Anagrams

Anagrams are **transposition ciphers**. Encryption is by *permuting* the *locations* of the characters in the message, by contrast to *cryptograms* which permute the *alphabet*.

A **block-transposition cipher** breaks a larger message into uniform-sized blocks and applies the same mixing to each block.

For example, the first-to-last anagram reverses the order of the characters in a message, so that

`no business like show business`

becomes

`SSENISUB WOHS EKIL SSENISUB ON`

Decryption is by the same reversal process.

A fatal problem in attempting to decrypt an anagram is that there are typically many ways a jumble of letters can be arranged to form sensible English.

For example, `EEBDLYAORT` decrypts as

```
early to bed
barely dote
barely toed
barley toed
lo beet yard
lordy a beet
o lardy beet
lady to beer
o ready bet
or beet lady
to be dearly
```

This problem gets worse for larger messages. Indeed, the pastime of rearranging the characters of meaningful phrases in a natural languages into completely different but meaningful phrases is **anagramming**.

But anagrams are vulnerable to attack by **multiple anagramming**, in which the attacker has two or more messages encrypted with the same key.

Multiple anagramming uses two or more messages encrypted by the *same* permutation of positions.

The **contact method** is to *exclude* unlikely bigrams and try to *maximize* likely ones, done interactively until the list becomes manageable.

For example, among the 6 permutations of 3 characters applied *simultaneously* to `dog`, `cat`

```
dog cat
dgo cta
odg act
ogd atc
gdo tca
god tac
```

exclusion of bigrams `dg` and `gd` leaves only two:

```
dog cat
god tac
```

As with cryptograms, it is critical to run through the tree of possibilities as efficiently as possible, pruning whole branches of possibilities rather than individual leaves.

Note that transposition ciphers do *not* affect single-letter frequencies.

If the single-letter frequencies of a cipher text are the same as those of English we should infer that it *is* English, encrypted with a tranposition cipher.

Note that a text length more than twice the block length amounts to sending two messages with the same key, which is bad, in light of the mulitple anagramming attack.

Thus, with a text more than twice the block length, transposition ciphers are quite breakable.

As a simple but artificial example: how many *simultaneous* permutations of strings `PYOG` and `OPGY` do *not* have `P` adjacent to `Y` and do *not* have `O` adjacent to `G` in either rearranged string?

This is d**ouble anagramming**. Because the permutation is to act on *both* strings at the same time, the first characters `P` and `O` of the two strings will move together, as will the second characters `Y` and `P`, third characters `O` and `G`, and fourth characters `G` and `Y`. The forbidden adjacencies apply *simultaneously* to both these rearrangements.

First, what can be adjacent to P in the rearrangement of the first string? Y is prohibited, so it could only be O or G. But O in the first string is tied to G in the second, and G cannot be adjacent to O (tied to P) (in the second string). The only character adjacent to P in the first string can be G, and correspondingly (double anagramming) in the second string only Y can be next to O.

With only one character that can be next to P in the first string, P must be at the *end* or *beginning* of any legal permutation of the first string. Thus, in the first string, the character G must have another character next to it, in addition to P. Since O is prohibited, that other character must be Y. This drags along with it the P in the second string, which would then be adjacent to Y in the second string, which is prohibited. Thus, there are *no* simultaneous rearrangements which meet the conditions.

Permutations

Permutations of a given set may be viewed as *mixing* the set around.

The definition is that a permutation f of a set S is a *bijective function* $f : S \to S$.

Often we use a set of integers $S = \{1, 2, \ldots, n\}$ as a convenient choice for a set with n elements.

The standard notation for a permutation f of n things is to list the outputs under the corresponding inputs, with the inputs in order:

$$\begin{pmatrix} 1 & 2 & 3 & \ldots & n \\ f(1) & f(2) & f(3) & \ldots & f(n) \end{pmatrix}$$

For example, the permutation on $\{1, 2, 3, 4, 5\}$ which sends 1 to 2, 2 to 3, 3 to 4, 4 to 5, and 5 back to 1 is written

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 3 & 4 & 5 & 1 \end{pmatrix}$$

The **product** of two permutations f, g of a set S is simply the *composite function* $f \circ g$, defined by

$$(f \circ g)(s) = f(g(s)) \quad (\text{for } s \in S)$$

For example, the product

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 3 & 4 & 5 & 1 \end{pmatrix} \circ \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 3 & 4 & 1 & 5 & 2 \end{pmatrix}$$

is determined by following what happens to each of the 5 inputs. The right-hand permutation sends 1 to 3 (because 3 is below 1), and then the left-hand permutation sends that 3 to 4 (because 4 is under 3). Thus, the product sends 1 to 4.

Similarly, the right-hand one sends 2 to 4 (because 4 is under 2), and then the left one sends that 4 to 5 (because 5 is under 4). Thus, the product sends 2 to 5.

After looking at all 5 inputs, one finds

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 3 & 4 & 5 & 1 \end{pmatrix} \circ \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 3 & 4 & 1 & 5 & 2 \end{pmatrix}$$

$$= \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 4 & 5 & 2 & 1 & 3 \end{pmatrix}$$

The **do-nothing** or **identity** permutation of a set is the permutation that sends every element to itself. In the present notation it is written

$$\begin{pmatrix} 1 & 2 & 3 & \dots & n \\ 1 & 2 & 3 & \dots & n \end{pmatrix}$$

In the spirit of thinking of composition of permutations as a kind of *multiplication*, we use *exponential* notation for repeated application of a permutation:

$$f^n = \underbrace{f \circ \ldots \circ f}_{n}$$

The **order** of a permutation f is the smallest positive integer ℓ such that

$$f^n = \text{do-nothing permutation}$$

The usage of this word **order** *has a very precise technical sense, and must not be confused with colloquial uses!*

It may not be clear that there *is* such a number, but there is. (Its existence is a very special case of *Lagrange's Theorem* in group theory. Later.)

To determine the order of a permutation f, at worst we can use *brute force*. That is, successively compute f, f^2, f^3, and so on until one of these *iterates* is the do-nothing permutation.

Yes, this is potentially tedious.

For example, to determine the order of

$$f = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \end{pmatrix}$$

we first see that f itself is not the do-nothing permutation. Compute the square

$$f^2 = \begin{pmatrix} 1 & 2 & 3 \\ 3 & 1 & 2 \end{pmatrix} \neq \text{do-nothing}$$

and then the cube

$$f^3 = \begin{pmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \end{pmatrix}$$

which *is* the identity permutation, so

$$\text{order}(f) = 3$$

Disjoint cycle decomposition

There is internal structure to permutations, which incidentally makes computation of *orders* and other things easier.

A k**-cycle** is a permutation on n things (with $n \geq k$) which moves k things *in a 'cycle'* and does not move anything else. That is, there are distinct elements $s_1, \ldots, s_k$ such that

$$f(s_i) = \begin{cases} s_{i+1} & (\text{for } i < k) \\ s_1 & (\text{for } i = k) \end{cases}$$

and

$$f(s) = s \quad (\text{for } s \text{ not among the } s_i)$$

There is a separate notation for such a k-cycle

$$f = (s_1\ s_2\ \ldots\ s_k)$$

For example

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 4 & 1 & 3 & 2 \end{pmatrix}$$

is the 3-cycle also denoted

$$(1\ 4\ 2)$$

There are k different ways to write the same k-cycle in this cycle notation. The last 3 cycle is

$$(1\ 4\ 2) = (2\ 1\ 4) = (4\ 2\ 1)$$

Two cycles are **disjoint** if the two sets of elements they *move* are disjoint. For example, as permutations of the set $\{1, 2, \ldots, 10\}$, the two cycles

$$(1\ 4\ 5\ 9\ ) \quad (2\ 8\ 7\ )$$

are disjoint. The two cycles

$$(1\ 4\ 5\ 9\ ) \quad (2\ 8\ 9\ 6\ 7\ )$$

are *not* disjoint, since 9 is moved by both of them.

Theorem: every permutation of n things can be written as a product of disjoint cycles.

Any such expression is a **disjoint cycle decomposition**.

A further question is **how** to compute the disjoint cycle decomposition of a given permutation.

The method is *recursive*. To get started, given permutation of $1, 2, 3, \ldots, n$, compute the successive images $f(1)$, $f^2(1)$, $f^3(1)$, $\ldots$, until the first moment at which these successive images come back to 1 again, that is, $f^\ell(1) = 1$. Then the first cycle in the decomposition is

$$(1\ f(1)\ f^2(1)\ f^3(1)\ \ldots\ f^{\ell-1}(1))$$

Note that we do *not* repeat the 1 at the tail.

We continue *recursively*. Suppose that we have extracted some cycles from the permutation already (as we did above starting with 1).

Take the first index i in the list $1, 2, \ldots, n$ that has not already been included in a cycle. (If *nothing* is left, we're done!)

Compute $f(i)$, $f^2(i)$, $f^3(i)$, and so on, until the first time that these successive images come back to i again, that is, until $f^\ell(i) = i$. Then include the cycle

$$(i\ f(i)\ f^2(i)\ f^3(i)\ \ldots\ f^{\ell-1}(i))$$

in the cycle decomposition of f.

To determine the disjoint cycle decomposition of

$$f = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 7 & 1 & 4 & 3 & 2 & 6 & 5 & 8 \end{pmatrix}$$

track the successive images of 1, namely

$$\begin{array}{lllllll} f(1) & = & 7 & & & & \\ f^2(1) & = & f(7) & = & 5 & & \\ f^3(1) & = & f(f^2(1)) & = & f(5) & = & 2 \\ f^4(1) & = & f(f^3(1)) & = & f(2) & = & 1 \end{array}$$

so 1 is in the cycle (1 7 5 2).

The first element in the list $1, 2, \ldots, 8$ *not* in the 4-cycle including 1 is 3, which has iterated images $f(3) = 4$, and $f(f(3)) = f(4) = 3$, so we have a 2-cycle (3 4). The first leftover element is 6, which generates a 1-cycle, which does nothing, so we ignore it. The last leftover is 8, also generating a 1-cycle. Thus, the disjoint cycle decomposition is

$$f = (1\ 7\ 5\ 2\ )\ (3\ 4)$$

Theorem: The *order* of a k-cycle is its length k. The order of a product of cycles is the *least common multiple* (often abbreviated *l.c.m.* of their lengths.

The **least common multiple** of several integers $k_1, \ldots, k_t$ is the smallest positive integer M which is a multiple of every k_i. That is, $M > 0$ and $M \% k_i = 0$ for all indices i.

Thus, to compute the order of a permutation, it is often wise to determine its disjoint cycle decomposition and compute the least common multiple of the cycle lengths.

Note that it does not matter whether 1-cycles are included or not, since they do not move anything and also do not contribute to least common multiple computations.

How to compute *lcm*s?

Brute force is possible, but suboptimal.

A simple case is that **the l.c.m. of two numbers with no common prime factors is simply their product**.

Thus, for example, by the theorem, the *lcm* of 15 and 68 is their product, 1020, because 15 and 68 have no common prime factor.

How can we assert that 15 and 68 have no common prime factor?

For small integers, *factorization into primes* of the two integers (by trial division) and *comparison* of prime factors occurring might verify that two integers have no common prime factors: by trial division $15 = 3 \cdot 5$ and $68 = 2^2 \cdot 17$ are the prime factorizations.

(In real life, one would only use the Euclidean Algorithm to find common factors.)

More generally,

Theorem:

$$\mathrm{lcm}(m, n) = \frac{m \cdot n}{\mathrm{gcd}(m, n)}$$

where $\mathrm{gcd}(m, n)$ is the *greatest common divisor* of m, n.

The *gcd* of m, n is defined to be the largest integer d such that d divides both m and n (evenly), meaning that $m \,\%\, d = 0$ and $n \,\%\, d = 0$.

Ok, now how do we compute *gcd*s? Again, brute force is possible.

Also, looking directly at prime factorizations.

In real life, one would only use the Euclidean Algorithm to find *gcd*s.