COL702: Backtracking and Dynamic Programming
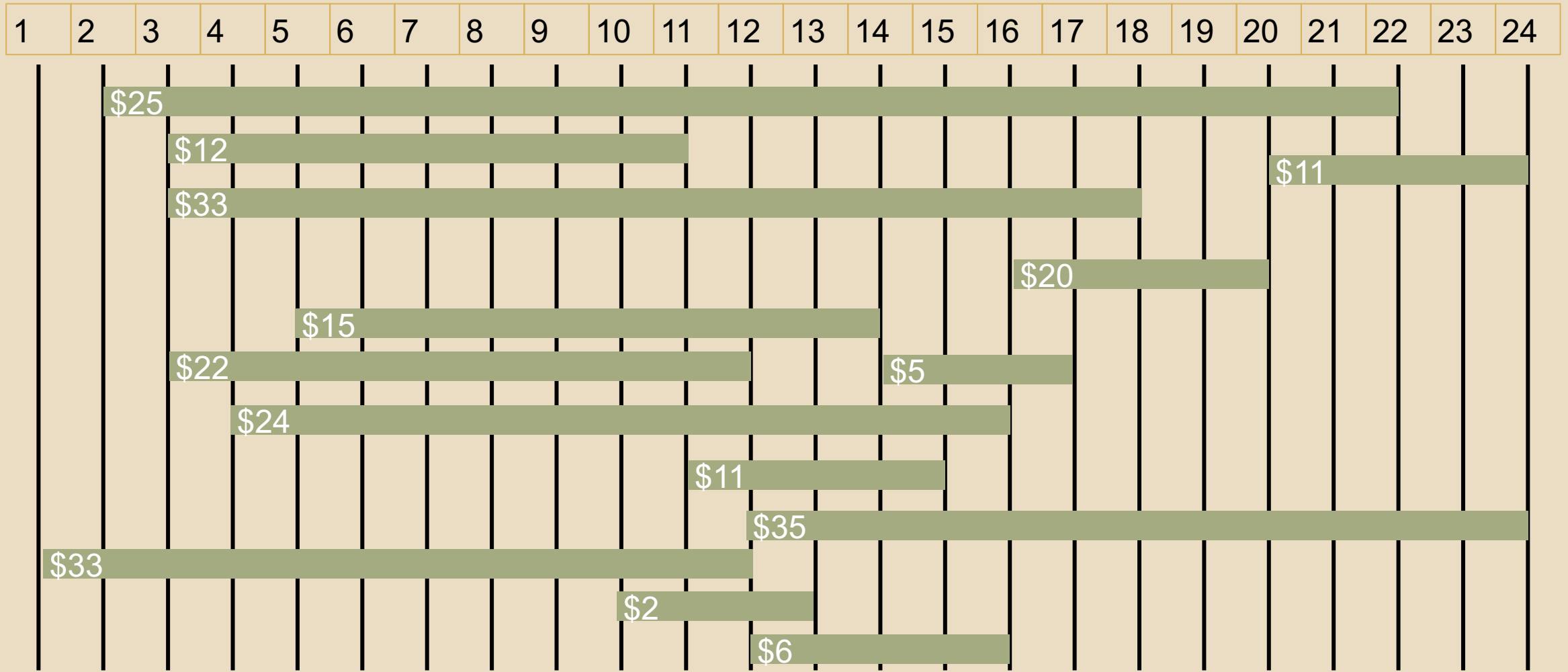
FROM BACKTRACKING TO DYNAMIC PROGRAMMING

- Backtracking = recursive exhaustive local searches

- **Dynamic Programming = Backtracking + Memoization**

  Memoization = store and re-use, like Fibonacci algorithm from intro

Basic principle: "If an algorithm is *recomputing* the same thing many times, we should *store and re-use* instead of recomputing."

FORMAL SPECIFICATION

- Instance:

- Solution:

- Constraints:

- Objective:

FORMAL SPECIFICATION

- Instance: List of n intervals $I = (s, f, v)$, with values $v > 0$

- Solution: subset of intervals $S = \{(s_1, f_1, v_1), (s_2, f_2, v_2) \ldots (s_k, f_k, v_k)\}$

- Constraints: cannot pick intersecting intervals: $s_1 < f_1 \leq s_2 < f_2 \leq \cdots . s_k \leq f_k$

- Objective: maximize total value of intervals chosen: $\Sigma\, v_i$

NO KNOWN GREEDY ALGORITHM

In fact, some people (Borodin, Nielsen, and Rackoff) have proved that no greedy algorithm even approximates the optimal solution.

Let's try back-tracking (as warm-up to dynamic programming)...

BACKTRACKING

- Sort events by start time. Call them $I_1 \ldots I_n$.
- Pick first of these: I_1.
- Should we include I_1 or not? Try both possibilities.

BTWES $(I_1 \ldots I_n)$:
      If n=0 return 0
      If n=1 return V_1
      Exclude := BTWES$(I_2 .. I_n)$
      J:=2
      Until (J > n or s_J > f_1) do:
            J++
      Include:= V_1 + BTWES$(I_J .. I_n)$
      Return Max(Include, Exclude)

TIME IS HORRIBLE

$O(2^n)$ worst-case time, same as exhaustive search.

We could try to improve it, like we did for Maximum Independent Set.
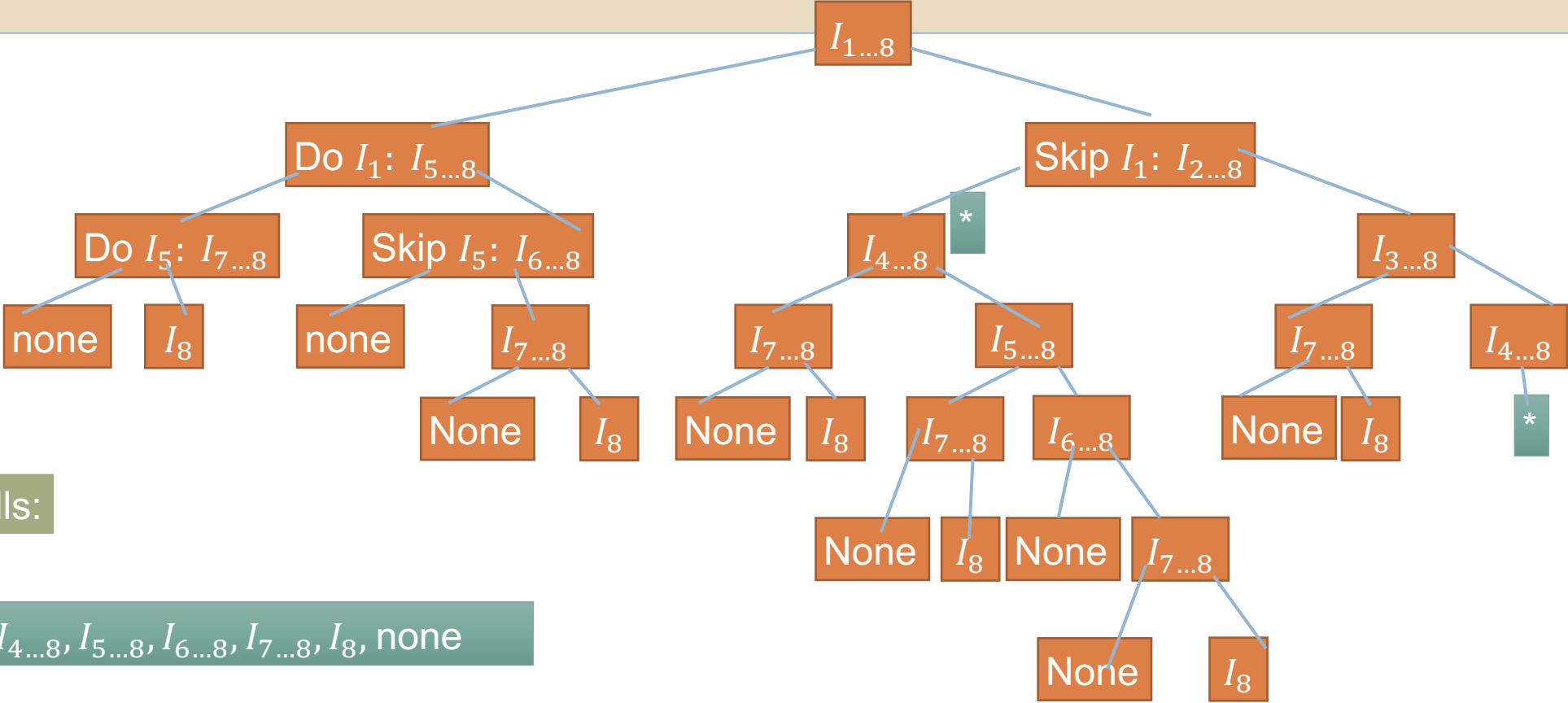
But our goal is a dynamic programming algorithm, so improving the backtracking time is irrelevant.

EXAMPLE

- $I_1 = (1,5), V_1 = 4$
- $I_2 = (2,4), V_2 = 3$
- $I_3 = (3,7), V_3 = 5$
- $I_4 = (4,9), V_4 = 6$
- $I_5 = (5,8), V_5 = 3$
- $I_6 = (6,11), V_6 = 4$
- $I_7 = (9,13), V_7 = 5$
- $I_8 = (10,12), V_8 = 3$

EXAMPLE

I1 = (1,5), V1=4
I2 = (2,4), V2=3
I3 = (3,7), V3=5
I4 = (4,9), V4=6
I5 = (5,8), V5=3
I6= (6,11), V6=4
I7 =(9,13), V7=5
I8= (10,12), V8=3

Distinct calls:

$I_{1...8}, I_{2...8}, I_{3...8}, I_{4...8}, I_{5...8}, I_{6...8}, I_{7...8}, I_8$, none

CHARACTERIZE CALLS MADE

All of the recursive calls BTWES makes are to arrays of the form
$I_{K\dots n}$, with K=1…n, or empty

So of the 2^n recursive calls we might make, most are duplicates… there are only n+1 distinct possibilities!

- Just like Fibonacci numbers: many calls made exponentially often.
- Solution same: Create array to store and re-use answers, rather than repeatedly solving them.

DEFINE SUBPROBLEMS

The values needed are the solutions to the subproblems $(I_K..I_n)$ for all $K = 1 \ldots n$ and the empty set. There are $n + 1$ subproblems of this form so we need an array of size $n + 1$.

- Let MV[1...n+1] be this array
- Let MV[K] hold the total weight of the maximum weight non-intersecting set of events from the sub-problem $(I_K..I_n)$
- We'll use MV[n+1] to hold the best weight for the empty list, 0.
- So K ranges from 1 to n+1.

SIMULATE RECURSION ON SUBPROBLEM

What happens when we run BTWES $(I_K \dots I_n)$?

BTWES $(I_K \dots I_n)$
    If K=n+1 return 0
    If K=n return V_n
    Exclude:= BTWES$(I_{K+1} .. I_n)$
    J:=K+1
    Until (J > n or s_J > f_K) do:
        J++
    Include:= V_K + BTWES$(I_J .. I_n)$
    Return Max(Include, Exclude)

REPLACE RECURSION WITH ARRAY/MATRIX

MV[n+1]:=0
MV[n]:= V_n

For K in the range 1 to n-1:
    Exclude:=MV[K+1]
    J:=K+1
    Until (J > n or s_J > f_K) do:
        J++
    Include:= V_K + MV[J]
    MV[K]:= Max(Include, Exclude)

Recall: MV[K] is the solution to the subproblem ($I_K..I_n$)

INVERT TOP-DOWN RECURSION ORDER TO GET BOTTOM UP ORDER

BTWES $(I_K \dots I_n)$

  If K=n+1 return 0

  If K=n return V_n

  Exclude:= BTWES$(I_{K+1} \dots I_n)$

  J:=K+1

  Until (J > n or $s_J > f_K$) do:

      J++

  Include:= V_K + BTWES$(I_J \dots I_n)$

  Return Max(Include, Exclude)

Top-down: recursive calls increase K, go from K=1 to K=n+1

Bottom-up: Need to fill in array from K=n+1 to K=1

ASSEMBLE INTO FINAL DP ALGORITHM

Fill in base cases of array. Fill in rest of array in bottom up order.

DPWES$[I_1..I_n]$
  MV[n+1]:=0
  MV[n]:= V$_n$
  FOR K=n-1 down to 1 do:
    Exclude:=MV[K+1]
    J:=K+1
    Until (J > n or s$_J$ > f$_K$) do:
      J++
    Include:= V_K + MV[J]
    MV[K]:= Max(Include, Exclude)
  Return MV[1]

Along with your pseudocode, must include a description in words of what your array holds:
MV[K] is the maximum weight of all non-intersecting subsets of the events $(I_K, …, I_n)$
And MV[n+1]=0

EXAMPLE

- I1 = (1,5), V1=4
- I2 = (2,4), V2=3
- I3 = (3, 7), V3=5
- I4 = (4,9), V4=6
- I5 = (5,8), V5=3
- I6= (6,11), V6=4
- I7 =(9,13), V7=5
- I8= (10,12), V8=3

Include	Exclude	MV

EXAMPLE

	Include	Exclude	MV
I1 = (1,5), V1=4.			
I2 = (2,4), V2=3			
I3 = (3, 7), V3=5			
I4 = (4,9), V4=6			
I5 = (5,8), V5=3	3+MV[7]=8	MV[6]=5	8
I6= (6,11), V6=4	4+MV[9]=4	MV[7]=5	5
I7 =(9,13), V7=5	5+MV[9]=5	MV[8]=3	5
I8= (10,12), V8=3			3
			0

EXAMPLE

	Include	Exclude	MV
I1 = (1,5), V1=4.	4+MV[5]=12	MV[2]=14	14
I2 = (2,4), V2=3	3+MV[4]=14	MV[3]=11	14
I3 = (3, 7), V3=5	5+MV[7]=10	MV[4]=11	11
I4 = (4,9), V4=6	6+MV[7]=11	MV[5]=8	11
I5 = (5,8), V5=3	3+MV[7]=8	MV[6]=5	8
I6= (6,11), V6=4	4+MV[9]=4	MV[7]=5	5
I7 =(9,13), V7=5	5+MV[9]=5	MV[8]=3	5
I8= (10,12), V8=3			3
			0

TRACING FORWARDS

	Include	Exclude	MV	
I1 = (1,5), V1=4.	4+MV[5]=12	MV[2]=14	14	Exclude 1, go to 2
I2 = (2,4), V2=3	3+MV[4]=14	MV[3]=11	14	Include 2, go to 4
I3 = (3, 7), V3=5	5+MV[7]=10	MV[4]=11	11	
I4 = (4,9), V4=6	6+MV[7]=11	MV[5]=8	11	Include 4, go to 7
I5 = (5,8), V5=3	3+MV[7]=8	MV[6]=5	8	
I6= (6,11), V6=4	4+MV[9]=4	MV[7]=5	5	Include 7 go to 9
I7 =(9,13), V7=5	5+MV[9]=5	MV[8]=3	5	
I8= (10,12), V8=3			3	
			0	None left
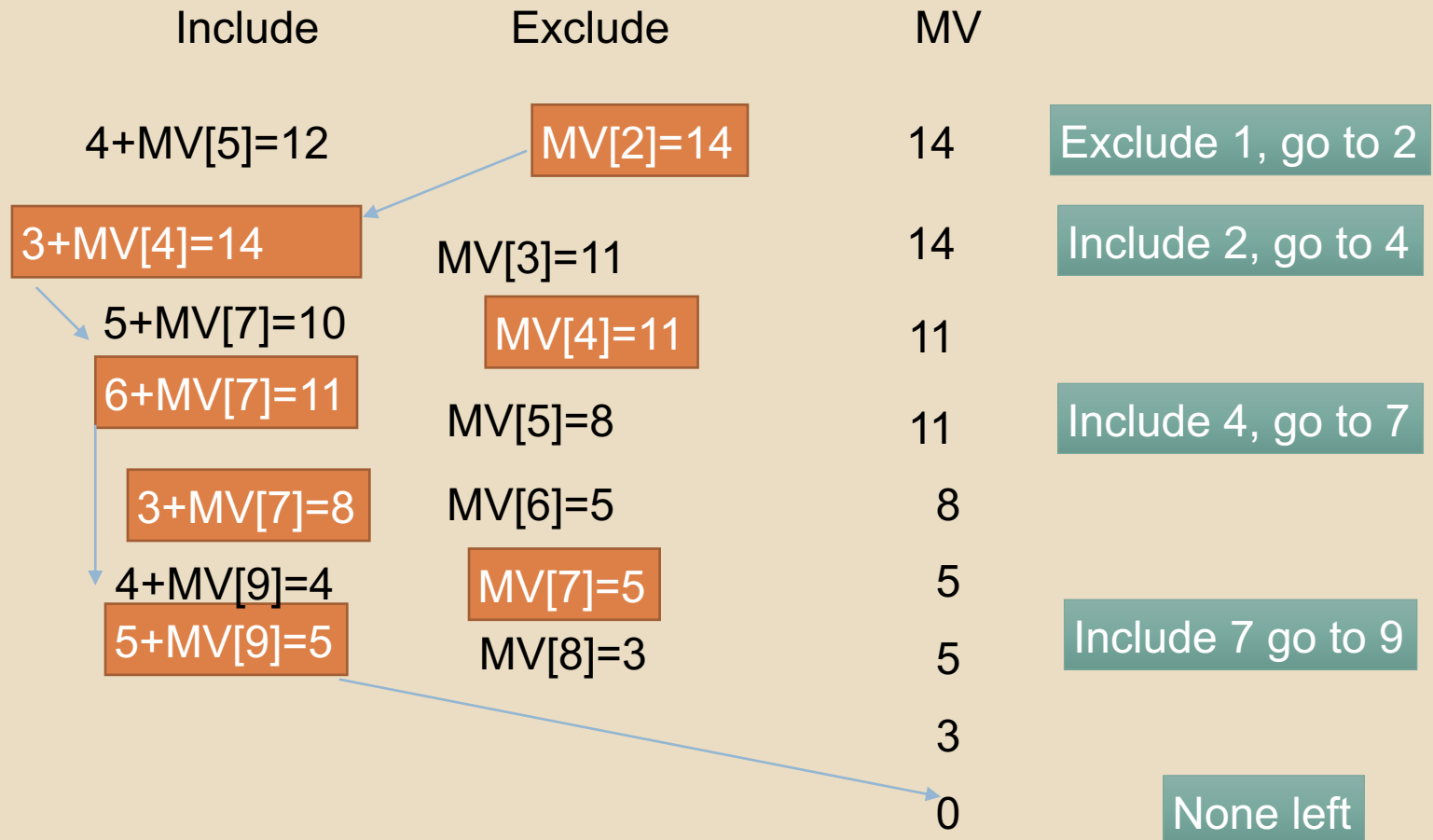
Best set: 2,4, 7, Total value: 3+6+5=14

CORRECTNESS

Prove BT algorithm correct, and explain translation, to show DP=BT.

TIME ANALYSIS

DP: Fill in base cases of array. Fill in rest of array in bottom up order
Time = size of array/matrix times time per entry

DPWES$[I_1..I_n]$
    MV[n+1]:=0
    MV[n]:= V_n
    FOR K=n-1 down to 1 do:
        Exclude:=MV[K+1]
        J:=K+1
        Until (J > n or s_J > f_K) do:
            J++
        Include:= V_K + MV[J]
        MV[K]:= Max(Include, Exclude)
    Return MV[1]

TIME ANALYSIS

DP: Fill in base cases of array. Fill in rest of array in bottom up order
Time = size of array/matrix. O(n) times time per entry O(n) = O(n^2)
(Can you think of ways to speed this up for this example?)

DPWES[$I_1..I_n$]
      MV[n+1]:=0
      MV[n]:= V_n
      FOR K=n-1 down to 1 do:
            Exclude:=MV[K+1]
            J:=K+1
            Until (J > n or s_J > f_K) do:
                  J++
            Include:= V_K + MV[J]
            MV[K]:= Max(Include, Exclude)
      Return MV[1]

DP = BT + MEMOIZE

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

Where is the recursion?   (Final algorithm is iterative, but based on recursion)

Have I made a decision? (Only conditionally, like BT, not fixed, like greedy)

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

One of the most useful ideas around

DYNAMIC PROGRAMMING

Dynamic programming is an algorithmic paradigm in which a problem is solved by:

- identifying a collection of subproblems
- tackling them one by one, smallest first, using the answers to small problems to help figure out larger ones, until they are all solved.

COL702: Backtracking and Dynamic Programming

DYNAMIC PROGRAMMING

Dynamic programming is an algorithmic paradigm in which a problem is solved by:

Identifying a collection of subproblems.

Tackling them one by one, smallest first, using the answers to small problems to help figure out larger ones, until they are all solved.

DP STEPS (BEGINNER)

1. Design simple backtracking algorithm
2. Characterize subproblems that can arise in backtracking
3. Simulate backtracking algorithm on subproblems
4. Define array/matrix to hold different subproblems
5. Translate recursion from step 3 in terms of matrix positions: Recursive call becomes array position; return becomes write to array position
6. Invert top-down recursion order to get bottom up order
7: Assemble: Fill in base cases

      In bottom-up order do:

        Use step 5 to fill in each array position

      Return array position corresponding to whole input

DYNAMIC PROGRAMMING STEPS (EXPERT)

Step1: Define the subproblems

Step 2: Define the base cases

Step 3: Express subproblems recursively

Step 4: Order the subproblems

EITHER WAY

1. You MUST explain what each cell of the table/matrix means AS a solution to a subproblem.

That is, clearly define the subproblems.

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

That is, explain how subproblems are solved using other, "smaller", subproblems.

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

LONGEST INCREASING SUBSEQUENCE

Given a sequence of distinct positive integers a[1],…,a[n]

An increasing subsequence is a sequence $a[i_1],…,a[i_k]$ such that $i_1<…<i_k$ and $a[i_1]<…<a[i_k]$.

For example: 15, 18, 8, 11, 5, 12, 16, 2, 20, 9, 10, 4

5, 16, 20 is an increasing subsequence.

How long is the longest increasing subsequence?

DYNAMIC PROGRAMMING: EXPERT MODE

What is a suitable notion of subproblem?

For example: 15, 18, 8, 11, 5, 12, 16, 2, 20, 9, 10, 4

DYNAMIC PROGRAMMING: EXPERT MODE

Step1: Define the subproblems

L(k) = length of the longest increasing subsequence ending exactly at position k

Step 2: Base Case

L(1)=1

Step 3: Express subproblems recursively

$L(k) = 1 + \max(\{L(i): i < k, a_i < a_k\})$

Step 4: Order the subproblems

Solve them in the order L(1), L(2), L(3), ...

Try it out!   a = [15, 18, 8, 11, 5, 12, 16, 2, 20, 9, 10, 4].

LONGEST INCREASING SUBSEQUENCE

Subproblem: L[k] = length of LIS ending exactly at position k

L[1] = 1
For k = 2 to n:
  Len = 1
  For i = 1 to k-1:
    If a[i] < a[k] and Len < 1+L[i]:
      Len = 1+L[i]
  L[k] = Len
return max(L[1], L[2], …, L[n])

LONGEST INCREASING SUBSEQUENCE

Given a sequence of distinct positive integers a[1],…,a[n]

An increasing subsequence is a sequence $a[i_1],…,a[i_k]$ such that $i_1<…<i_k$ and $a[i_1]<…<a[i_k]$.

For example: 15, 18, 8, 11, 5, 12, 16, 2, 20, 9, 10, 4

5, 16, 20 is an increasing subsequence.

How long is the longest increasing subsequence?

THE LONG WAY

1. Come up with simple backtracking algorithm
2. Characterize subproblems
3. Define matrix to store answers to the above
4. Simulate BT algorithm on subproblem
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
   - Use translated recurrence to fill in each matrix element
   - Return "main problem" answer
   - (Trace-back to get corresponding solution)

LONGEST INCREASING SUBSEQUENCE

What is a **local decision**?

More than one possible answer…

LONGEST INCREASING SUBSEQUENCE

What is a local decision?

Version 1: For each element, is it in the subsequence?
Possible answers: Yes, No

Version 2:   What is the first element in the subsequence? The second?
Possible answers: 1…n.

Either way, we need to generalize the problem a bit to solve recursively.

FIRST CHOICE, RECURSION

Assume we're only allowed to use entries bigger than V.

(Initially, set V=-1, and branch on whether or not to include A[1].)

We'll just return the length of the LIS.

BTLIS1(V, A[1…n])

   If n=0 then return 0

   If n=1 then if A[1] > V then return 1 else return 0

   OUT:= BTLIS(V, A[2..n]) {if we do not include A[1]}

   IF A[1] > V then IN:= 1+BTLIS(A[1],A[2..n]) else IN:= 0

   Return max (IN, OUT)

EXAMPLE

A[1:12] = [15, 18, 8, 11, 5, 12, 16, 2, 20, 9, 10, 4]

WHAT DO SUBPROBLEMS LOOK LIKE?

Arrays in subcalls are:

V in subcalls are:

Total number of distinct subcalls:

SUBPROBLEMS

Array A[J..n], where J ranges from 1 to n
V is either -1 or of the form A[K]

To simplify things, define A[0] = -1

Define
L[K,J] = (length of) LIS of A[J..n], with elements > A[K]

SIMULATING RECURRENCE

BTLIS(A[K], A[J…n])

If J=n then if A[K] < A[n] return 1 else return 0

OUT:= BTLIS(A[K], A[J+1..n])

IF A[J] > A[K] then IN:= 1 + BTLIS(A[J], A[J+1..n]) else IN:= 0

Return max (IN, OUT)

TRANSLATE RECURRENCE IN TERMS OF MATRIX

BTLIS(A[K], A[J…n])
     If J=n then if A[K] < A[n] return 1 else return 0
     OUT:= BTLIS(A[K], A[J+1..n])
     IF A[J] > A[K] then IN:= 1 + BTLIS(A[J], A[J+1..n]) else IN:= 0
     Return max (IN, OUT)

Recall: L[K,J] = (length of) LIS of A[J..n], with elements > A[K]

     If A[K] < A[n] then L[K,n] := 1 else L[K,n]:=0

     OUT: = L[K,J+1]

     IF A[J] > A[K] then IN:= 1 + L[J,J+1] else IN: = 0

     L[K,J]:= max (IN, OUT)

INVERT TOP-DOWN ORDER TO GET BOTTOM-UP ORDER

Recall: L[K,J] = (length of) LIS of A[J..n], with elements > A[K]

As we recurse, J gets incremented, K sometimes increases

Bottom-up: J gets decremented, K any order

FILL IN MATRIX IN BOTTOM UP ORDER

A[0] := -1

For K=0 to n-1 do:

      IF A[n] > A[K] then L[K,n] := 1 else L[K,n] := 0

For J=n-1 downto 1 do:

      For K=0 to J-1 do:

            OUT := L[K, J+1]

            IF A[J] > A[K] then IN := 1 + L[J,J+1] else IN := 0

            L[K,J] := max(IN, OUT)

Return L[0,1]

Recall: L[K,J] = (length of) LIS of A[J..n], with elements > A[K]

EXAMPLE

A[0:4] = [-1, 15, 8, 11, 2]

	1	2	3	4
0				
1				
2				
3				

Recall: L[K,J] = (length of) LIS of A[J..n], with elements > A[K]

TIME ANALYSIS

```
A[0] := -1
For K=0 to n-1 do:
        IF A[n] > A[K] then L[K,n] := 1 else L[K,n] := 0
For J=n-1 downto 1 do:
        For K=0 to J-1 do:
                OUT := L[K, J+1]
                IF A[J] > A[K] then IN := 1 + L[J,J+1] else IN := 0
                L[K,J] := max(IN, OUT)
Return L[0,1]
```

LONGEST INCREASING SUBSEQUENCE

What is a local decision?

Version 1: For each element, is it in the subsequence?
Possible answers: Yes, No

Version 2: What is the first element in the subsequence? The second?
Possible answers: 1…n.

Either way, we need to generalize the problem a bit to solve recursively.

ANOTHER VIEW OF LONGEST INCREASING SUBSEQUENCE

Let's make a DAG out of our example...

15   18   8   11   5   12   16   2   20   9   10   4

WHY DAGS ARE CANONICAL FOR DP

Consider a graph whose vertices are the distinct recursive calls an algorithm makes, and where calls are edges from the subproblem to the main problem.

This graph had better be a DAG or we're in deep trouble!

This graph should be small or DP won't help much.

Bottom-up order = topological sort

BT TO DP: TREES TO DAGS

BT:

Create a tree of possible subproblems, where branching is based on all consistent next choices for local searches

DP:

Make this tree into a DAG by identifying paths that lead to same problems.

Array indices = names for vertices in this DAG

Expert's method: Skip directly to DAG.

VERSION 2, BACKTRACKING

If the current position we've chosen is A[J], what is the next choice?
Possibilities: J+1,...n, none (need to check greater than A[J])
Again, set A[0]=-1 and start J=0
Only counting choices after A[J]

BTLIS2(A[J...n]) {LIS of A[J+1..n], assuming we've taken A[J]}
      IF n=J return 0
      Max := 0
      FOR K=J+1 TO n do:
            IF A[K] > A[J] THEN:
                  L:= BTLIS2(A[K..n])
                  IF Max < 1+L THEN Max := 1+L
      Return Max

WHAT ARE THE SUB-PROBLEMS?

BTLIS2(A[J...n]) {LIS of A[J+1..n], assuming we've taken A[J]}

    IF n=J return 0
    Max := 0
    FOR K=J+1 TO n do:
        IF A[K] > A[J] THEN:
            L:= BTLIS2(A[K..n])
            IF Max < 1+L THEN Max := 1+L
    Return Max

Again, set A[0]=-1 and start J=0

What are the distinct recursive calls we make throughout this algorithm?

DEFINE ARRAY AND TRANSLATE

Let $M[J] = BTLIS2(A[J..n])$, $J=0…n$

REPLACE RECURSION WITH ARRAY

BTLIS2(A[J...n]) {LIS of A[J+1..n], assuming we've taken A[J]}

    IF n=J return 0

    Max := 0

    FOR K=J+1 TO n do:

        IF A[K] > A[J] THEN:

            L:= BTLIS2(A[K..n])

            IF Max < 1+L THEN Max := 1+L

    Return Max

M[n] := 0

For J in 0 to n-1:

    Max:=0

    FOR K=J+1 TO n do:

        IF A[K] > A[J] THEN:

            L:= M[K]

            IF Max < 1+L THEN Max:= 1+L

    M[J]:= Max

IDENTIFY TOP DOWN ORDER

When we make recursive calls, J is:

So bottom up order means J is:

FILL IN ARRAY IN BOTTOM-UP ORDER

```
DPLIS2(A[1..n])
        A[0] :=-1
        M[n] := 0
        FOR J=n-1 downto 0 do:
                Max := 0
                FOR K=J+1 TO n do:
                        IF A[K] > A[J] THEN:
                                L:= M[K]
                                IF Max < 1+L THEN Max:= 1+L
                M[J] := Max
        Return M[0]
```

Recall: M[J] = (length of) LIS of A[J+1..n], assuming we've taken A[J]

EXAMPLE

A: -1, 15, 18, 8, 11, 5, 12, 16, 2, 20, 9, 10, 4

Recall: M[J] = (length of) LIS of A[J+1..n], assuming we've taken A[J]

TIME ANALYSIS

```
DPLIS2(A[1..n])
      A[0] :=-1
      M[n] := 0
      FOR J=n-1 downto 0 do:
             Max := 0
             FOR K=J+1 TO n do:
                    IF A[K] > A[J] THEN:
                           L:= M[K]
                           IF Max < 1+L THEN Max:= 1+L
             M[J] := Max
      Return M[0]
```

CORRECTNESS

Invariant:

M[J] is length of increasing sequence from A[J+1...n] with elements greater than A[J]

Strong induction on n-J

Base case: When J=n, no choices possible, M[n] = 0

Induction step: We try all possible values for first element.

COL702: Backtracking and Dynamic Programming

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

- <u>General issue</u>: Comparing strings

- <u>Applications</u>: 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

- <u>Hamming distance</u>: 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

- **ALO**HA
- 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

- $\text{LCS}(u_1, \ldots, u_n;\ v_1, \ldots, v_m)$
-     IF $n = 0$ or $m = 0$ return 0
-     IF $u_1 = v_1$ return $1+$ $\text{LCS}(u_2, \ldots, u_n;\ v_2, \ldots, v_m)$
-     ELSE return $\max(\text{LCS}(u_2, \ldots, u_n;\ v_1, \ldots, v_m), \text{LCS}(u_1, \ldots, u_n;\ v_2, \ldots, v_m))$
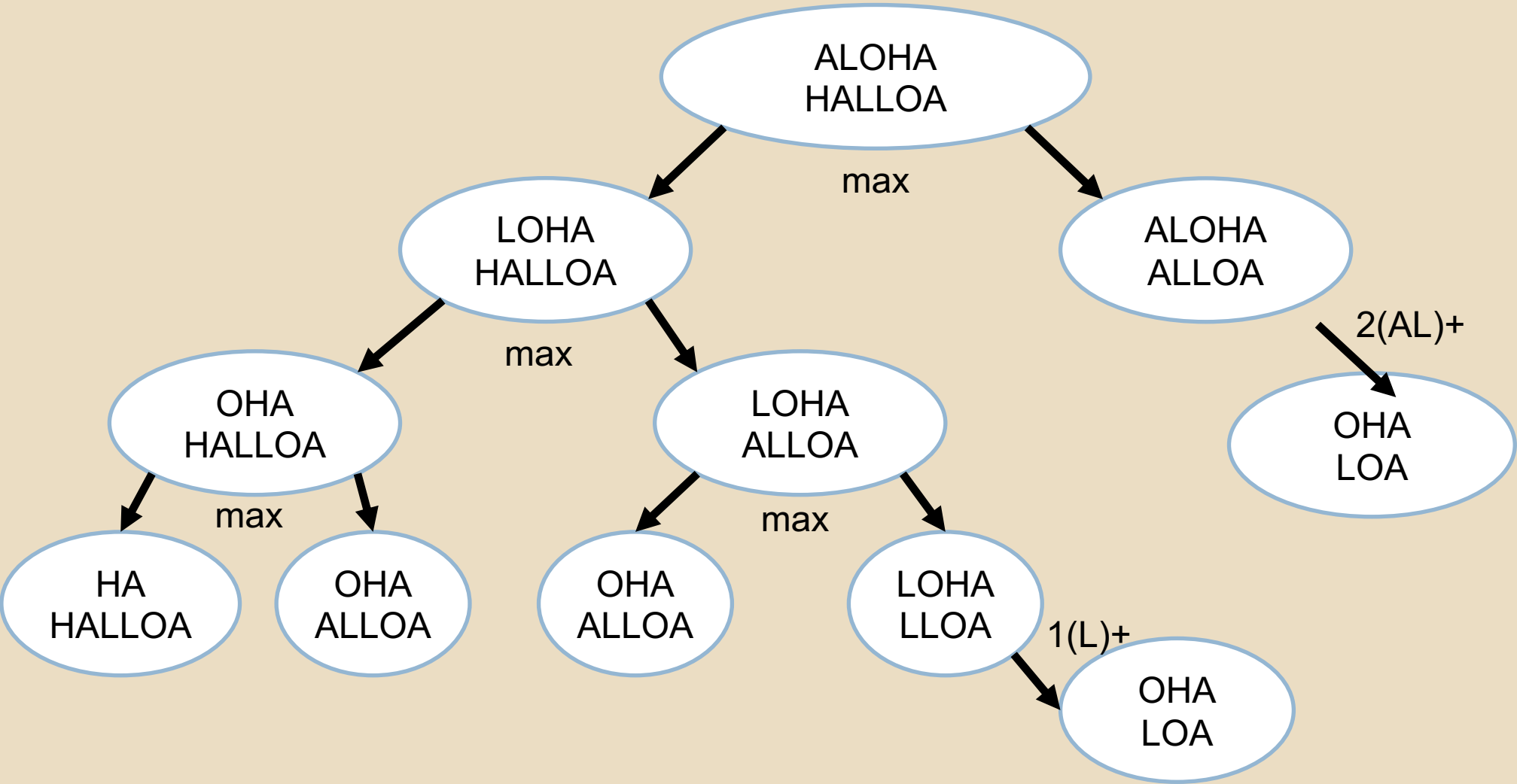
SUBPROBLEMS

- Say we start with words $u_1 \ldots, u_n$

$$v_1, \ldots, v_m$$

- 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 $u_1, \ldots, u_n$
$$v_1, \ldots v_m$$

- In recursive calls, we recurse on: $u_I, \ldots, u_n, I = 1 \ldots n + 1$
$$\text{to: } v_J, \ldots, v_m, J = 1 \ldots m + 1$$

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

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

BT ALGORITHM

- $LCS(u_1, \ldots, u_n \, ; \, v_1, \ldots, v_m)$
- $LCS(u_I, \ldots, u_n \, ; \, v_J, \ldots, v_m)$
  - IF $n = 0$ or $m = 0$ return 0
  - IF   or   return 0
  - IF $u_1 = v_1$ return 1+ $LCS(u_2, \ldots, u_n \, ; \, v_2, \ldots, v_m)$
  - IF   return 1+ LCS (;   )
  - ELSE return max($LCS(u_2, \ldots, u_n \, ; \, v_1, \ldots, v_m)$,
                    $LCS(u_1, \ldots, u_n \, ; \, v_2, \ldots, v_m)$)
  - ELSE return max(LCS(;   )
                    LCS(;   ))

BT ALGORITHM

- LCS($u_I, \ldots, u_n;\ v_J, \ldots, v_m$)
- 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($u_{I+1}, \ldots, u_n;\ v_{J+1}, \ldots, v_m$)
  - IF $u_I = v_J$ THEN L[I, J]:= 1+
    - ELSE return max(LCS($u_{I+1}, \ldots, u_n;\ v_J, \ldots, v_m$),
    - LCS($u_I, \ldots, u_n;\ v_{J+1}, \ldots, v_m$)
  - ELSE L[I, J] : = max (L[,], L[,])

FINAL RECURRENCES

- $L[I, J] \equiv$ 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($u_1, \ldots, u_n; \ v_1, \ldots, v_m$)
  - Initialize L$[1 \ldots n + 1, 1 \ldots m + 1]$
  - 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*"
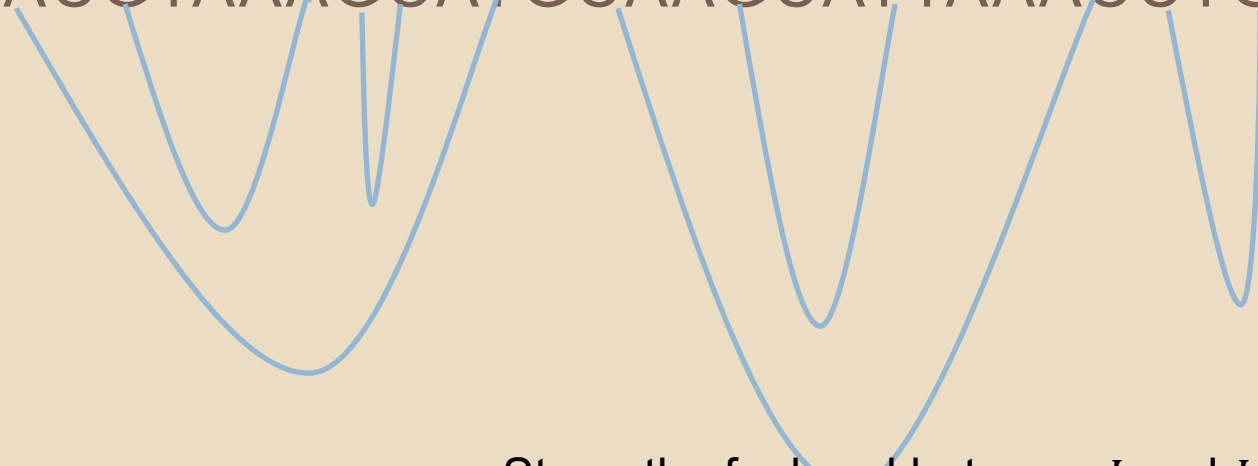
SECONDARY STRUCTURE
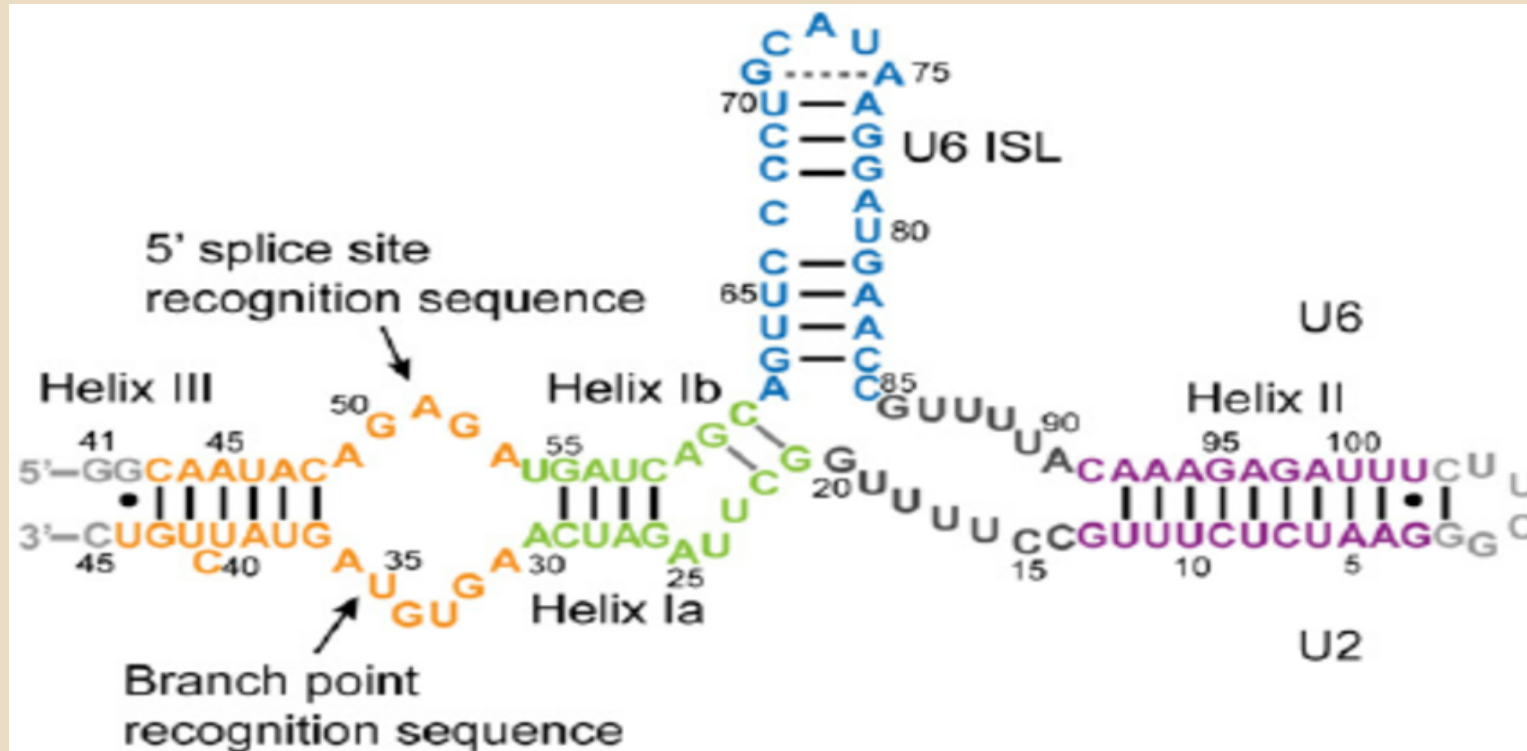
RNA folds back on itself, forming chemical bonds between amino acids in the sequence

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 $2 \dots I - 1$ and $I + 1 \dots n$

- BTSS$(w_1 \dots w_n)$
-     IF $n = 0$ or $n = 1$ return 0
-     Max:= BTSS$[w_2, \dots w_n]$ *//(case when w_1 unbonded)*
-     FOR $I = 2$ to n do:
-        THISCASE:= strength(w_1, w_I) + BTSS$(w_2, \dots, w_{I-1})$ + BTSS$(w_{I+1}, \dots, w_n)$
-        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]: $O(n^{2.86\dots})$ *time algorithm for RNA secondary structure, using speeded-up min-plus product and improved matrix multiply algorithms*