```
// ==============================
// Bubble Sort
// ==============================
// Bubble Sort repeatedly steps through the list, compares adjacent elements and swaps them if they are in the wrong order.
void bubbleSort(vector<int>& arr) {
    int n = arr.size() - 1;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n - i; j++) {
            if (arr[j] > arr[j + 1])
                swap(arr[j], arr[j + 1]); // Swap if elements are out of order
        }
    }
}
// Time: O(n^2), Space: O(1)
// Use case: Simple and small data sets where performance is not critical.
// Advantages: Easy to implement, intuitive.
// Disadvantages: Very slow on large datasets.


// ==============================
// Insertion Sort
// ==============================
// Insertion Sort builds the sorted array one item at a time by inserting elements into their correct position.
void insertionSort(vector<int>& arr) {
    int n = arr.size();
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;
        // Move elements greater than key to one position ahead
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key; // Insert key at correct position
    }
}
// Time: O(n^2), Space: O(1)
// Use case: Nearly sorted arrays.
// Advantages: Efficient for small or nearly sorted arrays.
// Disadvantages: Poor performance on large or random arrays.


// ==============================
// Selection Sort
// ==============================
// Selection Sort repeatedly finds the minimum element from the unsorted part and puts it at the beginning.
void selectionSort(vector<int>& arr) {
    int n = arr.size();
    for (int i = 0; i < n - 1; i++) {
        int min_idx = i;
        for (int j = i + 1; j < n; ++j)
            if (arr[j] < arr[min_idx])
            min_idx = j;
        swap(arr[i], arr[min_idx]); // Swap found minimum with first element
    }
}
// Time: O(n^2), Space: O(1)
// Use case: Simple small-size data, educational purposes.
// Advantages: Easy to understand and implement.
// Disadvantages: Inefficient on large lists.


// ==============================
// Merge Sort (Divide and Conquer)
// ==============================
// Merge Sort divides the array into halves, sorts each half, and merges them.
void merge(vector<int>& arr, int l, int m, int r) {
    vector<int> L(arr.begin() + l, arr.begin() + m + 1);
    vector<int> R(arr.begin() + m + 1, arr.begin() + r + 1);
    int i = 0, j = 0, k = l;
    // Merge the two halves
    while (i < L.size() && j < R.size())
        arr[k++] = (L[i] <= R[j]) ? L[i++] : R[j++];
    while (i < L.size()) arr[k++] = L[i++];
    while (j < R.size()) arr[k++] = R[j++];
}

void mergeSort(vector<int>& arr, int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2;
        mergeSort(arr, l, m);      // Sort left half
        mergeSort(arr, m + 1, r);  // Sort right half
        merge(arr, l, m, r);       // Merge sorted halves
    }
}

// psuedocode
MERGE_SORT(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left = MERGE_SORT(arr[0:mid])
    right = MERGE_SORT(arr[mid:])
    return MERGE(left, right)

MERGE(left, right):
    result = []
    while left and right:
        if left[0] <= right[0]:
            result.append(left.pop(0))
        else:
            result.append(right.pop(0))
    return result + left + right
// Time: O(n log n), Space: O(n)
// Use case: Efficient for large datasets, stable sort.
// Advantages: Stable, consistent O(n log n) performance.
// Disadvantages: Requires O(n) extra space.
// ============================
// Quick Sort (Divide and Conquer)
// ============================
// Quick Sort picks a pivot, partitions the array, and recursively sorts the partitions.
int partition(vector<int>& arr, int low, int high) {
    //pick a random pivot
    int pivotIndex = low + rand() % (high - low + 1);
    swap(nums[pivotIndex], nums[high]); // Move pivot to end

    int pivot = arr[high];
    int i = low - 1;
    for (int j = low; j < high; j++) {
        if (arr[j] < pivot)
            swap(arr[++i], arr[j]); // Place smaller elements before pivot
    }
    swap(arr[i + 1], arr[high]); // Place pivot in correct position
    return i + 1;
}

void quickSort(vector<int>& arr, int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high); // Partition index
        quickSort(arr, low, pi - 1);        // Sort left partition
        quickSort(arr, pi + 1, high);       // Sort right partition
    }
}
// Time: O(n log n) avg, O(n^2) worst; Space: O(log n)
// Use case: Efficient average-case performance, commonly used in libraries.
// Advantages: Very fast in practice, in-place.
// Disadvantages: Worst-case O(n^2), not stable.


// ============================
// Kruskal's Algorithm
// ============================
// Kruskal's Algorithm finds the Minimum Spanning Tree using union-find.
struct Edge {
    int u, v, weight;
    bool operator<(const Edge& other) const {
        return weight < other.weight;
    }
};

int find(vector<int>& parent, int i) {
    if (parent[i] != i) parent[i] = find(parent, parent[i]); // Path compression
    return parent[i];
}

void unionSets(vector<int>& parent, vector<int>& rank, int u, int v) {
    int rootU = find(parent, u), rootV = find(parent, v);
    if (rootU != rootV) {
        if (rank[rootU] < rank[rootV])
            parent[rootU] = rootV;
        else if (rank[rootU] > rank[rootV])
            parent[rootV] = rootU;
        else {
            parent[rootV] = rootU;
            rank[rootU]++;
        }
    }
}

int kruskal(int V, vector<Edge>& edges) {
    sort(edges.begin(), edges.end()); // Sort edges by weight
    vector<int> parent(V), rank(V, 0);
    iota(parent.begin(), parent.end(), 0); // Initialize parent array
    int mst_weight = 0;
    for (auto& edge : edges) {
        if (find(parent, edge.u) != find(parent, edge.v)) {
            unionSets(parent, rank, edge.u, edge.v); // Union the sets
            mst_weight += edge.weight;               // Add edge to MST
        }
    }
    return mst_weight;
}
// Time: O(E log E), Space: O(V)
// Use case: Finding minimum spanning tree in sparse graphs.
// Advantages: Simple to implement with union-find.
// Disadvantages: Not efficient for dense graphs.


// ==============================
// Prim's Algorithm
// ==============================
// Prim's Algorithm grows the MST by adding the smallest edge from the tree to a new vertex.
int prim(int V, vector<vector<pair<int, int>>>& adj) {
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<>> pq;
    vector<bool> inMST(V, false);
    pq.push({0, 0}); // {weight, vertex}
    int total_weight = 0;

    while (!pq.empty()) {
        auto [weight, u] = pq.top(); pq.pop();
        if (inMST[u]) continue; // Skip if already in MST
        inMST[u] = true;
        total_weight += weight;
        for (auto& [v, w] : adj[u]) {
            if (!inMST[v]) pq.push({w, v}); // Add adjacent edges
        }
    }
    return total_weight;
}
// Time: O(E log V), Space: O(V)
// Use case: Minimum spanning tree in dense graphs.
// Advantages: Efficient with priority queue, better for dense graphs.
// Disadvantages: Harder to implement than Kruskal in some cases.


// ==============================
// Dijkstra's Algorithm
// =============================
// Dijkstra's Algorithm finds the shortest path from a source to all vertices in a graph with non-negative weights.
vector<int> dijkstra(int V, vector<vector<pair<int, int>>>& adj, int src) {
    vector<int> dist(V, INT_MAX);
    dist[src] = 0;
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<>> pq;
    pq.push({0, src});

    while (!pq.empty()) {
        auto [d, u] = pq.top(); pq.pop();
        if (d > dist[u]) continue; // Skip if not optimal
        for (auto& [v, w] : adj[u]) {
            if (dist[u] + w < dist[v]) {
                dist[v] = dist[u] + w; // Update distance
                pq.push({dist[v], v});
            }
        }
    }
    return dist;
}
// Time: O(E log V), Space: O(V)
// Use case: Single-source shortest path on graphs with non-negative weights.
// Advantages: Fast with heaps, supports dynamic graphs.
// Disadvantages: Doesn't work with negative weights.


// =============================
// Min Heap (Priority Queue)
// =============================
// MinHeap supports efficient insertion, removal, and retrieval of the minimum element.
class MinHeap {
    vector<int> heap;

    void heapifyUp(int idx) {
        while (idx > 0 && heap[(idx - 1) / 2] > heap[idx]) {
            swap(heap[(idx - 1) / 2], heap[idx]);
            idx = (idx - 1) / 2;
        }
    }

    void heapifyDown(int idx) {
        int size = heap.size();
        while (2 * idx + 1 < size) {
            int smallest = idx, left = 2 * idx + 1, right = 2 * idx + 2;
            if (left < size && heap[left] < heap[smallest]) smallest = left;
            if (right < size && heap[right] < heap[smallest]) smallest = right;
            if (smallest == idx) break;
            swap(heap[idx], heap[smallest]);
            idx = smallest;
        }
    }

public:
    void push(int val) {
        heap.push_back(val);
        heapifyUp(heap.size() - 1); // Maintain heap property
    }

    void pop() {
        if (heap.empty()) return;
        heap[0] = heap.back();
        heap.pop_back();
        heapifyDown(0); // Restore heap property
    }

    int top() {
        return heap.empty() ? -1 : heap[0];
    }

    bool empty() {
        return heap.empty();
    }
};
// Time: O(log n) for insert/delete, O(1) for getMin, O(n) for search or heapify
// Space: O(n)
// Use case: Priority queues, Dijkstra's and Prim's algorithms, job scheduling, merging K sorted lists
// Advantages:
//   - Efficient access to the minimum element (O(1))
//   - Dynamic structure (resizes easily)
//   - Heapify builds a heap in linear time (O(n))
// Disadvantages:
//   - Searching arbitrary elements is O(n)
//   - Only the minimum element is directly accessible
//   - Poor cache performance compared to arrays



//priority queue
// KthLargest maintains the k largest elements in a min-heap.
class KthLargest {
private:
    priority_queue<int, vector<int>, greater<int>> minHeap;
    int k;

public:
    KthLargest(int k, vector<int>& nums) {
        this->k = k;
        for(int num : nums) {
            minHeap.push(num);
            if(minHeap.size() > k) {
                minHeap.pop(); // Keep only k largest elements
            }
        }
    }

    int add(int val) {
        minHeap.push(val);
        if(minHeap.size() > k) {
            minHeap.pop();
        }
        return minHeap.top(); // Return kth largest
    }
};

/**
 * Your KthLargest object will be instantiated and called as such:
 * KthLargest* obj = new KthLargest(k, nums);
 * int param_1 = obj->add(val);
 */

// Time: O(log n) for push/pop, O(1) for top
// Use case: Priority queue, efficient scheduling, graph algorithms
// Advantages: Efficient insertions and removals
// Disadvantages: Tree-based structure more complex than array


// ==============================
// Huffman Encoding
// ==============================
// Huffman Encoding builds an optimal prefix code for lossless data compression.
struct HuffmanNode {
    char ch;
    int freq;
    HuffmanNode *left, *right;
    HuffmanNode(char c, int f) : ch(c), freq(f), left(nullptr), right(nullptr) {}
};

struct Compare {
    bool operator()(HuffmanNode* a, HuffmanNode* b) {
        return a->freq > b->freq;
    }
};

void generateCodes(HuffmanNode* root, string str, unordered_map<char, string>& huffmanCode) {
    if (!root) return;
    if (!root->left && !root->right) huffmanCode[root->ch] = str; // Leaf node
    generateCodes(root->left, str + "0", huffmanCode);
    generateCodes(root->right, str + "1", huffmanCode);
}

unordered_map<char, string> huffmanEncoding(string text) {
    unordered_map<char, int> freq;
    for (char ch : text) freq[ch]++; // Count frequencies

    priority_queue<HuffmanNode*, vector<HuffmanNode*>, Compare> pq;
    for (auto& p : freq)
        pq.push(new HuffmanNode(p.first, p.second));

    // Build Huffman Tree
    while (pq.size() > 1) {
        HuffmanNode *left = pq.top(); pq.pop();
        HuffmanNode *right = pq.top(); pq.pop();
        HuffmanNode *node = new HuffmanNode('$', left->freq + right->freq);
        node->left = left;
        node->right = right;
        pq.push(node);
    }

    unordered_map<char, string> huffmanCode;
    generateCodes(pq.top(), "", huffmanCode); // Generate codes
    return huffmanCode;
}
// Time: O(n log n), Space: O(n)
// Use case: Lossless compression.
// Advantages: Produces optimal prefix codes, saves storage.
// Disadvantages: Overhead of maintaining tree, less useful for short strings.


// ==============================
// Binary Search - First Occurrence
// ==============================
// Finds the first occurrence of a target value in a sorted array.
int binarySearchFirstOccurrence(vector<int>& arr, int target) {
    int left = 0, right = arr.size() - 1, result = -1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (arr[mid] == target) {
            result = mid;      // Record index and search left half
            right = mid - 1;
        } else if (arr[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    return result;
}
// Time: O(log n), Space: O(1)
// Finds the first occurrence of a number in a sorted array.
// Advantages: Fast in sorted arrays. Disadvantages: Only works on sorted arrays.


// ==============================
// Binary Search - Rotation Point
// ==============================
// Finds the index of the smallest value in a rotated sorted array.
int findRotationPoint(vector<int>& arr) {
    int left = 0, right = arr.size() - 1;
    while (left < right) {
        int mid = left + (right - left) / 2;
        if (arr[mid] > arr[right]) {
            left = mid + 1; // Rotation point is to the right
        } else {
            right = mid;    // Rotation point is at mid or to the left
        }
    }
    return left;
}
// Time: O(log n), Space: O(1)
// Finds index of the smallest value in a rotated sorted array.
// Advantage: Efficient. Disadvantage: Assumes rotated sorted array.


// ==============================
// Linear Search
// ==============================
// Searches for a target value in an array by checking each element.
int linearSearch(vector<int>& arr, int target) {
    for (int i = 0; i < arr.size(); ++i) {
        if (arr[i] == target) return i; // Return index if found
    }
    return -1;
}
// Time: O(n), Space: O(1)
// Simple search through an unsorted array.
// Advantage: Works on any array. Disadvantage: Slow on large arrays.


// ==============================
// Ternary Search (for unimodal function max/min)
// ==============================
// Ternary Search finds the maximum or minimum of a unimodal function.
double ternarySearch(function<double(double)> f, double left, double right, double eps = 1e-6) {
    while (right - left > eps) {
        double m1 = left + (right - left) / 3;
        double m2 = right - (right - left) / 3;
        if (f(m1) < f(m2)) {
            left = m1;
        } else {
            right = m2;
        }
    }
    return (left + right) / 2; // Approximate extremum
}
// Time: O(log n), Space: O(1)
// Used to find max/min of unimodal function.
// Advantage: Effective for continuous unimodal functions. Disadvantage: Not applicable for general
search.


// ==============================
// Exponential Search
// ==============================
// Exponential Search finds the range where the target may exist, then uses binary search.
int exponentialSearch(vector<int>& arr, int target) {
    if (arr[0] == target) return 0;
    int i = 1;
    while (i < arr.size() && arr[i] <= target)
        i *= 2;
    int left = i / 2, right = min(i, (int)arr.size() - 1);
    // Binary search in found range
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (arr[mid] == target) return mid;
        else if (arr[mid] < target) left = mid + 1;
        else right = mid - 1;
    }
    return -1;
}
// Time: O(log i), Space: O(1), where i is the position of the target
// Efficient for unbounded or large sorted arrays
// Advantage: Great for unbounded sorted arrays. Disadvantage: Requires sorted input.


// ==============================
// Count Inversions using Merge Sort
// ==============================
// Counts the number of inversions in an array using a modified merge sort.
int mergeAndCount(vector<int>& arr, int l, int m, int r) {
    vector<int> left(arr.begin() + l, arr.begin() + m + 1);
    vector<int> right(arr.begin() + m + 1, arr.begin() + r + 1);
    int i = 0, j = 0, k = l, inv_count = 0;
    while (i < left.size() && j < right.size()) {
        if (left[i] <= right[j]) {
            arr[k++] = left[i++];
        } else {
            arr[k++] = right[j++];
            inv_count += (left.size() - i); // Count inversions
        }
    }
    while (i < left.size()) arr[k++] = left[i++];
    while (j < right.size()) arr[k++] = right[j++];
    return inv_count;
}

int countInversions(vector<int>& arr, int l, int r) {
    int inv_count = 0;
    if (l < r) {
        int m = l + (r - l) / 2;
        inv_count += countInversions(arr, l, m);
        inv_count += countInversions(arr, m + 1, r);
        inv_count += mergeAndCount(arr, l, m, r);
    }
    return inv_count;
}
// Time: O(n log n), Space: O(n)
// Counts the number of inversions in an array.
// Advantage: Efficient for large arrays. Disadvantage: More complex implementation.


// ==============================
// Minimum Spanning Tree - Explanation
// ==============================
// A Minimum Spanning Tree (MST) connects all vertices with minimum total weight.
// There can be more than one MST in a graph if there are edges with equal weight.
// Proof:
// - If two different edges have the same weight and swapping them doesn't disconnect the tree, then both are valid in an MST.
// Example: Graph with 4 vertices, edges (0-1, 1), (1-2, 1), (2-3, 1), (0-3, 1). Any 3 of these form a valid
```

MST.

// Use Kruskal's or Prim's to find MSTs.
// Total weight will be the same for all MSTs, but edge selection may differ.

```cpp
// ==============================
// Closest Pair of Points (Divide and Conquer)
// ==============================
// Finds the smallest distance between any two points in a 2D plane using divide and conquer.
#include <bits/stdc++.h>
using namespace std;

struct Point {
    int x, y;
};

bool compareX(Point a, Point b) { return a.x < b.x; }
bool compareY(Point a, Point b) { return a.y < b.y; }

double dist(Point a, Point b) {
    return sqrt((a.x - b.x) * (a.x - b.x) + (a.y - b.y) * (a.y - b.y));
}

double closestPairUtil(vector<Point>& Px, vector<Point>& Py) {
    int n = Px.size();
    if (n <= 3) {
        double minDist = DBL_MAX;
        for (int i = 0; i < n; ++i)
            for (int j = i + 1; j < n; ++j)
                minDist = min(minDist, dist(Px[i], Px[j]));
        return minDist;
    }

    int mid = n / 2;
    Point midPoint = Px[mid];

    vector<Point> Pyl, Pyr;
    for (auto p : Py) {
        if (p.x <= midPoint.x) Pyl.push_back(p);
        else Pyr.push_back(p);
    }

    vector<Point> left(Px.begin(), Px.begin() + mid);
    vector<Point> right(Px.begin() + mid, Px.end());

    double dl = closestPairUtil(left, Pyl);
    double dr = closestPairUtil(right, Pyr);
    double d = min(dl, dr);

    vector<Point> strip;
    for (auto p : Py) if (abs(p.x - midPoint.x) < d) strip.push_back(p);

    double minStripDist = d;
    for (int i = 0; i < strip.size(); ++i)
```

```
        for (int j = i + 1; j < strip.size() && (strip[j].y - strip[i].y) < d; ++j)
            minStripDist = min(minStripDist, dist(strip[i], strip[j]));

    return minStripDist;
}

double closestPair(vector<Point>& points) {
    vector<Point> Px = points, Py = points;
    sort(Px.begin(), Px.end(), compareX);
    sort(Py.begin(), Py.end(), compareY);
    return closestPairUtil(Px, Py);
}
// Time: O(n log n), Space: O(n)
// Finds the smallest distance between any two points in a 2D plane.
// Advantage: Much faster than brute-force (O(n^2)). Disadvantage: Complex to implement.


// ==============================
// Brute Force clock problem
// ==============================
// Each clock is represented as 0 (12 o'clock), 1 (3), 2 (6), 3 (9)
const int SWITCHES[4][4] = {
    {0, 2, 3, 4},  // Switch 1 affects clocks 0,2,3,4
    {1, 2, 5, 8},  // Switch 2 affects clocks 1,2,5,8
    {1, 4, 6, 7},  // Switch 3 affects clocks 1,4,6,7
    {0, 5, 6, 8}   // Switch 4 affects clocks 0,5,6,8
};

// Apply switch effects
void applySwitch(vector<int>& clocks, int sw, int times) {
    for (int i = 0; i < 4; ++i) {
        int idx = SWITCHES[sw][i];
        clocks[idx] = (clocks[idx] + times) % 4; // Advance clock by times*3 hours
    }
}

// Check if all clocks are at 12 (i.e., 0)
bool allReset(const vector<int>& clocks) {
    for (int c : clocks)
        if (c != 0) return false;
    return true;
}


// ==============================
// Uniform distribution
// ==============================
// This will return at most RAND_MAX different values
// equally spaced in the range [a .. b].
double uniform(double a, double b)
{
    return rand() / (RAND_MAX + 1.0) * (b - a) + a;
}
```

```
// ============================
// Implementation of a Least Connections Load Balancer algorithm
// ============================
// This algorithm assigns each new task to the node with the least current load.
#include <iostream>       // For standard input and output
#include <vector>         // To use the vector container (dynamic arrays)
#include <string>         // To use the string data type
#include <limits>         // To get the maximum possible integer value using numeric_limits

using namespace std;

// Define a structure to represent each computing node
struct Node {
    string name;          // Name of the node (e.g., "NodeA")
    int currentLoad;      // Number of tasks currently assigned to this node

    // Constructor to initialize node name and set initial load to 0
    Node(string n) : name(n), currentLoad(0) {}
};

// Function to distribute tasks using the Least Connections strategy
void loadBalance(vector<string> tasks, vector<Node>& nodes) {
    // Loop over each task in the task list
    for (const string& task : tasks) {
        int minLoad = numeric_limits<int>::max();  // Start with a very large value
        int minIndex = 0;                          // Will store the index of the node with the least load

        // Find the node with the smallest current load
        for (int i = 0; i < nodes.size(); ++i) {
            if (nodes[i].currentLoad < minLoad) {
                minLoad = nodes[i].currentLoad;    // Update minimum load
                minIndex = i;                      // Update the index of the node
            }
        }

        // Assign the task to the selected node
        cout << "Assigning task '" << task << "' to node " << nodes[minIndex].name << endl;

        // Increase the load count of the selected node
        nodes[minIndex].currentLoad++;
    }
}


// ============================
// Memoization (Top-Down DP)
// ============================
// Solves problems recursively
// Before solving a subproblem, checks a lookup table (memo) to see if it's already computed

const int NIL = -1;
int lookup[1000]; // assume max n = 999

int fib(int n) {
    if (lookup[n] == NIL) {
        if (n <= 1)
            lookup[n] = n;
        else
            lookup[n] = fib(n - 1) + fib(n - 2);
    }
    return lookup[n];
}

// time complexity = O(n), space complexity = O(n)
// adv = Intuitive and easy to write, Efficient for problems where not all subproblems are needed.
// dis = Recursive: may cause stack overflow for large n, Slower due to function call overhead.


// ============================
// Tabulation (Bottom-Up DP)
// ============================
// Solves subproblems iteratively starting from the base case
// Stores intermediate results in a table and builds up to the final result

int fib(int n) {
    int f[n+1];
    f[0] = 0;
    f[1] = 1;
    for (int i = 2; i <= n; i++)
        f[i] = f[i - 1] + f[i - 2];
    return f[n];
}

// time complexity = O(n), space complexity = O(n) for table but can be optimized to O(1)
// adv = No recursion = no stack overflow risk, Generally faster than memoization (no function call overhead).
// dis = Always computes all subproblems even if not required, Code may be less intuitive than recursion.


// ============================
// Memoization vs Tabulation - Comparison
// ============================
```

Feature	Memoization (Top-Down)	Tabulation (Bottom-Up)
Approach	Recursive	Iterative
When Subproblems Needed?	Only computes required ones	Computes all subproblems
Time Complexity	O(n)	O(n)
Space Complexity	O(n) memo + O(n) call stack	O(n)
Stack Overflow Risk	Yes (deep recursion)	No
Speed	Slightly slower (function calls)	Faster
Ease of Implementation	More intuitive for recursive	Slightly more verbose
Use Case	Sparse subproblems	When all subproblems are useful

```
// ============================
// Rod Cutting Problem - DP Example
// ============================
// Given a rod of length n and a list of prices for each possible piece length, determine the maximum value by cutting the rod into smaller pieces.
int rodCut(int price[], int n) {
    int dp[n+1];
    dp[0] = 0;

    for (int i = 1; i <= n; i++) {
        int max_val = INT_MIN;
        for (int j = 0; j < i; j++)
            max_val = max(max_val, price[j] + dp[i - j - 1]);
        dp[i] = max_val;
    }

    return dp[n];
}

// time complexity = O(n^2) due to nested loop, space complexity = O(n) for the dp array


// ============================
// Generate Exponential Random Numbers
// ============================
// Function to generate exponential random variable using uniform distribution
double exponentialRandom(double lambda) {
    double u = (double)rand() / RAND_MAX; // Uniform [0,1]
    return -log(u) / lambda;
}

int main() {
    srand(time(0)); // Seed the random number generator

    double lambda;
    int n;

    std::cout << "Enter the rate parameter lambda (λ): ";
    std::cin >> lambda;
    std::cout << "Enter the number of exponential random numbers to generate: ";
    std::cin >> n;

    std::cout << "Generated Exponential Random Numbers:\n";
    for (int i = 0; i < n; ++i) {
        std::cout << exponentialRandom(lambda) << std::endl;
    }

    return 0;
}
 // time and space complexities of O(1)


// ============================
// Graph Representation: Adjacency Matrix vs Adjacency List vs Edge List
// ============================
```

1. Adjacency Matrix

Definition:
- A 2D array adj[V][V] where adj[i][j] = 1 (or weight w) if there's an edge from vertex i to vertex j, else 0.

Advantages:
- O(1) time to check if an edge exists between two nodes.
- Easy to implement.

Disadvantages:
- Space inefficient for sparse graphs: uses O(V^2) space.
- Iterating over neighbors takes O(V) time per node.

Space Complexity:
- O(V^2)

Use Case:
- Best for dense graphs or where fast edge lookup is needed.

2. Adjacency List

Definition:
- An array of lists. Each index i has a list of all vertices adjacent to vertex i.

Example:
adj[0] = {1, 3} means edges 0→1 and 0→3.

Advantages:
- Space efficient: O(V + E).
- Efficient neighbor iteration.

Disadvantages:
- Slower edge existence check: O(degree(v)).

Space Complexity:
- O(V + E)

Use Case:
- Best for sparse graphs and typical in real-world applications.

3. Edge List

Definition:
- A simple list of all edges. Each edge is stored as a pair (u, v) or triplet (u, v, w).

Example:
edges = [(0, 1), (0, 3), (1, 2)]

Advantages:
- Very compact for storing only edge data.
- Ideal for algorithms like Kruskal's.

Disadvantages:
- Edge existence check and neighbor lookup is slow: O(E).

Space Complexity:
- O(E)

Use Case:
- Best for edge-centric algorithms (e.g., Kruskal's MST).

Summary Table

Feature	Adjacency Matrix	Adjacency List	Edge List
Edge Lookup	O(1)	O(degree(v))	O(E)
Neighbors Iteration	O(V)	O(degree(v))	O(E)
Space Complexity	O(V^2)	O(V + E)	O(E)
Best For	Dense graphs	Sparse graphs	Edge-based algos
Suitable for Kruskal	No	No	Yes

// ============================
// Parallel Algorithms to Check if x ∈ S (x == Sk for some k)
// ============================
Problem:

Given a sequence S = {S1, S2, ..., Sn} and an integer x,
determine if x equals any element in S using parallel algorithms
under various memory access models.

I) EREW (Exclusive Read Exclusive Write)

- No two processors can read from or write to the same memory cell simultaneously.
- Safe parallelism; must avoid conflicts.

Algorithm:
----
1. Each processor Pi reads its own S[i].
2. Compares S[i] == x.
3. Writes 1 to result[i] if match, else 0.
4. Use parallel reduction (OR) to determine if any result[i] == 1.

Time Complexity:

- Comparison: O(1)
- OR reduction: O(log n)

II) ERCW (Exclusive Read Concurrent Write)

- Each processor reads its unique element.
- All processors write to a shared variable if match occurs.

Algorithm:
----
1. Each Pi reads S[i].
2. If S[i] == x, write 1 to a shared variable `found`.
3. Use concurrent OR or max for conflict resolution.

Time Complexity:

- All steps in O(1)

III) CRCW (Concurrent Read Concurrent Write)

- All processors can read/write the same location concurrently.

Algorithm:
----
1. All processors read x and their own S[i].
2. If S[i] == x, write 1 to a shared `found` variable.
3. Write conflicts resolved using OR or priority write.

Time Complexity:

- O(1)

IV) Tree-Connected SIMD with n Leaves

- Processors connected in a binary tree, each leaf has one element.

Algorithm:
----
1. Each leaf compares S[i] == x and sets a flag.
2. Perform an OR reduction up the tree.

Time Complexity:

- O(log n)

Summary Table:

Model	Read Type	Write Type	Time Complexity
EREW	Exclusive	Exclusive	O(log n)
ERCW	Exclusive	Concurrent	O(1)
CRCW	Concurrent	Concurrent	O(1)
Tree-connected SIMD	Leaf Read	Tree Write	O(log n)

SISD (single instruction stream, single data stream)
MISD (multiple instruction stream, single data stream)
SIMD (single instruction stream, multiple data stream)
MIMD (multiple instruction stream, multiple data stream)

C++ Implementation (Simulated EREW with OpenMP)

```
#include <iostream>
#include <vector>
#include <omp.h>

using namespace std;

bool parallel_search(const vector<int>& S, int x) {
    int n = S.size();
    vector<int> result(n, 0);

    // Step 1 & 2: Compare in parallel
    #pragma omp parallel for
    for (int i = 0; i < n; i++) {
        if (S[i] == x)
            result[i] = 1;
    }
    // Step 3: OR-reduction (serial for simulation)
    int found = 0;
    for (int i = 0; i < n; i++) {
        found |= result[i];
    }

    return found == 1;
}


// ==============================
// Knapsack Problem (Greedy - Fractional Knapsack)
// ==============================
// Structure to store weight and value of an item
struct Item {
    int value, weight;

    // Constructor
    Item(int v, int w) : value(v), weight(w) {}
};

// Comparator to sort items by value-to-weight ratio in descending order
bool cmp(Item a, Item b) {
    double r1 = (double)a.value / a.weight;
    double r2 = (double)b.value / b.weight;
    return r1 > r2;
}

// Function to perform greedy fractional knapsack
double fractionalKnapsack(int W, vector<Item>& items) {
    // Sort items by value-to-weight ratio
    sort(items.begin(), items.end(), cmp);

    double totalValue = 0.0; // Result

    for (Item& item : items) {
        if (W == 0)
            break;

        if (item.weight <= W) {
            // Take the whole item
            W -= item.weight;
            totalValue += item.value;
        } else {
            // Take the fraction of the item
            totalValue += item.value * ((double)W / item.weight);
            W = 0; // Knapsack is full
        }
    }

    return totalValue;
}


// ==============================
// QuickSelect
// ==============================
// Partition function (like Lomuto's partitioning)
int partition(vector<int>& arr, int left, int right) {
    int pivot = arr[right]; // Pivot is last element
    int i = left;           // Place for swapping smaller elements

    for (int j = left; j < right; j++) {
        if (arr[j] <= pivot) {
            swap(arr[i], arr[j]);
            i++;
        }
    }
    swap(arr[i], arr[right]); // Place pivot in correct position
    return i;
}

// QuickSelect function to find kth smallest (1-based index)
int quickSelect(vector<int>& arr, int left, int right, int k) {
    if (left <= right) {
        // Choose a random pivot to avoid worst case
        int pivotIndex = left + rand() % (right - left + 1);
        swap(arr[pivotIndex], arr[right]);

        int pos = partition(arr, left, right);

        // pos is index of the pivot (0-based)
        if (pos == k - 1)
            return arr[pos];
        else if (pos > k - 1)
            return quickSelect(arr, left, pos - 1, k);
        else
            return quickSelect(arr, pos + 1, right, k);
    }

    return -1; // If k is out of bounds
}
```

Asymptotic Notations in Algorithms
====================================

1. Introduction

Asymptotic notations are mathematical tools to describe the running time or space complexity of an algorithm in terms of input size (n) as it approaches infinity.
These notations help in analyzing the efficiency and scalability of algorithms.

Main Types:
- Big-O (O)
- Big-Omega (Ω)
- Big-Theta (Θ)
- Little-o (o)
- Little-omega (ω)

----

2. Big-O Notation (O)

Definition:

    O(f(n)) describes the upper bound of the algorithm's runtime.
    It gives the worst-case growth rate.

Mathematically:

    $T(n) \in O(f(n))$ if $\exists$ constants $c > 0$ and $n_0 \geq 0$ such that
        $T(n) <= c * f(n) \ \forall \ n \geq n_0$

Example:

    If $T(n) = 3n^2 + 5n$, then $T(n) \in O(n^2)$

Use Case:

    Used for worst-case analysis

----

3. Big-Omega Notation (Ω)

Definition:

    Ω(f(n)) gives the lower bound of the algorithm's runtime.
    It represents the best-case growth rate.

Mathematically:

    $T(n) \in \Omega(f(n))$ if $\exists$ constants $c > 0$ and $n_0 \geq 0$ such that
        $T(n) \geq c * f(n) \ \forall \ n \geq n_0$

Example:

    If $T(n) = 2n + 10$, then $T(n) \in \Omega(n)$

Use Case:

    Used for best-case analysis

----

4. Big-Theta Notation (Θ)

Definition:

    Θ(f(n)) tightly bounds the function from above and below.
    It represents the average-case or the exact growth rate.

Mathematically:

    $T(n) \in \Theta(f(n))$ if $\exists$ constants $c1, c2 > 0$ and $n_0 \geq 0$ such that
        $c1 * f(n) <= T(n) <= c2 * f(n) \ \forall \ n \geq n_0$

Example:

    If $T(n) = 4n + 6$, then $T(n) \in \Theta(n)$

Use Case:

    Used for tight bound analysis (both upper and lower)

----

5. Little-o Notation (o)

Definition:
o(f(n)) describes an upper bound that is not tight.
The function grows strictly slower than f(n).

Mathematically:
$T(n) \in o(f(n))$ if for all $c > 0$, $\exists\, n_0 \geq 0$ such that
$T(n) < c * f(n)\ \forall\ n \geq n_0$

Example:
If $T(n) = n$, then $T(n) \in o(n \log n)$

Use Case:
Used when growth is strictly less than f(n)

---

6. Little-omega Notation (ω)

Definition:
ω(f(n)) describes a lower bound that is not tight.
The function grows strictly faster than f(n).

Mathematically:
$T(n) \in \omega(f(n))$ if for all $c > 0$, $\exists\, n_0 \geq 0$ such that
$T(n) > c * f(n)\ \forall\ n \geq n_0$

Example:
If $T(n) = n \log n$, then $T(n) \in \omega(n)$

Use Case:
Used when growth is strictly greater than f(n)

---

7. Common Order of Growth (From Slowest to Fastest)

1. O(1)	Constant time
2. O(log n)	Logarithmic
3. O(n)	Linear
4. O(n log n)	Linearithmic
5. O(n^2)	Quadratic
6. O(n^3)	Cubic
7. O(2^n)	Exponential
8. O(n!)	Factorial

---

8. Notes & Tips

- Focus on dominant terms (e.g., ignore constants and lower order terms).
- Big-O is the most commonly used notation in interviews and analysis.
- Proving Big-Theta gives the tightest performance guarantee.
- Use asymptotic notations to compare algorithms as input size grows.

Question 1:

a) Time Complexity Analysis:
i. Given:

```
def find_pairs(arr):
    n = len(arr)
    for i in range(n):
        for j in range(i + 1, n):
            print(arr[i], arr[j])
```

Analysis: Two nested loops, each up to n => Time Complexity: O(n^2)

ii. Optimization: If only pairs are printed, cannot improve time. If for other logic, use hash set to avoid duplicates or sort to reduce comparisons.

iii. Complexity Types:
- Best-case: Minimum time, e.g., linear search finds item at start -> O(1)
- Worst-case: Maximum time, e.g., item not found -> O(n)
- Average-case: Expected time over all inputs -> O(n/2) ≈ O(n)

b) Statement: 2n + 100n^2 + n^100 = O(n^101)
True. Since n^100 is the dominant term and n^101 bounds it from above.

c) Quickselect:
- Problem: Finds the k-th smallest element
- Avg Time Complexity: O(n)
- Worst-case: O(n^2)

Question 2:

i. Merge Sort Pseudocode:

```
MERGE_SORT(arr):
    if len(arr) <= 1: return arr
    mid = len(arr) // 2
    left = MERGE_SORT(arr[0:mid])
    right = MERGE_SORT(arr[mid:])
    return MERGE(left, right)
```

MERGE(left, right): merge two sorted arrays

Time Complexity: T(n) = 2T(n/2) + O(n) → Master Theorem Case 2 → O(n log n)

ii. Given T(10^4) = 1ms:
T(n) = c * n log n, solve for c:
c ≈ 7.53 x 10^-9

T(10^9) ≈ 7.53 x 10^-9 * 10^9 * log2(10^9) ≈ 225.3 seconds

iii. Sort Comparison:
- Merge Sort: O(n log n) time, O(n) space
- Quick Sort: O(n log n) avg, O(n^2) worst, O(log n) space
- Bubble Sort: O(n^2) time, O(1) space

Question 3:

i. Dijkstra's Algorithm:

```
DIJKSTRA(graph, source):
    dist = [∞] * len(graph)
    dist[source] = 0
    visited = set()
    pq = priority queue with (0, source)

    while pq not empty:
        (d, u) = pq.pop()
        if u in visited: continue
        visited.add(u)
        for (v, w) in neighbors of u:
            if dist[v] > dist[u] + w:
                dist[v] = dist[u] + w
                pq.push((dist[v], v))
```

- Input: Graph, source node
- Output: Shortest path from source to all nodes
- Time Complexity: O((V+E) log V)

ii. Prim's Algorithm:

```
PRIM(graph):
    mst = []
    visited = set()
    pq = priority queue with (0, start_node)

    while pq not empty:
        (w, u, v) = pq.pop()
        if v not in visited:
            visited.add(v)
            mst.append((u, v, w))
            for (v2, w2) in neighbors of v:
                if v2 not in visited:
                    pq.push((w2, v, v2))
```

- Input: Graph
- Output: Minimum Spanning Tree
- Time Complexity: O((V+E) log V)

Question 4:

i. Fractional Knapsack:

```
FRACTIONAL_KNAPSACK(items, maxWeight):
    sort items by value/weight descending
    totalValue = 0
    for item in items:
        if item.weight <= maxWeight:
            totalValue += item.value
            maxWeight -= item.weight
        else:
            totalValue += item.value * (maxWeight / item.weight)
            break
    return totalValue
```

- Sort by value/weight ratio
- Greedily take highest ratio items
- Time Complexity: O(n log n)

ii. Huffman Code for 'e':
Construct Huffman Tree, get binary code for 'e' by tree traversal. Exact code depends on tree.
Use Huffman Tree → Build min-heap, combine lowest frequencies.

After building tree:

Traverse to find 'e'

Path from root gives the code (e.g., 010)

Question 5:

i. Bisection Method Pseudocode:

```
BISECTION(f, a, b, tol):
    while (b - a)/2 > tol:
        c = (a + b)/2
        if f(c) == 0: return c
        elif f(a)*f(c) < 0: b = c
        else: a = c
    return (a + b)/2
```

Input: Function f, interval [a, b], tolerance
Output: Approximate root
Time Complexity: O(log2((b-a)/tol))

ii. Closest to Y-Axis:

```
FIND_CLOSEST_Y_AXIS(points):
    minDist = ∞
    closest = null
    for (x, y) in points:
        if abs(x) < minDist:
            minDist = abs(x)
            closest = (x, y)
    return closest
```

Given: Unsorted points
Goal: Find point with smallest |x|
- Input: list of (x, y)
- Find min |x|: linear scan
- Binary search not useful unless sorted

iii. Generate Passwords Recursively (n letters):

```
void generate(string current, int n) {
    if (current.length() == n) print(current);
    else for (char c = 'A'; c <= 'Z'; ++c)
        generate(current + c, n);
}
```

Time Complexity: O(26^n)

Question 6:

Given a sequence S = {S1, S2, ..., Sn} and x, find if x exists using parallel algorithms:

I) EREW PRAM (Exclusive Read Exclusive Write)

- Each processor Pi handles Si.
- Step 1: All processors read their respective Si and compare with x.
- Step 2: Each Pi stores 1 if Si == x, else 0 (in private memory).
- Step 3: Perform a parallel OR reduction (binary tree style) to check if any processor found x.

Pseudocode:

```
for each processor Pi in parallel do
    local[i] := (S[i] == x) ? 1 : 0

for d := 1 to log2(n) do
    for each processor Pi where i % (2^d) == 0 in parallel do
        local[i] := local[i] OR local[i + 2^(d-1)]

Result is in local[0]
```

Time Complexity: O(log n)

II) ERCW PRAM (Exclusive Read Concurrent Write)

- Each processor Pi handles Si.
- Step 1: All processors read their own Si (exclusive reads).
- Step 2: If Si == x, Pi writes 1 to a shared memory location "found".
- Step 3: Use OR conflict resolution (write 1 if any processor writes 1).

Pseudocode:

```
shared found := 0

for each processor Pi in parallel do
    if S[i] == x then
        found := 1   // Concurrent write with OR resolution

Result is in shared 'found'
```

Time Complexity: O(1)

III) CRCW PRAM (Concurrent Read Concurrent Write)

- All processors can read x and Si concurrently.
- Step 1: All processors read x from shared memory and their own Si.
- Step 2: If Si == x, write 1 to a shared memory location "found".
- Step 3: Use OR resolution for write conflict.

Pseudocode:

```
shared found := 0

for each processor Pi in parallel do
    if S[i] == x then
        found := 1   // Concurrent write with OR resolution

Result is in shared 'found'
```

Time Complexity: O(1)

IV) Tree-Connected SIMD Computer with n Leaves

- Each leaf processor holds one element Si.
- Step 1: Each leaf compares Si to x and returns 1 if equal, 0 otherwise.
- Step 2: Internal nodes of the tree perform an OR operation as results are passed upward.
- Step 3: Root receives the final OR result indicating whether x was found.

Pseudocode:

```
Leaf[i] := (S[i] == x) ? 1 : 0

for each internal node from leaves to root do
    Node.value := OR(left_child.value, right_child.value)

Root.value indicates if x is present in S
```

Time Complexity: O(log n)

Summary Table:

Model	Time Complexity	Notes
EREW	O(log n)	Tree-based OR reduction
ERCW	O(1)	Concurrent write with OR logic
CRCW	O(1)	Full concurrency allowed
Tree-Connected SIMD Tree	O(log n)	Log-depth OR propagation