**Design decision for Toeplitz matrix filling**

The first step would be generating (2n-1) elements, that is the number of elements required to fill the Toeplitz matrix.

Possible approaches to fill up the Toeplitz matrix.

1. **Fill the upper triangular matrix first with the n elements and then fill the lower part of that matrix with remaning (n-1) elements**

Structure: 2x2 Matrix

Explanation: First n elements will be filled and shifted to right, creating upper triangular part.

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 0 | 1 | 1 |
| \* | 1 | 0 | 1 |
| \* | \* | 1 | 0 |
| \* | \* | \* | 1 |

The \* are the remaining n-1 elements that will be filled once upper triangular part is done.

Asymptotic time complexity: O(n2)

Actual time complexity = t(fill upper triangular matrix) \* t(lower triangular matrix)

= O(n2) \* O(n2) = O(n4).

This is the first implementation of Toeplitz matrix, it is taking huge time.

1. Create a temporary queue like structure using array and replace the end elements.
2. Instead of a 2X2 matrix, use a single array containing 2n-1 elements and perform the operation.

**Choice of Data Structures:**

The current implementation will be based on integer array. The advantage of using integer array is the code is simpler to understand and implement.

The disadvantage though, outweigh the advantage since, integer takes roughly 8 bits, and we use it to represent a bit.

Calculate():

* The method computes multiplication of packed key with the input. Since, this is an array implementation, the bits can be accessed using the index of the array.
* temp\_key is the packed secret key. We know that computation of Z1 will require K11, K21, K31, K41,K51, K61, K71, K81 and last bits of X1 which is right rotated.
* Since we are using the array implementation, we don’t need to actually shift the bits, instead the bits of X1 can be accessed by their index.

Solving the Word package example:

According to the document shared by you last week about word packing, the packing takes place as follows.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 |
| 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |
| 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |

The yellow highlighted shows the K1,1, K2,1,….K8,1.

Let’s say the input is 10110110, which makes x1=1011 and x2 = 0110.

According to the formula, shared in word packing document, we get the following computation

Z1 = 1\*(1100) + 1\*(0110) + 0\*(1011) + 1\*(0101) + 0\*(1010) + 1\*(0101) + 1\*(1010) + 0\*(0101)

= 1100 + 0110 + 0101 + 0101 + 1010

= 2422

Z2 = 1\*(1100) + 1\*(0110) + 0\*(0011) + 1\*(1001) + 0\*(1100) + 1\*(0110) + 1(1011) + 0\*(0101)

=1100 + 0110 + 1001 + 0110 + 1011

=3332

The output is divided into two parts. The output is 33243222 which is not as same as we get in the naïve implementation which is 33243233. The last two bits are wrong.

**Correction:**

I tried packing the numbers differently.

Take the same key matrix

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 |
| 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |
| 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |

This makes K11 = 1010, K21 = 1101 …. and K12 = 1010, K22 = 0101 ….

If we pack the numbers in this way, the computation is as follows:

Z1 = 1\*(1010) + 1\*(1101) + 0\*(0110) + 1\*(0011) + 0\*(1001) + 1\*(1100) + 1\*(0110) + 0\*(0011)

= 1010 + 1101 + 0011 + 1100 + 0110

= 3332

Z2 = 1\*(1010) + 1\*(0101) + 0\*(1010) + 1\*(0111) + 0\*(1010) + 1\*(1101) + 1(0110) + 0\*(0011)

=1010 + 0101 + 0111 + 1101 + 0110

=2433

nd is now correct and equivalent to the original answer. 33243233.