








25 Essential Algorithms Every Programmer Should Know

 Curated with  by: `coding_erorr1`

-  1. Binary Search
-  2. Bubble Sort
-  3. Selection Sort
-  4. Insertion Sort
-  5. Merge Sort
-  6. Quick Sort
-  7. Kadane's Algorithm (Maximum Subarray Sum)
-  8. Floyd-Warshall Algorithm (All-Pairs Shortest Path)
-  9. Bellman-Ford Algorithm (Single Source Shortest Path with Negatives)
-  10. 0/1 Knapsack Problem (Dynamic Programming)
-  11. Longest Common Subsequence (LCS)
-  12. Sieve of Eratosthenes (Prime Numbers Generator)
-  13. Depth-First Search (DFS)
-  14. Breadth-First Search (BFS)
-  15. Dijkstra's Algorithm (Shortest Path - Greedy)
-  16. Prim's Algorithm (Minimum Spanning Tree - MST)
-  17. Kruskal's Algorithm (MST Using Disjoint Set)
-  18. Topological Sort (Using DFS - DAG)

-  **19. Cycle Detection in Directed Graph (DFS + Recursion Stack)**
-  **20. Union-Find / Disjoint Set (Path Compression + Union by Rank)**
-  **21. Matrix Chain Multiplication (MCM - DP)**
-  **22. Longest Increasing Subsequence (LIS)**
-  **23. Edit Distance (Minimum Operations to Convert Strings)**
-  **24. Trie – Insert and Search**
-  **25. Sliding Window Maximum (Deque Approach)**

1. Binary Search

Description:

Binary Search is an efficient algorithm to find the position of a target element in a sorted array. It works by repeatedly dividing the search interval in half.

Time Complexity:

- Best: $O(1)$
- Average & Worst: $O(\log n)$

Java Code:

```
java
CopyEdit
int binarySearch(int[] arr, int target) {
    int left = 0, right = arr.length - 1;
    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;
}
```

2. Bubble Sort

Description:

Bubble Sort is a simple sorting algorithm that repeatedly swaps adjacent elements if they are in the wrong order.

Time Complexity:

- Best: $O(n)$
- Average & Worst: $O(n^2)$

Java Code:

```
java
CopyEdit
void bubbleSort(int[] arr) {
    for (int i = 0; i < arr.length - 1; i++) {
        for (int j = 0; j < arr.length - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}
```

3. Selection Sort

Description:

Selection Sort divides the array into a sorted and an unsorted part, and repeatedly selects the smallest element from the unsorted part.

Time Complexity:

- Best, Average & Worst: $O(n^2)$

Java Code:

```
java
CopyEdit
void selectionSort(int[] arr) {
    for (int i = 0; i < arr.length - 1; i++) {
        int minIndex = i;
        for (int j = i + 1; j < arr.length; j++) {
            if (arr[j] < arr[minIndex]) minIndex = j;
        }
        int temp = arr[minIndex];
        arr[minIndex] = arr[i];
        arr[i] = temp;
    }
}
```

}



4. Insertion Sort



Description:

Insertion Sort builds the final sorted array one item at a time by inserting elements into their correct position.



Time Complexity:

- Best: $O(n)$
- Average & Worst: $O(n^2)$



Java Code:

```
java
CopyEdit
void insertionSort(int[] arr) {
    for (int i = 1; i < arr.length; 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;
    }
}
```



5. Merge Sort



Description:

Merge Sort is a divide-and-conquer algorithm that splits the array in half, recursively sorts them, and then merges the sorted halves.



Time Complexity:

- Best, Average & Worst: $O(n \log n)$



Java Code:

```
java
CopyEdit
void mergeSort(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);
    }
}
```

```

}

void merge(int[] arr, int l, int m, int r) {
    int[] left = Arrays.copyOfRange(arr, l, m + 1);
    int[] right = Arrays.copyOfRange(arr, m + 1, r + 1);
    int i = 0, j = 0, k = l;
    while (i < left.length && j < right.length)
        arr[k++] = (left[i] <= right[j]) ? left[i++] : right[j++];
    while (i < left.length) arr[k++] = left[i++];
    while (j < right.length) arr[k++] = right[j++];
}

```

6. Quick Sort

Description:

Quick Sort is another divide-and-conquer algorithm that picks a pivot and partitions the array around the pivot.

Time Complexity:

- Best & Average: $O(n \log n)$
- Worst: $O(n^2)$ (when pivot is poorly chosen)

Java Code:

```

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

```

7. Kadane's Algorithm

Description:

Kadane's Algorithm finds the contiguous subarray with the maximum sum in linear time.

Time Complexity:

- Best, Average, Worst: $O(n)$

Java Code:

```
java
CopyEdit
int maxSubArraySum(int[] arr) {
    int maxSoFar = arr[0], currMax = arr[0];
    for (int i = 1; i < arr.length; i++) {
        currMax = Math.max(arr[i], currMax + arr[i]);
        maxSoFar = Math.max(maxSoFar, currMax);
    }
    return maxSoFar;
}
```

8. Floyd-Warshall Algorithm

Description:

An all-pairs shortest path algorithm using dynamic programming.

Time Complexity:

- $O(n^3)$

Java Code:

```
java
CopyEdit
void floydWarshall(int[][] dist, int V) {
    for (int k = 0; k < V; k++)
        for (int i = 0; i < V; i++)
            for (int j = 0; j < V; j++)
                if (dist[i][k] + dist[k][j] < dist[i][j])
                    dist[i][j] = dist[i][k] + dist[k][j];
}
```

9. Bellman-Ford Algorithm

Description:

Finds shortest paths from a single source to all vertices, handles negative weights.

Time Complexity:

- $O(VE)$

Java Code:

```

java
CopyEdit
void bellmanFord(int[][] edges, int V, int src) {
    int[] dist = new int[V];
    Arrays.fill(dist, Integer.MAX_VALUE);
    dist[src] = 0;

    for (int i = 1; i < V; i++) {
        for (int[] edge : edges) {
            int u = edge[0], v = edge[1], w = edge[2];
            if (dist[u] != Integer.MAX_VALUE && dist[u] + w < dist[v])
                dist[v] = dist[u] + w;
        }
    }
}

```

10. 0/1 Knapsack Problem

Description:

A classic dynamic programming problem where we maximize the value in a knapsack without exceeding the weight limit.

Time Complexity:

- $O(n * W)$, where W = max capacity

Java Code:

```

java
CopyEdit
int knapsack(int W, int[] wt, int[] val, int n) {
    int[][] dp = new int[n + 1][W + 1];
    for (int i = 0; i <= n; i++) {
        for (int w = 0; w <= W; w++) {
            if (i == 0 || w == 0) dp[i][w] = 0;
            else if (wt[i - 1] <= w)
                dp[i][w] = Math.max(val[i - 1] + dp[i - 1][w - wt[i - 1]],
                dp[i - 1][w]);
            else dp[i][w] = dp[i - 1][w];
        }
    }
    return dp[n][W];
}

```

11. Longest Common Subsequence (LCS)

Description:

Finds the longest subsequence common to two sequences (not necessarily contiguous).

Time Complexity:

- $O(m \times n)$

Java Code:

```
java
CopyEdit
int lcs(String X, String Y) {
    int m = X.length(), n = Y.length();
    int[][] dp = new int[m + 1][n + 1];
    for (int i = 1; i <= m; i++) {
        for (int j = 1; j <= n; j++) {
            if (X.charAt(i - 1) == Y.charAt(j - 1))
                dp[i][j] = 1 + dp[i - 1][j - 1];
            else
                dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
        }
    }
    return dp[m][n];
}
```

12. Sieve of Eratosthenes

Description:

Efficient algorithm to generate all prime numbers up to n .

Time Complexity:

- $O(n \log \log n)$

Java Code:

```
java
CopyEdit
void sieveOfEratosthenes(int n) {
    boolean[] prime = new boolean[n + 1];
    Arrays.fill(prime, true);
    for (int p = 2; p * p <= n; p++) {
        if (prime[p]) {
            for (int i = p * p; i <= n; i += p)
                prime[i] = false;
        }
    }
    for (int i = 2; i <= n; i++) {
        if (prime[i]) System.out.print(i + " ");
    }
}
```

13. Depth-First Search (DFS)

Description:

Graph traversal method that explores as far as possible before backtracking.

Time Complexity:

- $O(V + E)$

Java Code:

```
java
CopyEdit
void dfs(int v, boolean[] visited, List<List<Integer>> adj) {
    visited[v] = true;
    System.out.print(v + " ");
    for (int u : adj.get(v)) {
        if (!visited[u]) dfs(u, visited, adj);
    }
}
```

14. Breadth-First Search (BFS)

Description:

Graph traversal technique that explores all neighbors at the current depth before going deeper.

Time Complexity:

- $O(V + E)$

Java Code:

```
java
CopyEdit
void bfs(int start, List<List<Integer>> adj) {
    boolean[] visited = new boolean[adj.size()];
    Queue<Integer> queue = new LinkedList<>();
    queue.add(start);
    visited[start] = true;
    while (!queue.isEmpty()) {
        int v = queue.poll();
        System.out.print(v + " ");
        for (int u : adj.get(v)) {
            if (!visited[u]) {
                visited[u] = true;
                queue.add(u);
            }
        }
    }
}
```

15. Dijkstra's Algorithm

Description:

Finds the shortest path from a source to all vertices using a priority queue.

Time Complexity:

- $O((V + E) \log V)$ with min-heap

Java Code:

```
java
CopyEdit
void dijkstra(int V, List<List<int[]>> adj, int src) {
    int[] dist = new int[V];
    Arrays.fill(dist, Integer.MAX_VALUE);
    dist[src] = 0;
    PriorityQueue<int[]> pq = new PriorityQueue<>(Comparator.comparingInt(a
-> a[1]));
    pq.add(new int[]{src, 0});

    while (!pq.isEmpty()) {
        int[] curr = pq.poll();
        int u = curr[0], d = curr[1];
        if (d > dist[u]) continue;
        for (int[] neighbor : adj.get(u)) {
            int v = neighbor[0], weight = neighbor[1];
            if (dist[u] + weight < dist[v]) {
                dist[v] = dist[u] + weight;
                pq.add(new int[]{v, dist[v]});
            }
        }
    }
}
```

16. Prim's Algorithm

Description:

Constructs a Minimum Spanning Tree using a greedy approach.

Time Complexity:

- $O((V + E) \log V)$

Java Code:

```
java
CopyEdit
void primMST(int V, List<List<int[]>> adj) {
    boolean[] visited = new boolean[V];
    PriorityQueue<int[]> pq = new PriorityQueue<>(Comparator.comparingInt(a
-> a[1]));
    pq.offer(new int[]{0, 0}); // {node, weight}
    int mstWeight = 0;

    while (!pq.isEmpty()) {
        int[] curr = pq.poll();
        int u = curr[0], wt = curr[1];
        if (visited[u]) continue;
```

```

        visited[u] = true;
        mstWeight += wt;

        for (int[] edge : adj.get(u)) {
            int v = edge[0], w = edge[1];
            if (!visited[v]) pq.offer(new int[]{v, w});
        }
    }
    System.out.println("MST weight: " + mstWeight);
}

```

17. Kruskal's Algorithm

Description:

Builds MST using union-find for edge selection based on weight.

Time Complexity:

- $O(E \log E)$

Java Code:

```

java
CopyEdit
int find(int[] parent, int x) {
    if (parent[x] != x) parent[x] = find(parent, parent[x]);
    return parent[x];
}

void union(int[] parent, int[] rank, int x, int y) {
    int px = find(parent, x), py = find(parent, y);
    if (rank[px] < rank[py]) parent[px] = py;
    else if (rank[px] > rank[py]) parent[py] = px;
    else {
        parent[py] = px;
        rank[px]++;
    }
}

void kruskalMST(int[][] edges, int V) {
    Arrays.sort(edges, Comparator.comparingInt(e -> e[2]));
    int[] parent = new int[V], rank = new int[V];
    for (int i = 0; i < V; i++) parent[i] = i;

    int mstWeight = 0;
    for (int[] edge : edges) {
        int u = edge[0], v = edge[1], wt = edge[2];
        if (find(parent, u) != find(parent, v)) {
            union(parent, rank, u, v);
            mstWeight += wt;
        }
    }
    System.out.println("MST weight: " + mstWeight);
}

```



18. Topological Sort (DFS-based)



Description:

Sorts a Directed Acyclic Graph (DAG) in linear order respecting dependencies.



Time Complexity:

- $O(V + E)$



Java Code:

```
java
CopyEdit
void topologicalSort(int V, List<List<Integer>> adj) {
    boolean[] visited = new boolean[V];
    Stack<Integer> stack = new Stack<>();

    for (int i = 0; i < V; i++) {
        if (!visited[i]) dfsTopo(i, visited, stack, adj);
    }

    while (!stack.isEmpty()) System.out.print(stack.pop() + " ");
}

void dfsTopo(int v, boolean[] visited, Stack<Integer> stack,
List<List<Integer>> adj) {
    visited[v] = true;
    for (int u : adj.get(v)) {
        if (!visited[u]) dfsTopo(u, visited, stack, adj);
    }
    stack.push(v);
}
```



19. Cycle Detection in Directed Graph



Description:

Detects cycles using DFS and recursion stack in directed graphs.



Time Complexity:

- $O(V + E)$



Java Code:

```
java
CopyEdit
boolean hasCycle(int V, List<List<Integer>> adj) {
    boolean[] visited = new boolean[V];
    boolean[] recStack = new boolean[V];

    for (int i = 0; i < V; i++) {
        if (!visited[i] && dfsCycle(i, visited, recStack, adj))
```

```

        return true;
    }
    return false;
}

boolean dfsCycle(int v, boolean[] visited, boolean[] recStack,
List<List<Integer>> adj) {
    visited[v] = true;
    recStack[v] = true;

    for (int u : adj.get(v)) {
        if (!visited[u] && dfsCycle(u, visited, recStack, adj)) return
true;
        else if (recStack[u]) return true;
    }

    recStack[v] = false;
    return false;
}

```

20. Union-Find (Disjoint Set)

Description:

Maintains partitioned sets with fast union and find operations.

Time Complexity:

- Nearly $O(1)$ (with path compression & union by rank)

Java Code:

```

java
CopyEdit
int find(int[] parent, int x) {
    if (parent[x] != x) parent[x] = find(parent, parent[x]);
    return parent[x];
}

void union(int[] parent, int[] rank, int x, int y) {
    int px = find(parent, x), py = find(parent, y);
    if (px == py) return;
    if (rank[px] < rank[py]) parent[px] = py;
    else if (rank[px] > rank[py]) parent[py] = px;
    else {
        parent[py] = px;
        rank[px]++;
    }
}

```

21. Floyd-Warshall Algorithm

Description:

Dynamic programming algorithm to find all pairs shortest paths in a weighted graph.

Time Complexity:

- $O(V^3)$

Java Code:

```
java
CopyEdit
void floydWarshall(int[][] dist, int V) {
    for (int k = 0; k < V; k++) {
        for (int i = 0; i < V; i++) {
            for (int j = 0; j < V; j++) {
                if (dist[i][k] + dist[k][j] < dist[i][j])
                    dist[i][j] = dist[i][k] + dist[k][j];
            }
        }
    }
}
```

22. Rabin-Karp Algorithm (Pattern Matching)

Description:

Efficient string matching using hashing to find any one of a set of pattern strings in a text.

Time Complexity:

- Average: $O(n + m)$, Worst: $O(nm)$

Java Code:

```
java
CopyEdit
void rabinKarp(String text, String pattern, int q) {
    int d = 256, m = pattern.length(), n = text.length();
    int p = 0, t = 0, h = 1;

    for (int i = 0; i < m - 1; i++)
        h = (h * d) % q;

    for (int i = 0; i < m; i++) {
        p = (d * p + pattern.charAt(i)) % q;
        t = (d * t + text.charAt(i)) % q;
    }

    for (int i = 0; i <= n - m; i++) {
        if (p == t) {
            boolean match = true;
            for (int j = 0; j < m; j++) {
                if (text.charAt(i + j) != pattern.charAt(j)) {
                    match = false;
                    break;
                }
            }
            if (match) System.out.println("Pattern found at index " + i);
        }
    }
}
```

```

    }
    if (i < n - m) {
        t = (d * (t - text.charAt(i) * h) + text.charAt(i + m)) % q;
        if (t < 0) t = (t + q);
    }
}
}

```



23. KMP Algorithm (Knuth-Morris-Pratt)



Description:

Pattern matching algorithm that avoids redundant comparisons.



Time Complexity:

- $O(n + m)$



Java Code:

```

java
CopyEdit
void KMP(String text, String pattern) {
    int[] lps = computeLPSArray(pattern);
    int i = 0, j = 0;

    while (i < text.length()) {
        if (pattern.charAt(j) == text.charAt(i)) {
            i++; j++;
        }
        if (j == pattern.length()) {
            System.out.println("Pattern found at index " + (i - j));
            j = lps[j - 1];
        } else if (i < text.length() && pattern.charAt(j) !=
text.charAt(i)) {
            if (j != 0) j = lps[j - 1];
            else i++;
        }
    }
}

int[] computeLPSArray(String pat) {
    int[] lps = new int[pat.length()];
    int len = 0, i = 1;
    while (i < pat.length()) {
        if (pat.charAt(i) == pat.charAt(len)) {
            lps[i++] = ++len;
        } else {
            if (len != 0) len = lps[len - 1];
            else lps[i++] = 0;
        }
    }
    return lps;
}

```

24. Bellman-Ford Algorithm

Description:

Computes shortest paths from a single source in a weighted graph, handling negative weights.

Time Complexity:

- $O(V \times E)$

Java Code:

```
java
CopyEdit
void bellmanFord(int V, int[][] edges, int src) {
    int[] dist = new int[V];
    Arrays.fill(dist, Integer.MAX_VALUE);
    dist[src] = 0;

    for (int i = 0; i < V - 1; i++) {
        for (int[] edge : edges) {
            int u = edge[0], v = edge[1], w = edge[2];
            if (dist[u] != Integer.MAX_VALUE && dist[u] + w < dist[v])
                dist[v] = dist[u] + w;
        }
    }

    for (int[] edge : edges) {
        int u = edge[0], v = edge[1], w = edge[2];
        if (dist[u] != Integer.MAX_VALUE && dist[u] + w < dist[v])
            System.out.println("Graph contains negative weight cycle");
    }

    for (int i = 0; i < V; i++)
        System.out.println("Distance from " + src + " to " + i + " is " +
dist[i]);
}
```

25. Boyer-Moore Algorithm

Description:

A highly efficient string searching algorithm using bad character and good suffix heuristics.

Time Complexity:

- Best case: $O(n/m)$, Worst case: $O(n \times m)$

Java Code (Bad Character Heuristic Only):

```
java
CopyEdit
void badCharHeuristic(char[] str, int size, int[] badchar) {
    Arrays.fill(badchar, -1);
}
```



```

        for (int i = 0; i < size; i++)
            badchar[str[i]] = i;
    }

    void boyerMoore(String txt, String pat) {
        int m = pat.length();
        int n = txt.length();

        int[] badchar = new int[256];
        badCharHeuristic(pat.toCharArray(), m, badchar);

        int s = 0;
        while (s <= (n - m)) {
            int j = m - 1;
            while (j >= 0 && pat.charAt(j) == txt.charAt(s + j))
                j--;
            if (j < 0) {
                System.out.println("Pattern occurs at index " + s);
                s += (s + m < n) ? m - badchar[txt.charAt(s + m)] : 1;
            } else {
                s += Math.max(1, j - badchar[txt.charAt(s + j)]);
            }
        }
    }
}

```