DSA

ALGORITHMS

Sorting Algorithms

1. **Bubble Sort**
2. **Merge Sort**
3. **Selection Sort**
4. **Insertion Sort**
5. **Quick Sort**
6. **Heap Sort**
7. **Counting Sort**

Searching Algorithms

1. **Linear Search**
2. **Binary Search**
3. **Jump Search**
4. **Interpolation Search**
5. **Exponential Search**
6. **Ternary Search**

Dynamic Programming Algorithms

1. **Fibonacci Sequence**
2. **0/1 Knapsack Problem**
3. **Longest Common Subsequence (LCS)**
4. **Longest Increasing Subsequence (LIS)**
5. **Matrix Chain Multiplication**

6. **Edit Distance (Levenshtein Distance)**

Graph Algorithms

1. **Depth First Search (DFS)**
2. **Breadth First Search (BFS)**
3. **Dijkstra's Algorithm**
4. **Bellman-Ford Algorithm**
5. **Floyd-Warshall Algorithm**
6. **A* Search Algorithm**
7. **Kruskal's Algorithm**
8. **Prim's Algorithm**
9. **Tarjan's Algorithm (SCC)**
10. **Kosaraju's Algorithm (SCC)**
11. **Cycle Detection (Directed & Undirected)**

Greedy Algorithms

1. **Activity Selection Problem**
2. **Kruskal's Algorithm**
3. **Prim's Algorithm**

Backtracking Algorithms

1. **N-Queens Problem**
2. **Sudoku Solver**

Mathematical Algorithms

1. **Euclidean Algorithm (GCD)**
2. **Sieve of Eratosthenes**
3. **Modular Exponentiation**

Tree Algorithms

1. **Tree Traversals (In-order, Pre-order, Post-order)**
2. **Binary Search Tree (BST): Insertion, Deletion, Search**
3. **Lowest Common Ancestor (LCA)**
4. **AVL Tree (Self-Balancing BST)**

Sorting Algorithms

Searching Algorithms

Graph Algorithms

Dynamic Programming Algorithms

Greedy Algorithms

Backtracking Algorithms

Mathematical Algorithms

Tree Algorithms

Sorting Algorithms

1. Bubble Sort

Theory:

Bubble Sort is a simple comparison-based sorting algorithm. It repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. This process is repeated until the list is sorted.

It is called **Bubble Sort** because smaller elements "bubble" to the top of the list, and larger elements sink to the bottom.

Algorithm:

1. Start from the first element, compare it with the next element.
2. If the current element is greater than the next element, swap them.
3. Move to the next element and repeat the process for all elements.
4. Continue the process for all elements until no swaps are needed, meaning the list is sorted.

Code (Java):

java

```
public class BubbleSort {
    public static void bubbleSort(int[] arr) {
        int n = arr.length;
        boolean swapped;
        for (int i = 0; i < n - 1; i++) {
            swapped = false;
            for (int j = 0; j < n - i - 1; j++) {
                if (arr[j] > arr[j + 1]) {
                    // Swap arr[j] and arr[j + 1]
                    int temp = arr[j];
                    arr[j] = arr[j + 1];
                    arr[j + 1] = temp;
                    swapped = true;
                }
            }
            // If no two elements were swapped, break
            if (!swapped) break;
        }
    }

    public static void main(String[] args) {
        int[] arr = {5, 1, 4, 2, 8};
        bubbleSort(arr);
        for (int i : arr) {
            System.out.print(i + " ");
        }
```

```
    }
}
```

Time Complexity:

- **Best Case**: O(n) (when the list is already sorted)
- **Average Case**: $O(n^2)$
- **Worst Case**: $O(n^2)$

---

2. Binary Search

Theory:

Binary Search is an efficient search algorithm that works on sorted arrays. It repeatedly divides the search interval in half. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half; otherwise, narrow it to the upper half.

Algorithm:

1. Start with two pointers, low and high, representing the current search range.
2. Find the middle element: mid = (low + high) / 2.
3. If the middle element equals the target, return the index.
4. If the target is less than the middle element, adjust the high pointer to mid - 1.
5. If the target is greater than the middle element, adjust the low pointer to mid + 1.
6. Repeat the process until low > high.

Code (Java):

java

```
public class BinarySearch {
    public static int binarySearch(int[] arr, int target) {
        int low = 0, high = arr.length - 1;

        while (low <= high) {
            int mid = low + (high - low) / 2; // Avoid overflow
```

```
            if (arr[mid] == target) {
                return mid;  // Target found
            } else if (arr[mid] < target) {
                low = mid + 1;  // Search the right half
            } else {
                high = mid - 1;  // Search the left half
            }
        }
        return -1;  // Target not found
    }

    public static void main(String[] args) {
        int[] arr = {1, 3, 5, 7, 9, 11};
        int target = 7;
        int result = binarySearch(arr, target);

        if (result == -1) {
            System.out.println("Element not found.");
        } else {
            System.out.println("Element found at index: " + result);
        }
    }
}
```

Time Complexity:

- **Best Case**: O(1) (if the element is found at the middle)
- **Average/Worst Case**: O(log n)

---

3. Merge Sort

Theory:

Merge Sort is a Divide and Conquer algorithm. It divides the input array into two halves, recursively sorts the two halves, and then merges them to produce the sorted array.

Algorithm:

1. Divide the unsorted array into two halves.
2. Recursively sort both halves.
3. Merge the two halves back together.

Code (Java):

java

```
public class MergeSort {
    public static void mergeSort(int[] arr, int left, int right) {
        if (left < right) {
            int mid = (left + right) / 2;

            // Sort first and second halves
            mergeSort(arr, left, mid);
            mergeSort(arr, mid + 1, right);

            // Merge sorted halves
            merge(arr, left, mid, right);
        }
    }

    public static void merge(int[] arr, int left, int mid, int right) {
        int n1 = mid - left + 1;
        int n2 = right - mid;

        // Create temporary arrays
        int[] L = new int[n1];
        int[] R = new int[n2];
```

```
// Copy data to temp arrays
for (int i = 0; i < n1; i++) L[i] = arr[left + i];
for (int j = 0; j < n2; j++) R[j] = arr[mid + 1 + j];

// Merge the temp arrays
int i = 0, j = 0;
int k = left;
while (i < n1 && j < n2) {
    if (L[i] <= R[j]) {
        arr[k] = L[i];
        i++;
    } else {
        arr[k] = R[j];
        j++;
    }
    k++;
}

// Copy remaining elements of L[] and R[]
while (i < n1) {
    arr[k] = L[i];
    i++;
    k++;
}
while (j < n2) {
    arr[k] = R[j];
    j++;
    k++;
```

```
    }
  }

  public static void main(String[] args) {
    int[] arr = {12, 11, 13, 5, 6, 7};
    mergeSort(arr, 0, arr.length - 1);
    for (int i : arr) {
      System.out.print(i + " ");
    }
  }
}
```

Time Complexity:

- **Best Case**: O(n log n)
- **Average Case**: O(n log n)
- **Worst Case**: O(n log n)

---

4. Fibonacci Sequence (Dynamic Programming)

Theory:

The Fibonacci sequence is a series of numbers where each number is the sum of the two preceding ones. The dynamic programming approach avoids recalculating Fibonacci values by storing the results of subproblems.

Algorithm:

1. Use an array to store the Fibonacci values.
2. Start with base cases: Fib(0) = 0, Fib(1) = 1.
3. For each next value, calculate Fib(n) = Fib(n-1) + Fib(n-2) and store it.

Code (Java):

java

```
public class FibonacciDP {
  public static int fibonacci(int n) {
```

```
        if (n <= 1) {
            return n;
        }

        int[] fib = new int[n + 1];
        fib[0] = 0;
        fib[1] = 1;

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

        return fib[n];
    }

    public static void main(String[] args) {
        int n = 10;
        System.out.println("Fibonacci of " + n + " is: " + fibonacci(n));
    }
}
```

Time Complexity:

- **Time Complexity**: O(n)
- **Space Complexity**: O(n)

---

5. Selection Sort

Theory:

Selection Sort is a simple comparison-based sorting algorithm. It repeatedly selects the smallest element from the unsorted portion of the array and swaps it with the first unsorted element. This process continues until the entire array is sorted.

Algorithm:

1. Start with the first element and search the array to find the minimum element.
2. Swap the minimum element with the first element.
3. Move to the next position and repeat the process until the array is sorted.

Code (Java):

java

```
public class SelectionSort {
    public static void selectionSort(int[] arr) {
        int n = arr.length;

        for (int i = 0; i < n - 1; i++) {
            int minIndex = i;

            // Find the minimum element in the unsorted portion
            for (int j = i + 1; j < n; j++) {
                if (arr[j] < arr[minIndex]) {
                    minIndex = j;
                }
            }

            // Swap the found minimum element with the first element
            int temp = arr[minIndex];
            arr[minIndex] = arr[i];
            arr[i] = temp;
        }
    }

    public static void main(String[] args) {
```

```
        int[] arr = {64, 25, 12, 22, 11};
        selectionSort(arr);
        for (int i : arr) {
            System.out.print(i + " ");
        }
    }
}
```

Time Complexity:

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

---

6. Insertion Sort

Theory:

Insertion Sort is a comparison-based sorting algorithm that builds the final sorted array one element at a time. It is much less efficient on large lists than more advanced algorithms like Quick Sort, Heap Sort, or Merge Sort, but it is efficient for small data sets.

Algorithm:

1. Start with the second element, compare it with the previous elements, and insert it into its correct position.
2. Repeat the process for all elements, inserting each one into its correct position in the sorted portion of the array.

Code (Java):

java

```
public class InsertionSort {
    public static void insertionSort(int[] arr) {
        int n = arr.length;

        for (int i = 1; i < n; i++) {
            int key = arr[i];
```

```
        int j = i - 1;

        // Move elements of arr[0..i-1], that are greater than key, to one position ahead
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
  }

  public static void main(String[] args) {
    int[] arr = {12, 11, 13, 5, 6};
    insertionSort(arr);
    for (int i : arr) {
        System.out.print(i + " ");
    }
  }
}
```

Time Complexity:

- **Best Case**: O(n) (when the list is already sorted)
- **Average Case**: O(n^2)
- **Worst Case**: O(n^2)

---

7. Quick Sort

Theory:

Quick Sort is an efficient, comparison-based, divide-and-conquer sorting algorithm. It works by selecting a pivot element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. The sub-arrays are then sorted recursively.

Algorithm:

1. Choose a pivot element (can be the first, last, or a random element).
2. Partition the array such that elements smaller than the pivot are on the left, and elements greater than the pivot are on the right.
3. Recursively apply the same process to the sub-arrays on the left and right of the pivot.

Code (Java):

java

```
public class QuickSort {
    public static void quickSort(int[] arr, int low, int high) {
        if (low < high) {
            int pi = partition(arr, low, high);  // Partitioning index

            // Recursively sort elements before and after partition
            quickSort(arr, low, pi - 1);
            quickSort(arr, pi + 1, high);
        }
    }

    public static int partition(int[] arr, int low, int high) {
        int pivot = arr[high];  // Pivot is taken as the last element
        int i = (low - 1);  // Index of smaller element

        for (int j = low; j < high; j++) {
            // If the current element is smaller than or equal to the pivot
            if (arr[j] <= pivot) {
                i++;
                // Swap arr[i] and arr[j]
                int temp = arr[i];
```

```
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }

        // Swap arr[i+1] and arr[high] (or pivot)
        int temp = arr[i + 1];
        arr[i + 1] = arr[high];
        arr[high] = temp;

        return i + 1;
    }

    public static void main(String[] args) {
        int[] arr = {10, 7, 8, 9, 1, 5};
        quickSort(arr, 0, arr.length - 1);
        for (int i : arr) {
            System.out.print(i + " ");
        }
    }
}
```

Time Complexity:

- **Best Case**: O(n log n) (when the pivot divides the array into two equal parts)
- **Average Case**: O(n log n)
- **Worst Case**: O(n^2) (when the pivot divides the array into an extremely unbalanced partition)

---

8. Heap Sort

Theory:

Heap Sort is a comparison-based sorting algorithm that uses a binary heap data structure. It is similar to selection sort, where the maximum element is selected and placed at the end of the array. The key idea is to use a heap to find the maximum efficiently.

Algorithm:

1. Build a max-heap from the input data.
2. At each step, remove the largest element from the heap (which is the root) and rebuild the heap.
3. Repeat the process until the heap is empty.

Code (Java):

java

```
public class HeapSort {
    public static void heapSort(int[] arr) {
        int n = arr.length;

        // Build max heap
        for (int i = n / 2 - 1; i >= 0; i--) {
            heapify(arr, n, i);
        }

        // One by one extract elements from heap
        for (int i = n - 1; i >= 0; i--) {
            // Move current root to end
            int temp = arr[0];
            arr[0] = arr[i];
            arr[i] = temp;

            // Call max heapify on the reduced heap
            heapify(arr, i, 0);
        }
```

```
}

public static void heapify(int[] arr, int n, int i) {
    int largest = i;  // Initialize largest as root
    int left = 2 * i + 1;  // Left child
    int right = 2 * i + 2;  // Right child

    // If left child is larger than root
    if (left < n && arr[left] > arr[largest]) {
        largest = left;
    }

    // If right child is larger than largest so far
    if (right < n && arr[right] > arr[largest]) {
        largest = right;
    }

    // If largest is not root
    if (largest != i) {
        int swap = arr[i];
        arr[i] = arr[largest];
        arr[largest] = swap;

        // Recursively heapify the affected subtree
        heapify(arr, n, largest);
    }
}

public static void main(String[] args) {
```

```
        int[] arr = {12, 11, 13, 5, 6, 7};
        heapSort(arr);
        for (int i : arr) {
            System.out.print(i + " ");
        }
    }
}
```

Time Complexity:

- **Best Case**: O(n log n)
- **Average Case**: O(n log n)
- **Worst Case**: O(n log n)

---

9. Counting Sort

Theory:

Counting Sort is a non-comparison-based sorting algorithm. It counts the occurrences of each unique element in the input array. The counts are then used to place the elements in the correct position in the sorted array.

Algorithm:

1. Find the maximum and minimum values in the array.
2. Create a count array to store the count of each unique value.
3. Calculate the cumulative count to determine the position of each element.
4. Build the sorted array using the count array.

Code (Java):

java

```
public class CountingSort {
    public static void countingSort(int[] arr) {
        int n = arr.length;
        int max = getMax(arr);  // Get maximum value in the array
```

```
    // Create count array
    int[] count = new int[max + 1];

    // Store count of each element
    for (int i = 0; i < n; i++) {
        count[arr[i]]++;
    }

    // Build the sorted array
    int index = 0;
    for (int i = 0; i <= max; i++) {
        while (count[i] > 0) {
            arr[index++] = i;
            count[i]--;
        }
    }
}

public static int getMax(int[] arr) {
    int max = arr[0];
    for (int i = 1; i < arr.length; i++) {
        if (arr[i] > max) {
            max = arr[i];
        }
    }
    return max;
}

public static void main(String[] args) {
```

```
        int[] arr = {4, 2, 2, 8, 3, 3, 1};
        countingSort(arr);
        for (int i : arr) {
            System.out.print(i + " ");
        }
    }
}
```

Time Complexity:

- **Best Case**: O(n + k) (where k is the range of the input)
- **Average Case**: O(n + k)
- **Worst Case**: O(n + k)

Searching Algorithms

---

1. Linear Search

Theory:

Linear search is a simple search algorithm that checks each element in the array one by one until the target element is found or the list is exhausted.

Algorithm:

1. Start from the first element and compare it with the target.
2. If the element matches the target, return the index.
3. If the end of the array is reached without finding the target, return -1 (not found).

Code (Java):

java

```
public class LinearSearch {
    public static int linearSearch(int[] arr, int target) {
        for (int i = 0; i < arr.length; i++) {
            if (arr[i] == target) {
```

```
            return i;  // Return the index of the target
        }
    }
    return -1;  // Return -1 if target is not found
}

public static void main(String[] args) {
    int[] arr = {10, 23, 45, 70, 11, 15};
    int target = 70;
    int result = linearSearch(arr, target);
    if (result == -1) {
        System.out.println("Element not found");
    } else {
        System.out.println("Element found at index: " + result);
    }
}
}
```

Time Complexity:

- **Best Case**: O(1) (when the target is the first element)
- **Average Case**: O(n)
- **Worst Case**: O(n)

---

2. Binary Search

Theory:

Binary search is an efficient search algorithm that works on sorted arrays. It repeatedly divides the array in half, checking whether the middle element is the target or whether the target is in the left or right half.

Algorithm:

1. Compare the target with the middle element of the array.
2. If the target is equal to the middle element, return the index.

3. If the target is smaller than the middle element, repeat the search in the left half.
4. If the target is larger than the middle element, repeat the search in the right half.

Code (Java):

java

```
public class BinarySearch {
    public static int binarySearch(int[] arr, int target) {
        int left = 0;
        int right = arr.length - 1;

        while (left <= right) {
            int mid = left + (right - left) / 2;  // Avoids overflow

            // Check if target is at the mid
            if (arr[mid] == target) {
                return mid;
            }

            // If target is smaller than mid, ignore the right half
            if (arr[mid] > target) {
                right = mid - 1;
            }
            // If target is larger than mid, ignore the left half
            else {
                left = mid + 1;
            }
        }

        return -1;  // Return -1 if the target is not found
```

```
    }

    public static void main(String[] args) {
        int[] arr = {2, 3, 4, 10, 40};
        int target = 10;
        int result = binarySearch(arr, target);
        if (result == -1) {
            System.out.println("Element not found");
        } else {
            System.out.println("Element found at index: " + result);
        }
    }
}
```

Time Complexity:

- **Best Case**: O(1) (when the middle element is the target)
- **Average Case**: O(log n)
- **Worst Case**: O(log n)

---

3. Jump Search

Theory:

Jump Search is a search algorithm for sorted arrays. It works by jumping ahead by a fixed number of steps (block size), and once it finds a block where the target might be, it performs a linear search within that block.

Algorithm:

1. Jump ahead by a fixed block size (sqrt(n)).
2. When the block where the target might be is found, perform a linear search within that block.
3. If the target is found, return the index; otherwise, return -1.

Code (Java):

java

```
public class JumpSearch {
    public static int jumpSearch(int[] arr, int target) {
        int n = arr.length;
        int step = (int) Math.sqrt(n);  // Block size
        int prev = 0;

        // Jump ahead by block size
        while (arr[Math.min(step, n) - 1] < target) {
            prev = step;
            step += (int) Math.sqrt(n);
            if (prev >= n) {
                return -1;
            }
        }

        // Linear search within the block
        for (int i = prev; i < Math.min(step, n); i++) {
            if (arr[i] == target) {
                return i;
            }
        }

        return -1;  // Target not found
    }

    public static void main(String[] args) {
        int[] arr = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
        int target = 8;
```

```
        int result = jumpSearch(arr, target);
        if (result == -1) {
            System.out.println("Element not found");
        } else {
            System.out.println("Element found at index: " + result);
        }
    }
}
```

Time Complexity:

- **Best Case**: O(1)
- **Average Case**: O(√n)
- **Worst Case**: O(√n)

---

4. Interpolation Search

Theory:

Interpolation Search is an improved variant of binary search for uniformly distributed data. It estimates the position of the target value based on the value's relation to the endpoints. The idea is to use interpolation rather than dividing the array in half as in binary search.

Algorithm:

1. Calculate the position using the formula: pos=low+(target−arr[low])×(high−low)arr[high]−arr[low]pos = low + \frac{(target - arr[low]) \times (high - low)}{arr[high] - arr[low]}pos=low+arr[high]−arr[low](target−arr[low])×(high−low)
2. If the target matches the element at the calculated position, return the index.
3. If the target is smaller, search the left half; otherwise, search the right half.
4. Repeat the process until the target is found or the range becomes invalid.

Code (Java):

java

```
public class InterpolationSearch {
    public static int interpolationSearch(int[] arr, int target) {
```

```
int low = 0, high = arr.length - 1;

while (low <= high && target >= arr[low] && target <= arr[high]) {
    if (low == high) {
        if (arr[low] == target) {
            return low;
        }
        return -1;
    }

    // Estimate the position of the target
    int pos = low + ((target - arr[low]) * (high - low) / (arr[high] - arr[low]));

    // Check if the target is at the estimated position
    if (arr[pos] == target) {
        return pos;
    }

    // If target is larger, move to the right sub-array
    if (arr[pos] < target) {
        low = pos + 1;
    }
    // If target is smaller, move to the left sub-array
    else {
        high = pos - 1;
    }
}

return -1;  // Target not found
```

```
    }

    public static void main(String[] args) {
        int[] arr = {10, 12, 13, 16, 18, 19, 21, 23, 24, 33, 35};
        int target = 18;
        int result = interpolationSearch(arr, target);
        if (result == -1) {
            System.out.println("Element not found");
        } else {
            System.out.println("Element found at index: " + result);
        }
    }
}
```

Time Complexity:

- **Best Case**: O(1)
- **Average Case**: O(log log n)
- **Worst Case**: O(n)

---

5. Exponential Search

Theory:

Exponential Search works on sorted arrays and is useful when the size of the array is not known in advance. It first finds a range where the target might be and then uses binary search within that range.

Algorithm:

1. Start with an index of 1 and double the index in each step until you find an index such that the value at that index is greater than the target.
2. Perform a binary search in the range [previous_index, current_index].

Code (Java):

java

```
public class ExponentialSearch {
    public static int exponentialSearch(int[] arr, int target) {
        if (arr[0] == target) {
            return 0;  // If target is at the first position
        }

        // Find range for binary search by repeated doubling
        int i = 1;
        while (i < arr.length && arr[i] <= target) {
            i = i * 2;
        }

        // Perform binary search in the found range
        return binarySearch(arr, i / 2, Math.min(i, arr.length - 1), target);
    }

    public static int binarySearch(int[] arr, int low, int high, int target) {
        while (low <= high) {
            int mid = low + (high - low) / 2;

            if (arr[mid] == target) {
                return mid;
            }

            if (arr[mid] < target) {
                low = mid + 1;
            } else {
                high = mid - 1;
            }
```

```
        }
        return -1;  // Target not found
    }

    public static void main(String[] args) {
        int[] arr = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
        int target = 70;
        int result = exponentialSearch(arr, target);
        if (result == -1) {
            System.out.println("Element not found");
        } else {
            System.out.println("Element found at index: " + result);
        }
    }
}
```

Time Complexity:

- **Best Case**: O(1)
- **Average Case**: O(log n)
- **Worst Case**: O(log n)

---

6. Ternary Search

Theory:

Ternary Search is similar to binary search, but instead of dividing the array into two halves, it divides the array into three parts. It is performed on sorted arrays.

Algorithm:

1. Divide the array into three parts.
2. Compare the target with the first and second midpoints.
3. If the target is equal to any of the midpoints, return the index.
4. Depending on the comparison, continue searching in one of the three parts.
5. Repeat the process until the target is found or the range becomes invalid.

Code (Java):

java

```
public class TernarySearch {
    public static int ternarySearch(int left, int right, int target, int[] arr) {
        if (right >= left) {
            int mid1 = left + (right - left) / 3;
            int mid2 = right - (right - left) / 3;

            // Check if the target is at mid1 or mid2
            if (arr[mid1] == target) {
                return mid1;
            }
            if (arr[mid2] == target) {
                return mid2;
            }

            // If target is smaller than mid1, search in the left third
            if (target < arr[mid1]) {
                return ternarySearch(left, mid1 - 1, target, arr);
            }
            // If target is greater than mid2, search in the right third
            else if (target > arr[mid2]) {
                return ternarySearch(mid2 + 1, right, target, arr);
            }
            // Otherwise, search in the middle third
            else {
                return ternarySearch(mid1 + 1, mid2 - 1, target, arr);
            }
```

```
        }

        return -1;  // Target not found
    }

    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
        int target = 5;
        int result = ternarySearch(0, arr.length - 1, target, arr);
        if (result == -1) {
            System.out.println("Element not found");
        } else {
            System.out.println("Element found at index: " + result);
        }
    }
}
```

Time Complexity:

- **Best Case**: O(1)
- **Average Case**: O(log3 n)
- **Worst Case**: O(log3 n)

Dynamic Programming Algorithms

---

1. Fibonacci Sequence

Theory:

The Fibonacci sequence is a series where each number is the sum of the two preceding ones, starting from 0 and 1. Using dynamic programming, we store intermediate results to avoid redundant calculations.

Algorithm:

1. Create an array to store Fibonacci numbers.

2. Initialize the first two Fibonacci numbers.
3. Iterate through the remaining numbers and fill in the array using the relation: F(n)=F(n−1)+F(n−2)F(n) = F(n-1) + F(n-2)F(n)=F(n−1)+F(n−2)

Code (Java):

java

```
public class FibonacciDP {
    public static int fibonacci(int n) {
        if (n <= 1) {
            return n;
        }

        int[] fib = new int[n + 1];
        fib[0] = 0;
        fib[1] = 1;

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

        return fib[n];
    }

    public static void main(String[] args) {
        int n = 10;
        System.out.println("Fibonacci number at index " + n + " is: " + fibonacci(n));
    }
}
```

Time Complexity:

- **Time Complexity**: O(n)

- **Space Complexity**: O(n)

---

2. 0/1 Knapsack Problem

Theory:

In the 0/1 Knapsack problem, given a set of items with specific weights and values, the goal is to maximize the total value without exceeding a given weight limit. Dynamic programming helps to solve this by storing solutions of subproblems.

Algorithm:

1. Create a 2D array dp[][], where dp[i][j] stores the maximum value for i items and weight limit j.
2. For each item, decide whether to include it or not based on the current capacity.
3. Return the value from the table for all items and the full capacity.

Code (Java):

java

```
public class KnapsackDP {
    public static int knapsack(int[] weights, int[] values, int capacity) {
        int n = values.length;
        int[][] dp = new int[n + 1][capacity + 1];

        for (int i = 1; i <= n; i++) {
            for (int w = 1; w <= capacity; w++) {
                if (weights[i - 1] <= w) {
                    dp[i][w] = Math.max(values[i - 1] + dp[i - 1][w - weights[i - 1]], dp[i - 1][w]);
                } else {
                    dp[i][w] = dp[i - 1][w];
                }
            }
        }
```

```
        return dp[n][capacity];
    }

    public static void main(String[] args) {
        int[] weights = {1, 3, 4, 5};
        int[] values = {1, 4, 5, 7};
        int capacity = 7;
        System.out.println("Maximum value: " + knapsack(weights, values, capacity));
    }
}
```

Time Complexity:

- **Time Complexity**: O(n * capacity)
- **Space Complexity**: O(n * capacity)

---

3. Longest Common Subsequence (LCS)

Theory:

The Longest Common Subsequence (LCS) problem finds the longest subsequence common to two strings. A dynamic programming table is used to store solutions to subproblems, helping to avoid recomputation.

Algorithm:

1. Create a 2D table dp[][] where dp[i][j] holds the length of LCS of strings X[0..i-1] and Y[0..j-1].
2. If characters match, add 1 to the LCS length of the previous characters.
3. Otherwise, take the maximum of the LCS by excluding one character from either string.

Code (Java):

java

```
public class LCS {
    public static int lcs(String X, String Y) {
        int m = X.length();
```

```
        int 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] = dp[i - 1][j - 1] + 1;
                } else {
                    dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
                }
            }
        }

        return dp[m][n];
    }

    public static void main(String[] args) {
        String X = "AGGTAB";
        String Y = "GXTXAYB";
        System.out.println("Length of LCS: " + lcs(X, Y));
    }
}
```

Time Complexity:

- **Time Complexity**: O(m * n)
- **Space Complexity**: O(m * n)

---

4. Longest Increasing Subsequence (LIS)

Theory:

The Longest Increasing Subsequence (LIS) problem involves finding the longest subsequence of a given sequence where the elements are in sorted order, strictly increasing.

Algorithm:

1. Create an array dp[] to store the longest increasing subsequence up to each index.
2. For each element, compare it with the previous elements to check if it can extend the subsequence.
3. Update the dp[] array accordingly and find the maximum.

Code (Java):

java

```
public class LIS {
    public static int lis(int[] arr) {
        int n = arr.length;
        int[] dp = new int[n];
        int maxLength = 1;

        // Initialize dp array with 1
        for (int i = 0; i < n; i++) {
            dp[i] = 1;
        }

        // Build the LIS for each element
        for (int i = 1; i < n; i++) {
            for (int j = 0; j < i; j++) {
                if (arr[i] > arr[j] && dp[i] < dp[j] + 1) {
                    dp[i] = dp[j] + 1;
                }
            }
            maxLength = Math.max(maxLength, dp[i]);
        }

        return maxLength;
```

```
    }

    public static void main(String[] args) {
        int[] arr = {10, 22, 9, 33, 21, 50, 41, 60};
        System.out.println("Length of LIS: " + lis(arr));
    }
}
```

Time Complexity:

- **Time Complexity**: O(n^2)
- **Space Complexity**: O(n)

---

5. Matrix Chain Multiplication

Theory:

The Matrix Chain Multiplication problem is about finding the most efficient way to multiply a given sequence of matrices. Dynamic programming helps in finding the optimal parenthesization of the matrices to minimize the number of scalar multiplications.

Algorithm:

1. Create a 2D array dp[][] where dp[i][j] represents the minimum cost of multiplying matrices from i to j.
2. Use a bottom-up approach to fill in the table by considering different places to split the sequence of matrices.
3. Return the value from the table for the full sequence of matrices.

Code (Java):

java

```
public class MatrixChainMultiplication {
    public static int matrixChainOrder(int[] p) {
        int n = p.length;
        int[][] dp = new int[n][n];

        // dp[i][i] is 0 because we need 0 multiplications for a single matrix
```

```
        for (int i = 1; i < n; i++) {
            dp[i][i] = 0;
        }

        for (int len = 2; len < n; len++) { // len is the chain length
            for (int i = 1; i < n - len + 1; i++) {
                int j = i + len - 1;
                dp[i][j] = Integer.MAX_VALUE;
                for (int k = i; k < j; k++) {
                    int cost = dp[i][k] + dp[k + 1][j] + p[i - 1] * p[k] * p[j];
                    if (cost < dp[i][j]) {
                        dp[i][j] = cost;
                    }
                }
            }
        }

        return dp[1][n - 1];
    }

    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4};
        System.out.println("Minimum number of multiplications: " + matrixChainOrder(arr));
    }
}
```

Time Complexity:

- **Time Complexity**: O(n^3)
- **Space Complexity**: O(n^2)

---

6. Edit Distance (Levenshtein Distance)

Theory:

The Edit Distance problem finds the minimum number of operations (insertions, deletions, and substitutions) required to convert one string into another. Dynamic programming stores solutions to subproblems and helps to minimize operations.

Algorithm:

1. Create a 2D table dp[][] where dp[i][j] represents the edit distance between substrings of lengths i and j.
2. If characters are the same, no operation is needed. Otherwise, perform insertion, deletion, or substitution.
3. Fill in the table and return the final value.

Code (Java):

java

```
public class EditDistance {
    public static int editDistance(String s1, String s2) {
        int m = s1.length();
        int n = s2.length();
        int[][] dp = new int[m + 1][n + 1];

        for (int i = 0; i <= m; i++) {
            for (int j = 0; j <= n; j++) {
                if (i == 0) {
                    dp[i][j] = j;  // Insert all characters
                } else if (j == 0) {
                    dp[i][j] = i;  // Remove all characters
                } else if (s1.charAt(i - 1) == s2.charAt(j - 1)) {
                    dp[i][j] = dp[i - 1][j - 1];  // No operation
                } else {
                    dp[i][j] = 1 + Math.min(dp[i - 1][j], // Remove
                                    Math.min(dp[i][j - 1], // Insert
```

```
                                dp[i - 1][j - 1])); // Replace
                    }
                }
            }

            return dp[m][n];
        }

        public static void main(String[] args) {
            String s1 = "sunday";
            String s2 = "saturday";
            System.out.println("Edit Distance: " + editDistance(s1, s2));
        }
}
```

Time Complexity:

- **Time Complexity**: O(m * n)
- **Space Complexity**: O(m * n)

Graph Algorithms

---

1. Depth First Search (DFS)

Theory:

DFS is a graph traversal algorithm that starts at a selected node and explores as far as possible along each branch before backtracking. It uses a stack (either explicitly or via recursion) to keep track of the visited nodes.

Algorithm:

1. Start at a selected node, mark it as visited.
2. Recursively explore adjacent unvisited nodes.
3. Once all adjacent nodes are visited, backtrack.

Code (Java):

java

```
import java.util.*;

public class DFS {
    public static void dfs(int node, boolean[] visited, List<List<Integer>> adj) {
        visited[node] = true;
        System.out.print(node + " ");

        for (int neighbor : adj.get(node)) {
            if (!visited[neighbor]) {
                dfs(neighbor, visited, adj);
            }
        }
    }

    public static void main(String[] args) {
        int V = 6;
        List<List<Integer>> adj = new ArrayList<>();
        for (int i = 0; i < V; i++) {
            adj.add(new ArrayList<>());
        }

        adj.get(0).add(1);
        adj.get(0).add(2);
        adj.get(1).add(0);
        adj.get(1).add(3);
        adj.get(2).add(0);
        adj.get(2).add(3);
```

```
        adj.get(3).add(1);
        adj.get(3).add(2);
        adj.get(3).add(4);
        adj.get(4).add(3);
        adj.get(4).add(5);
        adj.get(5).add(4);

        boolean[] visited = new boolean[V];
        System.out.println("DFS traversal:");
        dfs(0, visited, adj);
    }
}
```

Time Complexity:

- **Time Complexity**: O(V + E), where V is the number of vertices and E is the number of edges.
- **Space Complexity**: O(V)

---

2. Breadth First Search (BFS)

Theory:

BFS is a graph traversal algorithm that explores all neighbors at the present depth level before moving on to nodes at the next depth level. It uses a queue to explore nodes in a level-wise manner.

Algorithm:

1. Start at the selected node, enqueue it, and mark it as visited.
2. Dequeue a node and explore all its unvisited neighbors, enqueue them.
3. Repeat the process until the queue is empty.

Code (Java):

java

```
import java.util.*;
```

```
public class BFS {
    public static void bfs(int start, List<List<Integer>> adj) {
        boolean[] visited = new boolean[adj.size()];
        Queue<Integer> queue = new LinkedList<>();

        visited[start] = true;
        queue.add(start);

        while (!queue.isEmpty()) {
            int node = queue.poll();
            System.out.print(node + " ");

            for (int neighbor : adj.get(node)) {
                if (!visited[neighbor]) {
                    visited[neighbor] = true;
                    queue.add(neighbor);
                }
            }
        }
    }

    public static void main(String[] args) {
        int V = 6;
        List<List<Integer>> adj = new ArrayList<>();
        for (int i = 0; i < V; i++) {
            adj.add(new ArrayList<>());
        }
```

```
        adj.get(0).add(1);
        adj.get(0).add(2);
        adj.get(1).add(0);
        adj.get(1).add(3);
        adj.get(2).add(0);
        adj.get(2).add(3);
        adj.get(3).add(1);
        adj.get(3).add(2);
        adj.get(3).add(4);
        adj.get(4).add(3);
        adj.get(4).add(5);
        adj.get(5).add(4);

        System.out.println("BFS traversal:");
        bfs(0, adj);
    }
}
```

Time Complexity:

- **Time Complexity**: O(V + E), where V is the number of vertices and E is the number of edges.
- **Space Complexity**: O(V)

---

3. Dijkstra's Algorithm

Theory:

Dijkstra's algorithm is used to find the shortest path between nodes in a graph with non-negative edge weights. It uses a priority queue to explore the closest unvisited node.

Algorithm:

1. Initialize the distance to the source node as 0 and all other nodes as infinity.
2. Use a priority queue to select the unvisited node with the smallest known distance.
3. For each unvisited neighbor, update the shortest known distance.

4. Repeat until all nodes have been visited.

Code (Java):

java

```
import java.util.*;

public class Dijkstra {
    public static void dijkstra(int src, int V, List<List<Node>> adj) {
        int[] dist = new int[V];
        Arrays.fill(dist, Integer.MAX_VALUE);
        dist[src] = 0;

        PriorityQueue<Node> pq = new PriorityQueue<>(Comparator.comparingInt(node -> node.cost));
        pq.add(new Node(src, 0));

        while (!pq.isEmpty()) {
            Node node = pq.poll();
            int u = node.vertex;

            for (Node neighbor : adj.get(u)) {
                int v = neighbor.vertex;
                int weight = neighbor.cost;

                if (dist[u] + weight < dist[v]) {
                    dist[v] = dist[u] + weight;
                    pq.add(new Node(v, dist[v]));
                }
            }
        }
```

```
    System.out.println("Shortest distances from source:");
    for (int i = 0; i < V; i++) {
        System.out.println("Node " + i + " distance: " + dist[i]);
    }
}

static class Node {
    int vertex;
    int cost;

    public Node(int vertex, int cost) {
        this.vertex = vertex;
        this.cost = cost;
    }
}

public static void main(String[] args) {
    int V = 5;
    List<List<Node>> adj = new ArrayList<>();
    for (int i = 0; i < V; i++) {
        adj.add(new ArrayList<>());
    }

    adj.get(0).add(new Node(1, 9));
    adj.get(0).add(new Node(2, 6));
    adj.get(0).add(new Node(3, 5));
    adj.get(0).add(new Node(4, 3));
    adj.get(2).add(new Node(1, 2));
```

```
        adj.get(2).add(new Node(3, 4));


        dijkstra(0, V, adj);
    }
}
```

Time Complexity:

- **Time Complexity**: O((V + E) log V), where V is the number of vertices and E is the number of edges.
- **Space Complexity**: O(V)

---

4. Bellman-Ford Algorithm

Theory:

The Bellman-Ford algorithm is used to find the shortest path from a single source to all other vertices in a graph, even when edge weights are negative. Unlike Dijkstra's algorithm, it works on graphs with negative weights but is slower.

Algorithm:

1. Initialize the distance to the source node as 0 and all other nodes as infinity.
2. For each edge, attempt to relax it by updating the distance to its destination.
3. Repeat the process V-1 times (where V is the number of vertices).
4. Check for negative-weight cycles by running the relaxation process once more.

Code (Java):

java

```
import java.util.Arrays;


public class BellmanFord {
    public static 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 = 0; i < V - 1; i++) {
        for (int[] edge : edges) {
            int u = edge[0];
            int v = edge[1];
            int weight = edge[2];

            if (dist[u] != Integer.MAX_VALUE && dist[u] + weight < dist[v]) {
                dist[v] = dist[u] + weight;
            }
        }
    }

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

    System.out.println("Shortest distances from source:");
    for (int i = 0; i < V; i++) {
        System.out.println("Node " + i + " distance: " + dist[i]);
    }
}
```

```
    public static void main(String[] args) {
        int V = 5;
        int[][] edges = {
            {0, 1, -1},
            {0, 2, 4},
            {1, 2, 3},
            {1, 3, 2},
            {1, 4, 2},
            {3, 2, 5},
            {3, 1, 1},
            {4, 3, -3}
        };

        bellmanFord(edges, V, 0);
    }
}
```

Time Complexity:

- **Time Complexity**: O(V * E), where V is the number of vertices and E is the number of edges.
- **Space Complexity**: O(V)

Floyd-Warshall Algorithm (All-Pairs Shortest Paths)

Theory:

The **Floyd-Warshall Algorithm** is used to find the shortest paths between all pairs of vertices in a graph, which can have positive or negative edge weights (but no negative weight cycles). It works by comparing all possible paths through the vertices to find the shortest path between each pair.

Algorithm:

1. Initialize a matrix dist[][], where dist[i][j] is the direct distance between node i and node j.

2. For each intermediate vertex k, check if the path from i to j through k is shorter than the current known path dist[i][j]. If so, update dist[i][j].
3. After processing all vertices as intermediates, the matrix dist[][] will hold the shortest paths between all pairs of vertices.

Code (Java):

java

```
public class FloydWarshall {
    static final int INF = 99999; // Use a large number to represent infinity

    public static void floydWarshall(int[][] graph) {
        int V = graph.length;
        int[][] dist = new int[V][V];

        // Initialize the solution matrix as input graph matrix
        for (int i = 0; i < V; i++) {
            for (int j = 0; j < V; j++) {
                dist[i][j] = graph[i][j];
            }
        }

        // Update the distance matrix using Floyd-Warshall Algorithm
        for (int k = 0; k < V; k++) {
            for (int i = 0; i < V; i++) {
                for (int j = 0; j < V; j++) {
                    if (dist[i][j] > dist[i][k] + dist[k][j]) {
                        dist[i][j] = dist[i][k] + dist[k][j];
                    }
                }
            }
```

```
    }

    printSolution(dist);
}

private static void printSolution(int[][] dist) {
    int V = dist.length;
    System.out.println("Shortest distances between every pair of vertices:");
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            if (dist[i][j] == INF) {
                System.out.print("INF ");
            } else {
                System.out.print(dist[i][j] + " ");
            }
        }
        System.out.println();
    }
}

public static void main(String[] args) {
    int V = 4;
    int[][] graph = {
        {0, 3, INF, 7},
        {8, 0, 2, INF},
        {5, INF, 0, 1},
        {2, INF, INF, 0}
    };
```

```
        floydWarshall(graph);
    }
}
```

Time Complexity:

- **Time Complexity**: O(V^3), where V is the number of vertices.
- **Space Complexity**: O(V^2) for the distance matrix.

***A Search Algorithm**

Theory:

The *A Search Algorithm** is a popular pathfinding and graph traversal algorithm, widely used in AI, especially in games. It combines the advantages of Dijkstra's Algorithm and Greedy Best-First Search by using a heuristic to guide its search.

Algorithm:

1. **Open list**: Nodes to be evaluated.
2. **Closed list**: Nodes already evaluated.
3. For each node, calculate f(n) = g(n) + h(n):
    - g(n): The cost to reach the node.
    - h(n): Heuristic estimate of the cost to reach the goal (e.g., Euclidean distance).
4. At each step, choose the node from the open list with the lowest f(n) value and evaluate it.
5. Repeat until the goal is reached.

Code (Java):

java

```
import java.util.*;

class AStar {
    static class Node implements Comparable<Node> {
        int x, y;
        int gCost, hCost, fCost;
```

```
    Node parent;

    Node(int x, int y) {
        this.x = x;
        this.y = y;
    }

    @Override
    public int compareTo(Node o) {
        return Integer.compare(this.fCost, o.fCost);
    }
}

static int[][] directions = { {0, 1}, {1, 0}, {0, -1}, {-1, 0} };

public static List<Node> aStarSearch(Node[][] grid, Node start, Node goal) {
    PriorityQueue<Node> openList = new PriorityQueue<>();
    Set<Node> closedList = new HashSet<>();

    start.gCost = 0;
    start.hCost = heuristic(start, goal);
    start.fCost = start.gCost + start.hCost;
    openList.add(start);

    while (!openList.isEmpty()) {
        Node current = openList.poll();

        if (current.equals(goal)) {
            return reconstructPath(current);
```

```
    }

    closedList.add(current);

    for (int[] direction : directions) {
        int newX = current.x + direction[0];
        int newY = current.y + direction[1];

        if (isValid(grid, newX, newY)) {
            Node neighbor = grid[newX][newY];

            if (closedList.contains(neighbor)) {
                continue;
            }

            int tentativeGCost = current.gCost + 1;  // Assuming uniform cost for moving
            if (tentativeGCost < neighbor.gCost || !openList.contains(neighbor)) {
                neighbor.gCost = tentativeGCost;
                neighbor.hCost = heuristic(neighbor, goal);
                neighbor.fCost = neighbor.gCost + neighbor.hCost;
                neighbor.parent = current;

                if (!openList.contains(neighbor)) {
                    openList.add(neighbor);
                }
            }
        }
    }
}
```

```
    return null;  // No path found
}

private static List<Node> reconstructPath(Node current) {
    List<Node> path = new ArrayList<>();
    while (current != null) {
        path.add(current);
        current = current.parent;
    }
    Collections.reverse(path);
    return path;
}

private static boolean isValid(Node[][] grid, int x, int y) {
    return x >= 0 && x < grid.length && y >= 0 && y < grid[0].length;
}

private static int heuristic(Node a, Node b) {
    return Math.abs(a.x - b.x) + Math.abs(a.y - b.y);  // Manhattan Distance
}

public static void main(String[] args) {
    int size = 5;
    Node[][] grid = new Node[size][size];
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            grid[i][j] = new Node(i, j);
        }
    }
```

```
        Node start = grid[0][0];
        Node goal = grid[4][4];

        List<Node> path = aStarSearch(grid, start, goal);
        if (path != null) {
            for (Node node : path) {
                System.out.println("Path: (" + node.x + ", " + node.y + ")");
            }
        } else {
            System.out.println("No path found");
        }
    }
}
```

Time Complexity:

- **Time Complexity**: O(b^d), where b is the branching factor and d is the depth of the solution.
- **Space Complexity**: O(b^d) for the open and closed lists.

Kruskal's Algorithm (Minimum Spanning Tree)

Theory:

Kruskal's Algorithm is a greedy algorithm used to find the **Minimum Spanning Tree** (MST) for a connected, undirected graph. It works by selecting edges in increasing order of their weight and adding them to the MST if they do not form a cycle.

Algorithm:

1. Sort all the edges in non-decreasing order of their weight.
2. Pick the smallest edge and check if it forms a cycle with the MST formed so far. If it does not, include it in the MST.
3. Use **Union-Find** to detect cycles.
4. Repeat until the MST contains exactly (V-1) edges, where V is the number of vertices.

Code (Java):

java

```
import java.util.*;

public class Kruskal {
    static class Edge implements Comparable<Edge> {
        int src, dest, weight;

        public Edge(int src, int dest, int weight) {
            this.src = src;
            this.dest = dest;
            this.weight = weight;
        }

        @Override
        public int compareTo(Edge compareEdge) {
            return this.weight - compareEdge.weight;
        }
    }

    static class Subset {
        int parent, rank;
    }

    int V, E;
    Edge[] edges;

    Kruskal(int V, int E) {
        this.V = V;
```

```
    this.E = E;
    edges = new Edge[E];
}

int find(Subset[] subsets, int i) {
    if (subsets[i].parent != i) {
        subsets[i].parent = find(subsets, subsets[i].parent);
    }
    return subsets[i].parent;
}

void union(Subset[] subsets, int x, int y) {
    int rootX = find(subsets, x);
    int rootY = find(subsets, y);

    if (subsets[rootX].rank < subsets[rootY].rank) {
        subsets[rootX].parent = rootY;
    } else if (subsets[rootX].rank > subsets[rootY].rank) {
        subsets[rootY].parent = rootX;
    } else {
        subsets[rootY].parent = rootX;
        subsets[rootX].rank++;
    }
}

void kruskalMST() {
    Edge[] result = new Edge[V];
    int e = 0;
    int i = 0;
```

```
for (i = 0; i < V; ++i) {
    result[i] = new Edge(0, 0, 0);
}

Arrays.sort(edges);

Subset[] subsets = new Subset[V];
for (i = 0; i < V; ++i) {
    subsets[i] = new Subset();
    subsets[i].parent = i;
    subsets[i].rank = 0;
}

i = 0;
while (e < V - 1) {
    Edge nextEdge = edges[i++];
    int x = find(subsets, nextEdge.src);
    int y = find(subsets, nextEdge.dest);

    if (x != y) {
        result[e++] = nextEdge;
        union(subsets, x, y);
    }
}

System.out.println("Edges in the MST:");
for (i = 0; i < e; ++i) {
    System.out.println(result[i].src + " -- " + result[i].dest + " == " + result[i].weight);
}
```

```
    }

    public static void main(String[] args) {
        int V = 4;
        int E = 5;
        Kruskal graph = new Kruskal(V, E);

        graph.edges[0] = new Edge(0, 1, 10);
        graph.edges[1] = new Edge(0, 2, 6);
        graph.edges[2] = new Edge(0, 3, 5);
        graph.edges[3] = new Edge(1, 3, 15);
        graph.edges[4] = new Edge(2, 3, 4);

        graph.kruskalMST();
    }
}
```

Time Complexity:

- **Time Complexity**: O(E log E + V log V), where E is the number of edges and V is the number of vertices.
- **Space Complexity**: O(V) for the union-find structure.

Prim's Algorithm (Minimum Spanning Tree)

Theory:

Prim's Algorithm is a greedy algorithm that finds the **Minimum Spanning Tree** (MST) for a weighted, undirected graph. It starts with a single vertex and grows the MST by adding the smallest edge that connects a vertex in the MST to a vertex outside of it.

Algorithm:

1. Start with any vertex as the initial MST.
2. At each step, add the smallest edge that connects a vertex in the MST to a vertex outside the MST.
3. Repeat until all vertices are included in the MST.

Code (Java):

java

```
import java.util.*;

public class PrimMST {
    class Edge implements Comparable<Edge> {
        int vertex, weight;

        public Edge(int vertex, int weight) {
            this.vertex = vertex;
            this.weight = weight;
        }

        public int compareTo(Edge compareEdge) {
            return this.weight - compareEdge.weight;
        }
    }

    public void primMST(int[][] graph) {
        int V = graph.length;
        boolean[] inMST = new boolean[V];
        PriorityQueue<Edge> pq = new PriorityQueue<>();
        int[] key = new int[V];
        int[] parent = new int[V];

        Arrays.fill(key, Integer.MAX_VALUE);
        key[0] = 0;
        parent[0] = -1;
```

```
    pq.add(new Edge(0, key[0]));

    while (!pq.isEmpty()) {
        Edge node = pq.poll();
        int u = node.vertex;
        inMST[u] = true;

        for (int v = 0; v < V; v++) {
            if (graph[u][v] != 0 && !inMST[v] && graph[u][v] < key[v]) {
                key[v] = graph[u][v];
                pq.add(new Edge(v, key[v]));
                parent[v] = u;
            }
        }
    }

    printMST(parent, graph);
}

public void printMST(int[] parent, int[][] graph) {
    System.out.println("Edge \tWeight");
    for (int i = 1; i < graph.length; i++) {
        System.out.println(parent[i] + " - " + i + "\t" + graph[i][parent[i]]);
    }
}

public static void main(String[] args) {
    PrimMST prim = new PrimMST();
    int[][] graph = {
```

```
        {0, 2, 0, 6, 0},
        {2, 0, 3, 8, 5},
        {0, 3, 0, 0, 7},
        {6, 8, 0, 0, 9},
        {0, 5, 7, 9, 0}
    };
    prim.primMST(graph);
  }
}
```

Time Complexity:

- **Time Complexity**: O(V^2) for adjacency matrix representation.
- **Space Complexity**: O(V) for storing key and parent arrays.

Tarjan's Algorithm (Strongly Connected Components)

Theory:

Tarjan's Algorithm is used to find the **Strongly Connected Components (SCCs)** in a directed graph. It uses **depth-first search (DFS)** and a stack to track the nodes. The key idea is to use a low-link value to determine if a node is part of a strongly connected component.

Algorithm:

1. Perform a DFS traversal, keeping track of discovery times of nodes.
2. Use a low-link value to track the smallest discovery time reachable from the current node.
3. If a node's discovery time matches its low-link value, it is the root of an SCC.
4. Continue this process until all nodes are processed.

Code (Java):

java

```
import java.util.*;

public class TarjanSCC {
```

```
private int time = 0;
private int[] disc, low;
private boolean[] stackMember;
private Stack<Integer> stack;

public void SCCUtil(int u, List<List<Integer>> adj, int V) {
    disc[u] = low[u] = ++time;
    stack.push(u);
    stackMember[u] = true;

    for (int v : adj.get(u)) {
        if (disc[v] == -1) {
            SCCUtil(v, adj, V);
            low[u] = Math.min(low[u], low[v]);
        } else if (stackMember[v]) {
            low[u] = Math.min(low[u], disc[v]);
        }
    }

    int w = -1;
    if (low[u] == disc[u]) {
        while (w != u) {
            w = stack.pop();
            System.out.print(w + " ");
            stackMember[w] = false;
        }
        System.out.println();
    }
}
```

```
public void SCC(List<List<Integer>> adj, int V) {
    disc = new int[V];
    low = new int[V];
    stackMember = new boolean[V];
    stack = new Stack<>();

    Arrays.fill(disc, -1);
    Arrays.fill(low, -1);

    for (int i = 0; i < V; i++) {
        if (disc[i] == -1) {
            SCCUtil(i, adj, V);
        }
    }
}

public static void main(String[] args) {
    int V = 5;
    List<List<Integer>> adj = new ArrayList<>(V);
    for (int i = 0; i < V; i++) {
        adj.add(new ArrayList<>());
    }

    adj.get(0).add(2);
    adj.get(2).add(1);
    adj.get(1).add(0);
    adj.get(0).add(3);
    adj.get(3).add(4);
```

```
        TarjanSCC tarjan = new TarjanSCC();
        System.out.println("Strongly Connected Components in the graph:");
        tarjan.SCC(adj, V);
    }
}
```

Time Complexity:

- **Time Complexity**: O(V + E), where V is the number of vertices and E is the number of edges.
- **Space Complexity**: O(V) for storing discovery times and low-link values.

Kosaraju's Algorithm (Strongly Connected Components)

Theory:

Kosaraju's Algorithm is another method to find **Strongly Connected Components (SCCs)** in a directed graph. It is based on two depth-first searches (DFS). The first DFS is to sort vertices by finishing time, and the second DFS is to explore SCCs in the reverse graph.

Algorithm:

1. Perform DFS on the original graph, noting the finishing times of vertices.
2. Reverse the graph.
3. Perform DFS on the reversed graph in the order of decreasing finishing times from step 1. Each DFS will reveal a strongly connected component.

Code (Java):

java

```
import java.util.*;

public class KosarajuSCC {
    private void dfs(int v, boolean[] visited, List<List<Integer>> adj, Stack<Integer> stack) {
        visited[v] = true;
        for (int neighbor : adj.get(v)) {
            if (!visited[neighbor]) {
```

```
            dfs(neighbor, visited, adj, stack);
        }
    }
    stack.push(v);
}

private void reverseDfs(int v, boolean[] visited, List<List<Integer>> revAdj) {
    visited[v] = true;
    System.out.print(v + " ");
    for (int neighbor : revAdj.get(v)) {
        if (!visited[neighbor]) {
            reverseDfs(neighbor, visited, revAdj);
        }
    }
}

public void kosaraju(List<List<Integer>> adj, int V) {
    Stack<Integer> stack = new Stack<>();
    boolean[] visited = new boolean[V];

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

    List<List<Integer>> revAdj = new ArrayList<>();
    for (int i = 0; i < V; i++) {
        revAdj.add(new ArrayList<>());
```

```
    }
    for (int i = 0; i < V; i++) {
        for (int neighbor : adj.get(i)) {
            revAdj.get(neighbor).add(i);
        }
    }

    Arrays.fill(visited, false);

    while (!stack.isEmpty()) {
        int v = stack.pop();
        if (!visited[v]) {
            reverseDfs(v, visited, revAdj);
            System.out.println();
        }
    }
}

public static void main(String[] args) {
    int V = 5;
    List<List<Integer>> adj = new ArrayList<>();
    for (int i = 0; i < V; i++) {
        adj.add(new ArrayList<>());
    }

    adj.get(0).add(2);
    adj.get(2).add(1);
    adj.get(1).add(0);
    adj.get(0).add(3);
```

```
        adj.get(3).add(4);

        KosarajuSCC kosaraju = new KosarajuSCC();
        System.out.println("Strongly Connected Components in the graph:");
        kosaraju.kosaraju(adj, V);
    }
}
```

Time Complexity:

- **Time Complexity**: O(V + E), where V is the number of vertices and E is the number of edges.
- **Space Complexity**: O(V) for visited and stack.

Topological Sorting

Theory:

Topological Sorting is an ordering of vertices in a directed acyclic graph (DAG) such that for every directed edge u -> v, vertex u comes before vertex v in the ordering.

Algorithm:

1. Perform DFS on the graph, and track the finishing times of vertices.
2. Once DFS is completed, push vertices to a stack. The topological order is the reverse of the finishing order.

Code (Java):

java

```
import java.util.*;

public class TopologicalSort {
    private void topologicalSortUtil(int v, boolean[] visited, Stack<Integer> stack,
List<List<Integer>> adj) {
        visited[v] = true;
        for (int neighbor : adj.get(v)) {
            if (!visited[neighbor]) {
```

```
            topologicalSortUtil(neighbor, visited, stack, adj);
        }
    }
    stack.push(v);
}

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

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

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

public static void main(String[] args) {
    int V = 6;
    List<List<Integer>> adj = new ArrayList<>();
    for (int i = 0; i < V; i++) {
        adj.add(new ArrayList<>());
    }

    adj.get(5).add(2);
```

```
        adj.get(5).add(0);
        adj.get(4).add(0);
        adj.get(4).add(1);
        adj.get(2).add(3);
        adj.get(3).add(1);

        TopologicalSort topo = new TopologicalSort();
        System.out.println("Topological Sorting of the graph:");
        topo.topologicalSort(adj, V);
    }
}
```

Time Complexity:

- **Time Complexity**: O(V + E), where V is the number of vertices and E is the number of edges.
- **Space Complexity**: O(V) for visited array and stack.

Union-Find (Disjoint Set Union)

Theory:

Union-Find is a data structure that supports two operations: finding the set of a particular element and merging two sets. It's useful in various applications like detecting cycles in graphs and Kruskal's algorithm for Minimum Spanning Trees.

Algorithm:

1. **Find**: Determine which set a particular element is in (using path compression).
2. **Union**: Join two subsets into a single subset (using union by rank).

Code (Java):

java

```
public class UnionFind {
    int[] parent, rank;
```

```
public UnionFind(int n) {
    parent = new int[n];
    rank = new int[n];
    for (int i = 0; i < n; i++) {
        parent[i] = i;
        rank[i] = 0;
    }
}

public int find(int u) {
    if (parent[u] != u) {
        parent[u] = find(parent[u]); // Path compression
    }
    return parent[u];
}

public void union(int u, int v) {
    int rootU = find(u);
    int rootV = find(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]++;
        }
    }
```

```
    }

    public static void main(String[] args) {
        UnionFind uf = new UnionFind(5);

        uf.union(0, 2);
        uf.union(1, 3);
        uf.union(1, 4);
        System.out.println("Set of element 3: " + uf.find(3));
        System.out.println("Set of element 2: " + uf.find(2));
    }
}
```

Time Complexity:

- **Time Complexity**: O(α(n)) for each operation, where α is the inverse Ackermann function, which is nearly constant for practical purposes.
- **Space Complexity**: O(n).

Cycle Detection (Undirected and Directed Graphs)

Theory:

Cycle detection determines if a graph contains any cycles. In **undirected graphs**, cycles can be detected using DFS or Union-Find. In **directed graphs**, DFS can be used with a "visited" array to track recursion stack calls.

Algorithm for Undirected Graph (Using DFS):

1. Use DFS to explore each node.
2. If a node is visited and is not the parent of the current node, a cycle is found.

Code (Undirected Graph - DFS, Java):

java

```
import java.util.*;
```

```
public class CycleDetectionUndirected {
    public boolean isCyclicUtil(int v, boolean[] visited, int parent, List<List<Integer>> adj) {
        visited[v] = true;
        for (int neighbor : adj.get(v)) {
            if (!visited[neighbor]) {
                if (isCyclicUtil(neighbor, visited, v, adj)) {
                    return true;
                }
            } else if (neighbor != parent) {
                return true;
            }
        }
        return false;
    }

    public boolean isCyclic(List<List<Integer>> adj, int V) {
        boolean[] visited = new boolean[V];
        for (int i = 0; i < V; i++) {
            if (!visited[i]) {
                if (isCyclicUtil(i, visited, -1, adj)) {
                    return true;
                }
            }
        }
        return false;
    }

    public static void main(String[] args) {
        int V = 5;
```

```
        List<List<Integer>> adj = new ArrayList<>();
        for (int i = 0; i < V; i++) {
            adj.add(new ArrayList<>());
        }

        adj.get(0).add(1);
        adj.get(1).add(0);
        adj.get(1).add(2);
        adj.get(2).add(1);
        adj.get(2).add(0);
        adj.get(0).add(2);
        adj.get(3).add(4);
        adj.get(4).add(3);

        CycleDetectionUndirected cd = new CycleDetectionUndirected();
        if (cd.isCyclic(adj, V)) {
            System.out.println("Graph contains cycle");
        } else {
            System.out.println("Graph does not contain cycle");
        }
    }
}
```

Time Complexity (Undirected Graph):

- **Time Complexity**: O(V + E), where V is the number of vertices and E is the number of edges.
- **Space Complexity**: O(V) for visited array.

Greedy Algorithms

---

1. Activity Selection Problem

Theory:

The Activity Selection Problem involves selecting the maximum number of activities that don't overlap. Greedy algorithms are used to select activities based on their finish times.

Algorithm:

1. Sort activities by their finish times.
2. Select the first activity and then repeatedly select the next activity that starts after the last selected activity finishes.

Code (Java):

java

```
import java.util.Arrays;
import java.util.Comparator;

public class ActivitySelection {
    public static void activitySelection(int[] start, int[] end) {
        int n = start.length;
        Integer[] indices = new Integer[n];
        for (int i = 0; i < n; i++) {
            indices[i] = i;
        }

        // Sort activities by end time
        Arrays.sort(indices, Comparator.comparingInt(i -> end[i]));

        System.out.println("Selected activities:");
        int lastEnd = -1;
        for (int i : indices) {
            if (start[i] >= lastEnd) {
```

```
            System.out.println("Activity: " + i + " (" + start[i] + ", " + end[i] + ")");
            lastEnd = end[i];
        }
    }
  }

  public static void main(String[] args) {
    int[] start = {1, 3, 0, 5, 8, 5};
    int[] end = {2, 4, 6, 7, 9, 9};
    activitySelection(start, end);
  }
}
```

Time Complexity:

- **Time Complexity**: O(n log n), due to sorting.
- **Space Complexity**: O(n)

---

2. Kruskal's Algorithm (Minimum Spanning Tree)

Theory:

Kruskal's algorithm finds the Minimum Spanning Tree (MST) of a graph by sorting edges in increasing order and adding them one by one, ensuring no cycles are formed.

Algorithm:

1. Sort all edges by their weights.
2. Use a union-find data structure to add edges to the MST, avoiding cycles.
3. Continue until the MST contains (V-1) edges.

Code (Java):

java

```
import java.util.*;

public class Kruskal {
```

```
static class Edge {
    int src, dest, weight;

    Edge(int src, int dest, int weight) {
        this.src = src;
        this.dest = dest;
        this.weight = weight;
    }
}

static class DisjointSet {
    int[] parent, rank;

    DisjointSet(int n) {
        parent = new int[n];
        rank = new int[n];
        for (int i = 0; i < n; i++) {
            parent[i] = i;
            rank[i] = 0;
        }
    }

    int find(int u) {
        if (u != parent[u]) {
            parent[u] = find(parent[u]);
        }
        return parent[u];
    }
```

```
    void union(int u, int v) {
        int rootU = find(u);
        int rootV = find(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]++;
            }
        }
    }
}

public static void kruskal(int V, List<Edge> edges) {
    Collections.sort(edges, Comparator.comparingInt(e -> e.weight));

    DisjointSet ds = new DisjointSet(V);
    List<Edge> mst = new ArrayList<>();

    for (Edge edge : edges) {
        int u = edge.src;
        int v = edge.dest;
        if (ds.find(u) != ds.find(v)) {
            ds.union(u, v);
            mst.add(edge);
```

```
            }
        }

        System.out.println("Edges in MST:");
        for (Edge edge : mst) {
            System.out.println(edge.src + " - " + edge.dest + ": " + edge.weight);
        }
    }

    public static void main(String[] args) {
        int V = 4;
        List<Edge> edges = Arrays.asList(
            new Edge(0, 1, 10),
            new Edge(0, 2, 6),
            new Edge(0, 3, 5),
            new Edge(1, 3, 15),
            new Edge(2, 3, 4)
        );

        kruskal(V, edges);
    }
}
```

Time Complexity:

- **Time Complexity**: O(E log E), due to sorting and union-find operations.
- **Space Complexity**: O(V + E)

---

3. Prim's Algorithm (Minimum Spanning Tree)

Theory:

Prim's algorithm constructs the Minimum Spanning Tree (MST) by starting from a single vertex and expanding the tree by adding the shortest edge from the tree to a new vertex.

Algorithm:

1. Start from an arbitrary node.
2. Add the smallest edge that connects the tree to a new vertex.
3. Repeat until all vertices are included in the MST.

Code (Java):

java

```
import java.util.*;

public class Prim {
   static class Edge {
      int dest, weight;

      Edge(int dest, int weight) {
         this.dest = dest;
         this.weight = weight;
      }
   }

   public static void prim(int V, List<List<Edge>> adj) {
      boolean[] inMST = new boolean[V];
      int[] key = new int[V];
      int[] parent = new int[V];
      PriorityQueue<Edge> pq = new PriorityQueue<>(Comparator.comparingInt(e -> e.weight));

      Arrays.fill(key, Integer.MAX_VALUE);
      key[0] = 0;
      pq.add(new Edge(0, 0));
      parent[0] = -1;
```

```
    while (!pq.isEmpty()) {
        int u = pq.poll().dest;
        inMST[u] = true;

        for (Edge edge : adj.get(u)) {
            int v = edge.dest;
            int weight = edge.weight;

            if (!inMST[v] && weight < key[v]) {
                key[v] = weight;
                pq.add(new Edge(v, key[v]));
                parent[v] = u;
            }
        }
    }

    System.out.println("Edges in MST:");
    for (int i = 1; i < V; i++) {
        System.out.println(parent[i] + " - " + i + ": " + key[i]);
    }
}

public static void main(String[] args) {
    int V = 5;
    List<List<Edge>> adj = new ArrayList<>();
    for (int i = 0; i < V; i++) {
        adj.add(new ArrayList<>());
    }
```

```
        adj.get(0).add(new Edge(1, 2));
        adj.get(0).add(new Edge(3, 6));
        adj.get(1).add(new Edge(0, 2));
        adj.get(1).add(new Edge(2, 3));
        adj.get(1).add(new Edge(3, 8));
        adj.get(1).add(new Edge(4, 5));
        adj.get(2).add(new Edge(1, 3));
        adj.get(2).add(new Edge(4, 7));
        adj.get(3).add(new Edge(0, 6));
        adj.get(3).add(new Edge(1, 8));
        adj.get(3).add(new Edge(4, 9));
        adj.get(4).add(new Edge(1, 5));
        adj.get(4).add(new Edge(2, 7));
        adj.get(4).add(new Edge(3, 9));

        prim(V, adj);
    }
}
```

Time Complexity:

- **Time Complexity**: O(E log V), due to priority queue operations.
- **Space Complexity**: O(V + E)

Backtracking Algorithms

---

1. N-Queens Problem

Theory:

The N-Queens problem involves placing N queens on an N x N chessboard so that no two queens threaten each other. Backtracking helps to explore all possible placements.

Algorithm:

1. Place queens one by one in different columns.
2. Check if the current placement is safe.
3. If safe, recursively place the next queen.
4. Backtrack if placing further queens is not possible.

Code (Java):

java

```
public class NQueens {
    public static void solveNQueens(int n) {
        int[] board = new int[n];
        solve(board, 0, n);
    }

    private static void solve(int[] board, int row, int n) {
        if (row == n) {
            printBoard(board);
            return;
        }

        for (int col = 0; col < n; col++) {
            if (isSafe(board, row, col, n)) {
                board[row] = col;
                solve(board, row + 1, n);
            }
        }
    }

    private static boolean isSafe(int[] board, int row, int col, int n) {
```

```
        for (int i = 0; i < row; i++) {
            if (board[i] == col || board[i] - i == col - row || board[i] + i == col + row) {
                return false;
            }
        }
        return true;
    }

    private static void printBoard(int[] board) {
        int n = board.length;
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (board[i] == j) {
                    System.out.print("Q ");
                } else {
                    System.out.print(". ");
                }
            }
            System.out.println();
        }
        System.out.println();
    }

    public static void main(String[] args) {
        int n = 4;
        solveNQueens(n);
    }
}
```

Time Complexity:

- **Time Complexity**: O(N!), since the solution involves exploring permutations.
- **Space Complexity**: O(N^2) for storing the board.

---

2. Sudoku Solver

Theory:

The Sudoku Solver fills a 9x9 Sudoku grid using backtracking to ensure that each number 1-9 appears only once per row, column, and 3x3 subgrid.

Algorithm:

1. Find an empty cell.
2. Try placing digits 1 through 9.
3. Check if the placement is valid.
4. Recursively solve the next cell.
5. Backtrack if needed.

Code (Java):

java

```
public class SudokuSolver {
    public static void solveSudoku(int[][] board) {
        solve(board);
    }

    private static boolean solve(int[][] board) {
        for (int row = 0; row < 9; row++) {
            for (int col = 0; col < 9; col++) {
                if (board[row][col] == 0) {
                    for (int num = 1; num <= 9; num++) {
                        if (isValid(board, row, col, num)) {
                            board[row][col] = num;
                            if (solve(board)) {
                                return true;
```

```
                    }
                    board[row][col] = 0;  // Backtrack
                }
            }
            return false;  // No valid number found
        }
    }
    }
    return true;  // Sudoku solved
}

private static boolean isValid(int[][] board, int row, int col, int num) {
    for (int i = 0; i < 9; i++) {
        if (board[row][i] == num || board[i][col] == num ||
            board[row - row % 3 + i / 3][col - col % 3 + i % 3] == num) {
            return false;
        }
    }
    return true;
}

private static void printBoard(int[][] board) {
    for (int[] row : board) {
        for (int num : row) {
            System.out.print(num + " ");
        }
        System.out.println();
    }
}
```

```
    public static void main(String[] args) {
        int[][] board = {
            {5, 3, 0, 0, 7, 0, 0, 0, 0},
            {6, 0, 0, 1, 9, 5, 0, 0, 0},
            {0, 9, 8, 0, 0, 0, 0, 6, 0},
            {8, 0, 0, 0, 6, 0, 0, 0, 3},
            {4, 0, 0, 8, 0, 3, 0, 0, 1},
            {7, 0, 0, 0, 2, 0, 0, 0, 6},
            {0, 6, 0, 0, 0, 0, 2, 8, 0},
            {0, 0, 0, 4, 1, 9, 0, 0, 5},
            {0, 0, 0, 0, 8, 0, 0, 7, 9}
        };

        solveSudoku(board);
        printBoard(board);
    }
}
```

Time Complexity:

- **Time Complexity**: O(9^(N^2)), where N is the size of the grid (9).
- **Space Complexity**: O(1)

Bit Manipulation Algorithms

---

1. Counting Set Bits

Theory:

Counting set bits involves determining the number of bits that are set to 1 in a binary representation of a number.

Algorithm:

1. Iterate through each bit of the number.
2. Use bitwise AND with 1 to check if the least significant bit is set.
3. Right shift the number and repeat until it becomes 0.

Code (Java):

java

```
public class CountSetBits {
    public static int countSetBits(int n) {
        int count = 0;
        while (n > 0) {
            count += n & 1;
            n >>= 1;
        }
        return count;
    }

    public static void main(String[] args) {
        int n = 29;  // Binary: 11101
        System.out.println("Number of set bits: " + countSetBits(n));
    }
}
```

Time Complexity:

- **Time Complexity**: O(log n), where n is the number of bits.
- **Space Complexity**: O(1)

---

2. Finding the Only Non-Duplicated Element

Theory:

Given an array where every element appears twice except for one, find the non-duplicated element using bitwise XOR.

Algorithm:

1. Initialize a result variable to 0.
2. XOR each element of the array with the result.
3. The result will be the non-duplicated element.

Code (Java):

java

```
public class SingleNonDuplicate {
    public static int findSingle(int[] nums) {
        int result = 0;
        for (int num : nums) {
            result ^= num;
        }
        return result;
    }

    public static void main(String[] args) {
        int[] nums = {1, 2, 3, 2, 1};
        System.out.println("The single element is: " + findSingle(nums));
    }
}
```

Time Complexity:

- **Time Complexity**: O(n), where n is the number of elements.
- **Space Complexity**: O(1)

Tree Algorithms

Tree Traversals (In-order, Pre-order, Post-order)

Theory:

Tree Traversals are methods to visit all the nodes of a tree. The common traversals are:

1. **In-order** (Left, Root, Right): Visits the left subtree, then the root, then the right subtree.
2. **Pre-order** (Root, Left, Right): Visits the root first, then the left subtree, and then the right subtree.
3. **Post-order** (Left, Right, Root): Visits the left subtree, then the right subtree, and finally the root.

Code (Java):

java

```
class Node {
    int data;
    Node left, right;

    public Node(int item) {
        data = item;
        left = right = null;
    }
}

public class TreeTraversal {
    Node root;

    void inorder(Node node) {
        if (node == null)
            return;
        inorder(node.left);
        System.out.print(node.data + " ");
        inorder(node.right);
    }
```

```
void preorder(Node node) {
    if (node == null)
        return;
    System.out.print(node.data + " ");
    preorder(node.left);
    preorder(node.right);
}

void postorder(Node node) {
    if (node == null)
        return;
    postorder(node.left);
    postorder(node.right);
    System.out.print(node.data + " ");
}

public static void main(String[] args) {
    TreeTraversal tree = new TreeTraversal();
    tree.root = new Node(1);
    tree.root.left = new Node(2);
    tree.root.right = new Node(3);
    tree.root.left.left = new Node(4);
    tree.root.left.right = new Node(5);

    System.out.println("Inorder traversal:");
    tree.inorder(tree.root);

    System.out.println("\nPreorder traversal:");
    tree.preorder(tree.root);
```

```
        System.out.println("\nPostorder traversal:");
        tree.postorder(tree.root);
    }
}
```

Time Complexity:

- **Time Complexity**: O(n), where n is the number of nodes in the tree.
- **Space Complexity**: O(h), where h is the height of the tree.

Binary Search Tree (BST) Insertion, Deletion, Search

Theory:

A **Binary Search Tree (BST)** is a binary tree in which every node has a value such that all the values in the left subtree are smaller, and all the values in the right subtree are larger than the node itself.

- **Insertion**: Traverse the tree to find the correct position and insert the new node.
- **Search**: Traverse the tree, comparing the value with the current node, and move left or right.
- **Deletion**: Three cases: node to be deleted is a leaf, has one child, or has two children. The most complex case is when the node has two children, where we find the in-order successor or predecessor.

Code (Java):

java

```
class BST {
    class Node {
        int key;
        Node left, right;

        public Node(int item) {
            key = item;
            left = right = null;
```

```
    }
}

Node root;

BST() {
    root = null;
}

// Insertion
void insert(int key) {
    root = insertRec(root, key);
}

Node insertRec(Node root, int key) {
    if (root == null) {
        root = new Node(key);
        return root;
    }
    if (key < root.key)
        root.left = insertRec(root.left, key);
    else if (key > root.key)
        root.right = insertRec(root.right, key);

    return root;
}

// Search
boolean search(int key) {
```

```
    return searchRec(root, key);
}

boolean searchRec(Node root, int key) {
    if (root == null)
        return false;
    if (root.key == key)
        return true;
    if (root.key > key)
        return searchRec(root.left, key);
    return searchRec(root.right, key);
}

// Deletion
void delete(int key) {
    root = deleteRec(root, key);
}

Node deleteRec(Node root, int key) {
    if (root == null)
        return root;

    if (key < root.key)
        root.left = deleteRec(root.left, key);
    else if (key > root.key)
        root.right = deleteRec(root.right, key);
    else {
        if (root.left == null)
            return root.right;
```

```
        else if (root.right == null)
            return root.left;

        root.key = minValue(root.right);
        root.right = deleteRec(root.right, root.key);
    }

    return root;
}

int minValue(Node root) {
    int minValue = root.key;
    while (root.left != null) {
        minValue = root.left.key;
        root = root.left;
    }
    return minValue;
}

public static void main(String[] args) {
    BST tree = new BST();
    tree.insert(50);
    tree.insert(30);
    tree.insert(20);
    tree.insert(40);
    tree.insert(70);
    tree.insert(60);
    tree.insert(80);
```

```
        System.out.println("Search 40: " + tree.search(40));
        tree.delete(40);
        System.out.println("Search 40 after deletion: " + tree.search(40));
    }
}
```

Time Complexity:

- **Insertion**: O(h), where h is the height of the tree.
- **Search**: O(h).
- **Deletion**: O(h).

Lowest Common Ancestor (LCA)

Theory:

The **Lowest Common Ancestor (LCA)** of two nodes p and q in a tree is the deepest node that has both p and q as descendants. The LCA can be found efficiently using recursion.

Algorithm:

1. If the current node is p or q, return the node.
2. Recursively search the left and right subtrees.
3. If both left and right return non-null, the current node is the LCA.

Code (Java):

java

```
class LCA {
    static class Node {
        int data;
        Node left, right;

        Node(int value) {
            data = value;
            left = right = null;
        }
```

```
}

Node root;

Node findLCA(Node node, int n1, int n2) {
   if (node == null)
      return null;

   if (node.data == n1 || node.data == n2)
      return node;

   Node left_lca = findLCA(node.left, n1, n2);
   Node right_lca = findLCA(node.right, n1, n2);

   if (left_lca != null && right_lca != null)
      return node;

   return (left_lca != null) ? left_lca : right_lca;
}

public static void main(String[] args) {
   LCA tree = new LCA();
   tree.root = new Node(1);
   tree.root.left = new Node(2);
   tree.root.right = new Node(3);
   tree.root.left.left = new Node(4);
   tree.root.left.right = new Node(5);
   tree.root.right.left = new Node(6);
   tree.root.right.right = new Node(7);
```

```
        System.out.println("LCA(4, 5): " + tree.findLCA(tree.root, 4, 5).data);
        System.out.println("LCA(4, 6): " + tree.findLCA(tree.root, 4, 6).data);
    }
}
```

Time Complexity:

- **Time Complexity**: O(n), where n is the number of nodes in the tree.
- **Space Complexity**: O(h), where h is the height of the tree.

AVL Tree (Self-Balancing BST)

Theory:

An **AVL Tree** is a self-balancing Binary Search Tree where the difference between heights of left and right subtrees (called the **balance factor**) is at most 1 for all nodes. When the balance factor becomes more than 1 or less than -1, rotations are performed to balance the tree.

- **Insertion**: Similar to a BST but includes updating the height and balancing the tree by performing rotations (left-right, right-left).
- **Deletion**: Similar to insertion, but also involves balancing after removal.

Code (Java):

java

```
class AVLTree {
    class Node {
        int key, height;
        Node left, right;

        Node(int d) {
            key = d;
            height = 1;
        }
    }
```

```
Node root;

int height(Node N) {
    if (N == null)
        return 0;
    return N.height;
}

int getBalance(Node N) {
    if (N == null)
        return 0;
    return height(N.left) - height(N.right);
}

Node rightRotate(Node y) {
    Node x = y.left;
    Node T2 = x.right;
    x.right = y;
    y.left = T2;
    y.height = Math.max(height(y.left), height(y.right)) + 1;
    x.height = Math.max(height(x.left), height(x.right)) + 1;
    return x;
}

Node leftRotate(Node x) {
    Node y = x.right;
    Node T2 = y.left;
    y.left = x;
```

```
    x.right = T2;
    x.height = Math.max(height(x.left), height(x.right)) + 1;
    y.height = Math.max(height(y.left), height(y.right)) + 1;
    return y;
}

Node insert(Node node, int key) {
    if (node == null)
        return new Node(key);

    if (key < node.key)
        node.left = insert(node.left, key);
    else if (key > node.key)
        node.right = insert(node.right, key);
    else
        return node;

    node.height = 1 + Math.max(height(node.left), height(node.right));

    int balance = getBalance(node);

    // Left Left Case
    if (balance > 1 && key < node.left.key)
        return rightRotate(node);

    // Right Right Case
    if (balance < -1 && key > node.right.key)
        return leftRotate(node);
```

```
    // Left Right Case
    if (balance > 1 && key > node.left.key) {
        node.left = leftRotate(node.left);
        return rightRotate(node);
    }

    // Right Left Case
    if (balance < -1 && key < node.right.key) {
        node.right = rightRotate(node.right);
        return leftRotate(node);
    }

    return node;
}

void preOrder(Node node) {
    if (node != null) {
        System.out.print(node.key + " ");
        preOrder(node.left);
        preOrder(node.right);
    }
}

public static void main(String[] args) {
    AVLTree tree = new AVLTree();
    tree.root = tree.insert(tree.root, 10);
    tree.root = tree.insert(tree.root, 20);
    tree.root = tree.insert(tree.root, 30);
    tree.root = tree.insert(tree.root, 40);
```

```
        tree.root = tree.insert(tree.root, 50);
        tree.root = tree.insert(tree.root, 25);

        System.out.println("Preorder traversal of the AVL tree is:");
        tree.preOrder(tree.root);
    }
}
```

Time Complexity:

- **Insertion**: O(log n), where n is the number of nodes.
- **Rotation**: O(1).
- **Space Complexity**: O(log n) due to the recursive stack.

Mathematical Algorithms

Euclidean Algorithm (GCD)

Theory:

The **Euclidean Algorithm** is used to compute the **Greatest Common Divisor (GCD)** of two integers, which is the largest integer that divides both without leaving a remainder. The algorithm is based on the property that GCD(a, b) = GCD(b, a % b).

Algorithm:

1. Let a and b be two numbers.
2. While b ≠ 0, set a = b and b = a % b.
3. When b becomes 0, a will be the GCD.

Code (Java):

java

```
public class EuclideanAlgorithm {
    public static int gcd(int a, int b) {
        while (b != 0) {
            int temp = b;
```

```
        b = a % b;
        a = temp;
    }
    return a;
  }

  public static void main(String[] args) {
    int a = 56, b = 98;
    System.out.println("GCD of " + a + " and " + b + " is: " + gcd(a, b));
  }
}
```

Time Complexity:

- **Time Complexity**: O(log(min(a, b))).
- **Space Complexity**: O(1).

Sieve of Eratosthenes (Prime Numbers)

Theory:

The **Sieve of Eratosthenes** is an efficient algorithm for finding all prime numbers up to a given limit n. It marks the multiples of each prime starting from 2.

Algorithm:

1. Create an array of boolean values, initialized to true.
2. Start with the first prime number, 2.
3. Mark all multiples of 2 as non-prime.
4. Move to the next number that is still marked as prime, and repeat the process for its multiples.
5. Continue until you have processed numbers up to √n.

Code (Java):

java

```
public class SieveOfEratosthenes {
```

```
    public static void sieve(int n) {
        boolean[] isPrime = new boolean[n + 1];
        for (int i = 2; i <= n; i++) {
            isPrime[i] = true;
        }

        for (int p = 2; p * p <= n; p++) {
            if (isPrime[p]) {
                for (int i = p * p; i <= n; i += p) {
                    isPrime[i] = false;
                }
            }
        }

        for (int i = 2; i <= n; i++) {
            if (isPrime[i]) {
                System.out.print(i + " ");
            }
        }
    }

    public static void main(String[] args) {
        int n = 50;
        System.out.println("Prime numbers up to " + n + ":");
        sieve(n);
    }
}
```

Time Complexity:

- **Time Complexity**: O(n log log n).

- **Space Complexity**: O(n).

Modular Exponentiation

Theory:

Modular Exponentiation efficiently computes (base^exponent) % mod using the property of exponentiation by squaring. This method is efficient for very large numbers.

Algorithm:

1. If exponent == 0, return 1.
2. Recursively compute base^exponent/2 % mod and square the result.
3. If the exponent is odd, multiply by the base once more.

Code (Java):

java

```
public class ModularExponentiation {
    public static long modExp(long base, long exp, long mod) {
        if (exp == 0) return 1;
        long half = modExp(base, exp / 2, mod);
        half = (half * half) % mod;
        if (exp % 2 != 0) {
            half = (half * base) % mod;
        }
        return half;
    }

    public static void main(String[] args) {
        long base = 2, exp = 10, mod = 1000000007;
        System.out.println("Result: " + modExp(base, exp, mod));
    }
}
```

Time Complexity:

- **Time Complexity**: O(log exp).
- **Space Complexity**: O(log exp) for recursion.

Other Algorithms

---

1. Floyd-Warshall Algorithm (Shortest Paths)

Theory:

The Floyd-Warshall algorithm finds the shortest paths between all pairs of vertices in a weighted graph, handling negative weights but no negative weight cycles.

Algorithm:

1. Initialize the distance matrix with edge weights.
2. For each pair of nodes, check if an intermediate node can provide a shorter path.
3. Update the distance matrix accordingly.

Code (Java):

java

```
public class FloydWarshall {
    static final int INF = 99999;

    public static void floydWarshall(int[][] graph) {
        int V = graph.length;
        int[][] dist = new int[V][V];

        for (int i = 0; i < V; i++) {
            for (int j = 0; j < V; j++) {
                dist[i][j] = graph[i][j];
            }
        }
```

```
    for (int k = 0; k < V; k++) {
        for (int i = 0; i < V; i++) {
            for (int j = 0; j < V; j++) {
                if (dist[i][j] > dist[i][k] + dist[k][j]) {
                    dist[i][j] = dist[i][k] + dist[k][j];
                }
            }
        }
    }

    printSolution(dist);
}

private static void printSolution(int[][] dist) {
    int V = dist.length;
    System.out.println("Shortest distances between every pair of vertices:");
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            if (dist[i][j] == INF) {
                System.out.print("INF ");
            } else {
                System.out.print(dist[i][j] + " ");
            }
        }
        System.out.println();
    }
}
```

```
    public static void main(String[] args) {
        int V = 4;
        int[][] graph = {
            {0, 3, INF, 7},
            {8, 0, 2, INF},
            {5, INF, 0, 1},
            {2, INF, INF, 0}
        };

        floydWarshall(graph);
    }
}
```

Time Complexity:

- **Time Complexity**: O(V^3), where V is the number of vertices.
- **Space Complexity**: O(V^2)

---

2. Knapsack Problem (0/1 Knapsack)

Theory:

The Knapsack problem involves selecting items with given weights and values to maximize the total value without exceeding the weight capacity. This is solved using dynamic programming.

Algorithm:

1. Initialize a DP table where dp[i][w] represents the maximum value achievable with the first i items and weight capacity w.
2. Fill the table based on whether including an item provides a higher value than excluding it.
3. The final value is the maximum value that can be achieved with the given weight capacity.

Code (Java):

java

```
public class Knapsack {
    public static int knapSack(int W, int[] weights, int[] values, 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 (weights[i - 1] <= w) {
                    dp[i][w] = Math.max(values[i - 1] + dp[i - 1][w - weights[i - 1]], dp[i - 1][w]);
                } else {
                    dp[i][w] = dp[i - 1][w];
                }
            }
        }
        return dp[n][W];
    }

    public static void main(String[] args) {
        int[] values = {60, 100, 120};
        int[] weights = {10, 20, 30};
        int W = 50;
        int n = values.length;
        System.out.println("Maximum value in Knapsack = " + knapSack(W, weights, values, n));
    }
}
```

Time Complexity:

- **Time Complexity**: O(n * W), where n is the number of items and W is the maximum weight capacity.
- **Space Complexity**: O(n * W)