Let us define m variables which denotes the number of times each button was toggled.
Variables:
$C = C_1, C_2 ... C_j, ..., C_m$, where C_j is a non-negative integer.

Light i stays lit if it was toggled odd number of times. Hence, for each of the i light bulbs, we have the following constraint (total n constraints):

Factors: $f_1, f_2, ... f_i, ..., f_n$ with each $f_i = 1$

where each factor/constraint is:

$$f_i(C) = [(\sum_k^m C_k * \mathbb{1}[i \epsilon T_k])\%2 = 1]$$

Though we sum over all buttons, the indicator function is essentially ensuring that the summation is only performed over a subset of buttons where for each of these buttons k, $i \epsilon T_k$. Hence, we can also write the same as:

$$f_i(C) = [(\sum_{k;\text{ such that } i \epsilon T_k}^m C_k)\%2 = 1]$$

Let this subset by denoted by S_i. Then S_i is the scope of f_i and $|S_i|$ is its arity.

Also, note that the real count of how many times a button was toggled does not matter. What matters is whether the button was toggled odd or even number of times which can be used as an additional optional info to aid solving the CST. In that case, the variables will just be binary valued (0 or 1).

i) Checking all possible assignments:

X_1	X_2	X_3	$t_1 = (X_1 \oplus X_2)$	$t2 = (X_2 \oplus X_3)$	$W = t1 * t2$
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	1	1	**1**
0	1	1	1	0	0
1	0	0	1	0	0
1	0	1	1	1	**1**
1	1	0	0	1	0
1	1	1	0	0	0

The product of the factors is 1 for 2 assignments. Hence, there are two consistent assignments mentioned above.
1. $X_1 = 0, X_2 = 1, X_3 = 0$
2. $X_1 = 1, X_2 = 0, X_3 = 1$

ii) Call stack:

- backtrack({}, 1, {x1:{0, 1}, x2:{0, 1}, x3: {0, 1}})
  - backtrack({x1:0}, 1, {x2:{0, 1}, x3: {0, 1}})
    - backtrack({x1:0, x3:0}, 1, {x2:{0, 1}})
      - backtrack({x1:0, x3:0, x2:1}, 1, {})
    - backtrack({x1:0, x3:1}, 1, {x2:{0, 1}})
  - backtrack({x1:1}, 1, {x2:{0, 1}, x3: {0, 1}})
    - backtrack({x1:1, x3:0}, 1, {x2:{0, 1}})
    - backtrack({x1:1, x3:1}, 1, {x2:{0, 1}})
      - backtrack({x1:1, x3:1, x2:0}, 1, {})

Total number of calls = 9

iii) Call stack:

- backtrack({}, 1, {x1:{0, 1}, x2:{0, 1}, x3: {0, 1}})
  - backtrack({x1:0}, 1, {x2:{1}, x3: {0}})
    - backtrack({x1:0, x3:0}, 1, {x2:{1}})
      - backtrack({x1:0, x3:0, x2:1}, 1, {})
  - backtrack({x1:1}, 1, {x2:{0}, x3: {1}})
    - backtrack({x1:1, x3:1}, 1, {x2:{0}})
      - backtrack({x1:1, x3:1, x2:0}, 1, {})

Total number of calls = 7.

Let us take few auxiliary variables S_1, S_2 and S_3 such that S_i is supposed to hold the pair denoting the sum till $i-1$ elements and i elements.

Hence, $S_i[1] = S_i[0] + X_i$
Also, $S_i[0] = S_{i-1}[1]$

Also, $S_1[0] = 0$ since it is the sum of 0 elements by definition.

To summarize,
Auxiliary Variables: S_1, S_2, S_3

Unary Factors:
– $S_1[0] = 0$
– $S_3[1] \leq K$

Binary Factors:
– $S_2[0] = S_1[1]$
– $S_3[0] = S_2[1]$
– $S_1[1] = S_1[0] + X_1$
– $S_2[1] = S_2[0] + X_2$
– $S_3[1] = S_3[0] + X_3$

It is easy to verify this works since $S_i[1]$ is fetching the sum of first i elements and we restrict $S_3[1]$ to be $\leq K$. Further, we copy $S_{i-1}[1]$ to $S_i[0]$ so that all our factors are binary factors so as to prevent creation of 3-ary factor (between previous sum, current term and new sum).

Profile
minUnits 3
maxUnits 6

register Spr2020
register Sum2020
register Aut2020
register Win2020

taken CS124
taken CS229
taken CS107
taken CS103
taken CS106X
taken CS106B
taken CS210A

request CS399 # all (1, 9)
request CS221 # Aut,Sum (3,4)
request CS224N # Aut (3, 4)
request CS224S # Spr (2, 4) cannot fulfill due to dependency on cs221 and cs224n
request CS210B in Spr2020 after CS221 # Spr(3, 4) cannot fulfill due to dependency on cs221
request CS223A # Win (3)

The best schedule is:

Quarter	*Units*	*Course*
Spr2020	6	CS399
Sum2020	4	CS221
Aut2020	4	CS224N
Win2020	3	CS223A

Key things to note about this schedule are:
1. CS339 is chosen for first quarter with 6 credits.
2. CS224S and CS210B cannot be fulfilled because they depend on CS221 and CS221 can be taken earliest in Sum2020 but these two courses are only available in Spr2020 (Constraint added from Bulletin and request).
3. CS221 is scheduled in Sum2020 so that CS224N can be scheduled in Aut2020 otherwise the dependency could not have been satisfied.

Since the size of the factors may be as large as n, the tree-width will be at least n in such cases. Tree-width cannot be greater than n since that is the number of variables. Hence, tree-width is n in the worst case.

Let us reformulate this problem a bit to only pattern searches. We define two set of patterns first:
1. P containing the provided list of patterns that each have a weight of γ.
2. G containing the list of patterns with all repeated characters: {11, 22, 33, ..., KK}. Each of these have a weight of 5. These patterns mimic the provided g function.

We define R as the union of patterns from both P and G. Regarding the weight of each pattern in R:
1. A pattern present in both P and G will have weights coming from both of the original sets. Hence the total weight will be 5γ.
2. A pattern present in P, but not in G, will have a weight of γ.
3. A pattern present in G, but not in P, will have a weight of 5.

Now the question reduces to finding the weight of maximal weighted string where weight is defined as the product of weight of each pattern from R present in the string.

Let us first take a simpler problem. Given a string and a pattern, find all occurrences of the pattern in the given string (even overlapping ones)?
For a simple pattern search like this, KMP is well known and very efficient algorithm as it can process the target string in a single pass and maintains states iteratively (processing character by character) which as we will see is very useful for our case. In this case though, we have multiple patterns which makes KMP infeasible to use. But there is a simple extension to the KMP algorithm called Aho-Corasick algorithm that builds a trie to keep track of all patterns iteratively (processing character by character) in the tree. Updating the state during this iterative processing is $O(1)$.

Now in order to solve the origin problem, we first initialize Aho-corasick trie with all the patterns in R. Note that the number of nodes in the Aho-corasick tree will be $|R| * avg\_length\_of\_patterns$ which in worst case is: $K * 2 + |P| * n$. Also a state on this tree is just reference to one of these nodes on this tree. So there are $2K + n|P|$ states a given Aho-corasick trie can be in. Let this set of state space be denoted as $ACSpace$.

Solving the original problem is done using state search along with Dynamic Programming. Here, $MaxWeight[i][node\_index]$ denotes the maximum weight we can get with an string of length i and the final state of the Aho-corasick tree on node $node\_index$. Hence, this DP has a total of $n * (2K + n|P|)$ states.
Given a state $S[i][node\_index]$, we define edges out from it by trying to add a character to the Aho-corasick tree represented by $node\_index$. We try to add each of the characters in the domain, i.e., {1, 2, ..., K}. Hence, the algorithm is:

```
for i in {0, 1, 2, ...., n-1} {
    for node_index in ACSpace {
        for k in {1, 2, .... K} {
            next_index, delta_weight = AhoCorasickMove(aho_corasick_trie, node_index, k)
            next_weight = MaxWeight[i][node_index] * delta_weight
            if MaxWeight[i+1][next_index] < next_weight {
                MaxWeight[i+1][next_index] = next_weight
            }
        }
    }
}

best_weight = 0   # this is the answer
for node_index in ACSpace {
    best_weight = max(best_weight, MaxWeight[n][node_index]
}
```

Here:
1. Base Case: $i = 0$ denotes the state when the string is empty. So, if $zero\_index$ is the state of the trie when no character has been consumed (essentially the root of the trie), then $MaxWeight[0][node\_index]$ is initialized as 1 if $node\_index = zero\_index$ and 0 otherwise.
2. Since each edge increases the length of the string (i), the graph is a DAG and DP is applicable in this case.
3. $AhoCorasickMove(pattern\_trie, current\_state, next\_char)$ is assumed to be a function that takes the trie created from all the input patterns ($pattern\_trie$), the current state ($current\_state$), and moves it one step by adding character $next\_char$ to the trie state. It returns the new state on the trie, and the delta weight which is 1 if $next\_index$ does not denote the end of a pattern, or the corresponding weight from R if it denotes the end of a pattern. This weight can easily be stored in the tree itself for ease.

Time Complexity Analysis:
Precomputation: This is the cost of creating the trie. A standard implementation of Aho-corasick takes the sum of length of all patterns which is $O(2K + n|P|)$.
DP: The number of nodes in the tree as discussed before are $n * (2K + n|P|)$. There are K edges to evaluate on each of these nodes. Hence, time complexity = $O(K * n * (2K + n|P|)$.
Overall: Overall time complexity = sum of the two above = $O(2K + n|P| + nK(2K + n|P|) = O(n^2P + 2nK^2)$