

# CSE 549:

# Computational Biology

Space-efficient alignment

# Space is often the limiting factor

$O(nm)$  time is a problem, but as I've said, we **strongly believe** we can't do much better.

Can we do better in terms of *space*?

It turns out we can — at the same asymptotic time complexity!

Combining dynamic programming with the divide-and-conquer algorithm design technique.

Hirshberg's algorithm

# Warmup — optimal *score* in linear space

Consider our DP matrix:

**y**

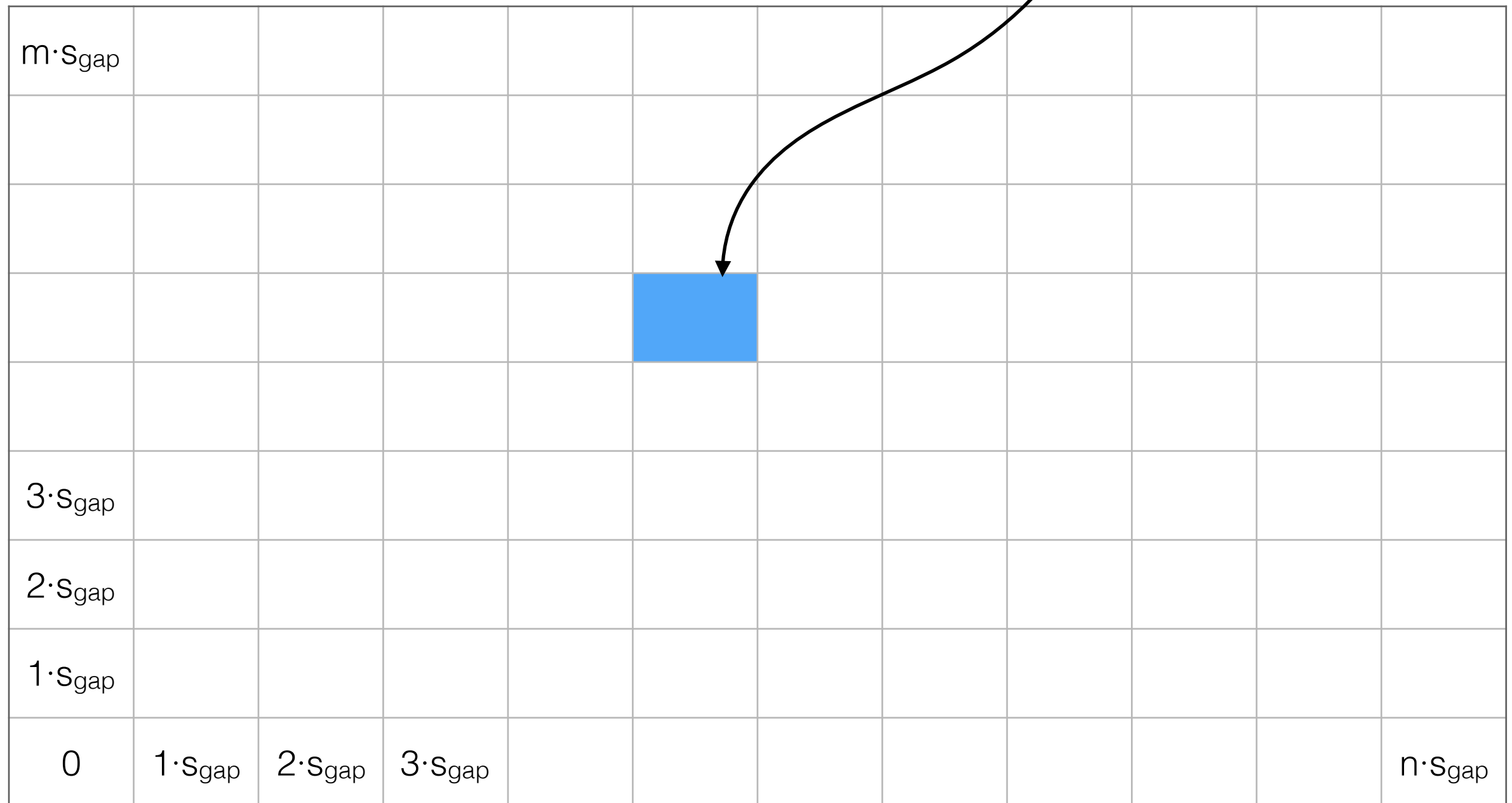
$m \cdot s_{\text{gap}}$											
$3 \cdot s_{\text{gap}}$											
$2 \cdot s_{\text{gap}}$											
$1 \cdot s_{\text{gap}}$											
0	$1 \cdot s_{\text{gap}}$	$2 \cdot s_{\text{gap}}$	$3 \cdot s_{\text{gap}}$								$n \cdot s_{\text{gap}}$

**x**

# Warmup — optimal *score* in linear space

What scores do I need to know to fill in the answer here?

**y**



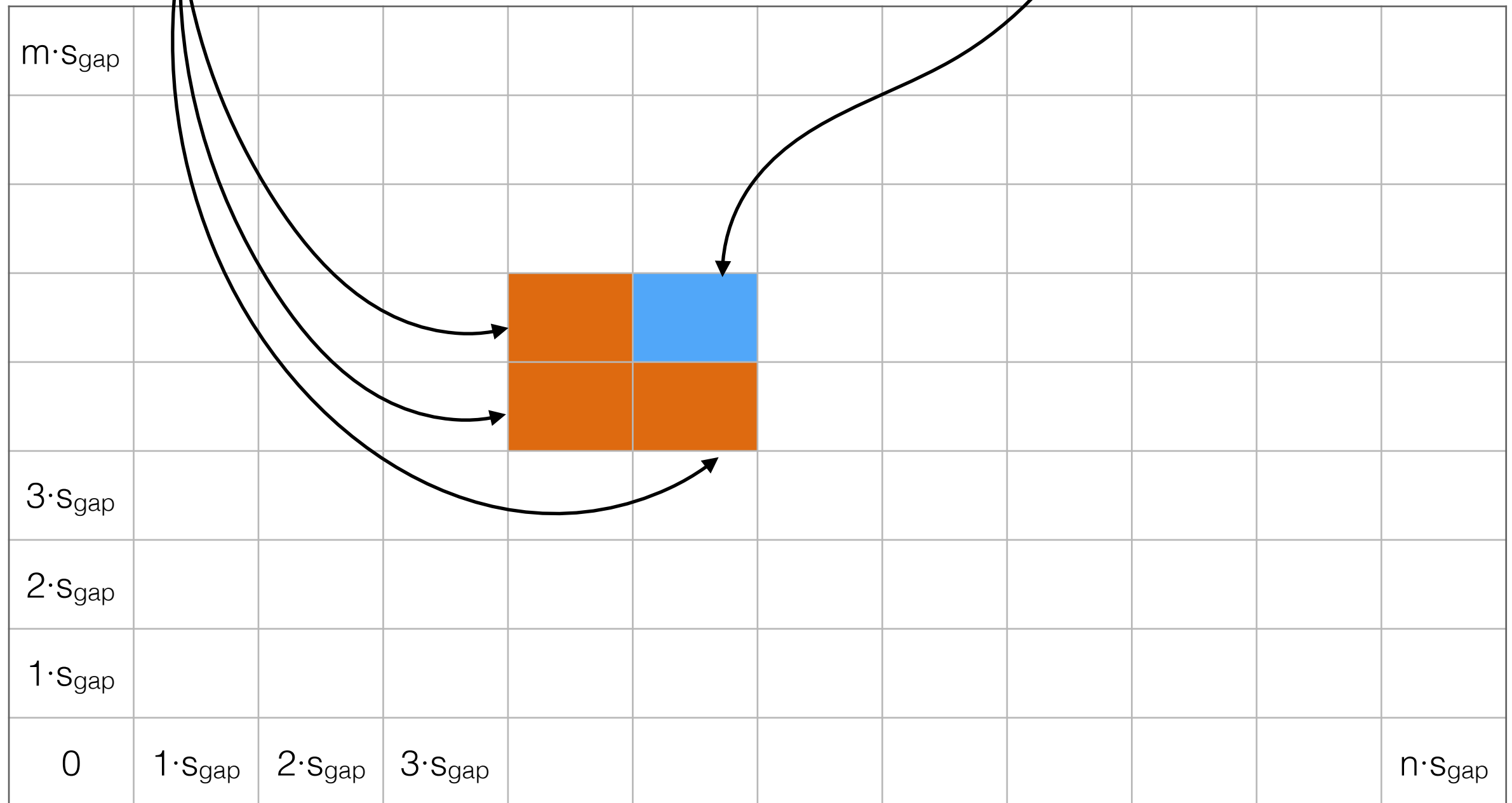
**x**

# Warmup — optimal *score* in linear space

What scores do I need to know to fill in the answer here?

These

**y**

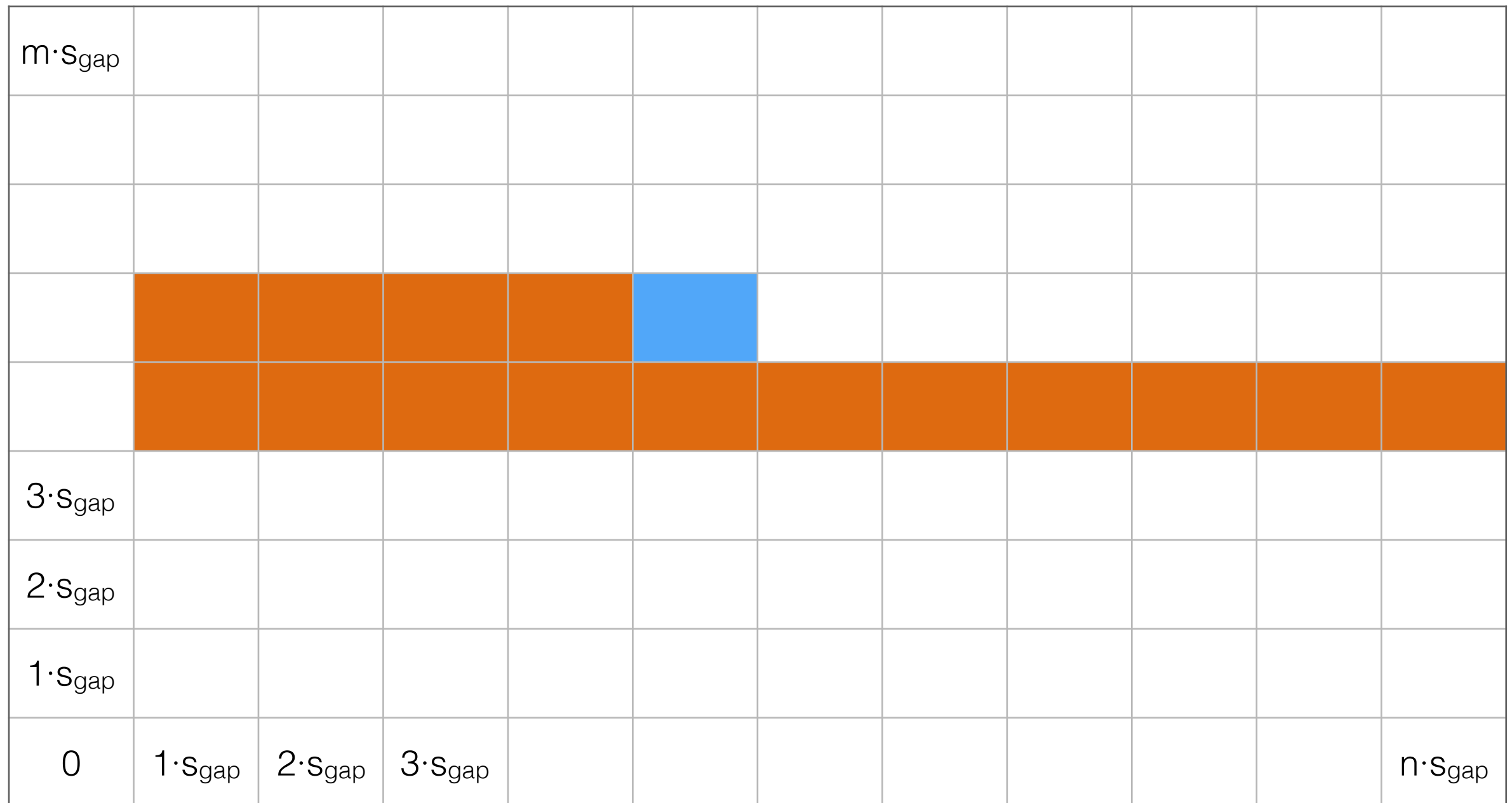


**x**

# Warmup — optimal *score* in linear space

If we fill rows left - right, and bottom to top, to fill in row  $i$ , we *only* need scores from row  $i-1$ .

**y**

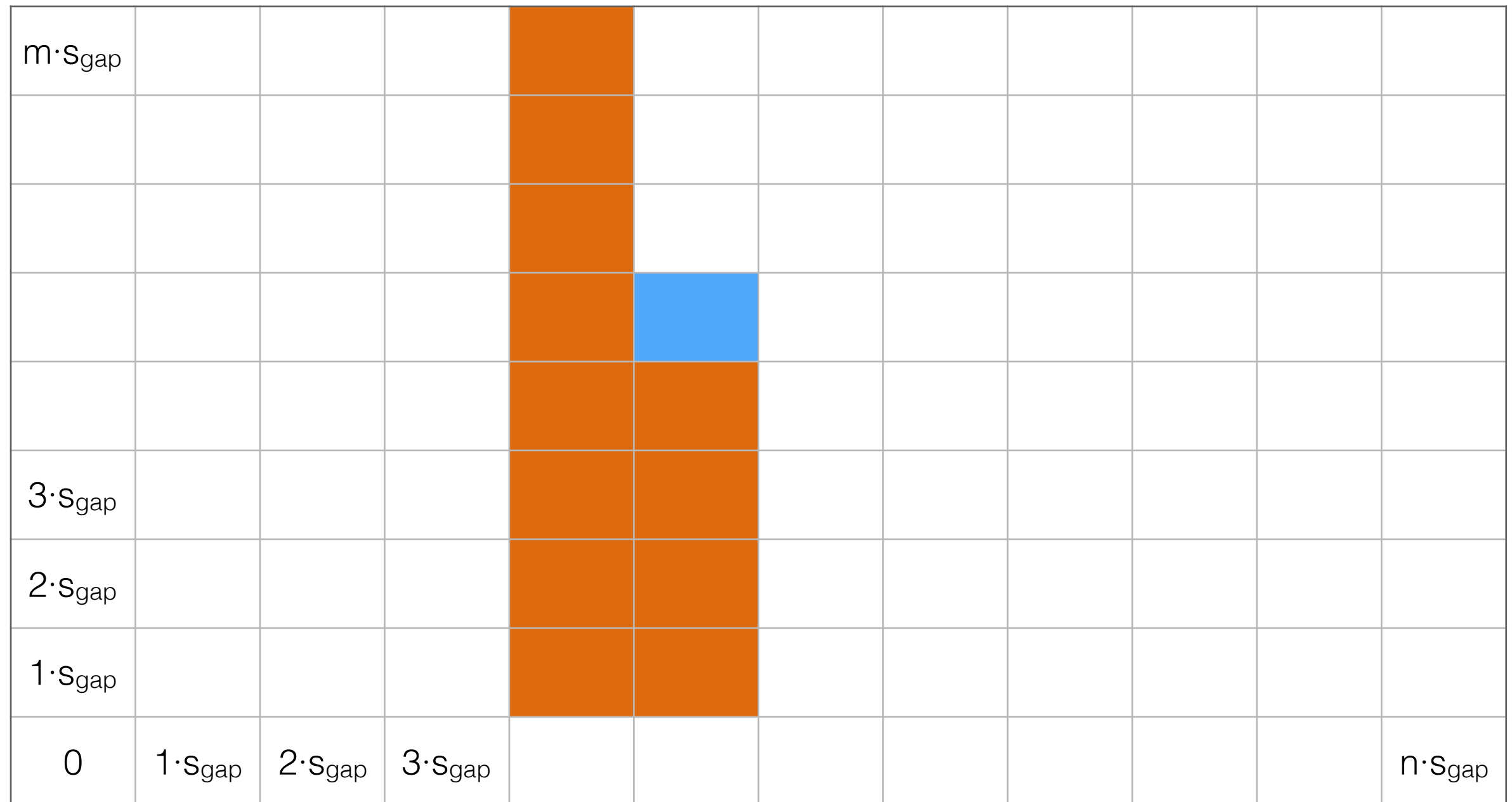


**X**

# Warmup — optimal *score* in linear space

*Columns also work*; if we go left - right, and bottom to top, to fill in column  $i$ , we *only* need scores from col  $i-1$ .

**y**



**x**

# Warmup — optimal *score* in linear space

If we fill rows left - right, and bottom to top, to fill in row  $i$ , we *only* need scores from row  $i-1$ .

Thus, we can compute the optimal *score*, keeping at most 2 rows / columns in memory at once.

Each row / column is *linear* in the length of one of the strings, and so we can compute the optimal *score*, in *linear space*.



# How can we compute the optimal *alignment*?

This method won't work for computing the optimal alignment; we need *all* rows to be able to follow the backtracking arrows.

How can we find the optimal *alignment* in linear space?

*Hirschberg's* algorithm provides a solution.

# Re-using subproblems

Consider, again, the meaning of the DP matrix

What is contained in the highlighted row?

**y**

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

**x**

# Re-using subproblems

Consider, again, the meaning of the DP matrix  
score of *every* prefix of **x** against *all* of **y** in this row

**y**

m·S <sub>gap</sub>											
3·S <sub>gap</sub>											
2·S <sub>gap</sub>											
1·S <sub>gap</sub>											
0	1·S <sub>gap</sub>	2·S <sub>gap</sub>	3·S <sub>gap</sub>								n·S <sub>gap</sub>

**x**

# Re-using subproblems

Consider, again, the meaning of the DP matrix

What is contained in the highlighted column?

**y**

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

**x**

# Re-using subproblems

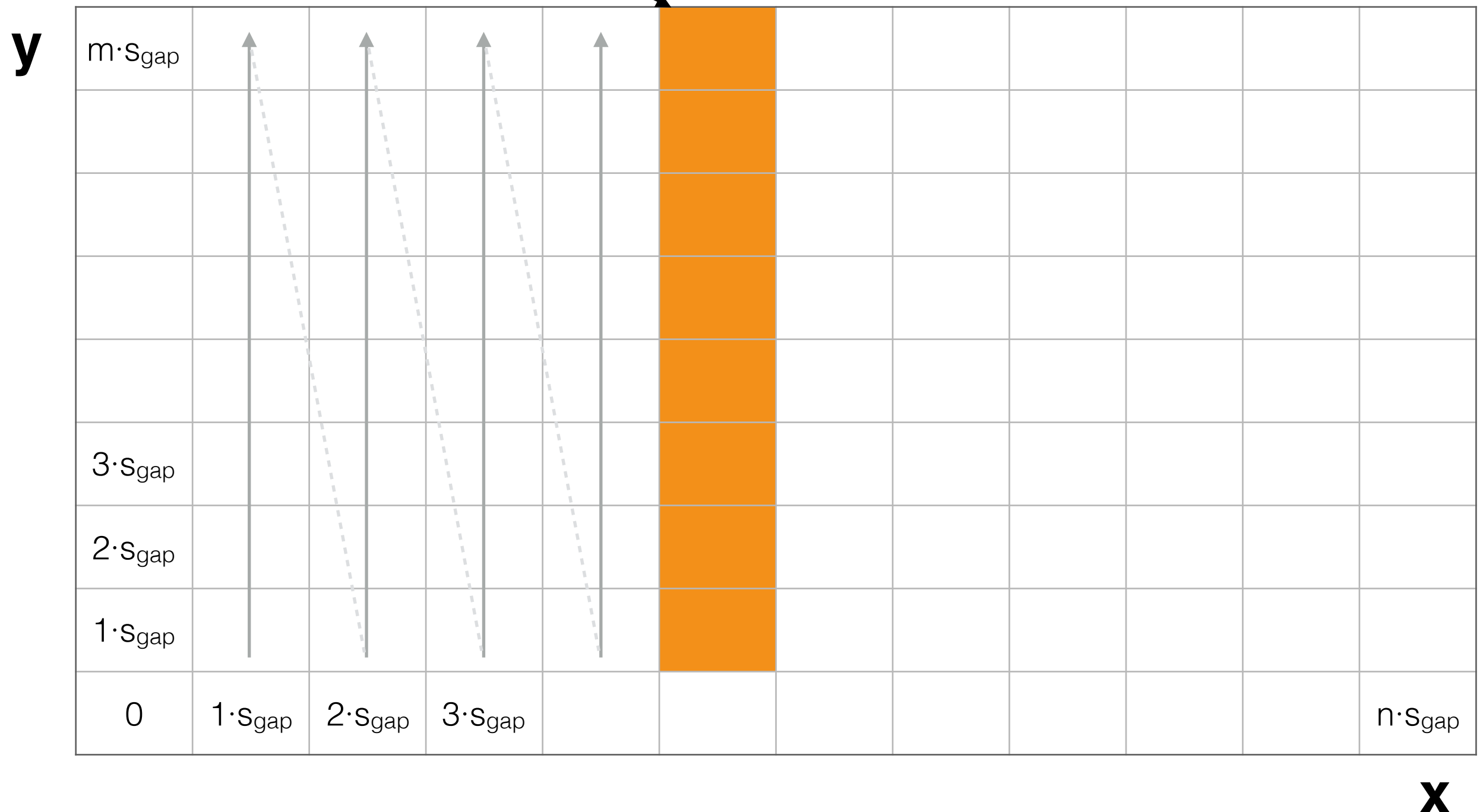
Consider, again, the meaning of the DP matrix

score of *every* prefix of **y** against *all* of **x** in this column

The figure shows a grid representing a coordinate system where both axes are measured in units of  $S_{\text{gap}}$ . The vertical axis ( $y$ ) ranges from  $0$  to  $m \cdot S_{\text{gap}}$ , and the horizontal axis ( $x$ ) ranges from  $0$  to  $n \cdot S_{\text{gap}}$ . A series of orange squares forms a diagonal path from the point  $(0, m \cdot S_{\text{gap}})$  down to  $(n \cdot S_{\text{gap}}, 0)$ . This visualizes the relationship between the two dimensions in terms of the gap size.

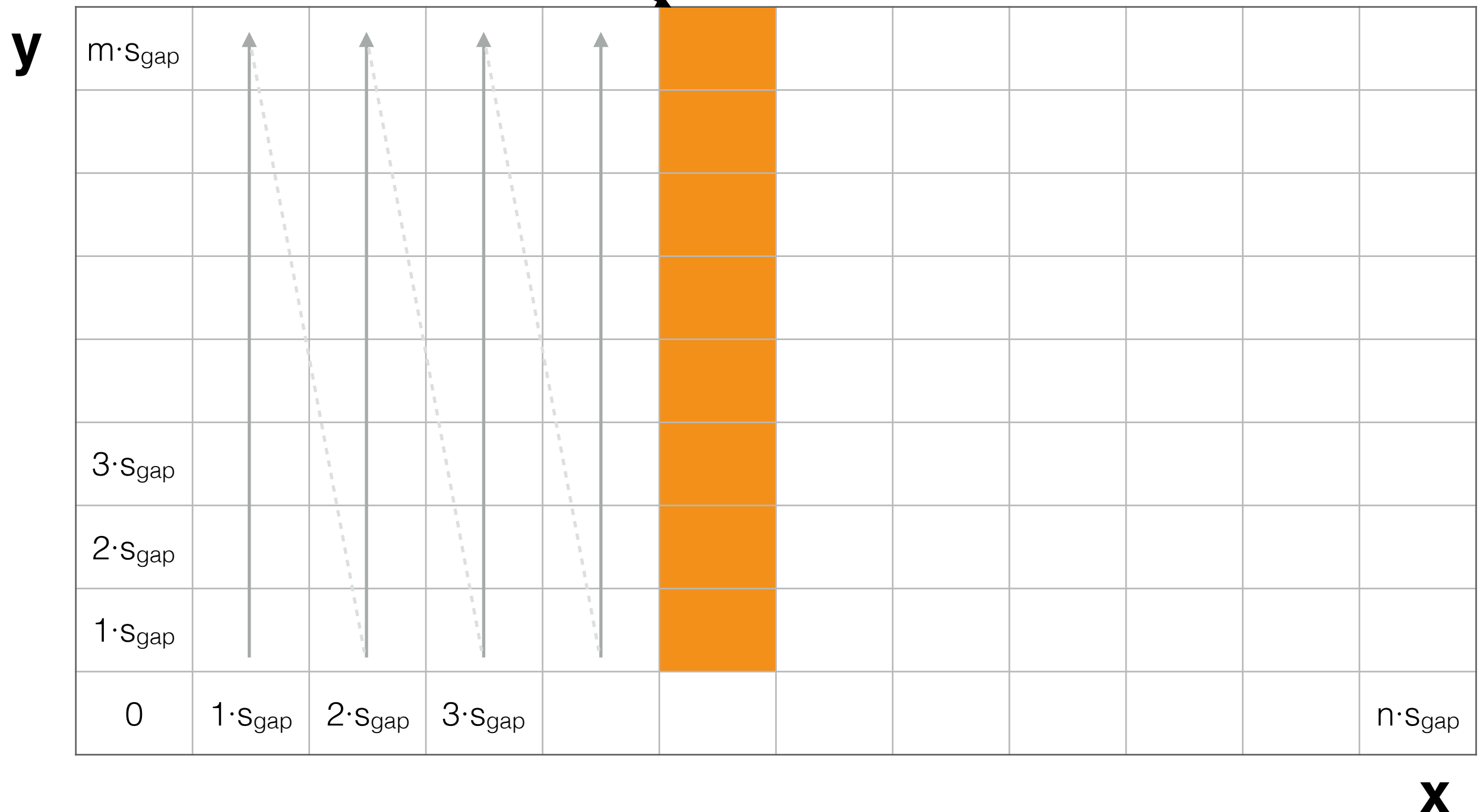
# Re-using subproblems

score of *every* prefix of **y** against  $i^{\text{th}}$  prefix of **x** in the  $i^{\text{th}}$  column. How do we get these values efficiently?



# Re-using subproblems

score of *every* prefix of **y** against  $i^{\text{th}}$  prefix of **x** in the  $i^{\text{th}}$  column. Easy if we fill in by columns instead of rows.



# What about suffixes?

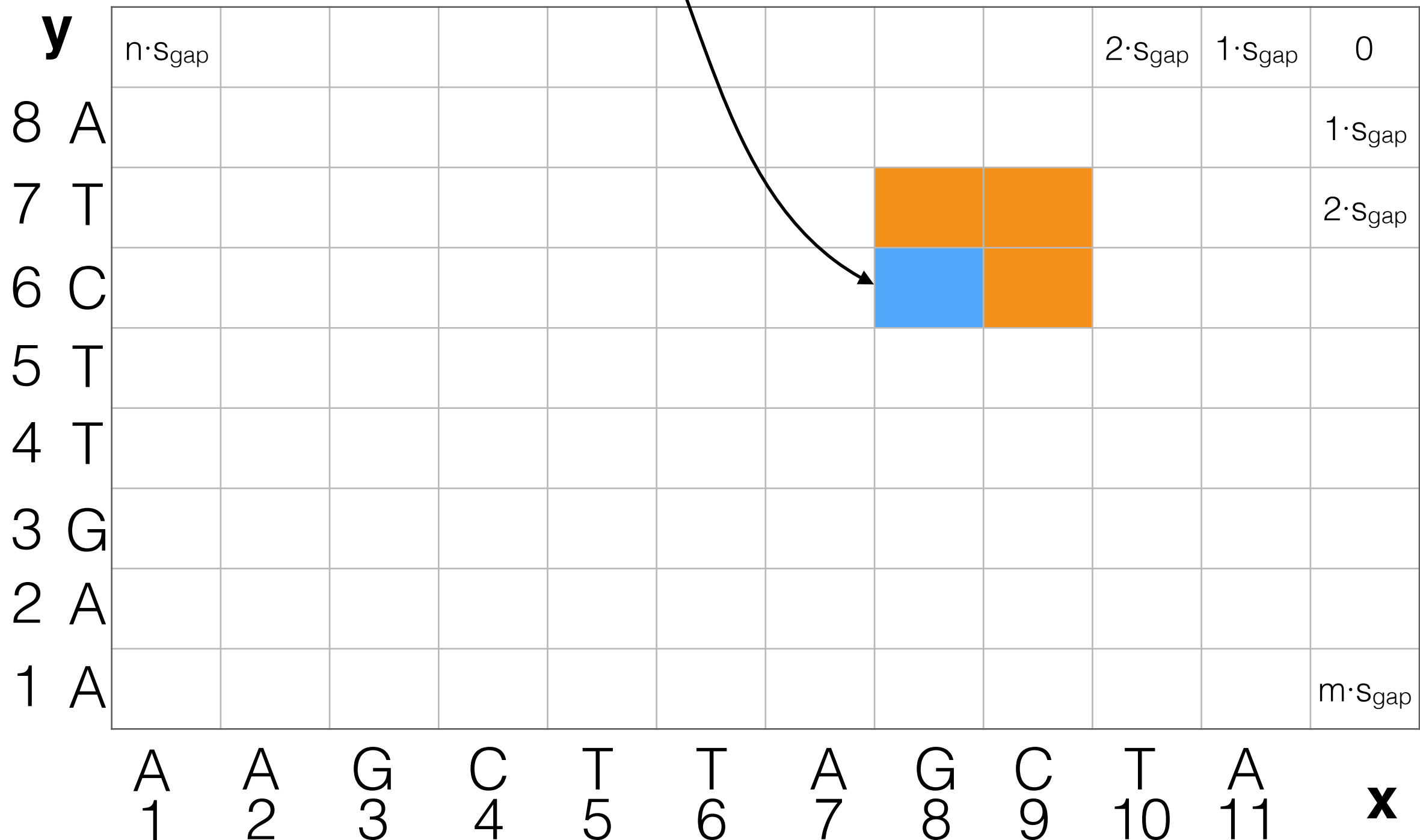
Consider filling in the DP matrix from the *opposite* direction (top right to bottom left)

<b>y</b>		$n \cdot S_{\text{gap}}$							$2 \cdot S_{\text{gap}}$		$1 \cdot S_{\text{gap}}$	0	
8	A										$1 \cdot S_{\text{gap}}$		
7	T										$2 \cdot S_{\text{gap}}$		
6	C												
5	T												
4	T												
3	G												
2	A												
1	A										$m \cdot S_{\text{gap}}$		
		A 1	A 2	G 3	C 4	T 5	T 6	A 7	G 8	C 9	T 10	A 11	<b>x</b>



# What about suffixes?

Optimal alignment between  $x[8:]$  and  $y[6:]$



# What about suffixes?

This lets us compute optimal score between a *suffix* of **x** with *all suffixes* of **y**

<b>y</b>		$n \cdot S_{\text{gap}}$								$2 \cdot S_{\text{gap}}$	$1 \cdot S_{\text{gap}}$	0	
8	A											$1 \cdot S_{\text{gap}}$	
7	T											$2 \cdot S_{\text{gap}}$	
6	C												
5	T												
4	T												
3	G												
2	A												
1	A											$m \cdot S_{\text{gap}}$	
		A 1	A 2	G 3	C 4	T 5	T 6	A 7	G 8	C 9	T 10	A 11	<b>x</b>

# What about suffixes?

Prefixes (forward):

$$\text{OPT}[i, j] = \max \begin{cases} \text{score}(x_i, y_j) + \text{OPT}'[i-1, j-1] \\ \text{gap} + \text{OPT}[i, j-1] \\ \text{gap} + \text{OPT}[i-1, j] \end{cases}$$

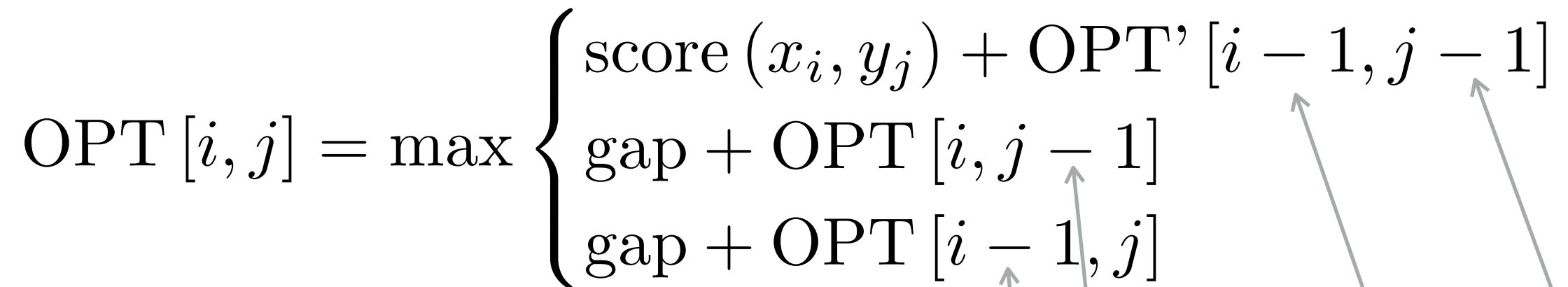
Suffixes (backward):

$$\text{OPT}'[i, j] = \max \begin{cases} \text{score}(x_{i+1}, y_{j+1}) + \text{OPT}'[i+1, j+1] \\ \text{gap} + \text{OPT}'[i, j+1] \\ \text{gap} + \text{OPT}'[i+1, j] \end{cases}$$

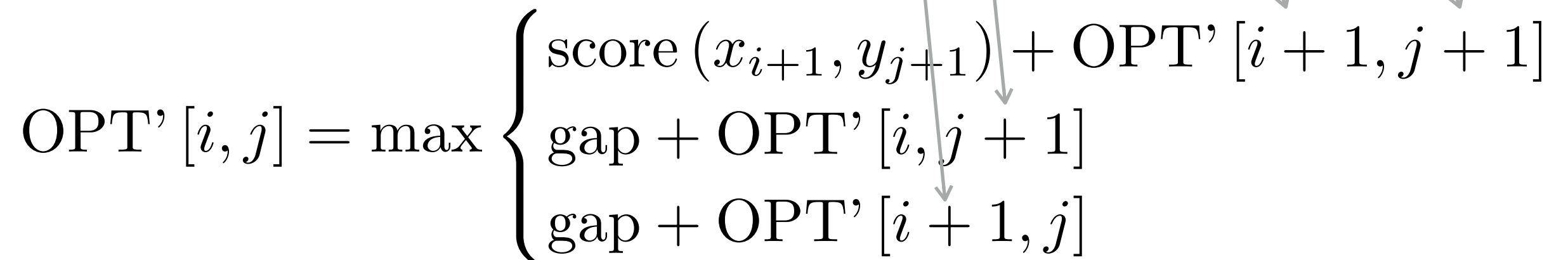
This lets us build up optimal alignments for increasing length suffixes of **x** and **y**

# What about suffixes?

Prefixes (forward):

$$\text{OPT}[i, j] = \max \begin{cases} \text{score}(x_i, y_j) + \text{OPT}'[i-1, j-1] \\ \text{gap} + \text{OPT}[i, j-1] \\ \text{gap} + \text{OPT}[i-1, j] \end{cases}$$


Suffixes (backward):

$$\text{OPT}'[i, j] = \max \begin{cases} \text{score}(x_{i+1}, y_{j+1}) + \text{OPT}[i+1, j+1] \\ \text{gap} + \text{OPT}'[i, j+1] \\ \text{gap} + \text{OPT}'[i+1, j] \end{cases}$$


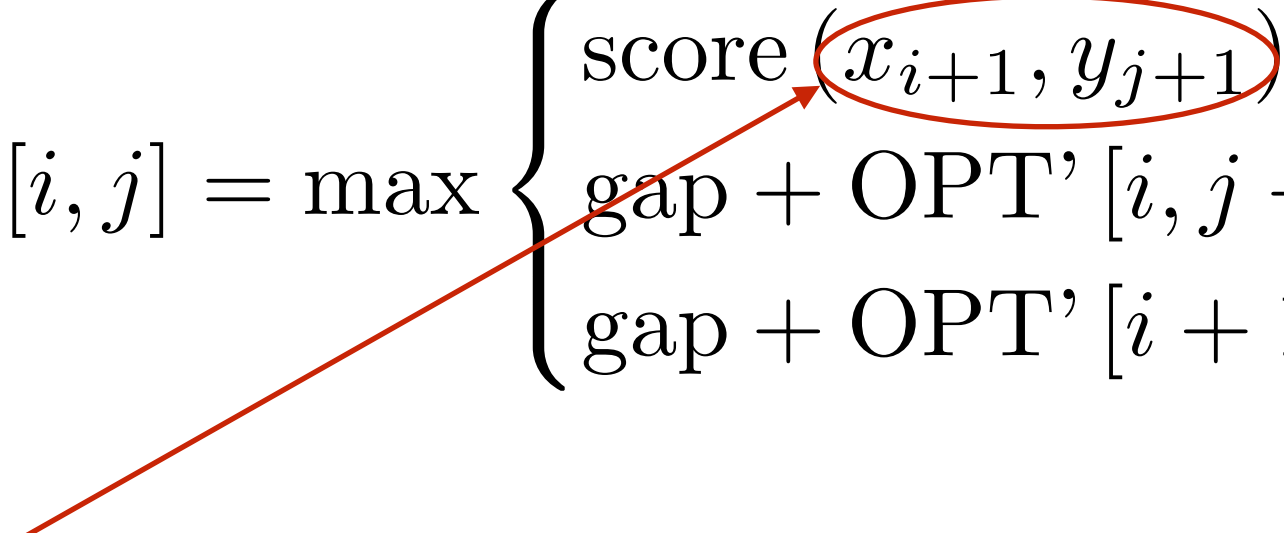
This lets us build up optimal alignments for increasing length suffixes of **x** and **y**

# What about suffixes?

Prefixes (forward):

$$\text{OPT}[i, j] = \max \begin{cases} \text{score}(x_i, y_j) + \text{OPT}'[i-1, j-1] \\ \text{gap} + \text{OPT}[i, j-1] \\ \text{gap} + \text{OPT}[i-1, j] \end{cases}$$

Suffixes (backward):

$$\text{OPT}'[i, j] = \max \begin{cases} \text{score}(x_{i+1}, y_{j+1}) + \text{OPT}'[i+1, j+1] \\ \text{gap} + \text{OPT}'[i, j+1] \\ \text{gap} + \text{OPT}'[i+1, j] \end{cases}$$


note: the slight change in indexing here. It will make writing our solution easier.

# Finding the optimal alignment

How does this help us compute the optimal alignment in linear space?

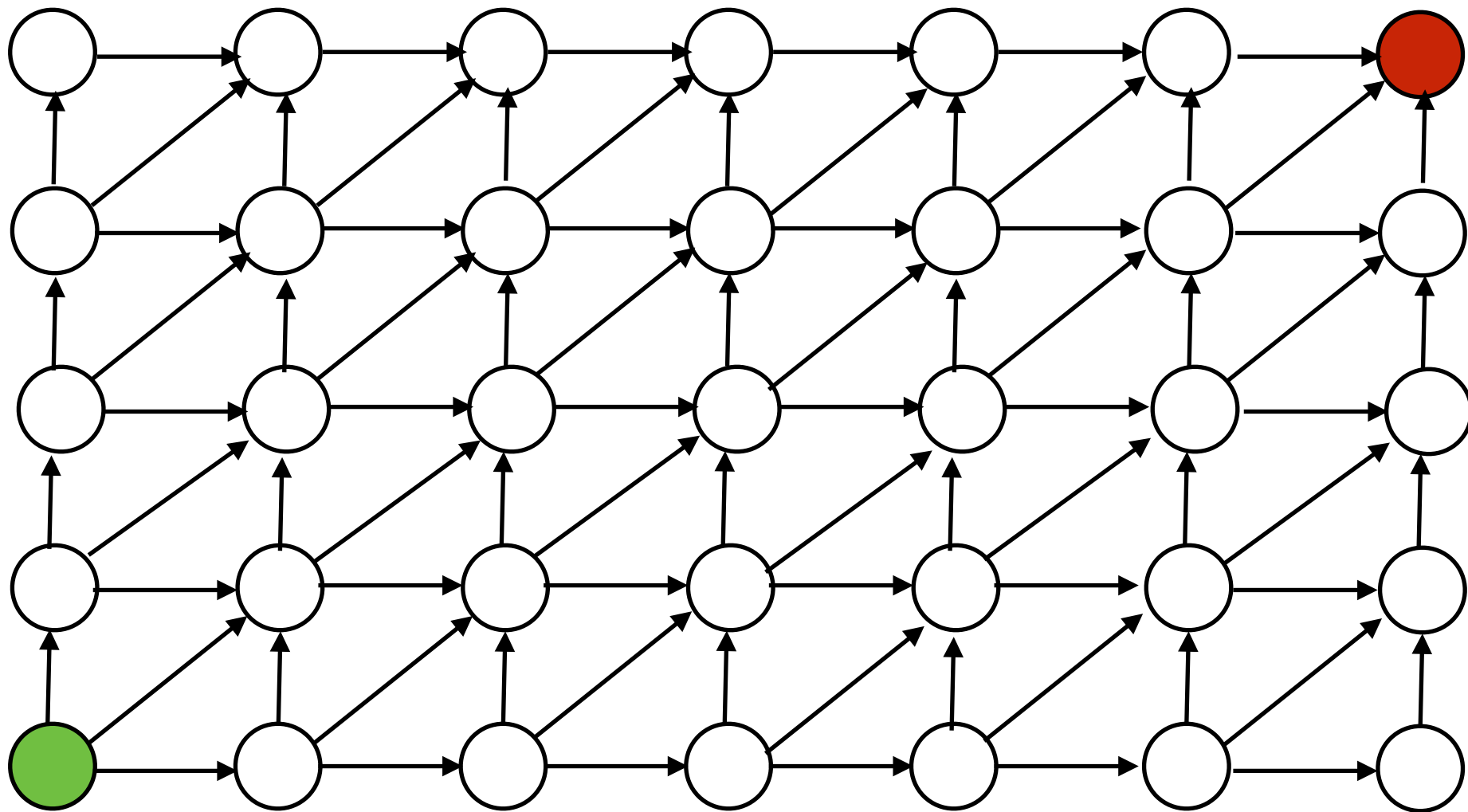
**Algorithmic idea:** Combine both dynamic programs using *divide-and-conquer*

Divide-and-conquer splits a problem into smaller sub-problems and combines the results (much like DP).

Examples: MergeSort & Karatsuba multiplication

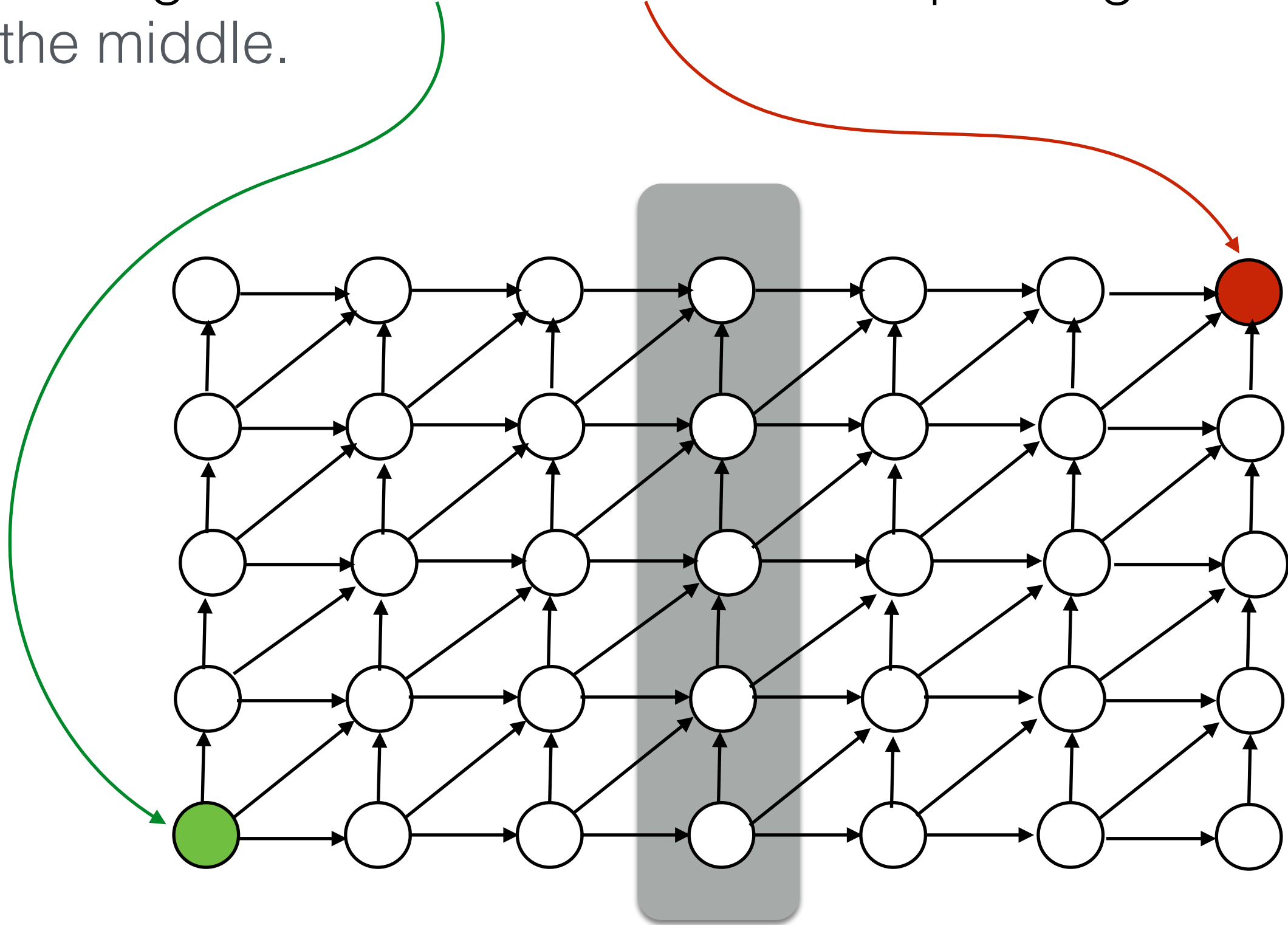
# Think about this in “graph” land

What do we know about the structure of the optimal path in our “edit-DAG”?



# Think about this in “graph” land

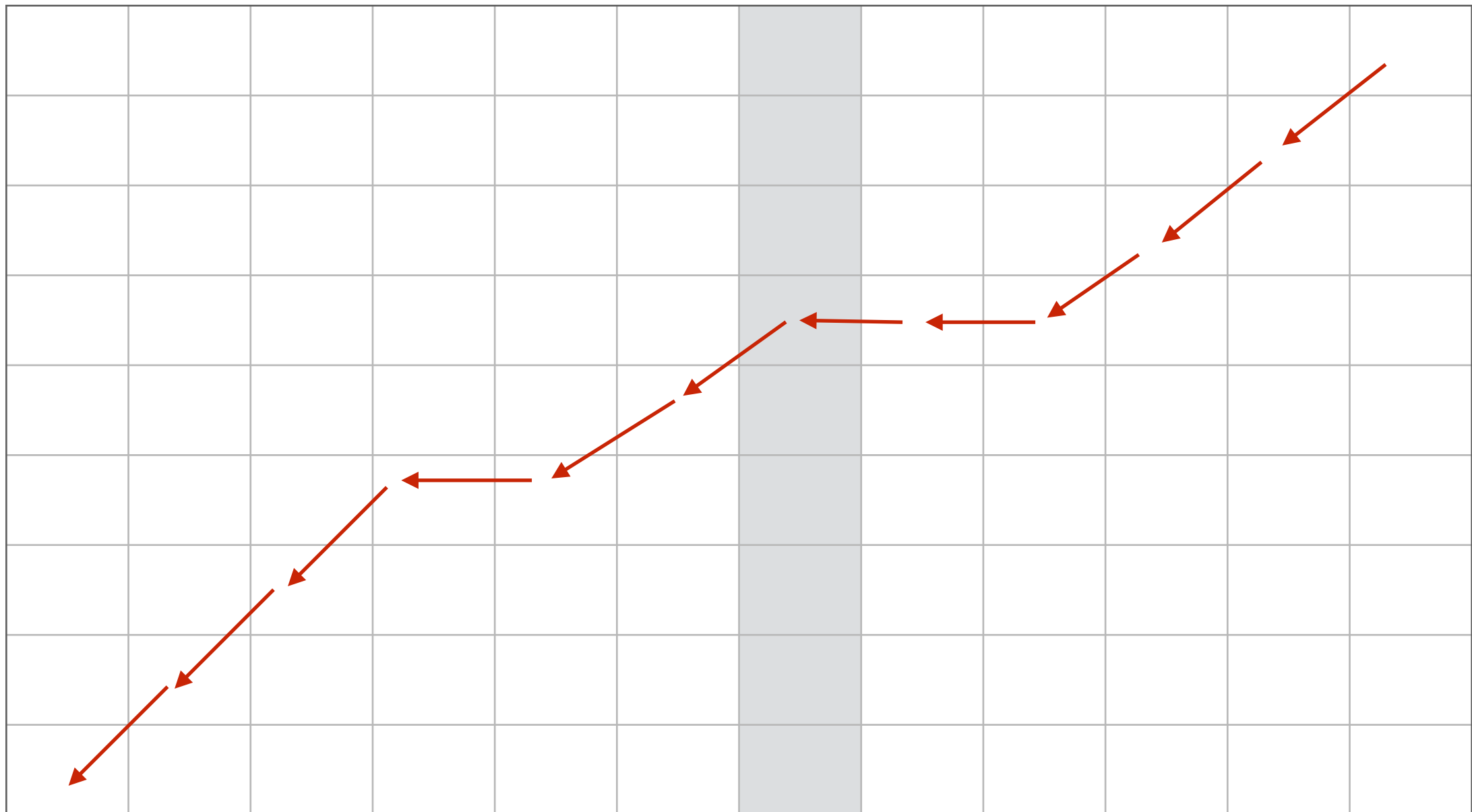
Can't get from **here** to **there** without passing through the middle.





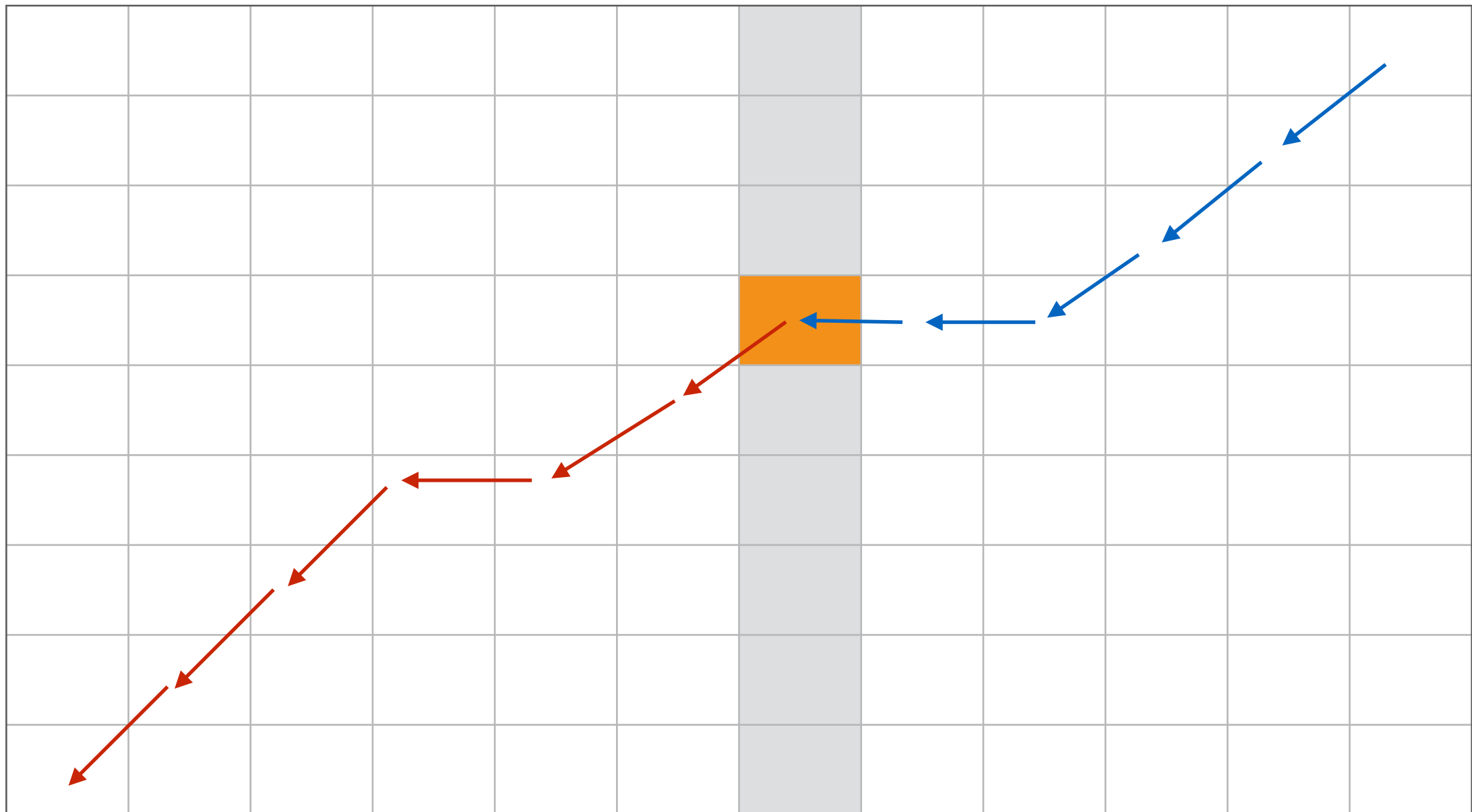
# Finding the optimal alignment

Consider the middle column — we *know* that the optimal aln. must use some cell in this column; which one?



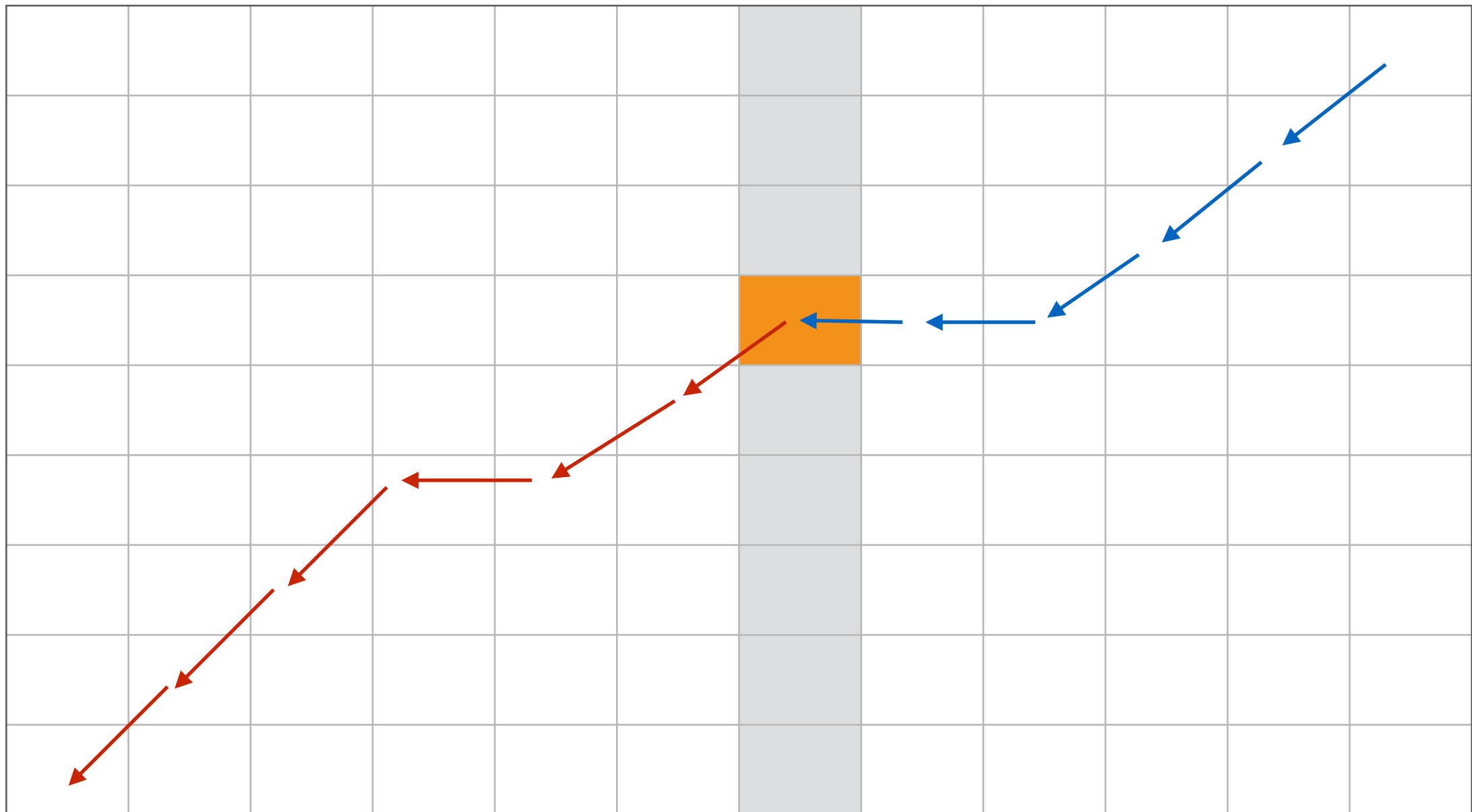
# Finding the optimal alignment

It uses the cell  $(i,j)$  such that  $\text{OPT}[i,j] + \text{OPT}'[i,j]$  has the **highest score**. Equivalently, the *best path* uses some vertex  $v$  in the middle col. and glues together the best paths from the source *to*  $v$  and *from*  $v$  to the sink.



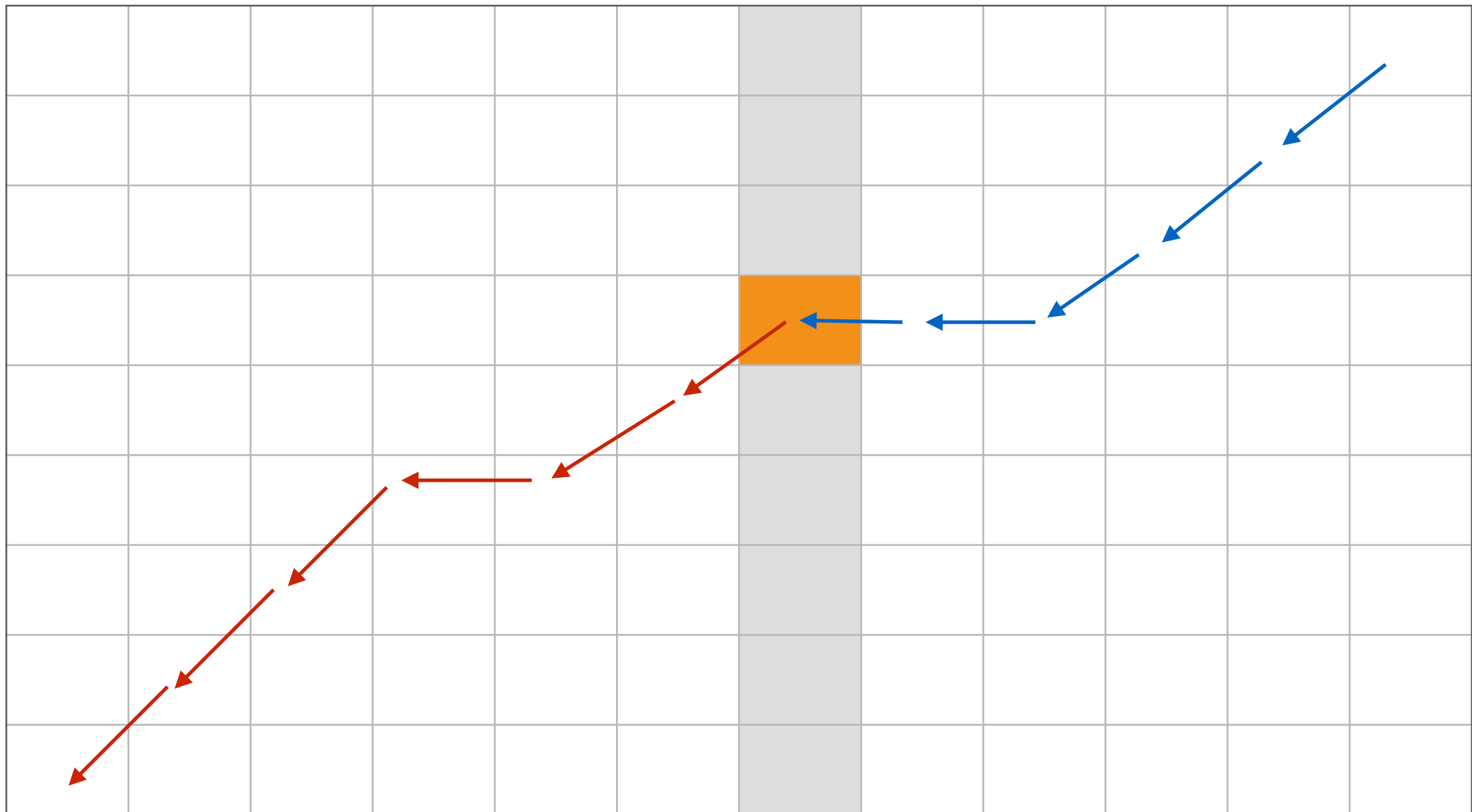
# Finding the optimal alignment

Claim:  $OPT[i,j]$  and  $OPT'[i,j]$  can be computed in linear space using the trick from above for finding an optimal **score** in linear space



# Algorithmic Idea

Devise a D&C algorithm that finds the optimal alignment path recursively, using the space-efficient scoring algorithm for each subproblem.



# D&C Alignment

```
DCAalignment(x, y):  
  n = |x|  
  m = |y|  
  if m <= 2 or n <= 2:  
    use “normal” DP to compute OPT(x, y)  
  compute space-efficient OPT(x[1:n/2], y)  
  compute space-efficient OPT'(x[n/2+1:n], y)  
  let q be the index maximizing OPT[n/2,q] + OPT'[n/2,q]  
  add back pointer of (n/2,q) to the optimal alignment P  
  DCAalignment(x[1:n/2], y[1:q])  
  DCAalignment(x[n/2+1:n], y[q+1:m])  
  return P
```

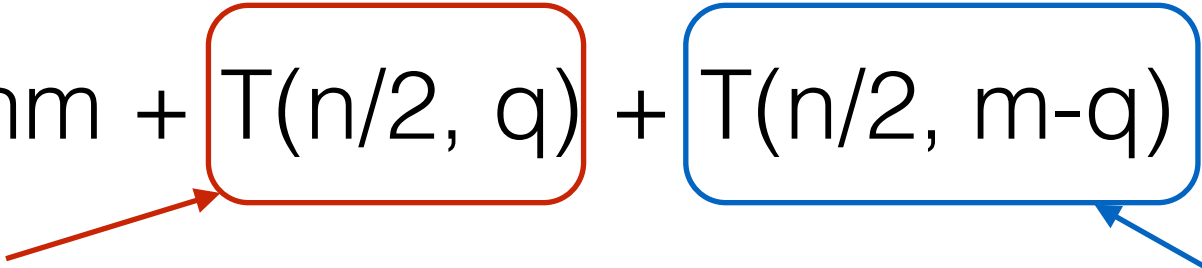
# D&C Alignment

How can we show that this entire process still takes quadratic time?

Let  $T(n,m)$  be the running time on strings **x** and **y** of length  $n$  and  $m$ , respectively. We have:

$$T(n,m) \leq cnm + \boxed{T(n/2, q)} + \boxed{T(n/2, m-q)}$$

$\text{DCAAlignment}(x[1:n/2], y[1:q])$        $\text{DCAAlignment}(x[n/2+1:n], y[q+1:m])$



with base cases:

$$T(n,2) \leq cn$$

$$T(2,m) \leq cm$$

# D&C Alignment

Base:

$$T(n, 2) \leq cn$$

$$T(2, m) \leq cm$$

Inductive:

$$T(n, m) \leq cnm + T(n/2, q) + T(n/2, m-q)$$

*Problem:* we don't know what  $q$  is. First, assume both  **$x$**  and  **$y$**  have length  $n$  and  $q=n/2$   
(will remove this restriction later)

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

This recursion solves as  $T(n) = O(n^2)$

Leads us to guess  $T(n, m)$  grows like  $O(nm)$

# Smarter Induction

Base:

$$T(n,2) \leq cn$$

$$T(2,m) \leq cm$$

Inductive:

$$T(n,m) \leq knm$$

Proof:

$$\begin{aligned} T(n,m) &\leq cnm + T(n/2, q) + T(n/2, m-q) \\ &\leq cnm + kqn/2 + k(m-q)n/2 \\ &\leq cnm + kqn/2 + kmn/2 - kqn/2 \\ &= [c+(k/2)] mn \end{aligned}$$

Thus, our proof holds if  $k=2c$ , and  $T(n,m) = O(nm)$  QED



# Conclusion

Trivially, we can compute the *cost* of an optimal alignment in linear space

By arranging subproblems intelligently we can define a “reverse” DP that works on suffixes instead of prefixes

Combining the “forward” and “reverse” DP using a divide and conquer technique, we can compute the optimal *solution* (not just the score) in linear space.

This still only takes  $O(nm)$  time; constant factor more work than the “forward”-only algorithm.