Comprehensive Algorithm Reference

◆ 1. Selection Sort

Definition: Selection Sort is a simple comparison-based sorting algorithm. It divides the array into a sorted and unsorted part and repeatedly selects the smallest element from the unsorted part and moves it to the sorted part.

Explanation:

- Start with the first element.
- Find the smallest element in the remaining array.
- Swap it with the first element.
- Repeat this for all positions.

Pseudocode:

```
for i ← 0 to n-2 do
    min ← i
    for j ← i+1 to n-1 do
        if A[j] < A[min] then
            min ← j
    swap A[i] with A[min]
```

Example:
Array: [29, 10, 14, 37, 13]
Step 1: Smallest is 10 → swap with 29 → [10, 29, 14, 37, 13]
Step 2: Smallest in [29,14,37,13] is 13 → [10, 13, 14, 37, 29] ... and so on.

Time Complexity: $O(n^2)$ in all cases (best, average, worst)
Space Complexity: O(1) (in-place sorting)
Stable: No (relative order of equal elements not preserved)

◆ 2. Bubble Sort

***Definition:** Bubble Sort repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order.*

Explanation:

- In each pass, the largest unsorted element “bubbles” to the end.
- Repeats until the list is sorted.

Pseudocode:

```
for i ← 0 to n-1 do
    for j ← 0 to n-i-2 do
        if A[j] > A[j+1] then
            swap A[j], A[j+1]
```

Example:
Array: [5, 3, 8, 4, 2]
Pass 1: [3, 5, 4, 2, 8]
Pass 2: [3, 4, 2, 5, 8]
Pass 3: [3, 2, 4, 5, 8]
Pass 4: [2, 3, 4, 5, 8]

Time Complexity:
Best: O(n) (already sorted)
Average/Worst: O(n^2)
Space Complexity: O(1)
Stable: Yes

◆ 3. Merge Sort

***Definition:** Merge Sort is a divide-and-conquer algorithm. It divides the input array into two halves, recursively sorts them, and then merges the two sorted halves.*

Steps:

1. Divide the unsorted list into sublists until each contains a single element.

2. Merge sublists to produce new sorted sublists.
3. Repeat until one sorted list remains.

Pseudocode:

```
function MergeSort(A, low, high):
    if low < high:
        mid ← (low + high)/2
        MergeSort(A, low, mid)
        MergeSort(A, mid+1, high)
        Merge(A, low, mid, high)
```

> ***Example:***
> *Array: [38, 27, 43, 3, 9]*
> *Divide: [38, 27], [43, 3, 9] → [38], [27], [43], [3], [9]*
> *Merge: [27, 38], [3, 9, 43] → [3, 9, 27, 38, 43]*

Time Complexity: O(n log n) for all cases
Space Complexity: O(n) (temporary arrays)
Stable: Yes

◆ 4. Quick Sort

Definition: Quick Sort is a divide-and-conquer algorithm that picks a pivot element, partitions the array into two parts, and then recursively applies the same logic.

Explanation:

- Choose a pivot (commonly the last element).
- Rearrange the array so that all elements less than pivot are on the left, and greater on the right.
- Recursively apply the same logic to the subarrays.

Pseudocode:

```
function QuickSort(A, low, high):
    if low < high:
        pivot ← Partition(A, low, high)
        QuickSort(A, low, pivot-1)
        QuickSort(A, pivot+1, high)
```

Partition Function:

```
function Partition(A, low, high):
    pivot ← A[high]
    i ← low - 1
    for j ← low to high - 1:
        if A[j] ≤ pivot:
            i ← i + 1
            swap A[i], A[j]
    swap A[i+1], A[high]
    return i + 1
```

> ***Example:***
> *Array: [10, 7, 8, 9, 1, 5], Pivot = 5*
> *After Partition: [1, 5, 8, 9, 10, 7] and recurse on each side*

Time Complexity:
Best/Average: O(n log n)
Worst: O(n^2) (when array is already sorted and pivot is worst)
Space Complexity: O(log n) (recursive stack)
Stable: No

◆ 5. Sequential Search

Definition: Also known as linear search, it checks each element of the list until the desired element is found or the list ends.

Pseudocode:

```
for i ← 0 to n-1 do
    if A[i] == key:
        return i
return -1
```

Explanation:

- Begin from the first element.
- Compare each element to the target value.
- Return index if found, else return -1.

Example:
Array: [4, 2, 7, 1, 3], Key: 7 → Found at index 2

Time Complexity:
Best: O(1) (if first element)
Worst: O(n) (if not found or at last index)

◆ 6. Binary Search

Definition: Binary Search is an efficient algorithm for finding an item in a sorted array by repeatedly dividing the search interval in half.

Pseudocode:

```
low ← 0
high ← n - 1
while low ≤ high:
    mid ← (low + high) / 2
    if A[mid] == key:
        return mid
    else if key < A[mid]:
        high ← mid - 1
    else:
        low ← mid + 1
return -1
```

Explanation:

- Find the middle element.
- If it matches the key, return the index.
- If key is smaller, search the left half.
- If key is larger, search the right half.

Example:
Array: [1, 3, 5, 7, 9], Key: 5 → Mid = 2 → Found

Time Complexity: O(log n)
Space Complexity: O(1) for iterative, O(log n) for recursive
Limitation: Requires sorted input array

◆ 7. Brute Force String Matching

Definition: Brute Force String Matching is the simplest technique to find a pattern in a text. It checks every possible position in the text where the pattern may occur and compares character by character.

Explanation:

- Compare pattern P with text T starting at position 0.
- Move the pattern one position forward each time and repeat the comparison.
- Stop when a match is found or all positions are checked.

Pseudocode:

```
for i ← 0 to n - m do
    j ← 0
    while j < m and T[i + j] == P[j] do
        j ← j + 1
    if j == m:
        return i
return -1
```

> ***Example:***
> *Text: "ABCABCD", Pattern: "ABC"*
> *Match at index 0 → "ABC" == "ABC"*

Time Complexity:
Worst-case: O(n * m)
Best-case: O(n) (if mismatches early)
Space Complexity: O(1)
Use Case: Simple to implement but inefficient for large texts or patterns.

◆ 8. Matrix Multiplication

*Definition: Matrix multiplication involves multiplying two matrices A and B to produce a result matrix C such that each element C[i][j] is the sum of A[i][k] * B[k][j].*

Pseudocode:

```
for i ← 0 to n-1:
    for j ← 0 to n-1:
        C[i][j] ← 0
        for k ← 0 to n-1:
            C[i][j] ← C[i][j] + A[i][k] * B[k][j]
```

Explanation:

- Outer loops iterate over row i and column j.
- Inner loop calculates the dot product of row i of A and column j of B.

> ***Example:***
> *A = [[1, 2], [3, 4]], B = [[2, 0], [1, 2]]*
> *C[0][0] = 1*2 + 2*1 = 4*
> *C[0][1] = 1*0 + 2*2 = 4*

Time Complexity: $O(n^3)$ for naive method
Space Complexity: $O(n^2)$ for result matrix

◆ 9. Decimal to Binary Conversion

Definition: Convert a decimal number to binary by repeatedly dividing the number by 2 and noting the remainder.

Pseudocode:

```
while n > 0:
    print(n % 2)
    n ← n // 2
```

Explanation:

- Divide the number by 2.
- Remainders give binary digits from LSB to MSB.
- Read the remainders in reverse order for final binary number.

> ***Example:***
> *Decimal = 13*
> *Steps: 13%2 = 1, 6%2 = 0, 3%2 = 1, 1%2 = 1 → Binary = 1101*

Time Complexity: O(log n)
Space Complexity: O(log n) (due to bit storage)

◆ 10. Find Maximum Element in Array

Definition: This algorithm scans through the array and returns the largest value.

Pseudocode:

```
max ← A[0]
for i ← 1 to n-1:
    if A[i] > max:
        max ← A[i]
return max
```

> ***Example:***
> *Array: [5, 17, 3, 9] → Max = 17*

Time Complexity: O(n)
Space Complexity: O(1)

◆ 11. Find Unique Elements in Array

Definition: Check whether all elements in the array are unique.

Pseudocode:

```
for i ← 0 to n-1:
    for j ← i+1 to n-1:
        if A[i] == A[j]:
            return False
return True
```

> ***Example:***
> *Array: [3, 1, 4, 2] → All unique → returns True*
> *Array: [3, 1, 4, 1] → Duplicate → returns False*

Time Complexity: $O(n^2)$ (brute-force)
Optimized: O(n log n) with sorting or O(n) with hash set
Space Complexity: O(1) (brute-force)

◆ 12. Horspool's String Matching Algorithm

Definition: Horspool's algorithm improves over brute force by using a shift table to skip unnecessary comparisons.

Steps:

1. Preprocess pattern P to build a shift table.
2. Compare pattern from right to left.
3. On mismatch, use shift table to jump ahead.

Pseudocode Overview:

```
BuildShiftTable(P)
Set i = m-1 (end of pattern)
while i < n:
    compare P from end with T[i]
    if match → report position
    else shift using table
```

> ***Example:***
> *Text: "abcdabcxabcdabcdabcy"*
> *Pattern: "abcdabcy"*
> *Mismatch occurs early, so pattern jumps forward efficiently.*

Time Complexity:
Best/Average: O(n)
Worst-case: O(nm)
Space Complexity: O(k) where k = alphabet size
Advantage: Faster than brute force for large texts

◆ 13. Prim's Algorithm (Minimum Spanning Tree)

Definition: Prim's algorithm finds a Minimum Spanning Tree (MST) for a weighted undirected graph. It starts with a single vertex and grows the MST by adding the minimum weight edge from the tree to a vertex not yet in the tree.

Steps:

1. Initialize all vertices as not in MST.
2. Choose a starting vertex and mark it as part of MST.
3. At each step, pick the smallest edge that connects a vertex in MST to a vertex not in MST.
4. Repeat until all vertices are included.

Pseudocode:

```
Initialize key[] ← ∞, parent[] ← -1, MST[] ← false
key[0] ← 0
for count ← 0 to V-1:
    u ← vertex with minimum key[] not in MST
    MST[u] ← true
    for each neighbor v of u:
        if weight[u][v] < key[v] and v not in MST:
            parent[v] ← u
            key[v] ← weight[u][v]
```

> ***Example:***
> *Graph: A-B (2), A-C (3), B-C (1), B-D (4), C-D (5)*
> *Start at A → MST edges: A-B, B-C, B-D with total weight = 7*

Time Complexity: O(V^2) with adjacency matrix, O(E log V) with priority queue + adjacency list
Space Complexity: O(V)

◆ 14. Kruskal's Algorithm (Minimum Spanning Tree)

Definition: Kruskal's algorithm builds the MST by sorting all the edges by weight and adding them one by one to the MST if they don't form a cycle.

Steps:

1. Sort all edges in non-decreasing order of weights.
2. Initialize MST as empty.

3. For each edge (u, v):
    - If u and v are in different sets, add the edge to MST and union their sets.

Pseudocode:

```
Sort all edges by weight
Initialize parent[] for union-find
for each edge (u, v):
    if find(u) ≠ find(v):
        add (u, v) to MST
        union(u, v)
```

> ***Example:***
> *Same graph as above. Edges in order: B-C (1), A-B (2), A-C (3), B-D (4), C-D (5)*
> *Pick B-C, A-B, B-D → MST weight = 7*

Time Complexity: O(E log E) due to sorting edges
Space Complexity: O(V) with union-find structure

🔹 15. Dijkstra's Algorithm (Single Source Shortest Path)

Definition: Dijkstra's algorithm finds the shortest path from a source node to all other nodes in a weighted graph with non-negative weights.

Steps:

1. Set distance to all vertices as ∞ and source as 0.
2. Use a priority queue to pick the minimum distance vertex not yet processed.
3. Update distances of its adjacent vertices if a shorter path is found.

Pseudocode:

```
Initialize dist[] ← ∞, dist[source] ← 0
Create a min-priority queue of all vertices
while queue is not empty:
    u ← extract-min
    for each neighbor v of u:
        if dist[u] + weight(u, v) < dist[v]:
            dist[v] ← dist[u] + weight(u, v)
```

Example:
Graph with edges: A-B(4), A-C(1), C-B(2), B-D(1), C-D(5)
From A → A-C = 1, C-B = 3, B-D = 4

Time Complexity:
$O(V^2)$ with array
O((V + E) log V) with priority queue
Space Complexity: O(V)

16. Huffman Coding

Definition: Huffman Coding is a greedy algorithm used for lossless data compression. It assigns shorter binary codes to more frequent characters and longer codes to less frequent ones.

Steps:

1. Create a min-heap with all characters and their frequencies.
2. While the heap has more than one node:
    - Extract the two nodes with the smallest frequency.
    - Create a new internal node with their sum as frequency.
    - Insert the new node back into the heap.
3. The remaining node is the root of the Huffman Tree.

Pseudocode:

```
Build a priority queue (min-heap) with all characters
while heap.size > 1:
    left ← extractMin()
    right ← extractMin()
    newNode ← internal node with freq = left.freq + right.freq
    newNode.left = left, newNode.right = right
    insert newNode into heap
Return root of Huffman Tree
```

Example:
Characters: A:5, B:9, C:12, D:13, E:16, F:45
Huffman Tree gives codes like: F=0, C=100, D=101, etc.

Time Complexity: O(n log n) where n = number of characters
Space Complexity: O(n)

◆ 17. Floyd-Warshall Algorithm (All-Pairs Shortest Path)

Definition: The Floyd-Warshall algorithm finds shortest paths between all pairs of vertices in a weighted graph (can handle negative weights, but not negative cycles).

Steps:

1. Initialize distance matrix dist with edge weights.
2. For each vertex k, update dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])

Pseudocode:

```
for k ← 1 to n:
    for i ← 1 to n:
        for j ← 1 to n:
            if dist[i][k] + dist[k][j] < dist[i][j]:
                dist[i][j] ← dist[i][k] + dist[k][j]
```

> ***Example:***
> *Initial matrix:*
> *0 5 ∞ 10*
> *∞ 0 3 ∞*
> *∞ ∞ 0 1*
> *∞ ∞ ∞ 0*
> *After running the algorithm, we get shortest distances between all pairs.*

Time Complexity: $O(n^3)$
Space Complexity: $O(n^2)$

🔹 18. Warshall's Algorithm (Transitive Closure of a Graph)

Definition: Warshall's algorithm computes the transitive closure of a directed graph — determining which vertices are reachable from each vertex.

Steps:

1. Use the adjacency matrix of the graph.
2. For each intermediate vertex k, update path[i][j] = path[i][j] OR (path[i][k] AND path[k][j])

Pseudocode:

```
for k ← 0 to n-1:
    for i ← 0 to n-1:
        for j ← 0 to n-1:
            path[i][j] ← path[i][j] or (path[i][k] and path[k][j])
```

> ***Example:***
> *Adjacency matrix:*
> *0 1 0*
> *0 0 1*
> *1 0 0*
> *After applying Warshall's algorithm, the matrix tells you reachability between all vertex pairs.*

Time Complexity: $O(n^3)$
Space Complexity: $O(n^2)$

🔹 19. 0/1 Knapsack Problem (Dynamic Programming)

Definition: Given n items, each with weight w[i] and value v[i], and a knapsack of capacity W, determine the maximum total value of items that can be put in the knapsack such that each item is either included or excluded (0/1).

Steps:

1. Define dp[i][w] as the max value using first i items with total weight w.
2. For each item i:
    - If w[i] > w: not included → dp[i][w] = dp[i-1][w]
    - Else: included or not → dp[i][w] = max(dp[i-1][w], v[i] + dp[i-1][w - w[i]])

Pseudocode:

```
for i ← 0 to n:
    for w ← 0 to W:
        if i == 0 or w == 0:
            dp[i][w] ← 0
        else if wt[i-1] ≤ w:
            dp[i][w] ← max(val[i-1] + dp[i-1][w-wt[i-1]], dp[i-1][w])
        else:
            dp[i][w] ← dp[i-1][w]
```

> ***Example:***
> *Items: val = [60, 100, 120], wt = [10, 20, 30], Capacity = 50*
> *Max value: 220 (items 2 and 3)*

Time Complexity: O(nW)
Space Complexity: O(nW) (can be reduced to O(W) using 1D array)

◆ 20. N-Queens Problem (Backtracking)

Definition: Place N queens on an N×N chessboard such that no two queens threaten each other (no two queens share the same row, column, or diagonal).

Steps:

1. Place a queen row by row.
2. At each step, check if current position is safe.
3. If safe, place the queen and recurse.
4. If a solution is found, print it.

Pseudocode:

```
function solveNQueens(row):
    if row == N:
        print board
        return
    for col ← 0 to N-1:
```

```
        if isSafe(row, col):
            placeQueen(row, col)
            solveNQueens(row + 1)
            removeQueen(row, col)
```

Time Complexity: O(N!)

Space Complexity: O(N^2) for board or O(N) with arrays

21. Subset Sum Problem (Backtracking)

Definition: Given a set of integers and a value sum, determine if a subset exists whose sum equals the given value.

Steps:

1. At each step, include or exclude the current item.
2. If sum becomes 0 → return true.
3. If end of array is reached → return false.

Pseudocode:

```
function isSubsetSum(i, sum):
    if sum == 0: return true
    if i == n: return false
    if A[i] > sum:
        return isSubsetSum(i+1, sum)
    return isSubsetSum(i+1, sum) or isSubsetSum(i+1, sum - A[i])
```

Time Complexity: O(2^n)

Space Complexity: O(n)