1. What is time complexity and space complexity? What is best case, worst case and average case time complexity?

Time complexity refers to the amount of time it takes for an algorithm to run, while space complexity refers to the amount of memory an algorithm requires.

Best case time complexity is the minimum amount of time an algorithm takes to run, while **worst case** time complexity is the maximum amount of time an algorithm takes to run. **Average case** time complexity is the average amount of time an algorithm takes to run, taking into account all possible inputs.

For example, consider the problem of searching for a specific element in an array. One possible algorithm for solving this problem is to check each element in the array until the desired element is found. The time complexity of this algorithm is O(n), where n is the number of elements in the array. This means that the time taken by the algorithm increases linearly with the size of the input. In the best case, the desired element is found on the first try, so the time complexity is O(1). In the worst case, the desired element is not in the array, or is the last element in the array, so the time complexity is O(n). The average case time complexity is also O(n).

Space complexity refers to the amount of memory an algorithm requires to run. For example, the space complexity of the above algorithm is O(1), since it only requires a constant amount of memory to store the variables needed to run the algorithm, regardless of the size of the input.

2. Implement quick sort.

```
import java.util.Arrays;

public class QuickSort {
    public static void quickSort(int[] arr, int left, int right) {  // 1ST METHOD
        if (left < right) {
            int pivotIndex = partition(arr, left, right);
            quickSort(arr, left, pivotIndex - 1);
            quickSort(arr, pivotIndex + 1, right);
        }
    }
    // IF LEFT IS NOT LESS THAN RIGHT THE METHOD DOES NOTHING AND RETURN

    //OTHERWISE

    private static int partition(int[] arr, int left, int right) {
      //2 ND METHOD
        int pivot = arr[right]; //IT CHOOSES THE RIGHT MOST ARRAY
        int i = left - 1;

        for (int j = left; j < right; j++) {
            if (arr[j] <= pivot) {
                i++;
                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
```

```
        }

        int temp = arr[i + 1];
        arr[i + 1] = arr[right];
        arr[right] = temp;

        return i + 1;
    }

    public static void main(String[] args) {
        int[] arr = {5, 2, 3, 1, 4};
        quickSort(arr, 0, arr.length - 1);
        System.out.println(Arrays.toString(arr));  // Output: [1, 2, 3, 4, 5]
    }
}
```

In this implementation, the partition function is used to partition the array into the left and right subarrays. It works by selecting the rightmost element as the pivot and using a two-pointer approach to partition the array. The left pointer starts at the beginning of the array and the right pointer starts at the second-to-last element. The left pointer is incremented for every element that is less than or equal to the pivot, and the right pointer is decremented for every element that is greater than the pivot. When the pointers cross, the partition is complete and the pivot element is placed at the correct position in the sorted array.

Quick sort has a time complexity of O(n*log(n)) in the average and best case, and a time complexity of O(n^2) in the worst case. It has a space complexity of O(log(n)) in the average and worst case, and a space complexity of O(1) in the best ca

3. Implement heap sort.

```
import java.util.Arrays;

public class HeapSort {
    public static void heapSort(int[] arr) {
        int n = arr.length;

        // Build a max heap
        for (int i = n / 2 - 1; i >= 0; i--) {
            heapify(arr, n, i);
        }

        // Extract elements from the heap one by one
        for (int i = n - 1; i > 0; i--) {
            // Swap the root (maximum element) with the last element
            int temp = arr[i];
            arr[i] = arr[0];
            arr[0] = temp;

            // Heapify the remaining elements
            heapify(arr, i, 0);
        }
    }
```

```
    private static void heapify(int[] arr, int n, int i) {
        // Initialize largest as root
        int largest = i;
        // left child
        int l = 2 * i + 1;
        // right child
        int r = 2 * i + 2;

        // Find the largest element among the root, left child, and right child
        if (l < n && arr[l] > arr[largest]) {
            largest = l;
        }
        if (r < n && arr[r] > arr[largest]) {
            largest = r;
        }

        // If the largest element is not the root, swap the root with the largest
element and heapify the affected subheap
        if (largest != i) {
            int temp = arr[i];
            arr[i] = arr[largest];
            arr[largest] = temp;
            heapify(arr, n, largest);
        }
    }

    public static void main(String[] args) {
        int[] arr = {5, 2, 3, 1, 4};
        heapSort(arr);
        System.out.println(Arrays.toString(arr));  // Output: [1, 2, 3, 4, 5]
    }
}
```

To build the max heap, the heapify function is used to maintain the max heap property for the subtree rooted at index i. It works by comparing the root element with its left and right children, and swapping the root with the largest child if necessary. This process is then repeated recursively for the affected subtree.

Heap sort has a time complexity of O(n*log(n)) in all cases. It has a space complexity of O(1), since it sorts the array in-place and does not require any additional space.

4. What is hashtable? Implement hashtable using chaining method.

A **hashtable** is a data structure that allows you to efficiently store and retrieve data using keys. In a hashtable, data is stored in an array of "buckets," and each bucket is associated with a key. When you want to retrieve a piece of data from the hashtable, you use the key to determine which bucket it is stored in, and then you can retrieve the data from the bucket.

One way to implement a hashtable in Java is to use the chaining method, which involves creating a linked list for each bucket in the hashtable. When a new piece of data is added to the hashtable, it is stored in the appropriate bucket as a node in the linked list. When you want to retrieve a piece of data from the hashtable,

you use the key to determine which bucket it is stored in, and then you can search through the linked list for the node containing the data.

Here is an example of how you might implement a hashtable using the chaining method in Java:

```
import java.util.LinkedList;

public class Hashtable {
    private static final int TABLE_SIZE = 128;

    // An array of linked lists to represent the buckets in the hashtable
    private LinkedList<HashtableNode>[] buckets;

    public Hashtable() {
        // Initialize the array of buckets
        buckets = new LinkedList[TABLE_SIZE];
    }

    // A simple hash function that maps a key to a bucket
    private int hash(int key) {
        return key % TABLE_SIZE;
    }

    // Method to add a key-value pair to the hashtable
    public void put(int key, int value) {
        // Determine which bucket the key belongs in
        int bucketIndex = hash(key);

        // Initialize the bucket if it hasn't been already
        if (buckets[bucketIndex] == null) {
            buckets[bucketIndex] = new LinkedList<>();
        }

        // Search the bucket for a node with the given key
        LinkedList<HashtableNode> bucket = buckets[bucketIndex];
        for (HashtableNode node : bucket) {
            if (node.key == key) {
                // If a node with the given key already exists, update its value
                node.value = value;
                return;
            }
        }

        // If no node with the given key exists, add a new one
        bucket.add(new HashtableNode(key, value));
    }

    // Method to retrieve the value for a given key from the hashtable
    public int get(int key) {
        // Determine which bucket the key belongs in
        int bucketIndex = hash(key);
```

```
        // Get the bucket for the key
        LinkedList<HashtableNode> bucket = buckets[bucketIndex];
        if (bucket == null) {
            // If the bucket is null, the key is not in the hashtable
            return -1;
        }

        // Search the bucket for a node with the given key
        for (HashtableNode node : bucket) {
            if (node.key == key) {
                // If a node with the given key is found, return its value
                return node.value;
            }
        }

        // If no node with the given key is found, return -1
        return -1;
    }

    // A simple class to represent a key
```

5. What is load factor of hashtable? How it is related to collision handling technique?

In a **hash table**, the load factor is a measure of how full the hash table is. It is calculated by dividing the number of elements in the hash table by the size of the hash table's underlying array.

A **high load factor** means that the hash table is full and that there is a higher likelihood of collisions, or situations where two or more elements are mapped to the same hash code and therefore the same index in the array. To avoid this, the hash table may need to be resized or a different collision handling technique may need to be used.

There are several techniques that can be used to handle collisions in a hash table. One common technique is open addressing, where the hash table uses a probe sequence to search for an empty slot in the array when a collision occurs. Another technique is chaining, where each element in the hash table is stored in a linked list and collisions are resolved by adding new elements to the end of the list.

The choice of collision handling technique and the load factor of the hash table can have significant effects on the performance of the hash table. A high load factor may result in slower insertion and lookup times, while a low load factor may result in wasted space and increased memory usage.

6. Write a program to convert infix expression into postfix and then solve it?.

```
import java.util.Stack;

public class Main {
    // Dictionary to store the precedence of operators
    private static final Map<Character, Integer> precedence = new HashMap<>();
    static {
        precedence.put('+', 1);
        precedence.put('-', 1);
```

```
        precedence.put('*', 2);
        precedence.put('/', 2);
        precedence.put('^', 3);
    }


    Function to convert an infix expression to a postfix expression
    private static String infixToPostfix(String expression) {
        StringBuilder postfix = new StringBuilder();
        Stack<Character> stack = new Stack<>();
        for (char c : expression.toCharArray()) {
            // If the character is an operand, append it to the postfix expression
            if (Character.isLetterOrDigit(c)) {
                postfix.append(c);
            }
            // If the character is an opening parenthesis, push it onto the stack
            else if (c == '(') {
                stack.push(c);
            }
            // If the character is a closing parenthesis, pop operators from the
stack
            // and append them to the postfix expression until an opening
parenthesis is found
            else if (c == ')') {
                while (!stack.isEmpty() && stack.peek() != '(') {
                    postfix.append(stack.pop());
                }
                stack.pop();
            }
            // If the character is an operator, pop operators from the stack and
append them
            // to the postfix expression if they have higher or equal precedence
            else {
                while (!stack.isEmpty() && precedence.get(c) <=
precedence.get(stack.peek())) {
                    postfix.append(stack.pop());
                }
                stack.push(c);
            }
        }
        // Pop any remaining operators from the stack and append them to the
postfix expression
        while (!stack.isEmpty()) {
            postfix.append(stack.pop());
        }
        return postfix.toString();
    }
```

```
// Function to evaluate a postfix expression
private static int evaluatePostfix(String expression) {
    Stack<Integer> stack = new Stack
```

7. How to use stack to find a path between two points in a maze game?

Here's one way you could use a stack to find a path between two points in a maze game:

Create a stack and push the starting point of the maze onto the stack. While the stack is not empty: Pop the top element from the stack and make it the current point. If the current point is the destination point, you have found a path and you can stop. Otherwise, add any valid moves (up, down, left, or right) from the current point to the stack. Mark the current point as visited to avoid revisiting it. This algorithm is called depth-first search (DFS) and it explores the maze by following a path as far as possible before backtracking and trying a different path.

Here is some pseudocode that demonstrates this approach:

```
function findPath(maze, start, destination):
  stack = new Stack()
  stack.push(start)
  visited = new Set()

  while stack is not empty:
    current = stack.pop()
    visited.add(current)

    if current == destination:
      return true  # path found

    for next in getValidMoves(maze, current):
      if next not in visited:
        stack.push(next)

  return false  # no path found
```

8. How to display a singly linear linked list in reverse order (without modifying contents)?

Here's one way you can display the contents of a singly linked list in reverse order without modifying the original list:

Create a new stack. Iterate through the linked list and push each node onto the stack. After all the nodes have been pushed onto the stack, pop each node from the stack and print its value. Here is some pseudocode that demonstrates this approach:

```
void displayReverse(Node head) {
  Stack stack = new Stack()

  while (head != null) {
    stack.push(head)
    head = head.next
  }
```

```
  while (!stack.isEmpty()) {
    Node node = stack.pop()
    print(node.value)
  }
}
```

This approach has a time complexity of O(n) and a space complexity of O(n), where n is the number of nodes in the linked list.

9. How to reverse a singly linear linked list using recursion and without recursion?

Here's one way you can reverse a singly linked list using recursion:

Base case: If the head of the list is null, return null. Recursive case: Reverse the rest of the list, then set the head's next pointer to null and return the head. Here is some pseudocode that demonstrates this approach:

```
Node reverseRecursive(Node head) {
  if (head == null) return null
  if (head.next == null) return head

  Node reversed = reverseRecursive(head.next)
  head.next.next = head
  head.next = null
  return reversed
}
```

This approach has a time complexity of O(n) and a space complexity of O(n), where n is the number of nodes in the linked list.

Here's one way you can reverse a singly linked list without using recursion:

Create three pointers: **prev**, **current**, and **next**. Initialize current to the head of the list and prev to null. Iterate through the list: Set next to the next node in the list. Set the next pointer of the current node to prev. Update prev to be the current node. Update current to be next. Return prev, which is now the head of the reversed list. Here is some pseudocode that demonstrates this approach:

```
Node reverseIterative(Node head) {
  Node prev = null
  Node current = head
  Node next = null

  while (current != null) {
    next = current.next
    current.next = prev
    prev = current
    current = next
  }
```

```
    return prev
  }
```

This approach has a time complexity of O(n) and a space complexity of O(1), where n is the number of nodes in the linked list.

10. How to check if given singly linear linked list is palindrome or not? Time complexity should be O(n) and space complexity should be O(1).

Here's one way you can check if a singly linked list is a palindrome with a time complexity of O(n) and a space complexity of O(1):

Find the middle of the linked list. Reverse the second half of the linked list. Compare the first half and the reversed second half of the linked list, element by element. Reverse the second half of the linked list again and return to the original form. Return true if the linked list is a palindrome, false otherwise. *Here is some pseudocode that demonstrates this approach:*

```
boolean isPalindrome(Node head) {
  if (head == null) return true

  Node slow = head
  Node fast = head
  Node prev = null

  while (fast != null && fast.next != null) {
    fast = fast.next.next
    Node next = slow.next
    slow.next = prev
    prev = slow
    slow = next
  }

  if (fast != null) {
    slow = slow.next
  }

  while (prev != null && slow != null) {
    if (prev.value != slow.value) {
      return false
    }
    prev = prev.next
    slow = slow.next
  }

  return true
}
```

This algorithm has a time complexity of O(n) and a space complexity of O(1), where n is the number of nodes in the linked list.

11. Implement add last, traverse and search function in a generic (for any data type) doubly linked list.

Here is an implementation of a generic doubly linked list in Java that includes addLast, traverse, and search methods:

```
class DoublyLinkedList<T> {
  private static class Node<T> {
    T value;
    Node<T> next;
    Node<T> prev;

    public Node(T value, Node<T> next, Node<T> prev) {
      this.value = value;
      this.next = next;
      this.prev = prev;
    }
  }

  private Node<T> head;
  private Node<T> tail;

  public void addLast(T value) {
    Node<T> newNode = new Node<T>(value, null, tail);
    if (tail != null) {
      tail.next = newNode;
    }
    tail = newNode;
    if (head == null) {
      head = newNode;
    }
  }

  public void traverse() {
    Node<T> current = head;
    while (current != null) {
      System.out.println(current.value);
      current = current.next;
    }
  }

  public boolean search(T value) {
    Node<T> current = head;
    while (current != null) {
      if (current.value.equals(value)) {
        return true;
      }
      current = current.next;
    }
    return false;
  }
}
```

The addLast method adds a new node with the given value to the end of the list. The traverse method prints the value of each node in the list. The search method returns true if the given value is found in the list, and false otherwise

12. Implement binary search tree in-order and post-order using recursion and without recursion?

Here is an implementation of an in-order traversal of a binary search tree using recursion in Java:

```
void inOrderRecursive(Node root) {
  if (root == null) return;

  inOrderRecursive(root.left);
  visit(root);
  inOrderRecursive(root.right);
}
```

The in-order traversal of a binary search tree visits the nodes in the tree in the order left, root, right.

Here is an implementation of an in-order traversal of a binary search tree using an iterative approach in Java:

```
void inOrderIterative(Node root) {
  Stack<Node> stack = new Stack<>();
  Node current = root;

  while (current != null || !stack.isEmpty()) {
    while (current != null) {
      stack.push(current);
      current = current.left;
    }
    current = stack.pop();
    visit(current);
    current = current.right;
  }
}
```

This iterative approach uses a stack to simulate the recursive calls.

Here is an implementation of a post-order traversal of a binary search tree using recursion in Java:

```
void postOrderRecursive(Node root) {
  if (root == null) return;

  postOrderRecursive(root.left);
  postOrderRecursive(root.right);
  visit(root);
}
```

The post-order traversal of a binary search tree visits the nodes in the tree in the order left, right, root.

Here is an implementation of a post-order traversal of a binary search tree using an iterative approach in Java:

```
void postOrderIterative(Node root) {
  Stack<Node> stack = new Stack<>();
  Node current = root;

  while (current != null || !stack.isEmpty()) {
    if (current != null) {
      stack.push(current);
      current = current.left;
    } else {
      Node temp = stack.peek().right;
      if (temp == null) {
        temp = stack.pop();
        visit(temp);
        while (!stack.isEmpty() && temp == stack.peek().right) {
          temp = stack.pop();
          visit(temp);
        }
      } else {
        current = temp;
      }
    }
  }
}
```

This iterative approach uses a stack to keep track of the nodes that need to be visited and to determine when a right child has been fully processed.

13. What are different methods of implementing graph? Explain with diagram and code snippet.

There are several ways to implement a graph data structure in computer science. Here are two common methods:

Adjacency Matrix: In this method, the graph is represented as a two-dimensional matrix of size V x V, where V is the number of vertices in the graph. The matrix contains a boolean value at each position (i, j), indicating whether there is an edge between vertex i and vertex j. For example, the following adjacency matrix represents a directed graph with three vertices and four edges: Copy code 0 1 2 0 0 1 0 1 0 0 1 2 1 0 0 This representation is suitable for dense graphs, where the number of edges is close to the maximum possible number of edges.

Here is an example of how to implement a graph using an adjacency matrix in Java:

```
class Graph {
  private final int V;
  private int E;
  private boolean[][] adj;
```

```
  public Graph(int V) {
    this.V = V;
    this.E = 0;
    this.adj = new boolean[V][V];
  }

  public void addEdge(int v, int w) {
    adj[v][w] = true;
    E++;
  }

  public Iterable<Integer> adj(int v) {
    List<Integer> neighbors = new ArrayList<>();
    for (int i = 0; i < V; i++) {
      if (adj[v][i]) {
        neighbors.add(i);
      }
    }
    return neighbors;
  }
}
```

Adjacency List: In this method, the graph is represented as an array of linked lists. Each element of the array represents a vertex in the graph, and the linked list contains the vertices that are connected to it by an edge. This representation is suitable for sparse graphs, where the number of edges is much smaller than the maximum possible number of edges. *Here is an example of how to implement a graph using an adjacency list in Java:*

```
class Graph {
  private final int V;
  private int E;
  private List<Integer>[] adj;

  public Graph(int V) {
    this.V = V;
    this.E = 0;
    this.adj = (List<Integer>[]) new List[V];
    for (int i = 0; i < V; i++) {
      adj[i] = new LinkedList<>();
    }
  }

  public void addEdge(int v, int w) {
    adj[v].add(w);
    E++;
  }

  public Iterable<Integer> adj(int v) {
    return adj[v];
  }
}
```

14. How to implement BFS and DFS algorithm on graph?

Breadth-first search (BFS) and **depth-first search (DFS)** are two common algorithms for traversing the nodes of a graph.

Here is an example of how you might implement BFS in Java:

```
import java.util.LinkedList;
import java.util.Queue;

public class BFS {
    // A class to represent a graph node
    static class Node {
        int data;
        boolean visited;
        LinkedList<Node> neighbors;

        Node(int data) {
            this.data = data;
            this.visited = false;
            this.neighbors = new LinkedList<>();
        }
    }

    // A method to perform BFS on a graph
    public void bfs(Node startNode) {
        // Create a queue to store the nodes to be visited
        Queue<Node> queue = new LinkedList<>();

        // Mark the start node as visited and add it to the queue
        startNode.visited = true;
        queue.add(startNode);

        // While the queue is not empty
        while (!queue.isEmpty()) {
            // Remove the next node from the queue
            Node currentNode = queue.remove();

            // Visit the node
            visit(currentNode);

            // Add the node's neighbors to the queue if they have not been visited
            for (Node neighbor : currentNode.neighbors) {
                if (!neighbor.visited) {
                    neighbor.visited = true;
                    queue.add(neighbor);
                }
            }
        }
    }
```

```
    // A method to visit a node (in this example, simply prints the node's data)
    public void visit(Node node) {
        System.out.println(node.data);
    }
}
```

Here is an example of how you might implement DFS in Java:

```
import java.util.LinkedList;

public class DFS {
    // A class to represent a graph node
    static class Node {
        int data;
        boolean visited;
        LinkedList<Node> neighbors;

        Node(int data) {
            this.data = data;
            this.visited = false;
            this.neighbors = new LinkedList<>();
        }
    }

    // A method to perform DFS on a graph
    public void dfs(Node startNode) {
        // Mark the start node as visited
        startNode.visited = true;

        // Visit the node
        visit(startNode);

        // Recursively visit the node's neighbors
        for (Node neighbor : startNode.neighbors) {
            if (!neighbor.visited) {
                dfs(neighbor);
            }
        }
    }

    // A method to visit a node (in this example, simply prints the node's data)
    public void visit(Node node) {
        System.out.println(node.data);
    }
}
```

Both of these algorithms can be used to traverse the nodes of a graph and perform some action on each node. The main difference is that BFS uses a queue to visit the nodes in a breadth-first manner, while DFS uses recursion to visit the nodes in a depth-first manner

15. What is minimum spanning tree and its applications? Implement Prim's MST algorithm

A **minimum spanning tree (MST)** of a graph is a subgraph that is a tree, contains all the vertices of the graph, and has the minimum total weight among all the possible subgraphs that can be formed.

Minimum spanning trees have several applications in computer science and engineering, including:

Network design: MSTs can be used to design networks with the minimum cost, such as road networks or communication networks. Approximation algorithms: MSTs can be used to approximate the solution to optimization problems, such as the traveling salesman problem. One algorithm for finding the minimum spanning tree of a graph is Prim's algorithm. Here is an implementation of Prim's algorithm in Java:

```
class PrimMST {
  private boolean[] marked;
  private Queue<Edge> mst;
  private PriorityQueue<Edge> pq;

  public PrimMST(EdgeWeightedGraph G) {
    marked = new boolean[G.V()];
    mst = new LinkedList<>();
    pq = new PriorityQueue<>();

    visit(G, 0);
    while (!pq.isEmpty()) {
      Edge e = pq.poll();
      int v = e.either(), w = e.other(v);
      if (marked[v] && marked[w]) continue;
      mst.add(e);
      if (!marked[v]) visit(G, v);
      if (!marked[w]) visit(G, w);
    }
  }

  private void visit(EdgeWeightedGraph G, int v) {
    marked[v] = true;
    for (Edge e : G.adj(v)) {
      if (!marked[e.other(v)]) pq.add(e);
    }
  }

  public Iterable<Edge> edges() {
    return mst;
  }
}
```

This implementation uses a priority queue to store the edges, ordered by their weight, and a marked array to keep track of the vertices that have been visited.