Project 2 Report: Generating Permutations

Zhang Naifu
znf18@mails.tsinghua.edu.cn

9 October 2018

Abstract

Project 2 looks to enumerate all permutations from {1,2,...,n} without loss or repetition, where n is a non-zero natural number input by the user. The Python program would also return the number of permutations, which is $n!$

Permutations are generated using Heap's algorithm[1]- not to be confused with heap sort - which is chosen for its efficiency and will be the focus of this report.

1 Implementation

Given n elements, Heap's algorithm constructs every permutation by swapping a single pair of elements in the set. The recursive version of the algorithm is reproduced below in pseudocode without comment for subsequent reference.

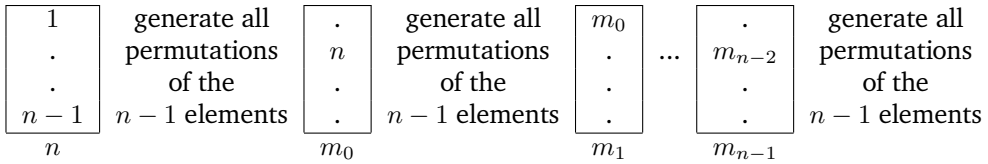
Algorithm: `heaps_recursive`

Input: (a, n), a list of elements and its length
Result: Every permutation of the n elements without loss or repetition

```
1 if this is the original permutation a: print(a)
2 i = 0
3 loop:
4     if n > 2: heaps_recursive(a, n − 1)
5     if n <= i + 1: break
6     elif n is odd: swap last element with 1st element
7     else n is even: swap last element with element i
8     print(a)
9     i+ = 1
```

The algorithm works by enumerating all permutations of the first $n-1$ elements, appending the n_{th} element at the end of each permutation *(line 4)*. It then swaps the one of the $n-1$ elements with the n_{th} element. Depending on whether n is odd or even, *lines 6-7* decide which element to swap - this is the only trick of the algorithm.

The algorithm does such swaps for a total of $n-1$ times, each time recursively permutating the new set of $n-1$ elements of course. The process could be illustrated graphically as such.



[1]While checking Heap's algorithm implementation on Wikipedia, the author discovered a trivial bug, which the author promptly corrected as his very first Wikipedia edit. Details at Wikipedia logs.

With $n = 3$, we have the following output, including the original permutation.

Sample Output

```
$ ./Project_2.py

Please enter a positive integer: 3
[1, 2, 3]
[2, 1, 3]     # swapped elements[0] & elements[1]
[3, 1, 2]     # swapped elements[0] & elements[2]
[1, 3, 2]     # swapped elements[0] & elements[1]
[2, 3, 1]     # swapped elements[0] & elements[2]
[3, 2, 1]     # swapped elements[0] & elements[1]
```

2 Correctness

What's of real interest is that the non-intuitive Heap's algorithm produces the correct output. We can prove this by induction.

Hypothesis: Heap's enumerates all $n!$ unique permutations of length n without loss or repetition.
Base case: For $n = 2$, Heap's is trivially correct, and the set of all permutations is {[1,2],[2,1]}.
Assume: Heap's correctly enumerates $n!$ permutations for n.
Prove: Heap's correctly enumerates $(n + 1)!$ permutations for $n + 1$.

Given the assumed correctness of the n run, the $n + 1$ run is correct iff it permutates a different set of n elements each recursion, or equivalently, iff each swap places a different element at the $n + 1_{th}$ position.

For $n = 2$, this is easy. In the Sample Output above, permutation 3 and permutation 5 each places a different element at the 3_{rd} position, due to the swapping mechanism in *lines 6-7* of **Algorithm:** heaps_recursive.

However, extension of the proof to a general n turns out to be harder and messier. Eric Martin provides a technically true but rather convoluted proof - not reproduced here - that involves going through a total of 13 steps for odd and even n. This author fails to work out a shorter and more elegant proof after several days of agonizing. If such a proof exists, it is certainly not made accessible.

3 Complexity

We examine the loop in *lines 2-9* of **Algorithm:** heaps_recursive. The loop is run $n - 1$ times, each time incurring the cost of the recursion *(line 4)* and a constant cost from the swap and printing *(lines 8-9)*. On the n_{th} loop, we do the recursion too, but break *(line 5)* before incurring the constant cost. Therefore,

$$T(n) = \sum_{i=0}^{n-2}(T(n-1) + C) + T(n-1)$$

Equivalently

$$T(n) = \sum_{i=1}^{n}\sum_{i=1}^{n-1}...\sum_{i=1}^{3}(T(2) + C) - C$$

Since

$$T(2) = C$$

Then

$$T(n) = C(n! - 1)$$

The main loop actually enumerates $n! - 1$ permutations. *Line 1* of **Algorithm:** heaps_recursive prints out the additional original permutation.

As expected, Heap's is asymptotically efficient, as it's not possible to enumerate $n!$ permutations in less than $O(n!)$.

Although similar to Steinhaus-Johnson-Trotter algorithm, Heap's algorithm is more memory efficient as it does not keep track of the offset used in SJT.

Heap's algorithm is also compared with the implementation of *itertools.permutations()* in Python (i.e. not the library in C). Both produce comparable runtimes.

References

Wikipedia *https://en.wikipedia.org/wiki/Heap's_algorithm*
B.R. Heap (1963) *Permutation by Interchanges.* The Computer Journal.
E. Martin (2015) *Notes on Cryptarithm Solver and Permutations.*
Python itertools documentation *https://docs.python.org/3.5/library/itertools.html#itertools.permutations*