Programming skill Preparation

Basic Algorithms

1. Linear Search: An algorithm that searches for a given element in a list or array, one element at a time, until the desired element is found.
2. Binary Search: A more efficient search algorithm that works by repeatedly dividing the search interval in half.
3. Bubble Sort: A simple sorting algorithm that repeatedly steps through the list, compares adjacent elements and swaps them if they are in the wrong order.
4. Selection Sort: Another simple sorting algorithm that sorts an array by repeatedly finding the minimum element from the unsorted part of the array and putting it at the beginning.
5. Insertion Sort: A sorting algorithm that builds the final sorted array one element at a time by inserting each element in its proper place in the sorted array.
6. Merge Sort: A divide-and-conquer sorting algorithm that sorts an array by dividing it into two halves, sorting the two halves independently, and then merging the sorted halves.
7. Quick Sort: Another divide-and-conquer sorting algorithm that works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot.
8. Depth-First Search (DFS): A graph traversal algorithm that explores as far as possible along each branch before backtracking.
9. Breadth-First Search (BFS): A graph traversal algorithm that visits all the vertices of a graph in breadth-first order, i.e., it visits all the vertices at distance k from the source vertex before visiting the vertices at distance k+1.
10. Dijkstra's Algorithm: A shortest path algorithm that finds the shortest path between a source node and all other nodes in a weighted graph. It uses a priority queue to keep track of the minimum distance from the source to each node.

Basic Data Structures:

1. Array: A collection of elements of the same data type, stored in contiguous memory locations and accessed using an index.
2. Linked List: A data structure consisting of a sequence of nodes, where each node contains data and a pointer to the next node in the sequence.
3. Stack: A data structure that follows the Last In First Out (LIFO) principle, where the last element added to the stack is the first one to be removed.
4. Queue: A data structure that follows the First In First Out (FIFO) principle, where the first element added to the queue is the first one to be removed.
5. Binary Tree: A tree data structure in which each node has at most two children, referred to as the left child and the right child.
6. Hash Table: A data structure that maps keys to values using a hash function, which computes an index in an array where the corresponding value is stored.
7. Heap: A binary tree-based data structure that satisfies the heap property, which states that the parent node is always greater than or equal to its child nodes (max heap) or less than or equal to its child nodes (min heap).
8. Graph: A data structure consisting of a set of vertices or nodes, connected by edges or arcs.

9. Trie: A tree-based data structure that stores a set of strings, where each node represents a prefix or a complete string.
10. Set: A collection of unique elements, where each element is stored only once.

Commonly Used Design Patterns

1. Singleton pattern: Ensures that only one instance of a class is created, and provides a global point of access to that instance.
2. Factory pattern: Provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.
3. Adapter pattern: Allows classes with incompatible interfaces to work together by wrapping the interface of one class with a compatible interface.
4. Decorator pattern: Allows behavior to be added to an individual object, either statically or dynamically, without affecting the behavior of other objects from the same class.
5. Observer pattern: Defines a one-to-many dependency between objects, so that when one object changes state, all of its dependents are notified and updated automatically.
6. Strategy pattern: Defines a family of algorithms, encapsulates each one, and makes them interchangeable. This lets the algorithm vary independently from clients that use it.
7. Template method pattern: Defines the skeleton of an algorithm in a superclass, but lets subclasses override specific steps of the algorithm without changing its structure.
8. Command pattern: Encapsulates a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
9. Facade pattern: Provides a simplified interface to a complex system of classes, interfaces, or libraries.
10. Iterator pattern: Provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

General Guidelines for Programming Interview Questions

1. Understand the problem: Make sure you fully understand the problem you are being asked to solve. Ask clarifying questions if needed.
2. Consider different approaches: Think about different ways to solve the problem, and weigh the pros and cons of each approach.
3. Choose an approach: Once you have considered different approaches, choose one and explain why you have chosen it.
4. Write clear code: Write clean and readable code, and use appropriate variable names and comments.
5. Test your code: Test your code thoroughly to make sure it works correctly, and consider edge cases and error handling.
6. Optimize if necessary: If your solution is not efficient enough, think about ways to optimize it.
7. Communicate effectively: Explain your thought process and solution clearly, and be open to feedback or questions from the interviewer.

Clean Code Guidelines:

8. Keep it simple: Write code that is easy to read, understand, and maintain. Avoid unnecessary complexity.

9. Use meaningful names: Use descriptive names for variables, functions, and classes. Make your code self-explanatory.
10. Write small functions: Write functions that do one thing and do it well. Keep them small and focused.
11. Follow a consistent style: Follow a consistent coding style throughout your code. Use the same naming conventions, indentation, and formatting.
12. Write readable code: Use white space, comments, and indentation to make your code more readable.
13. Avoid repetition: Don't repeat yourself. Use functions, loops, and other constructs to avoid writing the same code multiple times.
14. Handle errors gracefully: Handle errors and exceptions gracefully. Use try-catch blocks and error handling techniques to avoid crashes and bugs.
15. Write tests: Write tests for your code to ensure it works as expected. You can use test-driven development (TDD) to guide your coding process.
16. Refactor as necessary: Refactor your code as necessary to improve its readability, maintainability, and performance.

Exercises:

Solve these exercises in the Java/Python and upload the solution to your GitHub repo

1. Arrays and Strings:
   - Write a program to reverse a string in place.
   - Write a program to find the maximum and minimum elements in an array.
   - Write a program to remove duplicates from a sorted array.
2. Linked Lists:
   - Write a program to reverse a linked list.
   - Write a program to find the middle element of a linked list.
   - Write a program to detect if a linked list has a cycle.
3. Stacks and Queues:
   - Implement a stack using two queues.
   - Implement a queue using two stacks.
   - Write a program to check if a given string of parentheses is balanced.
4. Trees and Graphs:
   - Write a program to find the lowest common ancestor of two nodes in a binary tree.
   - Write a program to find the shortest path between two nodes in a graph.
   - Implement a binary search tree and write functions to insert, delete and search for elements.
5. Sorting Algorithms:
   - Implement the bubble sort algorithm.
   - Implement the merge sort algorithm.
   - Implement the quicksort algorithm.
6. Searching Algorithms:
   - Implement the linear search algorithm.
   - Implement the binary search algorithm.
7. Recursion:
   - Write a program to calculate the factorial of a number using recursion.

- Write a program to generate all permutations of a string using recursion.

8. Hash Tables:
   - Implement a hash table and write functions to insert, delete, and search for elements.
9. Bit Manipulation:
   - Write a program to check if a given number is even or odd using bit manipulation.
   - Write a program to find the number of set bits in a given integer using bit manipulation.
10. Big O Notation and Time Complexity Analysis:
    - Analyze the time complexity of the sorting algorithms you implemented earlier.
    - Analyze the space complexity of the recursive programs you implemented earlier.
11. Suppose you have a grid of n x m cells, where each cell is either empty or contains a rock. You're given a starting position in the grid (x,y), and you want to reach a target position (tx,ty), but you can only move in four directions (up, down, left, right) and you can only move through empty cells.You're also given a limited number of moves k that you can make. Write a program to determine if it's possible to reach the target position from the starting position within k moves.

    Example:

    n = 3, m = 3
    grid = [[0, 0, 0], [1, 1, 0], [0, 0, 0]]
    start = (0, 0)
    target = (2, 2)
    k = 6
    Output: true

12. Suppose you're building a logging library for a web application that needs to keep track of all requests and responses. You want to use the Singleton pattern to ensure that there's only one instance of the logger class throughout the application.

These exercises should give you a good starting point to practice the essential algorithm and data structure concepts mentioned earlier. You can find more programming exercises and challenges on websites like **LeetCode, HackerRank, and Codeforces**. Please solve additional exercise form the mention web sites, solve a list 15 level 1 (easy) exercise, 10 level 2 (medium) & 5 level 3 (hard) exercise.

Prepare your Uni./Course (Final) project.

Be able to answer any question, both business-wise and tech related. I.e., by ably to answer the flowing question:

1. What you did? – general description of the project
2. Why did you do that? – what purpose does your project serve?
3. How did you do that? –
4. how did you convert the business needs- the purposes – into a software project?
5. which technologies did you use?
6. What did you learn from the process?