Technische Hochschule Nürnberg

Fakultät Informatik

Sequence Learning

Comparing Sequences

Korbinian Riedhammer

Dynamic programming and edit distance

Ben Langmead

JOHNS HOPKINS
WHITING SCHOOL of ENGINEERING

Department of Computer Science

Beyond approximate matching: sequence similarity

In many settings, Hamming and edit distance are too simple. Biologically-relevant distances require algorithms. We will expand our tool set accordingly.

```
Score =  248 bits (129), Expect = 1e-63
Identities = 213/263 (80%), Gaps = 34/263 (12%)
Strand = Plus / Plus

Query: 161 atatcaccacgtcaaaggtgactccaactcca---ccactccattttgttcagataatgc 217
           |||||||||||||||||||||||||||||  |     | |   || ||||||||||||||
Sbjct: 481 atatcaccacgtcaaaggtgactccaact-tattgatagtgttttatgttcagataatgc 539

Query: 218 ccgatgatcatgtcatgcagctccaccgattgtgagaacgacagcgacttccgtcccagc 277
           |||||||   ||||||||||||||||||||| || |            ||||||||||||
Sbjct: 540 ccgatgactttgtcatgcagctccaccgattttg-g------------ttccgtcccagc 586

Query: 278 c-gtgcc--aggtgctgcctcagattcaggttatgccgctcaattcgctgcgtatatcgc 334
           |  || |  | ||||||||||||||||||||||||||||||||||||||| |||||||||
Sbjct: 587 caatgacgta-gtgctgcctcagattcaggttatgccgctcaattcgctgggtatatcgc 645

Query: 335 ttgctgattacgtgcagctttcccttcaggcggga------------ccagccatccgtc 382
           |||||||||||||||||||||||||||||||||||            |||||||||||||
Sbjct: 646 ttgctgattacgtgcagctttcccttcaggcgggattcatacagcggccagccatccgtc 705

Query: 383 ctccatatc-accacgtcaaagg 404
            |||||||| |||||||||||||
Sbjct: 706 atccatatcaaccacgtcaaagg 728
```

Example BLAST alignment

JOHNS HOPKINS
WHITING SCHOOL
of ENGINEERING

Approximate string matching

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
```

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
```

Gap in X

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

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
```

AKA *insertion* in *X* or *deletion* in *Y*

Gap in Y

JOHNS HOPKINS
WHITING SCHOOL
of ENGINEERING

Alignment

```
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
```

Above is an *alignment*: a way of lining up the characters of *x* and *y*

Could include mismatches, gaps or both

Vertical lines are drawn where opposite characters match

JOHNS HOPKINS
WHITING SCHOOL
of ENGINEERING

Hamming and edit distance

Finding Hamming distance between 2 strings is easy:

```
def hammingDistance(x, y):
    assert len(x) == len(y)
    nmm = 0
    for i in xrange(0, len(x)):
        if x[i] != y[i]:
            nmm += 1
    return nmm
```

```
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
```

Edit distance is harder:

```
def editDistance(x, y):
    ???
```

```
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
```

JOHNS HOPKINS
WHITING SCHOOL
of ENGINEERING

Edit distance

```
def editDistance(x, y):
    return ???
```

```
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
```

If strings *x* and *y* are same length, what can we say about `editDistance`(*x*, *y*) relative to `hammingDistance`(*x*, *y*)?

`editDistance`(*x*, *y*) ≤ `hammingDistance`(*x*, *y*)

If strings *x* and *y* are different lengths, what can we say about `editDistance`(*x*, *y*)?

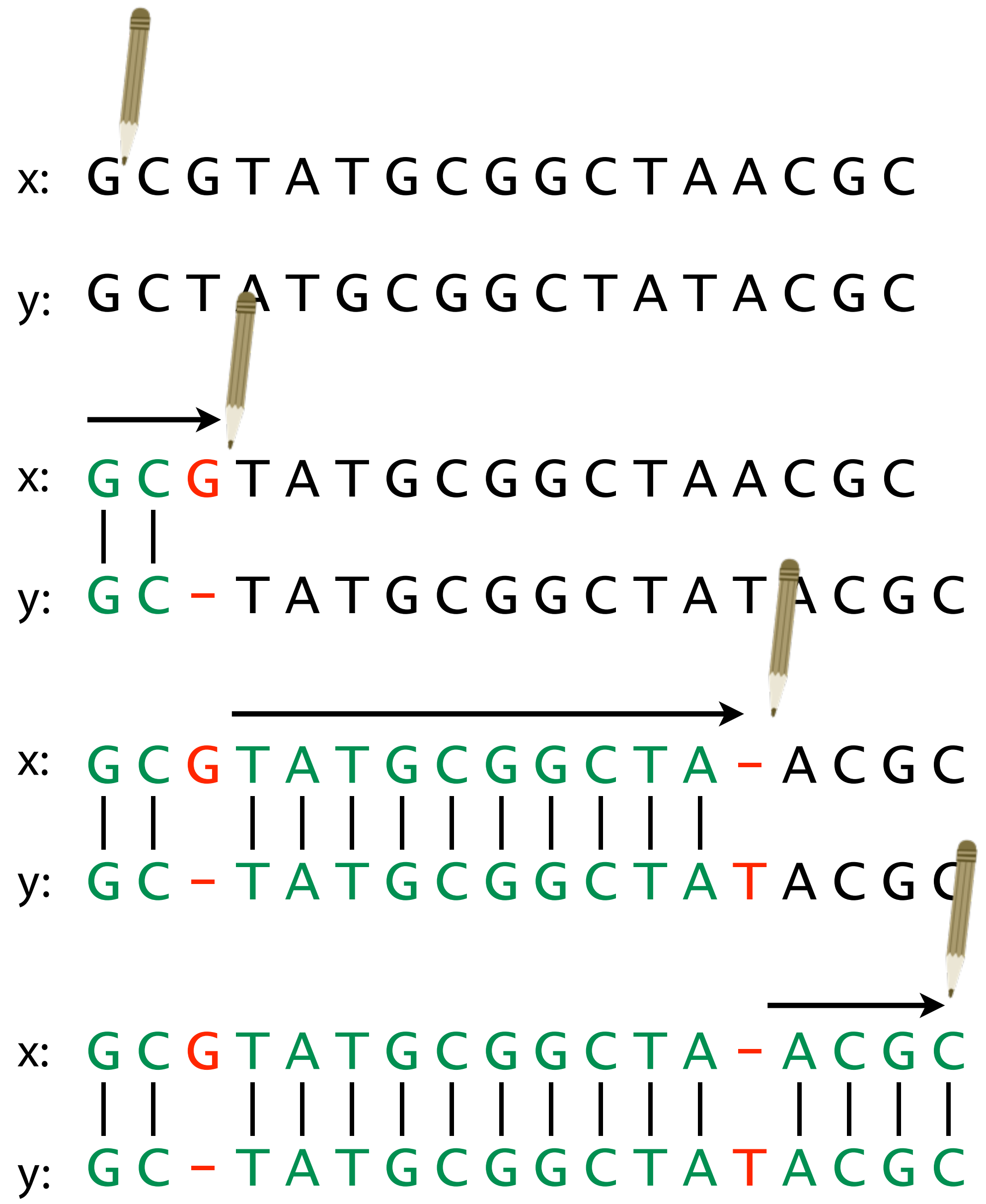
`editDistance`(*x*, *y*) ≥ $\left| |x| - |y| \right|$

Python example: http://bit.ly/CG_DP_EditDist

JOHNS HOPKINS
WHITING SCHOOL
of ENGINEERING

Edit distance

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:  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 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 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 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
```

Operations:
M = match, **R** = replace,
I = insert into *x*, **D** = delete from *x*

MMD

MMDMMMMMMMMMMI

MMDMMMMMMMMMMIMMMM

Johns Hopkins
Whiting School
of Engineering

Edit distance

Alignments:

Edit transcripts with respect to *x*:

```
x: GCGTATGCGGCTA-ACGC
   || ||||||||||| ||||
y: GC-TATGCGGCTATACGC
```

MMDMMMMMMMMMMIMMMM   Distance = 2

```
x: GCGTATGAGGCTA-ACGC
   || |||| ||||| ||||
y: GC-TATGCGGCTATACGC
```

MMDMMMMRMMMMMIMMMM   Distance = 3

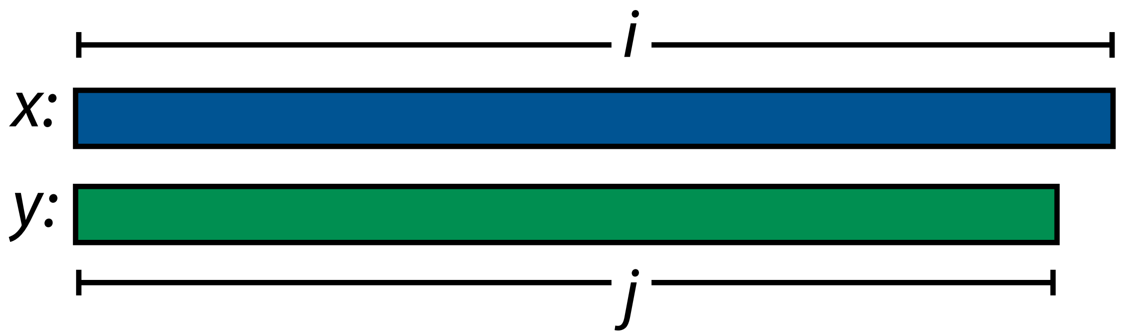
```
x: the longest----
       |||||||
y: ----longest day
```

DDDDMMMMMMMIIII   Distance = 8

JOHNS HOPKINS
WHITING SCHOOL
of ENGINEERING

Edit distance

D[*i*, *j*]: edit distance between length-*i* prefix of *x* and length-*j* prefix of *y*



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

Edit distance

Let $D[0, j] = j$, and let $D[i, 0] = i$

$$\text{Otherwise, let } D[i, j] = \min \begin{cases} D[i-1, j] + 1 & \swarrow \textbf{D} \\ D[i, j-1] + 1 & \swarrow \textbf{I} \\ D[i-1, j-1] + \delta(x[i-1], y[j-1]) & \swarrow \textbf{M} \text{ or } \textbf{R} \end{cases}$$

$\delta(a, b)$ is 0 if $a = b$, 1 otherwise

Edit distance

Let $D[0,j] = j$, and let $D[i,0] = i$

$$\text{Otherwise, let } D[i,j] = \min \begin{cases} D[i-1,j]+1 \\ D[i,j-1]+1 \\ D[i-1,j-1]+\delta(x[i-1],y[j-1]) \end{cases}$$

$\delta(a,b)$ is 0 if $a = b$, 1 otherwise

A simple recursive algorithm:

prefixes of *x* and *y* currently under consideration

```
def edDistRecursive(x, y):
    if len(x) == 0: return len(y)
    if len(y) == 0: return len(x)
    delt = 1 if x[-1] != y[-1] else 0
    diag = edDistRecursive(x[:-1], y[:-1]) + delt
    vert = edDistRecursive(x[:-1], y) + 1
    horz = edDistRecursive(x, y[:-1]) + 1
    return min(diag, vert, horz)
```

Recursively solve smaller problems

Python example: http://bit.ly/CG_DP_EditDist

Edit distance

```python
def edDistRecursive(x, y):
    if len(x) == 0: return len(y)
    if len(y) == 0: return len(x)
    delt = 1 if x[-1] != y[-1] else 0
    diag = edDistRecursive(x[:-1], y[:-1]) + delt
    vert = edDistRecursive(x[:-1], y) + 1
    horz = edDistRecursive(x, y[:-1]) + 1
    return min(diag, vert, horz)
```

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

Simple, but takes >30 seconds for a small problem

JOHNS HOPKINS
WHITING SCHOOL
of ENGINEERING

Edit distance: dynamic programming

Subproblems (D[*i*, *j*]s) can be reused instead of being recalculated:

```
def edDistRecursive(x, y):
    if len(x) == 0: return len(y)
    if len(y) == 0: return len(x)
    delt = 1 if x[-1] != y[-1] else 0
    diag = edDistRecursive(x[:-1], y[:-1]) + delt
    vert = edDistRecursive(x[:-1], y) + 1
    horz = edDistRecursive(x, y[:-1]) + 1
    return min(diag, vert, horz)
```

Reusing solutions to subproblems is *memoization:*

Return memoized answer, if avaialable $\rightarrow$

Memoize D[*i*, *j*] $\rightarrow$

```
def edDistRecursiveMemo(x, y, memo=None):
    if memo is None: memo = {}
    if len(x) == 0: return len(y)
    if len(y) == 0: return len(x)
    if (len(x), len(y)) in memo:
        return memo[(len(x), len(y))]
    delt = 1 if x[-1] != y[-1] else 0
    diag = edDistRecursiveMemo(x[:-1], y[:-1], memo) + delt
    vert = edDistRecursiveMemo(x[:-1], y, memo) + 1
    horz = edDistRecursiveMemo(x, y[:-1], memo) + 1
    ans = min(diag, vert, horz)
    memo[(len(x), len(y))] = ans
    return ans
```

Python example: http://bit.ly/CG_DP_EditDist

Edit distance: dynamic programming

```python
def edDistRecursiveMemo(x, y, memo=None):
    if memo is None: memo = {}
    if len(x) == 0: return len(y)
    if len(y) == 0: return len(x)
    if (len(x), len(y)) in memo:
        return memo[(len(x), len(y))]
    delt = 1 if x[-1] != y[-1] else 0
    diag = edDistRecursiveMemo(x[:-1], y[:-1], memo) + delt
    vert = edDistRecursiveMemo(x[:-1], y, memo) + 1
    horz = edDistRecursiveMemo(x, y[:-1], memo) + 1
    ans = min(diag, vert, horz)
    memo[(len(x), len(y))] = ans
    return ans
```

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

Much better

Edit distance: dynamic programming

edDistRecursiveMemo is a *top-down* dynamic programming approach

Alternative is *bottom-up*. Here, bottom-up recursion is pretty intuitive and interpretable, so this is how edit distance algorithm is usually explained.

Fills in a table (matrix) of D(*i*, *j*)s:

```
import numpy
```

$\longleftarrow$ numpy: package for matrices, etc

```
def edDistDp(x, y):
    """ Calculate edit distance between sequences x and y using
        matrix dynamic programming.  Return distance. """
    D = numpy.zeros((len(x)+1, len(y)+1), dtype=int)
    D[0, 1:] = range(1, len(y)+1)
    D[1:, 0] = range(1, len(x)+1)
    for i in xrange(1, len(x)+1):
        for j in xrange(1, len(y)+1):
            delt = 1 if x[i-1] != y[j-1] else 0
            D[i, j] = min(D[i-1, j-1]+delt, D[i-1, j]+1, D[i, j-1]+1)
    return D[len(x), len(y)]
```

Fill 1st row, col

Fill rest of matrix

Edit distance: dynamic programming

ϵ is empty string

y

D: *x*

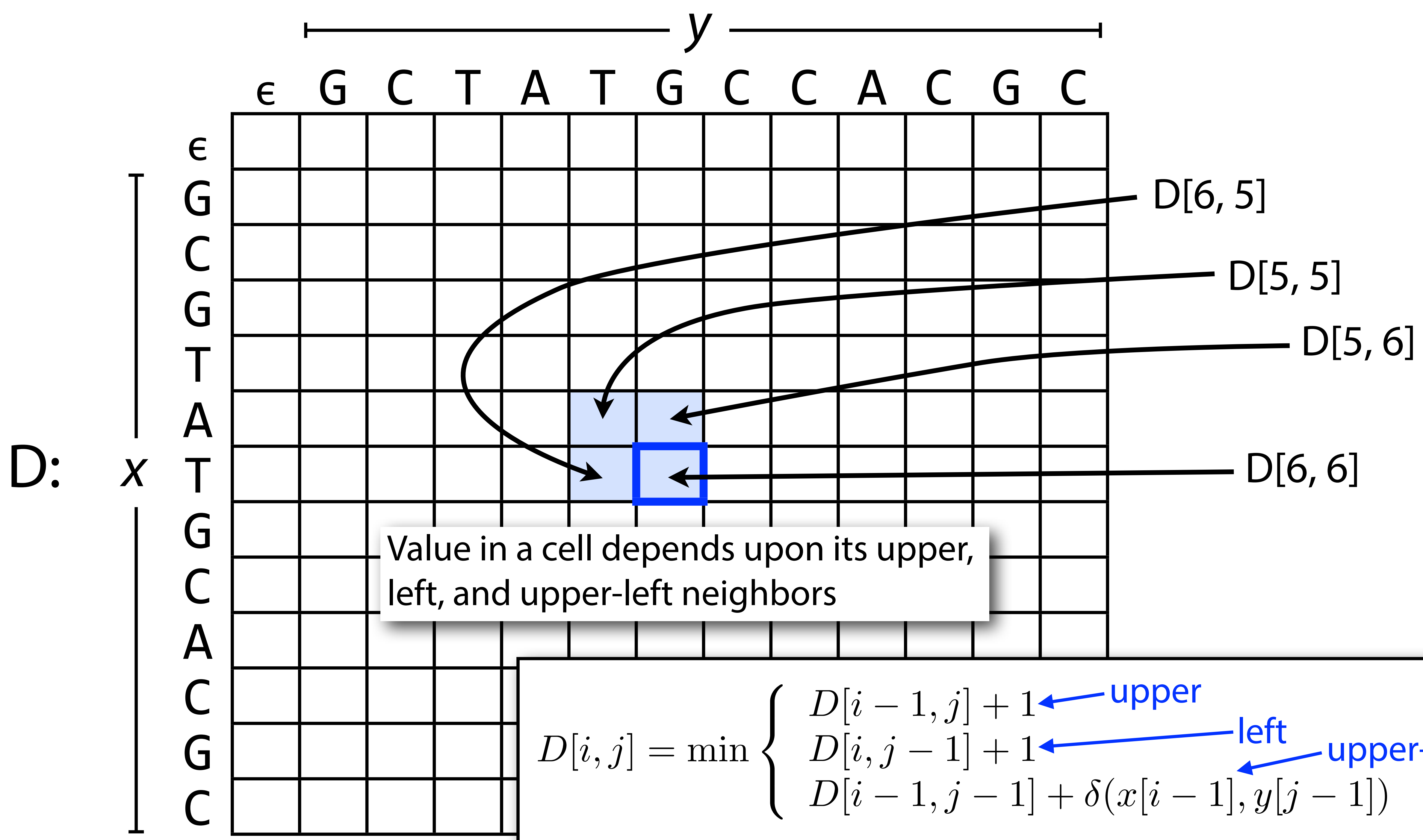
	ϵ	G	C	T	A	T	G	C	C	A	C	G	C
ϵ													
G													
C													
G													
T													
A													
T													
G													
C													
A													
C													
G													
C													

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

Edit distance: dynamic programming



Value in a cell depends upon its upper, left, and upper-left neighbors

$$D[i,j] = \min \begin{cases} D[i-1,j]+1 & \leftarrow \text{upper} \\ D[i,j-1]+1 & \leftarrow \text{left} \\ D[i-1,j-1]+\delta(x[i-1],y[j-1]) & \leftarrow \text{upper-left} \end{cases}$$

Edit distance: dynamic programming

First few lines of **edDistDp**:

```
D = numpy.zeros((len(x)+1, len(y)+1), dtype=int)
D[0, 1:] = range(1, len(y)+1)
D[1:, 0] = range(1, len(x)+1)
```

	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												

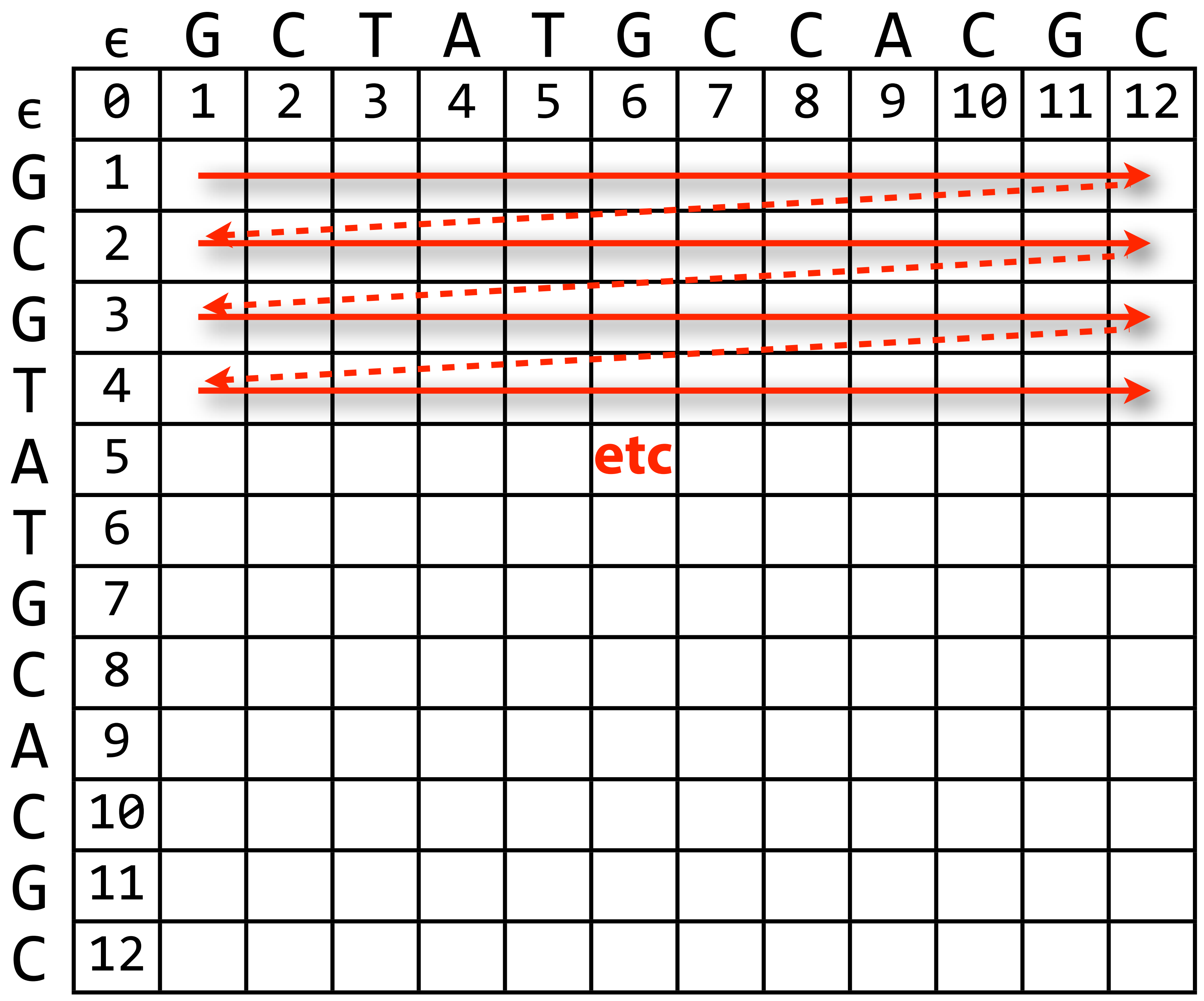
Initialize D[0, *j*] to *j*, D[*i*, 0] to *i*

JOHNS HOPKINS
WHITING SCHOOL *of* ENGINEERING

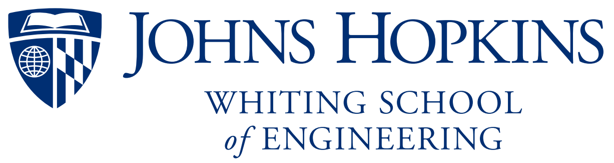
Edit distance: dynamic programming

Loop from edDistDp:

```
for i in xrange(1, len(x)+1):
    for j in xrange(1, len(y)+1):
        delt = 1 if x[i-1] != y[j-1] else 0
        D[i, j] = min(D[i-1, j-1]+delt, D[i-1, j]+1, D[i, j-1]+1)
```



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



Edit distance: dynamic programming

Loop from **edDistDp**:

```
for i in xrange(1, len(x)+1):
    for j in xrange(1, len(y)+1):
        delt = 1 if x[i-1] != y[j-1] else 0
        D[i, j] = min(D[i-1, j-1]+delt, D[i-1, j]+1, D[i, j-1]+1)
```

	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]` = '`G`'

so `delt = 0`

```
D[i, j] = min(D[i-1, j-1]+delt,
              D[i-1, j]+1,
              D[i, j-1]+1)
        = min(0 + 0, 1 + 1, 1 + 1)
        = 0
```

JOHNS HOPKINS
WHITING SCHOOL
of ENGINEERING

Edit distance: dynamic programming

Loop from
edDistDp:

```
for i in xrange(1, len(x)+1):
    for j in xrange(1, len(y)+1):
        delt = 1 if x[i-1] != y[j-1] else 0
        D[i, j] = min(D[i-1, j-1]+delt, D[i-1, j]+1, D[i, j-1]+1)
```

	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

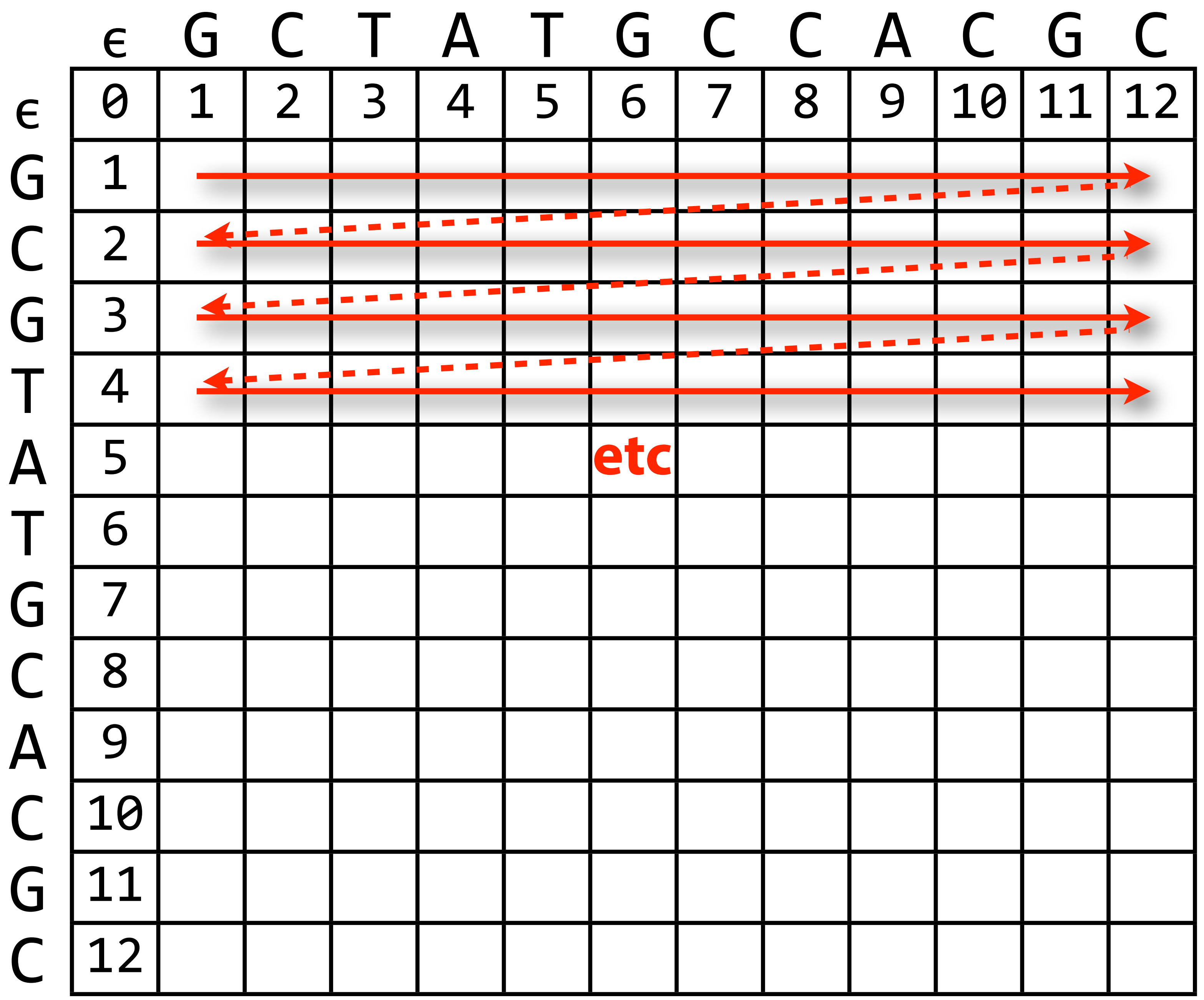
Edit distance for x, y

JOHNS HOPKINS
WHITING SCHOOL
of ENGINEERING

Edit distance: dynamic programming

Loop from `edDistDp`:

```
for i in xrange(1, len(x)+1):
    for j in xrange(1, len(y)+1):
        delt = 1 if x[i-1] != y[j-1] else 0
        D[i, j] = min(D[i-1, j-1]+delt, D[i-1, j]+1, D[i, j-1]+1)
```

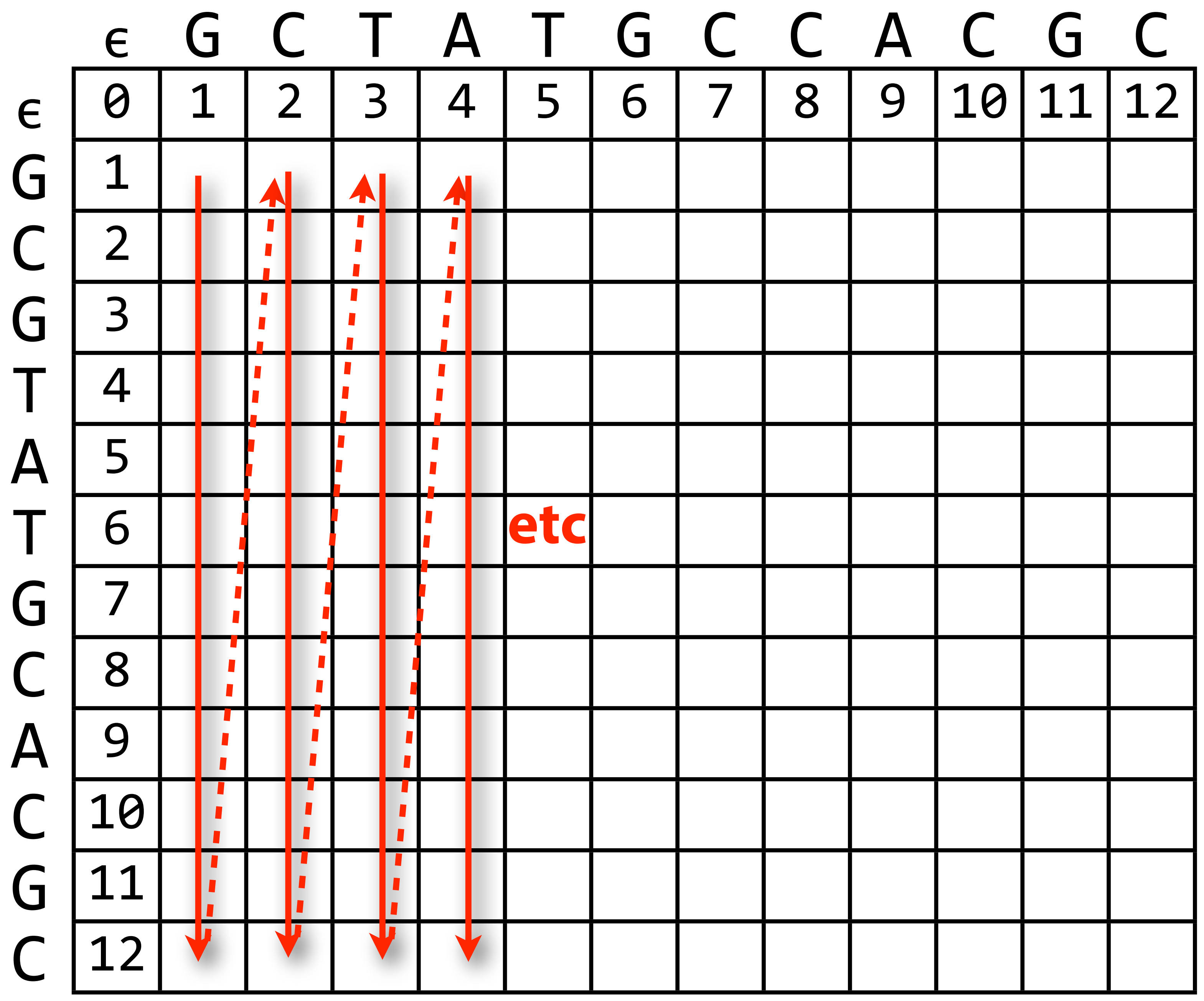


Could we have filled the cells in a different order?

Edit distance: dynamic programming

Switched

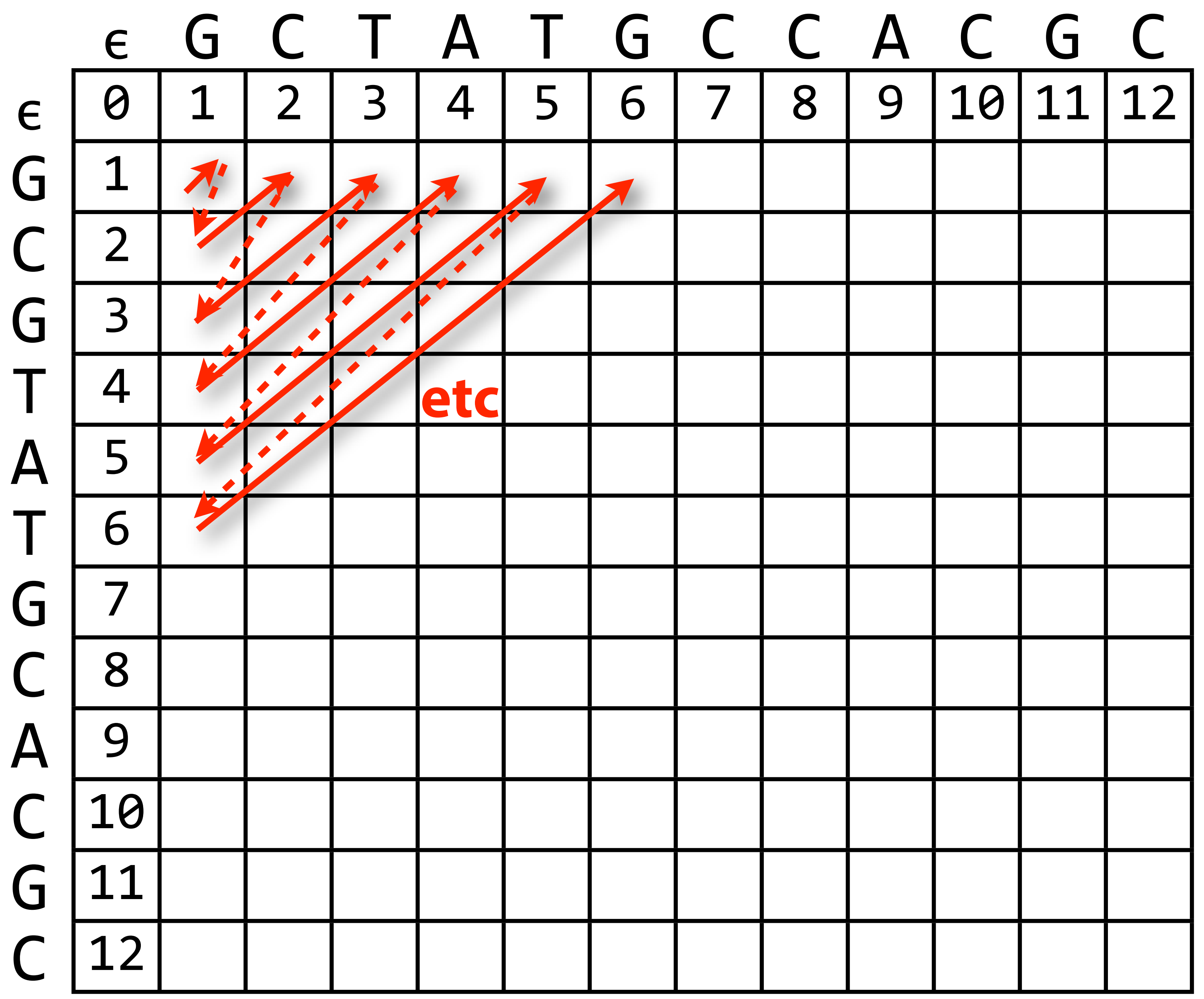
```
for j in xrange(1, len(y)+1):
    for i in xrange(1, len(x)+1):
        delt = 1 if x[i-1] != y[j-1] else 0
        D[i, j] = min(D[i-1, j-1]+delt, D[i-1, j]+1, D[i, j-1]+1)
```



	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					etc							
G	7												
C	8												
A	9												
C	10												
G	11												
C	12												

Yes: e.g. invert the loops

Edit distance: dynamic programming

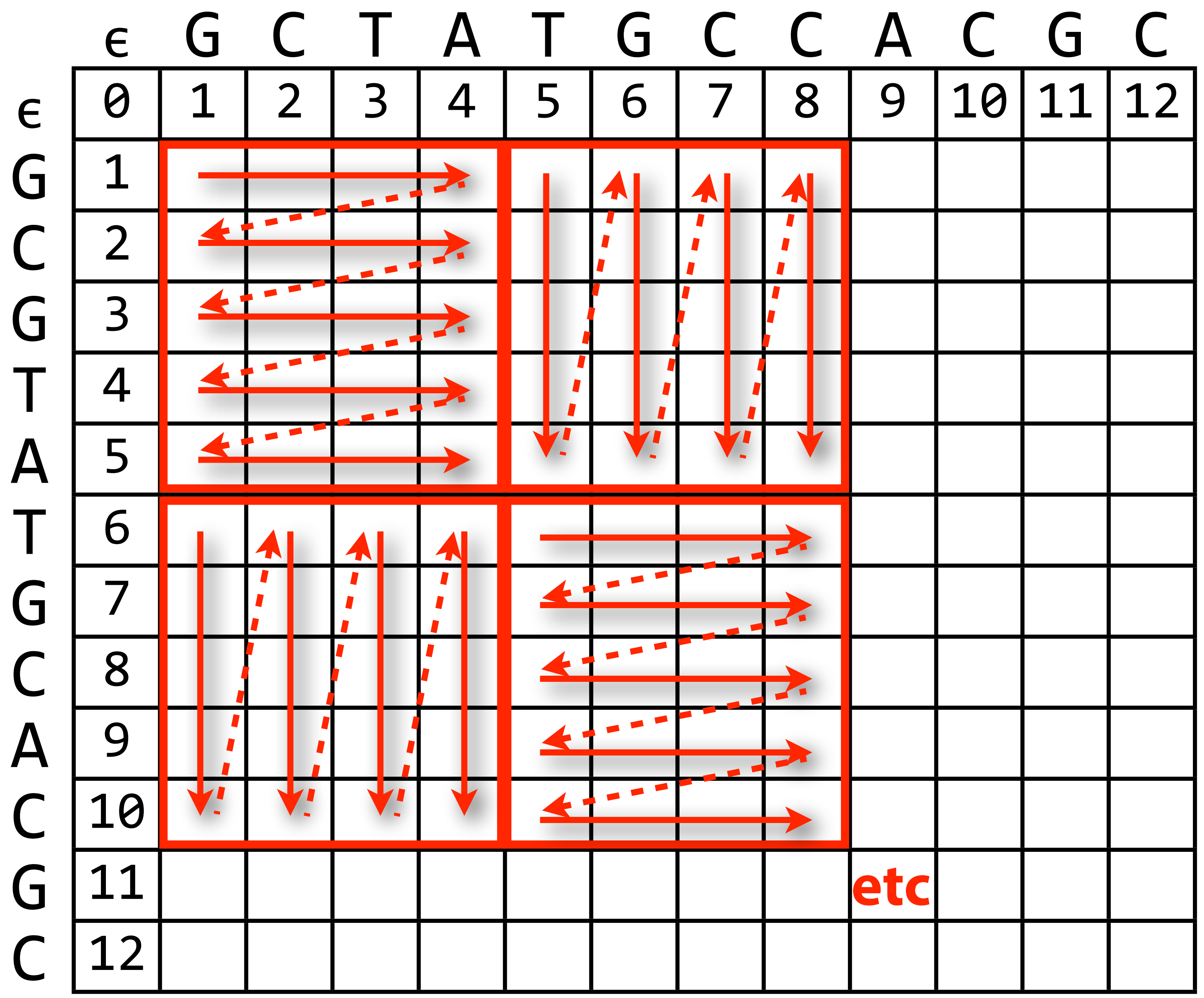


	ϵ	G	C	T	A	T	G	C	C	A	C	G	C
ϵ	0	1	2	3	4	5	6	7	8	9	10	11	12
G	1												
C	2												
G	3												
T	4												
A	5												
T	6												
G	7												
C	8												
A	9												
C	10												
G	11												
C	12												

Or by anti-diagonal

JOHNS HOPKINS
WHITING SCHOOL *of* ENGINEERING

Edit distance: dynamic programming



Or blocked

Edit distance: getting the alignment

Full backtrace path corresponds to an optimal alignment / edit transcript:

Start at 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?

Edit distance: getting the alignment

Full backtrace path corresponds to an optimal alignment / edit transcript:

Start at 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

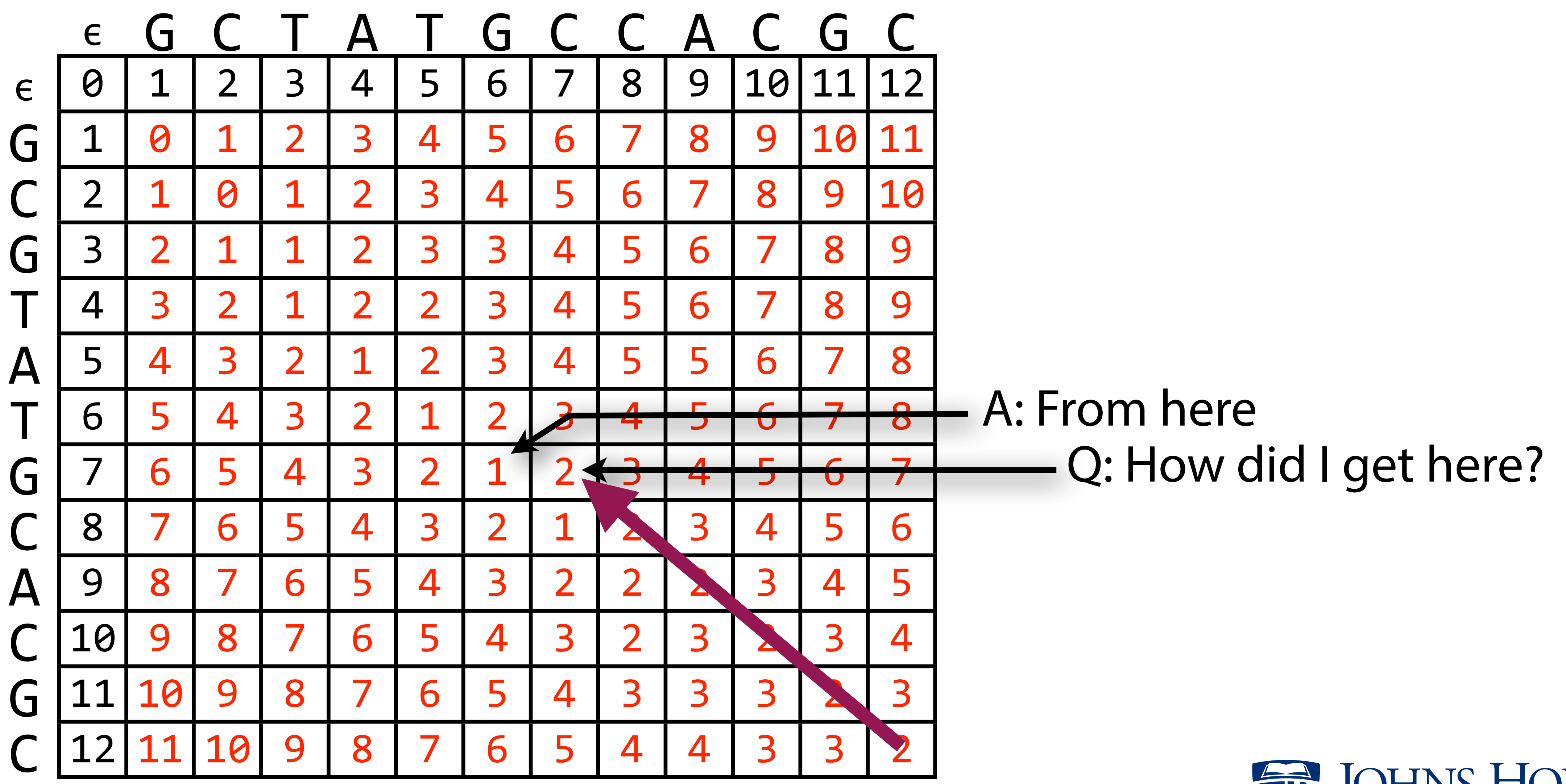
A: From here

Q: How did I get here?

Edit distance: getting the alignment

Full backtrace path corresponds to an optimal alignment / edit transcript:

Start at end; at each step, ask: which predecessor gave the minimum?

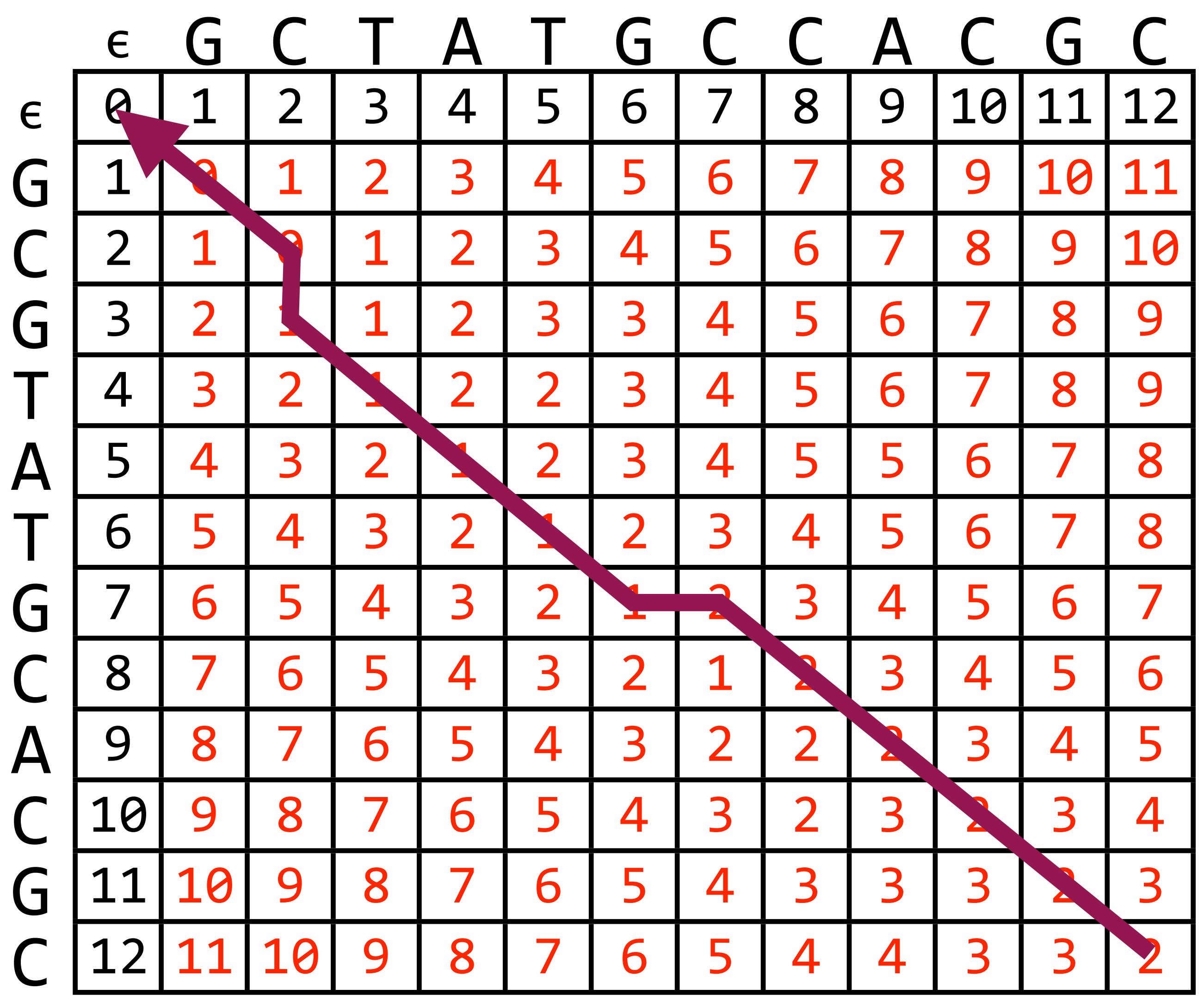


	ϵ	G	C	T	A	T	G	C	C	A	C	G	C
ϵ	0	1	2	3	4	5	6	7	8	9	10	11	12
G	1	0	1	2	3	4	5	6	7	8	9	10	11
C	2	1	0	1	2	3	4	5	6	7	8	9	10
G	3	2	1	1	2	3	3	4	5	6	7	8	9
T	4	3	2	1	2	2	3	4	5	6	7	8	9
A	5	4	3	2	1	2	3	4	5	5	6	7	8
T	6	5	4	3	2	1	2	3	4	5	6	7	8
G	7	6	5	4	3	2	1	2	3	4	5	6	7
C	8	7	6	5	4	3	2	1	2	3	4	5	6
A	9	8	7	6	5	4	3	2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

Edit distance: getting the alignment

Full backtrace path corresponds to an optimal alignment / edit transcript:

Start at 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

Alignment:

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

Edit transcript:

MMDMMMMIMMMMM

Edit distance: 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

Levenshtein

Basic edit distance

- Works for different-length sequences
- Uniform cost for substitution/insertion/deletion

```
def edit(x, y):
    D = np.zeros((len(x) + 1, len(y) + 1), dtype=int)
    # for the empty word, costs match the length of
    # the other string
    D[0, 1:] = range(1, len(y) + 1)
    D[1:, 0] = range(1, len(x) + 1)

    for i in range(1, len(x) + 1):
        for j in range(1, len(y) + 1):
            delta = 1 if x[i-1] != y[j-1] else 0
            D[i, j] = min(
                D[i-1, j] + 1,
                D[i, j-1] + 1,
                D[i-1, j-1] + delta
            )

    return D[len(x), len(y)]
```

Levenshtein

Custom Cost

```
def edit(x, y, cost={'m': 0, 's': 1, 'i': 1, 'd': 1}):
    D = np.zeros((len(x) + 1, len(y) + 1), dtype=int)
    # for the empty word, costs match the length of the
    # other string
    D[0, 1:] = range(1, len(y) + 1)
    D[1:, 0] = range(1, len(x) + 1)
    for i in range(1, len(x) + 1):
        for j in range(1, len(y) + 1):
            delta = cost['m'] if x[i-1] == y[j-1] else cost['s']
            D[i, j] = min(
                D[i-1, j] + cost['d'],
                D[i, j-1] + cost['i'],
                D[i-1, j-1] + delta
            )
    return D[len(x), len(y)]
```

Needleman-Wunsch Algorithm

- Biology: not all edits “cost the same”, gaps typically treated separately
- Gap penalty (eg. -1)
- Similarity (match reward, mismatch penalty)
- DP finds maximum similarity

	A	G	C	T
A	10	−1	−3	−4
G	−1	7	−5	−3
C	−3	−5	9	0
T	−4	−3	0	8

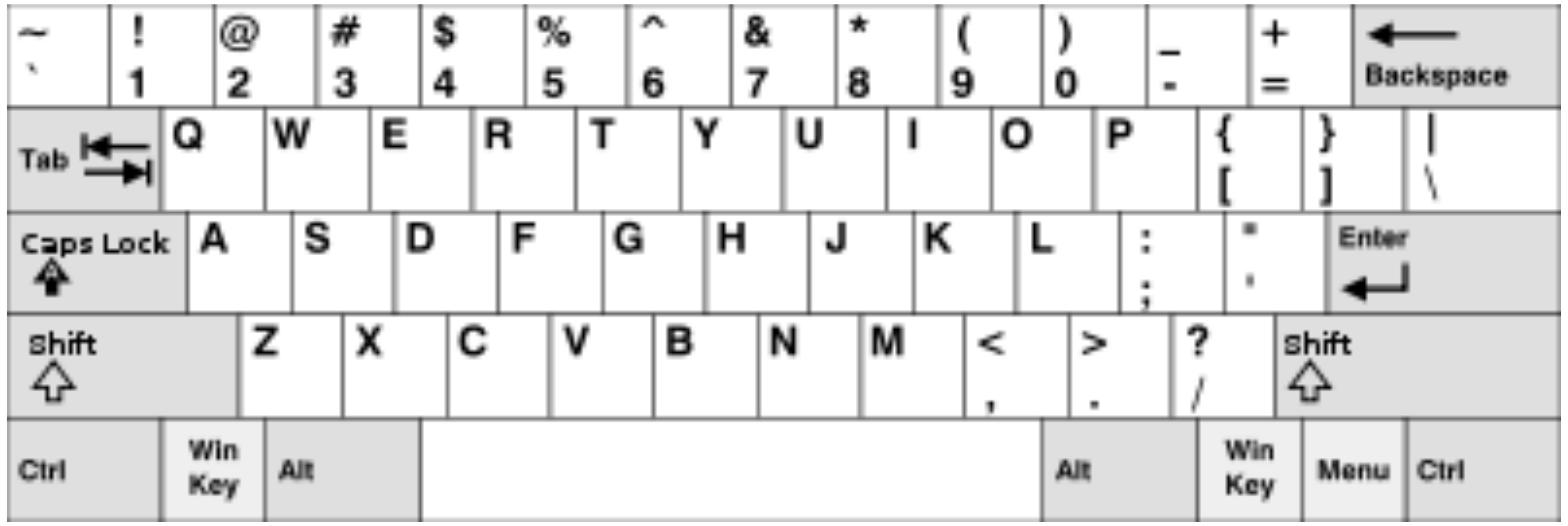
https://commons.wikimedia.org/wiki/File:KB_United_States.svg#/media/File:KB_United_States.svg

Needleman-Wunsch Algorithm

```python
def nw(x, y, d, sim):
    D = np.zeros((len(x) + 1, len(y) + 1), dtype=int)

    # for the empty word, costs match the length of the other string
    D[0, 1:] = range(1, len(y) + 1); D[0, 1:] *= d
    D[1:, 0] = range(1, len(x) + 1); D[1:, 0] *= d

    for i in range(1, len(x)):
        for j in range(1, len(y)):
            cs = D[i-1, j-1] + sim(x[i], y[j])
            cd = D[i-1, j] + d
            ci = D[i, j-1] + d
            D[i,j] = max(cs, cd, ci)

    print(D)
    return D[len(x)][len(y)]
```

Keyboard-aware Substitutions?

- Compute cost of substitution by proximity of keys
- Map keys to grid, compute euclidean distance
- What about g <> h etc.?



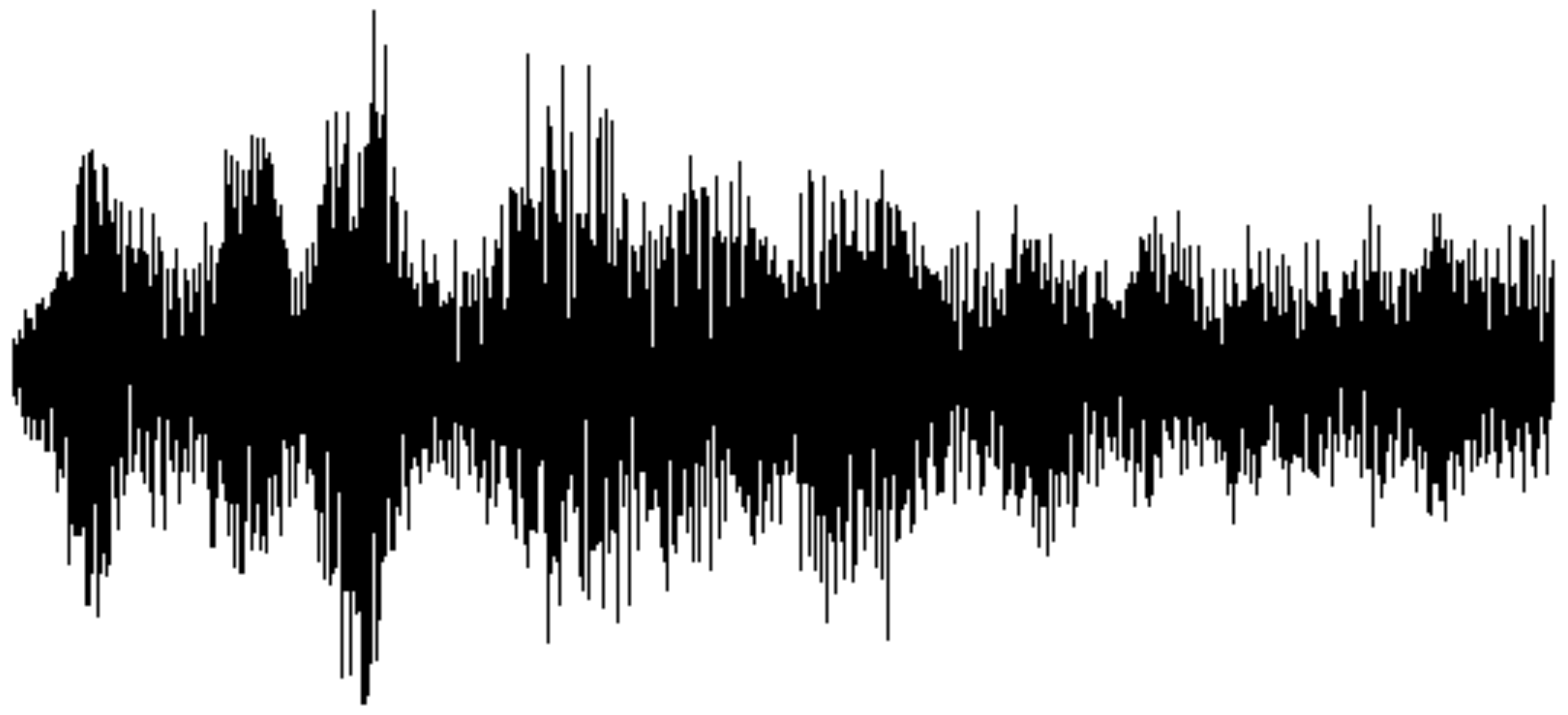
https://commons.wikimedia.org/wiki/File:KB_United_States.svg#/media/File:KB_United_States.svg

Outlook: Damerau-Levenshtein with adjacent transpositions

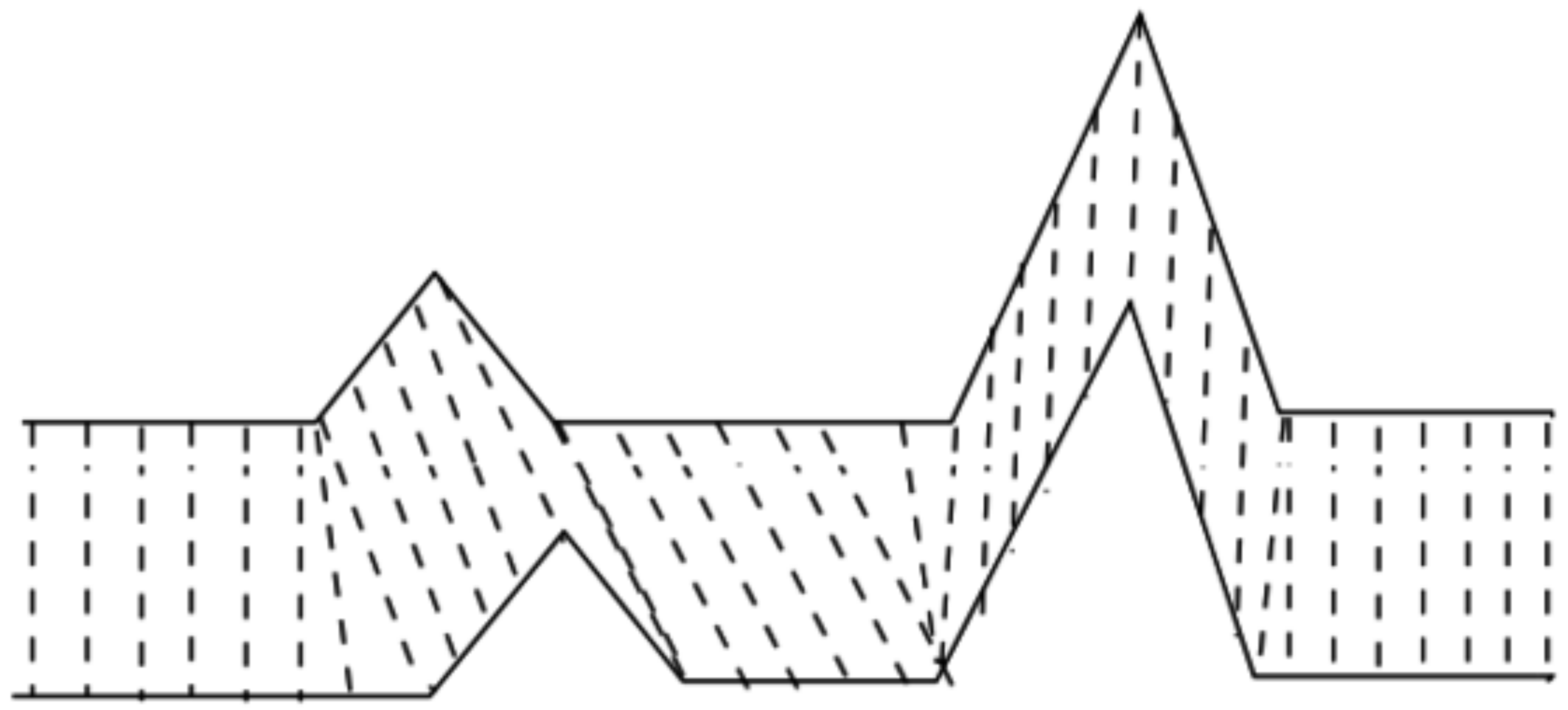
Comparing Sequences

…of non-discrete signals?

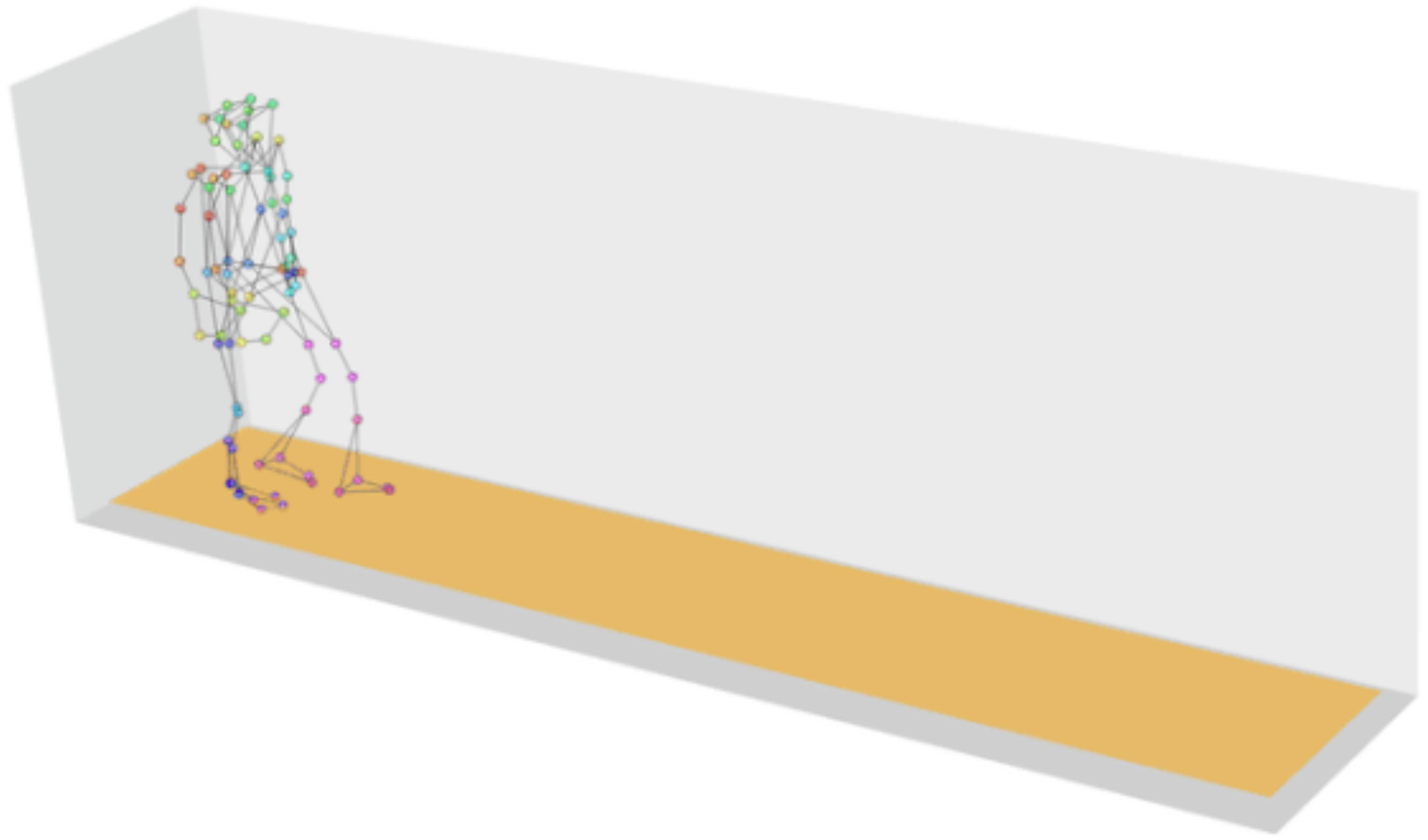
vs.



Dynamic Time Warping



https://en.wikipedia.org/wiki/File:Dynamic_time_warping.png



https://en.wikipedia.org/wiki/File:Two_repetitions_of_a_walking_sequence_of_an_individual_recorded_using_a_motion-capture_system.gif

Dynamic Time Warping

- Sequences assumed to be similar, no notion of insert
  —> first row/col is *inf*!
- Observations are continuous (not drawn from vocab)
  - There is *always* a cost
  - No explicit modeling of insertion/deletion/substitution

Dynamic Time Warping

```python
def dtw(x: list, y: list, d) -> float:
    D = np.full((len(x) + 1, len(y) + 1), np.inf, dtype=float)
    D[0, 0] = 0

    for i in range(1, len(x)):
        for j in range(1, len(y)):
            cost = d(x[i], y[j])
            D[i, j] = cost + min(D[i-1, j],
                                 D[i, j-1],
                                 D[i-1, j-1])

    return D[len(x)][len(y)]
```

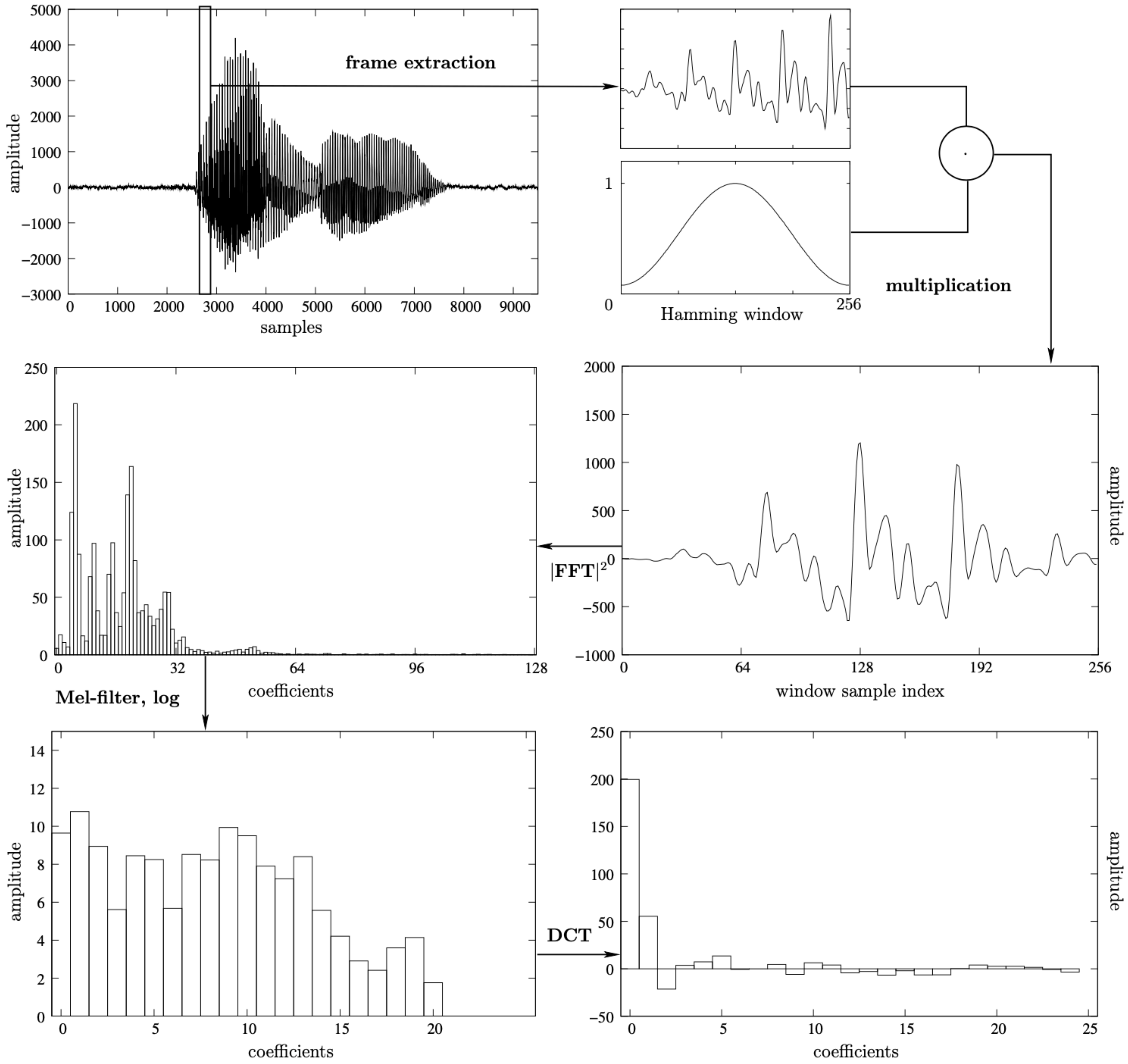
Dynamic Time Warping

On audio data

- Raw sample data is way to numerous
- Compute spectral (cepstral) features, eg. MFCC
- Use euclidean distance on feature vectors

MFCC

pipeline

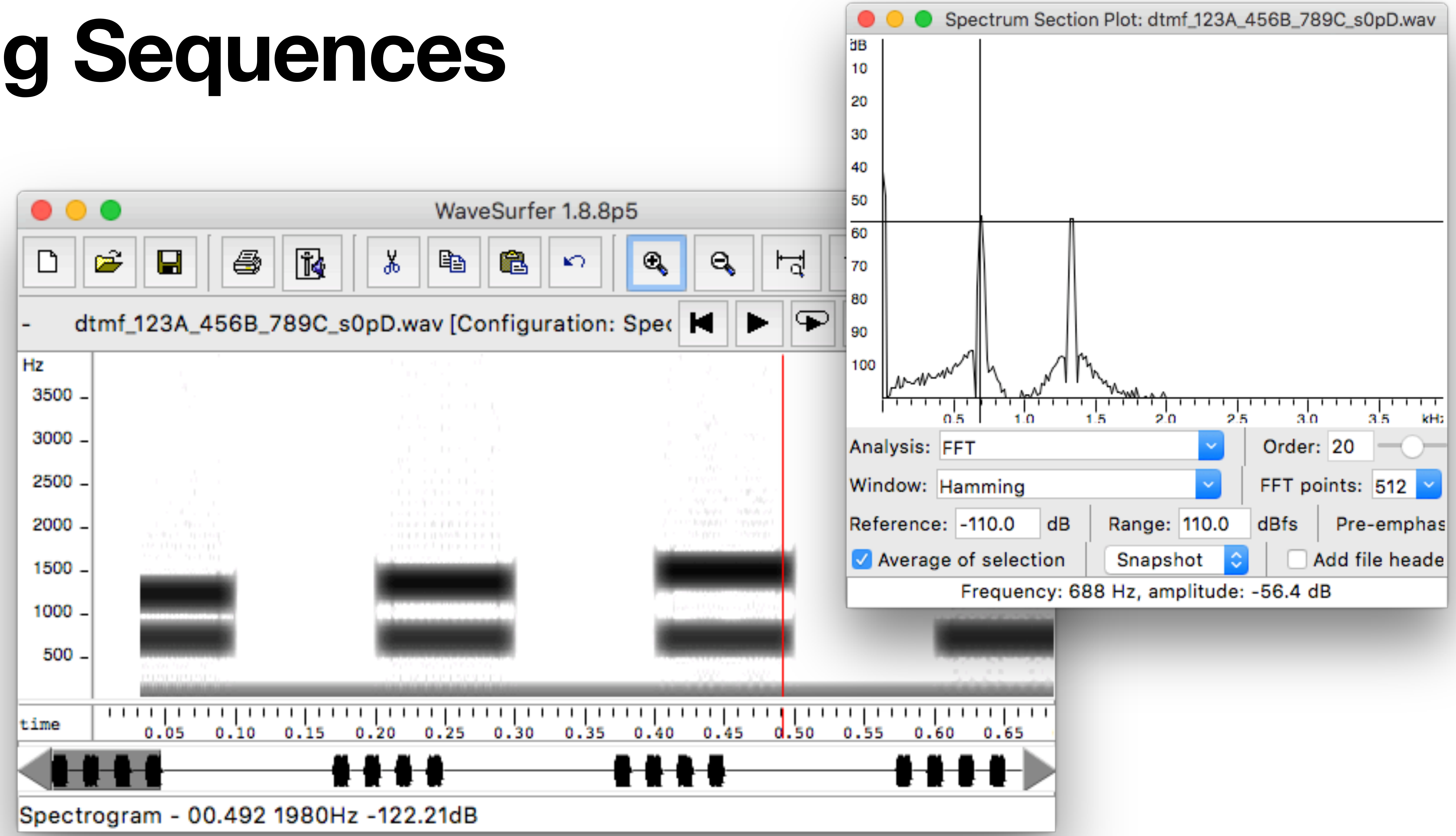


Multi-class Sequence Classification

Eg. Isolated word recognition

- Have (at least) one reference sequence per class (word)
- Compute DTW distance for test sequence to each reference
- Chose class with minimum distance
- How to speed up…?

Decoding Sequences



WaveSurfer 1.8.8p5

dtmf_123A_456B_789C_s0pD.wav [Configuration: Spec

Spectrogram - 00.492 1980Hz -122.21dB

Spectrum Section Plot: dtmf_123A_456B_789C_s0pD.wav

Analysis: FFT

Order: 20

Window: Hamming

FFT points: 512

Reference: -110.0 dB

Range: 110.0 dBfs

Pre-emphas

[✓] Average of selection

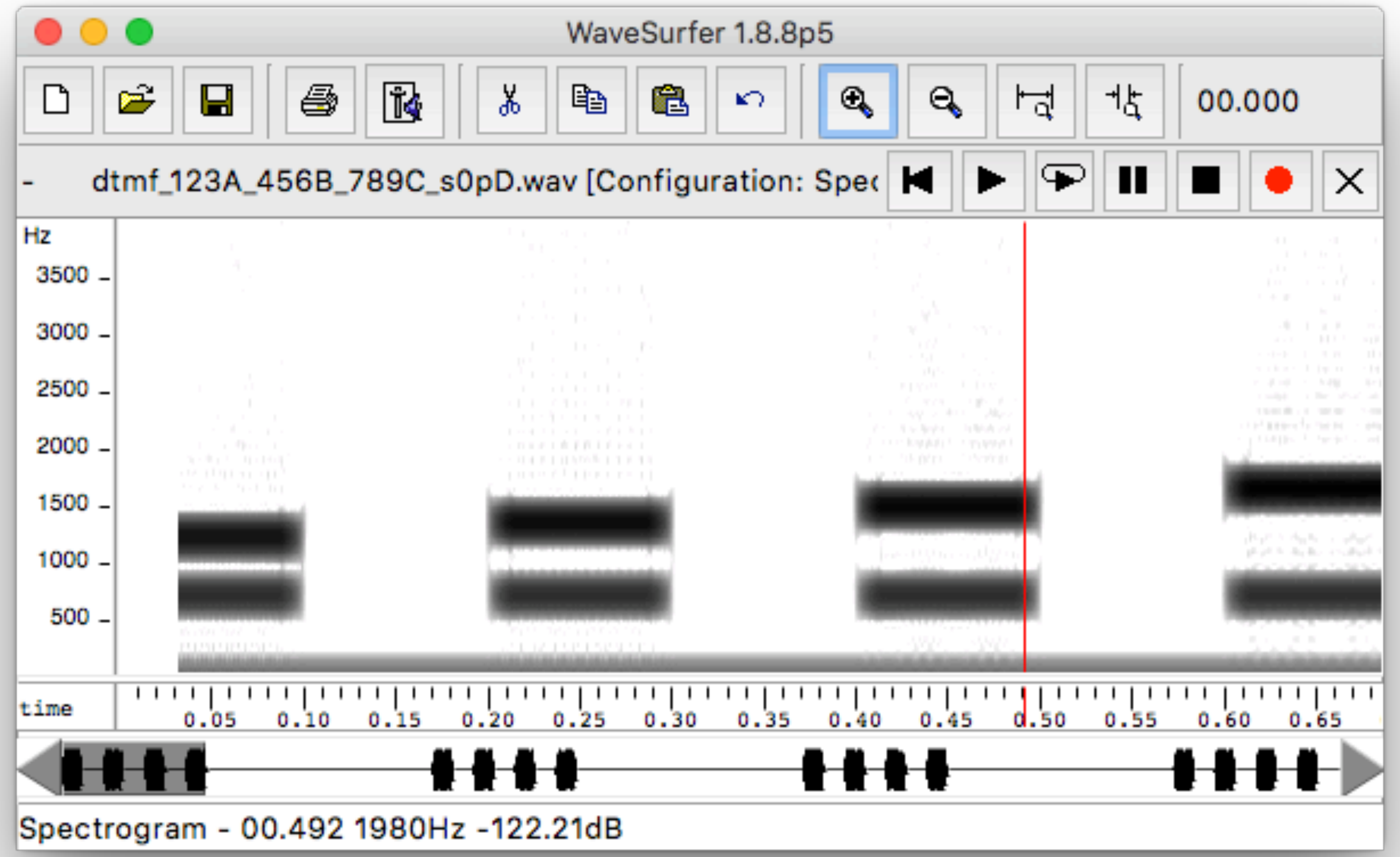
Snapshot

[ ] Add file heade

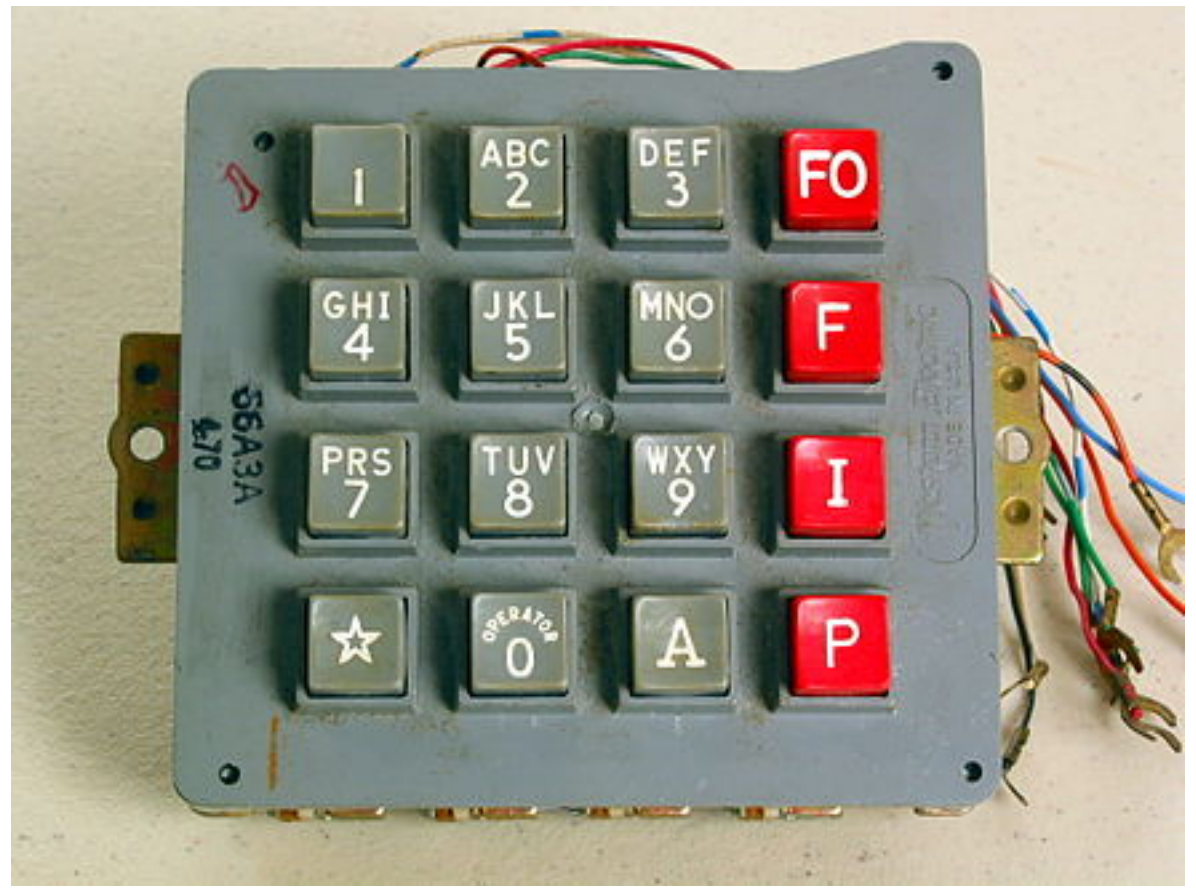
Frequency: 688 Hz, amplitude: -56.4 dB

Dual-tone multi-frequency signaling (DTMF)

Decoding States



- Observation: *pause 1 pause 2 pause …*
- If we ignore noise etc.: 12+1 classes
  - model each class, eg. with a reference vector
  - apply a sliding window, map each window to class
    - output: `– – 1 1 1 – – 2 2 2 2 – – 3 3` …
  - collapse (`uniq`)
    - output: `123`



https://en.wikipedia.org/wiki/File:66a3aDTMFpad.jpg

Assignment 1

Due April 8

- Edit distances (Hamming, Levenshtein, Needleman-Wunsch)
- Auto-Complete (using basic word stats)
- Isolated Word Recognition using DTW (on digits)
- DTMF sequence decoding using DP on states
- Submit via Teams to Files > Assignment Submissions

https://github.com/seqlrn/assignments/tree/master/1-dynamic-programming