GOAP, Part 1

I was recently reviewing some problems on LeetCode. One of the weekly challenge problems appeared to require computing ***each and every permutation*** of a given string. This reminded me of the name of an algorithm to do just that – but this time, before I searched the name on DuckDuckGo and spoiled the answer, I figured I should try to derive it by myself. I recall trying to do just that many years ago and failing – so I figured, let me try to unpack this thing once and for all, in the form of a blog post. After all, these blog posts tend to be my way of unscrambling the universe.

As is clear to any sufficiently experienced stats student, given a string (or array) of size n, there will be n! permutations, because “(x_0, x_1, x_2)” is counted separately from “(x_1, x_0, x_2),” but you cannot “reuse” any element x_i. So there are `n` choices for the first element, times (n-1) choices for the element in the second position, time (n-2) choices for the third position, times (n-3)… all the way down to 1 remaining choice for the element in the nth position. As is clear to any sufficiently experienced computer science student, this **number of permutations** is simple to compute using recursion (done in O(n) time).

But a program to print each and every one of these permutations, without any omissions or repeated entries? Now we’re getting into interesting territory.

There is, in fact, a recursive solution, and it is probably what the LeetCode problem was calling for. If you’re not aware of it, I will not spoil it, but I will give you the hints that I used: (1) you do not need to start with a permutation and perform “swaps” on it, and (2) feel free to exploit the time-memory tradeoff. You can make a faster algorithm if you aren’t afraid to use some memory.

But what if we **did** insist upon starting with a permutation and performing swaps on it? Is there a way to get an efficient algorithm by doing that, without storing anything in memory?

I believe there is. But it is not so “obvious” to me, that’s for sure. My goal for this lesson is to show that it is a difficult problem. So let us consider some examples where n is small (starting with n=2) and work our way upward, shall we?

There are only (2!) = 2 permutations, namely:

- a, b
- b, a

That seems pretty simple, eh? You just swap ‘em! So you could do something simple like:

```
// initialize `nums` to a lexicographically-sorted array
// then use this base-case code:
if (2 == n) {
    // swap the "outer" pair of elements in the permutation
    int swap_placeholder = permutation[size-1];
    permutation[size-1] = permutation[start_ind];
    permutation[start_ind] = swap_placeholder;

    // print this instance
    std::cout << "{";
    for (size_t full_index=0; full_index < permutation.size()-1; full_index ++) {
```

```
            std::cout << permutation[full_index];
            std::cout << ", ";
        }
        std::cout << permutation[permutation.size() - 1] << "}" << std::endl;
    }
```

and you’re done!

Now let’s consider n=3.

- a, b, c
- a, c, b
- b, a, c
- b, c, a
- c, a, b
- c, b, a

Now, if we delegate to the n=2 case, we clearly need to run that exactly three times: once when a is in the first position, once when b is in the first position, and once when c is in the first position.

The question becomes, how do we “glue” these three iterations together?

It helps to begin the list of permutations with the simple, lexicographically sorted example; but the rest of the entries don’t need to be sorted, and in fact, to come up with the simplest, most scalable recursive algorithm, we probably shouldn’t do this.

Why do I make this remark? In the order I have presented the size-3 permutations above, “a, c, b” is followed by “b, a, c.” Well, guess what. If we **mandated** lexicographic ordering, then we’d have to do more than one “swap of consecutive elements” (i.e., a transposition) to go from the previous permutation to the next one.

This sounds, to me, like a really bad idea. If we’re going with a recursive approach, let’s make it as simple as possible to get from one permutation to the next (without visiting the same permutation multiple times).

When phrased this way – “*without visiting the same permutation twice*” – the problem is just begging for us to analyze it through the lens of graph theory. If we consider each permutation as a node in the graph and each transposition as an EDGE between two nodes, then we are looking for some kind of path that visits all nodes once and only once, i.e., a Hamiltonian Path!

> *A graph that possesses a Hamiltonian path is called a traceable graph.*
>
> *In general, the problem of finding a Hamiltonian path is NP-complete (Garey and Johnson 1983, pp. 199-200), so the only known way to determine whether a given general graph has a Hamiltonian path is to undertake an exhaustive search.*
>
> *Any bipartite graph with a vertex parity unbalance > 1 has no Hamiltonian paths.*
>
> *– Wolfram MathWorld*

The only regrettable fact is that this graph, that includes *all permutations in one*, cannot be called a "permutation graph," because evidently, that term is reserved for a different thing – [one graph assigned to one permutation](). Instead, I will call it the "graph of all permutations of size n," or GOAP(n). If you know a better name for this concept, please tell me. Surely I am not the first to visualize it this way!

Getting back on track, is there a Hamiltonian Path in the graph containing six permutations? Can we re-arrange the size-3 permutations so that any two consecutive permutations are only one swap away?

- a, b, c (even permutation, sgn = 1)
- a, c, b (odd)
- c, a, b (even permutation, sgn = 1)
- c, b, a (odd)
- b, c, a (even permutation, sgn = 1)
- b, a, c (odd)

Sure enough, we can! And now that we've done it this way, something becomes obvious: to get from entry 0 to entry 1, we swap the final pair of two elements; to get from entry 1 to entry 2, swap the penultimate pair; to get from entry 2 to entry 3, swap the new final pair; to get from entry 3 to entry 4, swap the penultimate pair; from entry 4 to entry 5, swap the final pair one last time.

Five total swaps, but you can clearly alternate between swapping the ultimate pair and swapping the penultimate pair. A clear separation between the base case and the recursive step. Sounds beautiful, doesn't it?

So, let's update our algorithm-in-progress with this notion:

```
void print_permutations(std::vector<int> &permutation, int start_ind, int size) {
  // int size = permutation.size();
  if (2 == size - start_ind) { // base case
    // swap the "outer" pair of elements in the permutation
    int swap_placeholder = permutation[size-1];
    permutation[size-1] = permutation[start_ind];
    permutation[start_ind] = swap_placeholder;

    // print this instance
    std::cout << "{";
    for (size_t full_index=0; full_index < permutation.size()-1; full_index ++) {
      std::cout << permutation[full_index];
      std::cout << ", ";
    }
    std::cout << permutation[permutation.size() - 1] << "}" << std::endl;
  } else if (2 < size - start_ind) { // recursive step
    for (int mandatory_cycles = 0; mandatory_cycles < size; mandatory_cycles ++) {

      // step 1. run the recursive step
      print_permutations(permutation, start_ind+1, size);

      // step 2. swap this "inner pair" of elements
      int swap_placeholder = permutation[size-1];
      permutation[size-1] = permutation[start_ind];
      permutation[start_ind] = swap_placeholder;

      // step 3. print 'em all again
      std::cout << "{";
      for (size_t full_index=0; full_index < permutation.size()-1; full_index ++) {
        std::cout << permutation[full_index];
        std::cout << ", ";
      }
```

```
      std::cout << permutation[permutation.size() - 1] << "}" << std::endl;
    }
  }
```

You'll notice, if you've taken a course in some of the best math ever, that the n=3 case is where things are already starting to get interesting: we have a nontrivial subgroup of the permutation group.

Let's see how things change when we kick it up to n=4, shall we? We're looking for 4! = 24 distinct permutations. Let us attempt the same general algorithm, applying transpositions recursively, beginning with the ordinary lexicographic "a, b, c, d."

1. a, b, c, d
2. a, b, d, c
3. a, d, b, c
4. a, d, c, b
5. a, c, d, b
6. a, c, b, d
7. c, a, b, d
8. c, a, d, b
9. c, d, a, b
10. c, d, b, a
11. c, b, d, a
12. c, b, a, d
13. b, c, a, d
14. b, c, d, a
15. ***b, d, c, a → missed detour through dbca and all sequences beginning with 'd'***
16. b, d, a, c
17. b, a, d, c
18. b, a, c, d

It is at this point that we face an inconvenient truth. Our method from before breaks in the n=4 case, because now, if we continue the pattern and swap the first pair of elements, we end up at "a, b, c, d" – the same permutation we started with – but we've only seen eighteen of the 24 distinct permutations! Specifically, we've completely neglected the permutations beginning with `d`! There is no Hamiltonian Cycle.

It is at this point that Younger Me threw up his hands and gave up.

But with this new framing of graph theory, I can at least try to continue. What does the graph look like when n = 4?

1. a, b, c, d
2. a, b, d, c
3. a, d, b, c → d, a, b, c
4. a, d, c, b → d, a, c, b
5. a, c, d, b
6. a, c, b, d
7. c, a, b, d
8. c, a, d, b
9. c, d, a, b → d, c, a, b

10. c, d, b, a → d, c, b, a
11. c, b, d, a
12. c, b, a, d
13. b, c, a, d
14. b, c, d, a
15. b, d, c, a → d, b, c, a
16. b, d, a, c → d, b, a, c
17. b, a, d, c
18. b, a, c, d

Dirac's Theorem tells us that "any simple graph with n > 2 vertices in which each graph vertex has vertex degree > n/2 has a Hamiltonian cycle."

Do you think this will come in handy?

The graph is simple.
It has n! > 2 vertices as long as n > 2.
Does each vertex have degree > n/2 ?
Well, since any permutation can be the object of (n-1) different transpositions (seriously. Think about it. How many consecutive pairs exist in any single permutation?), each vertex has degree equal to (n-1).
This expression *is* greater than n/2 as long as n is greater than 2.
So it does apply! These graphs do have Hamiltonian cycle. But what does that cycle look like…?

Well, if the proof of Dirac's Theorem is constructive, then perhaps we can use that!

But before we dive into that: let's try one last-ditch effort. An internet friend claimed that my restriction to *transpositions between adjacent elements in a given permutation* (rather than allowing transpositions between non-consecutive elements) made the problem "significantly harder."

Is this true?

I decided to try my previous approach, a recursive algorithm, permitting all transpositions, even those between non-consecutive elements. Here's what I came up with:

Select (a, b, c, d) as the starting node without loss of generality.

exchange indices 2 and 3 to get (a, b, d, c)
exchange indices 1 and 3 to get (a, c, d, b)
exchange indices 2 and 3 to get (a, c, b, d)
exchange indices 1 and 3 to get (a, d, b, c)
exchange indices 2 and 3 to get (a, d, c, b)
exchange indices 0 and 3 to get (b, d, c, a)
exchange indices 2 and 3 to get (b, d, a, c)
exchange indices 1 and 3 to get (b, c, a, d)
exchange indices 2 and 3 to get (b, c, d, a)
exchange indices 1 and 3 to get (b, a, d, c)
exchange indices 2 and 3 to get (b, a, c, d)
exchange indices 0 and 3 to get (d, a, c, b)
exchange indices 2 and 3 to get (d, a, b, c)

exchange indices 1 and 3 to get (d, c, b, a)
exchange indices 2 and 3 to get (d, c, a, b)
exchange indices 1 and 3 to get (d, b, a, c)
exchange indices 2 and 3 to get (d, b, c, a)

....and now we're back at square zero. If we "continue the pattern" and swap indices 0 and 3, then we'd end up with (a, b, c, d), the same node at which we started!

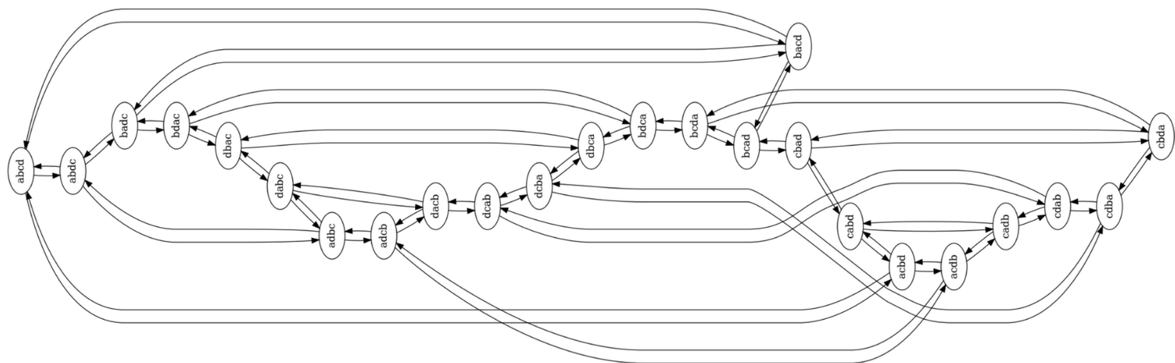
Notice that I tried to keep this algorithm as simple as possible, which is why the *final element* (whichever one is currently located at index 3) *always gets swapped*, but otherwise, I believe I have further proof that I'm not insane. This problem is counter-intuitive even ***if*** we broaden the constraints!

When confronted with this data, my friend essentially back-pedaled and said "I can be more precise […] I found this other version with no such requirement easier." Maybe I need to switch this up and swap a *different* element (not located at index 3) at one point, one particular "critical juncture." But when should we do this? It's not clear a priori. So in the meantime, let me consider the simpler problem that doesn't use all these extraneous transpositions.

I decided to write an algorithm in python to actually generate the visual graph of all 4-permutations. The result of that is given below in Figure 1.

The key point is, we want to find a Hamiltonian Cycle in this graph: a path that starts and ends at the same node and visits each nodes once and only once!

Here's a key insight: because any path will visit each node once and only once, it follow that for each



node in the graph, the path will use two of that node's adjacent edges and ***avoid*** *the third one.* So if we choose one starting node and select two of its edges, then we're identifying a whole family of one (or more) valid Hamiltonian cycles that include those two edges! Therefore, we may "cut" that node's third edge our of the graph, without disrupting the Hamiltonian cycle(s) we've chosen. For example, if we simply mandate that the Hamiltonian cycle (to which we will converge) uses the edges (b,a,c,d) → (b,c,a,d) and (b,c,a,d) → (c,b,a,d), then it follows that the cycle cannot contain (b,c,d,a) → (b,c,a,d). Therefore it must contain (c,d,b,a) → (b,c,d,a) and (b,c,d,a) → (b,d,c,a).

Using this strategy and some trial and error, I found the following cycle. This time, I underline consecutive elements that are being swapped in order to find the next element.

Counter	Permutation	Index of the Next Element to be Swapped
1	b,a,c,d	1
2	b,c,a,d	0
3	c,b,a,d	1
4	c,a,b,d	0
5	a,c,b,d	2
6	a,c,d,b	0
7	c,a,d,b	1
8	c,d,a,b	2
9	c,d,b,a	1
10	c,b,d,a	0
11	b,c,d,a	1
12	b,d,c,a	0
13	d,b,c,a	1
14	d,c,b,a	2
15	d,c,a,b	1
16	d,a,c,b	0
17	a,d,c,b	2
18	a,d,b,c	0
19	d,a,b,c	1
20	d,b,a,c	0
21	b,d,a,c	1
22	b,a,d,c	0
23	a,b,d,c	2
24	a,b,c,d	0

Obviously, we can confirm that this is a cycle because it visits all twenty-four nodes and wraps back to the start. But what's really weird about this is that, when we look at the third column, we can potentially discern the "instructions necessary" to generate all permutations and then find a general pattern. Notice that it has four general "sequences" with variations applied to each one.

Clearly, for n=4, we can divide our instructions into four segments: ||1, 0, 1, 0, 2, 0|| followed by ||1, 2, 1, 0, 1, 0|| followed by ||1, 2, 1, 0, 2, 0||, followed by ||1, 0, 1, 0, 2, 0||. You might be able to change the ordering in which you apply those four sequences, but I haven't confirmed it yet.

A majority (three) of the sequences end with a "0, 2, 0," except one time when we execute "0, 1, 0." Remarkably, this exceptional case isn't just a coincidence: it represents the fact that we cannot "backtrack" from (b,c,d,a) to (b,c,a,d), because this is the edge that we removed! As for the beginning of a sequence, we run "1, 2, 1" two times and "1, 0, 1" the other two times. This also codifies the fact that we cannot "backtrack to an already-visited node;" in fact, one of them prevents going from (b,c,a,d) to (b,c,d,a), which is another consequence of "cutting" that edge out of the picture.

But what does this mean for our Algorithm? You tell me.

One Last Excursion Into Graph Theory

Maybe this obsession over indices isn't getting us anywhere. Maybe we need to focus on basic graph algorithms.

Well, this section is designed to show that the typical graph algorithms we learn are not immediately helpful.

The two famous means of traversing a graph are the Breadth-First Search and the Depth-First Search. Recall that we are looking for a Hamiltonian Cycle, which when you think about it, is really a **subgraph** of our original graph.

A typical Breadth-First Search (BFS) would start with one node ("a,b,c,d" or perhaps "b,a,c,d" above) and analyze all of its edges. This doesn't seem helpful, because it is only **two** of these edges that will be selected for our subgraph.

A Depth-First Search (DFS) has an advantage in that it delves "deeper" into the original graph, leaving most edges open so that there will still be paths available leading back to the starting node. It also is pretty dang useful in finding cycles in a graph.

And whether we choose BFS or DFS, our original graph is still densely-connected, so we would perpetually run the risk of following a new edge and finding a node that we've already visited. This would (in most cases) be a "mistake" that would prompt us to do some **backtracking**. One way to catch this would be to add a parameter to the function for "number of nodes visited so far" and then only returning true (success) if and only if we've visited all nodes.

But here's the problem with that: the algorithm we're looking for traverses **all `n` nodes** without using any unnecessary edges. I'm inclined to think that *if you backtrack, you've already lost the game*[1].

So, what if, rather than starting at one node and inserting nodes to the subgraph, we instead started with the entire graph and started "cutting out" unwanted edges? This way, we could guarantee that all nodes are retained and have a clear victory condition: each node has two and only two edges connected to it in the subgraph.

But here's the catch: we can't do this "cut out unwanted edges" approach, because it presupposes that the data has been organized into a graph in the first place! We do not have access to the complete graph from the start.

1 Speaking of which, I lost the game.