Holy Theory

Vladimir Bolshakov

Contents

\newpage

Binary search

Binary Search is defined as a searching algorithm used in a sorted array by repeatedly dividing the search interval in half. The idea of binary search is to use the information that the array is sorted,.

How it works:

Step 1: Start with a sorted array and a target value.

Step 2: Set the left and right pointers to the start and end of the array.

Step 3: Calculate the middle index and compare the middle element with the target.

Step 4: Adjust the search interval based on the comparison.

Step 5: Repeat until the target is found or the search interval is empty.

Key Characteristics:

Divide and Conquer Approach: Binary search follows the principle of "divide and conquer." It repeatedly separate the search interval in half until the target element is found or the interval is empty.

Efficiency: This algorithm is highly efficient, particularly for large datasets, because it eliminates half of the remaining elements in each step. It has a time complexity of O(log n), where n is the number of elements in the array.

Requirement of Sorted Data: Binary search requires the data to be sorted beforehand. This ensures that the algorithm can reliably make decisions about which half of the array to search in.

Iterative or Recursive Implementation: Binary search can be implemented either iteratively or recursively. Both approaches follow the same logic but differ in their implementation details.

Midpoint Calculation: In each iteration, the algorithm calculates the midpoint of the search interval to determine whether the target element lies in the left or right half of the interval.

Termination Condition: Binary search terminates when the search interval is empty, indicating that the target element is not present in the array, or when the target element is found.

Space Complexity: The algorithm has a space complexity of O(1), meaning it requires constant extra space for storing variables regardless of the size of the input array.

Works on Random Access Data Structures: Binary search is ideal for arrays or other data structures that allow random access to elements, such as arrays or certain types of lists.

Applications:

Searching in Databases: Binary search is widely used in databases for fast retrieval of records based on keys. Databases often use sorted indexes to enable binary search, which significantly reduces the time required to locate specific records.

Searching in Trees and Graphs: The algorithm is employed in various tree-based and graph-based data structures like binary search trees, AVL trees, and red-black trees for efficient searching and retrieval of elements.

Sorting Algorithms: Binary search is utilized in sorting algorithms such as quicksort and mergesort to efficiently partition and merge elements. Although these sorting algorithms primarily rely on divide-and-conquer techniques, binary search plays a crucial role in the process.

Finding Median and Percentile: The algorithm is used to find the median or percentile in a sorted array or list. By repeatedly narrowing down the search range, binary search efficiently locates the desired element, providing quick statistical analysis capabilities.

Searching in Texts: Binary search can be applied in text processing applications for tasks such as searching for keywords or phrases in sorted text files or dictionaries. This is particularly useful in applications like spell checkers or autocomplete features.

Network Routing: The algorithm is utilized in network routing algorithms to efficiently locate routes in routing tables. By organizing routing information in sorted structures, binary search can quickly identify the optimal path for data packets in computer networks.

Approximate Matching and Fuzzy Searching: Binary search can be adapted for approximate matching or fuzzy searching applications where exact matches are not required. By defining appropriate search criteria and tolerances, binary search can efficiently locate approximate matches in sorted datasets.

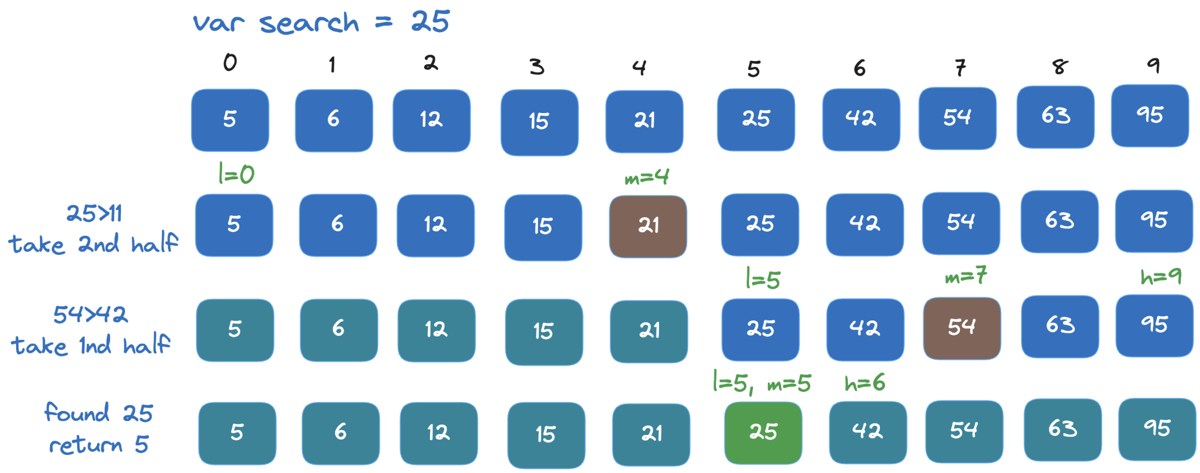
Genetics and Bioinformatics: Binary search is used in various bioinformatics applications for tasks such as searching for specific DNA sequences or analyzing genomic data. Its efficiency makes it valuable for processing large volumes of genetic information.

Game Development: Binary search is employed in various game development scenarios, such as pathfinding algorithms or determining optimal strategies. For example, in certain types of games, binary search can be used to efficiently locate targets or identify the best course of action.

Resource Allocation: Binary search can be applied in resource allocation problems, such as scheduling tasks or managing inventory. By efficiently searching for available resources or optimal scheduling slots, binary search helps optimize resource utilization and allocation.

Time Complexity:

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



Example:

```
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])
```

\newpage

Breadth First search

Breadth-First Search (BFS) is a graph traversal algorithm that systematically explores all the vertices of a graph in breadthward motion, level by level. It starts at a chosen vertex and visits all its neighbors before moving on to other ones. BFS is commonly used to find the shortest path in an unweighted graph and to explore the structure of a graph.

How it works:

Step 1: Initialize queue and begin by selecting a starting vertex and enqueue it into a queue.

Step 2: Dequeue a vertex from the front of the queue and explore all its neighbors. Enqueue any unvisited ones, marking them as visited to avoid duplication.

Step 3: Continue the process level by level, exploring all vertices at the current one before moving on to the next level.

Step 4: Repeat until the queue is empty, ensuring that all reachable vertices are visited.

Key Characteristics:

Queue-Based: BFS utilizes a queue data structure to keep track of the nodes that need to be visited.

Level Order Traversal: BFS systematically explores all nodes at the current level before moving on to the next one. This ensures that nodes are visited in increasing order of their distance from the starting node.

Non-Recursive (typically): While BFS can be implemented recursively, it is typically implemented iteratively using a queue due to better space efficiency and avoidance of potential stack overflow issues, especially for large graphs.

Complete (for finite graphs): The algorithm will visit all nodes reachable from the starting node in a finite graph. However, for infinite or cycled graphs, proper termination conditions need to be implemented to avoid infinite loops.

Optimal for Shortest Path (unweighted graphs): BFS is optimal for finding the shortest path between two nodes in an unweighted graph. This is because BFS guarantees that the first occurrence of a node during traversal will result in the shortest path to that node.

Memory Consumption: The algorithm typically requires more memory compared to Depth First Search (DFS) as it needs to store all the nodes at the current level in the queue. However, in practice, BFS can be more memory-efficient for very deep graphs because it does not suffer from the recursion depth limitations of DFS.

Traversal from Source Node: BFS starts traversal from a source node and explores all its adjacent ones before moving on to the next level. This process continues until all reachable nodes are visited.

Applications:

Shortest Path and Minimum Spanning Tree: BFS can be used to find the shortest path between two nodes in an unweighted graph.

Web Crawling and Search Engine Indexing: BFS is used in web crawlers to systematically traverse the web graph starting from a given seed URL.

Social Network Analysis: BFS is employed to analyze social networks, where nodes represent individuals, and edges represent relationships between them.

Network Broadcasting and Routing: The algorithm can be utilized in network protocols for broadcasting messages or routing packets.

Shortest Path in a Maze: BFS is commonly used to find the shortest path from a starting point to a destination in a maze. Each cell of the maze is considered a node, and BFS explores adjacent cells level by level until the destination is reached.

Puzzle Solving: BFS is employed in puzzle-solving algorithms, such as solving sliding block puzzles (e.g., 15-puzzle), Rubik's Cube, or Sudoku. It helps in finding the shortest sequence of moves or steps to reach the goal state.

Garbage Collection: The algorithm is used in memory management systems for garbage collection. It helps identify and mark reachable objects from the root set, ensuring that only reachable objects are retained in memory.

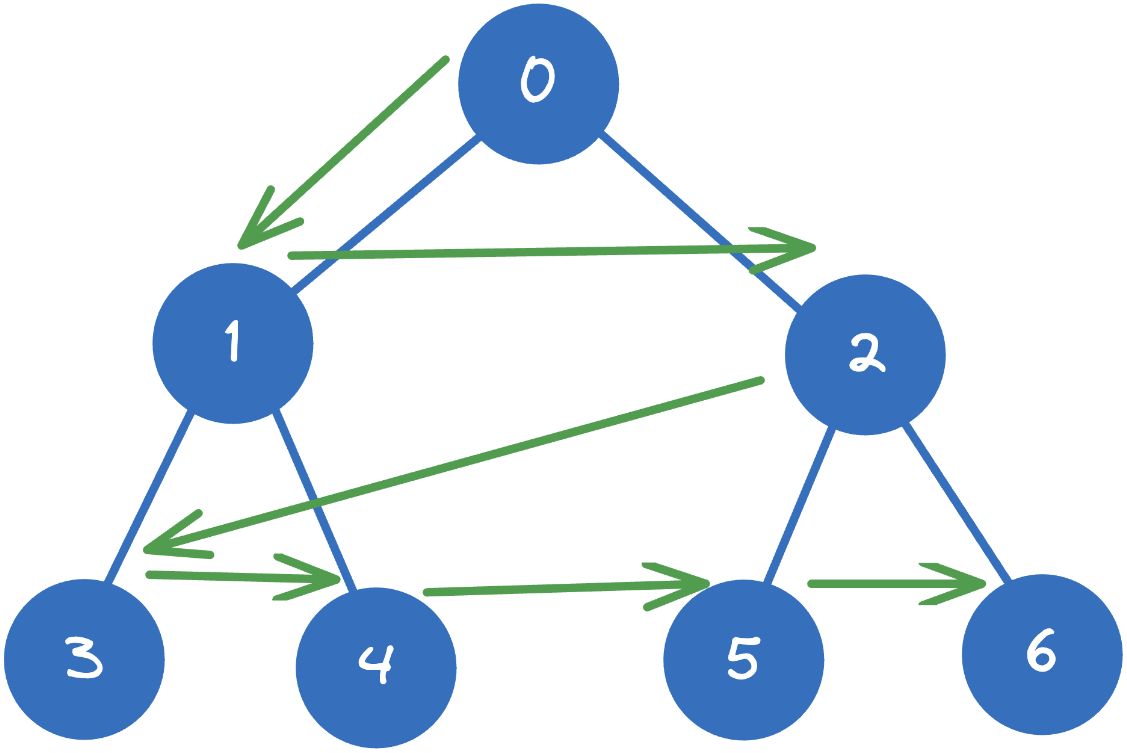
Process Scheduling and Resource Allocation: BFS can be applied in operating systems for process scheduling and resource allocation. It helps in scheduling processes based on their priority levels or allocating resources optimally while considering dependencies.

Routing in Networks and GPS Navigation: In networks, BFS can be used for routing packets based on the hop count, ensuring efficient data transmission. In GPS navigation systems, BFS assists in finding the shortest route between two locations on a map.

Game AI and Pathfinding: BFS is utilized in game development for pathfinding algorithms to navigate characters or units in a game environment. It helps in finding optimal paths while avoiding obstacles or enemies.

Time Complexity:

The time complexity of BFS is (O(V + E)), where (V) is the number of vertices and (E) is the number of edges. The algorithm visits each vertex and edge once.



Example:

```
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");
```

\newpage

Bubble sort

Bubble sorting is one of the simplest sorting algorithms that is not used in practice, but is actively used for training purposes. It works by repeatedly going through the sorted list, comparing each pair of neighboring elements and replacing them if they are in the wrong order. The pass through the list is repeated until no permutations are needed, indicating that the list is sorted.

How it works:

Step 1: The algorithm starts by comparing the first two elements of the array. If the first element is greater than the second, they are swapped. Otherwise, they remain in their positions.

Step 2: This process is then repeated for each pair of neighboring elements throughout the array. After the first iteration, the largest element will "pop up" to the last position.

Step 3: The algorithm then repeats the process for the remaining elements (excluding those already sorted at the end of the array). At each pass, the next largest element is placed in the correct position.

Step 4: The algorithm terminates when a pass through the entire array is made without any swaps, indicating that the array is now sorted.

Key Characteristics:

Basic Algorithm: Bubble sort compares adjacent elements and swaps them if they're in the wrong order, repeating until sorted.

In-place Sorting: Sorts elements by swapping them in place, no extra storage needed.

Stability: Preserves relative order of equal elements, making it stable.

Adaptive: Can perform better if the input is nearly sorted, but still O(nˆ2) in the worst case.

Easy to Implement: Straightforward to understand and implement.

Inefficiency: Not suitable for large datasets due to its quadratic time complexity.

Applications:

Educational Purposes: Bubble sort is commonly used to teach sorting algorithms due to its simplicity.

Small Datasets: Suitable for sorting small datasets where simplicity outweighs efficiency.

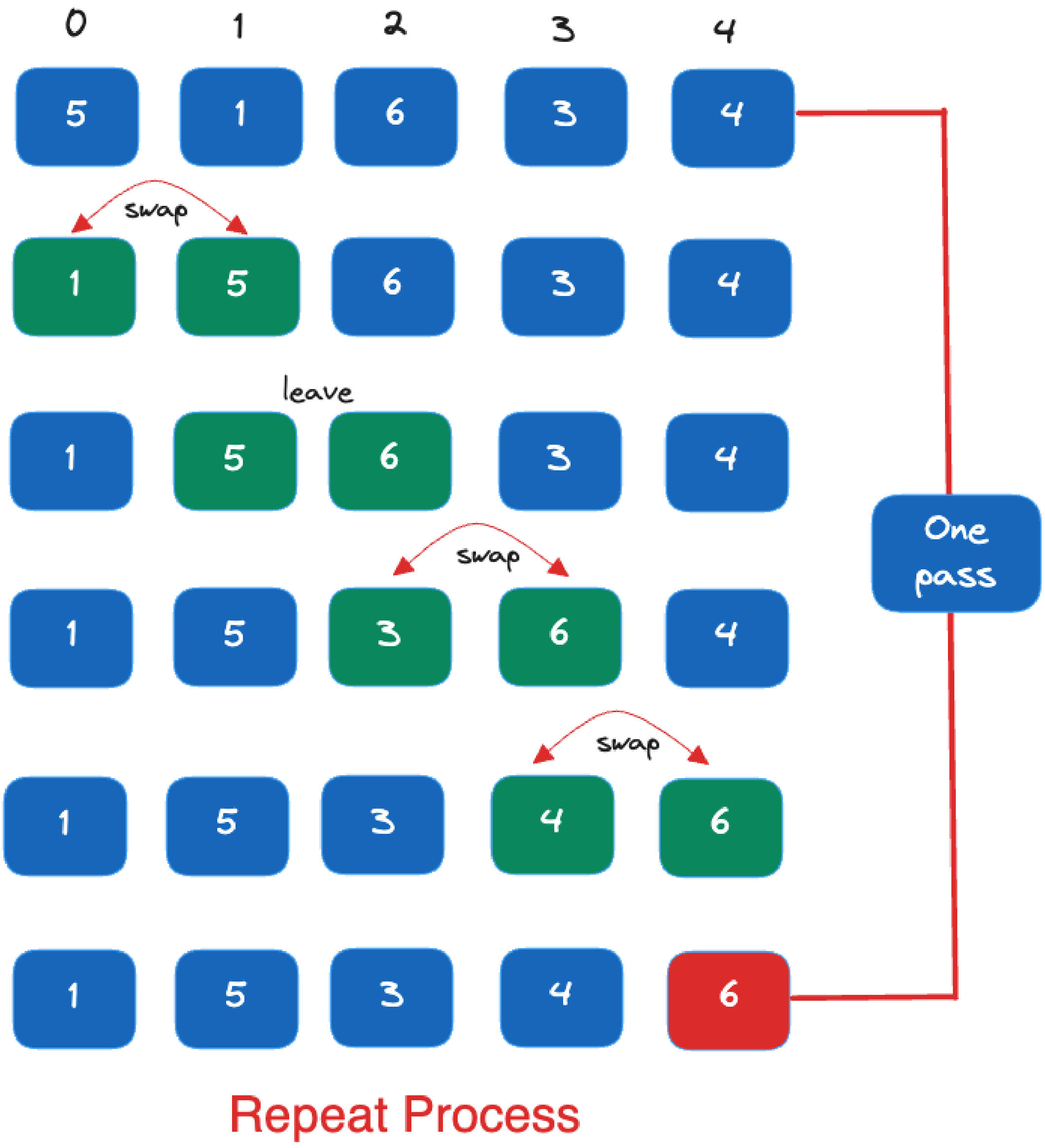
Embedded Systems: Can be used in environments with limited resources for sorting small arrays.

Testing and Debugging: Useful for quick implementation and verification of sorting functionality.

Ad Hoc Sorting: Provides a quick-and-dirty solution for one-time or temporary sorting needs.

Time complexity:

Bubble sort has a time complexity of O(nˆ2) in the worst and average cases, where n is the number of elements in the array. This makes it inefficient for large datasets but useful for educational purposes because of its simplicity.



Example:

```
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;
                }
            }
        }
    }
```

\newpage

Depth-first search

Depth-First Search (DFS) is a graph traversal algorithm that systematically explores the vertices of a graph by going as deep as possible along each branch before backtracking. It starts at a chosen vertex, explores as far as possible along one branch, and then backtracks to explore other ones. DFS is commonly used to detect cycles in a graph, topologically sort vertices, and solve problems related to connected components.

How it works:

Step 1: Begin by selecting a starting vertex and mark it as visited.

Step 2: Move to an unvisited neighbor of the current vertex and repeat the process. If all neighbors are visited, backtrack to the previous vertex.

Step 3: DFS can be implemented using recursion or an explicit stack to keep track of the vertices to be visited.

Step 4: Mark each visited vertex to avoid revisiting and use backtracking to explore other branches.

Step 5: Continue the process until all reachable vertices are visited.

Key Characteristics:

Stack-Based (or Recursive): DFS can be implemented using a stack data structure or by recursion. In both cases, it relies on last-in, first-out (LIFO) ordering to keep track of nodes to be visited and backtracked.

Non-Optimal for Shortest Paths: The algorithm does not guarantee finding the shortest path between two nodes in a graph. It explores as far as possible along each branch before backtracking, which may not necessarily lead to the shortest path.

Memory-Efficient: DFS typically requires less memory compared to BFS because it only needs to store the path from the starting node to the current node. This makes it suitable for large graphs or graphs with limited memory.

Completeness (for finite graphs): DFS will visit all the nodes reachable from the starting node in a finite graph. However, it may not terminate in the presence of cycles in an infinite graph without proper cycle detection.

Recursive Nature: DFS is inherently recursive in its nature. When implemented recursively, the function calls itself for each adjacent node until it reaches a leaf node or a node without unvisited neighbors.

Depth-First Search Forest: DFS produces a depth-first search forest, which is a collection of depth-first trees, one for each connected component of the graph. Each tree is rooted at a distinct vertex.

Backtracking Mechanism: DFS employs a backtracking mechanism to explore all possible paths in the graph. When it reaches a dead end (a node with no unvisited neighbors), it backtracks to the most recent node with unexplored neighbors.

Stack Overflow Risk (for recursive implementation): When implemented recursively, DFS may encounter a stack overflow error if the recursion depth exceeds the system's limit. This risk can be mitigated by using an iterative approach or increasing the stack size.

Applications:

Traversal and Search: Depth first search can be used to traverse and search a graph, visiting all the nodes reachable from a given starting node. This application is useful in scenarios such as finding connected components, identifying cycles, or determining reachability.

Topological Sorting: DFS can be employed to perform topological sorting on directed acyclic graphs (DAGs). Topological sorting is essential in scheduling tasks with dependencies, such as in project management or task scheduling.

Pathfinding: While Depth first sarch does not guarantee finding the shortest path between two nodes, it can be used for pathfinding in scenarios where looking for any path is sufficient. Examples include maze solving, puzzle solving, and navigating game maps.

Cycle Detection: The algorithm is commonly used to detect cycles in a graph. They can be identified when revisiting a node that is already in the current path by maintaining a set of visited nodes and tracking the path taken during traversal.

Strongly Connected Components (SCC): DFS can be applied to find strongly connected components in a directed graph. SCCs are subsets of vertices in a graph where every vertex is reachable from every other vertex within the subset.

Network Analysis: DFS is utilized in network analysis tasks, such as finding bridges and articulation points in a network. Bridges are edges whose removal would disconnect the graph, while articulation points are vertices whose removal would increase the number of connected components.

Solving Puzzles: The algorithm can be used to solve various puzzles, including Sudoku, N-Queens, and word games like Boggle. By exploring all possible configurations or solutions, DFS can efficiently find a valid solution.

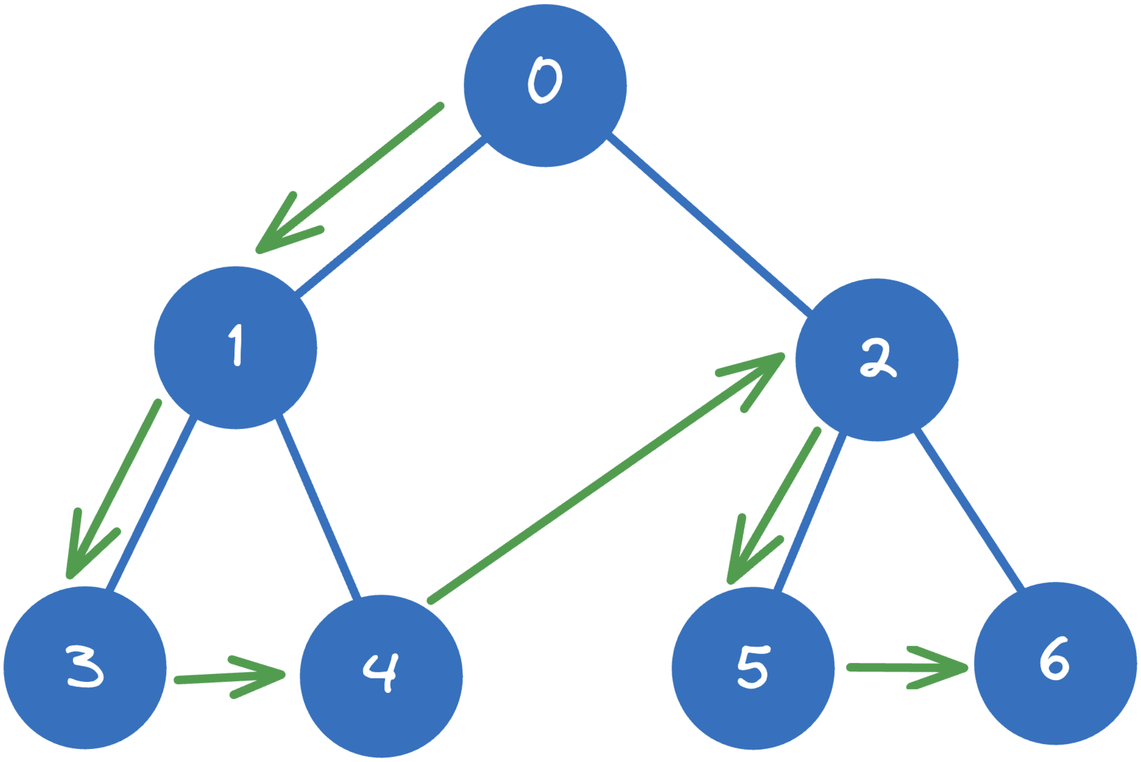
XML Parsing and Document Traversal: DFS can be employed in parsing XML documents and traversing hierarchical data structures. It allows for efficient exploration of the document structure and extraction of relevant information.

Decision Trees and Game Trees: DFS is used in decision trees and game trees to explore possible outcomes and make decisions. It helps in determining optimal strategies or paths in decision-making processes.

Backtracking Algorithms: Many backtracking algorithms, such as N-Queens, Subset Sum, and Sudoku solvers, rely on DFS for exploring the search space and finding solutions by systematically trying different combinations.

Time Complexity:

The time complexity of DFS is (O(V + E)), where (V) is the number of vertices and (E) is the number of edges. The algorithm visits each vertex and edge once. Recursive DFS has a space complexity of (O(V)) due to the call stack, while an explicit stack implementation can have a space complexity of (O(E + V)).



Example:

```
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");
```

\newpage

Dijkstra's algorithm

Dijkstra's algorithm is a graph traversal one used to find the shortest path from a source node to all other ones in a weighted graph. It maintains a set of tentative distances for each node, gradually updating them as it explores the graph. At each step, it selects the node with the smallest tentative distance, visits its neighbors, and updates their tentative distances if a shorter path is found. The algorithm continues until all nodes have been visited or the destination one is reached. Dijkstra's algorithm is widely used in network routing protocols and pathfinding applications in computer science.

How it works:

Step 1: Set the initial distance to the starting vertex as 0 and all other onces to infinity. Create a priority queue or a min-heap to store vertices based on their current tentative distances.

Step 2: While there are vertices in the priority queue, select the vertex with the smallest tentative distance. Explore its neighbors and update their tentative distances if a shorter path is found.

Step 3: For each neighbor, calculate the sum of the tentative distance to the current vertex and the weight of the edge between them. If this sum is smaller than the current tentative distance to the neighbor, update the tentative distance.

Step 4: Mark the current vertex as visited to avoid redundant calculations.

Step 5: Repeat steps 2-4 until all vertices are visited or the destination vertex is reached.

Step 6: The final result is an array of shortest distances from the starting vertex to all other vertices.

Key Characteristics:

Single-Source Shortest Path: Dijkstra's algorithm finds the shortest paths from a single source node to all other nodes in a graph with non-negative edge weights.

Greedy Algorithm: The algorithm is a greedy algorithm as it selects the next node to visit based on the shortest known distance from the source node. It iteratively expands the frontier of explored nodes by selecting the node with the smallest tentative distance.

Initialization: Dijkstra's algorithm initializes the distances from the source node to all other nodes as infinity, except it itself, which is assigned a distance of zero. It also maintains a priority queue (often implemented using a min-heap) to keep track of nodes with the smallest tentative distances.

Relaxation: The algorithm performs relaxation of edges during each iteration. For each node in the priority queue, it considers all its outgoing edges and updates the tentative distance to its neighboring nodes if a shorter path is found through the current node.

Termination: Dijkstra's algorithm terminates when all nodes have been visited or when the priority queue becomes empty. At termination, the shortest path distances from the source node to all other nodes have been calculated.

Non-Negative Edge Weights: The algorithm requires non-negative edge weights. If negative edge weights are present, the algorithm may not produce correct results.

Space Complexity: The space complexity of Dijkstra's algorithm is (O(V)) for storing the distances and priority queue, where (V) is the number of vertices in the graph.

Applications:

Routing in Networks: Dijkstra's algorithm is commonly used in network routing protocols, such as Open Shortest Path First (OSPF) and Intermediate System to Intermediate System (IS-IS). It helps in finding the shortest paths for data packets to traverse through computer networks efficiently. It's also applicable to Telecommunications Networks

Navigation Systems: The algorithm is used in GPS navigation systems to calculate the shortest routes between locations. It helps in providing users with optimal directions for reaching their destinations while considering factors like traffic congestion and road conditions.

Transportation Networks: Dijkstra's algorithm is applied in transportation planning and management systems to optimize routes for vehicles, such as buses, taxis, and delivery trucks. It helps in minimizing travel time and fuel consumption while maximizing efficiency.

Airline Route Planning: The algorithm is used in airline route planning systems to determine the shortest paths between airports. It helps airlines in optimizing flight schedules, minimizing travel distances, and reducing operational costs.

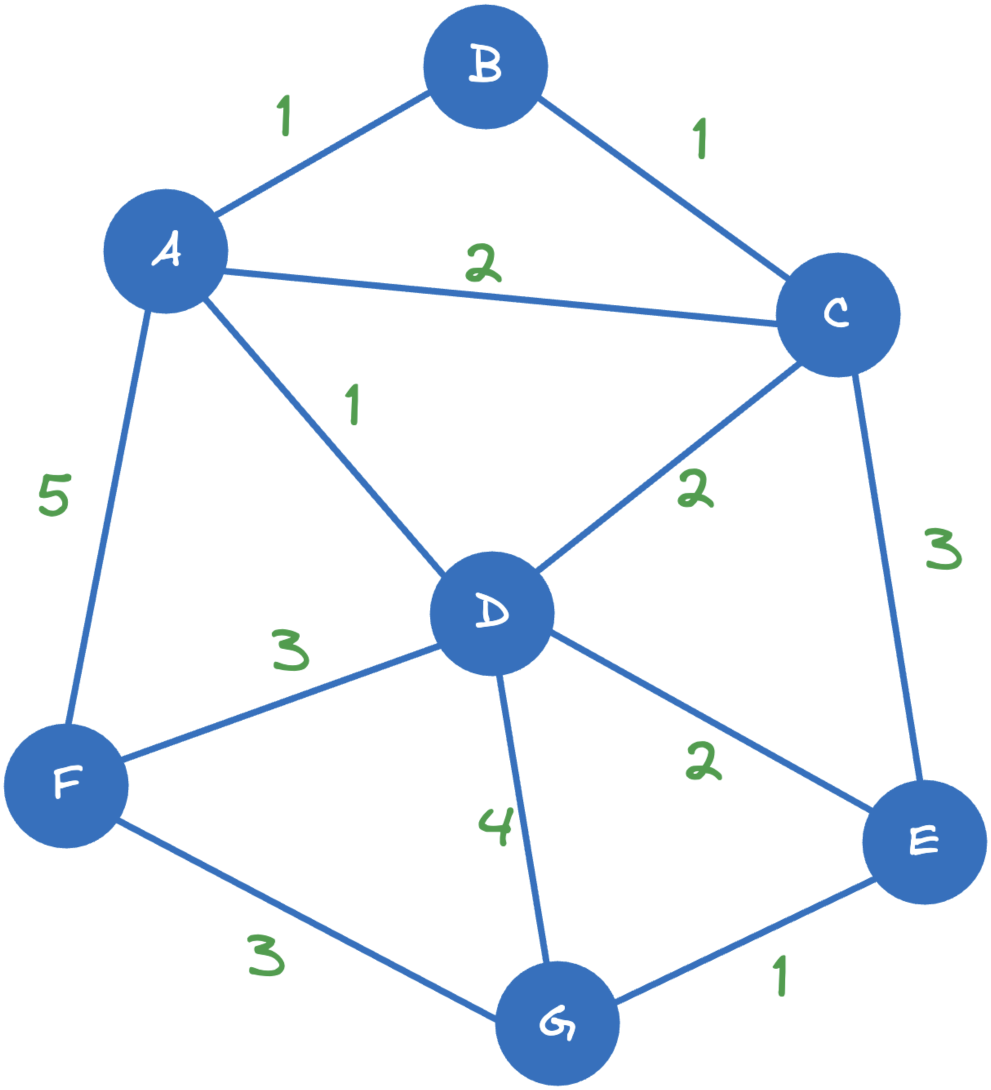
Robotics and Autonomous Vehicles: Dijkstra's algorithm is applied in robotics and autonomous vehicle navigation systems to plan collision-free paths between locations. It helps in ensuring safe and efficient movement of robots and vehicles in dynamic environments.

Supply Chain Optimization: The algorithm is used in supply chain management systems to optimize transportation routes for goods and products. It helps in minimizing shipping costs, reducing delivery times, and improving overall supply chain efficiency.

Emergency Response Planning: Dijkstra's algorithm is applied in emergency response planning systems to calculate the shortest routes for emergency vehicles, such as ambulances and fire trucks, to reach incident locations. It helps in improving emergency response times and saving lives.

Time Complexity:

The time complexity of Dijkstra's Algorithm is O((V + E) log V) using a priority queue or min-heap, where V is the number of vertices and E is the number of edges.



Example:

```
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
```

\newpage

Floyd-Warshall algorithm

The Floyd-Warshall Algorithm is a dynamic programming algorithm used to find the shortest paths between all pairs of vertices in a weighted graph. Unlike Dijkstra's algorithm and Bellman-Ford one, Floyd-Warshall works with graphs that can have both positive and negative edge weights and can handle graphs with cycles. The algorithm iteratively updates the shortest path distances between all pairs until reaching the optimal solution.

How it works:

Step 1: Create a matrix to represent the distances between all pairs of vertices. Initialize it with the direct edge weights and set the distances to infinity where there is no direct edge. Define the diagonal of the matrix to zeros since the distance from a vertex to itself is zero.

Step 2: For each vertex 'k', iterate through all pairs of vertices 'i' and 'j'. Check if the path from 'i' to 'j' through 'k' is shorter than the current known path from 'i' to 'j'. If yes, update the distance from 'i' to 'j' with the shorter path.

Step 3: Repeat the process for all vertices as intermediate ones ('k'). After each iteration, the matrix reflects the shortest distances between all pairs of vertices considering all possible intermediate vertices.

Step 4: The final matrix contains the shortest distances between all pairs of vertices.

Key Characteristics:

All-pairs shortest path: Unlike Dijkstra's algorithm, which finds the shortest paths from a single source vertex to all other vertices, Floyd-Warshall finds the shortest paths between all pairs of vertices in a weighted graph.

Dynamic programming approach: The algorithm uses a dynamic programming approach to solve the problem. It iteratively builds up solutions to subproblems, which are then used to solve larger subproblems until the entire problem is solved.

Space complexity: The space complexity of the algorithm is O(V^2), where V is the number of vertices in the graph. This is because the algorithm requires a matrix to store the shortest distances between all pairs of vertices.

Handling negative edge weights: Unlike Dijkstra's algorithm, the Floyd-Warshall one can handle graphs with negative edge weights, as long as there are no negative cycles. However, if there are negative cycles, the algorithm can't produce correct results, as there would be no shortest path in such cases.

Matrix representation: The algorithm typically uses a matrix to represent the graph and compute the shortest paths. The matrix contains the distances between all pairs of vertices, with initial values being the weights of the edges if they exist and infinity otherwise.

Intermediate vertices: In each iteration of the algorithm, it considers all vertices as potential intermediate ones in the shortest path between any pair of them. It updates the shortest distances between vertices based on whether using the intermediate vertex reduces the path length.

Path reconstruction: While the primary purpose of the Floyd-Warshall algorithm is to compute the shortest paths between all pairs of vertices, it can also be modified to reconstruct the actual paths if required, by storing additional information during the execution of the algorithm.

Applications:

Routing algorithms: In computer networking, the Floyd-Warshall algorithm can be used in routing protocols to compute shortest paths between all pairs of nodes in a network. It helps in finding the optimal paths for data packets to traverse from a source to a destination through intermediate nodes.

Traffic management: Transportation networks, such as road or rail networks, can utilize the Floyd-Warshall algorithm to optimize traffic flow by determining the shortest paths between all pairs of locations. This helps in minimizing travel time and congestion.

Flight scheduling: Airlines use the Floyd-Warshall algorithm to optimize flight routes and schedules by computing the shortest paths between airports. This ensures efficient utilization of resources and minimizes flight delays.

Geographic information systems (GIS): GIS applications use the Floyd-Warshall algorithm for route planning, navigation, and network analysis. It helps in finding the shortest paths between different geographic locations considering various factors such as terrain, road conditions, and traffic.

Robotics and autonomous vehicles: In robotics and autonomous vehicle navigation, the Floyd-Warshall algorithm can be employed to plan optimal paths for robots or vehicles to navigate in complex environments while avoiding obstacles and reaching their destinations efficiently.

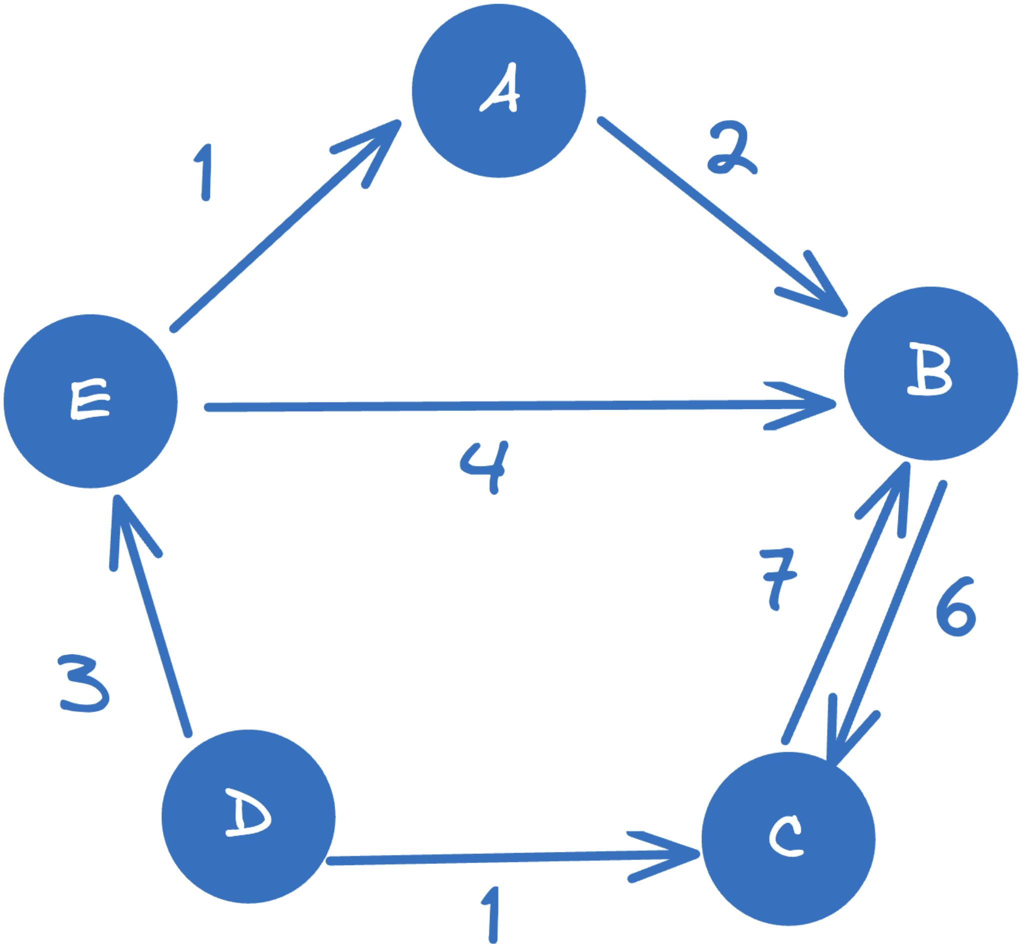
Telecommunication networks: Telecommunication networks, including telephone and data networks, use the Floyd-Warshall algorithm to optimize communication paths between all pairs of nodes. This ensures efficient data transmission and minimizes network congestion.

Social network analysis: In social network analysis, the Floyd-Warshall algorithm can be applied to model relationships between individuals or entities in a network. It helps in identifying the shortest paths between users, analyzing network connectivity, and studying information diffusion processes.

Game development: The Floyd-Warshall algorithm can be used in game development for pathfinding in game environments. It helps in determining the shortest paths for characters or objects to navigate through obstacles in a game world.

Time Complexity:**

The time complexity of Floyd-Warshall Algorithm is (O(Vˆ3)), where (V) is the number of vertices in the graph.



Example:

```
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);
}
```

\newpage

Ford Fulkerson algorithm

The Ford-Fulkerson Algorithm is an iterative method to compute the maximum flow in a flow network. It was initially designed to solve the max-flow min-cut problem, where the objective is to find the maximum amount of flow that can be sent from a designated source to a designated sink in a directed graph. The algorithm iteratively augments paths from the source to the sink, increasing the flow until it reaches its maximum value.

How is works:

Step 1: Begin with an initial flow of zero. Determine the residual graph, which represents the remaining capacity for each edge.

Step 2: Find an augmenting path from the source to the sink in the residual graph. "An augmenting path" is a path with available capacity on all its edges.

Step 3: Determine the maximum flow that can be added along the augmenting path. This is the minimum capacity value of the edges on this.

Step 4: Update the residual graph by subtracting the flow added along the augmenting path and adding the reverse flow.

Step 5: Repeat steps 2-4 until there are no more augmenting paths.

Step 6: The final flow is the maximum flow in the network.

Key Characteristics:

Maximum flow: The Ford-Fulkerson algorithm finds the maximum flow that can be sent from a source node to a sink node in a flow network. This is useful in various applications such as transportation networks, network flow optimization, and resource allocation.

Augmenting paths: The algorithm relies on the concept of augmenting paths, which are paths from the source to the sink that have available capacity for additional flow. It repeatedly finds augmenting paths and increases the flow along those paths until no more augmenting ones can be found.

Residual graph: To efficiently find augmenting paths, the algorithm maintains a residual graph, which represents the remaining capacity on each edge after the current flow has been sent. This allows the method to explore different paths and incrementally increase the flow.

Capacity constraints: The Ford-Fulkerson algorithm respects capacity constraints on edges, ensuring that the flow along each edge does not exceed its maximum capacity. It increases the flow along augmenting paths without violating these constraints.

Termination condition: The algorithm terminates when no more augmenting paths can be found in the residual graph. At this point, the flow obtained is the maximum flow possible in the network.

Applications:

Network flow optimization: The primary application of the Ford-Fulkerson algorithm is in network flow optimization. It is used in various scenarios such as transportation networks, communication networks, and supply chain management to maximize the flow of goods, data, or resources from a source to a destination while respecting capacity constraints.

Maximum flow problems: The algorithm is specifically designed to solve maximum flow problems, where the goal is to determine the maximum amount of flow that can be sent through a network from a source to a sink. This has applications in tasks such as traffic management, fluid dynamics, and resource allocation.

Minimum cut problems: The Ford-Fulkerson algorithm can also be used to solve minimum cut problems, where the objective is to find the smallest set of edges whose removal disconnects the source from the sink in a flow network. Minimum cut problems have applications in network reliability analysis, image segmentation, and clustering algorithms.

Bipartite matching: The algorithm can be applied to solve bipartite matching problems, where the goal is to find the maximum matching between elements of two disjoint sets. This has applications in job assignment, resource allocation, and stable marriage problems.

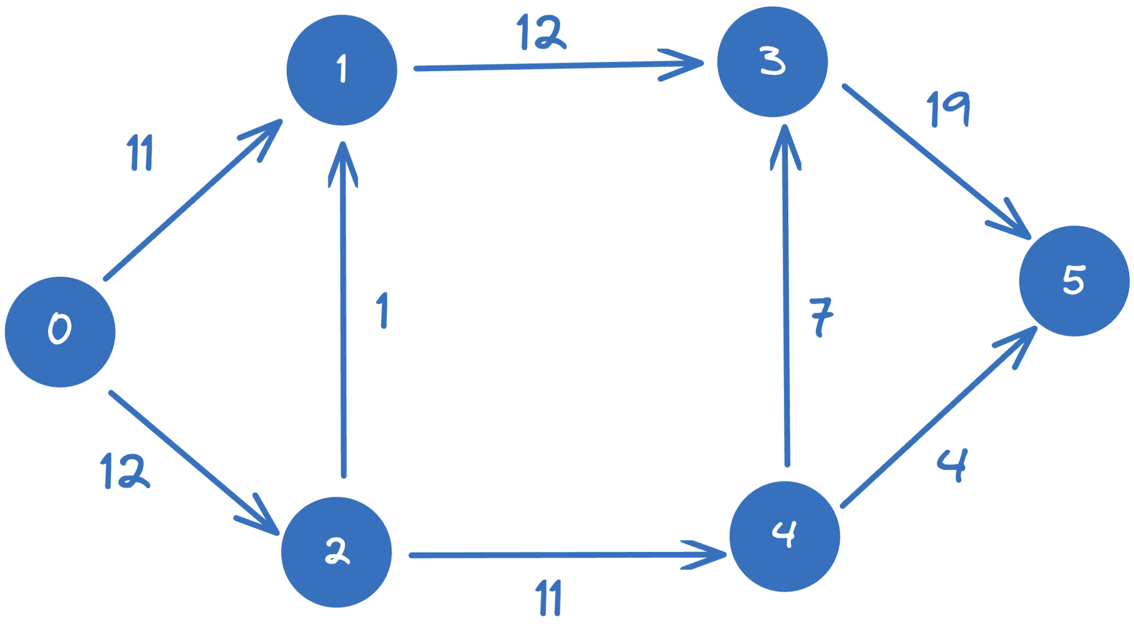
Telecommunications: Telecommunication networks utilize the Ford-Fulkerson algorithm for optimizing data transmission rates, routing calls through phone networks, and allocating bandwidth efficiently. It ensures that communication networks operate at maximum capacity while maintaining quality of service.

Hydraulic engineering: In hydraulic engineering, the Ford-Fulkerson algorithm can be used to model fluid flow in pipes, channels, and rivers. It helps in designing efficient water distribution systems, flood control measures, and irrigation networks.

Supply chain management: The Ford-Fulkerson algorithm has applications in supply chain management for optimizing inventory management, production scheduling, and distribution networks. It helps in minimizing transportation costs and maximizing throughput in supply chain operations.

Time Complexity:

The time complexity of the Ford-Fulkerson Algorithm is not strictly defined, as it depends on the choice of augmenting paths. In the worst case, the algorithm may not terminate if the paths are not chosen carefully. When implemented with the Edmonds-Karp variant, where the shortest augmenting paths are chosen using Breadth-First Search, the time complexity is (O(VEˆ2)), where (V) is the number of vertices and (E) is the number of edges.



Example:

```
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);
```

\newpage

- Binary search
- Binary tree in order traversal
- Binary tree postorder traversal
- Binary tree preorder traversal
- Breadth-first search
- Bubble sort
- Depth-first search
- Diffie hellman algorithm
- Dijkstra's algorithm
- Floyd-Warshall algorithm
- Ford Fulkerson algorithm
- Graph adjacency list
- Graph adjacency matrix
- algorithms all
- Insertion sort
- Interpolation search
- Interval search
- Jump search
- Linear search
- Merge sort
- Quick sort
- Selection sort
- Ternary search \newpage

Insertion sort

Insertion Sort is a straightforward sorting algorithm that builds the sorted array one element at a time. It is much less efficient on large lists than more advanced algorithms such as quicksort, heapsort, or merge sort. However, it has some advantages: it is simple to implement, efficient for small datasets, and performs well for partially sorted arrays.

How it works:

Step 1: Consider the first element to be a sorted subarray and the rest as an unsorted subarray

Step 2: Sequentially iterate over the unsorted elements of the array and move them to the sorted subarray.

Step 3: For each unsorted element, compare the current element with the elements before it

Step 4: If the current element is greater than its preceding one, leave it there, as it is already at the desired position. If not, keep comparing it with the elements before it until a smaller or equal one is found: Insert the current element after and all comparisons are made, and no smaller one is found: Insert the current item at the beginning of the array

Step 5: Repeat the above process for every element of the unsorted subarray until the array is sorted

Key Characteristics:

Basic Algorithm: Insertion sort builds the final sorted array one element at a time, by repeatedly taking the next element from the unsorted part and inserting it into its correct position in the sorted part.

In-place Sorting: It sorts elements by moving them within the array, without requiring additional storage space.

Stable: It is a stable sorting algorithm, meaning that it preserves the relative order of equal elements.

Adaptive: Insertion sort is adaptive to some extent. If the input list is almost sorted, it can perform better than its worst-case time complexity.

Efficient for Small Datasets: It is efficient for sorting small datasets or arrays, particularly when the array is nearly sorted.

Simple Implementation: It is straightforward to implement and understand, making it a good choice for educational purposes or for sorting small lists.

Efficient for Nearly Sorted Data: The algorithm performs well on data that is already partially sorted, as it only requires shifting a few elements to insert a new one.

Applications:

Small Datasets: Insertion sort is suitable for sorting small datasets efficiently due to its simplicity and low overhead.

Nearly Sorted Data: It performs well on data that is already partially sorted, requiring minimal adjustments to insert new elements.

Online Algorithms: It can be useful in situations where data is continuously arriving, as it can efficiently incorporate new elements into a sorted list.

Embedded Systems: Due to its simple implementation and low memory requirements, Insertion sort is suitable for sorting small arrays in resource-constrained environments, such as embedded systems.

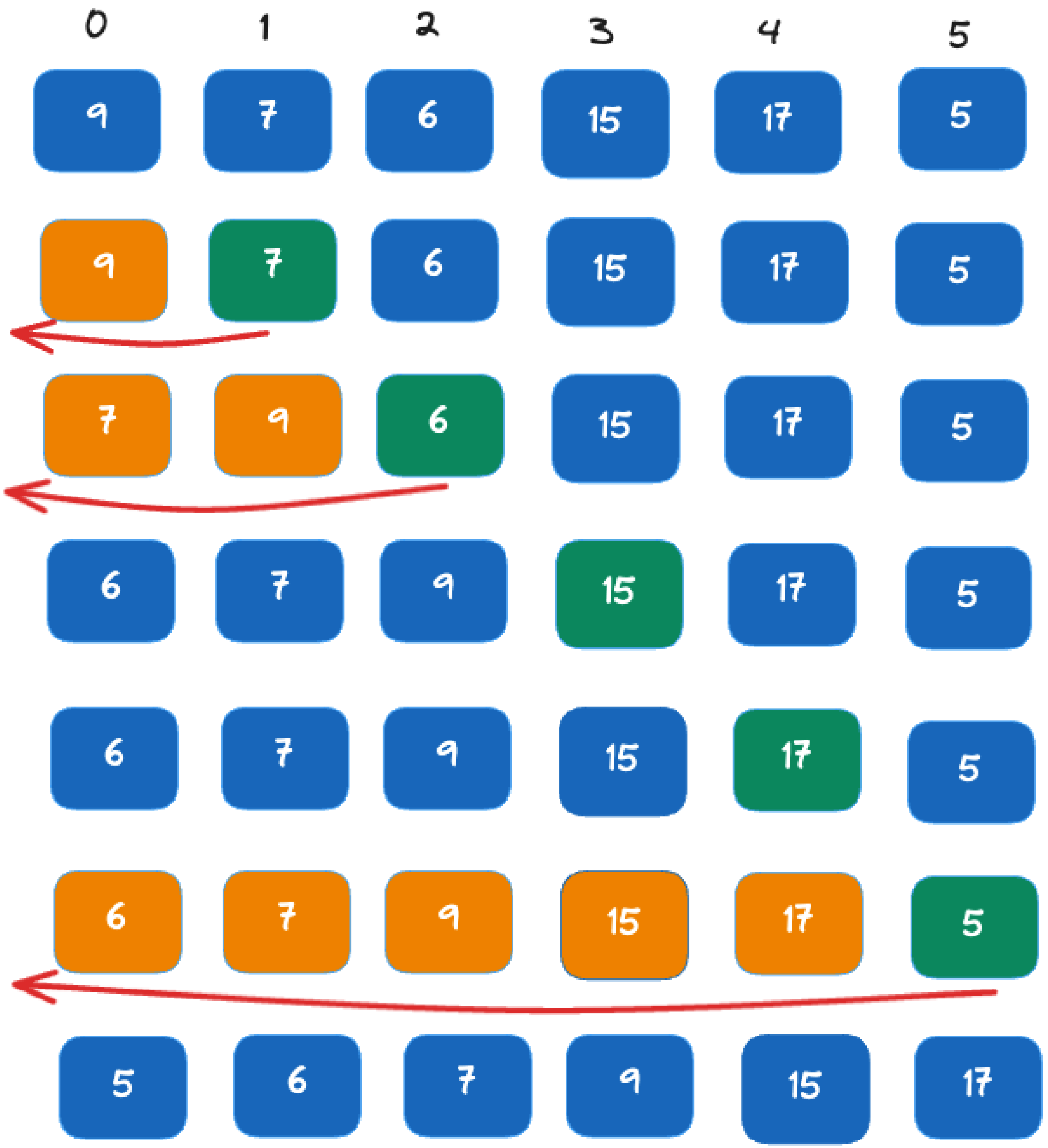
Educational Purposes: It is often used as a teaching tool to introduce sorting algorithms due to its straightforward implementation and understanding.

Stable Sorting: Insertion sort's stability makes it suitable for applications where preserving the relative order of equal elements is necessary.

Auxiliary Sorting Algorithm: It can be used as a building block within more complex sorting algorithms or algorithms requiring a partially sorted list.

Time Complexity:

Insertion Sort has a time complexity of O(nˆ2) in the worst case, where 'n' is the number of elements in the array. Despite its quadratic time complexity, the algorithm is often more efficient on small datasets or partially sorted arrays compared to other quadratic sorting algorithms. It's also an in-place sorting algorithm, meaning it doesn't require additional memory.



Example

```
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]));
```

```
  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;
          }
      }
  }
```

\newpage

Interpolation search

Interpolation Search is a an algorithm designed for finding a specific target value in a sorted array. Unlike linear or binary search, this algorithm utilizes the characteristics of the data distribution to make more informed decisions about where to look for the target. It is particularly effective when the data has a uniform distribution.

How it works:

Step 1: It utilizes linear interpolation to estimate the likely position of the target value in the array.

Step 2: Instead of evenly dividing the search space, as in binary search, the algorithm estimates the probable position of the target based on its value relative to the minimum and maximum values in the array.

Step 3: It calculates an estimate of the target's position by considering the relative location of the target with respect to the minimum and maximum values in the current search space.

Step 4: Based on the calculated estimate, the algorithm narrows down the search space and repeats the process until the target is found or the search space is exhausted.

Key Characteristics:

Estimation of Position: Interpolation search estimates the probable position of the target value within the array based on the values at the ends of the array. Unlike binary search, which always divides the search space in half, interpolation search calculates a position closer to the target based on the distribution of values in the array.

Requirement of Sorted Array: Similar to binary search, interpolation search requires the input array to be sorted in ascending order. This is necessary to make assumptions about the distribution of values within the array and to perform efficient searches.

Variable Step Size: In interpolation search, the step size for narrowing down the search interval varies depending on the estimated position of the target value. This variable step size allows for more efficient convergence towards the target value compared to fixed step sizes used in binary search.

Calculation of Position: The estimated position of the target value is calculated using interpolation formula, which typically involves linear interpolation. However, other interpolation techniques such as quadratic or exponential interpolation can also be used depending on the nature of the data.

// **Complexity**: The time complexity of interpolation search is (O(\log \log n)) on average, where (n) is the number of elements in the array. This complexity is better than binary search in certain scenarios, especially when the elements are uniformly distributed.

// **Worst-case Scenario**: While interpolation search generally performs well, it can degrade to (O(n)) time complexity in the worst-case scenario, particularly when the distribution of values in the array is highly skewed or uneven.

Handling Non-uniform Distributions: Interpolation search is particularly effective when the values in the array are uniformly distributed. However, in cases where the distribution is non-uniform, interpolation search may not provide significant advantages over other search algorithms like binary search.

Implementation: Interpolation search can be implemented recursively or iteratively. Iterative implementation is often preferred due to its simplicity and efficiency. However, recursive implementation may be more intuitive for some programmers.

// **Space Complexity**: Interpolation search has a space complexity of (O(1)) since it does not require any additional space proportional to the input size beyond a few variables used for indices and calculations.

Applications:

Numerical Data Retrieval: Interpolation search is commonly used in databases and data retrieval systems where data is stored in sorted arrays. It allows for fast retrieval of numerical data, especially when the values are uniformly distributed.

Scientific Computing: In scientific computing, the algorithm can be applied in scenarios such as searching for specific data points within large datasets or performing numerical simulations where fast access to data is crucial.

Symbol Tables: Interpolation search can be used in symbol tables and compilers for quick retrieval of symbols or identifiers. This is particularly useful in programming languages or applications that deal with large symbol tables.

Searching in Large Datasets: The algorithm can efficiently handle large datasets where binary search may become less efficient due to the fixed step size. It's used in applications dealing with large volumes of data such as search engines, databases, and data analytics platforms.

Approximate Search: Interpolation search can be adapted for approximate search tasks where finding an exact match is not necessary. For example, in spell checkers or auto-complete features in text editors, interpolation search can quickly narrow down the search space for approximate matches.

// **Time-Critical Systems**: In real-time systems where response time is critical, interpolation search can be beneficial due to its average-case time complexity of (O(\log \log n)). This makes it suitable for applications requiring fast search operations, such as in financial trading systems or telecommunications networks.

Finding Closest Values: This algorithm can be used to find the closest value to a given target value within a sorted array. This application is useful in scenarios such as image processing, signal processing, and sensor data analysis.

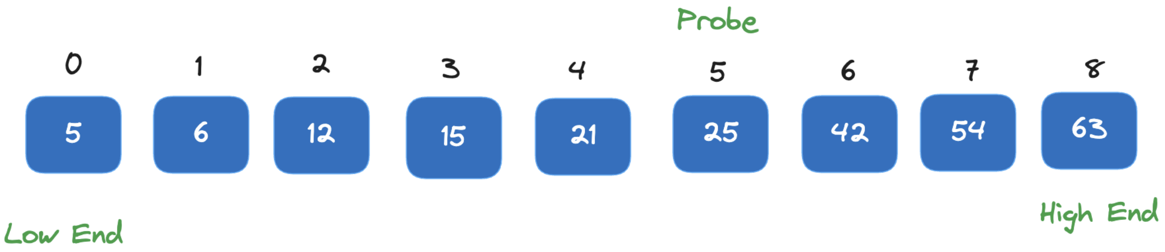
Data Compression and Encoding: Interpolation search algorithms can be used in compression and encoding techniques to efficiently search for patterns or symbols within compressed data streams. This is common in multimedia compression algorithms like JPEG and MPEG.

Geospatial Data Processing: In geographic information systems (GIS) and mapping applications, interpolation search can be used to quickly retrieve spatial data points or locations based on their coordinates, enabling efficient spatial analysis and visualization.

Machine Learning and Data Mining: Interpolation search can be applied in machine learning algorithms and data mining tasks for efficient searching and retrieval of data points during training or querying processes, especially in high-dimensional spaces.

Time Complexity:

The time complexity of Interpolation Search is O(log log n) on average, where "n" is the number of elements in the array. In the best case, it can be O(1), and in the worst case, it can be O(n). However, the average case is often more relevant, and it is O(log log n) under certain assumptions about the distribution of the data.



Example

```
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;
}
```

\newpage

Jump search

Jump Search is an algorithm designed for sorted arrays. It is a block-based search algorithm that works by jumping ahead by fixed steps and then linearly searching within the block for the target element. The algorithm combines the efficiency of binary search with the simplicity of linear search.

How it work:

Step 1: Determine the jump size by taking the square root of the array length. This ensures a balanced trade-off between the number of jumps and the linear search within a block.

Step 2: Start at the beginning of the array and move ahead by the calculated jump size until finding a value that is greater than or equal to the target.

Step 3: Perform a linear search within the block from the previous jump until finding the target element or checking that it is not there.

Step 4: Repeat the process until the entire array is searched or the target element is found.

Key Characteristics:

Sorted Array: Jump search requires the array to be sorted in ascending order. This is required characteristic.

Block Jumping: The algorithm works by jumping ahead by fixed steps (often referred to as block size or jump size) instead of dividing the search space in half at each step like binary search.

Optimal Jump Size: The optimal jump size is usually calculated as the square root of the length of the array. This ensures a balance between the number of moves and the amount of comparisons needed.

Linear Search Within Blocks: After jumping to a particular block, the algorithm performs a linear search within that block.

Requires Random Access: Jump search requires random access to elements in the array. This means it is more suitable for data structures like arrays or lists.

Adaptive: The algorithm can be adapted to different scenarios by adjusting the jump size. Smaller jump sizes can reduce the number of comparisons but increase the number of jumps, while larger jump sizes have the opposite effect.

Not Recursive: Unlike binary search, jump search is typically implemented iteratively rather than recursively.

Suitable for Large Arrays: The algorithm is particularly suitable for large arrays due to more efficient work with memory.

Balances Efficiency and Simplicity: Jump search strikes a balance between the simplicity of linear search and the efficiency of binary search, making it a practical choice in scenarios where a trade-off between simplicity and efficiency is acceptable.

Space Complexity: Jump search has a space complexity of O(1), meaning it requires only a constant amount of additional space.

Applications: The algorithm can be useful in scenarios where binary search is not suitable due to memory constraints or when the dataset is relatively large and needs a more efficient search than linear search.

Applications:

Search in Databases: Jump search can be applied in databases to search for records based on certain criteria, especially when the data is stored in sorted arrays or tables.

File Systems: In file systems, the algorithm can be utilized to locate files within directories or to search for specific content within files.

Web Indexing: Jump search can be employed in web indexing algorithms to search for keywords within web pages or documents.

Data Compression Algorithms: The algorithm can be used in data compression ones to quickly locate patterns or repeated sequences within data streams.

Sparse Arrays: Jump search is effective for searching in sparse arrays where there are large gaps between elements. It helps to decrease the number of unnecessary comparisons.

Database Indexing: The algorithm can be utilized in database indexing structures, such as B-trees or skip lists, to quickly locate entries within the index. It provides a fast method for searching within indexed structures.

Text Editors and Word Processors: Jump search can be used into text editors or word processors to implement features like find word or bunch of phrase.

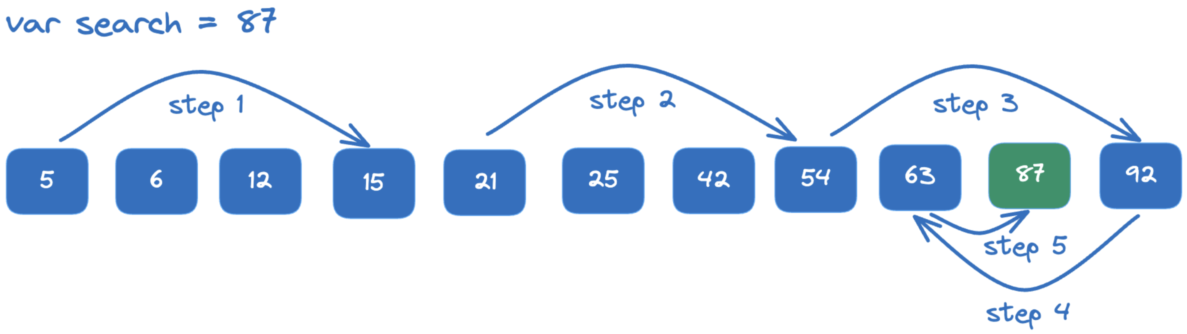
Genetic Sequence Analysis: In bioinformatics, the algorithm can be applied to efficiently search for specific patterns or sequences within genetic data. It helps analyze DNA or protein sequences in large datasets.

Network Routing: Jump search can be used in network routing algorithms to efficiently search for routes or paths between network nodes. It helps to find an optimal route while minimizing the number of hops or comparisons.

Data Mining: The algorithm can be employed in data mining tasks to search for patterns or anomalies within large datasets. It helps in quickly identify relevant data points or clusters.

Time Complexity:

The time complexity of Jump Search is -add formula-„ where-add formula-, is the size of the array. This makes it efficient for large datasets when compared to linear search but may be outperformed by binary search for certain scenarios.



Example:

```
function jumpSearch(arr: number[], target: number): number {
  const n = arr.length;
  const step = Math.floor(Math.sqrt(n));
  let prev = 0;

  while (arr[Math.min(step, n) - 1] < target) {
    prev = step;
    step += Math.floor(Math.sqrt(n));
    if (prev >= n) {
      return -1;
    }
  }

  for (let i = prev; i < Math.min(step, n); i++) {
    if (arr[i] === target) {
      return i;
    }
  }

  return -1;
}
```

\newpage

Linear search

Linear Search, also known as sequential search, is a simple searching algorithm that finds the position of a target value within a list or array. It works by iterating through the elements one by one until the target value is found or the entire list has been searched.

How Linear Search Works:

Step1 : Looking at the first element in the list.

Step 2: Compare with Target

Step 3: If the current element is equal to the target value, the search is successful, and the index or position of the element is returned, if not continue by moving to the next element in the list.

Key Characteristics:

Sequential Nature: Linear search sequentially checks each element in the list or array until the target element is found or until the end of the list is reached.

Unordered List Search: The algorithm is effective for searching elements in unordered lists, as it does not rely on any specific ordering of elements.

No Pre-processing Required: Unlike some other search algorithms like binary search, linear search does not require the list to be sorted beforehand.

Suitable for Small Lists: Linear search can be suitable for small lists or arrays where the overhead of sorting the list or using more complex algorithms is unnecessary.

Applicability: It can be used in various scenarios, such as searching for an item in a database, finding a specific record in a file, or locating a value in an unsorted array.

Ease of Implementation: Linear search is straightforward to implement, making it a good choice for introductory programming exercises and situations where simplicity is preferred over performance.

Search Completeness: This algorithm guarantees that if the element exists in the list, it will eventually be found after traversing through the entire list.

No Auxiliary Space Required: Linear search typically does not require any additional memory beyond what is already allocated for the list itself.

Suboptimal for Large Lists: While this algorithm is efficient for small lists, its linear time complexity makes it suboptimal for large lists, especially compared to more efficient search algorithms like binary search on sorted lists.

Applications:

Database Management Systems: In database systems, linear search can be used to search for records based on certain criteria when there's no specific indexing available or when the data is not ordered in any particular way.

Finding an Item in a List: Linear search is commonly used in programming scenarios to find an element within a list or array. This can be applicable in tasks such as searching for a specific value in a list of student grades, employee IDs, or product codes.

Searching Files: The algorithm can be used in applications that involve searching for a particular record or line within a file. For instance, searching for a specific keyword in a text file or finding a particular entry in a log file.

User Interface Development: In user interfaces, the algorithm can be used to locate elements within a list or menu. For example, searching for an item in a dropdown menu or a list of options presented to the user.

Unsorted Data: When the data is not sorted or when sorting the data beforehand is not feasible due to time or resource constraints, linear search becomes the go-to method for searching.

Debugging: Linear search can be utilized during debugging sessions to search through an array or data structure to locate specific values or objects that are causing issues.

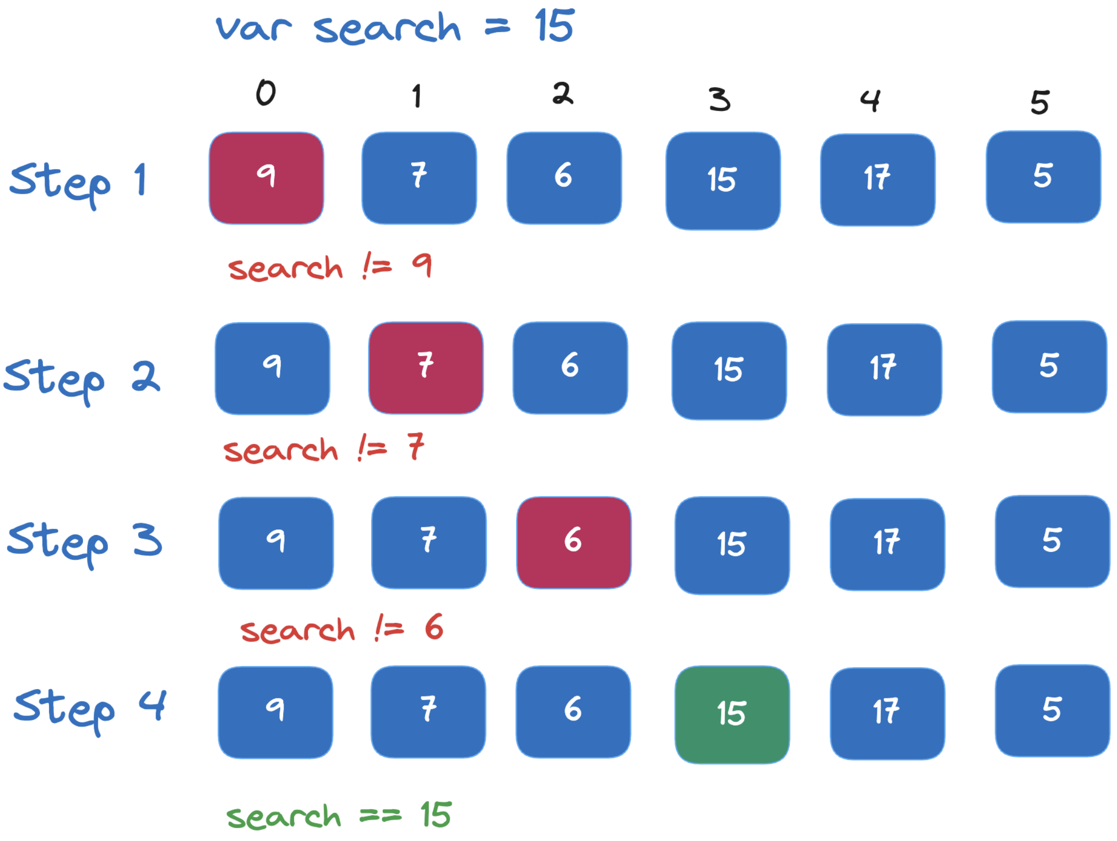
Educational Purposes: Linear search serves as a fundamental concept in computer science education, providing a simple example of a search algorithm that helps students understand the concept of searching and algorithms in general.

Small Datasets: The algorithm is suitable for scenarios where the dataset is small or the performance overhead of more complex algorithms is unnecessary.

Backup Systems: Linear search can be used in backup systems to locate and retrieve specific files or pieces of data from a backup storage when needed.

Time Complexity:

The time complexity of Linear Search is O(n), where 'n' is the number of elements in the array. In the worst case, the algorithm may need to iterate through the entire list to find the target value. While Linear Search is simple, it may not be the most efficient for large datasets, especially when compared to more advanced search algorithms like binary search on sorted lists. However, it is easy to understand and implement.



Example

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

  return -1;
}
```

\newpage

Merge sort

Merge Sort is a comparison-based sorting algorithm that follows the divide-and-conquer paradigm. It works by dividing the unsorted array into 'n' sub-arrays, each containing one element. It then repeatedly merges these sub-arrays to produce new sorted ones until there is only one sub-array remaining - the fully sorted array.

How it works:

Step 1: The unsorted array is recursively divided into two halves until each sub-array contains only one element. This is the base case of the recursion.

Step 2: The adjacent sub-arrays are then recursively merged to produce new sorted ones. This process continues until there is only one sub-array remaining the fully sorted array.

Step 3: The key operation in Merge Sort is the merging of two sorted sub-arrays to produce a single, sorted one. This involves comparing elements from the two sub-arrays and placing them in the correct order.

Key Characteristics:

Divide and Conquer: Merge Sort is a divide-and-conquer algorithm that breaks down the array into smaller sub-arrays, sorts each of them recursively, and then merges the result to final sorted array.

Stable Sorting: It is a stable sorting algorithm, meaning that it preserves the relative order of equal elements during sorting.

Efficient for Large Datasets: This efficient time complexity makes it well-suited for sorting large datasets, outperforming many other sorting algorithms for such scenarios.

In-place vs. Out-of-place: While Merge Sort can be implemented both in-place (where it sorts the array within its original memory space) and out-of-place (where it requires additional memory space), the out-of-place implementation is more common due to its simplicity and efficiency.

Parallelizability: The algorithm can be parallelized easily, allowing it to take advantage of multiple processors or threads for faster sorting of data.

Space Complexity: Merge Sort typically has a space complexity of O(n), as it requires additional memory space for the temporary arrays used during the merging process.

Reliable Performance: Merge Sort's consistent time complexity and stable sorting make it a reliable choice for applications where predictable performance and stability are essential.

Applications:

Sorting Large Datasets: Merge Sort is highly efficient for sorting large datasets, making it suitable for various applications where sorting a significant amount of data is required. This includes tasks like sorting database records, organizing files on disk, or processing large sets of data in scientific research.

External Sorting: It is commonly used in external sorting algorithms where data is too large to fit into memory entirely. It efficiently sorts data stored on external storage devices like hard drives by dividing it into manageable chunks, sorting them in memory, and then merging them back together.

Network Routing: Merge sort is utilized in network routing algorithms for organizing and sorting routing tables efficiently. In networking applications, sorting large sets of routing information is crucial for optimizing data transmission and network performance.

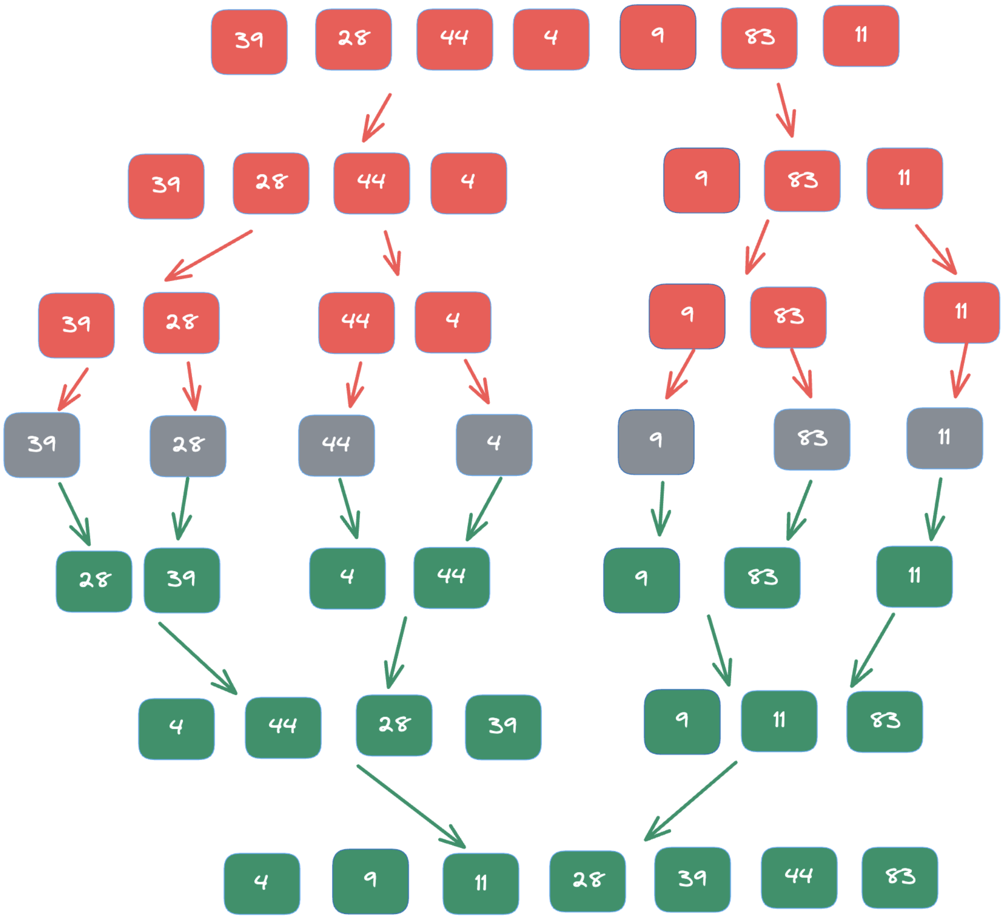
Parallel Processing: Its divide-and-conquer approach lends itself well to parallel processing. It can be parallelized across multiple processors or threads, allowing for faster sorting of data in parallel computing environments.

Optimizing Data Processing Pipelines: Merge Sort can be used in data processing pipelines to efficiently sort and merge streams of data from different sources. This is common in real-time data analytics, where sorted data is essential for performing efficient queries and analyses.

Operating Systems: It is employed in various components of operating systems for tasks like sorting file directories, managing memory allocation, and organizing system resources efficiently.

Time Complexity:

Merge Sort has a consistent time complexity of O(n log n) in all cases, where 'n' is the number of elements in the array. It is a stable sorting algorithm, meaning that equal elements maintain their relative order in the sorted output. While Merge Sort has a slightly higher space complexity due to the need for additional memory, its stability and predictable performance make it a widely used and reliable sorting algorithm.



Example:

```
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));
}
```

\newpage

Quicksort

Quick Sort is an efficient, comparison-based sorting algorithm that follows the divide-and-conquer paradigm. 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 recursively sorted.

How it works:

Step 1: Select a pivot. It might be any element at random, the first or last element, middle one.

Step 2: Partition the array around the pivot element. Move all the elements that less than pivot to the left of the it and move all elements that more or equal to the right side.

Step 3: After Step 2, the pivot element is in its correct position

Step 4: Apply the quicksort recursively on the left partition and then on the right partition

Step 5: Stop recursion when array is sorted when reach out the base case. It's an array of zero or one element.

Key Characteristics:

Divide and Conquer: it is a divide-and-conquer algorithm that partitions the array into smaller sub-arrays, sorts each of them recursively, and then combines them to obtain a sorted array.

In-place Sorting: Quick Sort often sorts elements in place within the array, which means it does not require additional memory beyond a few stack frames for recursion.

Unstable: The algorithm is generally an unstable, meaning that it may change the relative order of equal elements.

Efficiency: Quick Sort is highly efficient for large datasets due to its average-case time complexity of O(n log n). It outperforms many other sorting algorithms like Bubble Sort, Insertion Sort, and Selection Sort for larger datasets.

Pivot Selection: It's efficiency heavily relies on the selection of a good pivot element, which partitions the array into smaller sub-arrays. Common strategies for pivot selection include choosing the first, last, or middle element of the array, or using techniques like median-of-three.

Adaptive: The algorithm is adaptive, meaning that it performs better when the input data is partially sorted. This adaptability can lead to improved performance in many real-world scenarios.

Parallelizability: Quick Sort is easily parallelizable, which means it can take advantage of multiple processors or threads to sort data more quickly.

Applications:

General Sorting: Quick Sort is widely used for sorting large datasets efficiently in various software applications, including databases, operating systems, and programming languages.

Programming Competitions: It is a popular choice for sorting algorithms in programming competitions due to its efficiency and ease of implementation.

Libraries and Frameworks: The algorithm is often implemented in standard libraries and frameworks for sorting data structures like arrays, lists, and trees.

Online Sorting: Quick Sort's efficiency makes it suitable for online sorting scenarios where new data is continuously arriving and needs to be sorted quickly.

Data Analysis: It is utilized in data analysis tasks where sorting large datasets is a common requirement, such as in data mining, machine learning, and statistical analysis.

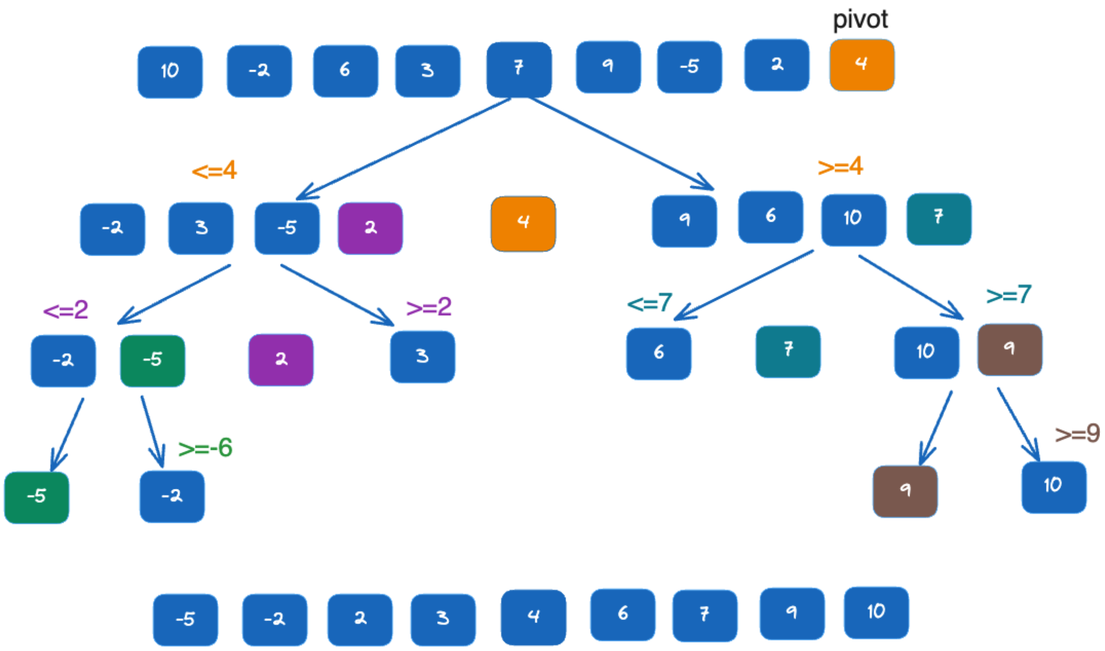
Operating Systems: It is used in various components of operating systems for tasks like file system management and memory allocation.

Numerical Methods: The algorithm can be applied in numerical methods and scientific computing for sorting arrays of numerical data efficiently.

Network Routing: Quick Sort is used in network routing algorithms for organizing and sorting routing tables efficiently.

Time Complexity:

Quick Sort has an average and best-case time complexity of O(n log n), where 'n' is the number of elements in the array. In the worst case, it is O(nˆ2), but this is rare when a good pivot selection strategy is used. This algorithm is often faster in practice than other O(n log n) ones, and it is widely used in various applications due to its efficiency.



Example:

```
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)];
  }
}
```

\newpage

Selection sort

Selection sort is a simple and efficient sorting algorithm that works by repeatedly selecting the smallest (or largest) element from the unsorted portion of the list and moving it to the sorted portion of the list.

How it works:

Step 1: Start with an unsorted list/array of elements.

Step 2: Repeat until the entire list is sorted: Find the minimum element in the unsorted portion and swap the minimum element with the leftmost element.

Step 3: Move the boundary between the portions one position to the right.

Step 4: Continue this process until the entire list is sorted.

Step 5: Once all elements are sorted, the process ends, and you have a fully ready list.

Key Characteristics:

Basic Algorithm: Selection sort divides the input list into a sorted and an unsorted region, repeatedly selects the smallest (or largest) element from the unsorted region and swaps it with the leftmost unsorted element.

In-place Sorting: Sorts elements by swapping them in place within the array or list.

Not Adaptive: Selection sort's performance doesn't improve even if the input is partially sorted.

Not Stable: May change the relative order of equal elements.

Simple Implementation: Easy to understand and implement.

Inefficient for Large Datasets: Due to its quadratic time complexity, it's inefficient for large datasets compared to more efficient algorithms like quicksort or mergesort.

Applications:

Educational Purposes: Selection sort is commonly used in educational settings to teach sorting algorithms due to its simplicity.

Small Datasets: Suitable for sorting small datasets where simplicity is prioritized over efficiency.

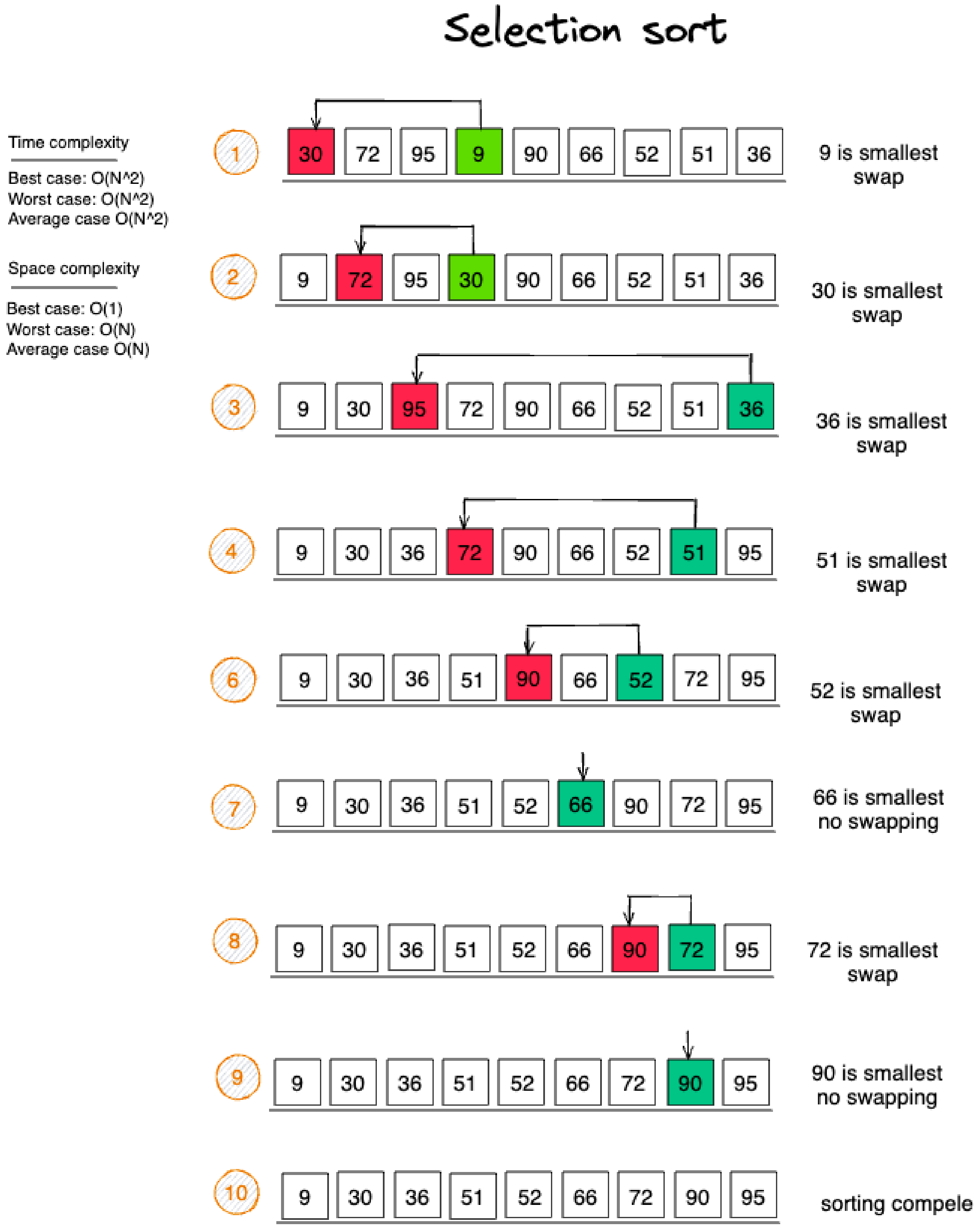
Embedded Systems: Can be used in environments with limited resources for sorting small arrays due to its simplicity and minimal memory requirements.

Testing and Debugging: Useful for quick implementation and verification of sorting functionality in software development.

Ad Hoc Sorting: Provides a simple and quick solution for one-time or temporary sorting needs in situations where efficiency is not critical.

Time complexity:

The time complexity of selection sort is O(nˆ2), where n is the number of elements in the array, due to its nested loops. The outer loop iterates through each element, and the inner loop searches for the minimum element in the unsorted portion, both running n times. This results in a quadratic time complexity of O(nˆ2), making selection sort less efficient for large datasets compared to other sorting algorithms.



Example:

```
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;
}
```

```
    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
```

\newpage

Ternary search

Ternary Search is a divide-and-conquer algorithm designed for efficiently finding the position of a target value in a sorted array. It works by separating the array into three parts and recursively narrowing down the search space until the target is found or not.

How it Works:

Step 1: Divide the sorted array into three parts.

Step 2: Then compare the target value with the elements at two points within the array, separating it into three segments. If the target is found at one of these points, the search is successful.

Step 3: Based on the comparisons, Ternary Search identifies whether the target lies in the first, second, or third segment of the array.

Step 4: The algorithm then recursively applies the same process to the identified segment. This recursion continues until the target is found or the search space is reduced to an empty array, indicating that the target is not present.

Key Characteristics:

Divide and Conquer Approach: Ternary search follows the "divide and conquer" paradigm. It repeatedly divides the search interval into three parts and narrows down the search space based on the value being searched for.

Applicability: Ternary search is applicable only on sorted arrays or functions that are unimodal (i.e., having a single peak or trough). It's commonly used to find the maximum or minimum value of a unimodal function.

Comparison Count: Ternary search reduces the size of the search space by one-third in each iteration, which means it usually performs fewer comparisons compared to binary search, especially when the desired element is closer to the ends of the array.

Implementation: Ternary search can be implemented recursively or iteratively. The recursive implementation is straightforward and elegant, but it might suffer from stack overflow for large input sizes. The iterative implementation, on the other hand, is more efficient in terms of space.

// **Mid-Point Calculation**: In each iteration, ternary search calculates two mid-points (m1 and m2) to divide the search space into three equal parts. The value of the mid-points is calculated as follows: (m1 = \text{start} + \frac{\text{end} - \text{start}}{3}) and (m2 = \text{end} - \frac{\text{end} - \text{start}}{3}).n.

// **Space Complexity**: Ternary search has a space complexity of (O(1)) since it does not require any extra space proportional to the input size beyond a few variables used for indices and comparison values.

Applications:

Finding Local Extrema: Ternary search is commonly used to find the maximum or minimum value of a unimodal function over a given interval. This application arises in various fields such as optimization problems, mathematical modeling, and computer graphics.

// **Finding a Peak in Peak Finding Problem**: In a 1D peak finding problem where a peak is defined as an element greater than or equal to its neighbors, ternary search can efficiently locate a peak in (O(\log_3{n})) time complexity, where (n) is the size of the array.

Optimizing Functions: Ternary search can be applied in optimization problems where a function needs to be optimized within a certain range. For example, in numerical methods like Newton's one or gradient descent, the algorithm can help refine the search for the optimal solution.

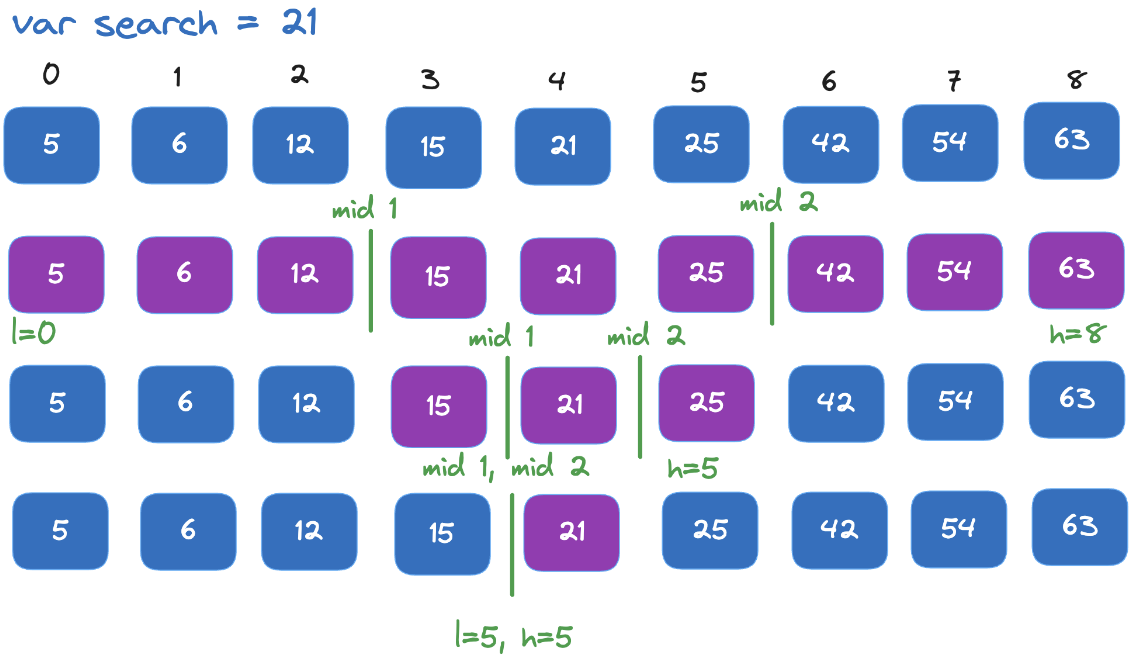
Searching in Sorted Arrays with Large Number of Comparisons: Ternary search can be advantageous in scenarios where the number of comparisons required for binary one becomes too high due to the nature of the data. The algorithm reduces the number of comparisons needed by approximately one-third in each iteration, potentially leading to faster search times.

Approximate Search: Ternary search can also be adapted for approximate tasks where finding an exact match is not necessary. For example, in databases or search engines, the algorithm can efficiently narrow down the search space for approximate matches, reducing the computational cost.

Finding Roots of Equations: The algorithm can be applied in numerical analysis to find roots of equations within a certain interval. Although methods like Newton's method are more commonly used for this purpose, ternary search can provide an alternative approach, especially when the function is not differentiable or its derivative is difficult to compute.

Time Complexity:

Ternary Search has a time complexity of O(log3 n), where 'n' is the size of the array. This is an improvement over binary search when the search space can be significantly reduced at each step. However, it's worth noting that constant factors play a role, and in practice, binary search might be faster for smaller datasets due to simpler arithmetic operations. Ternary Search is particularly beneficial when the dataset is large and the space can be significantly reduced with each iteration.



Example

```
function ternarySearch(
  func: (x: number) => number,
  left: number,
  right: number,
  epsilon: number
): number {
  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;
}
```