String Compression

Bonus 1 Code

```
def compress_string(s):
    if not s:  # if the string is empty, return an empty string
        return ""

    compressed = ""
    curr_char = s[0]
    curr_count = 1

    for i in range(1, len(s)):
        if s[i] == curr_char:
            curr_count += 1
        else:
            compressed += f"{curr_char}{curr_count}"
            curr_char = s[i]
            curr_count = 1

    compressed += f"{curr_char}{curr_count}"  # add the last character and count to the compressed
string

    return compressed if len(compressed) < len(s) else s  # return the compressed string if it is shorter
than the original, otherwise return the original string
print(compress_string("abbbaaaaaaccdaaab"))
```

#Bonus 2 Code

```
def solve(s):
  res = ""
  cnt = 1
  for i in range(1, len(s)):
    if s[i - 1] == s[i]:
      cnt += 1
    else:
      res = res + s[i - 1]
      if cnt > 1:
        res += str(cnt)
      cnt = 1
  res = res + s[-1]
  if cnt > 1:
    res += str(cnt)
  return res
s = "abbbaaaaaaccdaaab"
print(solve(s))
```

#Linked list Find Kth to the last element

```
def kth_to_last(head, k):
    # Initialize two pointers
    p1 = head
    p2 = head

    # Move p1 k steps forward in the linked list
    for i in range(k):
        if p1 is None:  # if the list is too short, return None
            return None
        p1 = p1.next

    # Move p1 and p2 forward together until p1 reaches the end of the list
    while p1 is not None:
        p1 = p1.next
        p2 = p2.next

    # At this point, p2 is pointing to the kth to the last element of the linked list
    return p2.value
```

```
#Stack Minimum in 0(1) time
class Stack:
    def __init__(self):
        self.stack = []
        self.min_stack = []

    def push(self, val):
        self.stack.append(val)
        if not self.min_stack or val <= self.min_stack[-1]:
            self.min_stack.append(val)

    def pop(self):
        if not self.stack:
            return None
        val = self.stack.pop()
        if val == self.min_stack[-1]:
            self.min_stack.pop()
        return val

    def min(self):
        if not self.min_stack:
            return None
        return self.min_stack[-1]
```

real world use case of stack is better used than arrays

Stacks are often preferred over arrays when the order of element insertion and removal is critical. They are commonly used for undo/redo functionality, function call stacks in programming languages, expression evaluation, browser history, and text editors. Stacks provide a natural and efficient way to implement these functionalities since they allow elements to be inserted and removed in a specific order. Overall, the use of stacks is beneficial in scenarios where the order of element insertion and removal is important and provides an effective solution for handling such situations.

d) Given an array of integers representing the elevation of a roof structure at various positions, each position is separated by a unit length, Write a program to determine the amount of water that will be trapped on the roof after heavy rainfall

I did not understand how to implement analog values mam

question E

The problem is to find the minimum number of coins needed to make change for a given amount. One way to solve this problem is to use a recursive function that checks all the possible combinations of coins to make change for the given amount. However, this approach can be very slow as it computes the same subproblems repeatedly.

To solve this problem efficiently, we can use a dynamic programming approach. In this approach, we create a table to store the minimum number of coins needed to make change for all possible amounts up to the given amount. We initialize the table with 0 for the first amount and infinity for all other amounts.

Then, for each coin denomination, we iterate through all the possible amounts and update the table with the minimum number of coins needed to make change for that amount using that coin denomination. We repeat this process for all coin denominations until we reach the final amount.

Finally, the value in the last cell of the table is the minimum number of coins needed to make change for the given amount.

Overall, the dynamic programming approach is faster and more efficient than the recursive approach because it avoids computing the same subproblems multiple times. """

Explain what is a greedy algorithm and how dynamic programming helps in this case.

A greedy algorithm is a problem-solving approach that makes the locally optimal choice at each step with the hope of finding a global optimum. In other words, it chooses the best option available at the moment without considering the future implications of that choice.

While greedy algorithms can be fast and efficient, they do not always produce the optimal solution. Sometimes, a suboptimal choice at an earlier stage can lead to a better outcome later on.

This is where dynamic programming can help. Dynamic programming is a problem-solving technique that uses a combination of recursion and memoization to efficiently solve problems by breaking them down into smaller subproblems and solving them only once.

By using dynamic programming, we can find the optimal solution to a problem by considering all possible choices at each step and choosing the one that leads to the best outcome in the long run. In contrast to the greedy algorithm approach, dynamic programming takes a more global view of the problem and considers the future implications of each choice."""

given a number N, remove one digit and print the largest possible number.

```
def remove_one_digit(N):
    N_str = str(N)
    largest = -1

    for i in range(len(N_str)):
        new_str = N_str[:i] + N_str[i+1:]
        new_num = int(new_str)

        if new_num > largest:
            largest = new_num

    return largest
print(remove_one_digit(1234))
```

What is dot product and cross product? Explain use cases of where dot product is used andcross product is used in graphics environment. Add links to places where you studied thisinformation and get back with the understanding.

I studied it on geeksforgeeks.org and stackoverflow.com

The dot product and cross product are two fundamental operations used in vector algebra and are widely used in graphics and computer vision applications.

The dot product is also known as the scalar product, and it is the product of the magnitudes of two vectors and the cosine of the angle between them. The dot product of two vectors A and B is given by:

$A \cdot B = |A|\ |B| \cos\theta$

where $|A|$ and $|B|$ are the magnitudes of vectors A and B, respectively, and θ is the angle between them.

The dot product is useful in many graphics applications, such as lighting and shading calculations, where it is used to compute the angle between the light source and the surface normal of an object. It is also used in physics simulations to calculate work done by a force on an object.

The cross product is also known as the vector product, and it is a binary operation that produces a vector perpendicular to the two vectors being multiplied. The cross product of two vectors A and B is given by:

$A \times B = |A|\ |B| \sin\theta\ n$

where $|A|$ and $|B|$ are the magnitudes of vectors A and B, respectively, θ is the angle between them, and n is the unit vector perpendicular to both A and B. """

How do you calculate the intersection between a ray and a plane/sphere/triangle?

For the intersection between a ray and a sphere, we use the equation of a sphere and the distance formula. For the intersection between a ray and a triangle, we use barycentric coordinates and the equations of the edges of the triangle.

Explain a piece of code that you wrote which you are proud of? If you have not written anycode, please write your favorite subject in engineering studies. We can go deep into thatsubject.

My favourite subject is machine learning and i wanna build something like chat gpt.

why do random crashes occur? How do you isolate the root cause of the crash?

Random crashes in a codebase could be caused by a variety of factors, including memory leaks, uninitialized variables, pointer errors, stack/heap overflows, threading issues, race conditions, and external dependencies. To isolate the cause of the crash, one could start by analyzing the stack trace generated by the debugger, which would provide clues about where the crash occurred. One could also examine the code for common coding mistakes that could lead to crashes, such as null pointer dereferences or division by zero. It may also be necessary to use tools like memory profilers and debuggers to track down the root cause of the issue. Additionally, conducting regression testing to identify when the issue started can also be helpful in isolating the root cause .