Next lexicographical permutation algorithm

Introduction

Suppose we have a finite sequence of numbers like (0, 3, 3, 5, 8), and want to generate all its permutations. What is the best way to do so?

The naive way would be to take a top-down, recursive approach. We could pick the first element, then recurse and pick the second element from the remaining ones, and so on. But this method is tricky because it involves recursion, stack storage, and skipping over duplicate values. Moreover, if we insist on manipulating the sequence in place (without producing temporary arrays), then it's difficult to generate the permutations in lexicographical order.

It turns out that the best approach to generating all the permutations is to start at the lowest permutation, and repeatedly compute the next permutation in place. The simple and fast algorithm for performing this is what will be described on this page. We will use concrete examples to illustrate the reasoning behind each step of the algorithm.

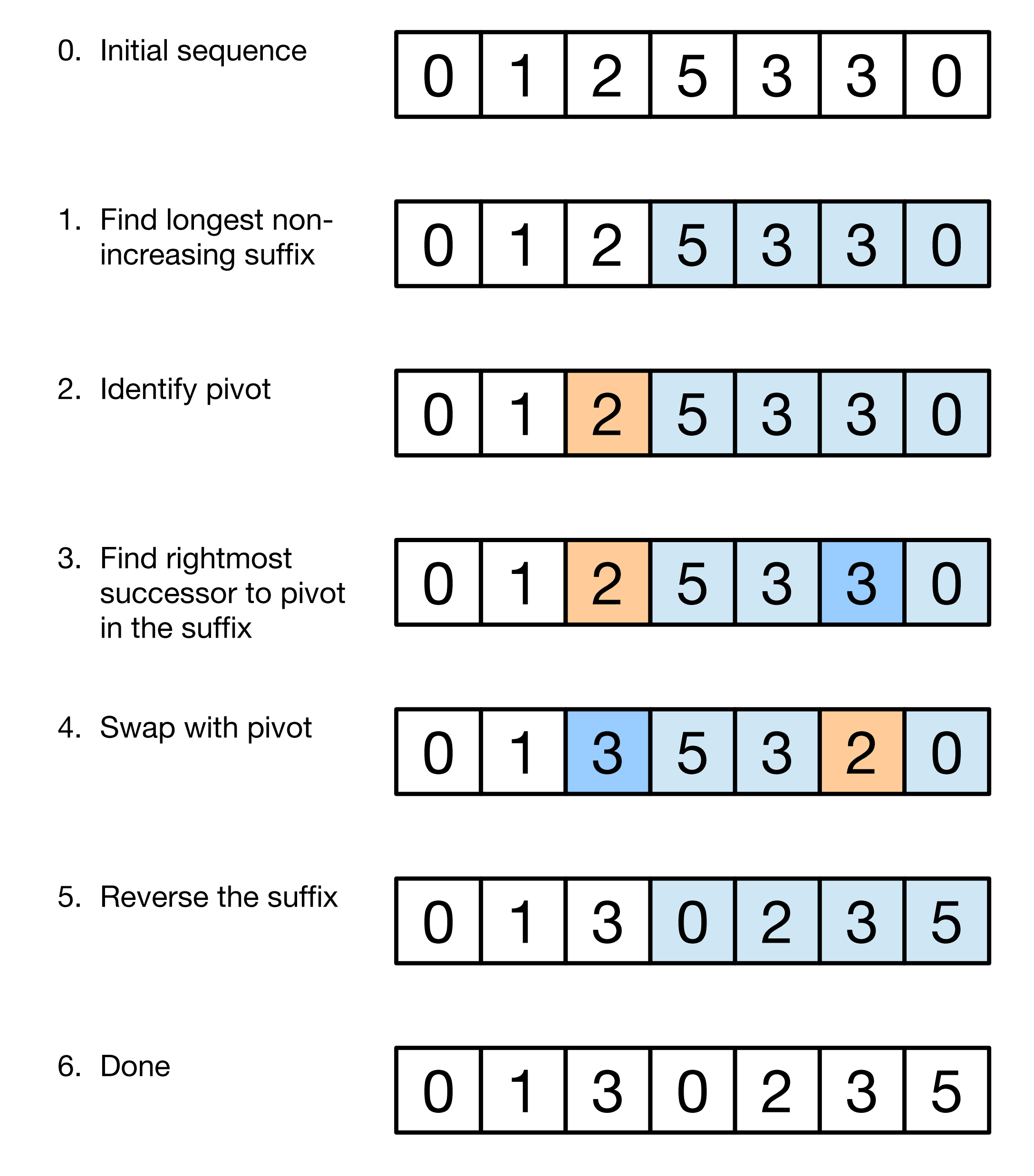
The algorithm

We will use the sequence (0, 1, 2, 5, 3, 3, 0) as a running example.

The key observation in this algorithm is that when we want to compute the next permutation, we must "increase" the sequence *as little as possible*. Just like when we count up using numbers, we try to modify the rightmost elements and leave the left side unchanged. For example, there is no need to change the first element from 0 to 1, because by changing the prefix from (0, 1) to (0, 2) we get an even closer next permutation. In fact, there is no need to change the second element either, which brings us to the next point.

Firstly, identify the longest suffix that is non-increasing (i.e. weakly decreasing). In our example, the suffix with this property is (5, 3, 3, 0). This suffix is already the highest permutation, so we can't make a next permutation just by modifying it – we need to modify some element(s) to the left of it. (Note that we can identify this suffix in $O(n)$ time by scanning the sequence from right to left. Also note that such a suffix has at least one element, because a single element substring is trivially non-increasing.)

Secondly, look at the element immediately to the left of the suffix (in the example it's 2) and call it the pivot. (If there is no such element – i.e. the entire sequence is non-increasing – then this is already the last permutation.) The pivot is necessarily less than the head of the suffix (in the example it's 5). So some element in the suffix is greater than the pivot. If we swap the pivot with the smallest element in the suffix that is greater than the pivot, then the prefix is minimized. (The prefix is everything in the sequence except the suffix.) In the example, we end up with the new prefix (0, 1, 3) and new suffix (5, 3, 2, 0). (Note that if the suffix has multiple copies of the new pivot, we should take the rightmost copy – this plays into the next step.)



Finally, we sort the suffix in non-decreasing (i.e. weakly increasing) order because we increased the prefix, so we want to make the new suffix as low as possible. In fact, we can avoid sorting and simply reverse the suffix, because the replaced element respects the weakly decreasing order. Thus we obtain the sequence (0, 1, 3, 0, 2, 3, 5), which is the next permutation that we wanted to compute.

Condensed mathematical description:

1. Find largest index *i* such that *array*[$i - 1$] < *array*[*i*].
   (If no such *i* exists, then this is already the last permutation.)

2. Find largest index *j* such that j ≥ i and *array*[*j*] > *array*[$i - 1$].

3. Swap *array*[*j*] and *array*[$i - 1$].

4. Reverse the suffix starting at *array*[*i*].

Now if you truly understand the algorithm, here's an extension exercise for you: Design the algorithm for stepping backward to the *previous* lexicographical permutation. (Spoilers at the bottom.)

Annotated code (Java)

```java
boolean nextPermutation(int[] array) {
    // Find longest non-increasing suffix
    int i = array.length - 1;
    while (i > 0 && array[i - 1] >= array[i])
        i--;
    // Now i is the head index of the suffix

    // Are we at the last permutation already?
    if (i <= 0)
        return false;

    // Let array[i - 1] be the pivot
    // Find rightmost element that exceeds the pivot
    int j = array.length - 1;
    while (array[j] <= array[i - 1])
        j--;
    // Now the value array[j] will become the new pivot
    // Assertion: j >= i

    // Swap the pivot with j
    int temp = array[i - 1];
    array[i - 1] = array[j];
    array[j] = temp;

    // Reverse the suffix
    j = array.length - 1;
    while (i < j) {
        temp = array[i];
```

```
            array[i] = array[j];
            array[j] = temp;
            i++;
            j--;
        }

        // Successfully computed the next permutation
        return true;
    }
```

This code can be mechanically translated to a programming language of your choice, with minimal understanding of the algorithm. (Note that in Java, arrays are indexed from 0.)

Example usages

Print all the permutations of (0, 1, 1, 1, 4):

```
int[] array = {0, 1, 1, 1, 4};
do {  // Must start at lowest permutation
    System.out.println(Arrays.toString(array));
} while (nextPermutation(array));
```

Project Euler #24: Find the millionth (1-based) permutation of (0, 1, 2, 3, 4, 5, 6, 7, 8, 9). My Java solution: p024.java

Project Euler #41: Find the largest prime number whose base-10 digits are a permutation of (1, 2, 3, 4, 5, 6, 7, 8, 9). My Java solution: p041.java

Source code

- Python: nextperm.py
- JavaScript: nextperm.js
- TypeScript: nextperm.ts
- Java: nextperm.java
- C#: nextperm.cs
- C++: nextperm.cpp
- C: nextperm.c
- Rust: nextperm.rs