NOTES ON CRYPTARITHM SOLVER AND PERMUTATIONS

ERIC MARTIN

1. Heap’s algorithm

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.

| 0 | all permutations of the first $n$ numbers |
|   |                                           |
| n |                                           |
| n-1 |

$0 \stackrel{\bullet}{\to} n \leftarrow m_0 \leftarrow m_1 \leftarrow \cdots \leftarrow m_{n-2} \leftarrow m_{n-1}$

| $m_0$ | all permutations of the first $n$ numbers |
| $m_1$ |

So the algorithm generates all permutations of the form $L \ast n$, then all permutations of the form $L \ast m_0$, then all permutations of the form $L \ast m_1$, \ldots, and eventually all permutations of the form $L \ast m_{n-1}$. 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 $n = 3$.

```
0 1 2 0 1 2 3 1 0 3 1 0 0 2 3 0 2 3 3 2 1 3 2 1
1 0 0 2 2 1 1 3 3 0 0 1 2 0 0 3 3 2 2 3 3 1 1 2
2 2 1 1 0 0 0 0 1 1 3 3 3 3 3 2 2 0 0 1 1 2 2 3 3
3 3 3 3 3 3 3 2 2 2 2 2 1 1 1 1 1 1 0 0 0 0 0 0
```

Note that starting with $(0, 1, 2, 3)$, Heap’s algorithm produces $(1, 2, 3, 0)$ as last permutation. We will see that starting with $(0, 1, 2, 3, 4, 5)$, it would produce $(3, 4, 1, 2, 5, 0)$ as last permutation; starting with $(0, 1, 2, 3, 4, 5, 6, 7)$, it would produce $(5, 6, 1, 2, 3, 4, 7, 0)$ as last permutation. More generally, starting with $(0, 1, 2, \ldots, 2n)$, Heap’s algorithm will produce as last permutation $(2n - 1, 2n, 1, 2, \ldots, 2n - 2, 2n + 1, 0)$.

Note that starting with $(0, 1, 2)$, Heap’s algorithm produces $(2, 1, 0)$ as last permutation. We will see that starting with $(0, 1, 2, 3, 4)$, it would produce $(4, 1, 2, 3, 0)$ as last permutation; starting with $(0, 1, 2, 3, 4, 5, 6)$, it would produce $(6, 1, 2, 3, 4, 5, 0)$ as last permutation. More generally, starting with $(0, 1, 2, \ldots, 2n)$, Heap’s algorithm will produce as last permutation $(2n, 1, 2, \ldots, 2n - 1, 0)$.

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, \ldots, and the
The following is a possible implementation of Heap's algorithm.

```python
def permute(L):
    for L in heap_permute(L, len(L)):
        yield L

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

2. PROOF OF CORRECTNESS

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 \((2n - 1, 2n, 1, 2, \ldots, 2n - 2, 2n + 1, 0)\)

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

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, . . . , 2n, 2n + 1) have been generated, ending in (2n − 1, 2n, 1, 2, . . . , 2n − 2, 2n + 1, 0), so after all permutations of (0, 1, 2, . . . , 2n, 2n + 1) × 2n + 2 have been generated, ending in (2n − 1, 2n, 1, 2, . . . , 2n − 2, 2n + 1, 0) × 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 a 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, . . . , 2n − 1, 2n, 2n + 1) × 0. So we have established that (1) with n replaced by n + 1 holds.

3. Notes on the implementation of the cryptarithm solver

The first version is a minor adaptation of code written by Raymond Hettinger as part of ActiveState Code Recipes. It uses the permutations function, imported from itertools (and also the findall function, imported from re). The second version does not import anything.

The first version filters out the permutations that assign 0 to one of the letters that start a word; the second version does not produce those permutations. The first version creates a string where the letters starting a word come first, followed by the letters not starting a word; it is the other way around for the second version. For instance, with the cryptarithm ENDORYSM, the first version creates a string which could be SMENDORY, whereas the second version creates a string which could be ENDORYSM. Let us still use that example to explain how we proceed in the second version.

- Starting with the list \( L = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) \), we use the generalisation of Heap’s algorithm to generate lists of the form \( L_1 L_2 \), one for each possible list \( L_2 \) of two nonzero digits (so we ignore 0, as if \( L \) started with the element of index 1). This determines a possible assignment to SM.
- For each list of the form \( L_1 L_2 \) generated as described, we make a copy of \( L_1 L_2 \) and we use again the generalisation of Heap’s algorithm to generate from the copy lists of the form \( L_{11} L_{12} L_2 \), one for each possible list \( L_{11} \) of six digits, amongst those in \( L_1 \) (so 0 is now allowed but we ignore \( L_2 \), as if we were working with a list that ended at index 7). We return the last 8 digits of \( L_{11} L_{12} L_2 \), that is, \( L_{12} L_2 \), allocating the digits in \( L_{12} \) to ENDORY and the digits in \( L_2 \) to SM.