

**CSE 421**

**Dynamic Programming**

Shayan Oveis Gharan

# RNA Secondary Structure

# RNA Secondary Structure (Formal)

**Secondary structure.** A set of pairs  $S = \{ (b_i, b_j) \}$  that satisfy:

[Watson-Crick.]

- $S$  is a *matching* and
- each pair in  $S$  is a Watson-Crick pair: A-U, U-A, C-G, or G-C.

[No sharp turns.]: The ends of each pair are separated by at least 4 intervening bases. If  $(b_i, b_j) \in S$ , then  $i < j - 4$ .

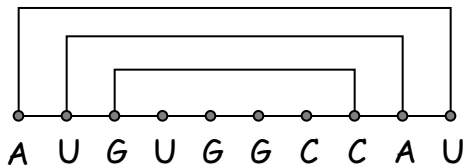
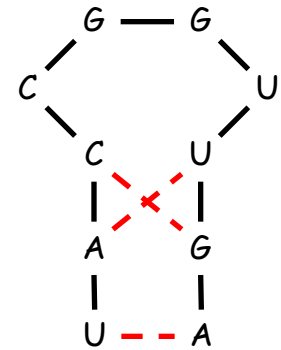
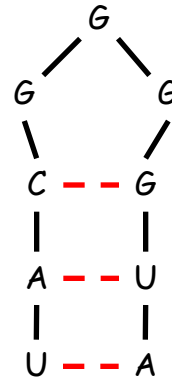
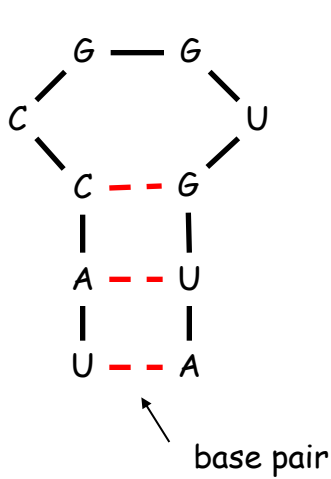
[Non-crossing.] If  $(b_i, b_j)$  and  $(b_k, b_l)$  are two pairs in  $S$ , then we cannot have  $i < k < j < l$ .

**Free energy:** Usual hypothesis is that an RNA molecule will maximize total free energy.

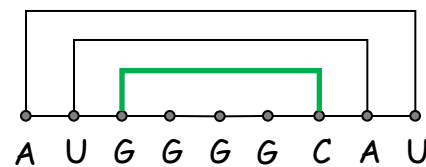
↑  
approximate by number of base pairs

**Goal:** Given an RNA molecule  $B = b_1b_2\dots b_n$ , find a secondary structure  $S$  that maximizes the number of base pairs.

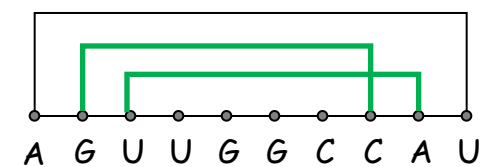
# Secondary Structure (Examples)



ok



~~sharp turn~~



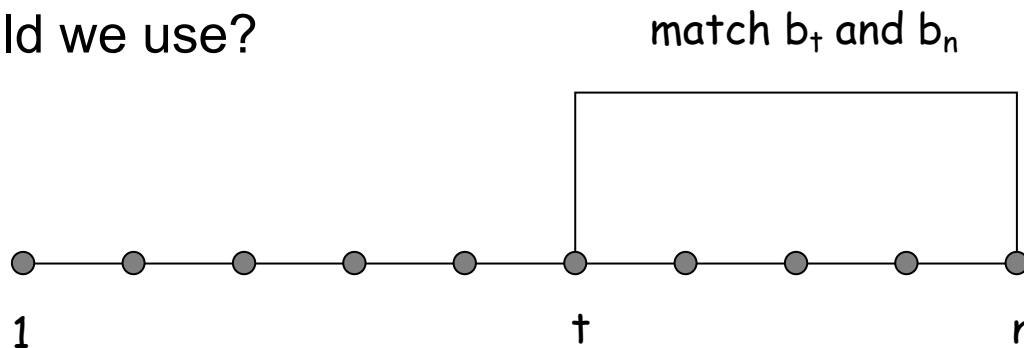
~~crossing~~

# DP: First Attempt

**First attempt.** Let  $OPT(n)$  = maximum number of base pairs in a secondary structure of the substring  $b_1b_2\dots b_n$ .

Suppose  $b_n$  is matched with  $b_t$  in  $OPT(n)$ .

What IH should we use?



**Difficulty:** This naturally reduces to two subproblems

- Finding secondary structure in  $b_1, \dots, b_{t-1}$ , i.e.,  $OPT(t-1)$
- Finding secondary structure in  $b_{t+1}, \dots, b_{n-1}$ , ???

# DP: Second Attempt

**Definition:**  $OPT(i, j)$  = maximum number of base pairs in a secondary structure of the substring  $b_i, b_{i+1}, \dots, b_j$

The most important part of a correct DP; It fixes IH

**Case 1:** If  $j - i \leq 4$ .

- $OPT(i, j) = 0$  by no-sharp turns condition.

**Case 2:** Base  $b_j$  is not involved in a pair.

- $OPT(i, j) = OPT(i, j-1)$

**Case 3:** Base  $b_j$  pairs with  $b_t$  for some  $i \leq t < j - 4$

- non-crossing constraint **decouples** resulting sub-problems
- $OPT(i, j) = \max_{t: b_i \text{ pairs with } b_t} \{ 1 + OPT(i, t - 1) + OPT(t + 1, j - 1) \}$

# Recursive Code

Let  $M[i,j]$ =empty for all  $i,j$ .

```
Compute-OPT(i,j){
  if (j-i <= 4)
    return 0;
  if (M[i,j] is empty)
    M[i,j]=Compute-OPT(i,j-1)
  for t=i to j-5 do
    if ( $b_t, b_j$  is in {A-U, U-A, C-G, G-C})
      M[i,j]=max(M[i,j], 1+Compute-OPT(i,t-1) +
        Compute-OPT(t+1,j-1))
  return M[j]
}
```

Does this code terminate?  
What are we inducting on?

# Formal Induction

Let  $OPT(i, j)$  = maximum number of base pairs in a secondary structure of the substring  $b_i, b_{i+1}, \dots, b_j$

**Base Case:**  $OPT(i, j) = 0$  for all  $i, j$  where  $|j - i| \leq 4$ .

**IH:** For some  $\ell \geq 4$ , Suppose we have computed  $OPT(i, j)$  for all  $i, j$  where  $|i - j| \leq \ell$ .

**IS:** Goal: We find  $OPT(i, j)$  for all  $i, j$  where  $|i - j| = \ell + 1$ . Fix  $i, j$  such that  $|i - j| = \ell + 1$ .

**Case 1:** Base  $b_j$  is not involved in a pair.

- $OPT(i, j) = OPT(i, j-1)$  [this we know by IH since  $|i - (j - 1)| = \ell$ ]

**Case 2:** Base  $b_j$  pairs with  $b_t$  for some  $i \leq t < j - 4$

- $OPT(i, j) = \max_{t: b_i \text{ pairs with } b_t} \{ 1 + OPT(i, t - 1) + OPT(t + 1, j - 1) \}$

We know by IH since difference  $\leq \ell$



# Bottom-up DP

```
for k = 1, 2, ..., n-1
  for i = 1, 2, ..., n-1
    j = i + k
    if (j-i <= 4)
      M[i,j]=0;
    else
      M[i,j]=M[i,j-1]
      for t=i to j-5 do
        if ( $b_t, b_j$  is in {A-U, U-A, C-G, G-C})
          M[i,j]=max(M[i,j], 1+ M[i,t-1] + M[t+1,j-1])

return M[1, n]
}
```

4	0	0	0	↗
3	0	0	↗	↗
2	0	↗	↗	↗
1	↗	↗	↗	↗
	6	7	8	9

j

Running Time:  $O(n^3)$

# Lesson

We may not always induct on  $i$  or  $w$  to get to smaller subproblems.

We may have to induct on  $|i - j|$  or  $i + j$  when we are dealing with more complex problems, e.g., intervals

# Sequence Alignment

# Word Alignment

How similar are two strings?

**ocurrance**

**occurrence**

o	c	u	r	r	a	n	c	e	-
---	---	---	---	---	---	---	---	---	---

o	c	c	u	r	r	e	n	c	e
---	---	---	---	---	---	---	---	---	---

5 mismatches, 1 gap

o	c	-	u	r	r	a	n	c	e
---	---	---	---	---	---	---	---	---	---

o	c	c	u	r	r	e	n	c	e
---	---	---	---	---	---	---	---	---	---

1 mismatch, 1 gap

o	c	-	u	r	r	-	a	n	c	e
---	---	---	---	---	---	---	---	---	---	---

o	c	c	u	r	r	e	-	n	c	e
---	---	---	---	---	---	---	---	---	---	---

0 mismatches, 3 gaps

# Edit Distance

**Edit distance.** [Levenshtein 1966, Needleman-Wunsch 1970]

Cost = # of gaps + #mismatches.

## Applications.

- Basis for Unix diff and Word correct in editors.
- Speech recognition.
- Computational biology.

C T G A C C T A C C T

C C T G A C T A C A T

Cost: 5

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

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

Cost: 3

# Sequence Alignment

Given two strings  $x_1, \dots, x_m$  and  $y_1, \dots, y_n$  find an alignment with minimum number of mismatch and gaps.

An alignment is a set of ordered pairs  $(x_{i_1}, y_{j_1}), (x_{i_2}, y_{j_2}), \dots$  such that  $i_1 < i_2 < \dots$  and  $j_1 < j_2 < \dots$

**Example:** CTACCG VS. TACATG.

Sol: We aligned

$x_2-y_1, x_3-y_2, x_4-y_3, x_5-y_4, x_6-y_6$ .

So, the cost is 3.

$x_1$	$x_2$	$x_3$	$x_4$	$x_5$		$x_6$
C	T	A	C	C	-	G
	T	A	C	A	T	G
	$y_1$	$y_2$	$y_3$	$y_4$	$y_5$	$y_6$

# DP for Sequence Alignment

Let  $OPT(i, j)$  be min cost of aligning  $x_1, \dots, x_i$  and  $y_1, \dots, y_j$

**Case 1:** OPT matches  $x_i, y_j$

- Then, pay mis-match cost if  $x_i \neq y_j$  + min cost of aligning  $x_1, \dots, x_{i-1}$  and  $y_1, \dots, y_{j-1}$  i.e.,  $OPT(i-1, j-1)$

**Case 2:** OPT leaves  $x_i$  unmatched

- Then, pay gap cost for  $x_i$  +  $OPT(i-1, j)$

**Case 3:** OPT leaves  $y_j$  unmatched

- Then, pay gap cost for  $y_j$  +  $OPT(i, j-1)$

# Bottom-up DP

```
Sequence-Alignment(m, n, x1x2...xm, y1y2...yn) {  
  for i = 0 to m  
    M[0, i] = i  
  for j = 0 to n  
    M[j, 0] = j  
  
  for i = 1 to m  
    for j = 1 to n  
      M[i, j] = min( (xi=yj ? 0:1) + M[i-1, j-1],  
                    1 + M[i-1, j],  
                    1 + M[i, j-1])  
  
  return M[m, n]  
}
```

**Analysis:**  $\Theta(mn)$  time and space.

**English words or sentences:**  $m, n \leq 10, \dots, 20$ .

**Computational biology:**  $m = n = 100,000$ . 10 billions ops OK,  
but 40GB array?



# Optimizing Memory

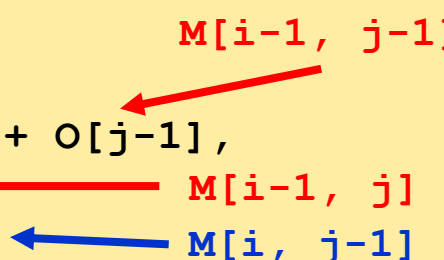
If we are not using strong induction in the DP, we just need to use the last (row) of computed values.

```
Sequence-Alignment(m, n, x1x2...xm, y1y2...yn) {  
  for i = 0 to m  
    M[0, i] = i  
  for j = 0 to n  
    M[j, 0] = j  
  
  for i = 1 to m  
    for j = 1 to n  
      M[i, j] = min( (xi=yj ? 0:1) + M[i-1, j-1],  
                    1 + M[i-1, j],  
                    1 + M[i, j-1])  
  
  return M[m, n]  
}
```

Just need  $i - 1, i$  rows  
to compute  $M[i, j]$

# DP with $O(m + n)$ memory

- Keep an Old array containing values of the last row
- Fill out the new values in a New array
- Copy new to old at the end of the loop

```
Sequence-Alignment(m, n, x1x2...xm, y1y2...yn) {  
  for i = 0 to m  
    O[i] = i  
  for i = 1 to m  
    N[0]=i  
    for j = 1 to n  
      N[j] = min( (xi=yj ? 0:1) + O[j-1],  
                 1 + O[j],  
                 1 + N[j-1])  
        
    for j = 1 to n  
      O[j]=N[j]  
  return N[n]  
}
```

# Lesson

Advantage of a bottom-up DP:

It is much easier to optimize the space.

# Longest Path in a DAG

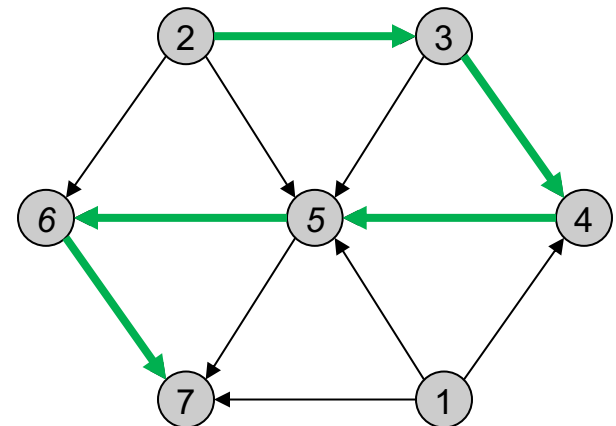
# Longest Path in a DAG

**Goal:** Given a DAG  $G$ , find the longest path.

**Recall:** A directed graph  $G$  is a DAG if it has no cycle.

This problem is NP-hard for general directed graphs:

- It has the Hamiltonian Path as a special case

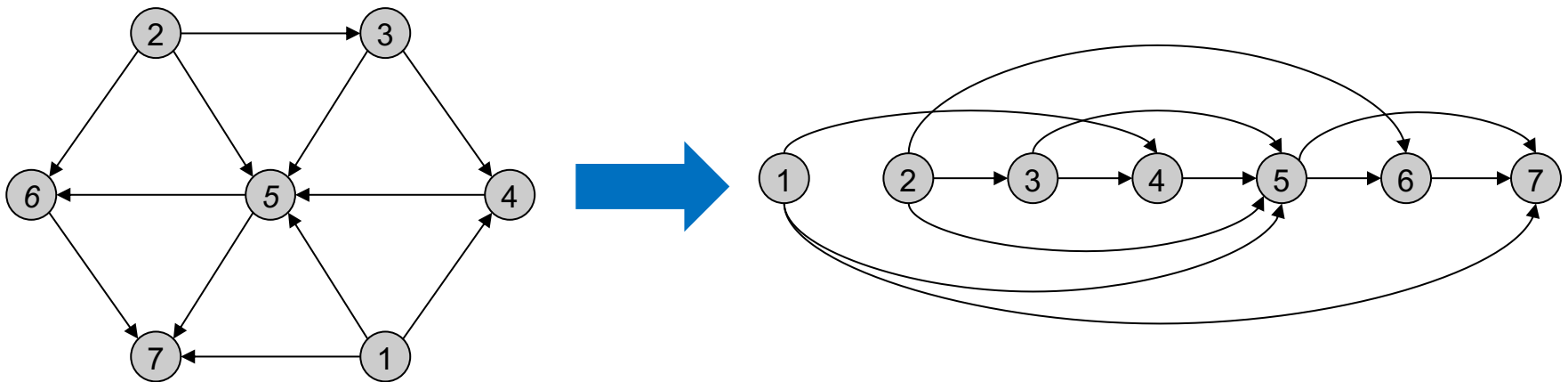


# DP for Longest Path in a DAG

Q: What is the right **ordering**?

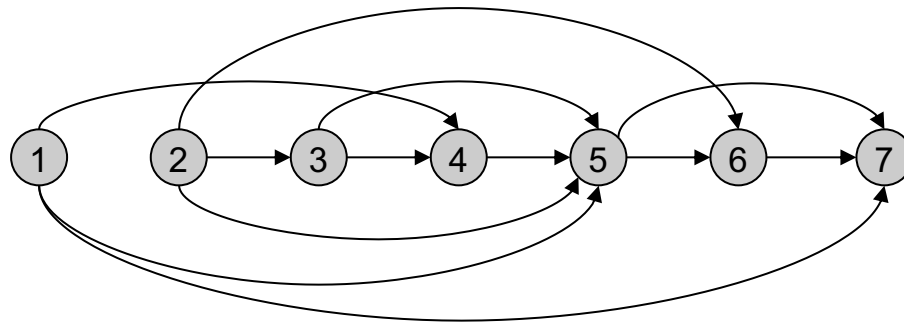
Remember, we have to use that G is a DAG, ideally in defining the ordering

We saw that every DAG has a **topological sorting**  
So, let's use that as an ordering.



# DP for Longest Path in a DAG

Suppose we have labelled the vertices such that  $(i, j)$  is a directed edge only if  $i < j$ .



Let  $OPT(j)$  = length of the longest path ending at  $j$

Suppose in the longest path ending at  $j$ , last edge is  $(i, j)$ .

Then, **none** of the  $i + 1, \dots, j - 1$  are in this path since topological ordering. Furthermore the path ending at  $i$  must be the longest path ending at  $i$ ,

$$OPT(j) = OPT(i) + 1.$$

# DP for Longest Path in a DAG

Suppose we have labelled the vertices such that  $(i, j)$  is a directed edge only if  $i < j$ .

Let  $OPT(j)$  = length of the longest path ending at  $j$

$$OPT(j) = \begin{cases} 0 & \text{If } j \text{ is a source} \\ 1 + \max_{i:(i,j) \text{ an edge}} OPT(i) & \text{o.w.} \end{cases}$$



# DP for Longest Path in a DAG

Let  $G$  be a DAG given with a topological sorting: For all edges  $(i, j)$  we have  $i < j$ .

```
Compute-OPT(j) {
    if (in-degree(j) == 0)
        return 0
    if (M[j] == empty)
        M[j] = 0;
    for all edges (i, j)
        M[j] = max(M[j], 1 + Compute-OPT(i))
    return M[j]
}
Output max(M[1], ..., M[n])
```

Running Time:  $O(n + m)$

Memory:  $O(n)$

Can we output the longest path?

# Outputting the Longest Path

Let  $G$  be a DAG given with a topological sorting: For all edges  $(i, j)$  we have  $i < j$ .

**Initialize**  $\text{Parent}[j] = -1$  for all  $j$ .

**Compute-OPT**( $j$ ) {

**if** ( $\text{in-degree}(j) == 0$ )

**return** 0

**if** ( $M[j] == \text{empty}$ )

$M[j] = 0$ ;

**for** all edges  $(i, j)$

**if** ( $M[j] < 1 + \text{Compute-OPT}(i)$ )

$M[j] = 1 + \text{Compute-OPT}(i)$

**Parent**[ $j$ ]= $i$

**return**  $M[j]$

}

Let  $M[k]$  be the maximum of  $M[1], \dots, M[n]$

**While** ( $\text{Parent}[k] \neq -1$ )

    Print  $k$

$k = \text{Parent}[k]$

Record the entry that  
we used to compute  $\text{OPT}(j)$

