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++

```cpp
#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;
}
```

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
```

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;
}
```

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

true

Peak element in C++

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

using namespace std;

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

    while (low <= high) {
        int mid = (low + high) / 2;

        if ((mid == 0 || arr[mid - 1] <=
arr[mid])
                && (mid == arr.size() - 1 ||
arr[mid + 1] <= arr[mid])) {
            return mid;
        }

        if (mid > 0 && arr[mid - 1] >=
arr[mid]) {
            high = mid - 1;
        } else {
            low = mid + 1;
        }
    }

    return -1; // Peak element not found
}

int main() {
    vector<int> arr = {10, 7, 8, 20, 12};
    cout << findPeakElement(arr) <<
endl;
    return 0;
}
```

Dry Run Table:

Iteration	low	high	mid	arr[mid-1]	arr[mid]	arr[mid+1]	Condition Met	Action
1	0	4	2	7	8	20	Right neighbor > mid	low = mid + 1 = 3
2	3	4	3	8	20	12	Both neighbors ≤ mid → **peak found!**	Return 3

✅ **Output:**

3

3

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:

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

Step 1: Sort the array

Sorted 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 (arr[i])	Max (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

6

Count Triplets in C++

```cpp
#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:

arr[i] + arr[j] == arr[k]

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

✅ Input Array:

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

n = 6

🔄 After Sorting:

arr[] = {1, 2, 3, 4, 5, 7}
↑ ↑ ↑ ↑ ↑ ↑
0 1 2 3 4 5 (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

	• (3, 4) → 3 + 4 = 7 • (1, 4) → 1 + 4 = 5 • (2, 3) → 2 + 3 = 5 • (1, 3) → 1 + 3 = 4 • (1, 2) → 1 + 2 = 3
Number of triplets: 6	

Count Zeroes In Sorted Matrix in C++

```cpp
#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;
}
```

Input:

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

🔍 **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

Number of buildings facing the sun: 3

Find K closest elements in C++

```cpp
#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;
}
```

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)	abs(arr[hi] - x)	Decision	New lo	New hi
1	0	5	5	abs(10 - 45) = 35	abs(60 - 45) = 15	35 > 15 → shrink left	1	5
2	1	5	4	abs(20 - 45) = 25	abs(60 - 45) = 15	25 > 15 → 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

Closest elements to 45: 30 40 50

Find Rotation Count in C++

```cpp
#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] <= arr[hi]`, return 0 — not true here (4 > 2)

🔍 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] <= arr[mid] → 4 <= 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] → 0 < 8 ✅	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**.

🔍 **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++

```cpp
#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) ⇒ 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"

	✅ **Final Output:** 975343
975343	

Largest Perimeter triangle in C++

```cpp
#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 < 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

33

Marks of PCM in C++

```
#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};
```

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**

- ✅ 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++

```cpp
#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;
}
```

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 arr[j] - arr[i] == 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

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	