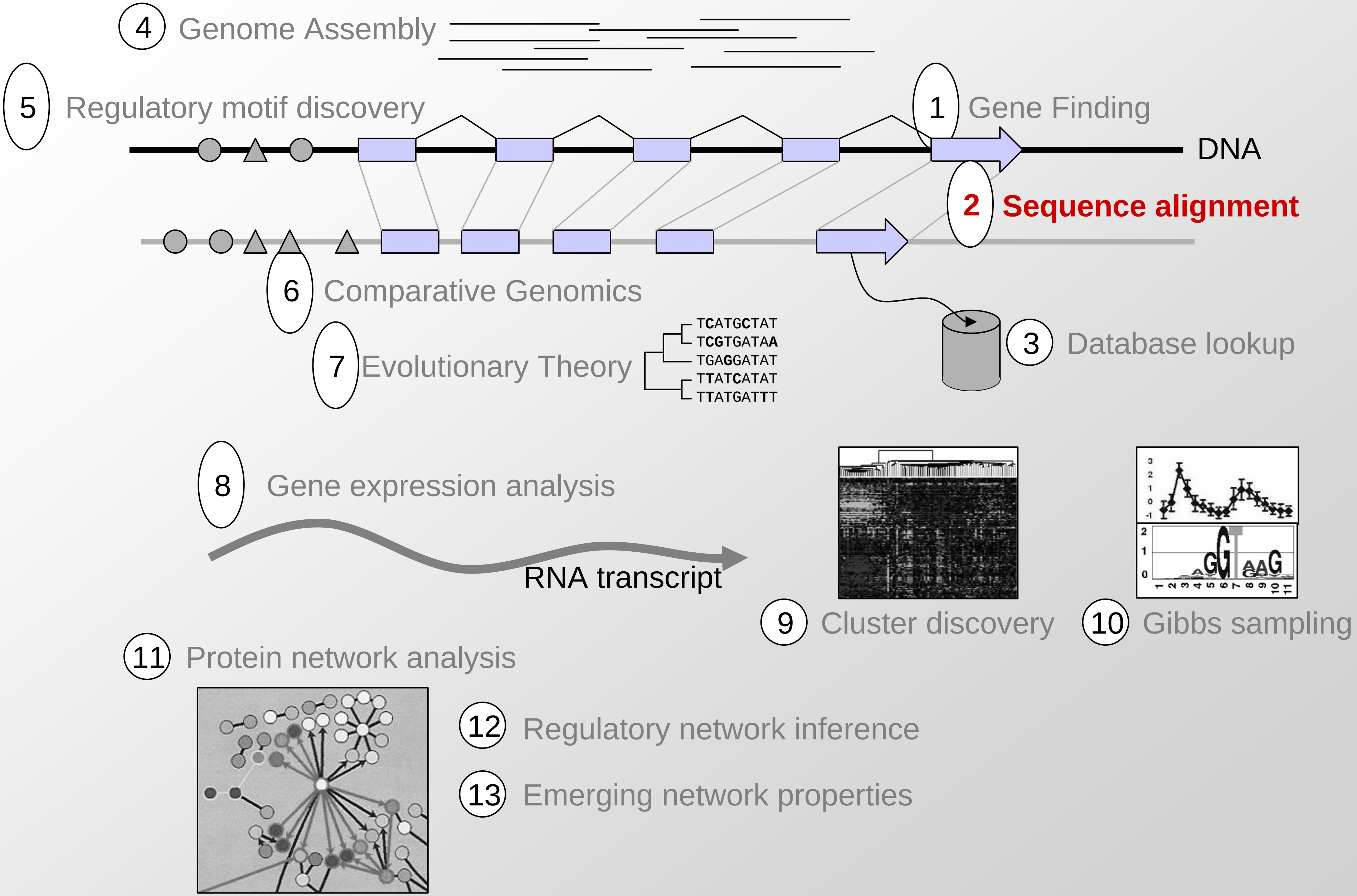
6.047 / 6.878 Computational Biology: Genomes, Networks, Evolution
Fall 2008

6.047/6.878 - Computational Biology: Genomes, Networks, Evolution

Sequence Alignment
and Dynamic Programming

Tue Sept 9, 2008

Challenges in Computational Biology



4 Genome Assembly

5 Regulatory motif discovery

1 Gene Finding

2 Sequence alignment

6 Comparative Genomics

7 Evolutionary Theory

3 Database lookup

8 Gene expression analysis

9 Cluster discovery

10 Gibbs sampling

11 Protein network analysis

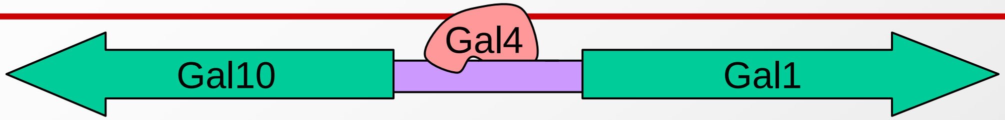
12 Regulatory network inference

13 Emerging network properties

Reminder: Last lecture / recitation

- Schedule for the term
  - ‘Foundations’ till midterm
  - ‘Frontiers’ lead to final project
  - Duality: basic problems / fundamental techniques
- Biology introduction
  - DNA, RNA, protein, transcription, translation
  - Why <u>computational</u> biology
- Today: Comparative genomics is everywhere!
  - Problem set 1: dating vertebrate whole-genome duplication
  - Problem set 2: discover genes using their conservation properties
  - Problem set 3: discover all motifs across entire yeast genome
  - Problem set 4: reversing human/mouse genome rearrangements

Evolution preserved functional elements!



```
Scer    TTATATTGAATTTTCAAAAATTCTTACTTTTTTTTTTGGATGGACGCAAAGAAGTTTAATAATCATATTACATGGCATTACCACCATATACA
Spar    CTATGTTGATCTTTTCAGAATTTTT-CACTATATTAAGATGGGTGCAAAGAAGTGTGATTATTATATTACATCGCTTTCCTATCATACACA
Smik    GTATATTGAATTTTTCAGTTTTTTTTCACTATCTTCAAGGTTATGTAAAAAA-TGTCAAGATAATATTACATTTCGTTACTATCATACACA
Sbay    TTTTTTTGATTTCTTTAGTTTTCTTTCTTTAACTTCAAAATTATAAAAGAAAGTGTAGTCACATCATGCTATCT-GTCACTATCACATATA
         * * ****  * *  *   ** ** *  *   **           **  ** * *    *    **   **    *  * * ** * * *

                          TBP
Scer    TATCCATATCTAATCTTACTTATATGTTGT-GGAAAT-GTAAAGAGCCCCATTATCTTAGCCTAAAAAAACC--TTCTCTTTGGAACTTTCAGTAATACG
Spar    TATCCATATCTAGTCTTACTTATATGTTGT-GAGAGT-GTTGATAACCCCAGTATCTTAACCCAAGAAAGCC--TT-TCTATGAAACTTGAACTG-TACG
Smik    TACCGATGTCTAGTCTTACTTATATGTTAC-GGGAATTGTTGGTAATCCCAGTCTCCCAGATCAAAAAAGGT--CTTTCTATGGAGCTTTG-CTA-TATG
Sbay    TAGATATTTCTGATCTTTCTTATATATTATAGAGAGATGCCAATAAACGTGCTACCTCGAACAAAAGAAGGGGATTTTCTGTAGGGCTTTCCCTATTTTG
        **   ** ***  **** ******* **   *  *   *     *  *    *  *       ** **       * *** *    ***    *  *  *

                                         GAL4               GAL4              GAL4
Scer    CTTAACTGCTCATTGC-----TATATTGAAGTACGGATTAGAAGCCGCCGAGCGGGCGACAGCCCTCCGACGGAAGACTCTCCTCCGTGCGTCCTCGTCT
Spar    CTAAACTGCTCATTGC-----AATATTGAAGTACGGATCAGAAGCCGCCGAGCGGACGACAGCCCTCCGACGGAATATTCCCCTCCGTGCGTCGCCGTCT
Smik    TTTAGCTGTTCAAG--------ATATTGAAATACGGATGAGAAGCCGCCGAACGGACGACAATTCCCCGACGGAACATTCTCCTCCGCGCGGCGTCCTCT
Sbay    TCTTATTGTCCATTACTTCGCAATGTTGAAATACGGATCAGAAGCTGCCGACCGGATGACAGTACTCCGGCGGAAAACTGTCCTCCGTGCGAAGTCGTCT
              **  **          ** ***** ******* ****** ***** ***  ****   * *** ***** * *  ****** ***    * ***

                                         GAL4
Scer    TCACCGG-TCGCGTTCCTGAAACGCAGATGTGCCTCGCGCCGCACTGCTCCGAACAATAAAGATTCTACAA-----TACTAGCTTTT--ATGGTTATGAA
Spar    TCGTCGGGTTGTGTCCCTTAA-CATCGATGTACCTCGCGCCGCCCTGCTCCGAACAATAAGGATTCTACAAGAAA-TACTTGTTTTTTTATGGTTATGAC
Smik    ACGTTGG-TCGCGTCCCTGAA-CATAGGTACGGCTCGCACCACCGTGGTCCGAACTATAATACTGGCATAAAGAGGTACTAATTTCT--ACGGTGATGCC
Sbay    GTG-CGGATCACGTCCCTGAT-TACTGAAGCGTCTCGCCCCGCCATACCCCGAACAATGCAAATGCAAGAACAAA-TGCCTGTAGTG--GCAGTTATGGT
             ** *   ** *** *      *      ***** ** *  *   ****** **     *   * **     * *             ** ***

                                     MIG1
Scer    GAGGA-AAAATTGGCAGTAA----CCTGGCCCCACAAACCTT-CAAATTAACGAATCAAATTAACAACCATA-GGATGATAATGCGA------TTAG--T
Spar    AGGAACAAAATAAGCAGCCC----ACTGACCCCATATACCTTTCAAACTATTGAATCAAATTGGCCAGCATA-TGGTAATAGTACAG------TTAG--G
Smik    CAACGCAAAATAAACAGTCC----CCCGGCCCCACATACCTT-CAAATCGATGCGTAAAACTGGCTAGCATA-GAATTTTGGTAGCAA-AATATTAG--G
Sbay    GAACGTGAAATGACAATTCCTTGCCCCT-CCCCAATATACTTTGTTCCGTGTACAGCACACTGGATAGAACAATGATGGGGTTGCGGTCAAGCCTACTCG
               ****    *         *   *****     ***               * * *    *  * *    *     *           **

                 MIG1                                                      TBP
Scer    TTTTTAGCCTTATTTCTGGGGTAATTAATCAGCGAAGCG--ATGATTTTT-GATCTATTAACAGATATATAAATGGAAAAGCTGCATAACCAC-----TT
Spar    GTTTT--TCTTATTCCTGAGACAATTCATCCGCAAAAAATAATGGTTTTT-GGTCTATTAGCAAACATATAAATGCAAAAGTTGCATAGCCAC-----TT
Smik    TTCTCA--CCTTTCTCTGTGATAATTCATCACCGAAATG--ATGGTTTA--GGACTATTAGCAAACATATAAATGCAAAAGTCGCAGAGATCA-----AT
Sbay    TTTTCCGTTTTACTTCTGTAGTGGCTCAT--GCAGAAAGTAATGGTTTTCTGTTCCTTTTGCAAACATATAAATATGAAAGTAAGATCGCCTCAATTGTA
         * *      *    ***       * **   *  *     *** ***   *  *  **  ** * ********   ****    *

Scer    TAACTAATACTTTCAACATTTTCAGT--TTGTATTACTT-CTTATTCAAAT----GTCATAAAAGTATCAACA-AAAAATTGTTAATATACCTCTATACT
Spar    TAAATAC-ATTTGCTCCTCCAAGATT--TTTAATTTCGT-TTTGTTTTATT----GTCATGGAAATATTAACA-ACAAGTAGTTAATATACATCTATACT
Smik    TCATTCC-ATTCGAACCTTTGAGACTAATTATATTTAGTACTAGTTTTCTTTGGAGTTATAGAAATACCAAAA-AAAAATAGTCAGTATCTATACATACA
```

We can ‘read’ evolution to reveal functional elements

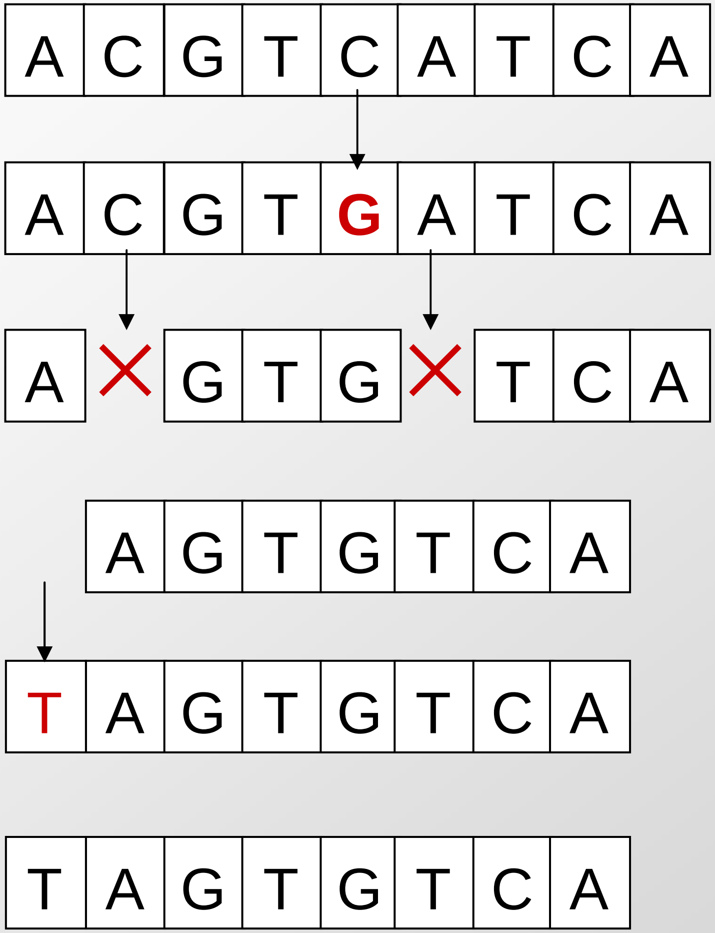
Conservation island

Kellis et al, Nature 2003

Today’s goal:

How do we actually align two genes?

Genomes change over time



begin

A C G T C A T C A

mutation

A C G T G A T C A

deletion

A ✕ G T G ✕ T C A

A G T G T C A

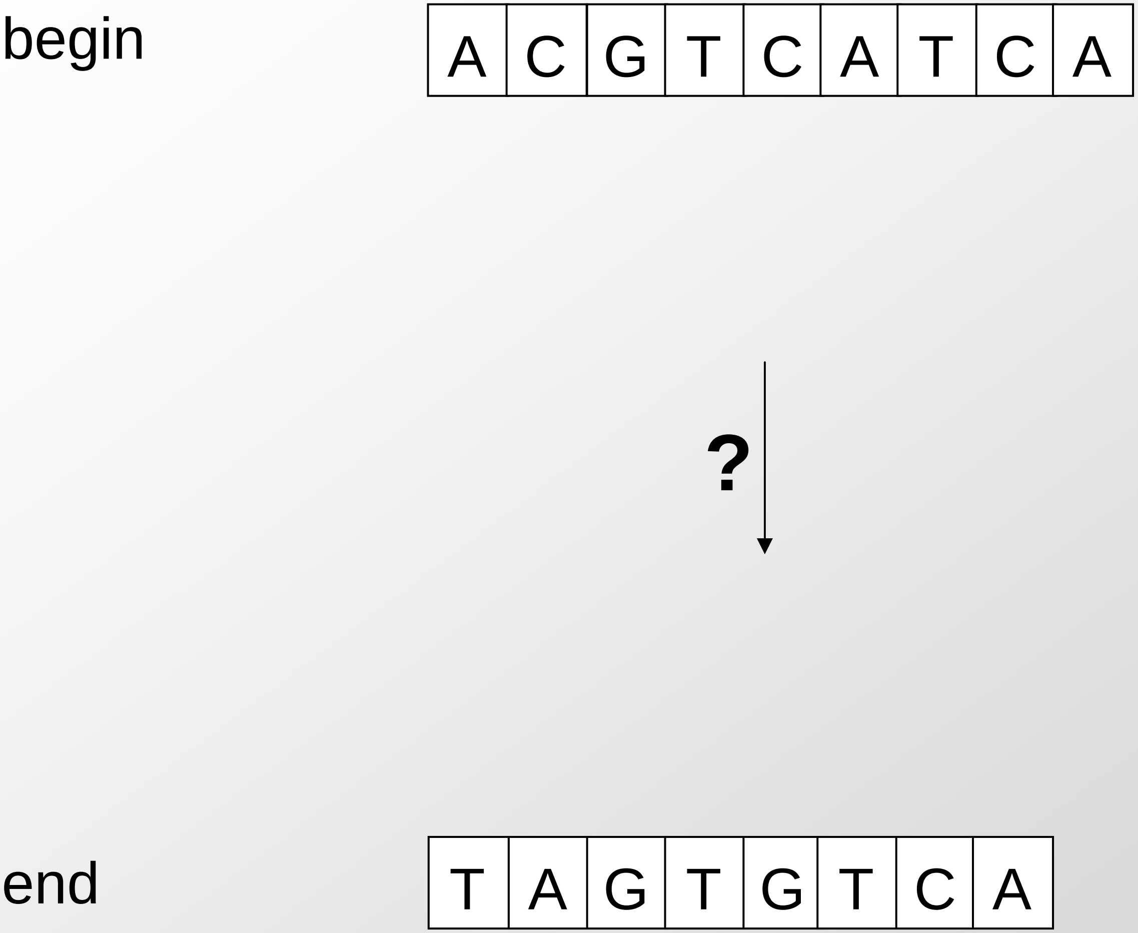
insertion

T A G T G T C A

end

T A G T G T C A

Goal of alignment: Infer edit operations



begin

A	C	G	T	C	A	T	C	A

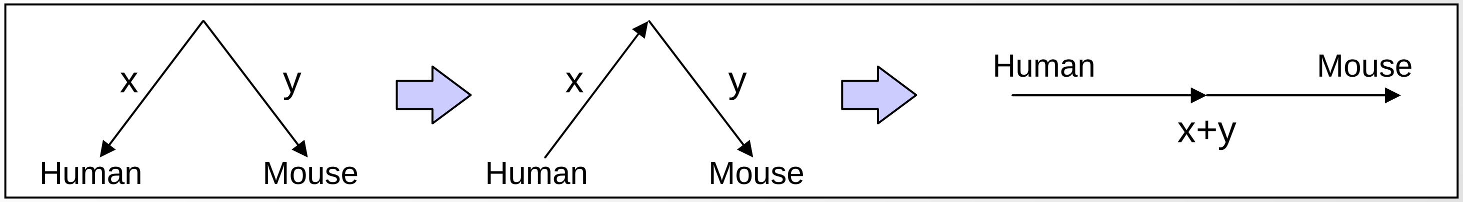
?

end

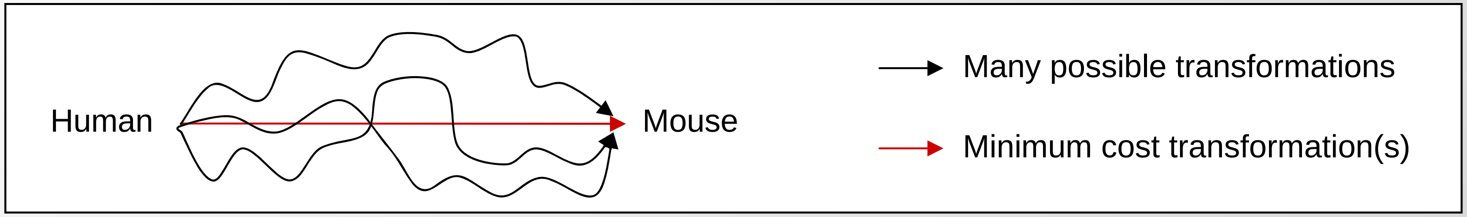
T	A	G	T	G	T	C	A

From Bio to CS: Formalizing the problem

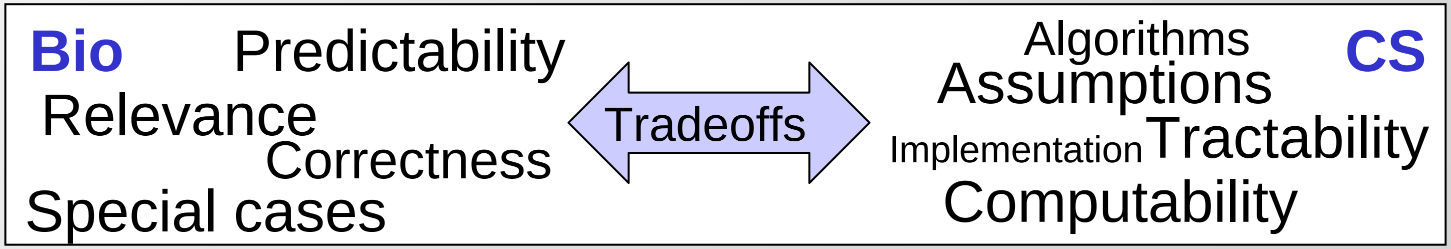
- Define set of evolutionary operations (insertion, deletion, mutation)
  - Symmetric operations allow time reversibility (part of design choice)



- Define optimality criterion (min number, min cost)
  - Impossible to infer exact series of operations (Occam's razor: find min)



- Design algorithm that achieves that optimality (or approximates it)
  - Tractability of solution depends on assumptions in the formulation



Bio Predictability

Relevance

Correctness

Special cases

Tradeoffs

Algorithms **CS**

Assumptions

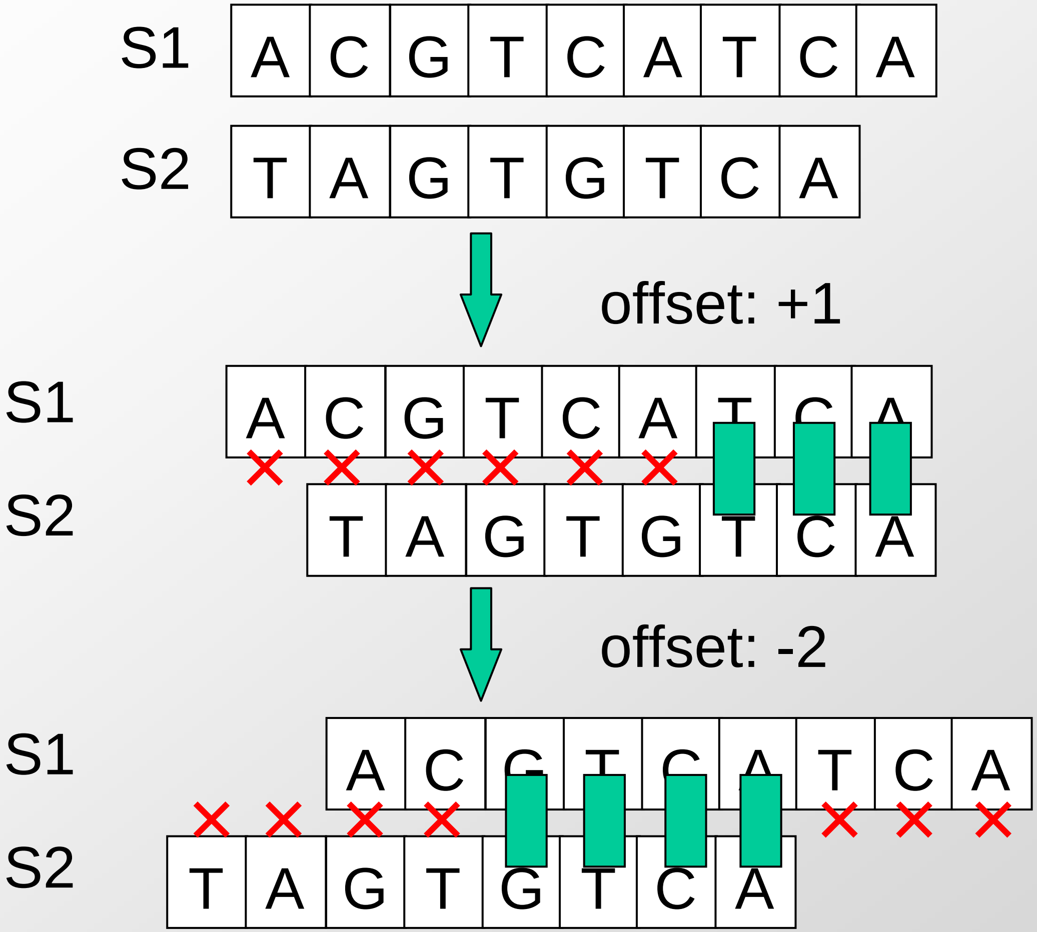
Implementation Tractability

Computability

Note: Not all decisions are conflicting (some are both relevant and tractable) (e.g. Pevzner vs. Sankoff and directionality in chromosomal inversions)

Formulation 1: Longest common substring

- Given two possibly related strings S1 and S2
  - What is the longest common substring? (no gaps)



S1 A C G T C A T C A

S2 T A G T G T C A

offset: +1

S1 A C G T C A T C A

S2 T A G T G T C A

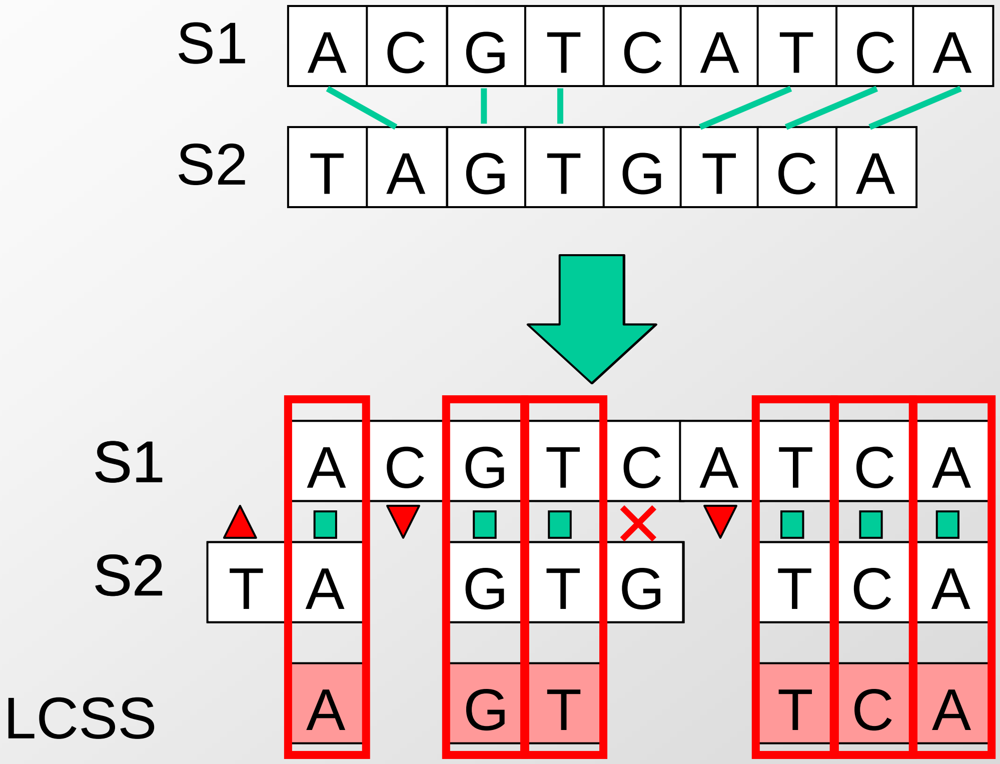
offset: -2

S1 A C G T C A T C A

S2 T A G T G T C A

Formulation 2: Longest common subsequence

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



S1 A C G T C A T C A

S2 T A G T G T C A

S1		A	C	G	T	C	A	T	C	A
	▲	■	▼	■	■	✕	▼	■	■	■
S2	T	A		G	T	G		T	C	A
LCSS		A		G	T			T	C	A

Edit distance:

- Number of changes needed for S1→S2
- Uniform scoring function

Formulation 3: Sequence alignment

- Allow gaps (fixed penalty)
  - Insertion & deletion operations
  - Unit cost for each character inserted or deleted
- Varying penalties for edit operations
  - Transitions (Pyrimidine⇔Pyrimidine, Purine⇔Purine)
  - Transversions (Purine ⇔ Pyrimidine changes)
  - Polymerase confuses Aw/G and Cw/T more often

Scoring function:

Match(x,x) = +1

Mismatch(A,G)= -½

Mismatch(C,T)= -½

Mismatch(x,y) = -1

	A	G	T	C
A	+1	-½	-1	-1
G	-½	+1	-1	-1
T	-1	-1	+1	-½
C	-1	-1	-½	+1

purine pyrimid.

Transitions:

A⇔G, C⇔T common (lower penalty)

Transversions:

All other operations

Etc…

(e.g. varying gap penalties)

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	

- Given additive scoring function:
  - Cost of mutation (AG, CT, other)
  - Cost of insertion / deletion
  - Reward of match
- Need algorithm for inferring best alignment
  - Enumeration?
  - How would you do it?
  - How many alignments are there?

Can we simply enumerate all possible alignments?

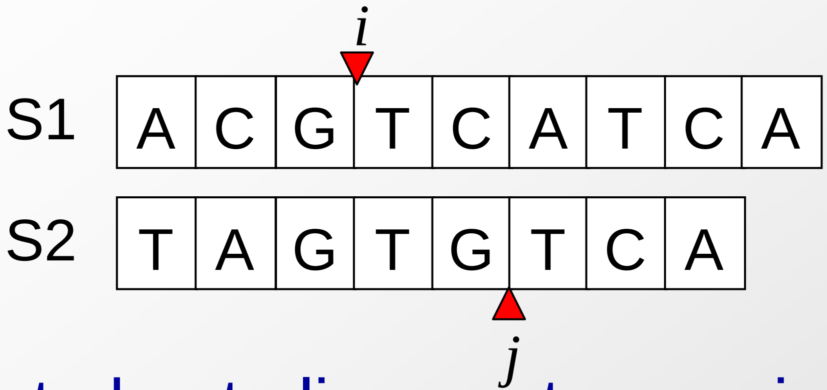
- Ways to align two sequences of length m, n

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

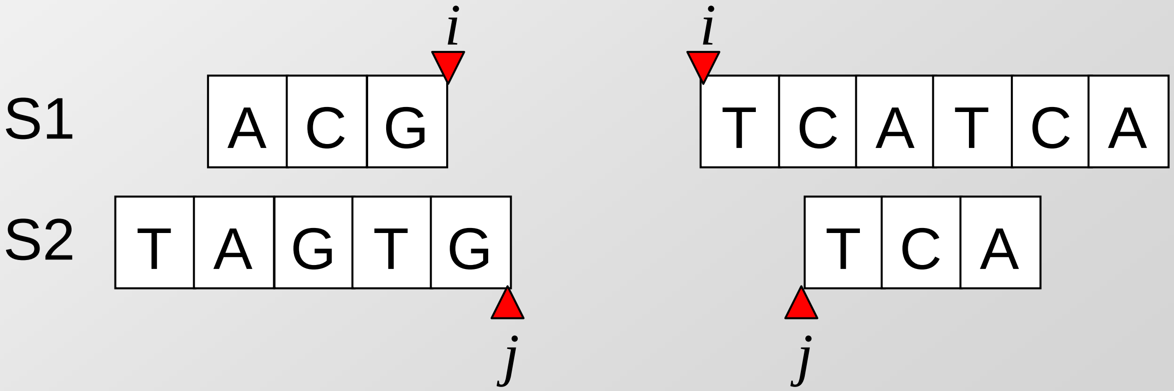
- For two sequences of length n

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

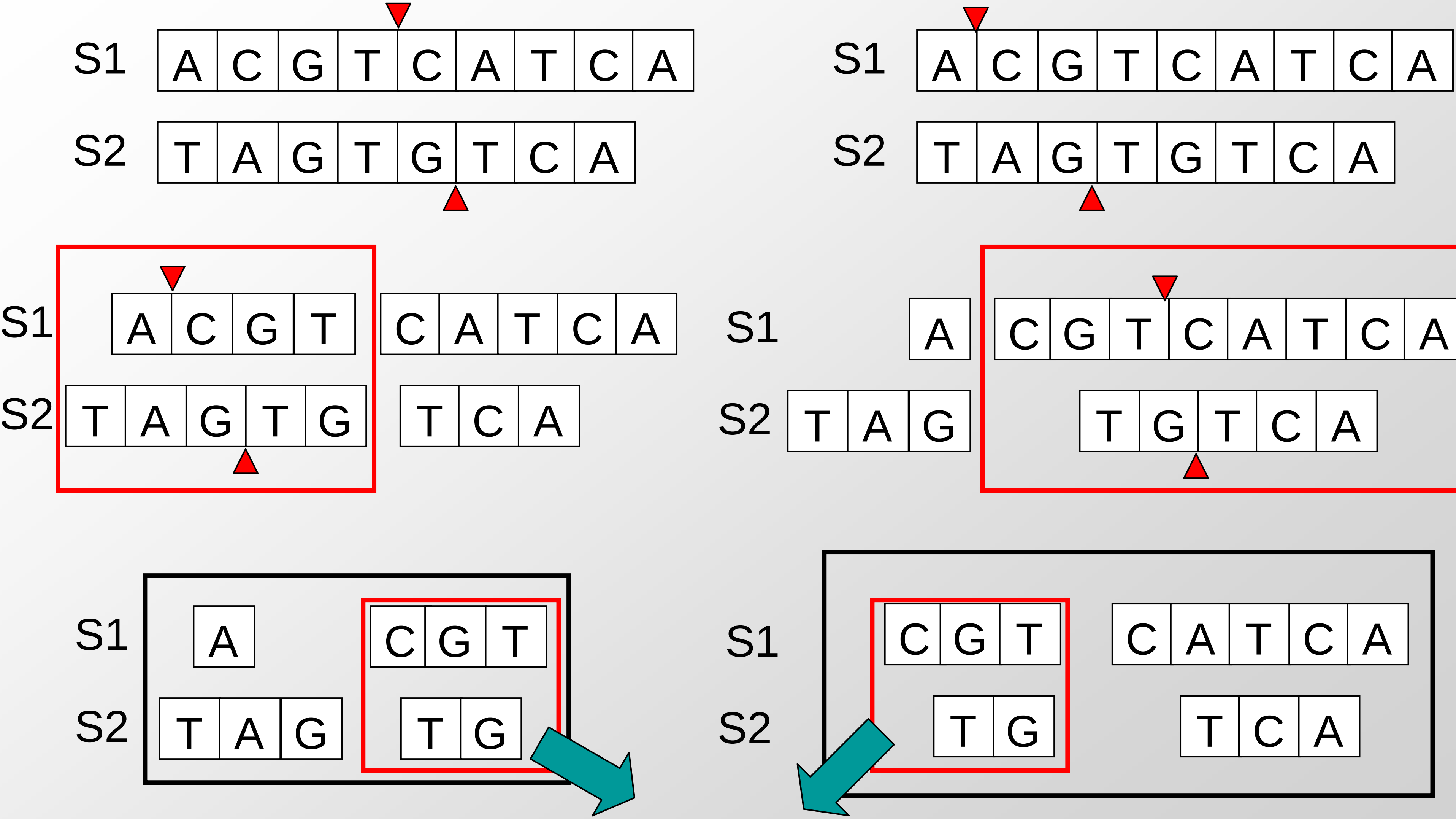
Key insight: score is additive!



- Compute best alignment recursively
  - For a given aligned pair ***(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]
  - Proof: cut-and-paste argument (see 6.046)



Key insight: re-use computation



S1 A C G T C A T C A

S2 T A G T G T C A

S1 A C G T C A T C A

S2 T A G T G T C A

S1 A C G T C A T C A

S2 T A G T G T C A

S1 A C G T C A T C A

S2 T A G T G T C A

S1 A C G T

S2 T A G T G

S1 C G T C A T C A

S2 T A G

S1 A C G T

S2 T A G T G

S1 C G T C A T C A

S2 T G T C A

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

A simple introduction to Dynamic Programming

- Fibonacci numbers

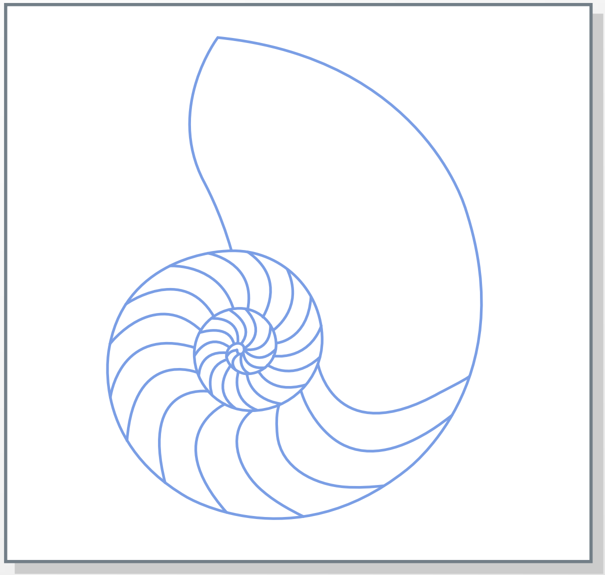
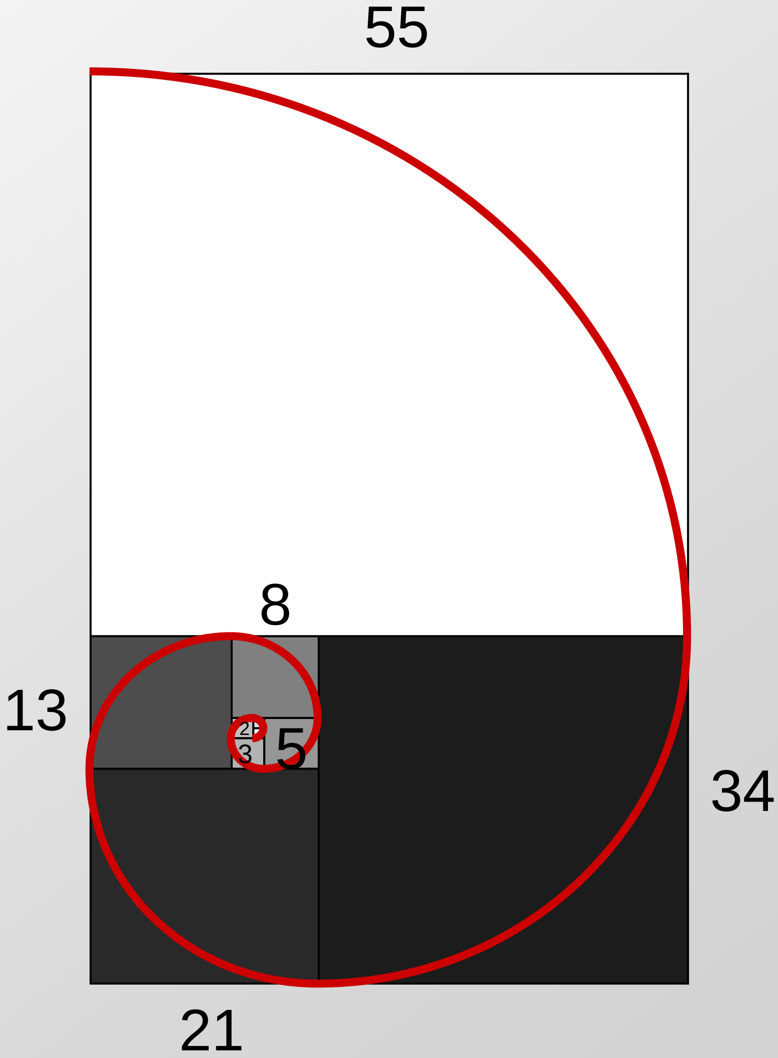
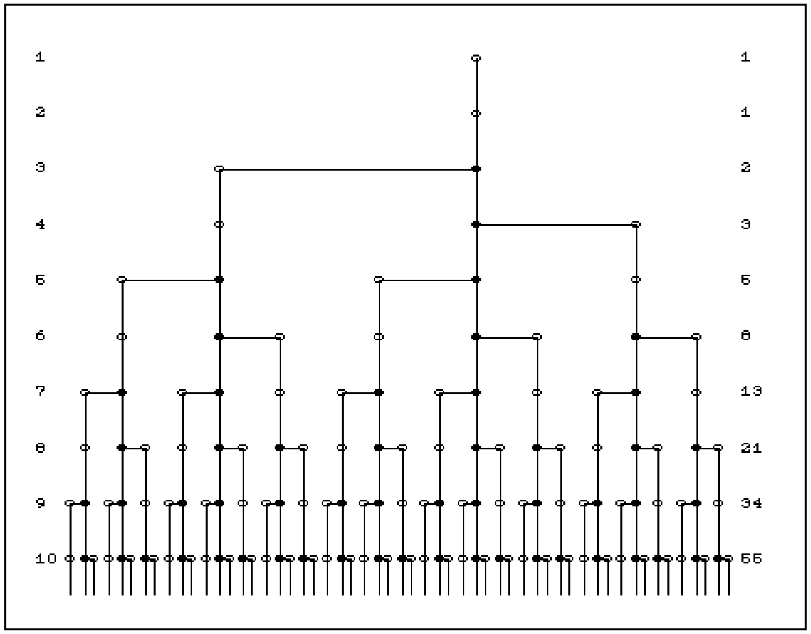


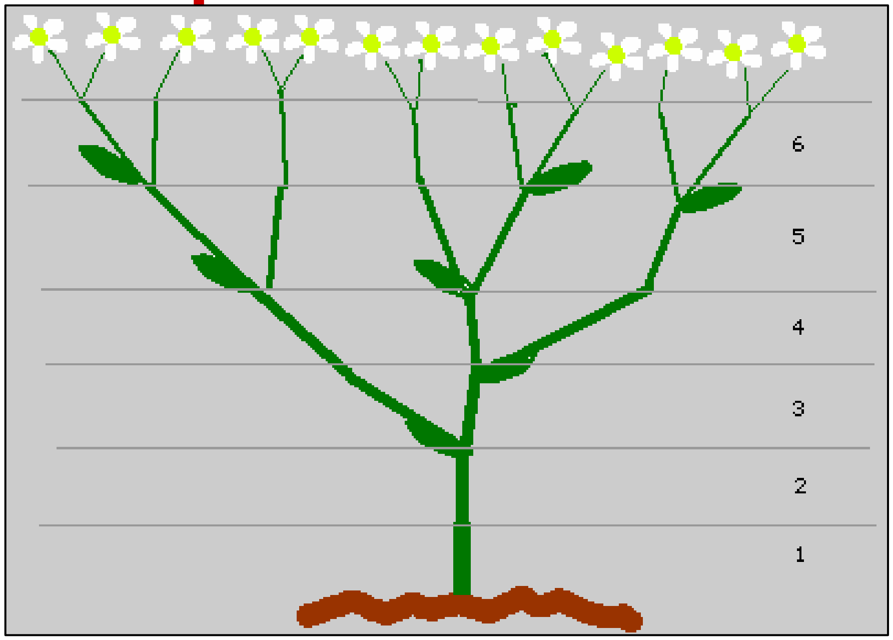
Figure by MIT OpenCourseWare.



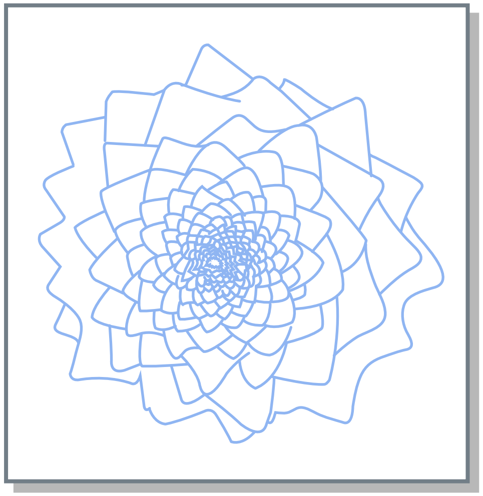
Fibonacci numbers are ubiquitous in nature



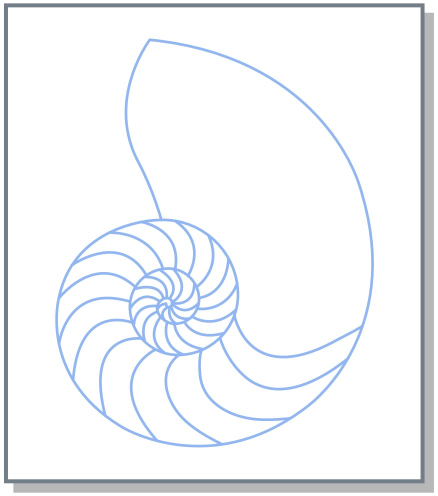
Rabbits per generation



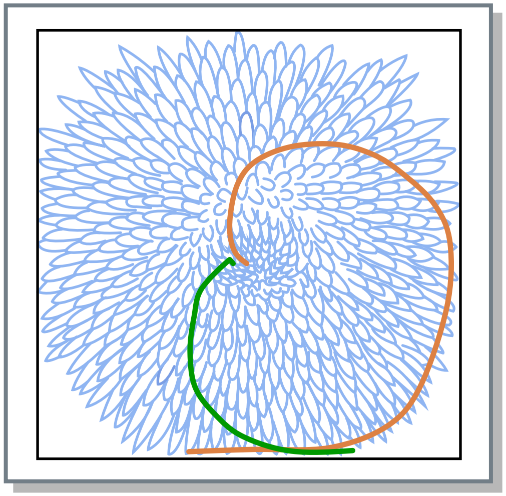
Leaves per height



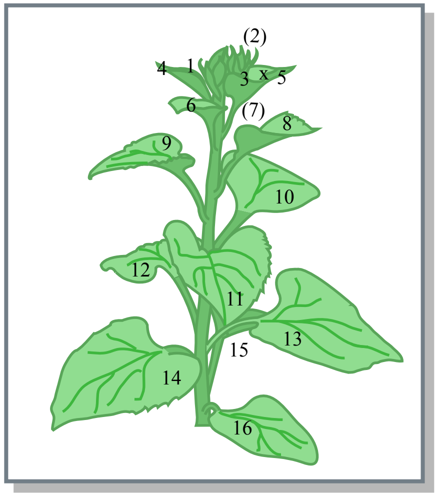
Romanesque spirals



Nautilus size



Coneflower spirals



Leaf ordering

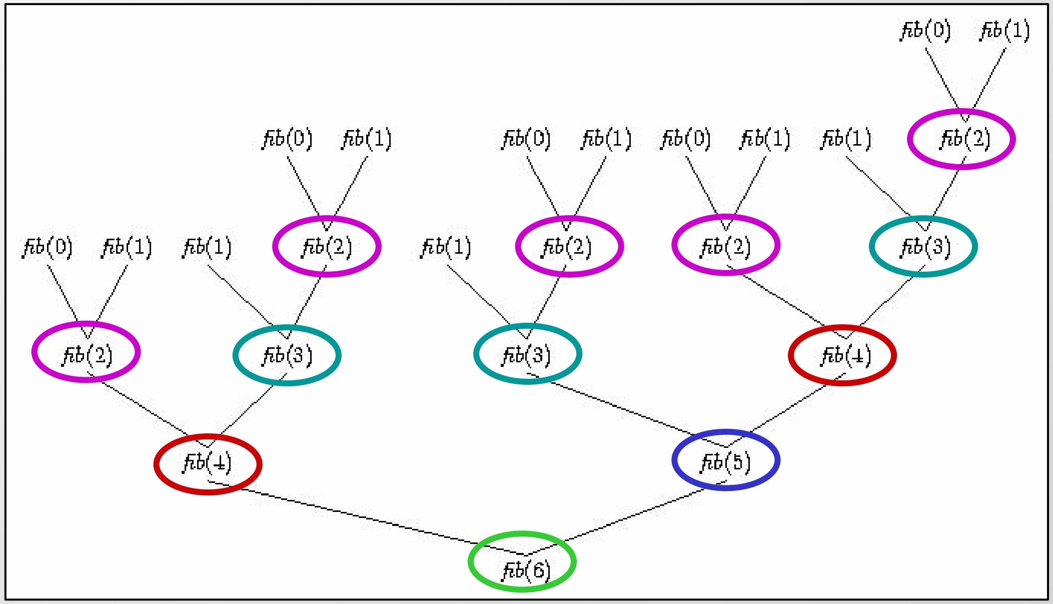
Figures by MIT OpenCourseWare.

Computing Fibonacci numbers: Top down

- Fibonacci numbers are defined recursively:
  - Python code

```
def fibonacci(n):
    if n==1 or n==2: return 1
    return fibonacci(n-1) + fibonacci(n-2)
```

- Goal: Compute n^th^ Fibonacci number.
  - F(0)=1, F(1)=1, F(n)=F(n-1)+F(n-2)
  - 1,1,2,3,5,8,13,21,34,55,89,144,233,377,…
- Analysis:
  - T(n) = T(n-1) + T(n-2) = (…) = $O(2^n)$



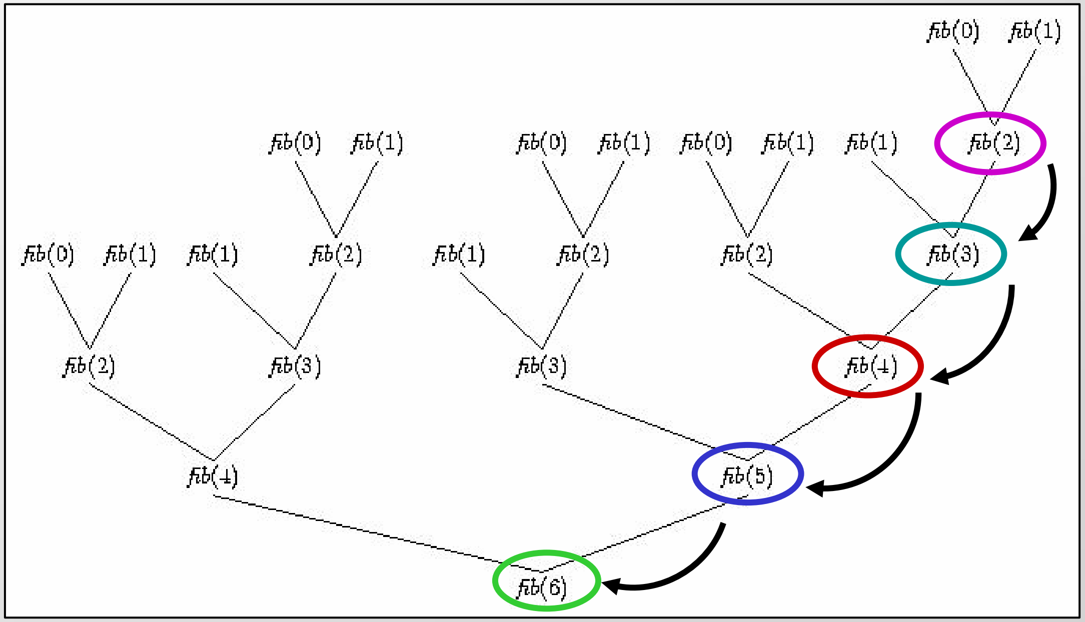
Computing Fibonacci numbers: Bottom up

- Top-down approach
  - Python code

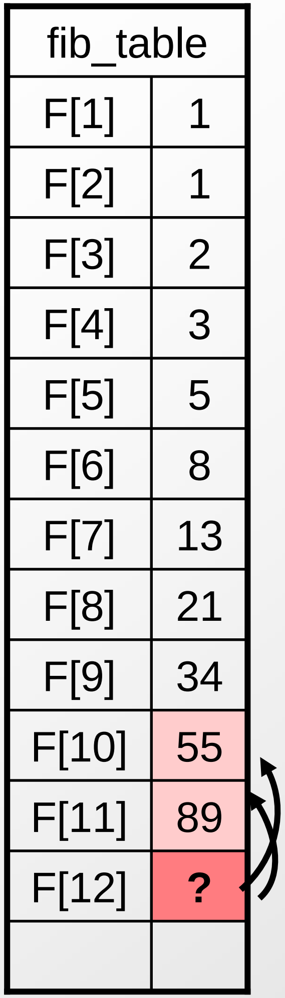
fib_table	
F[1]	1
F[2]	1
F[3]	2
F[4]	3
F[5]	5
F[6]	8
F[7]	13
F[8]	21
F[9]	34
F[10]	55
F[11]	89
F[12]	?

```
def fibonacci(n):
        fib_table[1] = 1
        fib_table[2] = 1
        for i in range(3,n+1):
          fib_table[i] = fib_table[i-1]+fib_table[i-2]
        return fib_table[n]
```

  - Analysis: T(n) = O(n)



Lessons from iterative Fibonacci algorithm



fib_table	
F[1]	1
F[2]	1
F[3]	2
F[4]	3
F[5]	5
F[6]	8
F[7]	13
F[8]	21
F[9]	34
F[10]	55
F[11]	89
F[12]	?

- What did the iterative solution do?
  - Reveal identical sub-problems
  - Order computation to enable result reuse
  - Systematically filled-in table of resluts
  - Expressed larger problems from their subparts
- Ordering of computations matters
  - Naïve top-down approach very slow
    - results of smaller problems not available
    - repeated work
  - Systematic bottom-up approach successful
    - Systematically solve each sub-problem
    - Fill-in table of sub-problem results in order.
    - Look up solutions instead of recomputing

Dynamic Programming in Theory

- Hallmarks of Dynamic Programming
  - **Optimal substructure:** Optimal solution to problem (instance) contains optimal solutions to sub-problems
  - **Overlapping subproblems:** Limited number of distinct subproblems, repeated many many times
- Typically for optimization problems (unlike Fib example)
  - Optimal choice made locally: max(subsolution score)
  - Score is typically added through the search space
  - Traceback common, find optimal path from indiv. choices
- Middle of the road in range of difficulty
  - Easier: greedy choice possible at each step
  - DynProg: requires a traceback to find that optimal path
  - Harder: no opt. substr., e.g. subproblem dependencies

Hallmarks of optimization problems

Greedy algorithms | **Dynamic Programming**

1. Optimal substructure

An optimal solution to a problem (instance) contains optimal solutions to subproblems.

2. Overlapping subproblems

A recursive solution contains a "small" number of distinct subproblems repeated many times.

3. Greedy choice property

Locally optimal choices lead to globally optimal solution

Greedy Choice is not possible

Globally optimal solution requires trace back through many choices

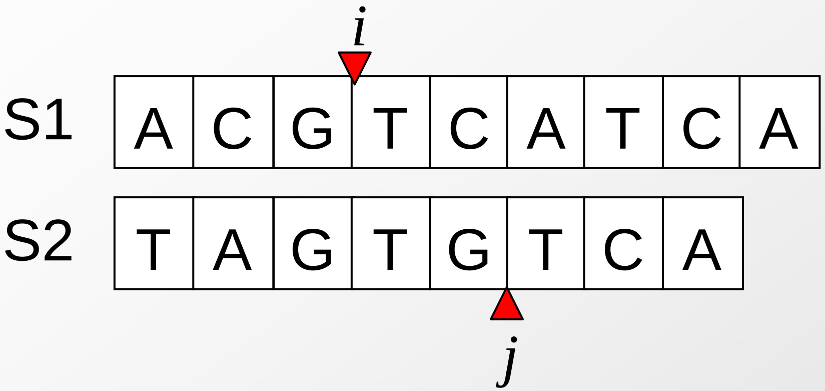
Dynamic Programming in Practice

- Setting up dynamic programming
  1. Find ‘matrix’ parameterization (# dimensions, variables)
  2. Make sure sub-problem space is finite! (not exponential)
     - If not all subproblems are used, better off using memoization
     - If reuse not extensive, perhaps DynProg is not right solution!
  3. Traversal order: sub-results ready when you need them
     - Computation order matters! (bottom-up, but not always obvious)
  4. Recursion formula: larger problems = F(subparts)
  5. Remember choices: typically F() includes min() or max()
     - Need representation for storing pointers, is this polynomial !
- Then start computing
  1. Systematically fill in table of results, find optimal score
  2. Trace-back from optimal score, find optimal solution

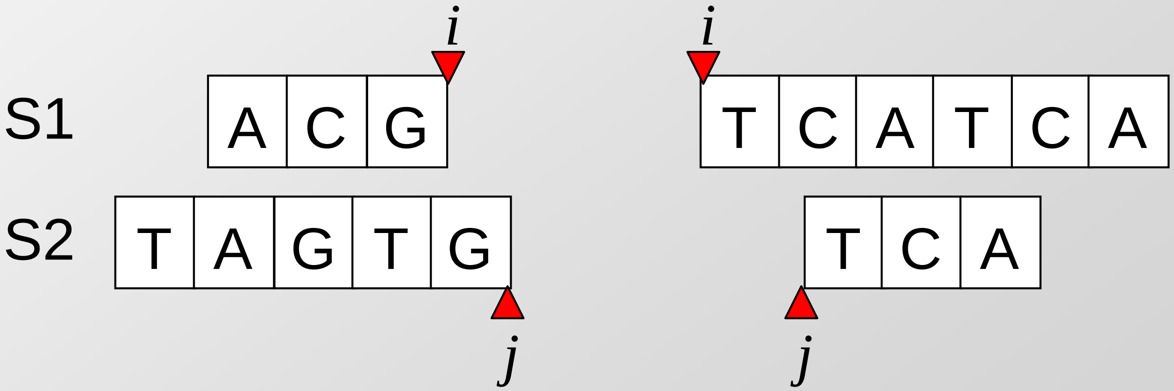
How do we apply dynamic programming

to sequence alignment ?

Key insight: score is additive!



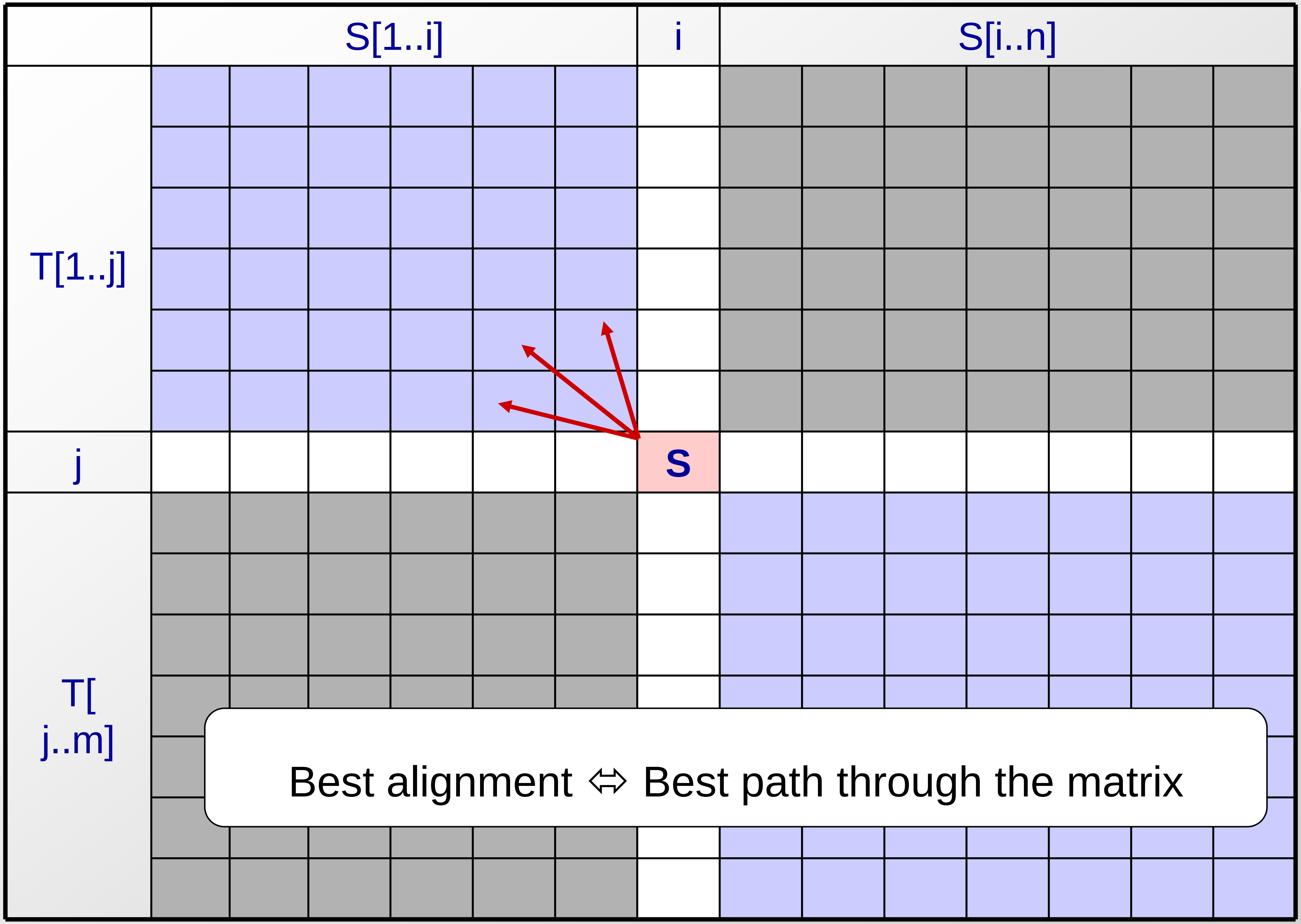
- Compute best alignment recursively
  - For a given aligned pair ***(i, j)***, the best alignment is:
    - Best alignment of S1[1..i] and S2[1..j]
    - + Best alignment of S1[i..n] and S2[j..m]



Dynamic Programming for sequence alignment

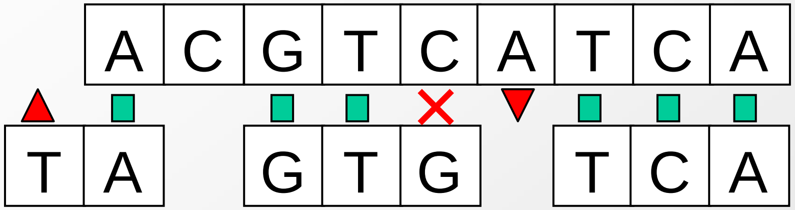
- Setting up dynamic programming
  1. Find ‘matrix’ parameterization
  2. Make sure sub-problem space is finite! (not exponential)
  3. Traversal order: sub-results ready when you need them
  4. Recursion formula: larger problems = F(subparts)
  5. Remember choices: typically F() includes min() or max()
- Then start computing
  1. Systematically fill in table of results, find optimal score
  2. Trace-back from optimal score, find optimal solution

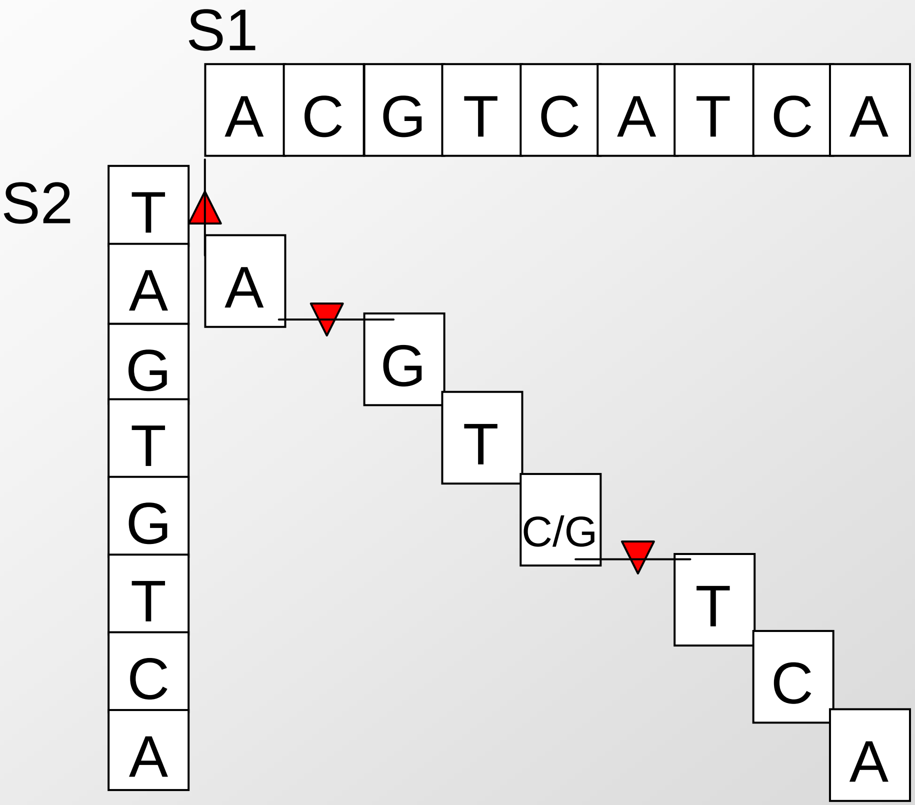
(1, 2, 3) Store score of aligning (i,j) in matrix M(i,j)



Best alignment ⇔ Best path through the matrix

Duality: seq. alignment ⇔ path through the matrix





Goal:

Find best path through the matrix

(4) Filling in the dynamic programming matrix

- Local update rules:
  - Compute next alignment based on previous alignment
  - Just like Fibonacci numbers: F[i] = F[i-1] + F[i-2]
  - Table lookup!
- Compute scores for prefixes of increasing length
  - This allows a single recursion (top-left to bottom-right) instead of two recursions (middle-to-outside top-down)
  - Only three possibilities for extending by one nucleotide: a gap in one species, a gap in the other, a (mis)match
  - When you reach bottom right, prefix of length *n* is seq *S*
- Computing the score of a cell from its neighbors
  - $$F(i,j) = \max\left\{ \begin{array}{l} F(i-1, j) - \text{gap} \\ F(i, j) + \text{score} \\ F(i, j-1) - \text{gap} \end{array} \right\}$$

0. Setting up the scoring matrix

	-	A	G	T
-	0			
A				
A				
G				
C				

Initialization:

- Top left: 0

Update Rule:

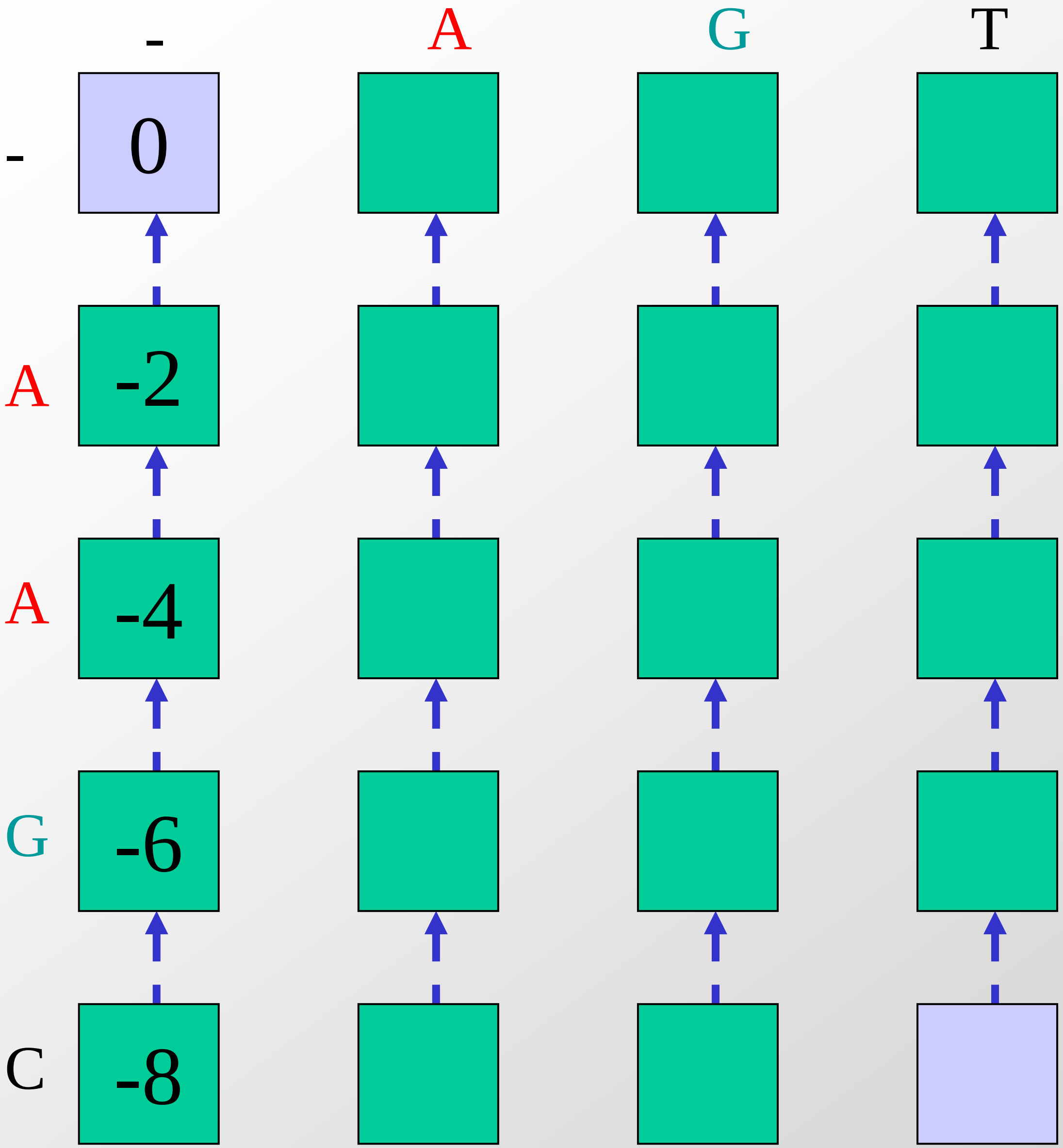
A(*i*,*j*)=max{

}

Termination:

- Bottom right

1. Allowing gaps in s



	-	A	G	T
-	0			
A	-2			
A	-4			
G	-6			
C	-8			

Initialization:

- Top left: 0

Update Rule:

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

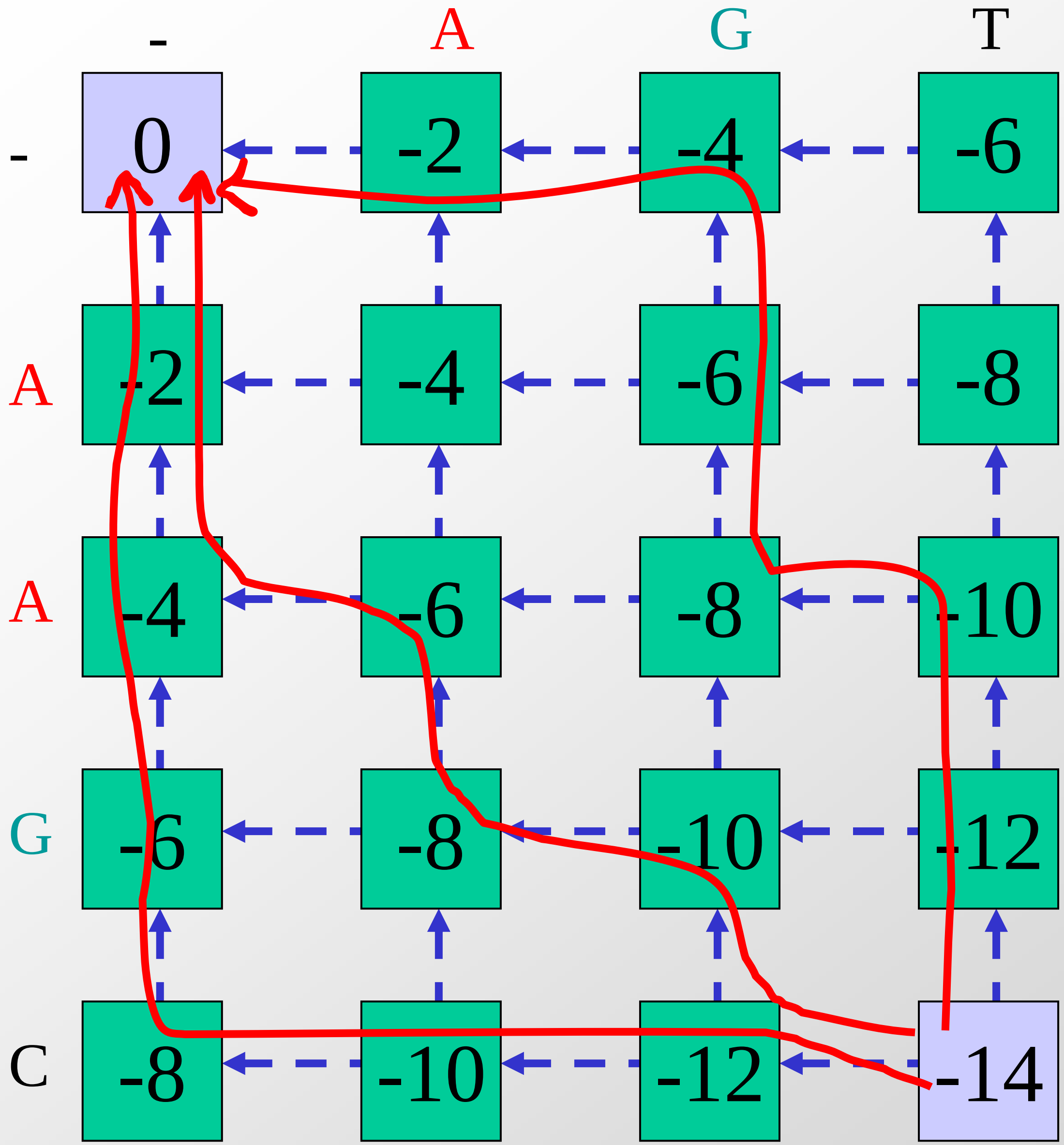
- A(*i*-1 , *j*) - 2

}

Termination:

- Bottom right

2. Allowing gaps in t



	-	A	G	T
-	0	-2	-4	-6
A	-2	-4	-6	-8
A	-4	-6	-8	-10
G	-6	-8	-10	-12
C	-8	-10	-12	-14

Initialization:

- Top left: 0

Update Rule:

A(i,j)=max{

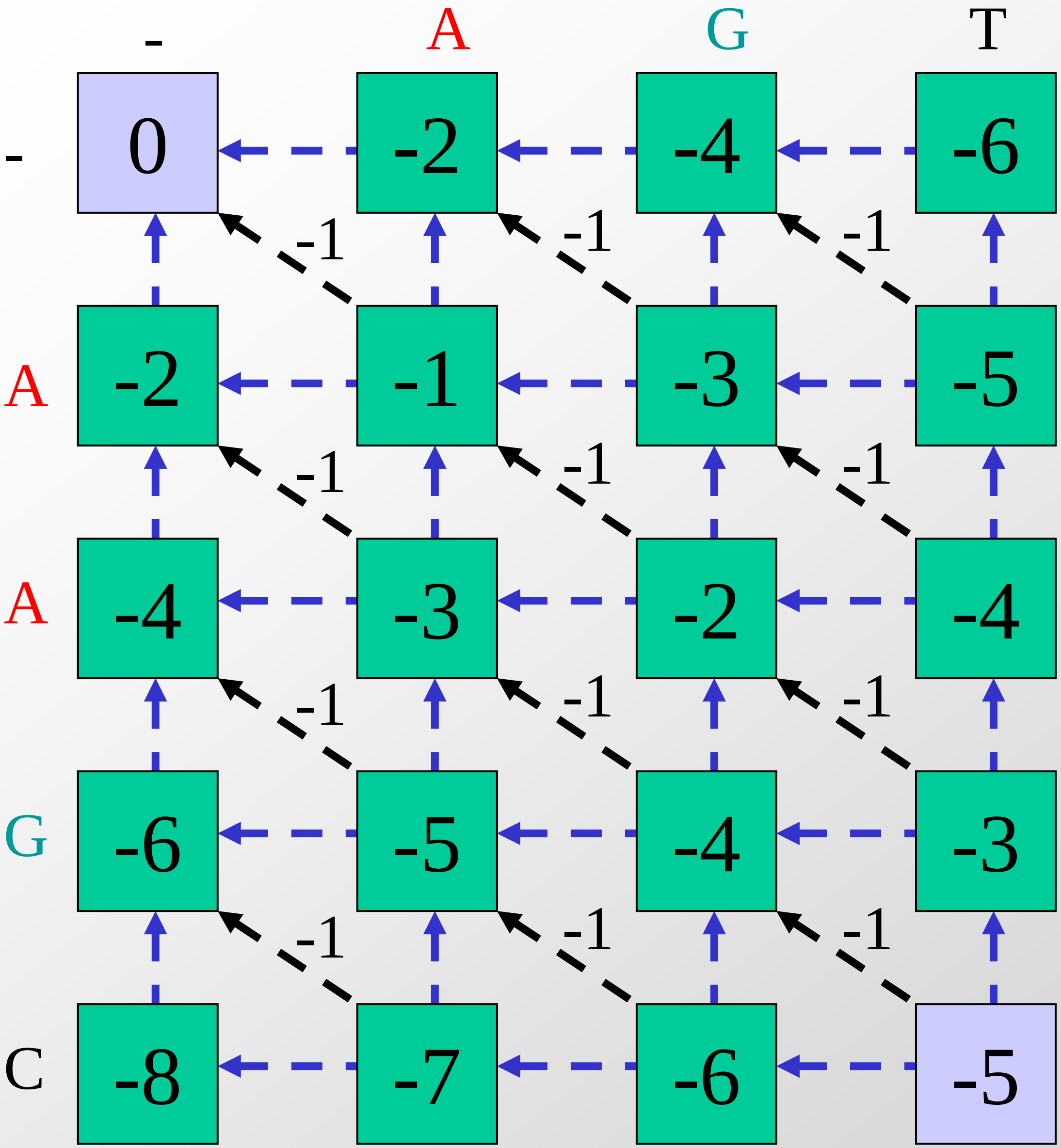
- A(i-1 , j) - 2
- A(i , j-1) - 2

}

Termination:

- Bottom right

3. Allowing mismatches



	-	A	G	T
-	0	-2	-4	-6
A	-2	-1	-3	-5
A	-4	-3	-2	-4
G	-6	-5	-4	-3
C	-8	-7	-6	-5

Initialization:

- Top left: 0

Update Rule:

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

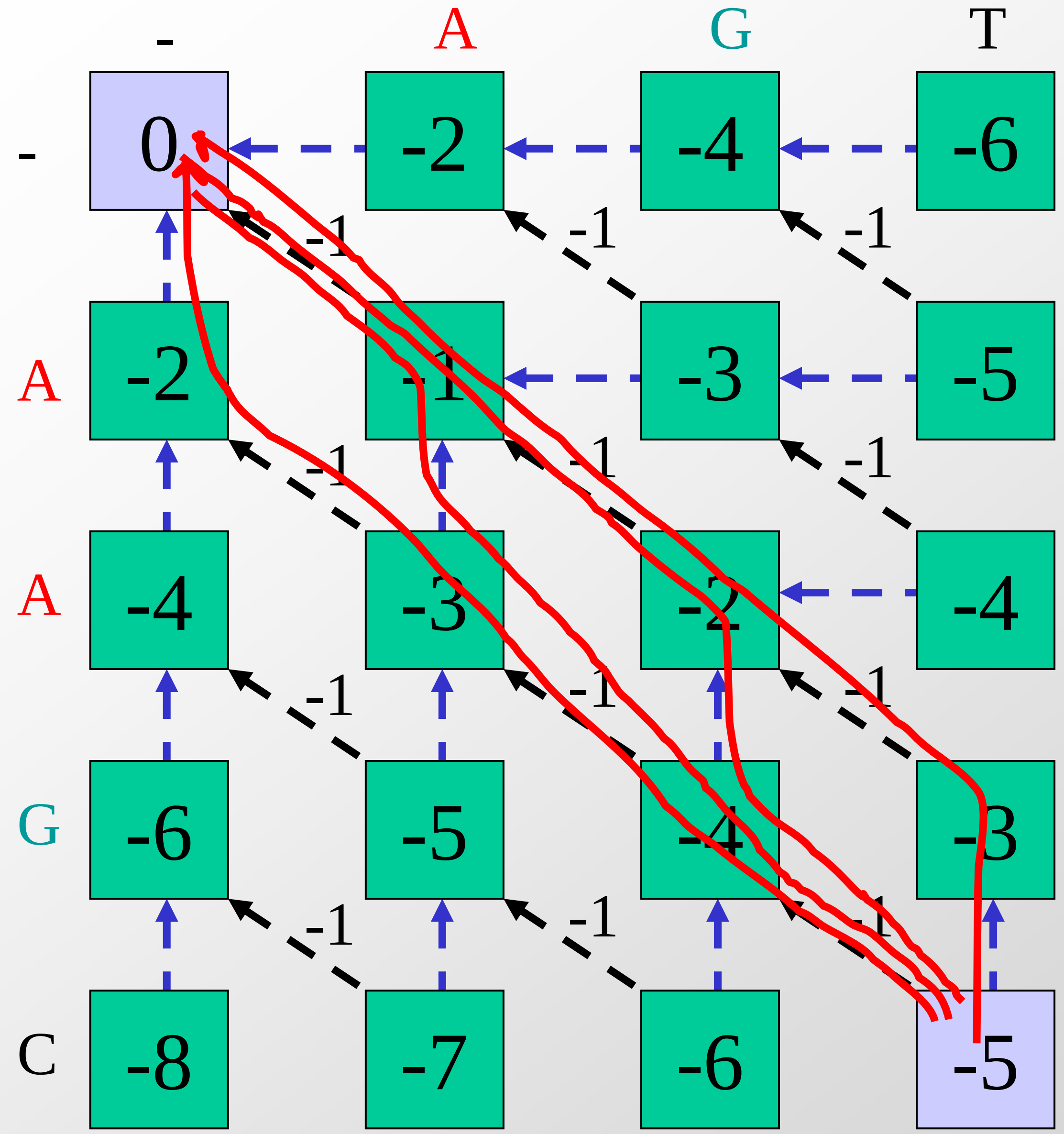
- A(*i*-1 , *j*) - 2
- A(*i* , *j*-1) - 2
- A(*i*-1 , *j*-1) -1

}

Termination:

- Bottom right

4. Choosing optimal paths



	-	A	G	T
-	0	-2	-4	-6
A	-2	-1	-3	-5
A	-4	-3	-2	-4
G	-6	-5	-4	-3
C	-8	-7	-6	-5

Initialization:

- Top left: 0

Update Rule:

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

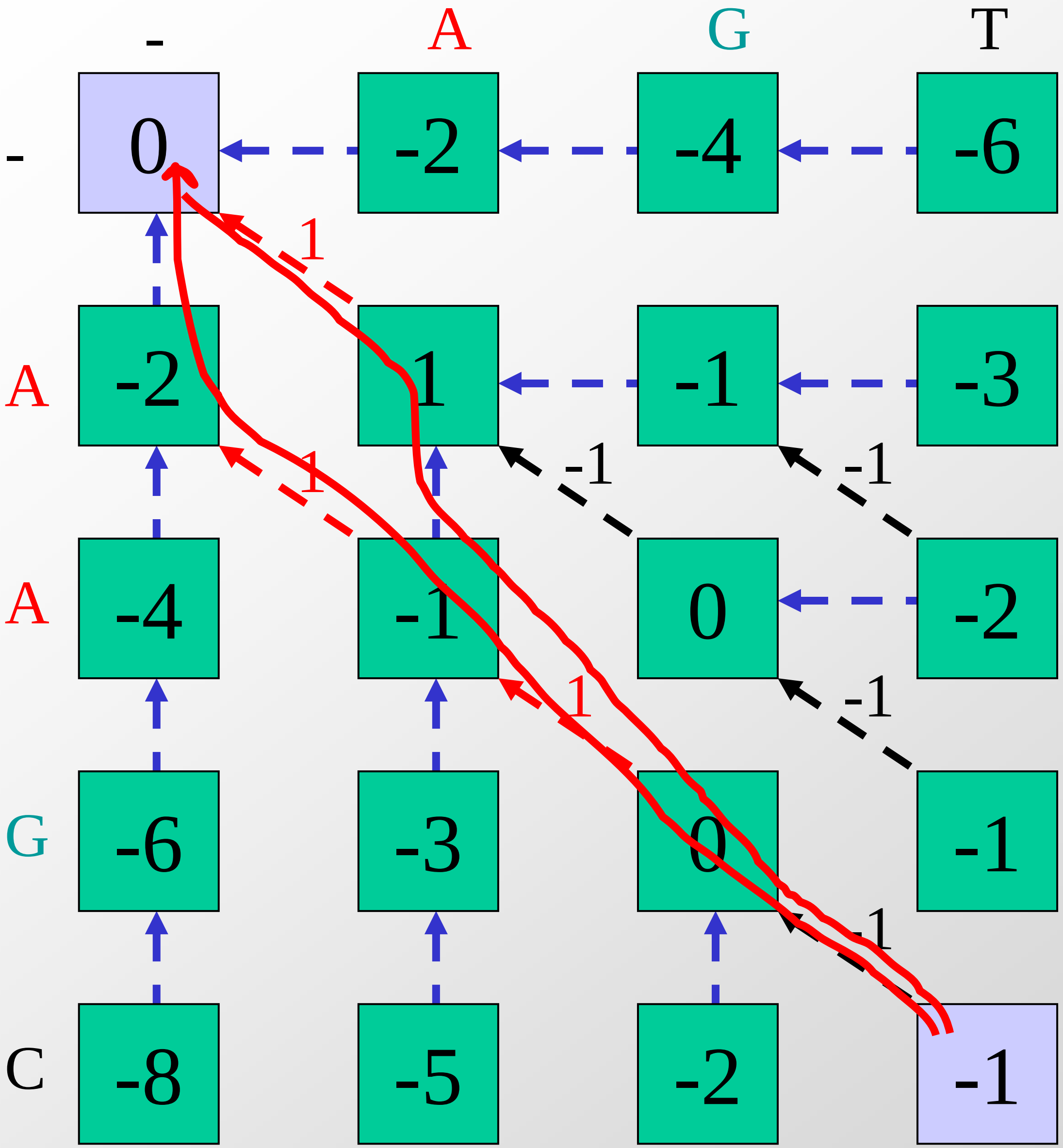
- A(*i*-1 , *j*) - 2
- A(*i* , *j*-1) - 2
- A(*i*-1 , *j*-1) -1

}

Termination:

- Bottom right

5. Rewarding matches



	-	A	G	T
-	0	-2	-4	-6
A	-2	1	-1	-3
A	-4	-1	0	-2
G	-6	-3	0	-1
C	-8	-5	-2	-1

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

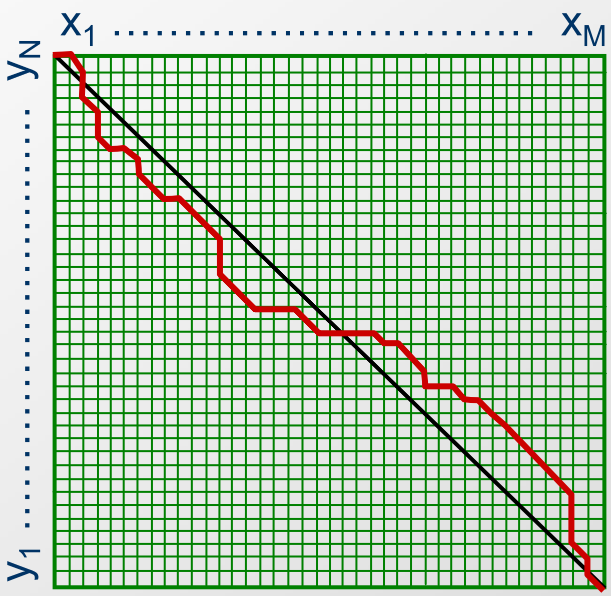
}

Termination:

- Bottom right

What is missing? (5) Returning the actual path!

- We know how to compute the best score
  - – Simply the number at the bottom right entry
- But we need to remember where it came from
  - – Pointer to the choice we made at each step
- Retrace path through the matrix
  - – Need to remember all the pointers



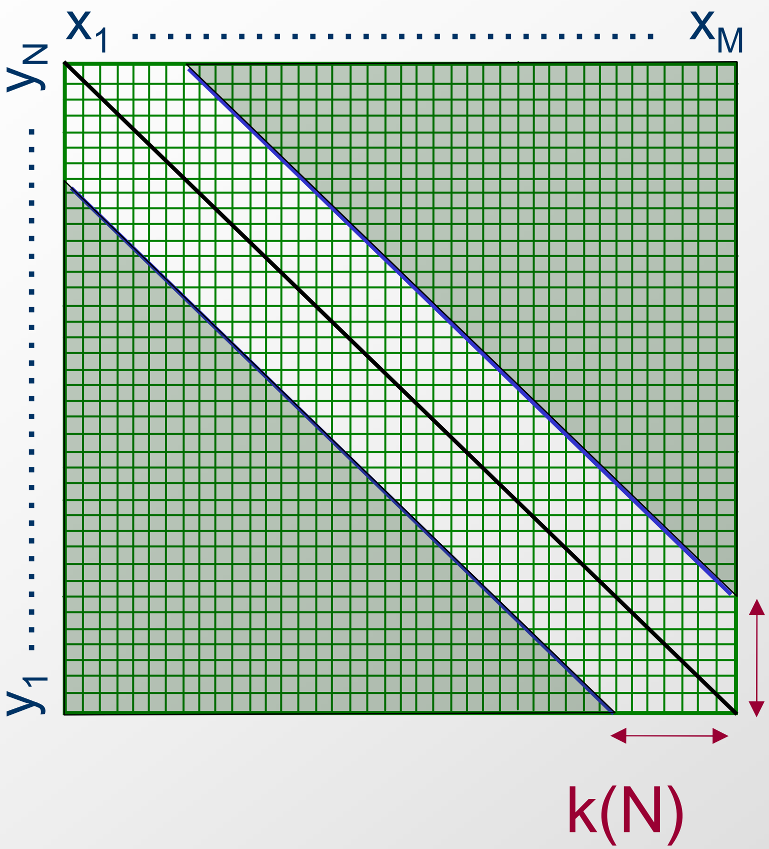
Time needed: O(m*n)

Space needed: O(m*n)

Summary

- Dynamic programming
  - Reuse of computation
  - Order sub-problems. Fill table of sub-problem results
  - Read table instead of repeating work (ex: Fibonacci)
- Sequence alignment
  - Edit distance and scoring functions
  - Dynamic programming matrix
  - Matrix traversal path ⬄ Optimal alignment
- Thursday: Variations on sequence alignment
  - Local/global alignment, affine gaps, algo speed-ups
  - Semi-numerical alignment, hashing, database lookup
- Recitation:
  - Dynamic programming applications
  - Probabilistic derivations of alignment scores

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)
```

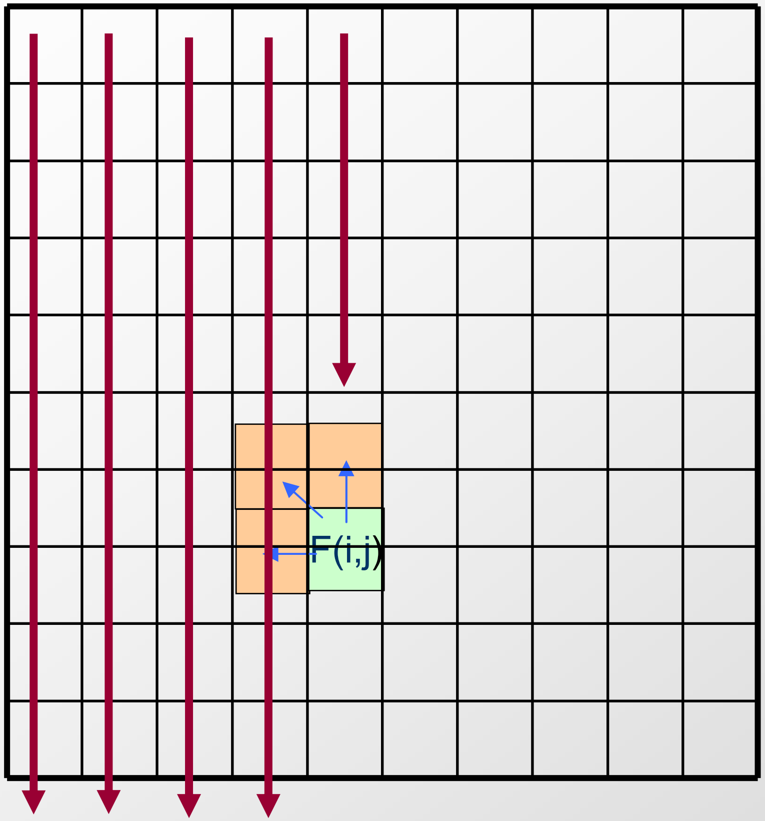
$$F(i, j) = \max \begin{cases} F(i-1, j-1) + s(x_i, y_j) \\ F(i, j-1) - d, \text{ if } j > i - k(N) \\ F(i-1, j) - d, \text{ if } j < i + k(N) \end{cases}$$

Termination: same

Slides credit: Serafim Batzoglou

Linear space alignment

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



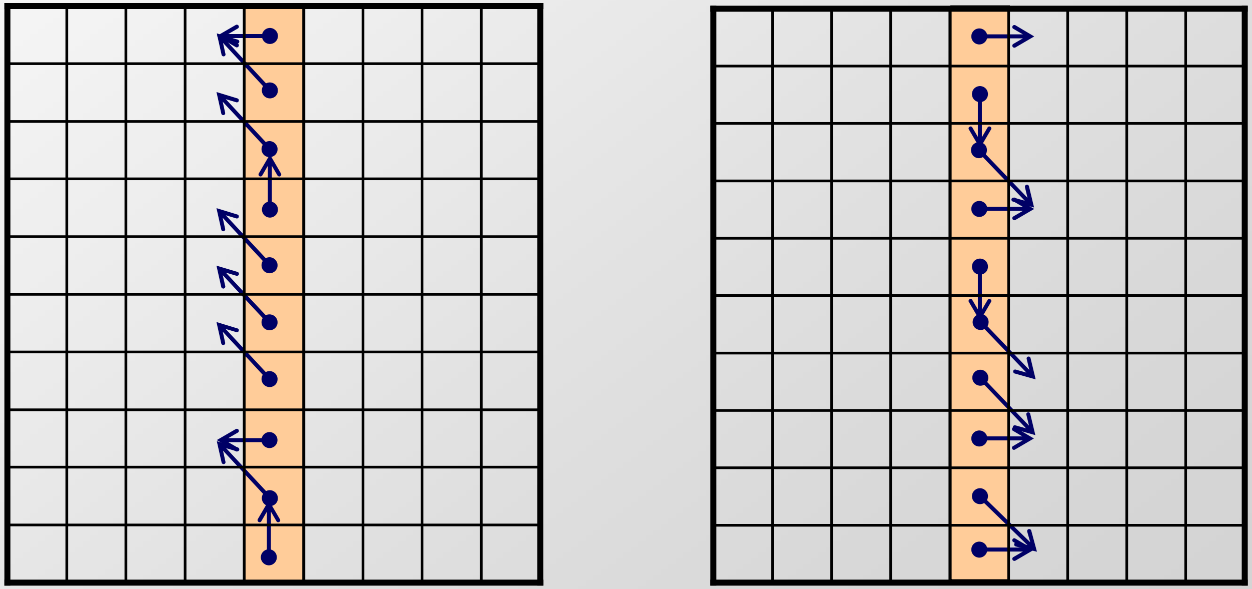
```
Allocate ( column[1] )
Allocate ( column[2] )

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

What about the pointers?

Finding the best back-pointer for current column

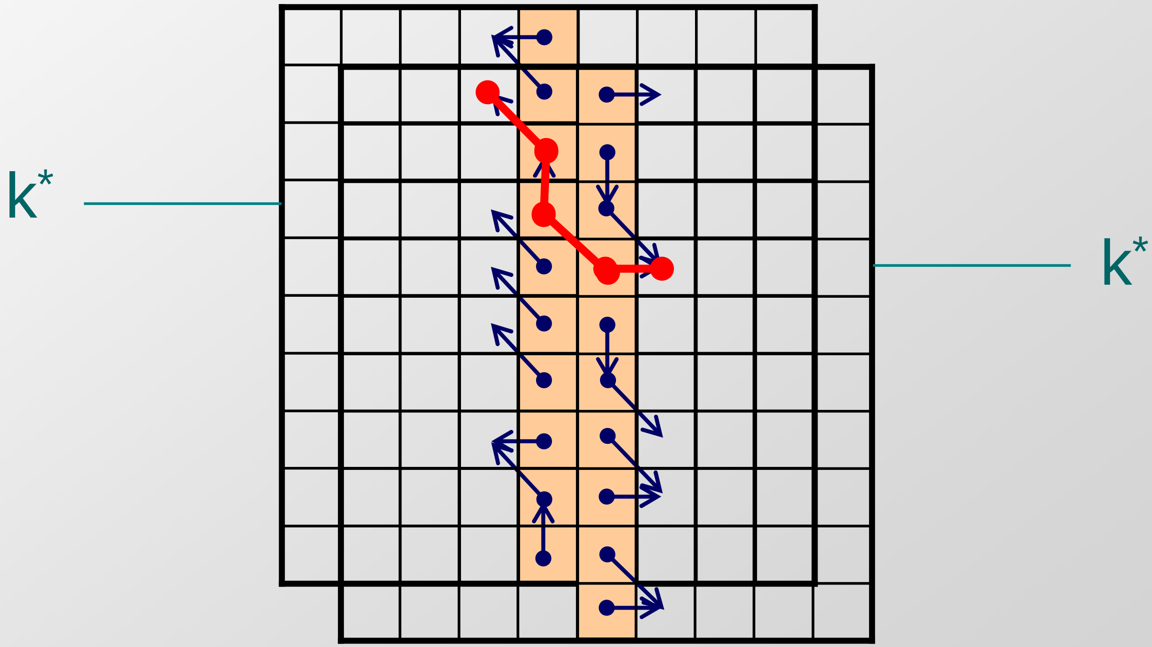
- Now, using 2 columns of space, we can compute for k = 1…M, F(M/2, k), F^r(M/2, N-k)

  PLUS the backpointers



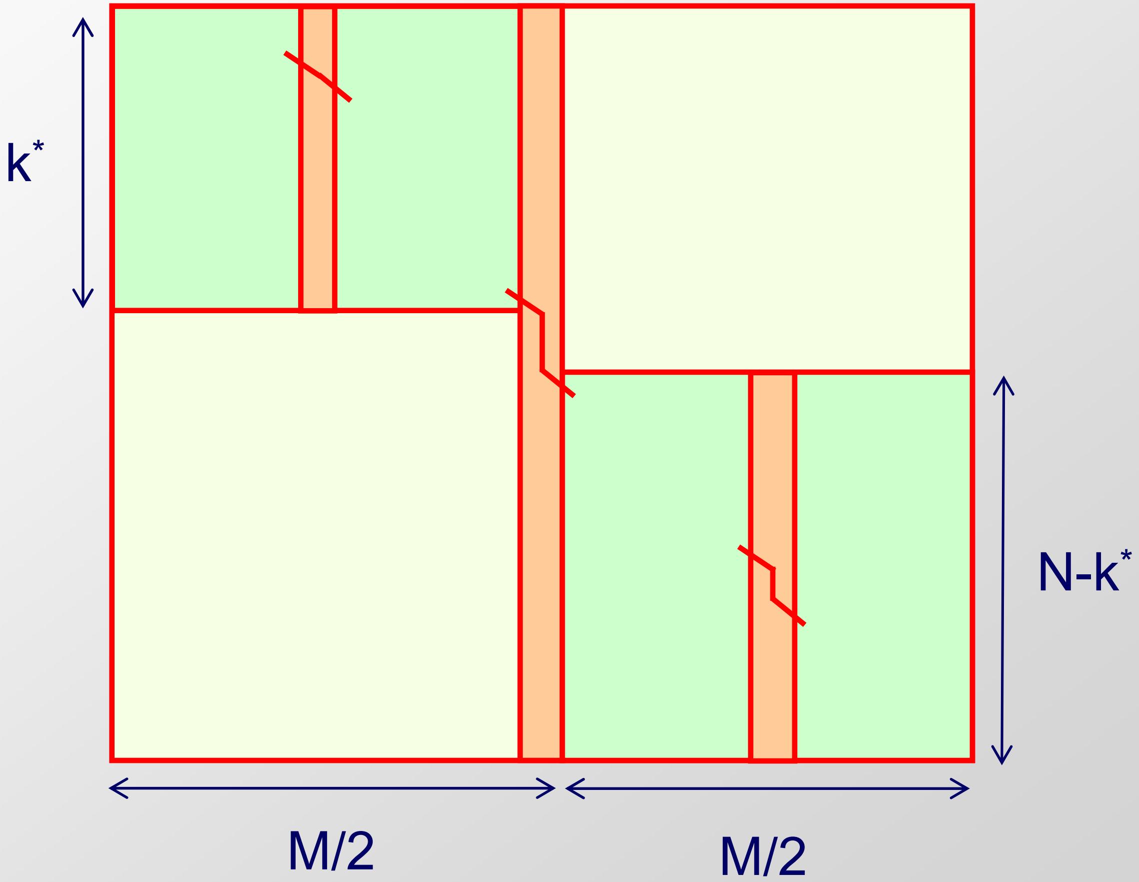
Best forward-pointer for current column

- 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^*

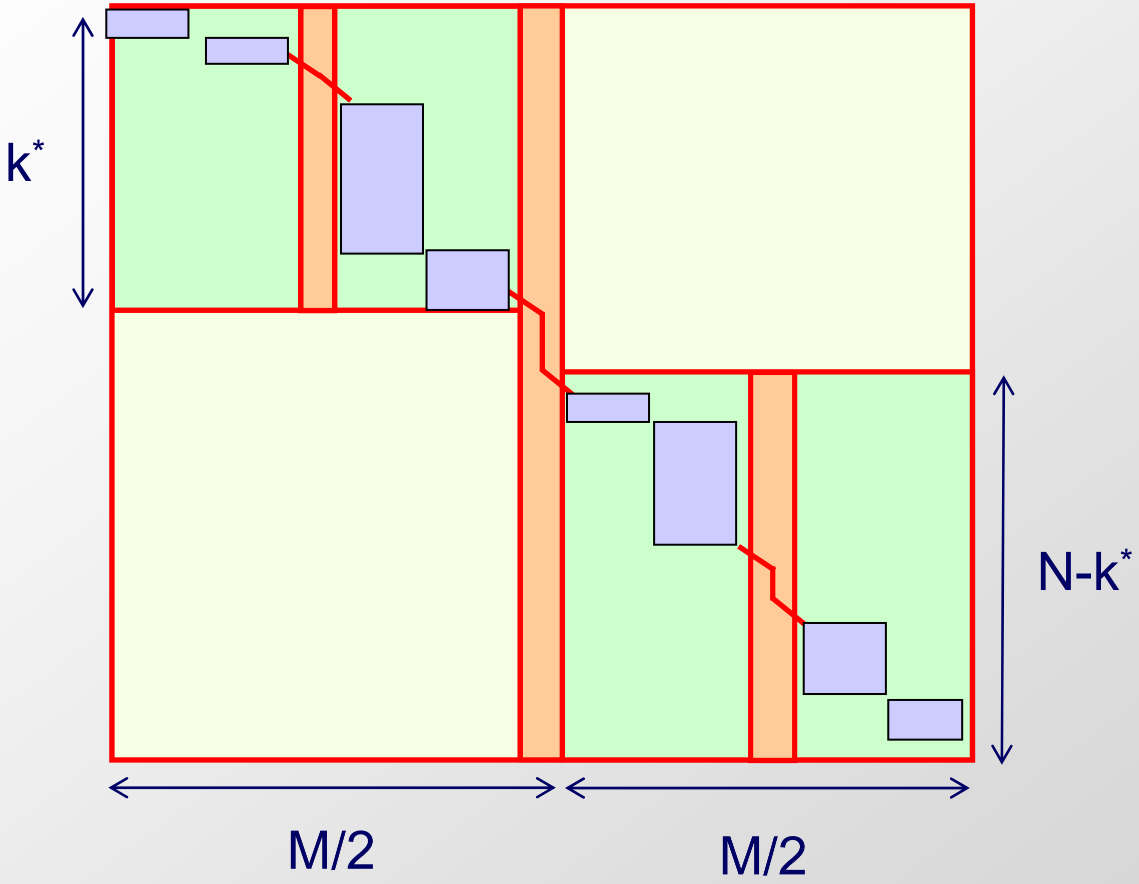


Recursively find midpoint for left & right

- Iterate this procedure to the left and right!



Total time cost of linear-space alignment



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

Total Space: O(N) for computation,
O(N+M) to store the optimal alignment

Formulation 4: Varying gap cost models (next time)

(still) Varying penalties for edit operations

Now allow gaps of varying penalty:

1. Linear gap penalty
   - Same as before,
2. Affine gap penalty
   - Big initial cost for starting or ending a gap
   - Small incremental cost for each additional character
3. General gap penalty
   - Any cost function
   - No longer computable using the same model
4. Seek duplicated regions, rearrangements, …