CSE 373: Gap Penalties (structural constraints) and Linear Space Alignment

Stony Brook University

slides (w/*) courtesy of Carl Kingsford

General Gap Penalties

```
AAAGAATTCA          AAAGAATTCA
A-A-A-T-CA    vs.   AAA----TCA
```

These have the same score, but the second one is often more plausible.

A single insertion of "GAAT" into the first string could change it into the second — Biologically, this is much more likely as **x** could be transformed into **y** in "one fell swoop".

- Currently, the score of a run of k gaps is $s_{gap} \times k$
- It might be more realistic to support general gap penalty, so that the score of a run of k gaps is $|\mathbf{gscore}(k)| < |(s_{gap} \times k)|$.
- Then, the optimization will prefer to group gaps together.

*

General Gap Penalties — The Problem

```
AAAGAATTCA          AAAGAATTCA
A-A-A-T-CA    vs.   AAA----TCA
```

Previous DP no longer works with general gap penalties.

Why?

*

General Gap Penalties — The Problem

```
AAAGAATTCA        vs.        AAAGAATTCA
A-A-A-T-CA                   AAA----TCA
```

The score of the *last character* depends on *details* of the previous alignment:

```
AAAGAAC           vs.        AAAGAATC
AAA----                      AAA-----
```

We need to "know" how long a final run of gaps is in order to give a score to the last subproblem.

*

General Gap Penalties — The Problem

The score of the *last character* depends on *details* of the previous alignment:

Knowing the optimal alignment at the substring ending here.

```
AAAGAAC          AAAGAATC
AAA----    vs.   AAA-----
```

Doesn't let us simply build the optimal alignment ending here.

*

Three Matrices

We now keep 3 different matrices:

M(i,j) = score of best alignment of x[1..i] and y[1..j] ending with a character-character **match or mismatch**.

X(i,j) = score of best alignment of x[1..i] and y[1..j] ending with a **gap in X.**

Y(i,j) = score of best alignment of x[1..i] and y[1..j] ending with a **gap in Y.**

$$\mathrm{M}(i,j) = \mathrm{score}(x_i, y_j) + \max \begin{cases} \mathrm{M}(i-1, j-1) \\ \mathrm{X}(i-1, j-1) \\ \mathrm{Y}(i-1, j-1) \end{cases}$$

$$\mathrm{X}(i,j) = \max \begin{cases} \mathrm{M}(i, j-k) + \mathrm{gscore}(k) & \text{for } 1 \le k \le j \\ \mathrm{Y}(i, j-k) + \mathrm{gscore}(k) & \text{for } 1 \le k \le j \end{cases}$$

$$\mathrm{Y}(i,j) = \max \begin{cases} \mathrm{M}(i-k, j) + \mathrm{gscore}(k) & \text{for } 1 \le k \le i \\ \mathrm{X}(i-k, j) + \mathrm{gscore}(k) & \text{for } 1 \le k \le i \end{cases}$$

*

The M Matrix

We now keep 3 different matrices:

M(i,j) = score of best alignment of x[1..i] and y[1..j] ending with a character-character **match or mismatch**.

X(i,j) = score of best alignment of x[1..i] and y[1..j] ending with a **gap in X**.

Y(i,j) = score of best alignment of x[1..i] and y[1..j] ending with a **gap in Y**.

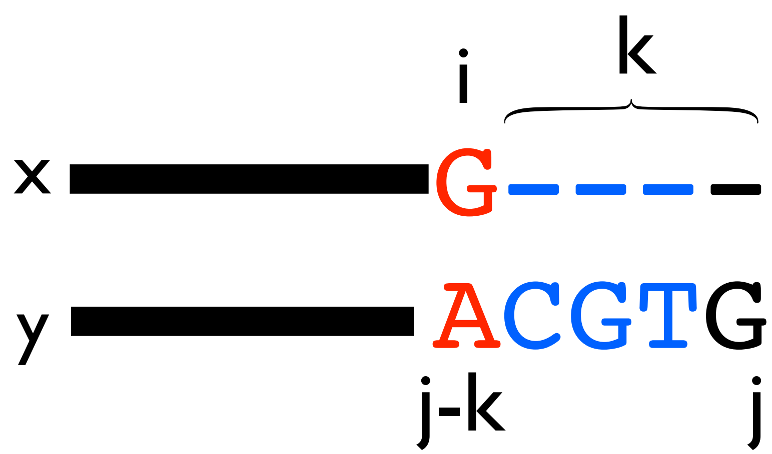
By definition, alignment ends in a match/mismatch.

$$\mathrm{M}(i,j) = \mathrm{score}(x_i, y_j) + \max \begin{cases} \mathrm{M}(i-1, j-1) \\ \mathrm{X}(i-1, j-1) \\ \mathrm{Y}(i-1, j-1) \end{cases}$$

Any kind of alignment is allowed before the match/mismatch.

*

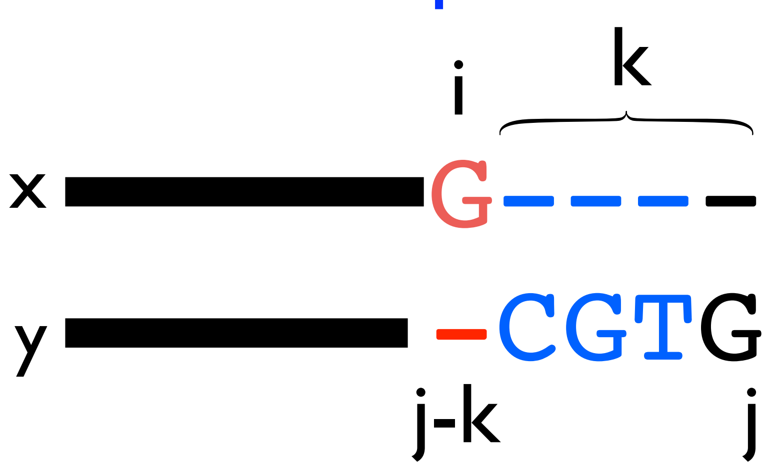
The X (and Y) matrices



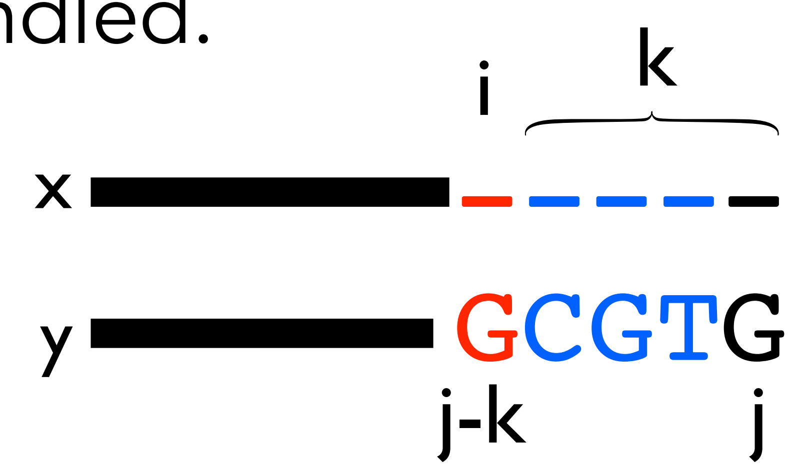
k decides how long to make the gap.

We have to make the whole gap at once in order to know how to score it.

$$X(i,j) = \max \begin{cases} M(i, j-k) + \text{gscore}(k) & \text{for } 1 \le k \le j \\ Y(i, j-k) + \text{gscore}(k) & \text{for } 1 \le k \le j \end{cases}$$



This case is automatically handled.



*

Running Time for Gap Penalties

$$\mathrm{M}(i,j) = \mathrm{score}(x_i, y_j) + \max \begin{cases} \mathrm{M}(i-1, j-1) \\ \mathrm{X}(i-1, j-1) \\ \mathrm{Y}(i-1, j-1) \end{cases}$$

$$\mathrm{X}(i,j) = \max \begin{cases} \mathrm{M}(i, j-k) + \mathrm{gscore}(k) & \text{for } 1 \le k \le j \\ \mathrm{Y}(i, j-k) + \mathrm{gscore}(k) & \text{for } 1 \le k \le j \end{cases}$$

$$\mathrm{Y}(i,j) = \max \begin{cases} \mathrm{M}(i-k, j) + \mathrm{gscore}(k) & \text{for } 1 \le k \le i \\ \mathrm{X}(i-k, j) + \mathrm{gscore}(k) & \text{for } 1 \le k \le i \end{cases}$$

Final score is max {M(n,m), X(n,m), Y(n,m)}.

How do you do the traceback?

Runtime:

- Assume |X| = |Y| = n for simplicity: $3n^2$ subproblems
- $2n^2$ subproblems take O(n) time to solve (**because we have to try all k**)

⇒ O(n^3) total time

*

Affine Gap Penalties

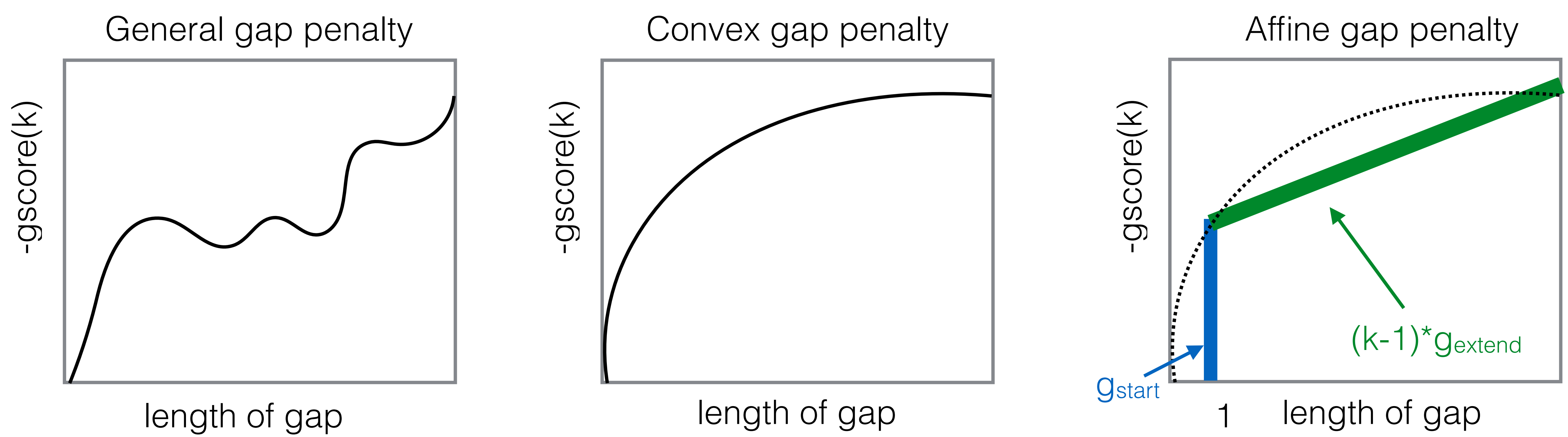
- $O(n^3)$ for general gap penalties is usually too slow…
- We can still encourage spaces to group together using a special case of general penalties called *affine gap penalties*:

  g_{start} = the cost of starting a gap

  g_{extend} = the cost of extending a gap by one more space

  $gscore(k) = g_{start} + (k-1) \times g_{extend}$

less restrictive ⇒ more restrictive

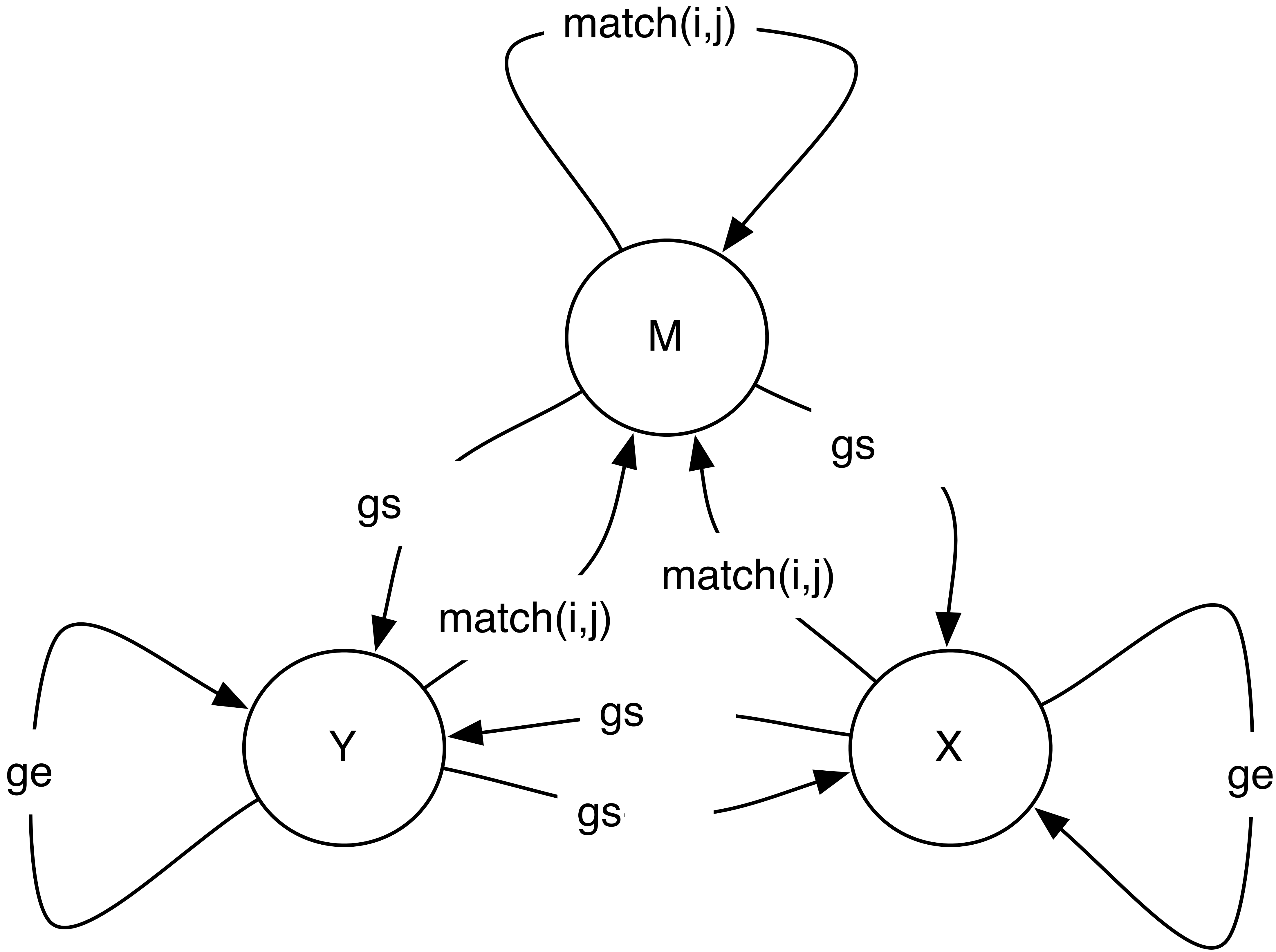


Benefit of Affine Gap Penalties

- Same idea of using 3 matrices, but now we *don't need to search over all gap lengths*, we just have to know whether we are **starting a new gap** or **not**.

*

Affine Gap as Finite State Machine



*

Affine Gap Penalties

$$M(i,j) = \text{score}(x_i, y_i) + \max \begin{cases} M(i-1, j-1) \\ X(i-1, j-1) \\ Y(i-1, j-1) \end{cases}$$

(mis)match between x and y

If previous alignment ends in (mis)match, this is a new gap

$$X(i,j) = \max \begin{cases} g_{\text{start}} + M(i, j-1) \\ g_{\text{extend}} + X(i, j-1) \\ g_{\text{start}} + Y(i, j-1) \end{cases}$$

gap in x

If we're using the X matrix, then we're extending a gap.

$$Y(i,j) = \max \begin{cases} g_{\text{start}} + M(i-1, j) \\ g_{\text{start}} + X(i-1, j) \\ g_{\text{extend}} + Y(i-1, j) \end{cases}$$

gap in y

If we're using the Y matrix, then we're starting a new gap in this string.

*

Affine Base Cases (Global)

- $M(0, i)$ = "score of best alignment between 0 characters of x and i characters of y that ends in a match" $= -\infty$ because no such alignment can exist.

- $X(0, i)$ = "score of best alignment between 0 characters of x and i characters of y that ends in a gap in x" = gap_start + (i-1) × gap_extend because this alignment looks like:

```
---------
yyyyyyyyy
```

- X(i, 0) = "score of best alignment between i characters of x and 0 characters of y that ends in a gap in X" $= -\infty$

```
xxxxxxxxx-
----------
```

← not allowed

- M(i, 0) = M(0, i) and Y(0, i) and Y(i, 0) are computed using the same logic as X(i, 0) and X(0, i)

*

Affine Gap Runtime

- 3*mn* subproblems
- Each one takes **constant** time
- Total runtime O(*mn*):
  - back to the run time of the basic running time.

Traceback

- Arrows now can point **between** matrices.
- The possible arrows are given, as usual, by the recurrence.
  - E.g. What arrows are possible leaving a cell in the M matrix?

*

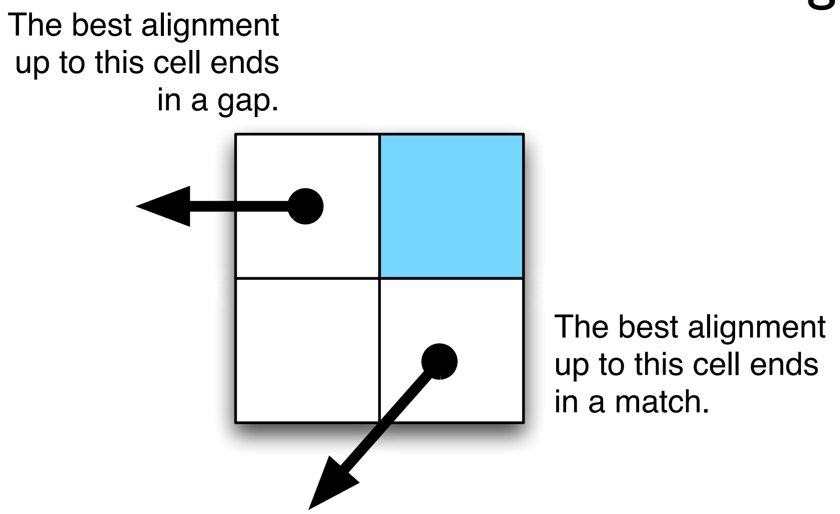
Why do you "need" 3 functions?

- Alternative WRONG algorithm:

```
M(i,j) = max(
    M(i-1, j-1) + cost(xi, yj),
    M(i-1, j) +(gstart if Arrow(i-1, j) != ←, else gextend),
    M(j, i-1) + (gstart if Arrow(i, j-1) != ↓ , else gextend)
)
```

WRONG Intuition: we only need to know whether we are starting a gap or extending a gap.

The arrows coming out of each subproblem tell us how the best alignment ends, so we can use them to decide if we are starting a new gap.



PROBLEM: The best alignment for strings x[1..i] and y[1..j] doesn't have to be used in the best alignment between x[1..i+1] and y[1..j+1]

*

Why 3 Matrices: Example

match = 5, mismatch = -2, gap = -1, gap_start = -10

`x=CARTS, y=CAT`

```
CART
CA-T
```

OPT(4, 3) = optimal score = $\underbrace{15 - 10}$ = 5

```
CARTS
CA-T-
```

WRONG(5, 3) = $\overbrace{15 - 10}$ - 10 = -5

```
CARTS
CAT--
```

OPT(5, 3) = 10 - 2 - 10 - 1 = -3

this is why we need to keep the X and Y matrices around.
they tell us the score of ending with a gap in one of the sequences.

*

Side Note: Lower Bounds

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

(Alrazarov, Dinic, Kronrod, Faradzev, 1970)

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

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

*

Space is often the limiting factor

O(nm) time is a problem, but as I've said, we **strongly believe** we can't to 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:

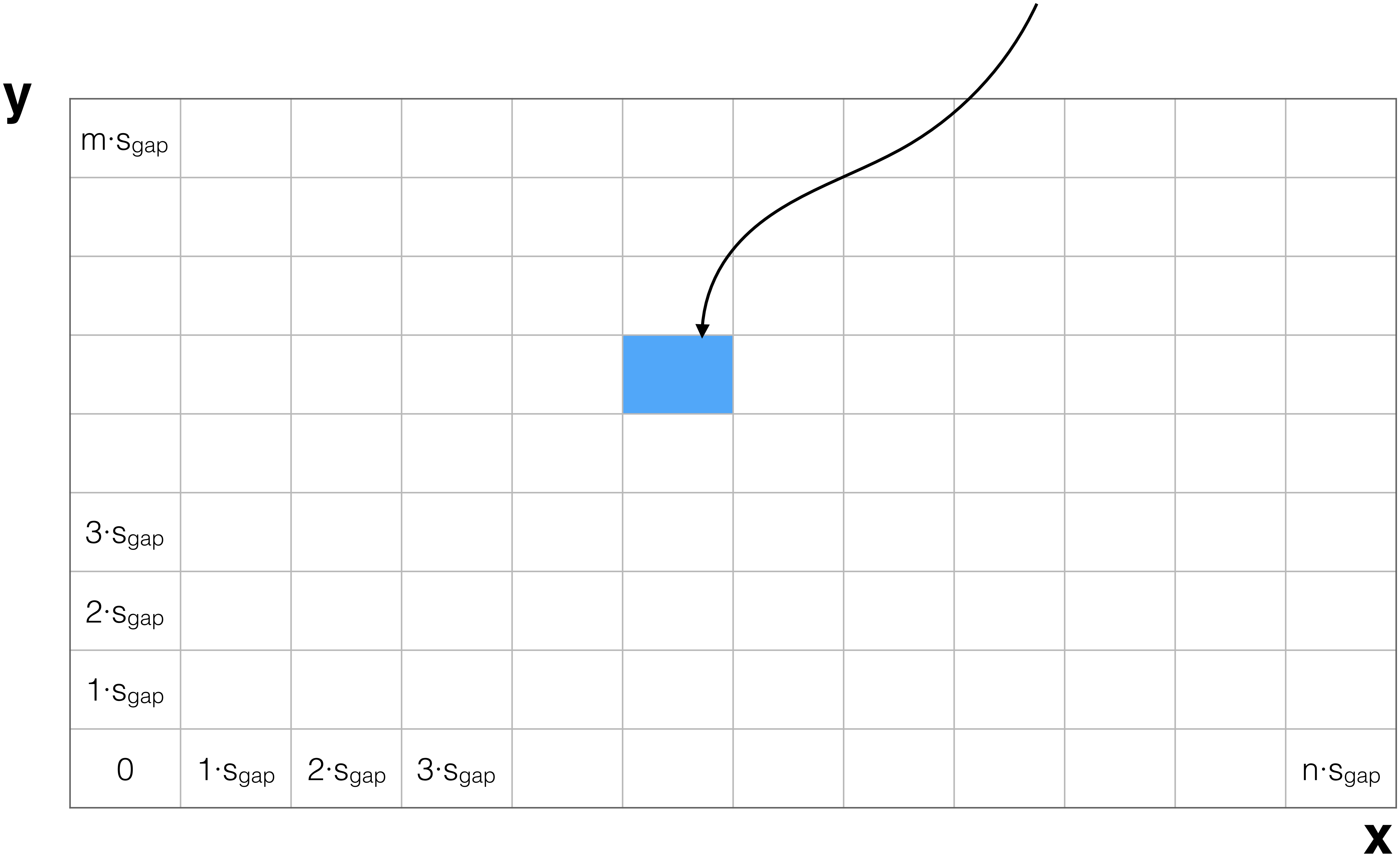
y

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

x

Warmup — optimal *score* in linear space

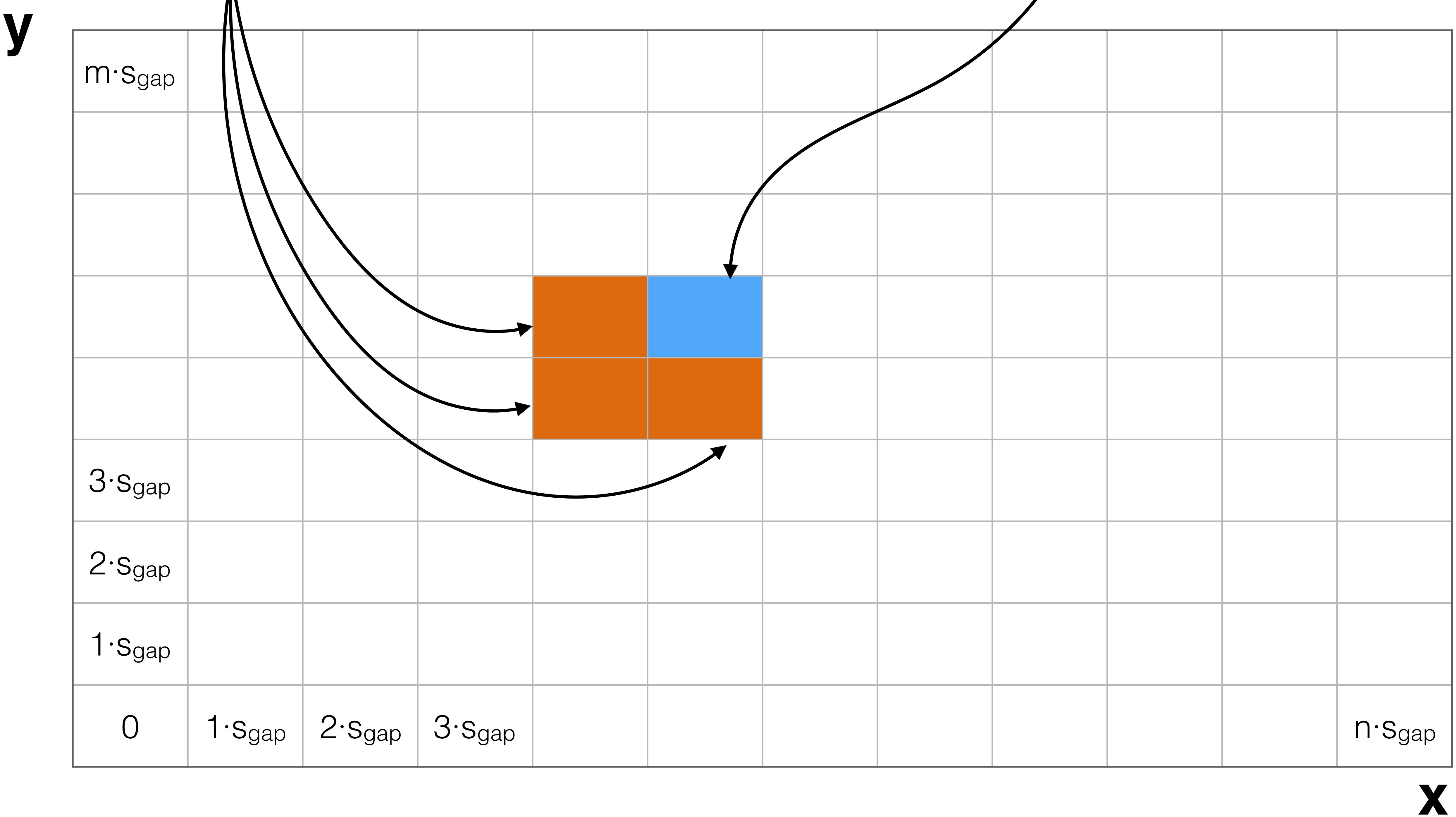
What scores to I need to know to fill in the answer here?



Warmup — optimal *score* in linear space

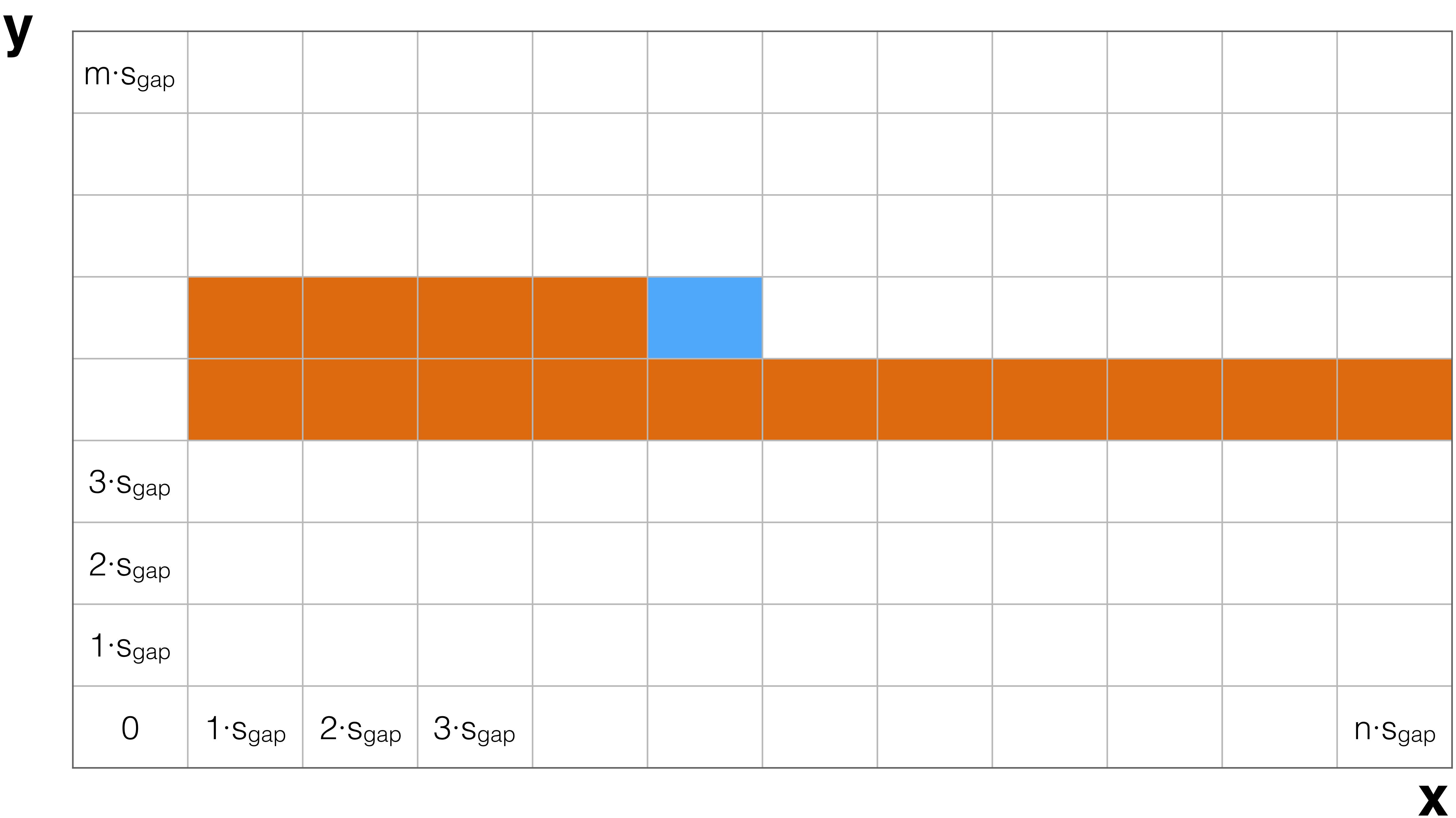
What scores to I need to know to fill in the answer here?

These



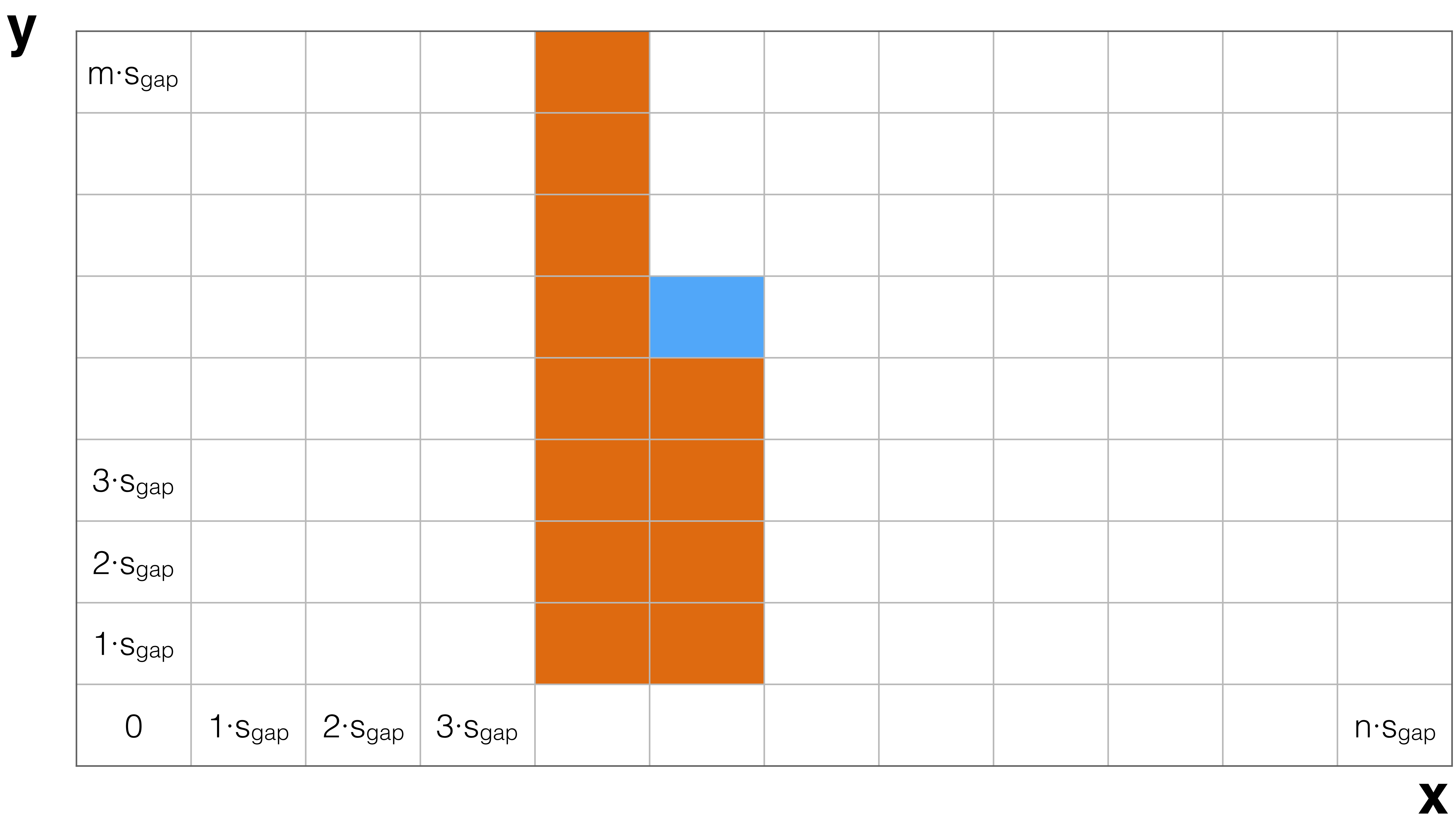
Warmup — optimal *score* in linear space

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



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.



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

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

x

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

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

x

Re-using subproblems

Consider, again, the meaning of the DP matrix

What is contained in the highlighted column?

y

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

x

Re-using subproblems

Consider, again, the meaning of the DP matrix

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

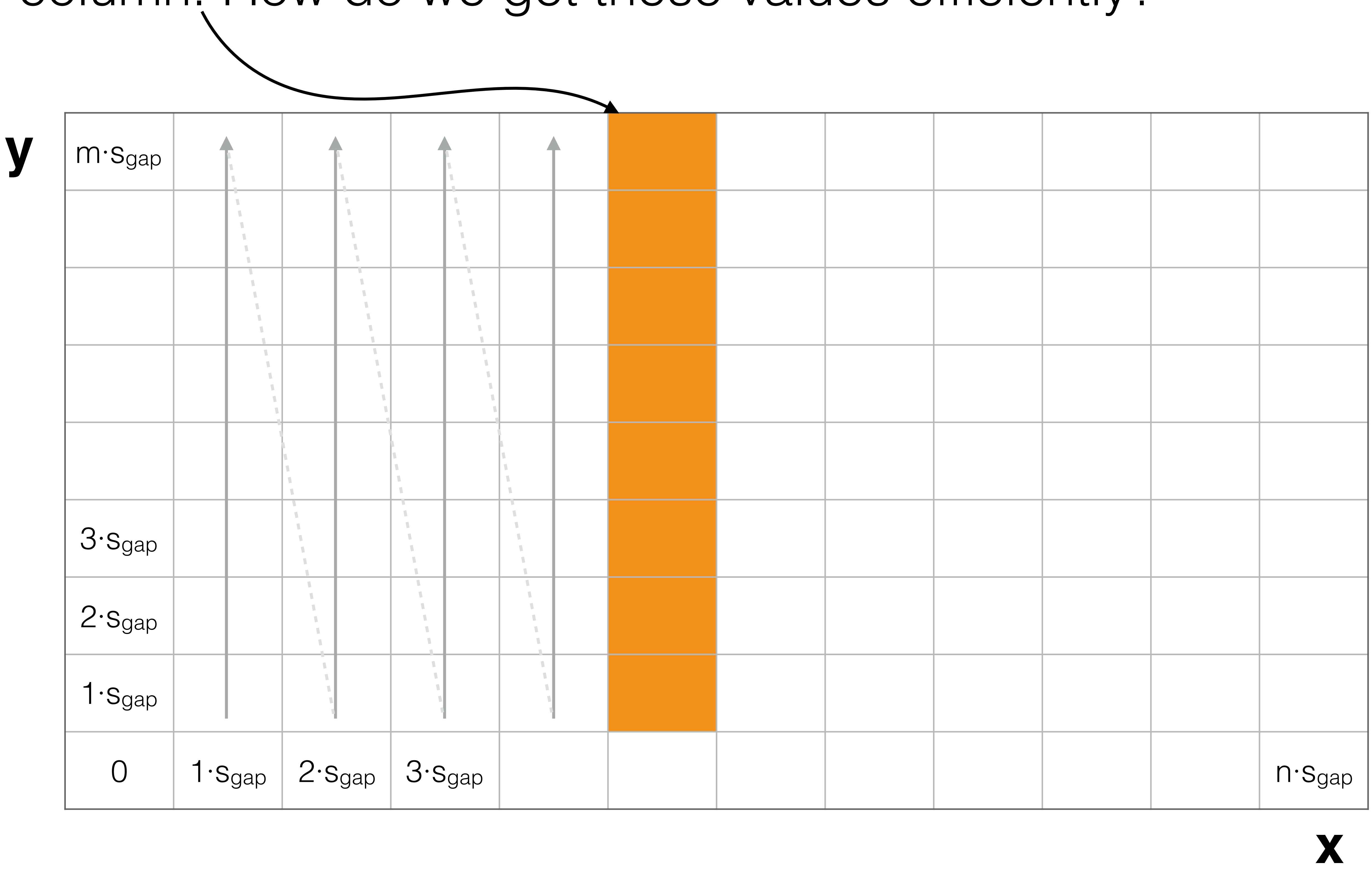
y

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

x

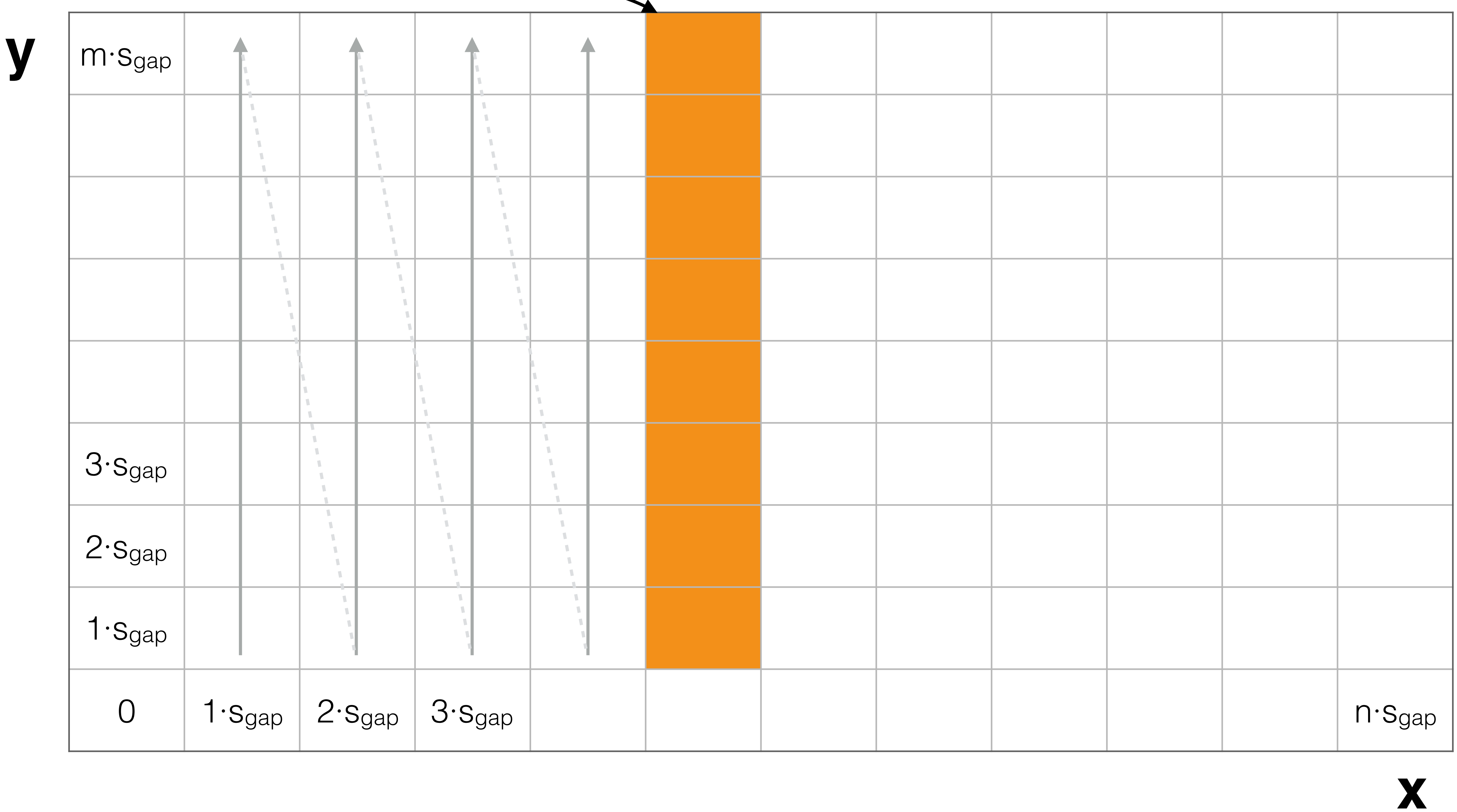
Re-using subproblems

score of *every* prefix of **y** against ith prefix of **x** in the ith column. How do we get these values efficiently?



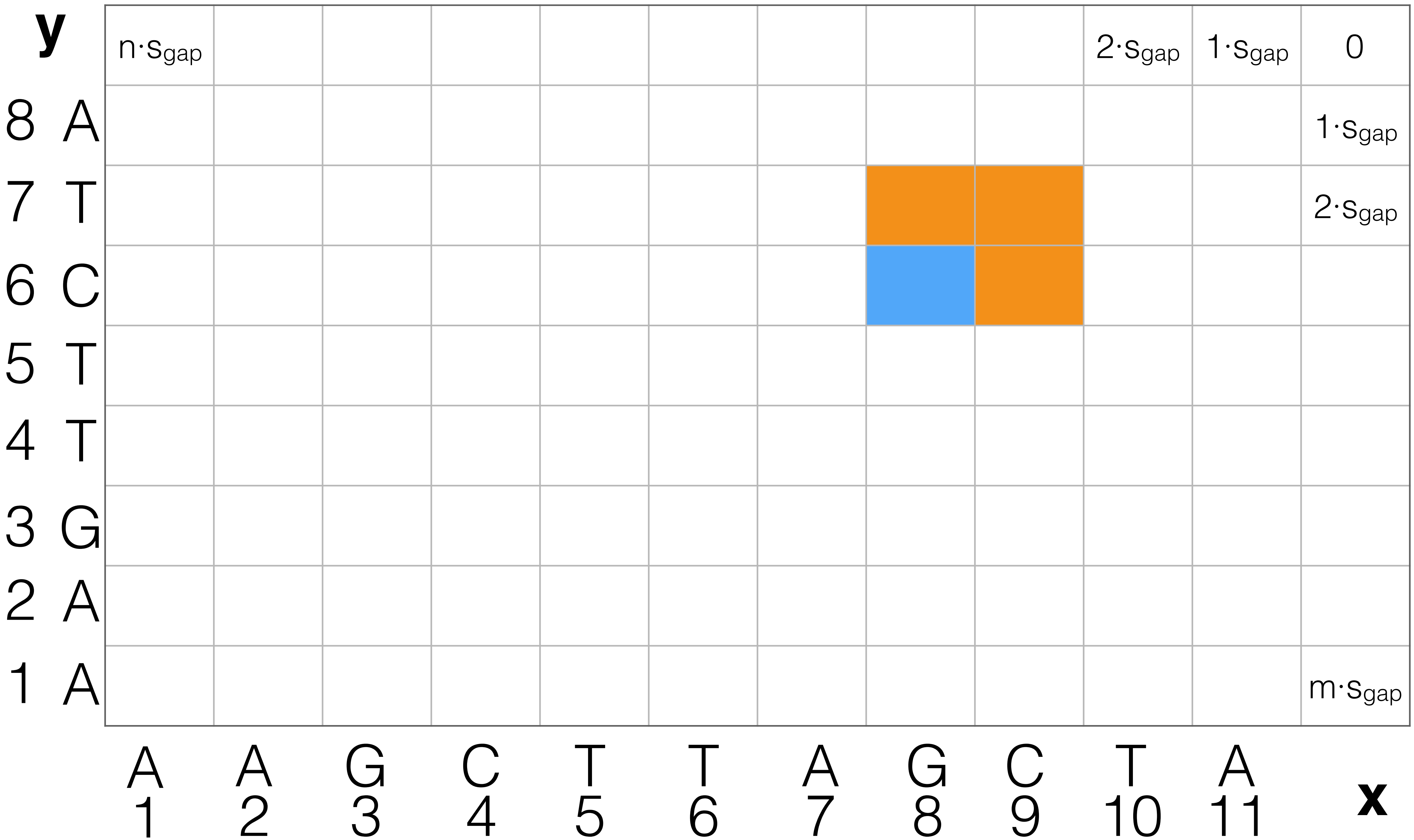
Re-using subproblems

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



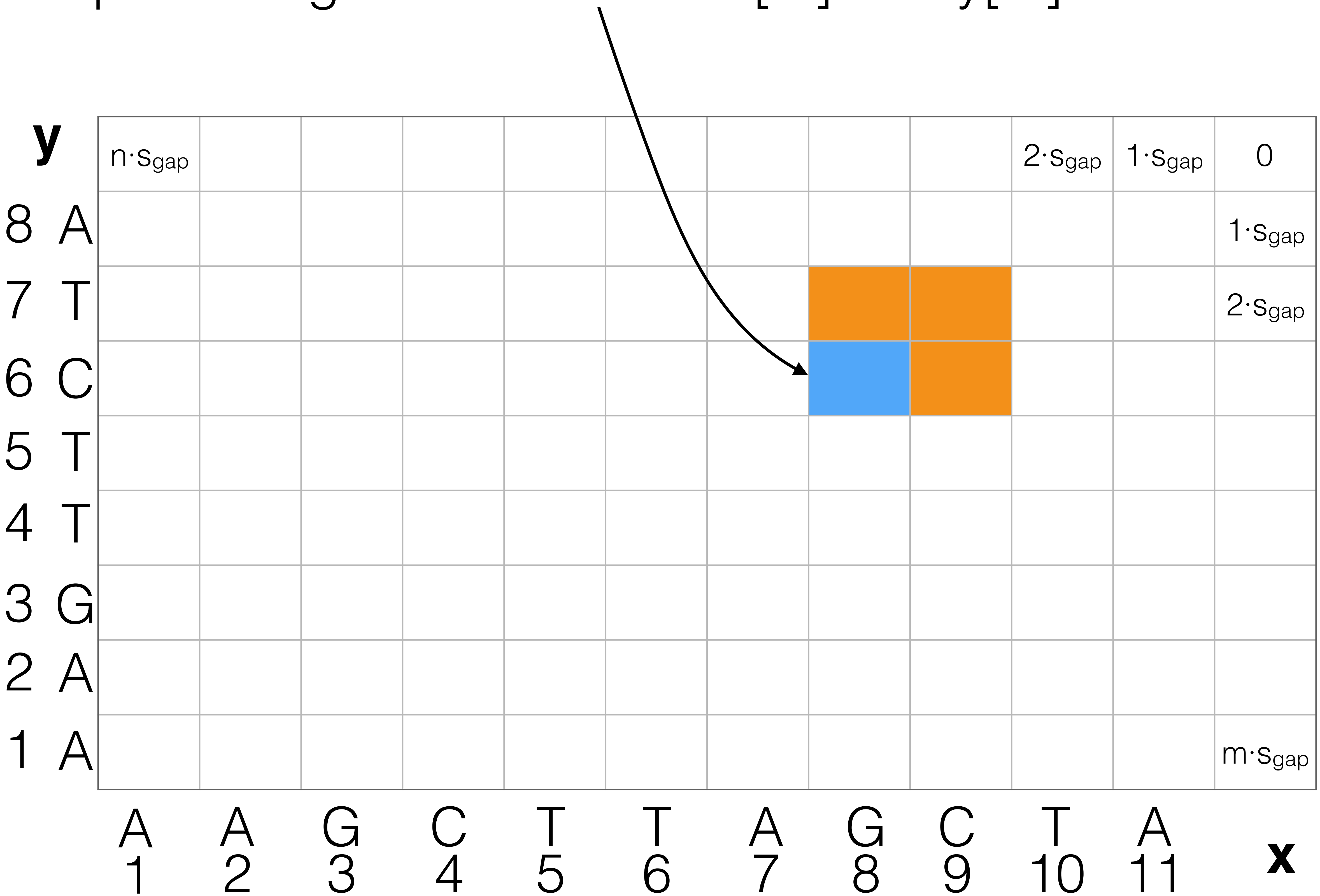
What about suffixes?

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



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**

y												
	$n \cdot s_{gap}$									$2 \cdot s_{gap}$	$1 \cdot s_{gap}$	0
8 A												$1 \cdot s_{gap}$
7 T												$2 \cdot s_{gap}$
6 C												
5 T												
4 T												
3 G												
2 A												
1 A												$m \cdot s_{gap}$
	A 1	A 2	G 3	C 4	T 5	T 6	A 7	G 8	C 9	T 10	A 11	**x**

What about suffixes?

Prefixes (forward):

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

Suffixes (backward):

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

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?

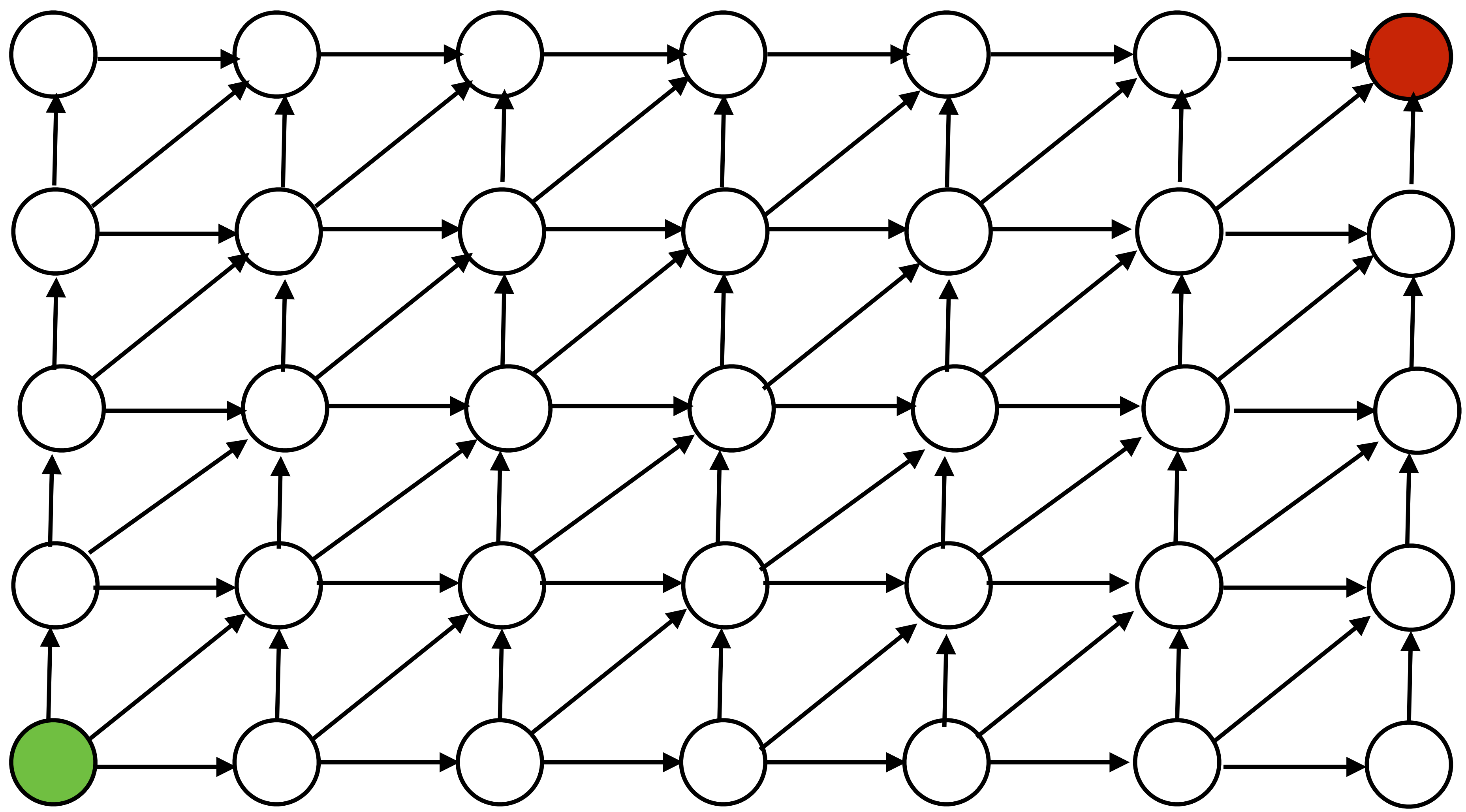
Algorithmic idea: Combine both dynamic programs using *divide-and-conquer*

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

Examples: MergeSort & Karatsuba multiplication

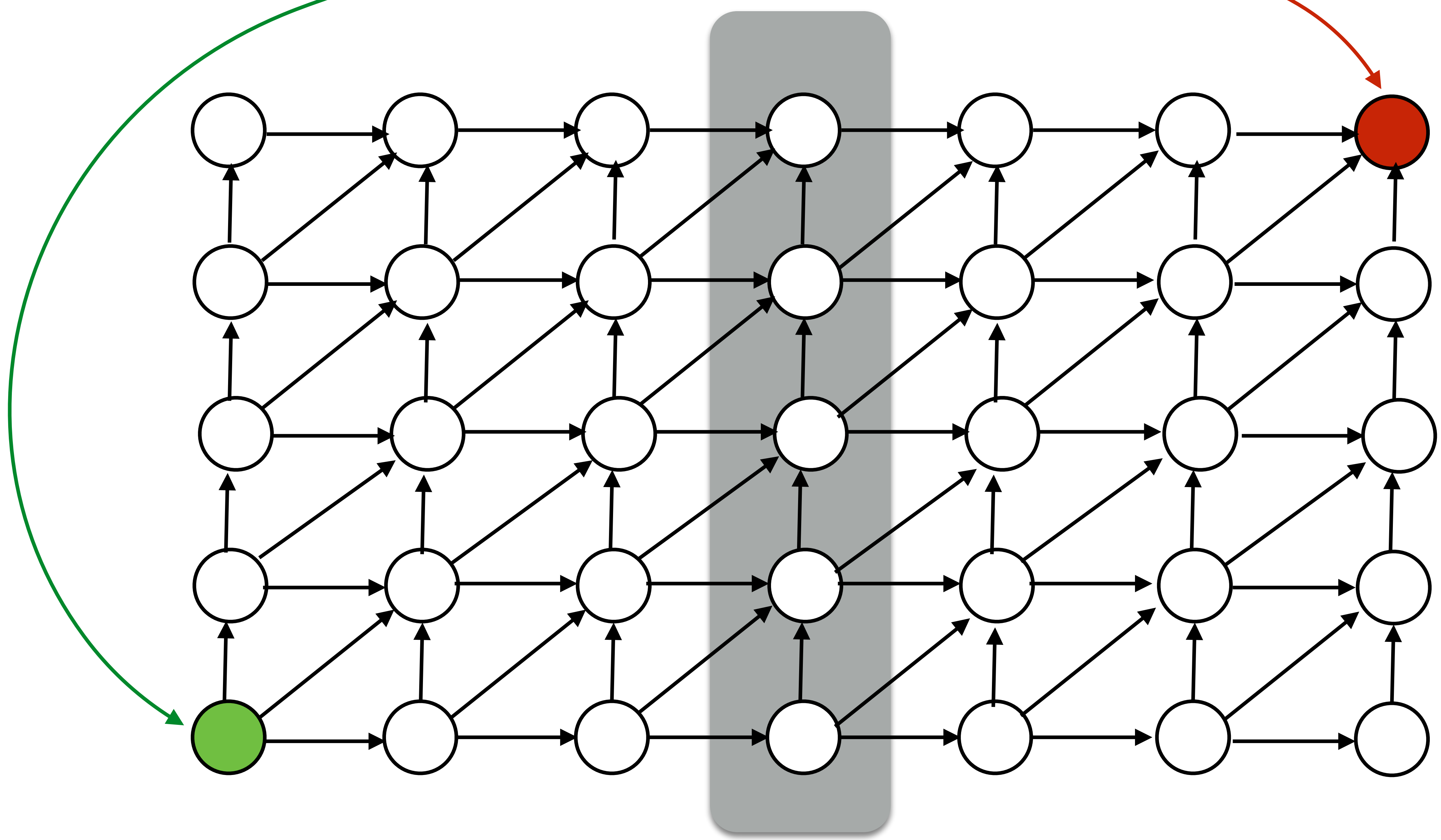
Think about this in “graph” land

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



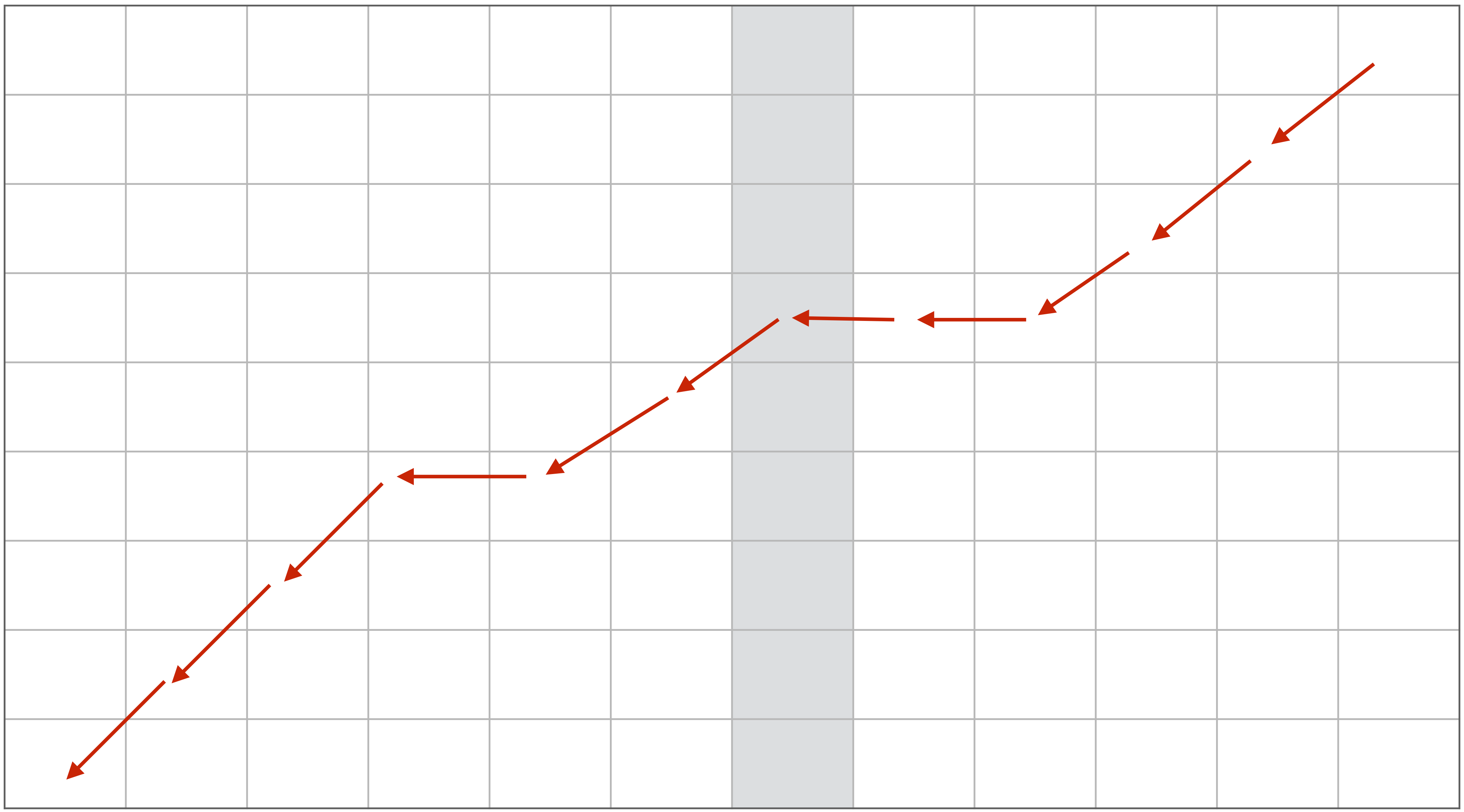
Think about this in "graph" land

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



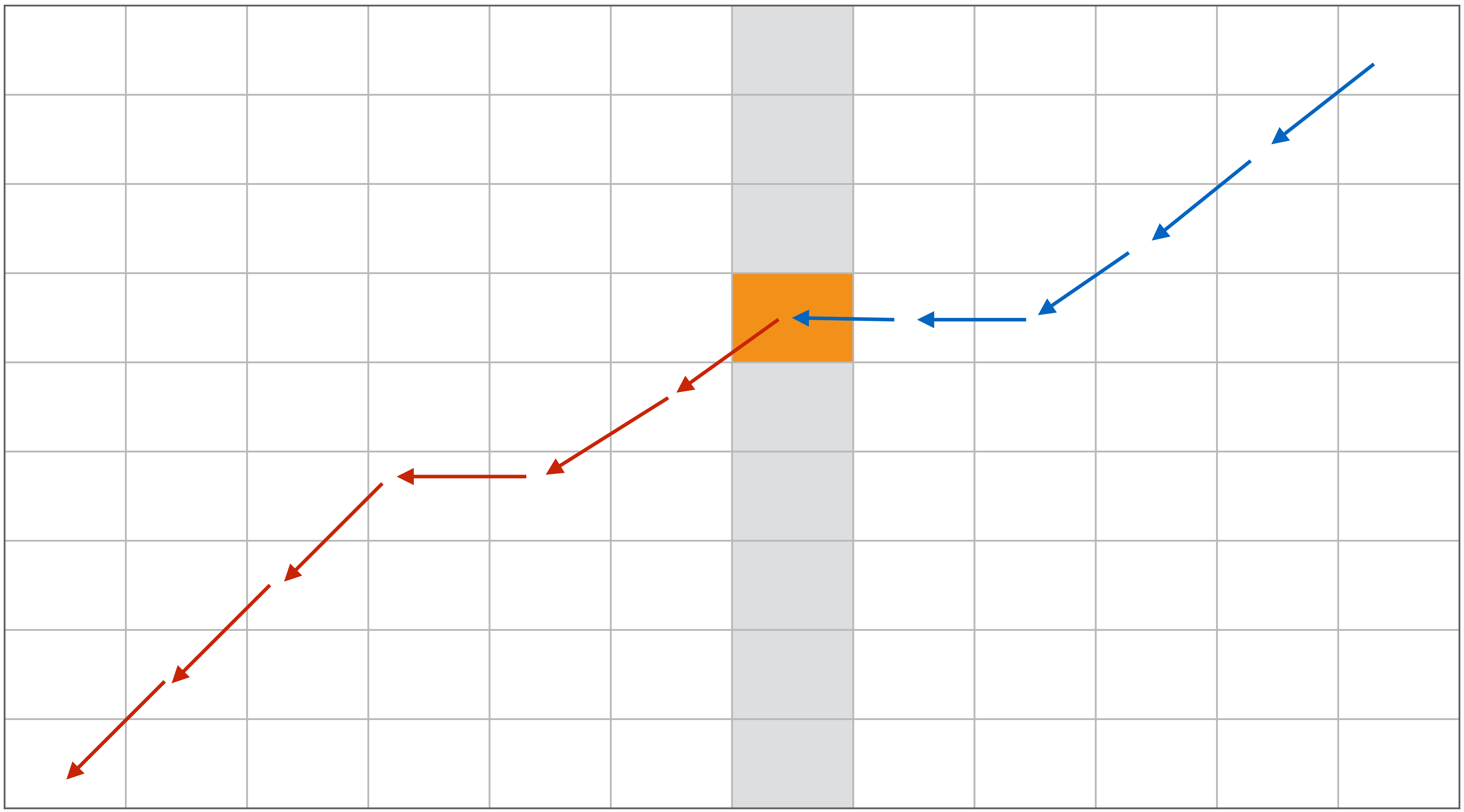
Finding the optimal alignment

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



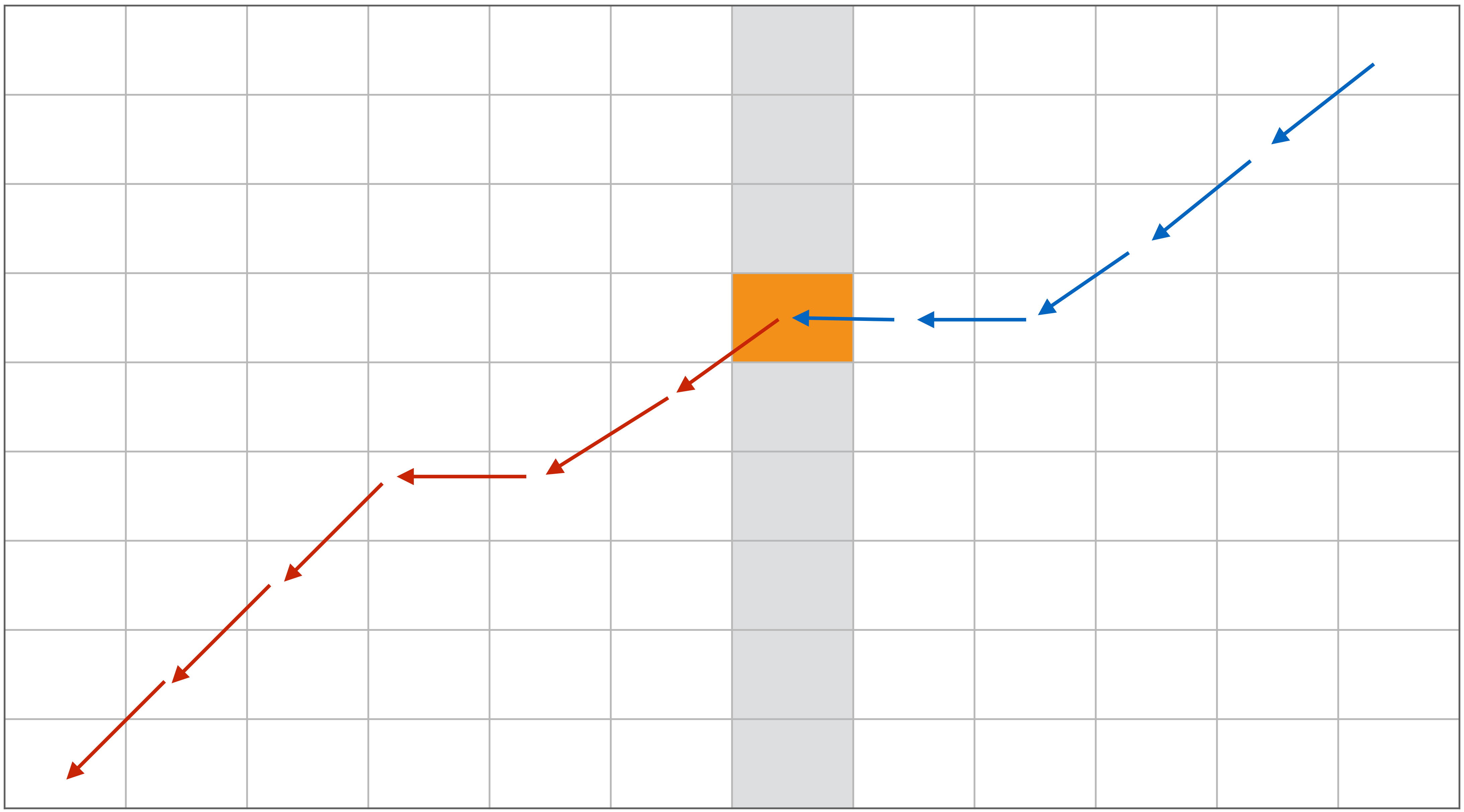
Finding the optimal alignment

It uses the cell (i,j) such that OPT[i,j] + 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.



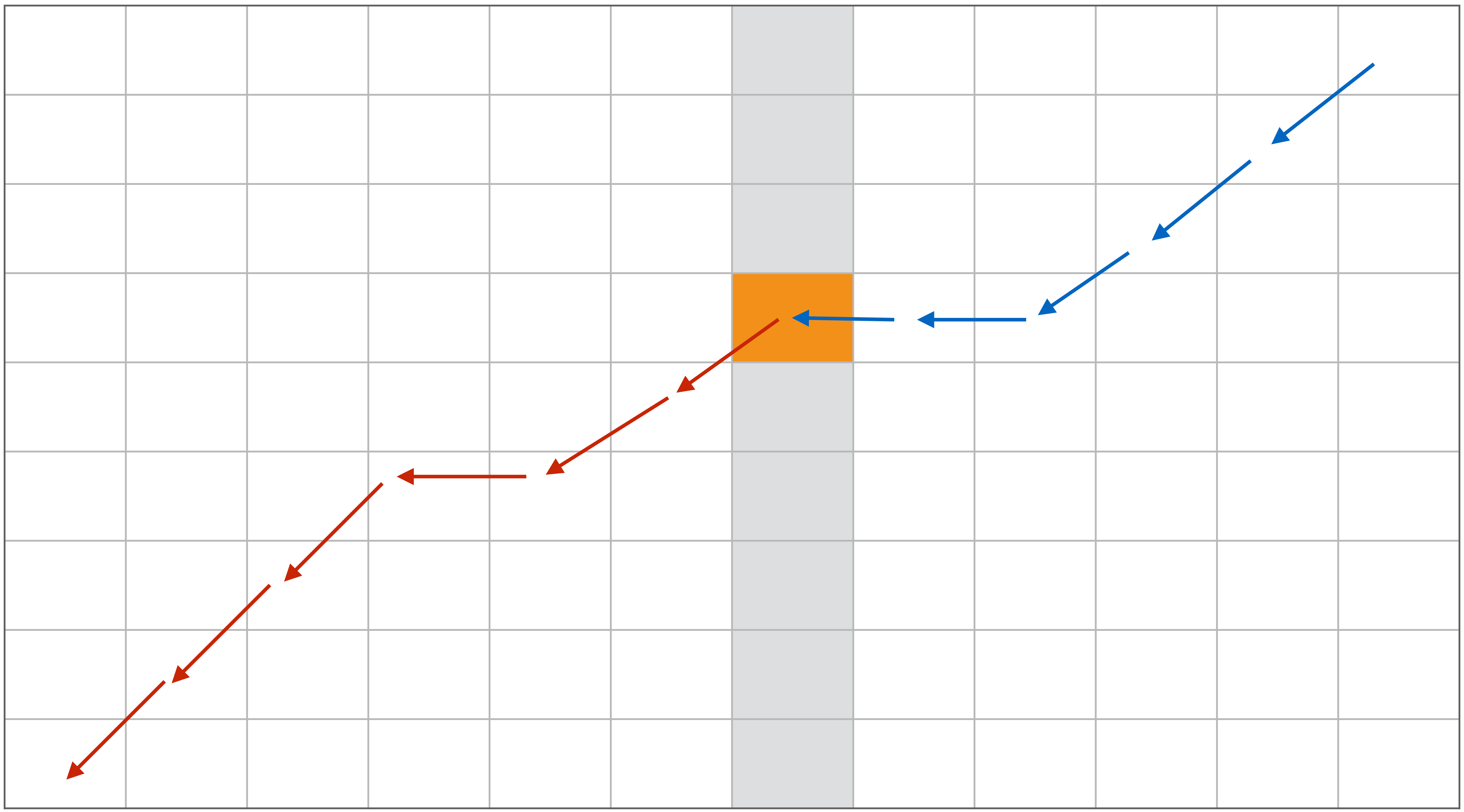
Finding the optimal alignment

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

```
DCAlignment(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
    DCAlignment(x[1:n/2], y[1:q])
    DCAlignment(x[n/2+1:n], y[q+1:m])
    return P
```

Adopted from “Algorithm Design” Kleinberg & Tardos (Ch. 6.7 pg 288)

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:

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

`DCAlignment(x[1:n/2], y[1:q])` `DCAlignment(x[n/2+1:n], y[q+1:m])`

with base cases:

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

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

Adopted from "Algorithm Design" Kleinberg & Tardos (Ch. 6.7 pg 289 — 290)

D&C Alignment

Base:

$T(n,2) \leq cn$

$T(2,m) \leq cm$

Inductive:

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

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 $O(nm)$

Adopted from "Algorithm Design" Kleinberg & Tardos (Ch. 6.7 pg 289 — 290)

Smarter Induction

Base:

$T(n,2) \leq cn$

$T(2,m) \leq cm$

Inductive:

$T(n,m) \leq knm$

Proof:

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

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

Adopted from “Algorithm Design” Kleinberg & Tardos (Ch. 6.7 pg 289 — 290)

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.

Recap

- General gap penalties require 3 matrices and $O(n^3)$ time.
- Affine gap penalties require 3 matrices, but only $O(n^2)$ time.
- Sub-quadratic time general alignment is likely not possible.
- Linear space alignment can be obtained at no asymptotic cost to runtime.

*