Heap's Algorithm for All Permutations Using a TI84

Timothy W. Jones

July 20, 2025

Abstract

Heap's algorithm uses recursion in a way that can seem baffling. The various youtube videos, Wikipedia accounts, and textbooks all mention that the key swap criterion is based on the even value of an index. There is a simpler way to understand the algorithm. Given the permutation of one object is just a print it out (program PERM1) then the permutations of two objects is print the original out (a call to PERM1), then rotate it and call PERM1 on the result. We show using a TI84 program how this pattern can generate all permutations on three, four, five, and six objects and make it transparent how recursion can work for the general case.

Introduction

Most computer science students will know about HEAPSORT. This algorithm is not eponymous but suggested by a graph that resembles a heap and performs a sort in $n\log n$ time – fast. *Heap* [1] is also suggested by the use of the same data structure for garbage collection.

This article is about the eponymous *Heap algorithm* that generates all permutations of an array's elements [2]. Brian Robert Heap was actually a biologist. His algorithm appears at first flush to be wondrously clever and mysterious. Watching videos of coders discussing it that seems to be their general disposition [4]. Wikipedia offers some intriguing graphs and looks at odd and even cases in some detail, but doesn't really motivate how the

algorithm could have been discovered. The code provided in videos and at Wikipedia possesses this cryptic swap when k is even idea. Line 23 in Figure 1 shows this. This code is in Javascript and it gives an amalgam of various on-line code and my own patches. It does work, but to my mind it is not apparent why it should work.

```
13  function heapsPermute(array, n)
14  {
15    n = n || array.length;
16    results = [];
17    if (n === 1) {
18      results.push(array);
19      myString += array.join('').toString();
20    } else {
21      for (var i = 1; i <= n; i++) {  //i = i+1
22        heapsPermute(array, n - 1);  //deleting , results
23        if (n % 2) {
24          var j = 1;
25        } else {
26          var j = i;
27        }
28        swap(array, j - 1, n - 1);
29      }
30    }
31    return results;
32  }
```

Figure 1: Heap's algorithm is all too brief. Why should even and odd cases be treated differently?

That said, some set of swaps or flips or, technically transpositions should deliver each permutation according to group theory [3], but Heap doesn't reference any abstract algebra or group theory text. The beginning programming book *Data Structures and Program Design in C* does mention Heap's algorithm but doesn't elucidate it [5].

There is a simpler idea that drops the even versus odd cases and goes right to what is sought: rotations in both even and odd cases that are sent in recursive fashion to early cases. We demonstrate this with programs on a TI84 calculator.

Hand Calculations

Let's create a program for generating all the permutations of one element, the number 1 in the list L_1. In generally we will limit our discussion to such $1 \rightarrow n$ consecutive lists. PERM1 consists of the lines $\{1\} \rightarrow L_1$ and Disp L_1. There needs no ghost from the grave for this first program. Notice that the TI84 calculator can make lists and work with their elements: $L_1(1)$ gives the first element and Disp L_1 will display whatever is in L_1. That is: we don't have to make for loops for such things and can use various built in functions on these array like TI84 lists.

Here is the entire concept: make a list with two elements $\{1, 2\}$ and notice that the 0 rotation is the original and one rotation (or RIGHT SHIFT)

gives another permutation. We apply PERM1 to both and we have the two permutations of $\{1, 2\}$. For $\{1, 2, 3\}$ we will generate three rotation (no rotation counts as a rotation): 123, 312, and 231. The last element rotates to the first. PERM2 is called on each of these: it generates the original and then rotates the first two. This gives $\{123, 213, 312, 132, 231, 321\}$; all the permutations of these three objects.

Once more on $n = 4$. $ROTATE_4$ to generate 4 permutations and call previous $PERM_3$ using each of these permutations: 1234, 4123, 3412, and 2341 and then call PERM3 on each of these. It will generate all the permutations using the first three elements of each; that's 6 for each of these 4 primer permutations for 24 total, the right number: $4! = 24$. Notice the action of PERM2 is transparent: the first two elements are swapped. The rotation ensures no repeats; the starting elements are different.

The forthcoming code makes it all the more believable.

Programs

VAR NAME: PERM2

```
001  L1→L2
002  Disp L1
003  L1(1)→A
004  L1(2)→B
005  A→L1(2):B→L1(1)
006  Disp L1
007  L2→L1
```

Figure 2: The pattern: rotate and call n-1 program.

Figure 2 shows PERM1 combined with PERM2. Given any elements in L_1 the first two will be swapped and displayed. We prefer to say that the given contents of L_1, the first two items will be rotated. In TI84 Basic all variables are global, so we have to push (store, line 001) and pop (retrieve, line 007) contents.

Let's keep going. Figure 3 may look more complicated, but it really is just another rotation only know with three elements. These are stored and with each of the two calls generated by the for loop another rotation occurs: 123 begets 312 begets 231. The next rotation returns to 123. Or course sometimes soon this *not at all in place* rotating is going to eat up computer

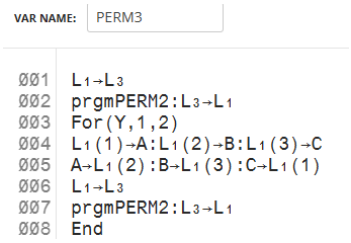

Figure 3: The three elements of L_1 are rotated twice.

memory and this brute force style is going to need an upgrade, but, don't miss the main point, it works. Figure 4 specifies the list to use and Figure 5 gives the printout.

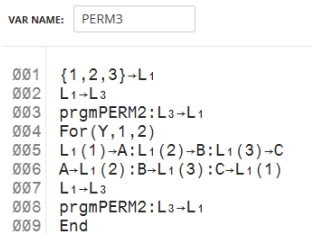

Figure 4: PERM3 filled in with a specific list, L_1.

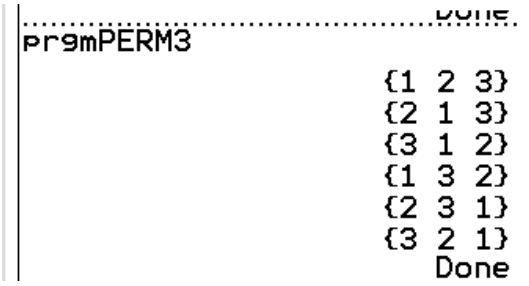

Figure 5: All permutations of $1, 2, 3$ are generated.

Figure 6 shows how the code needed to rotate four elements keeps growing in number of lines and variables. The number of permutations, $4! = 24$ can be confirmed by putting a counter in PERM2 every time this program uses *Disp*. We give code for the five element case, Figure 7. This will generate 120 permutations.

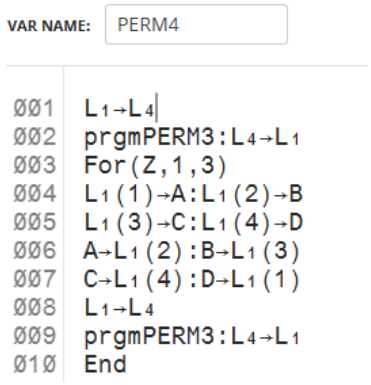

Figure 6: The store with separate variables is costly.

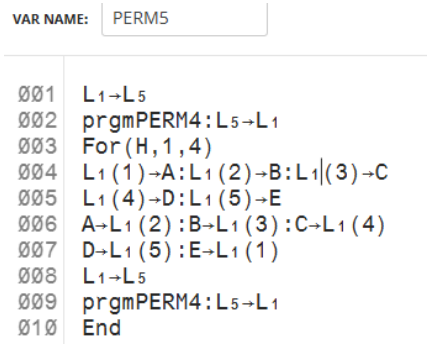

Figure 7: How to generate the rotations more elegantly?

Figure 8 gives the first pass at a generic rotator. The code in Figure 9 generates the 5 seed permutations shown in Figure 10

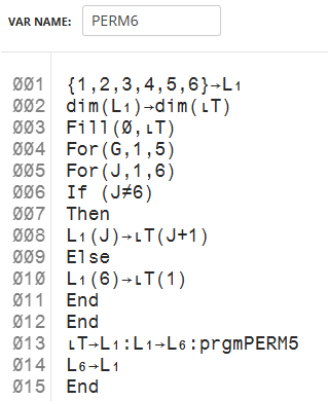

Figure 8: A generic rotation can be parsed out from this case.

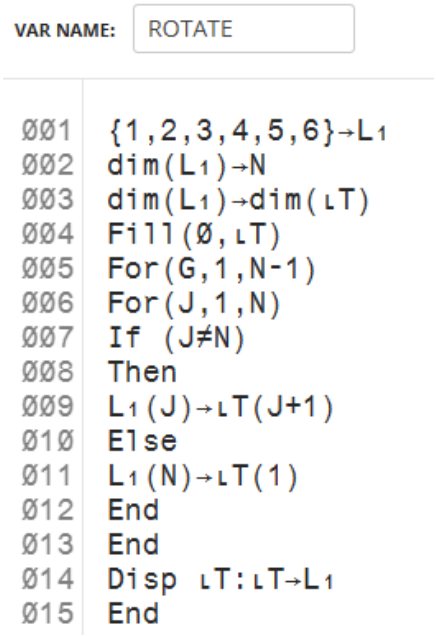

Figure 9: This code can take a general list and generate the seed permutations.

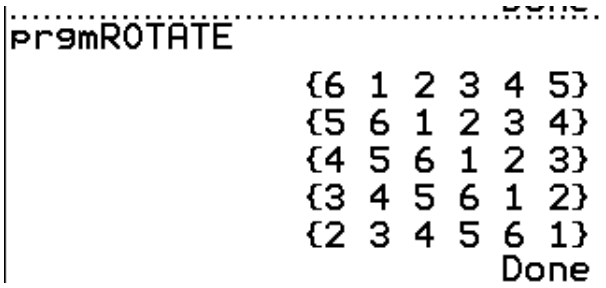

Figure 10: The seed permutations needed for PERM6.

Conclusion

Heap's algorithm gives a clever *in place* way of rotating elements of an array.

References

[1] T. H. Cormen and C. E. Leiserson and R. L. Rivest and C. Stein (2002), *Introduction to Algorithms*, 2nd ed., Cambridge, MA: MIT Press.

[2] B. R. Heap (1963), Permutations by interchanges. *The Computer Journal*, 6(3): 293 – 4.

[3] I. N. Herstein (1975), *Topics in Algebra*, 2nd ed., New York: Wiley.

[4] J. Kim (2025), retrieved at `https://youtu.be/xghJNlMibX4?si=9z2yYeTSyGkIVEV-`.

[5] R L. Kruse and B.P. Leung and C. L. Tondo (1991), *Data Structures and Program Design in C*, Englewood Cliffs, NJ: Prentice-Hall.