

Inexact matching, sequence alignment and dynamic programming

—

Edit distance, alignment and dynamic programming

Overview

- Reflection on string matching and string similarity
- Edit distance
- Recursive calculation of edit distance
- Dynamic programming and calculation of edit distance

Exact vs inexact matching

- Up until now we were referring to exact matching
 - Good for sequence search problems but not for sequence comparison
- Effect of sequencing error and mutational processes
- Approximate matching: certain errors are acceptable in valid matches

Exact vs inexact matching

- **Alignment:** lining up characters of strings, allowing mismatches as well as matches, and allowing characters of one string to be placed opposite spaces made in opposing string.
- Usage:
 - Difference between two strings, string similarity
 - String alignment
 - Global alignment (end-to-end alignment)
 - Local alignment

Exact vs inexact matching

- Substring vs subsequence
 - Substring: contiguous
 - Subsequence: doesn't have to be contiguous

Example:

- String: **axayaz**
- Subsequence: **xyz** - not a substring

First fact of biological sequence analysis

In biomolecular sequence (DNA, RNA or amino acid sequences) high sequence similarity usually implies significant functional or structural similarity.

(Note: opposite not necessarily true)

Evolution:

- Reuses, builds on and modifies “successful” structures
- Duplication with modification - central paradigm of protein evolution

Importance of (sub)sequence comparison in Molecular Biology

Redundancy:

- “...built in characteristics of protein sequences”
- New sequences resemble already known sequences (between species)

Example:

- Same genes that work in flies are the ones that work in humans
- “...studies on yeast are remarkable predictors of the human system”

Consequence: **sequence similarity (redundancy) implies functional similarity**

- Helps us to infer *conserved* features
- Large-scale sequence comparison (database search) is a very powerful tool in biological inference in molecular biology

Measure for sequence similarity (distance)

If we could measure distance between two strings we could quantify similarity between organisms

Need for measure of difference or *distance* between two strings (**distance function** or **metrics**)

- Spelling correction, evolutionary studies of biological strings...

Hamming distance

Number of single letter substitutions needed from S to T.

A	C	T	G	A	C	T	G	A	C	T	G
A	C	T	G	A	G	T	G	T	T	T	G

Hamming distance of 3

Hamming distance - not so great

- Works only on strings of the same length
- Does not properly reflect mutational processes occurring in the genome
- For strings of the same length:
 - $\text{editDistance}(x,y) \leq \text{hammingDistance}(x,y)$
- If they are not:
 - $\text{editDistance}(x,y) \geq ||x|-|y||$

G	A	G	G	T	A	G	C	G	G	C	G	T	T	T	A	A	C
G	T	G	G	T	A	A	C	G	G	G	G	T	T	T	A	A	C

Hamming distance

G	C	G	T	A	T	G	C	G	G	C	T	A	-	A	C	G	C
G	C	-	T	A	T	G	C	G	G	C	T	A	T	A	C	G	C

Edit distance

Edit distance (Levenshtein distance)

Edit distance: focuses on editing(transforming) one string into the other by a series of edit operations on individual characters.

Which operations?

Edit distance - edit operations

A *mismatch* is a single-character substitution:

X: G T A G C G G C G
| | | | |
Y: G T A A C G G C G

denoted R - replacement(substitution)

An *edit* is a single-character substitution or gap (*insertion* or *deletion*):

X: G T A G C G G C G
| | | | |
Y: G T A A C G G C G

X: G T A G C - G C G
| | | | |
Y: G T A G C G G C G

Gap in X

AKA insertion in Y or deletion in X

X: G T A G C G G C G
| | | | |
Y: G T - G C G G C G

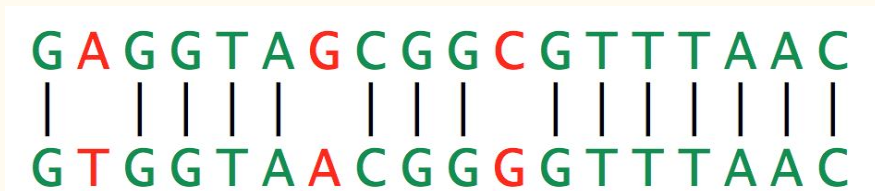
Gap in Y

AKA insertion in X or deletion in Y

And M for *match* or nonoperation...

Alignment

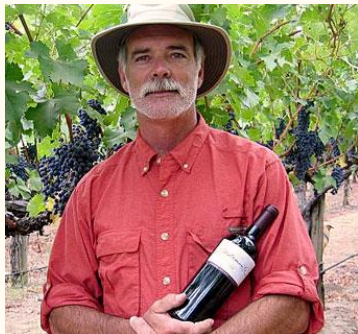
This is an alignment: way of lining up characters of x and y.



Alignment: lining up characters of strings, allowing mismatches as well as matches, and allowing characters of one string to be placed opposite spaces made in opposing string.

Edit distance - an example

Transforming “vintner” into “writers”



RIMDMDMMI

v i n t n e r

| | | |

w r i t e r s



Edit distance and edit transcript

Can think of edits as being introduced by an optimal editor working left-to-right.
Edit transcript describes how editor turns x into y.

x: GCGTATGCGGCTAACGC
y: GCTATGCGGCTATACGC

x: GC **G** TATGCGGCTAACGC
| |
y: GC - TATGCGGCTA **T** ACGC

x: GC **G** TATGCGGCTA - ACGC
| | | | | | | | | |
y: GC - TATGCGGCTA **T** ACGC

x: GC **G** TATGCGGCTA - ACGC
| | | | | | | | | |
y: GC - TATGCGGCTA **T** ACGC

Operations:

M = match, **R** = replace,

I = insert into x, **D** = delete from x

MMD

MMDMMMMMMMMMMI

MMDMMMMMMMMMMMMIMMMM

Edit distance and edit transcript

Edit transcript: A string over alphabet D, R, M, I that describes a transformation of one string to another is called an *edit transcript*, or transcript for short, of two strings.

Edit distance: Edit distance between two strings is defined as a minimum number of edit operations - insertions, deletions and substitutions - needed to transform first string into second. **Note:** matches are not counted.

Optimal transcript: Edit transcript that uses minimal number of edit operations.
(note: there may be more than one optimal edit transcripts, *cooptimal transcripts*)

Edit distance and edit transcript

Alignments:

x: G C G T A T G C G G C T A - A C G C
| | | | | | | | | | | |
y: G C - T A T G C G G C T A T A C G C

x: G C G T A T G A G G C T A - A C G C
| | | | | | | | | | | |
y: G C - T A T G C G G C T A T A C G C

```
x: the longest ---
    | | | | |
y: --- longest day
```

Edit transcripts with respect to x :

MM**D**MMMMMMMMMM**I**MMM

Distance = 2

MMDMMMRMMMMIMMM

Distance = 3

DDDDMMMMMMMMIIII

Distance = 8

Edit distance problem

Edit distance problem: compute edit distance between two strings with an optimal edit transcript that describes the transformation.

Important note: edit distance is defined as a number of operation on first string needed to transform it into the second string.

But!: it's also sometimes though as the minimum number of operations done on either of two strings to transform them into the common third string. This is ok.

Why?: Insertion in one string can be viewed as deletion in another.

Digression: more on alignment

(Global) alignment of two strings S1 and S2 is obtained by first inserting chosen spaces (or dashes) either into (or at the end of) S1 or S2 and then placing one resulting string above the other so that so that every character (or space) in one string is opposite a unique character (or space) in another string.

- An alternative way (to edit transcript) to represent transformation of one string to another

```
v_intner_  
wri_t_ers
```

Alignment vs edit distance

- **Mathematically they are equivalent** ways in describing the relations between two strings
- Can be converted from one another easily
 - Example: vrinter, writer
- Edit distance is given by the alignment minimizing the number of opposing characters that mismatch plus the number of characters opposite spaces.

Alignment vs edit distance

- **Not equivalent** from modelling perspective

Edit transcript **vs** **Alignment**

Mutational events

Process

vs

Relationships between the two strings

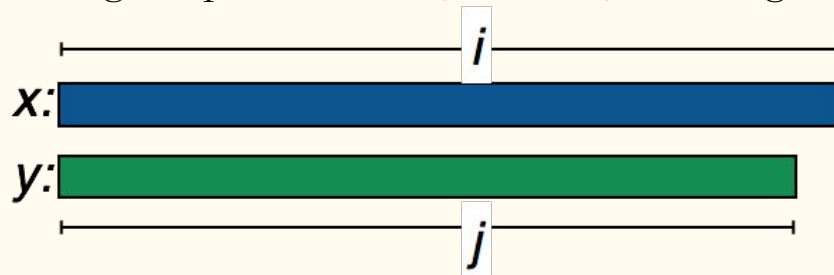
Product

- Alignment alone blurs mutational model
 - Different evolutionary models have different edit operations which can result with same alignments

Dynamic programming and calculation of edit distance

Def: $D(i, j)$ denotes the minimum number of edit operations needed to transform the first i characters of $S1$ into the first j characters of $S2$.

$D[i, j]$: edit distance between length- i prefix of $S1$ ($S1[1...i]$) and length- j prefix of $S2$ ($S2[1...i]$).



Think in terms of edit transcript. Optimal transcript for $D[i, j]$ can be built by extending a shorter one by 1 operation. Only 3 options:

- Append **D** to transcript for $D[i-1, j]$
- Append **I** to transcript for $D[i, j-1]$
- Append **M** or **R** to transcript for $D[i-1, j-1]$

$D[i, j]$ is minimum of the three, and $D[|x|, |y|]$ is the overall edit distance


Recurrence relation

Base conditions:

$$D(i,0)=i$$

$$D(0,j)=j$$

and recurrence relation:

$$D(i,j)=\min [D(i-1,j)+1, D(i,j-1)+1, D(i-1,j-1)+\delta(i,j)], \text{ where}$$


$\delta(i,j)=1$ if $i \neq j$ (**R**) else if $i=j$, $\delta(i,j)=0$ (**M**).

Recurrence relation

We can easily implement it as a recursion (see notebook and play a bit with it).

Issue? -> Running time

```
>>> import datetime as d
>>> st = d.datetime.now(); \
... edDistRecursive("Shakespeare", "shake spear"); \
... print (d.datetime.now()-st).total_seconds()
3
31.498284
```

Because we have $(n+1) \times (m+1)$ potential recursive calls.

Recurrence relation

Solution: memoization.

Remember memoize when we once calculate certain $D(i,j)$, then reuse it if the need appears (code in the notebook).

Dynamic programming

Dynamic programming is a method for solving a complex problem by breaking it down into a collection of simpler subproblems, solving each of those subproblems just once, and storing their solutions.

The technique of storing solutions to subproblems instead of recomputing them is called "memoization".

Dynamic programming

“For many optimization problems, the following principle of optimality holds: **an optimal solution is composed of optimal solutions to subproblems**. If a subproblem has several optimal solutions, it does not matter which one is used.

The idea of dynamic programming is to build an exhaustive table of optimal solutions. We start with trivial subproblems. We build optimal solutions for increasingly larger problems by constructing them from tabulated solutions of smaller problems.”

K. Mehlhorn, P. Sanders(2010): “Algorithms and Data structures”, Springer

Dynamic programming organizes computations to avoid re- computing values that you already know, which can often save a great deal of time.

N.C.Jones, P. Pevzner, P. Sanders(2004): “An Introduction to Bioinformatics Algorithms”, MIT Press

Recurrence relation

Solution: memoization

Remember memoize when we once calculate certain $D(i,j)$, then reuse it if the need appears (code in the notebook).

```
>>> import datetime as d
>>> st = d.datetime.now(); \
... edDistRecursiveMemo("Shakespeare", "shake spear"); \
... print (d.datetime.now()-st).total_seconds()
3
0.000593
```

Recurrence relation

But!

For large strings, it would still take a lot of time (remember, we are talking about genome-scale data).

Recursive approach is **top-down** approach.

How about **bottom-up** approach?

Dynamic programming and calculation of edit distance

Dynamic programming is a method for solving a complex problem by breaking it down into a collection of simpler subproblems, solving each of those subproblems just once, and storing their solutions.

If $\text{len}(S1)=n$ and $\text{len}(S2)=m$, we compute $D(n,m)$ by computing $D(i,j)$ for all i,j combinations where $i \in [1,n]$ and $j \in [1,m]$.

Dynamic programming (in our case) has three components:

1. Recurrence relation
2. Tabular (bottom-up) computation
3. Traceback


Recurrence relation

Base conditions:

$$D(i,0)=i$$

$$D(0,j)=j$$

and recurrence relation:

$$D(i,j)=\min [D(i-1,j)+1, D(i,j-1)+1, D(i-1,j-1)+\delta(i,j)], \text{ where}$$


$\delta(i,j)=1$ if $i \neq j$ (**R**) else if $i=j$, $\delta(i,j)=0$ (**M**).

Tabular (bottom-up) computation

ϵ is empty string

y

ϵ G C T A T G C C A C G C

$D: x$

ϵ G C G T A T G C A C G C

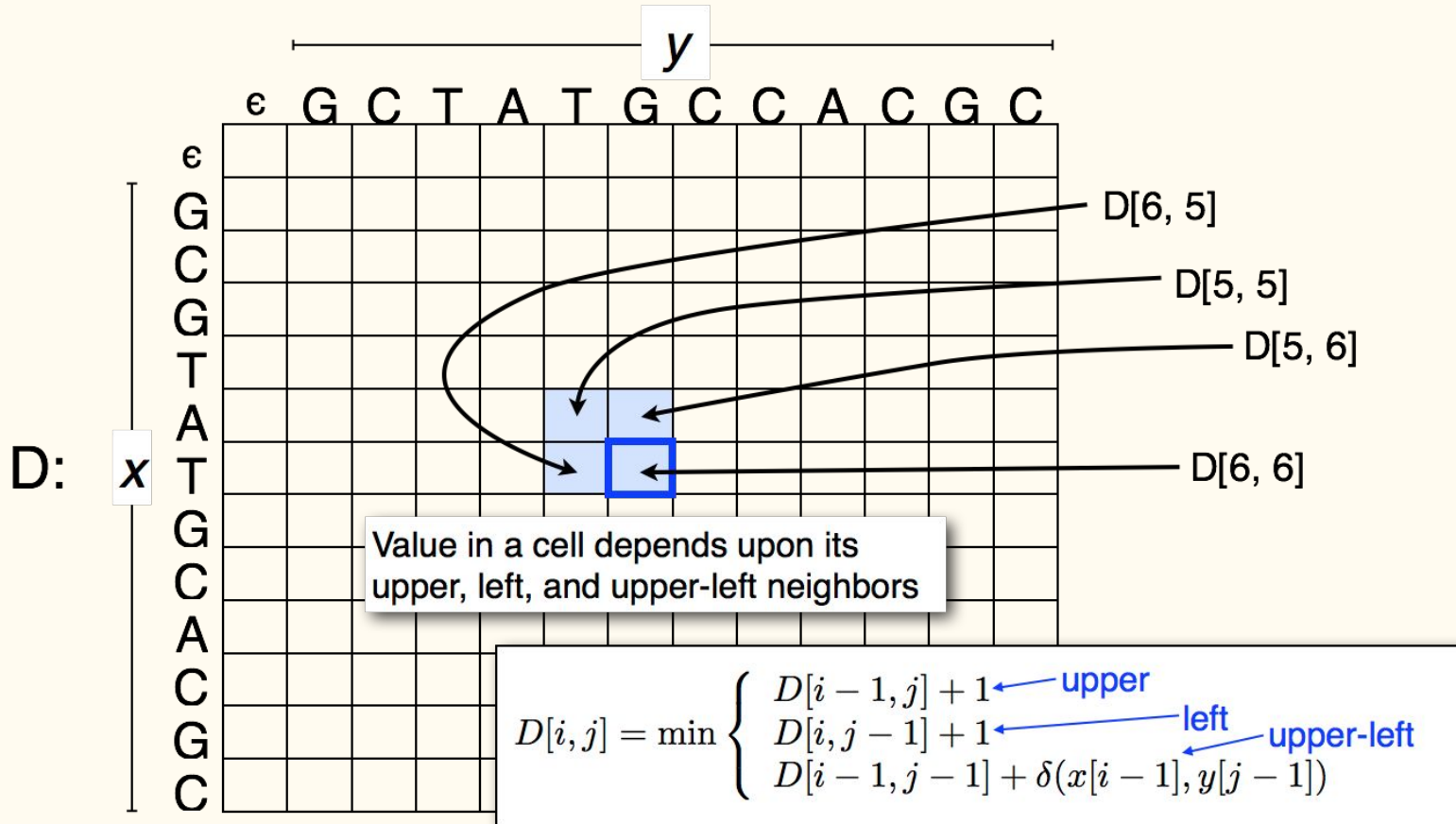
ϵ	ϵ	G	C	T	A	T	G	C	C	A	C	G	C
G													
C													
G													
T													
A													
T													
G													
C													
A													
C													
G													
C													

Let $n = |x|$, $m = |y|$

D : $(n+1) \times (m+1)$
matrix

$D[i, j]$ = edit distance b/t
length- i prefix of x and
length- j prefix of y

Tabular (bottom up) computation - recurrence relation



Tabular (bottom up) computation - table initialization

	ε	G	C	T	A	T	G	C	C	A	C	G	C
ε	0	1	2	3	4	5	6	7	8	9	10	11	12
G	1												
C	2												
G	3												
T	4												
A	5												
T	6												
G	7												
C	8												
A	9												
C	10												
G	11												
C	12												

Initialize $D[0, j]$ to j ,
 $D[i, 0]$ to i

Tabular (bottom up) computation - filling up the table

	ε	G	C	T	A	T	G	C	C	A	C	G	C
ε	0	1	2	3	4	5	6	7	8	9	10	11	12
G	1												
C	2												
G	3												
T	4												
A	5						etc						
T	6												
G	7												
C	8												
A	9												
C	10												
G	11												
C	12												

Fill remaining cells from
top row to bottom and
from left to right

Tabular (bottom up) computation

	ε	G	C	T	A	T	G	C	C	A	C	G	C
ε	0	1	2	3	4	5	6	7	8	9	10	11	12
G	1	?											
C	2												
G	3												
T	4												
A	5												
T	6												
G	7												
C	8												
A	9												
C	10												
G	11												
C	12												

Fill remaining cells from
top row to bottom and
from left to right

What goes here in $i=1, j=1$?

$x[i-1] = y[j-1] = \text{'G'}$,

SO $\text{delt} = 0$

$D[i, j] = \min(D[i-1, j-1] + \text{delt},$
 $D[i-1, j] + 1,$
 $D[i, j-1] + 1)$
 $= \min(0 + 0, 1 + 1, 1 + 1)$
 $= 0$

Fast forward...

	ε	G	C	T	A	T	G	C	C	A	C	G	C
ε	0	1	2	3	4	5	6	7	8	9	10	11	12
G	1	0	1	2	3	4	5	6	7	8	9	10	11
C	2	1	0	1	2	3	4	5	6	7	8	9	10
G	3	2	1	1	2	3	3	4	5	6	7	8	9
T	4	3	2	1	2	2	3	4	5	6	7	8	9
A	5	4	3	2	1	2	3	4	5	5	6	7	8
T	6	5	4	3	2	1	2	3	4	5	6	7	8
G	7	6	5	4	3	2	1	2	3	4	5	6	7
C	8	7	6	5	4	3	2	1	2	3	4	5	6
A	9	8	7	6	5	4	3	2	2	2	3	4	5
C	10	9	8	7	6	5	4	3	2	3	2	3	4
G	11	10	9	8	7	6	5	4	3	3	3	2	3
C	12	11	10	9	8	7	6	5	4	4	3	3	2

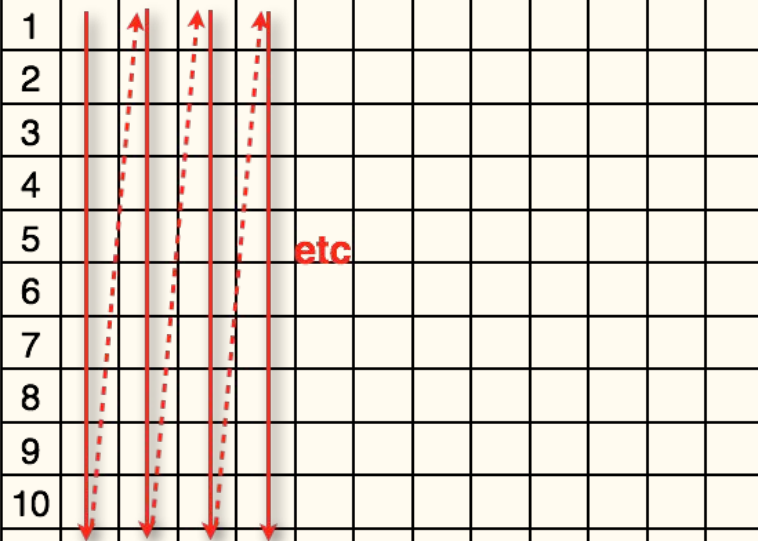
Fill remaining cells from
top row to bottom and
from left to right

Edit distance for x, y



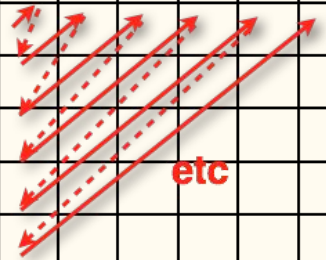
Filling up the matrix - vertically

	ε	G	C	T	A	T	G	C	C	A	C	G	C
ε	0	1	2	3	4	5	6	7	8	9	10	11	12
G	1												
C	2												
G	3												
T	4												
A	5												
T	6												
G	7												
C	8												
A	9												
C	10												
G	11												
C	12												



Filling up the matrix - antidiagonal

	ε	G	C	T	A	T	G	C	C	A	C	G	C
ε	0	1	2	3	4	5	6	7	8	9	10	11	12
G	1												
C	2												
G	3												
T	4												
A	5												
T	6												
G	7												
C	8												
A	9												
C	10												
G	11												
C	12												



Filling up the matrix - antidiagonal

	ε	G	C	T	A	T	G	C	C	A	C	G	C
ε	0	1	2	3	4	5	6	7	8	9	10	11	12
G	1												
C	2												
G	3												
T	4												
A	5												
T	6												
G	7												
C	8												
A	9												
C	10												
G	11												
C	12												

etc

Tabular computation - time complexity

To calculate value of cell (i,j) we need to examine cells $(i-1,j)$, $(i,j-1)$, $(i-1,j-1)$ and characters $S1(i)$ and $S2(j)$.

This is constant amount of time (per cell).

There are $n \times m$ cells, so it's $O(nm)$ time.

Part 3: Traceback

- Leaving pointers during table computation so we can use them to find a way back (Hansel and Gretel/Ivica & Marica)
- We set a pointer from the cell(i,j) to the cell from which the value was computed



Part 3: Traceback

How do we get optimal edit transcript then? We traceback from cell (m,n) to $(0,0)$ by using pointers which were set while the table is computed.

- (During table computation) Set pointers from cell (i,j) to cell from where the value (of (i,j)) was computed
- Follow (any) path of pointers from cell (m,n) to cell $(0,0)$
- Generating edit transcript (for each edge in the path):
 - a. Horizontal edge interpret as insertion (of $S2(j)$ to $S1^*$)
 - b. Vertical edge interpret as deletion (of $S1(i)$)
 - c. Diagonal edge as match or substitution (depending if $S1(i)=S2(j)$ or not)
- Traceback corresponds to optimal alignment/edit transcript

***Note:** $S1$ (or x) is reference, while $S2$ (or y) is read being compared to the reference.

Part 3: Traceback

Note: In all our examples/definitions for alignment S1 (or x) is reference, while S2 (or y) is read compared to the reference.

This is important when retrieving edit transcript from traceback through table. When we define horizontal movement/edge as insertion, it needs to be defined with respect to which string is on the horizontal and which is on the vertical axis of the table. Typically reference (S1) is on the vertical and read (S2) is on horizontal axis.

This is important because we can always define edit either as insertion in S1 or as deletion from S2, so it's important to know which string is the reference in order to be consistent.

Traceback

$D(i, j)$			w	r	i	t	e	r	s
		0	1	2	3	4	5	6	7
	0	0	← 1	← 2	← 3	← 4	← 5	← 6	← 7
v	1	↑ 1	↖ 1	↖ ← 2	↖ ← 3	↖ ← 4	↖ ← 5	↖ ← 6	↖ ← 7
i	2	↑ 2	↖ ↑ 2	↖ 2	↖ 2	← 3	← 4	← 5	← 6
n	3	↑ 3	↖ ↑ 3	↖ ↑ 3	↖ ↑ 3	↖ 3	↖ ← 4	↖ ← 5	↖ ← 6
t	4	↑ 4	↖ ↑ 4	↖ ↑ 4	↖ ↑ 4	↖ 3	↖ ← 4	↖ ← 5	↖ ← 6
n	5	↑ 5	↖ ↑ 5	↖ ↑ 5	↖ ↑ 5	↑ 4	↖ 4	↖ ← 5	↖ ← 6
e	6	↑ 6	↖ ↑ 6	↖ ↑ 6	↖ ↑ 6	↑ 5	↖ 4	↖ ← 5	↖ ← 6
r	7	↑ 7	↖ ↑ 7	↖ 6	↖ ← ↑ 7	↑ 6	↑ 5	↖ 4	← 5

Possible alignments:

S1: v i n t n e r s
 S2: w r i t _ e r s

v _ i n t n e r _
 w r i _ t _ e r s

_ v i n t n e r _
 w r i _ t _ e r s

Part 3: Traceback

How do we get optimal edit transcript then? We traceback from cell (m,n) to $(0,0)$ by using pointers which were set while the table is computed.

What if there are multiple pointers pointing from the cell?

- If there is more than one pointer from the cell, pick any
 - a. Each cell (except $(0,0)$) has a pointer out of it, you can't get stuck
 - b. Any path from cell (m,n) to $(0,0)$ represents optimal edit transcript
 - c. Pointer allow all optimal edit transcripts to be retrieved

Traceback in practice

Without pointers: start at the end, at each step ask, which predecessor **gave** the minimum*

	ε	G	C	T	A	T	G	C	C	A	C	G	C
ε	0	1	2	3	4	5	6	7	8	9	10	11	12
G	1	0	1	2	3	4	5	6	7	8	9	10	11
C	2	1	0	1	2	3	4	5	6	7	8	9	10
G	3	2	1	1	2	3	3	4	5	6	7	8	9
T	4	3	2	1	2	2	3	4	5	6	7	8	9
A	5	4	3	2	1	2	3	4	5	5	6	7	8
T	6	5	4	3	2	1	2	3	4	5	6	7	8
G	7	6	5	4	3	2	1	2	3	4	5	6	7
C	8	7	6	5	4	3	2	1	2	3	4	5	6
A	9	8	7	6	5	4	3	2	2	2	3	4	5
C	10	9	8	7	6	5	4	3	2	3	2	3	4
G	11	10	9	8	7	6	5	4	3	3	3	2	3
C	12	11	10	9	8	7	6	5	4	4	3	3	2

*Note: This does not mean “which predecessor has the minimum value”. It means we need to calculate which predecessor **gave** the minimum (value of our current cell). That is: which predecessor plus the edit operation penalty needed for moving from this predecessor to current cell gives the minimal (current) cell value.

A: From here

Q: How did I get here?

Traceback in practice

Without pointers: start at the end, at each step ask, which predecessor gave the minimum

	ε	G	C	T	A	T	G	C	C	A	C	G	C
ε	0	1	2	3	4	5	6	7	8	9	10	11	12
G	1	0	1	2	3	4	5	6	7	8	9	10	11
C	2	1	0	1	2	3	4	5	6	7	8	9	10
G	3	2	1	1	2	3	3	4	5	6	7	8	9
T	4	3	2	1	2	2	3	4	5	6	7	8	9
A	5	4	3	2	1	2	3	4	5	5	6	7	8
T	6	5	4	3	2	1	2	3	4	5	6	7	8
G	7	6	5	4	3	2	1	2	3	4	5	6	7
C	8	7	6	5	4	3	2	1	2	3	4	5	6
A	9	8	7	6	5	4	3	2	2	2	3	4	5
C	10	9	8	7	6	5	4	3	2	3	2	3	4
G	11	10	9	8	7	6	5	4	3	3	3	2	3
C	12	11	10	9	8	7	6	5	4	4	3	3	2

A: From here

Q: How did I get here?

Traceback in practice

Without pointers: start at the end, at each step ask, which predecessor gave the minimum

	ε	G	C	T	A	T	G	C	C	A	C	G	C
ε	0	1	2	3	4	5	6	7	8	9	10	11	12
G	1	0	1	2	3	4	5	6	7	8	9	10	11
C	2	1	0	1	2	3	4	5	6	7	8	9	10
G	3	2	1	1	2	3	3	4	5	6	7	8	9
T	4	3	2	1	2	2	3	4	5	6	7	8	9
A	5	4	3	2	1	2	3	4	5	5	6	7	8
T	6	5	4	3	2	1	2	3	4	5	6	7	8
G	7	6	5	4	3	2	1	2	3	4	5	6	7
C	8	7	6	5	4	3	2	1	2	3	4	5	6
A	9	8	7	6	5	4	3	2	2	2	3	4	5
C	10	9	8	7	6	5	4	3	2	3	2	3	4
G	11	10	9	8	7	6	5	4	3	3	3	2	3
C	12	11	10	9	8	7	6	5	4	4	3	3	2

A: From here

Q: How did I get here?

Traceback in practice

Without pointers: start at the end, at each step ask, which predecessor gave the minimum.
Path corresponds to edit transcript.

	ε	G	C	T	A	T	G	C	C	A	C	G	C
ε	0	1	2	3	4	5	6	7	8	9	10	11	12
G	1	0	1	2	3	4	5	6	7	8	9	10	11
C	2	1	0	1	2	3	4	5	6	7	8	9	10
G	3	2	1	0	1	2	3	3	4	5	6	7	8
T	4	3	2	1	0	1	2	3	4	5	6	7	8
A	5	4	3	2	1	0	1	2	3	4	5	6	7
T	6	5	4	3	2	1	0	1	2	3	4	5	6
G	7	6	5	4	3	2	1	0	1	2	3	4	5
C	8	7	6	5	4	3	2	1	0	1	2	3	4
A	9	8	7	6	5	4	3	2	1	0	1	2	3
C	10	9	8	7	6	5	4	3	2	1	0	1	2
G	11	10	9	8	7	6	5	4	3	2	1	0	1
C	12	11	10	9	8	7	6	5	4	3	2	1	0

Alignment:

```

G C G T A T G - C A C G C
| |   | | |   | | | | |
G C - T A T G C A C G C
  
```

Edit transcript:

MM**D**M**M**M**M**M**I**M**M**M**M**M

Traceback complexity

If the table is already filled, retrieving optimal edit transcript can be computed in $O(n+m)$ time.

Any cooptimal edit transcript can also be calculated in $O(n+m)$ time.

Bottom up computation - summary

Matrix-filling dynamic programming algorithm is $O(mn)$ time and space:

- Filling matrix is $O(mn)$ space and time, and yields edit distance
- Backtrace is $O(m + n)$ time, yields optimal alignment / edit transcript

Acknowledgment

Slides/pictures on some of the slides used with permission from Ben Langmead from John Hopkins university.