UNIVERSITÀ DEGLI STUDI DI NAPOLI
PARTHENOPE

Natural Language Processing

Spelling correction

LESSON 15

prof. Antonino Staiano

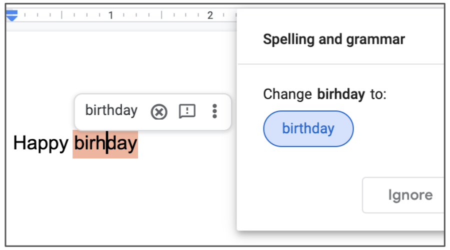
M.Sc. In ''Machine Learning e Big Data'' - University Parthenope of Naples

Slides taken from DeepLearning.AI and from the slides accompanying the textbook by D. Jurafsky

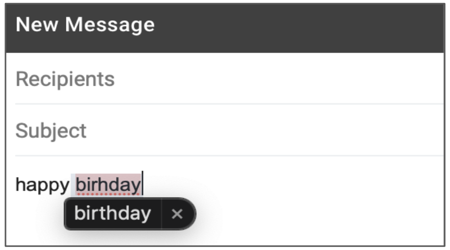
What is autocorrect?

- Is a task that changes misspelled words into correct ones
  - Phones
  - Tablets
  - Computers

Happy birhday



Happy birhday



New Message

Recipients

Subject

happy birhday

Autocorrect: Example

- *Happy birthday <u>deah</u> friend!*
  - *Happy birthday dear friend!*
    - *Non-word spelling correction*

- What if you typed *deer* instead of *dear*?
  - *Happy birthday <u>deer</u> friend!*
    - The word is spelled correctly, but its context is incorrect
      - Real-world spelling correction

How autocorrect works

- Identify a misspelled word
- Find strings n edit distance away
- Filter candidates
- Compute word probabilities

Steps for autocorrect

- Identify a misspelled word
- Find strings n edit distance away
- Filter candidates
- Compute word probabilities

deah

Steps for autocorrect

- Identify a misspelled word
- Find strings n edit distance away
- Filter candidates
- Compute word probabilities

deah

_eah

d_ar

de_r

etc.

Steps for autocorrect

- Identify a misspelled word
- Find strings n edit distance away
- Filter candidates
- Compute word probabilities

deah

yeah

dear

dean

…

Steps for autocorrect

- Identify a misspelled word
- Find strings n edit distance away
- Filter candidates
- Compute word probabilities

deah

yeah

dear

dean

…

Steps for autocorrect

- Identify a misspelled word
- Find strings n edit distance away
- Filter candidates
- Compute word probabilities

deah → dear

yeah

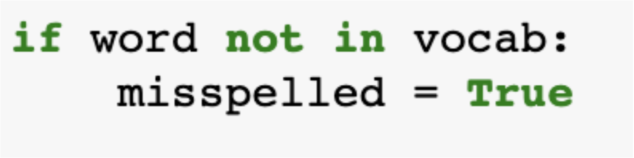
dear

dean

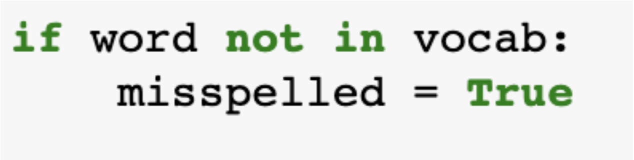
…

Building the model

- Identify a misspelled word
  - How to do that?
    - If it’s spelled correctly, it can be found in the dictionary, otherwise, it’s probably a misspelled word



```
if word not in vocab:
    misspelled = True
```



```
if word not in vocab:
    misspelled = True
```

deah ?? 🤔

deah ✗ deer 🦌 ✓

Happy birthday deer ! 🦌 ✓ 🎂

Building the model

- Find string n edit distance away
  - Given a string find all possible strings that are n edit distance away using
    - Insert
    - Delete
    - Switch
    - Replace
  - Edit distance counts the number of these operations so that the n edit distance tells you how many operations away one string is from another

deah

_eah

d_ar

de_r

etc.

Edit distance

- **Edit**: an operation performed on a string to change it
  - Insert (add a letter)
    - to: top, two, …
  - Delete (remove a letter)
    - hat: ha, at, ht
  - Switch (swap two adjacent letters)
    - eta: eat, tea, …
    - It does not include switching two letters that are not next to each other (e.g., ate)
  - Replace (change one letter to another)
    - jaw: jar, paw
- By combining these edits, you can find a list of all possible strings that's are n edit away
  - For autocorrect, n is typically 1-3 edits

Building the model

- Filter candidates
  - Many of the strings that are generated do not look like actual words
    - Keep ones that are real words (correctly spelled)
      - Compare it to a dictionary or vocabulary

deah	deah
_eah	yeah
d_ar	dear
de_r	dean
etc.	…

Building the model

- Compute word probabilities
  - "*I am happy because I am learning*"

Word	Count
I	2
am	2
happy	1
because	1
learning	1

Total : 7

$$P(w) = \frac{C(w)}{V}$$

$P(w)$	Probability of a word
$C(w)$	Number of times the word appears
V	Total size of the corpus

$$\boxed{P(\mathrm{am}) = \frac{C(\mathrm{am})}{V} = \frac{2}{7}}$$

deah → dear

yeah

dear

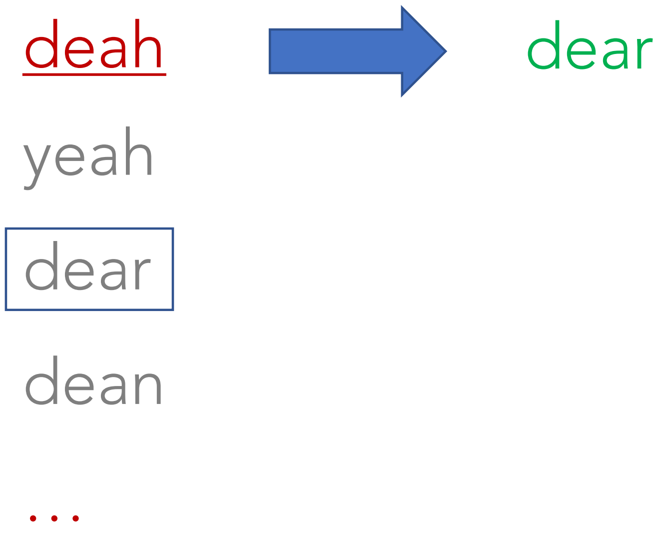
dean

. . .

Summarizing

- Identify a misspelled word
- Find strings n edit distance away
  - Insert
  - Delete
  - Switch
  - Replace
- Filter candidates
- Calculate word probabilities

$$P(w) = \frac{C(w)}{V}$$

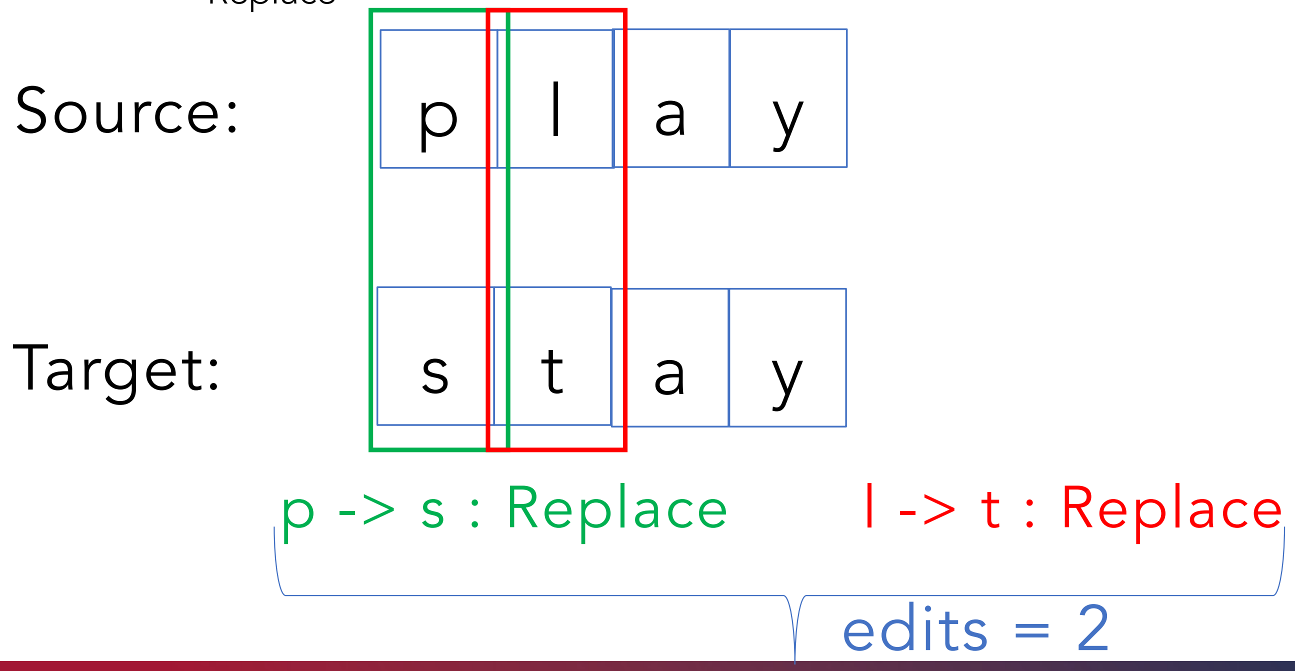


Minimum edit distance

- How to evaluate similarity between two strings?
  - Minimum number of edits needed to transform one string into the other
  - Several applications
    - Spelling correction, document similarity, machine translation, DNA sequencing, and more

Minimum edit distance: Example

- What is the minimum number of edits to turn *play* into *stay*?
  - Remember that edits include
    - Insert
    - Delete
    - Replace



p -> s : Replace

l -> t : Replace

edits = 2

Edit cost:

Insert & Delete	1
Replace	2

Edit distance = 2+2=4

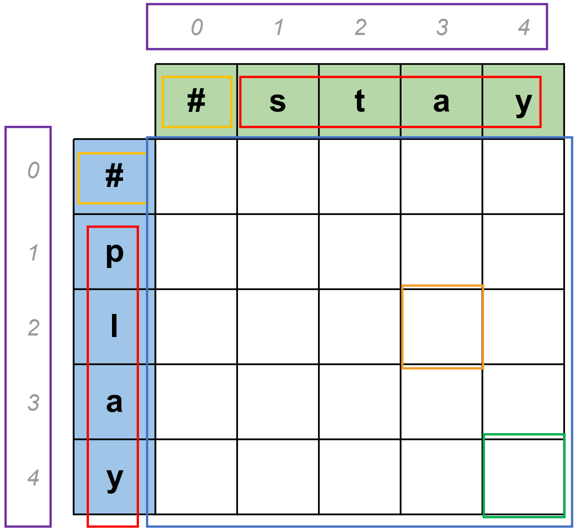
Minimum edit distance

- Note that as your strings get larger it gets much harder to calculate the minimum edit distance
  - We could use a brute force approach adding one edit distance at a time and enumerating all possibilities until one string changes to the other
    - Exponential computational complexity in the size of the string!!!
- Example
  - "*convolutionalneuralnetworks*"
  - CCAAGGGGTGACTCTAGTTTAATATAACTTTAAGGGGTAGTTTAT
- We need to speed up the enumeration of all possible strings and edits
  - A tabular approach -> dynamic programming!!!

Minimum edit distance

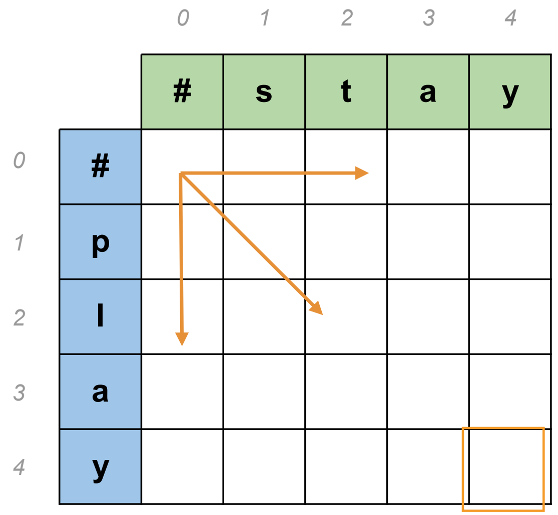
- Source: play -> Target: stay

- D[]
- D[2,3] = pl -> sta
- D[2,3] = source[:2] -> target[:3]
- D[i,j] = source[:i] -> target[:j]
- D[m,n] = source -> target



Minimum edit distance

- Source: play -> Target: stay
  - D[]
  - D[i,j] = source[:i] -> target[:j]
  - D[m,n] = source -> target



		0	1	2	3	4
		#	s	t	a	y
0	#					
1	p					
2	l					
3	a					
4	y					

Minimum edit distance

Source: play → Target: stay

Cost: insert: 1, delete: 1, replace: 2

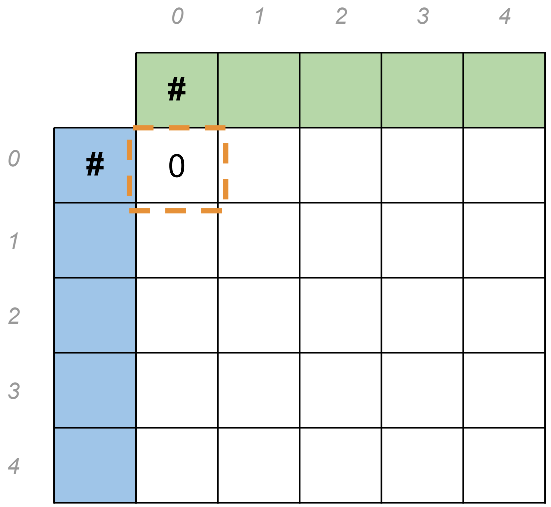
		0	1	2	3	4
		#	**s**	**t**	**a**	**y**
0	**#**					
1	**p**					
2	**l**					
3	**a**					
4	**y**					

Minimum edit distance

Source: play → Target: stay

Cost: insert: 1, delete: 1, replace: 2

→

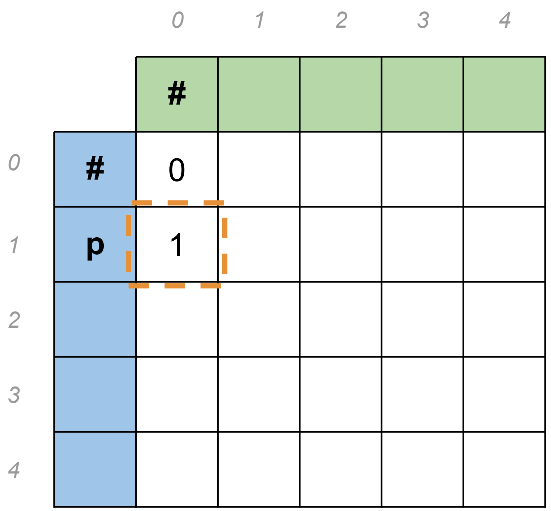


		0	1	2	3	4
		#				
0	#	0				
1						
2						
3						
4						

Minimum edit distance

Source: play → Target: stay
Cost: insert: 1, delete: 1, replace: 2

p → #
delete

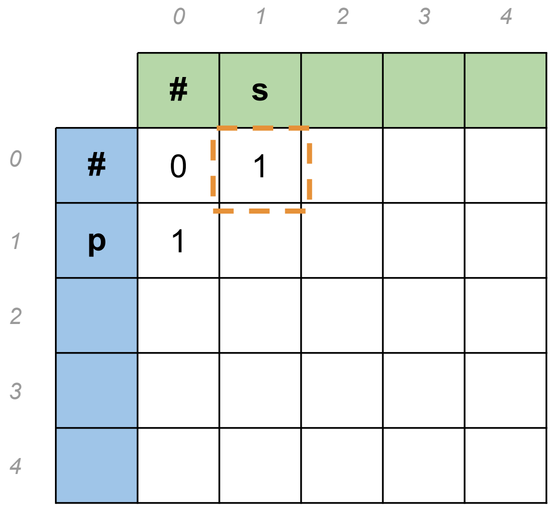


		0	1	2	3	4
		#				
0	**#**	0				
1	**p**	1				
2						
3						
4						

Minimum edit distance

Source: play → Target: stay
Cost: insert: 1, delete: 1, replace: 2

\# → s
insert



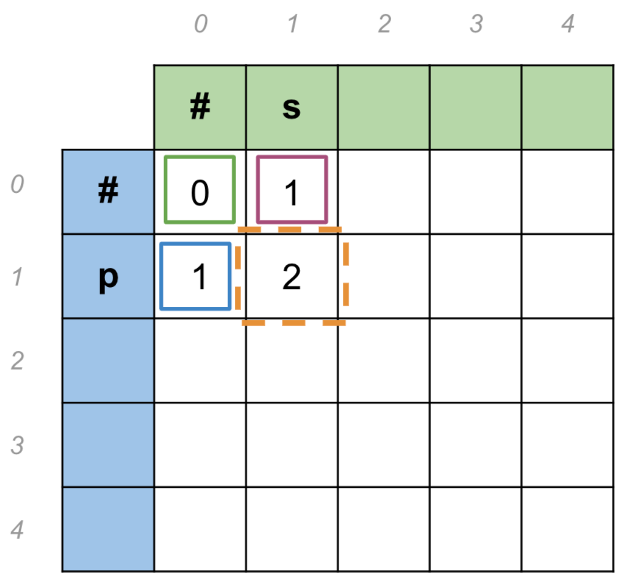
		0	1	2	3	4
		#	**s**			
0	**#**	0	1			
1	**p**	1				
2						
3						
4						

Minimum edit distance algorithm

- When computing the minimum edit distance, one would start with a source word and transform it into the target word

Source: play → Target: stay
Cost: insert: 1, delete: 1, replace: 2

p → s
insert + *delete*: p → ps → s: 2
delete + *insert*: p → # → s: 2
replace: p → s: 2

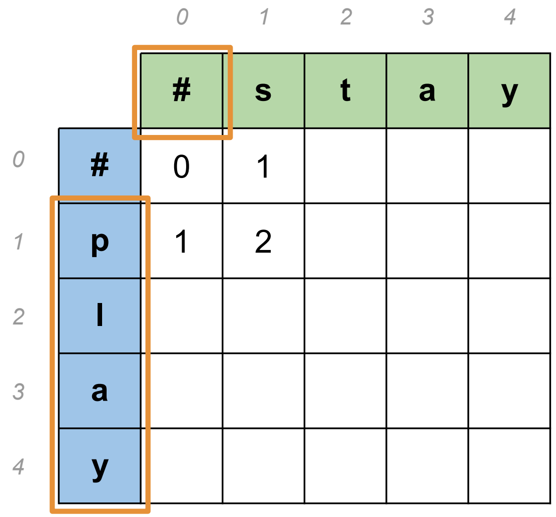


		0	1	2	3	4
		#	**s**			
0	**#**	0	1			
1	**p**	1	2			
2						
3						
4						

Minimum edit distance

Source: play → Target: stay
Cost: insert: 1, delete: 1, replace: 2

play → #



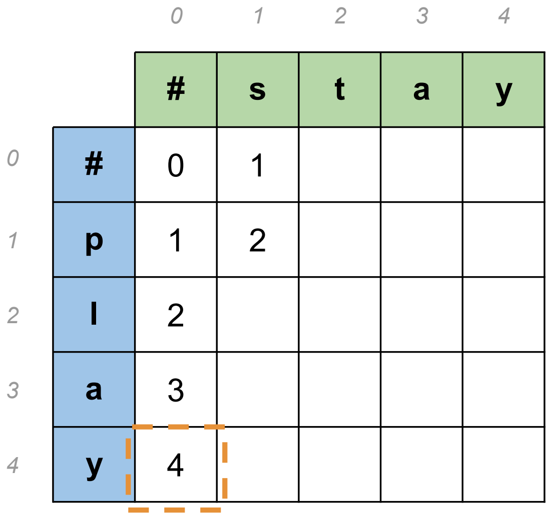
		0	1	2	3	4
		#	**s**	**t**	**a**	**y**
0	**#**	0	1			
1	**p**	1	2			
2	**l**					
3	**a**					
4	**y**					

Source: play → Target: stay
Cost: insert: 1, delete: 1, replace: 2

play → #

$D[\, i, j \,] = D[\, i\text{-}1, j \,] + del\_cost$

$D[4,0]$ = play → #
= source[:4] → target[0]



		0	1	2	3	4
		#	**s**	**t**	**a**	**y**
0	**#**	0	1			
1	**p**	1	2			
2	**l**	2				
3	**a**	3				
4	**y**	4				

Minimum edit distance

Source: play → Target: stay
Cost: insert: 1, delete: 1, replace: 2

\# → play

		0	1	2	3	4
		#	**s**	**t**	**a**	**y**
0	**#**	0	1			
1	**p**	1	2			
2	**l**	2				
3	**a**	3				
4	**y**	4				

Source: play → Target: stay
Cost: insert: 1, delete: 1, replace: 2

\# → play

$D[\, i, j \,] = D[\, i, j\text{-}1 \,] + ins\_cost$

		0	1	2	3	4
		#	**s**	**t**	**a**	**y**
0	**#**	0	1	2	3	4
1	**p**	1	2			
2	**l**	2				
3	**a**	3				
4	**y**	4				

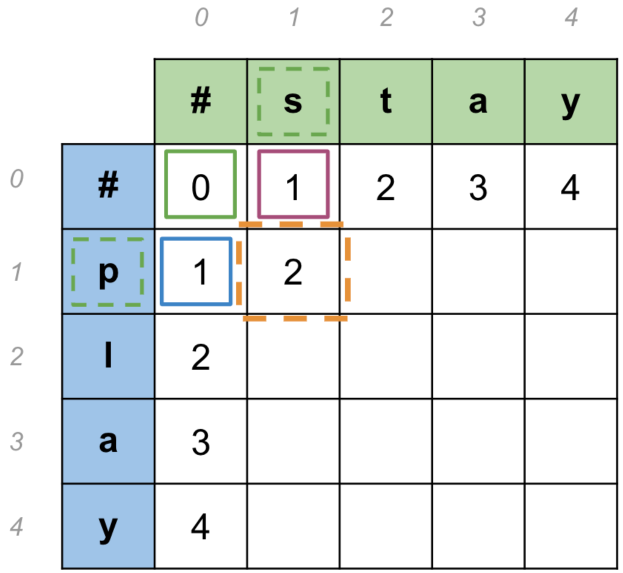
Minimum edit distance algorithm

- To populate the table

Source: play → Target: stay
Cost: insert: 1, delete: 1, replace: 2

p → s

$$D[i, j] = \min \begin{cases} D[i-1, j] + del\_cost \\ D[i, j-1] + ins\_cost \\ D[i-1, j-1] + \begin{cases} rep\_cost; & \text{if } src[i] \neq tar[j] \\ 0; & \text{if } src[i] = tar[j] \end{cases} \end{cases}$$



		0	1	2	3	4
		#	s	t	a	y
0	#	0	1	2	3	4
1	p	1	2			
2	l	2				
3	a	3				
4	y	4				

- At every time step one checks the three possible paths where he can come from and select the least expensive one

Defining Min Edit Distance (Levenshtein)

- Initialization

```
D(i,0) = i
D(0,j) = j
```

- Recurrence Relation:

```
For each  i = 1…M
      For each  j = 1…N
```

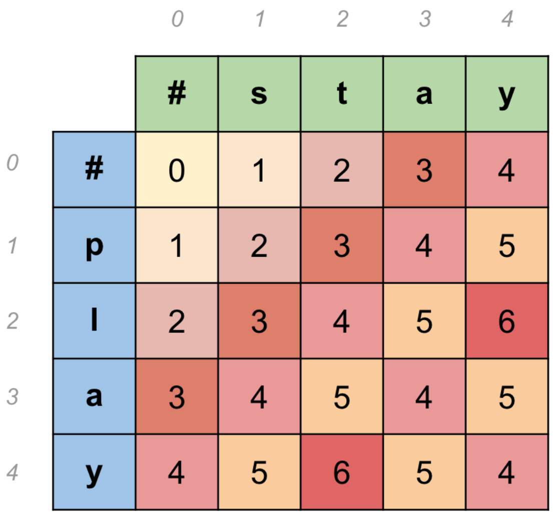
$$D(i,j) = \min \begin{cases} D(i-1,j) + 1 & \text{delete} \\ D(i,j-1) + 1 & \text{insert} \\ D(i-1,j-1) + \begin{cases} 2; & \text{if } X(i) \neq Y(j) \\ 0; & \text{if } X(i) = Y(j) \end{cases} & \text{replace} \end{cases}$$

- Termination:

`D(M,N) is the minimum distance`

Minimum edit distance

Source: play → Target: stay
Cost: insert: 1, delete: 1, replace: 2

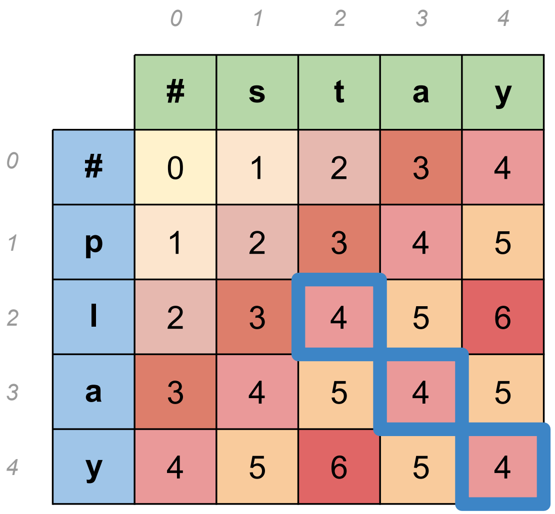


		0	1	2	3	4
		#	**s**	**t**	**a**	**y**
0	**#**	0	1	2	3	4
1	**p**	1	2	3	4	5
2	**l**	2	3	4	5	6
3	**a**	3	4	5	4	5
4	**y**	4	5	6	5	4

Source: play → Target: stay
Cost: insert: 1, delete: 1, replace: 2

play → stay

D[m, n] = 4



		0	1	2	3	4
		#	**s**	**t**	**a**	**y**
0	**#**	0	1	2	3	4
1	**p**	1	2	3	4	5
2	**l**	2	3	4	5	6
3	**a**	3	4	5	4	5
4	**y**	4	5	6	5	4

Computing alignment

- Edit distance isn’t sufficient
  - We often need to **align** each character of the two strings to each other
- We do this by keeping a “backtrace”
- Every time we enter a cell, remember where we came from
- When we reach the end,
  - Trace back the path from the upper right corner to read off the alignment

Adding Backtrace to Minimum Edit Distance

- Base conditions:

  `D(i,0) = i` `D(0,j) = j`

Termination:

`D(N,M) is distance`

- Recurrence Relation:

```
For each  i = 1…M
    For each  j = 1…N
```

$$D(i,j) = \min \begin{cases} D(i-1,j) + 1 & \text{delete} \\ D(i,j-1) + 1 & \text{insert} \\ D(i-1,j-1) + \begin{cases} 2; \text{ if } X(i) \neq Y(j) & \text{replace} \\ 0; \text{ if } X(i) = Y(j) \end{cases} \end{cases}$$

$$ptr(i,j) = \begin{cases} \text{LEFT} & \text{insert} \\ \text{UP} & \text{delete} \\ \text{DIAG} & \text{replace} \end{cases}$$

Computational complexity

- Time
    - O(nm)
- Space
    - O(nm)
- Backtrace
    - O(n+m)