Longest Common Subsequence

The Longest Common Subsequence (LCS) of two strings is the longest sequence of characters that appear in the same order in both strings.

For example the LCS of `"Hello World"` and `"Bonjour le monde"` is `"oorld"`. If you go through both strings from left-to-right, you'll find that the characters `o`, `o`, `r`, `l`, `d` appear in both strings in that order.

Other possible subsequences are `"ed"` and `"old"`, but these are all shorter than `"oorld"`.

> **Note:** This should not be confused with the Longest Common Substring problem, where the characters must form a substring of both strings, i.e they have to be immediate neighbors. With a subsequence, it's OK if the characters are not right next to each other, but they must be in the same order.

One way to find the LCS of two strings `a` and `b` is using dynamic programming and a backtracking strategy.

Finding the length of the LCS with dynamic programming

First, we want to find the length of the longest common subsequence between strings `a` and `b`. We're not looking for the actual subsequence yet, only how long it is.

To determine the length of the LCS between all combinations of substrings of `a` and `b`, we can use a *dynamic programming* technique. Dynamic programming basically means that you compute all possibilities and store them inside a look-up table.

> **Note:** During the following explanation, `n` is the length of string `a`, and `m` is the length of string `b`.

To find the lengths of all possible subsequences, we use a helper function, `lcsLength(_:)`. This creates a matrix of size `(n+1)` by `(m+1)`, where `matrix[x][y]` is the length of the LCS between the substrings `a[0...x-1]` and `b[0...y-1]`.

Given strings `"ABCBX"` and `"ABDCAB"`, the output matrix of `lcsLength(_:)` is the following:

```
|   | ∅ | A | B | D | C | A | B |
| ∅ | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| A | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| B | 0 | 1 | 2 | 2 | 2 | 2 | 2 |
| C | 0 | 1 | 2 | 2 | 3 | 3 | 3 |
| B | 0 | 1 | 2 | 2 | 3 | 3 | 4 |
| X | 0 | 1 | 2 | 2 | 3 | 3 | 4 |
```

In this example, if we look at `matrix[3][4]` we find the value `3`. This means the length of the LCS between `a[0...2]` and `b[0...3]`, or between `"ABC"` and `"ABDC"`, is 3. That is correct, because

these two substrings have the subsequence `ABC` in common. (Note: the first row and column of the matrix are always filled with zeros.)

Here is the source code for `lcsLength(_:)`; this lives in an extension on `String`:

```
func lcsLength(_ other: String) -> [[Int]] {

  var matrix = [[Int]](repeating: [Int](repeating: 0, count: other.characters.count-

  for (i, selfChar) in self.characters.enumerated() {
        for (j, otherChar) in other.characters.enumerated() {
          if otherChar == selfChar {
                // Common char found, add 1 to highest lcs found so far.
                matrix[i+1][j+1] = matrix[i][j] + 1
          } else {
                // Not a match, propagates highest lcs length found so far.
                matrix[i+1][j+1] = max(matrix[i][j+1], matrix[i+1][j])
          }
        }
  }

  return matrix
}
```

First, this creates a new matrix -- really a 2-dimensional array -- and fills it with zeros. Then it loops through both strings, `self` and `other`, and compares their characters in order to fill in the matrix. I two characters match, we increment the length of the subsequence. However, if two characters are different, then we "propagate" the highest LCS length found so far.

Let's say the following is the current situation:

```
|   | Ø | A | B | D | C | A | B |
| Ø | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| A | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| B | 0 | 1 | 2 | 2 | 2 | 2 | 2 |
| C | 0 | 1 | 2 | * |   |   |   |
| B | 0 |   |   |   |   |   |   |
| X | 0 |   |   |   |   |   |   |
```

The `*` marks the two characters we're currently comparing, `C` versus `D`. These characters are not the same, so we propagate the highest length we've seen so far, which is `2`:

```
|   | Ø | A | B | D | C | A | B |
| Ø | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| A | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| B | 0 | 1 | 2 | 2 | 2 | 2 | 2 |
```

```
| C | 0 | 1 | 2 | 2 | * |   |   |
| B | 0 |   |   |   |   |   |   |
| X | 0 |   |   |   |   |   |   |
```

Now we compare `C` with `C`. These are equal, and we increment the length to `3`:

```
|   | Ø | A | B | D | C | A | B |
| Ø | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| A | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| B | 0 | 1 | 2 | 2 | 2 | 2 | 2 |
| C | 0 | 1 | 2 | 2 | 3 | * |   |
| B | 0 |   |   |   |   |   |   |
| X | 0 |   |   |   |   |   |   |
```

And so on... this is how `lcsLength(_:)` fills in the entire matrix.

Backtracking to find the actual subsequence

So far we've calculated the length of every possible subsequence. The length of the longest subsequence is found in the bottom-right corner of matrix, at `matrix[n+1][m+1]`. In the above example it is 4, so the LCS consists of 4 characters.

Having the length of every combination of substrings makes it possible to determine *which* characters are part of the LCS itself by using a backtracking strategy.

Backtracking starts at `matrix[n+1][m+1]` and walks up and left (in this priority) looking for changes that do not indicate a simple propagation.

```
|   |  Ø|  A|  B|  D|  C|  A|  B|
| Ø |  0|  0|  0|  0|  0|  0|  0|
| A |  0|↖ 1|  1|  1|  1|  1|  1|
| B |  0|  1|↖ 2|← 2|  2|  2|  2|
| C |  0|  1|  2|  2|↖ 3|← 3|  3|
| B |  0|  1|  2|  2|  3|  3|↖ 4|
| X |  0|  1|  2|  2|  3|  3|↑ 4|
```

Each ↖ move indicates a character (in row/column header) that is part of the LCS.

If the number on the left and above are different than the number in the current cell, no propagation happened. In that case `matrix[i][j]` indicates a common char between the strings `a` and `b`, so the characters at `a[i - 1]` and `b[j - 1]` are part of the LCS that we're looking for.

One thing to notice is, as it's running backwards, the LCS is built in reverse order. Before returning, the result is reversed to reflect the actual LCS.

Here is the backtracking code:

```
  func backtrack(_ matrix: [[Int]]) -> String {
    var i = self.characters.count
    var j = other.characters.count

    var charInSequence = self.endIndex

    var lcs = String()

    while i >= 1 && j >= 1 {
          // Indicates propagation without change: no new char was added to lcs.
          if matrix[i][j] == matrix[i][j - 1] {
            j -= 1
          }
          // Indicates propagation without change: no new char was added to lcs.
          else if matrix[i][j] == matrix[i - 1][j] {
            i -= 1
            charInSequence = self.index(before: charInSequence)
          }
          // Value on the left and above are different than current cell.
          // This means 1 was added to lcs length.
          else {
            i -= 1
            j -= 1
            charInSequence = self.index(before: charInSequence)
            lcs.append(self[charInSequence])
          }
    }

    return String(lcs.characters.reversed())
  }
```

This backtracks from `matrix[n+1][m+1]` (bottom-right corner) to `matrix[1][1]` (top-left corner), looking for characters that are common to both strings. It adds those characters to a new string, `lcs`

The `charInSequence` variable is an index into the string given by `self`. Initially this points to the last character of the string. Each time we decrement `i`, we also move back `charInSequence`. When the two characters are found to be equal, we add the character at `self[charInSequence]` to the new `lcs` string. (We can't just write `self[i]` because `i` may not map to the current position inside the Swift string.)

Due to backtracking, characters are added in reverse order, so at the end of the function we call `reversed()` to put the string in the right order. (Appending new characters to the end of the string and then reversing it once is faster than always inserting the characters at the front of the string.)

Putting it all together

To find the LCS between two strings, we first call `lcsLength(_:)` and then `backtrack(_:)`:

```
extension String {
  public func longestCommonSubsequence(_ other: String) -> String {

    func lcsLength(_ other: String) -> [[Int]] {
      ...
    }

    func backtrack(_ matrix: [[Int]]) -> String {
      ...
    }

    return backtrack(lcsLength(other))
  }
}
```

To keep everything tidy, the two helper functions are nested inside the main `longestCommonSubsequence()` function.

Here's how you could try it out in a Playground:

```
let a = "ABCBX"
let b = "ABDCAB"
a.longestCommonSubsequence(b)   // "ABCB"

let c = "KLMK"
a.longestCommonSubsequence(c)   // "" (no common subsequence)

"Hello World".longestCommonSubsequence("Bonjour le monde")   // "oorld"
```

Written for Swift Algorithm Club by Pedro Vereza