Day 6 / 100 :

Topic - Strings

1 Problem statement: Minimum distance between words (easy)

Given a string s and two words w1 and w2 that are present in S. The task is to find the minimum distance between w1 and w2. Here, distance is the number of steps or words between the first and the second word.

Examples:
Input : s = “geeks for geeks contribute practice”, w1 = “geeks”, w2 = “practice”
Output : 1
There is only one word between the closest occurrences of w1 and w2.

Input : s = “the quick the brown quick brown the frog”, w1 = “quick”, w2 = “frog”
Output : 2

Solutions :

Approach 1 - brute force

The brute force approach involves iterating through the string and keeping track of the positions of both words. We initialize the minimum distance as a large value and update it whenever we find a pair of occurrences that have a smaller distance.

```
int findMinDistance(const std::string& s, const std::string& w1, const
std::string& w2) {
    std::vector<int> positions_w1;
    std::vector<int> positions_w2;

    // Find positions of both words in the string
    for (int i = 0; i < s.length(); ++i) {
        if (s.substr(i, w1.length()) == w1)
            positions_w1.push_back(i);
        else if (s.substr(i, w2.length()) == w2)
            positions_w2.push_back(i);
    }
```

```
    int minDistance = INT_MAX;
    // Calculate the minimum distance
    for (int i = 0; i < positions_w1.size(); ++i) {
        for (int j = 0; j < positions_w2.size(); ++j) {
            int distance = abs(positions_w1[i] - positions_w2[j]) - 1;
            if (distance < minDistance)
                minDistance = distance;
        }
    }

    return minDistance;
}

int main() {
    std::string s = "geeks for geeks contribute practice";
    std::string w1 = "geeks";
    std::string w2 = "practice";

    int minDistance = findMinDistance(s, w1, w2);
    std::cout << "Minimum distance: " << minDistance << std::endl;

    return 0;
}
```

The brute force approach has a time complexity of O(n^2), where n is the length of the string. This is because we iterate through all the positions of both words to find the minimum distance.

Solutions :

Approach 2 - Efficient Approach

The efficient approach involves iterating through the string only once and updating the minimum distance whenever we encounter the words w1 or w2. We maintain two variables to keep track of the positions of w1 and w2.

```
int findMinDistance(const std::string& s, const std::string& w1,
const std::string& w2) {
    std::vector<int> positions;
    int minDistance = INT_MAX;

    int pos1 = -1;
```

```
    int pos2 = -1;

    for (int i = 0; i < s.length(); ++i) {
        if (s.substr(i, w1.length()) == w1) {
            pos1 = i;
            if (pos2 != -1)
                minDistance = std::min(minDistance, pos1 - pos2 - 1);
        } else if (s.substr(i, w2.length()) == w2) {
            pos2 = i;
            if (pos1 != -1)
                minDistance = std::min(minDistance, pos2 - pos1 - 1);
        }
    }

    return minDistance;
}
```

1. The efficient approach also starts with the findMinDistance function that takes the same parameters as the brute force approach.
2. In this approach, we eliminate the need for storing positions in separate vectors. Instead, we use two variables pos1 and pos2 to keep track of the positions of w1 and w2.
3. We initialize both variables to -1, indicating that the words haven't been found yet.
4. We then iterate through the input string s using a for loop, checking if substrings match with w1 or w2. If a match is found, we update the corresponding position variable.
5. Inside each if condition, we also check if the other position variable is valid (i.e., not -1). If both position variables are valid, we calculate the distance as the difference between the positions minus 1 (since the distance is the number of words between the first and second word).
6. If the calculated distance is smaller than the current minimum distance, we update the minDistance variable.

2 Problem statement: [Maximum Number of Events That Can Be Attended](#) (Medium)

You are given an array of events where events[i] = [startDayi, endDayi]. Every event i starts at startDayi and ends at endDayi.
You can attend an event i at any day d where startTimei <= d <= endTimei. You can only attend one event at any time d.
Return the maximum number of events you can attend.

Example 1:
Input: events = [[1,2],[2,3],[3,4]]
Output: 3
Explanation: You can attend all the three events.
One way to attend them all is as shown.
Attend the first event on day 1.
Attend the second event on day 2.
Attend the third event on day 3.
Example 2:
Input: events= [[1,2],[2,3],[3,4],[1,2]]
Output: 4

Solutions :

Approach 1 - Brute Force

Sure! I'll provide you with both a brute force approach and an efficient approach to solve this problem in C++. Let's start with the brute force approach.

Brute Force Approach:
The brute force approach involves trying out all possible combinations of events and finding the maximum number of events you can attend. Here are the steps:

1. Sort the events array in ascending order based on the end time of the events. This step ensures that we prioritize events with earlier end times.
2. Initialize a set or an array to keep track of the days when events are attended.
3. Iterate through each event in the sorted array.
4. For each event, iterate from its start day to end day.

5. Check if the current day is already occupied by attending another event. If not, mark the current day as attended and increment the count of attended events.
6. Continue to the next event and repeat steps 4-5.
7. Finally, return the count of attended events.

```
int maxEvents(vector<vector<int>>& events) {
    sort(events.begin(), events.end(), [](const vector<int>& a, const
vector<int>& b) {
        return a[1] < b[1]; // Sort events based on end time
    });

    vector<bool> attended(100001, false); // Track attended days
    int count = 0;

    for (const vector<int>& event : events) {
        for (int day = event[0]; day <= event[1]; ++day) {
            if (!attended[day]) {
                attended[day] = true;
                ++count;
                break;
            }
        }
    }

    return count;
}
```

The time complexity of this brute force approach is O(N^2), where N is the number of events. The sorting step takes O(N log N) time, and for each event, we may iterate through O(N) days in the worst case.

Solutions :

Approach 2 - Brute Force

The brute force approach can be optimized using a priority queue or a min-heap. Here are the steps for the efficient approach:

1. Sort the events array in ascending order based on the start time of the events. This step ensures that we prioritize events with earlier start times.
2. Initialize a priority queue or min-heap to store the end days of the attended events in ascending order.
3. Initialize a variable to keep track of the current day.
4. Iterate through each event in the sorted array.
5. Check if the current day is less than the start day of the current event. If so, increment the current day until it reaches the start day of the event.
6. Push the end day of the current event into the priority queue.
7. While the priority queue is not empty and the top element (earliest end day) is less than or equal to the current day, remove that element from the queue.
8. Increment the count of attended events.
9. Continue to the next event and repeat steps 5-8.
10. Finally, return the count of attended events.

```
int maxEvents(vector<vector<int>>& events) {
    sort(events.begin(), events.end()); // Sort events based on start
time

    priority_queue<int, vector<int>, greater<int>> pq; // Min-heap
for end days
    int count = 0;
    int currentDay = 0;
    int n = events.size();
    int i = 0;

    while (i < n || !pq.empty()) {
        if (pq.empty()) {
            currentDay = events[i][0];
        }

        while (i < n && events[i][0] <= currentDay) {
            pq.push(events[i][1]);
            ++i;
        }

        if (!pq.empty()) {
            pq.pop();
            ++count;
```

```
            ++currentDay;
        }
    }

    return count;
}
```