6.096 – Algorithms for Computational Biology

Sequence Alignment and Dynamic Programming

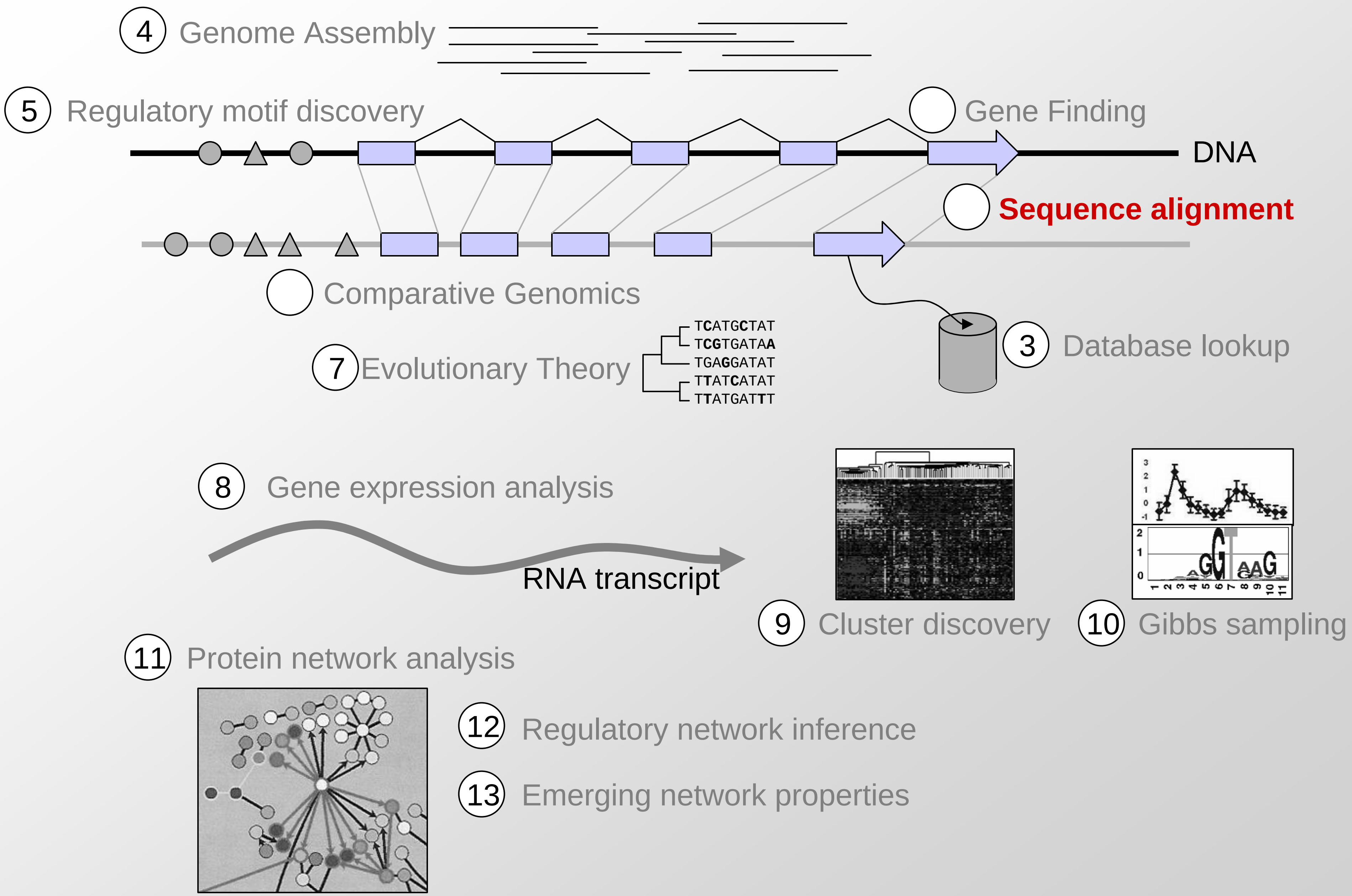
Lecture 1 - Introduction

Lecture 2 - Hashing and BLAST

Lecture 3 - Combinatorial Motif Finding

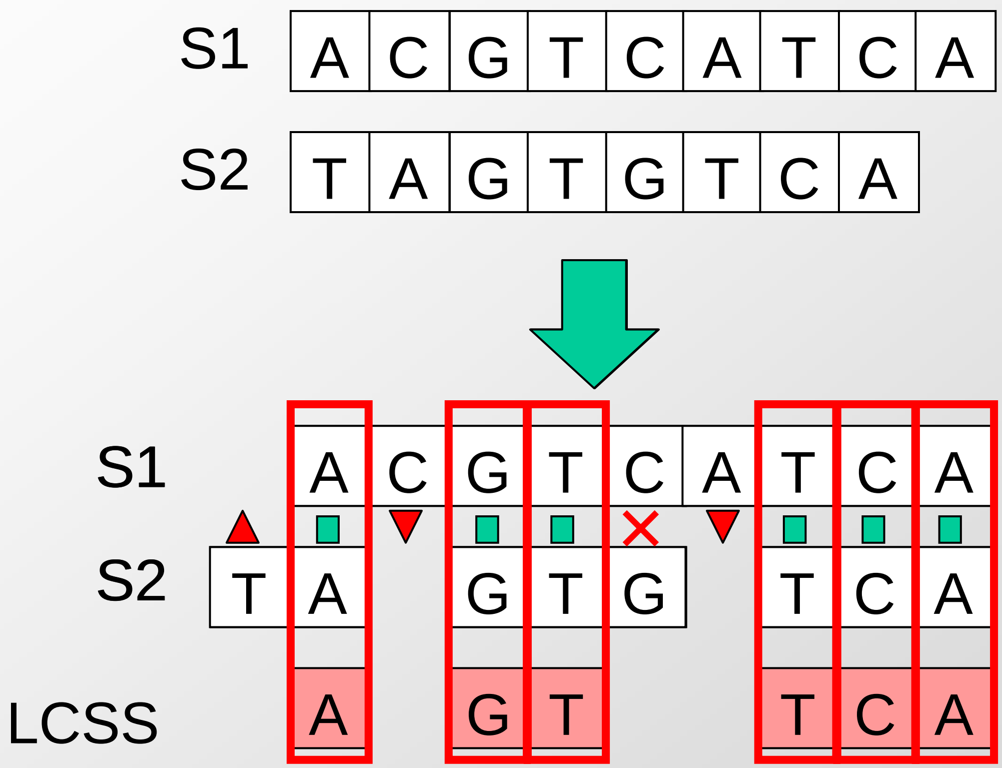
Lecture 4 - Statistical Motif Finding

Challenges in Computational Biology



Comparing two DNA sequences

- Given two possibly related strings S1 and S2
  - What is the longest common subsequence?



Edit distance:

•Number of changes needed for S1→S2

How can we compute best alignment

S1	A	C	G	T	C	A	T	C	A
S2	T	A	G	T	G	T	C	A	

- Need scoring function:
  - Score(alignment) = Total cost of editing S1 into S2
  - Cost of mutation
  - Cost of insertion / deletion
  - Reward of match
- Need algorithm for inferring best alignment
  - Enumeration?
  - How would you do it?
  - How many alignments are there?

Why we need a smart algorithm

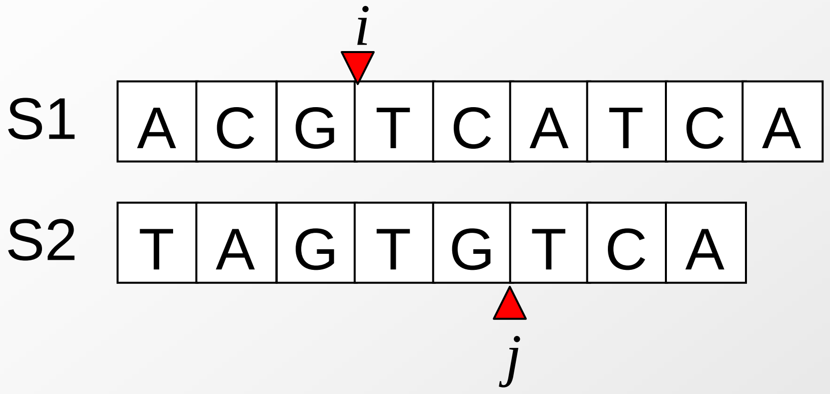
- Ways to align two sequences of length m, n

$$\binom{n+m}{m} \qquad \frac{(m+n)!}{(m!)^2} \approx \frac{2^{m+n}}{\sqrt{\pi \cdot m}}$$

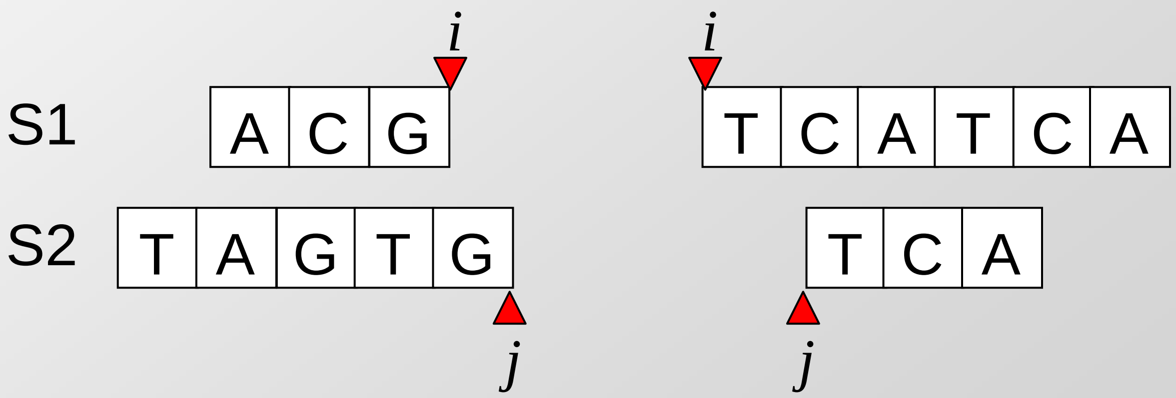
- For two sequences of length n

n	Enumeration	Today's lecture
10	184,756	100
20	1.40E+11	400
100	9.00E+58	10,000

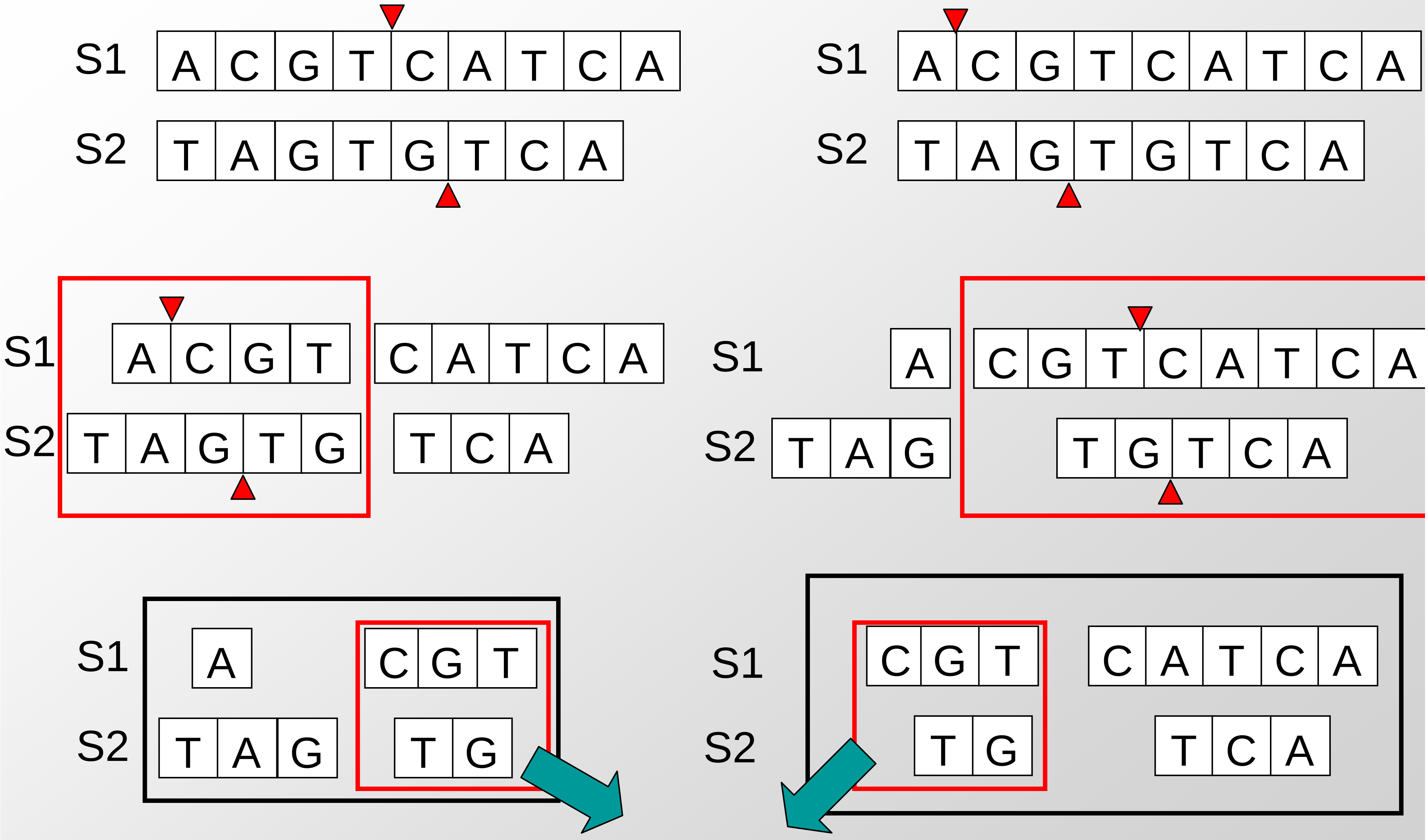
Key insight: score is additive!



- Compute best alignment recursively
  - For a given split ***(i, j)***, the best alignment is:
    - Best alignment of S1[1..i] and S2[1..j]
    - + Best alignment of S1[i..n] and S2[j..m]



Key insight: re-use computation



Identical sub-problems! We can reuse our work!

Solution #1 – Memoization

- Create a big dictionary, indexed by aligned seqs
  - When you encounter a new pair of sequences
  - If it is in the dictionary:
    - Look up the solution
  - If it is not in the dictionary
    - Compute the solution
    - Insert the solution in the dictionary
- Ensures that there is no duplicated work
  - Only need to compute each sub-alignment once!

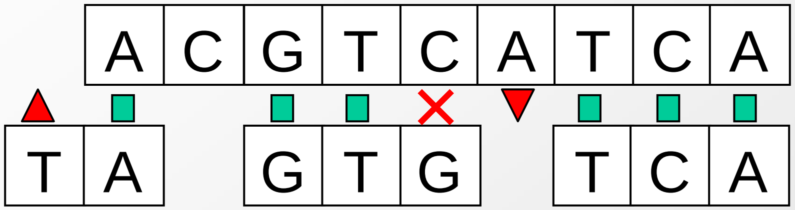
Top down approach

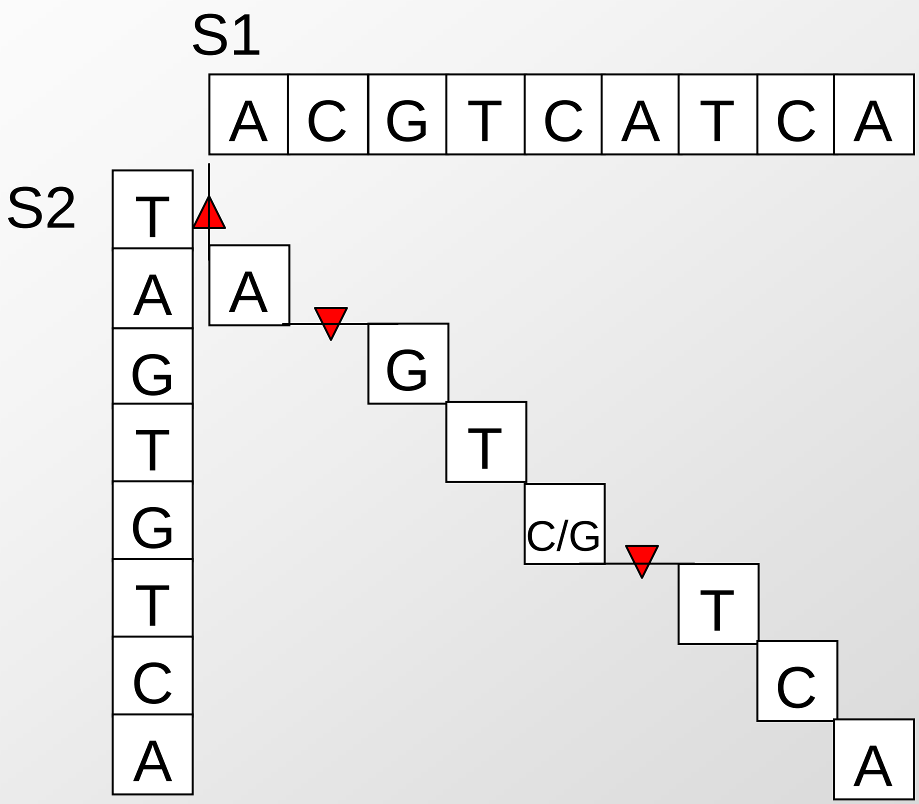
Solution #2 – Dynamic programming

- Create a big table, indexed by (i,j)
  - Fill it in from the beginning all the way till the end
  - You know that you'll need every subpart
  - Guaranteed to explore entire search space
- Ensures that there is no duplicated work
  - Only need to compute each sub-alignment once!
- Very simple computationally!

Bottom up approach

Key insight: Matrix representation of alignments





Goal:

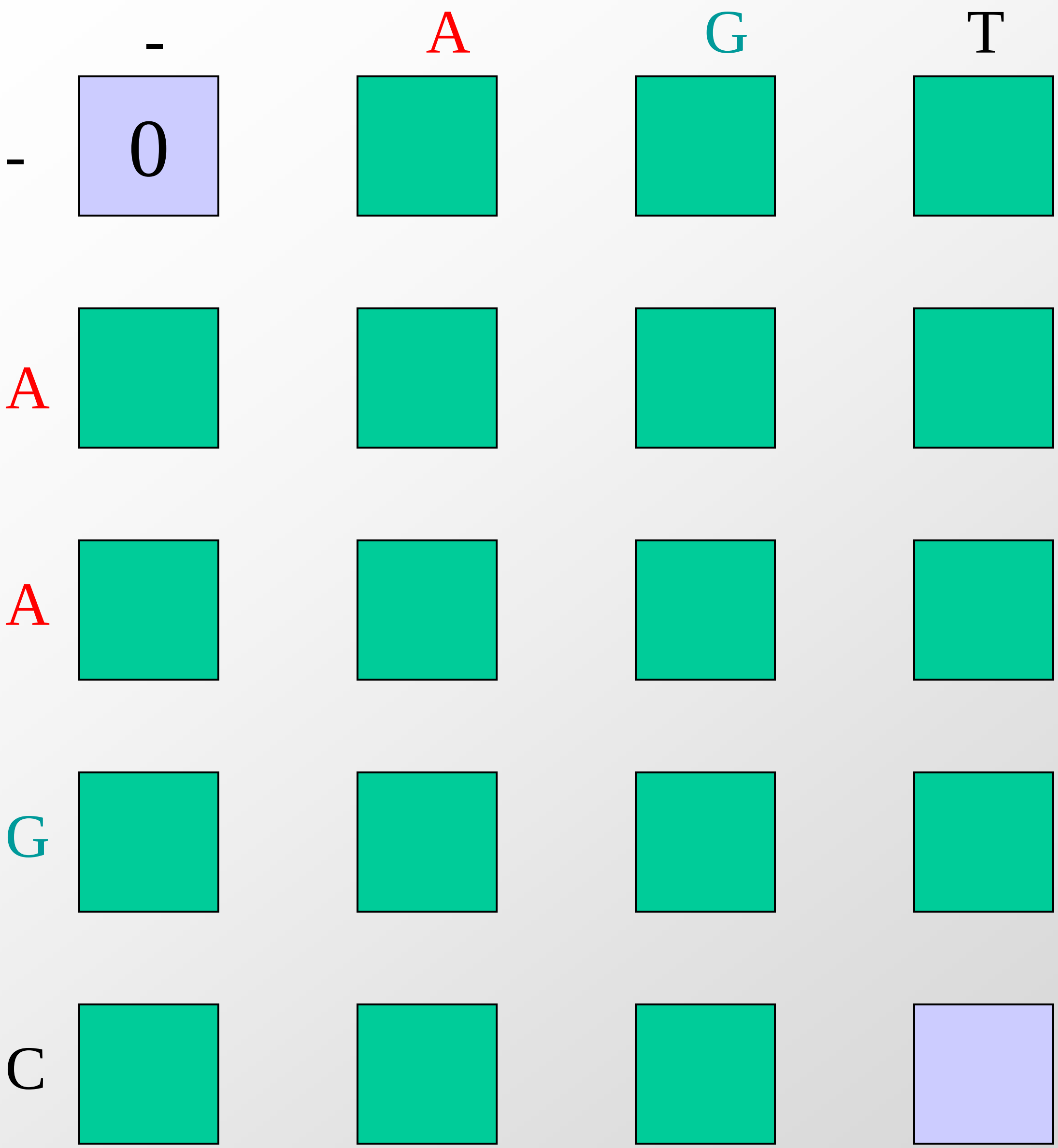
Find best path through the matrix

Sequence alignment

Dynamic Programming

Global alignment

0. Setting up the scoring matrix



Initialization:

- Top right: 0

Update Rule:

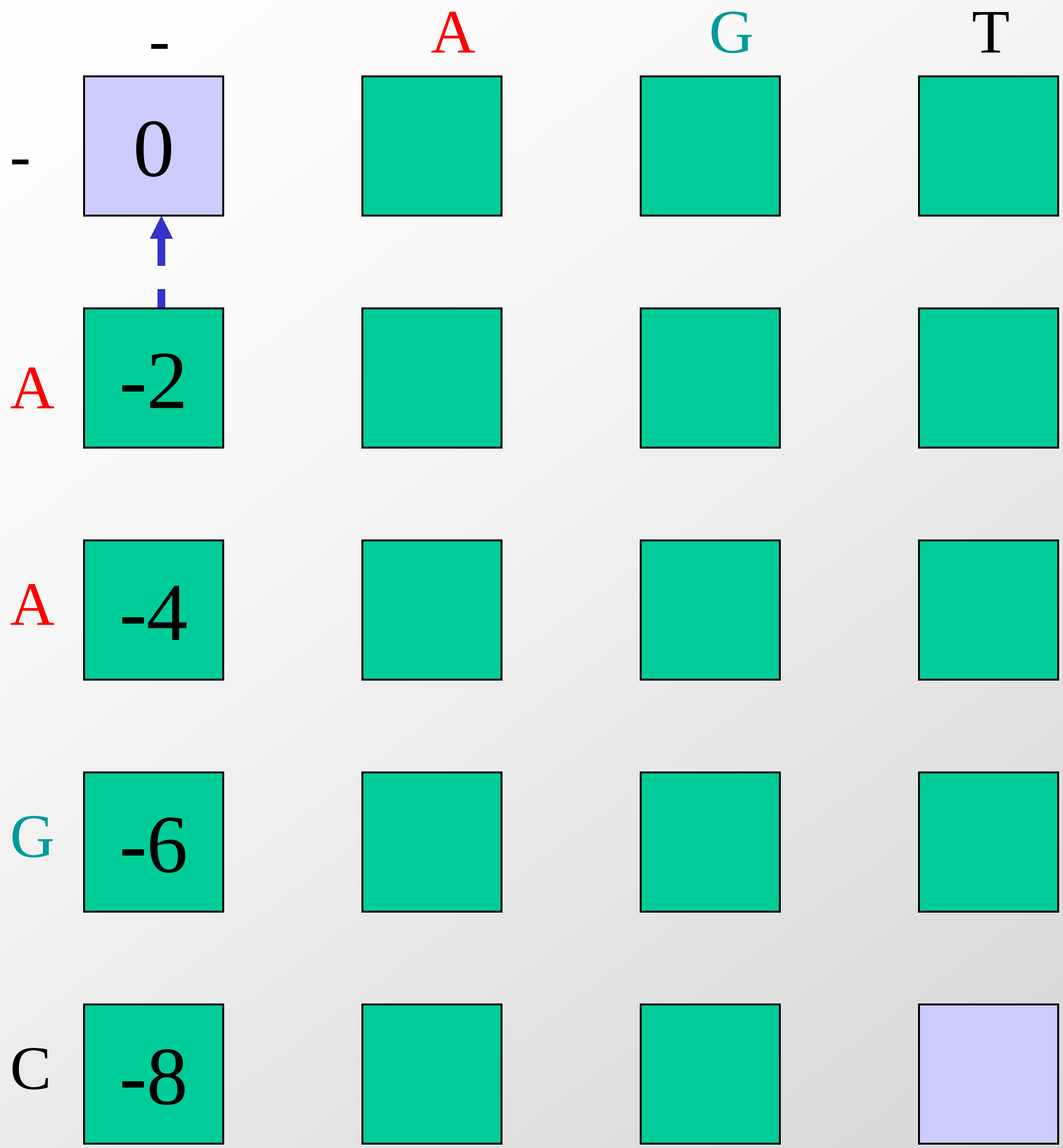
A(*i*,*j*)=max{

}

Termination:

- Bottom right

1. Allowing gaps in s



Initialization:

- Top right: 0

Update Rule:

A(i,j)=max{

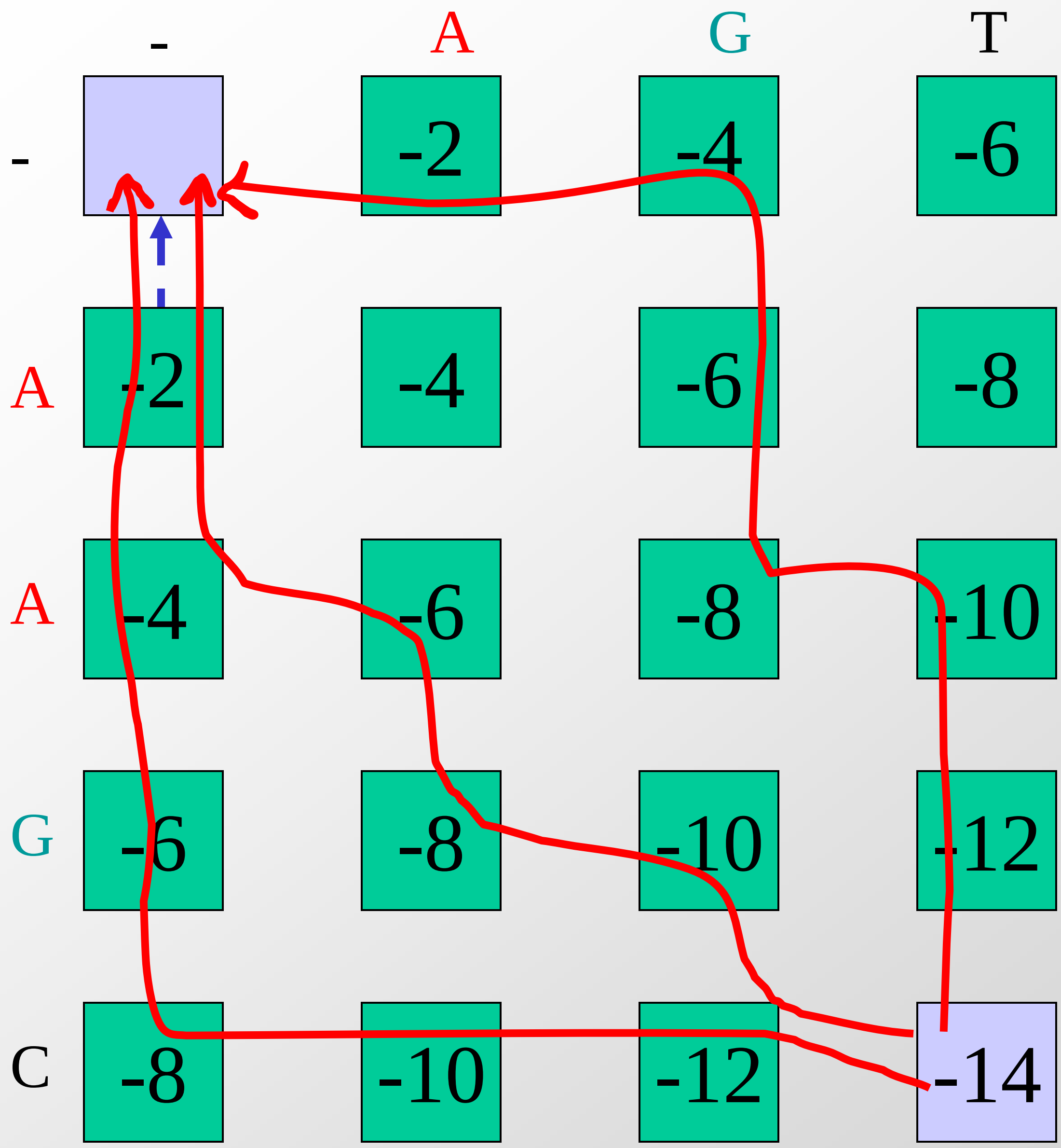
- A(i-1 , j) - 2

}

Termination:

- Bottom right

2. Allowing gaps in t



Initialization:

- Top right: 0

Update Rule:

A(i,j)=max{

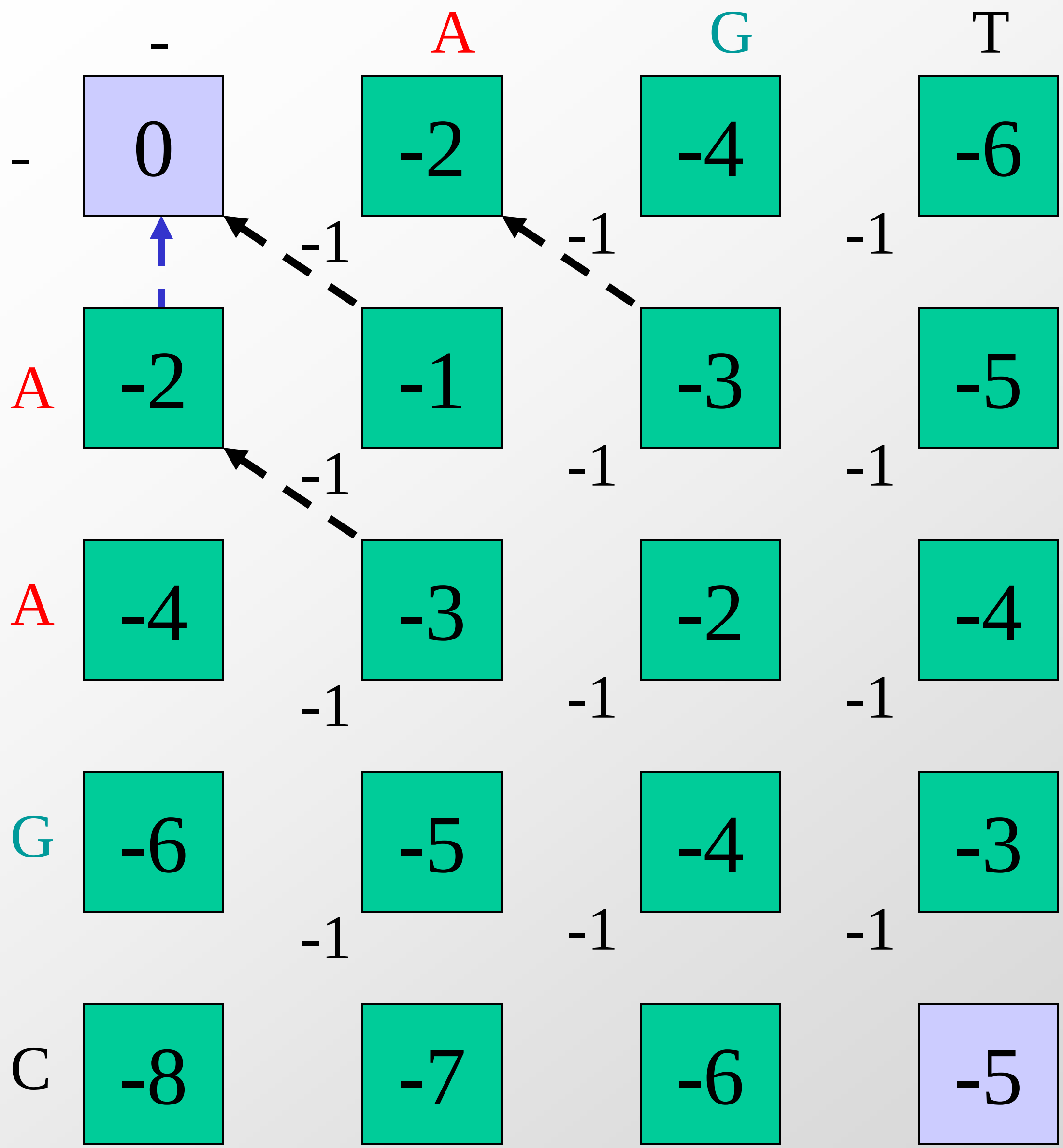
- A(i-1 , j) - 2
- A(i , j-1) - 2

}

Termination:

- Bottom right

3. Allowing mismatches



Initialization:

- Top right: 0

Update Rule:

A(*i*,*j*)=max{

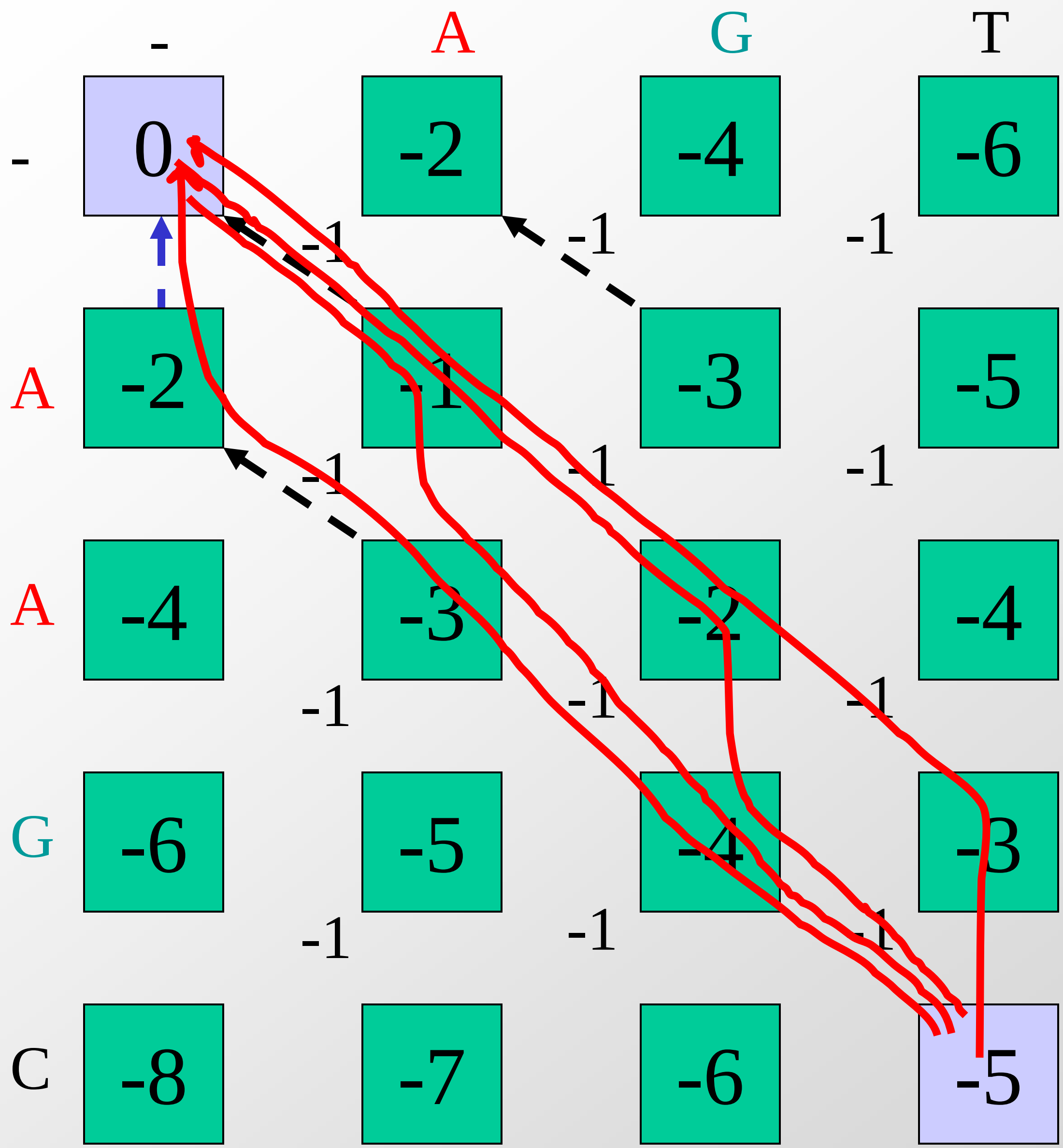
- A(*i*-1 , *j*) - 2
- A(*i* , *j*-1) - 2
- A(*i*-1 , *j*-1) -1

}

Termination:

- Bottom right

4. Choosing optimal paths



Initialization:

- Top right: 0

Update Rule:

A(i,j)=max{

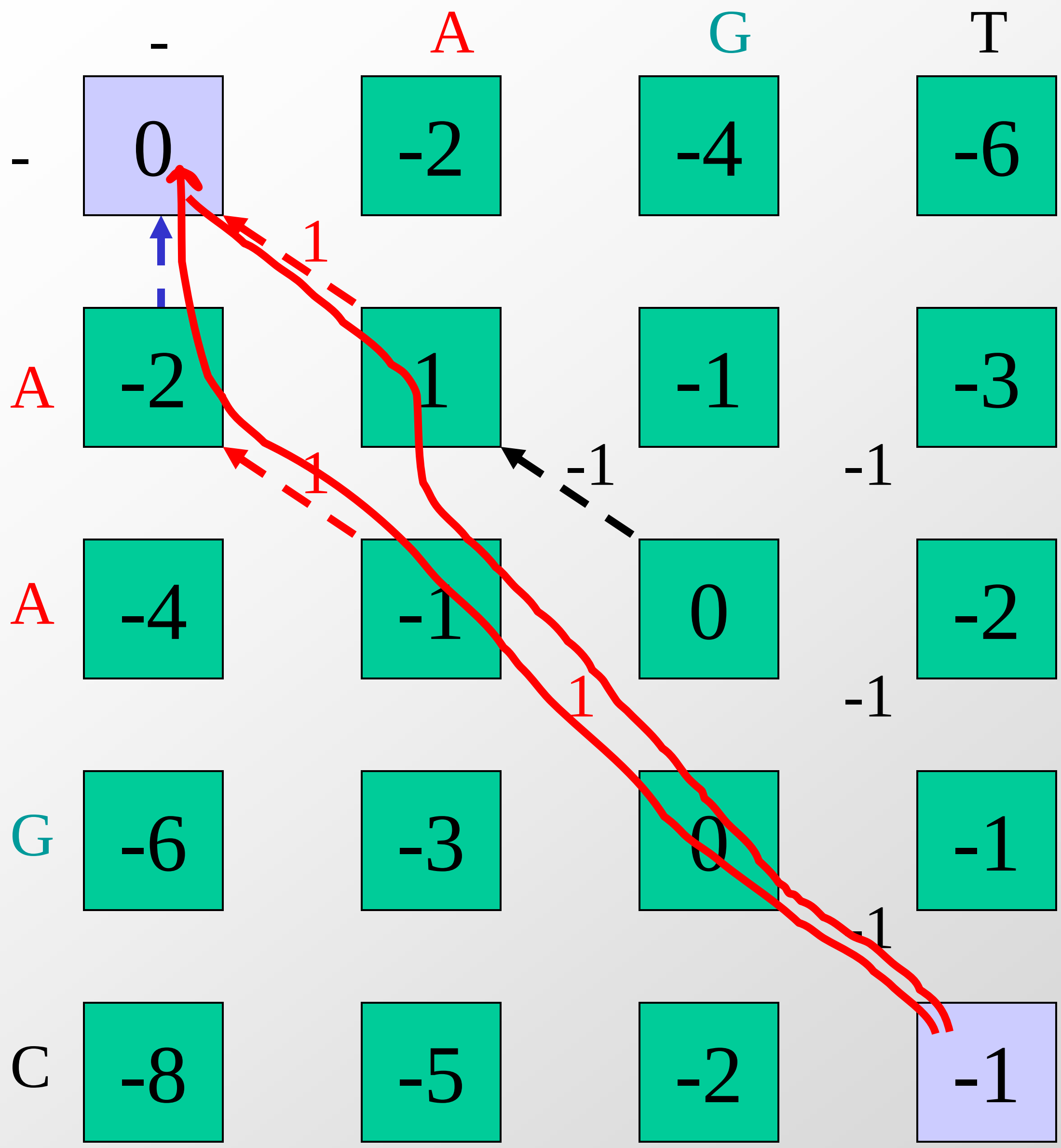
- A(i-1 , j) - 2
- A(i , j-1) - 2
- A(i-1 , j-1) -1

}

Termination:

- Bottom right

5. Rewarding matches



Initialization:

- Top right: 0

Update Rule:

A(*i*,*j*)=max{

- A(*i*-1 , *j*) - 2
- A(*i* , *j*-1) - 2
- A(*i*-1 , *j*-1) ±1

}

Termination:

- Bottom right

Sequence alignment

Dynamic Programming

Global Alignment

Semi-Global

Semi-Global Motivation

- Aligning the following sequences

```
CAGCACTTGGATTCTCGG
CAGC-----G-T----GG
vvvv-----v-v----vv  =  8(1)+0(-1)+10(-2) = -12
```

- We might prefer the alignment

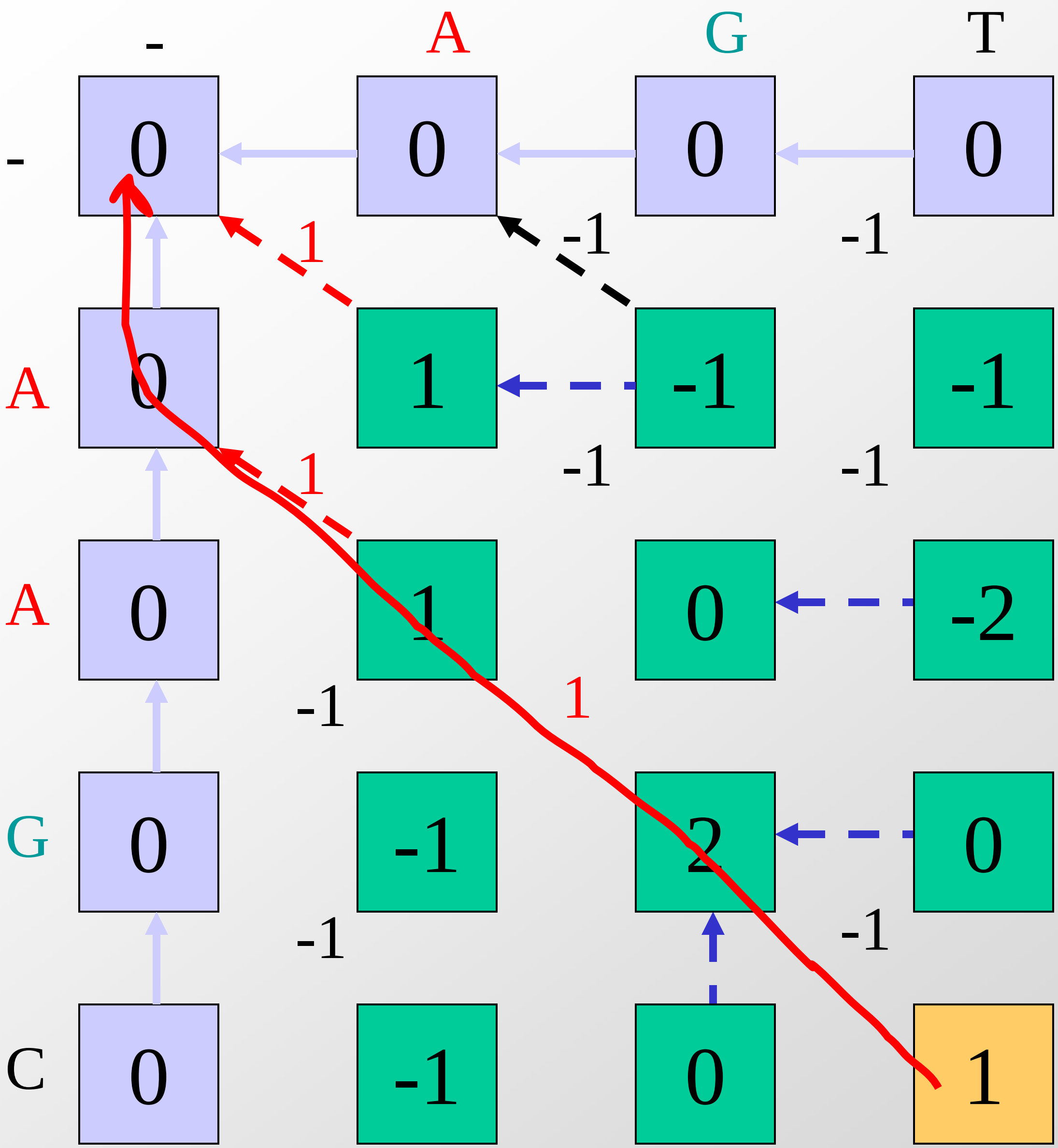
```
CAGCA-CTTGGATTCTCGG
---CAGCGTGG--------
```

match mismatch gap

```
---vv-vxvvv--------  =  6(1)+1(-1)+12(-2) = -19
```

- New qualities sought, new scoring scheme designed
  - Intuitively, don't penalize "missing" end of the sequence
  - We'd like to model this intuition

Ignoring starting gaps



Initialization:

- 1st row/col: 0

Update Rule:

A(i,j)=max{

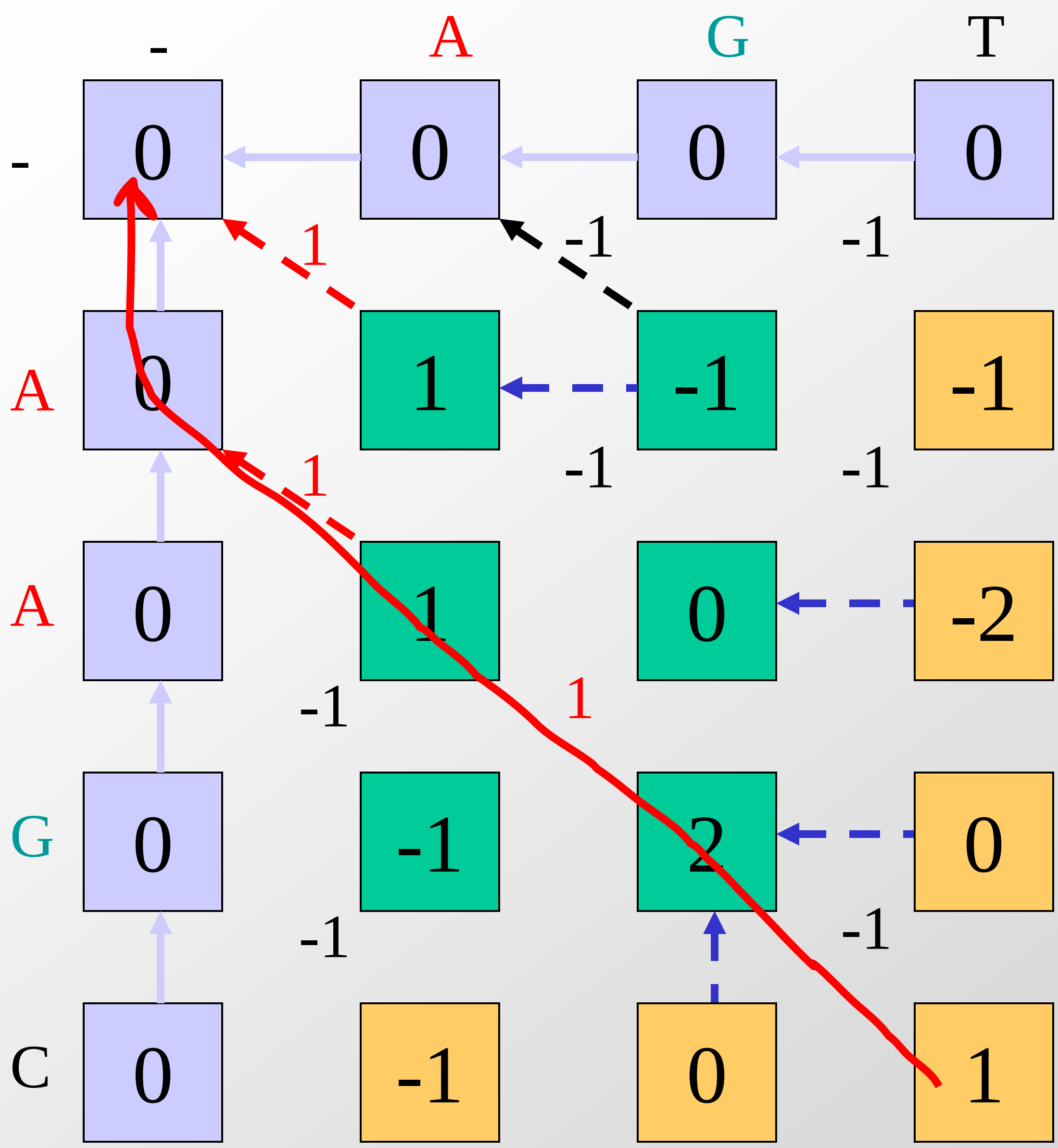
- A(i-1 , j) - 2
- A(i , j-1) - 2
- A(i-1 , j-1) ±1

}

Termination:

- Bottom right

Ignoring trailing gaps



Initialization:

- 1st row/col: 0

Update Rule:

A(*i*,*j*)=max{

- A(*i*-1 , *j*) - 2
- A(*i* , *j*-1) - 2
- A(*i*-1 , *j*-1) ±1

}

Termination:

- max(last row/col)

Using the new scoring scheme

- With the old scoring scheme (all gaps count -2)

```
CAGCACTTGGATTCTCGG
CAGC-----G-T----GG
vvvv-----v-v----vv
```

= 8(1)+0(-1)+10(-2)+0(-0) = -12

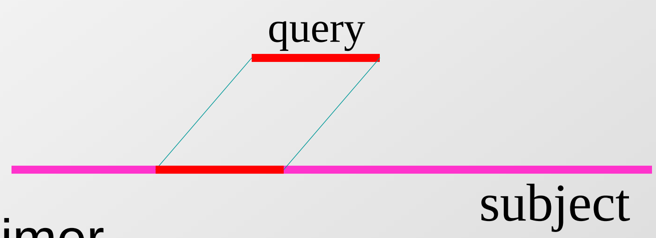
- New score (end gaps are free)

```
CAGCA-CTTGGATTCTCGG
---CAGCGTGG--------
---vv-vxvvv--------
```

= 6(1)+1(-1)+1(-2)+11(-0) = 3

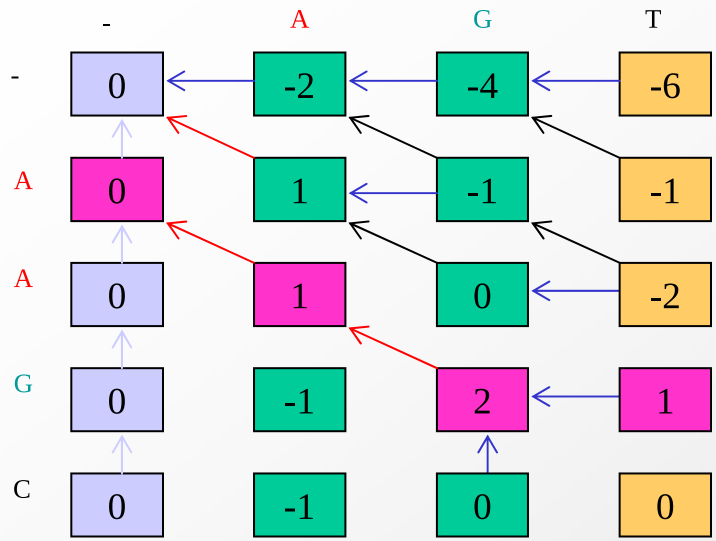
match mismatch gap endgap

Semi-global alignments



- Applications:
  - Finding a gene in a genome
  - Aligning a read onto an assembly
  - Finding the best alignment of a PCR primer
  - Placing a marker onto a chromosome
- These situations have in common
  - One sequence is much shorter than the other
  - Alignment should span the entire length of the smaller sequence
  - No need to align the entire length of the longer sequence
- In our scoring scheme we should
  - Penalize end-gaps for subject sequence
  - Do not penalize end-gaps for query sequence

Semi-Global Alignment



Initialization:

- 1st col

Update Rule:

A(*i*,*j*)=max{

- A(*i*-1 , *j*) - 2
- A(*i* , *j*-1) - 2
- A(*i*-1 , *j*-1) ±1

}

Termination:

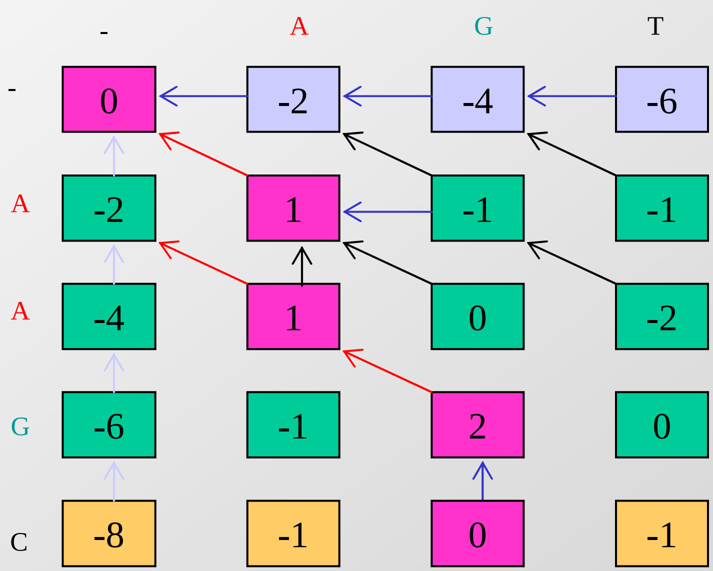
- max(last col)

Query: t

Subject: s

align all of t

...or...



Initialization:

- 1st row

Update Rule:

A(*i*,*j*)=max{

- A(*i*-1 , *j*) - 2
- A(*i* , *j*-1) - 2
- A(*i*-1 , *j*-1) ±1

}

Termination:

- max(last row)

Query: s

Subject: t

align all of s

Sequence alignment

Dynamic Programming

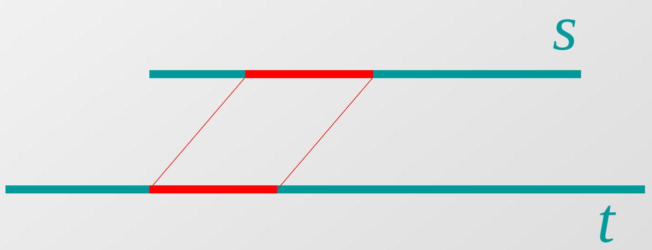
Global Alignment

Semi-Global

Local Alignment

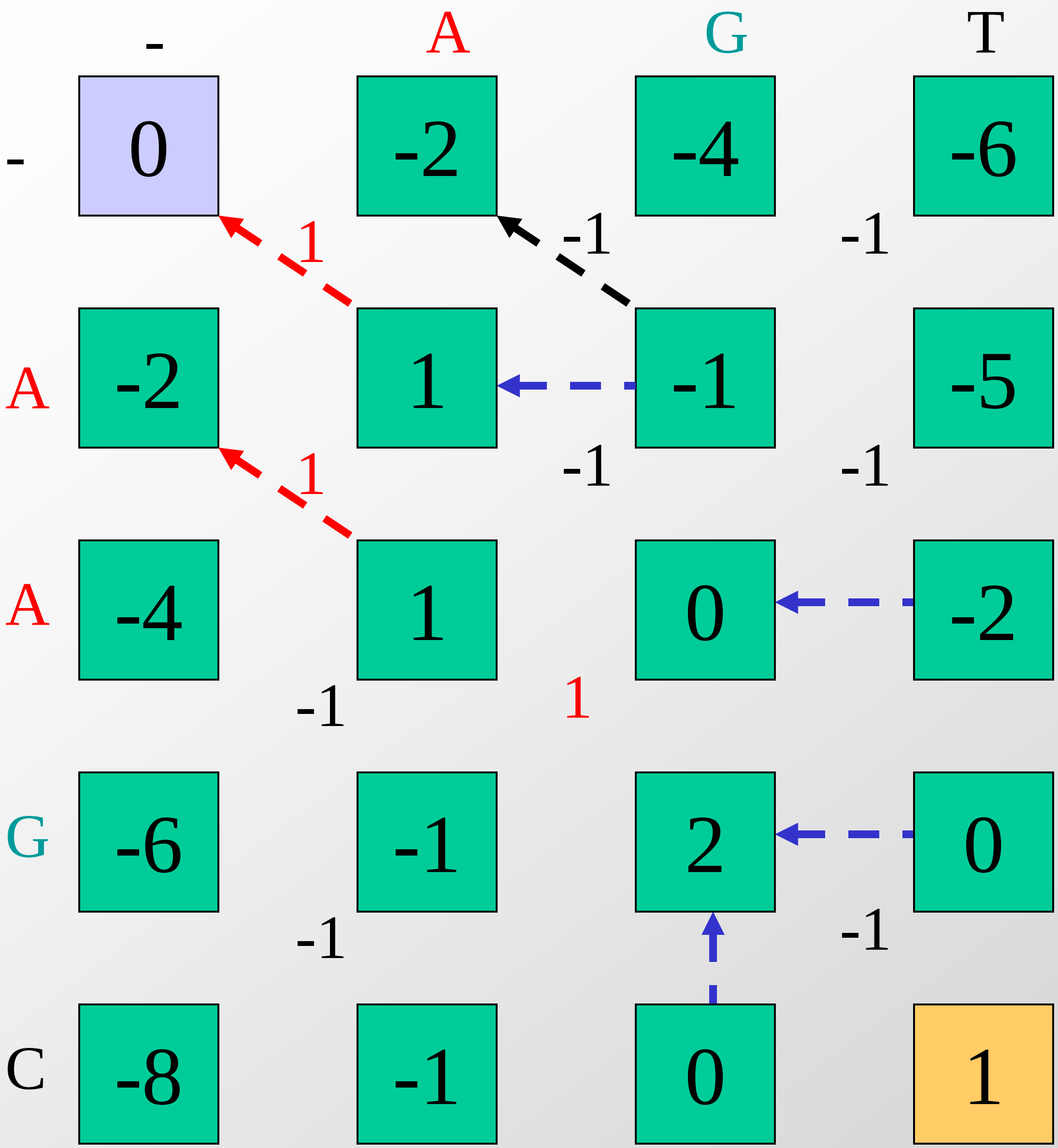
Intro to Local Alignments

- Statement of the problem
  - A *local alignment* of strings *s* and *t* is an alignment of a substring of *s* with a substring of *t*



- Definitions (reminder):
  - A **substring** consists of consecutive characters
  - A *subsequence* of *s* needs not be contiguous in *s*
- Naïve algorithm
  - Now that we know how to use dynamic programming
  - Take all $O((nm)^2)$, and run each alignment in $O(nm)$ time
- Dynamic programming
  - By modifying our existing algorithms, we achieve $O(mn)$

Global Alignment



Initialization:

- Top left: 0

Update Rule:

A(*i*,*j*)=max{

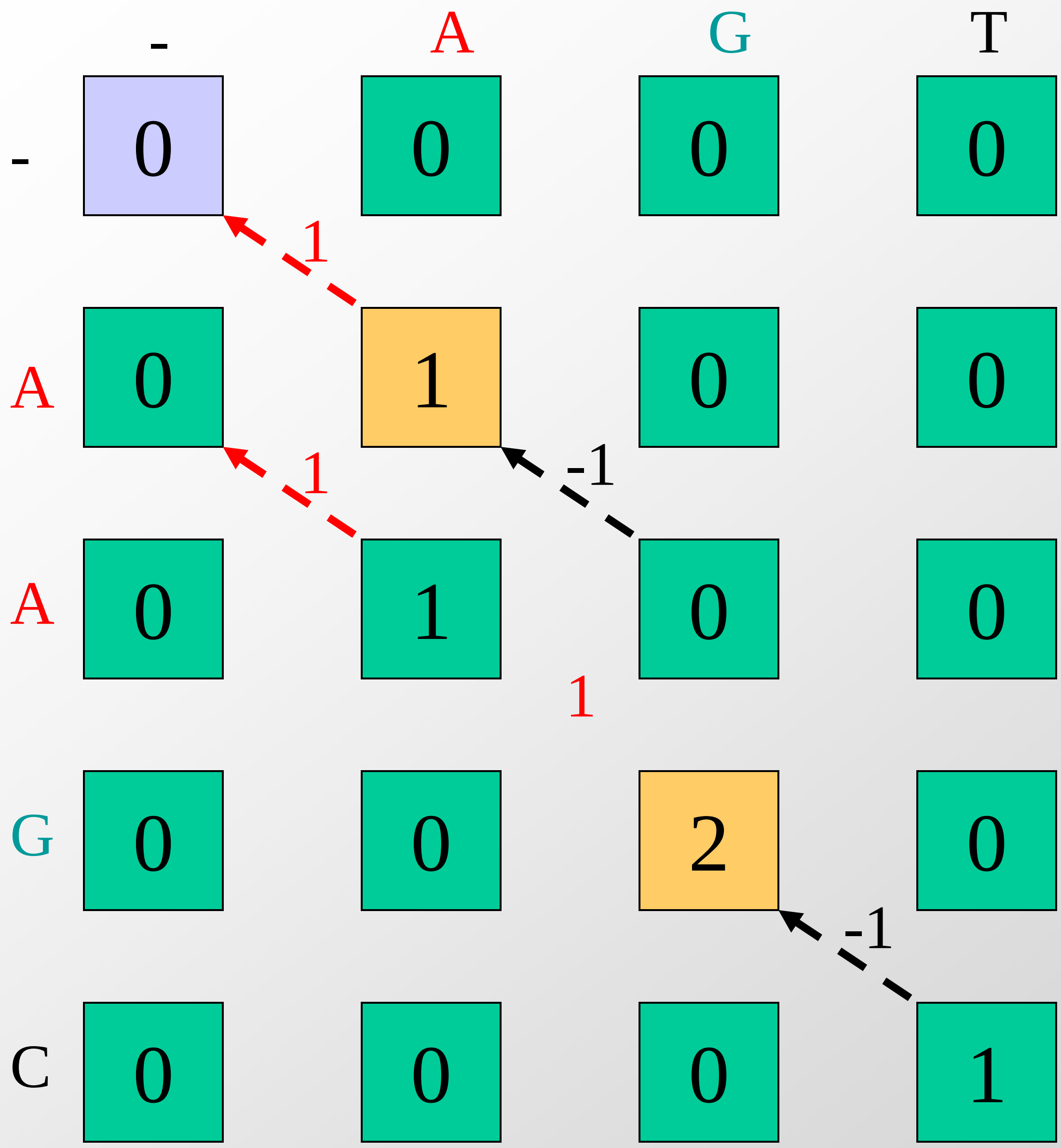
- A(*i*-1 , *j*) - 2
- A(*i* , *j*-1) - 2
- A(*i*-1 , *j*-1) ±1

}

Termination:

- Bottom right

Local Alignment



Initialization:

- Top left: 0

Update Rule:

A(i,j)=max{

- A(i-1 , j) - 2
- A(i , j-1) - 2
- A(i-1 , j-1) ±1
- 0

}

Termination:

- Anywhere

Local Alignment issues

- Resolving ambiguities
  - When following arrows back, one can stop at any of the zero entries. Only stop when no arrow leaves. Longest.
- Correctness sketch by induction
  - Assume we've correctly aligned up to (i,j)
  - Consider the four cases of our max computation
  - By inductive hypothesis recurse on (i-1,j-1), (i-1,j), (i,j-1)
  - Base case: empty strings are suffixes aligned optimally
- Time analysis
  - O(mn) time
  - O(mn) space, can be brought to O(m+n)

Sequence alignment

Dynamic Programming

Global Alignment

Semi-Global

Local Alignment

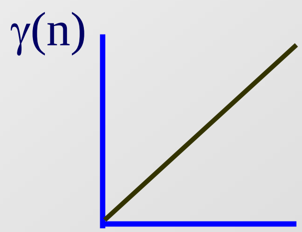
Affine Gap Penalty

Scoring the gaps more accurately

Current model:

Gap of length n

incurs penalty $n \times d$

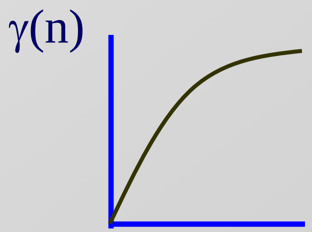


However, gaps usually occur in bunches

Convex gap penalty function:

$\gamma(n)$:

for all n, $\gamma(n + 1) - \gamma(n) \leq \gamma(n) - \gamma(n - 1)$



General gap dynamic programming

Initialization: same

Iteration:

$$F(i, j) = \max \begin{cases} F(i-1, j-1) + s(x_i, y_j) \\ \max_{k=0\ldots i-1} F(k,j) - \gamma(i-k) \\ \max_{k=0\ldots j-1} F(i,k) - \gamma(j-k) \end{cases}$$

Termination: same

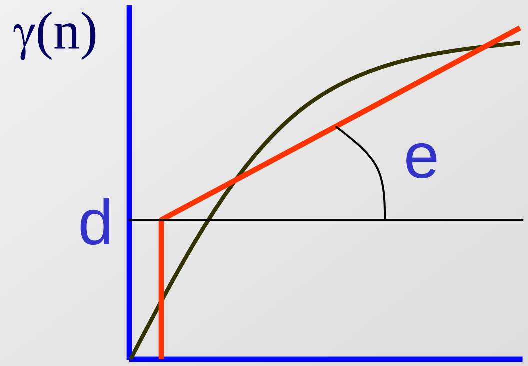
Running Time: $O(N^2M)$ (assume N>M)

Space: $O(NM)$

Compromise: affine gaps

$\gamma(n) = d + (n - 1) \times e$

gap open	gap extend



To compute optimal alignment,

At position i,j, need to "remember" best score if gap is open
best score if gap is not open

F(i, j): score of alignment $x_1 \ldots x_i$ to $y_1 \ldots y_j$
if x_i aligns to y_j

G(i, j): score **if** x_i, or y_j, aligns to a gap

Motivation for affine gap penalty

- Modeling evolution
  - To introduce the first gap, a break must occur in DNA
  - Multiple consecutive gaps likely to be introduced by the same evolutionary event. Once the break is made, it’s relatively easy to make multiple insertions or deletions.
  - Fixed cost for opening a gap: p+q
  - Linear cost increment for increasing number of gaps: q
- Affine gap cost function
  - New gap function for length k: $w(k) = p+q*k$
  - p+q is the cost of the first gap in a run
  - q is the additional cost of each additional gap in same run

Additional Matrices

- The amount of state needed increases
  - In scoring a single entry in our matrix, we need remember an extra piece of information
    - Are we continuing a gap in s? (if not, start is more expensive)
    - Are we continuing a gap in t? (if not, start is more expensive)
    - Are we continuing from a match between s(i) and t(j)?
- Dynamic programming framework
  - We encode this information in three different states for each element (i,j) of our alignment. Use three matrices
    - a(i,j): best alignment of s[1..i] & t[1..j] that aligns s[i] with t[j]
    - b(i,j): best alignment of s[1..i] & t[1..j] that aligns gap with t[j]
    - c(i,j): best alignment of s[1..i] & t[1..j] that aligns s[i] with gap

Update rules

When s[j] and t[j] are aligned

$$a(i,j) \quad score(s[i],t[j]) + \max \begin{pmatrix} a(i-1,j-1) \\ b(i-1,j-1) \\ c(i-1,j-1) \end{pmatrix}$$

Score can be different for each pair of chars

When t[j] aligns with a gap in s

$$b(i,j) \quad \max \begin{pmatrix} a(i,j-1)-(p+q) \\ b(i,j-1)-q \\ c(i,j-1)-(p+q) \end{pmatrix}$$

← starting a gap in s

← extending a gap in s

← Stopping a gap in t, and starting one in s

When s[i] aligns with a gap in t

$$c(i,j) \quad \max \begin{pmatrix} a(i-1,j)-(p+q) \\ c(i-1,j)-q \\ b(i-1,j)-(p+q) \end{pmatrix}$$

Find maximum over all three arrays *max(a[m,n],b[m,n],c[m,n]).*
Follow arrows back, skipping from matrix to matrix

Simplified rules

- Transitions from b to c are not necessary...

  ...if the worst mismatch costs less than p+q

```
ACC-GGTA        ACCGGTA
A--TGGTA   →    A-TGGTA
```

When s[j] and t[j] are aligned

$$a(i,j) = score(s[i],t[j]) + \max\begin{pmatrix} a(i-1,j-1) \\ b(i-1,j-1) \\ c(i-1,j-1) \end{pmatrix}$$

Score can be different for each pair of chars

When t[j] aligns with a gap in s

$$b(i,j) = \max\begin{pmatrix} a(i,j-1)-(p+q) \\ b(i,j-1)-q \end{pmatrix}$$

← starting a gap in s

← extending a gap in s

When s[i] aligns with a gap in t

$$c(i,j) = \max\begin{pmatrix} a(i-1,j)-(p+q) \\ c(i-1,j)-q \end{pmatrix}$$

General Gap Penalty

- Gap penalties are limited by the amount of state
  - Affine gap penalty: $w(k) = k*p$
    - State: Current index tells if in a gap or not
  - Linear gap penalty: $w(k) = p + q*k$, where $q<p$
    - State: add binary value for each sequence: starting a gap or not
  - What about quadriatic: $w(k) = p+q*k+rk^2$.
    - State: needs to encode the length of the gap, which can be O(n)
    - To encode it we need O(log n) bits of information. Not feasible
  - What about a (mod 3) gap penalty for protein alignments
    - Gaps of length divisible by 3 are penalized less: conserve frame
    - This is feasible, but requires more possible states
    - Possible states are: starting, mod 3=1, mod 3=2, mod 3=0

Sequence alignment

Dynamic Programming

Global Alignment

Semi-Global

Local Alignment

Linear Gap Penalty

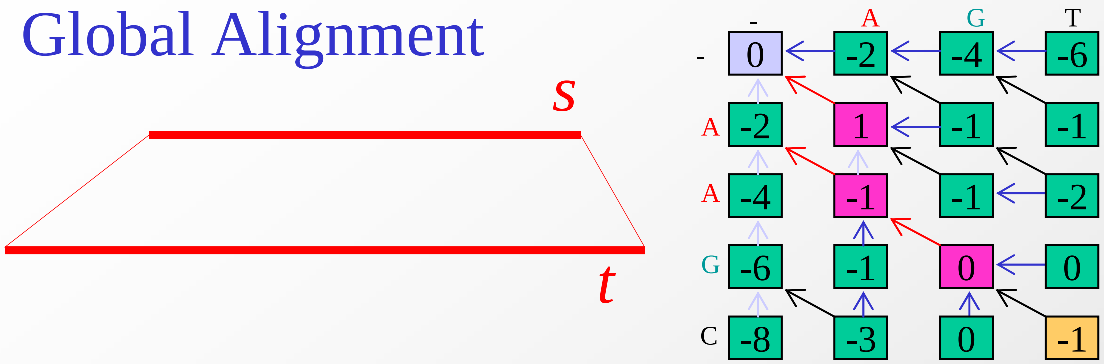
Variations on the Theme

Dynamic Programming Versatility

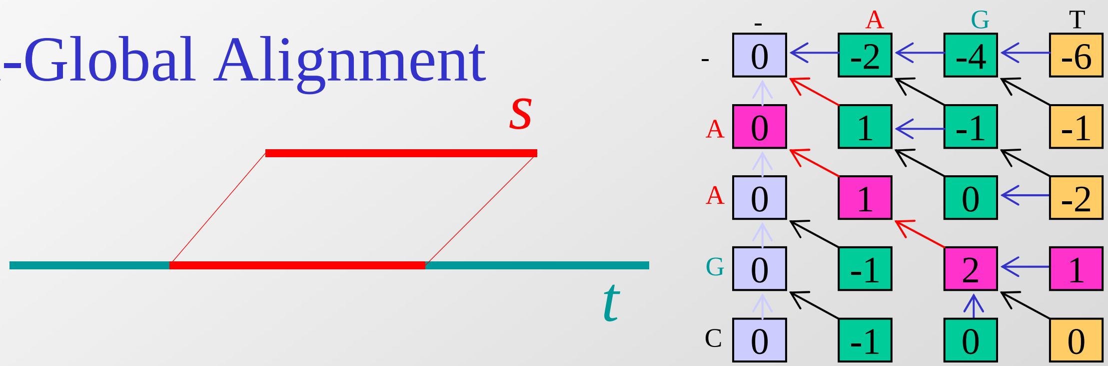
- Unified framework
  - Dynamic programming algorithm. Local updates.
  - Re-using past results in future computations.
  - Memory usage optimizations
- Tools in our disposition
  - Global alignment: entire length of two orthologous genes
  - Semi-global alignment: piece of a larger sequence aligned entirely
  - Local alignment: two genes sharing a functional domain
  - Linear Gap Penalty: penalize first gap more than subsequent gaps
  - Edit distance, min # of edit operations. M=0, m=g=-1, every operation subtracts 1, be it mutation or gap
  - Longest common subsequence: M=1, m=g=0. Every match adds one, be it contiguous or not with previous.

DP Algorithm Variations

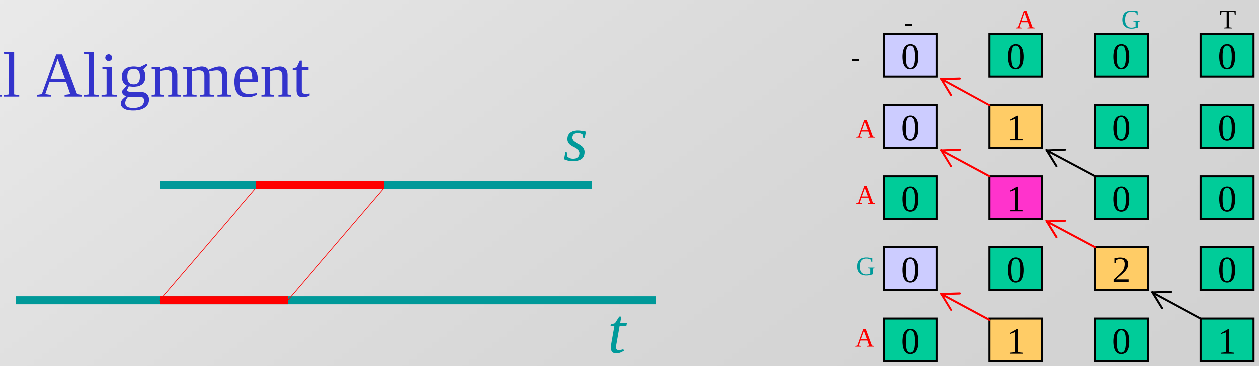
Global Alignment



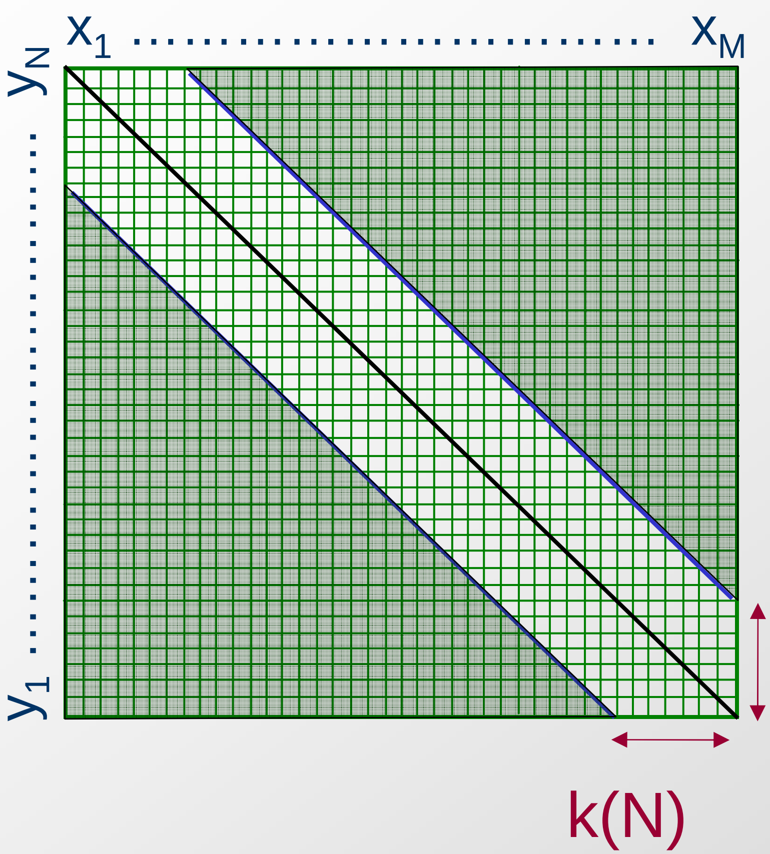
Semi-Global Alignment



Local Alignment



Bounded Dynamic Programming



Initialization:

F(i,0), F(0,j) undefined for i, j > k

Iteration:

For i = 1…M

For j = max(1, i – k)…min(N, i+k)

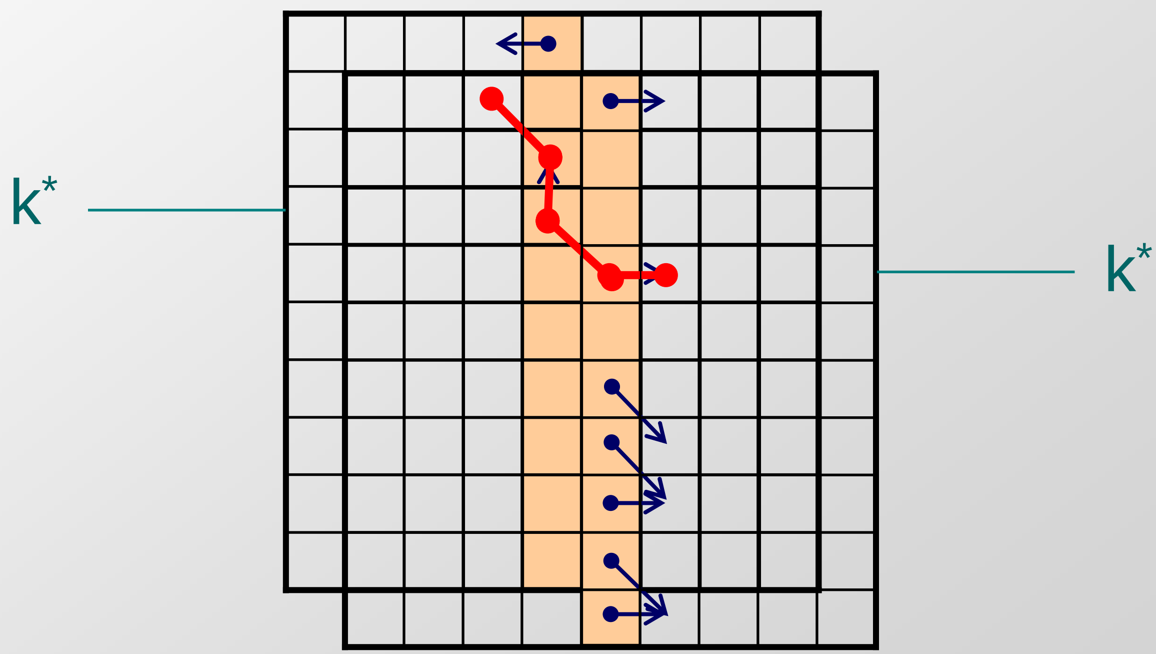
$$F(i, j) = \max \begin{cases} F(i-1, j-1) + s(x_i, y_j) \\ F(i, j-1) - d, \text{ if } j > i - k(N) \\ F(i-1, j) - d, \text{ if } j < i + k(N) \end{cases}$$

Termination: same

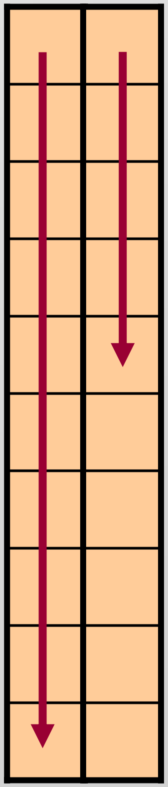
Easy to extend to the affine gap case

Linear-space alignment

- Now, we can find k^* maximizing $F(M/2, k) + F^r(M/2, N-k)$
- Also, we can trace the path exiting column M/2 from k^*



Linear-Space Alignment

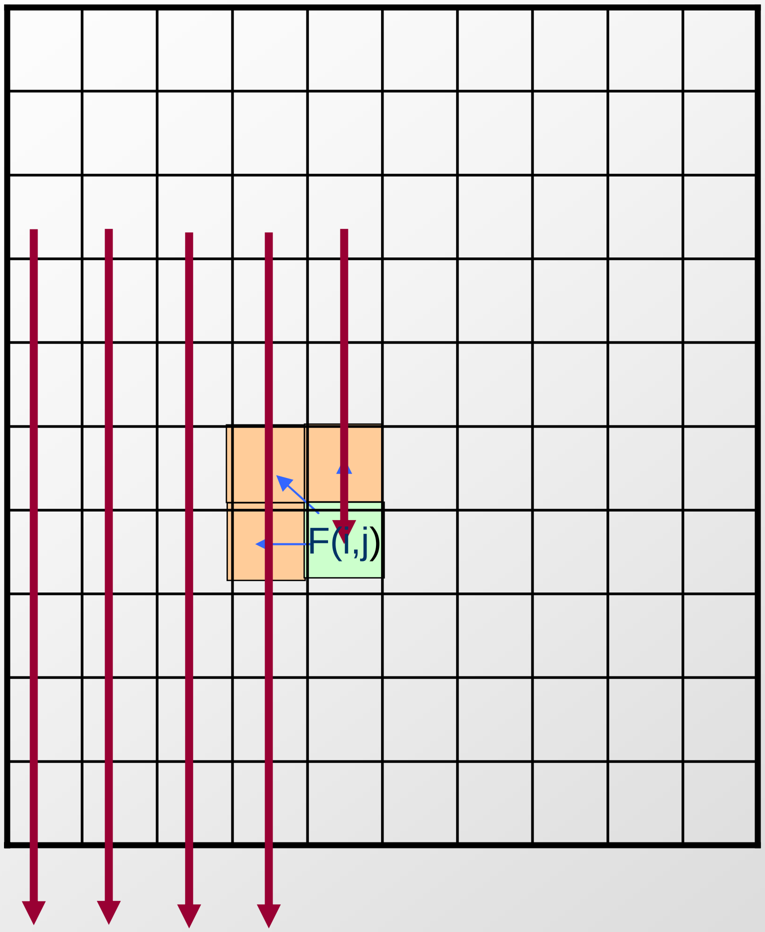


Hirschberg's algorithm

- Longest common subsequence
  - Given sequences $s = s_1\, s_2 \ldots s_m$, $t = t_1\, t_2 \ldots t_n$,
  - Find longest common subsequence $u = u_1 \ldots u_k$
- Algorithm:
  - $$F(i, j) = \max \begin{cases} F(i-1, j) \\ F(i, j-1) \\ F(i-1, j-1) + [1, \text{ if } s_i = t_j; 0 \text{ otherwise}] \end{cases}$$
- Hirschberg's algorithm solves this in linear space

Introduction: Compute optimal score

It is easy to compute F(M, N) in linear space



Allocate (column[1])
Allocate (column[2])

For i = 1….M
  If i > 1, then:
    Free(column[i – 2])
    Allocate(column[i])
  For j = 1…N
    F(i, j) = …

Linear-space alignment

To compute both the optimal score **and** the optimal alignment:

Divide & Conquer approach:

Notation:

x^r, y^r: reverse of x, y

E.g. x = accgg;

x^r = ggcca

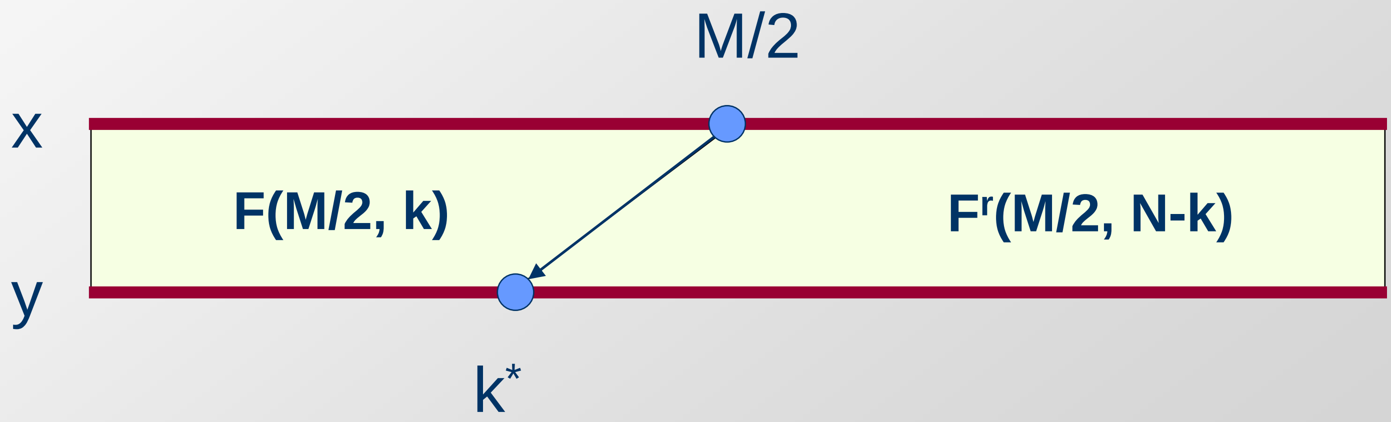
$F^r(i, j)$: optimal score of aligning $x^r_1 \ldots x^r_i$ & $y^r_1 \ldots y^r_j$

same as F(M-i+1, N-j+1)

Linear-space alignment

Lemma:

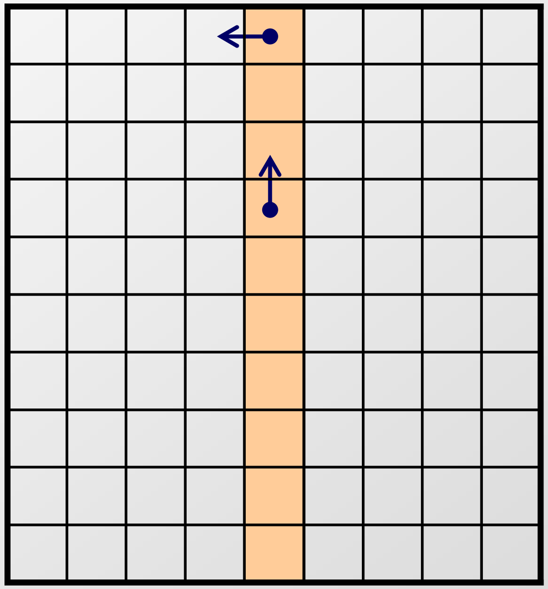
$$F(M, N) = \max_{k=0\ldots N}(F(M/2, k) + F^r(M/2, N-k))$$

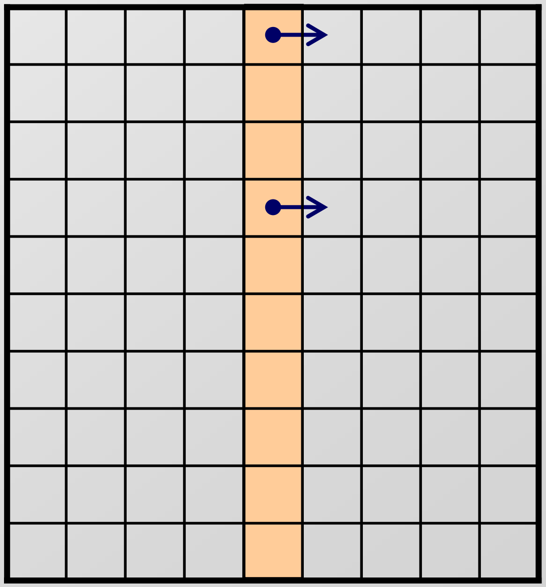


Linear-space alignment

- Now, using 2 columns of space, we can compute for k = 1…M, F(M/2, k), F^r(M/2, N-k)

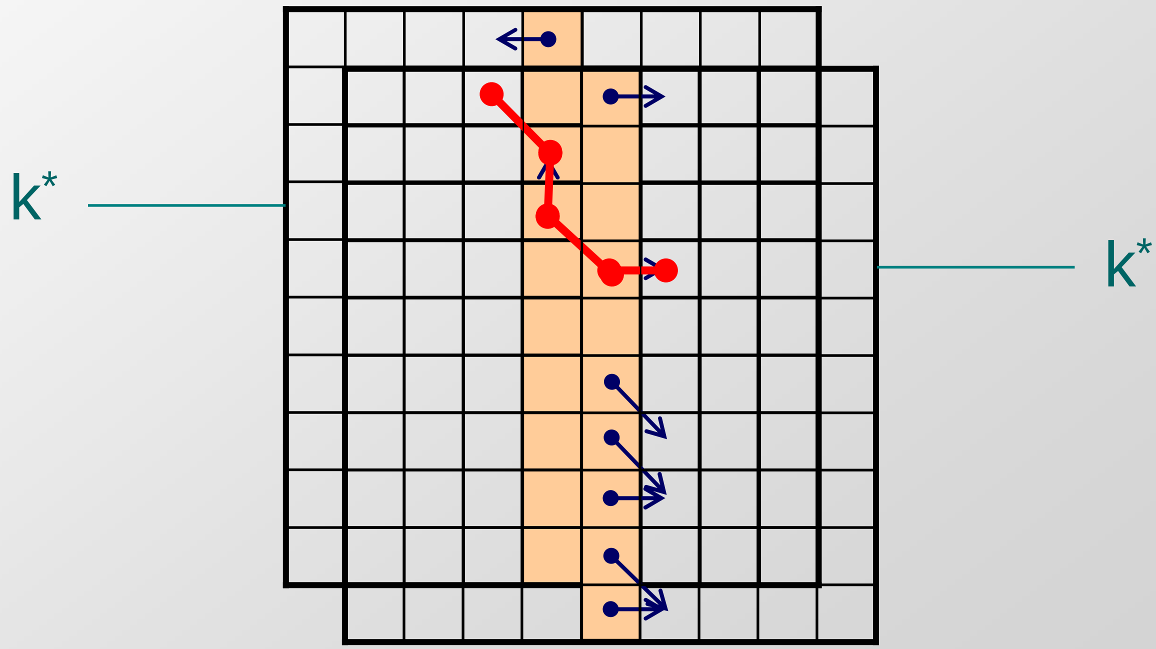
PLUS the backpointers





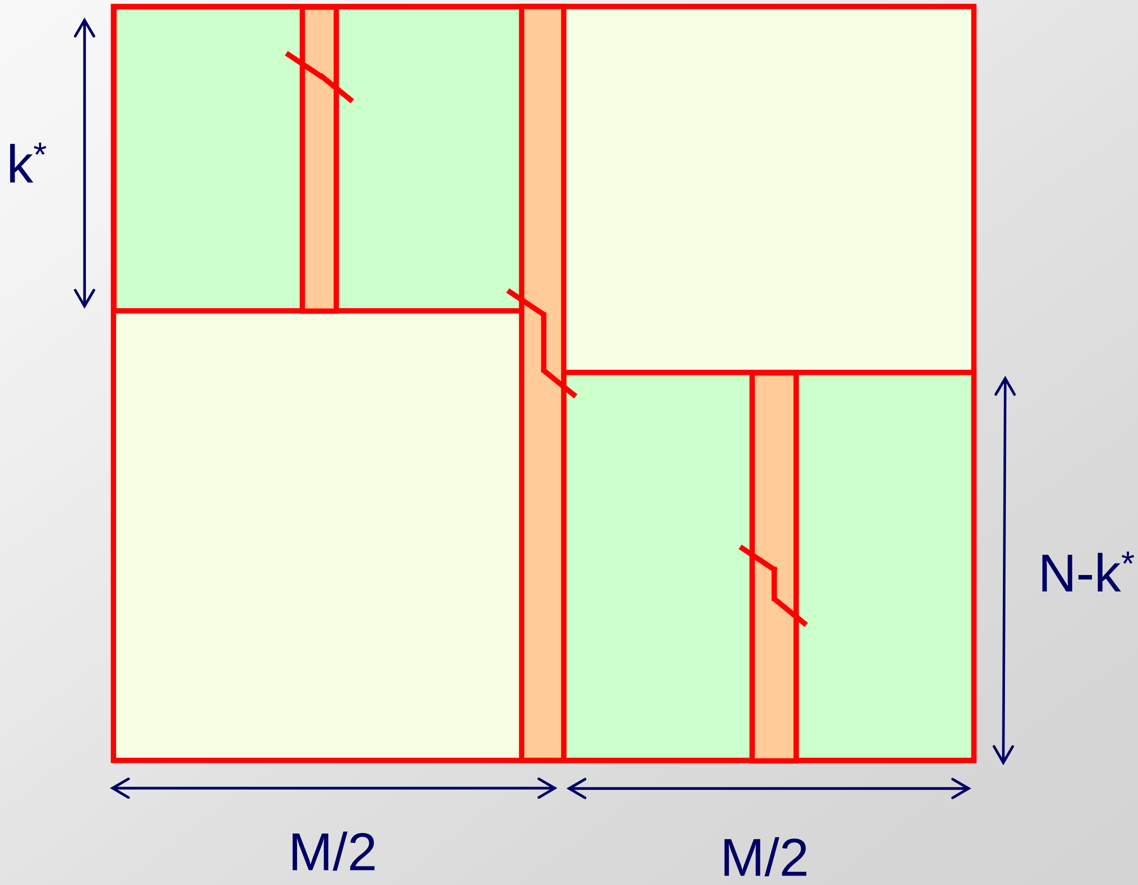
Linear-space alignment

- Now, we can find k^* maximizing $F(M/2, k) + F^r(M/2, N-k)$
- Also, we can trace the path exiting column M/2 from k^*



Linear-space alignment

- Iterate this procedure to the left and right!



Linear-space alignment

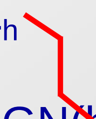
Hirschberg's Linear-space algorithm:

MEMALIGN(l, l', r, r'): (aligns $x_l \ldots x_{l'}$ with $y_r \ldots y_{r'}$)

1. Let $h = \lceil (l'-l)/2 \rceil$
2. Find in Time $O((l' - l) \times (r'-r))$, Space $O(r'-r)$
   the optimal path, L_h, entering column h-1, exiting column h
   Let k_1 = pos'n at column h – 2 where L_h enters
   k_2 = pos'n at column h + 1 where L_h exits

3. MEMALIGN(l, h-2, r, k_1)

4. Output L_h



5. MEMALIGN(h+1, l', k_2, r')

Top level call: MEMALIGN(1, M, 1, N)

Linear-space alignment

Time, Space analysis of Hirschberg's algorithm:

To compute optimal path at middle column,

For box of size $M \times N$,

Space: $2N$

Time: cMN, for some constant c

Then, left, right calls cost $c(M/2 \times k^* + M/2 \times (N-k^*)) = cMN/2$

All recursive calls cost

Total Time: $cMN + cMN/2 + cMN/4 + \ldots.. = 2cMN = O(MN)$

Total Space: $O(N)$ for computation,

$O(N+M)$ to store the optimal alignment

The Four-Russian Algorithm

A useful speedup of Dynamic Programming

Main Observation

Within a rectangle of the DP matrix,
values of D depend only
on the values of A, B, C,
and substrings $x_{l...l'}$, $y_{r...r'}$

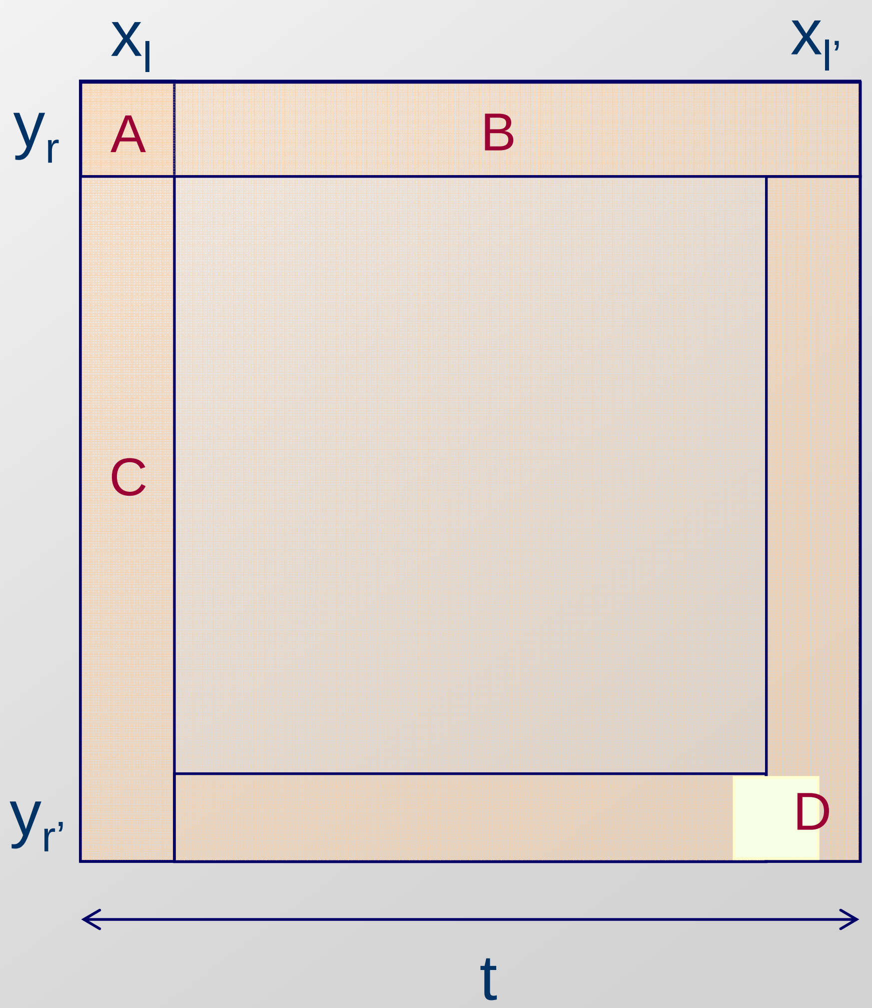
Definition:

A t-block is a t $\times$ t square of the DP matrix

Idea:

Divide matrix in t-blocks,
Precompute t-blocks

Speedup: O(t)



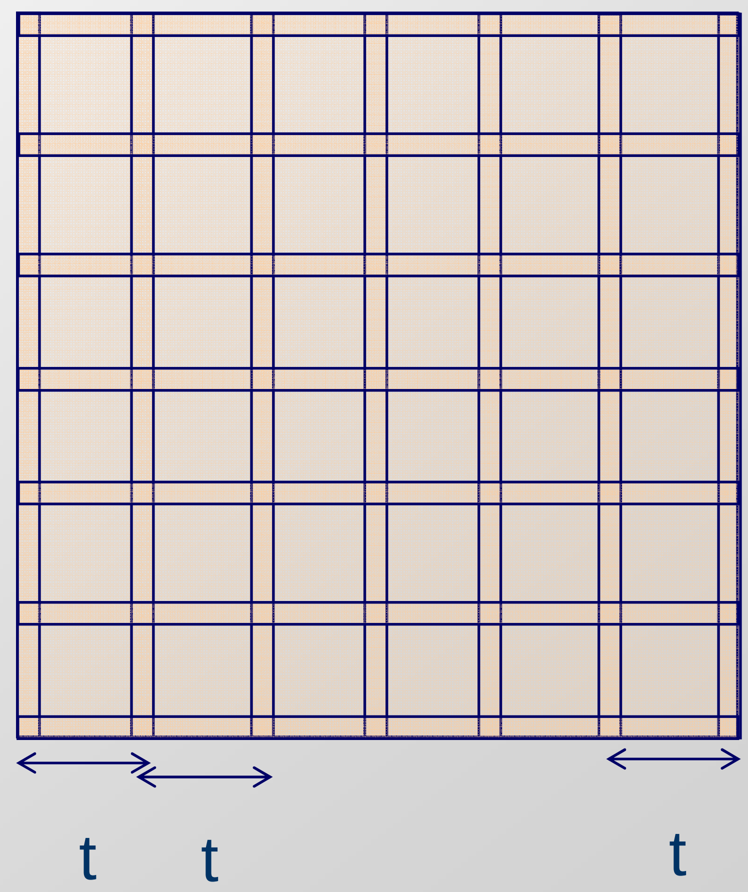
The Four-Russian Algorithm

<u>Main structure of the algorithm:</u>

- Divide N×N DP matrix into K×K $\log_2 N$-blocks that overlap by 1 column & 1 row
- For i = 1……K
- For j = 1……K
- Compute $D_{i,j}$ as a function of

$A_{i,j}$, $B_{i,j}$, $C_{i,j}$, $x[l_i \ldots l'_i]$, $y[r_j \ldots r'_j]$

<u>Time:</u> $O(N^2 / \log^2 N)$
times the cost of step 4



The Four-Russian Algorithm

Another observation:

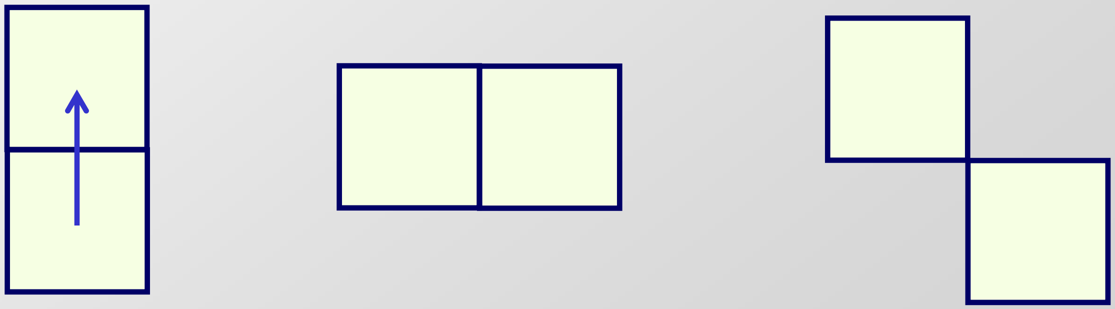
(Assume m = 0, s = 1, d = 1)

Lemma. Two adjacent cells of F(.,.) differ by at most 1

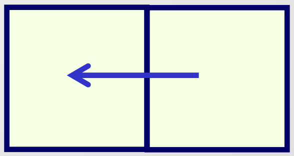
Gusfield's book covers case where m = 0,

called the *edit distance* (p. 216):

minimum # of substitutions + gaps to transform one string to another



The Four-Russian Algorithm



Proof of Lemma:

1. Same row:
   a. $F(i, j) - F(i-1, j) \le +1$

      At worst, one more gap:

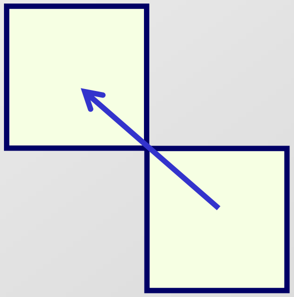
      $x_1 \ldots\ldots x_{i-1}\ x_i$
      $y_1 \ldots\ldots y_j\ \ -$

   b. $F(i, j) - F(i-1, j) \ge -1$

F(i, j)	F(i – 1, j – 1)	F(i, j) – F(i – 1, j – 1)
$x_1 \ldots\ldots x_{i-1}\ x_i$ $y_1 \ldots\ldots y_{a-1} y_a\ y_{a+1} \ldots y_j$	$x_1 \ldots\ldots x_{i-1}\ -$ $y_1 \ldots\ldots y_{a-1} y_a\ y_{a+1} \ldots y_j$	≥ -1
$x_1 \ldots\ldots x_{i-1}\ x_i$ $y_1 \ldots\ldots y_{a-1} - y_a \ldots y_j$	$x_1 \ldots\ldots x_{i-1}$ $y_1 \ldots\ldots y_{a-1} y_a \ldots y_j$	$+1$

2. Same column: similar argument

The Four-Russian Algorithm



Proof of Lemma:

3. Same diagonal:

   a. $F(i, j) - F(i-1, j-1) \le +1$
   At worst, one additional mismatch in F(i, j)

   b. $F(i, j) - F(i-1, j-1) \ge -1$

F(i, j)	F(i – 1, j – 1)	F(i, j) – F(i – 1, j – 1)
$x_1 \ldots\ldots x_{i-1}\ x_i$ $\quad\quad\quad\quad\quad |$ $y_1 \ldots\ldots y_{i-1}\ y_j$	$x_1 \ldots\ldots x_{i-1}$ $y_1 \ldots\ldots y_{j-1}$	$\ge$-1
$x_1 \ldots\ldots x_{i-1}\ x_i$ $y_1 \ldots\ldots y_{a-1} - y_a \ldots y_j$	$x_1 \ldots\ldots x_{i-1}$ $y_1 \ldots\ldots y_{a-1} y_a \ldots y_j$	+1

The Four-Russian Algorithm

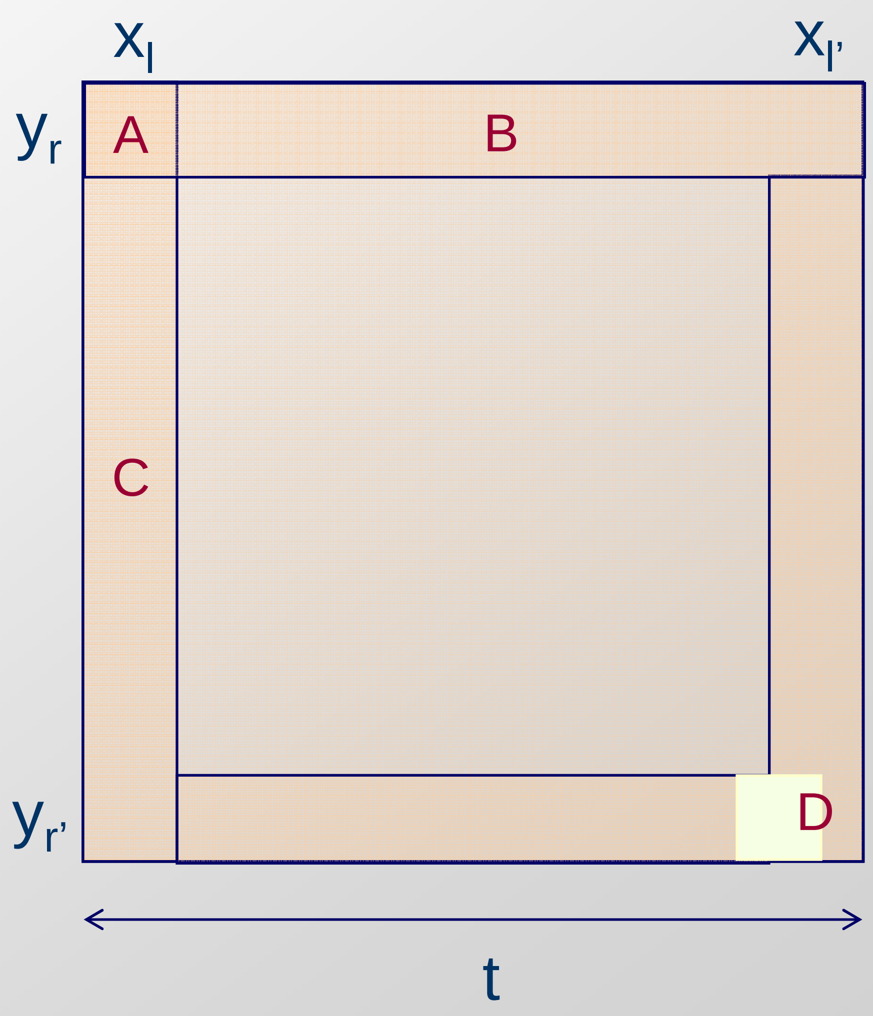
Definition:

The offset vector is a t-long vector of values from {-1, 0, 1}, where the first entry is 0

If we know the value at A, and the top row, left column offset vectors,

and $x_l \ldots\ldots x_{l'}$, $y_r \ldots\ldots y_{r'}$,

Then we can find D



The Four-Russian Algorithm

Example:

x = AACT

y = CACT

A A C T

0 1 -1 -1

5	6	5	4
5	5	6	5
4	5	5	6
5	5	6	5

C 0

A 0

C -1

T 1

0

1

1

-1

0 0 1 -1

The Four-Russian Algorithm

<u>Example:</u>

x = AACT

y = CACT

		A	A	C	T	
		0	1	-1	-1	
C	0	1	2	1	0	0
A	0	1	1	2	1	1
C	-1	0	1	1	2	1
T	1	1	1	2	1	-1
		0	0	1	-1	

The Four-Russian Algorithm

Definition:

The offset function of a t-block

is a function that for any

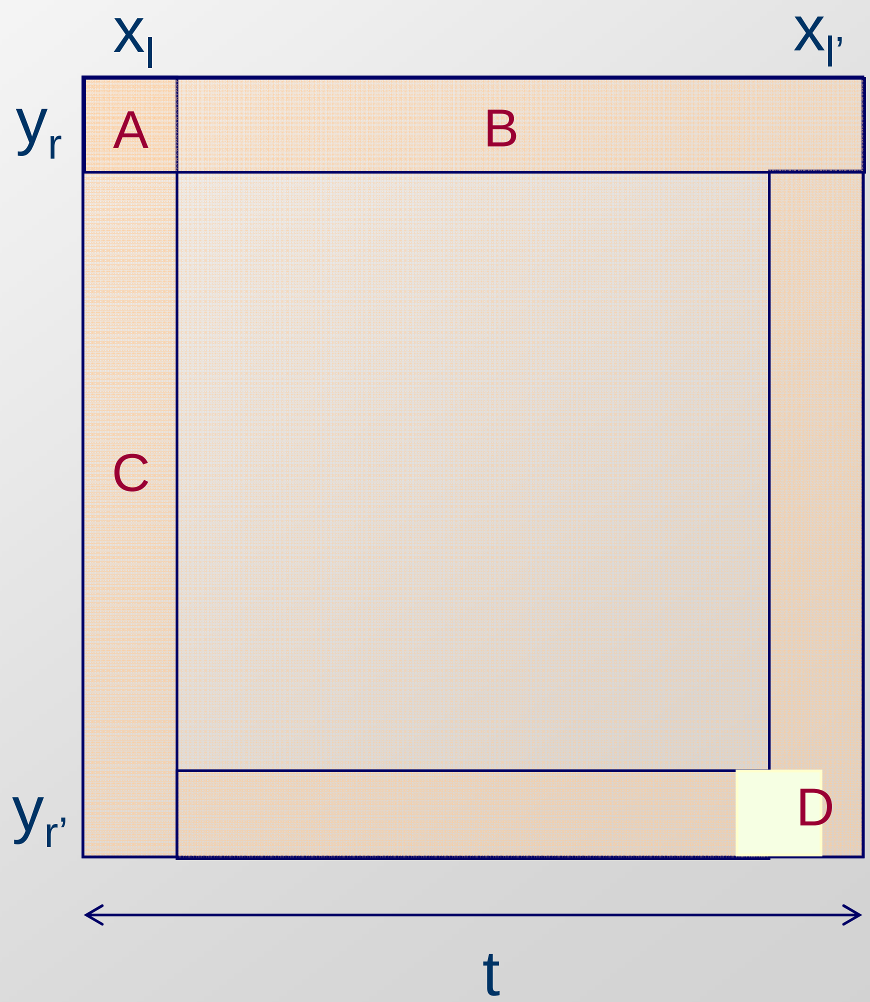
given offset vectors

of top row, left column,

and $x_l \ldots\ldots x_{l'}$, $y_r \ldots\ldots y_{r'}$,

produces offset vectors

of bottom row, right column



The Four-Russian Algorithm

We can pre-compute the offset function:

$3^{2(t-1)}$ possible input offset vectors

4^{2t} possible strings $x_l \ldots\ldots x_{l'}$, $y_r \ldots\ldots y_{r'}$

Therefore $3^{2(t-1)} \times 4^{2t}$ values to pre-compute

We can keep all these values in a table, and look up in linear time,
or in O(1) time if we assume
constant-lookup RAM for log-sized inputs

The Four-Russian Algorithm

Four-Russians Algorithm: (Arlazarov, Dinic, Kronrod, Faradzev)

1. Cover the DP table with t-blocks

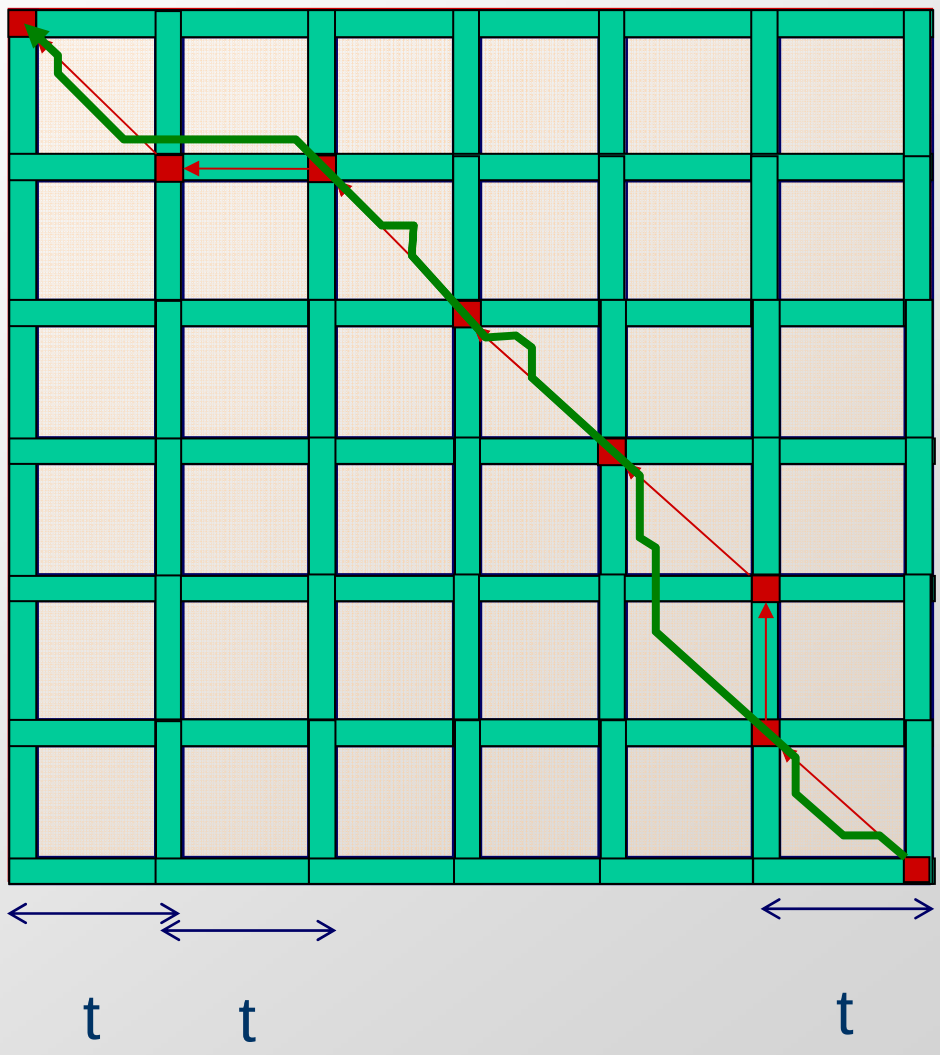
2. Initialize values F(.,.) in first row & column

3. Row-by-row, use offset values at leftmost column and top row of each block, to find offset values at rightmost column and bottom row

4. Let Q = total of offsets at row N

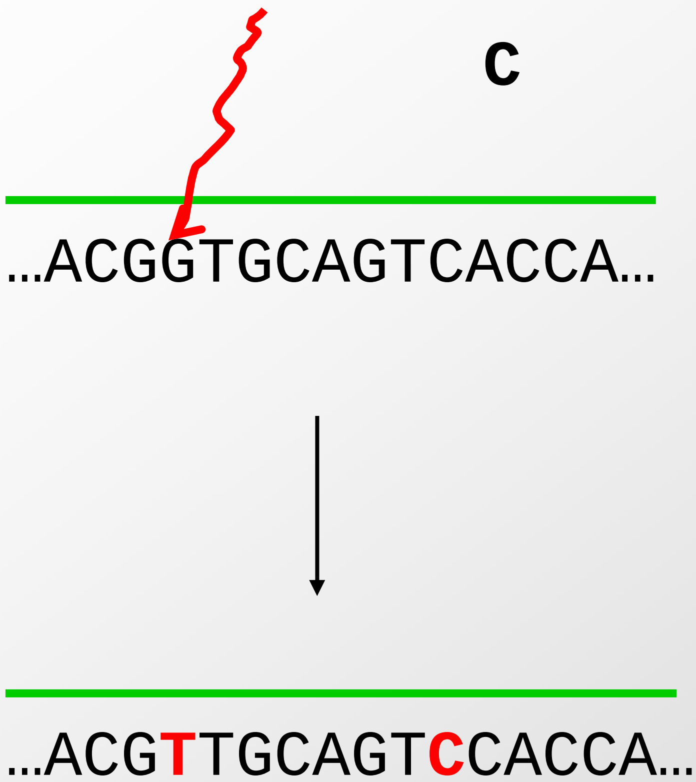
   F(N, N) = Q + F(N, 0)

The Four-Russian Algorithm



Evolution at the DNA level

Sequence Changes



Computing best alignment

- In absence of gaps

Sequence Alignment

AGGCTATCACCTGACCTCCAGGCCGATGCCC
TAGCTATCACGACCGCGGTCGATTTGCCCGAC

-AGGCTATCACCTGACCTCCAGGCCGA--TGCCC---
TAG-CTATCAC--GACCGC--GGTCGATTTGCCCGAC

Definition

Given two strings $x = x_1x_2...x_M$, $y = y_1y_2…y_N$,

an alignment is an assignment of gaps to positions 0,…, M in x, and 0,…, N in y, so as to line up each letter in one sequence with either a letter, or a gap in the other sequence

Scoring Function

- Sequence edits:

AGGCCTC

  - Mutations

AGGACTC

  - Insertions

AGGGCCTC

  - Deletions

AGG.CTC

Scoring Function:

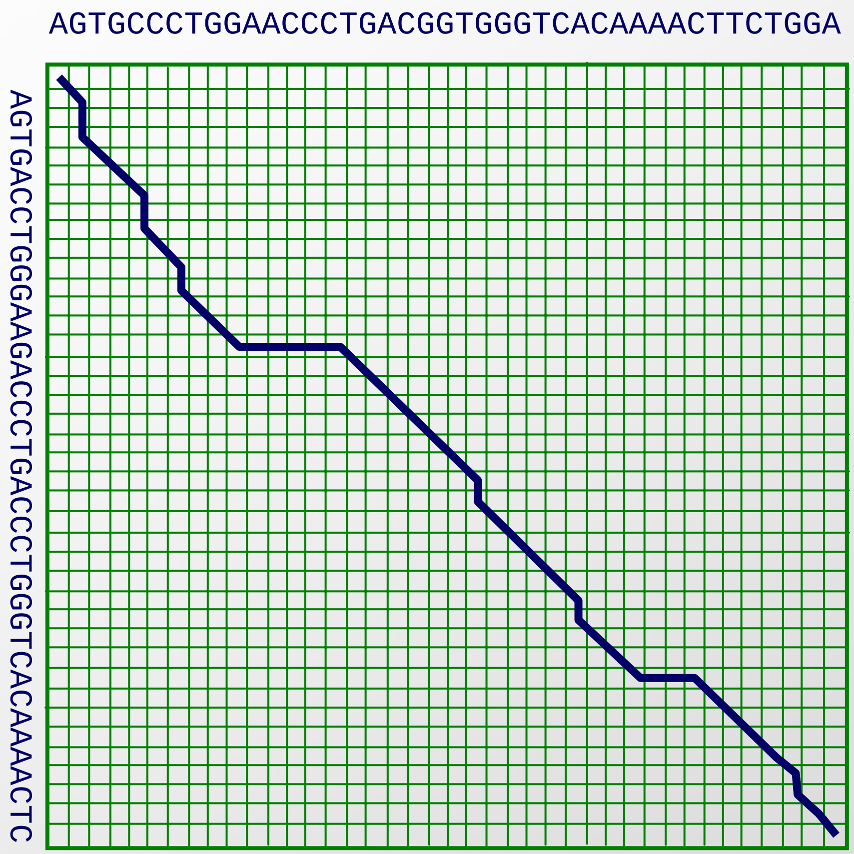
Match: +m

Mismatch: -s

Gap: -d

Score $F = (\#\text{ matches}) \times m - (\#\text{ mismatches}) \times s - (\#\text{gaps}) \times d$

How do we compute the best alignment?



Too many possible alignments:

$$O(\ 2^{M+N})$$

Alignment is additive

Observation:

The score of aligning $x_1 \ldots\ldots x_M$

$y_1 \ldots\ldots y_N$

is additive

Say that $x_1 \ldots x_i$ $x_{i+1} \ldots x_M$

aligns to $y_1 \ldots y_j$ $y_{j+1} \ldots y_N$

The two scores add up:

$$F(x[1:M], y[1:N]) = F(x[1:i], y[1:j]) + F(x[i+1:M], y[j+1:N])$$

Dynamic Programming

- We will now describe a dynamic programming algorithm

Suppose we wish to align

$$x_1 \ldots\ldots x_M$$

$$y_1 \ldots\ldots y_N$$

Let

F(i,j) = optimal score of aligning

$$x_1 \ldots\ldots x_i$$

$$y_1 \ldots\ldots y_j$$

Dynamic Programming (cont'd)

Notice three possible cases:

1. x_i aligns to y_j

   $x_1 \ldots\ldots x_{i-1} \quad x_i$

   $y_1 \ldots\ldots y_{j-1} \quad y_j$

   $$F(i,j) = F(i-1, j-1) + \begin{cases} m, \text{ if } x_i = y_j \\ -s, \text{ if not} \end{cases}$$

2. x_i aligns to a gap

   $x_1 \ldots\ldots x_{i-1} \quad x_i$

   $y_1 \ldots\ldots y_j \quad -$

   $$F(i,j) = F(i-1, j) - d$$

3. y_j aligns to a gap

   $x_1 \ldots\ldots x_i \quad -$

   $y_1 \ldots\ldots y_{j-1} \quad y_j$

   $$F(i,j) = F(i, j-1) - d$$

Dynamic Programming (cont'd)

- How do we know which case is correct?

<u>**Inductive assumption:**</u>

F(i, j-1), F(i-1, j), F(i-1, j-1) are optimal

Then,

$$F(i, j) = \max \begin{cases} F(i-1, j-1) + s(x_i, y_j) \\ F(i-1, j) - d \\ F(i, j-1) - d \end{cases}$$

Where $s(x_i, y_j) = m$, if $x_i = y_j$; $-s$, if not

Example

x = AGTA
y = ATA

m = 1
s = -1
d = -1

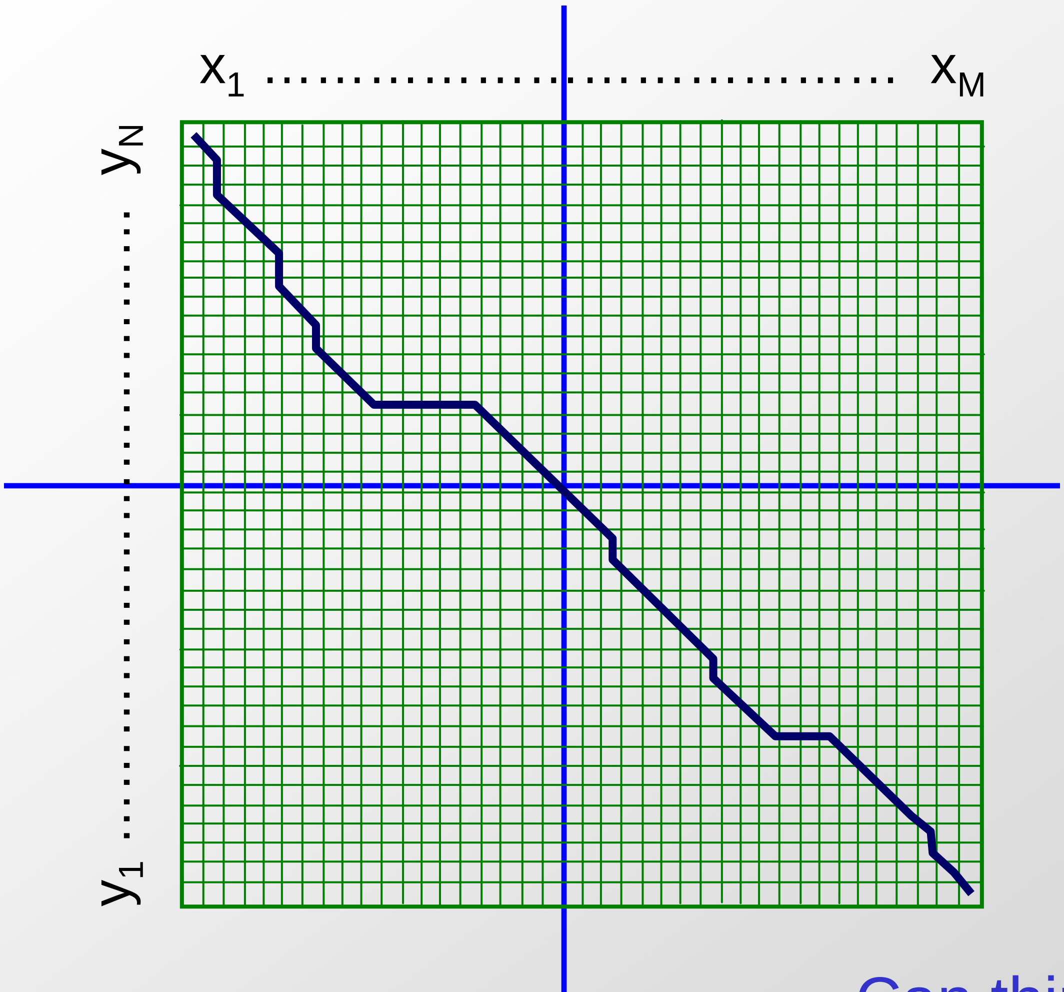
F(i,j)		i = 0	1	2	3	4
			A	G	T	A
j = 0		0	-1	-2	-3	-4
1	A	-1	1	0	-1	-2
2	T	-2	0	0	1	0
3	A	-3	-1	-1	0	2

Optimal Alignment:

F(4,3) = 2

AGTA
A - TA

The Needleman-Wunsch Matrix



Every nondecreasing path

from (0,0) to (M, N)

corresponds to
an alignment
of the two sequences

Can think of it as a
divide-and-conquer algorithm

The Needleman-Wunsch Algorithm

1. Initialization.
   a. $F(0, 0) = 0$
   b. $F(0, j) = -j \times d$
   c. $F(i, 0) = -i \times d$

2. Main Iteration. Filling-in partial alignments
   a. For each $i = 1 \ldots\ldots M$
      For each $j = 1 \ldots\ldots N$

$$F(i, j) = \max \begin{cases} F(i-1,j-1) + s(x_i, y_j) & \text{[case 1]} \\ F(i-1, j) - d & \text{[case 2]} \\ F(i, j-1) - d & \text{[case 3]} \end{cases}$$

$$Ptr(i,j) = \begin{cases} \text{DIAG,} & \text{if [case 1]} \\ \text{LEFT,} & \text{if [case 2]} \\ \text{UP,} & \text{if [case 3]} \end{cases}$$

3. Termination. F(M, N) is the optimal score, and from Ptr(M, N) can trace back optimal alignment

Performance

- Time:

  $O(NM)$

- Space:

  $O(NM)$

- Later we will cover more efficient methods

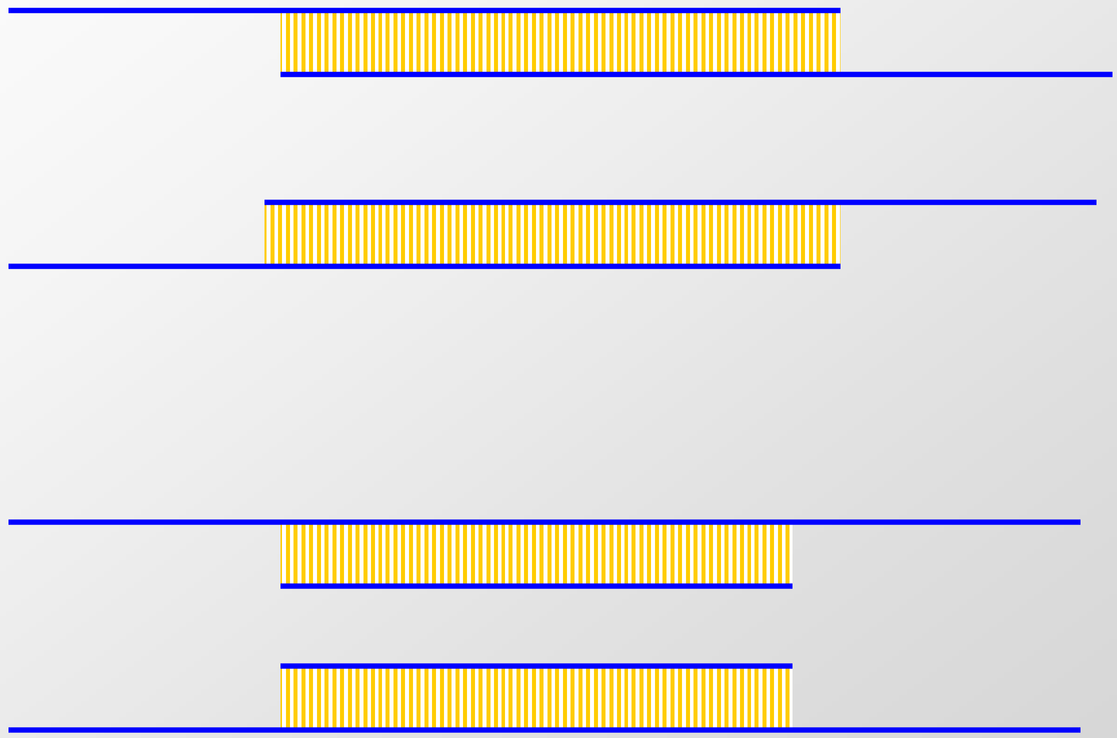
A variant of the basic algorithm:

- Maybe it is OK to have an unlimited # of gaps in the beginning and end:

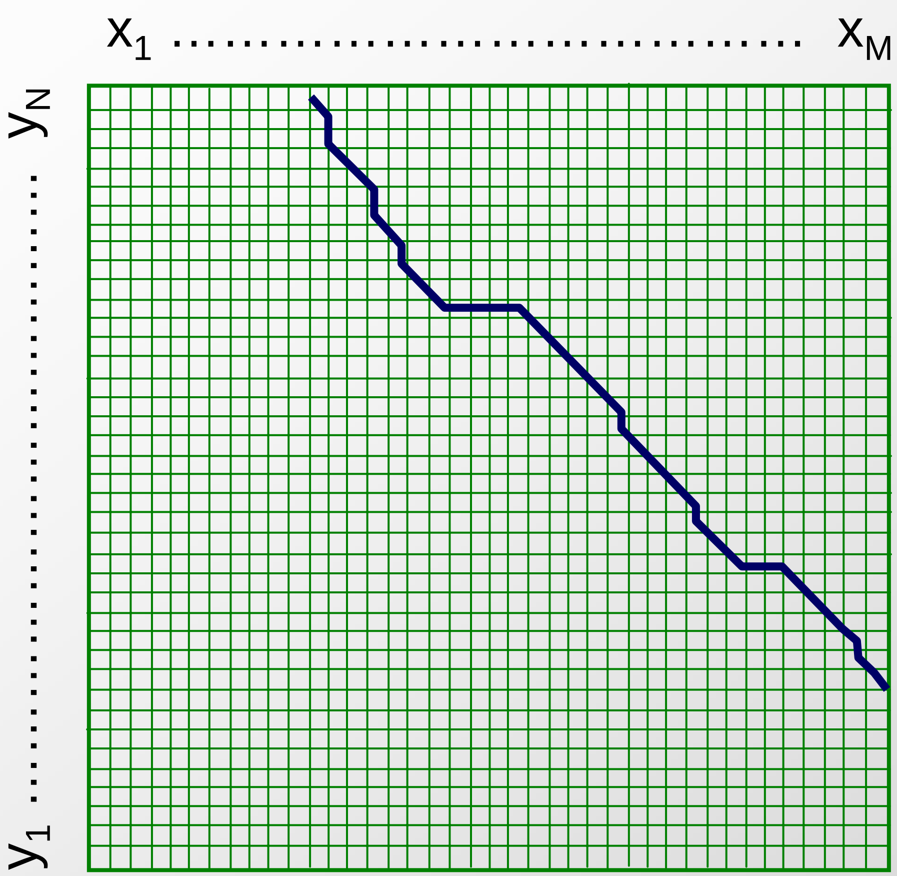
```
----------CTATCACCTGACCTCCAGGCCGATGCCCCTTCCGGC
GCGAGTTCATCTATCAC--GACCGC--GGTCG--------------
```

- Then, we don't want to penalize gaps in the ends

Different types of overlaps



The Overlap Detection variant



Changes:

1. Initialization

   For all i, j,

   $F(i, 0) = 0$

   $F(0, j) = 0$

2. Termination

$$F_{OPT} = \max \begin{cases} \max_i F(i, N) \\ \max_j F(M, j) \end{cases}$$

The local alignment problem

Given two strings $x = x_1\ldots\ldots x_M$,
$y = y_1\ldots\ldots y_N$

Find substrings x', y' whose similarity
(optimal global alignment value)
is maximum

e.g. x = aaaacccccggggg
y = cccgggaaccaacc

Why local alignment

- Genes are shuffled between genomes
- Portions of proteins (domains) are often conserved

Image removed due to copyright restrictions.

Cross-species genome similarity

- 98% of genes are conserved between any two mammals
- >70% average similarity in protein sequence

```
hum_a : GTTGACAATAGAGGGTCTGGCAGAGGCTC--------------------- @ 57331/400001
mus_a : GCTGACAATAGAGGGGCTGGCAGAGGCTC--------------------- @ 78560/400001
rat_a : GCTGACAATAGAGGGGCTGGCAGAGACTC--------------------- @ 112658/369938
fug_a : TTTGTTGATGGGGAGCGTGCATTAATTTCAGGCTATTGTTAACAGGCTCG @ 36008/68174

hum_a : CTGGCCGCGGTGCGGAGCGTCTGGAGCGGAGCACGCGCTGTCAGCTGGTG @ 57381/400001
mus_a : CTGGCCCCGGTGCGGAGCGTCTGGAGCGGAGCACGCGCTGTCAGCTGGTG @ 78610/400001
rat_a : CTGGCCCCGGTGCGGAGCGTCTGGAGCGGAGCACGCGCTGTCAGCTGGTG @ 112708/369938
fug_a : TGGGCCGAGGTGTTGGATGGCCTGAGTGAAGCACGCGCTGTCAGCTGGCG @ 36058/68174

hum_a : AGCGCACTCTCCTTTCAGGCAGCTCCCCGGGGAGCTGTGCGGCCACATTT @ 57431/400001
mus_a : AGCGCACTCG-CTTTCAGGCCGCTCCCCGGGGAGCTGAGCGGCCACATTT @ 78659/400001
rat_a : AGCGCACTCG-CTTTCAGGCCGCTCCCCGGGGAGCTGCGCGGCCACATTT @ 112757/369938
fug_a : AGCGCTCGCG------------------------AGTCCCTGCCGTGTCC @ 36084/68174

hum_a : AACACCATCATCACCCCTCCCCGGCCTCCTCAACCTCGGCCTCCTCCTCG @ 57481/400001
mus_a : AACACCGTCGTCA-CCCTCCCCGGCCTCCTCAACCTCGGCCTCCTCCTCG @ 78708/400001
rat_a : AACACCGTCGTCA-CCCTCCCCGGCCTCCTCAACCTCGGCCTCCTCCTCG @ 112806/369938
fug_a : CCGAGGACCCTGA------------------------------------- @ 36097/68174
```

“atoh” enhancer in human, mouse, rat, fugu fish

The Smith-Waterman algorithm

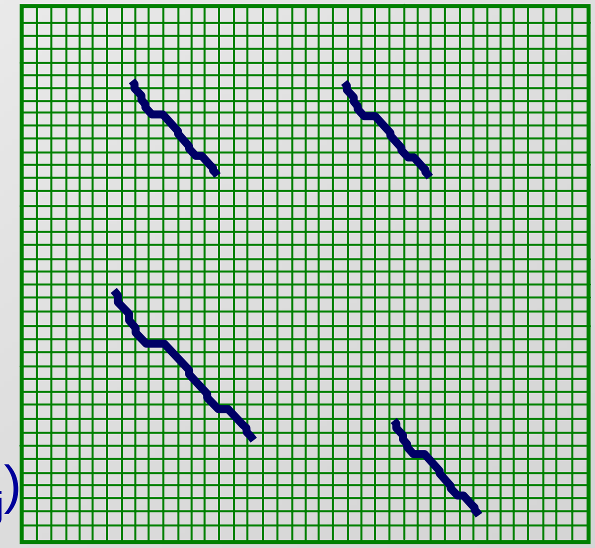
Idea: Ignore badly aligning regions

Modifications to Needleman-Wunsch:

Initialization: $F(0, j) = F(i, 0) = 0$

Iteration:

$$F(i, j) = \max \begin{cases} 0 \\ F(i-1, j) - d \\ F(i, j-1) - d \\ F(i-1, j-1) + s(x_i, y_j) \end{cases}$$



The Smith-Waterman algorithm

Termination:

1. If we want the best local alignment…

$$F_{OPT} = \max_{i,j} F(i, j)$$

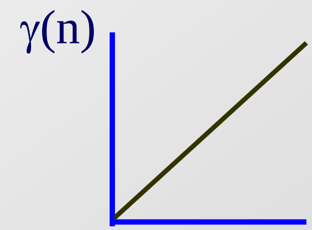
2. If we want all local alignments scoring > t

For all i, j find F(i, j) > t, and trace back

Scoring the gaps more accurately

Current model:

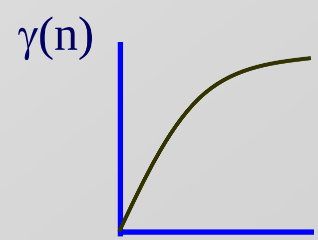
Gap of length n
incurs penalty $n \times d$



However, gaps usually occur in bunches

Convex gap penalty function:

$\gamma(n)$:
for all n, $\gamma(n + 1) - \gamma(n) \leq \gamma(n) - \gamma(n - 1)$



General gap dynamic programming

Initialization: same

Iteration:

$$F(i, j) = \max \begin{cases} F(i-1, j-1) + s(x_i, y_j) \\ \max_{k=0\dots i-1} F(k,j) - \gamma(i-k) \\ \max_{k=0\dots j-1} F(i,k) - \gamma(j-k) \end{cases}$$

Termination: same

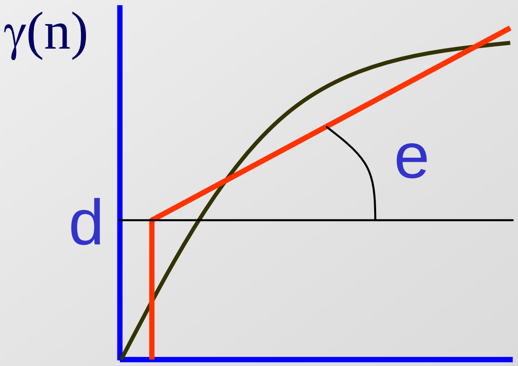
Running Time: $O(N^2M)$ (assume N>M)

Space: $O(NM)$

Compromise: affine gaps

$\gamma(n) = d + (n - 1) \times e$

| |
gap open   gap extend



To compute optimal alignment,

At position i,j, need to "remember" best score if gap is open
best score if gap is not open

F(i, j): score of alignment $x_1 \ldots x_i$ to $y_1 \ldots y_j$
if x_i aligns to y_j

G(i, j): score **if** x_i, or y_j, aligns to a gap

Needleman-Wunsch with affine gaps

Initialization:

$F(i, 0) = d + (i - 1) \times e$

$F(0, j) = d + (j - 1) \times e$

Iteration:

$$F(i, j) = \max \begin{cases} F(i-1, j-1) + s(x_i, y_j) \\ G(i-1, j-1) + s(x_i, y_j) \end{cases}$$

$$G(i, j) = \max \begin{cases} F(i-1, j) - d \\ F(i, j-1) - d \\ G(i, j-1) - e \\ G(i-1, j) - e \end{cases}$$

Termination: same

Sequence Alignment

AGGCTATCACCTGACCTCCAGGCCGATGCCC
TAGCTATCACGACCGCGGTCGATTTGCCCGAC

-AGGCTATCACCTGACCTCCAGGCCGA--TGCCC---
TAG-CTATCAC--GACCGC--GGTCGATTTGCCCGAC

Definition

Given two strings $x = x_1x_2...x_M$, $y = y_1y_2…y_N$,

an alignment is an assignment of gaps to positions 0,…, M in x, and 0,…, N in y, so as to line up each letter in one sequence with either a letter, or a gap in the other sequence

Scoring Function

- Sequence edits:
  - Mutations
  - Insertions
  - Deletions

AGGCCTC

AGGACTC

AGGGCCTC

AGG.CTC

Scoring Function:

Match: +m

Mismatch: -s

Gap: -d

Score F = (# matches) × m - (# mismatches) × s – (#gaps) × d

The Needleman-Wunsch Algorithm

1. Initialization.
   a. $F(0, 0) = 0$
   b. $F(0, j) = - j \times d$
   c. $F(i, 0) = - i \times d$

2. Main Iteration. Filling-in partial alignments
   a. For each $i = 1 \ldots\ldots M$

      For each $j = 1 \ldots\ldots N$

$$F(i, j) = \max \begin{cases} F(i-1, j-1) + s(x_i, y_j) & \text{[case 1]} \\ F(i-1, j) - d & \text{[case 2]} \\ F(i, j-1) - d & \text{[case 3]} \end{cases}$$

$$Ptr(i,j) = \begin{cases} \text{DIAG,} & \text{if [case 1]} \\ \text{LEFT,} & \text{if [case 2]} \\ \text{UP,} & \text{if [case 3]} \end{cases}$$

3. Termination. F(M, N) is the optimal score, and from Ptr(M, N) can trace back optimal alignment

The Smith-Waterman algorithm

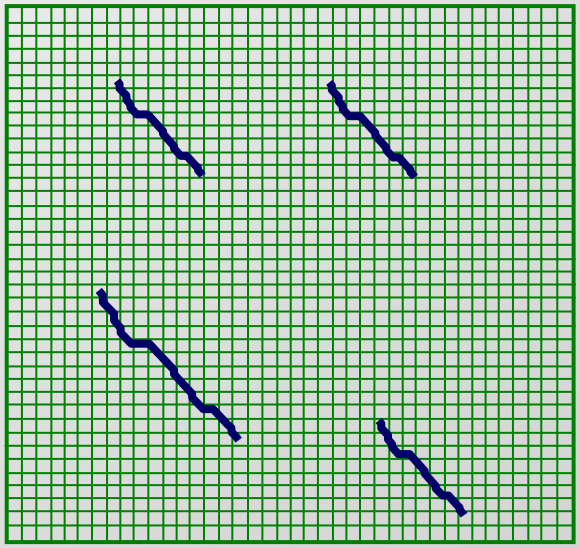
Idea: Ignore badly aligning regions

Modifications to Needleman-Wunsch:

Initialization: $F(0, j) = F(i, 0) = 0$

Iteration:

$$F(i, j) = \max \begin{cases} 0 \\ F(i-1, j) - d \\ F(i, j-1) - d \\ F(i-1, j-1) + s(x_i, y_j) \end{cases}$$

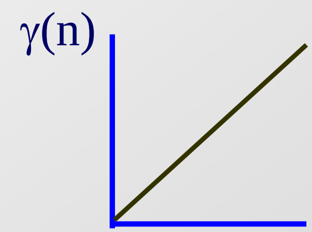


Scoring the gaps more accurately

Simple, linear gap model:

Gap of length n

incurs penalty $n \times d$

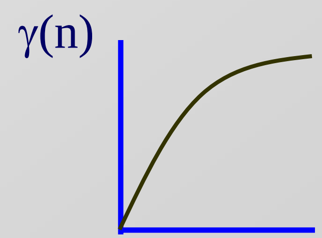


However, gaps usually occur in bunches

Convex gap penalty function:

$\gamma(n)$:

for all n, $\gamma(n + 1) - \gamma(n) \leq \gamma(n) - \gamma(n - 1)$

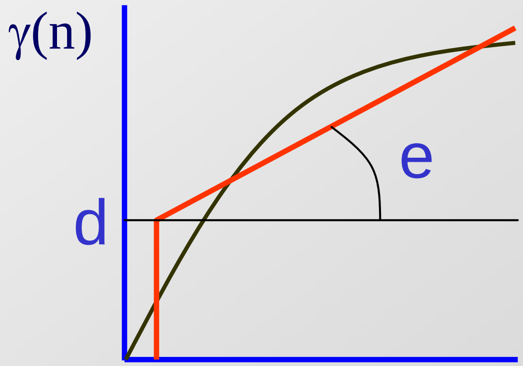


Algorithm: $O(N^3)$ time, $O(N^2)$ space

Compromise: affine gaps

$\gamma(n) = d + (n - 1) \times e$

gap open	gap extend
d	$(n-1) \times e$



To compute optimal alignment,

At position i,j, need to "remember" best score if gap is open
best score if gap is not open

F(i, j): score of alignment $x_1 \ldots x_i$ to $y_1 \ldots y_j$
if x_i aligns to y_j

G(i, j): score **if** x_i, or y_j, aligns to a gap

Needleman-Wunsch with affine gaps

Why do we need two matrices?

- x_i aligns to y_j

  $x_1 \ldots\ldots x_{i-1}\ \ x_i\ \ \ x_{i+1}$

  $y_1 \ldots\ldots y_{j-1}\ \ y_j\ \ \ \ -$

2. x_i aligns to a gap

   $x_1 \ldots\ldots x_{i-1}\ \ x_i\ \ \ x_{i+1}$

   $y_1 \ldots\ldots y_j\ \ldots -\ \ \ \ -$

Add -d

Add -e

Needleman-Wunsch with affine gaps

Initialization:

$F(i, 0) = d + (i - 1) \times e$
$F(0, j) = d + (j - 1) \times e$

Iteration:

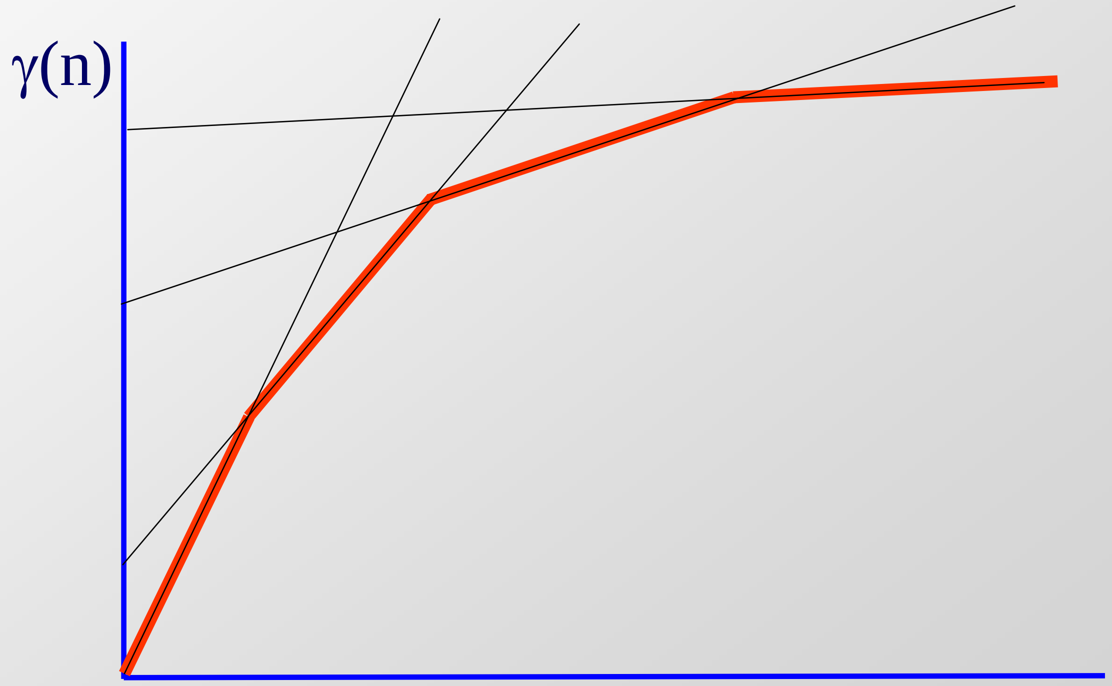
$$F(i, j) = \max \begin{cases} F(i-1, j-1) + s(x_i, y_j) \\ G(i-1, j-1) + s(x_i, y_j) \end{cases}$$

$$G(i, j) = \max \begin{cases} F(i-1, j) - d \\ F(i, j-1) - d \\ G(i, j-1) - e \\ G(i-1, j) - e \end{cases}$$

Termination: same

To generalize a little…

… think of how you would compute optimal alignment with this gap function



….in time O(MN)

Bounded Dynamic Programming

Assume we know that x and y are very similar

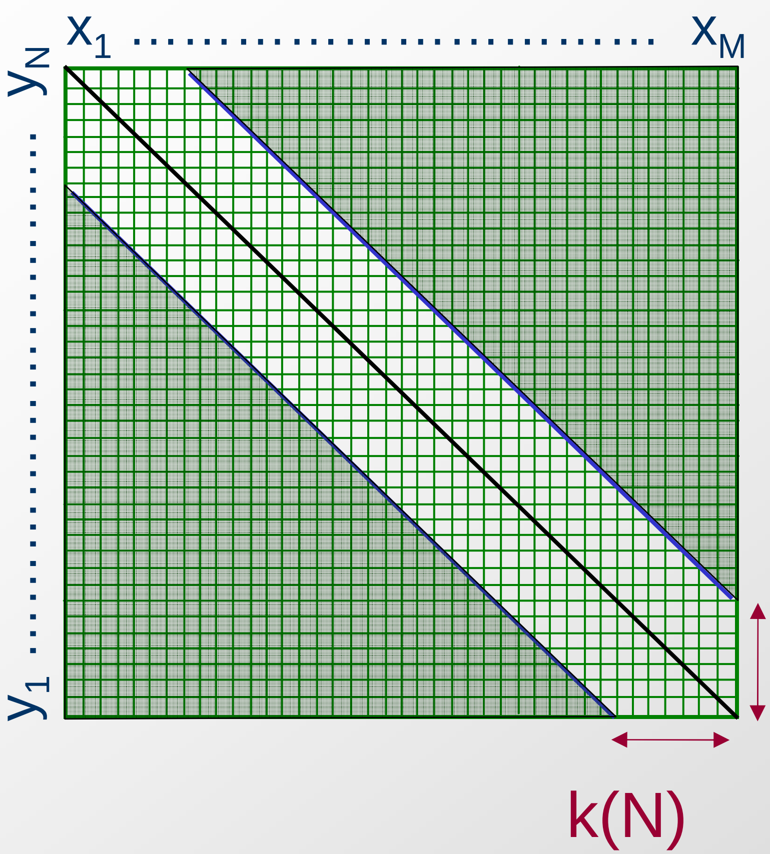
Assumption: # gaps(x, y) < k(N) (say N>M)

Then, $\begin{matrix} x_i \\ | \\ y_j \end{matrix}$ implies $| i - j | < k(N)$

We can align x and y more efficiently:

Time, Space: $O(N \times k(N)) << O(N^2)$

Bounded Dynamic Programming



Initialization:

F(i,0), F(0,j) undefined for i, j > k

Iteration:

For i = 1…M

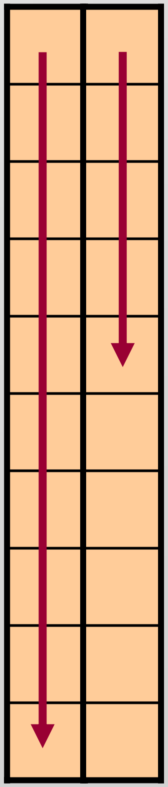
For j = max(1, i – k)…min(N, i+k)

$$F(i, j) = \max \begin{cases} F(i-1, j-1) + s(x_i, y_j) \\ F(i, j-1) - d, \text{ if } j > i - k(N) \\ F(i-1, j) - d, \text{ if } j < i + k(N) \end{cases}$$

Termination: same

Easy to extend to the affine gap case

Linear-Space Alignment



Hirschberg's algortihm

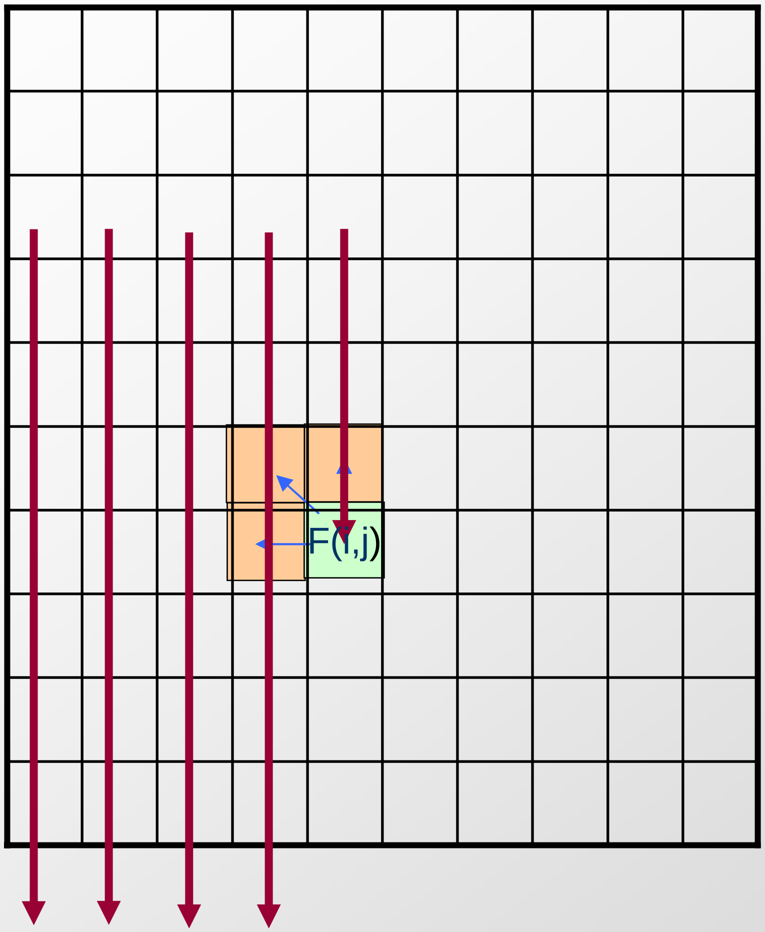
- Longest common subsequence
  - Given sequences $s = s_1\ s_2 \ldots s_m$, $t = t_1\ t_2 \ldots t_n$,
  - Find longest common subsequence $u = u_1 \ldots u_k$

- Algorithm:
  - $$F(i, j) = \max \begin{cases} F(i-1, j) \\ F(i, j-1) \\ F(i-1, j-1) + [1, \text{if } s_i = t_j; 0 \text{ otherwise}] \end{cases}$$

- Hirschberg's algorithm solves this in linear space

Introduction: Compute optimal score

It is easy to compute F(M, N) in linear space



Allocate (column[1])
Allocate (column[2])

For i = 1….M
    If i > 1, then:
        Free(column[i – 2])
        Allocate(column[i])
    For j = 1…N
        F(i, j) = …

Linear-space alignment

To compute both the optimal score **and** the optimal alignment:

Divide & Conquer approach:

Notation:

x^r, y^r: reverse of x, y

E.g. x = accgg;

x^r = ggcca

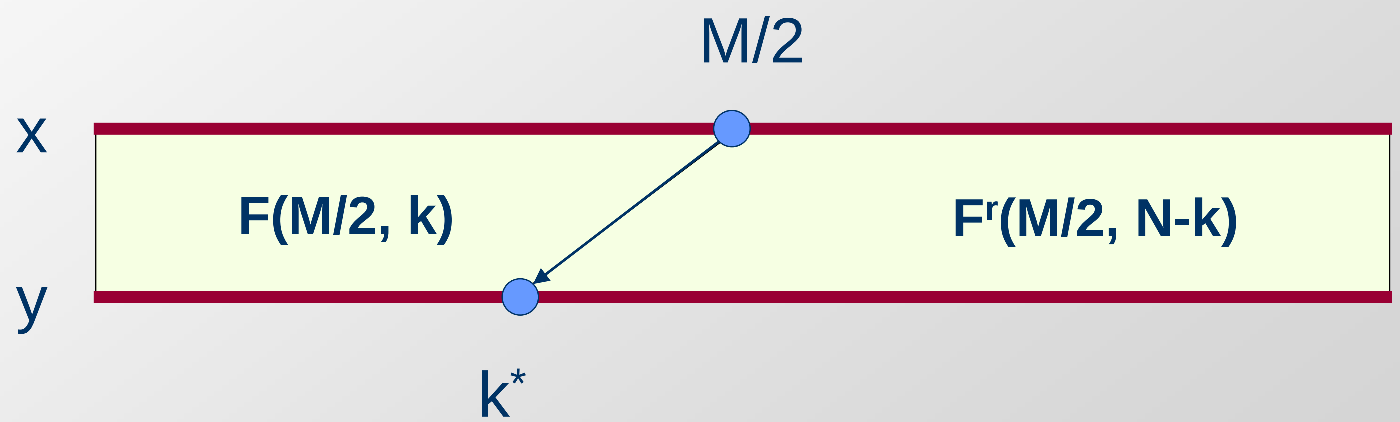
$F^r(i, j)$: optimal score of aligning $x^r_1 \ldots x^r_i$ & $y^r_1 \ldots y^r_j$

same as F(M-i+1, N-j+1)

Linear-space alignment

Lemma:

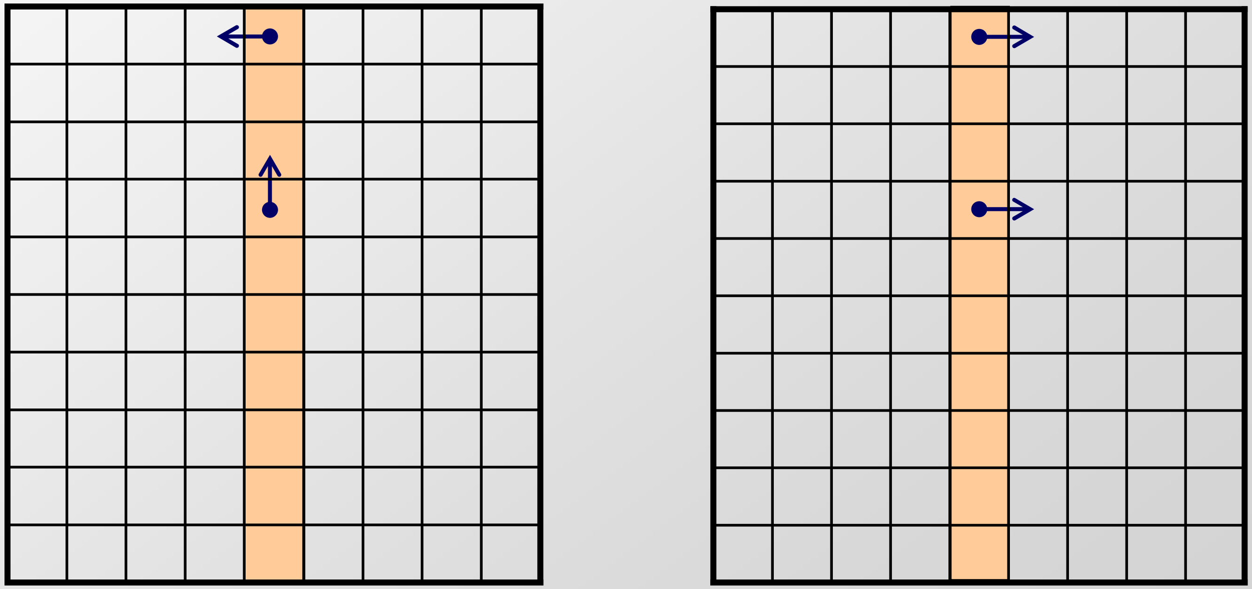
$$F(M, N) = \max_{k=0\ldots N}(F(M/2, k) + F^r(M/2, N-k))$$



Linear-space alignment

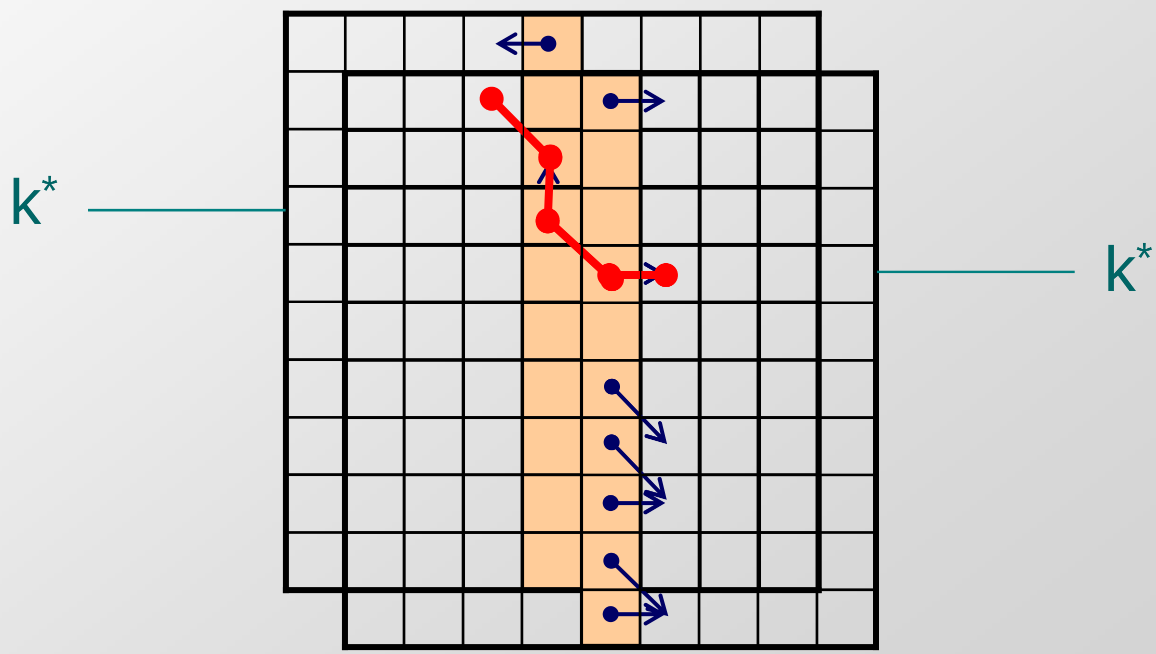
- Now, using 2 columns of space, we can compute for k = 1…M, F(M/2, k), F^r(M/2, N-k)

  PLUS the backpointers



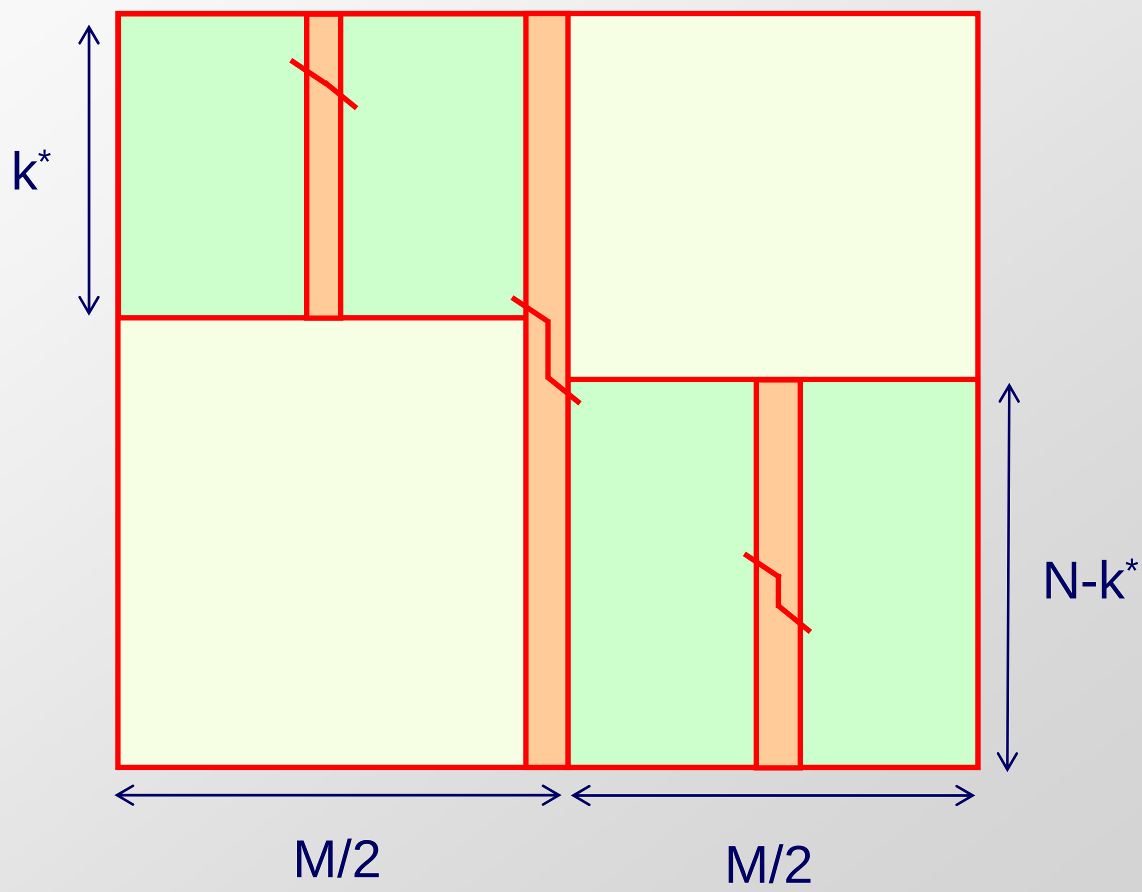
Linear-space alignment

- Now, we can find k^* maximizing $F(M/2, k) + F^r(M/2, N-k)$
- Also, we can trace the path exiting column M/2 from k^*



Linear-space alignment

- Iterate this procedure to the left and right!

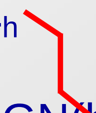


Linear-space alignment

Hirschberg's Linear-space algorithm:

MEMALIGN(l, l', r, r'): (aligns $x_l \ldots x_{l'}$ with $y_r \ldots y_{r'}$)

1. Let h = $\lceil (l'-l)/2 \rceil$
2. Find in Time O((l' – l) × (r'-r)), Space O(r'-r)
   the optimal path, L_h, entering column h-1, exiting column h
   Let k_1 = pos'n at column h – 2 where L_h enters
   k_2 = pos'n at column h + 1 where L_h exits
3. MEMALIGN(l, h-2, r, k_1)
4. Output L_h



5. MEMALIGN(h+1, l', k_2, r')

Top level call: MEMALIGN(1, M, 1, N)

Linear-space alignment

Time, Space analysis of Hirschberg's algorithm:

To compute optimal path at middle column,

For box of size $M \times N$,

Space: 2N

Time: cMN, for some constant c

Then, left, right calls cost $c(M/2 \times k^* + M/2 \times (N-k^*)) = cMN/2$

All recursive calls cost

Total Time: cMN + cMN/2 + cMN/4 + ….. = 2cMN = O(MN)

Total Space: O(N) for computation,

O(N+M) to store the optimal alignment

The Four-Russian Algorithm

A useful speedup of Dynamic Programming

Main Observation

Within a rectangle of the DP matrix,
values of D depend only
on the values of A, B, C,
and substrings $x_{l...l'}$, $y_{r...r'}$

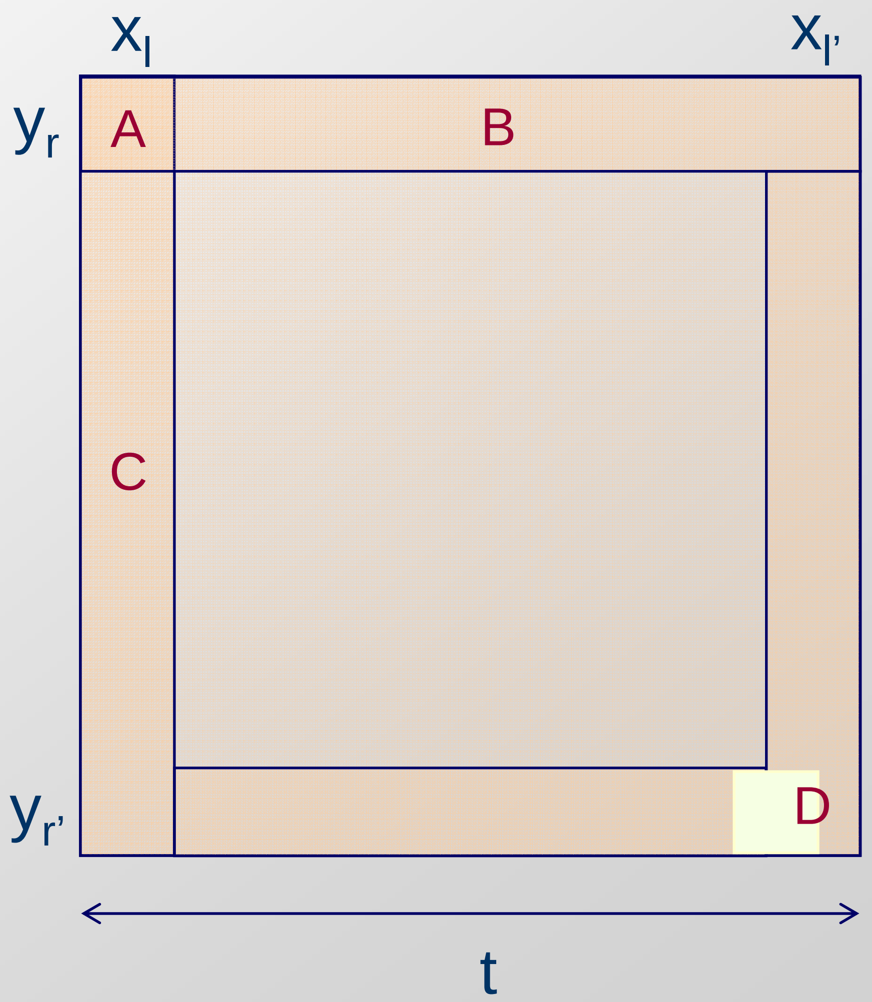
Definition:

A t-block is a t $\times$ t square of the DP matrix

Idea:

Divide matrix in t-blocks,
Precompute t-blocks

Speedup: O(t)



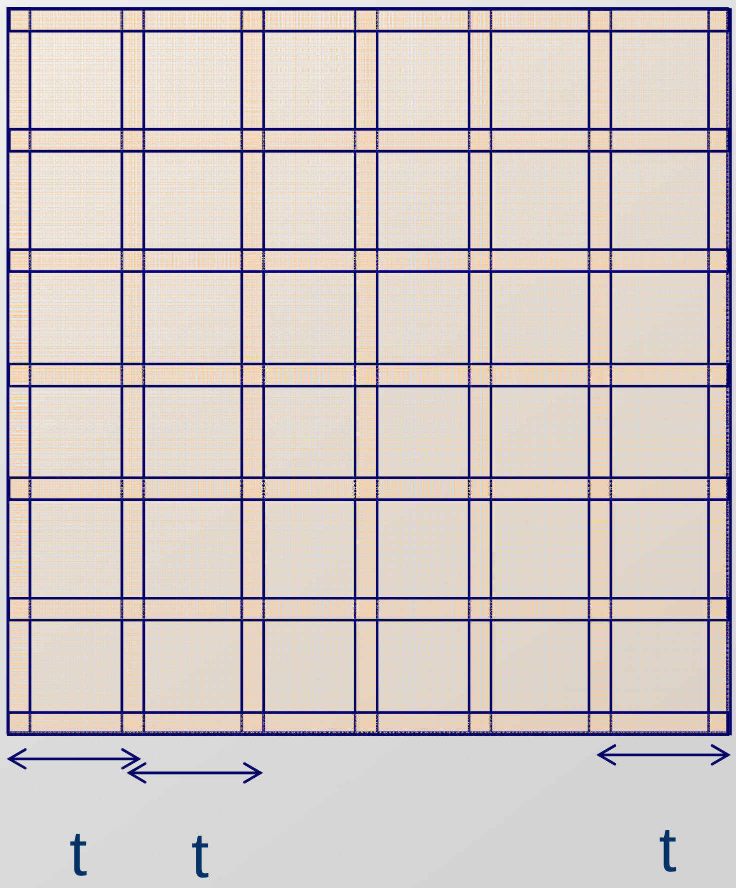
The Four-Russian Algorithm

<u>Main structure of the algorithm:</u>

- Divide N×N DP matrix into K×K $\log_2 N$-blocks that overlap by 1 column & 1 row
- For i = 1……K
- For j = 1……K
- Compute $D_{i,j}$ as a function of

$A_{i,j}$, $B_{i,j}$, $C_{i,j}$, $x[l_i \ldots l'_i]$, $y[r_j \ldots r'_j]$

<u>Time:</u> $O(N^2 / \log^2 N)$
times the cost of step 4

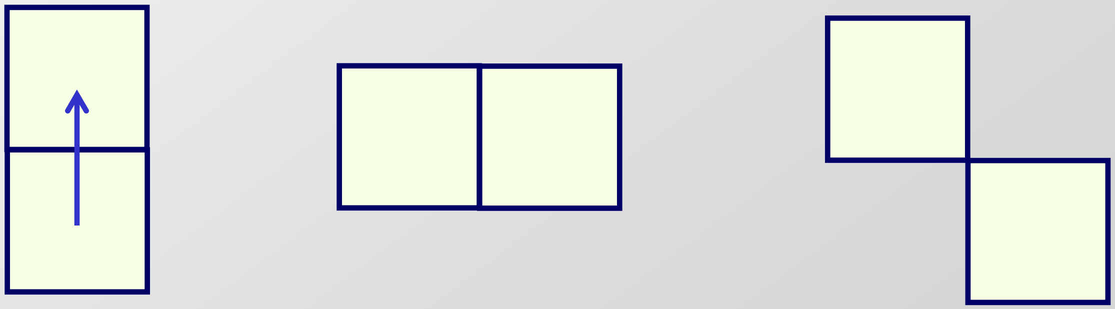


The Four-Russian Algorithm

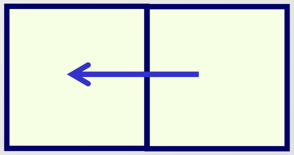
Another observation:
(Assume m = 0, s = 1, d = 1)

Lemma. Two adjacent cells of F(.,.) differ by at most 1

Gusfield's book covers case where m = 0,
called the *edit distance* (p. 216):
minimum # of substitutions + gaps to transform one string to another



The Four-Russian Algorithm



Proof of Lemma:

1. Same row:

   a. $F(i, j) – F(i – 1, j) \leq +1$

   At worst, one more gap:

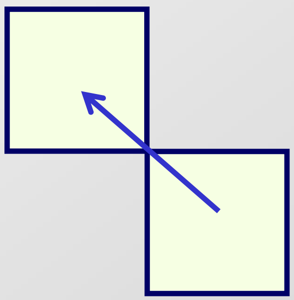
   $x_1 \ldots\ldots x_{i-1}\ x_i$
   $y_1 \ldots\ldots y_j\ \ –$

   b. $F(i, j) – F(i – 1, j) \geq -1$

$F(i, j)$	$F(i – 1, j – 1)$	$F(i, j) – F(i – 1, j – 1)$
$x_1 \ldots\ldots x_{i-1}\ x_i$ $y_1 \ldots\ldots y_{a-1} y_a\ y_{a+1} \ldots y_j$	$x_1 \ldots\ldots x_{i-1}\ –$ $y_1 \ldots\ldots y_{a-1} y_a\ y_{a+1} \ldots y_j$	≥ -1
$x_1 \ldots\ldots x_{i-1}\ x_i$ $y_1 \ldots\ldots y_{a-1} – y_a \ldots y_j$	$x_1 \ldots\ldots x_{i-1}$ $y_1 \ldots\ldots y_{a-1} y_a \ldots y_j$	$+1$

2. Same column: similar argument

The Four-Russian Algorithm



Proof of Lemma:

3. Same diagonal:

   a. $F(i, j) - F(i - 1, j - 1) \leq +1$
   At worst, one additional mismatch in F(i, j)

   b. $F(i, j) - F(i - 1, j - 1) \geq -1$

F(i, j)	F(i – 1, j – 1)	F(i, j) – F(i – 1, j – 1)
$x_1 \ldots\ldots x_{i-1}\ x_i$ | $y_1 \ldots\ldots y_{i-1}\ y_j$	$x_1 \ldots\ldots x_{i-1}$ $y_1 \ldots\ldots y_{j-1}$	≥ -1
$x_1 \ldots\ldots x_{i-1}\ x_i$ $y_1 \ldots\ldots y_{a-1} - y_a \ldots y_j$	$x_1 \ldots\ldots x_{i-1}$ $y_1 \ldots\ldots y_{a-1} y_a \ldots y_j$	+1

The Four-Russian Algorithm

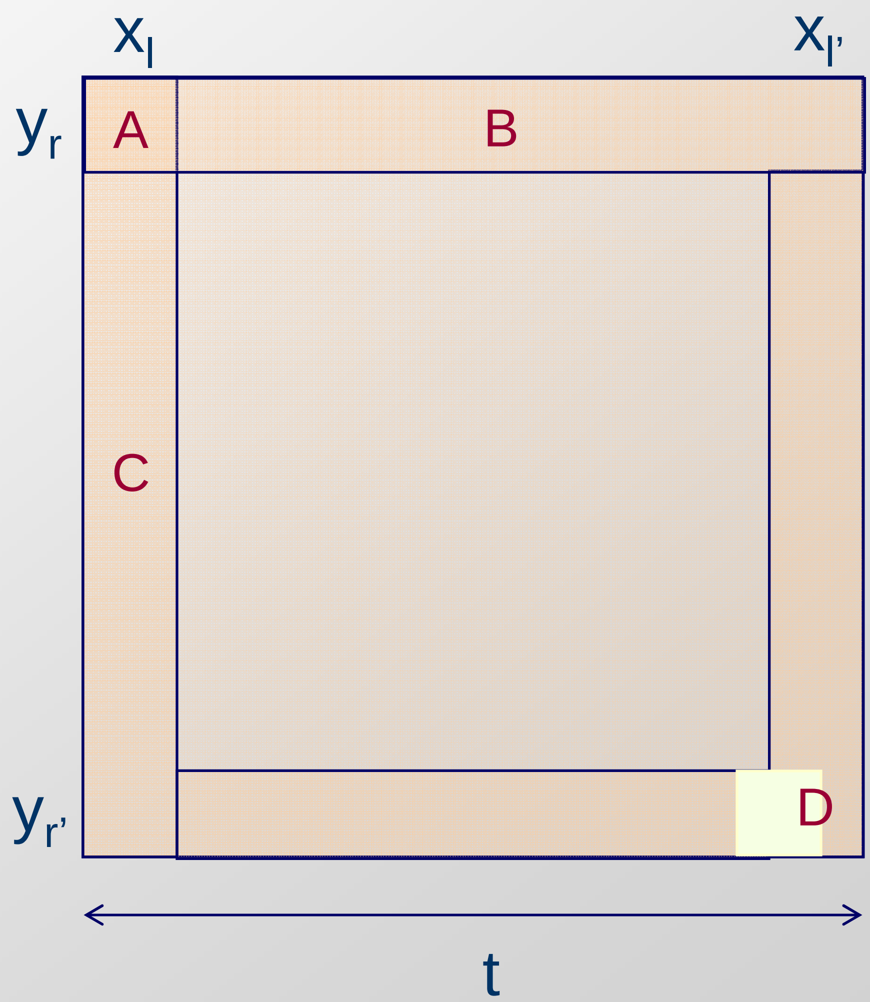
<u>**Definition:**</u>

The offset vector is a t-long vector of values from {-1, 0, 1}, where the first entry is 0

If we know the value at A, and the top row, left column offset vectors,

and $x_l \ldots\ldots x_{l'}$, $y_r \ldots\ldots y_{r'}$,

Then we can find D



The Four-Russian Algorithm

Example:

x = AACT

y = CACT

A	A	C	T
0	1	-1	-1

5	6	5	4
5	5	6	5
4	5	5	6
5	5	6	5

C 0 — 0

A 0 — 1

C -1 — 1

T 1 — -1

0 0 1 -1

The Four-Russian Algorithm

Example:

x = AACT

y = CACT

	A	A	C	T	
	0	1	-1	-1	
	1	2	1	0	
	1	1	2	1	
C 0					0
	0	1	1	2	
A 0					1
C -1	1	1	2	1	1
T 1					-1
	0	0	1	-1	

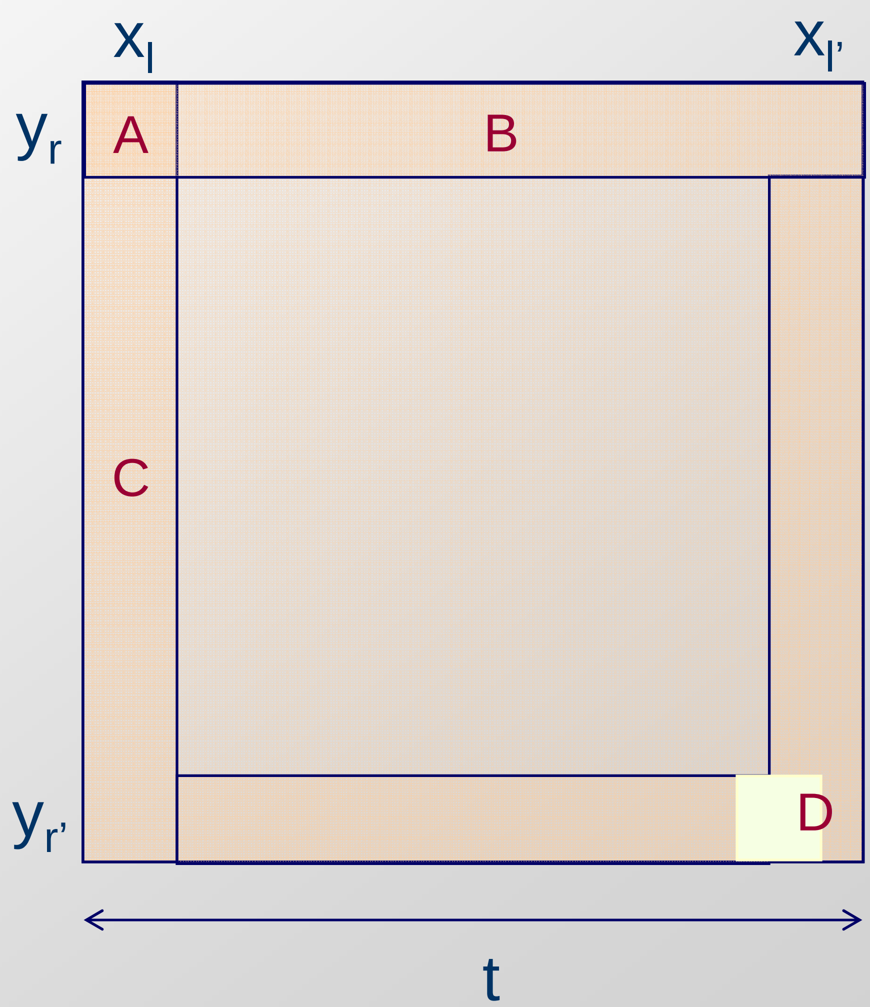
The Four-Russian Algorithm

Definition:

The offset function of a t-block is a function that for any

given offset vectors of top row, left column, and $x_l \ldots\ldots x_{l'}$, $y_r \ldots\ldots y_{r'}$,

produces offset vectors of bottom row, right column



The Four-Russian Algorithm

We can pre-compute the offset function:

$3^{2(t-1)}$ possible input offset vectors

4^{2t} possible strings $x_l \ldots\ldots x_{l'}$, $y_r \ldots\ldots y_{r'}$

Therefore $3^{2(t-1)} \times 4^{2t}$ values to pre-compute

We can keep all these values in a table, and look up in linear time,
or in O(1) time if we assume
constant-lookup RAM for log-sized inputs

The Four-Russian Algorithm

Four-Russians Algorithm: (Arlazarov, Dinic, Kronrod, Faradzev)

1. Cover the DP table with t-blocks

2. Initialize values F(.,.) in first row & column

3. Row-by-row, use offset values at leftmost column and top row of each block, to find offset values at rightmost column and bottom row

4. Let Q = total of offsets at row N

   F(N, N) = Q + F(N, 0)

The Four-Russian Algorithm