

# CS42\_Tabulation

November 7, 2018

## 1 Tabulation and Dynamic Programming

### 2 Warm-up: Pythonic make-change

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

```
(define (make-change total coin-list)
  (cond
    [(= total 0) true]

    [(empty? coin-list) false]

    [else (let* ([it (first coin-list)]
                 [lose-it (rest coin-list)]
                 [lose-it-solution (make-change total lose-it)]
                 [use-it-solution (and (>= total it)
                                       (make-change (- total it) lose-it))])
            (or use-it-solution lose-it-solution))]))
```

Firstname Lastname

T. 11/6

### 3 Tabulated Fibonacci

```
In [8]: def 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)
        '''

        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]

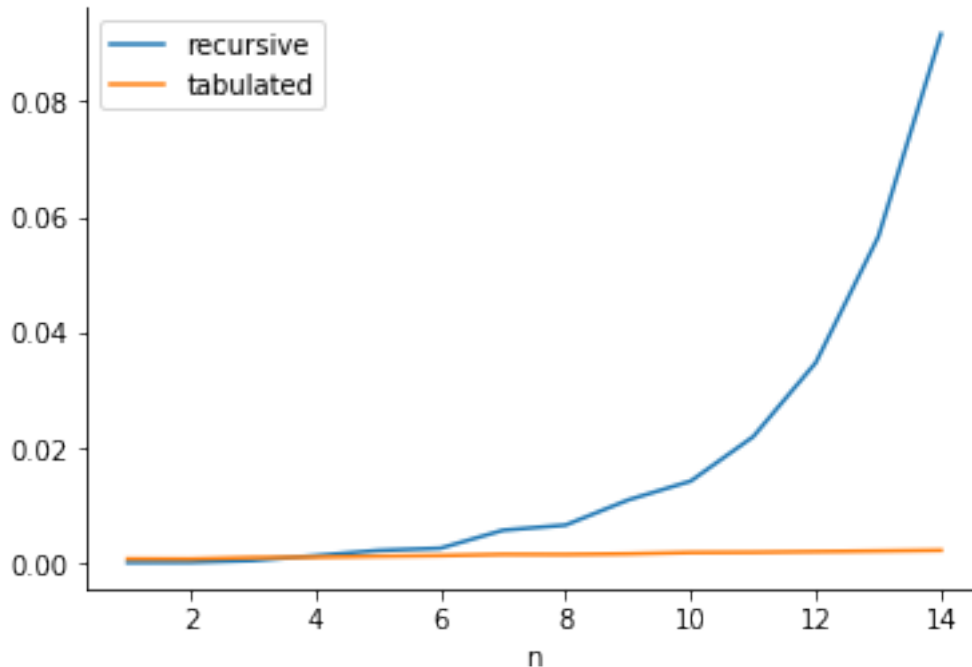
        return table[n]

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=['rec

```



### 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]: def factorial(n):
         '''
         Given a non-negative integer n, returns n!

         The function computes n! recursively
         '''
```

```

    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)`

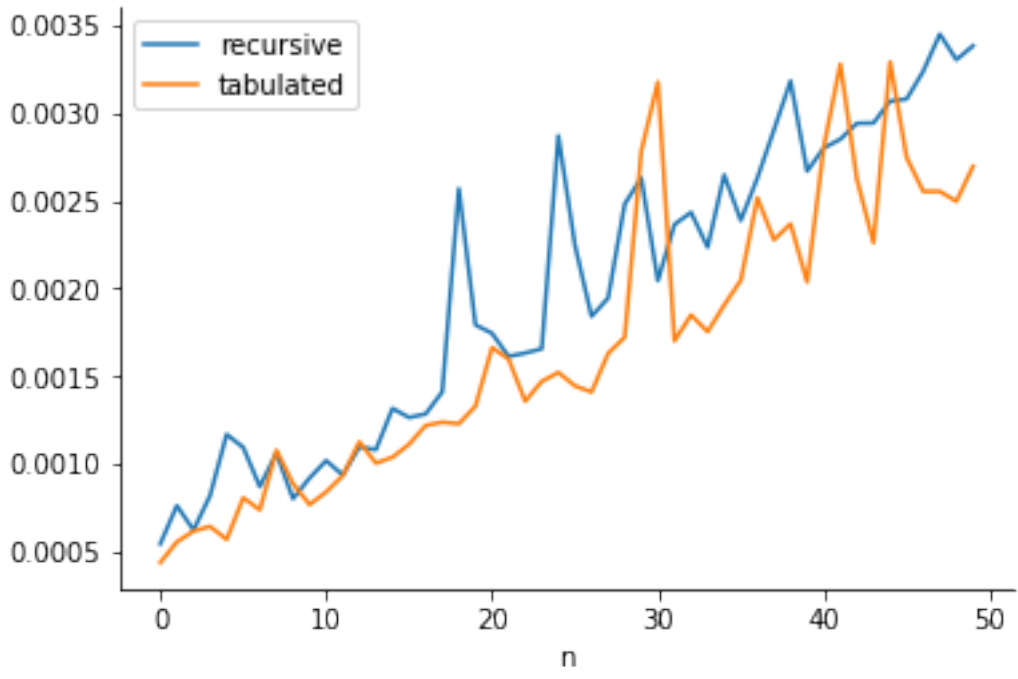
In [28]: `tabulatedFactResults = experimental.timeTrials(tabulated_factorial, factTrials, iterations)`

In [29]: `# plot the results`

```

experimental.plot(factTrials, [recursiveFactResults, tabulatedFactResults], legend=['recursive', 'tabulated'])

```







ORNITHORHYNCHUS ANATINUS .

*Shool and W. C. Beckler del. et sculp.*

*W. H. Bennett del. H. Wallis sculp.*



# How did this happen?!

				260							270										
Platypus	G	G	A	L	G	G	S	S	M	K	N	S	L	R	N	I	P	G	T	F	M
Medaka	-	G	R	L	S	G	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Chicken	G	G	T	F	V	G	S	A	M	K	N	S	L	R	S	L	P	A	T	Y	M
Pig	G	N	A	L	G	G	S	P	V	K	N	S	L	R	G	L	P	A	P	Y	V
Mouse	G	N	S	L	G	G	S	P	V	K	N	S	L	R	S	L	P	A	P	Y	V
Human	G	N	P	L	G	G	S	P	V	K	N	S	L	R	G	L	P	G	P	Y	V

*In the platypus a meiotic chain of ten sex chromosomes shares genes with the bird Z and mammal X chromosomes,*  
Grützner, et al. Nature 2004.



[commons.wikimedia.org/wiki/File:Platypus-sketch.jpg](http://commons.wikimedia.org/wiki/File:Platypus-sketch.jpg)

"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.

[www.sciencebuzz.org/blog/platypus-genome-reveals-natures-frankenstein-creature-one](http://www.sciencebuzz.org/blog/platypus-genome-reveals-natures-frankenstein-creature-one)

# Strings as inductive data structures

A string is a list of characters

$$\begin{array}{l} \text{character} \swarrow \\ w \in \Sigma \quad \swarrow \begin{array}{l} \text{"alphabet"} \\ \text{(e.g., \{A...Z\} or \{0,1\})} \end{array} \\ \\ s \in \text{String} ::= \varepsilon \quad \swarrow \text{empty string} \\ \quad \quad \quad | \quad w \bullet s \\ \quad \quad \quad \quad \quad \quad \quad \swarrow \text{concatenation} \end{array}$$

Python strings

`''`

`s[0]`

`s[1:]`

Racket lists

`# empty`

`# (first s)`

`# (rest s)`



# longest-common substring (LCS)

How similar are these strings?

The longest-common substring of **s1** and **s2** is the longest string that is a *non-consecutive* substring of both **s1** and **s2**.

`lcs('x', 'y') == 0`

`lcs('car', 'cat') == 2`

`lcs('x', '') == 0`

`lcs('human', 'chimpanzee') == 4`

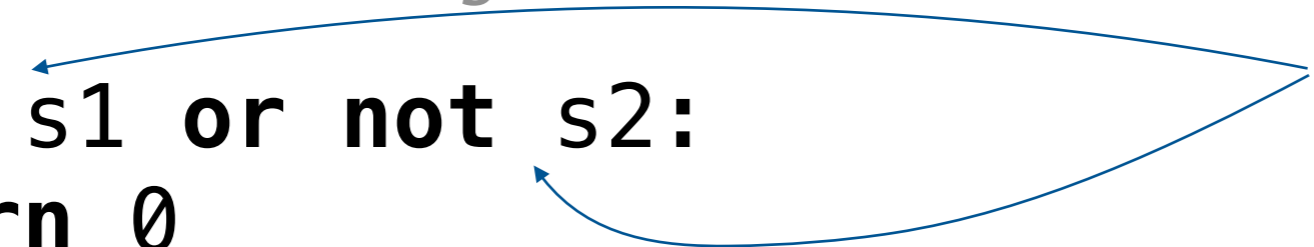
`lcs('', 'x') == 0`

# longest-common substring (LCS)

How similar are these strings?

The longest-common substring of **s1** and **s2** is the longest string that is a *non-consecutive* substring of both **s1** and **s2**.

```
def lcs(s1, s2):  
    '''Returns the longest-common substring of s1 and s2'''  
    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))
```



Python idiom  
for empty sequence

# lcs ⇒ DNA sequence alignment!

[Nature 2004](#)

				260				270					280				290				300																													
Platypus	G	G	A	L	G	G	S	S	M	K	N	S	L	R	N	I	P	G	T	F	M	S	S	Q	S	G	N	Q																						
Medaka	-	G	R	L	S	G	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	H	S	M	P	S	Q	Y	R	M	H	S	F	Y										
Chicken	G	G	T	F	V	G	S	A	M	K	N	S	L	R	S	L	P	A	T	Y	M	S	S	Q	S	G	K	Q	W	Q	M	K	G	M	E	N	R	H	A	M	S	S	Q	Y	R	M	C	S	Y	Y
Pig	G	N	A	L	G	G	S	P	V	K	N	S	L	R	G	L	P	A	P	Y	V	P	G	Q	T	G	N	Q	W	Q	M	K	N	S	E	T	R	H	A	V	S	S	Q	Y	R	M	H	S	Y	Y
Mouse	G	N	S	L	G	G	S	P	V	K	N	S	L	R	S	L	P	A	P	Y	V	P	A	Q	T	G	N	Q	W	Q	M	K	T	S	E	S	R	H	P	V	S	S	Q	Y	R	M	H	S	Y	Y
Human	G	N	P	L	G	G	S	P	V	K	N	S	L	R	G	L	P	G	P	Y	V	P	G	Q	T	G	N	Q	W	Q	M	K	N	M	E	N	R	H	A	M	S	S	Q	Y	R	M	H	S	Y	Y



**WIKIPEDIA**  
The Free Encyclopedia

- Main page
- Contents
- Featured content
- Current events
- Random article
- Donate to Wikipedia
- Wikimedia Shop

Interaction

- Help
- About Wikipedia
- Community portal
- Recent changes
- Contact page

Tools

- What links here
- Related changes
- Upload file
- Special pages
- Permanent link
- Page information


Create account Log in

Article [Talk](#)

Read [Edit](#) [View history](#)

## Sequence alignment

From Wikipedia, the free encyclopedia

 This article **needs additional citations for verification**. Please help [improve this article](#) by [adding citations to reliable sources](#). Unsourced material may be challenged and removed. *(March 2009)*

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.<sup>[1]</sup> 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.

```

AAB24882      TYHMCQFHCRYVNNHSGEKLYECNERSKAFSCPSHLQCHKRRQIGEKTHEHNQCGKAFPT  60
AAB24881      -----YECNQCCKAFAQHSSLKCHYRTHIGEKPYECNQCCKAFSK  40
                ****: .***: * *:* * :****.:* *****..

AAB24882      PSHLQYHERHTHTGKPYECHQCGQAFKKCSLLQRHKRHTHTGKPYE-CNQCCKAFAQ-  116
AAB24881      HSHLQCHKRHTHTGKPYECNQCCKAFSQHLLQRHKRHTHTGKPYMNVINMVKPLHNS  98
                **** *:*****:***:**.: *****: *.: :
    
```

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

Key: Single letters: **amino acids**. Red: small, hydrophobic, aromatic, not Y. Blue: acidic. Magenta: basic. Green: hydroxyl, amine, amide, basic. Gray: others. "": identical. ".": conserved substitutions (same colour group). ":": semi-conserved substitution (similar

# 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

cat vs  $\epsilon$  3

cat vs hat 1

cat vs at 1

hello vs below 3

spam vs scramble 5



# Edit distance

A recursive implementation

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

# Edit distance

A recursive implementation

```
def editDistance(first, second):
```

```
    '''
```

```
    Returns the edit distance between  
    the strings first and second.
```

```
    '''
```

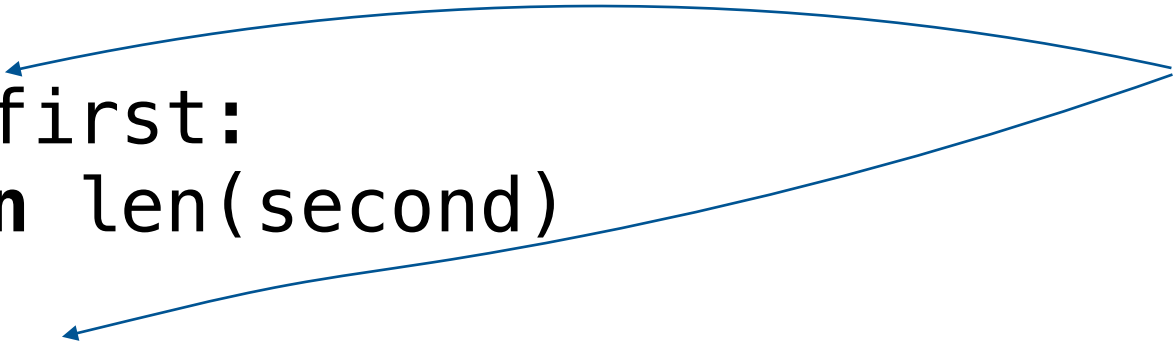
```
if not first:
```

```
    return len(second)
```

```
elif not second:
```

```
    return len(first)
```



Python idiom  
for empty sequence



# Edit 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)  
    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)
```



Python idiom  
for empty sequence

# Edit distance $\Rightarrow$ spell-checker!



WIKIPEDIA  
The Free Encyclopedia

[Main page](#)  
[Contents](#)  
[Featured content](#)  
[Current events](#)  
[Random article](#)  
[Donate to Wikipedia](#)  
[Wikimedia Shop](#)

Interaction  
[Help](#)

[Create account](#) [Log in](#)

Article [Talk](#) [Read](#) [Edit](#) [View history](#)

## 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