**Q1: 20 wallets in sorted order**

**Theory:**

* Wallets already sorted.
* If we still want to sort, we need an algorithm that works very fast when input is already sorted.
* **Insertion Sort** is best → time complexity = **O(n)** for sorted input.

**Explanation for beginners:**

* Insertion Sort takes one element and puts it in the correct position.
* If already sorted, no swaps needed → very efficient.

**C++ Program:**

#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--;

}

arr[j+1] = key;

}

}

int main() {

int arr[20];

int money = 20;

for(int i=0; i<20; i++) {

arr[i] = money;

money += 10;

}

insertionSort(arr, 20);

cout << "Sorted wallets: ";

for(int i=0; i<20; i++) cout << arr[i] << " ";

return 0;

}

**Q2: Wallets (max value = 6)**

**Theory:**

* Values are very small (0–6).
* **Counting Sort** works best → it counts how many times each value appears.
* Time complexity = O(n + k) (where k = max value = 6). Very efficient here.

**Explanation for beginners:**

* We count frequency of each wallet value.
* Then place them back in sorted order.

**C++ Program:**

#include <iostream>

using namespace std;

void countingSort(int arr[], int n, int maxVal) {

int count[maxVal+1] = {0};

for(int i=0; i<n; i++) count[arr[i]]++;

int index=0;

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

while(count[i]--) arr[index++] = i;

}

}

int main() {

int arr[10] = {2,2,2,3,3,0,0,1,4,6};

countingSort(arr, 10, 6);

cout << "Sorted wallets: ";

for(int i=0;i<10;i++) cout << arr[i] << " ";

return 0;

}

**Q3: Scores of 12 students → Quick Sort**

**Theory:**

* Data is unsorted and spread widely.
* **Quick Sort** is best: divide-and-conquer, O(n log n) average time.

**Explanation:**

* Pick a pivot, divide array into smaller/larger, and sort recursively.

**C++ Program:**

#include <iostream>

using namespace std;

int partition(int arr[], int low, int high) {

int pivot = arr[high];

int i = low - 1;

for(int j=low; j<high; j++) {

if(arr[j] < pivot) {

i++;

swap(arr[i], arr[j]);

}

}

swap(arr[i+1], arr[high]);

return i+1;

}

void quickSort(int arr[], int low, int high) {

if(low < high) {

int pi = partition(arr, low, high);

quickSort(arr, low, pi-1);

quickSort(arr, pi+1, high);

}

}

int main() {

int arr[12] = {45,12,78,34,23,89,67,11,90,54,32,76};

quickSort(arr, 0, 11);

cout << "Sorted scores: ";

for(int i=0;i<12;i++) cout << arr[i] << " ";

return 0;

}

**Q4: Deadlines (divide & conquer hint)**

**Theory:**

* Manager hints at “divide into unequal parts” → this means **Quick Sort** (not Merge Sort).

✅ Same reasoning + code as Q3.

**Q5: 50 squares → Searching**

**Theory:**

* Data (areas) stored in an array.
* Best searching = **Binary Search** because array is sorted.
* Time complexity = O(log n).

**Explanation:**

* Binary Search divides array into half repeatedly until element is found or not.

**C++ Program:**

#include <iostream>

using namespace std;

bool binarySearch(int arr[], int n, int key) {

int low = 0, high = n-1;

while(low <= high) {

int mid = (low+high)/2;

if(arr[mid] == key) return true;

else if(arr[mid] < key) low = mid+1;

else high = mid-1;

}

return false;

}

int main() {

int arr[50];

for(int i=0;i<50;i++) arr[i] = (50-i)\*(50-i); // Example values

int area = 100; // Area to search

if(binarySearch(arr, 50, area))

cout << "Area found";

else

cout << "Area not found";

return 0;

}

**Q6: Chief guest meets players → Linked List**

**Theory:**

* Players introduce next players → linear structure → **Singly Linked List**.

**Explanation:**

* Each node stores player name and pointer to next player.

**C++ Program:**

#include <iostream>

using namespace std;

struct Node {

string name;

Node\* next;

};

int main() {

Node\* head = new Node{"Player1", NULL};

Node\* second = new Node{"Player2", NULL};

Node\* third = new Node{"Player3", NULL};

head->next = second;

second->next = third;

Node\* temp = head;

cout << "Chief guest meets: ";

while(temp != NULL) {

cout << temp->name << " ";

temp = temp->next;

}

return 0;

}

**Q7: Bus journey → Doubly Linked List**

**Theory:**

* Journey is forward (A→B→C→D) and backward (D→C→B→A).
* Requires **Doubly Linked List** (can move forward and backward).

**C++ Program:**

#include <iostream>

using namespace std;

struct Node {

string stop;

Node\* next;

Node\* prev;

};

int main() {

Node\* A = new Node{"A", NULL, NULL};

Node\* B = new Node{"B", NULL, A};

A->next = B;

Node\* C = new Node{"C", NULL, B};

B->next = C;

Node\* D = new Node{"D", NULL, C};

C->next = D;

cout << "Onward journey: ";

Node\* temp = A;

while(temp) {

cout << temp->stop << " ";

temp = temp->next;

}

cout << "\nReturn journey: ";

temp = D;

while(temp) {

cout << temp->stop << " ";

temp = temp->prev;

}

return 0;

}

**Q8: Gang matrices → Multiply 2D arrays**

**Theory:**

* Dalta Gang (4×2) × Malta Gang (2×3) → Result is (4×3).
* Normal matrix multiplication rule.

**C++ Program:**

#include <iostream>

using namespace std;

int main() {

int A[4][2] = {{1,2},{3,4},{5,6},{7,8}};

int B[2][3] = {{1,2,3},{4,5,6}};

int C[4][3];

for(int i=0;i<4;i++) {

for(int j=0;j<3;j++) {

C[i][j] = 0;

for(int k=0;k<2;k++) {

C[i][j] += A[i][k] \* B[k][j];

}

}

}

cout << "Result matrix:\n";

for(int i=0;i<4;i++) {

for(int j=0;j<3;j++) {

cout << C[i][j] << " ";

}

cout << endl;

}

return 0;

}

**SECTION – B (All BST-based)**

We’ll use a **BST template** and modify for each question.

**Q1: Store names in BST & find successor of M**

**Theory:**

* BST stores data in sorted order.
* Successor = next larger node in BST.

**C++ Program:**

#include <iostream>

using namespace std;

struct Node {

char data;

Node\* left;

Node\* right;

Node(char val) : data(val), left(NULL), right(NULL) {}

};

Node\* insert(Node\* root, char key) {

if(!root) return new Node(key);

if(key < root->data) root->left = insert(root->left, key);

else root->right = insert(root->right, key);

return root;

}

Node\* minValue(Node\* node) {

Node\* current = node;

while(current && current->left) current = current->left;

return current;

}

Node\* findSuccessor(Node\* root, char key) {

Node\* succ = NULL;

while(root) {

if(key < root->data) {

succ = root;

root = root->left;

}

else root = root->right;

}

return succ;

}

int main() {

char arr[] = {'Q','S','R','T','M','A','B','P','N'};

Node\* root = NULL;

for(char c : arr) root = insert(root, c);

Node\* succ = findSuccessor(root, 'M');

if(succ) cout << "Successor of M is " << succ->data;

else cout << "No successor found";

return 0;

}

**Q2: In-order, Pre-order, Post-order Traversals**

**Theory:**

* In-order = Left, Root, Right (sorted order for BST).
* Pre-order = Root, Left, Right.
* Post-order = Left, Right, Root.

**C++ Program:**

void inorder(Node\* root) {

if(root) {

inorder(root->left);

cout << root->data << " ";

inorder(root->right);

}

}

void preorder(Node\* root) {

if(root) {

cout << root->data << " ";

preorder(root->left);

preorder(root->right);

}

}

void postorder(Node\* root) {

if(root) {

postorder(root->left);

postorder(root->right);

cout << root->data << " ";

}

}

**Q3: Search in BST**

Node\* search(Node\* root, int key) {

if(!root || root->data == key) return root;

if(key < root->data) return search(root->left, key);

return search(root->right, key);

}

**Q4: Roll numbers → Insert in BST + Inorder Traversal**

**Theory:**

* Inorder traversal of BST gives numbers in ascending order.

✅ Just insert roll numbers {45,12,78,34,23,89,67,11,90,54} and print inorder.

**Q5: Delete node from BST**

**Theory:**

* 3 cases:
  1. Leaf node → just delete.
  2. One child → replace with child.
  3. Two children → replace with inorder successor.

**C++ Function:**

Node\* deleteNode(Node\* root, int key) {

if(!root) return root;

if(key < root->data) root->left = deleteNode(root->left, key);

else if(key > root->data) root->right = deleteNode(root->right, key);

else {

if(!root->left) return root->right;

else if(!root->right) return root->left;

Node\* temp = minValue(root->right);

root->data = temp->data;

root->right = deleteNode(root->right, temp->data);

}

return root;

}