(a) String Compression

- Compressing a2b2c1a3c3 into ab2c1ac3

```
function compress(str) {
  let compress1 = '';
  const count = 1;
  for (let i = 0; i < str.slength; i++) {
    if (str[i] === str[i+1]) {
      count++;
    } else {
      Compress1 += str[i] + (count > 1 ? count : '');
      count = 1;
    }
  }
  return compress1;
}

let input = 'a2b2c1a3c3';
let output = compress(input);
console.log(output); // prints 'ab2c1ac3'
```

- This function iterates through the input string, keeping track of the count of consecutive character that are the same. When a different character is encountered, the function appends the character and count to be compressed string, resetting the count to 1 for the new character. If the count is 1 it does not append the count to the compressed string. Finally, the function returns the compressed string.

- Decompress the string

```
function decompressString(str) {
  let decompresStr = "";
  const currChar = "";
  const charCount = "";

  for (let i = 0; i < str.length; i++) {
    const currCharCode = str.charCodeAt(i);

    if (currCharCode >= 97 && currCharCode <= 122) {
      // lowercase letter
      if (charCount === "") {
        // no count yet, just append the letter
        decompresStr += currChar;
      } else {

        decompressStr += currChar.repeat(Number(charCount));
        charCount = "";
        decompressStr += str.charAt(i);
        currChar = str.charAt(i);
      }
    } else {
      charCount += str.charAt(i)
```

```
    }
  }

  // handle the last character group
  if (charCount === "") {
    decompressStr += currChar;
  } else {
    decompressStr += currChar.repeat(Number(charCount));
  }

  return decompressStr;
}

console.log(decompressString("ab2c1ac3")); // output: "aabbcaaaccc"
```

- In the specific case, the input string “ab2c1ac3” is decompressed to “aabbcaaaccc” as desired.
- The decompress string function takes a compressed string as input and returns the decompress version of that string. It uses a loop to iterate through each character in the input string. For each character, it checks if it’s a lowercase letter or a digit. If it’s a letter, it appends the current character to the decompressed string if there’s no count yet, or repeats the current character the number of times specified by the count if there’s a count. If it’s a digit, it updates the count.

(b) The Kth to the last element of the Linked list

```
function kthToLastElement(head, k) {
  const slow = head;
  const fast = head;

  // move the fast pointer K nodes ahead of the slow pointer
  for (let i = 0; i < k; i++) {
    if (fast === null) {
      // list length is less than K, return null
      return null;
    }
    fast = fast.next;
  }

  // move both pointers until the fast pointer reaches the end of the list
  while (fast !== null) {
    slow = slow.next;
    fast = fast.next;
  }

  // slow pointer is now pointing at the Kth to the last element
  return slow.value;
}
```

- The function takes the head of the linked list and the value of K as input, and returns the value of the Kth to the last element of the linked list. It initializes both pointers to the head of the list and moves the fast pointer K nodes ahead of the slow pointer. If the list length is less than K,

it returns null. Otherwise, it moves both pointers until the fast pointer reaches the end of the list, at which point the slow pointer will be the pointing at the Kth to the last element.

(c) Stack is better used data structure than array

- A stack is the linear and abstract data structure, which stores elements in LIFO (Last in first out) order. This is the approach used in all data structures, however, there are many real-life applications of data structures.
- The operating system utilizes a stack for keeping track of memory in the operating systems. After the program is allocated to memory you will see that the operating system pushes the memory address to the stack.
  Some of the real-life applications of data structures are discussed below:
- When an individual is trying to make changes in a document, the text editor will store the previous versions in a stack. Hence, when you press the undo button, the previous version will be restored. This 'do' and 'undo' of text editors is done through the data structure.
- By evaluating the arithmetic expressions, you might be able to use a stack for storing intermediate results or performing necessary calculations in the right order. Hence, from data structures, you can perform an evaluation of arithmetic expressions.
- When you visit websites, you will find browser stores the URL of a page in the stack. After hitting the back button you will see that browser history will show the most recent URL where you navigated to it.

d) Program to determine the amount of water of trapped

```
import java.io.*;
import java.util.*;

// Java implementation of the approach
class GFG {

    // Function to return the maximum
    // water that can be stored
    public static int maximumWater(int[] height)
    {
        // Stores the indices of the bars
        Stack<Integer> stack = new Stack<>();

        // size of the array
        int n = height.length;

        // Stores the final result
        int ans = 0;

        // Loop through the each bar
        for (int i = 0; i < n; i++) {

            // Remove bars from the stack
            // until the condition holds
```

```
                while ((!stack.isEmpty())
                        && (height[stack.peek()] < height[i])) {

                    // store the height of the top
                    // and pop it.
                    int pop_height = height[stack.peek()];
                    stack.pop();

                    // If the stack does not have any
                    // bars or the popped bar
                    // has no left boundary
                    if (stack.isEmpty())
                        break;

                    // Get the distance between the
                    // left and right boundary of
                    // popped bar
                    int distance = i - stack.peek() - 1;

                    // Calculate the min. height
                    int min_height
                        = Math.min(height[stack.peek()],
                                   height[i])
                        - pop_height;

                    ans += distance * min_height;
                }

                // If the stack is either empty or
                // height of the current bar is less than
                // or equal to the top bar of stack
                stack.push(i);
            }

            return ans;
        }
        // Driver code
        public static void main(String[] args)
        {
            int arr[] = { 0, 3, 0, 1, 2, 0, 1, 3, 3, 4};
            System.out.print(maximumWater (arr));
        }
    }
```

- 8 units of water will be trapped

e) Scenarios of coin change

- You are given an array of coins with varying denominations and an integer sum representing the total amount of money; you must return the fewest coins required to make up that sum; if that sum cannot be constructed, return -1.

- And you are given a sequence of coins of various denominations as part of the coin change problem. For example, consider the following array a collection of coins, with each element representing a different denomination.
- For example, if you want to reach 76 using the above denominations, you will need the four coins listed below.
  { 4, 2, 10, 60 };

Greedy algorithm & dynamic programming of this case.

- Greedy programming is the approach that tries to solve a problem as quickly as possible, while dynamic programming is the approach that tries to solve a problem as efficiently as possible. In greedy programming, you try to solve a problem as quickly as possible by trying to find the smallest possible solution.
- Dynamic programming is an algorithmic technique for solving optimization problems by breaking them down into simpler subproblems and exploiting the fact that the optimal solution to the overall situation is dependent on the optimal solution to its subproblems and taking advantage of the fact that the optimal solution to the overall situation is dependent on the optimal solution to its subproblems.
  Dynamic Programming is a programming procedure that combines the precision of a complete search with the efficiency of greedy algorithms.

f) Dot product of Cross product

The cross product is mostly used to determine the vector, which is perpendicular to the plane surface spanned by two vectors, whereas the dot product is used to find the angle between two vectors or the length of the vector.

- Bonus – Calculate the intersection between a ray and a plane
  A plane is defined by the equation: Xx + Yy + Zz + W= 0, or the vector [X Y Z W]. ...
  A ray is defined by: R1 = [A1, B1, C1]
  Rq = [Aq, Bq, Cq]
  so R(t) = R1 + t * Rq , t > 0.
  To determine if there is an intersection with the plane, substitute for R(t) into the plane equation.

g) I don't have any favorite engineering subjects. So I have enrolled the Java full stack course and in that my favorite subject in that is HTML, CSS , JAVASCRIPT and REACT JS

h) Random Crashes

There could be some general cases which may result in such behavior,

- Random Variable : The application may be using some random variable, like random number generator or a specific time of the day or a user input etc which may be causing this.
- Uninitialized Variable : May be there is some uninitialized variable, which takes an arbitrary value each time and those values are causing such drastic behavior.
- Memory Leak : The program may have run out of memory, maybe heap overflow or something.
- External Dependencies : The program may depend on some other application which is causing this.
- Other process on machine : Maybe there are some other processes, running on machine causing it.

We can approach this problem by elimination, like close all other applications in the system or use some runtime tools to dig deeper when the problem occurs.

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