CMPSC 465
Data Structures and Algorithms
Fall 2024

Instructor: Debarati Das

November 5, 2024

Dynamic Programming

Dynamic Programming

Prelude

Dynamic programming vs. Greedy algorithms

- Similarity: optimal substructure
- Difference: greedy choice property

A greedy algorithm makes the greedy choice and it leaves a subproblem to solve

Sometimes, the greedy choice won't work — we need to check many subproblems to find the optimal solution $\rightarrow$ **Dynamic programming**

General steps for Dynamic Programming

- Break problem into overlapping smaller subproblems
- Solve smaller subproblems first (**bottom-up**)
- Use information from smaller subproblems to solve a larger subproblem
- Unlike greedy and divide and conquer, combining the subproblems is more non-trivial.

Dynamic Programming

Edit Distance (Textbook Section 6.3)

Edit distance

Motivation: consider DNA sequences $x = ACGTA$, $y = ATCTG$.

Note $|x| \neq |y|$ in general

Question: how far away are x and y?

Definition

The **edit distance** between x and y, denoted by $d(x, y)$, is the minimum number of character insertions, deletions, and substitutions needed to transform x to y

Consider the following **alignments**:

Edit distance

Motivation: consider DNA sequences $x = ACGTA$, $y = ATCTG$.

Note $|x| \neq |y|$ in general

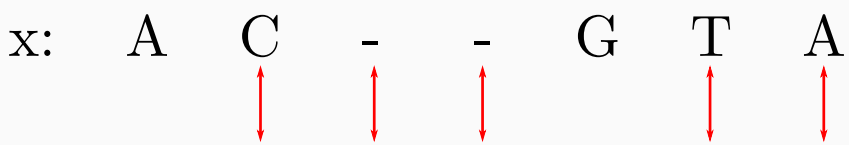
Question: how far away are x and y?

Definition

The **edit distance** between x and y, denoted by $d(x, y)$, is the minimum number of character insertions, deletions, and substitutions needed to transform x to y

Consider the following **alignments**:

cost : 3 (optimal)



cost : 5

So $d(x, y) = 3$

Edit distance — subproblem

Consider two strings

$$x = x_1 x_2 \cdots x_m \quad \text{and} \quad y = y_1 y_2 \cdots y_n$$

Subproblem: consider prefix $x_1 \cdots x_i$ and $y_1 \cdots y_j$ $(i \leq m, j \leq n)$

Define

$$E(i, j) = d(x_1 \cdots x_i, y_1 \cdots y_j)$$

Optimal solution: $E(m, n)$

How to use the solution to the subproblems to solve $E(i, j)$?

Recurrence (I)

Look at the rightmost characters from the input strings:

Case 1

$$\begin{matrix} x_1 & \cdots & x_{i-1} & x_i \\ y_1 & \cdots & y_j & - \end{matrix}$$

Contributes 1 to the cost plus the cost of alignment $\begin{matrix} x_1 & \cdots & x_{i-1} \\ y_1 & \cdots & y_j \end{matrix}$

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

Case 2

$$\begin{matrix} x_1 & \cdots & x_i & - \\ y_1 & \cdots & y_{j-1} & y_j \end{matrix}$$

Contributes 1 to the cost plus the cost of alignment $\begin{matrix} x_1 & \cdots & x_i \\ y_1 & \cdots & y_{j-1} \end{matrix}$

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

Case 3

$$\begin{matrix} x_1 & \cdots & x_{i-1} & x_i \\ y_1 & \cdots & y_{j-1} & y_j \end{matrix}$$

$$E(i,j) = \begin{cases} E(i-1,j-1) & \text{if } x_i = y_j \\ 1 + E(i-1,j-1) & \text{otherwise} \end{cases}$$

Recurrence (II)

The recurrence:

$$E(i,j) = \min\{1 + E(i-1,j), 1 + E(i,j-1), \mathrm{diff}(i,j) + E(i-1,j-1)\},$$

where

$$\mathrm{diff}(i,j) = \begin{cases} 1 & \text{if } x_i \neq y_j \\ 0 & \text{otherwise} \end{cases}$$

Optimal solution: $E(m,n)$

Base case: $E(0,0) = 0$, $E(i,0) = i$, $E(0,j) = j$

Filling the table

$$E(i,j) = \min\{1 + E(i-1,j), 1 + E(i,j-1), \mathrm{diff}(i,j) + E(i-1,j-1)\},$$

$$\begin{array}{ccccccc}
E(0,0) & & E(0,1) & \cdots & E(0,n-1) & & E(0,n) \\
 & \searrow & \downarrow & & & & \\
E(1,0) & \rightarrow & E(1,1) & \cdots & \cdots & & \cdots \\
\vdots & & & & & & \\
 & & & \searrow & \downarrow & \searrow & \downarrow \\
E(m-1,0) & & & \rightarrow & E(m-1,n-1) & \rightarrow & E(m-1,n) \\
 & & & \searrow & \downarrow & \searrow & \downarrow \\
E(m,0) & & & \rightarrow & E(m,n-1) & \rightarrow & E(m,n)
\end{array}$$

Running example

$x = ACGTA$ and $y = ATCTG$

		A	T	C	T	G
	0	1	2	3	4	5
A	1	0	1	2	3	4
C	2	1	1	1	2	3
G	3	2	2	2	2	2
T	4	3	2	3	2	3
A	5	4	3	3	3	3

Pseudocode

def EDIT_DISTANCE(x, y)**:**

- **for** $i = 0, \ldots, m$**:**
  - $E(i, 0) = i$;
- **for** $j = 0, \ldots, n$**:**
  - $E(0, j) = j$;
- **for** $i = 1, \ldots, m$**:**
  - **for** $j = 1, \ldots, n$**:**
    - $E(i, j) =$
      $\min\{1 + E(i-1, j), 1 + E(i, j-1), \text{diff}(i, j) + E(i-1, j-1)\}$;
- **return** $E(m, n)$;

Running time: $O(mn)$

Finding the alignment

We use an extra table prev to record where each entry of $E(i,j)$ was coming from:

$$\mathrm{prev}(i,j) = \begin{cases} (i-1,j) & \text{if } E(i,j) = 1 + E(i-1,j) \\ (i,j-1) & \text{if } E(i,j) = 1 + E(i,j-1) \\ (i-1,j-1) & \text{if } E(i,j) = \mathrm{diff}(i,j) + E(i-1,j-1) \end{cases}$$

def PRING_ALIGNMENT(x, y, prev)**:**

- Set $i = m, j = n$;
- **while** $i \geq 1$ *and* $j \geq 1$**:**
  - **if** $\mathrm{prev}(i,j) = (i-1, j-1)$**:**
    - $\mathrm{print\_back}\binom{y_i}{x_i}$;
    - $i = i - 1$, $j = j - 1$;
  - **if** $\mathrm{prev}(i,j) = (i-1, j)$**:**
    - $\mathrm{print\_back}\binom{-}{x_i}$;
    - $i = i - 1$;
  - **if** $\mathrm{prev}(i,j) = (i, j-)$**:**
    - $\mathrm{print\_back}\binom{y_j}{-}$;
    - $j = j - 1$;

Dynamic Programming

0-1 Knapsack (Textbook Section 6.4)

0-1 Knapsack

0-1 Knapsack Problem

A Thief has a backpack with capacity W. There is a set of n items. Item i has weight $w_i > 0$ and value $v_i > 0$. **Goal:** pack the backpack with the largest value.

- **Application:** assigning jobs to a machine to maximize the total value where the total processing capacity of the machine is bounded.

Subproblem

- **Idea 1** (Greedy Algorithm): repeatedly add item with max $\frac{v_i}{w_i}$ value.

Item (i)	Weight (w_i)	Value (v_i)	$\frac{v_i}{w_i}$
1	1	1	1
2	2	6	3
3	5	18	3.6
4	6	22	3.66
5	7	28	4

  - Total capacity $W = 11$
  - Algorithm picks items 5, 2, 1. Total value: 35
  - Optimal items 3,4. Total value 40

Subproblem

- **Idea 2** (DP with one variable):
  - **Subproblem:** $K(i)$ = maximum profit value on items $\{1, 2, \ldots, i\}$.
  - **Cse1:** $K(i)$ does not include item i. $K(i) = K(i-1)$.
  - **Cse2:** $K(i)$ includes item i. However without knowing the current weight limit and weight of item i we can't decide whether we are allowed to add item i.

Subproblem

- **Idea 3** (DP with two variables):
  - **Subproblem**: $K(w, j)$ — the maximum value achievable using a backpack of capacity w and items $1, \ldots, j$
  - **Optimal solution:** $K(W, n)$
  - **Recurrence:**

$$K(w, j) = \max\{K(w - w_j, j - 1) + v_j, K(w, j - 1)\}$$

  - **Base case:** $K(0, j) = 0$ for all j and $K(w, 0) = 0$ for all w

Pseudocode

def KNAPSACK(W, w, v)**:**

- Set $K(0, j) = 0, K(w, 0) = 0$ for all j, w;
- **for** $j = 1, \ldots, n$**:**
  - **for** $w = 1, \ldots, W$**:**
    - **if** $w_j > w$**:**
      - $K(w, j) = K(w, j-1)$;
    - **else:**
      - $K(w, j) = \max\{K(w - w_j, j-1) + v_j, K(w, j-1)\}$;
- **return** $K(W, n)$;

Running time: $O(nW)$

Question: is this a polynomial-time algorithm? No!

Input size: $n \log max(w_1, \ldots, w_n, W)$. The time is polynomial in the largest integer involved. We call this a pseudopolynomial algorithm.

Running example

Example: $W = 10$

item	1	2	3	4
w_j	6	3	4	2
v_j	30	14	16	9

Running example

Example: $W = 10$

item	1	2	3	4
w_j	6	3	4	2
v_j	30	14	16	9

The K table:

$w \backslash j$	0	1	2	3	4
0	0	0	0	0	0
1					
2					
3					
4					
5					
6					
7					
8					
9					
10					

Running example

Example: $W = 10$

item	1	2	3	4
w_j	6	3	4	2
v_j	30	14	16	9

The K table:

$w \backslash j$	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	0
2	0	0	0	0	9
3	0	0	14	14	14
4	0	0	14	16	16
5	0	0	14	16	23
6	0	30	30	30	30
7	0	30	30	30	30
8	0	30	30	30	39
9	0	30	44	44	44
10	0	30	44	46	46

Running example

Example: $W = 10$

item	1	2	3	4
w_j	6	3	4	2
v_j	30	14	16	9

The K table:

$w \backslash j$	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	0
2	0	0	0	0	9
3	0	0	14	14	14
4	0	0	14	16	16
5	0	0	14	16	23
6	0	30	30	30	30
7	0	30	30	30	30
8	0	30	30	30	39
9	0	30	44	44	44
10	0	30	44	46	46

Running example

Example: $W = 10$

item	1	2	3	4
w_j	6	3	4	2
v_j	30	14	16	9

The K table:

$w \backslash j$	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	0
2	0	0	0	0	9
3	0	0	14	14	14
4	0	0	14	16	16
5	0	0	14	16	23
6	0	30	30	30	30
7	0	30	30	30	30
8	0	30	30	30	39
9	0	30	44	44	44
10	0	30	44	46	46

Running example

Example: $W = 10$

item	1	2	3	4
w_j	6	3	4	2
v_j	30	14	16	9

The K table:

$w \backslash j$	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	0
2	0	0	0	0	9
3	0	0	14	14	14
4	0	0	14	16	16
5	0	0	14	16	23
6	0	30	30	30	30
7	0	30	30	30	30
8	0	30	30	30	39
9	0	30	44	44	44
10	0	30	44	46	46

Running example

Example: $W = 10$

item	1	2	3	4
w_j	6	3	4	2
v_j	30	14	16	9

The K table:

$w \backslash j$	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	0
2	0	0	0	0	9
3	0	0	14	14	14
4	0	0	14	16	16
5	0	0	14	16	23
6	0	30	30	30	30
7	0	30	30	30	30
8	0	30	30	30	39
9	0	30	44	44	44
10	0	30	44	46	46

Running example

Example: $W = 10$

item	1	2	3	4
w_j	6	3	4	2
v_j	30	14	16	9

The K table:

$w \backslash j$	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	0
2	0	0	0	0	9
3	0	0	14	14	14
4	0	0	14	16	16
5	0	0	14	16	23
6	0	30	30	30	30
7	0	30	30	30	30
8	0	30	30	30	39
9	0	30	44	44	44
10	0	30	44	46	46

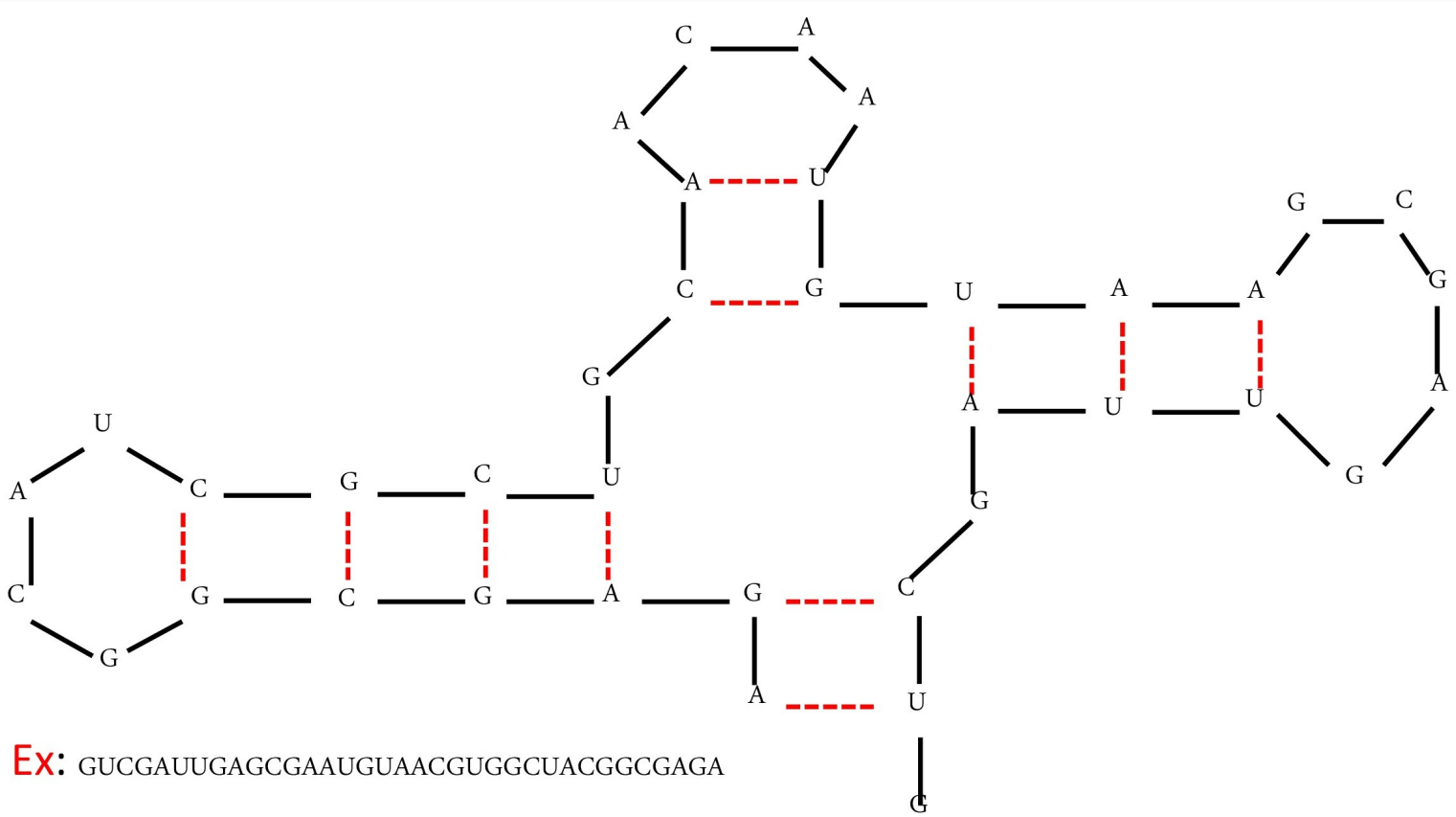
Dynamic Programming

RNA Secondary Structure (Kleinberg and Tardos Section 6.5)

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 behavior of the molecule.



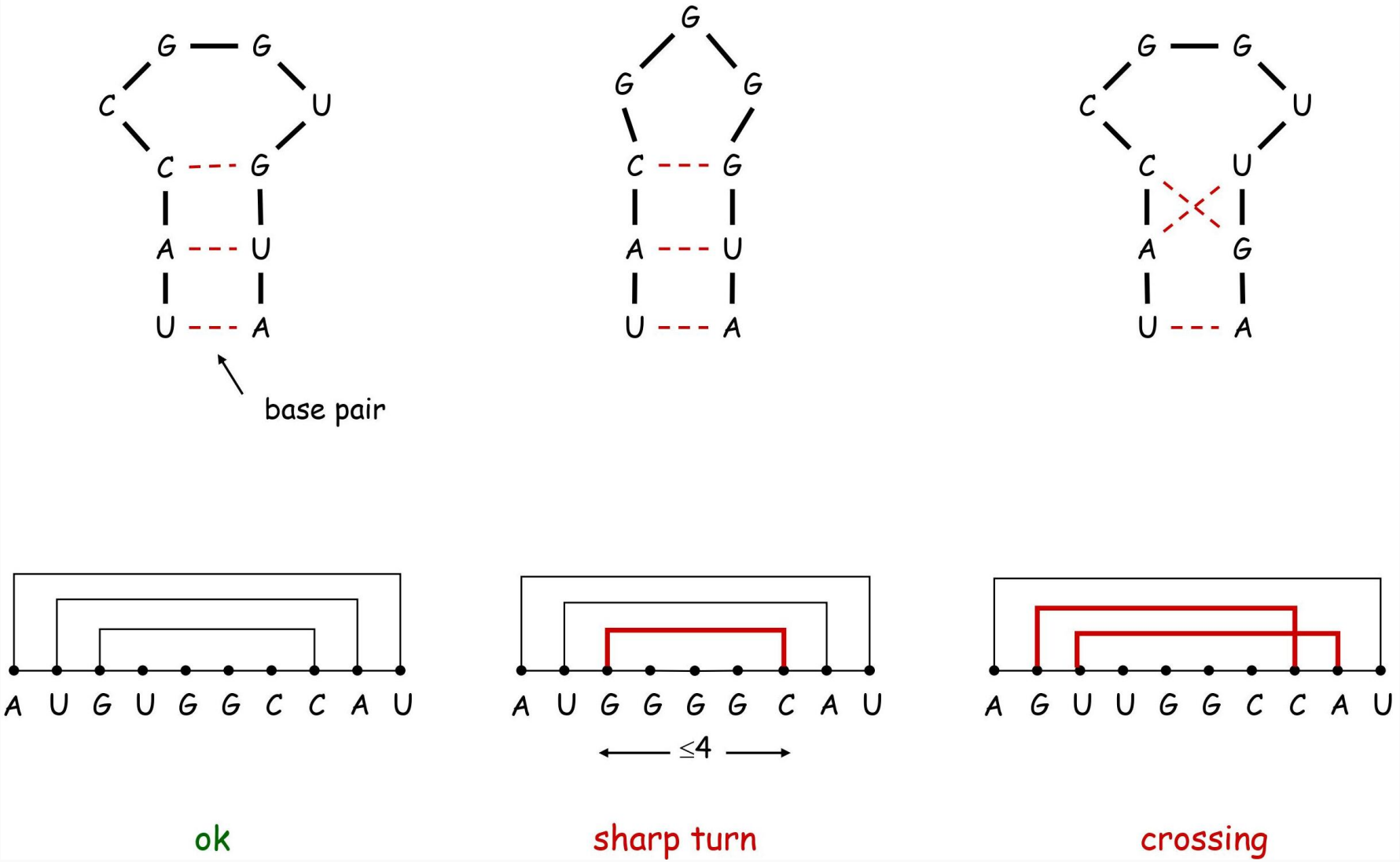
RNA Secondary Structure

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

- Watson-Crick. S is a matching and each pair in S is a *WatsonCrick 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 $i < j - 1$.
- Non Crossing. If (b_i, b_j) and (b_k, b_ℓ) are two pairs in S, then we cannot have $i < k < j < \ell$.

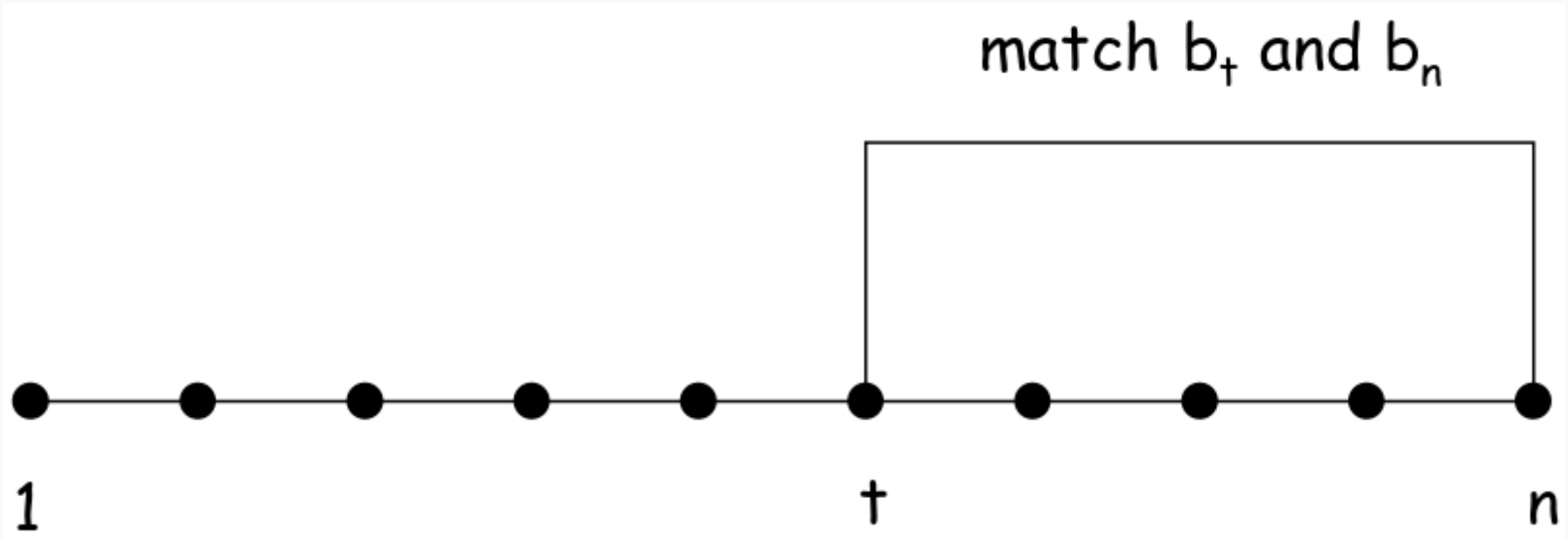
Goal: Given an RNA molecule $B = b_1 b_2 \dots b_n$, find a secondary structure S that maximizes the number of matching base pairs..

RNA Secondary Structure: Examples



RNA Secondary Structure: Subproblems

First attempt: $OPT(j)$ = maximum number of base pairs in a secondary structure of the substring $b_1 b_2 \ldots b_j$.



Difficulty: Results in two sub-problems.

- Finding secondary structure in: $b_1 b_2 \ldots b_{t-1}$. $(OPT(t-1))$
- Finding secondary structure in: $b_{t+1} b_{t+2} \ldots b_{n-1}$. (need more subproblems)

Dynamic Programming Over Intervals

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

Computing $OPT(i, j)$:

- Case 1. $i \geq j - 4$.

  by no-sharp turns condition we get

  $$OPT(i, j) = 0$$

- Case 2. Base b_j is not matched with any of $b_i b_{i+1} \dots b_{j-5}$.

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

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

  Using the non-crossing constraint we can decouple the resulting sub-problems.

$$OPT(i, j) = 1 + \max_{\substack{i \leq t < j-4 \\ b_t, b_j \text{ are wcc}}} \{OPT(i, t - 1) + OPT(t + 1, j - 1)\}$$

Pseudocode

Q. In what order should we solve the subproblems?

A. Do shortest intervals first.

The RNA Folding algorithm: $M[i, j]$ stores $OPT(i, j)$

def $\mathrm{RNA}(b_1 b_2 \ldots b_n)$**:**

- **for** $i = 1 \ldots n$**:**
  - **for** $j = 1 \ldots n$**:**
    - $M[i, j] = 0$;
- **for** $k = 5 \ldots n - 1$**:**
  - **for** $i = 1 \ldots n - k$**:**
    - $j = i + k$;
    - $M[i, j] = compute(b_i \ldots b_j)$;
- **return** $M[1, n]$;

Pseudocode

def COMPUTE($b_i \dots b_j$)**:**
    $M[i,j] = M[i,j-1]$;
    for $t = i \dots j-5$**:**
        if *If* b_t, b_j *are Watson-Crick complement*:
            $M[i,j] = \max\{M[i,j], 1 + M[i,t-1] + M[t+1,j]\}$;
    return $M[i,j]$;

Running time:

- Total number of sub-problems $O(n^2)$. Thus call Compute() $O(n^2)$ times.
- As there are $O(n)$ choices for t, each sub-problem can be solved in time $O(n)$.
- Total time: $O(n^3)$.

Dynamic Programming

All-pair shortest path (Textbook Section 6.6)

All-pair shortest path

Consider $G = (V, E)$ weighted, directed graph without negative cycles

How to compute the $\mathrm{shortest\_path}(u, v)$?

Recall Bellman-Ford: $\mathrm{shortest\_path}(u, v)$ for fixed u, all v takes $O(|V| \cdot |E|)$ time

If for all u, v, APSP takes $O(|V|^2|E|)$ time

When $|E| = O(|V|^2)$, its running time becomes $O(|V|^4)$

Rethink this problem using DP.

Subproblem

WLOG, index the vertices as $V = \{1, 2, \ldots, n\}$

Subproblem: find the shortest path $u \rightarrow v$ using intermediate vertices from $\{1, \ldots, k\} \subseteq V$.

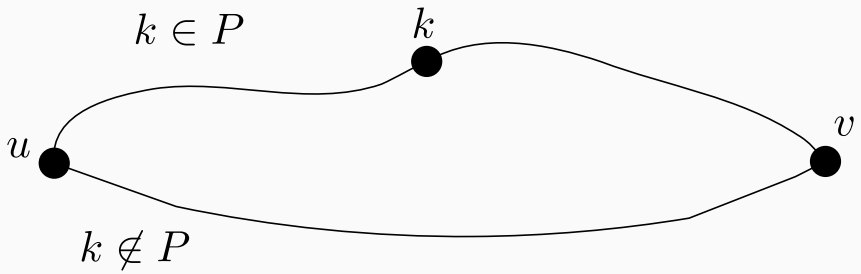
Denote it by $\mathrm{sp}(u, v, k)$

Optimal solution: the entries $\mathrm{sp}(u, v, n)$ for all u, v

To find out the recurrence relation, we need to relate $\mathrm{sp}(u, v, k)$ to smaller subproblems $\mathrm{sp}(u, v, k-1)$

Recurrence

Suppose $\mathrm{sp(u, v, k)} = P$



- if $k \notin P$, then $\mathrm{sp}(u, v, k) = \mathrm{sp}(u, v, k-1)$
- if $k \in P$, then consider

$$P: \quad u \xrightarrow{P_1} k \xrightarrow{P_2} v$$

P_1, P_2 are paths whose intermediate vertices are from $\{1, \ldots, k-1\}$. Because there's no negative cycles, there's no repeated vertices in shortest path

$$\text{Hence,} \quad P_1 = \mathrm{sp}(u, k, k-1), P_2 = \mathrm{sp}(k, v, k-1)$$

Using k is better if

$$|\mathrm{sp}(i, k, k-1)| + |\mathrm{sp}(k, v, k-1)| \leq |\mathrm{sp}(i, v, k-1)|$$

Dynamic programming

Let $\mathrm{dist}(u, v, k) = |\mathrm{sp}(u, v, k)|$

- **Recurrence:**

$$\mathrm{dist}(u, v, k) = \min\{\mathrm{dist}(u, v, k-1), \mathrm{dist}(u, k, k-1) + \mathrm{dist}(k, v, k-1)\}$$

- **Optimal solution:** $\mathrm{dist}(\cdot, \cdot, n)$
- **Base case:**

$$\mathrm{dist}(u, v, 0) = \begin{cases} w_{u,v} & \text{if } (u, v) \in E \\ \infty & \text{otherwise} \end{cases}$$

Pseudocode

The Floyd-Warshall algorithm:

def FLOYD_WARSHALL(G, w)**:**

- **for** $u = 1 \ldots n$**:**
  - **for** $v = 1 \ldots n$**:**
    - $\text{dist}(u, v, 0) = \begin{cases} w_{u,v} & \text{if } (u, v) \in E \\ \infty & \text{otherwise} \end{cases}$;
- **for** $k = 1 \ldots n$**:**
  - **for** $u = 1 \ldots n$**:**
    - **for** $v = 1 \ldots n$**:**
      - $\text{dist}(u, v, k) = \min\{\text{dist}(u, v, k-1), \text{dist}(u, k, k-1) + \text{dist}(k, v, k-1)\}$;
- **return** $\text{dist}(\cdot, \cdot, n)$;

Running time: $O(n^3) = O(|V|^3)$