```
// ============================
// Bubble Sort
// ============================
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]);
        }
    }
}
// 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
// ============================
void insertionSort(vector<int>& arr) {
    int n = arr.size();
    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;
    }
}
// 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
// ============================
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]);
    }
}
// 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)
// ============================
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;
    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);
        mergeSort(arr, m + 1, r);
        merge(arr, l, m, r);
    }
}
// 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)
// ============================
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]);

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

void quickSort(vector<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);
    }
}
```

```cpp
// 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
// ============================
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]);
    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());
    vector<int> parent(V), rank(V, 0);
    iota(parent.begin(), parent.end(), 0);
    int mst_weight = 0;
    for (auto& edge : edges) {
        if (find(parent, edge.u) != find(parent, edge.v)) {
            unionSets(parent, rank, edge.u, edge.v);
            mst_weight += edge.weight;
        }
    }
    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
// ============================
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;
        inMST[u] = true;
        total_weight += weight;
        for (auto& [v, w] : adj[u]) {
            if (!inMST[v]) pq.push({w, v});
        }
    }
    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
// ============================
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;
        for (auto& [v, w] : adj[u]) {
            if (dist[u] + w < dist[v]) {
                dist[v] = dist[u] + w;
                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)
// ============================
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);
    }

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

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

    bool empty() {
        return heap.empty();
    }
};
// 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
// ============================
struct HuffmanNode {
    char ch;
    int freq;
```

```cpp
    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;
    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]++;

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

    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);
    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
// ============================
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;
            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
// ============================
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;
        } else {
            right = mid;
        }
    }
    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
// ============================
int linearSearch(vector<int>& arr, int target) {
    for (int i = 0; i < arr.size(); ++i) {
        if (arr[i] == target) return i;
    }
    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)
// ============================
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;
}
// 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
// ============================
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
// ============================
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);
        }
    }
    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.


// ============================
// Closest Pair of Points (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.
```