String Alignment: Edit distance and dynamic programming

Dr. Jared Simpson
Ontario Institute for Cancer Research
&
Department of Computer Science
University of Toronto

Today's Lecture

- Solving the edit distance problem
- Global alignment between biological sequences
- Revisiting approximate pattern matching
- Local alignment
- Optimising alignment for space
- Read and discuss papers

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

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

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
```

```
GAGGTAGCGGCGTTTAAC
|  |||||  |||  |||||||
GTGGTAACGGGGTTTAAC
```

Edit distance is harder:

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

```
GCGTATGCGGCTA-ACGC
||  |||||||||||  |||
GC-TATGCGGCTATACGC
```

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*) ≥ | | *x* | - | *y* | |

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

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

Edit distance

Alignments:

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

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

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

Edit transcripts with respect to *x*:

MMDMMMMMMMMMMMIMMMM

Distance = 2

MMDMMMMRMMMMMMIMMMM

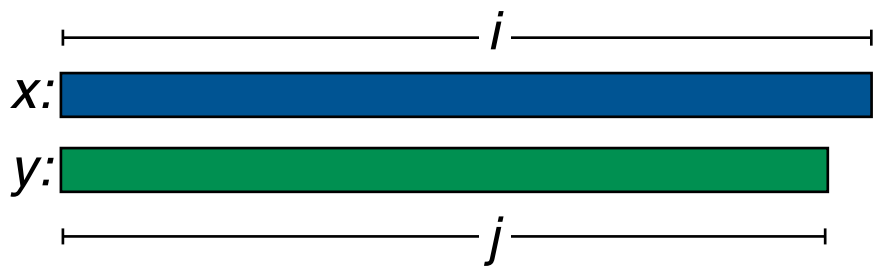
Distance = 3

DDDDMMMMMMMMIIII

Distance = 8

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 $D[i, 0] = i$

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

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

Edit distance

$$\text{Let } D[0,j] = j, \text{ and let } D[i,0] = i$$

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

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

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

```
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

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

Memoize D[*i*, *j*]

```
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

```
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

JOHNS HOPKINS
WHITING SCHOOL of ENGINEERING

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
```

`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

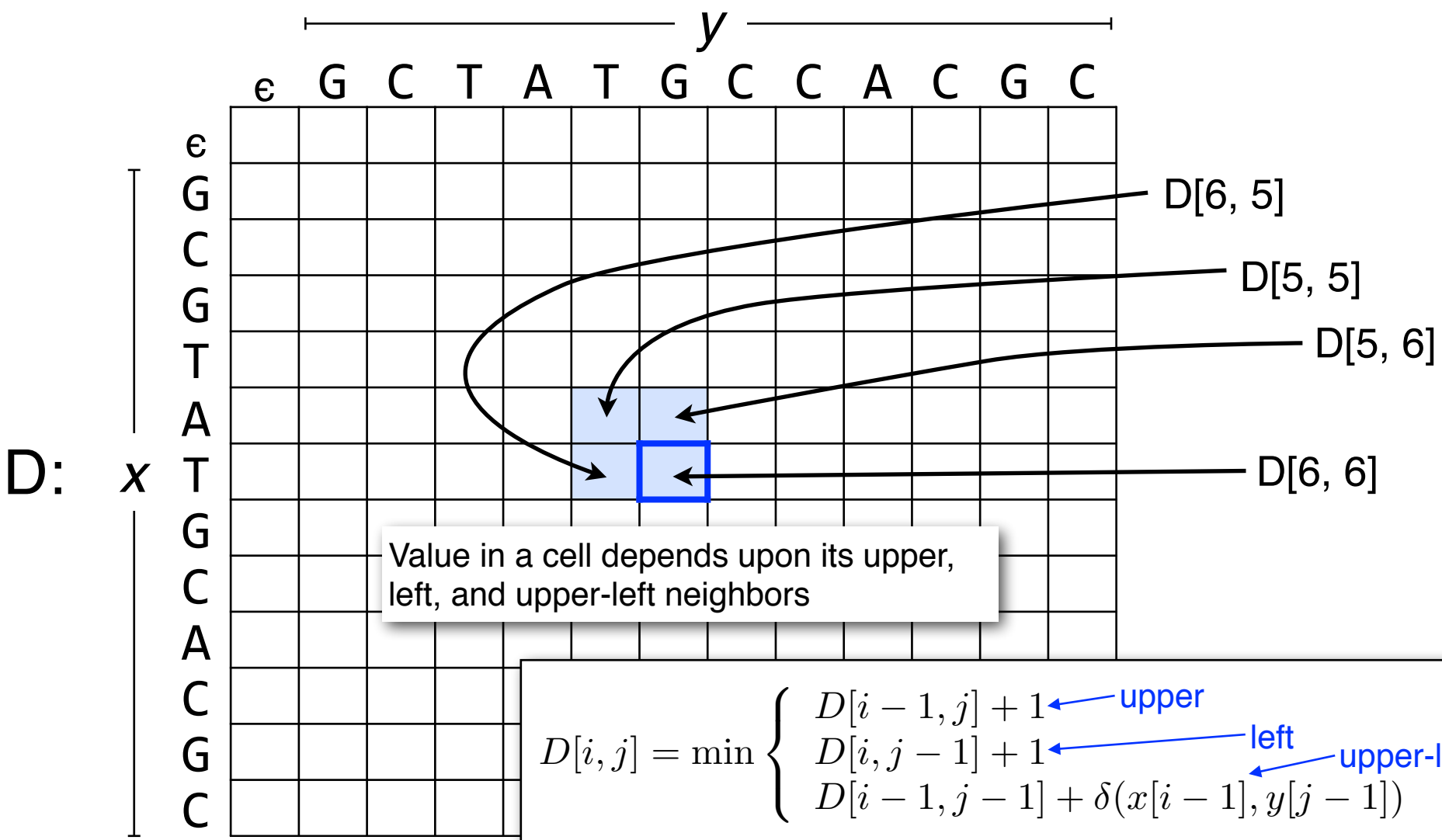
	ϵ	G	C	T	A	T	G	C	C	A	C	G	C
ϵ													
G													
C													
G													
T													
A													
T													
G													
C													
A													
C													
G													
C													

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

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

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*

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						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

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

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						etc						
T	6												
G	7												
C	8												
A	9												
C	10												
G	11												
C	12												

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

JOHNS HOPKINS
WHITING SCHOOL of ENGINEERING

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												

etc

Or by anti-diagonal

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									etc			
C	12												

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

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

Alignment:

```
GCGTATG-CACGC
||  |||||  |||||
GC-TATGCCACGC
```

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

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

Generalizing edit distance

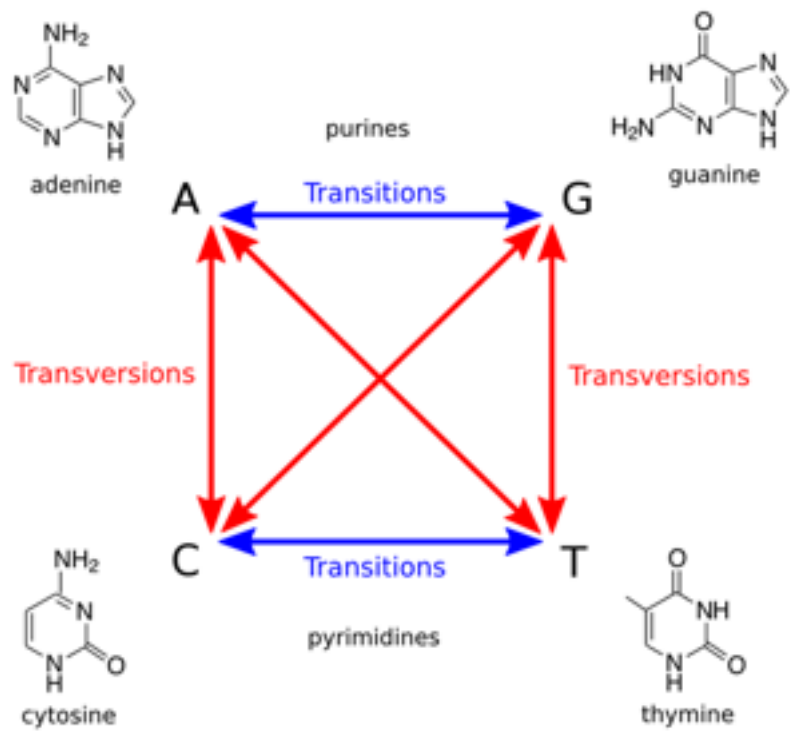
What if cost of edit could be ≠ 1?

E.g. sequencing errors tend to manifest as mismatches rather than gaps, so maybe gap penalty should be > mismatch penalty

It's also more likely for a genetic variant to be a mismatch rather than a gap

Also, some mismatches are more likely than others

Human *transition to transversion ratio* (AKA *ti/tv*) is ~2.1



http://en.wikipedia.org/wiki/Transversion

Global alignment

$s(a, b)$:

	A	C	G	T	-
A	0	4	2	4	8
C	4	0	4	2	8
G	2	4	0	4	8
T	4	2	4	0	8
-	8	8	8	8	

gaps in b

gaps in a

2	*Transitions* (A↔ G, C↔ T)
4	*Transversions* (everything else)
8	*Gaps*

Scoring function reflecting that transitions are more common than transversions and mismatches are more common than gaps

(Could have been even more specific, e.g. varying cost according to what character appears *opposite* a gap.)

Global alignment

Let $D[0,j] = \sum_{k=0}^{j-1} s(-, y[k])$, and let $D[i,0] = \sum_{k=0}^{i-1} s(x[k], -)$

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

$s(a,b)$ assigns a cost to a particular gap or substitution

$s(a,b)$:

	A	C	G	T	-
A	0	4	2	4	8
C	4	0	4	2	8
G	2	4	0	4	8
T	4	2	4	0	8
-	8	8	8	8	

2	*Transitions* (A ↔ G, C ↔ T)
4	*Transversions* (everything else)
8	*Gaps*

Global alignment: implementation

```
from numpy import zeros

def exampleCost(xc, yc):
    """ Cost function assigning 0 to match, 2 to transition, 4 to
        transversion, and 8 to a gap """
    if xc == yc: return 0 # match
    if xc == '-' or yc == '-': return 8 # gap
    minc, maxc = min(xc, yc), max(xc, yc)
    if minc == 'A' and maxc == 'G': return 2 # transition
    elif minc == 'C' and maxc == 'T': return 2 # transition
    return 4 # transversion
```

	A	C	G	T	-
A	0	4	2	4	8
C	4	0	4	2	8
G	2	4	0	4	8
T	4	2	4	0	8
-	8	8	8	8	

```
def globalAlignment(x, y, s):
    """ Calculate global alignment value of sequences x and y using
        dynamic programming.  Return global alignment value. """
    D = zeros((len(x)+1, len(y)+1), dtype=int)
    for j in xrange(1, len(y)+1):
        D[0, j] = D[0, j-1] + s('-', y[j-1])
    for i in xrange(1, len(x)+1):
        D[i, 0] = D[i-1, 0] + s(x[i-1], '-')
    for i in xrange(1, len(x)+1):
        for j in xrange(1, len(y)+1):
            D[i, j] = min(D[i-1, j-1] + s(x[i-1], y[j-1]), # diagonal
                          D[i-1, j  ] + s(x[i-1], '-'),    # vertical
                          D[i  , j-1] + s('-',    y[j-1])) # horizontal
    return D, D[len(x), len(y)]
```

Use of new cost function

Extremely similar to edit distance algorithm

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

BLOSUM Matrix for Protein Alignment

	Ala	Arg	Asn	Asp	Cys	Gln	Glu	Gly	His	Ile	Leu	Lys	Met	Phe	Pro	Ser	Thr	Trp	Tyr	Val
Ala	4																			
Arg	-1	5																		
Asn	-2	0	6																	
Asp	-2	-2	1	6																
Cys	0	-3	-3	-3	9															
Gln	-1	1	0	0	-3	5														
Glu	-1	0	0	2	-4	2	5													
Gly	0	-2	0	-1	-3	-2	-2	6												
His	-2	0	1	-1	-3	0	0	-2	8											
Ile	-1	-3	-3	-3	-1	-3	-3	-4	-3	4										
Leu	-1	-2	-3	-4	-1	-2	-3	-4	-3	2	4									
Lys	-1	2	0	-1	-3	1	1	-2	-1	-3	-2	5								
Met	-1	-1	-2	-3	-1	0	-2	-3	-2	1	2	-1	5							
Phe	-2	-3	-3	-3	-2	-3	-3	-3	-1	0	0	-3	0	6						
Pro	-1	-2	-2	-1	-3	-1	-1	-2	-2	-3	-3	-1	-2	-4	7					
Ser	1	-1	1	0	-1	0	0	0	-1	-2	-2	0	-1	-2	-1	4				
Thr	0	-1	0	-1	-1	-1	-1	-2	-2	-1	-1	-1	-1	-2	-1	1	5			
Trp	-3	-3	-4	-4	-2	-2	-3	-2	-2	-3	-2	-3	-1	1	-4	-3	-2	11		
Tyr	-2	-2	-2	-3	-2	-1	-2	-3	2	-1	-1	-2	-1	3	-3	-2	-2	2	7	
Val	0	-3	-3	-3	-1	-2	-2	-3	-3	3	1	-2	1	-1	-2	-2	0	-3	-1	4

https://en.wikipedia.org/wiki/BLOSUM

- 5 minute break now, then we'll talk about variations of these ideas

Approximate pattern matching

Can we search for the occurrence of *P* in *T* with the least edits?

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

Approximate pattern matching

Can we search for the occurrence of *P* in *T* with least edits?

First idea: initialize first row with 0's rather than increasing integers

T

		ϵ	A	A	C	C	C	T	A	T	G	T	C	A	T	G	C	C	T	T	G	G	A
	ϵ	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	T	1																					
	A	2																					
	C	3																					
	G	4																					
P	T	5																					
	C	6																					
	A	7																					
	G	8																					
	C	9																					

Approximate pattern matching

Fill in the matrix with the usual edit-distance recurrence

D[*i, j*] equals the optimal edit distance between the length-*i* prefix of *P* and a substring of *T* ending at position *j*.

T

		ϵ	A	A	C	C	C	T	A	T	G	T	C	A	T	G	C	C	T	T	G	G	A
	ϵ	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	T	1	1	1	1	1	1	0	1	0	1	0	1	1	0	1	1	1	0	0	1	1	1
	A	2	1	1	2	2	2	1	0	1	1	1	1	1	1	1	2	2	1	1	1	2	1
	C	3	2	2	1	2	2	2	1	1	2	2	1	2	2	2	1	2	2	2	2	2	2
	G	4	3	3	2	2	3	3	2	2	1	2	2	2	3	2	2	2	3	3	2	2	3
P	T	5	4	4	3	3	3	3	3	2	2	1	2	3	2	3	3	3	2	3	3	3	3
	C	6	5	5	4	3	3	4	4	3	3	2	1	2	3	3	3	3	3	3	4	4	4
	A	7	6	5	5	4	4	4	4	4	4	3	2	1	2	3	4	4	4	4	4	5	4
	G	8	7	6	6	5	5	5	5	5	4	4	3	2	2	2	3	4	5	5	4	4	5
	C	9	8	7	6	6	5	6	6	6	5	5	4	3	3	3	2	3	4	5	5	5	5

Approximate pattern matching

Second idea: Pick lowest edit distance *in the bottom row*, backtrace from there

Can also find for all occurrences of *P* in *T* with ≤ *k* edits

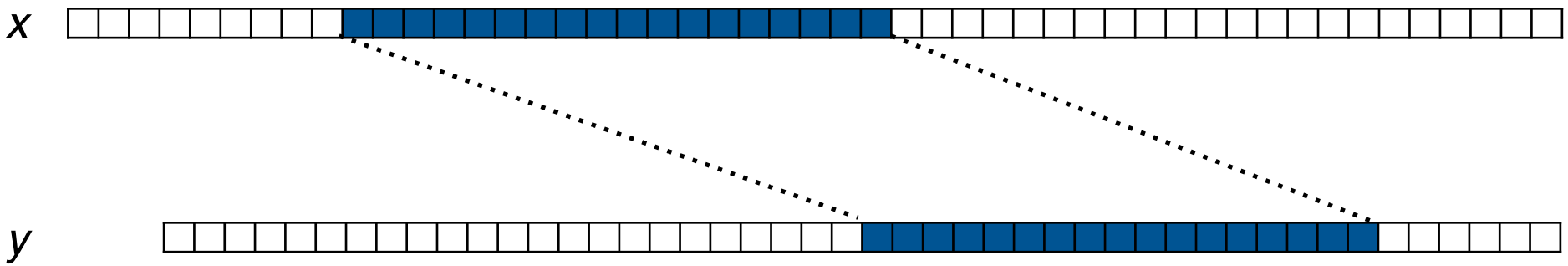
T

P	ϵ	A	A	C	C	C	T	A	T	G	T	C	A	T	G	C	C	T	T	G	G	A
ϵ	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
T	1	1	1	1	1	1	0	1	0	1	0	1	1	0	1	1	1	1	1	1	1	1
A	2	1	1	2	2	2	1	0	1	2	1	2	2	1	2	2	2	2	2	2	2	2
C	3	2	2	1	2	2	2	1	1	2	2	1	2	2	2	1	2	2	2	2	2	2
G	4	3	3	2	2	3	3	2	2	1	2	2	2	3	2	2	2	3	3	2	2	3
T	5	4	4	3	3	3	3	3	2	2	1	2	3	2	3	3	3	2	3	3	3	3
C	6	5	5	4	3	3	4	4	3	3	2	1	2	3	3	3	3	3	3	4	4	4
A	7	6	5	5	4	4	4	4	4	4	3	2	1	2	3	4	4	4	4	4	5	4
G	8	7	6	6	5	5	5	5	5	4	4	3	2	2	2	3	4	5	5	4	4	5
C	9	8	7	6	6	5	6	6	6	5	5	4	3	3	3	$\boxed{2}$	3	4	5	5	5	5

```
T:  AACCCTATGTCATGCCTTGGA
          ||  ||||  ||
P:        TACGTCA-GC
```

Local alignment

Given strings *x* and *y*, what is the optimal global alignment value of a *substring* of *x* to a *substring* of *y.* This is *local alignment.*



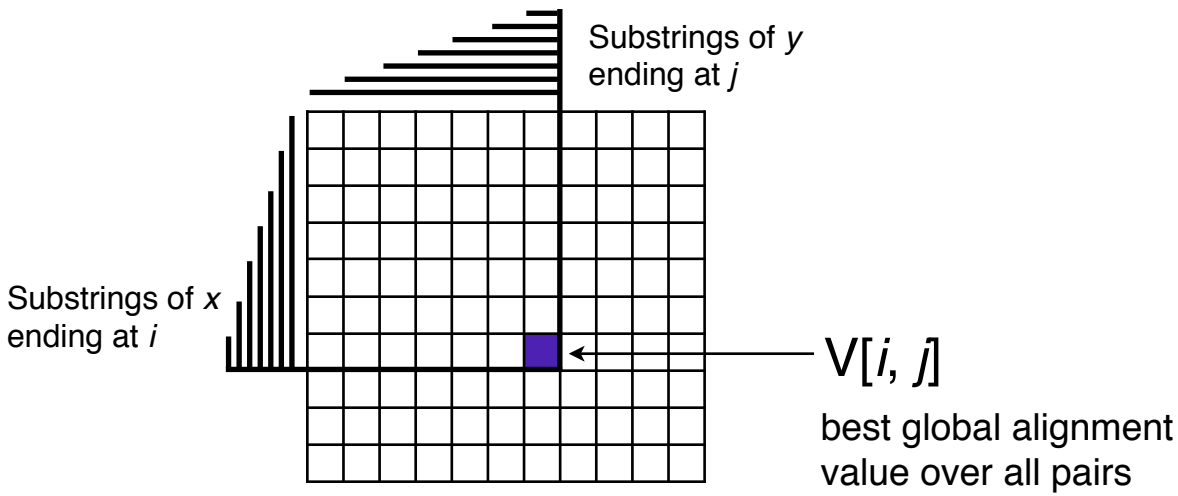
Assume scoring where: (a) similarities get > 0, (b) dissimilarities get < 0, (c) alignment of ϵ to any string has score 0

Somehow we must consider *all possible pairs* of substrings

What is bound for # substring pairs, assuming $|x| = n$, $|y| = m$? $O(m^2n^2)$

Local alignment

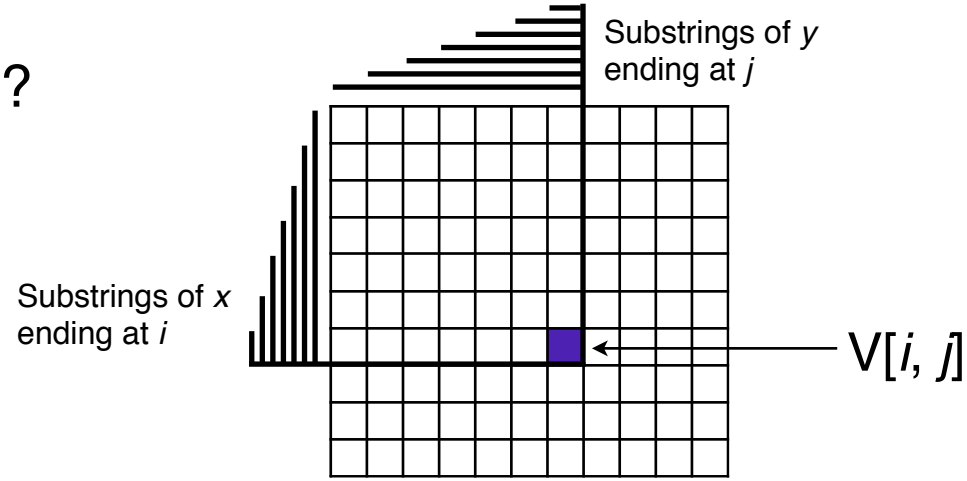
Let V[*i*, *j*] be the optimal global alignment value of a substring of *x* ending at *i* and a substring of *y* ending at *j*. The substrings may be empty.



The maximum V[*i*, *j*] over all *i*, *j* is the optimal score we're looking for

Local alignment

How to calculate V[*i*, *j*]?



Only 4 ways to build a new edit transcript from another one:

Vertical: append **I** to transcript for V[*i*-1, *j*], take gap penalty

Horizontal: append **D** to transcript for V[*i*, *j*-1], take gap penalty

Diagonal: append **M** or **R** to transcript for V[*i*-1, *j*-1], get match bonus or take replacement penalty as appropriate

Empty: let both substrings be empty, global alignment value = 0

Local alignment

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

$$\text{Otherwise, let } V[i, j] = \max \begin{cases} V[i-1, j] + s(x[i-1], -) \\ V[i, j-1] + s(-, y[j-1]) \\ V[i-1, j-1] + s(x[i-1], y[j-1]) \\ 0 \end{cases}$$

$s(a, b)$ assigns a score to a particular match, gap, or replacement

What's different from global alignment?

First row and columns initialized to all 0s

0 is one of the arguments of the max

Local alignment: Smith-Waterman

Does it make sense that first row and column get all 0s?

Yes, b/c global alignment value of ϵ, ϵ (0) always best

Y

X	ϵ	T	A	T	A	T	G	C	G	G	C	G	T	T	T
ϵ	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
G	0														
G	0														
T	0														
A	0														
T	0														
G	0														
C	0														
T	0														
G	0														
G	0														
C	0														
G	0														
C	0														
T	0														
A	0														

$s(a, b)$

	A	C	G	T	-
A	2	-4	-4	-4	-6
C	-4	2	-4	-4	-6
G	-4	-4	2	-4	-6
T	-4	-4	-4	2	-6
-	-6	-6	-6	-6	

JOHNS HOPKINS
WHITING SCHOOL of ENGINEERING

Local alignment: Smith-Waterman

$$V[i,j] = \max \begin{cases} V[i-1,j] + s(x[i-1],-) \\ V[i,j-1] + s(-,y[j-1]) \\ V[i-1,j-1] + s(x[i-1],y[j-1]) \\ 0 \end{cases}$$

	ϵ	T	A	T	A	T	G	C	G	G	C	G	T	T	T
ϵ	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
G	0	0	0	0	0	0	2	0	2	2	0	2	0	0	0
G	0	0	0	0	0	0	2	0	2	4	0	2	0	0	0
T	0	2	0	2	0	2	0	0	0	0	0	0	4	2	2
A	0	0	4	0	?										
T	0														
G	0														
C	0														
T	0														
G	0														
G	0														
C	0														
G	0														
C	0														
T	0														
A	0														

$s(a,b)$

	A	C	G	T	-
A	2	-4	-4	-4	-6
C	-4	2	-4	-4	-6
G	-4	-4	2	-4	-6
T	-4	-4	-4	2	-6
-	-6	-6	-6	-6	

JOHNS HOPKINS
WHITING SCHOOL of ENGINEERING

Local alignment: Smith-Waterman

$$V[i,j] = \max \begin{cases} V[i-1,j] + s(x[i-1],-) \\ V[i,j-1] + s(-,y[j-1]) \\ V[i-1,j-1] + s(x[i-1],y[j-1]) \\ 0 \end{cases}$$

	ϵ	T	A	T	A	T	G	C	G	G	C	G	T	T	T
ϵ	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
G	0	0	0	0	0	0	2	0	2	2	0	2	0	0	0
G	0	0	0	0	0	0	2	0	2	4	0	2	0	0	0
T	0	2	0	2	0	2	0	0	0	0	0	0	4	2	2
A	0	0	4	0	4	0	0	0	0	0	0	0	0	0	0
T	0	2	0	6	0	6	0	0	0	0	0	0	2	2	2
G	0	0	0	0	2	0	8	2	2	2	0	2	0	0	0
C	0	0	0	0	0	0	2	10	4	0	4	0	0	0	0
T	0	2	0	2	0	2	0	4	6	0	0	0	2	2	2
G	0	0	0	0	0	0	4	0	6	8	2	2	0	0	0
G	0	0	0	0	0	0	2	0	2	8	4	4	0	0	0
C	0	0	0	0	0	0	0	4	0	2	10	4	0	0	0
G	0	0	0	0	0	0	2	0	6	2	4	12	6	0	0
C	0	0	0	0	0	0	0	4	0	2	4	6	8	2	0
T	0	2	0	2	0	2	0	0	0	0	0	0	8	10	4
A	0	0	4	0	4	0	0	0	0	0	0	0	2	4	6

$s(a,b)$

	A	C	G	T	-
A	2	-4	-4	-4	-6
C	-4	2	-4	-4	-6
G	-4	-4	2	-4	-6
T	-4	-4	-4	2	-6
-	-6	-6	-6	-6	

0's in essence allow peaks of similarity to rise above "background" of 0s

JOHNS HOPKINS
WHITING SCHOOL of ENGINEERING

Local alignment: Smith-Waterman

Backtrace: (a) start from *maximal* cell in the matrix, (b) stop backtrace when we reach a cell with score = 0

	ϵ	T	A	T	A	T	G	C	G	G	C	G	T	T	T
ϵ	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
G	0	0	0	0	0	0	2	0	2	2	0	2	0	0	0
G	0	0	0	0	0	0	2	0	2	4	0	2	0	0	0
T	0	2	0	2	0	2	0	0	0	0	0	0	4	2	2
A	0	0	4	0	4	0	0	0	0	0	0	0	0	0	0
T	0	2	0	6	0	6	0	0	0	0	0	0	2	2	2
G	0	0	0	0	2	0	8	2	2	2	0	2	0	0	0
C	0	0	0	0	0	0	2	10	4	0	4	0	0	0	0
T	0	2	0	2	0	2	0	4	6	0	0	0	2	2	2
G	0	0	0	0	0	0	4	0	6	8	2	2			
G	0	0	0	0	0	0	2	0	2	8	4	4			
C	0	0	0	0	0	0	0	4	0	2	10	4			
G	0	0	0	0	0	0	2	0	6	2	4	12	6	0	0
C	0	0	0	0	0	0	0	4	0	2	4	6	8	2	0
T	0	2	0	2	0	2	0	0	0	0	0	0	8	10	4
A	0	0	4	0	4	0	0	0	0	0	0	0	2	4	6

$s(a, b)$

	A	C	G	T	-
A	2	-4	-4	-4	-6
C	-4	2	-4	-4	-6
G	-4	-4	2	-4	-6
T	-4	-4	-4	2	-6
-	-6	-6	-6	-6	

```
y:  TATATGC-GGCGTTT
        |||||  ||||
x:  GGTATGCTGGCGCTA
```

JOHNS HOPKINS
WHITING SCHOOL of ENGINEERING

Local alignment: Smith-Waterman

What if we didn't have a positive "bonus" for matches?

All cells would = 0

	ϵ	T	A	T	A	T	G	C	G	G	C	G	T	T	T
ϵ	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
G	0	0	0	0	0	0	2	0	2	2	0	2	0	0	0
G	0	0	0	0	0	0	2	0	2	4	0	2	0	0	0
T	0	2	0	2	0	2	0	0	0	0	0	0	4	2	2
A	0	0	4	0	4	0	0	0	0	0	0	0	0	0	0
T	0	2	0	6	0	6	0	0	0	0	0	0	2	2	2
G	0	0	0	0	2	0	8	2	2	2	0	2	0	0	0
C	0	0	0	0	0	0	2	10	4	0	4	0	0	0	0
T	0	2	0	2	0	2	0	4	6	0	0	0	2	2	2
G	0	0	0	0	0	0	4	0	6	8	2	2	0	0	0
G	0	0	0	0	0	0	2	0	2	8	4	4	0	0	0
C	0	0	0	0	0	0	0	4	0	2	10	4	0	0	0
G	0	0	0	0	0	0	2	0	6	2	4	12	6	0	0
C	0	0	0	0	0	0	0	4	0	2	4	6	8	2	0
T	0	2	0	2	0	2	0	0	0	0	0	0	8	10	4
A	0	0	4	0	4	0	0	0	0	0	0	0	2	4	6

$s(a, b)$

	A	C	G	T	-
A	2	-4	-4	-4	-6
C	-4	2	-4	-4	-6
G	-4	-4	2	-4	-6
T	-4	-4	-4	2	-6
-	-6	-6	-6	-6	

What if we didn't have negative "penalties" for edits?

Rule for ϵ, ϵ would never be used and alignment would essentially be global

$$\max \begin{cases} V[i-1,j] + s(x[i-1],-) \\ V[i,j-1] + s(-,y[j-1]) \\ V[i-1,j-1] + s(x[i-1],y[j-1]) \\ 0 \end{cases}$$

JOHNS HOPKINS
WHITING SCHOOL of ENGINEERING

Local alignment: Smith-Waterman

```
def smithWaterman(x, y, s):
    """ Calculate local alignment values of sequences x and y using
        dynamic programming.  Return maximal local alignment value. """
    V = numpy.zeros((len(x)+1, len(y)+1), dtype=int)
    for i in xrange(1, len(x)+1):
        for j in xrange(1, len(y)+1):
            V[i, j] = max(V[i-1, j-1] + s(x[i-1], y[j-1]), # diagonal
                          V[i-1, j  ] + s(x[i-1], '-'),    # vertical
                          V[i  , j-1] + s('-',    y[j-1]), # horizontal
                          0)                               # empty
    argmax = numpy.where(V == V.max())
    return int(V[argmax])
```

Python example: http://nbviewer.ipython.org/6994170

Optimising Global Alignment

- How can we reduce memory usage?

Space usage revisited

We said the fill step requires $O(mn)$ space

	ϵ	T	A	T	G	T	C	A	T	G	C
ϵ	0	8	16	24	32	40	48	56	64	72	80
T	8	0	8	16	24	32	40	48	56	64	72
A	16	8	0	8	16	24	32	40	48	56	64
C	24	16	8	2	10	18	24	32	40	48	56
G	32	24	16	10	2	10	18	26	34	40	48
T	40	32	24	16	10	2	10	18	26	34	42
C	48	40	32	24	18	10	2	10	18	26	34
A	56	48	40	32	26	18	10	2	10	18	26
G	64	56	48	40	32	26	18	10	6	10	18
C	72	64	56	48	40	34	26	18	12	10	10

Can we do better?

Assume we're only interested in cost / score in lower right-hand cell

Space usage revisited

	ϵ	T	A	T	G	T	C	A	T	G	C
ϵ	0	8	16	24	32	40	48	56	64	72	80
T	8	0	8	16	24	32	40	48	56	64	72
A											
C											
G											
T											
C											
A											
G											
C											

Space usage revisited

Idea: just store current and previous rows. Discard older rows as we go. (Likewise for columns or antidiagonals.)

	ϵ	T	A	T	G	T	C	A	T	G	C
ϵ	0	8	16	24	32	40	48	56	64	72	80
T	8	0	8	16	24	32	40	48	56	64	72
A	?										
C											
G											
T											
C											
A											
G											
C											

← Discard this row...

← ...once we begin this row

Only keeping $O(1)$ rows at a time, space bound becomes $O(\min(n, m))$ -- linear space

JOHNS HOPKINS
WHITING SCHOOL of ENGINEERING

Space usage revisited

Idea: just store current and previous rows. Discard older rows as we go. (Likewise for columns or antidiagonals.)

	ϵ	T	A	T	G	T	C	A	T	G	C
ϵ											
T											
A											
C											
G											
T											
C											
A											
G	**64**	**56**	**48**	**40**	**32**	**26**	**18**	**10**	**6**	**10**	**18**
C	**72**	**64**	**56**	**48**	**40**	**34**	**26**	**18**	**12**	**10**	**10**

We get desired value / score, by looking in the lower right cell (global alignment)

JOHNS HOPKINS
WHITING SCHOOL of ENGINEERING

Space usage revisited

More savings: discard *elements* as soon as they're no longer needed

Discard this element

	ϵ	T	A	T	G	T	C	A	T	G	C
ϵ											
T					24	32	40	48	56	64	72
A	16	8	0	8	16	24					
C											
G											
T											
C											
A											
G											
C											

Once we fill this element

Dynamic programming summary

- Edit distance is harder to calculate than Hamming distance, but there is a $O(mn)$ time dynamic progamming algorithm
- Global alignment generalizes edit distance to use a cost function
- Slight tweaks to global alignment turn it into an algorithm for:
  - Finding approximate occurrences of P in T
- Local alignment also has a $O(mn)$-time dynamic programming solution
- Further efficiencies are possible:
  - If no alignment is needed, global/local alignment can be made linear-space
  - If alignment is needed, global alignment can be made linear-space with Hirschberg (not discussed today)
  - SIMD instructions can fill in chunks of cells at a time (not discussed)

More ideas: http://en.wikipedia.org/wiki/Smith-Waterman_algorithm#Accelerated_versions

Paper discussion

- Basic Local Alignment Search Tool - BLAST - Altschul et al.
  - Fast approximation to local alignment
  - One of the first programs to allow fast searches of large sequence databases
  - Classic algorithm; ubiquitous in genomics (>62,000 citations)
- Mapping short DNA sequencing reads and calling variants using mapping quality scores - Li et al.
  - One of the most successful early tools for short read sequencing