for (int i = 0; i <= n; i++) {
        for (int j = 0; j <= target; j++) {
            if (i == 0 && j == 0) {
                dp[i][j] = true;
            } else if (i == 0) {
                dp[i][j] = false;
            } else if (j == 0) {
                dp[i][j] = true;
            } else {
                if (dp[i - 1][j]) {
                    dp[i][j] = true;
                } else {
                    int val = arr[i - 1];
                    if (j >= val && dp[i - 1][j - val]) {
                        dp[i][j] = true;
                    }
                }
            }
        }
    }
    return dp[n][target];
}
```

```
int main() {
    vector<int> arr = {4, 2, 7, 1, 3};
    int target = 10;

    if (targetSumSubsets(arr, target)) {
        cout << "True" << endl;
    } else {
        cout << "False" << endl;
    }

    return 0;
}
```

We have **array**:

$\text{arr} = \{4, 2, 7, 1, 3\}$, $\text{target} = 10$

We create a **dp table of size $(n+1) \times (\text{target}+1)$** :
 $\text{dp}[i][j] \rightarrow i$ is the first i elements, j is the sum.

Initial Table (Before Processing)

i \ j	0	1	2	3	4	5	6	7	8	9	10
0	T	F	F	F	F	F	F	F	F	F	F
1											
2											
3											
4											
5											

- **dp[0][0] = true** → A sum of 0 can be achieved with an empty subset.
- **dp[0][j] = false** for $j > 0$ → No subset can sum up to a positive number with zero elements.

Step 2: Fill the Table

We iterate through $i = 1$ to n , updating $\text{dp}[i][j]$.

Processing $\text{arr}[0] = 4$

We consider only element 4.

- $\text{dp}[1][4] = \text{true}$ (We can form sum 4 using $\{4\}$)

i \ j	0	1	2	3	4	5	6	7	8	9	10
0	T	F	F	F	F	F	F	F	F	F	F
1	T	F	F	F	T	F	F	F	F	F	F

Processing $\text{arr}[1] = 2$

Now considering $\{4, 2\}$:

- $\text{dp}[2][2] = \text{true}$ (Subset $\{2\}$)
- $\text{dp}[2][4] = \text{true}$ (Subset $\{4\}$)
- $\text{dp}[2][6] = \text{true}$ (Subset $\{4, 2\}$)

i\j	0	1	2	3	4	5	6	7	8	9	10
0	T	F	F	F	F	F	F	F	F	F	F
1	T	F	F	F	T	F	F	F	F	F	F
2	T	F	T	F	T	F	T	F	F	F	F

Processing arr[2] = 7

Now considering {4,2,7}:

- dp[3][7] = true (Subset {7})
- dp[3][9] = true (Subset {2,7})
- dp[3][10] = true (Subset {4,2,7})

i\j	0	1	2	3	4	5	6	7	8	9	10
0	T	F	F	F	F	F	F	F	F	F	F
1	T	F	F	F	T	F	F	F	F	F	F
2	T	F	T	F	T	F	T	F	F	F	F
3	T	F	T	F	T	F	T	T	F	T	T

Processing arr[3] = 1

Now considering {4,2,7,1}:

- dp[4][1] = true (Subset {1})
- dp[4][3] = true (Subset {2,1})
- dp[4][5] = true (Subset {4,1})
- dp[4][8] = true (Subset {7,1})

i\j	0	1	2	3	4	5	6	7	8	9	10
0	T	F	F	F	F	F	F	F	F	F	F
1	T	F	F	F	T	F	F	F	F	F	F
2	T	F	T	F	T	F	T	F	F	F	F
3	T	F	T	F	T	F	T	T	F	T	T
4	T	T	T	T	T	T	T	T	T	T	T

Processing arr[4] = 3

Including 3 confirms all sums, but **dp[5][10]** remains **true**.

Final Answer

	Since $dp[5][10] = \text{true}$, we return true , meaning a subset exists with the sum 10 . Output: True
--	--

Output:-
True
 $dp[n][\text{target}]$ is $dp[5][10] = \text{true}$

Tiling with Dominoes in C++

```
#include <iostream>
#include <vector>

using namespace std;

int main() {
    int n = 2;

    vector<int> dp(n + 1);
    dp[1] = 1;
    dp[2] = 2;

    for (int i = 3; i <= n; i++) {
        dp[i] = dp[i - 1] + dp[i - 2];
    }

    cout << dp[n] << endl;

    return 0;
}
```

Given:

- $n = 2$

We create a dp vector of size $n+1 = 3$ and initialize the base cases:

- $dp[1] = 1$
- $dp[2] = 2$

Initial dp Table:

i\dp	0	1	2
i=0	0		
i=1		1	
i=2			2

At this point:

- $dp[0] = 0$ (this entry is not used)
- $dp[1] = 1$
- $dp[2] = 2$

For Loop Execution:

The `for` loop iterates from $i = 3$ to n . But, since $n = 2$, the loop condition $i = 3 \leq 2$ is **false**, so the loop doesn't run.

Final dp Table:

The table remains unchanged from initialization:

i\dp	0	1	2
i=0	0		
i=1		1	
i=2			2

Final Output:

The program prints $dp[n]$, where $n = 2$, so $dp[2] = 2$ is printed.

Output:-
2

All paths minimum jumps in C++

```
#include <iostream>
#include <climits>
#include <queue>
using namespace std;

class Pair {
public:
    int i, s, j;
    string psf;

    Pair(int i, int s, int j, string psf) {
        this->i = i;
        this->s = s;
        this->j = j;
        this->psf = psf;
    }
};

void solution(const int arr[], int n) {
    int dp[n];
    fill_n(dp, n, INT_MAX);
    dp[n - 1] = 0;

    for (int i = n - 2; i >= 0; i--) {
        int steps = arr[i];
        int min_steps = INT_MAX;

        for (int j = 1; j <= steps && i + j < n; j++) {
            if (dp[i + j] != INT_MAX && dp[i + j] <
min_steps) {
                min_steps = dp[i + j];
            }
        }

        if (min_steps != INT_MAX) {
            dp[i] = min_steps + 1;
        }
    }

    cout << dp[0] << endl;

    queue<Pair> q;
    q.emplace(0, arr[0], dp[0], "0");

    while (!q.empty()) {
        Pair rem = q.front();
        q.pop();

        if (rem.j == 0) {
            cout << rem.psf << "." << endl;
        }

        for (int j = 1; j <= rem.s && rem.i + j < n;
j++) {
            int ci = rem.i + j;
            if (dp[ci] != INT_MAX && dp[ci] ==
rem.j - 1) {
                q.emplace(ci, arr[ci], dp[ci], rem.psf
+ "->" + to_string(ci));
            }
        }
    }
}
```

Dry Run:

Step 1: Calculate the dp array (minimum jumps to reach the end from each index)

The dp array keeps track of the minimum number of jumps required to reach the last index from any given index. Let's calculate the dp array starting from the last index (since we know that $dp[n-1] = 0$ as no jumps are needed from the last index):

- $dp[9] = 0$ (since we're already at the last index).
- $dp[8] = INT_MAX$ (can't reach the last index from index 8, because there are no valid jumps).
- $dp[7] = 1$ (one jump to index 9, because $arr[7] = 2$ allows jumping to index 9).
- $dp[6] = 1$ (one jump to index 9, because $arr[6] = 4$ allows jumping to index 9).
- $dp[5] = 2$ (minimum of $dp[6] + 1$ and $dp[7] + 1$, so $\min(1+1, 1+1) = 2$).
- $dp[4] = 2$ (minimum of $dp[5] + 1$ and $dp[6] + 1$, so $\min(2+1, 1+1) = 2$).
- $dp[3] = 2$ (minimum of $dp[4] + 1$ and $dp[5] + 1$, so $\min(2+1, 2+1) = 2$).
- $dp[2] = 3$ (can't jump to a valid position from here).
- $dp[1] = 3$ (same as above, can't jump to a valid position).
- $dp[0] = 4$ (minimum of $dp[1] + 1$, $dp[2] + 1$, and $dp[3] + 1$, so $\min(3+1, 3+1, 2+1) = 4$).

Thus, the dp array will look like this:

$$dp = \{4, 3, 3, 2, 2, 2, 1, 1, INT_MAX, 0\}$$

Step 2: Generate paths using BFS

Next, we use BFS to generate all valid paths from the start (index 0) to the end (index 9) using the minimum number of jumps ($dp[0] = 4$).

We initialize the queue with the first index 0 and process each index in the queue, exploring all possible jumps from that index:

1. Start from index 0, jump to index 3 (because $dp[3] = 2$ and $dp[0] = dp[3] + 1$).
2. From index 3, jump to index 5 (because $dp[5] = 2$ and $dp[3] = dp[5] + 1$).
3. From index 5, jump to index 6 (because $dp[6] = 1$ and $dp[5] = dp[6] + 1$).
4. From index 6, jump to index 9 (because $dp[9] = 0$ and $dp[6] = dp[9] + 1$).

This gives the path: 0 -> 3 -> 5 -> 6 -> 9.

```

    }
}

int main() {
    const int arr[] = {3, 3, 0, 2, 1, 2, 4, 2, 0, 0};
    int n = sizeof(arr) / sizeof(arr[0]);
    solution(arr, n);
    return 0;
}

```

Similarly, another valid path is:

1. Start from index 0, jump to index 3.
2. From index 3, jump to index 5.
3. From index 5, jump to index 7 (because $dp[7] = 1$ and $dp[5] = dp[7] + 1$).
4. From index 7, jump to index 9 (because $dp[9] = 0$).

This gives the path: 0 -> 3 -> 5 -> 7 -> 9.

Step 3: Final Output

The correct output should be:

4
0->3->5->6->9.
0->3->5->7->9.

Output:-

4
0->3->5->6->9.
0->3->5->7->9.

Arithmetic Slices in C++

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

int solution(const vector<int>& arr) {
    vector<int> dp(arr.size(), 0);
    //vector<int> dp;
    int ans = 0;
    for (size_t i = 2; i < arr.size(); i++) {
        if (arr[i] - arr[i - 1] == arr[i - 1] - arr[i - 2]) {
            dp[i] = dp[i - 1] + 1;
            ans += dp[i];
        }
    }
    return ans;
}

int main() {
    vector<int> arr = {2, 5, 9, 12, 15, 18, 22, 26, 30, 34,
    36, 38, 40, 41};
    cout << solution(arr) << endl;
    return 0;
}
```

Given Input

vector<int> arr = {2, 5, 9, 12, 15, 18, 22, 26, 30, 34, 36, 38, 40, 41};

- **Size of array:** n = 14

Step-by-Step Dry Run

We'll track how $dp[i]$ and ans evolve.

Initialization

Index (i)	arr[i]	dp[i]	ans (Sum of dp[i])
0	2	-	-
1	5	-	-

Loop Execution ($i = 2$ to $i = 13$)

i	arr[i]	Check Condition $arr[i] - arr[i-1] == arr[i-1] - arr[i-2]$	dp[i] Calculation	ans Update
2	9	$(9 - 5) == (5 - 2)$ → 4 == 3 ✗	$dp[2] = 0$	ans = 0
3	12	$(12 - 9) == (9 - 5)$ → 3 == 4 ✗	$dp[3] = 0$	ans = 0
4	15	$(15 - 12) == (12 - 9)$ → 3 == 3 ✓	$dp[4] = dp[3] + 1 = 1$	ans = 1
5	18	$(18 - 15) == (15 - 12)$ → 3 == 3 ✓	$dp[5] = dp[4] + 1 = 2$	ans = 3
6	22	$(22 - 18) == (18 - 15)$ → 4 == 3 ✗	$dp[6] = 0$	ans = 3
7	26	$(26 - 22) == (22 - 18)$ → 4 == 4 ✓	$dp[7] = dp[6] + 1 = 1$	ans = 4
8	30	$(30 - 26) == (26 - 22)$ → 4 == 4 ✓	$dp[8] = dp[7] + 1 = 2$	ans = 6
9	34	$(34 - 30) == (30 - 26)$ → 4 == 4 ✓	$dp[9] = dp[8] + 1 = 3$	ans = 9
10	36	$(36 - 34) == (34 - 30)$ → 2 == 4 ✗	$dp[10] = 0$	ans = 9
11	38	$(38 - 36) == (36 - 34)$ → 2 == 2 ✓	$dp[11] = dp[10] + 1 = 1$	ans = 10
12	40	$(40 - 38) == (38 - 36)$ → 2 == 2 ✓	$dp[12] = dp[11] + 1 = 2$	ans = 12
13	41	$(41 - 40) == (40 - 38)$ → 1 == 2 ✗	$dp[13] = 0$	ans = 12

Final dp Table

Index (i)	arr[i]	dp[i]	ans (Sum of dp[i])
0	2	-	-
1	5	-	-
2	9	0	0
3	12	0	0
4	15	1	1
5	18	2	3
6	22	0	3
7	26	1	4
8	30	2	6
9	34	3	9
10	36	0	9
11	38	1	10
12	40	2	12
13	41	0	12

Final Output

12

Output:-
12

Balanced Parenthesis in C++

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

int main() {
    int n = 5;
    vector<int> dp(n + 1, 0);
    dp[0] = 1;
    dp[1] = 1;

    for (int i = 2; i <= n; i++) {
        int inside = i - 1;
        int outside = 0;
        while (inside >= 0) {
            dp[i] += dp[inside] * dp[outside];
            inside--;
            outside++;
        }
    }

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

    // char c = 'b';
    // cout << (c - '0') << endl;
}

return 0;
}
```

Dry Run with Table

Let's analyze **step-by-step calculations for n = 5.**

Initialization

i	inside	outside	Computation	dp[i]
0	-	-	dp[0] = 1	1
1	-	-	dp[1] = 1	1

Filling dp Array

i	inside	outside	Computation (dp[i] += dp[inside] * dp[outside])	dp[i]
2	1	0	dp[2] += dp[1] * dp[0] = 1 * 1	1
	0	1	dp[2] += dp[0] * dp[1] = 1 * 1	2
3	2	0	dp[3] += dp[2] * dp[0] = 2 * 1	2
	1	1	dp[3] += dp[1] * dp[1] = 1 * 1	3
	0	2	dp[3] += dp[0] * dp[2] = 1 * 2	5
4	3	0	dp[4] += dp[3] * dp[0] = 5 * 1	5
	2	1	dp[4] += dp[2] * dp[1] = 2 * 1	7
	1	2	dp[4] += dp[1] * dp[2] = 1 * 2	9
	0	3	dp[4] += dp[0] * dp[3] = 1 * 5	14
5	4	0	dp[5] += dp[4] * dp[0] = 14 * 1	14
	3	1	dp[5] += dp[3] * dp[1] = 5 * 1	19
	2	2	dp[5] += dp[2] * dp[2] = 2 * 2	23
	1	3	dp[5] += dp[1] * dp[3] = 1 * 5	28

i	inside	outside	Computation ($dp[i] += dp[inside] * dp[outside]$)	$dp[i]$
0	4		$dp[5] += dp[0] * dp[4] = 1 * 14$	42

Final dp Array Output
1 1 2 5 14 42

Final Output ($dp[5]$)
42

This means **42 unique BSTs** can be formed using 5 nodes.

Output:-
1 1 2 5 14 42

Burst Balloons In C++

```
#include <iostream>
#include <climits>
using namespace std;

int sol(int arr[], int n) {
    int dp[n][n];

    // Initialize the dp array with zeros
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            dp[i][j] = 0;
        }
    }

    for (int g = 0; g < n; g++) {
        for (int i = 0, j = g; j < n; i++, j++) {
            int maxCoins = INT_MIN;
            for (int k = i; k <= j; k++) {
                int left = (k == i) ? 0 : dp[i][k - 1];
                int right = (k == j) ? 0 : dp[k + 1]
[j];
                int val = (i == 0 ? 1 : arr[i - 1]) *
arr[k] * (j == n - 1 ? 1 : arr[j + 1]);
                int total = left + right + val;
                maxCoins = max(maxCoins,
total);
            }
            dp[i][j] = maxCoins;
        }
    }
    return dp[0][n - 1];
}

int main() {
    int arr[] = {2, 3, 5};
    int n = sizeof(arr) / sizeof(arr[0]);
    cout << sol(arr, n) << endl;
    return 0;
}
```

Dry Run of sol(arr, 3)

Given Input:

arr[] = {2, 3, 5}
n = 3

Step 1: Initialize DP Table (dp[n][n])

dp = { {0, 0, 0},
 {0, 0, 0},
 {0, 0, 0} }

Step 2: Iterate Over Gaps (g)

Gap g = 0 (Single Balloons)

For g = 0, each cell dp[i][i] represents bursting a single balloon.

i	j	k (only choic e)	Left	Right	Value	dp[i] [j]
0	0	0	0	0	$1 \times 2 \times 3 = 6$	6
1	1	1	0	0	$2 \times 3 \times 5 = 30$	30
2	2	2	0	0	$3 \times 5 \times 1 = 15$	15

Updated DP Table:

dp = { {6, 0, 0},
 {0, 30, 0},
 {0, 0, 15} }

Gap g = 1 (Two Balloons)

Now we consider **two consecutive balloons**.

Case (i=0, j=1):

k	Left	Right	Value	Total
0	0	30	$1 \times 2 \times 5 = 10$	40
1	6	0	$1 \times 3 \times 5 = 15$	21

$$dp[0][1] = \max(40, 21) = 40$$

Case (i=1, j=2):

k	Left	Rig ht	Value	Total
1	0	15	$2 \times 3 \times 1 = 6$	21
2	30	0	$2 \times 5 \times 1 = 10$	40

$$dp[1][2] = \max(21, 40) = 40$$

Updated DP Table:

$$dp = \{ \{6, 40, 0\}, \\ \{0, 30, 40\}, \\ \{0, 0, 15\} \}$$

Gap g = 2 (Full Array)

Now we consider the **entire array** (i=0, j=2).

k	Left (dp[0] [k-1])	Right (dp[k+1] [2])	Value	Total
0	0	40	$1 \times 2 \times 1 = 2$	42

k	Left (dp[0] [k-1])	Right (dp[k+1] [2])	Value	Total
1	6	15	$1 \times 3 \times 1 = 3$	24
2	40	0	$1 \times 5 \times 1 = 5$	45

$$dp[0][2] = \max(42, 24, 45) = 45$$

Final DP Table:

$$dp = \{ \{6, 40, 45\}, \\ \{0, 30, 40\}, \\ \{0, 0, 15\} \}$$

Final Answer:

The function returns $dp[0][n-1] = dp[0][2] = 45$.

Final Output:

45

Output:-
45

Catalan in C++

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

int main() {
    int n = 6;
    int dp[n];
    dp[0] = 1;
    dp[1] = 1;

    for (int i = 2; i < n; i++) {
        dp[i] = 0;
        for (int j = 0; j < i; j++) {
            dp[i] += dp[j] * dp[i - j - 1];
        }
    }

    for (int i = 0; i < n; i++) {
        cout << dp[i] << " ";
    }

    return 0;
}
```

Step-by-Step Execution

Initialization:

$dp[0] = 1;$
 $dp[1] = 1;$

Iteration Table for $dp[2]$ to $dp[5]$

i	j	Computation	dp[i]
2	0	$dp[0] \times dp[1] = 1 \times 1 = 1$	1
2	1	$dp[1] \times dp[0] = 1 \times 1 = 1$	2

Final: $dp[2] = 2$

i	j	Computation	dp[i]
3	0	$dp[0] \times dp[2] = 1 \times 2 = 2$	2
3	1	$dp[1] \times dp[1] = 1 \times 1 = 1$	3
3	2	$dp[2] \times dp[0] = 2 \times 1 = 2$	5

Final: $dp[3] = 5$

i	j	Computation	dp[i]
4	0	$dp[0] \times dp[3] = 1 \times 5 = 5$	5
4	1	$dp[1] \times dp[2] = 1 \times 2 = 2$	7
4	2	$dp[2] \times dp[1] = 2 \times 1 = 2$	9
4	3	$dp[3] \times dp[0] = 5 \times 1 = 5$	14

Final: $dp[4] = 14$

i	j	Computation	dp[i]
5	0	$dp[0] \times dp[4] = 1 \times 14 = 14$	14
5	1	$dp[1] \times dp[3] = 1 \times 5 = 5$	19
5	2	$dp[2] \times dp[2] = 2 \times 2 = 4$	23
5	3	$dp[3] \times dp[1] = 5 \times 1 = 5$	28
5	4	$dp[4] \times dp[0] = 14 \times 1 = 14$	42

Final: $dp[5] = 42$

Final DP Array:

$dp[] = \{1, 1, 2, 5, 14, 42\}$

Final Output:

1 1 2 5 14 42

Output:-

1	1	2	5	14	42
---	---	---	---	----	----

Count Distinct Subsequence C++

```
#include <iostream>
#include <unordered_map>
using namespace std;

int countDistinctSubsequences(const string& str) {
    int n = str.length();
    int dp[n + 1];
    dp[0] = 1; // Empty subsequence

    unordered_map<char, int> lastOccurrence;

    for (int i = 1; i <= n; i++) {
        dp[i] = 2 * dp[i - 1];
        char ch = str[i - 1];
        if (lastOccurrence.find(ch) != lastOccurrence.end()) {
            int j = lastOccurrence[ch];
            dp[i] -= dp[j - 1];
        }
        lastOccurrence[ch] = i;
    }
    return dp[n] - 1;
}

int main() {
    string str = "abc";
    cout << countDistinctSubsequences(str) << endl;
    return 0;
}
```

Dry Run with Input "abc"

Initialization:

str = "abc";
n = 3;
dp[0] = 1; // Empty subsequence
lastOccurrence = {} // Initially empty

Iteration Table

i	str[i-1]	dp[i] Calculation	dp[i] Value	lastOccurrence Update
1	'a'	dp[1]=2×dp[0]=2×1	2	{'a': 1}
2	'b'	dp[2]=2×dp[1]=2×2	4	{'a': 1, 'b': 2}
3	'c'	dp[3]=2×dp[2]=2×4	8	{'a': 1, 'b': 2, 'c': 3}

Final Calculation

Result=dp[n]-1=8-1=7

(The -1 removes the empty subsequence.)

Final Output

7

The distinct non-empty subsequences of "abc":

a, b, c, ab, ac, bc, abc

Output:-

7

Count Palindromic Subsequence C++

```
#include <iostream>
#include <string>
using namespace std;

int
countPalindromicSubseq(const
string& str) {
    int n = str.length();
    int dp[n][n] = {0}; //
Initialize the 2D array

    for (int g = 0; g < n; g++) {
        for (int i = 0, j = g; j < n;
i++, j++) {
            if (g == 0) {
                dp[i][j] = 1;
            } else if (g == 1) {
                dp[i][j] = (str[i] ==
str[j]) ? 2 : 1;
            } else {
                if (str[i] == str[j]) {
                    dp[i][j] = dp[i][j - 1] + dp[i + 1][j] + 1;
                } else {
                    dp[i][j] = dp[i][j - 1] + dp[i + 1][j] - dp[i + 1][j - 1];
                }
            }
        }
    }

    return dp[0][n - 1];
}

int main() {
    string str = "abccbc";
    cout <<
    countPalindromicSubseq(str)
    << endl;
    return 0;
}
```

Step 1: Single Character ($g = 0$)

Each **single character** is a palindrome:

$$dp[i][i] = 1$$

Updated DP Table:

1	0	0	0	0	0
0	1	0	0	0	0
0	0	1	0	0	0
0	0	0	1	0	0
0	0	0	0	1	0
0	0	0	0	0	1

Step 2: Two-Character Substrings ($g = 1$)

i	j	Substring	str[i] == str[j]?	dp[i][j]
0	1	"ab"	✗	1
1	2	"bc"	✗	1
2	3	"cc"	✓	2
3	4	"cb"	✗	1
4	5	"bc"	✗	1

Updated DP Table:

1	1	0	0	0	0
0	1	1	0	0	0
0	0	1	2	0	0
0	0	0	1	1	0
0	0	0	0	1	1
0	0	0	0	0	1

Step 3: Three-Character Substrings ($g = 2$)

i	j	Substring	str[i] == str[j]?	Formula Used	dp[i][j]
0	2	"abc"	✗	$dp[0][2] = dp[0][1] + dp[1][2] - dp[1][1]$	2
1	3	"bcc"	✗	$dp[1][3] = dp[1][2] + dp[2][3] - dp[2][2]$	3
2	4	"ccb"	✗	$dp[2][4] = dp[2][3] + dp[3][4] - dp[3][3]$	3
3	5	"cbc"	✓	$dp[3][5] = dp[3][4] + dp[4][5] + 1$	3

Updated DP Table:

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

0 0 0 0 0 1

Step 4: Four-Character Substrings (g = 3)

i	j	Substring	str[i] == str[j]?	Formula Used	dp[i][j]
0	3	"abcc"	✗	$dp[0][3] = dp[0][2] + dp[1][3] - dp[1][2]$	4
1	4	"beccb"	✓	$dp[1][4] = dp[1][3] + dp[2][4] + 1$	7
2	5	"ccbc"	✓	$dp[2][5] = dp[2][4] + dp[3][5] + 1$	7

Updated DP Table:

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

Step 4: Four-Character Substrings (g = 4)

i	j	Substring	str[i] == str[j]?	Formula Used	dp[i][j]
0	4	"abccb"	✗	$dp[0][4] = dp[0][3] + dp[1][4] - dp[1][3]$	5
1	5	"bccbc"	✓	$dp[1][5] = dp[1][4] + dp[2][5] + 1$	9

Updated DP Table:

1 1 2 4 5 0
0 1 1 3 7 9
0 0 1 2 3 7
0 0 0 1 1 3
0 0 0 0 1 1
0 0 0 0 0 1

Step 5: Final Computation (g = 5)

$$\begin{aligned}
 dp[0][5] &= dp[0][4] + dp[1][5] - dp[1][4] \\
 dp[0][5] &= dp[0][4] + dp[1][5] - dp[1][4] \\
 dp[0][5] &= 7 + 7 - 5 = 9 \\
 dp[0][5] &= 9
 \end{aligned}$$

Output:-
9

Count Distinct Subsequence C++

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

int countValleysAndMountains(int n) {
    int dp[n + 1] = {0}; // Initialize the array with zeros
    dp[0] = 1; // Base case: empty sequence
    dp[1] = 1; // Sequence of length 1: either V or M

    for (int i = 2; i <= n; i++) {
        int valleys = 0;
        int mountains = i - 1;

        while (mountains >= 0) {
            dp[i] += dp[valleys] * dp[mountains];
            valleys++;
            mountains--;
        }
    }

    return dp[n];
}

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

Output:-
42

Step-by-Step Calculation

i	dp[i] Computation	dp[i] Value
0	dp[0] = 1	1
1	dp[1] = dp[0] * dp[0]	1
2	dp[2] = dp[0] * dp[1] + dp[1] * dp[0]	2
3	dp[3] = dp[0] * dp[2] + dp[1] * dp[1] + dp[2] * dp[0]	5
4	dp[4] = dp[0] * dp[3] + dp[1] * dp[2] + dp[2] * dp[1] + dp[3] * dp[0]	14
5	dp[5] = dp[0] * dp[4] + dp[1] * dp[3] + dp[2] * dp[2] + dp[3] * dp[1] + dp[4] * dp[0]	42

Final Output
42

Edit Distance C++

```
#include <iostream>
#include <string>
#include <algorithm>

using namespace std;

int main() {
    string s1 = "cat";
    string s2 = "cut";
    int m = s1.length();
    int n = s2.length();

    int dp[m + 1][n + 1];

    // Base cases
    for (int i = 0; i <= m; i++) dp[i][0] = i; // Deleting all characters
    for (int j = 0; j <= n; j++) dp[0][j] = j; // Inserting all characters

    // Fill the DP table
    for (int i = 1; i <= m; i++) {
        for (int j = 1; j <= n; j++) {
            if (s1[i - 1] == s2[j - 1]) {
                dp[i][j] = dp[i - 1][j - 1]; // No operation needed
            } else {
                dp[i][j] = 1 + min({dp[i - 1][j - 1], // Replace
                                    dp[i - 1][j], // Delete
                                    dp[i][j - 1]}); // Insert
            }
        }
    }

    cout << dp[m][n] << endl; // Output the minimum edit distance
}

return 0;
}
```

Dry Run ($s_1 = \text{"cat"}$, $s_2 = \text{"cut"}$)

Step 1: Initialize the DP Table

The **first row** (when s_1 is empty) represents **insertions**, and the **first column** (when s_2 is empty) represents **deletions**.

i\j	0	1	2	3
0	0	1	2	3
1	1	-	-	-
2	2	-	-	-
3	3	-	-	-

Step 2: Fill the DP Table

Iteration 1 ($i=1$, $s_1=\text{"c"}$):

- $j=1$, $s_2=\text{"c"}$ → **Same character, copy diagonal** → $dp[1][1] = dp[0][0] = 0$
- $j=2$, $s_2=\text{"cu"}$ → **Insert 'u'** → $dp[1][2] = \min(\text{Replace:1}, \text{Delete:2}, \text{Insert:0}) + 1 = 1$
- $j=3$, $s_2=\text{"cut"}$ → **Insert 't'** → $dp[1][3] = \min(\text{Replace:2}, \text{Delete:3}, \text{Insert:1}) + 1 = 2$

i\j	0	1	2	3
0	0	1	2	3
1	1	0	1	2
2	2	-	-	-
3	3	-	-	-

Iteration 2 ($i=2$, $s_1=\text{"ca"}$):

- $j=1$, $s_2=\text{"c"}$ → **Delete 'a'** → $dp[2][1] = \min(\text{Replace:1}, \text{Delete:0}, \text{Insert:2}) + 1 = 1$
- $j=2$, $s_2=\text{"cu"}$ → **Replace 'a' with 'u'** → $dp[2][2] = \min(\text{Replace:0}, \text{Delete:1}, \text{Insert:1}) + 1 = 1$
- $j=3$, $s_2=\text{"cut"}$ → **Insert 't'** → $dp[2][3] = \min(\text{Replace:1}, \text{Delete:2}, \text{Insert:1}) + 1 = 2$

i\j	0	1	2	3
0	0	1	2	3
1	1	0	1	2
2	2	-	-	-
3	3	-	-	-

i\j	0	1	2	3
1	1	0	1	2
2	2	1	1	2
3	3	-	-	-

Iteration 3 ($i=3$, $s1="cat"$):

- $j=1$, $s2="c" \rightarrow \text{Delete 'at'}$ $\rightarrow dp[3][1] = \min(\text{Replace:2}, \text{Delete:1}, \text{Insert:3}) + 1 = 2$
- $j=2$, $s2="cu" \rightarrow \text{Delete 't'}$ $\rightarrow dp[3][2] = \min(\text{Replace:1}, \text{Delete:1}, \text{Insert:2}) + 1 = 2$
- $j=3$, $s2="cut" \rightarrow \text{Replace 'a' with 'u'}$ $\rightarrow dp[3][3] = dp[2][2] = 1$ (since 'c' and 't' match)

i\j	0	1	2	3
0	0	1	2	3
1	1	0	1	2
2	2	1	1	2
3	3	2	2	1

Step 3: Output the Result

↙ The **minimum edit distance** is $dp[3][3] = 1$, meaning we need **one operation (replace 'a' with 'u')** to convert "cat" to "cut".

Output:-
1

Egg drop C++

```
#include <iostream>
#include <climits>
using namespace std;

int eggDrop(int n, int k) {
    // Initialize a 2D array for DP table
    int dp[n + 1][k + 1]; // Array with (n + 1) rows and
    (k + 1) columns
    for (int i = 0; i <= n; i++) {
        for (int j = 0; j <= k; j++) {
            dp[i][j] = 0;
        }
    }

    // Fill the DP table
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= k; j++) {
            if (i == 1) {
                dp[i][j] = j; // If only one egg is available, we
                need j trials
            } else if (j == 1) {
                dp[i][j] = 1; // If only one floor is there, one
                trial needed
            } else {
                int minDrops = INT_MAX;
                // Check all floors from 1 to j to find the
                minimum drops needed
                for (int floor = 1; floor <= j; floor++) {
                    int breaks = dp[i - 1][floor - 1]; // Egg
                    breaks, check below floors
                    int survives = dp[i][j] - floor; // Egg
                    survives, check above floors
                    int maxDrops = 1 + max(breaks,
                    survives); // Maximum drops needed in worst case
                    minDrops = min(minDrops, maxDrops); //
                    Minimum drops to find the critical floor
                }
                dp[i][j] = minDrops;
            }
        }
    }

    return dp[n][k]; // Return the minimum drops
    needed
}

int main() {
    int n = 4; // Number of eggs
    int k = 2; // Number of floors

    cout << eggDrop(n, k) << endl; // Output the
    minimum drops required
    return 0;
}
```

Step 1: Understanding the DP State

- $dp[i][j]$ = **Minimum number of trials** needed to find the critical floor with i eggs and j floors.
- If we have **1 egg**, we must check **each floor one by one** $\rightarrow dp[1][j] = j$
- If we have **1 floor**, only **1 trial** is needed $\rightarrow dp[i][1] = 1$

Step 2: Dry Run for $n = 4$ (eggs), $k = 2$ (floors)

We build the **DP table** from $dp[1][1]$ up to $dp[4][2]$.

Step 2.1: Initialize Base Cases

dp[i][j]	0 Floors	1 Floor	2 Floors
0 Eggs	0	0	0
1 Egg	0	1	2
2 Eggs	0	1	?
3 Eggs	0	1	?
4 Eggs	0	1	?

Step 2.2: Fill DP Table Using Recurrence

For $dp[i][j]$, we check all floors f from 1 to j , and take the worst-case minimum:

$$dp[i][j] = 1 + \min_{f=1}^j \max(dp[i-1][f-1], dp[i][j-f])$$

Filling for $dp[2][2]$

- Try dropping from **floor 1**:
 - If **breaks**, check below: $dp[1][0] = 0$
 - If **survives**, check above: $dp[2][1] = 1$
 - **Max** $\rightarrow \max(0,1) + 1 = 2$
- Try dropping from **floor 2**:
 - If **breaks**, check below: $dp[1][1] = 1$
 - If **survives**, check above: $dp[2][0] = 0$
 - **Max** $\rightarrow \max(1,0) + 1 = 2$
- **Final Result:** $dp[2][2] = \min(2,2) = 2$

Filling for $dp[3][2]$

- Try dropping from **floor 1**:
 - If **breaks**, check below: $dp[2][0] = 0$
 - If **survives**, check above: $dp[3][1] = 1$

- **Max** → $\max(0,1) + 1 = 2$
- Try dropping from **floor 2**:
 - If **breaks**, check below: $dp[2][1] = 1$
 - If **survives**, check above: $dp[3][0] = 0$
 - **Max** → $\max(1,0) + 1 = 2$
- **Final Result:** $dp[3][2] = \min(2,2) = 2$

Filling for $dp[4][2]$

- Try dropping from **floor 1**:
 - If **breaks**, check below: $dp[3][0] = 0$
 - If **survives**, check above: $dp[4][1] = 1$
 - **Max** → $\max(0,1) + 1 = 2$
- Try dropping from **floor 2**:
 - If **breaks**, check below: $dp[3][1] = 1$
 - If **survives**, check above: $dp[4][0] = 0$
 - **Max** → $\max(1,0) + 1 = 2$
- **Final Result:** $dp[4][2] = \min(2,2) = 2$

Final DP Table

dp[i][j]	0 Floors	1 Floor	2 Floors
0 Eggs	0	0	0
1 Egg	0	1	2
2 Eggs	0	1	2
3 Eggs	0	1	2
4 Eggs	0	1	2

Step 3: Final Answer
 $dp[4][2] = 2$

Thus, the **minimum trials needed** to determine the critical floor with **4 eggs** and **2 floors** is **2**.

Output:-
 2

Kadane Max Sum Subarray C++

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

int maxSubArraySum(const int arr[], int n) {
    int currentSum = arr[0]; // Initialize current sum
    int overallSum = arr[0];

    for (int i = 1; i < n; i++) {
        if (currentSum >= 0) {
            currentSum += arr[i]; // Add current element
            to current sum if positive
        } else {
            currentSum = arr[i]; // Start new subarray if
            current sum is negative
        }

        if (currentSum > overallSum) {
            overallSum = currentSum; // Update overall
            sum if current sum is greater
        }
    }

    return overallSum; // Return maximum sum found
}

int main() {
    const int arr[] = {5, 6, 7, 4, 3, 6, 4}; // Input array
    int n = sizeof(arr) / sizeof(arr[0]); // Determine the
    number of elements in the array

    cout << maxSubArraySum(arr, n) << endl; //
    Output maximum sum of subarray
    return 0;
}
```

Output:-
35

Dry Run with Given Input

Given array:

{5,6,7,4,3,6,4}

Step 2.1: Initialize Variables

currentSum = arr[0] = 5
overallSum = arr[0] = 5

Step 2.2: Iterate Through Array

Index (i)	Element (arr[i])	currentSum	overallSum
0	5	5	5
1	6	(5 + 6) = 11	11
2	7	(11 + 7) = 18	18
3	4	(18 + 4) = 22	22
4	3	(22 + 3) = 25	25
5	6	(25 + 6) = 31	31
6	4	(31 + 4) = 35	35

Step 3: Final Answer

Maximum Subarray Sum = 35

Largest submatrix C++

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

// Define the maximum size for the grid (you can
adjust this as needed)
const int MAX_ROWS = 100;
const int MAX_COLS = 100;

// Function to find the largest square submatrix
int largestSquareSubmatrix(const int
arr[MAX_ROWS][MAX_COLS], int rows, int cols) {
    int dp[MAX_ROWS][MAX_COLS] = {0}; // DP table
    int largestSide = 0;

    // Fill the dp array
    for (int i = rows - 1; i >= 0; i--) {
        for (int j = cols - 1; j >= 0; j--) {
            if (i == rows - 1 || j == cols - 1) {
                dp[i][j] = arr[i][j];
            } else {
                if (arr[i][j] == 0) {
                    dp[i][j] = 0;
                } else {
                    int minSide = min(dp[i][j + 1], min(dp[i +
1][j], dp[i + 1][j + 1]));
                    dp[i][j] = minSide + 1;
                }
            }
            if (dp[i][j] > largestSide) {
                largestSide = dp[i][j];
            }
        }
    }

    return largestSide; // Return the side length of the
largest square submatrix
}

int main() {
    // Define the array and its dimensions
    const int arr[MAX_ROWS][MAX_COLS] = {
        {0, 1, 0, 1, 0, 1},
        {1, 0, 1, 0, 1, 0},
        {0, 1, 1, 1, 1, 0},
        {0, 0, 1, 1, 1, 0},
        {1, 1, 1, 1, 1, 1}
    };
    int rows = 5;
    int cols = 6;

    cout << largestSquareSubmatrix(arr, rows, cols) <<
endl;

    return 0;
}
```

Output:-
3

Step 2.1: Given Matrix (arr)

```
0 1 0 1 0 1
1 0 1 0 1 0
0 1 1 1 1 0
0 0 1 1 1 0
1 1 1 1 1 1
```

Step 2.2: DP Table Construction

Step 2.2.1: Initialize dp[][] (Same as arr[][] for last row & last column)

```
0 1 0 1 0 1
1 0 1 0 1 0
0 1 1 1 1 0
0 0 1 1 1 0
1 1 1 1 1 1 <- (Same as `arr` because it's the
last row)
```

Step 2.2.2: Fill the dp[][] Table Bottom-Up

i, j	arr[i][j]	Formula Applied	dp[i][j]
(3,4)	1	1 + min(1, 1, 1)	2
(3,3)	1	1 + min(1, 1, 2)	2
(3,2)	1	1 + min(1, 2, 1)	2
(2,4)	1	1 + min(2, 1, 1)	2
(2,3)	1	1 + min(2, 2, 1)	2
(2,2)	1	1 + min(2, 2, 2)	3 (Largest Square Found)

Final dp[][] Matrix

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

Step 3: Final Answer
Largest Square Side = 3

LCS in C++

```
#include <iostream>
#include <string>
#include <algorithm> // For std::max
using namespace std;

// Define maximum possible sizes for
// the strings
const int MAX_M = 100;
const int MAX_N = 100;

int LCS(const string& s1, const
        string& s2) {
    int m = s1.length();
    int n = s2.length();

    // Initialize DP table with zeros
    int dp[MAX_M + 1][MAX_N + 1] =
{0};

    for (int i = m - 1; i >= 0; i--) {
        for (int j = n - 1; j >= 0; j--) {
            if (s1[i] == s2[j]) {
                dp[i][j] = 1 + dp[i + 1][j + 1];
            } else {
                dp[i][j] = max(dp[i + 1][j],
dp[i][j + 1]);
            }
        }
    }

    return dp[0][0];
}

int main() {
    string s1 = "abed";
    string s2 = "abbd";

    cout << LCS(s1, s2) << endl;

    return 0;
}
```

Step-by-Step Execution:

We initialize a **DP table** $dp [MAX_M+1] [MAX_N+1]$ with all zeros.

- **Strings Given:**

$s_1 = "abcd" \quad (m = 4)$
 $s_2 = "abbd" \quad (n = 4)$

- **Table Size:** $dp [5] [5]$ (since we use indices 0 to 4 inclusive)

Dry Run Table (Index-Based Execution of DP Table)

Step	i	j	s1[i]	s2[j]	Match?	Formula Used	dp[i][j] Value
1	3	3	'd'	'd'	Yes	$dp[i][j] = 1 + dp[i+1][j+1]$	$dp[3][3] = 1 + 0 = 1$
2	3	2	'd'	'b'	No	$dp[i][j] = \max(dp[i+1][j], dp[i][j+1])$	$dp[3][2] = \max(0, 1) = 1$
3	3	1	'd'	'b'	No	$dp[3][1] = \max(0, 1) = 1$	
4	3	0	'd'	'a'	No	$dp[3][0] = \max(0, 1) = 1$	
5	2	3	'c'	'd'	No	$dp[2][3] = \max(1, 0) = 1$	
6	2	2	'c'	'b'	No	$dp[2][2] = \max(1, 1) = 1$	
7	2	1	'c'	'b'	No	$dp[2][1] = \max(1, 1) = 1$	
8	2	0	'c'	'a'	No	$dp[2][0] = \max(1, 1) = 1$	
9	1	3	'b'	'd'	No	$dp[1][3] = \max(1, 0) = 1$	
10	1	2	'b'	'b'	Yes	$dp[1][2] = 1 + dp[2][3] = 1 + 1 = 2$	
11	1	1	'b'	'b'	Yes	$dp[1][1] = 1 + dp[2][2] = 1 + 1 = 2$	
12	1	0	'b'	'a'	No	$dp[1][0] = \max(1, 2) = 2$	
13	0	3	'a'	'd'	No	$dp[0][3] = \max(1, 0) = 1$	
14	0	2	'a'	'b'	No	$dp[0][2] = \max(2, 1) = 2$	
15	0	1	'a'	'b'	No	$dp[0][1] = \max(2, 2) = 2$	
16	0	0	'a'	'a'	Yes	$dp[0][0] = 1 + dp[1][1] = 1 + 2 = 3$	

Final DP Table After Execution

	a	b	b	d
a	3	2	2	1
b	2	2	2	1
c	1	1	1	1
d	1	1	1	1

Final Output

LCS ("abcd", "abbd") = 3

The longest common subsequence is "abd" (of length 3).

Output:-
3

LIS in C++

```
#include <iostream>
#include <vector>
#include <algorithm> // For std::max
using namespace std;

void LIS(const vector<int>& arr) {
    int n = arr.size();
    vector<int> dp(n, 1); // dp[i] will
    store the length of LIS ending at
    index i
    int omax = 1; // To store the overall
    maximum length of LIS

    // Compute the length of the
    Longest Increasing Subsequence
    for (int i = 1; i < n; i++) {
        int max_len = 0;
        for (int j = 0; j < i; j++) {
            if (arr[i] > arr[j]) {
                if (dp[j] > max_len) {
                    max_len = dp[j];
                }
            }
        }
        dp[i] = max_len + 1;
        if (dp[i] > omax) {
            omax = dp[i];
        }
    }

    cout << omax << " "; // Print the
    length of the LIS

    // Printing the LIS length values
    (optional)
    for (int i = 0; i < n; i++) {
        cout << dp[i] << " ";
    }
    cout << endl;
}

int main() {
    vector<int> arr = {10, 22, 9, 33, 21,
    50, 41, 60, 80, 3};

    LIS(arr);

    return 0;
}
```

Let's perform a **dry run** of the given C++ program with the input:

```
arr = {10, 22, 9, 33, 21, 50, 41, 60, 80, 3}
```

Understanding the Code

The program finds the **length of the Longest Increasing Subsequence (LIS)** using **dynamic programming**.

- $dp[i]$ stores the length of the **LIS ending at index i** .
- The **final answer** is the maximum value in $dp[]$.

Step-by-Step Dry Run

Step	i	j	arr[i]	arr[j]	arr[i] > arr[j]	dp[j]	max_len	dp[i]	omax
1	1	0	22	10	Yes	1	1	2	2
2	2	0	9	10	No	-	0	1	2
3	2	1	9	22	No	-	0	1	2
4	3	0	33	10	Yes	1	1	-	-
5	3	1	33	22	Yes	2	2	-	-
6	3	2	33	9	Yes	1	2	3	3
7	4	0	21	10	Yes	1	1	-	-
8	4	1	21	22	No	-	1	-	-
9	4	2	21	9	Yes	1	1	-	-
10	4	3	21	33	No	-	1	2	3
11	5	0	50	10	Yes	1	1	-	-
12	5	1	50	22	Yes	2	2	-	-
13	5	2	50	9	Yes	1	2	-	-
14	5	3	50	33	Yes	3	3	-	-
15	5	4	50	21	Yes	2	3	4	4
16	6	0	41	10	Yes	1	1	-	-
17	6	1	41	22	Yes	2	2	-	-
18	6	2	41	9	Yes	1	2	-	-
19	6	3	41	33	Yes	3	3	-	-
20	6	4	41	21	Yes	2	3	-	-
21	6	5	41	50	No	-	3	4	4
22	7	0	60	10	Yes	1	1	-	-
23	7	1	60	22	Yes	2	2	-	-
24	7	2	60	9	Yes	1	2	-	-
25	7	3	60	33	Yes	3	3	-	-
26	7	4	60	21	Yes	2	3	-	-
27	7	5	60	50	Yes	4	4	-	-
28	7	6	60	41	Yes	4	4	5	5

	29	8	0	80		10	Yes	1	1	-	-
	30	8	1	80		22	Yes	2	2	-	-
	31	8	2	80		9	Yes	1	2	-	-
	32	8	3	80		33	Yes	3	3	-	-
	33	8	4	80		21	Yes	2	3	-	-
	34	8	5	80		50	Yes	4	4	-	-
	35	8	6	80		41	Yes	4	4	-	-
	36	8	7	80		60	Yes	5	5	6	6
	37	9	0	3		10	No	-	0	-	-
	38	9	1	3		22	No	-	0	-	-
	39	9	2	3		9	No	-	0	-	-
	40	9	3	3		33	No	-	0	-	-
	41	9	4	3		21	No	-	0	-	-
	42	9	5	3		50	No	-	0	-	-
	43	9	6	3		41	No	-	0	-	-
	44	9	7	3		60	No	-	0	-	-
	45	9	8	3		80	No	-	0	1	6

Final Output

6 1 2 1 3 2 4 4 5 6 1

- **LIS Length:** 6
- **LIS DP Table:** [1, 2, 1, 3, 2, 4, 4, 5, 6, 1]

Output:-

6
1 2 1 2 4 4 5 6 1

Longest Bitonic Subseq In C++																																																																																				
Step-by-Step Dry Run																																																																																				
<pre>#include <iostream> #include <vector> using namespace std; int LongestBitonicSubseq(int arr[], int n) { vector<int> lis(n, 1); // lis[i] will store the // length of LIS ending at index i vector<int> lds(n, 1); // lds[i] will store the // length of LDS starting at index i // Computing LIS for (int i = 1; i < n; i++) { for (int j = 0; j < i; j++) { if (arr[j] <= arr[i]) { lis[i] = max(lis[i], lis[j] + 1); } } } // Computing LDS for (int i = n - 2; i >= 0; i--) { for (int j = n - 1; j > i; j--) { if (arr[j] <= arr[i]) lds[i] = max(lds[i], lds[j] + 1); } } int omax = 0; // To store the overall maximum // length of LBS // Finding the length of the Longest Bitonic // Subsequence for (int i = 0; i < n; i++) { omax = max(omax, lis[i] + lds[i] - 1); } return omax; } int main() { int arr[] = {10, 22, 9, 33, 21, 50, 41, 60, 80, 3}; int n = sizeof(arr) / sizeof(arr[0]); cout << LongestBitonicSubseq(arr, n) << endl; return 0; }</pre>	<p>Step 1: Compute lis[] (Longest Increasing Subsequence)</p> <p>We iterate from left to right, storing the longest increasing sequence ending at each index.</p> <table border="1"> <thead> <tr> <th>i</th><th>arr[i]</th><th>LIS Calculation (lis[i] = max(lis[i], lis[j] + 1))</th><th>lis[i]</th></tr> </thead> <tbody> <tr> <td>0</td><td>10</td><td>lis[0] = 1 (base case)</td><td>1</td></tr> <tr> <td>1</td><td>22</td><td>$10 < 22 \rightarrow \text{lis}[1] = \text{lis}[0] + 1 = 2$</td><td>2</td></tr> <tr> <td>2</td><td>9</td><td>No valid j</td><td>1</td></tr> <tr> <td>3</td><td>33</td><td>$10 < 33 \rightarrow \text{lis}[3] = \text{lis}[0] + 1 = 2$</td><td>2</td></tr> <tr> <td></td><td></td><td>$22 < 33 \rightarrow \text{lis}[3] = \text{lis}[1] + 1 = 3$</td><td>3</td></tr> <tr> <td>4</td><td>21</td><td>$10 < 21 \rightarrow \text{lis}[4] = \text{lis}[0] + 1 = 2$</td><td>2</td></tr> <tr> <td>5</td><td>50</td><td>$10 < 50 \rightarrow \text{lis}[5] = \text{lis}[0] + 1 = 2$</td><td>2</td></tr> <tr> <td></td><td></td><td>$22 < 50 \rightarrow \text{lis}[5] = \text{lis}[1] + 1 = 3$</td><td>3</td></tr> <tr> <td></td><td></td><td>$33 < 50 \rightarrow \text{lis}[5] = \text{lis}[3] + 1 = 4$</td><td>4</td></tr> <tr> <td>6</td><td>41</td><td>$10 < 41 \rightarrow \text{lis}[6] = \text{lis}[0] + 1 = 2$</td><td>2</td></tr> <tr> <td></td><td></td><td>$22 < 41 \rightarrow \text{lis}[6] = \text{lis}[1] + 1 = 3$</td><td>3</td></tr> <tr> <td></td><td></td><td>$33 < 41 \rightarrow \text{lis}[6] = \text{lis}[3] + 1 = 4$</td><td>4</td></tr> <tr> <td>7</td><td>60</td><td>$10 < 60 \rightarrow \text{lis}[7] = \text{lis}[0] + 1 = 2$</td><td>2</td></tr> <tr> <td></td><td></td><td>$22 < 60 \rightarrow \text{lis}[7] = \text{lis}[1] + 1 = 3$</td><td>3</td></tr> <tr> <td></td><td></td><td>$33 < 60 \rightarrow \text{lis}[7] = \text{lis}[3] + 1 = 4$</td><td>4</td></tr> <tr> <td></td><td></td><td>$50 < 60 \rightarrow \text{lis}[7] = \text{lis}[5] + 1 = 5$</td><td>5</td></tr> <tr> <td>8</td><td>80</td><td>$10 < 80 \rightarrow \text{lis}[8] = \text{lis}[0] + 1 = 2$</td><td>2</td></tr> <tr> <td></td><td></td><td>$22 < 80 \rightarrow \text{lis}[8] = \text{lis}[1] + 1 = 3$</td><td>3</td></tr> <tr> <td></td><td></td><td>$33 < 80 \rightarrow \text{lis}[8] = \text{lis}[3] + 1 = 4$</td><td>4</td></tr> </tbody> </table>	i	arr[i]	LIS Calculation (lis[i] = max(lis[i], lis[j] + 1))	lis[i]	0	10	lis[0] = 1 (base case)	1	1	22	$10 < 22 \rightarrow \text{lis}[1] = \text{lis}[0] + 1 = 2$	2	2	9	No valid j	1	3	33	$10 < 33 \rightarrow \text{lis}[3] = \text{lis}[0] + 1 = 2$	2			$22 < 33 \rightarrow \text{lis}[3] = \text{lis}[1] + 1 = 3$	3	4	21	$10 < 21 \rightarrow \text{lis}[4] = \text{lis}[0] + 1 = 2$	2	5	50	$10 < 50 \rightarrow \text{lis}[5] = \text{lis}[0] + 1 = 2$	2			$22 < 50 \rightarrow \text{lis}[5] = \text{lis}[1] + 1 = 3$	3			$33 < 50 \rightarrow \text{lis}[5] = \text{lis}[3] + 1 = 4$	4	6	41	$10 < 41 \rightarrow \text{lis}[6] = \text{lis}[0] + 1 = 2$	2			$22 < 41 \rightarrow \text{lis}[6] = \text{lis}[1] + 1 = 3$	3			$33 < 41 \rightarrow \text{lis}[6] = \text{lis}[3] + 1 = 4$	4	7	60	$10 < 60 \rightarrow \text{lis}[7] = \text{lis}[0] + 1 = 2$	2			$22 < 60 \rightarrow \text{lis}[7] = \text{lis}[1] + 1 = 3$	3			$33 < 60 \rightarrow \text{lis}[7] = \text{lis}[3] + 1 = 4$	4			$50 < 60 \rightarrow \text{lis}[7] = \text{lis}[5] + 1 = 5$	5	8	80	$10 < 80 \rightarrow \text{lis}[8] = \text{lis}[0] + 1 = 2$	2			$22 < 80 \rightarrow \text{lis}[8] = \text{lis}[1] + 1 = 3$	3			$33 < 80 \rightarrow \text{lis}[8] = \text{lis}[3] + 1 = 4$	4			
i	arr[i]	LIS Calculation (lis[i] = max(lis[i], lis[j] + 1))	lis[i]																																																																																	
0	10	lis[0] = 1 (base case)	1																																																																																	
1	22	$10 < 22 \rightarrow \text{lis}[1] = \text{lis}[0] + 1 = 2$	2																																																																																	
2	9	No valid j	1																																																																																	
3	33	$10 < 33 \rightarrow \text{lis}[3] = \text{lis}[0] + 1 = 2$	2																																																																																	
		$22 < 33 \rightarrow \text{lis}[3] = \text{lis}[1] + 1 = 3$	3																																																																																	
4	21	$10 < 21 \rightarrow \text{lis}[4] = \text{lis}[0] + 1 = 2$	2																																																																																	
5	50	$10 < 50 \rightarrow \text{lis}[5] = \text{lis}[0] + 1 = 2$	2																																																																																	
		$22 < 50 \rightarrow \text{lis}[5] = \text{lis}[1] + 1 = 3$	3																																																																																	
		$33 < 50 \rightarrow \text{lis}[5] = \text{lis}[3] + 1 = 4$	4																																																																																	
6	41	$10 < 41 \rightarrow \text{lis}[6] = \text{lis}[0] + 1 = 2$	2																																																																																	
		$22 < 41 \rightarrow \text{lis}[6] = \text{lis}[1] + 1 = 3$	3																																																																																	
		$33 < 41 \rightarrow \text{lis}[6] = \text{lis}[3] + 1 = 4$	4																																																																																	
7	60	$10 < 60 \rightarrow \text{lis}[7] = \text{lis}[0] + 1 = 2$	2																																																																																	
		$22 < 60 \rightarrow \text{lis}[7] = \text{lis}[1] + 1 = 3$	3																																																																																	
		$33 < 60 \rightarrow \text{lis}[7] = \text{lis}[3] + 1 = 4$	4																																																																																	
		$50 < 60 \rightarrow \text{lis}[7] = \text{lis}[5] + 1 = 5$	5																																																																																	
8	80	$10 < 80 \rightarrow \text{lis}[8] = \text{lis}[0] + 1 = 2$	2																																																																																	
		$22 < 80 \rightarrow \text{lis}[8] = \text{lis}[1] + 1 = 3$	3																																																																																	
		$33 < 80 \rightarrow \text{lis}[8] = \text{lis}[3] + 1 = 4$	4																																																																																	

		$50 < 80 \rightarrow lis[8] = lis[5] + 1 = 5$	5
		$60 < 80 \rightarrow lis[8] = lis[7] + 1 = 6$	6
9	3	No valid j	1

Final lis[] Array

lis = [1, 2, 1, 3, 2, 4, 4, 5, 6, 1]

Step 2: Compute lds[] (Longest Decreasing Subsequence)

We iterate from **right to left**, storing the longest decreasing sequence **starting from each index**.

i	arr[i]	LDS Calculation ($lds[i] = \max(lds[i], lds[j] + 1)$)	lds[i]
9	3	$lds[9] = 1$ (base case)	1
8	80	$lds[8] = 1$	1
7	60	$lds[7] = \max(lds[7], lds[8] + 1) = 2$	2
6	41	$lds[6] = \max(lds[6], lds[7] + 1) = 3$	3
5	50	$lds[5] = \max(lds[5], lds[6] + 1) = 4$	4
4	21	$lds[4] = 2$	2
3	33	$lds[3] = \max(lds[3], lds[4] + 1) = 3$	3
2	9	$lds[2] = \max(lds[2], lds[4] + 1) = 2$	2
1	22	$lds[1] = \max(lds[1], lds[2] + 1) = 3$	3
0	10	$lds[0] = \max(lds[0], lds[2] + 1) = 2$	2

Final lds[] Array

lds = [2, 3, 2, 3, 2, 4, 3, 2, 1, 1]

Step 3: Compute omax (Overall Maximum LBS)

Using:

$\text{omax} = \max(\text{lis}[i] + \text{lds}[i] - 1)$

i	lis[i]	lds[i]	lis[i] + lds[i] - 1
0	1	2	2
1	2	3	4
2	1	2	2
3	3	3	5
4	2	2	3
5	4	4	7
6	4	3	6
7	5	2	6
8	6	1	6
9	1	1	1

The **maximum** value in this list is 7.

Output:-7

Longest Common substring In C++

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

int LongestCommonSubstring(string s1,
string s2) {
    int m = s1.length();
    int n = s2.length();
    vector<vector<int>> dp(m + 1,
vector<int>(n + 1, 0));
    //int dp[m+1][n+1]={0};
    int maxLen = 0;

    for (int i = 1; i <= m; i++) {
        for (int j = 1; j <= n; j++) {
            if (s1[i - 1] == s2[j - 1]) {
                dp[i][j] = dp[i - 1][j - 1] + 1;
                maxLen = max(maxLen, dp[i][j]);
            } else {
                dp[i][j] = 0;
            }
        }
    }

    return maxLen;
}

int main() {
    string s1 = "abcp";
    string s2 = "abcy";

    cout << LongestCommonSubstring(s1, s2)
    << endl;

    return 0;
}
```

Step-by-Step DP Table Construction

i	j	s1[i-1]	s2[j-1]	Match?	dp[i][j] Calculation	Updated maxLen
1	1	a	a	✓	dp[0][0] + 1 = 1	1
1	2	a	b	✗	0	1
1	3	a	c	✗	0	1
1	4	a	y	✗	0	1
2	1	b	a	✗	0	1
2	2	b	b	✓	dp[1][1] + 1 = 2	2
2	3	b	c	✗	0	2
2	4	b	y	✗	0	2
3	1	c	a	✗	0	2
3	2	c	b	✗	0	2
3	3	c	c	✓	dp[2][2] + 1 = 3	3
3	4	c	y	✗	0	3
4	1	p	a	✗	0	3
4	2	p	b	✗	0	3
4	3	p	c	✗	0	3
4	4	p	y	✗	0	3

Final DP Table

	_	a	b	c	y
_	0	0	0	0	0
a	0	1	0	0	0
b	0	0	2	0	0
c	0	0	0	3	0
p	0	0	0	0	0

Final Answer

- **Longest Common Substring length = 3**
("abc")
- **Output:**

	3
Output:-	3

Longest Palindromic subseq In C++

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

int LongestPalindromicSubsequence(string str) {
    int n = str.length();
    //vector<vector<int>> dp(n, vector<int>(n, 0));
    int dp[n][n]={0};

    for (int g = 0; g < n; g++) {
        for (int i = 0, j = g; j < n; i++, j++) {
            if (g == 0) {
                dp[i][j] = 1;
            } else if (g == 1) {
                dp[i][j] = (str[i] == str[j]) ? 2 : 1;
            } else {
                if (str[i] == str[j]) {
                    dp[i][j] = 2 + dp[i + 1][j - 1];
                } else {
                    dp[i][j] = max(dp[i][j - 1], dp[i + 1][j]);
                }
            }
        }
    }

    return dp[0][n - 1];
}

int main() {
    string str = "abccba";

    int longestPalSubseqLen =
    LongestPalindromicSubsequence(str);
    cout << longestPalSubseqLen << endl;

    return 0;
}
```

Step-by-Step Dry Run

Let's walk through each step of filling the DP table for the input string "abccba".

Initial Setup

- Length of string $n = 6$
- Initialize a 2D DP table $dp[6][6]$ with all zeros.

Step 1: Base Case for Substrings of Length 1

When $g == 0$, each character is a subsequence of length 1.

	a	b	c	c	b	a
a	1					
b		1				
c			1			
c				1		
b					1	
a						1

Step 2: Substrings of Length 2

When $g == 1$, we check if adjacent characters match.

	a	b	c	c	b	a
a	1	1				
b		1	2			
c			1	2		
c				1	2	
b					1	2
a						1

Step 3: Substrings of Length 3 and Beyond

For substrings of length greater than 2, we

follow the general case rules.

g (Gap)	i	j	Formula Used	dp[i][j]
2	0	2	$dp[1][1] + 2$ (Match a == a)	3
2	1	3	$\max(dp[1][2], dp[2][3])$ (Max of 1 and 2)	2
2	2	4	$dp[3][3] + 2$ (Match b == b)	3
2	3	5	$\max(dp[3][4], dp[4][5])$ (Max of 1 and 2)	2
3	0	3	$dp[1][2] + 2$ (Match a == a)	3
3	1	4	$\max(dp[1][3], dp[2][4])$ (Max of 2 and 3)	3
3	2	5	$\max(dp[2][4], dp[3][5])$ (Max of 3 and 2)	3
4	0	4	$dp[1][3] + 2$ (Match a == a)	4
4	1	5	$\max(dp[1][4], dp[2][5])$ (Max of 3 and 3)	4
5	0	5	$dp[1][4] + 2$ (Match a == a)	6

Final DP Table

	a	b	c	c	b	a
a	1	1	3	3	4	6
b		1	2	2	3	4
c			1	2	3	3
c				1	2	3
b					1	2
a						1

Final Answer

The length of the **Longest Palindromic Subsequence** is stored in $dp[0][n-1] = dp[0][5] = 6$.

Output:

6

Output:-
6

Longest Palindromic substring In C++

```
#include <iostream>
#include <string>
using namespace std;

int LongestPalindromicSubstring(string str) {
    int n = str.length();
    bool dp[n][n];
    int len = 0;

    // Initialize dp array
    for (int i = 0; i < n; i++) {
        dp[i][i] = true;
    }

    // Check for substrings of length 2
    for (int i = 0; i < n - 1; i++) {
        if (str[i] == str[i + 1]) {
            dp[i][i + 1] = true;
            len = 2; // Update length of longest
palindromic substring
        } else {
            dp[i][i + 1] = false;
        }
    }

    // Check for substrings of length > 2
    for (int g = 2; g < n; g++) {
        for (int i = 0, j = g; j < n; i++, j++) {
            if (str[i] == str[j] && dp[i + 1][j - 1]) {
                dp[i][j] = true;
                len = g + 1; // Update length of longest
palindromic substring
            } else {
                dp[i][j] = false;
            }
        }
    }

    return len;
}

int main() {
    string str = "abccbc";
    int longestPalSubstrLen =
LongestPalindromicSubstring(str);
    cout << longestPalSubstrLen << endl;

    return 0;
}
```

Step-by-Step Dry Run

Step 1: Initialize DP Table ($g = 0$)

Each **single character** is a palindrome ($dp[i][i] = \text{true}$).

	a	b	c	c	b	c
a	✓					
b		✓				
c			✓			
c				✓		
b					✓	
c						✓

Longest palindrome so far: $\text{len} = 1$ (since all single characters are palindromes).

Step 2: Substrings of Length 2 ($g = 1$)

We check adjacent characters $\text{str}[i] == \text{str}[i+1]$.

	a	b	c	c	b	c
a	✓	✗				
b		✓	✗			
c			✓	✓		
c				✓	✗	
b					✓	✗
c						✓

Updated longest palindrome: $\text{len} = 2$ ("cc" at $dp[2][3]$).

Step 3: Substrings of Length 3+ ($g \geq 2$)

For substrings of length $g + 1$, we check:

$$dp[i][j] = (\text{str}[i] == \text{str}[j]) \text{ AND } dp[i+1][j-1]$$

For $g = 2$ (substrings of length 3):

	a	b	c	c	b	c
a	✓	✗	✗			
b		✓	✗	✗	✓	
c			✓	✓	✗	✗
c				✓	✗	✗
b					✓	✗
c						✓

Updated longest palindrome: len = 3 ("bccb" at $dp[1][4]$).

For $g = 3$ (substrings of length 4):

	a	b	c	c	b	c
a	✓	✗	✗	✗		
b		✓	✗	✗	✓	✗
c			✓	✓	✗	✗
c				✓	✗	✗
b					✓	✗
c						✓

Updated longest palindrome: len = 4 ("bccb")

at $dp[1][4]$).

For $g = 4$ (substrings of length 5):

	a	b	c	c	b	c
a	✓	✗	✗	✗	✗	
b		✓	✗	✗	✓	✗
c			✓	✓	✗	✗
c				✓	✗	✗
b					✓	✗
c						✓

No update to len (remains 4).

For $g = 5$ (full string, length 6):

	a	b	c	c	b	c
a	✓	✗	✗	✗	✗	✗
b		✓	✗	✗	✓	✗
c			✓	✓	✗	✗
c				✓	✗	✗
b					✓	✗
c						✓

Final longest palindrome: len = 4 ("bccb").

Final Answer

The longest palindromic substring in "abccbc"

	has length 4 ("bccb").
	Output: 4
Output:- 4	

Max Sum Increasing subseq In C++

```
#include <iostream>
#include <climits>
using namespace std;

int MaxSumIncreasingSubseq(int arr[], int size) {
    int omax = INT_MIN;
    int* dp = new int[size];
    //int dp[size];

    for (int i = 0; i < size; i++) {
        int maxSum = arr[i];
        for (int j = 0; j < i; j++) {
            if (arr[j] <= arr[i]) {
                maxSum = max(maxSum, dp[j] +
arr[i]);
            }
        }
        dp[i] = maxSum;
        omax = max(omax, dp[i]);
    }

    delete[] dp; // Don't forget to free the allocated
memory
    return omax;
}

int main() {
    int arr[] = {10, 22, 9, 33, 21, 50, 41, 60, 80, 3};
    int size = sizeof(arr) / sizeof(arr[0]);

    int maxSum = MaxSumIncreasingSubseq(arr,
size);
    cout << maxSum << endl;

    return 0;
}
```

arr = {10, 22, 9, 33, 21, 50, 41, 60, 80, 3}

Step-by-Step Dry Run (Table Format)

Index (i)	arr[i]	Initial dp[i]	Comparisons (j < i, arr[j] ≤ arr[i])	Updated dp[i]
0	10	10	-	10
1	22	22	j=0 (10 ≤ 22) → dp[1] = max(22, 10+22)	32
2	9	9	-	9
3	33	33	j=0 (10 ≤ 33) → dp[3] = max(33, 10+33) j=1 (22 ≤ 33) → dp[3] = max(43, 32+33)	65
4	21	21	j=0 (10 ≤ 21) → dp[4] = max(21, 10+21)	31
5	50	50	j=0 (10 ≤ 50) → dp[5] = max(50, 10+50) j=1 (22 ≤ 50) → dp[5] = max(60, 32+50) j=3 (33 ≤ 50) → dp[5] = max(100, 65+50)	100
6	41	41	j=0 (10 ≤ 41) → dp[6] = max(41, 10+41) j=1 (22 ≤ 41) → dp[6] = max(51, 32+41) j=3 (33 ≤ 41) → dp[6] = max(91, 65+41)	91
7	60	60	j=0 (10 ≤ 60) → dp[7] = max(60, 10+60) j=1 (22 ≤ 60) → dp[7] = max(70, 32+60) j=3 (33 ≤ 60) → dp[7] = max(110, 65+60) j=5 (50 ≤ 60) → dp[7] = max(150, 100+60)	150

			j=0,1,3,5,6,7 (comparing all increasing values) → dp[8] = max(10+80, 32+80, 65+80, 100+80, 91+80, 150+80)	
8	80	80		255

9	3	3	-	3
---	---	---	---	---

Final DP Table

Index (i)	arr[i]	dp[i] (Max Sum IS Ending at i)
0	10	10
1	22	32
2	9	9
3	33	65
4	21	31
5	50	100
6	41	91
7	60	150
8	80	255
9	3	3

Final Answer

Output: 255

Summary:

- The **largest increasing subsequence** contributing to **255** is:

10 → 22 → 33 → 50 → 60 → 80

- Sum = 10 + 22 + 33 + 50 + 60 + 80 = 255

Output:-

255

{10, 22, 33, 50, 60, 80} → sum = 10 + 22 + 33 + 50 + 60 + 80 = 255

Min Cost to make strings identical C++

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

int minCostToMakeIdentical(string s1, string s2, int c1, int c2) {
    int m = s1.length();
    int n = s2.length();

    // Initialize dp array with size (m+1)x(n+1)
    vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));

    // Fill dp array
    for (int i = m - 1; i >= 0; i--) {
        for (int j = n - 1; j >= 0; j--) {
            if (s1[i] == s2[j]) {
                dp[i][j] = 1 + dp[i + 1][j + 1];
            } else {
                dp[i][j] = max(dp[i + 1][j], dp[i][j + 1]);
            }
        }
    }

    // Calculate length of LCS
    int lcsLength = dp[0][0];
    cout << "Length of Longest Common Subsequence: "
    << lcsLength << endl;

    // Calculate remaining characters in s1 and s2 after
    // LCS
    int s1Remaining = m - lcsLength;
    int s2Remaining = n - lcsLength;

    // Calculate minimum cost to make strings identical
    int cost = s1Remaining * c1 + s2Remaining * c2;
    return cost;
}

int main() {
    string s1 = "cat";
    string s2 = "cut";
    int c1 = 1;
    int c2 = 1;

    int minCost = minCostToMakeIdentical(s1, s2, c1, c2);
    cout << "Minimum cost to make strings identical: "
    << minCost << endl;

    return 0;
}
```

Step-by-Step DP Table Construction

Strings:

s1 = "cat"
s2 = "cut"

We create a **(m+1) × (n+1) DP table**, where:

- dp[i][j] stores the **length of LCS of s1[i:] and s2[j:]**.

DP Table Initialization (Bottom-Up)

i\j	c	u	t	(empty)
c	?	?	?	0
a	?	?	?	0
t	?	?	?	0
(empty)	0	0	0	0

Filling the Table

We start from the **bottom-right** and move **backwards**.

1. Comparing 't' in s1 with 't' in s2:

s1[2] == s2[2] ('t' == 't')

- So, dp[2][2] = 1 + dp[3][3] = 1

2. Comparing 't' in s1 with 'u' in s2:

s1[2] != s2[1] ('t' ≠ 'u')

- So, dp[2][1] = max(dp[3][1], dp[2][2]) = max(0, 1) = 1

3. Comparing 't' in s1 with 'c' in s2:

s1[2] != s2[0] ('t' ≠ 'c')

- So, dp[2][0] = max(dp[3][0], dp[2][1]) = max(0, 1) = 1

4. Comparing 'a' in s1 with 't' in s2:

s1[1] != s2[2] ('a' ≠ 't')

- So, dp[1][2] = max(dp[2][2], dp[1][3]) = max(1, 0) = 1

5. Comparing 'a' in s1 with 'u' in s2:

s1[1] != s2[1] ('a' ≠ 'u')

- So, $dp[1][1] = \max(dp[2][1], dp[1][2]) = \max(1, 1) = 1$

6. Comparing 'a' in s1 with 'c' in s2:

$s1[1] \neq s2[0]$ ('a' ≠ 'c')

- So, $dp[1][0] = \max(dp[2][0], dp[1][1]) = \max(1, 1) = 1$

7. Comparing 'c' in s1 with 't' in s2:

$s1[0] \neq s2[2]$ ('c' ≠ 't')

- So, $dp[0][2] = \max(dp[1][2], dp[0][3]) = \max(1, 0) = 1$

8. Comparing 'c' in s1 with 'u' in s2:

$s1[0] \neq s2[1]$ ('c' ≠ 'u')

- So, $dp[0][1] = \max(dp[1][1], dp[0][2]) = \max(1, 1) = 1$

9. Comparing 'c' in s1 with 'c' in s2:

$s1[0] == s2[0]$ ('c' == 'c')

- So, $dp[0][0] = 1 + dp[1][1] = 2$

Final DP Table

i\j	c	u	t	(empty)
c	2	1	1	0
a	1	1	1	0
t	1	1	1	0
(empty)	0	0	0	0

Final Calculation

- **LCS Length** = $dp[0][0] = 2$
- **Remaining characters to delete:**

s1: "cat" → Remove 1 character ('a')
 s2: "cut" → Remove 1 character ('u')

- **Total Cost:**

$$\text{Cost} = (1 \times 1) + (1 \times 1) = 1 + 1 = 2$$

Output:-

Length of Longest Common Subsequence: 2

Minimum cost to make strings identical: 2

Optimal strategy for a game In C++

```
#include <iostream>
#include <algorithm>

using namespace std;

int main() {
    int arr[] = {20, 30, 2, 10};
    int n = sizeof(arr) / sizeof(arr[0]);

    int dp[n][n]; // Create a 2D array of size n x n

    for (int g = 0; g < n; g++) {
        for (int i = 0, j = g; j < n; i++, j++) {
            if (g == 0) {
                dp[i][j] = arr[i];
            } else if (g == 1) {
                dp[i][j] = max(arr[i], arr[j]);
            } else {
                int val1 = arr[i] + min((i + 2 <= j ? dp[i + 2][j] : 0), (i + 1 <= j - 1 ? dp[i + 1][j - 1] : 0));
                int val2 = arr[j] + min((i + 1 <= j - 1 ? dp[i + 1][j - 1] : 0), (i <= j - 2 ? dp[i][j - 2] : 0));
                dp[i][j] = max(val1, val2);
            }
        }
    }

    cout << dp[0][n - 1] << endl; // Print the maximum value that can be collected
}

return 0;
}
```

Step-by-Step Dry Run with Table

Initialization

Given input:

```
int arr[] = {20, 30, 2, 10};
```

Size of arr:

$n = 4$;

A **2D DP table ($dp[i][j]$)** is used, where $dp[i][j]$ represents the **maximum score the first player can collect from $arr[i]$ to $arr[j]$** .

Step 1: Fill Diagonal ($g = 0$)

When $i == j$, only one element is available, so:

i	j	dp[i][j]
0	0	20
1	1	30
2	2	2
3	3	10

Step 2: Fill $g = 1$ (Two Elements)

When $g = 1$, two elements are available, so the first player picks the maximum:

i	j	Computation	dp[i][j]
0	1	max(20, 30)	30
1	2	max(30, 2)	30

i	j	Computation	dp[i][j]
2	3	max(2, 10)	10

Step 3: Fill g = 2 (Three Elements)

Now, we consider **three elements** and the optimal choices:

i	j	Computation	dp[i][j]
0	2	max(20 + min(2, 30), 2 + min(30, 20)) → max(20+2, 2+20) = 22	22
1	3	max(30 + min(10, 2), 10 + min(2, 30)) → max(30+2, 10+2) = 32	32

Step 4: Fill g = 3 (Entire Array)

i\j	0	1	2	3
0	20	30	22	40
1		30	30	32
2			2	10
3				10

Final Output:

40

Output:-
40


```

vector<int>(cap + 1, 0));

for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= cap; j++) {
        dp[i][j] = dp[i - 1][j];

        if (j >= wts[i - 1]) {
            dp[i][j] = max(dp[i][j], dp[i - 1][j - wts[i - 1]] + vals[i - 1]);
        }
    }
}

int ans = dp[n][cap];
cout << "Maximum value: " << ans <<
endl;

deque<Pair> que;
que.push_back(Pair(n, cap, ""));

printPaths(dp, vals, wts, n, cap, "", que);
}

int main() {
    vector<int> vals = {15, 14, 10, 45, 30};
    vector<int> wts = {2, 5, 1, 3, 4};
    int cap = 7;

    knapsackPaths(vals, wts, cap);

    return 0;
}

```

3 (val=10, wt=1)	0	10	15	25	25	25	25	25	
4 (val=45, wt=3)	0	10	15	45	55	60	70	70	
5 (val=30, wt=4)	0	10	15	45	55	60	70	75	

The **maximum value** obtained is 75 at dp[5][7].

Step 2: Print All Paths

Using **backtracking**, the function printPaths reconstructs paths that lead to dp[n][cap] = 75.

Backtracking Paths

1. Start at dp[5][7] = 75
 - o dp[4][3] = 45 → Item 5 (index 4, value 30, weight 4) is included.
2. Now at dp[4][3] = 45
 - o dp[3][0] = 0 → Item 4 (index 3, value 45, weight 3) is included.

Thus, one of the optimal selections is {30, 45}.

Final Output

Maximum value: 75

4 3

Output:-	
Maximum value: 75	
3 4	

Perfect Square In C++

```
#include <iostream>
#include <vector>
#include <climits>
#include <cmath>
using namespace std;

int main() {
    vector<int> arr = {0, 1, 2, 3, 1, 2, 3, 4, 2,
1, 2, 3};
    int n = arr.size();
    vector<int> dp(n + 1, INT_MAX); // dp
array where dp[i] represents the minimum
number of perfect squares summing up to i
    //int dp[n+1]={INT_MAX};
    dp[0] = 0; // Base case: 0 requires 0
squares
    dp[1] = 1; // 1 requires 1 square (1)

    for (int i = 2; i <= n; i++) {
        for (int j = 1; j * j <= i; j++) {
            dp[i] = min(dp[i], dp[i - j * j] + 1);
        }
    }

    // Output the dp array
    for (int i = 0; i <= n; i++) {
        cout << dp[i] << " ";
    }
    cout << endl;

    return 0;
}
```

Dry Run with Table

We compute $dp[i]$ for $i = 0$ to 12 using the given transition formula.

i	Perf ect Squ ares (≤ i)	dp[i] Computation	dp[i]
0	-	$dp[0] = 0$	0
1	1	$dp[1] = \min(dp[1 - 1] + 1) = 1$	1
2	1	$dp[2] = \min(dp[2 - 1] + 1) = 2$	2
3	1	$dp[3] = \min(dp[3 - 1] + 1) = 3$	3
4	1, 4	$dp[4] = \min(dp[4 - 1] + 1, dp[4 - 4] + 1) = \min(4, 1) = 1$	1
5	1, 4	$dp[5] = \min(dp[5 - 1] + 1, dp[5 - 4] + 1) = \min(2, 2) = 2$	2
6	1, 4	$dp[6] = \min(dp[6 - 1] + 1, dp[6 - 4] + 1) = \min(3, 3) = 3$	3
7	1, 4	$dp[7] = \min(dp[7 - 1] + 1, dp[7 - 4] + 1) = \min(4, 4) = 4$	4
8	1, 4	$dp[8] = \min(dp[8 - 1] + 1, dp[8 - 4] + 1) = \min(5, 2) = 2$	2
9	1, 4, 9	$dp[9] = \min(dp[9 - 1] + 1, dp[9 - 4] + 1, dp[9 - 9] + 1) = \min(3, 3, 1) = 1$	1
10	1, 4, 9	$dp[10] = \min(dp[10 - 1] + 1, dp[10 - 4] + 1, dp[10 - 9] + 1) = \min(2, 4, 2) = 2$	2
11	1, 4, 9	$dp[11] = \min(dp[11 - 1] + 1, dp[11 - 4] + 1, dp[11 - 9] + 1) = \min(3, 5, 3) = 3$	3
12	1, 4, 9	$dp[12] = \min(dp[12 - 1] + 1, dp[12 - 4] + 1, dp[12 - 9] + 1) = \min(4, 3, 4)$	3

		= 3	
--	--	-----	--

Final Output (dp Array)

The DP array will be:

0 1 2 3 1 2 3 4 2 1 2 3 3

Output:-

0 1 2 3 1 2 3 4 2 1 2 3 3

Print all LIS In C++

```

#include <iostream>
#include <vector>
#include <deque>
using namespace std;

struct Pair {
    int l; // length of the LIS
    int i; // index in the array
    int v; // value at index i in the array
    string psf; // path so far

    Pair(int l, int i, int v, string psf) {
        this->l = l;
        this->i = i;
        this->v = v;
        this->psf = psf;
    }
};

void printAllLIS(vector<int>& arr) {
    int n = arr.size();
    vector<int> dp(n, 1); // dp array to store
    the length of LIS ending at each index
    int omax = 0; // maximum length of LIS
    found
    int omi = 0; // index where the LIS with
    maximum length ends

    // Finding the length of LIS ending at
    each index
    for (int i = 0; i < n; i++) {
        int maxLen = 0;
        for (int j = 0; j < i; j++) {
            if (arr[i] > arr[j]) {
                if (dp[j] > maxLen) {
                    maxLen = dp[j];
                }
            }
        }
        dp[i] = maxLen + 1;

        if (dp[i] > omax) {
            omax = dp[i];
            omi = i;
        }
    }

    deque<Pair> q;
    q.push_back(Pair(omax, omi, arr[omi],
    to_string(arr[omi])));

    while (!q.empty()) {

```

Dry Run Example

Input:

vector<int> arr = {10, 22, 9, 33, 21, 50, 41, 60, 80, 3};

Step 1: Compute dp Array

Index i	arr[i]	LIS Length (dp[i])	Previous LIS Contributor (dp[j])
0	10	1	-
1	22	2	10 (dp[0] + 1)
2	9	1	-
3	33	3	22 (dp[1] + 1)
4	21	2	10 (dp[0] + 1)
5	50	4	33 (dp[3] + 1)
6	41	4	33 (dp[3] + 1)

```

Pair rem = q.front();
q.pop_front();

if (rem.l == 1) {
    cout << rem.psf << endl; // print the
path when the length of LIS is 1
} else {
    for (int j = rem.i - 1; j >= 0; j--) {
        if (dp[j] == rem.l - 1 && arr[j] <=
rem.v) {
            q.push_back(Pair(dp[j], j,
arr[j], to_string(arr[j]) + " -> " + rem.psf));
        }
    }
}
}

int main() {
    vector<int> arr = {10, 22, 9, 33, 21, 50,
41, 60, 80, 3};
    printAllLIS(arr);
    return 0;
}

```

7	60	5 (Max LIS)	50 (dp[5] + 1)
8	80	6 (Max LIS)	60 (dp[7] + 1)
9	3	1	-

Step 2: Print All LIS Paths

The **longest increasing subsequence has length 6** and ends at 80.

Backtracking from 80, possible LIS paths:

10 -> 22 -> 33 -> 50 -> 60 -> 80
10 -> 22 -> 33 -> 41 -> 60 -> 80

Output:-

10 -> 22 -> 33 -> 41 -> 60 -> 80
10 -> 22 -> 33 -> 50 -> 60 -> 80

Print all path with max gold In C++																																																																																													
<pre>#include <iostream> #include <vector> #include <queue> using namespace std; struct Pair { int i, j; string psf; Pair(int i, int j, string psf) { this->i = i; this->j = j; this->psf = psf; } }; void printMaxGoldPath(vector<vector<int>>& arr) { int m = arr.size(); int n = arr[0].size(); // dp array to store maximum gold // collected to reach each cell vector<vector<int>> dp(m, vector<int>(n, 0)); // Initialize dp array for the last column for (int i = 0; i < m; i++) { dp[i][n - 1] = arr[i][n - 1]; } // Fill dp array using dynamic // programming approach for (int j = n - 2; j >= 0; j--) { for (int i = 0; i < m; i++) { int maxGold = dp[i][j + 1]; // Maximum gold by going right from current cell if (i > 0) { maxGold = max(maxGold, dp[i - 1][j + 1]); // Maximum gold by going diagonal-up-right } if (i < m - 1) { maxGold = max(maxGold, dp[i + 1][j + 1]); // Maximum gold by going diagonal-down-right } dp[i][j] = arr[i][j] + maxGold; // Total gold collected to reach current cell } } }</pre>	<p>Given Input Matrix (arr):</p> <table style="margin-left: auto; margin-right: auto;"> <tr><td>3</td><td>2</td><td>3</td><td>1</td></tr> <tr><td>2</td><td>4</td><td>6</td><td>0</td></tr> <tr><td>5</td><td>0</td><td>1</td><td>3</td></tr> <tr><td>9</td><td>1</td><td>5</td><td>1</td></tr> </table> <p>Step 1: Initialize dp Table</p> <ul style="list-style-type: none"> • Copy the last column ($j = 3$) from arr to dp: <table style="margin-left: auto; margin-right: auto;"> <tr><td>0</td><td>0</td><td>0</td><td>1</td></tr> <tr><td>0</td><td>0</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>0</td><td>0</td><td>3</td></tr> <tr><td>0</td><td>0</td><td>0</td><td>1</td></tr> </table> <p>Step 2: Fill dp Table from Right to Left</p> <p>Column 2 ($j = 2$)</p> <p>Each $dp[i][j] = arr[i][j] + \max(dp[i][j+1], dp[i-1][j+1], dp[i+1][j+1])$</p> <table style="margin-left: auto; margin-right: auto;"> <tr><td>0</td><td>0</td><td>4</td><td>1</td><td>→ 3 + max(1) = 4</td></tr> <tr><td>0</td><td>0</td><td>9</td><td>0</td><td>→ 6 + max(3,0) = 9</td></tr> <tr><td>0</td><td>0</td><td>6</td><td>3</td><td>→ 1 + max(3,1) = 6</td></tr> <tr><td>0</td><td>0</td><td>8</td><td>1</td><td>→ 5 + max(3) = 8</td></tr> </table> <p>Column 1 ($j = 1$)</p> <table style="margin-left: auto; margin-right: auto;"> <tr><td>0</td><td>11</td><td>4</td><td>1</td><td>→ 2 + max(4,9) = 11</td></tr> <tr><td>0</td><td>13</td><td>9</td><td>0</td><td>→ 4 + max(9,6) = 13</td></tr> <tr><td>0</td><td>9</td><td>6</td><td>3</td><td>→ 0 + max(6,8) = 9</td></tr> <tr><td>0</td><td>14</td><td>8</td><td>1</td><td>→ 1 + max(8) = 14</td></tr> </table> <p>Column 0 ($j = 0$)</p> <table style="margin-left: auto; margin-right: auto;"> <tr><td>13</td><td>11</td><td>4</td><td>1</td><td>→ 3 + max(11,13) = 13</td></tr> <tr><td>15</td><td>13</td><td>9</td><td>0</td><td>→ 2 + max(13,9) = 15</td></tr> <tr><td>18</td><td>9</td><td>6</td><td>3</td><td>→ 5 + max(9,14) = 18 ↘ (Expected max value)</td></tr> <tr><td>23</td><td>14</td><td>8</td><td>1</td><td>→ 9 + max(14) = 23</td></tr> </table> <p>Step 3: Find Maximum Gold in Column 0</p>	3	2	3	1	2	4	6	0	5	0	1	3	9	1	5	1	0	0	0	1	0	0	0	0	0	0	0	3	0	0	0	1	0	0	4	1	→ 3 + max(1) = 4	0	0	9	0	→ 6 + max(3,0) = 9	0	0	6	3	→ 1 + max(3,1) = 6	0	0	8	1	→ 5 + max(3) = 8	0	11	4	1	→ 2 + max(4,9) = 11	0	13	9	0	→ 4 + max(9,6) = 13	0	9	6	3	→ 0 + max(6,8) = 9	0	14	8	1	→ 1 + max(8) = 14	13	11	4	1	→ 3 + max(11,13) = 13	15	13	9	0	→ 2 + max(13,9) = 15	18	9	6	3	→ 5 + max(9,14) = 18 ↘ (Expected max value)	23	14	8	1	→ 9 + max(14) = 23
3	2	3	1																																																																																										
2	4	6	0																																																																																										
5	0	1	3																																																																																										
9	1	5	1																																																																																										
0	0	0	1																																																																																										
0	0	0	0																																																																																										
0	0	0	3																																																																																										
0	0	0	1																																																																																										
0	0	4	1	→ 3 + max(1) = 4																																																																																									
0	0	9	0	→ 6 + max(3,0) = 9																																																																																									
0	0	6	3	→ 1 + max(3,1) = 6																																																																																									
0	0	8	1	→ 5 + max(3) = 8																																																																																									
0	11	4	1	→ 2 + max(4,9) = 11																																																																																									
0	13	9	0	→ 4 + max(9,6) = 13																																																																																									
0	9	6	3	→ 0 + max(6,8) = 9																																																																																									
0	14	8	1	→ 1 + max(8) = 14																																																																																									
13	11	4	1	→ 3 + max(11,13) = 13																																																																																									
15	13	9	0	→ 2 + max(13,9) = 15																																																																																									
18	9	6	3	→ 5 + max(9,14) = 18 ↘ (Expected max value)																																																																																									
23	14	8	1	→ 9 + max(14) = 23																																																																																									

```

}

// Find the maximum gold collected in
// the first column
int maxGold = dp[0][0];
int maxRow = 0;
for (int i = 1; i < m; i++) {
    if (dp[i][0] > maxGold) {
        maxGold = dp[i][0];
        maxRow = i;
    }
}

// Print the maximum gold collected
cout << maxGold << endl;

// Queue to perform BFS for path tracing
queue<Pair> q;
q.push(Pair(maxRow, 0,
to_string(maxRow))); // Start from the cell
with maximum gold in the first column

// BFS to print all paths with maximum
gold collected
while (!q.empty()) {
    Pair rem = q.front();
    q.pop();

    if (rem.j == n - 1) {
        cout << rem.psf << endl; // Print
path when reaching the last column
    } else {
        int currentGold = dp[rem.i][rem.j];
        int rightGold = dp[rem.i][rem.j + 1];
        int diagonalUpGold = (rem.i > 0) ?
dp[rem.i - 1][rem.j + 1] : -1;
        int diagonalDownGold = (rem.i < m
- 1) ? dp[rem.i + 1][rem.j + 1] : -1;

        // Add paths to queue based on the
direction with maximum gold
        if (rightGold == currentGold -
arr[rem.i][rem.j + 1]) {
            q.push(Pair(rem.i, rem.j + 1,
rem.psf + " H")); // Move horizontally to the
right
        }
        if (diagonalUpGold == currentGold
- arr[rem.i - 1][rem.j + 1]) {
            q.push(Pair(rem.i - 1, rem.j + 1,
rem.psf + " LU")); // Move diagonally up-
right
        }
    }
}

```

- The **maximum gold collected is 18 at row 2.**

Step 4: Find All Paths (Using BFS)

Starting from $dp[2][0] = 18$:

- $dp[2][1] = 9$
- $dp[3][1] = 14$
- $dp[3][2] = 8$
- $dp[3][3] = 1$

Valid Path:

$2 \rightarrow LD \rightarrow 3 \rightarrow LU \rightarrow 3 \rightarrow H \rightarrow 1$

Final Output

Maximum Gold: 18
Path: 2 LD 3 LU 3 H 1

```
        if (diagonalDownGold ==  
currentGold - arr[rem.i + 1][rem.j + 1]) {  
            q.push(Pair(rem.i + 1, rem.j + 1,  
rem.psf + " LD")); // Move diagonally down-  
right  
        }  
    }  
}  
  
int main() {  
    vector<vector<int>> arr = {  
        {3, 2, 3, 1},  
        {2, 4, 6, 0},  
        {5, 0, 1, 3},  
        {9, 1, 5, 1}  
    };  
  
    printMaxGoldPath(arr);  
  
    return 0;  
}
```

Output:-

18

Print all path with minimum Cost In C++

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

struct Pair {
    string psf; // path so far
    int i; // current row index
    int j; // current column index

    Pair(string psf, int i, int j) {
        this->psf = psf;
        this->i = i;
        this->j = j;
    }
};

void printAllPaths(vector<vector<int>>& arr) {
    int m = arr.size();
    int n = arr[0].size();

    // dp array to store minimum cost to
    // reach each cell
    vector<vector<int>> dp(m,
    vector<int>(n, 0));

    // Initialize dp table
    dp[m-1][n-1] = arr[m-1][n-1];
    for (int i = m - 2; i >= 0; i--) {
        dp[i][n-1] = arr[i][n-1] + dp[i + 1][n -
1];
    }
    for (int j = n - 2; j >= 0; j--) {
        dp[m-1][j] = arr[m-1][j] + dp[m - 1][j +
1];
    }
    for (int i = m - 2; i >= 0; i--) {
        for (int j = n - 2; j >= 0; j--) {
            dp[i][j] = arr[i][j] + min(dp[i][j + 1],
dp[i + 1][j]);
        }
    }

    // Minimum cost to reach the top-left
    // corner
    cout << dp[0][0] << endl;

    // Queue to perform BFS
    queue<Pair> q;
    q.push(Pair("", 0, 0));
}
```

Dry Run of Minimum Cost Path Problem

We will compute the **dynamic programming (DP)** **table** step-by-step to ensure that we get the minimum cost sum **46** for the given matrix.

Given Input Matrix (arr):

```
{1, 2, 3, 4},
{5, 6, 7, 8},
{9, 10, 11, 12},
{13, 14, 15, 16}
```

Step 1: Understanding the DP Approach

1. **Base Case:** The last cell ($dp[3][3]$) is the same as $arr[3][3] = 16$.
2. **Filling Last Row (Right to Left):**
 - o $dp[i][j] = arr[i][j] + dp[i][j+1]$
3. **Filling Last Column (Bottom to Top):**
 - o $dp[i][j] = arr[i][j] + dp[i+1][j]$
4. **Filling the Rest (Bottom-Up, Right-to-Left):**
 - o $dp[i][j] = arr[i][j] + \min(dp[i+1][j], dp[i][j+1])$

Step 2: Construct DP Table Step-by-Step

1. Initialize $dp[3][3]$ (Bottom-Right Cell)

$$dp[3][3] = arr[3][3] = 16$$

2. Fill the Last Row (Right to Left)

$$\begin{aligned} dp[i][j] &= arr[i][j] + dp[i][j+1] \\ dp[3][j] &= arr[3][j] + dp[3][j+1] \end{aligned}$$

$i=3$	$j=3$	$j=2$ $(15+16)$	$j=1$ $(14+31)$	$j=0$ $(13+45)$

```

while (!q.empty()) {
    Pair rem = q.front();
    q.pop();

    if (rem.i == m - 1 && rem.j == n - 1) {
        cout << rem.psf << endl; // print
        path when reaching the bottom-right
        corner
    } else if (rem.i == m - 1) {
        q.push(Pair(rem.psf + "H", rem.i,
        rem.j + 1)); // go right
    } else if (rem.j == n - 1) {
        q.push(Pair(rem.psf + "V", rem.i +
        1, rem.j)); // go down
    } else {
        if (dp[rem.i][rem.j + 1] < dp[rem.i + 1][rem.j]) {
            q.push(Pair(rem.psf + "H", rem.i,
            rem.j + 1)); // go right
        } else if (dp[rem.i][rem.j + 1] >
        dp[rem.i + 1][rem.j]) {
            q.push(Pair(rem.psf + "V", rem.i +
            1, rem.j)); // go down
        } else {
            q.push(Pair(rem.psf + "V", rem.i +
            1, rem.j)); // go down
            q.push(Pair(rem.psf + "H", rem.i,
            rem.j + 1)); // go right
        }
    }
}

int main() {
    vector<vector<int>> arr = {
        {1, 2, 3, 4},
        {5, 6, 7, 8},
        {9, 10, 11, 12},
        {13, 14, 15, 16}
    };

    printAllPaths(arr);
    return 0;
}

```

arr	16	15	14	13
dp	16	31	45	58

3. Fill the Last Column (Bottom to Top)

$$dp[i][j] = arr[i][j] + dp[i+1][j]$$

$$dp[i][j] = arr[i][j] + dp[i+1][j]$$

i=2	j=3 (12+16)	j=2	j=1	j=0
arr	12	11	10	9
dp	28	-	-	-
i=1	j=3 (8+28)	j=2	j=1	j=0
arr	8	7	6	5
dp	36	-	-	-
i=0	j=3 (4+36)	j=2	j=1	j=0
arr	4	3	2	1
dp	40	-	-	-

4. Fill the Rest of the DP Table

$$dp[i][j] = arr[i][j] + \min(dp[i+1][j], dp[i][j+1])$$

$$dp[i][j] = arr[i][j] + \min(dp[i+1][j], dp[i][j+1])$$

i=2	j=2 (11+min(31, 28))	j=1 (10+min(41, 38))	j=0 (9+min(45, 40))
arr	11	10	9
dp	39	38	40
i=1	j=2 (7+min(39, 36))	j=1 (6+min(38, 44))	j=0 (5+min(45, 43))
arr	7	6	5
dp	43	44	45

$i=0$	$j=2$ $(3+\min(43, 40))$	$j=1$ $(2+\min(41, 44))$	$j=0$ $(1+\min(45, 43))$
arr	3	2	1
dp	43	45	46

Final DP Table

46	45	43	40
45	44	43	36
40	38	39	28
58	45	31	16

✓ Minimum Cost Path Sum = 46 (Matches G++ Output)

Step 3: Extracting All Paths

Now, we use BFS (`queue<Pair>`) to trace all paths from $(0, 0)$ to $(3, 3)$ following the minimum cost. The paths may vary but should sum up to 46.

1. **Move Right H** if $dp[i][j+1]$ is smaller.
2. **Move Down V** if $dp[i+1][j]$ is smaller.
3. **If both are equal, try both paths (H and V).**

Possible Paths (psf values in BFS)

V V V H H H (Down-Down-Down-Right-Right-Right)

Output:-

46
HHVVVV

Rod cutting In C++																																																										
<pre>#include <iostream> #include <vector> #include <algorithm> using namespace std; int solution(vector<int>& prices) { vector<int> np(prices.size() + 1); for (int i = 0; i < prices.size(); i++) { np[i + 1] = prices[i]; } vector<int> dp(np.size()); dp[0] = 0; dp[1] = np[1]; for (int i = 2; i < dp.size(); i++) { dp[i] = np[i]; int li = 1; int ri = i - 1; while (li <= ri) { if (dp[li] + dp[ri] > dp[i]) { dp[i] = dp[li] + dp[ri]; } li++; ri--; } } return dp[dp.size() - 1]; } int main() { vector<int> prices = {1, 5, 8, 9, 10, 17, 17, 20}; cout << solution(prices) << endl; return 0; }</pre>	Dry Run (Tabular) <p>Given Prices</p> <table style="margin-left: auto; margin-right: auto;"> <tr> <td>Length:</td> <td>1</td> <td>2</td> <td>3</td> <td>4</td> <td>5</td> <td>6</td> <td>7</td> <td>8</td> </tr> <tr> <td>Prices:</td> <td>1</td> <td>5</td> <td>8</td> <td>9</td> <td>10</td> <td>17</td> <td>17</td> <td>20</td> </tr> </table> <p>DP Computation Table</p> <table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="text-align: center;">Rod Length (np[i])</th> <th style="text-align: center;">Price (np[i])</th> <th style="text-align: center;">Possible Cuts (li, ri)</th> <th style="text-align: center;">Best Revenue (dp[i])</th> </tr> </thead> <tbody> <tr> <td style="text-align: center;">1</td> <td style="text-align: center;">1</td> <td style="text-align: center;">(1)</td> <td style="text-align: center;">1</td> </tr> <tr> <td style="text-align: center;">2</td> <td style="text-align: center;">5</td> <td style="text-align: center;">$(1,1) \rightarrow 1+1=2$</td> <td style="text-align: center;">5</td> </tr> <tr> <td style="text-align: center;">3</td> <td style="text-align: center;">8</td> <td style="text-align: center;">$(1,2) \rightarrow 1+5=6, (2,1) \rightarrow 5+1=6$</td> <td style="text-align: center;">8</td> </tr> <tr> <td style="text-align: center;">4</td> <td style="text-align: center;">9</td> <td style="text-align: center;">$(1,3) \rightarrow 1+8=9, (2,2) \rightarrow 5+5=10$</td> <td style="text-align: center;">10</td> </tr> <tr> <td style="text-align: center;">5</td> <td style="text-align: center;">10</td> <td style="text-align: center;">$(1,4) \rightarrow 1+10=11, (2,3) \rightarrow 5+8=13$</td> <td style="text-align: center;">13</td> </tr> <tr> <td style="text-align: center;">6</td> <td style="text-align: center;">17</td> <td style="text-align: center;">$(1,5) \rightarrow 1+13=14, (2,4) \rightarrow 5+10=15, (3,3) \rightarrow 8+8=16$</td> <td style="text-align: center;">17</td> </tr> <tr> <td style="text-align: center;">7</td> <td style="text-align: center;">17</td> <td style="text-align: center;">$(1,6) \rightarrow 1+17=18, (2,5) \rightarrow 5+13=18, (3,4) \rightarrow 8+10=18$</td> <td style="text-align: center;">18</td> </tr> <tr> <td style="text-align: center;">8</td> <td style="text-align: center;">20</td> <td style="text-align: center;">$(1,7) \rightarrow 1+18=19, (2,6) \rightarrow 5+17=22, (3,5) \rightarrow 8+13=21, (4,4) \rightarrow 10+10=20$</td> <td style="text-align: center;">22</td> </tr> </tbody> </table> <p>Final answer</p> <p>The maximum revenue we can get for length = 8 is 22.</p>				Length:	1	2	3	4	5	6	7	8	Prices:	1	5	8	9	10	17	17	20	Rod Length (np[i])	Price (np[i])	Possible Cuts (li, ri)	Best Revenue (dp[i])	1	1	(1)	1	2	5	$(1,1) \rightarrow 1+1=2$	5	3	8	$(1,2) \rightarrow 1+5=6, (2,1) \rightarrow 5+1=6$	8	4	9	$(1,3) \rightarrow 1+8=9, (2,2) \rightarrow 5+5=10$	10	5	10	$(1,4) \rightarrow 1+10=11, (2,3) \rightarrow 5+8=13$	13	6	17	$(1,5) \rightarrow 1+13=14, (2,4) \rightarrow 5+10=15, (3,3) \rightarrow 8+8=16$	17	7	17	$(1,6) \rightarrow 1+17=18, (2,5) \rightarrow 5+13=18, (3,4) \rightarrow 8+10=18$	18	8	20	$(1,7) \rightarrow 1+18=19, (2,6) \rightarrow 5+17=22, (3,5) \rightarrow 8+13=21, (4,4) \rightarrow 10+10=20$	22
Length:	1	2	3	4	5	6	7	8																																																		
Prices:	1	5	8	9	10	17	17	20																																																		
Rod Length (np[i])	Price (np[i])	Possible Cuts (li, ri)	Best Revenue (dp[i])																																																							
1	1	(1)	1																																																							
2	5	$(1,1) \rightarrow 1+1=2$	5																																																							
3	8	$(1,2) \rightarrow 1+5=6, (2,1) \rightarrow 5+1=6$	8																																																							
4	9	$(1,3) \rightarrow 1+8=9, (2,2) \rightarrow 5+5=10$	10																																																							
5	10	$(1,4) \rightarrow 1+10=11, (2,3) \rightarrow 5+8=13$	13																																																							
6	17	$(1,5) \rightarrow 1+13=14, (2,4) \rightarrow 5+10=15, (3,3) \rightarrow 8+8=16$	17																																																							
7	17	$(1,6) \rightarrow 1+17=18, (2,5) \rightarrow 5+13=18, (3,4) \rightarrow 8+10=18$	18																																																							
8	20	$(1,7) \rightarrow 1+18=19, (2,6) \rightarrow 5+17=22, (3,5) \rightarrow 8+13=21, (4,4) \rightarrow 10+10=20$	22																																																							
Output:- 22																																																										

Temple offering In C++																																													
Dry Run (Tabular)																																													
<pre>#include <iostream> #include <algorithm> using namespace std; int totalOfferings(int* height, int n) { int* larr = new int[n]; // Left offerings array int* rarr = new int[n]; // Right offerings array // Calculate left offerings larr[0] = 1; for (int i = 1; i < n; i++) { if (height[i] > height[i - 1]) { larr[i] = larr[i - 1] + 1; } else { larr[i] = 1; } } // Calculate right offerings rarr[n - 1] = 1; for (int i = n - 2; i >= 0; i--) { if (height[i] > height[i + 1]) { rarr[i] = rarr[i + 1] + 1; } else { rarr[i] = 1; } } // Calculate total offerings int ans = 0; for (int i = 0; i < n; i++) { ans += max(larr[i], rarr[i]); } // Free allocated memory delete[] larr; delete[] rarr; return ans; } int main() { int height[] = {2, 3, 5, 6, 4, 8, 9}; int n = sizeof(height) / sizeof(height[0]); cout << totalOfferings(height, n) << endl; return 0; }</pre>	<p>Input:</p> <p>height[] = {2, 3, 5, 6, 4, 8, 9}</p> <table border="1"> <thead> <tr> <th>Index i</th> <th>Height height[i]</th> <th>Left Offerings larr[i]</th> <th>Right Offerings rarr[i]</th> <th>Final Offerings max(lar r[i], rarr[i])</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>2</td> <td>1</td> <td>1</td> <td>1</td> </tr> <tr> <td>1</td> <td>3</td> <td>2</td> <td>1</td> <td>2</td> </tr> <tr> <td>2</td> <td>5</td> <td>3</td> <td>1</td> <td>3</td> </tr> <tr> <td>3</td> <td>6</td> <td>4</td> <td>2</td> <td>4</td> </tr> <tr> <td>4</td> <td>4</td> <td>1</td> <td>1</td> <td>1</td> </tr> <tr> <td>5</td> <td>8</td> <td>2</td> <td>2</td> <td>2</td> </tr> <tr> <td>6</td> <td>9</td> <td>3</td> <td>3</td> <td>3</td> </tr> </tbody> </table> <p>Total Offerings:</p> <p>1 + 2 + 3 + 4 + 1 + 2 + 3 = 16</p> <p>✓ Output:</p> <p>16</p>					Index i	Height height[i]	Left Offerings larr[i]	Right Offerings rarr[i]	Final Offerings max(lar r[i], rarr[i])	0	2	1	1	1	1	3	2	1	2	2	5	3	1	3	3	6	4	2	4	4	4	1	1	1	5	8	2	2	2	6	9	3	3	3
Index i	Height height[i]	Left Offerings larr[i]	Right Offerings rarr[i]	Final Offerings max(lar r[i], rarr[i])																																									
0	2	1	1	1																																									
1	3	2	1	2																																									
2	5	3	1	3																																									
3	6	4	2	4																																									
4	4	1	1	1																																									
5	8	2	2	2																																									
6	9	3	3	3																																									
<p>Output:-</p> <p>16</p>																																													

Word Break In C++

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

bool solution(string sentence,
unordered_set<string>& dict) {
    int n = sentence.length();
    vector<int> dp(n, 0);

    for (int i = 0; i < n; i++) {
        for (int j = 0; j <= i; j++) {
            string word = sentence.substr(j, i - j
+ 1);
            if (dict.find(word) != dict.end()) {
                if (j > 0) {
                    dp[i] += dp[j - 1];
                } else {
                    dp[i] += 1;
                }
            }
        }
    }

    cout << dp[n - 1] << endl;
    return dp[n - 1] > 0;
}

int main() {
    unordered_set<string> dict = {"i", "like",
"pep", "coding", "pepper", "eating",
"mango", "man", "go", "in", "pepcoding"};
    string sentence =
"ilikepeppereatingmangoinpepcoding";

    cout << boolalpha << solution(sentence,
dict) << endl;
    return 0;
}
```

Iterative Tabular Dry Run for Word Break Problem

We will dry-run the **fixed DP approach** using the sentence: **"ilikepeppereatingmangoinpepcoding"**

Dictionary:

```
{"i", "like", "pep", "coding", "pepper",
"eating", "mango", "man", "go", "in",
"pepcoding"}
```

Step 1: Define DP Table

- Let $dp[i]$ represent whether the substring $s[0...i-1]$ can be segmented.
- We initialize $dp[0] = \text{true}$ (empty string is always valid).
- We will iterate over all positions i and check all possible substrings $s[j...i-1]$ to see if they exist in the dictionary and if $dp[j]$ is true.

Step 2: Iterative Dry Run in Tabular Form

i	Substring (sentence[0...i-1])	Valid Segment Found?	dp[i] Value
0	""	Base case	true
1	"i"	✓ ("i" in dict)	true
2	"il"	✗	false
3	"ili"	✗	false
4	"ilik"	✗	false
5	"ilike"	✓ ("like" in dict, $dp[1]$ is true)	true
6	"ilikep"	✗	false
7	"ilikepe"	✗	false
8	"ilike pep"	✓ ("pep")	true

		in dict, dp[5] is true)	
9	"ilikepepp"	✗	false
10	"ilikepeppe"	✗	false
11	"ilikepepper"	✓ ("pepper" in dict, dp[5] is true)	true
12	"ilikepeppere"	✗	false
13	"ilikepepperea"	✗	false
14	"ilikepeppereat"	✗	false
15	"ilikepeppereati"	✗	false
16	"ilikepeppereatin"	✗	false
17	"ilikepeppereating"	✓ ("eating" in dict, dp[11] is true)	true
18	"ilikepeppereatingm"	✗	false
19	"ilikepeppereatingma "	✗	false
20	"ilikepeppereatingma n"	✓ ("man" in dict, dp[17] is true)	true
21	"ilikepeppereatingma ng"	✗	false
22	"ilikepeppereatingma ngo"	✓ ("mango" in dict, dp[17] is true)	true
23	"ilikepeppereatingma ngoi"	✗	false
24	"ilikepeppereatingma ngoin"	✓ ("in" in dict, dp[22] is true)	true

25	"ilikepeppereatingma ngoinp"	✗	false
26	"ilikepeppereatingma ngoinpe"	✗	false
27	"ilikepeppereatingma ngoinpep"	✓ ("pep" in dict, dp[24] is true)	true
28	"ilikepeppereatingma ngoinpepc"	✗	false
29	"ilikepeppereatingma ngoinpepcō"	✗	false
30	"ilikepeppereatingma ngoinpepcod"	✗	false
31	"ilikepeppereatingma ngoinpepcodi"	✗	false
32	"ilikepeppereatingma ngoinpepcodin"	✗	false
33	"ilikepeppereatingma ngoinpepcoding"	✓ ("pepcodi ng" in dict, dp[24] is true)	true

Step 3: Final dp Array

```
[ T  T  F  F  F  T  F  F  T  F  F  F  T  F
F  F  F  F  T  F  F  T  F  T  F  T  F  T  F
T  F  F  F  F  F  T ]
```

Since $dp[n] = dp[33] = \text{true}$, we conclude that the sentence can be segmented into words from the dictionary.

Output:-

```
4
true
```

Activity Selection in C++

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

class Activity {
public:
    int start;
    int finish;

    Activity(int s, int f) {
        start = s;
        finish = f;
    }
};

struct MyCmp {
    bool operator()(const Activity& a1, const Activity& a2) const {
        return a1.finish < a2.finish;
    }
};

int maxActivity(vector<Activity>& arr) {
    sort(arr.begin(), arr.end(), MyCmp());
    int res = 1;
    int prev = 0;
    for (int curr = 1; curr < arr.size(); curr++) {
        if (arr[curr].start >= arr[prev].finish) {
            res++;
            prev = curr;
        }
    }
    return res;
}

int main() {
    vector<Activity> arr = {Activity(12, 25),
                           Activity(10, 20), Activity(20, 30)};
    cout << maxActivity(arr) << endl;
    return 0;
}
```

Activity Selection Problem Summary:

Given n activities with start and finish times, select the maximum number of activities that **don't overlap** and **finish earliest** (greedy approach).

Input Activities (Before Sorting):

Index	Start	Finish
0	12	25
1	10	20
2	20	30

Step 1: Sort by Finish Time

Using the comparator:

```
return a1.finish < a2.finish;
```

After Sorting:

Index	Start	Finish
1	10	20
0	12	25
2	20	30

Sorted vector:

```
[ {10,20}, {12,25}, {20,30} ]
```

Step 2: Activity Selection (Greedy)

We initialize:

- res = 1 (we pick the first activity)
- prev = 0 (index of the last selected activity)

Now we iterate from curr = 1 to n-1.

Iteration Table:

curr	Activity (start, finish)	prev	arr[curr].start >= arr[prev].finish	Action	res	prev
1	(12, 25)	0	12 >= 20 → ✗ False	Skip	1	0
2	(20, 30)	0	20 >= 20 → ✓ True	Select this	2	2

curr	Activity (start, finish)	prev	arr[curr].start >= arr[prev].finish	Action	res	prev
				activity		

❖ Final Result:

- Maximum activities: **2**
- Selected activities:
 - {10, 20}
 - {20, 30}

❖ Output:
2

2

Fractional Knapsack in C++

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

class Item {
public:
    int wt, val;

    Item(int w, int v) {
        wt = w;
        val = v;
    }

    bool operator<(const Item& i) const {
        return (double)val / wt >
            (double)i.val / i.wt;
    }
};

double fracKnapsack(Item arr[], int n,
int W) {
    sort(arr, arr + n);
    double res = 0.0;

    for (int i = 0; i < n; i++) {
        if (arr[i].wt <= W) {
            res += arr[i].val;
            W -= arr[i].wt;
        } else {
            res += (arr[i].val * (double)W) /
                arr[i].wt;
            break;
        }
    }
    return res;
}

int main() {
    Item arr[] = {Item(10, 60), Item(40, 40),
        Item(20, 100), Item(30, 120)};
    int n = sizeof(arr) / sizeof(arr[0]);
    int W = 50;
    cout << fracKnapsack(arr, n, W) <<
    endl;
    return 0;
}
```

Problem Summary:

You are given:

- Items with weight wt and value val
- A maximum capacity W of the knapsack
- You can **take fractions of items**

Goal: Maximize the total value in the knapsack.

Input

Item arr[] = {Item(10, 60), Item(40, 40), Item(20, 100), Item(30, 120)};
int W = 50;

► Step 1: Calculate Value/Weight Ratio and Sort Descending

Item Weight Value Value/Weight

0	10	60	6.00
1	40	40	1.00
2	20	100	5.00
3	30	120	4.00

► After Sorting by Value/Weight (Descending):

Index	Weight	Value	Value/Weight
0	10	60	6.00
2	20	100	5.00
3	30	120	4.00
1	40	40	1.00

► Step 2: Fill the Knapsack

Initial:

$W = 50$, $res = 0.0$

► Iteration Table

Iteration	Item	Weight	Value	Can Take Fully?	Action	New W	res
0	0	10	60	✓ Yes	Take full item: $res += 60$, $W -= 10$	40	60.0
1	2	20	100	✓ Yes	Take full	20	160.0

	Iteration	Item	Weight	Value	Can Take Fully?	Action	New W	res
						item: res += 100, W -= 20		
	2	3	30	120	✗ No	Take fraction: res += 120 * 20/30 = 80.0	0	240.0
	3	1	-	-	-	Not processed (knapsack full)	0	240.0

↙ Final Output
240

Job Sequencing in deadline in C++

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <set>

class Job {
public:
    char id;
    int deadline;
    int profit;

    Job(char id, int deadline, int profit) {
        this->id = id;
        this->deadline = deadline;
        this->profit = profit;
    }
};

struct JobComparator {
    bool operator()(const Job& j1, const Job& j2) {
        if (j1.profit != j2.profit)
            return j2.profit < j1.profit;
        else
            return j2.deadline < j1.deadline;
    }
};

void
printJobScheduling(std::vector<Job> & jobs) {
    std::sort(jobs.begin(), jobs.end(),
              JobComparator());

    std::set<int> ts;
    for (int i = 0; i < jobs.size(); i++)
        ts.insert(i);

    for (const auto& job : jobs) {
        auto it =
        ts.upper_bound(job.deadline - 1);
        if (it != ts.begin()) {
            --it;
            std::cout << job.id << " ";
            ts.erase(it);
        }
    }
}

int main() {
    std::vector<Job> jobs = {
        Job('a', 2, 100),
        Job('b', 1, 19),
        Job('c', 2, 27),
        Job('d', 1, 25),
        Job('e', 3, 15)
    };
    printJobScheduling(jobs);
}
```

Input
jobs = {
 Job('a', 2, 100),
 Job('b', 1, 19),
 Job('c', 2, 27),
 Job('d', 1, 25),
 Job('e', 3, 15)
}

► Step 1: Sort Jobs by Profit (Descending), Break Tie with Deadline

Job	Deadline	Profit
a	2	100
c	2	27
d	1	25
b	1	19
e	3	15

After sorting, order remains the same.

► Step 2: Initialize Available Time Slots

We simulate time slots using a std::set<int> ts.

ts = { 0, 1, 2, 3, 4 } // these are slot *indices*, not actual times.

We only need max_deadline = 3, so slots {0, 1, 2} are enough, but in the code ts.insert(i) for all jobs is used — let's assume the set size is sufficient.

► Step 3: Process Jobs One by One

We use upper_bound(job.deadline - 1) to find the latest available slot before deadline.

Job	Deadline	Profit	Find Slot \leq Deadline - 1	Result	Scheduled?	ts After
a	2	100	upper_bound(1) → 2 → step back → 1	✓ Use slot 1	Yes	{0, 2, 3, 4}
c	2	27	upper_bound(1) → 2 → step back → 0	✓ Use slot 0	Yes	{2, 3, 4}
d	1	25	upper_bound(0) → 2 → step back → X	✗ None available	No	{2, 3, 4}
b	1	19	upper_bound(0) → 2 → step	✗ None	No	{2, 3, }

```

    std::cout << std::endl;
    return 0;
}

```

Job	Deadline	Profit	Find Slot \leq Deadline - 1	Result	Scheduled?	ts After
			back → X	available		4}
e	3	15	upper_bound(2) → 3 → step back → 2	✓ Use slot 2	Yes	{3, 4}

✓ Final Output (Jobs Scheduled)
 Output: a c e

a c e

Trie in C++

```
#include <iostream>
#include <string>
using namespace std;

class Trie {
private:
    struct TrieNode {
        char data;
        bool isTerminating;
        TrieNode* children[26];
    };

    TrieNode(char data) {
        this->data = data;
        isTerminating = false;
        for (int i = 0; i < 26; i++) {
            children[i] = nullptr;
        }
    }

    TrieNode* root;
public:
    Trie() {
        root = new TrieNode('\0');
    }

    bool search(string word) {
        return search(root, word);
    }

    void add(string word) {
        add(root, word);
    }

private:
    bool search(TrieNode* root, string word) {
        if (word.length() == 0) {
            return root->isTerminating;
        }
        int childIndex = word[0] - 'a';
        TrieNode* child = root->children[childIndex];
        if (child == nullptr) {
            return false;
        }
        return search(child, word.substr(1));
    }

    void add(TrieNode* root, string word) {
        if (word.length() == 0) {
            root->isTerminating = true;
            return;
        }
        int childIndex = word[0] - 'a';
        if (root->children[childIndex] == nullptr) {
            root->children[childIndex] = new
TrieNode(word[0]);
        }
        add(root->children[childIndex], word.substr(1));
    }
};
```

Dry Run (Step-by-Step)

💡 Step 1: Trie Initialization

- A root TrieNode is created with data = '\0', and all children set to nullptr.

📝 Step 2: Adding "this"

Word: "this"

Characters processed in order: 't' → 'h' → 'i' → 's'

Step	Char	Index	Action
1	't'	19	root->children[19] is nullptr, so create new TrieNode('t')
2	'h'	7	Create new TrieNode('h') as child of 't'
3	'i'	8	Create new TrieNode('i') as child of 'h'
4	's'	18	Create new TrieNode('s') as child of 'i', mark isTerminating = true

✓ "this" added to trie.

📝 Step 3: Adding "news"

Word: "news"

Characters: 'n' → 'e' → 'w' → 's'

Step	Char	Index	Action
1	'n'	13	root->children[13] is nullptr, create TrieNode('n')
2	'e'	4	Create TrieNode('e') under 'n'
3	'w'	22	Create TrieNode('w') under 'e'
4	's'	18	Create TrieNode('s') under 'w', mark isTerminating = true

✓ "news" added to trie.

🔍 Step 4: Searching "news"

Traversal: 'n' → 'e' → 'w' → 's'

```

int main() {
    Trie t;
    t.add("this");
    t.add("news");

    cout << boolalpha; // Print bool values as "true" or
    "false"
    cout << t.search("news") << endl; // Output: true
    cout << t.search("test") << endl; // Output: false

    return 0;
}

```

true
false

- All nodes exist and 's' has isTerminating = true.

✓ Output: true

🔍 Step 5: Searching "test"

Traversal: 't' → 'e'

- 't' exists (from "this")
- 'e' does **not** exist under 't' → return false

✗ Output: false

✓ Final Output

true
false

Fenwick in C++

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

class FenwickTree {
private:
    vector<int> fenwick;
    int n;

public:
    FenwickTree(int size) {
        n = size + 1;
        fenwick.assign(n, 0);
    }

    void add(int idx, int val) {
        idx++; // 1 based index
        while (idx < n) {
            fenwick[idx] += val;
            idx += idx & (-idx); // add last set bit
        }
    }

    int sum(int idx) {
        idx++; // 1 based index
        int ans = 0;
        while (idx > 0) {
            ans += fenwick[idx];
            idx -= idx & (-idx); // remove last set bit
        }
        return ans;
    }

    int rangeSum(int l, int r) {
        return sum(r) - sum(l - 1);
    }
};

int main() {
    vector<int> v = {1, 2, 3, 4, 5, 6, 7};
    FenwickTree tree(v.size());

    // Initialize Fenwick Tree
    for (int i = 0; i < v.size(); i++) {
        tree.add(i, v[i]);
    }

    // Query range sum [3, 5]
    cout << tree.rangeSum(3, 5) << endl; // Output: 15

    // Update index 4 with new value -3
    tree.add(4, -3);

    // Query range sum [3, 5] after update
    cout << tree.rangeSum(3, 5) << endl; // Output: 12

    return 0;
}
```

Initial Array v = {1, 2, 3, 4, 5, 6, 7}

Step 1: Build Fenwick Tree

i (0-based)	v[i]	Operation	Fenwick Tree (1-based index after update)
0	1	add(0, 1)	fenwick[1] += 1 → [0, 1, 1, 0, 1, 0, 0, 1]
1	2	add(1, 2)	fenwick[2] += 2 → [0, 1, 3, 0, 3, 0, 0, 1]
2	3	add(2, 3)	fenwick[3] += 3 → [0, 1, 3, 3, 6, 0, 0, 1]
3	4	add(3, 4)	fenwick[4] += 4 → [0, 1, 3, 3, 10, 0, 0, 1]
4	5	add(4, 5)	fenwick[5] += 5 → [0, 1, 3, 3, 10, 5, 5, 1]
5	6	add(5, 6)	fenwick[6] += 6 → [0, 1, 3, 3, 10, 5, 11, 1]
6	7	add(6, 7)	fenwick[7] += 7 → [0, 1, 3, 3, 10, 5, 11, 8]

△ Note: This is the internal fenwick[] array. Index 0 is unused.

Step 2: Query rangeSum(3, 5)

That means: sum(5) - sum(2)

► **sum(5):**

idx	fenwick[idx]	sum
6	11	11
4	10	21
0	—	21

→ sum(5) = 21

► **sum(2):**

idx	fenwick[idx]	sum
3	3	3
2	3	6
0	—	6

idx	fenwick[idx]	sum

→ sum(2) = 6

→ rangeSum(3, 5) = 21 - 6 = 15

Step 3: add(4, -3) (v[4] becomes 2)

This updates the tree:

idx	fenwick[idx] before	Change	New fenwick[idx]
5	5	-3	2
6	11	-3	8
8	— (out of range)		—

Updated fenwick[] = [0, 1, 3, 3, 10, 2, 8, 8]

Step 4: rangeSum(3, 5) again

Again: sum(5) - sum(2)

► sum(5):

idx	fenwick[idx]	sum
6	8	8
4	10	18
0	—	18

→ sum(5) = 18

► sum(2):

idx	fenwick[idx]	sum
3	3	3
2	3	6
0	—	6

→ sum(2) = 6

→ rangeSum(3, 5) = 18 - 6 = 12

	<p>✓ Final Output:</p> <p>15 12</p>
15 12	