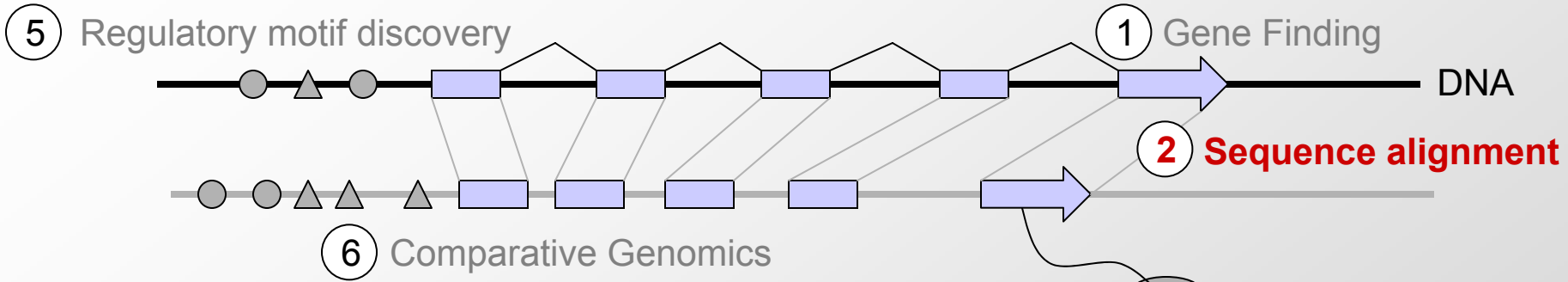
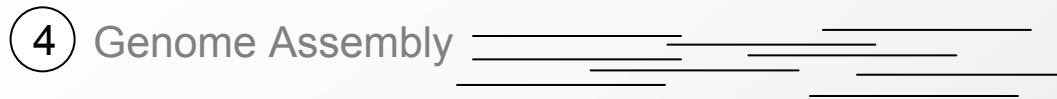


6.095/6.895 - Computational Biology: Genomes, Networks, Evolution

Sequence Alignment and Dynamic Programming

Tue Sept 13, 2005

Challenges in Computational Biology



① Gene Finding

② Sequence alignment

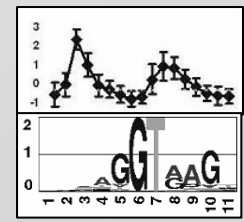
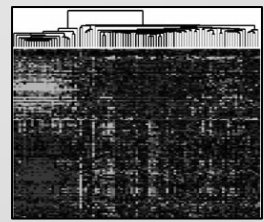
⑥ Comparative Genomics

⑦ Evolutionary Theory



③ Database lookup

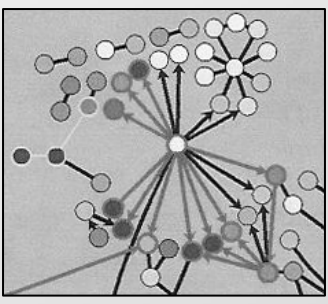
⑧ Gene expression analysis



⑨ Cluster discovery

⑩ Gibbs sampling

⑪ Protein network analysis



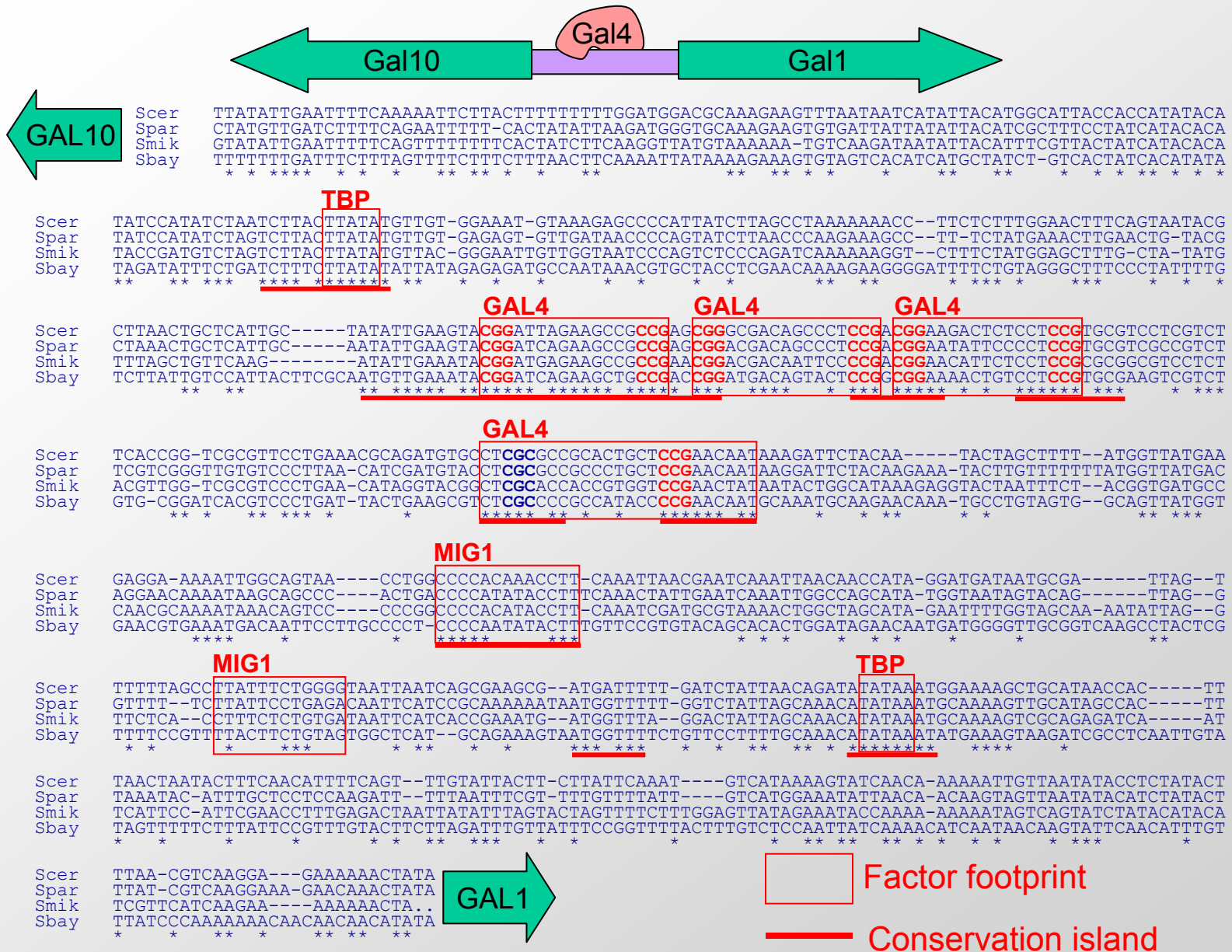
⑫ Regulatory network inference

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 computational biology
- First problem: Motif discovery
 - Counting motif instances across the genome
 - Counting conserved motif instances
 - Problem set: discover motifs in actual yeast genome

Counting 'conserved' motif instances



We can 'read' evolution to reveal functional elements

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	X	G	T	G	X	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

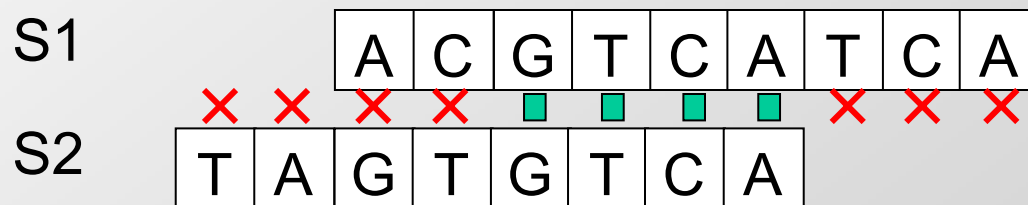
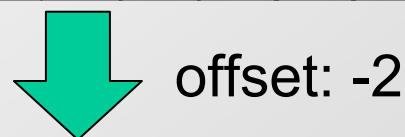
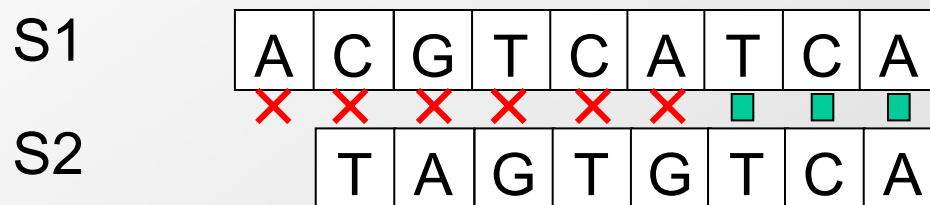
T	A	G	T	G	T	C	A
---	---	---	---	---	---	---	---

Question 1: Aligning two (ungapped) strings

- 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



Scoring function:

Match(x,x) = +1

Mismatch(A,G) = -1/2

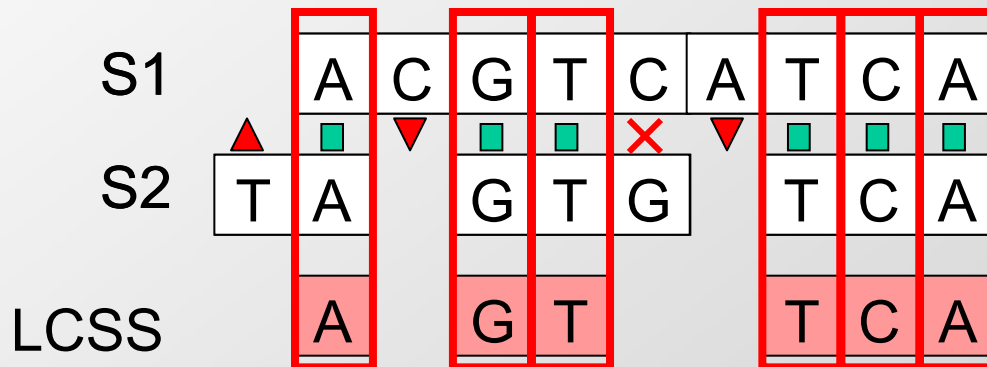
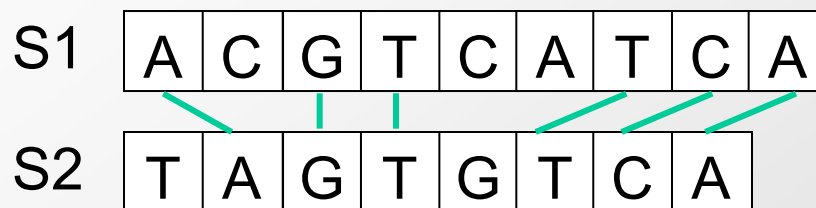
Mismatch(C,T) = -1/2

Mismatch(x,y) = -1

	A	G	T	C
A	+1	-1/2	-1	-1
G	-1/2	+1	-1	-1
T	-1	-1	+1	-1/2
C	-1	-1	-1/2	+1

Q2: Aligning two (possibly gapped) sequences

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



Edit distance:

- Number of changes needed for $S1 \rightarrow S2$
- Uniform scoring function

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 vs. CT)
 - 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?

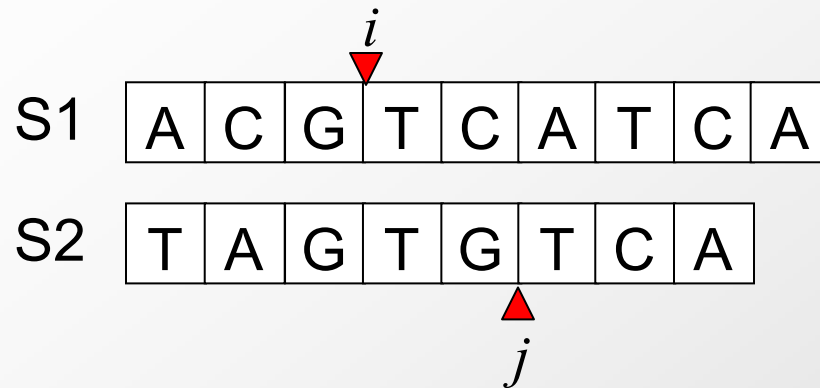
- Ways to align two sequences of length m, n

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

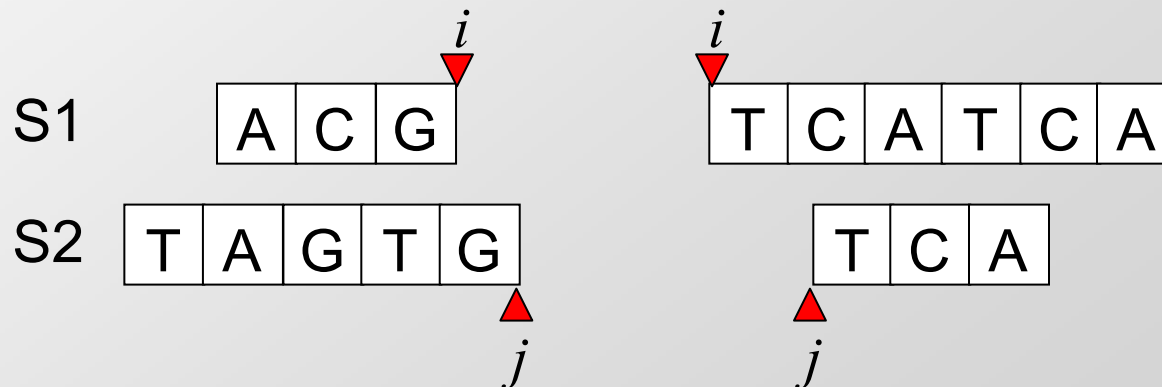
- For two sequences of length n

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

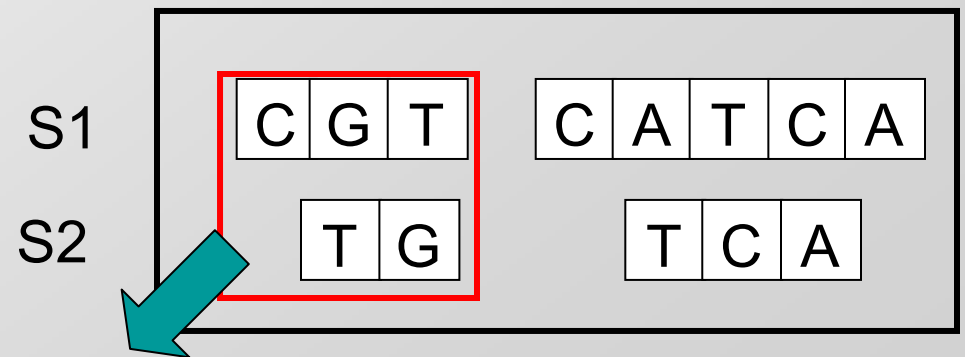
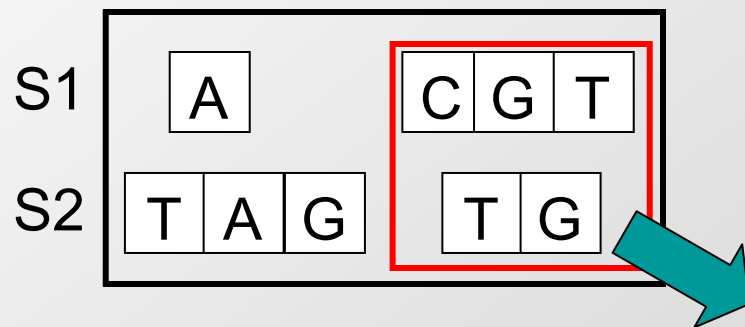
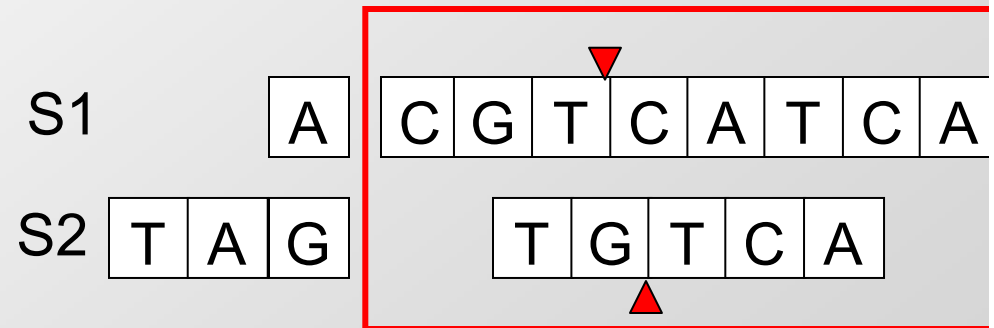
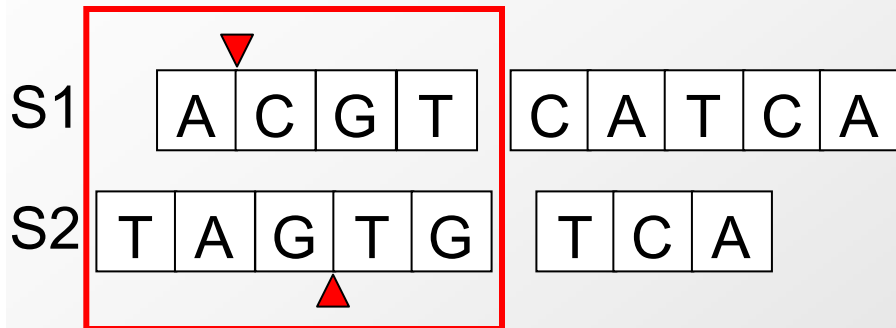
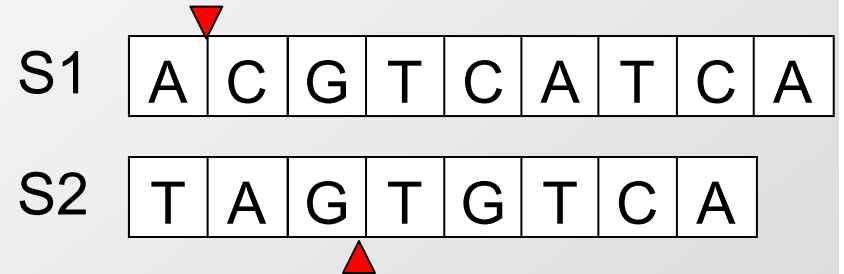
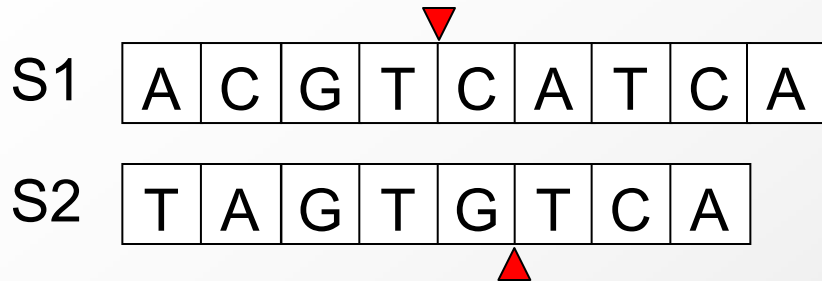
Key insight: score is additive!



- 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]



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

A simple introduction to Dynamic Programming

- Fibonacci numbers

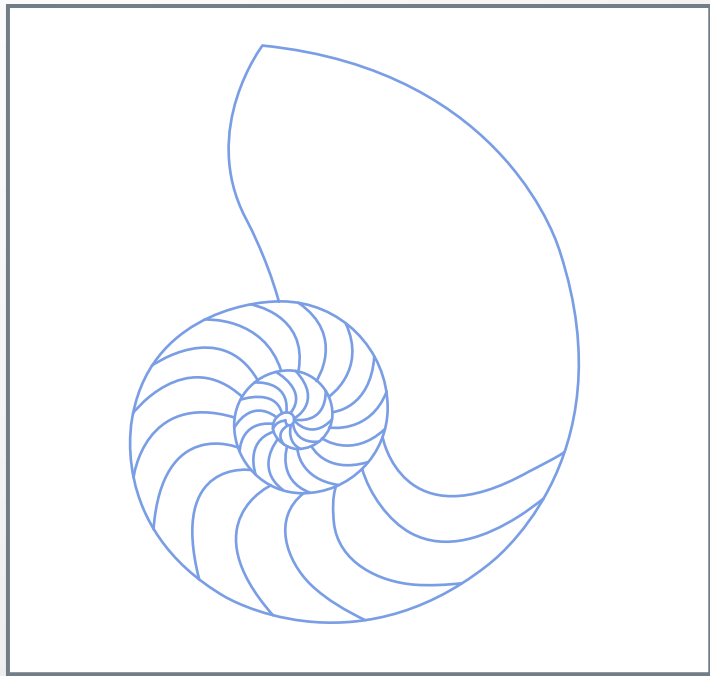
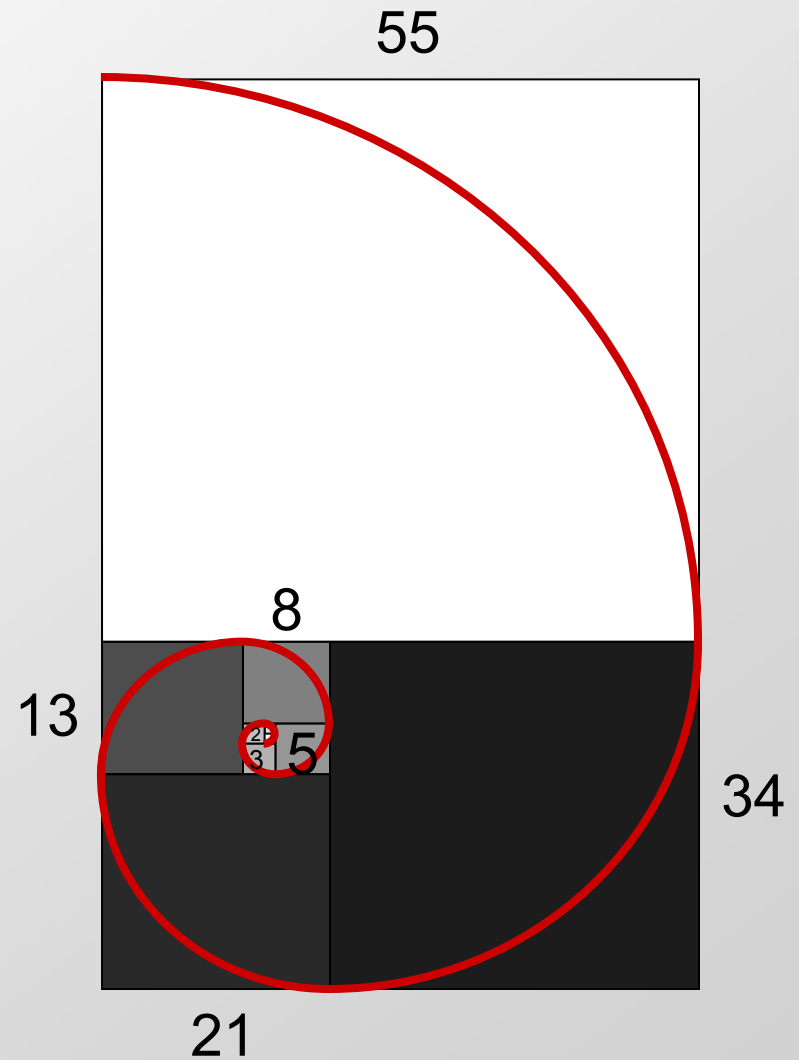
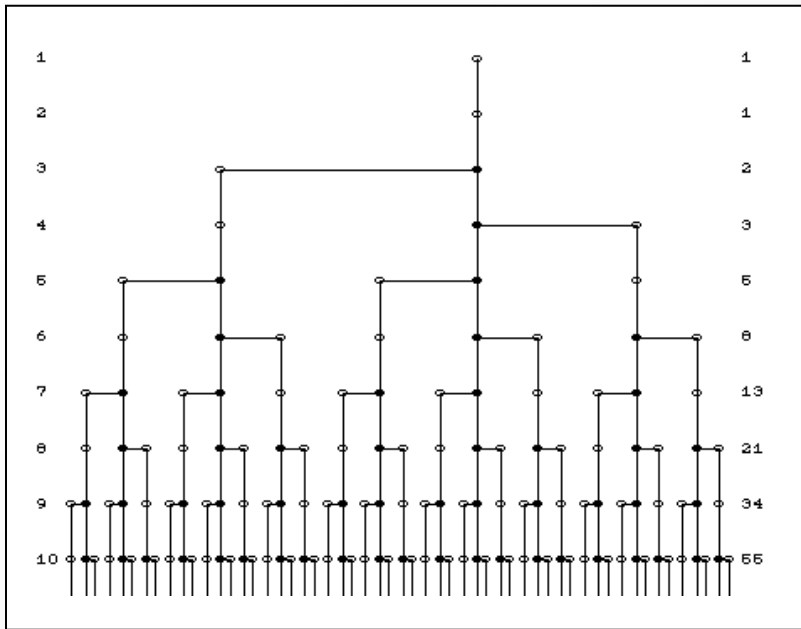


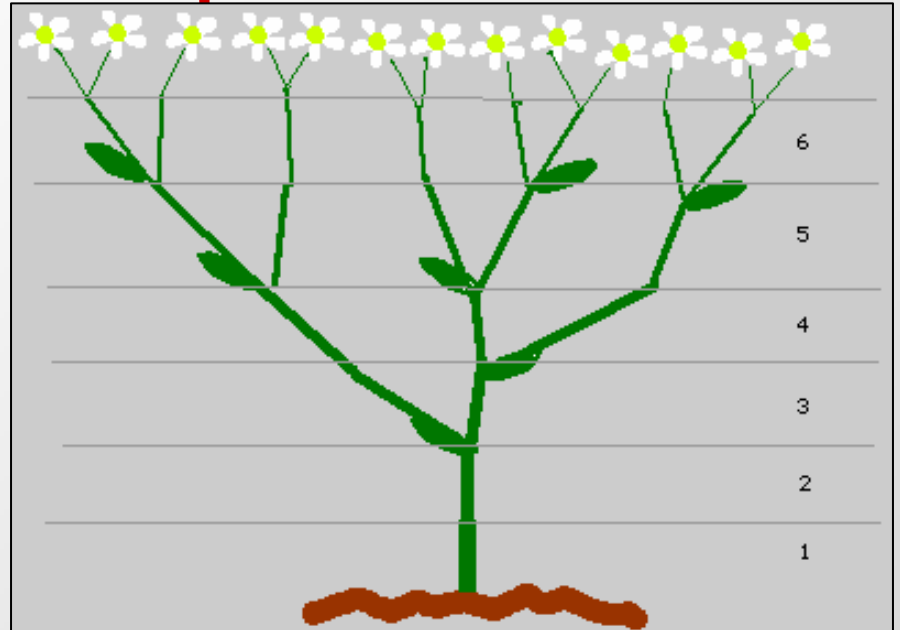
Figure by MIT OCW.



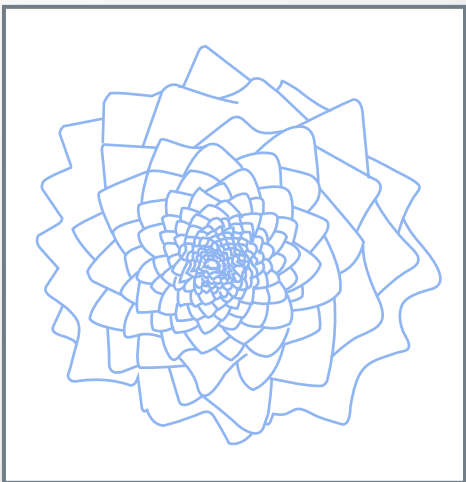
Fibonacci numbers are ubiquitous in nature



Rabbits per generation

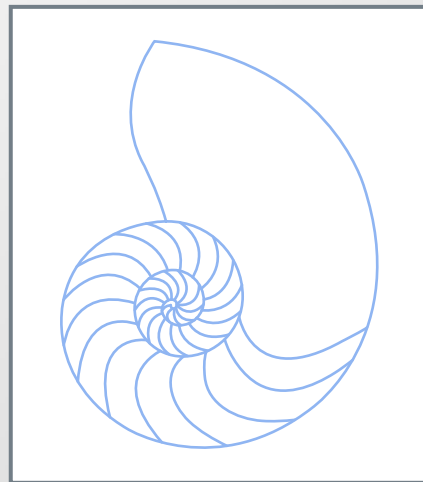


Leaves per height



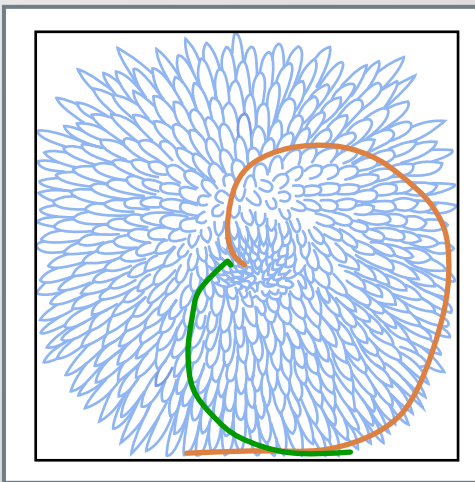
Romanesque spirals

Figure by MIT OCW.



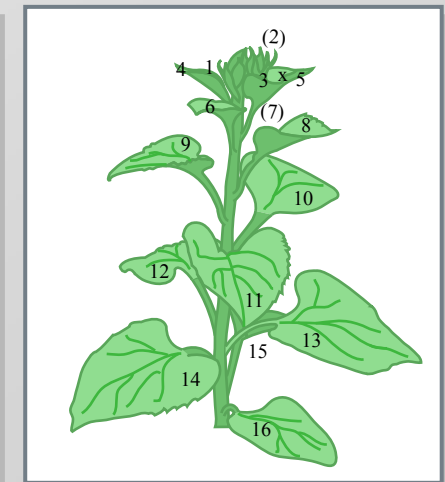
Nautilus size

Figure by MIT OCW.



Coneflower spirals

Figure by MIT OCW.



Leaf ordering

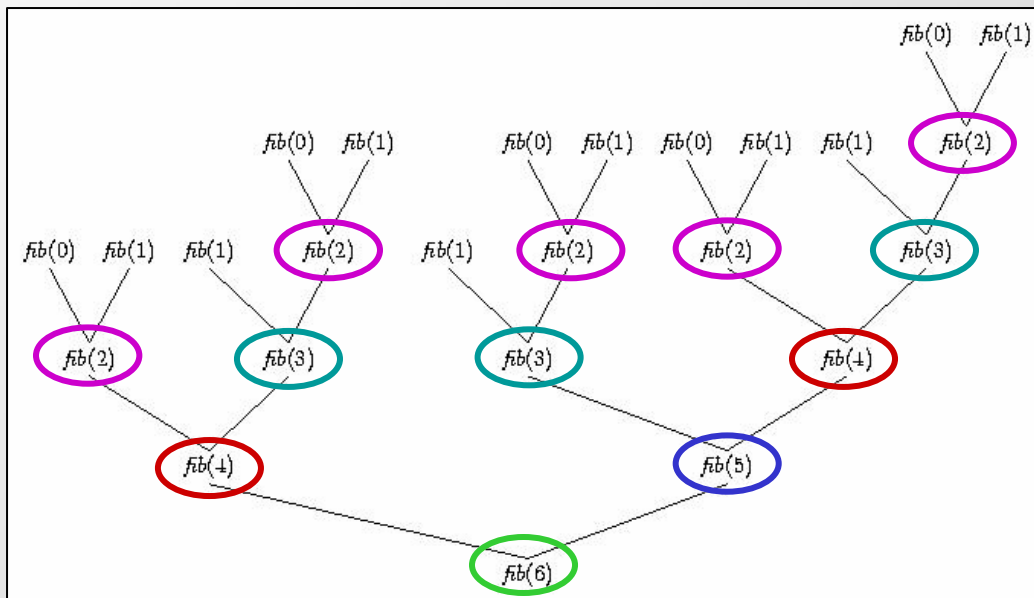
Figure by MIT OCW.

Computing Fibonacci numbers: Top down

- Goal: Compute nth 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,...
- Top-down approach:
 - Python code

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

- Analysis: $T(n) = T(n-1) + T(n-2) = (\dots) = O(2^n)$



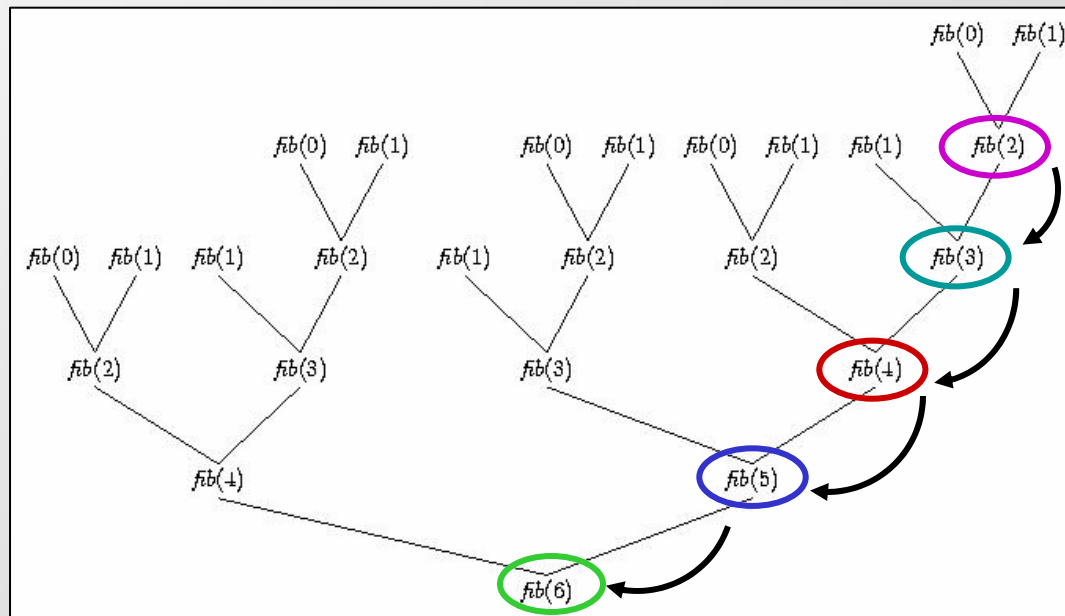
Computing Fibonacci numbers: Bottom up

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



What have we learned?

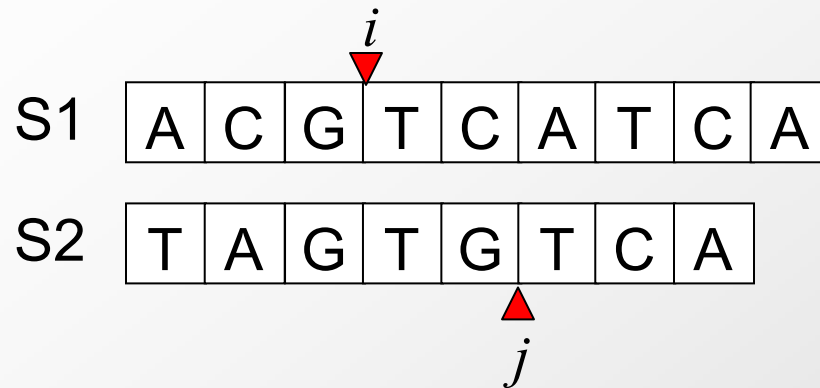
fib_table	
F[1]	1
F[2]	1
F[3]	2
F[4]	3
F[5]	5
F[6]	8
F[7]	13
F[8]	21
F[9]	34
F[10]	55
F[11]	89
F[12]	?



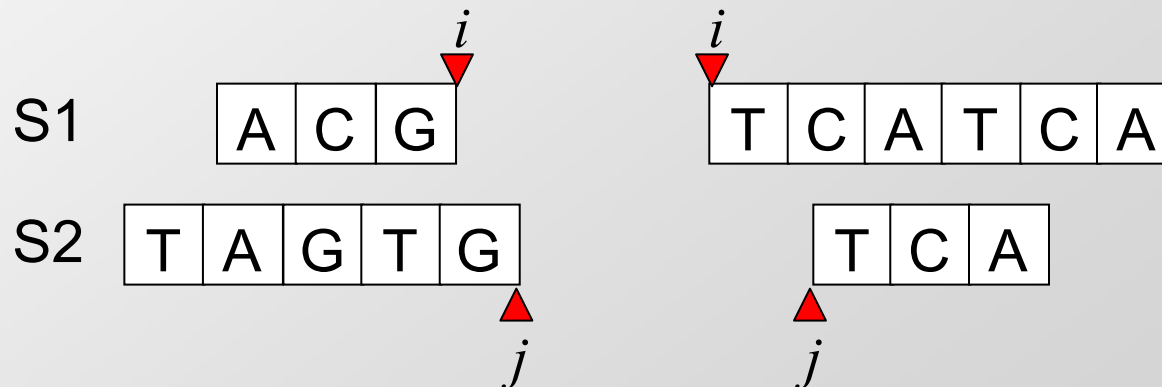
- Principles of dynamic programming
 - Reveal identical sub-problems
 - Order computation to enable result reuse
 - Systematically fill in table of results
 - Express larger problems from their subparts
- Ordering of computations matters
 - Naïve top-down approach very slow
 - results of smaller problems 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.

**How do we apply dynamic programming
to sequence alignment ?**

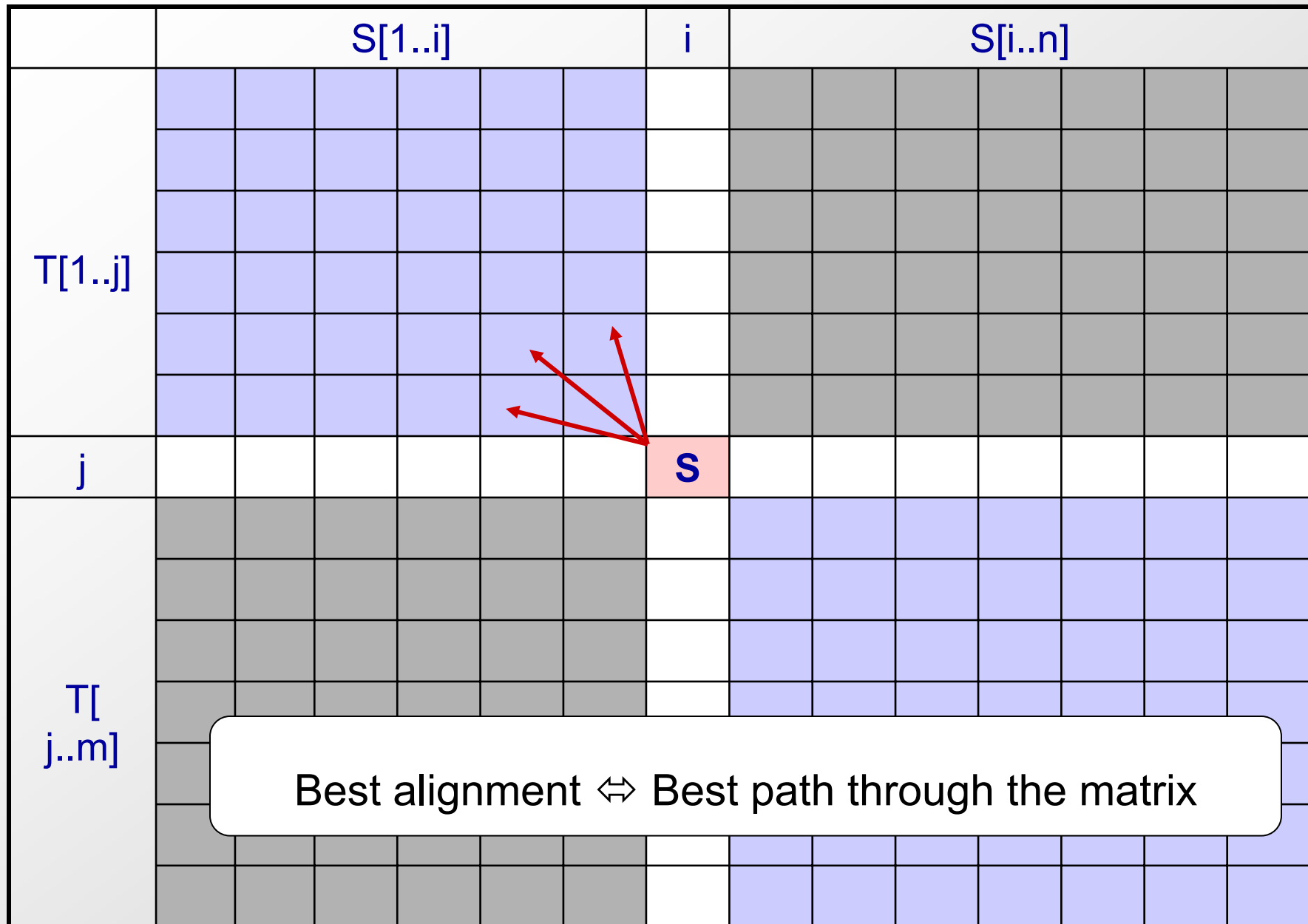
Key insight: score is additive!



- 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]



Store score of aligning (i,j) in matrix $M(i,j)$

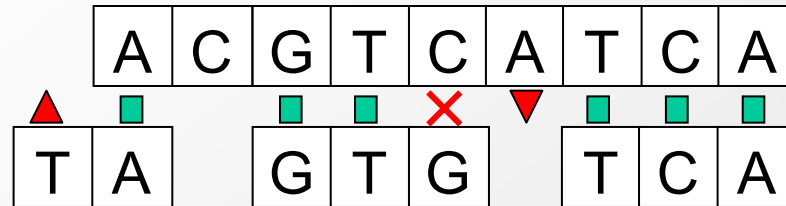


Filling in the dynamic programming matrix

- **Local update rules:**

- Compute next alignment based on previous alignment
- Just like Fibonacci numbers: $F[i] = F[i-1] + F[i-2]$
- Table lookup!

Duality: seq. alignment \Leftrightarrow path through the matrix



Goal:
Find best path
through the matrix

0. Setting up the scoring matrix

	-	A	G	T
-	0			
A				
A				
G				
C				

Initialization:

- Top right: 0

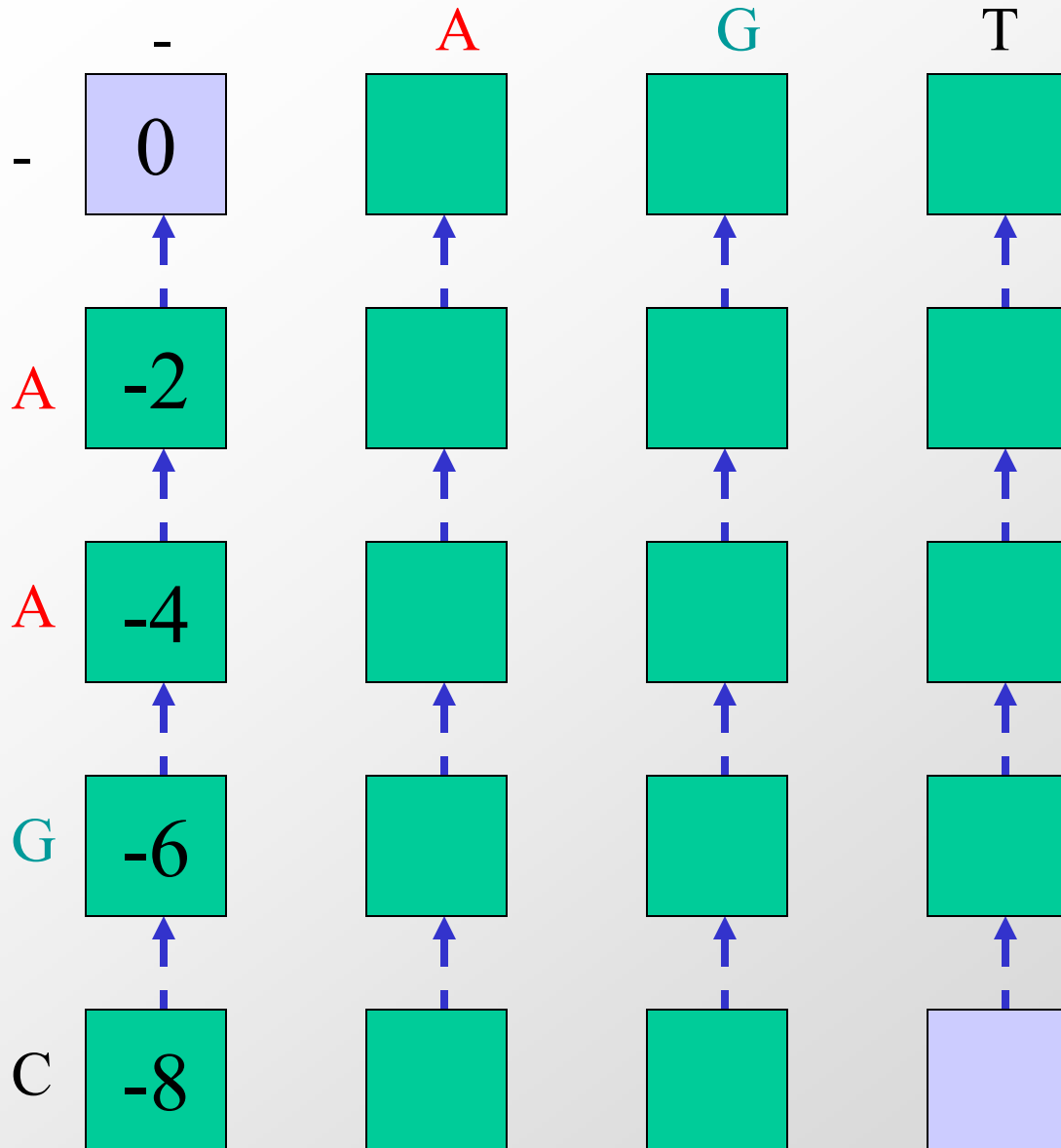
Update Rule:

$$A(i,j) = \max\{$$

Termination:

- Bottom right

1. Allowing gaps in s



Initialization:

- Top right: 0

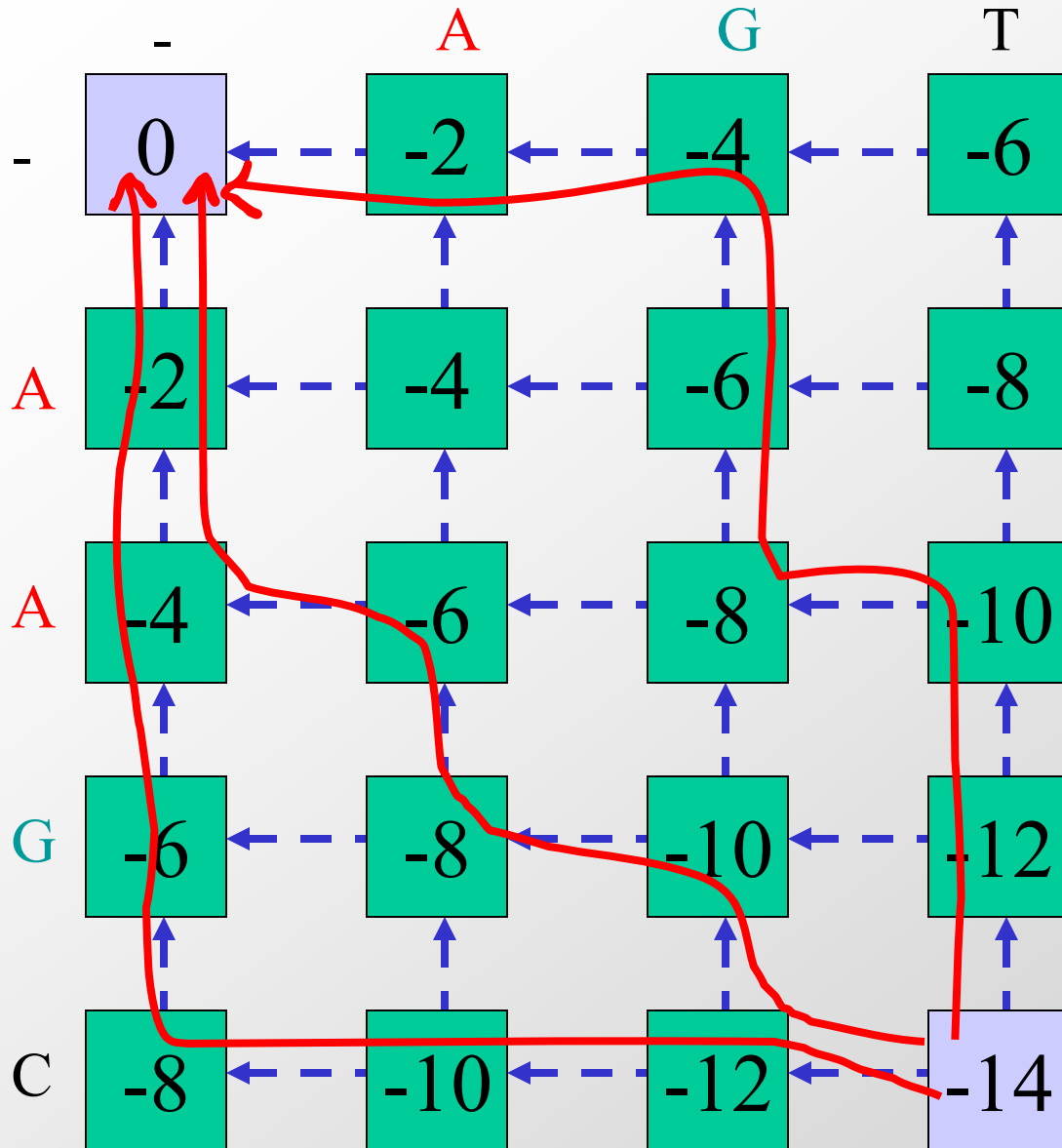
Update Rule:

$$A(i,j) = \max\{\begin{aligned} & \dots \\ & \bullet A(i-1, j) - 2 \end{aligned}$$

Termination:

- Bottom right

2. Allowing gaps in t



Initialization:

- Top right: 0

Update Rule:

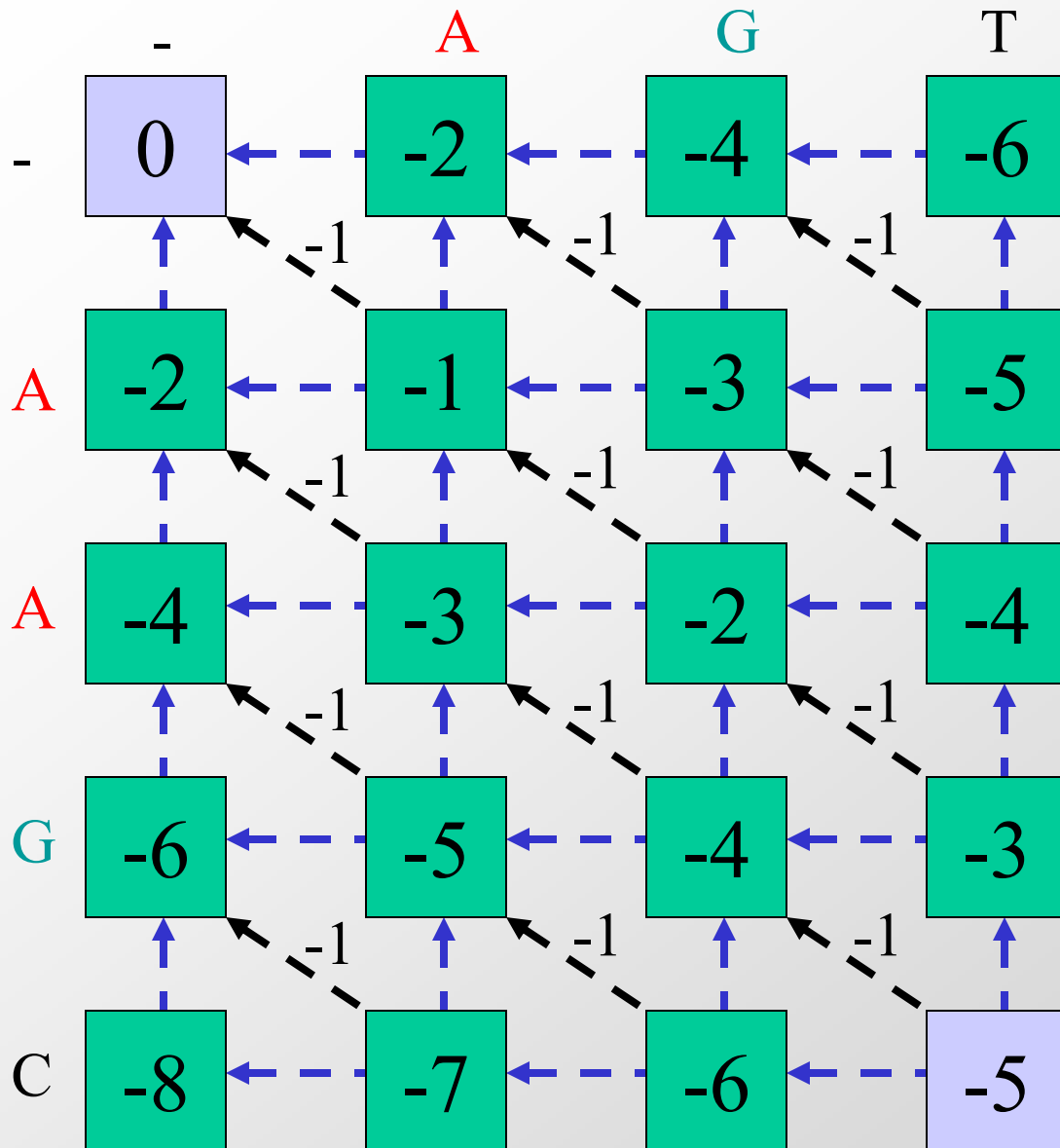
$$A(i,j) = \max\{$$

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

Termination:

- Bottom right

3. Allowing mismatches



Initialization:

- Top right: 0

Update Rule:

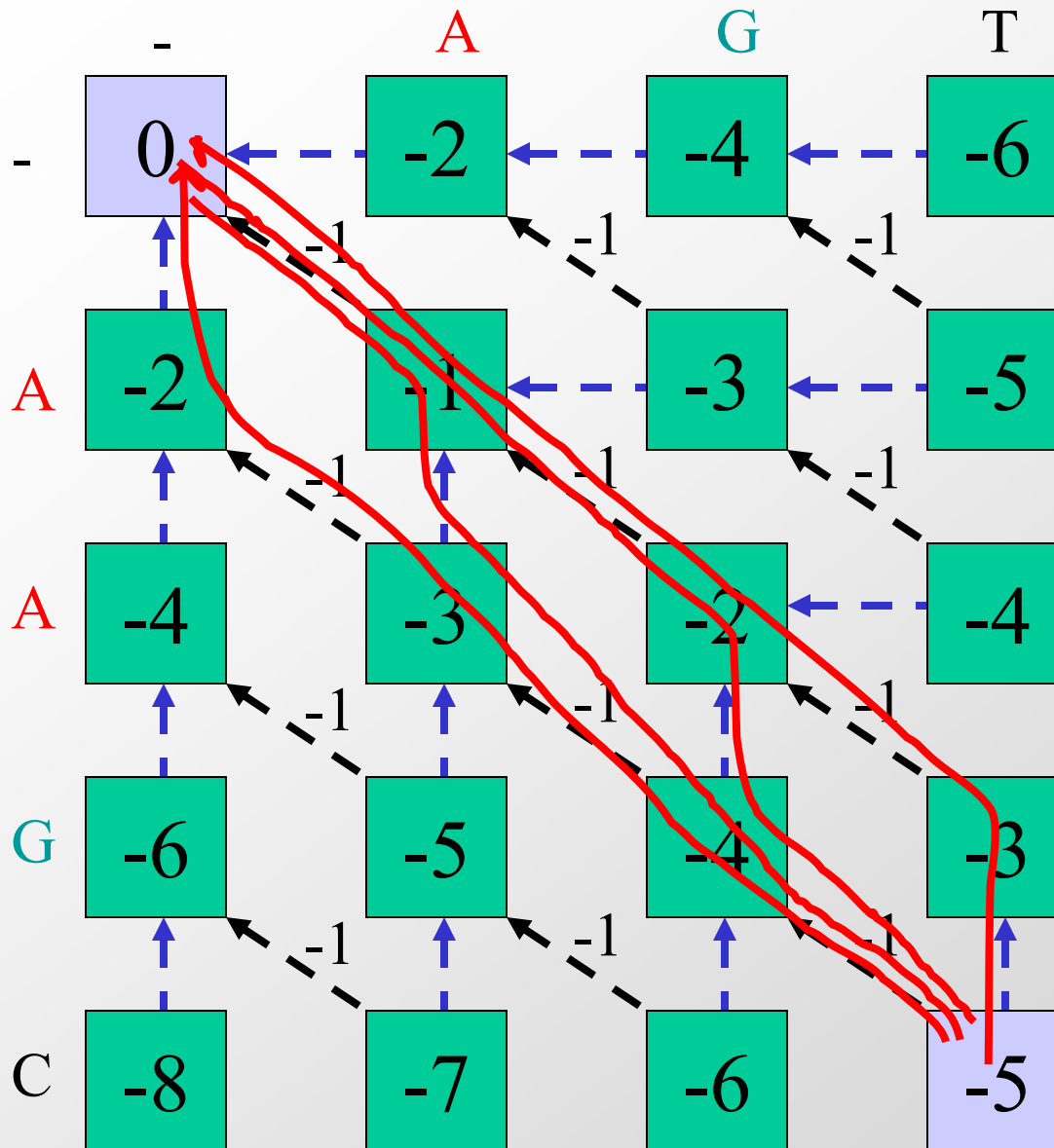
$$A(i,j) = \max\{$$

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

Termination:

- Bottom right

4. Choosing optimal paths



Initialization:

- Top right: 0

Update Rule:

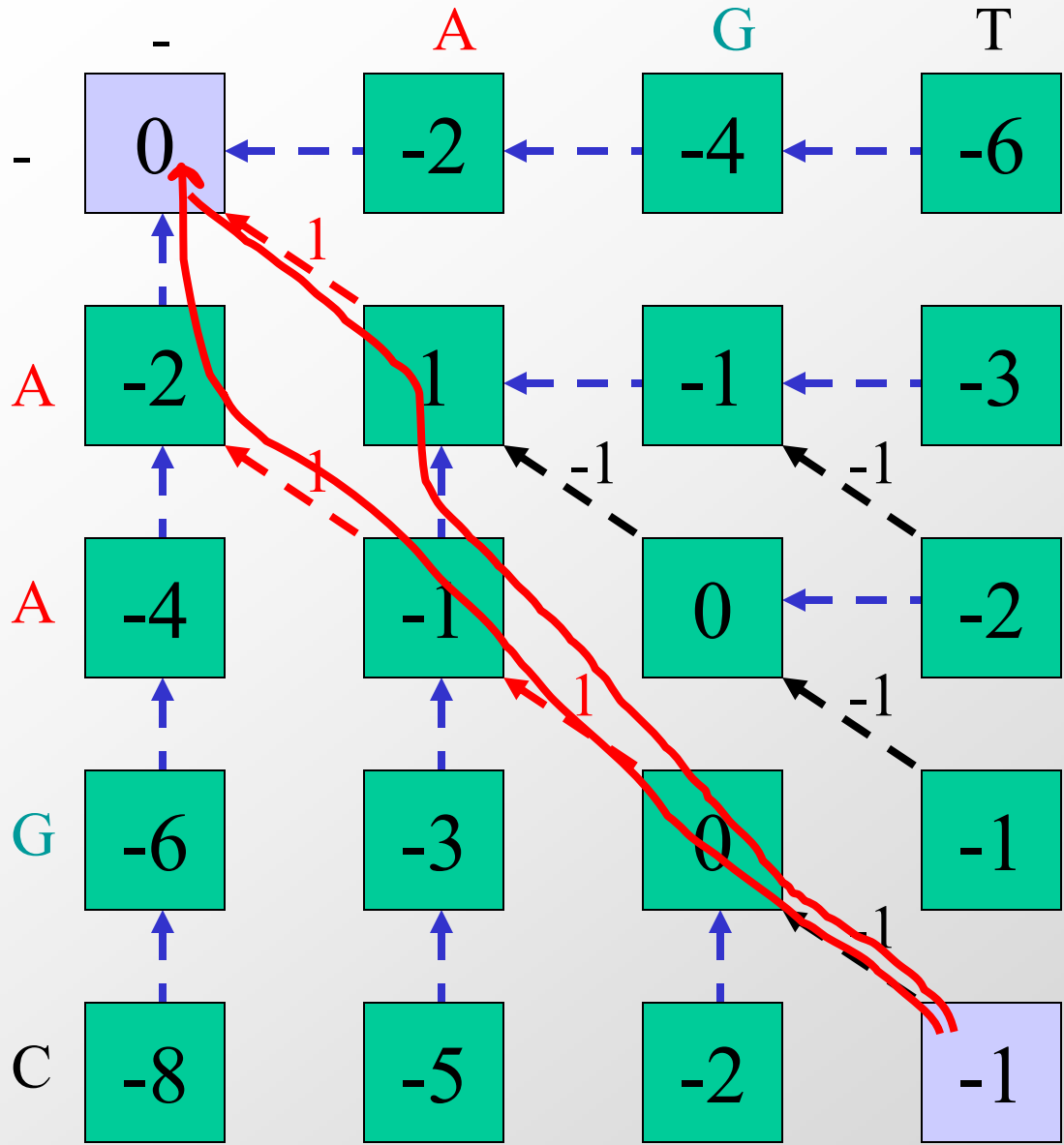
$$A(i,j) = \max\{$$

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

Termination:

- Bottom right

5. Rewarding matches



Initialization:

- Top right: 0

Update Rule:

$$A(i,j) = \max\{$$

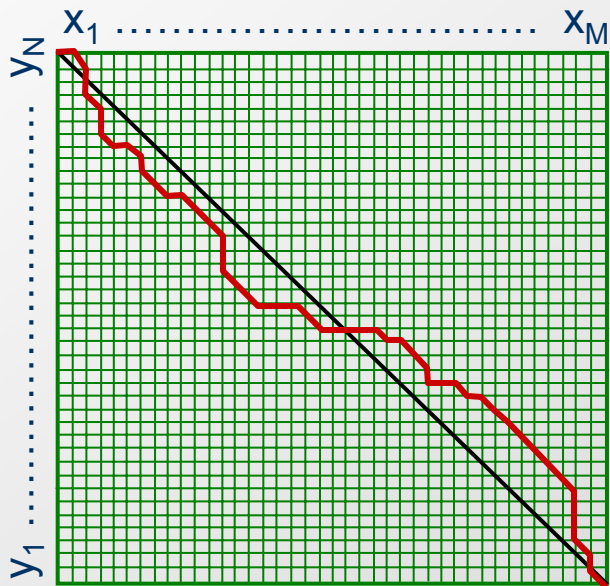
- $A(i-1, j) - 2$
- $A(i, j-1) - 2$
- $A(i-1, j-1) \pm 1$

Termination:

- Bottom right

What is missing?

- 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

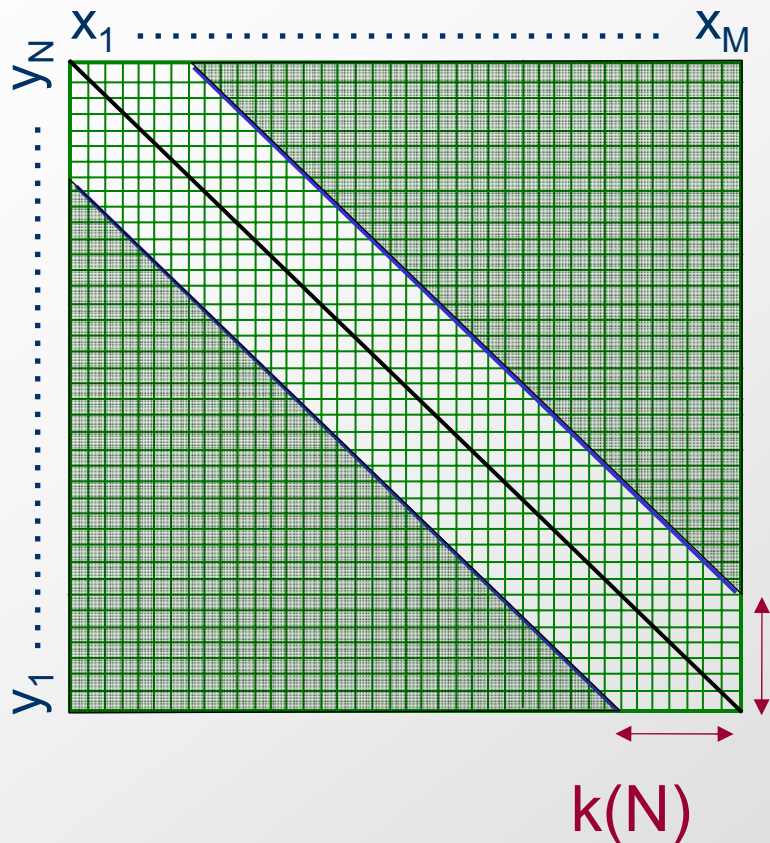


Time needed: $O(m*n)$

Space needed: $O(m*n)$

Can we do better than that?

Bounded Dynamic Programming



Initialization:

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

Iteration:

For $i = 1 \dots M$

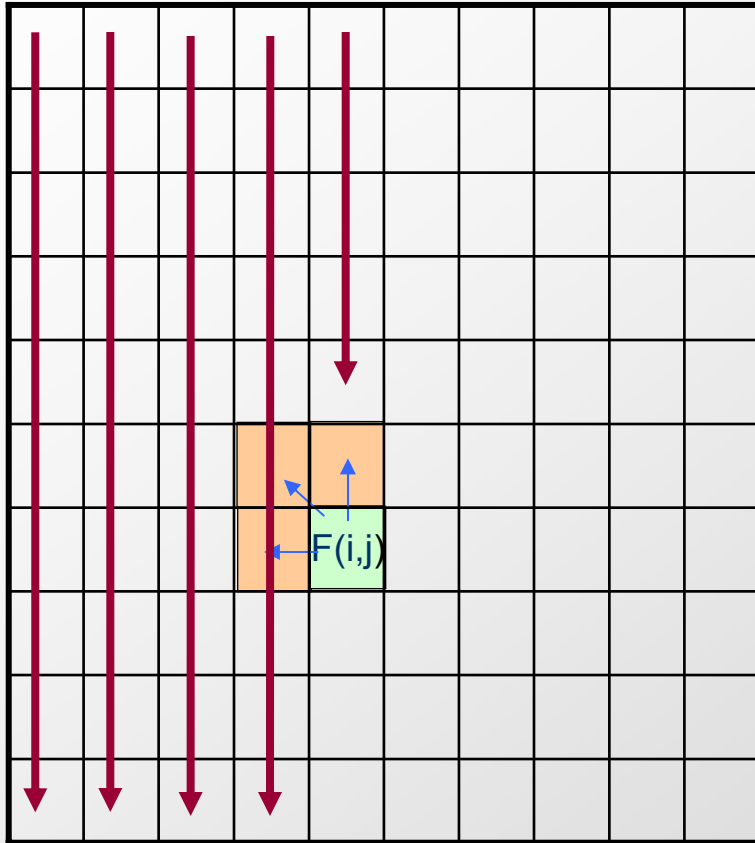
For $j = \max(1, i - k) \dots \min(N, i + k)$

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

Termination: same

Linear space alignment

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



Allocate (column[1])

Allocate (column[2])

For $i = 1 \dots M$

If $i > 1$, then:

Free(column[$i - 2$])

Allocate(column[i])

For $j = 1 \dots N$

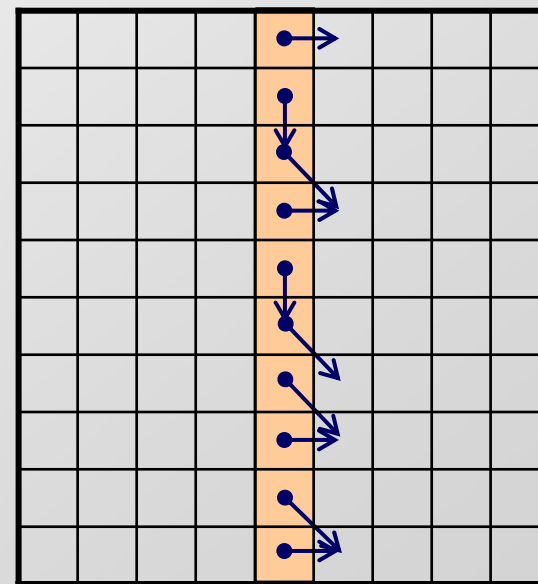
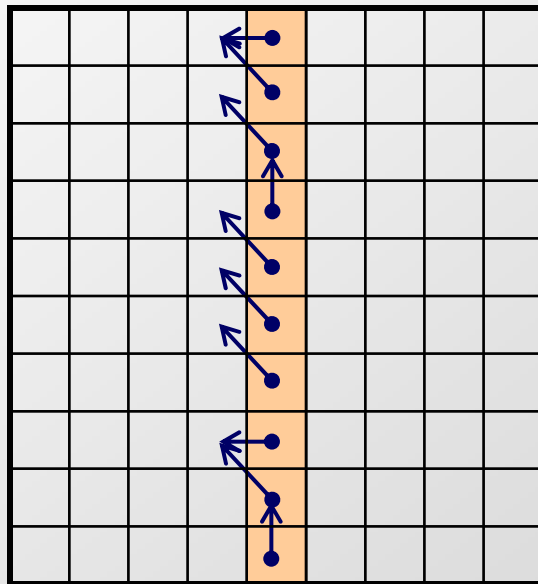
$F(i, j) = \dots$

What about the pointers?

Finding the best back-pointer for current column

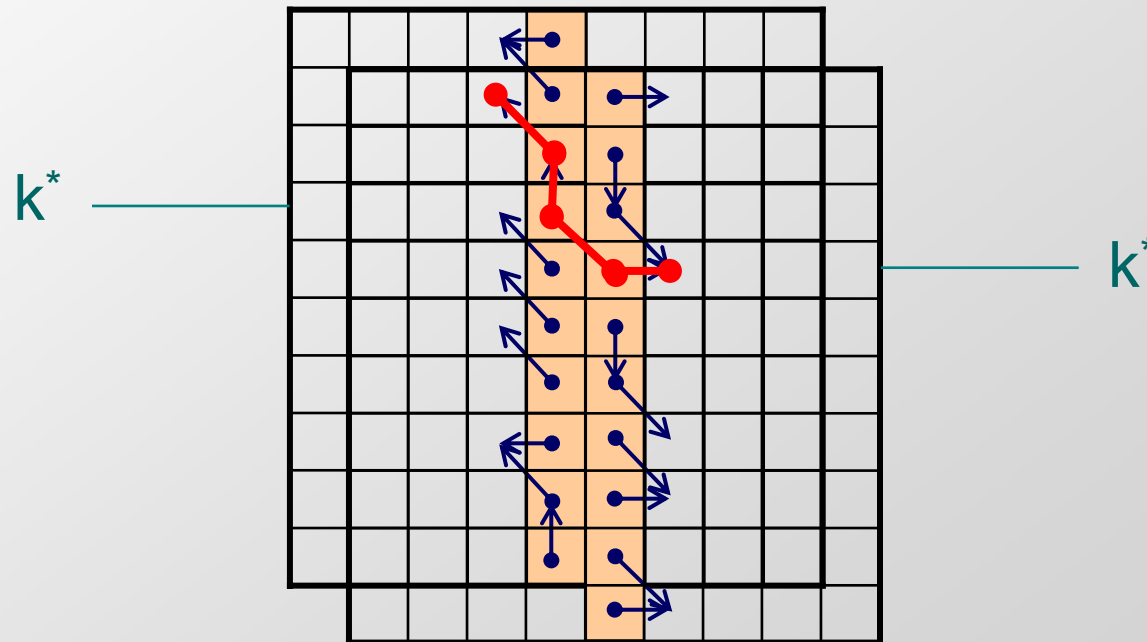
- Now, using 2 columns of space, we can compute for $k = 1 \dots M$, $F(M/2, k)$, $F^r(M/2, N-k)$

PLUS the backpointers



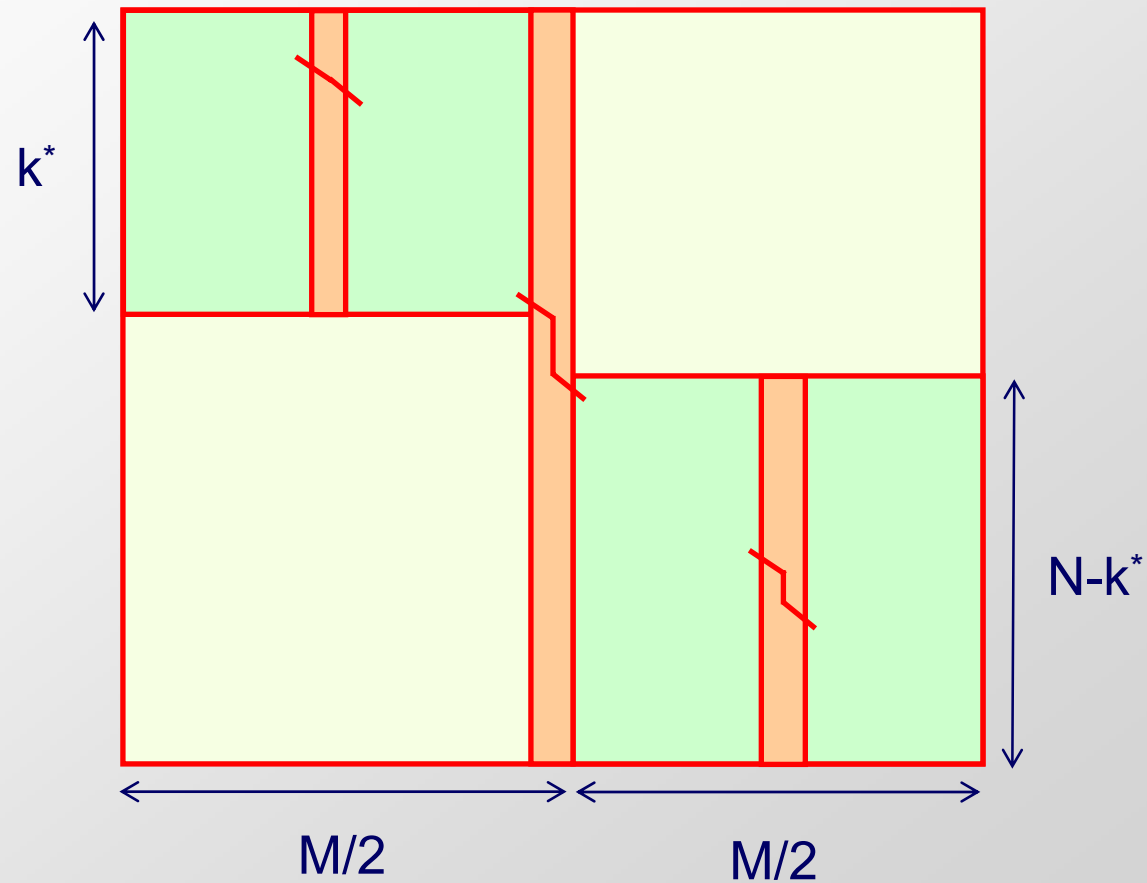
Best forward-pointer for current column

- 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^*

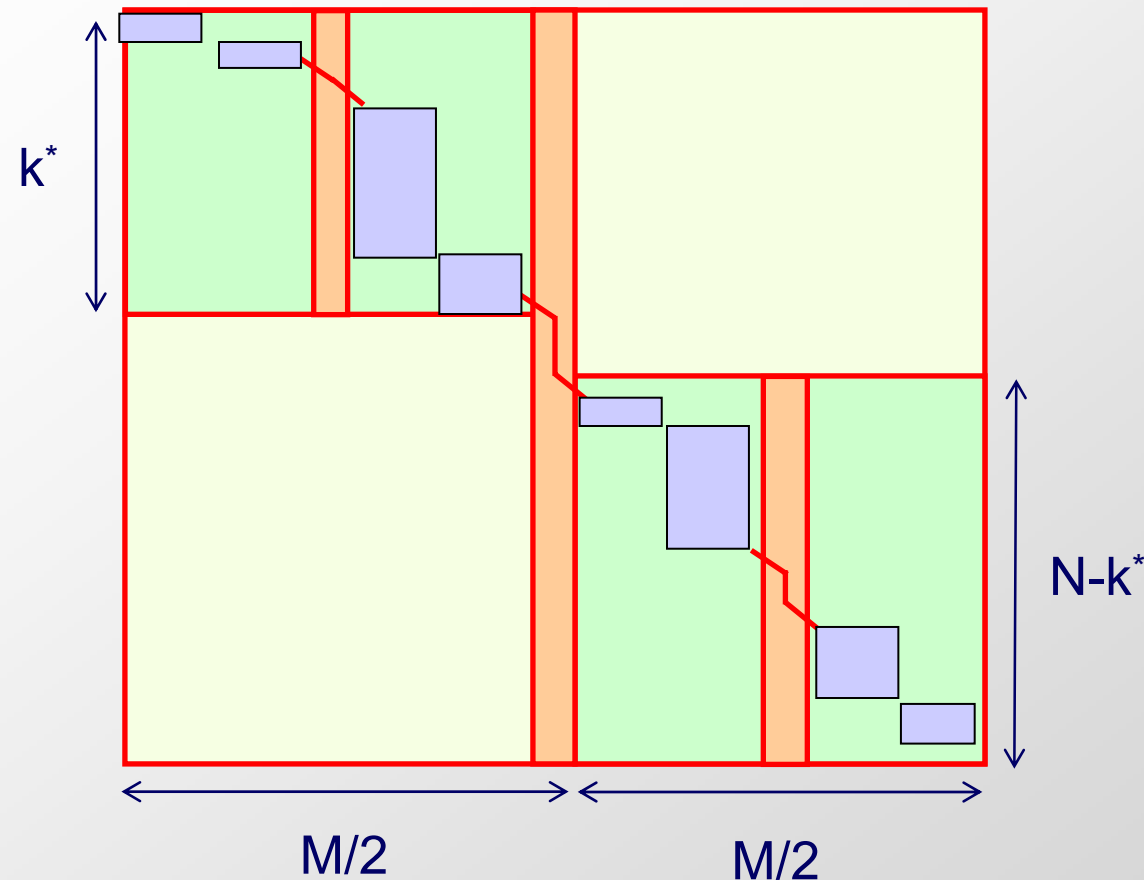


Recursively find midpoint for left & right

- Iterate this procedure to the left and right!



Total time cost of linear-space alignment



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

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

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 \Leftrightarrow Optimal alignment
- **Thursday: Variations on sequence alignment**
 - Local and global alignment
 - Affine gap penalties
 - Algorithmic speed-ups
- **Recitation:**
 - Dynamic programming applications
 - Probabilistic derivations of alignment scores