Question 1:

You're organizing a workshop day at a local community center, offering various activities that participants can join. Each activity has a start time and an end time, and due to space limitations, only one activity can take place at a time. Your task is to select the maximum number of activities that can be conducted throughout the day without any overlaps, ensuring participants have a diverse and engaging schedule. The goal is to make the most out of the available time, starting the next activity as soon as the previous one finishes.

Input Format:

- The first line contains a single integer, `n`, the number of activities planned for the day.
- The second line contains `n` space-separated integers, the start times of the activities.
- The third line contains `n` space-separated integers, the finish times of the activities.

Output Format:

- Print the selected activities in the format `(start_time, finish_time),` indicating the activities that can be conducted back-to-back throughout the day.

Title for Question 1: Activity Selection Problem

Solution:

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

void selectActivities(int start[], int finish[], int n) {

    for (int i = 0; i < n; ++i) {
        for (int j = i + 1; j < n; ++j) {
            if (finish[i] > finish[j]) {
                swap(finish[i], finish[j]);
                swap(start[i], start[j]);
            }
        }
    }

    cout << "Selected activities: ";
    cout << "(" << start[0] << ", " << finish[0] << ") ";

    int prev_finish = finish[0];
    for (int i = 1; i < n; i++) {
        if (start[i] >= prev_finish) {
            cout << "(" << start[i] << ", " << finish[i] << ") ";
            prev_finish = finish[i];
        }
    }
}

int main() {
    int n;
    cin >> n;
    int start[n], finish[n];
    for (int i = 0; i < n; i++) {
        cin >> start[i];
    }
    for (int i = 0; i < n; i++) {
        cin >> finish[i];
    }
    selectActivities(start, finish, n);
    return 0;
}
```

TestCases:

S.No	Inputs	Outputs
1	6 1 3 0 5 8 5 2 4 6 7 9 9	Selected activities: (1, 2) (3, 4) (5, 7) (8, 9)
2	4 3 2 1 5 4 3 2 6	Selected activities: (1, 2) (2, 3) (3, 4) (5, 6)
3	3 1 1 1 2 3 4	Selected activities: (1, 2)
4	7 1 2 3 4 5 6 7 2 3 4 5 6 7 8	Selected activities: (1, 2) (2, 3) (3, 4) (4, 5) (5, 6) (6, 7) (7, 8)
5	5 3 1 0 5 8 4 2 6 7 9	Selected activities: (1, 2) (3, 4) (5, 7) (8, 9)
6	7 5 6 1 3 0 5 8 9 7 4 2 6 7 9	Selected activities: (3, 2) (6, 7) (8, 9)

White List:

Black List:

Question 2:

You're a treasure hunter who has just discovered a cave filled with various artifacts and gems. Each item has a weight and a value associated with it. Your backpack can only carry a certain weight, so you must choose the items wisely to maximize the value of your haul. Some items can be divided and taken in parts if necessary. Your task is to determine the maximum profit you can make by selecting items in a way that the total weight doesn't exceed the capacity of your backpack.

Input Format:

- The first line contains a single integer, `W`, the maximum weight your backpack can carry.
- The second line contains a single integer, `N`, the number of items in the cave.
- The third line contains `N` space-separated integers, representing the values (profits) of the items.
- The fourth line contains `N` space-separated integers, representing the weights of the items.

Output Format:

- Print a single line stating "Maximum profit: " followed by the maximum value (profit) you can carry in your backpack, rounded to a decimal place if necessary.

Title for Question 2: Fractional Knapsack

Solution:

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

// Comparison function to sort items according to profit/weight ratio
static bool cmp(int profit1, int weight1, int profit2, int weight2)
{
    double r1 = (double)profit1 / (double)weight1;
    double r2 = (double)profit2 / (double)weight2;
    return r1 < r2;
}

double fractionalKnapsack(int W, int profit[], int weight[], int N)
{
    // Sorting items on basis of ratio
    for (int i = 0; i < N; ++i) {
        for (int j = i + 1; j < N; ++j) {
            if (cmp(profit[i], weight[i], profit[j], weight[j])) {
                swap(profit[i], profit[j]);
                swap(weight[i], weight[j]);
            }
        }
    }

    double finalvalue = 0.0;

    for (int i = 0; i < N; i++) {

        if (weight[i] <= W) {
            W -= weight[i];
            finalvalue += profit[i];
        }

        // If we can't add the current item, add fractional part of it
        else {
            finalvalue += profit[i] * ((double)W / weight[i]);
            break;
        }
    }

    return finalvalue;
}

int main()
{
    int W, N;
    cin >> W;
    cin >> N;
    int profit[N], weight[N];
    for (int i = 0; i < N; ++i) {
        cin >> profit[i];
    }
    for (int i = 0; i < N; ++i) {
        cin >> weight[i];
    }
    cout << "Maximum profit: " << fractionalKnapsack(W, profit, weight, N
    return 0;
}
```

TestCases:

S.No	Inputs	Outputs
1	50 3 60 100 120 10 20 30	Maximum profit: 240
2	25 4 10 12 20 15 2 3 5 7	Maximum profit: 57
3	15 5 25 30 35 40 45 3 4 5 6 7	Maximum profit: 110
4	30 2 100 120 10 20	Maximum profit: 220
5	40 3 20 30 25 5 8 6	Maximum profit: 75
6	35 3 50 60 70 10 15 20	Maximum profit: 145

White List:

Black List:

Question 3:

You're developing a feature for a text messaging app that compresses messages by encoding sequences of repeated characters. This feature is especially useful for reducing the size of messages that contain long sequences of the same character, which is common in certain types of informal communication or when sending visual patterns made of text. The app takes a string as input and outputs a compressed version where each sequence of identical characters is replaced with a single instance of that character followed by the count of repetitions. This encoding helps to save data and makes longer messages more compact without losing information.

Input Format:

- The first and only line contains a string, `str`, representing the message to be compressed.

Output Format:

- **?**Print a single string, the compressed version of the input message, where each sequence of identical characters is replaced by that character followed by the number of times it repeats consecutively.

Title for Question 3: String Compression

Solution:

```
#include<iostream>
using namespace std;
int main()
{
    string str, ans;
    int ind, itr, iter = 0;
    cin >> str;
    for (ind = 0; ind < str.length();)
    {
        char current_char = str[ind];
        int count = 1;
        for (itr = ind + 1; itr < str.length(); itr++)
        {
            if (str[itr] == current_char)
            {
                count++;
            }
            else
            {
                break;
            }
        }
        ans += current_char;
        ans += to_string(count);
        ind += count;
    }
    cout << ans;
    return 0;
}
```

TestCases:

S.No	Inputs	Outputs
1	aabcccccaaa	a2b1c5a3
2	aaabbcccc	a3b2c4
3	abcdefg	a1b1c1d1e1f1g1
4	abb	a1b2
5	ababab	a1b1a1b1a1b1
6	zzzxxxxxyyy	z3x5y3

White List:

Black List:

Question 4:

Imagine you are an event planner tasked with scheduling various workshops in a single venue during a one-day conference. Each workshop has a start time and an end time. To maximize the use of the venue, you want to select the maximum number of non-overlapping workshops. This means that no two selected workshops can run at the same time. Your goal is to provide a schedule that includes the greatest number of workshops, taking into account their start and end times.

Input Format:

- The first line contains an integer `n`, the number of workshops.
- The next line contains `n` space-separated integers, representing the start times of the workshops.
- The following line contains `n` space-separated integers, representing the end times of the workshops.

Output Format:

- A line with the phrase "Selected activities: ", followed by a list of the selected workshops in the format `(start_time, end_time),`.

Title for Question 4: Activity Selection Problem 1

Solution:

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

void selectActivities(int start[], int finish[], int n) {

    for (int i = 0; i < n; ++i) {
        for (int j = i + 1; j < n; ++j) {
            if (finish[i] > finish[j]) {
                swap(finish[i], finish[j]);
                swap(start[i], start[j]);
            }
        }
    }

    cout << "Selected activities: ";
    cout << "(" << start[0] << ", " << finish[0] << ") ";

    int prev_finish = finish[0];
    for (int i = 1; i < n; i++) {
        if (start[i] >= prev_finish) {
            cout << "(" << start[i] << ", " << finish[i] << ") ";
            prev_finish = finish[i];
        }
    }
}

int main() {
    int n;
    cin >> n;
    int start[n], finish[n];
    for (int i = 0; i < n; i++) {
        cin >> start[i];
    }
    for (int i = 0; i < n; i++) {
        cin >> finish[i];
    }
    selectActivities(start, finish, n);
    return 0;
}
```

TestCases:

S.No	Inputs	Outputs
1	7 5 6 1 3 0 5 8 9 7 4 2 6 7 9	Selected activities: (3, 2) (6, 7) (8, 9)
2	3 1 1 1 2 3 4	Selected activities: (1, 2)
3	6 1 3 0 5 8 5 2 4 6 7 9 9	Selected activities: (1, 2) (3, 4) (5, 7) (8, 9)
4	4 3 2 1 5 4 3 2 6	Selected activities: (1, 2) (2, 3) (3, 4) (5, 6)
5	7 1 2 3 4 5 6 7 2 3 4 5 6 7 8	Selected activities: (1, 2) (2, 3) (3, 4) (4, 5) (5, 6) (6, 7) (7, 8)
6	5 3 1 0 5 8 4 2 6 7 9	Selected activities: (1, 2) (3, 4) (5, 7) (8, 9)

White List:

Black List:

Question 5:

Implements a basic program for searching a pattern . This program checks every possible position in the text where the pattern could start and then verifies if the pattern indeed matches the text starting from that position.

Input Format:

- The first line contains the text T.
- The second line contains the pattern P.

Output Format:

- For each occurrence of the pattern P within the text T, print a line stating "Pattern found at index i", where i is the 0-based index in T where the pattern P starts. If the pattern is not found in the text, your program should not output anything.

Title for Question 5: Naive Algorithm

Solution:

```
#include <bits/stdc++.h>
using namespace std;

void search(string pat, string txt)
```

```
{
 int M = pat.size();
 int N = txt.size();

 for (int i = 0; i <= N - M; i++) {
  int j;

  for (j = 0; j < M; j++)
   if (txt[i + j] != pat[j])
    break;

  if (j == M)
   cout << "Pattern found at index " << i << endl;
 }
}

int main()
{
 string txt,pat;
    cin>>txt>>pat;
 search(pat, txt);
 return 0;
}
```

TestCases:

S.No	Inputs	Outputs
1	AABAACAADAABAAABAA AABA	Pattern found at index 0 Pattern found at index 9 Pattern found at index 13
2	aaaaaa aa	Pattern found at index 0 Pattern found at index 1 Pattern found at index 2 Pattern found at index 3 Pattern found at index 4
3	banana nan	Pattern found at index 2
4	mississippi iss	Pattern found at index 1 Pattern found at index 4
5	abracadabra abra	Pattern found at index 0 Pattern found at index 7
6	programming gram	Pattern found at index 3

White List:

Black List:

Question 6:

Implement the brute-force pattern matching algorithm. Your solution should accurately identify occurrences of a given pattern within a provided text. Refine the initial implementation to ensure the correct detection of pattern occurrences. Develop with diverse inputs to verify the functionality and accuracy of your implemented algorithm. Additionally, analyze the time complexity of the brute-force approach and its significance in the context of pattern matching tasks involving extensive texts and patterns.

Input Format:

- The text to search for the pattern, entered as a string.
- The pattern to be searched within the text, entered as a string.

Output Format:

- If the pattern is found in the text, it prints the positions (indices) where the pattern occurs, separated by commas.
- If the pattern is not found in the text, it prints "Pattern not found in the text."

Title for Question 6: Implementing a Brute-Force Pattern Matching Algorithm

Solution:

```
#include <iostream>
#include <string>
#include <vector>


using namespace std;


// Function to perform pattern matching using the brute-force approach
vector<int> bruteForcePatternMatch(const string& text, const string& patt
    vector<int> matches;


    int n = text.length();
    int m = pattern.length();


    for (int i = 0; i <= n - m; ++i) {
        bool found = true;
        for (int j = 0; j < m; ++j) {
            if (text[i + j] != pattern[j]) {
                found = false;
                break;
            }
        }
        if (found) {
            matches.push_back(i);
        }
    }


    return matches;
}


int main() {
    // Input text and pattern
    string text, pattern;
    getline(cin, text);
    getline(cin, pattern);


    // Perform pattern matching using the brute-force approach
    vector<int> matches = bruteForcePatternMatch(text, pattern);


    // Output the matches
    if (matches.empty()) {
        cout << "Pattern not found in the text." << endl;
    } else {
        cout << "Pattern found at positions: ";
        for (int i = 0; i < matches.size(); ++i) {
            cout << matches[i];
            if (i < matches.size() - 1) {
                cout << ", ";
            }
        }
        cout << endl;
    }


    return 0;
}
```

TestCases:

S.No	Inputs	Outputs
1	The quick brown fox jumps over the lazy dog fox	Pattern found at positions: 16
2	banana zoo	Pattern not found in the text.
3	ababababab aba	Pattern found at positions: 0, 2, 4, 6
4	mississippi issi	Pattern found at positions: 1, 4
5	abcdabcdabcdabcd abcd	Pattern found at positions: 0, 4, 8, 12
6	programming is fun bcd	Pattern not found in the text.

White List:

Black List: