

6. Project: 15-Puzzle

In this chapter, we consider the well known 15-puzzle where one needs to place in order 15 square pieces in a square box. It turns out that whether this puzzle can be solved or not is determined by mathematics: it is solvable if and only if the corresponding permutation is even. To understand what this means and why this is true, we will learn basic properties of even and odd permutations — an important tool in algebra and discrete mathematics. We will implement a number of simple methods for dealing with permutations. We will then use them as building blocks for a program that solves any configuration of this game in the blink of an eye!

6.1 The Puzzle

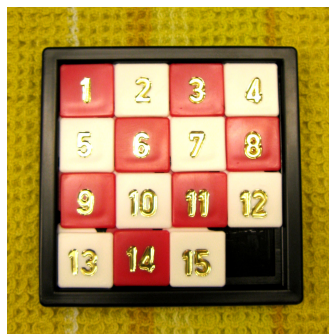


Figure 6.1: A photo of a 15-puzzle from [Wikipedia](#). You can see pins that prevent the pieces from being taken out from the box.

Take a square 4×4 box and put 15 square pieces numbered $1, 2, \dots, 15$ (Figure 6.1) in it. Each piece is of size 1×1 , so together the pieces occupy 15 places (cells) out of 16, and one cell remains empty. We can move the pieces as follows: a piece neighboring the empty cell moves into this cell, and its old position is vacated. The goal of this game is to rearrange the cells in some other order. For example, one may try to apply a sequence of moves transforming the left configuration from Figure 6.2 into the right configuration from Figure 6.2.

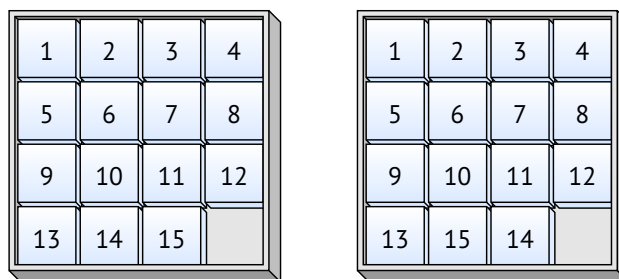


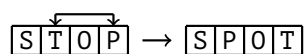
Figure 6.2: Can you transform the left configuration into the right one by several moves (not taking the pieces out)? The [Wikipedia](#) article says that once it was a \$1 000 question.

Equivalently, we may go backwards and ask you to restore the initial configuration (left) from the configuration where 14 and 15 are exchanged (right). This is an equivalent task (just reverse the sequence of the moves like in a video played backwards), that is more convenient in practice since you do not need to memorize the target configuration. [Try it online!](#)

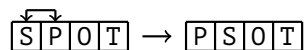
In fact, a sequence of moves required in Figure 6.2 does not exist at all! Thus, those who promised to pay for a solution took no risk. But how can we prove this? This proof is the final goal of this chapter, but first we should consider a more general object, *permutations*.

6.2 Permutations and Transpositions

Let us start with a sequence of letters (string), say, STOP. We can *exchange* two letters, for example, T and P. Then we get a string SPOT.



Now, we can exchange some other pair, for example, S and P, and get PSOT.



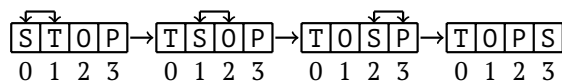
An exchange of two letters is called a *transposition*.

Stop and Think! Can you make one more transposition to get POST?

This is easy: exchange S and O.

Stop and Think! Can you convert STOP to TOPS by a sequence of transpositions (pair exchanges)?

Here is one such sequence.



We number the positions from left to right as 0, 1, 2, 3. Then, each transposition can be conveniently specified by two integers — indices of the swapped symbols. In particular, the transpositions applied above are

$$(0, 1), (1, 2), (2, 3).$$

The code below applies a sequence of transpositions to a string.

```
def apply(lst, trans):
    first, second = trans
    lst[first], lst[second] = lst[second], lst[first]

word = ['S', 'T', 'O', 'P']
transpositions = [(0, 1), (1, 2), (2, 3)]

for transposition in transpositions:
    apply(word, transposition)

print(word)
```

['T', 'O', 'P', 'S']

Here, we represent a sequence of objects to be permuted (letters in a string) as a Python list. For example, the string STOP is represented as a list of four elements 'S', 'T', 'O', and 'P' (one could use a string 'STOP', but strings are immutable in Python). This list is stored in the variable `word`, so, for example, `word[2]` is 'O' (recall that the indexing is 0-based).

The first argument of the function `apply` is a list, and the second argument is a pair of integers — the positions to be exchanged (in Python terms, pairs are “tuples” made of two elements). The line `first, second = trans` unpacks the pair `trans` and copies the two integers that constitute this pair into variables `first` and `second`. Note that we do not check whether `first` and `second` are different and the call `apply(lst, (0, 0))` will not cause any error (as long as `lst` is not empty).

Stop and Think! What does the call `apply(lst, (0, 0))` do?

In fact, nothing: the line `lst[0], lst[0] = lst[0], lst[0]` is perfectly legal in Python and doesn't change the values in `lst` (the new value `lst[0]` coincides with the old value).

However, when we talk about transpositions, we do *not* allow such “degenerate” transpositions: the *identity permutation* where all objects remain at their places, is *not* a transposition.

Finally, the `for` loop applies sequentially all the transpositions from the list `transpositions` to `word`; then we print the new (permuted) value of `word`.

Stop and Think! What happens with STOP after a sequence of transpositions `[(1, 3), (0, 1)]`?

It is exactly the sequence of transpositions we started with: STOP → SPOT → PSOT.

Let us emphasize again that we encode a transposition by a pair of *positions* of exchanged letters (and not by the pair of letters).

Problem 160 Find a sequence of transpositions that converts MARINE to AIRMEN. The positions are numbered 0, 1, 2, 3, 4, 5 from left to right. [Try it online!](#)

A systematic way to find the required sequence of transpositions is to fill positions correctly one by one, first making position 0 correct, then position 1 correct, etc. In our example, we first put A into position 0, i.e., exchange letters at positions 0 and 1.

M	A	R	I	N	E	→	A	M	R	I	N	E
0	1	2	3	4	5		0	1	2	3	4	5

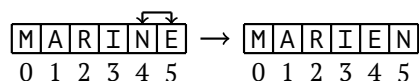
Then we fill position 1 with letter I by applying the transposition (1, 3):

A	M	R	I	N	E	→	A	I	R	M	N	E
0	1	2	3	4	5		0	1	2	3	4	5

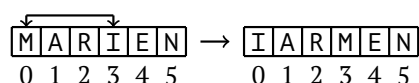
Now we are lucky: positions 2 and 3 are already filled correctly by R and M. But position 4 contains N instead of E, so we need to apply transposition $(4, 5)$. After that, position 4 contains E as required. Moreover, the last position 5 is also filled correctly: N was moved there from position 4, so nothing else is needed.

Stop and Think! Is the solution unique? Are there other sequences of transpositions that also transform MARINE to AIRMEN?

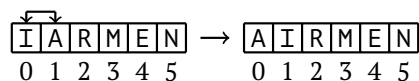
Sure, there are other sequences. For example, we could go from right to left and first make the *last* position correct:



Then we make position 3 correct:



Position 2 is already correct, so we make position 1 correct:



Stop and Think! Can you represent this sequence of transpositions in our Python notation?

Answer: $[(4, 5), (0, 3), (0, 1)]$. There are other equivalent answers; for example, $[(5, 4), (0, 3), (1, 0)]$ represents the same sequence of transpositions, since the ordering in pairs does not matter.

There are many other sequences of transpositions that convert MARINE to AIRMEN. For example, one could append some transposition twice at the end; such two transpositions cancel out.

Problem 161 Prove that *any* permutation can be obtained by a sequence of transpositions.

In fact, our approach to the AIRMEN example was general enough and it will work for every permutation. We fill all the positions one by one in some order (in our example, from left to right), until all the positions are filled correctly.

More formally, we prove by induction that the first k positions can be filled correctly. For $k = 0$, this statement is (vacuously) true. Induction step: assume that positions $0, 1, \dots, k-1$ (there are k of them) are already filled correctly. We need to fill the next position, i.e., position k , by a correct letter. It may happen that it is already filled correctly. Then we have nothing to do at this step. If position k is occupied by a wrong letter, we exchange this wrong letter with the letter we want to see at position k . After this transposition, the position k is filled correctly. (End of the induction step.)

Mathematicians would say that in the *permutation group* every permutation is a *product* of some transpositions.

Stop and Think! Is this argument convincing? If it is, could you tell what happens at the last step when we need to fill the last position correctly? And why don't we destroy any of the previous positions $0, 1, \dots, k-1$ while filling the k -th position with a correct letter?

The first question is easy. At the last step, all the positions, except for the last one, are filled by correct letters. So only one letter and only one position remain, and there is no choice.¹

¹If a group of people get back their passports after a passport control, and everybody except one person have checked that they got the correct passport, then this last person does not need to worry either.

The second question: assume that positions $0, 1, \dots, k-1$ are filled correctly. We need to put the correct letter in the k -th position if it is not already there. Can we destroy positions $0, 1, \dots, k-1$ while doing this? No, because they are filled by the letters whose target positions are $0, 1, \dots, k-1$, and we consider a letter whose target position is k , and its current position is also greater than $k-1$.

Problem 162 Prove that any permutation of n objects can be obtained by at most $n-1$ transpositions.

Problem 163 Give an example of a permutation of 4 objects that cannot be obtained by 2 or fewer transpositions. Give an example of a permutation of 10 objects that cannot be obtained by 8 or fewer transpositions.

Note that in this problem it is not enough to provide an example. We have to prove that *this permutation has the required property* (cannot be obtained by a small number of transpositions). For the first part of the problem (2 or fewer transpositions) it is easy to check all the possibilities, but for the second part (8 or fewer transpositions) some general argument is needed, and this is not so easy.

Problem 164 Implement a Python function

```
transform(first, second)
```

that takes two permutations of $\{0, 1, \dots, n-1\}$ (i.e., two lists containing each integer from $\{0, 1, \dots, n-1\}$ exactly once, in any order) and returns a list of transpositions that transforms the first list into the second one.

A solution to this problem is shown below.

```
def transform(first, second):
    assert len(first) == len(second)
    n = len(first)
    assert set(first) == set(range(n))
    assert set(second) == set(range(n))

    transpositions = []
    current = list(first)

    for i in range(n):
        if current[i] != second[i]:
            idx = current.index(second[i])
            assert idx != i
            transpositions.append((i, idx))
            current[i], current[idx] = \
                current[idx], current[i]

        assert current[i] == second[i]

    return transpositions

print(transform([3, 4, 0, 2, 1], [0, 1, 3, 4, 2]))
```

```
[(0, 2), (1, 4), (3, 4)]
```

First, we check that both input lists are indeed permutations of $0, 1, \dots, n-1$ for some n . Then, we prepare the transpositions list that is empty initially; the transpositions will be added to it one by one. We also make a copy of the list `first` and call it `current` (note that it is essential to call the method `list()` for this).

Then, for $i = 0, 1, \dots, n-1$, we fill the i -th position of `current` correctly. Our goal is to move `second[i]` there. It may be already there; if not, we use the built-in method `index` to find the position where `second[i]` appears in `current`. This is the position that needs to be exchanged with i (and it should be different from i due to our `if`-check). We add the corresponding transposition (i.e., pair) to the list `transpositions`, and apply it to `current`. After that, we check that our goal is achieved: `current[i]` is equal to `second[i]`. (In fact one could check more: i first elements are the same, so `current[:i+1]` should be equal to `second[:i+1]`.)

Problem 165 Replace the line “for `i` in `range(n)`.” by “for `i` in `range(n-1)`.”. Is this algorithm correct too?

6.2.1 Counting Permutations and Transpositions

Using two letters A and B (once each), we can get two strings: AB and BA.

Stop and Think! How many strings can we get using letters A, B, and C (once each)?

The first letter can be A, B, or C. First we list strings starting with A, then with B, then with C. For each first letter, we have (as we discussed above) two possibilities for the remaining two letters:

ABC, ACB; BAC, BCA; CAB, CBA.

So the answer is 6.

Problem 166 How many different strings can be made using 4 letters (once each)? Can you answer the same question for n letters?

You may want to run the following code to list all permutations of four letters.

```
from itertools import permutations
print(list(permutations('ABCD', 4)))
```

Now we are interested in the *number of transpositions needed to obtain a given permutation*. Our goal is the following result that will be used in the analysis of 15-puzzle:

Theorem 6.2.1 One cannot return to the original ordering after an *odd* number of transpositions.

In other words, if we have n different objects at n positions, and after k transpositions we come back (all objects are again where they were initially), then k is even (a multiple of 2, i.e., $k = 0, 2, 4$, etc.). Recall that a “transposition” means that two *different* objects exchange their positions.

Mathematicians would say that the *identity* permutation (each element remains in its place) is not a product of an odd number of transpositions.

Stop and Think! Can you prove this result for small number of objects, e.g., for $n = 1, 2$, or 3?

For $n = 1$ (one object), there are no transpositions at all, since for a transposition we need two different objects. So our statement is *vacuously* true: no transpositions means no odd sequences of transpositions.

Stop and Think! What does Theorem 6.2.1 say for $n = 2$?

For $n = 2$, there is only one transposition (no choice: we have only one pair of different objects that can be exchanged). Applying this transposition several times, we go back and forth:

$$AB \rightarrow BA \rightarrow AB \rightarrow BA \rightarrow \dots$$

Thus, an odd number of transpositions (1, 3, 5, etc.) is equivalent to one transposition, and an even number of transpositions (0, 2, 4, etc.) returns us to the original order. Hence, Theorem 6.2.1 is proven for $n = 2$.

Stop and Think! Can you prove Theorem 6.2.1 for $n = 3$?

We have six possible orderings, and three pairs of positions that can form a transposition, namely, $(0, 1)$, $(0, 2)$, and $(1, 2)$. Each transposition changes the ordering of letters, and we have to prove that we can return to the original ordering only after an *even* number of transpositions. This looks a bit messy; can we phrase this in a more structured way?

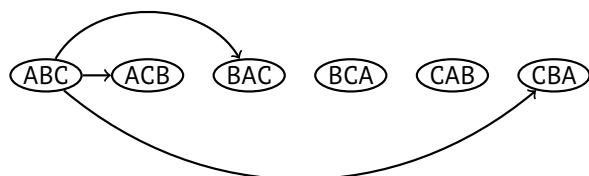
Let us start by listing all the six orderings (using letters A, B, and C):



This picture does not help much, since we do not show the effect of transpositions. Imagine that we start with ABC and apply one transposition. We may get BAC after $(0, 1)$, CBA after $(0, 2)$, and ACB after $(1, 2)$.

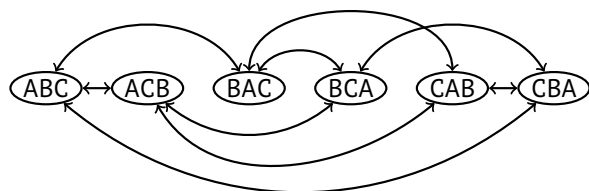
Stop and Think! How would you visualize this information?

Probably the easiest way is to add arrows that show all three possibilities (we do not show the transpositions, only their result):



Stop and Think! Since we need to study the effect of multiple transpositions, we should add arrows starting from other orderings too. Can you do this?

This requires some patience, but not much: for each vertex (three-letter string), we have three outgoing edges that correspond to three possible transpositions:



Stop and Think! In the picture above, all edges are bidirectional — can you explain why this happens? How many edges does each vertex have? Why?

If a string Y can be obtained from X by some transposition, and we apply the same transposition to Y , we come back to X (two exchanges of the same objects cancel each other). So, an arrow $X \rightarrow Y$ is accompanied by an arrow $Y \rightarrow X$. Note also that we have three transpositions $(0, 1)$, $(0, 2)$, and $(1, 2)$, and they transform each string X to three other strings (all different), so we have three bidirectional edges adjacent to each vertex.

Now, having this picture (a *transition graph*), let us recall what we need to prove. We need to prove that we cannot return to the original string after an odd number of transpositions. In terms of this picture: *we cannot come back to the initial vertex after an odd number of moves*. (Mathematicians would say that this graph doesn't have odd cycles.) It is not yet clear why this picture helps at all...

Instead of giving up, let us look at the picture more closely. Where can we go from, say, ABC in one, two, or three steps? For one step, we have three possibilities (connected to ABC by edges):

one step: ACB, BAC, CBA.

How can we find vertices that can be reached in two steps? We look at the vertices reachable in one step, and make one more step. We need to check three neighbors for each of three vertices reachable in one step. In fact (check this carefully!), these three neighbors are the same for all three vertices, and we get the answer:

two steps: ABC, BCA, CAB.

Stop and Think! How can we find vertices that can be reached from ABC in three steps?

To do this, one should look at all neighbors of all vertices reachable in two steps. It gives

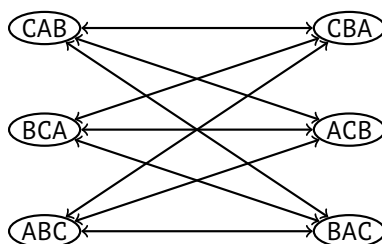
three steps: ACB, BAC, CBA.

Stop and Think! How can we find vertices reachable from ABC in four steps, in five steps, and so on?

In fact, we already have all the necessary information. Looking at our data, we observe that the lists for one and three steps are the same. Hence, taking all neighbors, we get the same lists for two and four steps, and our sequence starts repeating itself. Then we get the same lists for three and five steps, etc. This gives us a general answer:

1, 3, 5, 7, ... steps: ACB, BAC, CBA
2, 4, 6, 8, ... steps: ABC, BCA, CAB.

This behavior becomes even more clear if we reposition the vertices of our graph keeping the same edge connections:



(Check that we do not cheat and connections are the same!)

Stop and Think! Do you see why we come back to the original configuration only after an even number of moves?

Each move brings us from the left part to the right part or back, hence to return to the same part (left or right), we need an even number of moves. Thus — finally! — we have proved Theorem 6.2.1 for $n = 3$.

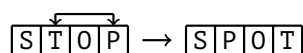
Problem 167 Prove Theorem 6.2.1 for $n = 4$ using the same idea.

It is boring but not that difficult: we need to draw a similar graph that classifies 24 possible strings (made of ABCD by permutations) into left and right part in such a way that every edge connects left and right part (no edges between two left or two right vertices). But this approach doesn't generalize to a general argument that can be used for every n .

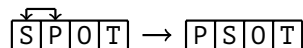
We will soon prove the general result, but some preparation is needed: we should look at *neighbor transpositions*.

6.2.2 Neighbor Transpositions

Recall what we started with: permuting letters in a string by exchanging pairs of letters. We called this exchange a *transposition*. If a transposition exchanges neighbor letters, we call it a *neighbor transposition*. For example, our first example,



is not a neighbor transposition: we exchange T and P that are not neighbors (there is O between them). On the other hand, our second example



is a neighbor transposition: the letters S and P that are exchanged are neighbors.

Stop and Think! We denoted a transposition by a pair of numbers (i, j) of exchanged positions. When (i, j) is a neighbor transposition?

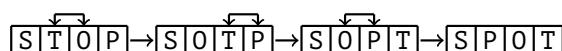
This is an easy question — just to check that we are on the same page: i and j should differ by 1; one can write this condition as $|i - j| = 1$. Note that we require $i \neq j$ for every transposition (we cannot exchange a letter with itself).

Stop and Think! We proved that every permutation can be obtained by a sequence of transpositions. Is it true that every permutation can be obtained by a sequence of *neighbor* transpositions?

The answer is yes, and we will see why soon. But let us first look at our example.

Problem 168 Represent the permutation $STOP \rightarrow SPOT$ as a sequence of neighbor transpositions.

If two people in a queue are not neighbors, they can still exchange their positions by moving along the queue and back. Here the letter T moves to the right (exchanging with O), then it makes an exchange with P, and then P moves back to the left:



In this example, we had one letter between the letters T and P that we needed to exchange. The same trick works for cases with several letters in between, see Figure 6.3.

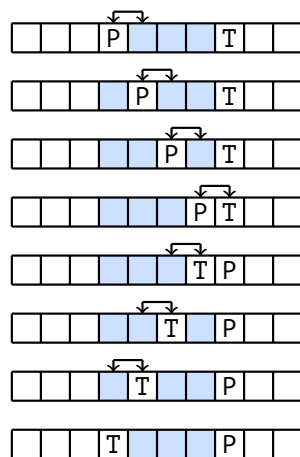


Figure 6.3: Representing an arbitrary transposition as a sequence of neighbor transpositions. The intermediate elements are shaded: P and T jump over them.

Problem 169 How many neighbor transpositions are used in this construction if there are m letters between the pair of letters that we want to exchange?

There are three stages:

- T crosses m intermediate letters by m neighbor exchanges (=transpositions) and becomes a neighbor of P (in Figure 6.3, $m = 3$);

- neighbor transposition of T and P;
- P moves to the left crossing the same m letters (m neighbor exchanges).


In total, we have $2m + 1$ neighbor exchanges. Note that this number is odd; this fact will be used later. Now we only need to know that a non-neighbor transposition can be replaced by a sequence of neighbor transpositions.

Problem 170 Prove that every permutation can be obtained as a sequence of *neighbor* transpositions.

Indeed, we know already that every permutation can be obtained by a sequence of transpositions; it remains to replace each non-neighbor transposition by a sequence of neighbor transpositions.

Another argument: let us show that any order can be reduced to any other by neighbor transpositions. Let us imagine n people in a queue that have different weights, and choose the weights in such a way that the desired ordering is weight ordering (maximal weight is assigned to the first person in the desired ordering, etc.). Then, if an ordering is incorrect, there is a place where a heavier person stands just behind a lighter one. Exchange them (they are neighbors) and repeat this procedure until the ordering is correct. This procedure is called *bubble sort*.

One needs to prove that this process terminates. For that we note that the center of gravity of the people moves forward at each step and there are only finitely many orderings.

Problem 171 Find a sequence of *neighbor* transpositions that converts MARINE to AIRMEN. [Try it online!](#) 

Problem 172 Implement a Python function

```
transform(first, second)
```

that takes two permutations of $\{0, 1, \dots, n-1\}$ and returns a list of *neighbor* transpositions that transforms the first list into the second one.

Here is a function that converts *one* transposition into a sequence of neighbor transpositions.

```
def convert(transposition):
    i, j = transposition
    assert i != j
    i, j = min(i, j), max(i, j)

    return [(s, s + 1) for s in range(i, j)] + \
           [(s, s + 1) for s in
            reversed(range(i, j - 1))]

for transposition in [(2, 3), (2, 5), (5, 2)]:
    print(transposition, '->', convert(transposition))
```

```
(2, 3) -> [(2, 3)]
(2, 5) -> [(2, 3), (3, 4), (4, 5), (3, 4), (2, 3)]
(5, 2) -> [(2, 3), (3, 4), (4, 5), (3, 4), (2, 3)]
```

6.2.3 Proof of the Main Theorem

Theorem 6.2.1 claims that we cannot return to the initial ordering after an odd number of transpositions. In other words, if we return to the initial ordering after k transpositions, then k is even.

We've proved this result for permutations of n elements for $n \leq 3$, but had no general argument. In this section, we give a proof for arbitrary n using the notion of a neighbor transposition.

We start by proving the special case of the theorem that allows only *neighbor* transpositions: *If we return to the initial ordering after k neighbor transpositions, then k is even.*

Stop and Think! Do you see why this is true?

To visualize the statement, imagine a queue with n people. At each step, two neighbors in the queue exchange their positions while others are standing still. After k steps, all people return to their initial positions. We need to prove that k is even.

A crucial idea: let us look at some pair of people in the queue, say, Alice and Bob, and consider only the exchanges that involve both Alice and Bob. At these steps, Alice and Bob exchange their places, so we call them " A - B steps". During these steps, Alice and Bob are neighbors (both before and after the exchange).

Stop and Think! Do you see why the number of A - B steps is even if the original ordering is restored at the end?

To see why, note that Alice may be either before Bob or after Bob in the queue. There could be people between them; we just look who is closer to the beginning of the queue, Alice or Bob. In the first case, we write $A < B$, in the second case we write $A > B$. When Alice and Bob exchange their positions (=during an A - B step), the situation changes: $A < B$ becomes $A > B$ and vice versa. When some other people exchange their positions, or even when Alice or Bob exchanges with somebody else, $A < B$ remains $A < B$ and $A > B$ remains $A > B$. (Do you see why?) But we know that at the end, Alice and Bob are where they were initially, hence the mutual ordering is the same at the beginning and at the end. Therefore, the total number of flips back and forth is even.

Stop and Think! Do you see why the total number of steps is even?

Let us classify all the exchanges according to their participants: for each pair X, Y of (different) people we consider X - Y exchanges. As we have seen, each class (for fixed X and Y) consists of an even number of exchanges. Thus, the total number of exchanges (=neighbor transpositions), being the sum of these even numbers, is even.

We proved theorem 6.2.1 for the special case of *neighbor* transpositions.

Stop and Think! Do you see how the general case can be reduced to this special case?

Recall how we replace one non-neighbor transposition by $2m + 1$ neighbor transpositions (where m is the number of elements in between). This replacement increases the total number of transpositions by $2m$, i.e., by an even number. So if the number of transpositions was even (or odd), it remain even (respectively, odd). Now assume that a sequence of transpositions returns us to the original ordering. Replacing non-neighbor transpositions by neighbor ones, we change the number of transpositions by some even number. And our special-case theorem says that after the replacement the number of transposition is even. Therefore, it was even before the replacement too. Theorem 6.2.1 is proven.

Problem 173 Consider the following sequence of transpositions:

TASTE \rightarrow ATSTE \rightarrow ATSET \rightarrow AESTT \rightarrow TESTA \rightarrow TASTE

We returned to the original configuration after 5 transpositions, and we have shown that the number of transpositions must be even. How is this possible?

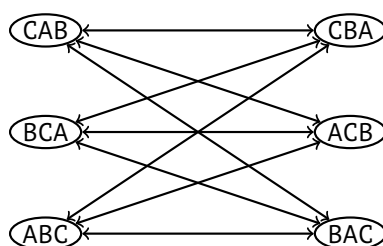
We have two letters T, and in fact these letters in the final ordering changed their places, so the resulting permutation is the $(0, 3)$ transposition, not the identity permutation. To return to the truly original ordering, we should add this $(0, 3)$ transposition, thus getting 6 transpositions (an even number, so everything is OK).

We should be careful and always apply permutations to distinct objects to avoid situations like this one.

We conclude by an important remark. The notion of neighbor transpositions makes sense if the permuted objects form a line (like characters in a string, or elements in an array, or people in a queue). The argument we used to prove Theorem 6.2.1 is valid only for this case. However, the statement of this theorem does not use this line ordering, and the theorem is valid for any n objects positioned in n different places. Indeed, we can create an imaginary line (curve) that goes through all the positions. In other words, we may number the positions by integers $1, 2, \dots, n$ and consider the exchanges between positions i and $i + 1$ as neighbor transpositions. (We will return to this discussion when applying our theory to 15-puzzle, in Section 6.3.)

6.2.4 Even and Odd Permutations

Recall the classification of permutations of n objects (different letters) for $n = 3$:



On the left, we have the original string ABC and all other strings that one gets after an even number of transpositions; on the right, we have strings that are obtained by an odd number of transpositions. The permutations that correspond to the left part (that are obtained by an even number of transpositions) are called *even*, and the permutations that correspond to the right part (that are obtained by an odd number of transpositions) are called *odd*.

The use of words *even* and *odd* for permutations may be confusing at first: this has nothing to do with the number of objects (n) being even or odd.

To make this classification well defined, we need to prove the following:

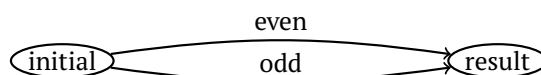
Theorem 6.2.2 If some permutation is obtained by an even number of transpositions, it cannot be obtained by an odd number of transpositions.

This theorem shows that these two classes of permutations (obtained by even an odd number of transpositions) are disjoint. Hence, we may call them *even* and *odd* permutations and no permutation can be even and odd at the same time.

Stop and Think! Is it true that every permutation is either even or odd?

Yes: we know that every permutation can be obtained by a sequence of transpositions, and this sequence may include even or odd number of transpositions. The problem would appear if one sequence of transpositions is even (i.e., consists of an even number of transpositions) while another is odd, and we need to prove Theorem 6.2.2 to exclude this case.

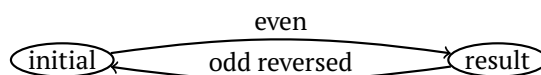
Imagine that the claim is false and there are two sequences of transpositions, one even and the other odd, that lead to the same ordering:



Each arrow denotes a sequence of transpositions; one contains an even number of transpositions, and the other contains an odd number of transpositions.

Stop and Think! Do you see a reason why this cannot happen?

Let us apply first the even sequence of transpositions, and then the odd one, but *in the reversed order*. In other words, append to the video of the even sequence the video of the odd one going backwards.



Since both sequences (even and odd) end at the same position, we get a consistent picture (no sudden changes in the middle of the combined video). In other words, we now have a sequence of transpositions that starts and ends in the same position, and the total number of transpositions is even + odd, which is odd. This is impossible by Theorem 6.2.1. Theorem 6.2.2 is proven.

6.2.5 Finding the Parity of a Permutation

Given a permutation, how can we find whether it is even or odd? How could one implement a function that checks whether a given permutation is even or odd? For convenience, let us assume that n objects that are permuted are integers from $\text{range}(n)$, i.e., $0, 1, 2, \dots, n-1$.

Stop and Think! Is the permutation $[4, 0, 1, 2, 3]$ even or odd?

To answer this question, we need to know how many transpositions are needed to get this ordering from $[0, 1, 2, 3, 4]$. We know how to represent a permutation as a sequence of transpositions (placing correct values one by one). For this example, we could do the following.

$[0, 1, 2, 3, 4] \rightarrow [4, 1, 2, 3, 0] \rightarrow [4, 0, 2, 3, 1] \rightarrow [4, 0, 1, 3, 2] \rightarrow [4, 0, 1, 2, 3]$.

We used four transpositions, so the permutation is even.

Problem 174 Implement a program that gets a list of length n that consists of the integers $0, 1, \dots, n-1$ in some order, and returns True if and only if the corresponding permutation is even. [Try it online!](#)

To solve this problem, one may use the program that computes a sequence of transpositions that transform one list into the other one.

Programmers could prefer another approach: we can go backwards (the number of transpositions is the same) and sort the given array by an exchange sorting algorithm (at each step, two elements are exchanged). Since our initial configuration is sorted, returning to the initial configuration means sorting. Then, the permutation is even if and only if the number of exchanges (made by the sorting algorithm) is even.

What are the advantages of this approach? The experts in algorithms design would say that our original approach requires about n^2 steps for a permutation of length n : we need to put n objects in place, and each object should be found in the list (which requires up to n comparisons, even if this is masked by a Python index operation).

On the other hand, there are sorting algorithms that are faster (require about $n \log n$ operations), and they can be adapted to provide a sequence of transpositions. This gives a faster way to check whether a permutation is even or odd. Still, this is not the optimal approach: there are methods where the number of operations is proportional to n .

Problem 175 Find an algorithm that determines whether a permutation of $\{1, \dots, n\}$ is even or odd using $O(n)$ operations.

Hint: one can decompose a permutation into cycles; another possibility is to keep the permutation and its inverse to avoid wasting time on index operations.

6.3 Why 15-puzzle Has No Solution

We are ready to return to the 15-puzzle and show that no sequence of game moves can exchange 14 and 15 (transform the left configuration into the right one shown in Figure 6.4).

For the proof, let us replace the empty cell by a special dummy piece 0, see Figure 6.5 (a). After this, pieces cannot move. If in the original game we move, say, 12 down, in this new representation we exchange 12 and 0, see Figure 6.5(b)–(c). Every move in the original game is now a transposition in the new game (exchanging the moving piece with 0-piece).

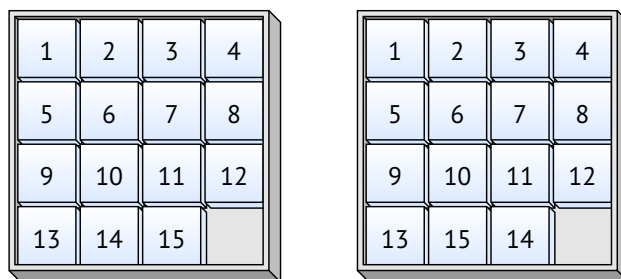


Figure 6.4: One cannot go from the left configuration to the right one, and soon we will see why.

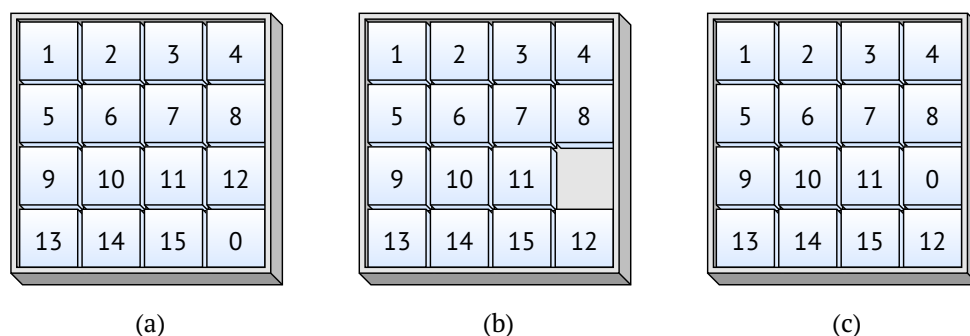


Figure 6.5: Empty cell replaced by 0-piece (a). A move in the original game (b) is now an exchange with the 0-piece (c).

Stop and Think! If the original 15-puzzle (where one needs to exchange 14 and 15) were solvable, the solution would require an odd number of moves. Do you see why?

This is a direct corollary of what we know about even and odd permutations. The exchange of 14 and 15 corresponds to a transposition in the new representation (as permutations of 16 pieces, including the dummy piece), so the required permutation is odd. This means that if this permutation is obtained by a sequence of moves, and each move is a transposition, then the sequence should contain an odd number of moves (otherwise we would get an even permutation).

Here, as we have discussed above, the permuted objects do not form a line. This does not invalidate the distinction between even and odd permutation. But if you prefer to deal with a list, write down the puzzle configuration as a list, in a reading order. The move shown in Figure 6.5 will then transform the list

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 0, 13, 14, 15, 12]

into

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 0].

On the other hand, a different argument shows that any solution of the 15-puzzle should have an *even* number of moves. For that, we do not need to look at the numbers; only the position of the empty cell is important.

Stop and Think! Do you see why the number of moves in the 15-puzzle must be even?

If we ignore the pieces and look only at the position of the empty cell, we see that at each move the empty cell makes one step in some direction (left, right, up or down). At the end of the game,

the empty cell should return to its original place, the right lower corner. Therefore, the number of steps left must be equal to the number of steps right ($\text{left} = \text{right}$); also the number of steps up must be equal to the number of steps down ($\text{up} = \text{down}$). Thus, the total number ($\text{left} + \text{right} + \text{up} + \text{down}$) is even (being equal to $2 \cdot \text{left} + 2 \cdot \text{up}$).

Stop and Think! Now we have two arguments: one is based on permutations and shows that the number of moves in a solution of the 15-puzzle must be odd; the other one uses the empty cell position and shows that the number of moves must be even. Is mathematics inconsistent?

Of course not: we proved only that *if a solution to this puzzle existed, it would* require an even number of moves — and, at the same time, an odd number of moves. This is a contradiction, but this contradiction proves only that there is no solution. Thus, we have proven that the puzzle (exchanging 14 and 15 by legal game moves) is unsolvable.

6.4 When 15-puzzle Has a Solution

In this section, we consider a general question: what configurations in the 15-puzzle are solvable? The answer is provided by the following criterion.

Theorem 6.4.1 Consider a configuration X in the 15-puzzle where the empty cell is located in the bottom right corner. The configuration X can be achieved from the initial configuration if and only if X is an even permutation of the initial configuration.

The statement of this theorem needs some clarification. The configuration X can be considered as a permutation of 15 pieces. Or we may add a dummy piece 0 and consider X as a permutation of 16 pieces, including the dummy one (which happens to stay in the bottom right corner). Which of these two permutations is used in Theorem 6.4.1?

The answer is that *this does not matter*. If X is even or odd as a permutation of 15 pieces, this means that X can be obtained by an even (resp. odd) number of transpositions (pair exchanges between these 15 pieces). The same transpositions can be considered as transpositions for the full board (with a dummy piece) that give X . In general, an even/odd permutation remains even/odd when we add one object that does not move.

How can we prove Theorem 6.4.1? In one direction, the proof repeats the argument we have already seen. If a sequence of moves transforms the original configuration into some other configuration that has the empty cell in the bottom right corner, then this sequence contains an even number of moves (since the empty cell returns to its place). Therefore, we get an even permutation of 16 pieces (including the dummy one) and, as we have mentioned, an even permutation of $1, 2, \dots, 15$.

The other direction is more difficult and some preparations are needed. Let us start with a simple question (just for a warm-up).

6.4.1 Moving the Empty Cell

Problem 176 Prove that you can move the empty cell wherever you want: for every configuration C (empty cell may be anywhere) and for every desired position p of the empty cell on the game field, there exists a sequence of moves that bring the empty cell to the position p (starting from configuration C).

To see why this is true, we ignore all the labels, and follow the position of the empty cell only. As we mentioned, we may move the empty cell left, right, up, and down. Of course, physically we move not the empty cell but the piece that is now placed in the cell where the empty cell is moving to.

Using these moves (left, right, up, and down) we can bring the “hole” (empty cell) wherever we want.

6.4.2 Moving a Given Piece to the Top Left Corner

Our next task will be more complicated. Imagine that you have chosen some piece and you want to bring it to, say, the top left corner. (Initially this piece can be anywhere, and the position of the hole may also be arbitrary.)

Problem 177 Prove that this is always possible.

If you've played this game a few minutes, this is perhaps already clear to you: there is a lot of movement freedom if we are interested only in one piece and do not care about the others. However, how could we prove this rigorously for any configuration? Or how can we implement a function that computes the required sequence of moves?

A tool that is useful here is a *cyclic move*. Let us explain what a cyclic move is using a metaphor. Put 16 chairs around a table, as it is often done for meetings. Imagine that one of 16 persons did not come for the meeting. So there is a “hole”: one chair is empty, and 15 others are occupied by the participants. Assume that the only allowed move is when someone moves to the empty chair near her (so that others are not disturbed). Figure 6.6 visualizes a single move.

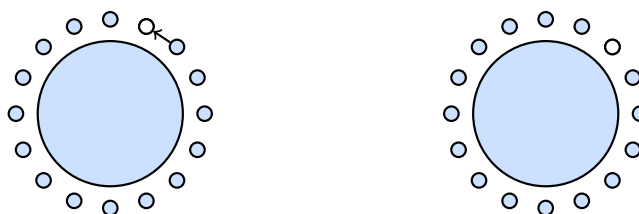


Figure 6.6: There are 16 chairs around a table, one of them is vacant. A person next to the vacant chair moves to this empty chair counterclockwise. As a result, the vacant chair (a “hole”) moves clockwise.

Stop and Think! How can we bring any participant to any chair using only allowed moves?

This is easy: if we start a wave of movements in some direction (I move right to the empty chair, my left neighbor moves right to my place, her left neighbor moves to her place, etc.), we get a full cycle where everybody just moved one chair to the right. Repeating this cycle, we can move any given participant to any given place.

Stop and Think! How does this remark help us? Recall that we need to bring a given piece in a 15-puzzle configuration to the top left corner.

When we speak about a cyclic movement, the cycle does not need to be a physical circle. We can draw an imaginary cycle (a “snake”) on the board, see Figure 6.7 (a).

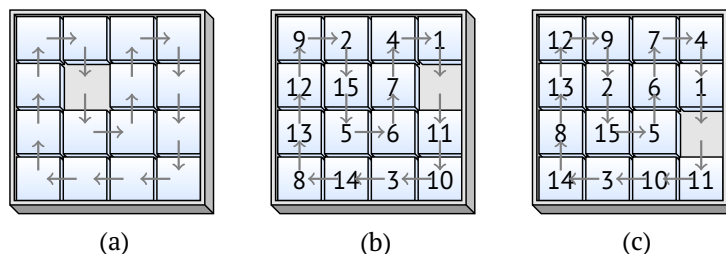


Figure 6.7: (a) A cycle through all the cells of the board. (b) A particular configuration. (c) The result of a cyclic move applied to this configuration. To perform the cyclic move, one first moves 1 down (and the hole moves up), then 4 moves into the new hole (a cell vacated by 1), etc.

Stop and Think! How many moves are needed to complete one full cycle?

Since every piece should move one step along the cycle, we need 15 moves.

Stop and Think! How many cycles are still needed (after the first one, Figure 6.7 (c)) to bring 3 to the top left corner?

Looking at the arrows, we see that after the first cycle (that moved 3 left), we need four more cycles to put 3 in the top left corner.

Of course, the same *U*-shaped cycle can be used for any configuration and any position of the hole. This way, we are able to move any given piece to the left top corner, as requested.

Stop and Think! What piece will be to the right of the hole after four more cycles, when 3 is in the top left corner?

To get the answer, we can draw all the pieces along the cycle, starting from 3 and keeping the cycle ordering. We get the following configuration (Figure 6.8; check the cycle ordering). Thus, to the right of the hole we have 1.

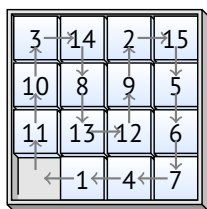


Figure 6.8: After five cycles, 3 is in the top left corner and the cycle ordering remains the same.

6.4.3 Moving One More Piece

It is time to think where we are, comparing what we wanted to achieve and what we have achieved. Our goal was to implement an arbitrary permutation, i.e., to put all 15 pieces in the right places — assuming that the permutation is even. What we have achieved so far is more modest: we know how to fill the top left corner with a given piece. Still, it is a step in the right direction. After the top left corner is filled with a correct piece, we may decide that we never move that piece again, and fill correctly some other cell, for example, the cell below the top left corner.

Problem 178 Prove that it is always possible: if we glue the top left corner piece and never move it, we still can put any of the remaining pieces to the cell below the top left corner.

How can we do this? In fact, we have just decreased the size of our field by deleting the top left corner. We can try to use the same approach: form a cycle that goes through 15 remaining cells, and move the pieces along this cycle.

Stop and Think! Can you draw such a cycle?

Trying to do this, you will see that one cell remains outside the cycle.² For example, Figure 6.9 shows such a cycle that avoids the top left corner (as we wanted), the left bottom corner (inconvenient but unavoidable), and goes through all other cells.

Having a cycle that covers all cells of the reduced field except one, we may use this cycle as before. This solves our problem assuming that (a) the cell we want to bring below the top left corner, is not in the bottom left corner and (b) the hole is not in the bottom left corner.

²There is a simple reason why this is unavoidable, the same as for the chessboard tiling: going along the cycle, we alternate between white and black cells (in a chessboard coloring). To close the cycle, we should return to the initial cell, and this cannot happen after 15 moves (or any other odd number of moves). Recall also Problem 150 where we proved that a piece on a chessboard cannot return to its original position after an odd number of moves.

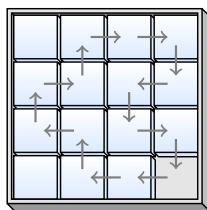


Figure 6.9: Using this cycle, we can move any piece (with one exception: left bottom corner) to the cell below the top left corner. Note the cycle must include a hole to make the cyclic movement possible. In our picture, the hole is shown in the right bottom corner, but any other place is possible (except the bottom left corner that is not in the cycle).

Stop and Think! Do you see why conditions (a) and (b) are important?

If (a) is not true, the cyclic move won't help. If (b) is not true, the cyclic move is impossible.

Problem 179 Consider Figure 6.10. How many cyclic moves (shown in the figure) are needed to bring the piece 9 in the cell below the top left corner?

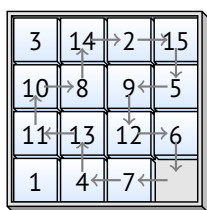


Figure 6.10: The top left corner is occupied by 3 forever; we want to bring 9 in the cell below the top left corner.

Now, we need to consider the exceptional cases when conditions (a) or (b) are not fulfilled.

Stop and Think! What should we do if (b) is false, i.e., the hole is in the bottom left corner?

We could move the hole right or up (by moving the neighbor pieces). Any of the two neighbor pieces would move the hole, but we should be careful. Recall that we wanted to move some piece below the top left corner, so we should not place this piece in the bottom left corner. Since we have a choice between two neighbors, this is always possible.

Stop and Think! What should we do if (a) is false, i.e., if the current position of the piece we want to bring below the top left corner, is the bottom left corner?

For example, this is the case for piece 1 in Figure 6.10.

Since we already know how to use cycles, we may just use another cycle that covers the bottom left corner (and the cell below the top left corner). One such cycle is shown in Figure 6.11.

6.4.4 Moving the Third Piece

Where are we now? A bit closer to our goal: we know how to fill *two* positions by the pieces we want: the top left corner and the cell below it. We can continue this process and fill the cell on the right of the top left corner, using the same approach.

For that, we need to cover the board without two cells (top left corner and the cell below it) by a cycle.

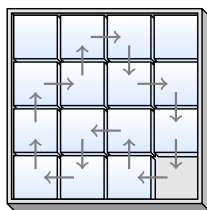


Figure 6.11: Using this cycle, we can move the piece from the bottom left corner (as well as others, except for the top right corner) to the place below the top left corner. Again, we assume that the cell outside the cycle (top right corner) is not a hole; if it is, we fill it by its neighbor.

Problem 180 Construct such a cycle.

One possible solution is shown in Figure 6.12.

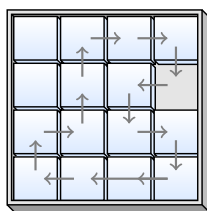


Figure 6.12: This cycle allows us to put any of the remaining pieces to the right of the top left piece, not touching the piece in the top left corner and the piece below it.

One could say that we are 20%-ready: to get an arbitrary permutation, we need to fill 15 places, and now we know how to fill three places (20% of 15).

Problem 181 Show how one can fill the cell in the second row and second column, using the same technique.

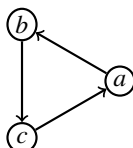
However, this 20% or even 26% ($4/15 = 0.2666\dots$) are too optimistic: the more pieces we bring to the desired positions (and do not want to move again), the less freedom we have for the cycle, and the more difficult the task is. For example, after we fill three places in the top row, we cannot use the cycle technique for the remaining (fourth) piece: there is no cycle, since this cell now has only one free neighbor. So we are stuck and some new idea is needed.

In the next section, we introduce new tools (3-cycles and conjugacy) that will allow us to move forward.

6.4.5 Decomposition into 3-cycles

For a while, we forget about 15-puzzle and consider permutations in general.

A *3-cycle* is a permutation where all the objects except three of them keep their places, and these three move in a cycle:



It is easy to see that a 3-cycle can be obtained by two transpositions and therefore is an even permutation.

Problem 182 Find two transpositions that (being performed sequentially) give the 3-cycle $a \rightarrow b \rightarrow c \rightarrow a$.

If we apply sequentially several 3-cycles, each of them can be replaced by two transpositions, therefore we get an even permutation. The following theorem shows that *every even permutation can be obtained this way*.

Theorem 6.4.2 Every even permutation of n objects can be represented as a sequence of 3-cycles.

Note that this theorem is vacuously true for $n = 1$ or $n = 2$: there are no even permutations except for the identity permutation (everybody stays in their place) that is a result of an empty sequence of 3-cycles.⁵

Proof sketch. Earlier, we proved that every permutation can be decomposed into transpositions — filling the places one after another by desired objects. We can do the same with 3-cycles instead of transpositions (2-cycles), if there are at least 3 objects left (we need a third spare object to form a cycle). Hence, compared to our previous argument for transpositions, we need one spare object, and we have to stop when only two positions remain unfilled. The good case is when the two remaining objects are already in the right places. The bad case is when they are not, and an additional transposition is needed to get the required permutation. But in this bad case the required permutation is obtained by several 3-cycles and one transposition, and therefore is odd, contrary to our assumption. ■

Problem 183 Implement a Python function

```
transform_by_3cycles(first, second)
```

that takes two lists that are permutations of $\{0, 1, \dots, n-1\}$ and returns a list of 3-cycles needed to get the second list from the first one. We assume that the permutation that transforms the first list into the second, is even. Each 3-cycle is represented by a triple of positions: (i, j, k) moves the element from position i (respectively, j, k) to position j (respectively k, i).

```
def transform_by_3_cycles(first, second):
    assert len(first) == len(second)
    n = len(first)
    assert set(first) == set(range(n))
    assert set(second) == set(range(n))

    cycles = []
    current = list(first)

    for i in range(n - 2):
        if current[i] != second[i]:
            idx = current.index(second[i])
            spare = i + 1 if idx != i + 1 else i + 2
            assert i != idx and i != spare and \
                idx != spare
            cycles.append((idx, i, spare))
            current[i], current[idx], current[spare] = \
                current[idx], current[spare], current[i]

        assert current[i] == second[i]

    return cycles
```

⁵3-cycles don't exist for $n = 1$ or $n = 2$, but this does not prevent us from considering an *empty* sequence of 3-cycles.

```
print(transform_by_3_cycles(
    [3, 4, 0, 2, 1, 5],
    [0, 5, 1, 4, 3, 2]
))
```

```
[(2, 0, 1), (5, 1, 2), (4, 2, 3), (5, 3, 4)]
```

In this code snippet, we check that `first` and `second` lists consist of $0, 1, \dots, n-1$, but do *not* check that the permutation required to transform the first list into the second one is even. The structure of the program is the same as before (for transpositions), but we need a spare element. We use $i + 1$ if possible (and if not, use $i + 2$; one of these two elements should be different from `idx`, and both are different from `i`). The `for` loop omits two last elements ($n-2$ and $n-1$), since there are no spare elements for them. If the permutation was even, they will go in the correct order automatically.

So far we developed a theory about even permutations and 3-cycles, but how does it help us to deal with 15-puzzle? Recall that we have to prove Theorem 6.4.1 for every even permutation of 15 pieces (here we do not add the dummy piece 0 and assume that the right bottom corner is empty). This permutation can be obtained as a sequence of 3-cycles. And if we can perform any 3-cycle according to the puzzle rules, we can combine the corresponding sequences of puzzle moves to achieve any even permutation. Therefore,

it is enough to prove Theorem 6.4.1 for a 3-cycle.

We do this in the next section.

6.4.6 How to Get an Arbitrary 3-cycle

Let us first recall what we have proven in Sections 6.4.2–6.4.4:

in the 15-puzzle, we can put three arbitrary pieces a, b, c in the upper left corner

as shown in Figure 6.13 (a).

From Section 6.4.1, we know how to move the empty cell ("hole") in any direction, so we can put it near them, as shown in Figure 6.13 (b).

Stop and Think! Why don't we destroy the a - b - c corner while moving the hole?

The problem may appear if we move the hole along a path that crosses the a - b - c corner. But this can be easily avoided (take a path avoiding the corner: the board without the corner is connected).⁴

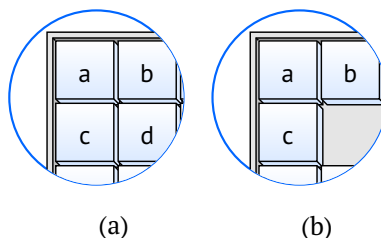


Figure 6.13: Three pieces a, b, c moved to the top left corner (a) with a hole near them (b).

How does this help us? Recall our cycle trick: now we can apply it to these four cells and organize a circular movement inside the 2×2 square (Figure 6.14).

⁴One can also use the cyclic move along a suitable cycle; see the discussion in the next section.

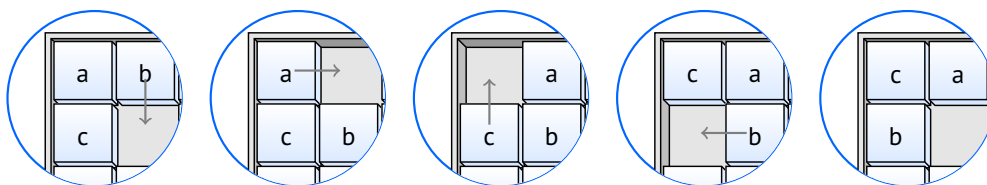


Figure 6.14: A 3-cycle performed inside a 2×2 square in the top left corner.

Recall that our goal was to implement a 3-cycle for any triple of positions; we have achieved this for a very special case of three positions near the top left corner. Again, what we achieved here is a very small part of what we wanted.

Still, we can combine our two observations to achieve the goal. Here is the key idea. To implement a 3-cycle that involves arbitrary three pieces a, b, c , one should:

Step 1. Bring three pieces a, b, c to the top left corner and put the hole near them (as we explained above).

Step 2. Perform that 3-cycle in the top left corner.

Step 3. Perform the reversed sequence of moves made during Step 1.

Mathematicians call this trick *conjugation*.

If we skipped Step 2, then Step 3 would restore the original configuration: video for Step 3 would be just a reversed video for Step 1. But after Step 2, c stands in place of a ; b stands in place of c , and c stands in place of b . Hence, performing all three steps, we obtain a 3-cycle that involves a, b , and c . Since this is possible for arbitrary 3 pieces, we can implement all 3-cycles, and therefore (Theorem 6.4.2) every even permutation.

6.5 Implementation

We have shown that a configuration in the 15-puzzle is solvable if and only if the corresponding permutation of 15 objects is even. This gives rise to a natural programming challenge: given a configuration (a list of positions of all pieces), find a sequence of allowed moves that transforms this position into the standard one.

We have to agree on the input and output formats. For the input, we represent the empty cell by 0, and all the pieces (including this 0-piece) are listed according to their labels in the “reading order”, so the standard configuration is represented as

$$[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 0],$$

whereas the impossible configuration we discussed is represented as

$$[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 15, 14, 0].$$

Hence, every configuration is now represented by a permutation of $0, 1, \dots, 15$, and we assume (as the statement of Theorem 6.4.1 does) that in the initial configuration the last number (bottom right cell) is 0.

For the output format, we have to decide how the game moves are encoded. The move is uniquely determined by the piece that is moved (since there is only one empty cell, the direction of the move is determined uniquely, if the move is possible at all). Then the sequence of moves is just a list of numbers that are written on the moved pieces, in the same order as the moves happen. For example, you may check that for the position


$$[1, 2, 3, 4, 5, 6, 7, 8, 13, 9, 11, 12, 10, 14, 15, 0]$$

one of the solutions is

$$[15, 14, 10, 13, 9, 10, 14, 15].$$

Problem 184 Implement a Python function

```
solution(configuration)
```

that gets a solvable configuration with the empty piece in the bottom right corner and outputs a sequence of moves that transforms it to a standard position. [Try it online!](#) 

An approach to this problem suggested by the discussion above is the following. Given an initial position,

- construct a sequence of 3-cycles needed to transform configuration into the standard position (recall Problem 183);
- decompose each 3-cycle (a, b, c) into a sequence of valid game moves as follows:
 - move a to the top left corner;
 - move c below a ;
 - move b to the right of a ;
 - move the empty cell (piece 0) below b ;
 - make four moves to mimic the 3-cycle that involves pieces a, b, c ;
 - repeat the moves from the first four steps in the reversed order.

(As the empty cell is in the bottom right corner initially, $a, b, c \neq 0$ during this process.)

We encourage you to implement this plan. It requires some patience and care, as we have found ourselves when trying to provide a reference solution. We reproduce this reference solution here in full. But (please!) try to solve the problem yourself before reading the solution. First, you may find a cleaner and nicer solution. Second, it will be much easier to understand (criticize, appreciate) our solution if you have your own experience.

First, we should be prepared to debug the program. For that it is convenient to print the configuration in a readable form.⁵ Recall that configuration is represented as a list of length 16 that includes all the pieces' numbers in the reading order. This list should be a permutation of `range(16)`; as we agreed, 0 stands for the empty cell. (The function prints an additional space character before one-digit piece numbers.)

```
def fancy_print(configuration):
    assert len(configuration) == 16
    print('-----')
    for i in range(len(configuration)):
        if configuration[i] < 10:
            print(' ', end='')
        print(configuration[i], end=' ')
        if i % 4 == 3:
            print()
```

We number the board cells in the reading order; 0 is the top left corner, 1 is the cell to the right of the top left corner, ..., 4 is the cell below the top left corner, ..., 15 is the bottom right corner. But we also need 2D coordinates of the cells in the form (row, column). Both row and column are integers in $0 \dots 3$; for example, the top left corner is (0,0), the cell on the right of it is (0,1), and the bottom right corner is (3,3) (Figure 6.15).

The following function converts 1D position to its 2D representation and returns a pair of integers:

```
def to_2d_index(index):
    assert 0 <= index < 16
    return index // 4, index % 4
```

⁵In our solution, this function is not used since it is needed only for debugging; we provide it for your convenience and to illustrate the input encoding.

	0	1	2	3
0	0	1	2	3
1	4	5	6	7
2	8	9	10	11
3	12	13	14	15

Figure 6.15: Indexing of pieces as well as row and columns of the board.

The puzzle moves are represented as transpositions (a piece is exchanged with an empty cell near it).⁶ Each transposition is encoded as a pair of 1D coordinates of cells that are exchanged. The following function checks that the cells numbers are valid and different, the cells are neighbors, and one of the cells is empty in the configuration. Then (the last line) it performs the exchange.

```
def apply_transposition(configuration, transposition):
    i, j = transposition
    assert i in range(16) and j in range(16)
    assert i != j
    i2d, j2d = to_2d_index(i), to_2d_index(j)
    assert abs(i2d[0] - j2d[0]) + \
           abs(i2d[1] - j2d[1]) == 1
    assert configuration[i] == 0 or \
           configuration[j] == 0
    configuration[i], configuration[j] = \
        configuration[j], configuration[i]
```

The sequences of moves are represented by lists of transpositions. Each lists contains pairs (transpositions) that should be performed sequentially:

```
def apply_transpositions(configuration, transpositions):
    for transposition in transpositions:
        apply_transposition(configuration, transposition)
```

The key role in the solution is played by sequence of moves performing a cyclic shift along some cycle. We represent a cycle as a list of 1D positions. The start position is arbitrary; for example, $[0, 1, 5, 4]$ and $[1, 5, 4, 0]$ represent the same cycle (while $[0, 4, 5, 1]$ represents another cycle, with the opposite orientation). A cyclic shift along the path $[a, b, \dots, y, z]$ is

$$a \leftarrow b \leftarrow \dots \leftarrow y \leftarrow z \leftarrow a$$

(piece moves from b -position to a -position, ..., from z to y , from a to z).

The following function applies the cyclic shift along the given path to the given configuration and returns the sequence of moves (transpositions) needed. First, we need to change the representation of the path in such a way that it starts with the empty cell. For that we search for the empty cell and then use slices and concatenation. We have to perform the transpositions sequentially: a (that is now a hole) is exchanged with b , then b is exchanged with c , etc. The list of transposition is prepared, then applied to configuration and then returned.

⁶This is not the required output format, so we will have to reencode the solution later.


```
def cyclic_shift(configuration, path):
    start = 0
    while configuration[path[start]] != 0:
        start = start+1
    rotated = path[start:]+path[:start]
    transpositions = []
    for i in range(len(rotated)-1):
        transpositions += [(rotated[i], rotated[i+1])]
    apply_transpositions(configuration, transpositions)
    return (transpositions)
```

The argument explained in the previous section used some carefully chosen cycles; in our program they are represented by the constants:

```
cycle1 = [0, 4, 8, 12, 13, 14, 15, 11, 7, 3, 2, 6, 10, 9, 5, 1]
cycle2a = [1, 5, 4, 8, 9, 13, 14, 15, 11, 10, 6, 7, 3, 2]
cycle2b = [1, 5, 4, 8, 12, 13, 9, 10, 14, 15, 11, 7, 6, 2]
cycle3 = [1, 5, 9, 8, 12, 13, 14, 15, 11, 10, 6, 7, 3, 2]
cycle4 = [2, 6, 5, 9, 8, 12, 13, 14, 15, 11, 7, 3]
```

Here, `cycle1` corresponds to the cycle shown in Figure 6.7. (Please do not trust us and check this carefully!) Arrows on the picture indicate the direction of the pieces' movements. In our program, the positions are listed in the reversed ordering (pieces are moved from right to left in the cycle: $0 \leftarrow 4 \leftarrow 8 \leftarrow \dots$). The next `cycle2a` corresponds to the cycle of Figure 6.9; the alternative cycle (Figure 6.10) is encoded as `cycle2b`. The constant `cycle3` corresponds to Figure 6.12 (used to bring the third piece to the position on the right of the top left corner). The last constant `cycle4` was not explained before; it is used to bring the empty cell to the position (1,1), see below.

The following function is the central part of our solution. It performs a 3-cycle that involves three pieces labeled a, b, c: the piece a is moved to the place where b was, etc. Here, the participants of the cycle are specified by their names (what is written on the piece), not their positions (where the piece is). The initial configuration is `cfg`; the sequence of transpositions (as pairs) is applied to `cfg` and returned.

```
def do_3_cycle(cfg, a, b, c):
    assert a in range(16) and b in range(16) and c in range(16)
    assert a != b and a != c and b != c
    assert a != 0 and b != 0 and c != 0
    transpositions = []
    # move a to top left corner (position 0)
    while cfg[0] != a:
        transpositions += cyclic_shift(cfg, cycle1)
    # move c below a
    if cfg[12] != c:
        # make sure the hole is not in the bottom left corner
        if cfg[12] == 0:
            transposition = (12, 8 if cfg[8] != c else 13)
            transpositions += [transposition]
            apply_transposition(cfg, transposition)
        while cfg[4] != c:
            transpositions += cyclic_shift(cfg, cycle2a)
    else:
        assert cfg[12] == c
        if cfg[3] == 0:
            transposition = (3, 7) # we know that cfg[7] != c
            transpositions += [transposition]
            apply_transposition(cfg, transposition)
```

```

    while cfg[4] != c:
        transpositions += cyclic_shift(cfg, cycle2b)
    # move b to the right of a
    while cfg[1] != b:
        transpositions += cyclic_shift(cfg, cycle3)
    # move the empty cell to the 2x2 block
    # ensure the hole is not at 10:
    if cfg[10] == 0:
        transposition = (10, 11)
        transpositions += [transposition]
        apply_transposition(cfg, transposition)
    while cfg[5] != 0: # use one more cycle to move the hole
        transpositions += cyclic_shift(cfg, cycle4)
    # now everything is ready for the conjugation
    assert cfg[0] == a and cfg[1] == b and cfg[4] == c and cfg[5] == 0
    abccycle_and_reverse = [(1, 5), (0, 1), (0, 4),
                           (4, 5)] + list(reversed(transpositions))
    apply_transpositions(cfg, abccycle_and_reverse)
    return transpositions + abccycle_and_reverse

```

This code consists of several parts. First we check that *a*, *b*, and *c* denote valid pieces (not the empty cell) and are different, and prepare the empty list *transpositions* where the moves will be placed.

Then, we compute a sequence of transpositions needed to place the piece labeled *a* to the top left corner. For that we perform the cyclic shift along *cycle1* until the label *cfg[0]* (the contents of the top left corner) becomes *a*. Next, we need to put *c* below *a*. This requires considering two cases.

To use *cycle2a*, we need to know that *c* is covered by this cycle, i.e., *c* is not in the avoided cell 12 (*cfg[12] != c*). Therefore, the *if* construction is used. But this is not all. The other exceptional case happens if the empty cell is outside the cycle (i.e., the bottom left corner is empty, *cfg[12] == 0*). In this case, we need to exchange the bottom left corner (position 12) with one of its neighbors (positions 8 or 13; we need to choose a neighbor that is different from *c*, to keep *c* in the cycle).

If *c* is in the bottom left corner, we use *cycle2b* instead of *cycle2a*. Again we need to ensure that the hole is in the cycle and not in the top right corner (i.e., that *cfg[3] != 0*). If *cfg[3] == 0*, we move the hole down to position 7. Here we are sure that *c* is not at position 7, being at position 12, so the conditional expression is not needed. After these preparations, we perform cyclic shifts along *cycle2b* until *c* moves in the desired position 4 (below the top left corner).

Then, we move *b* to the position 1 not touching *a* and *c*; here, we may use *cycle3* without hesitation.

The next step is to move the hole to the position 5 (to complete the 2×2 square as shown in Figure 6.13. In our argument above, we just noted that we can move the hole in any direction (and avoid three cells labeled *a*, *b*, *c*), but to make the program shorter (though using more moves) we use the cyclic shift along *cycle4* (Figure 6.16). The only precaution needed: if the hole is outside *cycle4*, at position 10, we move it elsewhere (say, to position 11 by exchanging 10 and 11).

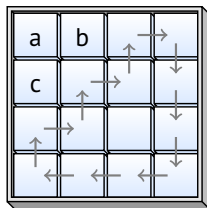


Figure 6.16: To move the hole at position (1,1), we use *cycle4*.

After that we have 2×2 square in the top left corner as shown in Figure 6.13 (b). It remains to perform the cyclic shift of a, b, c, as shown in Figure 6.14, and then play back the movements we have done at the first stage. The remaining transpositions are placed in `abccycle_and_reverse`, applied to `cfg` and then returned together with the `transpositions` list.

Now, we are able to perform any 3-cycle, and it remains to use them to put all pieces at the required places. Recall that we are able to do this for all pieces except the last two, and the correct ordering of the last two is guaranteed since we assume that our puzzle is solvable (the required permutation is even). First, we compute the required sequence of transpositions (and postpone the conversion into the output format):

```
def transpositions_solution(configuration):
    standard = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 0]
    transpositions = []
    current = list(configuration)
    for i in range(13): # two last pieces not processed
        if current[i] != standard[i]:
            idx = current.index(standard[i])
            assert idx > i
            # position idx should be moved to position i
            spare = i + 1 if i+1 != idx else i+2
            a, b, c = current[idx], current[i], current[spare]
            cycle_transpositions = do_3_cycle(current, a, b, c)
            transpositions += cycle_transpositions
        assert current[i] == standard[i]
    return transpositions
```

First, we copy the given configuration into `current`, and make the `transpositions` list (to be returned later) empty. Then, we fill correctly `current[0]`, ..., `current[12]`; the two last pieces `current[13]` and `current[14]` should be correctly filled (by 14 and 15 respectively), as explained above. To fill `current[i]` correctly (if it is not happened yet), we find the place `idx` where the desired piece happens to be now. It should be outside the part that is already filled or is to be filled (`idx > i`). Then, we find the spare position that is greater than `i` but is different from `idx`. For that we use `i+1` if it is different from `idx`, or the next position `i+2`. Finally, we find the labels a, b, c appearing in 3-cycle, compute the transpositions needed for this 3-cycle, apply them, and add them to the answer list.

The last step is to convert the answer (sequence of transpositions) to the required format, the list of labels for all moving pieces.

```
def solution(configuration):
    transpositions = transpositions_solution(configuration)
    answer = []
    current = list(configuration)
    for trans in transpositions:
        i, j = trans
        label = current[i] if current[i] != 0 else current[j]
        answer.append(label)
        apply_transposition(current, trans)
    return answer
```

We compute the required sequence of transpositions, copy the given configuration to `current`, and then convert every transposition `trans` to the required format. It exchanges positions `i` and `j`, we see which of them is not a hole, and append the corresponding label to the answer list.

Stop and Think! We did not check that the given permutation is even. What will our program return if it is odd (and the puzzle is not solvable)?

All the pieces except the two last ones will still be placed at their “standard” places, but the

remaining two pieces 14 and 15 will go in the wrong order (since the input permutation was odd). Thus, we will get the classical unsolvable position with 14 and 15 exchanged.

The approach described above leads to an efficient solution in practice, in the sense that it is reasonably fast. At the same time we have no guarantee that the number of moves is optimal (in most cases it uses a lot of moves). Finding the smallest number of moves is a more challenging problem. Google for “A* search 15-puzzle” to find out the details of one particular approach. In general, for boards of arbitrary size (instead of the 4×4 board), the problem of finding the optimal number of moves is [NP-complete](#) [↗](#) meaning that it is unlikely that it can be solved efficiently.