Breaking Enigma with a simulated Turing–Welchman Bombe

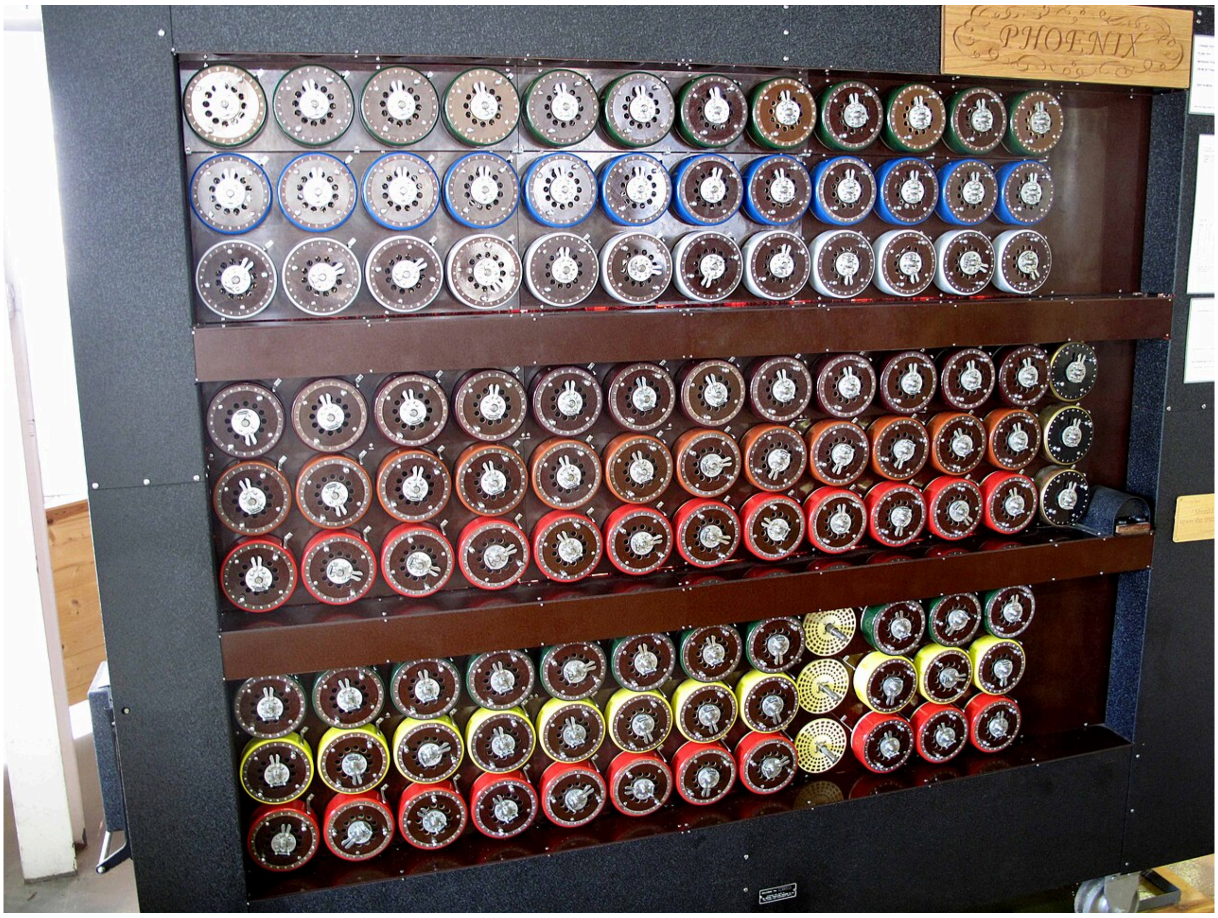


Figure 1: The rebuilt Bombe at Bletchley Park

Introduction

The Enigma machine is perhaps the best known cipher in history, and almost certainly one of the most historically significant. Alan Turing (aided by Gordon Welchman) devised the principles behind the cryptanalytic machine called the Bombe. Initially based at Bletchley Park, Bombes eventually enabled tens of thousands of Enigma messages to be broken every day. The stream of intelligence that came out of Bletchley Park was known as ULTRA.

In this document, I'll guide you through the concepts of the attack in what I hope are relatively simple terms. Follow along with pencil, paper, and Python, then have a crack at the challenges at the end. It's intended that you do all the programming tasks in order.

I'll assume that you are already familiar with Python (or the programming language of your choice!) and that you have a reasonable general knowledge of classical cryptography (*cipher*, *plaintext*, *ciphertext*, basic ciphers like *Caesar*, *Affine* and *Vigenère*) and you are confident writing simple recursive algorithms.

Substitution ciphers

Let's start with the basics. Caesar, Affine, and keyword substitution ciphers are all examples of *monoalphabetic* substitution ciphers. This means that there is a single substitution alphabet used throughout the entire process of encryption. In other words, each plaintext letter always maps to the same ciphertext letter, regardless of where it appears.

Ciphers that use many monoalphabetic substitutions in a sequence are called *polyalphabetic*. When each monoalphabetic substitution is a Caesar cipher and the pattern of alphabets repeats at a regular interval (the *period*), you've got a Vigenère cipher.

I'll explain Enigma in more detail in the next section, but mathematically speaking, it is a polyalphabetic substitution cipher; it's similar to a Vigenère except the period is very long (so long that you never come near to it in real-world messages) and the monoalphabetic substitutions all use completely mixed alphabets.

q	w	e	r	t	y	u	i	o	p	a	s	d	f	g	h	j	k	l	z	x	c	v	b	n	m
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z

Note that I am using the convention that ciphertext is uppercase and plaintext is lowercase.

> Tasks:
>
> 1. Write a function `substitute(letter, alphabet)` which encrypts a letter using a mixed substitution alphabet.[†]
> 2. Use your function to encrypt `CYHOZK` with the alphabet `QWERTYUIOPASDFGHJKLZXCVBNM`

Every substitution alphabet has an *inverse*, which is another alphabet which performs decryption when you use it with your `substitute()` function. This makes sense, because decryption is just another monoalphabetic substitution: of course it can be represented with a substitution alphabet. One way to compute the inverse substitution is to sort both rows of the substitution table by the bottom row (ciphertext alphabet):

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
K	X	V	M	C	N	O	P	H	Q	R	S	Z	Y	I	J	A	D	L	E	G	W	B	U	F	T

[†]To save yourself some effort later, you might want to represent each letter as a number between 0 and 25. If so, write some functions to convert between representations.

If a substitution alphabet is the same as its inverse, we call it *self-inverse*. I will use the notation A^{-1} to indicate the inverse of the alphabet A.

Tasks:

1. Write a function to compute the inverse of a substutition alphabet. See if you can do this in linear time.
2. Decrypt `STFKOI` by taking the inverse of `KEYWORDFGHIJLMNPQSTUVXZABC` and using your `substitute()` function.
3. Determine which of these alphabets are self-inverse:
   - `XYZGBHIJKLAMNOEPQFRSTUDVCW`
   - `OKFRNCPWUVBTYEAGXDZLIJHQMS`
   - `YMDRHWSZAOENKBXQLTUIPGJVCF`
4. Write a function called `compose()` which takes two monoalphabetic substitution alphabets (A and B) and computes the overall substitution when the two are performed in series (B first, then A).

The Enigma Machine

Onto our adversary: the Enigma machine. We already know that it's a polyalphabetic cipher with a long period... but how does it work?

The panel on the front face with the wires is the plugboard. It has one socket for each letter in the alphabet. There are thirteen plugs, each of which can connected up to two letters. When a plug is connected, it causes the plugboard to swap the letters it is connected to. When no plug is connected into a socket, the corresponding letter is mapped to itself. The plugboard is a configurable monoalphabetic substitution. Notice that, because the plugboard can only either encrypt letters as themselves or swap them in pairs, it is self-inverse.

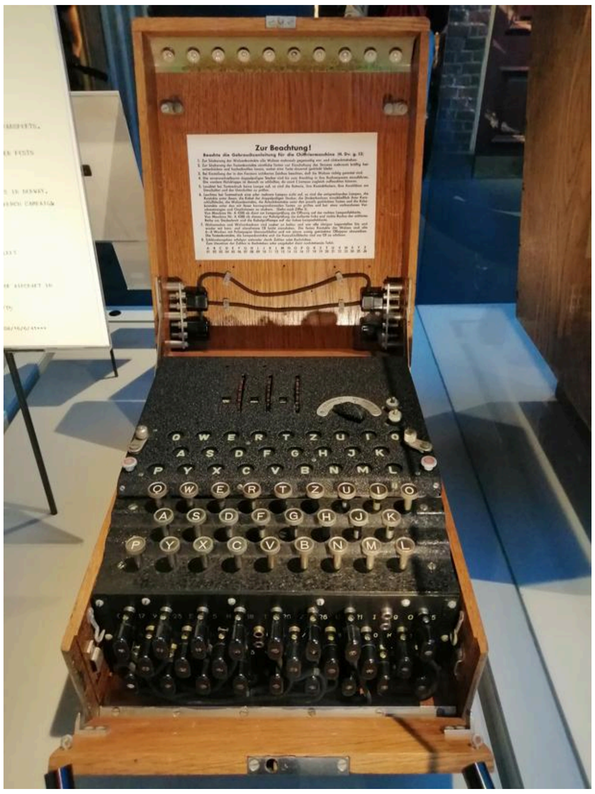


Figure 2: Enigma machine at Bletchley Park
© upsidedown

Under the black casing are three rotors and a reflector. The reflector remains stationary throughout the operation of the machine; it performs a self-inverse monoalphabetic substitution.

The rotors each perform a monoalphabetic substitution, but can rotate (unsurprisingly), causing the substitution to change. Each rotor has a ring containing each letter of the alphabet. There are three slits for the top of the rotors to poke through (so that the operator can set the position of the rotors), and three rectangular apertures through which the letters that each rotor is set to are visible. These three letters are called the indicator values.

The typewriter-like labelled keys at the front are where the operator types the message. The operator would press one of these keys, and that motion would cause the rotors to advance by one position. While the key is held down, an electrical connection is formed which travels through the plugboard, rotors, and reflector (which *reflects* the signal back to a different letter), then back through the rotors and plugboard in the other direction (equivalent to the inverse of the forward substitution) and to a letter on the light panel above.

When the rotors advance, the right rotor always turns, and like the minute hand on a clock, the middle rotor turns once every 26 turns of the right rotor. The left rotor turns every 26 turns of the middle rotor (technically this is not entirely accurate due to double-stepping).

There is a convention for referring to the rotors which you need to know. The electrical signal enters the rightmost rotor, passes through the middle rotor, then the left rotor, through the reflector, and back the other way. The right rotor rotates most frequently and is therefore called the *fast* rotor. The left rotor is called the *slow* rotor. When the rotor order, indicator positions and ring settings (we'll get to those later) are written down, the order is left-to-right.

Most of this complexity is unimportant for the actual attack and will be abstracted away as a monoalphabetic substitution that changes on each keypress.

We're now going to build up a program to simulate an Enigma machine.

Reflector

Reflector	**Substitution**
A	EJMZALYXVBWFCRQUONTSPIKHGD
B	YRUHQSLDPXNGOKMIEBFZCWVJAT
C	FVPJIAOYEDRZXWGCTKUQSBNMHL

Table 1: Reflector details

Tasks:

1. Store each of the reflector substitutions as variables or constants in your program.
2. Verify that each of the reflector substitutions are self-inverse.

3. Verify that none of the reflector substitutions ever encrypt a letter to itself. Why do you think this is?

Rotors

Rotor	Substitution	Notch
I	EKMFLGDQVZNTOWYHXUSPAIBRCJ	Q
II	AJDKSIRUXBLHWTMCQGZNPYFVOE	E
III	BDFHJLCPRTXVZNYEIWGAKMUSQO	V
IV	ESOVPZJAYQUIRHXLNFTGKDCMWB	J
V	VZBRGITYUPSDNHLXAWMJQOFECK	Z
VI	JPGVOUMFYQBENHZRDKASXLICTW	ZM
VII	NZJHGRCXMYSWBOUFAIVLPEKQDT	ZM
VIII	FKQHTLXOCBJSPDZRAMEWNIUYGV	ZM

Table 2: Rotor details

Indicator positions

The initial value of the indicator letter visible through the window on the Enigma machine is also called the *start position*, or *Grundstellung* in the original German. The rotor table above shows the substitutions performed by each rotor when its indicator position is at `A`.

If the indicator setting is i, and the rotor alphabet is $R(x)$, then you can compute the substitution performed by the same rotor at position i as $R(x+i) - i \bmod 26$. This is equivalent to a left-rotation by i of the substitution alphabet followed by a caesar shift of $-i$.

Tasks:

1. Store each of the rotor substitutions in your program.
2. Write a function that left-rotates an alphabet by i places then performs a caesar shift of $-i$.
3. Use your function to compute the substitution performed by rotor `IV` at indicator `B`. Now compute the inverse of that substitution.
4. Compute the inverse of the substitution performed by rotor `IV`, then use your function to left-rotate and shift it at indicator `B`. Notice that this is the same as the previous substitution.
5. Decrypt `ZKSOKWLYQUXGRW` which is encrypted with rotor `IV` at indicator `B`.

Ring positions

It is also possible to adjust the position of the internal wiring of the rotor (the part which performs the substitution) relative to the indicator ring. This offset is between 0 (`A`) and 25 (`Z`). The effect of this offset is another rotation of the alphabet, but in the opposite direction.

If the ring setting is r, we can modify the previous equation to obtain $R(x + (i - r)) + (i - r) \bmod 26$.[†]

> Tasks:
>
> 1. Compute the substitution performed by rotor `II` at indicator `X` and ring position `Q`. You can simply reuse the function you wrote before, but with $i - r$ replacing i.
> 2. Decrypt `QRGQA` using the inverse of that substitution.

Notch positions

You might be wondering what the notch positions mean. On a physical Enigma machine there is an actual notch that mechanically engages the next rotor and turns it. The notch position in the table is the indicator position at which the next turn of the rotor will cause the rotor to the left to turn.

It's actually a bit more complicated than that due to *double-stepping* which can be a bit fiddly to get right. So I'll just give you the stepping algorithm:

```
fn step(rotors, indicator_positions):
  if middle rotor is on its notch position:
    step the middle rotor
    step the left (slow) rotor
  else if right (fast) rotor is on its notch position:
    step the middle rotor
  step the right (fast) rotor
```

To be clear, the ring settings remain the same once set, and only the indicator positions change. The `step()` function only looks at the indicator positions to determine which rotors turn (by comparing to the notch position).

> Tasks:
>
> 1. Implement the `step()` function.

[†]Perhaps you can see how this is much neater when letters are represented as integers rather than strings or characters?

2. Write a function `step_many()` that steps the indicators n times.
3. What three-letter word do you get when you step `BNH` 1000 times?

These are a little more difficult and are optional:

4. Write a function `step_backward()` that perfoms a single step to the previous indicator values.
5. Modify `step_many()` to accept negative values for n.
6. What three-letter word do you get when you step `EBS` -1000 times?

Scrambler

Before we get to the plugboard, we're going to introduce an abstraction called a *scrambler*. A scrambler comprises a forward pass through all three rotors (applying the left-rotated and shifted rotor substitution), a pass through the reflector, then a backward pass through the three rotors (in other words, the inverse of the left-rotated and shifted rotor substitution is applied). For a given list of indicator and ring positions, this is simply a monoalphabetic substitution.

Remember that the order for encryption is: right, middle, left, reflector, left inverse, middle inverse, right inverse.

Tasks:

1. Write a function called `scrambler(reflector, rotors, rings, indicators)` which computes the substitution alphabet performed by a scrambler at a given list of indicator positions and ring positions (and rotors and reflector). This should not do any stepping of the indicators. You may find that your `compose()` function from earlier is helpful.
2. Use your function to compute the monoalphabetic substitution performed by reflector `B` and rotors `III`, `IV` and `II` at ring positions `ULT` and indicator positions `RON`.
3. Verify that the substitution is self-inverse.
4. Use your monoalphabetic substitution alphabet to decrypt `CFEGFSERHRVNTG`.
5. Use the settings listed above, in combination with your indicator stepping function, to decrypt `LWCLMIRHMGKJPR` (note that indicator stepping happens *before* each letter is encrypted/decrypted).

We can now derive two important properties about the scrambler:

- It is *self-inverse*.
- It *cannot encrypt a letter as itself*.

Proof 1: Scrambler is self-inverse

Let U be the reflector substitution, and L, M, and R be the left, middle and right rotor substitutions. The overall rotor substitution $O = LMR$[†]. The scrambler substitution is then $S = O^{-1}UO$. Let's take the inverse of S (note that the reflector is self-inverse, so $U^{-1} = U$):

$$\begin{aligned} S^{-1} &= \left(O^{-1}UO\right)^{-1} \\ &= O^{-1}U^{-1}O \\ &= O^{-1}UO = S \end{aligned}$$

Using the fact that[‡] $(AB\cdots Z)^{-1} = Z^{-1}\cdots B^{-1}A^{-1}$

Proof 2: Scrambler cannot encrypt a letter as itself

We verified earlier that the reflector U cannot encrypt a letter as itself. Suppose we pass a letter, x, through a scrambler.

$$S(x) = (O^{-1}UO)(x) = O^{-1}(U(O(x)))$$

Let's assume that $x = S(x)$. Then $x = O^{-1}(U(O(x))) \Rightarrow O(x) = U(O(x))$. This is a contradiction because we know that U can never encrypt a letter as itself, therefore it is proven that $S(x)$ can never equal x.

Plugboard

The final component is the plugboard. As I mentioned earlier, the plugboard connects letters in pairs, or otherwise maps them to themselves, which makes it self-inverse. If we represent the plugboard as a monoalphabetic substitution P, we can substitute PO for O in Proof 1 and Proof 2 and obtain similar proofs for the Enigma cipher as a whole.

Tasks:

1. Write a function that computes the plugboard substitution alphabet from a list of pairs. It should raise an error if the resulting substitution is not self-inverse.
2. Decrypt `TTDEVAXDVGGQDXLWGXBKNTMUQO` using the following settings:
   - Reflector: `B`

[†] This means right first, then middle, then left.

[‡] Often explained as: if you put your socks on then your shoes, to undo that you need to first take your shoes off then take your socks off. The inverse of a sequence of operations is the inverse of each operation in reverse order.

> - Rotors: `I`, `II` and `III`
> - Indicator settings: `ENI`
> - Ring settings: `GMA`
> - Plugboard: `GH`, `YU`, `JK`, `OP`, `QW`, `ER`

Summary

Enigma performs a monoalphabetic substitution at each indicator position. The plugboard substitution, scrambler substitutions, and the Enigma machine as a whole, are self-inverse; decryption and encryption are the same operation. Additionally, scrambler substitutions and the Enigma machine as a whole cannot encrypt a letter as itself.

Graph Theory

I'm going to use some mathematical graph terminology in the next section. To ensure we are on the same page, I'll explain what a graph is.

Suppose you have five towns, creatively named A, B, C, D, and E. There are roads from $A \leftrightarrow B$, $B \leftrightarrow C$, $C \leftrightarrow A$, and $D \leftrightarrow E$. I have represented this situation in Figure 3.

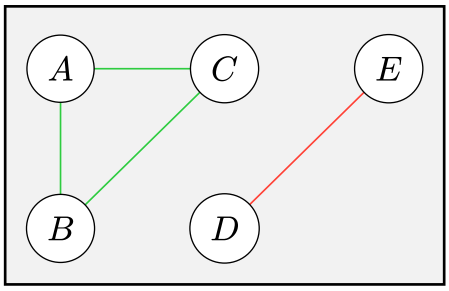


Figure 3: A simple graph

In mathematical terminology, the towns are *nodes* and the roads are *edges*. The combination of nodes and edges is called a *graph*.

Edges can be *directed* or *undirected*. An *undirected* edge can be traversed in either direction, whereas a *directed* edge can only be traversed one way. The graphs we're interested in will have undirected edges.

I'll also note that the arrangement on the page of the nodes and edges is completely arbitrary, what matters are the nodes and edges, not where or how they are drawn.

A *connected* graph is a graph where there's a path between every pair of nodes. Figure 3 is *not* a connected graph, because for example there is no path from C to D. If I added an edge connecting C to D, the graph would be connected.

A *cycle* is a path which starts and ends at the same node without traversing any edges more than once. There is one cycle (highlighted in green) in Figure 3, A→B→C→A.

A *subgraph* is any subset of the nodes and edges of a graph. See Figure 4 for some examples.

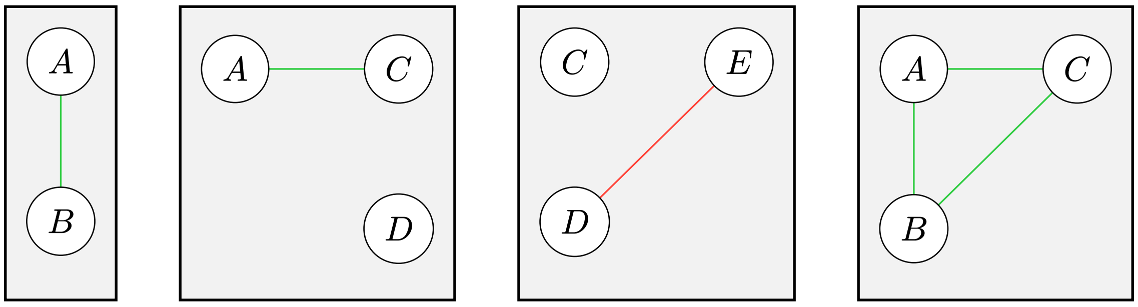


Figure 4: Examples of subgraphs of Figure 3

A *connected component* is a subgraph which is connected and where all edges connected to any of its nodes are part of the subgraph. There are two connected components: the nodes connected by the green edges; and the nodes connected by the red edges.

The Bombe

Now we have an understanding of the Enigma machine and its properties, we can move on to the Bombe: the machine that performed Alan Turing's attack on Enigma. The principle behind it is actually quite simple.

Assumptions

We assume that:

1. We know the reflector, rotors, and indicators.[†]
2. We have a correctly corresponding matching crib and ciphertext pair.
3. The middle rotor did not turn throughout the encryption of this crib.[‡]

Recall that the substitution performed by rotor R at indicator i and ring position r (and alphabet length m) is

$$R(x + (i - r)) + (i - r) \bmod m$$

Since we know that the middle rotor doesn't turn throughout the crib, we can just introduce a new indicator position $i' = i - r$ (so ring position $r' = 0$), removing all m^3 ring positions from the search space.

$$R(x + i') + i' \bmod m$$

If the middle rotor did turn, this wouldn't be valid because the ring position adjusts the offset between the rotor notch and its internal substitution (so the rotor would turn at the wrong point whenever the ring position was non-zero).

This can be summarised as a fourth assumption (actually a consequence of the first three assumptions):

4. The ring positions are zero.

So, given these assumptions, the only Enigma setting we don't know is the plugboard substitution.

Simplified Enigma

My implementation of Turing's attack (which is similar to the Bombe) uses a 26×26 boolean matrix which would take up a considerable amount of space on the page and make

[†]During the war, the reflector and rotors would be determined using statistical analysis of the ciphertext (called *banburismus*). The indicators are brute-forced.

[‡]Otherwise, we would have to try a middle rotor turn at each position in the crib. The Bombe operators did not have to hope that the middle rotor didn't turn: since the middle rotor only turns once every 26 letters, if you split a crib (with fewer than 26 letters) in two at any point then only one of those halves will contain a middle rotor turn.

it rather tiresome to follow by hand or debug your program. I'm therefore going to make up a simplified Enigma machine that only uses 6 letters: `ABCDEF`.

This also serves to illustrate the point that it's only the self-inverse property of the scrambler and plugboard substitutions that are important here: all the complexity of reflector, rotors, ring positions, indicators, and so on are not important beyond giving us a monoalphabetic substitution in each position.

Reflector	**Substitution**
A	BAFEDC

Table 3: Reflectors on the simplified Enigma

Rotor	**Substitution**	**Notch**
I	CBEADF	F
II	FEDACB	A
III	FCABED	B

Table 4: Rotors on the simplified Enigma

Tasks:

1. Modify your Enigma code to work with the above alphabet, rotors and reflector.
2. Verify that the reflector substitution is self-inverse.
3. Decrypt `ECFBCB` using reflector `A`, rotors `I`, `II`, `III`, indicators `ACE`, rings `BEE` and an empty plugboard.

Matching a crib

Turing's attack is a *known-plaintext* attack: it requires a crib. This crib needs to be exactly right and reasonably long[†] to solve on a Bombe, and coming up with a crib is not easy.[‡] However, the same key would be used for all the communications within a division of the army each day[§], so you would only need get lucky once to decrypt much of the day's intercepted traffic.

[†]With the actual Enigma (rather than my simplified variant) something like 20 letters would be good. Of course, the crib can't be longer than 25 letters, because the middle rotor is guaranteed to turn every 26 letters.

[‡]Traffic analysis is useful: you might be able to figure out who is talking to whom based on the origin of the radio signals. You know when the message was sent. You might know what sort of information the message is likely to contain, and the writing style of the sender. The usual example is a weather station: you can guess what the plaintext will say by observing the weather conditions near to the station.

[§]Later in the war the Enigma settings would change twice per day.

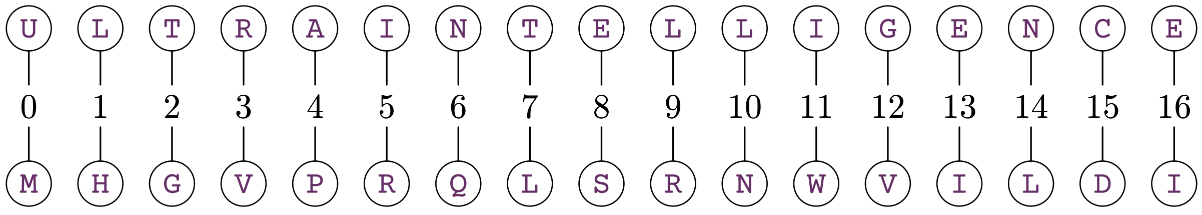
One of the properties of Enigma can be put to use here: no letter in the crib can be encrypted as the same letter in the ciphertext. We can use this to disregard some, hopefully most[†], incorrect crib and ciphertext pairings.

Tasks:

1. Write a function `is_valid_crib()` that checks whether a candidate crib and ciphertext pairing are possible, using the no-self-encryption property.
2. Find the position where the crib `dcafedeed` appears in this (simplified) Enigma ciphertext: `FDCFEAFBFADFC`.
3. Find all the positions where the crib `brownfoxjumpsoverthe` could appear in `MINDCCPJTAEFWRQVQTXSBPLNIKXCTPXWZUJ`, encrypted with a normal Enigma machine.

Let's add a valid crib, and the knowledge of its position in the ciphertext, to our growing list of assumptions.

Creating a menu


U L T R A I N T E L L I G E N C E
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
M H G V P R Q L S R N W V I L D I
Figure 5: Edges obtained from crib

Suppose our crib is `ultraintelligence` and the corresponding ciphertext is `MHGVPRQLSRNWVILDI`. As I've mentioned a few times before, the nth letter of the crib is related to the nth letter of the ciphertext by a monoalphabetic substitution (combination of plugboard and scrambler). We can think of these relationships as a bunch of edges, as illustrated in Figure 5 above.

Notice that there are several letters that appear multiple times. We can combine these edges to form a graph, which the Bombe operators referred to as a *menu*.

[†]Part of the reason that cribs need to be reasonably long is that longer cribs will typically have fewer false-positive matches with a piece of unrelated ciphertext.

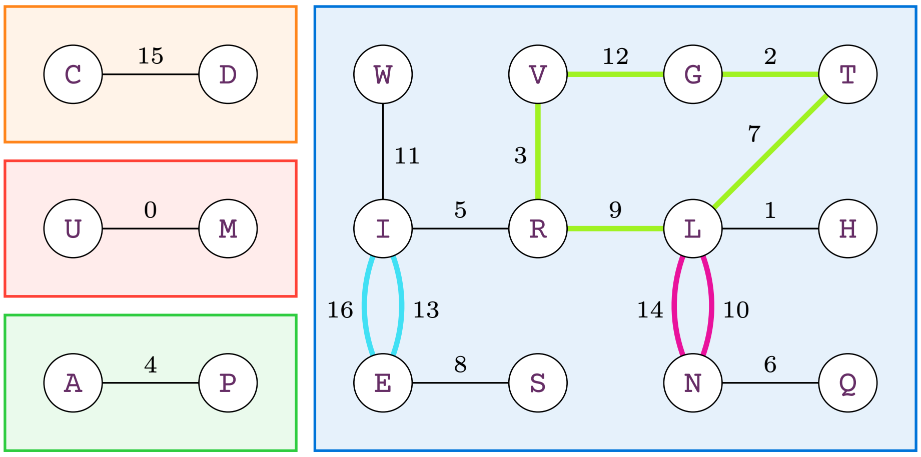


Figure 6: A menu constructed from the edges in Figure 5

Using our graph theory knowledge, we can see that the menu above has 4 connected components: 🟧, 🟥, 🟩 and 🟦. These are called *sub-menus*.

There are 3 fundamental cycles:[†] ○ IEI, ○ NLN, and ○ LTGVRL. Can you see the fourth, non-fundamental cycle?

Tasks:

1. Write a function `menu()` that constructs a graph when provided with a crib and ciphertext section. How you represent the graph is up to you. Make sure to store the edge labels (the offset numbers).
2. Use your graph structure to determine which edges are connected to the node `T`, including the labels, and the nodes at the other end of each edge.
3. Write a function that uses your graph structure to determine the node with most edges connected to it: this is called the *most-connected* node.

Proof by contradiction

Heard of proof by contradiction? The principle is that you make an assumption, show that it leads to a contradiction (e.g. $0 = 1$), then declare that this demonstrates that the original assumption is false.[§]

[†] Fundamental cycles can be thought of as the minimal set of cycles where the walk taken by each cycle visits each node no more than once (excluding the first/last node). More precisely, a fundamental cycle is any cycle that can be formed by adding a single edge to the spanning tree[‡] of the graph.

[‡] A *spanning tree* is a *tree* (a connected graph without cycles) which contains every node.

[§] This has made a lot of people very angry and been widely regarded as a bad move.

Proof 3: There are an infinite number of primes

Suppose there is a finite set of primes: $p_1, p_2, ..., p_n$. Let $P = p_1 \times p_2 \times ... \times p_n$ and $Q = P + 1$. Q is either prime or not prime:

- If it is prime, because Q is greater than any p_i in our list, we have discovered a new prime, a contradiction.
- If it is non-prime, then Q must be a multiple of some p from our list. P, the product of the primes, is necessarily a multiple of p as well. $Q - P = 1$ must therefore be a multiple of p. Since all primes are greater than 2, this is a contradiction.

Our original assumption that "*there is a finite set of primes*" is therefore false: there are an infinite number of primes.

Alan Turing realised that it was possible to apply proof by contradiction to a menu with a corresponding list of scrambler substitutions. Recall that we have assumed that we know the reflector, rotors, ring settings and indicators, and we can therefore compute the monoalphabetic scrambler subsitutions.

Let's give it a go: we assume that `T` is connected to **`Y`** on the plugboard. Looking at the connections to `T` in the extract of the menu (Figure 7) we can see two edges:

- `T` is connected to `L` at offset 7.
- `T` is connected to `G` at offset 2.

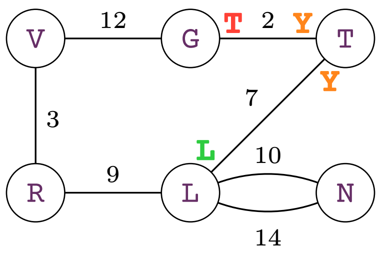


Figure 7: Extract of a submenu from Figure 6 with some **`assumed`** & **`implied`** plugboard pairings and a **`contradiction`**.

Since we know the substitution performed by each scrambler (think of each edge as a scrambler), we can follow the edges out from `T` by inputting **`Y`** to the scrambler at the offset for each edge.

Suppose the output at offset 7 was **`L`**. Since Enigma consists of plugboard, scrambler, plugboard, we now know that `LL` is a plugboard pair (`L` is mapped to itself). We can now follow each of the edges originating from the `L` node (and so on, recursively).

What if we encrypted **`Y`** using the scrambler at offset 2 and got **`T`**? This tells us that `TG` is a plugboard pair. But we already assumed that `TY` was a plugboard pair! Each letter can only be paired with one letter, so this is a contradiction: our original guess that `TY` is a plugboard pair must have been wrong[†].

[†]Or the indicator position is wrong, in which case all plugboard guesses will lead to contradiction (the overwhelming majority of the time).

Tasks:

From Figure 5 and Figure 6, The crib `ultraintelligence` corresponds to the ciphertext `MHGVPRQLSRNWVILDI`. I encrypted this using reflector `B` and rotors `III`,`I`,`II`. If you set the rings to zero (as we do in the Bombe attack), the indicator at the start is `BXA` (you do not need to step this forward before the first scrambler).

1. Use this information to compute the scrambler substitutions at each offset in the crib. This is as simple as repeatedly calling your `scrambler()` function, but you need to be careful about the indicators. While we are assuming that the rings are zero, this depends on the middle rotor not turning. You should use a simplified indicator step function that just adds 1 to the right indicator position (mod 26), without any notch logic.

The first scrambler substitution should be `OPGZNTCIHYSWXEABVUKFRQLMJD`.

2. Use your scrambler substitutions, with the initial plugboard guess `RM`, to determine what letter is steckered to `N`. Do this by hand using the substitution alphabets from the previous task.[†] Remember that the scramblers are self-inverse: you don't need to invert them.
3. Now try with an initial guess `RR`, and show this leads to contradiction.

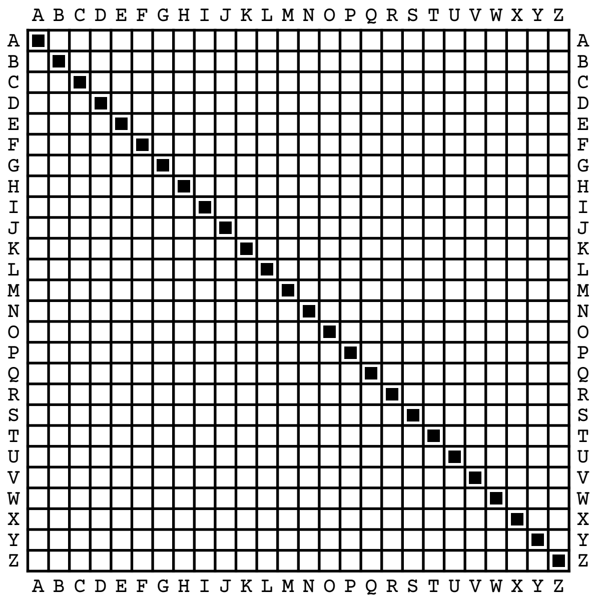
We now have a decent idea of how the Bombe works. It performs an automated proof by contradiction, which can be used to disregard the majority of indicator positions as impossible due to contradiction. I'll get into exactly how this works shortly, but first, let's write some code to automate the proof process.

[†]You can write code to do this if you prefer, but we will be automating this in the next section.

The Wiring Matrix

We're going to need a structure to store our assumptions and their implications. Something that works nicely (and is somewhat faithful to the design of the Bombe) is a 26×26 boolean matrix, which I will call the *wiring matrix*.[†]

Each row of the wiring matrix represents the letters (plural) that are connected to a letter on the plugboard. A consistent solution will of course have no more than one plugboard pairing per row, whereas inconsistent solutions will have more than one plugboard connection for at least one letter.


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

Wiring Matrix 1: Represents a plugboard where every letter is encrypted as itself

Since we are guessing an initial plugboard pairing, 25 times out of 26 we will be wrong and the rows will be inconsistent.

Tasks:

1. Write a function that creates a wiring matrix for an arbitrary alphabet length.
2. Write a function to reset every entry of a wiring matrix to false.
3. Write a function to display a wiring matrix in a tabular format similar to Wiring Matrix 1.

[†]There are, of course, other representation available. This one is conceptually simple and reasonably computationally efficient.

Tracing connections

At this point, I'm going to switch over to my simplified 6-letter Enigma machine. This will have a 6×6 wiring matrix.

I've obtained the ciphertext FDACFF by encrypting accede using reflector A and rotors II,I,III. The middle rotor was at the same indicator position throughout encryption.[†] Indicators for the scrambler at offset 0 are DDC.

As before, we obtain the edges in Figure 8 from the crib and ciphertext, then construct a menu in Figure 9. I have generated the scrambler substitutions at each crib offset in Table 5.

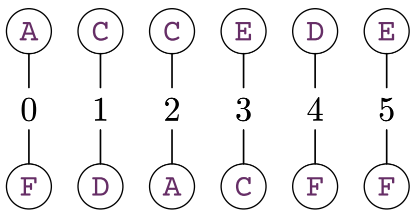


Figure 8: Edges obtained from crib

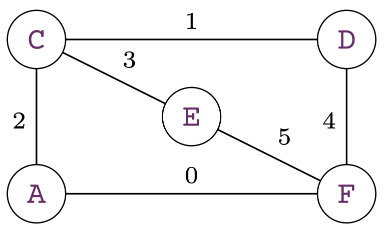


Figure 9: Menu constructed from edges in Figure 8

Each cell in the wiring matrix is like a wire: it can be live or dead, and when it is live any wires connected to it will become live. Which wires (other matrix cells) are connected to each wire (matrix cell) depends on the scramblers.

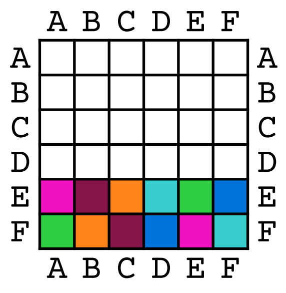
The relationship we want to encode is as follows. Suppose a and b are nodes connected by a scrambler at offset o. For each possible plugboard pairing p of a, the corresponding wiring matrix cell should be connected up to the b row, on the output of the scrambler at offset o when p is input.

Pos	**Ind**	**Substitution**
		ABCDEF
0	DDC	EFDCAB
1	DDD	FDEBCA
2	DDE	BAEFCD
3	DDF	CFAEDB
4	DDA	CDABFE
5	DDB	ECBFAD

Table 5: Scrambler substitutions for each offset in the crib

An example may help. E and F are connected by the scrambler at offset 5 (recall that this means that when the plugboard substitution of E is input to the scrambler at offset 5, it outputs the plugboard substitution of F).

From Table 5 we can see that the scrambler substitution at offset 5 is ECBFAD. If we suppose that E is connected to C on the plugboard, we see that the scrambler substitution for C is B. So this tells us that B is connected to F on the plugboard. This is denoted in Figure 10 by the C cell in the E row and the B cell in the F row being the same colour.

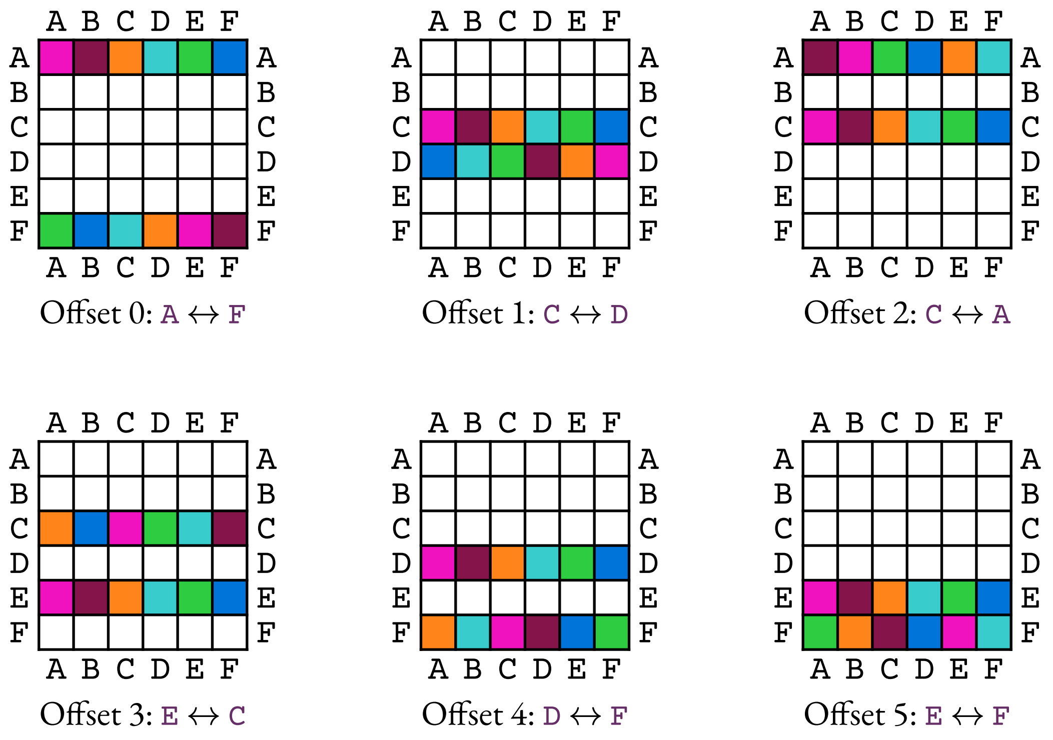


Offset 5: E ↔ F

Figure 10: Wiring matrix connections at offset 5

[†] Since the message has 6 letters, and the middle rotor turns every 6 letters, when must the most recent turn have occured?

We can visually verify that the scrambler substitutions are self-inverse on the diagrams: for example the **green** and **teal** squares form an alternating pair.


Offset 0: `A` ↔ `F`   Offset 1: `C` ↔ `D`   Offset 2: `C` ↔ `A`

Offset 3: `E` ↔ `C`   Offset 4: `D` ↔ `F`   Offset 5: `E` ↔ `F`

Figure 11: Wiring matrix connections for each scrambler substitution[†]

Recall that the proof by contradiction begins with an assumption, which is represented by making a single wire (matrix cell) live (true). It is best to use the *most-connected* letter to begin this process.[‡] We can see from Figure 9 that this is `C` or `F` (each with with 3 edges). Menus that contain more cycles are generally better at generating contradictions.

Let's begin with the assumption that `C` is connected to `D` on the plugboard. We start with an empty wiring matrix, then set the `D` wire of the `C` row as live. You can see in the first matrix in Figure 12 (next page) that the `DC` wire is live.

Live wires are black or white; white indicates new live wires or the other end of the scrambler used to derive a new live wire.

We continue the process of deriving new plugboard pairings, by *tracing* the wires in the matrix, until every wire connected to the initial hypothesis has been traced.

[†]The Bombe uses massive 26-way cables to connect up rows of the wiring matrix.

[‡]It leads us to contradictions[§] more quickly, and there are more constraints on the row of the wiring matrix corresponding to the most-connected letter, a fact we will use later.

[§]Remember, we are not trying to discover the plugboard pairings (a useful side-effect), but rather to prove that a given indicator position is impossible.

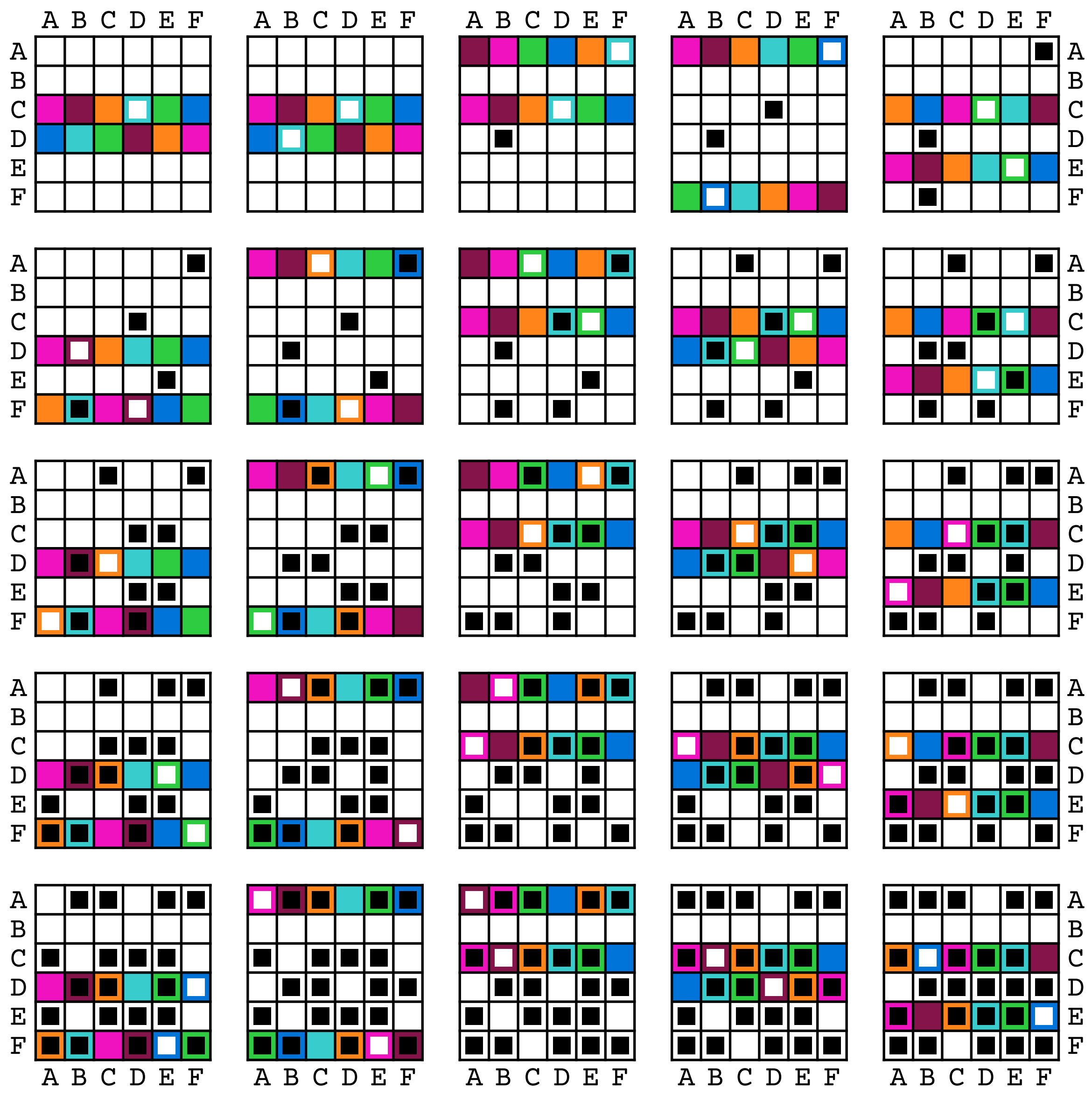


Figure 12: Iterative process of following connections in the wiring matrix until the whole circuit has been found. The background colours show the scrambler substitutions used at each step to derive the new live wire. These scrambler substitutions are from Figure 11

After exhausting all the connections, we obtain the wiring matrix shown in Wiring Matrix 2. I'll go in to how we interpret this result in the next section.

Notice that the `B` row is completely empty: this is because the letter `B` does not appear in the menu. Gordon Welchman contributed a simple but crucial improvement to the Bombe's design that fixes this problem, as well as improving the overall effectiveness of the method. I'll talk about it later, but perhaps you can figure it out for yourself if I tell you that it relies on the self-inverse property of the plugboard substitution (it connects letters in pairs).

	A	B	C	D	E	F	
A	■	■	■		■	■	A
B							B
C	■	■	■	■	■		C
D		■	■	■	■	■	D
E	■		■	■	■	■	E
F	■	■		■	■	■	F
	A	B	C	D	E	F	

Wiring Matrix 2: Matrix after every wire connected to the initial hypothesis has been followed

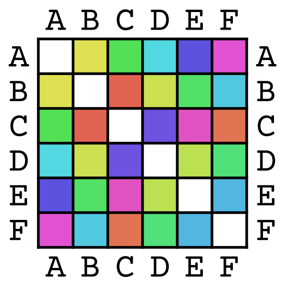
Tasks:

1. Write a recursive function `trace()` which operates on a wiring matrix and uses the edges from your menu graph structure. Here's an outline of the algorithm:

```
fn trace(row, col):
  if (row, col) is live:
    return

  set (row, col) to live

  let letter = letter represented by `row`

  for scrambler in (scramblers connected to `letter`):
    # Remember, each column represents a possible
    # plugboard pair connected up to `letter`
    let input = letter represented by "col"

    # Input the plugboard substitution of `letter` (which
    # is `input`) to the `scrambler`
    let output = output of `scrambler` applied to `input`

    # The scrambler offset is the label of an edge that
    # connects two nodes.  Get the other node.
    let other = other node connected to `scrambler`

    trace(other, output)
```

2. Use your `trace()` function to obtain the wiring matrix for the same menu I used in this section (Figure 9), and verify that it matches Wiring Matrix 2.

The Diagonal Board



Wiring Matrix 3: Diagonal board connections

So what was Gordon Welchman's contribution? It rests on the fact that, if `A` is connected to `B` on the plugboard, then `B` is connected to `A`. In the context of the wiring matrix, this means that a live wire on the `B` column of the `A` row represents an equivalent assumption to a live wire on the `A` column of the `B` row. This means that the two wires can be connected.

In general, each wire on the top half of the wiring matrix gets an additional connection to its reflection in the leading diagonal. This series of additional wire connections is called *the diagonal board*.

The diagonal board improves the performance of the Bombe: it greatly increases the proportion of invalid solutions that lead to contradiction, and can provide connections between submenus (which are otherwise completely independent[†]).

Tasks:

1. Implement the diagonal board by modifying your `trace()` function. This is a single additional line of code.
2. Use your improved `trace()` function to obtain the wiring matrix for the menu I used in the previous section (Figure 9) with an initial guess that `D` is the plugboard pairing for the `C` row, and verify that it matches Wiring Matrix 4.
3. Run `trace()` again with an initial guess of `A` in the `C` row. Notice that you obtain the same wiring matrix.
4. Run `trace()` again with an initial guess of `F` in the `C` row (the correct pairing). Since this initial pairing is correct, all of the additional pairings it implies will also be correct.

	A	B	C	D	E	F	
A	■	■	■		■	■	A
B	■		■	■		■	B
C	■	■	■	■	■		C
D		■	■	■	■	■	D
E	■		■	■	■	■	E
F	■	■		■	■	■	F
	A	B	C	D	E	F	

Wiring Matrix 4: Final matrix with diagonal board[‡]

Interpreting the wiring matrix

All of the live wires in a wiring matrix are connected. This has the useful consequence that energising any one of those wires will energise all of them. Restated in terms of our proof by contradiction: the relationship between a plugboard pairing and its implication(s) is *if and only if* (⇔). This means that when a guess leads to contradiction, we can eliminate from consideration any of its implications.

	A	B	C	D	E	F	
A	■	■	■		■	■	A
B	■		■	■		■	B
C	■	■	■	■	■		C
D		■	■	■	■	■	D
E	■		■	■	■	■	E
F	■	■		■	■	■	F
	A	B	C	D	E	F	

Trace `CA`.

	A	B	C	D	E	F	
A				■			A
B					■		B
C						■	C
D	■						D
E		■					E
F			■				F
	A	B	C	D	E	F	

Trace `CF`.

	A	B	C	D	E	F	
A							A
B		■					B
C							C
D							D
E							E
F							F
	A	B	C	D	E	F	

Trace `BB`.

	A	B	C	D	E	F	
A	■	■	■	■	■	■	A
B	■		■	■	■	■	B
C	■	■	■	■	■	■	C
D	■	■	■	■	■	■	D
E	■	■	■	■	■	■	E
F	■	■	■	■	■	■	F
	A	B	C	D	E	F	

Trace `CA`.

Figure 13: Different completed wiring matrices using the same menu

We see in Figure 13 that the wiring matrix differs depending on the initial guess.

[†]The scramblers were the only connections between nodes, and each submenu is, by definition, not connected to the nodes in any other submenu by scramblers. This is why the `B` row was completely empty in the wiring matrix at the end of the previous section.

[‡]My brother says these diagrams look like minesweeper.

1. The first matrix is inconsistent: there is at least one row with more than one plugboard connection to each letter. However, there are no rows with every wire live. This means that while our initial guess (`CA`) was wrong, this *may* be the correct indicator position. Our proof by contradiction can only prove that an indicator position is invalid (see 4), so we can't say for sure whether this is a valid solution. This happens when the correct indicator value is chosen 25 times out of 26 (because we usually guess the plugboard pair incorrectly).
2. The second matrix is consistent: there is only one implied plugboard pairing in each row. It tells us all of the plugboard pairs I used to encrypt the crib. This will happen with the correct indicator value 1 in 26 times, so is quite rare.
3. The third matrix only has a single live wire: this is because the letter `B` does not appear in the menu. Notice that we can still derive the plugboard pairing for `B` despite it not appearing in the menu, because `B` can be the output of a scrambler. This would never actually happen, because we always choose the most-connected letter to start the trace.
4. For the fourth matrix I used scrambler substitutions at the wrong indicator value. This has resulted in an inconsistent solution, but because at least one row has every plugboard pairing implied, and we know that trying any other guess (except `BB` for the reason above) will generate the same matrix, we can eliminate this indicator position from consideration. This is by far the most common case.

Possible solutions (case 1 or 2[†]) would cause the Bombe to stop rotating (and actually rotate the drums back a few places because they would overshoot), so we refer to this situation as a *stop*.

We will use the *test register* to determine whether a completed wiring matrix is a stop. This is simply the row of the wiring matrix corresponding to the most-connected letter in our menu, the letter we used to start the tracing process. It follows that the test register must have a minimum of one live wire. There are three common cases:

1. ■■■■■■: if every wire in the test register is live, the solution is invalid.
2. ■■□■■■: if every wire except one in the test register is live, we have a stop. Clear the matrix and retrace starting at the wire that was not live (the correct plugboard pairing), then proceed to the next case.
3. □□■□□□: if the test register has one wire live, we have a stop (and we have correctly guessed the plugboard pairing). We can now read off all the implied plugboard pairings from the matrix; usually we don't get the pairing for every letter from this process.[‡]

■■□■□■: a fourth, much less common case: there are between 2 and $n - 1$ (for alphabet length n) live wires; this *might* be a stop. Since not every wire in the test register is live, there

[†]And technically case 3 as well, but you would never use a letter that's not in your menu as an initial guess.

[‡]It's possible to narrow down the set of possibilities for the remaining pairings by iteratively eliminating (see the fourth case) all the implications of each invalid circuit (which is any circuit that implies multiple pairings for a single letter).

must be 2 or more circuits reachable from the wires in the test register. The example I showed in Figure 13 had three circuits (the first three matrices). We want to determine whether any of these additional circuits represents a consistent solution.

The Bombe operators would do this by making one of the dead wires in the test register live: if doing this caused one or more additional dead wire to become live in the test register then all of the newly live wires formed an invalid circuit; if no more wires became live in the test register then there was a stop with that wire representing a valid plugboard pair.

We can do this a little differently, by recording which wires are live in the test register initially, in a list `invalid_pairs`. We can then clear the wiring matrix and try tracing each of the possible plugboard pairs that are not in the list. If the result of tracing one of these possibilities results in an invalid solution, we add all of the live wires in the test register to `invalid_pairs`, and carry on. If `invalid_pairs` contains every pair, the solution is invalid; otherwise, we have a stop.

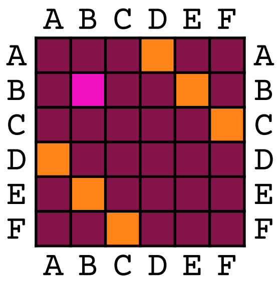


Figure 14: The circuits of the wiring matrix

Another way to think about all of this: we are tracing the matrix to find all of its circuits (shown in different colours in Figure 14). A circuit is valid if it contains 0 or 1 connections in each row. A set of circuits that we obtain for a specific indicator value is valid (a stop) if it contains at least one valid circuit and the union of the invalid circuits contains no row with 26 live wires.

Tasks:

1. Write a function `interpret()` that takes a completed wiring matrix, and the most-connected letter, and uses the test register to determine:
   1. Whether it is a valid stop.
   2. If it is valid, what plugboard pairings can be determined.
2. Run the simulation for the earlier menu using a 26×26 matrix, with an initial guess of `RR`.

   I'll repeat the crib `ultraintelligence`, ciphertext `MHGVPRQLSRNWVILDI`, and settings: reflector `B`, rotors `III`,`I`,`II`, initial indicator position `BXA`.

   This will lead to a case where 25 of 26 wires are live in the test register. Recover the scrambler pairings.

Searching through the indicator positions

Our initial assumptions state that we know the rotors and reflector (which can be determined statistically); that the middle rotor doesn't turn throughout the crib (which you can ensure by splitting a crib); and that we know the indicator positions… but we don't!

If you've ever seen the rebuilt Bombe in action you will know there is a lot of spinning round involved.[†] This is because the Bombe is exhaustively trying all $26^3 = 17,576$ indicator positions: this is exactly what you need to do. It's as simple as three nested `for` loops.

So do we have the solution now? Not exactly. The Bombe attack will give us indicator values and most of the plugboard pairs for each stop, but we still need to determine the rings and any remaining plugboard pairs, then work out the indicator value at the start of the message. A curious detail: the ring setting of the left rotor doesn't matter as it does not advance the (nonexistant) next rotor when it reaches its notch position. Likewise, the ring setting for the middle rotor only matters when your message has, on average, $26^2/2 = 338$ letters. I'll leave the rest to you.

The fastest version of the Bombe took around 20 minutes to do a complete search. Read on to the next section to see how you can make yours faster.

> Tasks:
>
> 1. Write a function that performs the attack for each of the 17,576 indicator positions. As a reminder, when you compute the scrambler substitutions the middle rotor *must not turn*: you should not use your `step()` function, and instead just increment the right indicator value by one.

Optimisation

If you've followed along you'll have a working proof-by-contradiction attack on Enigma. Here are some things you can do to make it faster.

These get a bit technical, so if you're unclear just implement the first one. My Python program takes 23s to run the complete attack on my laptop using the first 3 optimisations (so on a single thread).

1. You should precompute the monoalphabetic substitution performed by the scrambler at each of the 26^3 rotor indicator positions. Each position uses one scrambler for each edge in the menu, and will input letters many times as the hypotheses proliferate, so it's worthwhile to only do this work once.
2. Avoid memory allocations in a loop. The wiring matrix is a big 26×26 array and if you allocate a new copy on every loop iteration it will slow your program down a lot. Use a

[†]Each vertical stack of three drums on the Bombe is equivalent to a scrambler. The top drum performs the substitution of the right (fast) rotor, and the bottom drum is the left (slow) rotor. Reading across the row of scramblers, the fast rotor of each one is place further than the fast rotor of the previous scrambler: this encodes the offsets in the crib. Note that the middle and slow rotors are always in the same position on every scrambler; only the fast rotors differ.

single variable to store the wiring matrix and just set all the entries back to zero before the next iteration.

3. Use integers to represent letters rather than strings. You can make your code tidier (less `chr(((ord(c) - x + y) % m) + x))` and faster by using lists of numbers rather than characters. Write a function to convert a string `"ULTRA"` into its alphabet indices `[20, 11, 19, 17, 0]`, and another function to convert back. You'll notice that a substitution can now be represented as `key[letter]`. Indexing with integers helps performance. Keep in mind that `array.index(letter)` is doing a linear search.
4. Use multiple threads. The computer you are reading this on is probably capable of running 4 or more threads simultaneously. This basically requires splitting up the for loop described in the previous section into blocks which run in different threads. It's easier than it sounds!
5. Use (unsigned 32-bit) integers to represent each row of the wiring matrix. Since each cell is storing a boolean value, and each row has only 26 entries, we can use bitwise operations on a single integer value rather than indexing into a big array. A bit (no pun intended) like bitboards in computer chess[†].
6. Re-implement your code in an optimised compiled language like C or Rust.
7. Program an FPGA to do the attack[‡].

Conclusion

I hope you found this approachable and interesting! Best of luck with the challenges below, and if you have any questions or feedback please let me (upsidedown) know on the National Cipher Challenge forum.

I've only scratched the surface of the logistical challenges and awesome cryptanalytic prowess displayed at Bletchley Park and at the Polish Cipher Bureau.

See the references for more resources on the Bombe. Madness' book (predictably) has units on Enigma and the Bombe as well.

> In intellectual challenge and commitment of resources, the production of ULTRA is equalled only by the MANHATTAN project and putting a man on the Moon. Its effect on history ranks with either.
>
> — Graham Ellsbury

[†]Another fascinating rabbit hole! https://www.chessprogramming.org/Bitboards
[‡]This one's a joke :)

Challenges

Important: all of these challenges use reflector `B` and rotors `I`, `II` and `III`. *Bombe indicators* means the indicators of the first scrambler (which are actually a combination indicators — rings); the Enigma indicators would be one step before (because the rotors step immediately before each letter is encrypted).

You can use the following Python code to generate a proof which you can post on the National Cipher Challenge forum when you've solved one of them. The plugboard pairs must only include the pairs you obtain from the wiring matrix.

```
def md5(data):
  from hashlib import md5
  return md5(bytes(data, encoding="utf-8")).digest().hex()

def sorted_plugboard_pairs(plugboard):
  return ",".join(sorted(
    "".join(sorted(pair.upper()))
    for pair in plugboard
  ))

def proof(username, indicators, plugboard):
  secret = username
  secret += f"indicators:{indicators.upper()}"
  secret += f"plugboard:{sorted_plugboard_pairs(plugboard)}"
  return md5(secret)

proof(
  "upsidedown",
  "YAY", # indicators
  (
    "QW", "ER", "TY", "UI", "OP",
    "AS", "DF", "GH", "JK", "LZ",
  ),
)

# 0d9232f45a7b32cd3ceb8d7afba52cda
```

upsidedown-2025-4

I encrypted `belowthelunarsurface` with unknown rings and indicators, obtaining the ciphertext `GSHFCLFFCJLPYLCBEBFC`. The scrambler at offset 0 should be set to `ZZY`. The middle rotor did not turn. Find the plugboard pairs.

upsidedown-2025-5

I encrypted `icanfeelitmymindisgoing` with unknown rings, indicators, and plugboard settings, obtaining the ciphertext `GSJSEDXBPDONIVUBLEWRAXY`. The middle rotor didn't turn. Find the Bombe indicators and plugboard pairs.

upsidedown-2025-6

I encrypted `giveyoumycompleteassurance` with unknown rings, indicators, and plugboard settings, obtaining the ciphertext `HWTMKHYZHRIHFCKCPMQMZQJLQS`. The middle rotor turned during the crib.[†] Find the Bombe indicators and the plugboard pairs.

upsidedown-2025-7

I encrypted a plaintext with an unknown cipher which uses a 27-letter alphabet, obtaining an intermediate ciphertext. I then encrypted the result with Enigma[‡] with unknown rings, indicators, and plugboard pairs (full stops are unchanged by this step).

`SIAZZNXRFIKNKICVIQAQEHLPDSIEUAINFMIRFIHA` is a "crib" from the intermediate ciphertext, corresponding to `HNVQKJEHCHQIWVSIEBNBWWYAYHBYDPWGBCJMLUDO`. The middle rotor turned somewhere during encryption of this crib (it has 40 letters).

First find the Bombe indicators and plugboard pairings; you will need to split the menu. Then, if you solve the intermediate ciphertext, post `md5(YOUR_USERNAME + UPPER_PLAINTEXT)`.

```
RAQJP.NDIUPNNTRSK.KCWQFFXIIRTBSQLKBWDGISZGMBSOOHWGHNWFMVGYUSHZWK
FBDCFWBDOOOCXYSTJCZKHPRSEJLDOROYNWDNBJUBYQTYHLEHMDUKARV.LLFSA.JY
GTRGINZIVQCVUOYEUXZFNRZSSCFNPIDCSHATMOBNADJJWVJZQWAJKQCLPSNWVGDR
JQIYFNBAOXVTNUKJXHGEZQOXYKXCNIJZZBNOSFCB.GLPVEX.BVCRKFOOPABWSFBC
EJPUHZHNVQKJEHCHQIWVSIEBNBWWYAYHBYDPWGBCJMLUDOXOEFASWMVZRLYIAHFD
NWARIKQEEVDMVMRONVCFCYGWZKWGHVAQEBLJFIXUMPIMCZTBDKWNWUHPRWFCOY.E
XQFKZUEQJDCRMFFQBGOUNBAPHOJEZFOUPNUEZQPWYVPCXQM.KVRDQHGSH.EDOJCH
KCAJJHV.HJG.XRRXCEEUSVMJQWVRVOPFWKZSGFGSQTETLKVO.HKFSGMUCFKVAVXT
ETGVUTLLEMBUTUTXJDCUNYKWGSXXGCCPDOWDBVDWJKZOFFAGHBTNXWKUYUHXBRQJ
NTMPSIXJBZOOCLJGFSQLLQYWRCQMLBGVCNQMZTIZACCEQWHUJZVBUGYHUQJCZOZ.
XMKOQRKFXOKMCIVHZXAJULPYB.DGANEWXEEFOJBDDZCDLZDLMZMDAWTVKWWXRUHN
ATYT.MQVHSKVLTYBXCXUFYMKUWBJKFFEGPIXJXPXDKQKTGQDQ.DFGYR.SXSHJBMA
QAHCZKAFYTOHNUPSNJRQXSXVXYCRAKQCVCNEJIFZPRVOPFYYFMNMKYCXRBFYHLBF
KHXUDCHTGBJPAUQX.VIEHWIPJYDEOYBDHHSPPBLSZSEMUTEXMNWMKDMBRGLPAKDS
IYLNJCTKGNO.WPQIUCLQZDKVCATLUMEORUSFTLQXMTUUGULVUQQPKCWELKZCLHXQ
TQCGPCGYJKWRNWSREYLHFEGKJNDVWSXSLNRCVJL
```

[†] Split the menu in two, then one of the halves will not contain a middle rotor turn. You might want to try a few splits (not just down the middle) to find one with a good menu (a menu with loops).

[‡] Using reflector `B` and rotors `I`,`II`,`III` as usual.

What's next?

Many other incredible cryptographic feats (the stuff of legend) took place around this time.

The British attack on Enigma was built on the research of Marian Rejewski (*ray-eff-skee*). He reconstructed the wirings of the army Enigma machine from ciphertext-only analysis using a theory of permutations. He had a model of the commercial Enigma, which is identical to the military version except for the rotor wirings. I would love to recreate this analysis.

> The solution was Rejewski's own stunning achievement, one that elevates him to the pantheon of the greatest cryptanalysts of all time.
>
> — David Kahn, *The Codebreakers*

Similarly, the wirings, and operation, of the Lorenz machine (used by German high command) were derived from a single transmission error. Lorenz implements a Vernam (xor stream) cipher. A long message was transmitted twice with the same key, but many typos and abbreviated words. John Tiltman used this intercept to discern the encryption method, and break the two plaintexts and key apart. Bill Tutte was then able to derive the wirings of each of the 12 rotors, without ever seeing the machine.

References

1. *The Enigma and the Bombe* (web), Graham Ellsbury, ellsbury.com
2. *Enigma rotor details* (web), Wikipedia, en.wikipedia.org
3. *The Bombe Breakthrough* (book), Sir Dermot Turing and Dr David Kenyon, ISBN 978-1-841658-21-6
4. *Inside Enigma* (book), Prof. Tom Perera, ISBN 978-1-905086-64-1
5. *The Hut Six Story* (book), Gordon Welchman, ISBN 978-0-947712-34-1

Copyright