String Matching

CS 4364/5364

Exact String Matching

- Given string *P*, called the pattern, and a longer string *T*, called the text, the **exact matching** problem is to find all occurrences, if any, of *P* in *T*.
- Example:
  - *P* = "`aba`", *T* = "`bbabaxababay`"
  - *P* occurs in *T* at positions: 3, 7, & 9
  - Note, that 2 occurrences overlap

The naive method

```
bbabaxababay
```

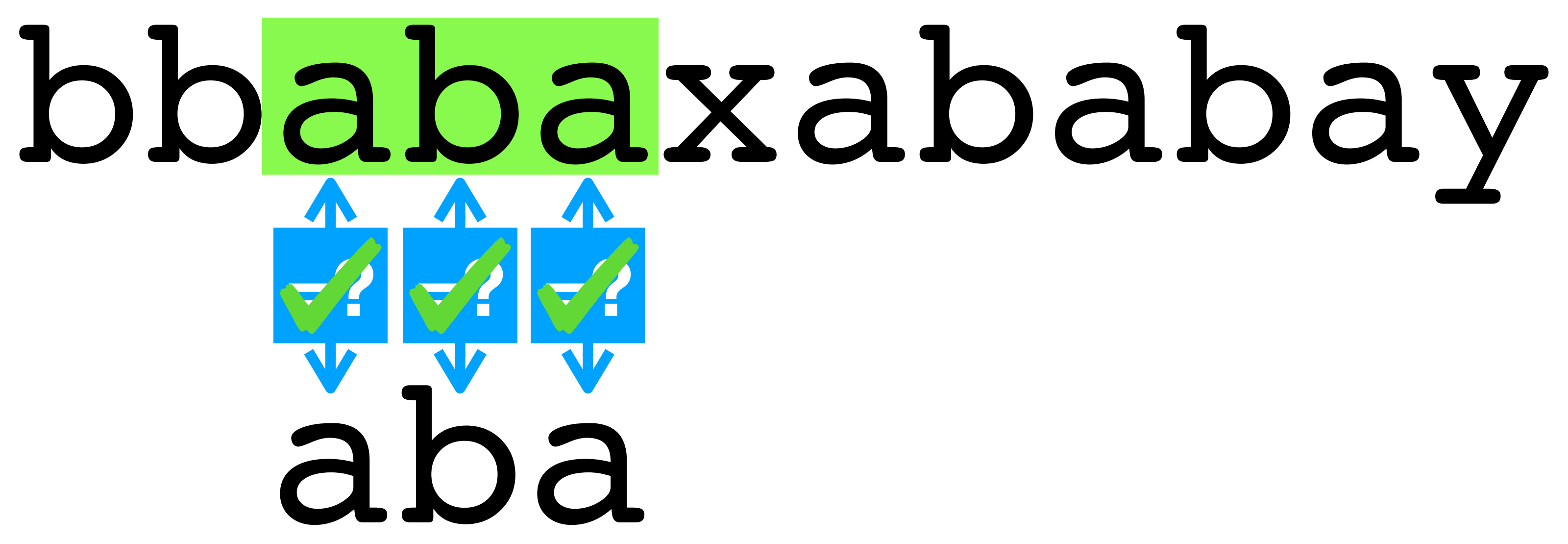
$\updownarrow \cancel{\overset{?}{=}}$

```
aba
```

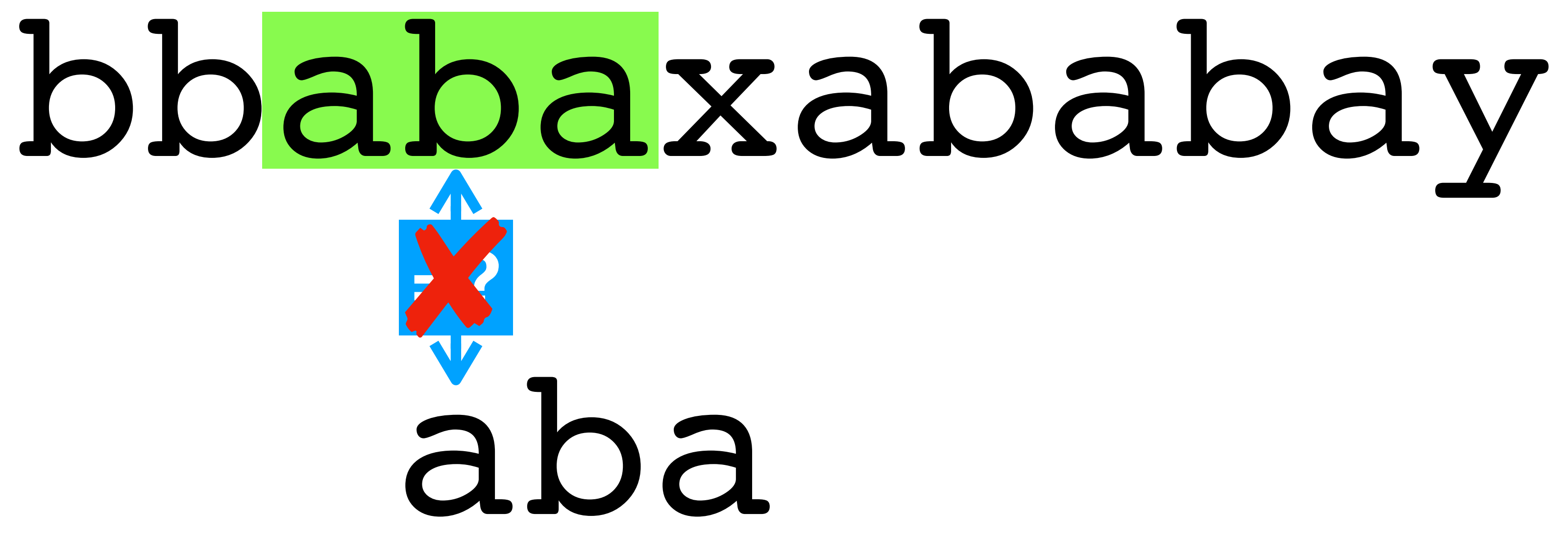
The naive method

```
bbabaxababay
 aba
```

The naive method

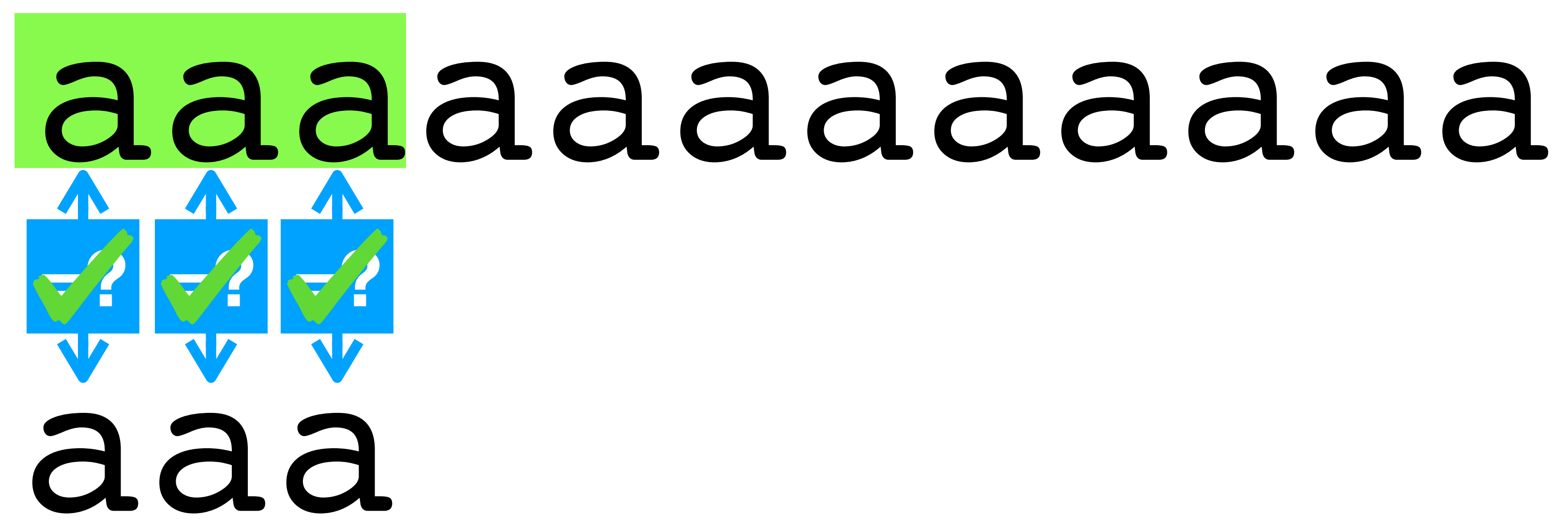


The naive method



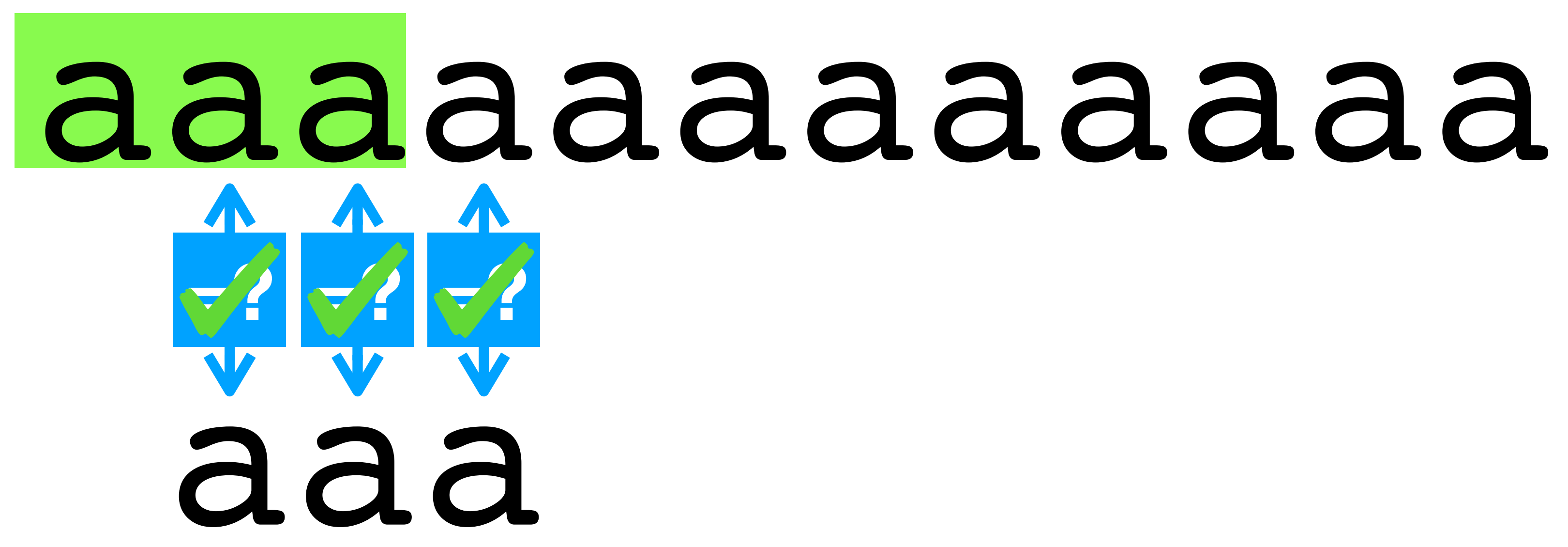
The naive method

Another example



The naive method

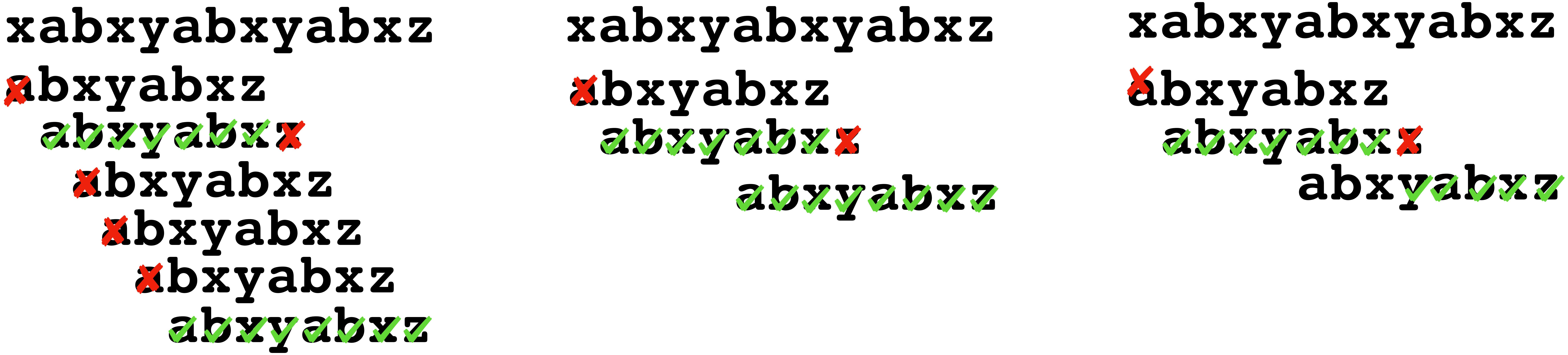
Another example



How many comparisons would the naive algorithm do on this example?

What did we already know about this comparison before we started?

Example of faster methods



If we knew ahead of time the comparisons we could skip (based on where in P the mismatches are), we could decrease the running time!

Preprocessing

Definition Given a string *S* and a position $i>1$, let $\boldsymbol{M_i(S)}$ be the length of the longest substring if *S* that matches a prefix of *S.*

that is, $M_i(S)=j$ if $S[1...j] = S[i...(i+j-1)]$ (and $S[1...(j+1)] \neq S[i...(i+j)]$)

for *S* = "`aabcaabxaaz`"

$M_5(S) = 3$

$M_6(S) = 1$

$M_7(S) = M_8(S) = 0$

$M_9(S) = 2$

Calculation of all M_i can be done in $O(n)$ time (where $|S| = n$)

Linear-time exact sequence matching

Algorithm

- Construct a new sequence *S* = *P$T* where $ is a letter thats not in either of the two strings
- Compute all of the $M_i(S)$ values
- Do a linear scan to find and return all positions $i > |P|$ where $M_i(S) == P$

Do we need to save $M_{(n+5)}$?
(where $|P| = n$)

Boyer-Moore (at a high level)

- Worst-case running time is still *O(m+n)*, but in practice it is sub-linear
- Re-think the problem and look at suffixes of the pattern, then use some preprocessing information to skip the pattern ahead by more than 1

Boyer-Moore (at a high level)

- The bad character rule
  - when matching (from the right) and a mismatch is found (say *x* in *T* ≠ *y* in *P*)
  - shift the pattern so that the next instance of *x* in *P* that is to the left of the current position is at the current position in *T*

xabxyabxyabxz

 abxyabx~~x~~

Does this skip any instances of *P* in *T*?

Boyer-Moore (at a high level)

- The good suffix rule
  - suppose a substring of *T*, *t* matches a suffix of *P* but a mismatch occurs at the next character
  - find the right most copy of *t* in *P* such that the preceding character is different
  - shift *P* so that this substring is matched with *t*

```
prstabstubabvqxrst
  qcabdabdab
```

Does this skip any instances of *P* in *T*?

Boyer-Moore

1. given *P* and *T* (s.t. $|P|=n$, $|T|=m$)
2. let $k=n$
3. compare *P* and *T* starting with *P[n]* and *T[k]*
4. if no mismatch is found:
    - report a solution
    - increment *k* as much as possible
5. otherwise: increment k using the **good suffix** & **bad character** rules
6. go to (2) while $k < m$

The good suffix rule

- suppose a substring of *T*, *t* matches a suffix of *P* but a mismatch occurs at the next character
- find the right most copy of *t* in *P* such that the preceding character is different
- shift *P* so that this substring is matched with *t*

The bad character rule

- when matching (from the right) and a mismatch is found
  (say *x* in $T \neq y$ in *P*)
- shift the pattern so that the next instance of *x* in *P* that is to the left of the current position is at the current position in *T*

Other methods

- Knuth-Morris-Pratt (KMP)
  - able to work with *T* arriving **online**
  - running time is still *O(m+n)*
  - uses pattern suffixes (closer to suffix tree which we will see next)
- Aho-Corasick
  - efficiently finds all occurrences of a **set** of patterns
  - puts patterns into a tree to search all at once
- Karp-Rabin
  - uses bit operations in place of character comparisons
  - structure similar to on-hot-encodings to represent strings

Edit Distance

- Up to now, need the exact string in a set
- What if I wanted to know how similar two strings are?
- Problem: Given strings S_1 and S_2 what is the minimum number of edits (insertions, deletions, replacements) needed to convert S_1 into S_2?
- Example: S_1 = **`baseball`** & S_2 = **`ballcap`**.
  -
    ```
      RRR DR
    baseball
    ballca p
    ```
  - 5 operations: change `s`→`l`, e→`l`, `b`→`c`, delete `l`, `l`→`p`

Global Alignment Problem

- An **alignment** of two sequences is formed by inserting gap characters,'-', in arbitrary locations along the sequences so that they end up wit the same length and there are no two spaces at the same position of the two augmented strings.

```
baseball        baseball---        baseball
-ballcap        ----ballcap        ballca-p
```

How do we know which one of these is best?

Alignment

- In general, associate a similarity score with each pair of aligned characters:
  - for characters $x,y \in \Sigma \cup \{-\}$, let $\delta(x,y)$ be the similarity of x and y
- Let the score, Δ, of an alignment, $A = (S'_1,S'_2)$, be defined as $\Delta(A) =: \sum_{1 \le i \le |S'_1|} \delta(S'_1[i], S'_s[i])$
- Goal of alignment is to maximize that sum

	-	a	b	c	e	l	p	s
-		-1	-1	-1	-1	-1	-1	-1
a	-1	0	-1	-1	-1	-1	-1	-1
b	-1	-1	0	-1	-1	-1	-1	-1
c	-1	-1	-1	0	-1	-1	-1	-1
e	-1	-1	-1	-1	0	-1	-1	-1
l	-1	-1	-1	-1	-1	0	-1	-1
p	-1	-1	-1	-1	-1	-1	0	-1
s	-1	-1	-1	-1	-1	-1	-1	0

```
baseball
ballca-p
```

Alignment

- In general, associate a similarity score with each pair of aligned characters:
    - for characters $x,y \in \Sigma \cup \{-\}$, let $\delta(x,y)$ be the similarity of x and y
- Let the score, Δ, of an alignment, $A = (S'_1, S'_2)$, be defined as $\Delta(A) =: \sum_{1 \le i \le |S'_1|} \delta(S'_1[i], S'_s[i])$
- Goal of alignment is to maximize that sum

	-	a	b	c	e	l	p	s
-		0	0	0	0	0	0	0
a	0	2	-1	-1	-1	-1	-1	-1
b	0	-1	2	-1	-1	-1	-1	-1
c	0	-1	-1	2	-1	-1	-1	-1
e	0	-1	-1	-1	2	-1	-1	-1
l	0	-1	-1	-1	-1	2	-1	-1
p	0	-1	-1	-1	-1	-1	2	-1
s	0	-1	-1	-1	-1	-1	-1	2

```
baseball---
----ballcap
```

How can we compute an alignment that optimizes Δ?

Needleman-Wunsch

- brute-force, compute all possible alignments and score them, would take exponential time to compute the optimal alignment
- using dynamic programming Needleman and Wunsch [1970] found that the optimal alignment can be computed in *O(mn)*-time.

Needleman-Wunsch

- Dynamic programming, generally, works by solving sub-problems, storing the results, and combining the solution to (most times) multiple sub-problems to find the answer.
- Given two strings *S[1...n]* and *T[1...m]*, find the best alignment.

Needleman-Wunsch

- Dynamic programming, generally, works by solving sub-problems, storing the results, and combining the solution to (most times) multiple sub-problems to find the answer.

- Given two strings *S[1...n]* and *T[1...m]*, find the best alignment given the best alignments of:
  - *S[1...(n-1)]* and *T[1...m],*
  - *S[1...n]* and *T[1...(m-1)],* and
  - *S[1...(n-1)]* and *T[1...(m-1)]*

Needleman-Wunsch

- Define an *nxm* array *V*, the cell *V(i,j)* will hold the score of the best sub alignments of *S[1...i]* and *T[1...j]*

- The recurrence relation (the base of any DP)

$$V(i,j) = \max \begin{cases} V(i-1,j-1) + \delta(S[i],T[i]) & \text{match/mismatch} \\ V(i-1,j) + \delta(S[i],-) & \text{delete} \\ V(i,j-1) + \delta(-,T[j]) & \text{insert} \end{cases}$$

- The initialization is:
  $V(0,0) = 0$
  $V(0,j) = V(0,j-1) + \delta(-,T[j])$
  $V(i,0) = V(i-1,0) + \delta(S[i],-)$

Optimal alignment score is in *V(n,m)*

Needleman-Wunch

$\delta(-,x) = -1$ for $x \in \Sigma$

$\delta(x,-) = -1$ for $x \in \Sigma$

$\delta(x,y) = 1$ for $y = x$

$\delta(x,y) = -1$ for $y \neq x$

		A	A	C	C	C	G
	0						
A	-1						
A							
G							
G							
C							
C							

The cost of the optimal alignment of "A" and ""

Needleman-Wunch

$\delta(-,x) = -1$ for $x \in \Sigma$

$\delta(x,-) = -1$ for $x \in \Sigma$

$\delta(x,y) = 1$ for $y = x$

$\delta(x,y) = -1$ for $y \neq x$

		A	A	C	C	C	G
	0						
A	-1						
A	-2						
G							
G							
C							
C							

The cost of the optimal alignment of "AA" and ""

Needleman-Wunch

$\delta(-,x) = -1$ for $x \in \Sigma$

$\delta(x,-) = -1$ for $x \in \Sigma$

$\delta(x,y) = 1$ for $y = x$

$\delta(x,y) = -1$ for $y \neq x$

		A	A	C	C	C	G
	0	-1					
A	-1						
A	-2						
G	-3						
G	-4						
C	-5						
C	-6						

The cost of the optimal alignment of "" and "A"

Needleman-Wunch

$\delta(-,x) = -1$ for $x \in \Sigma$

$\delta(x,-) = -1$ for $x \in \Sigma$

$\delta(x,y) = 1$ for y = x

$\delta(x,y) = -1$ for y ≠ x

		A	A	C	C	C	G
	0	-1	-2	-3	-4	-5	-6
A	-1						
A	-2						
G	-3						
G	-4						
C	-5						
C	-6						

The cost of the optimal alignment of "A" and "A"

Needleman-Wunch

$\delta(-,x) = -1$ for $x \in \Sigma$

$\delta(x,-) = -1$ for $x \in \Sigma$

$\delta(x,y) = 1$ for $y = x$

$\delta(x,y) = -1$ for $y \neq x$

$V(0,0) + \delta(A,A) = 1$

$$+ \begin{array}{c} \mathtt{A} \\ \mathtt{A} \end{array}$$

$V(0,1) + \delta(A,-) = -2$

$$\begin{array}{c} - \\ \mathtt{A} \end{array} + \begin{array}{c} \mathtt{A} \\ - \end{array}$$

$V(1,0) + \delta(-,A) = -2$

$$\begin{array}{c} \mathtt{A} \\ - \end{array} + \begin{array}{c} - \\ \mathtt{A} \end{array}$$

$$V(i,j) = \max \begin{cases} V(i-1,j-1) + \delta(S[i],T[i]) & \text{match/mismatch} \\ V(i-1,j) + \delta(S[i],-) & \text{delete} \\ V(i,j-1) + \delta(-,T[j]) & \text{insert} \end{cases}$$

		A	A	C	C	C	G
	0	-1	-2	-3	-4	-5	-6
A	-1	1					
A	-2						
G	-3						
G	-4						
C	-5						
C	-6						

Needleman-Wunch

$\delta(-,x) = -1$ for $x \in \Sigma$

$\delta(x,-) = -1$ for $x \in \Sigma$

$\delta(x,y) = 1$ for $y = x$

$\delta(x,y) = -1$ for $y \neq x$

		A	A	C	C	C	G
	0	-1	-2	-3	-4	-5	-6
A	-1	1	$+\delta(A,A)$ $+\delta(-,A)$ $+\delta(A,-)$ 0				
A	-2						
G	-3						
G	-4						
C	-5						
C	-6						

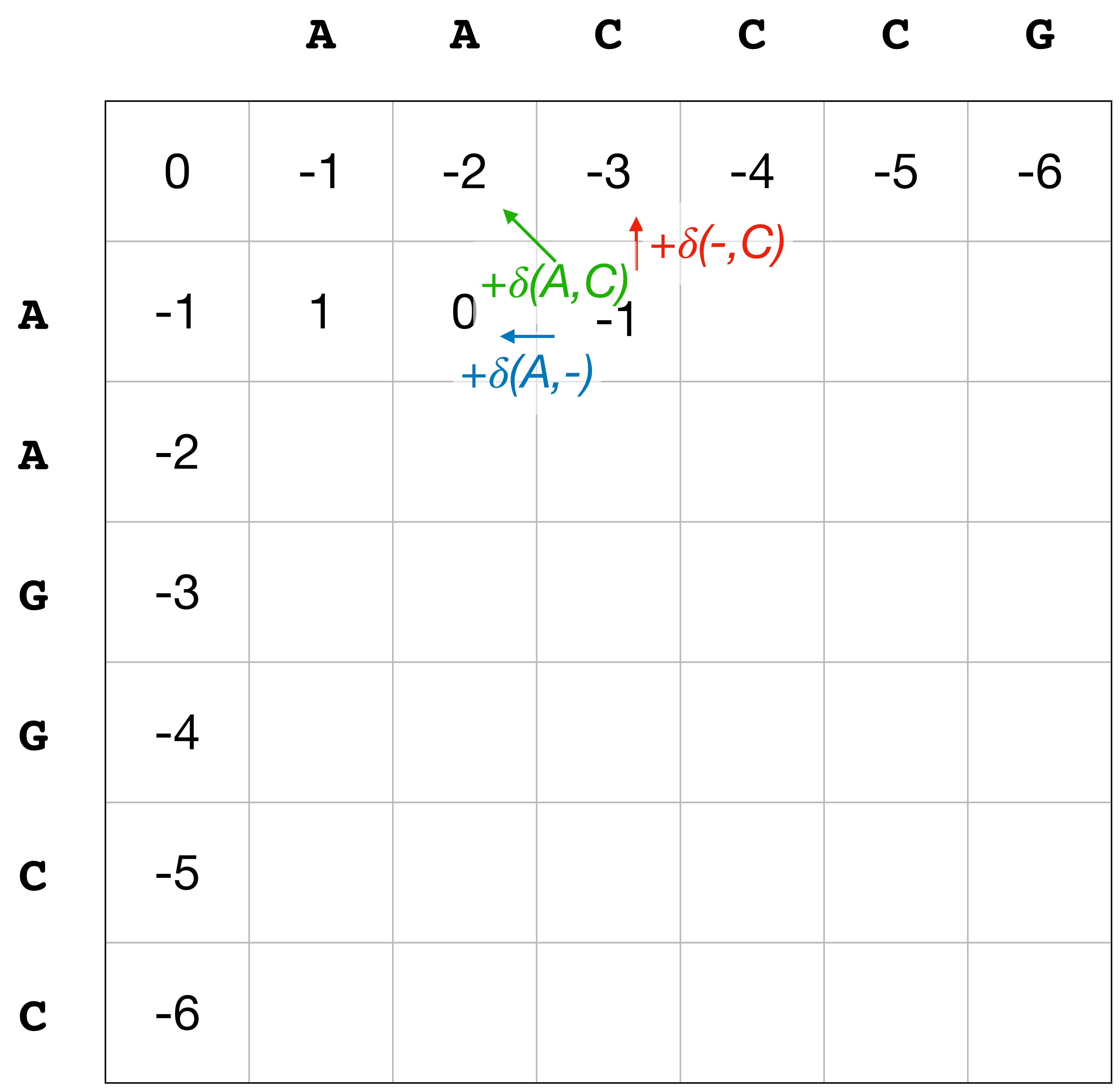
Needleman-Wunch

$\delta(-,x) = -1$ for $x \in \Sigma$

$\delta(x,-) = -1$ for $x \in \Sigma$

$\delta(x,y) = 1$ for $y = x$

$\delta(x,y) = -1$ for $y \neq x$



		A	A	C	C	C	G
	0	-1	-2	-3	-4	-5	-6
A	-1	1	0	-1			
A	-2						
G	-3						
G	-4						
C	-5						
C	-6						

Needleman-Wunch

$\delta(-,x) = -1$ for $x \in \Sigma$

$\delta(x,-) = -1$ for $x \in \Sigma$

$\delta(x,y) = 1$ for $y = x$

$\delta(x,y) = -1$ for $y \neq x$

		A	A	C	C	C	G
	0	-1	-2	-3	-4	-5	-6
A	-1	1	0	-1	-2	-3	-4
A	-2	0	2	1	0	-1	-2
G	-3	-1	1	1	0	-1	0
G	-4	-2	0	0	0	-1	0
C	-5	-3	-1	1	1	1	0
C	-6	-4	-2	0	2	2	1

Needleman-Wunch

$\delta(-,x) = -1$ for $x \in \Sigma$

$\delta(x,-) = -1$ for $x \in \Sigma$

$\delta(x,y) = 1$ for $y = x$

$\delta(x,y) = -1$ for $y \neq x$

$$
\begin{array}{c|ccccccc}
 & & \mathtt{A} & \mathtt{A} & \mathtt{C} & \mathtt{C} & \mathtt{C} & \mathtt{G} \\
\hline
 & 0 & -1 & -2 & -3 & -4 & -5 & -6 \\
\mathtt{A} & -1 & 1 & & & & & \\
\mathtt{A} & -2 & & & & & & \\
\mathtt{G} & -3 & & & & & & \\
\mathtt{G} & -4 & & & & & & \\
\mathtt{C} & -5 & & & & & & \\
\mathtt{C} & -6 & & & & & & \\
\end{array}
$$

$\nwarrow +\delta(A,A)$

$\uparrow +\delta(-,A)$

$\leftarrow +\delta(A,-)$

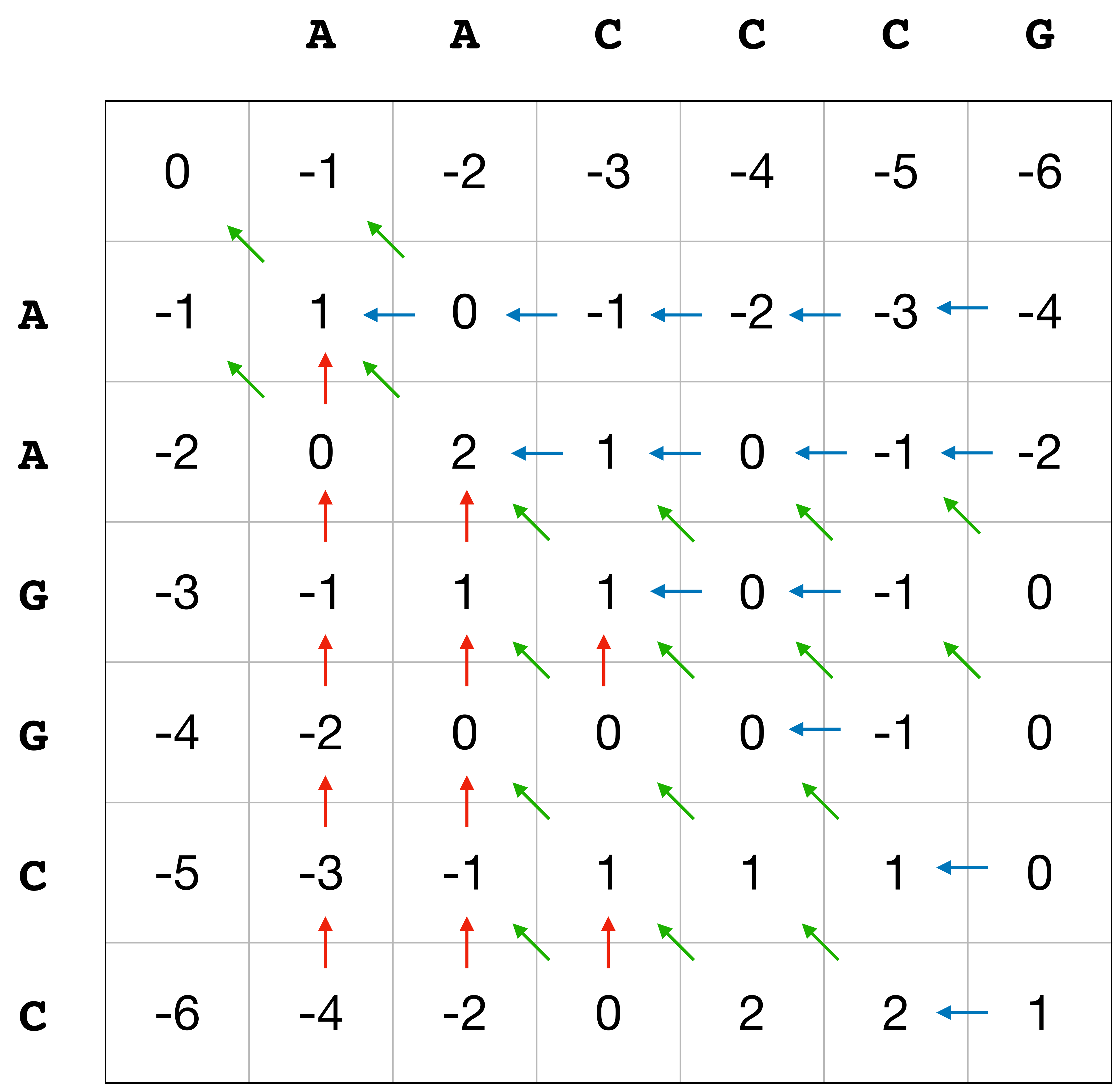
Needleman-Wunch

$\delta(-,x) = -1$ for $x \in \Sigma$

$\delta(x,-) = -1$ for $x \in \Sigma$

$\delta(x,y) = 1$ for $y = x$

$\delta(x,y) = -1$ for $y \neq x$



		A	A	C	C	C	G
	0	-1	-2	-3	-4	-5	-6
A	-1	↖ 1	↖ ← 0	← -1	← -2	← -3	← -4
A	-2	↖ ↑ 0	↖ 2	← 1	← 0	← -1	← -2
G	-3	↑ -1	↑ 1	↖ 1	↖ ← 0	↖ ← -1	↖ 0
G	-4	↑ -2	↑ 0	↖ ↑ 0	↖ 0	↖ ← -1	↖ 0
C	-5	↑ -3	↑ -1	↖ 1	↖ 1	↖ 1	← 0
C	-6	↑ -4	↑ -2	↖ ↑ 0	↖ 2	↖ 2	← 1

Needleman-Wunch

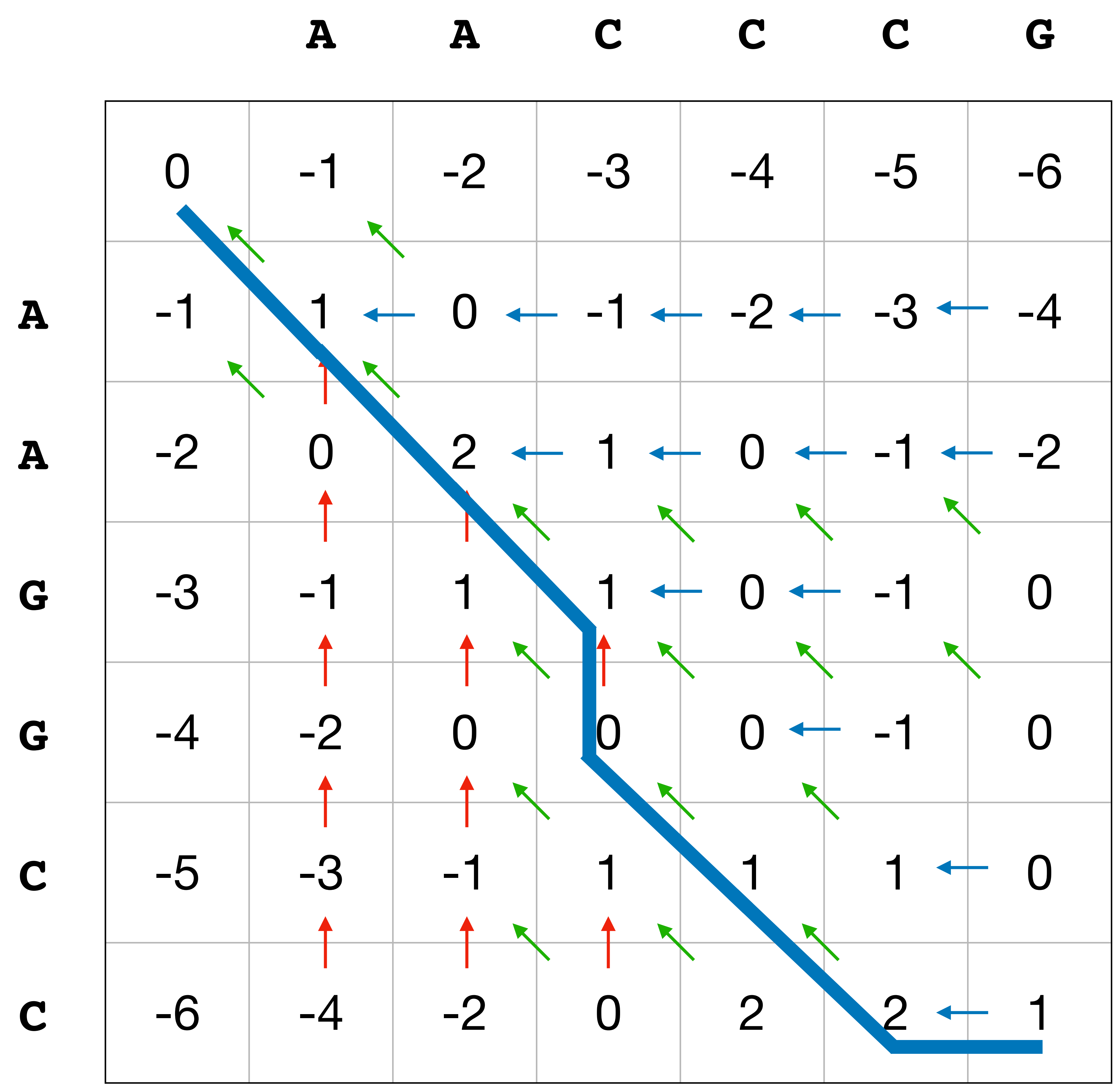
$\delta(-,x) = -1$ for $x \in \Sigma$

$\delta(x,-) = -1$ for $x \in \Sigma$

$\delta(x,y) = 1$ for $y = x$

$\delta(x,y) = -1$ for $y \neq x$

```
A A C - C C G
A A G G C C -
```



		A	A	C	C	C	G
	0	-1	-2	-3	-4	-5	-6
A	-1	1	0	-1	-2	-3	-4
A	-2	0	2	1	0	-1	-2
G	-3	-1	1	1	0	-1	0
G	-4	-2	0	0	0	-1	0
C	-5	-3	-1	1	1	1	0
C	-6	-4	-2	0	2	2	1

Needleman-Wunch

$\delta(-,x) = -1$ for $x \in \Sigma$

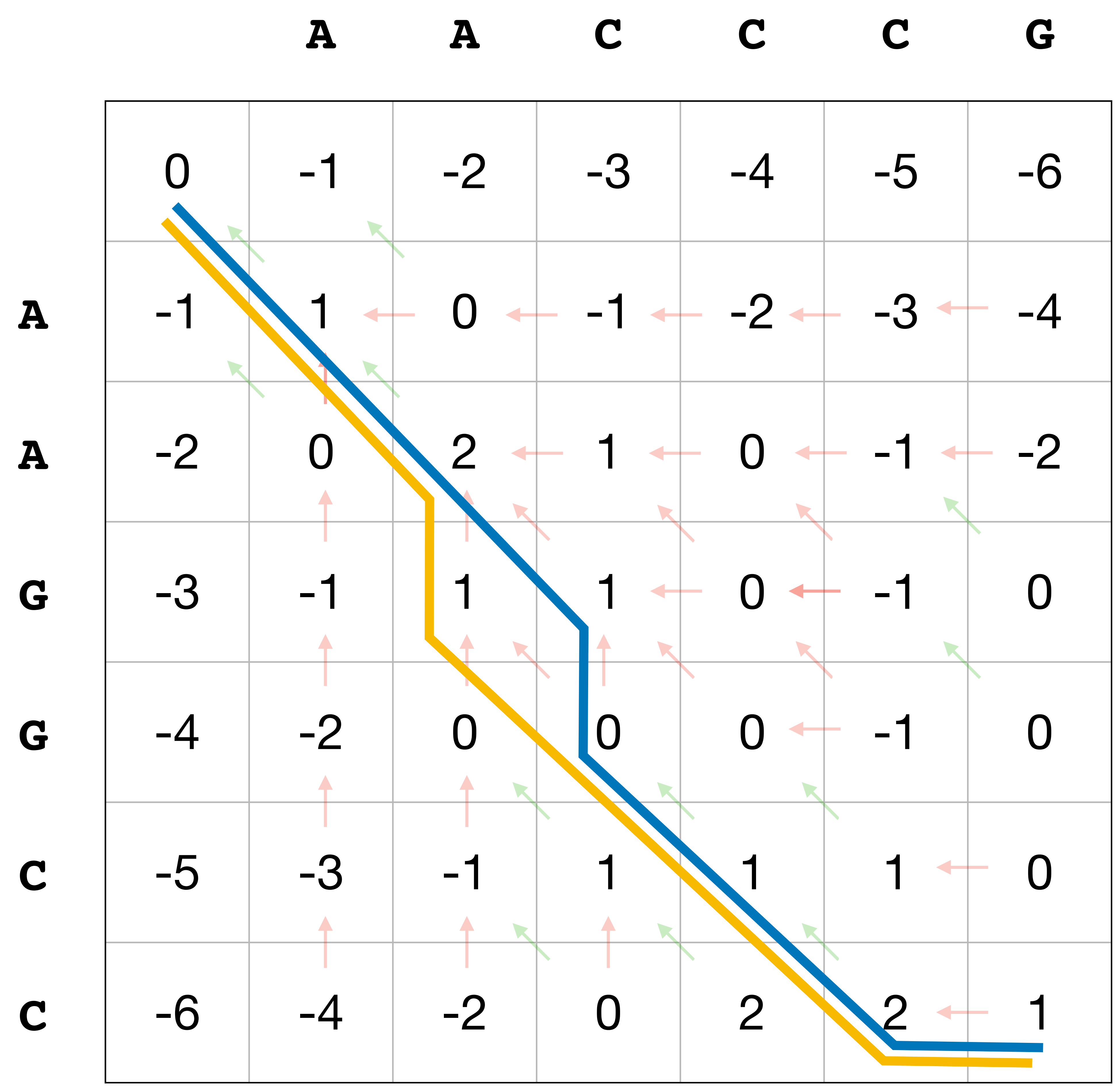
$\delta(x,-) = -1$ for $x \in \Sigma$

$\delta(x,y) = 1$ for $y = x$

$\delta(x,y) = -1$ for $y \neq x$

```
A A C - C C G
A A G G C C -
```

```
A A - C C C G
A A G G C C -
```



		A	A	C	C	C	G
	0	-1	-2	-3	-4	-5	-6
A	-1	1	0	-1	-2	-3	-4
A	-2	0	2	1	0	-1	-2
G	-3	-1	1	1	0	-1	0
G	-4	-2	0	0	0	-1	0
C	-5	-3	-1	1	1	1	0
C	-6	-4	-2	0	2	2	1

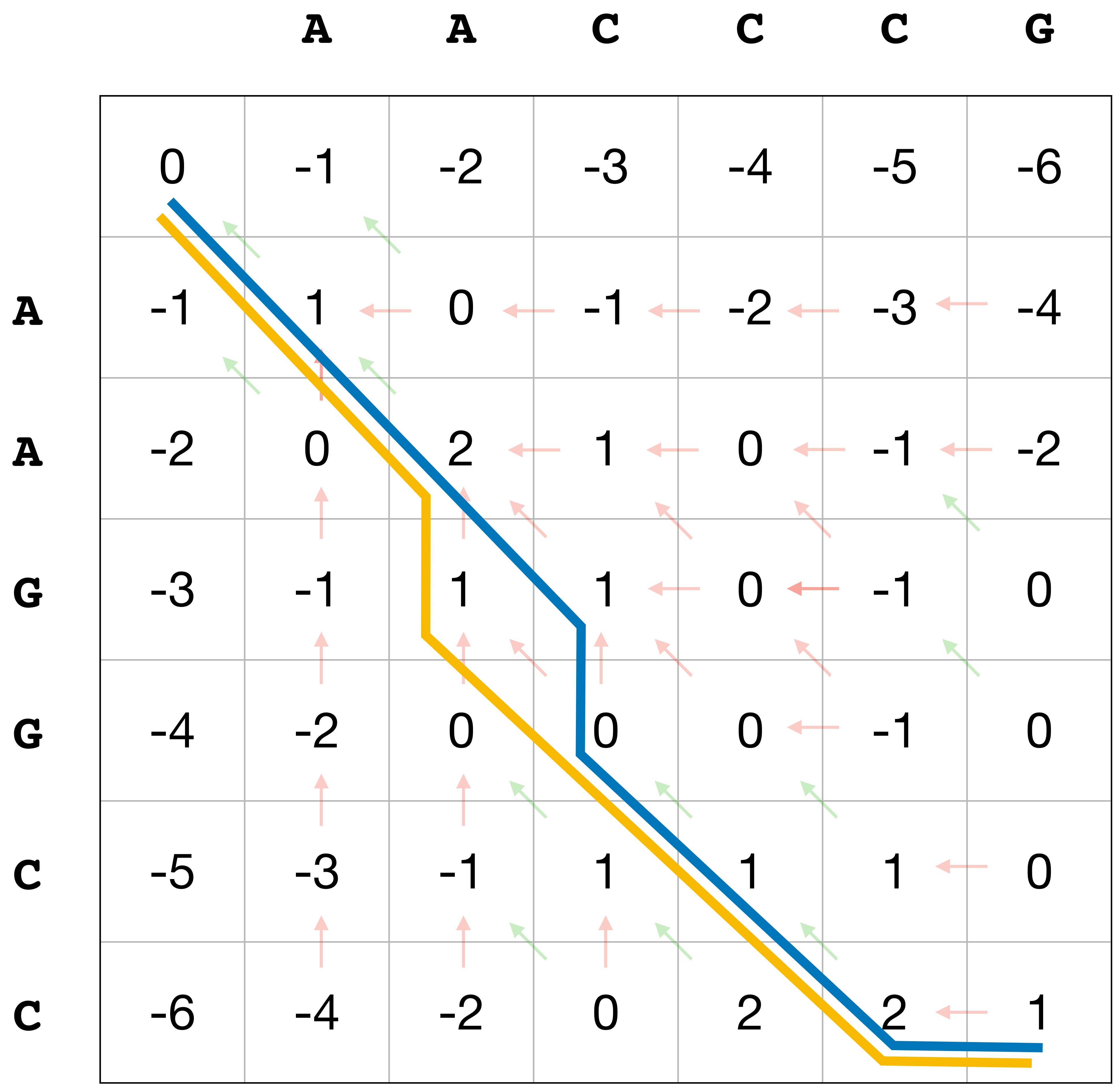
Needleman-Wunch

What about the running time and memory requirements?

- Filling in each cell of the table:
  O(1)-time, *O(1)*-space
- Table is *nxm*
- Filling in the table:
  O(mn)-time, *O(mn)*-space
- Traceback?
  - Each column of the alignment:
    O(1)-time
  - Maximum Alignment Length:
    O(m+n)
    (times the number of optimal alignments)

```
A A C – C C G
A A G G C C –
```

```
A A – C C C G
A A G G C C –
```



		A	A	C	C	C	G
	0	-1	-2	-3	-4	-5	-6
A	-1	1	0	-1	-2	-3	-4
A	-2	0	2	1	0	-1	-2
G	-3	-1	1	1	0	-1	0
G	-4	-2	0	0	0	-1	0
C	-5	-3	-1	1	1	1	0
C	-6	-4	-2	0	2	2	1

Local Alignment

- Given two strings *S* and *T*, find the two substrings, *A* of *S* and *B* of *T,* with the highest alignment score.

- Brute-force: Align all substrings of *S* with all substrings of *T.* There are $\binom{n}{2}$ substrings of *S*, and $\binom{m}{2}$ substrings of *T*. The total running time would be $O(n^3m^3)$*!*

- Smith and Waterman [1981] developed an algorithm, similar to Needleman-Wunch, that is able to find the optimal local alignment in *O(mn)*-time.

Smith-Waterman

- Still going to use an *nxm* sized matrix *V*, but:
    - each index *(i,j)* will hold the maximum global alignment score for all substrings *S[k....i]* and *T[h...j]* where *1≤k≤i* and *1≤h≤j* (the substrings could be empty).
- Then the score of the best local alignment is not necessarily in *V(i,j)*, but is $\max_{i,j} V(i,j)$

Smith-Waterman

- The recurrence relation

$$V(i,j) = \max \begin{cases} 0 & \text{align empty strings} \\ V(i-1,j-1) + \delta(S[i], T[i]) & \text{match/mismatch} \\ V(i-1,j) + \delta(S[i], -) & \text{delete} \\ V(i,j-1) + \delta(-, T[j]) & \text{insert} \end{cases}$$

- The initialization is:

  $V(0,j) = V(i,0) = 0$

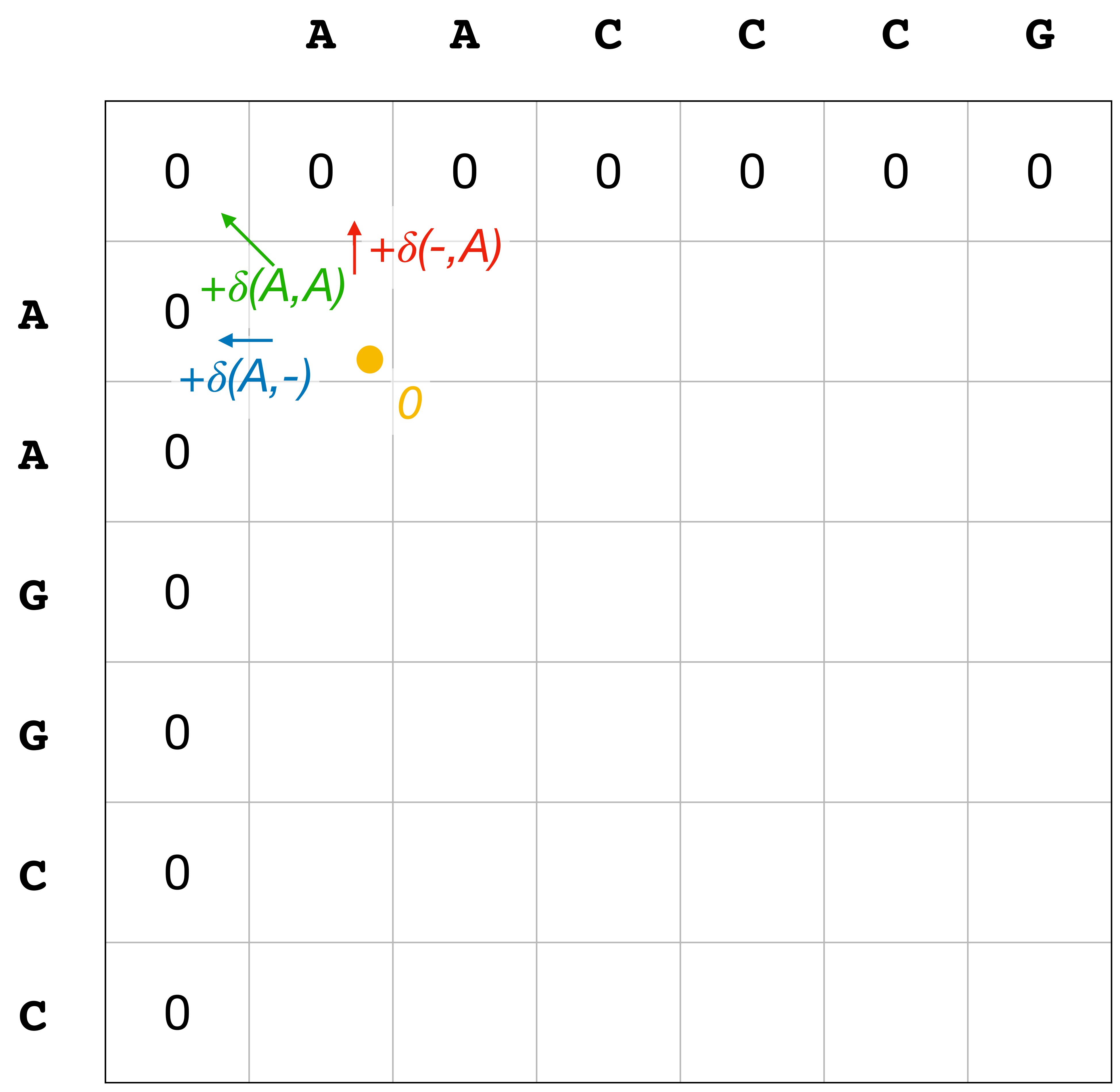
Smith-Waterman

$\delta(-,x) = -1$ for $x \in \Sigma$

$\delta(x,-) = -1$ for $x \in \Sigma$

$\delta(x,y) = 1$ for $y = x$

$\delta(x,y) = -1$ for $y \neq x$



$$\begin{array}{c|ccccccc}
 & & A & A & C & C & C & G \\
\hline
 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
A & 0 & & & & & & \\
A & 0 & & & & & & \\
G & 0 & & & & & & \\
G & 0 & & & & & & \\
C & 0 & & & & & & \\
C & 0 & & & & & &
\end{array}$$

$$V(i,j) = \max \begin{cases} 0 & \text{align empty strings} \\ V(i-1,j-1) + \delta(S[i], T[i]) & \text{match/mismatch} \\ V(i-1,j) + \delta(S[i], -) & \text{delete} \\ V(i,j-1) + \delta(-, T[j]) & \text{insert} \end{cases}$$

Smith-Waterman

δ(-,x) = -1 for *x∈Σ*

δ(x,-) = -1 for *x∈Σ*

δ(x,y) = 1 for y = x

δ(x,y) = -1 for y ≠ x

$$V(i,j) = \max \begin{cases} 0 & \text{align empty strings} \\ V(i-1,j-1) + \delta(S[i], T[i]) & \text{match/mismatch} \\ V(i-1,j) + \delta(S[i], -) & \text{delete} \\ V(i,j-1) + \delta(-, T[j]) & \text{insert} \end{cases}$$

		A	A	C	C	C	G
	0	0	0	0	0	0	0
A	0	1					
A	0						
G	0						
G	0						
C	0						
C	0						

Smith-Waterman

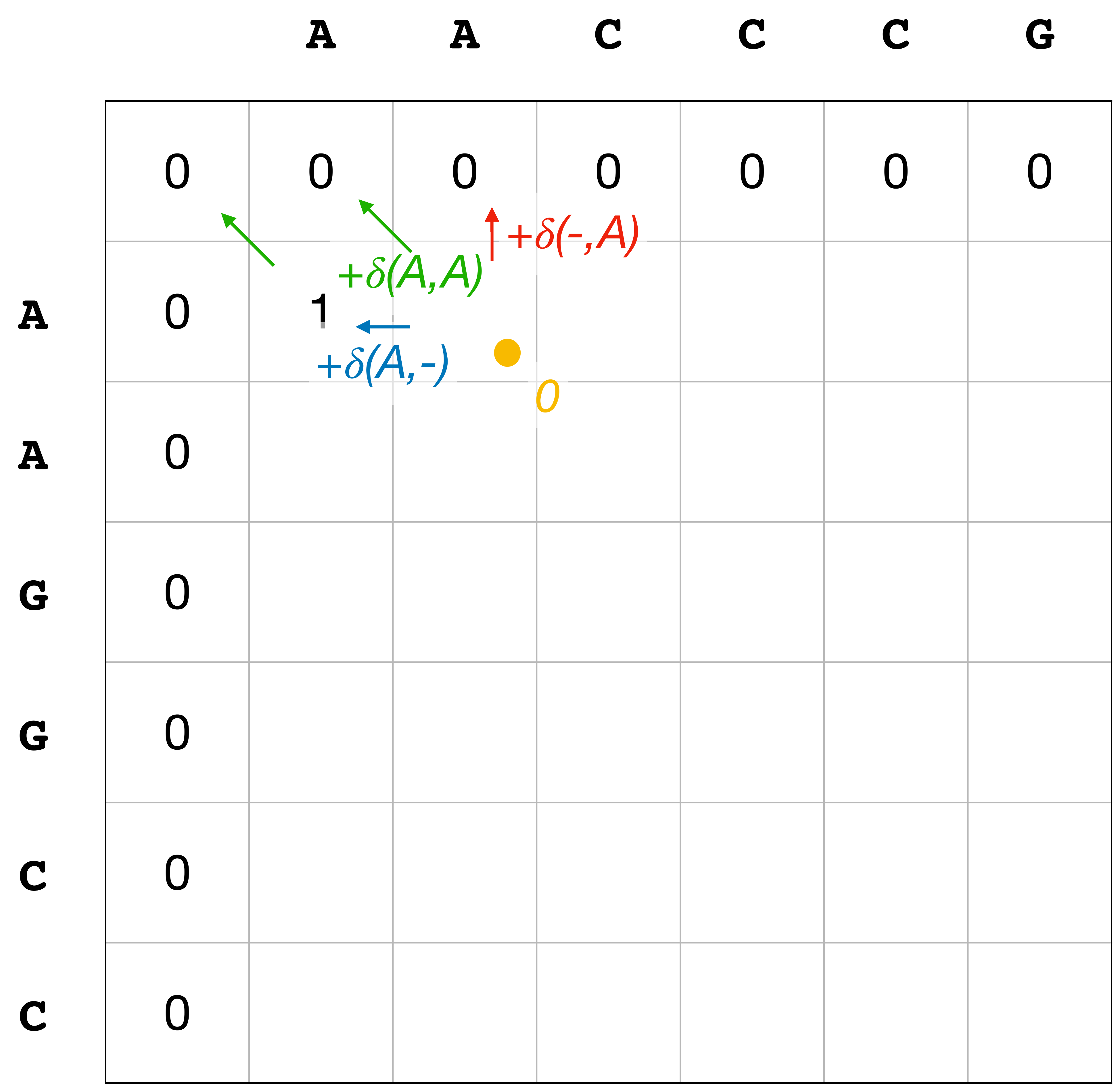
$\delta(-,x) = -1$ for $x \in \Sigma$

$\delta(x,-) = -1$ for $x \in \Sigma$

$\delta(x,y) = 1$ for $y = x$

$\delta(x,y) = -1$ for $y \neq x$

$$V(i,j) = \max \begin{cases} 0 & \text{align empty strings} \\ V(i-1,j-1) + \delta(S[i],T[i]) & \text{match/mismatch} \\ V(i-1,j) + \delta(S[i],-) & \text{delete} \\ V(i,j-1) + \delta(-,T[j]) & \text{insert} \end{cases}$$



		A	A	C	C	C	G
	0	0	0	0	0	0	0
A	0	1					
A	0						
G	0						
G	0						
C	0						
C	0						

Smith-Waterman

$\delta(-,x) = -1$ for $x \in \Sigma$

$\delta(x,-) = -1$ for $x \in \Sigma$

$\delta(x,y) = 1$ for $y = x$

$\delta(x,y) = -1$ for $y \neq x$

$$V(i,j) = \max \begin{cases} 0 & \text{align empty strings} \\ V(i-1,j-1) + \delta(S[i],T[i]) & \text{match/mismatch} \\ V(i-1,j) + \delta(S[i],-) & \text{delete} \\ V(i,j-1) + \delta(-,T[j]) & \text{insert} \end{cases}$$

		A	A	C	C	C	G
	0	0	0	0	0	0	0
A	0	↖ 1	↖ 1				
A	0						
G	0						
G	0						
C	0						
C	0						

Smith-Waterman

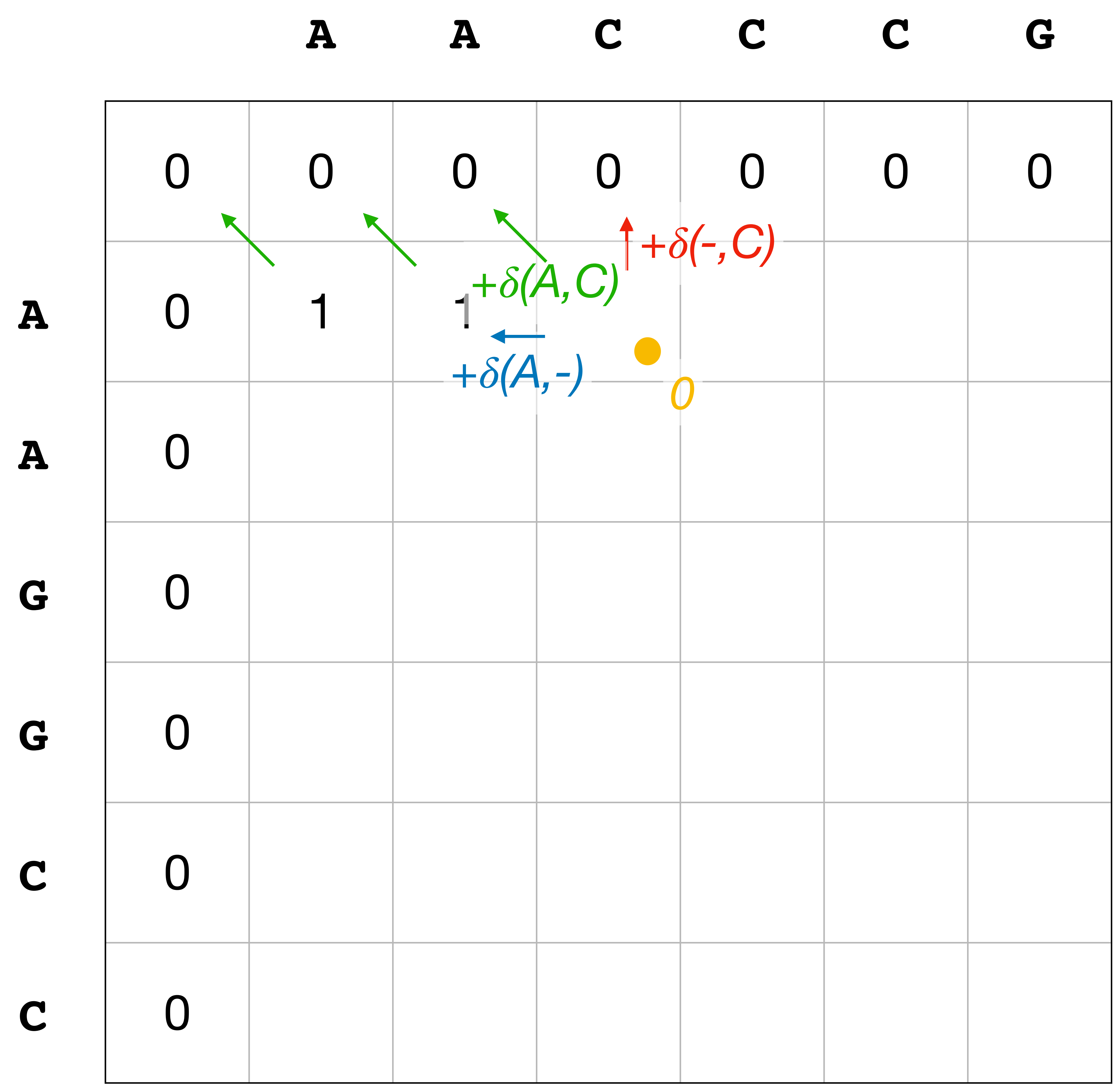
δ(-,x) = *-1* for *x*∈*Σ*

δ(x,-) = *-1* for *x*∈*Σ*

δ(x,y) = *1* for y = x

δ(x,y) = *-1* for y ≠ x

$$V(i,j) = \max \begin{cases} 0 & \text{align empty strings} \\ V(i-1,j-1) + \delta(S[i],T[i]) & \text{match/mismatch} \\ V(i-1,j) + \delta(S[i],-) & \text{delete} \\ V(i,j-1) + \delta(-,T[j]) & \text{insert} \end{cases}$$



$$\begin{array}{c|ccccccc} & & \mathtt{A} & \mathtt{A} & \mathtt{C} & \mathtt{C} & \mathtt{C} & \mathtt{G} \\ \hline & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \mathtt{A} & 0 & 1 & 1 & & & & \\ \mathtt{A} & 0 & & & & & & \\ \mathtt{G} & 0 & & & & & & \\ \mathtt{G} & 0 & & & & & & \\ \mathtt{C} & 0 & & & & & & \\ \mathtt{C} & 0 & & & & & & \end{array}$$

Smith-Waterman

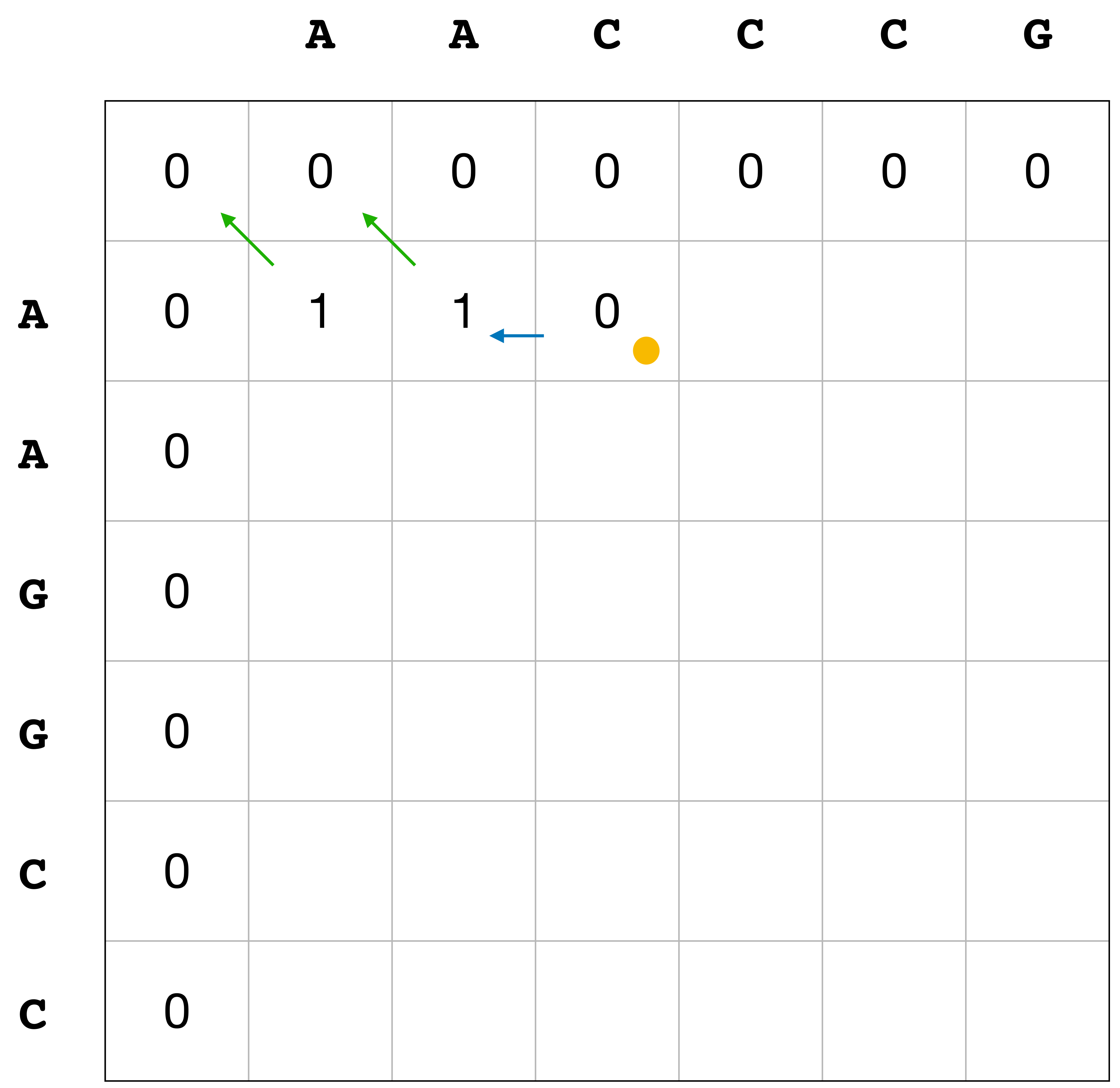
$\delta(-,x) = -1$ for $x \in \Sigma$

$\delta(x,-) = -1$ for $x \in \Sigma$

$\delta(x,y) = 1$ for $y = x$

$\delta(x,y) = -1$ for $y \neq x$

$$V(i,j) = \max \begin{cases} 0 & \text{align empty strings} \\ V(i-1,j-1) + \delta(S[i],T[i]) & \text{match/mismatch} \\ V(i-1,j) + \delta(S[i],-) & \text{delete} \\ V(i,j-1) + \delta(-,T[j]) & \text{insert} \end{cases}$$



		A	A	C	C	C	G
	0	0	0	0	0	0	0
A	0	1	1	0 ●			
A	0						
G	0						
G	0						
C	0						
C	0						

Smith-Waterman

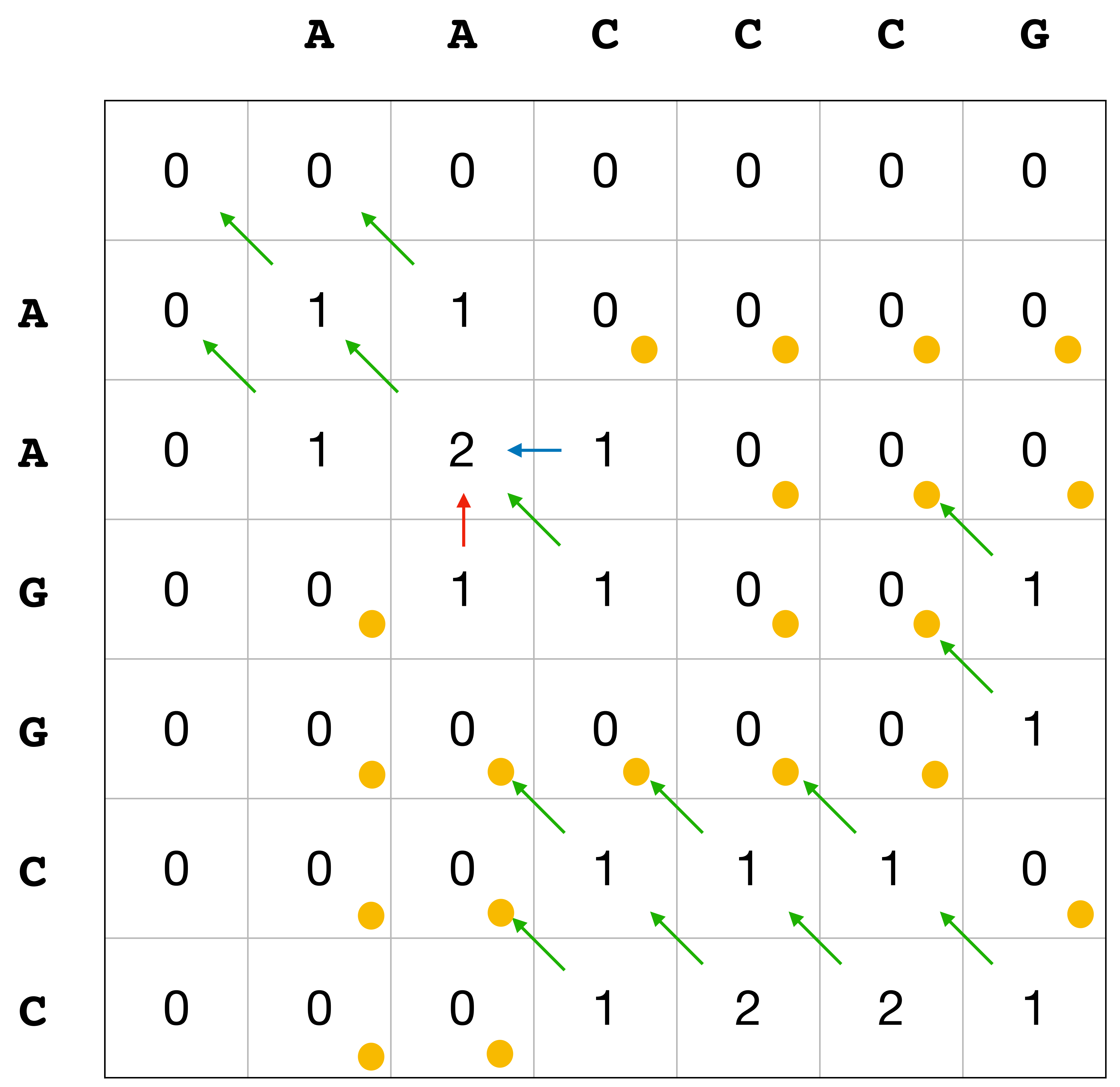
$\delta(-,x) = -1$ for $x \in \Sigma$

$\delta(x,-) = -1$ for $x \in \Sigma$

$\delta(x,y) = 1$ for $y = x$

$\delta(x,y) = -1$ for $y \neq x$

$$V(i,j) = \max \begin{cases} 0 & \text{align empty strings} \\ V(i-1,j-1) + \delta(S[i],T[i]) & \text{match/mismatch} \\ V(i-1,j) + \delta(S[i],-) & \text{delete} \\ V(i,j-1) + \delta(-,T[j]) & \text{insert} \end{cases}$$



		A	A	C	C	C	G
	0	0	0	0	0	0	0
A	0	1	1	0	0	0	0
A	0	1	2	1	0	0	0
G	0	0	1	1	0	0	1
G	0	0	0	0	0	0	1
C	0	0	0	1	1	1	0
C	0	0	0	1	2	2	1

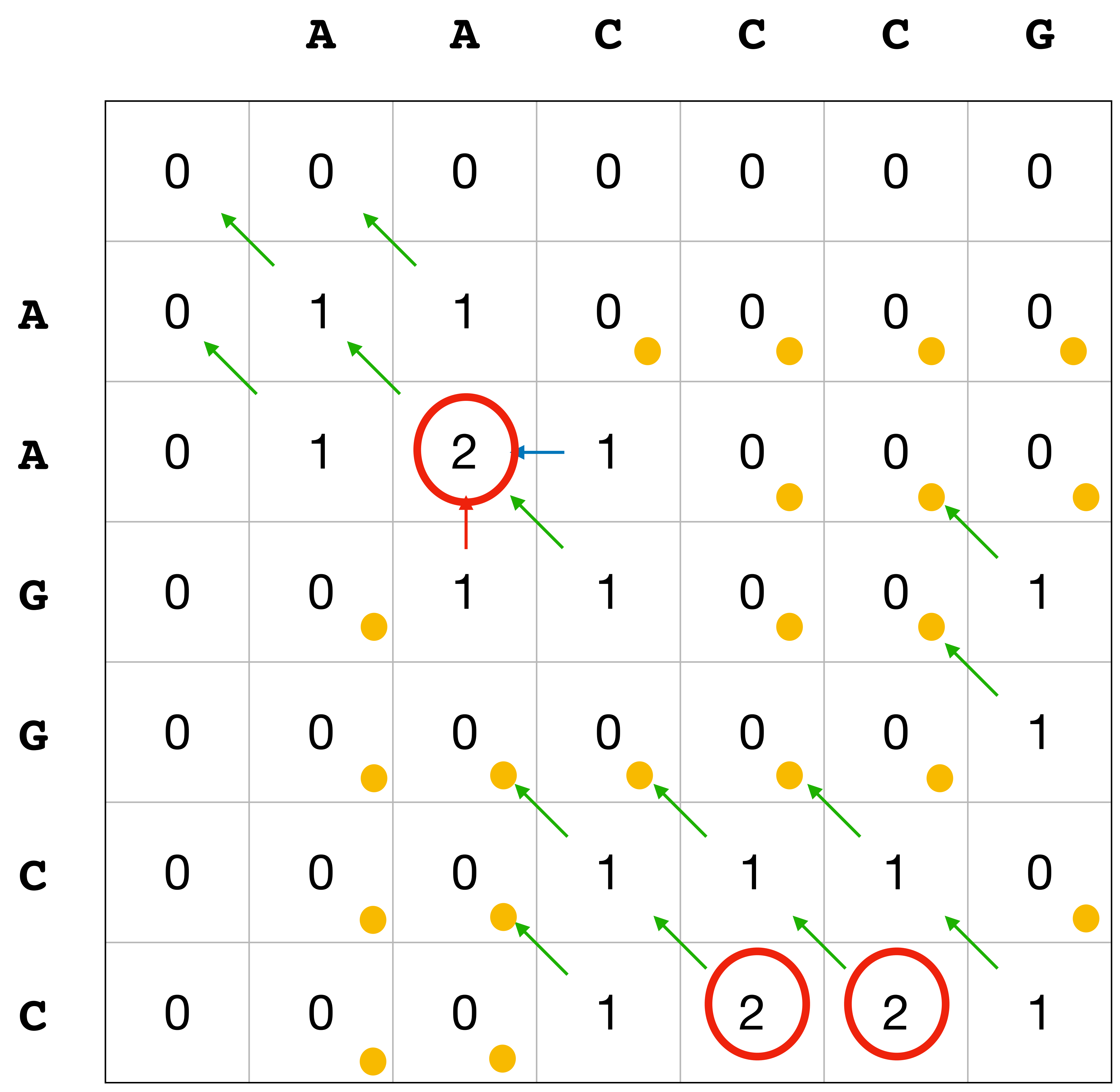
Smith-Waterman

$\delta(-,x) = -1$ for $x \in \Sigma$

$\delta(x,-) = -1$ for $x \in \Sigma$

$\delta(x,y) = 1$ for $y = x$

$\delta(x,y) = -1$ for $y \neq x$



		A	A	C	C	C	G
	0	0	0	0	0	0	0
A	0	1	1	0	0	0	0
A	0	1	2	1	0	0	0
G	0	0	1	1	0	0	1
G	0	0	0	0	0	0	1
C	0	0	0	1	1	1	0
C	0	0	0	1	2	2	1

Smith-Waterman

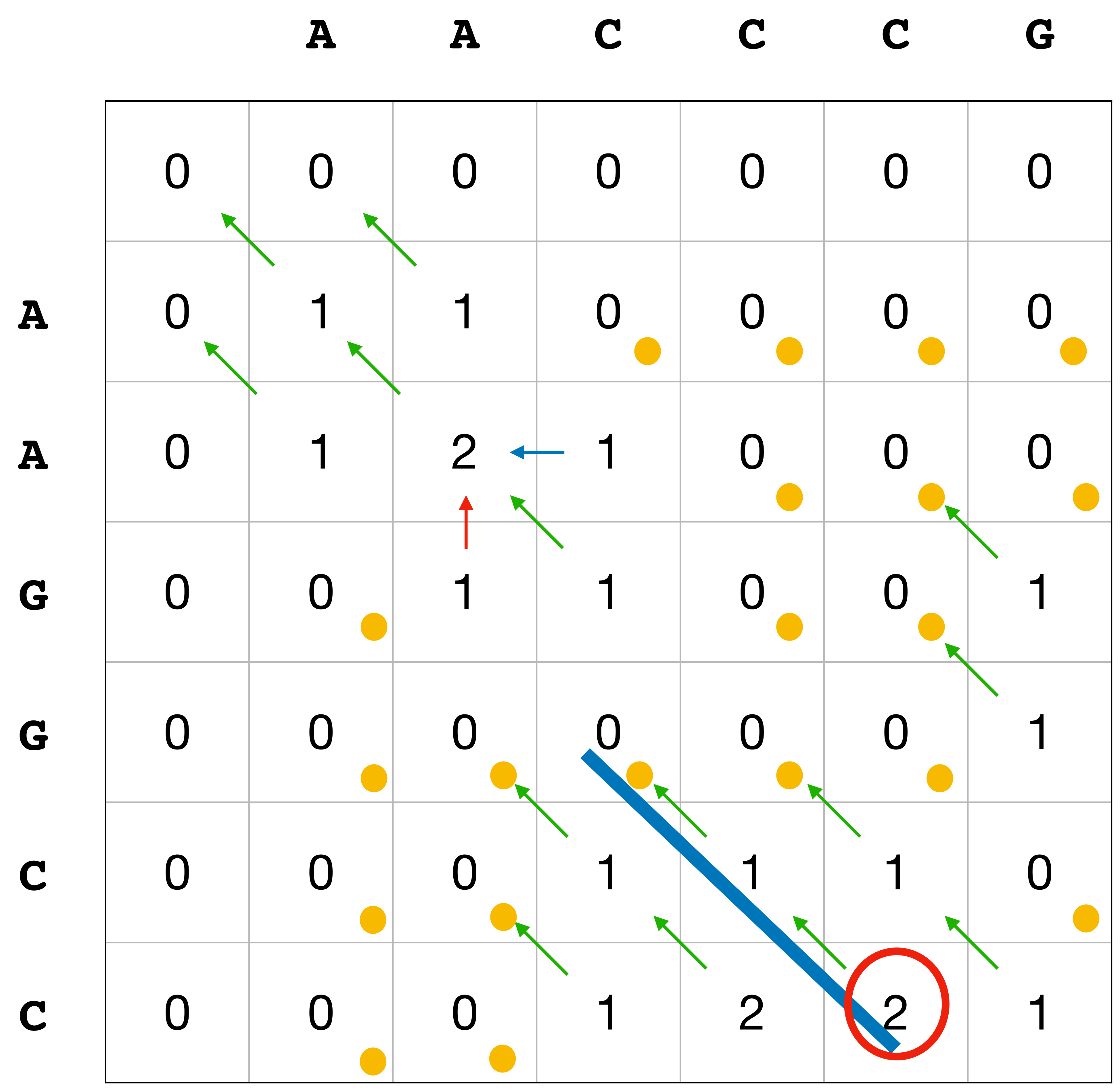
$\delta(-,x) = -1$ for $x \in \Sigma$

$\delta(x,-) = -1$ for $x \in \Sigma$

$\delta(x,y) = 1$ for $y = x$

$\delta(x,y) = -1$ for $y \neq x$

```
C C
C C
```



		A	A	C	C	C	G
	0	0	0	0	0	0	0
A	0	1	1	0	0	0	0
A	0	1	2	1	0	0	0
G	0	0	1	1	0	0	1
G	0	0	0	0	0	0	1
C	0	0	0	1	1	1	0
C	0	0	0	1	2	2	1

Semi-global Alignment

- In some cases one of the sequence may be smaller than the other
    - in biology, this may be due to sequencing ending early or some process in the cell
- In this case, we may want to not penalize gaps at the beginning or end of one of the sequences
- How can this be done using a slight adjustment to the Myers-Miller algorithm?

Lets talk about gaps!

- Up to now inserting a gap character (-) was always a constant cost operation
- In real life (i.e. in biology) starting an inserting 10 bases and inserting 12 bases causes close the same amount of disruption, whereas inserting 0 bases and inserting 2 is a big difference
- So we want to score alignments with this in mind
- Generally, for a gap (set of contiguous insertions or deletions) of length *k,* we want the score to be some function *f(k)* of the length

General Gap Costs

- Given that a gap of length *k* receives a score *f(k)*, in global alignment we change the recursion to the following:

$$V(i,j) = \max \begin{cases} V(i-1,j-1) + \delta(S[i], T[i]) & \text{match/mismatch} \\ \max_{0 \le h \le j-1} \left\{ V(i,h) - f(j-h) \right\} & \text{insert T[h+1...j]} \\ \max_{0 \le h \le j-1} \left\{ V(h,j) - f(i-h) \right\} & \text{delete S[h+1...i]} \end{cases}$$

- And initialization changes slightly:
  V(0,0) = 0
  V(0,j) = -f(j)
  V(i,0) = -f(i)

- This change also increases the running time. Each entry now takes *O(m+n)*-time, therefore the total running time is *O(mn(m+n))*

Affine Gap Costs

- The one everyone uses!
- Attributed to Gotoh [1982]
- Define the function $f_{a,b}(k) =: a + b * k$ where *a* and *b* are tunable parameters (if $a=0$, this is the same as before)
- Can still be solved in *O(mn)*-time and *O(mn)*-space, but we need a bit more sophistication

Gotoh's Algorithm

- Define 3 *nxm* matricies:
    - *G* -- the alignment score for alignments that end in a match (the last column)
    - *F* -- the alignment score for alignments that end in an insertion
    - *E* -- the alignment score for alignments that end in a deletion

Gotoh's Algorithm

Recursion

$$F(i,j) = \max \begin{cases} F(i-1,j) - b \\ G(i-1,j) - f_{a,b}(1) \end{cases}$$

$$E(i,j) = \max \begin{cases} E(i,j-1) - b \\ G(i,j-1) - f_{a,b}(1) \end{cases}$$

$$G(i,j) = \max \begin{cases} G(i-1,j-1) + \delta(S[i],T[j]) \\ E(i,j) \\ F(i,j) \end{cases}$$

Initialization

$G(0,j) = E(0,j) = -1f_{a,b}(j)$

$G(i,0) = F(i,0) = -1f_{a,b}(i)$

$E(i,0) = -\infty$

$F(0,j) = -\infty$

Gotoh's Algorithm

$G(0,j) = E(0,j) = -1f_{a,b}(j)$

$G(i,0) = F(i,0) = -1f_{a,b}(i)$

$E(i,0) = -\infty$

$F(0,j) = -\infty$

-b -a

E

		A	A	C	C	C	G
	-∞	-∞ -b	-∞	-∞	-∞	-∞	-∞
A	-4	-8					
A	-5						
G	-6						
G	-7						
C	-8						
C	-9						

G

		A	A	C	C	C	G
	0	-4	-5	-6	-7	-8	-9
A	-4						
A	-5						
G	-6						
G	-7						
C	-8						
C	-9						

F

		A	A	C	C	C	G
	-∞	-4	-5	-6	-7	-8	-9
A	-∞						
A	-∞						
G	-∞						
G	-∞						
C	-∞						
C	-∞						

$\delta(x,y) = 10$ for y = x

$\delta(x,y) = -2$ for y ≠ x

$$E(i,j) = \max \begin{cases} E(i,j-1) - b \\ G(i,j-1) - f_{a,b}(1) \end{cases}$$

$$G(i,j) = \max \begin{cases} G(i-1,j-1) + \delta(S[i],T[j]) \\ E(i,j) \\ F(i,j) \end{cases}$$

$$F(i,j) = \max \begin{cases} F(i-1,j) - b \\ G(i-1,j) - f_{a,b}(1) \end{cases}$$

a = 3

b = 1

Gotoh's Algorithm

$G(0,j) = E(0,j) = -1f_{a,b}(j)$

$G(i,0) = F(i,0) = -1f_{a,b}(i)$

$E(i,0) = -\infty$

$F(0,j) = -\infty$

E

		A	A	C	C	C	G
	-∞	-∞	-∞	-∞	-∞	-∞	-∞
A	-4	-8					
A	-5						
G	-6						
G	-7						
C	-8						
C	-9						

G

		A	A	C	C	C	G
	0	-4	-5	-6	-7	-8	-9
A	-4						
A	-5						
G	-6						
G	-7						
C	-8						
C	-9						

-b -a

F

		A	A	C	C	C	G
	-∞	-4	-5	-6	-7	-8	-9
A	-∞ ← -b	-8					
A	-∞						
G	-∞						
G	-∞						
C	-∞						
C	-∞						

δ(x,y) = 10 for y = x

δ(x,y) = -2 for y ≠ x

$$E(i,j) = \max \begin{cases} E(i,j-1) - b \\ G(i,j-1) - f_{a,b}(1) \end{cases}$$

$$G(i,j) = \max \begin{cases} G(i-1,j-1) + \delta(S[i],T[j]) \\ E(i,j) \\ F(i,j) \end{cases}$$

$$F(i,j) = \max \begin{cases} F(i-1,j) - b \\ G(i-1,j) - f_{a,b}(1) \end{cases}$$

a = 3
b = 1

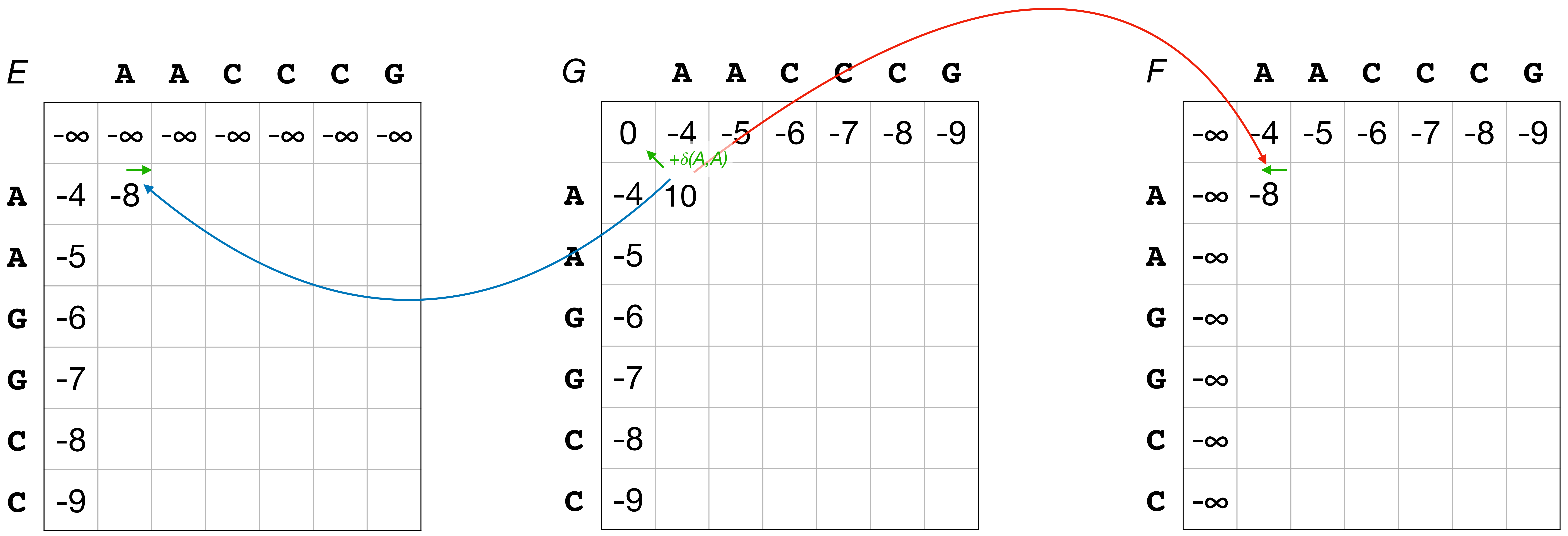
Gotoh's Algorithm

$G(0,j) = E(0,j) = -1f_{a,b}(j)$

$G(i,0) = F(i,0) = -1f_{a,b}(i)$

$E(i,0) = -\infty$

$F(0,j) = -\infty$



E

		A	A	C	C	C	G
	-∞	-∞	-∞	-∞	-∞	-∞	-∞
A	-4	-8					
A	-5						
G	-6						
G	-7						
C	-8						
C	-9						

G

		A	A	C	C	C	G
	0	-4	-5	-6	-7	-8	-9
A	-4	10					
A	-5						
G	-6						
G	-7						
C	-8						
C	-9						

F

		A	A	C	C	C	G
	-∞	-4	-5	-6	-7	-8	-9
A	-∞	-8					
A	-∞						
G	-∞						
G	-∞						
C	-∞						
C	-∞						

δ(x,y) = 10 for y = x

δ(x,y) = -2 for y ≠ x

$$E(i,j) = \max \begin{cases} E(i,j-1) - b \\ G(i,j-1) - f_{a,b}(1) \end{cases}$$

$$G(i,j) = \max \begin{cases} G(i-1,j-1) + \delta(S[i],T[j]) \\ E(i,j) \\ F(i,j) \end{cases}$$

$$F(i,j) = \max \begin{cases} F(i-1,j) - b \\ G(i-1,j) - f_{a,b}(1) \end{cases}$$

a = 3
b = 1

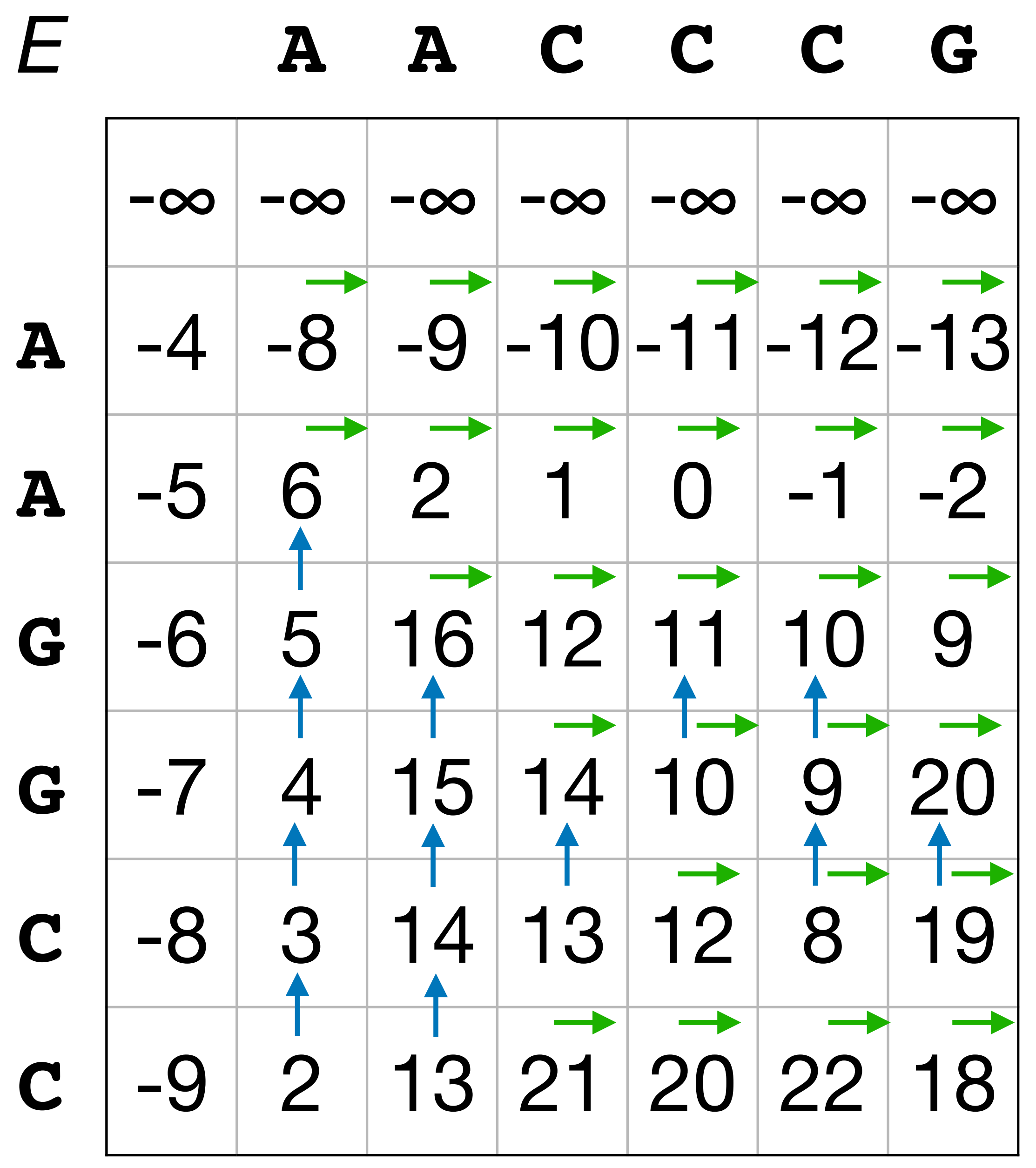
Gotoh's Algorithm

$G(0,j) = E(0,j) = -1 f_{a,b}(j)$

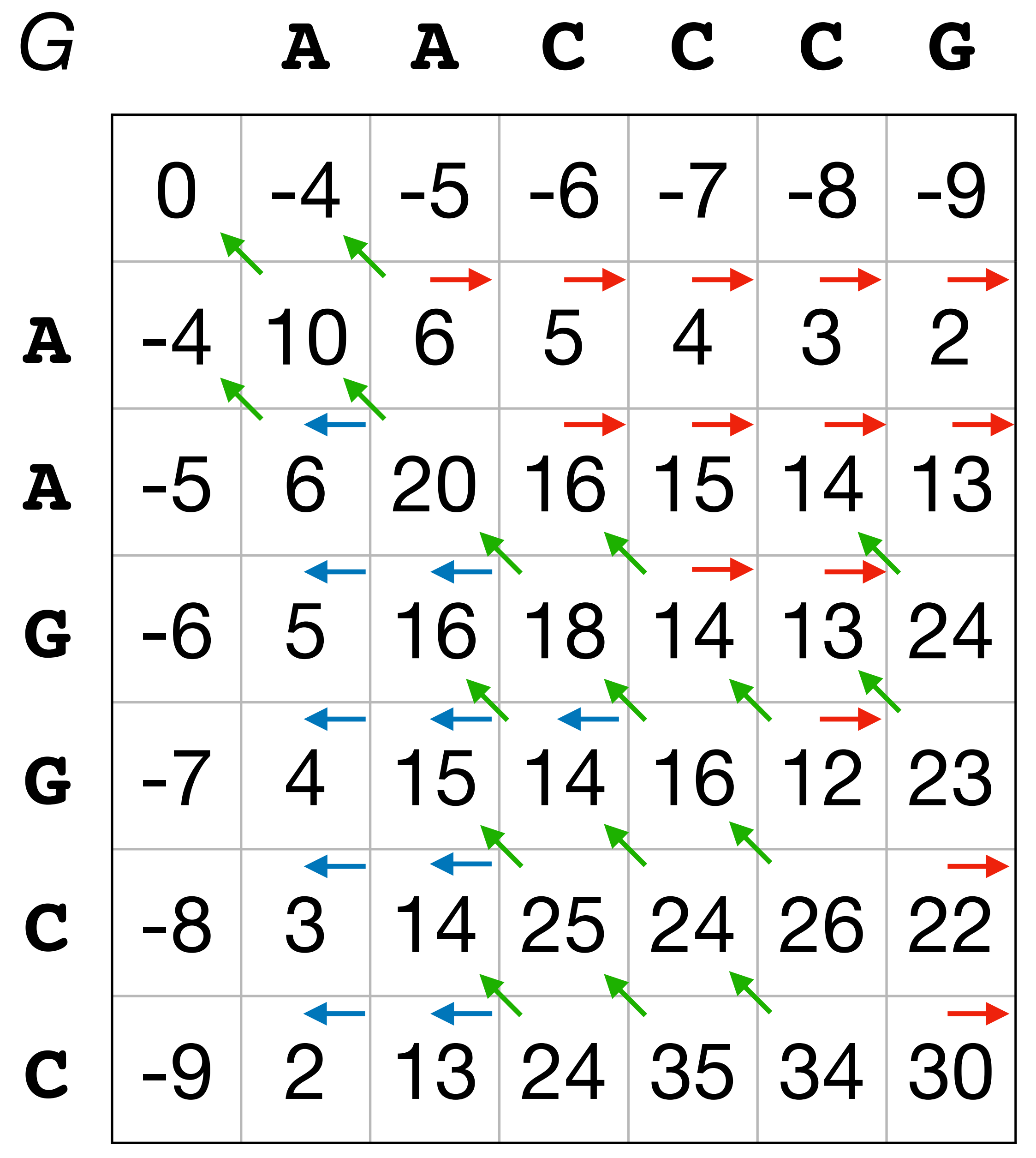
$G(i,0) = F(i,0) = -1 f_{a,b}(i)$

$E(i,0) = -\infty$

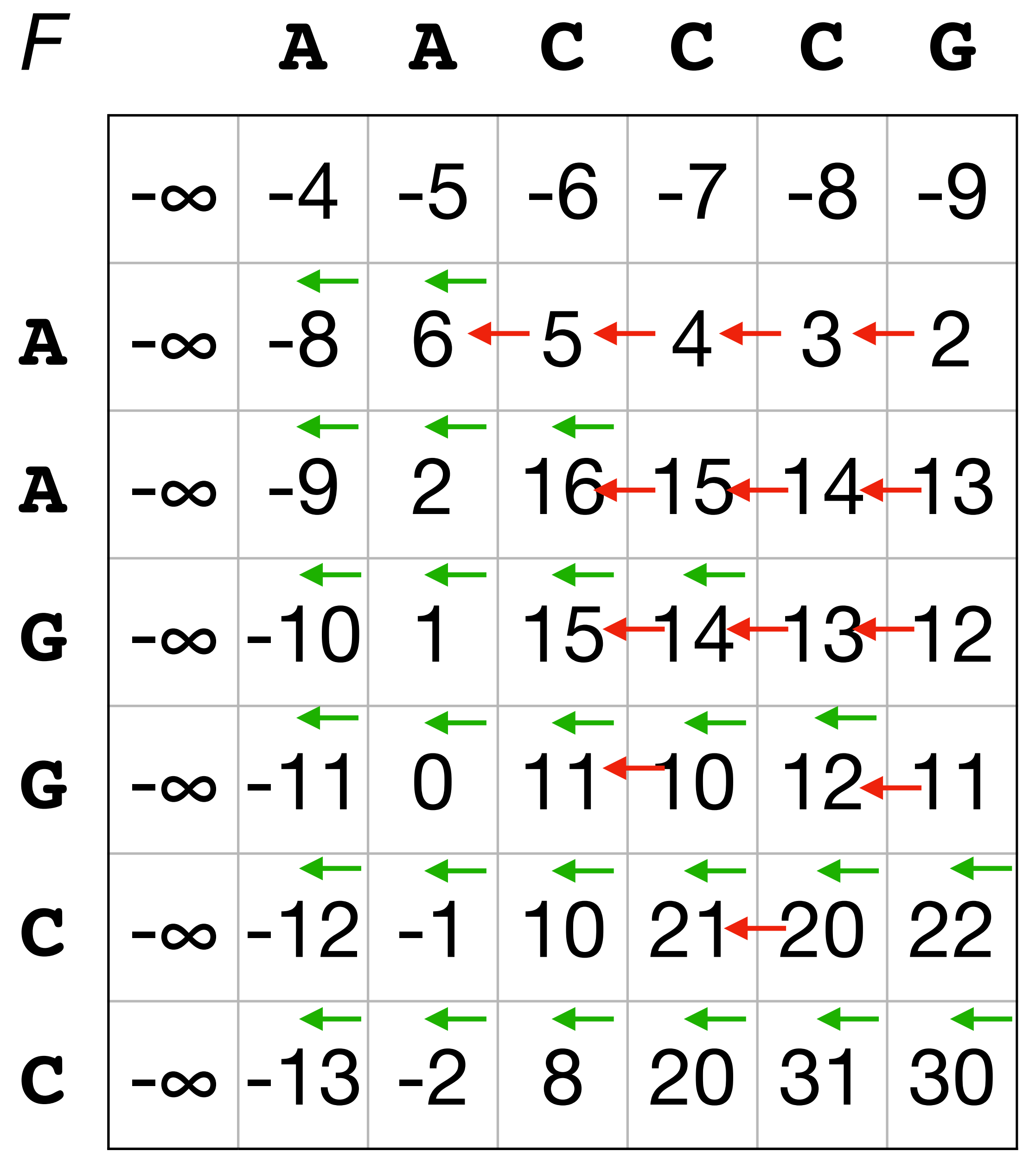
$F(0,j) = -\infty$



E		A	A	C	C	C	G
	-∞	-∞	-∞	-∞	-∞	-∞	-∞
A	-4	-8	-9	-10	-11	-12	-13
A	-5	6	2	1	0	-1	-2
G	-6	5	16	12	11	10	9
G	-7	4	15	14	10	9	20
C	-8	3	14	13	12	8	19
C	-9	2	13	21	20	22	18



G		A	A	C	C	C	G
	0	-4	-5	-6	-7	-8	-9
A	-4	10	6	5	4	3	2
A	-5	6	20	16	15	14	13
G	-6	5	16	18	14	13	24
G	-7	4	15	14	16	12	23
C	-8	3	14	25	24	26	22
C	-9	2	13	24	35	34	30



F		A	A	C	C	C	G
	-∞	-4	-5	-6	-7	-8	-9
A	-∞	-8	6	5	4	3	2
A	-∞	-9	2	16	15	14	13
G	-∞	-10	1	15	14	13	12
G	-∞	-11	0	11	10	12	11
C	-∞	-12	-1	10	21	20	22
C	-∞	-13	-2	8	20	31	30

δ(x,y) = 10 for y = x

δ(x,y) = -2 for y ≠ x

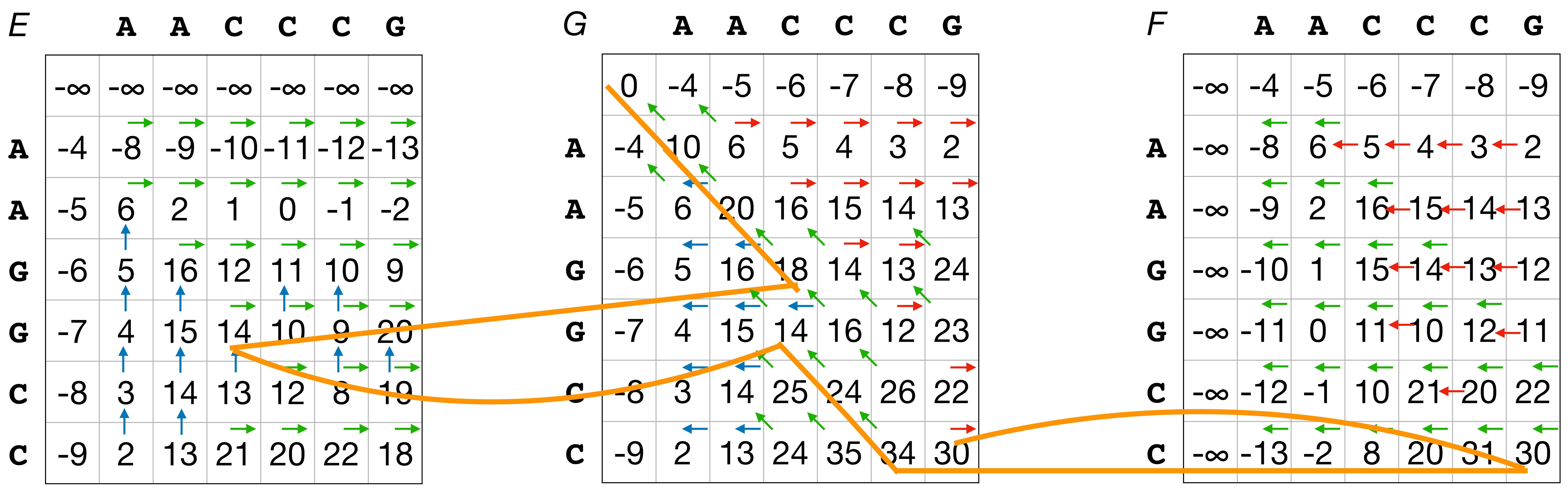
$$E(i,j) = \max \begin{cases} E(i,j-1) - b \\ G(i,j-1) - f_{a,b}(1) \end{cases}$$

$$G(i,j) = \max \begin{cases} G(i-1,j-1) + \delta(S[i],T[j]) \\ E(i,j) \\ F(i,j) \end{cases}$$

$$F(i,j) = \max \begin{cases} F(i-1,j) - b \\ G(i-1,j) - f_{a,b}(1) \end{cases}$$

a = 3
b = 1

Gotoh's Algorithm



E

		A	A	C	C	C	G
	-∞	-∞	-∞	-∞	-∞	-∞	-∞
A	-4	-8	-9	-10	-11	-12	-13
A	-5	6	2	1	0	-1	-2
G	-6	5	16	12	11	10	9
G	-7	4	15	14	10	9	20
C	-8	3	14	13	12	8	19
C	-9	2	13	21	20	22	18

G

		A	A	C	C	C	G
	0	-4	-5	-6	-7	-8	-9
A	-4	10	6	5	4	3	2
A	-5	6	20	16	15	14	13
G	-6	5	16	18	14	13	24
G	-7	4	15	14	16	12	23
C	-8	3	14	25	24	26	22
C	-9	2	13	24	35	34	30

F

		A	A	C	C	C	G
	-∞	-4	-5	-6	-7	-8	-9
A	-∞	-8	6	5	4	3	2
A	-∞	-9	2	16	15	14	13
G	-∞	-10	1	15	14	13	12
G	-∞	-11	0	11	10	12	11
C	-∞	-12	-1	10	21	20	22
C	-∞	-13	-2	8	20	31	30

```
A A C – C C G
A A G G C C –
```

([illegible])

Computing alignments in linear space

- Up to now, global, local, and semi-global alignments using fixed, general, and affine gap costs have all taken *O(mn)*-time and *O(mn)*-space

- For long sequences we may not be able to keep the whole table in main memory, therefore if the space can be reduced it would improve the practicality of the algorithms

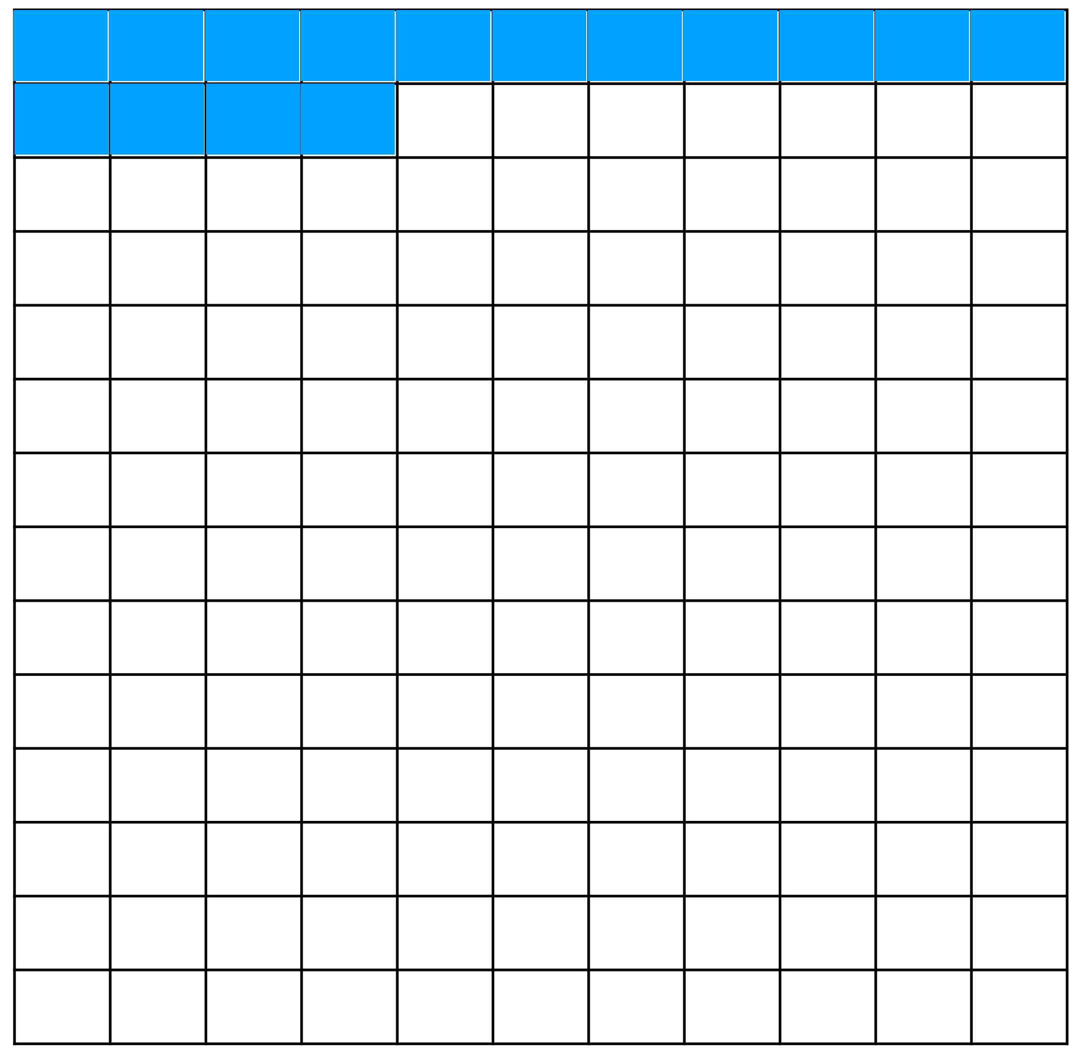
- Hershberg [1975] shows that global alignment can be computed in *O(n+m)*-space[1]

- The key is that when computing *V*, (in row-major, left to right order) the value only depends on the row above, and the current values in the same row to the left

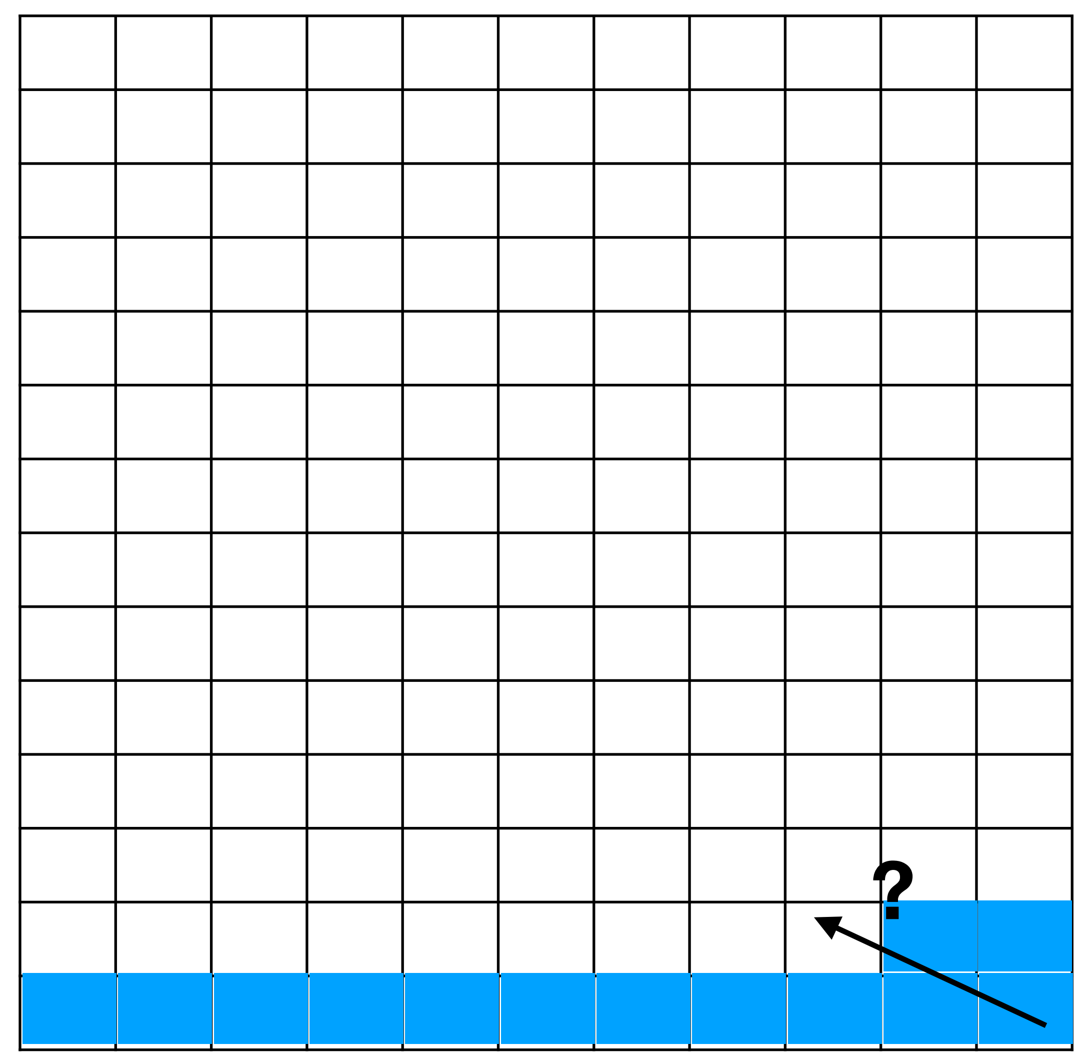
- But with only that observation you can find the best alignment **score** in linear space, but not the alignment itself

[1]Myers and Miller [1988] later show that using the same ideas, affine gap alignments can be computed in O*(m)*-space (*m<n*)

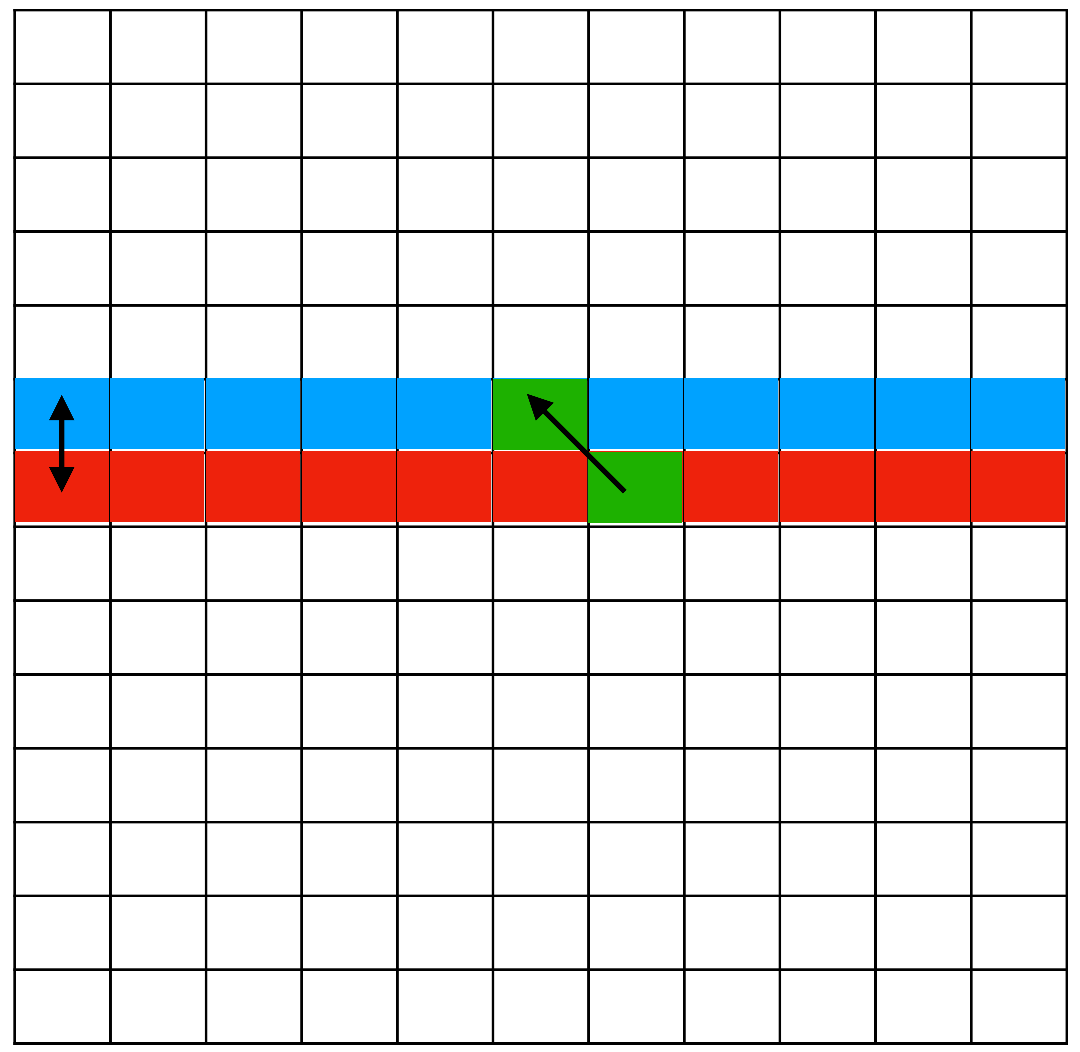
Hirschberg's Algorithm



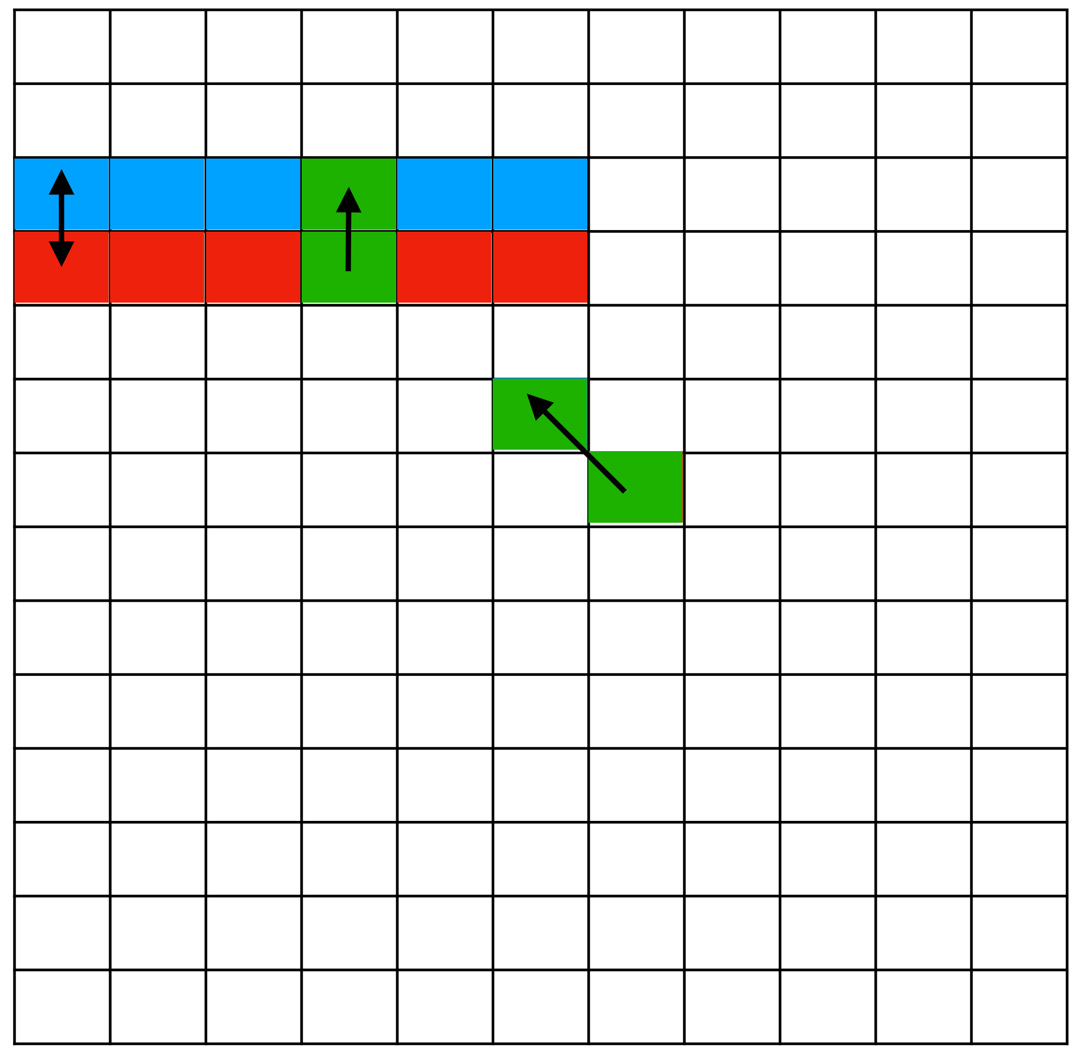
Hirschberg's Algorithm



Hirschberg's Algorithm



Hirschberg's Algorithm



Hirschberg's Algorithm

```
FindMid(S[1...n],T[1...n]):
  F = CostOnlyNWForward(S[1...n/2],T[1...m])
  B = CostOnlyNWBackward(S[n/2+1...n],T[1...m])
  return argmax_j $\{F[j] + B[j]\}$
```

O(m)-space, freed after each call

```
HirschbergAlign(S[i...j],T[x...y]):
   if i=j then  compute full Needleman-Wunsch and return the alignment
   mid = (i+j)/2
   z = FindMid(S[1...n],T[1...n])
   return HirschbergAlign(S[i...mid],T[x...z]) · HirschbergAlign(S[mid+1...j],T[z+1...y])
```

O(n)-space, to store the stack

Concatenation

Hirschberg's Algorithm

```
FindMid(S[1...n],T[1...n]):
  F = CostOnlyNWForward(S[1...n/2],T[1...m])
  B = CostOnlyNWBackward(S[n/2+1...n],T[1...m])
  return argmax_j $\{F[j] + B[j]\}$

HirschbergAlign(S[i...j],T[x...y]):
    if i=j then  compute full Needleman-Wunsch and return the alignment
    mid = (i+j)/2
    z = FindMid(S[1...n],T[1...n])
    return HirschbergAlign(S[i...mid],T[x...z]) · HirschbergAlign(S[mid+1...j],T[z+1...y])
```

O(mn)-time

O(mn)-time

O(1)-time

O(mn)-time for each call

Time(n,m)

O(mn)-time

O(mn)-time total

Time(n/2,z) + *Time(n/2,m-z)*

Concatenation

Alignment Scores

- In all of the examples we have been given *δ, a, b,* etc. and using fixed values we are able to find the optimal alignment in *O(mn)*-time
- Since different values of those parameter induce different alignments, how do we know which parameter values are best?
- This is a problem known as *inverse parametric alignment* (or just parametric alignment) where you want to find ***all*** possible alignments of the two sequences.
- How many *optimal* alignments do you think there are? (notice that optimal is emphasized)