**Week 1 - Data Structures: Arrays**

**Learnings**

* **Introduction to Arrays**:
  + **Definition**: An array is a data structure that holds a fixed number of elements of the same type.

**Array Creation**:

int[] arr = new int[5]; // Creating an array of size 5

**Insertion**:

int[] arr = {1, 2, 3, 4, 5};

int pos = 2; // Position where to insert

int newElement = 10;

// Shifting elements to the right

for (int i = arr.length - 1; i > pos; i--) {

arr[i] = arr[i - 1];

}

arr[pos] = newElement;

**Deletion**:

int[] arr = {1, 2, 3, 4, 5};

int pos = 2; // Position to delete

// Shifting elements to the left

for (int i = pos; i < arr.length - 1; i++) {

arr[i] = arr[i + 1];

}

**Traversal**:

for (int i = 0; i < arr.length; i++) {

System.out.println(arr[i]);

* **Introduction to Linked Lists**:
* **Definition**: A linked list is a linear data structure where each element (node) contains a reference (or link) to the next node in the sequence.
* **Types of Linked Lists**:
  + **Singly Linked List**: Each node has a reference to the next node.
  + **Doubly Linked List**: Each node has references to both the next and previous nodes.
  + **Circular Linked List**: The last node points back to the first node, forming a circle

**Defining a basic node class:**

class Node {

int data;

Node next;

Node(int data) {

this.data = data;

this.next = null;

}

}

**Creation**:

class LinkedList {

Node head;

// Constructor to create an empty list

LinkedList() {

this.head = null;

}

}

**Insertion**:

public void insertAtBeginning(int data) {

Node newNode = new Node(data);

newNode.next = head;

head = newNode;

}

**//Inserting a node at the end:**

public void insertAtEnd(int data) {

Node newNode = new Node(data);

if (head == null) {

head = newNode;

} else {

Node temp = head;

while (temp.next != null) {

temp = temp.next;

}

temp.next = newNode;

}

}

**Deletion**:

public void deleteNode(int key) {

Node temp = head, prev = null;

// If head node holds the key to be deleted

if (temp != null && temp.data == key) {

head = temp.next;

return;

}

// Search for the key to be deleted

while (temp != null && temp.data != key) {

prev = temp;

temp = temp.next;

}

// If key was not present

if (temp == null) return;

// Unlink the node

prev.next = temp.next;

}

**Traversal**:

public void traverse() {

Node temp = head;

while (temp != null) {

System.out.print(temp.data + " -> ");

temp = temp.next;

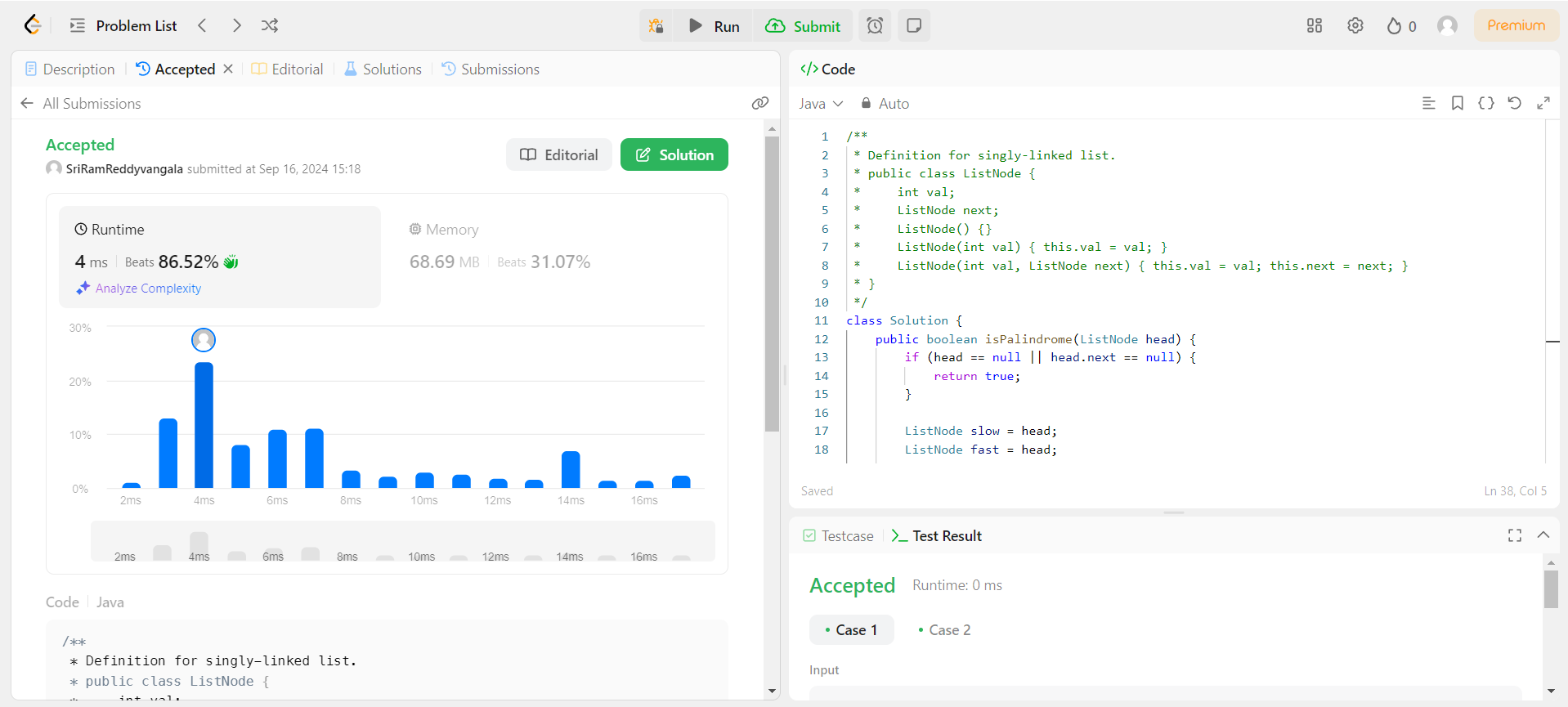
}

System.out.println("null");

}

* **Hands-On Practice:**

**Leet Code Problem:**



**Challenges and how I overcome it:**

1. **Finding the Middle of the List:**

Challenge: Correctly finding the middle node of the linked list is crucial, and ensuring that this is done efficiently is challenging. The slow and fast pointer technique must handle both even- and odd-length lists.

Solution: The slow pointer moves one step while the fast pointer moves two steps at a time. When the fast pointer reaches the end, the slow pointer will be at the middle of the list.

2. **Reversing the Second Half of the List:**

Challenge: Reversing the second half of the list in place requires careful manipulation of pointers. The difficulty lies in making sure that the reversal doesn't affect the first half and doesn't lose reference to any node.

Solution: Use a temporary pointer to store the next node before changing links during the reversal process. This ensures that each node is correctly reversed without breaking the list.

3. **Comparing Two Halves:**

Challenge: After splitting the list into two halves (first half and reversed second half), comparing them node by node is tricky, especially in handling lists of different lengths (e.g., odd-length lists).

Solution: The comparison is done node by node. If the lengths of the two halves differ (due to an odd number of nodes), the middle element can be skipped during the comparison since it doesn't affect the palindrome condition.

4. **Edge Cases:**

Challenge: Handling edge cases like:

A list with only one node (which is trivially a palindrome).

An empty list (which is also trivially a palindrome).

A list with two nodes that are not equal.

Solution: These cases can be addressed early by checking if the head is null or if the list contains only one node, returning true immediately for these conditions.

* **Introduction to Stacks**:
* **Definition**: A stack is a linear data structure that follows the Last In First Out (LIFO) principle.
* **Key Operations**:
  + **Push**: Add an element to the top of the stack.
  + **Pop**: Remove the top element from the stack.
  + **Peek**: Get the value of the top element without removing it.

**Basic Stack Operations Using an Array**:

class Stack {

int[] stack;

int top;

int capacity;

Stack(int size) {

stack = new int[size];

capacity = size;

top = -1;

}

void push(int x) {

if (top == capacity - 1) {

System.out.println("Stack Overflow");

return;

}

stack[++top] = x;

}

int pop() {

if (top == -1) {

System.out.println("Stack Underflow");

return -1;

}

return stack[top--];

}

int peek() {

if (top == -1) {

System.out.println("Stack is empty");

return -1;

}

return stack[top];

}

boolean isEmpty() {

return top == -1;

}

}

* **Introduction to Queues**:
* **Definition**: A queue is a linear data structure that follows the First In First Out (FIFO) principle.
* **Key Operations**:
  + **Enqueue**: Add an element to the back of the queue.
  + **Dequeue**: Remove an element from the front of the queue.
  + **Peek/Front**: Get the value of the front element without removing it.

**Basic Queue Operations Using an Array:**

class Queue {

int[] queue;

int front, rear, capacity;

Queue(int size) {

queue = new int[size];

capacity = size;

front = rear = -1;

}

void enqueue(int x) {

if (rear == capacity - 1) {

System.out.println("Queue Overflow");

return;

}

if (front == -1) front = 0;

queue[++rear] = x;

}

int dequeue() {

if (front == -1 || front > rear) {

System.out.println("Queue Underflow");

return -1;

}

return queue[front++];

}

int peek() {

if (front == -1 || front > rear) {

System.out.println("Queue is empty");

return -1;

}

return queue[front];

}

boolean isEmpty() {

return front == -1 || front > rear;

}

}

* **Introduction to Hash Tables**:
* **Definition**: A hash table is a data structure that maps keys to values using a hash function.
* **Key Operations**:
  + **Insert**: Insert a key-value pair.
  + **Search**: Retrieve the value for a given key.
  + **Delete**: Remove a key-value pair.

**Hash Table with Chaining:**

class HashTable {

private LinkedList<Node>[] table;

private int size;

class Node {

String key;

int value;

Node(String key, int value) {

this.key = key;

this.value = value;

}

}

HashTable(int size) {

table = new LinkedList[size];

for (int i = 0; i < size; i++) {

table[i] = new LinkedList<>();

}

this.size = size;

}

private int hash(String key) {

return Math.abs(key.hashCode()) % size;

}

void insert(String key, int value) {

int index = hash(key);

for (Node node : table[index]) {

if (node.key.equals(key)) {

node.value = value;

return;

}

}

table[index].add(new Node(key, value));

}

int get(String key) {

int index = hash(key);

for (Node node : table[index]) {

if (node.key.equals(key)) {

return node.value;

}

}

return -1; // Key not found

}

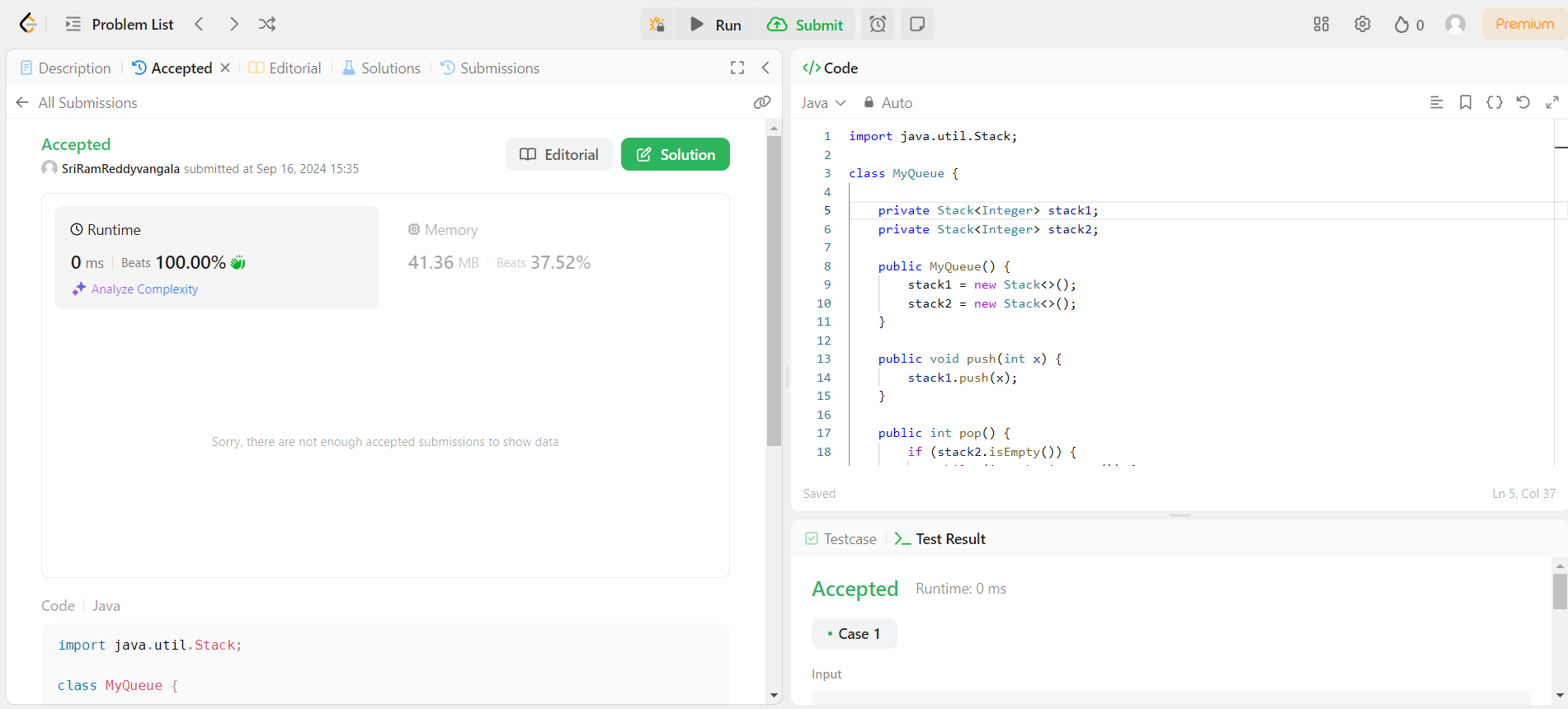
void remove(String key) {

int index = hash(key);

table[index].removeIf(node -> node.key.equals(key));

}

}  
  
**Leet Code:**



**Challenges and how I overcome it:**

**1.Maintaining Queue Order Using Stacks:**

Challenge: Stacks are LIFO (Last In, First Out) data structures, while a queue is FIFO (First In, First Out). The core challenge lies in reversing the LIFO order of the stack to achieve FIFO behavior for the queue.

Solution: By using two stacks (stack1 for push operations and stack2 for pop/peek operations), the transfer of elements from stack1 to stack2 reverses the order, allowing the queue to behave as FIFO.

2. **Efficiently Transferring Elements Between Stacks:**

Challenge: The key operation occurs when transferring elements from stack1 to stack2. This can be inefficient if done frequently. To maintain efficiency, transfers should only happen when stack2 is empty.

Solution: Only transfer elements from stack1 to stack2 when stack2 is empty. This ensures that each element is moved at most once, resulting in amortized O(1) complexity for pop() and peek().

**Week 1 - Algorithms: Sorting and Searching**

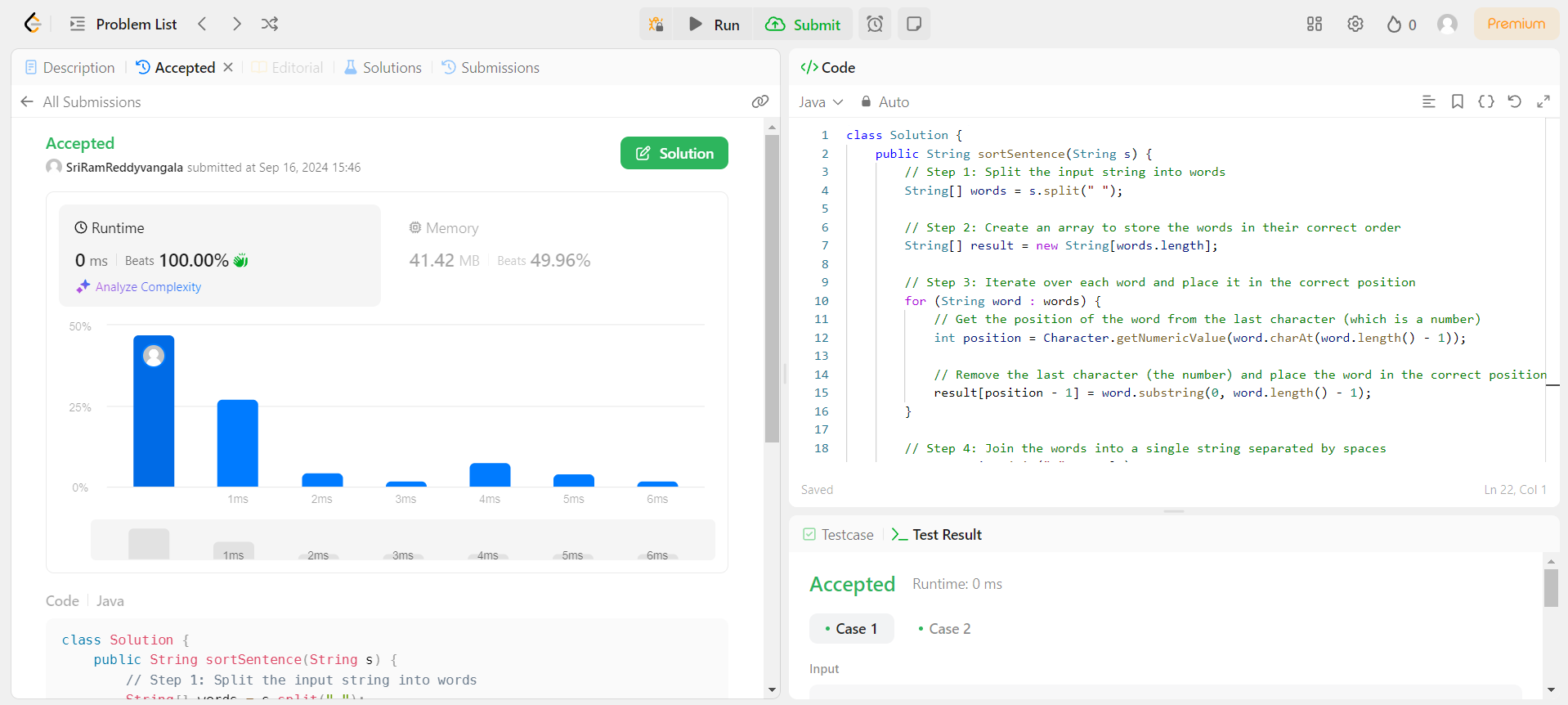
**Learnings**

* **Introduction to Sorting Algorithms**:
  + Sorting algorithms are used to arrange data in a particular order, typically in ascending or descending order.
  + Common sorting algorithms include:
    - **Bubble Sort**: Repeatedly compares adjacent elements and swaps them if they are in the wrong order.
    - **Selection Sort**: Finds the minimum element in the array and swaps it with the first unsorted element.
    - **Insertion Sort**: Builds the sorted array one element at a time by comparing and inserting elements in their correct position.
    - **Merge Sort**: A divide-and-conquer algorithm that splits the array in half, sorts each half, and then merges them.
    - **Quick Sort**: A divide-and-conquer algorithm that selects a pivot element and partitions the array around the pivot.
    - **Heap Sort**: Converts the array into a heap and repeatedly extracts the maximum (or minimum) element.
    - **Radix Sort**: Sorts numbers digit by digit starting from the least significant digit.
  + **Efficiency**:
    - Time complexities of sorting algorithms:
      * **Bubble Sort, Selection Sort, Insertion Sort**: O(n²)
      * **Merge Sort, Quick Sort**: O(n log n)
      * **Heap Sort**: O(n log n)
      * **Radix Sort**: O(d \* (n + k)), where d is the number of digits, and k is the base of the number system.
* **Introduction to Searching Algorithms**:
  + Searching algorithms are used to find the location of a specific element within a data structure.
  + Common searching algorithms include:
    - **Linear Search**: Sequentially checks each element of the list until the desired element is found or the list is exhausted.
    - **Binary Search**: Finds the middle element and eliminates half of the remaining elements based on whether the target is greater or less than the middle element. It requires a sorted array.
  + **Efficiency**:
    - **Linear Search**: O(n)
    - **Binary Search**: O(log n)

**Challenges Faced**

1. **Understanding Time and Space Complexity**:
   * **Problem**: It was challenging to understand how different algorithms perform in terms of time and space complexity, especially when comparing them.
   * **Solution**: I reviewed examples and practiced calculating Big-O notations for various algorithms, which clarified how to determine efficiency and scalability.
2. **Quick Sort Partitioning**:
   * **Problem**: Implementing the partition step in Quick Sort was tricky, especially in terms of correctly placing the pivot element.
   * **Solution**: I visualized the partitioning process and practiced several times, using both **Lomuto** and **Hoare** partition schemes, which helped solidify my understanding.

**Leet Code Problem:**



**Challenges and how I overcome it:**

* **Extracting the correct position from the word:**

Challenge faced: Initially, it can be confusing to extract the position of each word since the number is appended at the end of the word. Misinterpreting this or forgetting to remove the number can lead to incorrect sorting.

Solution: Using Character.getNumericValue() to extract the numeric value of the last character (which represents the position) solved this issue. The correct position is then calculated by subtracting 1 (since positions are 1-indexed but arrays are 0-indexed). Finally, the number is removed by using substring().

* **Handling Uppercase and Lowercase Letters:**

Challenge faced: The input contains both lowercase and uppercase letters, but the problem does not require any conversion of case. There might be confusion around whether to ignore case during sorting.

Solution: Since the sorting is based only on the numeric value appended to each word, the case of the letters doesn’t need to be considered. Just removing the numbers and placing words in their correct positions resolves this.

* **Edge Cases:**

Challenge faced: Handling edge cases like minimum or maximum length of the string (e.g., only two words) or inputs with different cases (like mixed case or single case words).

Solution: Thoroughly tested with strings of varying lengths and cases, ensuring the algorithm can handle different input sizes and combinations without failing.

* **Dijkstra’s Algorithm (Shortest Path Algorithm)**

**Learnings:**

* Dijkstra’s algorithm finds the shortest path from a source node to all other nodes in a weighted graph. It works by selecting the smallest unvisited node and updating the shortest paths.

**Code Example (Dijkstra’s Algorithm in Java):**

import java.util.\*;

class Dijkstra {

private static final int INF = Integer.MAX\_VALUE;

public static void dijkstra(int[][] graph, int src) {

int n = graph.length;

int[] dist = new int[n];

boolean[] visited = new boolean[n];

Arrays.fill(dist, INF);

dist[src] = 0;

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

int u = minDistance(dist, visited);

visited[u] = true;

for (int v = 0; v < n; v++) {

if (!visited[v] && graph[u][v] != 0 && dist[u] != INF && dist[u] + graph[u][v] < dist[v]) {

dist[v] = dist[u] + graph[u][v];

}

}

}

printSolution(dist);

}

private static int minDistance(int[] dist, boolean[] visited) {

int min = INF, minIndex = -1;

for (int v = 0; v < dist.length; v++) {

if (!visited[v] && dist[v] <= min) {

min = dist[v];

minIndex = v;

}

}

return minIndex;

}

private static void printSolution(int[] dist) {

System.out.println("Vertex \t Distance from Source");

for (int i = 0; i < dist.length; i++) {

System.out.println(i + " \t\t " + dist[i]);

}

}

public static void main(String[] args) {

int[][] graph = {

{0, 10, 20, 0, 0},

{10, 0, 5, 16, 0},

{20, 5, 0, 20, 1},

{0, 16, 20, 0, 2},

{0, 0, 1, 2, 0}

};

dijkstra(graph, 0); // Source node is 0

}

}

**Challenges Faced:**

* Handling graphs with multiple edges and updating the shortest path dynamically.

**Solution:**

* Used a priority queue (not shown here) for a more efficient approach to extract the minimum distance node.
* **Bellman-Ford Algorithm (Shortest Path Algorithm)**

**Learnings:**

* Bellman-Ford algorithm calculates the shortest paths from a source vertex to all vertices, even with negative weight edges. It works by relaxing all edges multiple times.

**Code Example (Bellman-Ford Algorithm in Java):**

class BellmanFord {

public void bellmanFord(int[][] graph, int V, int E, int src) {

int[] dist = new int[V];

for (int i = 0; i < V; i++) {

dist[i] = Integer.MAX\_VALUE;

}

dist[src] = 0;

// Relax all edges |V| - 1 times

for (int i = 1; i < V; i++) {

for (int j = 0; j < E; j++) {

int u = graph[j][0];

int v = graph[j][1];

int weight = graph[j][2];

if (dist[u] != Integer.MAX\_VALUE && dist[u] + weight < dist[v]) {

dist[v] = dist[u] + weight;

}

}

}

// Check for negative-weight cycles

for (int j = 0; j < E; j++) {

int u = graph[j][0];

int v = graph[j][1];

int weight = graph[j][2];

if (dist[u] != Integer.MAX\_VALUE && dist[u] + weight < dist[v]) {

System.out.println("Graph contains negative weight cycle");

return;

}

}

printSolution(dist, V);

}

public void printSolution(int[] dist, int V) {

System.out.println("Vertex Distance from Source");

for (int i = 0; i < V; i++) {

System.out.println(i + "\t\t" + dist[i]);

}

}

public static void main(String[] args) {

int V = 5; // Number of vertices

int E = 8; // Number of edges

int[][] graph = {

{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 bf = new BellmanFord();

bf.bellmanFord(graph, V, E, 0); // Source node is 0

}

}

**Challenges Faced:**

* Handling negative weight cycles.

**Solution:**

* Implemented a check for negative-weight cycles after the relaxation step.
* **Graph Traversals (BFS and DFS)**

**Learnings:**

* **Breadth-First Search (BFS)** explores nodes layer by layer using a queue.
* **Depth-First Search (DFS)** explores nodes depth-wise using recursion or a stack.

**Code Example (BFS and DFS in Java):**

import java.util.\*;

class GraphTraversal {

private LinkedList<Integer>[] adjList;

GraphTraversal(int vertices) {

adjList = new LinkedList[vertices];

for (int i = 0; i < vertices; i++) {

adjList[i] = new LinkedList<>();

}

}

public void addEdge(int v, int w) {

adjList[v].add(w);

}

// BFS

public void bfs(int start) {

boolean[] visited = new boolean[adjList.length];

Queue<Integer> queue = new LinkedList<>();

visited[start] = true;

queue.add(start);

while (!queue.isEmpty()) {

int vertex = queue.poll();

System.out.print(vertex + " ");

for (int neighbor : adjList[vertex]) {

if (!visited[neighbor]) {

visited[neighbor] = true;

queue.add(neighbor);

}

}

}

}

// DFS

public void dfs(int start) {

boolean[] visited = new boolean[adjList.length];

dfsUtil(start, visited);

}

private void dfsUtil(int vertex, boolean[] visited) {

visited[vertex] = true;

System.out.print(vertex + " ");

for (int neighbor : adjList[vertex]) {

if (!visited[neighbor]) {

dfsUtil(neighbor, visited);

}

}

}

public static void main(String[] args) {

GraphTraversal graph = new GraphTraversal(6);

graph.addEdge(0, 1);

graph.addEdge(0, 2);

graph.addEdge(1, 3);

graph.addEdge(2, 4);

graph.addEdge(3, 5);

graph.addEdge(4, 5);

System.out.println("BFS starting from node 0:");

graph.bfs(0);

System.out.println("\nDFS starting from node 0:");

graph.dfs(0);

}

}

* **Divide and Conquer**

**Learnings:**

* Divide and conquer is a strategy where problems are divided into smaller subproblems, solved recursively, and combined for the final solution. Examples include Merge Sort and Quick Sort.

**Code Example (Merge Sort in Java):**

import java.util.Arrays;

public class MergeSort {

public static void mergeSort(int[] arr) {

if (arr.length < 2) {

return;

}

int mid = arr.length / 2;

int[] left = Arrays.copyOfRange(arr, 0, mid);

int[] right = Arrays.copyOfRange(arr, mid, arr.length);

mergeSort(left);

mergeSort(right);

merge(arr, left, right);

}

private static void merge(int[] arr, int[] left, int[] right) {

int i = 0, j = 0, k = 0;

while (i < left.length && j < right.length) {

if (left[i] <= right[j]) {

arr[k++] = left[i++];

} else {

arr[k++] = right[j++];

}

}

while (i < left.length) {

arr[k++] = left[i++];

}

while (j < right.length) {

arr[k++] = right[j++];

}

}

public static void main(String[] args) {

int[] arr = {12, 11, 13, 5, 6, 7};

mergeSort(arr);

System.out.println(Arrays.toString(arr)); // Output: [5, 6, 7, 11, 12, 13]

}

}

* **Fractional Knapsack Problem**

**Learnings:**

* The **Fractional Knapsack** problem allows the division of items, unlike the standard Knapsack problem, to maximize value. This problem is solved using a greedy approach.

**Fractional Knapsack Problem in Java:**

import java.util.Arrays;c

import java.util.Comparator;

class Item {

int weight;

int value;

public Item(int weight, int value) {

this.weight = weight;

this.value = value;

}

}

public class FractionalKnapsack {

// Method to get the maximum value in the knapsack

public static double getMaxValue(int capacity, Item[] items) {

// Sort items by value-to-weight ratio in decreasing order

Arrays.sort(items, new Comparator<Item>() {

@Override

public int compare(Item item1, Item item2) {

double r1 = (double) item1.value / item1.weight;

double r2 = (double) item2.value / item2.weight;

return Double.compare(r2, r1);

}

});

double totalValue = 0.0;

for (Item item : items) {

int currentWeight = item.weight;

int currentValue = item.value;

if (capacity - currentWeight >= 0) {

// If the item can fully fit in the knapsack, take it

capacity -= currentWeight;

totalValue += currentValue;

} else {

// Take the fraction of the item that fits in the remaining capacity

double fraction = ((double) capacity / (double) currentWeight);

totalValue += currentValue \* fraction;

break; // Knapsack is full

}

}

return totalValue;

}

public static void main(String[] args) {

// Example items (weight, value)

Item[] items = {

new Item(10, 60), // 60/10 = 6.0 value-to-weight ratio

new Item(20, 100), // 100/20 = 5.0 value-to-weight ratio

new Item(30, 120) // 120/30 = 4.0 value-to-weight ratio

};

int capacity = 50; // Knapsack capacity

double maxValue = getMaxValue(capacity, items);

System.out.println("Maximum value in Knapsack = " + maxValue);

}

}