1 Tabulation and Dynamic Programming

2 Warm-up: Pythonic make-change

Below is Racket code for make-change. Translate the code to Python.

\[
\text{(define (make-change total coin-list)} \\
\quad \text{(cond)} \\
\quad \quad \text{[(= total 0) true]} \\
\quad \quad \text{[(empty? coin-list) false]} \\
\quad \quad \text{[else (let* ([it (first coin-list)])} \\
\quad \quad \quad \text{[lose-it (rest coin-list)]} \\
\quad \quad \quad \text{[lose-it-solution (make-change total lose-it)]} \\
\quad \quad \quad \text{[use-it-solution (and (> it total) \\
\quad \quad \quad \quad \text{(make-change (- total it) lose-it)))]} \\
\quad \quad \quad \text{(or use-it-solution lose-it-solution))])} \\
\text{)} \\
\]

3 Tabulated Fibonacci

In [8]: def fib(n):
   
   ' 
   Given a positive integer n, returns the nth fibonacci number, where 
   \( \text{fib(1) = fib(2) = 1} \) 
   \( \text{fib(n) = fib(n-1) + fib(n-2)} \) 
   ' 
   
   assert n > 0, 'fib requires a positive number'
   
   if (n == 1) or (n == 2):
       return 1
   
   return fib(n-1) + fib(n-2)
In [16]: def tabulated_fib(n):
    '''
    Given a positive integer n, returns the nth fibonacci number, where
    fib(1) = fib(2) = 1
    fib(n) = fib(n-1) + fib(n-2)
    
    This function computes fibonacci using tabulation.
    '''

    assert n > 0, 'fib requires a positive number'

    size = max(n + 1, 3)
    table = [0 for i in range(size)]

    table[1] = 1
    table[2] = 1

    for i in range(3, size):
        table[i] = table[i - 1] + table[i - 2]

    return table[n]

In [18]: import experimental
   fibTrials = range(1, 15)
   iterations = 1000

In [19]: # take some measurements
   recursiveFibResults = experimental.timeTrials(fib, fibTrials, iterations)
   tabulatedFibResults = experimental.timeTrials(tabulated_fib, fibTrials, iterations)

In [20]: # plot the results
   %matplotlib inline
   experimental.plot(fibTrials, [recursiveFibResults, tabulatedFibResults], legend=['recursive', 'tabulated'])
3.1 Dynamic programming with tabulation: what we learned

1. We can write the recursive version first, to gain intuition about the dynamic-programming version.
2. For each recursive call + input, there is a corresponding cell in our dynamic-programming table.
3. To build the dynamic-programming table, we ask:
   - What do the cells mean? (recursion / table connection)
   - How many cells are there, for an input of size N?
   - Which cell contains the result, i.e., the answer to the full problem of size N?
   - What cells are easy to fill in? (base cases)
   - What rule fills in a cell? (inspired by the recursive call)
   - In what order should we fill the cells?

4 Tabulated factorial

In [25]: \texttt{def factorial(n):}
""
   \texttt{\_\_\_\_}
   \texttt{Given a non-negative integer n, returns n!}
   \texttt{\_\_\_\_}
   \texttt{The function computes n! recursively}
   \texttt{\_\_\_\_}
assert n >= 0, 'factorial requires a non-negative number'

if n == 0:
    return 1

return n * factorial(n-1)

In [22]: def tabulated_factorial(n):
    
    '''
    Given a non-negative integer n, returns n!
    
    The function computes n! using tabulation
    '''
    
    assert n >= 0, 'factorial requires a non-negative number'

    table = [None] * (n + 1)  # create the dynamic-programming table

    # base case
    table[0] = 1

    # fill the table
    for i in range(1, n + 1):
        table[i] = i * table[i - 1]

    return table[n]

In [26]: # take some measurements
    factTrials = range(50)
    iterations = 500

In [27]: recursiveFactResults = experimental.timeTrials(factorial, factTrials, iterations)

In [28]: tabulatedFactResults = experimental.timeTrials(tabulated_factorial, factTrials, iterations)

In [29]: # plot the results
    experimental.plot(factTrials, [recursiveFactResults, tabulatedFactResults], legend=['recursive', 'tabulated'])
How did this happen?!

"Its probably the most eagerly awaited genome since the chimp genome because platypuses are so weird," said Jenny Graves, one of the paper’s authors, and head of the Comparative Genomics Group at the Australian National University.

"You see genes that look reptile-like, genes that look bird-like and genes that look mammal-like. Its a pretty amazing picture," said Rick Wilson, director of the Genome Center at Washington University in St Louis.

Strings as inductive data structures

A string is a list of characters

\[ w \in \Sigma \]

\[ s \in \text{String} ::= \varepsilon \]

\[ | w \cdot s \]

- **Character**: \( w \in \Sigma \)
- **“Alphabet”**: (e.g., \{A...Z\} or \{0,1\})
- **Empty string**: \( \varepsilon \)
- **Concatenation**: \( w \cdot s \)

Python strings

- `'` # empty
- `s[0]` # (first s)
- `s[1:]` # (rest s)

Racket lists

- `# empty`
- `(first s)`
- `(rest s)`
longest-common substring (LCS)

How similar are these strings?

The longest-common substring of \textbf{s1} and \textbf{s2} is the longest string that is a \textit{non-consecutive} substring of both \textbf{s1} and \textbf{s2}.

\[
lcs('x', 'y') == 0 \quad lcs('car', 'cat') == 2
\]
\[
lcs('x', '') == 0 \quad lcs('human', 'chimpanzee') == 4
\]
\[
lcs('', 'x') == 0
\]
The longest-common substring of $s_1$ and $s_2$ is the longest string that is a non-consecutive substring of both $s_1$ and $s_2$.

```python
def lcs(s1, s2):
    '''Returns the longest-common substring of s1 and s1'''
    if not s1 or not s2:
        return 0
    elif s1[0] == s2[0]:
        return 1 + lcs(s1[1:], s2[1:])
    else:
        return max(lcs(s1, s2[1:]), lcs(s1[1:], s2))
```

How similar are these strings?
Sequence alignment

From Wikipedia, the free encyclopedia

In bioinformatics, a sequence alignment is a way of arranging the sequences of DNA, RNA, or protein to identify regions of similarity that may be a consequence of functional, structural, or evolutionary relationships between the sequences.[1] Aligned sequences of nucleotide or amino acid residues are typically represented as rows within a matrix. Gaps are inserted between the residues so that identical or similar characters are aligned in successive columns.

A sequence alignment, produced by ClustalW, of two human zinc finger proteins, identified on the left by GenBank accession number.

Edit distance

How different are these strings?

What’s the minimum number of modifications it takes to turn $s_1$ into $s_2$?

A “modification” can be:

- **substitute** one letter for another in one of the strings
- **delete** a letter from one of the strings
- **insert** a letter into one of the strings

<table>
<thead>
<tr>
<th>String Pair</th>
<th>Edit Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>cat vs $\varepsilon$</td>
<td>3</td>
</tr>
<tr>
<td>cat vs hat</td>
<td>1</td>
</tr>
<tr>
<td>cat vs at</td>
<td>1</td>
</tr>
<tr>
<td>hello vs below</td>
<td>3</td>
</tr>
<tr>
<td>spam vs scramble</td>
<td>5</td>
</tr>
</tbody>
</table>
def editDistance(first, second):
    ...  
    Returns the edit distance between the strings first and second.  
    ...
Editor distance

A recursive implementation

def editDistance(first, second):
    ... 
    
    Returns the edit distance between the strings first and second.

    ...
    
    if not first:
        return len(second)

    elif not second:
        return len(first)

Python idiom for empty sequence
def editDistance(first, second):
    
    Returns the edit distance between the strings first and second.
    
    if not first:
        return len(second)
    elif not second:
        return len(first)
    elif first[0] == second[0]:
        return editDistance(first[1:], second[1:])
    else:
        substitution = 1 + editDistance(first[1:], second[1:])
        deletion = 1 + editDistance(first[1:], second)
        insertion = 1 + editDistance(first, second[1:])
        return min(substitution, deletion, insertion)
Edit distance ⇒ spell-checker!

Ispell

From Wikipedia, the free encyclopedia

Ispell is a spelling checker for Unix that supports most Western languages. It offers several interfaces, including a programmatic interface for use by editors such as emacs. Unlike GNU Aspell, ispell will only suggest corrections that are based on a Damerau–Levenshtein distance of 1; it will not attempt to guess more distant corrections based on English pronunciation rules. Ispell has a very long history that can be traced back to a program that was originally written in 1971 in PDP-10 Assembly language by R. E. Gorin, and later ported to the C programming language and expanded by many others. It is currently maintained by Geoff Kuenning. The generalized affix description system introduced by ispell has since been imitated by other spelling checkers such as MySpell.

Like most computerized spelling checkers, ispell works by reading an input file word by word, stopping when a word is not