Next lexicographical permutation algorithm

Introduction

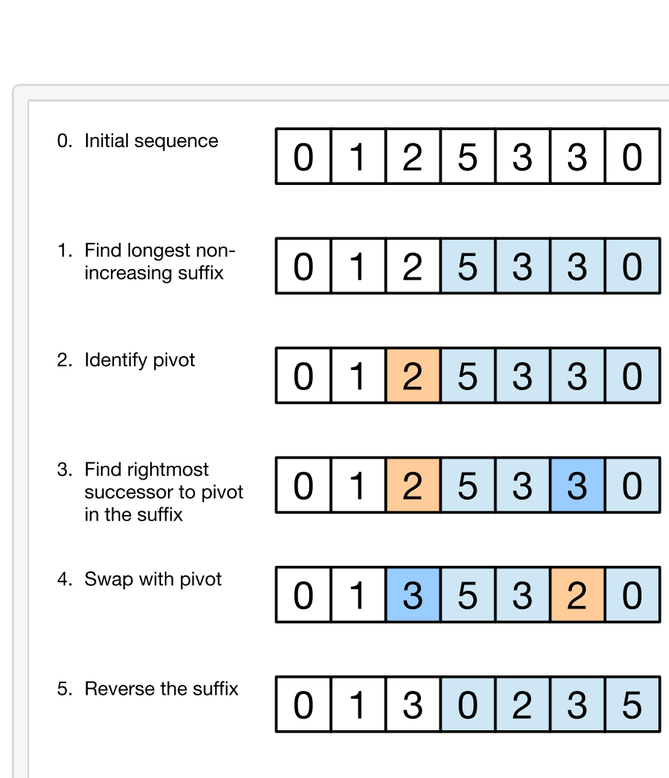
Suppose we have a finite sequence of numbers like (0, 3, 3, 5, 8), and want to generate all its permutations. What is the best way to do so?

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 $\Theta(n)$ 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 minimally increased. (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 $array[i]$.

Overall, this algorithm to compute the next lexicographical permutation has $\Theta(n)$ worst-case time complexity, and $\Theta(1)$ space complexity. Thus, computing every permutation requires $\Theta(n! \times n)$ run time.

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 greater than 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):

```java
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

LeetCode #31: Next Permutation.

Source code

- Python: nextperm.py
- JavaScript: nextperm.js
- TypeScript: nextperm.ts
- Java: nextperm.java
- C#: nextperm.cs
- C++: nextperm.hpp
- C: nextperm.c
- Rust: nextperm.rs
- Haskell: nextperm.hs (probably suboptimal)
- Mathematica: nextperm.mat.txt
- MATLAB: nextperm.m

Code previews

Python

```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

```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]
```

Java

```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#

```csharp
/*
 * Computes the next lexicographical permutation of the given array
 * of integers in place, returning whether a next permutation existed.
 * (Returns false when the argument is already the last possible permutation.)
 */
public static bool 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++

```cpp
#include <algorithm>
#include <vector>

/*
 * Computes the next lexicographical permutation of the specified vector
 * of values in place, returning whether a next permutation existed.
 * (Returns false when the argument is already the last possible permutation.)
 */
template <typename T>
bool nextPermutation(std::vector<T> &vec) {
    // Find non-increasing suffix
    if (vec.size() == 0)
        return false;
    typename std::vector<T>::iterator i = vec.end() - 1;
    while (i > vec.begin() && *(i - 1) >= *i)
        --i;
    if (i == vec.begin())
        return false;
    
    // Find successor to pivot
    typename std::vector<T>::iterator j = vec.end() - 1;
    while (*j <= *(i - 1))
        --j;
    std::iter_swap(i - 1, j);
    
    // Reverse suffix
    std::reverse(i, vec.end());
    return true;
}
```

C

```c
#include <stdbool.h>
#include <stddef.h>

/*
 * 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.)
 */
bool next_permutation(int array[], size_t length) {
    // Find non-increasing suffix
    if (length == 0)
        return false;
    size_t i = length - 1;
    while (i > 0 && array[i - 1] >= array[i])
        i--;
    if (i == 0)
        return false;
    
    // Find successor to pivot
    size_t j = 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 = length - 1;
    while (i < j) {
        temp = array[i];
        array[i] = array[j];
        array[j] = temp;
        i++;
        j--;
    }
    return true;
}
```

Rust

```rust
fn next_permutation<T: std::cmp::Ord>(array: &mut [T]) -> bool {
    // Find non-increasing suffix
    if array.len() == 0 {
        return false;
    }
    let mut i: usize = array.len() - 1;
    while i > 0 && array[i - 1] >= array[i] {
        i -= 1;
    }
    if i == 0 {
        return false;
    }
    
    // Find successor to pivot
    let mut j: usize = array.len() - 1;
    while array[j] <= array[i - 1] {
        j -= 1;
    }
    array.swap(i - 1, j);
    
    // Reverse suffix
    array[i .. ].reverse();
    true
}
```

Haskell

```haskell
{-
 - Computes the next lexicographical permutation of the specified
 - finite list of numbers. Returns Nothing if the argument is
 - already the highest permutation.
 -}
nextPermutation :: Ord a => [a] -> Maybe [a]
nextPermutation xs =
    let
        len = length xs
        -- Reverse of longest non-increasing suffix
        revSuffix = findPrefix (reverse xs)
        suffixLen = length revSuffix
        prefixMinusPivot = take (len - suffixLen - 1) xs
        pivot = xs !! (len - suffixLen - 1)
        suffixHead = takeWhile (<= pivot) revSuffix
        newPivot : suffixTail = drop (length suffixHead) revSuffix
        newSuffix = suffixHead ++ (pivot : suffixTail)
    in
        if suffixLen == len then Nothing else
            Just (prefixMinusPivot ++ (newPivot : newSuffix))
    where
        findPrefix [] = []
        findPrefix (x:xs) = x : (if xs /= [] && x <= (head xs)
            then (findPrefix xs) else [])

-- Example: nextPermutation [0, 1, 0] -> Just [1, 0, 0]
```

Mathematica

```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}} *)
```

MATLAB

```matlab
function result = nextperm(arr)
% Computes and returns the next lexicographical permutation of the given vector,
% or returns [] when the argument is already the last possible permutation.
% Example: nextperm([0, 1, 0]) -> [1, 0, 0]

    % Find non-increasing suffix
    i = length(arr);
    while i > 1 && arr(i - 1) >= arr(i)
        i = i - 1;
    end
    if i <= 1
        result = [];
        return;
    end

    % Find successor to pivot
    result = arr;
    j = length(result);
    while result(j) <= result(i - 1)
        j = j - 1;
    end
    temp = result(i - 1);
    result(i - 1) = result(j);
    result(j) = temp;

    % Reverse suffix
    result(i : end) = result(end : -1 : i);
end
```

Bonus: Previous permutation (Java)

```java
boolean previousPermutation(int[] array) {
    // Find longest non-decreasing 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 first permutation already?
    if (i <= 0)
        return false;
    
    // Let array[i - 1] be the pivot
    // Find rightmost element less than 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 previous permutation
    return true;
}
```

More info

- Wikipedia: Permutation - Generation in lexicographic order
- LeetCode Articles: Next Permutation

Categories: Programming, Java, JavaScript / TypeScript, Python, C++, C

Last updated: 2018-06-20

Browse Project Nayuki

- Fast Fourier transform in x86 assembly
- Forcing a file's CRC to any value
- Zeller's congruence
- The photographic exposure equation
- Extending the use of logarithmic scales

Recent

- PNG library
- About
- Unspecified edge cases in the DEFLATE standard
- DEFLATE specification v1.3 (HTML)
- Overview of Project Nayuki software licenses
- (more)

Random

- Reference arithmetic coding
- Barrett reduction algorithm
- 1D barcode generator (JavaScript)
- (more)

RSS feed

Subscribe for updates