In [1]:
```
import numpy as np
# from IPython.display import display
```

Tracy Michaels

CSC 4740 Data Mining - Assignment 2

Implementing an algorithm for computing edit distance

Defining Edit Distance:

> Edit distance is a way of quantifying how dissimilar two strings are to one another by counting the minimum number of operations required to transfor one string into the other

Applications of Edit Distance:

> - Natual Language Processing (Spell check, speach recognition)
> - Computational Biology (DNA analysis)

Levenshtein Distance:

> - For this assignment I will focus on Levenshtein distance, which is one type of edit distance
> - This will calculate the minimum number of single-character edits (insertions, deletions, substitutions) required to transform one string into the other

> 3 operations for altering a string (each with a cost of 1 to perform):
>
> - insert
> - delete
> - replace

> last index in array will indicate the least number of operations to edit the strings to match

Algorithm:

In [2]:

```python
def LevDis(s1: str, s2: str) -> int:
    # construct array of size len(s1) by len(s2)
    # fill row 0 and col 0 with incremental numbers from 0 to length of strings + 1 respectively
    # adding one is done so that position [0][0] is a 0
    lev_arr = np.zeros((len(s2) + 1, len(s1) + 1))

    # filling initial values in row 0 and col 0
    lev_arr[0] = [i for i in range(len(lev_arr[0]))]
    lev_arr[:, 0] = [i for i in range(len(lev_arr[:, 0]))]


    # iterate over array
    for i in range(len(s2)):
        for j in range(len(s1)):
            # char matches, no incrementation
            if s1[j] == s2[i]:
                # take the lowest value from 'backwards-adjacent' indecies
                # (i.e. [i-1][j-1], [i-1][j], or [i][j-1]) and place into target index
                # need +1 for target index as strings are 0-indexed but array will be
                # effectively 1-indexed since col[0] and row[0] are reserved for initial set-up
                lev_arr[i + 1, j + 1] = min(lev_arr[i+1, j], lev_arr[i, j+1], lev_arr[i, j])
            else:
                # doesn't match, will need to perform insert, delete, or replace
                # increment lowest value from 'backwards-adjacent' indecies
                # (i.e. [i-1][j-1], [i-1][j], or [i][j-1]) then place into target index
                lev_arr[i + 1, j + 1] = min(lev_arr[i+1, j], lev_arr[i, j+1], lev_arr[i, j]) + 1
            # prints each step
            # print(f'target = [{j+1}][{i+1}]')
            # print(np.matrix(lev_arr))

    # displays array (might not work outside of jupyter as it depends on IPython.display)
    #  display(lev_arr)
    # just use this one so it works outside notebook
    print(np.matrix(lev_arr))

    # return min edit distance
    return lev_arr[-1, -1]
```

Example 1:

> sString1 = "kitten"
> sString2 = "sitting"

In [3]:

```python
s1_1 = "kitten"
s2_1 = "sitting"

res_ex1 = LevDis(s1_1, s2_1)

print(f'Minimum edit distance between \'{s1_1}\' and \'{s2_1}\' is {res_ex1}' )
```

```
[[0. 1. 2. 3. 4. 5. 6.]
 [1. 1. 2. 3. 4. 5. 6.]
 [2. 2. 1. 2. 3. 4. 5.]
 [3. 3. 2. 1. 1. 2. 3.]
 [4. 4. 3. 1. 1. 2. 3.]
 [5. 5. 3. 2. 2. 2. 3.]
 [6. 6. 4. 3. 3. 3. 2.]
 [7. 7. 5. 4. 4. 4. 3.]]
Minimum edit distance between 'kitten' and 'sitting' is 3.0
```

Example 2:

sString1 = "GAMBOL"
sString2 = "GUMBO"

In [4]:

```
s1_2 = "GAMBOL"
s2_2 = "GUMBO"

res_ex2 = LevDis(s1_2, s2_2)

print(f'Minimum edit distance between \'{s1_2}\' and \'{s2_2}\' is {res_ex2}' )
```

```
[[0. 1. 2. 3. 4. 5. 6.]
 [1. 0. 1. 2. 3. 4. 5.]
 [2. 1. 1. 2. 3. 4. 5.]
 [3. 2. 2. 1. 2. 3. 4.]
 [4. 3. 3. 2. 1. 2. 3.]
 [5. 4. 4. 3. 2. 1. 2.]]
Minimum edit distance between 'GAMBOL' and 'GUMBO' is 2.0
```

Observations:

One observation I found interesting was the relationship between the target index and those indicies surrounding it, and how that relationship played a large role in the algorithm itself. The output makes sense as in each step the algorithm is selecting the minimum value from the select adjacent indicies culminiating in the last one being the one that would represent the least amount of steps required to transform one into the others.