String Alignment

02-251

Slides by Carl Kingsford

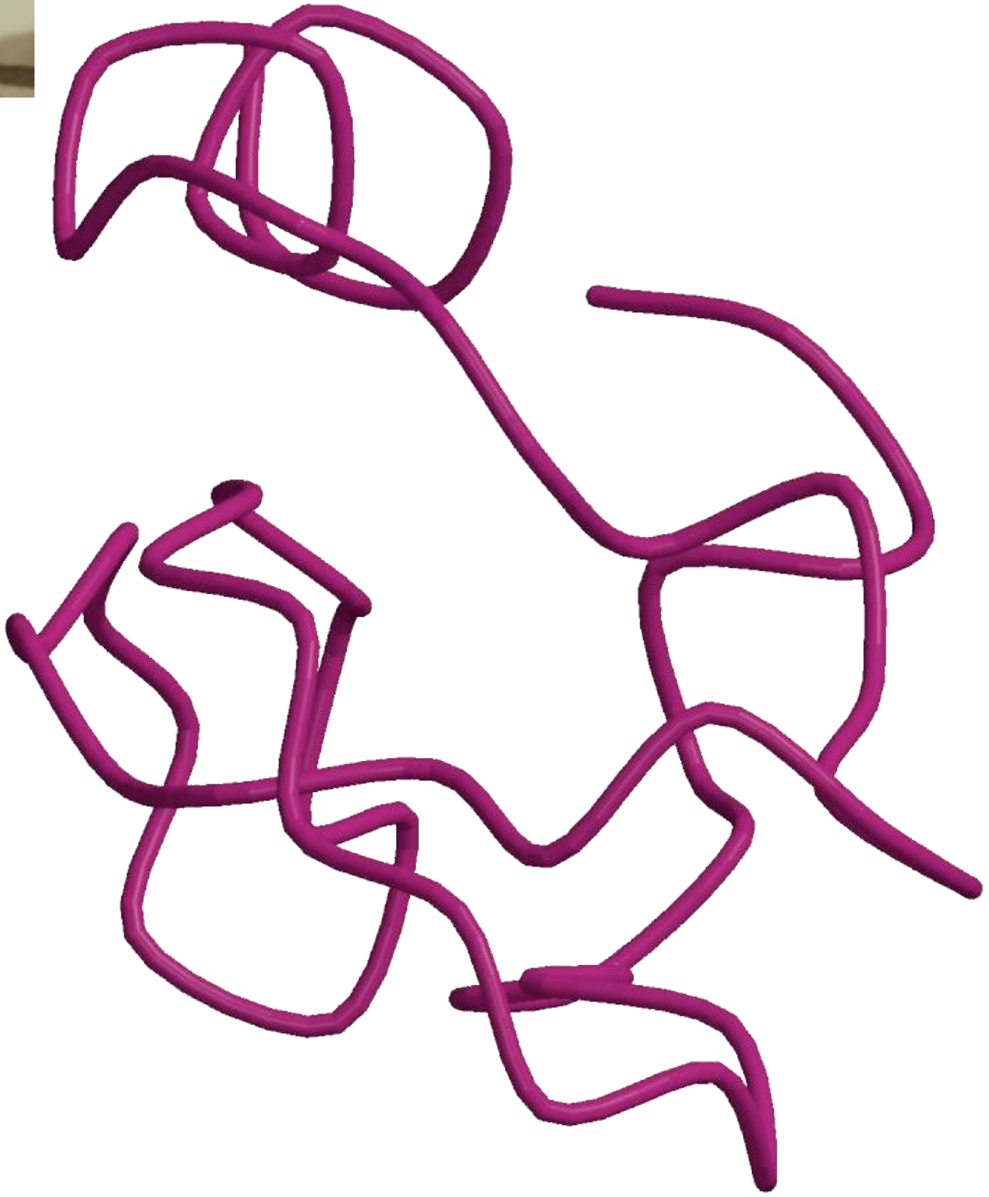
Why compare DNA or protein sequences?

Partial CTCF protein sequence in 8 organisms:

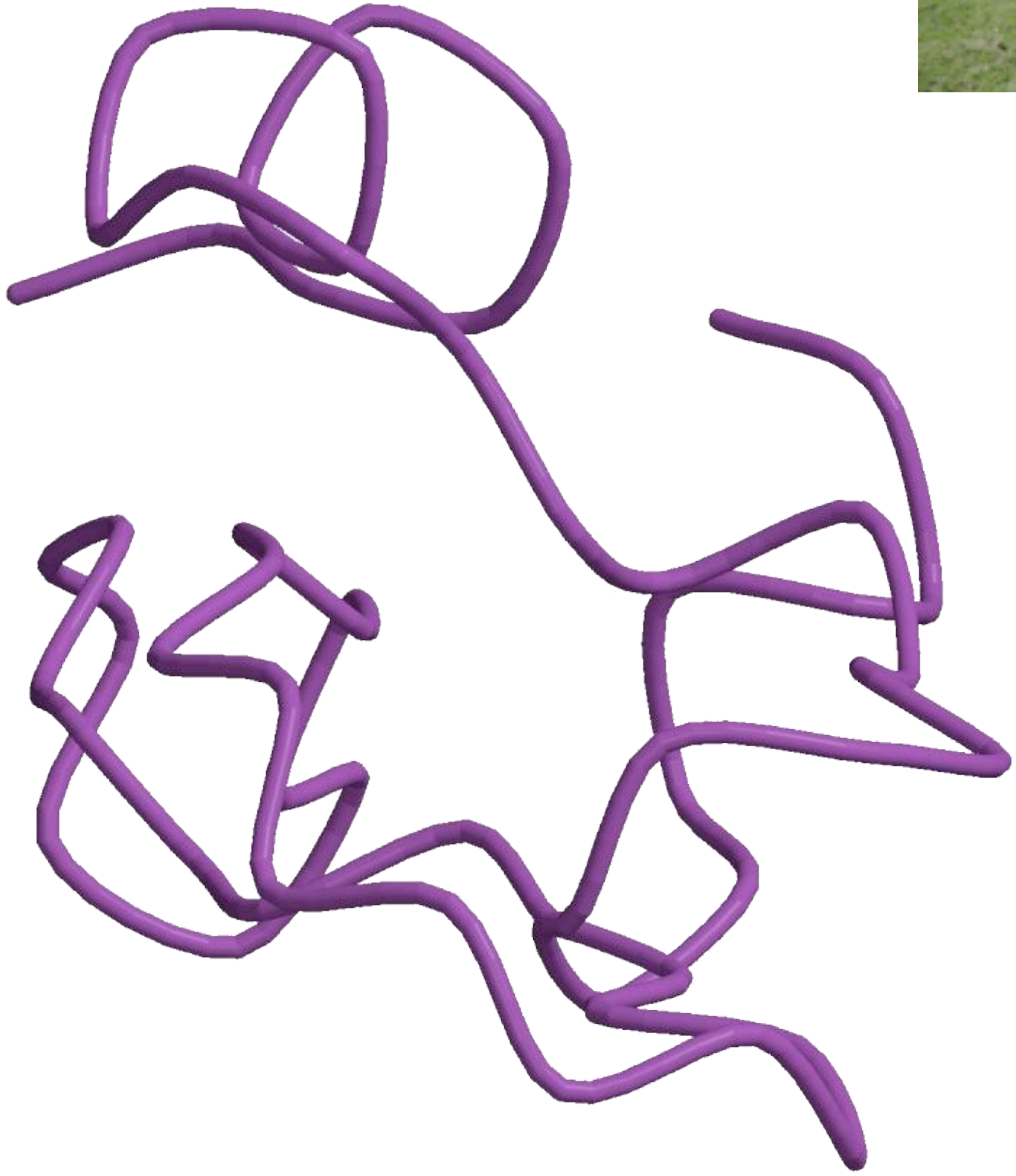
```
H. sapiens      -EDSSDS-ENAEPDLDDNEDEEEPAVEIEPEPE----------PQPVTPA
P. troglodytes  -EDSSDS-ENAEPDLDDNEDEEEPAVEIEPEPE----------PQPVTPA
C. lupus        -EDSSDS-ENAEPDLDDNEDEEEPAVEIEPEPE----------PQPVTPA
B. taurus       -EDSSDS-ENAEPDLDDNEDEEEPAVEIEPEPE----------PQPVTPA
M. musculus     -EDSSDSEENAEPDLDDNEEEEEPAVEIEPEPE--PQPQPPPPPQPVAPA
R. norvegicus   -EDSSDS-ENAEPDLDDNEEEEEPAVEIEPEPEPQPQPQPQPQPQPVAPA
G. gallus       -EDSSDSEENAEPDLDDNEDEEETAVEIEAEPE----------VSAEAPA
D. rerio        DDDDDDSDEHGEPDLDDIDEEDEDDL-LDEDQMGLLDQAPPSVPIP-APA
```

- Identify important sequences by finding conserved regions.
- Find genes similar to known genes.
- Understand evolutionary relationships and distances (D. rerio aka zebrafish is farther from humans than G. gallus aka chicken).
- Interface to databases of genetic sequences.
- As a step in genome assembly, and other sequence analysis tasks.
- Provide hints about protein structure and function (next slide).

Sequence can reveal structure



(a) 1dtk



(b) 5pti

```
1dtk   XAKYCKLPLRIGPCKRKIPSFYYKWKAKQCLPFDYSGCGGNANRFKTIEECRRTCVG-
5pti   RPDFCLEPPYTGPCKARIIRYFYNAKAGLCQTFVYGGCRAKRNNFKSAEDCMRTCGGA
```

The Simplest String Comparison Problem

Given: Two strings

$$a = a_1a_2a_3a_4 \ldots a_m$$
$$b = b_1b_2b_3b_4 \ldots b_n$$

where a_i, b_i are letters from some alphabet like {A,C,G,T}.

Compute how similar the two strings are.

What do we mean by "similar"?

Edit distance between strings *a* and *b* = the smallest number of the following operations that are needed to transform *a* into *b*:

- mutate (replace) a character
- delete a character
- insert a character

$$\text{riddle} \xrightarrow{\text{delete}} \text{ridle} \xrightarrow{\text{mutate}} \text{riple} \xrightarrow{\text{insert}} \text{triple}$$

Representing edits as alignments

```
prin-ciple
|||| |||xx
prinncipal
(1 gap, 2 mm)
```

```
prin-cip-le
|||| ||| |
prinncipal-
(3 gaps, 0 mm)
```

```
misspell
||| ||||
mis-pell
(1 gap)
```

```
prehistoric
   ||||||||
---historic
(3 gaps)
```

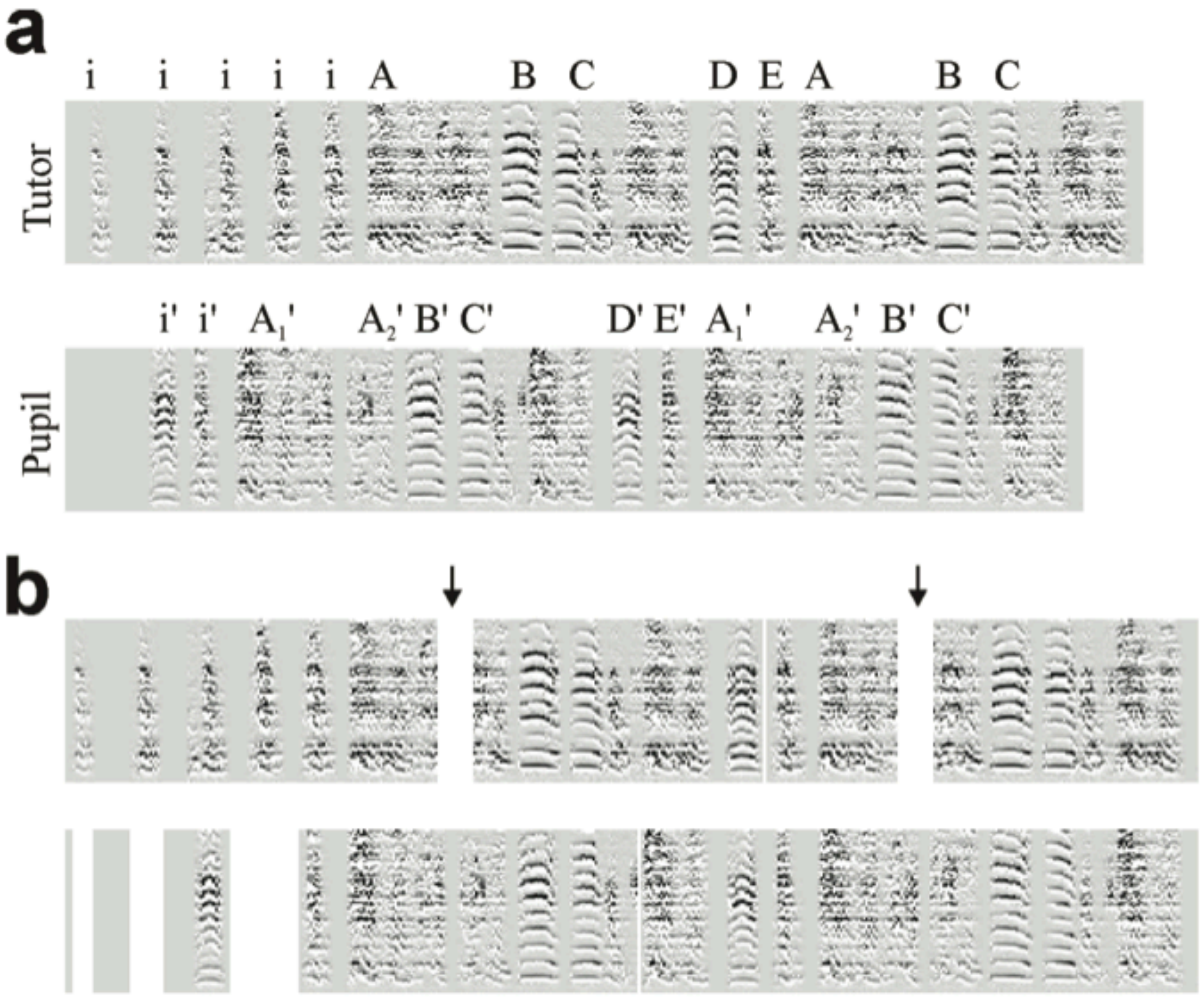
```
aa-bb-ccaabb
|x || | | |
ababbbc-a-b-
(5 gaps, 1 mm)
```

```
al-go-rithm-
|| xx ||x |
alKhwariz-mi
(4 gaps, 3 mm)
```

NCBI BLAST DNA Alignment

```
>gb|AC115706.7| Mus musculus chromosome 8, clone RP23-382B3, complete sequence

Query  1650   gtgtgtgtgggtgcacatttgtgtgtgtgtgcgcctgtgtgtgtgggtgcctgtgtgtgt  1709
              ||||||||||  |    ||  | ||||||||| | ||||||||    ||| ||  |||||
Sbjct  56838  GTGTGTGTGGAAGTGAGTTCATCTGTGTGTGCACATGTGTGTGCA--TGCATGCATGTGT  56895

Query  1710   gtg-gggcacatttgtgtgtgtgtgtgtgcctgtgtgtgggtgcacatttgtgtgtgc  1768
              ||  |||||      ||  ||| ||||||| |||||||| |||  |||  ||||| || |
Sbjct  56896  GTCCGGGCA------TGCATGTCTGTGTGCATGTGTGTGTGTGTGCAT--GTGTGAGTAC  56947

Query  1769   ctgtgtgtgtgtgcctgtgtgtgggggtgcacatttgtgtgtgtgtgtgcctgtgtgtgg  1828
              |||||||||| ||| ||| |||| | |||  ||| |||||  ||||||   |||||  |
Sbjct  56948  CTGTGTGTGTATGCTTGTATGTGTGTGTGTGCATGTGTGTAGGTGTGTATATGTGTAAGT  57007

Query  1829   gggtgcacatttgtgtgtgtgtgtgcctgtgtgtgtgggtgcacatttgtgtgtgtgtgt  1888
                     ||| ||||||| ||||||  |||| | |||  ||||    |||||||||| ||
Sbjct  57008  T------CATCTGTGTGTATGTGTG--TGTGAGAGTGCATGCA----TGTGTGTGTGAGT  57055

Query  1889   gcctgtgtgt--gtgggtgcacatttgtgtgtgtgtgcctgtg--tgtgt--gggtgcac  1942
               | | |||||  |||  |||  ||  || |   | |  |||||  |||||  | |||  |
Sbjct  57056  TCATCTGTGTCAGTGTATGCTTATGGGTATAACT-TAACTGTGCATGTGTAAGTGTGTTC  57114

Query  1943   atttgtgtgtgtgtgtgcctgtgtgtgtgggtgcacatttgtgtgtgtgtgcctgtgtgtgg  2002
              || ||||| ||||||||  |||||| ||  || ||    | ||||||||   |||||||
Sbjct  57115  ATCTGTGTATGTGTGTG--TGTGTGAGTTAGTTCA----TCTGTGTGTGAGAGTGTGTGA  57168

Query  2003   gtgcacatttgtgtgtgtgtgcctgtgtgtgtgtgcctgtgtgtgtgggtgcacatttgt  2062
              |  | ||| |||||||| || | | ||| || |||  ||||| |||| ||| |||  ||
Sbjct  57169  G--CTCATCTGTGTGTGAGTTCATCTGTATGAGTG--TGTGTATGTGTGTGTACAAATGA  57224

Query  2063   gtgtgtgtgtgcctgtgtgtgtgggtgcacatttgtgtgtgtgtgtgtgcctgtgtgtgt  2122
              ||   | ||||| ||||||||||        |||  ||||||   | ||  |||| ||||
Sbjct  57225  GTTCATCTGTGCATGTGTGTGTG--------TTTAAGTGTGTTCATCTG--TGTGCGTGT  57274
```

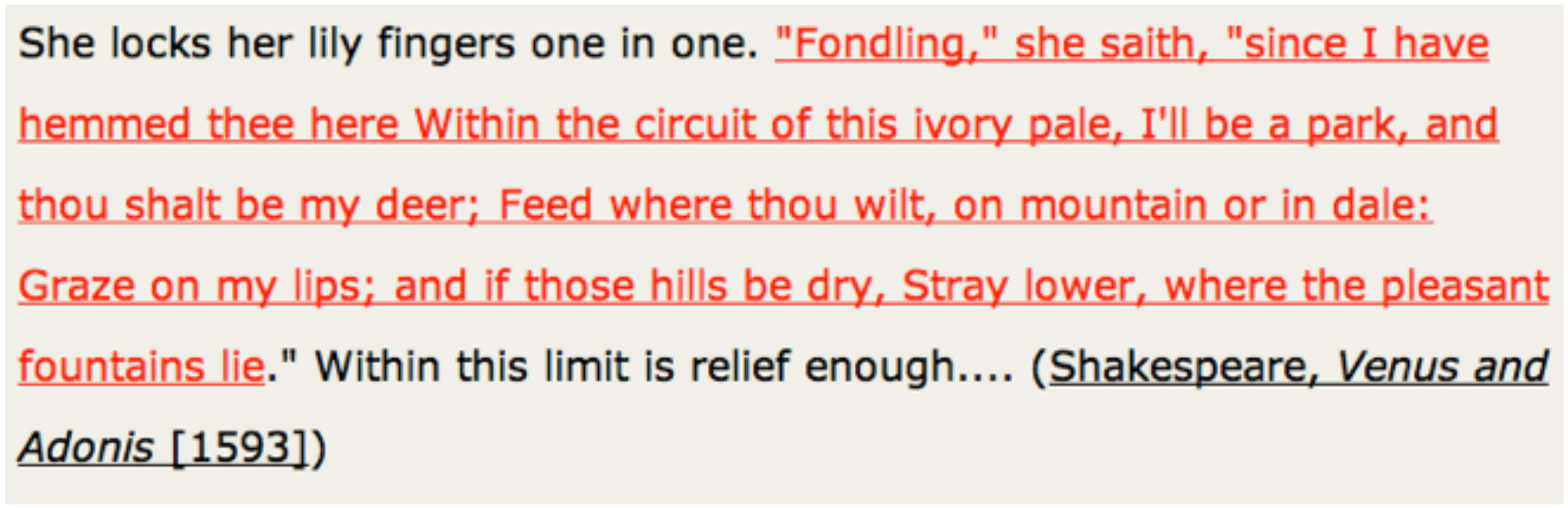
Comparing Bird Songs



Florian et al. Hidden Markov Models in the Neurosciences

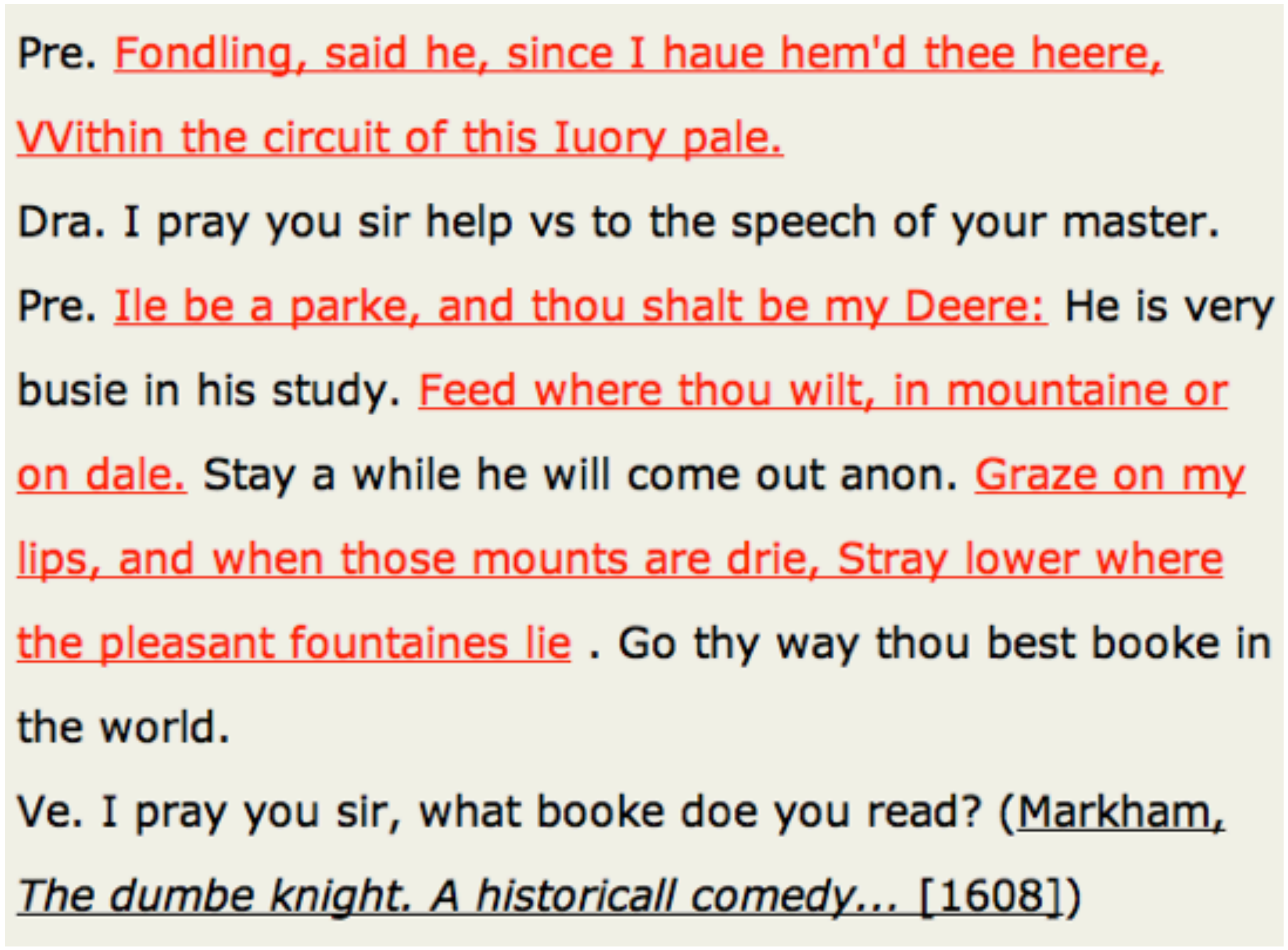
Tracing Textual Influences

Example from Horton, Olsen, Roe, Digital Studies / Le champ numérique, Vol 2, No 1 (2010)



She locks her lily fingers one in one. "Fondling," she saith, "since I have hemmed thee here Within the circuit of this ivory pale, I'll be a park, and thou shalt be my deer; Feed where thou wilt, on mountain or in dale: Graze on my lips; and if those hills be dry, Stray lower, where the pleasant fountains lie." Within this limit is relief enough.... (Shakespeare, *Venus and Adonis* [1593])

This later play by Markham references Shakespeare's poem.



Pre. Fondling, said he, since I haue hem'd thee heere, VVithin the circuit of this Iuory pale.

Dra. I pray you sir help vs to the speech of your master.

Pre. Ile be a parke, and thou shalt be my Deere: He is very busie in his study. Feed where thou wilt, in mountaine or on dale. Stay a while he will come out anon. Graze on my lips, and when those mounts are drie, Stray lower where the pleasant fountaines lie . Go thy way thou best booke in the world.

Ve. I pray you sir, what booke doe you read? (Markham, *The dumbe knight. A historicall comedy...* [1608])

Common passages identified by sequence alignment algorithms.

The String Alignment Problem

Parameters:

- "*gap*" is the cost of inserting a "-" character, representing an insertion or deletion
- *cost*(x,y) is the cost of aligning character x with character y.
  In the simplest case, *cost*(x,x) = 0 and *cost*(x,y) = mismatch penalty.

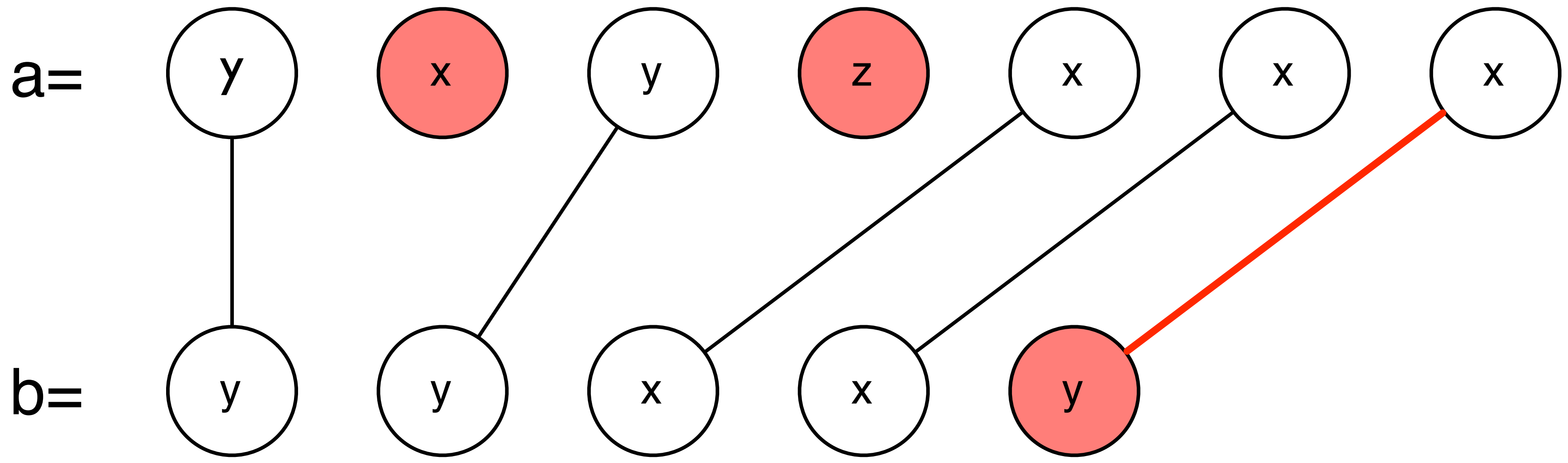
Goal:

- Can compute the edit distance by finding the **lowest cost alignment**.
- Cost of an alignment is: sum of the *cost*(x,y) for the pairs of characters that are aligned + *gap* × number of - characters inserted.

Another View: Alignment as a Matching

Each string is a set of nodes, one for each character.

Looking for a low-cost matching (pairing) between the sequences.



Cost of a matching is:

$$\text{gap} \times \#\mathit{unmatched} + \sum_{(a_i, b_j)} \mathit{cost}(a_i, b_j)$$

Edges are not allowed to cross!

Algorithm for Computing Edit Distance

Consider the last characters of each string:

$$a = a_1 a_2 a_3 a_4 \ldots a_m$$
$$b = b_1 b_2 b_3 b_4 \ldots b_n$$

One of these possibilities must hold:

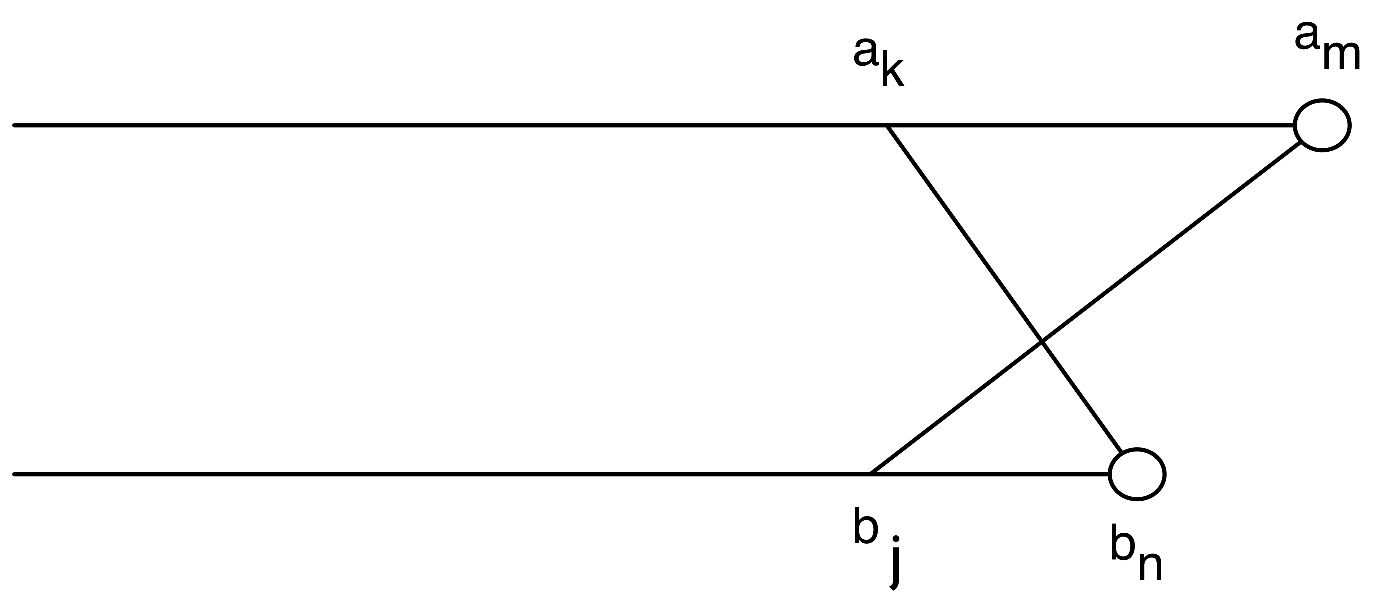
1. (a_m, b_n) are matched to each other
2. a_m is not matched at all
3. b_n is not matched at all
4. a_m is matched to some b_j $(j \neq n)$ and b_n is matched to some a_k $(k \neq m)$.

$\uparrow$

#4 can't happen! Why?

No Crossing Rule Forbids #4

4. a_m is matched to some b_j ($j \neq n$) and b_n is matched to some a_k ($k \neq m$).



So, the only possibilities for what happens to the last characters are:

1. (a_m, b_n) are matched to each other
2. a_m is not matched at all
3. b_n is not matched at all

Recursive Solution

Turn the 3 possibilities into 3 cases of a recurrence:

$$\underset{\substack{\uparrow\\\text{Cost of the optimal}\\\text{alignment between}\\a_1\ldots a_i\text{ and }b_1\ldots b_j}}{OPT(i,j)} = \min \begin{cases} \text{cost}(a_i, b_j) + OPT(i-1, j-1) & \text{match } a_i, b_j \\ \text{gap} + OPT(i-1, j) & a_i \text{ is not matched} \\ \text{gap} + \underset{\substack{\uparrow\\\text{Written in terms of}\\\text{the costs of smaller}\\\text{problems}}}{OPT(i, j-1)} & b_j \text{ is not matched} \end{cases}$$

Key: we don't know which of the 3 possibilities is the right one, so we try them all.

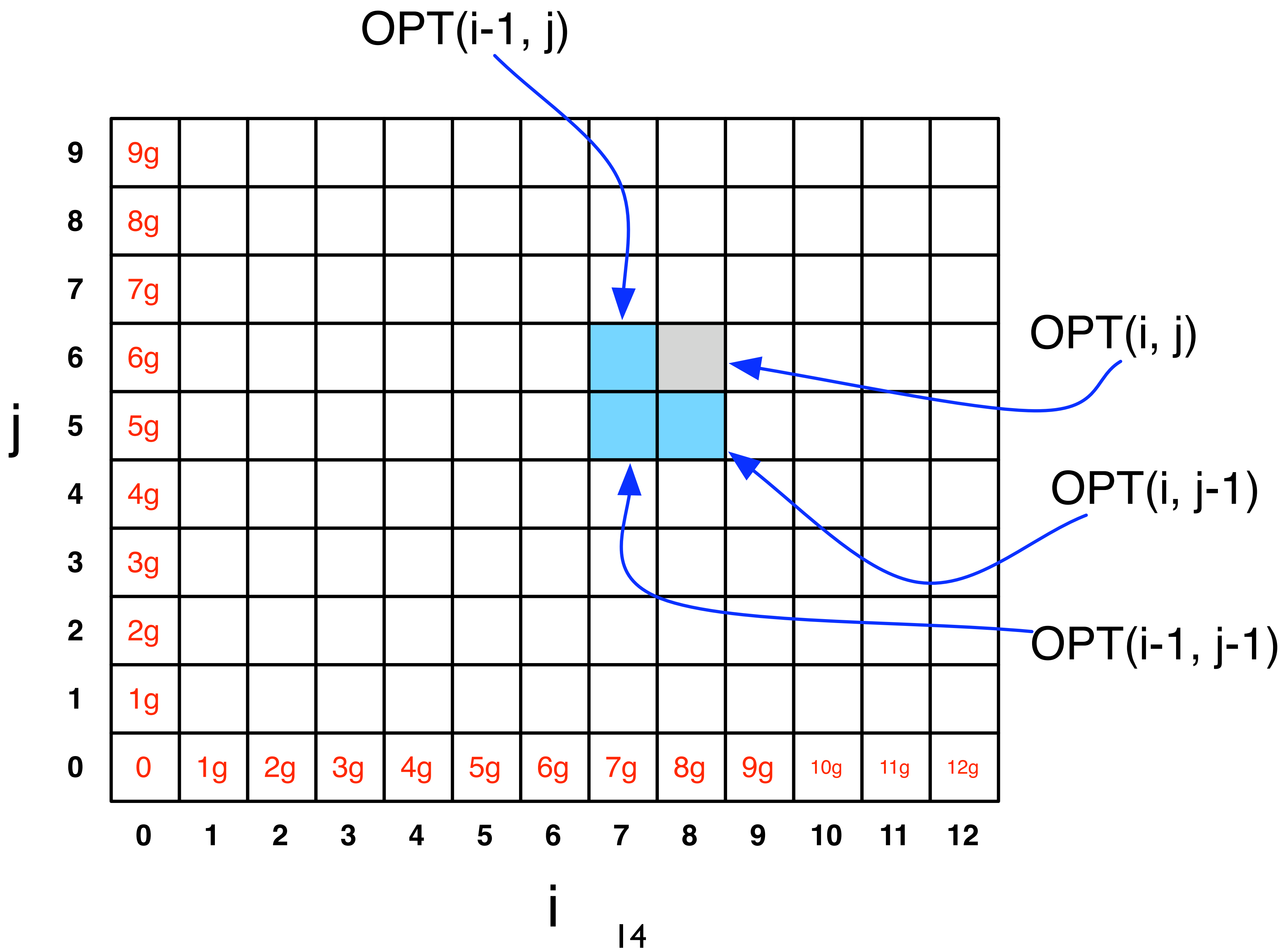
Base case: $OPT(i,0) = i \times \text{gap}$ and $OPT(0,j) = j \times \text{gap}$.

(Aligning *i* characters to 0 characters must use *i* gaps.)

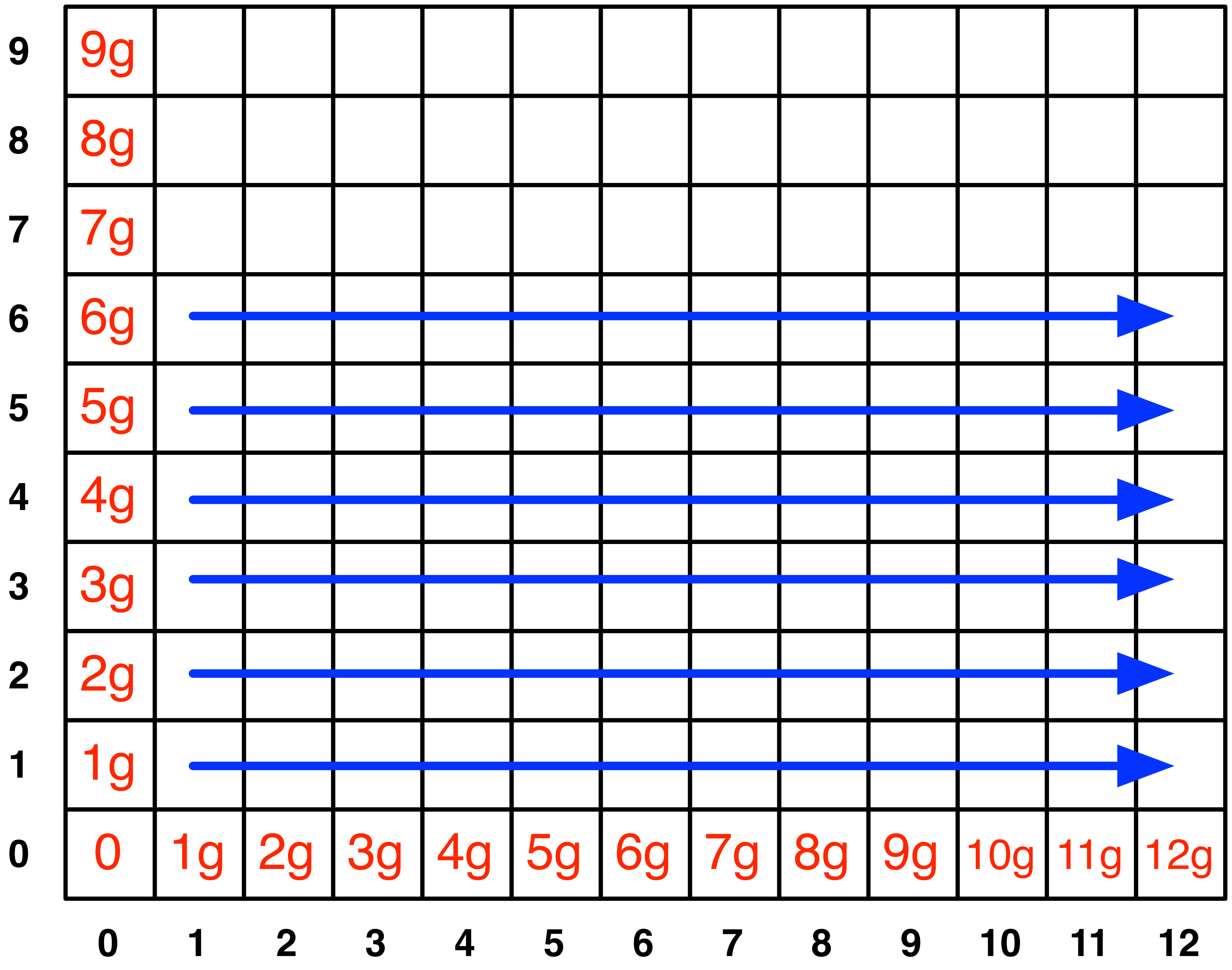
Computing OPT(i,j) Efficiently

We're ultimately interested in *OPT*(n,m), but we will compute all other *OPT*(i,j) ($i \leq n$, $j \leq m$) on the way to computing *OPT*(n,m).

Store those values in a 2D array:



Filling in the 2D Array



9	9g												
8	8g												
7	7g												
6	6g												
5	5g												
4	4g												
3	3g												
2	2g												
1	1g												
0	0	1g	2g	3g	4g	5g	6g	7g	8g	9g	10g	11g	12g
	0	1	2	3	4	5	6	7	8	9	10	11	12

Edit Distance Computation

```
EditDistance(X,Y):
   For i = 1,...,m: A[i,0] = i*gap
   For j = 1,...,n: A[0,j] = j*gap

   For i = 1,...,m:
      For j = 1,...,n:
         A[i,j] = min(
            cost(a[i],b[j]) + A[i-1,j-1],
            gap + A[i-1,j],
            gap + A[i,j-1]
         )
      EndFor
   EndFor
   Return A[m,n]
```

Where's the answer?

OPT(n,m) contains the edit distance between the two strings.

Why? By induction: EVERY cell contains the optimal edit distance between some prefix of string 1 with some prefix of string 2.

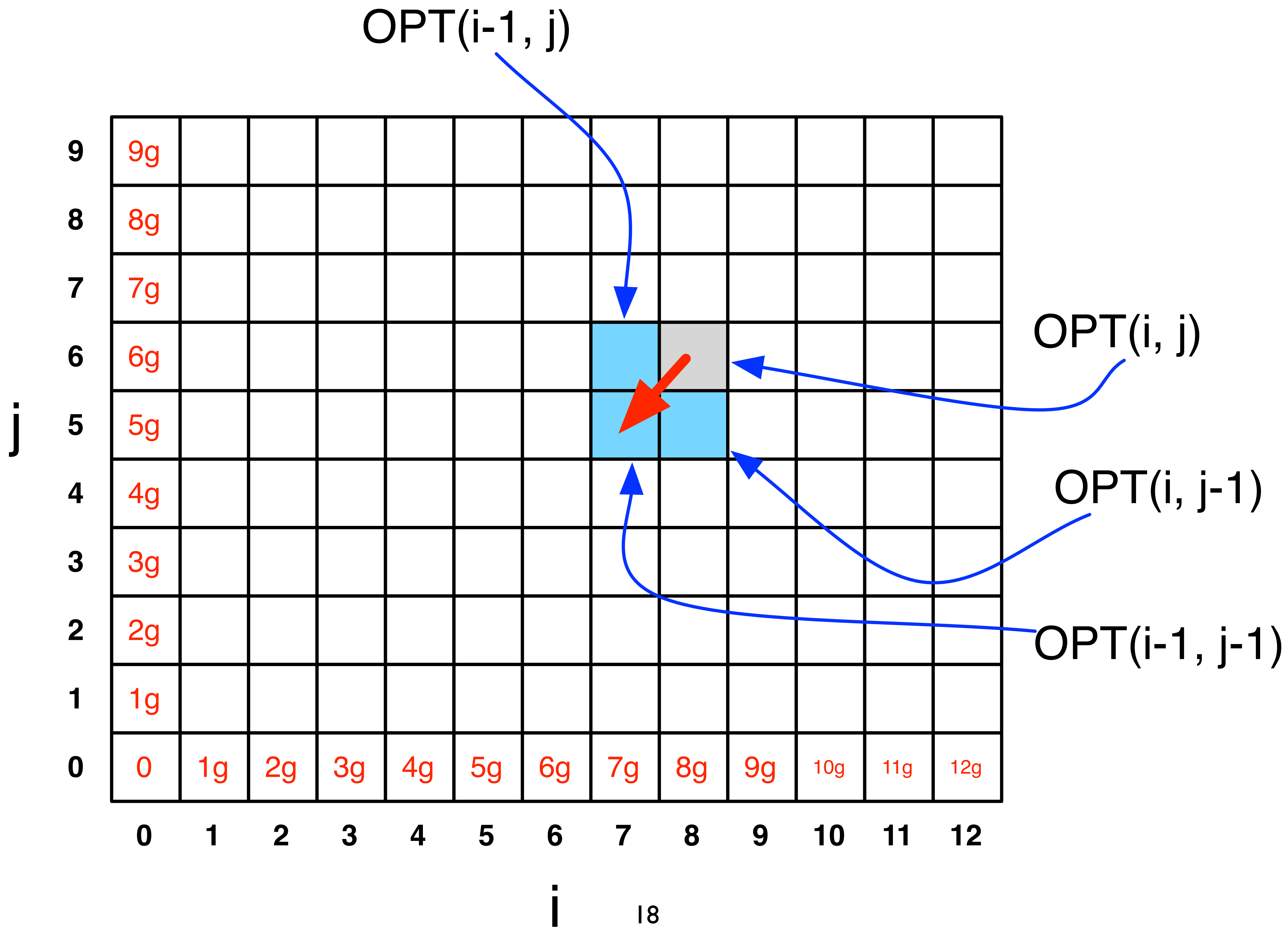
Running Time

Number of entries in array = O($m \times n$), where m and n are the lengths of the 2 strings.

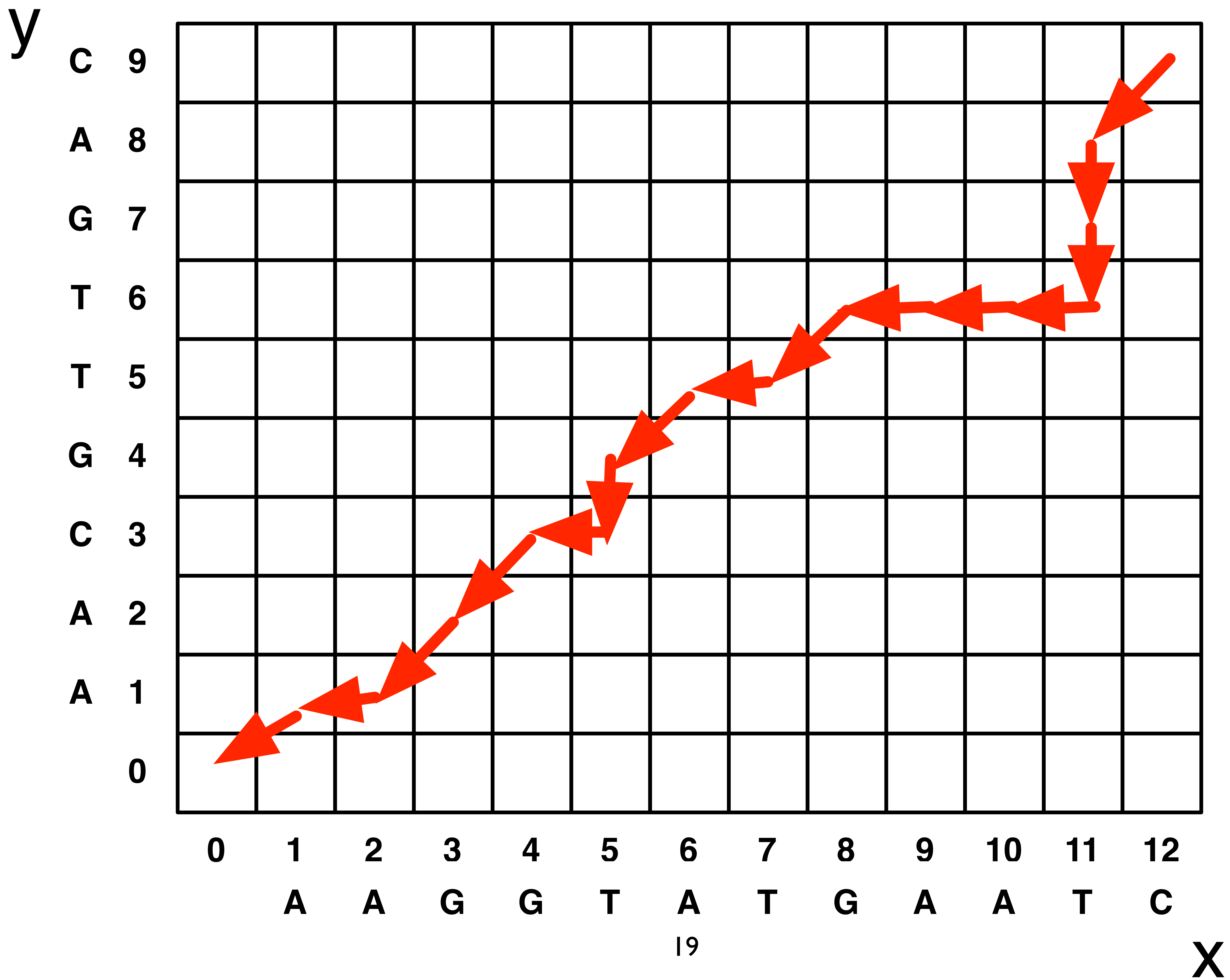
Filling in each entry takes constant O(1) time.

Total running time is O(mn).

Finding the actual alignment



Trace the arrows all the way back



Outputting the Alignment

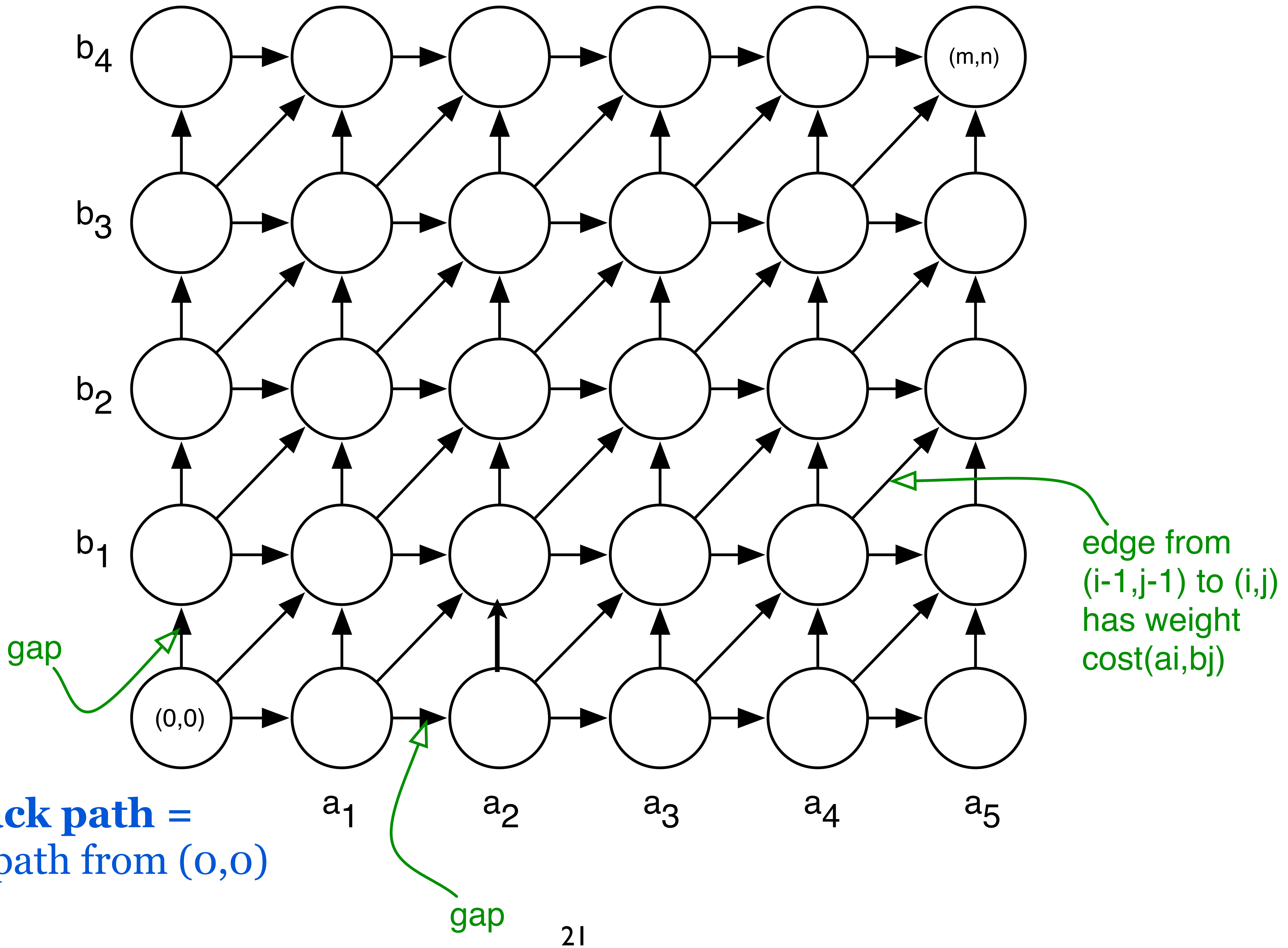
Build the alignment from right to left.

```
ACGT
A-GA
```

Follow the backtrack pointers starting from entry (n,m).

- If you follow a diagonal pointer, add both characters to the alignment,
- If you follow a left pointer, add a gap to the y-axis string and add the x-axis character
- If you follow a down pointer, add the y-axis character and add a gap to the x-axis string.

Another View: Recasting as a Graph



Traceback path = shortest path from (0,0) to (m,n)

Dynamic Programming

The previous sequence alignment / edit distance algorithm is an example of dynamic programming.

Main idea of dynamic programming: solve the subproblems in an order so that when you need an answer, it’s ready.

Requirements for DP to apply:

1. Optimal value of the original problem can be computed from some similar subproblems.
2. There are only a polynomial # of subproblems
3. There is a “natural” ordering of subproblems, so that you can solve a subproblem by only looking at **smaller** subproblems.

Local Alignment

02-251

Slides by Carl Kingsford

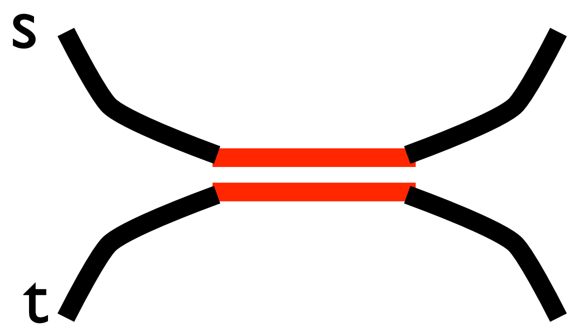
Maximization vs. Minimization

Edit distance:

$$OPT(i,j) = \min \begin{cases} \text{cost}(a_i, b_j) + OPT(i-1, j-1) & \text{match } a_i, b_j \\ \text{gap} + OPT(i-1, j) & a_i \text{ is not matched} \\ \text{gap} + OPT(i, j-1) & b_j \text{ is not matched} \end{cases}$$

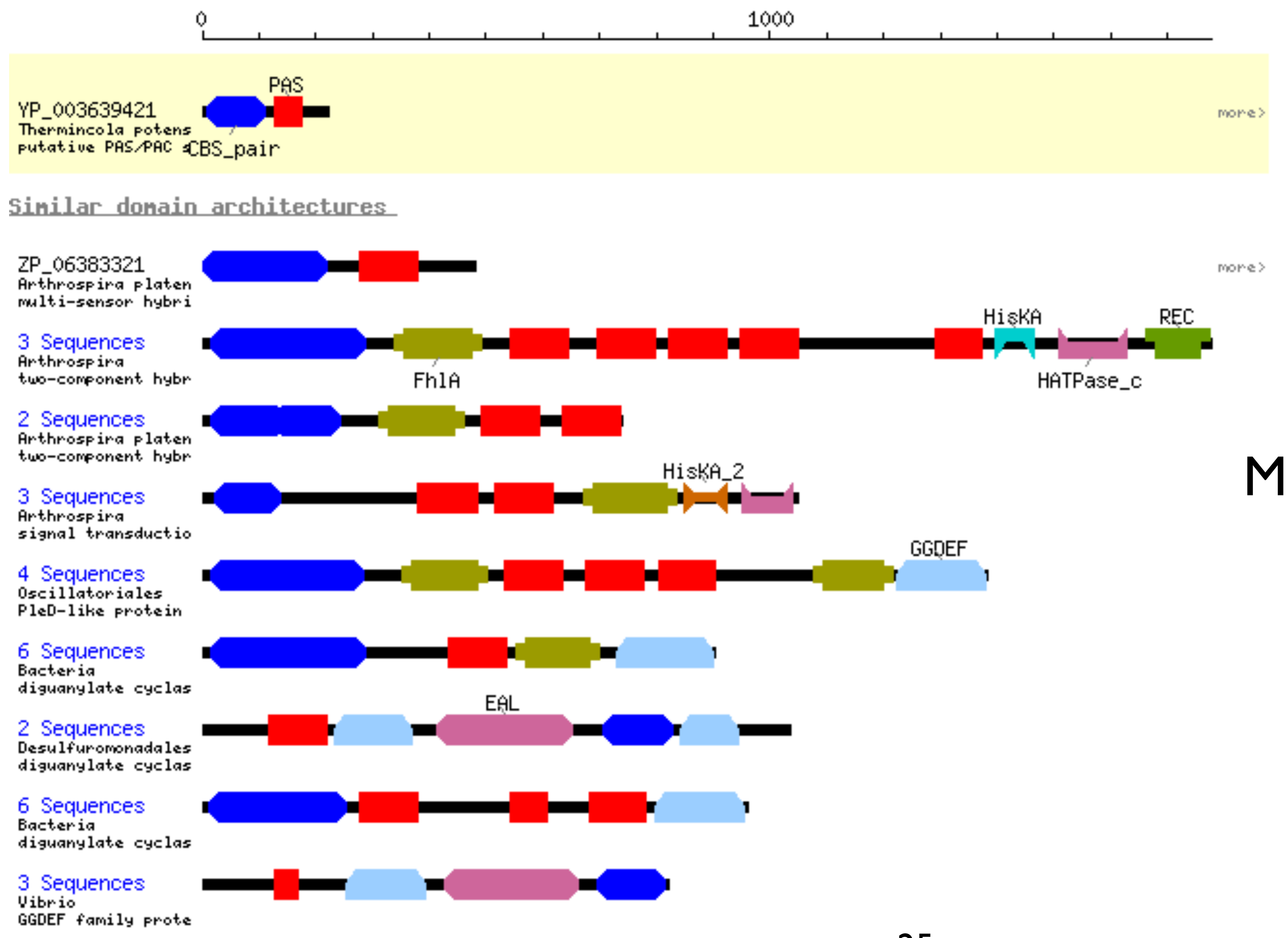
Sequence Similarity: replace min with a max and negate the parameters.

gap penalty → gap benefit (probably negative)
cost → score



Local Alignment

Local alignment between s and t: Best alignment between a subsequence of s and a subsequence of t.



Motivation:

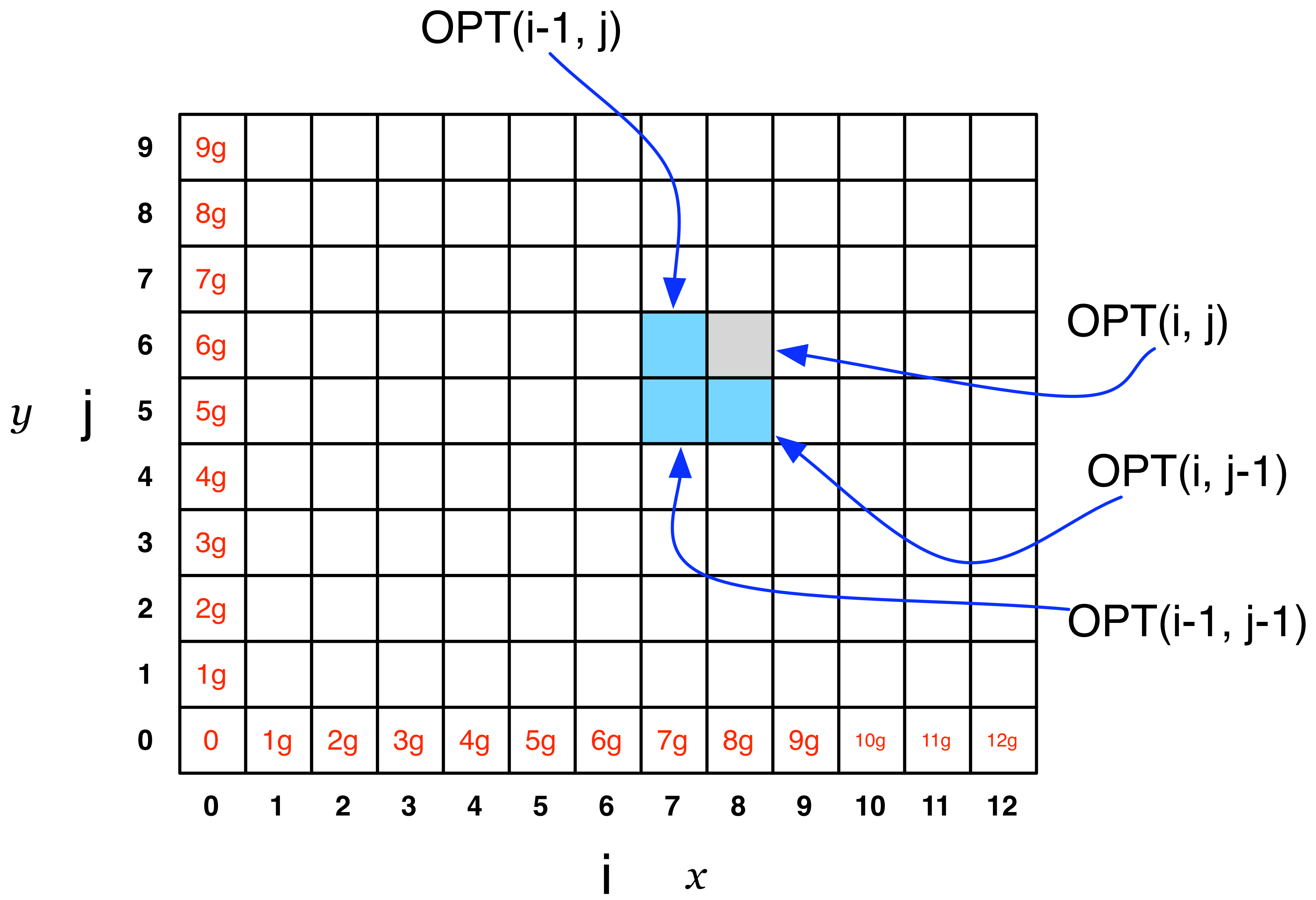
Many genes are composed of *domains*, which are subsequences that perform a particular function.

Recall: Global Alignment Matrix

$OPT(i,j)$ contains the score for the best alignment between:

the first i characters of string x [prefix i of x]

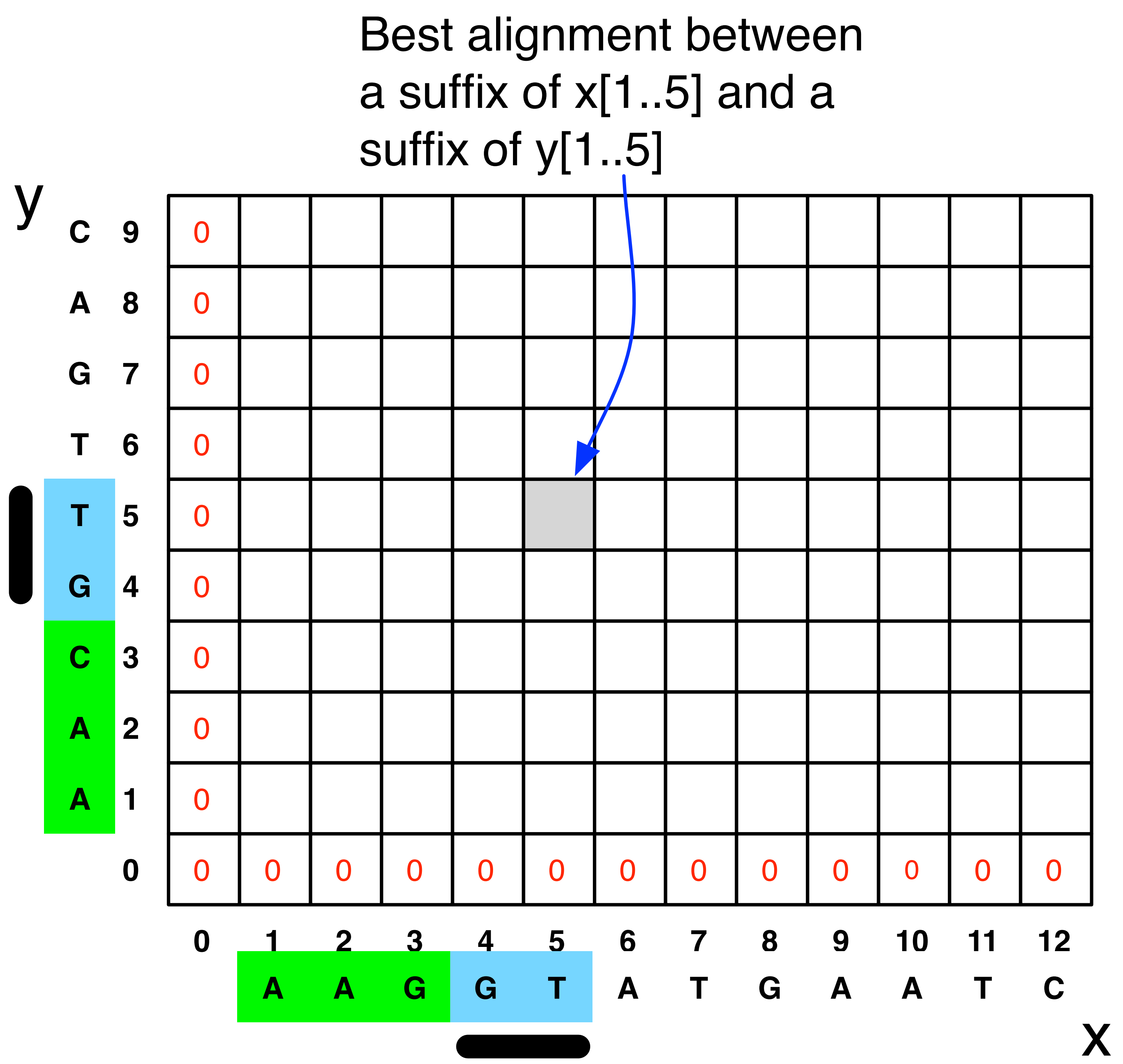
the first j character of string y [prefix j of y]



Local Alignment

New meaning of entry of matrix entry:

A[i, j] = best score between:
  some suffix of *x*[1…*i*]
  some suffix of *y*[1…*j*]



How do we fill in the local alignment matrix?

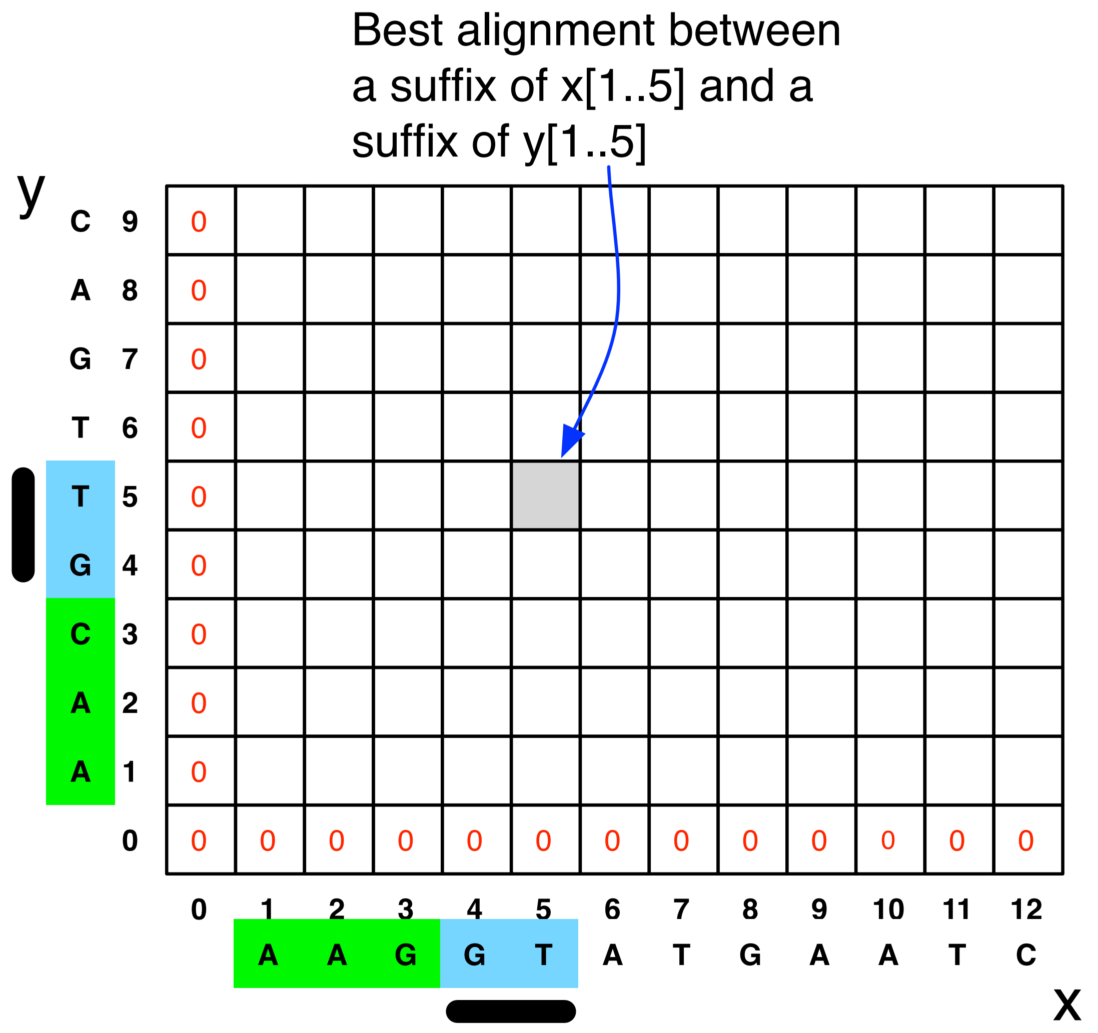
$$A[i,j] = \max \begin{cases} A[i,j-1] + \text{gap} & (1) \\ A[i-1,j] + \text{gap} & (2) \\ A[i-1,j-1] + \text{match}(i,j) & (3) \\ 0 \end{cases}$$

(1), (2), and (3): same cases as before:

gap in x, gap in y, match x and y

New case: 0 allows you to say the best alignment between a suffix of *x* and a suffix of *y* is the empty alignment.

Lets us "start over"



Local Alignment

- Initialize first row and first column to be 0.
- The score of the best local alignment is the largest value in the entire array.
- To find the actual local alignment:
    - start at an entry with the maximum score
    - traceback as usual
    - stop when we reach an entry with a score of 0

Local Alignment Python Code

```
def local_align(x, y, score=ScoreParam(-7, 10, -5)):
    """Do a local alignment between x and y"""
    # create a zero-filled matrix
    A = make_matrix(len(x) + 1, len(y) + 1)

    best = 0
    optloc = (0,0)

    # fill in A in the right order
    for i in xrange(1, len(y)):
        for j in xrange(1, len(x)):

            # the local alignment recurrence rule:
            A[i][j] = max(
               A[i][j-1] + score.gap,
               A[i-1][j] + score.gap,
               A[i-1][j-1] + (score.match if x[i] == y[j] else score.mismatch),
               0
            )

            # track the cell with the largest score
            if A[i][j] >= best:
                best = A[i][j]
                optloc = (i,j)

    # return the opt score and the best location
    return best, optloc
```

Local Alignment Python Code

```
def make_matrix(sizex, sizey):
    """Creates a sizex by sizey matrix filled with zeros."""
    return [[0]*sizey for i in xrange(sizex)]


class ScoreParam:
    """The parameters for an alignment scoring function"""
    def __init__(self, gap, match, mismatch):
        self.gap = gap
        self.match = match
        self.mismatch = mismatch
```

Local Alignment Example #1

```
local_align("AGCGTAG", "CTCGTC")
```

	*	A	G	C	G	T	A	G
*	0	0	0	0	0	0	0	0
C	0	0	0	10	3	0	0	0
T	0	0	0	3	5	13	6	0
C	0	0	0	↖10	3	6	8	1
G	0	0	10	3	↖20	13	6	18
T	0	0	3	5	13	↖**30**	23	16
C	0	0	0	13	6	23	25	18

```
Score(match) = 10
Score(mismatch) = -5
Score(gap) = -7
```

```
Note: this table written top-to-bottom
instead of bottom-to-top
```

Local Alignment Example #2

```
local_align("bestoftimes", "soften")
```

	*	b	e	s	t	o	f	t	i	m	e	s
*	0	0	0	0	0	0	0	0	0	0	0	0
s	0	0	0	10	3	0	0	0	0	0	0	10
o	0	0	0	3	5	13	6	0	0	0	0	3
f	0	0	0	0	0	6	23	16	9	2	0	0
t	0	0	0	0	10	3	16	33	26	19	12	5
e	0	0	10	3	3	5	9					
n	0	0	3	5	0	0	2					

```
Score(match) = 10
Score(mismatch) = -5
Score(gap) = -7
```

Note: this table written top-to-bottom instead of bottom-to-top

Local Alignment Example #2

```
local_align("bestoftimes", "soften")
```

	*	b	e	s	t	o	f	t	i	m	e	s
*	0	0	0	0	0	0	0	0	0	0	0	0
s	0	0	0	10	3	0	0	0	0	0	0	10
o	0	0	0	3	5	13	6	0	0	0	0	3
f	0	0	0	0	0	6	23	16	9	2	0	0
t	0	0	0	0	10	3	16	33	26	19	12	5
e	0	0	10	3	3	5	9	26	28	21	29	22
n	0	0	3	5	0	0	2	19	21	23	22	24

```
Score(match) = 10
Score(mismatch) = -5
Score(gap) = -7
```

Note: this table written top-to-bottom instead of bottom-to-top

More Local Alignment Examples

```
Score(match) = 10
Score(mismatch) = -5
Score(gap) = -7
```

```
local_align("catdogfish", "dog")
        *    c    a    t    d    o    g    f    i    s    h
   *    0    0    0    0    0    0    0    0    0    0    0
   d    0    0    0    0   10    3    0    0    0    0    0
   o    0    0    0    0    3   20   13    6    0    0    0
   g    0    0    0    0    0   13   30   23   16    9    2
```

```
local_align("mississippi", "issp")
        *    m    i    s    s    i    s    s    i    p    p    i
   *    0    0    0    0    0    0    0    0    0    0    0    0
   i    0    0   10    3    0   10    3    0   10    3    0   10
   s    0    0    3   20   13    6   20   13    6    5    0    3
   s    0    0    0   13   30   23   16   30   23   16    9    2
   p    0    0    0    6   23   25   18   23   25   33   26   19
```

```
local_align("aaaa", "aa")
        *    a    a    a    a
   *    0    0    0    0    0
   a    0   10   10   10   10
   a    0   10   20   20   20
```

Local / Global Recap

- Alignment score sometimes called the "edit distance" between two strings.
- Edit distance is sometimes called Levenshtein distance.

- Algorithm for local alignment is sometimes called "Smith-Waterman"
- Algorithm for global alignment is sometimes called "Needleman-Wunsch"

- Same basic algorithm, however.
- Underlies BLAST