Kick Start 2022 - Round B

Analysis: Unlock the Padlock

In the given problem, the range $[l_{i-1}, r_{i-1}]$ chosen in the $(i-1)$-th operation needs to be completely contained within the range $[l_i, r_i]$ chosen in the i-th operation; that is, $l_i \leq l_{i-1} \leq r_{i-1} \leq r_i$. The initial range ($[l_1, r_1]$) can be chosen arbitrarily. If we reverse the order of these operations, we would get the same final combination. For the remaining analysis, we would assume the reverse order of operations, that is, $l_{i-1} \leq l_i \leq r_i \leq r_{i-1}$. Whenever we say rotations in the analysis, it means downward by default unless mentioned otherwise.

Test set 1

In this test set, we can start from range $[1, \mathbf{N}]$ and greedily choose the side that requires lesser rotations and make it zero, and solve for the next number on that side. We should also keep track of the number of rotations done till now. Let the current range that we are trying to solve be $[i, j]$ and let the number of rotations modulo $\mathbf{D}$ be k. Then, the current value of index i would be $(\mathbf{V_i} - k + \mathbf{D}) \mod \mathbf{D}$. Similarly, the current value of index j would be $(\mathbf{V_j} - k + \mathbf{D}) \mod \mathbf{D}$. Let the current values of indexes i and j be $currentVal_i$ and $currentVal_j$ respectively. The number of rotations required to make dial at index i zero would be $\min(currentVal_i, \mathbf{D} - currentVal_i)$. The number of rotations required to make dial at index j zero would be $\min(currentVal_j, \mathbf{D} - currentVal_j)$. If index i requires less number of rotations, we then add $\min(currentVal_i, \mathbf{D} - currentVal_i)$ to the answer and solve for range $[i+1, j]$. Otherwise, if index j requires less number of rotations, we then add $\min(currentVal_j, \mathbf{D} - currentVal_j)$ to the answer and solve for range $[i, j-1]$. If the minimum number of rotations done are 0, k remains same for the subsequent range. Otherwise, we do $k = (k+1) mod \mathbf{D}$ as we would need to do 1 rotation. The transition is explained below in detail.

Why would the greedy approach work? Let $solve(i, j, k)$ denote the number of operations required to make all the elements in the range $[i, j]$ zero given that k rotations have happened. When we are solving for range $[i, j]$, there are 4 possible cases that could occur:

- $currentVal_i = 0, currentVal_j = 0$. In this case, we can reduce the range from both the sides. Hence, $solve(i, j, k) = solve(i+1, j-1, k)$.
- $currentVal_i = 1, currentVal_j = 1$. In this case, we can reduce the range from both the sides. Hence, $solve(i, j, k) = 1 + solve(i+1, j-1, (k+1) mod \mathbf{D})$.
- $currentVal_i = 1, currentVal_j = 0$. In this case, it is optimal to reduce the range from the right side because the value of index j is already zero and we would not require any further operations. If we choose to convert index i to zero, we would need one operation to convert the value of index i to zero. And this would change the value of index j to 1 which may require one more operation to convert it to zero. So, we might end up taking at least 1 more operation if we choose to make index i to 0. So, it is optimal to exclude the index j. Hence, $solve(i, j, k) = solve(i, j-1, k)$.
- $currentVal_i = 0, currentVal_j = 1$. Similar explanation as above, it is optimal to exclude the index i. Hence, $solve(i, j, k) = solve(i+1, j, k)$.

At each step, we perform constant operations to check which index requires less number of rotations and then make that element zero. Hence, it would take $O(\mathbf{N})$ time complexity to make all the elements zero.

Test set 2

In this test set, the greedy approach would not work because the number of digits is more. We can use dynamic programming to solve this test set. Let $dp(i, j, k)$ denote the minimum number of operations required to make all elements from i to j equal to 0 and k be the number of rotations modulo $\mathbf{D}$ that have been done till now. Then, $currentVal_i = (\mathbf{V_i} - k + \mathbf{D}) \mod \mathbf{D}$. Similarly, $currentVal_j = (\mathbf{V_j} - k + \mathbf{D}) \mod \mathbf{D}$. We then solve for both the cases.

- For converting index i to 0, we could do one of the following:
  - Rotate downwards, it would take $currentVal_i$ operations. We then recursively solve for $dp(i+1, j, (k + currentVal_i) \mod \mathbf{D})$.
  - Rotate upwards, it would take $\mathbf{D}$ - $currentVal_i$ operations. We then recursively solve for $dp(i+1, j, (k - (\mathbf{D} - currentVal_i) + \mathbf{D}) \mod \mathbf{D})$.

  Let $val_1 = \min(currentVal_i + dp(i+1, j, (k + currentVal_i) \mod \mathbf{D}), \mathbf{D} - currentVal_i + dp(i+1, j, (k - (\mathbf{D} - currentVal_i) + \mathbf{D}) \mod \mathbf{D}$
- For converting index j to 0, we could do one of the following:
  - Rotate downwards, it would take $currentVal_j$ operations. We then recursively solve for $dp(i, j-1, (k + currentVal_j) \mod \mathbf{D})$.
  - Rotate upwards, it would take $\mathbf{D}$ - $currentVal_j$ operations. We then recursively solve for $dp(i, j-1, (k - (\mathbf{D} - currentVal_j) + \mathbf{D}) \mod \mathbf{D})$.

  Let $val_2 = \min(currentVal_j + dp(i, j-1, (k + currentVal_j) \mod \mathbf{D}), \mathbf{D} - currentVal_j + dp(i, j-1, (k - (\mathbf{D} - currentVal_j) + \mathbf{D}) \mod \mathbf{D}$

Finally, $dp(i, j, k) = \min(val_1, val_2)$. Please take a look at the pseudocode below:

Sample Code(C++)

```
int solve(int i, int j, int k) {
  if(i > j) {
    return 0;
  }
  if(dp[i][j][k] != -1) {
    return dp[i][j][k];
  }
  int currentVal_i = (V[i] - k + D) % D;
  int currentVal_j = (V[j] - k + D) % D;
  // Convert index i to 0.
  int val_1 = min(currentVal_i + solve(i+1, j, (k + currentVal_i) % D), D - currentVal_i + solve(i+1, j, (k
  // Convert index j to 0.
  int val_1 = min(currentVal_j + solve(i, j-1, (k + currentVal_j) % D), D - currentVal_j + solve(i, j-1, (k
  return dp[i][j][k] = min(val_1, val_2);
}
```

We can get the final answer from $dp(1, \mathbf{N}, 0)$. There are a total of $O(\mathbf{N}^2 \cdot \mathbf{D})$ number of states. And we perform constant operations for each state. Hence, the overall complexity of the solution is $O(\mathbf{N}^2 \cdot \mathbf{D})$.

Test set 3

In this test set, $\mathbf{D}$ is very large, so the solution for the previous test set would time out. An interesting observation is that we do not need to keep track of the number of rotations when solving for a range $[i, j]$. If we keep track of which side (left or right) was made zero in the previous operation, we can calculate the number of rotations modulo $\mathbf{D}$ that has been done till now. The last element that becomes zero will be because of the combination of all the rotations done so far. So, let us say initial value of the element that was made zero in the previous operation was x. Then, total rotations applied so far have combinely made it zero. Then all numbers inside the range $[i, j]$ are affected by x downward rotations.

We can then solve the current test set by having $dp(i, j, bit)$ denoting the minimum number of operations required to make all elements from i to j equal to 0 given that the element that was made zero in the previous operation is on the left side if $bit = 0$, and is on the right side if $bit = 1$.

Let the number of rotations done till now modulo $\mathbf{D}$ be k. The value of k can be determined as follows:

- If $bit = 0$:
  - If $i > 1$, then $k = \mathbf{V_{i-1}}$.
  - Otherwise, $k = 0$.
- If $bit = 1$:
  - If $j < \mathbf{N}$, then $k = \mathbf{V_{j+1}}$.
  - Otherwise, $k = 0$.

Then $currentVal_i = (\mathbf{V_i} + k) \ mod \ \mathbf{D}$ and $currentVal_j = (\mathbf{V_j} + k) \ mod \ \mathbf{D}$. For a particular state $dp(i, j, bit)$, we would transition to the next state as follows.

- For converting index i to 0, it would take $\min(currentVal_i, \mathbf{D} - currentVal_i)$ operations. Let $currOperations = \min(currentVal_i, \mathbf{D} - currentVal_i)$. We then recursively solve for $dp(i + 1, j, 0)$. Let $val_1 = dp(i + 1, j, 0) + currOperations$.
- For converting index j to 0, it would take $\min(currentVal_j, \mathbf{D} - currentVal_j)$ operations. Let $currOperations = \min(currentVal_j, \mathbf{D} - currentVal_j)$. We then recursively solve for $dp(i, j - 1, 1)$. Let $val_2 = dp(i, j - 1, 1) + currOperations$.

Finally, $dp(i, j, bit) = \min(val_1, val_2)$. Please take a look at the pseudocode below:

Sample Code(C++)

```
int solve(int i, int j, bool bit) {
  if(i > j) {
    return 0;
  }
  if(dp[i][j][bit] != -1) {
    return dp[i][j][bit];
  }
  int k = 0;
  if(!bit) {
    if(i > 1) {
      k = V[i-1];
    }
  }
  else {
    if(j < N ) {
       k = V[j+1];
    }
  }
  int currentVal_i = (V[i] - (D - k) + D) % D;
  int currentVal_j = (V[j] - (D - k) + D) % D;
  // Convert index i to 0.
  int currOperations = min(currentVal_i, D - currentVal_i);
  int val_1 = currOperations + solve(i+1, j, 0);
  // Convert index j to 0.
  currOperations = min(currentVal_j, D - currentVal_j);
  int val_2 = currOperations + solve(i, j - 1, 1);
  return dp[i][j][bit] = min(val_1, val_2);
}
```

We can get the final answer from $dp(1, \mathbf{N}, 0)$. There are a total of $O(\mathbf{N}^2)$ number of states. And we perform constant operations for each state. Hence, the overall complexity of the solution is $O(\mathbf{N}^2)$.
Note that, if we solve the dynamic programming solution mentioned in Test Set 2 recursively, we will only visit $O(\mathbf{N}^2)$ states because the number of rotations from left side and right side will always be fixed. Hence, the recursive version of Test Set 2 solution where we keep track of only the states that we visit would pass this test set.