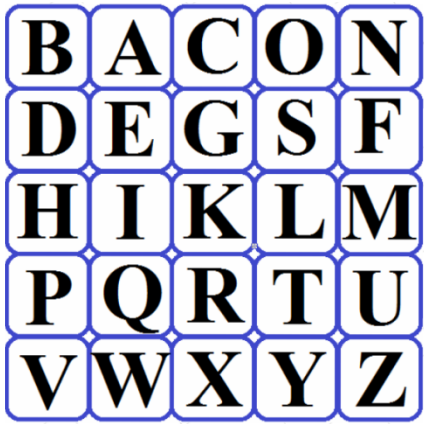
Rules for using the Playfair Matrix

- ### Constructing the Playfair matrix

The key could be any word or phrase. Removing spaces between the words of the phrase; it is then filled in the first row of the 5x5 matrix, with caution to avoid repetition. Once the keyword is filled, remaining letters of the English language script are filled lexicographically in the remaining slots of the matrix. Since the English alphabet contains twenty six alphabets, and a 5x5 grid contains only twenty five spaces, we must choose to conform to one of two conventions:

1. We omit "Q" from all our messages, given that it is statistically the least used alphabet in the English language.
2. We treat "I" and "J" equivalently in our encryption process and encrypt phrases like "TAJ MAHAL", as "TAI MAHAL" .

Throughout this paper, we have assumed the second convention, and have used the example key "BACON AND EGGS". The Playfair Matrix generated with this key, under this convention, looks something like this:



- ### Encoding messages using the Playfair matrix

The plaintext is first condensed into single string of letters by removing spaces and then dividing it into pairs. If last letter remains unpaired, which may happen if total number of letters in the string is odd, then the last letter is paired up with letter 'X' to even it out.

If there is a pair containing same letters (ex. 'LL') then this pair is broken by pairing the first letter with 'X' and pairing the other letter with the next letter of the string. For example, "BALLOONS" => "BALXLOXONS".

NOTE: Some people prefer to use "Q" as their filler alphabet, claiming that any encrypted message that is devoid of Q's is almost screaming, "I have been encrypted using a Playfair Matrix!!", but in actuality, whatever convention you choose to use is of little consequence as long as you pick one and stick with it throughout your encryption process.

Now, the manner in which a pair of alphabets is encoded using the Playfair Matrix, is dependent on their relative positions in the given Matrix. They will lie in one of the three following relative positions:

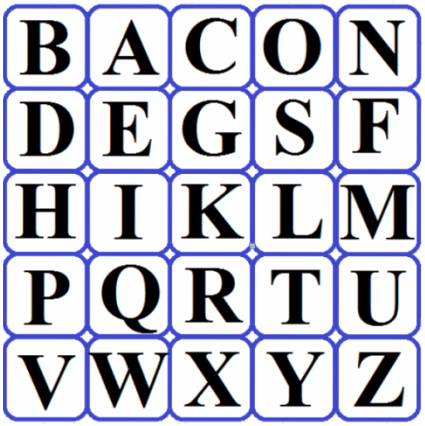
- They will be in the same row.
- They will be in the same column.
- They will be on two opposed corners of a smaller rectangle within the Matrix.

Let's take a sample Plaintext, and learn the rules to deal with each of the above mention scenarios through implementation in the Matrix generated from our key "BACON AND EGGS".

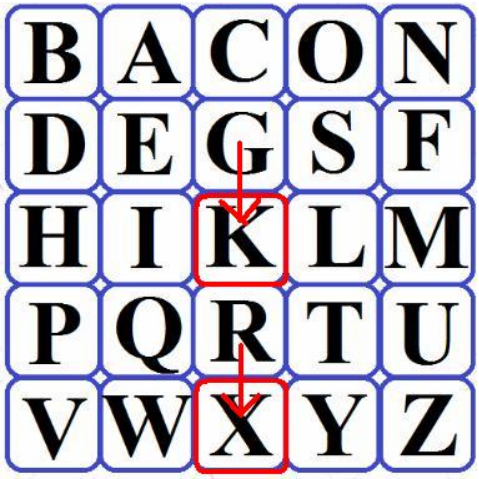
Suppose the plaintext is: **GROTESQUE**

Pairs: **GR – OT – ES – QU – EX**

Playfair Matrix:

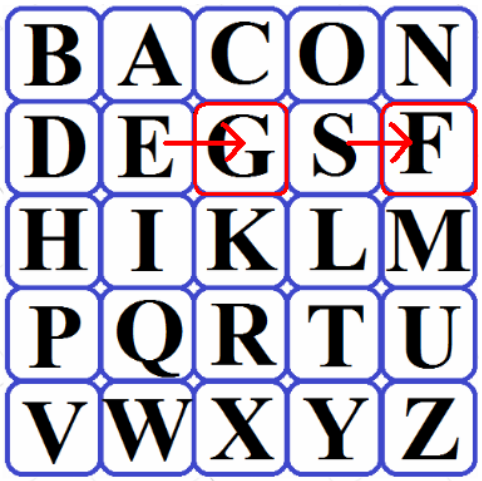


Encoding 'GR': It can be seen that in the above key, **G** and **R** lie in the same column. In such cases, we substitute each letter with the letter adjacent to it, down the column. So, **G** becomes **K** and **R** becomes **X** as shown:



Similarly **OT** would map onto **SY**. Also keep in mind however, that the order is of relevance. **TO** for instance would map onto **YS**.

Encoding 'ES': It can be seen that in the above key, **E** and **S** lie in the same row. In such cases, we substitute each letter with the letter adjacent to it in the right of the row. So, **E** becomes **G** and S becomes **F** as depicted below.

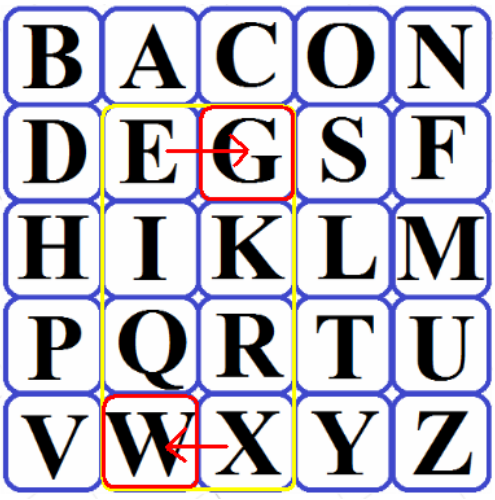


Similarly **QU** maps to **RP**. This brings up an interesting instance of what we do when one of the alphabets in a row pair falls in the right most column. Or equivalently, one of the alphabets on a column pair falls in the bottom most row.

The Playfair Matrix, is a wrap-around Matrix, and it very naturally facilitates these scenarios. What this means is, the bottom most row can be viewed as though it wraps around to join the top most row so that the continuity in the alphabets remains unbroken. The same can be said for the right most column. It can be viewed as though it wraps around to join with the left most column so that continuity in the map remains unbroken. So in the wrap around view of the above Playfair matrix, **P** is to the immediate right of **U**, and **A** is immediately below **W** and so on.

In the wrap around view of the Playfair matrix, one can view the entire grid to be a Toroidal (Doughnut like) surface, with an unbroken continuity of alphabets all around.

Encoding 'EX': E and X do not lie in the same row nor do they lie in the same column. In these situations, we view the smaller rectangle mapped out by these two corners and we substitute each alphabet with its sister corner in the SAME-ROW as itself such that the new pair lies on the opposite two corners of the rectangle formed. So, **E** becomes **G** and **X** becomes **W** as depicted below.



For the sake of a few more examples, the pair **UK** would map onto **RM** and the pair **PO** would map onto **TB.**

The Problem at Hand

This paper mainly addresses the question, “Given a certain length of Plain Text, with its corresponding Cipher Text (roughly 200 characters worth), is it possible to extract the Playfair Matrix that was used in the encryption process? If so, how would one systematically and algorithmically attack this problem? ”

The Solution

It is systematically possible to reconstruct the Playfair Matrix that was used in a given encryption, provided a sufficient amount of Plain Text and the corresponding Cipher Text that was generated using that matrix.

We begin with an empty grid of 25 squares and depending upon which convention we are following, we must find the relative placement of alphabets within it.

But before we find the exact positions of the alphabets within the matrix, we would like to reduce the problem into a simpler one by finding pieces or chunks of characters lie adjacent to each other within the matrix.

- ## Finding Triplets

We set out by examining the pairs of plain text and corresponding cipher text. The key idea is to find Plain text-Cipher text pairs, such that they share a common character. This idea is powerful because any such occurrence guarantees that the three DISTINCT characters of the four present must lie in a straight line i.e. they must lie in the same row or column due the very nature of the rules of Playfair encoding. We call this set of three characters that will lie in the same row/column a ‘Triplet’.

As an example consider the figure below where the original text is “Mutton pie with scrambled eggs”. We have illustrated the encoding of this message using our “Bacon and Eggs” matrix. The plain text and the cipher text have been broken into diagraphs and the pairs with common alphabets are marked in red text.

Take the first instance highlighted in red. Because “M” and “U” lie adjacent to each other in the Matrix, they are bound to encode to another pair of alphabets that are also adjacent in the Matrix along the same line as “M” and “U”; in this case, “U” and “Z”. And what is more, they are bound to share a common alphabet (in this case “U”). This is a tell-tale sign that the triplet piece “MUZ” exists in the matrix that was used to encrypt this message.

Original text - MUTTON PIE WITH SCRAMBLED EGGS

Plain text -	MU	TQ	TO	NP	IE	WI	TH	SC	RA	MB	LE	DE	GQ	GS
Cipher text -	UZ	UR	YS	BU	QI	AQ	PL	GO	QC	HN	IS	EG	ER	SF

DE = EG
Straight tripplet

B	A	C	O	N
D	E	G	S	F
H	I	K	L	M
P	Q	R	T	U
V	W	X	Y	Z

IE = QI
Skew Tripplet

EI = IQ

It must be noted that triplets can be of two types. For example: If "KL" maps on to "LM", we call it a Straight Triplet since the common character lies between two distinct characters and we know now that K,L and M will lie together in a row/column. But if we find that "LK" maps onto "ML" (essentially the same piece of information) we call it a Skew Triplet. In Skew Triplets, the common alphabet lies at the edges of the mapping.

Skew Triplets can be easily converted to Straight Triplets by a trivial lateral flip, so that it becomes obvious to us how the pieces must join to form a piece of three. After the lateral inversion, we find that "KL" maps onto "LM" and it becomes obvious to us that the Triplet "KLM" must exist in the matrix that was used in the encryption.

In the figure above we have a straight triplet formed by the Plain text-Cipher text pairs "DE" and "EG" respectively. This is a straight triplet which happens to lie in a row in the original Playfair matrix. Similarly we have a Skew triplet formed by the Plain text-Cipher text pairs "IE" and "QI", which can be flipped over and joined to give the triplet 'EIQ'. This triplet happens to lie in a column in the original matrix.

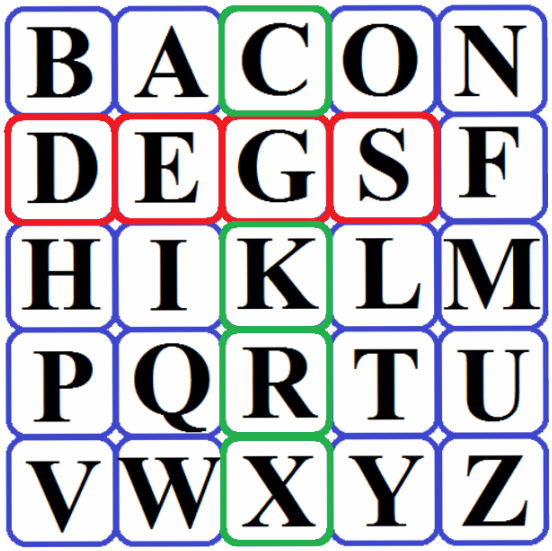
- ## Finding Quadruplets

After having generated a list of Triplets, we apply a similar idea to this generated list of Triplets as we did to the plain text-cipher text pairs. We scan the entire list of triplets to find pairs of triplets such that they have exactly two common characters between them.

This idea is just an extrapolation of the one applied to the diagraphs to find the triplets. It empowers us with the knowledge of four distinct characters that are placed linearly next to each other in the

matrix. These could lie together either in a row or a column. At this stage, there is no way of knowing. We call the set of these four characters a ‘Quadruplet’. We proceed to find all such quadruplets and thus create a list of quadruplets.

Triplets: EGS YOS DEG HIK ACO RXC OSL NBA KRX
Quadruplets: DEGS KRXC



In the figure above, the list of triplets has been stated and the ones with two common alphabets shared between them have been marked in the same colour text (red/green).

We can clearly see that ‘EGS’ and ‘DEG’ will form a quadruplet and so will the pair of ‘RXC’ and ‘KRX’.

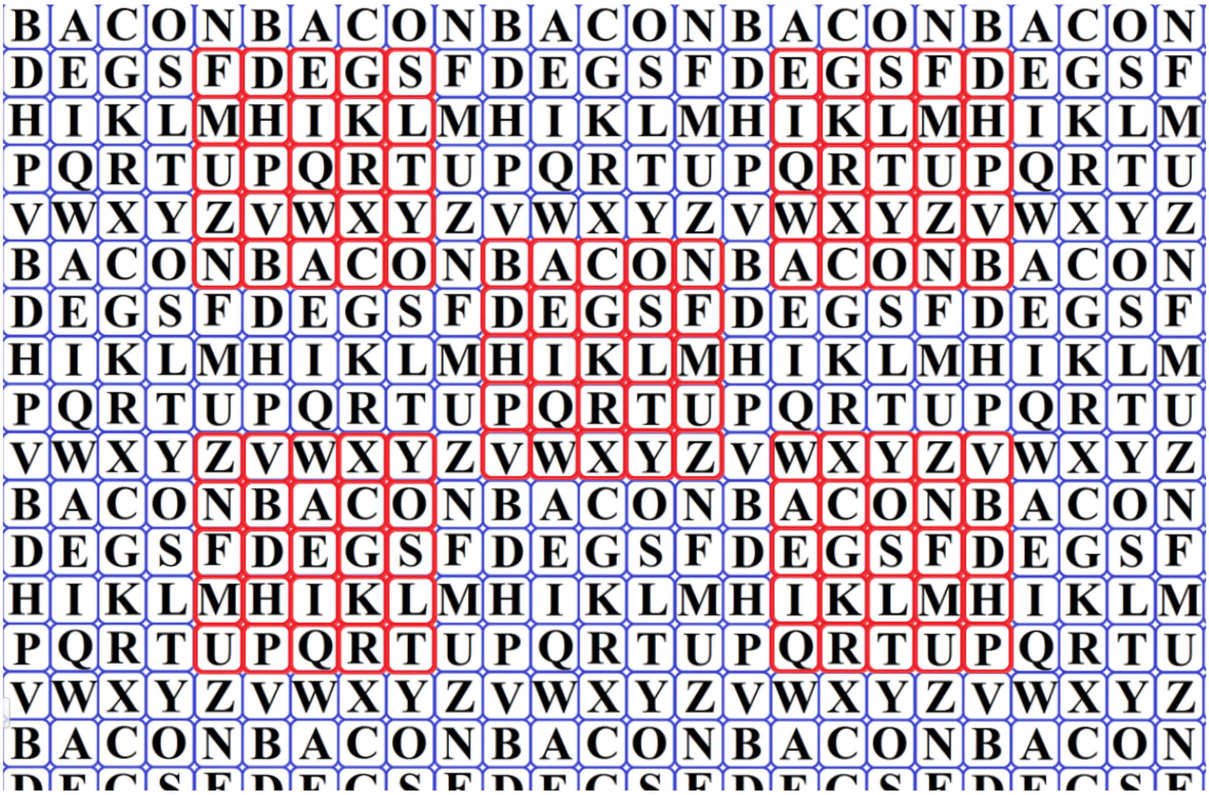
‘DEG’ and ‘EGS’ will join at ‘EG’ to from the quadruplet ‘DEGS’. And ‘RXC’ and ‘KRX’ will join at ‘RX’ to form the quadruplet ‘KRXC’.

And, in the figure of the original matrix, we can clearly see that ‘DEGS’ fits in to a row while ‘KRXC’ fits in to a column (subject of course, to wrap around equivalency).

- ## Wrap Around Equivalency

As we encountered in our example illustrating the rules of Playfair, one of the key properties of a Playfair Matrix, is that it is a Wrap-around Matrix. We went on to discuss the three dimensional visualization of the matrix as a toroidal shape, to facilitate the continuity of alphabets all around, in every direction.

But, there is a simpler way of visualizing the same property whilst restricting ourselves to two dimensions.



Instead of thinking of the Matrix as standing by itself in isolation, we can think of it as part of an infinite plane of similar matrices that surround it in every direction. This way, we facilitate the continuity of alphabets in every direction, without the annoyance of having to imagine the surface of the matrix bending to meet itself in a toroidal fashion.

Also, in this view it becomes immediately obvious how the mapping "IM = KH" is facilitated without 'M' having to wrap all the way around to meet 'H'. As seen in the above map, 'M' simply has to move to the 'H' on its right in the exact same way that 'I' simply has to move to the 'K' on its right, regardless of the confines of the original 5x5 matrix.

This observation however, shows us another very important consequence of the wrap around property of the Playfair matrix. It shows us that the "location" of the 5x5 grid that defines the borders of the matrix, is immaterial because all locations of the matrix plane are equivalent. What this means, in simple terms is that one could take a 5x5 stencil, and cut out any piece that they like from the infinite matrix plane (not necessarily rooted at the original key-phrase), and the resultant matrix would encode text in a manner identical to the original matrix.

Five instances have been illustrated by being highlighted in red in the above diagram of the matrix plane. The central one, is the original "Bacon and Eggs" matrix, anchored at "B", but around it we have illustrated four equivalent matrices that have been obtained by using the 5x5 stencil to pull out matrices anchored at "F", "E", "Z" and "W".

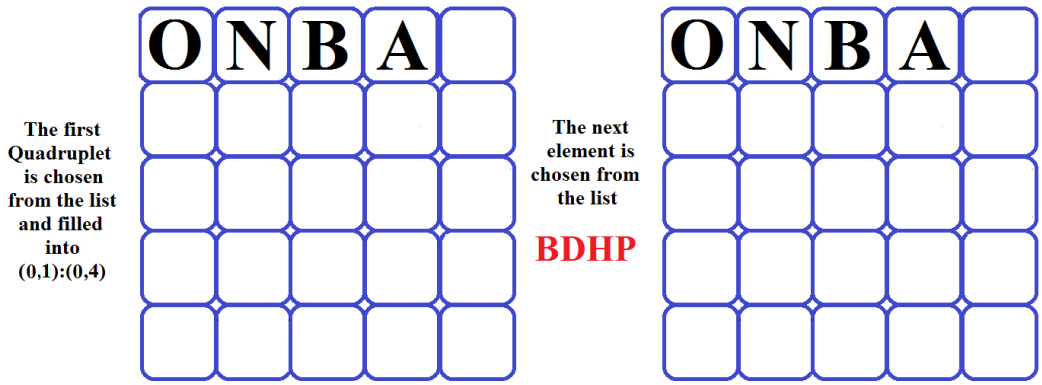
In fact, the top left alphabet, or the "Anchor Alphabet", can be any one of the 25 characters available on the matrix plane. This is why, they unique key-space of the Classical Playfair Matrix is actually 24! matrices large and not 25!, as one would expect.

- **Fill Matrix Function**

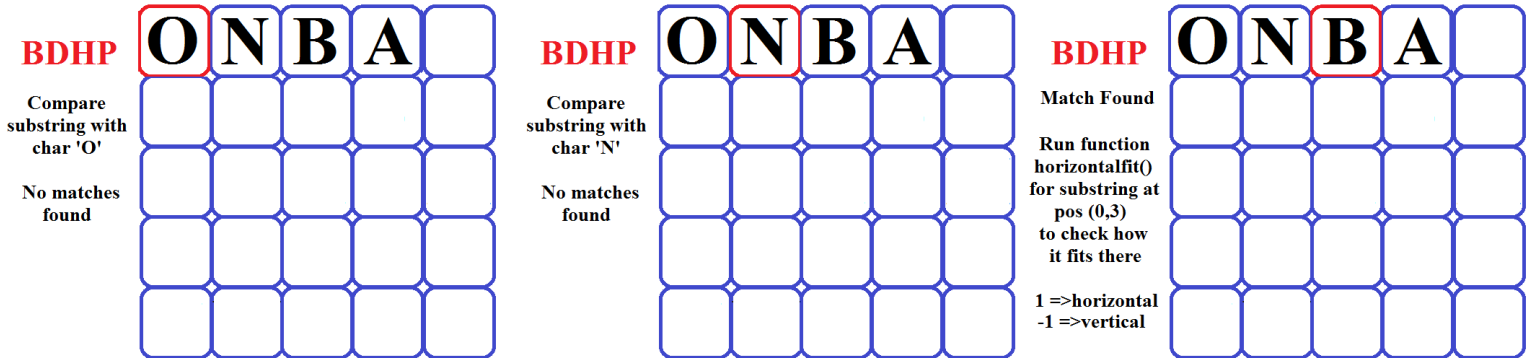
After we have acquired the distinct pieces of the matrix, in the form of triplets and quadruplets, we use the fill matrix function to piece these together to get closer to the construction of our final Playfair matrix.

Our realisation about the equivalency of Wrap around matrices makes our task of finding the Playfair matrix, computationally a whole lot simpler. Now we don't need to worry about which piece will fit in the top left corner, as long as the positions of all the other pieces, relative to that piece, are maintained. So let us now begin piecing together our matrix.

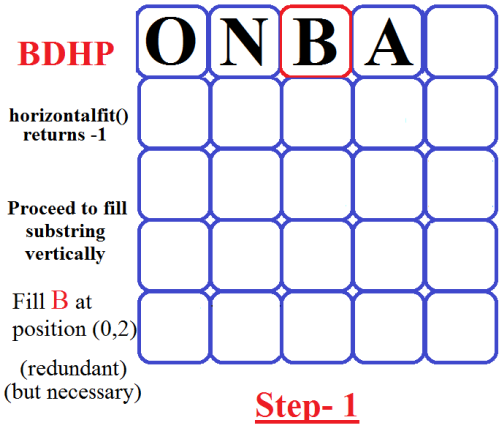
From the list of quadruplets/triplets, the first piece is chosen and fit into the first row of the matrix.



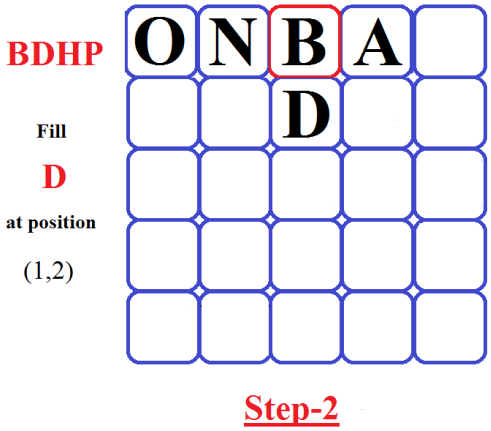
We then pick the next piece and try to fit it together with this element, into the matrix. In our example, ONBA was filled and the next element BDHP was chosen. We check the partially constructed matrix, element by element, until we hit a commonality.



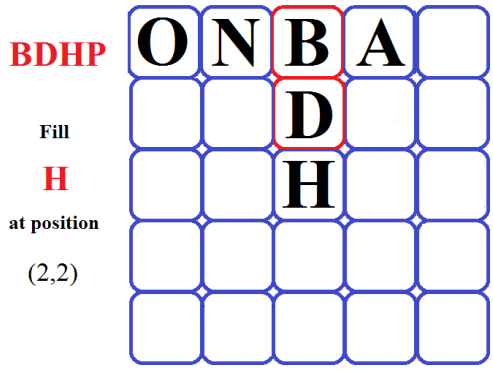
We see that they have a common character, i.e. 'B'. We then run our horizontal_fit function to check whether BDHP fits vertically or horizontally. If it fits horizontally, the function returns a +1, and if it fits vertically, the function returns a -1. In this case, the function returns a -1, and we observe that the fit is vertical, so by means of a loop, we fill in the element character by character until the piece has been fit into the matrix.



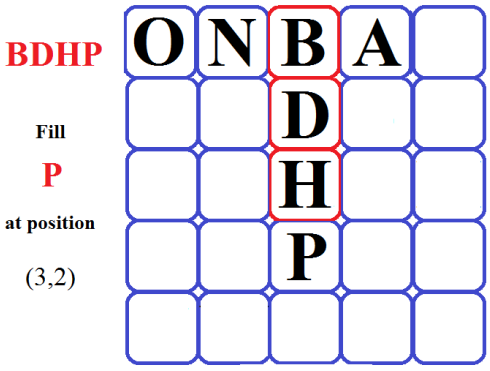
Step- 1



Step-2



Step- 3



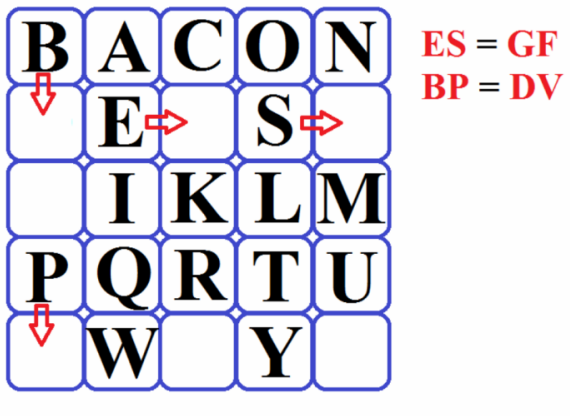
Step- 4

This entire process is repeated until the list of quadruplets has been scanned through at all positions in the matrix as progressively more and more of the matrix gets filled in. We have coined the filling in loop in a manner such that wrap around insertions are appropriately taken care of. For instance, the quadruplet "PQRT" will be filled in such that "QR" will be to the right of "P", but once the indices exceed the confines of the frame, modular algebra will kick in and the "T" will fill two positions to the left of "P".

We repeat this process with the list of triplets. After all the quadruplets and triplets have been pieced together, we obtain at this stage, what we call the Swiss Cheese Matrix.

- **Linear Fill Function**

Once we have exhausted our reserve of "Conjoined Linear Mappings" by filling in all our Triplets and Quadruplets to the matrix, we are left with a Matrix that resembles a sample of Swiss cheese. i.e. a matrix with holes in it. With a sample of Plain text with Cipher text about 200 characters long, so far your constructed Matrix should look something like this:



The above diagram is ripe with instances is “Disjoint Linear Mappings” that haven’t been filled in yet. A Disjoint Linear Mapping is one in which both the Plain Text pair and the Cipher Text pair lie in the same row or column, but unlike triplets, do not share a common alphabet.

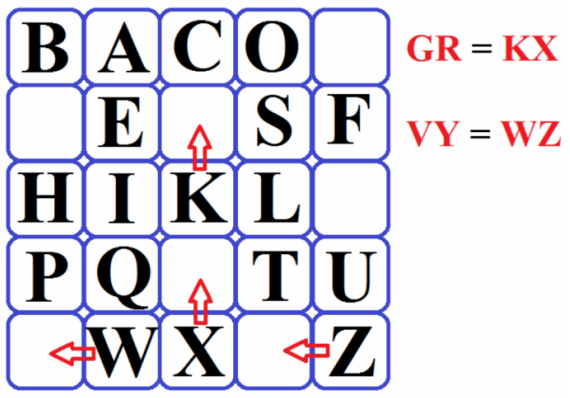
In the above diagram, two such instances have been illustrated.

1. We are given that ES maps onto GF.
2. We are given that BP maps onto DV.

From the Matrix we have partially constructed so far, we can say with certainty that “E” and “S” lie in the same row. Therefore, by the rules of Playfair we may infer that “G” and “F” will map to the locations to the right of “E” and “S” respectively in the Matrix, subject to the appropriate wrap around conditions.

Similarly, the same can be said for “D” and “V” , which will map column wise to the positions below “B” and “P” respectively, subject to the appropriate wrap around conditions.

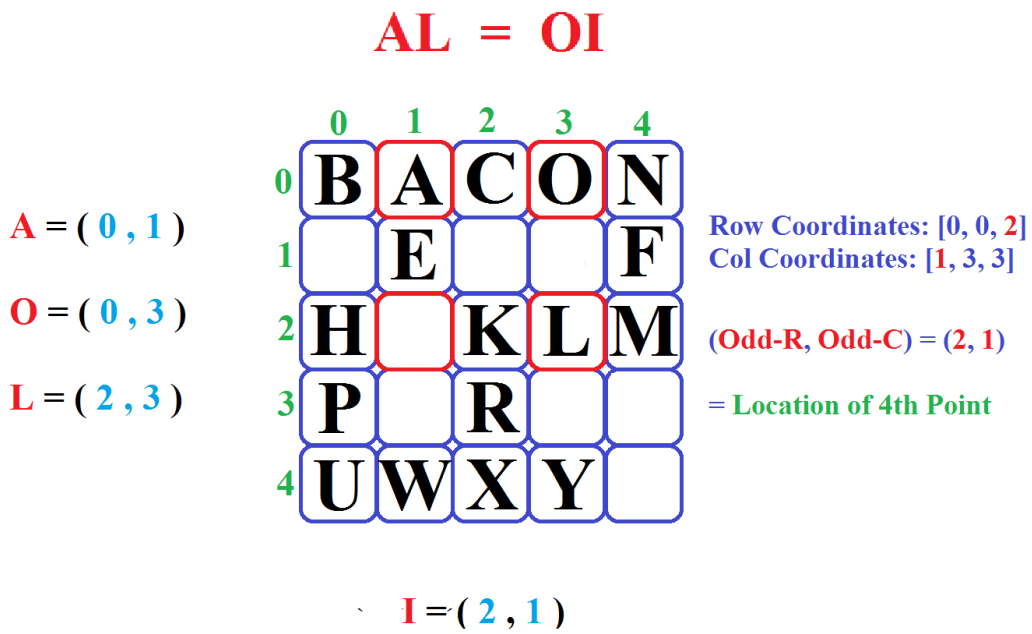
But why stop there? Suppose we had instances of Linear Cipher text pairs within our partially constructed matrix, for which we couldn’t find corresponding Plain text?



It would be just as simple to map our way backwards from “KX” to “GR” or from “WZ” to “VY” in the Matrix, subject of course to the appropriate wrap around condition.

- **Use Rectangles Function**

So far, in the extraction of our matrix, we have relied only on linear mappings to pave our way for us. This is because it is very difficult to get useful information about the matrix from rectangular mappings unless we already have a large portion of the matrix already in place.



The only instance wherein a rectangular mapping is of any actual use to us is illustrated above. Here we have the rectangular mapping "AL = OI ", and three points of this mapping ("A", "O" and "L") are already present in the partially constructed matrix. If you are solving it by hand, then it becomes intuitively obvious where the fourth point "I" should be inserted to complete the rectangle. But the question we must address algorithmically is, "How does one triangulate the coordinates of the fourth point in the matrix, given only the matrix coordinates of the other three points?"

We have addressed this problem with what we call our "Odd Man Out" algorithm:

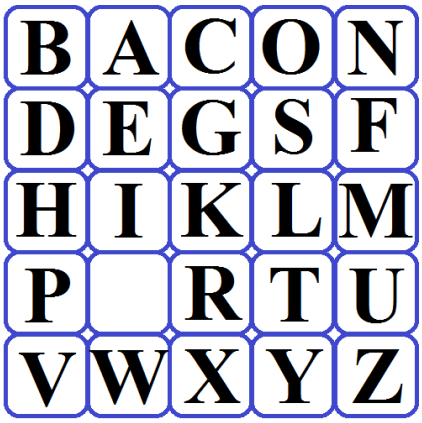
- The Row and Column coordinates of each of the three known points are separately lined up.
- In both cases, there will be an Odd coordinate that doesn't match with the other two.
- The Odd Row coordinate and the Odd Column coordinate together represent the coordinates of the missing point.

In this manner, our three point triangulation method should help us fill out a large portion of the remainder of the matrix. Note that this method becomes easier and easier to employ as and when a greater fraction of the matrix has been filled out.

A WORD OF CAUTION: Some who haven't completely thought the problem through may think that one could map their way forward given only two opposite Plain Text corners of a rectangle or two opposite Cipher Text corners of a rectangle, in a manner similar to the Linear Fill function. But be warned, this could lead to disastrous consequences. See the section on Transposing the Matrix to better understand why one should stick with a three point triangulation system whilst dealing with rectangular mappings at this stage in the construction of the Matrix.

- ## Last Resort Function

By this stage, given a sufficiency of data, you should have ideally extracted the Matrix in its entirety. But in some rare circumstances, when you have exhausted all the data given to you, and are left with one annoying little blank in your matrix, the fix is simple and there is no need to fret:



Depending on which convention you have used, simply figure out which of your alphabets is missing, and where the blank space is. Then it is a simple task of inserting the missing alphabet in its place. In the above illustration, I have employed the convention wherein I treat "I" and "J" equivalently, and therefore the missing alphabet is "Q".

A WORD OF CAUTION: I have had people come up to me and ask me whether it is okay to simply guess the alphabet that comes in between if they come across a sequence of alphabets in the Matrix that appear together in the English alphabet. Like in the above illustration for instance, it should hit any casual observer that we have reached the portion of the Matrix that is sequentially laid out. Moreover, PQR lies sequentially in the alphabet, so one should be able to fill out Q without sparing too much thought. My answer, to people who find such tactics enticing is that these may work every now and then if you are lucky, but are not always reliable. They will more often than not, lead you into trouble. Always remember that you can never really know where in the Matrix the original password is, because of the positional equivalence of wrap around matrices. Imagine for a moment that other parts of the matrix were also missing; particularly, the alphabet "A". For all you know, the original password to the matrix was something that contained the fragment "PART", and your guess to fill out "Q" in the blank would have completely led you astray.

- ## Transposing the Matrix

By now you should have constructed the Matrix in its entirety. There is just one last thing to consider. Recall that in the Fill Matrix function, the very first step was to pick up ANY quadruplet and fill it into the first four slots in the Matrix. In the example illustrated below, the quadruplet we chanced upon in the beginning was EIQW. While this is a perfectly valid choice, on comparison to the original Matrix, we find that EIQW was a quadruplet that fit into the Matrix column-wise, while we in our construction of the Matrix filled it in row-wise.

E	I	Q	W	A
G	K	R	X	C
S	L	T	Y	O
F	M	U	Z	N
D	H	P	V	B

Constructed Matrix

B	A	C	O	N
D	E	G	S	F
H	I	K	L	M
P	Q	R	T	U
V	W	X	Y	Z

Original Matrix

That and all the subsequent construction that followed it, has led us to the construction of the transpose of the required Matrix. If you look carefully, you will find that the code phrase “Bacon and Eggs” can be read vertically instead of horizontally in the constructed matrix.

This poses an actual problem. The transposed Matrix is not equivalent to the original matrix in the way it encodes text. Unlike Translation along the wrap around plane of a Playfair matrix, Transpositions of the plane do not conserve all mappings.

Linear mappings are conserved in the Transposition of a Matrix, but Rectangular mappings end up being inverted. For instance, consider the Linear mapping “TO” in both the matrices above. In both cases, “TO = YS”. But now consider the Rectangular mapping “FR” in both matrices. In the Original Matrix “FR = GU”, but the Constructed Matrix “FR = UG”.

So after all this work put into extraction, the Matrix we finally arrive at cannot in the true sense be considered our Original Playfair Matrix. But there is a simple fix at hand. After construction of the Matrix, we simply check whether all the Plain text in hand correctly maps onto the corresponding Cipher text in hand. If in fact, we have accidentally ended up constructing the Transpose of the required Matrix, then we will encounter inverted contradictions where rectangular mappings should be. And should such contradictions arise, we fix it by simply Transposing the Matrix back (i.e. by switching all the rows for columns).

This should finally land you up with a Playfair Matrix that is completely equivalent to the one originally used to encode the message you have at hand.

E	G	S	F	D
I	K	L	M	H
Q	R	T	U	P
W	X	Y	Z	V
A	C	O	N	B

NOTE: This was the reason that we could not triangulate the other two vertices of a rectangle given only two opposite vertices of its diagonal in the Use Rectangles function. At this stage in the construction, we have no idea whether we are constructing the Matrix in its original

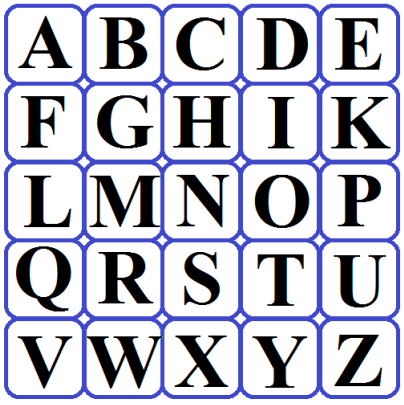
orientation, or we are constructing its Transpose. Without this knowledge, if we try inserting the other two corners of the rectangular mapping we will (half of the time) insert them in the inverted configuration without intending to, consequently messing up the construction of the remainder of our matrix.

- **Efficiency Comparisons**

In order to gauge the efficacy of our algorithm, we first must analyse some other basic methods.

Now the most basic algorithm to extract the matrix, which we referred to as "The Stoopid Matrix Extractor" is essentially a brute force extraction process applied to the Playfair Matrix problem.

This algorithm involves permuting through every single possible matrix starting from the default A-Z matrix, and checking at every step whether the changed matrix satisfies the given Ciphertext-Plaintext encryption. This is the most basic way of trying to crack any kind of encryption and understandably takes the longest.



In the most basic program, the matrix permutes 7! or 5040 times every 13 seconds. This figure is seriously limited by the fact that the computer processor has to visually show every alphabet being changed in real time as the process is taking place, i.e. is this constraint is removed then the process is sped up significantly, which takes us to the next improved method.

The 'Optimal rate of scanning' method involved the same brute force technique, except that in this case the program does not show every visual change but rather just tells us the time lapse that occurs when 10! Loops have been permuted through. The number we arrive at is 10! or 3.5 million matrices every 15 seconds. This is significant improvement from the timings of the basic brute force technique.

If we were to calculate the total time it would take to find the Playfair matrix using this improved brute force method, drawing an analogy from the odometer of a car, if the 11^{th} letter from the bottom changes every 15 seconds, every cycle it changes 11 times, the 12^{th}

letter will change once. Similarly every cycle when the 12^{th} letter changes 12 times, the 13^{th} letter will change once. Applying this idea across the entire matrix, it will take 15 x (11x12x13x14x..x24*) seconds to permute through all the matrices.

The reason the calculation stops at 24 stems from the fact that the Playfair matrix can be permuted around any one letter equivalently. What this means is that we can fix the top left (or any other) letter and build the matrix around it.

Coming back to the calculation, this number is approximately $2.8x\ 10^{18}$ seconds.

Which is approximately 8.12×10^{10} years. This figure is roughly 8 times the age of the universe.

By comparison, the algorithm which has been discussed in this paper can find the correct Playfair matrix in about 0.05 seconds.