

Unit 1 — Algorithms & Complexity (Q1–Q10)

Q1 — Factorial (Recursive)

```
#include <iostream>

using namespace std;

long long fact(int n) {
    if (n <= 1) return 1LL;
    return n * fact(n - 1);
}

int main() {
    int n = 5; // sample input
    cout << "5! = " << fact(n) << endl;
    return 0;
}
```

Explanation: Classic recursion. Time Complexity: $O(n)$, Space Complexity: $O(n)$ (call stack).

Q2 — Fibonacci Series (Iterative)

```
#include <iostream>

using namespace std;

int main() {
    int n = 10;
    int a = 0, b = 1;
    cout << "Fibonacci Series: ";
    for (int i = 0; i < n; ++i) {
        cout << a << (i+1==n ? '\n' : ' ');
        int c = a + b;
        a = b; b = c;
    }
}
```

```
}  
  
return 0;  
  
}
```

Explanation: Iterative approach, $O(n)$ time, $O(1)$ space.

Q3 — Count Operations in Loop

```
#include <iostream>  
  
using namespace std;  
  
int main() {  
    int n = 100;  
  
    long long count = 0;  
  
    for (int i = 0; i < n; ++i) count++;  
  
    cout << "Operations Count: " << count << endl;  
  
    return 0;  
  
}
```

Explanation: Demonstrates linear complexity $O(n)$.

Q4 — Nested Loops ($O(n^2)$)

```
#include <iostream>  
  
using namespace std;  
  
int main() {  
    int n = 50;  
  
    long long cnt = 0;  
  
    for (int i = 0; i < n; ++i)  
        for (int j = 0; j < n; ++j)  
            ++cnt;  
  
    cout << "Total Operations: " << cnt << endl;  
  
    return 0;  
  
}
```

```
}
```

Explanation: Two nested loops $\rightarrow O(n^2)$.

Q5 — GCD (Euclidean Algorithm, Recursive)

```
#include <iostream>
```

```
using namespace std;
```

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

```
    return (b == 0) ? a : gcd(b, a % b);
```

```
}
```

```
int main() {
```

```
    cout << "GCD of 60 and 24 = " << gcd(60, 24) << endl;
```

```
    return 0;
```

```
}
```

Explanation: Time Complexity $\sim O(\log \min(a,b))$.

Q6 — Swap Two Numbers (Pass by Reference)

```
#include <iostream>
```

```
using namespace std;
```

```
void swapNum(int &x, int &y) {
```

```
    int t = x; x = y; y = t;
```

```
}
```

```
int main() {
```

```
    int a = 10, b = 20;
```

```
    swapNum(a, b);
```

```
    cout << "a = " << a << ", b = " << b << endl;
```

```
    return 0;
```

```
}
```

Explanation: Swaps two numbers without returning; $O(1)$ time.

Q7 — Reverse a Number

```
#include <iostream>
```

```
using namespace std;
```

```
int main() {
```

```
    int n = 12345;
```

```
    int rev = 0;
```

```
    while (n > 0) {
```

```
        rev = rev * 10 + n % 10;
```

```
        n /= 10;
```

```
    }
```

```
    cout << "Reversed Number: " << rev << endl;
```

```
    return 0;
```

```
}
```

Explanation: Extracts digits using modulo/division; $O(\log n)$ time.

Q8 — Prime Check (Sqrt Method)

```
#include <iostream>
```

```
#include <cmath>
```

```
using namespace std;
```

```
bool isPrime(int n) {
```

```
    if (n <= 1) return false;
```

```
    if (n <= 3) return true;
```

```
    if (n % 2 == 0) return false;
```

```
    for (int i = 3; i * 1LL * i <= n; i += 2)
```

```
        if (n % i == 0) return false;
```

```

        return true;
    }

    int main() {
        cout << "13 is " << (isPrime(13) ? "Prime" : "Not Prime") << endl;
    }

```

Explanation: Checks primes efficiently using $O(\sqrt{n})$ time.

Q9 — Power (Recursive)

```

#include <iostream>

using namespace std;

long long power(long long x, int y) {
    if (y == 0) return 1;
    return x * power(x, y - 1);
}

```

```

int main() {
    cout << "2^5 = " << power(2, 5) << endl;
}

```

Explanation: Simple recursion; can be optimized with exponentiation by squaring.

Q10 — Sum of Digits (Recursive)

```

#include <iostream>

using namespace std;

int sumDigits(int n) {
    if (n == 0) return 0;
    return (n % 10) + sumDigits(n / 10);
}

```

```
int main() {  
    cout << "Sum of digits of 1234 = " << sumDigits(1234) << endl;  
}
```

Explanation: Recursive sum of digits; $O(\log n)$ recursion depth.

Unit 2 — Arrays, Stack & Queue (Q11–Q25)

Overview

This section contains problems Q11–Q25 focused on **arrays, stacks, and queues**. Each question includes a short problem statement, the full C++ code wrapped with triple backticks (``), and a concise explanation — ready for direct copy into your GitHub .md file.

Q11 — Insert element into array (shift right)

Problem: Insert an element at position pos (1-based) by shifting elements to the right.

```
#include <iostream>

using namespace std;

int main() {
    int arr[20] = {1,2,3,4,5};
    int n = 5, pos = 3, val = 99;
    for (int i = n; i >= pos; --i) arr[i] = arr[i-1];
    arr[pos-1] = val; ++n;
    for (int i = 0; i < n; ++i) cout << arr[i] << " ";
    cout << endl;
    return 0;
}
```

Explanation: Shift elements from the end to pos rightwards, place val at pos-1. Time: **O(n)**.

Q12 — Delete element from array (shift left)

Problem: Delete element at position pos (1-based) by shifting elements left.

```
#include <iostream>

using namespace std;

int main() {
```

```

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

int n = 5, pos = 2;

for (int i = pos-1; i < n-1; ++i) arr[i] = arr[i+1];

--n;

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

cout << endl;

return 0;

}

```

Explanation: Overwrite the element at pos-1 with the next element repeatedly. Time: **$O(n)$** .

Q13 — Reverse an array in-place

Problem: Reverse an array using two-pointer swaps.

```

#include <iostream>

using namespace std;

int main() {

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

    int n = 5;

    for (int i = 0; i < n/2; ++i) swap(a[i], a[n-1-i]);

    for (int x: a) cout << x << " ";

    cout << endl;

    return 0;

}

```

Explanation: Swap symmetric elements until the middle. Time: **$O(n)$** , Space: **$O(1)$** .

Q14 — Find maximum element

Problem: Find the maximum element in an array.

```

#include <iostream>

```

```
using namespace std;
```

```
int main() {  
    int a[] = {10,50,20,5,60};  
    int n = 5, mx = a[0];  
    for (int i = 1; i < n; ++i) if (a[i] > mx) mx = a[i];  
    cout << "Max = " << mx << endl;  
    return 0;  
}
```

Explanation: Single-pass scan keeping the current maximum. Time: **$O(n)$** .

Q15 — Stack (array implementation)

Problem: Implement a simple stack using a fixed-size array.

```
#include <iostream>
```

```
using namespace std;
```

```
struct Stack {  
    int st[100];  
    int top = -1;  
    void push(int x) { if (top < 99) st[++top] = x; }  
    int pop() { return (top >= 0) ? st[top--] : -1; }  
    int peek() { return (top >= 0) ? st[top] : -1; }  
};
```

```
int main() {  
    Stack s;  
    s.push(10); s.push(20);  
    cout << s.pop() << endl; // 20
```

```

    cout << s.peek() << endl; // 10

    return 0;
}

```

Explanation: Constant-time push/pop/peek. Space limited by array size. Time: **$O(1)$** per operation.

Q16 — Queue (array implementation, simple)

Problem: Simple queue using front and rear indices (no reuse of freed space).

```

#include <iostream>

using namespace std;

struct Queue {
    int q[100];
    int f = 0, r = -1;
    void enqueue(int x) { q[++r] = x; }
    int dequeue() { return (f <= r) ? q[f++] : -1; }
};

```

```

int main() {
    Queue qq;
    qq.enqueue(10); qq.enqueue(20);
    cout << qq.dequeue() << endl; // 10
    return 0;
}

```

Explanation: Simple fixed-array queue; does not wrap around — may waste space. Operations: **$O(1)$** .

Q17 — Circular Queue (array)

Problem: Fixed-size circular queue using modulo arithmetic.

```

#include <iostream>

using namespace std;

struct CQueue {
    int q[5], f = -1, r = -1;
    bool isFull() { return ((r+1)%5 == f); }
    bool isEmpty() { return f == -1; }
    void enqueue(int x) {
        if (isFull()) return;
        if (isEmpty()) f = 0;
        r = (r + 1) % 5; q[r] = x;
    }
    int dequeue() {
        if (isEmpty()) return -1;
        int v = q[f];
        if (f == r) f = r = -1;
        else f = (f + 1) % 5;
        return v;
    }
};

int main() {
    CQueue cq;
    cq.enqueue(1); cq.enqueue(2); cq.enqueue(3);
    cout << cq.dequeue() << endl; // 1
    return 0;
}

```

Explanation: Reuses freed slots, avoiding wasted space. Operations: **O(1)**.

Q18 — Stack via Linked List

Problem: Implement a stack using a singly linked list for dynamic size.

```
#include <iostream>

using namespace std;

struct Node { int data; Node *next; Node(int v):data(v),next(nullptr){} };

struct StackLL {

    Node *top = nullptr;

    void push(int x) { Node* n = new Node(x); n->next = top; top = n; }

    int pop() { if(!top) return -1; Node* t = top; int v = t->data; top = top->next; delete t; return v; }

};

int main() {

    StackLL s; s.push(10); s.push(20);

    cout << s.pop() << endl; // 20

    return 0;

}
```

Explanation: Dynamic stack; push/pop at head for **O(1)** operations. Remember to free memory.

Q19 — Queue via Linked List

Problem: Implement a queue using singly linked list with front and rear pointers.

```
#include <iostream>

using namespace std;

struct Node { int data; Node* next; Node(int v):data(v),next(nullptr){} };

struct QueueLL {
```

```

Node *front = nullptr, *rear = nullptr;

void enqueue(int x) {
    Node* n = new Node(x);
    if (!rear) front = rear = n;
    else rear->next = n, rear = n;
}

int dequeue() {
    if (!front) return -1;
    Node* t = front; int v = t->data; front = front->next;
    if (!front) rear = nullptr;
    delete t; return v;
}

};

int main() {
    QueueLL q; q.enqueue(5); q.enqueue(6);
    cout << q.dequeue() << endl; // 5
    return 0;
}

```

Explanation: Maintain both front and rear to achieve amortized **O(1)** enqueue and dequeue. Be careful with empty queue cleanup.

Q20 — Reverse a Stack (recursively)

Problem: Reverse a stack using recursion and an insertBottom helper.

```
#include <iostream>
```

```
#include <stack>
```

```
using namespace std;
```

```

void insertBottom(stack<int>& s, int x) {
    if (s.empty()) { s.push(x); return; }
    int t = s.top(); s.pop();
    insertBottom(s, x);
    s.push(t);
}

void reverseStack(stack<int>& s) {
    if (s.empty()) return;
    int x = s.top(); s.pop();
    reverseStack(s);
    insertBottom(s, x);
}

int main() {
    stack<int> s; s.push(1); s.push(2); s.push(3);
    reverseStack(s);
    while (!s.empty()) { cout << s.top() << " "; s.pop(); } // prints 1 2 3
    cout << endl;
    return 0;
}

```

Explanation: reverseStack pops all elements recursively; insertBottom places each popped element at bottom, reversing order. Complexity: $O(n^2)$ due to repeated insertBottom operations.

Q21 — Evaluate Postfix Expression

Problem: Evaluate a postfix expression with single-digit operands using a stack.

```
#include <iostream>
```

```
#include <stack>
```

```
#include <string>
```

```
using namespace std;
```

```
int evalPostfix(const string& s) {  
    stack<int> st;  
    for (char c: s) {  
        if (isdigit(c)) st.push(c - '0');  
        else {  
            int b = st.top(); st.pop();  
            int a = st.top(); st.pop();  
            if (c == '+') st.push(a + b);  
            else if (c == '-') st.push(a - b);  
            else if (c == '*') st.push(a * b);  
            else if (c == '/') st.push(a / b);  
        }  
    }  
    return st.top();  
}
```

```
int main() {  
    cout << evalPostfix("23*54*+") << endl; // example  
    return 0;  
}
```

Explanation: Use stack to push operands; on operator, pop two operands, compute, push result. Time: $O(n)$.

Q22 — Parentheses Balancing

Problem: Check if parentheses in a string are balanced using a stack.

```
#include <iostream>
```

```

#include <stack>

#include <string>

using namespace std;

bool isBalanced(const string& s) {
    stack<char> st;
    for (char c: s) {
        if (c == '(') st.push(c);
        else if (c == ')') {
            if (st.empty()) return false;
            st.pop();
        }
    }
    return st.empty();
}

int main() {
    cout << (isBalanced("()")) ? "Balanced" : "Not Balanced" << endl;
}

```

Explanation: Push '(' and pop when seeing ')'. If stack empty when popping or stack non-empty at end → not balanced. Time: **O(n)**.

Q23 — Peek / Top of Stack (using std::stack)

Problem: Demonstrate peek/top operation on a stack.

```

#include <iostream>

#include <stack>

using namespace std;

```

```

int main() {
    stack<int> s;
    s.push(10);
    s.push(20);
    cout << "Top = " << s.top() << endl;
    s.pop();
    cout << "Top after pop = " << s.top() << endl;
    return 0;
}

```

Explanation: top() returns the top element without removing it. pop() removes it. Both are **O(1)**.

Q24 — Deque (double-ended queue) basics

Problem: Demonstrate basic deque operations (push/pop at both ends).

```

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

int main() {
    deque<int> dq;
    dq.push_back(1);
    dq.push_front(2);
    cout << dq.front() << " " << dq.back() << endl;
    dq.pop_front();
    dq.pop_back();
    cout << "Size = " << dq.size() << endl;
    return 0;
}

```

Explanation: deque supports $O(1)$ average push/pop at both ends. Useful for sliding window problems.

Q25 — Implement Stack using Two Queues (brief)

Problem: Use two queues to simulate stack push and pop.

```
#include <iostream>

#include <queue>

using namespace std;

struct StackUsingQueues {
    queue<int> q1, q2;

    void push(int x) {
        q2.push(x);
        while (!q1.empty()) { q2.push(q1.front()); q1.pop(); }
        swap(q1, q2);
    }

    int pop() {
        if (q1.empty()) return -1;
        int v = q1.front(); q1.pop(); return v;
    }

    int top() { return q1.empty() ? -1 : q1.front(); }

    bool empty() { return q1.empty(); }
};

int main() {
    StackUsingQueues s;

    s.push(10); s.push(20); s.push(30);

    cout << s.pop() << endl; // 30
```

```
cout << s.top() << endl; // 20  
return 0;  
}
```

Explanation: push is made costly: new element enqueued to q2, then all q1 moved to q2 and queues swapped — ensures LIFO order in q1. pop is $O(1)$. Amortized complexities depend on chosen approach.

Unit 3 — Linked Lists (Q36–Q55)

Q36 — Traverse a Singly Linked List

```
#include <iostream>

using namespace std;

struct Node {
    int data;
    Node* next;
    Node(int val): data(val), next(nullptr) {}
};

void traverse(Node* head) {
    Node* temp = head;
    while(temp) {
        cout << temp->data << " ";
        temp = temp->next;
    }
    cout << endl;
}

int main() {
    Node* head = new Node(1);
    head->next = new Node(2);
    head->next->next = new Node(3);
    traverse(head);
}
```

Explanation: Iterate through each node using next; $O(n)$ time.

Q37 — Search in Linked List

```

#include <iostream>

using namespace std;

struct Node { int data; Node* next; Node(int val):data(val),next(nullptr){} };

bool search(Node* head, int key) {
    Node* temp = head;
    while(temp){
        if(temp->data==key) return true;
        temp = temp->next;
    }
    return false;
}

int main(){
    Node* head = new Node(5);
    head->next = new Node(10);
    cout << (search(head,10)?"Found":"Not Found") << endl;
}

```

Explanation: Linear search in linked list; $O(n)$.

Q38 — Insert at Beginning

```

#include <iostream>

using namespace std;

struct Node { int data; Node* next; Node(int v):data(v),next(nullptr){} };

void insertBegin(Node*& head, int val){
    Node* n = new Node(val);

```

```

n->next = head;

head = n;
}

```

```

int main(){
    Node* head = nullptr;
    insertBegin(head, 10);
    insertBegin(head, 20);
    Node* temp = head;
    while(temp){ cout << temp->data << " "; temp=temp->next;}
}

```

Explanation: $O(1)$ insertion at head.

Q39 — Insert at End

```

#include <iostream>
using namespace std;

```

```

struct Node { int data; Node* next; Node(int v):data(v),next(nullptr){} };

```

```

void insertEnd(Node*& head, int val){
    Node* n = new Node(val);
    if(!head){ head=n; return;}
    Node* temp=head;
    while(temp->next) temp=temp->next;
    temp->next=n;
}

```

```

int main(){
    Node* head=nullptr;

```

```

insertEnd(head,5);

insertEnd(head,15);

Node* temp=head;

while(temp){ cout<<temp->data<<" "; temp=temp->next;}

}

```

Explanation: Traverse to end and append; $O(n)$.

Q40 — Delete a Node by Value

```

#include <iostream>

using namespace std;

struct Node { int data; Node* next; Node(int v):data(v),next(nullptr){} };

void deleteNode(Node*& head, int val){

    if(!head) return;

    if(head->data==val){ Node* t=head; head=head->next; delete t; return;}

    Node* temp=head;

    while(temp->next && temp->next->data!=val) temp=temp->next;

    if(temp->next){ Node* t=temp->next; temp->next=temp->next->next; delete t;}

}

int main(){

    Node* head=new Node(1);

    head->next=new Node(2);

    deleteNode(head,1);

    Node* temp=head;

    while(temp){ cout<<temp->data<<" "; temp=temp->next;}

}

```

Explanation: $O(n)$ deletion by value.

Q41 — Count Nodes in Linked List

```
#include <iostream>

using namespace std;

struct Node { int data; Node* next; Node(int v):data(v),next(nullptr){} };

int countNodes(Node* head){

    int cnt=0;

    while(head){ cnt++; head=head->next;}

    return cnt;

}

int main(){

    Node* head=new Node(5);

    head->next=new Node(10);

    cout << "Node count: " << countNodes(head) << endl;

}
```

Explanation: Simple traversal; $O(n)$.

Q42 — Reverse a Linked List Iteratively

```
#include <iostream>

using namespace std;

struct Node { int data; Node* next; Node(int v):data(v),next(nullptr){} };

Node* reverseliter(Node* head){

    Node *prev=nullptr, *curr=head, *next;

    while(curr){ next=curr->next; curr->next=prev; prev=curr; curr=next;}

    return prev;

}
```

```
}
```

```
int main(){  
    Node* head=new Node(1);  
    head->next=new Node(2);  
    head->next->next=new Node(3);  
    head=reverselter(head);  
    Node* temp=head;  
    while(temp){ cout<<temp->data<<" "; temp=temp->next;}  
}
```

Explanation: Iterative reversal; $O(n)$ time, $O(1)$ space.

Q43 — Reverse a Linked List Recursively

```
#include <iostream>
```

```
using namespace std;
```

```
struct Node{ int data; Node* next; Node(int v):data(v),next(nullptr){} };
```

```
Node* reverseRec(Node* head){  
    if(!head || !head->next) return head;  
    Node* rest=reverseRec(head->next);  
    head->next->next=head;  
    head->next=nullptr;  
    return rest;  
}
```

```
int main(){  
    Node* head=new Node(1);  
    head->next=new Node(2);
```

```

head->next->next=new Node(3);

head=reverseRec(head);

Node* temp=head;

while(temp){ cout<<temp->data<<" "; temp=temp->next;}

}

```

Explanation: Recursive reversal; $O(n)$ time, $O(n)$ stack.

Q44 — Detect Loop in Linked List (Floyd's Cycle)

```

#include <iostream>

using namespace std;

struct Node{ int data; Node* next; Node(int v):data(v),next(nullptr){} };

bool detectLoop(Node* head){

    Node *slow=head,*fast=head;

    while(fast && fast->next){

        slow=slow->next;

        fast=fast->next->next;

        if(slow==fast) return true;

    }

    return false;

}

int main(){

    Node* head=new Node(1);

    head->next=new Node(2);

    head->next->next=head; // create loop

    cout << (detectLoop(head)?"Loop detected":"No loop") << endl;

}

```

Explanation: Floyd's cycle detection; $O(n)$ time, $O(1)$ space.

Q45 — Merge Two Sorted Linked Lists

```
#include <iostream>
```

```
using namespace std;
```

```
struct Node{ int data; Node* next; Node(int v):data(v),next(nullptr){} };
```

```
Node* mergeSorted(Node* a, Node* b){  
    if(!a) return b;  
    if(!b) return a;  
    if(a->data<b->data){ a->next=mergeSorted(a->next,b); return a;}  
    else { b->next=mergeSorted(a,b->next); return b;}  
}
```

```
int main(){  
    Node* a=new Node(1); a->next=new Node(3);  
    Node* b=new Node(2); b->next=new Node(4);  
    Node* head=mergeSorted(a,b);  
    while(head){ cout<<head->data<<" "; head=head->next;}  
}
```

Explanation: Merge step similar to merge sort; $O(n+m)$.

Q46 — Find Middle of Linked List

```
#include <iostream>
```

```
using namespace std;
```

```
struct Node{ int data; Node* next; Node(int v):data(v),next(nullptr){} };
```

```
Node* findMiddle(Node* head){
```

```

Node *slow=head,*fast=head;

while(fast && fast->next){ slow=slow->next; fast=fast->next->next;}

return slow;
}

```

```

int main(){

    Node* head=new Node(1);

    head->next=new Node(2);

    head->next->next=new Node(3);

    cout << "Middle: " << findMiddle(head)->data << endl;

}

```

Explanation: Slow-fast pointer; $O(n)$.

Q47 — Remove Duplicates from Sorted Linked List

```

#include <iostream>

```

```

using namespace std;

```

```

struct Node{ int data; Node* next; Node(int v):data(v),next(nullptr){} };

```

```

void removeDuplicates(Node* head){

    Node* curr=head;

    while(curr && curr->next){

        if(curr->data==curr->next->data){

            Node* t=curr->next; curr->next=curr->next->next; delete t;

        }

    }
}

```

Unit 4 — Searching & Sorting (Q56–Q80)

Q56 — Interpolation Search (Advanced)

```
#include <iostream>

using namespace std;

int interpolationSearch(int a[], int n, int x) {
    int low = 0, high = n - 1;
    while (low <= high && x >= a[low] && x <= a[high]) {
        if (a[low] == a[high]) {
            return (a[low] == x) ? low : -1;
        }
        int pos = low + (double)(high - low) * (x - a[low]) / (a[high] - a[low]);
        if (a[pos] == x) return pos;
        if (a[pos] < x) low = pos + 1;
        else high = pos - 1;
    }
    return -1;
}

int main() {
    int a[] = {10,20,30,40,50,60};
    cout << "Index of 40: " << interpolationSearch(a,6,40) << endl;
}
```

Explanation: Best for uniformly distributed arrays; average $O(\log \log n)$.

Q57 — Bubble Sort (Standard)

```
#include <iostream>

using namespace std;
```

```

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

```

```

int main(){
    int a[]={5,1,4,2,8};
    bubbleSort(a,5);
    for(int x:a) cout<<x<<" ";
}

```

Explanation: $O(n^2)$; stable sort.

Q58 — Bubble Sort Optimized

```

#include <iostream>
using namespace std;

void bubbleSortOptimized(int a[], int n){
    for(int i=0;i<n-1;i++){
        bool swapped=false;
        for(int j=0;j<n-i-1;j++){
            if(a[j]>a[j+1]){swap(a[j],a[j+1]);swapped=true;}
        }
        if(!swapped) break;
    }
}

```

```
int main(){
    int a[]={5,1,4,2,8};
    bubbleSortOptimized(a,5);
    for(int x:a) cout<<x<<" ";
}
```

Explanation: Early exit if already sorted; best $O(n)$.

Q59 — Selection Sort

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

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

```
int main(){
    int a[]={3,1,5,2,4};
    selectionSort(a,5);
    for(int x:a) cout<<x<<" ";
}
```

Explanation: Always $O(n^2)$; not stable.

Q60 — Insertion Sort

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

```

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

```

```

int main(){
    int a[]={9,7,5,3,1};
    insertionSort(a,5);
    for(int x:a) cout<<x<<" ";
}

```

Explanation: $O(n)$ best-case, $O(n^2)$ worst-case; stable.

Q61 — Quick Sort (Pivot = Last Element)

```

#include <iostream>
using namespace std;

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

```

```

void quickSort(int a[], int low, int high){
    if(low<high){
        int p=partition(a,low,high);
        quickSort(a,low,p-1);
        quickSort(a,p+1,high);
    }
}

```

```

int main(){
    int a[]={8,4,7,2,6};
    quickSort(a,0,4);
    for(int x:a) cout<<x<<" ";
}

```

Explanation: Average $O(n \log n)$, worst $O(n^2)$.

Q62 — Merge Sort

```

#include <iostream>
using namespace std;

```

```

void merge(int a[], int l, int m, int r){
    int n1=m-l+1,n2=r-m;
    int L[n1],R[n2];
    for(int i=0;i<n1;i++) L[i]=a[l+i];
    for(int i=0;i<n2;i++) R[i]=a[m+1+i];
    int i=0,j=0,k=l;
    while(i<n1 && j<n2) a[k++]=(L[i]<=R[j])?L[i++]:R[j++];
    while(i<n1) a[k++]=L[i++];
    while(j<n2) a[k++]=R[j++];
}

```

```

void mergeSort(int a[], int l, int r){
    if(l<r){
        int m=l+(r-l)/2;
        mergeSort(a,l,m);
        mergeSort(a,m+1,r);
        merge(a,l,m,r);
    }
}

```

```

int main(){
    int a[]={6,3,9,5,2};
    mergeSort(a,0,4);
    for(int x:a) cout<<x<<" ";
}

```

Explanation: Stable, $O(n \log n)$, needs $O(n)$ extra space.

Q63 — Counting Sort

```

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

void countingSort(vector<int>& a){
    int mx=*max_element(a.begin(),a.end());
    vector<int> cnt(mx+1,0);
    for(int x:a) cnt[x]++;
    int idx=0;
    for(int i=0;i<=mx;i++) while(cnt[i]--) a[idx++]=i;
}

```

```
int main(){
    vector<int> a={4,2,2,8,3,3,1};
    countingSort(a);
    for(int x:a) cout<<x<<" ";
}
```

Explanation: $O(n+k)$, stable.

Q64 — Radix Sort

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

int getMax(vector<int>& a){ return *max_element(a.begin(),a.end()); }

void countSort(vector<int>& a,int exp){
    int n=a.size();
    vector<int> out(n,cnt(10,0));
    for(int i=0;i<n;i++) cnt[(a[i]/exp)%10]++;
    for(int i=1;i<10;i++) cnt[i]+=cnt[i-1];
    for(int i=n-1;i>=0;i--) out[--cnt[(a[i]/exp)%10]]=a[i];
    a=out;
}

void radixSort(vector<int>& a){
    int m=getMax(a);
    for(int exp=1;m/exp>0;exp*=10) countSort(a,exp);
}
```

```

int main(){
    vector<int> a={170,45,75,90,802,24,2,66};
    radixSort(a);
    for(int x:a) cout<<x<<" ";
}

```

Explanation: $O(d*(n+k))$, stable; d = max digits.

Q65 — Sort Strings Alphabetically

```

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

int main(){
    vector<string> v={"Ravi","Amit","Neha","Deep"};
    sort(v.begin(),v.end());
}

```