Dynamic Programming II
DP over Intervals

DP Over Intervals

All of the problems in this lecture share the following structural properties (similar to Matrix-Chain Multiplication)

- Goal is to find optimal (min or max) solution on problem with
  - Problem of size n
  - *ordered* input of items 1,2…,n

- Define substructure as
  - Ordered input of items i..j
  - Problem of size j-i+1

- Recurrence gives optimal solution of subproblem
  as function of optimal solution of smaller subproblems

- Algorithm fills in DP table from smallest to largest problem size

- Often, final subproblem filled is solution for original problem
  Sometimes, solution of original problem is min/max over table values

Longest Palindromic Substring

Def: A palindrome is a string that reads the same backward or forward.

Ex:

- **radar, level, racecar, madam**
- "A man, a plan, a canal – Panama!" (ignoring space, punctuation, etc.)

Problem: Given a string $X = x_1 x_2 \dots x_n$, find the longest palindromic substring.

Ex:

- $X = $ **ACCABA**
- Palindromic substrings: **ACCA, ABA**
- Longest palindromic substring: **ACCA**

Note:

- Brute-force algorithm takes $O(n^3)$ time.
- Recall: A substring must be contiguous

Dynamic Programming Solution

Def: Let $p[i,j]$ be $true$ iff $X[i..j]$ is a palindrome.

The Recurrence:

Initial Conditions (subproblems of sizes 1 & 2)

- $p[i,i] = true$, **for all** i
  - **A**CBBCABA

- $p[i,i+1] = true$ **if** $x_i = x_{i+1}$
  - **ACBBCABA**

The Actual Recurrence

- $p[i,j] = true$
  - if $x_i = x_j$ **AND** $p[i+1,j-1] = true$
    - **ACBBCABA**
    - **ACBBCABA**

A Completed DP Table

i	1	2	3	4	5	6	7	8
	B	A	B	B	C	C	C	B

Initial Condition
j=i; j=i+1

i/j	1	2	3	4	5	6	7	8
1	T	F						
2		T	F					
3			T	T				
4				T	F			
5					T	T		
6						T	T	
7							T	F
8								T

j=i+2

i/j	1	2	3	4	5	6	7	8
1	T	F	T					
2		T	F	F				
3			T	T	F			
4				T	F	F		
5					T	T	T	
6						T	T	F
7							T	F
8								T

j=i+3

i/j	1	2	3	4	5	6	7	8
1	T	F	T	F				
2		T	F	F	F			
3			T	T	F	F		
4				T	F	F	F	
5					T	T	T	F
6						T	T	F
7							T	F
8								T

j=i+4

i/j	1	2	3	4	5	6	7	8
1	T	F	T	F	F			
2		T	F	F	F	F		
3			T	T	F	F	F	
4				T	F	F	F	T
5					T	T	T	F
6						T	T	F
7							T	F
8								T

j>i+4

i/j	1	2	3	4	5	6	7	8
1	T	F	T	F	F	F	F	F
2		T	F	F	F	F	F	F
3			T	T	F	F	F	F
4				T	F	F	F	T
5					T	T	T	F
6						T	T	F
7							T	F
8								T

Largest is BCCCB

5
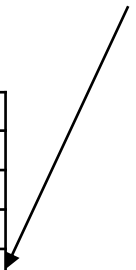
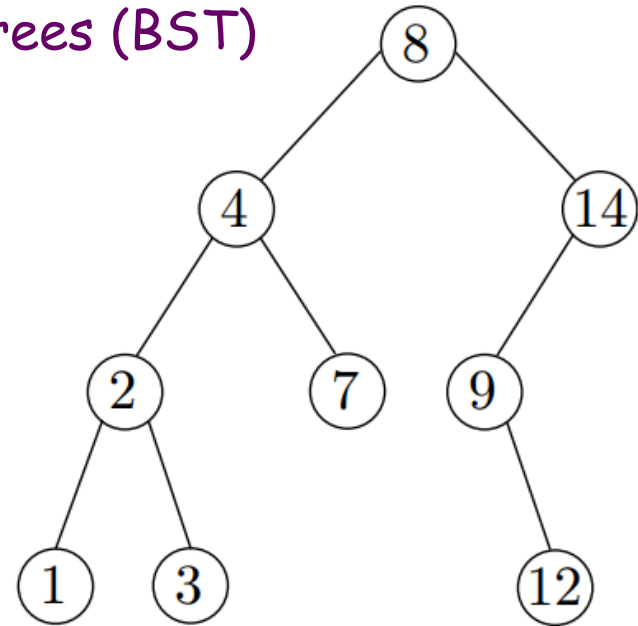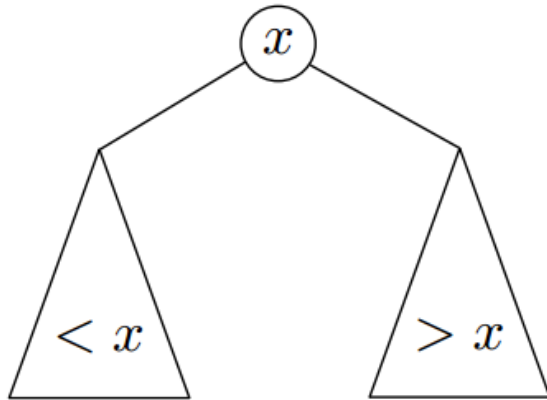# The Algorithm

```
max ← 1
for i ← 1 to n − 1 do                    initial conditions
    p[i, i] ← true                       j=i
    if xᵢ = xᵢ₊₁ then
        p[i, i + 1] ← true, max ← 2      j=i+1
    else p[i, i + 1] ← false
for l ← 3 to n do
    for i ← 1 to n − l + 1 do            j=i + (l-1)
        j ← i + l − 1
        if p[i + 1, j − 1] = true and xᵢ = xⱼ then
            p[i, j] ← true, max ← l
        else p[i, j] ← false
return max
```

Running time: $O(n^2)$

Space: $O(n^2)$ but can be improved to $O(n)$

Binary search trees (BST)


```
Tree-Search(T,k):
x ← T.root
while  x ≠ nil  and  k ≠ x.key  do
    if  k < x.key  then  x ← x.left
    else  x ← x.right
return  x
```

The (worst-case) search time in a balanced BST is $\Theta(\log n)$

Q: If we know the probability of each key being searched for, can we design a (possibly unbalanced) BST to optimize the expected search time?

The Optimal Binary Search Tree Problem

Problem Definition (simpler than the version in textbook):

Given n keys $a_1 < a_2 < \cdots < a_n$, with weights $f(a_1), \ldots, f(a_n)$, find a binary search tree T on these n keys such that

$$B(T) = \sum_{i=1}^{n} f(a_i)(d(a_i) + 1)$$

is minimized, where $d(a_i)$ is the depth of a_i.

Note: This is similar to the Huffman coding problem but with 2 key differences:

- The tree has to be a BST, i.e., the keys are stored in sorted order. In a Huffman tree, there is no ordering among the leaves.
- Keys appear as both internal and leaf nodes. In a Huffman tree, keys (characters) appear only at the leaf nodes.

Motivation: If the weights are the probabilities of the elements being searched for, then such a BST will minimize the expected search cost.

Greedy Won't Work

Greedy strategy: Always pick the heaviest key as root, then recursively build the tree top-down.
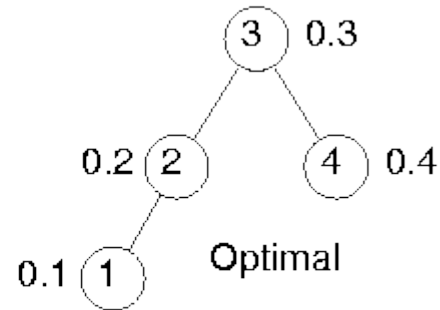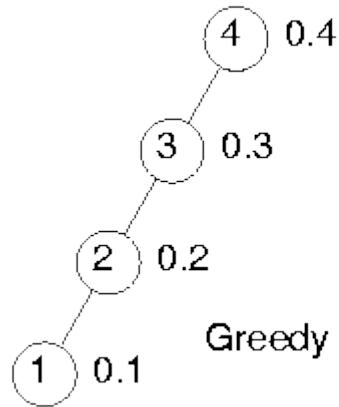
$$B(T) = 0.4 \cdot 2 + 0.3 \cdot 1 + 0.2 \cdot 2 + 0.1 \cdot 3 = 1.8$$

$$B(T) = 0.4 \cdot 1 + 0.3 \cdot 2 + 0.2 \cdot 3 + 0.1 \cdot 4 = 2$$

Dynamic Programming: The Recurrence

Def: $e[i,j]$ = the minimum cost of any BST on $a_i, \ldots, a_j$

Idea: The root of the BST can be any of $a_i, \ldots, a_j$. We try each of them.
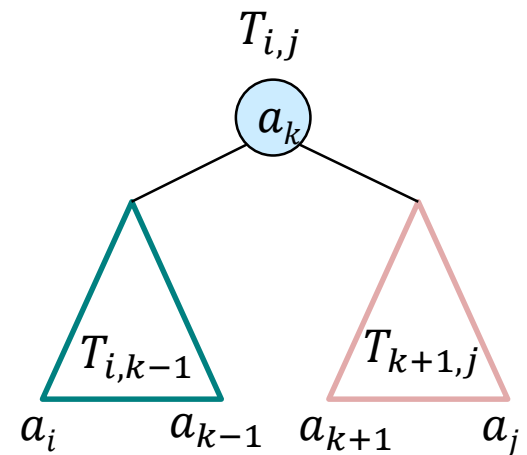
Recurrence:
Let $w[i,j] = f(a_i) + \cdots + f(a_j)$

Suppose we knew min-cost BST $T_{i,j}$ for [i,j] and that its root was a_k.

Its left subtree $T_{i,k-1}$ must be optimal for [i,k-1]
and its right subtree $T_{i,k-1}$ must be optimal for [k+1,j]

Nodes in $T_{i,k-1}$ and $T_{i,k-1}$ are one level deeper in $T_{i,j}$ than in their original trees. So the cost of the new tree is

$e[i, k-1] + w[i, k-1] + e[k+1, j] + w[k+1, j] + f(a_k)$

$= e[i, k-1] + e[k+1, j] + w[i,j]$

$T_{i,j}$

We try out every possible value of k and return the one which minimizes cost!

10

Dynamic Programming: The Recurrence

Def: $e[i,j]$ = the minimum cost of any BST on $a_i, \ldots, a_j$

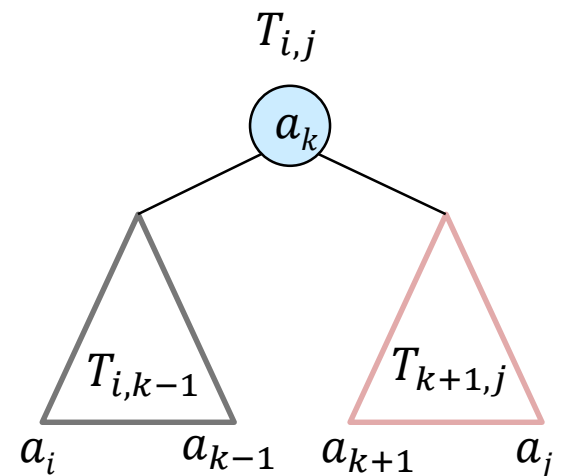Idea: The root of the BST can be any of $a_i, \ldots, a_j$. We try each of them.

Recurrence:

Let $w[i,j] = f(a_i) + \cdots + f(a_j)$

$$e[i,j] = \min_{i \leq k \leq j}\{e[i,k-1] + w[i,k-1] + e[k+1,j] + w[k+1,j] + f(a_k)\}$$

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

$e[i,j] = 0$ for $i > j$.

Note: All $w[i,j]$'s can be pre-computed in $O(n^2)$ time.

$T_{i,j}$

a_k

$T_{i,k-1}$

$T_{k+1,j}$

$a_i \qquad a_{k-1} \quad a_{k+1} \qquad a_j$

The Algorithm

Idea: We will do the bottom-up computation by the increasing order of the problem size.

```
let e[1..n, 1..n], w[1..n, 1..n], root[1..n, 1..n] be new arrays of all 0
for i = 1 to n
    w[i, i] ← f(aᵢ)
    for j = i + 1 to n
        w[i, j] ← w[i, j − 1] + f(aⱼ)
for l ← 1 to n                              length of [i,j]
    for i ← 1 to n − l + 1
        j ← i + l − 1
        e[i, j] ← ∞
        for k ← i to j                      Find k that
            t ← e[i, k − 1] + e[k + 1, j] + w[i, j]   minimizes
            if t < e[i, j] then             e[i,k−1]+e[k+1,j]+w[i,j]
                e[i, j] ← t
                root[i, j] ← k
return Construct-BST(root, 1, n)
```

Running time: $O(n^3)$

Space: $O(n^2)$

Construct the Optimal BST

```
Construct-BST(root, i, j):
if i > j then return nil
create a node z
z.key ← a[root[i, j]]
z.left ← Construct-BST(root, i, root[i, j] − 1)
z.right ← Construct-BST(root, root[i, j] + 1, j)
return z
```

Running time of this part: $O(n)$

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
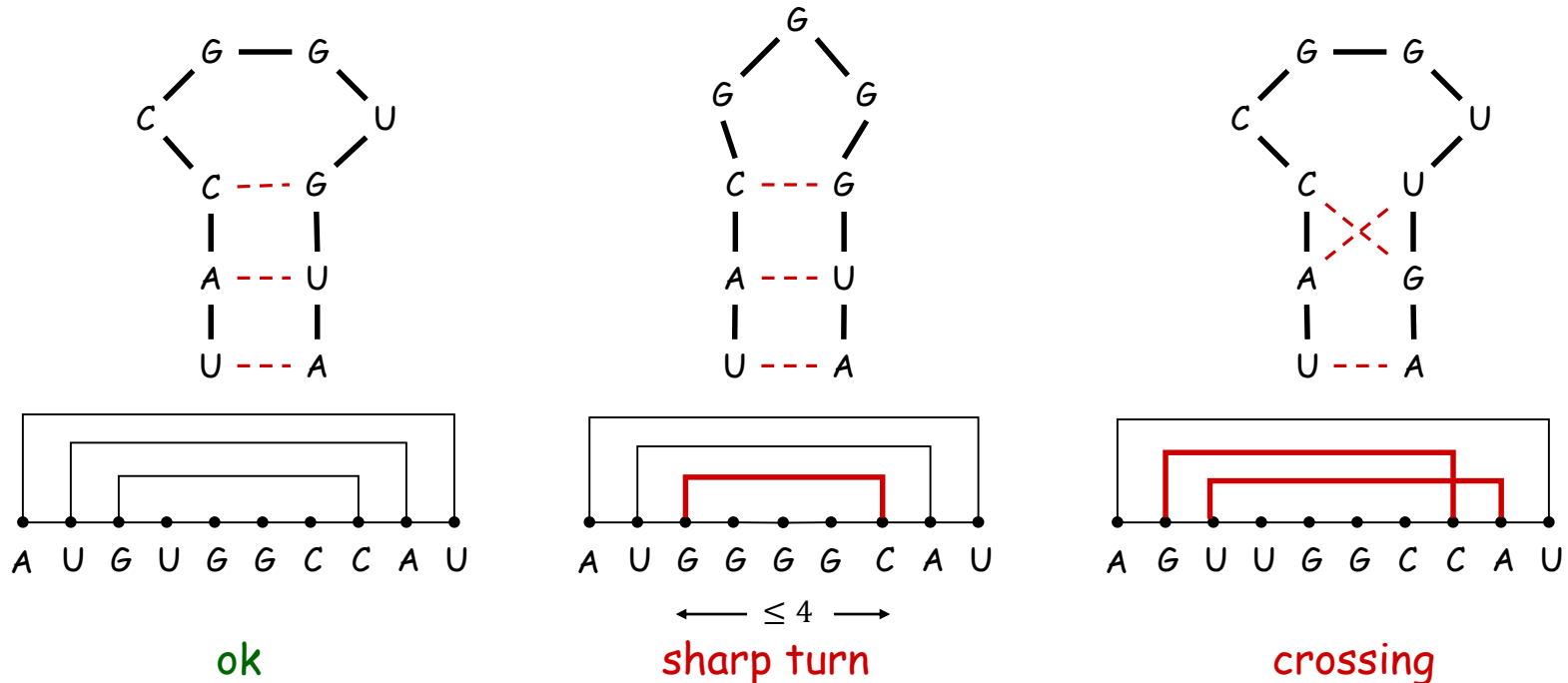
The Problem

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 $B = b_1 b_2 ... b_n$, 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 $b_i b_{i+1} \ldots b_j$.

Recurrence.

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

- Case 2. i< j-4

  - 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)
```