CSE 6140/ CX 4140:

Computational Science and Engineering ALGORITHMS

Instructor: Anne Benoit

Visiting Associate Professor, CSE

Based on slides by Bistra Dilkina

Assignments and Midterm

- NO late assignments accepted. Please always set your internal deadline to the **Friday**, 2 weeks after HW released. Keep extra days for unexpected events, network problems, …
- Only 90/160 HW1 submitted Mon at 1pm! I expect 160/160 HW2 on Fri Sept. 29, and you can still resubmit until Sun 6pm
- Rule: Only 4 best HWs kept in final grade

- **MIDTERM (Oct. 5):** covers all we have seen so far + NPC (we will start on Thursday).
  - **Videos** now accessible for all students if you need to revise
  - **Help sessions** organized by TAs next week (Thu/Fri) instead of office hours, for practice problems; keep an eye on schedule
  - **Correction of HWs** posted on T-square for A sections, please do not discuss on Piazza, since DL students have one more week

Longest Common Subsequence (LCS)

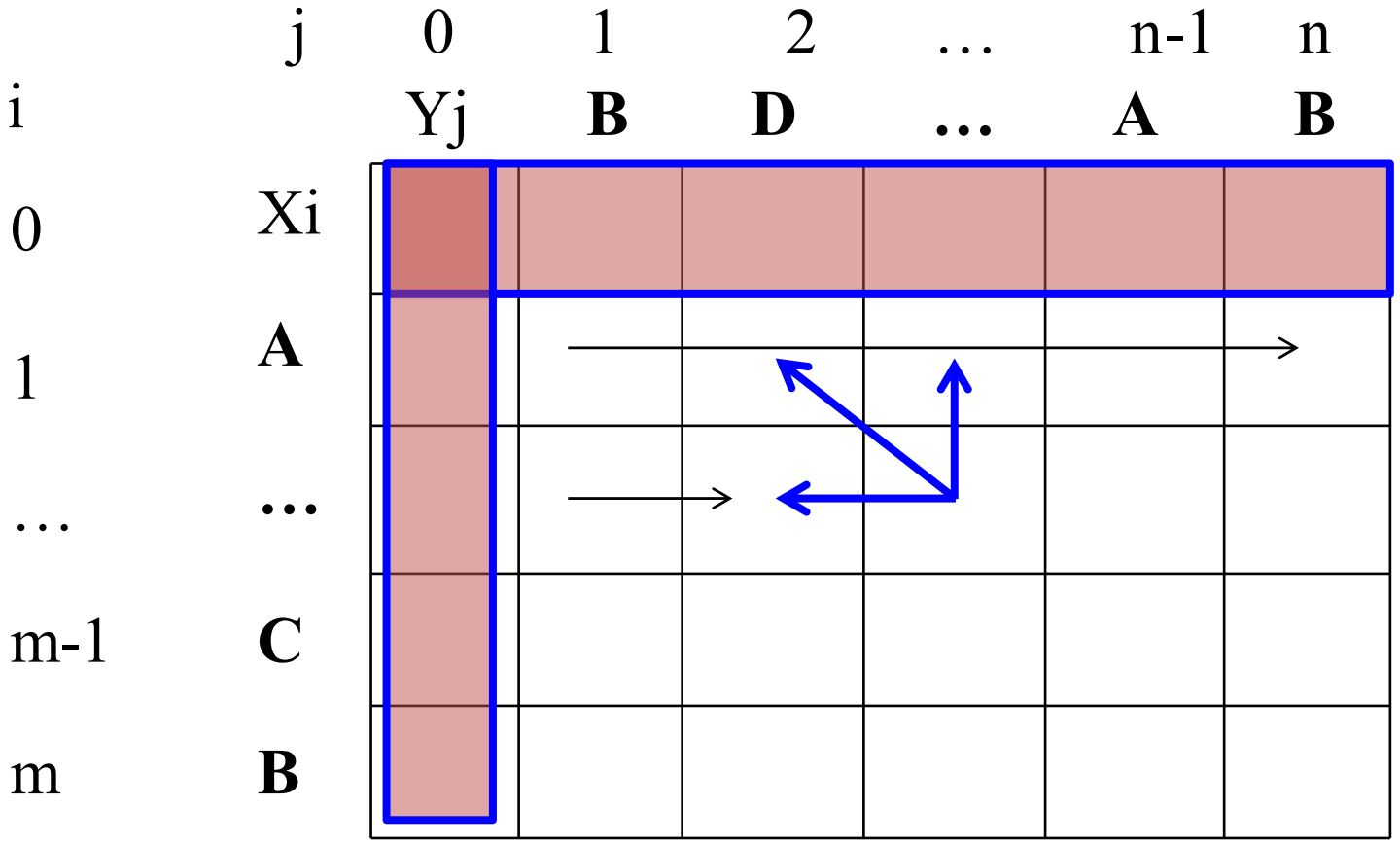
- Given two sequences/strings
- Example:
- X= {A B C B D A B }
- Y= {B D C A B A}

- find Longest Common Subsequence:
- X = A B C B D A B
- Y = B D C A B A

- Maximize number of matched pairs of characters

LCS subproblem order

$$c[i,j] = \begin{cases} c[i-1,j-1]+1 & \text{if } x_i = y_j \\ \max\{c[i-1,j], c[i,j-1]\} & \text{otherwise} \end{cases}$$



Allocate array c[m+1,n+1]

6.6 Sequence Alignment

String Similarity

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

6 mismatches, 1 gap

Allow to match characters that are not equal – incur a Mismatch cost

Allow gaps – incur a Gap cost

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

Applications.

- Basis for Unix diff.
- Speech recognition.
- Computational biology.

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

- Gap penalty δ; mismatch penalty α_{pq}.
- Cost = sum of gap and mismatch penalties.

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

$\alpha_{TC} + \alpha_{GT} + \alpha_{AG} + 2\alpha_{CA}$

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

$2\delta + \alpha_{CA}$

Sequence Alignment

Goal: Given two strings $X = x_1\, x_2 \ldots x_m$ and $Y = y_1\, y_2 \ldots y_n$ find alignment of minimum cost.

Def. An alignment M is a set of ordered pairs x_i-y_j such that each item occurs in at most one pair and no crossings.

Def. The pair x_i-y_j and $x_{i'}$-$y_{j'}$ cross if $i < i'$, but $j > j'$.

$$\text{cost}(M) = \underbrace{\sum_{(x_i, y_j) \in M} \alpha_{x_i y_j}}_{\text{mismatch}} + \underbrace{\sum_{i : x_i \text{ unmatched}} \delta + \sum_{j : y_j \text{ unmatched}} \delta}_{\text{gap}}$$

Ex: CTACCG vs. TACATG.

Sol: $M = x_2$-y_1, x_3-y_2, x_4-y_3, x_5-y_4, x_6-y_6.

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

Sequence Alignment: Problem Structure

Def. OPT(i, j) = min cost of aligning strings $x_1\, x_2 \ldots x_i$ and $y_1\, y_2 \ldots y_j$.

- Case 1: OPT matches x_i-y_j.
  - pay mismatch for x_i-y_j + min cost of aligning two strings $x_1\, x_2 \ldots x_{i-1}$ and $y_1\, y_2 \ldots y_{j-1}$
- Case 2a: OPT leaves x_i unmatched.
  - pay gap for x_i and min cost of aligning $x_1\, x_2 \ldots x_{i-1}$ and $y_1\, y_2 \ldots y_j$
- Case 2b: OPT leaves y_j unmatched.
  - pay gap for y_j and min cost of aligning $x_1\, x_2 \ldots x_i$ and $y_1\, y_2 \ldots y_{j-1}$

$$OPT(i, j) = \begin{cases} j\delta & \text{if } i = 0 \\ \min \begin{cases} \alpha_{x_i y_j} + OPT(i-1, j-1) \\ \delta + OPT(i-1, j) \\ \delta + OPT(i, j-1) \end{cases} & \text{otherwise} \\ i\delta & \text{if } j = 0 \end{cases}$$

Sequence Alignment: Algorithm

```
Sequence-Alignment(m, n, x1x2...xm, y1y2...yn, δ, α) {
   for i = 0 to m
      M[i, 0] = iδ
   for j = 0 to n
      M[0, j] = jδ

   for i = 1 to m
      for j = 1 to n
         M[i, j] = min(α[xi, yj] + M[i-1, j-1],
                       δ + M[i-1, j],
                       δ + M[i, j-1])
   return M[m, n]
}
```

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

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

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

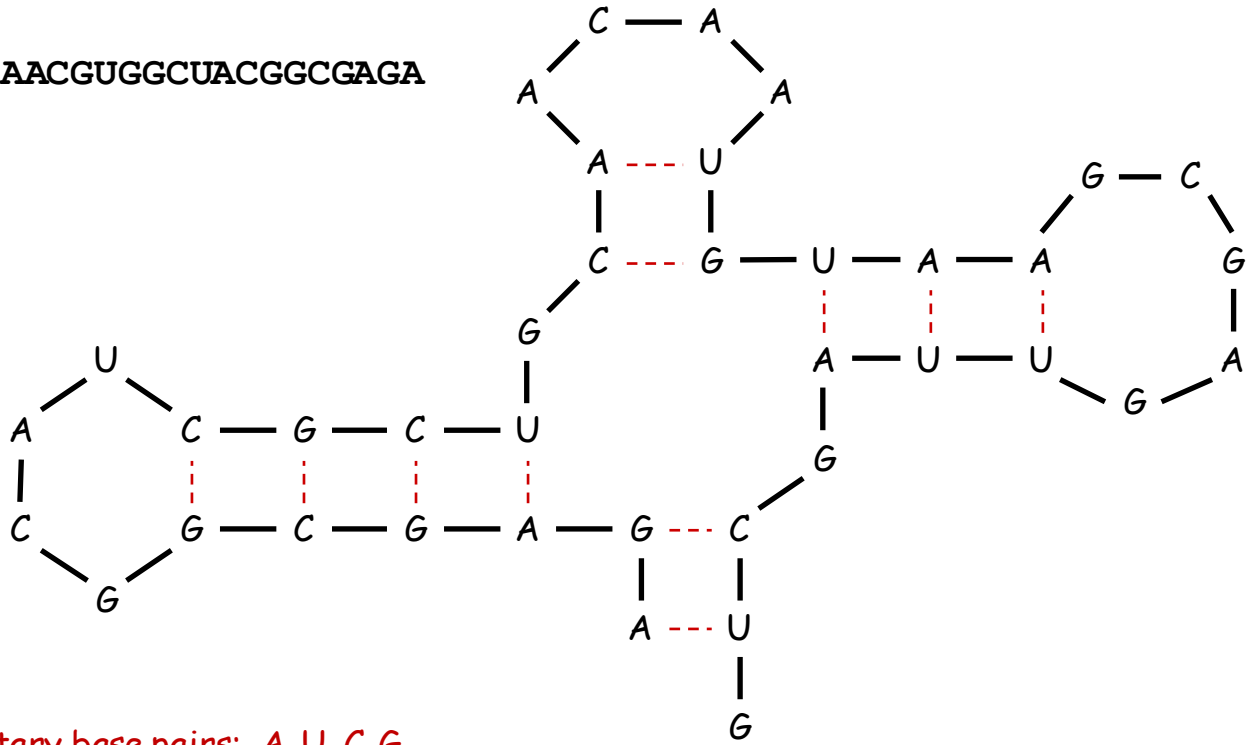
6.5 RNA Secondary Structure

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 molecule.

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 $i < k < j < l$.

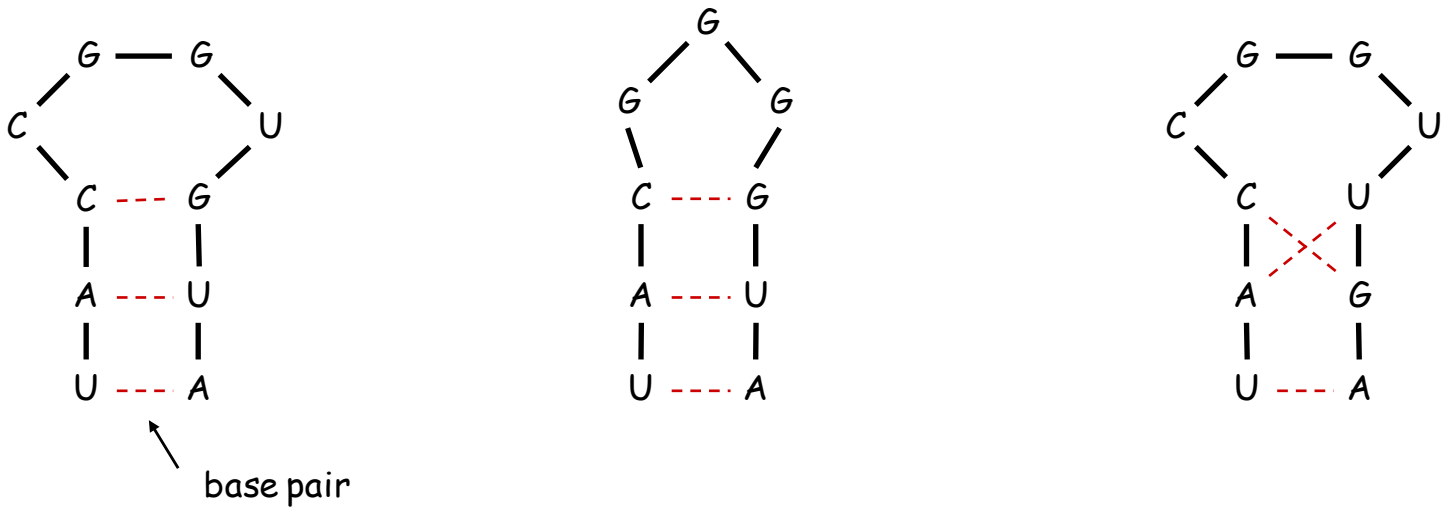
Free energy. Usual hypothesis is that an RNA molecule will form the secondary structure with the optimum total free energy.

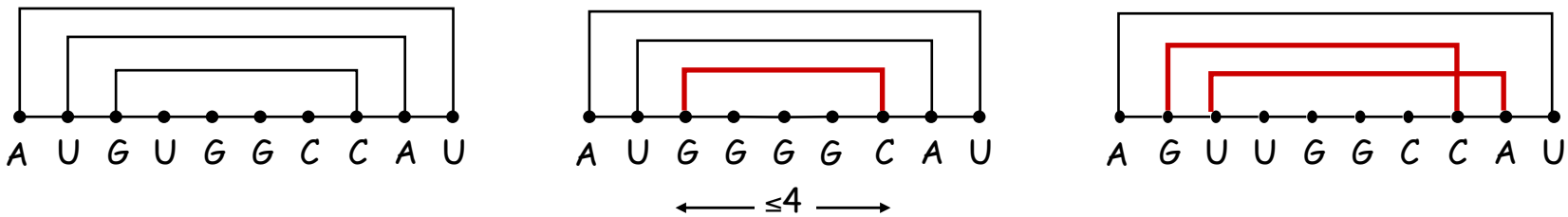
approximate by number of base pairs

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

RNA Secondary Structure: Examples

Examples.





ok

sharp turn

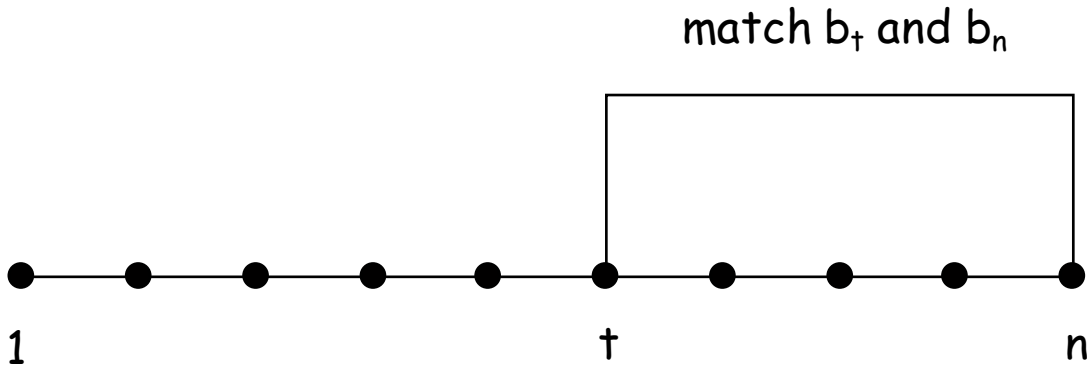
If $(b_i, b_j) \in S$, then $i < j - 4$.

crossing

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

RNA Secondary Structure: Subproblems

First attempt. OPT(j) = maximum number of base pairs in a secondary structure of the substring $b_1b_2\ldots b_j$.



Difficulty. Results in two sub-problems.

- Finding secondary structure in: $b_1b_2\ldots b_{t-1}$. ← OPT(t-1)
- Finding secondary structure in: $b_{t+1}b_{t+2}\ldots b_{n-1}$. ← need more sub-problems

Dynamic Programming Over Intervals

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

- Case 1. If $i \geq j - 4$.
  - 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 $i \leq t < j - 4$. (Multi-way choice)
  - non-crossing constraint decouples resulting sub-problems
  - $OPT(i, j) = 1 + \max_t \{ OPT(i, t-1) + OPT(t+1, j-1) \}$

take max over all t such that $i \leq t < j-4$ and
b_t and b_j are Watson-Crick complements (A-U, C-G)

Bottom Up Dynamic Programming Over Intervals

Q. What order to solve the sub-problems?

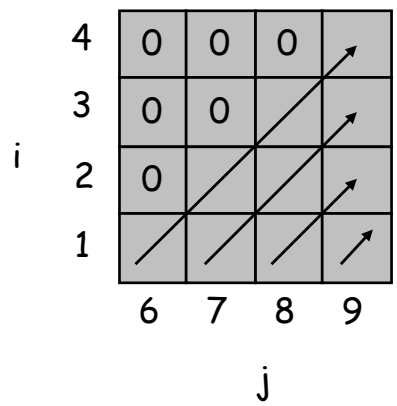
A. Do shortest intervals first.

All substrings of length k+1

```
RNA(b1,…,bn) {
   for k = 5, 6, …, n-1
      for i = 1, 2, …, n-k
         j = i + k
         Compute M[i, j]

   return M[1, n]
}
```

using recurrence



Running time. $O(n^3)$.

When subproblems correspond to intervals, filling by diagonal is usually the case

6.4 Knapsack Problem

Knapsack Problem

Knapsack problem.

- Given n objects and a "knapsack."
- Item i weighs $w_i > 0$ kilograms and has value $v_i > 0$.
- Knapsack has capacity of W kilograms.
- Goal: fill knapsack so as to maximize total value.

Ex: { 3, 4 } has value 40.

W = 11

#	value	weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Greedy: try to use your ideas from HW1 for this variant of the problem? Most probably not optimal…. (DL students still working)

Dynamic Programming: False Start

Def. OPT(i) = max profit subset of items 1, …, i.

- Case 1: OPT does not select item i.
  - OPT selects best of { 1, 2, …, i-1 }

- Case 2: OPT selects item i.
  - accepting item i does not immediately imply that we will have to reject other items
  - without knowing what other items were selected before i, we don't even know if we have enough room for i

Conclusion. Need more sub-problems!

Dynamic Programming: Adding a New Variable

Def. OPT(i, w) = max profit subset of items 1, …, i with weight limit w.

- Case 1: OPT does not select item i.
  - OPT selects best of { 1, 2, …, i-1 } using weight limit w

- Case 2: OPT selects item i.
  - Gains v_i, and has new remaining weight limit = $w - w_i$
  - OPT selects best of { 1, 2, …, i-1 } using this new weight limit $w - w_i$

- TODO: Write the recurrence relation, base cases, where is the solution, and write bottom-up algorithm (check dependences)