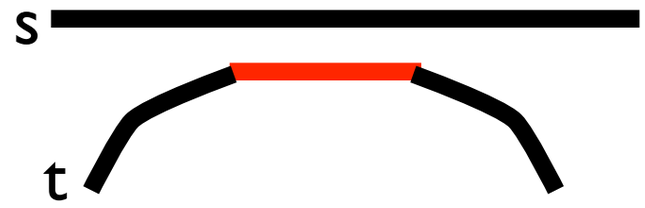
Local Alignment & Gap Penalties

CMSC 423

Global, Semi-global, Local Alignments

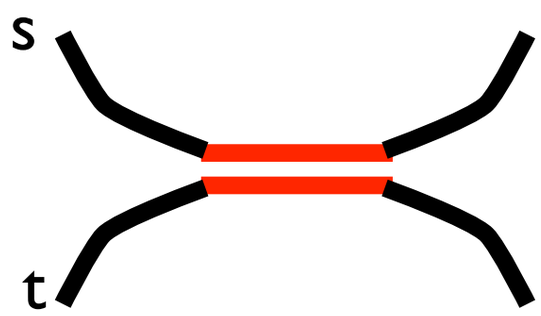
- Last time, we saw a dynamic programming algorithm for *global* alignment: both strings s and t must be completely matched:

- Semiglobal:



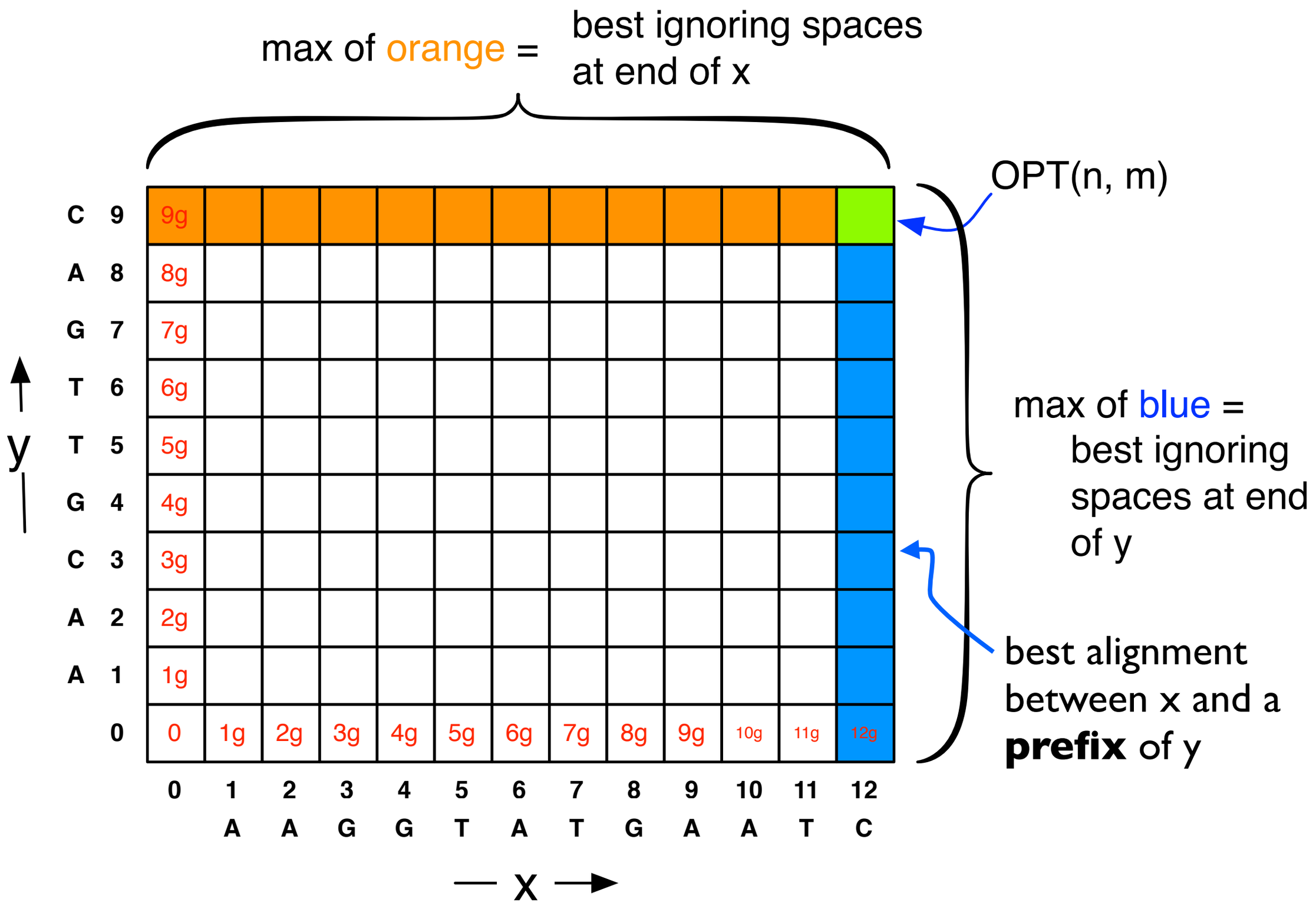
Gaps at starts and ends of some sequences come for free

- Local:

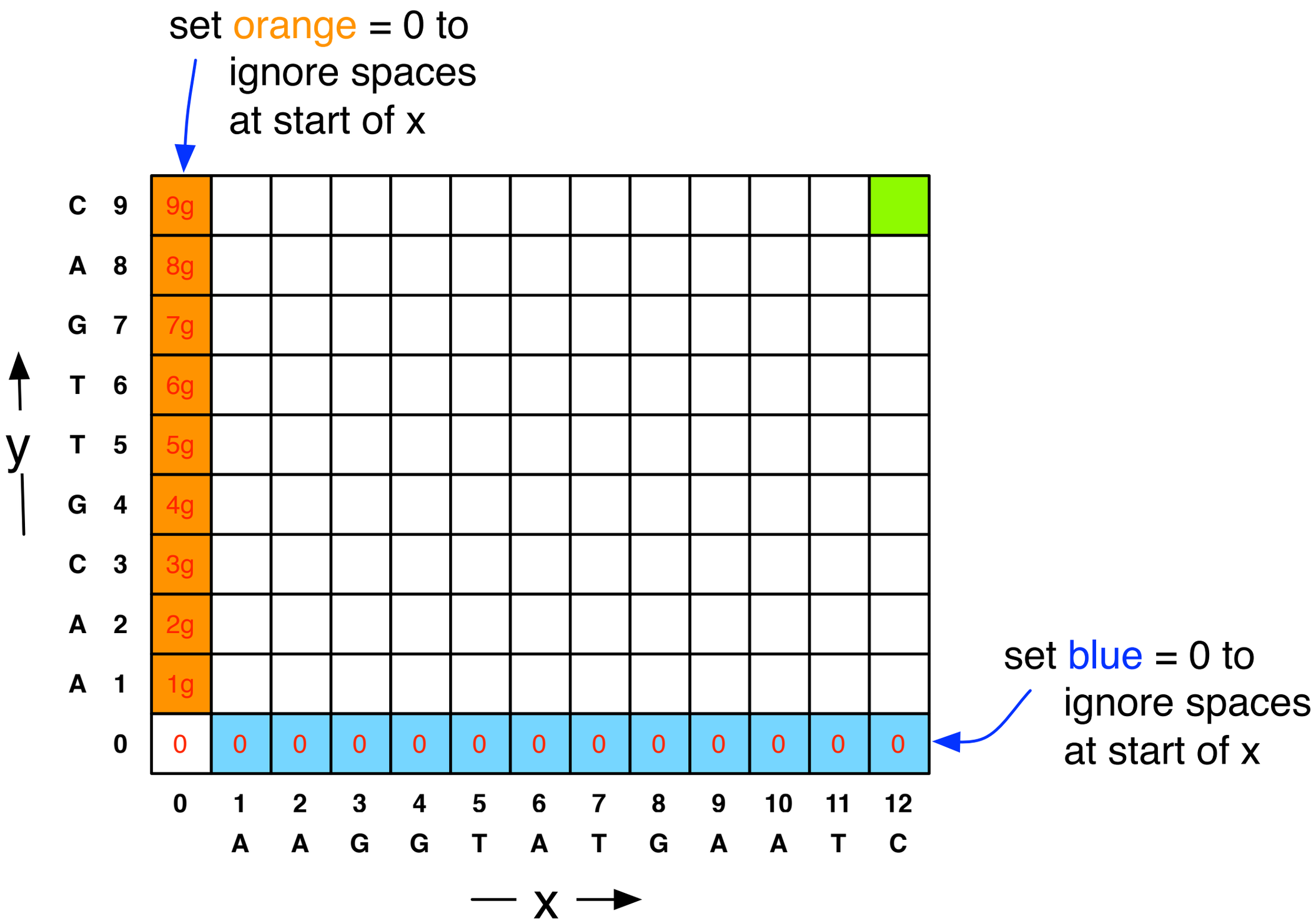


Best alignment between substrings of s and t.

Free gaps at ends of sequences



Free gaps at the start of sequences

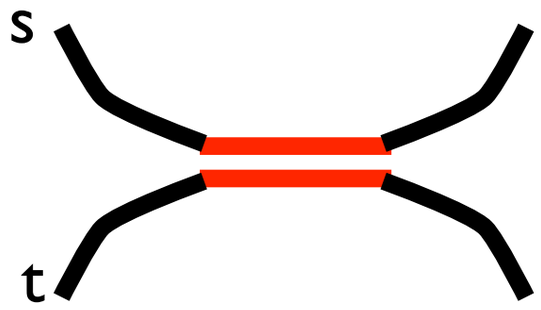


Semiglobal Recap

Free spaces @	**What to do**
end of x	take max of topmost row
end of y	take max of rightmost row
start of x	set bottommost row to 0
start of y	set leftmost row to 0

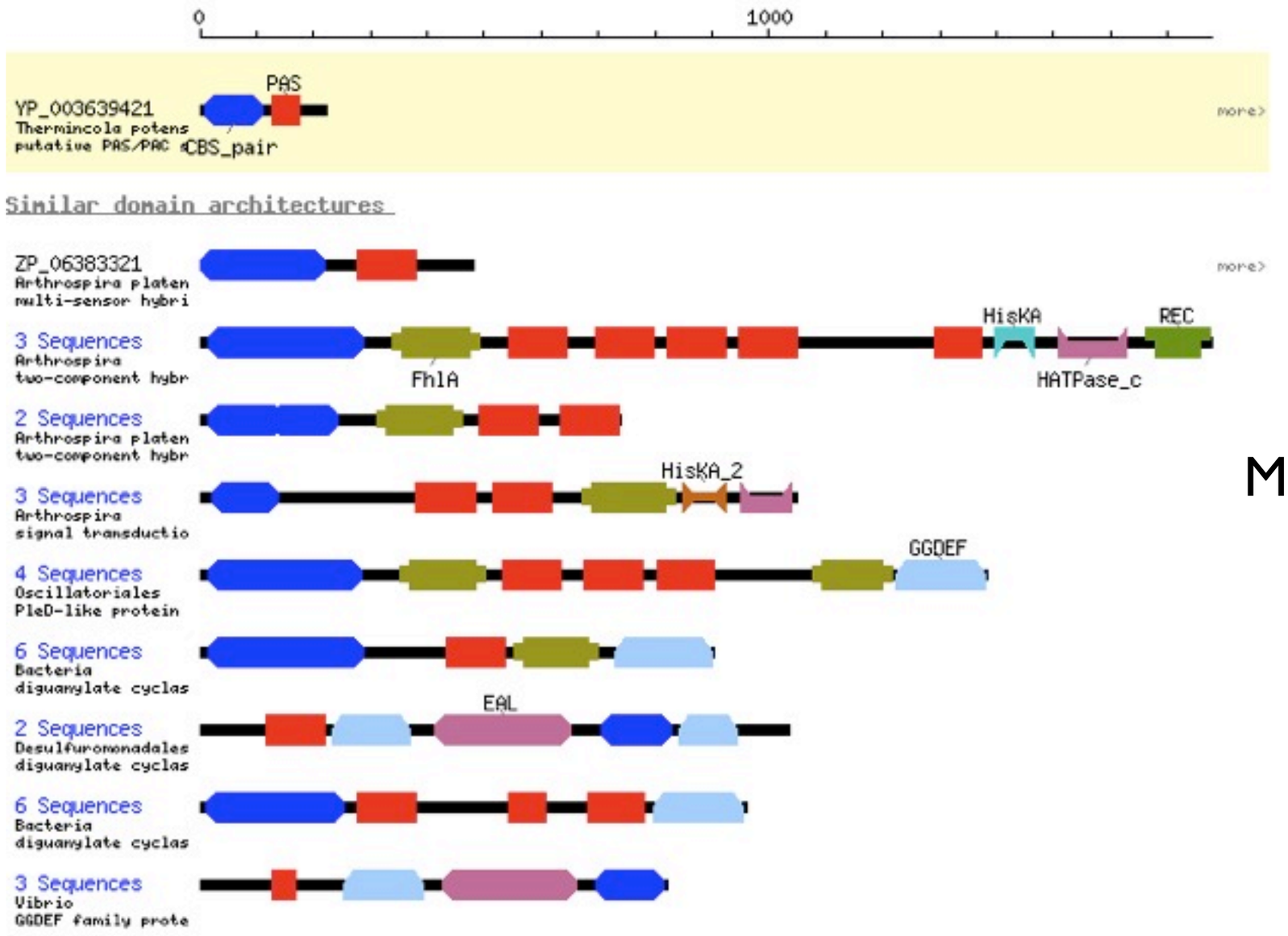
- Can combine these arbitrarily: e.g. to have free spaces at the start of x and both ends of y:

  set bottom- and left-most rows to 0 and take the max of the rightmost row.



Local Alignment

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



Motivation:

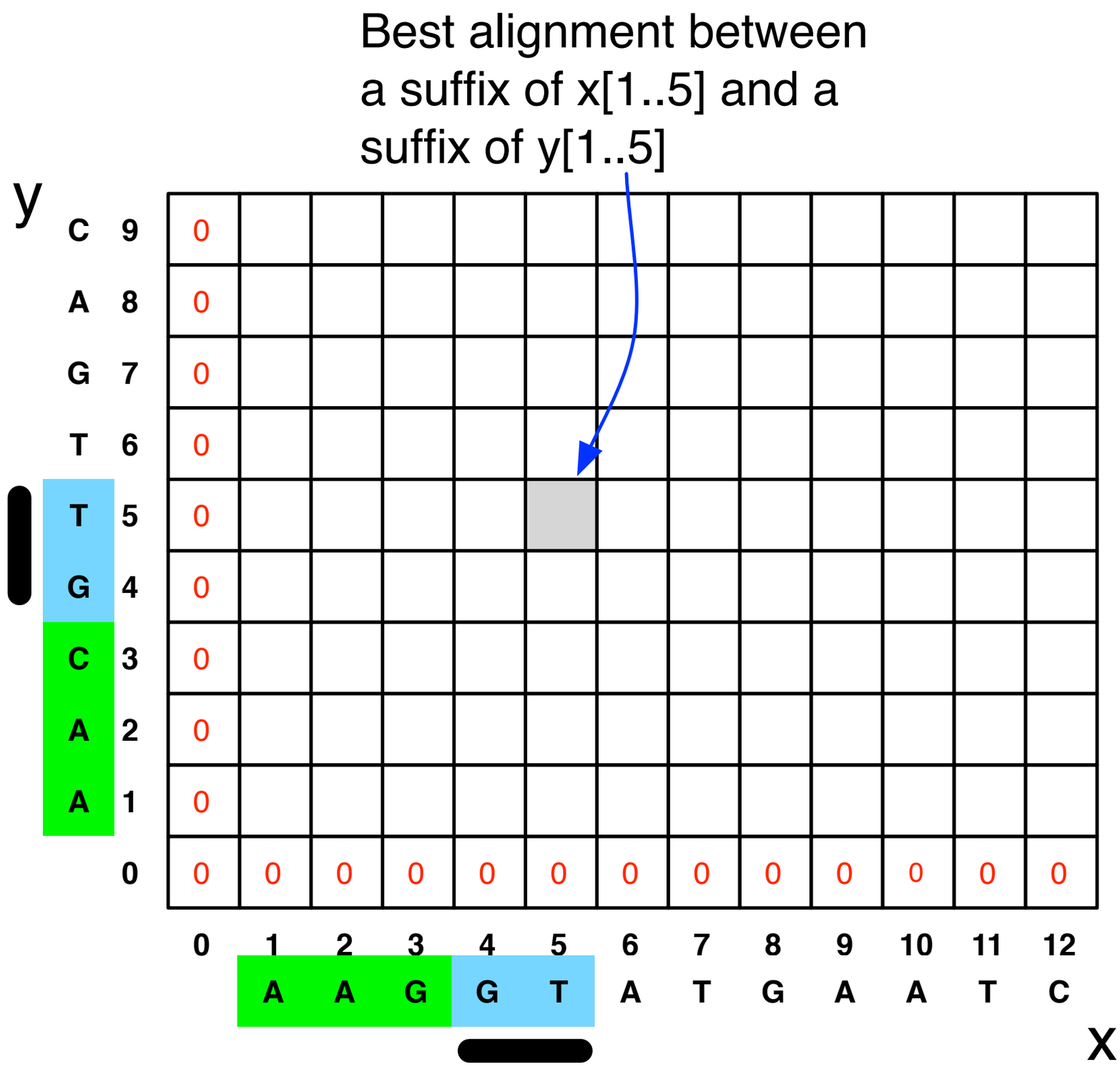
Many genes are composed of *domains*, which are subsequences that perform a particular function.

Local Alignment

New meaning of entry of matrix entry:

A[i, j] = best score between a suffix of s[1...i] and a suffix of t[1...j]

Initialize first row and first column to be 0.



How do we fill in the local alignment matrix?

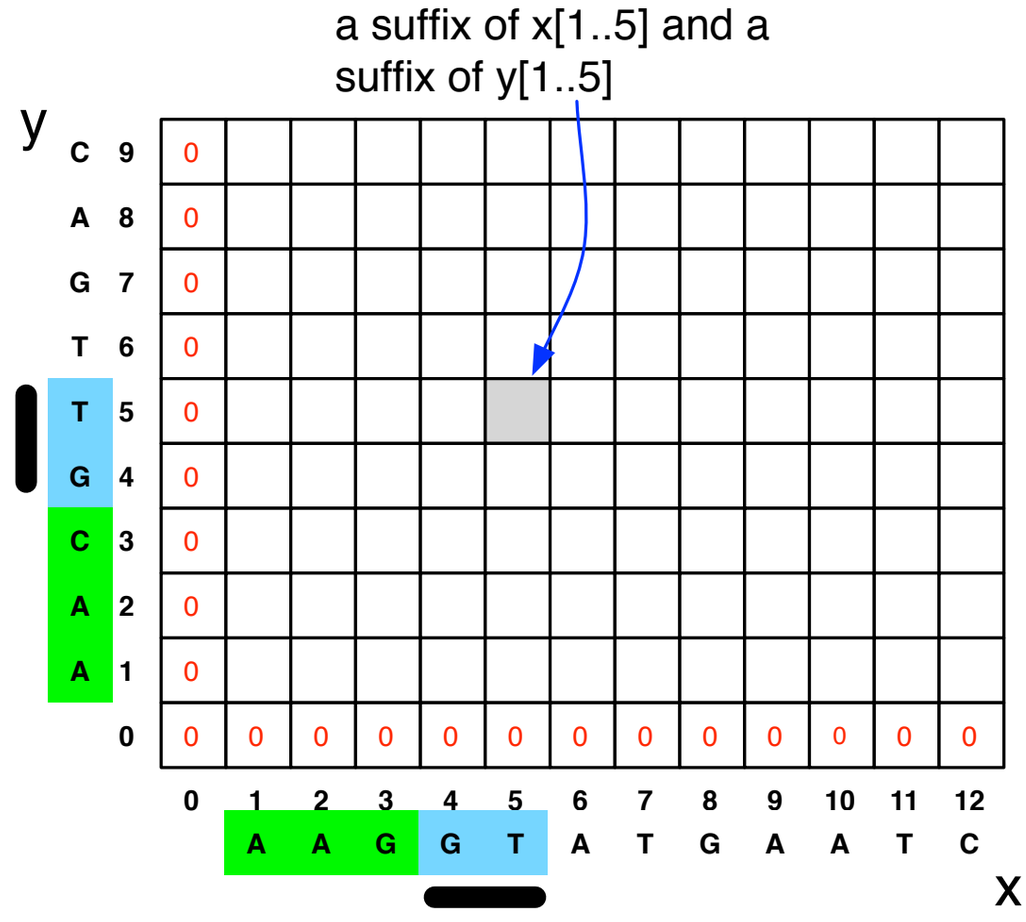
$$A[i,j] = \max \begin{cases} A[i,j-1] + \text{gap} \\ A[i-1,j] + \text{gap} \\ A[i-1,j-1] + \text{match}(i,j) \\ 0 \end{cases}$$

(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

- 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(-2, 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(x)):
        for j in xrange(1, len(y)):

            # the local alignment recurrance 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	8	6	4	2
T	0	0	0	8	6	18	16	14
C	0	0	0	10	8	16	14	12
G	0	0	10	8	20	18	16	24
T	0	0	8	6	18	**30**	28	26
C	0	0	6	18	16	28	26	24

Score(a,a) = 10
Score(a,b) = -5
Score(a,-) = -2

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

More Local Alignment Examples

```
Score(a,a) = 10
Score(a,b) = -5
Score(a,-) = -2
```

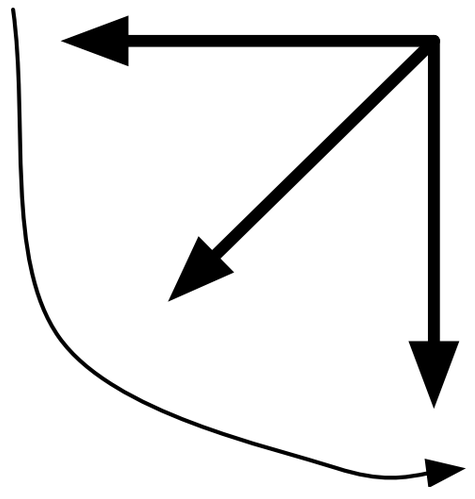
```
local_align("catdogfish", "dog")
      *   c   a   t   d   o   g   f   i   s   h
  *   0   0   0   0   0   0   0   0   0   0   0
  d   0   0   0   0  10   8   6   4   2   0   0
  o   0   0   0   0   8  20  18  16  14  12  10
  g   0   0   0   0   6  18  30  28  26  24  22
```

```
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   8   6  10   8   6  10   8   6  10
  s   0   0   8  20  18  16  20  18  16  14  12  10
  s   0   0   6  18  30  28  26  30  28  26  24  22
  p   0   0   4  16  28  26  24  28  26  38  36  34
```

```
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
```

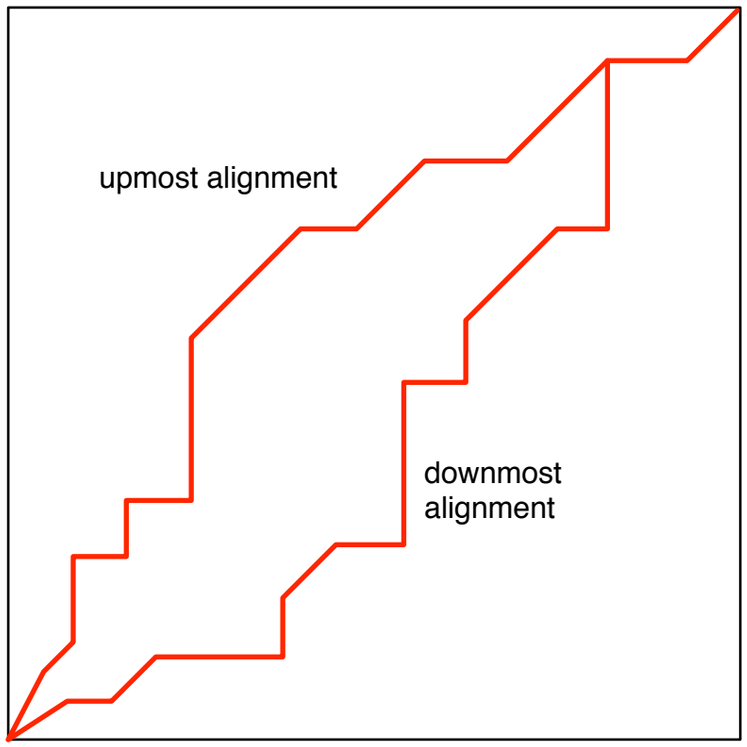
Upmost and Downmost Alignments

When there are ties in the max{}, we have a choice about which arrow to follow.



If we prefer arrows higher in the matrix, we get the *upmost* alignment.

If we prefer arrows lower in the matrix, we get the *downmost* alignment.



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.

General Gap Penalties

```
AAAGAATCCA
A-A-A-T-CA
```

vs.

```
AAAGAATCCA
AAA----TCA
```

These have the same score, but the second one is often more plausible.

A single insertion of "GAAT" into the first string could change it into the second.

- Now the cost of a run of k gaps is GAP × k
- A solution to the problem above is to support general gap penalty, so that the score of a run of k gaps is gap(k) < GAP × k.
- Then, the optimization will prefer to group gaps together.

General Gap Penalties

```
AAAGAATCCA
A-A-A-T-CA
```

vs.

```
AAAGAATCCA
AAA----TCA
```

Previous DP no longer works with general gap penalties because the score of the last character depends on details of the previous alignment:

```
AAAGAA|C
AAA---|T
```

vs.

```
AAAGAAT|C
AAA----|T
```

Instead, we need to "know" how the previous alignment ends in order to give a score to the last subproblem.

Three Matrices

We now keep 3 different matrices:

M[i,j] = score of best alignment of x[1..i] and y[1..j] ending with a character-character **match or mismatch**.

X[i,j] = score of best alignment of x[1..i] and y[1..j] ending with a **space in X**.

Y[i,j] = score of best alignment of x[1..i] and y[1..j] ending with a **space in Y**.

$$M[i,j] = \text{match}(i,j) + \max \begin{cases} M[i-1,j-1] \\ X[i-1,j-1] \\ Y[i-1,j-1] \end{cases}$$

$$X[i,j] = \max \begin{cases} M[i,j-k] - \text{gap}(k) & \text{for } 1 \leq k \leq j \\ Y[i,j-k] - \text{gap}(k) & \text{for } 1 \leq k \leq j \end{cases}$$

$$Y[i,j] = \max \begin{cases} M[i-k,j] - \text{gap}(k) & \text{for } 1 \leq k \leq i \\ X[i-k,j] - \text{gap}(k) & \text{for } 1 \leq k \leq i \end{cases}$$

The M Matrix

We now keep 3 different matrices:

M[i,j] = score of best alignment of x[1..i] and y[1..j] ending with a character-character **match or mismatch**.

X[i,j] = score of best alignment of x[1..i] and y[1..j] ending with a **space in X**.

Y[i,j] = score of best alignment of x[1..i] and y[1..j] ending with a **space in Y**.

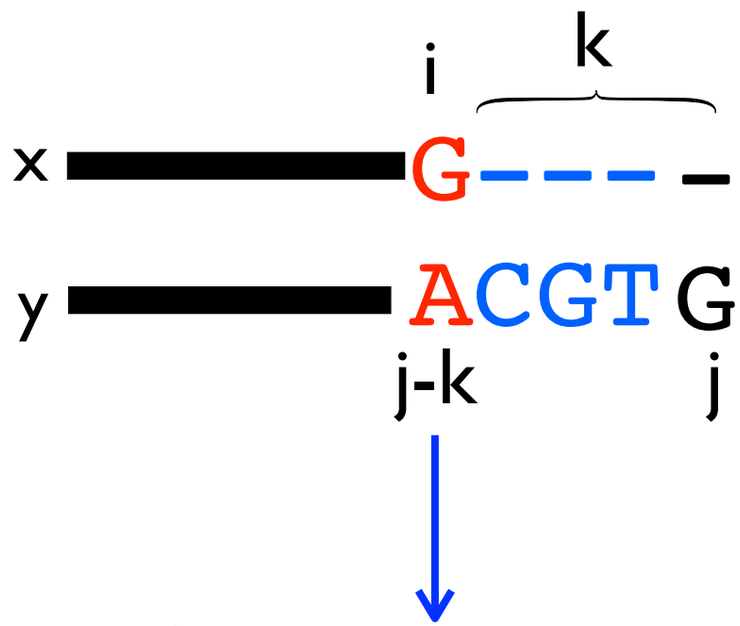
By definition, alignment ends in a match.

$$M[i,j] = \text{match}(i,j) + \max \begin{cases} M[i-1,j-1] \\ X[i-1,j-1] \\ Y[i-1,j-1] \end{cases}$$

Any kind of alignment is allowed before the match.

A
G

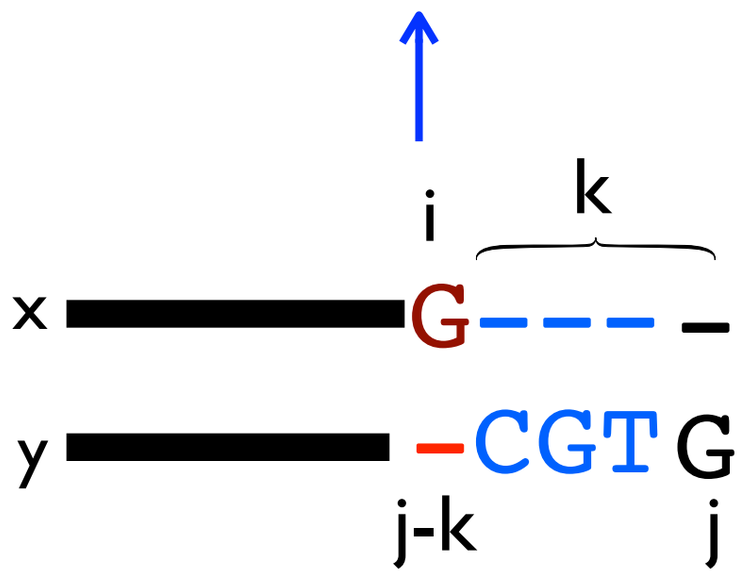
The X (and Y) matrices



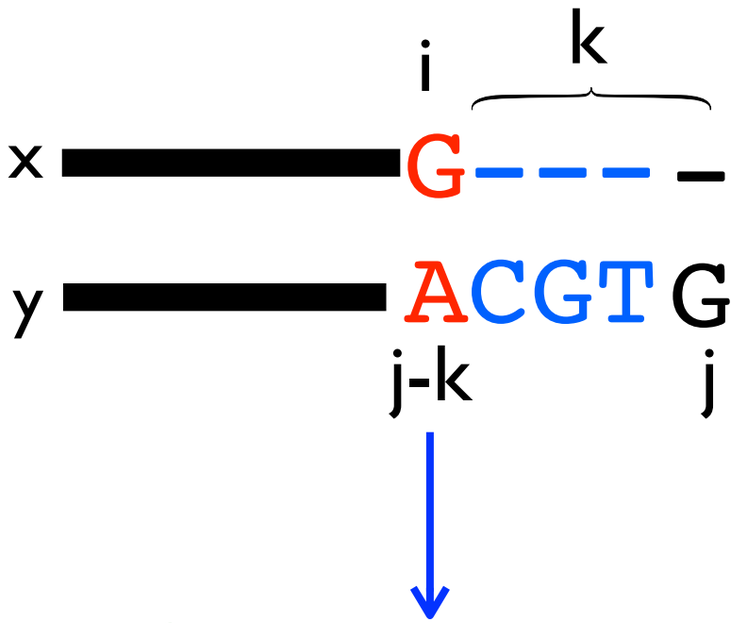
k decides how long to make the gap.

We have to make the whole gap at once in order to know how to score it.

$$X[i,j] = \max \begin{cases} M[i, j-k] - \text{gap}(k) & \text{for } 1 \leq k \leq j \\ Y[i, j-k] - \text{gap}(k) & \text{for } 1 \leq k \leq j \end{cases}$$



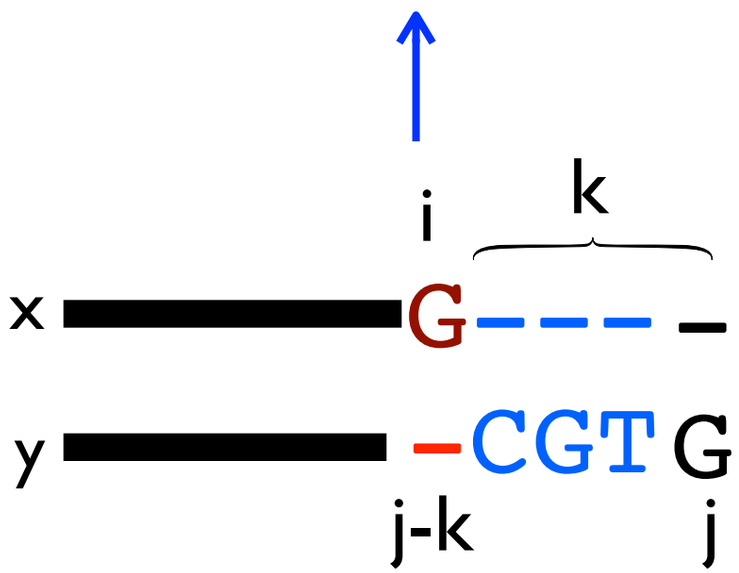
The X (and Y) matrices



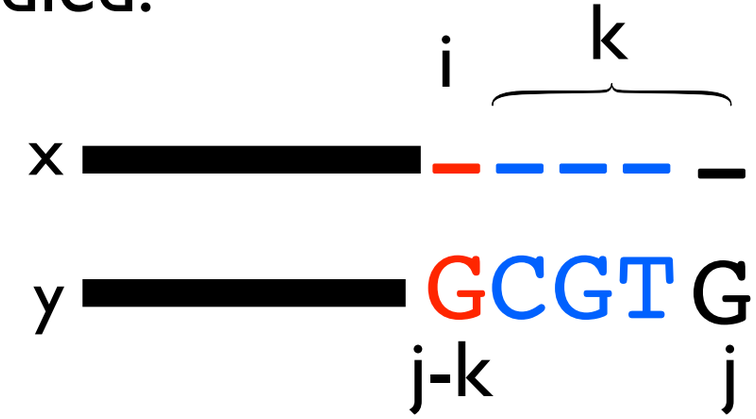
k decides how long to make the gap.

We have to make the whole gap at once in order to know how to score it.

$$X[i,j] = \max \begin{cases} M[i, j-k] - \text{gap}(k) & \text{for } 1 \leq k \leq j \\ Y[i, j-k] - \text{gap}(k) & \text{for } 1 \leq k \leq j \end{cases}$$



This case is automatically handled.



Running Time for Gap Penalties

$$M[i,j] = \text{match}(i,j) + \max \begin{cases} M[i-1,j-1] \\ X[i-1,j-1] \\ Y[i-1,j-1] \end{cases}$$

$$X[i,j] = \max \begin{cases} M[i,j-k] - \text{gap}(k) & \text{for } 1 \le k \le j \\ Y[i,j-k] - \text{gap}(k) & \text{for } 1 \le k \le j \end{cases}$$

$$Y[i,j] = \max \begin{cases} M[i-k,j] - \text{gap}(k) & \text{for } 1 \le k \le i \\ X[i-k,j] - \text{gap}(k) & \text{for } 1 \le k \le i \end{cases}$$

Final score is max {M[n,m], X[n,m], Y[n,m]}.

How do you do the traceback?

Runtime:

- Assume |X| = |Y| = n for simplicity: $3n^2$ subproblems
- $2n^2$ subproblems take O(n) time to solve (because we have to try all k)

$\Rightarrow$ $O(n^3)$ total time

Affine Gap Penalties

- $O(n^3)$ for general gap penalties is usually too slow...

- We can still encourage spaces to group together using a special case of general penalties called *affine gap penalties*:

  gap_start = the cost of starting a gap

  gap_extend = the cost of extending a gap by one more space

- Same idea of using 3 matrices, but now we don't need to search over all gap lengths, we just have to know whether we are starting a new gap or not.

Affine Gap Penalties

$$M[i,j] = \text{match}(i,j) + \max \begin{cases} M[i-1,j-1] \\ X[i-1,j-1] \\ Y[i-1,j-1] \end{cases}$$

match between x and y

If previous alignment ends in match, this is a new gap

$$X[i,j] = \max \begin{cases} \text{gap\_start} + \text{gap\_extend} + M[i,j-1] \\ \text{gap\_extend} + X[i,j-1] \\ \text{gap\_start} + \text{gap\_extend} + Y[i,j-1] \end{cases}$$

gap in x

$$Y[i,j] = \max \begin{cases} \text{gap\_start} + \text{gap\_extend} + M[i-1,j] \\ \text{gap\_start} + \text{gap\_extend} + X[i-1,j] \\ \text{gap\_extend} + Y[i-1,j] \end{cases}$$

gap in y

Affine Gap Runtime

- $3n^2$ subproblems
- Each one takes constant time
- Total runtime $O(n^2)$, back to the run time of the basic running time.

Recap

- Semiglobal alignment: 0 initial columns or take maximums over last row or column.

- local alignment: extra "0" case.

- General gap penalties require 3 matrices and $O(n^3)$ time.
- Affine gap penalties require 3 matrices, but only $O(n^2)$ time.