Dynamic Programming Overview

Main idea of DP

STEP 1: Recursively define the value of an optimal solution (difficult part)

STEP 2: Compute the value of an optimal solution from the smallest to the largest problems.

One Dimensional (1D) Dynamic Programming

- Solve problems of minimum size first.
- Store solutions in 1D array.
- Gradually increase problem size.
- Choose the order that you solve problems, so that you re-use solutions of smaller problems stored in the 1D array.
- How to gradually increase the problem size:
  - If you need to select among a set of n items, order items (sometimes in arbitrary order) and allow selection among the first $1, 2, \ldots, n$ items.
  - If you need to solve for an integer amount or length n, gradually solve for $1, 2, \ldots, n$

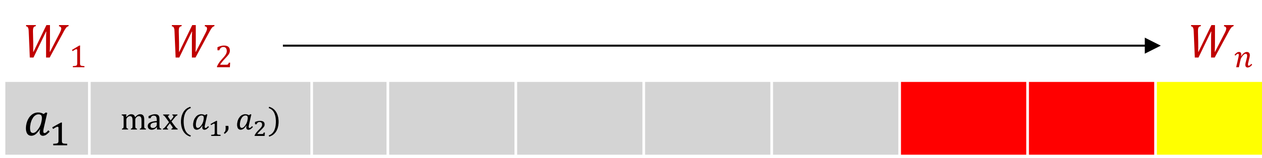
1D DP Simple Problems $\Theta(n)$

1. Fibonacci Numbers $\Theta(n)$

$$F(n) = F(n-1) + F(n-2), \qquad F(1) = 1, \; F(2) = 2$$

2. Maximum Sum problem $\Theta(n)$: Given a sequence A of n positive numbers $a_1, a_2, \dots, a_n$, find a subset S of A that has the maximum sum, provided that if we select a_i in S, then we cannot select a_{i-1} or a_{i+1}.

- Let A_i be the subsequence of the first i numbers ($i \le n$): $a_1, a_2, \dots, a_i$
- Let W_i be the sum of numbers in the optimal solution for A_i.
- Recurrence: $W_i = \max\{W_{i-2} + a_i, W_{i-1}\}$



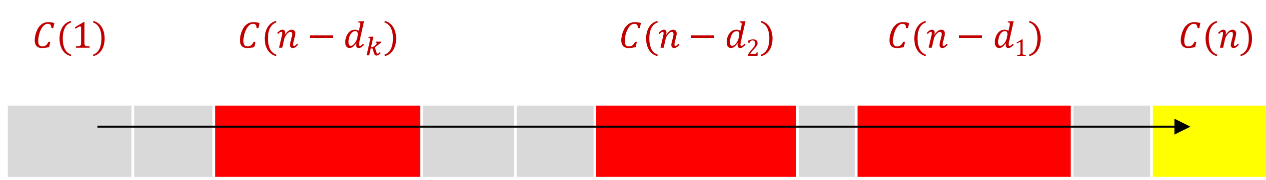
1D DP More Complex Problems

1. Rod Cutting Problem $\Theta(n^2)$: Given a rod of length n and prices p_i for $i = 1, \ldots, n$, where p_i is the price of a rod of length i, cut the rod in order to maximize the total revenue.

- r_n is maximum revenue from cutting rod of length n
- Recurrence: $r_n = \max_{1 \le i \le n} \{p_i + r_{n-i}\}$, $\quad r_0 = 0$

2. Minimum number of Coins $\Theta(nk)$: Given amount n and k denominations $d_1, \ldots, d_k$, find minimum number of coins for amount n

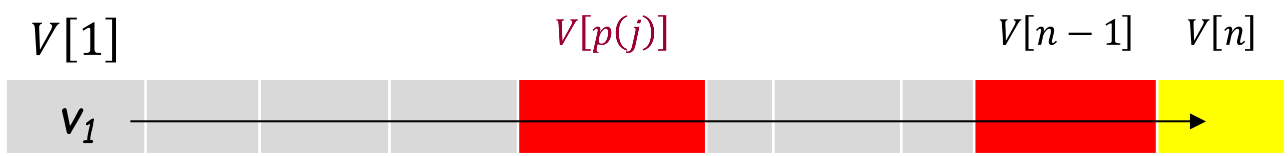
- Let d_i be the last denomination used
- Then: $C[n] = 1 + C[n - d_i]$
  - Because after using 1 coin, the amount left is $n - d_i$
- But I have to consider all possible (k) denominations
- Recurrence: $C[n] = 1 + \min\{C[n - d_i] : i \in [1, k]\}$



1D DP More Complex Problems -2

1. Weighted interval scheduling $\Theta(n \log n)$.
- Job j starts at s_j, finishes at f_j, and has weight (or value) v_j.
- Two jobs compatible if they don't overlap.
- Goal: find maximum-weight subset of mutually compatible jobs.

Sort jobs by finishing time: $f_1 \le f_2 \le \cdots \le f_n$.
- $p(j)$ = largest index $i < j$ such that job i is compatible with job j.
- $V[j]$ = value of optimal solution to the problem on jobs $1, 2, \ldots, j$.
- Recurrence: $V[j] = \max\{ v_j + V[p(j)], V[j-1] \}, V[0] = 0$



2. Similar problem- Highway Billboards $\Theta(n \log n)$: Consider a highway from west to east. You can place billboards at locations $x_1, x_2, \ldots, x_n$. If you place a billboard at x_i, you receive revenue of $r_i > 0$. You wish to place billboards as to maximize your total revenue, subject to the restriction that any two billboards must be at least 5 miles apart.
- Sort the sites in increasing order of location $\{x_1, x_2, \ldots, x_n\}$.
- Recurrence and algorithm same as Weighted interval scheduling.

Exercise on Longest Oscillating Subsequence

A sequence of numbers $a_1, a_2, \ldots a_n$ is *oscillating* if

- $a_i < a_{i+1}$ for every odd index i and
- $a_i > a_{i+1}$ for every even index i

Example:

1	2	3	4	5	6	7	8
3	8	2	7	4	10	5	6

Describe a DP algorithm to find a longest oscillating subsequence (LOS) in a sequence of n integers. Assume that all numbers are distinct. Successive numbers in LOS do not have to be immediate neighbors in the initial sequence.

Longest Oscillating Subsequence: Recurrence

$o[i]$: length of the longest oscillating subsequence that ends at a_i and has an odd length

- if a_j is the last number before a_i in subsequence ($j < i$) with even length, it must hold $a_j > a_i$.

$e[i]$: length of the longest oscillating subsequence that ends at a_i and has an even length

- if a_j is the last number before a_i in subsequence ($j < i$) with odd length, it must hold $a_j < a_i$.

Base Case: $o[1] = 1; \quad e[1] = -\infty$.

General Recurrences:

- $$\mathrm{o}[i] = 1 + \max_{j<i \ \& \ a_j > a_i} \{0, e[j]\}$$
- $$e[i] = \begin{cases} -\infty & \text{if } \ a_i < a_j \text{ for all } j < i \\ 1 + \max_{j<i \ \& \ a_j < a_i} \{o[j]\} & \text{otherwise} \end{cases}$$

Longest Oscillating Subsequence: Algorithm

- 1D Dynamic Programming
- Calculate the solution to problems of size $1, 2, \ldots, i, \ldots, n$, where size i contains the first i numbers.
- As opposed to previous problems we now need 2 separate tables for $\mathrm{o}[i]$ and $e[i]$
- Values in the tables are stored in order $o[1], e[1], o[2], e[2], o[3], e[3], \ldots$.
- What is the running time of the algorithm?
- $\Theta(n^2)$

Two Dimensional (2D) Dynamic Programming

- These problems require a 2D array for the storage of solutions
- In all the problems, we fill the 2D array row-by-row
  - That is, first we finish the first row, then the second and so on.
- Usually, but not always, the final solution is at the bottom right corner of the array.
- In all the problems discussed during class, the running time is the same as the array size.
- However, in some cases, we do not need to keep the entire array, as algorithms only require the last two rows.

2D DP 0-1 Knapsack $\Theta(nW)$

1. Input: A set of n items, where item i has weight w_i and value v_i, and a knapsack with capacity W.

Goal: Find $x_1, \ldots, x_n \in \{0,1\}$ such that $\sum_{i=1}^{n} x_i w_i \le W$ and $\sum_{i=1}^{n} x_i v_i$ is maximized.

- $V[i,j]$ be the largest obtained value for a knapsack with capacity j, choosing only from the first i items.
- Recurrence: $V[i,j] = \max(V[i-1,j],\ \ v_i + V[i-1, j-w_i])$

$$V[i,j] = 0 \quad \text{if} \quad i = 0 \ or \ j = 0$$

	j=0	j=1	j=2			
i=0	0	0	0	0	0	0
i=1	0	0	…	v_1	v_1	v_1
i=2	0	0	…	v_1 or v_2	$\cdots$	$v_1 + v_2$
	0					
	0					
	0					
	0					

	j=0	j=1	j=2			$j = W$
i=0	0	0	0	0	0	0
i=1	0	0	…	v_1	…	v_1
i=2	0	0	…	v_1 or v_2	…	$v_1 + v_2$
	0					
	0					
	0			$V[n-1, W-w_n]$	…	$V[n-1, W]$
i=n	0					$V[n, W]$

2D DP Longest Common Subsequence $\Theta(mn)$

Given two sequences $X = (x_1, x_2, \ldots, x_m)$ and $Y = (y_1, y_2, \ldots, y_n)$, Z is a common subsequence of X and Y if Z has a strictly increasing sequence of indices i and j of both X and Y such that we have $x_{i_p} = y_{j_p} = z_p$ for all $p = 1, 2, \ldots, k$. Find the LCS of X and Y.

- $c[i, j]$ is the length of the LCS of $X[1..i]$ and $Y[1..j]$
- Recurrence: $c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1, j-1] + 1 & \text{if } x_i = y_j \\ \max\{c[i, j-1], c[i-1, j]\} & \text{if } x_i \neq y_j \end{cases}$

	j=0	j=1	j=2			
i=0	0	0	0	0	0	0
i=1	0	0	…	1	…	1
i=2	0	0	…	1	…	2
	0					
	0					
	0					
	0					

	j=0	j=1	j=2			$j = n$
i=0	0	0	0	0	0	0
i=1	0	0	…	1		1
i=2	0	0	…	1		2
	0					
	0					
	0				$c[m-1, n-1]$	$c[m-1, n]$
i=m	0				$c[m, n-1]$	$c[m, n]$

2D DP Longest Common Substring $\Theta(mn)$

Given two strings $X = x_1 x_2 \dots x_m$ and $Y = y_1 y_2 \dots y_n$, we wish to find their longest common substring Z, that is, the largest k for which there are indices i and j with $x_i x_{i+1} \dots x_{i+k-1} = y_j y_{j+1} \dots y_{j+k-1}$.

- $d[i,j]$ = the length of the longest common substring of $X[1..i]$ and $Y[1..j]$ that ends at x_i and y_j.

Recurrence: $d[i,j] = \begin{cases} d[i-1,j-1]+1 & \text{if } x_i = y_j \\ 0 & \text{if } x_i \neq y_j \end{cases}$

	j=0	j=1	j=2			
i=0	0	0	0	0	0	0
i=1	0	0	…	1	…	0
i=2	0	0	…	2	…	0
	0					
	0					
	0					
	0					

	j=0	j=1	j=2			$j = n$
i=0	0	0	0	0	0	0
i=1	0	0	…	1		0
i=2	0	0	…	1		0
	0					
	0					max d
	0				$d[m-1, n-1]$	
i=m	0					$d[m,n]$

2D DP Edit Distance $\Theta(mn)$

Find the edit distance between strings $X = x_1x_2 \dots x_m$ and $Y = y_1y_2 \dots y_n$.
Edit distance is the smallest number of operations to turn X into Y :
1. Insertion: add a letter 2. Deletion: remove a letter 3. Substitution: replace a character with another one.

- $E[i,j]$ = edit distance of $X[1..i]$ and $Y[1..j]$

$$E[i,j] = min\begin{cases} 1 + E[i, j-1] \\ 1 + E[i-1, j] \\ E[i-1, j-1] \quad \text{if } x_i = y_j, or\ E[i-1, j-1] + 1 \text{ if } x_i \neq y_j, \end{cases}$$

	j=0	j=1	j=2			
i=0	0	1	2			n
i=1	1	0 or 1	…		…	
i=2	2		…		…	
	m					

	j=0	j=1	j=2			$j = n$
i=0	0	1	2			n
i=1	1	0/1				
i=2	2					
					$E[m-1, n-1]$	$E[m-1, n]$
i=m	m				$E[m, n-1]$	$E[m, n]$

Exercise on Min-Cost Path in 2-D Matrix

Given a cost matrix where $\mathrm{Cost}[i, j]$ is the positive cost of visiting cell with coordinates (i, j), find a min-cost path to reach a cell (x, y) from cell (1,1) at the top-left corner, under the condition that you can only travel one step right or one step down.

	Column 1	Column 2	Column 3
Row 1	$\mathrm{Cost}[1,1]=0$	2 2	4 6
Row 2	4 4	7 9	2 8
Row 3	1 5	1 6	3 9

Recurrence:

$$\mathrm{MinCost}(i, j) = \min\{\mathrm{MinCost}(i-1, j), \mathrm{MinCost}(i, j-1)\} + \mathrm{Cost}[i, j]$$

Question: *give a recurrence for the number of ways* ($num(i, j)$) to reach cell with coordinates (i, j) starting from cell (1,1), under the condition that you can only travel one step right or one step down.

$$num(i, j) = num(i-1, j) + num(i, j-1)$$

Exercise on Min-Cost Path in 2-D Matrix

Given a cost matrix where $\text{Cost}[i,j]$ is the positive cost of visiting cell with coordinates (i,j), find a min-cost path to reach a cell (x,y) from cell (1,1) at the top-left corner, if one is also allowed to move diagonally lower from cell (i,j) to cell $(i+1,j+1)$.

	Column 1	Column 2	Column 3
Row 1	$\text{Cost}[1,1] = 0$	2 2	4 6
Row 2	4 4	7 7	2 4
Row 3	1 5	1 5	3 7

Recurrence:

$$\text{MinCost}(i,j) = min\left\{\begin{matrix} \text{MinCost}(i-1,j) \\ \text{MinCost}(i,j-1) \\ \text{MinCost}(i-1,j-1) \end{matrix}\right\} + \text{Cost}[i,j]$$

Exercise Max Square Sub-Matrix with all 1s

Given an $n \times m$ binary matrix M filled with 0's and 1's, find the dimension of the largest square containing all 1's .

Example:

M

0	1	1	0	1
1	1	0	1	0
0	1	1	1	0
1	1	1	1	0
1	1	1	1	1
0	0	0	0	0

- $S[i,j]$: size of the square sub-matrix with all 1s including $M[i,j]$, where $M[i,j]$ is the right bottom entry in sub-matrix.
- Base Cases: $S[1,j] = M[1,j]$, $S[i,1] = M[i,1]$

j=1

i=1

S

0	1	1	0	1
1				
0				
1				
1				
0				

Max Square Sub-Matrix with all 1s: Recurrence

Recurrence:

- If $M[i,j] = 1$, then $S[i,j] = \min\{S[i,j-1], S[i-1,j], S[i-1,j-1]\} + 1$
- If $M[i,j] = 0$, then $S[i,j] = 0$

M

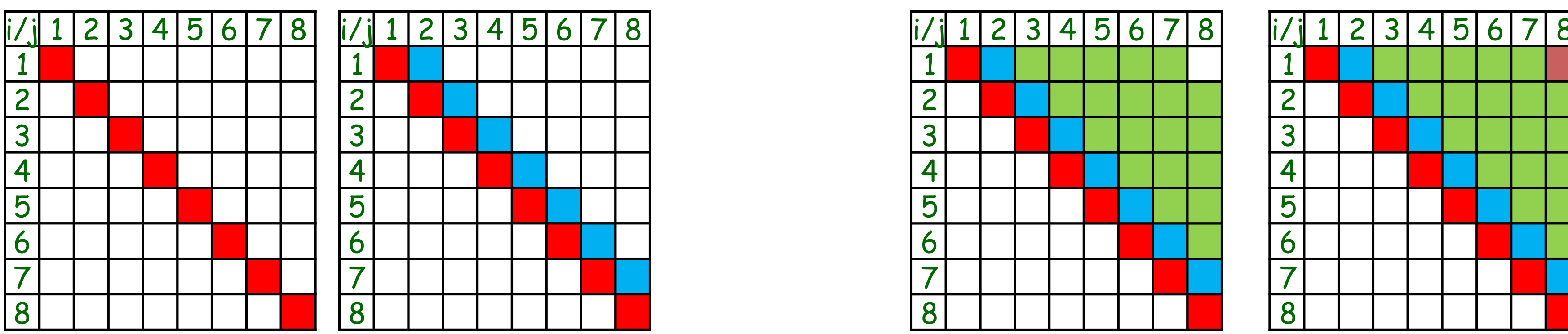
0	1	1	0	1
1	1	0	1	0
0	1	1	1	0
1	1	1	1	0
1	1	1	1	1
0	0	0	0	0

S

	j=1				
i=1	0	1	1	0	1
	1	1	0	1	0
	0	1	1	1	0
	1	1	2	2	0
	1	2	2	3	1
	0	0	0	0	0

Interval Dynamic Programming

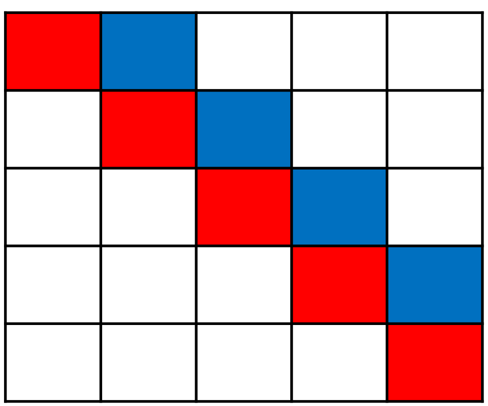
- The problem input has some inherent order $1, 2, \dots, n$
- First we solve n problems of length 1, i.e., $\text{problem}[1,1], \text{problem}[2,2], \dots, \text{problem}[n, n]$.
  - All solutions are stored in diagonals of a 2D array. Solutions to small problems are re-used by bigger ones.
- Then, we solve $n - 1$ problems of length 2, i.e., $\text{problem}[1,2], \text{problem}[2,3], \dots, \text{problem}[n - 1, n]$.
- ....
- Then, we solve $n - l + 1$ problems of length l, i.e., $\text{problem}[1, l], \text{problem}[2,l], \dots, \text{problem}[n - l + 1, n]$.
- Finally, we solve 1 problem of size n: $\text{problem}[1, n]$.

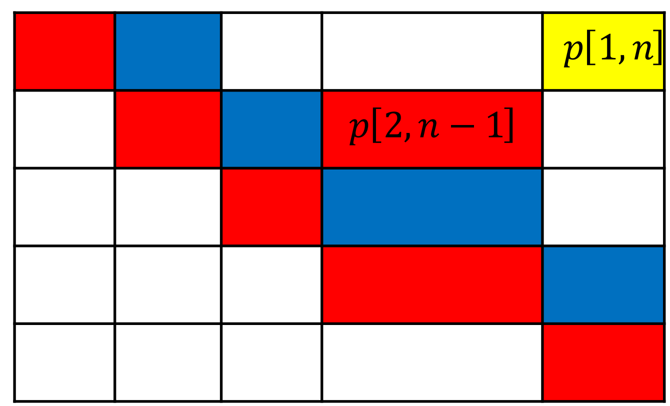


Longest Palindromic Substring $\Theta(n^2)$

A palindrome is a string that reads the same backward or forward. Given a string $X = x_1 x_2 \dots x_n$, find the longest palindromic substring.

- $p[i, j]$ is $true$ iff $X[i..j]$ is a palindrome
- Problems of size 1: $p[i, i] = true$, for all i
- Problems of size 2: $p[i, i+1] = true$ if $x_i = x_{i+1}$
- Recurrence: $p[i, j] = true$ if $x_i = x_j$ AND $p[i+1, j-1] = true$

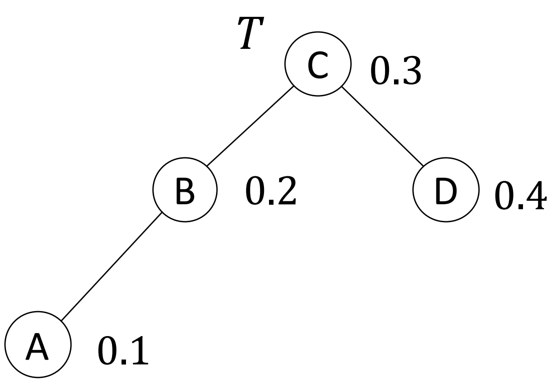




Optimal Binary Search Tree $\Theta(n^3)$

Given n keys $a_1 < a_2 < \cdots < a_n$, with weights $f(a_1), \ldots, f(a_n)$, find a binary search tree (BST) T on these n keys such that $\sum_{i=1}^{n} f(a_i)(d(a_i)+1)$ is minimized, where $d(a_i)$ is the depth of a_i.

- $w[i,j] = f(a_i) + \cdots + f(a_j)$
- $e[i,j]$ = the minimum cost of any BST on $a_i, \ldots, a_j$ ($e[i,j] = 0$ for $i > j$)
- Problems of size 1: $e[i,i] = f(a_i)$ for all i
- Recurrence: $e[i,j] = \min_{i \le k \le j} \{e[i,k-1] + e[k+1,j] + w[i,j]\}$

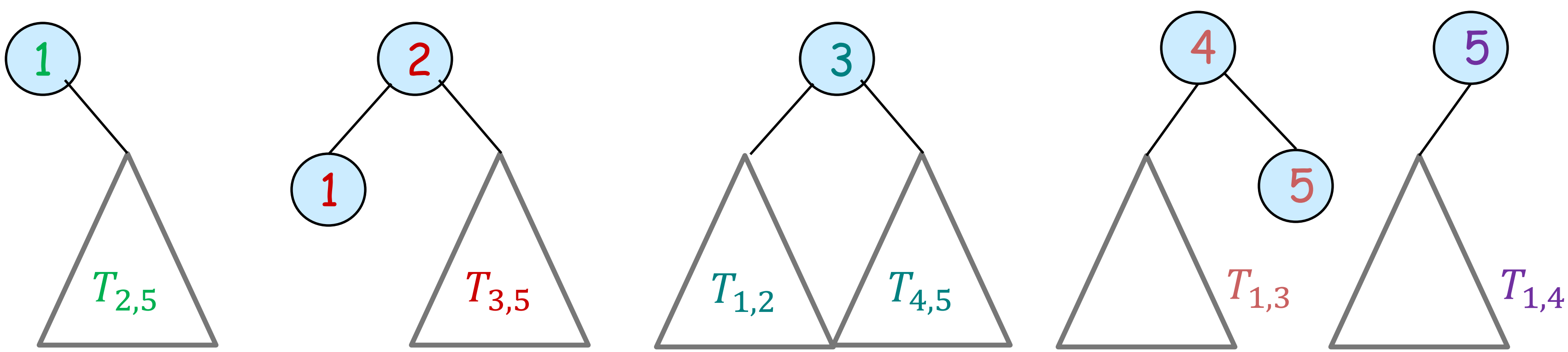


Optimal Binary Search Tree $\Theta\,(n^3)$ - Cont

Recurrence: $e[i,j] = \min_{i \le k \le j} \{e[i,k-1] + e[k+1,j] + w[i,j]\}$

- $e[1{,}5] = \min_{i \le k \le j} \{e[1{,}0] + e[2{,}5], e[1{,}1] + e[3{,}5], e[1{,}2] + e[4{,}5], e[1{,}3] + e[5{,}5], e[1{,}4] + e[6{,}5]\}$

	1	2	3	4	5
1		$e[1{,}2]$	$e[1{,}3]$	$e[1{,}4]$	$e[1{,}5]$
2					$e[2{,}5]$
3					$e[3{,}5]$
4					$e[4{,}5]$
5					



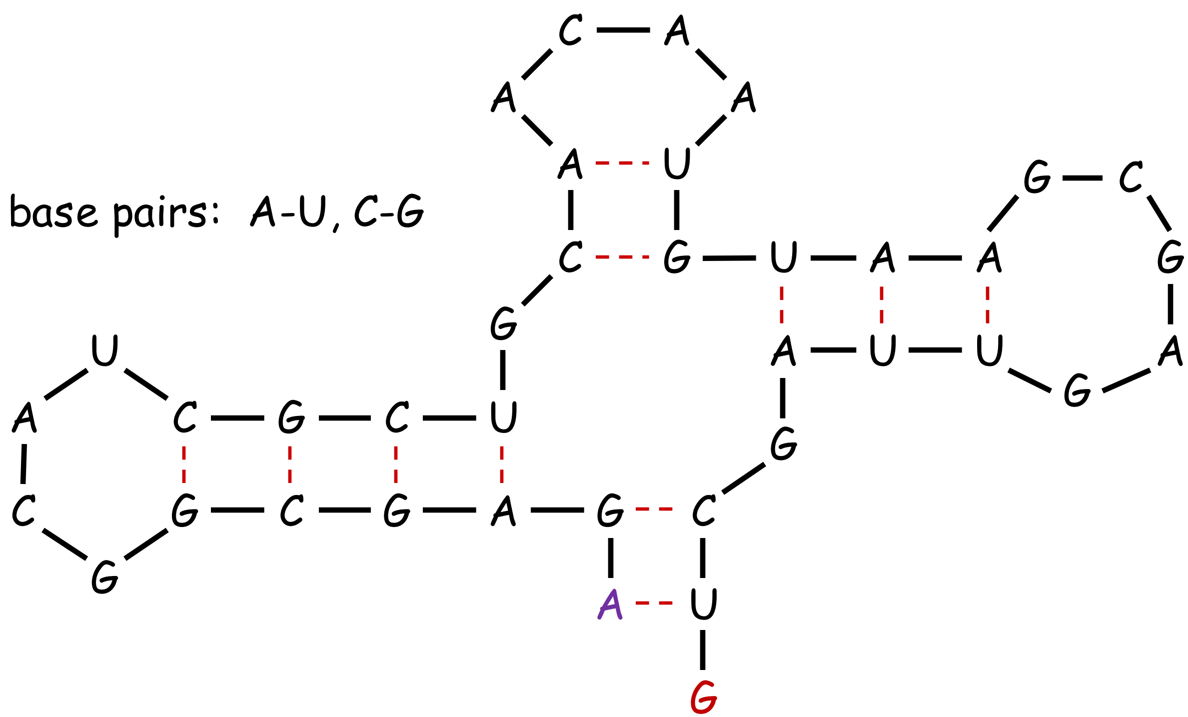
RNA Secondary Structure

RNA. String $B = b_1 b_2 \ldots b_n$ over alphabet { A, C, G, U }.

Secondary structure. RNA is single-stranded so it tends to loop back and form base pairs with itself. This structure is essential for understanding the behavior of molecules.

Ex: **GUCGAUUGAGCGAAUGUAACAACGUGGCUACGGCGAGA**

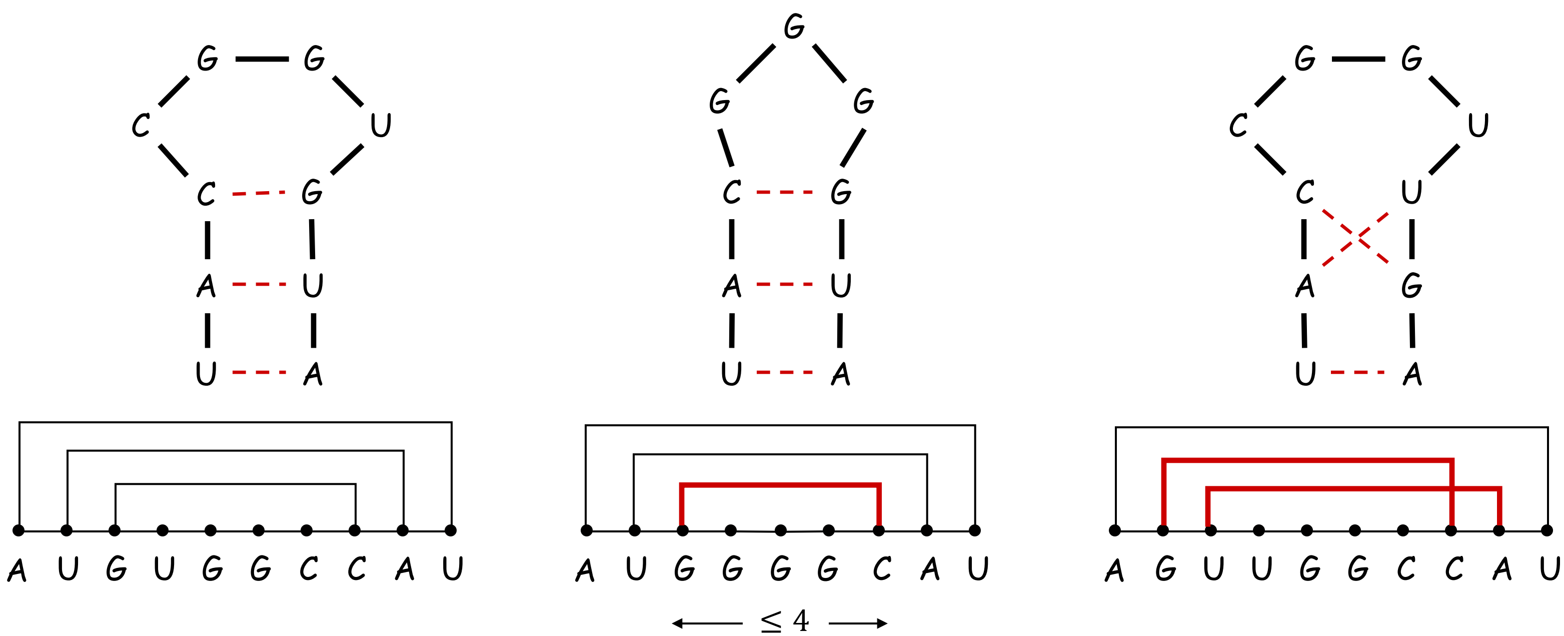
complementary base pairs: A-U, C-G



RNA Secondary Structure

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 complement: 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 $i < k < j < l$.



ok sharp turn crossing

The Problem

Free energy. Usual hypothesis is that an RNA molecule will form the secondary structure with the optimum total free energy, which is proportional to the number of base pairs.

Goal. Given an RNA molecule $B = b_1 b_2 \ldots b_n$, find a secondary structure S that maximizes the number of base pairs.

That is, find the maximum number of base pairs that can be matched satisfying the no sharp turns and no crossing constraints

The Recurrence

Def. $M[i, j]$ = maximum number of base pairs in a secondary structure of the substring $b_i b_{i+1} \dots b_j$.

Recurrence.

- Case 1. If $i \geq j - 4$.
  - $M[i, j] = 0$ by no-sharp turns condition.

- Case 2. $i < j - 1$
  - Case 2a: Base b_j is not matched in optimal solution for $[i, j]$
    - $M[i, j] = M[i, j-1]$
  - Case 2b: Base b_j pairs with b_k for some $i \leq k \leq j - 5$.
    - Try matching b_j to all possible b_k.
    - non-crossing constraint decouples problem into sub-problems
    - $M[i, j] = 1 + \max_k \{M[i, k-1] + M[k+1, j-1]\}$

Take max over k such that $i \leq k \leq j - 5$ and b_k and b_j are Watson-Crick complements

The Algorithm

```
let M[1..n, 1..n], s[1..n, 1..n] be new arrays of all 0
for l ← 1 to n
    for i ← 1 to n − l + 1
        j ← i + l − 1
        M[i, j] ← M[i, j − 1]
        for k ← i to j − 5
            if b_k and b_j are not complements then continue
            t ← 1 + M[i, k − 1] + M[k + 1, j − 1]
            if t > M[i, j] then
                M[i, j] ← t
                s[i, j] ← k
Construct-RNA(s, 1, n)
```

Running time: $O(n^3)$

Space: $O(n^2)$

```
Construct-RNA(s, i, j):
if i ≥ j − 4 then return
if s[i, j] = 0 then Construct-RNA(s, i, j − 1)
print s[i, j], "-", j
Construct-RNA(s, i, s[i, j] − 1)
Construct-RNA(s, s[i, j] + 1, j − 1)
```