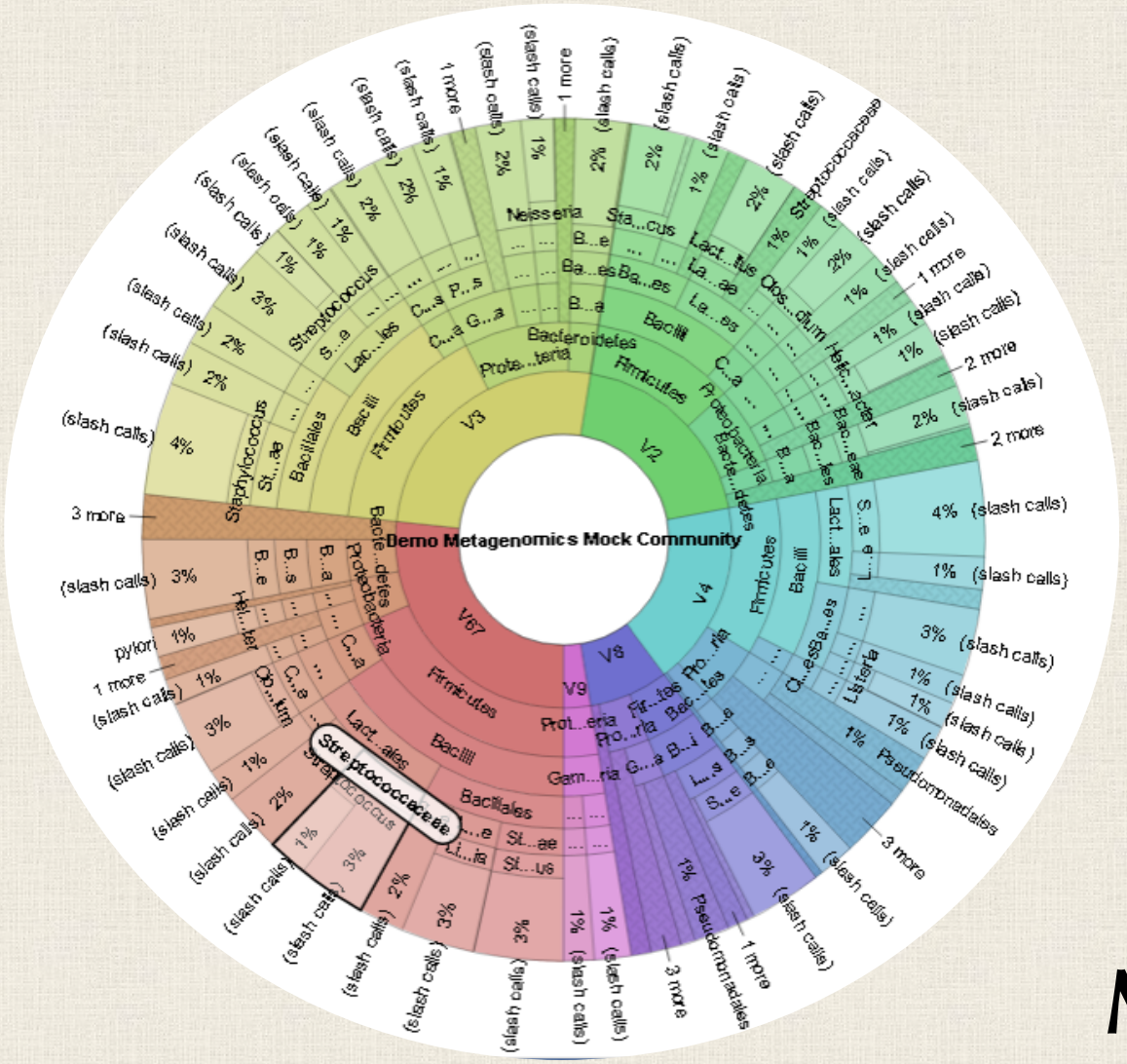
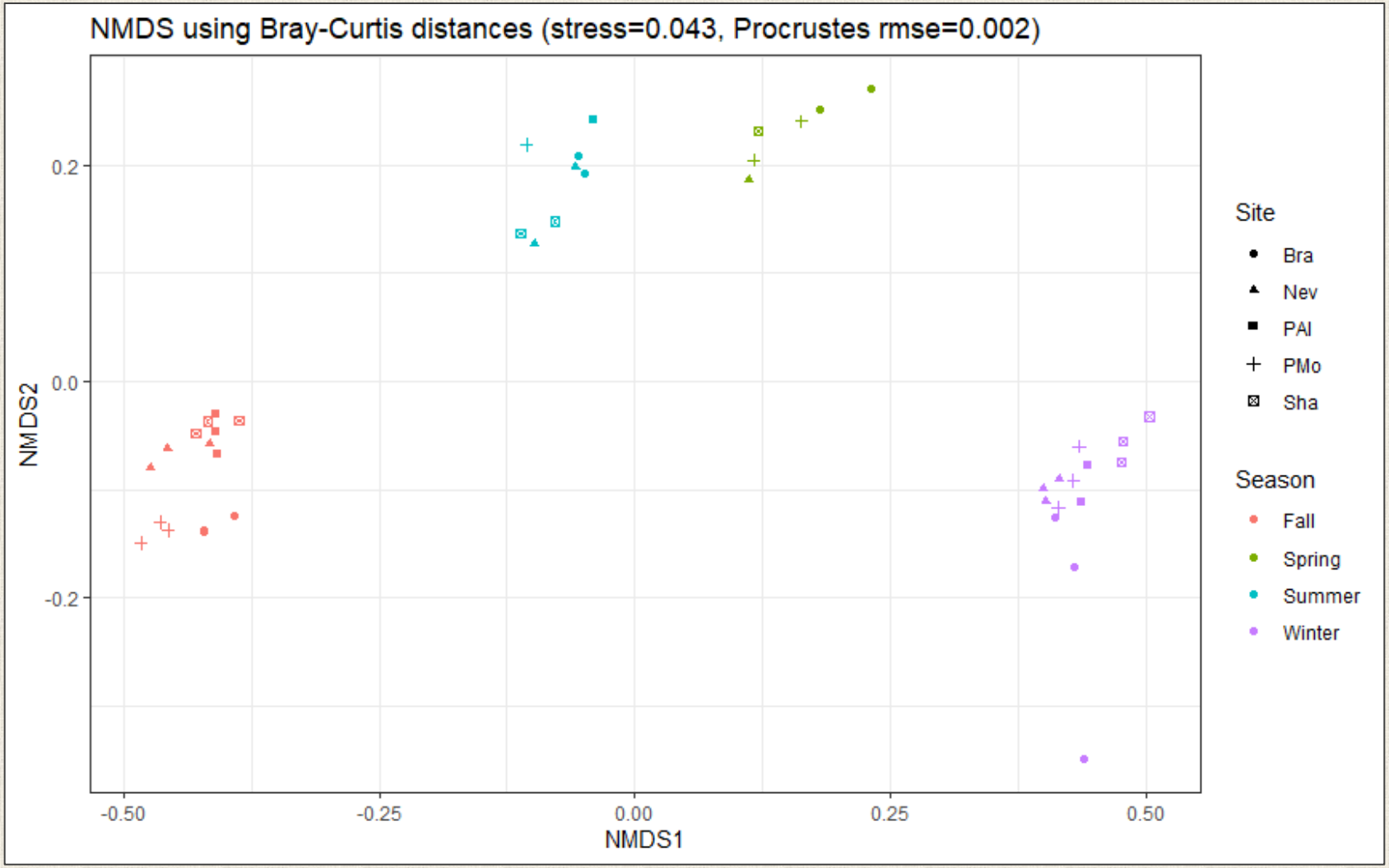


Module 2

Comparing Biological Sequences

We Have a Pretty Picture Showing Seasonality of Bacteria. Now What?



We Have a Pretty Picture Showing Seasonality of Bacteria. Now What?

Goal: given a sample of 16S genes and a database of 16S genes from different species, which species did each fragment in our sample come from?

We Have a Pretty Picture Showing Seasonality of Bacteria. Now What?

Goal: given a sample of 16S genes and a database of 16S genes from different species, which species did each fragment in our sample come from?

Recall: when comparing *n* metagenomics samples, we first learned to compare *pairs* of samples.

We Have a Pretty Picture Showing Seasonality of Bacteria. Now What?

Goal: given a sample of 16S genes and a database of 16S genes from different species, which species did each fragment in our sample come from?

Recall: when comparing *n* metagenomics samples, we first learned to compare *pairs* of samples.

So, let's examine a simpler problem: how can we compare *two* strings to determine how different they are?

COMPARING STRINGS

Comparing Genes

Comparing Strings Problem: *Compare two similar strings.*

- **Input:** Two strings.
- **Output:** How "different" these strings are.

Comparing Genes

Comparing Strings Problem: *Compare two similar strings.*

- **Input:** Two strings.
- **Output:** How "different" these strings are.

This is not a well-defined problem, since we need to clearly define what we mean by "different".

Comparing Genes

Comparing Strings Problem: *Compare two similar strings.*

- **Input:** Two strings.
- **Output:** How "different" these strings are.

This is not a well-defined problem, since we need to clearly define what we mean by "different".

STOP: How might we define a function $d(s1, s2)$ that represents a distance (or similarity) between strings $s1$ and $s2$?

Try 1: Hamming Distance

Hamming Distance Problem:

- **Input:** Two strings.
- **Output:** The number of "mismatched" symbols in the two strings.

Try 1: Hamming Distance

Hamming Distance Problem:

- **Input:** Two strings.
- **Output:** The number of "mismatched" symbols in the two strings.

```
ATGCATGC
TGCATGCA
```

Hamming distance = 8

Try 1: Hamming Distance

Hamming Distance Problem:

- **Input:** Two strings.
- **Output:** The number of "mismatched" symbols in the two strings.

```
ATGCATGC
TGCATGCA
```

Hamming distance = 8

STOP: What are the issues with using Hamming distance to compare strings?

Try 1: Hamming Distance

Hamming Distance Problem:

- **Input:** Two strings.
- **Output:** The number of "mismatched" symbols in the two strings.

```
ATGCATGC
TGCATGCA
```

Hamming distance = 8

Note: these strings have a long shared substring, it just doesn't line up perfectly.

Try 2: Longest Substring

Longest Shared Substring Problem:

- **Input:** Two strings.
- **Output:** The longest substring shared by both strings.

STOP: What are the weaknesses of using the length of a longest shared substring to represent the similarity between two strings?

Similar Strings Don't Necessarily Have a Long Shared Substring

Longest Shared Substring Problem:

- **Input:** Two strings.
- **Output:** The longest substring shared by both strings.

Consider the strings AAACAAACAAACAAACAAACAAA and AAAGAAAGAAAGAAAGAAAGAAAGAAA. These strings are very similar, but they don't have a long shared substring in common!

Try 3: Counting Shared *k*-Mers

Instead of finding a longest shared substring of two strings, we will count the number of shared substrings.

For simplicity, we restrict to substrings of the same length; recall that a ***k*-mer** is the term we use in comp bio for a string of length *k*.

Try 3: Counting Shared *k*-Mers

`s1 = ACGTATACACGTAT`

String	Count
ACA	1
ACG	2
ATA	1
CAC	1
CGT	2
GTA	2
TAC	1
TAT	2

`s2 = TATCGGTATATCCTAC`

String	Count
ATA	1
ATC	2
CCT	1
CGG	1
CTA	1
GGT	1
GTA	1
TAC	1
TAT	3
TCC	1
TCG	1

STOP: How should we count the # of shared 3-mers of two strings?

Try 3: Counting Shared *k*-Mers

`s1 = ACGTATACACGTAT`

String	Count
ACA	1
ACG	2
ATA	1
CAC	1
CGT	2
GTA	2
TAC	1
TAT	2

`s2 = TATCGGTATATCCTAC`

String	Count
ATA	1
ATC	2
CCT	1
CGG	1
CTA	1
GGT	1
GTA	1
TAC	1
TAT	3
TCC	1
TCG	1

Take minimum counts for each shared *k*-mer:

1 + 1 + 1 + 2 = 5

Try 3: Counting Shared *k*-Mers

Counting Shared *k*-Mers Problem:

- **Input:** Two strings and an integer *k*.
- **Output:** The number of *k*-mers shared by the two strings.

Code Challenge: Write a function solving this problem.

Hint: How can the frequency map help?

Try 3: Counting Shared *k*-Mers

Counting Shared *k*-Mers Problem:

- **Input:** Two strings and an integer *k*.
- **Output:** The number of *k*-mers shared by the two strings.

STOP: What remaining weakness do you see with modeling string comparison in this way?

Try 3: Counting Shared *k*-Mers

Counting Shared *k*-Mers Problem:

- **Input:** Two strings and an integer *k*.
- **Output:** The number of *k*-mers shared by the two strings.

STOP: What remaining weakness do you see with modeling string comparison in this way?

Answer: We lose any information about the *order* of the shared strings.

INTRODUCTION TO SEQUENCE ALIGNMENT

Toward a Better Approach

STOP: What similarities do you see in these strings?

```
ATGCTTA
TGCATTAA
```

Toward a Better Approach

STOP: What similarities do you see in these strings?

```
ATGCTTA
TGCATTAA
```

Key Point: we can find similarities if we "slide" the strings, letting symbols shift (but stay in same order).

```
ATGC-TTA-
-TGCATTAA
```

Toward a Better Approach

Symbol Matching Problem:

- **Input:** Two strings.
- **Output:** The greatest number of matched symbols in any "alignment" of the two strings.

```
ATGC-TTA-
-TGCATTAA
```

Toward a Better Approach

Symbol Matching Problem:

- **Input:** Two strings.
- **Output:** The greatest number of matched symbols in any "alignment" of the two strings.

Exercise: How many matches can you find if the strings are `ATGTTATA` and `ATCGTCC`? What algorithm did you use?

Matching Symbols as a Game

Growing alignment	Remaining symbols	Score
	A T G T T A T A A T C G T C C	

Matching Symbols as a Game

Growing alignment	Remaining symbols	Score
	A T G T T A T A A T C G T C C	
A **A**	T G T T A T A T C G T C C	**+1**

Matching Symbols as a Game

Growing alignment	Remaining symbols	Score
	A T G T T A T A A T C G T C C	
A **A**	T G T T A T A T C G T C C	+1
A T **A T**	G T T A T A C G T C C	+1

Matching Symbols as a Game

Growing alignment	Remaining symbols	Score
	A T G T T A T A A T C G T C C	
A A	T G T T A T A T C G T C C	+1
A T A T	G T T A T A C G T C C	+1
A T – A T C	G T T A T A G T C C	

Matching Symbols as a Game

Growing alignment	Remaining symbols	Score
	A T G T T A T A A T C G T C C	
A A	T G T T A T A T C G T C C	+1
A T A T	G T T A T A C G T C C	+1
A T – A T C	G T T A T A G T C C	
A T – G A T C G	T T A T A T C C	+1

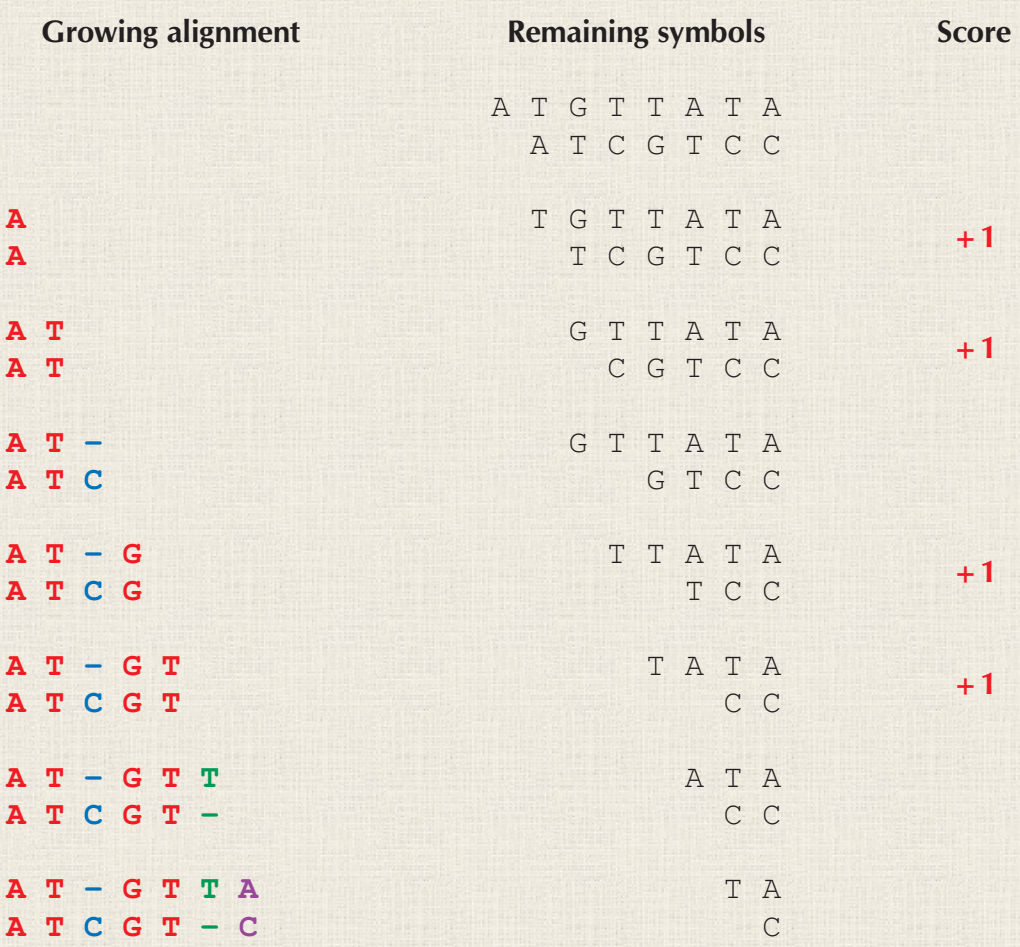
Matching Symbols as a Game

Growing alignment	Remaining symbols	Score
	A T G T T A T A A T C G T C C	
A A	T G T T A T A T C G T C C	+1
A T A T	G T T A T A C G T C C	+1
A T – A T C	G T T A T A G T C C	
A T – G A T C G	T T A T A T C C	+1
A T – G T A T C G T	T A T A C C	+1

Matching Symbols as a Game

Growing alignment	Remaining symbols	Score
	A T G T T A T A A T C G T C C	
A A	T G T T A T A T C G T C C	+1
A T A T	G T T A T A C G T C C	+1
A T – A T C	G T T A T A G T C C	
A T – G A T C G	T T A T A T C C	+1
A T – G T A T C G T	T A T A C C	+1
A T – G T T A T C G T –	A T A C C	

Matching Symbols as a Game



Matching Symbols as a Game

Growing alignment	Remaining symbols	Score
	A T G T T A T A A T C G T C C	
A A	T G T T A T A T C G T C C	+1
A T A T	G T T A T A C G T C C	+1
A T – A T C	G T T A T A G T C C	
A T – G A T C G	T T A T A T C C	+1
A T – G T A T C G T	T A T A C C	+1
A T – G T T A T C G T –	A T A C C	
A T – G T T A A T C G T – C	T A C	
A T – G T T A T A T C G T – C –	A C	

Matching Symbols as a Game

Growing alignment	Remaining symbols	Score
	A T G T T A T A A T C G T C C	
A A	T G T T A T A T C G T C C	+1
A T A T	G T T A T A C G T C C	+1
A T – A T C	G T T A T A G T C C	
A T – G A T C G	T T A T A T C C	+1
A T – G T A T C G T	T A T A C C	+1
A T – G T T A T C G T –	A T A C C	
A T – G T T A A T C G T – C	T A C	
A T – G T T A T A T C G T – C –	A C	
A T – G T T A T A A T C G T – C – C		

From a Game to a Definition

Given two strings *v* and *w*, an **alignment** of *v* and *w* is a two-row matrix such that:

- the first row contains symbols of *v*
- the second row contains symbols of *w*
- each row may also contain **gap symbols** ("-")

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

From a Game to a Definition

Given two strings *v* and *w*, an **alignment** of *v* and *w* is a two-row matrix such that:

- the first row contains symbols of *v*
- the second row contains symbols of *w*
- each row may also contain **gap symbols** ("-")

A T - G T T A T A
A T C **G T** - C - C

Matches

From a Game to a Definition

Given two strings *v* and *w*, an **alignment** of *v* and *w* is a two-row matrix such that:

- the first row contains symbols of *v*
- the second row contains symbols of *w*
- each row may also contain **gap symbols** ("-")

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

Mismatches

From a Game to a Definition

Given two strings *v* and *w*, an **alignment** of *v* and *w* is a two-row matrix such that:

- the first row contains symbols of *v*
- the second row contains symbols of *w*
- each row may also contain **gap symbols** ("-")

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

Insertions

From a Game to a Definition

Given two strings *v* and *w*, an **alignment** of *v* and *w* is a two-row matrix such that:

- the first row contains symbols of *v*
- the second row contains symbols of *w*
- each row may also contain **gap symbols** ("-")

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

Deletions

Longest Common Subsequence

A **common subsequence** of *v* and *w* is a sequence of symbols occurring (not necessarily contiguously) in both *v* and *w*.

Longest Common Subsequence

A **common subsequence** of *v* and *w* is a sequence of symbols occurring (not necessarily contiguously) in both *v* and *w*.

Note that the **matches** in an alignment of *v* and *w* form a common subsequence of *v* and *w*.

A T – **G T** T A T A

A T C **G T** – C – C

The Problems are the Same!

Longest Common Subsequence Length Problem:

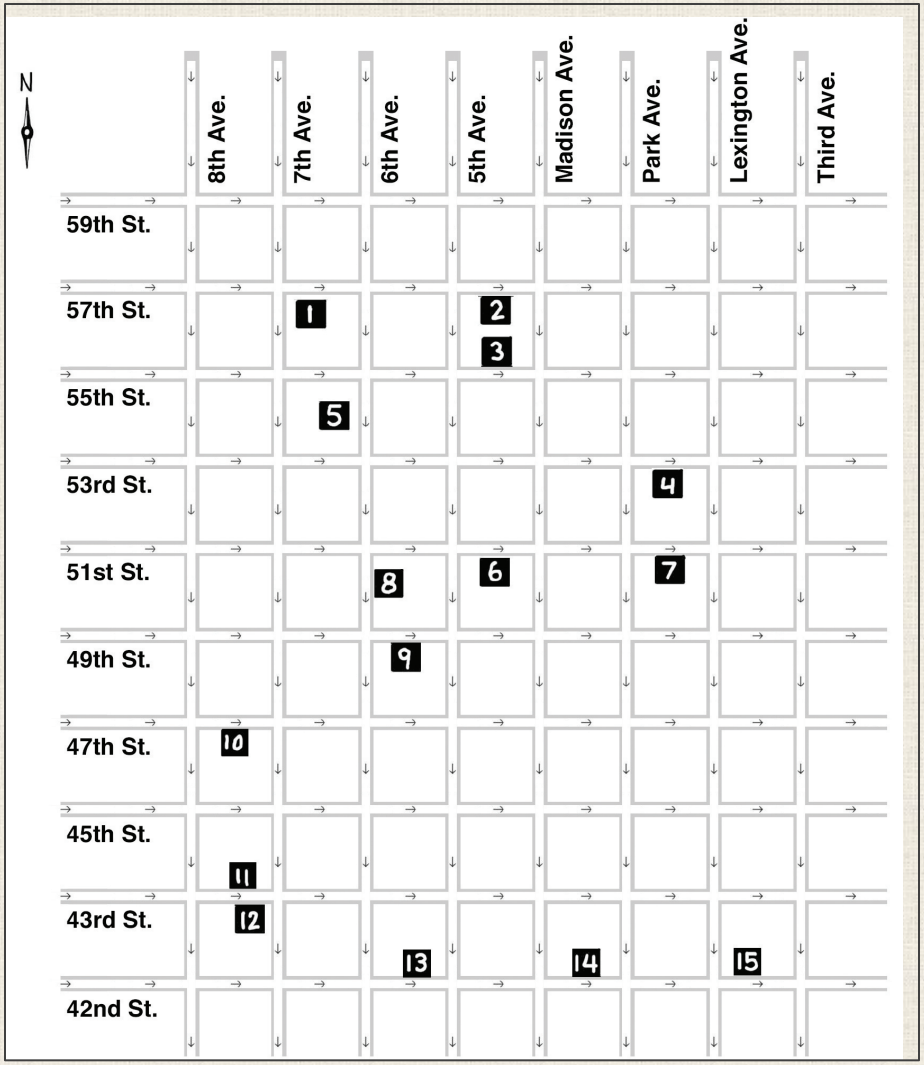
- **Input:** Two strings.
- **Output:** The length of a longest common subsequence of these strings.

Symbol Matching Problem:

- **Input:** Two strings.
- **Output:** The greatest number of matched symbols in any "alignment" of the two strings.

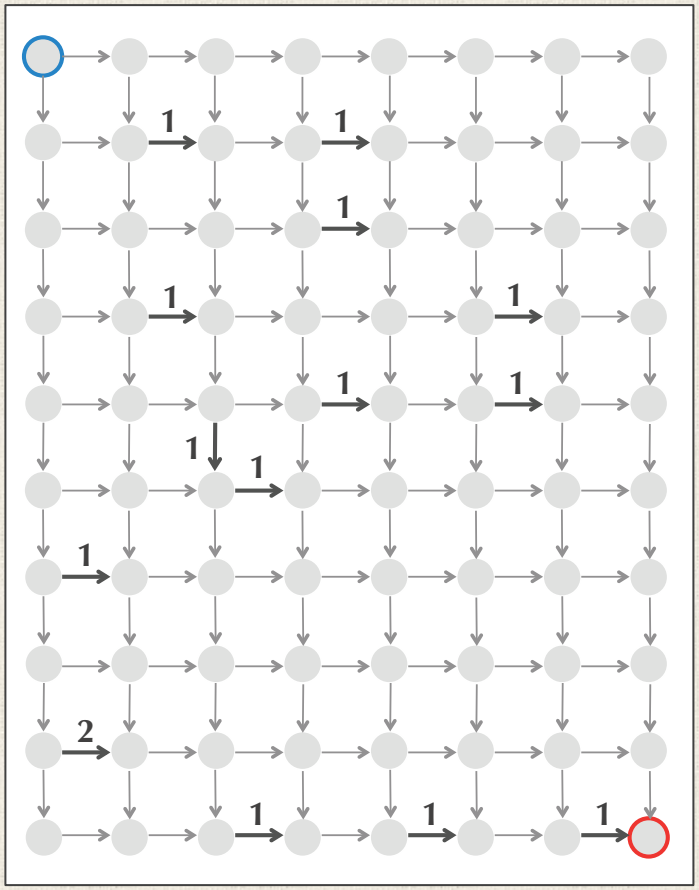
THE MANHATTAN TOURIST PROBLEM

Manhattan Tourist Problem



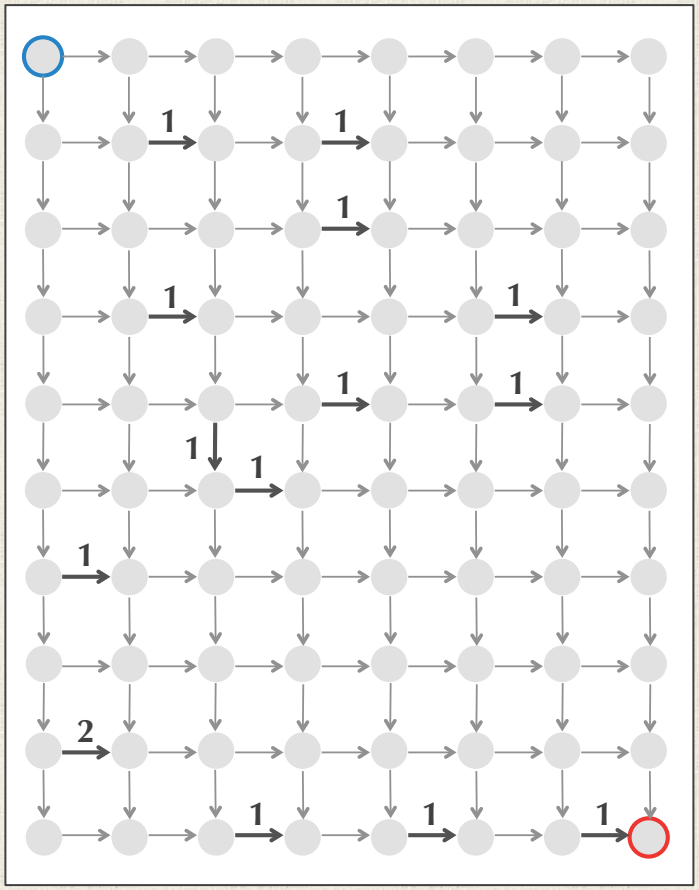
STOP: How can we see the most sites if we move from 59th and 8th to 42nd and 3rd, moving south or east at each step? (And what algorithm did you use?)

Manhattan Tourist as a Network



Weight of edge: number of attractions along the edge.

Manhattan Tourist as a Network



Weight of edge: number of attractions along the edge.

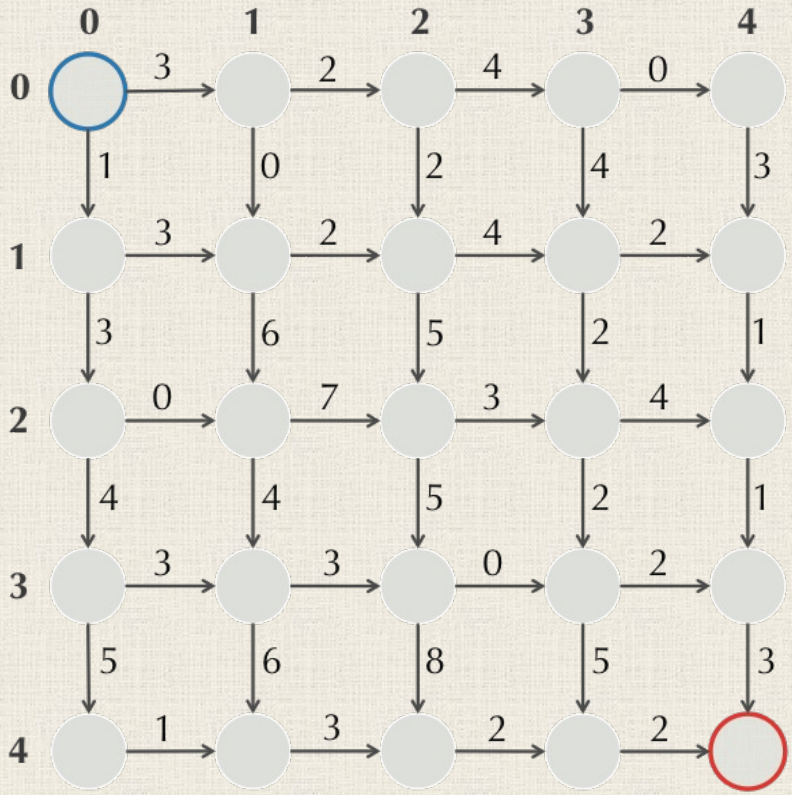
Goal: Find a ***longest path*** from ***source*** (top left) to ***sink*** (bottom right).

Manhattan Tourist as a Network

Manhattan Tourist Problem:

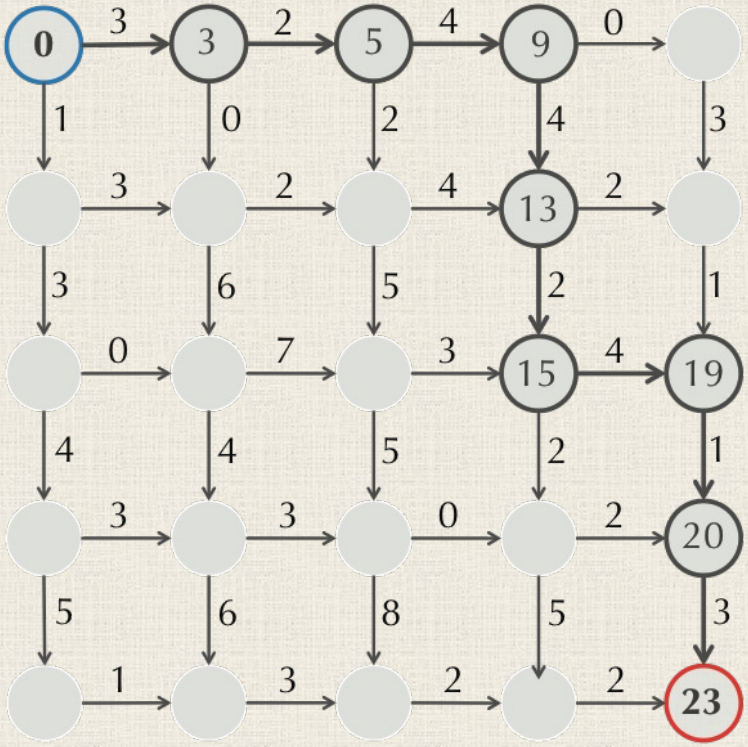
- **Input:** A weighted n x m rectangular grid (n + 1 rows and m + 1 columns).
- **Output:** A longest path from source (0, 0) to sink (n, m) in the grid.

Designing a Manhattan Algorithm



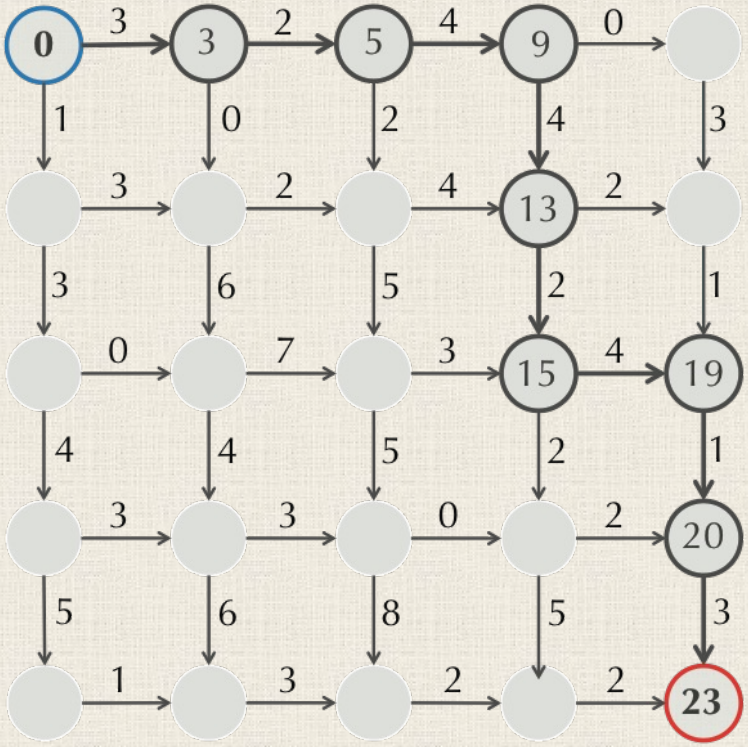
Exercise: What do you think is the longest path in this city? What algorithm did you use?

A "Greedy" Manhattan Algorithm



STOP: Does this algorithm solve the problem? Is there a better path?

A "Greedy" Manhattan Algorithm



Answer: No! We need a more clever approach.

SEQUENCE ALIGNMENT AS A PATH IN A NETWORK

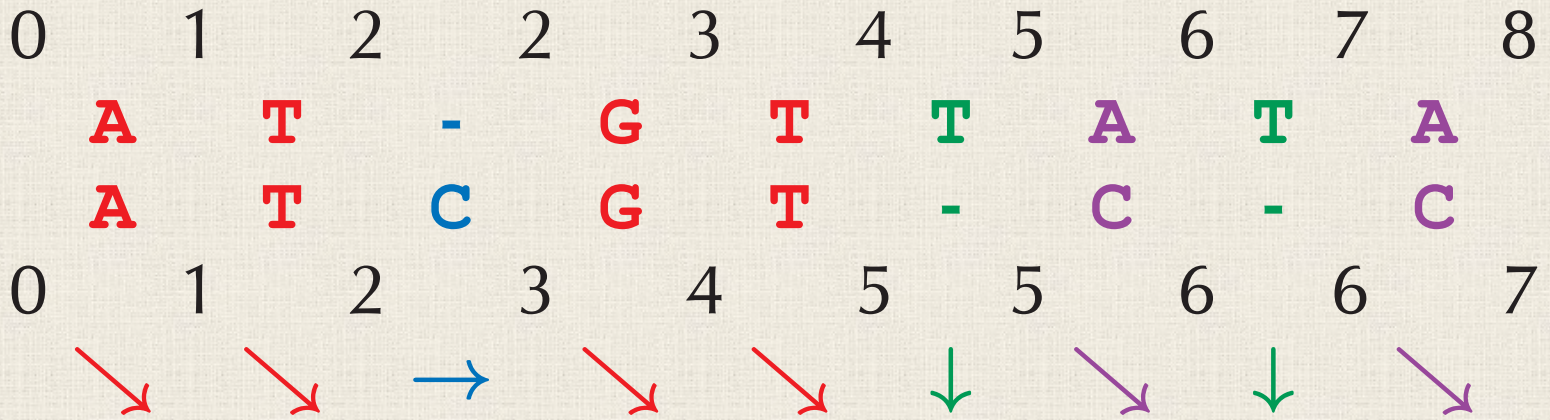
Returning to Sequence Alignment ...

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

Returning to Sequence Alignment ...

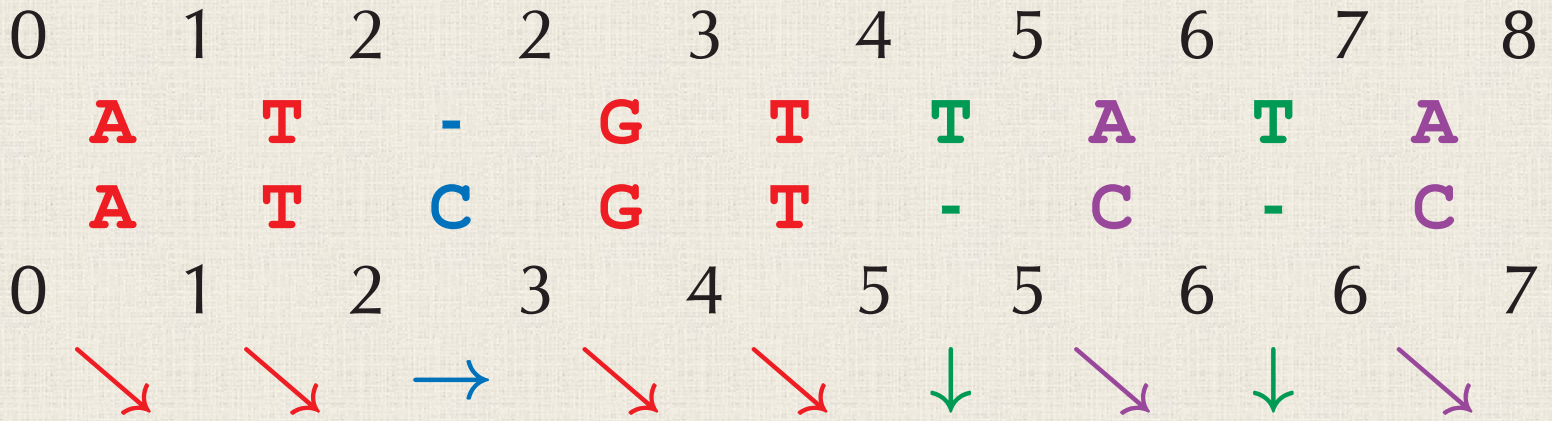
0	1	2	2	3	4	5	6	7	8
	A	T	-	G	T	T	A	T	A
	A	T	C	G	T	-	C	-	C
0	1	2	3	4	5	5	6	6	7

Returning to Sequence Alignment ...



(0, 0) ↘ (1, 1) ↘ (2, 2) → (2, 3) ↘ (3, 4)
↘ (4, 5) ↓ (5, 5) ↘ (6, 6) ↓ (7, 6) ↘ (8, 7)

Returning to Sequence Alignment ...



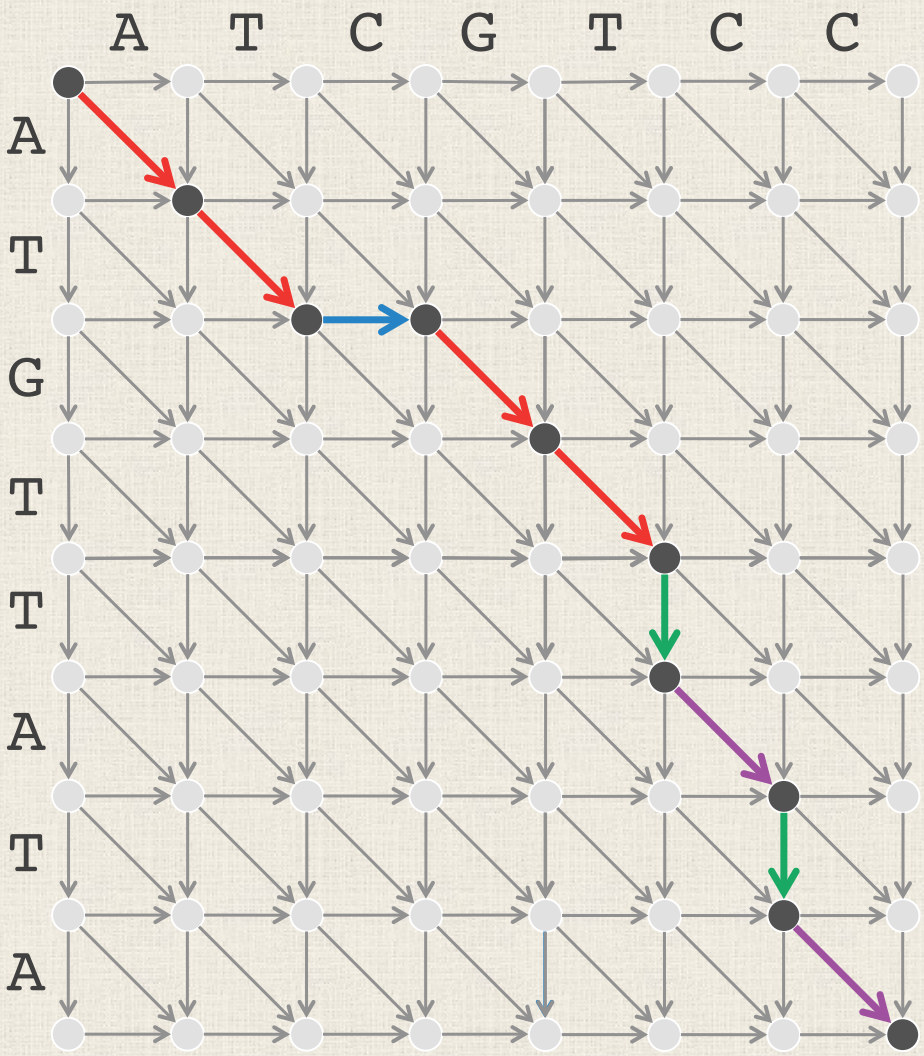
(0, 0) ↘ (1, 1) ↘ (2, 2) → (2, 3) ↘ (3, 4)
↘ (4, 5) ↓ (5, 5) ↘ (6, 6) ↓ (7, 6) ↘ (8, 7)

This is a path in a 2-D network!

Representing an Alignment as a Path

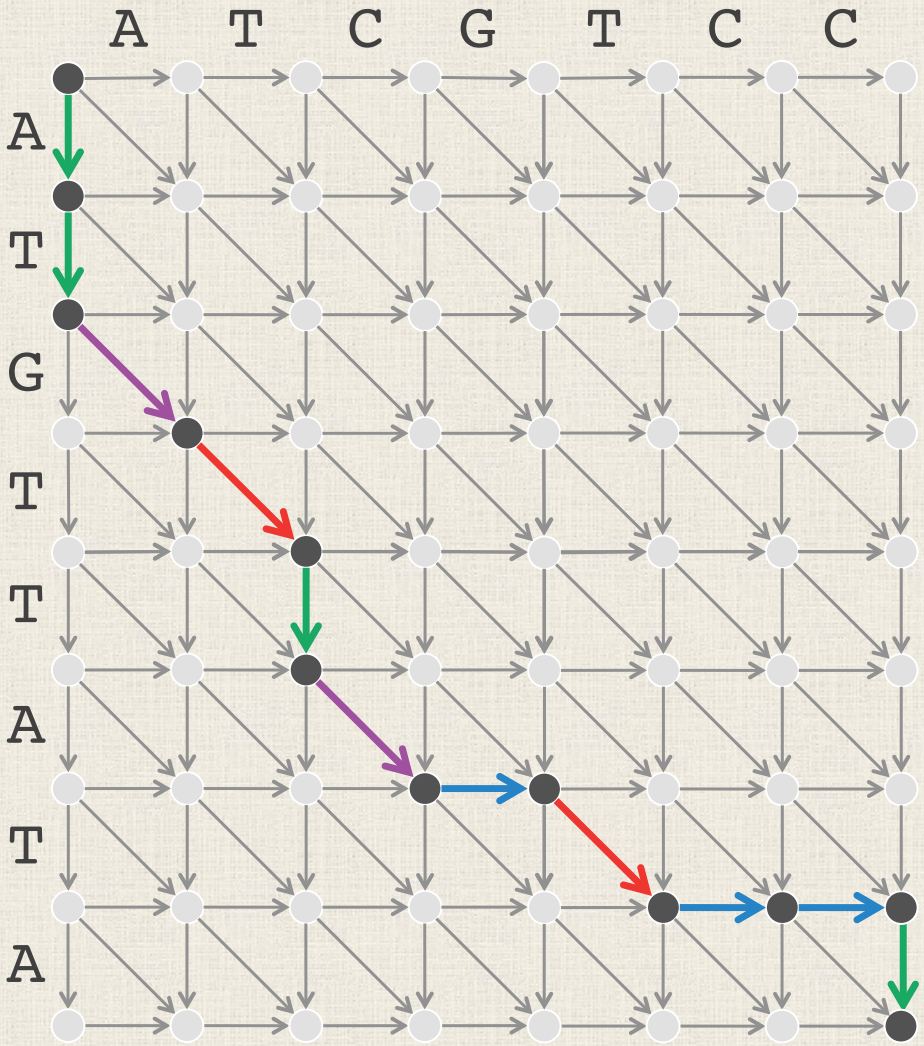
A T - G T T A T A
A T C G T - C - C

This network is called the **alignment network** of the strings ATGTTATA and ATCGTCC.



We can also construct an alignment from a path

Exercise: What alignment does this path correspond to?



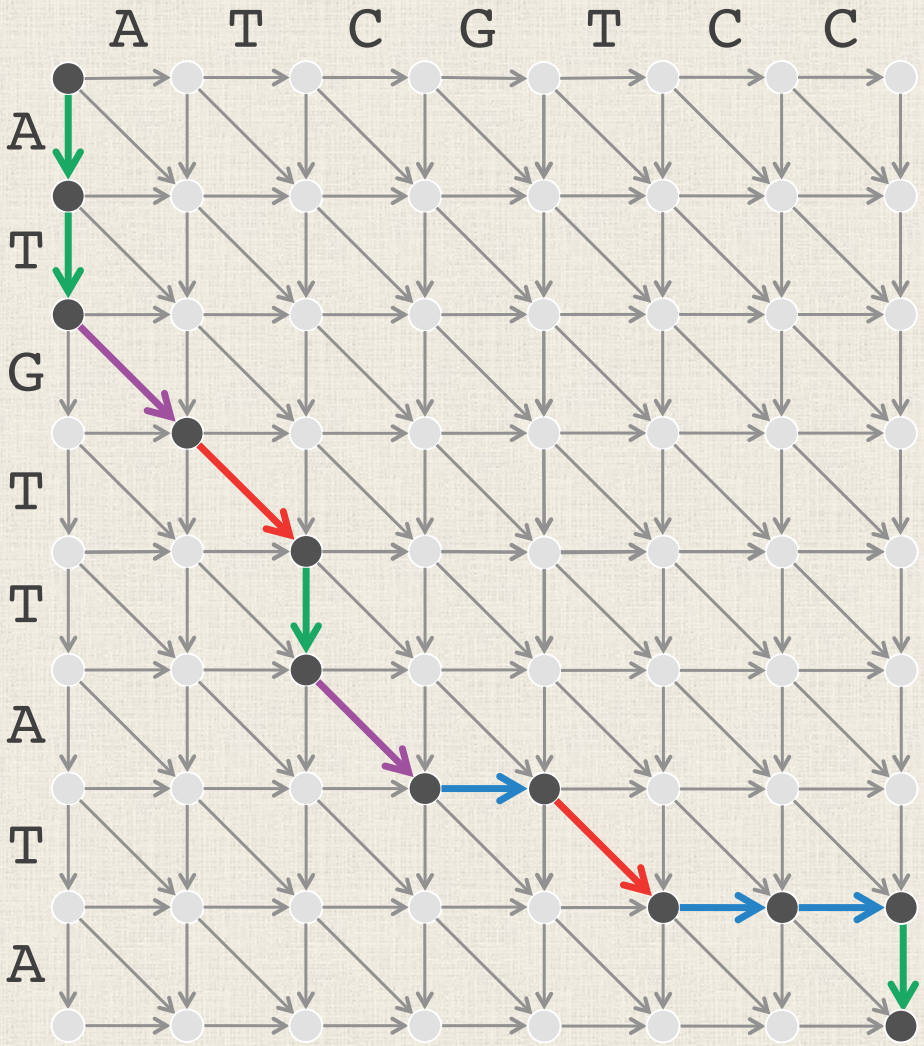
We can also construct an alignment from a path

Exercise: What alignment does this path correspond to?

Answer:

A T G T T A - T - - A

- - A T - C G T C C -



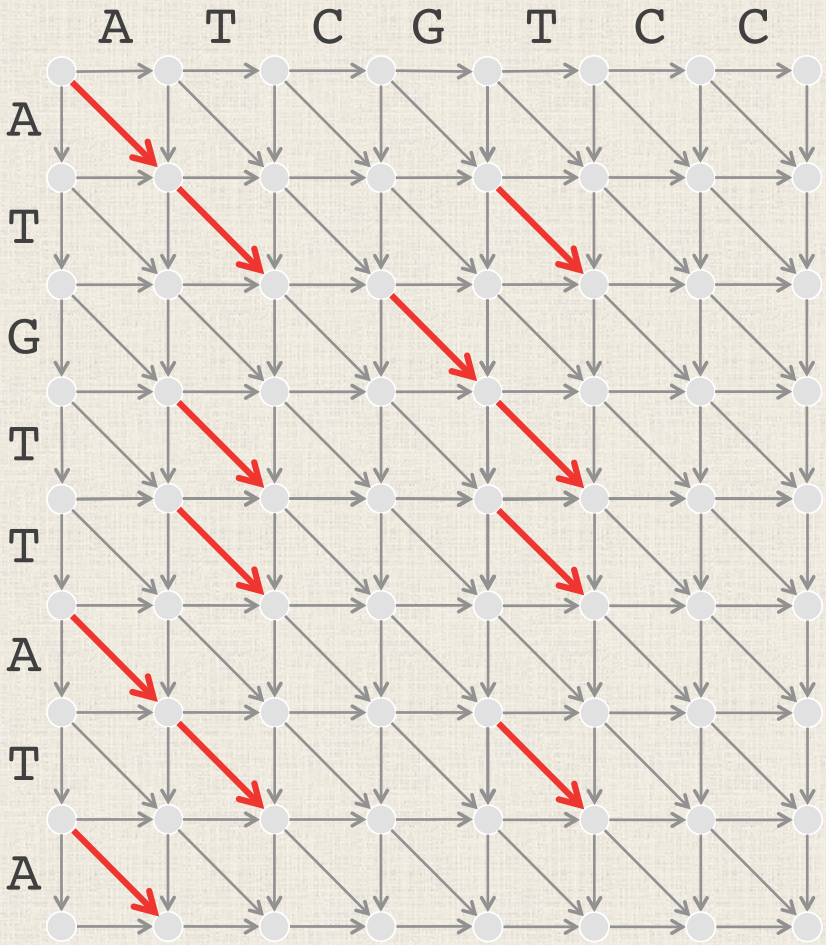
Solving the Symbol Matching Problem

Symbol Matching Problem:

- **Input:** Two strings.
- **Output:** The greatest number of matched symbols in any alignment of the two strings.

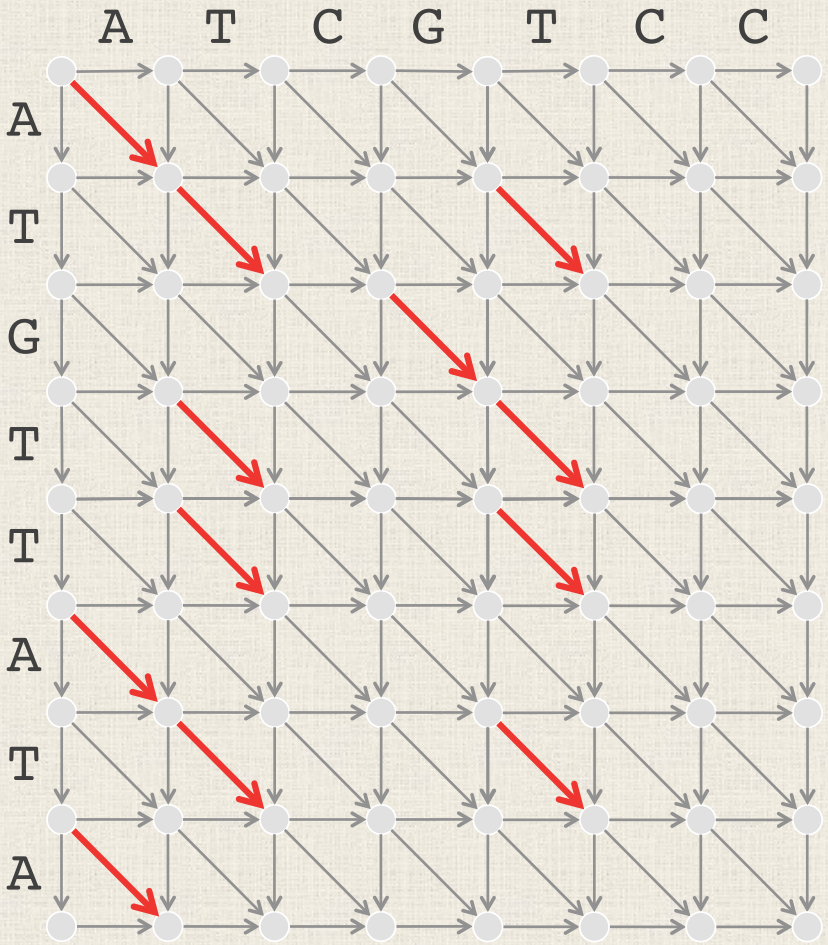
STOP: How can we use the alignment network to solve this problem?

Counting Matches Only



Answer: If we weight the red edges as 1 and the other edges as 0, then a maximum-weight path from source to sink solves the Symbol Matching Problem!

Counting Matches Only



Answer: If we weight the red edges as 1 and the other edges as 0, then a maximum-weight path from source to sink solves the Symbol Matching Problem!

But we haven't said how to *find* the maximum length of a path.

AN INTRO TO RECURSION AND DYNAMIC PROGRAMMING

From Long Ago: Summing Integers

Summing Integers Problem

- **Input:** An integer *n*.
- **Output:** The sum of the first *n* positive integers.

From Long Ago: Summing Integers

Summing Integers Problem

- **Input:** An integer n.
- **Output:** The sum of the first n positive integers.

If $S(n)$ is the sum of the first n positive integers, then

$$S(n) = n + (n - 1) + (n - 2) + \ldots + 2 + 1 = n + S(n - 1)$$

From Long Ago: Summing Integers

Summing Integers Problem

- **Input:** An integer n.
- **Output:** The sum of the first n positive integers.

If $S(n)$ is the sum of the first n positive integers, then

$$S(n) = n + (n - 1) + (n - 2) + \ldots + 2 + 1 = n + S(n - 1)$$

Recurrence relation: An expression for a function f(x) in terms of values of f(y) where $y < x$.

From Long Ago: Summing Integers

Summing Integers Problem

- **Input:** An integer n.
- **Output:** The sum of the first n positive integers.

Recursive algorithm: "subcontracts" work of a program by calling itself on a smaller input.

```
RecursiveSum(n)
        if n > 1
                return n + RecursiveSum(n-1)
        else
                return 1
```

From Long Ago: Summing Integers

Summing Integers Problem

- **Input:** An integer *n*.
- **Output:** The sum of the first *n* positive integers.

Base Case: simplest case of the program, where we return a "trivial" value.

```
RecursiveSum(n)
        if n > 1
                return n + RecursiveSum(n-1)
        else
                return 1
```

From Long Ago: Summing Integers

Summing Integers Problem

- **Input:** An integer n.
- **Output:** The sum of the first n positive integers.

Inductive Step: Other, normal cases, where we apply the recursive call(s) to the function.

```
RecursiveSum(n)
        if n > 1
                return n + RecursiveSum(n-1)
        else
                return 1
```

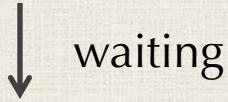
How Recursion Works

RecursiveSum(6) = 6 + **RecursiveSum**(5)

```
RecursiveSum(n)
        if n > 1
                return n + RecursiveSum(n-1)
        else
                return 1
```

How Recursion Works

RecursiveSum(6) = 6 + **RecursiveSum**(5)

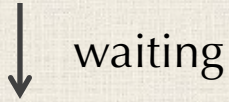


RecursiveSum(5) = 5 + **RecursiveSum**(4)

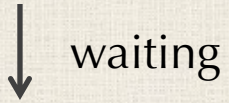
```
RecursiveSum(n)
        if n > 1
                return n + RecursiveSum(n-1)
        else
                return 1
```

How Recursion Works

RecursiveSum(6) = 6 + **RecursiveSum**(5)



RecursiveSum(5) = 5 + **RecursiveSum**(4)



RecursiveSum(4) = 4 + **RecursiveSum**(3)

```
RecursiveSum(n)
        if n > 1
                return n + RecursiveSum(n-1)
        else
                return 1
```

How Recursion Works

RecursiveSum(6) = 6 + **RecursiveSum**(5)

waiting

RecursiveSum(5) = 5 + **RecursiveSum**(4)

waiting

RecursiveSum(4) = 4 + **RecursiveSum**(3)

waiting

RecursiveSum(3) = 3 + **RecursiveSum**(2)

How Recursion Works

RecursiveSum(6) = 6 + **RecursiveSum**(5)

waiting

RecursiveSum(5) = 5 + **RecursiveSum**(4)

waiting

RecursiveSum(4) = 4 + **RecursiveSum**(3)

waiting

RecursiveSum(3) = 3 + **RecursiveSum**(2)

waiting

RecursiveSum(2) = 2 + **RecursiveSum**(1)

This collection of functions that are waiting to be finished is called the **stack** – these function calls are stored in the computer's memory.

How Recursion Works

RecursiveSum(6) = 6 + **RecursiveSum**(5)

RecursiveSum(5) = 5 + **RecursiveSum**(4)

RecursiveSum(4) = 4 + **RecursiveSum**(3)

RecursiveSum(3) = 3 + **RecursiveSum**(2)

RecursiveSum(2) = 2 + **RecursiveSum**(1)

RecursiveSum(1) = 1

How Recursion Works

RecursiveSum(6) = 6 + **RecursiveSum**(5)

RecursiveSum(5) = 5 + **RecursiveSum**(4)

RecursiveSum(4) = 4 + **RecursiveSum**(3)

RecursiveSum(3) = 3 + **RecursiveSum**(2)

RecursiveSum(2) = 2 + **RecursiveSum**(1)

RecursiveSum(1) = 1

How Recursion Works

RecursiveSum(6) = 6 + **RecursiveSum**(5)

waiting

RecursiveSum(5) = 5 + **RecursiveSum**(4)

waiting

RecursiveSum(4) = 4 + **RecursiveSum**(3)

waiting

RecursiveSum(3) = 3 + **RecursiveSum**(2)

waiting

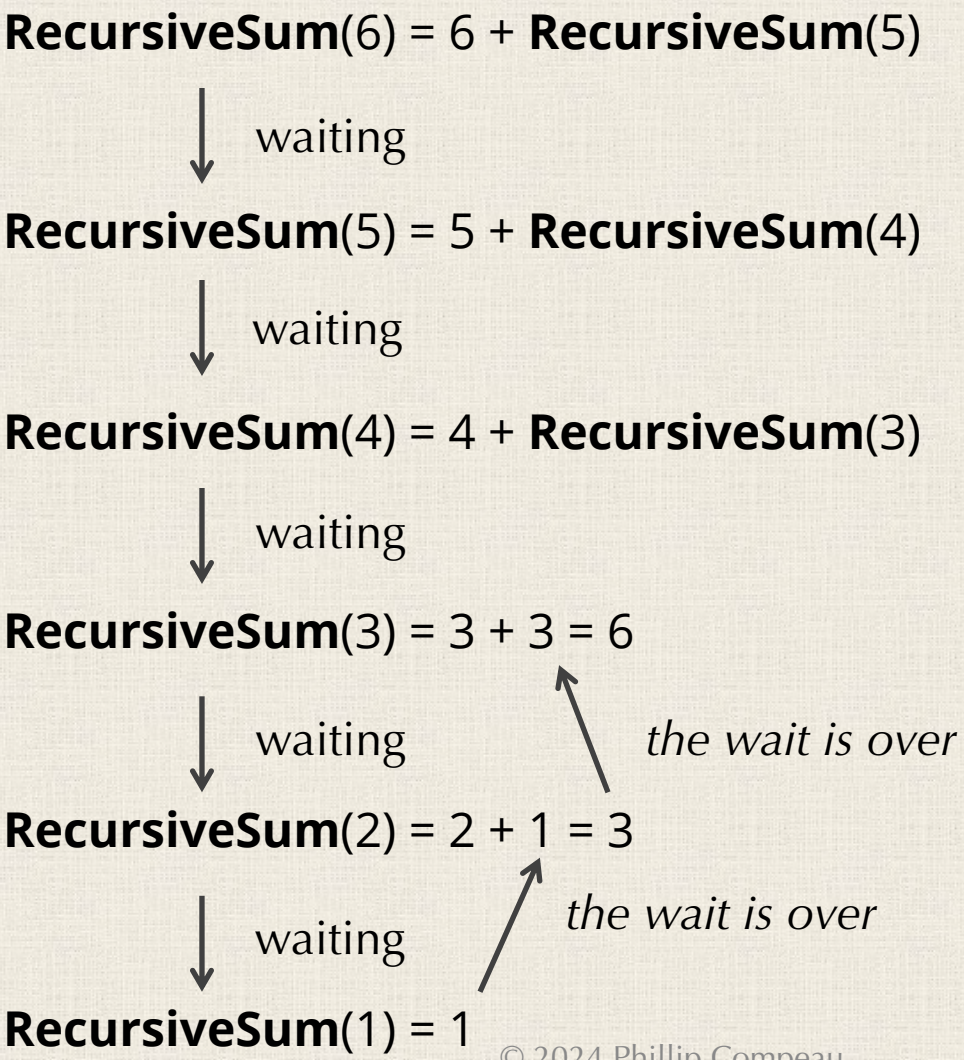
RecursiveSum(2) = 2 + 1 = 3

waiting

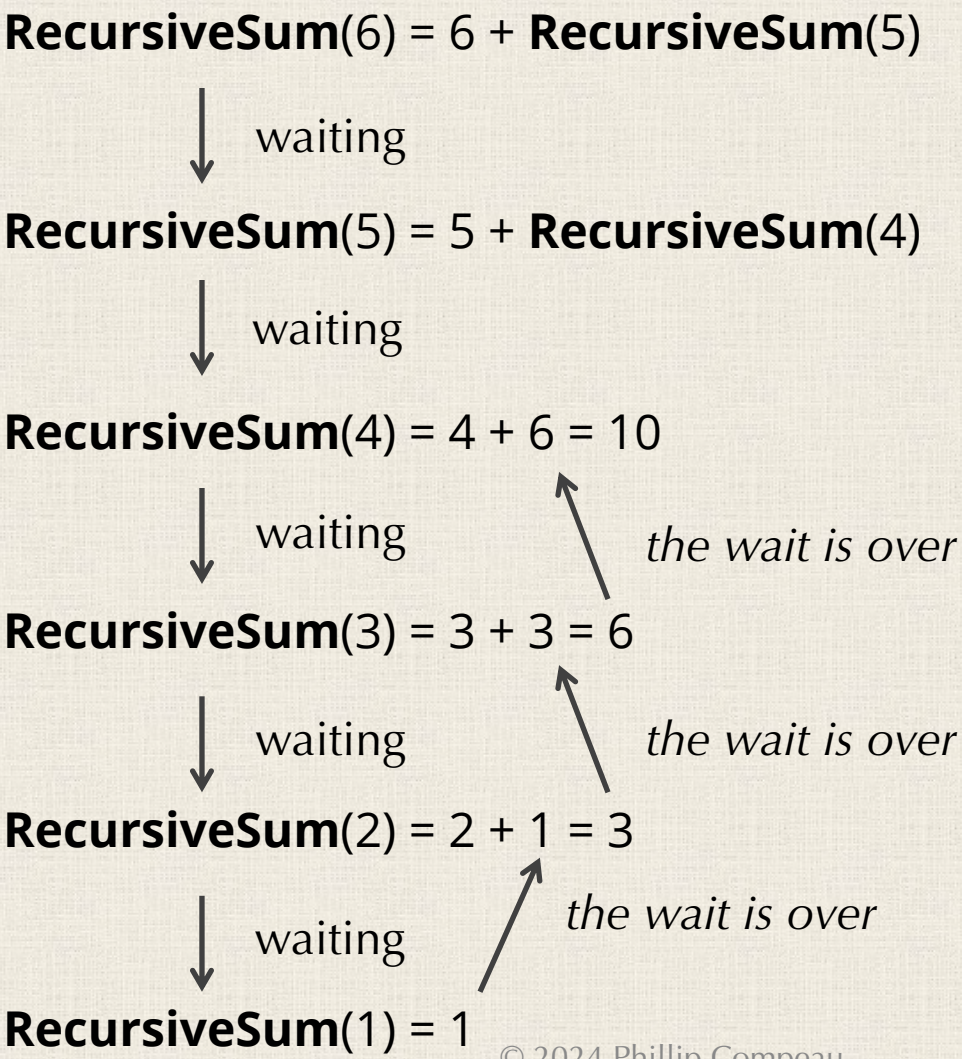
the wait is over

RecursiveSum(1) = 1

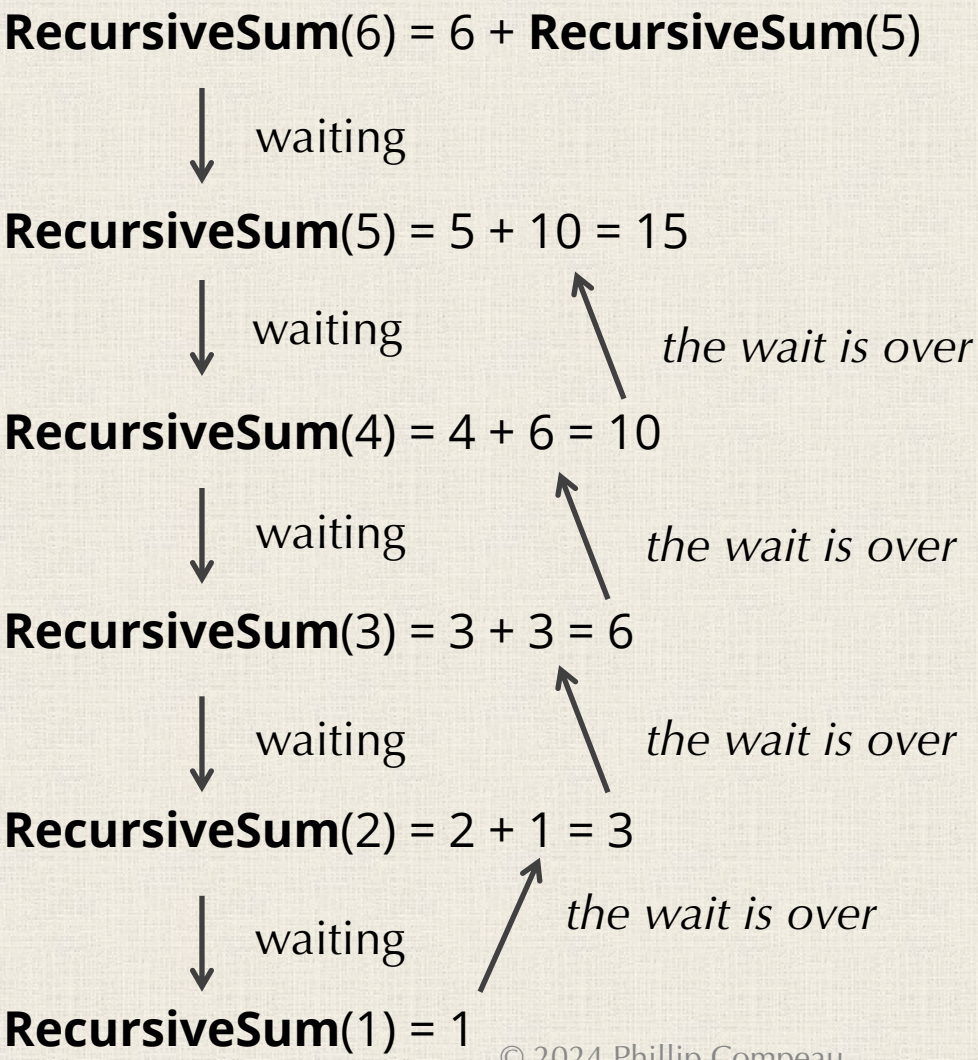
How Recursion Works



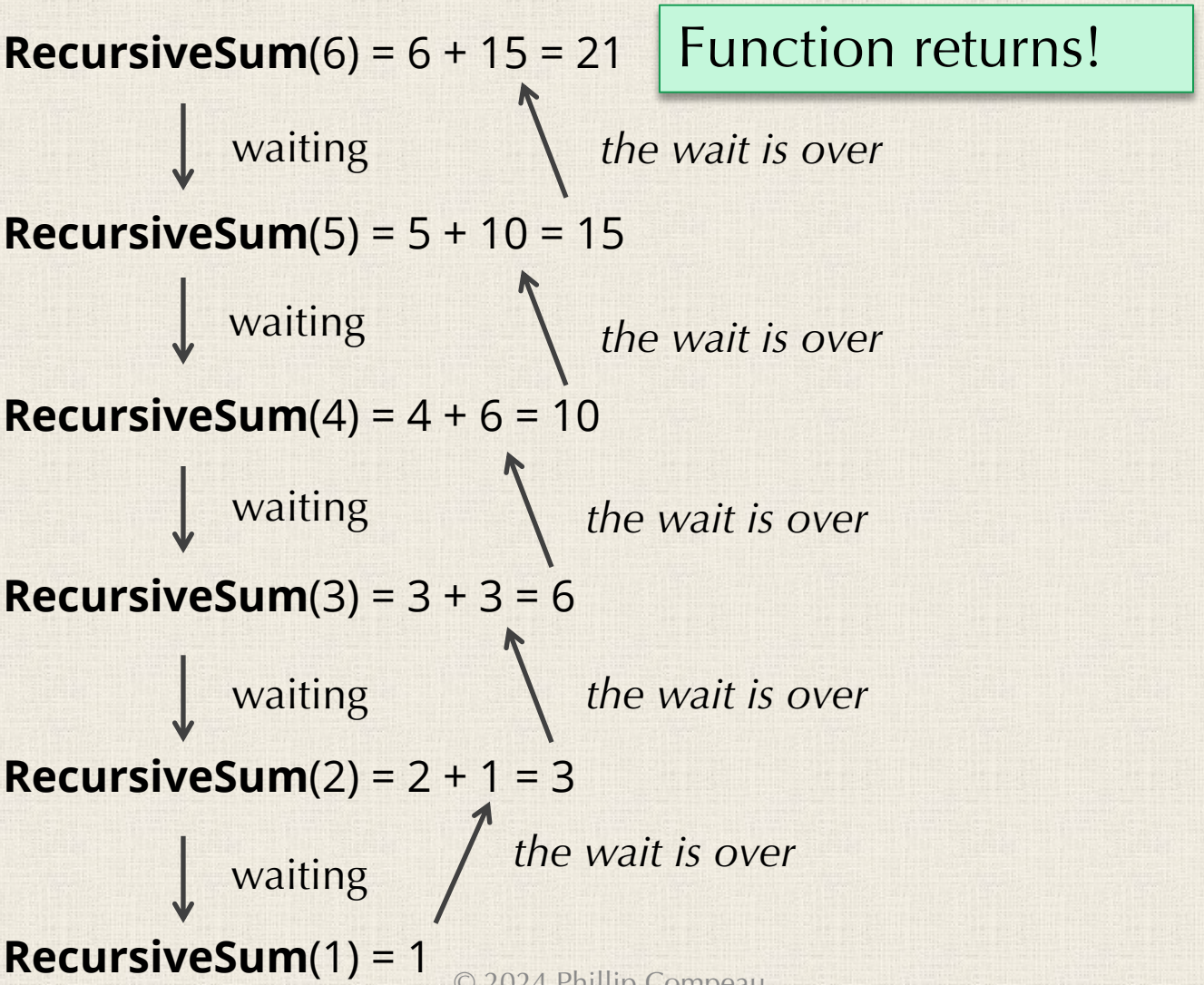
How Recursion Works



How Recursion Works



How Recursion Works



From Long Ago: Summing Integers

STOP: What happens if we forget the base case?

```
RecursiveSum(n)
        if n > 0
                return n + RecursiveSum(n-1)
```

From Long Ago: Summing Integers

STOP: What happens if we forget the base case?

```
RecursiveSum(n)
        if n > 0
                return n + RecursiveSum(n-1)
```

Answer: Once we hit **RecursiveSum**(0), there is no behavior that the function can take! (That is, without a base case, the stack will just sit there forever…)

Recursive Factorials

Exercise: Write pseudocode for a recursive function that takes an integer n as an argument and returns $n!$

Factorial Problem

- **Input:** An integer n.
- **Output:** $n!$, the product of the first n positive integers.

Recursive Factorials

Exercise: Write pseudocode for a recursive function that takes an integer *n* as an argument and returns *n*!

```
RecFact(n)
        if n = 0
                return 1
        else
                return n * RecFact(n – 1)
```

Recursive Factorials

Note: The order of the base case and inductive step don't really matter. This function will work just as well.

```
RecFact(n)
        if n > 0
                return n * RecFact(n – 1)
        else
                return 1
```

Recursive Fibonacci Numbers

Exercise: Write pseudocode for a recursive function that takes an integer n as an argument and returns the n-th Fibonacci number. Assume 0-based indexing.

If $Fib(n)$ is the n-th Fibonacci number, then

$$Fib(n) = Fib(n - 1) + Fib(n - 2)$$

Recursive Fibonacci Numbers

Exercise: Write pseudocode for a recursive function that takes an integer *n* as an argument and returns the *n*-th Fibonacci number. Assume 0-based indexing.

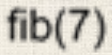
```
RecFib(n)
        if n = 0 or n = 1
                return 1
        else
                return RecFib(n-1) + RecFib(n-2)
```

Recursive Fibonacci Numbers

Exercise: Implement this function in the Go playground (play.golang.org) and call RecFib(30), then RecFib(60). What answers do you get?

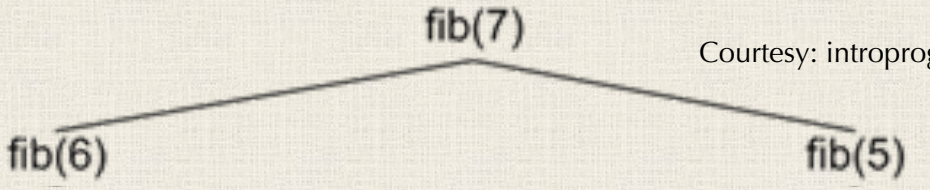
```
RecFib(n)
        if n = 0 or n = 1
                return 1
        else
                return RecFib(n-1) + RecFib(n-2)
```

Calling Fib(7) Shows the Problem with Using Recursion



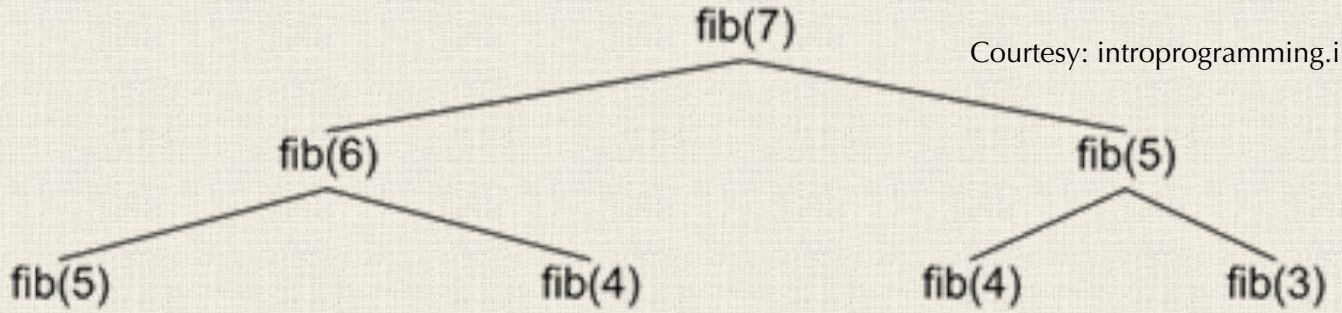
Courtesy: introprogramming.info

Calling Fib(7) Shows the Problem with Using Recursion



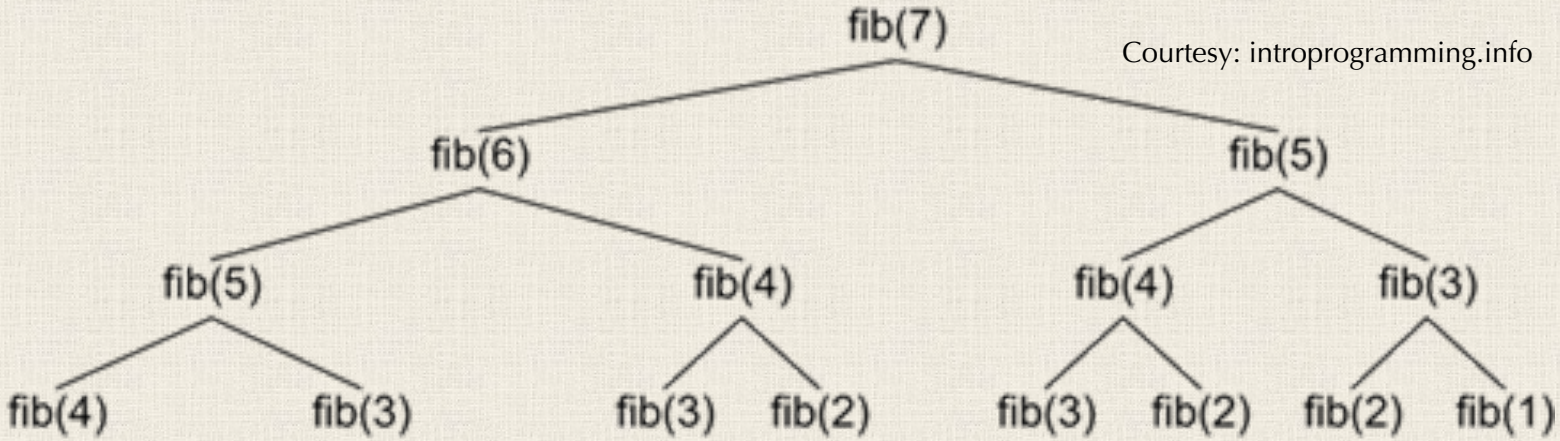
Courtesy: introprogramming.info

Calling Fib(7) Shows the Problem with Using Recursion



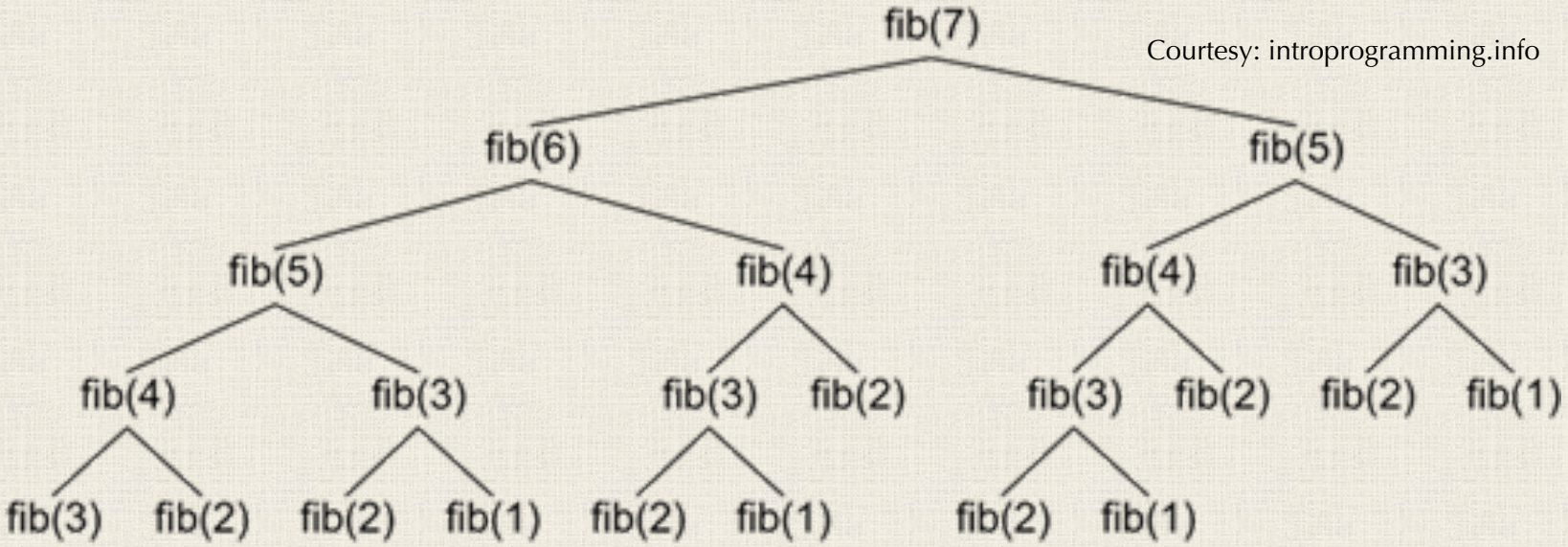
Courtesy: introprogramming.info

Calling Fib(7) Shows the Problem with Using Recursion



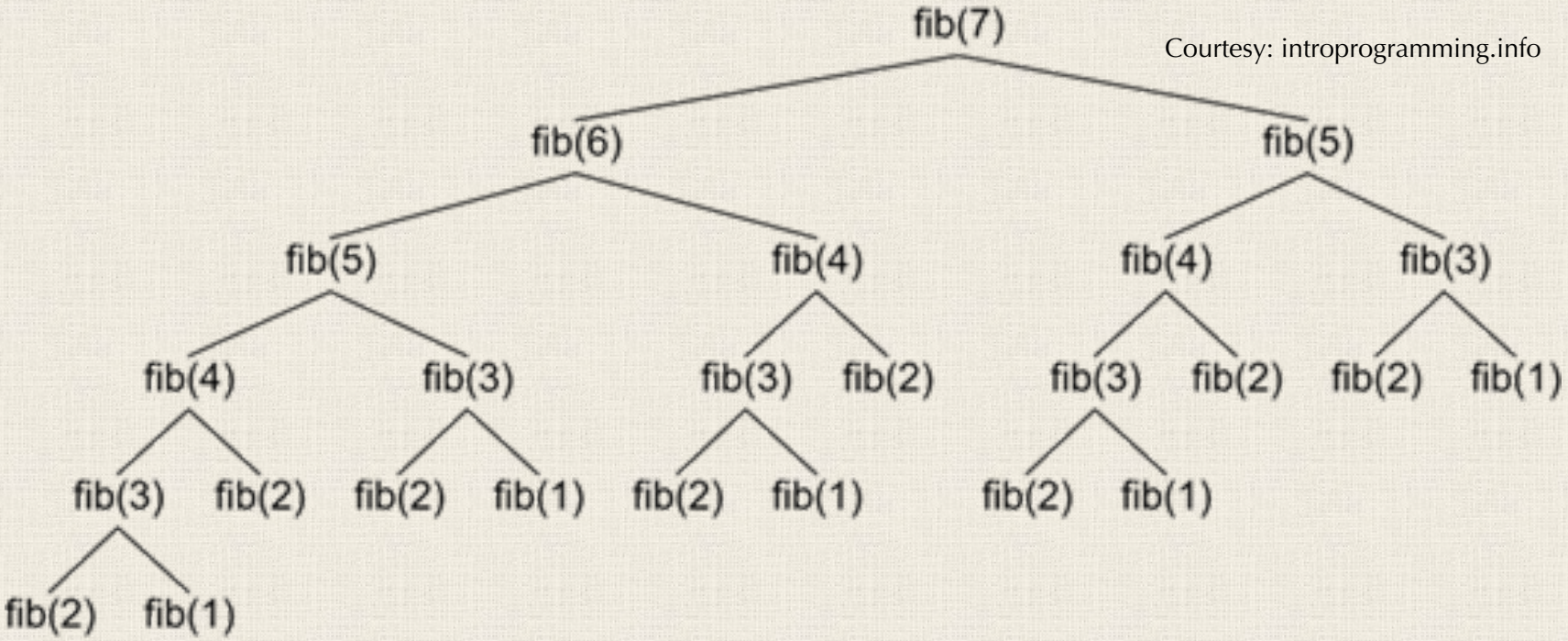
Courtesy: introprogramming.info

Calling Fib(7) Shows the Problem with Using Recursion



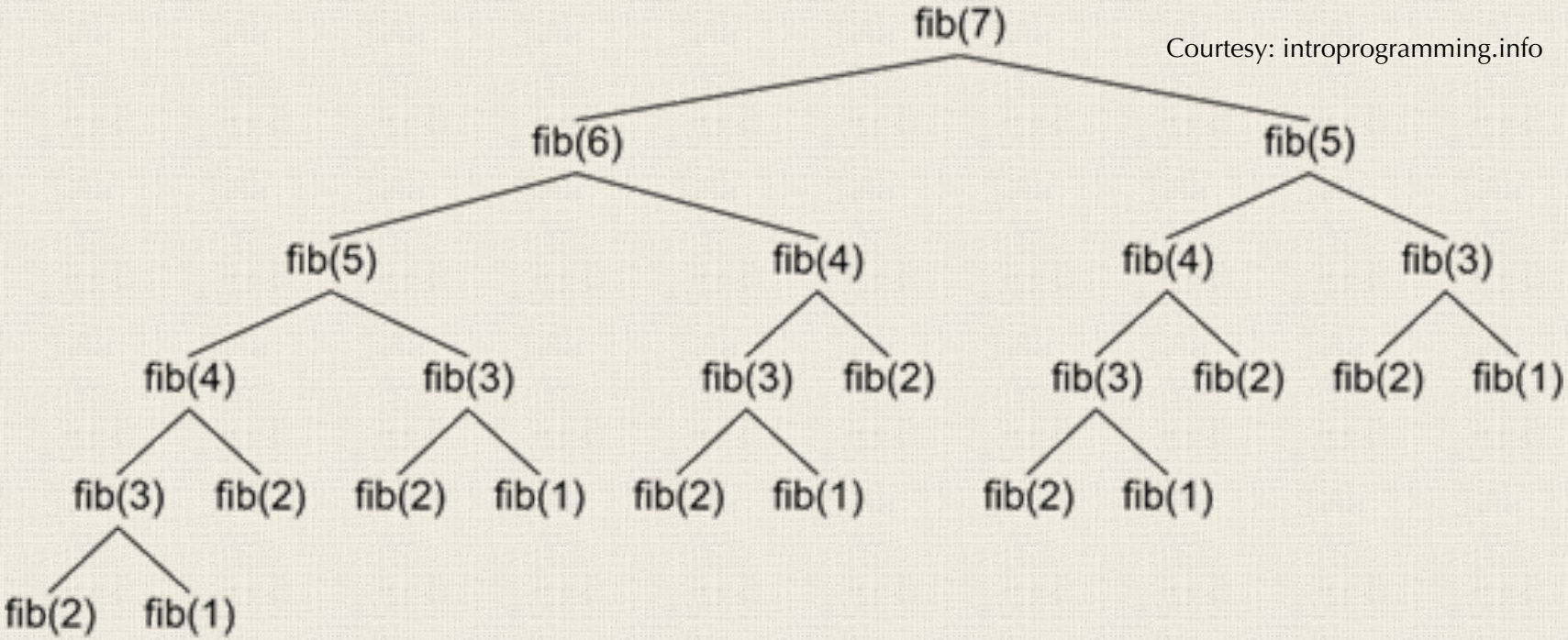
Courtesy: introprogramming.info

Calling Fib(7) Shows the Problem with Using Recursion



Courtesy: introprogramming.info

Calling Fib(7) Shows the Problem with Using Recursion



Courtesy: introprogramming.info

STOP: Approximately how many calls do you think are made for **RecFib**(20)? What about **RecFib**(45)?

The Issue with Fibonacci Recursion

When we call RecFib(n), there are ~ 2^n calls on the stack. For most values of n, this will exhaust the memory allocated to the stack and produce what is called **stack overflow**, crashing the program.

The Issue with Fibonacci Recursion

When we call RecFib(n), there are ~ 2^n calls on the stack. For most values of n, this will exhaust the memory allocated to the stack and produce what is called **stack overflow**, crashing the program.

Moral: We should evaluate whether recursion is a good approach for solving a problem based on whether there is a chance of stack overflow.

Computing Values "Bottom-Up" Avoids Many Recursive Calls

Instead of computing Fibonacci numbers top-down recursively, we compute them bottom-up.


```
Fibonacci(n)
        a ← array of length n
        a[0] ← 1
        a[1] ← 1
        for i ← 2 to n
                a[i] ← a[i-1] + a[i-2]
        return a
```

Computing Values "Bottom-Up" Avoids Many Recursive Calls

Instead of computing Fibonacci numbers top-down recursively, we compute them bottom-up.


```
Fibonacci(n)
        a ← array of length n
        a[0] ← 1
        a[1] ← 1
        for i ← 2 to n
                a[i] ← a[i-1] + a[i-2]
        return a
```

Computing Values "Bottom-Up" Avoids Many Recursive Calls

Instead of computing Fibonacci numbers top-down recursively, we compute them bottom-up.

Fibonacci(*n*)
    a ← array of length *n*
    a[0] ← 1
    a[1] ← 1
    for *i* ← 2 to *n*
        a[*i*] ← *a*[*i*-1] + *a*[*i*-2]
    return *a*

Computing Values "Bottom-Up" Avoids Many Recursive Calls

Instead of computing Fibonacci numbers top-down recursively, we compute them bottom-up.

Fibonacci(*n*)
    a ← array of length *n*
    a[0] ← 1
    a[1] ← 1
    for *i* ← 2 to *n*
        a[*i*] ← *a*[*i*-1] + *a*[*i*-2]
    return *a*

Computing Values "Bottom-Up" Avoids Many Recursive Calls

Instead of computing Fibonacci numbers top-down recursively, we compute them bottom-up.

Fibonacci(*n*)
    a ← array of length *n*
    a[0] ← 1
    a[1] ← 1
    for *i* ← 2 to *n*
        a[*i*] ← *a*[*i*-1] + *a*[*i*-2]
    return *a*

Computing Values "Bottom-Up" Avoids Many Recursive Calls

Instead of computing Fibonacci numbers top-down recursively, we compute them bottom-up.

1	1	2	3	5	8	13				*a*

Fibonacci(*n*)
    a ← array of length *n*
    a[0] ← 1
    a[1] ← 1
    for *i* ← 2 to *n*
        a[*i*] ← *a*[*i*-1] + *a*[*i*-2]
    return *a*

Computing Values "Bottom-Up" Avoids Many Recursive Calls

Instead of computing Fibonacci numbers top-down recursively, we compute them bottom-up.

1	1	2	3	5	8	13	21		

a

Fibonacci(*n*)
    a ← array of length *n*
    a[0] ← 1
    a[1] ← 1
    for *i* ← 2 to *n*
        a[*i*] ← *a*[*i*-1] + *a*[*i*-2]
    return *a*

Computing Values "Bottom-Up" Avoids Many Recursive Calls

Instead of computing Fibonacci numbers top-down recursively, we compute them bottom-up.

1	1	2	3	5	8	13	21	34	

a

Fibonacci(*n*)
    a ← array of length *n*
    a[0] ← 1
    a[1] ← 1
    for *i* ← 2 to *n*
        a[*i*] ← *a*[*i*-1] + *a*[*i*-2]
    return *a*

Computing Values "Bottom-Up" Avoids Many Recursive Calls

Instead of computing Fibonacci numbers top-down recursively, we compute them bottom-up.

1	1	2	3	5	8	13	21	34	55

a

Fibonacci(*n*)
    a ← array of length *n*
    a[0] ← 1
    a[1] ← 1
    for *i* ← 2 to *n*
        a[*i*] ← *a*[*i*-1] + *a*[*i*-2]
    return *a*

Computing Values "Bottom-Up" Avoids Many Recursive Calls

Computing a recurrence relation bottom-up using an array is called **dynamic programming**.

1	1	2	3	5	8	13	21	34	55

a

Fibonacci(*n*)
    a ← array of length *n*
    a[0] ← 1
    a[1] ← 1
    for *i* ← 2 to *n*
        a[*i*] ← *a*[*i*-1] + *a*[*i*-2]
    return *a*

Computing Fibonacci Numbers

Computing a recurrence relation bottom-up using an array is called **dynamic programming**.

STOP: Wait … why would such a simple idea be called "dynamic programming"?

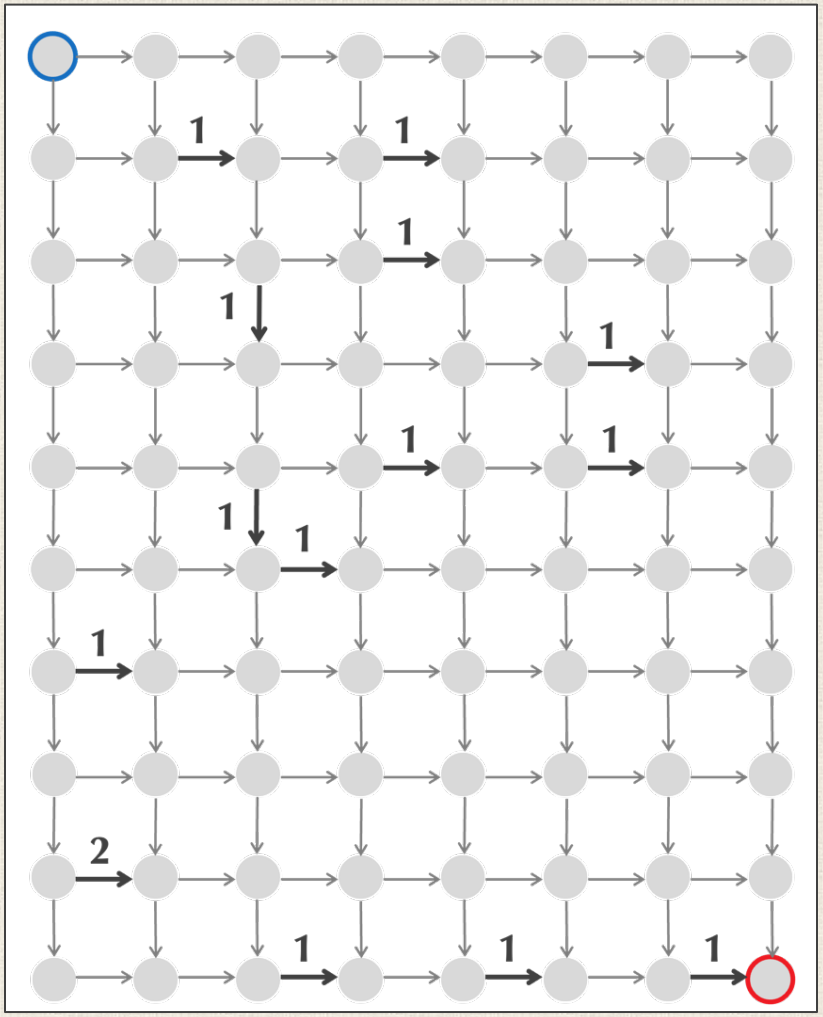
Richard Bellman

Richard Bellman, a Wise Man

“We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word "research". I’m not using the term lightly; I’m using it precisely. His face would suffuse, he would turn red, and he would get violent if people used the term research in his presence. You can imagine how he felt, then, about the term mathematical. The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose? In the first place I was interested in planning, in decision making, in thinking. But planning, is not a good word for various reasons. I decided therefore to use the word "programming". I wanted to get across the idea that this was dynamic, this was multistage, this was time-varying. I thought, let's kill two birds with one stone. Let's take a word that has an absolutely precise meaning, namely dynamic, in the classical physical sense. It also has a very interesting property as an adjective, and that is it's impossible to use the word dynamic in a pejorative sense. Try thinking of some combination that will possibly give it a pejorative meaning. It's impossible. Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities.”

RETURNING TO MANHATTAN

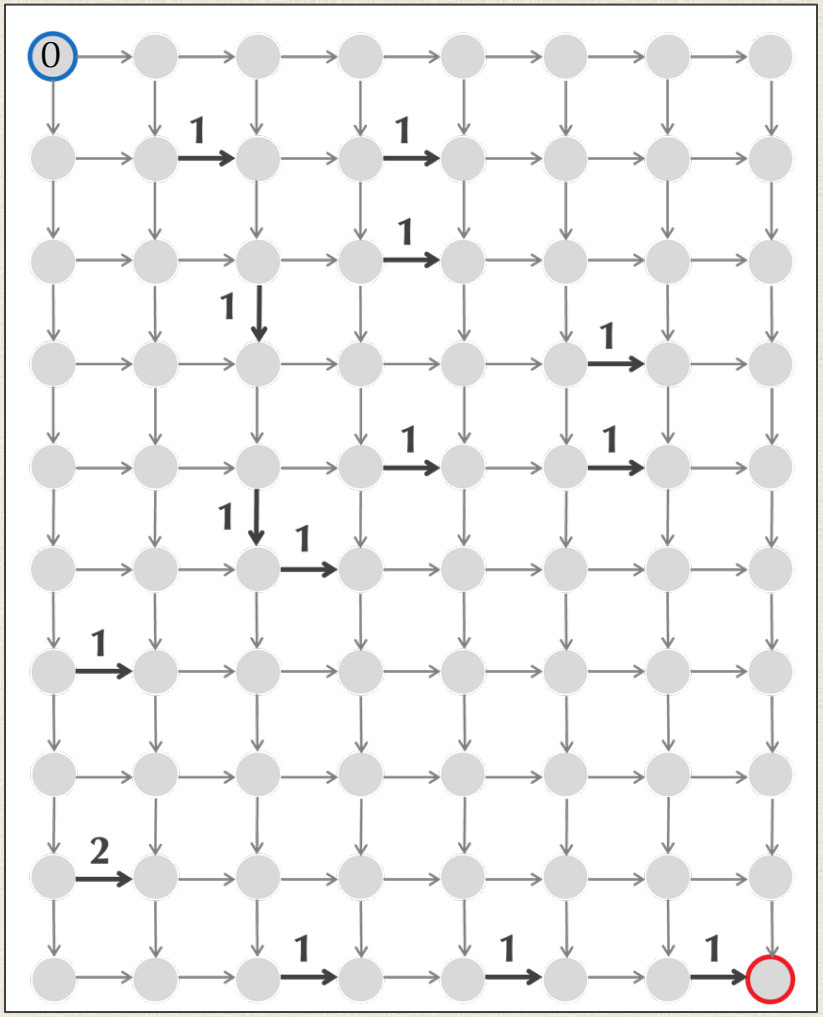
We can now use dynamic programming to find the best Manhattan tourist path



Define $s(i, j)$ as the score of a maximum-weight path from the source to node (i, j).

We start with $s(0, 0)$, which must be equal to zero.

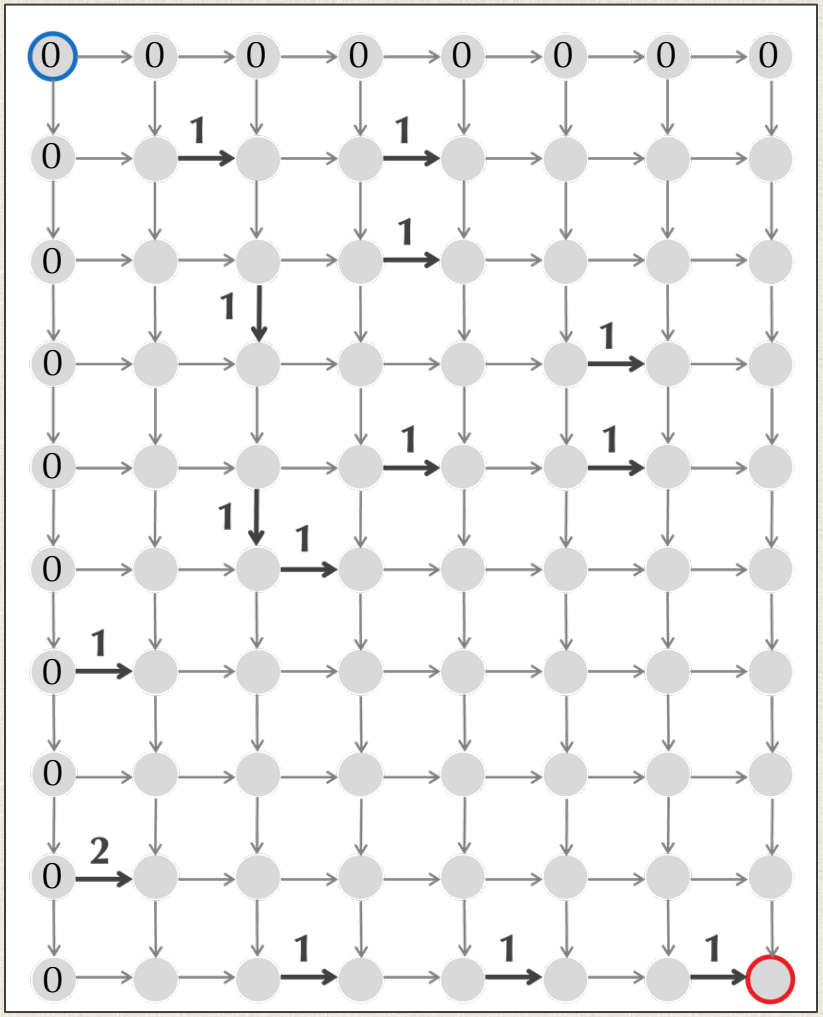
We can now use dynamic programming to find the best Manhattan tourist path



Define $s(i, j)$ as the score of a maximum-weight path from the source to node (i, j).

STOP: Which other values of $s(i, j)$ are easy to set?

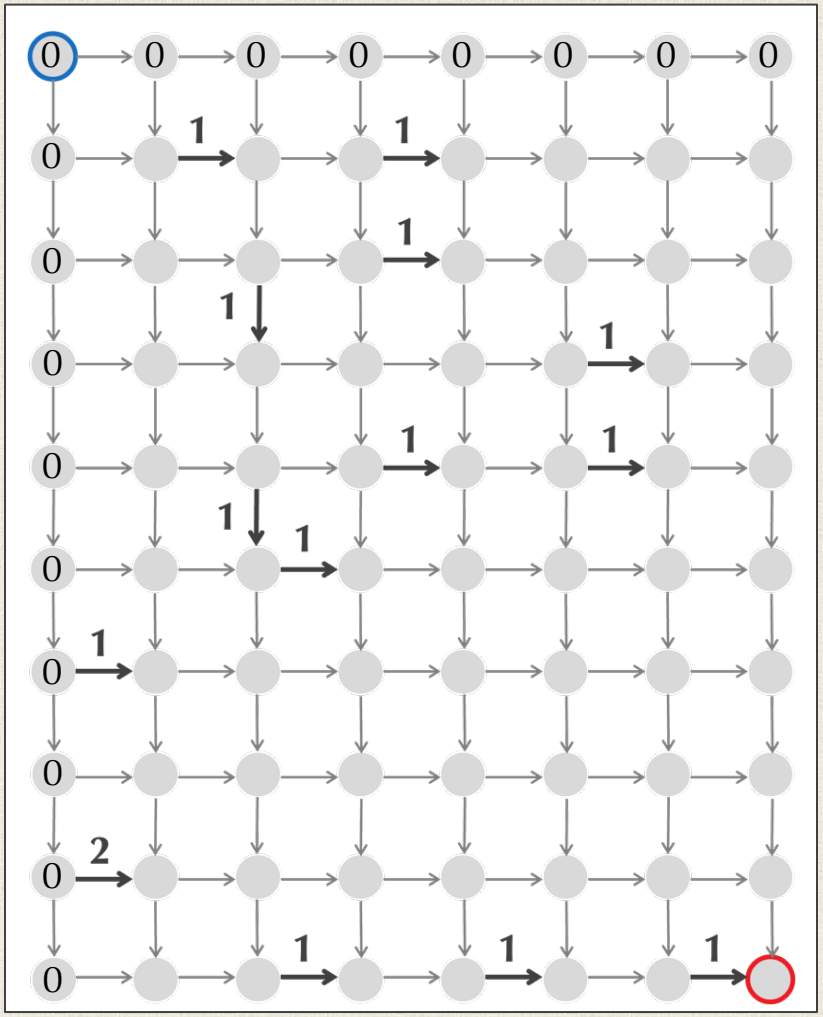
We can now use dynamic programming to find the best Manhattan tourist path



Define $s(i, j)$ as the score of a maximum-weight path from the source to node (i, j).

Answer: we can set the scores of the entire first row and column because there is only one path into these nodes.

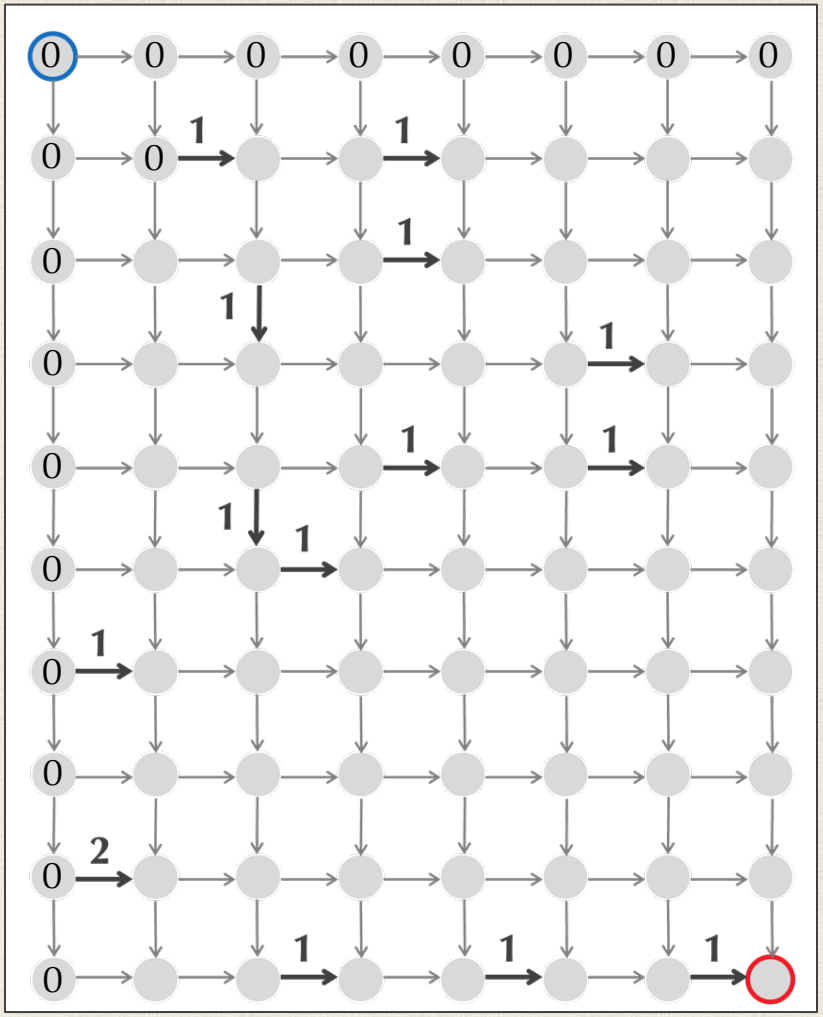
We can now use dynamic programming to find the best Manhattan tourist path



Define $s(i, j)$ as the score of a maximum-weight path from the source to node (i, j).

STOP: Which value(s) of $s(i, j)$ can we set now? Why?

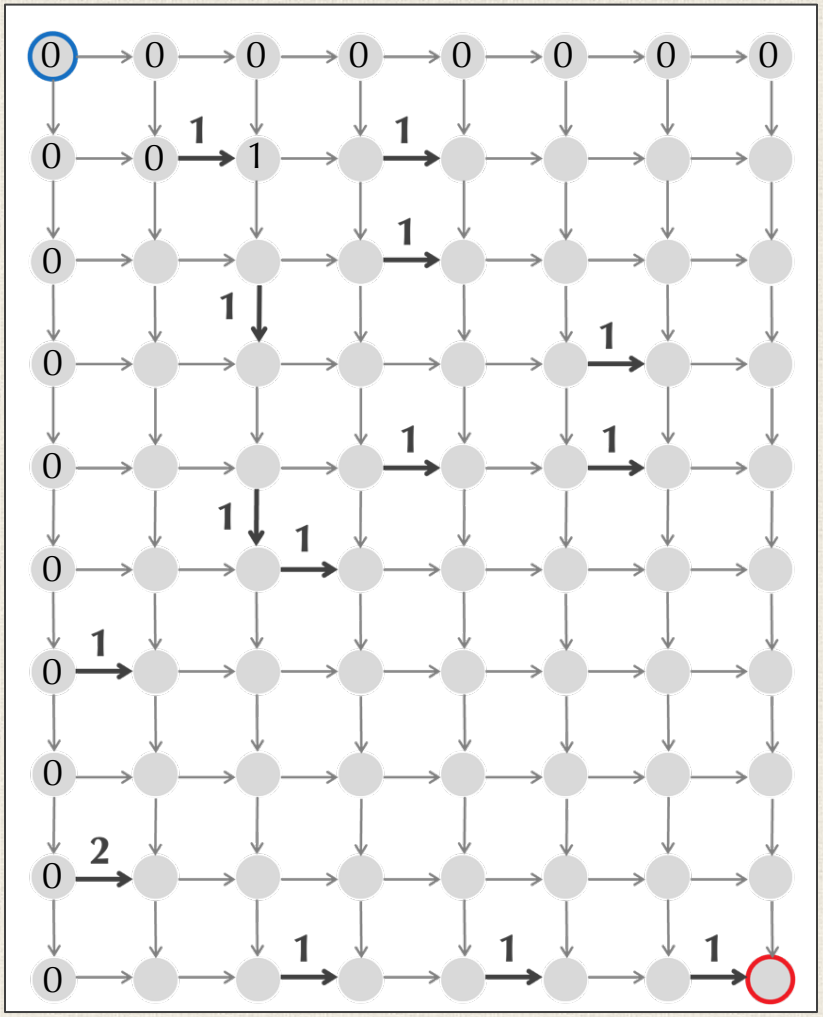
We can now use dynamic programming to find the best Manhattan tourist path



Define $s(i, j)$ as the score of a maximum-weight path from the source to node (i, j).

Answer: we can only set $s(1, 1)$ because we know that it must come from either the node above or the node below.

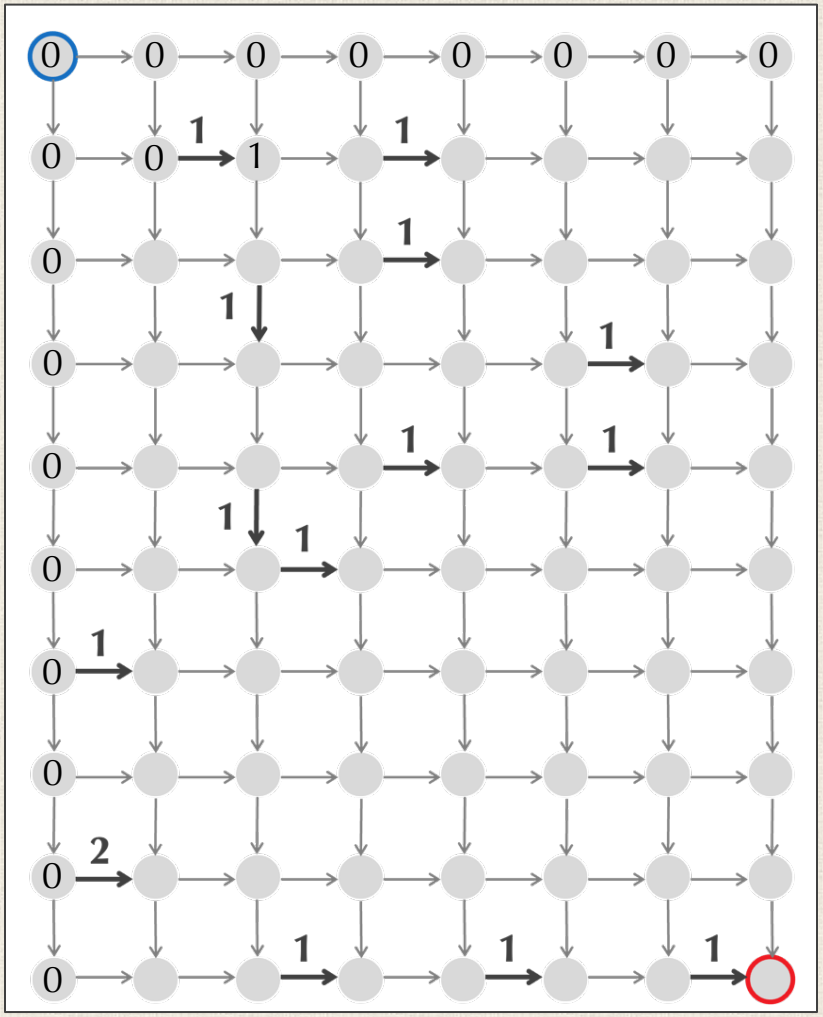
We can now use dynamic programming to find the best Manhattan tourist path



Define $s(i, j)$ as the score of a maximum-weight path from the source to node (i, j).

We can now set $s(1, 2)$ because the best path into (1, 2) must come from (0, 2) or (1, 2): $s(1, 2) = s(1, 1) + 1 = 1$.

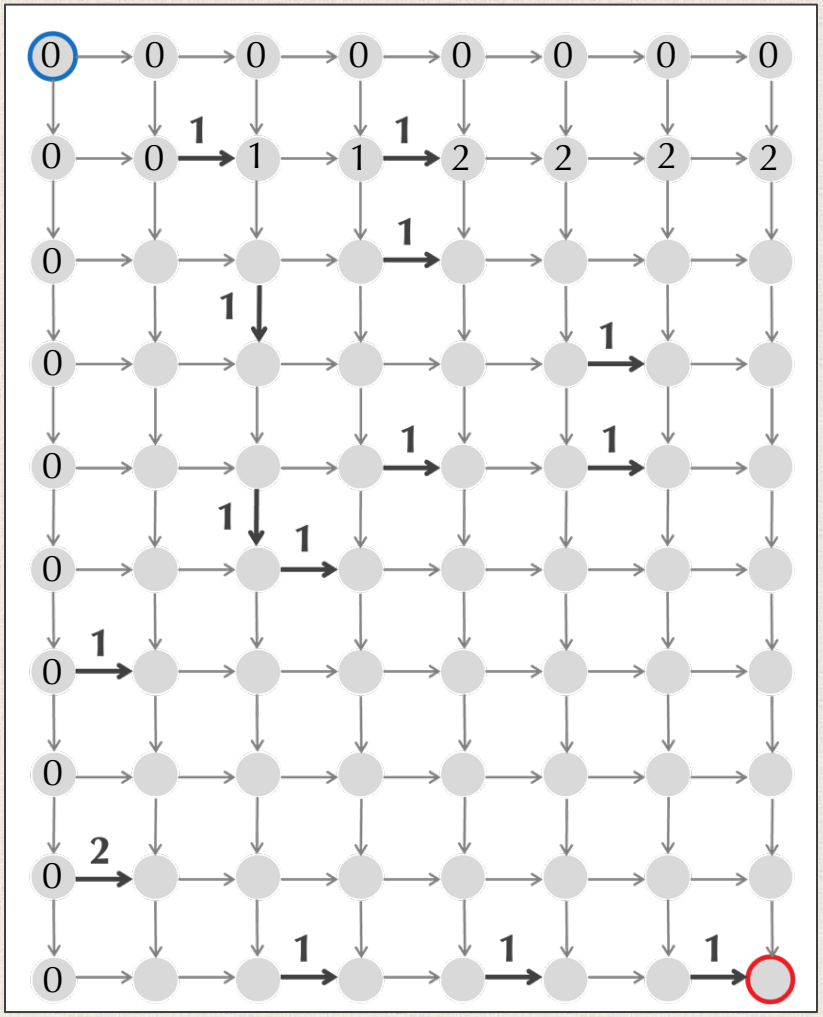
We can now use dynamic programming to find the best Manhattan tourist path



Define $s(i, j)$ as the score of a maximum-weight path from the source to node (i, j).

Exercise: Fill in the rest of row 1.

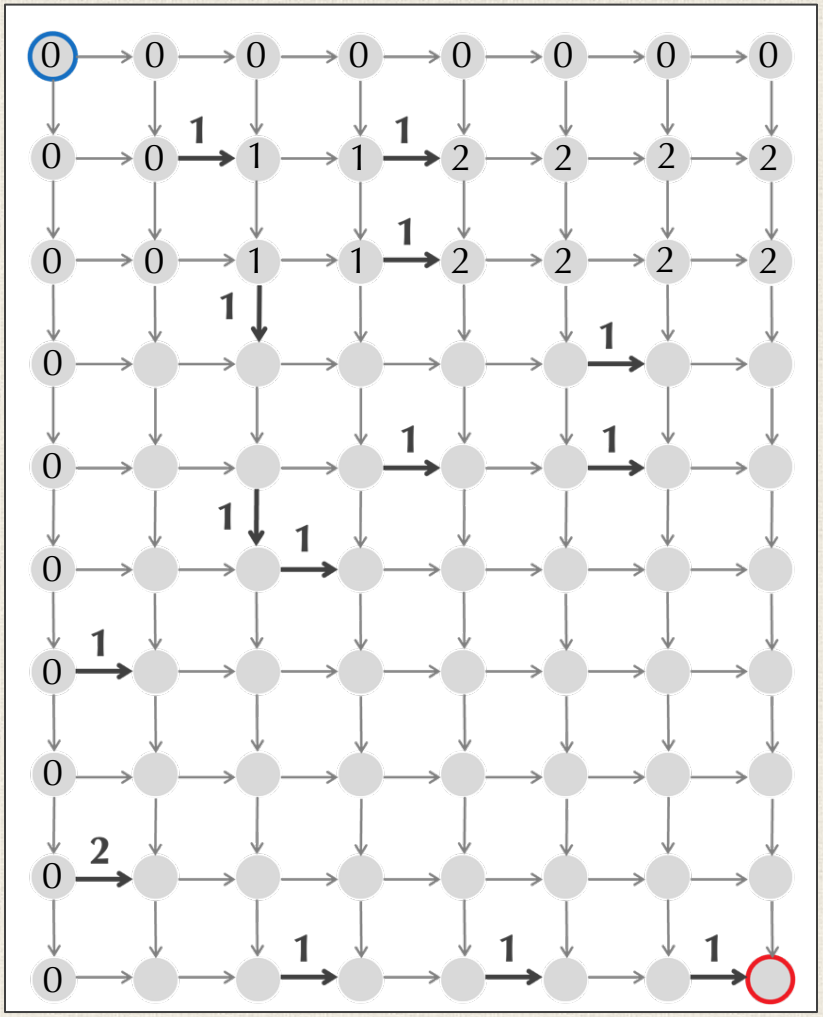
We can now use dynamic programming to find the best Manhattan tourist path



Define $s(i, j)$ as the score of a maximum-weight path from the source to node (i, j).

Exercise: Fill in the rest of row 1.

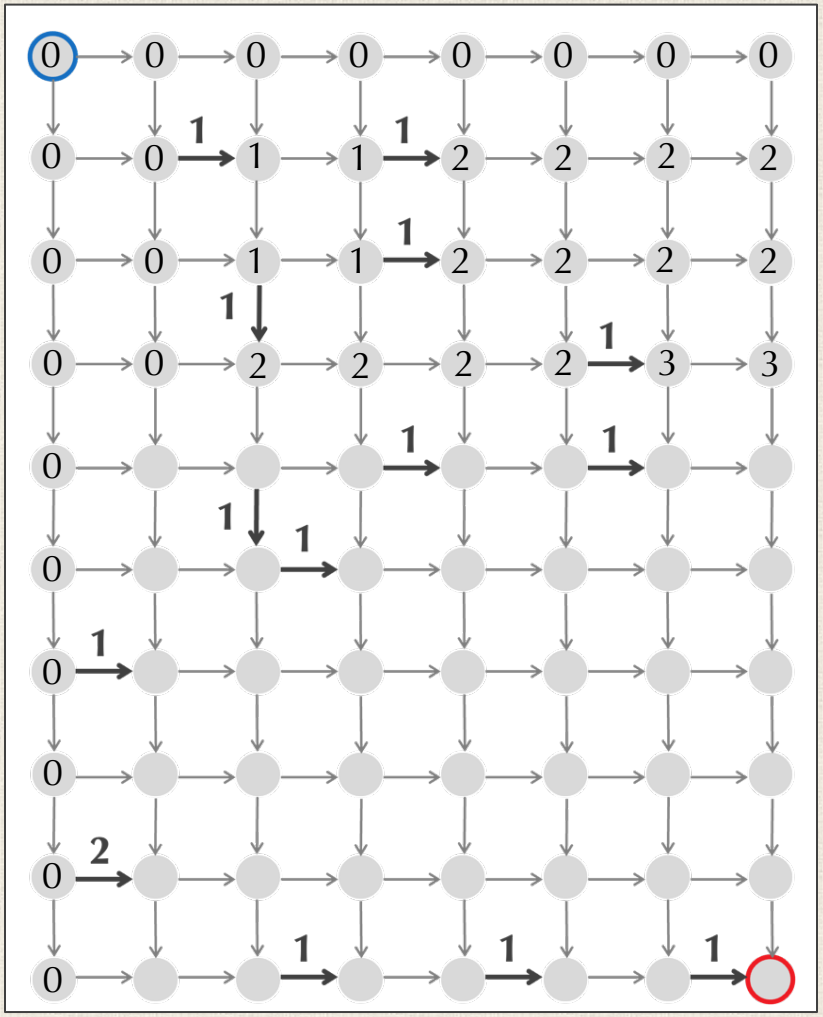
We can now use dynamic programming to find the best Manhattan tourist path



Define $s(i, j)$ as the score of a maximum-weight path from the source to node (i, j).

In this way, we can fill in values of $s(i, j)$ row-by-row.

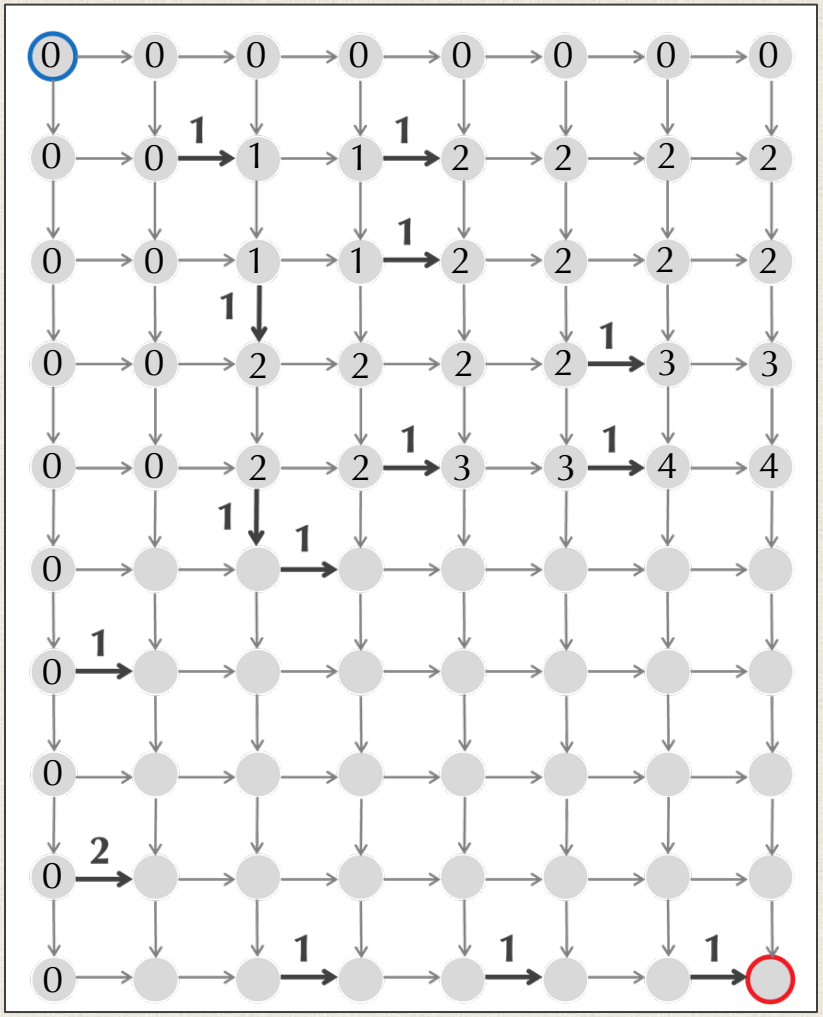
We can now use dynamic programming to find the best Manhattan tourist path



Define $s(i, j)$ as the score of a maximum-weight path from the source to node (i, j).

In this way, we can fill in values of $s(i, j)$ row-by-row.

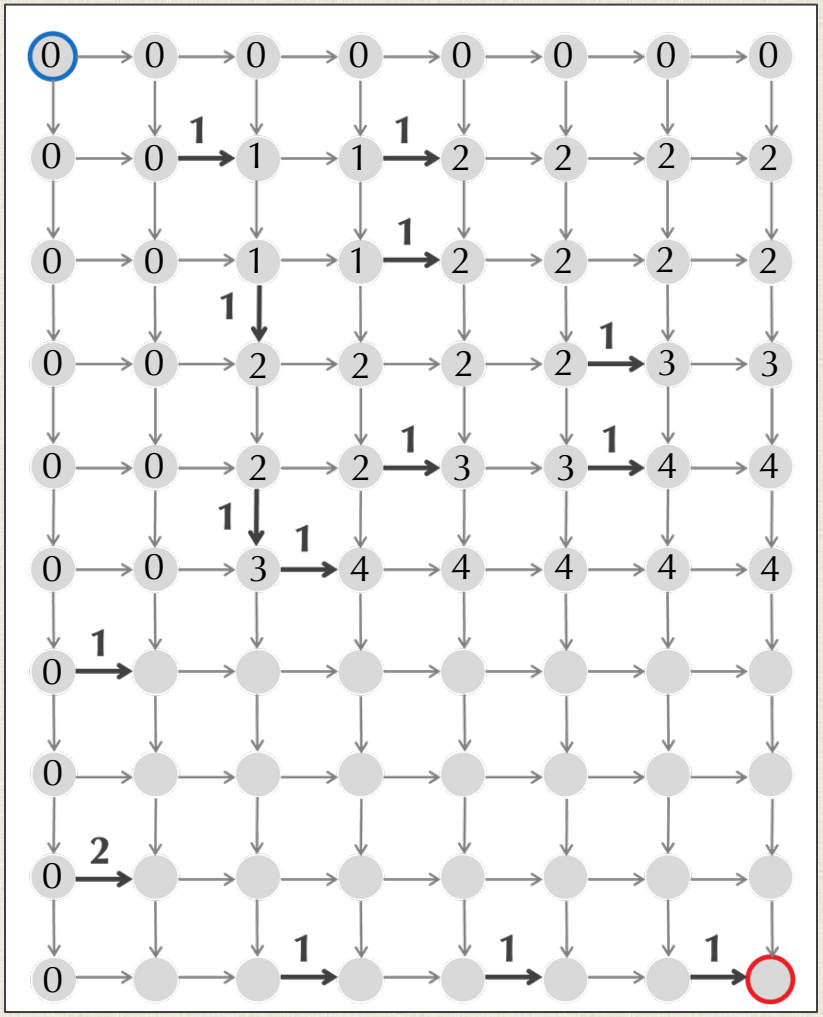
We can now use dynamic programming to find the best Manhattan tourist path



Define $s(i, j)$ as the score of a maximum-weight path from the source to node (i, j).

In this way, we can fill in values of $s(i, j)$ row-by-row.

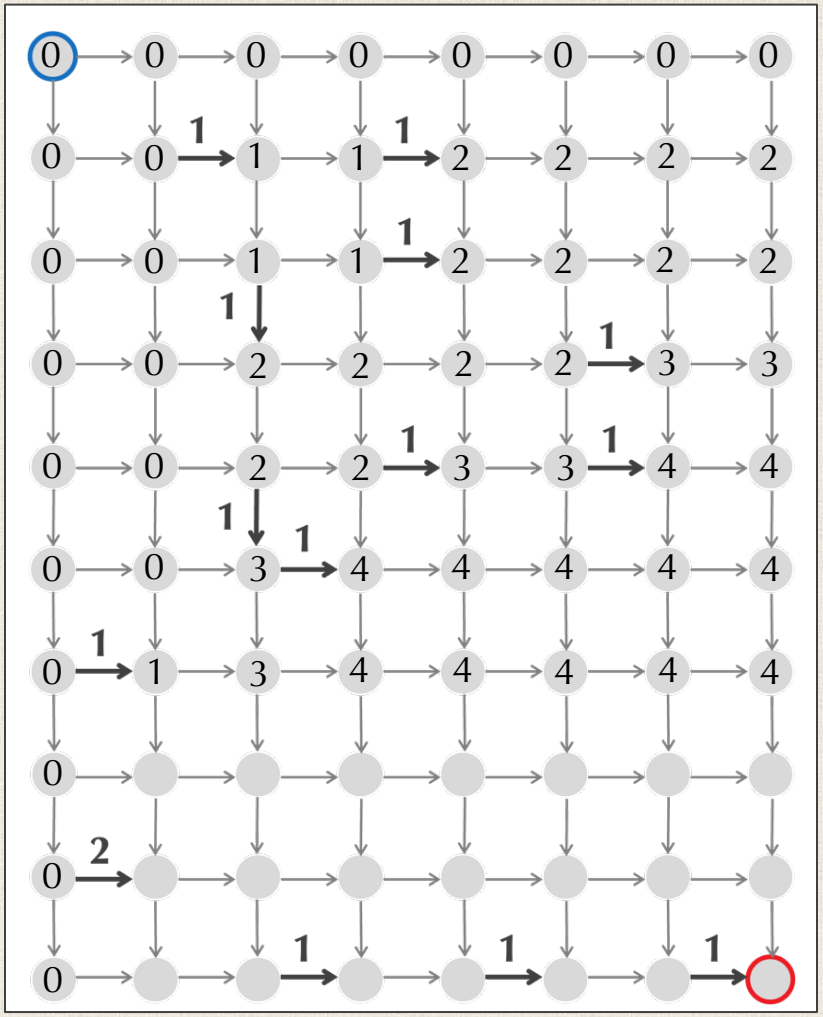
We can now use dynamic programming to find the best Manhattan tourist path



Define $s(i, j)$ as the score of a maximum-weight path from the source to node (i, j).

In this way, we can fill in values of $s(i, j)$ row-by-row.

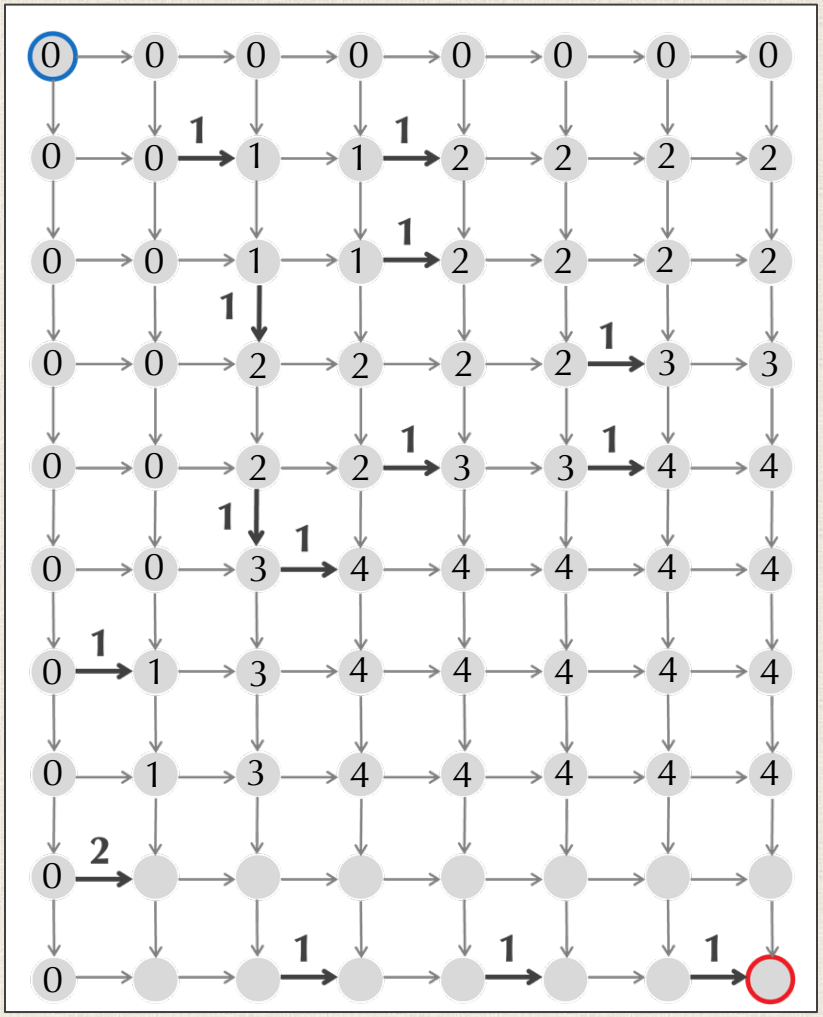
We can now use dynamic programming to find the best Manhattan tourist path



Define $s(i, j)$ as the score of a maximum-weight path from the source to node (i, j).

In this way, we can fill in values of $s(i, j)$ row-by-row.

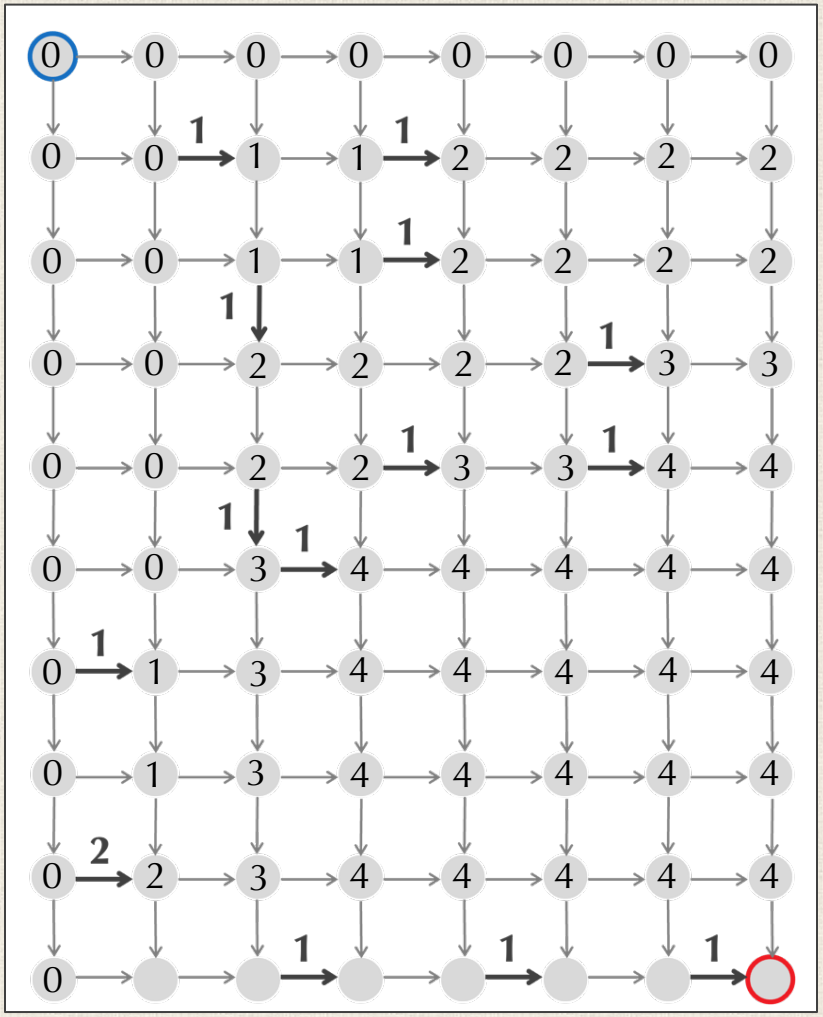
We can now use dynamic programming to find the best Manhattan tourist path



Define $s(i, j)$ as the score of a maximum-weight path from the source to node (i, j).

In this way, we can fill in values of $s(i, j)$ row-by-row.

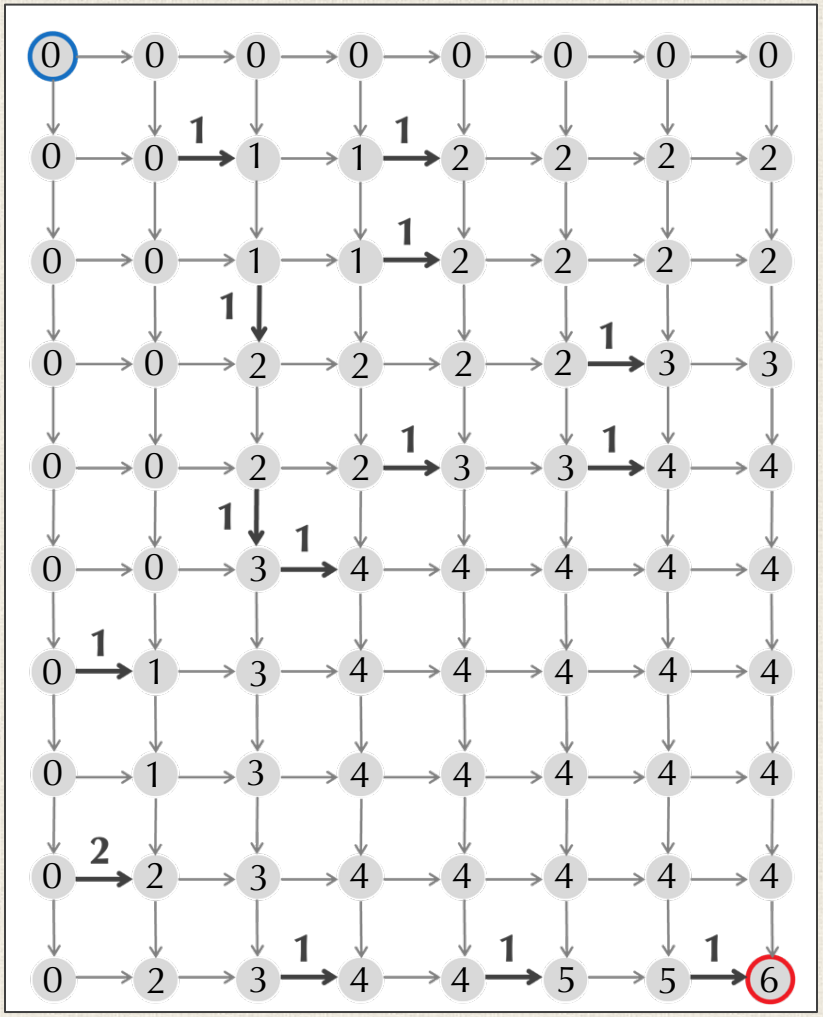
We can now use dynamic programming to find the best Manhattan tourist path



Define $s(i, j)$ as the score of a maximum-weight path from the source to node (i, j).

In this way, we can fill in values of $s(i, j)$ row-by-row.

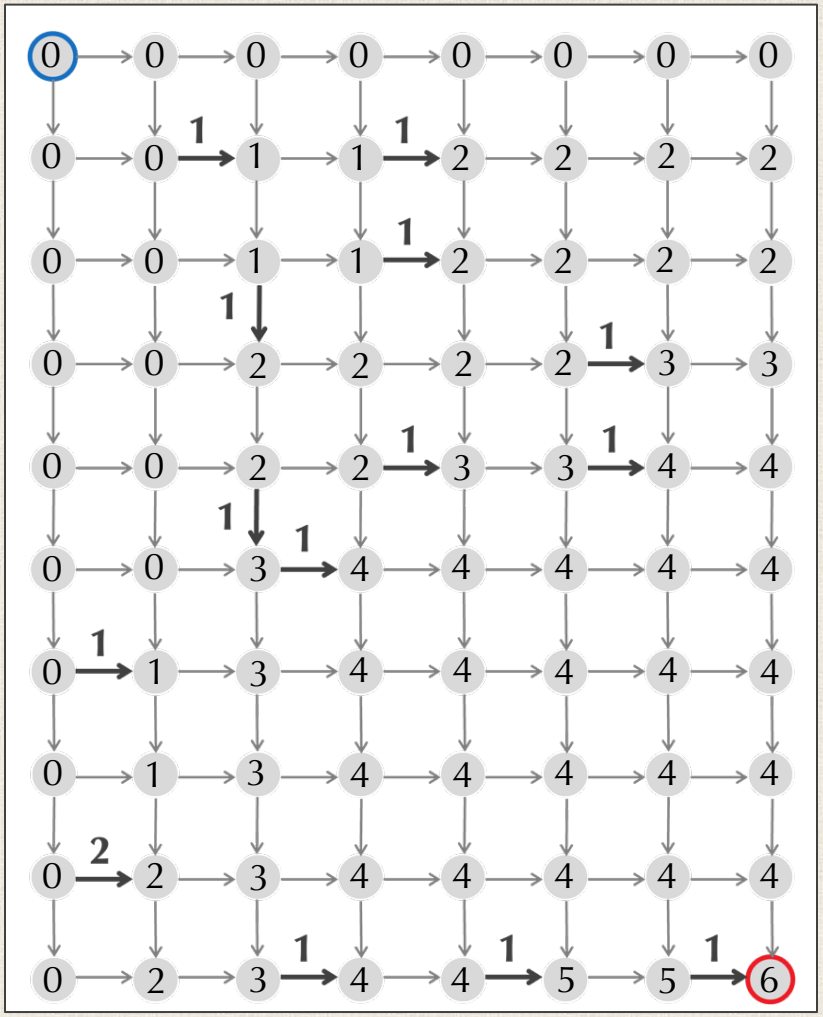
We can now use dynamic programming to find the best Manhattan tourist path



Define $s(i, j)$ as the score of a maximum-weight path from the source to node (i, j).

In this way, we can fill in values of $s(i, j)$ row-by-row.

We can now use dynamic programming to find the best Manhattan tourist path



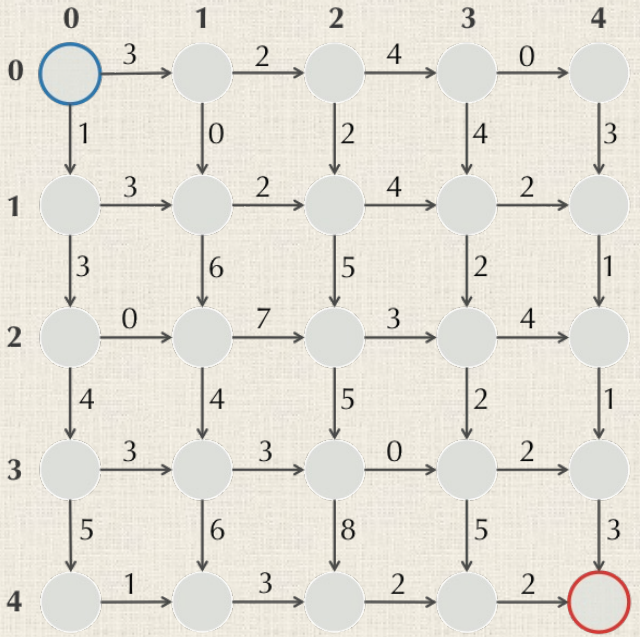
Define $s(i, j)$ as the score of a maximum-weight path from the source to node (i, j).

Key point: We now have a concrete dynamic programming algorithm that will find the longest path in any rectangular grid!

Generalizing to an Arbitrary Manhattan

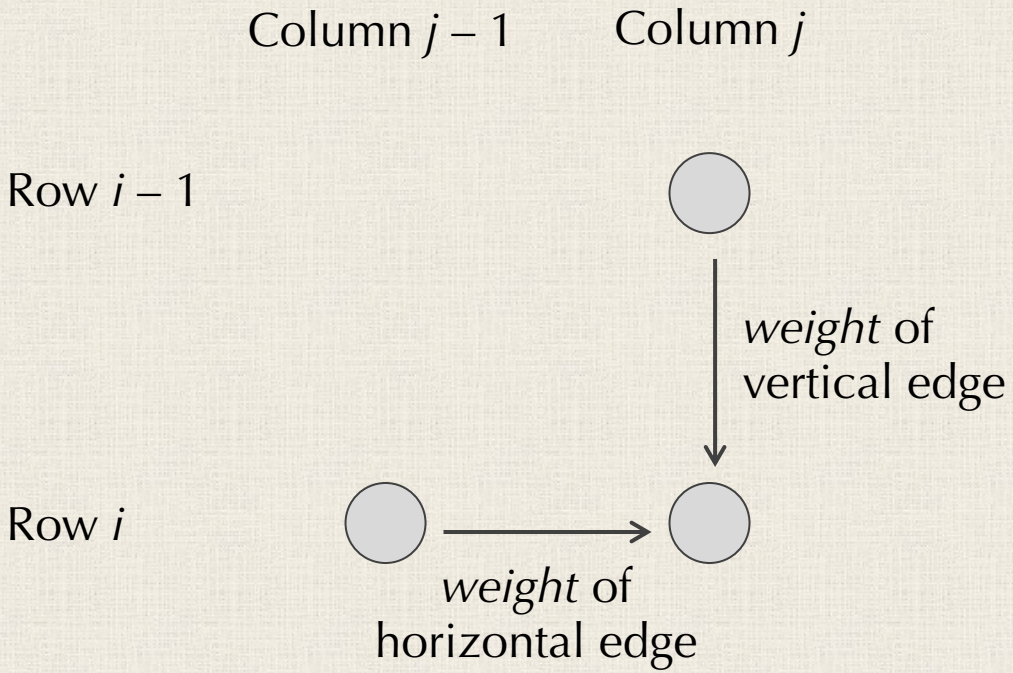
Manhattan Tourist Problem:

- **Input:** A weighted n x m rectangular grid (n + 1 rows and m + 1 columns).
- **Output:** A longest path from source (0, 0) to sink (n, m) in the grid.



Exercise: Find a recurrence relation for the length of a longest path from (0,0) to node (i, j), which we will call *length*(i,j).

Generalizing to an Arbitrary Manhattan

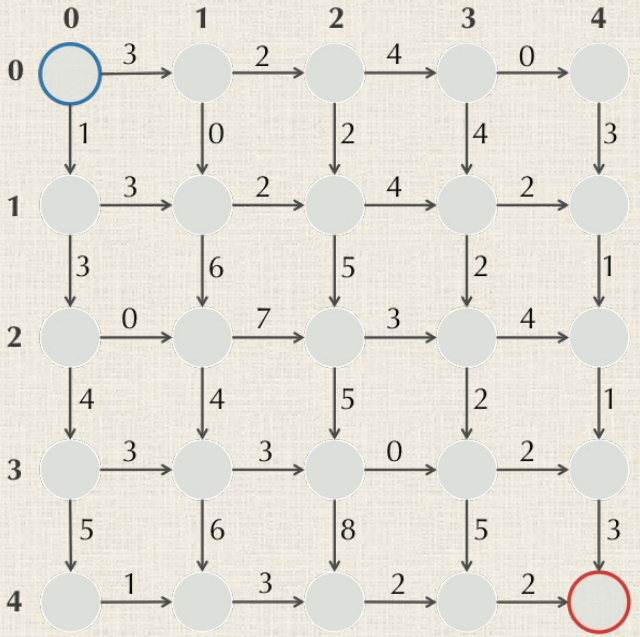


Answer: $length(i,j) = \max\{$
$length(i-1,j) + \text{weight(vertical edge into } i,j),$
$length(i, j-1) + \text{weight(horizontal edge into } i,j)\}.$

Generalizing to an Arbitrary Manhattan

Manhattan Tourist Problem:

- **Input:** A weighted n x m rectangular grid (n + 1 rows and m + 1 columns).
- **Output:** A longest path from source (0, 0) to sink (n, m) in the grid.

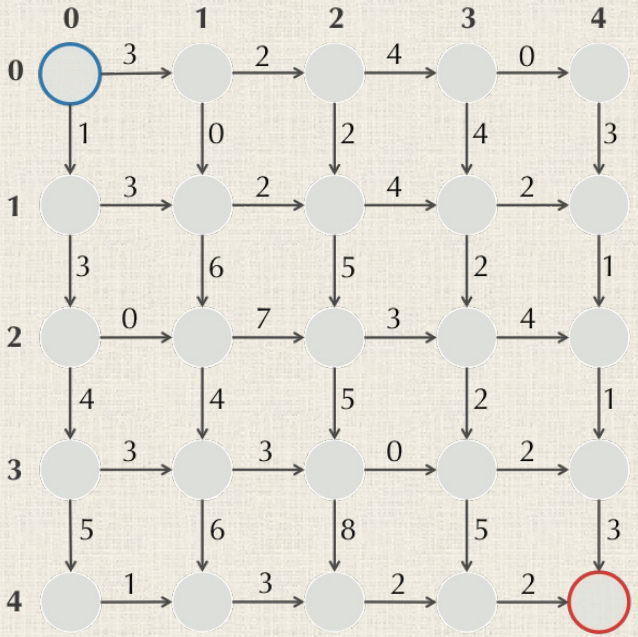


STOP: Will a recursive algorithm for Manhattan Tourist have the same problem that the recursive fibonacci function encountered?

Generalizing to an Arbitrary Manhattan

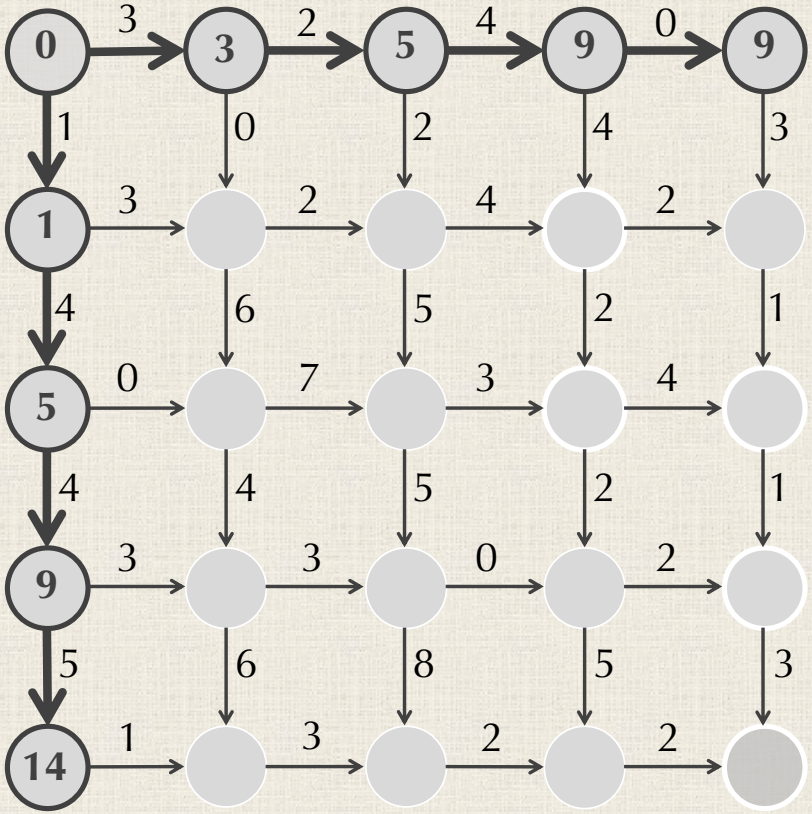
Manhattan Tourist Problem:

- **Input:** A weighted n x m rectangular grid (n + 1 rows and m + 1 columns).
- **Output:** A longest path from source (0, 0) to sink (n, m) in the grid.



Answer: Yes! Because the same *length*(i, j) can get re-computed many times…

Let's Use Dynamic Programming Instead

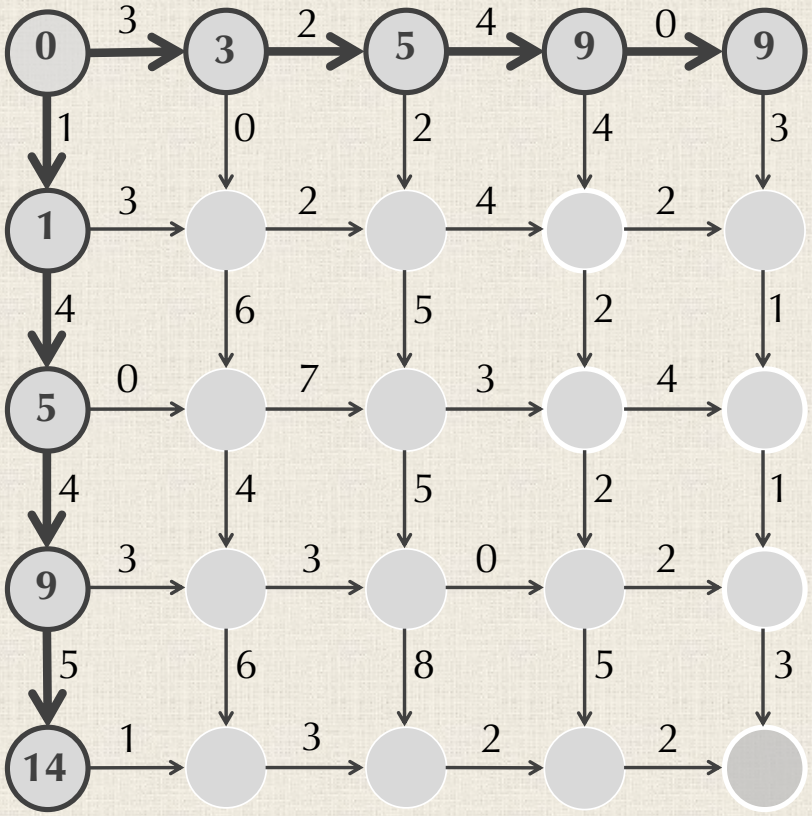


Recurrence relation

$$length(i, j) = \max\{length(i\text{-}1, j) + down(i, j),\ length(i, j\text{-}1) + right(i, j)\}$$

Let's Use Dynamic Programming Instead

STOP: Which element of the table should we fill in next and what should its value be?

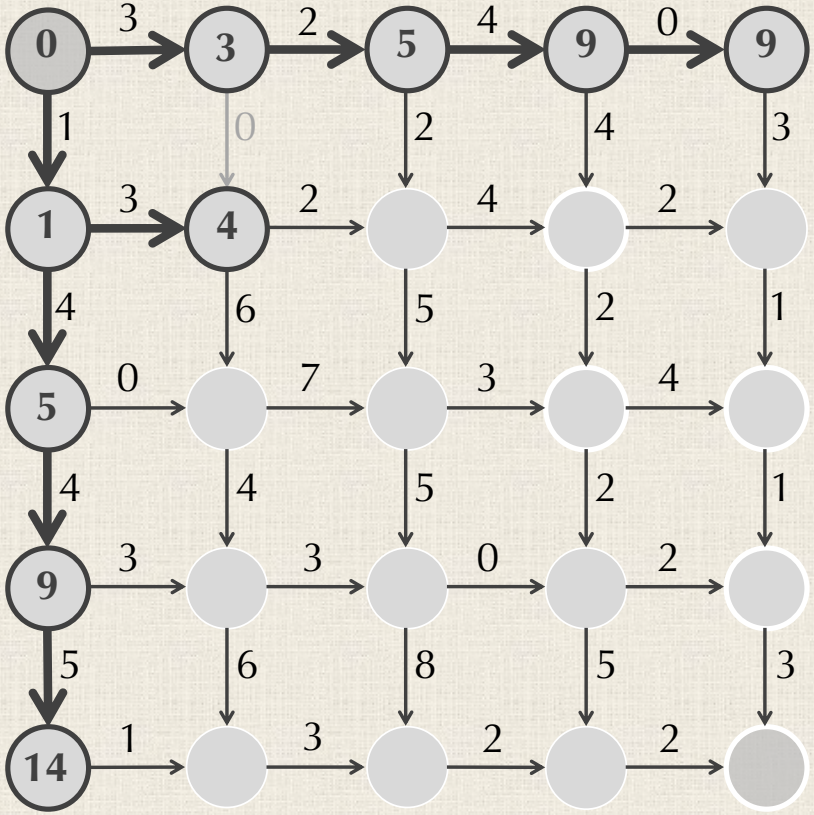


Recurrence relation

$$length(i, j) = \max\{length(i\text{-}1, j) + down(i, j),\ length(i, j\text{-}1) + right(i, j)\}$$

Let's Use Dynamic Programming Instead

Answer: We only know the values of MaxWeight for the two nodes adjacent to the node (1, 1); it gets the value max(3+0, 1+3) = 4.

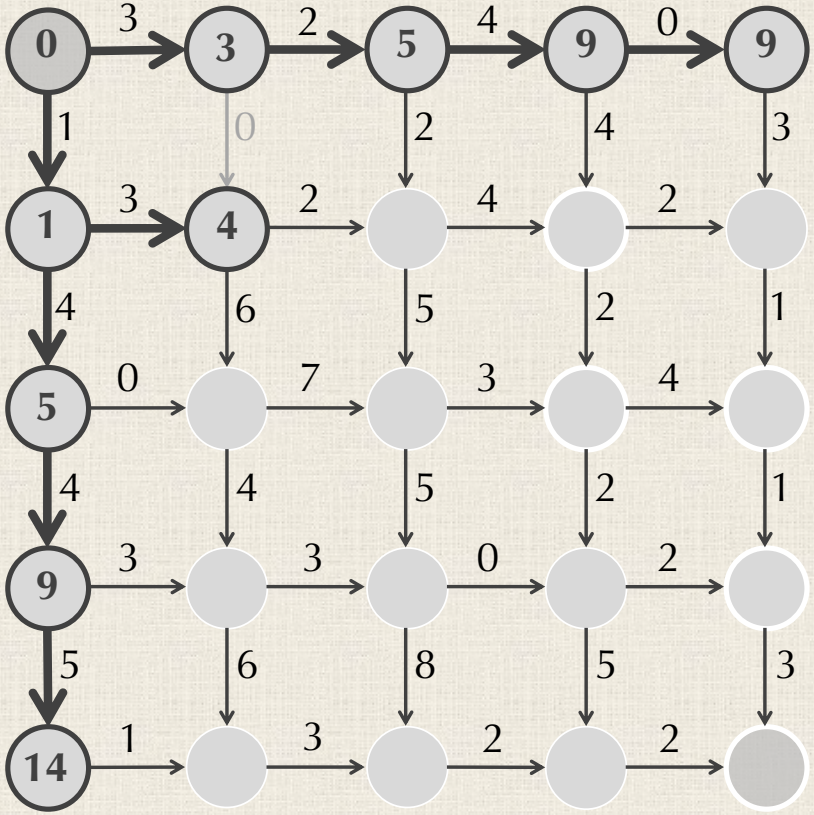


Recurrence relation

$$length(i, j) = \max\{length(i\text{-}1, j) + down(i, j),\ length(i, j\text{-}1) + right(i, j)\}$$

Let's Use Dynamic Programming Instead

STOP: Which elements should we fill in next and what should their values be?

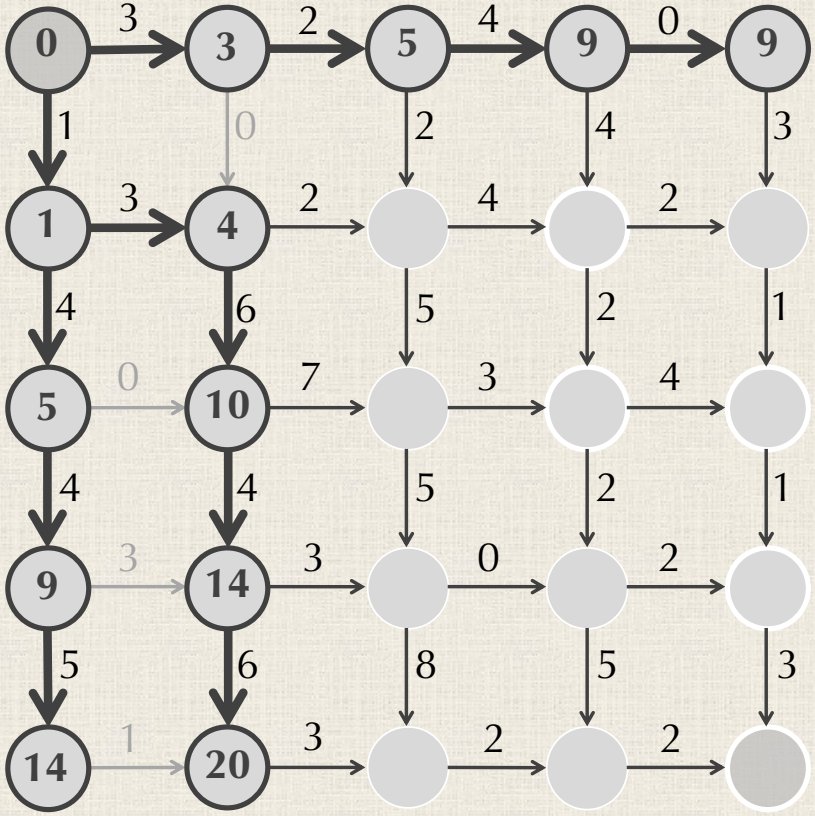


Recurrence relation

$$length(i, j) = \max\{length(i\text{-}1, j) + down(i, j),\ length(i, j\text{-}1) + right(i, j)\}$$

Let's Use Dynamic Programming Instead

Answer: We can fill in all of row 1 or all of column 1 (it doesn't matter which).

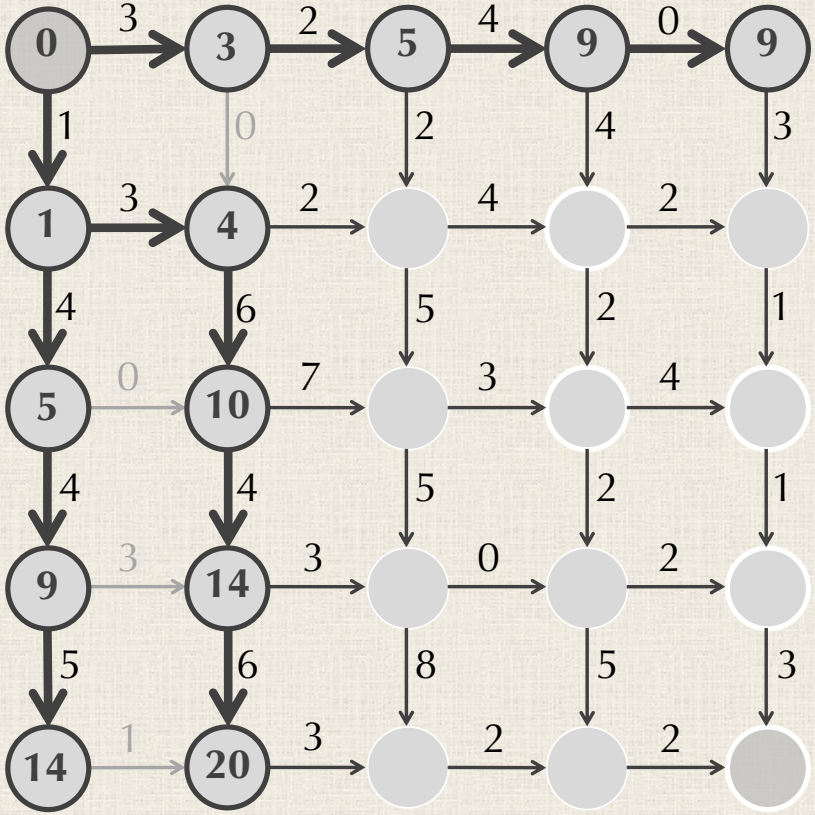


Recurrence relation

$$\text{MaxWeight}(i, j) = \max\{\text{MaxWeight}(i\text{-}1, j) + \textit{down}(i, j), \text{MaxWeight}(i, j\text{-}1) + \textit{right}(i, j)\}$$

Let's Use Dynamic Programming Instead

Exercise: Fill in the remaining values of *length* for this network.

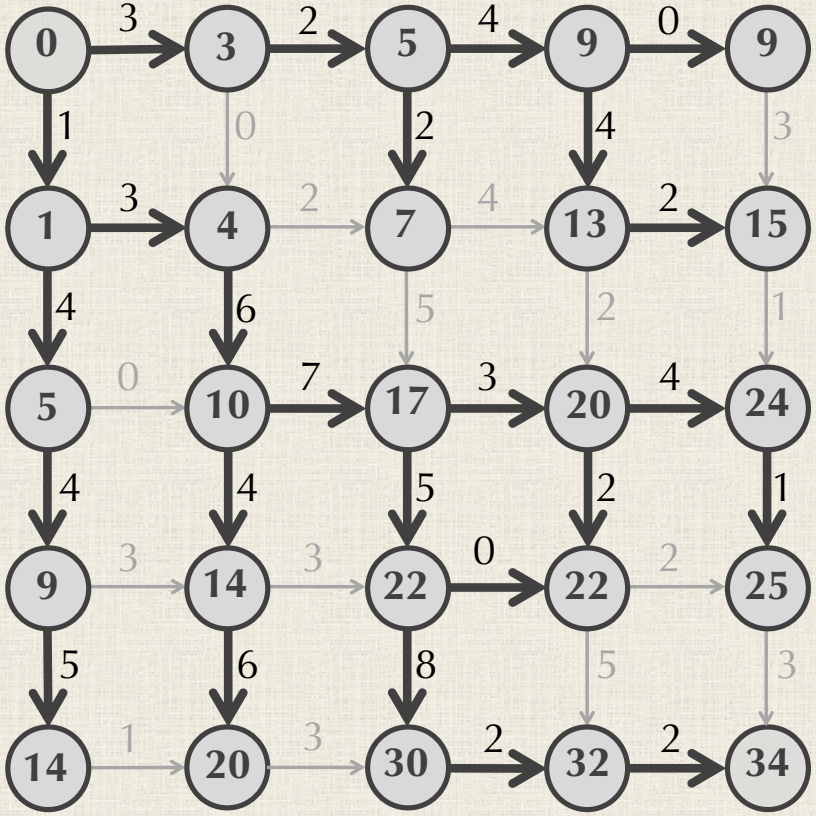


Recurrence relation

$$length(i, j) = \max\{length(i\text{-}1, j) + down(i, j),\ length(i, j\text{-}1) + right(i, j)\}$$

Let's Use Dynamic Programming Instead

STOP: Now do you see a longest path in this grid? How might we find one in general?



Recurrence relation

$$length(i, j) = \max\{length(i\text{-}1, j) + down(i, j),\ length(i, j\text{-}1) + right(i, j)\}$$

RETURNING TO SEQUENCE ALIGNMENT

Remember that These Two Problems are the Same

Longest Common Subsequence Length Problem:

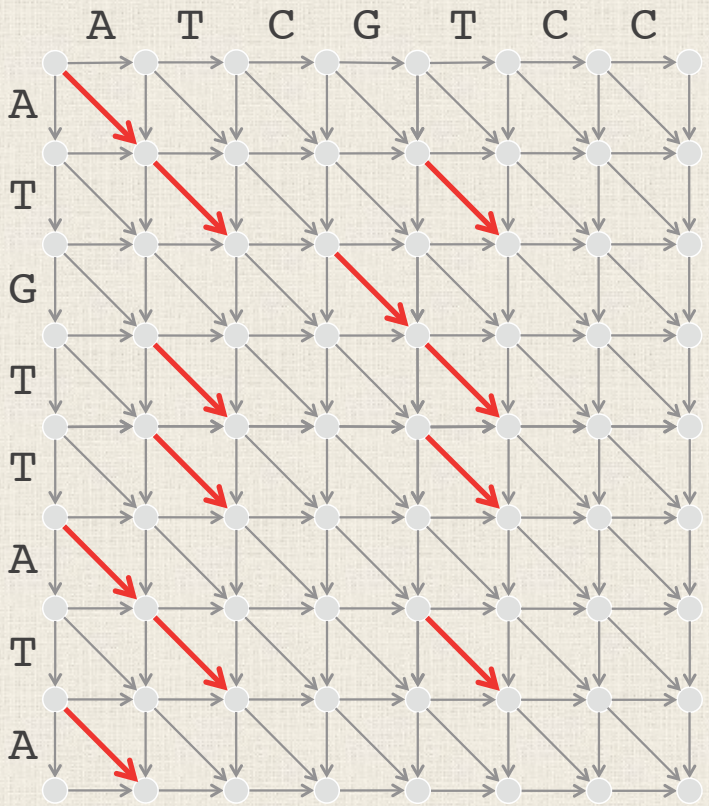
- **Input:** Two strings.
- **Output:** The length of a longest common subsequence of these strings.

Symbol Matching Problem:

- **Input:** Two strings.
- **Output:** The greatest number of matched symbols in any "alignment" of the two strings.

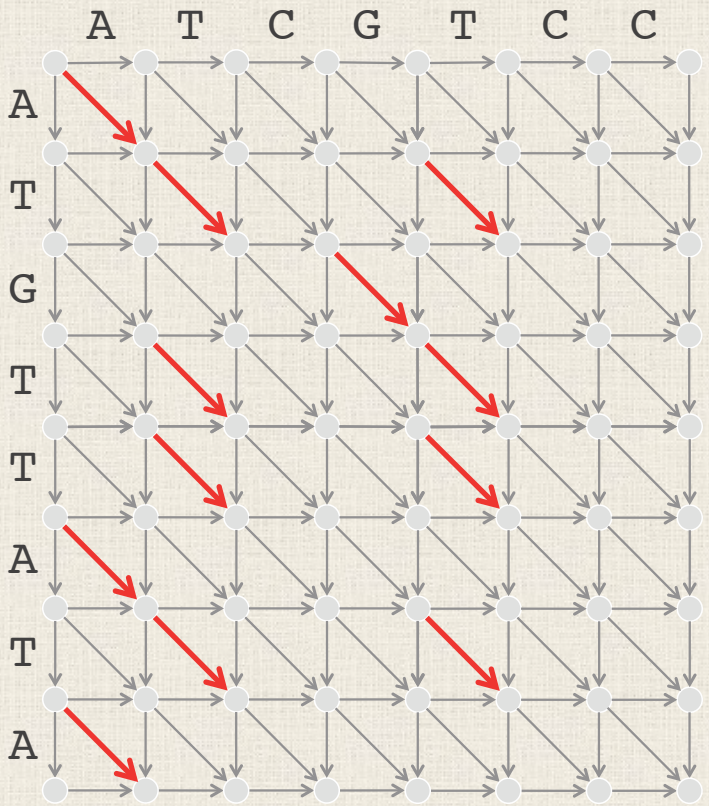
Connecting These Problems to the Alignment Graph

If we weight the red edges as 1 and the other edges as 0, then a maximum-weight path in the alignment graph from source to sink solves the Symbol Matching Problem.



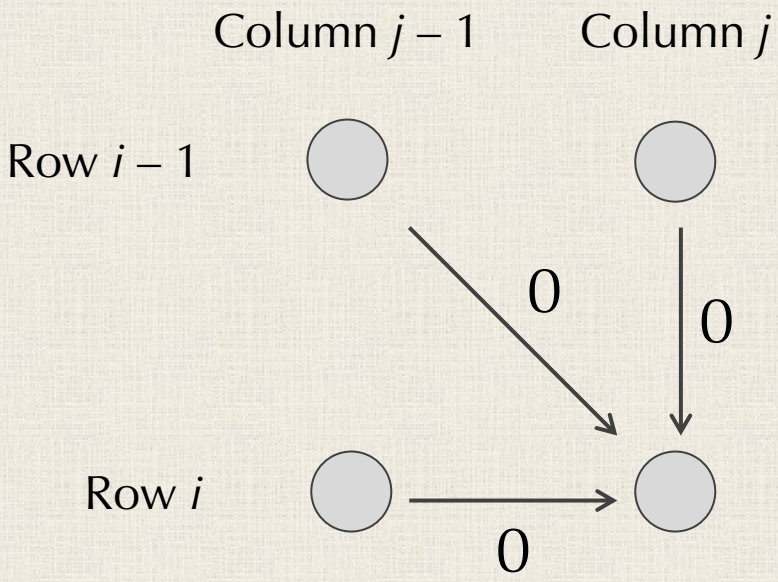
Finding an LCS

Exercise: What is the recurrence relation for finding the length of a longest common subsequence? (That is, maximizing the number of matched symbols?)



Our Recurrence Has Two Cases

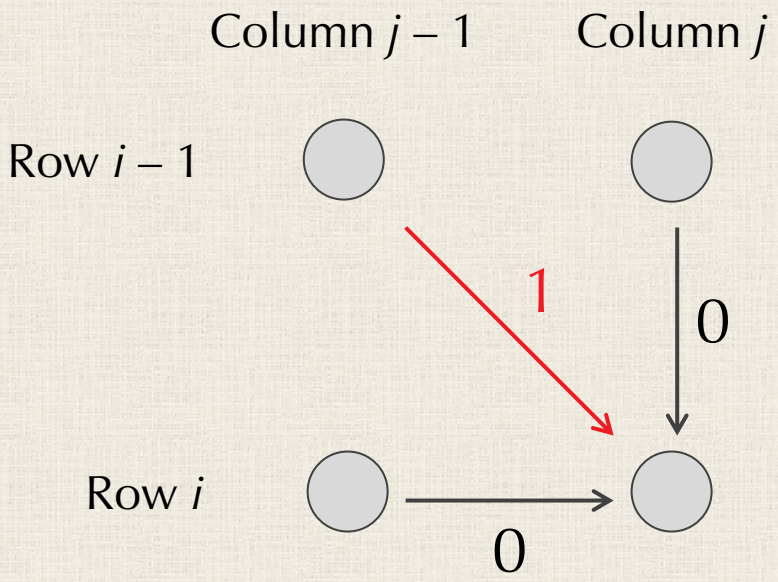
Case 1



$length(i, j)$ = maximum of:

- $length(i-1, j) + 0$
- $length(i, j-1) + 0$
- $length(i-1, j-1) + 0$

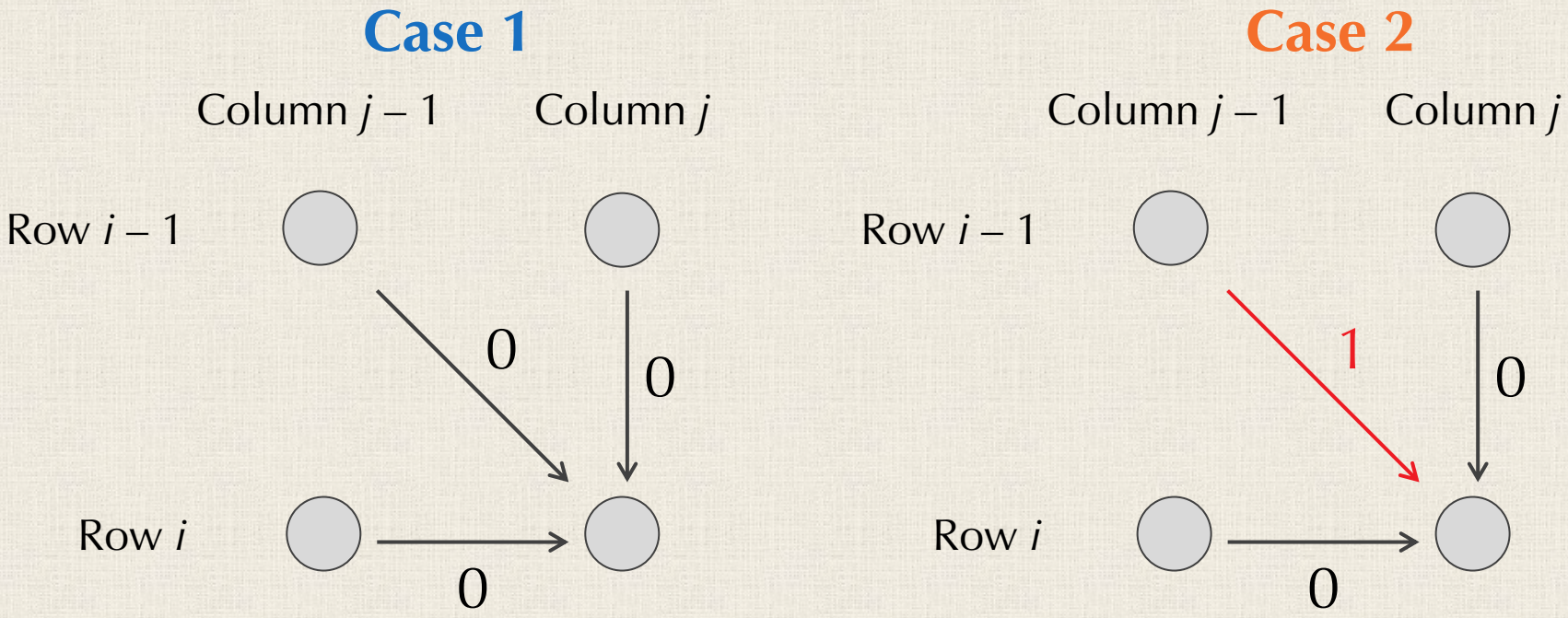
Case 2



$length(i, j)$ = maximum of:

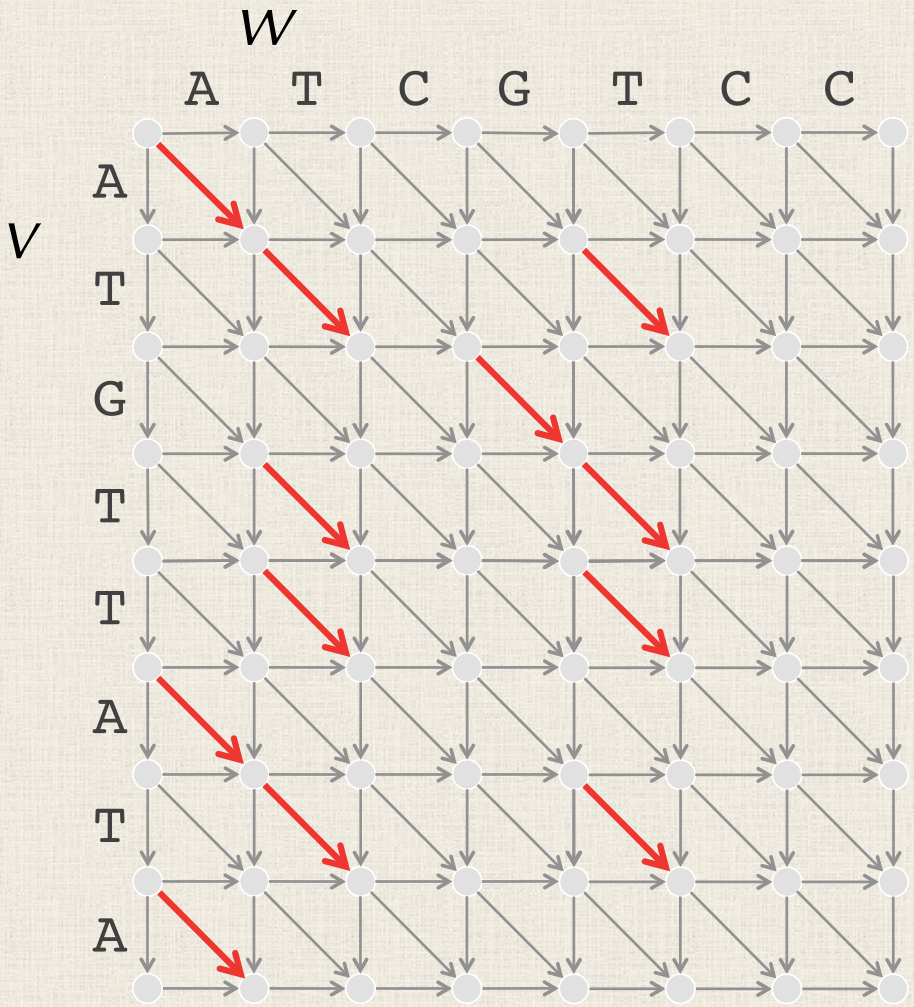
- $length(i-1, j) + 0$
- $length(i, j-1) + 0$
- $length(i-1, j-1) + 1$

Our Recurrence Has Two Cases



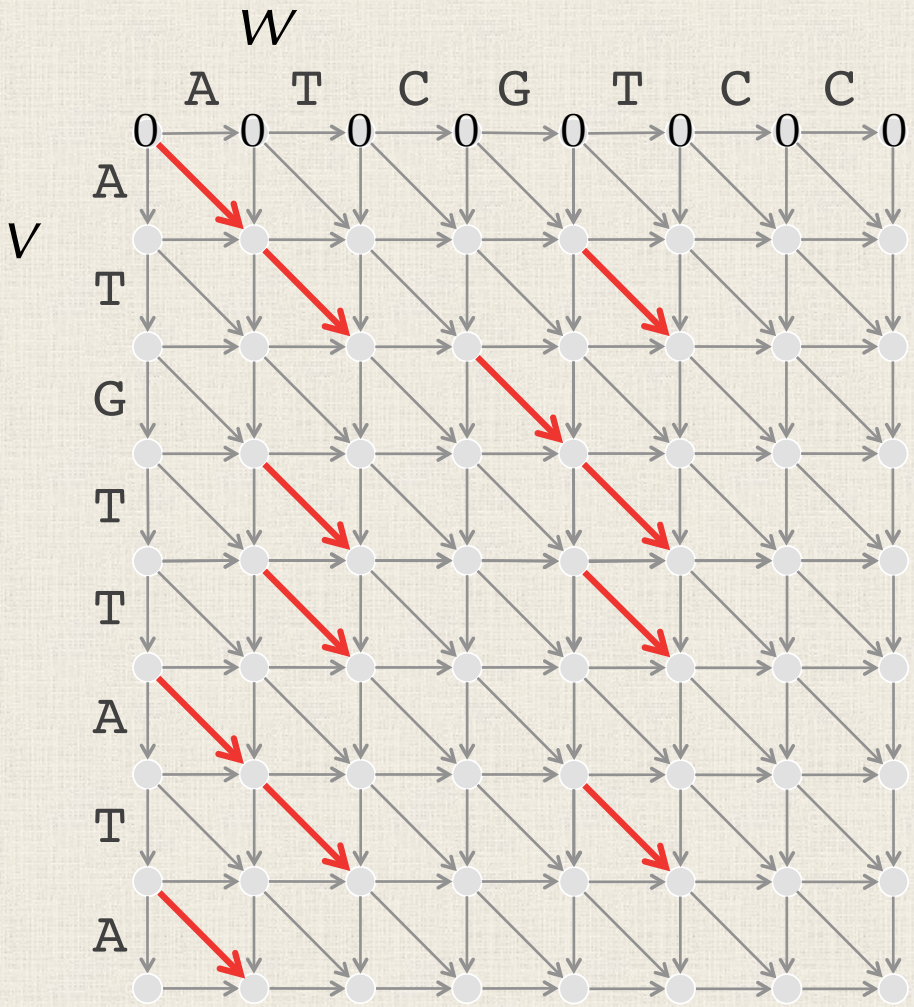
STOP: when will the diagonal edge weight be equal to 1?

Counting Matches Only



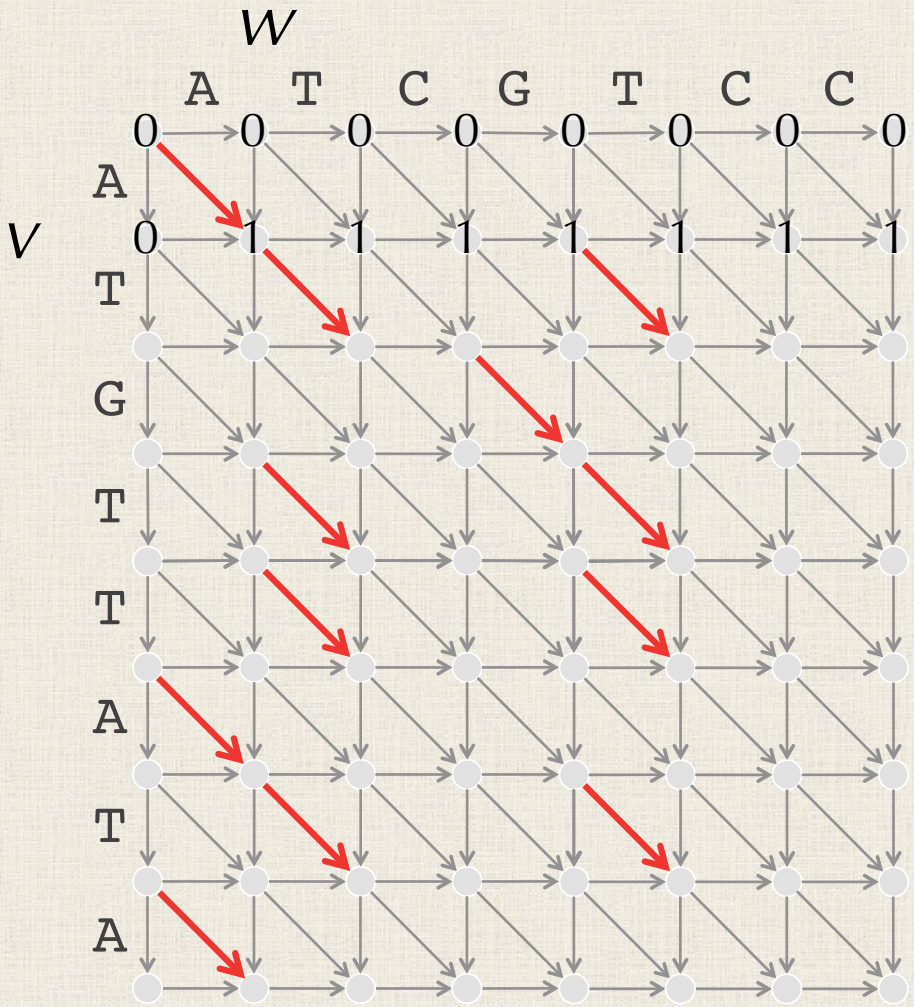
Answer: a diagonal edge connecting $(i - 1, j - 1)$ to (i, j) is 1 when the corresponding symbols $v[i - 1]$ and $w[j - 1]$ of the two strings *match*.

Filling in the LCS Dynamic Programming Table



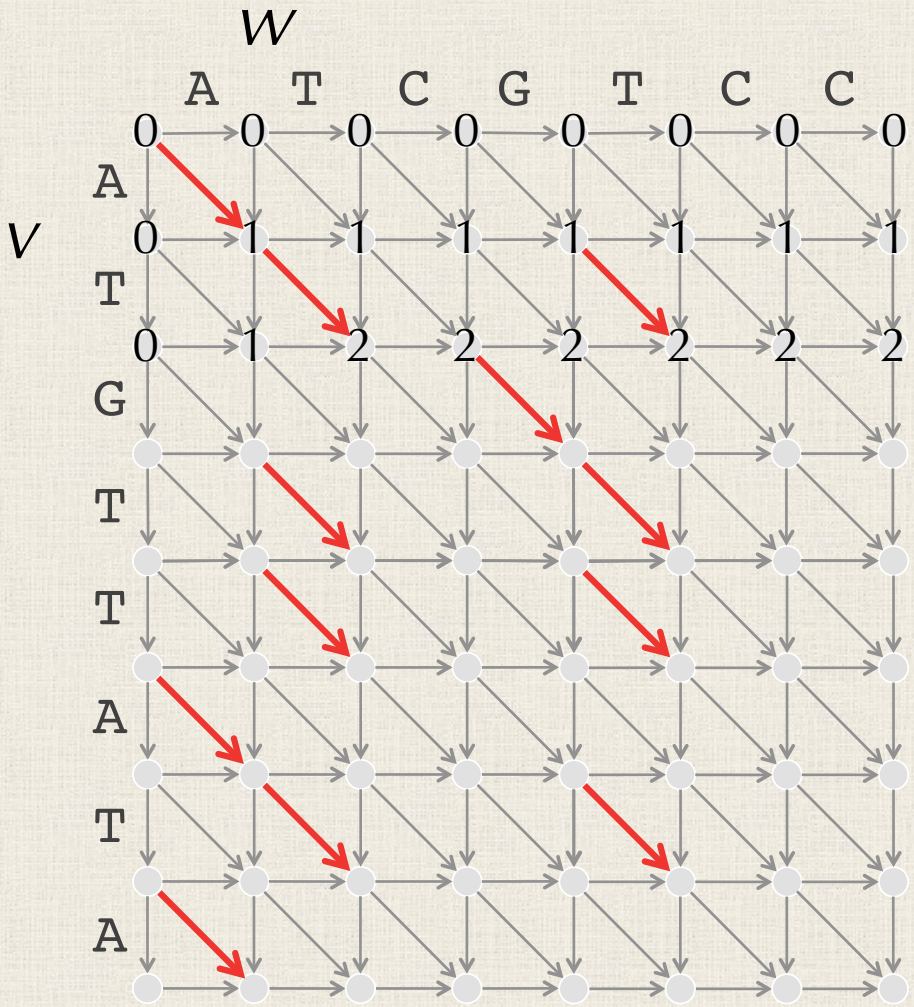
At left, we superimpose the dynamic programming table for finding the LCS of v = ATGTTATA and w = ATCGTCC.

Filling in the LCS Dynamic Programming Table



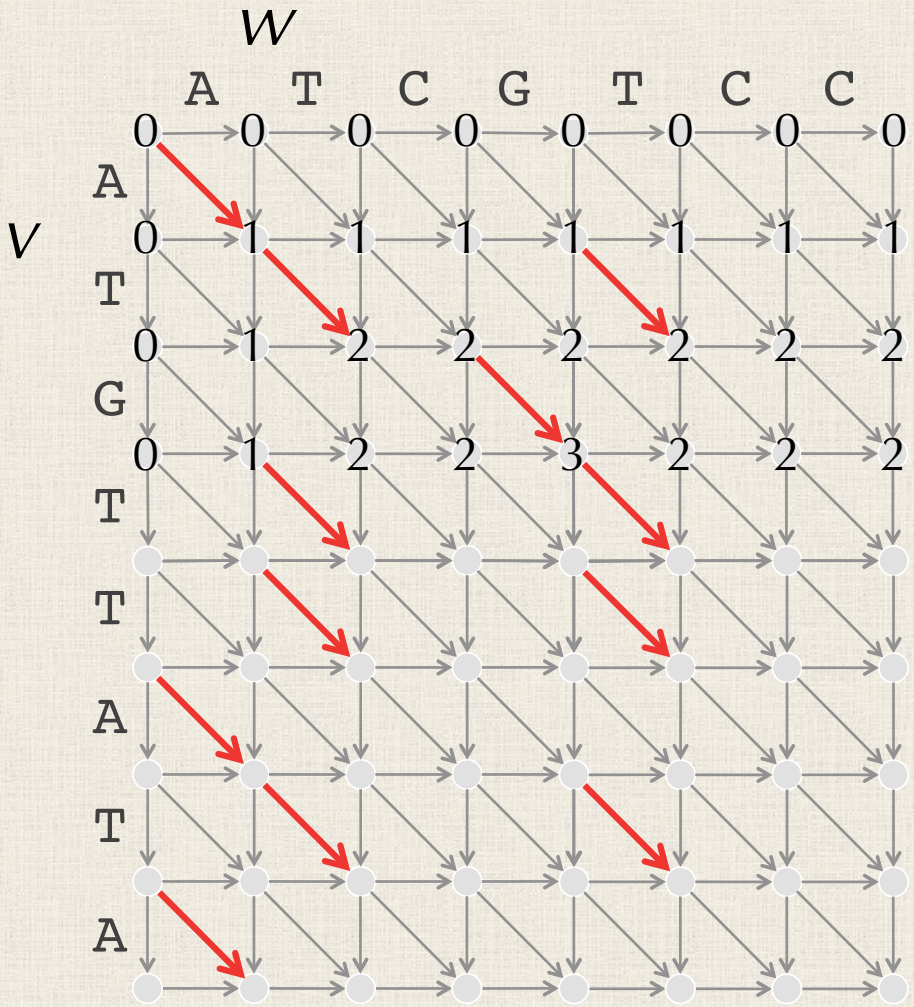
At left, we superimpose the dynamic programming table for finding the LCS of v = ATGTTATA and w = ATCGTCC.

Filling in the LCS Dynamic Programming Table



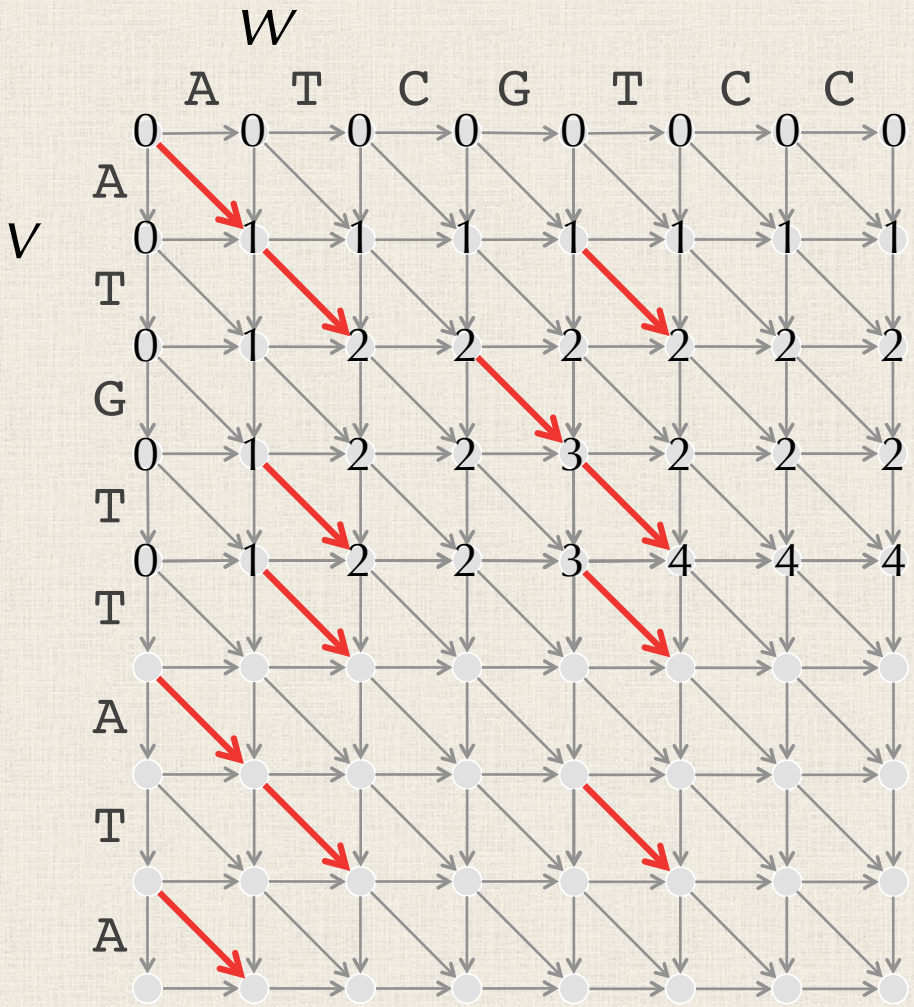
At left, we superimpose the dynamic programming table for finding the LCS of v = ATGTTATA and w = ATCGTCC.

Filling in the LCS Dynamic Programming Table



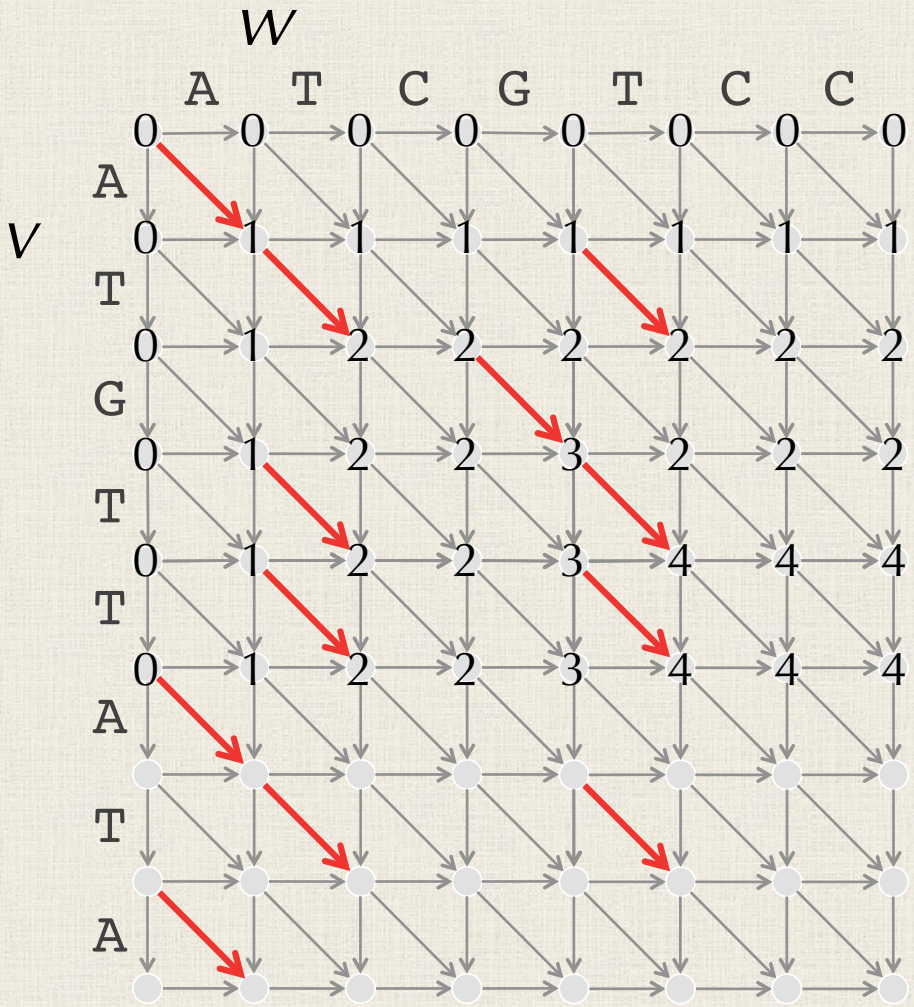
At left, we superimpose the dynamic programming table for finding the LCS of v = ATGTTATA and w = ATCGTCC.

Filling in the LCS Dynamic Programming Table



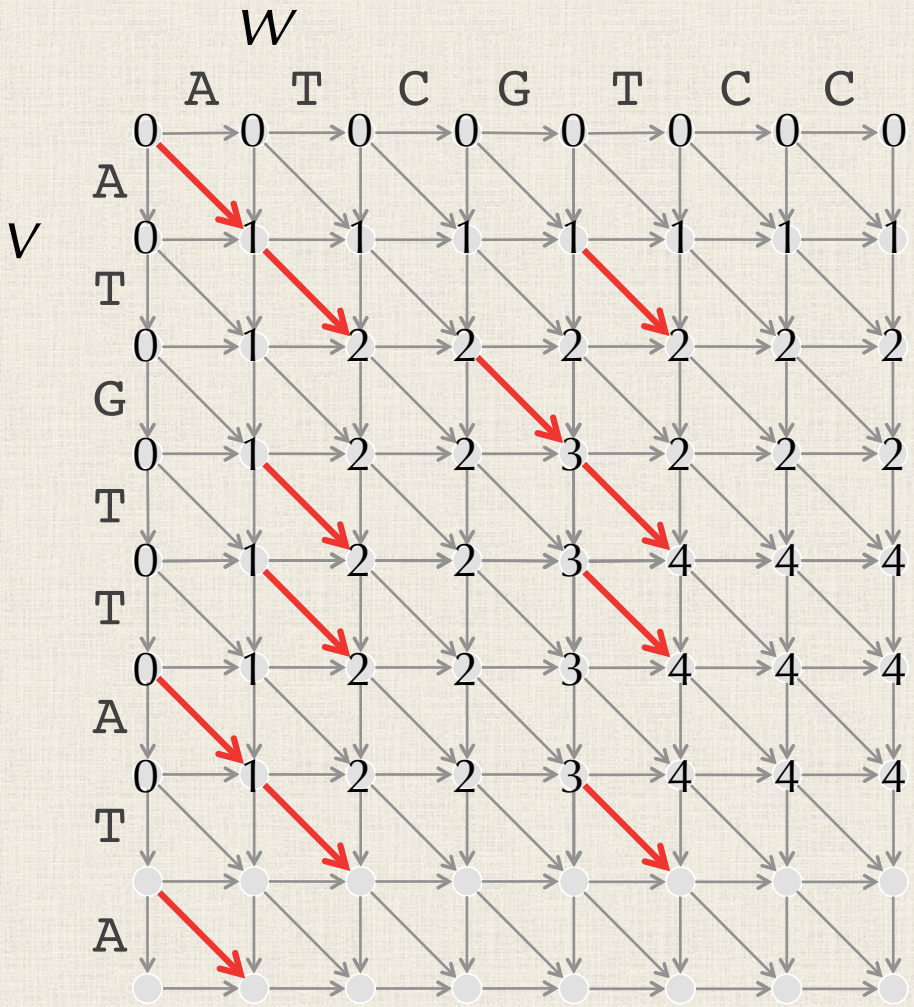
At left, we superimpose the dynamic programming table for finding the LCS of v = `ATGTTATA` and w = `ATCGTCC`.

Filling in the LCS Dynamic Programming Table



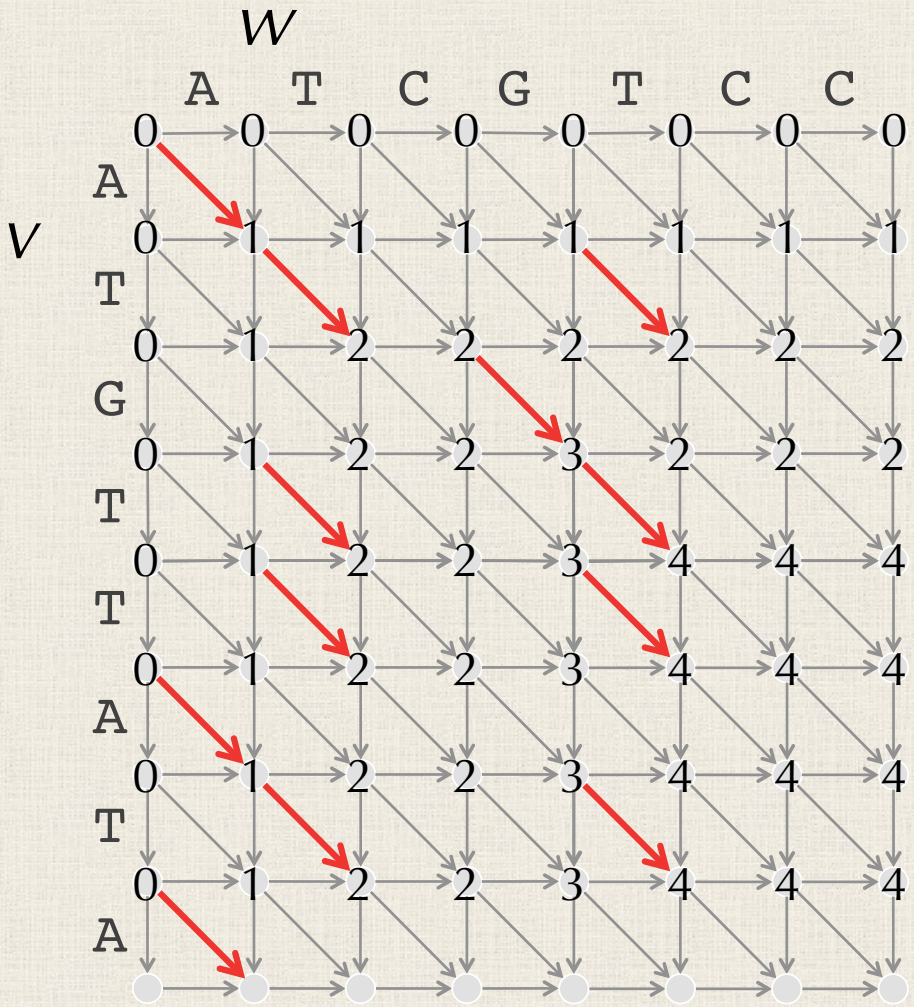
At left, we superimpose the dynamic programming table for finding the LCS of v = ATGTTATA and w = ATCGTCC.

Filling in the LCS Dynamic Programming Table



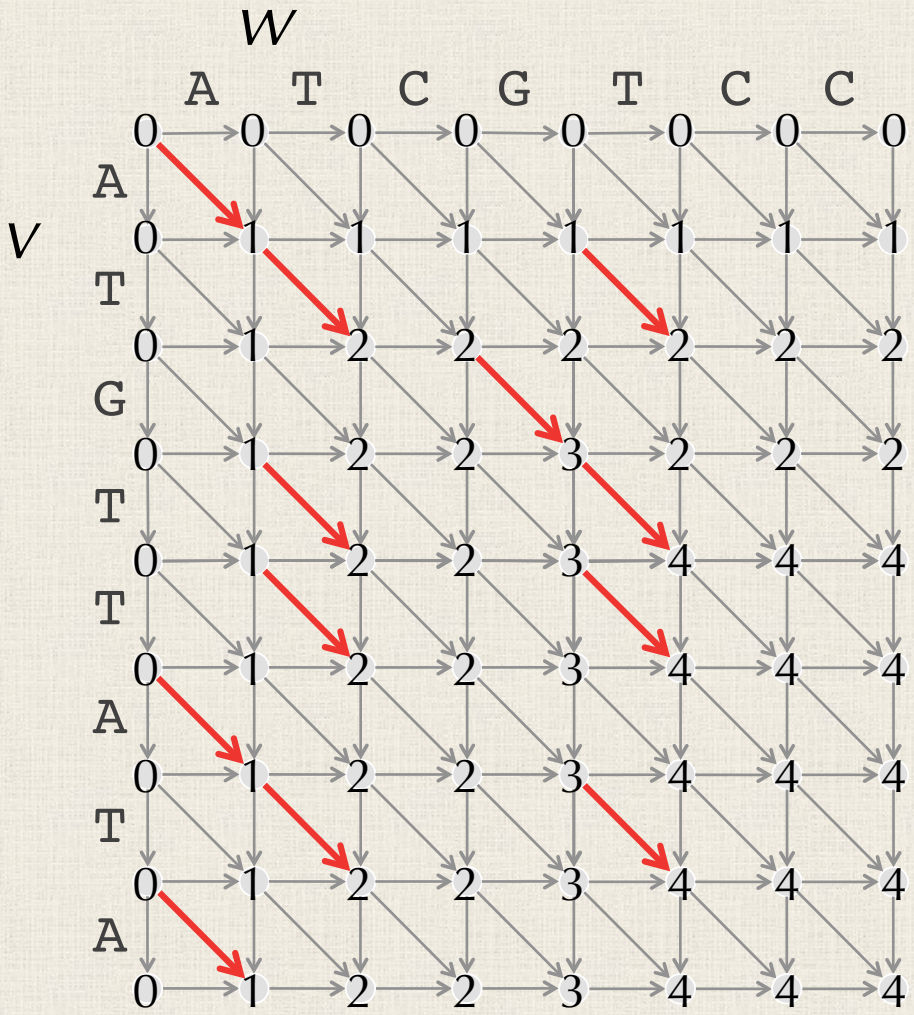
At left, we superimpose the dynamic programming table for finding the LCS of v = ATGTTATA and w = ATCGTCC.

Filling in the LCS Dynamic Programming Table



At left, we superimpose the dynamic programming table for finding the LCS of v = `ATGTTATA` and w = `ATCGTCC`.

Filling in the LCS Dynamic Programming Table



At left, we superimpose the dynamic programming table for finding the LCS of v = ATGTTATA and w = ATCGTCC.

Solving the Symbol Matching Problem

Symbol Matching Problem:

- **Input:** Two strings.
- **Output:** The greatest number of matched symbols in any "alignment" of the two strings.

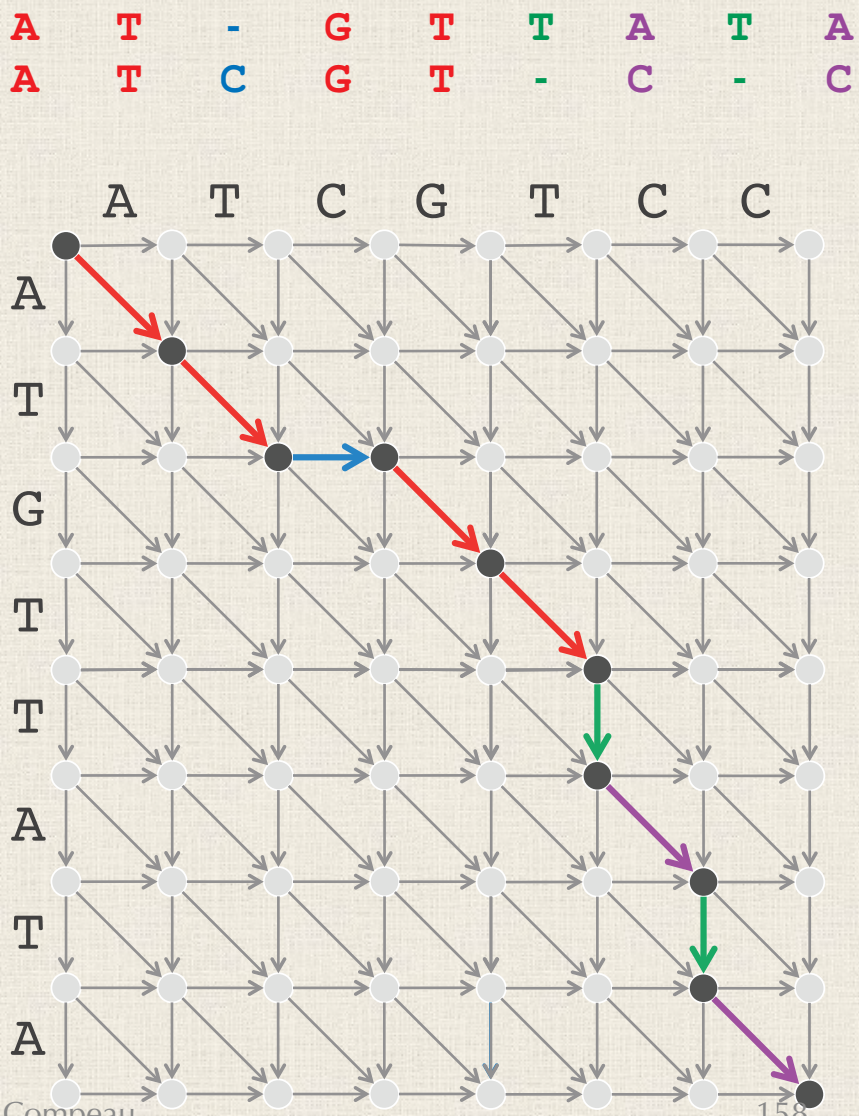
Code Challenge: write a function solving the Symbol Matching Problem (LCS Length).

Note: remember that the maximum number of matched symbols in an alignment equals the length of a longest common subsequence of the strings.

Toward a Distance Between Strings

The **edit distance** between two strings is the min # of mismatches, insertions, and deletions, needed to change one string into the other.

The transformation at right gives 5 total operations, but this may not be the minimum number …



Edit Distance

Edit Distance Problem:

- **Input:** Two strings.
- **Output:** The minimum number of combined insertions, deletions, and mismatches in any "alignment" of the two strings.

Code Challenge: write a function solving the Edit Distance Problem.

Hint: you should first find a recurrence relation for edit distance using the alignment network.

From Pairs to Many Strings

Pairwise Edit Distance Problem:

- **Input:** A collection of n strings *patterns*.
- **Output:** An n x n matrix D such that $D(i, j)$ is the edit distance between *patterns*$[i]$ and *patterns*$[j]$.

Code Challenge: write a function solving the Pairwise Edit Distance Problem.

Dealing with Both Strands

STOP: How could we change our code to take into account that the two strings we are comparing come from *opposing strands* of DNA?

Dealing with Both Strands

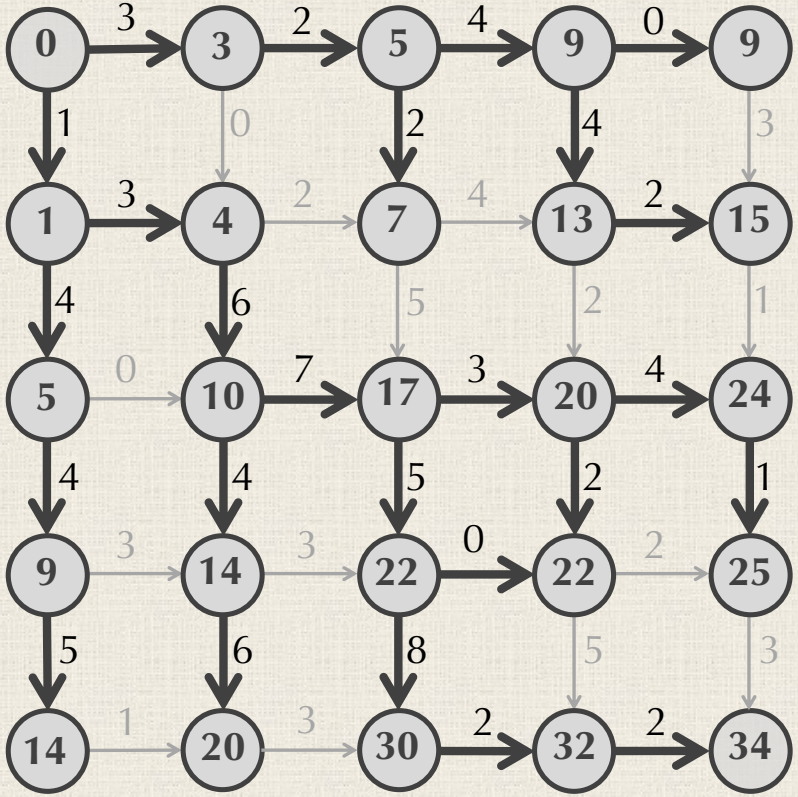
STOP: How could we change our code to take into account that the two strings we are comparing come from *opposing strands* of DNA?

Answer: For a given pair of strings *s*1 and *s*2, take the minimum of:

- the edit distance between *s*1 and *s*2
- the edit distance between *s*1 and the *reverse complement* of *s*2

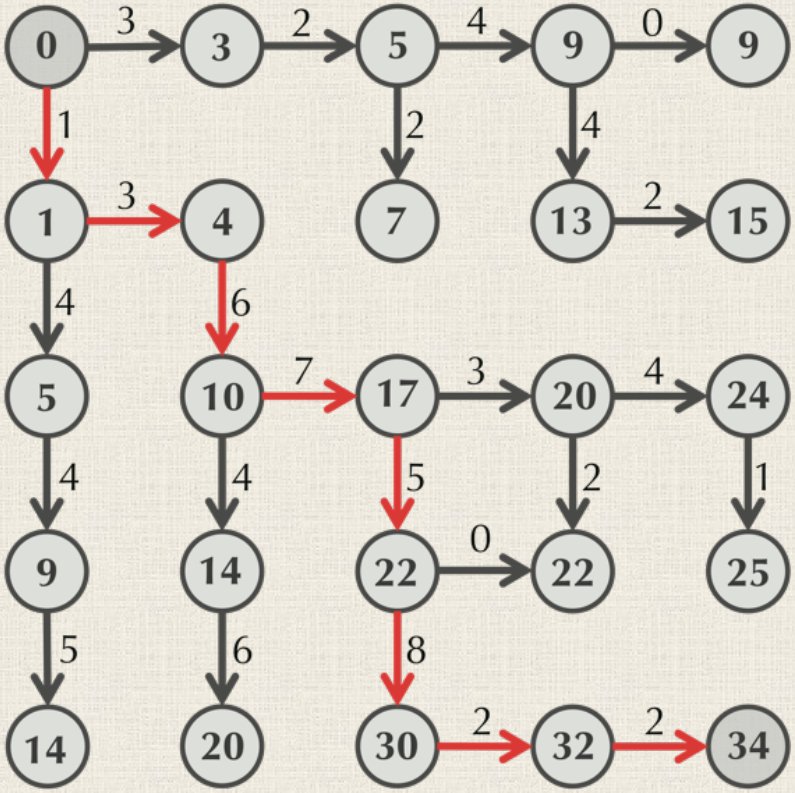
RECONSTRUCTING A LONGEST PATH IN A NETWORK

Reconstructing an Optimal Path



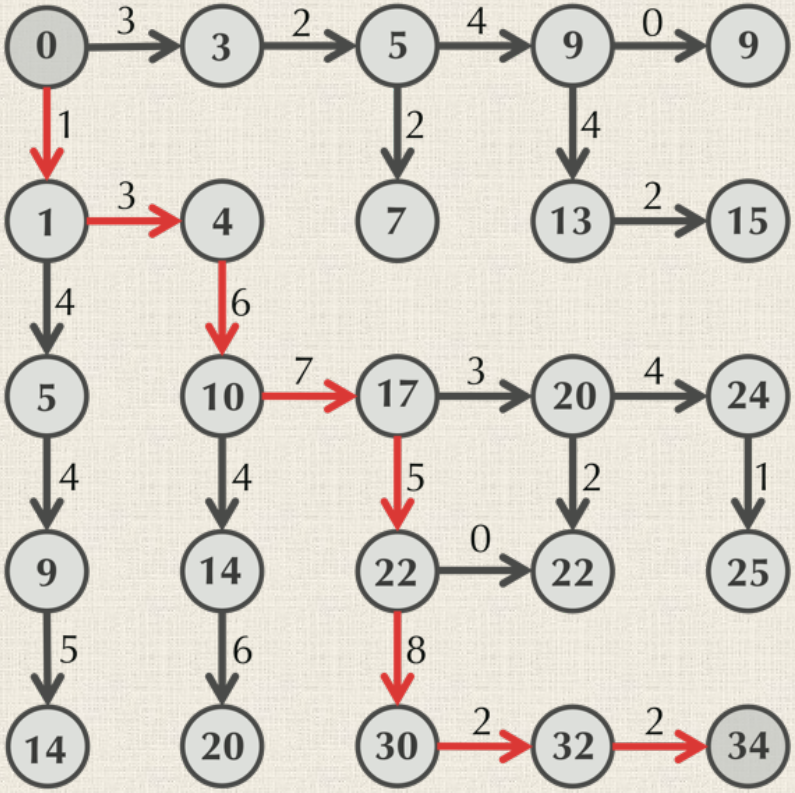
Where is the longest path hiding in this network?

Reconstructing an Optimal Path



Answer: start at *ending* node and follow edges *backwards* to the beginning node.

Reconstructing an Optimal Path



Note: to construct a longest path, we need to store *backtracking pointers* pointing to previous node.

Recall that these Problems are the Same

Longest Common Subsequence Length Problem:

- **Input:** Two strings.
- **Output:** The length of a longest common subsequence of these strings.

Symbol Matching Problem:

- **Input:** Two strings.
- **Output:** The greatest number of matched symbols in any "alignment" of the two strings.

Putting it All Together

Longest Common Subsequence ~~Length~~ Problem:.

- **Input:** Two strings.
- **Output:** The ~~length of a~~ longest common subsequence of these strings.

STOP: How can we find an LCS of two strings?

Putting it All Together

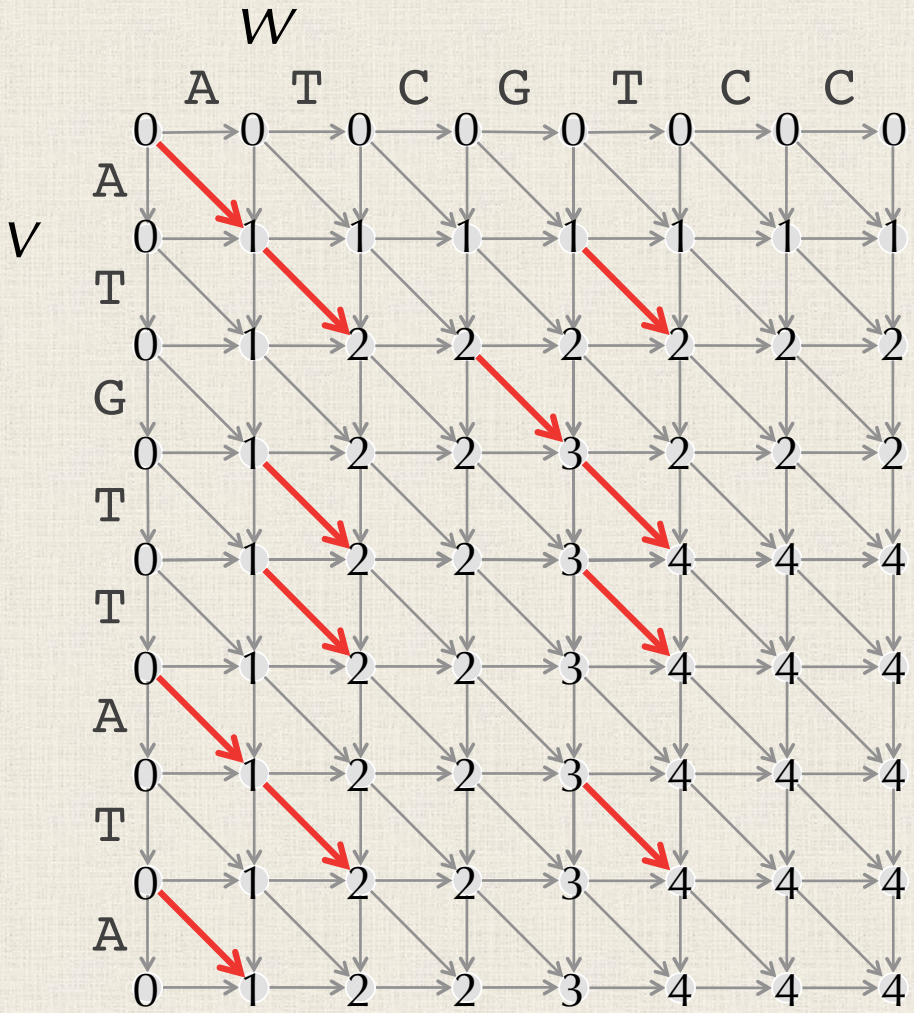
Longest Common Subsequence ~~Length~~ Problem:.

- **Input:** Two strings.
- **Output:** The ~~length of a~~ longest common subsequence of these strings.

Answer:

1. Build the alignment graph, with ″match″ edges weighted 1.
2. Find the length of an LCS using recurrence relation.
3. Backtrack to find longest path.

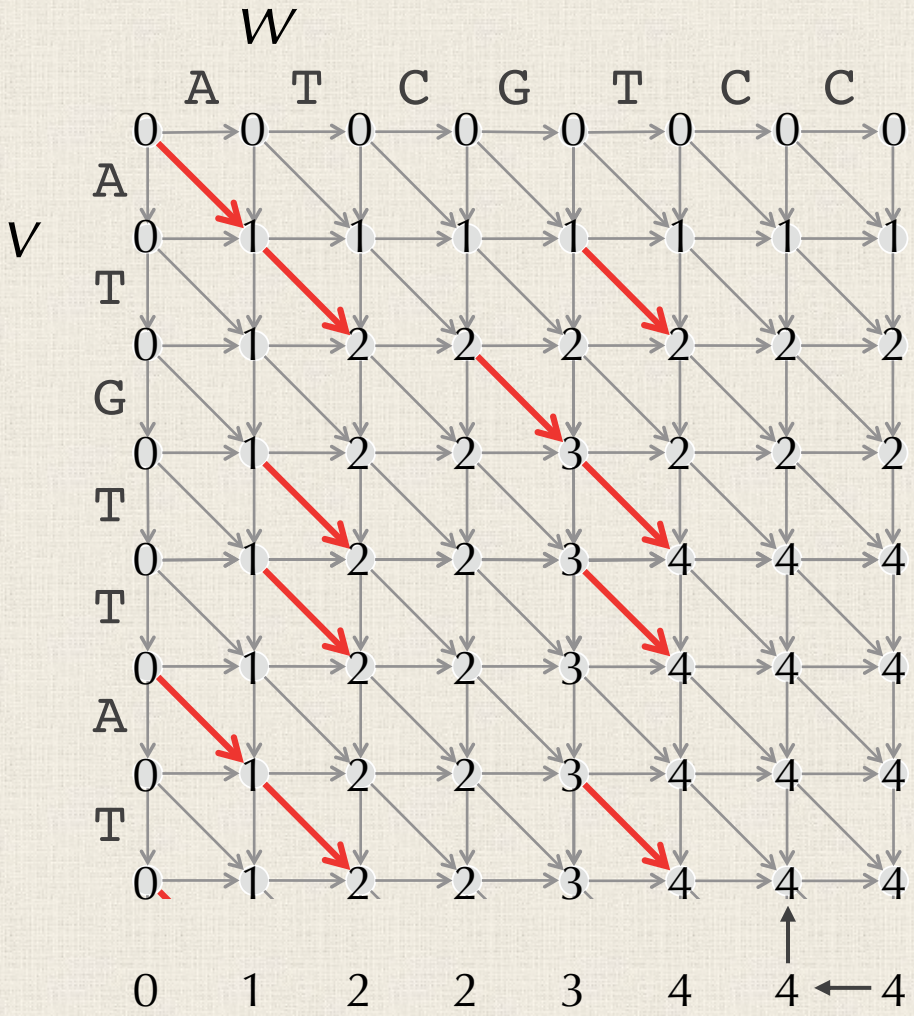
Filling in the LCS Dynamic Programming Table



We remind you of the dynamic programming table for finding the LCS of v = `ATGTTATA` and w = `ATCGTCC`.

STOP: Can you see how to backtrack?

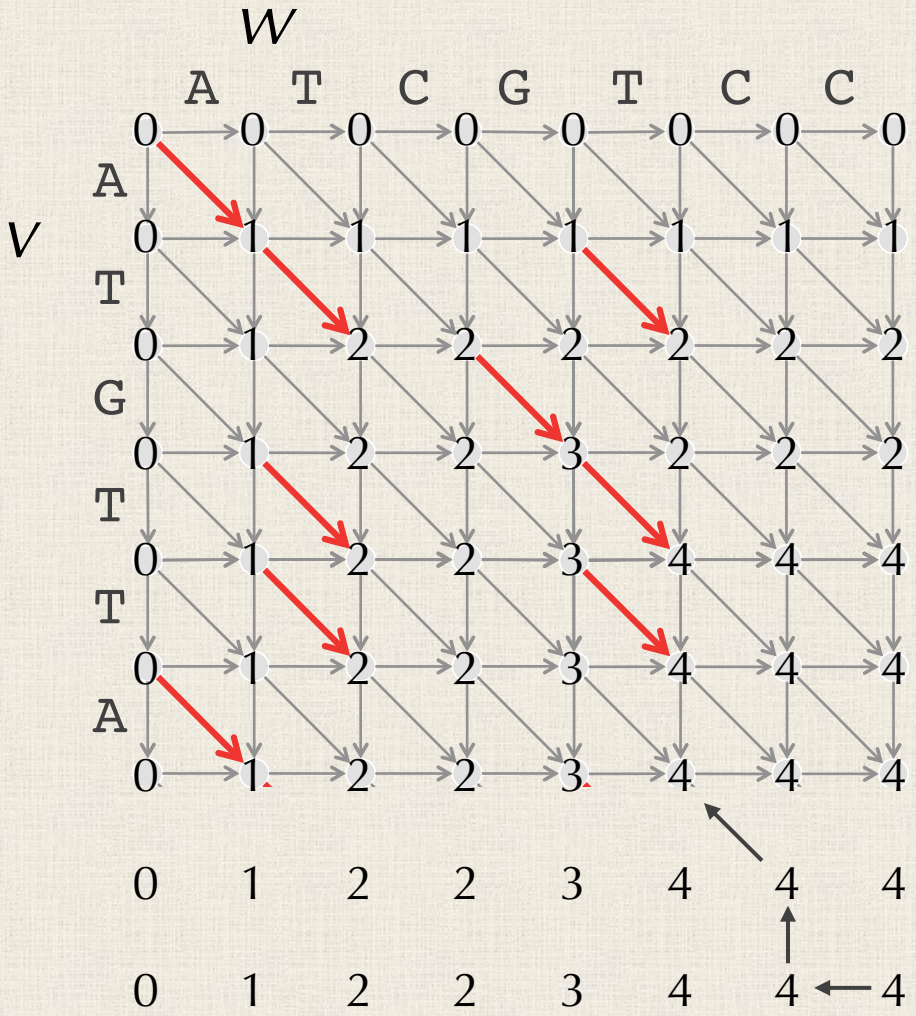
Filling in the LCS Dynamic Programming Table



We remind you of the dynamic programming table for finding the LCS of v = `ATGTTATA` and w = `ATCGTCC`.

Answer: There are lots of answers. Here is one.

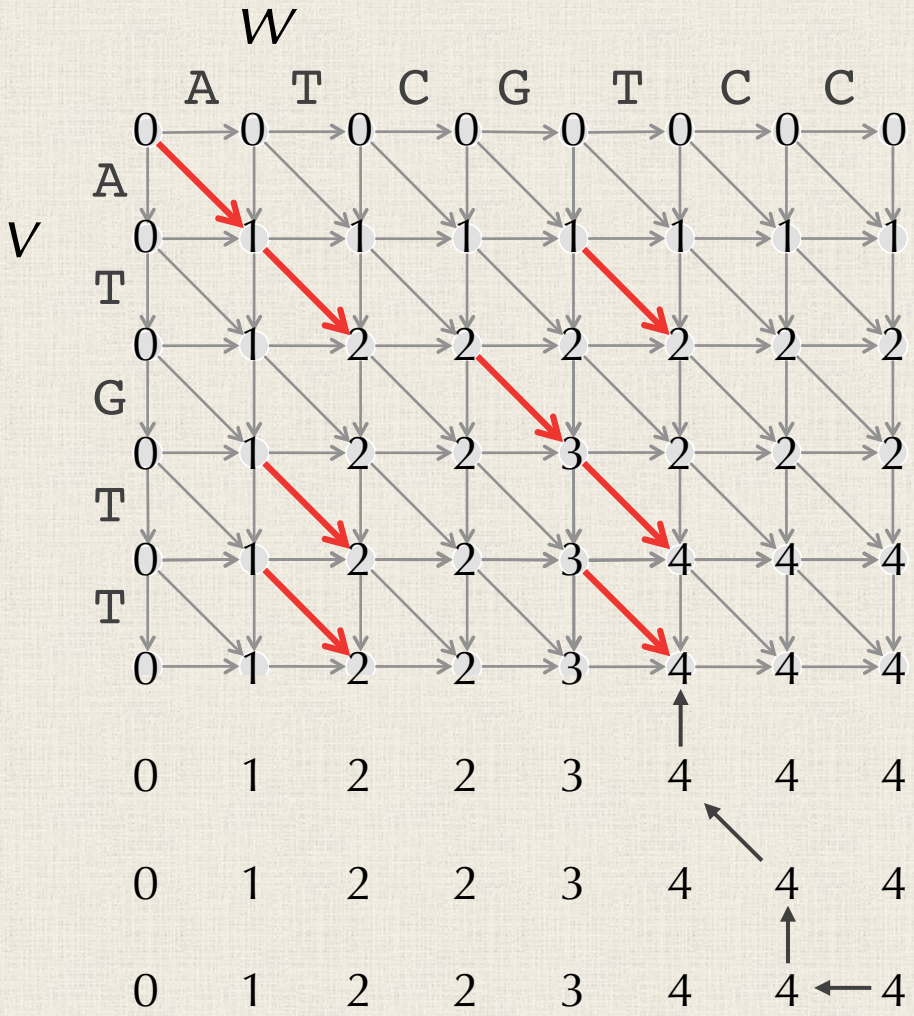
Filling in the LCS Dynamic Programming Table



We remind you of the dynamic programming table for finding the LCS of v = `ATGTTATA` and w = `ATCGTCC`.

Answer: There are lots of answers. Here is one.

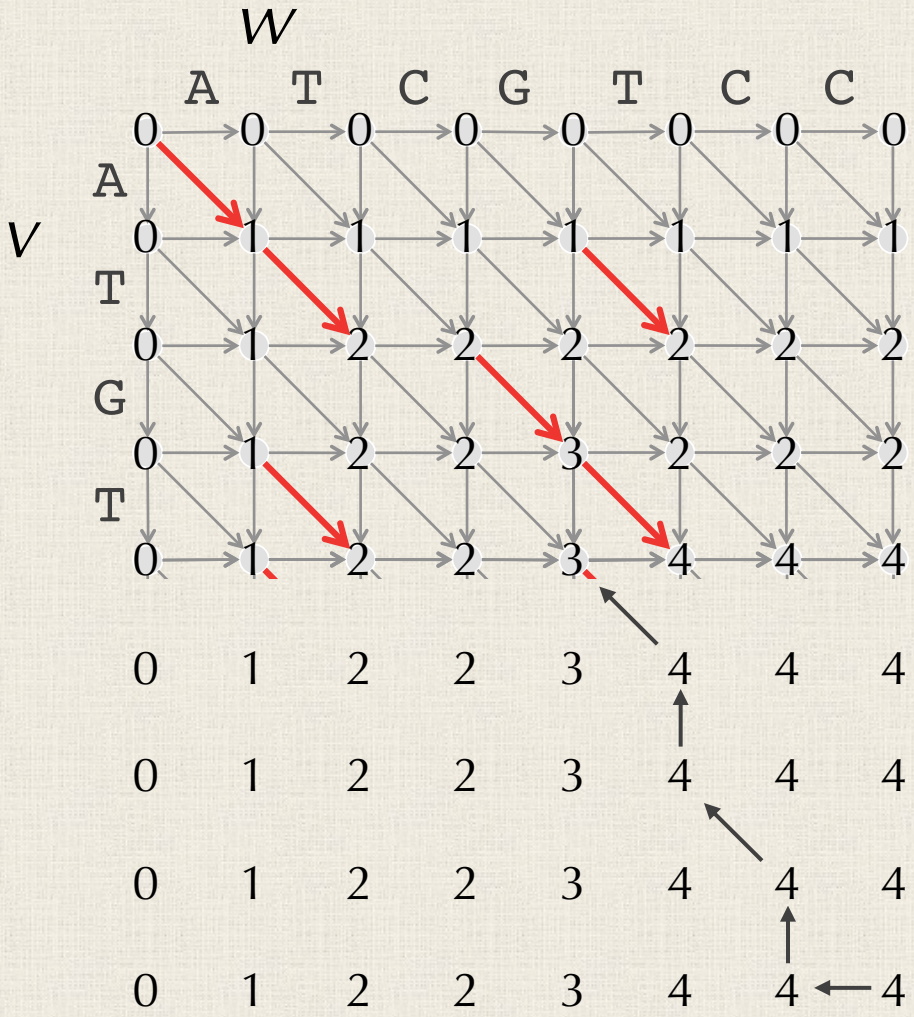
Filling in the LCS Dynamic Programming Table



We remind you of the dynamic programming table for finding the LCS of v = `ATGTTATA` and w = `ATCGTCC`.

Answer: There are lots of answers. Here is one.

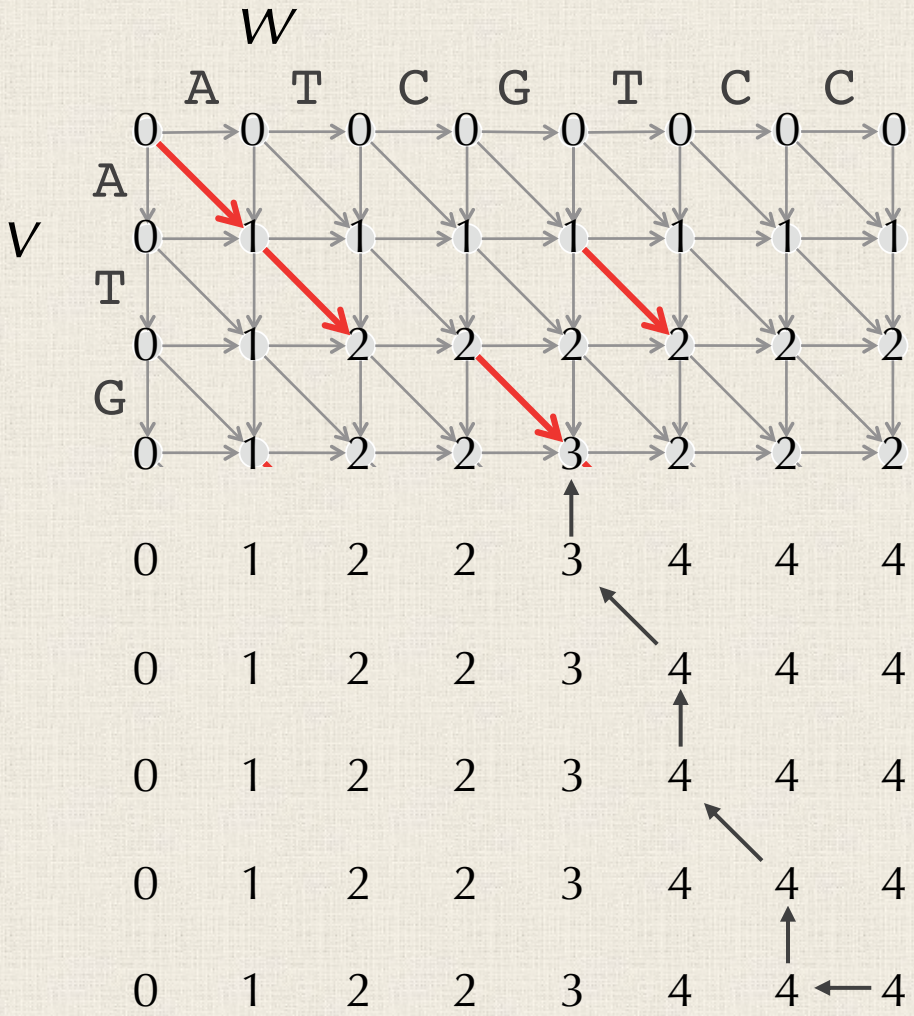
Filling in the LCS Dynamic Programming Table



We remind you of the dynamic programming table for finding the LCS of v = `ATGTTATA` and w = `ATCGTCC`.

Answer: There are lots of answers. Here is one.

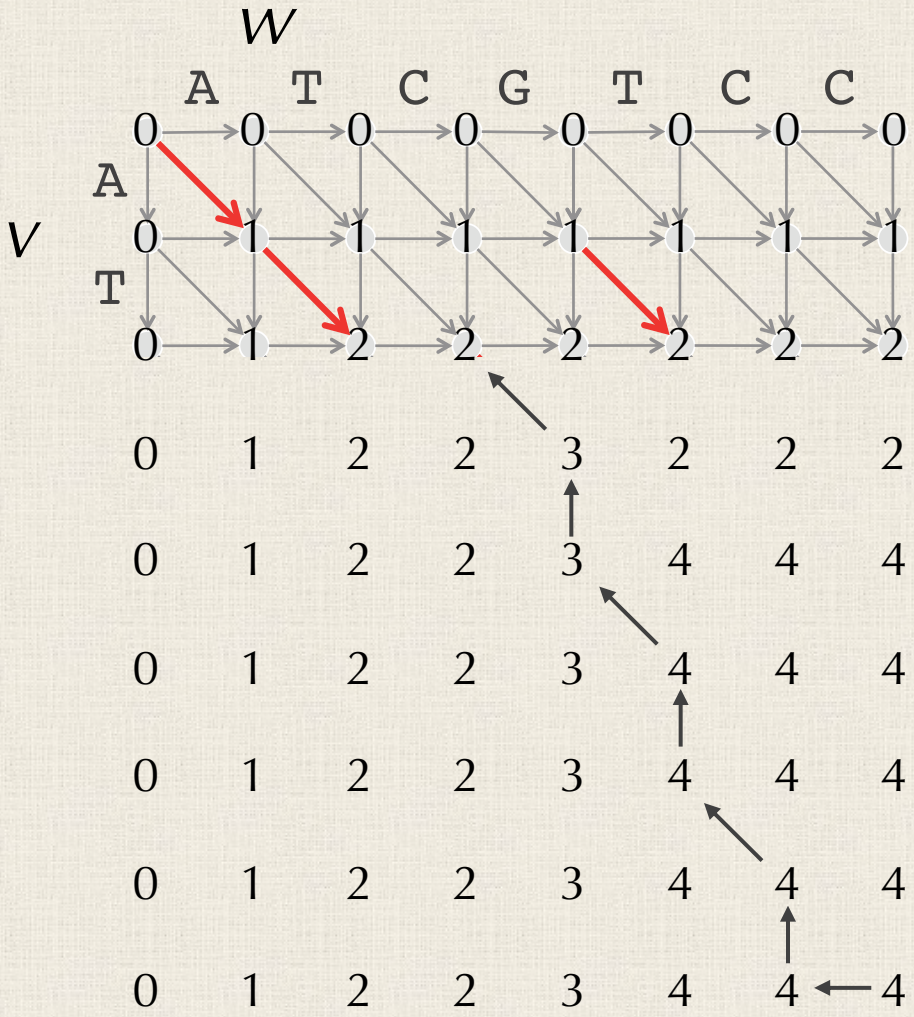
Filling in the LCS Dynamic Programming Table



We remind you of the dynamic programming table for finding the LCS of v = `ATGTTATA` and w = `ATCGTCC`.

Answer: There are lots of answers. Here is one.

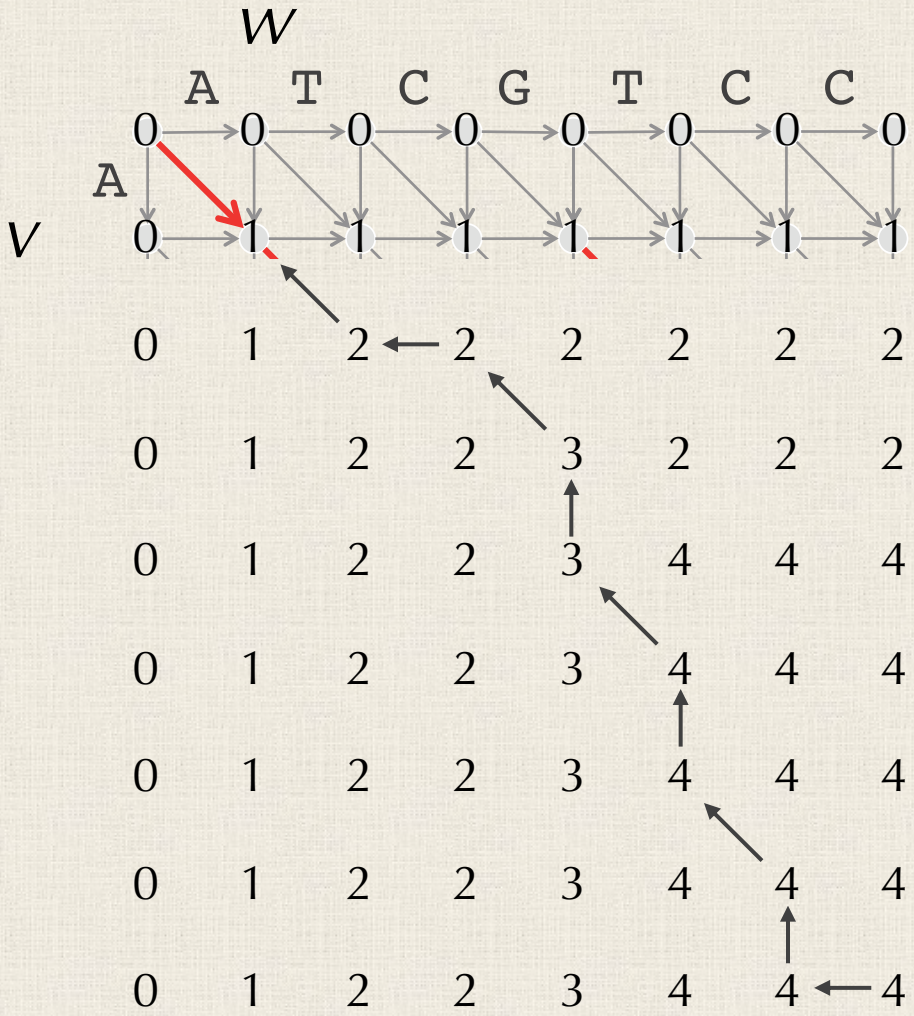
Filling in the LCS Dynamic Programming Table



We remind you of the dynamic programming table for finding the LCS of v = `ATGTTATA` and w = `ATCGTCC`.

Answer: There are lots of answers. Here is one.

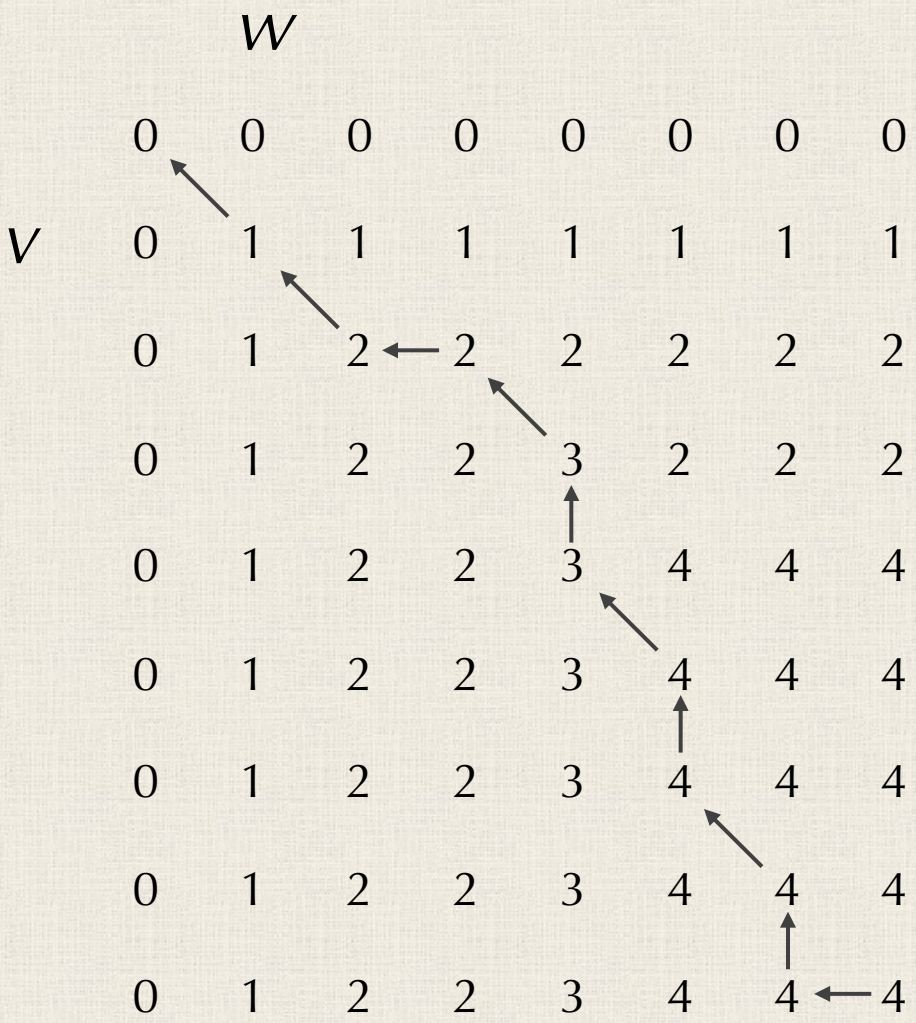
Filling in the LCS Dynamic Programming Table



We remind you of the dynamic programming table for finding the LCS of v = `ATGTTATA` and w = `ATCGTCC`.

Answer: There are lots of answers. Here is one.

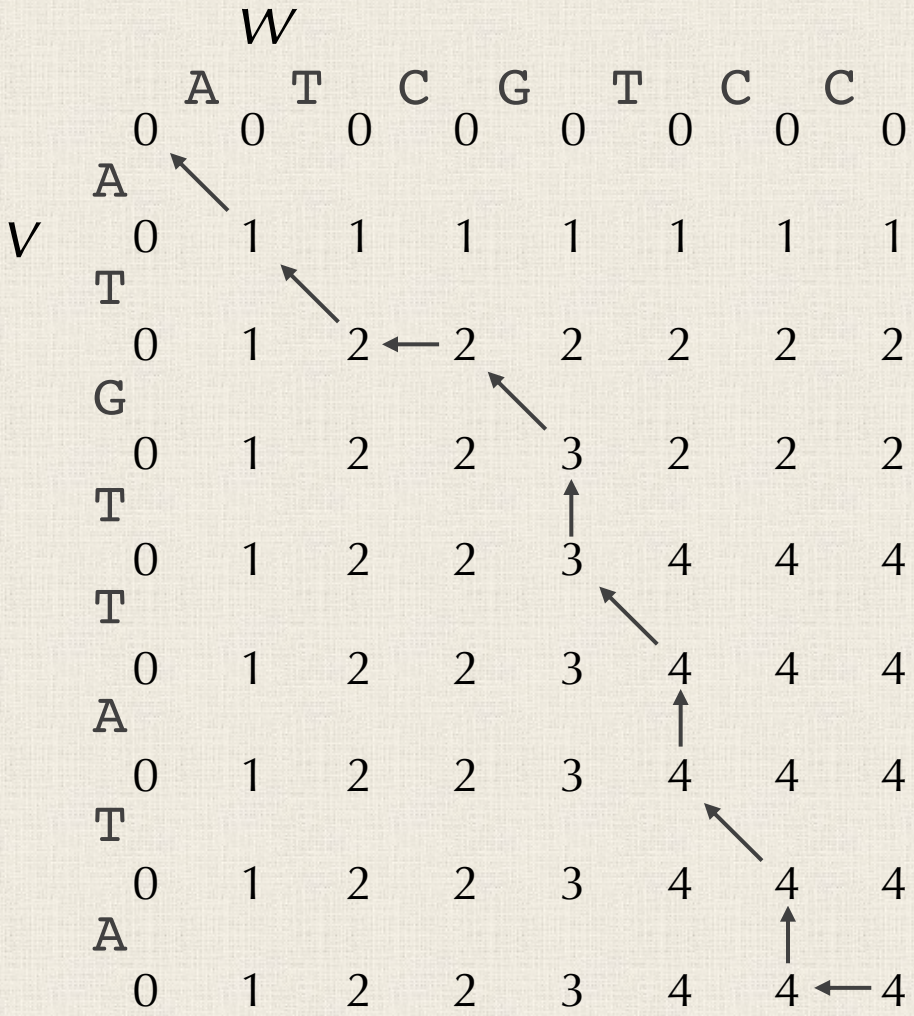
Filling in the LCS Dynamic Programming Table



We remind you of the dynamic programming table for finding the LCS of v = `ATGTTATA` and w = `ATCGTCC`.

Answer: There are lots of answers. Here is one.

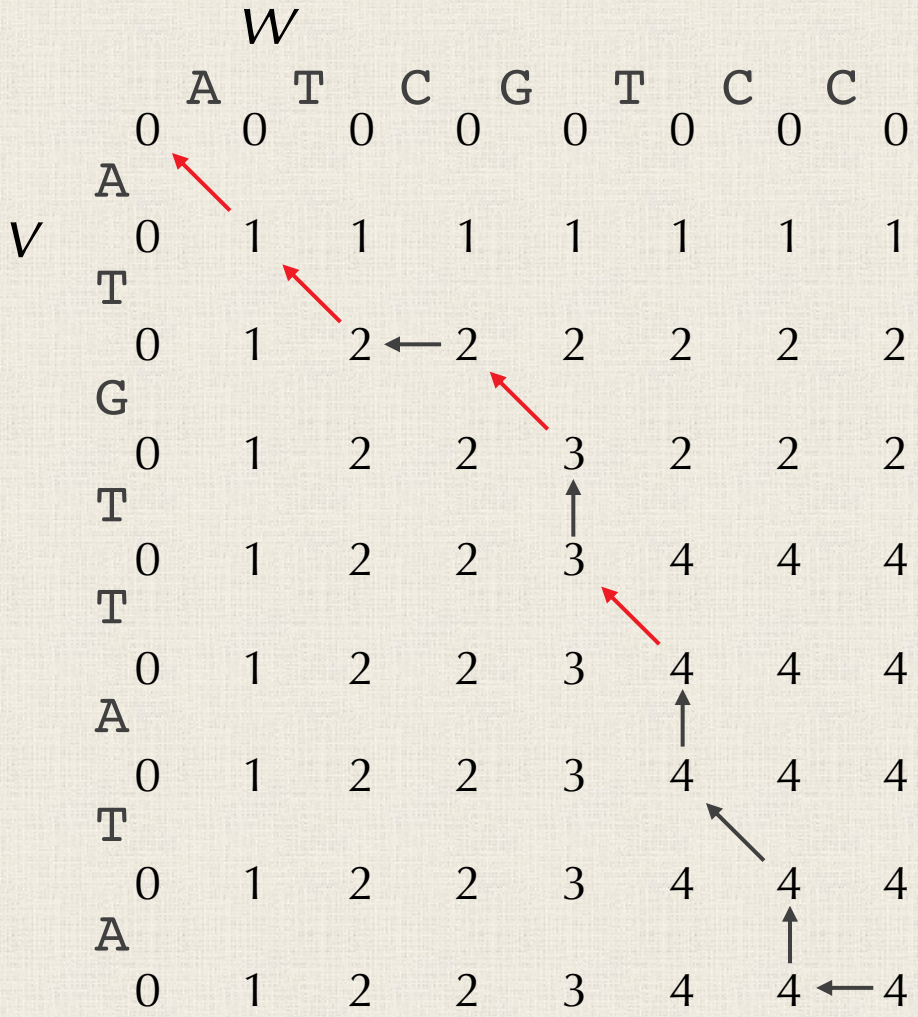
Filling in the LCS Dynamic Programming Table



We remind you of the dynamic programming table for finding the LCS of v = `ATGTTATA` and w = `ATCGTCC`.

STOP: What is the LCS corresponding to this path?

Filling in the LCS Dynamic Programming Table



We remind you of the dynamic programming table for finding the LCS of v = `ATGTTATA` and w = `ATCGTCC`.

Answer: We highlight diagonal edges that increase score, giving the LCS `ATGT`.

Finding an LCS

Say that we want to find not just the *length* of an LCS, but an LCS itself.

Longest Common Subsequence Problem:

- **Input:** Two strings.
- **Output:** A longest common subsequence of these strings.

Code Challenge: Try solving this problem. (It's tough! How can we store backtracking pointers?).

GLOBAL ALIGNMENT

Strengthening Alignment Scoring

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

Alignment score: Divided into three components:

- **match** reward (+1)
- **mismatch** penalty (-μ)
- **insertion**/**deletion** penalty (-σ)

Strengthening Alignment Scoring

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

Alignment score: Divided into three components:

- **match** reward (+1)
- **mismatch** penalty (-μ)
- **insertion**/**deletion** penalty (-σ)

STOP: What were μ and σ when finding a longest common subsequence?

Strengthening Alignment Scoring

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

Alignment score: Divided into three components:

- **match** reward (+1)
- **mismatch** penalty (-μ)
- **insertion**/**deletion** penalty (-σ)

Answer: They were both equal to zero…

Strengthening Alignment Scoring

Global Alignment Problem:

- **Input:** Two strings along with mismatch and insertion/deletion penalties μ and σ.
- **Output:** An alignment of the strings with maximum alignment score.

Strengthening Alignment Scoring

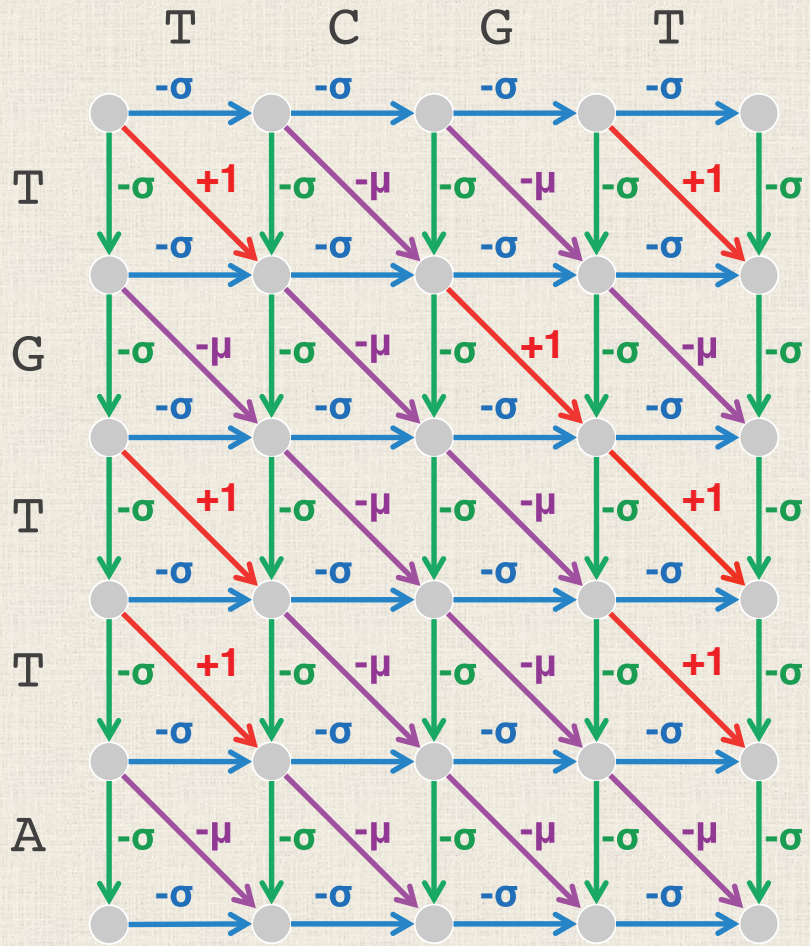
Global Alignment Problem:

- **Input:** Two strings along with mismatch and insertion/deletion penalties μ and σ.
- **Output:** An alignment of the strings with maximum alignment score.

STOP: How can we modify the alignment network to solve this problem?

Modifying the Alignment Network

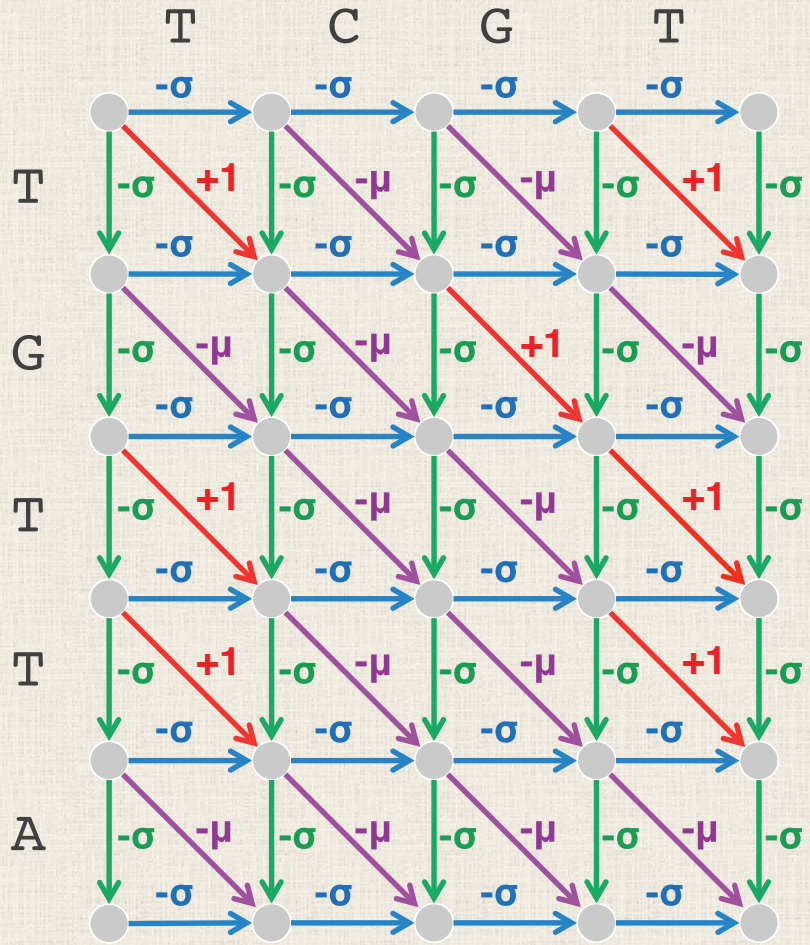
Answer: Slight modification to alignment network ... a longest path will yield an alignment of maximum score!



Strengthening Alignment Scoring

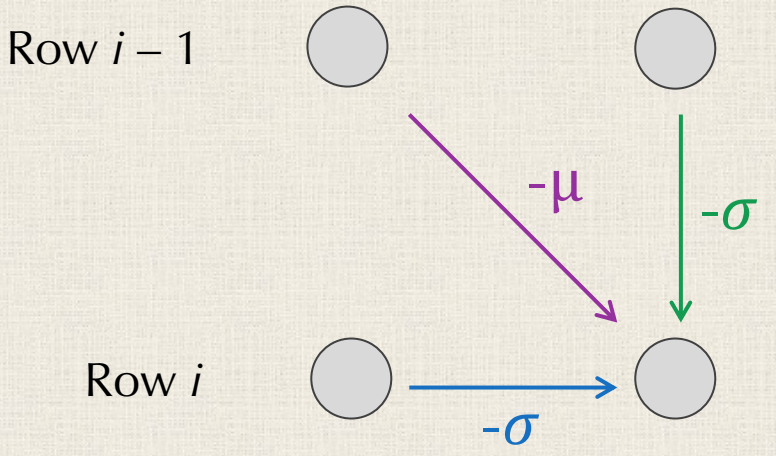
Answer: Slight modification to alignment network ... a longest path will yield an alignment of maximum score!

Exercise: What is the recurrence relation?



Two Cases: Mismatch vs. Match

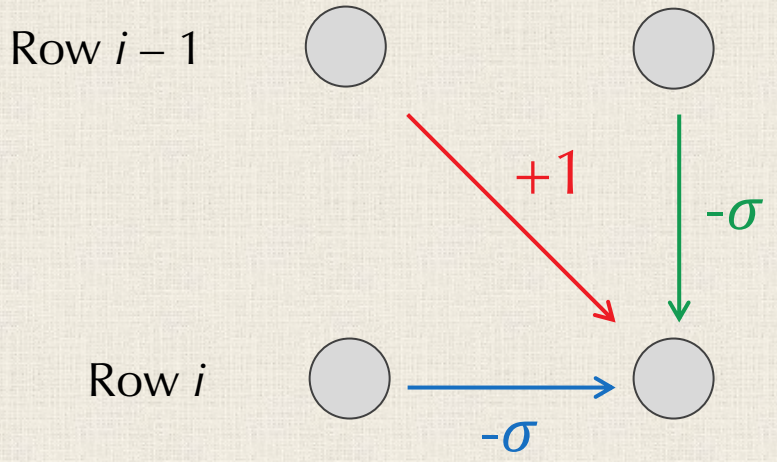
Case 1



length(i, j) = maximum of:

- *length*$(i-1, j) - \sigma$
- *length*$(i, j-1) - \sigma$
- *length*$(i-1, j-1) - \mu$

Case 2



length(i, j) = maximum of:

- *length*$(i-1, j) - \sigma$
- *length*$(i, j-1) - \sigma$
- *length*$(i-1, j-1) + 1$

Finding a Max-Score Global Alignment

Global Alignment Scoring Problem:

- **Input:** Two strings along with mismatch and insertion/deletion penalties μ and σ.
- **Output:** The global alignment "score matrix" for the strings according to these penalties.

Code Challenge: Write a function solving this problem.

Solving Global Alignment

Global Alignment Problem:

- **Input:** Two strings along with mismatch and insertion/deletion penalties μ and σ.
- **Output:** An alignment of the strings with maximum alignment score.

Code Challenge: Write a function solving this problem.

Hint: Use your solution to the previous problem along with your knowledge of backtracking.

Applying to Real Data

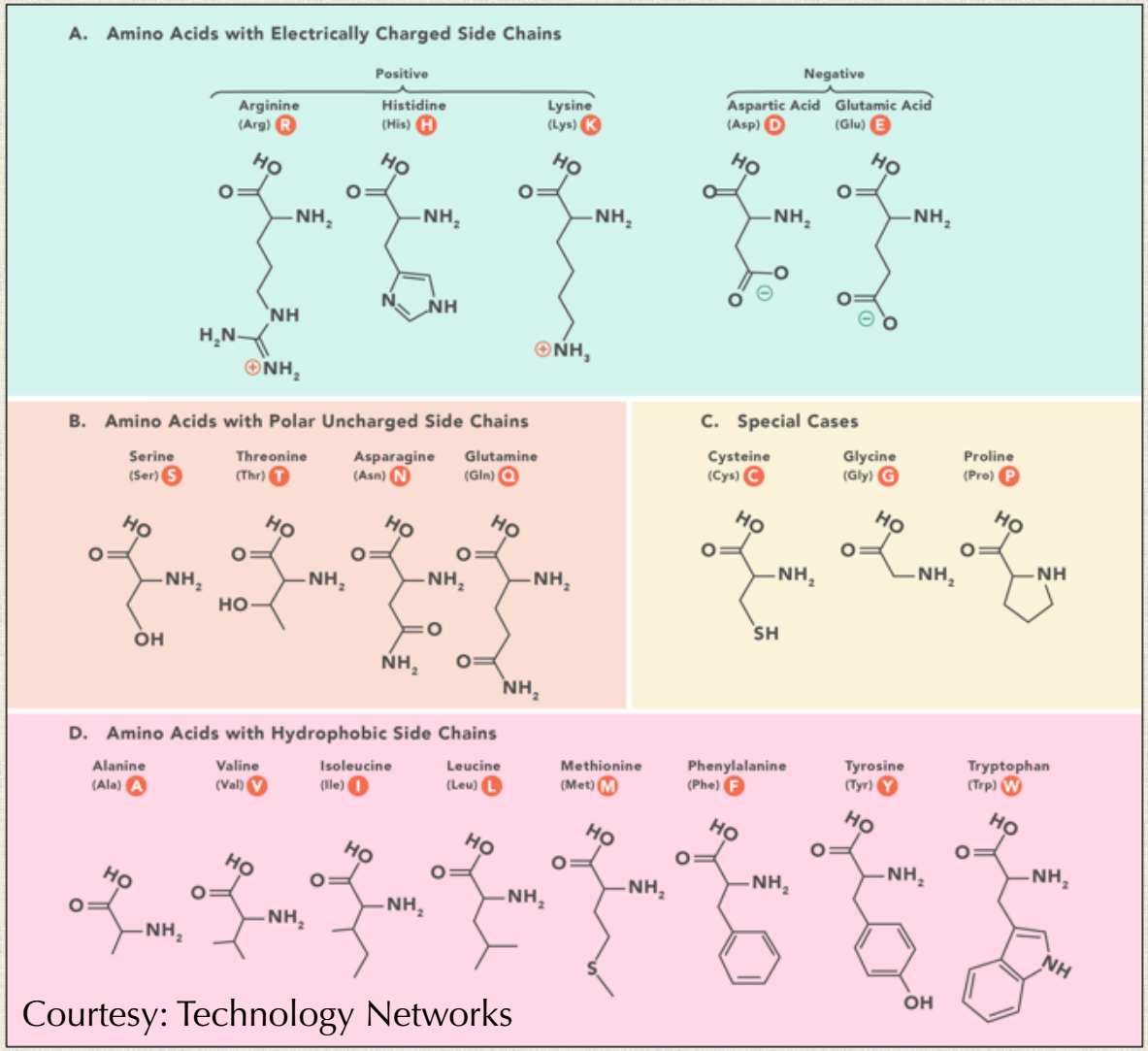
STOP: Let's apply this to the same protein (say, hemoglobin subunit beta ☹) in a few different species. What do you think we will see?

- *Homo sapiens* vs. *Gorilla gorilla gorilla*
- *Homo sapiens* vs. *Bos Taurus* (cow)
- *Homo sapiens* vs. *Danio rerio* (zebrafish)

Homo sapiens:	https://www.uniprot.org/uniprot/P69905
Gorilla gorilla gorilla:	https://www.uniprot.org/uniprot/P01923
Bos taurus:	https://www.uniprot.org/uniprot/P01966
Danio rerio:	https://www.uniprot.org/uniprot/Q90487

EMBOSS "Needle" server: https://www.ebi.ac.uk/Tools/psa/emboss_needle/

Amino acids' side chain variety produces different chemical properties



Courtesy: Technology Networks

Results of Hemoglobin Alignments

Note: "|" means exact similarity, ":" means strong similarity, and "." means weak similarity.

STOP: What is our hypothesis?

```
HBA_HUMAN          1 MVLSPADKTNVKAAWGKVGAHAGEYGAEALERMFLSFPTTKTYFPHFDLS     50
                     |||||||||||||||||||||||:|||||||||||||||||||||||||||
HBA_GORGO          1 -VLSPADKTNVKAAWGKVGAHAGDYGAEALERMFLSFPTTKTYFPHFDLS     49

HBA_HUMAN         51 HGSAQVKGHGKKVADALTNAVAHVDDMPNALSALSDLHAHKLRVDPVNFK    100
                     ||||||||||||||||||||||||||||||||||||||||||||||||||||
HBA_GORGO         50 HGSAQVKGHGKKVADALTNAVAHVDDMPNALSALSDLHAHKLRVDPVNFK     99

HBA_HUMAN        101 LLSHCLLVTLAAHLPAEFTPAVHASLDKFLASVSTVLTSKYR    142
                     |||||||||||||||||||||||||||||||||||||||||||
HBA_GORGO        100 LLSHCLLVTLAAHLPAEFTPAVHASLDKFLASVSTVLTSKYR    141
```

```
HBA_HUMAN          1 MVLSPADKTNVKAAWGKVGAHAGEYGAEALERMFLSFPTTKTYFPHFDLS     50
                     ||||.|||.||||||||||.||.||||||||||||||||||||||||||||
HBA_BOVIN          1 MVLSAADKGNVKAAWGKVGGHAAEYGAEALERMFLSFPTTKTYFPHFDLS     50

HBA_HUMAN         51 HGSAQVKGHGKKVADALTNAVAHVDDMPNALSALSDLHAHKLRVDPVNFK    100
                     ||||||||||.|||.|||.||.|:||:|.|||.||||||||||||||||||
HBA_BOVIN         51 HGSAQVKGHGAKVAAALTKAVEHLDDLPGALSELSDLHAHKLRVDPVNFK    100

HBA_HUMAN        101 LLSHCLLVTLAAHLPAEFTPAVHASLDKFLASVSTVLTSKYR    142
                     ||||.||||||:|||::||||||||||||||:||||||||||
HBA_BOVIN        101 LLSHSLLVTLASHLPSDFTPAVHASLDKFLANVSTVLTSKYR    142
```

```
HBA_HUMAN          1 MVLSPADKTNVKAAWGKVGAHAGEYGAEALERMFLSFPTTKTYFPHF-DL     49
                     |.||..||..|||.|.|:...|.|.|||||.||...:|.|||||.|: ||
HBA_DANRE          1 MSLSDTDKAVVKAIWAKISPKADEIGAEALARMLTVYPQTKTYFSHWADL     50

HBA_HUMAN         50 SHGSAQVKGHGKKVADALTNAVAHVDDMPNALSALSDLHAHKLRVDPVNF     99
                     |.||..||.|||.:..|:..|::.:||:...|:|||:|||.||||||.||
HBA_DANRE         51 SPGSGPVKKHGKTIMGAVGEAISKIDDLVGGLAALSELHAFKLRVDPANF    100

HBA_HUMAN        100 KLLSHCLLVTLAAHLPAEFTPAVHASLDKFLASVSTVLTSKYR    142
                     |:|||.::|.:|...||:|||.||.|:|||..:::..|:.|||
HBA_DANRE        101 KILSHNVIVVIAMLFPADFTPEVHVSVDKFFNNLALALSEKYR    143
```

Biology c. 1963

From the point of view of hemoglobin structure, it appears that gorilla is just an abnormal human.

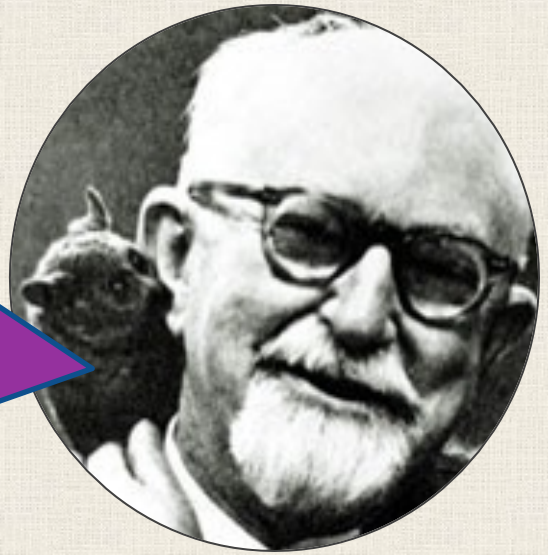
Émile Zuckerkandl

Biology c. 1963

From the point of view of hemoglobin structure, it appears that gorilla is just an abnormal human.

Émile Zuckerkandl

...that is of course nonsense. What the comparison really indicates is that hemoglobin is a bad choice and has nothing to tell us about attributes.



Gaylord Simpson

Biology c. 2020

STOP: Now let's apply our approach to align the SARS-CoV and SARS-CoV-2 genomes and see what we can find (it may take a few minutes to run). Why must we align these as DNA strings and not as protein strings?

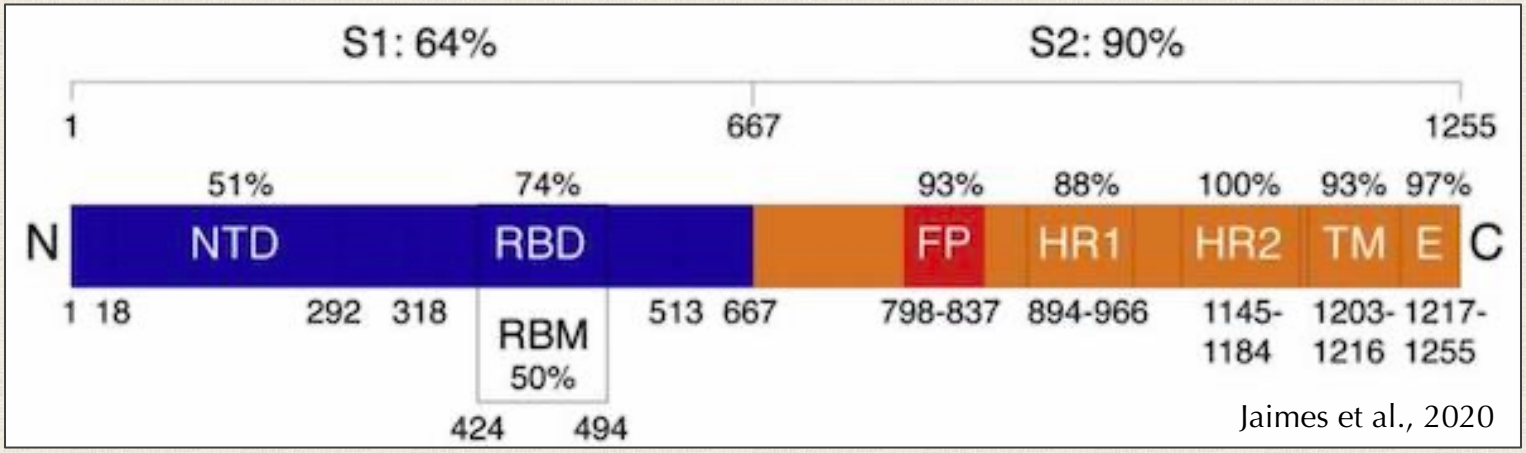
SARS-CoV genome (2003): https://www.ncbi.nlm.nih.gov/nuccore/30271926
SARS-CoV-2 original genome (2020): https://www.ncbi.nlm.nih.gov/nuccore/1798174254

NCBI alignment viewer: https://www.ncbi.nlm.nih.gov/projects/msaviewer/.

FROM GLOBAL TO LOCAL ALIGNMENT

Finding "Local" Similarities

Real genes have *variable* and *conserved* regions; the figure below shows the sequence similarity of the spike protein between SARS-CoV and SARS-CoV-2.



Jaimes et al., 2020

We also will need "local" alignment to compare genes against a database

Database Comparison Problem:

- **Input:** A string *query* and a (much longer) string *database*.
- **Output:** One or more "high-scoring" similarities between *query* (or a substring of *query*) and some substring of *database*.

This (poorly defined) problem is probably the most frequent application in computational biology.

Finding "Local" Similarities

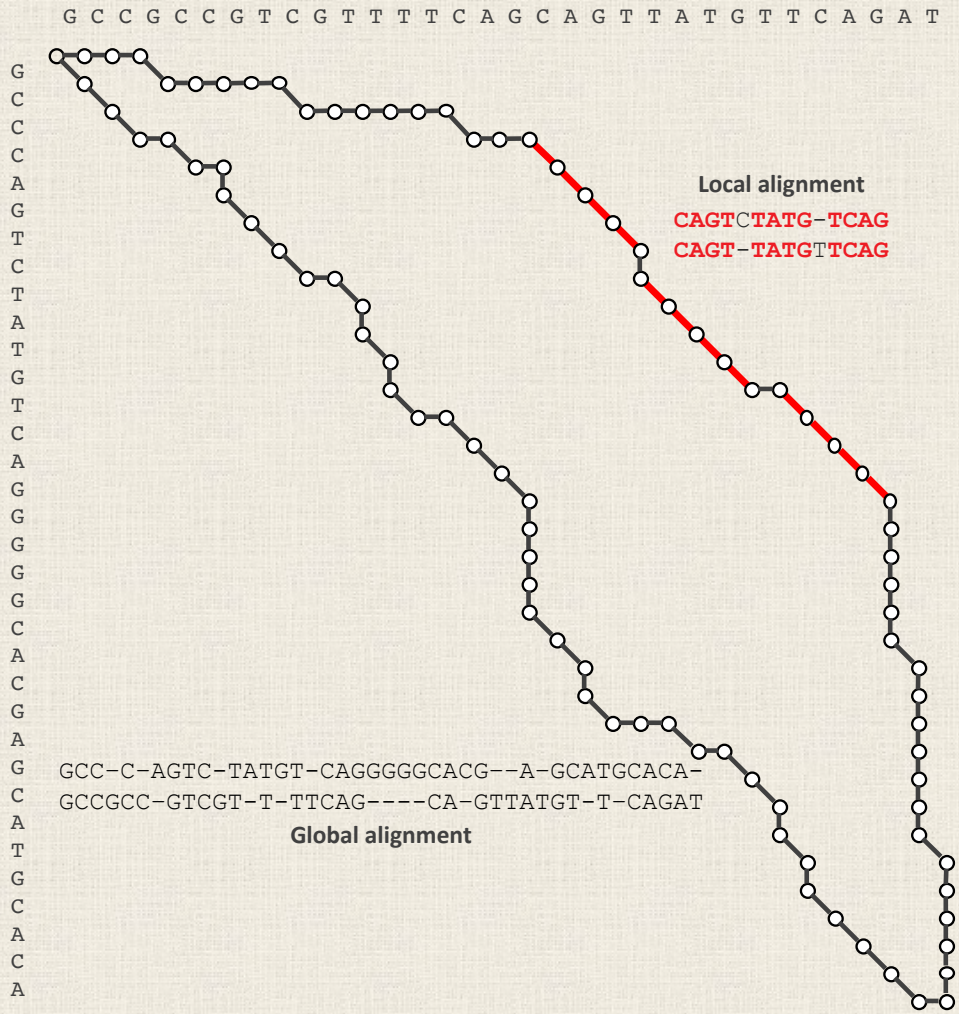
```
GCC-C-AGTC-TATGT-CAGGGGGCACG--A-GCATGCACA-
GCCGCC-GTCGT-T-TTCAG----CA-GTTATGT-T-CAGAT
```

Exercise: Score these alignments ($\sigma = \mu = 1$). Which alignment is "better"? Which gets the higher score?

```
---G----C------C--CAGTCTATG-TCAGGGGGCACGAGCATGCACA
GCCGCCGTCGTTTTCAGCAGT-TATGTTCAG------A-------T-----
```

Visualizing Local Alignments

Local alignments may be well away from "main diagonal" because they have a lot of indels on ends of the alignment.



Revisiting Global Alignment

Global Alignment Problem:

- **Input:** Two strings.
- **Output:** An alignment of the strings with maximum alignment score.

STOP: How can we reformulate the problem to find areas of "local" similarity?

Local Alignment is the Best Scoring Global Alignment of Substrings

Local Alignment Problem:

- **Input:** Two strings *v* and *w* and scoring parameters.
- **Output:** The best scoring alignment of substrings of *v* and *w*.

Local Alignment is the Best Scoring Global Alignment of Substrings

Local Alignment Problem:

- **Input:** Two strings *v* and *w* and scoring parameters.
- **Output:** The local alignment "score matrix" for the strings according to these penalties.

STOP: One idea for solving this is to solve the Global Alignment Problem for every pair of substrings of *v* and *w*. Why is this an issue?

Local Alignment is the Best Scoring Global Alignment of Substrings

Local Alignment Problem:

- **Input:** Two strings v and w and scoring parameters.
- **Output:** The best scoring alignment of substrings of v and w.

Answer: There are $C(|v|, 2)$ substrings of v and $C(|w|, 2)$ substrings of w. As a result we have about $|v|^2|w|^2$ alignments to construct!

This was known in 1970 and remained unsolved …

Ten Years Go By …

A general method applicable to the search for similarities in the amino ...

https://www.sciencedirect.com/science/article/pii/0022283670900574

by SB Needleman - 1970 - Cited by 12553 - Related articles

A computer adaptable **method** for **finding similarities in the amino acid sequences of two proteins** has been developed. From these findings it is possible to determine whether significant homology exists between the **proteins**. ... The maximum match is a number dependent upon the **similarity** of the **sequences**.

Identification of common molecular subsequences. - NCBI

https://www.ncbi.nlm.nih.gov/pubmed/7265238 ▼

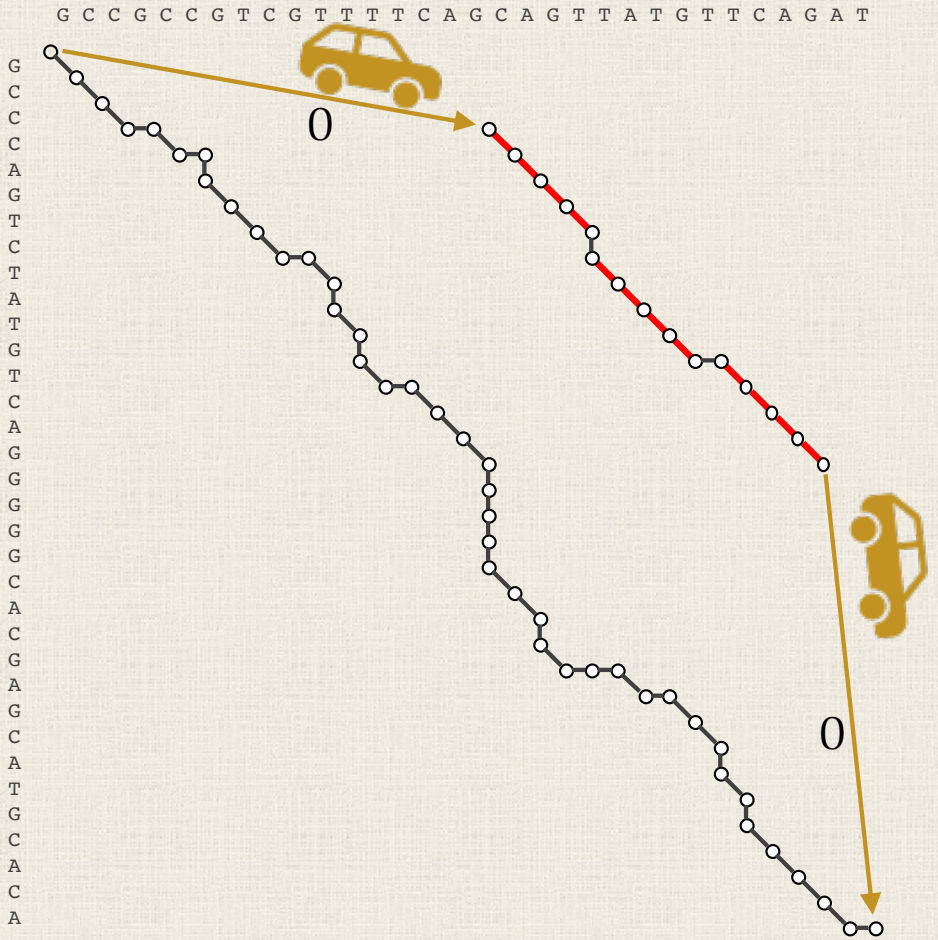
by TF Smith - 1981 - Cited by 11181 - Related articles

Identification of common molecular subsequences. Smith TF, Waterman MS. PMID: 7265238; [Indexed for ... MeSH terms. Base **Sequence***; Models, Chemical *

“Free Rides” for Local Alignment

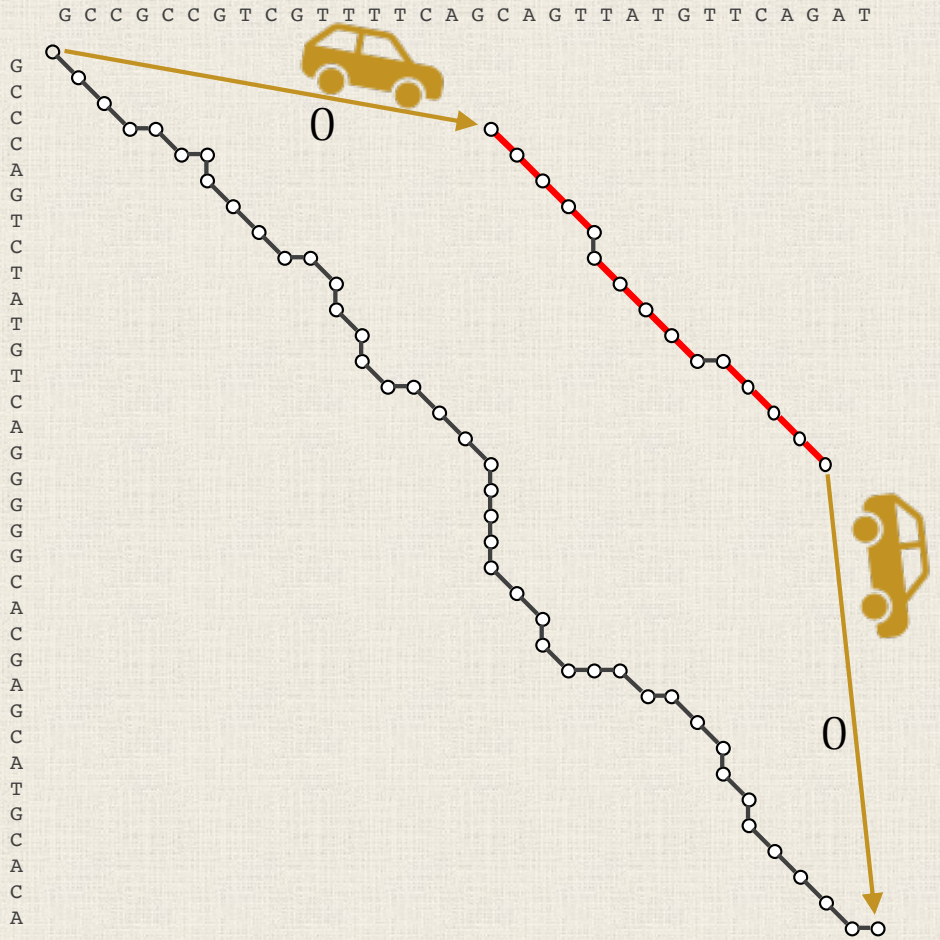
Add a zero-weight edge from the source to every node and from every node to the sink.

This will allow a local alignment to start and end anywhere with no penalty.



"Free Rides" for Local Alignment

Exercise: What is the recurrence relation for the local alignment problem?

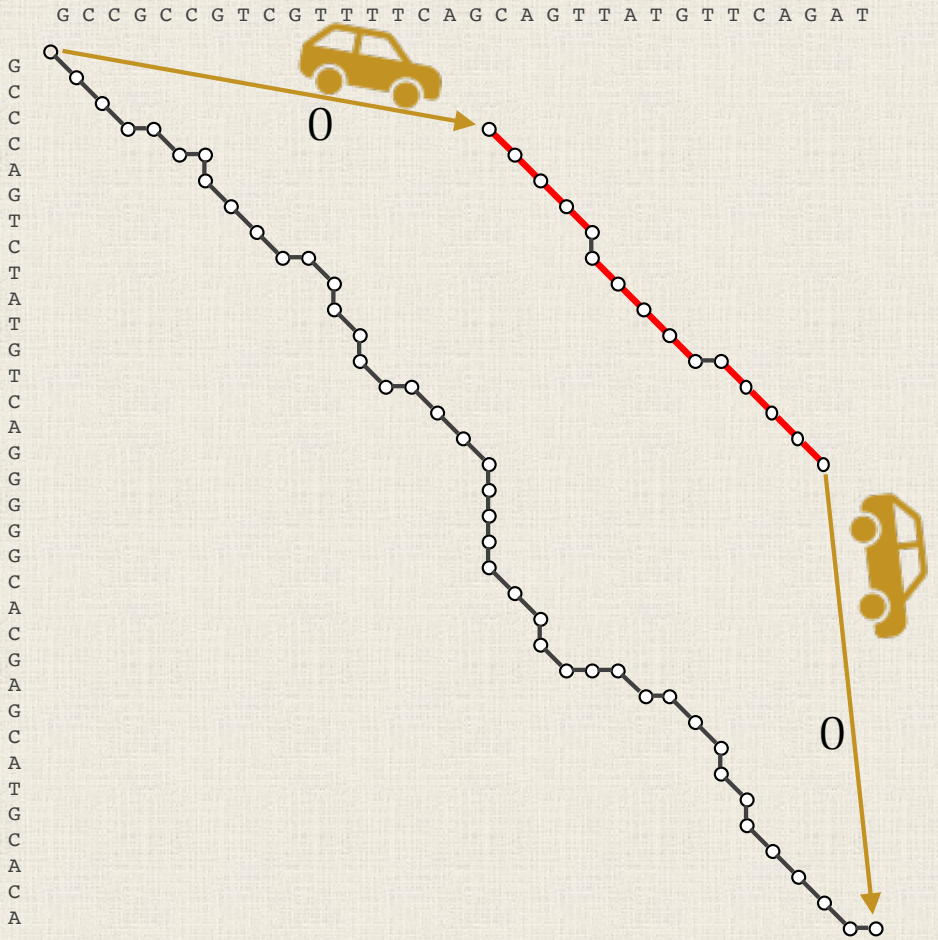


"Free Rides" for Local Alignment

Answer: It is given by

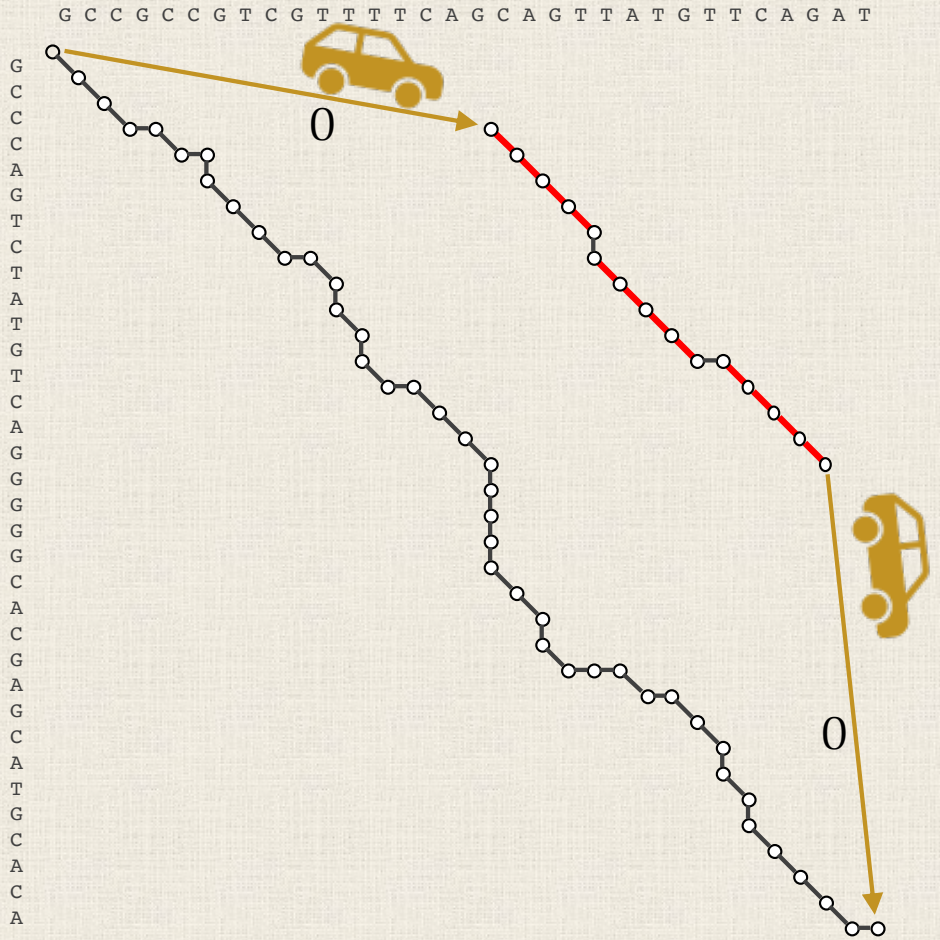
$$s_{i,j} = \max \begin{cases} 0 & \\ s_{i-1,j} & + Score(v_i, -) \\ s_{i,j-1} & + Score(-, w_j) \\ s_{i-1,j-1} & + Score(v_i, w_j) \end{cases}$$

where the scores here are $-\sigma$, $-\sigma$, and either $+1$ or $-\mu$ (depending on a match vs. a mismatch).



"Free Rides" for Local Alignment

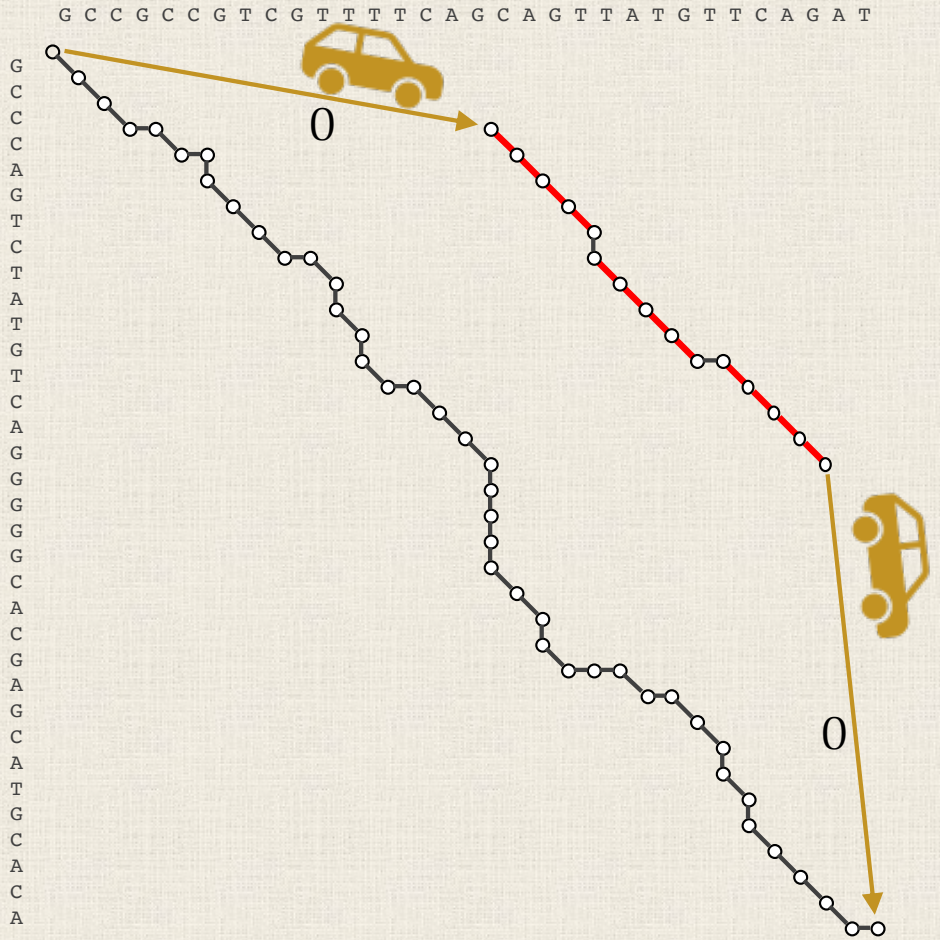
Exercise: After we apply the recurrence, where should we start backtracking? (That is, where does the best local alignment end?)



"Free Rides" for Local Alignment

Exercise: After we apply the recurrence, where should we start backtracking? (That is, where does the best local alignment end?)

Answer: Wherever the *maximum* value of the scoring table is.



The Solution to a Problem Unsolved for Ten Years Nearly Fits on One Slide

J. Mol. Biol. (1981), **147**, 195–197

Identification of Common Molecular Subsequences

The identification of maximally homologous subsequences among sets of long sequences is an important problem in molecular sequence analysis. The problem is straightforward only if one restricts consideration to contiguous subsequences (segments) containing no internal deletions or insertions. The more general problem has its solution in an extension of sequence metrics (Sellers 1974; Waterman *et al.*, 1976) developed to measure the minimum number of "events" required to convert one sequence into another.

These developments in the modern sequence analysis began with the heuristic homology algorithm of Needleman & Wunsch (1970) which first introduced an iterative matrix method of calculation. Numerous other heuristic algorithms have been suggested including those of Fitch (1966) and Dayhoff (1969). More mathematically rigorous algorithms were suggested by Sankoff (1972), Reichert *et al.* (1973) and Beyer *et al.* (1979), but these were generally not biologically satisfying or interpretable. Success came with Sellers (1974) development of a true metric measure of the distance between sequences. This metric was later generalized by Waterman *et al.* (1976) to include deletions/insertions of arbitrary length. This metric represents the minimum number of "mutational events" required to convert one sequence into another. It is of interest to note that Smith *et al.* (1980) have recently shown that under some conditions the generalized Sellers metric is equivalent to the original homology algorithm of Needleman & Wunsch (1970).

In this letter we extend the above ideas to find a pair of segments, one from each of two long sequences, such that there is no other pair of segments with greater similarity (homology). The similarity measure used here allows for arbitrary length deletions and insertions.

Algorithm

The two molecular sequences will be $\underset{\sim}{A}=a_1a_2\ldots a_n$ and $\underset{\sim}{B}=b_1b_2\ldots b_m$. A similarity $s(a,b)$ is given between sequence elements a and b. Deletions of length k are given weight W_k. To find pairs of segments with high degrees of similarity, we set up a matrix H. First set

$$H_{k0}=H_{0l}=0 \text{ for } 0\leq k\leq n \text{ and } 0\leq l\leq m.$$

Preliminary values of H have the interpretation that H_{ij} is the maximum similarity of two segments *ending* in a_i and b_j, respectively. These values are obtained from the relationship

$$H_{ij}=\max\{H_{i-1,j-1}+s(a_i,b_j), \max_{k\geq 1}\{H_{i-k,j}-W_k\}, \max_{l\geq 1}\{H_{i,j-l}-W_l\}, 0\}, \quad (1)$$

$1\leq i\leq n$ and $1\leq j\leq m$.

0022–2836/80/090195–03 $02.00/0

The formula for H_{ij} follows by considering the possibilities for ending the segments at any a_i and b_j.

(1) If a_i and b_j are associated, the similarity is

$$H_{i-1,j-1}+s(a_i,b_j).$$

(2) If a_i is at the end of a deletion of length k, the similarity is

$$H_{i-k,j}-W_k.$$

(3) If b_j is at the end of a deletion of length l, the similarity is

$$H_{i-k,j}-W_l.$$

(4) Finally, a zero is included to prevent calculated negative similarity, indicating no similarity up to a_i and b_j.†

The pair of segments with maximum similarity is found by first locating the maximum element of H. The other matrix elements leading to this maximum value are than sequentially determined with a traceback procedure ending with an element of H equal to zero. This procedure identifies the segments as well as produces the corresponding alignment. The pair of segments with the next best similarity is found by applying the traceback procedure to the second largest element of H not associated with the first traceback.

A simple example is given in Figure 1. In this example the parameters $s(a_ib_j)$ and W_k required were chosen on an *a priori* statistical basis. A match, $a_i=b_j$, produced an $s(a_ib_j)$ value of unity while a mismatch produced a minus one-third. These values have an average for long, random sequences over an equally probable four letter set of zero. The deletion weight must be chosen to be at least equal to the difference between a match and a mismatch. The value used here was $W_k=1{\cdot}0+1/3{*}k$.

	Δ	C	A	G	C	C	U	C	G	C	U	U	A	G
Δ	0·0	0·0	0·0	0·0	0·0	0·0	0·0	0·0	0·0	0·0	0·0	0·0	0·0	0·0
A	0·0	0·0	1·0	0·0	0·0	0·0	0·0	0·0	0·0	0·0	0·0	0·0	1·0	0·0
A	0·0	0·0	1·0	0·7	0·0	0·0	0·0	0·0	0·0	0·0	0·0	0·0	1·0	0·7
U	0·0	0·0	0·0	0·7	0·3	0·0	1·0	0·0	0·0	0·0	1·0	1·0	0·0	0·7
G	0·0	0·0	0·0	1·0	0·3	0·0	0·0	0·7	1·0	0·0	0·0	0·7	0·7	1·0
C	0·0	1·0	0·0	0·0	2·0	1·3	0·3	1·0	0·3	2·0	0·7	0·3	0·3	0·3
C	0·0	1·0	0·7	0·0	1·0	3·0	1·7	1·3	1·0	1·3	1·7	0·3	0·0	0·0
A	0·0	0·0	2·0	0·7	0·3	1·7	2·7	1·3	1·0	0·7	1·0	1·3	1·3	0·0
U	0·0	0·0	0·7	1·7	0·3	1·3	2·7	2·3	1·0	0·7	1·7	2·0	1·0	1·0
U	0·0	0·0	0·3	0·3	1·3	1·0	2·3	2·3	2·0	0·7	1·7	2·7	1·7	1·0
G	0·0	0·0	0·0	1·3	0·0	1·0	1·0	2·0	3·3	2·0	1·7	1·3	2·3	2·7
A	0·0	0·0	1·0	0·0	1·0	0·3	0·7	0·7	2·0	3·0	1·7	1·3	2·3	2·0
C	0·0	1·0	0·0	0·7	1·0	2·0	0·7	1·7	1·7	3·0	2·7	1·3	1·0	2·0
G	0·0	0·0	0·7	1·0	0·3	0·7	1·7	0·3	2·7	1·7	2·7	2·3	1·0	2·0
G	0·0	0·0	0·0	1·7	0·7	0·3	0·3	1·3	1·3	2·3	1·3	2·3	2·0	2·0

FIG. 1. H_{ij} matrix generated from the application of eqn (1) to the sequences A-A-U-G-C-C-A-U-U-G-A-C-G-G and C-A-G-C-C-U-C-G-C-U-U-A-G. The underlined elements indicate the trackback path from the maximal element 3·30.

† Zero need not be included unless there are negative values of $s(a,b)$.

The Solution to a Problem Unsolved for Ten Years Nearly Fits on One Slide

Note, in this simple example, that the alignment obtained:

-G-C-C-A-U-U-G-
-G-C-C— U-C-G-

contains both a mismatch and an internal deletion. It is the identification of the latter which has not been previously possible in any rigorous manner.

This algorithm not only puts the search for pairs of maximally similar segments on a mathematically rigorous basis but it can be efficiently and simply programmed on a computer.

Northern Michigan University T. F. SMITH

Los Alamos Scientific Laboratory M. S. WATERMAN
P.O. Box 1663, Los Alamos
N. Mex. 87545, U.S.A.

Received 14 July 1980

REFERENCES

Beyer, W. A., Smith, T. F., Stein, M. L. & Ulam, S. M. (1979). *Math. Biosci.* **19**, 9–25.
Dayhoff, M. O. (1969). *Atlas of Protein Sequence and Structure*, National Biomedical Research Foundation, Silver Springs, Maryland.
Fitch, W. M. (1966). *J. Mol. Biol.* **16**, 9–13.
Needleman, S. B. & Wunsch, C. D. (1970). *J. Mol. Biol.* **48**, 443–453.
Reichert, T. A., Cohen, D. N. & Wong, A. K. C. (1973). *J. Theoret. Biol.* **42**, 245–261.
Sankoff, D. (1972). *Proc. Nat. Acad. Sci., U.S.A.* **61**, 4–6.
Sellers, P. H. (1974). *J. Appl. Math. (Siam)*, **26**, 787–793.
Smith, T. F., Waterman, M. S. & Fitch, W. M. (1981). *J. Mol. Evol.* In the press.
Waterman, M. S., Smith, T. F. & Beyer, W. A. (1976). *Advan. Math.* **20**, 367–387.

Note added in proof: A weighting similar to that given above was independently developed by Walter Goad of Los Alamos Scientific Laboratory.

Smith and Waterman's Scoring Table

	Δ	C	A	G	C	C	U	C	G	C	U	U	A	G
Δ	0·0	0·0	0·0	0·0	0·0	0·0	0·0	0·0	0·0	0·0	0·0	0·0	0·0	0·0
A	0·0	0·0	1·0	0·0	0·0	0·0	0·0	0·0	0·0	0·0	0·0	0·0	1·0	0·0
A	0·0	0·0	1·0	0·7	0·0	0·0	0·0	0·0	0·0	0·0	0·0	0·0	1·0	0·7
U	0·0	0·0	0·0	0·7	0·3	0·0	1·0	0·0	0·0	0·0	1·0	1·0	0·0	0·7
G	0·0	0·0	0·0	1·0	0·3	0·0	0·0	0·7	1·0	0·0	0·0	0·7	0·7	1·0
C	0·0	1·0	0·0	0·0	2·0	1·3	0·3	1·0	0·3	2·0	0·7	0·3	0·3	0·3
C	0·0	1·0	0·7	0·0	1·0	3·0	1·7	1·3	1·0	1·3	1·7	0·3	0·0	0·0
A	0·0	0·0	2·0	0·7	0·3	1·7	2·7	1·3	1·0	0·7	1·0	1·3	1·3	0·0
U	0·0	0·0	0·7	1·7	0·3	1·3	2·7	2·3	1·0	0·7	1·7	2·0	1·0	1·0
U	0·0	0·0	0·3	0·3	1·3	1·0	2·3	2·3	2·0	0·7	1·7	2·7	1·7	1·0
G	0·0	0·0	0·0	1·3	0·0	1·0	1·0	2·0	3·3	2·0	1·7	1·3	2·3	2·7
A	0·0	0·0	1·0	0·0	1·0	0·3	0·7	0·7	2·0	3·0	1·7	1·3	2·3	2·0
C	0·0	1·0	0·0	0·7	1·0	2·0	0·7	1·7	1·7	3·0	2·7	1·3	1·0	2·0
G	0·0	0·0	0·7	1·0	0·3	0·7	1·7	0·3	2·7	1·7	2·7	2·3	1·0	2·0
G	0·0	0·0	0·0	1·7	0·7	0·3	0·3	1·3	1·3	2·3	1·3	2·3	2·0	2·0

FIG. 1. H_{ij} matrix generated from the application of eqn (1) to the sequences A-A-U-G-C-C-A-U-U-G-A-C-G-G and C-A-G-C-C-U-C-G-C-U-U-A-G. The underlined elements indicate the trackback path from the maximal element 3·30.

Local Alignment is the Best Scoring Global Alignment of Substrings

Local Alignment Scoring Problem:

- **Input:** Two strings *v* and *w* and insertion/deletion penalties μ and σ.
- **Output:** The score of a maximum-score alignment over all substrings of *v* and *w*.

Code Challenge: Solve the Local Alignment Scoring Problem.

Local Alignment is the Best Scoring Global Alignment of Substrings

Local Alignment Problem:

- **Input:** Two strings *v* and *w* and insertion/deletion penalties μ and σ.
- **Output:** The best scoring alignment of substrings of *v* and *w*, along with the positions of the start and end of these substrings in *v* and *w*.

Code Challenge: Solve the Local Alignment Problem.

Local Alignment is the Best Scoring Global Alignment of Substrings

Local Alignment Problem:

- **Input:** Two strings v and w and insertion/deletion penalties μ and σ.
- **Output:** The best scoring alignment of substrings of v and w, along with the positions of the start and end of these substrings in v and w.

STOP: Let's apply local sequence alignment to align the known spike protein from SARS-CoV to the SARS-CoV-2 genome. What would you expect to see?

ADDITIONAL ISSUES IN SEQUENCE ALIGNMENT

Scoring Matrices for Aligning Proteins

Scoring matrix: penalizes different matches/mismatches differently.

BLOSUM62 Matrix

	Ala	Arg	Asn	Asp	Cys	Gln	Glu	Gly	His	Ile	Leu	Lys	Met	Phe	Pro	Ser	Thr	Trp	Tyr	Val
Ala	4																			
Arg	−1	5																		
Asn	−2	0	6																	
Asp	−2	−2	1	6																
Cys	0	−3	−3	−3	9															
Gln	−1	1	0	0	−3	5														
Glu	−1	0	0	2	−4	2	5													
Gly	0	−2	0	−1	−3	−2	−2	6												
His	−2	0	1	−1	−3	0	0	−2	8											
Ile	−1	−3	−3	−3	−1	−3	−3	−4	−3	4										
Leu	−1	−2	−3	−4	−1	−2	−3	−4	−3	2	4									
Lys	−1	2	0	−1	−3	1	1	−2	−1	−3	−2	5								
Met	−1	−1	−2	−3	−1	0	−2	−3	−2	1	2	−1	5							
Phe	−2	−3	−3	−3	−2	−3	−3	−3	−1	0	0	−3	0	6						
Pro	−1	−2	−2	−1	−3	−1	−1	−2	−2	−3	−3	−1	−2	−4	7					
Ser	1	−1	1	0	−1	0	0	0	−1	−2	−2	0	−1	−2	−1	4				
Thr	0	−1	0	−1	−1	−1	−1	−2	−2	−1	−1	−1	−1	−2	−1	1	5			
Trp	−3	−3	−4	−4	−2	−2	−3	−2	−2	−3	−2	−3	−1	1	−4	−3	−2	11		
Tyr	−2	−2	−2	−3	−2	−1	−2	−3	2	−1	−1	−2	−1	3	−3	−2	−2	2	7	
Val	0	−3	−3	−3	−1	−2	−2	−3	−3	3	1	−2	1	−1	−2	−2	0	−3	−1	4

Scoring Matrices for Aligning Proteins

STOP 1: Why would we reward/penalize different pairs differently?

BLOSUM62 Matrix

	Ala	Arg	Asn	Asp	Cys	Gln	Glu	Gly	His	Ile	Leu	Lys	Met	Phe	Pro	Ser	Thr	Trp	Tyr	Val
Ala	4																			
Arg	−1	5																		
Asn	−2	0	6																	
Asp	−2	−2	1	6																
Cys	0	−3	−3	−3	9															
Gln	−1	1	0	0	−3	5														
Glu	−1	0	0	2	−4	2	5													
Gly	0	−2	0	−1	−3	−2	−2	6												
His	−2	0	1	−1	−3	0	0	−2	8											
Ile	−1	−3	−3	−3	−1	−3	−3	−4	−3	4										
Leu	−1	−2	−3	−4	−1	−2	−3	−4	−3	2	4									
Lys	−1	2	0	−1	−3	1	1	−2	−1	−3	−2	5								
Met	−1	−1	−2	−3	−1	0	−2	−3	−2	1	2	−1	5							
Phe	−2	−3	−3	−3	−2	−3	−3	−3	−1	0	0	−3	0	6						
Pro	−1	−2	−2	−1	−3	−1	−1	−2	−2	−3	−3	−1	−2	−4	7					
Ser	1	−1	1	0	−1	0	0	0	−1	−2	−2	0	−1	−2	−1	4				
Thr	0	−1	0	−1	−1	−1	−1	−2	−2	−1	−1	−1	−1	−2	−1	1	5			
Trp	−3	−3	−4	−4	−2	−2	−3	−2	−2	−3	−2	−3	−1	1	−4	−3	−2	11		
Tyr	−2	−2	−2	−3	−2	−1	−2	−3	2	−1	−1	−2	−1	3	−3	−2	−2	2	7	
Val	0	−3	−3	−3	−1	−2	−2	−3	−3	3	1	−2	1	−1	−2	−2	0	−3	−1	4

Scoring Matrices for Aligning Proteins

STOP 2: How would researchers have formed this matrix?

BLOSUM62 Matrix

	Ala	Arg	Asn	Asp	Cys	Gln	Glu	Gly	His	Ile	Leu	Lys	Met	Phe	Pro	Ser	Thr	Trp	Tyr	Val
Ala	4																			
Arg	−1	5																		
Asn	−2	0	6																	
Asp	−2	−2	1	6																
Cys	0	−3	−3	−3	9															
Gln	−1	1	0	0	−3	5														
Glu	−1	0	0	2	−4	2	5													
Gly	0	−2	0	−1	−3	−2	−2	6												
His	−2	0	1	−1	−3	0	0	−2	8											
Ile	−1	−3	−3	−3	−1	−3	−3	−4	−3	4										
Leu	−1	−2	−3	−4	−1	−2	−3	−4	−3	2	4									
Lys	−1	2	0	−1	−3	1	1	−2	−1	−3	−2	5								
Met	−1	−1	−2	−3	−1	0	−2	−3	−2	1	2	−1	5							
Phe	−2	−3	−3	−3	−2	−3	−3	−3	−1	0	0	−3	0	6						
Pro	−1	−2	−2	−1	−3	−1	−1	−2	−2	−3	−3	−1	−2	−4	7					
Ser	1	−1	1	0	−1	0	0	0	−1	−2	−2	0	−1	−2	−1	4				
Thr	0	−1	0	−1	−1	−1	−1	−2	−2	−1	−1	−1	−1	−2	−1	1	5			
Trp	−3	−3	−4	−4	−2	−2	−3	−2	−2	−3	−2	−3	−1	1	−4	−3	−2	11		
Tyr	−2	−2	−2	−3	−2	−1	−2	−3	2	−1	−1	−2	−1	3	−3	−2	−2	2	7	
Val	0	−3	−3	−3	−1	−2	−2	−3	−3	3	1	−2	1	−1	−2	−2	0	−3	−1	4

Scoring Matrices for Aligning Proteins

STOP 3: How would a scoring matrix change our alignment network?

BLOSUM62 Matrix

	Ala	Arg	Asn	Asp	Cys	Gln	Glu	Gly	His	Ile	Leu	Lys	Met	Phe	Pro	Ser	Thr	Trp	Tyr	Val
Ala	4																			
Arg	−1	5																		
Asn	−2	0	6																	
Asp	−2	−2	1	6																
Cys	0	−3	−3	−3	9															
Gln	−1	1	0	0	−3	5														
Glu	−1	0	0	2	−4	2	5													
Gly	0	−2	0	−1	−3	−2	−2	6												
His	−2	0	1	−1	−3	0	0	−2	8											
Ile	−1	−3	−3	−3	−1	−3	−3	−4	−3	4										
Leu	−1	−2	−3	−4	−1	−2	−3	−4	−3	2	4									
Lys	−1	2	0	−1	−3	1	1	−2	−1	−3	−2	5								
Met	−1	−1	−2	−3	−1	0	−2	−3	−2	1	2	−1	5							
Phe	−2	−3	−3	−3	−2	−3	−3	−3	−1	0	0	−3	0	6						
Pro	−1	−2	−2	−1	−3	−1	−1	−2	−2	−3	−3	−1	−2	−4	7					
Ser	1	−1	1	0	−1	0	0	0	−1	−2	−2	0	−1	−2	−1	4				
Thr	0	−1	0	−1	−1	−1	−1	−2	−2	−1	−1	−1	−1	−2	−1	1	5			
Trp	−3	−3	−4	−4	−2	−2	−3	−2	−2	−3	−2	−3	−1	1	−4	−3	−2	11		
Tyr	−2	−2	−2	−3	−2	−1	−2	−3	2	−1	−1	−2	−1	3	−3	−2	−2	2	7	
Val	0	−3	−3	−3	−1	−2	−2	−3	−3	3	1	−2	1	−1	−2	−2	0	−3	−1	4

Reformulating Global Alignment

Global Alignment Problem:

- **Input:** Two strings and a scoring matrix.
- **Output:** An alignment of the strings with maximum alignment score, given this scoring matrix.

Problem 2: Moving to Multiple Sequences

Multiple Alignment Problem:

- **Input:** A collection of *t* strings.
- **Output:** A multiple alignment of these strings having maximal score.

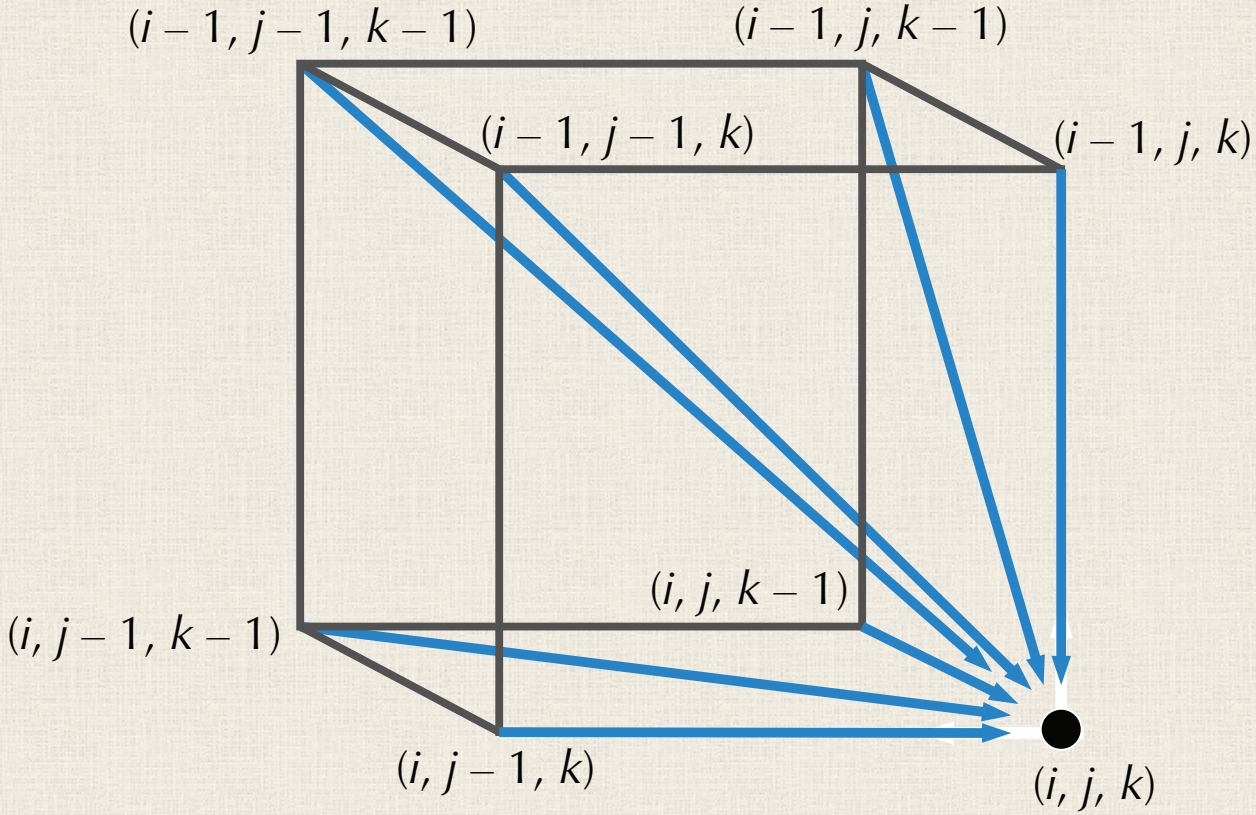
Problem 2: Moving to Multiple Sequences

Multiple Alignment Problem:

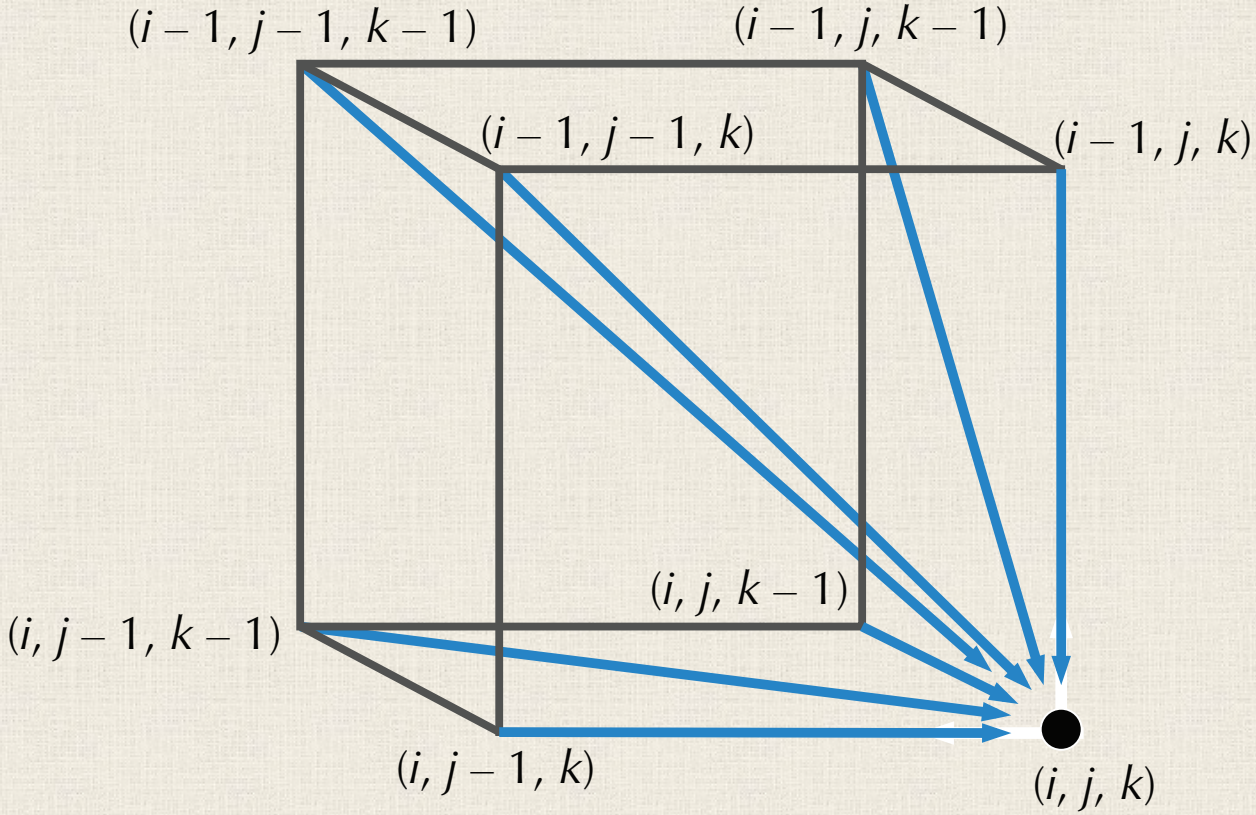
- **Input:** A collection of *t* strings.
- **Output:** A multiple alignment of these strings having maximal score.

STOP: What algorithm would you propose to solve this problem?

Multiple Strings, Multiple *Dimensions*

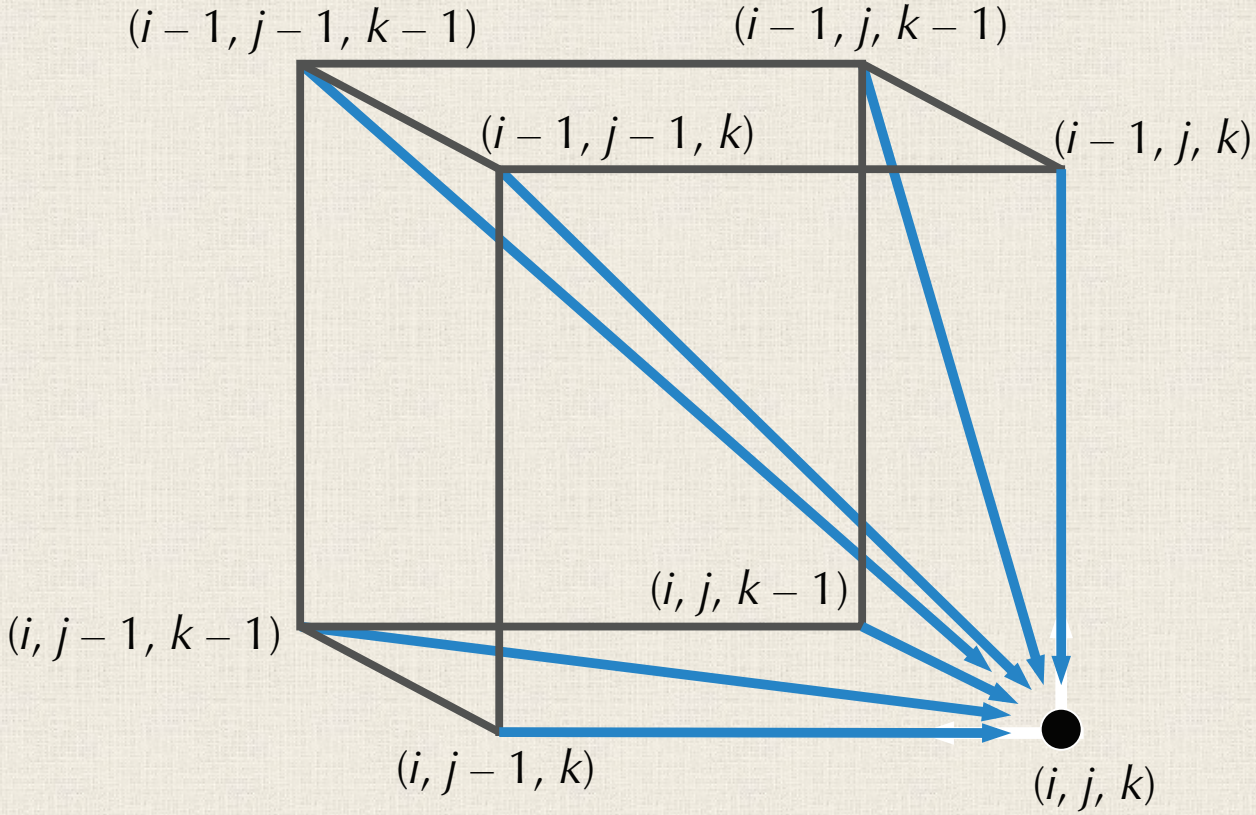


Multiple Strings, Multiple *Dimensions*



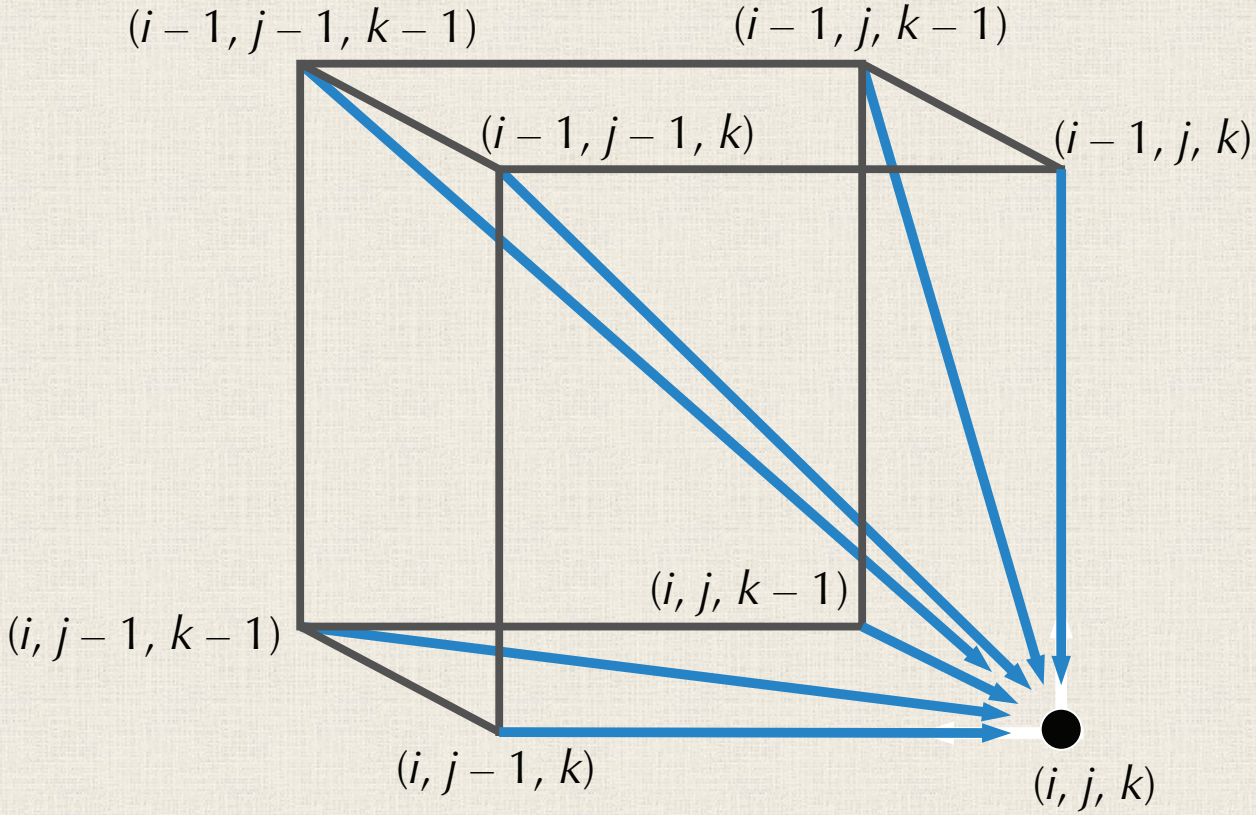
STOP: What is the issue with the dynamic programming approach in multiple dimensions?

Multiple Strings, Multiple *Dimensions*



Answer: The number of edges in a single block grows like $2^t - 1$...

Multiple Strings, Multiple *Dimensions*



STOP: What heuristic might you propose to align multiple sequences?

Greedy Heuristic for Multiple Alignment

1. Find an optimal pairwise alignment of each pair of strings.
2. Combine the set of optimal pairwise alignments into a multiple alignment.

Greedy Heuristic for Multiple Alignment

1. Find an optimal pairwise alignment of each pair of strings.
2. Combine the set of optimal pairwise alignments into a multiple alignment.

STOP: Try this approach on the strings `CCCCTTTT`, `TTTTGGGG`, and `GGGGCCCC`.

Greedy Heuristic for Multiple Alignment

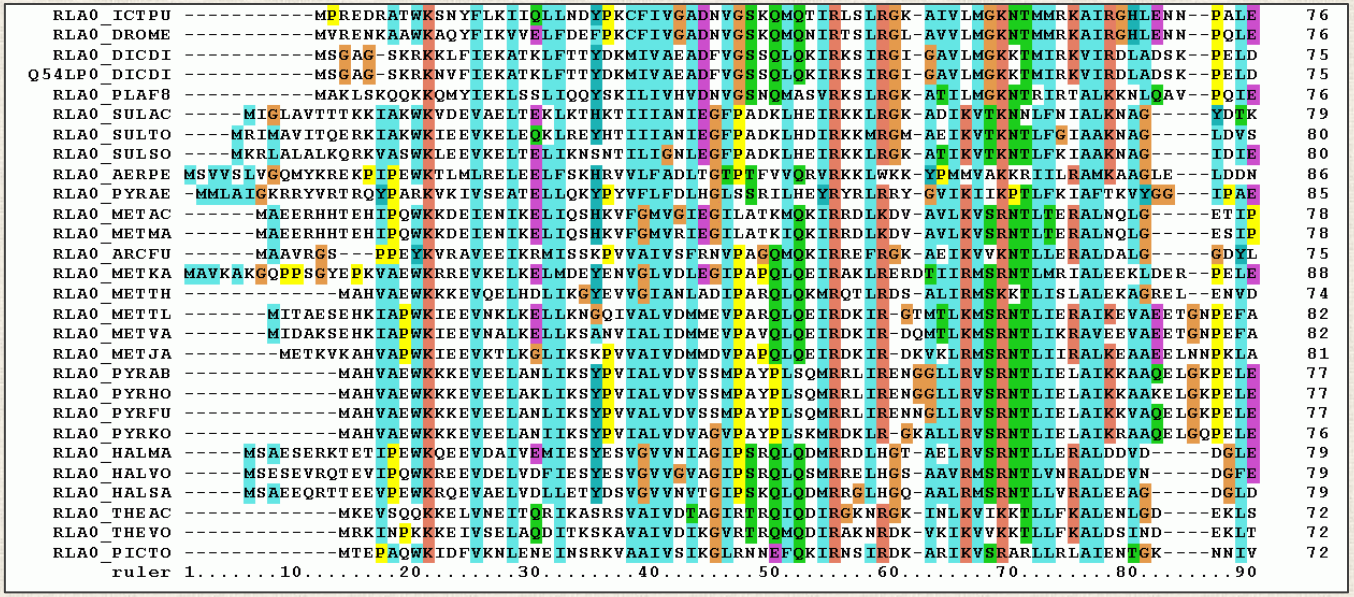
1. Find an optimal pairwise alignment of each pair of strings.
2. Combine the set of optimal pairwise alignments into a multiple alignment.

There is no way to combine these optimal pairwise alignment into a meaningful multiple alignment!

```
CCCCTTTT----        ----CCCCTTTT        TTTTGGGG----
----TTTTGGGG        GGGGCCCC----        ----GGGGCCCC
```

Pairwise Alignment Whispers, Multiple Alignment Shouts

Fortunately, strings that we are aligning will often be so similar that even simple heuristics will find correct alignments. *Stay tuned later in the class!* ☺



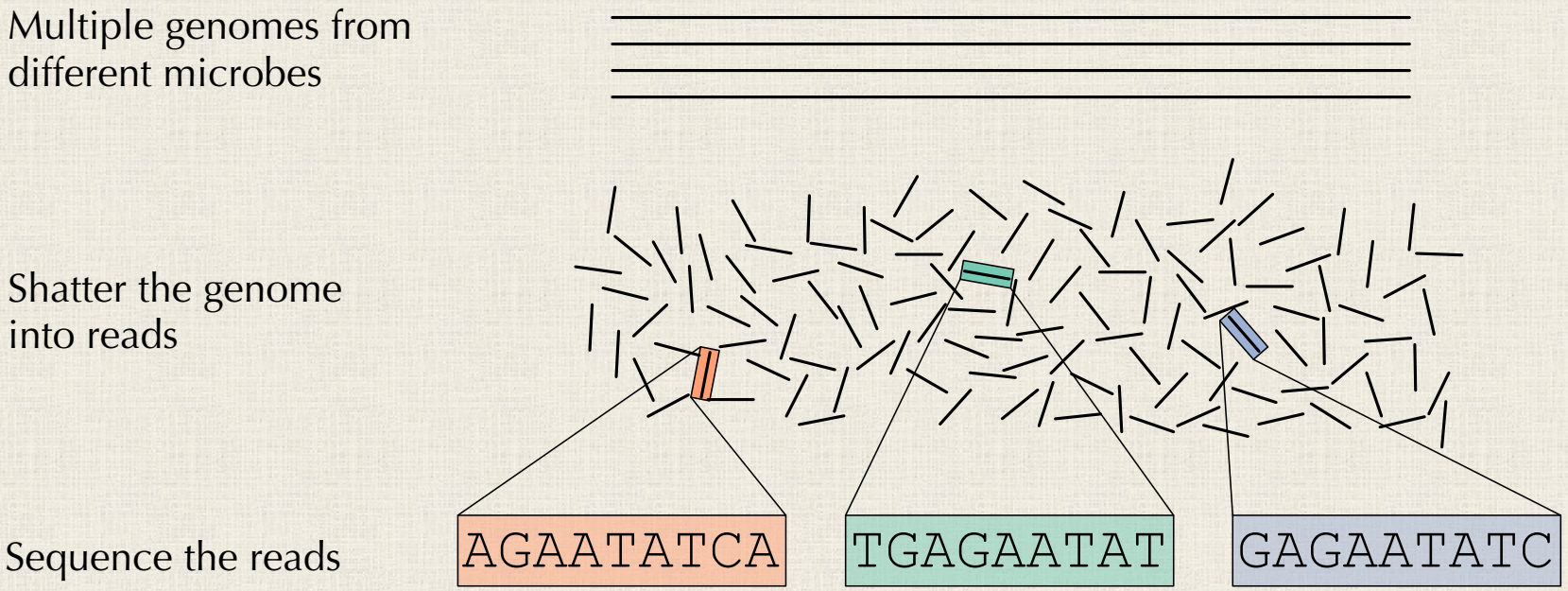
BLAST: ALIGNMENT OF STRINGS TO A DATABASE

Recall that we want to find *all* similarities of a query against a database

Database Comparison Problem:

- **Input:** A string *query* and a (much longer) string *database*.
- **Output:** **One or more** "high-scoring" similarities between *query* (or a substring of *query*) and some substring of *database*.

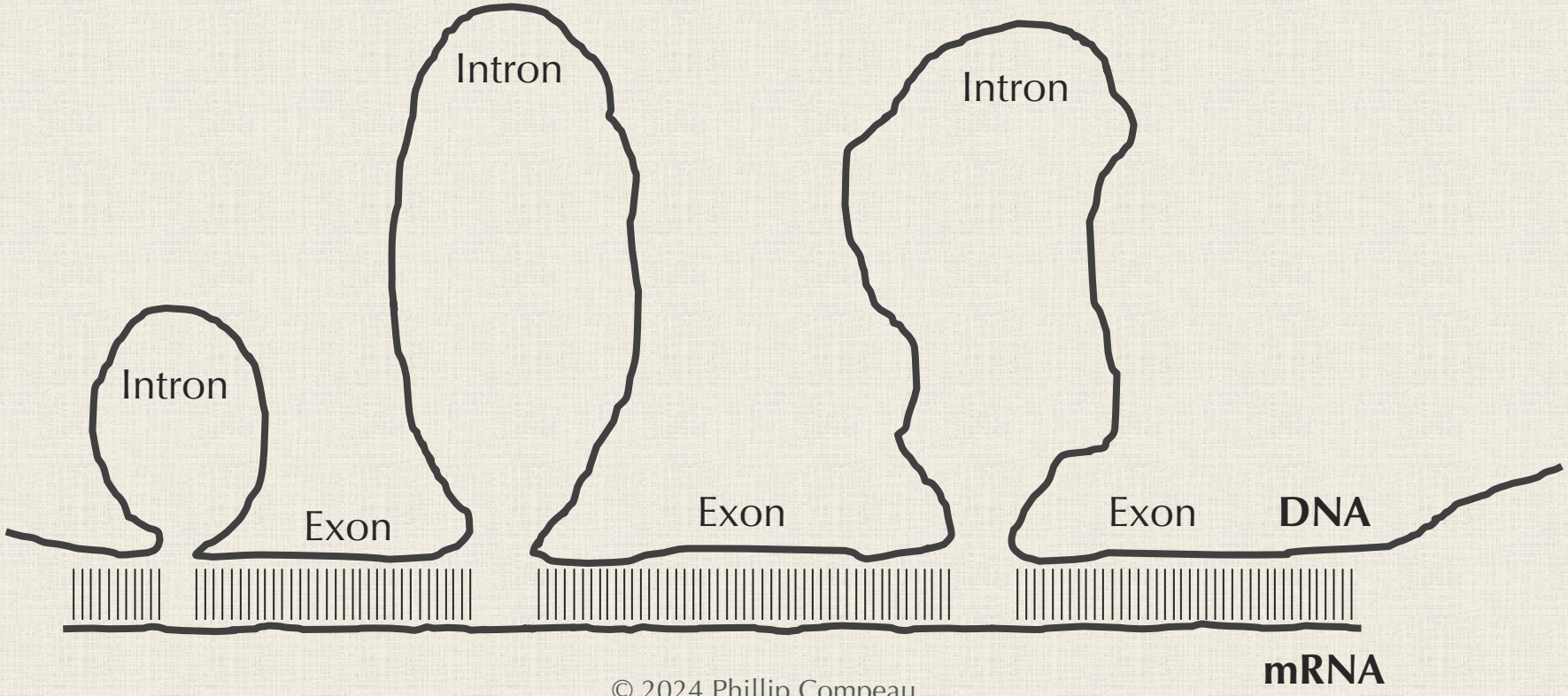
Binning Sequencing Reads



Read binning: compare metagenomics reads to database of known these genomes to find which ones (if any) each read may have come from.

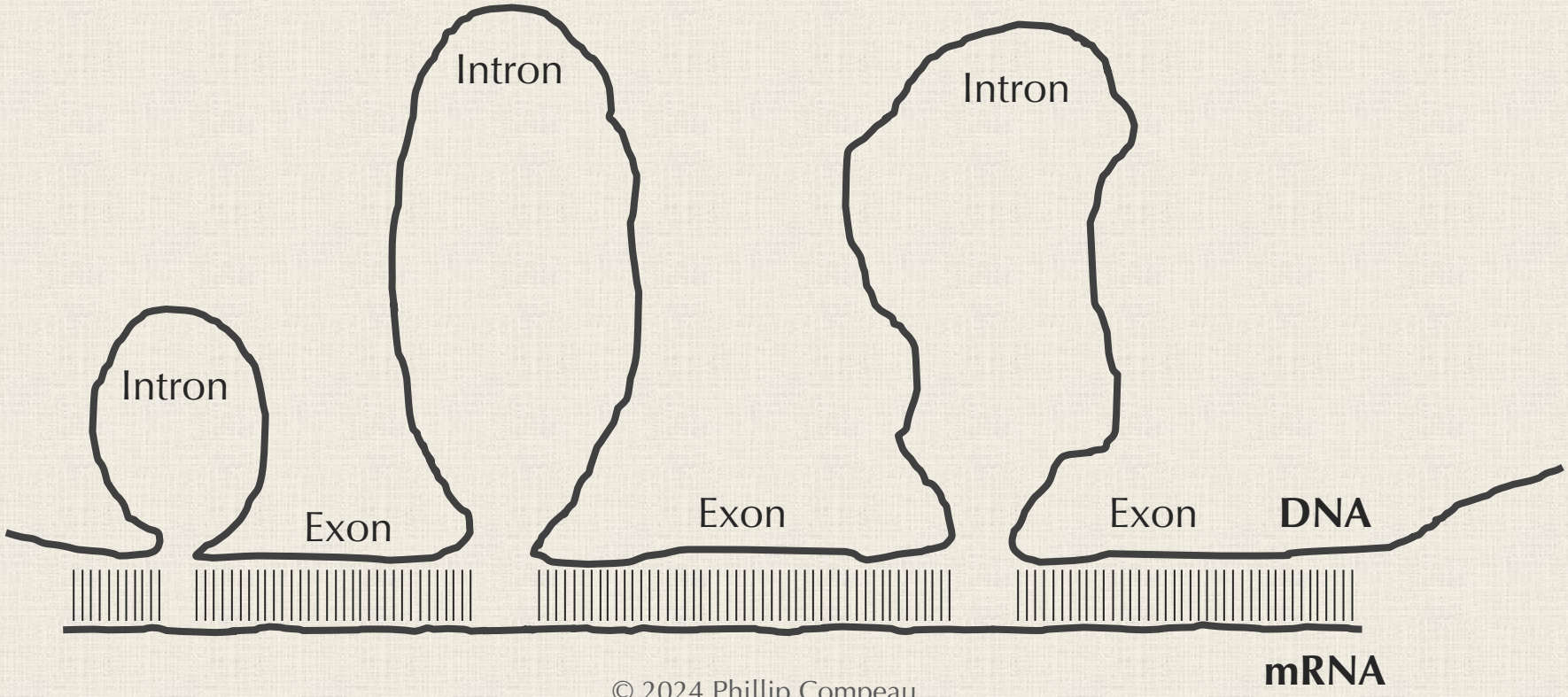
Prokaryotes (Bacteria and Archaea) Don't Have Introns

Intron: Region of a gene that does not get translated into RNA (and therefore protein).

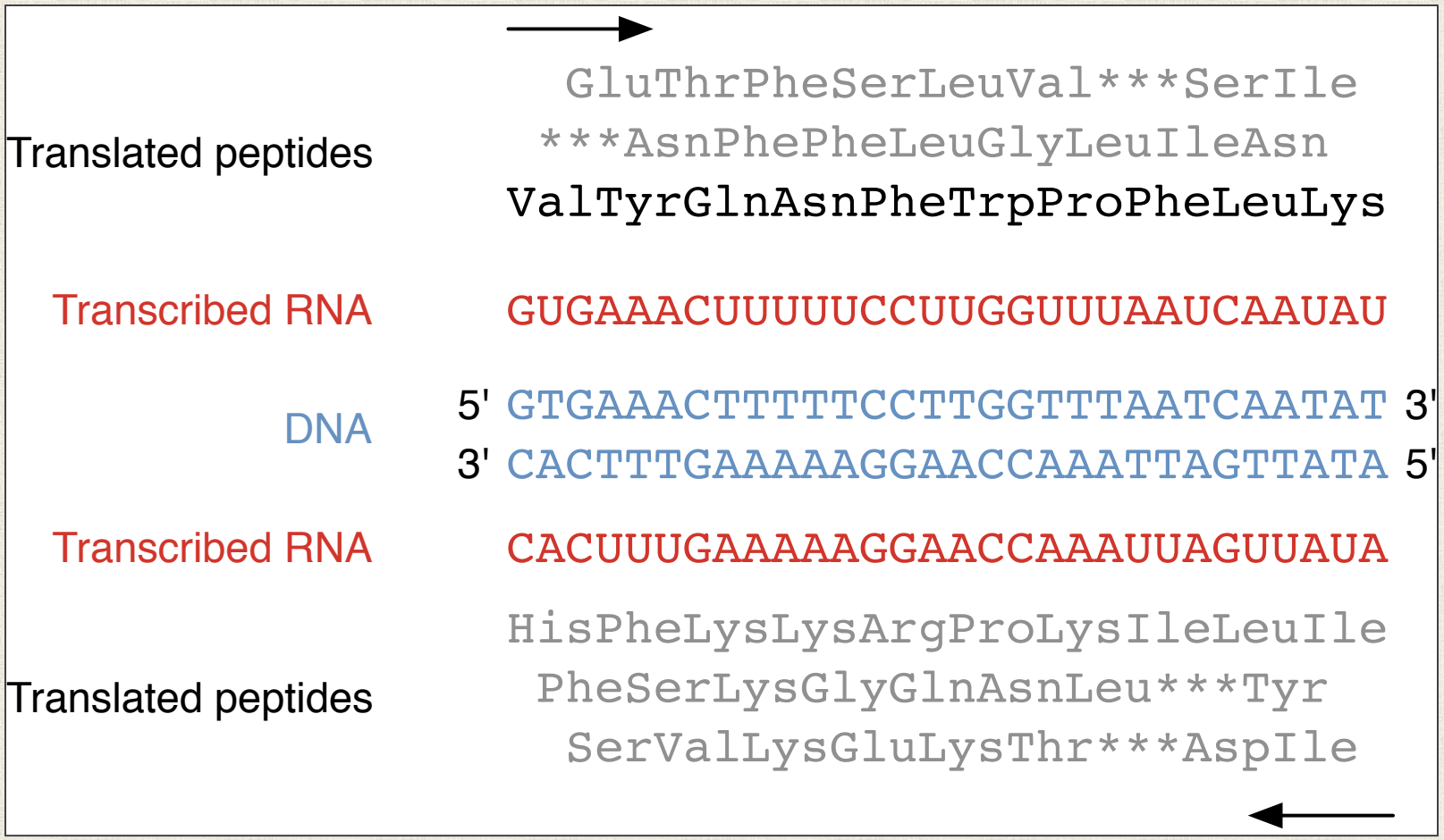


Prokaryotes (Bacteria and Archaea) Don't Have Introns

Good news: prokaryotes don't have introns ☺



DNA Has Six "Reading Frames" for Translation into Protein



Binning Reads is a Protein Comparison Problem

Moreover, most of the prokaryotic genome is made up of genes.

Binning Reads is a Protein Comparison Problem

Moreover, most of the prokaryotic genome is made up of genes.

Key point: We could perform a local alignment of each protein product of each read against the entire database, but just *building* the array would require |*Database*|*|*Patterns*| space.

Recall Our Alignment of Hemoglobin Subunit Alpha in Humans and Zebrafish

```
HBA_HUMAN          1 MVLSPADKTNVKAAWGKVGAHAGEYGAEALERMFLSFPTTKTYFPHF-DL     49
                     |.||..||..|||.|.|:...|.|.|||||.||...:|.|||||.|: ||
HBA_DANRE          1 MSLSDTDKAVVKAIWAKISPKADEIGAEALARMLTVYPQTKTYFSHWADL     50

HBA_HUMAN         50 SHGSAQVKGHGKKVADALTNAVAHVDDMPNALSALSDLHAHKLRVDPVNF     99
                     |.||..||.|||.:..|:..|::.:||:...|:|||:|||.||||||.||
HBA_DANRE         51 SPGSGPVKKHGKTIMGAVGEAISKIDDLVGGLAALSELHAFKLRVDPANF    100

HBA_HUMAN        100 KLLSHCLLVTLAAHLPAEFTPAVHASLDKFLASVSTVLTSKYR    142
                     |:|||.::|.:|...||:|||.||.|:|||..::::..|:.|||
HBA_DANRE        101 KILSHNVIVVIAMLFPADFTPEVHVSVDKFFNNLALALSEKYR    143
```

The protein is the same, and yet the amino acid strings have more mismatches than matches!

Recall Our Alignment of Hemoglobin Subunit Alpha in Humans and Zebrafish

```
HBA_HUMAN          1 MVLSPADKTNVKAAWGKVGAHAGEYGAEALERMFLSFPTTKTYFPHF-DL     49
                     |.||..||..|||.|.|:...|.|.|||||.||...:|.|||||.|: ||
HBA_DANRE          1 MSLSDTDKAVVKAIWAKISPKADEIGAEALARMLTVYPQTKTYFSHWADL     50

HBA_HUMAN         50 SHGSAQVKGHGKKVADALTNAVAHVDDMPNALSALSDLHAHKLRVDPVNF     99
                     |.||..||.|||.:..|:..|::.:||:...|:|||:|||.||||||.||
HBA_DANRE         51 SPGSGPVKKHGKTIMGAVGEAISKIDDLVGGLAALSELHAFKLRVDPANF    100

HBA_HUMAN        100 KLLSHCLLVTLAAHLPAEFTPAVHASLDKFLASVSTVLTSKYR    142
                     |:|||.::|.:|...||:|||.||.|:|||..::::..|:.|||
HBA_DANRE        101 KILSHNVIVVIAMLFPADFTPEVHVSVDKFFNNLALALSEKYR    143
```

The protein is the same, and yet the amino acid strings have more mismatches than matches!

Key Point: Finding database hits isn't just a matter of finding substring matches.

We Need a "Just Right" Binning Approach

Accuracy: Able to take protein-level comparisons into account and find "correct" alignments.

Speed: But still fast enough to be practical.

How Many Citations You Got?

How Many Citations You Got?

Basic local alignment search tool. - NCBI - NIH

https://www.ncbi.nlm.nih.gov/pubmed/2231712 ▾

by SF Altschul - 1990 - Cited by 78111 - Related articles

J Mol Biol. 1990 Oct 5;215(3):403-10. **Basic local alignment search tool**. Altschul SF(1), Gish W, Miller W, Myers EW, Lipman DJ. Author information: (1)National ...

Key Point: BLAST is a ***heuristic***, meaning that it does not promise that it will bin all reads correctly; in other words, it is not solving a computational problem exactly.

BLAST in a Nutshell

- **Input:** a *database* (in this case a protein) and a *query* (in this case a protein corresponding to one of six reading frame translations of sequencing read). Plus, some parameters (later).
- **Output:** A collection of high-scoring local alignments of the query against the database (there may be none, or more than are found).

Step 1: Divide A Given Read into *k*-mers

Note: *k* is one of the additional parameters that we mentioned.

```
CFCDIQL
CFC
 FCD
  CDI
   DIQ
    IQL
```

Step 1: Divide A Given Read into *k*-mers

Note: *k* is one of the additional parameters that we mentioned.

The number of alignments that BLAST produces for a given collection of parameters is called its **sensitivity**.

```
CFCDIQL
CFC
 FCD
  CDI
   DIQ
    IQL
```

Step 1: Divide A Given Read into *k*-mers

Note: *k* is one of the additional parameters that we mentioned.

The number of alignments that BLAST produces for a given collection of parameters is called its **sensitivity**.

```
CFCDIQL
CFC
 FCD
  CDI
   DIQ
    IQL
```

STOP: What do you think happens to the sensitivity of BLAST as *k* increases/decreases?

Step 1: Divide A Given Read into *k*-mers

Note: *k* is one of the additional parameters that we mentioned.

```
CFCDIQL
CFC
 FCD
  CDI
   DIQ
    IQL
```

The number of alignments that BLAST produces for a given collection of parameters is called its **sensitivity**.

Answer: Sensitivity and *k* have an inverse relationship: as *k* increases, sensitivity decreases.

Step 2: For Each *k*-mer *x*, what other *k*-mers score well against it?

	C	S	T	P	A	G	N	D	E	Q	H	R	K	M	I	L	V	F	Y	W	B	Z
C	12																					
S	0	2																				
T	-2	1	3																			
P	-3	1	0	6																		
A	-2	1	1	1	2																	
G	-3	1	0	-1	1	5																
N	-4	1	0	-1	0	0	2															
D	-5	0	0	-1	1	2	2	4														
E	-5	0	0	-1	0	0	1	3	4													
Q	-5	-1	-1	0	0	-1	1	2	2	4												
H	-3	-1	-1	0	-1	-2	2	1	1	3	6											
R	-4	0	-1	0	-2	-3	0	-1	-1	1	2	6										
K	-5	0	0	-1	-1	-2	1	0	0	1	0	3	5									
M	-5	-2	-1	-2	-1	-3	-2	-3	-2	-1	-2	0	0	6								
I	-2	-1	0	-2	-1	-3	-2	-2	-2	-2	-2	-2	-2	2	5							
L	-6	-3	-2	-3	-2	-4	-3	-4	-3	-2	-2	-3	-3	4	2	6						
V	-2	-1	0	-1	0	-1	-2	-2	-2	-2	-2	-2	-2	2	4	2	4					
F	-4	-3	-3	-5	-4	-5	-4	-6	-5	-5	-2	-4	-5	0	1	2	-1	9				
Y	0	-3	-3	-5	-3	-5	-2	-4	-4	-4	0	-4	-4	-2	-1	-1	-2	7	10			
W	-8	-2	-5	-6	-6	-7	-4	-7	-7	-5	-3	2	-3	-4	-5	-2	-6	0	0	17		
B	-4	0	0	-1	0	0	2	3	2	1	1	-1	1	-2	-2	-3	-2	-5	-3	-5	2	
Z	-5	0	-1	0	0	-1	1	3	3	3	2	0	0	-2	-2	-3	-2	-5	-4	-6	2	3

PAM_{250}

```
CFCDIQL
CFC
 FCD
  CDI
   DIQ
    IQL
```

Step 2: For Each *k*-mer *x*, what other *k*-mers score well against it?

	C	S	T	P	A	G	N	D	E	Q	H	R	K	M	I	L	V	F	Y	W	B	Z
C	12																					
S	0	2																				
T	-2	1	3																			
P	-3	1	0	6																		
A	-2	1	1	1	2																	
G	-3	1	0	-1	1	5																
N	-4	1	0	-1	0	0	2															
D	-5	0	0	-1	1	2	2	4														
E	-5	0	0	-1	0	0	1	3	4													
Q	-5	-1	-1	0	0	-1	1	2	2	4												
H	-3	-1	-1	0	-1	-2	2	1	1	3	6											
R	-4	0	-1	0	-2	-3	0	-1	-1	1	2	6										
K	-5	0	0	-1	-1	-2	1	0	0	1	0	3	5									
M	-5	-2	-1	-2	-1	-3	-2	-3	-2	-1	-2	0	0	6								
I	-2	-1	0	-2	-1	-3	-2	-2	-2	-2	-2	-2	-2	2	5							
L	-6	-3	-2	-3	-2	-4	-3	-4	-3	-2	-2	-3	-3	4	2	6						
V	-2	-1	0	-1	0	-1	-2	-2	-2	-2	-2	-2	-2	2	4	2	4					
F	-4	-3	-3	-5	-4	-5	-4	-6	-5	-5	-2	-4	-5	0	1	2	-1	9				
Y	0	-3	-3	-5	-3	-5	-2	-4	-4	-4	0	-4	-4	-2	-1	-1	-2	7	10			
W	-8	-2	-5	-6	-6	-7	-4	-7	-7	-5	-3	2	-3	-4	-5	-2	-6	0	0	17		
B	-4	0	0	-1	0	0	2	3	2	1	1	-1	1	-2	-2	-3	-2	-5	-3	-5	2	
Z	-5	0	-1	0	0	-1	1	3	3	3	2	0	0	-2	-2	-3	-2	-5	-4	-6	2	3

PAM_{250}

Exercise: Four 3-mers score > 23 against `CFC`. What are they?

```
CFCDIQL
CFC
 FCD
  CDI
   DIQ
    IQL
```

Step 2: For Each *k*-mer *x*, what other *k*-mers score well against it?

High Scoring *k*-mers Problem:

- **Input:** an amino acid *k*-mer *x*, a scoring matrix *Score*, and a threshold value *T*.
- **Output:** All amino acid *k*-mers *y* such that $Score(x_1, y_1) + Score(x_2, y_2) + \ldots + Score(x_k, y_k)$ is $> T$.

Step 2: For Each *k*-mer *x*, what other *k*-mers score well against it?

High Scoring *k*-mers Problem:

- **Input:** an amino acid *k*-mer *x*, a scoring matrix *Score*, and a threshold value *T*.
- **Output:** All amino acid *k*-mers *y* such that $Score(x_1, y_1) + Score(x_2, y_2) + \ldots + Score(x_k, y_k)$ is $> T$.

Note: The *k*-mers produced are the *k*-mers that we should look for in the database that match well against *x* – so we're back to exact pattern matching!

Step 2: For Each *k*-mer *x*, what other *k*-mers score well against it?

High Scoring *k*-mers Problem:

- **Input:** an amino acid *k*-mer *x*, a scoring matrix *Score*, and a threshold value *T*.
- **Output:** All amino acid *k*-mers *y* such that $Score(x_1, y_1) + Score(x_2, y_2) + \ldots + Score(x_k, y_k)$ is $> T$.

STOP: Note that *Score* and *T* add more parameters to BLAST. What effect do you think increasing/decreasing *T* has on sensitivity?

Step 2: For Each *k*-mer *x*, what other *k*-mers score well against it?

High Scoring *k*-mers Problem:

- **Input:** an amino acid *k*-mer *x*, a scoring matrix *Score*, and a threshold value *T*.
- **Output:** All amino acid *k*-mers *y* such that $Score(x_1, y_1) + Score(x_2, y_2) + \ldots + Score(x_k, y_k)$ is $> T$.

Note: There is an efficient algorithm for solving this, but we won't cover it here.

Step 2: For Each *k*-mer *x*, what other *k*-mers score well against it?

High Scoring *k*-mers Problem:

- **Input:** an amino acid *k*-mer *x*, a scoring matrix *Score*, and a threshold value *T*.
- **Output:** All amino acid *k*-mers *y* such that $Score(x_1, y_1) + Score(x_2, y_2) + \ldots + Score(x_k, y_k)$ is $> T$.

Key point: Once we solve this problem, we have a (small) set of *k*-mers to search the genome for.

Step 3: Search for High-Scoring *k*-mers in Database

In summary, each read produces a collection of *k*-mers, and each *k*-mer produces its own high-scoring *k*-mers.

```
CFCDIQL
CFC
 FCD
  CDI
   DIQ
    IQL
```

Step 3: Search for High-Scoring *k*-mers in Database

In summary, each read produces a collection of *k*-mers, and each *k*-mer produces its own high-scoring *k*-mers.

We then "slide" all *k*-mers across the database, looking for exact matches. Any exact matches become our **seeds**.

```
CFCDIQL
CFC
 FCD
  CDI
   DIQ
    IQL
```

Step 3: Search for High-Scoring *k*-mers in Database

In summary, each read produces a collection of *k*-mers, and each *k*-mer produces its own high-scoring *k*-mers.

```
CFCDIQL
CFC
 FCD
  CDI
   DIQ
    IQL
```

We then "slide" all *k*-mers across the database, looking for exact matches. Any exact matches become our **seeds**.

This is a good idea because it means we can organize the *k*-mers (say, with a map) and we only need to make a *single pass* down the database.

Step 4: Initial Extension into Maximal Segment Pairs

Database ...CICDVQ...

Query CDI

```
CFCDIQL
CFC
 FCD
  CDI
   DIQ
    IQL
```

Note: Just because a *k*-mer has a good score doesn't mean that it can't be *extended* into a longer match with the read.

Step 4: Initial Extension into Maximal Segment Pairs

Database ...CICDVQ...

Query CFCDIQ

```
CFCDIQL
CFC
 FCD
  CDI
   DIQ
    IQL
```

Note: Just because a *k*-mer has a good score doesn't mean that it can't be *extended* into a longer match with the read.

Step 4: Initial Extension into Maximal Segment Pairs

Database ...CICDVQ...

Query CFCDIQ

```
CFCDIQL
CFC
 FCD
  CDI
   DIQ
    IQL
```

We extend each seed as far to the left and right as we can until the score w/r/t the database stops increasing. The result is a pair of substrings of the query and database called a **maximal segment pair**.

Step 4: Initial Extension into Maximal Segment Pairs

Database ...CICDVQ...

Query CFCDIQ

```
CFCDIQL
CFC
 FCD
  CDI
   DIQ
    IQL
```

(Think of MSPs as high-scoring local alignments of query against database without gaps.)

Step 5: Trim Maximal Segment Pairs

Yet another threshold parameter S is now used; we throw away any MSPs whose score (with respect to the scoring matrix) is $< S$.

Step 5: Trim Maximal Segment Pairs

Yet another threshold parameter S is now used; we throw away any MSPs whose score (with respect to the scoring matrix) is $< S$.

STOP: Does increasing S increase or decrease the sensitivity of BLAST?

Step 6: How "Good" is Each MSP?

STOP: Say we have one MSP of length 12 with score 56 and another MSP of length 9 with score 45. Which one is better?

Step 6: How "Good" is Each MSP?

STOP: Say we have one MSP of length 12 with score 56 and another MSP of length 9 with score 45. Which one is better?

Answer: The way to answer this question is to not say "which is better" but to say "which would be less likely in a random environment?"

Step 6: How "Good" is Each MSP?

p-value: The probability that an event we observe would have occurred in a random reconstruction of whatever we are working with. (This is a very bad statistical definition.)

Step 6: How "Good" is Each MSP?

p-value: The probability that an event we observe would have occurred in a random reconstruction of whatever we are working with. (This is a very bad statistical definition.)

In this case, the p-value we want to compute is the probability Pr(s >= Q) that we would observe an MSP of score s at least some threshold Q in a *random* database.

Step 6: How "Good" is Each MSP?

p-value: The probability that an event we observe would have occurred in a random reconstruction of whatever we are working with. (This is a very bad statistical definition.)

In this case, the p-value we want to compute is the probability Pr(s >= Q) that we would observe an MSP of score s at least some threshold Q in a *random* database.

(Constructing the database and computing this p-value would take two lectures of statistics.)

Step 6: How "Good" is Each MSP?

p-value: The probability that an event we observe would have occurred in a random reconstruction of whatever we are working with. (This is a very bad statistical definition.)

STOP: Are we hoping for MSPs with lower p-values or higher p-values?

Step 7: Combine Nearby MSPs

Database ...XXAAAAAAAAXXXCCCCCCCXX...

Query ZZAAAAAAAAZZZCCCCCCCZZ

MSP1 MSP2

Step 7: Combine Nearby MSPs

Database ...XXAAAAAAAAXXXCCCCCCCXX...

Query ZZAAAAAAAAZZZCCCCCCCZZ

Combined MSP

We will merge two nearby MSPs into one MSP if their combined p-value is still significant (i.e., below yet another threshold parameter).

Step 8: Align Step in BLAST

We *still* haven't added any gaps to our alignments, but we have trimmed away everything but MSP regions that we know are very interesting.

Step 8: Align Step in BLAST

We *still* haven't added any gaps to our alignments, but we have trimmed away everything but MSP regions that we know are very interesting.

We now can perform a (local) alignment of every MSP we found and report the resulting alignment (with p-value).

Sample BLAST Output

```
 Score =  224 bits (113),  Expect = 6e-56
 Identities = 161/161 (100%), Gaps = 0/161 (0%)
 Strand=Plus/Plus

Query  213   GACTGTGCAATACTTAGAGAACCTATAGCATCTTCTCATTCCCATGTGGAACAGGATGCC  272
             ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Sbjct  1205  GACTGTGCAATACTTAGAGAACCTATAGCATCTTCTCATTCCCATGTGGAACAGGATGCC  1264

Query  273   CACATACTGTCTAATTAATAAATTTTCCAtttttttCAAACAAGTATGAATCTAGTTGG  332
             ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Sbjct  1265  CACATACTGTCTAATTAATAAATTTTCCATTTTTTTTCAAACAAGTATGAATCTAGTTGG  1324

Query  333   TTGATGCCtttttttCATGACATAATAAAGTATTTTCTTT  373
             |||||||||||||||||||||||||||||||||||||||
Sbjct  1325  TTGATGCCTTTTTTTTCATGACATAATAAAGTATTTTCTTT  1365
```

Sample BLAST Output

```
 Score =  224 bits (113),  Expect = 6e-56
 Identities = 161/161 (100%), Gaps = 0/161 (0%)
 Strand=Plus/Plus

Query  213   GACTGTGCAATACTTAGAGAACCTATAGCATCTTCTCATTCCCATGTGGAACAGGATGCC  272
             ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Sbjct  1205  GACTGTGCAATACTTAGAGAACCTATAGCATCTTCTCATTCCCATGTGGAACAGGATGCC  1264

Query  273   CACATACTGTCTAATTAATAAATTTTCCAtttttttCAAACAAGTATGAATCTAGTTGG  332
             ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Sbjct  1265  CACATACTGTCTAATTAATAAATTTTCCATTTTTTTTCAAACAAGTATGAATCTAGTTGG  1324

Query  333   TTGATGCCtttttttCATGACATAATAAAGTATTTTCTTT  373
             |||||||||||||||||||||||||||||||||||||||
Sbjct  1325  TTGATGCCTTTTTTTTCATGACATAATAAAGTATTTTCTTT  1365
```

In practice, BLAST returns an **E-value:** the expected number of hits of comparable score in a random database.

Sample BLAST Output

```
 Score =  224 bits (113),  Expect = 6e-56
 Identities = 161/161 (100%), Gaps = 0/161 (0%)
 Strand=Plus/Plus

Query  213   GACTGTGCAATACTTAGAGAACCTATAGCATCTTCTCATTCCCATGTGGAACAGGATGCC  272
             ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Sbjct  1205  GACTGTGCAATACTTAGAGAACCTATAGCATCTTCTCATTCCCATGTGGAACAGGATGCC  1264

Query  273   CACATACTGTCTAATTAATAAATTTTCCAtttttttCAAACAAGTATGAATCTAGTTGG  332
             ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Sbjct  1265  CACATACTGTCTAATTAATAAATTTTCCATTTTTTTTCAAACAAGTATGAATCTAGTTGG  1324

Query  333   TTGATGCCtttttttCATGACATAATAAAGTATTTTCTTT  373
             |||||||||||||||||||||||||||||||||||||||
Sbjct  1325  TTGATGCCTTTTTTTTCATGACATAATAAAGTATTTTCTTT  1365
```

Note: Very small p-values are approximately equal to their corresponding E-values.

SOFTWARE DEMO: MAPPING METAGENOMICS READS TO A DATABASE