Next lexicographical permutation algorithm

Introduction

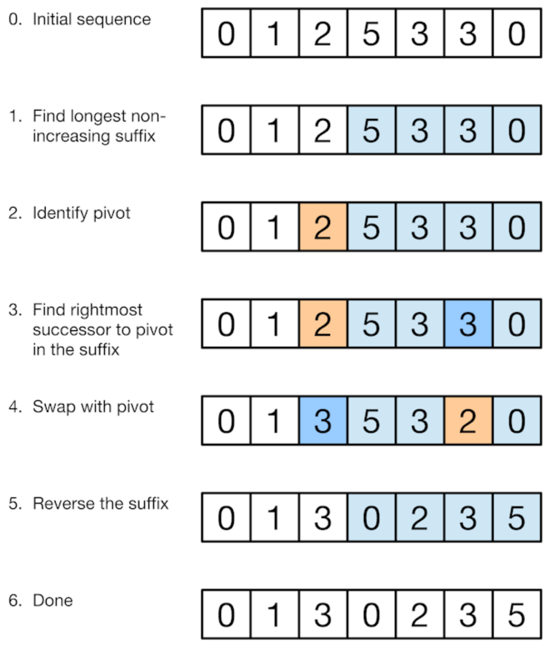
Suppose we have a sequence of numbers (e.g. (0, 3, 3, 5, 8)) and want to generate all its permutations. What's the best way to do this?

We could take a top-down approach by picking the first element, then recursing and picking the second element and so on. But this is tricky because it involves recursion, stack storage, and skiping over duplicate values. Moreover, if we insist on manipulating the sequence in place (no temporary new sequences), then it's difficult to generate the permutations in lexicographical order.

As a matter of fact, 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 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 alone. 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.

First, 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 1 element, because a single element substring is trivially non-increasing.)

Second, 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-decreasing – 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. So finally, we get the sequence (0, 1, 3, 0, 2, 3, 5), which is the next permutation that we wanted.

Condensed mathematical description:

1. Find largest index i such that $array[i - 1] < array[i]$.
2. Find largest index j such that $j \geq i$ 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 going backward to the *previous* lexicographical permutation.

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};  // Must start at lowest permutation
do {
    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

- Java: nextperm.java
- C: nextperm.c
- C++: nextperm.cpp
- Python: nextperm.py
- JavaScript: nextperm.js
- Mathematica: nextperm.mat.txt
- Haskell: nextperm.hs (probably quite suboptimal)

Code previews

Java

```
/**
 * Computes the next lexicographical permutation of the specified array of integers in place,
 * returning whether a next permutation existed. (Returns {@code false} when the argument
 * is already the last possible permutation.)
 * @param array the array of integers to permute
 * @return whether the array was permuted to the next permutation
 */
public static boolean nextPermutation(int[] array) {
    // Find non-increasing suffix
    int i = array.length - 1;
    while (i > 0 && array[i - 1] >= array[i])
        i--;
    if (i <= 0)
        return false;

    // Find successor to pivot
    int j = array.length - 1;
    while (array[j] <= array[i - 1])
        j--;
    int temp = array[i - 1];
    array[i - 1] = array[j];
    array[j] = temp;

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

C

```
/*
 * Computes the next lexicographical permutation of the specified array of integers in place,
 * returning a Boolean to indicate whether a next permutation existed.
 * (Returns false when the argument is already the last possible permutation.)
 */
int next_permutation(int *array, int length) {
    int i, j;
    int temp;

    // Find non-increasing suffix
    i = length - 1;
    while (i > 0 && array[i - 1] >= array[i])
        i--;
    if (i <= 0)
        return 0;

    // Find successor to pivot
    j = length - 1;
    while (array[j] <= array[i - 1])
        j--;
    temp = array[i - 1];
    array[i - 1] = array[j];
    array[j] = temp;

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

C++

```
/*
 * Computes the next lexicographical permutation of the specified array in place,
 * returning whether a next permutation existed. (Returns false when the argument
 * is already the last possible permutation.)
 */
template <typename T>
bool next_permutation(T *array, int length) {
    // Find non-increasing suffix
    int i = length - 1;
    while (i > 0 && array[i - 1] >= array[i])
        i--;
    if (i <= 0)
        return false;

    // Find successor to pivot
    int j = length - 1;
    while (array[j] <= array[i - 1])
        j--;
    T temp = array[i - 1];
    array[i - 1] = array[j];
    array[j] = temp;

    // Reverse suffix
    j = length - 1;
    while (i < j) {
        temp = array[i];
        array[i] = array[j];
        array[j] = temp;
        i++;
        j--;
    }
    return true;
}
```

Python

```
#
# Computes the next lexicographical permutation of the specified list in place,
# returning whether a next permutation existed. (Returns False when the argument
# is already the last possible permutation.)
#
def next_permutation(arr):
    # Find non-increasing suffix
    i = len(arr) - 1
    while i > 0 and arr[i - 1] >= arr[i]:
        i -= 1
    if i <= 0:
        return False

    # Find successor to pivot
    j = len(arr) - 1
    while arr[j] <= arr[i - 1]:
        j -= 1
    arr[i - 1], arr[j] = arr[j], arr[i - 1]

    # Reverse suffix
    arr[i : ] = arr[len(arr) - 1 : i - 1 : -1]
    return True

# Example:
#   arr = [0, 1, 0]
#   next_permutation(arr)  (returns True)
#   arr has been modified to be [1, 0, 0]
```

JavaScript

```
/*
 * Computes the next lexicographical permutation of the specified array of numbers in place,
 * returning whether a next permutation existed. (Returns false when the argument
 * is already the last possible permutation.)
 */
function nextPermutation(array) {
    // Find non-increasing suffix
    var i = array.length - 1;
    while (i > 0 && array[i - 1] >= array[i])
        i--;
    if (i <= 0)
        return false;

    // Find successor to pivot
    var j = array.length - 1;
    while (array[j] <= array[i - 1])
        j--;
    var temp = array[i - 1];
    array[i - 1] = array[j];
    array[j] = temp;

    // Reverse suffix
    j = array.length - 1;
    while (i < j) {
        temp = array[i];
        array[i] = array[j];
        array[j] = temp;
        i++;
        j--;
    }
    return true;
}

// Example:
//   arr = [0, 1, 0];
//   nextPermutation(arr);  (returns true)
//   arr has been modified to be [1, 0, 0]
```

Mathematica

```
(*
 * Computes the next lexicographical permutation of the specified vector of numbers.
 * Returns the pair {Boolean, permuted vector}, where the Boolean value indicates
 * whether a next permutation existed or not.
 *)
NextPermutation[arr_] := Module[{i, j},
  (* Find non-increasing suffix *)
  For[i = Length[arr], i > 1 && arr[[i - 1]] >= arr[[i]], i--];
  If[i <= 1,
    Return[{False, arr}]];
  (* Find successor to pivot *)
  For[j = Length[arr], arr[[j]] <= arr[[i - 1]], j--];
  (* Return new list with indexes i and j swapped, followed by the suffix reversed *)
  {True, Join[Take[arr, i - 2], {arr[[j]]}, Reverse[Drop[ReplacePart[arr, arr[[i - 1]], j], i - 1]]]}]

(* Example: NextPermutation[{0, 1, 0}] -> {True, {1, 0, 0}} *)
```

Haskell

```
{-
 - Computes the next lexicographical permutation of the specified finite list of numbers.
 - Returns the pair {status, permuted list}, where the Boolean value indicates
 - whether a next permutation existed or not.
 -}
nextPermutation :: [Integer] -> (Bool, [Integer])
nextPermutation [] = (False, [])
nextPermutation xs =
    let suffix = fst (findSuffix xs)  -- Longest non-decreasing suffix
        suflen = length suffix
        len = length xs
        prefixMinusPivot = take (len - suflen - 1) xs
    in if suflen == len then (False, xs) else
        let pivotIndex = len - suflen - 1
            pivot = xs !! pivotIndex
            -- Index of rightmost element in suffix greater than pivot
            newIndex = (length (takeWhile (> pivot) suffix)) - 1
            newSuffix = reverse $ (take newIndex suffix) ++ (pivot : (drop (newIndex + 1) suffix))
        in (True, prefixMinusPivot ++ ((suffix !! newIndex) : newSuffix))
    where
        findSuffix [x] = ([x], True)
        findSuffix (x:xs) =
            let (suf, cont) = findSuffix xs
            in if cont && x >= (head suf) then (x:suf, True) else (suf, False)

-- Example: nextPermutation [0, 1, 0] -> (True, [1, 0, 0])
```

More info

- Wikipedia: Permutation - Generation in lexicographic order

Browse Project Nayuki

- Fast SHA-1 hash implementation in x86 assembly
- Sinc-based image resampler
- Propositional sequent calculus prover
- About
- Transcription of Kana's Theme

Categories: Java, JavaScript, Python, Programming

Last updated: 2014-01-23

Feedback

Question? Comment? Contact me

ProjectNayuki: Like, comment, follow updates on Facebook