RNA || MAPPING

$\Sigma = \{A, U, C, G\}$

Σ^* base case : $\varepsilon \in \Sigma^*$

Recursive: If $w \in \Sigma^*$

then $w\sigma \in \Sigma^*$

where $\sigma \in \Sigma$

$\Sigma^* = \{\varepsilon, A, U, C, G, AA, AG, AU, \ldots\}$.

RNA Folding

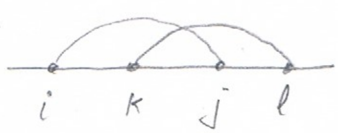
① → Mapping ⇒ $A \leftrightarrow U$, $C \leftrightarrow G$

② → if we match positions i & j, then $i < j - 4$.

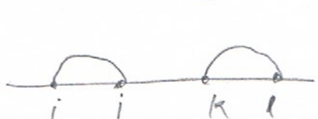
i.e. No sharp turns

③ → if (i, j) is a matching pair of positions, then i & j do not participate in any other pair

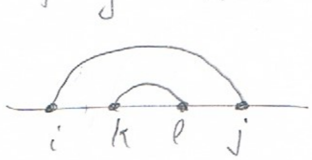
④ → if (i, j) & (k, l) are two matching pairs, then



⇒ $i < k < j < l$ ⇒ X not Allowed.



⇒ $i < j < k < l$ ⇒ ✓ Allowed



⇒ $i < k < l < j$ ⇒ ✓ Allowed

$B = b_1 b_2 b_3 b_4 \ldots b_n$ $b_i \in \{A, U, C, G\}$

$1 \; 2 \; 3 \; 4 \ldots n$

$OPT(1, n)$ = Maximum number of pairs, of indices that I can match together to satisfy the four feasible requirements in the feasibility of the problem.

case 1: position n is not part of the solution

$$OPT(1, n) = OPT(1, n-1)$$

case 2: position n pairs with some position t

$$OPT(1, n) = 1 + OPT(1, t-1) + OPT(t+1, n-1).$$

where; $1 \le t < n-4$

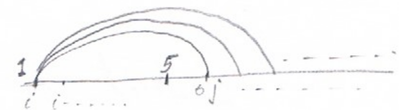
& b_t, b_n satisfy complementary

$$OPT(i,j) = \max\Big\{\underbrace{OPT(i, j-1)}_{\text{case 1, } j \text{ is not part of solution}},\ \underbrace{\max_t\big(1 + OPT(i, t-1) + OPT(t+1, j-1)\big)}_{\text{case 2, } j \text{ is part of solution paired with } t.}\Big\}$$

max of these two makes $OPT(i,j)$.

Dynamic Programming Algorithm

(1) Initialize $OPT(i,j) \leftarrow 0$ whenever $i \geq j-4$;



(2) For $k \leftarrow 5, 6, 7, \ldots, n-1$

(3) For $i \leftarrow 1, 2, \ldots, n-k$

(4) $j \leftarrow i+k$;

(5) $OPT(i,j) = \max\{OPT(i,j-1), \max_t (1 + OPT(i,t-1) + OPT(t+1,j-1))\}$.

(6) Return $OPT(1,n)$;

Eg. → I/p [A C C G G U A G U]
1 2 3 4 5 6 7 8 9.

OPT

line1 line1 line1

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

2 → Ans

1st loop ⇒ $i = 1$, $j = 6$

$OPT(1,6) = \max\{\underbrace{OPT(1,5)}_{0}, \max_t (1 + OPT(1,t-1) + OPT(t+1,5$

Big-O, Running Time of DP Algo.

line 2 → $O(n)$
line 3 → $O(n)$
line 5 → $O(n)$

} → $O(n^3)$ ⇒ Way better than Brut Force Algo.

Recursion vs Dynamic Programming

Problem: → Given set of n, $\binom{n}{k}$ is number of ways of chosing a subset of size k from a set of size N.

$\{1, 2, \ldots n\}$ ⇒ $\binom{n}{k}$

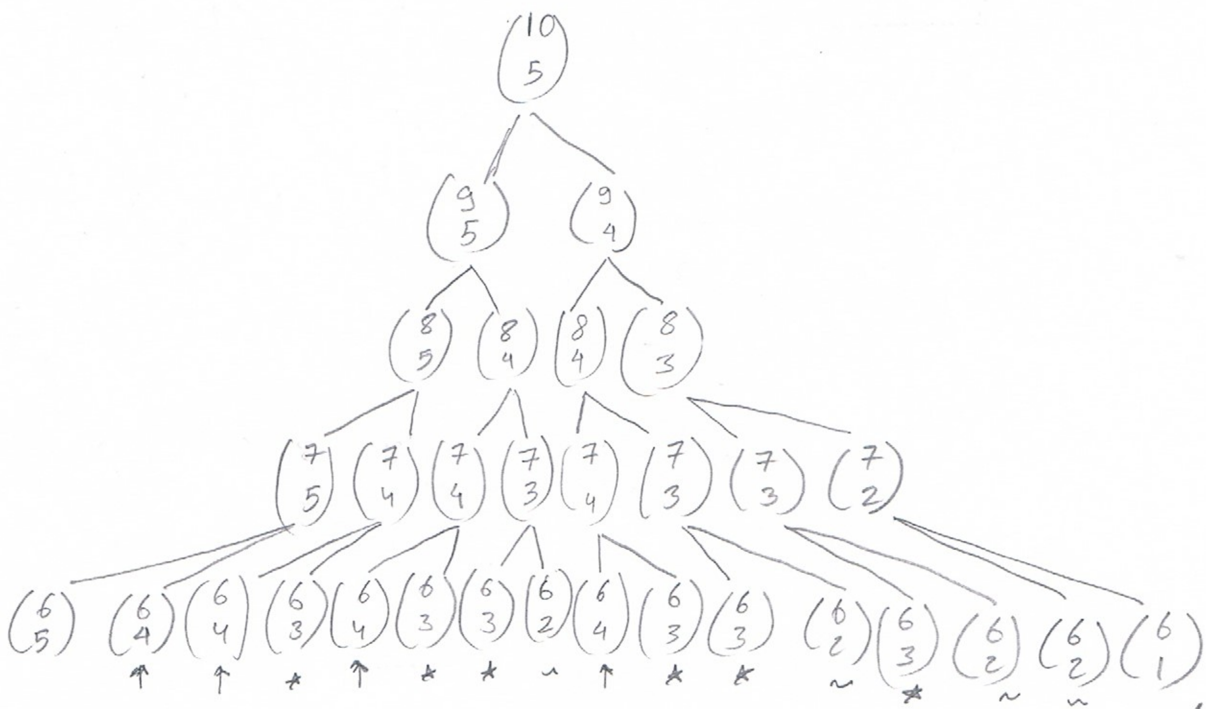
$n \in$	$n \notin$	→ (n belong or not in
$\{\ldots, n\}$	$\{\ldots\}$	⇒ subset)
$k-1$	k	⇒ now we have to chose this number
$\{1, \ldots, n-1\}$	$\{1, \ldots, n-1)$	⇒ from this set
⇓	⇓	
$\binom{n-1}{k-1}$	$\binom{n-1}{k}$	⇒ ways of chosing now

Thus, → $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$

Base Cases ⇒ $\binom{n}{0} = 1$, $\binom{0}{k} = 0$.

Algo ⇒

1. Compute N choseK (n, k)
2. If $n \geq 0$ & $k = 0$
3. └ return 1
4. else if $n = 0$ & $k > 0$
5. └ return 0
6. else
7. └ Return Compute N Chose K (n-1, k) + Compute N chose K (n-1, k-1)



Thus for similar sub-problems there is lot of computational repetition which icreases the time.

Both DP & recursive has some recursive calls underneath but DP keeps a copy of already computed sub-problem & doesn't compute it again.