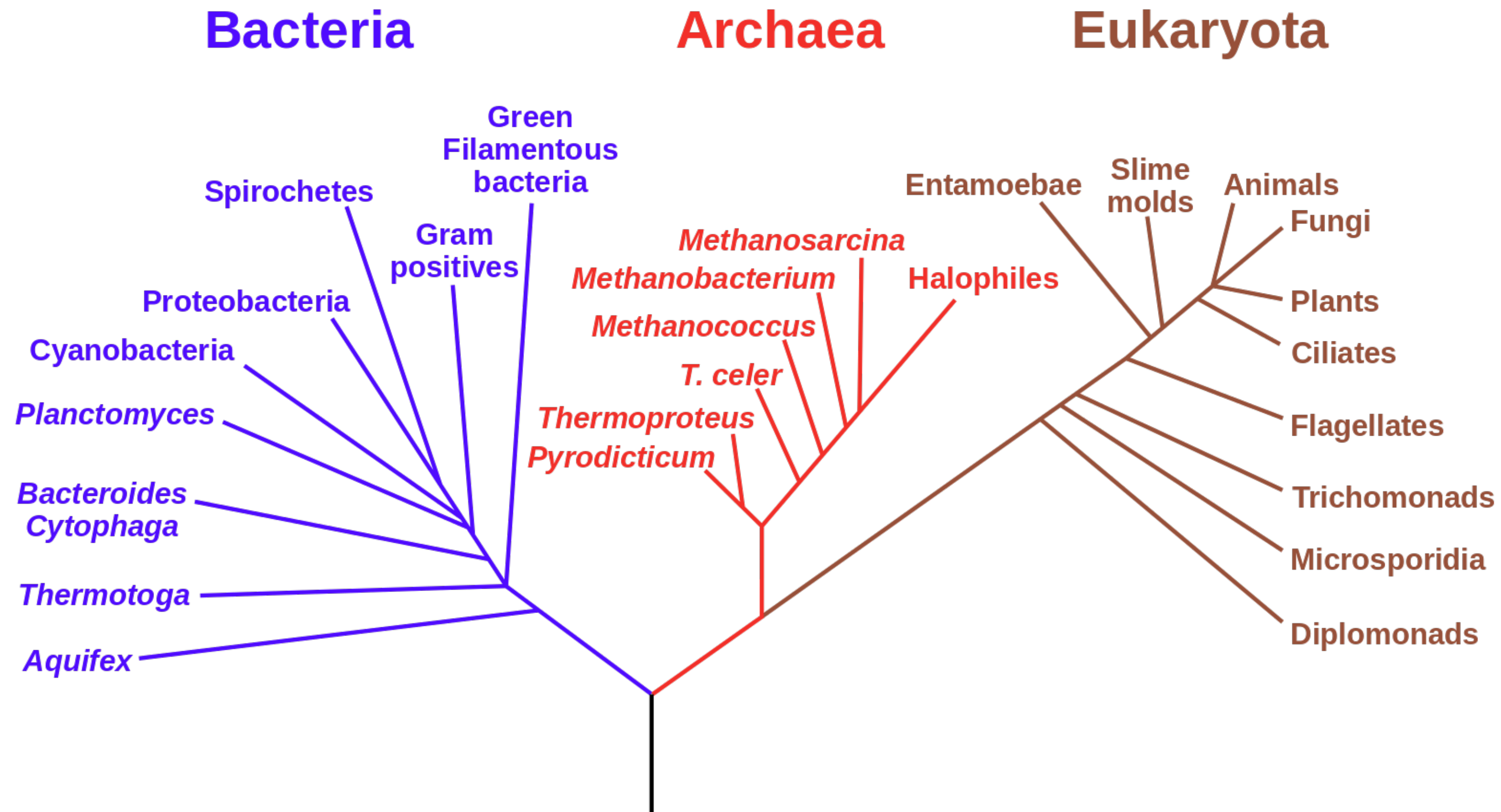


Sequence similarity and global alignment

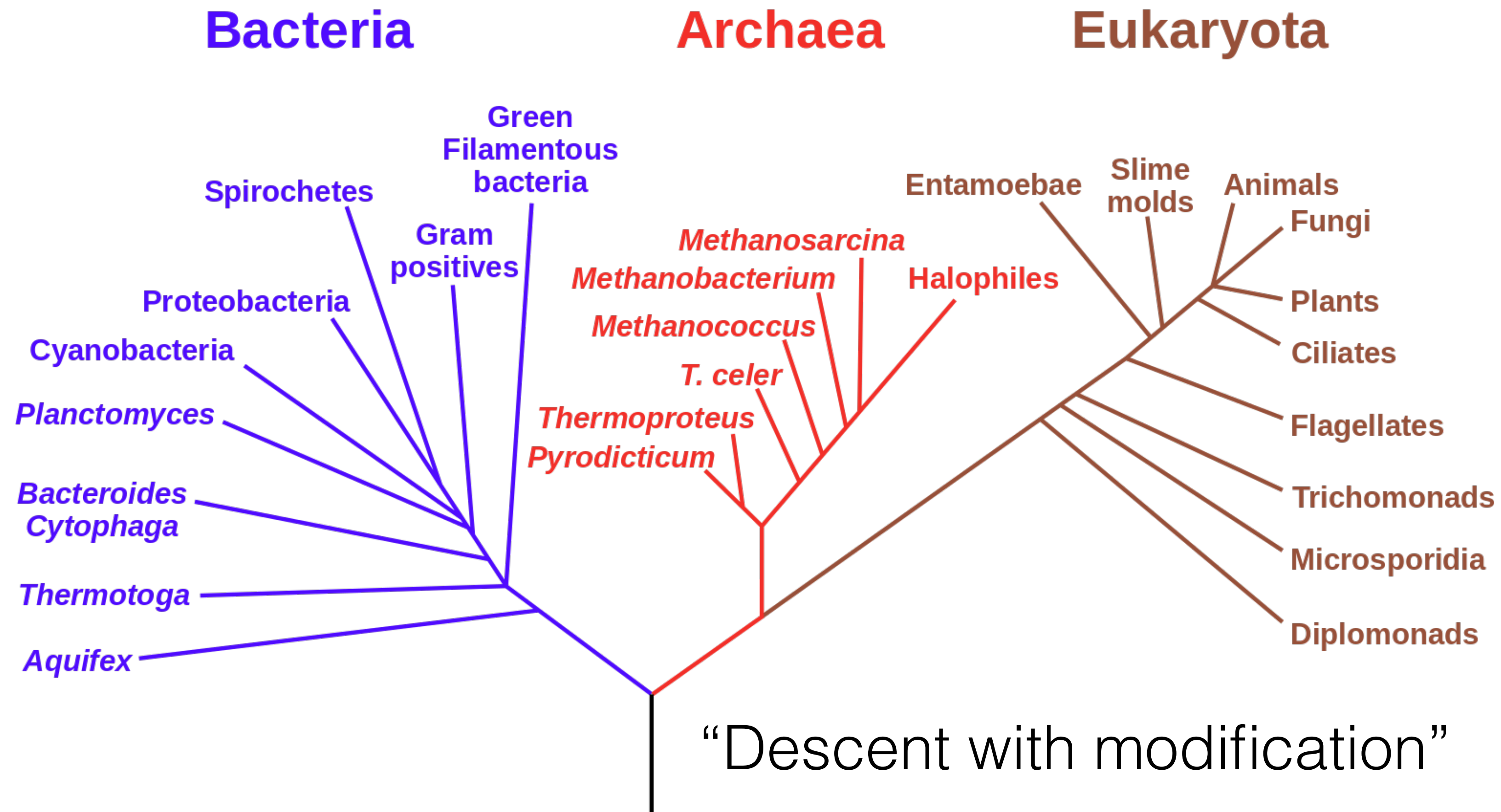
Relatedness of Biological Sequence

Phylogenetic Tree of Life



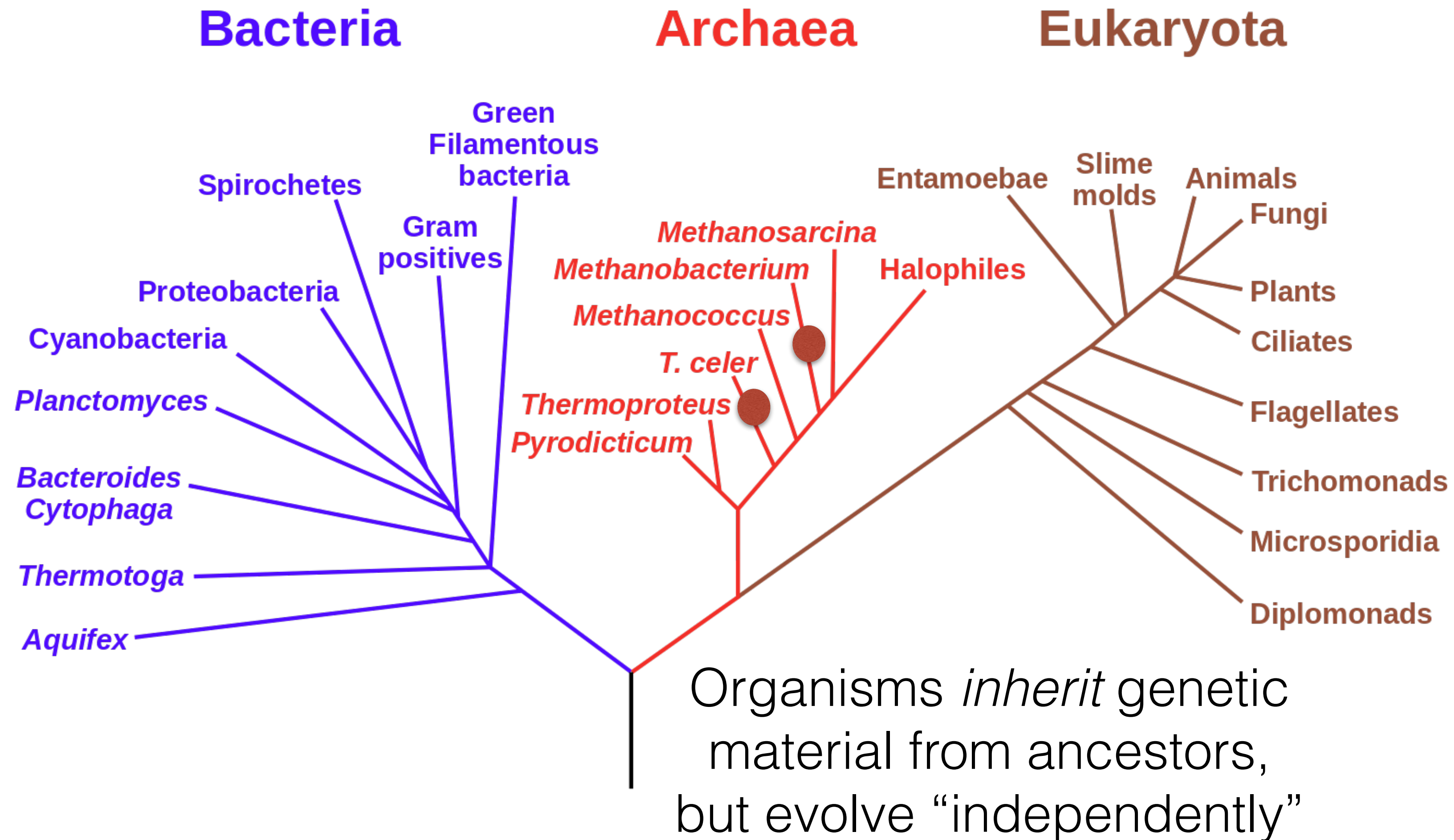
Relatedness of Biological Sequence

Phylogenetic Tree of Life



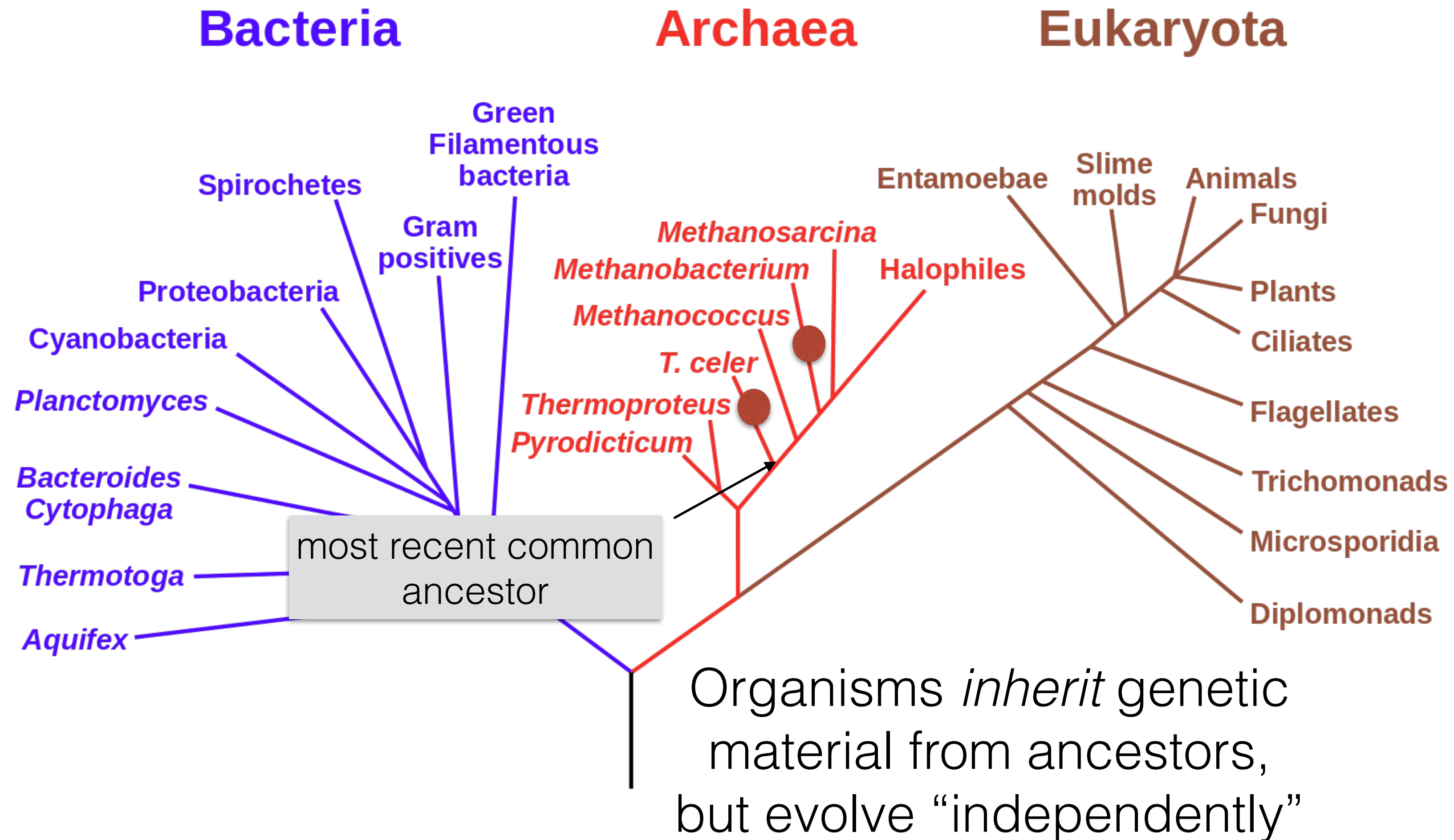
Relatedness of Biological Sequence

Phylogenetic Tree of Life

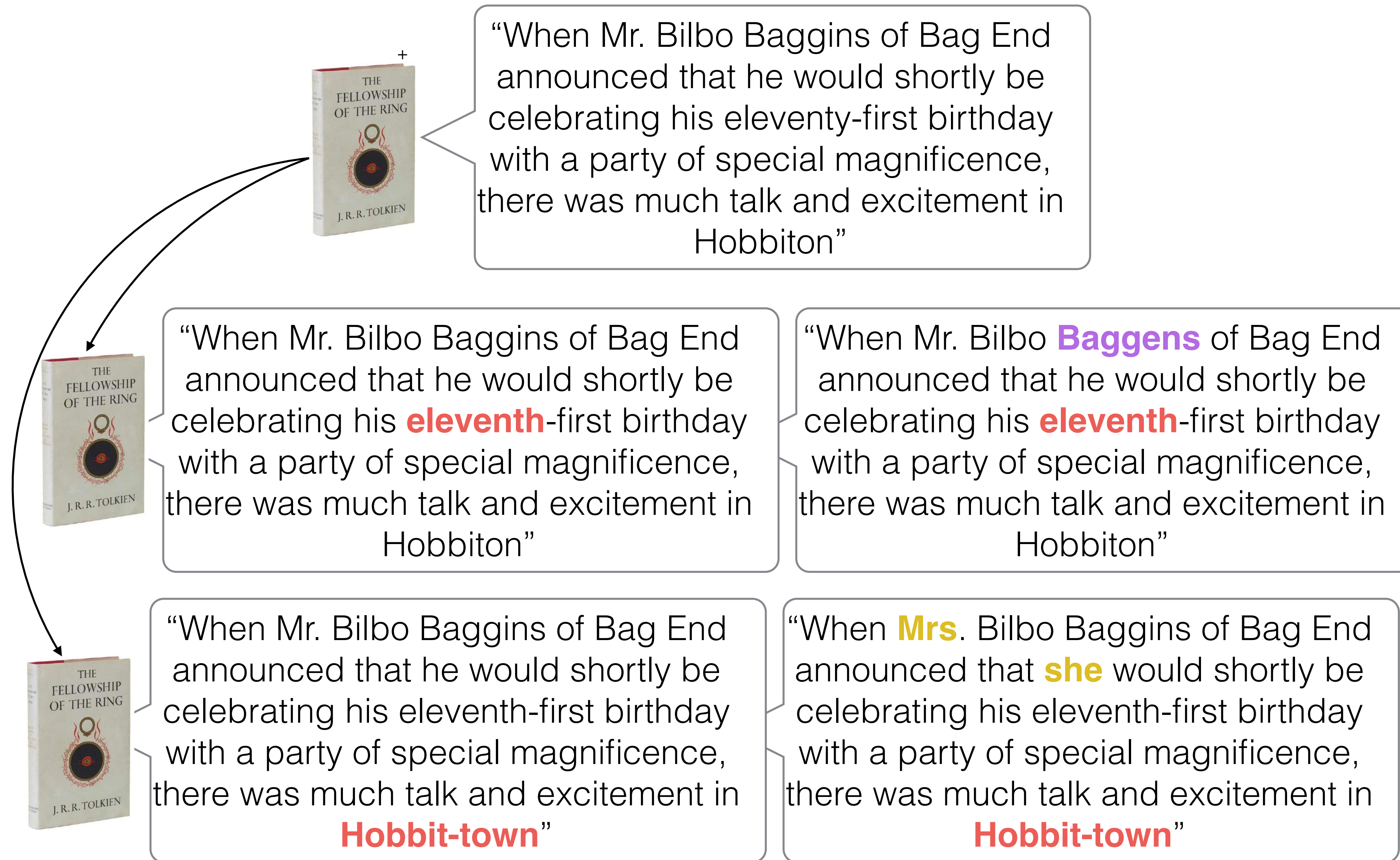


Relatedness of Biological Sequence

Phylogenetic Tree of Life



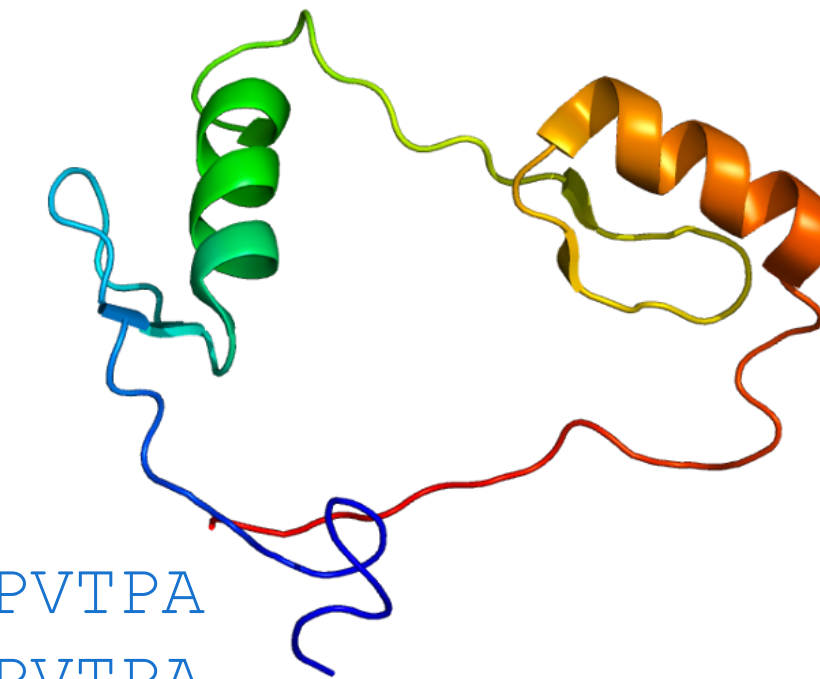
Consider an analogy



Sequence tells a story

- If two sequences are *similar*, this provides evidence of descent from a common ancestor
- Sequences are *conserved* at different rates
- Very similar sequence can indicate a very *recent common ancestor*, or a *highly conserved function*

Why compare DNA or protein sequences?



Partial CTCF protein sequence in 8 organisms:

<i>H. sapiens</i>	-EDSSDS-ENAEPDLDDNEDEEEPAVEIEPEPE-----PQPVTTPA
<i>P. troglodytes</i>	-EDSSDS-ENAEPDLDDNEDEEEPAVEIEPEPE-----PQPVTTPA
<i>C. lupus</i>	-EDSSDS-ENAEPDLDDNEDEEEPAVEIEPEPE-----PQPVTTPA
<i>B. taurus</i>	-EDSSDS-ENAEPDLDDNEDEEEPAVEIEPEPE-----PQPVTTPA
<i>M. musculus</i>	-EDSSDSEENAEPDLDDNEEEEPAVEIEPEPE--PQPQPPPPPQPVAPA
<i>R. norvegicus</i>	-EDSSDS-ENAEPDLDDNEEEEPAVEIEPEPEPQPQPQPQPQPQPVAPA
<i>G. gallus</i>	-EDSSDSEENAEPDLDDNEDEEETAVEIEAEPE-----VSAEAPA
<i>D. rerio</i>	DDDDDSDEHGEPDLDDIDEEDEDDL-LDEDQMGLLDQAPPSVPIP-APA

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

Sequence can reveal structure



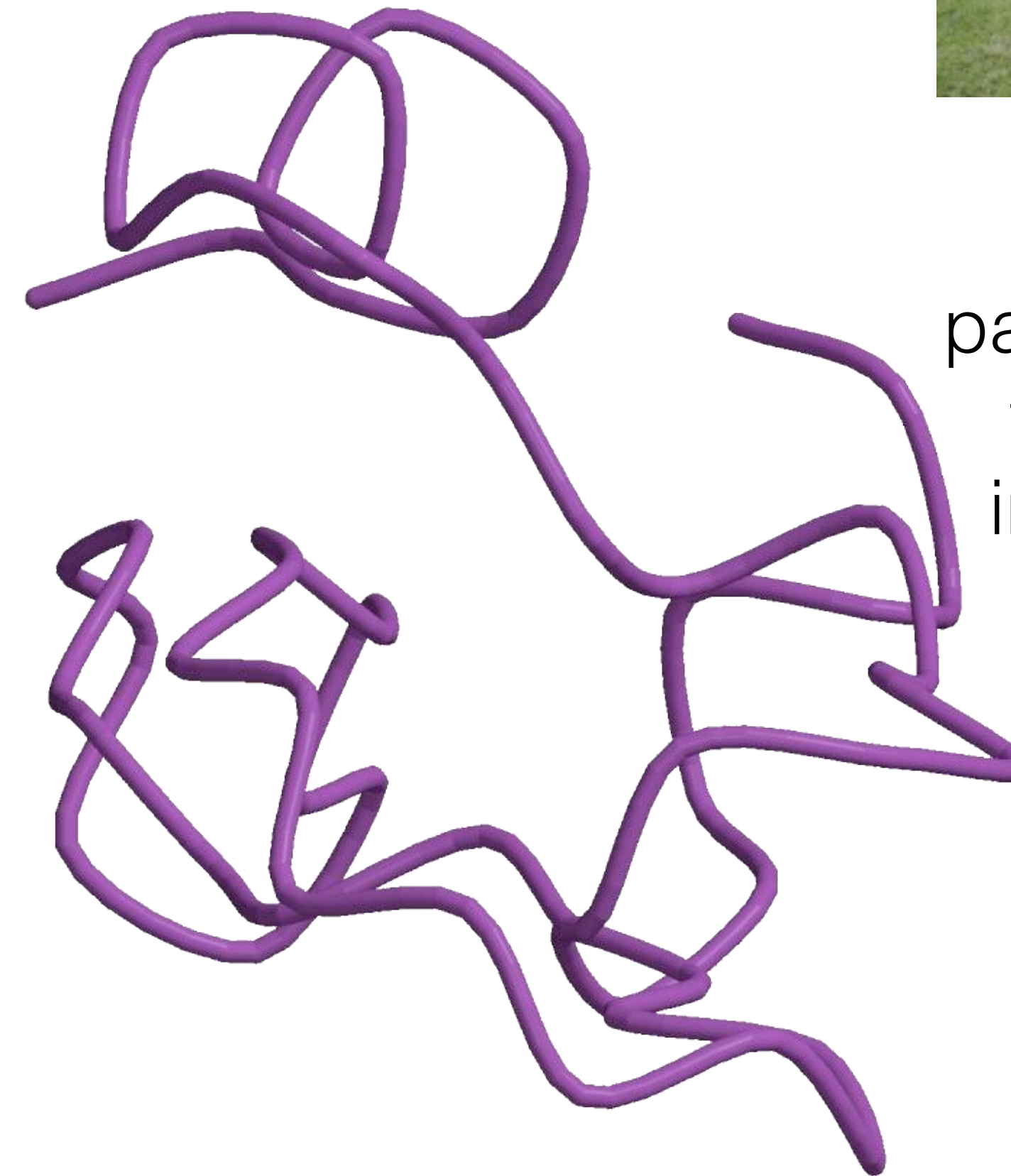
dendrotoxin K



(a) 1dtk



Bovine
pancreatic
trypsin
inhibitor



(b) 5pti

1dtk XAKYCKLPLRIGPCKRKIPSFYKWKAKQCLPFDYSGCGGNANRFKTIEECRRTC VG-
5pti RPDFCLEPPYTGPCKARIIRYFYNAKAGLCQTFVYGGCRAKRNNFKSAEDCMRTC GGA

Why Not Exact Matching?

Suffix tree / array and BWT / FM-index are powerful tools for finding exact patterns in a large text, but exact matching is insufficient. Reads have **errors** and there is **true genomic variation** between a reference and a sample.

Typical strategy (many variants):

- Find all places where a substring of the query matches the reference exactly (seeds)

Requires efficient exact search, as we covered with the suffix array/FM-index

- Filter out regions with insufficient exact matches to warrant further investigation

“Filtering”; we may discuss this later

- Perform a “constrained” alignment that includes these exact matching “seeds”

Here is where we use our “alignment” scoring (current lecture)

The Language of Strings

A **string** \mathbf{s} is a finite sequence of characters

$|\mathbf{s}|$ denotes the **length** of the string — the number of characters in the sequence.

A string is defined over an **alphabet**, Σ

$$\Sigma_{\text{DNA}} = \{A, T, C, G\}$$

$$\Sigma_{\text{RNA}} = \{A, U, C, G\}$$

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

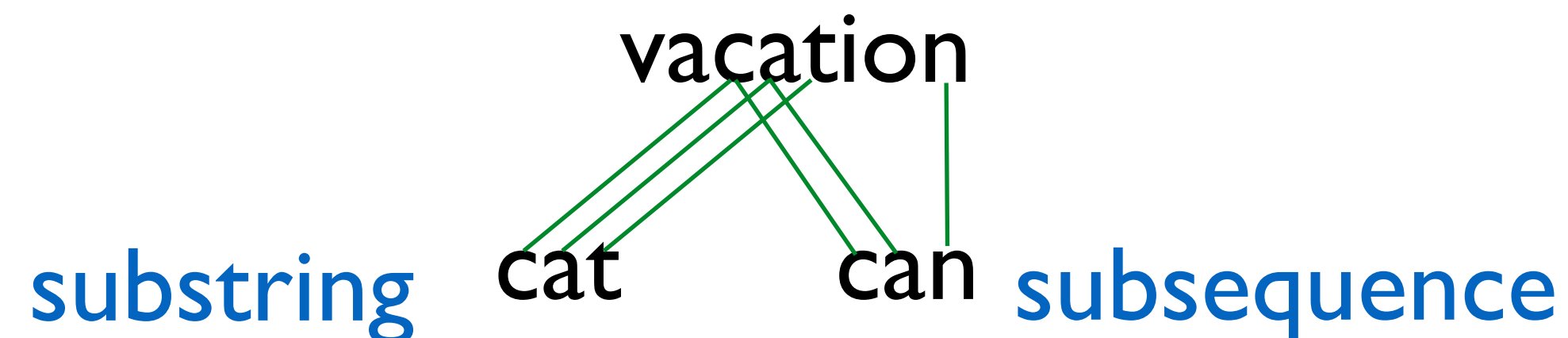
The **empty string** is denoted ϵ — $|\epsilon| = 0$

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**



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

The Simplest String Comparison Problem

Given: Two strings

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

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

riddle $\xrightarrow{\text{delete}}$ ridle $\xrightarrow{\text{mutate}}$ riple $\xrightarrow{\text{insert}}$ 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 $cost(x,y) = \text{mismatch penalty}$.

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 + $gap \times \text{number of - characters inserted}$.

e.g. $gap = 3$
 $cost('D', 'P') = 1$

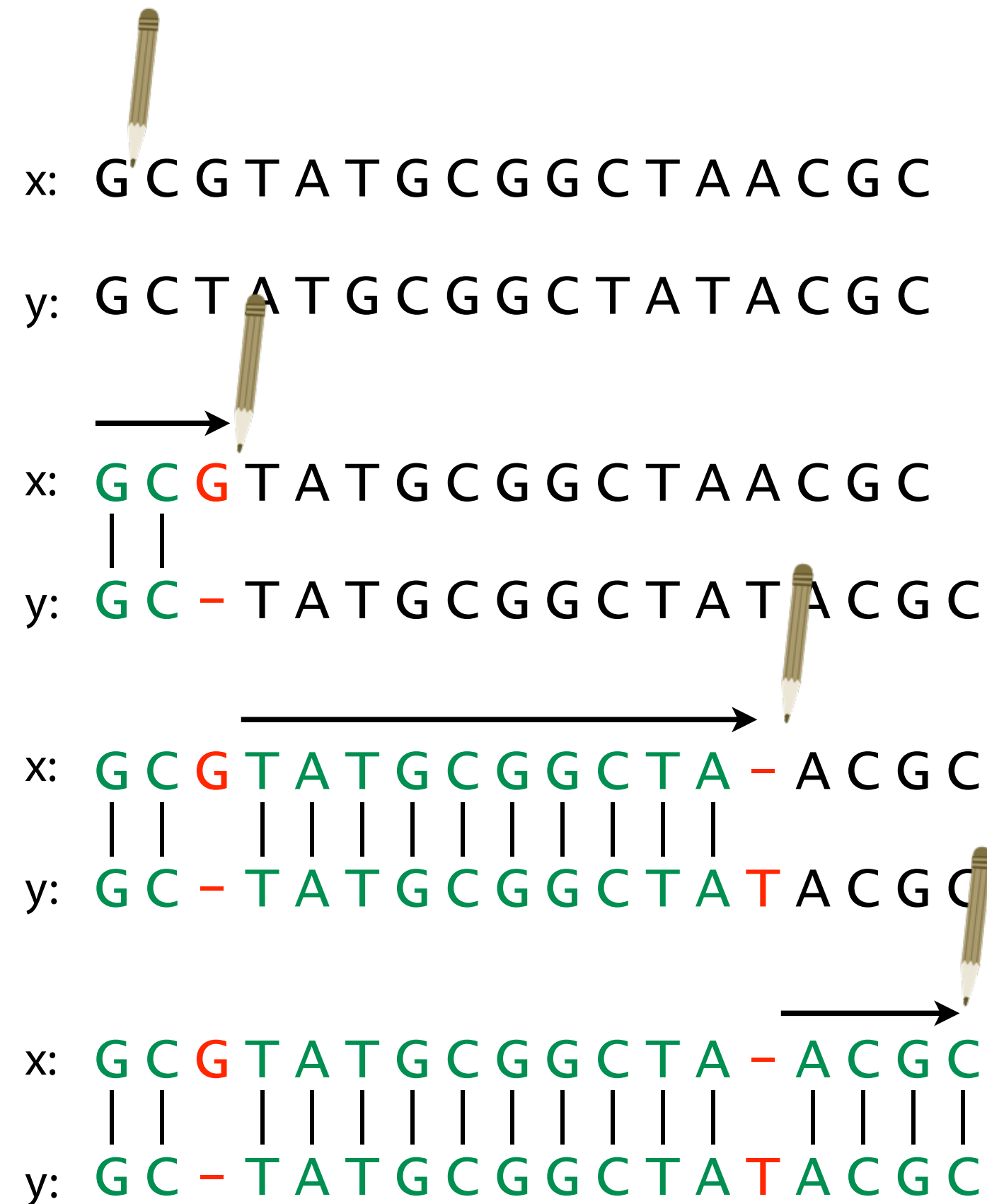
- RIDDLE
TRIP - LE

Total cost = $3+0+0+1+3+0+0 = 7$

*

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 .



Operations:

M = match, **R** = replace,

I = insert into x , **D** = delete from x

MMD

MMDMMMMMMMMMI

MMDMMMMMMMMMI MMM

Representing edits as alignments

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

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

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

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

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

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

NCBI BLAST DNA Alignment

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

```
Query 1650 gtgtgtgtgggtgcacatttgtgtgtgtgtg'gcgcctgtgtgtgtgggtgcctgtgtgtgt 1709
          ||||| ||| | ||||| ||| ||| |||||
Sbjct 56838 GTGTGTGTGGAAGTGAGTTCATCTGTGTGTGCACATGTGTGTGCA--TGCATGCATGTGT 56895

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

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

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

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

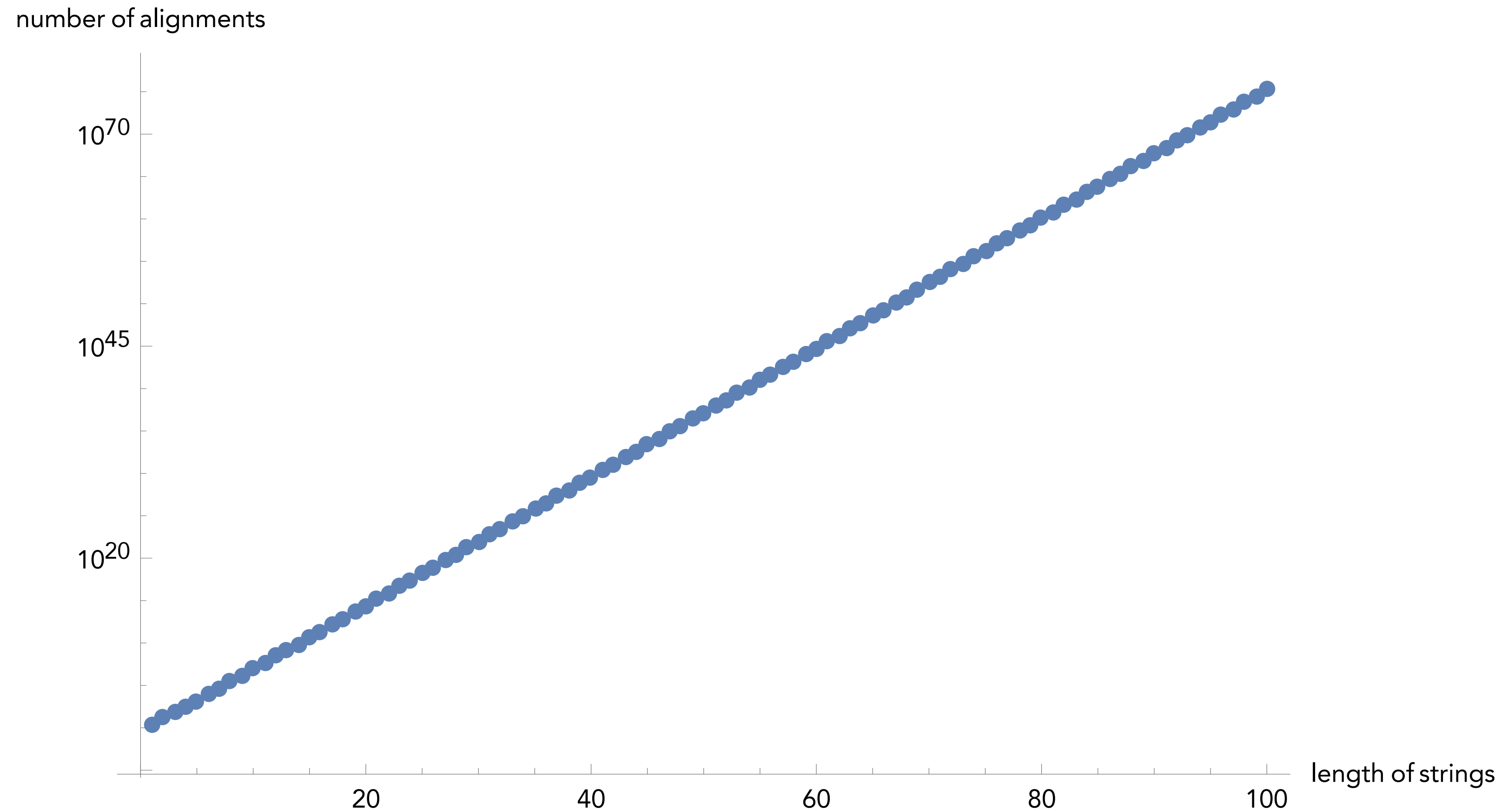
Query 1943 atttgtgtgtgtgtgtgcctgtgtgtgtgggtgcacatttgtgtgtgtgcctgtgtgtgg 2002
          || |||| | ||||| ||||| || ||| | ||||| |||||
Sbjct 57115 ATCTGTGTATGTGTGTG--TGTGTGAGTTAGTTCA----TCTGTGTGTGAGAGTGTGTGA 57168

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

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

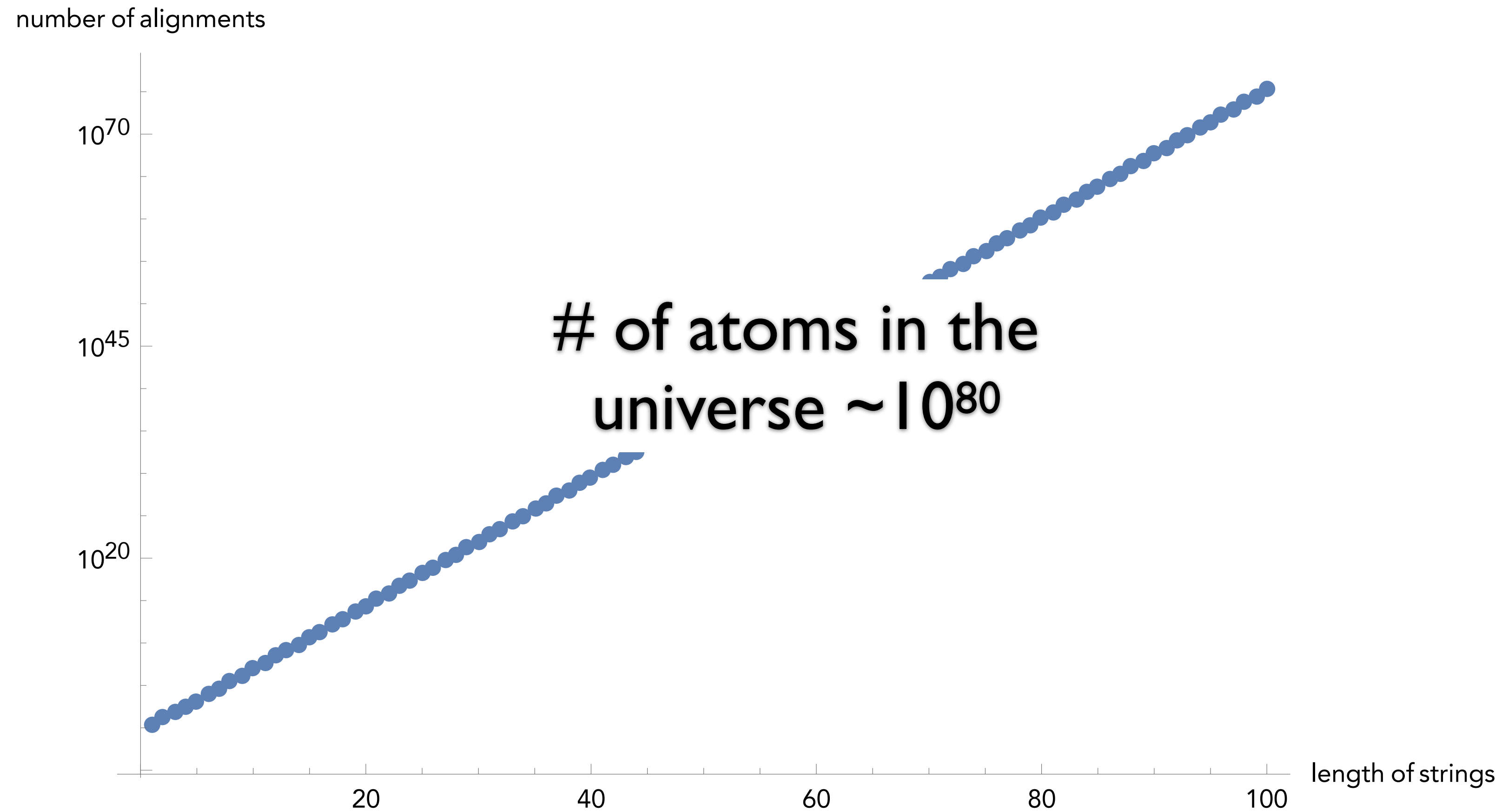
*

How many alignments are there?



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

How many alignments are there?



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

Interlude: Dynamic Programming

General and powerful *algorithm design* technique

“Programming” in the mathematical sense —
nothing to do with e.g. code

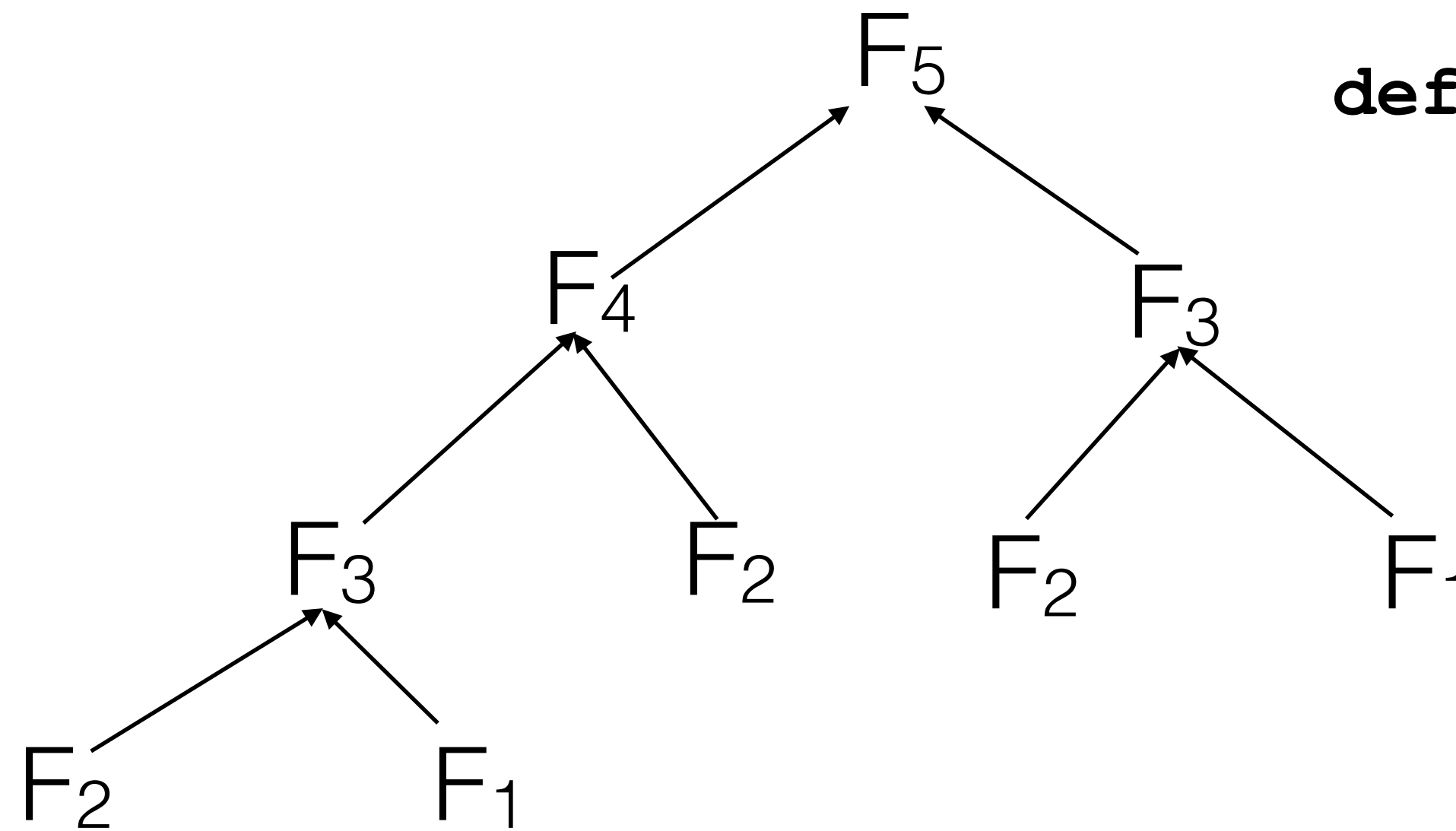
To apply DP, we need **optimal substructure** and
overlapping subproblems

optimal substructure — can combine solutions to
“smaller” problems to generate solutions to “larger”
problems.

overlapping subproblems — solutions to
subproblems can be “re-used” in multiple contexts
(to solve multiple) larger problems

Example 1: Fibonacci Sequence

$$F_n = F_{n-1} + F_{n-2} \quad \text{with } F_1 = F_2 = 1$$



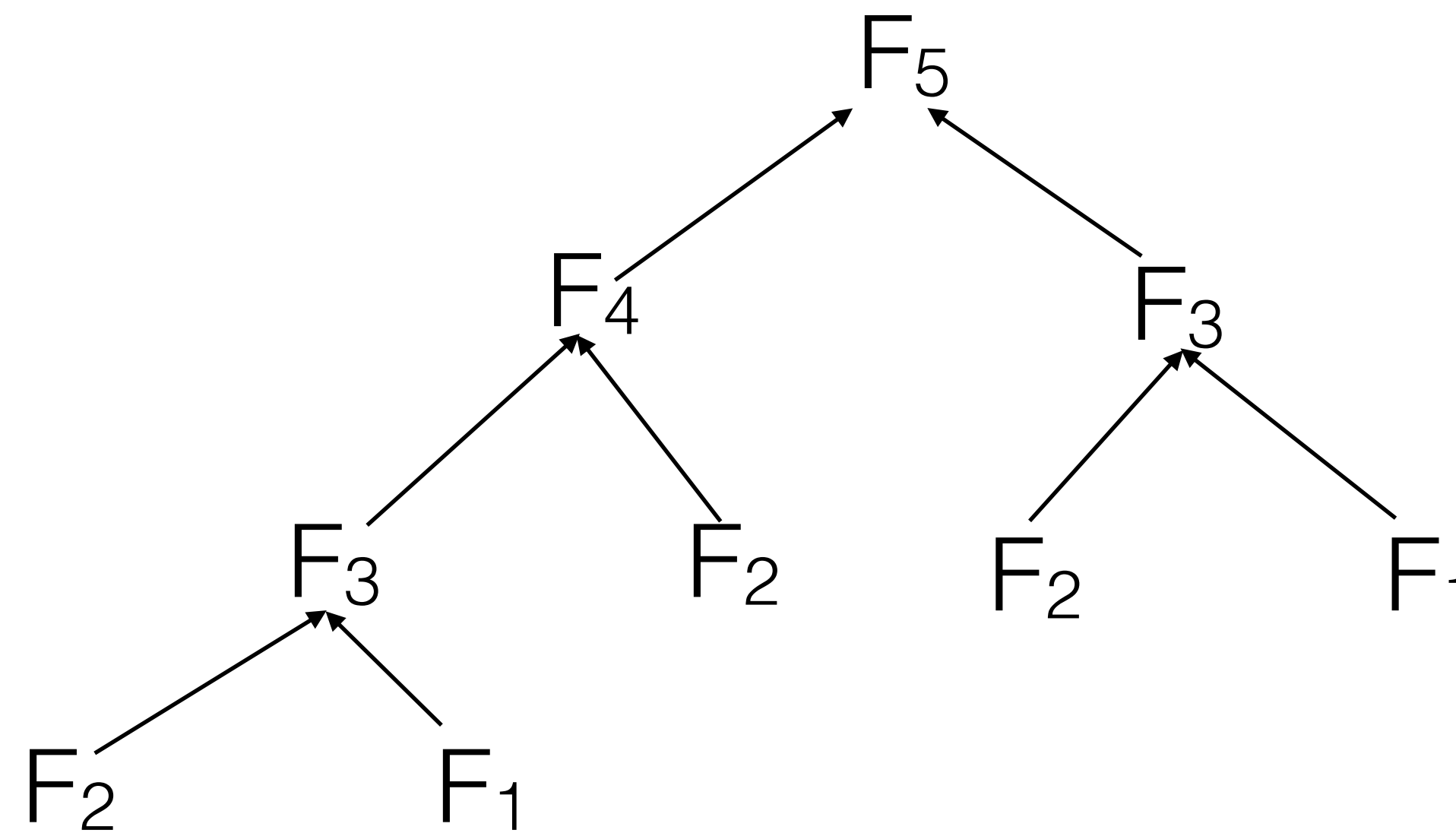
```
def fib(n):  
    if n == 1 or n == 2:  
        return 1  
    else:  
        return fib(n-1) + fib(n-2)
```

This recursive way of computing $\text{fib}(n)$ is **very** inefficient!

What is the runtime of this approach (i.e. $\text{fib}(n) = O(?)$)

Example 1: Fibonacci Sequence

$$F_n = F_{n-1} + F_{n-2} \quad \text{with } F_1 = F_2 = 1$$



```
def fib(n):  
    if n == 1 or n == 2:  
        return 1  
    else:  
        return fib(n-1) + fib(n-2)
```

This recursive way of computing $\text{fib}(n)$ is **very** inefficient!

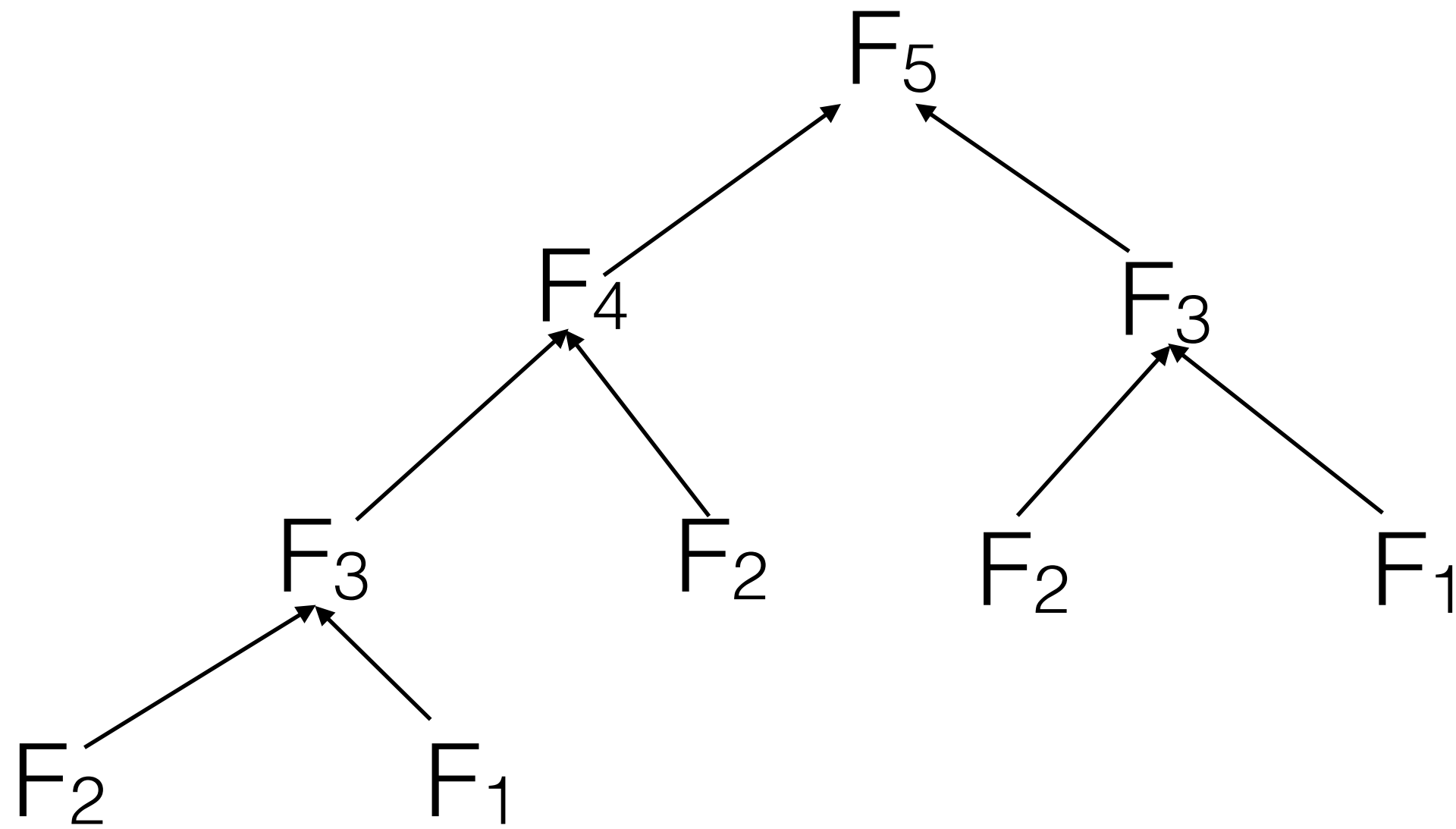
Runtime of this approach is $\text{fib}(n) = O(\phi^n) = O(2^n)$

golden ratio

Example 1: Fibonacci Sequence

$$F_n = F_{n-1} + F_{n-2} \quad \text{with } F_1 = F_2 = 1$$

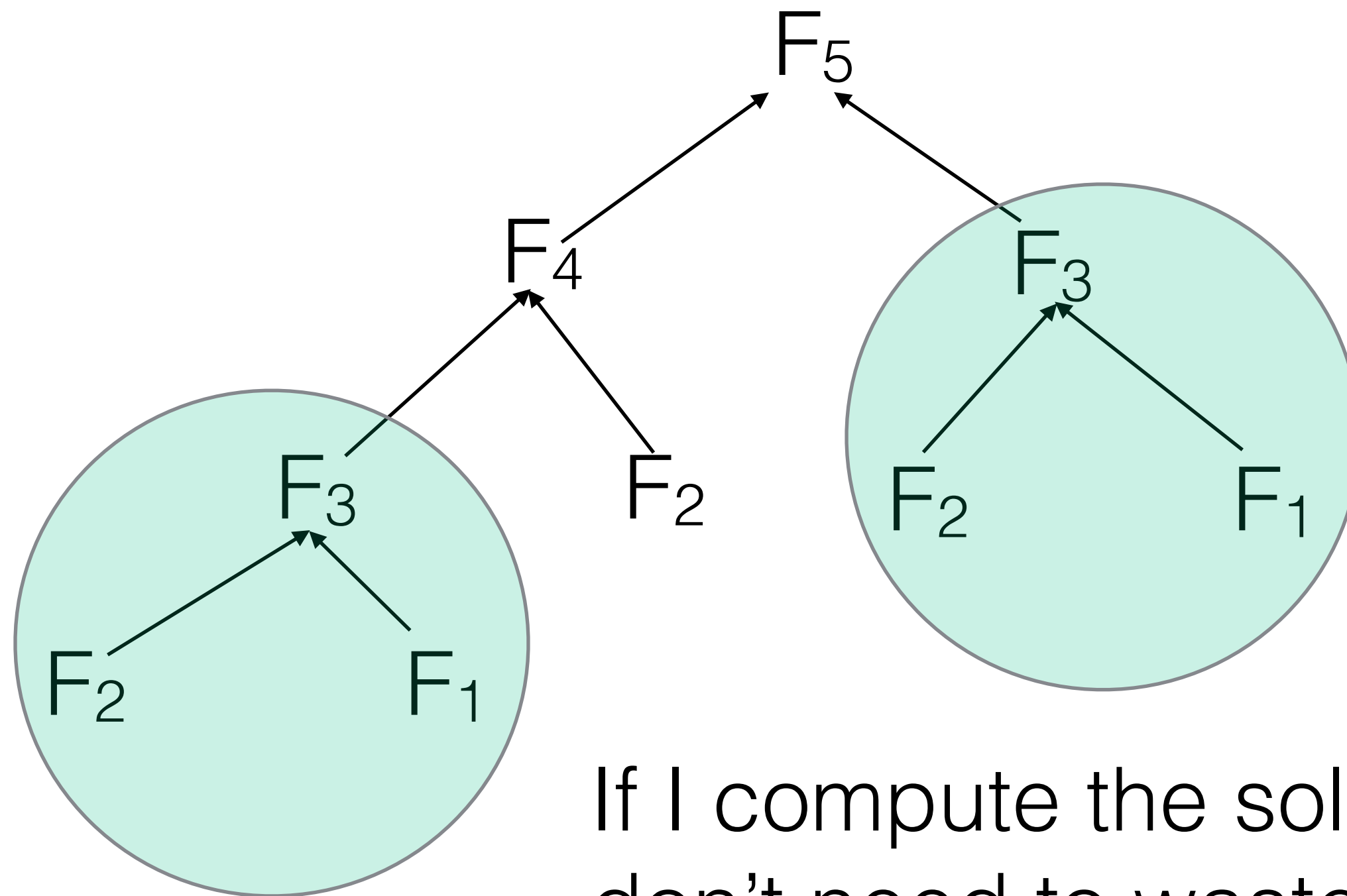
How do we do better than $O(\phi^n)$?



Example 1: Fibonacci Sequence

$$F_n = F_{n-1} + F_{n-2} \quad \text{with } F_1 = F_2 = 1$$

How do we do better than $O(\phi^n)$?

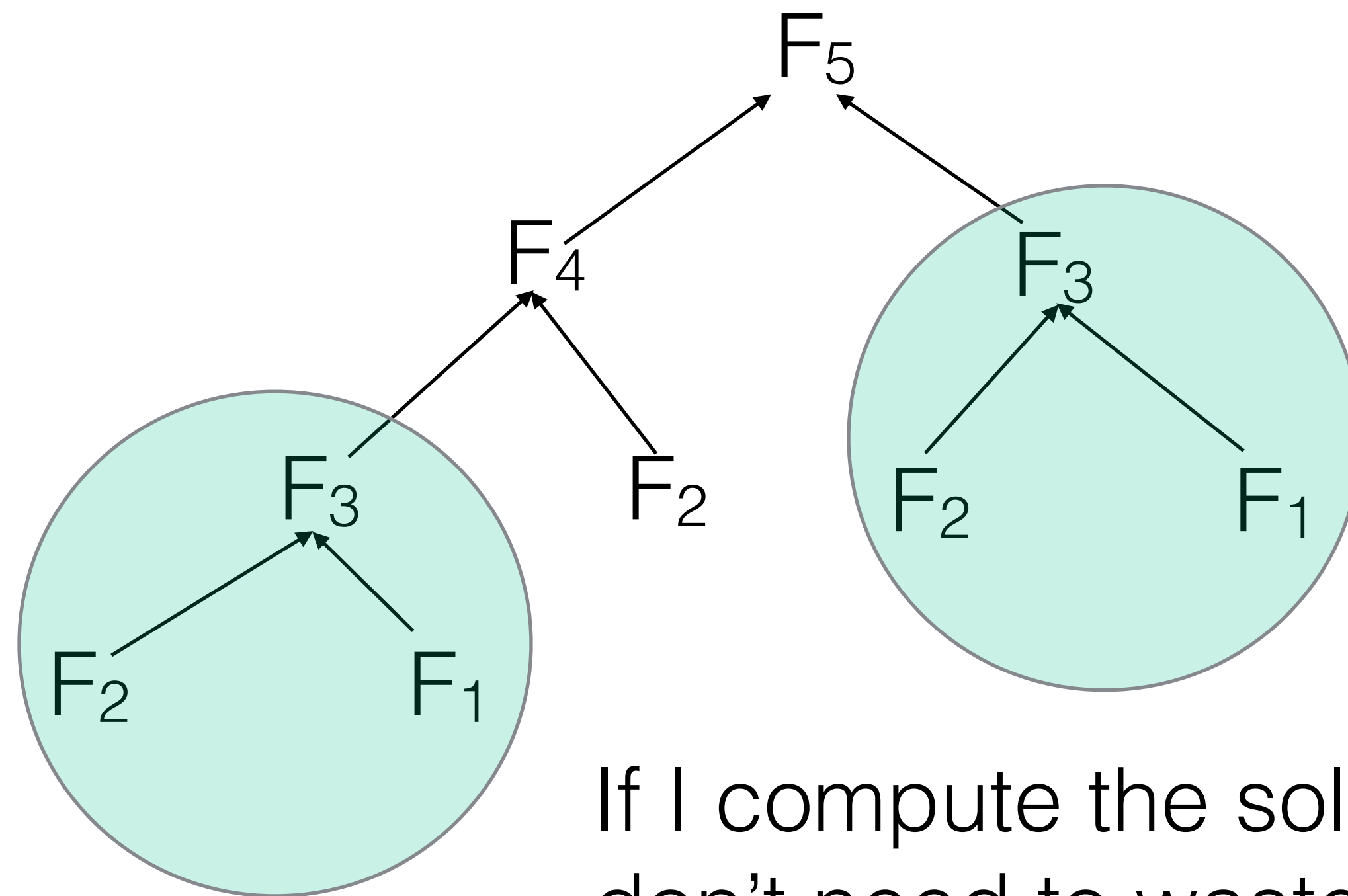


If I compute the solutions in the “right order”, I don’t need to waste time re-computing the same values.

Example 1: Fibonacci Sequence

$$F_n = F_{n-1} + F_{n-2} \quad \text{with } F_1 = F_2 = 1$$

How do we do better than $O(\phi^n)$?



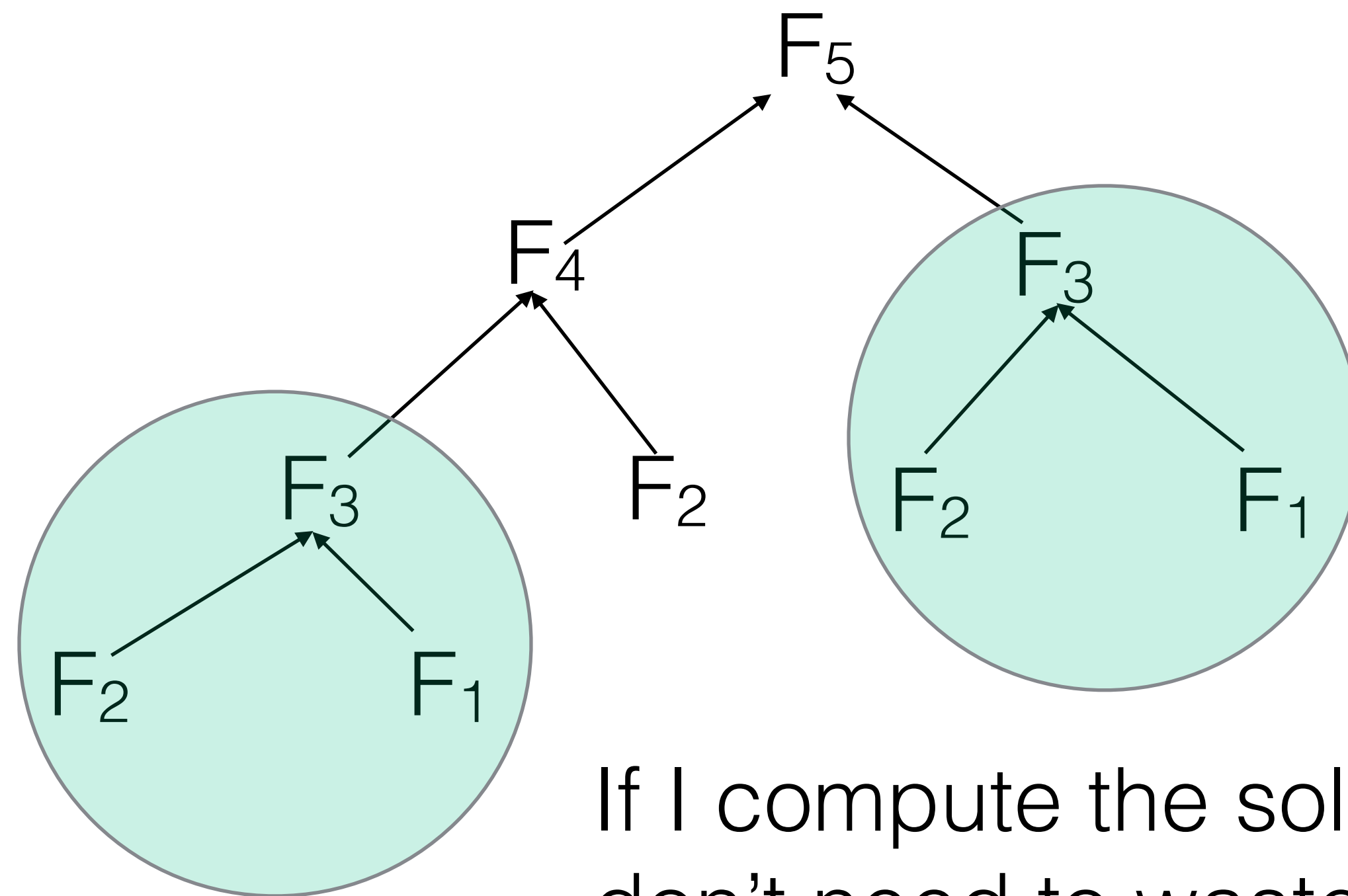
What is the “right order”?

If I compute the solutions in the “right order”, I don't need to waste time re-computing the same values.

Example 1: Fibonacci Sequence

$$F_n = F_{n-1} + F_{n-2} \quad \text{with } F_1 = F_2 = 1$$

How do we do better than $O(\phi^n)$?



What is the “right order”?

$$F_1 \rightarrow F_2 \rightarrow F_3 \rightarrow F_4 \rightarrow F_5 \dots$$

If I compute the solutions in the “right order”, I don't need to waste time re-computing the same values.

Example 1: Fibonacci Sequence

$$F_n = F_{n-1} + F_{n-2} \quad \text{with } F_1 = F_2 = 1$$

How do we do better than $O(\phi^n)$?

Take 2:

```
def fib(n):  
    if n == 1 or n == 2:  
        return 1  
    fm2, fm1 = 1, 1  
    for i in xrange(2, n):  
        fm2, fm1 = fm1, fm2 + fm1  
    return fm1
```

We loop up to n , and perform an addition in each iteration $\rightarrow O(n)$; **much better!** Note: $O(n)$ assumes addition is constant, not true for large enough n .

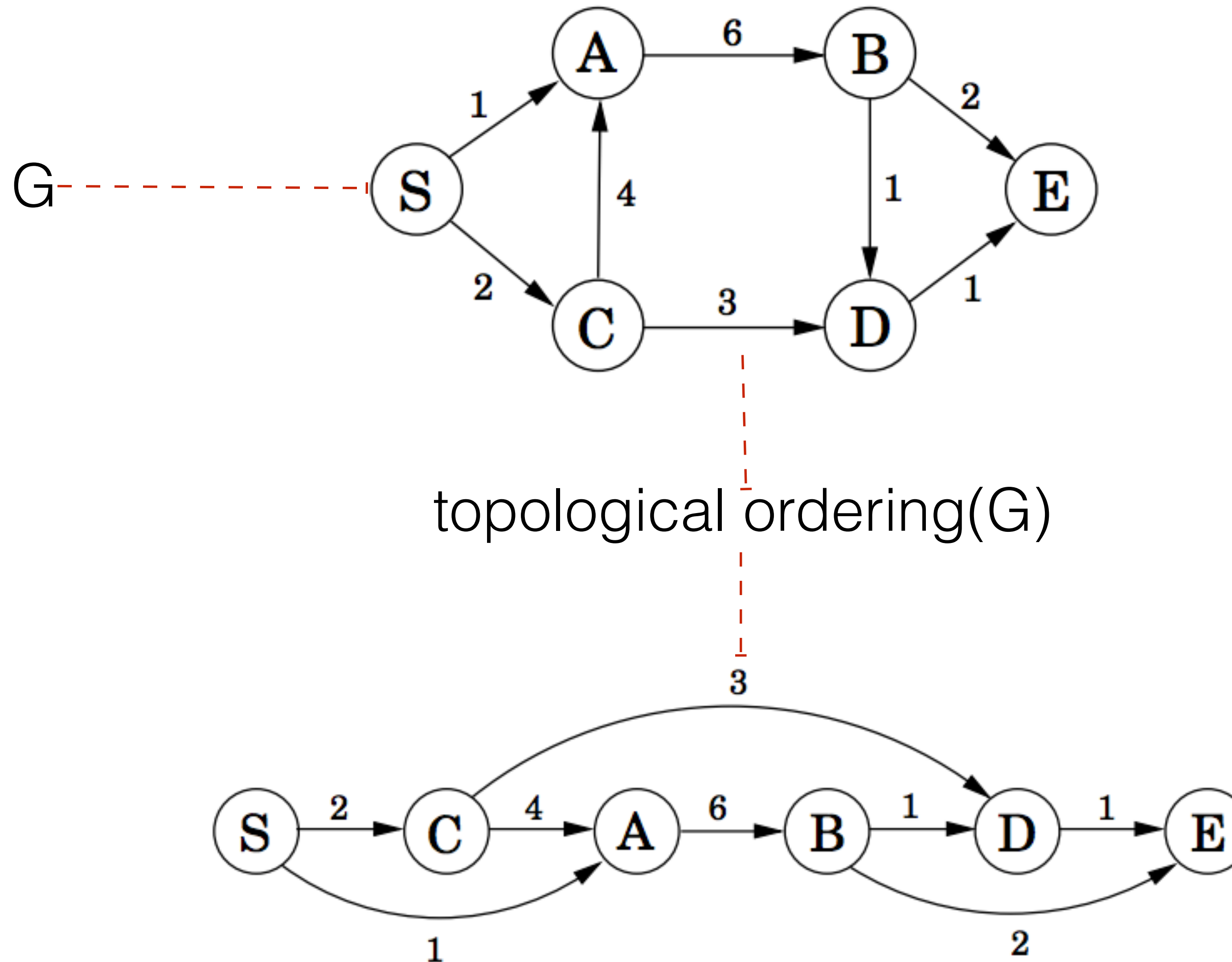
Example 2: Shortest Path in a DAG

Let $G = (V, E)$ be a **d**irected **a**cyclic **g**raph (DAG) with vertex set V and edge set E .

Since G directed and free of cycles, there exists a (at least one) **topological order** of G — an ordering $p(v_1), p(v_2), \dots, p(v_n)$ such that for all $e = (v_i, v_j)$ in E , $p(v_i) < p(v_j)$

In other words, we can label the nodes of G such that all edges point from a vertex with a smaller label to a vertex with a larger label.

Example 2: Shortest Path in a DAG



Obtaining a topological ordering

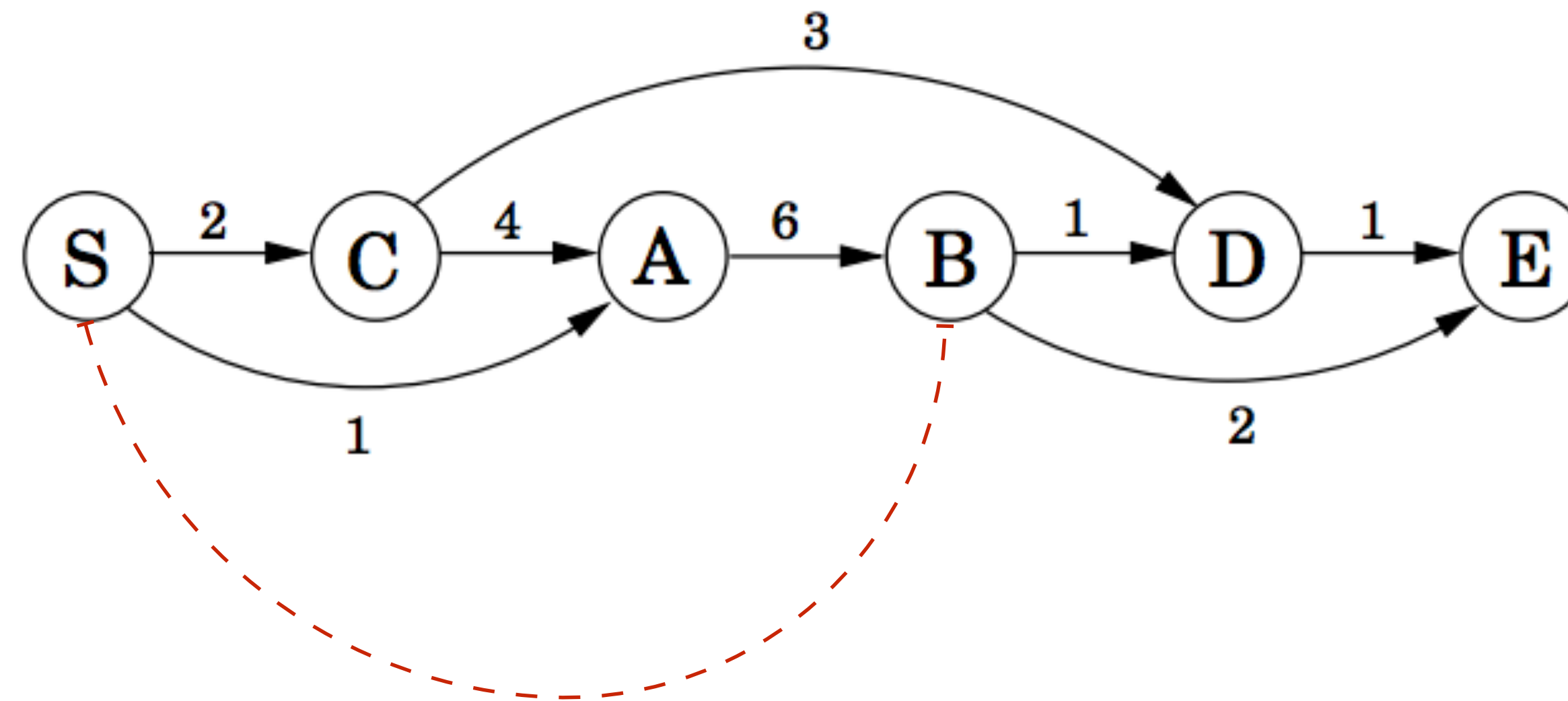
Kahn's algorithm

Builds up a valid topo order node-by-node

```
L ← Empty list that will contain the sorted elements
S ← Set of all nodes with no incoming edges
while S is non-empty do
    remove a node n from S
    add n to tail of L
    for each node m with an edge e from n to m do
        remove edge e from the graph
        if m has no other incoming edges then
            insert m into S
if graph has edges then
    return error (graph has at least one cycle)
else
    return L (a topologically sorted order)
```

$O(|V| + |E|)$; why?

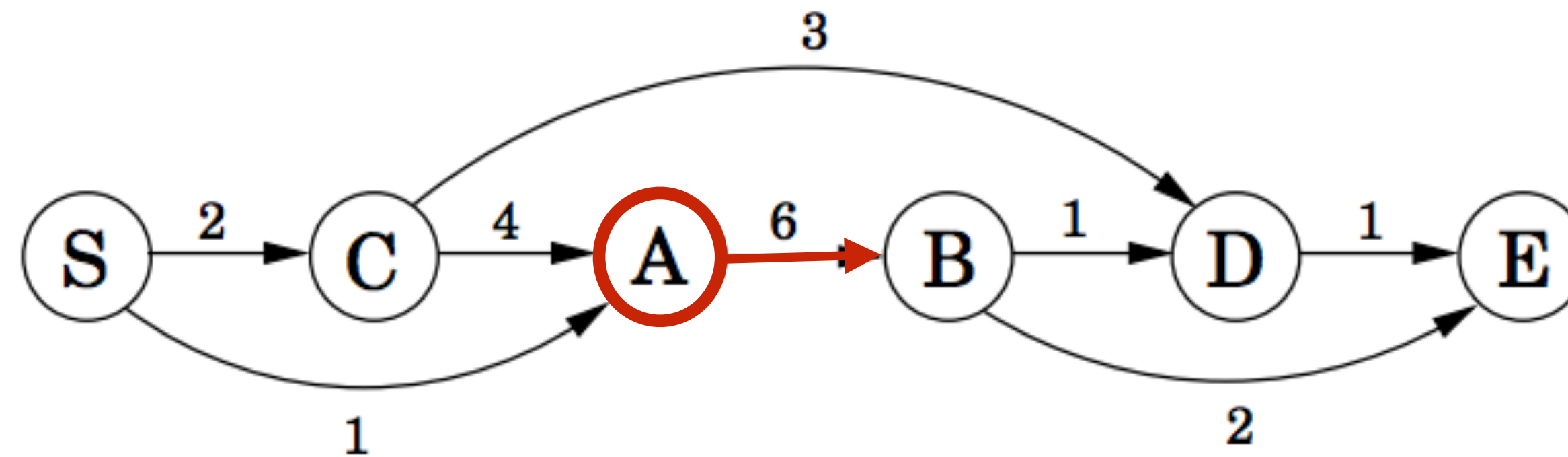
Example 2: Shortest Path in a DAG



What's the distance from S to B — $d(S,B)$?

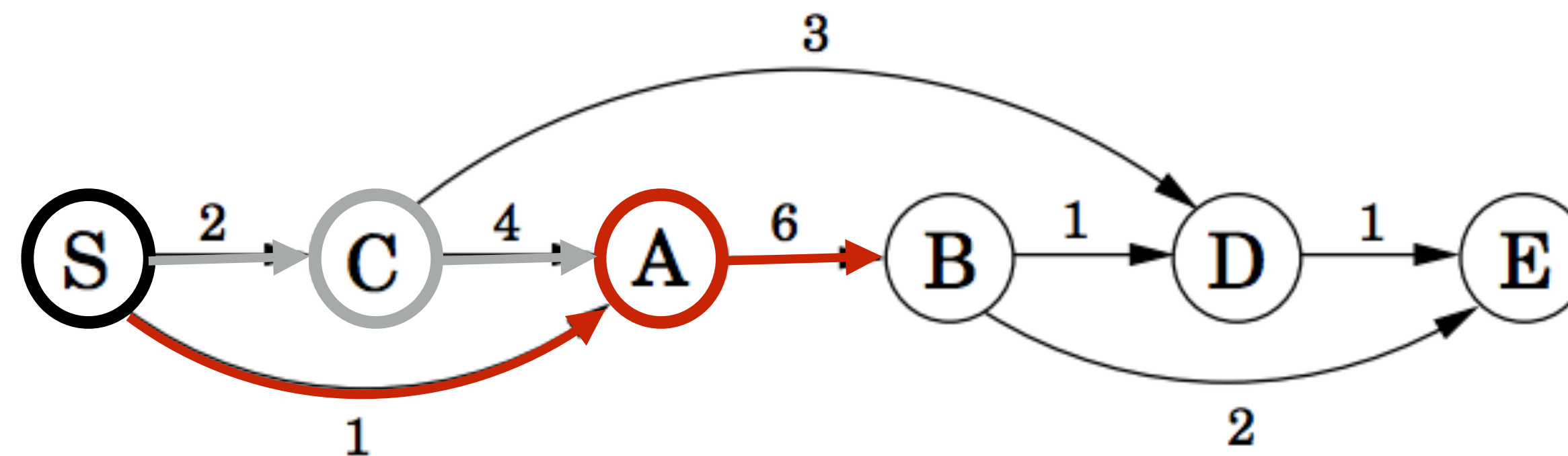
Example 2: Shortest Path in a DAG

First, I **must** go through A, so it's at least $d(S,A) + 6$



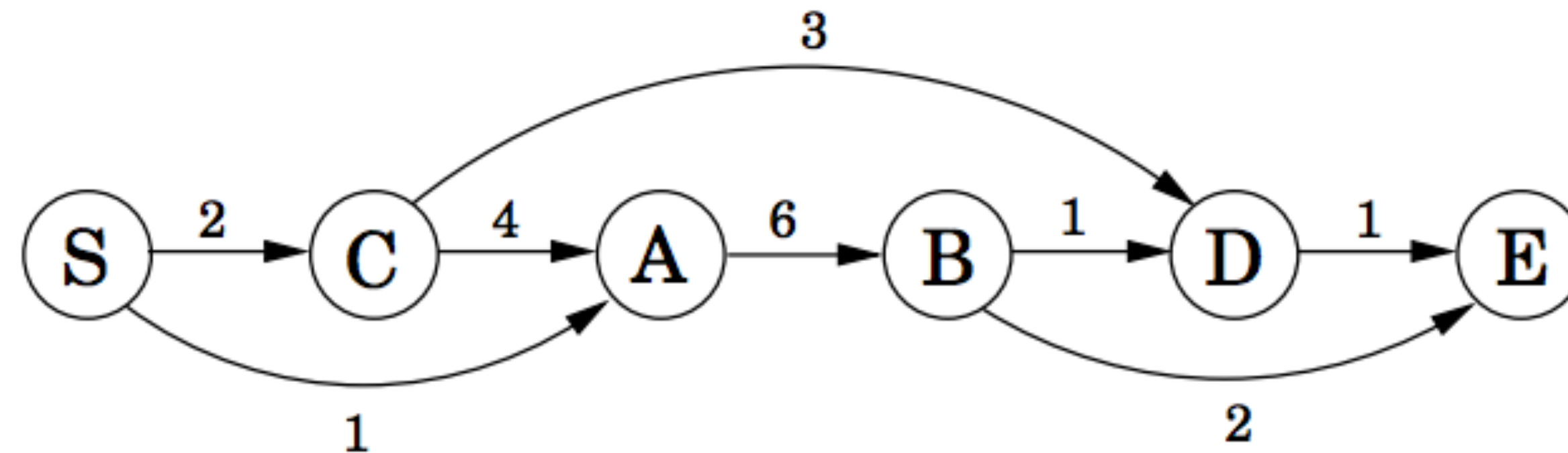
Example 2: Shortest Path in a DAG

Then, there are 2 ways of getting to A — we choose the shortest.



Example 2: Shortest Path in a DAG

In general, $d(S, X)$ is the minimum value of $d(S, Y) + d(Y, X)$ for all Y that precede X and are connected by an edge



$$d(S, X) = \min_{Y \mid (Y, X) \in E} \{d(S, Y) + d(Y, X)\}$$

This becomes the DP recurrence for our problem

Example 2: Shortest Path in a DAG

The problem is solved efficiently by the following algorithm

```
initialize all  $\text{dist}(\cdot)$  values to  $\infty$   
 $\text{dist}(s) = 0$   
for each  $v \in V \setminus \{s\}$ , in linearized order:  
     $\text{dist}(v) = \min_{(u,v) \in E} \{\text{dist}(u) + l(u,v)\}$ 
```

Algorithm for Computing Edit Distance

Consider the last characters of each string:

$$\begin{aligned}a &= a_1a_2a_3a_4\dots a_m \\ b &= b_1b_2b_3b_4\dots b_n\end{aligned}$$

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:

$$\begin{aligned}a &= a_1a_2a_3a_4\dots a_m \\ b &= b_1b_2b_3b_4\dots b_n\end{aligned}$$

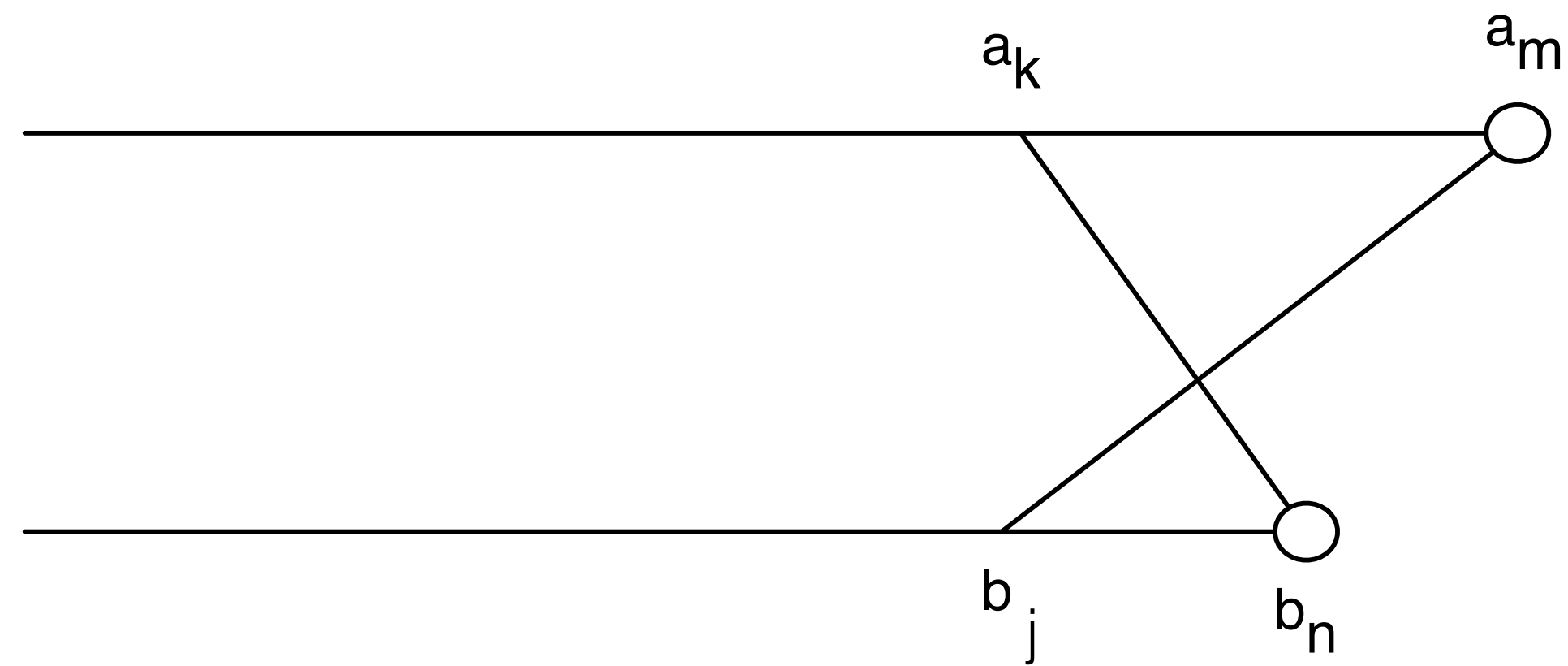
One of these possibilities must hold:

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:

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

↑
Cost of the optimal alignment between $a_1...a_i$ and $b_1...b_j$

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

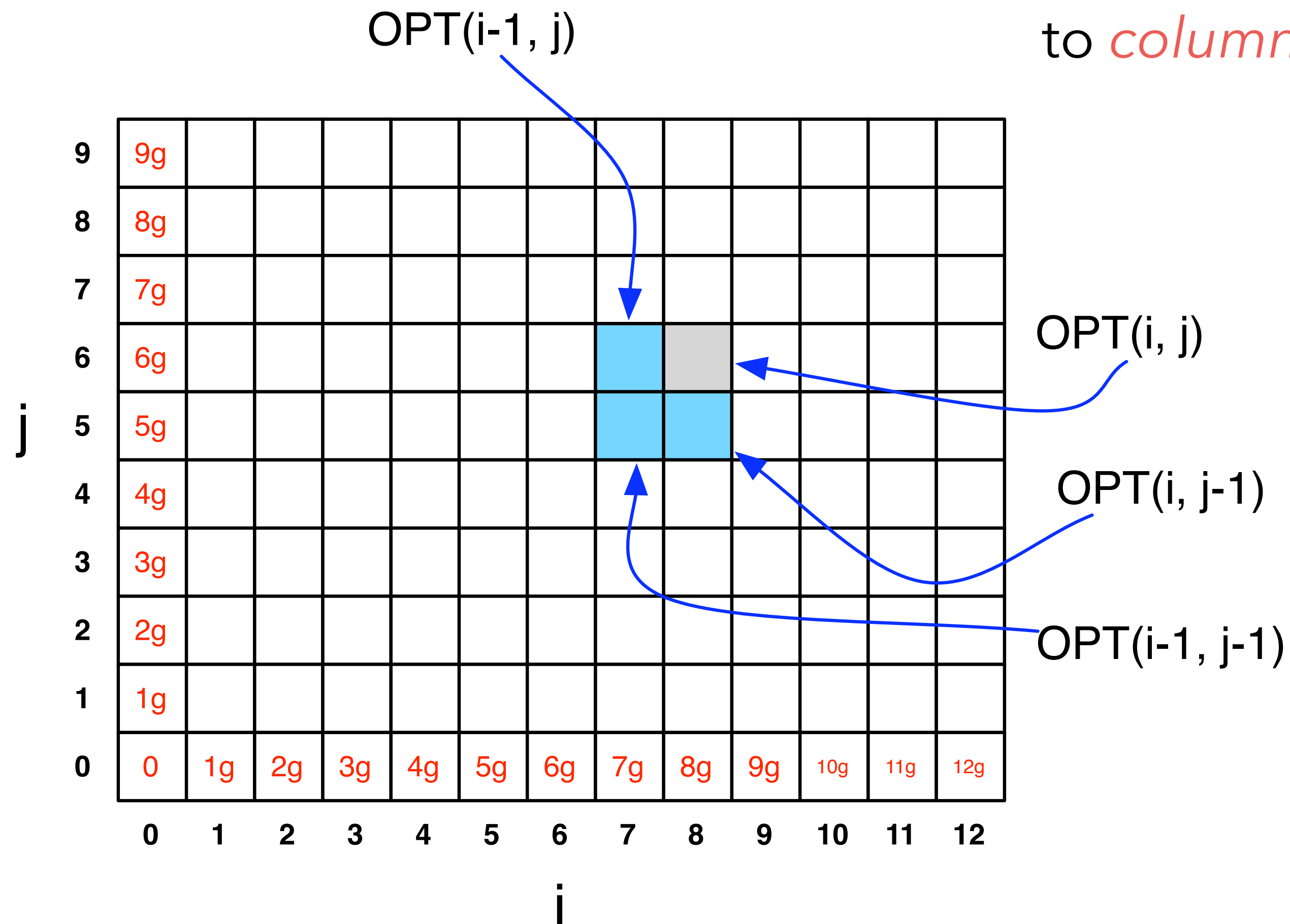
(Aligning i characters to 0 characters must use i gaps.)

Computing $OPT(i,j)$ Efficiently

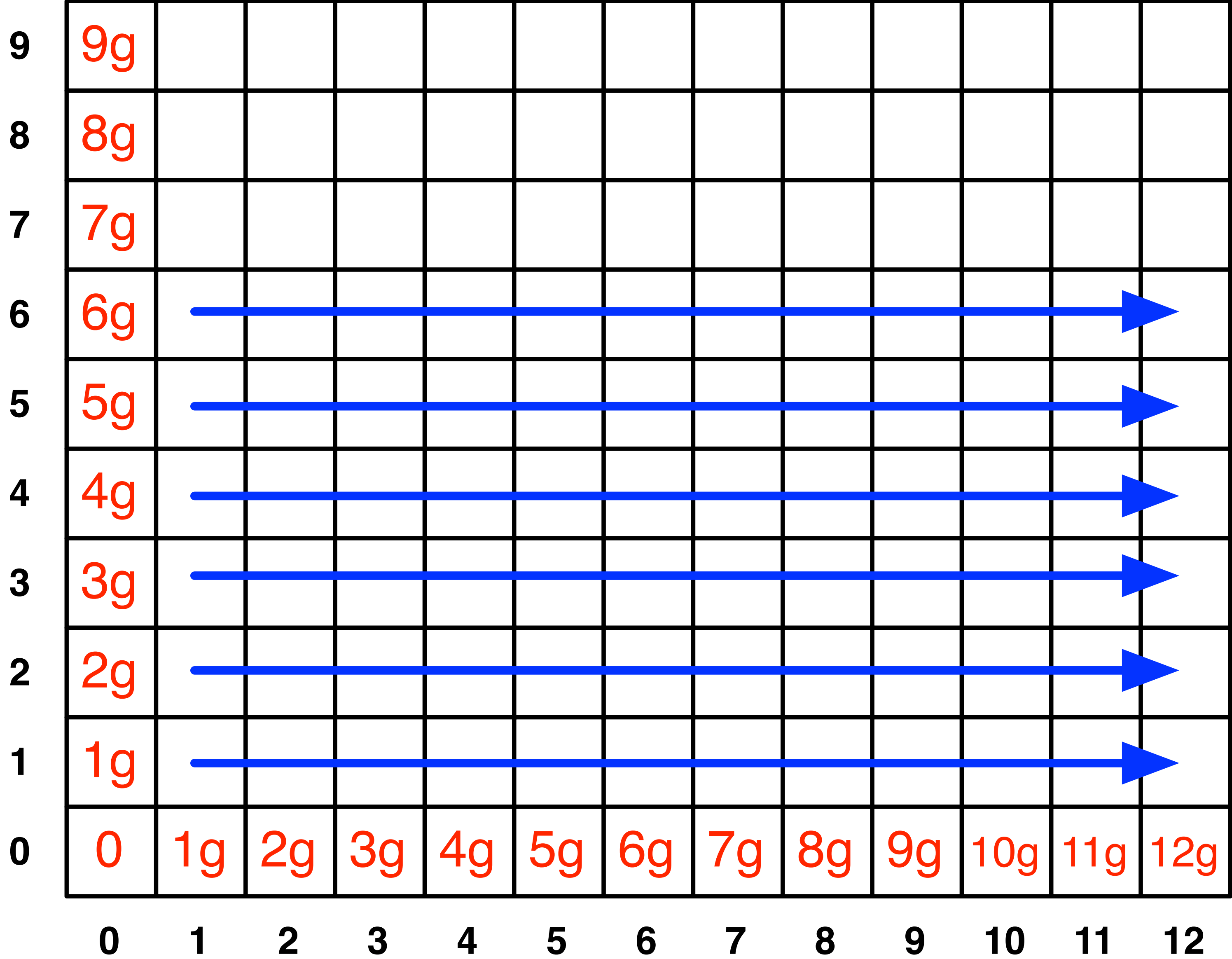
We're ultimately interested in $OPT(n,m)$, but we will compute all other $OPT(i,j)$ ($i \leq n, j \leq m$) on the way to computing $OPT(n,m)$.

Store those values in a 2D array:

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



Filling in the 2D Array



*

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?

$\text{OPT}(n,m)$ 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.

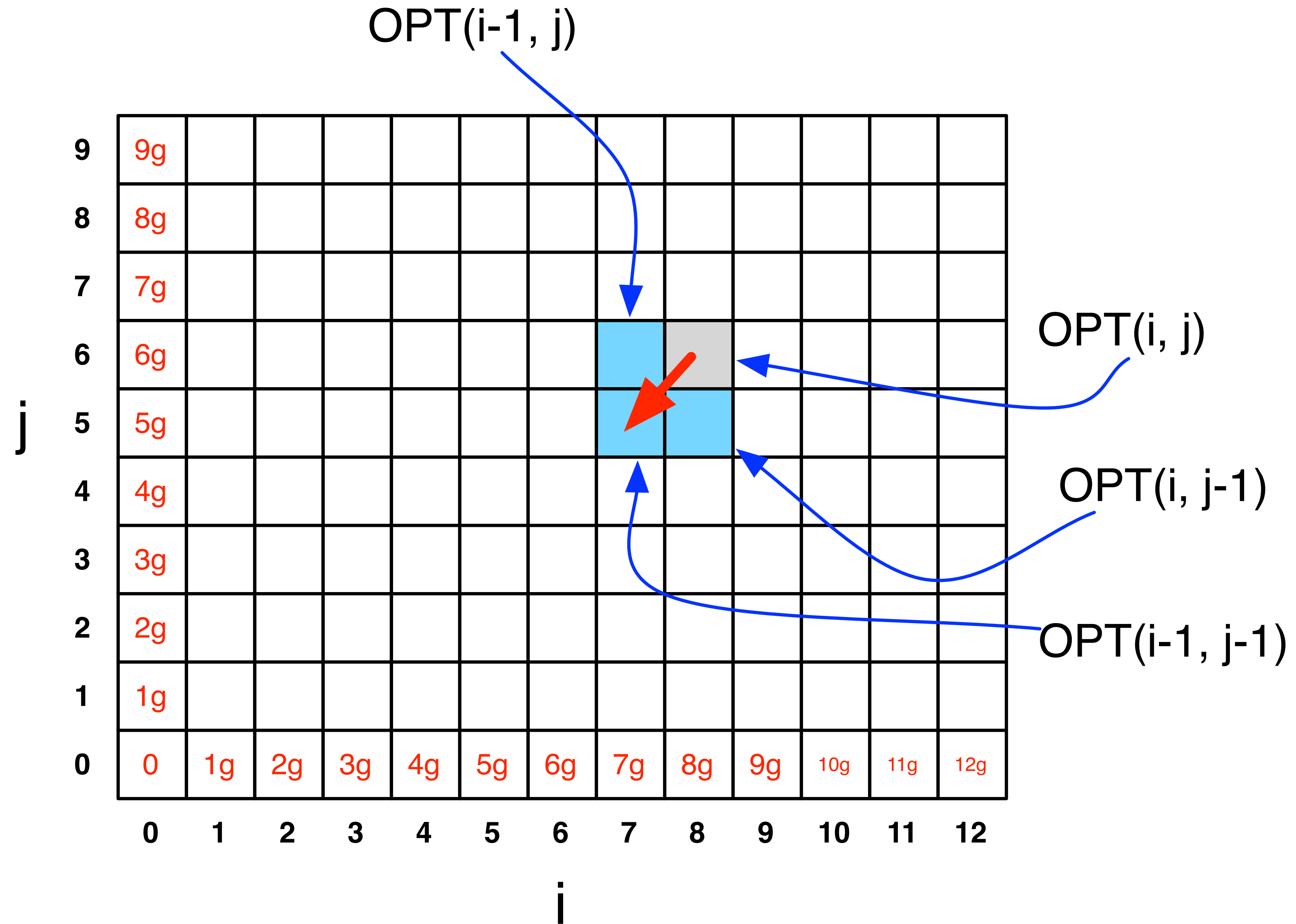
Running Time

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.

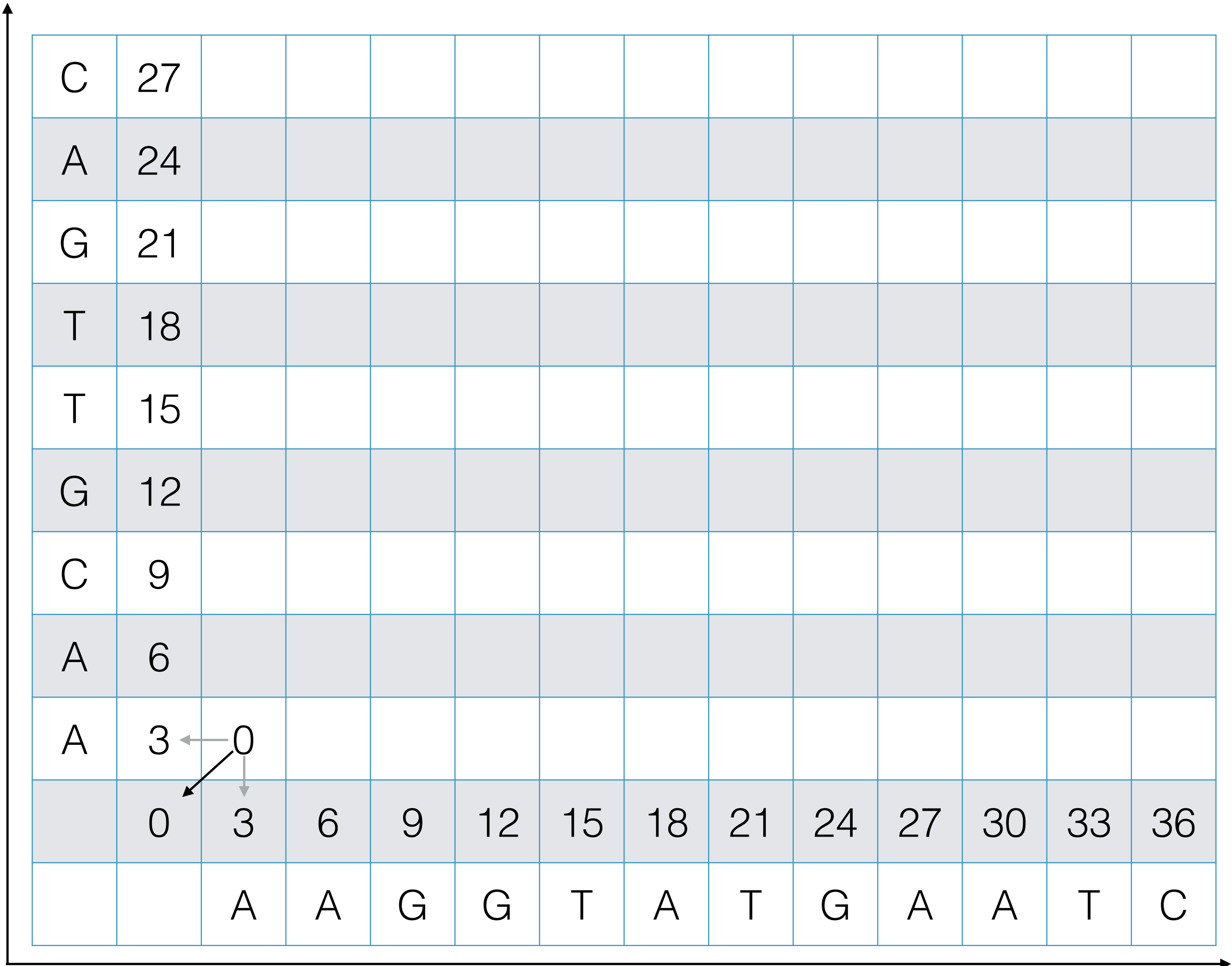
Total running time is $O(mn)$.

Finding the actual alignment



gap cost = 3
mismatch cost = 1

Example



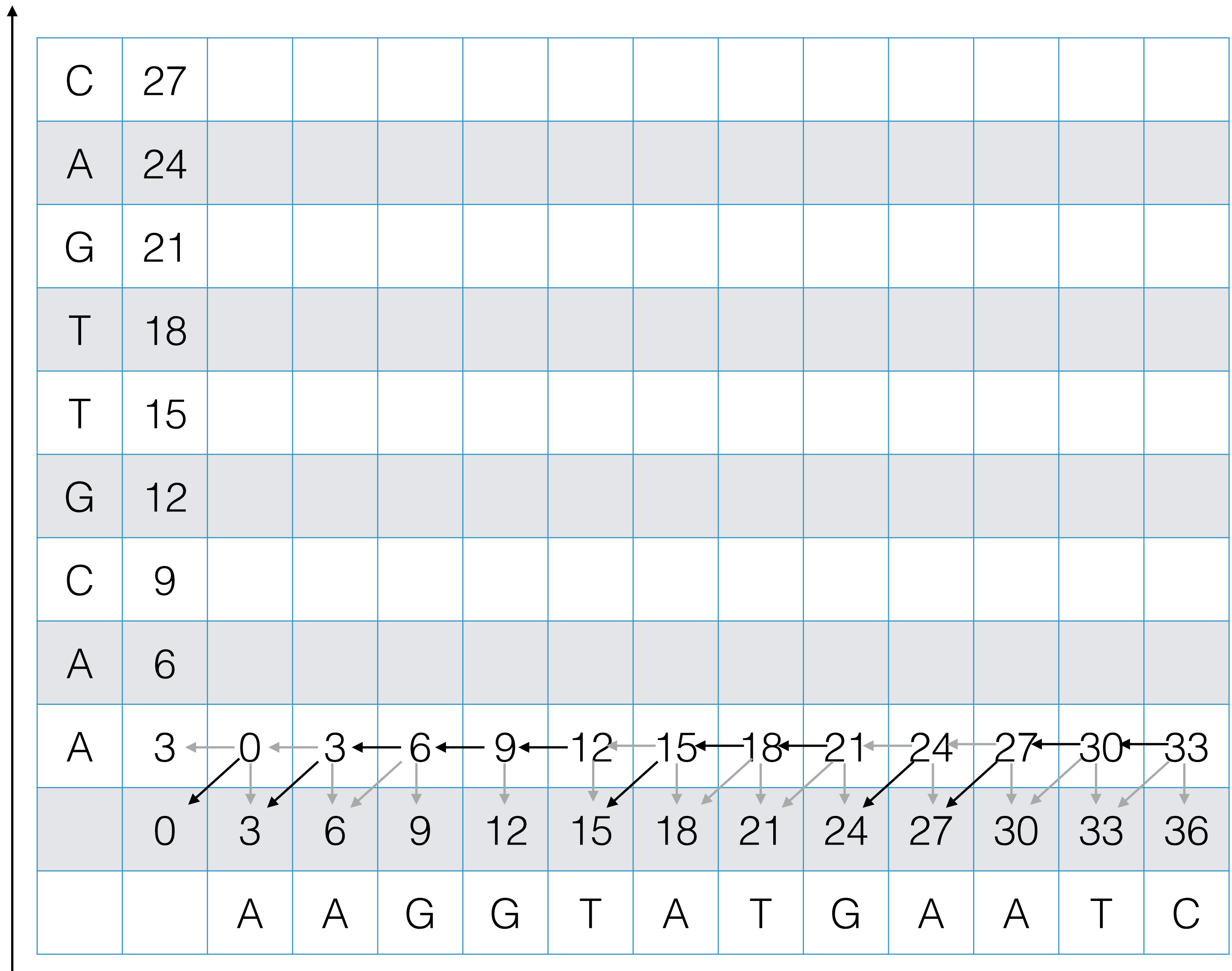
Example

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

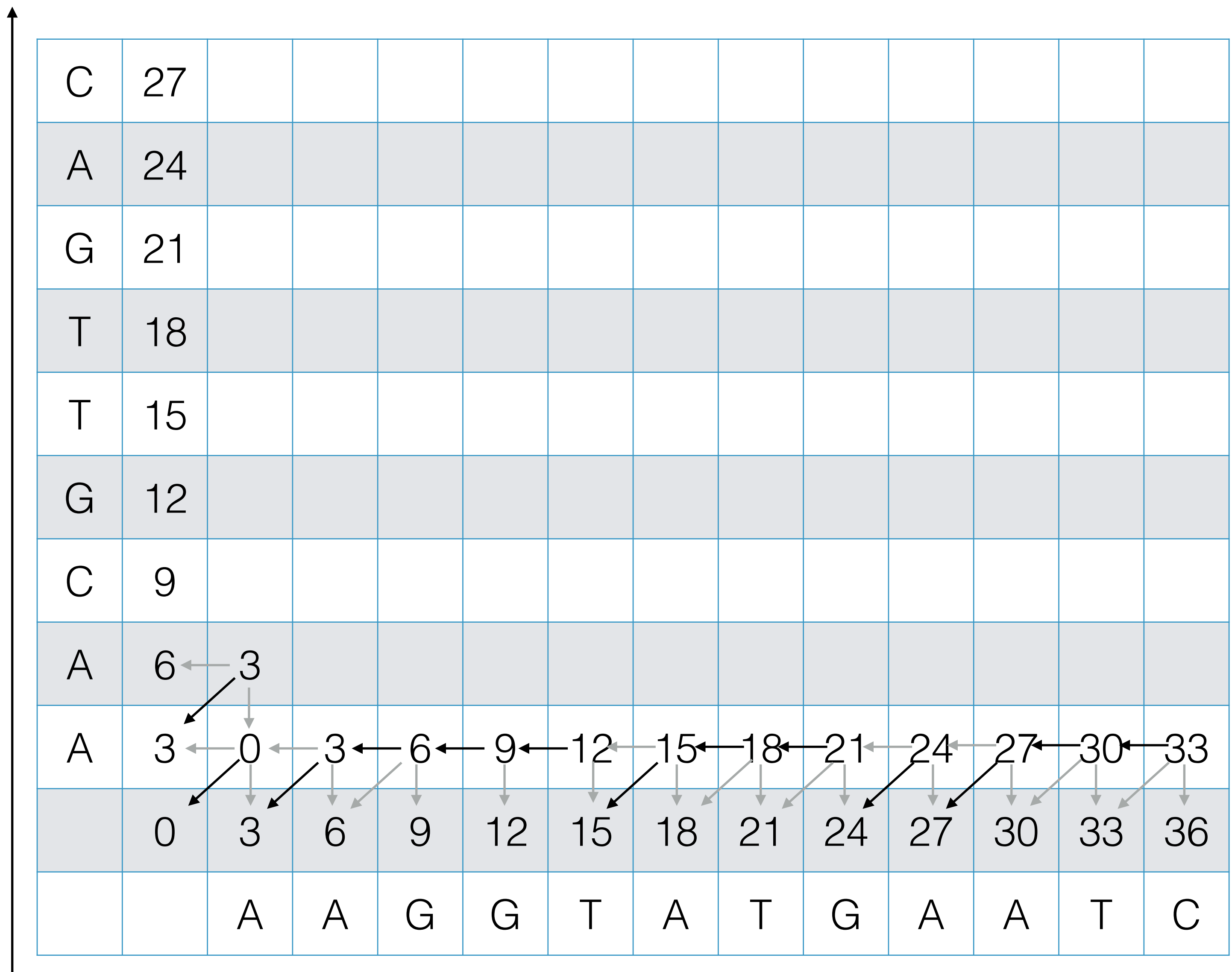
Example

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

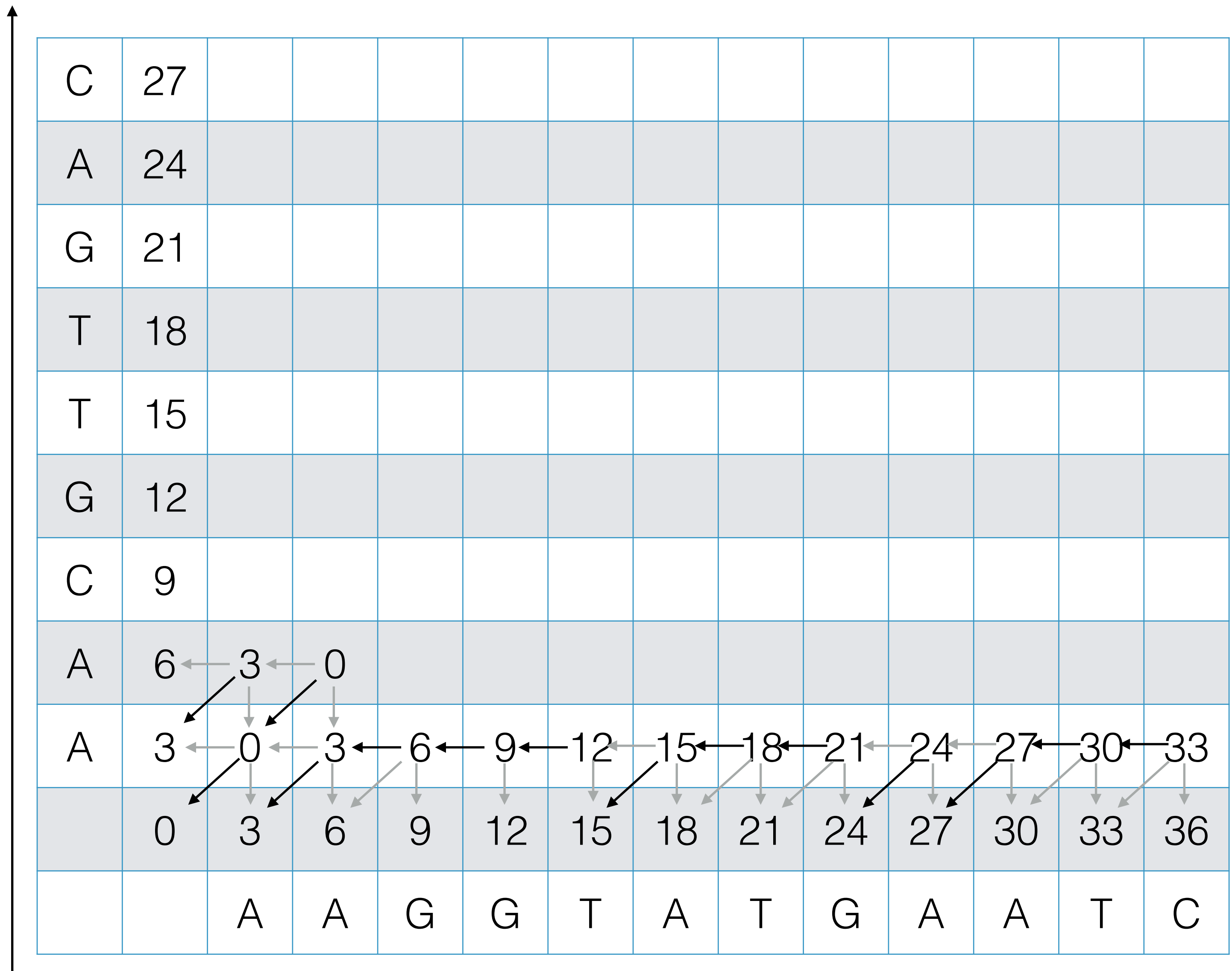
Example



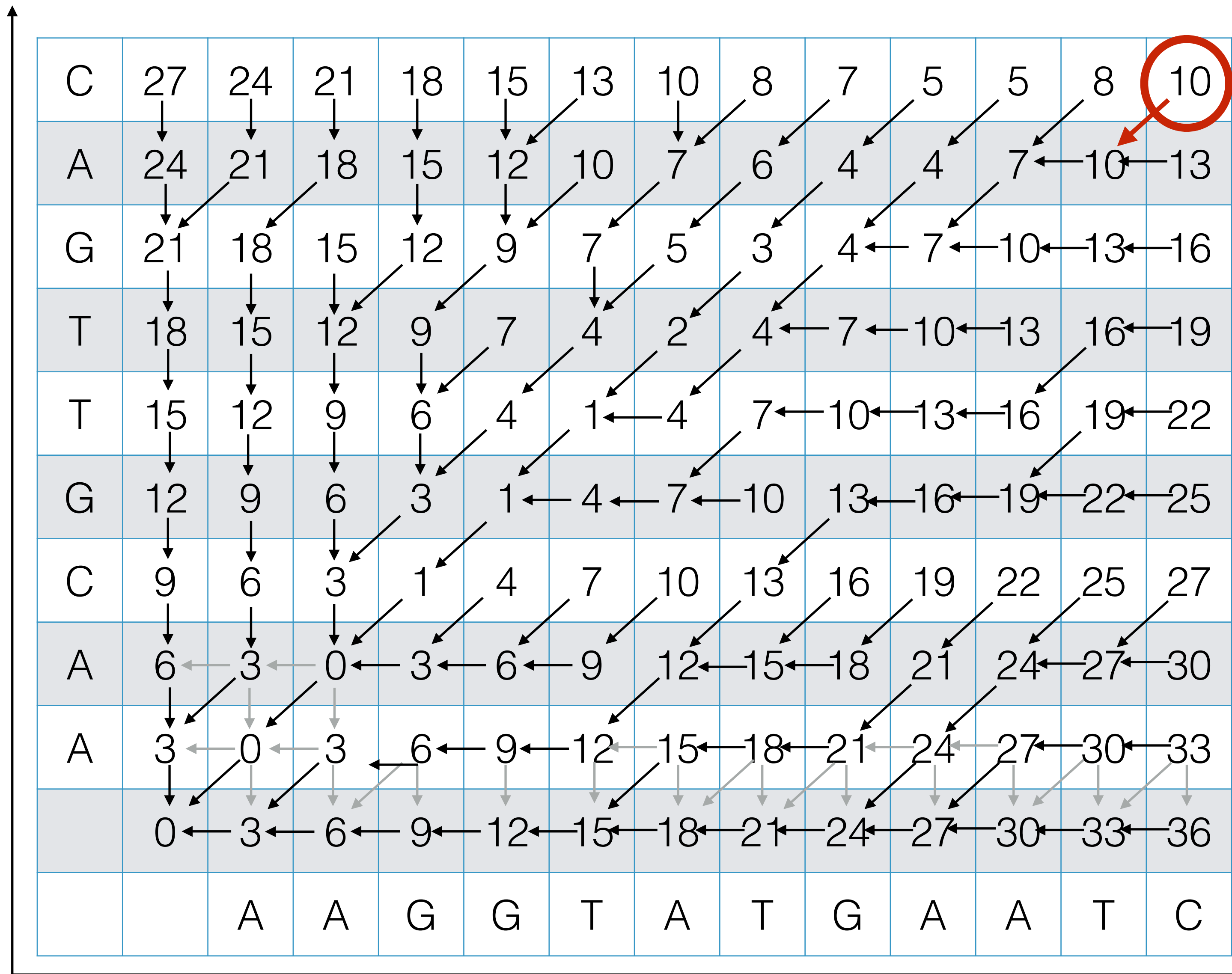
Example



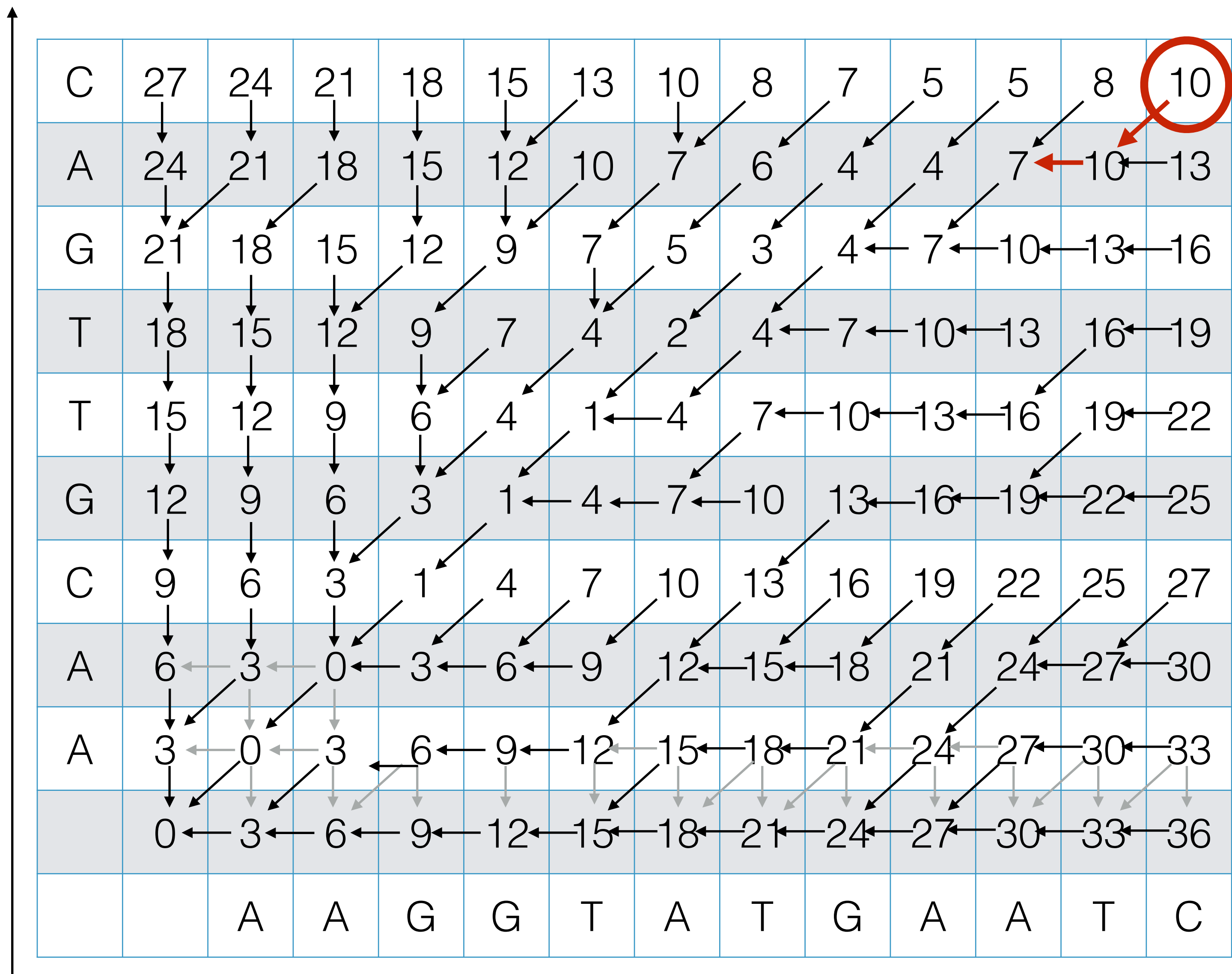
Example



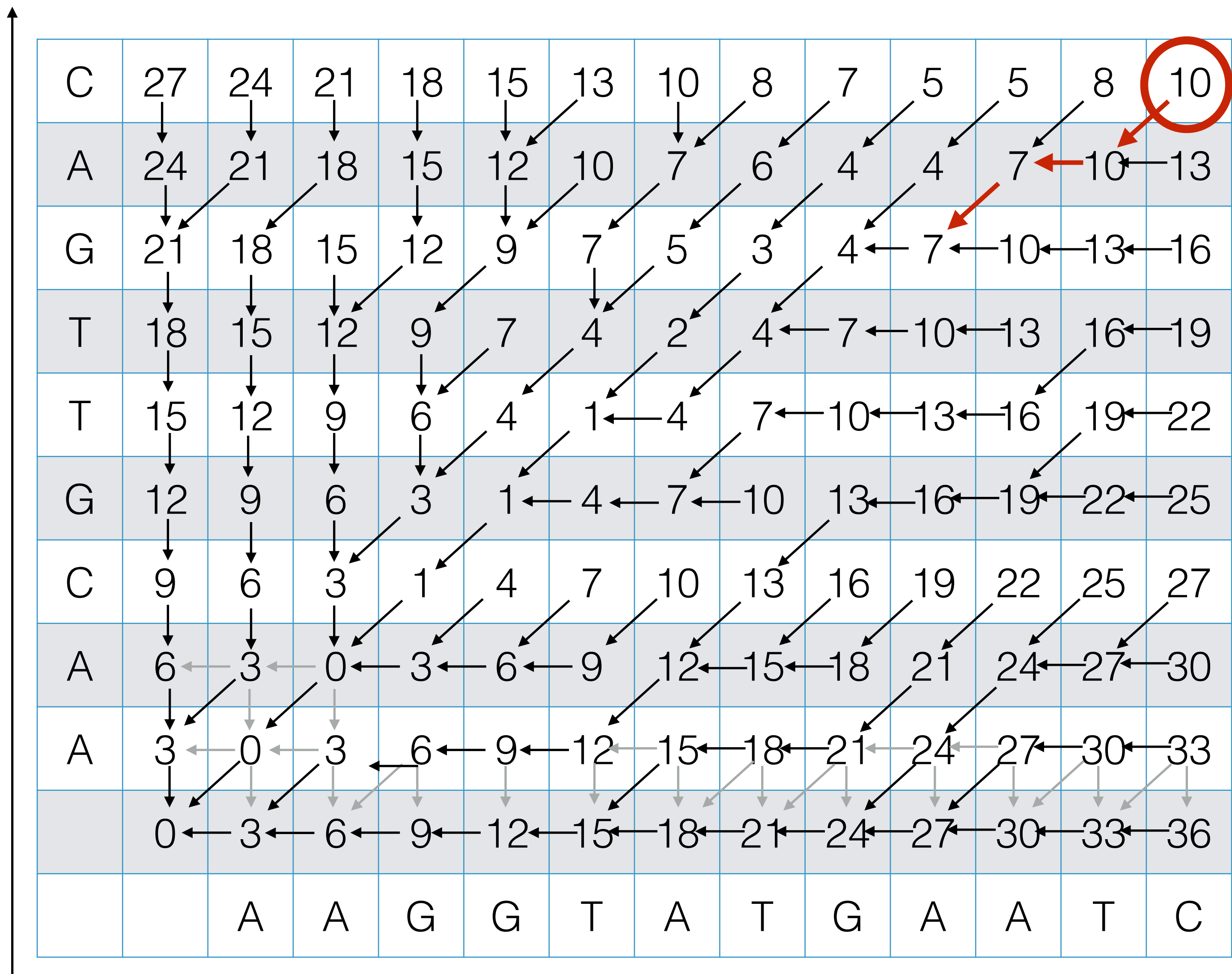
Example



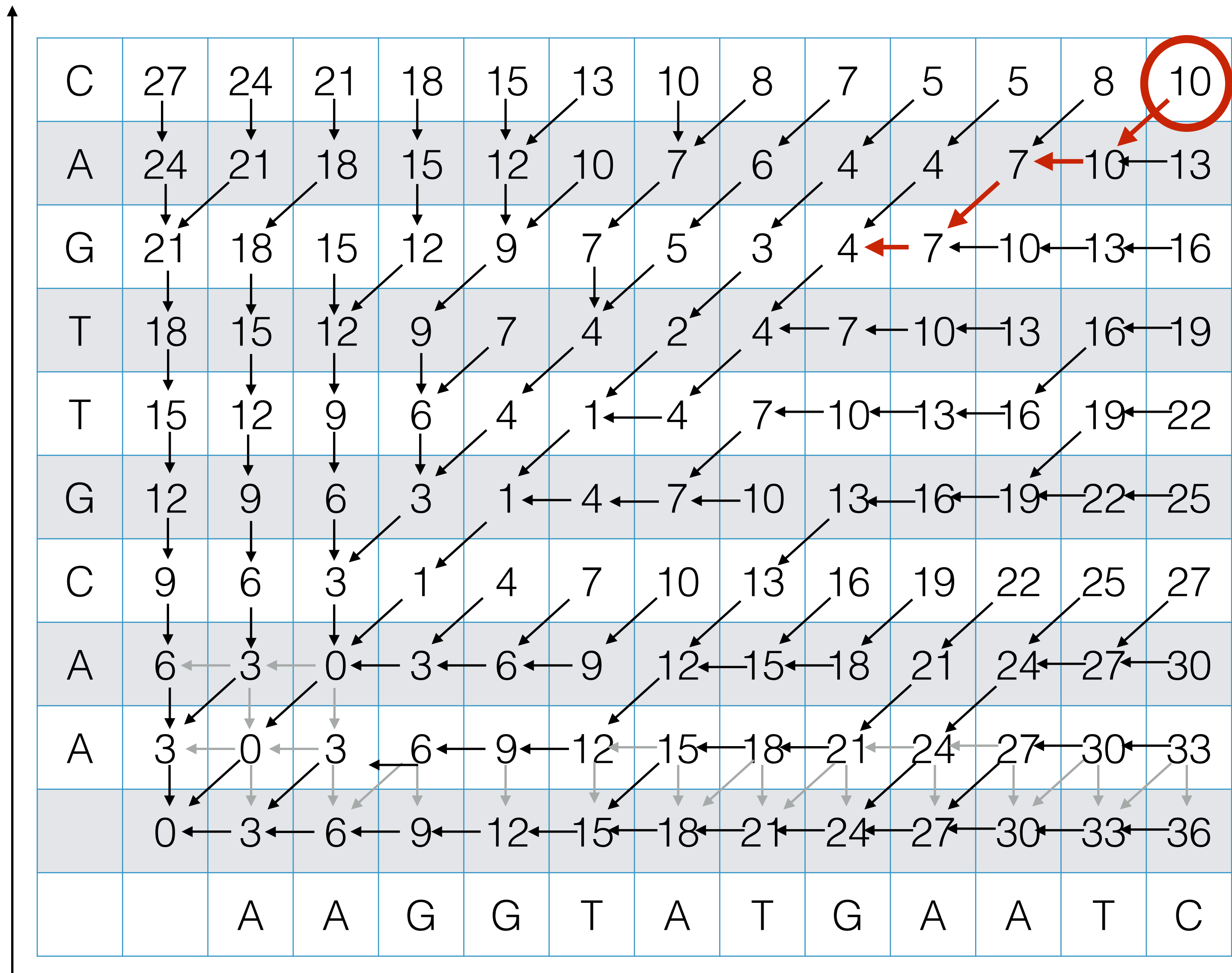
Example



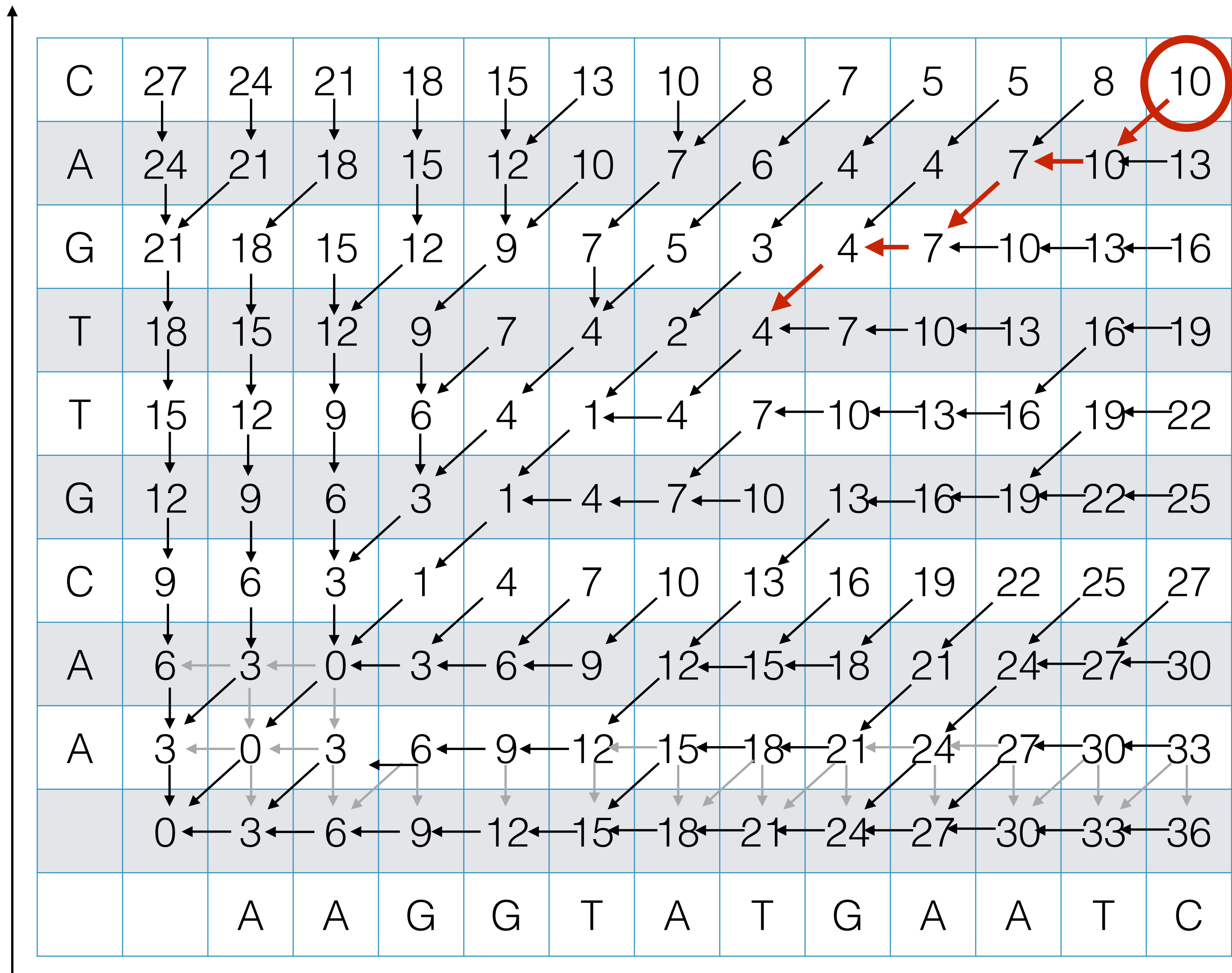
Example



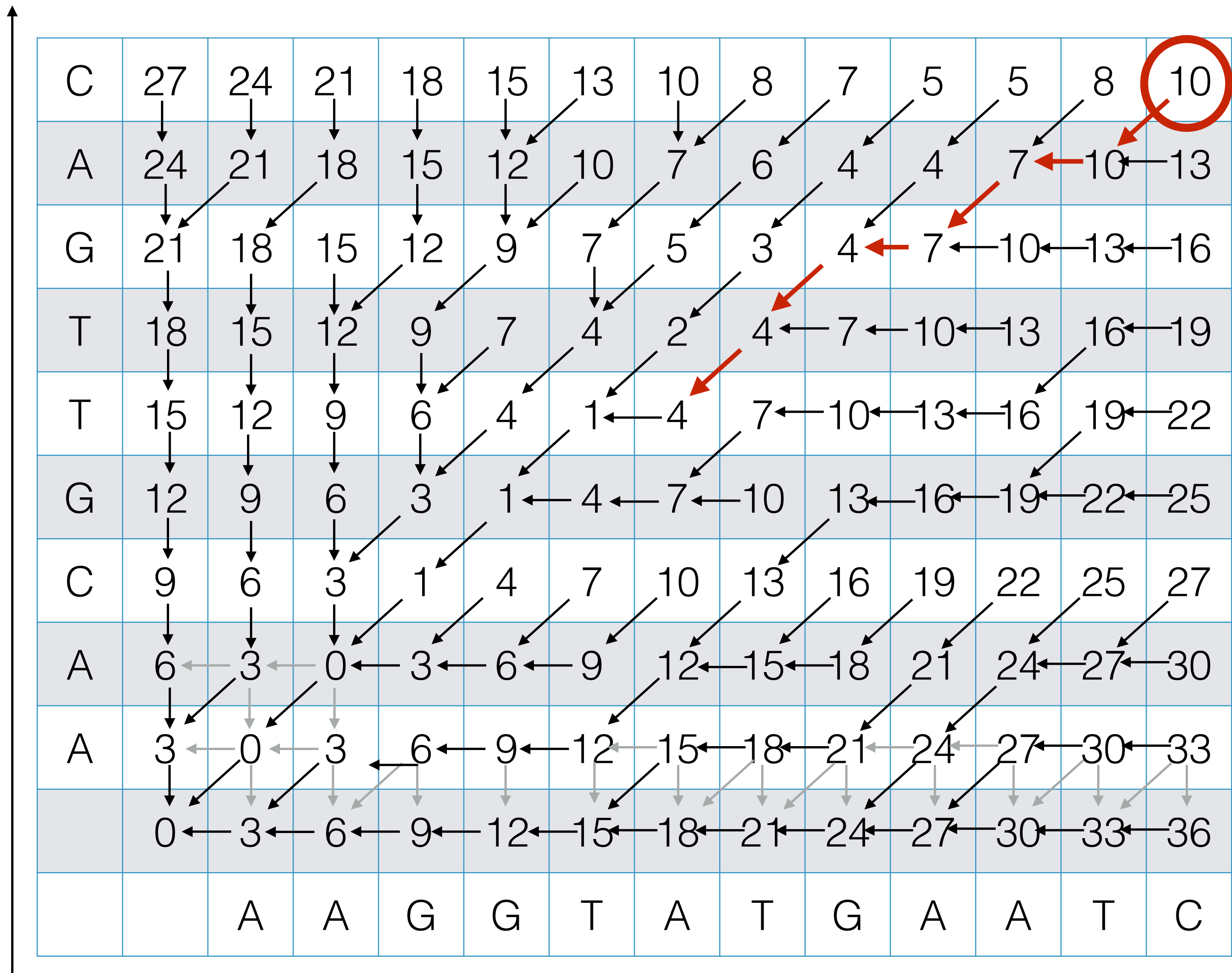
Example



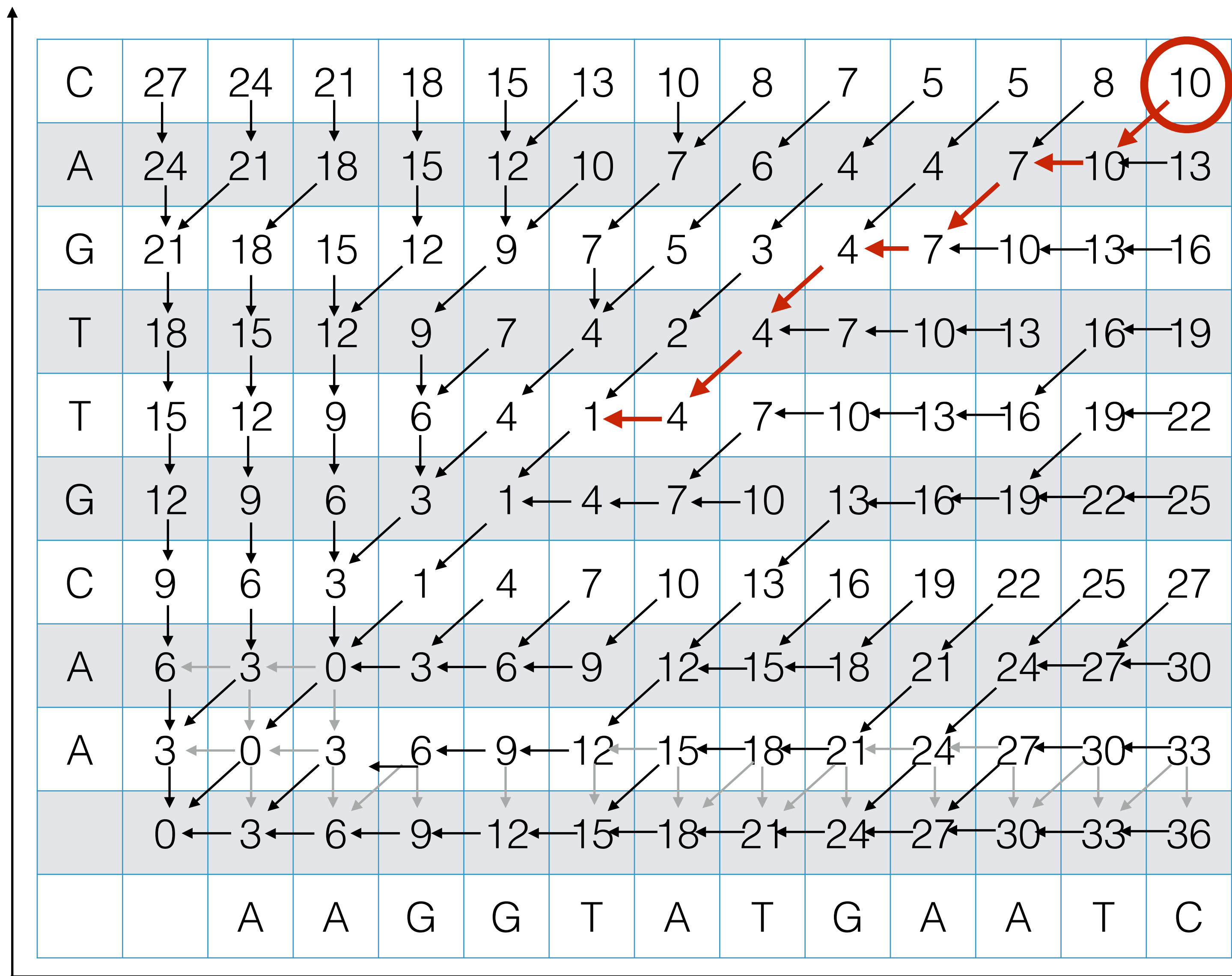
Example



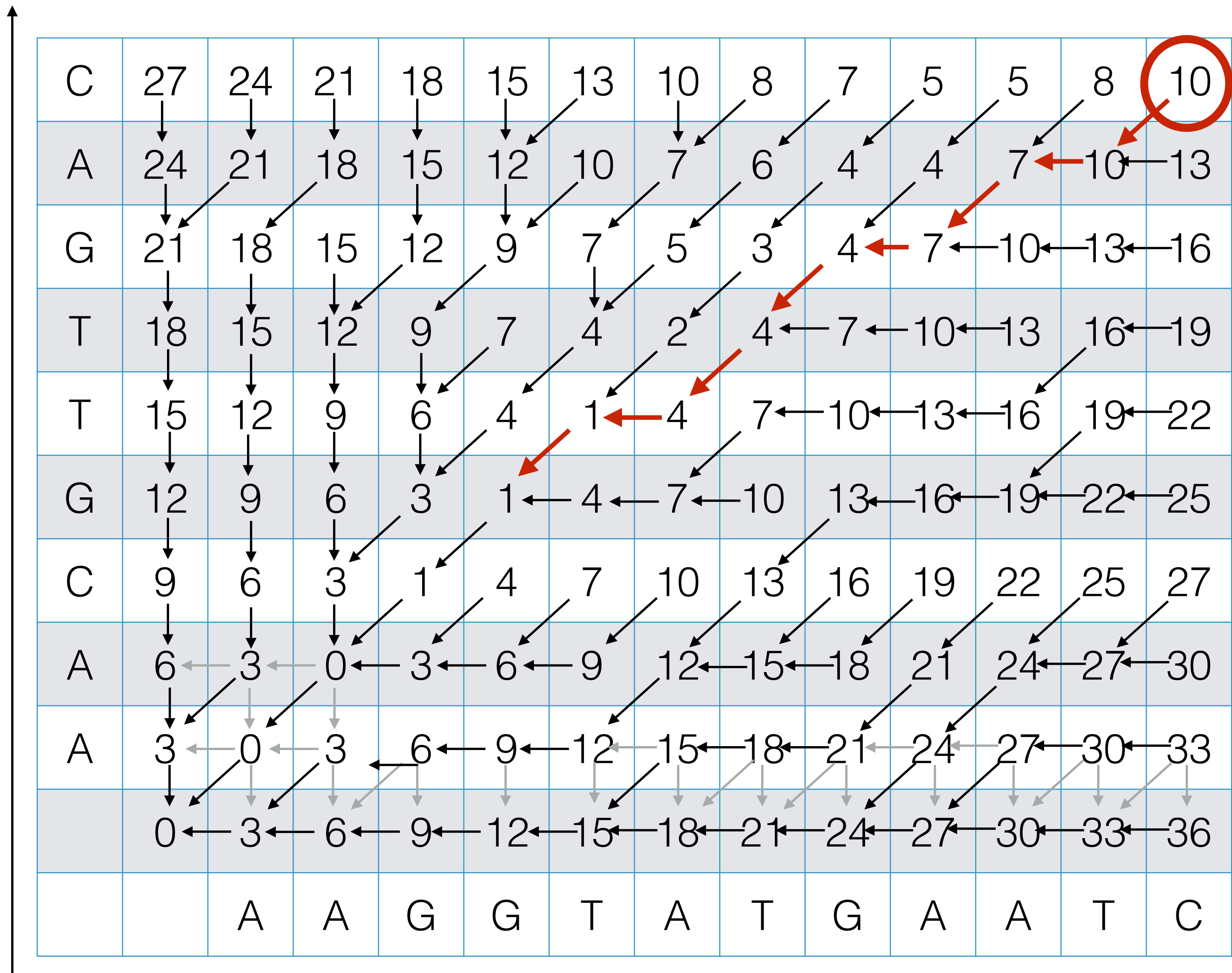
Example



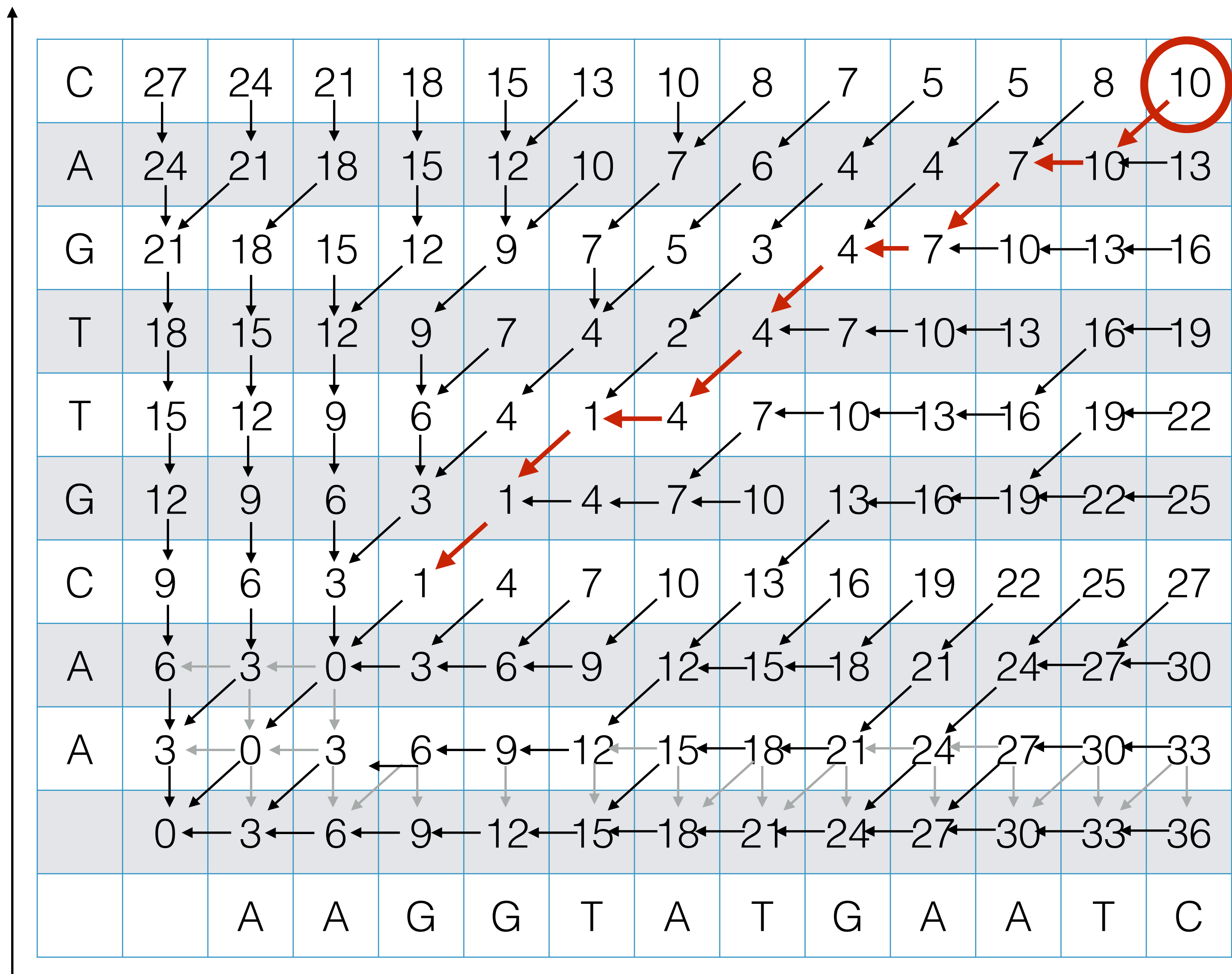
Example



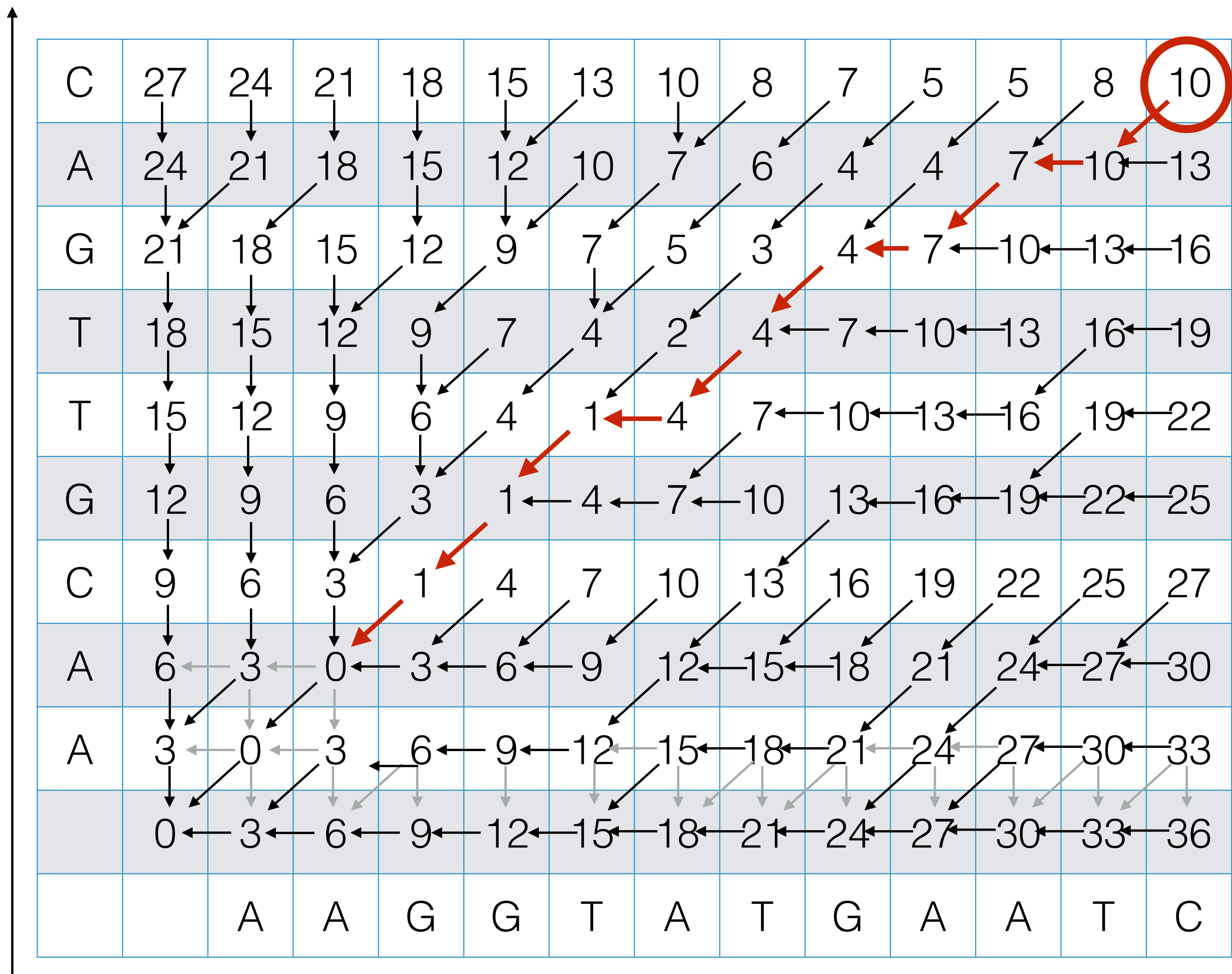
Example



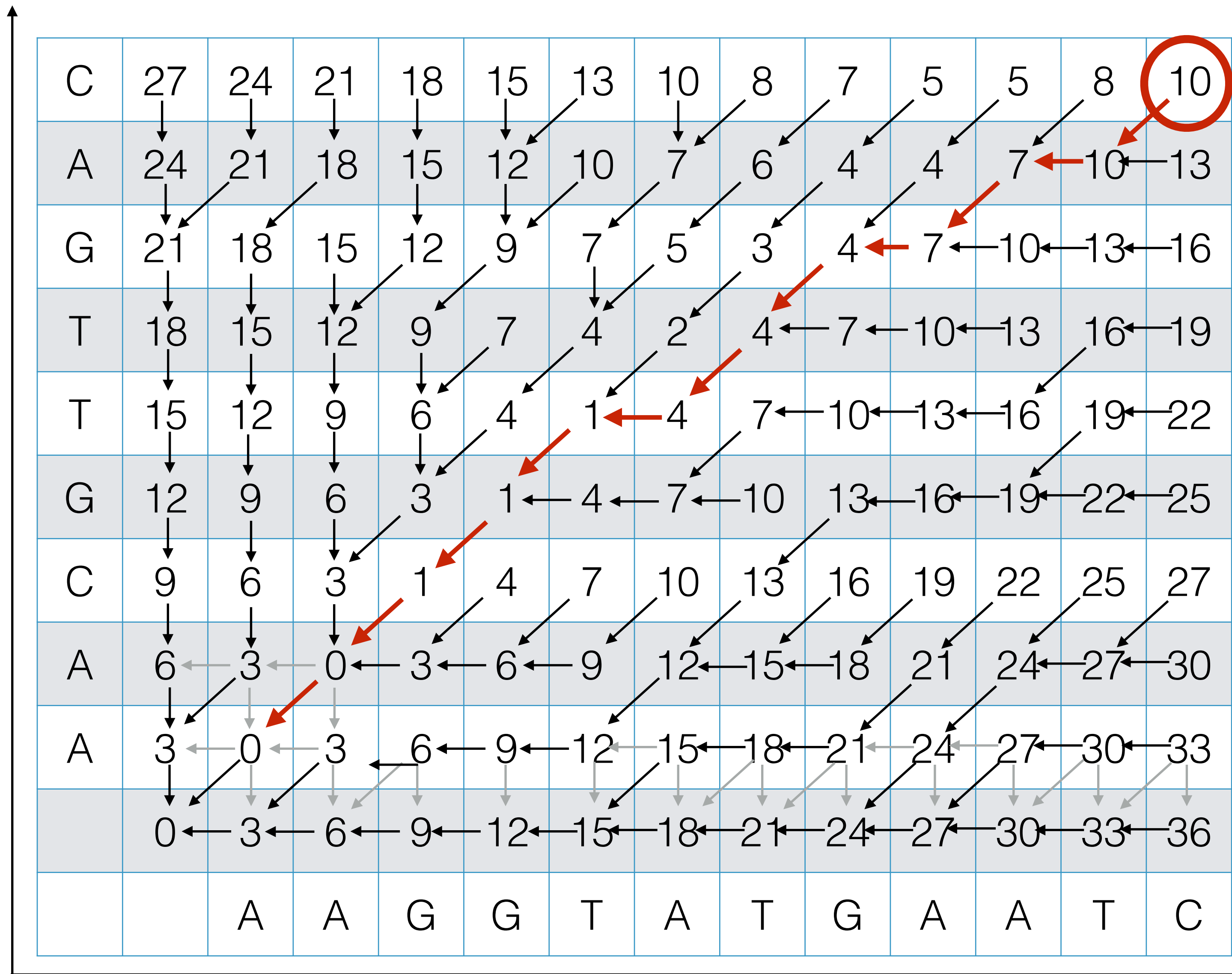
Example



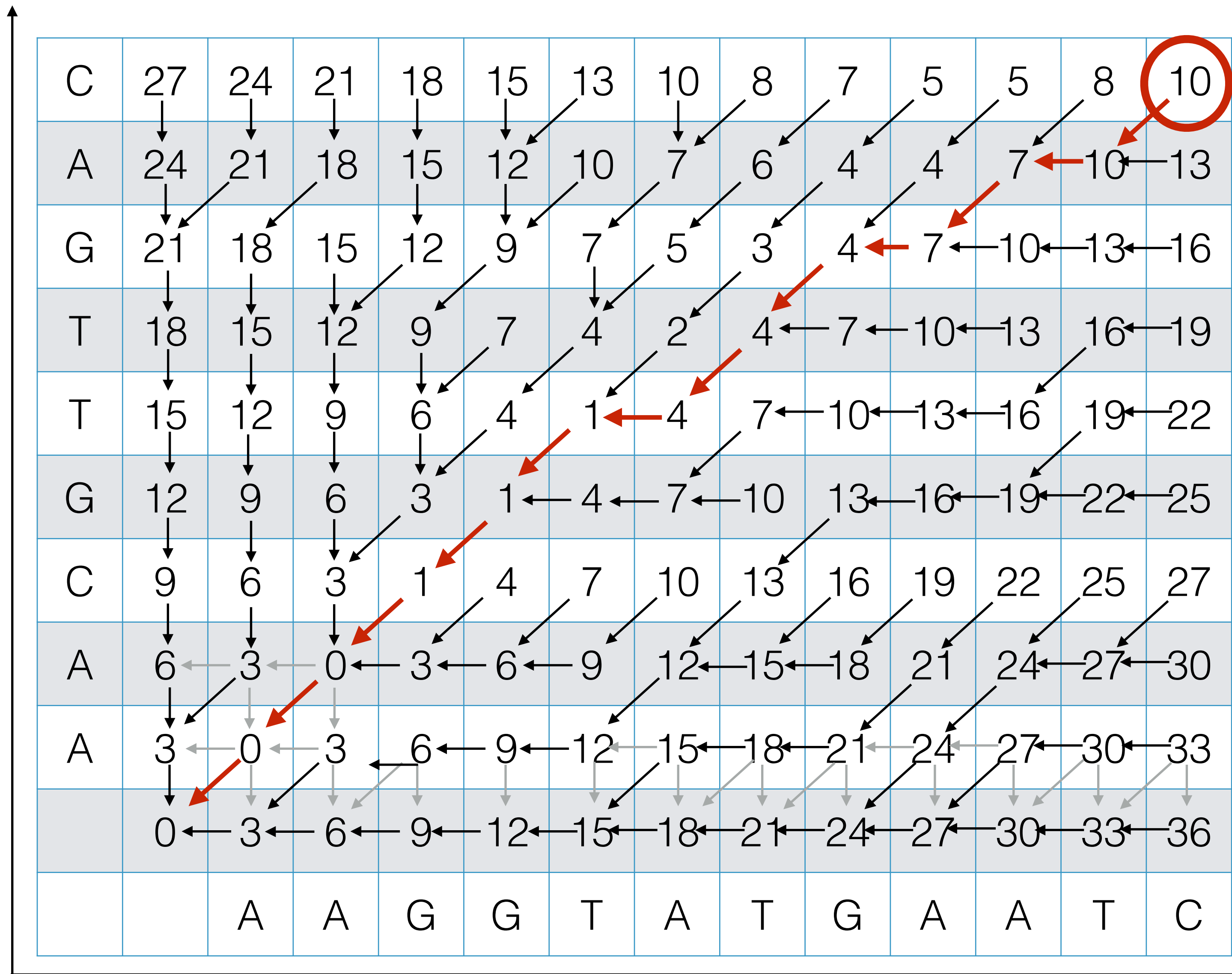
Example



Example



Example



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.

Recap: Dynamic Programming

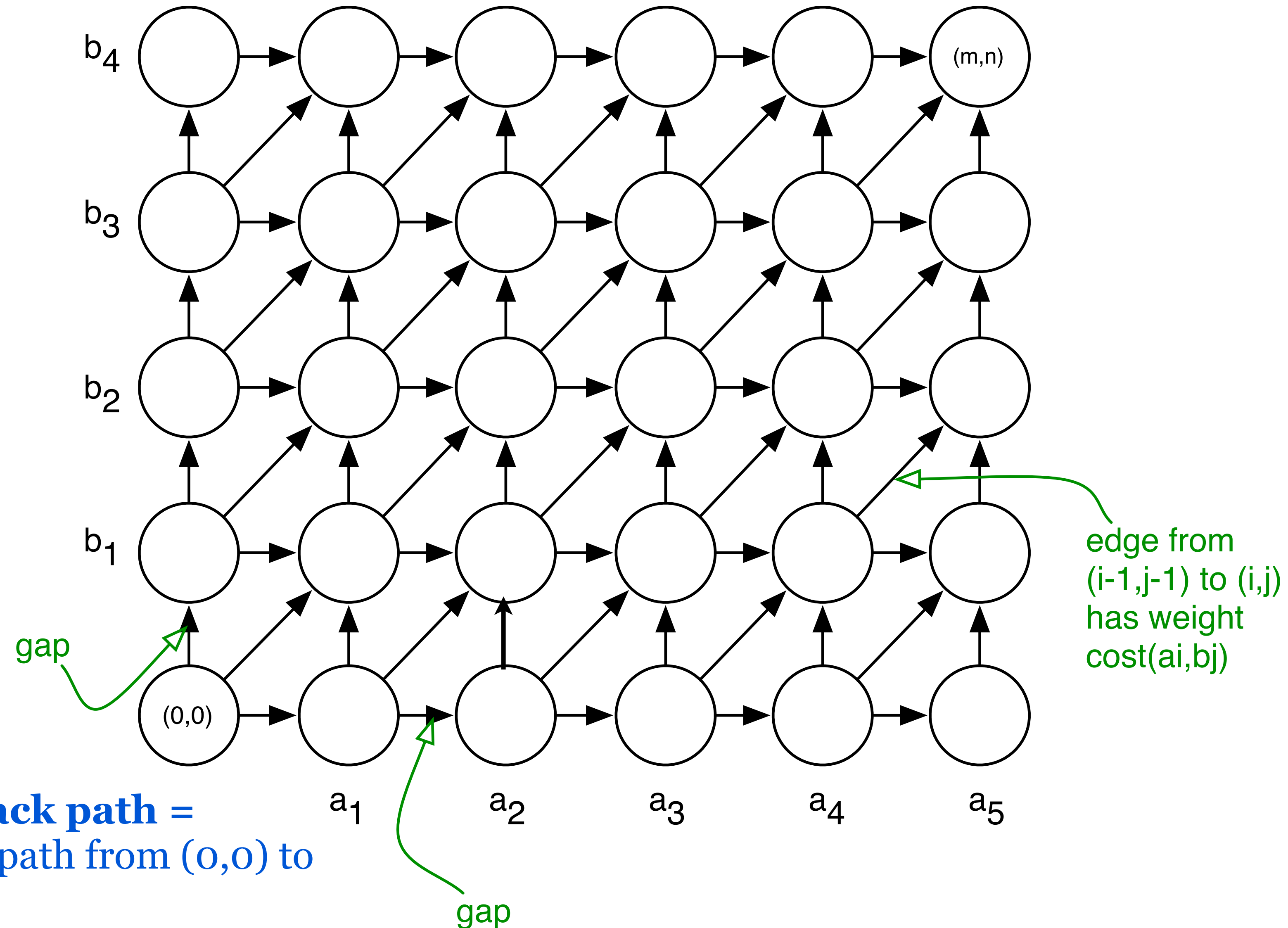
The previous sequence alignment / edit distance algorithm is an example of dynamic programming.

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

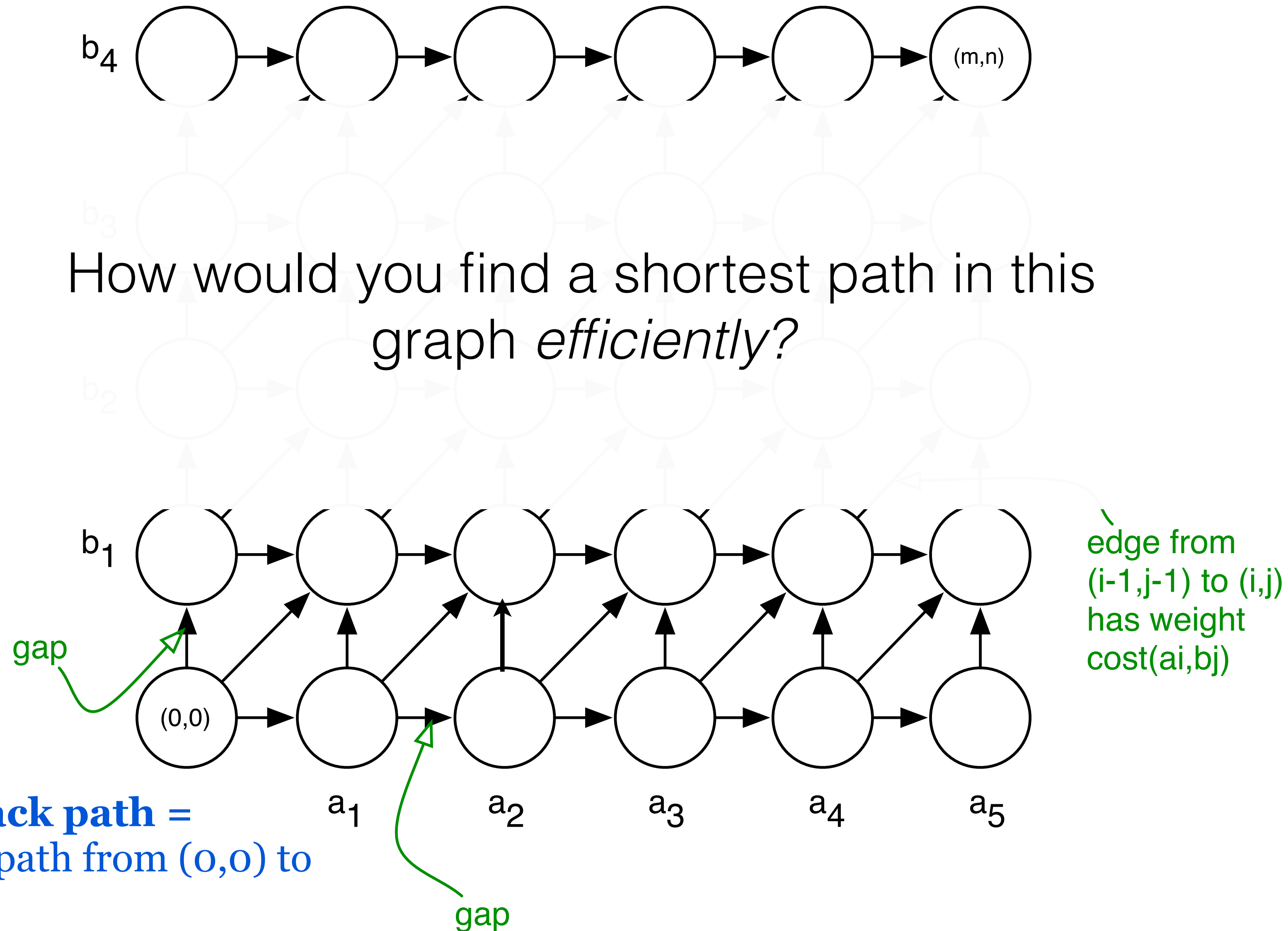
Requirements for DP to apply:

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



Another View: Recasting as a Graph



Side Note: Lower Bounds

- Suppose the lengths of x and y are n .
- Clearly, need at least $\Omega(n)$ 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.

Side Note: Lower Bounds

- Suppose the lengths of x and y are n .
- Clearly, need at least $\Omega(n)$ 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...
(Alazarov, 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.