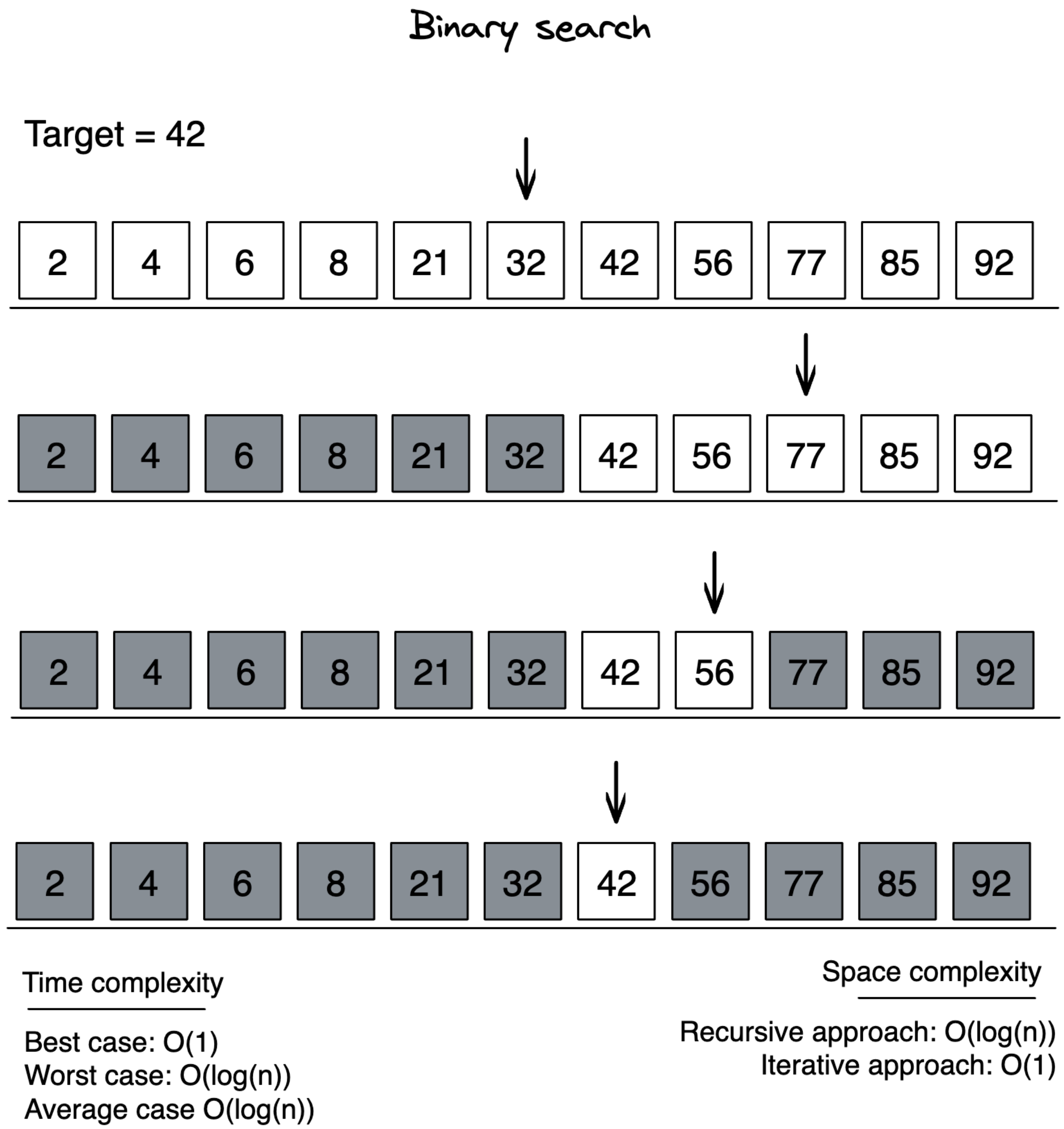
Algorithms Handbook

Vladimir Bolshakov

Contents

Binary search

Binary search



Steps:

- Step 1 - Read the search element from the user.
- Step 2 - Find the middle element in the sorted list.
- Step 3 - Compare the search element with the middle element in the sorted list.
- Step 4 - If both are matched, then display "Given element is found!!!" and terminate the function.

- Step 5 - If both are not matched, then check whether the search element is smaller or larger than the middle element.
- Step 6 - If the search element is smaller than middle element, repeat steps 2, 3, 4 and 5 for the left sublist of the middle element.
- Step 7 - If the search element is larger than middle element, repeat steps 2, 3, 4 and 5 for the right sublist of the middle element.
- Step 8 - Repeat the same process until we find the search element in the list or until sublist contains only one element.
- Step 9 - If that element also doesn't match with the search element, then returns -1;

Time Complexity:

- Worst case: O(log n)
- Average case: O(log n)
- Best case: O(1)

```
function binarySearch(nums: number[], target: number): number {
  let left: number = 0;
  let right: number = nums.length - 1;

  while (left <= right) {
    const mid: number = Math.floor((left + right) / 2);

    if (nums[mid] === target) return mid;
    if (target < nums[mid]) right = mid - 1;
    else left = mid + 1;
  }

  return -1;
}
```

```
class Solution {
    private static int binarySearch(int[] array, int target) {

        int low = 0;
        int high = array.length - 1;

        while(low <= high) {
            int middle = low + (high - low) / 2;
            int value = array[middle];

            if(value < target) {
                low = middle + 1;
            } else if(value > target) {
                high = middle - 1;

            } else {
                return middle;
            }
        }
```

```
        return -1;
    }
}
```

```
def binary_search(list, item):
    low = 0
    high = len(list) - 1
    while low <= high:
        mid = (low+high)/2
        guess = list[mid]
        if guess == item:
            return mid
        if guess > item:
            high = mid - 1
        else:
            low = mid +1
    return None

my_list = [1, 3, 5, 7, 9]

res = binary_search(my_list, 3)

print(my_list[res])
```

Interval search

Interval search

```
type Interval = [number, number];

function intervalSearch(intervals: Interval[], queryInterval: Interval): number[] {
  const result: number[] = [];

  for (let i = 0; i < intervals.length; i++) {
    const [start, end] = intervals[i];
    const [queryStart, queryEnd] = queryInterval;

    if (start <= queryEnd && end >= queryStart) {
      result.push(i);
    }
  }

  return result;
}
```

Linear search

Linear search

```
function linearSearch(arr: number[], target: number): number {
  for (let i = 0; i < arr.length; i++) {
    if (arr[i] === target) {
      return i;
    }
  }

  return -1;
}
```

Ternary search

Ternary search

```
function ternarySearch(func: (x: number) => number, left: number, right: number, epsilon: number): numbe
  while (right - left > epsilon) {
    const mid1 = left + (right - left) / 3;
    const mid2 = right - (right - left) / 3;

    const value1 = func(mid1);
    const value2 = func(mid2);

    if (value1 < value2) {
      left = mid1;
    } else {
      right = mid2;
    }
  }

  return (left + right) / 2;
}
```

Bubble sort

Bubble sort

```
function bubbleSort(array: number[] | string[]) {
    for (let i = 0; i < array.length; i++) {
        for (let j = 0; j < array.length - 1 - i; j++) {
            if (array[j] > array[j + 1]) {
```

```
                [array[j], array[j + 1]] = [array[j + 1], array[j]];
            }
        }
    }
    return array;
}

console.log(bubbleSort([2,5,2,6,7,2,22,5,7,9,0,2,3]))
```

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

Insertion sort

Insertion sort

TypeScript

```
function insertionSort(array: number[] | string[]) {
    for (let i = 1; i < array.length; i++) {
        let curr = array[i];
        let j = i - 1;
        for (j; j >= 0 && array[j] > curr; j--) {
            array[j + 1] = array[j];
        }
        array[j + 1] = curr;
    }
    return array;
}

console.log(insertionSort([1, 4, 2, 8, 345, 123, 43, 32, 5643, 63, 123, 43, 2, 55, 1, 234, 92]));
```

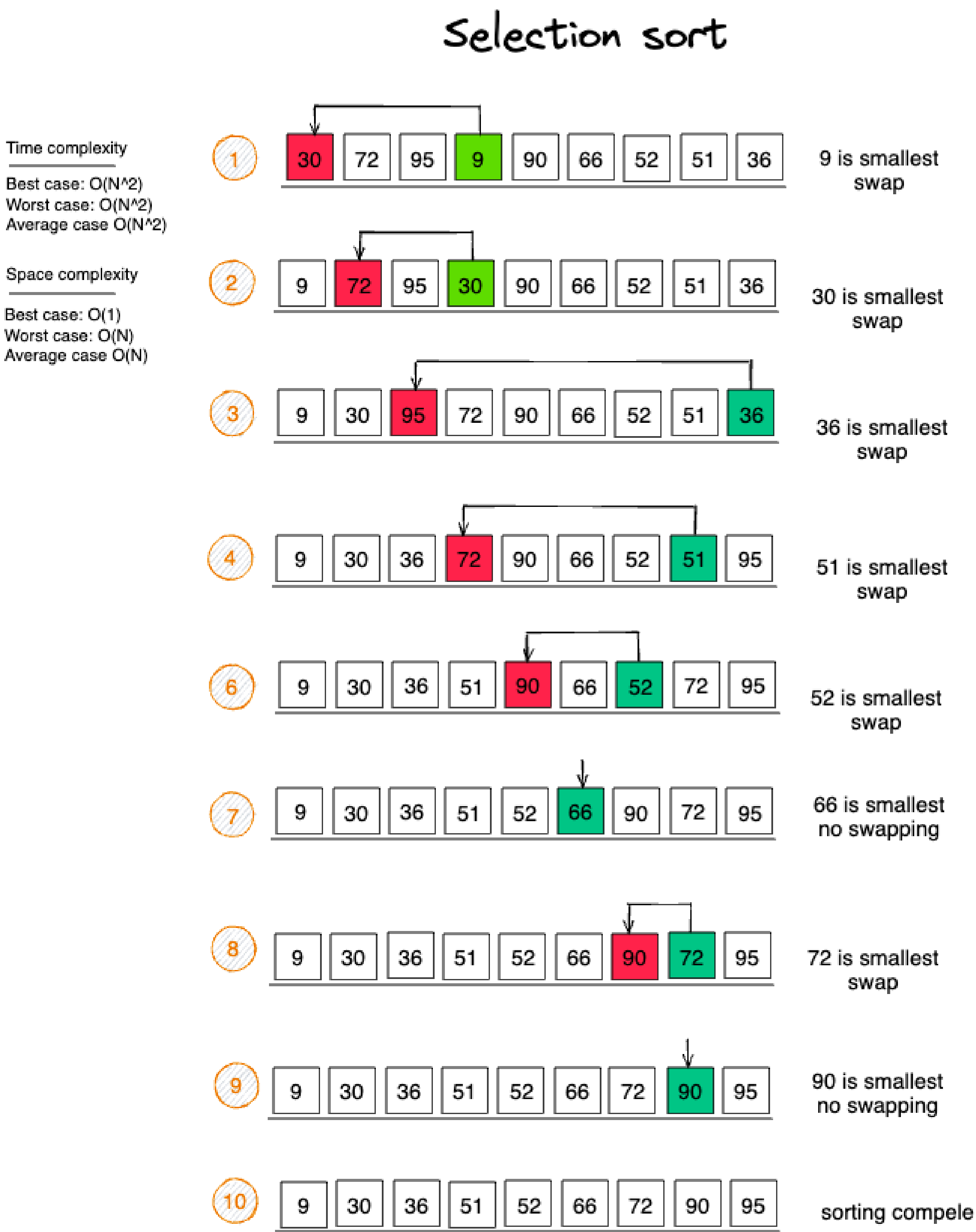
Java

```
    class Solution {
        void insertionSort (int[] arr) {
            int n = arr.length;
            for(int i = 1; i < n; i++) {
```

```
            int current = arr[i];
            int position = i - 1;
            while(position >= 0 && arr[position] > current) {
                arr[position + 1] = arr[position];
                position--;
            }
            arr[position + 1] = current;
        }
    }
}
```

Selection sort

Selection sort



```
function selectionSort(array: any[]) {
  for (let i = 0; i < array.length - 1; i++) {
    let min = i;
    for (let j = i + 1; j < array.length; j++) {
      if (array[min] > array[j]) min = j;
    }
    [array[i], array[min]] =[array[min], array[i]]
  }
  return array;
}

console.log(selectionSort([1, 4, 2, 8, 345, 123, 43, 32, 5643, 63, 123, 43, 2, 55, 1, 234, 92]));
```

```
    public static void selectionSort(int[] array) {
        for(int i = 0; i < array.length - 1; i++) {
            int min = i;
            for(int j = i + 1; j < array.length; j++) {
                if(array[min] > array[j]) {
                    min = j;
                }
            }
            int temp = array[i];
            array[i] = array[min];
            array[min] = temp;
        }
    }
```

```
print('This is selection sort')

def find_smallest(arr):
    smallest = arr[0]
    smallest_index = 0
    for i in range(1, len(arr)):
        if arr[i] < smallest:
            smallest = arr[i]
            smallest_index = i
    return smallest_index

def selection_sort(arr):
    newArr = []
    for i in range(len(arr)):
        smallest = find_smallest(arr)
        newArr.append(arr.pop(smallest))
    return newArr

print(selection_sort([5,4,6,2,1,123, 2, 3,1,23 ,1,1,]))
```

Quick sort

Quicksort

```
class Solution {

    int makePartition(int [] arr, int low, int high) {
        int pivot = arr[high];
        int currentIndex = low - 1;
        for(int i = low; i < high; i++) {
            if(arr[i] < pivot) {
                currentIndex++;
                int temp = arr[i];
                arr[i] = arr[currentIndex];
                arr[currentIndex] = temp;
            }

        }
        int temp = arr[high];
        arr[high] = arr[currentIndex + 1];
        arr[currentIndex + 1] = temp;
        return currentIndex + 1;
    }

    void quicksort(int[] arr, int low, int high) {
        if(low < high) {
            int pivot = makePartition(arr, low, high);
            quicksort(arr, low, pivot - 1);
            quicksort(arr, pivot + 1, high);
        }
    }

    void quickSort (int[] arr) {
        int n = arr.length;
        quicksort(arr, 0, n - 1);
    }
}
```

```
def quicksort(arr):
    if len(arr) < 2:
        return arr
    else:
        pivot = arr[len(arr)/2]
        less = [i for i in arr[1:] if i <= pivot]
        greater = [i for i in arr[1:] if i > pivot]
        return quicksort(less) + [pivot] + quicksort(greater)

print(quicksort([10,2,3,1,5,4]))
```

```
class Solution {
    static void swap(int[] array, int i, int j) {
```

```
        int temp = array[i];
        array[i] = array[j];
        array[j] = temp;
    }

    private static void quickSort(int[] array, int start, int end) {
        if(end <= start) return; // base case

        int pivot = partition(array, start, end);

        quickSort(array, start, pivot -1);
        quickSort(array, pivot + 1, end);
    }

    private static int partition(int[] array, int start, int end) {
        int pivot = array[end];

        int i = start - 1;

        for(int j = start; j <= end -1; j++) {
            if(array[j] < pivot) {
                i++;
                swap(array, i, j);
            }
        }
        i++;
        swap(array, i, end);

        return i;
    }
}
```

```
function quicksort(arr: number[]): number[] {
  if (arr.length < 2) {
    return arr;
  } else {
    const pivot = arr[Math.floor(arr.length / 2)];
    const less = arr.slice(1).filter((i) => i <= pivot);
    const greater = arr.slice(1).filter((i) => i > pivot);
    return [...quicksort(less), pivot, ...quicksort(greater)];
  }
}
```

- Go back

Merge sort

Merge sort

Java

```
class Solution {

    void merge(int[] arr, int low, int mid, int high) {
        int subArr1Size = mid - low + 1;
        int subArr2Size = high - mid;

        int [] subArr1 = new int[subArr1Size];
        int [] subArr2 = new int[subArr2Size];

        for (int i = 0; i < subArr1Size; i++) {
           subArr1[i] = arr[low + i];
           }
           for (int i = 0; i < subArr2Size; i++) {
           subArr2[i] = arr[mid + 1 + i];
        }
        int i = 0, j = 0, k = low;

        while(i < subArr1Size && j < subArr2Size) {
            if(subArr1[i] <= subArr2[j]) {
                arr[k] = subArr1[i];
                i++;
            } else {
                arr[k] = subArr2[j];
                j++;
            }
            k++;
        }
        while(i < subArr1Size) {
            arr[k++] = subArr1[i++];
        }
        while (j < subArr2Size) {
           arr[k++] = subArr2[j++];
       }
    }

    void mergesort(int[] arr, int low, int high){
        if(high > low) {
            int mid = (high + low) / 2;
            mergesort(arr, low, mid);
            mergesort(arr, mid + 1, high);
            merge(arr, low, mid, high);
        }
    }

    void mergeSort (int[] arr) {
        int n = arr.length;
```

```
        mergesort(arr, 0, n - 1);
    }
}
```

```
function mergeSort(arr: number[]): number[] {
  if (arr.length <= 1) {
    return arr;
  }

  const middle = Math.floor(arr.length / 2);
  const left = arr.slice(0, middle);
  const right = arr.slice(middle);

  return merge(mergeSort(left), mergeSort(right));
}

function merge(left: number[], right: number[]): number[] {
  let result: number[] = [];
  let leftIndex = 0;
  let rightIndex = 0;

  while (leftIndex < left.length && rightIndex < right.length) {
    if (left[leftIndex] < right[rightIndex]) {
      result.push(left[leftIndex]);
      leftIndex++;
    } else {
      result.push(right[rightIndex]);
      rightIndex++;
    }
  }

  return result.concat(left.slice(leftIndex)).concat(right.slice(rightIndex));
}
```

Interpolation search

Interpolation search

```
class Solution {

    private static int interpolationSearch(int[] array, int value) {
        int low = 0;
        int high = array.length - 1;

        while(value >=array[low] && value <= array[high] && low <= high) {
            int probe = low + (high - low) * (value - array[low]) / (array[high] - array[low]);
            if(array[probe] == value) {
                return probe;
            } else if(array[probe] > value) {
                low = probe + 1;
```

```
            } else {
                high = probe -1;
            }

        }

        return -1;
    }

}
```

```
function interpolationSearch(array: number[], value: number): number {
  let low = 0;
  let high = array.length - 1;

  while (value >= array[low] && value <= array[high] && low <= high) {
    const probe = low + ((high - low) * (value - array[low])) / (array[high] - array[low]);
    const roundedProbe = Math.floor(probe);

    if (array[roundedProbe] === value) {
      return roundedProbe;
    } else if (array[roundedProbe] < value) {
      low = roundedProbe + 1;
    } else {
      high = roundedProbe - 1;
    }
  }

  return -1;
}
```

Diffie hellman algorithm

Diffie hellman algorithm

```
function power(a: any, b: any, p: any) {
    if(b === 1) {
        return 1
    } else {
        return Math.pow(a,b) % p
    }
}


function DiffieHellman() {

    let P, G, x, a, y, b, ka, kb;

    P = 23
```

```
    console.log("The value of P :", P);

    G = 9;

    console.log("The value of G :", G);

    a = 4;

    console.log("The private key a for Alice : ", a);

    x = power(G,a,P);

    b = 3;

    console.log("The private key a for Bob : ", b);

    y = power(G,b,P);


    ka = power(y, a, P);
    kb = power(x, b, P);

    console.log("Secret key for the Alice is : ", ka);
    console.log("Secret key for the Bob is : ", kb);
}


DiffieHellman()
```

Breadth-first search

Breadth-first search

```
class Graph {
  private adjacencyList: Map<string, string[]>;

  constructor() {
    this.adjacencyList = new Map();
  }

  addVertex(vertex: string) {
    if (!this.adjacencyList.has(vertex)) {
      this.adjacencyList.set(vertex, []);
    }
  }

  addEdge(vertex1: string, vertex2: string) {
    this.adjacencyList.get(vertex1)?.push(vertex2);
    this.adjacencyList.get(vertex2)?.push(vertex1);
  }
```

```
  bfs(startingVertex: string) {
    const visited: Set<string> = new Set();
    const queue: string[] = [];

    visited.add(startingVertex);
    queue.push(startingVertex);

    while (queue.length > 0) {
      const currentVertex = queue.shift()!;
      console.log(currentVertex);

      const neighbors = this.adjacencyList.get(currentVertex) || [];

      for (const neighbor of neighbors) {
        if (!visited.has(neighbor)) {
          visited.add(neighbor);
          queue.push(neighbor);
        }
      }
    }
  }
}

// Example usage:
const graph = new Graph();
graph.addVertex("A");
graph.addVertex("B");
graph.addVertex("C");
graph.addVertex("D");
graph.addEdge("A", "B");
graph.addEdge("A", "C");
graph.addEdge("B", "D");

graph.bfs("A");
```

Depth-first search

Depth-first search

```
class Graph {
  private adjacencyList: Map<string, string[]>;

  constructor() {
    this.adjacencyList = new Map();
  }

  addVertex(vertex: string) {
    if (!this.adjacencyList.has(vertex)) {
      this.adjacencyList.set(vertex, []);
```

```
    }
  }

  addEdge(vertex1: string, vertex2: string) {
    this.adjacencyList.get(vertex1)?.push(vertex2);
    this.adjacencyList.get(vertex2)?.push(vertex1);
  }

  dfs(startingVertex: string) {
    const visited: Set<string> = new Set();

    const dfsHelper = (vertex: string) => {
      console.log(vertex);
      visited.add(vertex);

      const neighbors = this.adjacencyList.get(vertex) || [];

      for (const neighbor of neighbors) {
        if (!visited.has(neighbor)) {
          dfsHelper(neighbor);
        }
      }
    };

    dfsHelper(startingVertex);
  }
}

// Example usage:
const graph = new Graph();
graph.addVertex("A");
graph.addVertex("B");
graph.addVertex("C");
graph.addVertex("D");
graph.addEdge("A", "B");
graph.addEdge("A", "C");
graph.addEdge("B", "D");

graph.dfs("A");
```

Dijkstra's algorithm

Dijkstra's algorithm

```
class Graph {
  private adjacencyList: Map<string, Map<string, number>>;

  constructor() {
    this.adjacencyList = new Map();
  }
```

```
addVertex(vertex: string) {
  if (!this.adjacencyList.has(vertex)) {
    this.adjacencyList.set(vertex, new Map());
  }
}

addEdge(vertex1: string, vertex2: string, weight: number) {
  this.adjacencyList.get(vertex1)?.set(vertex2, weight);
  this.adjacencyList.get(vertex2)?.set(vertex1, weight);
}

dijkstra(startingVertex: string) {
  const distances: Map<string, number> = new Map();
  const previous: Map<string, string | null> = new Map();
  const priorityQueue = new PriorityQueue();

  for (const vertex of this.adjacencyList.keys()) {
    distances.set(vertex, vertex === startingVertex ? 0 : Infinity);
    previous.set(vertex, null);
    priorityQueue.enqueue(vertex, distances.get(vertex)!);
  }

  while (!priorityQueue.isEmpty()) {
    const currentVertex = priorityQueue.dequeue()!;
    const neighbors = this.adjacencyList.get(currentVertex);

    if (neighbors) {
      for (const neighbor of neighbors.keys()) {
        const distance = distances.get(currentVertex)! + neighbors.get(neighbor)!;

        if (distance < distances.get(neighbor)!) {
          distances.set(neighbor, distance);
          previous.set(neighbor, currentVertex);
          priorityQueue.enqueue(neighbor, distance);
        }
      }
    }
  }

  return { distances, previous };
}

shortestPath(startingVertex: string, targetVertex: string) {
  const { distances, previous } = this.dijkstra(startingVertex);

  const path: string[] = [];
  let currentVertex = targetVertex;

  while (currentVertex !== null) {
    path.unshift(currentVertex);
    currentVertex = previous.get(currentVertex)!;
  }
```

```
    return { path, distance: distances.get(targetVertex) };
  }
}

class PriorityQueue {
  private items: [string, number][] = [];

  enqueue(element: string, priority: number) {
    this.items.push([element, priority]);
    this.sort();
  }

  dequeue() {
    return this.items.shift();
  }

  isEmpty() {
    return this.items.length === 0;
  }

  private sort() {
    this.items.sort((a, b) => a[1] - b[1]);
  }
}

// Example usage:
const graph = new Graph();
graph.addVertex("A");
graph.addVertex("B");
graph.addVertex("C");
graph.addVertex("D");
graph.addEdge("A", "B", 1);
graph.addEdge("A", "C", 4);
graph.addEdge("B", "C", 2);
graph.addEdge("B", "D", 5);
graph.addEdge("C", "D", 1);

const { path, distance } = graph.shortestPath("A", "D");
console.log("Shortest Path:", path); // Output: Shortest Path: [ 'A', 'B', 'C', 'D' ]
console.log("Distance:", distance); // Output: Distance: 4
```

Floyd-Warshall algorithm

Floyd-Warshall algorithm

```
class Graph {
  private adjacencyMatrix: number[][];

  constructor(numVertices: number) {
    this.adjacencyMatrix = Array.from({ length: numVertices }, () =>
```

```
      Array(numVertices).fill(Infinity)
    );

    // Set diagonal elements to 0
    for (let i = 0; i < numVertices; i++) {
      this.adjacencyMatrix[i][i] = 0;
    }
  }

  addEdge(source: number, destination: number, weight: number) {
    this.adjacencyMatrix[source][destination] = weight;
  }

  floydWarshall() {
    const numVertices = this.adjacencyMatrix.length;

    for (let k = 0; k < numVertices; k++) {
      for (let i = 0; i < numVertices; i++) {
        for (let j = 0; j < numVertices; j++) {
          if (
            this.adjacencyMatrix[i][k] + this.adjacencyMatrix[k][j] <
            this.adjacencyMatrix[i][j]
          ) {
            this.adjacencyMatrix[i][j] =
              this.adjacencyMatrix[i][k] + this.adjacencyMatrix[k][j];
          }
        }
      }
    }

    return this.adjacencyMatrix;
  }
}

// Example usage:
const graph = new Graph(4);

graph.addEdge(0, 1, 3);
graph.addEdge(0, 2, 6);
graph.addEdge(1, 2, 1);
graph.addEdge(1, 3, 4);
graph.addEdge(2, 3, 2);

const result = graph.floydWarshall();

console.log("Shortest Path Matrix:");
for (const row of result) {
  console.log(row);
}
```

Ford Fulkerson algorithm

Ford Fulkerson algorithm

```
class FordFulkerson {
  private graph: number[][];
  private numVertices: number;

  constructor(graph: number[][]) {
    this.graph = graph;
    this.numVertices = graph.length;
  }

  fordFulkerson(source: number, sink: number): number {
    let maxFlow = 0;

    // Create a residual graph and initialize it with the original capacities.
    const residualGraph = this.graph.map((row) => [...row]);

    while (true) {
      const path = this.bfs(source, sink, residualGraph);
      if (!path) {
        break; // No augmenting path found, terminate the algorithm
      }

      // Find the minimum capacity along the augmenting path
      let minCapacity = Number.POSITIVE_INFINITY;
      for (let i = 0; i < path.length - 1; i++) {
        const u = path[i];
        const v = path[i + 1];
        minCapacity = Math.min(minCapacity, residualGraph[u][v]);
      }

      // Update residual capacities and reverse edges along the path
      for (let i = 0; i < path.length - 1; i++) {
        const u = path[i];
        const v = path[i + 1];
        residualGraph[u][v] -= minCapacity;
        residualGraph[v][u] += minCapacity;
      }

      // Add the flow of the augmenting path to the total flow
      maxFlow += minCapacity;
    }

    return maxFlow;
  }

  bfs(source: number, sink: number, graph: number[][]): number[] | null {
    const visited: boolean[] = new Array(this.numVertices).fill(false);
    const queue: number[] = [source];
    const parent: number[] = new Array(this.numVertices).fill(-1);
```

```
    while (queue.length > 0) {
      const u = queue.shift()!;

      for (let v = 0; v < this.numVertices; v++) {
        if (!visited[v] && graph[u][v] > 0) {
          queue.push(v);
          parent[v] = u;
          visited[v] = true;
        }
      }
    }

    if (!visited[sink]) {
      return null; // No augmenting path found
    }

    const path: number[] = [];
    for (let v = sink; v !== source; v = parent[v]) {
      path.unshift(v);
    }
    path.unshift(source);

    return path;
  }
}

// Example usage:
const graph = [
  [0, 16, 13, 0, 0, 0],
  [0, 0, 10, 12, 0, 0],
  [0, 4, 0, 0, 14, 0],
  [0, 0, 9, 0, 0, 20],
  [0, 0, 0, 7, 0, 4],
  [0, 0, 0, 0, 0, 0],
];

const fordFulkerson = new FordFulkerson(graph);
const maxFlow = fordFulkerson.fordFulkerson(0, 5);
console.log("Maximum Flow:", maxFlow);
```