CSE 397-497:
Computational Issues in Molecular Biology

Lecture 4

Spring 2004

Student lectures: final vote totals

Topic	1st	2nd	3rd	Total
sequence comparison & alignment	5	2	0	7
sequencing & assembly	2	4	1	7
physical mapping of DNA	0	3	6	9
phylogenetic trees	1	2	2	5
genome rearrangements	2	1	0	3
RNA & protein structure	1	0	1	2
DNA microarrays	0	2	3	5
DNA computing	4	1	2	7

Student lectures: schedule is online

See schedule posted on Blackboard (to be updated periodically).

Date	Lecture Topics	Meetings
Tu 2/3	sequence comparison & alignment – Dan Lopresti	AL1
Th 2/5	sequence comparison & alignment – Dan Lopresti	JW1
Tu 2/10	TBD – Dan Lopresti	AL2, LN1
Th 2/12	TBD – Dan Lopresti	JW2, UM1
M 2/16		AL3
Tu 2/17	sequence comparison & alignment – Arthur Loder	LN2, ST1
W 2/18		JW3
Th 2/19	sequence comparison & alignment – Jesse Wolfgang	UM2, SC1
M 2/23		LN3

Interpretation: *UP1* = Upmanyu's first meeting with me (2/12).

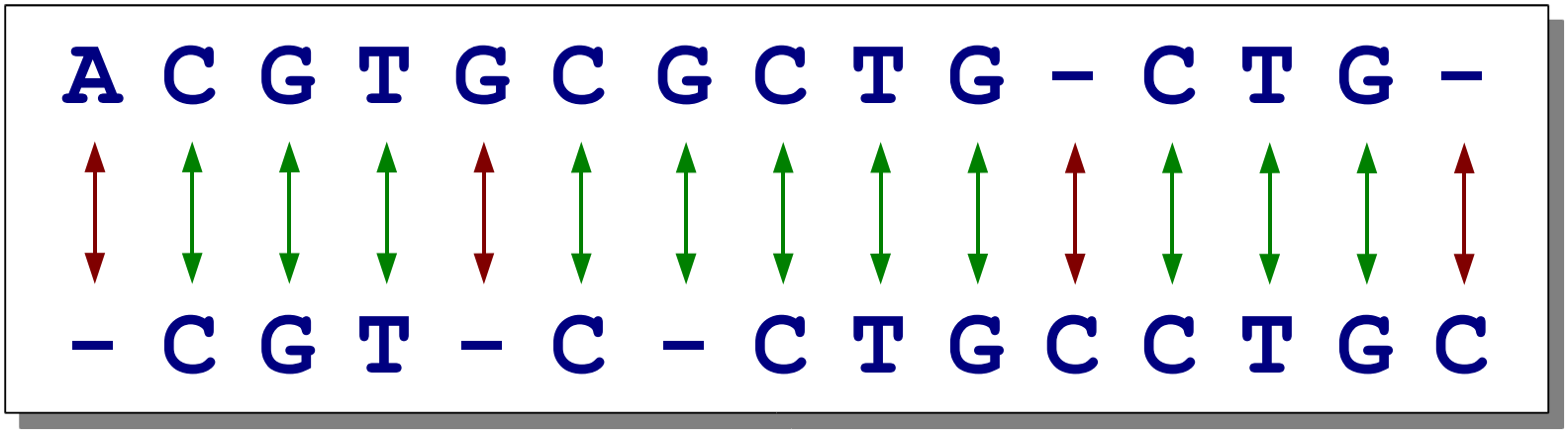
LN2 = Lan's second meeting with me (2/17).

AL3 = Arthur's third meeting with me (2/16).

Sequence comparison and alignment

How do these two sequences relate?

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

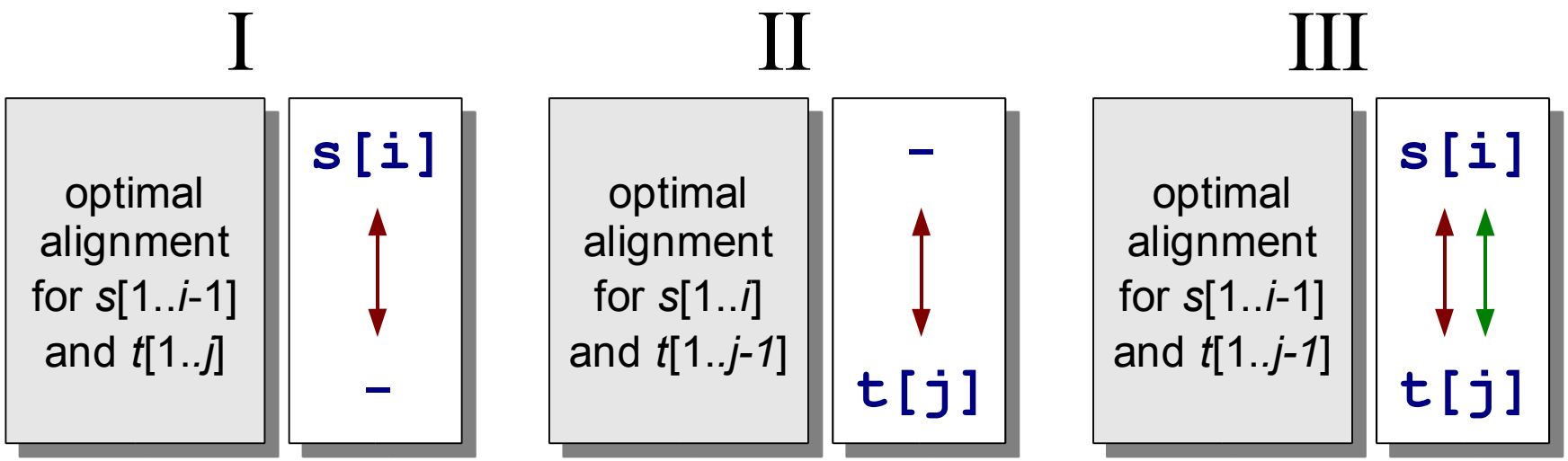


If matches = +1 and mismatches = -1, then score = 7 (= 11-4).

How can we find the best (i.e., highest scoring) alignment?

Observation leading to a solution

Given two sequences *s* and *t*, consider what's required to compute optimal alignment for prefixes *s*[1..*i*] and *t*[1..*j*]. Based on our rules for alignments, there are three possible cases:



So, assuming we've already computed solutions for all shorter prefixes, we can compute the alignment for *s*[1..*i*] and *t*[1..*j*].

Sequence comparison: the basic algorithm

Stated more generally, say that our two sequences are:

$$s[i]s[2]s[3]\ldots s[m] \qquad t[i]t[2]t[3]\ldots t[n]$$

and c_{del}, c_{ins}, and c_{sub} are costs of a deletion, an insertion, and a substitution, respectively.

Then:

$$a[0,0]=0$$
$$a[i,0]=a[i-1,0]+c_{del}(s[i]) \quad 1\le i\le m$$
$$a[0,j]=a[0,j-1]+c_{ins}(t[j]) \quad 1\le j\le n$$

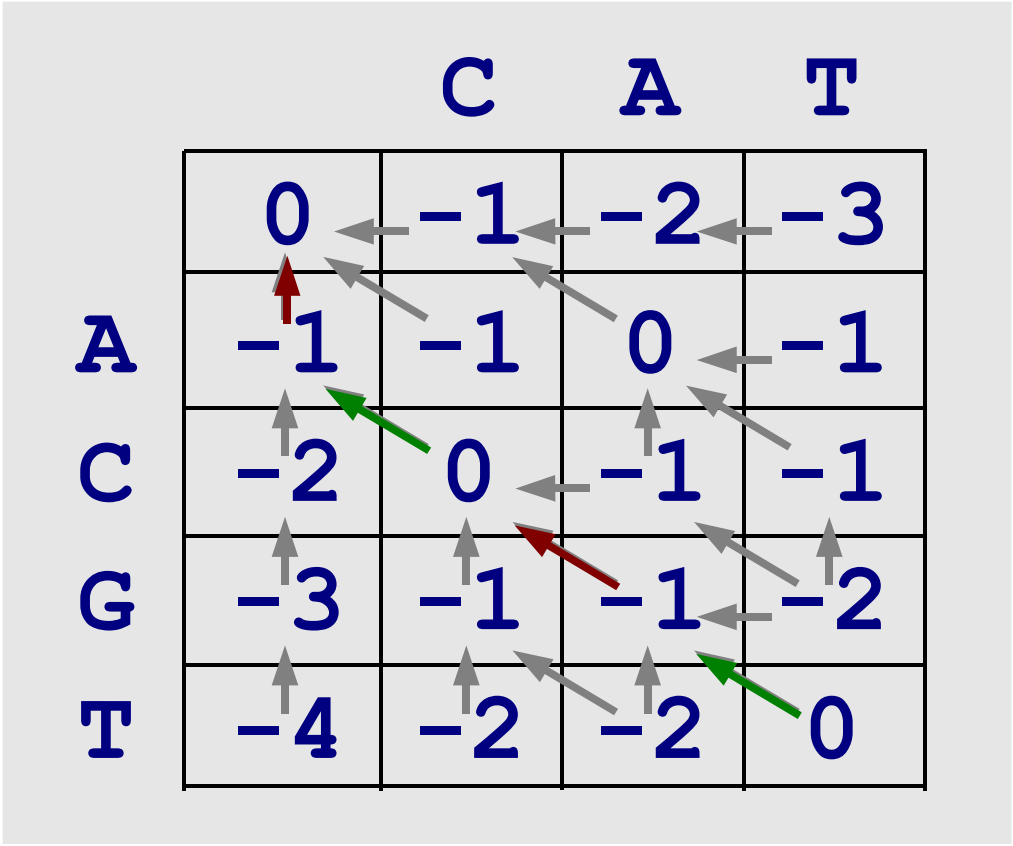
And:

$$a[i,j] = \max\begin{cases} a[i-1,j]+c_{\text{del}}\,s([i]) \\ a[i,j-1]+c_{\text{ins}}(t[j]) \\ a[i-1,j-1]+c_{\text{sub}}(s[i],t[j]) \end{cases} \quad 1\le i\le m, 1\le j\le n$$

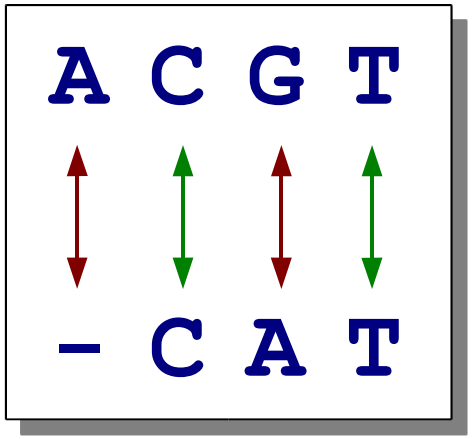
Sequence comparison: tracing back for alignment

We started with the notion of alignment. How do we get this?

By keeping track of optimal decisions made during algorithm:



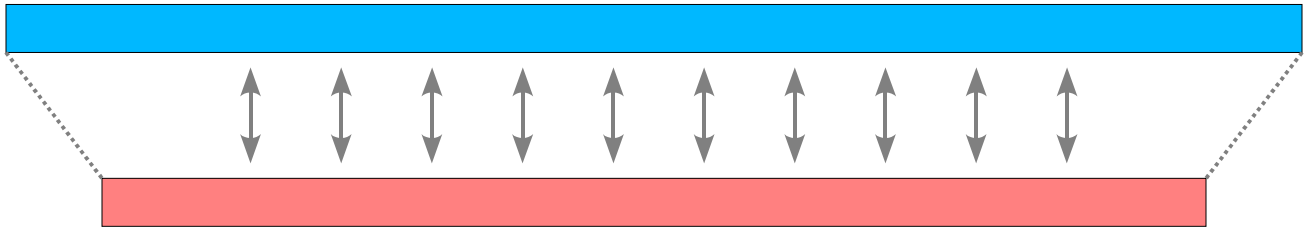
... and then tracing back optimal path:



It may not be unique!

Keeping track of the variations

Algorithm 1 Determines score and optimal global alignment.
Requires time $O(mn)$, space $O(mn)$.



This is often referred to as *global sequence comparison*.

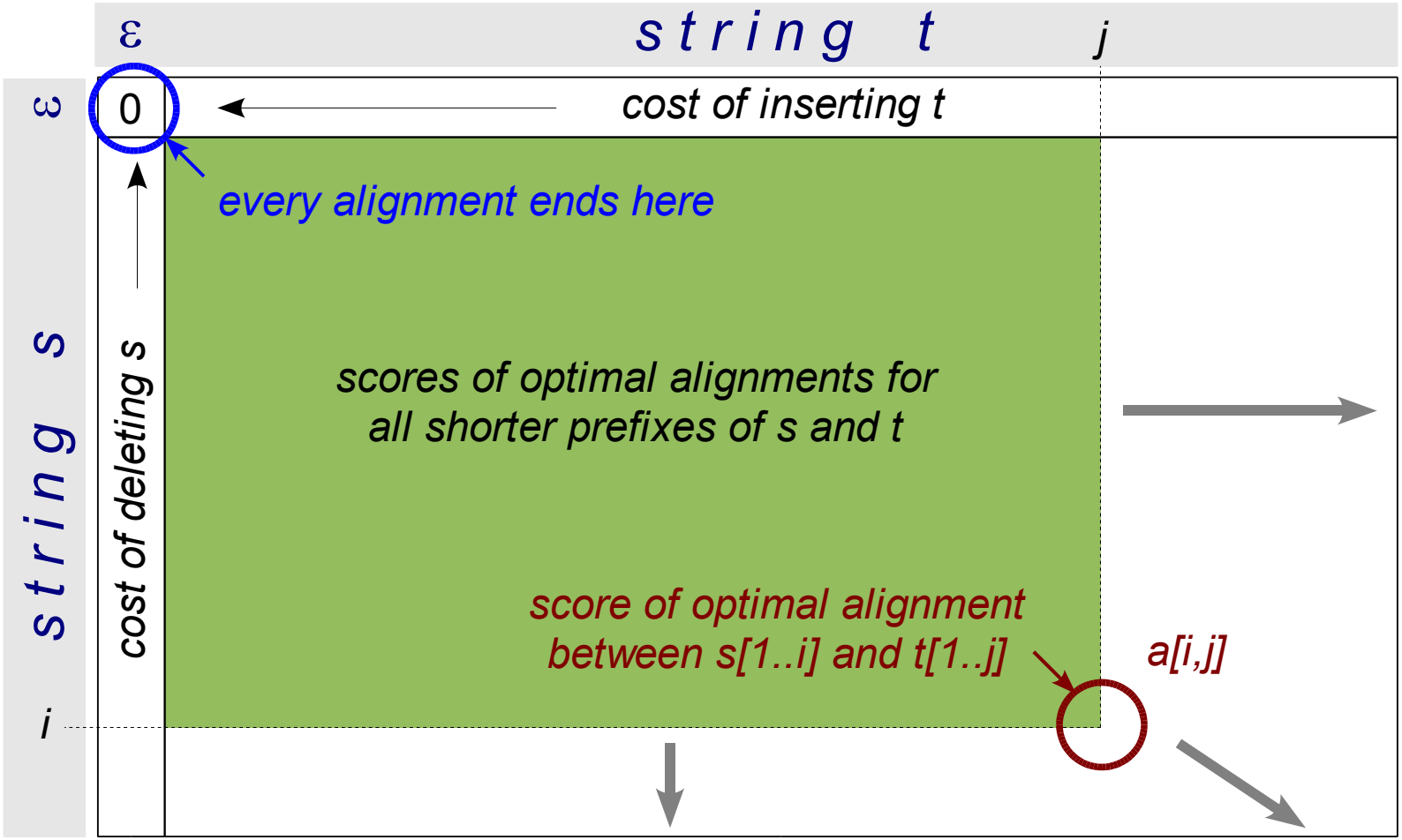
Early literature:

"Binary codes capable of correcting deletions, insertions and reversals," V. Levenshtein, *Soviet Physics Daklady*, 10:707-710, 1966.
(So far as I know, this is only available in Russian.)

"A general method applicable to the search for similarities in the amino acid sequences of two proteins," S. B. Needleman and C. D. Wunsch, *Journal of Molecular Biology*, 48:443-453, 1970.

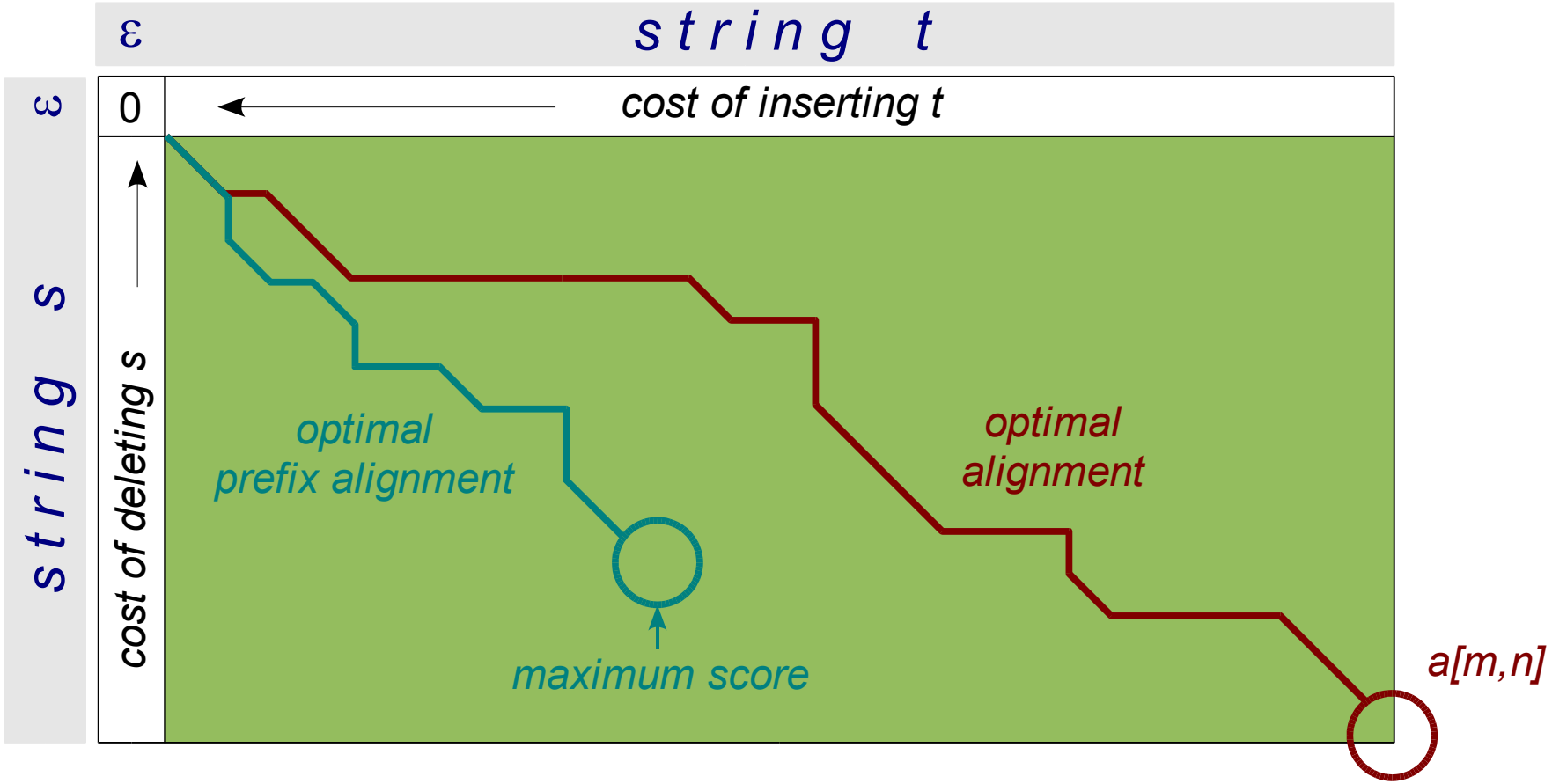
"The string to string correction problem," R. A. Wagner and M. J. Fisher, *Journal of the Association for Computing Machinery*, 21(1):168-173, 1974.

A prefix view of the world



Note that the dynamic programming matrix contains data on all prefix alignments – this will come in handy later.

A prefix view of the world

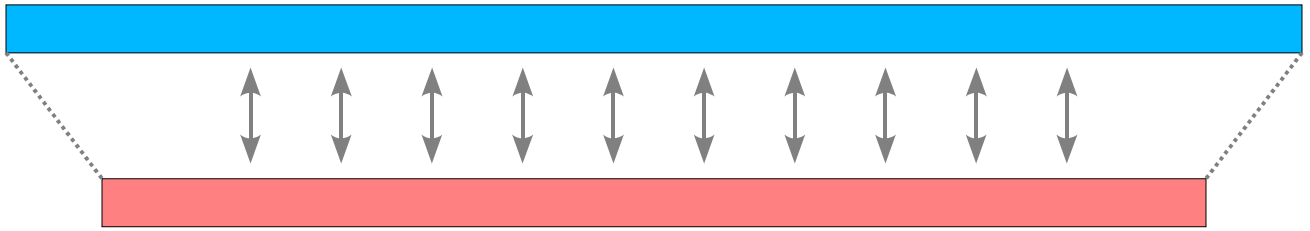


a[m,n] = start of optimal alignment between all of *s* and *t*.

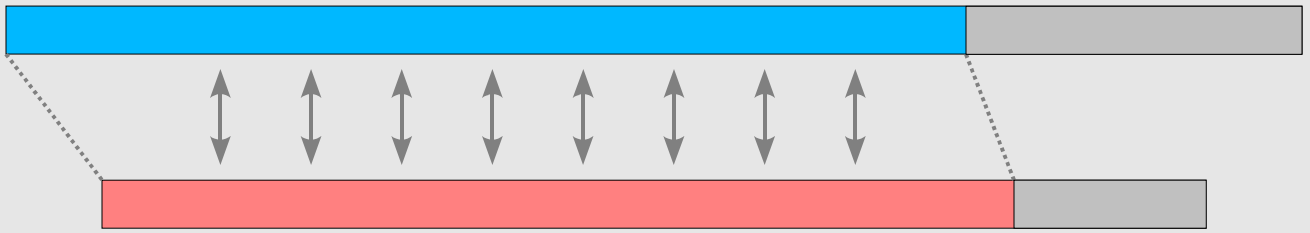
Searching matrix for largest value yields optimal alignment between a *prefix* of *s* and a *prefix* of *t*.

Keeping track of the variations

Algorithm 1 Determines score and optimal global alignment.
Requires time $O(mn)$, space $O(mn)$.

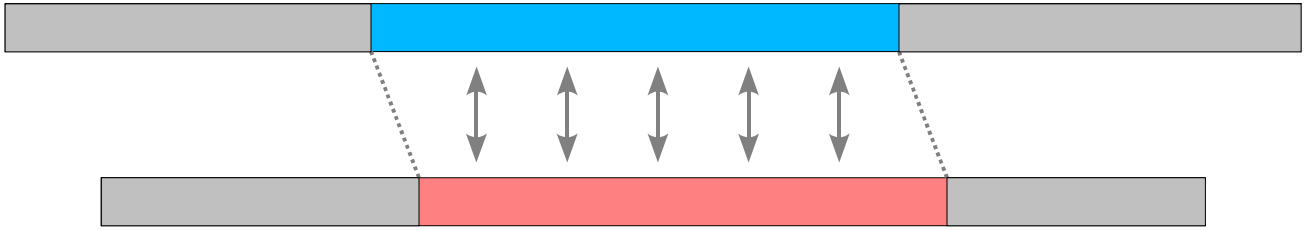


Algorithm 2 Algorithm 1 + search for largest score.
Determines score and optimal prefix alignment.
Requires time $O(mn)$, space $O(mn)$.



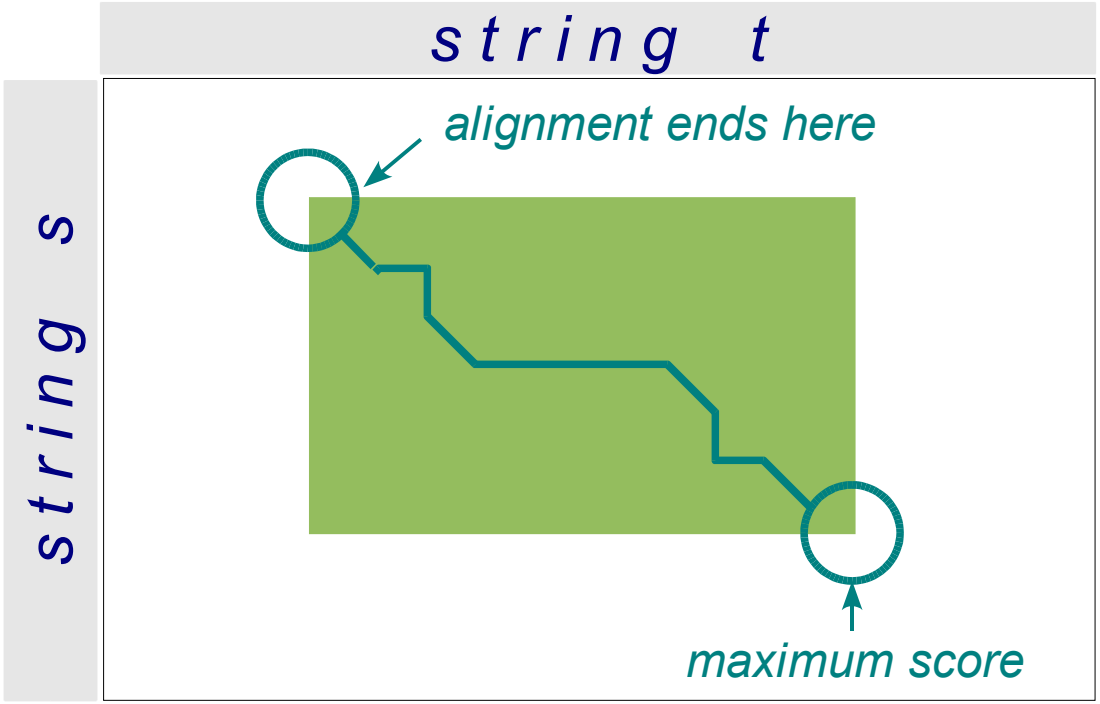
Local comparison

There are times when the best substring match is desired:



Allow alignment to start and end at any point.

View: want best *suffix* for a given prefix.



Local comparison

Make a couple straightforward modifications to basic algorithm:

Initial conditions:

$$a[0,0] = 0$$
$$a[i,0] = 0 \quad 1 \le i \le m$$
$$a[0,j] = 0 \quad 1 \le j \le n$$

allow alignment to ignore unmatched prefix of s or t (initial deletions or insertions don't contribute to best match)

Recurrence:

$$a[i,j] = \max \begin{cases} a[i-1,j] + c_{\text{del}}\, s([i]) \\ a[i,j-1] + c_{\text{ins}}(t[j]) \\ a[i-1,j-1] + c_{\text{sub}}(s[i], t[j]) \\ 0 \end{cases} \qquad 1 \le i \le m, 1 \le j \le n$$

allow alignment to ignore prefixes if not good match

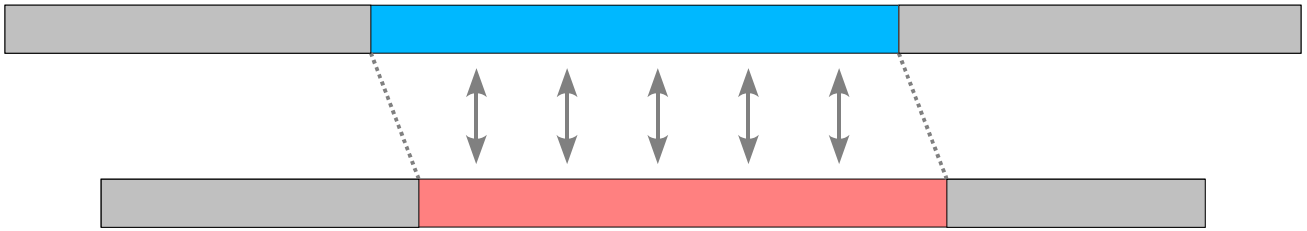
Finally, we search resulting matrix for largest score.

Local comparison

Algorithm 3 Algorithm 1 + new initial conditions + new choice in recurrence + search for largest score.

Determines score and optimal local alignment.

Requires time $O(mn)$, space $O(mn)$.

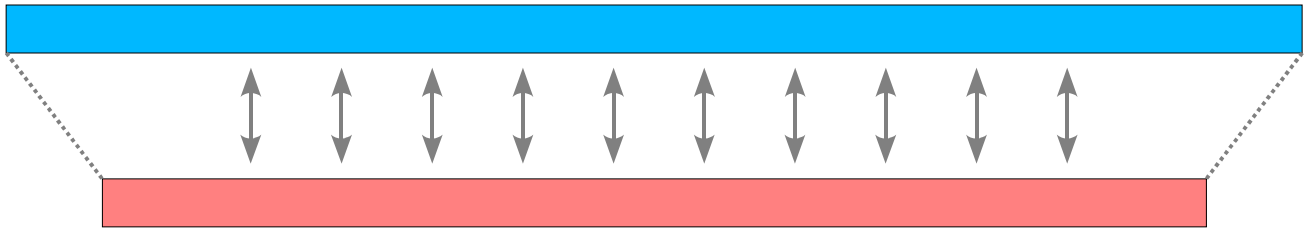


Often, we care not just about the single best local alignment, but other local alignments that are almost as good.

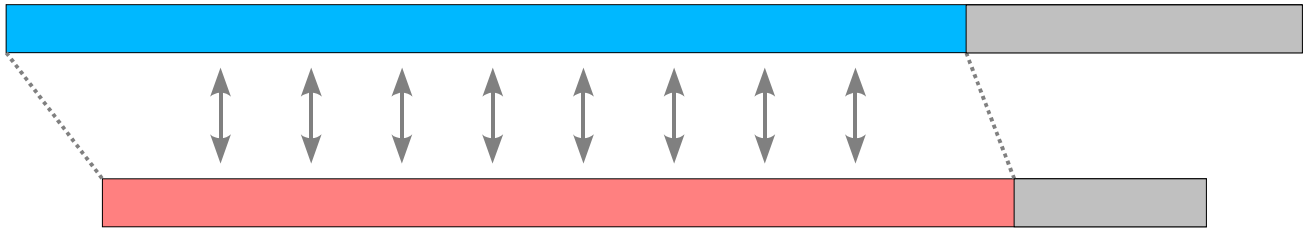
"Identification of common molecular sequences," T. F. Smith and M. S. Waterman, *Journal of Molecular Biology*, 147:195-197, 1981.

Variations: what might we be missing?

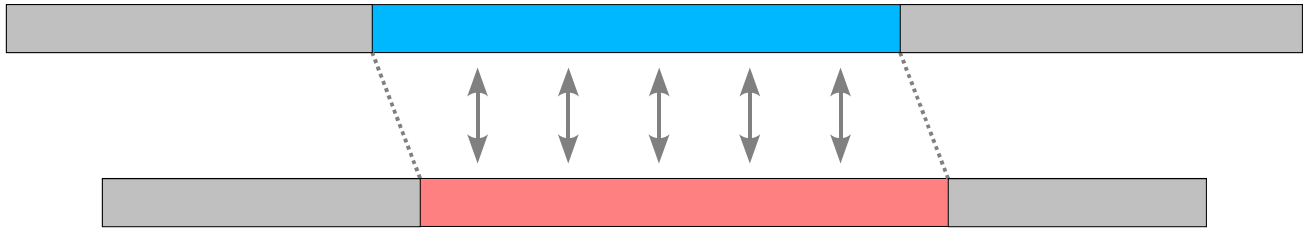
Algorithm 1 (global)



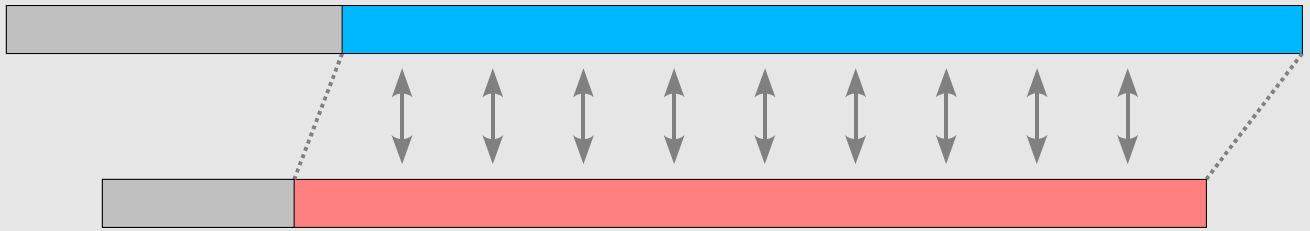
Algorithm 2 (prefix)



Algorithm 3 (local)



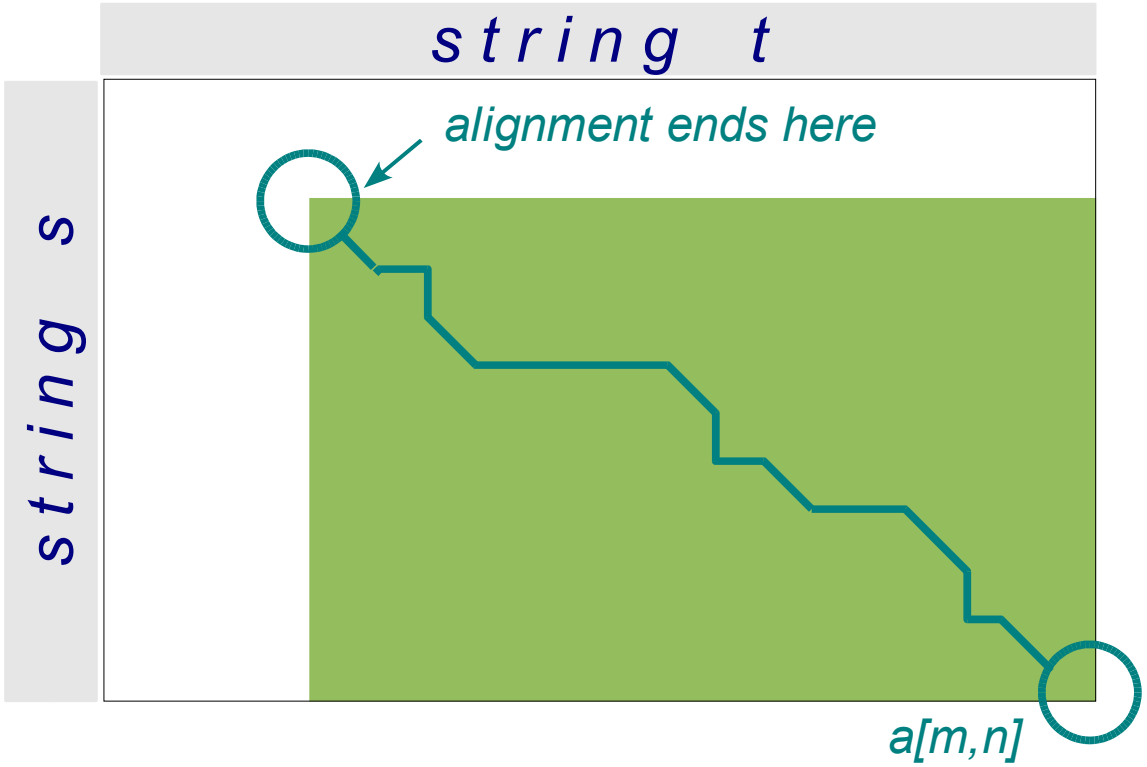
Algorithm 4 (suffix)



A suffix view of the world

As before, want best suffix for given prefix.

Allow alignment to end at any point, but must start at [*m*,*n*].



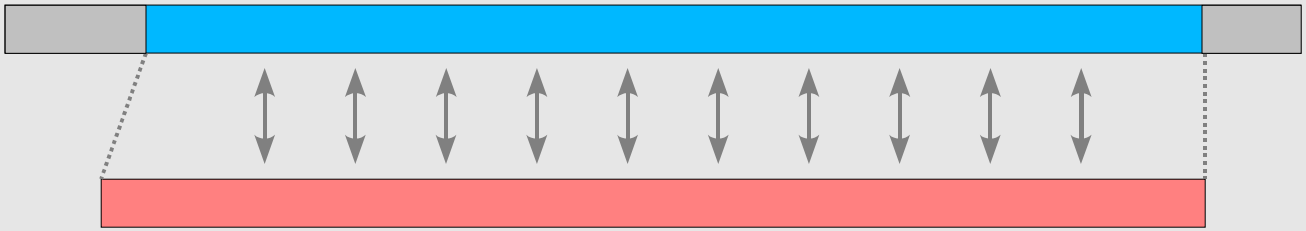
Use same initial conditions and recurrence as Algorithm 3 for local comparison, except don't search entire matrix for maximum score – just use value at [*m*,*n*].

Semiglobal comparison

Note that global comparison algorithm forces both sequences to be aligned in their entireties.

Conversely, the other three algorithms make no such demands on either sequence.

There is a middle ground, however. We could insist one string (or other) be used completely, while other is unconstrained:



(Note: in some disciplines this is called “word spotting.”)

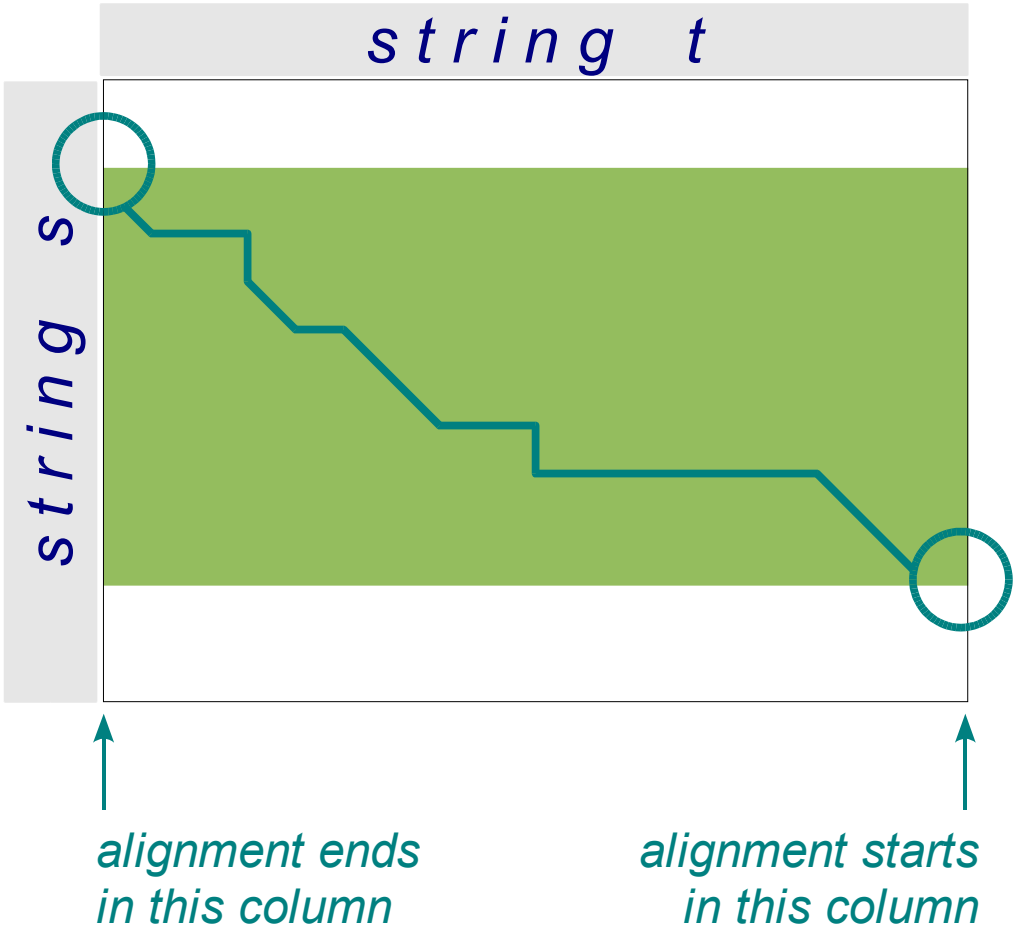
Semiglobal comparison

Say that string *s* is unconstrained while string *t* must be used in entirety (other case is symmetric).

Recurrence is same as Algorithm 1 for global comparison.

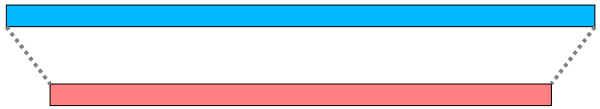
Initialize first column to 0's (allow alignment to end anywhere).

Search last column for max (allow alignment to start anywhere).

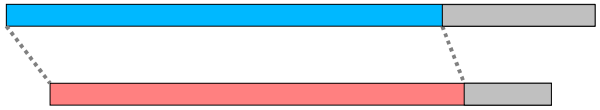


Keeping track of the variations

Algorithm 1 (global)

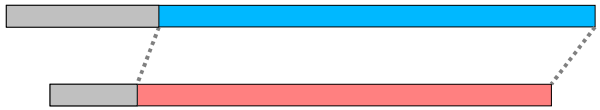


Algorithm 2 (prefix)



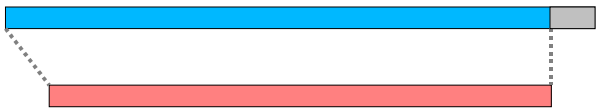
Algorithm 3 (local)

Algorithm 4 (suffix)

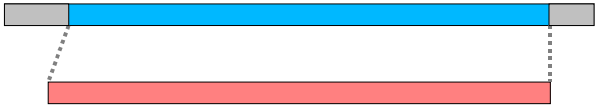


Algorithm 5 (semiglobal 1)

Algorithm 6 (semiglobal 2)



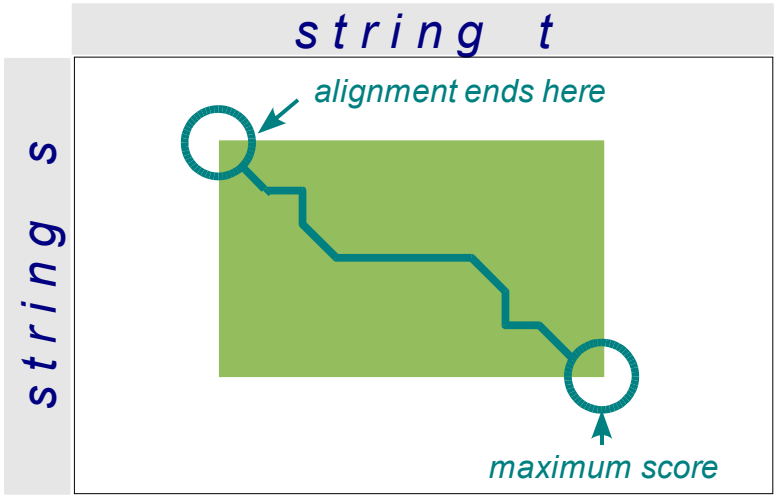
Algorithm 7 (semiglobal 3)



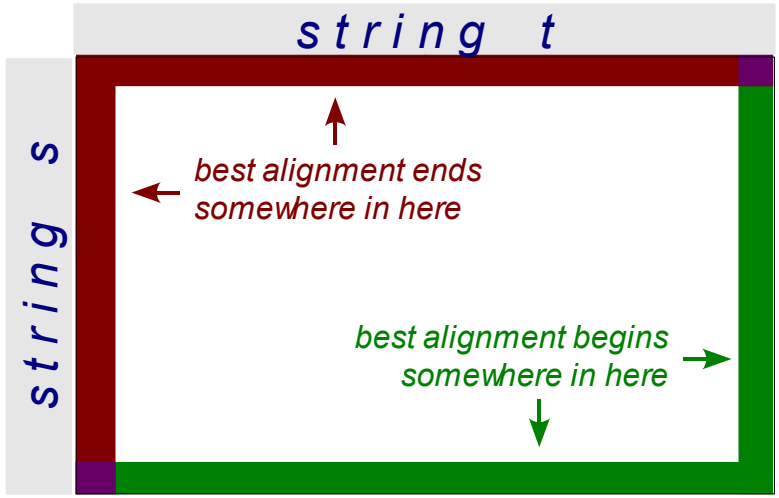
All are variations on same dynamic programming algorithm requiring time and space $O(mn)$.

Potential confusion

What is the difference between local comparison and semiglobal comparison when applied to both sequences?



vs.



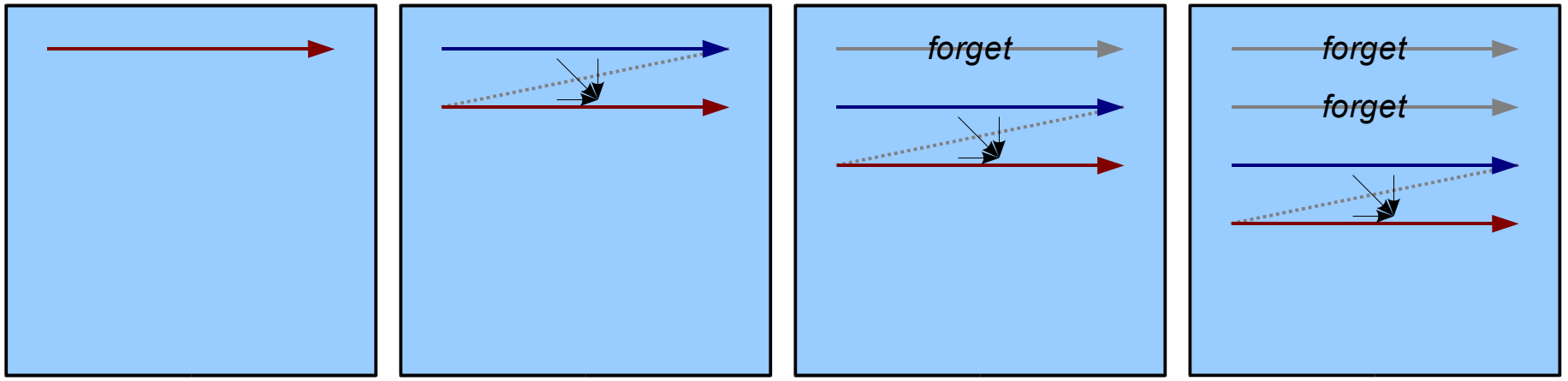
- Former allows ignoring prefixes and suffixes in both strings.
- Latter is equivalent to ignoring one or the other prefix, but not both, and one or the other suffix, but not both.

Saving space

Recall that the initial formulation requires space $O(mn)$.

$$a[i,j] = \max \begin{cases} a[i-1,j] + c_{\text{del}}\, s([i]) \\ a[i,j-1] + c_{\text{ins}}(t[j]) \\ a[i-1,j-1] + c_{\text{sub}}(s[i], t[j]) \end{cases} \qquad 1 \le i \le m, 1 \le j \le n$$

Observe, though, that each cell only depends on neighbors:



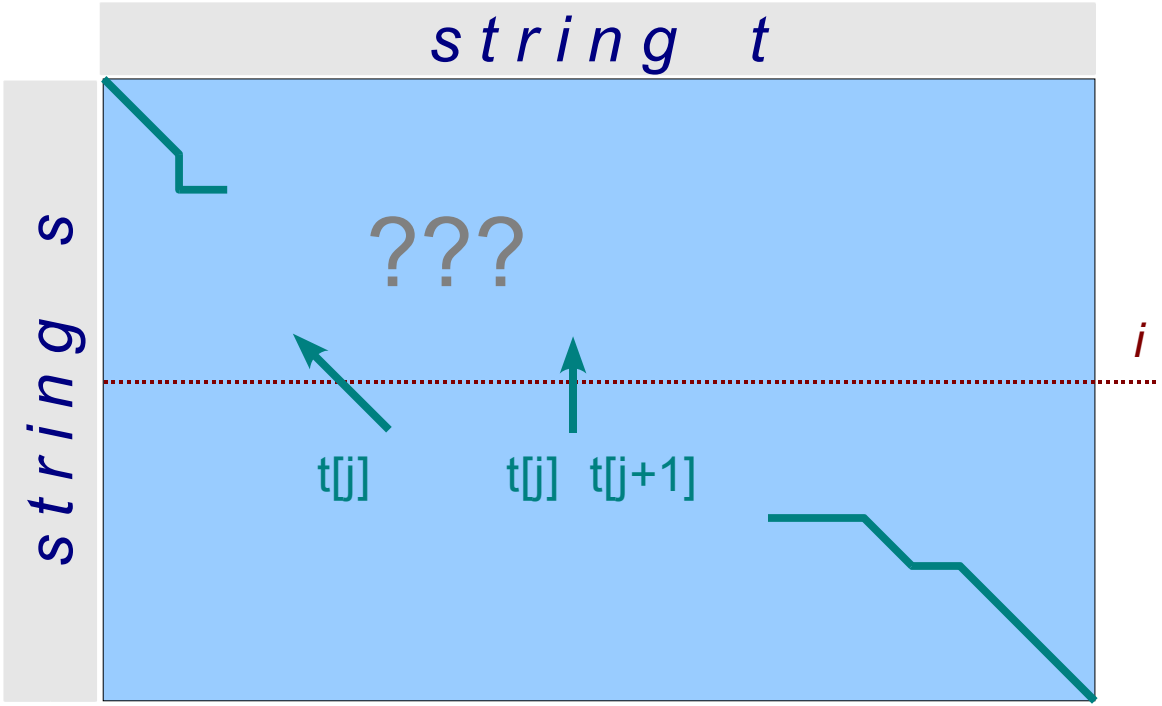
Hence, we can compute optimal score in linear space.

Saving space when computing alignments

Determining optimal alignment in linear space is trickier.

Consider index *i* in string *s*.

We know optimal alignment must cross this line at some point, but we don't know where.



Two possible cases:

(1) $s[i]$ matches $t[j]$ for $1 \leq j \leq n$.

(2) $s[i]$ matches space between $t[j]$ and $t[j+1]$ for $0 \leq j \leq n$.

Saving space when computing alignments

Let's say case (1) applies, so $s[i]$ matches with $t[j]$:

$$optimal\begin{pmatrix} s[1..i-1] \\ t[1..j-1] \end{pmatrix} + \begin{matrix} s[i] \\ t[j] \end{matrix} + optimal\begin{pmatrix} s[i+1..m] \\ t[j+1..n] \end{pmatrix}$$

On other hand, if case (2) applies and $s[i]$ matches a space:

$$optimal\begin{pmatrix} s[1..i-1] \\ t[1..j] \end{pmatrix} + \begin{matrix} s[i] \\ - \end{matrix} + optimal\begin{pmatrix} s[i+1..m] \\ t[j+1..n] \end{pmatrix}$$

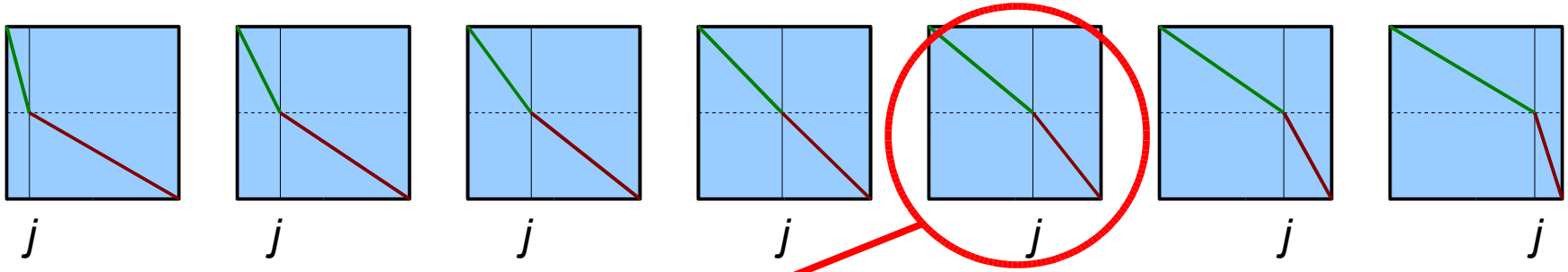
This gives us a recursive algorithm for computing alignment.

For given value of *i*, we already know how to compute similarity between *s*[1..*i*-1] and all prefixes of *t* in linear space. This yields first recursive term.

Analogous approach applied to suffixes of *t* yields second term.

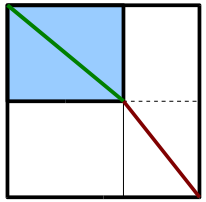
Saving space when computing alignments

As a result of one pair of prefix and suffix computations:

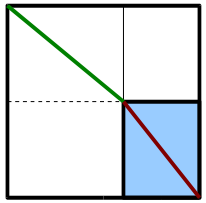


try all possible values for j (and spaces inbetween) to find optimal

choose best value of j and recurse



etc.



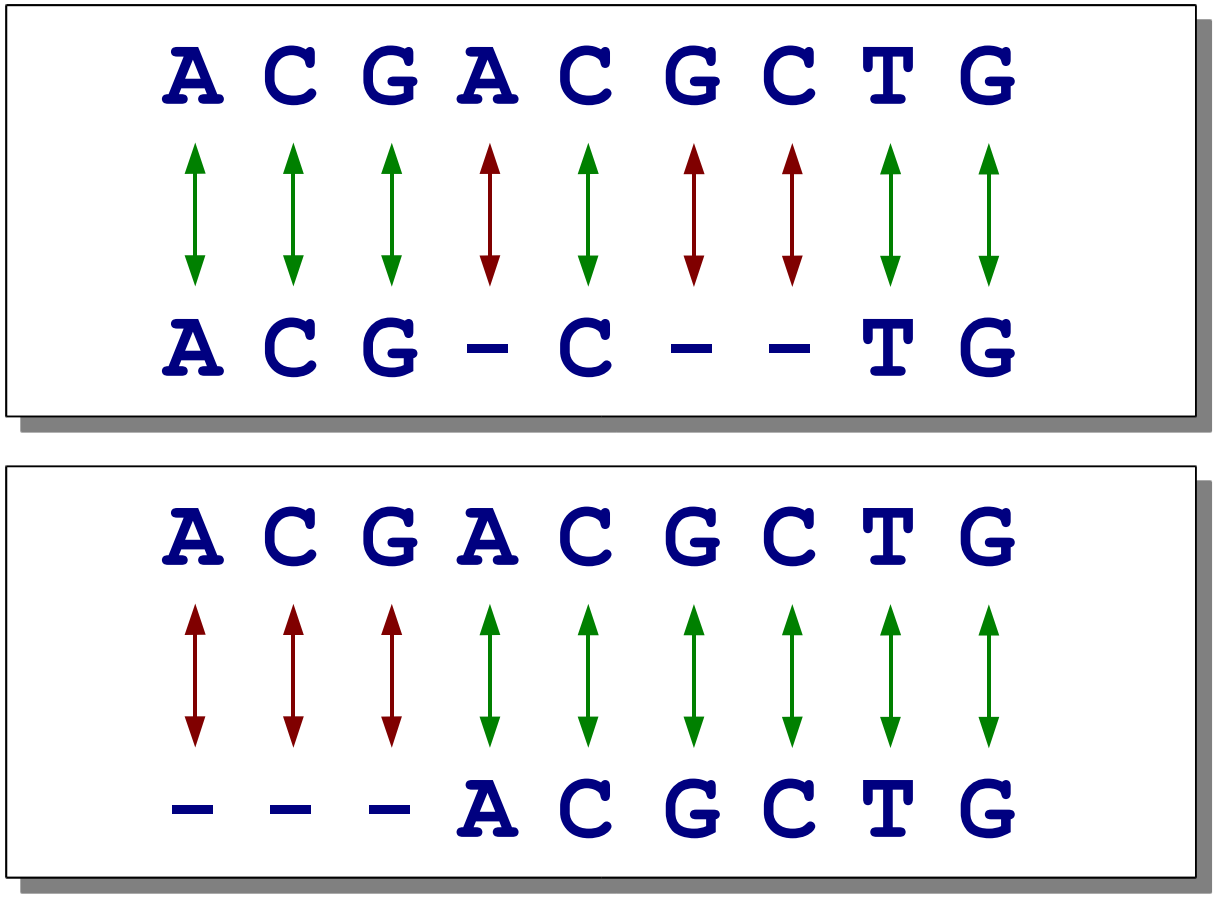
etc.

Hence, space complexity drops to $O(m)$.

This picture also gives clue as to why increase in time complexity is constant factor (x2), so asymptotic time complexity remains $O(mn)$.

General gap penalties

What is the difference between these two alignments?



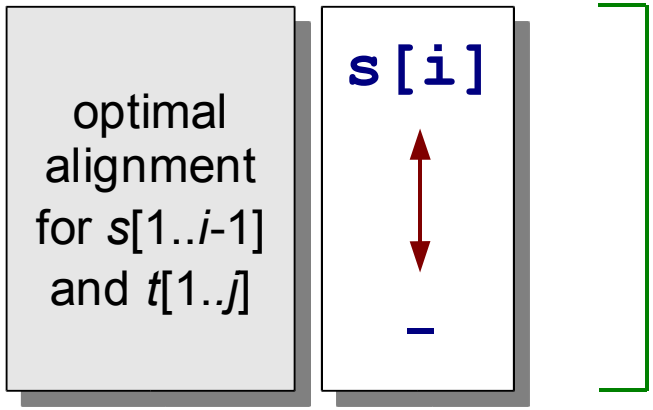
While scores are the same, second is preferable (fewer gaps).

General gap penalties

A *gap* is a series of $k > 1$ consecutive spaces. All else being equal, we prefer one gap of k spaces over k gaps of one space.

Let $w(k)$ be our gap penalty. Before, we had $w(k) = bk$ where b was a constant (the deletion and insertion cost). This is an *additive* gap penalty, which may not be true in general.

Assumptions we used to partition problem no longer valid. E.g.,

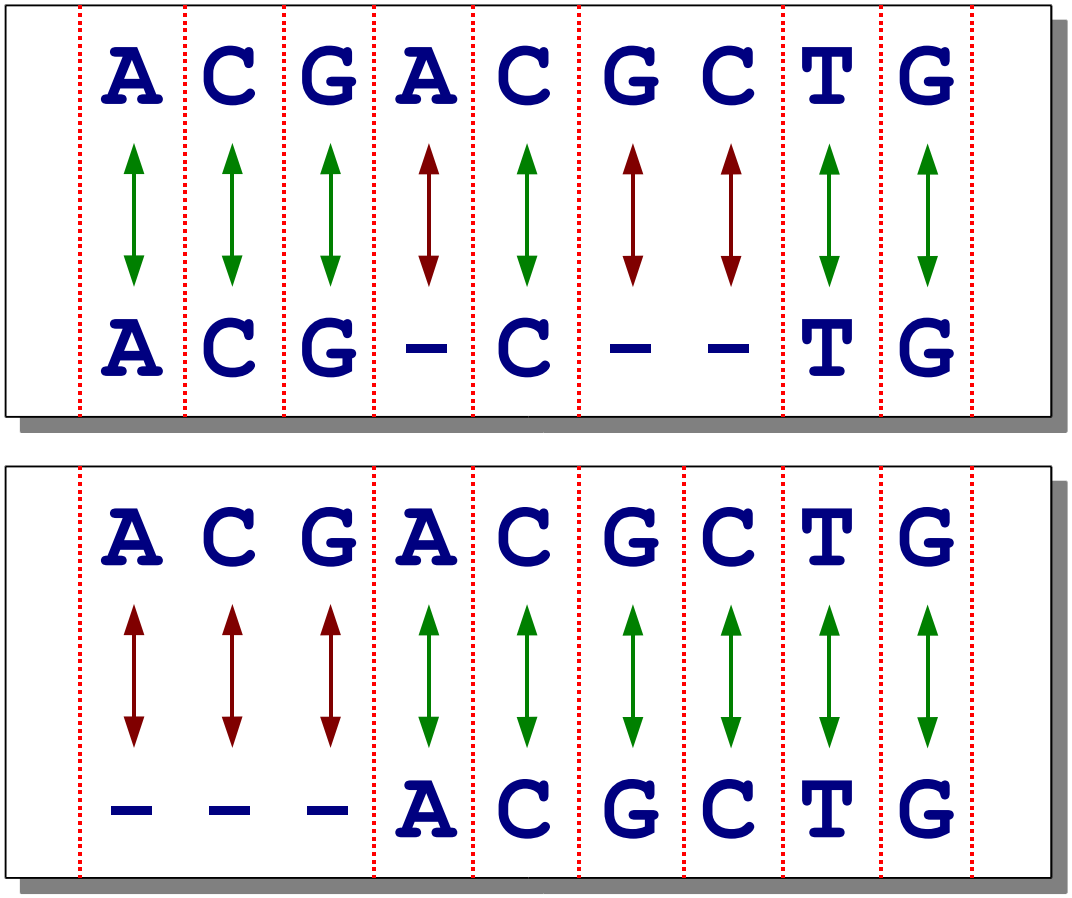


Cost of this alignment depends on whether or not optimal alignment for $s[1..i-1]$ and $t[1..j]$ has gap in t and, if so, its length.

So we can't just call this: $a[i-1, j] + c_{\text{del}}\, s([i])$

General gap penalties

We must now think of computing alignments in terms of *blocks*, maximal substrings where score additivity still holds.



6 matches + w(1) + w(2)

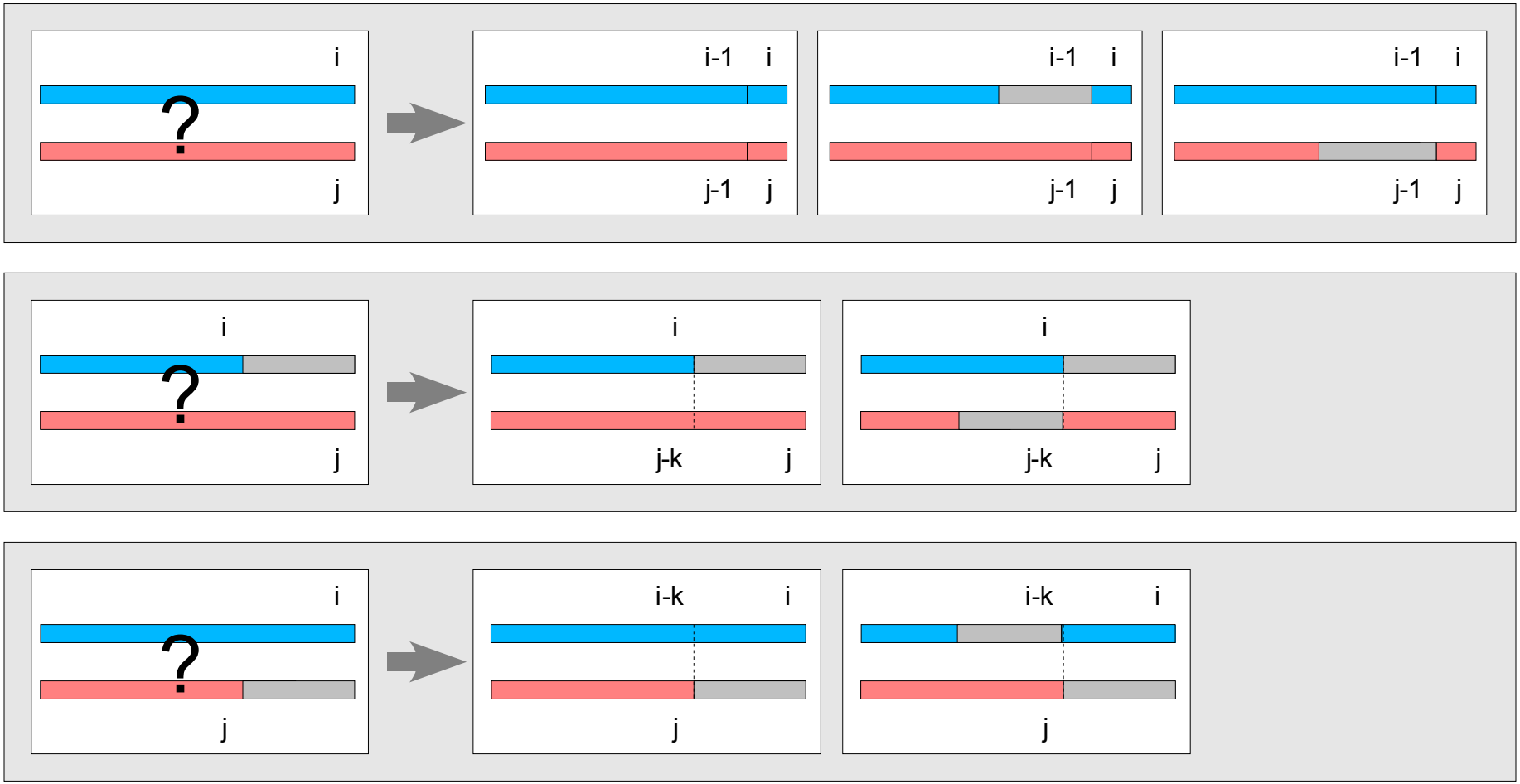
6 matches + w(3)

In general,
$w(3) \neq w(1) + w(2)$

General gap penalties

Cost of gap of consecutive spaces depends on final length.

As before, let's look at index *i* in string *s*, index *j* in string *t*:



General gap penalties

At any given point in time, alignment is in one of three states:
(1) last column may be pair of symbols from *s* and *t*,
(2) last column may be last space in gap in *s*,
(3) last column may be last space in gap in *t*,

Since we don't know which alternative the optimal solution will employ, we need to maintain three separate arrays *a*, *b*, and *c*.

a[*i*,*j*] assumes *s*[*i*] aligns with *t*[*j*].
b[*i*,*j*] assumes *t*[*j*] is matched to end of gap of spaces in *s*.
c[*i*,*j*] assumes *s*[*i*] is matched to end of gap of spaces in *t*.

General gap penalties

Initial conditions:

$$a[0,0]=0$$
$$b[0,j]=-w(j)$$
$$c[i,0]=-w(i)$$

Recurrences:

$$a[i,j] = c_{\text{sub}}(s[i],t[j]) + \max\begin{cases} a[i-1,j-1] \\ b[i-1,j-1] \\ c[i-1,j-1] \end{cases}$$

$$b[i,j] = \max\begin{cases} a[i,j-k]-w(k) & 1\le k\le j \\ c[i,j-k]-w(k) & 1\le k\le j \end{cases}$$

$$c[i,j] = \max\begin{cases} a[i-k,j]-w(k) & 1\le k\le i \\ b[i-k,j]-w(k) & 1\le k\le i \end{cases}$$

Final score is max of *a*[*m*,*n*], *b*[*m*,*n*], and *c*[*m*,*n*].

General gap penalties

What is time complexity of this algorithm?

Count number of times a previous value is read:

$$a[i,j] = c_{\text{sub}}(s[i],t[j]) + \max\begin{cases} a[i-1,j-1] \\ b[i-1,j-1] \\ c[i-1,j-1] \end{cases}$$ = 3

$$b[i,j] = \max\begin{cases} a[i,j-k]-w(k) & 1\le k\le j \\ c[i,j-k]-w(k) & 1\le k\le j \end{cases}$$ = 2j

$$c[i,j] = \max\begin{cases} a[i-k,j]-w(k) & 1\le k\le i \\ b[i-k,j]-w(k) & 1\le k\le i \end{cases}$$ = 2i

Hence, result we want is: $\sum_{i=1}^{m}\sum_{j=1}^{n}(2i+2j+3)$

General gap penalties

Computing time complexity: $\sum_{i=1}^{m}\sum_{j=1}^{n}(2i+2j+3)$

Look at inner summation:

$$\sum_{j=1}^{n}(2i+2j+3) = 2ni+3n+2\sum_{j=1}^{n} j =$$

$$2ni+3n+n(n+1) = 2ni+n^2+4n$$

Then:

$$\sum_{i=1}^{m}(2ni+n^2+4n) = mn^2+4mn+n2\sum_{i=1}^{m} i =$$

$$mn^2+4mn+nm(m+1) = m^2n+5mn+mn^2$$

Hence time complexity is $O(m^2n + mn^2)$.

Affine gap penalties

Assuming less generality about $w(k)$, we can do better.

Say that $w(k) = h + gk$. This is known as an *affine* gap penalty.

Now we only need to keep track of whether given space is first space in gap (in which case cost is $h + g$), or continuation of existing gap (in which case cost is g).

As before, we still need to maintain three arrays, a, b, and c:

$a[i,j]$ assumes $s[i]$ aligns with $t[j]$.

$b[i,j]$ assumes $t[j]$ aligns with space in s.

$c[i,j]$ assumes $s[i]$ aligns with space in t.

Affine gap penalties

Recurrences:

$$a[i,j] = c_{\text{sub}}(s[i],t[j]) + \max\begin{cases} a[i-1,j-1] \\ b[i-1,j-1] \\ c[i-1,j-1] \end{cases}$$

$$b[i,j] = \max\begin{cases} a[i,j-1]-(h+g) \\ b[i,j-1]-g \\ c[i,j-1]-(h+g) \end{cases}$$

$$c[i,j] = \max\begin{cases} a[i-1,j]-(h+g) \\ b[i-1,j]-(h+g) \\ c[i-1,j]-g \end{cases}$$

Time complexity here is $O(mn)$.

Wrap-up

Readings for next time:

- "The String-to-String Correction Problem with Block Moves," W. F. Tichy, *ACM Transactions on Computer Systems*, 2(4):309-321, November 1984.
- "Block Edit Models for Approximate String Matching," D. Lopresti and A. Tomkins, *Theoretical Computer Science*, vol. 181, no. 1, 1997, pp. 159-179.
  (Both papers available online or in Blackboard.)

Remember:

- Come to class prepared to discuss what you have read.
- Check Blackboard regularly for updates.