Permutation Generation with Heap's Algorithm and the Steinhaus-Johnson-Trotter Algorithm

Ryan Bernstein
Ruben Niculcea
Levi Schoen

Contents

1 Heap's Algorithm

1.1 Proof of Termination

1.2 Proof of Correctness

1.3 Time Complexity

The significant portion of our implementation of Heap's Algorithm is as follows:

```
 3 def heaps(set, time=False):
 4   def innerHeaps(n, set):
 5     if n == 1:
 6       print set
 7     else:
 8       for i in range(n):
 9         innerHeaps(n-1, set)
10         if n % 2 == 1: #odd number
11           j = 0
12         else:
13           j = i
14         set[j], set[n-1] = set[n-1], set[j] #swap
```

We'll analyze the running time of this algorithm in terms of swaps, comparisons, assignments, and processes. From the code above, we can count how many times each of these operations is performed, *not* including work done within the recursive calls.

Swaps ($O(1)$)

The swap operation appears only once in our implementation, on line 14. This line is never executed when $n = 1$; if $n > 1$, the loop containing it executes n times.

Comparisons ($O(1)$)

A comparison appears on line 5 regardless of the current value of n. Another comparison appears within the loop on line 10, which executes n times for all $n > 1$. The second comparison also includes a division for modular arithmetic; we will also consider this an $O(n)$ operation.

Assignments ($O(1)$)

Not counting the assignments that appear within the swaps, a single assignment occurs on either line 11 or line 13. This is dependent on the result of the comparison on line 10.

Processes/`prints` ($O(n)$)

We only process a permutation in the base case of our recursion, when $n = 1$. Thus we execute a process once when $n = 1$ and 0 times in any other case.

1.3.1 Factoring In Recursion

Our total number of operations (not counting recursion) can be represented by the following table:

Input Size	1	2	3	4	5	$n > 1$
Swaps	0	2	3	4	5	n
Comparisons	1	3	4	5	6	$n + 1$
Assignments	0	2	3	4	5	n
Processes	1	0	0	0	0	0
Recursive calls	0	2	3	4	5	n

Looking only at the recursive calls, it becomes clear that the number of recursive calls made for an input of size n is 0 when $n = 1$ and n for any $n > 1$ (we can confirm this by looking at the loop in our code). It follows that if we wish to include operations executed within recursive calls, the total number of times that each operation will execute on an input of size n will be the number of times that this operation executes in the work function plus n times the number of times that operation is performed for an input of size $n - 1$.

This means that with recursion factored in, the total number of swaps, assignments, and recursive calls can be expressed as follows:

$$
\begin{aligned}
T(n) &= n \cdot T(n-1) + n \\
\frac{T(n)}{n!} &= \frac{T(n-1)}{(n-1)!} + \frac{1}{(n-1)!} && \text{Divide by } n! \\
&= \sum_{i=1}^{n} \frac{1}{(i-1)!} && \text{Since } T(1) = 0 \\
&= \sum_{i=1}^{n-1} \frac{1}{i!} \\
T(n) &= n! \cdot \sum_{i=1}^{n-1} \frac{1}{i!} && \text{Multiply by } n!
\end{aligned}
$$

The number of comparisons is similar:

$$
\begin{aligned}
T(n) &= n \cdot T(n-1) + n + 1 \\
\frac{T(n)}{n!} &= \frac{T(n-1)}{(n-1)!} + \frac{1}{(n-1)!} + \frac{1}{n!} && \text{Divide by } n! \\
&= 1 + \frac{1}{(n-1)!} + \frac{1}{n!} && \text{Since } T(1) = 0! = 1 \\
&= 1 + \sum_{i=1}^{n-1} \frac{1}{i!} + \sum_{i=1}^{n} \frac{1}{i!} \\
&= 1 + 2\sum_{i=1}^{n-1} \frac{1}{i!} + \frac{1}{n!} \\
T(n) &= n! + 2n! \sum_{i=1}^{n-1} \frac{1}{i!} + 1 && \text{Multiply by } n!
\end{aligned}
$$

Combining this with the time complexity of each operation, the overall time complexity is therefore as follows:

$$
\begin{aligned}
&2 \cdot \left(n! \cdot \sum_{i=1}^{n-1} \frac{1}{i!} \right) + \left(n! + 2n! \cdot \sum_{i=1}^{n-1} \frac{1}{i!} + 1 \right) + n \\
&= n! \cdot \left(1 + 4 \cdot \sum_{1}^{n-1} \frac{1}{i!} \right) + 1 + n
\end{aligned}
$$

1.4 Space Complexity

Looking at the code above, we can see that we use the same set object throughout the series of recursive calls. Beyond that, since our recursion decrements n each time and stops at 1, we will have at most n (constant-size) stack-frames at any given time.

The space complexity of Heap's algorithm is therefore $O(n)$.

2 The Steinhaus-Johnson-Trotter Algorithm

2.1 Proof of Termination

2.2 Proof of Correctness

2.3 Time Complexity

Our proof of correctness shows that this algorithm correctly generates all $n!$ permutations of an n-element set; this means that our loop will run $n!$ times. However, unlike Heap's algorithm, this loop will not perform the same number of steps every time. Each loop iteration will perform:

- An $O(n)$ search for the largest mobile element
- A single $O(n)$ `process`
- Some number of $O(1)$ direction reversals
- One $O(1)$ swap

How can we determine the number of direction reversals? The key lies in our definition of mobility and the following fact: for each case in which item a_i is the largest mobile element (LME), we will perform a reversal for each item $a_j > a_i$ in our set. We can therefore define the total number of reversals across *all* iterations as follows:

$$R_{\text{total}} = \sum_{i=1}^{n} ((\text{number of times } a_i \text{ is the LME}) \cdot (n - i))$$

2.3.1 Solving for Largest Mobile Element Counts

We say that an element is mobile if and only if it points to a smaller element. Based on this definition, I propose the following axioms:

Axiom 2.1 *Since a_n is the largest element in the set, this means that any time it is mobile, it will be the largest mobile element.*

Axiom 2.2 *Furthermore, a_n will only be* immobile *if it points at an edge. At this point, if another mobile element exists, a_n will reverse direction; otherwise, no mobile elements exist and our loop terminates. Since we are sweeping a_n back and forth across all $(n-1)$-element permutations, it will point to an edge and lose mobility once for each of these.*

Axiom 2.3 *An element a_i can be mobile in an n-element permutation π if it would be mobile in the i-element permutation $\pi - a_j$ for all $j > i$.*

Extending axiom 2.3, we can see the following: since we're assuming that each possible permutation of an $(n-1)$-element set appears n times in permutations of an n-element set, any $(n-1)$-element permutation π_{n-1} in which a_{n-1} is the LME will appear n times in permutations of n elements. Of these, a_n will be the LME in any case where it is mobile by axiom 2.1. Since a_n will be mobile for all but one of the n repetitions of π_{n-1}. In this permutation π, a_{n-1} is the LME of $\pi - a_n$ and a_n is immobile. Therefore, a_{n-1} is the LME in π.

We can therefore conclude that in permutations of n elements, a_{n-1} is the LME exactly once for every $(n-1)$-element permutation for which a_{n-1} is the LME. By axiom 2.1, this is any $(n-1)$-element permutation in which a_{n-1} is mobile.

Axiom 2.2 tells us that a_n loses mobility exactly once for each $(n-1)$-element permutation, of which there are $(n-1)!$. Therefore, the number of times a_n is the largest mobile element in a permutation of m elements for all $m \geq n$ is equal to exactly $n! - (n-1)!$.

2.4 Space Complexity