Space-Efficient Alignment: Hirschberg's Algorithm

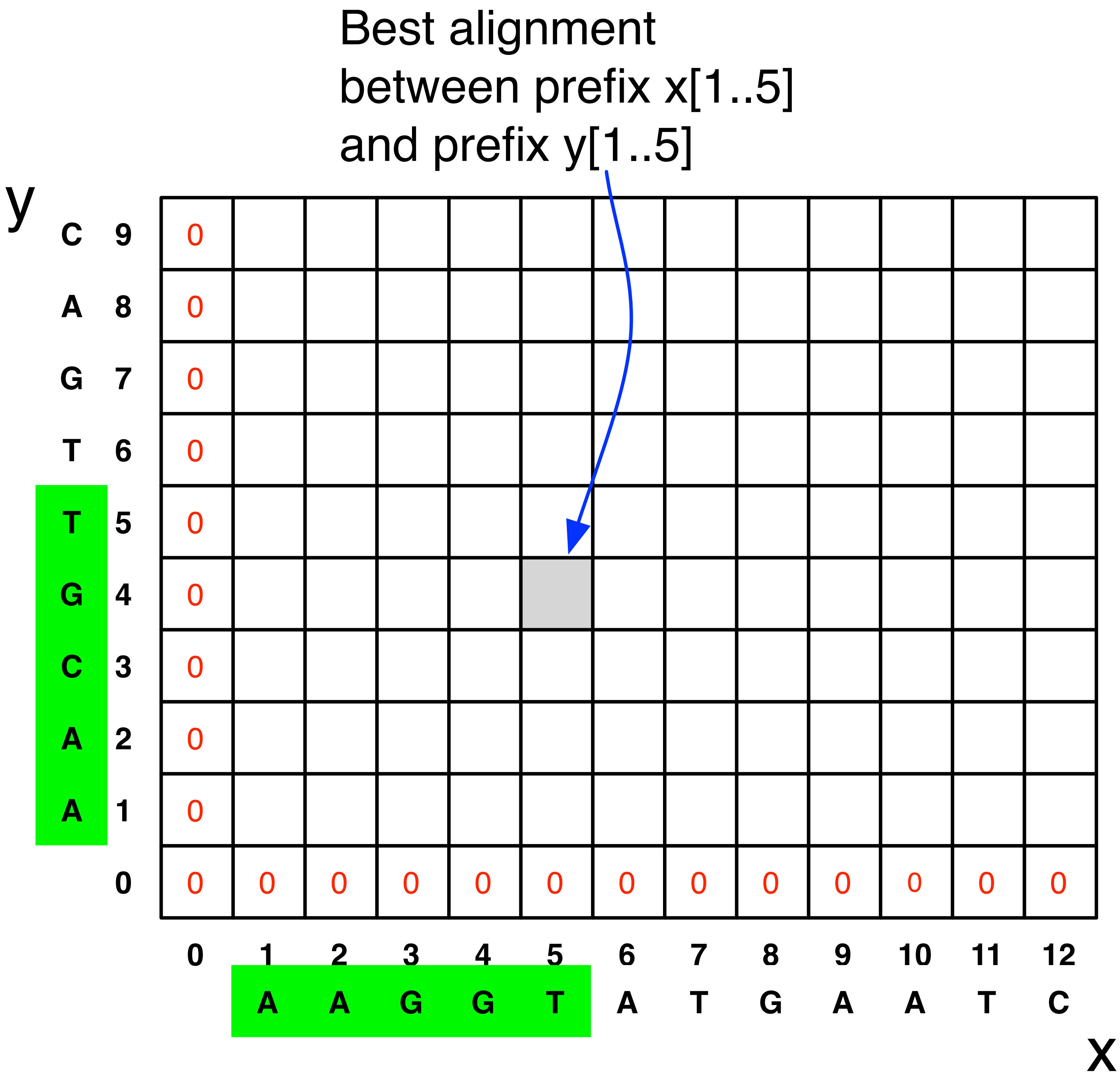
02-713

Slides by Carl Kingsford

Space Usage

- $O(n^2)$ is pretty low space usage, but for a 10 Gb genome, you'd need a huge amount of memory.
- Can we use less space?
    - Hirschberg's algorithm

Remember the meaning of a cell

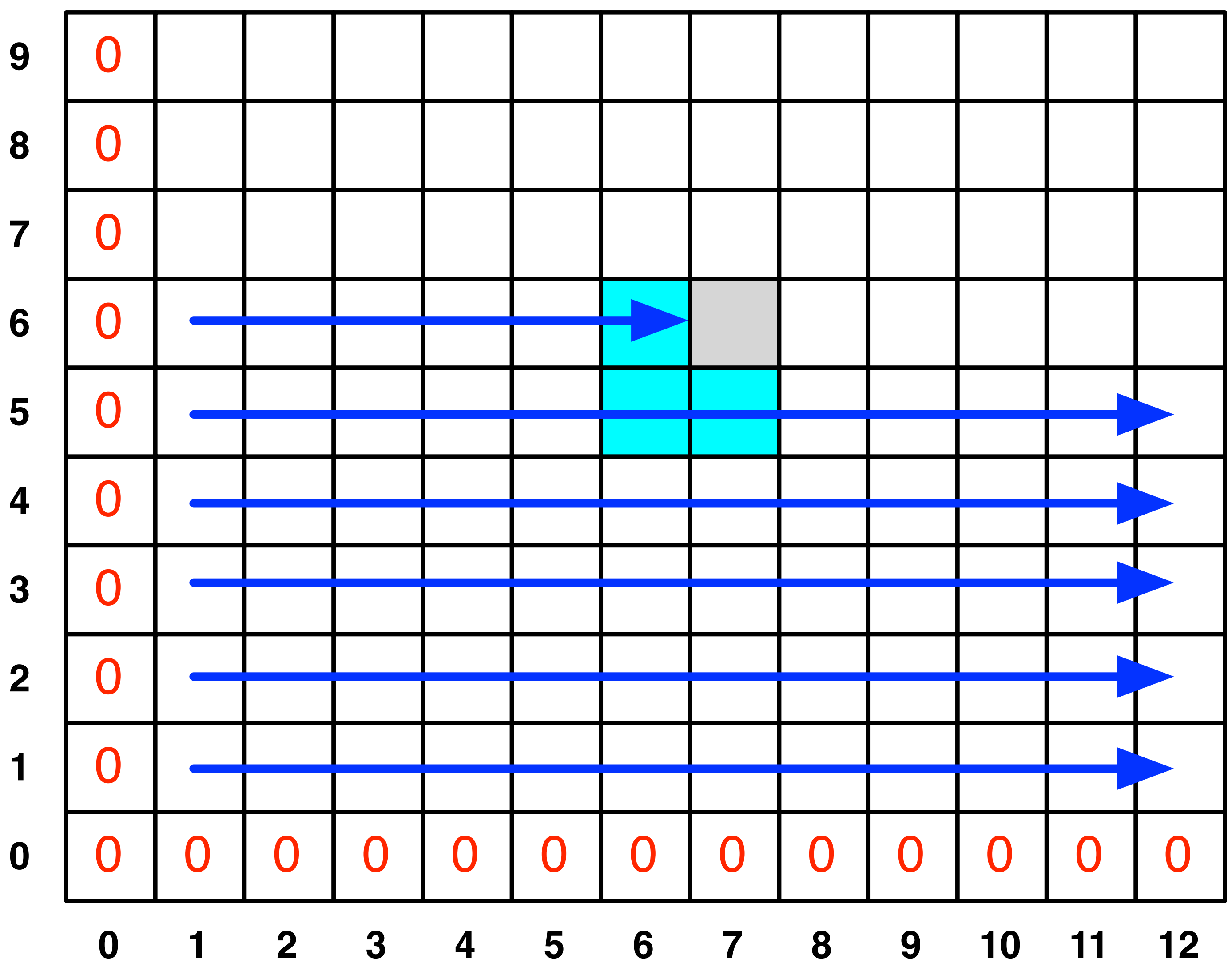


Linear Space for Alignment **Scores**

- If you are only interested in the **cost** or **score** of an alignment, you need to use only O(n) space.
- How?

Linear Space for Alignment **Scores**

- If you are only interested in the **cost** or **score** of an alignment, you need to use only O(n) space.
- How?



When filling in an entry (gray box) we only look at the current and previous rows.

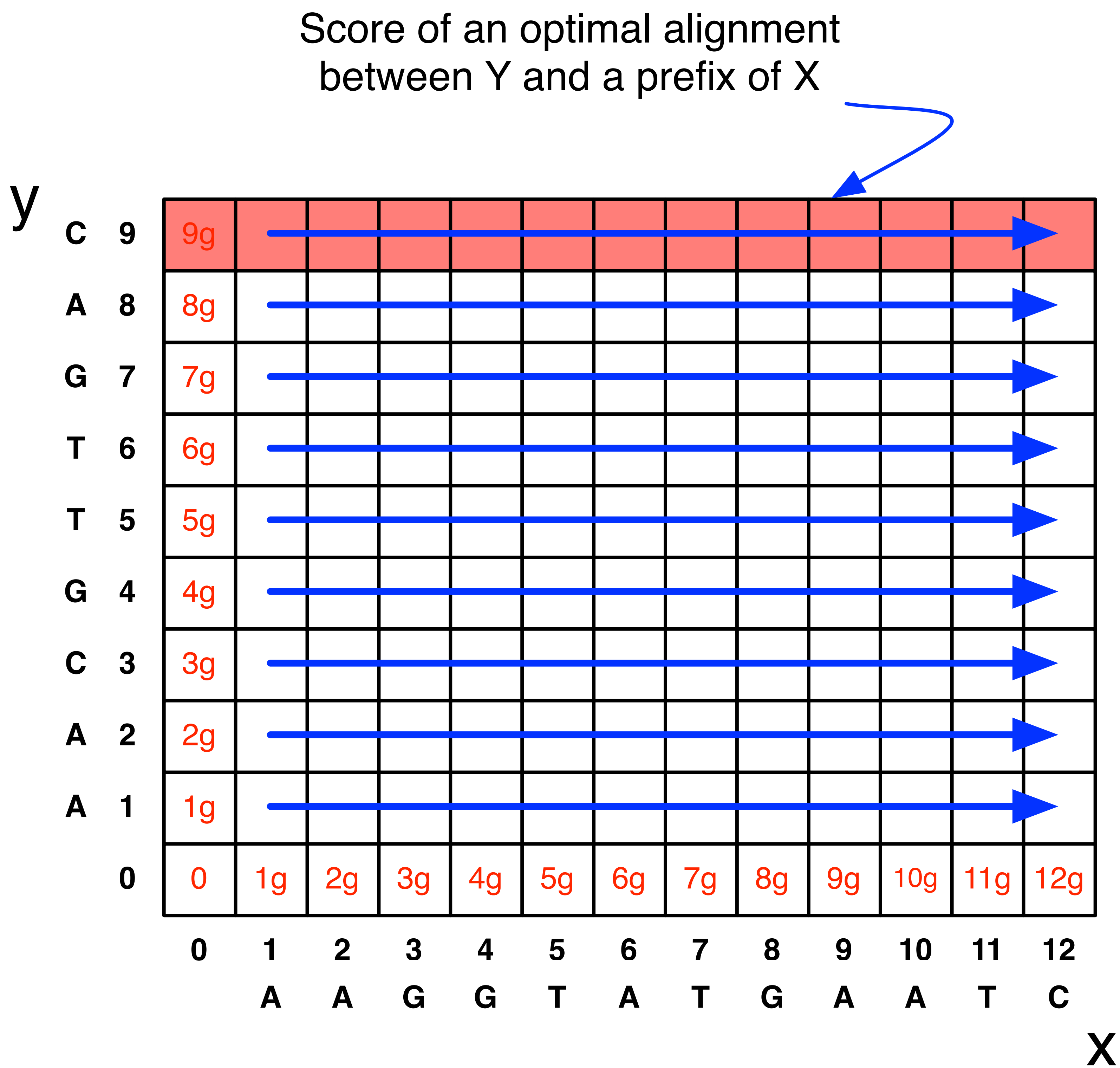
Only need to keep those two rows in memory.

We can do more...

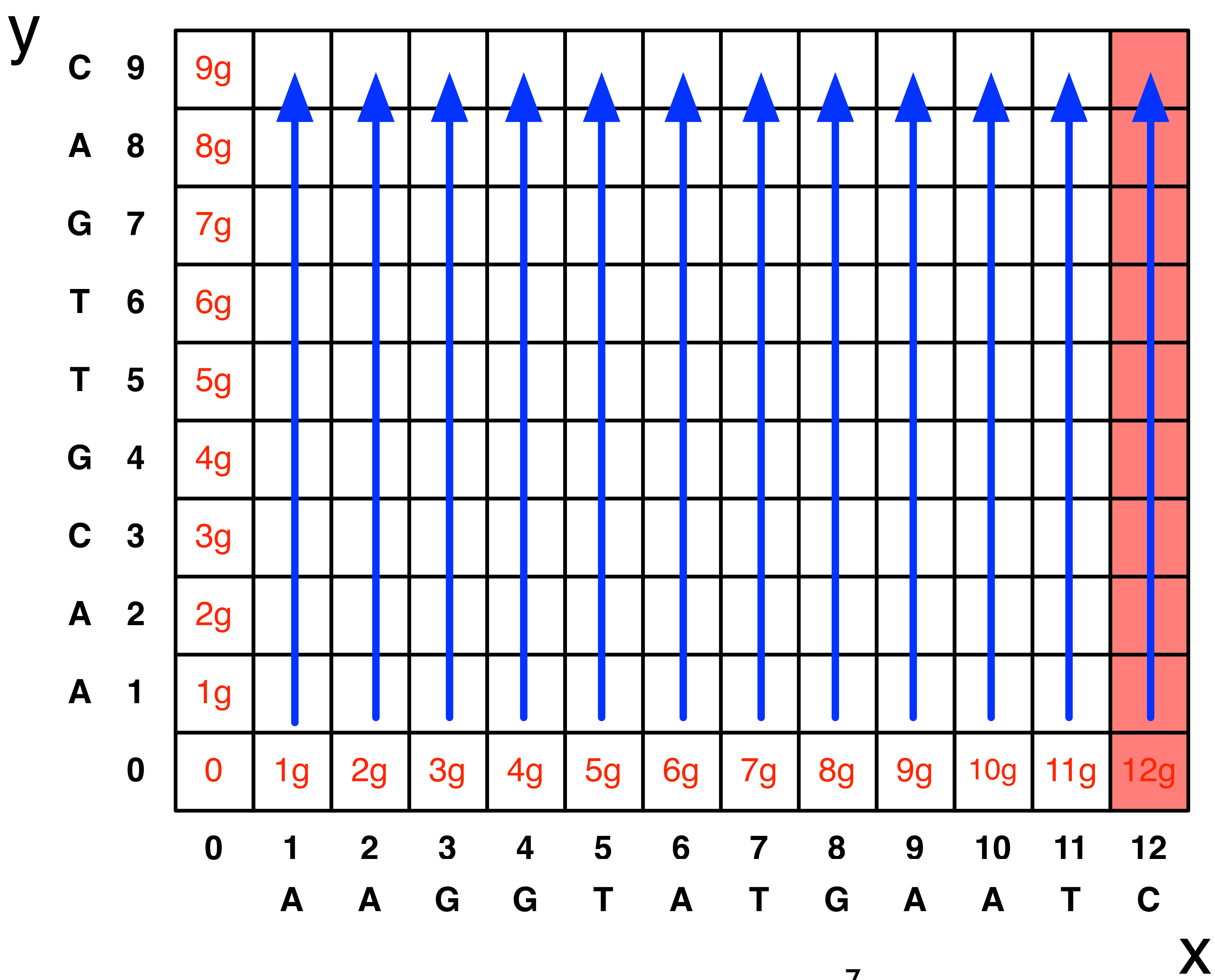
- Given 2 strings X and Y, we can, in linear space and O(*nm*) time, compute the **cost** of aligning...
  - every prefix of X with Y
  - X with every prefix of Y
  - a particular prefix of X with every prefix of Y
  - a particular suffix of X with every suffix of Y

- How can we do that?

Best Alignment Between Prefix of X and Y

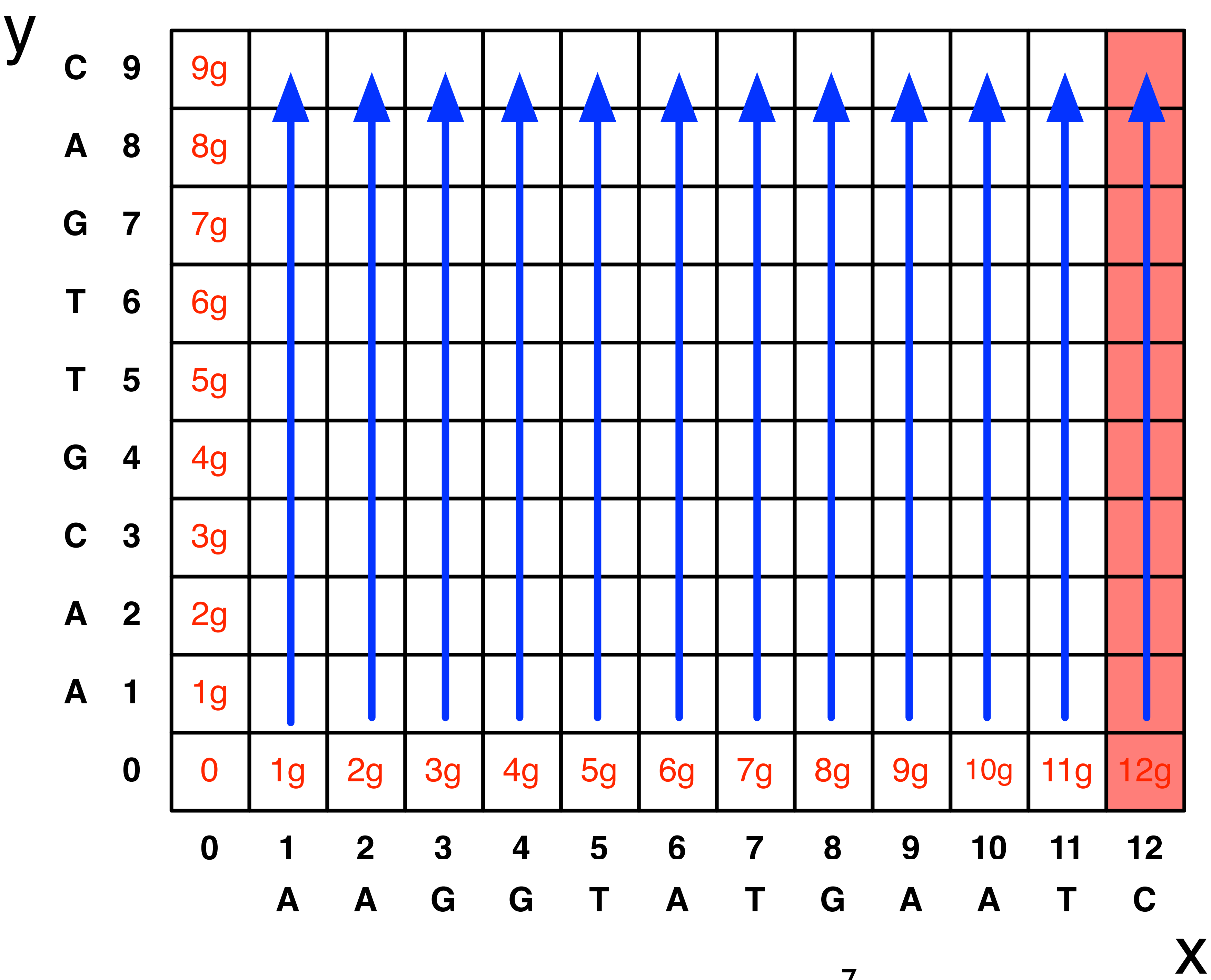


Fill in the matrix by columns...



What is this column?

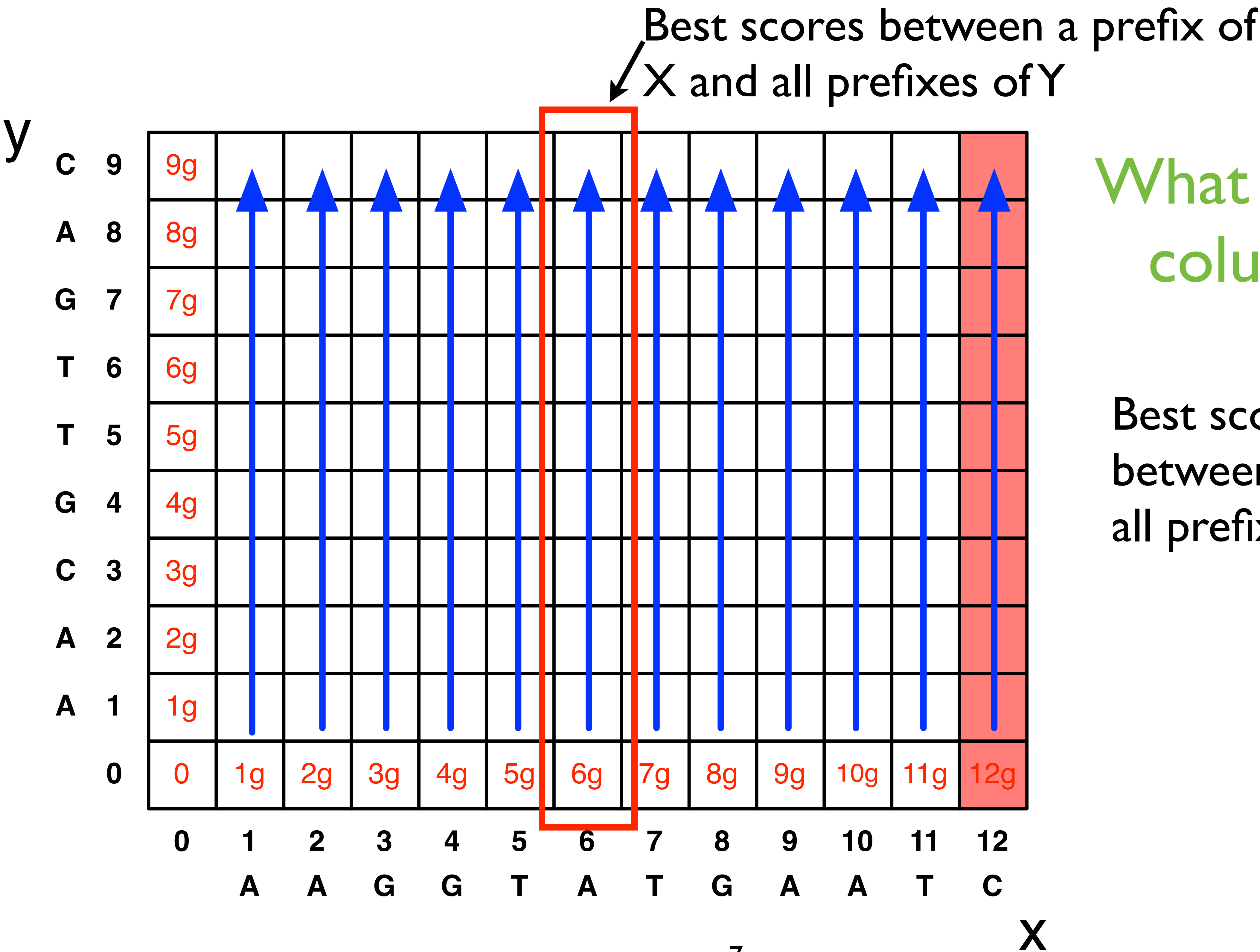
Fill in the matrix by columns...



What is this column?

Best scores between X and all prefixes of Y

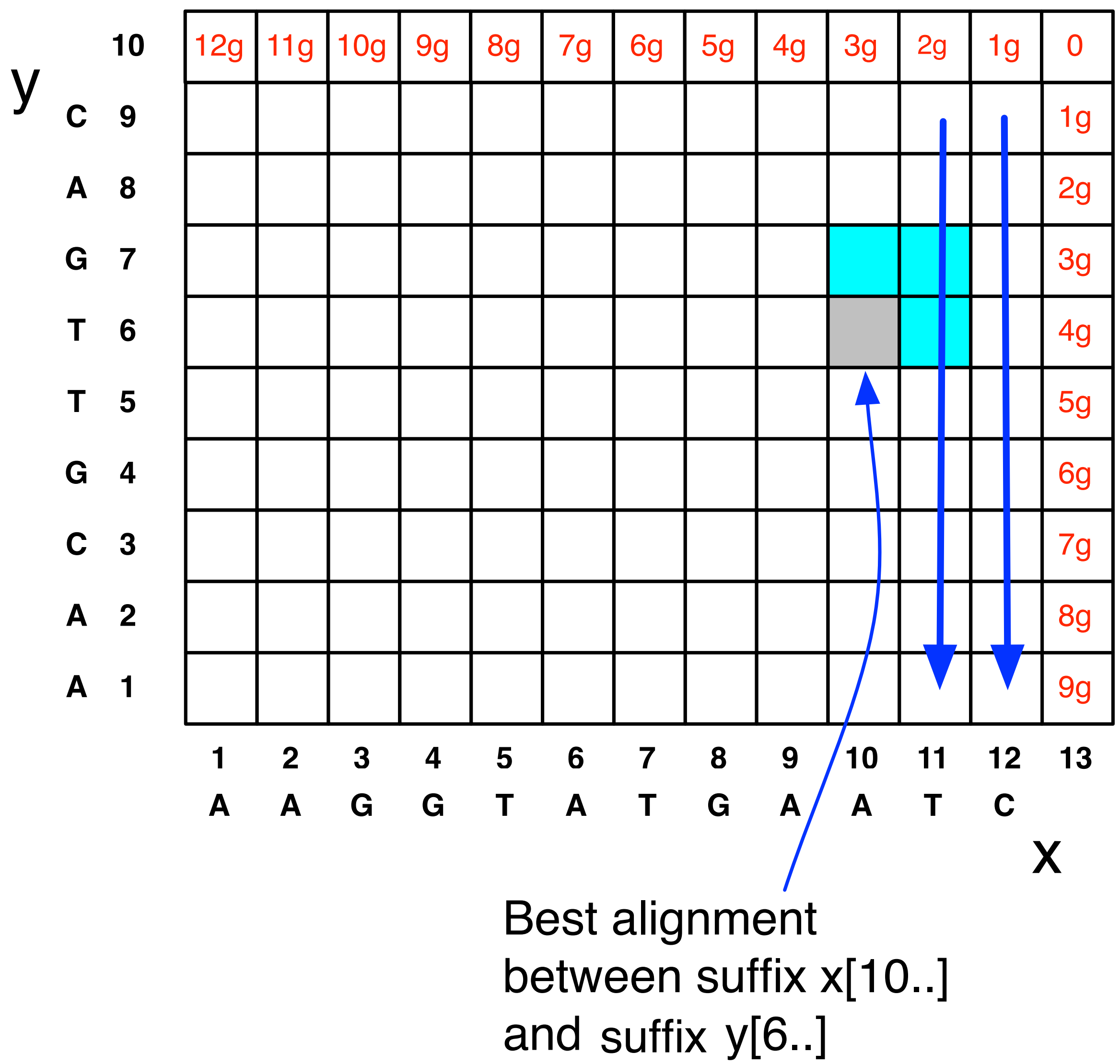
Fill in the matrix by columns...



What is this column?

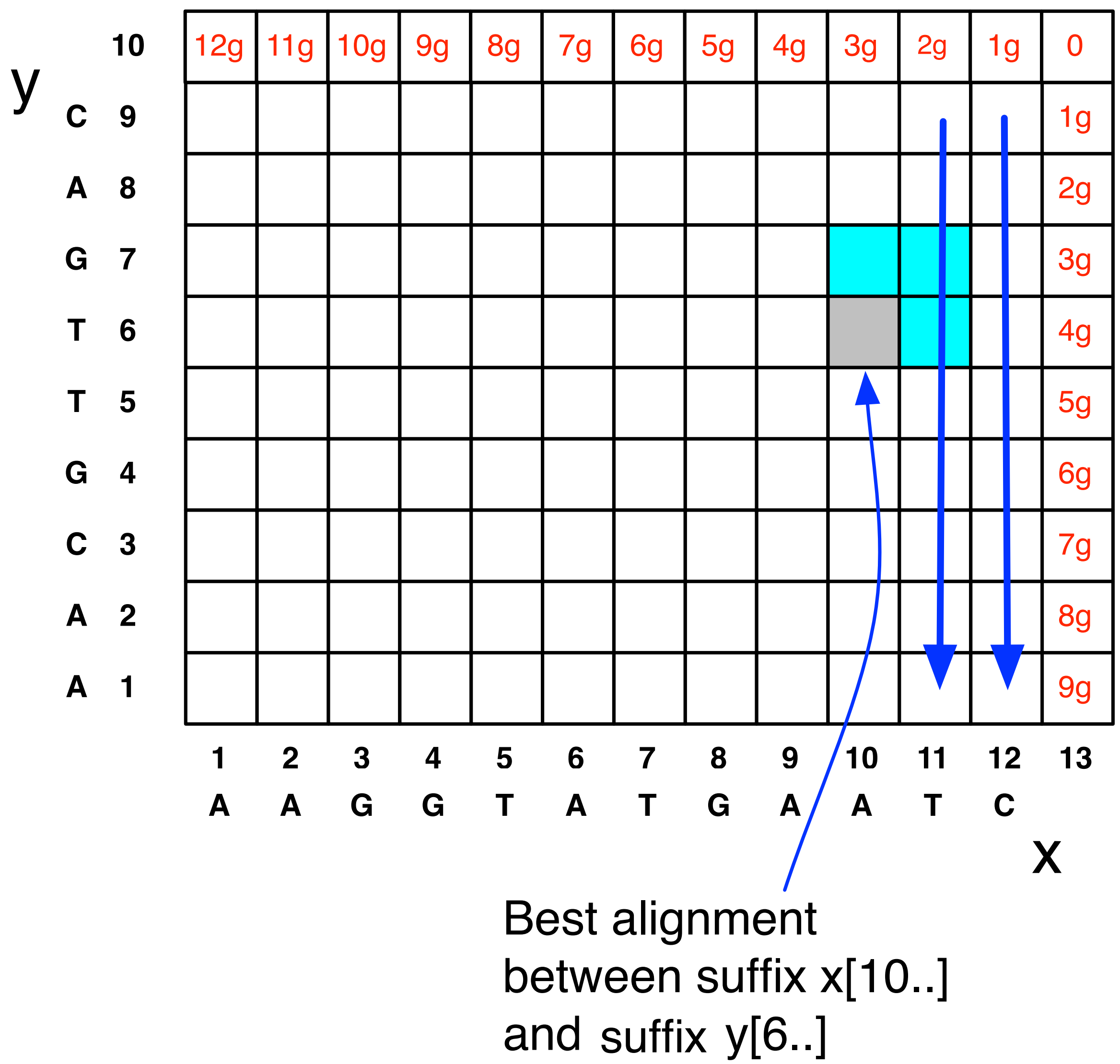
Best scores between X and all prefixes of Y

Cost of Alignment Between X and All Suffixes of Y



$$B[i,j] = \min \begin{cases} \text{cost}(x_i, y_j) + B[i+1, j+1] \\ \text{gap} + B[i, j+1] \\ \text{gap} + B[i+1, j] \end{cases}$$

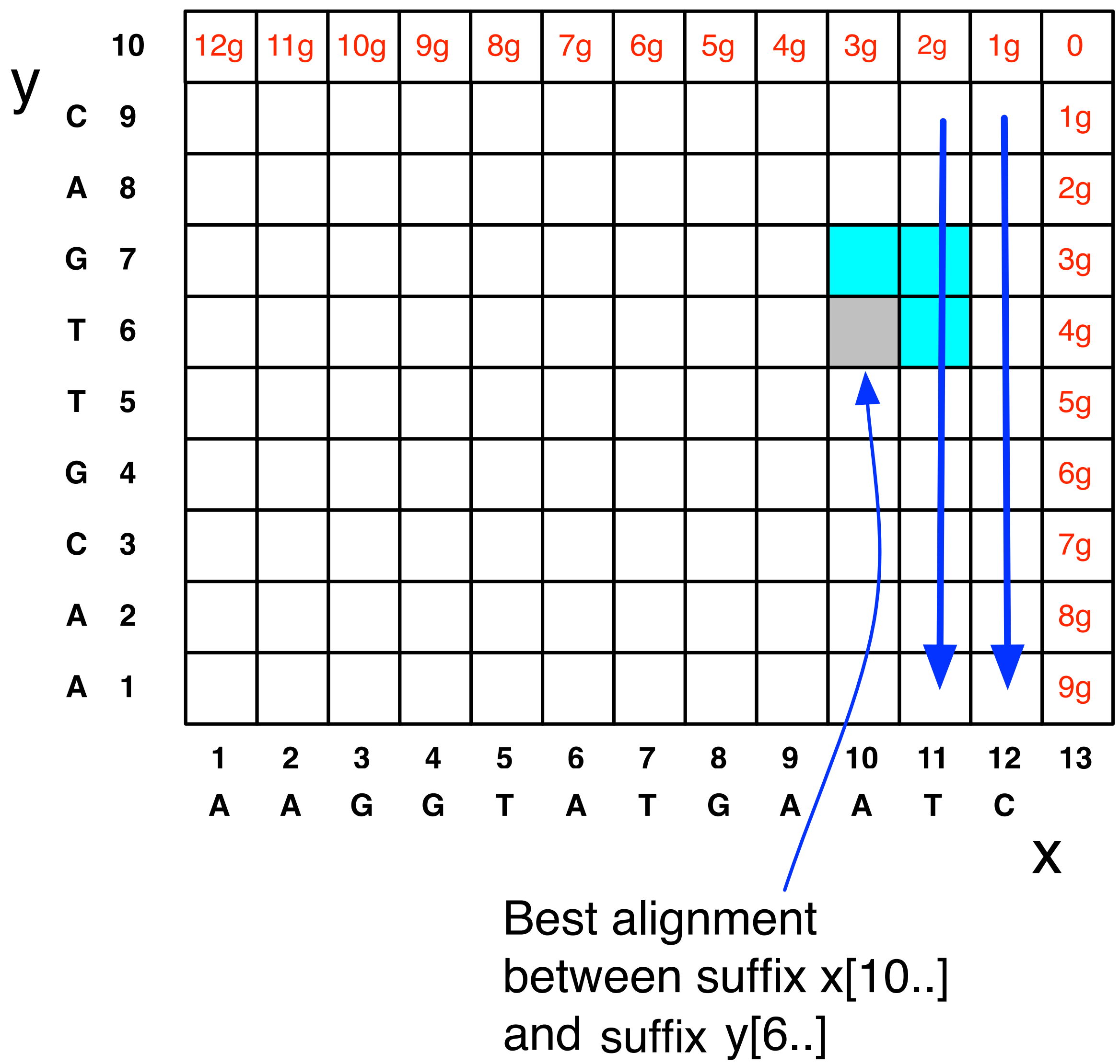
Cost of Alignment Between X and All Suffixes of Y



Exactly the same reasoning as doing the "forward" dynamic programming.

$$B[i,j] = \min \begin{cases} \text{cost}(x_i, y_j) + B[i+1, j+1] \\ \text{gap} + B[i, j+1] \\ \text{gap} + B[i+1, j] \end{cases}$$

Cost of Alignment Between X and All Suffixes of Y



"Backward" dynamic programming.

Exactly the same reasoning as doing the "forward" dynamic programming.

$$B[i,j] = \min \begin{cases} \mathrm{cost}(x_i, y_j) + B[i+1, j+1] \\ \mathrm{gap} + B[i, j+1] \\ \mathrm{gap} + B[i+1, j] \end{cases}$$

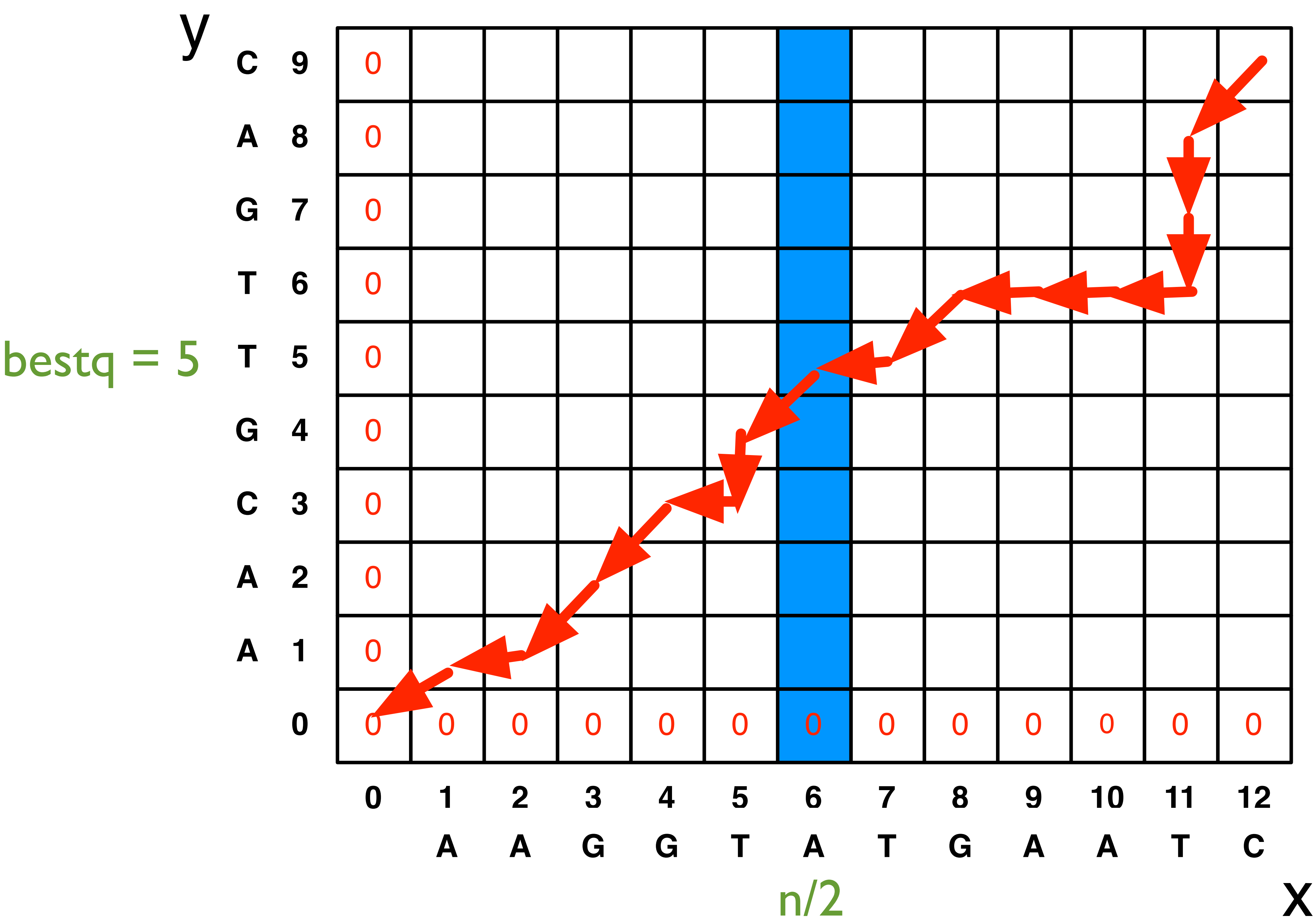
Can We Find the Alignment in O(n) Space?

- Surprisingly, yes, we can output the optimal alignment in linear space.
- This will cost us some extra computation but only a constant factor
- For such a dramatic reduction in space, it's often worth it.
- **Idea**: a divide-and-conquer algorithm to compute half alignments.

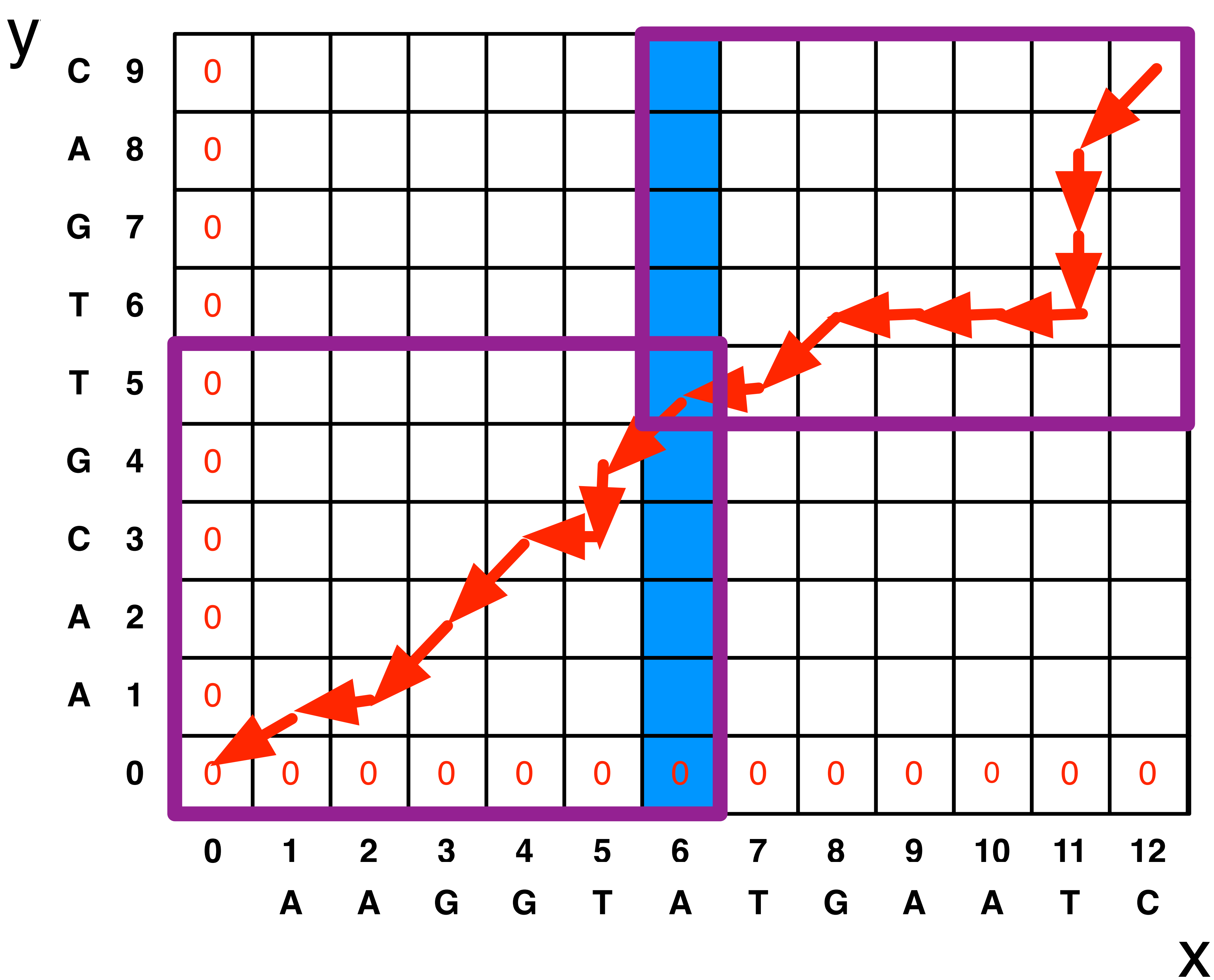
Divide & Conquer

- General algorithmic design technique:
  - Split large problem into a few subproblems.
  - Recursively solve each subproblem.
  - Merge the resulting answers.

The Best Path Uses Some Cell in the Middle Column



The Best Path Uses Some Cell in the Middle Column



Notation

- **AlignValue**(x, y) := compute the *cost* of the best alignment between x and y in O(min |x|, |y|) space.

- Finding the actual alignment is equivalent to finding all the cells that the optimal backtrace passes through.

- Call the optimal backtrace the **ArrowPath**.

First Attempt At Space Efficient Alignment

In the optimal alignment, the first *n*/2 characters of *x* are aligned with the first *q* characters of *y* for some *q*.

```
      12345678
x  =  ACGTACTG
y  =  A-GT-CTG
```

$\underbrace{\texttt{A-GT-}}_{q\,=\,3}$

We don't know *q*, so we have to try *all* possible *q*.

```
ArrowPath := []
def Align(x, y):
  n := |x|; m := |y|
  if n or m ≤ 2: use standard alignment
  for q := 0..m:
    v1 := AlignValue(x[1..n/2], y[1..q])
    v2 := AlignValue(x[n/2+1..n], y[q+1..m])
    if v1 + v2 < best: bestq = q; best = v1 + v2

  Add (n/2, bestq) to ArrowPath
  Align(x[1..n/2], y[1..bestq])
  Align(x[n/2+1..n], y[bestq+1..m])
```

O(n) or O(m) space (standard alignment)

O(n+m) space (v1)

O(n+m) space (v2)

find the q that minimizes the cost of the alignment

Problem

Problem

- This works in linear space.

Problem

- This works in linear space.
- BUT: not in O(*nm*) time. Why?

Problem

- This works in linear space.

- BUT: not in O(*nm*) time. Why?

- It's too expensive to solve all those AlignValue problems in the **for** loop.

Problem

- This works in linear space.
- BUT: not in O(*nm*) time. Why?
- It's too expensive to solve all those AlignValue problems in the **for** loop.
- Define:
  - **AllYPrefixCosts**(x, i, y) = returns an array of the scores of optimal alignments between x[1..i] and all prefixes of Y.
  - **AllYSuffixCosts**(x, i, y) = returns an array of the scores of optimal alignments between x[i..n] and all suffixes of y.
  - These are implemented as described in previous slides by returning the last row or last column of the DP matrix.

Space Efficient Alignment

```
    12345678
x = ACGTACTG
y = A-GT-CTG
    q = 3
```

We still try all possible q, but we use the fact that we can compute the cost between a given prefix and *all* suffixes in linear space.

```
ArrowPath := []

def Align(x, y):
  n := |x|; m := |y|
  if n or m ≤ 2: use standard alignment

  YPrefix := AllYPrefixCosts(x, n/2, y)
  YSuffix := AllYSuffixCosts(x, n/2+1, y)

  for q := 0..m:
    cost = YPrefix[q] + YSuffix[n-q]
    if cost < best: bestq = q; best = cost

  Add (n/2, bestq) to ArrowPath
  Align(x[1..n/2], y[1..bestq])
  Align(x[n/2+1..n], y[bestq+1..m])
```

O(n) or O(m) space (standard alignment)

O(n+m) space (YPrefix and YSuffix)

find the q that minimizes the cost of the alignment, using the costs of aligning X to prefixes and suffixes of Y (for loop)

Running Time Recurrence, I

Full recurrence:

$$T(n,2) \le cn$$
$$T(2,m) \le cm$$
$$T(n,m) \le cmn + \underbrace{T(n/2,q)}_{\texttt{Align(x[1..n/2], y[1..bestq])}} + \overbrace{T(n/2,m-q)}^{\texttt{Align(x[n/2+1..n], y[bestq+1..m])}}$$

Too complicated because we don't know what *q* is.

Simplify: assume both sequences have length *n*, and that we get a perfect split in half every time, *q*=*n*/2:

$$T(n) \le 2T(n/2) + cn^2$$

Solves as:

$$T(n) = O(n^2)$$ ⇒ guess O(nm)

Running Time Recurrence, 2

$$T(n,2) \le cn$$
$$T(2,m) \le cm$$
$$T(n,m) \le cmn + T(n/2,q) + T(n/2,m-q)$$

Guess: T(n,m) ≤ kmn, for some k.

Proof, by induction:

Base cases: If k ≥ c then T(n,2) ≤ cn ≤ c2n ≤ k2n = kmn

Induction step: Assume T(**n', m'**) ≤ k**m'n'** for pairs (n',m') with a product smaller than nm:

$$\begin{aligned} T(n,m) &\le cmn + T(n/2,q) + T(n/2,m-q) \\ &\le cmn + kqn/2 + k(m-q)n/2 \quad \leftarrow \text{apply induction hypothesis} \\ &= cmn + kqn/2 + kmn/2 - kqn/2 \\ &= (c+k/2)mn \end{aligned}$$

$$k = 2c \implies T(n,m) \le 2cmn = kmn \qquad \square$$

Recap

- Can compute the cost of an alignment easily in linear space.

- Can compute the cost of a string with all suffixes of a second string in linear space.

- Divide and conquer algorithm for computing the *actual* alignment (traceback path in the DP matrix) in linear space.

- Still uses O(nm) time!