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

Ryan Bernstein
Ruben Niculcea
Levi Schoen

Abstract

A formal mathematical analysis of two algorithms – Heap's and Steinhaus-Johnson-Trotter – for generating permutations of a set. Implementation, execution and analysis of space and time performance of these two algorithms in Python. Additional consideration is given to the nature of generating permutations and how these two algorithms differ in solving that problem.

Code and detailed test results for this project can be found at the URL below:
https://github.com/RyanLB/CS-350-Permutation-Generation-Algorithms

Contents

The application and study of algorithms in the field of combinatorics-particularly permutation generation-is an exciting area of theoretical study, albeit one with dull practical outcomes and application. This is due to the fact that generating all distinct permutations of a set is NP hard; since there are $n!$ possible permutations of an n-element set, even simply outputting each permutation will take $n!$ steps. As such, no deterministic algorithm exists that can generate all permutations of a set in less than $O(n!)$ time. The differences in the algorithms are wrinkles that allow for more space efficient computations, or that merely exist to show the cleverness of the designer with a novel application of a design pattern to the problem.

Algorithms for this problem fall into two camps: those that generate a unique permutation of a set and those that use exchange methods to generate *all* permutations of that set. Our first algorithm, Heap's Algorithm, is a recursive twist on the exchange method. The second, Steinhaus-Johnson-Trotter, is an adjacent exchange method implemented iteratively with one loop structure, although this algorithm can be implemented recursively as well.

The intractable nature of this problem has lead to the discovery and refinement of more efficient combinatorial search optimization algorithms, many of which emphasize the use of backtracking and mathematical programming; perhaps most notable is the solution to the four coloring graph problem.

1 Heap's Algorithm

Heap's algorithm is a "divide-and-conquer" permutation generation algorithm. It divides the set into subsets that all end with the same element and then generates all of the permutations that end with that element. It works by iterating from 1 to the cardinality of the set. Each iteration makes a new recursive call with $(n-1)$ as the new cardinality of the set. This recursion will continue until we have reached the case where n is equal to 1; it then outputs the current state of the set as a permutation. Between each recursive call, we check to see if n is odd. If so, we switch the first element of the set with the last element of the set. If n is even, we switch the ith element of the set with the last element of the set (where i the number of current iteration). This guarantees that each iteration will produce all the permutations that end with the element that has been moved into the last position in the set.

1.1 Proof of Termination

At each step of Heap's, we recurse and decrement n by 1. Eventually we reach the the base case of 1 element, as n must be a positive integer, so no infinite descent is possible.

1.2 Proof of Correctness

// I don't even know what's going on here

Given an array Elements[n] with n elements, proof follows by induction.

$A(1)$ is obviously true; a one element set is sorted Assume $A(n)$ for all $n \geq 1$. Prove for $A(n+1)$ We know the solution for $A(n)$ is distinct, and does not contain the element Elements[$n+1$]. First compute $A(n)$, then run Heap's by inserting Elements[$n+1$] into each subset of $A(n)$. Since each subset of $A(n)$ is distinct, and disjoint with Elements[$n+1$], this insertion preserves the distinctness of our result and gives us the correct solution to $A(n+1)$.

1.3 Time Complexity

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

```
def heaps(set, time=False):
  def innerHeaps(n, set):
    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:

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

Because comparisons have $n+1$ comparisons before recursion, we can use a similar process to discover that the total number of comparisons can be expressed as:

$$n! \cdot \sum_{i=1}^{n-1} \frac{1}{i!} + n! \cdot \sum_{i=1}^{n} \frac{1}{i!} = 2! \cdot \sum_{i=1}^{n-1} \frac{1}{i!} + 1$$

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

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

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

The Steinhaus-Johnson-Trotter algorithm is an example of a minimal change/decrease by one algorithmic approach to the permutation problem. It is based on the observation that all permutations of n elements can be generated by taking the element a_n and placing it in every possible position in each permutation of $n-1$ elements.

This behavior is induced by assigning each element a direction; the direction of each element is initialized to be left. We call an element mobile if and only if it points to a smaller element. As long as our set contains a mobile element, we take the largest mobile element a_ℓ and swap it with the element to which it points; at this point, we reverse the direction of every element larger than a_ℓ.

2.1 Proof of Termination and Correctness

By induction on n, the validity of the above algorithm is proven as follows. As a base case, we can easily verify that the algorithm is true for sets of two elements. Initially a_n is active and being the largest element moves left through all positions until it is not pointing to a smaller element. Now a_n is immobile and at an extreme position. Assuming that the algorithm works for $n-1$ integers, the next permutation of $n-1$ elements is generated, with n playing no part. At this point, a_n becomes mobile again and passes unobstructed to the opposite extreme. The process continues, with a_n reversing direction for each permutation $n-1$ elements, until the last permutation of $n-1$ elements is generated, leaving all but a_n immobile. After the final traversal of n, it also immobilized and the algorithm terminates. Thus each of the $n!$ permutations is generated once and only once, and this algorithm will terminate in finite time.

2.2 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:

- A search for the largest mobile element, which is broken into n mobility checks and c comparisons against the current largest mobile element (where c is the sum of the numbers of mobile elements for each iteration)
- A single $O(n)$ `process`
- Some number of $O(1)$ direction reversals
- One $O(1)$ swap (except for the last iteration, in which no swaps occur)

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:

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

2.2.1 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 and only 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 $n! - (n-1)!$.

Substituting this back into our equation for R_{total}, we get the following result:

$$R_{\text{total}} = \sum_{i=1}^{n-1} ((i! - (i-1)!) \cdot (n-i))$$

2.2.2 Mobility Counts

We can solve for mobility counts in a similar fashion. Every $(n-1)$-element permutation appears n times in the set of n-elements. Element a_n will be swept across each position in π_{n-1}. The combination of these two facts tells us that if an element a_i is mobile in π_{n-1}, it will be mobile in every n-element permutation that extends π_{n-1} except for the one case in which a_n is between a_i and the (smaller) element that a_i points to.

We know that a_n is mobile in $(n! - (n-1)!)$ of the n-element permutations in which it first appears. By this reasoning, it will now be mobile in $n \cdot (n! - (n-1)!)$ permutations of $n+1$ elements, $(n+1) \cdot n \cdot (n! - (n-1)!)$ permutations of $n+2$ elements, and so on.

We can therefore solve for the total number of mobile elements across all iterations (which is also the total number of comparisons made to find the largest mobile element in each iteration) as follows:

$$C_{\text{total}} = \sum_{i=1}^{n} \left((i! - (i-1)!) \cdot \frac{(n-1)!}{(i-1)!} \right)$$

The overall time complexity of the Steinhaus-Johnson-Trotter algorithm is therefore:

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

2.3 Space Complexity

Since this is an iterative algorithm, we can accomplish everything within a single stack frame (while we've divided our code with function calls, the number of simultaneously active stack frames remains constant with respect to n). This algorithm needs only an n-element array to hold its elements and another n-element array to hold the direction in which each element points.

3 Testing Methodology

3.1 Correctness Tests

Both algorithms were tested for correctness on random sets of up to six elements. These elements were then sorted, as this is required by the Steinhaus-Johnson-Trotter algorithm. The sets were then passed to the python permutation generator and the algorithm in test, and the results were compared to ensure that they were the same.

3.2 Time Complexity Profiling

In order to ensure that our time complexities were correct, basic operations (e.g. swaps, comparisons, assignments, and the like) were factored out into their own functions. Python's `cProfile` framework was then used to count the number of times each function was called.

3.2.1 Direct Comparisons and Hypotheses

By breaking down our functions into swaps, comparisons, assignments, and recursive calls, we can directly compare the running time of the two algorithms for an input of size n. For the purposes of comparison, we consider each direction reversal in SJT to be comprised of a comparison and an assignment and each mobility check to be comprised of two comparisons.

	Heap's	SJT	Larger
Swaps	$n! \cdot \sum_{i=1}^{n-1} \frac{1}{i!}$	$n! - 1$	Heap's
Compares	$2n! \sum_{i=1}^{n-1} \frac{1}{i!} + 1$	$n \cdot n! + 2\sum_{i=1}^{n} \left((i! - (i-1)!) \cdot \frac{(n-1)!}{(i-1)!}\right) + \sum_{i=1}^{n} ((i! - (i-1)!) \cdot (n-i))$	SJT
Assigns	$n! \cdot \sum_{i=1}^{n-1} \frac{1}{i!}$	$\sum_{i=1}^{n} ((i! - (i-1)!) \cdot (n-i))$	Heap's
Processes	$n!$	$n!$	Tie
Calls	$n! \cdot \sum_{i=1}^{n-1} \frac{1}{i!}$	0	Heap's

With this in mind, we expect Heap's algorithm to have the longest real-world running time.

4 Test Results

4.1 Correctness Results

Both algorithms passed the correctness check without issue.

4.2 Time Complexity Results

Average Running Times Across 10 Tests

Input Size	5	6	7	8
Heap's Average Running Time (seconds)				
SJT Average Running Time (seconds)				

5 Conclusions

This is not the result reached by Robert Sedgewick in his 1977 paper "Permutation Generation Methods". Sedgewick concluded that Heap's algorithm was, at the time, the fastest method for generating permutations. However, he did this by hand-writing assembly code in a pseudo-assembly language that he developed himself.

It is the opinion of this group that if both algorithms are implemented reasonably in a high-level programming language (e.g. Python), the Steinhaus-Johnson-Trotter algorithm will perform better.