COL351: Slides for Lecture Component 24

Thanks to Miles Jones, Russell Impagliazzo, and Sanjoy Dasgupta at UCSD for these slides.

DYNAMIC PROGRAMMING

DP = BT + memoization

Memoization = store and re-use, like the Fibonnacci algorithm (from first week lectures)

Two simple ideas, but easy to get confused if you rush:

- Where is the recursion? (It disappears into the memoization, like the Fib. Example did). Have I made a decision? (only temporarily, like BT)

If you don't rush, a surprisingly powerful and simple algorithm technique

One of the most useful ideas around

THE LONG WAY

1. Come up with simple back-tracking algorithm
2. Characterize sub-problems
3. Define matrix to store answers to the above
4. Simulate BT algorithm on sub-problem
5. Replace recursive calls with matrix elements
6. Invert "top-down" order of BT to get "bottom-up" order

FINAL ALGORITHM

1. Come up with simple back-tracking algorithm
2. Characterize sub-problems
3. Define matrix to store answers to the above
4. Simulate BT algorithm on sub-problem
5. Replace recursive calls with matrix elements
6. Invert "top-down" order of BT to get "bottom-up" order

7. Assemble into DP algorithm:
- Fill in base cases into matrix
- In bottom-up order do: Use translated recurrence to fill in each matrix element
- Return “main problem” answer
- (Trace-back to get corresponding solution)

THE EXPERT'S WAY

Define sub-problems and corresponding matrix

Give recursion for sub-problems

Find bottom-up order

Assemble as in the long way:

- Fill in base cases of the recursion
- In bottom-up order do:
  - Fill in each cell of the matrix according to recursion
- Return main case
- (Traceback to find corresponding solution)

EITHER WAY, A MUST

You MUST explain what each cell of the matrix means AS a solution to a sub-problem

You MUST explain what the recursion is in terms of a LOCAL, COMPLETE case analysis

Undocumented dynamic programing is indistinguishable from nonsense. Assumptions about optimal solution almost always wrong.

LONGEST COMMON SUBSEQUENCE

General issue: Comparing strings

Applications: Comparing versions of documents to highlight recent edits (diff), copyright infringement, plagiarism detection, genomics (comparing strands of DNA)

Many variants for particular applications, but use same general idea.

We'll look at one of the simplest, longest common subsequence

WHY HAMMING DISTANCE IS INADEQUATE

Hamming distance: Line the two strings up and compare them character by character. Count the number of identical symbols (distance= number of different symbols).

Example:

- ALOHA
- HALLOA
  No matches!!

Hamming distance is not robust under small shifts, spacing, insertions

- ALOHA
- HALLOA
  3 matches

LONGEST COMMON SUBSEQUENCE

A subsequence of a string is a string that appears left to right within the word, but not necessarily consecutively

The longest common subsequence (LCS) of two words is the largest string that is a subsequence of both words

ALOH**A**

H**AL**L**OA**

ALOA is a subsequence of both.

RECURSION

ALOHA
HALLOA
First letter mismatch: Must drop first letter from one or the other word

ALOHA	or	LOHA
ALLOA		HALLOA

First letter match: Can keep first letter, and find LCS in rest

ALOHA LOHA OHA
ALLOA = A + LLOA = AL + LOA

BT ALGORITHM

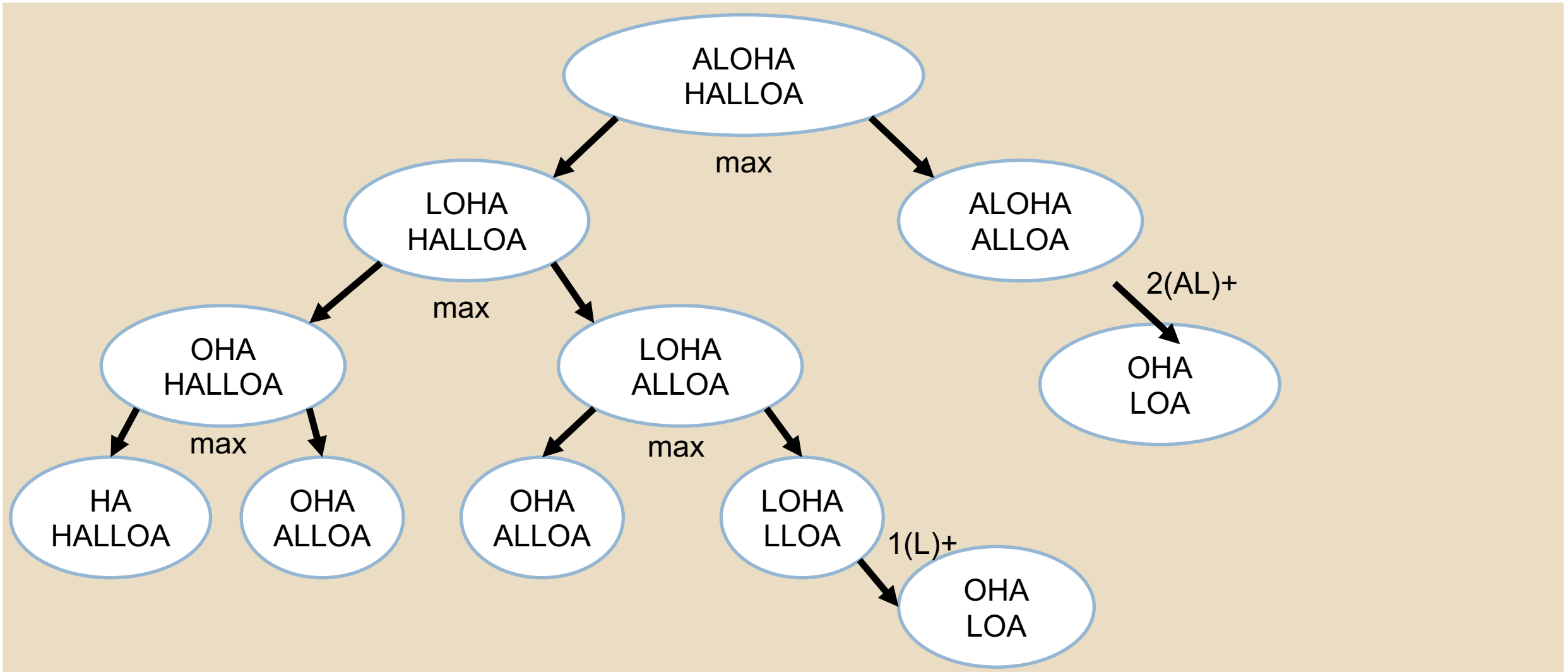
LCS($u_1, \dots, u_n;\ v_1, \dots, v_m$)

IF $n = 0$ or $m = 0$ return 0

IF $u_1 = v_1$ return 1+ LCS($u_2, \dots, u_n;\ v_2, \dots, v_m$)

ELSE return max(LCS($u_2, \dots, u_n;\ v_1, \dots, v_m$), LCS($u_1, \dots, u_n;\ v_2, \dots, v_m$))

EXAMPLE


ALOHA
HALLOA
max
LOHA
HALLOA
ALOHA
ALLOA
2(AL)+
max
OHA
HALLOA
LOHA
ALLOA
OHA
LOA
max
max
HA
HALLOA
OHA
ALLOA
OHA
ALLOA
LOHA
LLOA
1(L)+
OHA
LOA

SUBPROBLEMS

Say we start with words $u_1 \dots, u_n$

$v_1, \dots, v_m$

In recursive calls, we recursively compute the LCS

between one word of the form:

and another word of the form:

SUBPROBLEMS

Say we start with words $u_1, \dots, u_n$

$v_1, \dots v_m$

In recursive calls, we recurse on: $u_I, \dots, u_n, I = 1 \dots n + 1$

to: $v_J, \dots, v_m, J = 1 \dots m + 1$

($I = n + 1$: first word empty, $J = m + 1$: second word empty)

Use matrix L[I, J]:= LCS($u_I, \dots, u_n;\ v_J, \dots, v_m$)

BT ALGORITHM

LCS($u_1, \dots, u_n$; $v_1, \dots, v_m$)

LCS($u_I, \dots, u_n$; $v_J, \dots, v_m$)

IF $n = 0$ or $m = 0$ return 0

IF or return 0

IF $u_1 = v_1$ return 1+ LCS($u_2, \dots, u_n$; $v_2, \dots, v_m$)

IF return 1+ LCS (;)

ELSE return max(LCS($u_2, \dots, u_n$; $v_1, \dots, v_m$),
LCS($u_1, \dots, u_n$; $v_2, \dots, v_m$))

ELSE return max(LCS(;)
LCS(;))

BT ALGORITHM

LCS($u_I, \dots, u_n;\ v_J, \dots, v_m$)

To fill in L[I, J]

IF $I = n + 1$ or $J = m + 1$ return 0

Base cases: L[,] = L[,] = 0

IF $u_I = v_J$ return 1+ LCS($u_{I+1}, \dots, u_n;\ v_{J+1}, \dots, v_m$)

IF $u_I = v_J$ THEN L[I, J]:= 1+

ELSE return max(LCS($u_{I+1}, \dots, u_n;\ v_J, \dots, v_m$),

LCS($u_I, \dots, u_n;\ v_{J+1}, \dots, v_m$)

ELSE L[I, J] : = max (L[,], L[,])

FINAL RECURRENCES

L$[I, J]$ ≡ max length of common subsequence between $u_I, \ldots, u_n$, $v_J, \ldots, v_m$

Base cases: L$[m+1, J]$ =0, L$[I, n+1]$ =0

Recurrence: IF $u_I = v_J$ THEN L$[I, J]$:= 1+ L$[I+1, J+1]$

ELSE L$[I, J]$:= max (L$[I+1, J]$, L$[I, J+1]$)

BOTTOM UP ORDER

Top down: I increases OR J increases

Bottom up: Both I and J decrease

DP-VERSION

DPLCS($u_1, \dots, u_n;\ v_1, \dots, v_m$)

Initialize L[$1 \dots n+1, 1 \dots m+1$]

FOR $I = 1$ to $n+1$ do: L[$I, m+1$]:=0

FOR $J = 1$ to m do: L[$n+1, J$]:=0

FOR $I = n$ down to 1 do:

FOR $J = m$ down to 1 do:

IF $u_I = v_J$ THEN L[I, J] :=1+ L[$I+1, J+1$]

ELSE L[I, J] := max(L[$I+1, J$], L[$I, J+1$])

Return L[1,1]

EXAMPLE

	H	A	L	L	O	A	
A							0
L							0
O							0
H							0
A							0
	0	0	0	0	0	0	0

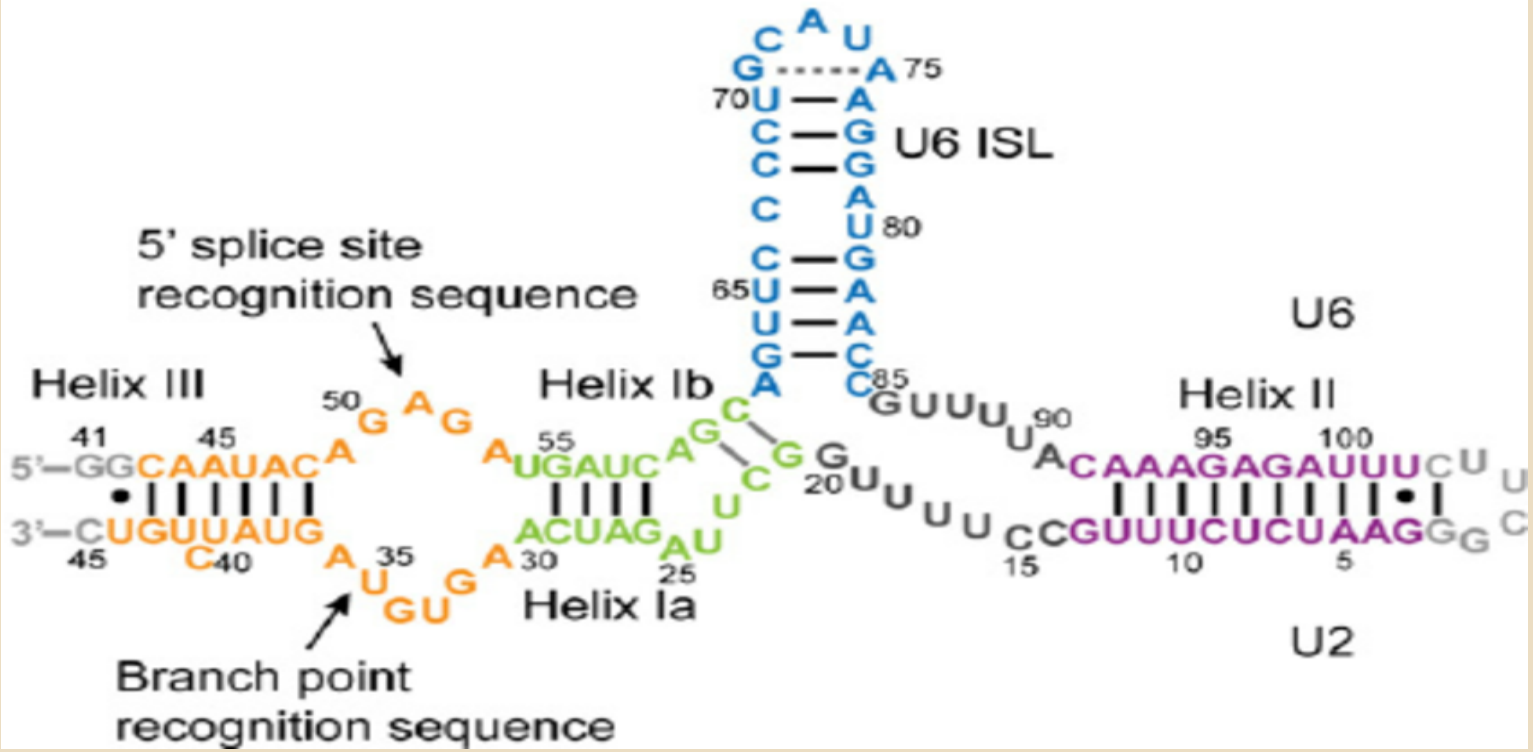
EXAMPLE

	H	A	L	L	O	A	
A	4	4	3	3	2	1	0
L	3	3	3	3	2	1	0
O	2	2	2	2	2	1	0
H	2	1	1	1	1	1	0
A	1	1	1	1	1	1	0
	0	0	0	0	0	0	0

SURPRISING RELATIONSHIP

[ABW, 2015]: *"If a conjecture by Impagliazzo-Paturi about the worst-case complexity of SAT (famous NP-complete problem) is true, then there is no substantial improvement in this algorithm for LCS possible"*

SECONDARY STRUCTURE

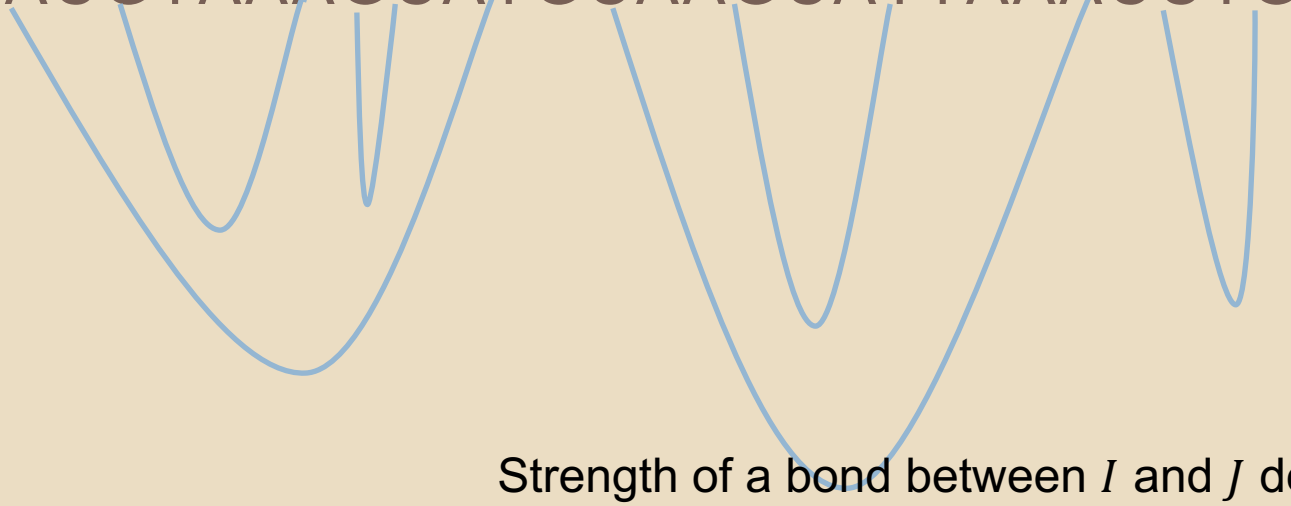


RNA folds back on itself, forming chemical bonds between amino acids in the sequence

SECONDARY STRUCTURE IS "OUTER PLANAR"

If we view the protein as a string, the secondary bonds form a matching on the characters of the string with a restriction: bonded pairs are either entirely inside or entirely outside other bonded pairs

ACGTAAAGCATGCAAGCATTAAACCTGG



Strength of a bond between I and J depends on the two amino acids, Strength(w_I, w_J) (given as a table with 10 numbers, for the 10 pairs possible)

MAX STRENGTH SECONDARY STRUCTURE

Given $w_1 \ldots w_n$, find the maximum possible strength of a secondary structure meeting the constraints of no intersecting bonds.

Cases:

- w_1 not matched
- w_1 bonded to w_I

Combines DP with divide and conquer

BACKTRACKING VERSION

Either w_1 bonds to some w_I, $I > 1$ or remains unbonded.

If it bonds to w_I, can only bond within $2 \ldots I-1$ and $I+1 \ldots n$

BTSS($w_1 \ldots w_n$)

IF $n = 0$ or $n = 1$ return 0

Max:= BTSS[$w_2, .. w_n$] *//(case when w_1 unbonded)*

FOR $I = 2$ to n do:

THISCASE:= strength(w_1, w_I) + BTSS($w_2, \ldots, w_{I-1}$) + BTSS($w_{I+1}, \ldots, w_n$)

IF THISCASE > Max THEN Max:=THISCASE

Return Max

SUBPROBLEMS

Subproblems all have the form $w_I, \ldots, w_J$, consecutive subsequences

As we recur, size = $J - I + 1$ gets smaller.

Bottom up: size gets larger

Size=1, 0 : no bonds possible (Use $J = I - 1$ for size 0)

MS$[I, J]$: = max strength of secondary structure for $w_I, \ldots, w_J$

DP ALGORITHM

DPSS($w_1 \dots w_n$)

Initialize MS[$1 \dots n, 0 \dots n$]

For $I = 1$ to n do:

MS[$I, I-1$] =0; MS[I, I]=0

For $K = 1$ to $n-1$ do:

FOR $I = 1$ to $n-K$ do:

MS[$I, I+K$] : = MS[$I+1, I+K$]

FOR $L = I+1$ to $I+K$ do:

MS[$I, I+K$]:= max(MS[$I, I+K$], Strength(w_I, w_L) + MS[$I, L-1$] + MS[$L+1, I+K$])

Return MS[$1, n$]

TIME ANALYSIS

DPSS($w_1 \dots w_n$)

Initialize MS[$1 \dots n, 0 \dots n$]

For $I = 1$ to n do:

MS[$I, I-1$] =0; MS[I, I]=0

For $K = 1$ to $n-1$ do:

FOR $I = 1$ to $n-K$ do:

MS[$I, I+K$] : = MS[$I+1, I+K$]

FOR $L = I+1$ to $I+K$ do:

MS[$I, I+K$]:= max(MS[$I, I+K$], Strength(w_I, w_L) + MS[$I, L-1$] + MS[$L+1, I+K$])

Return MS[$1, n$]

BEST ALGORITHM

Bringman, Grandoni, Saha, Vassilevska-Williams [FOCS, 2016]: *$O(n^{2.86\ldots})$ time algorithm for RNA secondary structure, using speeded-up min-plus product and improved matrix multiply algorithms*