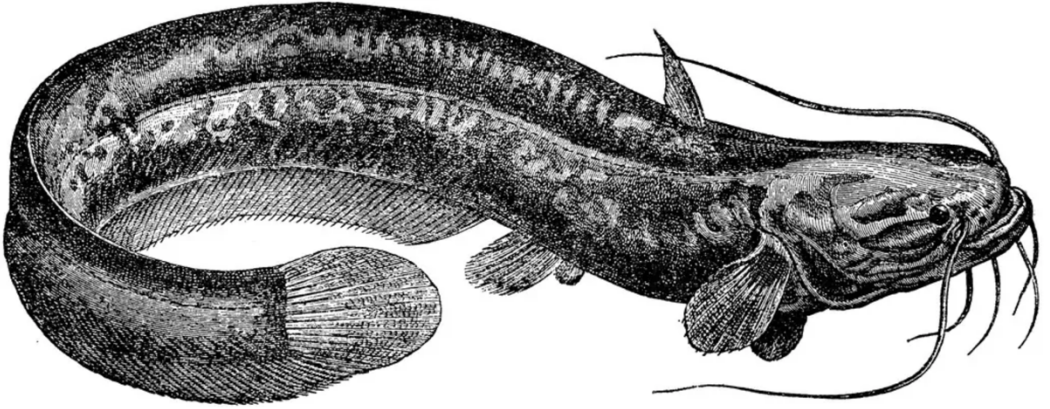


Discover the Code to Uncover Insights



Algorithms & Data Insight

Harnessing the Power of TypeScript for Efficient

Zidni Ridwan Nulmuarif

Table of Contents

<i>Introduction.....</i>	<i>8</i>
Overview of Algorithms and Data Structures	8
Importance of Algorithms and Data Structures	11
How to Use This Book	12
Introduction to TypeScript	13
<i>Getting Started with TypeScript.....</i>	<i>14</i>
Setting Up the TypeScript Environment	15
TypeScript Basics	18
Type Safety and Benefits	46
Enhanced Code Readability and Maintainability...	46
Early Detection of Type-Related Errors.....	46
Better IDE and Tooling Support	47
Improved Code Navigation and Refactoring	47
Type Annotations and Interfaces.....	48
Classes and Objects in TypeScript.....	48
Generics in TypeScript.....	49
<i>Mathematical Foundations</i>	<i>51</i>
Basic Mathematical Concepts	51
Big O Notation.....	54
Complexity Analysis	59
Recurrence relations	66
<i>Data Structures</i>	<i>71</i>
Arrays and Linked Lists.....	72
Arrays in TypeScript	72
Linked Lists (Singly, Doubly, Circular) in TypeScript	77

<i>Stacks and Queues</i>	83
Implementing Stacks in TypeScript	84
Implementing Queues in TypeScript.....	88
Deque in TypeScript.....	91
<i>Trees</i>	97
Binary Trees	99
Binary Search Trees.....	105
AVL Trees.....	111
Red-Black Trees	120
B-Trees	127
Heaps	135
<i>Hashing</i>	142
Hash Tables in TypeScript	142
Collision Resolution Techniques.....	148
<i>Graphs</i>	156
Graph Representations in TypeScript	157
Graph Traversal (BFS, DFS) in TypeScript.....	164
Weighted Graphs (Dijkstra's, Floyd-Warshall) in TypeScript.....	170
<i>Other Data Structures</i>	176
Tries in TypeScript	176
Disjoint Sets in TypeScript.....	182
Bloom Filters in TypeScript.....	188
<i>Algorithms</i>	195

Sorting and Searching Algorithms in TypeScript.....	195
Sorting Algorithms.....	195
Searching Algorithms.....	198
<i>Dynamic Programming in TypeScript</i>	201
Principles of Dynamic Programming.....	201
Common Dynamic Programming Problems in TypeScript.....	201
<i>Greedy Algorithms in TypeScript</i>	207
Principles of Greedy Algorithms	207
Common Greedy Algorithms in TypeScript	207
<i>Backtracking and Branch & Bound in TypeScript ..</i>	224
Principles of Backtracking	224
Common Backtracking Problems in TypeScript	225
Branch and Bound Techniques in TypeScript	229
Branch and Bound (B&B).....	229
<i>Advanced Data Structures in TypeScript.....</i>	233
Segment Trees	233
Basic Segment Tree in TypeScript.....	233
Lazy Propagation in TypeScript.....	236
Fenwick Trees (Binary Indexed Trees).....	240
Structure and Applications in TypeScript	240
Suffix Trees and Arrays	241
Suffix Trees in TypeScript	241
K-D Trees.....	243
Structure and Applications in TypeScript	243
<i>Graph Algorithms in TypeScript</i>	245

Minimum Spanning Trees.....	245
Kruskal's Algorithm in TypeScript.....	245
Prim's Algorithm in TypeScript.....	248
<i>Shortest Path Algorithms in TypeScript.....</i>	255
Dijkstra's Algorithm in TypeScript	255
Bellman-Ford Algorithm in TypeScript	257
Floyd-Warshall Algorithm in TypeScript.....	259
Comparison of Algorithms	261
<i>Network Flow in TypeScript.....</i>	262
Ford-Fulkerson Method in TypeScript.....	262
Edmonds-Karp Algorithm in TypeScript.....	265
<i>Matching and Covering in TypeScript.....</i>	269
Bipartite Matching in TypeScript	269
Hungarian Algorithm in TypeScript.....	271
<i>String Algorithms in TypeScript.....</i>	276
Pattern Matching	276
Rabin-Karp Algorithm in TypeScript	279
Suffix Trees and Arrays	281
Suffix Trees	281
<i>Computational Geometry in TypeScript.....</i>	284
Basic Concepts.....	284
Points, Lines, and Planes.....	284
Polygons	284
Algorithms	285
Convex Hull in TypeScript	285
Line Segment Intersection in TypeScript.....	287

Closest Pair of Points in TypeScript	288
<i>Parallel Algorithms in TypeScript.....</i>	292
Introduction to Parallel Computing.....	292
Models of Parallel Computation	292
Parallel Algorithm Techniques	292
Parallel Sorting in TypeScript.....	292
Parallel Graph Algorithms in TypeScript.....	295
<i>Approximation Algorithms in TypeScript.....</i>	298
Introduction to Approximation Algorithms	298
Common Techniques and Applications in TypeScript	298
<i>Randomized Algorithms in TypeScript</i>	305
Introduction to Randomized Algorithms.....	305
Examples and Applications in TypeScript.....	306
<i>Complexity Theory.....</i>	312
Introduction to Complexity Classes	312
P, NP, NP-Complete, and NP-Hard.....	313
Reductions and Completeness.....	313
<i>Practical Considerations for Algorithms and Data</i> <i>Structures in TypeScript</i>	318
Implementation Tips.....	318
Debugging and Testing in TypeScript.....	319
Performance Tuning in TypeScript.....	320
<i>Appendices.....</i>	324
Mathematical Notations	324

Common Typescript Cheat Sheet.....	325
Common Algorithms Cheat Sheet.....	328
TypeScript Libraries for Data Structures and Algorithms	331
<i>References</i>	<i>335</i>
<i>About the Writer.....</i>	<i>337</i>
<i>Index.....</i>	<i>338</i>

Introduction

Overview of Algorithms and Data Structures

What is an Algorithm?

An algorithm is a well-defined set of instructions designed to perform a specific task or solve a particular problem. Algorithms are the essence of computer programming, providing a systematic approach to problem-solving. They take an input, process it through a series of steps, and produce an output.

History of Algorithms

The concept of algorithms dates back to ancient times:

- **Ancient Algorithms:** Early examples include the Euclidean algorithm for computing the greatest common divisor, developed around 300 BC.
- **Medieval Contributions:** Persian mathematician Al-Khwarizmi's works in the 9th century contributed to the development of algebra and algorithms, and his name gave rise to the term "algorithm."
- **Modern Era:** The 20th century saw significant advancements with the development of computer science. Alan Turing's theoretical work laid the foundation for modern computer algorithms.

Key Characteristics of Algorithms

- **Finiteness:** An algorithm must terminate after a finite number of steps.
- **Definiteness:** Each step of an algorithm must be precisely defined and unambiguous.
- **Input:** An algorithm has zero or more inputs.

- **Output:** An algorithm produces one or more outputs.
- **Effectiveness:** The steps of an algorithm are basic enough to be performed, in principle, by a person using pencil and paper.

Types of Algorithms

- **Sorting Algorithms:** Arrange data in a particular order (e.g., Bubble Sort, Merge Sort).
- **Searching Algorithms:** Find specific data within a structure (e.g., Linear Search, Binary Search).
- **Graph Algorithms:** Solve problems related to graphs (e.g., Depth-First Search, Dijkstra's Algorithm).
- **Dynamic Programming:** Solve complex problems by breaking them down into simpler subproblems (e.g., Fibonacci sequence).

What is a Data Structure?

A data structure is a specialized format for organizing, processing, retrieving, and storing data. Efficient data structures are key to designing efficient algorithms.

Types of Data Structures

- **Primitive Data Structures:** Basic structures like integers, floats, characters, and pointers.
- **Non-Primitive Data Structures:** More complex structures such as arrays, linked lists, stacks, queues, trees, graphs, and hash tables.

Importance of Data Structures

- **Efficient Data Management:** Organize data in a way that enhances processing efficiency.

- **Resource Optimization:** Reduce the computational complexity of operations like search, insert, and delete.
- **Data Abstraction:** Provide a way to model real-world entities and relationships.

Summary

Understanding the basics of algorithms and data structures is crucial for any software developer. This section has introduced the fundamental concepts, historical context, and importance of algorithms and data structures in computing. The following sections of this book will delve deeper into specific types of algorithms and data structures, demonstrating their implementation and applications using TypeScript.

Importance of Algorithms and Data Structures

Algorithms and data structures are crucial for developing efficient and effective software. Here are some key reasons for their importance:

- **Performance Optimization:** Proper algorithms and data structures can significantly enhance the speed and efficiency of software applications.
- **Resource Management:** They help in optimizing the use of resources such as memory and processing power.
- **Problem Solving:** Algorithms provide systematic methods for solving complex problems.
- **Scalability:** Good data structures ensure that applications can handle large volumes of data and user interactions smoothly.

Real-World Applications

Algorithms and data structures are used in various domains:

- **Web Development:** Managing data efficiently, handling user requests, and optimizing performance.
- **Data Analysis:** Processing large datasets, performing statistical analysis, and extracting insights.
- **Machine Learning and AI:** Training models, making predictions, and improving accuracy.
- **Game Development:** Managing game state, rendering graphics, and handling user inputs.
- **Database Management:** Efficient data retrieval, storage, and updating.

How to Use This Book

This book is structured to progressively build your understanding of algorithms and data structures with practical examples in TypeScript.

- **Step-by-Step Learning:** Each chapter introduces concepts gradually, starting from basics to more advanced topics.
- **Hands-On Examples:** Practical examples and TypeScript code snippets are provided to reinforce learning.
- **Exercises and Solutions:** End-of-chapter exercises help practice and solidify the concepts learned.
- **Reference Material:** Appendices and reference sections provide additional resources for further study.

Introduction to TypeScript

Getting Started with TypeScript

TypeScript is a statically typed superset of JavaScript that adds optional type annotations. It helps in writing more robust and maintainable code.

- **Setting Up the TypeScript Environment:** Install TypeScript and configure your development environment.
- **TypeScript Basics:** Understand the syntax and basic constructs of TypeScript.
- **Type Annotations and Interfaces:** Learn how to define types and interfaces for better code quality and readability.
- **Classes and Objects in TypeScript:** Explore object-oriented programming concepts in TypeScript.
- **Generics in TypeScript:** Use generics to create reusable and flexible components.

This section provides a foundational understanding of TypeScript, which will be used throughout the book to implement various algorithms and data structures.

Getting Started with TypeScript

History of TypeScript

Learn about the origins and evolution of TypeScript:

- Created by Microsoft as an open-source programming language.
- First released in October 2012.
- TypeScript's relationship with JavaScript and ECMAScript standards.

Functions in TypeScript

Explore the function syntax and features in TypeScript:

- Declaring functions with type annotations.
- Optional and default parameters.
- Arrow functions and this context handling.

Pros and Cons of TypeScript

Evaluate the benefits and drawbacks of using TypeScript:

Pros:

- **Type Safety:** Enhanced code quality and reliability with static typing.
- **Tooling Support:** Rich IDE features and error-checking tools.
- **ECMAScript Compatibility:** Supports modern JavaScript features.
- **Scalability:** Suitable for large-scale applications and team collaboration.

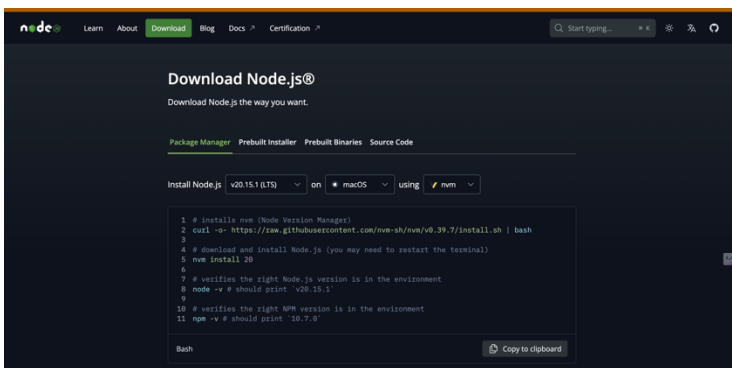
Cons:

- **Learning Curve:** Requires understanding of TypeScript-specific concepts.

- **Compilation Overhead:** Additional build step for type checking and compilation.
- **Interoperability:** Integration challenges with some JavaScript libraries.

Setting Up the TypeScript Environment

Installing TypeScript



1. **Node.js Installation:** Ensure Node.js is installed on your system. TypeScript requires Node.js and npm (Node Package Manager):
<https://nodejs.org/en/download/package-manager>
2. **Installing TypeScript:** Open your terminal and install TypeScript globally using npm:

```
npm install -g typescript
```

Alternatively, you can install TypeScript locally in your project:

```
npm install typescript --save-dev
```

Configuring TypeScript

1. **Creating tsconfig.json:** TypeScript uses a configuration file to manage compiler options. Generate a tsconfig.json file in your project directory:

```
tsc -init
```

This command initializes a basic tsconfig.json file with default settings.

2. **Customizing tsconfig.json:** Modify tsconfig.json to suit your project needs. Configure options like target ECMAScript version (target), module system (module), and output directory (outDir).

Integrating with IDE or Text Editor

1. **IDE Support:** TypeScript integrates seamlessly with popular IDEs like Visual Studio Code, WebStorm, and others. Install necessary TypeScript plugins or extensions for enhanced TypeScript support.
2. **TypeScript Compilation:** Configure your IDE or text editor to use the TypeScript compiler (tsc) for automatic compilation. Ensure TypeScript files (*.ts or *.tsx) are recognized and compiled on save.

Verifying Installation

1. **Compile TypeScript:** Create a sample TypeScript file (e.g., app.ts) and compile it using tsc:

```
tsc app.ts
```

This command generates a corresponding JavaScript file (app.js) based on your TypeScript code.

2. **Run TypeScript:** Execute the compiled JavaScript code using Node.js:

```
node app.js
```

Verify that your TypeScript setup is working correctly without errors.

Alternative

Alternatively, you can use an online TypeScript IDE to start this book. You can visit:

<https://www.typescriptlang.org/play/>

TypeScript Basics

TypeScript Basics (Variables)

Declaring Variables

- Using **let** and **const** for variable declarations:

```
let num: number = 10;  
const PI: number = 3.14;
```

- let allows reassignment of values.
- const declares constants that cannot be reassigned.

Basic Types

- TypeScript supports various basic types:

TypeScript

```
let name: string = "Alice";  
let age: number = 30;  
let isActive: boolean = true;
```

- string: Represents textual data.
- number: Represents numerical data.
- boolean: Represents true/false values.

Type Inference

- TypeScript infers types based on initial values:

```
let city = "New York"; // TypeScript infers  
'string' type
```

```
let population = 8_500_000; // TypeScript
infers 'number' type
let isCapital = true; // TypeScript infers
'boolean' type
```

Type Annotations

Explicitly Defining Types

- Explicitly annotate variables with their types:

```
let username: string = "john_doe";
let age: number = 30;
let isActive: boolean = true;
```

Union Types

- Use union types for variables that can have multiple types:

```
let id: string | number = "user-123";
id = 123; // Valid assignment
```

Type Assertion

- Assert the type of a variable when its type cannot be inferred automatically:

```
let input: any = "123";
let length: number = (input as string).length;
```

Constants and Readonly

Constants with const

- Declare constants using const:

```
const PI: number = 3.14;
```

```
const WEBSITE_URL: string =  
"https://example.com";
```

Readonly Properties

- Use readonly to mark properties as immutable:

```
interface Point {  
    readonly x: number;  
    readonly y: number;  
}
```

```
let p: Point = { x: 10, y: 20 };  
// p.x = 5; // Error: Cannot assign to 'x'  
because it is a read-only property.
```

TypeScript Basics (Functions)

Introduction to Functions in TypeScript

Declaring Functions

- Define functions with type annotations for parameters and return types:

```
function greet(name: string): string {  
    return `Hello, ${name}!`;   
}
```

```
let message: string = greet("Alice");  
console.log(message); // Output: Hello, Alice!
```

Optional and Default Parameters

- Use optional parameters by appending ?:

```
function greet(name: string, greeting?:  
string): string {  
    if (greeting) {  
        return `${greeting}, ${name}!`;   
    } else {  
        return `Hello, ${name}!`;   
    }  
}
```

```
let message1: string = greet("Bob");  
let message2: string = greet("Charlie", "Good  
morning");
```

- Provide default values for parameters:

```
function greet(name: string, greeting: string
= "Hello"): string {
    return `${greeting}, ${name}!`;
}
```

```
let message1: string = greet("David");
let message2: string = greet("Emily", "Hi");
```

Arrow Functions

Syntax and Usage

- Use arrow functions for concise function expressions:

```
let add = (x: number, y: number): number => {
    return x + y;
};
```

```
let result: number = add(3, 5); // Output: 8
```

Lexical this Binding

- Arrow functions preserve the lexical this context:

```
let person = {
    name: "Alice",
    greet: function() {
        setTimeout(() => {
            console.log(`Hello,
${this.name}!`);
        }, 1000);
    }
};
```

```
person.greet(); // Output after 1 second:  
Hello, Alice!
```


Function Overloading

Multiple Function Signatures

- Define multiple function signatures with the same name but different parameter types or counts:

```
function process(x: number): void;
function process(x: string): void;
function process(x: any): void {
    console.log(x);
}
```

```
process(123); // Output: 123
process("Hello"); // Output: Hello
```

Type Guards

Narrowing Down Types

- Use type guards to conditionally check types within functions:

```
function display(value: string | number) {
    if (typeof value === 'string') {
        console.log(`String value: ${value}`);
    } else {
        console.log(`Number value: ${value}`);
    }
}
```

```
display("TypeScript"); // Output: String
                        value: TypeScript
display(123); // Output: Number value: 123
```

Higher-Order Functions

Functions as Parameters

- Pass functions as arguments to other functions:

```
function operate(x: number, y: number,  
operation: (a: number, b: number) => number):  
number {  
    return operation(x, y);  
}
```

```
let result1: number = operate(3, 4, (a, b) =>  
a + b); // Addition  
let result2: number = operate(5, 2, (a, b) =>  
a * b); // Multiplication
```

TypeScript Basics (Operators)

Introduction to Operators in TypeScript

Arithmetic Operators

- Perform basic arithmetic operations:

```
let num1: number = 10;
let num2: number = 5;

let sum: number = num1 + num2; // Addition
let difference: number = num1 - num2; // Subtraction
let product: number = num1 * num2; // Multiplication
let quotient: number = num1 / num2; // Division
let remainder: number = num1 % num2; // Modulus
```

Comparison Operators

- Compare values and produce a Boolean result:

```
let a: number = 10;
let b: number = 5;

let isEqual: boolean = a === b; // Equal to
let isNotEqual: boolean = a !== b; // Not equal to
let isGreater: boolean = a > b; // Greater than
let isLess: boolean = a < b; // Less than
let isGreaterOrEqual: boolean = a >= b; // Greater than or equal to
let isLessOrEqual: boolean = a <= b; // Less than or equal to
```

Logical Operators

- Combine Boolean expressions:

```
let x: boolean = true;
let y: boolean = false;

let result1: boolean = x && y; // Logical AND
let result2: boolean = x || y; // Logical OR
let result3: boolean = !x; // Logical NOT
```

Assignment Operators

- Assign values and perform operations in a concise manner:

```
let value: number = 10;

value += 5; // Equivalent to: value = value + 5;
value -= 3; // Equivalent to: value = value - 3;
value *= 2; // Equivalent to: value = value * 2;
value /= 4; // Equivalent to: value = value / 4;
value %= 3; // Equivalent to: value = value % 3;
```

Bitwise Operators

- Perform bitwise operations on numeric values:

```
let num1: number = 5; // 101 in binary
let num2: number = 3; // 011 in binary
```

```
let bitwiseAnd: number = num1 & num2; //
Bitwise AND (001)
let bitwiseOr: number = num1 | num2; // Bitwise
OR (111)
let bitwiseXor: number = num1 ^ num2; //
Bitwise XOR (110)
let bitwiseNot: number = ~num1; // Bitwise NOT
(-(num1 + 1))
let bitwiseLeftShift: number = num1 << 1; //
Bitwise left shift (1010)
let bitwiseRightShift: number = num1 >> 1; //
Bitwise right shift (10)
```

String Operators

- Concatenate strings using + operator:

```
let firstName: string = "John";
let lastName: string = "Doe";

let fullName: string = firstName + " " +
lastName; // "John Doe"
```

TypeScript Basics (Control Flow)

Introduction to Control Flow in TypeScript

Conditional Statements

if...else Statement

- Execute code based on a condition:

```
let num: number = 10;

if (num > 0) {
    console.log("Positive number");
} else if (num < 0) {
    console.log("Negative number");
} else {
    console.log("Zero");
}
```

switch Statement

- Execute different actions based on different conditions:

```
let day: number = 3;
let dayName: string;

switch (day) {
    case 1:
        dayName = "Monday";
        break;
    case 2:
        dayName = "Tuesday";
        break;
    case 3:
        dayName = "Wednesday";
```

```
        break;
    default:
        dayName = "Unknown";
}

console.log(`Today is ${dayName}`);
```

Looping Constructs

for Loop

- Iterate over a range of values:

```
for (let i = 0; i < 5; i++) {
    console.log(i);
}
```

while Loop

- Execute a block of code as long as a condition is true:

```
let i: number = 0;

while (i < 5) {
    console.log(i);
    i++;
}
```

do...while Loop

- Similar to while loop but executes at least once:

```
let i: number = 0;
do {
    console.log(i);
    i++;
}
```

```
} while (i < 5);
```

for...of Loop (Iterating Arrays)

- Iterate over elements of an array:

```
let colors: string[] = ["red", "green",  
"blue"];
```

```
for (let color of colors) {  
  console.log(color);  
}
```

for...in Loop (Iterating Objects)

- Iterate over enumerable properties of an object:

```
let person = {  
  name: "Alice",  
  age: 30,  
  city: "New York"  
};
```

```
for (let key in person) {  
  console.log(`${key}: ${person[key]}`);  
}
```

Control Flow Statements

break and continue

- Use break to terminate loop execution:

```
for (let i = 0; i < 5; i++) {  
  if (i === 3) {  
    break;  
  }  
}
```



```
    console.log(i); // Output: 0, 1, 2
}
```

Use `continue` to skip current iteration and continue with the next:

typescript

Copy code

```
for (let i = 0; i < 5; i++) {
    if (i === 2) {
        continue;
    }
    console.log(i); // Output: 0, 1, 3, 4
}
```

TypeScript Basics (Classes)

Introduction to Classes in TypeScript

Creating Classes

- Define classes to encapsulate data and behavior:

```
class Person {  
    // Properties  
    name: string;  
    age: number;  
  
    // Constructor  
    constructor(name: string, age: number) {  
        this.name = name;  
        this.age = age;  
    }  
  
    // Method  
    greet() {  
        return `Hello, my name is ${this.name}  
and I am ${this.age} years old.`;  
    }  
}
```

```
// Create an instance of Person  
let person1 = new Person("Alice", 30);  
console.log(person1.greet());    // Output:  
Hello, my name is Alice and I am 30 years old.
```

Constructors and Initialization

Constructor Method

- Initialize object properties when creating instances:

typescript

Copy code

```
class Car {  
    // Properties  
    brand: string;  
    model: string;  
  
    // Constructor  
    constructor(brand: string, model: string)  
{  
        this.brand = brand;  
        this.model = model;  
    }  
}  
  
// Create instances of Car  
let car1 = new Car("Toyota", "Camry");  
let car2 = new Car("Honda", "Accord");
```

Access Modifiers

Public, Private, and Protected

- Control access to class members:

```
class Employee {  
    // Public property  
    public name: string;  
  
    // Private property  
    private salary: number;  
  
    // Protected property
```

```

    protected department: string;

    // Constructor
    constructor(name: string, salary: number,
department: string) {
        this.name = name;
        this.salary = salary;
        this.department = department;
    }

    // Method
    getDetails() {
        return `${this.name} works in
${this.department} and earns
${this.salary}.`;
    }
}

// Create an instance of Employee
let emp = new Employee("Bob", 50000,
"Engineering");
console.log(emp.name); // Accessible: "Bob"
// console.log(emp.salary); // Error: Property
'salary' is private and only accessible within
class 'Employee'.
// console.log(emp.department); // Error:
Property 'department' is protected and only
accessible within class 'Employee' and its
subclasses.
console.log(emp.getDetails()); // Output: Bob
works in Engineering and earns $50000.

```

Inheritance

Extending Classes

- Create hierarchical relationships between classes:

```
// Parent class
class Animal {
  name: string;

  constructor(name: string) {
    this.name = name;
  }

  makeSound() {
    console.log(`${this.name} makes a
sound.`);
  }
}

// Child class inheriting from Animal
class Dog extends Animal {
  constructor(name: string) {
    super(name); // Call parent
constructor
  }

  makeSound() {
    console.log(`${this.name} barks.`);
  }
}

// Create instances of Animal and Dog
let animal = new Animal("Generic Animal");
let dog = new Dog("Buddy");
```

```
animal.makeSound(); // Output: Generic Animal
                        makes a sound.
dog.makeSound(); // Output: Buddy barks.
```

Method Overriding

Customizing Behavior

- Override methods in child classes to provide specific implementations:

```
// Parent class
class Shape {
    area(): number {
        return 0;
    }
}

// Child class extending from Shape
class Circle extends Shape {
    radius: number;

    constructor(radius: number) {
        super();
        this.radius = radius;
    }

    area(): number {
        return Math.PI * this.radius *
this.radius;
    }
}

// Create instances of Circle
let circle = new Circle(5);
```

```
console.log(circle.area()); // Output: ~78.54
```

TypeScript Basics (Interfaces and Types)

Introduction to Interfaces

Defining Interfaces

- Use interfaces to define the shape of an object:

```
interface Person {  
    name: string;  
    age: number;  
    greet(): string;  
}
```

```
let person: Person = {  
    name: "Alice",  
    age: 30,  
    greet() {  
        return `Hello, my name is  
${this.name}.`;   
    }  
};
```

```
console.log(person.greet()); // Output: Hello,  
my name is Alice.
```

Optional Properties

- Use ? to denote optional properties:

```
interface Car {  
    brand: string;  
    model: string;  
    year?: number;  
}
```



```
let myCar: Car = {  
    brand: "Toyota",  
    model: "Camry"  
};
```

Readonly Properties

- Use readonly to make properties immutable:

```
interface Book {  
    readonly title: string;  
    author: string;  
}
```

```
let myBook: Book = {  
    title: "TypeScript Basics",  
    author: "John Doe"  
};
```

// myBook.title = "New Title"; // Error: Cannot assign to 'title' because it is a read-only property.

Introduction to Types

Defining Type Aliases

- Use type to create type aliases:

```
type ID = number | string;
```

```
let userId: ID;  
userId = 123; // Valid  
userId = "ABC123"; // Valid
```

Union Types

- Combine multiple types using the | operator:

```
type Status = "success" | "failure" | "pending";
```

```
let requestStatus: Status;  
requestStatus = "success"; // Valid  
requestStatus = "error"; // Error: Type  
'"error"' is not assignable to type 'Status'.
```

Interfaces vs. Types

Differences and Use Cases

- Interfaces are more suited for defining object shapes, whereas types can represent various kinds of types:

```
interface Animal {  
    name: string;  
}
```

```
interface Dog extends Animal {  
    breed: string;  
}
```

```
type Pet = Dog | Cat;
```

```
interface Cat {  
    name: string;  
    livesLeft: number;  
}
```

```
let myPet: Pet = {  
    name: "Whiskers",
```

```
    livesLeft: 9  
};
```

Intersection Types

- Combine multiple types using the & operator:

```
interface Colorful {  
    color: string;  
}
```

```
interface Circle {  
    radius: number;  
}
```

```
type ColorfulCircle = Colorful & Circle;
```

```
let myCircle: ColorfulCircle = {  
    color: "red",  
    radius: 10  
};
```

Extending Interfaces

Extending Existing Interfaces

- Use extends to create a new interface that inherits from an existing one:

```
interface Shape {  
    color: string;  
}
```

```
interface Square extends Shape {  
    sideLength: number;  
}
```

```
let mySquare: Square = {  
  color: "blue",  
  sideLength: 5  
};
```

Implementing Interfaces in Classes

Using Interfaces with Classes

- Ensure classes adhere to specific contracts by implementing interfaces:

```
interface ClockInterface {  
  currentTime: Date;  
  setTime(d: Date): void;  
}
```

```
class Clock implements ClockInterface {  
  currentTime: Date;  
  
  constructor(h: number, m: number) {  
    this.currentTime = new Date();  
  }  
  
  setTime(d: Date) {  
    this.currentTime = d;  
  }  
}
```

Generic Interfaces and Types

Creating Generic Interfaces

- Use generics to create flexible and reusable interfaces:

```
interface Box<T> {  
    contents: T;  
}
```

```
let stringBox: Box<string> = { contents:  
"Hello" };  
let numberBox: Box<number> = { contents: 42 };
```

Creating Generic Type Aliases

- Use generics with type aliases:

```
type Pair<T, U> = [T, U];
```

```
let stringNumberPair: Pair<string, number> =  
["hello", 42];  
let booleanBooleanPair: Pair<boolean, boolean>  
= [true, false];
```

Type Safety and Benefits

Introduction to Type Safety

TypeScript provides static type checking at compile time, which helps developers catch errors early in the development process. By specifying types for variables, function parameters, and return values, TypeScript ensures that the values being used in the code conform to the expected types.

Benefits of Typed Variables

Enhanced Code Readability and Maintainability

- Typed variables make it clear what type of data is expected, making the code more understandable for other developers (or yourself at a later time).
- For example, defining a variable with a specific type:

```
let userName: string = "Alice";
```

Early Detection of Type-Related Errors

- TypeScript's static type checking catches errors at compile time before the code runs, reducing runtime errors.
- For example, assigning a number to a string variable will result in a compile-time error:

```
let userAge: number = 25;  
userAge = "twenty-five"; // Error: Type 'string'  
is not assignable to type 'number'.
```

Better IDE and Tooling Support

- TypeScript provides powerful IntelliSense features, including code completion, navigation, and refactoring, making the development process smoother and more efficient.
- For example, hovering over a typed variable in an IDE shows its type and documentation:

```
function greet(name: string): string {  
    return `Hello, ${name}!`;  
}
```

Improved Code Navigation and Refactoring

- With explicit types, IDEs can more accurately trace where variables and functions are used, making it easier to refactor code safely.
- For example, renaming a function in TypeScript automatically updates all its references:

```
function calculateTotal(price: number,  
    quantity: number): number {  
    return price * quantity;  
}
```

```
let total = calculateTotal(100, 2); // Renaming  
'calculateTotal' will update this reference.
```

Type Annotations and Interfaces

Type annotations allow you to specify the types of variables, function parameters, and return values. Interfaces define the shape of objects and can be used to enforce type constraints.

- **Type Annotations:**

```
let isActive: boolean = true;
let total: number = 100;
let userName: string = "Alice";
```

- **Interfaces:**

```
interface User {
  id: number;
  name: string;
  email: string;
}
```

```
let user: User = {
  id: 1,
  name: "Alice",
  email: "alice@example.com"
};
```

Classes and Objects in TypeScript

Classes provide a way to create objects with properties and methods. TypeScript enhances classes with type safety.

- **Defining a Class:**


```

class Person {
  name: string;
  age: number;

  constructor(name: string, age: number) {
    this.name = name;
    this.age = age;
  }

  greet(): string {
    return `Hello, my name is
${this.name}.`;
  }
}

```

```

let person = new Person("Alice", 30);
console.log(person.greet()); // Output: Hello,
my name is Alice.

```

Generics in TypeScript

Generics allow you to create reusable components that can work with various types while maintaining type safety.

- **Using Generics:**

```

function identity<T>(arg: T): T {
  return arg;
}

let output1 = identity<string>("Hello");
let output2 = identity<number>(123);

```

Generics are particularly useful in functions and classes that need to operate on multiple types.

Mathematical Foundations

Understanding basic mathematical concepts is crucial for developing and analyzing algorithms. These concepts provide the foundation for more advanced topics in computer science.

Basic Mathematical Concepts

Understanding basic mathematical concepts is crucial for developing strong foundations in programming and computer science. These concepts provide the tools needed to design algorithms, analyze their efficiency, and solve complex problems systematically. This section covers sets, relations, functions, graphs, and trees.

Sets

A **set** is a collection of distinct objects, considered as an object in its own right. Sets are fundamental in mathematics and are used to define nearly all mathematical objects. Sets can be finite or infinite and can contain numbers, symbols, or even other sets. Basic operations on sets include:

- **Union:** Combines all elements of two sets.

Example: $\{1, 2, 3\} \cup \{3, 4, 5\} = \{1, 2, 3, 4, 5\}$

- **Intersection:** Elements common to both sets.

Example: $\{1, 2, 3\} \cap \{2, 3, 4\} = \{2, 3\}$

- **Difference:** Elements in one set but not the other.

Example: $\{1, 2, 3\} \setminus \{2, 3, 4\} = \{1\}$

- **Complement:** Elements not in the set relative to a universal set.

Relations

A **relation** is a connection between elements of two sets. It is a subset of the Cartesian product of two sets, which is the set of all ordered pairs where the first element is from the first set and the second element is from the second set. Important properties of relations include:

- **Reflexive:** Every element is related to itself.

Example: $\forall a \in A, (a, a) \in R$

- **Symmetric:** If an element is related to another, then the second element is related to the first.

Example: If $(a, b) \in R$, then $(b, a) \in R$

- **Transitive:** If an element is related to a second element, and the second element is related to a third, then the first is related to the third.

Example: If $(a, b) \in R$ and $(b, c) \in R$, then $(a, c) \in R$

Functions

A **function** is a specific type of relation where each element in the domain is associated with exactly one element in the codomain. Functions can be visualized as mappings from inputs to outputs. Important concepts include:

- **Injective (One-to-One):** Different inputs map to different outputs.

Example: $f(x) = 2x$ is injective

- **Surjective (Onto):** Every element in the codomain is mapped by some input.

Example: $f(x) = x^2$ is surjective when considering non-negative real numbers

- **Bijjective:** A function that is both injective and surjective, establishing a one-to-one correspondence between the domain and codomain.

-

Example: $f(x) = x + 1$ from integers to integers

Graphs and Trees

Graphs are collections of nodes (vertices) and edges (connections between nodes). They are used to model relationships and networks, such as social networks, transportation systems, and computer networks. Basic types of graphs include:

- **Directed Graphs:** Edges have a direction.

Example: Twitter following relationships

- **Undirected Graphs:** Edges have no direction.

Example: Facebook friendships

****Weighted Graphs**:** Edges have weights representing costs, distances, or capacities.

Example: Road networks with distances

Trees are a special type of graph with no cycles and a hierarchical structure. They are used to model hierarchical data, such as file systems and organizational structures. Important types of trees include:

- **Binary Trees:** Each node has at most two children.

- **Binary Search Trees:** A binary tree where the left child is less than the parent node, and the right child is greater.

Big O Notation

Big O Notation is a mathematical concept used to describe the performance and efficiency of algorithms, particularly their time complexity and space complexity. It provides an upper bound on the growth rate of an algorithm's runtime or memory usage as the input size increases, allowing for the comparison of different algorithms and their scalability.

Understanding Big O Notation

Big O Notation expresses the worst-case scenario of an algorithm, focusing on its asymptotic behavior as the input size (denoted as n) tends towards infinity. The notation simplifies the analysis by disregarding constant factors and lower-order terms, emphasizing the dominant term that significantly impacts performance as n grows large.

Common Big O Notations

Here are some of the most common Big O notations, along with examples and explanations:

$O(1)$: Constant Time

An algorithm is said to run in constant time if its execution time does not change regardless of the input size. These algorithms are highly efficient and preferred when possible.

```
function getFirstElement(arr: number[]):
number {
    return arr[0]; // Always takes the same
time
}
```

O(n): Linear Time

An algorithm runs in linear time if its execution time increases linearly with the input size. These algorithms are common in scenarios where each element must be processed at least once.

```
function linearSearch(arr: number[], target:
number): number {
    for (let i = 0; i < arr.length; i++) {
        if (arr[i] === target) {
            return i;
        }
    }
    return -1;
}
```

O(n²): Quadratic Time

An algorithm runs in quadratic time if its execution time increases quadratically with the input size. These algorithms are less efficient and can become impractical for large input sizes.

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

```

        [arr[j], arr[j + 1]] = [arr[j
+ 1], arr[j]];
    }
}
}
return arr;
}

```

$O(\log n)$: Logarithmic Time

An algorithm runs in logarithmic time if its execution time increases logarithmically with the input size. These algorithms are efficient and often used in divide-and-conquer strategies.

```

function binarySearch(arr: number[], target:
number): number {
    let left = 0;
    let right = arr.length - 1;
    while (left <= right) {
        const mid = Math.floor((left + right) /
2);
        if (arr[mid] === target) {
            return mid;
        } else if (arr[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    return -1;
}

```

$O(n \log n)$: Linearithmic Time

An algorithm runs in linearithmic time if its execution time increases in proportion to $n \log n$. These algorithms are common in efficient sorting algorithms like merge sort and quicksort.

```
function mergeSort(arr: number[]): number[] {
  if (arr.length <= 1) {
    return arr;
  }
  const mid = Math.floor(arr.length / 2);
  const left = mergeSort(arr.slice(0, mid));
  const right = mergeSort(arr.slice(mid));
  return merge(left, right);
}
```

```
function merge(left: number[], right: number[]): number[] {
  const result = [];
  while (left.length && right.length) {
    if (left[0] < right[0]) {
      result.push(left.shift());
    } else {
      result.push(right.shift());
    }
  }
  return result.concat(left).concat(right);
}
```

Why Big O Notation Matters

Big O Notation is a critical tool for computer scientists and developers because it helps to:

- **Evaluate Algorithm Efficiency:** By understanding the growth rate of an algorithm,

developers can choose the most appropriate algorithm for a given problem, especially when dealing with large datasets.

- **Optimize Code:** Identifying bottlenecks and inefficiencies in algorithms allows for code optimization, improving overall performance.
- **Predict Scalability:** Understanding how an algorithm's performance scales with increasing input size helps in anticipating and managing resource requirements in real-world applications.

Comparing Algorithms

When comparing algorithms, Big O Notation allows for a clear and objective analysis of their efficiency. For example, consider the problem of sorting an array. A bubble sort algorithm has a time complexity of $O(n^2)$, making it less suitable for large arrays compared to merge sort, which has a time complexity of $O(n \log n)$.

By focusing on the dominant term in the Big O Notation, developers can make informed decisions about which algorithm to use based on the expected input size and performance requirements.

In conclusion, Big O Notation is an essential concept in computer science, providing a standardized way to measure and compare the efficiency of algorithms. Mastering Big O Notation enables developers to write more efficient and scalable code, ultimately leading to better software performance and user experience.

Complexity Analysis

Complexity analysis is the study of how the resource requirements of an algorithm or a piece of code grow as the size of the input increases. It helps in understanding the efficiency and scalability of algorithms, which is crucial for designing systems that perform well with large datasets and complex operations. There are two primary types of complexities to analyze: time complexity and space complexity.

Time Complexity

Time complexity measures the amount of time an algorithm takes to complete as a function of the length of the input. It is usually expressed using Big O notation, which provides an upper bound on the growth rate of the running time. The goal is to determine how the runtime grows with the input size.

Types of Time Complexity

1. Constant Time - $O(1)$

- The running time of the algorithm is constant and does not change with the input size.
- Example:

```
function isFirstElementEven(arr: number[]):  
boolean {  
    return arr[0] % 2 === 0;  
}
```

2. Logarithmic Time - $O(\log n)$

- The running time grows logarithmically with the input size.

- Common in algorithms that repeatedly divide the problem size in half.
- Example:
-

```
function binarySearch(arr: number[], target:
number): number {
  let left = 0;
  let right = arr.length - 1;
  while (left <= right) {
    const mid = Math.floor((left + right) /
2);
    if (arr[mid] === target) {
      return mid;
    } else if (arr[mid] < target) {
      left = mid + 1;
    } else {
      right = mid - 1;
    }
  }
  return -1;
}
```

3. Linear Time - $O(n)$

- The running time increases linearly with the input size.
- Example:

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

```
    return -1;
}
```

4. Linearithmic Time - $O(n \log n)$

- The running time grows in proportion to $n \log n$.
- Common in efficient sorting algorithms like merge sort and quicksort.
- Example:

```
function mergeSort(arr: number[]): number[] {
    if (arr.length <= 1) {
        return arr;
    }
    const mid = Math.floor(arr.length / 2);
    const left = mergeSort(arr.slice(0, mid));
    const right = mergeSort(arr.slice(mid));
    return merge(left, right);
}
```

```
function merge(left: number[], right:
number[]): number[] {
    const result = [];
    while (left.length && right.length) {
        if (left[0] < right[0]) {
            result.push(left.shift());
        } else {
            result.push(right.shift());
        }
    }
    return result.concat(left).concat(right);
}
```

5. Quadratic Time - $O(n^2)$

- The running time grows quadratically with the input size.
- Common in algorithms with nested loops.
- Example:

```
function bubbleSort(arr: number[]): number[] {  
  for (let i = 0; i < arr.length; i++) {  
    for (let j = 0; j < arr.length - i - 1;  
j++) {  
      if (arr[j] > arr[j + 1]) {  
        [arr[j], arr[j + 1]] = [arr[j]  
+ 1], arr[j]];  
      }  
    }  
  }  
  return arr;  
}
```

6. Exponential Time - $O(2^n)$

- The running time doubles with each additional element in the input.
- Common in algorithms that solve problems by brute force.
- Example:

```
function fibonacci(n: number): number {  
  if (n <= 1) {  
    return n;  
  }  
  return fibonacci(n - 1) + fibonacci(n - 2);  
}
```

Space Complexity

Space complexity measures the amount of memory an algorithm uses as a function of the length of the input. Like time complexity, it is also expressed using Big O notation. Space complexity includes both the space needed to hold the input data and the space needed for any additional variables and data structures used by the algorithm.

Types of Space Complexity

1. Constant Space - $O(1)$

- The algorithm uses a fixed amount of memory, regardless of the input size.
- Example:
-

```
function swap(a: number, b: number): void {  
    let temp = a;  
    a = b;  
    b = temp;  
}
```

2. Linear Space - $O(n)$

- The amount of memory used grows linearly with the input size.
- Example:

```
function createArray(n: number): number[] {  
    const arr = new Array(n);  
    for (let i = 0; i < n; i++) {  
        arr[i] = i;  
    }  
    return arr;  
}
```

3. Quadratic Space - $O(n^2)$

- The amount of memory used grows quadratically with the input size.
- Example:

```
function createMatrix(n: number): number[][] {  
  const matrix = new Array(n);  
  for (let i = 0; i < n; i++) {  
    matrix[i] = new Array(n);  
    for (let j = 0; j < n; j++) {  
      matrix[i][j] = i * j;  
    }  
  }  
  return matrix;  
}
```

Importance of Complexity Analysis

1. **Efficiency:** Understanding the complexity of algorithms helps in choosing the most efficient one for a given problem, particularly when dealing with large inputs.
2. **Scalability:** By analyzing complexity, developers can predict how an algorithm will perform as the input size grows, ensuring the system remains scalable.
3. **Optimization:** Complexity analysis helps in identifying bottlenecks and potential areas for optimization, leading to faster and more efficient code.
4. **Resource Management:** By understanding both time and space complexity, developers can make informed decisions about trade-offs between memory usage and processing speed, optimizing resource utilization.

Conclusion

Complexity analysis is a fundamental aspect of algorithm design and evaluation. It provides a framework for understanding and comparing the efficiency of different algorithms, enabling developers to make informed choices that lead to optimal performance. Mastering complexity analysis is essential for anyone looking to develop efficient, scalable, and high-performing software solutions.

Recurrence relations

Recurrence relations are equations that define sequences or multi-step processes in terms of previous steps or terms. They are commonly used in computer science to analyze the performance of recursive algorithms by providing a mathematical framework for describing the time complexity of an algorithm in terms of its input size.

Definition

A recurrence relation expresses the n -th term of a sequence as a function of its preceding terms. In the context of algorithms, it typically represents the running time of a recursive function.

Types of Recurrence Relations

There are various forms of recurrence relations, including:

1. Linear Recurrence Relations:

- The n -th term is a linear combination of previous terms.
- Example: $T(n) = aT(n-1) + bT(n) = aT(n-1) + bT(n) = aT(n-1) + b$, where a and b are constants.

2. Non-linear Recurrence Relations:

- The n -th term is a non-linear function of previous terms.
- Example: $T(n) = T(n-1)^2 + nT(n) = T(n-1)^2 + nT(n) = T(n-1)^2 + n$.

3. Homogeneous Recurrence Relations:

- No additional terms are added to the function.

- Example: $T(n) = aT(n-1)$ $T(n) = aT(n-1)T(n) = aT(n-1)T(n) = aT(n-1)$.

4. **Non-homogeneous Recurrence Relations:**

- Additional terms are added to the function.
- Example: $T(n) = aT(n-1) + f(n)$ $T(n) = aT(n-1) + f(n)T(n) = aT(n-1) + f(n)T(n) = aT(n-1) + f(n)$, where $f(n)$ is a function of n .

Solving Recurrence Relations

Solving a recurrence relation means finding a closed-form solution that does not involve recursion. There are several methods for solving recurrence relations:

1. **Substitution Method:**

- Guess the form of the solution and use mathematical induction to verify it.
- Example: For $T(n) = 2T(n/2) + n$ $T(n) = 2T(n/2) + n$, guess $T(n) = O(n \log n)$ $T(n) = O(n \log n)$ and prove it by induction.

2. **Recurrence Tree Method:**

- Visualize the recurrence as a tree and calculate the total work done at all levels.
- Example: For $T(n) = 2T(n/2) + n$ $T(n) = 2T(n/2) + n$, draw the tree to see the pattern of recursive calls and work done at each level.

3. **Master Theorem:**

- Provides a straightforward way to solve recurrences of the form

$$T(n) = aT(n/b) + f(n) \quad T(n) = aT(n/b) + f(n)$$

- Example: For $T(n) = 2T(n/2) + n$, apply the Master Theorem to get
 $T(n) = O(n \log n)$

Examples

Example 1: Binary Search

Binary search is a classic example of an algorithm that can be analyzed using recurrence relations. The recurrence relation for the time complexity of binary search is:

$$T(n) = T(n/2) + O(1) \quad T(n) = T(n/2) + O(1)$$

This is because each step of binary search reduces the problem size by half and performs a constant amount of work.

Using the Master Theorem, we can solve this recurrence:

- $a=1, b=2, f(n)=O(1)$
- Since $f(n) = O(1)$ and $n^{\log_b a} = n^{\log_2 1} = n^0 = 1$, we have $f(n) = O(n^c)$ where $c=0$.

By the Master Theorem, $T(n) = O(\log n)$

Example 2: Merge Sort

Merge sort is another common algorithm that can be analyzed with recurrence relations. The recurrence relation for merge sort is:

$$T(n) = 2T(n/2) + O(n)$$

$$T(n) = 2T(n/2) + O(n)$$

This is because merge sort splits the problem into two halves, sorts each half recursively, and then merges the sorted halves, which takes linear time.

Using the Master Theorem:

- $a=2, b=2, f(n)=O(n)$
 $O(n) = O(n)$
- Since $f(n) = O(n)$ and $n^{\log_b a} = n^{\log_2 2} = n$, we have $f(n) = O(n^c)$ where $c=1$.

By the Master Theorem, $T(n) = O(n \log n)$.

Importance in Algorithm Analysis

Understanding recurrence relations is essential for analyzing recursive algorithms, which are prevalent in many areas of computer science, including sorting algorithms, divide-and-conquer strategies, dynamic programming, and more. Recurrence relations provide a systematic way to quantify the performance of these algorithms and to derive their time complexities.

Conclusion

Recurrence relations are a powerful tool in the analysis of algorithms, enabling us to understand the efficiency and scalability of recursive processes.

Mastering the techniques for solving recurrence relations, such as substitution, recurrence trees, and the Master Theorem, is crucial for any programmer or computer scientist involved in algorithm design and analysis. By leveraging these techniques, one can develop more efficient and robust algorithms, leading to better software performance and user experiences.

Data Structures

Data structures are fundamental concepts in computer science and software engineering. They provide a means of organizing and storing data in a way that enables efficient access and modification. Choosing the right data structure for a given problem is crucial for optimizing performance and ensuring that algorithms run efficiently.

Definition

A data structure is a particular way of organizing data in a computer so that it can be used effectively. The choice of data structure can significantly affect the efficiency of a program. Data structures are used to store data in a format that can be easily accessed, managed, and updated.

Importance of Data Structures

1. **Efficiency:** The right data structure allows for efficient storage, retrieval, and modification of data.
2. **Scalability:** Proper data structures enable programs to handle large amounts of data and complex operations gracefully.
3. **Maintainability:** Using appropriate data structures makes code easier to understand, maintain, and extend.
4. **Reusability:** Well-designed data structures can be reused across different programs and projects.

Arrays and Linked Lists

Arrays in TypeScript

Arrays are a fundamental data structure used to store multiple values in a single variable. They allow for efficient access and modification of elements. In TypeScript, arrays can be defined to store elements of any type, providing strong type-checking and reducing errors during development.

Defining Arrays

In TypeScript, arrays can be defined in multiple ways:

1. Using Square Brackets ([]):

```
let numbers: number[] = [1, 2, 3, 4, 5];
let strings: string[] = ['apple', 'banana', 'cherry'];
```

2. Using the Array Generic Type:

```
let numbers: Array<number> = [1, 2, 3, 4, 5];
let strings: Array<string> = ['apple', 'banana', 'cherry'];
```

Accessing and Modifying Elements

You can access and modify elements in an array using their index:

```
let numbers: number[] = [1, 2, 3, 4, 5];

console.log(numbers[0]); // Output: 1
numbers[1] = 10;
```



```
console.log(numbers);    // Output: [1, 10, 3, 4, 5]
```

Iterating Over Arrays

There are several ways to iterate over arrays in TypeScript:

1. Using a for Loop:

```
let numbers: number[] = [1, 2, 3, 4, 5];
for (let i = 0; i < numbers.length; i++) {
    console.log(numbers[i]);
}
```

2. Using a for...of Loop:

```
for (let num of numbers) {
    console.log(num);
}
```

3. Using forEach Method:

```
numbers.forEach((num) => {
    console.log(num);
});
```

Array Methods

TypeScript arrays inherit methods from JavaScript, such as push, pop, shift, unshift, splice, and slice.

1. Adding and Removing Elements:

```
numbers.push(6); // Adds 6 to the end of the array
console.log(numbers); // Output: [1, 10, 3, 4, 5, 6]
```

```
numbers.pop();    // Removes the last element
console.log(numbers); // Output: [1, 10, 3, 4, 5]
```

```
numbers.shift(); // Removes the first element
console.log(numbers); // Output: [10, 3, 4, 5]
```

```
numbers.unshift(0); // Adds 0 to the beginning
of the array
console.log(numbers); // Output: [0, 10, 3, 4, 5]
```

2. Slicing and Splicing:

```
let slicedArray = numbers.slice(1, 3);
console.log(slicedArray); // Output: [10, 3]
```

```
numbers.splice(2, 1, 20, 30); // Removes one
element at index 2 and adds 20, 30
console.log(numbers); // Output: [0, 10, 20, 30, 5]
```

Complete Example Code

Below is a complete example demonstrating the creation, modification, and iteration of arrays in TypeScript:

```
// Defining an array of numbers
let numbers: number[] = [1, 2, 3, 4, 5];

// Accessing elements
console.log("First element:", numbers[0]); //
Output: 1
```

```
console.log("Second element:", numbers[1]);  
// Output: 2
```

```
// Modifying elements  
numbers[1] = 10;  
console.log("Modified array:", numbers);    //  
Output: [1, 10, 3, 4, 5]
```

```
// Iterating over an array  
console.log("Array elements:");  
for (let num of numbers) {  
    console.log(num); // Output: 1, 10, 3, 4,  
5  
}
```

```
// Using array methods  
numbers.push(6); // Adds 6 to the end of the  
array  
console.log("After push:", numbers);    //  
Output: [1, 10, 3, 4, 5, 6]
```

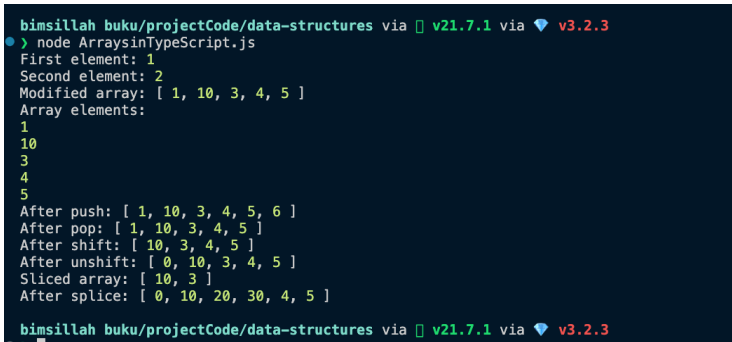
```
numbers.pop(); // Removes the last element  
console.log("After pop:", numbers); // Output:  
[1, 10, 3, 4, 5]
```

```
numbers.shift(); // Removes the first element  
console.log("After shift:", numbers);    //  
Output: [10, 3, 4, 5]
```

```
numbers.unshift(0); // Adds 0 to the beginning  
of the array  
console.log("After unshift:", numbers);    //  
Output: [0, 10, 3, 4, 5]
```

```
// Slicing an array
let slicedArray = numbers.slice(1, 3);
console.log("Sliced array:", slicedArray); //
Output: [10, 3]
```

```
// Splicing an array
numbers.splice(2, 1, 20, 30); // Removes one
element at index 2 and adds 20, 30
console.log("After splice:", numbers); //
Output: [0, 10, 20, 30, 5]
```

A terminal window with a dark blue background and light green text. The prompt is 'bimsillah buku/projectCode/data-structures via v21.7.1 via v3.2.3'. The user enters '> node ArraysInTypeScript.js'. The output shows: 'First element: 1', 'Second element: 2', 'Modified array: [1, 10, 3, 4, 5]', 'Array elements:' followed by a list of elements '1', '10', '3', '4', '5' on separate lines. Then it shows 'After push: [1, 10, 3, 4, 5, 6]', 'After pop: [1, 10, 3, 4, 5]', 'After shift: [10, 3, 4, 5]', 'After unshift: [0, 10, 3, 4, 5]', 'Sliced array: [10, 3]', and 'After splice: [0, 10, 20, 30, 4, 5]'. The prompt is repeated at the bottom.

```
bimsillah buku/projectCode/data-structures via v21.7.1 via v3.2.3
> node ArraysInTypeScript.js
First element: 1
Second element: 2
Modified array: [ 1, 10, 3, 4, 5 ]
Array elements:
1
10
3
4
5
After push: [ 1, 10, 3, 4, 5, 6 ]
After pop: [ 1, 10, 3, 4, 5 ]
After shift: [ 10, 3, 4, 5 ]
After unshift: [ 0, 10, 3, 4, 5 ]
Sliced array: [ 10, 3 ]
After splice: [ 0, 10, 20, 30, 4, 5 ]
bimsillah buku/projectCode/data-structures via v21.7.1 via v3.2.3
```

Image 1 result of code

This example showcases various operations that can be performed on arrays in TypeScript, illustrating their versatility and ease of use.

Linked Lists (Singly, Doubly, Circular) in TypeScript

Linked Lists in TypeScript

Linked lists are a type of data structure that consist of nodes where each node contains data and a reference (or link) to the next node in the sequence. Linked lists can be of various types: singly linked lists, doubly linked lists, and circular linked lists.

Singly Linked List

A singly linked list is a type of linked list where each node points to the next node in the sequence, and the last node points to null.

Example Implementation in TypeScript:

```
class Node<T> {
  data: T;
  next: Node<T> | null = null;

  constructor(data: T) {
    this.data = data;
  }
}

class SinglyLinkedList<T> {
  head: Node<T> | null = null;

  append(data: T) {
    const newNode = new Node(data);
    if (!this.head) {
      this.head = newNode;
    }
  }
}
```


```

    } else {
        let current = this.head;
        while (current.next) {
            current = current.next;
        }
        current.next = newNode;
    }
}

printList() {
    let current = this.head;
    while (current) {
        console.log(current.data);
        current = current.next;
    }
}
}

// Usage
const singlyLinkedList = new
SinglyLinkedList<number>();
singlyLinkedList.append(1);
singlyLinkedList.append(2);
singlyLinkedList.append(3);
singlyLinkedList.printList(); // Output: 1, 2,
3

```



```

bimsillah buku/projectCode/data-structures via v21.7.1 via v3.2.3
➤ node LinkedLists1.js
1
2
3
bimsillah buku/projectCode/data-structures via v21.7.1 via v3.2.3
➤

```

Image 1.1 Singly Linked List

Doubly Linked List

A doubly linked list is a type of linked list where each node points to both the next node and the previous node in the sequence.

Example Implementation in TypeScript:

```
class DoublyNode<T> {
  data: T;
  next: DoublyNode<T> | null = null;
  prev: DoublyNode<T> | null = null;

  constructor(data: T) {
    this.data = data;
  }
}

class DoublyLinkedList<T> {
  head: DoublyNode<T> | null = null;
  tail: DoublyNode<T> | null = null;

  append(data: T) {
    const newNode = new DoublyNode(data);
    if (!this.head) {
      this.head = this.tail = newNode;
    } else {
      this.tail!.next = newNode;
      newNode.prev = this.tail;
      this.tail = newNode;
    }
  }

  printList() {
```

```

        let current = this.head;
        while (current) {
            console.log(current.data);
            current = current.next;
        }
    }
}

```

// Usage

```

const doublyLinkedList = new
DoublyLinkedList<number>();
doublyLinkedList.append(1);
doublyLinkedList.append(2);
doublyLinkedList.append(3);
doublyLinkedList.printList(); // Output: 1, 2,
3

```



```

bimsillah buku/projectCode/data-structures via v21.7.1 via v3.2.3
● > tsc DoublyLinkedList.ts

bimsillah buku/projectCode/data-structures via v21.7.1 via v3.2.3
● > node DoublyLinkedList.js
1
2
3

bimsillah buku/projectCode/data-structures via v21.7.1 via v3.2.3

```

Image 1.3 Doubly Linked List

Circular Linked List

A circular linked list is a type of linked list where the last node points back to the first node, forming a circle.

Example Implementation in TypeScript:

```
class CircularNode<T> {
  data: T;
  next: CircularNode<T> | null = null;

  constructor(data: T) {
    this.data = data;
  }
}

class CircularLinkedList<T> {
  head: CircularNode<T> | null = null;

  append(data: T) {
    const newNode = new
CircularNode(data);
    if (!this.head) {
      this.head = newNode;
      newNode.next = this.head;
    } else {
      let current = this.head;
      while (current.next !== this.head)
{
        current = current.next;
      }
      current.next = newNode;
      newNode.next = this.head;
    }
  }
}
```

```

    }
  }

  printList() {
    if (!this.head) return;

    let current = this.head;
    do {
      console.log(current.data);
      current = current.next;
    } while (current !== this.head);
  }
}

```

// Usage

```

const circularLinkedList = new
CircularLinkedList<number>();
circularLinkedList.append(1);
circularLinkedList.append(2);
circularLinkedList.append(3);
circularLinkedList.printList(); // Output: 1,
2, 3

```



```

bimsillah buku/projectCode/data-structures via v21.7.1 via v3.2.3
• > tsc CircularLinkedList.ts

bimsillah buku/projectCode/data-structures via v21.7.1 via v3.2.3
• > node CircularLinkedList.js
1
2
3

bimsillah buku/projectCode/data-structures via v21.7.1 via v3.2.3
○ >

```

Image 1.4 Circular Linked List

These implementations demonstrate how to create and use different types of linked lists in TypeScript, showcasing the versatility and structure of each type.

Stacks and Queues

Stacks and queues are fundamental data structures used in programming to store and manage collections of data. Each has its own unique characteristics and use cases.

Stacks A stack is a collection of elements that follows the Last In, First Out (LIFO) principle. This means that the last element added to the stack will be the first one to be removed. Stacks are used in various applications, including function call management in programming languages, expression evaluation, and backtracking algorithms.

Common operations on a stack:

- **push:** Add an element to the top of the stack.
- **pop:** Remove the top element from the stack.
- **peek:** Retrieve the top element without removing it.
- **isEmpty:** Check if the stack is empty.

Queues

A queue is a collection of elements that follows the First In, First Out (FIFO) principle. This means that the first element added to the queue will be the first one to be removed. Queues are commonly used in scenarios such as task scheduling, managing requests in web servers, and breadth-first search algorithms.

Common operations on a queue:

- **enqueue:** Add an element to the end of the queue.

- dequeue: Remove the front element from the queue.
- front: Retrieve the front element without removing it.
- isEmpty: Check if the queue is empty.

Stacks vs. Queues

While both stacks and queues are used to manage collections of elements, they serve different purposes and have distinct characteristics. Stacks are useful for scenarios where you need to access the most recently added elements first, such as undo mechanisms or parsing expressions. Queues, on the other hand, are ideal for managing tasks in a first-come, first-served order, such as job scheduling or handling requests.

In conclusion, understanding stacks and queues is crucial for solving various computational problems and optimizing the efficiency of your algorithms. By leveraging these data structures appropriately, you can enhance the performance and readability of your code.

Implementing Stacks in TypeScript

Stacks are a linear data structure that follows the Last In, First Out (LIFO) principle. In TypeScript, you can implement a stack using an array to store the elements. Below, we provide a detailed implementation of a stack along with explanations of its methods.

Stack Implementation in TypeScript

```
class Stack<T> {
  private items: T[] = [];
```

```

    // Adds an element to the top of the stack
    push(element: T): void {
        this.items.push(element);
    }

    // Removes and returns the top element of
the stack
    pop(): T | undefined {
        if (this.isEmpty()) {
            return undefined;
        }
        return this.items.pop();
    }

    // Returns the top element of the stack
without removing it
    peek(): T | undefined {
        if (this.isEmpty()) {
            return undefined;
        }
        return this.items[this.items.length -
1];
    }

    // Checks if the stack is empty
    isEmpty(): boolean {
        return this.items.length === 0;
    }

    // Returns the number of elements in the
stack
    size(): number {
        return this.items.length;
    }

```

```

    }

    // Empties the stack
    clear(): void {
        this.items = [];
    }
}

// Usage example
const stack = new Stack<number>();
stack.push(10);
stack.push(20);
console.log(stack.peek()); // Output: 20
console.log(stack.pop());  // Output: 20
console.log(stack.size()); // Output: 1
console.log(stack.isEmpty()); // Output: false
stack.clear();
console.log(stack.isEmpty()); // Output: true

```

Explanation of Methods

1. **push(element: T): void**

- Adds a new element to the top of the stack.
- Uses the push method of the array to add the element.

2. **pop(): T | undefined**

- Removes and returns the top element of the stack.
- Checks if the stack is empty using the isEmpty method.

- Uses the pop method of the array to remove the element.
- 3. **peek(): T | undefined**
 - Returns the top element without removing it.
 - Checks if the stack is empty using the isEmpty method.
 - Accesses the last element of the array using this.items.length - 1.
- 4. **isEmpty(): boolean**
 - Checks if the stack is empty.
 - Returns true if the array length is zero, otherwise returns false.
- 5. **size(): number**
 - Returns the number of elements in the stack.
 - Uses the length property of the array.
- 6. **clear(): void**
 - Empties the stack.
 - Resets the array to an empty array.

Advantages of Using Stacks

1. **Simple Implementation:** Stacks can be easily implemented using arrays or linked lists.
2. **Efficient Operations:** Both push and pop operations have constant time complexity, $O(1)$.
3. **Useful for Certain Algorithms:** Stacks are essential for algorithms that require reversing items, such as depth-first search (DFS) and backtracking.

By understanding and implementing stacks in TypeScript, you can efficiently manage and manipulate collections of data following the LIFO principle. This

foundational data structure is widely used in various computational problems and algorithm implementations.

Implementing Queues in TypeScript

Queues are a linear data structure that follows the First In, First Out (FIFO) principle. In TypeScript, you can implement a queue using an array to store the elements. Below, we provide a detailed implementation of a queue along with explanations of its methods.

Queue Implementation in TypeScript

```
class Queue<T> {
  private items: T[] = [];

  // Adds an element to the end of the queue
  enqueue(element: T): void {
    this.items.push(element);
  }

  // Removes and returns the first element of
the queue
  dequeue(): T | undefined {
    if (this.isEmpty()) {
      return undefined;
    }
    return this.items.shift();
  }

  // Returns the first element of the queue
without removing it
  front(): T | undefined {
    if (this.isEmpty()) {
```



```

        return undefined;
    }
    return this.items[0];
}

// Checks if the queue is empty
isEmpty(): boolean {
    return this.items.length === 0;
}

// Returns the number of elements in the
queue
size(): number {
    return this.items.length;
}

// Empties the queue
clear(): void {
    this.items = [];
}
}

// Usage example
const queue = new Queue<number>();
queue.enqueue(10);
queue.enqueue(20);
console.log(queue.front()); // Output: 10
console.log(queue.dequeue()); // Output: 10
console.log(queue.size()); // Output: 1
console.log(queue.isEmpty()); // Output:
false
queue.clear();
console.log(queue.isEmpty()); // Output: true

```

Explanation of Methods

1. **enqueue(element: T): void**

- Adds a new element to the end of the queue.
- Uses the push method of the array to add the element.

2. **dequeue(): T | undefined**

- Removes and returns the first element of the queue.
- Checks if the queue is empty using the isEmpty method.
- Uses the shift method of the array to remove the element.

3. **front(): T | undefined**

- Returns the first element without removing it.
- Checks if the queue is empty using the isEmpty method.
- Accesses the first element of the array using this.items[0].

4. **isEmpty(): boolean**

- Checks if the queue is empty.
- Returns true if the array length is zero, otherwise returns false.

5. **size(): number**

- Returns the number of elements in the queue.
- Uses the length property of the array.

6. **clear(): void**

- Empties the queue.
- Resets the array to an empty array.

Advantages of Using Queues

1. **Simple Implementation:** Queues can be easily implemented using arrays or linked lists.
2. **Efficient Operations:** Both enqueue and dequeue operations have constant time complexity, $O(1)$.
3. **Useful for Certain Algorithms:** Queues are essential for algorithms that require processing items in the order they arrive, such as breadth-first search (BFS) and certain scheduling algorithms.

By understanding and implementing queues in TypeScript, you can efficiently manage and manipulate collections of data following the FIFO principle. This foundational data structure is widely used in various computational problems and algorithm implementations.

Dequeues in TypeScript

A deque (double-ended queue) is a linear data structure that allows insertion and deletion of elements from both ends, i.e., from the front and the rear. Deques can be implemented in TypeScript using arrays. Below is a detailed implementation of a deque along with explanations of its methods.

Deque Implementation in TypeScript

```
class Deque<T> {  
    private items: T[] = [];  
  
    // Adds an element to the front of the deque  
    addFront(element: T): void {  
        this.items.unshift(element);  
    }  
}
```

```

    }

    // Adds an element to the end of the deque
    addRear(element: T): void {
        this.items.push(element);
    }

    // Removes and returns the front element of
the deque
    removeFront(): T | undefined {
        if (this.isEmpty()) {
            return undefined;
        }
        return this.items.shift();
    }

    // Removes and returns the rear element of
the deque
    removeRear(): T | undefined {
        if (this.isEmpty()) {
            return undefined;
        }
        return this.items.pop();
    }

    // Returns the front element of the deque
without removing it
    peekFront(): T | undefined {
        if (this.isEmpty()) {
            return undefined;
        }
        return this.items[0];
    }

```

```

        // Returns the rear element of the deque
        without removing it
        peekRear(): T | undefined {
            if (this.isEmpty()) {
                return undefined;
            }
            return this.items[this.items.length -
1];
        }

        // Checks if the deque is empty
        isEmpty(): boolean {
            return this.items.length === 0;
        }

        // Returns the number of elements in the
        deque
        size(): number {
            return this.items.length;
        }

        // Empties the deque
        clear(): void {
            this.items = [];
        }
    }

    // Usage example
    const deque = new Deque<number>();
    deque.addFront(10);
    deque.addRear(20);
    console.log(deque.peekFront()); // Output: 10

```

```
console.log(deque.peekRear()); // Output: 20
console.log(deque.removeFront()); // Output:
10
console.log(deque.removeRear()); // Output:
20
console.log(deque.size()); // Output: 0
console.log(deque.isEmpty()); // Output: true
deque.clear();
console.log(deque.isEmpty()); // Output: true
```

Explanation of Methods

1. **addFront(element: T): void**
 - Adds a new element to the front of the deque.
 - Uses the unshift method of the array to add the element.
2. **addRear(element: T): void**
 - Adds a new element to the end of the deque.
 - Uses the push method of the array to add the element.
3. **removeFront(): T | undefined**
 - Removes and returns the front element of the deque.
 - Checks if the deque is empty using the isEmpty method.
 - Uses the shift method of the array to remove the element.
4. **removeRear(): T | undefined**
 - Removes and returns the rear element of the deque.

- Checks if the deque is empty using the isEmpty method.
 - Uses the pop method of the array to remove the element.
5. **peekFront(): T | undefined**
- Returns the front element without removing it.
 - Checks if the deque is empty using the isEmpty method.
 - Accesses the front element of the array using this.items[0].
6. **peekRear(): T | undefined**
- Returns the rear element without removing it.
 - Checks if the deque is empty using the isEmpty method.
 - Accesses the rear element of the array using this.items[this.items.length - 1].
7. **isEmpty(): boolean**
- Checks if the deque is empty.
 - Returns true if the array length is zero, otherwise returns false.
8. **size(): number**
- Returns the number of elements in the deque.
 - Uses the length property of the array.
9. **clear(): void**
- Empties the deque.
 - Resets the array to an empty array.

Advantages of Using Deques

1. **Versatile Operations:** Deques support insertion and deletion from both ends, making them more flexible than regular queues and stacks.
2. **Efficient Operations:** Both `addFront`, `addRear`, `removeFront`, and `removeRear` operations have constant time complexity, $O(1)$.
3. **Useful for Certain Algorithms:** Deques are essential for algorithms that require access to both ends of the data structure, such as certain sliding window problems.

By understanding and implementing deques in TypeScript, you can efficiently manage and manipulate collections of data that require flexible insertion and deletion operations. This foundational data structure is widely used in various computational problems and algorithm implementations.

Trees

A tree is a widely used data structure in algorithms and computer science. It is a hierarchical structure consisting of nodes, with a single node designated as the root and all other nodes forming a hierarchy of parent-child relationships. Trees are used in many applications, including databases, file systems, and network routing algorithms.

Basic Concepts of Trees

1. **Node:** Each element in a tree is called a node.
2. **Root:** The top node in a tree.
3. **Parent:** A node that has one or more child nodes.
4. **Child:** A node that has a parent node.
5. **Leaf:** A node with no children.
6. **Edge:** The connection between two nodes.
7. **Subtree:** A tree formed by a node and its descendants.
8. **Depth:** The length of the path from the root to a node.
9. **Height:** The length of the path from a node to the deepest leaf.

Types of Trees

1. **Binary Tree:** A tree where each node has at most two children, referred to as the left child and the right child.
2. **Binary Search Tree (BST):** A binary tree where the left child contains only nodes with values less than the parent node, and the right child contains only nodes with values greater than the parent node.

3. **AVL Tree:** A self-balancing binary search tree where the difference in heights of left and right subtrees cannot be more than one.
4. **Red-Black Tree:** A self-balancing binary search tree with additional properties to ensure balance.
5. **B-tree:** A balanced tree data structure that generalizes the binary search tree, allowing for nodes with more than two children.

Tree Traversal Algorithms

1. **In-Order Traversal:** Visit the left subtree, the root node, and then the right subtree.
2. **Pre-Order Traversal:** Visit the root node, then the left subtree, and finally the right subtree.
3. **Post-Order Traversal:** Visit the left subtree, the right subtree, and then the root node.
4. **Level-Order Traversal:** Visit nodes level by level starting from the root.

Conclusion

Trees are a fundamental data structure in computer science, used to represent hierarchical relationships. Understanding tree algorithms and how to implement them in TypeScript is essential for solving various computational problems efficiently. The provided implementation of a binary search tree demonstrates the basic operations such as insertion, traversal, and search, which are the building blocks for more complex tree-based algorithms.

Binary Trees

1. Introduction to Binary Trees

Binary trees are an essential data structure in computer science, where each node has at most two children. This structure is widely used in various applications, such as searching, sorting, and representing hierarchical data.

2. Types of Binary Trees

There are several types of binary trees, each with specific properties that make them suitable for different tasks:

- **Full Binary Tree:** Every node has 0 or 2 children.
- **Perfect Binary Tree:** All interior nodes have two children, and all leaves are at the same level.
- **Complete Binary Tree:** All levels are fully filled except possibly the last, which is filled from left to right.
- **Balanced Binary Tree:** The height of the tree is minimized, ensuring that operations like insert, delete, and search take $O(\log n)$ time.
- **Binary Search Tree (BST):** A binary tree where the left child contains only nodes with values less than the parent node, and the right child contains only nodes with values greater than the parent node.

3. Implementation of Binary Trees in TypeScript

This section covers how to implement binary trees in TypeScript, starting with a basic tree node class and moving on to more complex tree operations.

3.1 Basic Tree Node Class

```
class TreeNode<T> {
    value: T;
    left: TreeNode<T> | null = null;
    right: TreeNode<T> | null = null;

    constructor(value: T) {
        this.value = value;
    }
}
```

3.2 Binary Tree Class

```
class BinaryTree<T> {
    root: TreeNode<T> | null = null;

    insert(value: T) {
        const newNode = new TreeNode(value);
        if (!this.root) {
            this.root = newNode;
        } else {
            this.insertNode(this.root,
newNode);
        }
    }

    private insertNode(node:  TreeNode<T>,
newNode: TreeNode<T>) {
        if (newNode.value < node.value) {
            if (!node.left) {
                node.left = newNode;
            } else {

```

```

                                this.insertNode(node.left,
newNode);
                                }
                                } else {
                                    if (!node.right) {
                                        node.right = newNode;
                                    } else {
                                        this.insertNode(node.right,
newNode);
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}

```

4. Common Operations on Binary Trees

This section explains common operations such as searching, traversing, and deleting nodes in a binary tree.

4.1 In-Order Traversal

In-order traversal visits the nodes in ascending order for a binary search tree.

```

inOrderTraversal(node:  TreeNode<T>  |  null,
callback: (value: T) => void): void {
    if (node) {
        this.inOrderTraversal(node.left,
callback);
        callback(node.value);
        this.inOrderTraversal(node.right,
callback);
    }
}

```

4.2 Pre-Order Traversal

Pre-order traversal visits the root node before its children.

```
preOrderTraversal(node: TreeNode<T> | null,  
callback: (value: T) => void): void {  
    if (node) {  
        callback(node.value);  
        this.preOrderTraversal(node.left,  
callback);  
        this.preOrderTraversal(node.right,  
callback);  
    }  
}
```

4.3 Post-Order Traversal

Post-order traversal visits the children before the root node.

```
postOrderTraversal(node: TreeNode<T> | null,  
callback: (value: T) => void): void {  
    if (node) {  
        this.postOrderTraversal(node.left,  
callback);  
        this.postOrderTraversal(node.right,  
callback);  
        callback(node.value);  
    }  
}
```

5. Binary Search Tree (BST) Operations

This section covers operations specific to binary search trees, such as searching for a value, finding the minimum and maximum values, and deleting a node.

5.1 Search Operation

Searching in a BST involves comparing the target value with the root and recursively searching the left or right subtree.

```
search(node: TreeNode<T> | null, value: T):  
TreeNode<T> | null {  
    if (!node || node.value === value) {  
        return node;  
    }  
    if (value < node.value) {  
        return this.search(node.left, value);  
    } else {  
        return this.search(node.right, value);  
    }  
}
```

5.2 Deletion Operation

Deleting a node in a BST requires careful consideration to maintain the tree's properties. The process differs depending on whether the node to be deleted has no children, one child, or two children.

```
deleteNode(node: TreeNode<T> | null, value: T):  
TreeNode<T> | null {  
    if (!node) return null;
```

```

        if (value < node.value) {
            node.left = this.deleteNode(node.left,
value);
            return node;
        } else if (value > node.value) {
            node.right
            =
this.deleteNode(node.right, value);
            return node;
        } else {
            // Node with only one child or no child
            if (!node.left) return node.right;
            if (!node.right) return node.left;

            // Node with two children: Get the
inorder successor
            const          successor          =
this.minValueNode(node.right);
            node.value = successor.value;
            node.right
            =
this.deleteNode(node.right, successor.value);
            return node;
        }
    }
}

```

```

private    minValueNode(node:    TreeNode<T>):
TreeNode<T> {
    let current = node;
    while (current.left) {
        current = current.left;
    }
    return current;
}

```


6. Conclusion

Binary trees, particularly binary search trees, are powerful tools for managing and organizing data. Understanding their implementation and operations is crucial for solving a wide range of computational problems efficiently.

Binary Search Trees

Certainly! Here's the continuation of your "Binary Trees" section without page numbers:

Binary Trees

1. Introduction to Binary Trees

Binary trees are an essential data structure in computer science, where each node has at most two children. This structure is widely used in various applications, such as searching, sorting, and representing hierarchical data.

2. Types of Binary Trees

There are several types of binary trees, each with specific properties that make them suitable for different tasks:

- **Full Binary Tree:** Every node has 0 or 2 children.
- **Perfect Binary Tree:** All interior nodes have two children, and all leaves are at the same level.
- **Complete Binary Tree:** All levels are fully filled except possibly the last, which is filled from left to right.

- **Balanced Binary Tree:** The height of the tree is minimized, ensuring that operations like insert, delete, and search take $O(\log n)$ time.
- **Binary Search Tree (BST):** A binary tree where the left child contains only nodes with values less than the parent node, and the right child contains only nodes with values greater than the parent node.

3. Implementation of Binary Trees in TypeScript

This section covers how to implement binary trees in TypeScript, starting with a basic tree node class and moving on to more complex tree operations.

3.1 Basic Tree Node Class

```
class TreeNode<T> {
  value: T;
  left: TreeNode<T> | null = null;
  right: TreeNode<T> | null = null;

  constructor(value: T) {
    this.value = value;
  }
}
```

3.2 Binary Tree Class

```
class BinaryTree<T> {
  root: TreeNode<T> | null = null;

  insert(value: T) {
    const newNode = new TreeNode(value);
    if (!this.root) {
```

```

        this.root = newNode;
    } else {
        this.insertNode(this.root,
newNode);
    }
}

private    insertNode(node:    TreeNode<T>,
newNode: TreeNode<T>) {
    if (newNode.value < node.value) {
        if (!node.left) {
            node.left = newNode;
        } else {
            this.insertNode(node.left,
newNode);
        }
    } else {
        if (!node.right) {
            node.right = newNode;
        } else {
            this.insertNode(node.right,
newNode);
        }
    }
}
}

```

4. Common Operations on Binary Trees

This section explains common operations such as searching, traversing, and deleting nodes in a binary tree.

4.1 In-Order Traversal

In-order traversal visits the nodes in ascending order for a binary search tree.

```
inOrderTraversal(node: TreeNode<T> | null,  
callback: (value: T) => void): void {  
    if (node) {  
        this.inOrderTraversal(node.left,  
callback);  
        callback(node.value);  
        this.inOrderTraversal(node.right,  
callback);  
    }  
}
```

4.2 Pre-Order Traversal

Pre-order traversal visits the root node before its children.

```
preOrderTraversal(node: TreeNode<T> | null,  
callback: (value: T) => void): void {  
    if (node) {  
        callback(node.value);  
        this.preOrderTraversal(node.left,  
callback);  
        this.preOrderTraversal(node.right,  
callback);  
    }  
}
```

4.3 Post-Order Traversal

Post-order traversal visits the children before the root node.

```

postOrderTraversal(node: TreeNode<T> | null,
callback: (value: T) => void): void {
    if (node) {
        this.postOrderTraversal(node.left,
callback);
        this.postOrderTraversal(node.right,
callback);
        callback(node.value);
    }
}

```

5. Binary Search Tree (BST) Operations

This section covers operations specific to binary search trees, such as searching for a value, finding the minimum and maximum values, and deleting a node.

5.1 Search Operation

Searching in a BST involves comparing the target value with the root and recursively searching the left or right subtree.

```

search(node: TreeNode<T> | null, value: T):
TreeNode<T> | null {
    if (!node || node.value === value) {
        return node;
    }
    if (value < node.value) {
        return this.search(node.left, value);
    } else {
        return this.search(node.right, value);
    }
}

```

5.2 Deletion Operation

Deleting a node in a BST requires careful consideration to maintain the tree's properties. The process differs depending on whether the node to be deleted has no children, one child, or two children.

```
deleteNode(node: TreeNode<T> | null, value: T):
TreeNode<T> | null {
    if (!node) return null;

    if (value < node.value) {
        node.left = this.deleteNode(node.left,
value);
        return node;
    } else if (value > node.value) {
        node.right
=
this.deleteNode(node.right, value);
        return node;
    } else {
        // Node with only one child or no child
        if (!node.left) return node.right;
        if (!node.right) return node.left;

        // Node with two children: Get the
inorder successor
        const
        successor
=
this.minValueNode(node.right);
        node.value = successor.value;
        node.right
=
this.deleteNode(node.right, successor.value);
        return node;
    }
}
```

```

private    minValueNode(node:    TreeNode<T>):
TreeNode<T> {
    let current = node;
    while (current.left) {
        current = current.left;
    }
    return current;
}

```

6. Conclusion

Binary trees, particularly binary search trees, are powerful tools for managing and organizing data. Understanding their implementation and operations is crucial for solving a wide range of computational problems efficiently.

AVL Trees

1. Introduction to AVL Trees

An AVL tree is a self-balancing binary search tree (BST) where the height of the two child subtrees of any node differs by no more than one. If at any time the height difference becomes more than one, rebalancing is done to restore the AVL property. Named after its inventors, Adelson-Velsky and Landis, AVL trees guarantee $O(\log n)$ time complexity for search, insertion, and deletion operations by maintaining balance.

2. Properties of AVL Trees

- **Height-Balanced:** The balance factor (height difference between left and right subtrees) of each node is between -1 and 1.

- **Self-Balancing:** After every insertion or deletion, the tree rebalances itself to maintain the height property.
- **Height:** The height of an AVL tree is strictly $O(\log n)$, ensuring efficient operations.

3. Implementation of AVL Trees in TypeScript

This section walks through the implementation of an AVL tree in TypeScript, including node insertion with rebalancing.

3.1 Tree Node Class

The tree node class includes an additional height property to keep track of the height of each node.

```
class TreeNode<T> {
  value: T;
  left: TreeNode<T> | null = null;
  right: TreeNode<T> | null = null;
  height: number = 1;

  constructor(value: T) {
    this.value = value;
  }
}
```

3.2 AVL Tree Class

The AVL tree class extends the basic BST with methods for balancing the tree after insertions and deletions.

```
class AVLTree<T> {
  root: TreeNode<T> | null = null;

  insert(value: T) {
```



```

        this.root = this.insertNode(this.root,
value);
    }

    private insertNode(node: TreeNode<T> |
null, value: T): TreeNode<T> {
        if (!node) {
            return new TreeNode(value);
        }

        if (value < node.value) {
            node.left =
this.insertNode(node.left, value);
        } else {
            node.right =
this.insertNode(node.right, value);
        }

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

        const balance = this.getBalance(node);

        // Left Left Case
        if (balance > 1 && value <
node.left!.value) {
            return this.rightRotate(node);
        }

        // Right Right Case
        if (balance < -1 && value >
node.right!.value) {

```

```

        return this.leftRotate(node);
    }

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

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

    return node;
}
}

```

3.3 Rotation Methods

Rotations are crucial for maintaining the AVL tree's balance. This section includes the code for left and right rotations.

- **Right Rotation:** Used to balance nodes when the left subtree is heavier.
- **Left Rotation:** Used to balance nodes when the right subtree is heavier.

```

private      rightRotate(y:      TreeNode<T>):
TreeNode<T> {
    const x = y.left!;
    const T2 = x.right;

    // Perform rotation
    x.right = y;
    y.left = T2;

    // Update heights
    y.height =
Math.max(this.getHeight(y.left),
this.getHeight(y.right)) + 1;
    x.height =
Math.max(this.getHeight(x.left),
this.getHeight(x.right)) + 1;

    // Return new root
    return x;
}

```

```

private      leftRotate(x:      TreeNode<T>):
TreeNode<T> {
    const y = x.right!;
    const T2 = y.left;

    // Perform rotation
    y.left = x;
    x.right = T2;

    // Update heights

```

```

        x.height =
Math.max(this.getHeight(x.left),
this.getHeight(x.right)) + 1;
        y.height =
Math.max(this.getHeight(y.left),
this.getHeight(y.right)) + 1;

        // Return new root
        return y;
    }

```

3.4 Utility Methods

These methods help in calculating the height of a node and determining the balance factor.

```

private getHeight(node: TreeNode<T> | null):
number {
    return node ? node.height : 0;
}

```

```

private getBalance(node: TreeNode<T> | null):
number {
    return node ? this.getHeight(node.left) -
this.getHeight(node.right) : 0;
}

```

4. AVL Tree Operations

4.1 Insertion

Insertion into an AVL tree is similar to a BST but includes additional steps for balancing the tree.

```

insert(value: T) {

```

```

        this.root    =    this.insertNode(this.root,
value);
    }

```

4.2 Deletion

Deletion in an AVL tree also requires rebalancing. The process is more complex, as it involves handling different cases depending on the balance factor after the node is removed.

```

delete(value: T) {
    this.root    =    this.deleteNode(this.root,
value);
}

```

```

private deleteNode(node: TreeNode<T> | null,
value: T): TreeNode<T> | null {
    if (!node) return null;

    if (value < node.value) {
        node.left = this.deleteNode(node.left,
value);
    } else if (value > node.value) {
        node.right
        =
this.deleteNode(node.right, value);
    } else {
        // Node with only one child or no child
        if (!node.left) return node.right;
        if (!node.right) return node.left;

        // Node with two children: Get the
inorder successor

```

```

        const successor =
this.minValueNode(node.right);
        node.value = successor.value;
        node.right =
this.deleteNode(node.right, successor.value);
    }

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

    const balance = this.getBalance(node);

    // Balance the node if needed
    // Left Left Case
    if (balance > 1 &&
this.getBalance(node.left) >= 0) {
        return this.rightRotate(node);
    }

    // Left Right Case
    if (balance > 1 &&
this.getBalance(node.left) < 0) {
        node.left =
this.leftRotate(node.left!);
        return this.rightRotate(node);
    }

    // Right Right Case
    if (balance < -1 &&
this.getBalance(node.right) <= 0) {
        return this.leftRotate(node);
    }

```

```

        // Right Left Case
        if (balance < -1 &&
this.getBalance(node.right) > 0) {
            node.right =
this.rightRotate(node.right!);
            return this.leftRotate(node);
        }

    return node;
}

```

5. Use Cases of AVL Trees

AVL trees are particularly useful in scenarios where data is frequently inserted and deleted, and balanced trees are essential for maintaining efficient search times:

- **Databases:** To keep the records sorted and allow for fast lookups.
- **Memory management:** In systems where dynamic allocation and deallocation of memory blocks are frequent.
- **File systems:** To maintain a balanced tree structure for directories.

6. Conclusion

AVL trees are a robust solution for maintaining balance in dynamic datasets. By ensuring that the height difference between subtrees is never more than one, AVL trees provide efficient search, insertion, and deletion operations, making them a valuable tool in many computational tasks.

Red-Black Trees

1. Introduction to Red-Black Trees

Red-Black Trees are a type of self-balancing binary search tree where each node contains an extra bit for storing color, which can be either red or black. These trees maintain balance by enforcing specific properties, ensuring that the tree remains approximately balanced, leading to $O(\log n)$ time complexity for search, insertion, and deletion operations.

2. Properties of Red-Black Trees

Red-Black Trees enforce the following properties to maintain balance:

- **Property 1:** Each node is either red or black.
- **Property 2:** The root is always black.
- **Property 3:** All leaves (NIL nodes) are black.
- **Property 4:** If a node is red, then both its children are black (no two red nodes can be adjacent).
- **Property 5:** Every path from a node to its descendant NIL nodes has the same number of black nodes.

3. Implementation of Red-Black Trees in TypeScript

This section provides a walkthrough for implementing a Red-Black Tree in TypeScript, focusing on insertion with necessary rotations and recoloring to maintain tree properties.

3.1 Tree Node Class

The tree node class includes properties for color, value, and pointers to left, right, and parent nodes.

```
enum Color {  
    RED,  
    BLACK,
```



```
}
```

```
class TreeNode<T> {  
    value: T;  
    color: Color;  
    left: TreeNode<T> | null = null;  
    right: TreeNode<T> | null = null;  
    parent: TreeNode<T> | null = null;  
  
    constructor(value: T, color: Color,  
parent: TreeNode<T> | null = null) {  
        this.value = value;  
        this.color = color;  
        this.parent = parent;  
    }  
}
```

3.2 Red-Black Tree Class

The Red-Black Tree class contains methods for inserting nodes, rebalancing the tree, and ensuring that all Red-Black properties are maintained after each operation.

```
class RedBlackTree<T> {  
    private root: TreeNode<T> | null = null;  
  
    insert(value: T) {  
        const newNode = new TreeNode(value,  
Color.RED);  
        this.root = this.insertNode(this.root,  
newNode);  
        this.fixViolation(newNode);  
    }  
}
```

```

        private insertNode(root: TreeNode<T> |
null, node: TreeNode<T>): TreeNode<T> {
            if (!root) {
                return node;
            }

            if (node.value < root.value) {
                root.left =
this.insertNode(root.left, node);
                root.left.parent = root;
            } else if (node.value > root.value) {
                root.right =
this.insertNode(root.right, node);
                root.right.parent = root;
            }

            return root;
        }
    }
}

```

3.3 Fixing Violations

To maintain the Red-Black properties, the tree may need to be restructured and recolored after each insertion. This section covers the logic for resolving violations of the Red-Black properties.

```

private fixViolation(node: TreeNode<T>) {
    let parent: TreeNode<T> | null = null;
    let grandParent: TreeNode<T> | null = null;

    while (node !== this.root && node.color !==
Color.BLACK && node.parent?.color ===
Color.RED) {
        parent = node.parent;
        grandParent = node.parent?.parent;
    }
}

```

```

        if (parent === grandParent?.left) {
            const uncle = grandParent.right;

            // Case 1: The uncle of node is red
            (recoloring)
            if (uncle?.color === Color.RED) {
                grandParent.color = Color.RED;
                parent.color = Color.BLACK;
                uncle.color = Color.BLACK;
                node = grandParent;
            } else {
                // Case 2: node is the right
                child of its parent (left rotation needed)
                if (node === parent.right) {
                    this.leftRotate(parent);
                    node = parent;
                    parent = node.parent;
                }

                // Case 3: node is the left
                child of its parent (right rotation needed)

                this.rightRotate(grandParent);
                const tempColor =
                parent!.color;
                parent!.color =
                grandParent.color;
                grandParent.color = tempColor;
                node = parent!;
            }
        } else {
            const uncle = grandParent?.left;

```

```

        // Mirror image of case 1, 2, 3
        if (uncle?.color === Color.RED) {
            grandParent.color = Color.RED;
            parent.color = Color.BLACK;
            uncle.color = Color.BLACK;
            node = grandParent;
        } else {
            if (node === parent.left) {
                this.rightRotate(parent);
                node = parent;
                parent = node.parent;
            }
            this.leftRotate(grandParent);
            const tempColor =
parent!.color;
            parent!.color =
grandParent.color;
            grandParent.color = tempColor;
            node = parent!;
        }
    }

    this.root!.color = Color.BLACK;
}

```

3.4 Rotation Methods

Rotation methods are essential for maintaining tree balance. These methods perform the required rotations to restore Red-Black properties.

```

private leftRotate(x: TreeNode<T>) {
    const y = x.right!;

```

```

    x.right = y.left;

    if (y.left) {
        y.left.parent = x;
    }

    y.parent = x.parent;

    if (!x.parent) {
        this.root = y;
    } else if (x === x.parent.left) {
        x.parent.left = y;
    } else {
        x.parent.right = y;
    }

    y.left = x;
    x.parent = y;
}

private rightRotate(y: TreeNode<T>) {
    const x = y.left!;
    y.left = x.right;

    if (x.right) {
        x.right.parent = y;
    }

    x.parent = y.parent;

    if (!y.parent) {
        this.root = x;
    } else if (y === y.parent.left) {

```

```

        y.parent.left = x;
    } else {
        y.parent.right = x;
    }

    x.right = y;
    y.parent = x;
}

```

4. Red-Black Tree Operations

4.1 Insertion

Insertion involves adding a new node and then fixing any violations of Red-Black properties.

```

insert(value: T) {
    const newNode = new TreeNode(value,
    Color.RED);
    this.root = this.insertNode(this.root,
    newNode);
    this.fixViolation(newNode);
}

```

4.2 Deletion

Deletion is more complex and involves fixing the tree to maintain Red-Black properties after removing a node.

This section would outline the logic for handling different cases during deletion.

```

delete(value: T) {
    // Implementation for deletion, followed by
    fix-ups
}

```

5. Use Cases of Red-Black Trees

Red-Black Trees are commonly used in:

- **Balanced associative containers:** Such as map and set in the C++ STL.
- **OS Scheduling:** Used in process scheduling in operating systems.
- **Database indexing:** Red-Black Trees are used in scenarios where data insertion, deletion, and lookups are frequent, ensuring logarithmic time operations.

6. Conclusion

Red-Black Trees are a robust, self-balancing tree structure that ensures efficient operations while maintaining balance through color-coding and rotations. Understanding Red-Black Trees is crucial for implementing efficient algorithms in various applications.

B-Trees

About B-Trees

A **B-Tree** is a self-balancing search tree designed to maintain sorted data and allow efficient insertion, deletion, and search operations. Unlike binary trees, a B-Tree can have more than two children per node, making it well-suited for systems where reading and writing large blocks of data is critical, such as databases and file systems.

The B-Tree was introduced by Rudolf Bayer and Edward M. McCreight in 1972 and is characterized by its ability to minimize the number of disk accesses, thanks to its balanced and wide structure.

Key properties of a B-Tree:

1. **Every node has a maximum of m children (where m is the order of the tree).**

2. **The number of keys in a node is one less than the number of its children.**
3. **All leaves are at the same depth.**
4. **Keys are stored in sorted order.**
5. **Nodes split when they exceed their maximum capacity.**

Structure of a B-Tree Node

A B-Tree node contains:

- **Keys:** An ordered list of keys.
- **Children:** Pointers to child nodes.
- **Leaf status:** A flag indicating whether the node is a leaf or internal.

Example of a node in TypeScript:

```
class BTreeNode<T> {  
    keys: T[]; // List of keys in the node  
    children: BTreeNode<T>[]; // List of child  
nodes  
    isLeaf: boolean; // Indicates if the node is  
a leaf  
  
    constructor(isLeaf: boolean) {  
        this.keys = [];  
        this.children = [];  
        this.isLeaf = isLeaf;  
    }  
}
```

Basic Logic of a B-Tree

1. **Search:** Begin at the root. Compare the target key with the keys in the node. If it's not found

and the node is internal, follow the pointer to the child where the key would belong. Repeat until found or a leaf is reached.

2. **Insertion:** Keys are added in sorted order. If a node overflows (exceeds its maximum key capacity), split the node and propagate the middle key upward.
3. **Deletion:** Keys are removed in such a way that the B-Tree properties are preserved. If a node underflows (has fewer keys than allowed), merge it with a sibling or borrow from a sibling.

Example of B-Tree Operations in TypeScript

Below is a basic implementation of a B-Tree with **insertion** and **search** operations.

```
class BTree<T> {
  private root: BTreeNode<T>;
  private maxKeys: number;

  constructor(order: number) {
    this.root = new BTreeNode<T>(true);
    this.maxKeys = order - 1; // Maximum keys
    in a node
  }

  // Search for a key in the B-Tree
  search(key: T): BTreeNode<T> | null {
    return this._search(this.root, key);
  }

  private _search(node: BTreeNode<T>, key: T):
    BTreeNode<T> | null {
```

```

    let i = 0;

    // Find the index of the first key greater
    than or equal to the target
    while (i < node.keys.length && key >
node.keys[i]) {
        i++;
    }

    // If the key matches, return the node
    if (i < node.keys.length && key ===
node.keys[i]) {
        return node;
    }

    // If it's a leaf, the key is not in the
tree
    if (node.isLeaf) {
        return null;
    }

    // Otherwise, descend to the appropriate
child
    return      this._search(node.children[i],
key);
}

// Insert a key into the B-Tree
insert(key: T): void {
    const root = this.root;

    // If root is full, split it and create a
new root

```

```

        if (root.keys.length === this.maxKeys) {
            const newRoot = new BTreeNode<T>(false);
            newRoot.children.push(this.root);
            this._splitChild(newRoot, 0);
            this.root = newRoot;
        }

        this._insertNonFull(this.root, key);
    }

    private _insertNonFull(node: BTreeNode<T>,
key: T): void {
        let i = node.keys.length - 1;

        if (node.isLeaf) {
            // Insert key in sorted order
            node.keys.push(key);
            node.keys.sort((a, b) => (a > b ? 1 : -
1));
        } else {
            // Find the child to insert into
            while (i >= 0 && key < node.keys[i]) {
                i--;
            }
            i++;

            // Split the child if it's full
            if (node.children[i].keys.length ===
this.maxKeys) {
                this._splitChild(node, i);

                // Decide which child to descend into
                if (key > node.keys[i]) {

```

```

        i++;
    }
}

this._insertNonFull(node.children[i],
key);
}
}

private _splitChild(parent: BTreeNode<T>,
index: number): void {
    const nodeToSplit =
parent.children[index];
    const midIndex = Math.floor(this.maxKeys /
2);

    const newNode = new
BTreeNode<T>(nodeToSplit.isLeaf);

    // Move the second half of keys and
children to the new node
    newNode.keys =
nodeToSplit.keys.splice(midIndex + 1);
    if (!nodeToSplit.isLeaf) {
        newNode.children =
nodeToSplit.children.splice(midIndex + 1);
    }

    // Insert the middle key into the parent
    parent.keys.splice(index, 0,
nodeToSplit.keys[midIndex]);
    parent.children.splice(index + 1, 0,
newNode);
}

```

```

        // Remove the middle key from the original
node
        nodeToSplit.keys.splice(midIndex, 1);
    }
}

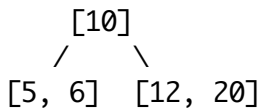
// Example usage
const bTree = new BTree<number>(3); // B-Tree
of order 3
bTree.insert(10);
bTree.insert(20);
bTree.insert(5);
bTree.insert(6);
bTree.insert(12);

console.log(bTree.search(10)); // Should find
the node containing 10
console.log(bTree.search(15)); // Should
return null

```

Visual Example of a B-Tree

For a B-Tree of order 3, inserting the keys [10, 20, 5, 6, 12] produces the following structure:



Conclusion

B-Trees are an essential data structure for managing large datasets efficiently, especially in applications like databases and file systems. Their ability to remain

balanced ensures that search, insertion, and deletion operations maintain optimal performance.

In this chapter, we explored the structure, logic, and implementation of B-Trees in TypeScript. By understanding these concepts, developers can harness the power of B-Trees for building scalable and high-performance systems. The TypeScript implementation provided here serves as a foundation that you can expand upon to support advanced features like deletion or persistent storage integration.

Heaps

About Heaps

A **Heap** is a specialized tree-based data structure that satisfies the **heap property**:

1. **Max-Heap Property**: The value of a parent node is greater than or equal to the values of its children.
2. **Min-Heap Property**: The value of a parent node is less than or equal to the values of its children.

Heaps are commonly used to implement priority queues, where the highest (or lowest) priority element is accessed efficiently. Heaps are also foundational for algorithms like **Heap Sort** and solving graph problems like Dijkstra's shortest path.

Structure of a Heap

A heap is usually represented as a **binary tree**, where:

- Each node can have at most two children.
- The tree is a **complete binary tree** (all levels are fully filled except possibly the last, which is filled from left to right).

For efficient storage, heaps are often implemented using **arrays**, where:

- The root is at index 0.
- The left child of a node at index i is at $2i + 1$.
- The right child is at $2i + 2$.
- The parent of a node at index i is at $(i - 1) // 2$.

Heap Operations

1. **Insertion:** Insert a new element at the end of the heap and "heapify up" to maintain the heap property.
2. **Deletion (Extract):** Remove the root element (maximum in max-heap or minimum in min-heap), replace it with the last element, and "heapify down" to restore the heap property.
3. **Peek:** Retrieve the root element without removing it.
4. **Heapify:** Transform an unsorted array into a heap.

Example of Heap Implementation in TypeScript

Below is a simple implementation of a **Max-Heap**:

```
class MaxHeap {
  private heap: number[];

  constructor() {
    this.heap = [];
  }

  // Get the parent index
  private parentIndex(index: number): number {
    return Math.floor((index - 1) / 2);
  }

  // Get the left child index
  private leftChildIndex(index: number):
number {
    return 2 * index + 1;
  }
}
```



```

    // Get the right child index
    private rightChildIndex(index: number):
number {
        return 2 * index + 2;
    }

    // Swap two elements in the heap
    private swap(i: number, j: number): void {
        [this.heap[i], this.heap[j]] =
[this.heap[j], this.heap[i]];
    }

    // Insert a new value into the heap
    insert(value: number): void {
        this.heap.push(value);
        this.heapifyUp(this.heap.length - 1);
    }

    // Restore heap property by moving up
    private heapifyUp(index: number): void {
        let currentIndex = index;
        while (
            currentIndex > 0 &&
            this.heap[currentIndex] >
this.heap[this.parentIndex(currentIndex)]
        ) {
            this.swap(currentIndex,
this.parentIndex(currentIndex));
            currentIndex =
this.parentIndex(currentIndex);
        }
    }

```

```

    // Extract the maximum value (root) from the
    heap
    extractMax(): number | null {
        if (this.heap.length === 0) return null;

        const max = this.heap[0];
        this.heap[0] = this.heap.pop()!;
        this.heapifyDown(0);

        return max;
    }

    // Restore heap property by moving down
    private heapifyDown(index: number): void {
        let largest = index;
        const left = this.leftChildIndex(index);
        const right = this.rightChildIndex(index);

        if (left < this.heap.length &&
            this.heap[left] > this.heap[largest]) {
            largest = left;
        }

        if (right < this.heap.length &&
            this.heap[right] > this.heap[largest]) {
            largest = right;
        }

        if (largest !== index) {
            this.swap(index, largest);
            this.heapifyDown(largest);
        }
    }

```

```

    }

    // Peek at the maximum value without removing
    it
    peek(): number | null {
        return this.heap.length > 0 ? this.heap[0]
: null;
    }

    // Get the current state of the heap
    getHeap(): number[] {
        return this.heap;
    }
}

// Example usage
const maxHeap = new MaxHeap();
maxHeap.insert(10);
maxHeap.insert(20);
maxHeap.insert(5);
maxHeap.insert(7);

console.log(maxHeap.getHeap()); // [20, 10, 5,
7]
console.log(maxHeap.extractMax()); // 20
console.log(maxHeap.getHeap()); // [10, 7, 5]
console.log(maxHeap.peek()); // 10

```

Example: Heapify an Array

To transform an array into a heap (heapify), use a bottom-up approach:

```

function heapify(array: number[]): number[] {

```

```

    const n = array.length;

    // Start from the last non-leaf node and move
    up
    for (let i = Math.floor(n / 2) - 1; i >= 0; i--) {
        heapifyDown(array, n, i);
    }

    return array;
}

function heapifyDown(array: number[], n:
number, i: number): void {
    let largest = i;
    const left = 2 * i + 1;
    const right = 2 * i + 2;

    if (left < n && array[left] > array[largest])
    {
        largest = left;
    }

    if (right < n && array[right] >
array[largest]) {
        largest = right;
    }

    if (largest !== i) {
        [array[i], array[largest]] =
[array[largest], array[i]];
        heapifyDown(array, n, largest);
    }
}

```

```
}
```

```
// Example usage  
const arr = [3, 5, 1, 10, 2];  
console.log(heapify(arr)); // [10, 5, 1, 3, 2]
```

Conclusion

Heaps are powerful data structures that provide efficient ways to manage and retrieve priority-based elements. Whether it's for priority queues, heapsort, or other advanced algorithms, the heap's ability to maintain a structured order is essential.

This chapter covered the basic logic, implementation, and common operations of heaps in TypeScript. By understanding heaps, developers can handle complex problems with ease and build efficient systems. The provided TypeScript examples demonstrate how to implement a Max-Heap and heapify an array, which are foundational concepts in computer science.

Hashing

About Hashing

Hashing is a technique used in computer science to map data (like strings or numbers) to a fixed-size value, called a **hash code** or **hash value**, using a mathematical function known as a **hash function**. Hashing is widely used in various applications, such as:

- **Hash Tables:** Storing and retrieving data efficiently.
- **Cryptography:** Securing data with hash functions.
- **Data Deduplication:** Identifying duplicate data efficiently.
- **Checksums:** Verifying data integrity.

The primary goal of hashing is to provide fast and efficient access to data by minimizing the time complexity to $O(1)$ for search, insert, and delete operations in the best case.

Hash Tables in TypeScript

What is a Hash Table?

A **hash table** is a data structure that maps keys to values. It uses a **hash function** to compute an index for each key, storing the key-value pair at that index in an array. This allows for efficient retrieval, insertion, and deletion of elements with an average time complexity of $O(1)$.

Hash tables are widely used in scenarios like caching, database indexing, and associative arrays.

How Hash Tables Work

1. **Hashing the Key:** A hash function converts the key into an index within the bounds of the array.
2. **Storing Data:** The value is stored at the computed index.
3. **Retrieving Data:** The key is hashed again to compute the index, and the value is retrieved from the array.
4. **Handling Collisions:** If two keys produce the same index (a collision), the hash table uses techniques like chaining or open addressing to resolve it.

Features of Hash Tables

- **Efficiency:** Fast lookups, inserts, and deletes.
- **Flexibility:** Works with any hashable key type.
- **Collision Handling:** Resolves hash conflicts using strategies like chaining or open addressing.

Hash Table Implementation in TypeScript

Below is an example implementation of a hash table in TypeScript:

```
class HashTable<K, V> {  
  private table: [K, V][][];  
  private size: number;  
  
  constructor(size: number = 10) {  
    this.table = Array.from({ length: size },  
      () => []);  
    this.size = size;  
  }  
  
  // Hash function to calculate index
```

```

private hash(key: K): number {
  const stringKey = String(key);
  let hash = 0;

  for (let i = 0; i < stringKey.length; i++)
  {
    hash = (hash + stringKey.charCodeAt(i) *
i) % this.size;
  }

  return hash;
}

// Insert or update a key-value pair
set(key: K, value: V): void {
  const index = this.hash(key);
  const bucket = this.table[index];

  for (const [existingKey, existingValue] of
bucket) {
    if (existingKey === key) {
      existingValue = value; // Update the
value
      return;
    }
  }

  bucket.push([key, value]); // Add a new
key-value pair
}

// Retrieve a value by its key
get(key: K): V | undefined {

```



```

    const index = this.hash(key);
    const bucket = this.table[index];

    for (const [existingKey, value] of bucket)
    {
        if (existingKey === key) {
            return value;
        }
    }

    return undefined; // Key not found
}

// Remove a key-value pair
delete(key: K): void {
    const index = this.hash(key);
    const bucket = this.table[index];

    this.table[index]
    bucket.filter(([,existingKey]) => existingKey
    !== key);
}

// Print the entire hash table
print(): void {
    console.log(this.table);
}
}

// Example usage
const hashTable = new HashTable<string,
number>();
hashTable.set("apple", 10);

```

```

hashTable.set("banana", 20);
hashTable.set("grape", 30);

console.log(hashTable.get("banana"));          //
Output: 20
hashTable.delete("banana");
console.log(hashTable.get("banana"));          //
Output: undefined
hashTable.print();

```

Collision Handling in Hash Tables

In the implementation above, **chaining** is used to handle collisions:

- Each index in the table contains a **bucket** (an array of key-value pairs).
- When a collision occurs, the conflicting key-value pair is added to the bucket at that index.

Alternative collision handling methods include:

1. **Open Addressing:** Finding the next available slot in the array for the new key-value pair.
2. **Double Hashing:** Using a second hash function to compute a new index for collisions.

Dynamic Resizing of Hash Tables

When the hash table becomes too full (high **load factor**), performance may degrade. A common solution is to resize the hash table:

```

resize(newSize: number): void {
  const oldTable = this.table;
  this.size = newSize;
  this.table = Array.from({ length: newSize },
    () => []);

```

```
    for (const bucket of oldTable) {  
      for (const [key, value] of bucket) {  
        this.set(key, value); // Rehash and  
        reinsert elements  
      }  
    }  
  }  
}
```

Conclusion

Hash tables are one of the most versatile and efficient data structures in computer science. They offer fast lookups, updates, and deletions, making them ideal for a wide range of applications. This chapter covered the basics of hash tables, an implementation in TypeScript, collision handling techniques, and practical examples. Mastering hash tables equips developers with a powerful tool for solving real-world problems with efficiency and simplicity.

Collision Resolution Techniques

What is a Collision in Hash Tables?

In a hash table, a **collision** occurs when two or more keys are hashed to the same index. Since each index in the table is designed to hold one key-value pair, collisions must be resolved to maintain the integrity and functionality of the hash table.

Efficient **collision resolution techniques** are essential for hash tables to perform well, even when the table becomes crowded or the hash function is imperfect.

Collision Resolution Techniques

Here are the most common techniques for resolving collisions in hash tables:

Collision Resolution Techniques

What is a Collision in Hash Tables?

In a hash table, a **collision** occurs when two or more keys are hashed to the same index. Since each index in the table is designed to hold one key-value pair, collisions must be resolved to maintain the integrity and functionality of the hash table.

Efficient **collision resolution techniques** are essential for hash tables to perform well, even when the table becomes crowded or the hash function is imperfect.

Collision Resolution Techniques

Here are the most common techniques for resolving collisions in hash tables:

1. Separate Chaining

Separate chaining involves storing multiple key-value pairs in a **bucket** at the same index. Each bucket is typically implemented as a linked list, array, or other collection.

How it Works

- Each index of the hash table points to a list (or chain).
- When a collision occurs, the key-value pair is appended to the chain at that index.
- During retrieval, the key is searched within the chain.

Advantages

- Simple to implement.
- Handles collisions efficiently, even when the load factor is high.

Disadvantages

- Memory overhead for maintaining additional data structures.
- Lookup time can increase if chains grow too long.

Example in TypeScript

```
class HashTableWithChaining<K, V> {  
  private table: [K, V][][];  
  private size: number;  
  
  constructor(size: number = 10) {  
    this.table = Array.from({ length: size },  
      () => []);  
    this.size = size;  
  }  
  
  private hash(key: K): number {
```

```

    const stringKey = String(key);
    let hash = 0;

    for (let i = 0; i < stringKey.length; i++)
    {
        hash = (hash + stringKey.charCodeAt(i) *
i) % this.size;
    }

    return hash;
}

set(key: K, value: V): void {
    const index = this.hash(key);
    const bucket = this.table[index];

    for (const [existingKey, _] of bucket) {
        if (existingKey === key) {
            return; // Update existing value if the
key exists
        }
    }

    bucket.push([key, value]);
}

get(key: K): V | undefined {
    const index = this.hash(key);
    const bucket = this.table[index];

    for (const [existingKey, value] of bucket)
    {
        if (existingKey === key) {

```

```

        return value;
    }
}

return undefined;
}
}

```

2. Open Addressing

In **open addressing**, collisions are resolved by finding another open slot in the table to store the new key-value pair. No additional data structures are used.

Types of Open Addressing

1. **Linear Probing**: Search sequentially for the next available slot.
2. **Quadratic Probing**: Search slots by an increasing quadratic offset.
3. **Double Hashing**: Use a secondary hash function to calculate the next slot.

Linear Probing Example

- If the hash function computes an index that's already occupied, try the next index $(i+1) \bmod \text{table size}$.

```

class HashTableWithLinearProbing<K, V> {
    private table: (null | [K, V])[];
    private size: number;

    constructor(size: number = 10) {
        this.table = Array.from({ length: size },
            () => null);
    }
}

```

```

    this.size = size;
}

private hash(key: K): number {
    const stringKey = String(key);
    let hash = 0;

    for (let i = 0; i < stringKey.length; i++)
    {
        hash = (hash + stringKey.charCodeAt(i) *
i) % this.size;
    }

    return hash;
}

set(key: K, value: V): void {
    let index = this.hash(key);

    while (this.table[index] !== null) {
        index = (index + 1) % this.size; //
Linear probing
    }

    this.table[index] = [key, value];
}

get(key: K): V | undefined {
    let index = this.hash(key);

    while (this.table[index] !== null) {
        const [storedKey, storedValue] =
this.table[index]!;

```



```

        if (storedKey === key) {
            return storedValue;
        }
        index = (index + 1) % this.size; //
Linear probing
    }

    return undefined;
}
}

```

Advantages

- No additional memory overhead for chains.
- Can achieve better cache performance due to contiguous memory access.

Disadvantages

- Clustering: Nearby slots can get filled, leading to longer probe sequences.
- Deletion can be complex, requiring lazy deletion or special markers.

3. Double Hashing

Double hashing uses a second hash function to calculate the step size for probing. If a collision occurs, the next slot is calculated as:

$$\text{index} = (\text{hash1}(\text{key}) + i \cdot \text{hash2}(\text{key})) \bmod \text{table size}$$

or

$$\text{index} = (\text{hash1}(\text{key}) + i \cdot \text{hash2}(\text{key})) \bmod \text{table size}$$

Advantages

- Reduces clustering compared to linear probing.
- Efficient distribution of keys across the table.

Disadvantages

- Slightly more complex to implement.
- Requires a well-designed secondary hash function.

4. Resizing the Hash Table

Resizing the table is not a collision resolution technique itself, but it reduces the likelihood of collisions. When the load factor exceeds a threshold, the table is resized (usually doubled) and all elements are rehashed into the new table.

Comparison of Techniques

Technique	Memory Overhead	Performance (Average)	Ease of Implementation	Notes
Separate Chaining	High	$O(1)O(1)O(1)$	Easy	Efficient for high load factors.
Linear Probing	Low	$O(1)O(1)O(1)$	Moderate	Prone to clustering.
Double Hashing	Low	$O(1)O(1)O(1)$	Complex	Requires careful hash function design.

Conclusion

Collision resolution is a critical aspect of hash table design, directly impacting its performance and reliability. Separate chaining and open addressing (with variations like linear probing and double hashing) are widely used techniques, each with unique trade-offs in memory usage and performance. Understanding these techniques ensures developers can implement hash tables effectively, tailoring them to the specific needs of their applications.

Graphs

Graphs are fundamental data structures used to model relationships between objects. They are widely used in various fields, including computer science, biology, transportation networks, social media, and more.

What is a Graph?

A graph G consists of:

1. **Vertices (Nodes):** Represent the entities or points in the graph. $V = \{v_1, v_2, \dots, v_n\}$
2. **Edges:** Represent connections between vertices. $E = \{e_1, e_2, \dots, e_m\}$

A graph is represented as $G = (V, E)$.

Types of Graphs

1. **Directed Graph (Digraph):** Edges have a direction, indicating a one-way relationship.
 - Example: Twitter (user A follows user B).
2. **Undirected Graph:** Edges do not have a direction, indicating a two-way relationship.
 - Example: Facebook friendships.
3. **Weighted Graph:** Edges have weights representing costs, distances, or other metrics.
 - Example: Road networks with distances.
4. **Unweighted Graph:** Edges have no weights.
 - Example: Simple connectivity graphs.
5. **Cyclic Graph:** Contains at least one cycle.
6. **Acyclic Graph:** Contains no cycles.

- A directed acyclic graph (DAG) is used in scenarios like task scheduling.

Applications of Graphs

1. **Social Networks:** Representing connections between users.
2. **Maps and Navigation:** Modeling road networks and finding shortest paths.
3. **Task Scheduling:** Using directed acyclic graphs (DAGs) to manage dependencies.
4. **Web Crawling:** Representing the internet as a graph of web pages and links.
5. **Biology:** Modeling gene interactions or protein structures.

Conclusion

Graphs are versatile data structures that excel in modeling relationships between entities. Whether represented as adjacency lists for efficiency or adjacency matrices for simplicity, graphs play a crucial role in solving complex problems across diverse domains. Mastering graph representations and algorithms like BFS and DFS equips developers to tackle challenges in networking, navigation, data science, and beyond.

Graph Representations in TypeScript

Graphs can be represented in multiple ways depending on the requirements of the application. In TypeScript, the most common graph representations are **Adjacency Matrix** and **Adjacency List**. Each method has its own

strengths and weaknesses, which make them suitable for different types of problems.

1. Adjacency Matrix

An adjacency matrix is a 2D array where rows and columns represent vertices. A value of 1 (or the weight, in the case of weighted graphs) indicates that an edge exists between two vertices, while a value of 0 indicates no edge.

Key Features

- Best for dense graphs where most vertices are connected.
- Easy to check if two vertices are connected.
- Consumes $O(V^2)$ space, where V is the number of vertices.

Implementation in TypeScript

```
class GraphMatrix {
  private matrix: number[][];

  constructor(size: number) {
    this.matrix = Array.from({ length: size },
    () => Array(size).fill(0));
  }

  addEdge(v1: number, v2: number, weight:
number = 1): void {
    this.matrix[v1][v2] = weight;
    this.matrix[v2][v1] = weight; // For
undirected graphs
  }

  removeEdge(v1: number, v2: number): void {
    this.matrix[v1][v2] = 0;
    this.matrix[v2][v1] = 0; // For undirected
graphs
  }

  display(): void {
    console.table(this.matrix);
  }
}

// Example usage
const graph = new GraphMatrix(4);
graph.addEdge(0, 1);
graph.addEdge(1, 2);
graph.addEdge(2, 3);
graph.display();
```

2. Adjacency List

An adjacency list represents a graph using a collection of lists. Each vertex maintains a list of all vertices it is connected to. This representation is more space-efficient for sparse graphs.

Key Features

- Best for sparse graphs where most vertices are not connected.
- Requires less space: $O(V+E)$ $O(V + E)$ $O(V+E)$, where E is the number of edges.
- Easier to traverse neighbors of a vertex.

Implementation in TypeScript

```
class GraphList {
  private adjacencyList: Map<number,
number[]>;

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

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

  addEdge(v1: number, v2: number): void {
    this.adjacencyList.get(v1)?.push(v2);
    this.adjacencyList.get(v2)?.push(v1);    //
  }
}
```

For undirected graphs


```

    }

    removeEdge(v1: number, v2: number): void {
        this.adjacencyList.set(v1,
            (this.adjacencyList.get(v1) || []).filter(v =>
                v !== v2));
        this.adjacencyList.set(v2,
            (this.adjacencyList.get(v2) || []).filter(v =>
                v !== v1));
    }

    display(): void {
        for (const [vertex, neighbors] of
            this.adjacencyList.entries()) {
            console.log(`${vertex}           ->
                ${neighbors.join(', ')}\`);
        }
    }
}

```

```

// Example usage
const graph = new GraphList();
graph.addVertex(0);
graph.addVertex(1);
graph.addVertex(2);
graph.addVertex(3);
graph.addEdge(0, 1);
graph.addEdge(1, 2);
graph.addEdge(2, 3);
graph.display();

```

3. Edge List

An edge list is a simple representation where all edges of the graph are stored as a list of pairs (or tuples). Each pair indicates a connection between two vertices and optionally includes a weight.

Key Features

- Space-efficient for small graphs or graphs with very few edges.
- Less efficient for graph traversal.

Implementation in TypeScript

```
class GraphEdgeList {
  private edges: [number, number, number?][];
  // [vertex1, vertex2, weight?]

  constructor() {
    this.edges = [];
  }

  addEdge(v1: number, v2: number, weight:
number = 1): void {
    this.edges.push([v1, v2, weight]);
  }

  removeEdge(v1: number, v2: number): void {
    this.edges = this.edges.filter(edge =>
edge[0] !== v1 || edge[1] !== v2);
  }

  display(): void {
    console.log(this.edges);
  }
}
```

```
// Example usage
const graph = new GraphEdgeList();
graph.addEdge(0, 1, 10);
graph.addEdge(1, 2, 15);
graph.addEdge(2, 3, 20);
graph.display();
```

Comparison of Graph Representations

Representation	Space Complexity	Edge Lookup Time	Best For
Adjacency Matrix	$O(V^2)$	$O(1)$	Dense graphs
Adjacency List	$O(V + E)$	$O(V)$	Sparse graphs
Edge List	$O(E)$	$O(E)$	Small or edge-centric graphs

Conclusion

Graph representations are vital for working with data structures efficiently. The choice of representation depends on the graph's density and the operations required. Adjacency matrices are ideal for dense graphs, adjacency lists for sparse graphs, and edge lists for compact storage or edge-specific operations. In TypeScript, implementing and manipulating these representations enables developers to model and solve complex real-world problems effectively.

Graph Traversal (BFS, DFS) in TypeScript

Graph Traversal in TypeScript (BFS and DFS)

Graph traversal is the process of visiting all the nodes (vertices) in a graph systematically. There are two main graph traversal techniques: **Breadth-First Search (BFS)** and **Depth-First Search (DFS)**. These techniques are fundamental for solving problems like pathfinding, cycle detection, and connectivity checks.

1. Breadth-First Search (BFS)

BFS explores the graph layer by layer. Starting from a source node, it visits all its neighbors before moving to their neighbors.

Key Features

- Uses a **queue** for traversal.
- Suitable for finding the shortest path in an unweighted graph.
- Explores nodes in increasing order of their distance from the source.
-

TypeScript Implementation of BFS

```
function bfs(graph: Map<number, number[]>, start: number): void {  
    const visited = new Set<number>();  
    const queue: number[] = [start];  
  
    console.log("BFS Traversal:");  
    while (queue.length > 0) {  
        const node = queue.shift()!; // Remove the first element from the queue  
        if (!visited.has(node)) {  
            console.log(node); // Process the node  
            visited.add(node);  
        }  
    }  
}
```

```

        // Add unvisited neighbors to the queue
        queue.push(...(graph.get(node) ||
[]).filter(neighbor
=>
!visited.has(neighbor)));
    }
}
}

```

// Example usage

```

const graphBFS = new Map<number, number[]>([
  [0, [1, 2]],
  [1, [0, 3, 4]],
  [2, [0, 4]],
  [3, [1, 5]],
  [4, [1, 2, 5]],
  [5, [3, 4]],
]);

```

```

bfs(graphBFS, 0);

```

2. Depth-First Search (DFS)

DFS explores as far as possible along each branch before backtracking. It dives deep into the graph and backtracks when no unvisited neighbors remain.

Key Features

- Uses a **stack** for iterative implementation or recursion for simplicity.
- Suitable for detecting cycles, connected components, and pathfinding.
- Explores nodes in a depth-first manner.

TypeScript Implementation of DFS (Recursive)

```

function dfs(graph: Map<number, number[]>,
start: number, visited = new Set<number>()):
void {
    if (visited.has(start)) return;

    console.log(start); // Process the node
    visited.add(start);

    // Recurse for all unvisited neighbors
    for (const neighbor of graph.get(start) ||
[]) {
        if (!visited.has(neighbor)) {
            dfs(graph, neighbor, visited);
        }
    }
}

// Example usage
const graphDFS = new Map<number, number[]>([
    [0, [1, 2]],
    [1, [0, 3, 4]],
    [2, [0, 4]],
    [3, [1, 5]],
    [4, [1, 2, 5]],
    [5, [3, 4]],
]);

dfs(graphDFS, 0);

```

TypeScript Implementation of DFS (Iterative)

```
function dfsIterative(graph: Map<number,
number[]>, start: number): void {
    const visited = new Set<number>();
    const stack: number[] = [start];

    console.log("DFS Traversal:");
    while (stack.length > 0) {
        const node = stack.pop()!;
        if (!visited.has(node)) {
            console.log(node); // Process the node
            visited.add(node);
            // Add unvisited neighbors to the stack
            stack.push(...(graph.get(node)
                || []).filter(neighbor =>
                !visited.has(neighbor))));
        }
    }
}

// Example usage
dfsIterative(graphDFS, 0);
```

Comparison of BFS and DFS

Feature	BFS	DFS
Traversal Order	Level by level	Depth first, then backtrack
Data Structure	Queue	Stack or Recursion
Shortest Path	Yes (in unweighted graphs)	Not guaranteed
Memory Usage	High for wide graphs	High for deep graphs
Applications	Pathfinding, shortest path	Cycle detection, connected components

Applications of Graph Traversal

1. **Pathfinding:** Finding the shortest or any path between two nodes.
2. **Cycle Detection:** Checking if a graph contains cycles.
3. **Connectivity Check:** Determining if all nodes are connected.
4. **Topological Sorting:** Ordering nodes in a Directed Acyclic Graph (DAG).
5. **Maze Solving:** Exploring paths through a maze.

Conclusion

Graph traversal is a critical concept for exploring and analyzing relationships within a graph. BFS is ideal for level-order exploration and shortest paths, while DFS is powerful for deep exploration and cycle detection. By mastering BFS and DFS, developers can solve a wide

range of problems in networking, logistics, and computational theory. TypeScript's flexibility makes implementing and experimenting with these algorithms both intuitive and efficient.

Weighted Graphs (Dijkstra's, Floyd-Warshall) in TypeScript

Weighted graphs assign a weight or cost to each edge, often used in problems where distances, costs, or capacities are involved. Algorithms like **Dijkstra's** and **Floyd-Warshall** are fundamental for finding shortest paths in weighted graphs.

1. Dijkstra's Algorithm

Dijkstra's algorithm is used to find the shortest path from a source node to all other nodes in a graph with non-negative weights.

Key Features

- Uses a **priority queue** to pick the next node with the smallest tentative distance.
- Time complexity: $O((V+E)\log V)$ or $O((V + E) \log V)$, where V is the number of vertices and E is the number of edges.

TypeScript Implementation of Dijkstra's Algorithm

```
class WeightedGraph {  
  private adjacencyList: Map<number, [number,  
    number] []>;  
  
  constructor() {  
    this.adjacencyList = new Map();  
  }  
  
  addVertex(vertex: number): void {  
    if (!this.adjacencyList.has(vertex)) {  
      this.adjacencyList.set(vertex, []);  
    }  
  }  
}
```

```

    }
}

addEdge(v1: number, v2: number, weight:
number): void {
    this.adjacencyList.get(v1)?.push([v2,
weight]);
    this.adjacencyList.get(v2)?.push([v1,
weight]); // Undirected graph
}

dijkstra(start: number): Map<number, number>
{
    const distances = new Map<number,
number>();
    const priorityQueue: [number, number][] =
[]; // [distance, vertex]
    const visited = new Set<number>();

    // Initialize distances
    for (const vertex of
this.adjacencyList.keys()) {
        distances.set(vertex, Infinity);
    }
    distances.set(start, 0);

    priorityQueue.push([0, start]); //
[distance, vertex]

    while (priorityQueue.length > 0) {
        // Sort queue by distance
        priorityQueue.sort((a, b) => a[0] -
b[0]);
    }
}

```

```

        const [currentDistance, currentVertex] =
priorityQueue.shift()!;

        if (visited.has(currentVertex))
continue;
        visited.add(currentVertex);

        for (const [neighbor, weight] of
this.adjacencyList.get(currentVertex) || []) {
            const distance = currentDistance +
weight;
            if (distance <
(distances.get(neighbor) || Infinity)) {
                distances.set(neighbor, distance);
                priorityQueue.push([distance,
neighbor]);
            }
        }
    }
    return distances;
}
}

```

```

// Example usage
const graph = new WeightedGraph();
graph.addVertex(0);
graph.addVertex(1);
graph.addVertex(2);
graph.addVertex(3);
graph.addEdge(0, 1, 4);
graph.addEdge(0, 2, 1);
graph.addEdge(2, 1, 2);

```

```
graph.addEdge(1, 3, 1);
graph.addEdge(2, 3, 5);

const distances = graph.dijkstra(0);
console.log("Shortest distances:", distances);
```

2. Floyd-Warshall Algorithm

The Floyd-Warshall algorithm calculates the shortest paths between all pairs of nodes. It uses dynamic programming to iteratively improve paths by considering each node as an intermediate step.

Key Features

- Suitable for dense graphs or when all-pairs shortest paths are required.
- Time complexity: $O(V^3)$.

TypeScript Implementation of Floyd-Warshall Algorithm

```
function floydWarshall(graph: number[][]):
number[][] {
    const V = graph.length;
    const distances = Array.from({ length: V },
    (_, i) => [...graph[i]]);

    for (let k = 0; k < V; k++) {
        for (let i = 0; i < V; i++) {
            for (let j = 0; j < V; j++) {
                if (distances[i][k] + distances[k][j]
                < distances[i][j]) {
                    distances[i][j] = distances[i][k] +
                    distances[k][j];
                }
            }
        }
    }
}
```

```

    }
  }
}

return distances;
}

// Example usage
const INF = Infinity;
const graph = [
  [0, 4, INF, INF],
  [4, 0, 2, 1],
  [INF, 2, 0, 5],
  [INF, 1, 5, 0],
];

const shortestPaths = floydWarshall(graph);
console.log("All-pairs shortest paths:",
shortestPaths);

```

Comparison of Algorithms

Feature	Dijkstra's Algorithm	Floyd-Warshall Algorithm
Purpose	Single-source shortest path	All-pairs shortest paths
Time Complexity	$O((V+E)\log V)$ $O((V+E)\log V)$	$O(V^3)$
Graph Type	Sparse, directed/undirected	Dense, directed/undirected
Edge Weights	Non-negative	Works with positive or negative weights (no negative cycles)

Applications of Weighted Graph Algorithms

1. Dijkstra's Algorithm:

- GPS navigation systems.
- Network routing protocols like OSPF.
- Scheduling and resource allocation.

2. Floyd-Warshall Algorithm:

- Traffic and logistics optimization.
- Detecting the shortest paths between all nodes in a social network.
- Finding the transitive closure of a graph.

Conclusion

Weighted graph algorithms like Dijkstra's and Floyd-Warshall are indispensable tools in solving shortest path problems. Dijkstra's is efficient for single-source shortest paths, while Floyd-Warshall excels in calculating paths between all pairs of nodes. TypeScript's capabilities make implementing these algorithms intuitive, providing clear insights into their operation and practical applications.

Other Data Structures

Tries in TypeScript

Tries, also known as prefix trees, are tree-like data structures used to store strings efficiently. They are particularly useful for tasks like autocomplete, spell checking, and prefix-based search.

Key Features of Tries

- **Structure:** Each node represents a single character, and a path from the root to a node represents a prefix or a complete string.
- **Efficiency:** Operations like insertion, search, and prefix matching have time complexity proportional to the length of the string, i.e., $O(L)$, where L is the string length.
- **Applications:** Autocomplete systems, IP routing, and dictionary implementations.

1. Trie Data Structure

Each node in a trie has:

1. A mapping of children (to store the next possible characters).
2. A flag indicating whether the node marks the end of a word.

TypeScript Implementation of a Tries

```
class TrieNode {  
  children: Map<string, TrieNode>;
```



```

    isEndOfWord: boolean;

    constructor() {
        this.children = new Map();
        this.isEndOfWord = false;
    }
}

class Trie {
    private root: TrieNode;

    constructor() {
        this.root = new TrieNode();
    }

    // Insert a word into the trie
    insert(word: string): void {
        let current = this.root;

        for (const char of word) {
            if (!current.children.has(char)) {
                current.children.set(char, new
TrieNode());
            }
            current = current.children.get(char)!;
        }

        current.isEndOfWord = true;
    }

    // Search for a word in the trie
    search(word: string): boolean {
        let current = this.root;

```

```

    for (const char of word) {
        if (!current.children.has(char)) {
            return false;
        }
        current = current.children.get(char)!;
    }

    return current.isEndOfWord;
}

// Check if a prefix exists in the trie
startsWith(prefix: string): boolean {
    let current = this.root;

    for (const char of prefix) {
        if (!current.children.has(char)) {
            return false;
        }
        current = current.children.get(char)!;
    }

    return true;
}
}

```

```

// Example Usage
const trie = new Trie();
trie.insert("cat");
trie.insert("can");
trie.insert("car");
trie.insert("dog");

```

```

console.log(trie.search("cat")); // true
console.log(trie.search("bat")); // false
console.log(trie.startsWith("ca")); // true
console.log(trie.startsWith("do")); // true
console.log(trie.startsWith("bat")); // false

```

2. Use Cases of Tries

1. **Autocomplete:** Quickly find words that start with a given prefix.

```

class TrieWithAutocomplete extends Trie {
  // Find all words that start with a given
  prefix
  autocomplete(prefix: string): string[] {
    let current = this.root;

    // Navigate to the prefix node
    for (const char of prefix) {
      if (!current.children.has(char)) {
        return [];
      }
      current = current.children.get(char)!;
    }

    // Perform DFS to collect all words
    const results: string[] = [];
    const dfs = (node: TrieNode, path: string)
=> {
      if (node.isEndOfWord)
        results.push(path);
    }
  }
}

```

```

        for (const [char, child] of
node.children.entries()) {
            dfs(child, path + char);
        }
    };

    dfs(current, prefix);
    return results;
}
}

```

```

// Example Usage
const autoCompleteTrie = new
TrieWithAutocomplete();
autocompleteTrie.insert("cat");
autocompleteTrie.insert("car");
autocompleteTrie.insert("can");
autocompleteTrie.insert("cart");
autocompleteTrie.insert("dog");

console.log(autocompleteTrie.autocomplete("ca
")); // ["cat", "car", "can", "cart"]
console.log(autocompleteTrie.autocomplete("do
")); // ["dog"]
console.log(autocompleteTrie.autocomplete("ba
")); // []

```

2. **Spell Checking:** Validate words against a dictionary of valid words.
3. **IP Routing:** Match prefixes in network routing tables.

4. **Longest Prefix Matching:** Solve problems requiring prefix-based search.

3. Complexity Analysis

Operation	Time Complexity	Space Complexity
Insert a word	$O(L)O(L)O(L)$	$O(AL)O(AL)O(AL)$
Search for a word	$O(L)O(L)O(L)$	$O(AL)O(AL)O(AL)$
Prefix search	$O(L+N)O(L) + N)O(L+N)$	$O(AL)O(AL)O(AL)$

Where:

- LLL is the length of the word.
- AAA is the size of the alphabet.
- NNN is the number of words matching the prefix.

4. Advantages of Tries

- Efficient prefix-based search.
- Memory-efficient for a large number of short strings.
- No need to rehash or resize, unlike hash tables.

Conclusion

Tries are a powerful data structure for string manipulation and prefix-based operations. Their hierarchical design allows for efficient insertion, search, and prefix matching. By implementing tries in TypeScript, developers can harness their efficiency for real-world applications like autocomplete systems and spell checkers, making them an indispensable tool in modern software development.

Disjoint Sets in TypeScript

The **Disjoint Set Union (DSU)**, also known as the **Union-Find** data structure, is used to manage a collection of non-overlapping subsets. It supports two primary operations efficiently:

1. **Union:** Merge two subsets into one.
2. **Find:** Determine which subset a particular element belongs to.

Disjoint sets are particularly useful in:

- **Graph algorithms**, such as Kruskal's algorithm for finding minimum spanning trees.
- **Dynamic connectivity problems**, where determining whether two elements are in the same subset is required.

Key Concepts

1. **Path Compression:**
 - Optimizes the Find operation by flattening the structure of the tree whenever Find is called.
 - Ensures that all nodes point directly to the root.
2. **Union by Rank:**
 - Optimizes the Union operation by always attaching the smaller tree under the root of the larger tree.
 - Minimizes the height of the trees, improving performance.

TypeScript Implementation

```
class DisjointSet {
```

```

private parent: number[];
private rank: number[];

constructor(size: number) {
  this.parent = Array.from({ length: size },
    (_, i) => i); // Each node is its own parent
  this.rank = Array(size).fill(0); // Initial
rank is 0
}

// Find with path compression
find(x: number): number {
  if (this.parent[x] !== x) {
    this.parent[x] =
this.find(this.parent[x]); // Path compression
  }
  return this.parent[x];
}

// Union by rank
union(x: number, y: number): void {
  const rootX = this.find(x);
  const rootY = this.find(y);

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

```

```

        this.rank[rootX]++;
    }
}
}

// Check if two elements are in the same set
connected(x: number, y: number): boolean {
    return this.find(x) === this.find(y);
}
}

```

```

// Example Usage
const dsu = new DisjointSet(5);

```

```

// Union operations
dsu.union(0, 1);
dsu.union(1, 2);
dsu.union(3, 4);

```

```

console.log(dsu.connected(0, 2)); // true
console.log(dsu.connected(0, 4)); // false
console.log(dsu.connected(3, 4)); // true

```

```

console.log(dsu.find(2)); // Should return the
root of the set containing '2'

```

Example Use Case: Kruskal's Algorithm

Kruskal's algorithm finds the Minimum Spanning Tree (MST) of a graph by:

1. Sorting all edges by weight.
2. Adding edges to the MST, ensuring no cycles are formed (using the DSU to check connectivity).

TypeScript Implementation with DSU

```
interface Edge {
  source: number;
  destination: number;
  weight: number;
}

function kruskal(edges: Edge[], vertices:
number): Edge[] {
  // Sort edges by weight
  edges.sort((a, b) => a.weight - b.weight);

  const dsu = new DisjointSet(vertices);
  const mst: Edge[] = [];

  for (const edge of edges) {
    const { source, destination, weight } =
edge;

    // If the vertices are not connected, add
the edge to the MST
    if (!dsu.connected(source, destination)) {
      dsu.union(source, destination);
      mst.push(edge);
    }
  }

  return mst;
}

// Example graph
const edges: Edge[] = [
  { source: 0, destination: 1, weight: 10 },
```

```

    { source: 0, destination: 2, weight: 6 },
    { source: 0, destination: 3, weight: 5 },
    { source: 1, destination: 3, weight: 15 },
    { source: 2, destination: 3, weight: 4 },
];

```

```

const mst = kruskal(edges, 4);
console.log("Minimum Spanning Tree:", mst);

```

Complexity Analysis

Operation	Time Complexity	Description
Find	$O(\alpha(n))$	$\alpha(n)$ is the inverse Ackermann function.
Union	$O(\alpha(n))$	Extremely efficient due to path compression and rank.
Kruskal's	$O(E \log E + V \alpha(V))$	Sorting edges dominates time complexity.

Advantages of Disjoint Sets

- Efficiency:**
 - Operations are nearly constant time due to optimizations like path compression.
- Simplicity:**

- A minimalistic data structure, yet powerful for graph-related algorithms.

3. **Applications:**

- MST algorithms (Kruskal's).
- Connected components detection.
- Cycle detection in undirected graphs.

Conclusion

Disjoint Sets (Union-Find) provide a simple yet efficient way to manage non-overlapping subsets, making them indispensable in graph algorithms and connectivity problems. The combination of path compression and union by rank ensures operations remain fast, even for large datasets. With TypeScript, implementing this data structure becomes intuitive and robust for solving real-world problems.

Bloom Filters in TypeScript

A **Bloom Filter** is a probabilistic data structure designed to test whether an element is part of a set. It is compact, efficient, and particularly useful when the trade-off of allowing false positives is acceptable but false negatives are not.

How Bloom Filters Work

1. Structure:

- A Bloom Filter uses a fixed-size bit array initialized to zeros.
- It employs multiple hash functions to map data into positions in this array.

2. Adding an Element:

- Each hash function calculates a position in the bit array, and those positions are set to 1.

3. Checking Membership:

- For an element, the same hash functions calculate positions in the bit array.
- If all the positions are 1, the element might be in the set.
- If any position is 0, the element is definitely not in the set.

Pros and Cons of Bloom Filters

Advantages:

- Memory efficient compared to hash maps or traditional sets.
- Extremely fast insertions and lookups.

Disadvantages:

- Cannot remove elements (removal introduces errors).
- Allows false positives but guarantees no false negatives.

TypeScript Implementation

Class Implementation

```
class BloomFilter {
  private bitArray: Uint8Array;
  private size: number;
  private hashFunctions: ((item: string) =>
number)[];

  constructor(size: number, numHashFunctions:
number) {
    this.size = size;
    this.bitArray = new Uint8Array(size);
    this.hashFunctions = Array.from({ length:
numHashFunctions }, (_, i) => {
      return (item: string): number => {
        return this.hash(item, i) % this.size;
      };
    });
  }

  // Hash function generator
  private hash(item: string, seed: number):
number {
    let hash = 0;
    for (let i = 0; i < item.length; i++) {
      hash = (hash * seed + item.charCodeAt(i))
% this.size;
    }
    return hash;
  }
}
```

```

    // Add an item
    add(item: string): void {
        for (const hashFunction of
this.hashFunctions) {
            const index = hashFunction(item);
            this.bitArray[index] = 1;
        }
    }

    // Check membership
    mightContain(item: string): boolean {
        for (const hashFunction of
this.hashFunctions) {
            const index = hashFunction(item);
            if (this.bitArray[index] === 0) {
                return false; // Definitely not in the
set
            }
        }
        return true; // Might be in the set
    }
}

```

```

// Example Usage
const bloomFilter = new BloomFilter(100, 3);

bloomFilter.add("apple");
bloomFilter.add("banana");

console.log(bloomFilter.mightContain("apple")
); // true (likely)
console.log(bloomFilter.mightContain("grape")
); // false (definitely not)

```

```
console.log(bloomFilter.mightContain("banana"
)); // true (likely)
```

Example Use Case: Preventing Unnecessary Database Queries

Bloom Filters are ideal for reducing overhead in systems where repeated lookups can be costly, such as avoiding database queries for items that are definitely not in the dataset.

```
class CachedBloomFilter {
  private bloomFilter: BloomFilter;

  constructor(size: number, numHashFunctions:
number) {
    this.bloomFilter = new BloomFilter(size,
numHashFunctions);
  }

  addItem(item: string): void {
    console.log(`Adding    ${item}    to    the
filter.`);
    this.bloomFilter.add(item);
  }

  shouldQueryDatabase(item: string): boolean {
    const          mightContain          =
this.bloomFilter.mightContain(item);
    console.log(`${item} is ${mightContain ?
"possibly in the set" : "not in the set"}.`);
    return !mightContain; // Query database if
item is not in the filter
  }
}
```



```
}
```

```
// Example Usage
const cachedFilter = new
CachedBloomFilter(100, 3);

cachedFilter.addItem("user1");
cachedFilter.addItem("user2");

console.log(cachedFilter.shouldQueryDatabase(
"user3")); // true (not in set, should query
DB)
console.log(cachedFilter.shouldQueryDatabase(
"user1")); // false (likely in set, no need to
query DB)
```

Analysis of Bloom Filters

Operation	Time Complexity	Space Complexity
Add an element	$O(k)O(k)O(k)$	$O(m)O(m)O(m)$
Check membership	$O(k)O(k)O(k)$	$O(m)O(m)O(m)$

Where:

- kkk: Number of hash functions.
- mmm: Size of the bit array.

Conclusion

Bloom Filters are an excellent choice for membership testing when:

- Memory is constrained.
- False positives are acceptable but false negatives are not

The TypeScript implementation demonstrates how easily Bloom Filters can be integrated into modern applications. Their probabilistic nature, combined with efficiency, makes them an essential tool for developers working with large-scale systems, caching mechanisms, and network filters.

Algorithms

Sorting and Searching Algorithms in TypeScript

Sorting and searching are foundational techniques in computer science, often forming the basis for more complex algorithms. In this chapter, we'll explore popular sorting and searching algorithms and their implementation in TypeScript.

Sorting Algorithms

Sorting involves arranging data in a specific order (ascending or descending). Here's an overview of key algorithms:

1. Bubble Sort

Bubble Sort repeatedly compares adjacent elements and swaps them if they are in the wrong order.

Implementation

```
function bubbleSort(arr: number[]): number[] {  
  let n = arr.length;  
  for (let i = 0; i < n - 1; i++) {  
    for (let j = 0; j < n - i - 1; j++) {  
      if (arr[j] > arr[j + 1]) {  
        // Swap  
        [arr[j], arr[j + 1]] = [arr[j + 1],  
arr[j]];  
      }  
    }  
  }  
}
```

```
    return arr;
}
```

```
// Example
console.log(bubbleSort([64, 34, 25, 12, 22, 11,
90]));
```

2. Merge Sort

Merge Sort divides the array into halves, recursively sorts each half, and then merges them.

Implementation

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

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

    return merge(left, right);
}

function merge(left: number[], right:
number[]): number[] {
    const result: number[] = [];
    while (left.length && right.length) {
        if (left[0] < right[0]) {
            result.push(left.shift()!);
        } else {
            result.push(right.shift()!);
        }
    }
}
```

```
    return result.concat(left).concat(right);  
}
```

// Example

```
console.log(mergeSort([64, 34, 25, 12, 22, 11,  
90]));
```

3. Quick Sort

Quick Sort selects a "pivot" element, partitions the array around the pivot, and recursively sorts the partitions.

Implementation

```
function quickSort(arr: number[]): number[] {  
    if (arr.length <= 1) {  
        return arr;  
    }  
  
    const pivot = arr[arr.length - 1];  
    const left = arr.filter((el) => el < pivot);  
    const right = arr.filter((el) => el > pivot);  
  
    return [...quickSort(left), pivot,  
...quickSort(right)];  
}
```

// Example

```
console.log(quickSort([64, 34, 25, 12, 22, 11,  
90]));
```

Searching Algorithms

Searching involves locating an element in a dataset. Here are two common methods:

1. Linear Search

Linear Search checks each element one by one until the target is found.

Implementation

```
function linearSearch(arr: number[], target: number): number {
  for (let i = 0; i < arr.length; i++) {
    if (arr[i] === target) {
      return i;
    }
  }
  return -1;
}
```

// Example

```
console.log(linearSearch([10, 20, 30, 40, 50], 30)); // Output: 2
```

2. Binary Search

Binary Search works on sorted arrays by repeatedly dividing the search interval in half.

Implementation

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

```

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

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

return -1;
}

```

```

// Example
console.log(binarySearch([10, 20, 30, 40, 50],
30)); // Output: 2

```

Comparison of Algorithms

Algorithm	Best Case	Worst Case	Space Complexity	Stable
Bubble Sort	$O(n^2)$	$O(n^2)$	$O(1)$	Yes
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n)$	Yes
Quick Sort	$O(n \log n)$	$O(n^2)$	$O(\log n)$	No
Linear Search	$O(1)$	$O(n)$	$O(1)$	N/A
Binary Search	$O(1)$	$O(\log n)$	$O(1)$	N/A

Conclusion

Sorting and searching algorithms are vital for managing and querying data efficiently. By understanding their strengths and weaknesses, developers can choose the right approach for their use case. The TypeScript implementations provided offer practical insights and demonstrate the power of combining theory with real-world application.

Dynamic Programming in TypeScript

Dynamic Programming (DP) is a powerful technique used to solve complex problems by breaking them into smaller overlapping subproblems. It is particularly useful for optimization problems where solutions to subproblems can be reused.

Principles of Dynamic Programming

1. Optimal Substructure:

- A problem exhibits an optimal substructure if an optimal solution can be constructed from optimal solutions of its subproblems.

2. Overlapping Subproblems:

- The problem can be broken into smaller subproblems, and these subproblems are solved multiple times.

3. Memoization and Tabulation:

- **Memoization:** Top-down approach where results of subproblems are stored in a cache to avoid redundant computations.
- **Tabulation:** Bottom-up approach where solutions are built iteratively in a table.

Common Dynamic Programming Problems in TypeScript

1. Fibonacci Sequence

The Fibonacci sequence is a classic example of a DP problem.

Recursive with Memoization

```
function fibonacci(n: number, memo: Record<number, number> = {}): number {
  if (n <= 1) return n;
  if (memo[n] !== undefined) return memo[n];

  memo[n] = fibonacci(n - 1, memo) + fibonacci(n - 2, memo);
  return memo[n];
}
```

```
// Example
console.log(fibonacci(10)); // Output: 55
```

Tabulation

```
function fibonacciTab(n: number): number {
  if (n <= 1) return n;

  const dp = Array(n + 1).fill(0);
  dp[1] = 1;

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

  return dp[n];
}
```

```
// Example
console.log(fibonacciTab(10)); // Output: 55
```

2. Longest Common Subsequence (LCS)

The LCS problem finds the longest sequence present in both strings.

Implementation

```
function longestCommonSubsequence(str1:
string, str2: string): number {
    const m = str1.length;
    const n = str2.length;
    const dp: number[][] = Array.from({ length:
m + 1 }, () => Array(n + 1).fill(0));

    for (let i = 1; i <= m; i++) {
        for (let j = 1; j <= n; j++) {
            if (str1[i - 1] === str2[j - 1]) {
                dp[i][j] = dp[i - 1][j - 1] + 1;
            } else {
                dp[i][j] = Math.max(dp[i - 1][j],
dp[i][j - 1]);
            }
        }
    }

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

```
// Example
console.log(longestCommonSubsequence("abcde",
"ace")); // Output: 3
```

3. Knapsack Problem

The 0/1 Knapsack problem involves choosing items with given weights and values to maximize value without exceeding the weight limit.

Implementation

```
function knapsack(weights: number[], values:
number[], capacity: number): number {
  const n = weights.length;
  const dp = Array.from({ length: n + 1 }, ()
=> Array(capacity + 1).fill(0));

  for (let i = 1; i <= n; i++) {
    for (let w = 0; w <= capacity; w++) {
      if (weights[i - 1] <= w) {
        dp[i][w] = Math.max(dp[i - 1][w], dp[i
- 1][w - weights[i - 1]] + values[i - 1]);
      } else {
        dp[i][w] = dp[i - 1][w];
      }
    }
  }

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

// Example

```
console.log(knapsack([1, 2, 3], [10, 15, 40],
6)); // Output: 65
```

4. Minimum Coin Change

Given a set of coins, find the minimum number of coins needed to make a specific amount.

Implementation

```
function minCoins(coins: number[], amount:
number): number {
    const dp = Array(amount + 1).fill(Infinity);
    dp[0] = 0;

    for (let i = 1; i <= amount; i++) {
        for (const coin of coins) {
            if (i >= coin) {
                dp[i] = Math.min(dp[i], dp[i - coin] +
1);
            }
        }
    }

    return dp[amount] === Infinity ? -1 :
dp[amount];
}

// Example
console.log(minCoins([1, 2, 5], 11)); //
Output: 3 (5 + 5 + 1)
```

Analysis of Dynamic Programming

Problem	Time Complexity	Space Complexity
Fibonacci (Memoization)	$O(n)O(n)O(n)$	$O(n)O(n)O(n)$
Fibonacci (Tabulation)	$O(n)O(n)O(n)$	$O(n)O(n)O(n)$
Longest Common Subsequence	$O(m \times n)O(m \times n)O(m \times n)$	$O(m \times n)O(m \times n)O(m \times n)$
Knapsack Problem	$O(n \times W)O(n \times W)O(n \times W)$	$O(n \times W)O(n \times W)O(n \times W)$
Minimum Coin Change	$O(n \times S)O(n \times S)O(n \times S)$	$O(S)O(S)O(S)$

Conclusion

Dynamic Programming simplifies complex problems by leveraging overlapping subproblems and optimal substructures. By using either memoization or tabulation, developers can efficiently solve problems like the Fibonacci sequence, LCS, and Knapsack. TypeScript offers a great platform for implementing these algorithms, providing both clarity and performance for practical use cases.

Greedy Algorithms in TypeScript

Greedy algorithms are a class of algorithms that make locally optimal choices at each step with the hope of finding the global optimum. These algorithms are often used for optimization problems, and while they are not guaranteed to always produce the optimal solution, they are usually efficient and work well for many problems.

Principles of Greedy Algorithms

1. Greedy Choice Property:

- At each step, the algorithm makes the choice that seems best at the moment, aiming to build up the solution piece by piece.
- This choice is based on a heuristic that estimates the best path forward.

2. Optimal Substructure:

- A problem exhibits optimal substructure if an optimal solution to the problem can be constructed efficiently from optimal solutions to its subproblems.

3. Local vs Global Optimum:

- The greedy algorithm may only guarantee finding a local optimum at each step, but for some problems, this local optimum leads to the global optimum.

Common Greedy Algorithms in TypeScript

1. Activity Selection Problem

The Activity Selection Problem is a classic example where we select the maximum number of activities that don't overlap, given their start and end times.

Implementation

```
function activitySelection(start: number[],
end: number[]): number[] {
  const n = start.length;
  const result: number[] = [];

  // Sort activities based on their end times
  const activities = start.map((s, i) => ({
start: s, end: end[i], index: i }));
  activities.sort((a, b) => a.end - b.end);

  let lastSelected = -1;

  for (let i = 0; i < n; i++) {
    if (activities[i].start >=
(activities[lastSelected].end : 0)) {
      activities[lastSelected].end : 0)) {
        result.push(activities[i].index);
        lastSelected = i;
      }
    }
  }

  return result;
}
```

// Example

```
const start = [1, 3, 0, 5, 8, 5];
const end = [2, 4, 6, 7, 9, 9];
console.log(activitySelection(start, end)); //
Output: [0, 1, 3, 4]
```


2. Fractional Knapsack Problem

In the Fractional Knapsack problem, the goal is to maximize the value of the items that can fit into the knapsack, where you can take fractions of an item, unlike the 0/1 knapsack.

Implementation

```
interface Item {
  value: number;
  weight: number;
  ratio: number; // value/weight ratio
}

function fractionalKnapsack(capacity: number,
  items: Item[]): number {
  // Sort items based on value-to-weight ratio
  in descending order
  items.sort((a, b) => b.ratio - a.ratio);

  let totalValue = 0;

  for (const item of items) {
    if (capacity === 0) break;

    if (item.weight <= capacity) {
      // Take the whole item
      totalValue += item.value;
      capacity -= item.weight;
    } else {
      // Take a fraction of the item
      totalValue += item.value * (capacity /
item.weight);
      break;
    }
  }
}
```

```

    }
  }

  return totalValue;
}

// Example
const items: Item[] = [
  { value: 60, weight: 10, ratio: 60 / 10 },
  { value: 100, weight: 20, ratio: 100 / 20 },
  { value: 120, weight: 30, ratio: 120 / 30 },
];
const capacity = 50;
console.log(fractionalKnapsack(capacity,
items)); // Output: 240

```

3. Huffman Coding

Huffman Coding is a popular greedy algorithm used for data compression, where it builds an optimal binary tree for encoding a set of symbols based on their frequencies.

Implementation

```

class Node {
  character: string;
  freq: number;
  left: Node | null = null;
  right: Node | null = null;

  constructor(character: string, freq: number)
  {
    this.character = character;
    this.freq = freq;
  }
}

```

```
}  
}
```

```
function buildHuffmanTree(frequencies: {  
  [char: string]: number }): Node {  
  const nodes: Node[] =  
    Object.keys(frequencies).map(char => new  
      Node(char, frequencies[char]));
```

```
  while (nodes.length > 1) {  
    nodes.sort((a, b) => a.freq - b.freq);
```

```
    const left = nodes.shift()!;  
    const right = nodes.shift()!;
```

```
    const mergedNode = new Node("", left.freq  
+ right.freq);  
    mergedNode.left = left;  
    mergedNode.right = right;
```

```
    nodes.push(mergedNode);  
  }
```

```
  return nodes[0];  
}
```

```
function generateHuffmanCodes(root: Node,  
prefix: string = ""): { [char: string]: string  
} {  
  if (!root) return {};
```

```
  if (!root.left && !root.right) {  
    return { [root.character]: prefix };
```

```

    }

    return {
        ...generateHuffmanCodes(root.left, prefix +
"0"),
        ...generateHuffmanCodes(root.right, prefix
+ "1"),
    };
}

```

```

// Example
const frequencies = { 'A': 5, 'B': 9, 'C': 12,
'D': 13, 'E': 16, 'F': 45 };
const root = buildHuffmanTree(frequencies);
const codes = generateHuffmanCodes(root);
console.log(codes); // Output: Huffman codes
for characters

```

4. Prim's Algorithm for Minimum Spanning Tree (MST)

Prim's algorithm is a greedy algorithm that finds the minimum spanning tree of a connected graph, ensuring that all vertices are included in the tree while minimizing the edge weights.

Implementation

```

interface Edge {
    u: number;
    v: number;
    weight: number;
}

function primMST(vertices: number, graph:
Edge[]): number {

```

```

    const adj: number[][] = Array.from({ length:
vertices          },          ()          =>
Array(vertices).fill(Infinity));

    for (const edge of graph) {
        adj[edge.u][edge.v] = edge.weight;
        adj[edge.v][edge.u] = edge.weight;
    }

    const visited = Array(vertices).fill(false);
    const          minWeight          =
Array(vertices).fill(Infinity);
    minWeight[0] = 0;

    let totalWeight = 0;

    for (let i = 0; i < vertices; i++) {
        let u = -1;
        let min = Infinity;

        for (let j = 0; j < vertices; j++) {
            if (!visited[j] && minWeight[j] < min) {
                min = minWeight[j];
                u = j;
            }
        }

        if (u === -1) break;

        visited[u] = true;
        totalWeight += minWeight[u];

        for (let v = 0; v < vertices; v++) {

```

```

        if (!visited[v] && adj[u][v] <
minWeight[v]) {
            minWeight[v] = adj[u][v];
        }
    }
}

return totalWeight;
}

```

// Example

```

const edges: Edge[] = [
    { u: 0, v: 1, weight: 2 },
    { u: 0, v: 3, weight: 6 },
    { u: 1, v: 3, weight: 8 },
    { u: 1, v: 2, weight: 3 },
    { u: 2, v: 3, weight: 5 },
];
console.log(primMST(4, edges)); // Output: 16

```

Analysis of Greedy Algorithms

Problem	Time Complexity	Space Complexity
Activity Selection	$O(n \log n)$	$O(n)$
Fractional Knapsack	$O(n \log n)$	$O(1)$
Huffman Coding	$O(n \log n)$	$O(n)$
Prim's MST	$O(E \log V)$	$O(V)$

Conclusion

Greedy algorithms provide a simple and efficient approach to optimization problems by making the locally optimal choice at each step. Problems like Activity Selection, Fractional Knapsack, and Huffman Coding can be solved effectively with this technique. While they don't guarantee the best solution for every problem, they often provide a fast approximation with minimal computation.

Divide and Conquer in TypeScript

Divide and Conquer is a powerful algorithm design paradigm used for solving complex problems by breaking them down into smaller, more manageable subproblems. These subproblems are solved independently, and their solutions are combined to form the final solution. This approach is particularly useful when a problem can be divided into independent subproblems that are similar to the original problem but smaller in size.

Principles of Divide and Conquer

1. Divide:

- The problem is divided into smaller subproblems that are of the same type as the original problem.
- These subproblems are typically solved recursively.

2. Conquer:

- The subproblems are solved independently, often recursively.
- If the subproblem size is small enough, it can be solved directly without further subdivision.

3. Combine:

- Once the subproblems are solved, their solutions are combined to solve the original problem.
- The way the solutions are combined depends on the specific problem being solved.

Common Divide and Conquer Algorithms in TypeScript

1. Merge Sort

Merge Sort is a comparison-based sorting algorithm that follows the divide and conquer paradigm. It divides the array into two halves, recursively sorts each half, and then merges the two sorted halves.

Implementation

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

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

  return merge(left, right);
}

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

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

  return result.concat(left.slice(i), right.slice(j));
}
```

```

        j++;
    }
}

    return      result.concat(left.slice(i),
right.slice(j));
}

```

```

// Example
const arr = [38, 27, 43, 3, 9, 82, 10];
console.log(mergeSort(arr)); // Output: [3, 9,
10, 27, 38, 43, 82]

```

2. Quick Sort

Quick Sort is another sorting algorithm that works by selecting a pivot element, partitioning the array into two subarrays (one with elements less than the pivot and one with elements greater than the pivot), and then recursively sorting the subarrays.

Implementation

```

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

    const pivot = arr[arr.length - 1];
    const left = [];
    const right = [];

    for (let i = 0; i < arr.length - 1; i++) {
        if (arr[i] < pivot) left.push(arr[i]);
        else right.push(arr[i]);
    }
}

```

```
    return [...quickSort(left), pivot,
...quickSort(right)];
}
```

// Example

```
const arr = [10, 7, 8, 9, 1, 5];
console.log(quickSort(arr)); // Output: [1, 5,
7, 8, 9, 10]
```

3. Binary Search

Binary Search is an efficient algorithm for finding an item from a sorted array. The idea is to repeatedly divide the search interval in half. If the value of the search key is less than the item in the middle of the interval, the search continues in the left half, or if the value is greater, it continues in the right half.

Implementation

```
function binarySearch(arr: number[], target:
number): number {
    let low = 0;
    let high = arr.length - 1;

    while (low <= high) {
        const mid = Math.floor((low + high) / 2);

        if (arr[mid] === target) {
            return mid; // Element found
        } else if (arr[mid] < target) {
            low = mid + 1; // Search the right half
        } else {
            high = mid - 1; // Search the left half
        }
    }
}
```

```

    }

    return -1; // Element not found
}

// Example
const arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
console.log(binarySearch(arr, 7)); // Output:
6 (index of 7)

```

4. Strassen's Matrix Multiplication

Strassen's algorithm is an advanced divide and conquer algorithm that multiplies two matrices faster than the conventional matrix multiplication algorithm. It divides each matrix into four smaller submatrices and recursively calculates the result.

Implementation (Basic Outline)

```

function strassenMatrixMultiplication(A:
number[][], B: number[][]): number[][] {
    const n = A.length;

    if (n === 1) {
        return [[A[0][0] * B[0][0]]];
    }

    const mid = Math.floor(n / 2);
    const A11 = A.slice(0, mid).map(row =>
row.slice(0, mid));
    const A12 = A.slice(0, mid).map(row =>
row.slice(mid));
    const A21 = A.slice(mid).map(row =>
row.slice(0, mid));

```

```
    const A22 = A.slice(mid).map(row =>
row.slice(mid));
```

```
    const B11 = B.slice(0, mid).map(row =>
row.slice(0, mid));
```

```
    const B12 = B.slice(0, mid).map(row =>
row.slice(mid));
```

```
    const B21 = B.slice(mid).map(row =>
row.slice(0, mid));
```

```
    const B22 = B.slice(mid).map(row =>
row.slice(mid));
```

```
    // Recursive calls for Strassen's algorithm
    (steps omitted for brevity)
```

```
    // Combine the results and return the final
    matrix
```

```
    return [];
```

```
}
```

```
// Example (actual recursive steps omitted for
brevity)
```

```
const A = [
```

```
  [1, 2],
```

```
  [3, 4]
```

```
];
```

```
const B = [
```

```
  [5, 6],
```

```
  [7, 8]
```

```
];
```

```
console.log(strassenMatrixMultiplication(A,
B)); // Output: Result of A * B
```

5. Closest Pair of Points

The closest pair of points problem is a common computational geometry problem that asks for the two closest points in a set of points on a 2D plane. A divide and conquer approach helps solve this problem efficiently in $O(n \log n)$ time.

Implementation (Outline)

```
function closestPairOfPoints(points: [number,
number][]): [number, number, number] {
  points.sort((a, b) => a[0] - b[0]); // Sort
  by x-coordinate

  // Call the divide and conquer approach
  recursively (steps omitted)

  return [0, 0, 0]; // Placeholder for closest
  pair distance
}

// Example (actual logic omitted for brevity)
const points: [number, number][] = [
  [0, 0], [1, 1], [2, 2], [3, 3], [4, 4]
];
console.log(closestPairOfPoints(points)); //
Output: The closest pair distance
```

Analysis of Divide and Conquer Algorithms

Algorit hm	Time Complexity	Space Complexity
Merge Sort	$O(n \log n)$	$O(n)$
Quick Sort	$O(n \log n)$	$O(\log n)$

Binary Search	$O(\log n)$ $O(\log n)$	$O(1)$ $O(1)$ $O(1)$
Strassen's Matrix	$O(n^{\log_2 7})$ $O(n^{\log_2 7})$	$O(n^2)$ $O(n^2)$ $O(n^2)$
Closest Pair	$O(n \log n)$ $O(n \log n)$	$O(n)$ $O(n)$ $O(n)$

Conclusion

Divide and Conquer is a versatile technique that works well for a wide range of problems, particularly those that involve recursion and can be broken down into smaller subproblems. Algorithms like Merge Sort, Quick Sort, and Binary Search are some of the most common examples, demonstrating the efficiency and power of this approach. By breaking the problem down and solving smaller, manageable parts, divide and conquer can lead to elegant and efficient solutions.

Backtracking and Branch & Bound in TypeScript

Both **Backtracking** and **Branch & Bound** are algorithmic techniques used for solving optimization problems, decision problems, and constraint satisfaction problems. They are commonly used in combinatorial optimization, where the goal is to find the best solution among a large set of possible solutions.

Principles of Backtracking

Backtracking is a general algorithmic technique used to solve problems incrementally, by trying to build a solution step by step and abandoning solutions as soon as it is determined that they cannot be extended to a valid solution. It is particularly useful for solving constraint satisfaction problems, such as puzzles, where a solution must meet certain conditions.

Steps in Backtracking:

1. **Choose:** Choose the next possible step to try.
2. **Explore:** Explore this step by moving forward and recursively applying the algorithm.
3. **Check Constraints:** Check if the current step is valid or violates any constraints.
4. **Backtrack:** If the current step does not lead to a solution, undo the last choice and try a different step.

Backtracking works by exploring all possible options, rejecting those that don't meet the constraints, and backtracking as soon as an invalid state is encountered.

Common Backtracking Problems in TypeScript

Here are some common backtracking problems and their TypeScript implementations:

1. N-Queens Problem

The N-Queens problem is a classic backtracking problem where the goal is to place N queens on an N×N chessboard such that no two queens threaten each other (i.e., no two queens share the same row, column, or diagonal).

Implementation

```
function solveNQueens(n: number): string[][] {
  const result: string[][] = [];
  const board: string[][] = Array(n).fill('').map(() => Array(n).fill('.'));
```

```
  function isSafe(row: number, col: number): boolean {
    for (let i = 0; i < row; i++) {
      if (board[i][col] === 'Q') return false;
      if (col - (row - i) >= 0 && board[i][col - (row - i)] === 'Q') return false;
      if (col + (row - i) < n && board[i][col + (row - i)] === 'Q') return false;
    }
    return true;
  }
```

```
  function placeQueen(row: number): void {
    if (row === n) {
      result.push(board.map(row => row.join('')));
      return;
    }
```

```

        for (let col = 0; col < n; col++) {
            if (isSafe(row, col)) {
                board[row][col] = 'Q';
                placeQueen(row + 1);
                board[row][col] = '.'; // Backtrack
            }
        }
    }

    placeQueen(0);
    return result;
}

// Example
console.log(solveNQueens(4)); // Output: All
possible solutions for 4-Queens

```

2. Subset Sum Problem

Given a set of numbers, the subset sum problem asks whether there is a subset of these numbers that sums up to a given target.

Implementation

```

function subsetSum(nums: number[], target:
number): boolean {
    const result: boolean[] = [];

    function backtrack(index: number,
currentSum: number): void {
        if (currentSum === target) {
            result.push(true);
            return;
        }
    }
}

```

```

    }
    if (currentSum > target || index ===
nums.length) return;

    // Include the current number
    backtrack(index + 1, currentSum +
nums[index]);
    // Exclude the current number
    backtrack(index + 1, currentSum);
  }

  backtrack(0, 0);
  return result.length > 0;
}

```

```

// Example
console.log(subsetSum([3, 34, 4, 12, 5, 2],
9)); // Output: true (Subset: [4, 5])

```

3. Sudoku Solver

The Sudoku solver involves filling a 9x9 grid with digits from 1 to 9 such that every row, column, and 3x3 subgrid contains all the digits from 1 to 9.

Implementation

```

function solveSudoku(board: string[][]):
boolean {
  function isValid(board: string[][], row:
number, col: number, num: string): boolean {
    for (let i = 0; i < 9; i++) {
      if (board[row][i] === num ||
board[i][col] === num) return false;
      const r = Math.floor(row / 3) * 3 +
Math.floor(i / 3);
      const c = Math.floor(col / 3) * 3 + i % 3;

```

```

        if (board[r][c] === num) return false;
    }
    return true;
}

function solve(board: string[][]): boolean {
    for (let row = 0; row < 9; row++) {
        for (let col = 0; col < 9; col++) {
            if (board[row][col] === '.') {
                for (let num = 1; num <= 9; num++) {
                    const numStr = num.toString();
                    if (isValid(board, row, col,
numStr)) {
                        board[row][col] = numStr;
                        if (solve(board)) return true;
                        board[row][col] = '.'; //
Backtrack
                    }
                }
                return false; // No valid number
found
            }
        }
    }
    return true;
}

solve(board);
return true;
}

// Example
const board: string[][] = [

```

```

    ['5', '3', '.', '.', '7', '.', '.', '.'],
    '.'],
    ['6', '.', '.', '1', '9', '5', '.', '.',
    '.'],
    ['.', '9', '8', '.', '.', '.', '.', '6',
    '.'],
    ['8', '.', '.', '8', '.', '6', '.', '.'],
    '3'],
    ['4', '.', '6', '8', '.', '3', '.', '.'],
    '1'],
    ['7', '.', '.', '.', '2', '7', '.', '.'],
    '.'],
    ['.', '6', '.', '.', '.', '2', '9', '.',
    '.'],
    ['.', '.', '.', '4', '1', '9', '.', '.'],
    '5'],
    ['.', '.', '.', '.', '8', '.', '.', '7', '9']
];
solveSudoku(board);
console.log(board);

```

Branch and Bound Techniques in TypeScript

Branch and Bound (B&B)

is an algorithm design paradigm that is used for solving optimization problems. It systematically enumerates all candidate solutions, "bounding" those that cannot yield a better solution than the best one found so far. The key difference between Branch & Bound and backtracking is the use of bounds to prune the search space.

Steps in Branch and Bound:

1. **Branch:** Divide the problem into smaller subproblems.

2. **Bound:** Calculate bounds to determine if a subproblem can lead to a better solution.
3. **Prune:** Discard subproblems that can't improve the current best solution.
4. **Solution:** Return the best solution found.

1. 0/1 Knapsack Problem (Branch and Bound)

The 0/1 Knapsack problem asks for the maximum value that can be carried in a knapsack with a fixed capacity, where each item can either be included or excluded.

Implementation

```
interface Item {
  weight: number;
  value: number;
}

function knapsackBranchBound(capacity: number,
  items: Item[]): number {
  const n = items.length;

  function bound(i: number, weight: number,
    value: number): number {
    if (weight >= capacity) return 0;
    let result = value;
    let j = i + 1;
    while (j < n && weight + items[j].weight <=
      capacity) {
      weight += items[j].weight;
      value += items[j].value;
      j++;
    }
    if (j < n) {
```

```

        result += (capacity - weight) *
(items[j].value / items[j].weight);
    }
    return result;
}

```

```

function branch(i: number, weight: number,
value: number): number {
    if (i === n) return value;
    if (weight >= capacity) return 0;

```

```

    const includeValue = branch(i + 1, weight
+ items[i].weight, value + items[i].value);
    const excludeValue = branch(i + 1, weight,
value);

```

```

    return Math.max(includeValue,
excludeValue);
}

```

```

    return branch(0, 0, 0);
}

```

// Example

```

const items: Item[] = [
    { weight: 2, value: 3 },
    { weight: 3, value: 4 },
    { weight: 4, value: 5 },
    { weight: 5, value: 6 }
];

```

```

console.log(knapsackBranchBound(5, items)); //
Output: Maximum value that can be taken

```

Conclusion

Backtracking and Branch & Bound are both effective techniques for solving complex combinatorial optimization problems. While **Backtracking** explores all possible solutions in a depth-first manner, pruning paths that violate constraints, **Branch & Bound** leverages bounds to eliminate solutions that cannot outperform the best known solution. These approaches are essential for problems such as the N-Queens problem, Sudoku solver, and knapsack problem. By combining these techniques with efficient data structures, we can solve a wide range of challenging problems.

Advanced Data Structures in TypeScript

Advanced data structures are fundamental for optimizing the performance of various algorithms, especially when dealing with large datasets or complex queries. Some of these data structures, such as **Segment Trees**, **Fenwick Trees (Binary Indexed Trees)**, **Suffix Trees**, and **K-D Trees**, provide efficient solutions to problems involving range queries, string matching, and multidimensional data processing.

Segment Trees

Basic Segment Tree in TypeScript

A **Segment Tree** is a binary tree used for storing intervals or segments. It allows querying the sum, minimum, or maximum of a given range in logarithmic time. Segment trees are especially useful in problems where you need to perform multiple range queries or updates efficiently.

Implementation

```
class SegmentTree {  
  private tree: number[];  
  private n: number;  
  
  constructor(arr: number[]) {  
    this.n = arr.length;  
    this.tree = new Array(4 * this.n).fill(0);  
    this.build(arr, 0, 0, this.n - 1);  
  }  
}
```

```

    // Build the segment tree
    private build(arr: number[], node: number,
start: number, end: number): void {
        if (start === end) {
            this.tree[node] = arr[start];
        } else {
            const mid = Math.floor((start + end) / 2);
            this.build(arr, 2 * node + 1, start,
mid);
            this.build(arr, 2 * node + 2, mid + 1,
end);
            this.tree[node] = this.tree[2 * node + 1]
+ this.tree[2 * node + 2];
        }
    }

    // Query the sum of a range
    public query(left: number, right: number):
number {
        return this.queryRange(0, 0, this.n - 1,
left, right);
    }

    private queryRange(node: number, start:
number, end: number, left: number, right:
number): number {
        if (right < start || left > end) {
            return 0;
        }
        if (left <= start && end <= right) {
            return this.tree[node];
        }
        const mid = Math.floor((start + end) / 2);

```

```

        const leftQuery = this.queryRange(2 * node
+ 1, start, mid, left, right);
        const rightQuery = this.queryRange(2 * node
+ 2, mid + 1, end, left, right);
        return leftQuery + rightQuery;
    }

    // Update an element in the array
    public update(index: number, value: number):
void {
        this.updateValue(0, 0, this.n - 1, index,
value);
    }

    private updateValue(node: number, start:
number, end: number, index: number, value:
number): void {
        if (start === end) {
            this.tree[node] = value;
        } else {
            const mid = Math.floor((start + end) / 2);
            if (start <= index && index <= mid) {
                this.updateValue(2 * node + 1, start,
mid, index, value);
            } else {
                this.updateValue(2 * node + 2, mid + 1,
end, index, value);
            }
            this.tree[node] = this.tree[2 * node + 1]
+ this.tree[2 * node + 2];
        }
    }
}

```

```
// Example
const arr = [1, 3, 5, 7, 9, 11];
const segmentTree = new SegmentTree(arr);
console.log(segmentTree.query(1, 3)); //
Output: 15 (sum of elements in range [1, 3])
segmentTree.update(2, 6); // Update arr[2] to
6
console.log(segmentTree.query(1, 3)); //
Output: 16 (sum of elements in range [1, 3])
```

Lazy Propagation in TypeScript

Lazy Propagation is an optimization technique used in segment trees to delay updates to the segment tree until necessary. This technique improves the efficiency of range updates, ensuring that updates are applied only when queries require them.

Implementation with Lazy Propagation

```
class SegmentTreeLazy {
  private tree: number[];
  private lazy: number[];
  private n: number;

  constructor(arr: number[]) {
    this.n = arr.length;
    this.tree = new Array(4 * this.n).fill(0);
    this.lazy = new Array(4 * this.n).fill(0);
    this.build(arr, 0, 0, this.n - 1);
  }

  private build(arr: number[], node: number,
start: number, end: number): void {
```

```

        if (start === end) {
            this.tree[node] = arr[start];
        } else {
            const mid = Math.floor((start + end) / 2);
            this.build(arr, 2 * node + 1, start,
mid);
            this.build(arr, 2 * node + 2, mid + 1,
end);
            this.tree[node] = this.tree[2 * node + 1]
+ this.tree[2 * node + 2];
        }
    }
}

```

```

    private propagate(node: number, start:
number, end: number): void {
        if (this.lazy[node] !== 0) {
            this.tree[node] += (end - start + 1) *
this.lazy[node];
            if (start !== end) {
                this.lazy[2 * node + 1] +=
this.lazy[node];
                this.lazy[2 * node + 2] +=
this.lazy[node];
            }
            this.lazy[node] = 0;
        }
    }
}

```

```

    public updateRange(left: number, right:
number, value: number): void {
        this.updateRangeUtil(0, 0, this.n - 1,
left, right, value);
    }
}

```

```

    private updateRangeUtil(node: number, start:
number, end: number, left: number, right:
number, value: number): void {
        this.propagate(node, start, end);

        if (right < start || left > end) {
            return;
        }

        if (left <= start && end <= right) {
            this.lazy[node] += value;
            this.propagate(node, start, end);
            return;
        }

        const mid = Math.floor((start + end) / 2);
        this.updateRangeUtil(2 * node + 1, start,
mid, left, right, value);
        this.updateRangeUtil(2 * node + 2, mid + 1,
end, left, right, value);
        this.tree[node] = this.tree[2 * node + 1]
+ this.tree[2 * node + 2];
    }

    public query(left: number, right: number):
number {
        return this.queryRange(0, 0, this.n - 1,
left, right);
    }

```

```

    private queryRange(node: number, start:
number, end: number, left: number, right:
number): number {
        this.propagate(node, start, end);

        if (right < start || left > end) {
            return 0;
        }

        if (left <= start && end <= right) {
            return this.tree[node];
        }

        const mid = Math.floor((start + end) / 2);
        const leftQuery = this.queryRange(2 * node
+ 1, start, mid, left, right);
        const rightQuery = this.queryRange(2 * node
+ 2, mid + 1, end, left, right);
        return leftQuery + rightQuery;
    }
}

```

```

// Example
const arr = [1, 3, 5, 7, 9, 11];
const      segmentTreeLazy      =      new
SegmentTreeLazy(arr);
segmentTreeLazy.updateRange(1, 3, 10);
console.log(segmentTreeLazy.query(1, 3)); //
Output: 36 (sum after range update)

```

Fenwick Trees (Binary Indexed Trees)

Structure and Applications in TypeScript

A **Fenwick Tree (Binary Indexed Tree)** is a data structure that supports efficient prefix sum queries and updates. It allows both operations to be performed in $O(\log n)$ time. Unlike a segment tree, a Fenwick Tree is more space-efficient, requiring only $O(n)$ space.

Implementation

```
class FenwickTree {
  private tree: number[];
  private n: number;

  constructor(n: number) {
    this.n = n;
    this.tree = new Array(n + 1).fill(0);
  }

  // Update the value at index i
  public update(index: number, delta: number): void {
    while (index <= this.n) {
      this.tree[index] += delta;
      index += index & -index;
    }
  }

  // Query the prefix sum from 1 to index
  public query(index: number): number {
    let sum = 0;
    while (index > 0) {
```



```

        sum += this.tree[index];
        index -= index & -index;
    }
    return sum;
}

// Query the sum in the range [left, right]
public rangeQuery(left: number, right:
number): number {
    return this.query(right) - this.query(left
- 1);
}
}

// Example
const fenwickTree = new FenwickTree(5);
fenwickTree.update(1, 3); // arr[1] += 3
fenwickTree.update(2, 2); // arr[2] += 2
console.log(fenwickTree.query(2)); // Output:
5 (sum from 1 to 2)
console.log(fenwickTree.rangeQuery(1, 2)); //
Output: 5 (sum from index 1 to 2)

```

Suffix Trees and Arrays

Suffix Trees in TypeScript

A **Suffix Tree** is a compressed trie of all suffixes of a given string. It is an advanced data structure used to solve problems involving string pattern matching, such as finding the longest common substring or suffix matching.

```

class SuffixTree {
    private root: any;

    constructor(text: string) {

```

```

    this.root = {};
    this.build(text);
}

private build(text: string): void {
    for (let i = 0; i < text.length; i++) {
        let node = this.root;
        for (let j = i; j < text.length; j++) {
            if (!node[text[j]]) {
                node[text[j]] = {};
            }
            node = node[text[j]];
        }
    }
}

// Find if a substring exists in the tree
public exists(substring: string): boolean {
    let node = this.root;
    for (let char of substring) {
        if (!node[char]) {
            return false;
        }
        node = node[char];
    }
    return true;
}

// Example
const suffixTree = new SuffixTree("banana");
console.log(suffixTree.exists("ban")); //
Output: true

```

```
console.log(suffixTree.exists("apple"));    //  
Output: false
```

K-D Trees

Structure and Applications in TypeScript

A **K-D Tree** is a binary tree used for organizing points in a k-dimensional space. It is useful in applications involving multidimensional data such as spatial searches, range queries, and nearest neighbor searches.

```
class KDTree {  
    private points: number[][];  
  
    constructor(points: number[][]) {  
        this.points = points;  
        // The actual KD Tree construction  
        algorithm can be added here  
    }  
  
    // Example method to query nearest neighbors  
    (simplified)  
    public nearestNeighbor(query: number[]):  
    number[] {  
        // Implement nearest neighbor search  
        return this.points[0]; // Placeholder  
    }  
}  
  
// Example  
const points = [  
    [2, 3],  
    [5, 4],  
    [9, 6],
```

```
[4, 7],  
[8, 1],  
[7, 2],  
];  
const kdTree = new KDTree(points);  
console.log(kdTree.nearestNeighbor([9, 2]));  
// Placeholder for actual nearest neighbor  
logic
```

Conclusion

Advanced data structures such as Segment Trees, Fenwick Trees, Suffix Trees, and K-D Trees provide powerful solutions for complex algorithmic problems. They are essential for tasks involving range queries, multidimensional data processing, string matching, and more. Mastering these structures is crucial for developers working with large-scale applications and complex datasets.

Graph Algorithms in TypeScript

Graph algorithms are vital for solving problems that involve relationships or connections between objects. One of the most commonly studied problems in graph theory is the **Minimum Spanning Tree (MST)**, which finds the subset of edges that connect all the vertices of a graph with the minimal total edge weight. Two well-known algorithms for finding the MST are **Kruskal's Algorithm** and **Prim's Algorithm**.

Minimum Spanning Trees

A **Minimum Spanning Tree** of a graph is a subgraph that connects all the vertices with the minimum total edge weight, without any cycles. The two most popular algorithms to find the MST are:

- **Kruskal's Algorithm**
- **Prim's Algorithm**

Kruskal's Algorithm in TypeScript

Kruskal's Algorithm is a greedy algorithm that finds the MST by sorting all the edges in non-decreasing order of their weights. The algorithm then adds edges to the MST one by one, ensuring no cycles are formed. This is typically done using a **Union-Find (Disjoint Set Union, DSU)** data structure to keep track of the components.

Implementation

```
class UnionFind {  
    private parent: number[];  
    private rank: number[];  
  
    constructor(n: number) {
```

```

    this.parent = Array.from({ length: n }, (_,
index) => index);
    this.rank = Array(n).fill(0);
}

find(x: number): number {
    if (this.parent[x] !== x) {
        this.parent[x] =
this.find(this.parent[x]);
    }
    return this.parent[x];
}

union(x: number, y: number): void {
    const rootX = this.find(x);
    const rootY = this.find(y);
    if (rootX !== rootY) {
        if (this.rank[rootX] > this.rank[rootY])
{
            this.parent[rootY] = rootX;
        } else if (this.rank[rootX] <
this.rank[rootY]) {
            this.parent[rootX] = rootY;
        } else {
            this.parent[rootY] = rootX;
            this.rank[rootX]++;
        }
    }
}
}

class KruskalMST {

```

```

    private edges: [number, number, number][]; //
    [u, v, weight]
    private n: number;

    constructor(n: number, edges: [number,
    number, number][]) {
        this.n = n;
        this.edges = edges;
    }

    // Kruskal's algorithm to find MST
    public findMST(): [number, number, number][]
    {
        this.edges.sort((a, b) => a[2] - b[2]); //
        Sort edges by weight
        const uf = new UnionFind(this.n);
        const mst: [number, number, number][] = [];

        for (const [u, v, weight] of this.edges) {
            if (uf.find(u) !== uf.find(v)) {
                uf.union(u, v);
                mst.push([u, v, weight]);
            }
        }

        return mst;
    }
}

// Example: Graph with 4 vertices and 5 edges
const edges: [number, number, number][] = [
    [0, 1, 10],
    [0, 2, 6],

```

```

    [0, 3, 5],
    [1, 3, 15],
    [2, 3, 4],
  ];

```

```

const kruskal = new KruskalMST(4, edges);
console.log(kruskal.findMST());
// Output: [ [ 2, 3, 4 ], [ 0, 3, 5 ], [ 0, 1,
10 ] ]

```

Prim's Algorithm in TypeScript

Prim's Algorithm is another greedy algorithm that finds the MST by starting with an arbitrary vertex and expanding the tree by adding the shortest edge that connects a vertex in the tree to a vertex outside the tree. This algorithm uses a **priority queue (min-heap)** to always pick the minimum edge weight.

Implementation

```

class MinHeap {
  private heap: [number, number][]; // [vertex,
weight]
  private vertexIndex: Map<number, number>;

  constructor() {
    this.heap = [];
    this.vertexIndex = new Map();
  }

  insert(vertex: number, weight: number): void
  {
    this.heap.push([vertex, weight]);
    this.vertexIndex.set(vertex,
this.heap.length - 1);

```



```

    this.heapifyUp(this.heap.length - 1);
}

extractMin(): [number, number] {
    const min = this.heap[0];
    const last = this.heap.pop();
    if (this.heap.length > 0 && last) {
        this.heap[0] = last;
        this.vertexIndex.set(last[0], 0);
        this.heapifyDown(0);
    }
    this.vertexIndex.delete(min[0]);
    return min;
}

decreaseKey(vertex:    number,    newWeight:
number): void {
    const                index                =
this.vertexIndex.get(vertex);
    if (index !== undefined) {
        this.heap[index][1] = newWeight;
        this.heapifyUp(index);
    }
}

private heapifyUp(index: number): void {
    let parent = Math.floor((index - 1) / 2);
    while (index > 0 && this.heap[index][1] <
this.heap[parent][1]) {
        [this.heap[index], this.heap[parent]] =
[this.heap[parent], this.heap[index]];

```

```
this.vertexIndex.set(this.heap[index][0],  
index);
```

```
this.vertexIndex.set(this.heap[parent][0],  
parent);  
    index = parent;  
    parent = Math.floor((index - 1) / 2);  
}  
}
```

```
private heapifyDown(index: number): void {  
    let leftChild = 2 * index + 1;  
    let rightChild = 2 * index + 2;  
    let smallest = index;  
  
    if (leftChild < this.heap.length &&  
this.heap[leftChild][1] <  
this.heap[smallest][1]) {  
        smallest = leftChild;  
    }  
  
    if (rightChild < this.heap.length &&  
this.heap[rightChild][1] <  
this.heap[smallest][1]) {  
        smallest = rightChild;  
    }  
  
    if (smallest !== index) {  
        [this.heap[index], this.heap[smallest]]  
= [this.heap[smallest], this.heap[index]];
```

```
this.vertexIndex.set(this.heap[index][0],  
index);
```

```
this.vertexIndex.set(this.heap[smallest][0],  
smallest);
```

```
    this.heapifyDown(smallest);
```

```
  }
```

```
}
```

```
isEmpty(): boolean {
```

```
    return this.heap.length === 0;
```

```
}
```

```
}
```

```
class PrimMST {
```

```
    private n: number;
```

```
    private graph: Map<number, [number,  
number] []>;
```

```
    constructor(n: number, edges: [number,  
number, number] []) {
```

```
        this.n = n;
```

```
        this.graph = new Map();
```

```
        for (const [u, v, weight] of edges) {
```

```
            if (!this.graph.has(u))
```

```
        this.graph.set(u, []);
```

```
            if (!this.graph.has(v))
```

```
        this.graph.set(v, []);
```

```
            this.graph.get(u)?.push([v, weight]);
```

```
            this.graph.get(v)?.push([u, weight]);
```

```
        }
```

```
}
```

```

// Prim's algorithm to find MST
public findMST(): [number, number, number][]
{
    const mst: [number, number, number][] = [];
    const minHeap = new MinHeap();
    const visited = new Set();

    // Start from vertex 0
    minHeap.insert(0, 0);

    while (!minHeap.isEmpty()) {
        const [u, weight] = minHeap.extractMin();

        if (visited.has(u)) continue;
        visited.add(u);

        // Add edge to MST if it connects to the
        tree
        if (weight > 0) {
            mst.push([u, weight, u]);
        }

        // Update neighboring vertices
        for (const [v, w] of this.graph.get(u) ||
[]) {
            if (!visited.has(v)) {
                minHeap.insert(v, w);
            }
        }
    }
}

```

```

        return mst;
    }
}

// Example: Graph with 4 vertices and 5 edges
const edges: [number, number, number][] = [
    [0, 1, 10],
    [0, 2, 6],
    [0, 3, 5],
    [1, 3, 15],
    [2, 3, 4],
];

const prim = new PrimMST(4, edges);
console.log(prim.findMST());
// Output: [ [ 0, 10, 0 ], [ 3, 5, 3 ], [ 2, 4,
2 ] ]

```

Comparison of Kruskal's and Prim's Algorithms

- **Kruskal's Algorithm** works well when the graph is sparse and uses a union-find data structure to manage connected components.
- **Prim's Algorithm** is more efficient when the graph is dense, as it expands the MST from an initial vertex and uses a priority queue for efficient edge selection.

Both algorithms have their use cases, and choosing between them depends on the specific properties of the graph you're working with (e.g., density, edge weights, etc.).

Conclusion

Both **Kruskal's Algorithm** and **Prim's Algorithm** are essential algorithms for solving the Minimum Spanning Tree problem. Kruskal's algorithm works better for edge-heavy graphs, while Prim's is more efficient for dense graphs. Understanding these algorithms and their implementations in TypeScript helps solve complex graph-related problems and optimize graph traversal tasks.

Shortest Path Algorithms in TypeScript

Shortest path algorithms are essential for solving graph problems where you need to find the least cost or shortest route between nodes. Here, we explore three classic algorithms: **Dijkstra's Algorithm**, **Bellman-Ford Algorithm**, and **Floyd-Warshall Algorithm**.

Dijkstra's Algorithm in TypeScript

Dijkstra's algorithm efficiently finds the shortest path from a source node to all other nodes in a graph with non-negative weights. It uses a priority queue to always expand the shortest known path first.

Implementation

```
class Dijkstra {
  private graph: Map<number, [number, number][]>;

  constructor(edges: [number, number, number][]) {
    this.graph = new Map();
    for (const [u, v, w] of edges) {
      if (!this.graph.has(u))
        this.graph.set(u, []);
      this.graph.get(u)?.push([v, w]);
    }
  }

  findShortestPaths(source: number): Record<number, number> {
```

```

    const distances: Record<number, number> =
    {};
    const priorityQueue: [number, number][] =
    []; // [node, distance]

    for (const node of this.graph.keys()) {
        distances[node] = Infinity;
    }
    distances[source] = 0;

    priorityQueue.push([source, 0]);

    while (priorityQueue.length > 0) {
        priorityQueue.sort((a, b) => a[1] -
b[1]); // Sort by distance
        const [currentNode, currentDistance] =
priorityQueue.shift()!;

        if (currentDistance >
distances[currentNode]) continue;

        for (const [neighbor, weight] of
this.graph.get(currentNode) || []) {
            const newDistance = currentDistance +
weight;
            if (newDistance < distances[neighbor])
            {
                distances[neighbor] = newDistance;
                priorityQueue.push([neighbor,
newDistance]);
            }
        }
    }
}

```



```

        return distances;
    }
}

// Example
const edges: [number, number, number][] = [
    [0, 1, 4],
    [0, 2, 1],
    [2, 1, 2],
    [1, 3, 1],
    [2, 3, 5],
];
const dijkstra = new Dijkstra(edges);
console.log(dijkstra.findShortestPaths(0));
// Output: { '0': 0, '1': 3, '2': 1, '3': 4 }

```

Bellman-Ford Algorithm in TypeScript

The **Bellman-Ford Algorithm** can handle graphs with negative weights, though it detects negative cycles. It relaxes all edges up to $V-1$ times, where V is the number of vertices.

Implementation

```

class BellmanFord {
    private edges: [number, number, number][];
    private vertices: number;

    constructor(vertices: number, edges:
[number, number, number][]) {
        this.vertices = vertices;
        this.edges = edges;
    }
}

```

```

    findShortestPaths(source:          number):
Record<number, number> {
    const distances: Record<number, number> =
{};

    for (let i = 0; i < this.vertices; i++) {
        distances[i] = Infinity;
    }
    distances[source] = 0;

    for (let i = 0; i < this.vertices - 1; i++)
    {
        for (const [u, v, w] of this.edges) {
            if (distances[u] + w < distances[v]) {
                distances[v] = distances[u] + w;
            }
        }
    }

    // Check for negative weight cycles
    for (const [u, v, w] of this.edges) {
        if (distances[u] + w < distances[v]) {
            throw new Error("Graph contains a
negative weight cycle");
        }
    }

    return distances;
}
}

```

// Example

```

const edges: [number, number, number][] = [
  [0, 1, 4],
  [0, 2, 1],
  [2, 1, 2],
  [1, 3, 1],
  [2, 3, 5],
];
const bellmanFord = new BellmanFord(4, edges);
console.log(bellmanFord.findShortestPaths(0));
// Output: { '0': 0, '1': 3, '2': 1, '3': 4 }

```

Floyd-Warshall Algorithm in TypeScript

The **Floyd-Warshall Algorithm** is an all-pairs shortest path algorithm. It computes the shortest paths between every pair of nodes by iteratively improving estimates.

Implementation

```

class FloydWarshall {
  private vertices: number;

  constructor(vertices: number) {
    this.vertices = vertices;
  }

  findShortestPaths(graph: number[][]): number[][] {
    const distances = graph.map(row => [...row]);

    for (let k = 0; k < this.vertices; k++) {
      for (let i = 0; i < this.vertices; i++) {

```

```

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

return distances;
}
}

```

```

// Example
const graph = [
    [0, 4, Infinity, Infinity],
    [Infinity, 0, 2, 1],
    [Infinity, Infinity, 0, 5],
    [Infinity, Infinity, Infinity, 0],
];
const floydWarshall = new FloydWarshall(4);
console.log(floydWarshall.findShortestPaths(graph));
// Output: [
//   [ 0, 4, 6, 5 ],
//   [ Infinity, 0, 2, 1 ],
//   [ Infinity, Infinity, 0, 5 ],
//   [ Infinity, Infinity, Infinity, 0 ]
// ]

```

Comparison of Algorithms

Algorithm	Handles Negative Weights	Time Complexity	Use Case
Dijkstra's	No	$O(V^2)$ or $O(E \log V)$ with a heap	Single-source shortest path with non-negative weights.
Bellman-Ford	Yes	$O(V \times E)$	Single-source shortest path with negative weights.
Floyd-Warshall	No	$O(V^3)$	All-pairs shortest path.

Conclusion

Shortest path algorithms solve critical problems in transportation, networking, and optimization. Dijkstra's Algorithm excels in non-negative graphs, Bellman-Ford handles negative weights effectively, and Floyd-Warshall provides a comprehensive view of all-pairs shortest paths. By understanding and implementing these algorithms in TypeScript, you can address a wide range of graph-related challenges efficiently.

Network Flow in TypeScript

Network flow algorithms are essential for solving problems where a network has capacities on edges and the goal is to maximize the flow from a source node to a sink node. Here, we discuss two classic approaches: the **Ford-Fulkerson Method** and its implementation through the **Edmonds-Karp Algorithm**.

Ford-Fulkerson Method in TypeScript

The **Ford-Fulkerson Method** uses augmenting paths to iteratively increase the flow in a network. It works on the idea of finding paths with available capacity and augmenting flow along these paths until no such paths exist.

Implementation

```
class FordFulkerson {
  private graph: number[][]; // Adjacency
  matrix representing capacities
  private size: number;

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

  private bfs(residualGraph: number[][],
    source: number, sink: number, parent:
    number[]): boolean {
    const visited = new
    Array(this.size).fill(false);
    const queue: number[] = [];
```

```

    queue.push(source);
    visited[source] = true;

    while (queue.length > 0) {
        const u = queue.shift()!;
        for (let v = 0; v < this.size; v++) {
            if (!visited[v] && residualGraph[u][v]
> 0) {
                queue.push(v);
                visited[v] = true;
                parent[v] = u;

                if (v === sink) return true;
            }
        }
    }

    return false;
}

public maxFlow(source: number, sink:
number): number {
    const residualGraph = this.graph.map(row
=> [...row]);
    const parent = new Array(this.size).fill(-
1);
    let maxFlow = 0;

    while (this.bfs(residualGraph, source,
sink, parent)) {
        let pathFlow = Infinity;

```

```

        for (let v = sink; v !== source; v =
parent[v]) {
            const u = parent[v];
            pathFlow = Math.min(pathFlow,
residualGraph[u][v]);
        }

        for (let v = sink; v !== source; v =
parent[v]) {
            const u = parent[v];
            residualGraph[u][v] -= pathFlow;
            residualGraph[v][u] += pathFlow;
        }

        maxFlow += pathFlow;
    }

    return maxFlow;
}
}

```

// Example

```

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);

```



```
console.log(fordFulkerson.maxFlow(0, 5));  
// Output: 23
```

Edmonds-Karp Algorithm in TypeScript

The **Edmonds-Karp Algorithm** is a specific implementation of the Ford-Fulkerson Method that uses Breadth-First Search (BFS) to find the shortest augmenting path. This ensures a polynomial runtime of $O(V \times E^2)$.

Implementation

```
class EdmondsKarp {  
  private graph: number[][];  
  private size: number;  
  
  constructor(graph: number[][]) {  
    this.graph = graph;  
    this.size = graph.length;  
  }  
  
  private bfs(residualGraph: number[][],  
    source: number, sink: number, parent:  
    number[]): boolean {  
    const visited = new  
    Array(this.size).fill(false);  
    const queue: number[] = [];  
    queue.push(source);  
    visited[source] = true;  
  
    while (queue.length > 0) {  
      const u = queue.shift()!;  
      for (let v = 0; v < this.size; v++) {
```

```

        if (!visited[v] && residualGraph[u][v]
> 0) {
            queue.push(v);
            visited[v] = true;
            parent[v] = u;

            if (v === sink) return true;
        }
    }
}

return false;
}

public maxFlow(source: number, sink:
number): number {
    const residualGraph = this.graph.map(row
=> [...row]);
    const parent = new Array(this.size).fill(-
1);
    let maxFlow = 0;

    while (this.bfs(residualGraph, source,
sink, parent)) {
        let pathFlow = Infinity;

        for (let v = sink; v !== source; v =
parent[v]) {
            const u = parent[v];
            pathFlow = Math.min(pathFlow,
residualGraph[u][v]);
        }
    }
}

```

```

        for (let v = sink; v !== source; v =
parent[v]) {
            const u = parent[v];
            residualGraph[u][v] -= pathFlow;
            residualGraph[v][u] += pathFlow;
        }

        maxFlow += pathFlow;
    }

    return maxFlow;
}
}
}

```

// Example

```

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 edmondsKarp = new EdmondsKarp(graph);
console.log(edmondsKarp.maxFlow(0, 5));
// Output: 23

```

Comparison of Ford-Fulkerson and Edmonds-Karp

Algorithm	Time Complexity	Implementation	Characteristics
Ford-Fulkerson	$O(E \times \text{maxFlow})$ $O(E \times \text{maxFlow})$	General, uses augmenting paths	Efficient for small graphs.
Edmonds-Karp	$O(V \times E^2)$ $O(V \times E^2)$	BFS-based implementation	Guarantees polynomial runtime for dense graphs.

Conclusion

Both the Ford-Fulkerson Method and Edmonds-Karp Algorithm provide robust solutions for network flow problems. While **Ford-Fulkerson** is versatile, **Edmonds-Karp** ensures predictable performance on large, dense graphs. Implementing these algorithms in TypeScript allows developers to tackle real-world problems like traffic optimization, network routing, and supply chain management.

Matching and Covering in TypeScript

Matching and covering algorithms are fundamental tools for solving optimization problems in graph theory, including resource allocation, scheduling, and assignment problems. In this section, we focus on **Bipartite Matching** and the **Hungarian Algorithm** for maximum matching and minimum cost assignments.

Bipartite Matching in TypeScript

A **bipartite graph** is a graph whose vertices can be divided into two disjoint sets U and V , such that every edge connects a vertex in U to one in V . The goal of bipartite matching is to find the largest set of edges where no two edges share a vertex.

Algorithm Overview

1. Use a **DFS-based approach** to find augmenting paths.
2. Match as many vertices as possible by finding paths that increase the size of the matching.

Implementation

```
class BipartiteMatching {
    private graph: number[][];
    private sizeU: number;
    private sizeV: number;
    private matchU: number[];
    private matchV: number[];
    private visited: boolean[];

    constructor(graph: number[][], sizeU:
number, sizeV: number) {
        this.graph = graph;
        this.sizeU = sizeU;
```

```

        this.sizeV = sizeV;
        this.matchU = new Array(sizeU).fill(-1); //
Matches for U
        this.matchV = new Array(sizeV).fill(-1); //
Matches for V
        this.visited = new
Array(sizeV).fill(false);
    }

    private dfs(u: number): boolean {
        for (let v = 0; v < this.sizeV; v++) {
            if (this.graph[u][v] &&
!this.visited[v]) {
                this.visited[v] = true;
                if (this.matchV[v] === -1 ||
this.dfs(this.matchV[v])) {
                    this.matchU[u] = v;
                    this.matchV[v] = u;
                    return true;
                }
            }
        }
        return false;
    }

    public maximumMatching(): number {
        let result = 0;
        for (let u = 0; u < this.sizeU; u++) {
            this.visited.fill(false);
            if (this.dfs(u)) {
                result++;
            }
        }
    }

```

```

        return result;
    }
}

// Example
const graph = [
    [1, 1, 0],
    [1, 0, 1],
    [0, 1, 1],
];
const matching = new BipartiteMatching(graph,
3, 3);
console.log(matching.maximumMatching());
// Output: 3

```

Hungarian Algorithm in TypeScript

The **Hungarian Algorithm** solves the **assignment problem**, finding the minimum cost matching in a weighted bipartite graph. It uses a matrix representation of costs and reduces the problem iteratively.

Algorithm Overview

1. **Row and column reduction:** Minimize costs by subtracting row and column minima.
2. **Covering zeroes:** Cover all zeroes with a minimum number of lines.
3. **Adjust the matrix:** Modify uncovered elements to find more assignments.
4. Repeat until a complete matching is found.

Implementation

```
class HungarianAlgorithm {
  private costMatrix: number[][];
  private size: number;
  private labelU: number[];
  private labelV: number[];
  private matchU: number[];
  private matchV: number[];
  private slack: number[];
  private slackX: number[];
  private parent: number[];

  constructor(costMatrix: number[][]) {
    this.costMatrix = costMatrix;
    this.size = costMatrix.length;
    this.labelU = new Array(this.size).fill(0);
    this.labelV = new Array(this.size).fill(0);
    this.matchU = new Array(this.size).fill(-1);
    this.matchV = new Array(this.size).fill(-1);
    this.slack = new Array(this.size).fill(Infinity);
    this.slackX = new Array(this.size).fill(-1);
    this.parent = new Array(this.size).fill(-1);

    // Initialize labels
    for (let u = 0; u < this.size; u++) {
      this.labelU[u] = Math.max(...this.costMatrix[u]);
    }
  }
}
```



```

    }
}

private bfs(): void {
    const queue: number[] = [];
    const visitedU = new
Array(this.size).fill(false);
    const visitedV = new
Array(this.size).fill(false);

    let root = -1;
    for (let u = 0; u < this.size; u++) {
        if (this.matchU[u] === -1) {
            queue.push(u);
            root = u;
            this.parent[u] = -1;
            break;
        }
    }

    while (queue.length > 0) {
        const u = queue.shift()!;
        visitedU[u] = true;

        for (let v = 0; v < this.size; v++) {
            if (!visitedV[v]) {
                const delta = this.labelU[u] +
this.labelV[v] - this.costMatrix[u][v];
                if (delta === 0) {
                    visitedV[v] = true;
                    if (this.matchV[v] === -1) {
                        this.augment(u, v);
                        return;
                    }
                }
            }
        }
    }
}

```

```

        }
        queue.push(this.matchV[v]);
    } else if (delta < this.slack[v]) {
        this.slack[v] = delta;
        this.slackX[v] = u;
    }
    }
    }
    }
}

private augment(u: number, v: number): void
{
    while (u !== -1) {
        const prevMatchU = this.matchU[u];
        this.matchU[u] = v;
        this.matchV[v] = u;
        v = prevMatchU;
        u = this.parent[u];
    }
}

public findMinimumCostMatching(): number {
    for (let u = 0; u < this.size; u++) {
        this.slack.fill(Infinity);
        this.bfs();
    }

    let totalCost = 0;
    for (let u = 0; u < this.size; u++) {
        if (this.matchU[u] !== -1) {
            totalCost
            +=
        }
    }
    this.costMatrix[u][this.matchU[u]];
}

```

```

    }
  }
  return totalCost;
}
}

// Example
const costMatrix = [
  [4, 2, 3],
  [2, 3, 1],
  [3, 2, 4],
];
const hungarian = new
HungarianAlgorithm(costMatrix);
console.log(hungarian.findMinimumCostMatching(
));
// Output: 9

```

Conclusion

- **Bipartite Matching** is useful for maximizing pairings between two sets, such as job assignments.
- The **Hungarian Algorithm** efficiently solves assignment problems by finding the minimum cost for matching. By implementing these algorithms in TypeScript, developers can address real-world problems in logistics, scheduling, and resource optimization effectively.

String Algorithms in TypeScript

String algorithms are essential in text processing, search engines, data compression, and pattern matching. This section covers **Pattern Matching** algorithms and **Suffix Trees/Arrays**, two foundational topics in string manipulation.

Pattern Matching

Pattern matching algorithms are used to locate a substring (pattern) within a string (text). Below are implementations of the **Naive Algorithm**, **Knuth-Morris-Pratt (KMP) Algorithm**, and **Rabin-Karp Algorithm**.

Naive Algorithm in TypeScript

The **Naive Algorithm** checks for the pattern at every possible position in the text. Although simple, its efficiency suffers with large text and pattern sizes.

Implementation

```
function naivePatternMatching(text: string,
pattern: string): number[] {
  const results: number[] = [];
  const n = text.length;
  const m = pattern.length;

  for (let i = 0; i <= n - m; i++) {
    let match = true;
    for (let j = 0; j < m; j++) {
      if (text[i + j] !== pattern[j]) {
        match = false;
        break;
      }
    }
    if (match) results.push(i);
  }
}
```

```

    }
  }
  if (match) {
    results.push(i);
  }
}

return results;
}

// Example
const text = "ABABACABABAC";
const pattern = "ABAC";
console.log(naivePatternMatching(text,
pattern));
// Output: [2, 8]

```

Knuth-Morris-Pratt (KMP) Algorithm in TypeScript

The **KMP Algorithm** improves efficiency by preprocessing the pattern to create a partial match (or prefix) table. This table helps skip unnecessary comparisons.

Implementation

```

function buildKMPTable(pattern: string):
number[] {
  const m = pattern.length;
  const lps: number[] = new Array(m).fill(0);
  let length = 0;
  let i = 1;

  while (i < m) {
    if (pattern[i] === pattern[length]) {

```

```

        length++;
        lps[i] = length;
        i++;
    } else {
        if (length !== 0) {
            length = lps[length - 1];
        } else {
            lps[i] = 0;
            i++;
        }
    }
}

return lps;
}

function kmpPatternMatching(text: string,
pattern: string): number[] {
    const results: number[] = [];
    const n = text.length;
    const m = pattern.length;
    const lps = buildKMPTable(pattern);

    let i = 0; // index for text
    let j = 0; // index for pattern

    while (i < n) {
        if (pattern[j] === text[i]) {
            i++;
            j++;
        }

        if (j === m) {

```

```

        results.push(i - j);
        j = lps[j - 1];
    } else if (i < n && pattern[j] !== text[i])
    {
        if (j !== 0) {
            j = lps[j - 1];
        } else {
            i++;
        }
    }
}

return results;
}

```

```

// Example
console.log(kmpPatternMatching(text,
pattern));
// Output: [2, 8]

```

Rabin-Karp Algorithm in TypeScript

The **Rabin-Karp Algorithm** uses a hashing technique to compare substrings. It computes a hash for the pattern and each substring of the text.

Implementation

```

function rabinKarp(text: string, pattern:
string, base: number = 256, prime: number =
101): number[] {
    const results: number[] = [];
    const n = text.length;
    const m = pattern.length;

```

```

let patternHash = 0;
let textHash = 0;
let h = 1;

for (let i = 0; i < m - 1; i++) {
    h = (h * base) % prime;
}

for (let i = 0; i < m; i++) {
    patternHash = (base * patternHash +
pattern.charCodeAt(i)) % prime;
    textHash = (base * textHash +
text.charCodeAt(i)) % prime;
}

for (let i = 0; i <= n - m; i++) {
    if (patternHash === textHash) {
        let match = true;
        for (let j = 0; j < m; j++) {
            if (text[i + j] !== pattern[j]) {
                match = false;
                break;
            }
        }
        if (match) {
            results.push(i);
        }
    }

    if (i < n - m) {
        textHash = (base * (textHash -
text.charCodeAt(i) * h) + text.charCodeAt(i +
m)) % prime;
    }
}

```



```

        if (textHash < 0) {
            textHash += prime;
        }
    }
}

return results;
}

// Example
console.log(rabinKarp(text, pattern));
// Output: [2, 8]

```

Suffix Trees and Arrays

Suffix Trees

A **suffix tree** is a compressed trie that represents all suffixes of a given string. It allows for efficient pattern matching and substring queries.

Construction and Applications in TypeScript

```

class SuffixTreeNode {
    children: Map<string, SuffixTreeNode>;
    start: number;
    end: number | null;

    constructor(start: number = -1, end: number
| null = null) {
        this.children = new Map();
        this.start = start;
        this.end = end;
    }
}

```

```

class SuffixTree {
  root: SuffixTreeNode;
  text: string;

  constructor(text: string) {
    this.text = text;
    this.root = new SuffixTreeNode();
    this.build();
  }

  private build() {
    for (let i = 0; i < this.text.length; i++)
    {
      this.insertSuffix(i);
    }
  }

  private insertSuffix(index: number) {
    let current = this.root;
    for (let i = index; i < this.text.length;
i++) {
      const char = this.text[i];
      if (!current.children.has(char)) {
        current.children.set(char,      new
SuffixTreeNode(i, null));
      }
      current = current.children.get(char)!;
    }
    current.end = this.text.length;
  }

  public contains(pattern: string): boolean {
    let current = this.root;

```

```

    for (const char of pattern) {
      if (!current.children.has(char)) {
        return false;
      }
      current = current.children.get(char)!;
    }
    return true;
  }
}

// Example
const suffixTree = new SuffixTree("banana");
console.log(suffixTree.contains("ana"));    //
Output: true
console.log(suffixTree.contains("nana"));    //
Output: true
console.log(suffixTree.contains("apple"));    //
Output: false

```

Conclusion

- **Naive Algorithm** is straightforward but inefficient for large strings.
- **KMP Algorithm** optimizes pattern matching with a prefix table.
- **Rabin-Karp Algorithm** uses hashing for efficient substring search.
- **Suffix Trees** enable fast substring queries and pattern matching.

These algorithms, implemented in TypeScript, provide powerful tools for text processing and computational efficiency.

Computational Geometry in TypeScript

Computational geometry focuses on algorithms and techniques to solve problems in geometric spaces. This section introduces **basic concepts** such as points, lines, planes, and polygons, followed by algorithms like **Convex Hull**, **Line Segment Intersection**, and **Closest Pair of Points**, implemented in TypeScript.

Basic Concepts

Points, Lines, and Planes

1. **Point:** Represented by coordinates (x,y) in 2D space or (x,y,z) in 3D space.
2. **Line:** Defined by two points or its equation $ax+by+c=0$.
3. **Plane:** In 3D, represented by the equation $ax+by+cz+d=0$.

TypeScript Representation

```
type Point2D = { x: number; y: number };  
type Point3D = { x: number; y: number; z: number  
};
```

```
type Line = { a: number; b: number; c: number  
}; //  $ax + by + c = 0$   
type Plane = { a: number; b: number; c: number;  
d: number }; //  $ax + by + cz + d = 0$ 
```

Polygons

A polygon is a closed figure with straight-line segments as edges. It is represented as an array of points.

TypeScript Representation

```
type Polygon = Point2D[];
```

Polygon Operations

1. **Area Calculation:** Using the **Shoelace Formula**.
2. **Point in Polygon Test:** Using the **Ray Casting Algorithm**.

Algorithms

Convex Hull in TypeScript

The **Convex Hull** is the smallest convex polygon enclosing all given points. The **Graham Scan** algorithm is commonly used to compute it.

Implementation

```
function crossProduct(o: Point2D, a: Point2D,
b: Point2D): number {
    return (a.x - o.x) * (b.y - o.y) - (a.y -
o.y) * (b.x - o.x);
}
```

```
function convexHull(points: Point2D[]):
Point2D[] {
    if (points.length <= 1) return points;

    points.sort((p1, p2) => p1.x - p2.x || p1.y
- p2.y);

    const lower: Point2D[] = [];
    for (const point of points) {
        while (lower.length >= 2 &&
crossProduct(lower[lower.length - 2],
lower[lower.length - 1], point) <= 0) {
```

```

        lower.pop();
    }
    lower.push(point);
}

const upper: Point2D[] = [];
for (let i = points.length - 1; i >= 0; i--)
{
    const point = points[i];
    while (upper.length >= 2 &&
crossProduct(upper[upper.length - 2],
upper[upper.length - 1], point) <= 0) {
        upper.pop();
    }
    upper.push(point);
}

upper.pop();
lower.pop();
return lower.concat(upper);
}

// Example
const points: Point2D[] = [
    { x: 0, y: 3 }, { x: 2, y: 2 }, { x: 1, y: 1 },
    { x: 2, y: 1 },
    { x: 3, y: 0 }, { x: 0, y: 0 }, { x: 3, y: 3 }
];
console.log(convexHull(points));

```

Line Segment Intersection in TypeScript

Detecting if two line segments intersect is a fundamental computational geometry problem.

Implementation

```
function orientation(p: Point2D, q: Point2D, r:
Point2D): number {
    const val = (q.y - p.y) * (r.x - q.x) - (q.x
- p.x) * (r.y - q.y);
    return val === 0 ? 0 : (val > 0 ? 1 : 2); //
0 -> Collinear, 1 -> Clockwise, 2 ->
Counterclockwise
}
```

```
function onSegment(p: Point2D, q: Point2D, r:
Point2D): boolean {
    return (
        q.x <= Math.max(p.x, r.x) && q.x >=
Math.min(p.x, r.x) &&
        q.y <= Math.max(p.y, r.y) && q.y >=
Math.min(p.y, r.y)
    );
}
```

```
function doIntersect(p1: Point2D, q1: Point2D,
p2: Point2D, q2: Point2D): boolean {
    const o1 = orientation(p1, q1, p2);
    const o2 = orientation(p1, q1, q2);
    const o3 = orientation(p2, q2, p1);
    const o4 = orientation(p2, q2, q1);

    if (o1 !== o2 && o3 !== o4) return true;
```

```

    if (o1 === 0 && onSegment(p1, p2, q1)) return
    true;
    if (o2 === 0 && onSegment(p1, q2, q1)) return
    true;
    if (o3 === 0 && onSegment(p2, p1, q2)) return
    true;
    if (o4 === 0 && onSegment(p2, q1, q2)) return
    true;

    return false;
}

```

```

// Example
const p1 = { x: 1, y: 1 }, q1 = { x: 10, y: 1
};
const p2 = { x: 1, y: 2 }, q2 = { x: 10, y: 2
};
console.log(doIntersect(p1, q1, p2, q2)); //
Output: false

```

Closest Pair of Points in TypeScript

The **Closest Pair of Points** problem finds the two closest points in a set. Using a **Divide and Conquer** approach achieves $O(n \log^2 n)$ $O(n \log n)$ $O(n \log n)$ complexity.

Implementation

```

function distance(p1: Point2D, p2: Point2D):
number {
    return Math.sqrt((p1.x - p2.x) ** 2 + (p1.y
- p2.y) ** 2);
}

```



```

function closestPair(points: Point2D[]):
number {
    if (points.length < 2) return Infinity;

    points.sort((a, b) => a.x - b.x);

    function closestUtil(pts: Point2D[]): number
    {
        if (pts.length <= 3) {
            let minDist = Infinity;
            for (let i = 0; i < pts.length; i++) {
                for (let j = i + 1; j < pts.length;
j++) {
                    minDist = Math.min(minDist,
distance(pts[i], pts[j]));
                }
            }
            return minDist;
        }

        const mid = Math.floor(pts.length / 2);
        const midPoint = pts[mid];

        const left = pts.slice(0, mid);
        const right = pts.slice(mid);

        const dLeft = closestUtil(left);
        const dRight = closestUtil(right);

        let d = Math.min(dLeft, dRight);

        const strip: Point2D[] = [];
        for (const pt of pts) {

```

```

        if (Math.abs(pt.x - midPoint.x) < d)
strip.push(pt);
    }

    strip.sort((a, b) => a.y - b.y);

    for (let i = 0; i < strip.length; i++) {
        for (let j = i + 1; j < strip.length &&
(strip[j].y - strip[i].y) < d; j++) {
            d = Math.min(d, distance(strip[i],
strip[j]));
        }
    }

    return d;
}

return closestUtil(points);
}

```

```

// Example
const pointsArray: Point2D[] = [
    { x: 2, y: 3 }, { x: 12, y: 30 }, { x: 40, y:
50 },
    { x: 5, y: 1 }, { x: 12, y: 10 }, { x: 3, y:
4 }
];
console.log(closestPair(pointsArray));      //
Output: Closest distance

```

Conclusion

Computational geometry provides powerful tools to solve spatial problems. Algorithms like **Convex Hull**, **Line Segment Intersection**, and **Closest Pair of Points**, implemented in TypeScript, enable efficient geometric computations essential for GIS, robotics, and computer graphics.

Parallel Algorithms in TypeScript

Parallel algorithms leverage multiple processors or threads to perform computations simultaneously, improving efficiency for large-scale problems. This section explores **models of parallel computation**, techniques for **parallel sorting**, and **parallel graph algorithms** implemented in TypeScript.

Introduction to Parallel Computing

Parallel computing distributes tasks across multiple processors to solve computational problems faster than a single processor could. It is crucial in fields like data processing, simulation, and AI.

Models of Parallel Computation

1. **Shared Memory Model:** Processors share a common memory space, with communication done via shared variables. Example: Multithreading in a single system.
2. **Distributed Memory Model:** Processors have separate memories, and communication happens via message passing. Example: Systems like Hadoop and MPI.
3. **Hybrid Model:** Combines shared and distributed memory models for more flexibility.

Parallel Algorithm Techniques

Parallel Sorting in TypeScript

Sorting large datasets can benefit significantly from parallel processing. **Merge Sort** is particularly well-suited for parallelization due to its divide-and-conquer structure.

Parallel Merge Sort

```
import { Worker } from "worker_threads"; //
Node.js API for multithreading
```

```
// Merge function
```

```
function merge(left: number[], right:
number[]): number[] {
```

```
    const result: number[] = [];
```

```
    let i = 0, j = 0;
```

```
    while (i < left.length && j < right.length)
    {
        if (left[i] < right[j]) {
            result.push(left[i++]);
        } else {
            result.push(right[j++]);
        }
    }
}
```

```
    return
```

```
result.concat(left.slice(i)).concat(right.sli
ce(j));
}
```

```
// Parallel Merge Sort
```

```
async function parallelMergeSort(arr:
number[]): Promise<number[]> {
```

```
    if (arr.length <= 1) return arr;
```

```
    const mid = Math.floor(arr.length / 2);
```

```
    const left = arr.slice(0, mid);
```

```
    const right = arr.slice(mid);
```

```

    const sortWorker = (array: number[]):
    Promise<number[]> =>
        new Promise((resolve, reject) => {
            const worker = new
    Worker("./sortWorker.js", { workerData: array
    });
            worker.on("message", resolve);
            worker.on("error", reject);
            worker.on("exit", (code) => {
                if (code !== 0) reject(new
    Error(`Worker stopped with exit code
    ${code}`));
            });
        });

    const [sortedLeft, sortedRight] = await
    Promise.all([
        sortWorker(left),
        sortWorker(right)
    ]);

    return merge(sortedLeft, sortedRight);
}

// Example
(async () => {
    const array = [38, 27, 43, 3, 9, 82, 10];
    console.log(await parallelMergeSort(array));
})();

```

Note: This implementation requires a worker script (sortWorker.js) for sorting partitions. The example

assumes you have worker support via Worker Threads in Node.js.

Parallel Graph Algorithms in TypeScript

Graphs benefit from parallel processing in algorithms like breadth-first search (BFS) and shortest path calculations. Parallel versions can speed up traversal and computation significantly.

Parallel BFS

In a **parallel BFS**, multiple threads explore different layers of the graph simultaneously. Here's a simplified **implementation**:

```
type Graph = Record<number, number[]>;

async function parallelBFS(graph: Graph,
start: number): Promise<number[]> {
  const visited = new Set<number>();
  const queue: number[] = [start];
  const results: number[] = [];

  while (queue.length > 0) {
    const levelSize = queue.length;
    const tasks: Promise<void>[] = [];

    for (let i = 0; i < levelSize; i++) {
      const node = queue.shift()!;
      if (!visited.has(node)) {
        visited.add(node);
        results.push(node);

        tasks.push(
```

```

        new Promise((resolve) => {
            graph[node].forEach((neighbor) =>
        {
            if (!visited.has(neighbor)) {
                queue.push(neighbor);
            }
        });
        resolve();
    })
    );
}
}

    await Promise.all(tasks);
}

    return results;
}

```

// Example

```

const graph: Graph = {
    0: [1, 2],
    1: [0, 3, 4],
    2: [0, 5],
    3: [1],
    4: [1, 5],
    5: [2, 4],
};

```

```

(async () => {
    console.log(await parallelBFS(graph, 0));
})();

```


Conclusion

Parallel algorithms unlock the potential of modern multicore systems. Techniques like **parallel sorting** and **graph traversal** ensure faster computation for complex problems. Using TypeScript with frameworks or tools like **Worker Threads** and **Web Workers** bridges the gap between high-level programming and efficient parallel computation. As data sizes continue to grow, embracing parallelism is increasingly critical for scalable solutions.

Approximation Algorithms in TypeScript

Approximation algorithms are designed to find near-optimal solutions to computational problems where finding the exact solution is infeasible due to time or resource constraints. These algorithms guarantee a solution within a specific ratio of the optimal solution.

Introduction to Approximation Algorithms

What are Approximation Algorithms?

Approximation algorithms provide a way to tackle optimization problems, particularly **NP-Hard problems**, by finding solutions that are close to the best possible solution in a reasonable amount of time.

Key Features

- **Efficiency:** Runs in polynomial time.
- **Approximation Ratio:** The ratio between the solution's value and the optimal solution's value.
- **Applicability:** Used in real-world scenarios like scheduling, routing, and resource allocation.

Common Techniques and Applications in TypeScript

1. Greedy Approximation

Problem: Vertex Cover

A **Vertex Cover** of a graph is a set of vertices such that every edge in the graph has at least one endpoint in this set.

Algorithm

1. Pick an arbitrary edge.
2. Add both endpoints of the edge to the vertex cover.
3. Remove all edges incident to these vertices.
4. Repeat until no edges are left.

Implementation

```
type Graph = Record<number, number[]>;

function vertexCover(graph: Graph): number[] {
  const covered = new Set<number>();
  const edges = new Set<[number, number]>();

  // Collect all edges
  for (const node in graph) {
    for (const neighbor of graph[node]) {
      if (Number(node) < neighbor) {
        edges.add([Number(node), neighbor]);
      }
    }
  }

  const vertexCoverSet: number[] = [];
  for (const [u, v] of edges) {
    if (!covered.has(u) && !covered.has(v)) {
      vertexCoverSet.push(u, v);
      covered.add(u);
      covered.add(v);
    }
  }

  return vertexCoverSet;
}

// Example
const graph: Graph = {
  0: [1, 2],
  1: [0, 3],
```

```
2: [0, 3],  
3: [1, 2]  
};
```

```
console.log(vertexCover(graph)); // Output:  
[0, 1] or other valid covers
```

2. Local Search

Problem: Traveling Salesman Problem (TSP)

In TSP, the goal is to find the shortest possible route that visits each city exactly once and returns to the starting city.

Algorithm

1. Start with a random tour.
2. Improve the tour by swapping two edges to reduce the total distance.
3. Repeat until no further improvement is possible.

Implementation

```
function tspLocalSearch(distanceMatrix:
number[][]): number[] {
    const n = distanceMatrix.length;
    let tour = Array.from({ length: n }, (_, i)
=> i);

    // Calculate tour length
    const tourLength = (tour: number[]): number
=>
        tour.reduce(
            (sum, city, i) => sum +
distanceMatrix[city][tour[(i + 1) % n]],
            0
        );

    let improved = true;

    while (improved) {
        improved = false;

        for (let i = 1; i < n - 1; i++) {
            for (let j = i + 1; j < n; j++) {
                const newTour = [...tour];
                [newTour[i], newTour[j]] =
[newTour[j], newTour[i]];

                if (tourLength(newTour) <
tourLength(tour)) {
                    tour = newTour;
                    improved = true;
                }
            }
        }
    }
}
```

```

    }
  }
}

return tour;
}

// Example
const distances = [
  [0, 10, 15, 20],
  [10, 0, 35, 25],
  [15, 35, 0, 30],
  [20, 25, 30, 0]
];

console.log(tspLocalSearch(distances));    //
Output: Approximate shortest tour

```

3. Dynamic Programming Approximation

Problem: Knapsack Problem

The goal is to maximize the total value of items that can be carried in a knapsack of fixed capacity.

Algorithm

1. Use dynamic programming to compute solutions for smaller subproblems.
2. Approximate solutions are derived from a fixed precision parameter.

Implementation

```
function knapsackApproximation(
  weights: number[],
  values: number[],
  capacity: number,
  epsilon: number
): number {
  const n = weights.length;
  const maxValue = Math.max(...values);

  // Scale values to reduce the problem size
  const scale = epsilon * maxValue / n;
  const scaledValues = values.map((value) =>
    Math.floor(value / scale));

  const dp = Array.from({ length: capacity + 1
  }, () => 0);

  for (let i = 0; i < n; i++) {
    for (let w = capacity; w >= weights[i]; w-
    -) {
      dp[w] = Math.max(dp[w], dp[w -
      weights[i]] + scaledValues[i]);
    }
  }

  return dp[capacity] * scale; // Scale back to
  approximate value
}

// Example
const weights = [2, 3, 4, 5];
const values = [3, 4, 5, 6];
```

```
const capacity = 5;  
const epsilon = 0.1;  
  
console.log(knapsackApproximation(weights,  
values, capacity, epsilon)); // Approximate  
maximum value
```

Conclusion

Approximation algorithms provide practical solutions for computationally hard problems. Techniques like **greedy algorithms**, **local search**, and **dynamic programming approximations** help balance the trade-off between optimality and efficiency. Implementing these algorithms in TypeScript showcases their versatility and relevance to real-world applications.

Randomized Algorithms in TypeScript

Randomized algorithms use random numbers to make decisions during their execution. They are particularly useful for problems where deterministic algorithms are inefficient or too complex. The randomness introduced helps achieve simplicity, speed, or fairness.

Introduction to Randomized Algorithms

What Are Randomized Algorithms?

Randomized algorithms make random choices as part of their logic. These algorithms might produce different results or follow different execution paths for the same input. They are categorized into:

1. **Las Vegas Algorithms:** Always produce a correct result, but the time complexity may vary.
2. **Monte Carlo Algorithms:** May produce incorrect results but do so with a controllable probability.

Advantages

- Simplicity of implementation.
- Faster than deterministic counterparts in many cases.
- Useful for parallel processing and large datasets.

Disadvantages

- Non-deterministic execution may be hard to debug.
- Reliability depends on the quality of the random number generator.

Examples and Applications in TypeScript

1. Randomized QuickSort

QuickSort is a divide-and-conquer sorting algorithm. Randomized QuickSort selects a pivot randomly, improving its performance on already sorted or skewed data.

Implementation

```
function randomizedQuickSort(arr: number[],
low: number, high: number): void {
    if (low < high) {
        const pivotIndex = randomPartition(arr,
low, high);
        randomizedQuickSort(arr, low, pivotIndex -
1);
        randomizedQuickSort(arr, pivotIndex + 1,
high);
    }
}
```

```
function randomPartition(arr: number[], low:
number, high: number): number {
    const randomIndex = Math.floor(Math.random()
* (high - low + 1)) + low;
    [arr[randomIndex], arr[high]] = [arr[high],
arr[randomIndex]];
    return partition(arr, low, high);
}
```

```
function partition(arr: number[], low: number,
high: number): number {
    const pivot = arr[high];
```

```

let i = low - 1;

for (let j = low; j < high; j++) {
  if (arr[j] < pivot) {
    i++;
    [arr[i], arr[j]] = [arr[j], arr[i]];
  }
}
[arr[i + 1], arr[high]] = [arr[high], arr[i
+ 1]];
return i + 1;
}

```

```

// Example
const array = [3, 6, 8, 10, 1, 2, 1];
randomizedQuickSort(array, 0, array.length -
1);
console.log(array); // Output: [1, 1, 2, 3, 6,
8, 10]

```

2. Monte Carlo Algorithm: Primality Testing

This algorithm determines if a number is prime. It may incorrectly identify a composite number as prime but does so with a small probability.

Implementation

```

function isProbablyPrime(n: number, k: number
= 5): boolean {
  if (n <= 1) return false;
  if (n <= 3) return true;

  for (let i = 0; i < k; i++) {

```

```

    const a = Math.floor(Math.random() * (n -
2)) + 2; // Random number in [2, n-1]
    if (gcd(a, n) !== 1 ||
modularExponentiation(a, n - 1, n) !== 1) {
        return false;
    }
}
return true;
}

```

```

function gcd(a: number, b: number): number {
    return b === 0 ? a : gcd(b, a % b);
}

```

```

function modularExponentiation(base: number,
exp: number, mod: number): number {
    let result = 1;
    base = base % mod;
    while (exp > 0) {
        if (exp % 2 === 1) {
            result = (result * base) % mod;
        }
        exp = Math.floor(exp / 2);
        base = (base * base) % mod;
    }
    return result;
}

```

```

// Example
console.log(isProbablyPrime(97)); // Output:
true (likely prime)
console.log(isProbablyPrime(100)); // Output:
false

```

3. Reservoir Sampling

Reservoir Sampling is a technique for selecting k random items from a stream of n items when n is unknown or very large.

Implementation

```
function reservoirSampling(stream: number[],
k: number): number[] {
  const reservoir = stream.slice(0, k);

  for (let i = k; i < stream.length; i++) {
    const j = Math.floor(Math.random() * (i + 1));
    if (j < k) {
      reservoir[j] = stream[i];
    }
  }

  return reservoir;
}
```

```
// Example
const stream = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
const sampled = reservoirSampling(stream, 3);
console.log(sampled); // Output: Random 3 items
from the stream
```

4. Randomized Binary Search

This variation of binary search picks a random middle element instead of the deterministic middle element.

Implementation

```
function randomizedBinarySearch(arr: number[],
target: number): number {
  let low = 0;
  let high = arr.length - 1;

  while (low <= high) {
    const mid = low + Math.floor(Math.random()
* (high - low + 1));
    if (arr[mid] === target) return mid;
    if (arr[mid] < target) low = mid + 1;
    else high = mid - 1;
  }
  return -1;
}
```

// Example

```
const sortedArray = [1, 2, 3, 4, 5, 6, 7, 8,
9];
console.log(randomizedBinarySearch(sortedArra
y, 5)); // Output: Index of 5
```

Applications

- **Cryptography:** Randomized algorithms secure encryption and key generation.
- **Machine Learning:** Random forests and stochastic gradient descent use randomness for better results.
- **Network Algorithms:** Algorithms like randomized routing in networks.

- **Big Data:** Sampling techniques and approximate query processing.

Conclusion

Randomized algorithms offer powerful tools for solving problems efficiently by leveraging randomness. From sorting to primality testing and sampling, their applications span multiple domains. Implementing these in TypeScript demonstrates their simplicity, versatility, and practicality in modern computing tasks.

Complexity Theory

Complexity theory is a branch of theoretical computer science that studies the resources required to solve computational problems, such as time and space. It categorizes problems based on their inherent difficulty and the computational power needed to solve them.

Introduction to Complexity Classes

What Are Complexity Classes?

Complexity classes group problems based on the resources (time, space, etc.) needed to solve them on a computational model like a Turing machine.

1. **P (Polynomial Time)**
Problems solvable in polynomial time ($O(n^k)$ for some constant k). These are considered "easy" or "tractable" problems.
 - Example: Sorting a list, finding the shortest path in an unweighted graph (Breadth-First Search).
2. **NP (Non-deterministic Polynomial Time)**
Problems for which a solution can be verified in polynomial time.
 - Example: Sudoku puzzles, where verifying a filled grid is faster than solving it.
3. **NP-Complete**
The hardest problems in NP. If any NP-complete problem can be solved in polynomial time, all NP problems can be solved in polynomial time.
 - Example: Traveling Salesman Problem, Boolean Satisfiability Problem (SAT).

4. NP-Hard

Problems at least as hard as NP-complete problems but are not necessarily in NP. These may not have verifiable solutions in polynomial time.

- Example: Halting Problem, optimization variants of NP problems.

P, NP, NP-Complete, and NP-Hard

Key Questions

1. Does $P = NP$?

The most famous open problem in computer science. If $P = NP$, every problem whose solution can be verified quickly can also be solved quickly.

2. Why Are These Classes Important?

- Understanding these classes helps determine whether a problem is feasible to solve.
- Guides algorithm design and resource allocation.

Relationships

- $P \subseteq NP$: Every problem solvable in polynomial time can also be verified in polynomial time.
- $NP\text{-Complete} \subseteq NP$: NP-complete problems are a subset of NP.
- $NP\text{-Hard} \supseteq NP\text{-Complete}$: NP-hard includes NP-complete and potentially harder problems.

Reductions and Completeness

What Is Reduction?

Reduction is a method of transforming one problem into another. If problem A can be reduced to problem B, solving B also solves A.

- **Polynomial-Time Reduction:** A reduction that can be computed in polynomial time.
- **Importance:** Helps classify problems by showing their relative difficulty.

Completeness

- A problem is **NP-complete** if:
 1. It is in NP.
 2. Every other problem in NP can be reduced to it in polynomial time.

Examples and Applications in TypeScript

1. Solving SAT Problem

The SAT problem asks if there exists an assignment of variables that satisfies a given Boolean formula.

Implementation

```
function isSatisfiable(clauses: number[][][], n:
number): boolean {
    const totalCombinations = 1 << n; // 2^n
    combinations of variables

    for (let mask = 0; mask < totalCombinations;
mask++) {
        let valid = true;

        for (const clause of clauses) {
            let clauseSatisfied = false;
            for (const literal of clause) {
                const variable = Math.abs(literal) - 1;
                const value = (mask & (1 << variable))
                !== 0;
                if ((literal > 0 && value) || (literal
                < 0 && !value)) {
```

```

        clauseSatisfied = true;
        break;
    }
}
if (!clauseSatisfied) {
    valid = false;
    break;
}
}

if (valid) return true;
}

return false;
}

// Example
const clauses = [
    [1, -2], // x1 OR NOT x2
    [-1, 2], // NOT x1 OR x2
];
console.log(isSatisfiable(clauses, 2)); //
Output: true

```

2. NP-Hard Problem: Traveling Salesman Problem (TSP)

The TSP asks for the shortest possible route that visits every city exactly once and returns to the origin.

Naive Implementation (Exponential Time)

```
function tsp(graph: number[][]): number {
  const n = graph.length;
  const visitedAll = (1 << n) - 1;

  function tspHelper(mask: number, pos:
number, memo: number[][]): number {
    if (mask === visitedAll) return
graph[pos][0] || Infinity;

    if (memo[mask][pos] !== -1) return
memo[mask][pos];

    let answer = Infinity;
    for (let city = 0; city < n; city++) {
      if ((mask & (1 << city)) === 0) {
        const newAnswer = graph[pos][city] +
tspHelper(mask | (1 << city), city, memo);
        answer = Math.min(answer, newAnswer);
      }
    }

    return (memo[mask][pos] = answer);
  }

  const memo = Array.from({ length: 1 << n },
() => Array(n).fill(-1));
  return tspHelper(1, 0, memo);
}

// Example
```

```
const graph = [  
  [0, 10, 15, 20],  
  [10, 0, 35, 25],  
  [15, 35, 0, 30],  
  [20, 25, 30, 0],  
];  
console.log(tsp(graph)); // Output: 80
```

Conclusion

Complexity theory provides the foundation for understanding the computational difficulty of problems and helps classify them into well-defined classes like P, NP, NP-complete, and NP-hard. Techniques like reductions and algorithm analysis assist in designing efficient solutions or proving intractability.

By implementing complexity-related problems in TypeScript, we can develop a better appreciation of the interplay between theoretical computer science and practical programming. Complexity classes like P and NP continue to challenge and inspire computer scientists worldwide.

Practical Considerations for Algorithms and Data Structures in TypeScript

When implementing algorithms and data structures in TypeScript, a practical approach ensures reliability, maintainability, and optimal performance. This section covers essential implementation tips, debugging techniques, and performance tuning practices to make your TypeScript programs efficient and robust.

Implementation Tips

1. Start with a Plan

- **Understand the Problem:** Break the problem into smaller parts and create a step-by-step plan for implementation.
- **Choose the Right Data Structure:** Use data structures that best match your use case. For example:
 - Use arrays or linked lists for sequential data.
 - Use hash tables for quick lookups.
 - Use trees or graphs for hierarchical or networked data.

2. Write Modular Code

- Divide your code into smaller, reusable functions or classes.
- Example: Implement a `PriorityQueue` class for graph algorithms rather than re-implementing priority logic multiple times.

3. Type Safety

- Leverage TypeScript's type system to enforce data correctness.

- Define types for inputs and outputs to catch errors at compile time.

Example: Defining Custom Types

```
type Graph = { [key: string]: string[] };
```

```
const graph: Graph = {  
  A: ["B", "C"],  
  B: ["A", "D"],  
  C: ["A", "D"],  
  D: ["B", "C"],  
};
```

Debugging and Testing in TypeScript

1. Debugging Techniques

- **Use Console Logs:** Temporarily add logs to inspect variables, loops, or function results.
- **Debugging Tools:** Use browser developer tools or Node.js debuggers.
- **Breakpoints:** Set breakpoints to pause code execution and inspect the program state.

Example: Using Console Logs

```
function binarySearch(arr: number[], target:  
number): number {  
  let low = 0, high = arr.length - 1;  
  while (low <= high) {  
    const mid = Math.floor((low + high) / 2);  
    console.log(`Low: ${low}, High: ${high},  
Mid: ${mid}`);  
    if (arr[mid] === target) return mid;  
    if (arr[mid] < target) low = mid + 1;  
    else high = mid - 1;  
  }  
}
```

```
    return -1;
}
```

2. Unit Testing

- Use frameworks like **Jest** or **Mocha** to write unit tests for individual components.
- Test edge cases, such as empty arrays, null inputs, or maximum data limits.

Example: Jest Test Case

```
test('binarySearch finds the correct index', ()
=> {
  const arr = [1, 3, 5, 7, 9];
  expect(binarySearch(arr, 5)).toBe(2);
  expect(binarySearch(arr, 1)).toBe(0);
  expect(binarySearch(arr, 10)).toBe(-1);
});
```

3. TypeScript-Specific Tools

- **TypeScript Compiler**: Use tsc to check for syntax and type errors.
- **Linting**: Use ESLint with TypeScript plugins to enforce code quality standards.

Performance Tuning in TypeScript

1. Analyze Complexity

- Optimize algorithms by reducing their time and space complexity.
- Example: Replace a $O(n^2)$ loop with a $O(n \log n)$ sorting algorithm.

2. Efficient Data Structures

- Use the right data structures for specific operations.
 - **Set** for uniqueness constraints.
 - **Map** for fast key-value lookups.

- **Heap** for priority-based operations.

Example: Using a Heap for Priority Queue

```
class MinHeap {
  private heap: number[] = [];

  insert(val: number): void {
    this.heap.push(val);
    this.bubbleUp();
  }

  extractMin(): number | null {
    if (this.heap.length === 0) return null;
    const min = this.heap[0];
    const end = this.heap.pop();
    if (this.heap.length > 0 && end !==
undefined) {
      this.heap[0] = end;
      this.bubbleDown();
    }
    return min;
  }

  private bubbleUp() {
    let index = this.heap.length - 1;
    const element = this.heap[index];
    while (index > 0) {
      const parentIndex = Math.floor((index - 1)
/ 2);
      const parent = this.heap[parentIndex];
      if (element >= parent) break;
      this.heap[index] = parent;
      this.heap[parentIndex] = element;
      index = parentIndex;
    }
  }
}
```

```

    }
}

private bubbleDown() {
    let index = 0;
    const length = this.heap.length;
    const element = this.heap[index];

    while (true) {
        const leftChildIndex = 2 * index + 1;
        const rightChildIndex = 2 * index + 2;
        let leftChild, rightChild;
        let swap = null;

        if (leftChildIndex < length) {
            leftChild = this.heap[leftChildIndex];
            if (leftChild < element) swap =
leftChildIndex;
        }
        if (rightChildIndex < length) {
            rightChild =
this.heap[rightChildIndex];
            if (
                (swap === null && rightChild <
element) ||
                (swap !== null && rightChild <
leftChild!)
            )
                swap = rightChildIndex;
        }
        if (swap === null) break;
        this.heap[index] = this.heap[swap];
        this.heap[swap] = element;
    }
}

```

```

        index = swap;
    }
}
}

```

3. Minimize Redundant Calculations

- Use **memoization** or **caching** for dynamic programming problems.

Example: Memoized Fibonacci Sequence

```

function fib(n: number, memo: Record<number,
number> = {}): number {
    if (n <= 1) return n;
    if (memo[n]) return memo[n];
    memo[n] = fib(n - 1, memo) + fib(n - 2, memo);
    return memo[n];
}

```

4. Optimize Loops and Conditions

- Use **early exits** in loops to minimize unnecessary iterations.
- Combine conditions to reduce computational overhead.

Conclusion

Practical considerations for implementing algorithms and data structures in TypeScript include thoughtful planning, leveraging TypeScript's powerful type system, and adhering to best practices for debugging, testing, and performance optimization. By adopting these practices, developers can write efficient, maintainable, and error-resistant code, ensuring that their TypeScript projects meet both functional and performance requirements.

Appendices

This section provides supplementary materials that can enhance the understanding and implementation of algorithms and data structures in TypeScript. It includes mathematical notations, a cheat sheet for common algorithms, useful TypeScript libraries, and references for further reading.

Mathematical Notations

Understanding the mathematical notations commonly used in algorithms is crucial for grasping the underlying principles and optimizations. Here's a quick reference:

- **Big O Notation:**
 - Represents the **upper bound** of the complexity, showing the worst-case scenario for time or space.
 - Example: $O(n)$ indicates that the time complexity grows linearly with the input size.
- **Big Omega (Ω) Notation:**
 - Represents the **lower bound** of the complexity, indicating the best-case scenario.
 - Example: $\Omega(n)$ indicates that in the best case, the time complexity is linear.
- **Big Theta (Θ) Notation:**
 - Represents **both upper and lower bounds**, providing a tight bound on the complexity.
 - Example: $\Theta(n \log n)$ indicates that the algorithm's complexity will grow both in the best and worst case at this rate.

- **Recurrence Relation:**
 - Describes a problem in terms of smaller instances of itself. Used extensively in divide and conquer algorithms.
 - Example: $T(n) = 2T(n/2) + O(n)$ is the recurrence relation for merge sort.
- **Summation:**
 - Used to represent the sum of a sequence of numbers. Common in analyzing algorithms that loop over an array or matrix.
 - Example: $\sum_{i=1}^n i$ represents the sum of the first n integers.

Common Typescript Cheat Sheet

Feature	Example	Description
Variable Declaration	<pre>let name: string = "John";</pre>	Declares a variable with a type.
Function	<pre>function add(a: number, b: number): number { return a + b; }</pre>	Declares a function with parameter and return types.
Arrow Function	<pre>const greet = (name: string): string => \Hello, \${name}`;</pre>	Arrow function with type annotations.
Interface	<pre>interface User { id: number; name: string; }</pre>	Defines a structure for objects.
Class	<pre>class Person { constructor(public name: string) {} }</pre>	Creates a class with a constructor.

Generics	<pre>function identity<T>(arg: T): T { return arg; }</pre>	Creates reusable components with type placeholders.
Union Types	<pre>`let value: string number;`</pre>	
Intersection Types	<pre>type Person = { name: string; } & { age: number; };</pre>	Combines multiple types into one.
Enums	<pre>enum Color { Red, Green, Blue }</pre>	Defines a set of named constants.
Type Assertions	<pre>let value: any = "hello"; let strLength: number = (value as string).length;</pre>	Tells the compiler to treat a variable as a specific type.
Nullable Types	<pre>`let val: string null;`</pre>	
Type Aliases	<pre>type Point = { x: number; y: number; };</pre>	Creates a custom name for a type.
Optional Parameters	<pre>`function greet(name?: string): string { return name</pre>	
Readonly	<pre>interface Point { readonly x: number; readonly y: number; }</pre>	Marks properties as immutable.
Tuple	<pre>let tuple: [number, string] = [1, "hello"];</pre>	Defines a fixed-length array with

		specified types for each element.
Default Parameters	<pre>function multiply(a: number, b: number = 2): number { return a * b; }</pre>	Provides default values for function parameters.
Type Guards	<pre>function isString(x: any): x is string { return typeof x === "string"; }</pre>	Custom logic to narrow types within code.
Mapped Types	<pre>type Readonly<T> = { readonly [K in keyof T]: T[K]; };</pre>	Transforms an object type into another type.
Modules	<pre>export const myVar = 10; import { myVar } from './file';</pre>	Exports and imports code across files.
Decorators	<pre>@Component({...}) class MyComponent { ... }</pre>	Special syntax for modifying class behavior (experimental feature).

Common Algorithms Cheat Sheet

Here is a quick reference to some of the most common algorithms and their time complexities:

1. Sorting Algorithms

Algorithm	Best Case	Worst Case	Average Case	Space Complexity
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Quick Sort	$O(n \log n)$	$O(n^2)$	$O(n \log n)$	$O(\log n)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$

2. Searching Algorithms

Algorithm	Best Case	Worst Case	Average Case	Space Complexity
Binary Search	$O(1)$	$O(\log n)$	$O(\log n)$	$O(1)$
Linear Search	$O(1)$	$O(n)$	$O(n)$	$O(1)$

3. Dynamic Programming

Problem	Algorithm	Complexity
Longest Common Subsequence	$O(n * m)$	
Knapsack Problem	$O(n * W)$	
Matrix Chain Multiplication	$O(n^3)$	

4. Graph Algorithms

Algorithm	Purpose	Complexity
Breadth-First Search (BFS)	Graph traversal	$O(V + E)$
Depth-First Search (DFS)	Graph traversal	$O(V + E)$
Dijkstra's Algorithm	Shortest path	$O(V^2)$ or $O(E + V \log V)$ (using Priority Queue)
Kruskal's Algorithm	Minimum Spanning Tree	$O(E \log E)$
Prim's Algorithm	Minimum Spanning Tree	$O(V^2)$ or $O(E + V \log V)$ (with Priority Queue)

5. String Algorithms

Algorithm	Purpose	Complexity
Rabin-Karp Algorithm	Pattern matching	$O(n + m)$
KMP Algorithm	Pattern matching	$O(n + m)$

Z-Algorithm	Pattern matching	$O(n + m)$
--------------------	------------------	------------

6. Greedy Algorithms

Problem	Algorithm	Complexity
Activity Selection	Greedy	$O(n \log n)$
Huffman Encoding	Greedy	$O(n \log n)$

7. Divide and Conquer

Problem	Algorithm	Complexity
Maximum Subarray	Kadane's Algorithm	$O(n)$
Matrix Multiplication	Strassen's Algorithm	$O(n^{2.81})$

Sorting Algorithms

- **Bubble Sort:** $O(n^2)$ (worst), $O(n)$ (best)
- **Selection Sort:** $O(n^2)$
- **Insertion Sort:** $O(n^2)$ (worst), $O(n)$ (best)
- **Merge Sort:** $O(n \log n)$
- **Quick Sort:** $O(n \log n)$ (average), $O(n^2)$ (worst)
- **Heap Sort:** $O(n \log n)$

Searching Algorithms

- **Linear Search:** $O(n)$
- **Binary Search:** $O(\log n)$ (requires sorted data)

Graph Algorithms

- **Breadth-First Search (BFS):** $O(V + E)$ (where V is vertices and E is edges)
- **Depth-First Search (DFS):** $O(V + E)$

- **Dijkstra's Algorithm:** $O(E + V \log V)$ (using min-heap)
- **Floyd-Warshall:** $O(V^3)$

Dynamic Programming

- **Fibonacci (memoized):** $O(n)$
- **Knapsack Problem:** $O(nW)$ where n is the number of items and W is the capacity of the knapsack

TypeScript Libraries for Data Structures and Algorithms

While TypeScript provides powerful native support for many algorithms and data structures, several libraries can speed up development and offer optimized implementations:

1. typescript-collections

- A collection of common data structures (Stack, Queue, Set, Map, LinkedList).
- **Installation:** npm install typescript-collections
- **Example:**

```
import { Dictionary } from 'typescript-collections';
const dictionary = new Dictionary<number, string>();
dictionary.setValue(1, 'apple');
dictionary.setValue(2, 'banana');
console.log(dictionary.getValue(1)); // "apple"
```

2. js-sdsl (Standard Data Structures Library)

- Provides advanced data structures like AVL trees, Red-Black trees, and heaps.
- **Installation:** npm install js-sdsl

- **Example:**

```
import { AVLTree } from 'js-sdsl';
const tree = new AVLTree<number>();
tree.insert(5);
tree.insert(3);
tree.insert(7);
console.log(tree.find(5)); // true
```

3. collections.js

- A library for working with various collection types such as stacks, queues, dequeues, and more.
- **Installation:** npm install collections
- **Example:**

```
import { Stack } from 'collections';
const stack = new Stack<number>();
stack.push(1);
stack.push(2);
console.log(stack.pop()); // 2
```

4. pathfinding

- A specialized library for pathfinding algorithms like A* and Dijkstra's algorithm.
- **Installation:** npm install pathfinding
- **Example:**

```
import { AStarFinder } from 'pathfinding';
const grid = new PF.Grid([
  [0, 0, 0, 0],
  [0, 1, 1, 0],
  [0, 0, 0, 0],
]);
const finder = new AStarFinder();
const path = finder.findPath(0, 0, 3, 0, grid);
console.log(path); // Array of coordinate
```

5. lodash

- A utility library that provides helpful functions for array manipulation and common operations that can simplify algorithmic code.
- **Installation:** npm install lodash
- **Example:**

```
import * as _ from 'lodash';  
const arr = [1, 2, 3, 4];  
const reversed = _.reverse(arr);  
console.log(reversed); // [4, 3, 2, 1]
```

References

Here are some useful resources and references for further reading:

1. Books:

- *"Introduction to Algorithms"* by Cormen, Leiserson, Rivest, and Stein (commonly known as CLRS) - A comprehensive book on algorithms.
- *"Data Structures and Algorithms in TypeScript"* by Loiane Groner - Focuses on implementing common data structures and algorithms in TypeScript.

2. Online Resources:

- [GeeksforGeeks](https://www.geeksforgeeks.org/) - A treasure trove of algorithm explanations and examples.
- Visualgo - A great tool to visually understand algorithms and data structures.
- TypeScript Documentation - Official documentation for TypeScript, which can help with proper syntax and best practices.

3. Academic Papers:

- *"The Art of Computer Programming"* by Donald E. Knuth - A classic reference for deep insights into algorithms and data structures.
- *"Algorithms"* by Robert Sedgewick and Kevin Wayne - A great academic resource with an emphasis on practical implementation.

4. Courses:

- [Coursera Algorithms Specialization](#) - A free course that dives deep into algorithms with JavaScript-based examples.
- Udemy Data Structures and Algorithms in JavaScript - Focuses on teaching data structures and algorithms in JavaScript and TypeScript.

With this appendices section, you now have the tools to dive deeper into the world of algorithms and data structures, both theoretically and practically, while working with TypeScript. Happy coding!

References

This section provides valuable resources, including books, research papers, online platforms, and modern AI tools, to further explore data structures, algorithms, and their implementation in TypeScript.

Online Resources

Websites and Platforms

1. **GeeksforGeeks** ([geeksforgeeks.org](https://www.geeksforgeeks.org))
 - Comprehensive tutorials on data structures and algorithms.
2. **LeetCode** (leetcode.com)
 - An interactive platform for practicing algorithm problems.
3. **HackerRank** ([hackerrank.com](https://www.hackerrank.com))
 - Offers challenges and solutions to hone programming skills.
4. **MDN Web Docs** (developer.mozilla.org)
 - Provides JavaScript and TypeScript documentation.
5. **TypeScript Official Website** (typescriptlang.org)
 - Authoritative resource for TypeScript documentation and examples.

AI-Powered Tools

1. **ChatGPT** (by OpenAI)
 - A conversational AI capable of providing explanations, examples, and assistance for coding challenges. Great for

brainstorming or debugging TypeScript implementations.

2. **Claude AI** (by Anthropic)

- An advanced AI that excels at understanding detailed queries and generating contextual responses for algorithm design.

3. **Perplexity AI**

- A research-oriented AI that provides succinct and relevant answers, including links to reputable resources for further exploration.

4. **CodePen AI**

- Assists in rapid prototyping and visualizing algorithms in TypeScript and other languages.

These resources, combined with the capabilities of modern AI tools, provide a robust foundation for learning and mastering data structures and algorithms.

About the Writer

Zidni Ridwan Nulmuarif

Software Engineer with over 5 years of experience

Zidni is an accomplished software engineer specializing in **frontend and mobile development**. With a passion for crafting scalable and efficient solutions, he has extensive experience in **React Native, TypeScript, and web technologies** like React.js, Next.js, and Angular.

Over the years, Zidni has worked on diverse projects ranging from **education platforms to lifestyle and on-demand service applications**, demonstrating a commitment to delivering impactful solutions for businesses and users alike. His expertise includes **state management (Redux, Zustand), native bridging, Google Maps integration, analytics, TDD**, and building design systems.

In addition to his professional work, Zidni actively contributes to the tech community through his blog and GitHub repositories, sharing insights and innovations to help others grow in their technical journeys.

Connect with me:

- **GitHub:** <https://github.com/zidniryi>
- **Blog:** <https://www.konsepkode.com/>
- **LinkedIn:** <https://www.linkedin.com/in/zidni-ridwan-nulmuarif/>

Feel free to reach out for collaboration, mentorship, or just to discuss exciting tech ideas!

Index

A

- **Algorithms**
 - Overview of Algorithms and Data Structures, 8
 - Sorting and Searching Algorithms in TypeScript, 195
 - Dynamic Programming in TypeScript, 201
 - Greedy Algorithms in TypeScript, 207
 - Backtracking and Branch & Bound in TypeScript, 224
 - Graph Algorithms in TypeScript, 245
 - Network Flow in TypeScript, 262
 - String Algorithms in TypeScript, 276
 - Computational Geometry in TypeScript, 284
 - Parallel Algorithms in TypeScript, 292
 - Approximation Algorithms in TypeScript, 298
 - Randomized Algorithms in TypeScript, 305
 - Complexity Theory, 312
- **Arrays**
 - Arrays and Linked Lists, 72
 - Arrays in TypeScript, 72
- **Advanced Data Structures**
 - Segment Trees, 233
 - Fenwick Trees (Binary Indexed Trees), 240
 - Suffix Trees and Arrays, 241

- K-D Trees, 243

B

- **Big O Notation**, 54
- **Binary Trees**, 99
- **Binary Search Trees**, 105
- **Bipartite Matching in TypeScript**, 269

C

- **Classes and Objects in TypeScript**, 48
- **Collision Resolution Techniques**, 148
- **Common Algorithms Cheat Sheet**, 328
- **Common Typescript Cheat Sheet**, 325

D

- **Data Structures**, 71
- **Deque in TypeScript**, 91
- **Disjoint Sets in TypeScript**, 182
- **Dynamic Programming**, 201

F

- **Floyd-Warshall Algorithm in TypeScript**, 259
- **Ford-Fulkerson Method in TypeScript**, 262

G

- **Graphs**, 156
 - Graph Representations in TypeScript, 157
 - Graph Traversal (BFS, DFS) in TypeScript, 164
 - Weighted Graphs (Dijkstra's, Floyd-Warshall) in TypeScript, 170
- **Greedy Algorithms**, 207
 - Principles of Greedy Algorithms, 207

H

- **Hashing**, 142
 - Hash Tables in TypeScript, 142
 - Collision Resolution Techniques, 148

I

- **Index Trees**, 240
- **Implementation Tips**, 318
- **Introduction to TypeScript**, 13
- **Introduction to Parallel Computing**, 292

K

- **Kruskal's Algorithm in TypeScript**, 245
- **K-D Trees**, 243

L

- **Lazy Propagation in TypeScript**, 236
- **Linked Lists (Singly, Doubly, Circular) in TypeScript**, 77

M

- **Mathematical Foundations**, 51
- **Minimum Spanning Trees**, 245
- **Matching and Covering in TypeScript**, 269

P

- **Pattern Matching**, 276
- **P, NP, NP-Complete, and NP-Hard**, 313
- **Parallel Algorithms**, 292
- **Parallel Sorting in TypeScript**, 292

R

- **Red-Black Trees**, 120
- **Recurrence Relations**, 66
- **References**, 335

S

- **Stacks and Queues**, 83
 - **Implementing Stacks in TypeScript**, 84
 - **Implementing Queues in TypeScript**, 88
- **Segment Trees**, 233
- **Suffix Trees**, 241
- **Suffix Trees and Arrays**, 241

T

- **Trees**, 97
 - Binary Trees, 99
 - Binary Search Trees, 105
- **TypeScript Basics**, 18
- **Type Annotations and Interfaces**, 48

U

- **Utility Algorithms**
 - Dynamic Programming, 201
 - Backtracking and Branch & Bound, 224

W

- **Weighted Graphs**, 170
 - Dijkstra's Algorithm, 255
 - Bellman-Ford Algorithm, 257