Interview questions:

a) String compression

i) Memory usage and improvements

String compression code(First Compressor)

```
#include<iostream>
#include <string>
using namespace std;

void compress(string s)
{
   int num = s.length(); //calculating length of the string
   int i = 0;
   while ( i < num) {
      // Counting the repetitions of each character
      int repetition = 1;
      while (s[i] == s[i+1] && i < num-1 ) {
         repetition++;
         i++;
      }
      // Print character and its count
      cout << s[i] << repetition;
      i++;
   }
}
int main()
{
   string str = "aaaabbcddddd";
   compress(str);
}
```

The memory usage of this code depends on various factors such as the size of the input string etc. However, there are some ways to improve the code.

Memory Usage:

The primary memory usage in this code comes from the input string s and some local variables used within the functions.

The space complexity is O(1) because the memory used by local variables (e.g., num, i, repetition) is constant regardless of the size of the input.

Improvements:

- The function compress modifies the value of i within the loop. It's generally good practice to avoid modifying loop variables inside the loop header. We can consider using a for loop to make the code more readable.

- Although the space complexity is already low (O(1)), we can further optimize the code by avoiding unnecessary variables. For example, we can directly use s.length() in the loop condition without storing it in a separate variable.

- We can pass the input string as a const reference to avoid unnecessary copying of the string. This can help improve performance and memory usage, especially for large strings.

- Instead of directly printing characters to the console, we can use a stringstream or string to build the compressed string. This allows for better control over the output and avoids multiple calls to cout, which can be less efficient.

- If we know the approximate size of the compressed string, we can use reserve to preallocate memory for the string builder. This can reduce the number of reallocations and improve performance.

ii) Second Compressor

```
string secondCompress(string s)
{
    int num = s.length();
    string result = "";
    int i = 0;
    while (i < num)
    {
        char current_char = s[i];
        int repetition = 1;
        // Count repetitions of each character
        while (i + 1 < num && s[i] == s[i + 1])
        {
            repetition++;
            i++;
        }
        // Append character and its count to result
        result += current_char;
        if (repetition > 1)
        {
            result += to_string(repetition);
        }
        i++;
    }
    return result;
}
```

iii) Decompress string

```
void decompress(const string &s) {

    int num = s.length();

    int i = 0;

    while (i < num) {

        // Read character

        char character = s[i++];

        // Read count (variable number of digits)
```

```
    int count = 0;
    while (i < num && isdigit(s[i])) {
      count = count * 10 + (s[i++] - '0');
    }
   // Print the character repeated count times
    for (int j = 0; j < count; ++j) {
      cout << character;
    }   } }
```

iv) Test cases

Basic Case:

Input: "a4b2c1d5"

Expected Output: "aaaabbcddddd"

Edge Case - Empty String:

Input: ""

Expected Output: ""

Single Character:

Input: "a1"

Expected Output: "a"

No Repeating Characters:

Input: "abcdef"

Expected Output: "abcdef"

Long String with Repeating Characters:

Input*:* "a121b111c151d511"

Expected Output:
"aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaabbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb

bccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
ccccccccccccccccccccccccccccccccccccccccccccccccccccaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaa"

Random Input:

Generate random strings with different lengths and characters to test the algorithm's performance on diverse inputs.

Performance Test:

Test the performance of the algorithm on large strings to ensure it runs efficiently.

Boundary Cases:

Test the code on the boundaries of the input space, such as very long strings, strings with special characters, and strings with a mix of upper and lower case characters.

b) Linked List

i) kth to the last element of linked list

```
class Node {
   int data;
   Node next;

   Node(int data) {
      this.data = data;
      this.next = null;
   }
}

class LinkedList {
   Node head;
   LinkedList() {
      this.head = null;
   }
   // Insert a new node at the beginning of the linked list
   void push(int newData) {
      Node newNode = new Node(newData);
      newNode.next = head;
      head = newNode;
   }

   // Find the kth-to-last element of the linked list
   int findKthToLast(int k) {
      if (head == null || k <= 0) {
         System.out.println("Invalid input");
```

```
            return -1;
        }
        Node slowPointer = head;
        Node fastPointer = head;
        // Move the fast pointer k nodes ahead
        for (int i = 0; i < k; i++) {
            if (fastPointer == null) {
                System.out.println("Invalid input");
                return -1;
            }
            fastPointer = fastPointer.next;
        }
        // Move both pointers until the fast pointer reaches the end
        while (fastPointer != null) {
            slowPointer = slowPointer.next;
            fastPointer = fastPointer.next;
        }
        // Now, slowPointer is at the kth-to-last node
        return slowPointer.data;
    }
    public static void main(String[] args) {
        LinkedList list = new LinkedList();
        list.push(10);
        list.push(20);
        list.push(30);
        list.push(40);
        list.push(50);

        int k = 2;
        int kthToLast = list.findKthToLast(k);

        if (kthToLast != -1) {
            System.out.println("The " + k + "th-to-last element is: " + kthToLast);
        }
    }
}
```

ii) Minimize the number of times you run through the loop

```
// Move both pointers until the fast pointer reaches the end
    while (fastPointer.next != null) { // Changed the loop condition here
        slowPointer = slowPointer.next;
        fastPointer = fastPointer.next;
    }
```

c) Stack

i)Function 'min'

```
import java.util.Stack;
```

```
class MinStack {
    private Stack<Integer> mainStack;
    private Stack<Integer> minStack;
    MinStack() {
        mainStack = new Stack<>();
        minStack = new Stack<>();
    }
    // Push element onto the stack
    void push(int value) {
        mainStack.push(value);

        // If the minStack is empty or the new element is smaller than or equal to the current
minimum,
        // push the new element onto the minStack.
        if (minStack.isEmpty() || value <= minStack.peek()) {
            minStack.push(value);
        }
    }

    // Pop element from the stack
    void pop() {
        if (!mainStack.isEmpty()) {
            int poppedElement = mainStack.pop();

            // If the popped element is the current minimum, pop from the minStack as well.
            if (poppedElement == minStack.peek()) {
                minStack.pop();
            }
        }
    }

    // Get the minimum element in the stack
    int min() {
        if (!minStack.isEmpty()) {
            return minStack.peek();
        } else {
            System.out.println("Stack is empty.");
            return -1; // or throw an exception
        }
    }

    public static void main(String[] args) {
        MinStack stack = new MinStack();

        stack.push(3);
        stack.push(5);
        System.out.println("Minimum: " + stack.min()); // Output: 3
```

```
        stack.push(2);
        stack.push(1);
        System.out.println("Minimum: " + stack.min()); // Output: 1
        stack.pop();
        System.out.println("Minimum: " + stack.min()); // Output: 2
        stack.pop();
        System.out.println("Minimum: " + stack.min()); // Output: 3
    }
}
```

In this implementation, the minStack is used to keep track of the minimum element at each level of the main stack. The push operation checks whether the new element is smaller than or equal to the current minimum, and if so, it's pushed onto the minStack. The pop operation checks whether the popped element is the current minimum, and if so, it's also popped from the minStack. The min function returns the top element of the minStack, which represents the current minimum in the main stack.

ii) Real-world scenario

Let's consider a scenario where a user navigates through various web pages while browsing. The order in which the user visits these pages can be effectively managed using a stack. Here's how it works:

- **Push Operation:**
  When the user opens a new page, the URL or page identifier is pushed onto the stack.
  For each page visited, it gets added to the top of the stack.
- **Pop Operation:**
  When the user clicks the "Back" button, the most recent page is popped from the stack.
  This operation allows the user to navigate back to the previously visited page.
- **Comparison with Arrays:**
  Using an array to store the browsing history might involve unnecessary resizing and shifting of elements when navigating back and forth.
  While a stack allows for efficient operations as elements are always inserted and removed from the top.
- **Benefits of Using a Stack:**
  The order of page visits is naturally maintained using the Last In First Out (LIFO) property of the stack.
  The stack ensures that the user goes back to the exact previous page visited, preserving the navigation history.
  It simplifies the management of the browsing history by keeping track of the visited pages in a structured manner.

d) Trapping rain water

```
public static int maxWater(int[] arr, int n)
    {
```

```
        // To store the maximum water
        // that can be stored
        int res = 0;
        // For every element of the array
        // except first and last element
        for (int i = 1; i < n - 1; i++) {
                // Find maximum element on its left
                int left = arr[i];
        for (int j = 0; j < i; j++) {
                        left = Math.max(left, arr[j]);
                }
                // Find maximum element on its right
                int right = arr[i];
                for (int j = i + 1; j < n; j++) {
                        right = Math.max(right, arr[j]);
                }
                // Update maximum water value
                res += Math.min(left, right) - arr[i];
        }
        return res;
    }
```

For input => int[] arr = { 2 ,1, 3 ,0, 1, 2, 3}; // Output : 7

e) Coin Change

Scenario 1: Base Case (Change is 0)

You need to provide change, but the change required is zero.

Explanation:

If no change is needed, there's no need to give any coins.

Optimal Solution: An empty set of coins.

Scenario 2: Negative Change

The amount of change required is negative (e.g., -5).

Explanation:

It doesn't make sense to give negative change.

Optimal Solution: Indicate that change is not possible.

Scenario 3: Dynamic Programming Approach

You have a list of coin denominations and a specific amount of change to provide.

Explanation:

Create a list to keep track of the minimum number of coins needed for each possible amount of change. For each coin denomination, figure out the minimum number of coins needed to make up each amount of change.

Scenario 4: Building the Solution

You have the dynamic programming list filled with minimum coin counts.

Explanation:

Start from the end of the list (representing the given change).

Move backward, selecting the coin that contributes to the minimum number of coins needed at each step.

As you backtrack, you're building the combination of coins that gives the optimal solution

ii) Greedy Algorithm and Dynamic Programming

Greedy Algorithm:

A greedy algorithm is an approach to problem-solving that makes locally optimal choices at each stage with the hope of finding a global optimum. It makes the best possible decision at each step without considering the consequences of that decision on future steps. Greedy algorithms are particularly useful for optimization problems where a series of choices need to be made to maximize or minimize some objective function.

Dynamic Programming:

Dynamic programming is a method for solving complex problems by breaking them down into simpler overlapping subproblems.Dynamic programming is particularly useful when a problem exhibits optimal substructure, meaning an optimal solution to the problem can be constructed from optimal solutions of its subproblems.

Role of Dynamic Programming with Greedy Algorithms:

*** Overlapping Subproblems:**

Greedy algorithms don't typically involve solving the same subproblem multiple times.

Dynamic programming is particularly beneficial when a problem has overlapping subproblems, and solutions to the same subproblems are reused.

***Memoization:**

Dynamic programming often uses memoization, where solutions to subproblems are stored in a table.

This can improve the efficiency of the algorithm by avoiding redundant computations.

***Optimizing Greedy Choices:**

Dynamic programming can be used to optimize the choices made by a greedy algorithm.

By storing and reusing solutions to subproblems, dynamic programming can enhance the efficiency of a greedy algorithm.

***Example: Shortest Path Problem:**

Dijkstra's algorithm, a greedy algorithm for finding the shortest path, can be optimized using dynamic programming techniques.

iii) Largest Possible Number

Example => **Input**: 19374

1. Choose 1 (current maximum), remove it => Remaining: 9375

2. Choose 9 (current maximum), remove it => Remaining: 375

3. Choose 3 (current maximum), remove it => Remaining: 75

4. Choose 7 (current maximum), remove it => Remaining: 5

5. Choose 5 (current maximum), remove it => Remaining: (empty)

Output: 9375

This problem can be solved using a greedy algorithm. The intuition behind the greedy approach in this case is to iteratively choose the current maximum digit as long as it doesn't compromise the overall goal of creating the largest number.

Explanation

Start from the Left:

Iterate through the digits from left to right.

For each digit, consider removing it if doing so results in a larger number.

Greedy Choice:

At each step, choose the current maximum digit.

The goal is to always have the largest possible digit as the leftmost digit.

Remove One Digit:

Remove the chosen digit, and continue the process with the remaining digits.

Repeat the process until the ultimate goal is achieved.

Why is this a Greedy Algorithm?

The choice made at each step (removing a digit) is based solely on the local optimal of selecting the current maximum digit. The algorithm doesn't consider the global context of all possible combinations of digits. Despite this, the algorithm often produces the correct and optimal result, demonstrating the greedy choice property.

f) Dot Product and Cross Product

Dot Product:

Definition: The dot product, also known as the scalar product, is a binary operation that takes two equal-length sequences of numbers and returns a single number. It is defined as the sum of the product of corresponding entries.

Formula: For two vectors A and B, the dot product is given by $A \cdot B = |A| * |B| * \cos(\theta)$, where θ is the angle between the vectors.

Use Cases in Graphics:

Cosine of the Angle: Dot product is used to find the cosine of the angle between two vectors. This is crucial in lighting calculations, where knowing the angle between the light source and the surface normal is important for shading.

Projection: It's used to find the projection of one vector onto another, which is useful in various geometric and physics calculations.

Cross Product:

Definition: The cross product is a binary operation on two vectors in three-dimensional space, producing another vector that is perpendicular to both input vectors.

Formula: For two vectors A and B, the cross product is given by $A \times B = |A| * |B| * \sin(\theta) * n$, where θ is the angle between the vectors, and n is the unit vector perpendicular to the plane formed by A and B.

Use Cases in Graphics:

Surface Normal: Cross product is commonly used to find the surface normal of a polygon. This normal is crucial for lighting calculations and determining how light interacts with the surface.

Rotation: Cross product is used in rotation calculations and determining orientation in 3D space.

Resources:
https://www.khanacademy.org/science/physics/magnetic-forces-and-magnetic-fields/electric-motors/v/dot-vs-cross-product
https://learnwebgl.brown37.net/09_lights/lights_diffuse.html

Calculating the intersection between a ray and a triangle involves mathematical computations such as determining

- if the ray intersects the plane of the triangle and
- then checking if the intersection point lies within the triangle.

g) My favorite subject is Data Structures because:

It forms the fundamental building blocks for organizing and storing data efficiently, crucial for optimizing algorithms and problem-solving.
Data Structures provide a systematic way to manage and manipulate data, enabling the development of robust and scalable software applications.
The study of Data Structures enhances problem-solving skills by offering diverse ways to represent and traverse information, fostering creativity in algorithm design.
Understanding Data Structures is essential for effective programming and contributes to writing clean, modular, and maintainable code.
Exploring Data Structures deepens my insight into the core principles of computer science, facilitating a solid foundation for various advanced topics like algorithms, databases, and software engineering.

h) Random crashes in a software application can be challenging to diagnose and fix. Here are some approaches

1. Review the Code:

Start by thoroughly reviewing the code for any obvious issues, such as null pointer dereferences, uninitialized variables, or buffer overflows.

Utilize static code analysis tools to identify potential vulnerabilities and coding errors.

2. Debugging Techniques:

Use a debugger to catch the crash when it occurs.

Examine the call stack, variables, and memory to understand the state of the application during the crash.

Look for memory-related issues, such as accessing freed memory or writing to read-only memory

3. Logging and Error Handling:

Introduce comprehensive logging to record the program state leading up to the crash.

Examine the logs to identify patterns or commonalities that may hint at the cause.

Ensure robust error handling to gracefully handle unexpected situations and prevent crashes

4. Environment and Dependencies:

Verify that the application's dependencies are up to date and compatible with each other.

Check if variations in environment variables contribute to the crashes.

5. Concurrency and Thread Safety:

Investigate potential race conditions or deadlock situations in a multithreaded environment.

Verify that shared data structures are accessed safely by multiple threads.

6. Resource Utilization:

Check for resource limits, such as reaching maximum file descriptors, memory, or CPU usage.

Look for memory leaks that may accumulate over time.

7. Dynamic Analysis Tools:

Use dynamic analysis tools, such as AddressSanitizer or Valgrind, to detect memory issues.

Apply fuzz testing to generate random inputs and discover potential edge cases.

8. Compiler and Optimization Settings:

Examine compiler flags and optimization settings. Sometimes, aggressive optimizations may introduce subtle bugs.

9.Memory Hierarchy:

Analyze how the application interacts with the memory hierarchy (caches, RAM).

Cache misses or memory contention can impact performance and stability.

10.Processor Architecture:

Understand the specifics of the underlying processor architecture.

Instruction reordering, out-of-order execution, or memory consistency models may contribute to issues.

11.Interrupts and Exception Handling:

Investigate if interrupts or exceptions are affecting the program flow.

12.Hardware Failures:

Consider the possibility of hardware issues, such as faulty RAM or overheating.

13.Low-Level Profiling:

Use low-level profiling tools to analyze CPU and memory usage at the assembly level.

14.Compiler Intrinsic Functions:

Check if the code uses compiler intrinsic functions that are architecture-dependent.

I declare that I have done the above work by myself and not worked with anyone or got help from any individual on the internet.