Permutations, Heap's algorithm and cryptarithms

Eric Martin, CSE, UNSW

COMP9021 Principles of Programming, session 2, 2018

```
In [1]: from itertools import permutations
```

To generate all permutations of a set S of cardinality 2 at least, it suffices to, for each member x of S, prepend or append x to each permutation of $S \setminus \{x\}$. This immediately translates into the following generator function:

```
In [2]: def naive_recursive_permute(S):
            if len(S) <= 1:
                yield list(S)
            else:
                for x in S:
                    for P in naive_recursive_permute(S - {x}):
                        yield [x] + P

        list(naive_recursive_permute({0}))
        list(naive_recursive_permute({0, 1}))
        list(naive_recursive_permute({0, 1, 2}))
        list(naive_recursive_permute({0, 1, 2, 3}))
```

```
Out[2]: [[0]]
```

```
Out[2]: [[0, 1], [1, 0]]
```

```
Out[2]: [[0, 1, 2], [0, 2, 1], [1, 0, 2], [1, 2, 0], [2, 0, 1], [2, 1, 0]]
```

```
Out[2]: [[0, 1, 2, 3],
         [0, 1, 3, 2],
         [0, 2, 1, 3],
         [0, 2, 3, 1],
         [0, 3, 1, 2],
         [0, 3, 2, 1],
         [1, 0, 2, 3],
         [1, 0, 3, 2],
         [1, 2, 0, 3],
         [1, 2, 3, 0],
         [1, 3, 0, 2],
         [1, 3, 2, 0],
         [2, 0, 1, 3],
```

```
[2, 0, 3, 1],
[2, 1, 0, 3],
[2, 1, 3, 0],
[2, 3, 0, 1],
[2, 3, 1, 0],
[3, 0, 1, 2],
[3, 0, 2, 1],
[3, 1, 0, 2],
[3, 1, 2, 0],
[3, 2, 0, 1],
[3, 2, 1, 0]]
```

Given $n > 0$ and a set S of cardinality n, generating all permutations of S with **naive_recursive_permute()** requires generates $n!$ many lists, each of which

- starts with a list L_1 of size 1,
- before a new list L_2 which is the concatenation of L_1 with a new singleton list is created,
- ...
- before eventually, a new list which is the concatenation of L_{n-1} with a new singleton list is created.

So altogether, $(2n - 1) \times n!$ many lists are created.

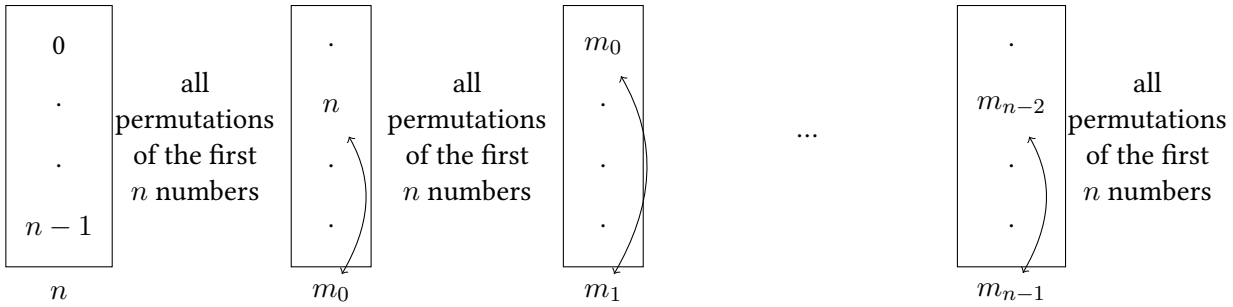
The minimal operation to transform one permutation P of S into another permutation of S is to exchange two of P's elements. For instance, all 6 permutations of $\{0, 1, 2\}$ can be generated, starting with $(0, 1, 2)$, thanks to five successive swaps:

- $(0, 1, 2)$
- $(0, 2, 1)$ (after swapping 1 and 2)
- $(2, 0, 1)$ (after swapping 0 and 2)
- $(2, 1, 0)$ (after swapping 0 and 1)
- $(1, 2, 0)$ (after swapping 2 and 1)
- $(1, 0, 2)$ (after swapping 2 and 0)

With this design, only one list is involved. A generator function can interrupt its execution with a **yield** statement at every stage of transformation of the list, that can then be copied or not into a new list, depending on the application (if our aim is only to display all permutations, then there is no need to make copies, but if our aim is to create a structure that embeds some or all permutations of S, then copies would be needed).

The key question is whether it is possible to, starting from a list of arbitrary length, exchange pairs of elements in such a way that all permutations of the list are eventually produced, without duplication; and in case that is possible, how to proceed.

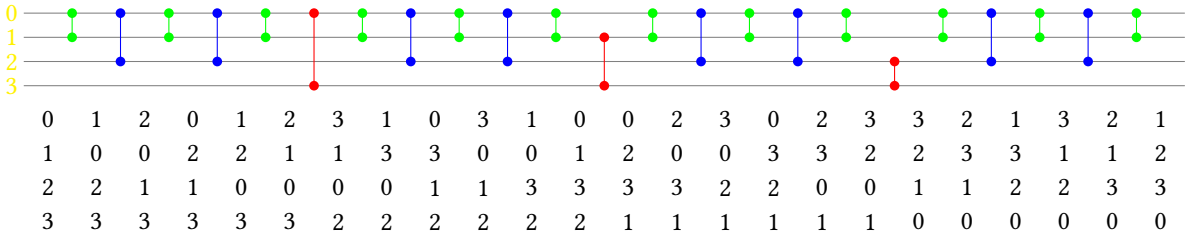
Let a nonzero natural number n be given. Heap's algorithm generates all permutations of a set S with $n + 1$ elements, in such a way that any permutation, the first one excepted, is obtained from the previous one by exchanging two of S's elements. Without loss of generality, take for S the set $\{0, 1, \ldots, n\}$. The recursive version of Heap's algorithm can be illustrated as follows.



So the algorithm generates all permutations of the form $L \star n$, then all permutations of the form $L \star m_0$, then all permutations of the form $L \star m_1$, ..., and eventually all permutations of the form $L \star m_{n-1}$. The scheme is correct if $\{m_0, m_1, \ldots, m_{n-1}\} = \{0, \ldots, n-1\}$: at every stage, the algorithm has to select a new number from the first n ones (and exchange it with the current $(n+1)$st number). Heap's algorithm uses the following strategy:

- In case n is odd, select the first number, then the second number, then the third number...
- In case n is even, always select the first number.

The following diagram illustrates with $n = 3$.



We prove that Heap's algorithm is correct and that moreover, the following holds for all $n \geq 1$:

1. starting with $(0, 1, 2, \ldots, 2n)$, all permutations of $(0, 1, 2, \ldots, 2n)$ are generated, ending in $(2n, 1, 2, \ldots, 2n-1, 0)$.
2. starting with $(0, 1, 2, \ldots, 2n+1)$, all permutations of $(0, 1, 2, \ldots, 2n+1)$ are generated, ending in $(2n-1, 2n, 1, 2, \ldots, 2n-2, 2n+1, 0)$.

For instance:

- For the case of lists with an even number of elements:
  - starting with $(0, 1, 2, 3)$, Heap's algorithm produces $(1, 2, 3, 0)$ as last permutation;
  - starting with $(0, 1, 2, 3, 4, 5)$, it produces $(3, 4, 1, 2, 5, 0)$ as last permutation;
  - starting with $(0, 1, 2, 3, 4, 5, 6, 7)$, it produces $(5, 6, 1, 2, 3, 4, 7, 0)$ as last permutation.
- For the case of lists with an odd number of elements:
  - starting with $(0, 1, 2)$, Heap's algorithm produces $(2, 1, 0)$ as last permutation;
  - starting with $(0, 1, 2, 3, 4)$, it produces $(4, 1, 2, 3, 0)$ as last permutation;
  - starting with $(0, 1, 2, 3, 4, 5, 6)$, it produces $(6, 1, 2, 3, 4, 5, 0)$ as last permutation.

The previous formulas can be used to generalise Heap's algorithm and generate all sequences of k numbers chosen from $\{0, 1, \ldots, n\}$: when the last number has been selected, and the penultimate number has been selected, ..., and the $(n-k+1)$st number has been selected, it suffices to stop the recursion and "simulate" all permutations of the remaining $n+1-k$ numbers by applying those formulas.

Proof of 1. and 2. is by induction. The base case $n = 1$ is straightforward, so let $n \geq 1$ be given, and assume that 1. holds. We show that 2. holds too.

- Starting from $(0, 1, 2, \ldots, 2n-1, 2n) \star 2n+1$, Heap's algorithm generates all permutations of the form $L \star 2n+1$, ending with $(2n, 1, 2, \ldots, 2n-1, 0) \star 2n+1$.
- Permuting first and last elements, $(2n, 1, 2, \ldots, 2n-1, 0) \star 2n+1$ is changed to $(2n+1, 1, 2, \ldots, 2n-1, 0) \star 2n$. Starting from $(2n+1, 1, 2, \ldots, 2n-1, 0) \star 2n$, the algorithm then generates all permutations of the form $L \star 2n$, ending with $(0, 1, 2, \ldots, 2n-1, 2n+1) \star 2n$.
- Permuting second and last elements, $(0, 1, 2, \ldots, 2n-1, 2n+1) \star 2n$ is changed to $(0, 2n, 2, \ldots, 2n-1, 2n+1) \star 1$. Starting from $(0, 2n, 2, \ldots, 2n-1, 2n+1) \star 1$, the algorithm then generates all permutations of the form $L \star 1$, ending with $(2n+1, 2n, 2, \ldots, 2n-1, 0) \star 1$.
- Permuting third and last elements, $(2n+1, 2n, 2, \ldots, 2n-1, 0) \star 1$ is changed to $(2n+1, 2n, 1, 3 \ldots, 2n-1, 0) \star 2$. Starting from $(2n+1, 2n, 1, 3 \ldots, 2n-1, 0) \star 2$, the algorithm then generates all permutations of the form $L \star 2$, ending with $(0, 2n, 1, 3 \ldots, 2n-1, 2n+1) \star 2$.
- Permuting fourth and last elements, $(0, 2n, 1, 3 \ldots, 2n-1, 2n+1) \star 2$ is changed to $(2n+1, 2n, 1, 2 \ldots, 2n-1, 0) \star 3$. Starting from $(2n+1, 2n, 1, 2 \ldots, 2n-1, 0) \star 3$, the algorithm then generates all permutations of the form $L \star 3$... till all permutations of the form $L \star 2n-1$, ending in $(2n+1, 2n, 1, 2, \ldots, 2n-2, 0) \star 2n-1$.
- Permuting last two elements, $(2n+1, 2n, 1, 2, \ldots, 2n-2, 0) \star 2n-1$ is changed to $(2n+1, 2n, 1, 2, \ldots, 2n-2, 2n-1) \star 0$. Starting from $(2n+1, 2n, 1, 2, \ldots, 2n-2, 2n-1) \star 0$, the algorithm then generates all permutations of the form $L \star 0$, ending with $(2n-1, 2n, 1, 2, \ldots, 2n-2, 2n+1) \star 0$.

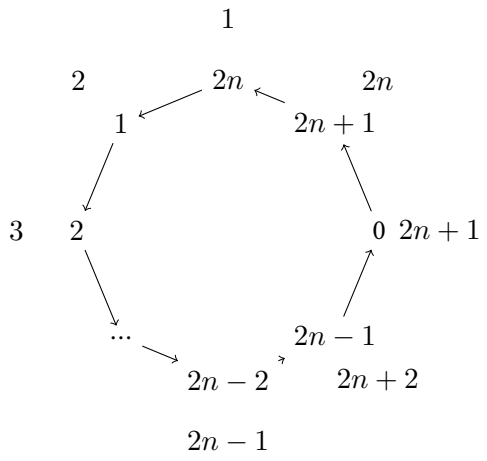
So we have established that 2. holds.

Now assume that 2. holds. We show that 1. with n replaced by $n+1$ holds too. The inner circle of the following diagram shows how elements move from one position to another one after all permutations of a list consisting of the $2n+2$ numbers 0, ..., $2n+1$ have been performed by Heap's algorithm. For instance, the first element ends up as the last element, moving from position (index) 0 and eventually ending up at position (index) $2n+1$. After all permutations of $(0, 1, 2, \ldots, 2n, 2n+1)$ have been generated, ending in $(2n-1, 2n, 1, 2, \ldots, 2n-2, 2n+1, 0)$, so after all permutations of $(0, 1, 2, \ldots, 2n, 2n+1) \star 2n+2$ have been generated, ending in $(2n-1, 2n, 1, 2, \ldots, 2n-2, 2n+1, 0) \star 2n+2$, Heap's algorithm replaces the element now at position 0, that is, $2n-1$ (originally at position $2n-1$), with $2n+2$. This is depicted in the following diagram with $2n+2$ on the outer circle facing $2n-1$ on the inner circle. At the end of each of the following stages, the algorithm permutes the element currently at position 0 with the element currently at position $2n+2$, that is, the element at position 0 at the end of previous stage. Hence as illustrated in the diagram, move to position $2n+2$: first $2n-1$ replaced by $2n+2$, then $2n-2$ replaced by $2n-1$, ..., then 2 replaced by 3, then 1 replaced by 2, then $2n$ replaced by 1, then $2n+1$ replaced by $2n$, and eventually 0 replaced by $2n+1$. Finally, all permutations of the numbers then at position 0, ..., $2n+1$ (those numbers being 1, 2, ..., $2n+2$) are generated, corresponding to a last, $(2n+2)$nd rotation in the following diagram, hence rotation following a "full circle". This means that:

- ends up at position 0 the element which at the beginning of this last round of permutations, is a position $2n-1$, that is, $2n+2$,
- ends up at position 1 the element which at the beginning of this last round of permutations, is a position $2n$, that is, 1,

- ...

resulting in the final list $(2n+2, 1, 2, 3, \ldots, 2n-1, 2n, 2n+1) \star 0$. So we have established that 1. with n replaced by $n+1$ holds.



Heap's algorithm naturally translates into the following recursive implementation:

```
In [3]: def heap_permute_by_recursion(L):
            yield from recursive_heap_permute(L, len(L))

        def recursive_heap_permute(L, length):
            if length <= 1:
                yield L
            else:
                length -= 1
                for i in range(length):
                    yield from recursive_heap_permute(L, length)
                    if length % 2:
                        L[i], L[length] = L[length], L[i]
                    else:
                        L[0], L[length] = L[length], L[0]
                yield from recursive_heap_permute(L, length)

        list(list(L) for L in heap_permute_by_recursion([0]))
        list(list(L) for L in heap_permute_by_recursion([0, 1]))
        list(list(L) for L in heap_permute_by_recursion([0, 1, 2]))
        list(list(L) for L in heap_permute_by_recursion([0, 1, 2, 3]))
```

```
Out[3]: [[0]]
```

```
Out[3]: [[0, 1], [1, 0]]
```

```
Out[3]: [[0, 1, 2], [1, 0, 2], [2, 0, 1], [0, 2, 1], [1, 2, 0], [2, 1, 0]]
```

```
Out[3]: [[0, 1, 2, 3],
         [1, 0, 2, 3],
         [2, 0, 1, 3],
         [0, 2, 1, 3],
         [1, 2, 0, 3],
         [2, 1, 0, 3],
         [3, 1, 0, 2],
         [1, 3, 0, 2],
         [0, 3, 1, 2],
         [3, 0, 1, 2],
         [1, 0, 3, 2],
         [0, 1, 3, 2],
         [0, 2, 3, 1],
         [2, 0, 3, 1],
         [3, 0, 2, 1],
         [0, 3, 2, 1],
         [2, 3, 0, 1],
         [3, 2, 0, 1],
         [3, 2, 1, 0],
         [2, 3, 1, 0],
         [1, 3, 2, 0],
         [3, 1, 2, 0],
         [2, 1, 3, 0],
         [1, 2, 3, 0]]
```

The following function traces the execution of the recursive implementation of Heap's algorithm:

```
In [4]: def trace_recursive_heap_permute(L, length, depth):
            if length <= 1:
                print('    ' * depth, f'Yield {L}')
                yield L
            else:
                length -= 1
                for i in range(length):
                    print('    ' * depth, f'Yield permutations of {L[: length]} '
                                           f'extended with {L[length: ]}'
                          )
                    yield from trace_recursive_heap_permute(L, length, depth + 1)
                    if length % 2:
                        L[i], L[length] = L[length], L[i]
                        print('    ' * depth, f'Exchanging L[{i}] and L[{length}]',
                                               '->', L
                              )
                    else:
                        L[0], L[length] = L[length], L[0]
                        print('    ' * depth, f'Exchanging L[{0}] and L[{length}]',
                                               '->', L
                              )
```

```
                print('    ' * depth, f'Yield permutations of {L[: length]} '
                                      f'extended with {L[length: ]}'
                      )
                yield from trace_recursive_heap_permute(L, length, depth + 1)

        for _ in trace_recursive_heap_permute([0, 1, 2, 3], 4, 0): pass;
```

```
Yield permutations of [0, 1, 2] extended with [3]
    Yield permutations of [0, 1] extended with [2, 3]
        Yield permutations of [0] extended with [1, 2, 3]
            Yield [0, 1, 2, 3]
        Exchanging L[0] and L[1] -> [1, 0, 2, 3]
        Yield permutations of [1] extended with [0, 2, 3]
            Yield [1, 0, 2, 3]
    Exchanging L[0] and L[2] -> [2, 0, 1, 3]
    Yield permutations of [2, 0] extended with [1, 3]
        Yield permutations of [2] extended with [0, 1, 3]
            Yield [2, 0, 1, 3]
        Exchanging L[0] and L[1] -> [0, 2, 1, 3]
        Yield permutations of [0] extended with [2, 1, 3]
            Yield [0, 2, 1, 3]
    Exchanging L[0] and L[2] -> [1, 2, 0, 3]
    Yield permutations of [1, 2] extended with [0, 3]
        Yield permutations of [1] extended with [2, 0, 3]
            Yield [1, 2, 0, 3]
        Exchanging L[0] and L[1] -> [2, 1, 0, 3]
        Yield permutations of [2] extended with [1, 0, 3]
            Yield [2, 1, 0, 3]
Exchanging L[0] and L[3] -> [3, 1, 0, 2]
Yield permutations of [3, 1, 0] extended with [2]
    Yield permutations of [3, 1] extended with [0, 2]
        Yield permutations of [3] extended with [1, 0, 2]
            Yield [3, 1, 0, 2]
        Exchanging L[0] and L[1] -> [1, 3, 0, 2]
        Yield permutations of [1] extended with [3, 0, 2]
            Yield [1, 3, 0, 2]
    Exchanging L[0] and L[2] -> [0, 3, 1, 2]
    Yield permutations of [0, 3] extended with [1, 2]
        Yield permutations of [0] extended with [3, 1, 2]
            Yield [0, 3, 1, 2]
        Exchanging L[0] and L[1] -> [3, 0, 1, 2]
        Yield permutations of [3] extended with [0, 1, 2]
            Yield [3, 0, 1, 2]
    Exchanging L[0] and L[2] -> [1, 0, 3, 2]
    Yield permutations of [1, 0] extended with [3, 2]
        Yield permutations of [1] extended with [0, 3, 2]
            Yield [1, 0, 3, 2]
        Exchanging L[0] and L[1] -> [0, 1, 3, 2]
```

```
        Yield permutations of [0] extended with [1, 3, 2]
            Yield [0, 1, 3, 2]
Exchanging L[1] and L[3] -> [0, 2, 3, 1]
Yield permutations of [0, 2, 3] extended with [1]
    Yield permutations of [0, 2] extended with [3, 1]
        Yield permutations of [0] extended with [2, 3, 1]
            Yield [0, 2, 3, 1]
        Exchanging L[0] and L[1] -> [2, 0, 3, 1]
        Yield permutations of [2] extended with [0, 3, 1]
            Yield [2, 0, 3, 1]
    Exchanging L[0] and L[2] -> [3, 0, 2, 1]
    Yield permutations of [3, 0] extended with [2, 1]
        Yield permutations of [3] extended with [0, 2, 1]
            Yield [3, 0, 2, 1]
        Exchanging L[0] and L[1] -> [0, 3, 2, 1]
        Yield permutations of [0] extended with [3, 2, 1]
            Yield [0, 3, 2, 1]
    Exchanging L[0] and L[2] -> [2, 3, 0, 1]
    Yield permutations of [2, 3] extended with [0, 1]
        Yield permutations of [2] extended with [3, 0, 1]
            Yield [2, 3, 0, 1]
        Exchanging L[0] and L[1] -> [3, 2, 0, 1]
        Yield permutations of [3] extended with [2, 0, 1]
            Yield [3, 2, 0, 1]
Exchanging L[2] and L[3] -> [3, 2, 1, 0]
Yield permutations of [3, 2, 1] extended with [0]
    Yield permutations of [3, 2] extended with [1, 0]
        Yield permutations of [3] extended with [2, 1, 0]
            Yield [3, 2, 1, 0]
        Exchanging L[0] and L[1] -> [2, 3, 1, 0]
        Yield permutations of [2] extended with [3, 1, 0]
            Yield [2, 3, 1, 0]
    Exchanging L[0] and L[2] -> [1, 3, 2, 0]
    Yield permutations of [1, 3] extended with [2, 0]
        Yield permutations of [1] extended with [3, 2, 0]
            Yield [1, 3, 2, 0]
        Exchanging L[0] and L[1] -> [3, 1, 2, 0]
        Yield permutations of [3] extended with [1, 2, 0]
            Yield [3, 1, 2, 0]
    Exchanging L[0] and L[2] -> [2, 1, 3, 0]
    Yield permutations of [2, 1] extended with [3, 0]
        Yield permutations of [2] extended with [1, 3, 0]
            Yield [2, 1, 3, 0]
        Exchanging L[0] and L[1] -> [1, 2, 3, 0]
        Yield permutations of [1] extended with [2, 3, 0]
            Yield [1, 2, 3, 0]
```

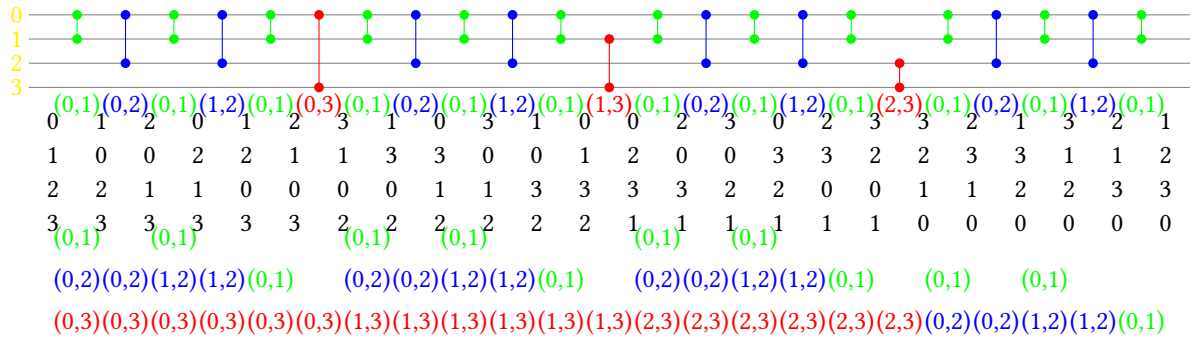
We see that **recursive_heap_permute()** yields its first argument **L**, and then essentially, again and again, exchanges two elements of **L** and yields **L** in its new state, through a chain of recursive calls of various lengths. Converting the recursive implementation of Heap's algorithm into an iterative implementation requires to know, after every exchange except the last one, which exchange to perform next. It is not enough to know the last exchange, if some took place already. Indeed, a given exchange often happens many times in the overall sequence of exchanges, and all the more often that it is associated with a smaller second argument **length**; one has to be able to identify where in the overall sequence a given exchange happens. Taking as before the case $n = 3$ as an example, and referring to the corresponding illustration and the chosen colours, the principle can be described in reference to the illustration below:

- There are 4 groups with as separators, the red exchanges. One can know in which of these 4 groups we are by noting $(0, 3)$ for the first group, $(1, 3)$ for the second group, $(2, 3)$ for the third group, and nothing for the fourth group.
- Within such a group, there are 3 groups with as separators, the blue exchanges. One can know in which of these 3 groups we are by noting $(0, 2)$ for the first group, $(1, 2)$ for the second group, and nothing for the third group.
- Within such a group, there are 2 groups with as separator, the green exchange. One can know in which of these 2 groups we are by noting $(0, 1)$ for the first group, and nothing for the second group.

Stacking up those existing pairs, listed from bottom to top in the order previously described, one has then all the information one needs:

- The next exchange to perform can be identified from the pair at the top, which can then then be popped from the top of the stack as the exchange is executed.
- One then moves one position in the overall sequence of exchanges, which requires pushing to the top of the stack at least one pair of numbers, in accordance with the principle just described; if a pair of the form (i, m) has been popped off the stack, then what to push to the top of the stack is:
  - $(i + 1, m)$ if $i < m - 1$, followed by
  - $(0, m - 1), (0, m - 2), \ldots (0, 1)$

The picture below represents, at the bottom, the various states of the stack, before the first exchange, between successive exchanges and after the last exchange, and below each exchange, the pair of numbers popped from the top of the stack.



These observations result in the followed iterative implementation of Heap's algorithm, using a list for the stack, the element at the end of the list corresponding to the element at the top of the stack, and using the list's **pop()** and **append()** methods for popping the element at the top of the stack and pushing an element to the top of the stack, respectively:

```
In [5]: def iterative_heap_permute(L):
            yield L
            stack = [(0, i) for i in range(len(L) - 1, 0, -1)]
            while stack:
                low, high = stack.pop()
                if high % 2:
                    L[low], L[high] = L[high], L[low]
                else:
                    L[0], L[high] = L[high], L[0]
                yield L
                if low + 1 != high:
                    stack.append((low + 1, high))
                for i in range(high - 1, 0, -1):
                    stack.append((0, i))

        list(list(L) for L in iterative_heap_permute([0]))
        list(list(L) for L in iterative_heap_permute([0, 1]))
        list(list(L) for L in iterative_heap_permute([0, 1, 2]))
        list(list(L) for L in iterative_heap_permute([0, 1, 2, 3]))
```

```
Out[5]: [[0]]
```

```
Out[5]: [[0, 1], [1, 0]]
```

```
Out[5]: [[0, 1, 2], [1, 0, 2], [2, 0, 1], [0, 2, 1], [1, 2, 0], [2, 1, 0]]
```

```
Out[5]: [[0, 1, 2, 3],
         [1, 0, 2, 3],
         [2, 0, 1, 3],
         [0, 2, 1, 3],
         [1, 2, 0, 3],
         [2, 1, 0, 3],
         [3, 1, 0, 2],
         [1, 3, 0, 2],
         [0, 3, 1, 2],
         [3, 0, 1, 2],
         [1, 0, 3, 2],
         [0, 1, 3, 2],
         [0, 2, 3, 1],
         [2, 0, 3, 1],
         [3, 0, 2, 1],
         [0, 3, 2, 1],
         [2, 3, 0, 1],
         [3, 2, 0, 1],
         [3, 2, 1, 0],
         [2, 3, 1, 0],
         [1, 3, 2, 0],
         [3, 1, 2, 0],
```

```
        [2, 1, 3, 0],
        [1, 2, 3, 0]]
```

The following function traces the execution of the iterative implementation of Heap's algorithm:

```
In [6]: def trace_iterative_heap_permute(L):
            print('Yield', L)
            yield L
            stack = [(0, i) for i in range(len(L) - 1, 0, -1)]
            print(f'stack is {stack}', end = '')
            while stack:
                low, high = stack.pop()
                print(f' ... popping ({low}, {high})')
                if high % 2:
                    L[low], L[high] = L[high], L[low]
                    print(f'Exchanging L[{low}] and L[{high}]', '->', L)
                else:
                    L[0], L[high] = L[high], L[0]
                    print(f'Exchanging L[0] and L[{high}]', '->', L)
                print('Yield', L)
                yield L
                if low + 1 != high:
                    stack.append((low + 1, high))
                for i in range(high - 1, 0, -1):
                    stack.append((0, i))
                print(f'Stack is {stack}', end = '')

        for _ in trace_iterative_heap_permute([0, 1, 2, 3]): pass;
```

```
Yield [0, 1, 2, 3]
stack is [(0, 3), (0, 2), (0, 1)] … popping (0, 1)
Exchanging L[0] and L[1] -> [1, 0, 2, 3]
Yield [1, 0, 2, 3]
Stack is [(0, 3), (0, 2)] … popping (0, 2)
Exchanging L[0] and L[2] -> [2, 0, 1, 3]
Yield [2, 0, 1, 3]
Stack is [(0, 3), (1, 2), (0, 1)] … popping (0, 1)
Exchanging L[0] and L[1] -> [0, 2, 1, 3]
Yield [0, 2, 1, 3]
Stack is [(0, 3), (1, 2)] … popping (1, 2)
Exchanging L[0] and L[2] -> [1, 2, 0, 3]
Yield [1, 2, 0, 3]
Stack is [(0, 3), (0, 1)] … popping (0, 1)
Exchanging L[0] and L[1] -> [2, 1, 0, 3]
Yield [2, 1, 0, 3]
Stack is [(0, 3)] … popping (0, 3)
Exchanging L[0] and L[3] -> [3, 1, 0, 2]
Yield [3, 1, 0, 2]
```

```
Stack is [(1, 3), (0, 2), (0, 1)] … popping (0, 1)
Exchanging L[0] and L[1] -> [1, 3, 0, 2]
Yield [1, 3, 0, 2]
Stack is [(1, 3), (0, 2)] … popping (0, 2)
Exchanging L[0] and L[2] -> [0, 3, 1, 2]
Yield [0, 3, 1, 2]
Stack is [(1, 3), (1, 2), (0, 1)] … popping (0, 1)
Exchanging L[0] and L[1] -> [3, 0, 1, 2]
Yield [3, 0, 1, 2]
Stack is [(1, 3), (1, 2)] … popping (1, 2)
Exchanging L[0] and L[2] -> [1, 0, 3, 2]
Yield [1, 0, 3, 2]
Stack is [(1, 3), (0, 1)] … popping (0, 1)
Exchanging L[0] and L[1] -> [0, 1, 3, 2]
Yield [0, 1, 3, 2]
Stack is [(1, 3)] … popping (1, 3)
Exchanging L[1] and L[3] -> [0, 2, 3, 1]
Yield [0, 2, 3, 1]
Stack is [(2, 3), (0, 2), (0, 1)] … popping (0, 1)
Exchanging L[0] and L[1] -> [2, 0, 3, 1]
Yield [2, 0, 3, 1]
Stack is [(2, 3), (0, 2)] … popping (0, 2)
Exchanging L[0] and L[2] -> [3, 0, 2, 1]
Yield [3, 0, 2, 1]
Stack is [(2, 3), (1, 2), (0, 1)] … popping (0, 1)
Exchanging L[0] and L[1] -> [0, 3, 2, 1]
Yield [0, 3, 2, 1]
Stack is [(2, 3), (1, 2)] … popping (1, 2)
Exchanging L[0] and L[2] -> [2, 3, 0, 1]
Yield [2, 3, 0, 1]
Stack is [(2, 3), (0, 1)] … popping (0, 1)
Exchanging L[0] and L[1] -> [3, 2, 0, 1]
Yield [3, 2, 0, 1]
Stack is [(2, 3)] … popping (2, 3)
Exchanging L[2] and L[3] -> [3, 2, 1, 0]
Yield [3, 2, 1, 0]
Stack is [(0, 2), (0, 1)] … popping (0, 1)
Exchanging L[0] and L[1] -> [2, 3, 1, 0]
Yield [2, 3, 1, 0]
Stack is [(0, 2)] … popping (0, 2)
Exchanging L[0] and L[2] -> [1, 3, 2, 0]
Yield [1, 3, 2, 0]
Stack is [(1, 2), (0, 1)] … popping (0, 1)
Exchanging L[0] and L[1] -> [3, 1, 2, 0]
Yield [3, 1, 2, 0]
Stack is [(1, 2)] … popping (1, 2)
Exchanging L[0] and L[2] -> [2, 1, 3, 0]
Yield [2, 1, 3, 0]
```

```
Stack is [(0, 1)] … popping (0, 1)
Exchanging L[0] and L[1] -> [1, 2, 3, 0]
Yield [1, 2, 3, 0]
Stack is []
```

A cryptarithm is a puzzle where words, written in all uppercase, are combined with some of the binary operators +, *, -, / and ** and one occurrence of the binary relational operator ==, some subexpressions being possibly parenthesised, so that the resulting expression is syntactically correct. Each letter should then be assigned a single digit, different to 0 in case the letter occurs at the beginning of a word, in such a way that replacing each occurrence of a letter by the digit assigned to it, the resulting arithmetic expression is valid. An example of cryptarithm is **DO + YOU + FEEL == LUCKY**: assigning

- **0** to **U**
- **1** to **L**
- **3** to **C**
- **4** to **E**
- **5** to **D**
- **6** to **K**
- **7** to **O**
- **8** to **Y**
- **9** to **F**

results in the arithmetic equation **57 + 870 + 9441 == 10368**, which indeed holds (it is actually the only solution to the cryptarithm).

We represent a cryptarithm as a string where spaces can occur anywhere, of course except within words:

```
In [7]: cryptarithm = 'DO + YOU + FEEL == LUCKY'
```

First, we determine the set of letters that occur at the start of a word in the cryptarithm, and the set of letters that occur in the cryptarithm, but never at the beginning of a word. We initialise both sets to the empty set:

```
In [8]: letters_starting_a_word = set()
        letters_not_starting_a_word = set()
```

We process each character **c** in **cryptarithm**. When **c** starts a word, the variable **in_word**, initialised to **False**, has the value **False**; we change it to **True** and add **c** to **letters_starting_a_word**. When **c** is within a word but is not the word's first letter, **in_word** has the value **True** and we add **c** to **letters_not_starting_a_word**. When **c** is not within a word, then we set the value of **in_word** to **False**, which is necessary only if **c** directly follows a word (otherwise, **in_word** is currently set to **False**, which is harmlessly overwritten with **False**):

```
In [9]: in_word = False
        for c in cryptarithm:
            if str.isalpha(c):
                if not in_word:
                    letters_starting_a_word.add(c)
                    in_word = True
```

```
                else:
                    letters_not_starting_a_word.add(c)
            else:
                in_word = False

        letters_starting_a_word
        letters_not_starting_a_word
```

```
Out[9]: {'D', 'F', 'L', 'Y'}
```

```
Out[9]: {'C', 'E', 'K', 'L', 'O', 'U', 'Y'}
```

Some of the letters might have been found out to occur both at the start of some words, and within but not at the start of some words. We want **letters_starting_a_word** and **letters_not_starting_a_word** to eventually be disjoint, so remove from the latter the letters that occur in the former:

```
In [10]: letters_not_starting_a_word -= letters_starting_a_word
         letters_not_starting_a_word
```

```
Out[10]: {'C', 'E', 'K', 'O', 'U'}
```

We then create a list consisting of all letters in **letters_not_starting_a_word** and **letters_starting_a_word**, the latter following the former:

```
In [11]: all_letters = list(letters_not_starting_a_word) +\
                       list(letters_starting_a_word)
         all_letters
```

```
Out[11]: ['O', 'C', 'K', 'U', 'E', 'L', 'F', 'Y', 'D']
```

A potential solution will be created by:

- assigning a distinct nonzero digit to each letter in **['D', 'L', 'F', 'Y']** (so 4 digits altogether), and
- assigning a distinct digit, not previously used, and possibly equal to 0, to each letter in **['C', 'U', 'K', 'E', 'O']** (so 5 digits altogether).

We proceed as follows. Let **digits** be **['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']** (we will see later the technical reason why we take for **digits** a list of digit characters rather than a list of digits; will still refer to the members of **digits** as digits).

- To assign candidate digits to **['D', 'L', 'F', 'Y']**, we "ignore" 0, so operate on **digits[1:]**, being interested in **digits[-4:]**, to go through all of the $9 \times 8 \times 7 \times 6$ possible ways of assigning 4 distinct nonzero digits to the last 4 positions in **digits**. To this aim, we essentially use **recursive_heap_permute()** to determine **digits[-1]**, ..., **digits[-4]**, but we then do not want to go through the $5!$ possible ways of assigning the remaining 5 nonzero digits to **digits[5]**, ..., **digits[1]**: after **digits[6:]** has been fixed, we only want to "jump over" the last 5 recursive calls to **recursive_heap_permute()** that operate on **digits[1: 6]**, exploiting the theoretical fact that we established above, namely, that by starting with **['1', '2', '3', '4', '5']**, Heap's algorithm yields **['5', '2', '3', '4', '1']** as last permutation (since by starting with $(0, 1, 2, 3, 4)$, it yields $(4, 1, 2, 3, 0)$ as last permutation).

- To assign candidate digits to **['C', 'U', 'K', 'E', 'O']**, we operate on **digits[: 6]**, being interested in **digits[1: 6]**, to go through all of the 6! possible ways to assign 5 distinct digits to **digits[1]**, ... **digits[5]**, amongst those not occupying **digits[6:]**. Because only one, unassigned but known digit will remain (in **digits[0]**), there is no need to "jump over" recursive calls; if **all_letters** was less than 9 characters long, we would then exploit again our knowledge of what is the last permutation that Heap's algorithm produces when operating on a list of a given size.

The following function, **permute()**, is designed to permute the last **length** members of **digits[: start + size]**, using **size** many indexes in **digits**, the first of which is **start**. It will be called to assign candidate digits first to **letters_starting_a_word**, and then to **letters_not_starting_a_word**. It assumes that **length** is at most equal to **size**. More precisely, to assign distinct nonzero digits to the letters in **['D', 'L', 'F', 'Y']**, it will first be called as **permute(digits, 1, 9, len(letters_starting_a_word))**, that is, **permute(digits, 1, 9, 4)**, and to assign other distinct digits (one of which is possibly 0) to the letters in **['C', 'U', 'K', 'E', 'O']**, it will then be called as **permute(digits, 0, 9 - len(letters_starting_a_word), len(letters_not_starting_a_word))**, that is, **permute(digits, 0, 5, 5)**.

At the top level, **permute(digits, start, size, length)** determines the **length**'th digit of **digits[start: start + size]**; recursive calls to **permute(digits, start, size - 1, length - 1)** determine the **(length - 1)**'st digit of **digits[start: start + size - 1]**... until recursive calls to **permute(digits, start, size - length, 0)** yield **None**, with the last **length** many digits of **digits[: start + size]** set to a particular assignment of digits to letters.

We start implementing **permute()**, dealing with the case where **length** is equal to 0. Then the generator function pauses with a **yield** statement, so that **digits** can be used in its current state. For now we comment out the **yield** statement, as we first focus on implementing correctly the "jump over" the last recursive calls that we previously discussed, and checking that that part of the code works in accordance with the theoretical results that we established above, namely:

- the last permutation of $(0, 1, 2, \ldots, 2n)$ ends in $(2n, 1, 2, \ldots, 2n-1, 0)$.
- the last permutation of $(0, 1, 2, \ldots, 2n+1)$ ends in $(2n-1, 2n, 1, 2, \ldots, 2n-2, 2n+1, 0)$.

We need to distinguish between the cases where **size** is even or odd, but **size** equal to 2 or 4 are best treated as special cases. Also, in case **size** is equal to 1 when **length** is equal to 0, the digit at position **start** is the unique unassigned digit in the section of **digits** that **permute()** has been operating on; a permutation of 1 element does not need to be "jumped over" (if **size** was equal to 1, then the body of the **if** statement below would just exchange **L[start]** with itself):

```
In [12]: def permute(digits, start, size, length):
             if length == 0:
         #        yield
                 if size > 1:
                     # "Simulates" the permutation of the "size"
                     # first elements of L[start: ] by computing
                     # what would be the last permutation.
                     if size % 2 or size == 2:
                         digits[start], digits[start + size - 1] =\
                                        digits[start + size - 1], digits[start]
                     elif size == 4:
                         digits[start: start + 3], digits[start + 3] =\
                                       digits[start + 1: start + 4], digits[start]
                     else:
```

```
                        digits[start: start + 2],
                        digits[start + 2: start + size - 2],\
                        digits[start + size - 2],
                        digits[start + size - 1] =\
                                    digits[start + size - 3: start + size - 1],\
                                    digits[start + 1: start + size - 3],
                                    digits[start + size - 1],
                                    digits[start]
            print(digits[: size], digits[size: ])

        for size in range(2, 10):
            permute([str(i) for i in range(10)], 0, size, 0)
```

```
['1', '0'] ['2', '3', '4', '5', '6', '7', '8', '9']
['2', '1', '0'] ['3', '4', '5', '6', '7', '8', '9']
['1', '2', '3', '0'] ['4', '5', '6', '7', '8', '9']
['4', '1', '2', '3', '0'] ['5', '6', '7', '8', '9']
['3', '4', '1', '2', '5', '0'] ['6', '7', '8', '9']
['6', '1', '2', '3', '4', '5', '0'] ['7', '8', '9']
['5', '6', '1', '2', '3', '4', '7', '0'] ['8', '9']
['8', '1', '2', '3', '4', '5', '6', '7', '0'] ['9']
```

We now complete the implementation of **permute()**, which is essentially the same as that of **recursive_heap_permute()**:

```
In [13]: def permute(digits, start, size, length):
             if length == 0:
                 yield
                 if size > 1:
                     # "Simulates" the permutation of the "size"
                     # first elements of L[start: ] by computing
                     # what would be the last permutation.
                     if size % 2 or size == 2:
                         digits[start], digits[start + size - 1] =\
                                             digits[start + size - 1], digits[start]
                     elif size == 4:
                         digits[start: start + 3], digits[start + 3] =\
                                              digits[start + 1: start + 4], digits[start]
                     else:
                         digits[start: start + 2],
                         digits[start + 2: start + size - 2],\
                         digits[start + size - 2],
                         digits[start + size - 1] =\
                                     digits[start + size - 3: start + size - 1],\
                                     digits[start + 1: start + size - 3],
                                     digits[start + size - 1],
                                     digits[start]
```

```
            else:
                size -= 1
                length -= 1
                for i in range(size):
                    yield from permute(digits, start, size, length)
                    if size % 2:
                        digits[start + i], digits[start + size] =\
                                           digits[start + size], digits[start + i]
                    else:
                        digits[start], digits[start + size] =\
                                                 digits[start + size], digits[start]
                yield from permute(digits, start, size, length)
```

As previously mentioned, **permute()** will be called twice: once to assign candidate digits to **letters_starting_a_word**, and a second time to assign candidate digits to **letters_not_starting_a_word**. Such is the purpose of the following function, **generate_possible_solutions()**, whose second argument, **length_1**, is meant to receive the value **len(letters_starting_a_word)**, while its third argument, **length_2**, is meant to receive the value **len(letters_not_starting_a_word)**. After the second call to **permute()**, **generate_possible_solutions()** can pause so that **digits** can be used in its current state, with the last **length_1** members of **digits** taking some nonzero values to assign to **letters_starting_a_word**, whose members make up the end of **all_letters**, and with the **length_2** previous members of **digits** taking some values (with 0 possibly included) to assign to **letters_not_starting_a_word**, whose members make up the beginning of **all_letters**.

When **permute(digits, 1, 9, adjusted_length_1)** pauses, we save the **10 - length_1** first members of **digits**, to resume the search for candidate digits for **letters_starting_a_word** after the search for candidates digits for **letters_not_starting_a_word**, operating on those **10 - length_1** first members of **digits**, has completed: that section of **digits** can then be restored to its state before **permute(digits, 0, size, adjusted_length_2)** was called, so that **permute(digits, 1, 9, adjusted_length_1)** can resume execution where it left.

A second aspect of **generate_possible_solutions()** is that **length_1** or **length_2** can be decreased by 1, in case their values are such that one element of the permutations of digits of length **length_1** or **length_2**, respectively, that **generate_possible_solutions()** is meant to produce, is fully determined by the others; it is a minor optimisation:

```
In [14]: def generate_possible_solutions(digits, length_1, length_2):
             adjusted_length_2 = length_2
             # Once nonzero digits have been allocated to all letters
             # starting a word, if all digits that remain have to be used
             # for the letters not starting a word, then there is no need
             # to allocate the last one once all others have been allocated.
             if length_2 == 10 - length_1:
                 adjusted_length_2 -= 1
             adjusted_length_1 = length_1
             # If all nonzero digits have to be used for the letters
             # starting a word, then there is no need to allocate
             # the last one once all others have been allocated.
             if length_1 == 9:
```

```
            adjusted_length_1 -= 1
        size = 10 - length_1
        for _ in permute(digits, 1, 9, adjusted_length_1):
            first_size_digits = list(digits[: size])
            yield from permute(digits, 0, size, adjusted_length_2)
            digits[: size] = first_size_digits
```

To replace letters by digits in a string, **str.maketrans()** together with **str.translate()** offer a good solution. When provided as arguments two strings of the same length **str.maketrans()** creates a dictionary whose keys and values are the point codes of the characters in both strings, position by position:

```
In [15]: str.maketrans('ABCD', 'abcd')
         'BCDEBCDE'.translate(str.maketrans('ABCD', 'abcd'))

Out[15]: {65: 97, 66: 98, 67: 99, 68: 100}

Out[15]: 'bcdEbcdE'
```

When **str.maketrans()** receives a single argument, that argument can be a dictionary whose keys and values are the characters to replace and the replacing characters, respectively:

```
In [16]: dict(zip(['A', 'B', 'C', 'D'], ['a', 'b', 'c', 'd']))
         str.maketrans(dict(zip(['A', 'B', 'C', 'D'], ['a', 'b', 'c', 'd'])))
         'BCDEBCDE'.translate(str.maketrans(dict(zip(['A', 'B', 'C', 'D'],
                                                     ['a', 'b', 'c', 'd']
                                                    )
                                                )
                                            )
                                    )

Out[16]: {'A': 'a', 'B': 'b', 'C': 'c', 'D': 'd'}

Out[16]: {65: 'a', 66: 'b', 67: 'c', 68: 'd'}

Out[16]: 'bcdEbcdE'
```

For our purpose, it seems preferable to avoid creating strings from lists and make use of the second option:

```
In [17]: all_letters
         cryptarithm

         digits = ['0', '5', '3', '8', '4', '2', '9', '7', '1', '6']
         cryptarithm.translate(str.maketrans(dict(zip(all_letters, digits))))

         digits = ['3', '0', '6', '4', '7', '5', '1', '9', '8']
         cryptarithm.translate(str.maketrans(dict(zip(all_letters, digits))))

Out[17]: ['O', 'C', 'K', 'U', 'E', 'L', 'F', 'Y', 'D']
```

```
Out[17]: 'DO + YOU + FEEL == LUCKY'

Out[17]: '10 + 708 + 9442 == 28537'

Out[17]: '83 + 934 + 1775 == 54069'
```

Checking whether the assignment of digits to the letters in the cryptarithm is a solution is then easy with the **eval()** function, which can be used to evaluate any syntactically correct Python statement, and for that reason, is not safe and should not be used in applications where malicious attacks are possible:

```
In [18]: eval('24 + 145 + 7889 == 95031')
         eval('57 + 870 + 9441 == 10368')

Out[18]: False

Out[18]: True
```

Putting things together:

```
In [19]: digits = [str(i) for i in range(10)]
         for _ in generate_possible_solutions(digits,
                                              len(letters_starting_a_word),
                                              len(letters_not_starting_a_word)
                                             ):
             equation = cryptarithm.translate(
                            str.maketrans(dict(zip(all_letters,
                                                   digits[10 - len(all_letters): ]
                                                  )
                                               )
                                          )
                                             )
             if eval(equation):
                 print(equation)

57 + 870 + 9441 == 10368
```

We could with less work, but learning much less, have designed the cryptarithm solver making use of the **permutations** function from the **functools** module, discarding permutations that assign 0 to the letters in **letters_starting_a_word**; **permutations()** takes as optional second argument argument a number which can be less than the number of elements in the iterable passed as first argument, and that plays the role of **length** in our implementation of **permute()** (with **start** set to 0 and **size** to the length of the first argument):

```
In [20]: list(permutations(range(4), 0))
         tuple(permutations(range(4), 2))
         set(permutations(range(4)))

Out[20]: [()]
```

```
Out[20]: ((0, 1),
          (0, 2),
          (0, 3),
          (1, 0),
          (1, 2),
          (1, 3),
          (2, 0),
          (2, 1),
          (2, 3),
          (3, 0),
          (3, 1),
          (3, 2))

Out[20]: {(0, 1, 2, 3),
          (0, 1, 3, 2),
          (0, 2, 1, 3),
          (0, 2, 3, 1),
          (0, 3, 1, 2),
          (0, 3, 2, 1),
          (1, 0, 2, 3),
          (1, 0, 3, 2),
          (1, 2, 0, 3),
          (1, 2, 3, 0),
          (1, 3, 0, 2),
          (1, 3, 2, 0),
          (2, 0, 1, 3),
          (2, 0, 3, 1),
          (2, 1, 0, 3),
          (2, 1, 3, 0),
          (2, 3, 0, 1),
          (2, 3, 1, 0),
          (3, 0, 1, 2),
          (3, 0, 2, 1),
          (3, 1, 0, 2),
          (3, 1, 2, 0),
          (3, 2, 0, 1),
          (3, 2, 1, 0)}
```