Dynamic Programming & Edit Distance

Ben Langmead

JOHNS HOPKINS
WHITING SCHOOL of ENGINEERING

Department of Computer Science

Please sign guestbook (www.langmead-lab.org/teaching-materials) to tell me briefly how you are using the slides. For original Keynote files, email me (ben.langmead@gmail.com).

Approximate matching for biosequences

Some widely used approximate matching algorithms for DNA (& other strings) came from the biological community, aiming to bring alignments into closer harmony with biological processes that mutate strings

```
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
```

Hamming / edit distance

For X, Y where $|X| = |Y|$, *hamming distance* = minimum # substitutions needed to turn one into the other

For X, Y, *edit distance* = minimum # edits (substitutions, insertions, deletions) needed to turn one into the other

Finding distances

```
def hammingDistance(x, y):

    ????
```

Strategy: walk along both strings. For each position, compare the characters in both strings at that position. If not equal, increment hamming distance:

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

Finding distances

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


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

Finding Hamming distance between strings is pretty easy.
What about edit distance?

Edit distance

If $|X| = |Y|$ what can we say about the relationship between **editDistance**(X, Y) and **hammingDistance**(X, Y)?

editDistance(X, Y) ≤ **hammingDistance**(X, 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
```

Edit distance

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

`editDistance`(*X*, *Y*) $\geq$ | | *X* | - | *Y* | |

X: ? ?

Y: ? ? ? ?

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

X

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

Y

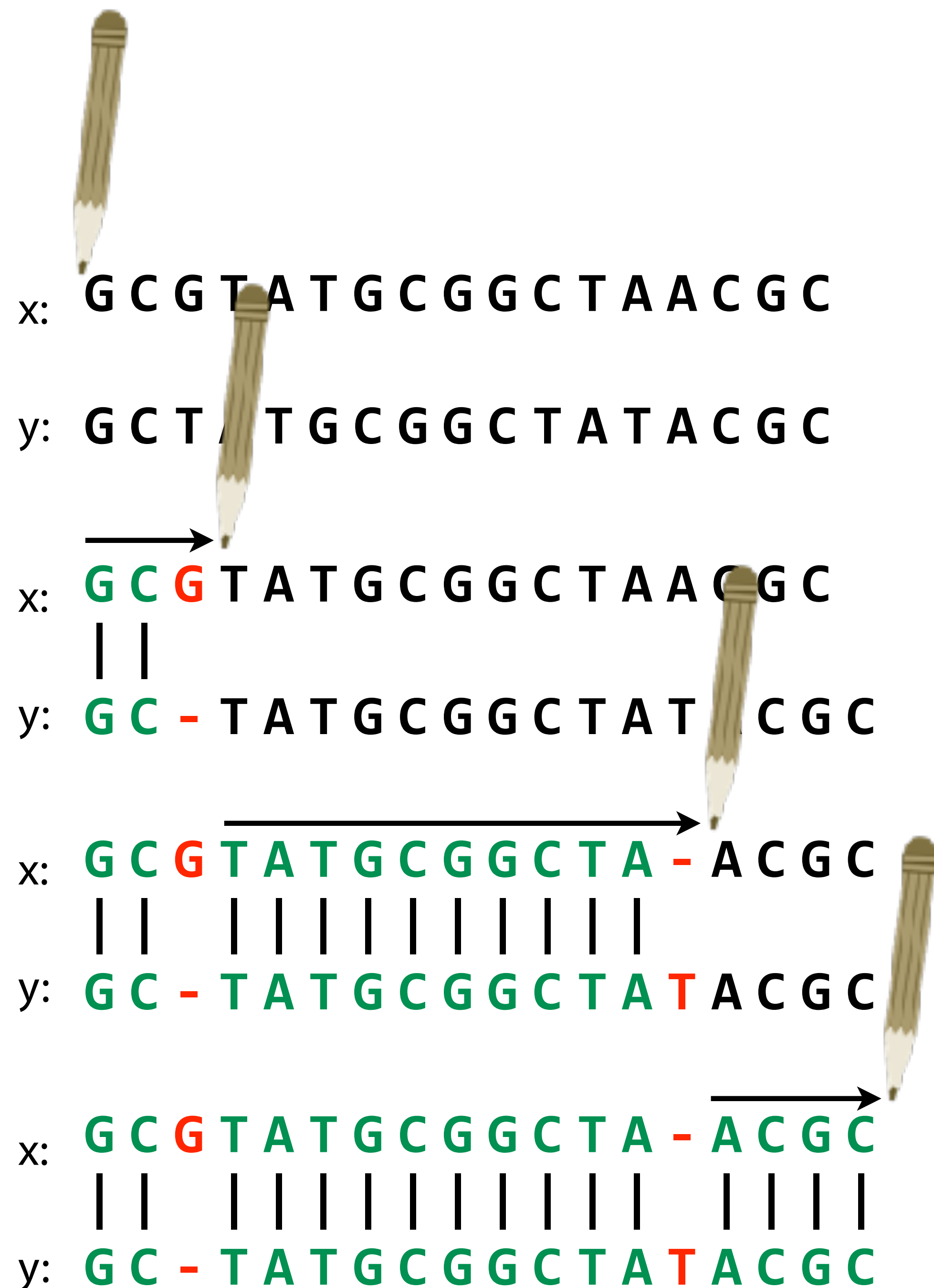
A A A T G C C G C G A A A A A C A T A

`editDistance`(*X*[:-1], *Y*[:-1]) = 147

Knowing distances between substrings of X & Y (or prefixes, suffixes) tells you something about overall distance

Edit distance

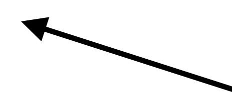
Imagine edits are introduced by an *optimal editor* working left-to-right:



Edit transcript summarizes how editor turns *x* into *y*

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

MMD



Reminder: this is D, not I, because we have to delete a character from *x* to make it more like *y*

MMDMMMMMMMMMMI

MMDMMMMMMMMMMIMMMM

Edit distance

Alignments:

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

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

replacement (AKA substitution/mismatch)

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

Edit transcripts (w/r/t *x*):

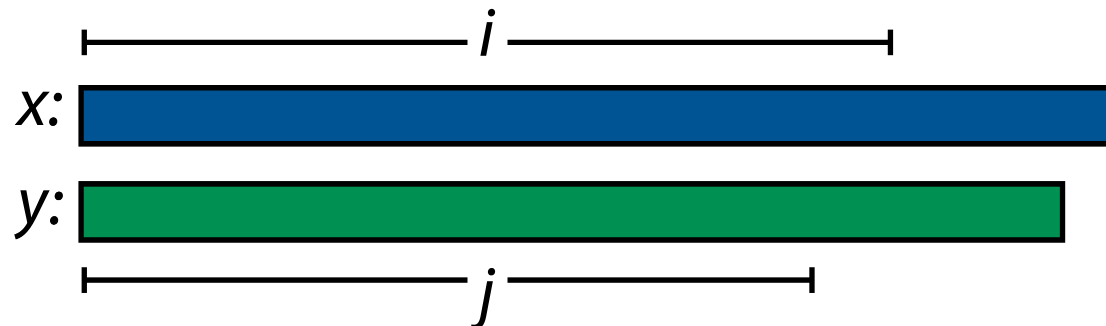
MMDMMMMMMMMMMIMMMM — Distance = 2

MMDMMMMRMMMMMIMMMM — Distance = 3

DDDDMMMMMMMIIII — Distance = 8

Edit distance

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



Optimal edit transcript for $D[i, j]$ is built by extending a shorter one by 1 operation. 3 options:

Append **D** to transcript for $D[i\text{-}1, j]$

Append **I** to transcript for $D[i, j\text{-}1]$

Append **M** or **R** to transcript for $D[i\text{-}1, j\text{-}1]$

$D[i, j]$ and its transcript come from 1 of these 3 choices, whichever has fewest edits

$D[|x|, |y|]$ is the overall edit distance

Edit distance

Is at beginning

Ds at beginning

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

vertical (**D**)

horizontal (**I**)

diagonal (**M** or **R**)

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

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

Direct implementation of recurrence above:

```
def edDistRecursive(a, b):
    if len(a) == 0:
        return len(b)
    if len(b) == 0:
        return len(a)
    delt = 1 if a[-1] != b[-1] else 0
    return min(edDistRecursive(a[:-1], b[:-1]) + delt,
               edDistRecursive(a[:-1], b) + 1,
               edDistRecursive(a, b[:-1]) + 1)
```

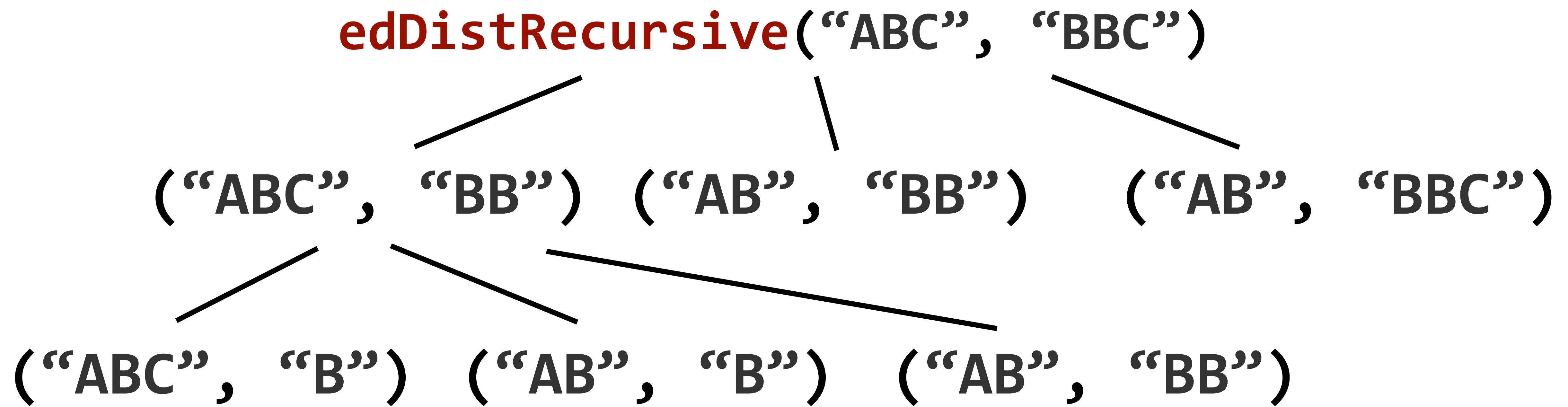
Python example: http://bit.ly/CG_DP_EditDist

Edit distance

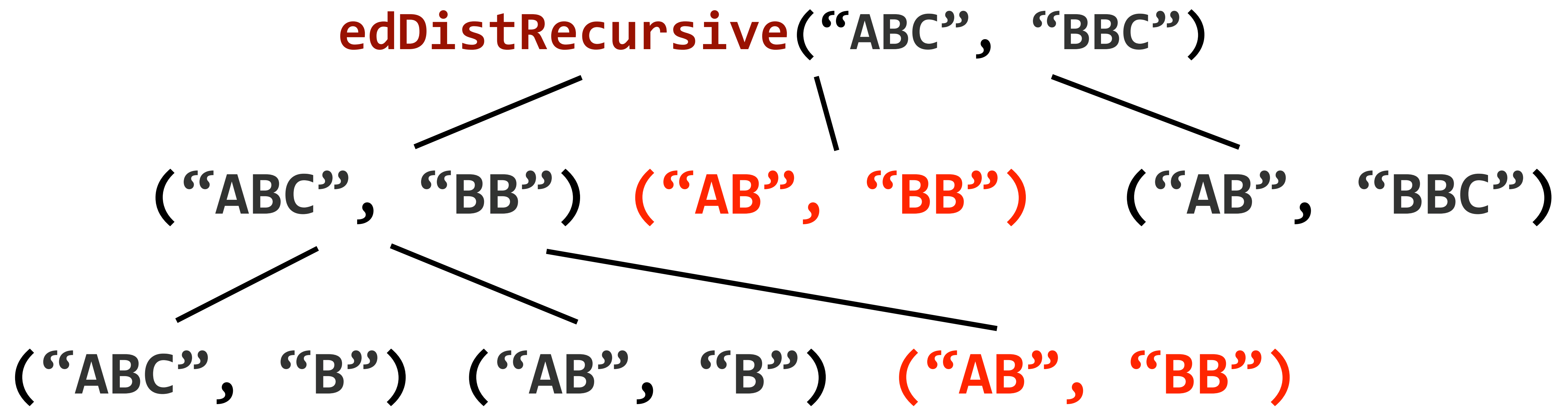
```
def edDistRecursive(a, b):
    if len(a) == 0:
        return len(b)
    if len(b) == 0:
        return len(a)
    delt = 1 if a[-1] != b[-1] else 0
    return min(edDistRecursive(a[:-1], b[:-1]) + delt,
               edDistRecursive(a[:-1], b) + 1,
               edDistRecursive(a, b[:-1]) + 1)
```

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

Takes >30 seconds for a small problem



(only part of recursion tree shown)



(only part of recursion tree shown)

```
n = 0
def edDistRecursive(a, b):
    global n
    if len(a) == 0:
        return len(b)
    if len(b) == 0:
        return len(a)
    if a == 'Shake' and b == 'shake':
        n += 1
    delt = 1 if a[-1] != b[-1] else 0
    return min(edDistRecursive(a[:-1], b[:-1]) + delt,
               edDistRecursive(a[:-1], b) + 1,
               edDistRecursive(a, b[:-1]) + 1)
```

```
>>> edDistRecursive("Shakespeare", "shake spear")
3
>>> n
8989
```

Edit distance

Subproblems (D[*i*, *j*]s) are reused many times; no need to recalculate them

Reusing solutions to subproblems is *memoization:*

Return memo, if available

Memoize D[*i*, *j*]

```
def edDistRecursiveMemo(a, b, memo):
    if len(a) == 0:
        return len(b)
    if len(b) == 0:
        return len(a)
    if (len(a), len(b)) in memo:
        return memo[(len(a), len(b))]
    delt = 1 if a[-1] != b[-1] else 0
    ans = min(edDistRecursiveMemo(a[:-1], b[:-1], memo) + delt,
              edDistRecursiveMemo(a[:-1], b, memo) + 1,
              edDistRecursiveMemo(a, b[:-1], memo) + 1)
    memo[(len(a), len(b))] = ans
    return ans
```

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

Edit distance: dynamic programming

```
def edDistRecursiveMemo(a, b, memo):
    if len(a) == 0:
        return len(b)
    if len(b) == 0:
        return len(a)
    if (len(a), len(b)) in memo:
        return memo[(len(a), len(b))]
    delt = 1 if a[-1] != b[-1] else 0
    ans = min(edDistRecursiveMemo(a[:-1], b[:-1], memo) + delt,
              edDistRecursiveMemo(a[:-1], b, memo) + 1,
              edDistRecursiveMemo(a, b[:-1], memo) + 1)
    memo[(len(a), len(b))] = 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
```

Edit distance: dynamic programming

edDistRecursiveMemo is a *top-down* dynamic programming approach

Alternative is *bottom-up*: fill a table (matrix) of D[*i*, *j*]s:

```
import numpy

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 range(1, len(x)+1):
        for j in range(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)]
```

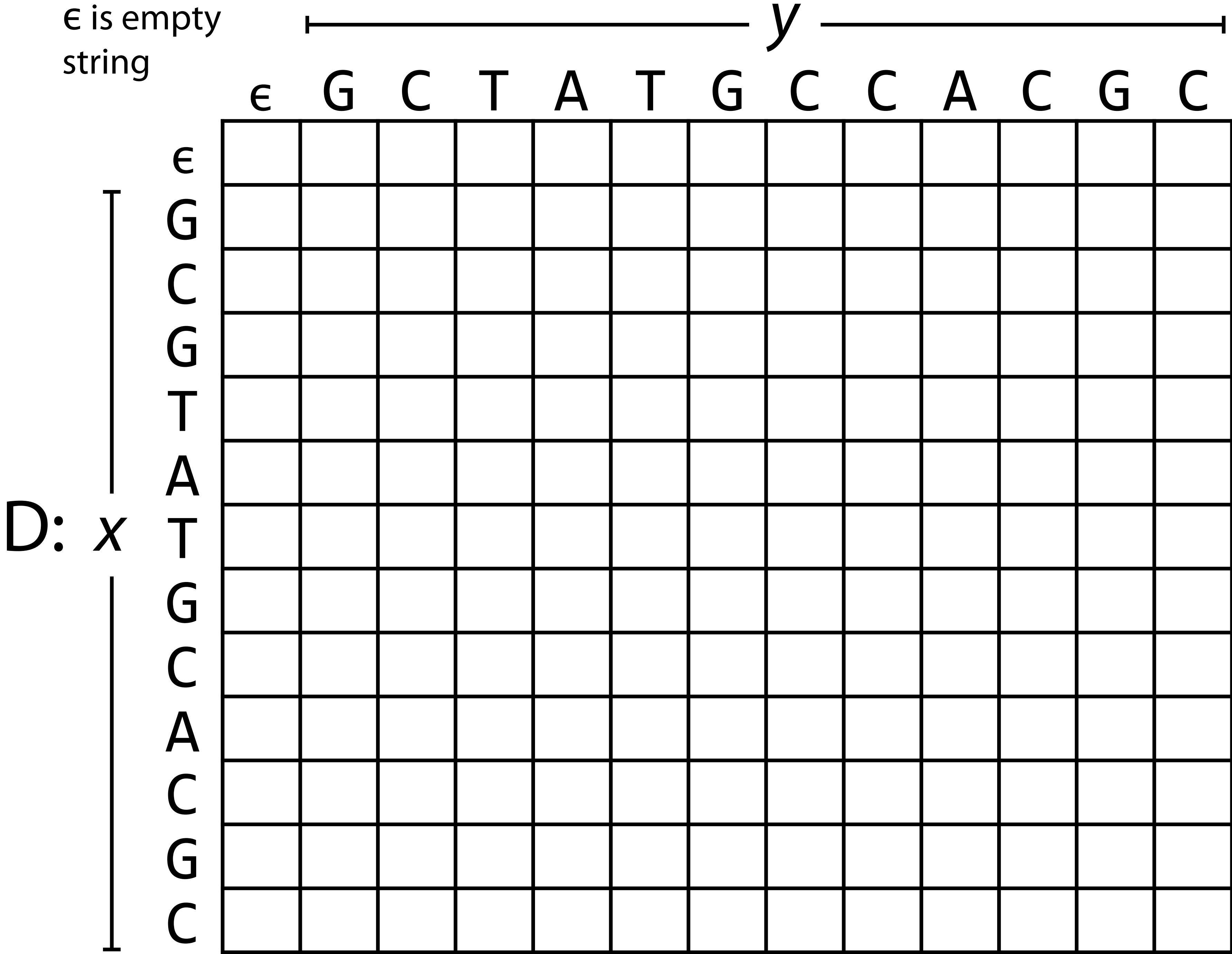
numpy: package for matrices, etc

Fill 1st row, col

Fill rest of matrix

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

Edit distance: dynamic programming

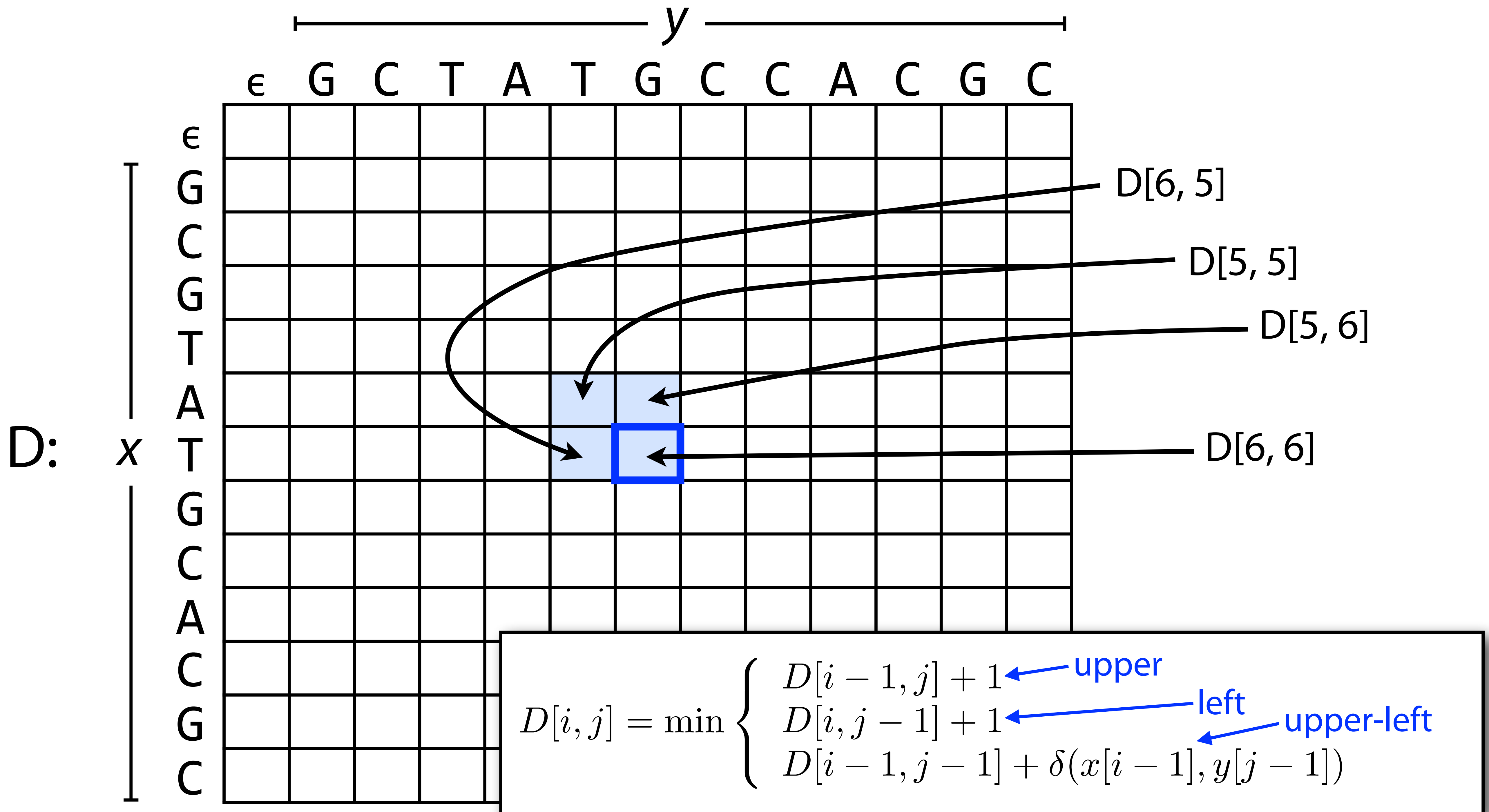


Let $n = | x |$, $m = | y |$

D: $(n+1)$ x $(m+1)$ matrix

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

Edit distance: dynamic programming



Cell depends upon its upper, left, and upper-left neighbors

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 range(1, len(x)+1):
    for j in range(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 range(1, len(x)+1):
    for j in range(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
```

Edit distance: dynamic programming

Loop from **edDistDp**:

```
for i in range(1, len(x)+1):
    for j in range(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 range(1, len(x)+1):
    for j in range(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 range(1, len(y)+1):
    for i in range(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												

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

Or in blocks

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	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 for x, y

But where and what are the 2 edits?

Edit distance: getting the alignment

Traceback corresponds to an optimal alignment / edit transcript

At each step, ask: which neighbor (⇖, ⇐ or ⇑) 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

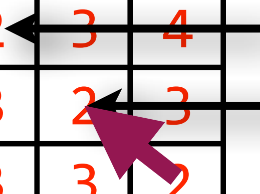
Q: How did I get here?

Edit distance: getting the alignment

Traceback corresponds to an optimal alignment / edit transcript

At each step, ask: which neighbor (⇖, ⇐ or ⇑) 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

Traceback corresponds to an optimal alignment / edit transcript

At each step, ask: which neighbor (⇖, ⇐ or ⇑) gave the minimum?

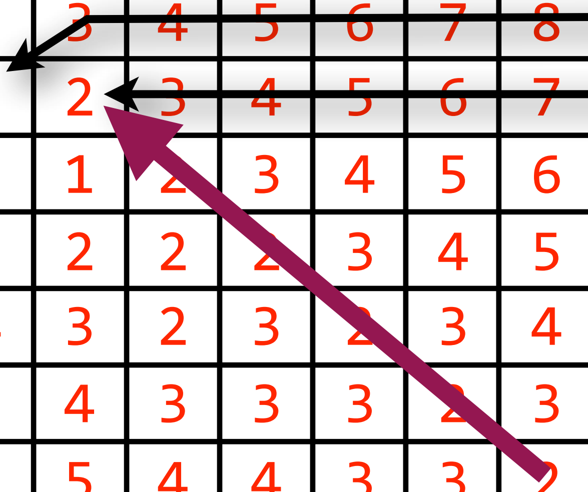
	ϵ	G	C	T	A	T	G	C	C	A	C	G	C
ϵ	0	1	2	3	4	5	6	7	8	9	10	11	12
G	1	0	1	2	3	4	5	6	7	8	9	10	11
C	2	1	0	1	2	3	4	5	6	7	8	9	10
G	3	2	1	1	2	3	3	4	5	6	7	8	9
T	4	3	2	1	2	2	3	4	5	6	7	8	9
A	5	4	3	2	1	2	3	4	5	5	6	7	8
T	6	5	4	3	2	1	2	3	4	5	6	7	8
G	7	6	5	4	3	2	1	2	3	4	5	6	7
C	8	7	6	5	4	3	2	1	2	3	4	5	6
A	9	8	7	6	5	4	3	2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

Traceback corresponds to an optimal alignment / edit transcript

At each step, ask: which neighbor (⇖, ⇐ or ⇑) 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

Traceback corresponds to an optimal alignment / edit transcript

At each step, ask: which neighbor (⇖, ⇐ or ⇑) 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: getting the alignment

To find the edit distance and the alignment we did:
(a) table filling, (b) traceback

	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

If $|X| = m$ and $|Y| = n$, how much work is table filling?

Filling $(m + 1)(n + 1)$ cells, each requiring constant work, so $O(mn)$

How much work is traceback?

Each step goes $\Nwarrow$, $\Leftarrow$ or $\Uparrow$.

Worst case: traceback never moves diagonally, requiring m $\Uparrow$'s and n $\Leftarrow$'s, so $O(m + n)$

Edit distance: summary

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

FillIng matrix is $O(mn)$ space and time, yields edit distance

Traceback is $O(m + n)$ time, yields optimal alignment / edit transcript

Approximate matching with edit distance

Can we search for the occurrence of *P* in *T* with 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	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 matching with edit distance

1. Initialize first row with 0's rather than increasing integers, then fill matrix

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																					
	A	2																					
	C	3																					
	G	4																					
	T	5																					
	C	6																					
	A	7																					
	G	8																					
	C	9																					

Approximate matching with edit distance

1. Initialize first row with 0's rather than increasing integers, then fill matrix

T (columns), *P* (rows)

	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
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 matching with edit distance

1. Initialize first row with 0's rather than increasing integers, then fill matrix
2. Pick lowest edit distance *in the bottom row*, traceback to top row

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	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
T	1	1	1	1	1	1	0	1	0	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
A	2	1	1	2	2	2	1	0	1	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
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	2	3	4	5	5	5	5

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

Approximate matching with edit distance

D[i, j] equals the optimal edit distance between the length-i prefix of P and...

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													
A	2	1	1	2	2	2	1	0	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
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

P

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

Approximate matching with edit distance

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

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													
T	1	1	1	1	1	1	0	1	0													
A	2	1	1	2	2	2	1	0	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
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

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

Approximate matching with edit distance

How would you find *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													
	T	1	1	1	1	1	1	0	1	0													
	A	2	1	1	2	2	2	1	0	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

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