Introduction to Alignment, Mapping and Indexing

UNIVERSITY OF MARYLAND

NOTE: This lecture is being recorded

The problem we want to solve

```
GGTACCCCCAAATTCGAAAGAGCCGAATGAAGTGGAAAGAATGGCCATTCAAATGGCTTGGGA
```

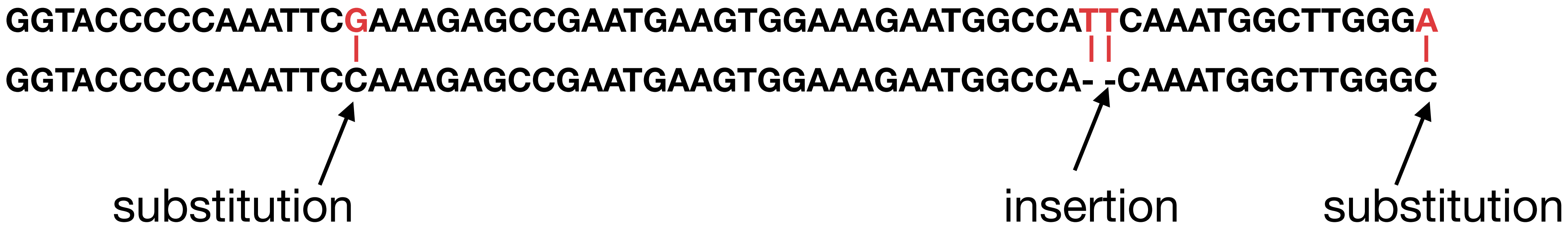
```
CAGGGCGCGCCGCTGGCCGAGAAAACCGCATTACAAATACTCTCAAAGCAGGATTTCTTTCTATAGGTAGTCAGTCCACAATTTTCACATTTGCAGCCA
GGTGGACATCTTTGGGCCAATGATGAGGAGAAGGTGGAGGAAGAAGGAAATGAGGAAGAGGAGAAAGAAGAGTATGGCATTTTACAAACTGTGACCGTT
TCTGTGTGAAGATTTTTAGCTGTCTGTCGGACTTGGGGGTCTCAGGGAAACTCACTTTGCCCCCCAGCTGAGGTTTTCAGGAAATCTGGAAACCTACAG
TCTCCAAGCCTGCTCAGCCAAGAACGCGGCGGGCGGCGGCGGTGGCGACGGCGGCGGCGGCGCAGGACCTCCGCGCCTCCATTATGCTATTCTGCCCGC
CGTGGGTGACAAACAGACGGATGCTACCAGCCCAGCCCAGTCCCGGGGAGCCAGCTGGCCTGGGGTTCGGTCCCGCGTCTTCCCTCATTCTGTGCCGCT
GCCGAGCCTGTCTCAGCTCCACACACGCTTGGGAGCTGCAGATGCCTCCGCCCCTCCTCTCTCCCAGGCTCTTCCTGCCGTTGAACCCCGGCGGCGGGC
TCTCGGCCAGCGGCGCGCCCTGGTACCCCCAAATTCCAAAGAGCCGAATGAAGTGGAAAGAATGGCCACAAATGGCTTGGGCCCGAGTGACCATGGGAT
GGTTAGGTAGGATTTTAGAGGCGACTGCTCCTGGAATTAGAGAAAGAGTTTCATTACAACCGCTACCTGACCCGAAGGCACTCGGGCCCATGCCTTCCT
CTCCTTCGCTGTTTGATTTCTATTCTGTTTAGCCTGAGTTGGAGATGCCGGAGGTGCCGTGGTGGAAGAAGTCTTGAACGAATTTGGAGGCGTCTCCGT
GGCAGCTAAGCGAGCACGGGTTCTGCTGGTGCAGGATGACACTGGCAGCCACTGCCGCGGACTTGTATCTCTTGTCTTCCTGCTTTTATAGAGAATAGA
ATGACACTCACAACTCTAACTACCTGTCAGAAGCAGGCAGGAGCTAGTAAGGATGAATTTGTAGCAAAATTAGCAAGTGGACTTCTTTCTCCTCTTCCT
CATTTCCTTCTTCCTCCACCTTCTCCTCATCTTAAATCTTTAACATACTACCTAAAGGGAACCTGCAATAATCTTGAAAAAGGACTTCAATCCGACGTT
TTCGTGTCAAATAAGGATTAAAGAGAAACTCCTCCGCGAGCCGTGCGCCGAGGGTGGCGGCGGGGGCCTGAAGCGTGAGGAGCCTTCAATATGTATTTA
ACCAGGGACCGTCGGTATGAGGTGGCCCGGGTTCTTATTTGTTTGGGGGCTGGAGGGGGGAGACGGAGAAACAGTGAAAAGTTCCTGAGCCCCATAAAG
GGACTGTCTGGGGAGCGCCTCGTAGCCATAGAATTCCACCGCCGCCGCCCGCCGCGTAGTCGTACTTGAAGCCGAGCGCAGGCGGGTGGTTCATTAACT
CTGACTTTGCCTTTGATTTTGCTCGACCTCTGCTTCGTCAAAATCTGGTTTCAGAATCGAAGGATGAAGATGAAAAAGATGAATAAAGGAGGAAAAGGA
AGAAAACAAGGACTAAGCAAAAAAGAAAGACCCCCCCTATAGCAGGATTTTAAAATTTTTCTCTTTTTCTTTTTCAAGATTATTGCAAGGCGAGCGTGG
TGCAATATCCCGACTGTAAATCCTCCGCCAACACTAACTTTTAAAAAAAAACACCCAGCAGGTACCATGCTAAGACAACATCACATGCATTATTATGAC
TCACGTATACAATACAAAGTACTTGGACCAGGAACAGGGTCTTTAATCCTTATTTGACACGAAAACGTCGGATTGAAGTCTCTCATGCCCAACTAGTGG
GGTTTCCTGGCACTGGACCCCAGCAAGTGGTCCTAGAGGCGAAAAGGAAGAAAACAAGGACTAAGCAAAAAAGAAAGACCCCCCCCCCTGAAAGGATA
TCATGGATTCCTTAAAGAATGAGAACTTCGACATGGTGATATCACATGGACTTCTGGGGCCGAGTGAAGAATTTTCTGATGTTCTTTAGTTAGCAGTCC
TAAACCCTACCCAGCCTGCTGCCTCAGCACAGCCAAGGGAAAATCAGCAAGGGCTATGCTGTGATTCTGTCTCTGAGTGACTTGGACCACTGTTGATTT
TTATTTTTTTCTCTTTTTTTCTCCTATAGCAGGATTTTAAAATCGGGCCCACCTTAACTCGGGAGGGCCGCGCTGAGGCTGGGAGCCGGAGATTCGGGC
GAGGGCAGTGTCTGCGGGGCGCGGTCGCGCAGCTCCCCGGGCGAGCCAGGTGCAGCCTTGGCGGGGTCTGTTTGGTGGGCGATGTCACCATTTCCCGCC
GCCGCCGTCGCCACCGCCGCCGCCCGCCGCGTAGTCGTACTTGAATAGCTGGACATAAAGACAAATGACAAAAAATTATTATTATAGATATATTTGGTC
TGTGTGTTATGTCCTAAGGTGTTTTGTCTGCAGTTTGAGAGCATGTTGCTGGTAGCCTGAGTTGGAGAT
```

The problem we want to solve

```
GGTACCCCCAAATTCGAAAGAGCCGAATGAAGTGGAAAGAATGGCCATTCAAATGGCTTGGGA
```

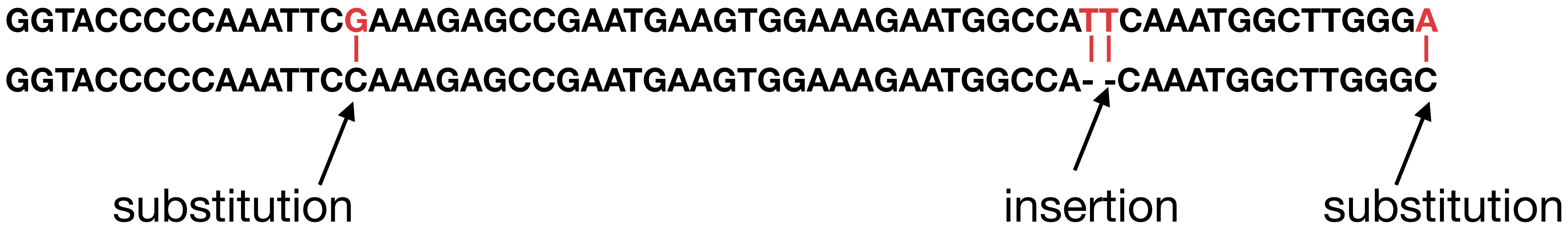
```
CAGGGCGCGCCGCTGGCCGAGAAAACCGCATTACAAATACTCTCAAAGCAGGATTTCTTTCTATAGGTAGTCAGTCCACAATTTTCACATTTGCAGCCA
GGTGGACATCTTTGGGCCAATGATGAGGAGAAGGTGGAGGAAGAAGGAAATGAGGAAGAGGAGAAAGAAGAGTATGGCATTTTACAAACTGTGACCGTT
TCTGTGTGAAGATTTTTAGCTGTCTGTCGGACTTGGGGGTCTCAGGGAAACTCACTTTGCCCCCCAGCTGAGGTTTTCAGGAAATCTGGAAACCTACAG
TCTCCAAGCCTGCTCAGCCAAGAACGCGGCGGGCGGCGGCGGTGGCGACGGCGGCGGCGGCGCAGGACCTCCGCGCCTCCATTATGCTATTCTGCCCGC
CGTGGGTGACAAACAGACGGATGCTACCAGCCCAGCCCAGTCCCGGGGAGCCAGCTGGCCTGGGGTTCGGTCCCGCGTCTTCCCTCATTCTGTGCCGCT
GCCGAGCCTGTCTCAGCTCCACACACGCTTGGGAGCTGCAGATGCCTCCGCCCCTCCTCTCTCCCAGGCTCTTCCTGCCGTTGAACCCCGGCGGCGGGC
TCTCGGCCAGCGGCGCGCCCTGGTACCCCCAAATTCCAAAGAGCCGAATGAAGTGGAAAGAATGGCCACAAATGGCTTGGGCCCGAGTGACCATGGGAT
GGTTAGGTAGGATTTTAGAGGCGACTGCTCCTGGAATTAGAGAAAGAGTTTCATTACAACCGCTACCTGACCCGAAGGCACTCGGGCCCATGCCTTCCT
CTCCTTCGCTGTTTGATTTCTATTCTGTTTAGCCTGAGTTGGAGATGCCGGAGGTGCCGTGGTGGAAGAAGTCTTGAACGAATTTGGAGGCGTCTCCGT
GGCAGCTAAGCGAGCACGGGTTCTGCTGGTGCAGGATGACACTGGCAGCCACTGCCGCGGACTTGTATCTCTTGTCTTCCTGCTTTTATAGAGAATAGA
ATGACACTCACAACTCTAACTACCTGTCAGAAGCAGGCAGGAGCTAGTAAGGATGAATTTGTAGCAAAATTAGCAAGTGGACTTCTTTCTCCTCTTCCT
CATTTCCTTCTTCCTCCACCTTCTCCTCATCTTAAATCTTTAACATACTACCTAAAGGGAACCTGCAATAATCTTGAAAAAGGACTTCAATCCGACGTT
TTCGTGTCAAATAAGGATTAAAGAGAAACTCCTCCGCGAGCCGTGCGCCGAGGGTGGCGGCGGGGGCCTGAAGCGTGAGGAGCCTTCAATATGTATTTA
ACCAGGGACCGTCGGTATGAGGTGGCCCGGGTTCTTATTTGTTTGGGGGCTGGAGGGGGGAGACGGAGAAACAGTGAAAAGTTCCTGAGCCCCATAAAG
GGACTGTCTGGGGAGCGCCTCGTAGCCATAGAATTCCACCGCCGCCGCCCGCCGCGTAGTCGTACTTGAAGCCGAGCGCAGGCGGGTGGTTCATTAACT
CTGACTTTGCCTTTGATTTTGCTCGACCTCTGCTTCGTCAAAATCTGGTTTCAGAATCGAAGGATGAAGATGAAAAAGATGAATAAAGGAGGAAAAGGA
AGAAAACAAGGACTAAGCAAAAAAGAAAGACCCCCCCTATAGCAGGATTTTAAAATTTTTCTCTTTTTCTTTTTCAAGATTATTGCAAGGCGAGCGTGG
TGCAATATCCCGACTGTAAATCCTCCGCCAACACTAACTTTTAAAAAAAAACACCCAGCAGGTACCATGCTAAGACAACATCACATGCATTATTATGAC
TCACGTATACAATACAAAGTACTTGGACCAGGAACAGGGTCTTTAATCCTTATTTGACACGAAAACGTCGGATTGAAGTCTCTCATGCCCAACTAGTGG
GGTTTCCTGGCACTGGACCCCAGCAAGTGGTCCTAGAGGCGAAAAGGAAGAAAACAAGGACTAAGCAAAAAAGAAAGACCCCCCCCCCTGAAAGGATA
TCATGGATTCCTTAAAGAATGAGAACTTCGACATGGTGATATCACATGGACTTCTGGGGCCGAGTGAAGAATTTTCTGATGTTCTTTAGTTAGCAGTCC
TAAACCCTACCCAGCCTGCTGCCTCAGCACAGCCAAGGGAAAATCAGCAAGGGCTATGCTGTGATTCTGTCTCTGAGTGACTTGGACCACTGTTGATTT
TTATTTTTTTCTCTTTTTTTCTCCTATAGCAGGATTTTAAAATCGGGCCCACCTTAACTCGGGAGGGCCGCGCTGAGGCTGGGAGCCGGAGATTCGGGC
GAGGGCAGTGTCTGCGGGGCGCGGTCGCGCAGCTCCCCGGGCGAGCCAGGTGCAGCCTTGGCGGGGTCTGTTTGGTGGGCGATGTCACCATTTCCCGCC
GCCGCCGTCGCCACCGCCGCCGCCCGCCGCGTAGTCGTACTTGAATAGCTGGACATAAAGACAAATGACAAAAAATTATTATTATAGATATATTTGGTC
TGTGTGTTATGTCCTAAGGTGTTTTGTCTGCAGTTTGAGAGCATGTTGCTGGTAGCCTGAGTTGGAGAT
```

The problem we want to solve



```
GGTACCCCCAAATTCGAAAGAGCCGAATGAAGTGGAAAGAATGGCCATTCAAATGGCTTGGGA
               |                               ||              |
GGTACCCCCAAATTCCAAAGAGCCGAATGAAGTGGAAAGAATGGCCA- -CAAATGGCTTGGGC
```

The problem we want to solve



```
GGTACCCCCAAATTCGAAAGAGCCGAATGAAGTGGAAAGAATGGCCATTCAAATGGCTTGGGA
GGTACCCCCAAATTCCAAAGAGCCGAATGAAGTGGAAAGAATGGCCA- -CAAATGGCTTGGGC
```

substitution

insertion

substitution

Given: A *reference* text (e.g. genome) and a query (e.g. read)

Find: The location in the reference with the “most similarity” / “smallest distance” to the query.

The Language of Strings

A string **s** is a finite sequence of characters

|**s**| denotes the length of the string — the number of characters in the sequence.

A string is defined *over* an alphabet, Σ

Σ_{DNA} = {A,T,C,G}
Σ_{RNA} = {A,U,C,G}
$\Sigma_{AminoAcid}$ = {A, R, N, D, C, E, Q, G, H, I, L, K, M, F, P, S, T, W, Y, V}

The empty string is denoted ϵ — |ϵ| = 0

Slides below with +/* adopted from Ben Langmead & Carl Kingsford, respectively

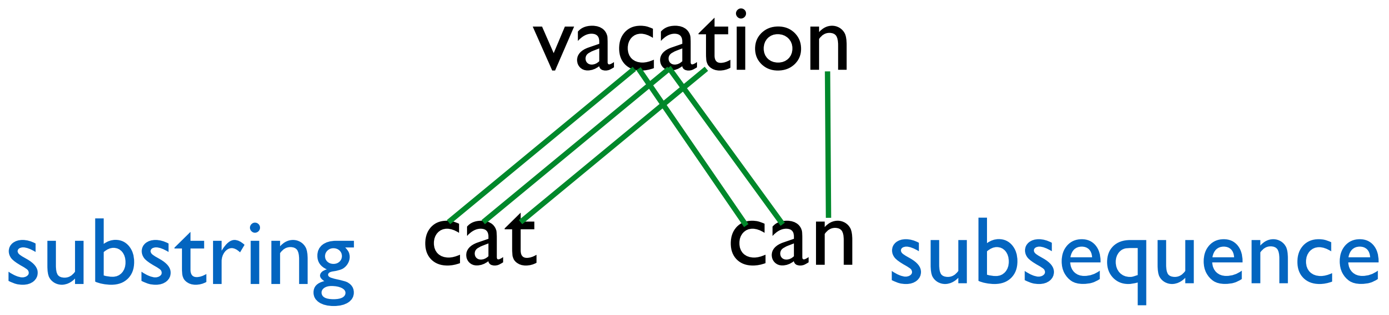
+

The Language of Strings

Given two strings **s,t** over the same alphabet Σ, we denote the concatenation as **st** — this is the sequence of **s** followed by the sequence of **t**

String **s** is a substring of **t** if there exist two (potentially empty) strings **u** and **v** such that **t = usv**

String **s** is a subsequence of **t** if the characters of **s** appear in order (but not necessarily consecutively) in **t**



vacation

substring cat

can subsequence

String **s** is a prefix/suffix of **t** if **t** = **su/us** — if neither **s** nor **u** are ϵ, then **s** is a proper prefix/suffix of **t**

The Simplest String Comparison Problem

Given: Two strings

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

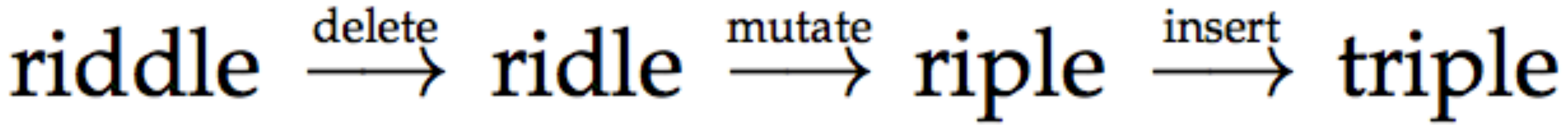
where a_i, b_i are letters from some alphabet, Σ, like {A,C,G,T}.

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



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

*

The String Alignment Problem

Parameters:

- "*gap*" is the cost of inserting a "-" character, representing an insertion or deletion (insertion/deletion are dual operations depending on the string)

- *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**. (often phrased as finding **highest scoring alignment**.)

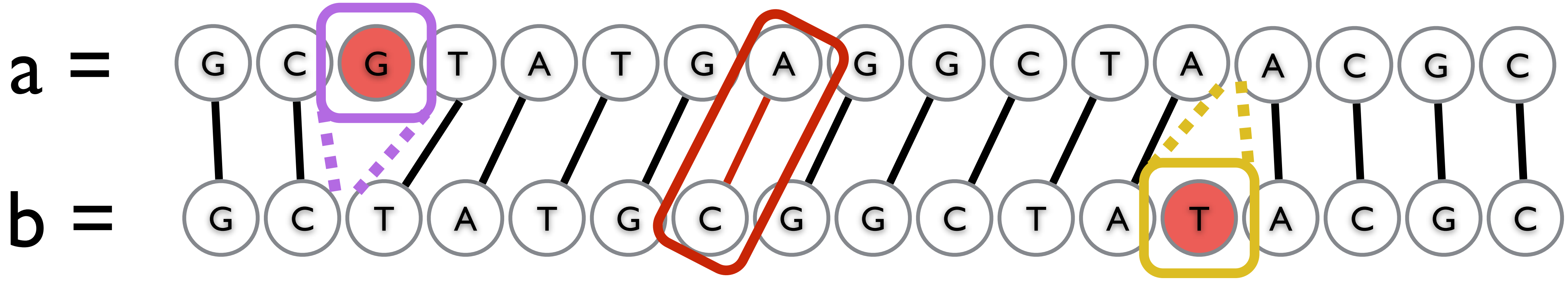
- Cost of an alignment is: sum of the *cost*(*x*,*y*) for the pairs of characters that are aligned + *gap* × number of - characters inserted.

*

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.


a =
G C G T A T G A G G C T A A C G C
b =
G C T A T G C G G C T A T A C G C

The operations at our disposal

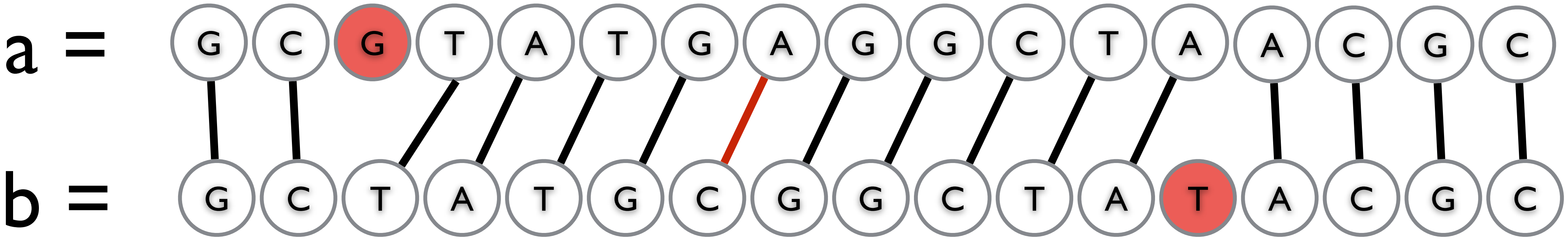
- Insertion (into **a** ~ deletion from **b**)
- Mutation
- Deletion (from **a** ~ insertion into **b**)

When we “delete a” character in **a** this is the same as inserting the character “-“ in **b**. Conceptually, you can think of this as aligning the deleted character with “-“. Under this model cost(x,’-‘) = cost(‘-‘,x) = gap for any x ∈ Σ

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.


a =
G C G T A T G A G G C T A A C G C
b =
G C T A T G C G G C T A T A C G C

Cost of a matching is:

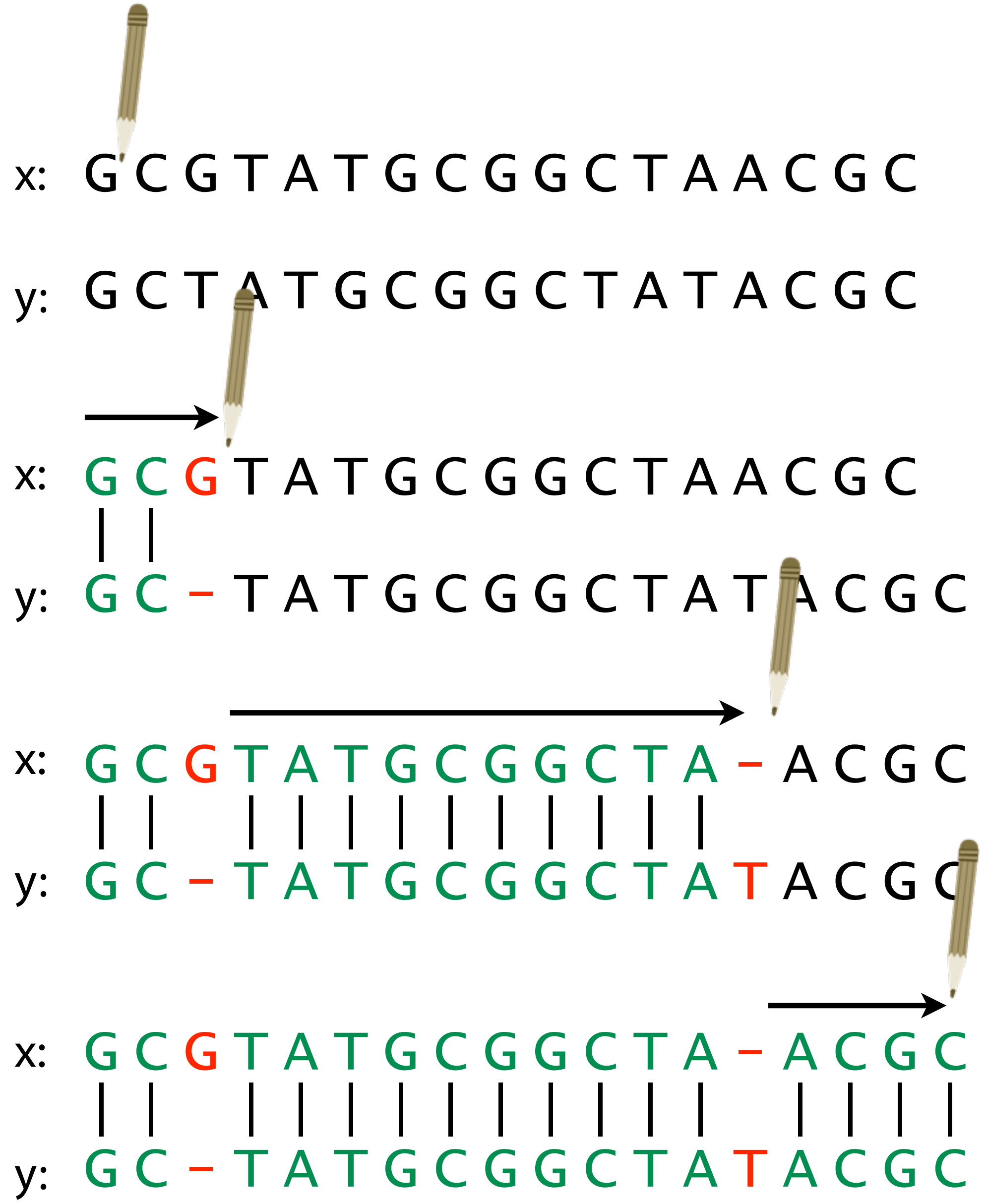
$$\text{gap} \times \#\mathit{unmatched} + \sum_{(a_i, b_j)} \mathit{cost}(a_i, b_j)$$

Edges are not allowed to cross!

*

Representing alignments as edit transcripts

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
```

MMD

```
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
```

MMDMMMMMMMMMMI

```
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
```

MMDMMMMMMMMMMIMMMM

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

Slide courtesy of Ben Langmead

Representing edits as alignments

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

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

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

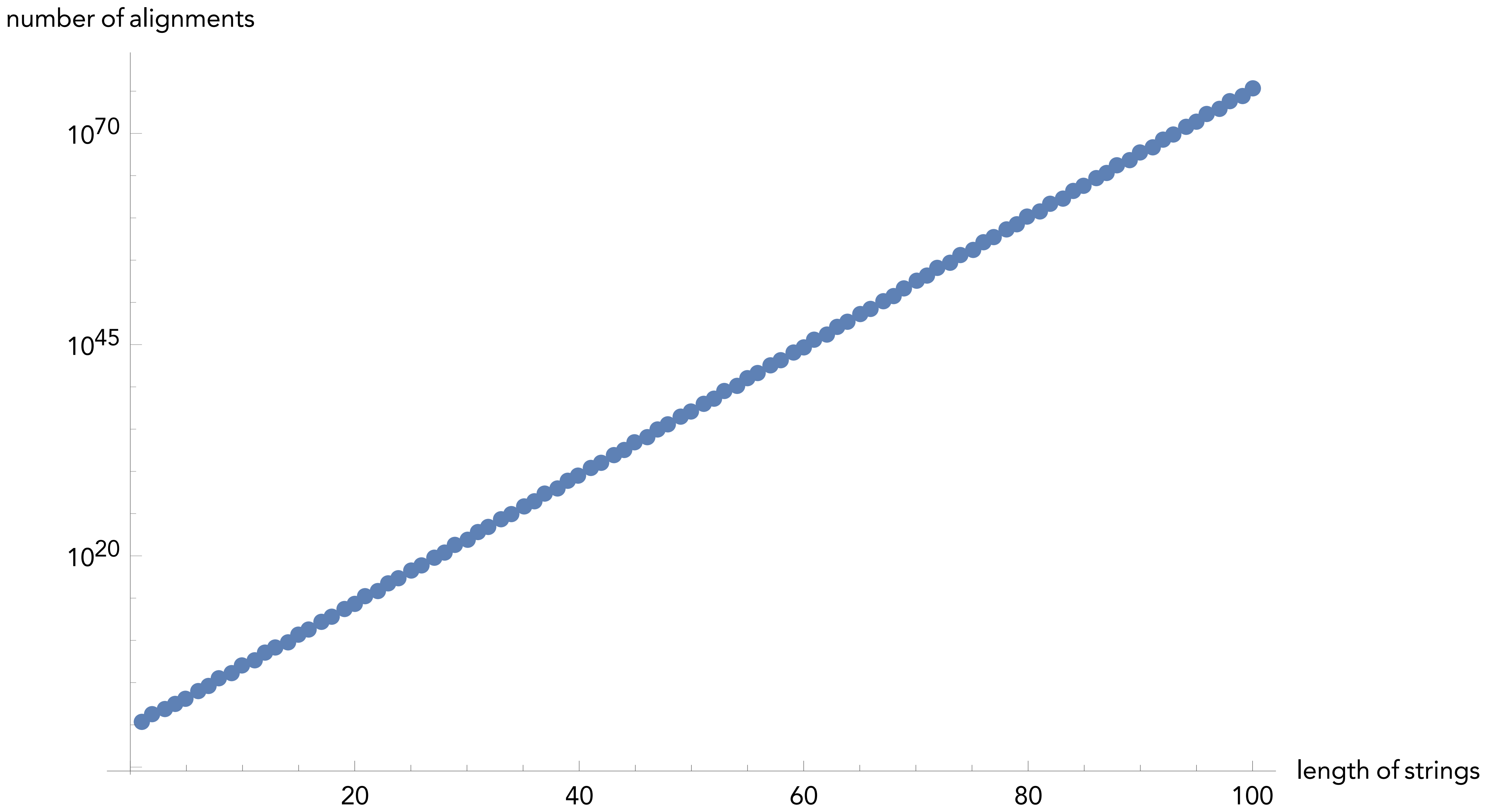
```
prehistoric
   ||||||||
---historic
(3 gaps)
DDDMMMMMMMM
```

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

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

*

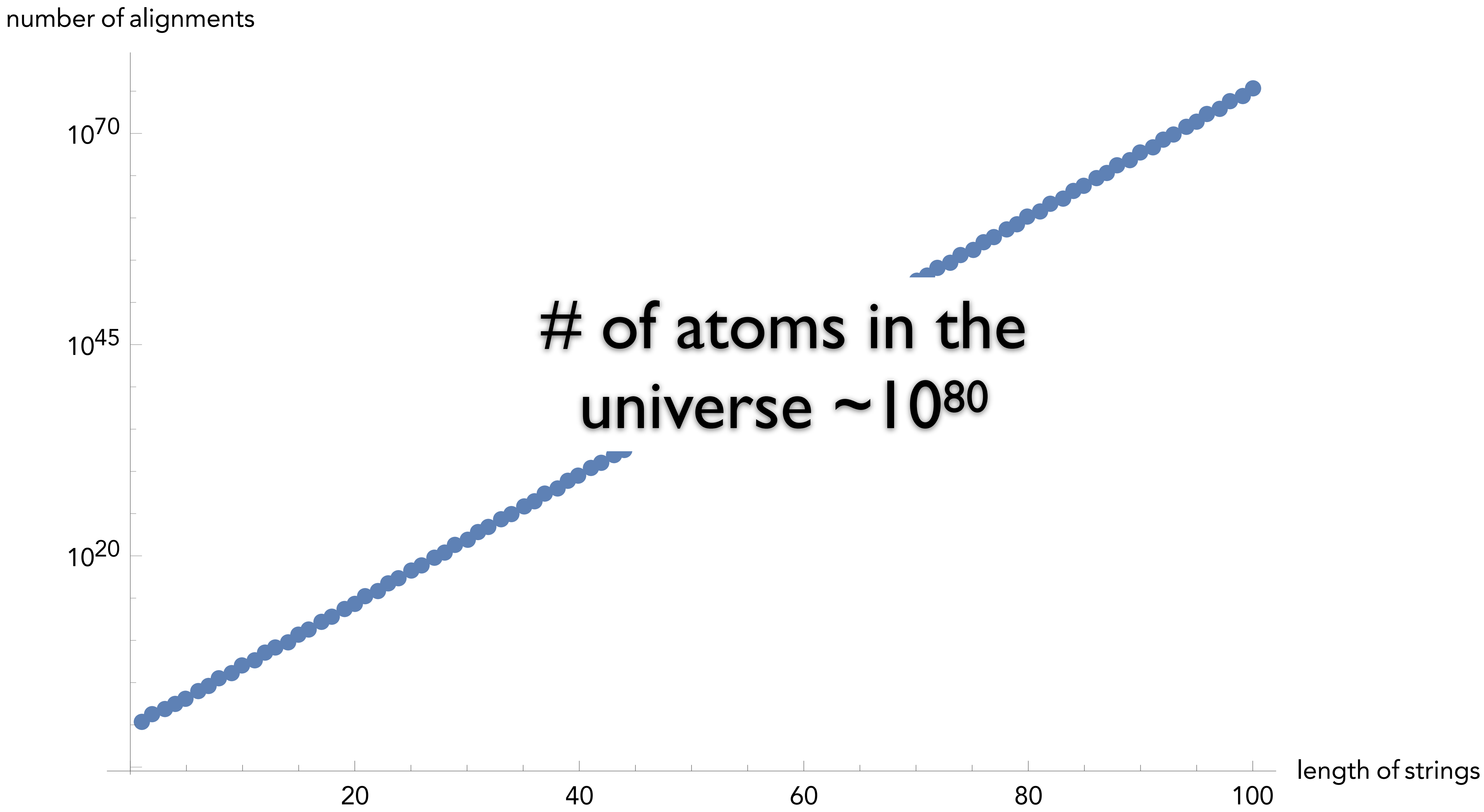
How many alignments are there?



$$f(n,m) = \sum_{k=0}^{\min(m,n)} 2^k \binom{m}{k} \binom{n}{k}$$

Andrade, Helena, et al. "The number of reduced alignments between two DNA sequences." BMC bioinformatics 15.1 (2014): 94.

How many alignments are there?



# of atoms in the universe ~10^{80}

$$f(n,m) = \sum_{k=0}^{\min(m,n)} 2^k \binom{m}{k} \binom{n}{k}$$

Andrade, Helena, et al. "The number of reduced alignments between two DNA sequences." BMC bioinformatics 15.1 (2014): 94.

Algorithm for Computing Edit Distance

Consider the last characters of each string:

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

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 $(k \neq m)$.

*

Algorithm for Computing Edit Distance

Consider the last characters of each string:

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

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 $(k \neq m)$.

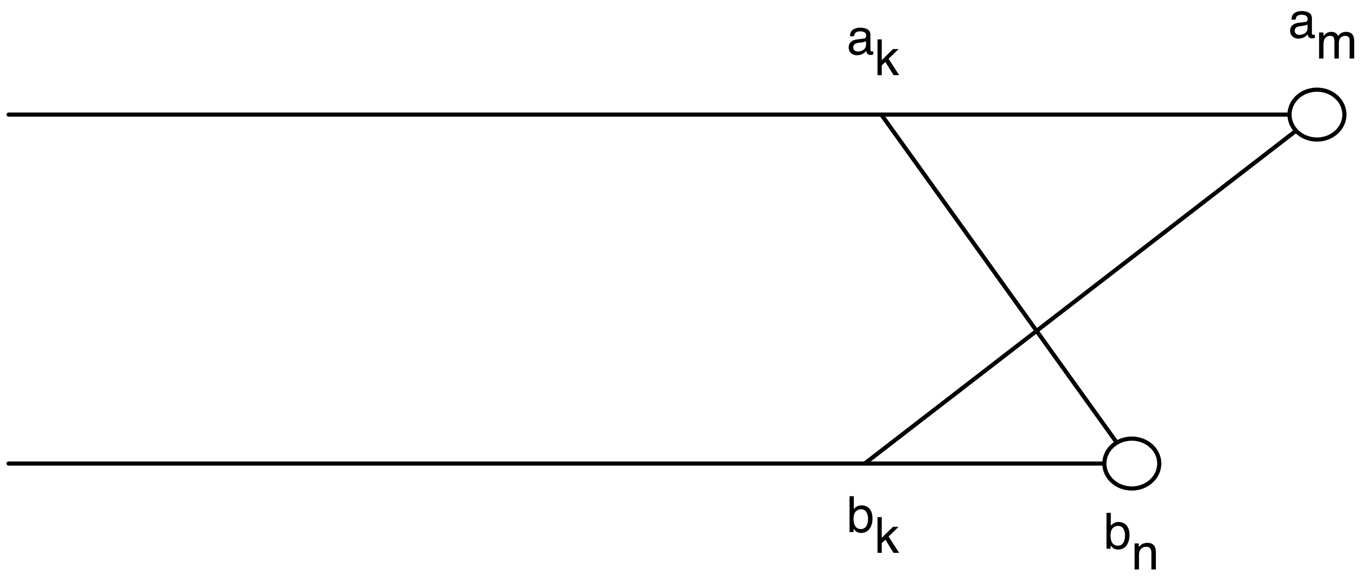
↑

#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:

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

Cost of the optimal alignment between $a_1 \ldots a_i$ and $b_1 \ldots b_j$

Written in terms of the costs of smaller problems

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

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

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

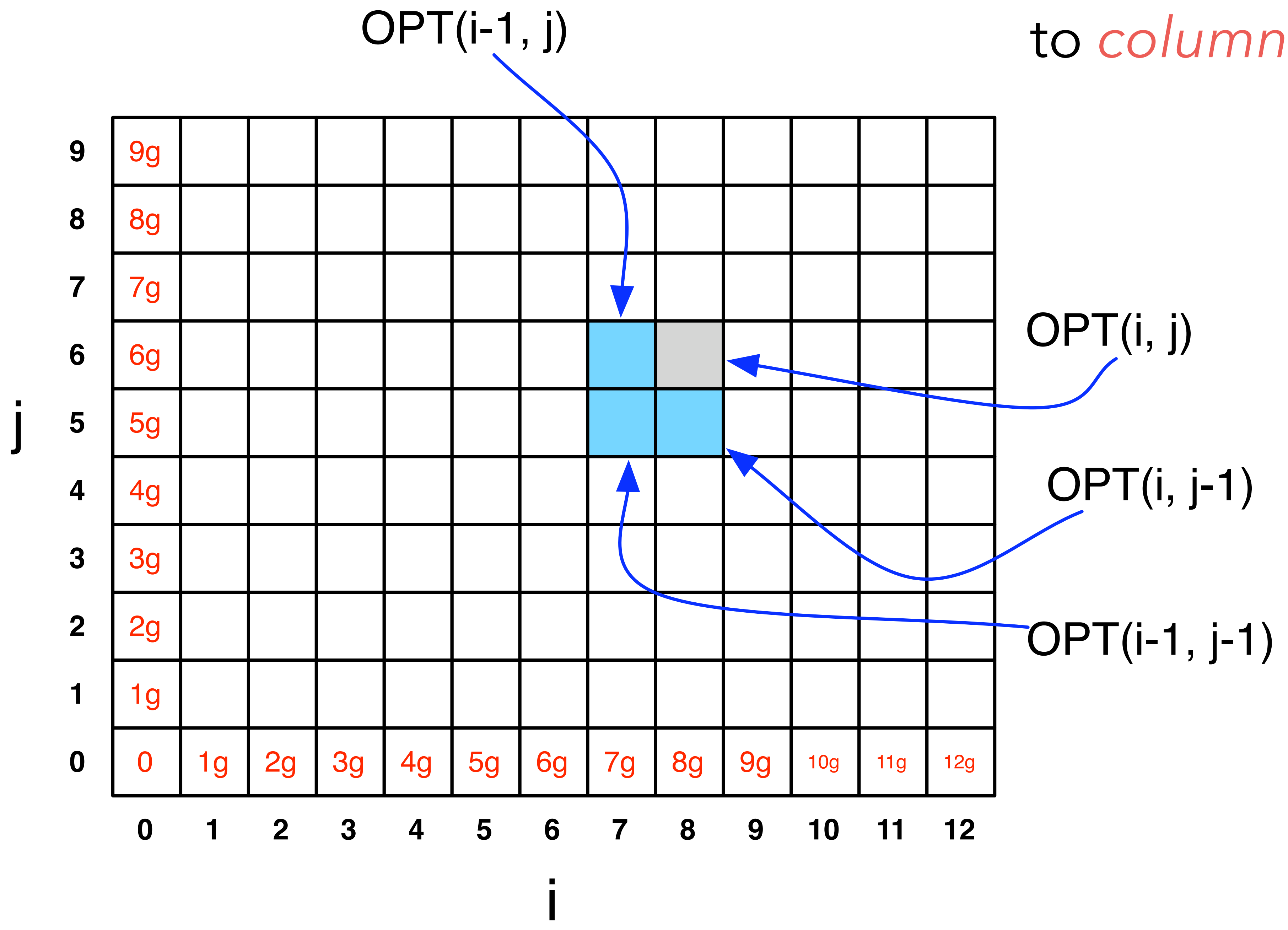
*

Computing OPT(i,j) Efficiently

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

Store those values in a 2D array:

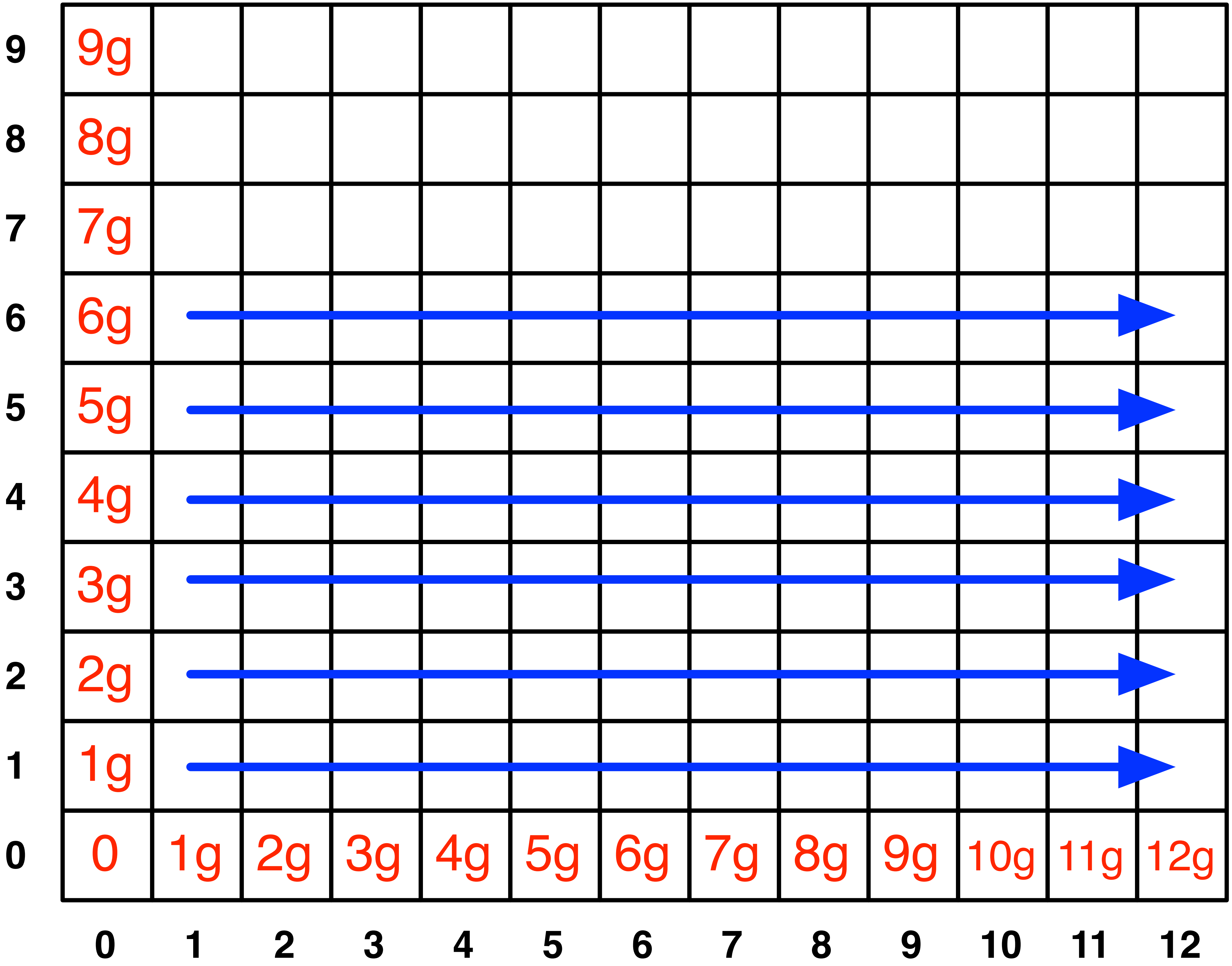
NOTE: observe the non-standard notation here; OPT(**i**,**j**) is referring to *column* i, *row* j of the matrix.



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

*

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

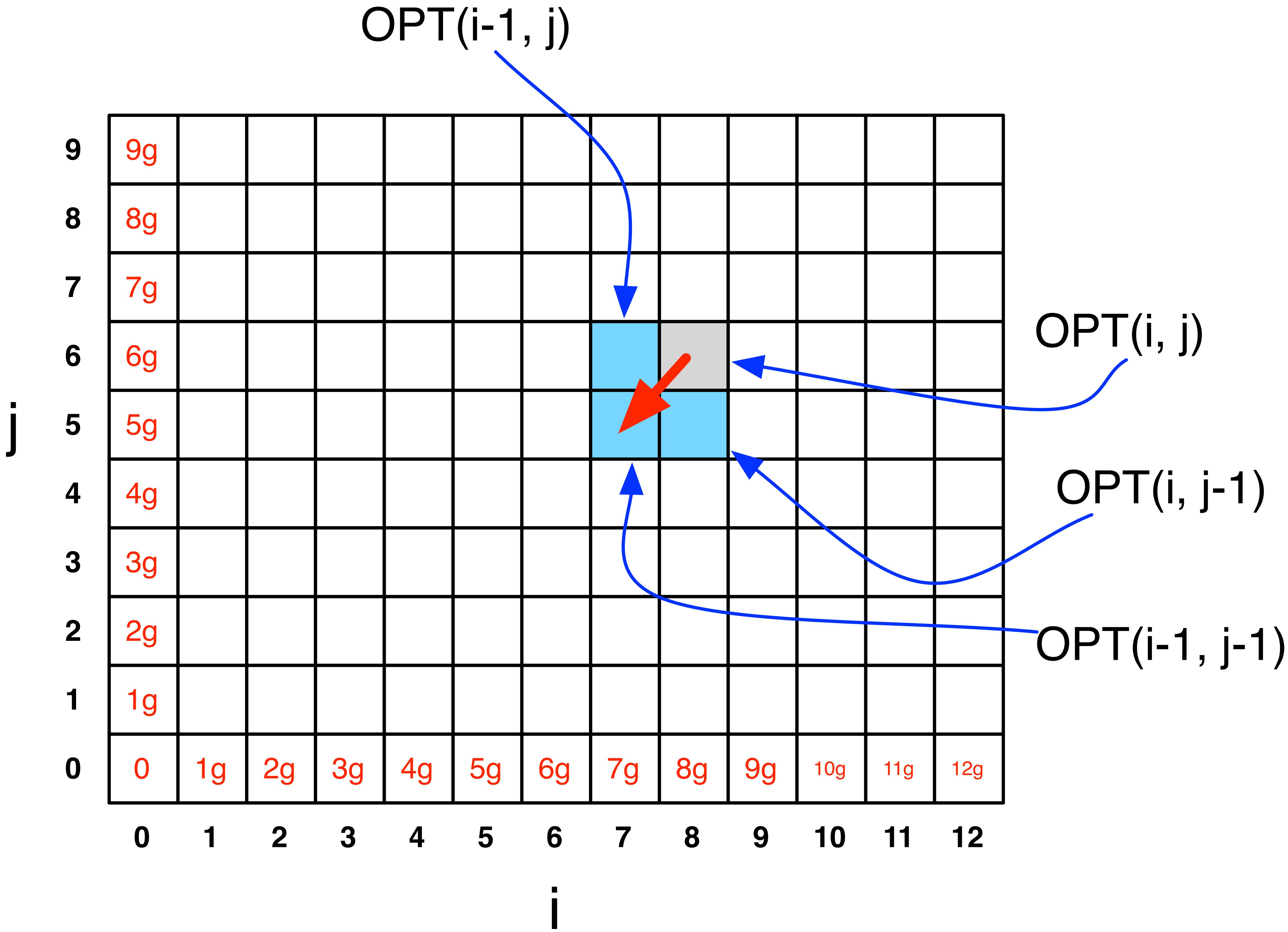
Number of entries in array = O($m \times n$), where m and n are the lengths of the 2 strings.

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

Total running time is O(mn).

*

Finding the actual alignment



OPT(i-1, j)

OPT(i, j)

OPT(i, j-1)

OPT(i-1, j-1)

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

*

Outputting the Alignment

Build the alignment from right to left.

```
ACGT
A-GA
```

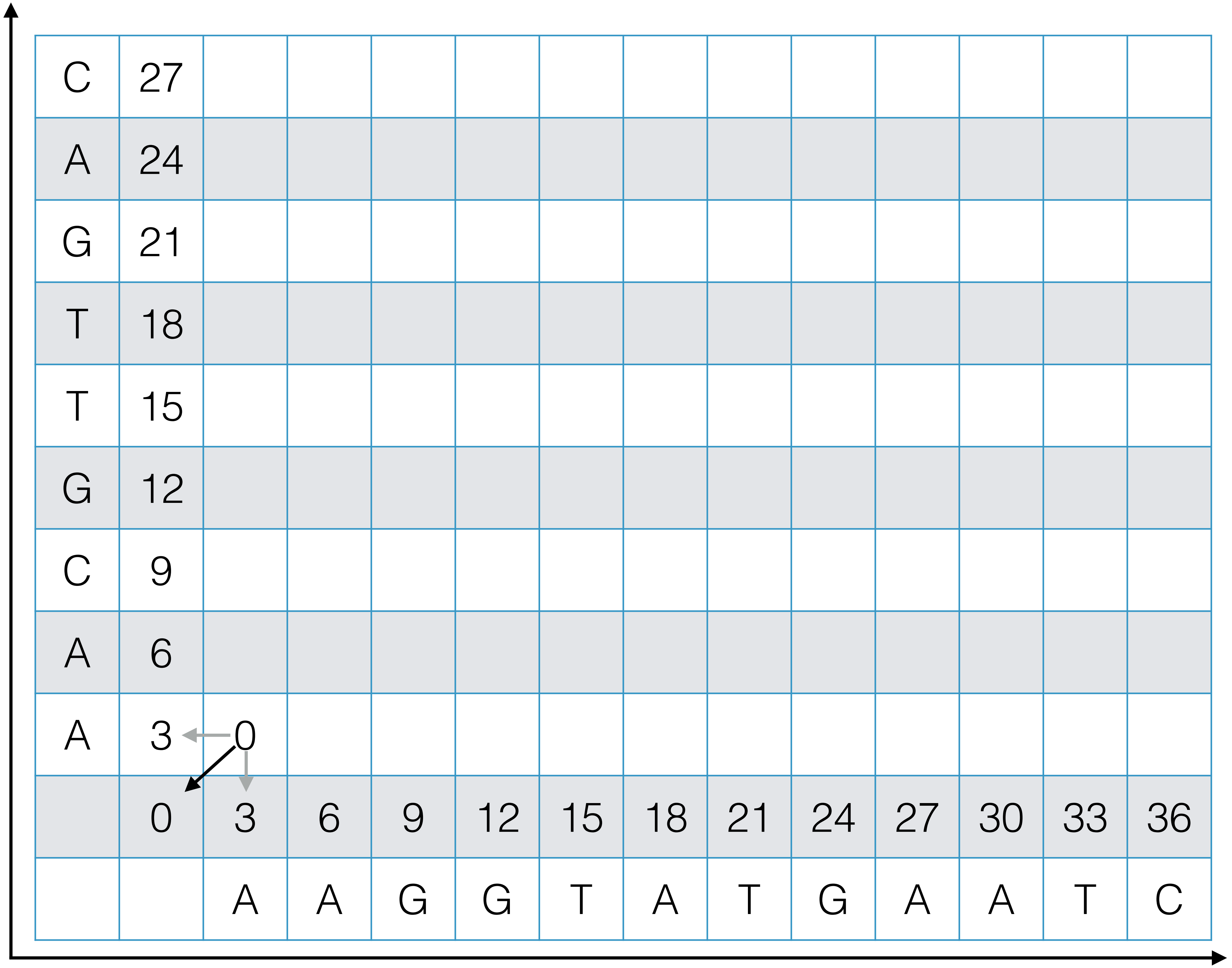
Follow the backtrack pointers starting from entry (n,m).

- 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.

*

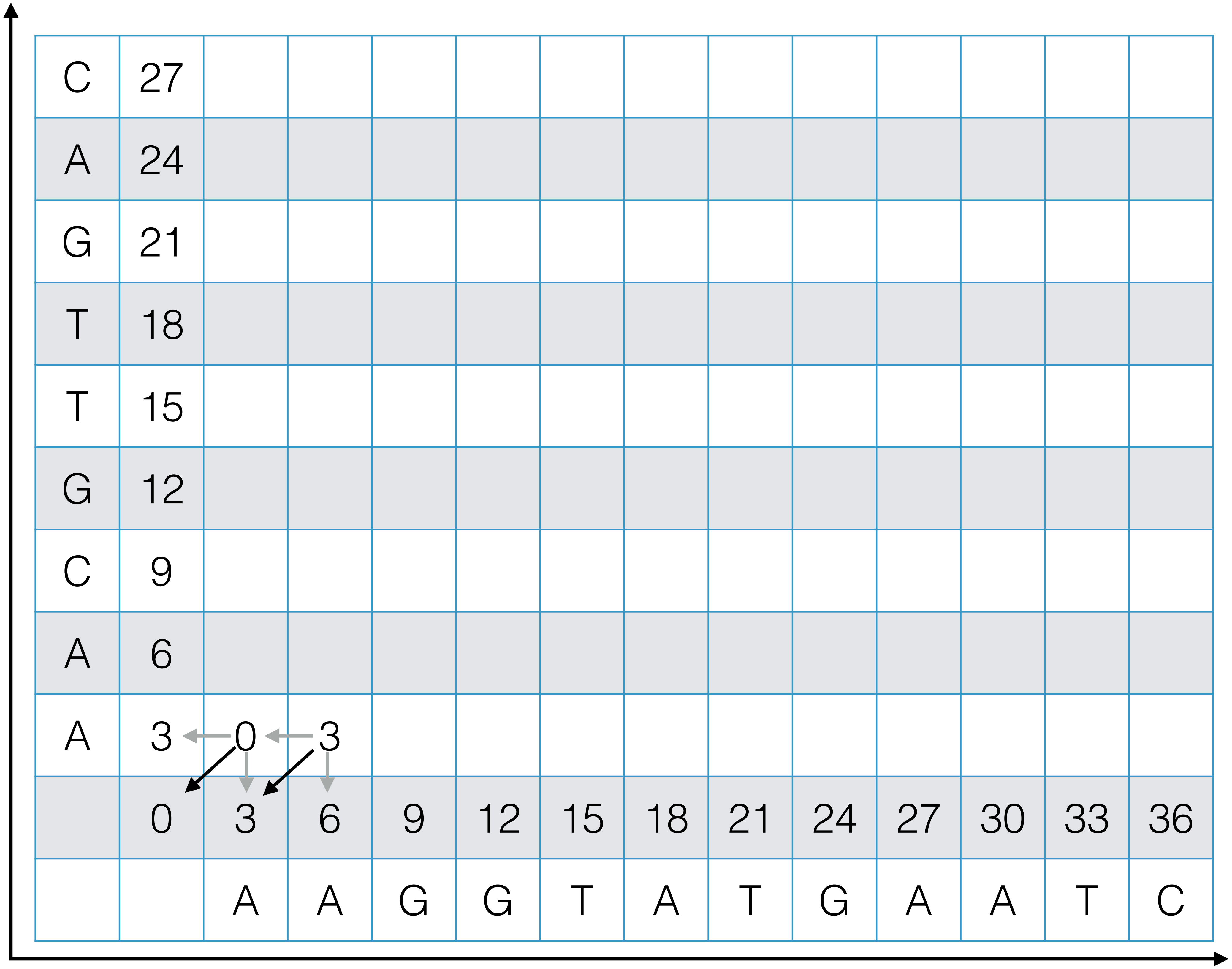
gap cost = 3
mismatch cost = 1

Example



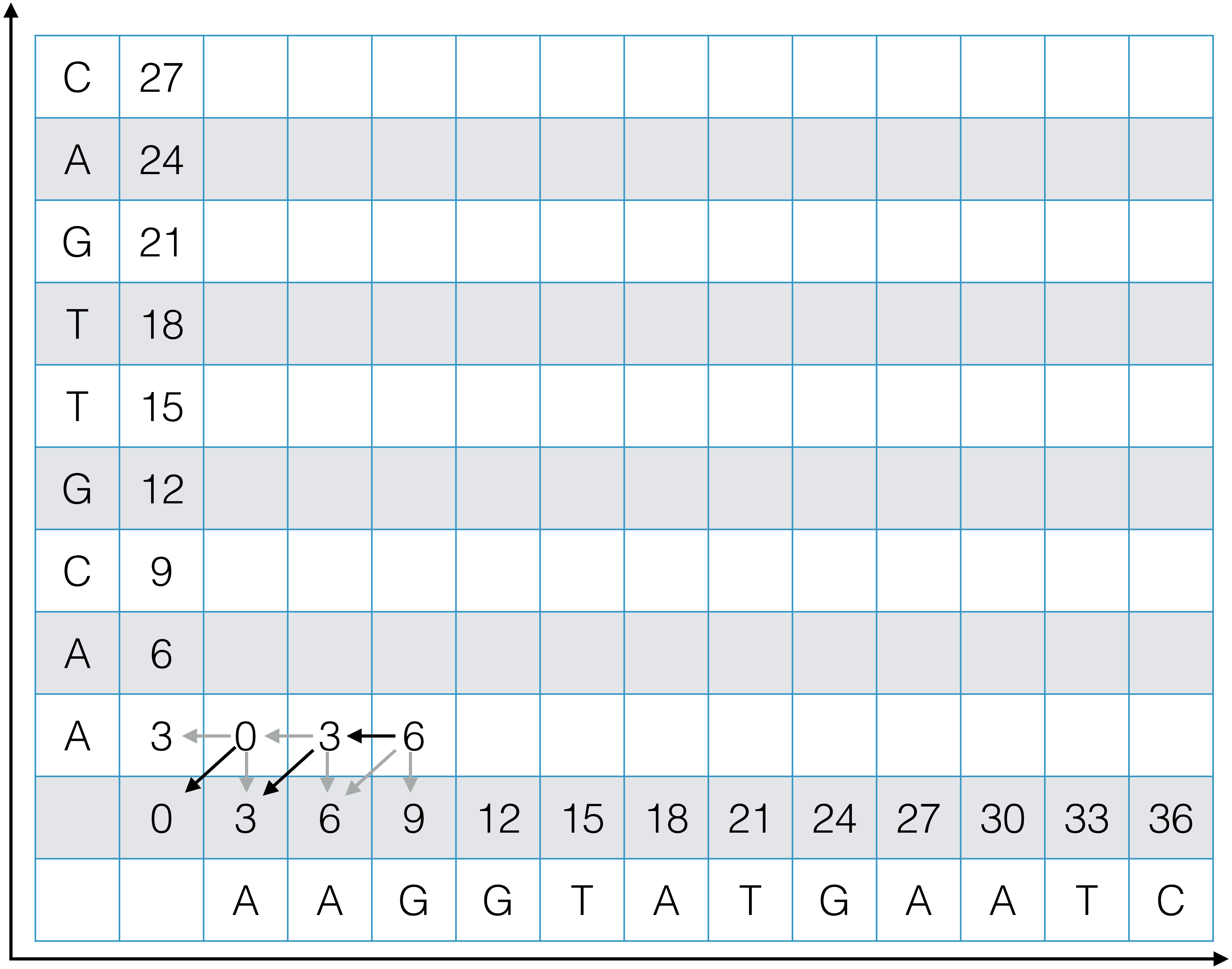
C	27												
A	24												
G	21												
T	18												
T	15												
G	12												
C	9												
A	6												
A	3	0											
	0	3	6	9	12	15	18	21	24	27	30	33	36
		A	A	G	G	T	A	T	G	A	A	T	C

Example



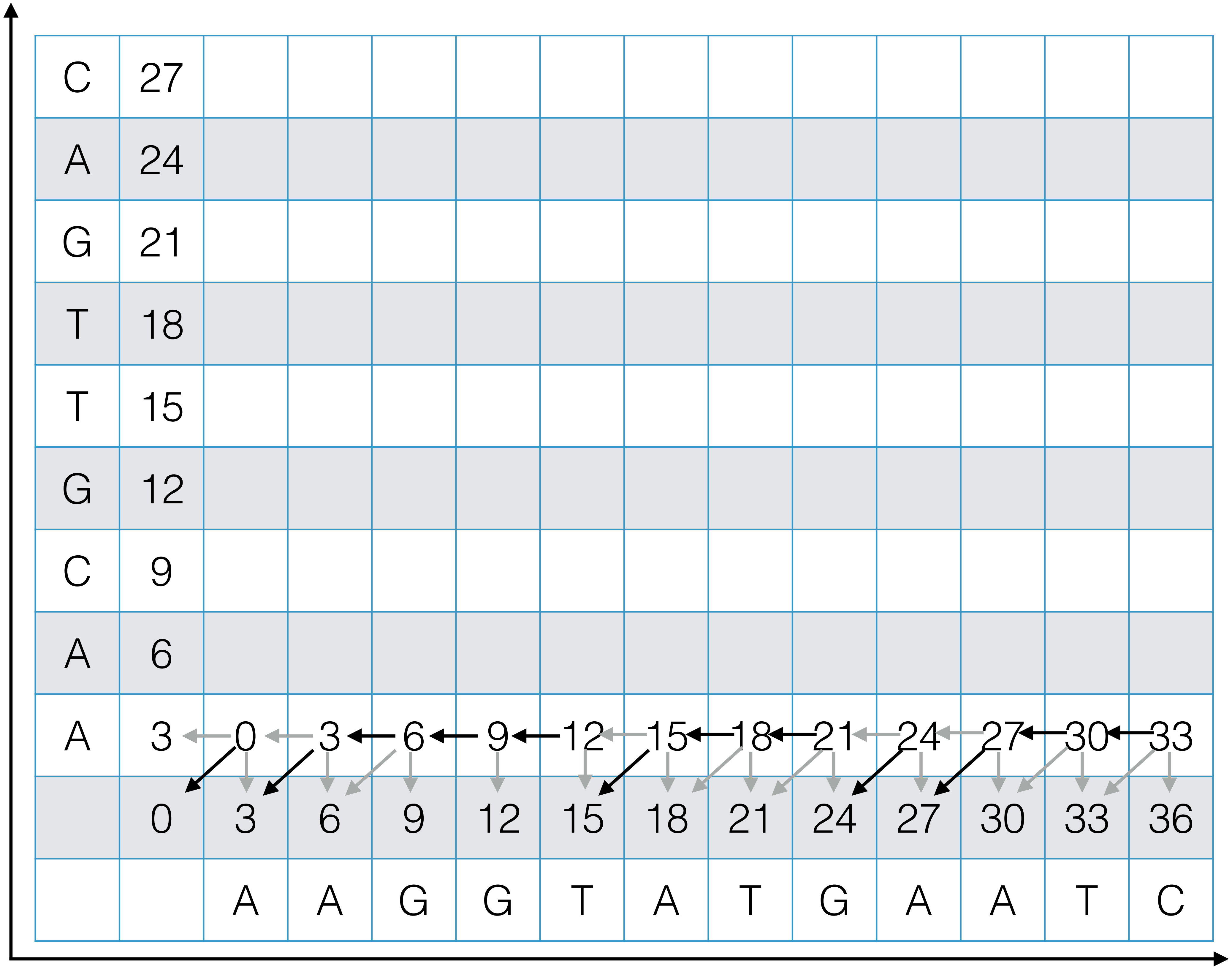
C	27												
A	24												
G	21												
T	18												
T	15												
G	12												
C	9												
A	6												
A	3	0	3										
	0	3	6	9	12	15	18	21	24	27	30	33	36
		A	A	G	G	T	A	T	G	A	A	T	C

Example



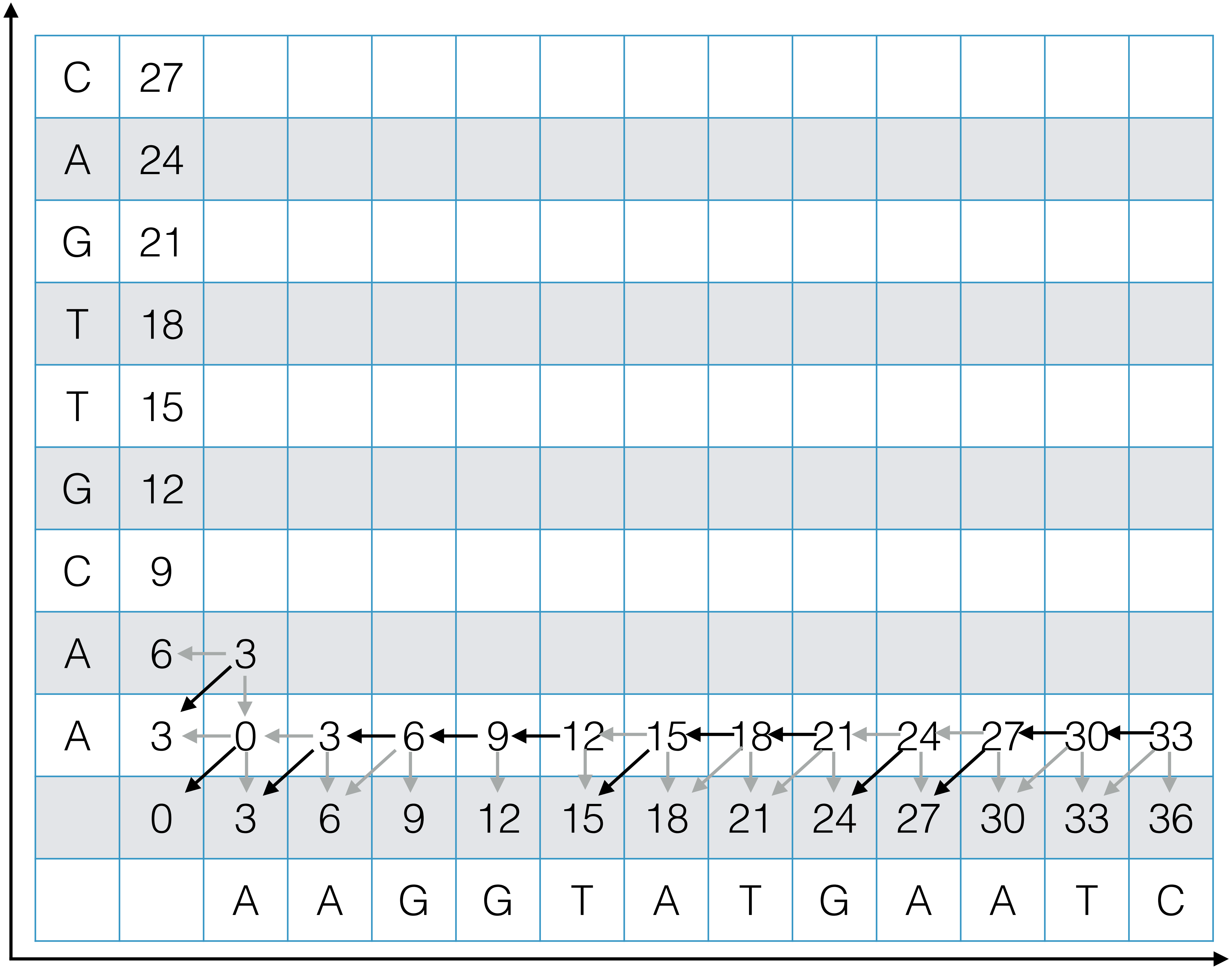
C	27												
A	24												
G	21												
T	18												
T	15												
G	12												
C	9												
A	6												
A	3	0	3	6									
	0	3	6	9	12	15	18	21	24	27	30	33	36
		A	A	G	G	T	A	T	G	A	A	T	C

Example



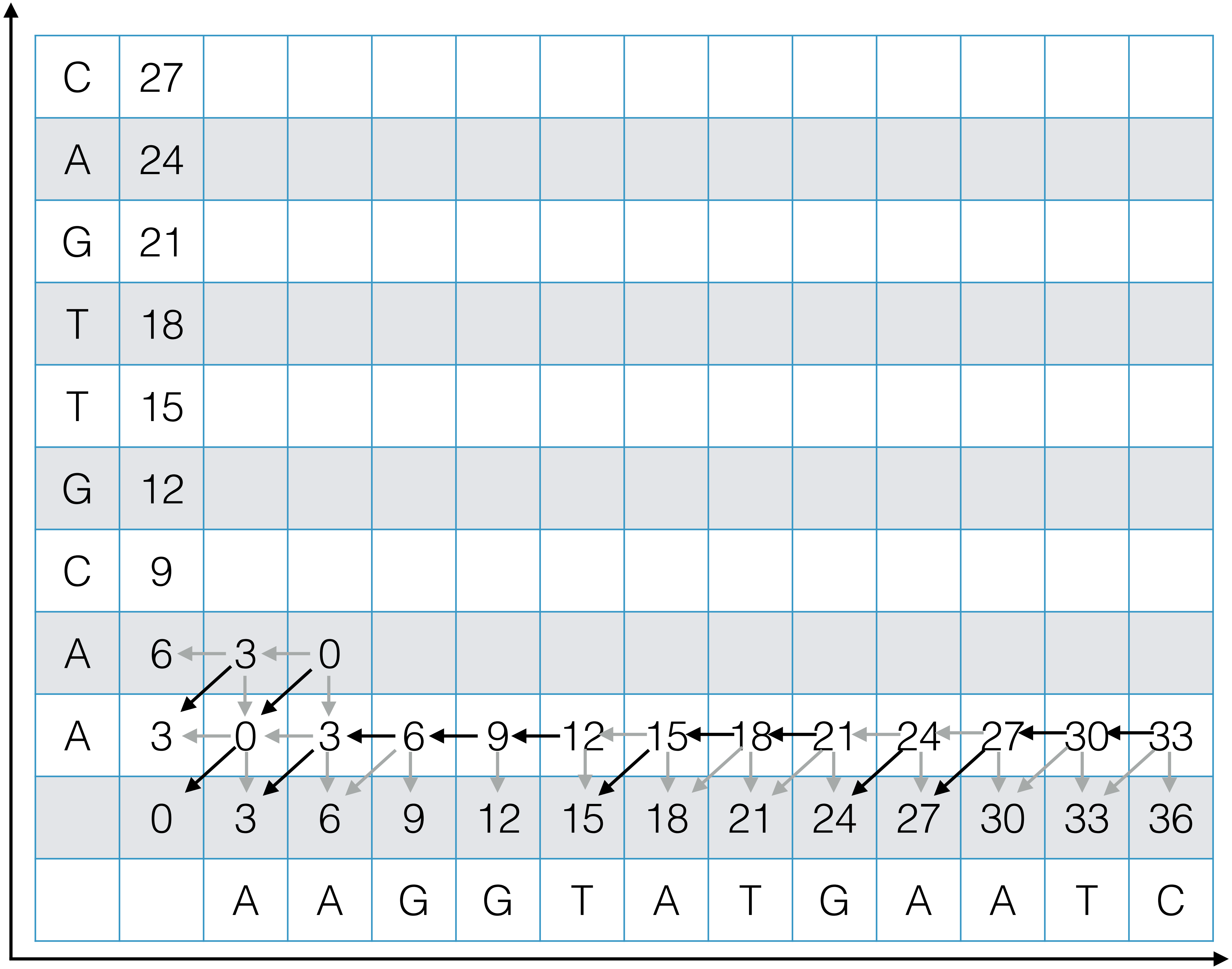
C	27												
A	24												
G	21												
T	18												
T	15												
G	12												
C	9												
A	6												
A	3	0	3	6	9	12	15	18	21	24	27	30	33
	0	3	6	9	12	15	18	21	24	27	30	33	36
		A	A	G	G	T	A	T	G	A	A	T	C

Example



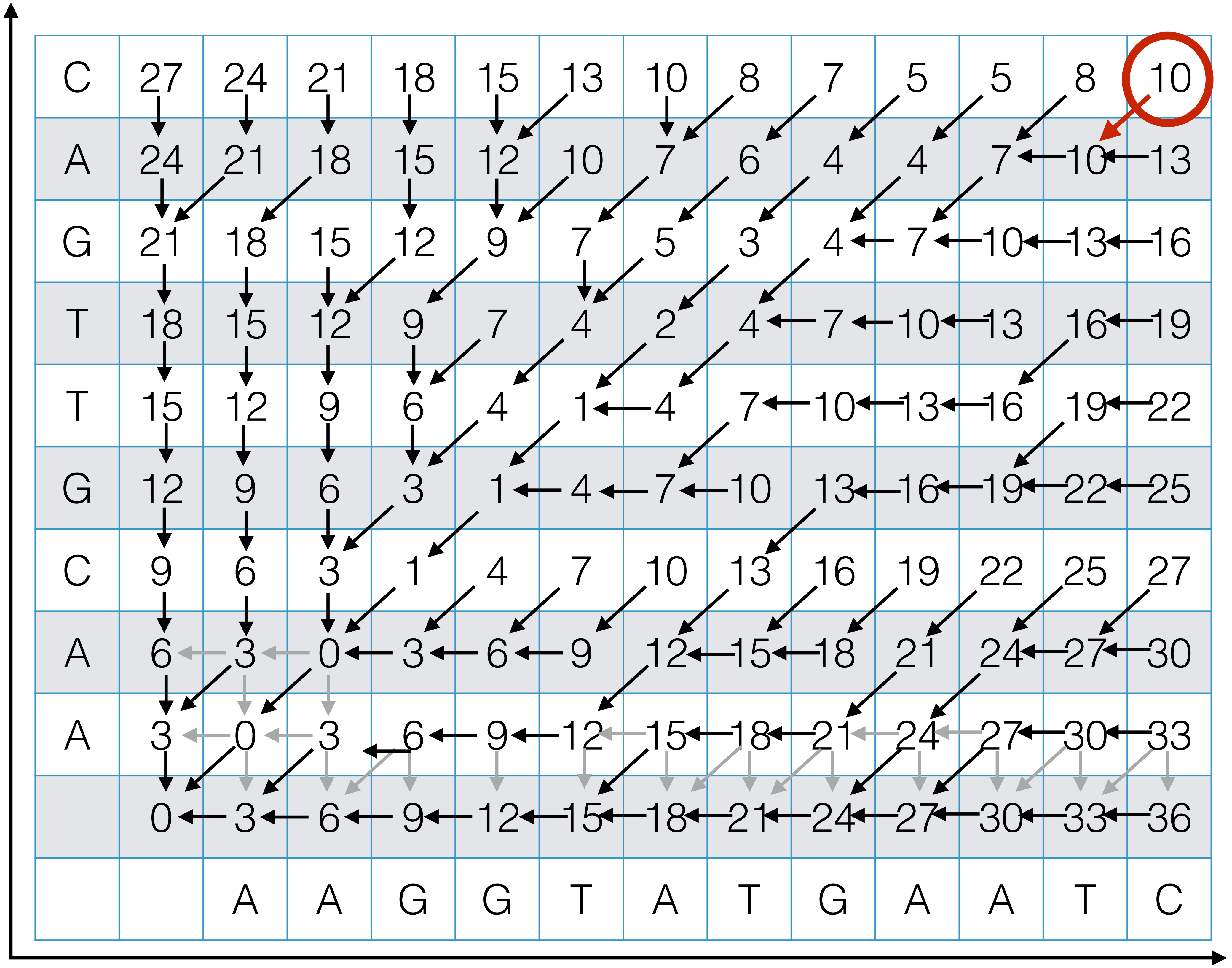
C	27												
A	24												
G	21												
T	18												
T	15												
G	12												
C	9												
A	6	3											
A	3	0	3	6	9	12	15	18	21	24	27	30	33
	0	3	6	9	12	15	18	21	24	27	30	33	36
		A	A	G	G	T	A	T	G	A	A	T	C

Example



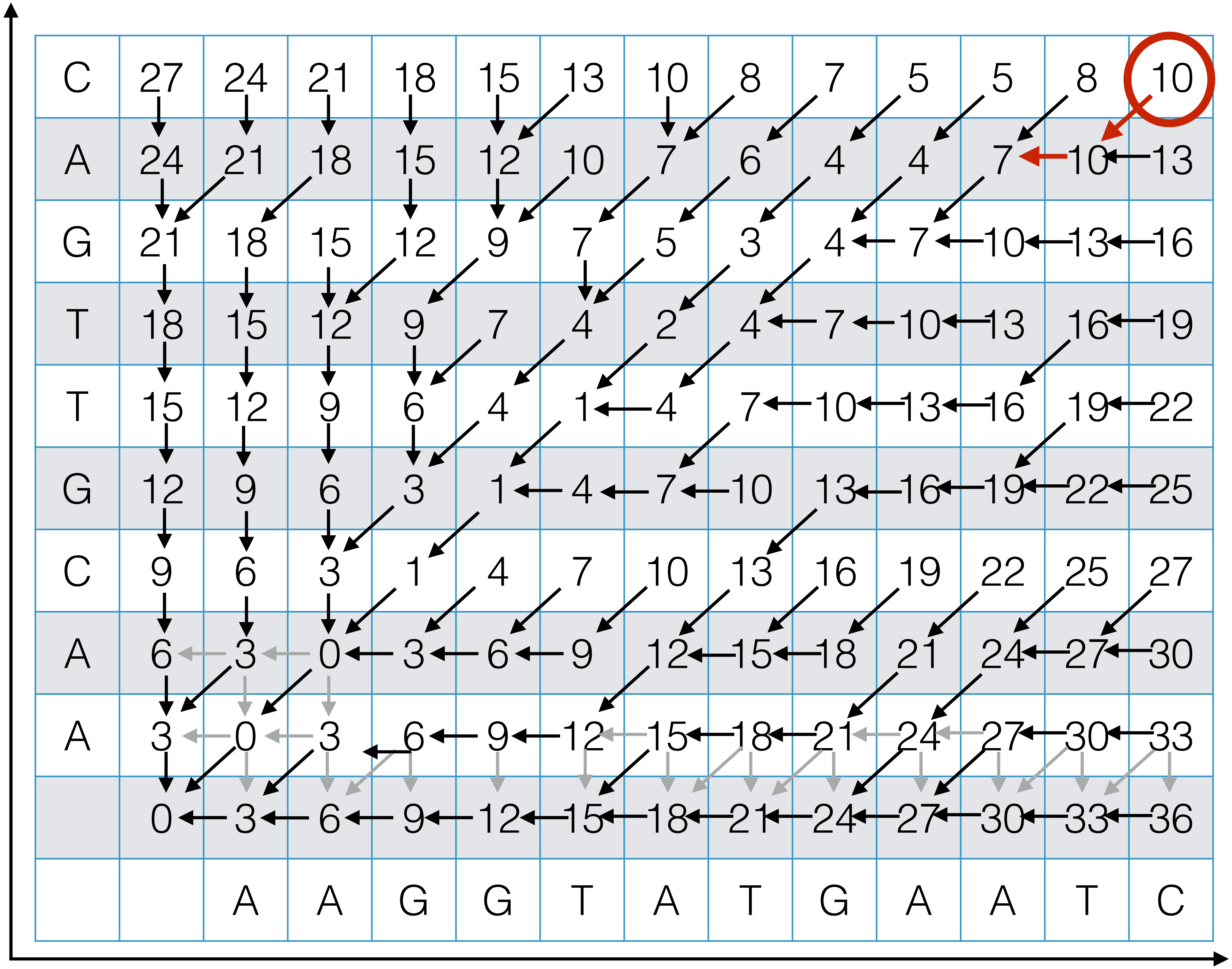
C	27												
A	24												
G	21												
T	18												
T	15												
G	12												
C	9												
A	6	3	0										
A	3	0	3	6	9	12	15	18	21	24	27	30	33
	0	3	6	9	12	15	18	21	24	27	30	33	36
		A	A	G	G	T	A	T	G	A	A	T	C

Example



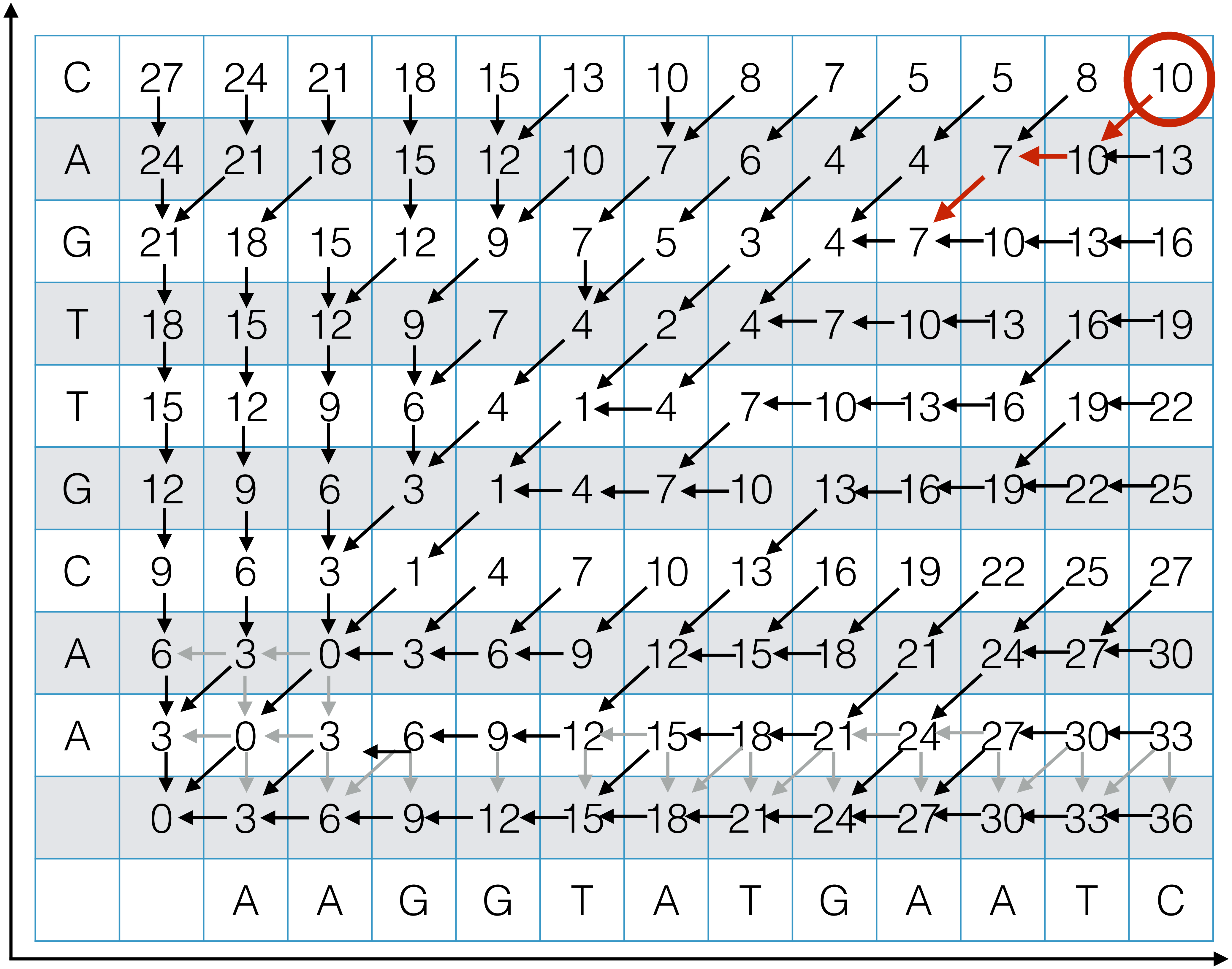
C	27	24	21	18	15	13	10	8	7	5	5	8	10
A	24	21	18	15	12	10	7	6	4	4	7	10	13
G	21	18	15	12	9	7	5	3	4	7	10	13	16
T	18	15	12	9	7	4	2	4	7	10	13	16	19
T	15	12	9	6	4	1	4	7	10	13	16	19	22
G	12	9	6	3	1	4	7	10	13	16	19	22	25
C	9	6	3	1	4	7	10	13	16	19	22	25	27
A	6	3	0	3	6	9	12	15	18	21	24	27	30
A	3	0	3	6	9	12	15	18	21	24	27	30	33
	0	3	6	9	12	15	18	21	24	27	30	33	36
		A	A	G	G	T	A	T	G	A	A	T	C

Example



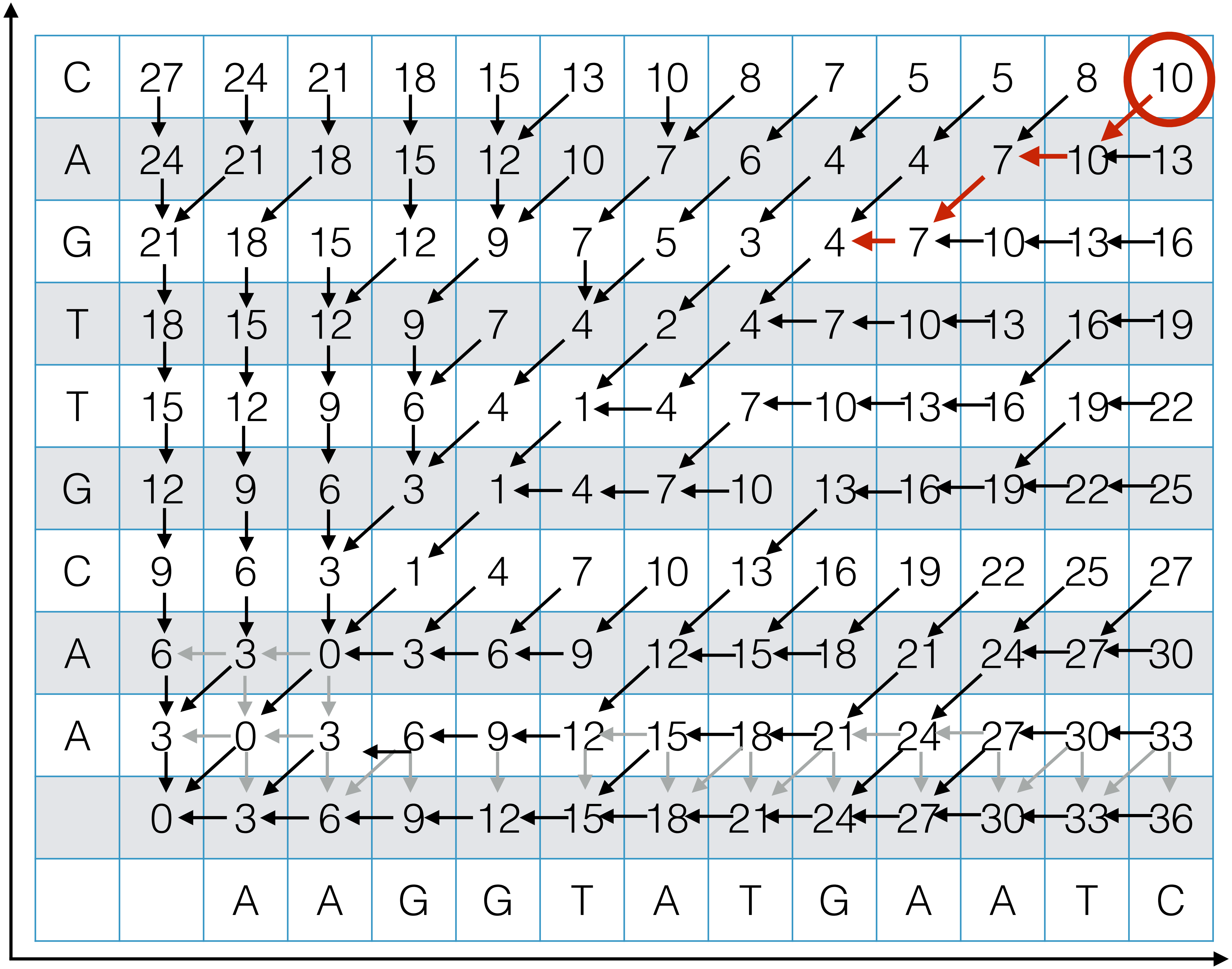
C	27	24	21	18	15	13	10	8	7	5	5	8	10
A	24	21	18	15	12	10	7	6	4	4	7	10	13
G	21	18	15	12	9	7	5	3	4	7	10	13	16
T	18	15	12	9	7	4	2	4	7	10	13	16	19
T	15	12	9	6	4	1	4	7	10	13	16	19	22
G	12	9	6	3	1	4	7	10	13	16	19	22	25
C	9	6	3	1	4	7	10	13	16	19	22	25	27
A	6	3	0	3	6	9	12	15	18	21	24	27	30
A	3	0	3	6	9	12	15	18	21	24	27	30	33
	0	3	6	9	12	15	18	21	24	27	30	33	36
		A	A	G	G	T	A	T	G	A	A	T	C

Example



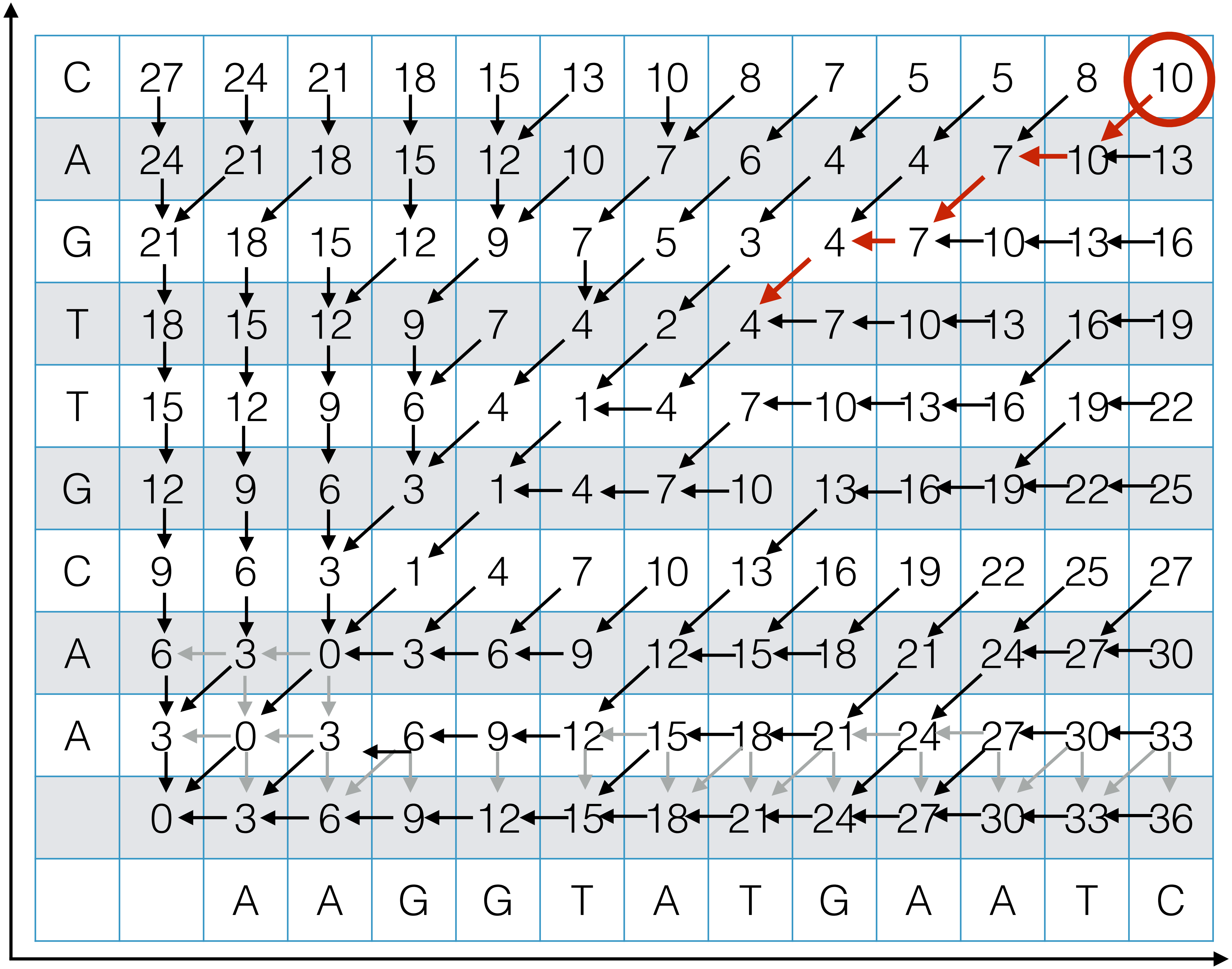
C	27	24	21	18	15	13	10	8	7	5	5	8	10
A	24	21	18	15	12	10	7	6	4	4	7	10	13
G	21	18	15	12	9	7	5	3	4	7	10	13	16
T	18	15	12	9	7	4	2	4	7	10	13	16	19
T	15	12	9	6	4	1	4	7	10	13	16	19	22
G	12	9	6	3	1	4	7	10	13	16	19	22	25
C	9	6	3	1	4	7	10	13	16	19	22	25	27
A	6	3	0	3	6	9	12	15	18	21	24	27	30
A	3	0	3	6	9	12	15	18	21	24	27	30	33
	0	3	6	9	12	15	18	21	24	27	30	33	36
		A	A	G	G	T	A	T	G	A	A	T	C

Example



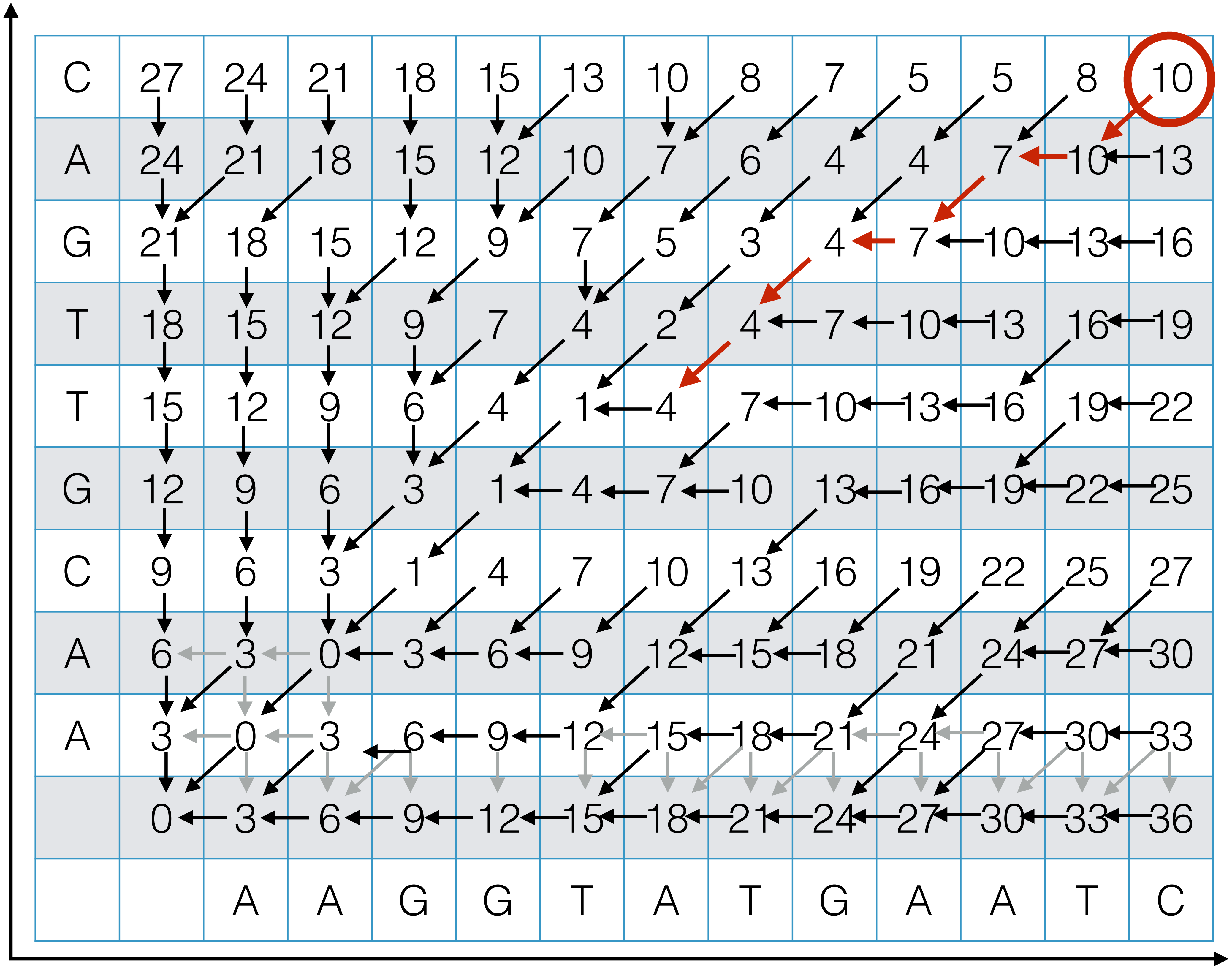
C	27	24	21	18	15	13	10	8	7	5	5	8	10
A	24	21	18	15	12	10	7	6	4	4	7	10	13
G	21	18	15	12	9	7	5	3	4	7	10	13	16
T	18	15	12	9	7	4	2	4	7	10	13	16	19
T	15	12	9	6	4	1	4	7	10	13	16	19	22
G	12	9	6	3	1	4	7	10	13	16	19	22	25
C	9	6	3	1	4	7	10	13	16	19	22	25	27
A	6	3	0	3	6	9	12	15	18	21	24	27	30
A	3	0	3	6	9	12	15	18	21	24	27	30	33
	0	3	6	9	12	15	18	21	24	27	30	33	36
		A	A	G	G	T	A	T	G	A	A	T	C

Example



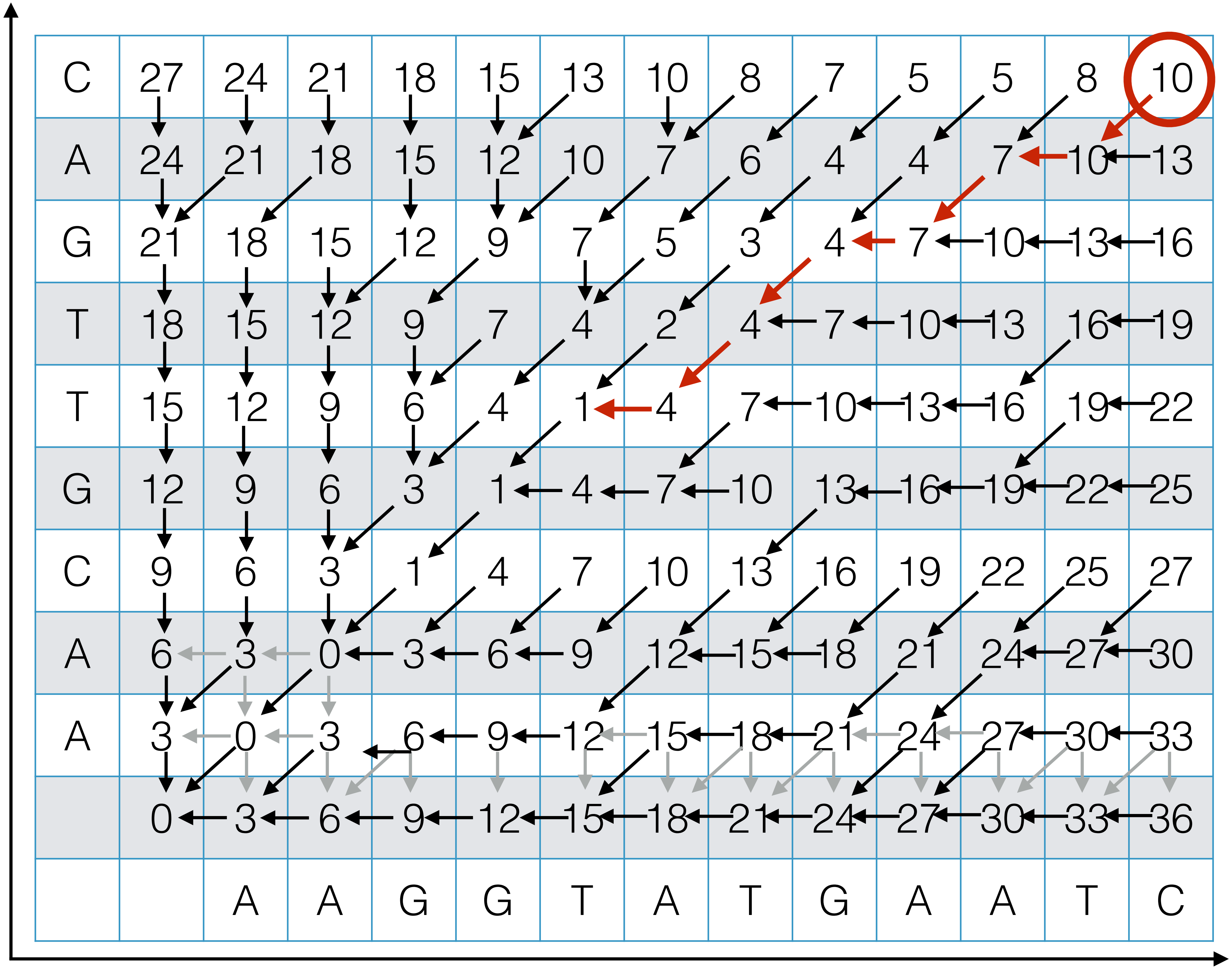
C	27	24	21	18	15	13	10	8	7	5	5	8	10
A	24	21	18	15	12	10	7	6	4	4	7	10	13
G	21	18	15	12	9	7	5	3	4	7	10	13	16
T	18	15	12	9	7	4	2	4	7	10	13	16	19
T	15	12	9	6	4	1	4	7	10	13	16	19	22
G	12	9	6	3	1	4	7	10	13	16	19	22	25
C	9	6	3	1	4	7	10	13	16	19	22	25	27
A	6	3	0	3	6	9	12	15	18	21	24	27	30
A	3	0	3	6	9	12	15	18	21	24	27	30	33
	0	3	6	9	12	15	18	21	24	27	30	33	36
		A	A	G	G	T	A	T	G	A	A	T	C

Example



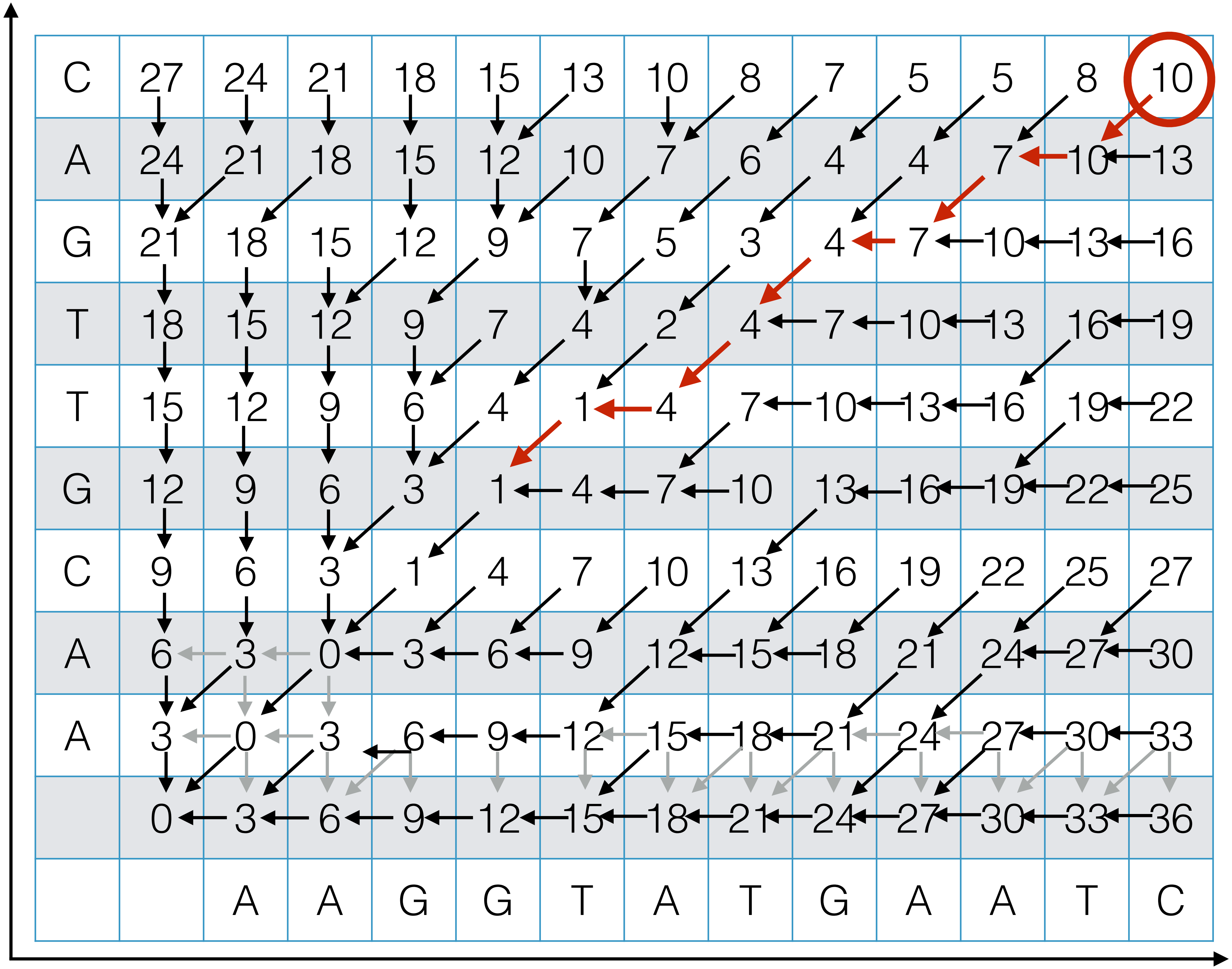
C	27	24	21	18	15	13	10	8	7	5	5	8	10
A	24	21	18	15	12	10	7	6	4	4	7	10	13
G	21	18	15	12	9	7	5	3	4	7	10	13	16
T	18	15	12	9	7	4	2	4	7	10	13	16	19
T	15	12	9	6	4	1	4	7	10	13	16	19	22
G	12	9	6	3	1	4	7	10	13	16	19	22	25
C	9	6	3	1	4	7	10	13	16	19	22	25	27
A	6	3	0	3	6	9	12	15	18	21	24	27	30
A	3	0	3	6	9	12	15	18	21	24	27	30	33
	0	3	6	9	12	15	18	21	24	27	30	33	36
		A	A	G	G	T	A	T	G	A	A	T	C

Example



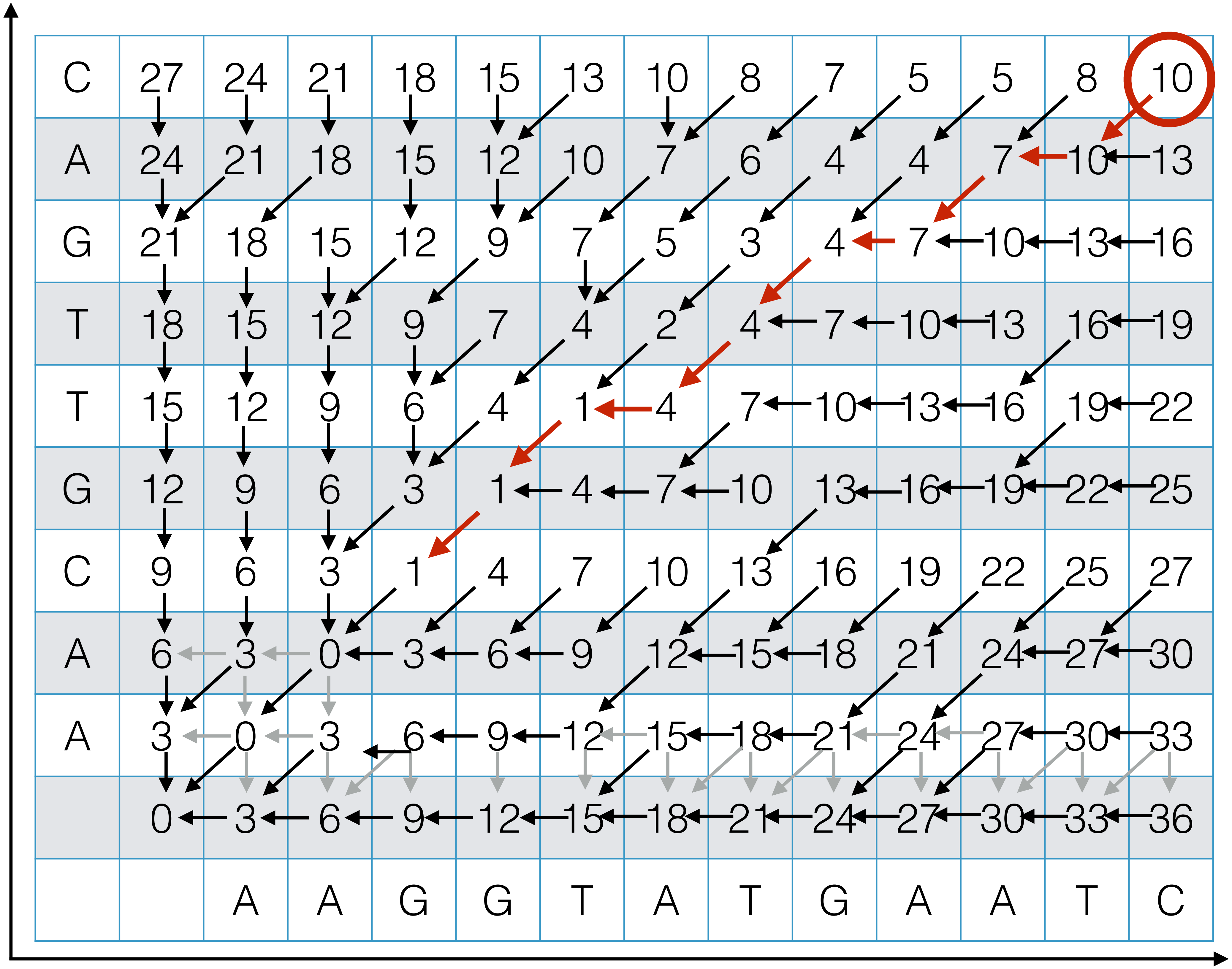
C	27	24	21	18	15	13	10	8	7	5	5	8	10
A	24	21	18	15	12	10	7	6	4	4	7	10	13
G	21	18	15	12	9	7	5	3	4	7	10	13	16
T	18	15	12	9	7	4	2	4	7	10	13	16	19
T	15	12	9	6	4	1	4	7	10	13	16	19	22
G	12	9	6	3	1	4	7	10	13	16	19	22	25
C	9	6	3	1	4	7	10	13	16	19	22	25	27
A	6	3	0	3	6	9	12	15	18	21	24	27	30
A	3	0	3	6	9	12	15	18	21	24	27	30	33
	0	3	6	9	12	15	18	21	24	27	30	33	36
		A	A	G	G	T	A	T	G	A	A	T	C

Example



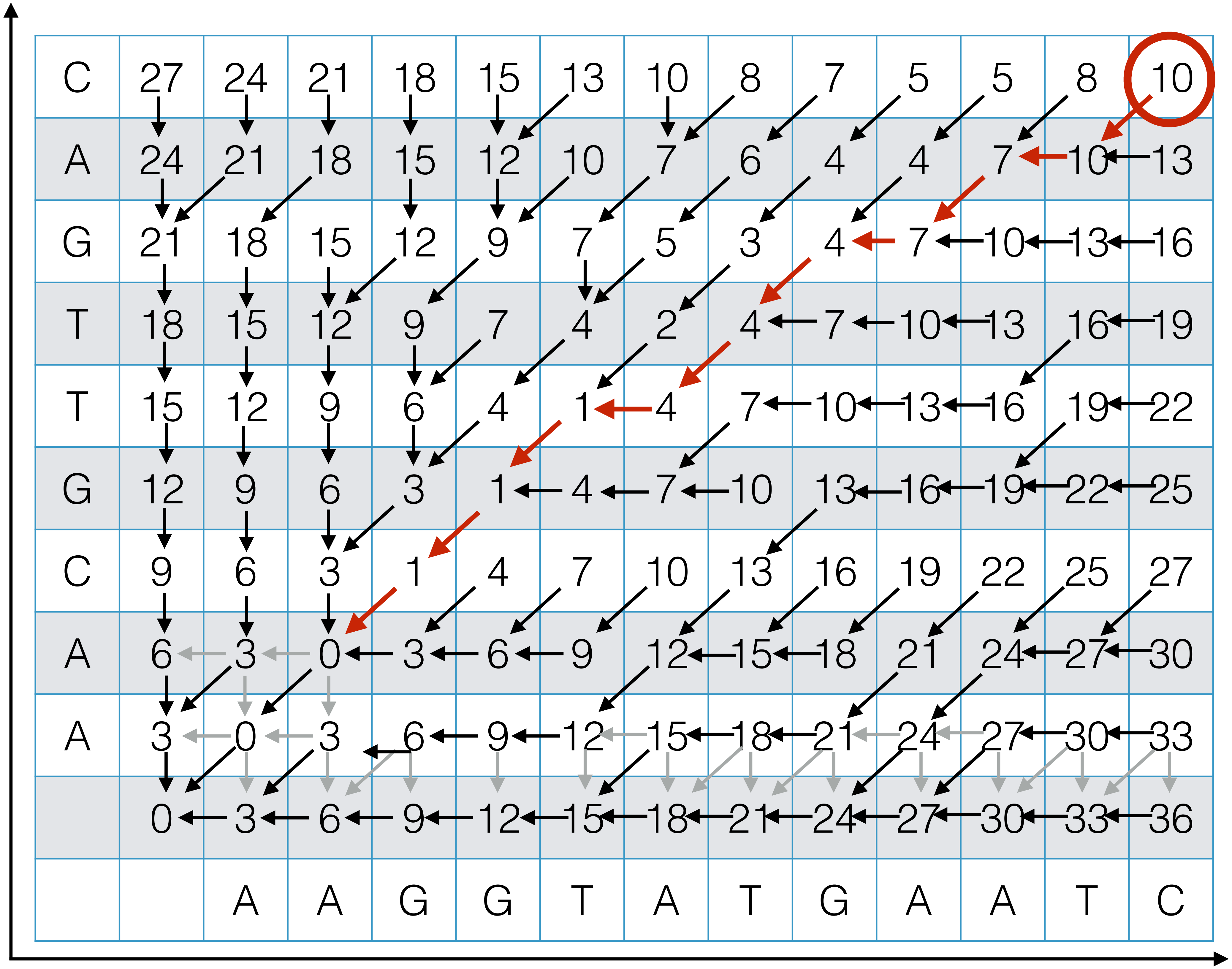
C	27	24	21	18	15	13	10	8	7	5	5	8	10
A	24	21	18	15	12	10	7	6	4	4	7	10	13
G	21	18	15	12	9	7	5	3	4	7	10	13	16
T	18	15	12	9	7	4	2	4	7	10	13	16	19
T	15	12	9	6	4	1	4	7	10	13	16	19	22
G	12	9	6	3	1	4	7	10	13	16	19	22	25
C	9	6	3	1	4	7	10	13	16	19	22	25	27
A	6	3	0	3	6	9	12	15	18	21	24	27	30
A	3	0	3	6	9	12	15	18	21	24	27	30	33
	0	3	6	9	12	15	18	21	24	27	30	33	36
		A	A	G	G	T	A	T	G	A	A	T	C

Example



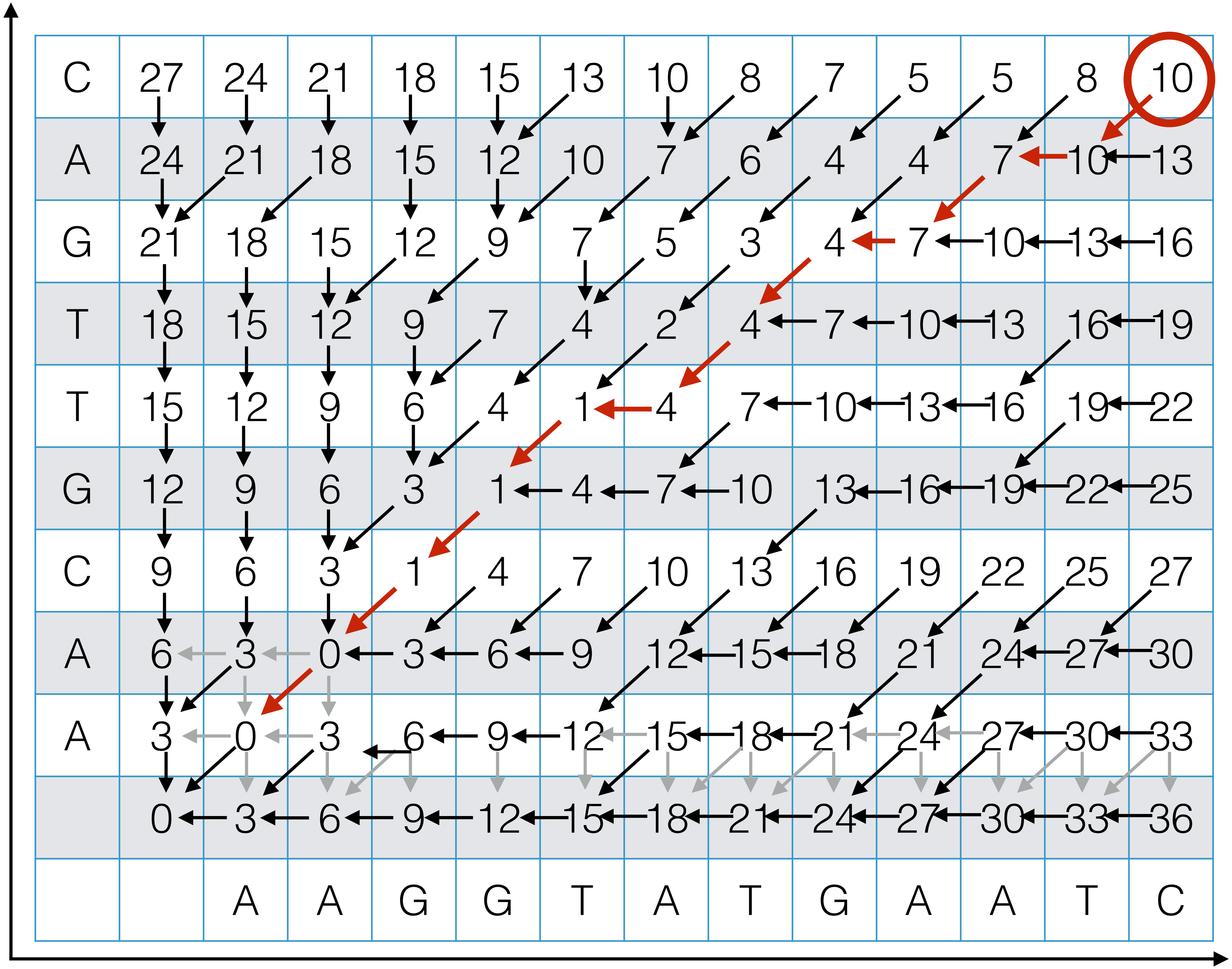
C	27	24	21	18	15	13	10	8	7	5	5	8	10
A	24	21	18	15	12	10	7	6	4	4	7	10	13
G	21	18	15	12	9	7	5	3	4	7	10	13	16
T	18	15	12	9	7	4	2	4	7	10	13	16	19
T	15	12	9	6	4	1	4	7	10	13	16	19	22
G	12	9	6	3	1	4	7	10	13	16	19	22	25
C	9	6	3	1	4	7	10	13	16	19	22	25	27
A	6	3	0	3	6	9	12	15	18	21	24	27	30
A	3	0	3	6	9	12	15	18	21	24	27	30	33
	0	3	6	9	12	15	18	21	24	27	30	33	36
		A	A	G	G	T	A	T	G	A	A	T	C

Example



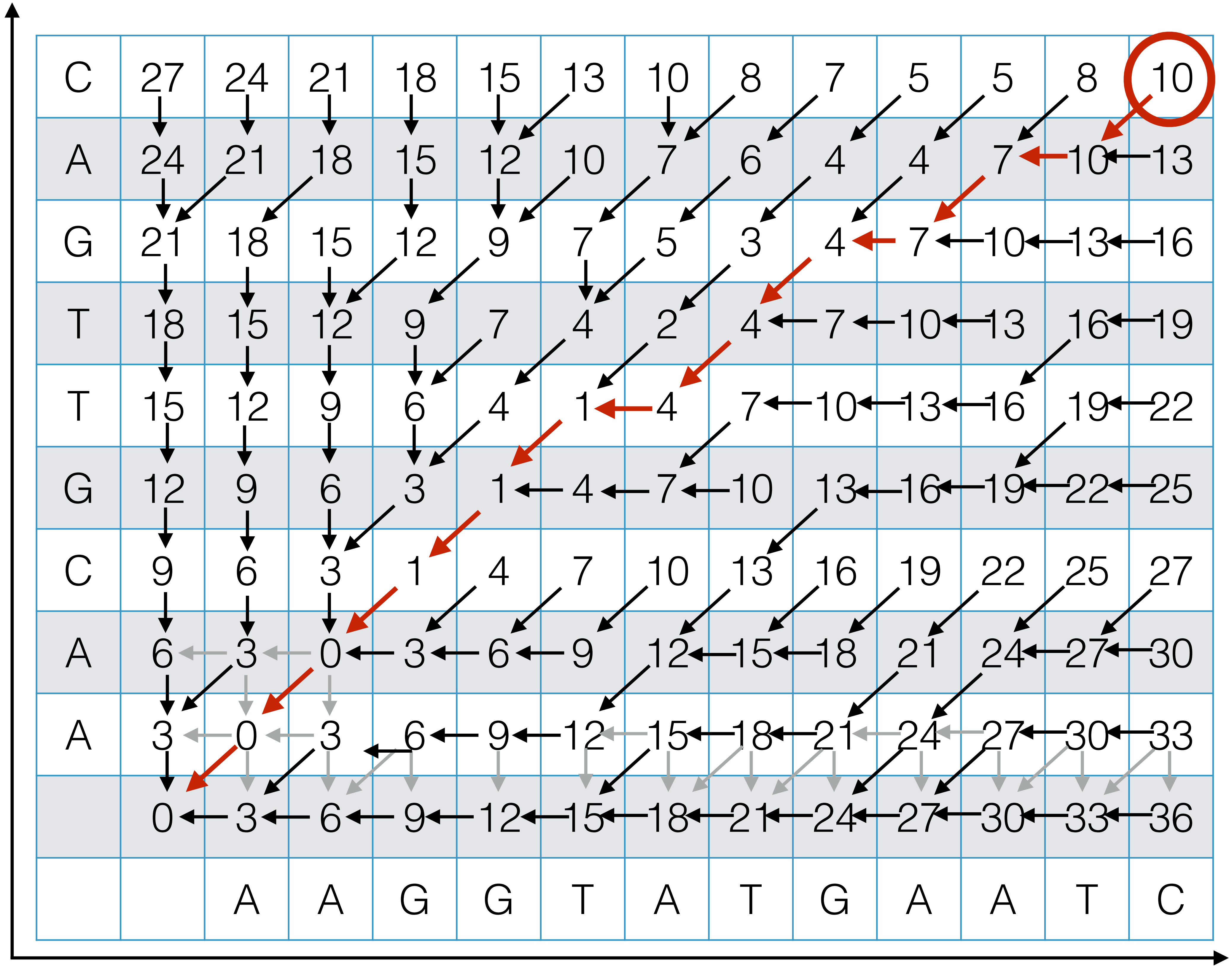
C	27	24	21	18	15	13	10	8	7	5	5	8	10
A	24	21	18	15	12	10	7	6	4	4	7	10	13
G	21	18	15	12	9	7	5	3	4	7	10	13	16
T	18	15	12	9	7	4	2	4	7	10	13	16	19
T	15	12	9	6	4	1	4	7	10	13	16	19	22
G	12	9	6	3	1	4	7	10	13	16	19	22	25
C	9	6	3	1	4	7	10	13	16	19	22	25	27
A	6	3	0	3	6	9	12	15	18	21	24	27	30
A	3	0	3	6	9	12	15	18	21	24	27	30	33
	0	3	6	9	12	15	18	21	24	27	30	33	36
		A	A	G	G	T	A	T	G	A	A	T	C

Example



C	27	24	21	18	15	13	10	8	7	5	5	8	10
A	24	21	18	15	12	10	7	6	4	4	7	10	13
G	21	18	15	12	9	7	5	3	4	7	10	13	16
T	18	15	12	9	7	4	2	4	7	10	13	16	19
T	15	12	9	6	4	1	4	7	10	13	16	19	22
G	12	9	6	3	1	4	7	10	13	16	19	22	25
C	9	6	3	1	4	7	10	13	16	19	22	25	27
A	6	3	0	3	6	9	12	15	18	21	24	27	30
A	3	0	3	6	9	12	15	18	21	24	27	30	33
	0	3	6	9	12	15	18	21	24	27	30	33	36
		A	A	G	G	T	A	T	G	A	A	T	C

Example



C	27	24	21	18	15	13	10	8	7	5	5	8	10
A	24	21	18	15	12	10	7	6	4	4	7	10	13
G	21	18	15	12	9	7	5	3	4	7	10	13	16
T	18	15	12	9	7	4	2	4	7	10	13	16	19
T	15	12	9	6	4	1	4	7	10	13	16	19	22
G	12	9	6	3	1	4	7	10	13	16	19	22	25
C	9	6	3	1	4	7	10	13	16	19	22	25	27
A	6	3	0	3	6	9	12	15	18	21	24	27	30
A	3	0	3	6	9	12	15	18	21	24	27	30	33
	0	3	6	9	12	15	18	21	24	27	30	33	36
		A	A	G	G	T	A	T	G	A	A	T	C

Recap: Dynamic Programming

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

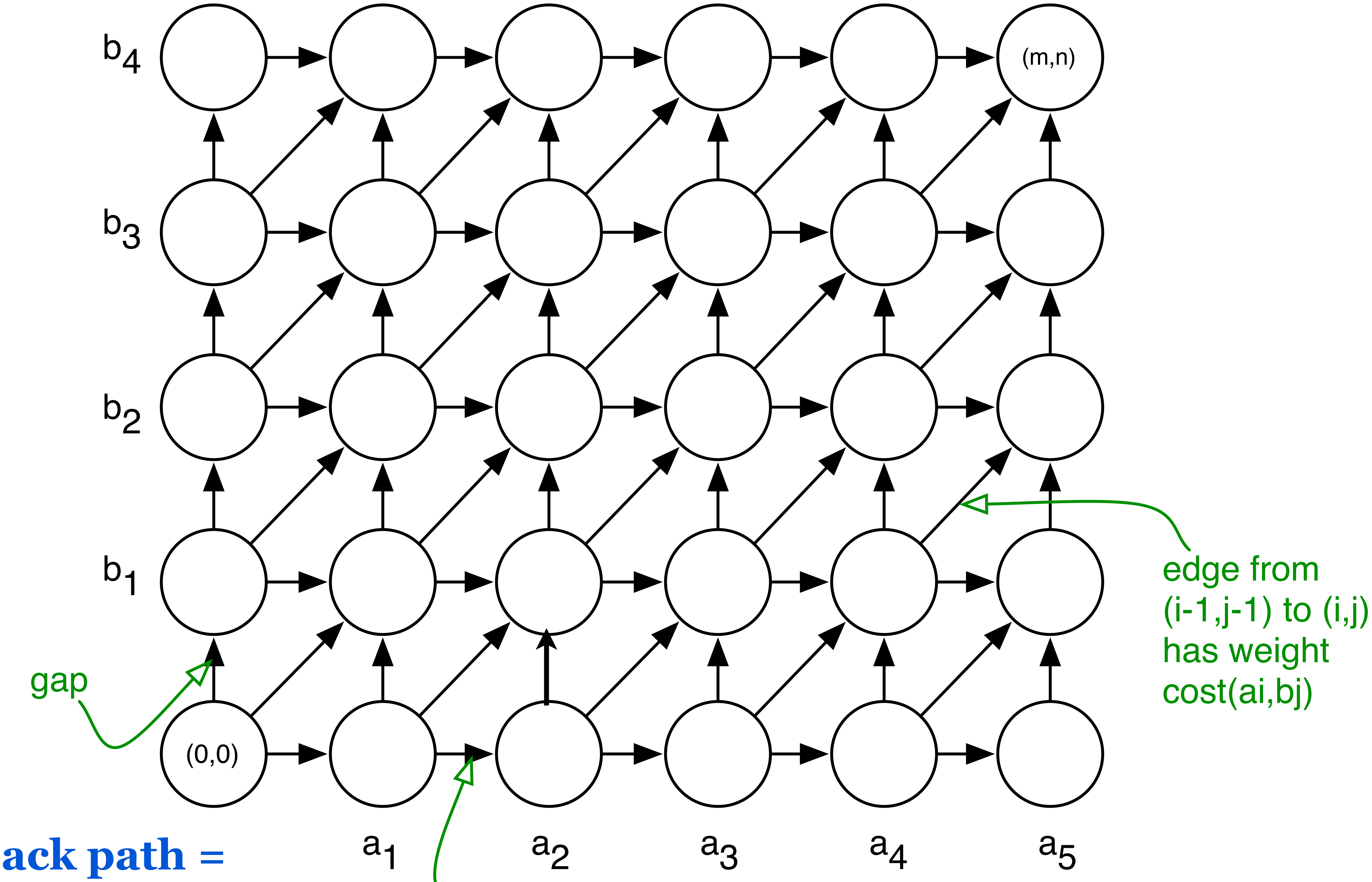
Main idea of dynamic programming: solve the subproblems in an order so that when you need an answer, it's ready.

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.

*

Another View: Recasting as a Graph



gap

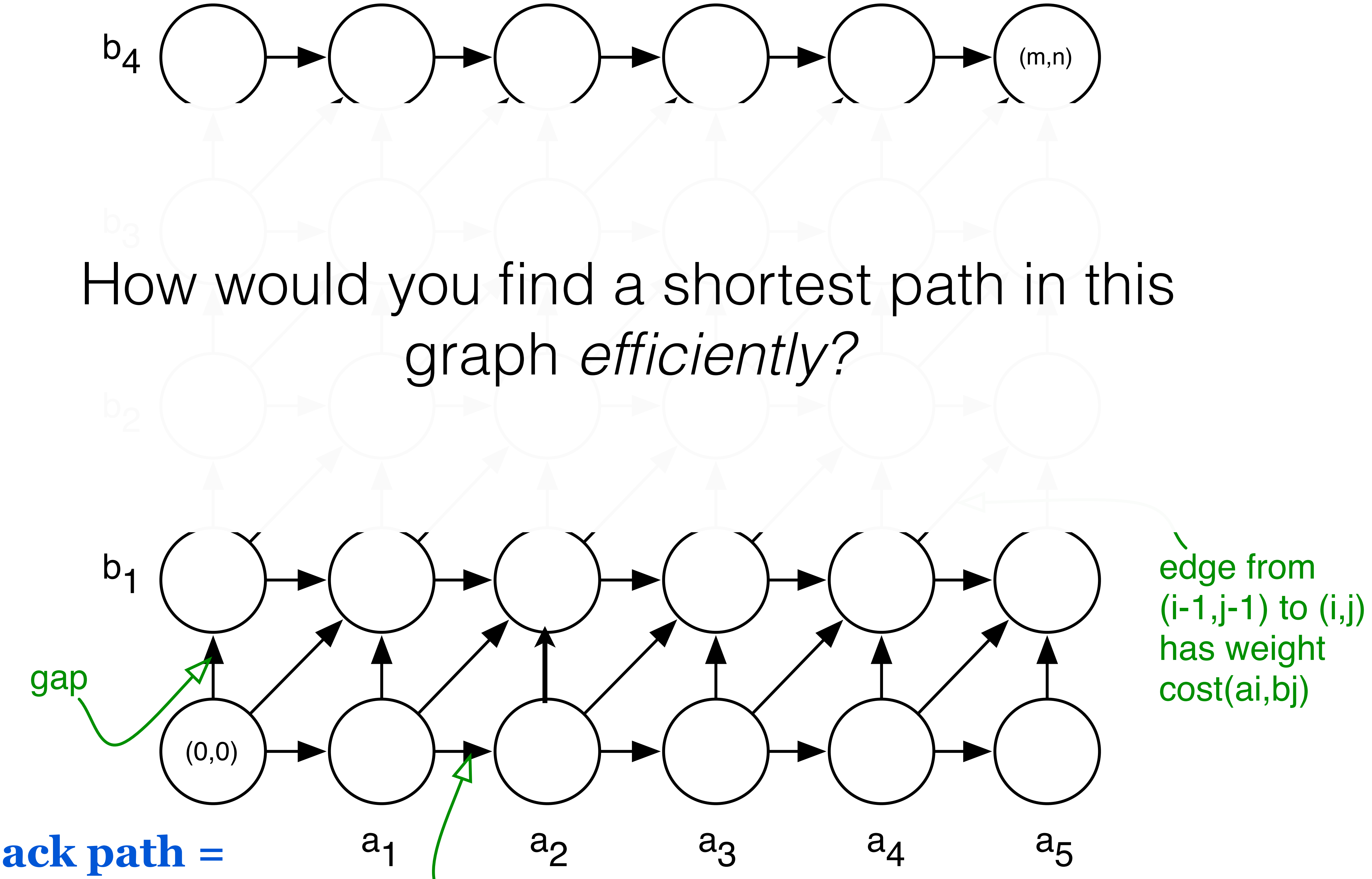
edge from (i-1,j-1) to (i,j) has weight cost(ai,bj)

gap

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

*

Another View: Recasting as a Graph



How would you find a shortest path in this graph *efficiently*?

gap

edge from (i-1,j-1) to (i,j) has weight cost(ai,bj)

gap

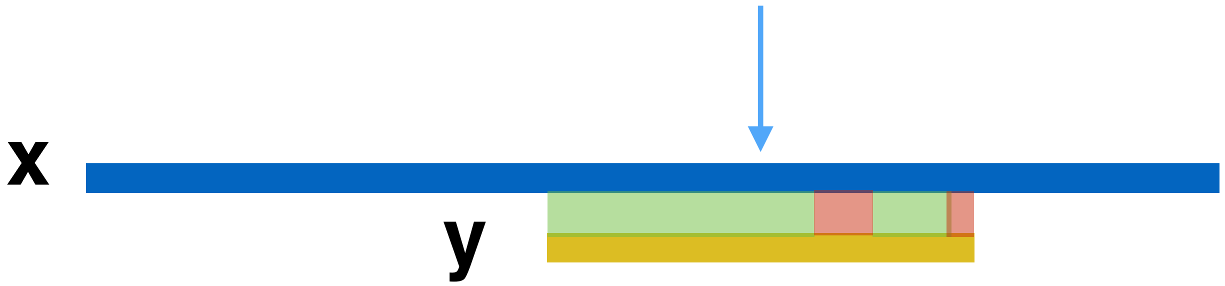
Traceback path = shortest path from (0,0) to (m,n)

*

Semi-global Alignment Example

Semi-global (glocal): Gaps at the beginning or end of **x** or **y** are free — one useful case is when one string is significantly shorter than the other

sometimes called "cost-free-ends" or "fitting" alignment



We'll discuss the "fitting" variant for in the next few slides for simplicity, but the same basic idea applies for the "overlap" variant as well.

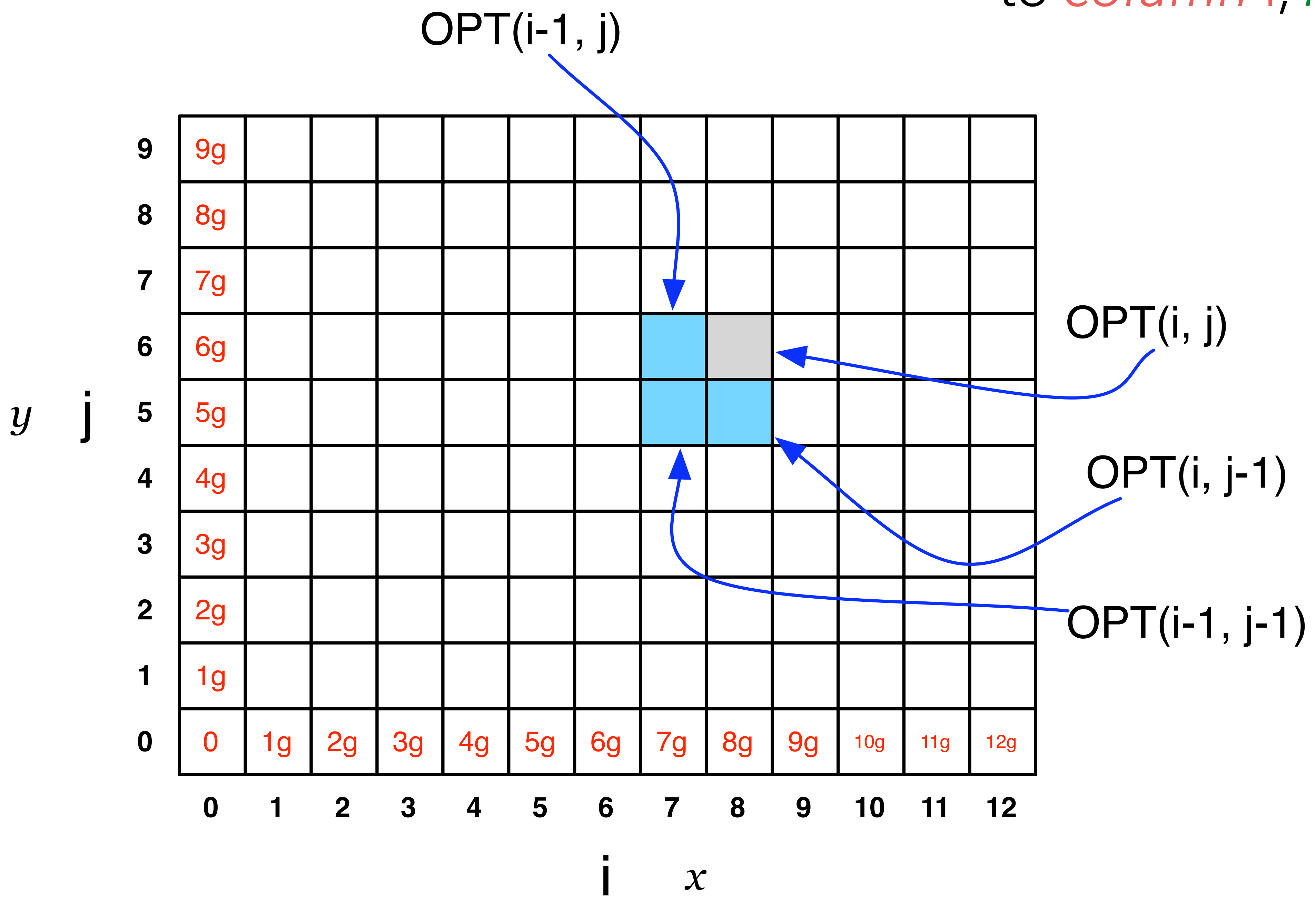
Recall: Global Alignment Matrix

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*]

NOTE: observe the non-standard notation here; OPT(**i**,**j**) is referring to *column* i, *row* j of the matrix.



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

*

How to do semi-global alignment?

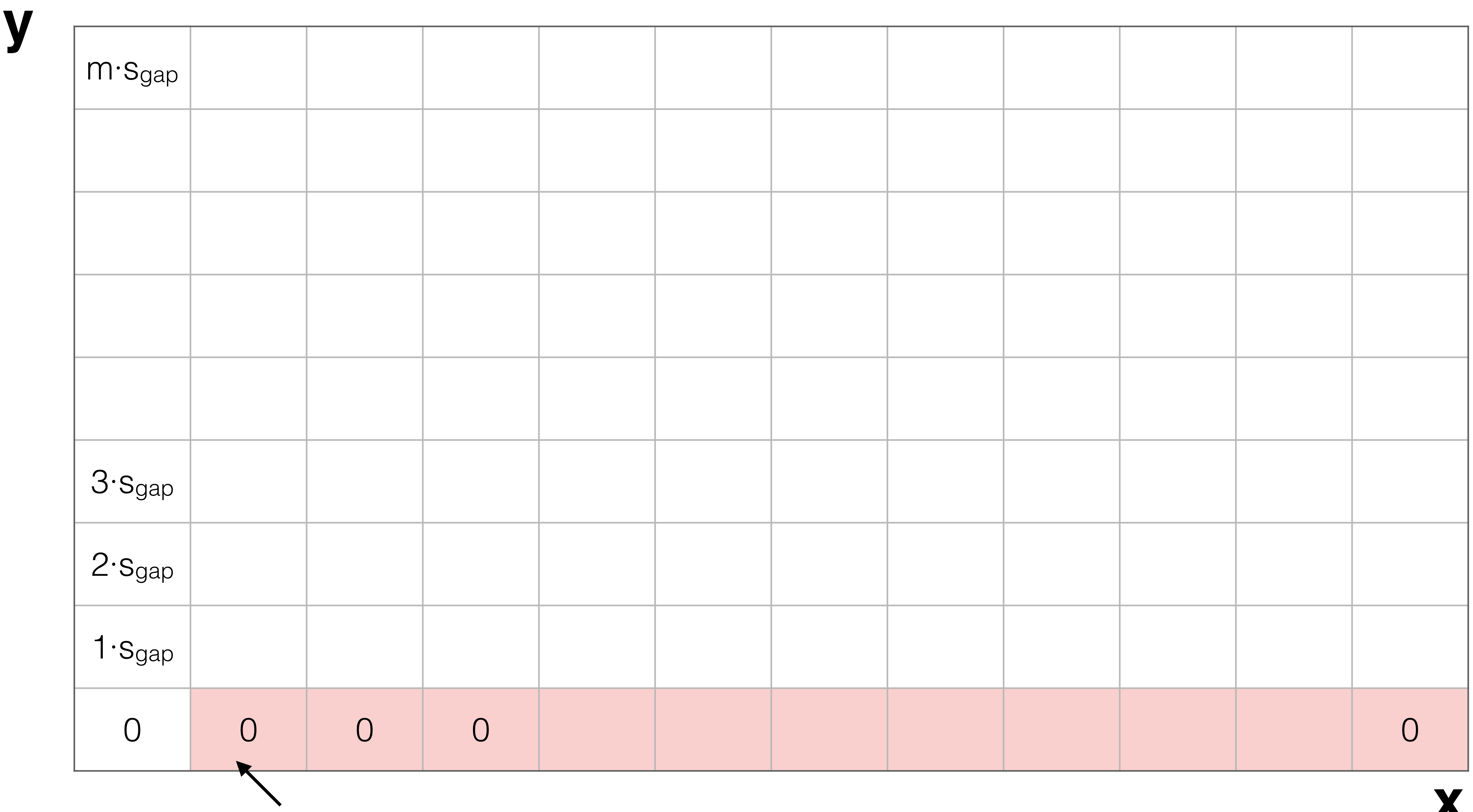
y

$m \cdot s_{gap}$											
$3 \cdot s_{gap}$											
$2 \cdot s_{gap}$											
$1 \cdot s_{gap}$											
0	$1 \cdot s_{gap}$	$2 \cdot s_{gap}$	$3 \cdot s_{gap}$								$n \cdot s_{gap}$

x

Start with the original global alignment matrix

How to do semi-global alignment?



$m \cdot s_{gap}$											
$3 \cdot s_{gap}$											
$2 \cdot s_{gap}$											
$1 \cdot s_{gap}$											
0	0	0	0								0

change the base case — allow gaps *before* y

How to do semi-global alignment?

y

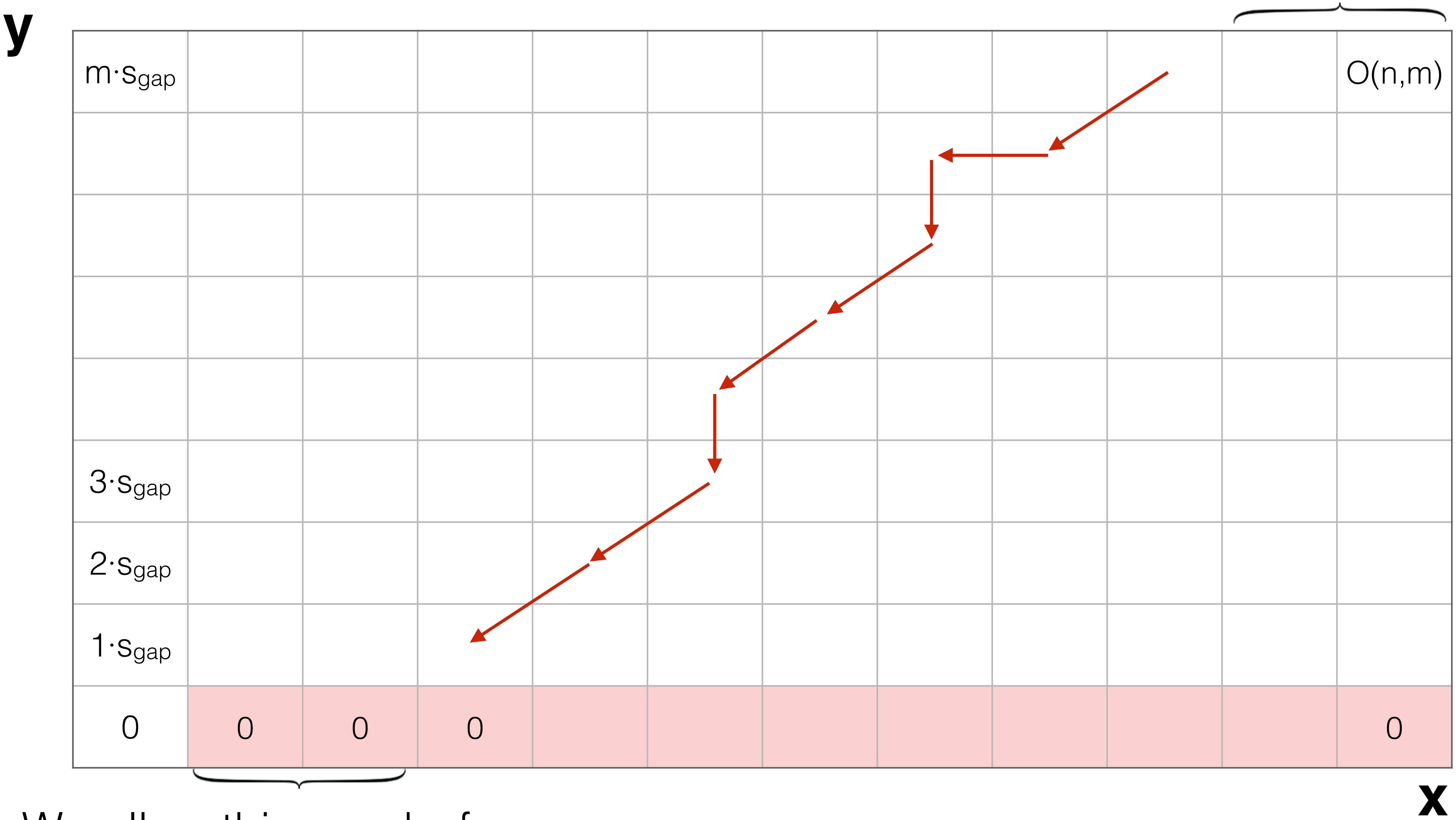
$m \cdot s_{gap}$											O(n,m)
$3 \cdot s_{gap}$											
$2 \cdot s_{gap}$											
$1 \cdot s_{gap}$											
0	0	0	0								0

x

start traceback at $\max_{0<i\leq n}$ OPT(i,m) — this allows gaps after **y**; why?

Semi-global alignment example

and this gap after **y**



We allow this gap before **y**

Semi-global Alignment

What is the same and different between the “global” and semi-global (“fitting”) alignment problems?

*assuming |y| < |x| and we are “fitting” y into x

Global	Semi-global (“fitting”)
$\text{OPT}(i,j) = \max \begin{cases} \text{score}(x_i, y_j) + \text{OPT}(i-1, j-1) \\ \text{s}_{\text{gap}} + \text{OPT}(i-1, j) \\ \text{s}_{\text{gap}} + \text{OPT}(i, j-1) \end{cases}$	$\text{OPT}(i,j) = \max \begin{cases} \text{score}(x_i, y_j) + \text{OPT}(i-1, j-1) \\ \text{s}_{\text{gap}} + \text{OPT}(i-1, j) \\ \text{s}_{\text{gap}} + \text{OPT}(i, j-1) \end{cases}$
Base case: OPT(i,0) = i x s_{gap}	Base case: OPT(i,0) = 0
Traceback starts at OPT(n,m)	Traceback starts at $\underset{0<j\leq n}{\textbf{max}}$ OPT(j,m)

Semi-global Alignment

The recurrence remains the *same*, we only change the base case of the recurrence and the origin of the backtrack

1) Ignore gaps before x → change base case;
$OPT(0,j) = 0$

2) Ignore gaps after x → change traceback;
start from $\max_{0<j\leq m} OPT(n,j)$

3) Ignore gaps before y → change base case;
$OPT(i,0) = 0$

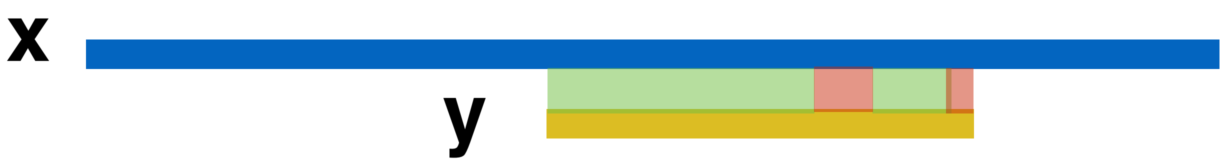
4) Ignore gaps after y → change traceback;
start from $\max_{0<i\leq n} OPT(i,m)$

Semi-global Alignment

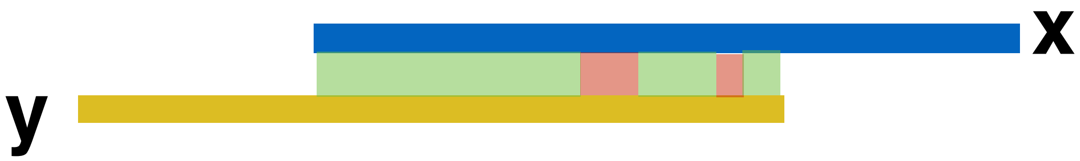
1) Ignore gaps before x
2) Ignore gaps after x
3) Ignore gaps before y
4) Ignore gaps after y

Types of semi-global alignments

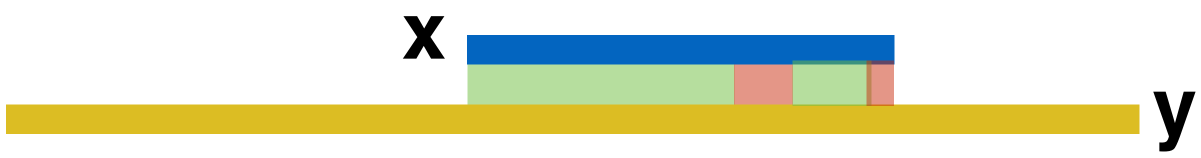
use mods 3&4



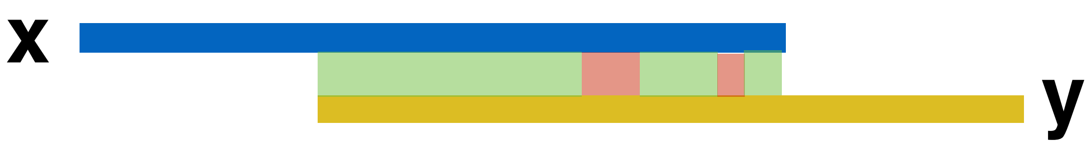
use mods 1&4



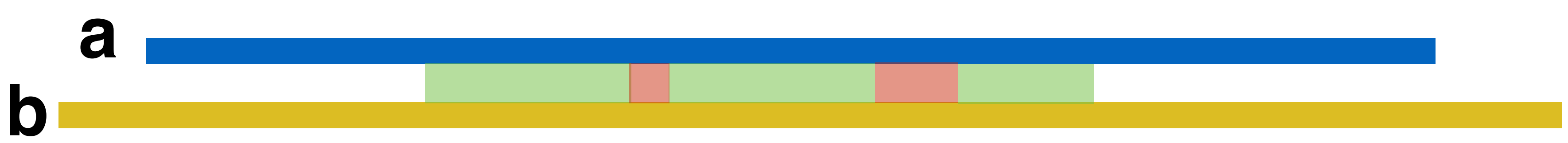
use mods 1&2



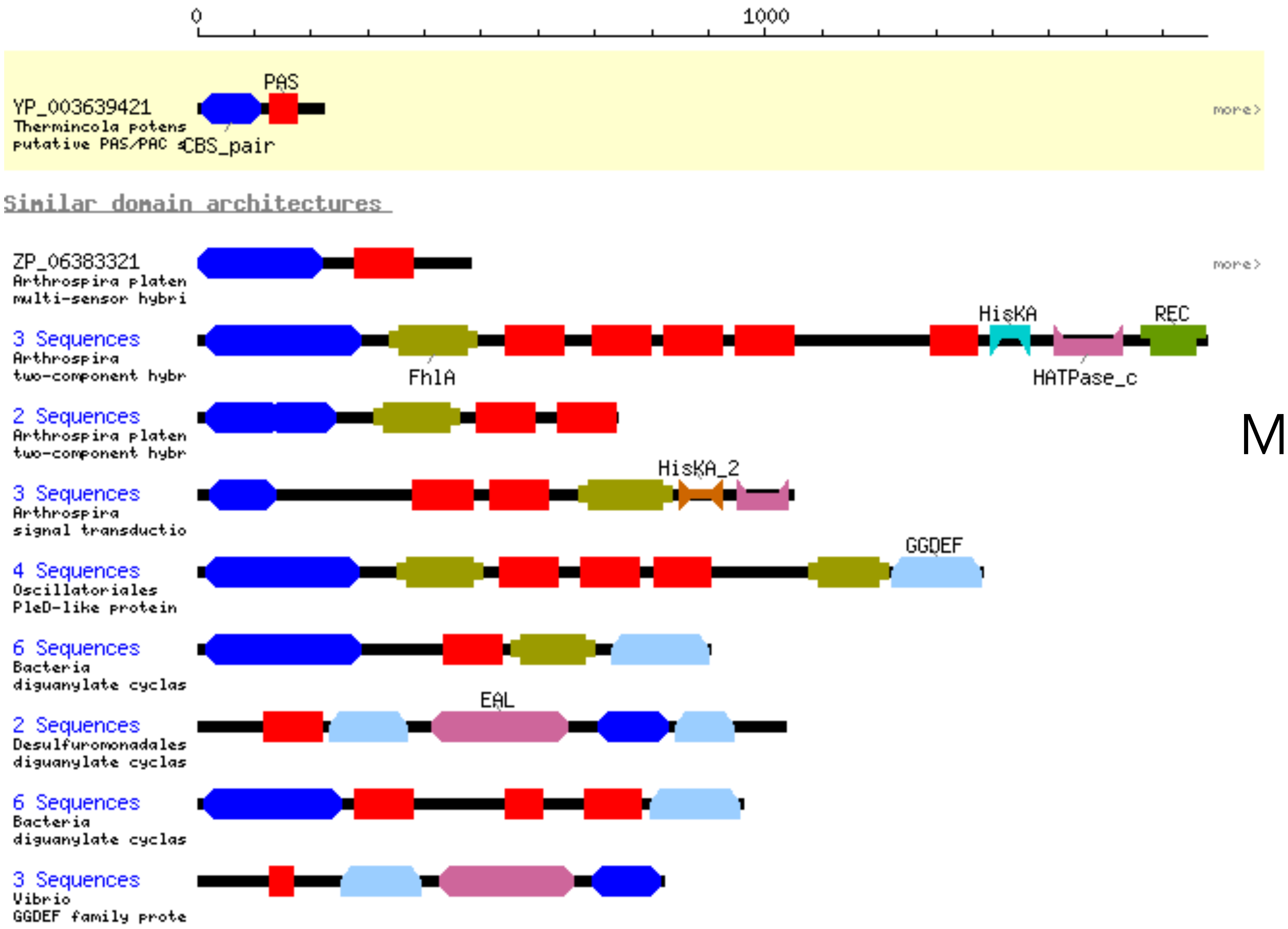
use mods 2&3



Local Alignment



Local alignment between a and b: Best alignment between a subsequence of a and **a** subsequence of **b**.



Motivation:

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

*

Local Alignment

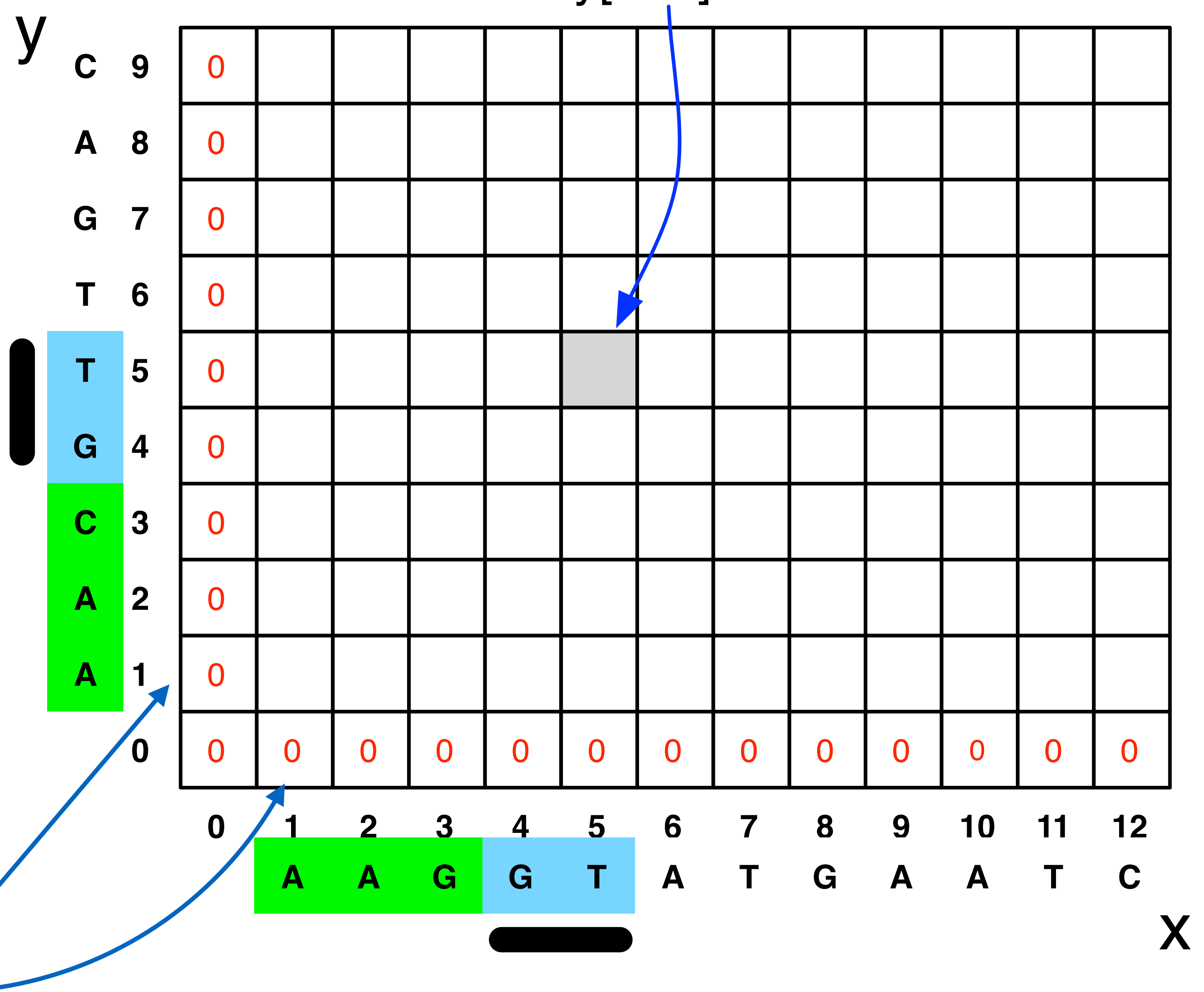
New meaning of entry of matrix entry:

OPT(i, j) = best score between:

some suffix of *x*[1…*i*]

some suffix of *y*[1…*j*]

Best alignment between a suffix of x[1..5] and a suffix of y[1..5]



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

Same base-case trick we used in semi-global alignment

*

Local Alignment

New meaning of entry of matrix entry:

OPT(i, j) = best score between:

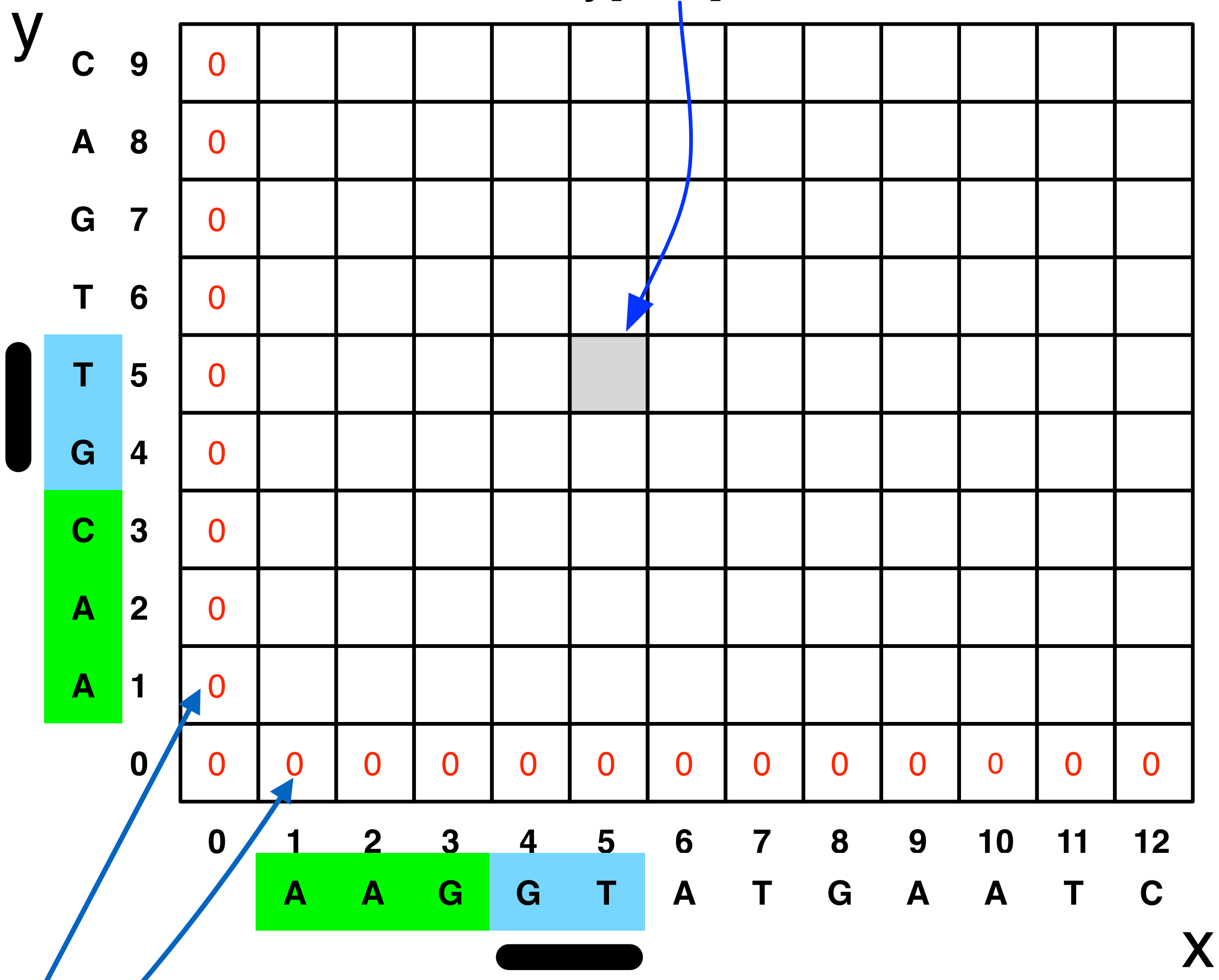
some suffix of *x*[1…*i*]

some suffix of *y*[1…*j*]

What else do we need to change to allow local alignments?

Hint: The empty alignment is always a valid local alignment!

Best alignment between a suffix of x[1..5] and a suffix of y[1..5]



Same base-case trick we used in semi-global alignment

*

How do we fill in the local alignment matrix?

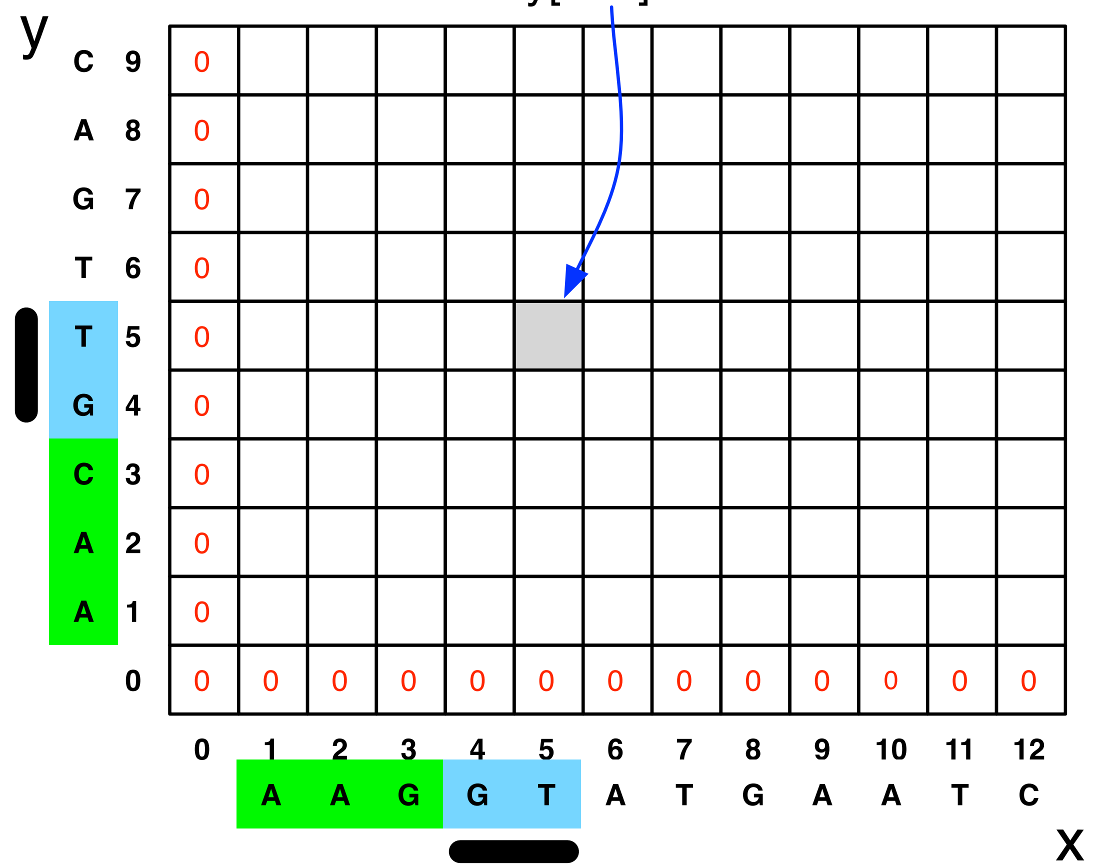
$$\mathrm{OPT}(i,j) = \max \begin{cases} \mathrm{score}(x_i, y_j) + \mathrm{OPT}(i-1, j-1) & (1) \\ s_{\mathrm{gap}} + \mathrm{OPT}(i-1, j) & (2) \\ s_{\mathrm{gap}} + \mathrm{OPT}(i, j-1) & (3) \\ 0 \end{cases}$$

(1), (2), and (3): same cases as before: match x and y, gap in y, gap in x

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"

Best alignment between a suffix of x[1..5] and a suffix of y[1..5]



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

x

*

Local Alignment

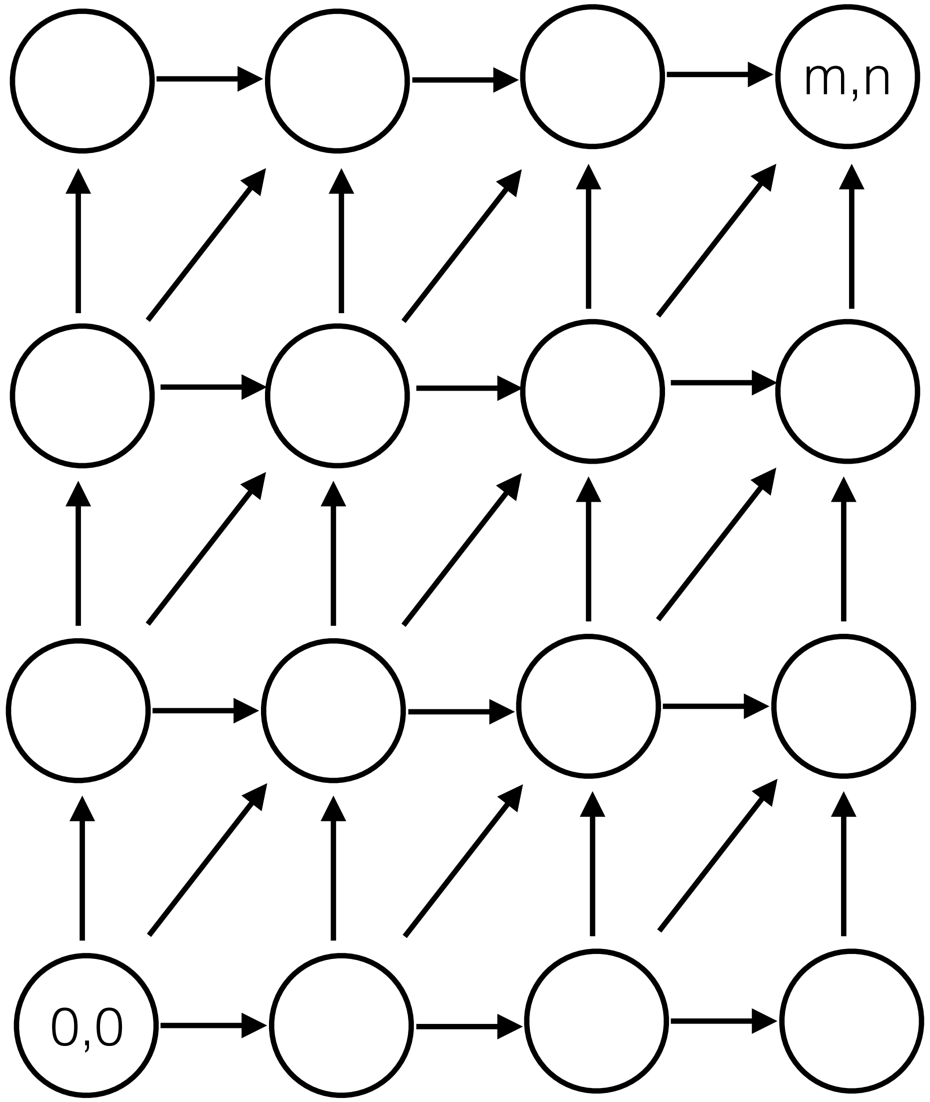
- Initialize first row and first column to be 0.

- 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

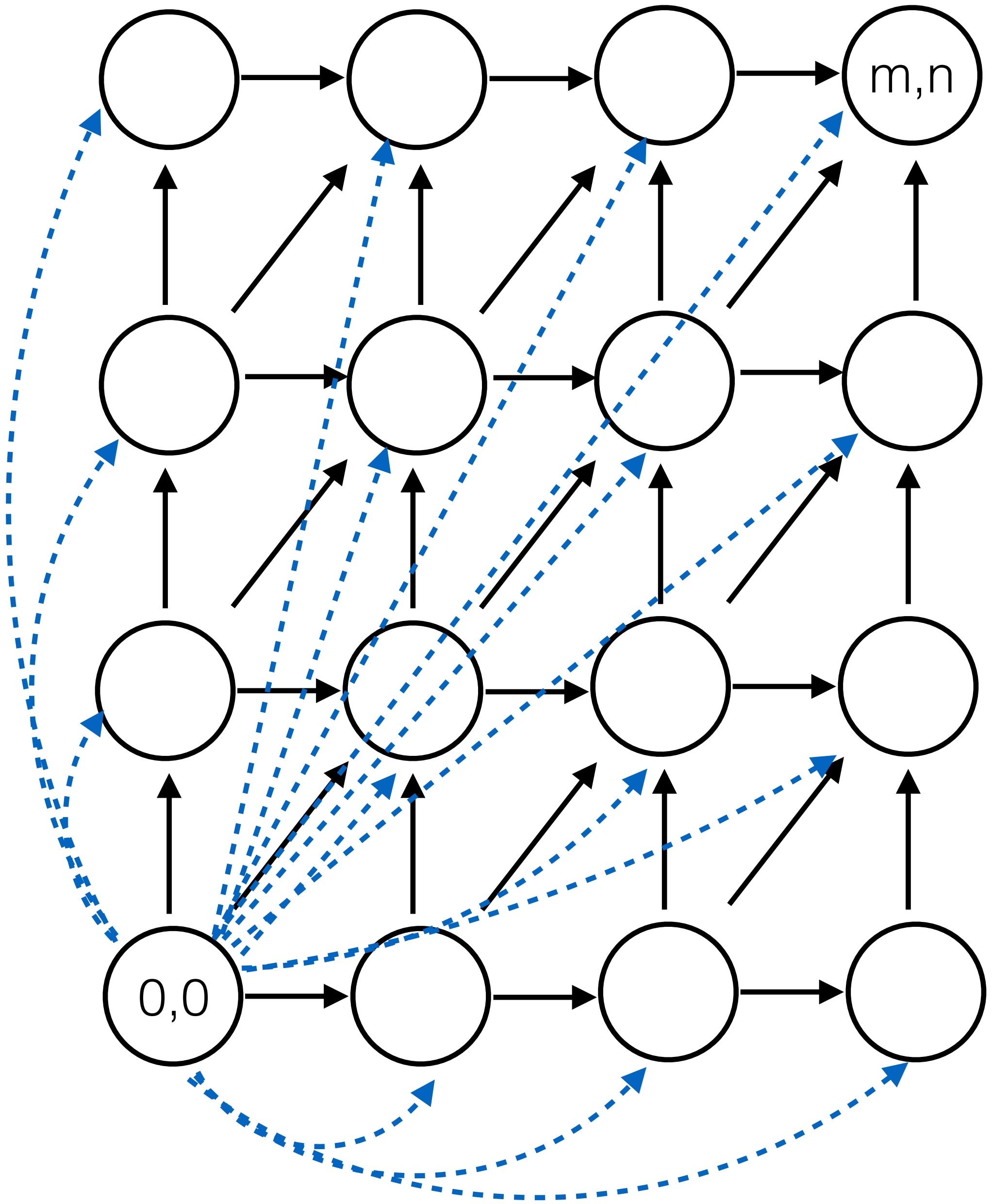
*

Local Alignment in the DAG framework



Local Alignment in the DAG framework

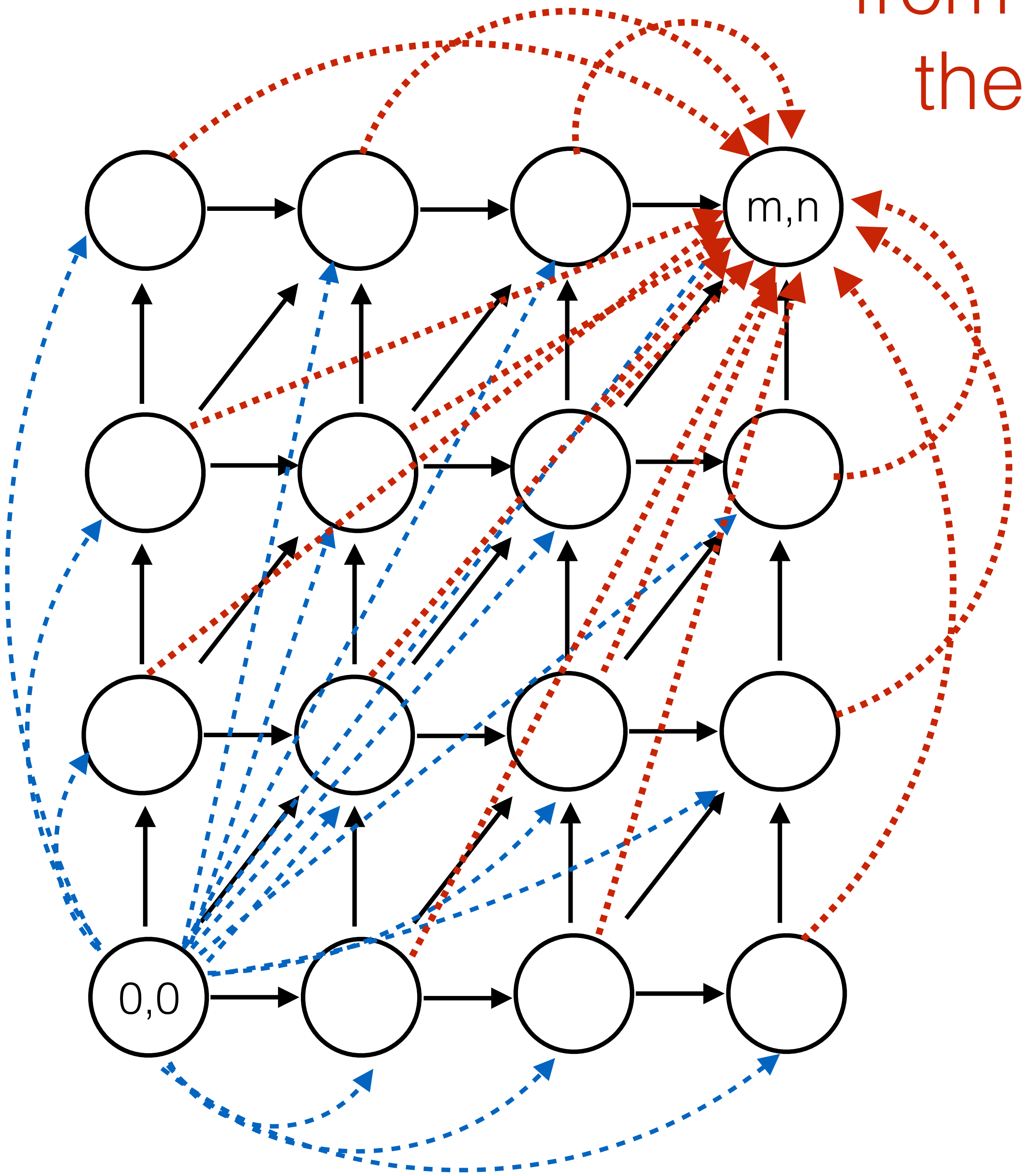
Add 0 score edge from the source to every node



Local Alignment in the DAG framework

Add 0 score edge from the source to every vertex

Add 0 score edge from every vertex to the target vertex



Side Note: Lower Bounds

- Suppose the lengths of *x* and *y* are *n*.
- Clearly, need at least Ω(n) time to find their global alignment (have to read the strings!)
- The DP algorithms show global alignment can be done in $O(n^2)$ time.
- A trick called the "Four Russians Speedup" can make a similar dynamic programming algorithm run in $O(n^2 / \log n)$ time.
  - We probably won't talk about the Four Russians Speedup.
  - The important thing to remember is that only one of the four authors is Russian...

(Alrazarov, Dinic, Kronrod, Faradzev, 1970)

- Open questions: Can we do better? Can we prove that we can't do better? No#

#: Backurs, Arturs, and Piotr Indyk. "Edit distance cannot be computed in strongly subquadratic time (unless SETH is false)." *Proceedings of the forty-seventh annual ACM symposium on Theory of computing*. ACM, 2015.

*

Using semi-global alignment is not feasible for read mapping

The best algorithms we have (and like the best that could exist) to compute the optimal alignment of two strings are *quadratic*

If we have N reads, each of length ℓ, and the genome is of length L, then applying the optimal algorithm at each possible position (to test the edit distance) is $\mathbf{O(N \cdot \ell \cdot L)}$

Consider a dataset with:

N = 20 x 10^6 reads

ℓ = 100

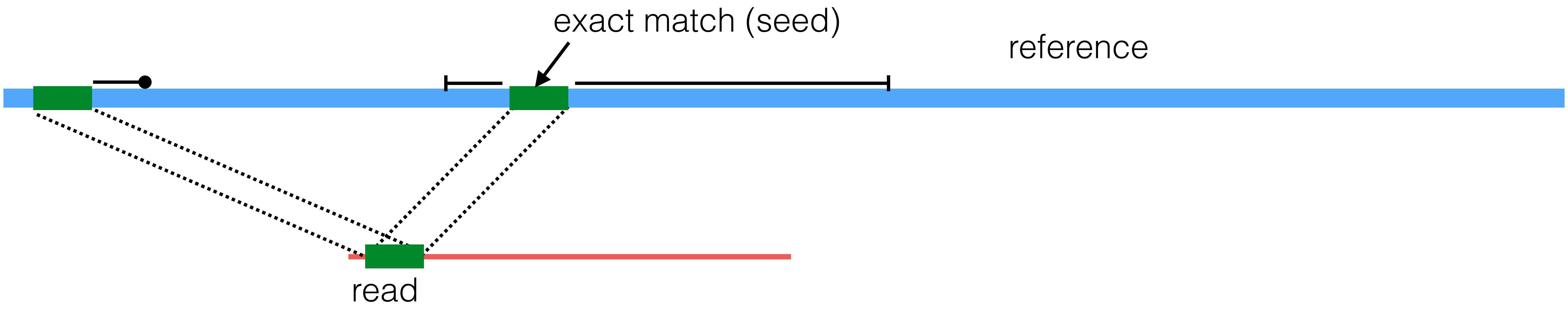
L = 3x10^9 nucleotides

and a processor that can do X = 3 x 10^9 operations / sec.

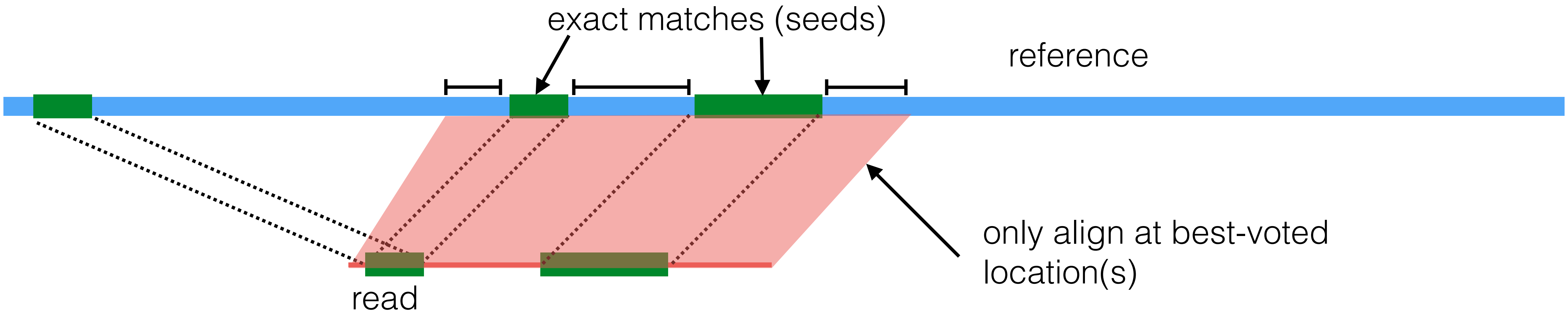
You'd wait about $(N \cdot \ell \cdot L) / X$ = 200,000,000 sec = 6.34 **years** to align your reads.

How can exact matching help?

Seed & Extend:



Seed & Vote:



How can exact matching help?

GGTACCCCCAAATTCGAAAGAGCCGAATGAAGTGGAAAGAATGGCCATTCAAATGGCTTGGGA

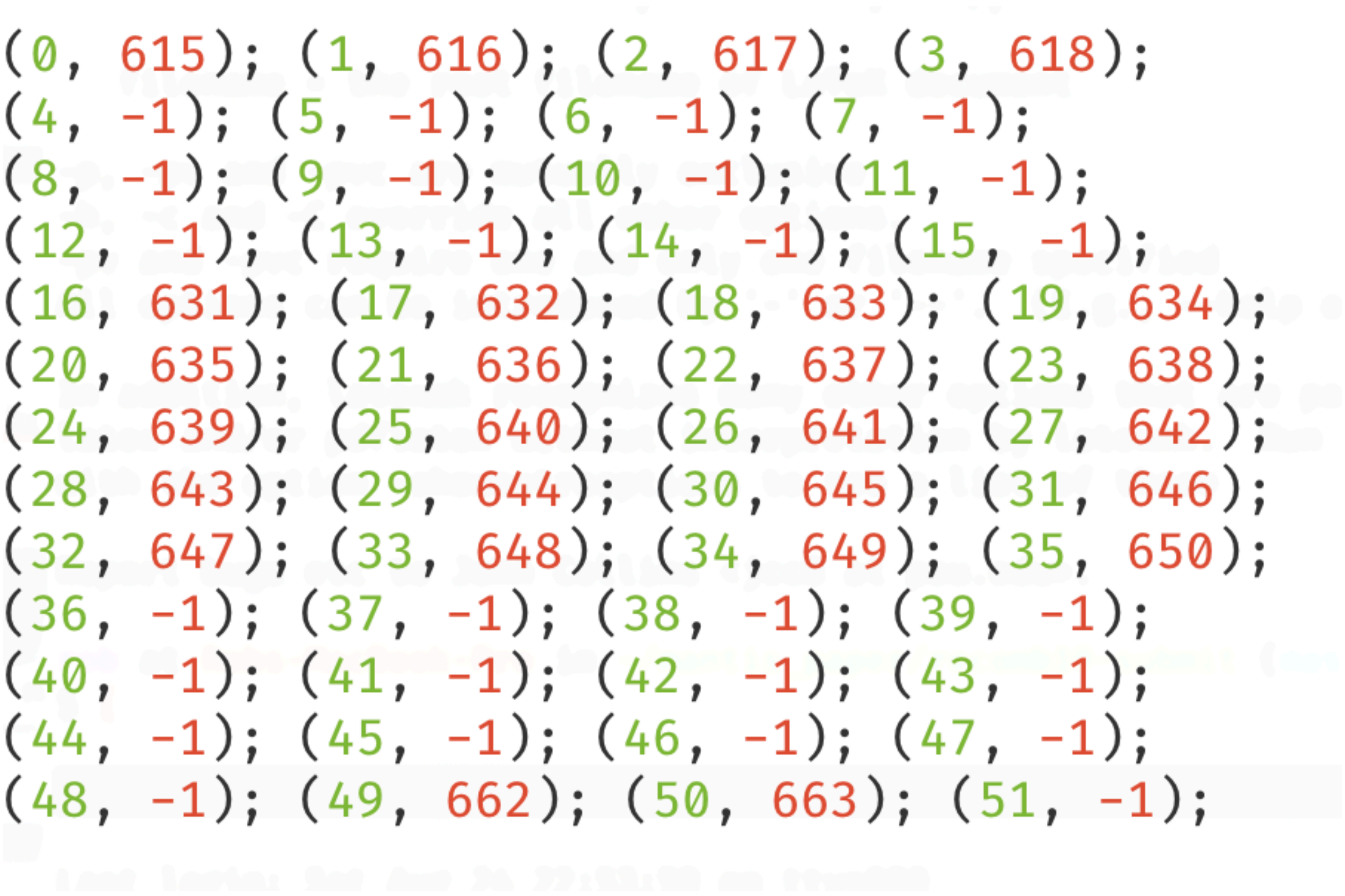
CAGGGCGCGCCGCTGGCCGAGAAAACCGCATTACAAATACTCTCAAAGCAGGATTTCTTTCTATAGGTAGTCAGTCCACAATTTTCACAT
TTGCAGCCAGGTGGACATCTTTGGGCCAATGATGAGGAGAAGGTGGAGGAAGAAGGAAATGAGGAAGAGGAGAAAGAAGAGTATGGC
ATTTTACAAACTGTGACCGTTTCTGTGTGAAGATTTTTAGCTGTCTGTCGGACTTGGGGGTCTCAGGGAAACTCACTTTGCCCCCCAGC
TGAGGTTTTCAGGAAATCTGGAAACCTACAGTCTCCAAGCCTGCTCAGCCAAGAACGCGGCGGGCGGCGGCGGTGGCGACGGCGG
CGGCGGCGCAGGACCTCCGCGCCTCCATTATGCTATTCTGCCCGCCGTGGGTGACAAACAGACGGATGCTACCAGCCCAGCCCAGT
CCCGGGGAGCCAGCTGGCCTGGGGTTCGGTCCCGCGTCTTCCCTCATTCTGTGCCGCTGCCGAGCCTGTCTCAGCTCCACACACG
CTTGGGAGCTGCAGATGCCTCCGCCCCTCCTCTCTCCCAGGCTCTTCCTGCCGTTGAACCCCGGCGGCGGGCTCTCGGCCAGCGG
CGCGCCCT**GGTACCCCCAAATTCCAAAGAGCCGAATGAAGTGGAAAGAATGGCCACAAATGGCTTGGGC**CCGAGTGACCATGGGA
TGGTTAGGTAGGATTTTAGAGGCGACTGCTCCTGGAATTAGAGAAAGAGTTTCATTACAACCGCTACCTGACCCGAAGGCACTCGGGC
CCATGCCTTCCTCTCCTTCGCTGTTTGATTTCTATTCTGTTTAGCCTGAGTTGGAGATGCCGGAGGTGCCGTGGTGGAAGAAGTCTTGA
ACGAATTTGGAGGCGTCTCCGTGGCAGCTAAGCGAGCACGGGTTCTGCTGGTGCAGGATGACACTGGCAGCCACTGCCGCGGACTT
GTATCTCTTGTCTTCCTGCTTTTATAGAGAATAGAATGACACTCACAACTCTAACTACCTGTCAGAAGCAGGCAGGAGCTAGTAAGGATGA
ATTTGTAGCAAAATTAGCAAGTGGACTTCTTTCTCCTCTTCCTCATTTCCTTCTTCCTCCACCTTCTCCTCATCTTAAATCTTTAACATACTA
CCTAAAGGGAACCTGCAATAATCTTGAAAAAGGACTTCAATCCGACGTTTTCGTGTCAAATAAGGATTAAAGAGAAACTCCTCCGCGAGC
CGTGCGCCGAGGGTGGCGGCGGGGGCCTGAAGCGTGAGGAGCCTTCAATATGTATTTAACCAGGGACCGTCGGTATGAGGTGGCCC
GGGTTCTTATTTGTTTGGGGGCTGGAGGGGGGAGACGGAGAAACAGTGAAAAGTTCCTGAGCCCCATAAAGGGACTGTCTGGGGAGC
GCCTCGTAGCCATAGAATTCCACCGCCGCCGCCCGCCGCGTAGTCGTACTTGAAGCCGAGCGCAGGCGGGTGGTTCATTAACTCTGA
CTTTGCCTTTGATTTTGCTCGACCTCTGCTTCGTCAAAATCTGGTTTCAGAATCGAAGGATGAAGATGAAAAAGATGAATAAAGGAGGAA
AAGGAAGAAAACAAGGACTAAGCAAAAAAGAAAGACCCCCCCTATAGCAGGATTTTAAAATTTTTCTCTTTTTCTTTTTCAAGATTATTGCA
AGGCGAGCGTGGTGCAATATCCCGACTGTAAATCCTCCGCCAACACTAACTTTTAAAAAAAAACACCCAGCAGGTACCATGCTAAGACA
ACATCACATGCATTATTATGACTCACGTATACAATACAAAGTACTTGGACCAGGAACAGGGTCTTTAATCCTTATTTGACACGAAAACGTCG
GATTGAAGTCTCTCATGCCCAACTAGTGGGGTTTCCTGGCACTGGACCCCAGCAAGTGGTCCTAGAGGCGAAAAGGAAGAAAACAAG
GACTAAGCAAAAAAGAAAGACCCCCCCCCCTGAAAGGATATCATGGATTCCTTAAAGAATGAGAACTTCGACATGGTGATATCACATGG
ACTTCTGGGGCCGAGTGAAGAATTTTCTGATGTTCTTTAGTTAGCAGTCCTAAACCCTACCCAGCCTGCTGCCTCAGCACAGCCAAGG
GAAAATCAGCAAGGGCTATGCTGTGATTCTGTCTCTGAGTGACTTGGACCACTGTTGATTTTTATTTTTTTCTCTTTTTTTCTCCTATAGCA
GGATTTTAAAATCGGGCCCACCTTAACTCGGGAGGGCCGCGCTGAGGCTGGGAGCCGGAGATTCGGGCGAGGGCAGTGTCTGCGG
GGCGCGGTCGCGCAGCTCCCCGGGCGAGCCAGGTGCAGCCTTGGCGGGGTCTGTTTGGTGGGCGATGTCACCATTTCCCGCCGCC
GCCGTCGCCACCGCCGCCGCCCGCCGCGTAGTCGTACTTGAATAGCTGGACATAAAGACAAATGACAAAAAATTATTATTATAGATATATT
TGGTCTGTGTGTTATGTCCTAAGGTGTTTTGTCTGCAGTTTGAGAGCATGTTGCTGGTAGCCTGAGTTGGAGAT

How can exact matching help?

List of tuples of 13-mers;

(position in the query, position in the reference)

-1 indicates not found (Python behavior)



```
(0, 615); (1, 616); (2, 617); (3, 618);
(4, -1); (5, -1); (6, -1); (7, -1);
(8, -1); (9, -1); (10, -1); (11, -1);
(12, -1); (13, -1); (14, -1); (15, -1);
(16, 631); (17, 632); (18, 633); (19, 634);
(20, 635); (21, 636); (22, 637); (23, 638);
(24, 639); (25, 640); (26, 641); (27, 642);
(28, 643); (29, 644); (30, 645); (31, 646);
(32, 647); (33, 648); (34, 649); (35, 650);
(36, -1); (37, -1); (38, -1); (39, -1);
(40, -1); (41, -1); (42, -1); (43, -1);
(44, -1); (45, -1); (46, -1); (47, -1);
(48, -1); (49, 662); (50, 663); (51, -1);
```

So the basic strategy is :

Find **exact** matches shared between the read and reference

Group exact matches into regions likely to support a high-quality alignment

Score / validate each hit location, and filter the ones that fail to yield a high-quality alignment.

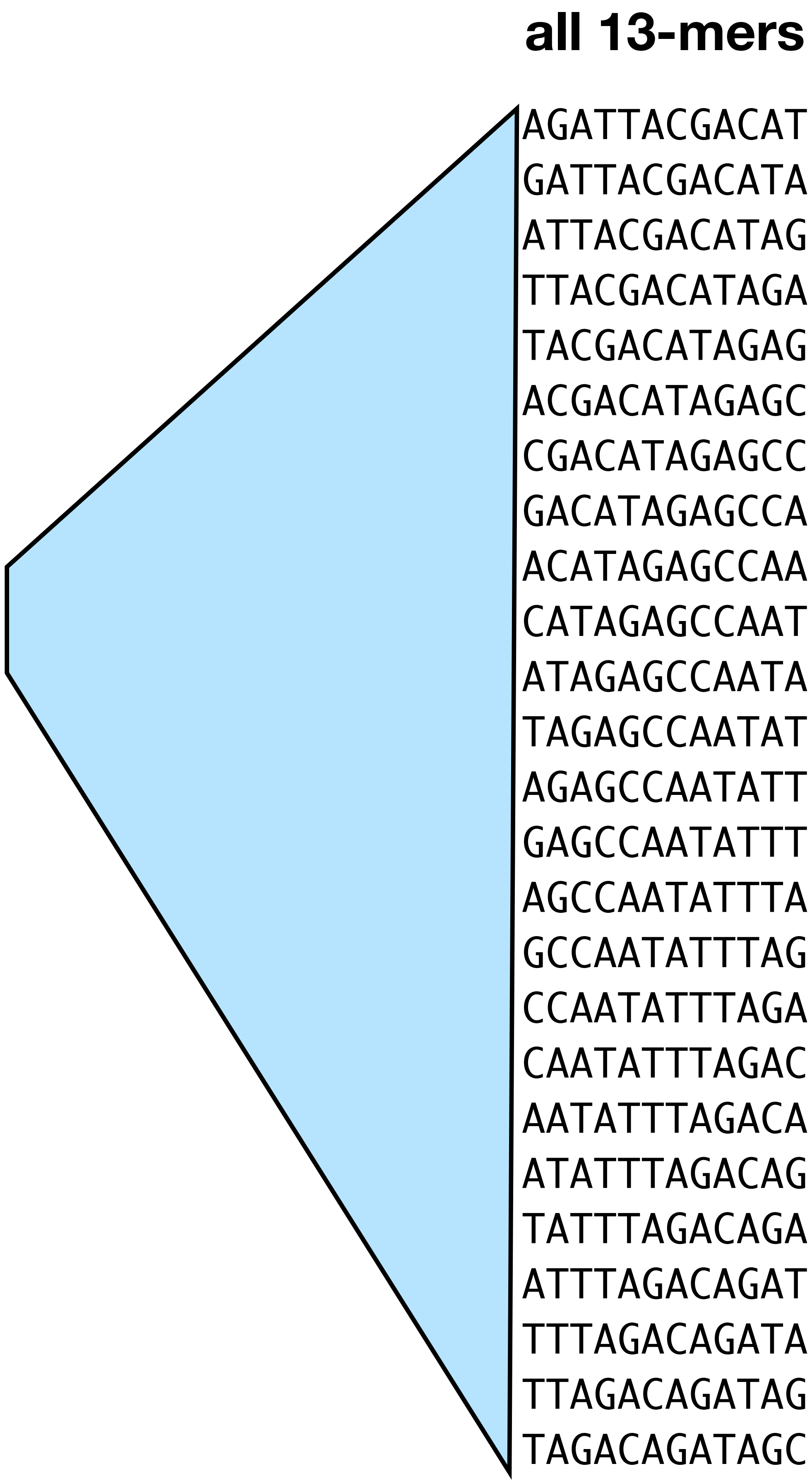
There are **many** variations on this theme. What is a good type of seed? How should we search for seeds? How should we group seeds? How aggressively should we filter potential loci?

Common types of seeds

K-mers: fixed substrings of length k.

string

AGATTACGACATAGAGCCAATATTTAGACAGATAGC



all 13-mers

AGATTACGACAT
GATTACGACATA
ATTACGACATAG
TTACGACATAGA
TACGACATAGAG
ACGACATAGAGC
CGACATAGAGCC
GACATAGAGCCA
ACATAGAGCCAA
CATAGAGCCAAT
ATAGAGCCAATA
TAGAGCCAATAT
AGAGCCAATATT
GAGCCAATATTT
AGCCAATATTTA
GCCAATATTTAG
CCAATATTTAGA
CAATATTTAGAC
AATATTTAGACA
ATATTTAGACAG
TATTTAGACAGA
ATTTAGACAGAT
TTTAGACAGATA
TTAGACAGATAG
TAGACAGATAGC

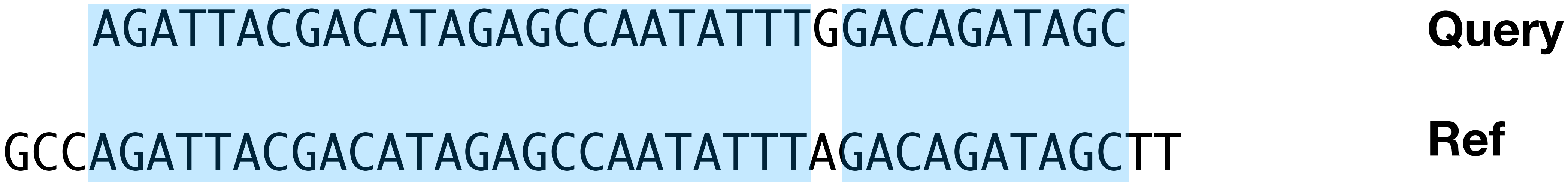
In some sense, k-mers are the “simplest” type of seed.

K-mers in the text are independent of the query.

How might we “index” k-mers in the text?

Common types of seeds

MEMs: Maximal Exact Matches



```
   AGATTACGACATAGAGCCAATATTTGGACAGATAGC        Query

GCCAGATTACGACATAGAGCCAATATTTAGACAGATAGCTT      Ref
```

An exact match shared between the query and the reference that *cannot be extended, in either direction, without introducing a mismatch.*

Unlike k-mers, the MEMs depend on both the reference and the specific query.

To find MEMs efficiently, we'll need a "full-text" index, not just a token / k-mer index.

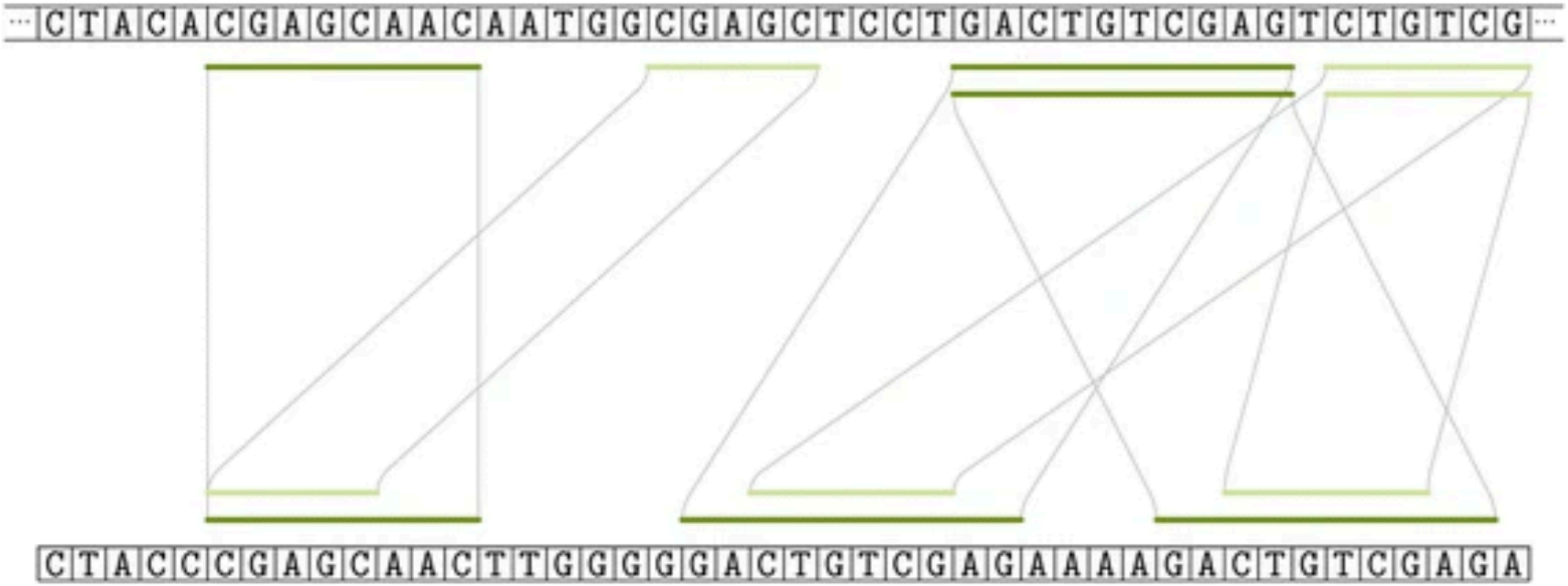
Because of their "context dependence", MEMs can be more specific than k-mers, though we don't often deal with individual seeds in isolation.

Common types of seeds

SMEMs: Super-Maximal Exact Matches

🟩 **MEMs**

🟩 **SMEMs**

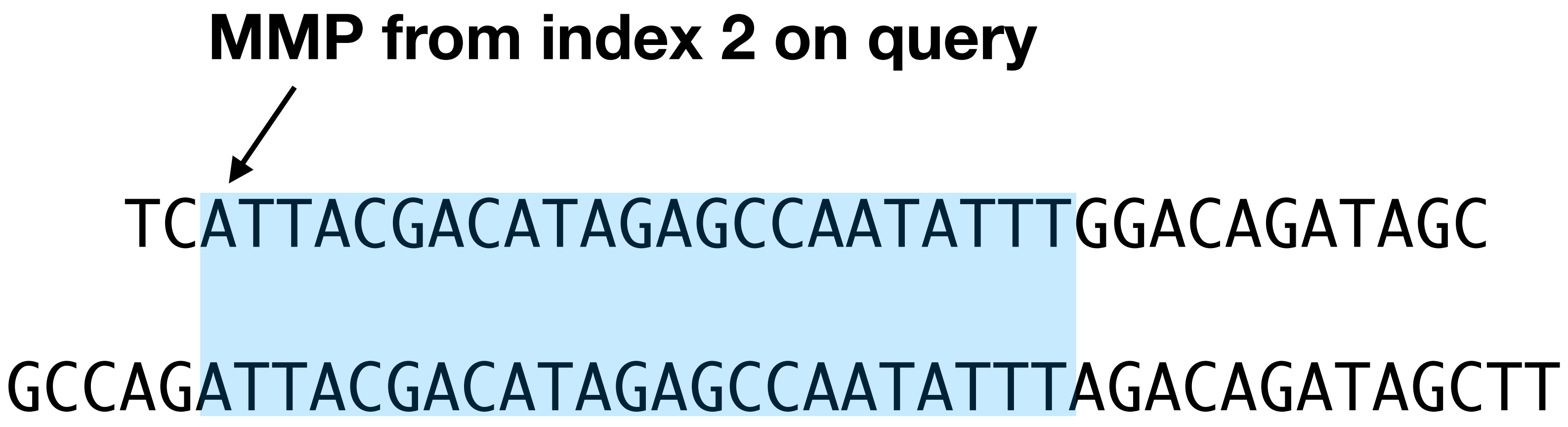

···CTACACGAGCAACAATGGCGAGCTCCTGACTGTCGAGTCTGTCG···
CTACCCGAGCAACTTGGGGGACTGTCGAGAAAAGACTGTCGAGA

An exact match shared between the query and the reference that *cannot be extended, in either direction, without introducing a mismatch.* Also, an SMEM is not contained within any other exact match on either the query or reference.

Vyverman, Michaël, et al. "A long fragment aligner called ALFALFA." *BMC bioinformatics* 16.1 (2015): 159.

Common types of seeds

MMPs: Maximum Mappable Prefixes



MMP from index 2 on query

```
    TCATTACGACATAGAGCCAATATTTGGACAGATAGC
GCCAGATTACGACATAGAGCCAATATTTAGACAGATAGCTT
```

A prefix of (some suffix of) a query that is an exact match with a substring of the reference, and which cannot be extended further without introducing a mismatch.

Similar to a MEM, but extension only works in one direction — the MMP depends on the query, reference and start position. Originally introduced in STAR aligner. Useful for mapping read “pieces” across exons.

Dobin, Alexander, et al. "STAR: ultrafast universal RNA-seq aligner." *Bioinformatics* 29.1 (2013): 15-21.

Indexing

Hopefully, I’ve convinced you of the importance of being able to quickly find different types of exact matches (seeds).

The next few lectures will be about data structures, and the corresponding search algorithms, that will enable this on *genome-scale* data.

We will consider both “full-text” indexing and inverted “token-based” indexing (k-mers).

There are many amazing results in the literature about indexing large text corpora; but there are likely still improvements to be made!