

COMP 182 Algorithmic Thinking

Dynamic Programming

*Luay Nakhleh
Computer Science
Rice University*

Strings vs. Sequences

- ❖ Both strings and sequences are ordered lists.
- ❖ Strings are finite, whereas sequences could be infinite.
- ❖ Strings are defined over some alphabet of letters (characters), whereas sequences can contain numbers, letters, etc.

Strings vs. Sequences

- ❖ u is a substring of v if there exists $x, y \in \Sigma^*$ such that $xuy = v$.
- ❖ u is a subsequence of v if u can be obtained by removing some letters from v .
- ❖ E.g., ACT is a substring of string ACTTT, but not a substring of ATCT.
- ❖ E.g., ACT is a subsequence of ATCT, but not a subsequence of TCA.

The Longest Increasing Subsequence Problem

The Problem

- ❖ Input: An array $A[0..n-1]$ of integers.
- ❖ Output: A longest sequence of indices $0 \leq i_1 < i_2 < \dots < i_k \leq n-1$ such that $A[i_j] < A[i_{j+1}]$ for every j .

The Problem

- ❖ Input: An array $A[0..n-1]$ of integers.
- ❖ Output: A longest sequence of indices $0 \leq i_1 < i_2 < \dots < i_k \leq n-1$ such that $A[i_j] < A[i_{j+1}]$ for every j .

What is the output on input $A=[3,6,9,7]$?

Solution

❖ The key question:

Is $A[0]$ part of an LIS (Longest Increasing Subsequence) of A ?

Solution

- ❖ The idea: The longest increasing subsequence (LIS) of $A[0..n-1]$ is either (whichever is longer):
 - ❖ Scenario 1: the LIS of $A[1..n-1]$
 - ❖ Scenario 2: $A[0]$ followed by the LIS of $A[1..n-1]$ with elements larger than $A[0]$, or

Solution

- ❖ The idea: The longest increasing subsequence (LIS) of $A[0..n-1]$ is either (whichever is longer):
 - ❖ Scenario 1: the LIS of $A[1..n-1]$
 - ❖ Scenario 2: $A[0]$ followed by the LIS of $A[1..n-1]$ with elements larger than $A[0]$, or



The recursive calls!

A Recursive Formulation

$$|LIS(A[0..n-1])| = \max\{|LIS(A[1..n-1])|, |[0] + LIS(A[1..n-1])|\}$$

A Recursive Algorithm

Algorithm 1: Rec-LIS

Input: Array $A[0..n - 1]$ of integers.

Output: A LIS of $A[0..n - 1]$.

```
return LIS-helper( $-\infty$ ,  $A[0..n - 1]$ , 0);
```

Algorithm 2: LIS-helper

Input: Integer k , array A of integers, and integer i .

Output: A LIS of $A[i \dots |A| - 1]$ where every element in the LIS is larger than k .

if $i \geq |A|$ **then**

```
return [];
```

else

```

1  |    $L \leftarrow \text{LIS-helper}(k, A, i + 1);$                                      // Scenario 1

```

```

if  $A[i] > k$  then

```

```

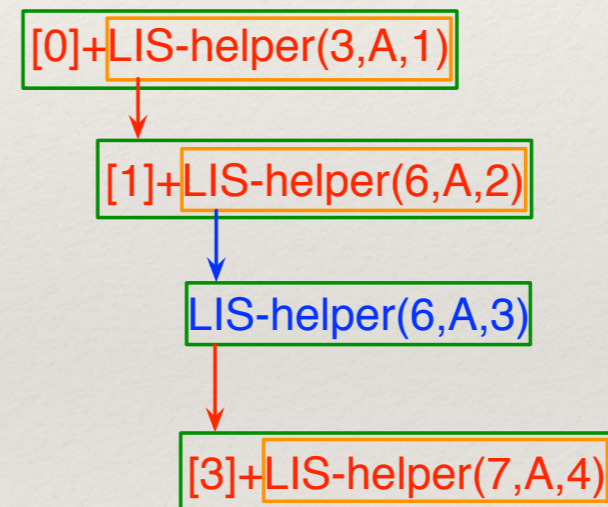
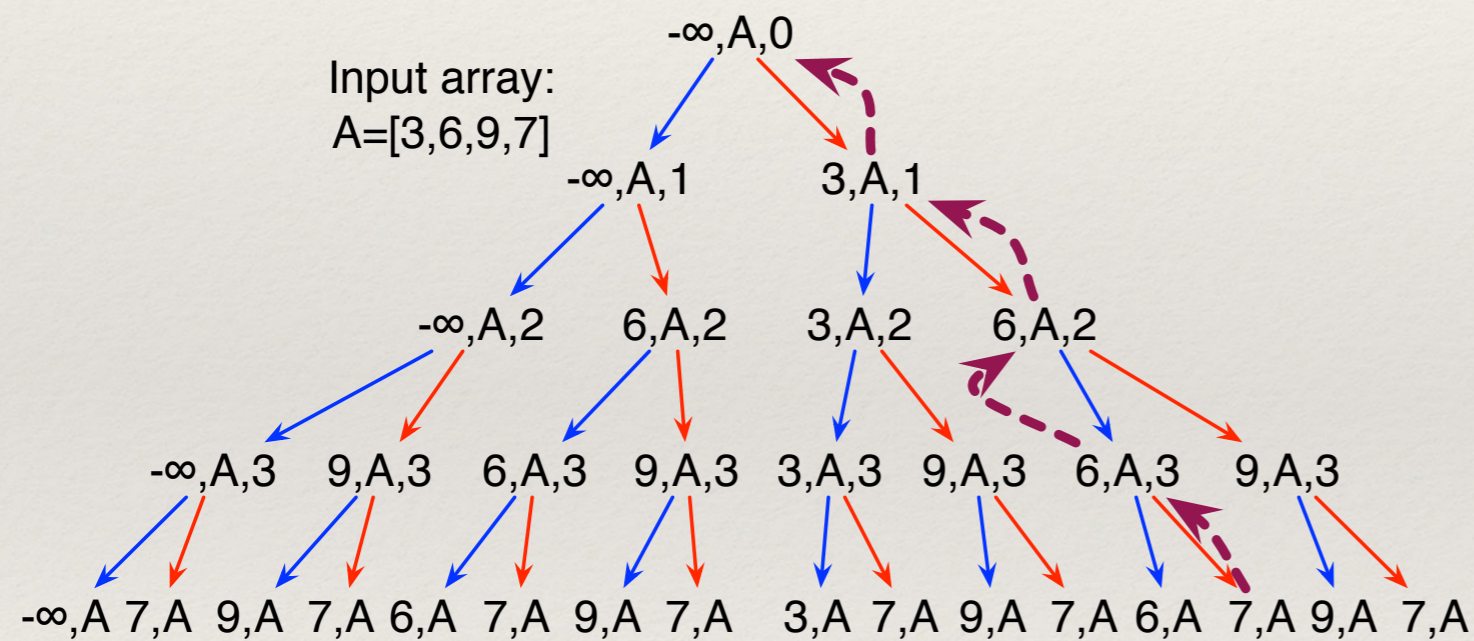
2       $L' \leftarrow [i] + \text{LIS-helper}(A[i], A, i + 1);$                                      // Scenario 2

```

if $|L'| > |L|$ **then**

$$L \leftarrow L';$$
return L ;

A Recursive Algorithm



Output:
 $[0, 1, 3]$

$= [0] + [1, 3] = [0, 1, 3]$

$= [1] + [3] = [1, 3]$

$= [3]$

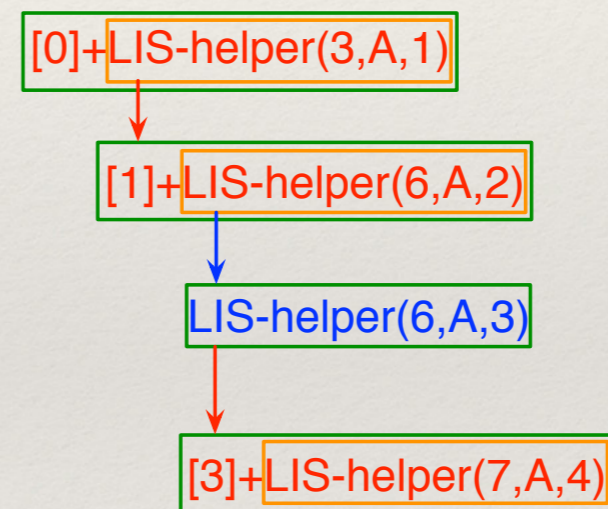
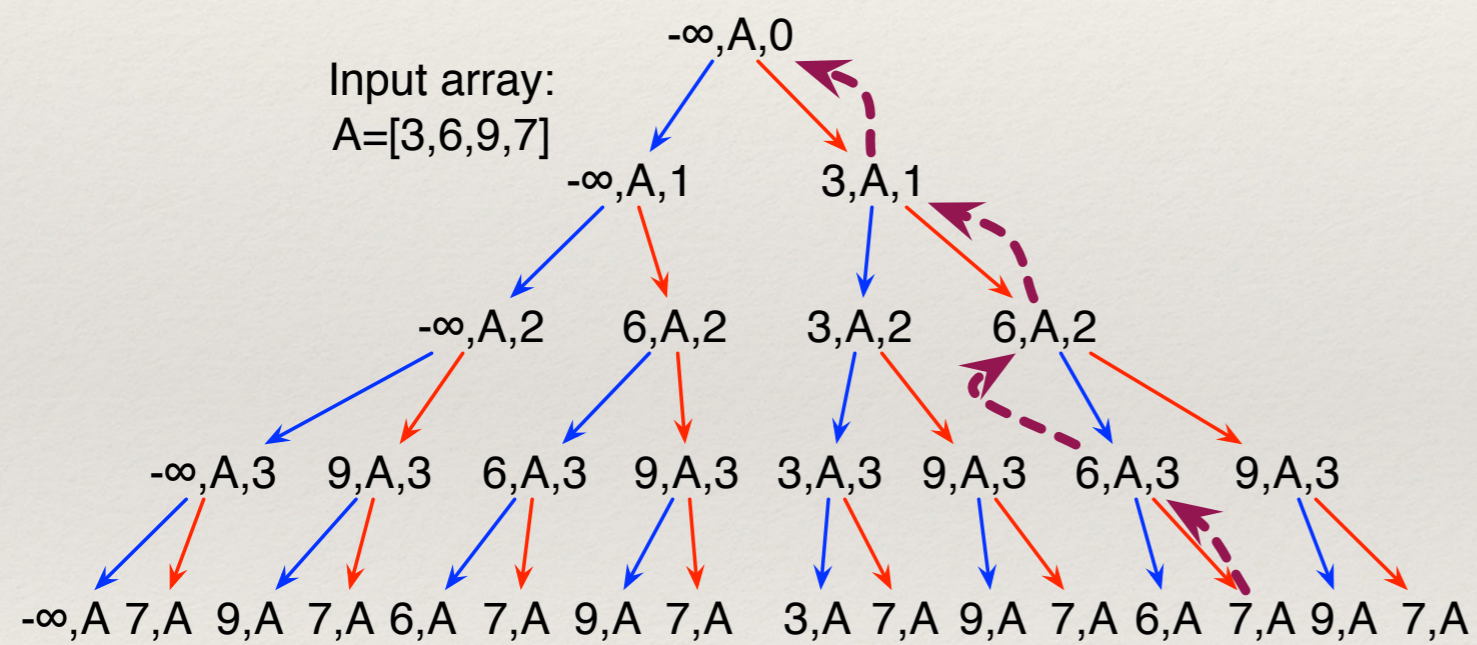
$= [3] + [] = [3]$

The Running Time

- ❖ $T(n) = 2T(n-1) + O(1)$
- ❖ Therefore, $T(n) = O(2^n)$

❖ Can we do better?

- ❖ Yes! The recursive implementation is wasting time by recomputing values that have been already computed over and over...



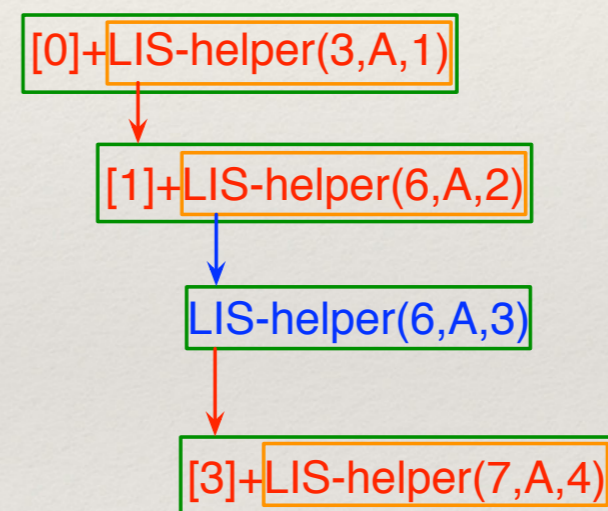
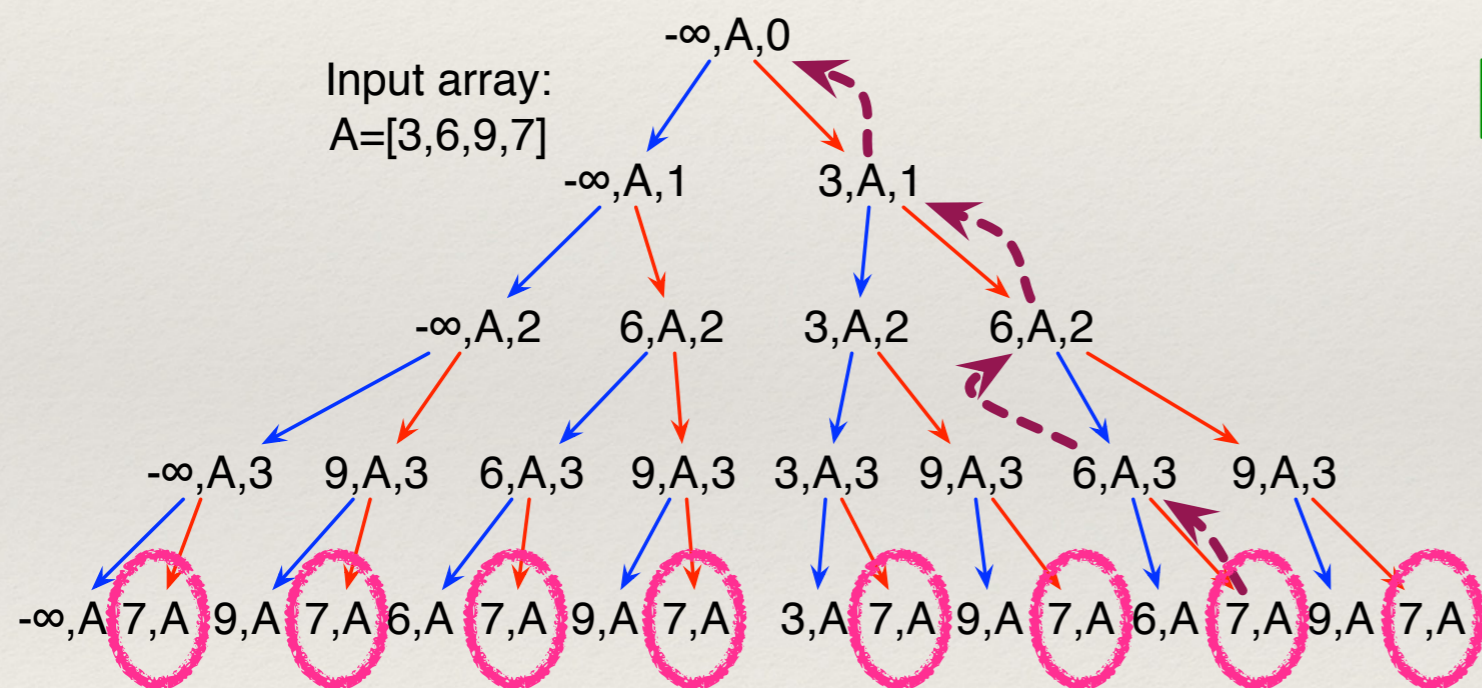
Output:
[0,1,3]

= [0] + [1,3] = [0,1,3]

= [1] + [3] = [1,3]

= [3]

= [3] + [] = [3]



Output:
[0,1,3]

= [0] + [1,3] = [0,1,3]

= [1] + [3] = [1,3]

= [3]

= [3] + [] = [3]

- ❖ If instead of calling the function recursively: we start from smaller subproblems, compute the solutions on them, store those solutions, and then make use of those solutions to construct solutions to larger subproblems...

- ❖ Since we call LIS-helper the first time with $k=-\infty$, let us assume that the array A in fact has $n+1$ elements, where $A[0]=-\infty$ (for convenience; that is, we are not restricting the problem).

- ❖ Define a matrix LIS-M, where entry LIS-M[i,j] holds the length of the LIS of $A[j\dots n]$ with all elements (of the LIS) larger than $A[i]$.
- ❖ If we can compute the entries of LIS-M, then the answer to our problem will be LIS-M[0,1].
- ❖ The question is: How do we compute LIS-M[i,j]?

$$LISM[i, j] \leftarrow \begin{cases} 0 & \text{if } j > n \\ LISM[i, j + 1] & \text{if } A[i] \geq A[j] \\ \max\{LISM[i, j + 1], 1 + LISM[j, j + 1]\} & \text{otherwise} \end{cases}$$

The Algorithm

Algorithm 3: DP-LIS

Input: Array $A[0..n]$ of integers, where $A[0] = -\infty$.

Output: Length of LIS of $A[1..n]$.

for $i \leftarrow 0$ **to** n **do**

$LISM[i, n+1] \leftarrow 0$;

for $j \leftarrow n$ **downto** 1 **do**

for $i \leftarrow 0$ **to** $j-1$ **do**

if $A[i] \geq A[j]$ **then**

$LISM[i, j] \leftarrow LISM[i, j+1]$;

else

$LISM[i, j] \leftarrow \max\{LISM[i, j+1], 1 + LISM[j, j+1]\}$;

return $LISM[0, 1]$;

The Algorithm

Algorithm 3: DP-LIS

Input: Array $A[0..n]$ of integers, where $A[0] = -\infty$.

Output: Length of LIS of $A[1..n]$.

```
for  $i \leftarrow 0$  to  $n$  do
   $LISM[i, n + 1] \leftarrow 0$ ;
for  $j \leftarrow n$  downto  $1$  do
  for  $i \leftarrow 0$  to  $j - 1$  do
    if  $A[i] \geq A[j]$  then
       $LISM[i, j] \leftarrow LISM[i, j + 1]$ ;
    else
       $LISM[i, j] \leftarrow \max\{LISM[i, j + 1], 1 + LISM[j, j + 1]\}$ ;
return  $LISM[0, 1]$ ;
```

What is the running time of this algorithm?

The Algorithm

Algorithm 3: DP-LIS

Input: Array $A[0..n]$ of integers, where $A[0] = -\infty$.

Output: Length of LIS of $A[1..n]$.

```
for  $i \leftarrow 0$  to  $n$  do
   $LISM[i, n+1] \leftarrow 0$ ;
for  $j \leftarrow n$  downto 1 do
  for  $i \leftarrow 0$  to  $j-1$  do
    if  $A[i] \geq A[j]$  then
       $LISM[i, j] \leftarrow LISM[i, j+1]$ ;
    else
       $LISM[i, j] \leftarrow \max\{LISM[i, j+1], 1 + LISM[j, j+1]\}$ ;
return  $LISM[0, 1]$ ;
```

What is the running time of this algorithm?

Answer: $O(n^2)$.

Algorithm 3: DP-LIS

Input: Array $A[0..n]$ of integers, where $A[0] = -\infty$.

Output: Length of LIS of $A[1..n]$.

```
for  $i \leftarrow 0$  to  $n$  do  
   $LISM[i, n + 1] \leftarrow 0$ ;  
for  $j \leftarrow n$  downto  $1$  do  
  for  $i \leftarrow 0$  to  $j - 1$  do  
    if  $A[i] \geq A[j]$  then  
       $LISM[i, j] \leftarrow LISM[i, j + 1]$ ;  
    else  
       $LISM[i, j] \leftarrow \max\{LISM[i, j + 1], 1 + LISM[j, j + 1]\}$ ;  
return  $LISM[0, 1]$ ;
```

$A=[3,6,9,7]$

	0	1	2	3	4	5
0		3	2	1	1	0
1			2	1	1	0
2				1	1	0
3					0	0
4						0

- ❖ Question: How do we get the actual LIS (rather than just its length)?
- ❖ Answer: Use traceback.

Algorithm 3: DP-LIS

Input: Array $A[0..n]$ of integers, where $A[0] = -\infty$.

Output: Length of LIS of $A[1..n]$.

```
for  $i \leftarrow 0$  to  $n$  do
   $LISM[i, n + 1] \leftarrow 0$ ;
for  $j \leftarrow n$  downto  $1$  do
  for  $i \leftarrow 0$  to  $j - 1$  do
    if  $A[i] \geq A[j]$  then
       $LISM[i, j] \leftarrow LISM[i, j + 1]$ ;
    else
       $LISM[i, j] \leftarrow \max\{LISM[i, j + 1], 1 + LISM[j, j + 1]\}$ ;
return  $LISM[0, 1]$ ;
```

Algorithm 4: ComputeLIS

Input: Array $A[0..n]$ of integers, where $A[0] = -\infty$, and matrix $LISM$.

Output: An LIS lis of $A[1..n]$.

```
 $lis \leftarrow []$ ;
 $k \leftarrow 0$ ;
 $i \leftarrow 0$ ;
for  $j \leftarrow 1$  to  $n$  do
  if  $LISM[i, j] = 1 + LISM[j, j + 1]$  then
     $lis[k] = j$ ;
     $i \leftarrow j$ ;
     $k \leftarrow k + 1$ ;
return  $lis$ ;
```

- ❖ Why did I call the algorithm DP-LIS? That is, what is DP?

Dynamic Programming

- ❖ The algorithm that we have just seen for solving the LIS Problem uses the **Dynamic Programming** (DP) algorithmic technique.
- ❖ Dynamic programming is a technique for solving problems with overlapping subproblems.
- ❖ Typically, these subproblems arise from a recurrence relating a solution to a given problem with solutions to its smaller subproblems of the same type.
- ❖ Rather than solving overlapping subproblems again and again, dynamic programming suggests solving each of the smaller subproblems only once and recording the results in a table from which we can then obtain a solution to the original problem.

Dynamic Programming

- ❖ To set about developing an algorithm based on dynamic programming, one needs a collection of subproblems derived from the original problem that satisfies a few basic properties:
 - ❖ The solution to the original problem can be easily computed from the solutions to the subproblems (for example, the original problem may actually be one of the subproblems).
 - ❖ There is a natural ordering on subproblems from “smallest” to “largest,” together with an easy-to-compute recurrence that allows one to determine the solution to a subproblem from the solutions to some number of smaller subproblems.

Illustrating the Properties of DP Algorithms: The LIS Problem

$$LISM[i, j] \leftarrow \begin{cases} 0 & \text{if } j > n \\ LISM[i, j + 1] & \text{if } A[i] \geq A[j] \\ \max\{LISM[i, j + 1], 1 + LISM[j, j + 1]\} & \text{otherwise} \end{cases}$$

- ❖ Solution from solutions to subproblems?
- ❖ Natural ordering of subproblems?

Polynomial DP Algorithms

- ❖ For the DP algorithm to be polynomial, the number of subproblems that need to be solved must be polynomial.

Pairwise Sequence Alignment

Life through Evolution

- ❖ All living organisms are related to each other through evolution.
- ❖ This means: any pair of organisms, no matter how different, have a common ancestor sometime in the past, from which they evolved.
- ❖ Evolution involves
 - ❖ inheritance: passing of characteristics from parent to offspring
 - ❖ variation: differentiation between parent and offspring
 - ❖ (and other processes, such as selection,...)

Sequence Variations Due to Mutations

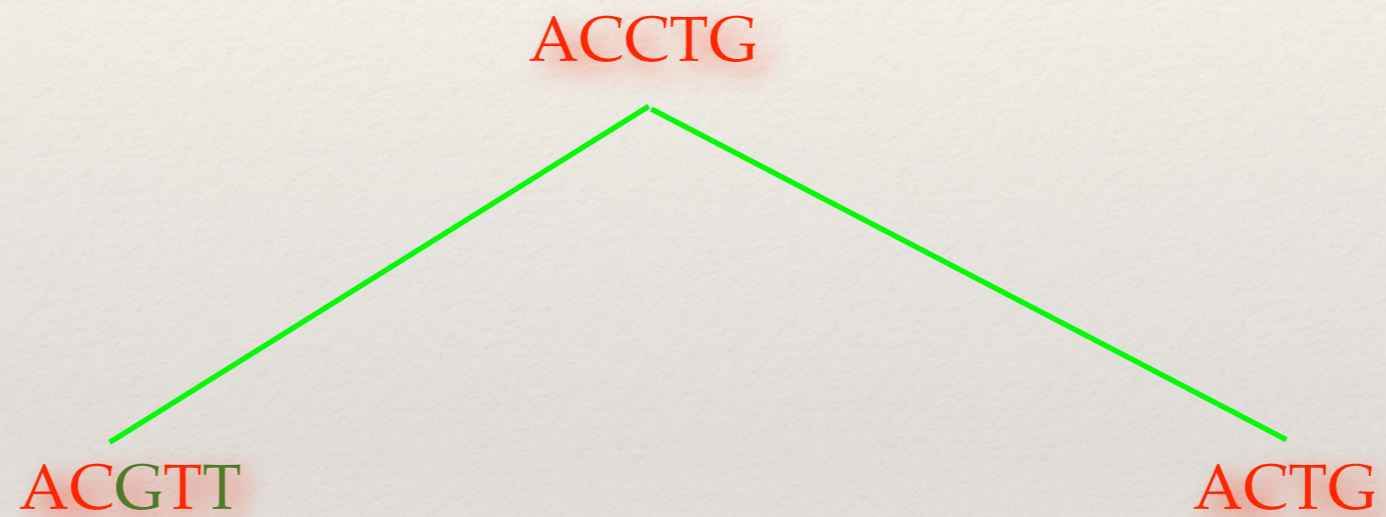
- ❖ Mutations and selection over millions of years can result in considerable divergence between present-day sequences derived from the same ancestral sequence.
- ❖ The base pair composition of the sequences can change due to point mutation (substitutions), and the sequence lengths can vary due to insertions / deletions.

Sequence Evolution

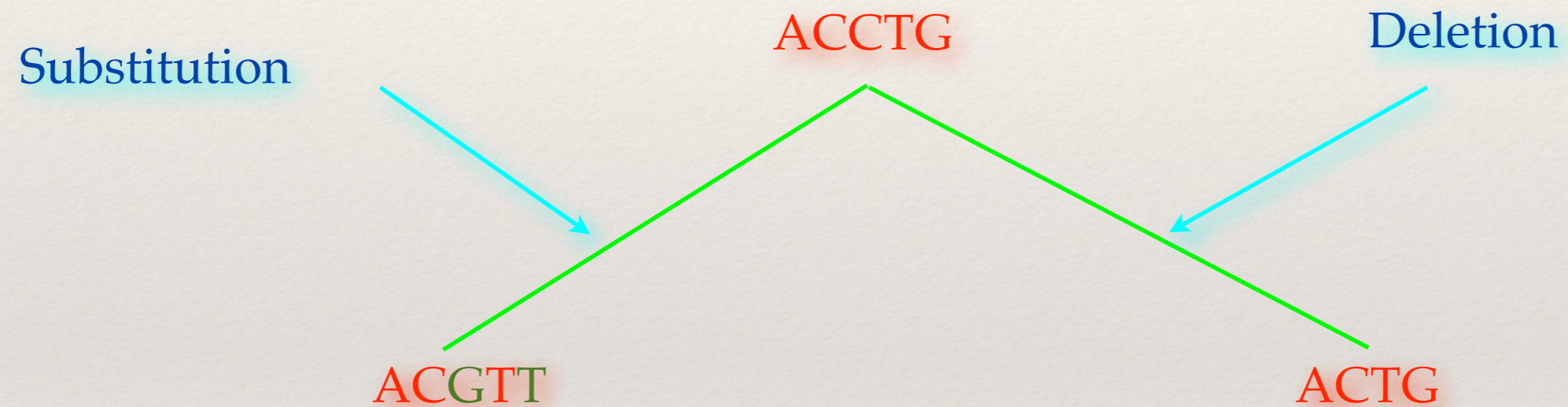
Sequence Evolution

ACCTG

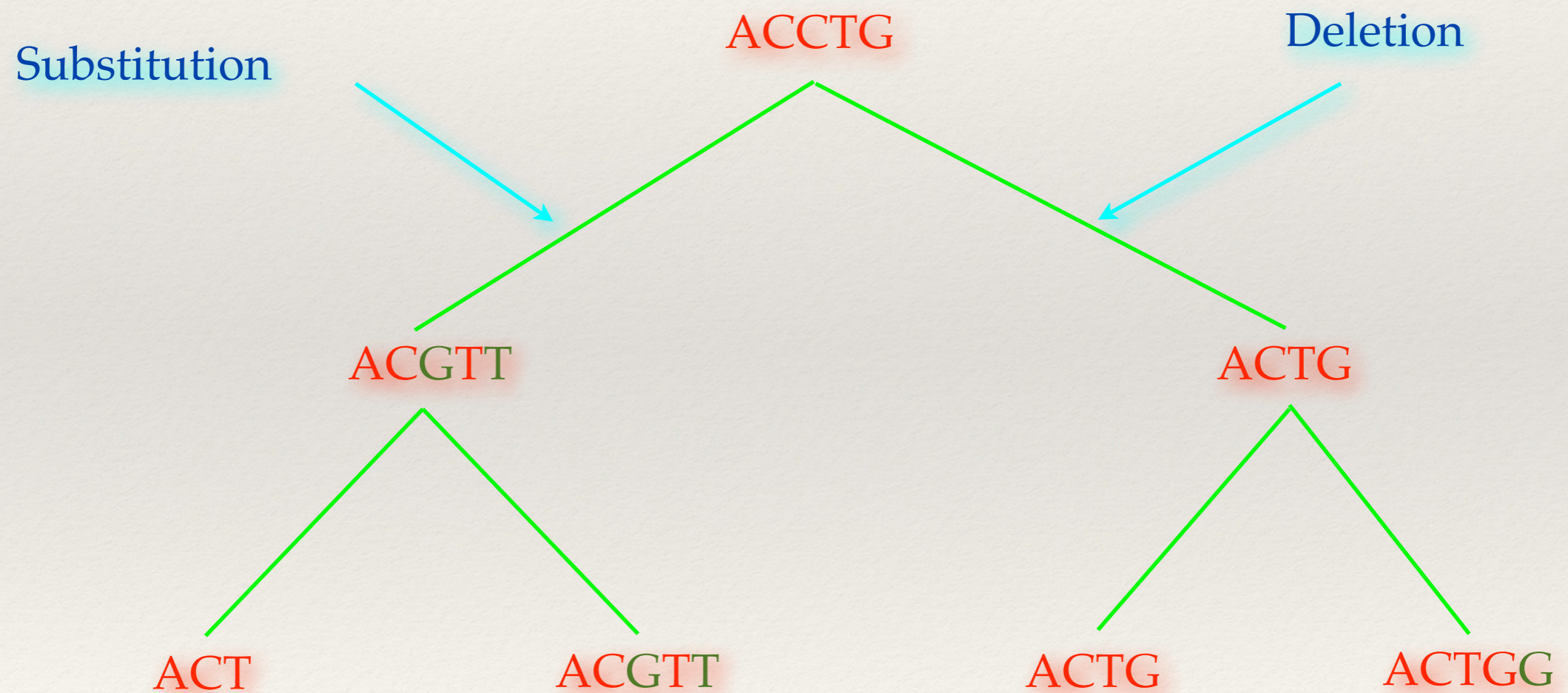
Sequence Evolution



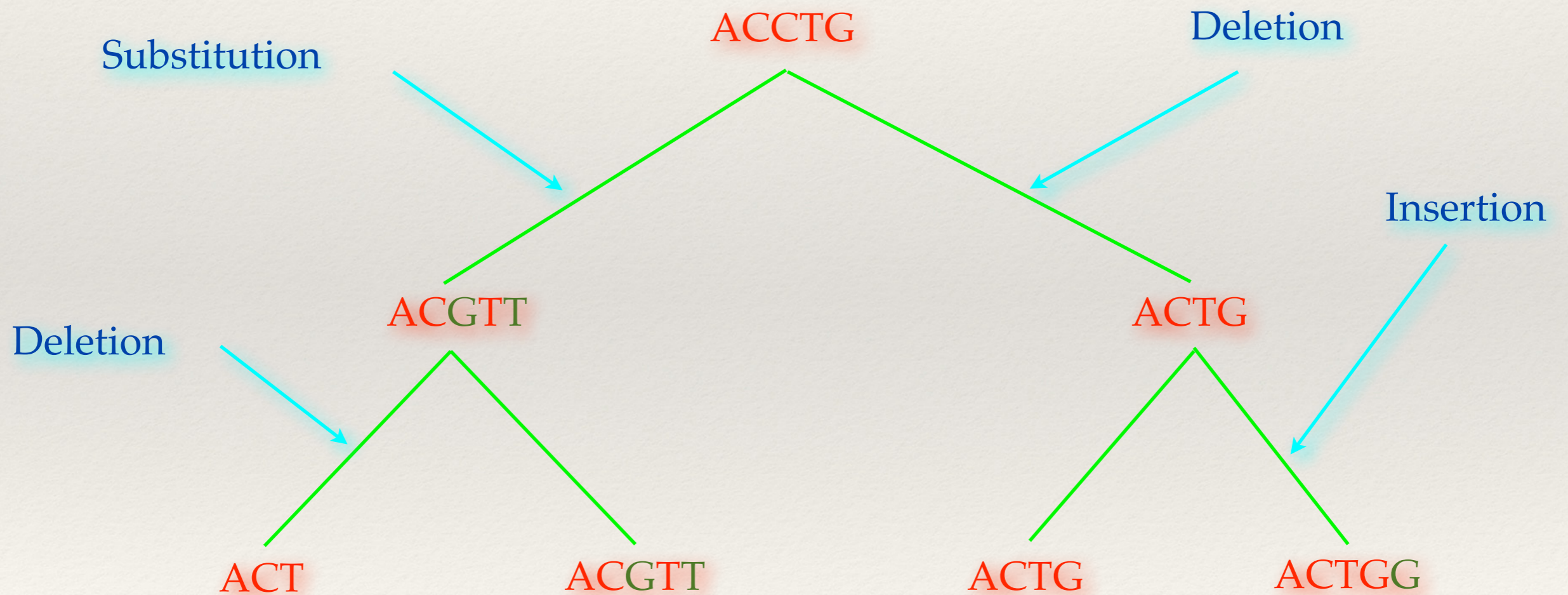
Sequence Evolution



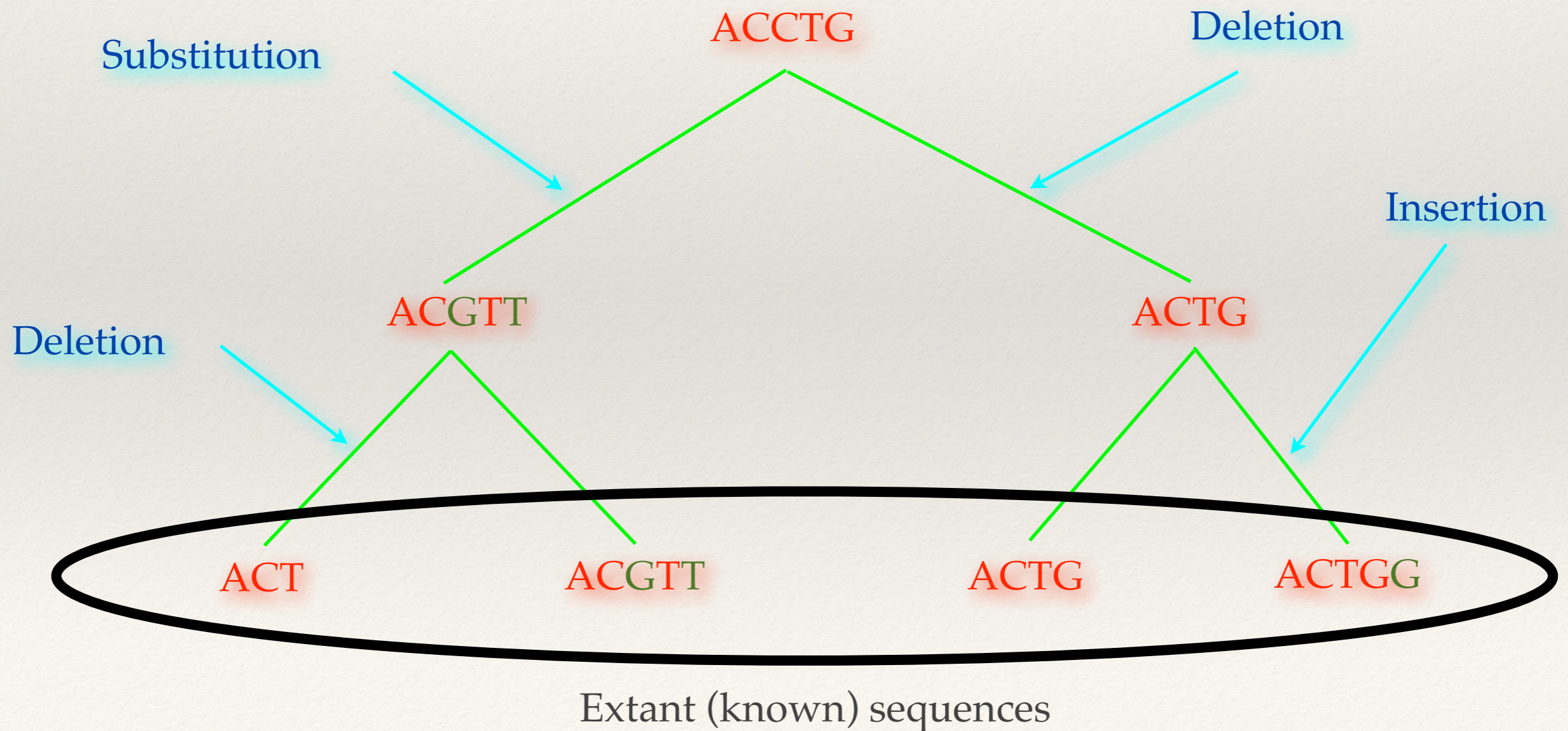
Sequence Evolution



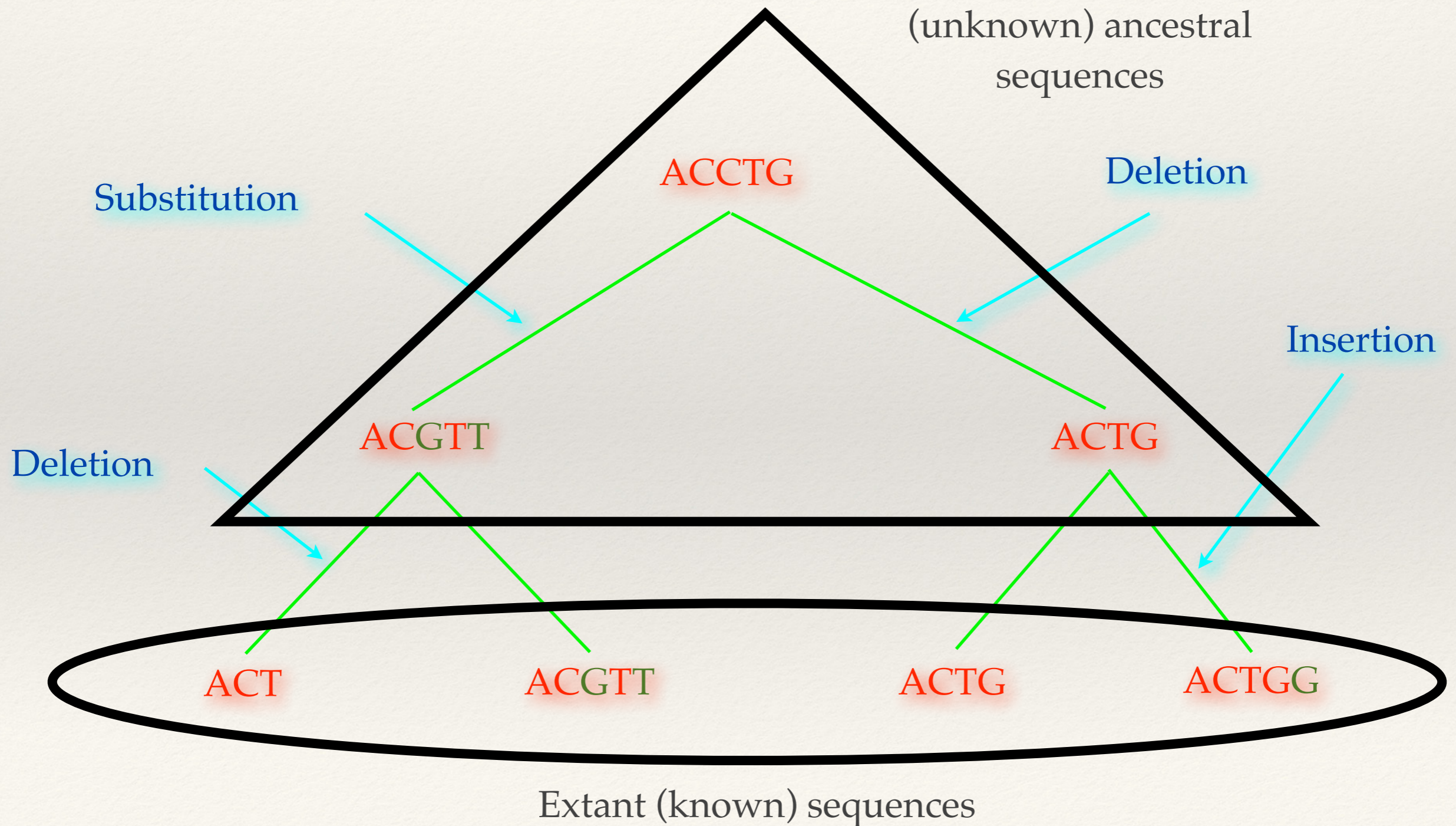
Sequence Evolution



Sequence Evolution



Sequence Evolution



Sequence Evolution

- ❖ In biology, we have access to the extant (known) sequences, but in most cases no knowledge of the ancestral sequences.
- ❖ Therefore, a central task in biology is to identify similarities and differences between extant sequences in an attempt to map the evolutionary past.
- ❖ Using the example of the previous slide, we are interested in finding the similarities, for example, between the two sequences ACT and ACGTT.
- ❖ As sequences change in length and content throughout evolution, we are often interested in regions of high similarities between the two sequences.
- ❖ We can be “very strict” (similarity=identity) or “less strict” (similarity includes matches, mismatches, and gaps).

The Longest Common Subsequence (LCS) Problem

- ❖ In the case of the LCS problem, we seek the longest sequence that is a subsequence of two input sequences X and Y .
- ❖ For example, if $X=ACT$ and $Y=ACGTT$,
 - ❖ AC is a subsequence of X as well as of Y .
 - ❖ CT is a subsequence of X as well as of Y .
 - ❖ However, ACT is the longest sequence that is a subsequence of X and at the same time a subsequence of Y .

The Longest Common Subsequence (LCS) Problem

- ❖ In the case of the LCS problem, we seek the longest sequence that is a subsequence of two input sequences X and Y .
- ❖ For example, if $X=ACT$ and $Y=ACGTT$,
 - ❖ AC is a subsequence of X as well as of Y .
 - ❖ CT is a subsequence of X as well as of Y .
 - ❖ However, ACT is the longest sequence that is a subsequence of X and at the same time a subsequence of Y .

$X=ACT$

$X=ACGTT$

The Longest Common Subsequence (LCS) Problem

- ❖ Give a brute-force algorithm for solving the LCS Problem.
- ❖ Do you think the algorithm is efficient?
- ❖ Now, reason about the problem “recursively”: Let $X=X'a$ and $Y=Y'b$, where a and b are single letters, and X' and Y' are strings (in other words, X ends with the letter a , and Y ends with the letter b).
- ❖ There are two cases:
 - ❖ $a=b$: Are they part of an LCS solution? If not, how do we proceed?
 - ❖ $a \neq b$: Are they part of an LCS solution? If not, how do we proceed?

The Longest Common Subsequence (LCS) Problem

- ❖ The recursive reasoning naturally gives rise to an algorithm for solving the LCS problem efficiently, by making use of solutions to sub-problems.
- ❖ While computationally the problem is “taken care of,” biologically it is expected that the more divergent the two sequences X and Y are, the shorter the subsequences that are common to both become.
- ❖ Therefore, in most cases, it is necessary that we relax the “identity constraint,” and instead seek similarities across the two sequences that include matches (letters that are identical in both sequences), mismatches (letters that are not identical in both, but are accepted as pairs), and gaps (letters that are present in one sequence but missing from the other).
- ❖ This is known as the **sequence alignment problem**.

Relaxing the Identity Constraint:

Sequence Alignment (matches, mismatches, and gaps)

T	H	A	T	S	E	Q	U	E	N	C	E
T	H	I	S	S	E	Q	U	E	N	C	E

Relaxing the Identity Constraint:

Sequence Alignment (matches, mismatches, and gaps)

T H A T S E Q U E N C E

T H I S S E Q U E N C E

LCS Solution → THSEQUENCE

Relaxing the Identity Constraint:

Sequence Alignment (matches, mismatches, and gaps)

T H A T S E Q U E N C E

T H I S S E Q U E N C E

LCS Solution → THSEQUENCE

T	H	A	T	S	E	Q	U	E	N	C	E
T	H	I	S	S	E	Q	U	E	N	C	E

Relaxing the Identity Constraint:

Sequence Alignment (matches, mismatches, and gaps)

T H A T S E Q U E N C E

T H I S S E Q U E N C E

LCS Solution → THSEQUENCE

T	H	A	T	S	E	Q	U	E	N	C	E
T	H	I	S	S	E	Q	U	E	N	C	E

Alignment with
Mismatches



Relaxing the Identity Constraint:

Sequence Alignment (matches, mismatches, and gaps)

T H A T S E Q U E N C E

T H I S S E Q U E N C E

LCS Solution → THSEQUENCE

T H A T S E Q U E N C E

T H I S S E Q U E N C E

Alignment with
Mismatches

T	H	I	S	I	S	A	—	S	E	Q	U	E	N	C	E
T	H	—	—	—	—	A	T	S	E	Q	U	E	N	C	E

Relaxing the Identity Constraint:

Sequence Alignment (matches, mismatches, and gaps)

T H A T S E Q U E N C E

T H I S S E Q U E N C E

LCS Solution → THSEQUENCE

T H A T S E Q U E N C E

T H I S S E Q U E N C E

Alignment with
Mismatches

T H I S I S A - S E Q U E N C E

T H - - - - A T S E Q U E N C E

Alignment with Gaps (indels: insertions/
deletions)

Sequence Alignment

- ❖ As you can imagine, since we don't enforce identity, any way of "aligning" the two sequences X and Y so that their lengths are equal is a "candidate" for sequence alignment (just pad them with dashes so that their lengths are equal; don't align dash with dash, though).
- ❖ So, how do we choose the "best" alignment?
- ❖ We define a scoring matrix that gives a score to every pair of aligned letters, and a penalty to every column with a dash.
- ❖ The score of an alignment is then the sum of the scores and penalties assigned to each column in the alignment.
- ❖ The "best" alignment is one with the highest score.

Sequence Alignment

- ❖ Here's an example of a scoring matrix M , where $M_{pq}=i$ indicates that if p and q are aligned with each other in the alignment, they contribute score i to the overall score of the alignment (and penalty i if either p or q is a dash).
- ❖ Consider the alphabet $\{A,C,T,G\}$.
- ❖ Here's an example of a scoring matrix M .

	A	C	T	G	-
A	10	5	7	3	-6
C	5	10	6	5	-5
T	2	1	15	1	-3
G	8	4	2	15	-1
-	-4	-4	-2	-2	N/A

Sequence Alignment

X=ACC
Y=AGC

	A	C	T	G	-
A	10	5	7	3	-6
C	5	10	6	5	-5
T	5	1	15	1	-3
G	8	4	2	15	-1
-	-4	-4	-2	-2	N/A

Sequence Alignment

X=ACC

Y=AGC

	A	C	T	G	-
A	10	5	7	3	-6
C	5	10	6	5	-5
T	5	1	15	1	-3
G	8	4	2	15	-1
-	-4	-4	-2	-2	N/A

Alignment 1

A	C	C	-	-
-	-	A	G	C

Alignment 2

A	C	-	C
A	-	G	C

Alignment 3

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

Alignment 4

A	C	C
A	G	C

X'=

Y'=

Sequence Alignment

X=ACC
Y=AGC

	A	C	T	G	-
A	10	5	7	3	-6
C	5	10	6	5	-5
T	5	1	15	1	-3
G	8	4	2	15	-1
-	-4	-4	-2	-2	N/A

Alignment 1

A	C	C	-	-
-	-	A	G	C
-6	-5	5	-2	-4

Alignment 2

A	C	-	C
A	-	G	C
10	-5	-2	10

Alignment 3

A	C	C	-	-	-
-	-	-	A	G	C
-6	-5	-5	-4	-2	-4

Alignment 4

A	C	C
A	G	C
10	5	10

X'=

Y'=

column
scores

Sequence Alignment

X=ACC
Y=AGC

	A	C	T	G	-
A	10	5	7	3	-6
C	5	10	6	5	-5
T	5	1	15	1	-3
G	8	4	2	15	-1
-	-4	-4	-2	-2	N/A

Alignment 1

X'	A	C	C	-	-
Y'	-	-	A	G	C
column scores	-6	-5	5	-2	-4

-12

Alignment 2

X'	A	C	-	C
Y'	A	-	G	C
column scores	10	-5	-2	10

13

Alignment 3

X'	A	C	C	-	-	-
Y'	-	-	-	A	G	C
column scores	-6	-5	-5	-4	-2	-4

-26

Alignment 4

X'	A	C	C
Y'	A	G	C
column scores	10	5	10

25

alignment
score

Sequence Alignment

X=ACC
Y=AGC

	A	C	T	G	-
A	10	5	7	3	-6
C	5	10	6	5	-5
T	5	1	15	1	-3
G	8	4	2	15	-1
-	-4	-4	-2	-2	N/A

Alignment 1

A	C	C	-	-
-	-	A	G	C
-6	-5	5	-2	-4

-12

Alignment 2

A	C	-	C
A	-	G	C
10	-5	-2	10

13

Alignment 3

A	C	C	-	-	-
-	-	-	A	G	C
-6	-5	-5	-4	-2	-4

-26

Alignment 4

A	C	C
A	G	C
10	5	10

25

X'=

Y'=-

column
scores

alignment
score

The best among these four alignments

Sequence Alignment

X=ACC
Y=AGC

	A	C	T	G	-
A	10	5	7	3	-6
C	5	10	6	5	-5
T	5	1	15	1	-3
G	8	4	2	15	-1
-	-4	-4	-2	-2	N/A

These are just 4 alignments.. there are many more possible ones!

Alignment 1

A	C	C	-	-
-	-	A	G	C
-6	-5	5	-2	-4

-12

Alignment 2

A	C	-	C
A	-	G	C
10	-5	-2	10

13

Alignment 3

A	C	C	-	-	-
-	-	-	A	G	C
-6	-5	-5	-4	-2	-4

-26

Alignment 4

A	C	C
A	G	C
10	5	10

25

X'=

Y'=

column
scores

alignment
score

The best among these four alignments

Sequence Alignment

- ❖ Is a brute-force algorithm feasible for this problem?
- ❖ Can we come up with a better algorithm that is feasible for practical cases?

Sequence Alignment

- ❖ Is a brute-force algorithm feasible for this problem?
- ❖ Can we come up with a better algorithm that is feasible for practical cases?

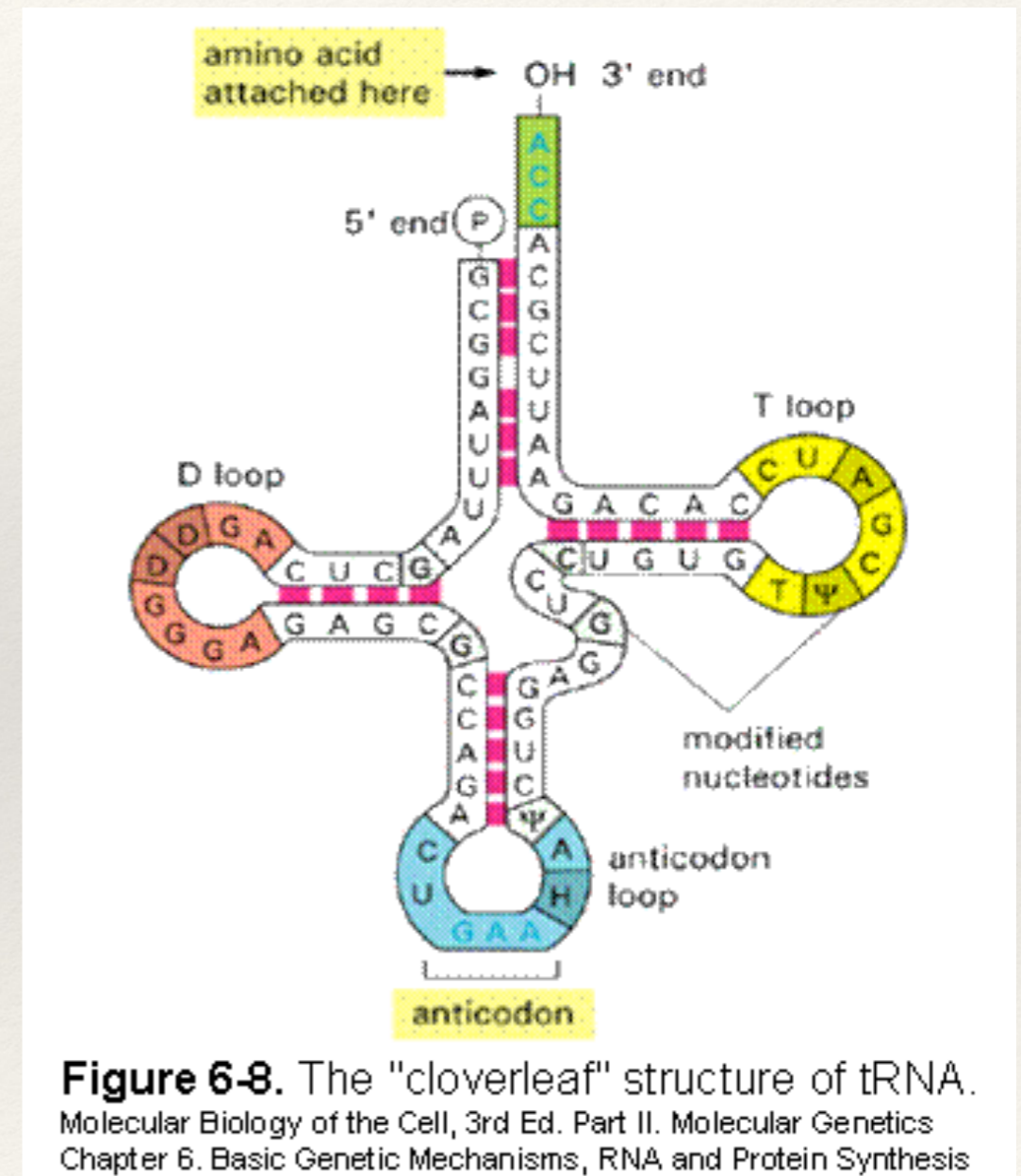
Answer: Enjoy Homework 4!

RNA Secondary Structure Prediction

- ❖ Step 1: Understanding the problem
 - ❖ What is RNA?
 - ❖ What is secondary structure?
 - ❖ What is prediction in this case?
 - ❖ ...

RNA Secondary Structure

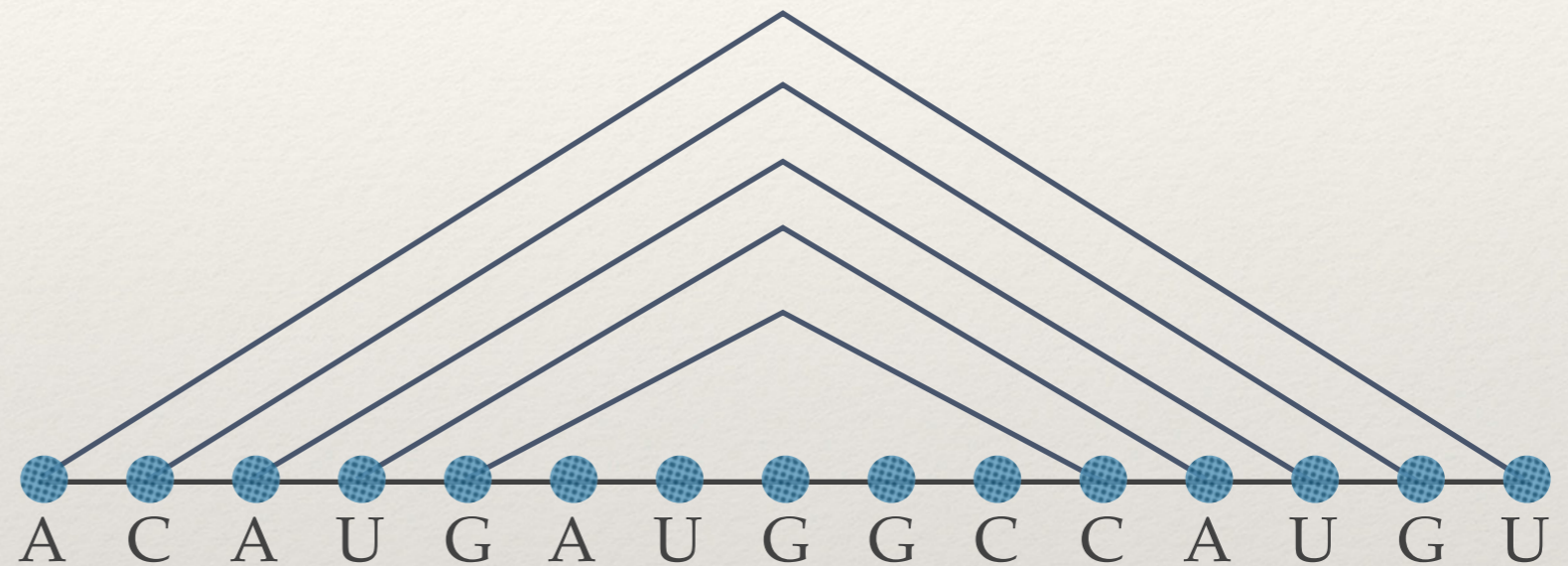
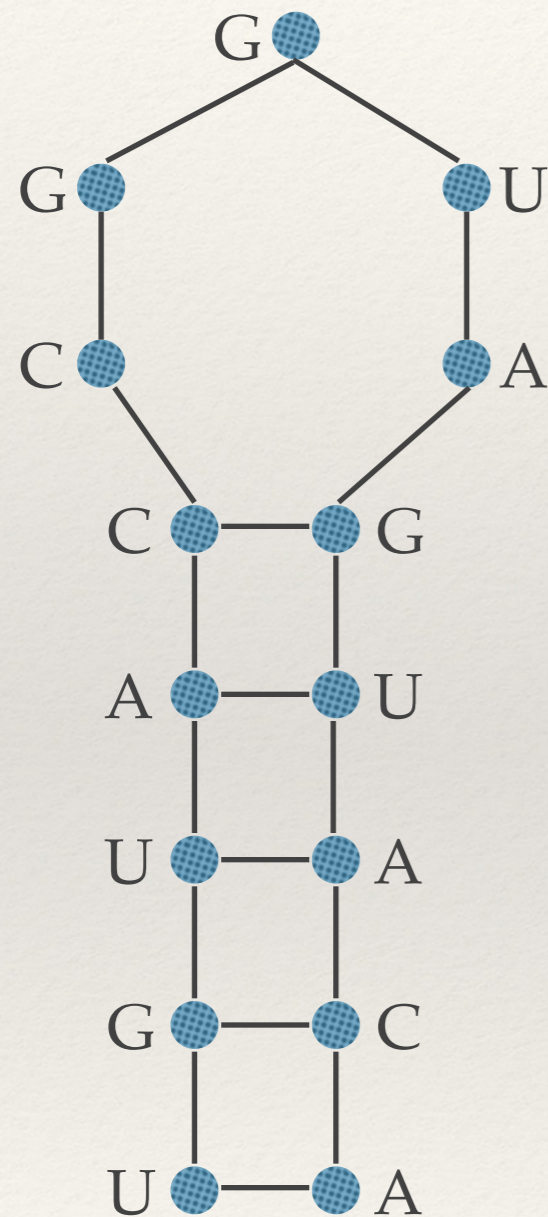
- ❖ An RNA molecule is a sequence of n symbols (bases) drawn from the alphabet $\Sigma = \{A, C, U, G\}$.
- ❖ Let $B = b_1b_2...b_n$ be an RNA molecule, where each $b_i \in \Sigma$.
- ❖ The RNA molecule forms a secondary structure based on a set of rules.



RNA Secondary Structure: Feasibility

- ❖ A secondary structure on B is a set of pairs $S=\{(i,j)\}$, where $i,j\in\{1,2,\dots,n\}$, that satisfies the following conditions:
 - ❖ The ends of each pair in S are separated by at least four intervening bases; that is, if $(i,j)\in S$, then $i < j-4$ (the “no sharp turns” condition).
 - ❖ The elements of any pair in S consist of either $\{A,U\}$ or $\{C,G\}$.
 - ❖ S is a matching: no base appears in more than one pair.
 - ❖ If (i,j) and (k,l) are two pairs in S , then we cannot have $i < k < j < l$ (the “noncrossing” condition).

RNA Secondary Structure



RNA Secondary Structure: Optimality

- ❖ Clearly, many RNA secondary structures may exist for a given RNA molecule.
- ❖ Out of all the feasible ones, which are the ones that are likely to arise under physiological conditions?
- ❖ A standard hypothesis is that an RNA molecule will form the secondary structure with the optimum total free energy.
- ❖ The correct model for the free energy of a secondary structure is a subject of much debate.
- ❖ A first approximation here is to assume that the free energy is proportional simply to the number of base pairs it contains.

- ❖ Step 2: Problem formulation
 - ❖ What are the input and output?

RNA Secondary Structure: Solution = Feasibility + Optimality

- ❖ The RNA Secondary Structure Prediction Problem can now be defined as:
 - ❖ Input: RNA molecule $B=b_1b_2...b_n$
 - ❖ Output: A secondary structure S with the maximum possible number of base pairs.

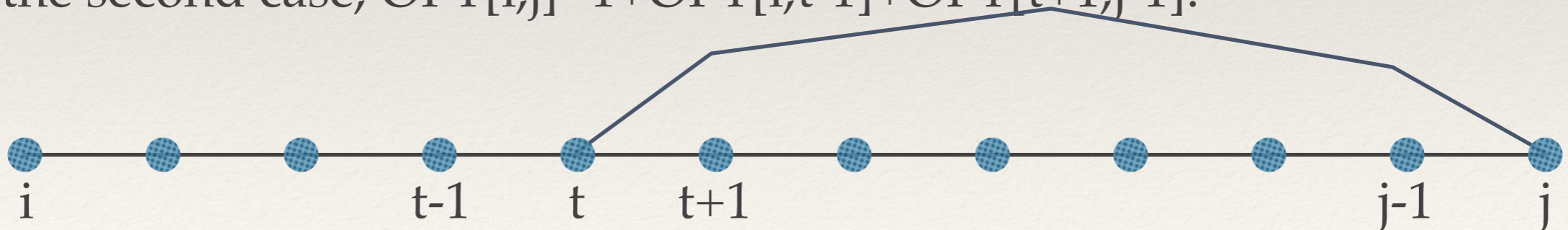
- ❖ Step 3: Designing an algorithm
 - ❖ We will use dynamic programming

RNA Secondary Structure Prediction

- ❖ Denote by $\text{OPT}[i,j]$ the maximum number of base pairs in a secondary structure on $b_i b_{i+1} \dots b_j$.
- ❖ By the no-sharp-turns condition, we know that $\text{OPT}[i,j]=0$ whenever $i \geq j-4$.
- ❖ Further, we know $\text{OPT}[1,n]$ is the solution we're looking for.
- ❖ Let's now reason about $\text{OPT}[i,j]$.

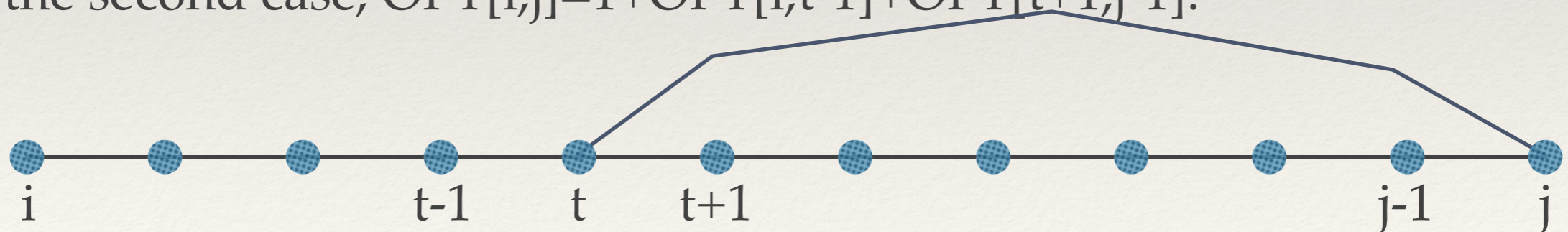
RNA Secondary Structure Prediction

- ❖ In the optimal secondary structure on $b_i b_{i+1} \dots b_j$, we have one of two cases:
 1. either j is not involved in a pair; or,
 2. j pairs with t , for some $t < j-4$.
- ❖ In the first case, we have $\text{OPT}[i,j] = \text{OPT}[i,j-1]$.
- ❖ In the second case, $\text{OPT}[i,j] = 1 + \text{OPT}[i,t-1] + \text{OPT}[t+1,j-1]$.



RNA Secondary Structure Prediction


- ❖ In the optimal secondary structure on $b_i b_{i+1} \dots b_j$, we have one of two cases:
 1. either j is not involved in a pair; or
 2. j pairs with t , for some $t < j-4$.
- ❖ In the first case, we have $\text{OPT}[i,j] = \text{OPT}[i,j-1]$.
- ❖ In the second case, $\text{OPT}[i,j] = 1 + \text{OPT}[i,t-1] + \text{OPT}[t+1,j-1]$.



RNA Secondary Structure Prediction

- ❖ There may be more than a single t value for which b_j and b_t can form a pair.
- ❖ Therefore, we have the following relationship:

$$(**) \text{OPT}[i,j] = \text{max}\{ \text{OPT}[i,j-1], \text{max}\{1+\text{OPT}[i,t-1]+\text{OPT}[t+1,j-1]\} \}$$



the max is taken over t such that
 b_j and b_t are an allowable base
pair.

RNA Secondary Structure Prediction

Initialize $\text{OPT}[i,j] \leftarrow 0$ whenever $i \geq j-4$;

For $k \leftarrow 5, 6, \dots, n-1$

For $i \leftarrow 1, 2, \dots, n-k$

$j \leftarrow i+k$;

$\text{OPT}[i,j] \leftarrow \max\{\text{OPT}[i,j-1], \max\{1+\text{OPT}[i,t-1]+\text{OPT}[t+1,j]: i \leq t < j, b_j \text{ and } b_t \text{ are complementary}\}\}$

Return $\text{OPT}[1,n]$

Initialize $\text{OPT}[i,j] \leftarrow 0$ whenever $i \geq j-4$;

For $k \leftarrow 5, 6, \dots, n-1$

For $i \leftarrow 1, 2, \dots, n-k$

$j \leftarrow i+k$;

$\text{OPT}[i,j] \leftarrow \max\{\text{OPT}[i,j-1], \max\{1 + \text{OPT}[i,t-1] + \text{OPT}[t+1,j] : i \leq t < j, b_j \text{ and } b_t \text{ are complementary}\}\}$

Return $\text{OPT}[1,n]$

```
Initialize  $\text{OPT}[i,j] \leftarrow 0$  whenever  $i \geq j-4$ ;  
For  $k \leftarrow 5, 6, \dots, n-1$   
  For  $i \leftarrow 1, 2, \dots, n-k$   
     $j \leftarrow i+k$ ;  
     $\text{OPT}[i,j] \leftarrow \max\{\text{OPT}[i,j-1], \max\{1 + \text{OPT}[i,t-1] + \text{OPT}[t+1,j] : i \leq t < j, b_j \text{ and } b_t \text{ are complementary}\}\}$   
Return  $\text{OPT}[1,n]$ 
```

RNA sequence ACCGGUAGU

```
Initialize OPT[i,j]←0 whenever i≥j-4;  
For k←5,6,...,n-1  
  For i←1,2,...,n-k  
    j←i+k;  
    OPT[i,j]←max{OPT[i,j-1],max{1+OPT[i,t-1]+OPT[t+1,j]: i≤t<j, bj and  
      bt are complementary}}  
Return OPT[1,n]
```

RNA sequence ACCGGUAGU

	4	0	0	0	
	3	0	0		
	2	0			
i	1				
j		6	7	8	9
	Initial values				

```
Initialize OPT[i,j]←0 whenever i≥j-4;  
For k←5,6,...,n-1  
  For i←1,2,...,n-k  
    j←i+k;  
    OPT[i,j]←max{OPT[i,j-1],max{1+OPT[i,t-1]+OPT[t+1,j]: i≤t<j, bj and  
      bt are complementary}}  
Return OPT[1,n]
```

RNA sequence ACCGGUAGU

4	0	0	0	
3	0	0		
2	0			
i 1				
j	6	7	8	9

Initial values

4	0	0	0	0
3	0	0	1	
2	0	0		
i 1	1			
j	6	7	8	9

k=5

```
Initialize OPT[i,j]←0 whenever i≥j-4;  
For k←5,6,...,n-1  
  For i←1,2,...,n-k  
    j←i+k;  
    OPT[i,j]←max{OPT[i,j-1],max{1+OPT[i,t-1]+OPT[t+1,j]: i≤t<j, bj and  
      bt are complementary}}  
Return OPT[1,n]
```

RNA sequence ACCGGUAGU

4	0	0	0	
3	0	0		
2	0			
i 1				
j	6	7	8	9

Initial values

4	0	0	0	0
3	0	0	1	
2	0	0		
i 1	1			
j	6	7	8	9

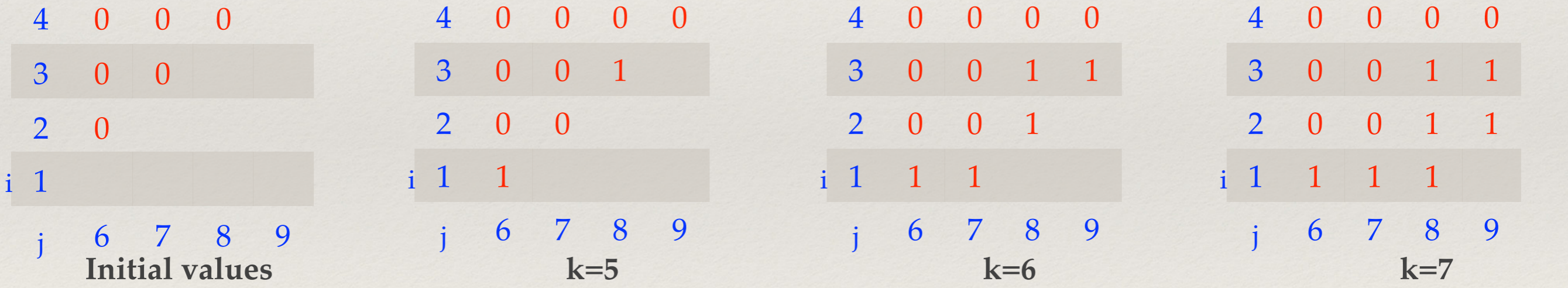
k=5

4	0	0	0	0
3	0	0	1	1
2	0	0	1	
i 1	1	1		
j	6	7	8	9

k=6

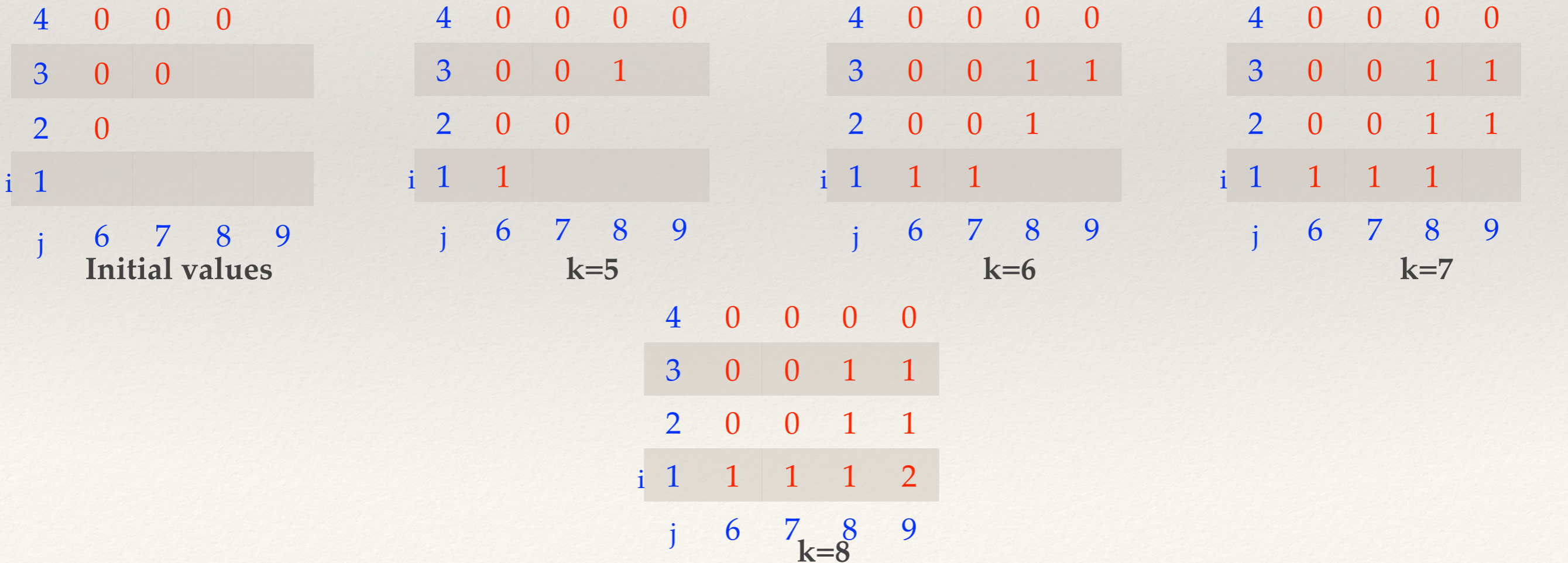
```
Initialize OPT[i,j]←0 whenever i≥j-4;  
For k←5,6,...,n-1  
  For i←1,2,...,n-k  
    j←i+k;  
    OPT[i,j]←max{OPT[i,j-1],max{1+OPT[i,t-1]+OPT[t+1,j]: i≤t<j, bj and  
      bt are complementary}}  
Return OPT[1,n]
```

RNA sequence ACCGGUAGU



Initialize $OPT[i,j] \leftarrow 0$ whenever $i \geq j-4$;
For $k \leftarrow 5, 6, \dots, n-1$
 For $i \leftarrow 1, 2, \dots, n-k$
 $j \leftarrow i+k$;
 $OPT[i,j] \leftarrow \max\{OPT[i,j-1], \max\{1+OPT[i,t-1]+OPT[t+1,j]: i \leq t < j, b_j \text{ and } b_t \text{ are complementary}\}\}$
Return $OPT[1,n]$

RNA sequence ACCGGUAGU



RNA Secondary Structure Prediction

- ❖ How do we get the actual secondary structure from the solution?
- ❖ What is the running time of the algorithm?

Questions?