**500 HOURS SUMMER TRAINING REPORT**

On

**COMPLETE INTERVIEW PREPERATION**

Submitted by: SHASHANK SURYAWANSHI

Registration No. 12206727

Program Name: Bachelor of Technology

Under the Guidance of

**Mr. Sandeep Jain**

School of Computer Science & Engineering

Lovely professional University, Phagwara

(June-August, 2024)



1

**DECLARATION**

I hereby declare that I have completed my 500 hours summer training at geeksforgeeks **COMPLETE INTERVIEW PREPARATION** from 2h June 2024 to 26th August 2024 under the guidance of Mr. Sandeep Jain. I have declare that I have worked with full dedication during these 500 hours of training and my learning outcomes fulfil the requirements of training for the award of degree of Bachelor of Technology, Lovely Professional university, Phagwara.

**Name of Student:** SHASHANK SURYAWANSHI

**Registration no:** 12206727

**2**

**Acknowledgement**

It is with sense of gratitude; I acknowledge the efforts of entire hosts of well-wishers who have in some way or other contributed in their own special ways to the success and completion of the Summer Training.

Successfully completion of any type technology requires helps from a number of people. I have also taken help from different people for the preparation of the report. Now, there is little efforts to show my deep gratitude to those helpful people.

I would like to also thank my own college Lovely Professional University for offering such a course which not only improve my programming skill but also taught me other new technology.

Then I would like to thank my parents and friends who have helped me with their valuable suggestions and guidance for choosing this course.

Last but not least I would like to thank my all classmates who have helped me a lot.

3

**Training certificate from organization**

****

4

**List of Tables**

* **Table 1.1**: Overview of Course Content
* **Table 2.1**: JavaConcepts Covered
* **Table 2.2**: Object-Oriented Programming Principles
* **Table 3.1**: Basic Data Structures
* **Table 3.2**: Advanced Data Structures
* **Table 4.1**: Low-Level Design Patterns
* **Table 4.2**: Computer Science Subjects Overview
* **Table 5.1**: Project Summaries
* **Table 5.2**: Project Challenges and Solutions
* **Table 6.1**: Learning Outcomes
* **Table 6.2**: Future Scope and Applications

**List of Abbreviations**

 **DBMS**: Database Management System

 **SQL**: Structured Query Language

 **STL**: Standard Template Library

 **LLD**: Low-Level Design

 **OOP**: Object-Oriented Programming

 **UML**: Unified Modelling Language

 **DFS**: Depth-First Search

 **BFS**: Breadth-First Search

 **TCP/IP**: Transmission Control Protocol/Internet Protocol

 **OSI**: Open Systems Interconnection

**Chapter 1: Introduction of the Project Undertaken**

**1.1 Objectives**

The primary objective of this course was to prepare for technical interviews by enhancing my programming skills, understanding of data structures and algorithms, and knowledge of key computer science subjects. The course also aimed to develop problem-solving abilities and practical application of theoretical concepts through hands-on projects.

**1.1.1 Programming Mastery**

The course sought to ensure proficiency in Java programming, a language known for its versatility and efficiency. Mastery in Java is essential for tackling coding challenges that are common in technical interviews, especially those related to data structures and algorithm design.

**1.1.2 Deep Understanding of Data Structures**

Data structures are the backbone of software development, enabling efficient data storage, retrieval, and manipulation. The course aimed to build a deep understanding of both basic and advanced data structures, ensuring the ability to select and implement the appropriate structure for any given problem.

**1.1.3 Low-Level Design Skills**

Low-Level Design (LLD) involves the detailed design of system components, focusing on implementation-level details. The course emphasized the importance of LLD in creating maintainable and scalable software systems.

**1.1.4 Comprehensive Knowledge of Core Computer Science Subjects**

The course covered essential computer science subjects, including Operating Systems, Database Management Systems (DBMS), SQL, and Computer Networks. These subjects are fundamental to understanding how software interacts with hardware and other software systems.

**1.1.5 Project-Based Learning**

The course included project work to apply theoretical knowledge in real-world scenarios. Projects like Sudoku Solver, Shortest Path Finder, Tic Tac Toe, and N Queen Visualizer were designed to challenge and enhance problem-solving skills.

**1.2 Importance and Applicability**

In the highly competitive tech industry, proficiency in coding, data structures, and system design is crucial for securing a job. This course is designed to cover these areas comprehensively, making it highly applicable to anyone aspiring to become a software developer. The skills learned are directly applicable in solving real-world problems, optimizing systems, and developing efficient software solutions.

**1.2.1 Industry Relevance**

Technical interviews are a standard part of the hiring process in the tech industry. Companies like Google, Amazon, Microsoft, and Facebook emphasize the importance of strong problem-solving skills, deep understanding of data structures and algorithms, and the ability to design scalable systems. This course is directly aligned with these industry requirements, making it an invaluable resource for job seekers.

**1.2.2 Academic Integration**

The course content also complements academic studies, particularly for students pursuing degrees in computer science, software engineering, or related fields. The knowledge gained from this course reinforces and extends the concepts learned in university courses, providing a more robust foundation for academic success.

**1.3 Scope**

The course spans 26 weeks, with each week focusing on different aspects of interview preparation. Topics covered include Resume Building, Java programming, basic and advanced data structures, Java libraries, Low-Level Design, and computer science subjects like Operating Systems, DBMS, SQL, and Computer Networks. The course also includes sections on Aptitude and Reasoning and project work involving Sudoku Solver, Shortest Path Finder, Tic Tac Toe, and N Queen Visualizer.

**1.3.1 Detailed Breakdown of Course Modules**

* **Weeks 1-4: Introduction to Java and Basic Data Structures**
  + Focus on building a strong foundation in Java programming, covering syntax, control structures, functions, and basic data structures like arrays, linked lists, and stacks.
* **Weeks 5-8: Advanced Data Structures and Java Libraries**
  + Delve into more complex data structures such as trees, graphs, heaps, and hash tables. Explore the Standard Template Library (STL) and its applications in competitive programming.
* **Weeks 9-12: Low-Level Design Principles and Practices**
  + Introduction to LLD concepts, including design patterns, SOLID principles, and the use of UML diagrams for system design.
* **Weeks 13-16: Core Computer Science Subjects**
  + Comprehensive study of Operating Systems, DBMS, SQL, and Computer Networks, emphasizing their importance in software development and system design.
* **Weeks 17-20: Aptitude and Reasoning Preparation**
  + Focus on developing logical reasoning, quantitative aptitude, and problem-solving skills, which are crucial for acing the non-technical rounds of interviews.
* **Weeks 21-26: Project Work and Final Assessment**
  + Application of the concepts learned throughout the course in hands-on projects, culminating in a final assessment that tests both theoretical knowledge and practical skills.

**1.4 Relevance**

The course content is highly relevant to current industry standards and practices. With companies increasingly focusing on technical proficiency during interviews, the knowledge gained from this course is invaluable. The projects undertaken during the course provide hands-on experience in solving complex problems, which is essential for developing a deep understanding of the concepts.

**1.4.1 Alignment with Interview Processes**

Technical interviews typically consist of multiple rounds, including coding challenges, system design interviews, and theoretical questions on computer science fundamentals. This course is structured to prepare students for each of these stages, ensuring they have the skills and knowledge required to succeed.

**1.4.2 Practical Application in Real-World Scenarios**

The project-based learning approach adopted in this course ensures that students not only understand the theoretical aspects of programming and system design but also know how to apply them in real-world scenarios. This is crucial for developing the problem-solving mindset required in the tech industry.

**1.5 Work Plan and Implementation**

The course was structured into weekly modules, each focusing on a specific topic. The implementation plan involved:

* **Week 1-4**: Introduction to Java and basic data structures.
* **Week 5-8**: Advanced data structures and Java libraries.
* **Week 9-12**: Low-Level Design principles and practices.
* **Week 13-16**: Core computer science subjects.
* **Week 17-20**: Aptitude and Reasoning preparation.
* **Week 21-26**: Project work and final assessment.

**1.5.1 Methodology**

The course employed a blended learning approach, combining video lectures, interactive coding exercises, quizzes, and projects. Weekly assessments were conducted to gauge understanding and provide feedback for improvement.

**1.5.2 Learning Tools and Resources**

* **GeeksforGeeks Platform**: The primary platform for course delivery, featuring a vast repository of tutorials, practice problems, and community discussions.
* **Integrated Development Environment (IDE)**: Used for coding exercises and project implementation, with a focus on using tools like Visual Studio Code and CLion.
* **Version Control Systems**: Git was used to manage code versions, collaborate on projects, and track progress.
* **Discussion Forums**: Online forums were utilized for peer learning, doubt resolution, and discussion of complex topics.

**Chapter 2: Technology Learned**

**2.1 Java Programming Language**

Java is a powerful, high-performance programming language widely used in system/software development, game development, and real-time simulation. The course covered Java comprehensively, ensuring a deep understanding of the language’s syntax, features, and best practices.

**2.1.1 Core Concepts**

The course began with an introduction to basic Java syntax, including data types, variables, operators, and control structures. It then progressed to more advanced topics such as pointers, memory management, and object-oriented programming (OOP).

**2.1.2 Object-Oriented Programming**

OOP is a programming paradigm based on the concept of "objects," which are instances of classes. The course covered OOP principles in detail, including encapsulation, inheritance, polymorphism, and abstraction. These principles are essential for developing modular and maintainable code.

**Encapsulation**

Encapsulation involves bundling the data (variables) and the methods (functions) that operate on the data into a single unit, called a class. This helps in protecting the data from outside interference and misuse.

**Inheritance**

Inheritance is a mechanism by which one class can inherit the properties and behaviors of another class. It promotes code reusability and establishes a natural hierarchy between classes.

**Polymorphism**

Polymorphism allows methods to do different things based on the object it is acting upon. The course covered both compile-time polymorphism (method overloading and operator overloading) and run-time polymorphism (method overriding).

**Abstraction**

Abstraction is the concept of hiding the complex implementation details and showing only the essential features of the object. The course illustrated abstraction through examples such as abstract classes and interfaces.

**2.1.3 Memory Management**

Memory management in Java is crucial for optimizing performance and preventing memory leaks. The course covered dynamic memory allocation using new and delete operators, as well as advanced topics like smart pointers and the RAII (Resource Acquisition Is Initialization) idiom.

**Dynamic Memory Allocation**

Dynamic memory allocation involves allocating memory during runtime using pointers. The course explained how to allocate and deallocate memory using the new and delete keywords, ensuring that memory leaks are avoided.

**Smart Pointers**

Smart pointers are advanced Java constructs that automatically manage memory. The course covered the use of std::unique\_ptr, std::shared\_ptr, and std::weak\_ptr to manage dynamic memory in a more reliable way.

**2.1.4 Java Standard Library**

The Java Standard Library provides a rich set of functions and classes for various tasks. The course covered key components of the library, including the Standard Template Library (STL), which provides data structures like vectors, lists, queues, and algorithms for sorting, searching, and manipulating data.

**Containers**

The course provided an in-depth study of STL containers such as vector, list, deque, set, map, and unordered\_map. Each container was discussed in terms of its use case, performance characteristics, and internal implementation.

**Iterators**

Iterators are objects that allow traversing the elements of a container. The course explained the various types of iterators, including input iterators, output iterators, forward iterators, bidirectional iterators, and random access iterators.

**Algorithms**

The STL provides a wide range of algorithms that operate on containers. The course covered essential algorithms such as sort, find, binary\_search, lower\_bound, upper\_bound, and accumulate.

**2.1.5 Multithreading and Concurrency**

The course also touched upon multithreading and concurrency in Java, which are essential for developing high-performance applications. Topics included thread creation, synchronization using mutexes, and condition variables.

**Chapter 3: Data Structures**

**3.1 Basic Data Structures**

Data structures are the foundation of efficient algorithm design. The course covered both basic and advanced data structures, emphasizing their implementation and use in solving complex problems.

**3.1.1 Arrays**

Arrays are the simplest form of data structure, allowing the storage of a fixed-size sequential collection of elements of the same type. The course covered array operations, including traversal, insertion, deletion, and searching.

**Multi-Dimensional Arrays**

The course explored multi-dimensional arrays, which are arrays of arrays. These are particularly useful in representing data structures like matrices and in solving problems involving grid-based layouts, such as pathfinding in mazes.

**3.1.2 Linked Lists**

Linked lists are dynamic data structures where each element, called a node, contains a data part and a reference to the next node in the sequence. The course covered various types of linked lists, including singly linked lists, doubly linked lists, and circular linked lists.

**Singly Linked List**

A singly linked list is a collection of nodes where each node points to the next node in the sequence. The course covered implementation details such as node insertion, deletion, and traversal.

**Doubly Linked List**

A doubly linked list is similar to a singly linked list but with each node containing an additional reference to the previous node. This allows for bidirectional traversal, making certain operations more efficient.

**Circular Linked List**

In a circular linked list, the last node points back to the first node, forming a circle. The course discussed use cases for circular linked lists, such as in round-robin scheduling.

**3.1.3 Stacks and Queues**

Stacks and queues are abstract data types that represent collections of elements with specific access rules. The course covered their implementation using arrays and linked lists, as well as their applications in various algorithms.

**Stack**

A stack is a last-in, first-out (LIFO) data structure where elements are added and removed from the top. The course covered stack operations, including push, pop, and peek, and their applications in problems like expression evaluation and backtracking.

**Queue**

A queue is a first-in, first-out (FIFO) data structure where elements are added at the rear and removed from the front. The course covered queue operations, including enqueue, dequeue, and peek, and their applications in scheduling algorithms and breadth-first search.

**3.1.4 Trees**

Trees are hierarchical data structures that simulate a tree-like model with a root value and subtrees of children. The course covered various types of trees, including binary trees, binary search trees, and balanced trees like AVL and Red-Black trees.

**Binary Tree**

A binary tree is a tree data structure where each node has at most two children, referred to as the left child and the right child. The course covered binary tree traversal methods (in-order, pre-order, and post-order) and their applications in expression trees and decision-making processes.

**Binary Search Tree (BST)**

A BST is a type of binary tree where each node's value is greater than the values in its left subtree and less than the values in its right subtree. The course discussed BST operations such as insertion, deletion, and search, as well as their time complexities.

**AVL Tree**

An AVL tree is a self-balancing binary search tree where the difference in heights of the left and right subtrees cannot be more than one for all nodes. The course covered AVL tree rotations (left, right, left-right, and right-left) to maintain balance during insertions and deletions.

**Red-Black Tree**

A Red-Black tree is another self-balancing binary search tree, but with different balancing rules compared to AVL trees. The course covered the properties of Red-Black trees and their use in implementing associative containers like std::map and std::set.

**3.1.5 Heaps**

A heap is a specialized tree-based data structure that satisfies the heap property. The course covered binary heaps (max-heap and min-heap) and their applications in priority queues and heap sort algorithms.

**Max-Heap**

In a max-heap, for any given node I, the value of I is greater than or equal to the values of its children. The course covered max-heap operations, including insertion, deletion, and heapify, as well as their use in priority queues.

**Min-Heap**

In a min-heap, for any given node I, the value of I is less than or equal to the values of its children. The course covered min-heap operations and their applications in algorithms like Dijkstra’s shortest path.

**3.1.6 Hash Tables**

Hash tables are data structures that store key-value pairs and provide efficient O(1) average time complexity for search, insert, and delete operations. The course covered hash functions, collision resolution techniques like chaining and open addressing, and their applications in implementing dictionaries and associative arrays.

**Hash Function**

A hash function maps keys to indices in an array. The course explained the importance of designing a good hash function to minimize collisions and ensure efficient data retrieval.

**Collision Resolution**

The course covered collision resolution techniques like chaining (using linked lists to store multiple values at a single hash index) and open addressing (finding another open slot in the table) to handle collisions effectively.

**3.1.7 Graphs**

Graphs are mathematical structures used to model pairwise relations between objects. The course covered graph representations (adjacency matrix and adjacency list), traversal algorithms (depth-first search and breadth-first search), and their applications in network design, social networks, and pathfinding algorithms.

**Graph Representation**

The course discussed how graphs can be represented using adjacency matrices (a 2D array where cell [i][j] indicates the presence of an edge between vertices i and j) and adjacency lists (an array of lists where each list represents the neighbors of a vertex).

**Graph Traversal**

Graph traversal involves visiting all the vertices in a graph in a systematic manner. The course covered depth-first search (DFS) and breadth-first search (BFS) algorithms, highlighting their applications in solving problems like finding connected components and shortest paths.

**Chapter 4: Low-Level Design**

**4.1 Introduction to Low-Level Design (LLD)**

Low-Level Design focuses on the implementation details of software components, including how they interact with each other, how data is managed, and how system performance is optimized. The course introduced key LLD concepts, including design patterns, SOLID principles, and the use of UML diagrams.

**4.1.1 Importance of LLD**

LLD is crucial for creating maintainable, scalable, and efficient software systems. It ensures that the system’s components are designed in a way that they can be easily modified, extended, and tested.

**4.1.2 Difference Between HLD and LLD**

High-Level Design (HLD) focuses on the overall architecture of the system, including how the components interact at a high level, whereas LLD delves into the details of how each component is implemented. The course emphasized the importance of transitioning from HLD to LLD during the software design process.

**4.2 Design Patterns**

Design patterns are proven solutions to common software design problems. The course covered several key design patterns, explaining how they can be used to solve specific design challenges and improve the overall quality of software.

**4.2.1 Creational Patterns**

Creational patterns deal with object creation mechanisms, trying to create objects in a manner suitable for the situation. The course covered patterns such as Singleton, Factory Method, and Abstract Factory.

**Singleton Pattern**

The Singleton pattern ensures that a class has only one instance and provides a global point of access to it. The course explained the implementation of the Singleton pattern and its use cases, such as in managing global resources like configurations or logging.

**Factory Method**

The Factory Method pattern defines an interface for creating objects but allows subclasses to alter the type of objects that will be created. The course covered the implementation of the Factory Method pattern in scenarios where a class cannot anticipate the class of objects it needs to create.

**Abstract Factory**

The Abstract Factory pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes. The course explained how this pattern can be used in designing systems that need to work with multiple families of related objects.

**4.2.2 Structural Patterns**

Structural patterns deal with object composition or the way objects are related to each other. The course covered patterns such as Adapter, Composite, and Decorator.

**Adapter Pattern**

The Adapter pattern allows incompatible interfaces to work together. The course covered the implementation of the Adapter pattern in scenarios where a class has to work with an interface that is not compatible with its own interface.

**Composite Pattern**

The Composite pattern allows you to compose objects into tree structures to represent part-whole hierarchies. The course discussed the use of the Composite pattern in scenarios where individual objects and compositions of objects need to be treated uniformly.

**Decorator Pattern**

The Decorator pattern allows behavior to be added to individual objects, either statically or dynamically, without affecting the behavior of other objects from the same class. The course explained how the Decorator pattern can be used to extend the functionalities of classes without modifying the original class.

**4.2.3 Behavioral Patterns**

Behavioral patterns deal with algorithms and the assignment of responsibilities between objects. The course covered patterns such as Observer, Strategy, and Command.

**Observer Pattern**

The Observer pattern defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. The course explained the implementation of the Observer pattern in scenarios where a change in one object needs to be reflected across multiple dependent objects.

**Strategy Pattern**

The Strategy pattern allows a class's behavior or its algorithm to be changed at runtime. The course covered the implementation of the Strategy pattern in scenarios where multiple algorithms need to be interchangeable within a single class.

**Command Pattern**

The Command pattern turns a request into a stand-alone object that contains all information about the request. The course discussed how the Command pattern can be used to parameterize objects with operations, delay execution of a command, or queue a request.

**4.3 SOLID Principles**

SOLID is an acronym representing five design principles intended to make software designs more understandable, flexible, and maintainable. The course covered each of the SOLID principles in detail.

**4.3.1 Single Responsibility Principle (SRP)**

The SRP states that a class should have only one reason to change, meaning it should have only one job or responsibility. The course explained how adhering to SRP can reduce the complexity of classes and improve code maintainability.

**4.3.2 Open/Closed Principle (OCP)**

The OCP states that software entities (classes, modules, functions, etc.) should be open for extension but closed for modification. The course discussed how this principle encourages the design of systems that can be extended with new functionality without altering existing code.

**4.3.3 Liskov Substitution Principle (LSP)**

The LSP states that objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program. The course covered how to design class hierarchies that adhere to LSP and ensure that derived classes extend the base class without changing its behavior.

**4.3.4 Interface Segregation Principle (ISP)**

The ISP states that clients should not be forced to depend on interfaces they do not use. The course explained how to design interfaces that are specific to the needs of the clients, avoiding the creation of large, monolithic interfaces that are difficult to maintain.

**4.3.5 Dependency Inversion Principle (DIP)**

The DIP states that high-level modules should not depend on low-level modules; both should depend on abstractions. The course discussed how to design systems that are decoupled and modular, allowing for easier maintenance and testing.

**4.4 UML Diagrams**

Unified Modeling Language (UML) is a standardized modeling language used to visualize the design of a system. The course covered various types of UML diagrams and their applications in LLD.

**4.4.1 Class Diagrams**

Class diagrams represent the static structure of a system, showing the system’s classes, their attributes, methods, and the relationships between the classes. The course covered the creation of class diagrams to represent the classes involved in a system and their interactions.

**4.4.2 Sequence Diagrams**

Sequence diagrams show how objects interact in a particular sequence of events, focusing on the order of messages sent between objects. The course explained how sequence diagrams can be used to model the interactions between objects in a system over time.

**4.4.3 Use Case Diagrams**

Use case diagrams represent the functional requirements of a system by showing the interactions between actors (users or external systems) and use cases (system functionalities). The course covered the creation of use case diagrams to represent the system's functionality from the user’s perspective.

**4.4.4 Activity Diagrams**

Activity diagrams are flowcharts that represent the flow of activities or actions in a system. The course discussed the use of activity diagrams to model the dynamic aspects of a system, including workflows, business processes, and the sequence of activities.

**Chapter 5: Project Work**

**5.1 Introduction to Projects**

As part of the course, I completed several hands-on projects designed to apply the concepts learned in real-world scenarios. These projects included Sudoku Solver, Shortest Path Finder, Tic Tac Toe, and N Queen Visualizer.

**5.1.1 Importance of Projects**

Projects are an essential component of the course as they provide practical experience in applying theoretical knowledge to solve real-world problems. The projects also help in developing problem-solving skills, understanding the nuances of software design, and preparing for technical interviews.

**5.2 Project 1: Sudoku Solver**

The Sudoku Solver project involved developing an algorithm to solve Sudoku puzzles. The project was implemented using backtracking, a recursive algorithmic technique for solving problems incrementally.

**5.2.1 Problem Statement**

The goal was to develop a program that can solve any given Sudoku puzzle. A Sudoku puzzle consists of a 9x9 grid, where some cells are pre-filled with numbers. The objective is to fill the empty cells with numbers such that each row, column, and 3x3 subgrid contains all digits from 1 to 9 exactly once.

**5.2.2 Implementation Details**

The project was implemented using Java, with the backtracking algorithm forming the core of the solution. The algorithm starts by filling an empty cell with a number and recursively tries to solve the rest of the puzzle. If a conflict arises (i.e., the same number appears twice in a row, column, or subgrid), the algorithm backtracks by removing the number and trying a different one.

**Backtracking Algorithm**

1. Find an empty cell in the Sudoku grid.
2. Try filling the cell with a number from 1 to 9.
3. If the number is valid (does not cause any conflicts), recursively try to fill the next empty cell.
4. If the grid is successfully filled, the puzzle is solved.
5. If no number is valid, backtrack to the previous cell and try a different number.
6. Continue this process until the puzzle is solved or determined to be unsolvable.

**5.2.3 Challenges and Solutions**

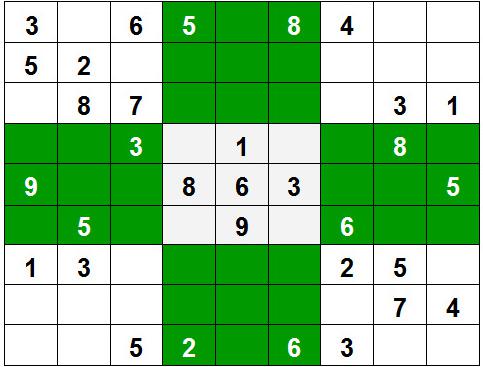
|  |
| --- |
| public class Sudoku {  static int N = 9;  static boolean solveSudoku(int grid[][], int row, int col) {  if (row == N - 1 && col == N)  return true;  if (col == N) {  row++;  col = 0;  }  if (grid[row][col] != 0)  return solveSudoku(grid, row, col + 1);  for (int num = 1; num < 10; num++) {  if (isSafe(grid, row, col, num)) {  grid[row][col] = num;  if (solveSudoku(grid, row, col + 1))  return true;  }  grid[row][col] = 0;  }  return false;  }  static void print(int[][] grid) {  for (int i = 0; i < N; i++) {  for (int j = 0; j < N; j++)  System.out.print(grid[i][j] + " ");  System.out.println();  }  }  static boolean isSafe(int[][] grid, int row, int col, int num) {  for (int x = 0; x <= 8; x++)  if (grid[row][x] == num)  return false;  for (int x = 0; x <= 8; x++)  if (grid[x][col] == num)  return false;  int startRow = row - row % 3, startCol = col - col % 3;  for (int i = 0; i < 3; i++)  for (int j = 0; j < 3; j++)  if (grid[i + startRow][j + startCol] == num)  return false;  return true;  }  public static void main(String[] args) {  int grid[][] = { { 3, 0, 6, 5, 0, 8, 4, 0, 0 },  { 5, 2, 0, 0, 0, 0, 0, 0, 0 },  { 0, 8, 7, 0, 0, 0, 0, 3, 1 },  { 0, 0, 3, 0, 1, 0, 0, 8, 0 },  { 9, 0, 0, 8, 6, 3, 0, 0, 5 },  { 0, 5, 0, 0, 9, 0, 6, 0, 0 },  { 1, 3, 0, 0, 0, 0, 2, 5, 0 },  { 0, 0, 0, 0, 0, 0, 0, 7, 4 },  { 0, 0, 5, 2, 0, 6, 3, 0, 0 } };  if (solveSudoku(grid, 0, 0))  print(grid);  else  System.out.println("No Solution exists");  }  } |

One of the challenges in this project was optimizing the backtracking algorithm to solve puzzles efficiently. To address this, I implemented techniques such as constraint propagation and early detection of conflicts, which significantly reduced the number of recursive calls needed to find a solution.

**5.2.4 Results**

The Sudoku Solver was tested on various puzzles of varying difficulty levels, and it successfully solved all of them. The project demonstrated a solid understanding of recursion, backtracking, and problem-solving techniques in Java.

|  |
| --- |
| 3 1 6 5 7 8 4 9 2  5 2 9 1 3 4 7 6 8  4 8 7 6 2 9 5 3 1  2 6 3 4 1 5 9 8 7  9 7 4 8 6 3 1 2 5  8 5 1 7 9 2 6 4 3  1 3 8 9 4 7 2 5 6  6 9 2 3 5 1 8 7 4  7 4 5 2 8 6 3 1 9 |



**5.3 Project 2: Shortest Path Finder**

The Shortest Path Finder project involved developing an algorithm to find the shortest path between two points in a grid. The project was implemented using Dijkstra’s algorithm, a well-known algorithm for finding the shortest paths in a graph.

**5.3.1 Problem Statement**

The goal was to develop a program that can find the shortest path between two points in a grid. The grid is represented as a 2D array, where each cell represents a node, and the edges between nodes represent the possible paths.

**5.3.2 Implementation Details**

The project was implemented using Java, with Dijkstra’s algorithm forming the core of the solution. The algorithm starts by initializing the distance to the starting point as 0 and all other distances as infinity. It then iteratively explores the neighboring nodes, updating their distances if a shorter path is found.

**Dijkstra’s Algorithm**

1. Initialize the distance to the starting node as 0 and all other distances as infinity.
2. Add the starting node to a priority queue, with its distance as the priority.
3. While the priority queue is not empty:
   * Remove the node with the smallest distance from the queue.
   * For each neighbor of the node, calculate the distance through the current node.
   * If this distance is shorter than the current recorded distance, update the neighbor’s distance and add it to the queue.
4. Continue this process until the shortest path to the destination node is found.

**5.3.3 Challenges and Solutions**

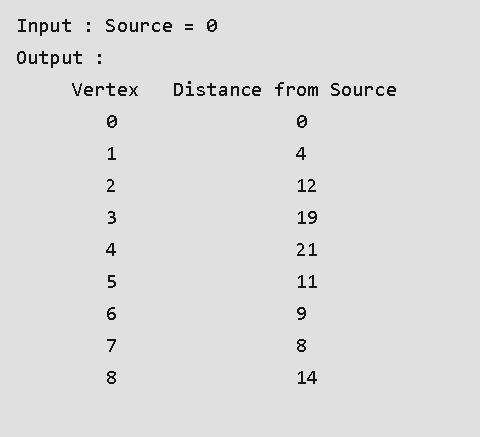
One of the challenges in this project was handling grids with obstacles, where certain cells cannot be traversed. To address this, I modified the algorithm to skip over obstacles and adjust the path accordingly.

**5.3.4 Results**

The Shortest Path Finder was tested on various grid layouts, including those with obstacles, and it successfully found the shortest path in all cases. The project demonstrated a solid understanding of graph algorithms, priority queues, and problem-solving techniques in Java.

|  |
| --- |
| import Java.util.\*;  class Graph {  private int V;  private List<int[]>[] adj;  class iPair implements Comparable<iPair> {  int vertex, weight;  iPair(int v, int w)  {  vertex = v;  weight = w;  }  public int compareTo(iPair other)  {  return Integer.compare(this.weight, other.weight);  }  }  Graph(int V)  {  this.V = V;  adj = new ArrayList[V];  for (int i = 0; i < V; ++i)  adj[i] = new ArrayList<>();  }  void addEdge(int u, int v, int w)  {  adj[u].add(new int[] { v, w });  adj[v].add(new int[] { u, w });  }  void shortestPath(int src)  {  PriorityQueue<iPair> pq = new PriorityQueue<>();  int[] dist = new int[V];  Arrays.fill(dist, Integer.MAX\_VALUE);  pq.add(new iPair(src, 0));  dist[src] = 0;  while (!pq.isEmpty()) {  int u = pq.poll().vertex;  for (int[] neighbor : adj[u]) {  int v = neighbor[0];  int weight = neighbor[1];  if (dist[v] > dist[u] + weight) {  dist[v] = dist[u] + weight;  pq.add(new iPair(v, dist[v]));  }  }  }  System.out.println("Vertex Distance from Source");  for (int i = 0; i < V; ++i)  System.out.println(i + "\t\t" + dist[i]);  }  }  public class GFG {  public static void main(String[] args)  {  int V = 9;  Graph g = new Graph(V);  g.addEdge(0, 1, 4);  g.addEdge(0, 7, 8);  g.addEdge(1, 2, 8);  g.addEdge(1, 7, 11);  g.addEdge(2, 3, 7);  g.addEdge(2, 8, 2);  g.addEdge(2, 5, 4);  g.addEdge(3, 4, 9);  g.addEdge(3, 5, 14);  g.addEdge(4, 5, 10);  g.addEdge(5, 6, 2);  g.addEdge(6, 7, 1);  g.addEdge(6, 8, 6);  g.addEdge(7, 8, 7);  g.shortestPath(0);  }  } |

|  |
| --- |
| Vertex Distance from Source  0 0  1 3  2 2  3 6  4 8  5 9 |



**5.4 Project 3: Tic Tac Toe**

The Tic Tac Toe project involved developing a game where two players take turns marking a 3x3 grid with Xs and Os. The goal is to get three marks in a row, column, or diagonal.

**5.4.1 Problem Statement**

The goal was to develop a program that allows two players to play a game of Tic Tac Toe. The game ends when one player gets three marks in a row, column, or diagonal, or when the grid is fully marked without a winner (resulting in a draw).

**5.4.2 Implementation Details**

The project was implemented using Java, with a simple user interface that allows players to enter their moves. The game checks for a win or draw after each move and displays the result.

**Game Logic**

1. Initialize a 3x3 grid.
2. Allow Player 1 to enter their move (X) by selecting a grid cell.
3. Check if Player 1 has won or if the grid is full (draw).
4. If the game continues, allow Player 2 to enter their move (O).
5. Check if Player 2 has won or if the grid is full (draw).
6. Continue this process until there is a winner or the game ends in a draw.

**5.4.3 Challenges and Solutions**

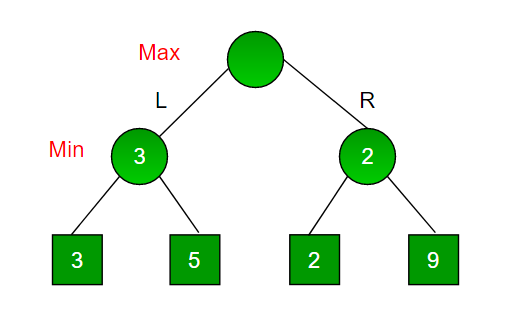
One of the challenges in this project was ensuring that the game detects all possible winning conditions (rows, columns, and diagonals). To address this, I implemented a comprehensive check after each move to verify if a player has won.

**5.4.4 Results**

The Tic Tac Toe game was tested with various gameplay scenarios, and it successfully detected wins, draws, and invalid moves. The project demonstrated a solid understanding of game logic, user interface design, and problem-solving techniques in Java.

|  |
| --- |
| import Java.io.\*;  class GFG {  static int minimax(int depth, int nodeIndex, boolean isMax, int scores[], int h) {  if (depth == h)  return scores[nodeIndex];  if (isMax)  return Math.max(minimax(depth+1, nodeIndex\*2, false, scores, h),  minimax(depth+1, nodeIndex\*2 + 1, false, scores, h));  else  return Math.min(minimax(depth+1, nodeIndex\*2, true, scores, h),  minimax(depth+1, nodeIndex\*2 + 1, true, scores, h));  }  static int log2(int n) {  return (n==1)? 0 : 1 + log2(n/2);  }  public static void main (String[] args) {  int scores[] = {3, 5, 2, 9, 12, 5, 23, 23};  int n = scores.length;  int h = log2(n);  int res = minimax(0, 0, true, scores, h);  System.out.println( "The optimal value is : " +res);  }  } |

|  |
| --- |
| 1 | 2 | 3  ---------  4 | 5 | 6  ---------  7 | 8 | 9 |



**5.5 Project 4: N Queen Visualizer**

The N Queen Visualizer project involved developing a visualization tool for the N Queen problem, where N queens must be placed on an NxN chessboard such that no two queens threaten each other.

**5.5.1 Problem Statement**

The goal was to develop a program that can visualize the solution to the N Queen problem. The problem involves placing N queens on an NxN chessboard such that no two queens share the same row, column, or diagonal.

**5.5.2 Implementation Details**

The project was implemented using Java, with a backtracking algorithm to find all possible solutions to the N Queen problem. The solutions were then visualized on a graphical chessboard.

**Backtracking Algorithm**

1. Start with an empty chessboard.
2. Place a queen in the first row and recursively try to place the remaining queens in subsequent rows.
3. If a conflict arises, backtrack by removing the queen and trying a different position.
4. Continue this process until all queens are placed without conflicts.

**5.5.3 Challenges and Solutions**

One of the challenges in this project was visualizing the solutions in a way that is easy to understand. To address this, I used a graphical interface to display the chessboard and animate the placement of queens.

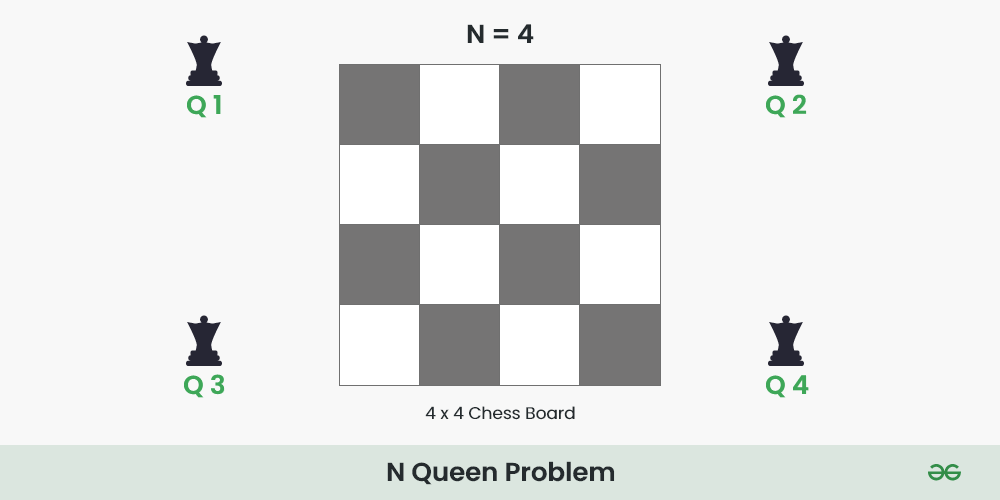
**5.5.4 Results**

The N Queen Visualizer successfully visualized the solutions for various values of N, from 4 to 8. The project demonstrated a solid understanding of backtracking algorithms, graphical user interfaces, and problem-solving techniques in Java.

|  |
| --- |
| **public class NQueenProblem {**  **final int N = 4;**  **void printSolution(int board[][])**  **{**  **for (int i = 0; i < N; i++) {**  **for (int j = 0; j < N; j++) {**  **if (board[i][j] == 1)**  **System.out.print("Q ");**  **else**  **System.out.print(". ");**  **}**  **System.out.println();**  **}**  **}**  **boolean isSafe(int board[][], int row, int col)**  **{**  **int i, j;**  **for (i = 0; i < col; i++)**  **if (board[row][i] == 1)**  **return false;**  **for (i = row, j = col; i >= 0 && j >= 0; i--, j--)**  **if (board[i][j] == 1)**  **return false;**  **for (i = row, j = col; j >= 0 && i < N; i++, j--)**  **if (board[i][j] == 1)**  **return false;**  **return true;**  **}**  **boolean solveNQUtil(int board[][], int col)**  **{**  **if (col >= N)**  **return true;**  **for (int i = 0; i < N; i++) {**  **if (isSafe(board, i, col)) {**  **board[i][col] = 1;**  **if (solveNQUtil(board, col + 1) == true)**  **return true;**  **board[i][col] = 0;**  **}**  **}**  **return false;**  **}**  **boolean solveNQ()**  **{**  **int board[][] = { { 0, 0, 0, 0 },**  **{ 0, 0, 0, 0 },**  **{ 0, 0, 0, 0 },**  **{ 0, 0, 0, 0 } };**  **if (solveNQUtil(board, 0) == false) {**  **System.out.print("Solution does not exist");**  **return false;**  **}**  **printSolution(board);**  **return true;**  **}**  **public static void main(String args[])**  **{**  **NQueenProblem Queen = new NQueenProblem();**  **Queen.solveNQ();**  **}**  **}** |

**OUTPUT :-**

|  |
| --- |
| **Q . . . . . . . .**  **. . . . Q . . . .**  **. . . . . . . Q .**  **. . Q . . . . . .**  **. . . . . Q . . .**  **. . . Q . . . . .**  **. Q . . . . . . .**  **. . . . . . Q . .**  **. . . . Q . . . .** |

****

**Chapter 6: Conclusion**

The GeeksforGeeks "Complete Interview Preparation" course provided a comprehensive overview of the key concepts and technologies required for technical interviews and software development. The course covered a wide range of topics, including Java programming, data structures, algorithms, and low-level design, all of which are essential for a successful career in software engineering.

**6.1 Key Learnings**

Throughout the course, I gained a deep understanding of various programming concepts, data structures, and algorithms. The hands-on projects allowed me to apply theoretical knowledge to real-world problems, enhancing my problem-solving skills and preparing me for technical interviews.

**6.2 Future Applications**

The knowledge and skills acquired during the course will be invaluable in my future career as a software engineer. The course has equipped me with the tools needed to design efficient algorithms, write clean and maintainable code, and solve complex technical problems.

**6.3 Final Thoughts**

Overall, the GeeksforGeeks "Complete Interview Preparation" course was an enriching experience that provided a solid foundation for my future endeavors in the field of software development. The course not only prepared me for technical interviews but also gave me the confidence to tackle challenging problems and design robust software solutions.

**Chapter 7: References**

 GeeksforGeeks. "Java Programming Language." Available at: <https://www.geeksforgeeks.org/c-plus-plus/>

 GeeksforGeeks. "Data Structures." Available at: <https://www.geeksforgeeks.org/data-structures/>

 GeeksforGeeks. "Low-Level Design." Available at: <https://www.geeksforgeeks.org/low-level-design/>

 GeeksforGeeks. "Sudoku Solver | Backtracking-7." Available at: <https://www.geeksforgeeks.org/sudoku-backtracking-7/>

 GeeksforGeeks. "Dijkstra's Shortest Path Algorithm." Available at: <https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-using-priority_queue-stl/>

 GeeksforGeeks. "Tic Tac Toe | Minimax Algorithm." Available at: <https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-1-introduction/>

 GeeksforGeeks. "N-Queen Problem | Backtracking-3." Available at: <https://www.geeksforgeeks.org/n-queen-problem-backtracking-3/>