To create a C++ program for the "Bit String" problem, we need to implement a dynamic programming solution to find the maximum "quality degradation value" for given bit strings. The problem can be modeled as finding the longest path in a Directed Acyclic Graph (DAG) where nodes are bit strings and edges represent degradation steps.

**Problem Analysis and Approach**

**Goal:** For a given bit string, find the maximum sum of weights (C values) accumulated during a degradation process until the string becomes all zeros.

**Degradation Rules:**

1. A '1' can be changed to '0'.
2. Two adjacent '1's ("11") can be changed to "00".
3. The process continues until the bit string is all '0's.
4. At each step, the

C value of the *current* bit string (pattern) is added to the total.

**Graph Representation:**

* **Nodes:** Each possible bit string of length N is a node. Since N is up to 20, there are 2^N (up to ~1 million) possible bit strings. We can represent these bit strings as unsigned int for efficient bitwise operations.
* **Edges:** A directed edge exists from bit string A to B if B can be obtained from A by one degradation step. Since '1's are always changed to '0's, the number of '1's always decreases or stays the same (if a 1 is changed to 0 and another 0 to 1 which is not allowed). The degradation process implies a directed acyclic graph (DAG).
* **Weights:** The weight associated with a node A is its C value (pattern\_weights[A]).

**Dynamic Programming (DP) Strategy:**

We will use memoization (top-down dynamic programming) to solve this longest path problem on a DAG. Let dp[bit\_string\_as\_int] store the maximum accumulated C value from bit\_string\_as\_int until it reaches the all-zero string.

**calculate\_degradation(current\_int\_bitstring) Function:**

1. **Memoization Check:** If the result for current\_int\_bitstring is already in the memo table, return it.
2. **Base Case:** If current\_int\_bitstring is 0 (representing the all-zero bit string), the degradation process ends. The

C value for the all-zero pattern is always 0. So, return 0.

1. **Recursive Step:**
   * Get current\_pattern\_weight by looking up the C value for current\_int\_bitstring.
   * Initialize max\_next\_degradation to LLONG\_MIN (a very small negative number). This is important because accumulated sums can be negative.
   * **Explore Single '1' to '0' Changes:** Iterate through each bit position i from 0 to N-1. If the i-th bit of current\_int\_bitstring is '1', flip it to '0' to get next\_int\_bitstring. Recursively call calculate\_degradation(next\_int\_bitstring) and update max\_next\_degradation with the maximum value found.
   * **Explore "11" to "00" Changes:** Iterate through adjacent bit positions i and i+1 from 0 to N-2. If both bits are '1', flip them both to '0' to get next\_int\_bitstring. Recursively call calculate\_degradation(next\_int\_bitstring) and update max\_next\_degradation.
   * **Compute and Store Result:** The maximum degradation from current\_int\_bitstring is current\_pattern\_weight + max\_next\_degradation. Store this in memo[current\_int\_bitstring] and return it.

**Data Structures:**

* N\_len: An integer to store the length of bit strings.
* pattern\_weights: A std::map<unsigned int, long long> (or std::unordered\_map) to store the C value for each bit string pattern. The key is the unsigned int representation of the bit string.
* memo: A std::map<unsigned int, long long> (or std::unordered\_map) to store computed dp values for memoization.

**Input/Output:**

* Read

N and Q from the first line.

* Read

2^N pattern strings and their weights, storing them in pattern\_weights after converting string patterns to unsigned int.

* For each of the

Q query strings, convert it to unsigned int, call calculate\_degradation, and print the result on a new line.

**Complexity:**

* **Time:** The calculate\_degradation function visits each of the 2^N states once. For each state, it performs O(N) operations (iterating through bits). Thus, the total time complexity for DP is O(N \* 2^N). For

N=20, this is approximately 20 \* 2^20 = 20 \* 1,048,576 ≈ 2.1 \* 10^7 operations, which is efficient enough for a 1-second time limit.

* **Space:** The pattern\_weights and memo maps each store 2^N entries. For N=20, this means 2 \* 2^20 entries. Each entry stores an unsigned int key and a long long value. This results in memory usage of approximately

2 \* 1MB \* (4 + 8) bytes = 24 MB, well within the 512 MB memory limit.