How to tell if one string is prefix of another

Minute paper

s1 is a prefix of s2 if the beginning of s2 is s1.

```python
def isPrefix(s1, s2):
    # fill me in
    # use slicing notation to compute the answer
```

(Your response)
Strings: notation

Strings are lists of characters

\[ w \in \Sigma \]

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

| \[ w \cdot s \]  

character

"alphabet"
(e.g., \{A...Z\} or \{0,1\})

empty string

concatenation
s1 is a prefix of s2 if the beginning of s2 is s1.

What are the inputs?
What is the output?
What are the base cases?
What are the recursive steps?
Recursive prefix

s1 is a prefix of s2 if the beginning of s2 is s1.

def isPrefix(s1, s2):
    # fill me in
    # use recursion to compute the answer
Recursive prefix

\( s_1 \) is a prefix of \( s_2 \) if the beginning of \( s_2 \) is \( s_1 \).

def isPrefix(s1, s2):
    if not s1:
        return True
    elif not s2:
        return False
    elif s1[0] == s2[0]:
        return isPrefix(s1[1:], s2[1:])
    else:
        return False
longest-common substring (LCS)

How similar are these strings?

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 \( s_2 \).

\[
\begin{align*}
\text{lcs('x', 'y')} & = 0 & \text{lcs('car', 'cat')} & = 2 \\
\text{lcs('x', '')} & = 0 & \text{lcs('human', 'chimpanzee')} & = 4 \\
\text{lcs('', 'x')} & = 0 \\
\end{align*}
\]
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 $s_2$.

What are the inputs?
What is the output?
What are the base cases?
What are the recursive steps?

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 \( s_1 \) 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):
    # fill me in with the length of the lcs
    # use recursion to compute the answer
```

```python
lcs('x', 'y') == 0  # lcs('car', 'cat') == 2
lcs('x', '') == 0   # lcs('human', 'chimpanzee') == 4
lcs('', 'x') == 0
```
How similar are these strings?

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

```python
def lcs(s1, 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))
```
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.


Sequence alignments are also used for non-biological sequences, such as those present in natural language or in financial data.
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 1</th>
<th>String 2</th>
<th>Edit Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>cat</td>
<td>$\varepsilon$</td>
<td>3</td>
</tr>
<tr>
<td>cat</td>
<td>hat</td>
<td>1</td>
</tr>
<tr>
<td>cat</td>
<td>at</td>
<td>1</td>
</tr>
<tr>
<td>hello</td>
<td>below</td>
<td>3</td>
</tr>
<tr>
<td>spam</td>
<td>scramble</td>
<td>5</td>
</tr>
</tbody>
</table>
Edit distance

A recursive algorithm

What are the inputs?
What is the output?
What are the base cases?
What are the recursive steps?
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)
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)
Edit distance ⇒ spell-checker!