**1. C++ std::vector vs. C# List<T>**

| **C++ STL** | **Description** | **C# Equivalent** | **Description** |
| --- | --- | --- | --- |
| std::vector | A dynamic array that can resize itself automatically. | List<T> | A strongly-typed dynamic array. |
| push\_back(value) | Adds an element to the end of the vector. | Add(item) | Adds an item to the end of the list. |
| pop\_back() | Removes the last element of the vector. | RemoveAt(list.Count-1) | Removes the last item. |
| size() | Returns the number of elements in the vector. | Count | Gets the number of elements in the list. |
| empty() | Checks if the vector is empty. | Count == 0 | Returns true if the list is empty. |
| resize(newSize) | Resizes the vector to contain the specified number of elements. | Capacity | Adjusts the storage capacity of the list. |
| at(index) | Accesses the element at the specified index. | list[index] | Gets or sets the element at the specified index. |
| insert(pos, value) | Inserts an element at the specified position. | Insert(index, item) | Inserts an item at the specified index. |
| erase(pos) | Removes the element at the specified position. | RemoveAt(index) | Removes the item at the specified index. |
| clear() | Removes all elements from the vector. | Clear() | Clears all items in the list. |
| begin() | Returns an iterator to the beginning. | list[0] | Access the first element (C# supports direct indexing). |
| end() | Returns an iterator to the end (one past the last element). | list[list.Count-1] | Access the last element. |
| swap(otherVector) | Swaps the contents of two vectors. | Manual logic | No direct method, requires swapping through temporary variables. |
| emplace(pos, args...) | Constructs and inserts a new element at the specified position. | Add() | No direct equivalent; use Add with constructor logic. |
| capacity() | Returns the total storage capacity of the vector. | Capacity | Gets the total storage capacity of the list. |
| shrink\_to\_fit() | Reduces capacity to fit the current size. | Not Available | No direct equivalent. |
| front() / back() | Accesses the first/last elements. | First() / Last() | Accesses the first/last elements (using LINQ). |

**2. C++ std::deque vs. C# LinkedList<T>**

| **C++ STL** | **Description** | **C# Equivalent** | **Description** |
| --- | --- | --- | --- |
| std::deque | A double-ended queue that supports insertion and deletion at both ends. | LinkedList<T> | A doubly-linked list supporting fast insertion and removal at both ends. |
| push\_back(value) | Adds an element to the end of the deque. | AddLast(item) | Adds an item to the end of the linked list. |
| push\_front(value) | Adds an element to the front of the deque. | AddFirst(item) | Adds an item to the beginning of the linked list. |
| pop\_back() | Removes the last element. | RemoveLast() | Removes the last node from the linked list. |
| pop\_front() | Removes the first element. | RemoveFirst() | Removes the first node from the linked list. |
| front() | Returns a reference to the first element. | First.Value | Gets the value of the first node. |
| back() | Returns a reference to the last element. | Last.Value | Gets the value of the last node. |
| size() | Returns the number of elements in the deque. | Count | Gets the number of elements in the linked list. |
| clear() | Removes all elements. | Clear() | Removes all nodes from the linked list. |
| empty() | Checks whether the deque is empty. | Count == 0 | Returns true if the list is empty. |

**3. C++ std::map vs. C# SortedDictionary<TKey, TValue>**

| **C++ STL** | **Description** | **C# Equivalent** | **Description** |
| --- | --- | --- | --- |
| std::map | A sorted associative container with unique keys. | SortedDictionary<TKey, TValue> | A collection of key-value pairs. |
| insert({key, value}) | Adds a key-value pair. | Add(key, value) | Adds the specified key-value pair. |
| erase(key) | Removes the key-value pair for the specified key. | Remove(key) | Removes the value with the specified key. |
| find(key) | Returns an iterator to the key-value pair, or end() if not found. | ContainsKey(key) | Returns true if the specified key exists. |
| size() | Returns the number of key-value pairs. | Count | Gets the number of key-value pairs. |
| clear() | Removes all key-value pairs. | Clear() | Clears the dictionary. |
| [] | Accesses the value for a given key (inserts if key doesn't exist). | dictionary[key] | Gets or sets the value for the specified key. |
| emplace(key, value) | Inserts a new key-value pair if the key is unique. | TryAdd(key, value) | Tries to add a new key-value pair; does nothing if key exists. |

**4. C++ std::set vs. C# HashSet<T>**

| **C++ STL** | **Description** | **C# Equivalent** | **Description** |
| --- | --- | --- | --- |
| std::set | A collection of unique elements. | HashSet<T> | A collection of unique elements. |
| insert(value) | Adds a new element to the set. | Add(item) | Adds an item to the set. |
| erase(value) | Removes an element from the set. | Remove(item) | Removes the specified item from the set. |
| find(value) | Searches for an element. | Contains(item) | Checks whether the set contains the specified item. |
| size() | Returns the number of elements. | Count | Gets the number of elements in the set. |
| clear() | Removes all elements. | Clear() | Clears all elements from the set. |
| begin()/end() | Returns iterators to the beginning/end. | Not Available | No direct equivalent; enumerations are used in C#. |

**5. C++ std::unordered\_map vs. C# Dictionary<TKey, TValue>**

| **C++ STL Method** | **Description** | **C# Method/Property** | **Description** |
| --- | --- | --- | --- |
| insert({key, value}) | Inserts a key-value pair. | Add(key, value) | Adds a key-value pair. |
| erase(key) | Removes the key-value pair by key. | Remove(key) | Removes the key-value pair by key. |
| find(key) | Returns iterator to the key-value pair or end() if not found. | ContainsKey(key) | Checks if the key exists. |
| count(key) | Returns 1 if the key exists, 0 otherwise. | ContainsKey(key) | Equivalent; returns true or false. |
| operator[] | Accesses or inserts a value for a given key. | dictionary[key] | Accesses or sets the value by key. |
| size() | Returns the number of elements. | Count | Returns the number of key-value pairs. |
| empty() | Checks if the map is empty. | Count == 0 | Returns true if the dictionary is empty. |
| clear() | Removes all elements. | Clear() | Clears all elements. |
| bucket\_count() | Returns the number of buckets in the hash table. | Not Available | Not directly supported. |
| load\_factor() | Returns the load factor of the hash table. | Not Available | Not directly supported. |
| begin() / end() | Returns iterators to the start/end of the container. | foreach | Enumerates over key-value pairs. |

**6. C++ std::multiset vs. C# SortedList<TKey, TValue>**

| **C++ STL** | **Description** | **C# Equivalent** | **Description** |
| --- | --- | --- | --- |
| std::multiset | A sorted associative container with duplicate elements allowed. | SortedList<TKey, TValue> | A collection of key-value pairs, sorted by keys. |
| insert(value) | Adds a new element to the multiset. | Add(key, value) | Adds the specified key-value pair. |
| erase(value) | Removes all occurrences of the element. | Remove(key) | Removes the value associated with the specified key. |
| size() | Returns the number of elements. | Count | Gets the number of elements in the list. |
| find(value) | Finds the first occurrence of the value. | ContainsKey(key) | Checks if a key exists. |
| clear() | Removes all elements. | Clear() | Clears all elements. |

**7. C++ std::stack vs. # Stack<T>**

| **C++ STL** | **Description** | **C# Equivalent** | **Description** |
| --- | --- | --- | --- |
| std::stack | A container adaptor with LIFO (Last In First Out) operations. | Stack<T> | A collection implementing a LIFO stack. |
| push(value) | Adds an element to the top of the stack. | Push(item) | Pushes an item onto the stack. |
| pop() | Removes the top element of the stack. | Pop() | Removes and returns the top item from the stack. |
| top() | Returns a reference to the top element of the stack. | Peek() | Returns the item at the top of the stack without removing it. |
| size() | Returns the number of elements in the stack. | Count | Gets the number of items in the stack. |
| empty() | Checks whether the stack is empty. | Count == 0 | Returns true if the stack is empty. |

**8. C++ std::queue vs. C# Queue<T>**

| **C++ STL** | **Description** | **C# Equivalent** | **Description** |
| --- | --- | --- | --- |
| std::queue | A container adaptor with FIFO (First In First Out) operations. | Queue<T> | A collection implementing a FIFO queue. |
| push(value) | Adds an element to the back of the queue. | Enqueue(item) | Adds an item to the back of the queue. |
| pop() | Removes the front element of the queue. | Dequeue() | Removes and returns the item at the front of the queue. |
| front() | Returns a reference to the front element. | Peek() | Returns the item at the front without removing it. |
| size() | Returns the number of elements in the queue. | Count | Gets the number of items in the queue. |
| empty() | Checks whether the queue is empty. | Count == 0 | Returns true if the queue is empty. |

**9. C++ std::priority\_queue vs. C# SortedSet<T>**

| **C++ STL** | **Description** | **C# Equivalent** | **Description** |
| --- | --- | --- | --- |
| std::priority\_queue | A container adaptor that maintains a heap structure. | SortedSet<T> | A collection that maintains sorted order of unique elements. |
| push(value) | Adds an element to the priority queue. | Add(item) | Adds an item to the set (maintains sorted order). |
| pop() | Removes the element with the highest priority. | Not Available | No direct equivalent; custom logic is required for heap functionality. |
| top() | Returns the element with the highest priority. | Max | Gets the largest element in the set. |
| size() | Returns the number of elements in the priority queue. | Count | Gets the number of elements in the set. |

**10. C++ std::unordered\_set vs. C# HashSet<T>**

| **C++ STL** | **Description** | **C# Equivalent** | **Description** |
| --- | --- | --- | --- |
| std::unordered\_set | A container that contains unique elements with fast access. | HashSet<T> | A collection of unique elements implemented as a hash table. |
| insert(value) | Adds an element to the set. | Add(item) | Adds an item to the set. |
| erase(value) | Removes an element from the set. | Remove(item) | Removes the specified item from the set. |
| find(value) | Searches for an element. | Contains(item) | Checks whether the set contains the specified item. |
| size() | Returns the number of elements in the set. | Count | Gets the number of elements in the set. |
| clear() | Removes all elements. | Clear() | Clears all elements from the set. |

**11. C++ std::array vs. C# Array**

| **C++ STL** | **Description** | **C# Equivalent** | **Description** |
| --- | --- | --- | --- |
| std::array | A fixed-size array with support for STL operations. | Array | A fixed-size array. |
| at(index) | Accesses the element at the specified index (bounds-checked). | array[index] | Accesses the element at the specified index. |
| size() | Returns the number of elements in the array. | Length | Gets the total number of elements. |
| begin()/end() | Returns iterators to the beginning/end. | Not Available | No direct equivalent; use loops for traversal. |

**C++ std::deque vs. C# Deque<T> (via System.Collections.Concurrent.ConcurrentQueue<T>)**

| **C++ STL Method** | **Description** | **C# Method/Property** | **Description** |
| --- | --- | --- | --- |
| push\_front(value) | Adds an element to the front. | Enqueue(item) | Adds an element to the front. |
| push\_back(value) | Adds an element to the back. | Enqueue(item) | Adds an element to the back. |
| pop\_front() | Removes the first element. | TryDequeue(out T item) | Removes the first element. |
| pop\_back() | Removes the last element. | Not Available | Use custom implementation. |
| size() | Returns the number of elements. | Count | Returns the number of elements. |
| at(index) | Accesses the element at the given index. | Not Available | Use a loop or access items directly. |

**C++ std::priority\_queue vs. C# PriorityQueue<T>**

| **C++ STL Method** | **Description** | **C# Method/Property** | **Description** |
| --- | --- | --- | --- |
| push(value) | Adds a new element. | Enqueue(item, priority) | Adds an item with a specified priority. |
| pop() | Removes the element with the highest priority. | Dequeue() | Removes and returns the highest-priority item. |
| top() | Returns the element with the highest priority. | Peek() | Returns the highest-priority item without removing it. |
| size() | Returns the number of elements. | Count | Gets the number of elements. |
| empty() | Checks whether the priority queue is empty. | Count == 0 | Checks whether the queue is empty. |

**//C++ Templates**

**//1. LeetCode Template**

#include <iostream>

#include <vector>

using namespace std;

class Solution {

public:

// Define your function here

int exampleFunction(vector<int>& nums) {

int sum = 0;

for (int num : nums) {

sum += num;

}

return sum;

}

};

int main() {

Solution solution;

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

cout << solution.exampleFunction(nums) << endl; // Example call

return 0;

}

//CodeChef

#include <bits/stdc++.h>

using namespace std;

void solve() {

int n;

cin >> n; // Input size

vector<int> arr(n);

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

cin >> arr[i];

}

// Implement the logic

int result = accumulate(arr.begin(), arr.end(), 0); // Example logic

cout << result << endl; // Output result

}

int main() {

ios::sync\_with\_stdio(0);

cin.tie(0);

int t; // Number of test cases

cin >> t;

while (t--) {

solve();

}

return 0;

}

//Code forces

#include <bits/stdc++.h>

using namespace std;

void solve() {

int n;

cin >> n;

vector<int> arr(n);

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

cin >> arr[i];

}

// Implement the solution

sort(arr.begin(), arr.end()); // Example operation

for (int num : arr) {

cout << num << " ";

}

cout << endl;

}

int main() {

ios::sync\_with\_stdio(0);

cin.tie(0);

int t = 1; // Default single test case

cin >> t; // Comment if the problem specifies a single test case

while (t--) {

solve();

}

return 0;

}

//C#

//CodeForces

using System;

using System.Linq;

class Program {

static void Solve() {

int n = int.Parse(Console.ReadLine());

int[] arr = Console.ReadLine().Split().Select(int.Parse).ToArray();

// Example logic

Array.Sort(arr);

Console.WriteLine(string.Join(" ", arr));

}

static void Main(string[] args) {

int t = int.Parse(Console.ReadLine()); // Multiple test cases

while (t-- > 0) {

Solve();

}

}

}