# COL702: Backtracking and Dynamic Programming

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

# SEARCH AND OPTIMIZATION PROBLEMS

Many problems involve finding the best solution from among a large space of possibilities.

- **Instance**: What does the input look like?
- **Solution format**: What does an output look like?
- **Constraints**: What properties must a solution have?
- **Objective function**: What makes a solution better or worse?

# GLOBAL SEARCH VS LOCAL SEARCHES

- Like greedy algorithms,

  **backtracking algorithms break the massive global search for a solution, into a series of simpler local searches.**

  "Which edge do we take first?  Then second? …”

- Unlike greedy algorithms, which guess the best local choice and only consider this possibility,

  **backtracking uses exhaustive search to try out all combinations of local decisions.**

# GLOBAL SEARCH VS LOCAL SEARCHES

- However, we can often use the **constraints** of the problem to **prune** cases that are dead ends.  Applying this recursively, we get a substantial savings over exhaustive search.

- This might take a long time to do. What are some other ideas in general?

# BACKTRACKING: PROS AND CONS

**The good:**

Very general, applies to almost any search problem

Can lead to exponential improvement over exhaustive search

Often better as heuristic than worst-case analysis

FIRST STEP TO DYNAMIC PROGRAMMING

**The bad:**

Since it works for very hard problems, usually only improved exponential time, not poly time

Hard to give exact time analysis

# MAXIMAL INDEPENDENT SET

Given a graph with nodes representing people, with an edge between any two people who are enemies, find the largest set of people such that no two are enemies. In other words, given an undirected graph, find the largest set of vertices such that no two are joined by an edge.

# MAXIMAL INDEPENDENT SET

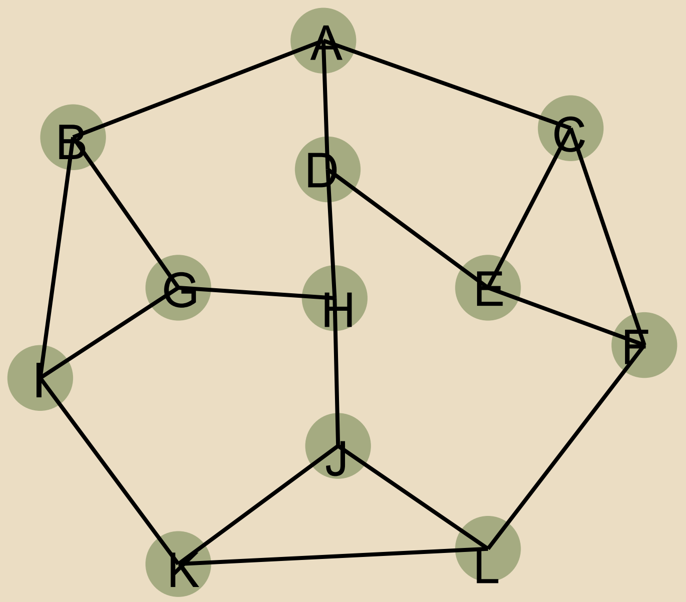
Given a graph with nodes representing people, with an edge between any two people who are enemies, find the largest set of people such that no two are enemies. In other words, given an undirected graph, find the largest set of vertices such that no two are joined by an edge.

- **Instance:**
- **Solution format:**
- **Constraint:**
- **Objective:**

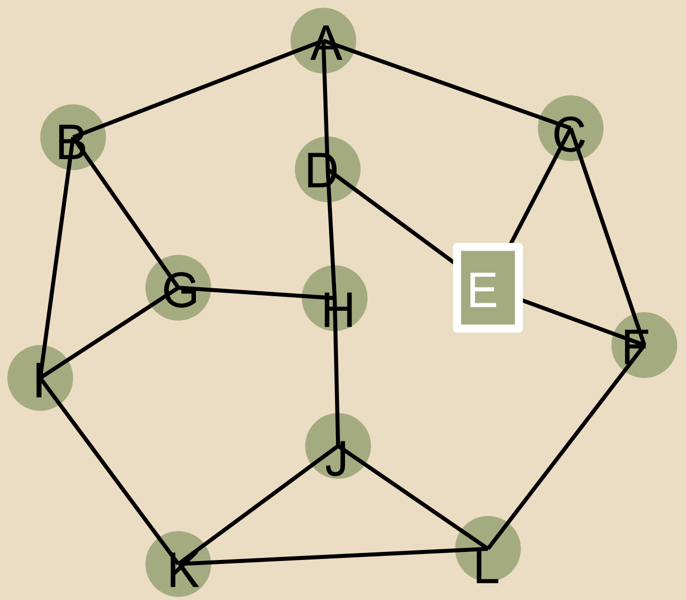
# MAXIMAL INDEPENDENT SET

- Greedy approaches?
- One may be tempted to choose the person with the fewest enemies, remove all of his enemies and recurse on the remaining graph.
- This is fast, but does not always find the best solution.

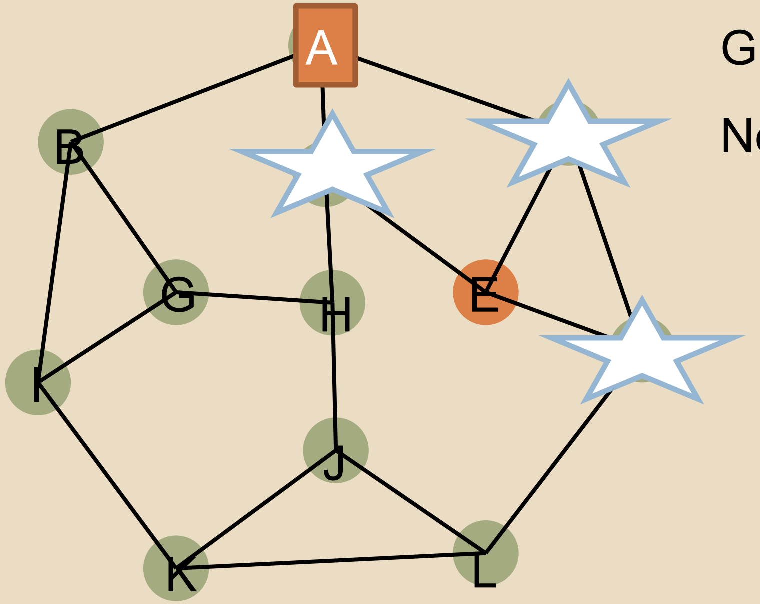
# AN EXAMPLE



# AN EXAMPLE



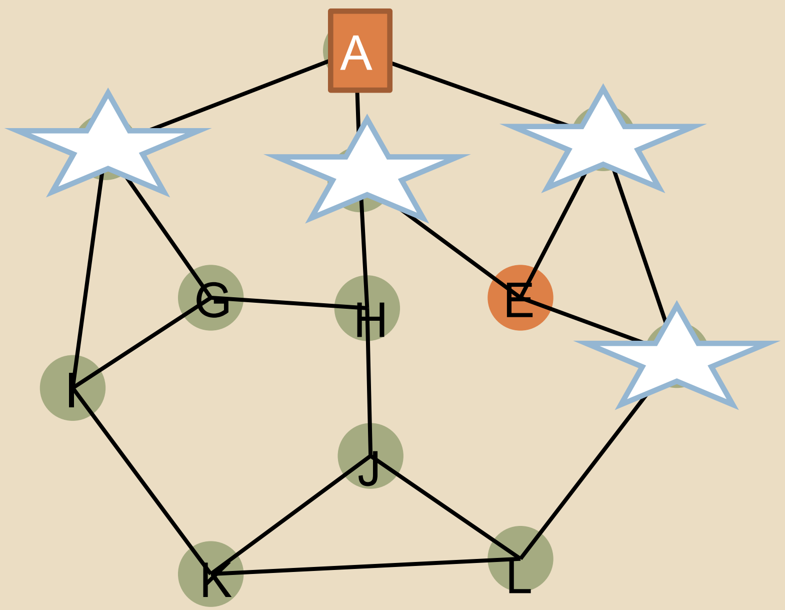
# AN EXAMPLE




Greedy: all degree 3, pick any, say E

Neighbors (enemies) of E forced out of set

# AN EXAMPLE

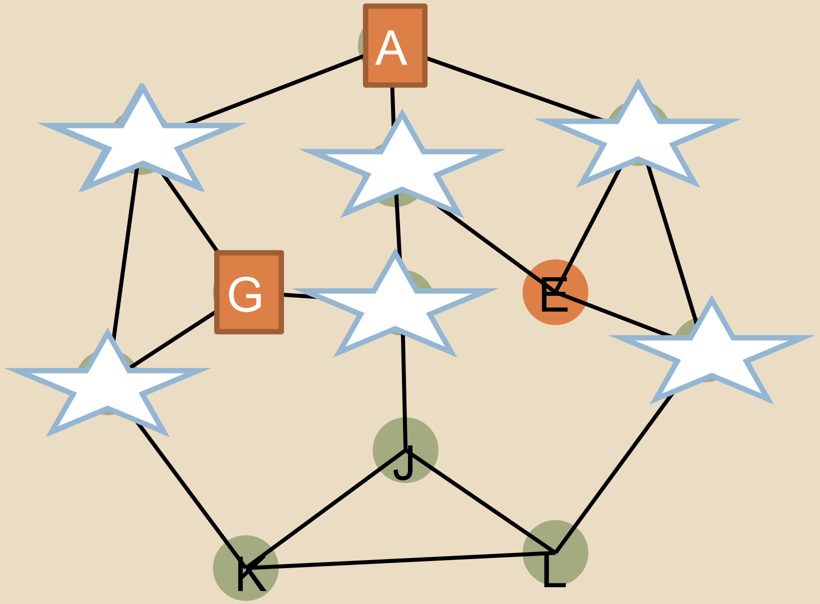



Greedy: all degree 3, pick any, say E

Neighbors (enemies) of E forced out of set

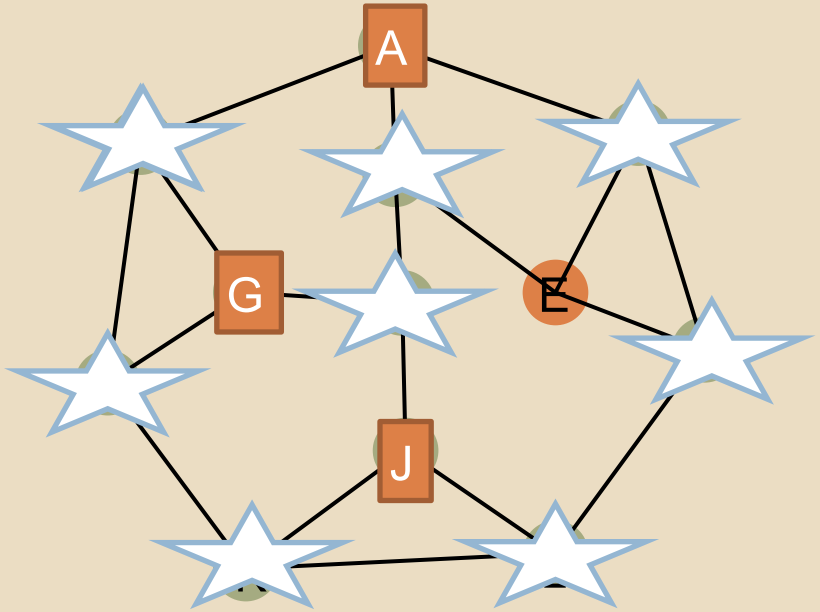
Lowest degree is now A

# AN EXAMPLE




Many degree  2 vertices we could choose next, say G

# AN EXAMPLE

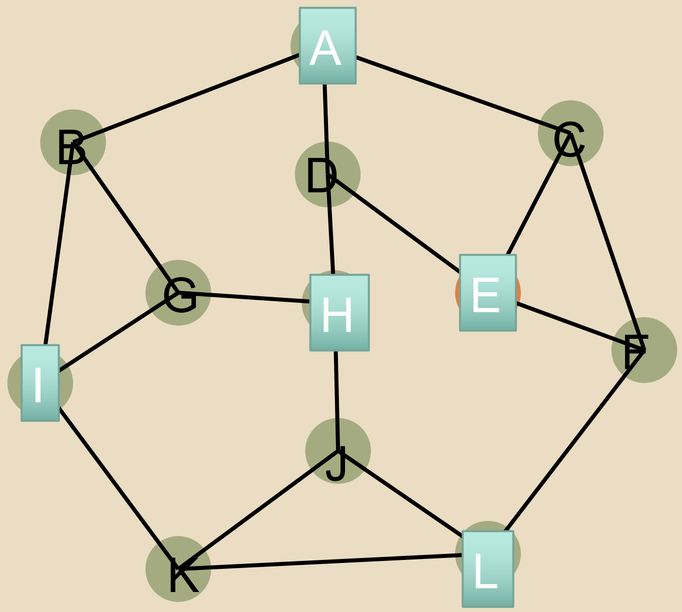



Many degree 2 vertices we could choose next, say G

Can pick any remaining one

Solution found by greedy is size 4

# BETTER SOLUTION



# MAXIMAL INDEPENDENT SET

- What is the solution space?
- How much is exhaustive search?
- What are the constraints?
- What is the objective?

# MAXIMAL INDEPENDENT SET

- What is the solution space?

All subsets S of V

- How much is exhaustive search?

2^{|V|}

- What are the constraints?

For each edge e={u,v}, cannot have both u and v in S

- What is the objective?

|S|

# MAXIMAL INDEPENDENT SET

- Backtracking: Do exhaustive search locally. Use constraints to simplify problem along the way.

- What is a local decision? Do we pick vertex E or not…..

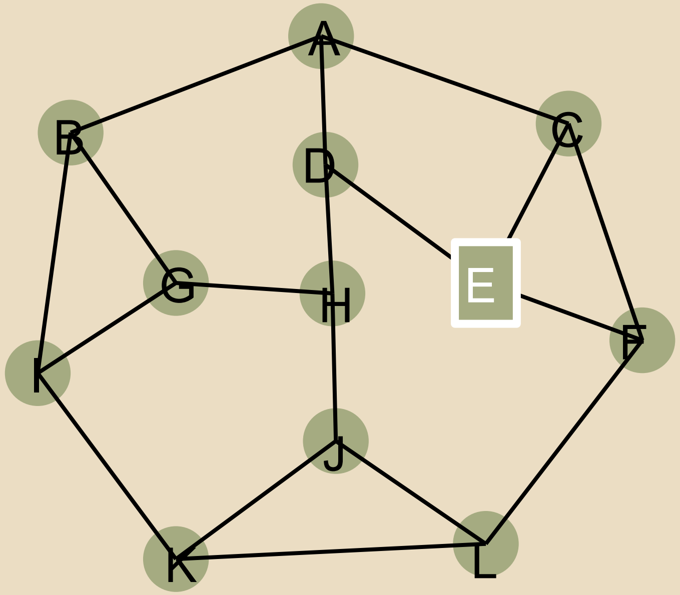
- What are the possible answers to this decision? Yes or No

- How do the answers affect the problem to be solved in the future?

If we pick E: Recurse on subgraph $G - \{E\} - \{E\text{'s neighbors}\}$ (and add 1)

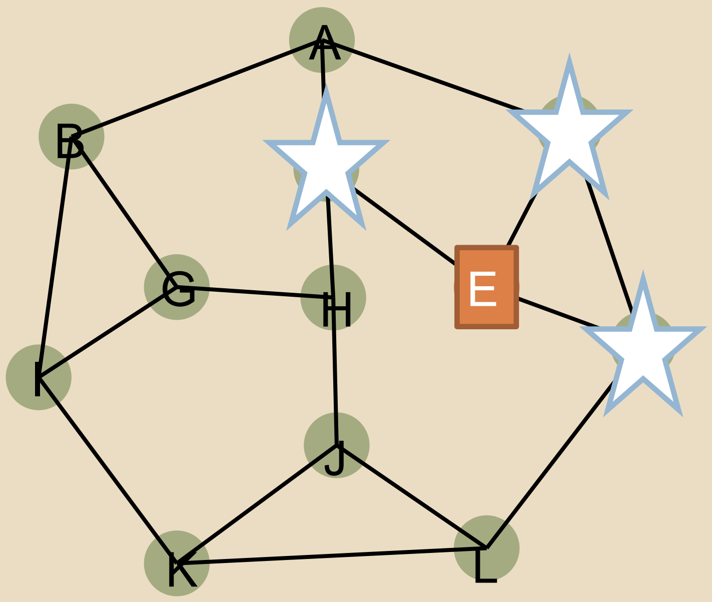
If we don't pick E: Recurse on subgraph $G - \{E\}$.

# AN EXAMPLE



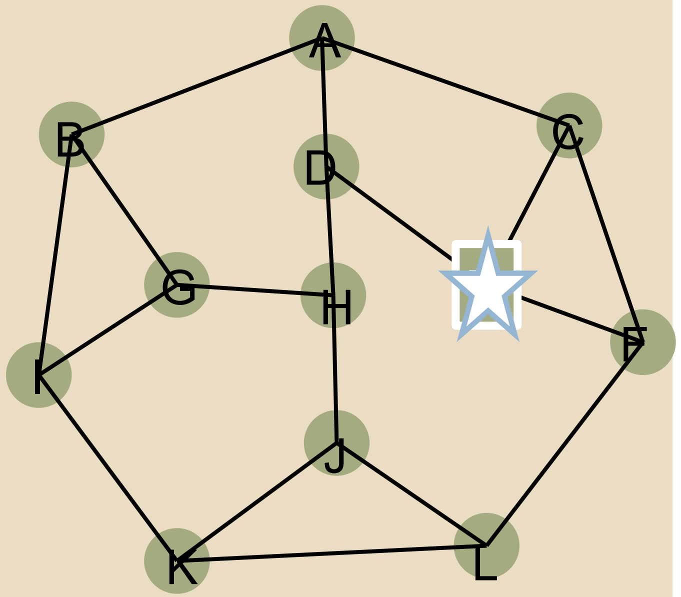

Local decision : Is E in S?
Possible answers: Yes, No

# AN EXAMPLE

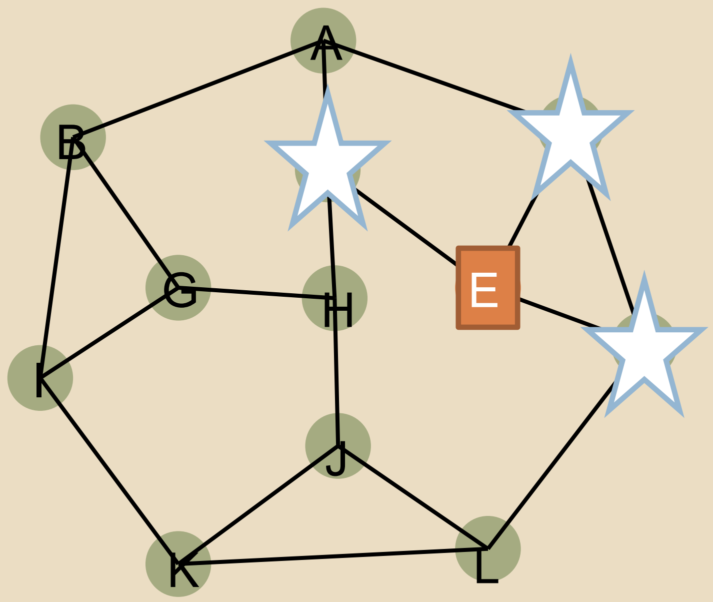



Local decision : Is E in S?
YES OR NO

MIS([A,B,C,D,E,F,G,H,I,J,K,L])=
(YES) 1 + MIS([A,B, G,H,I,J,K,L])
(NO) MIS([A,B,C,D, ,F,G,H,I,J,K,L])

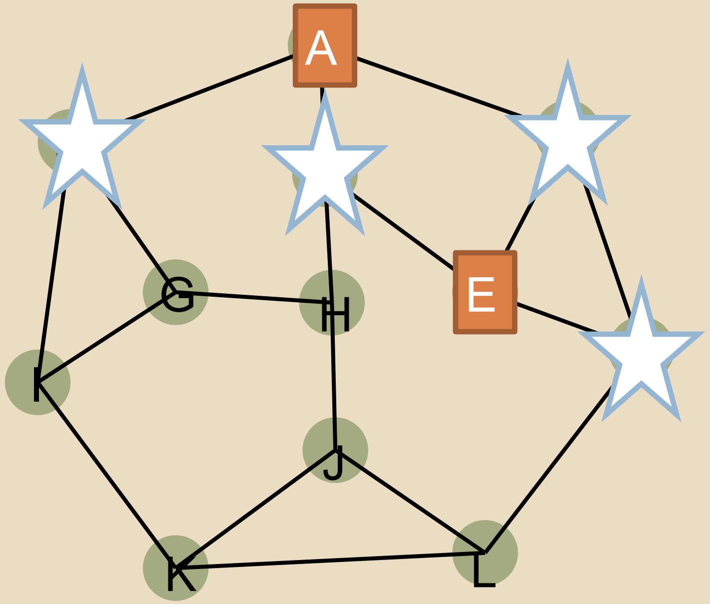


# AN EXAMPLE




Local decision : Is E in S?
Case 1 : Yes
Consequences: Neighbors not in S

# AN EXAMPLE



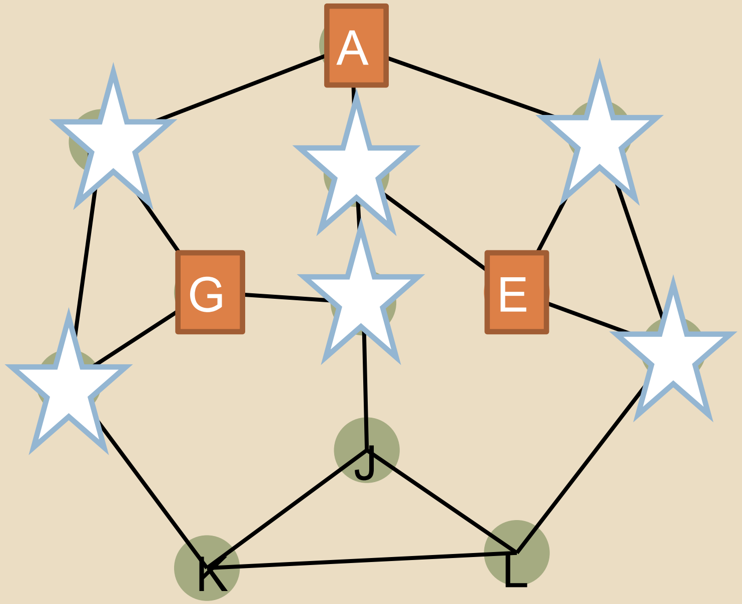

Local decision : Is E in S?
Case 1 : Yes
Consequences: Neighbors not in S

Claim: A is now in some largest IS

Go on to next local decision

Is G in S?

# AN EXAMPLE




Local decision : Is E in S?
Case 1 : Yes
Consequences: Neighbors not in S

Claim: A is now in some largest IS

Go on to next local decision

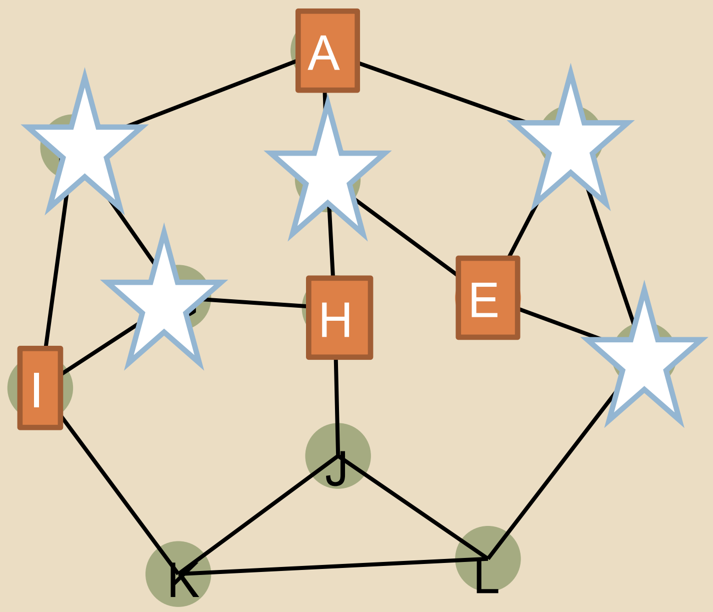
Is G in S?

Case 1a: Yes

Other three symmetrical: Get one more

Best set for Case 1a: 4, e.g, A,G,E,J

# BUT NOW WE BACKTRACK




Local decision : Is E in S?
Case 1 : Yes
Consequences: Neighbors not in S

Claim: A is now in some largest IS

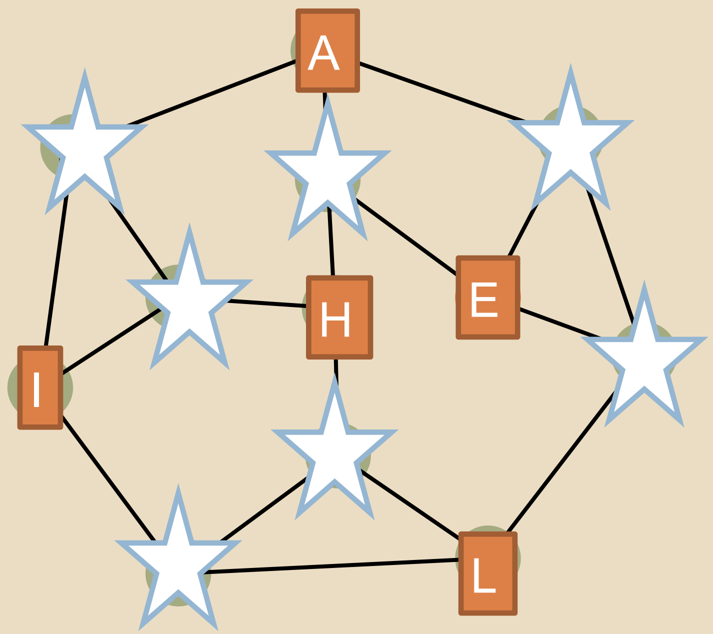
Go on to next local decision

Is G in S?

Case 1b: No

Claim: I, H in some smallest MIS in Case 1b

# BUT NOW WE BACKTRACK




Local decision : Is E in S?
Case 1 : Yes
Consequences: Neighbors not in S

Claim: A is now in some largest IS

Go on to next local decision

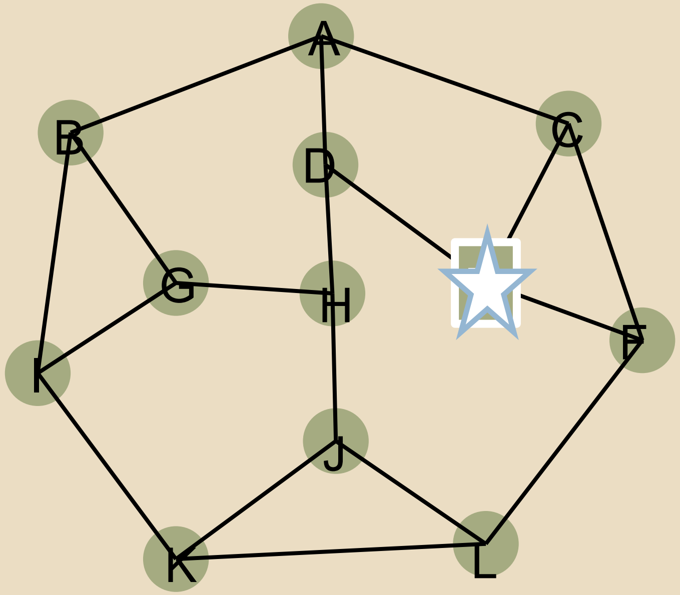
Is G in S?

Case 1b: No

Claim: I, H in some smallest MIS in Case 1b

Case 1 b: Get set of size 5

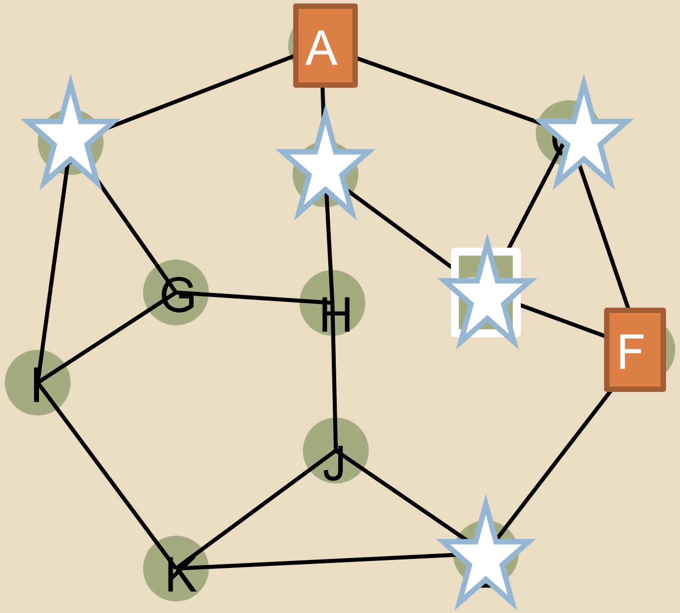
# BACKTRACK AGAIN




Case 1b is better than Case 1a, but we still don't know its optimal

Need to consider Case 2: E is not in S

# BACKTRACK AGAIN




Case 1b is better than Case 1a, but we still don't know its optimal

Need to consider Case 2: E is not in S

Case 2a:  A is in S

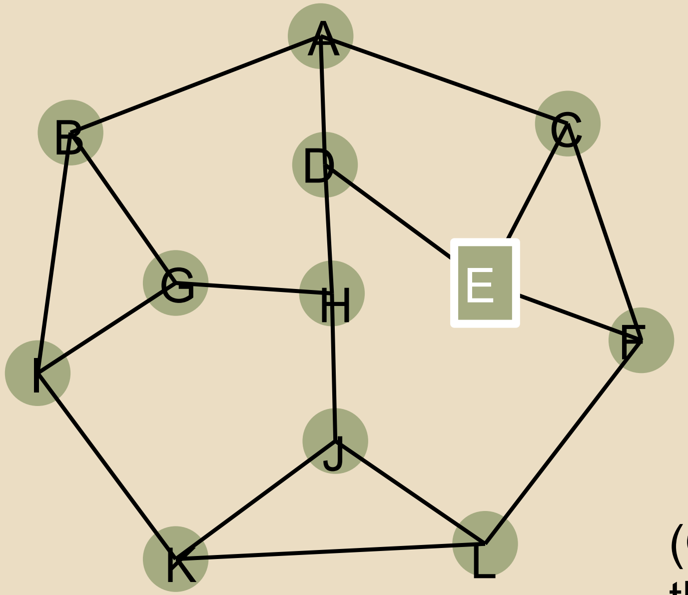
F is in S

Cycle of 5 : get 2

So this case eventually gets 4

Now we KNOW Case 1b is best

# AN EXAMPLE




12 vertices means 4096 subsets

But in the end, we only needed 4 cases

(OK, I used some higher principles, e.g. symmetry that our BT algorithm might not have)

# CASE ANALYSIS AS RECURSION

MIS1(G= (V,E))

- IF |V|=0 return the empty set
- Pick vertex v
- S_1:= v + MIS1(G-v-N(v))
- S_2: = MIS1(G-v)
- IF |S_2| > |S_1| return S_2, else return S_1

# CORRECTNESS

MIS1(G= (V,E))

- IF |V|=0 return the empty set
- Pick vertex v
- S_1:= v + MIS1(G-v-N(v))
- S_2: = MIS1(G-v)
- IF |S_2| > |S_1| return S_2, else return S_1

Induction on n. Base case n=0: MIS1 correctly returns empty set.
Otherwise, use strong induction: S_1 is max ind set containing v, S_2 max ind. set not containing v. Better of two is MIS in G.

# TIME ANALYSIS

MIS1(G= (V,E))

- IF |V|=0 return the empty set
- Pick vertex v
- S_1:= v + MIS1(G-v-N(v))
- S_2: = MIS1(G-v)
- IF |S_2| > |S_1| return S_2, else return S_1

# TIME ANALYSIS

MIS1(G= (V,E))

- IF |V|=0 return the empty set
- Pick vertex v:
- S_1:= v + MIS1(G-v-N(v)) Worst-case: T(n-1)
- S_2: = MIS1(G-v) T(n-1)
- IF |S_2| > |S_1| return S_2, else return S_1 poly(n)

T(n)= 2 T(n-1) + poly(n)

- Idea: bottom-heavy, so exact poly(n) doesn't affect asymptotic time
- T(n)=2^n

# WHAT IS THE WORST CASE FOR MIS1?

# WHAT IS THE WORST CASE FOR MIS1?

- An empty graph with no edges, i.e., the whole graph is an independent set
- But then we should just return all vertices without trying cases
- More generally, if a vertex has no neighbors, the case when we include it v+MIS(G-v) is always better than the case when we don't include it, MIS(G-v)

# GETTING RID OF THAT STUPID WORST CASE

MIS2($G = (V, E)$)

- IF $|V| = 0$ return the empty set
- Pick vertex $v$
- $S_1 := v + MIS2(G - v - N(v))$
- IF deg($v$)=0 return $S_1$
- $S_2 := MIS2(G - v)$
- IF $|S_2| > |S_1|$ return $S_2$, else return $S_1$

- Correctness: If deg($v$) = 0, $|S_2| < |S_1|$ so we'd return $S_1$ anyway
- So does same thing as MIS1

# WHAT IS THE WORST CASE FOR MIS2?

# WHAT IS THE WORST CASE FOR MIS2?

If the graph is a line and we always pick the end, we recurse on one line of size $n-1$ and one of size $n-2$

$$T(n) = T(n-1) + T(n-2) \ + poly(n)$$

$$T(n) = \ O(Fib(n)) \ = O(2^{0.7n})$$

Still exponential but for medium sized n, makes huge difference

$n = 80$: $2^{56}$ = minute of computer time, $2^{80}$ = 16 million minutes

# CAN WE DO BETTER?

In the example, we actually argued that we should add vertices of degree 1 as well.

Modify-the-solution proof:

- Say $v$ has one neighbor $u$.
- Let $S_1$ be the largest independent set with $v$ and let $S_2$ be the largest ind. set without $v$.
- Let $S' = S_2 - \{u\} + \{v\}$. $S'$ is an independent set, and is at least as big as $S_2$, and contains $v$. Thus, $S_1$ is at least as big as $S'$, which is at least as big as $S_2$. So don't bother computing $S_2$ in this case.

# IMPROVED ALGORITHM

MIS3($G = (V, E)$)

- IF $|V| = 0$ return the empty set
- Pick vertex $v$
- $S_1 := v + MIS3(G - v - N(v))$
- IF deg($v$)=0 or 1 return $S_1$
- $S_2 := MIS3(G - v)$
- IF $|S_2| > |S_1|$ return $S_2$, else return $S_1$

Correctness: If deg($v$) =0 or 1 $|S_2|$ is at most $|S_1|$, so we'd return $S_1$ anyway, so does same thing as MIS1

# TIME ANALYSIS

- $T(n)$ is at most $T(n-1) + T(n-3)$ + small amount

- Similar to Fibonacci numbers, but a bit better, about
- $2^{0.6n}$ rather than $2^{0.7n}$.

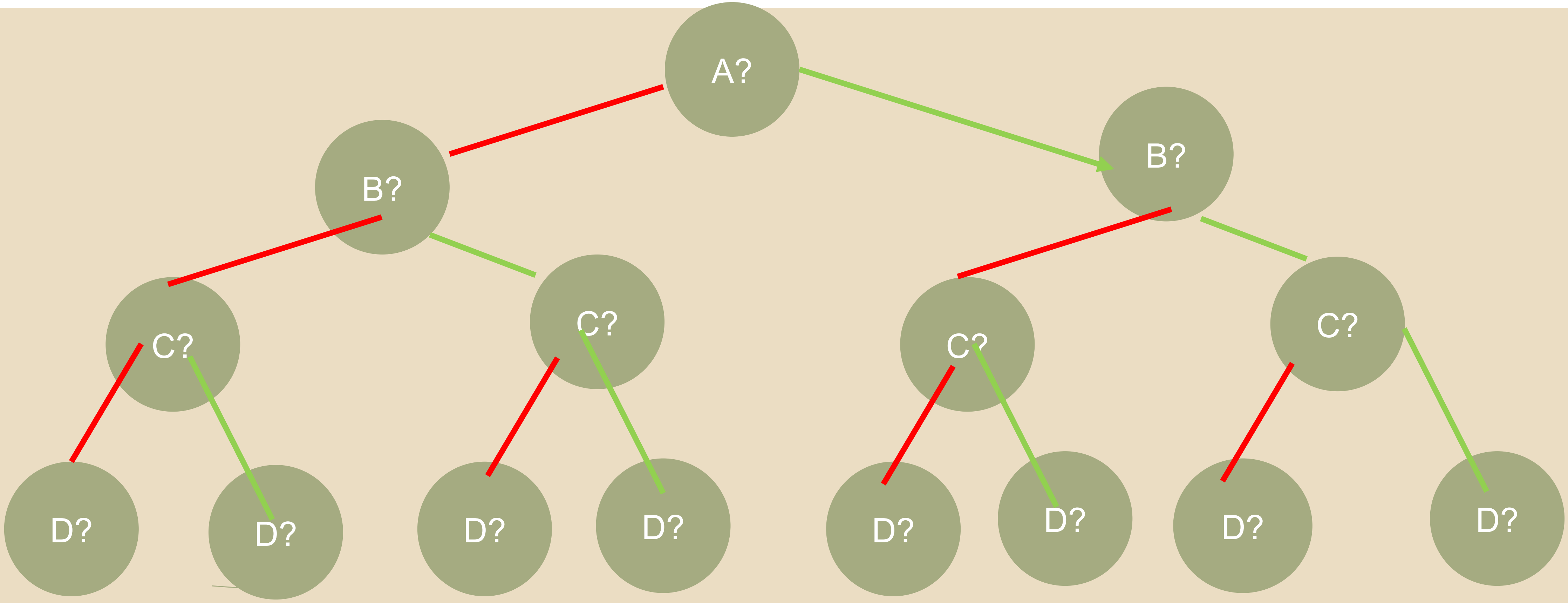
- $n = 80$: $2^{0.6n}$=$2^{48}$, less than a second.
- $n = 100$: $2^{60}$ = 16 minutes, $2^{70}$=16,000 minutes
- So while still exponential, big win for moderate $n$

# IS THIS TIGHT?

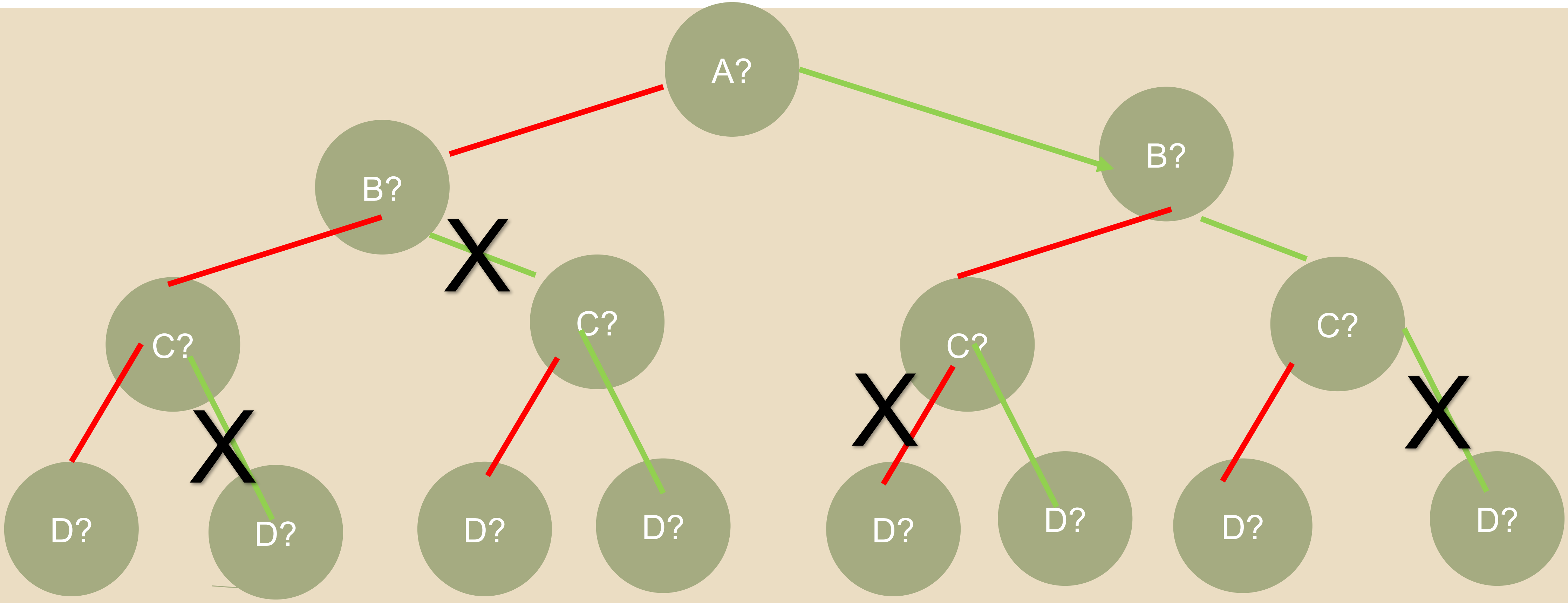
# IS THIS TIGHT?

- I don’t know whether there is any graph where MIS3 is that bad.
- Best known MIS algorithm around $2^{n/4}$ by Robson, building on Tarjan and Trojanowski. Does much more elaborate case analysis for small degree vertices
- Interesting research question: is there a limit to improvements?
- This question= Exponential Time Hypothesis, has interesting ramifications whether true or false

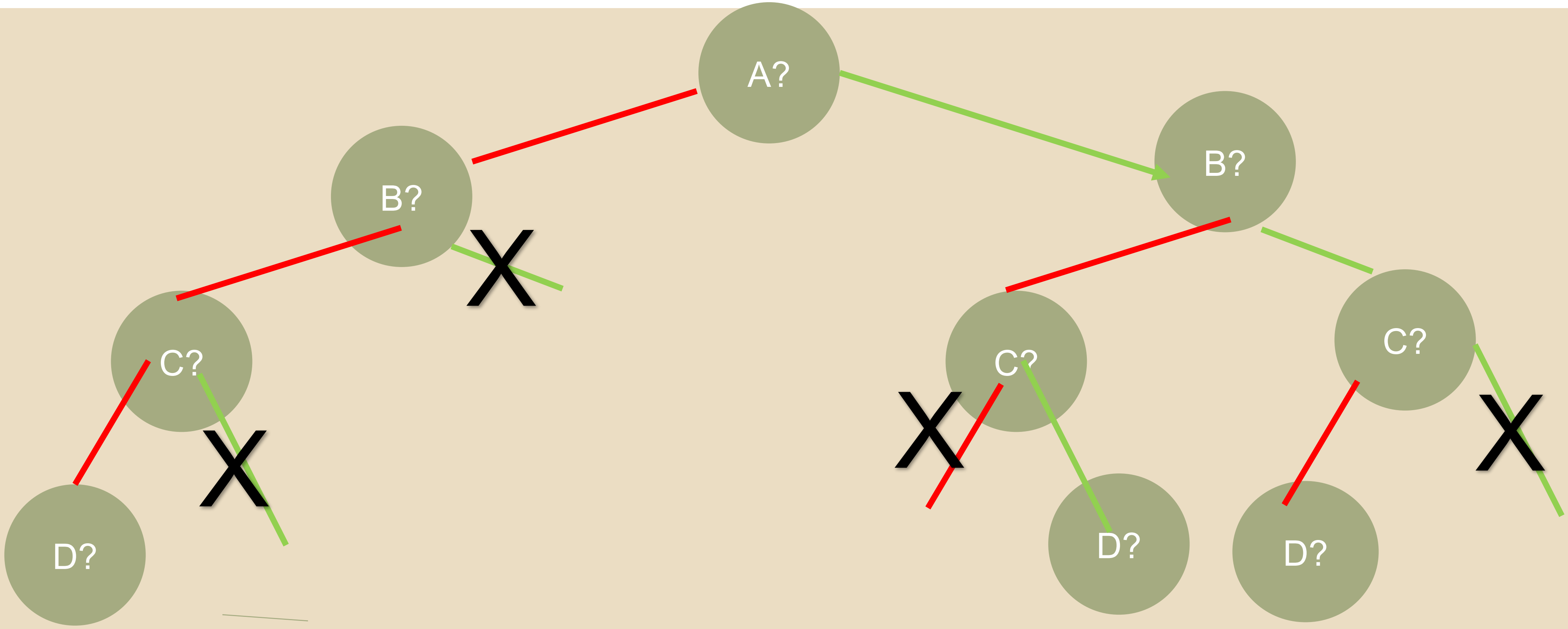
# HOW BACKTRACKING HELPS



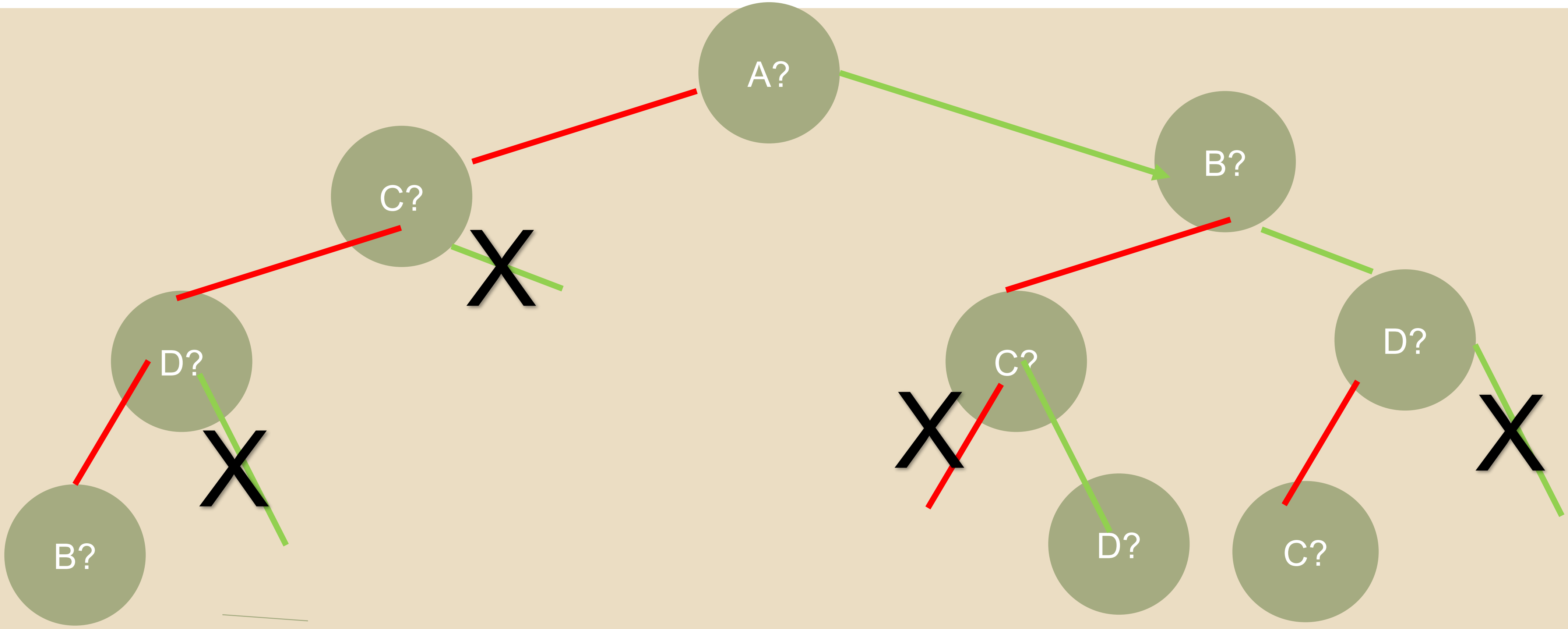
# HOW BACKTRACKING HELPS



# HOW BACKTRACKING HELPS



# ORDER CAN BE ADAPTIVE



# WHEN CAN WE PRUNE?

- Basic: when constraints would be violated
- Subtler: when that choice is dominated by another; the other choice is at least as likely to lead to a (good) solution (Need "modify-the-solution" argument)
- Branch-and-bound: dynamically track "best-so-far" solution. If current path cannot do better (using some function that bounds the achievable best), then we can prune our path.

# "SELF-SIMILARITY"?

- Self-similarity:  Problem+ choice = smaller problem of same type

- If we have self-similarity, it makes recursion in BT (and hence, DP) very clean.

- But if we don't have it, we can still use BT (and hence DP)
- Generalize the problem to keep partial solution
- Generalized problem will have self-similarity, original becomes special case

# 3-COLORING

- Instance: undirected graph G
- Solution format: Give each vertex v a color C(v) $\in \{R, G, B\}$
- Constraint: If {u,v} is an edge, C(v) ≠ C(u)
- Problem: Existence: is there any 3-coloring of G? (True, False)
- Originally came up in making maps: vertices=countries, colors must be distinct to show borders. Famous 4-color theorem said all planar graphs (including possible maps) can be colored with 4 colors

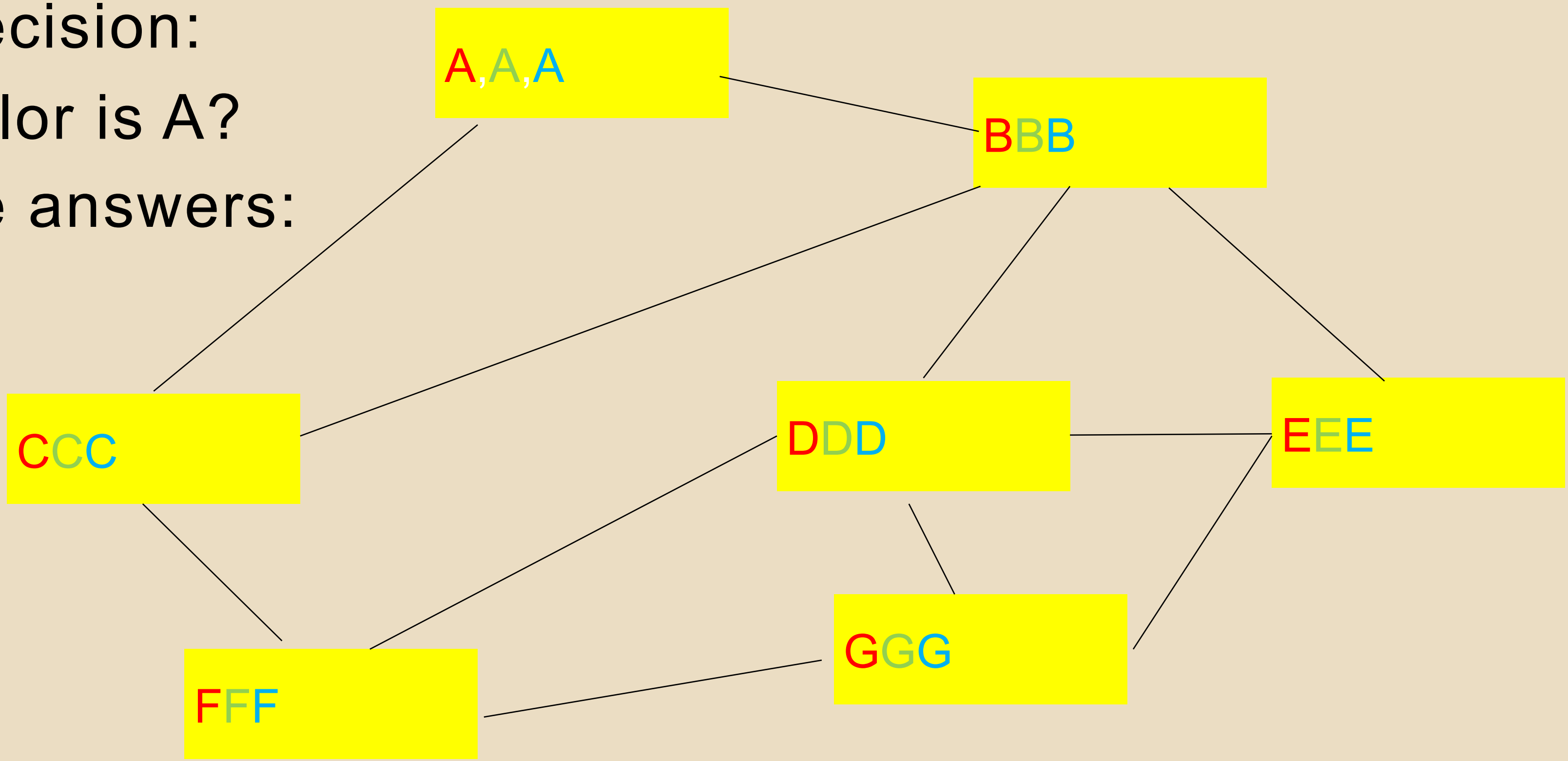
# EXAMPLE

Local decision:

What color is A?

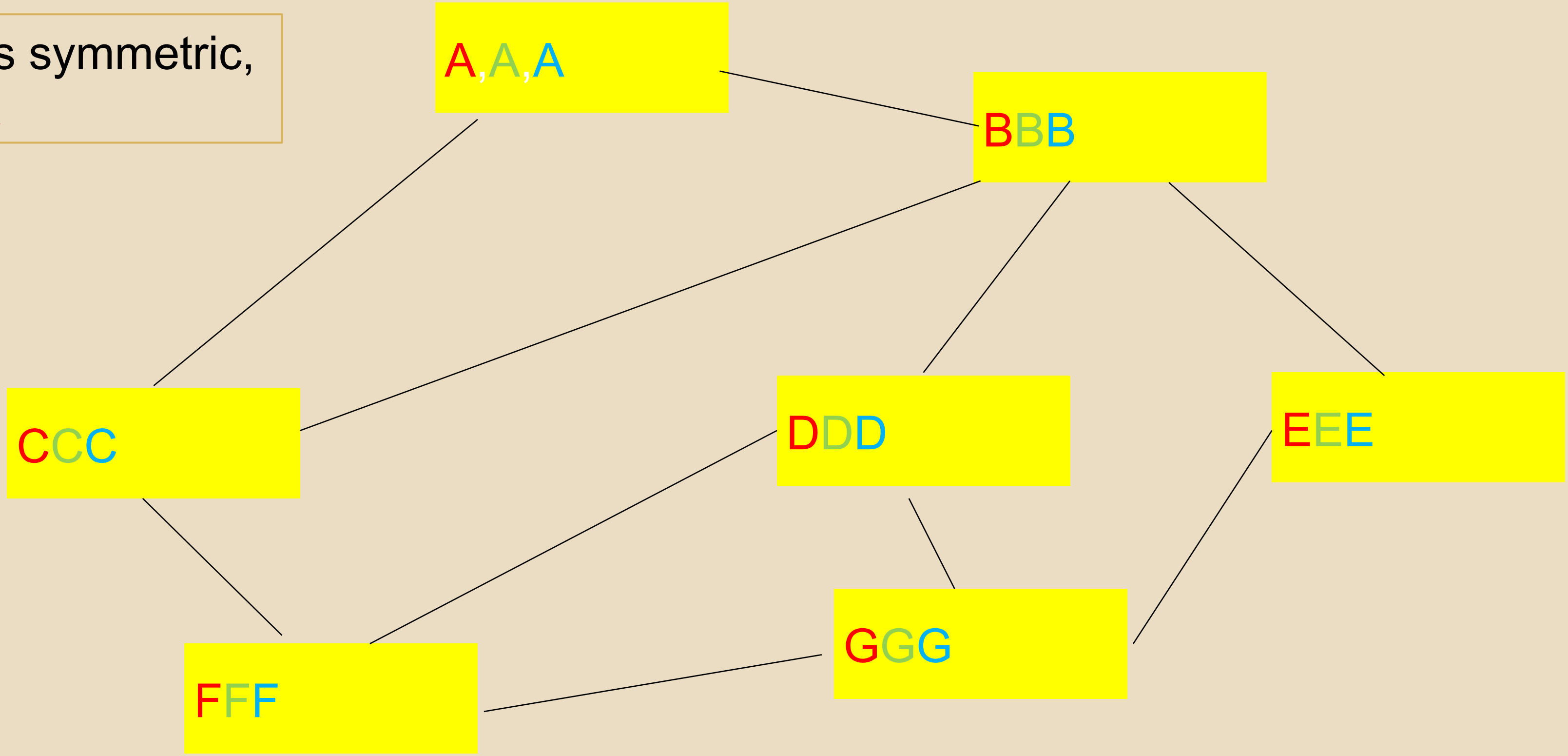
Possible answers:

AAA



# EXAMPLE

All answers symmetric,
Just pick A

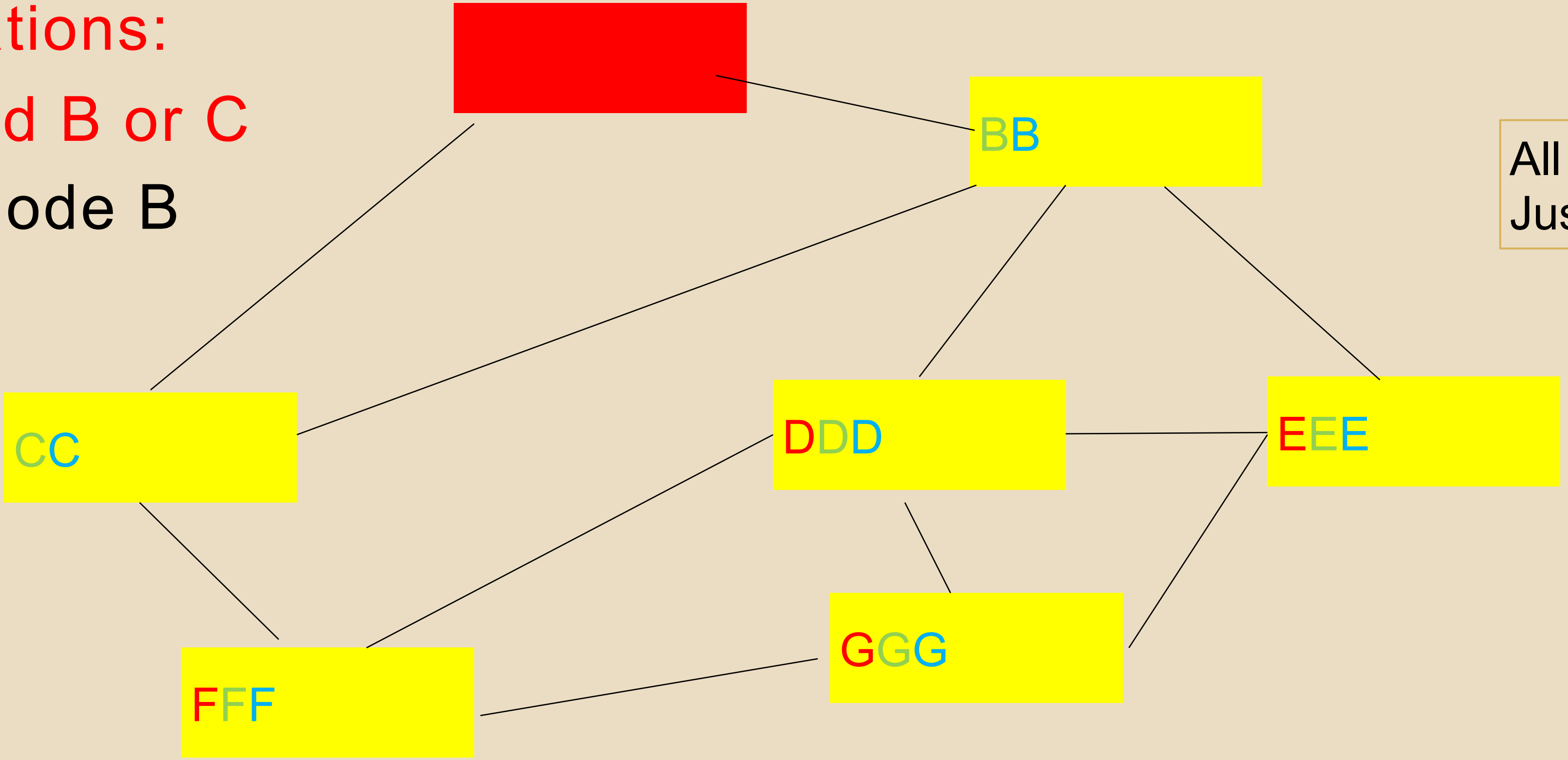


# EXAMPLE

Implications:

- No red B or C

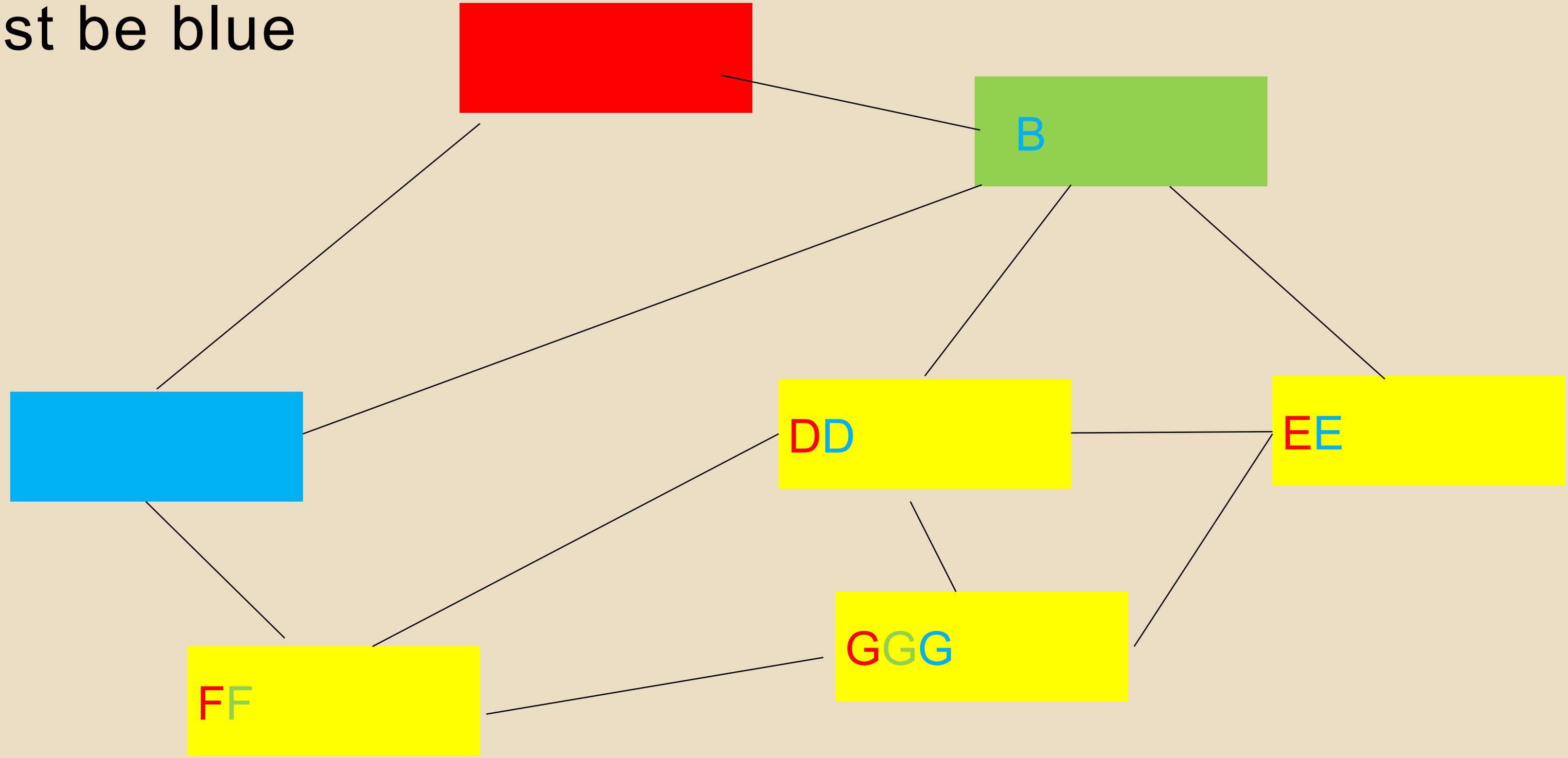
Next: node B




All answers symmetric,
Just pick B

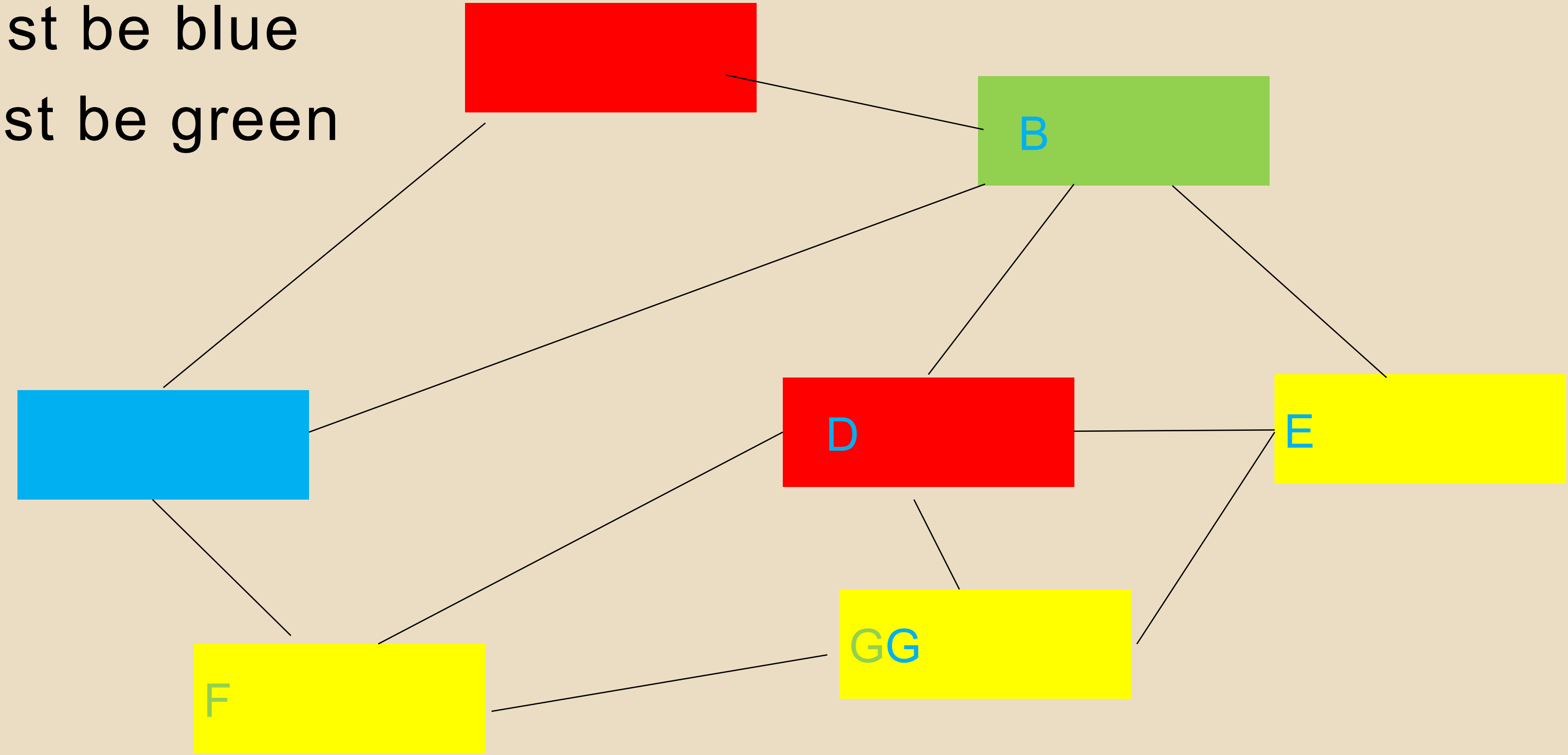
# EXAMPLE

- C must be blue



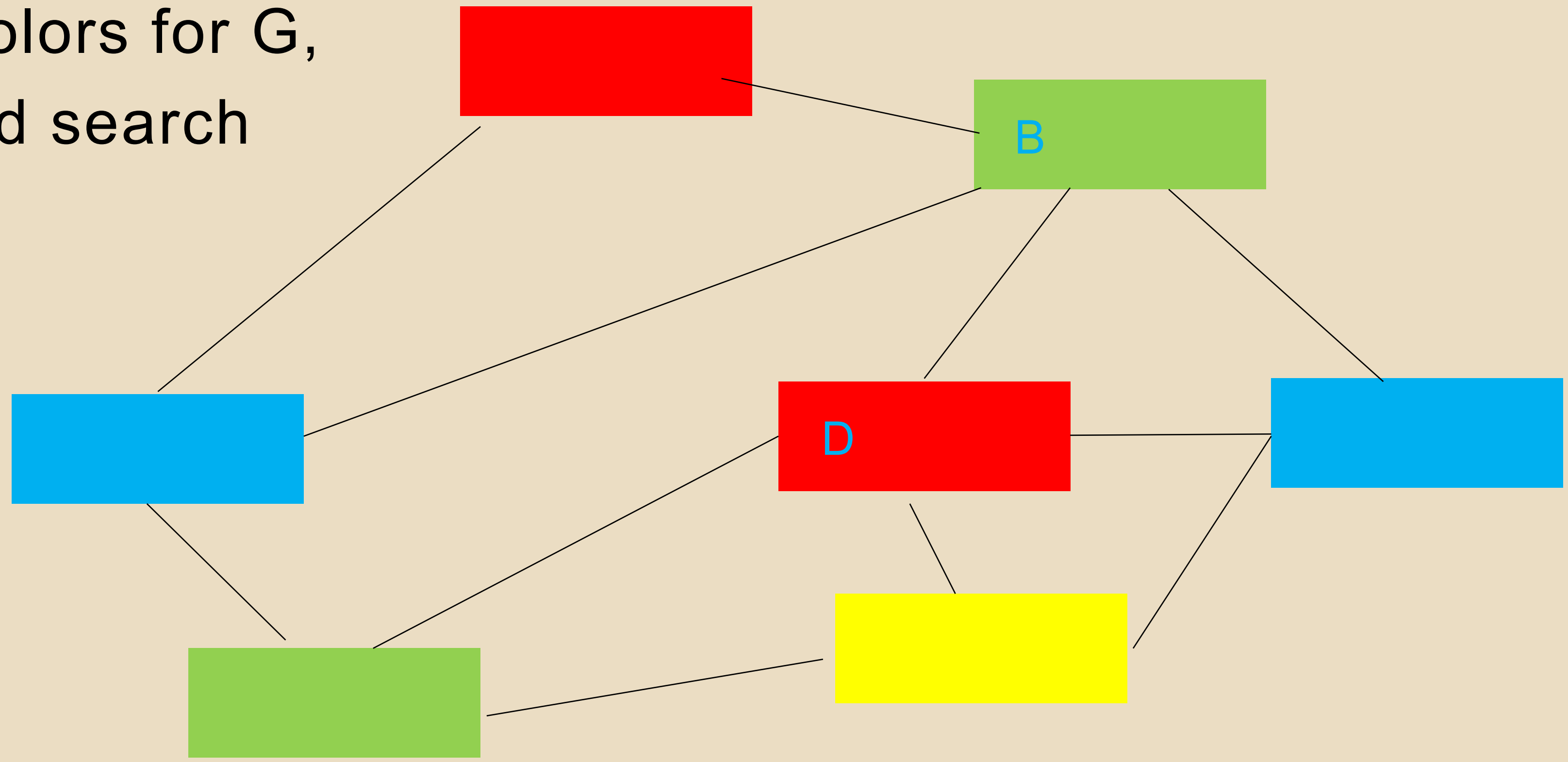
# CASE 1: D (CASE 2: D)

- E must be blue
- F must be green



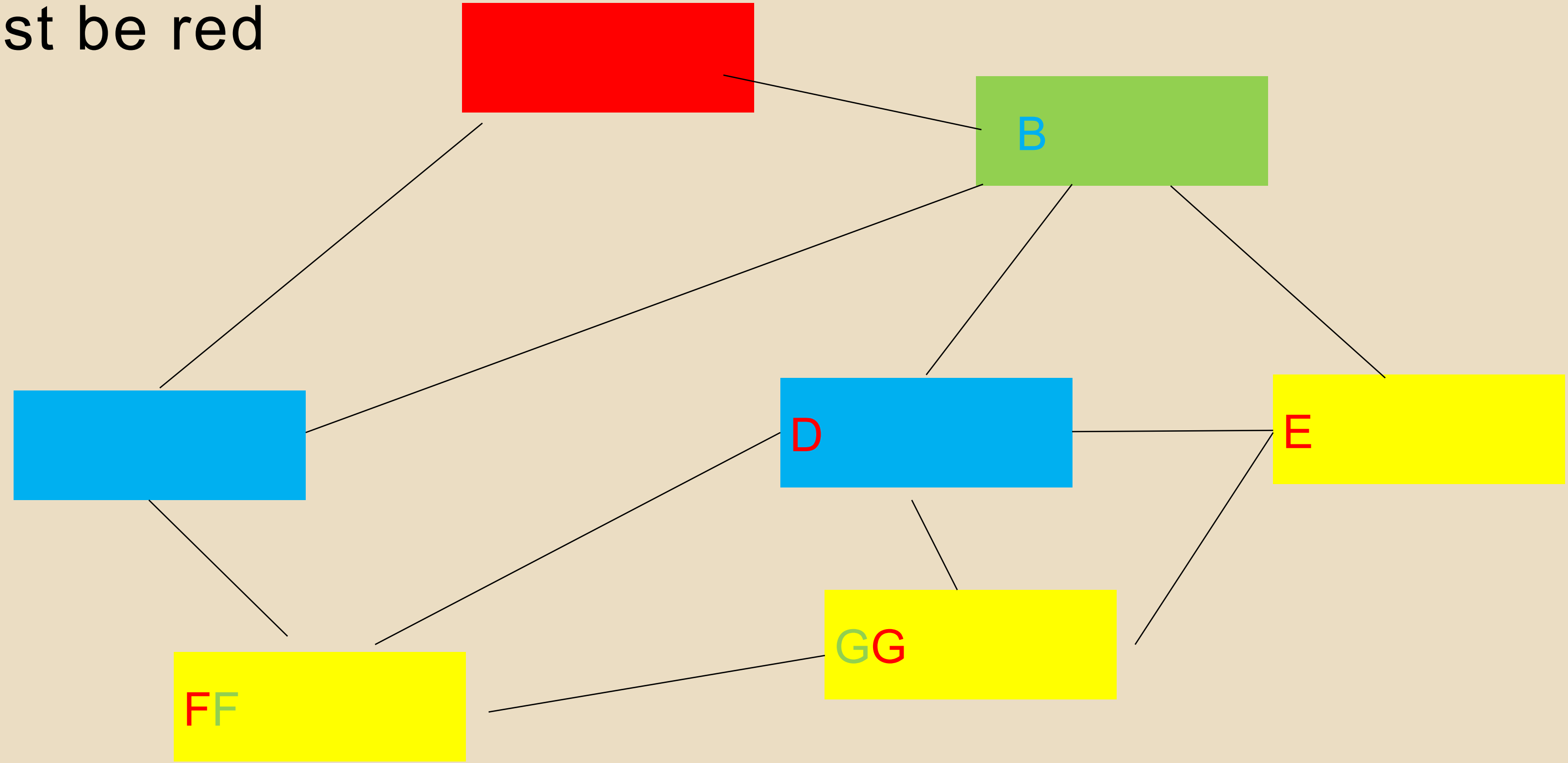
# CASE 1: D (CASE 2: D)

- No colors for G,
- Failed search



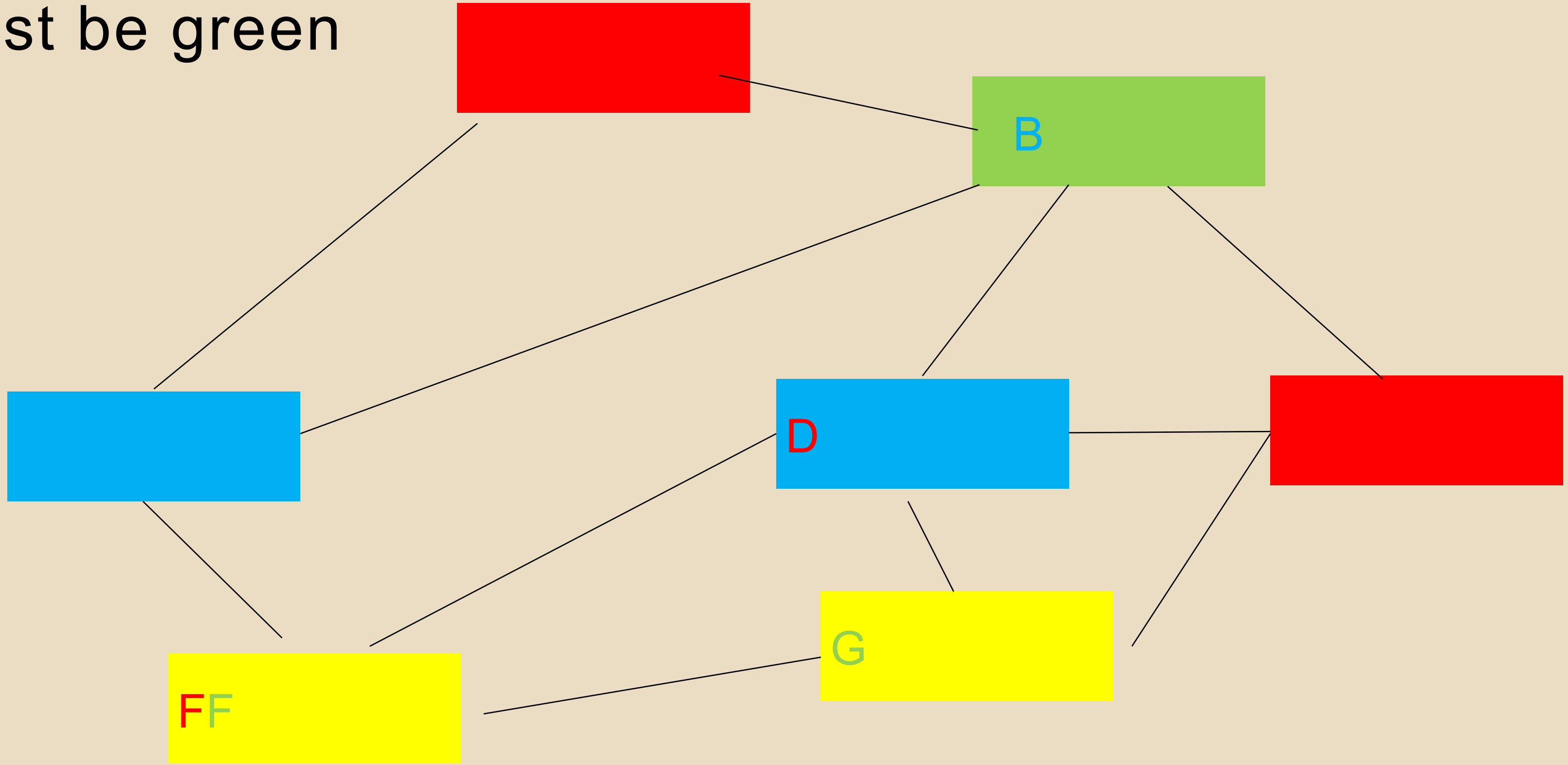
# CASE 2: D

- E must be red



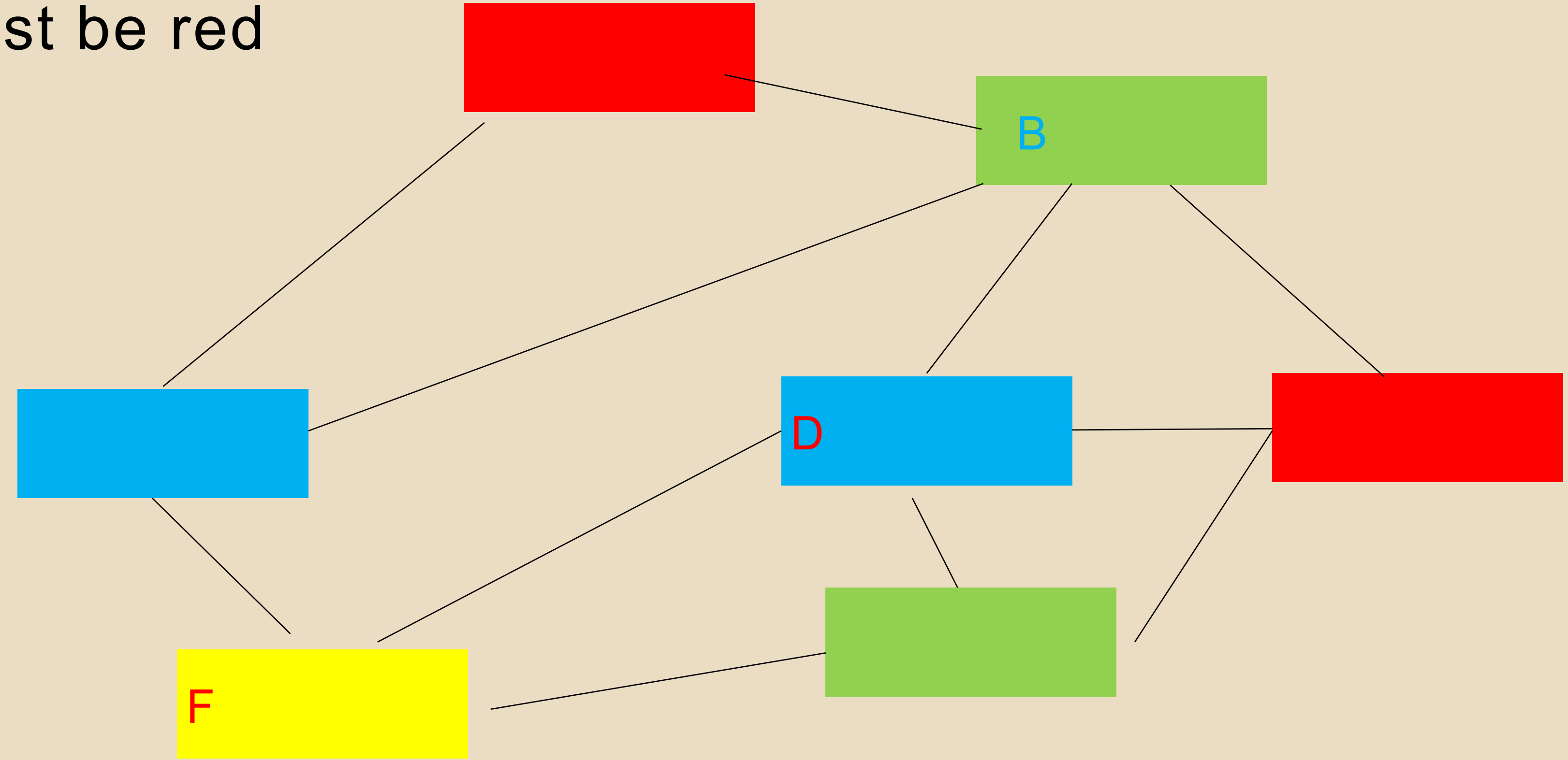
# CASE 2: D

- G must be green



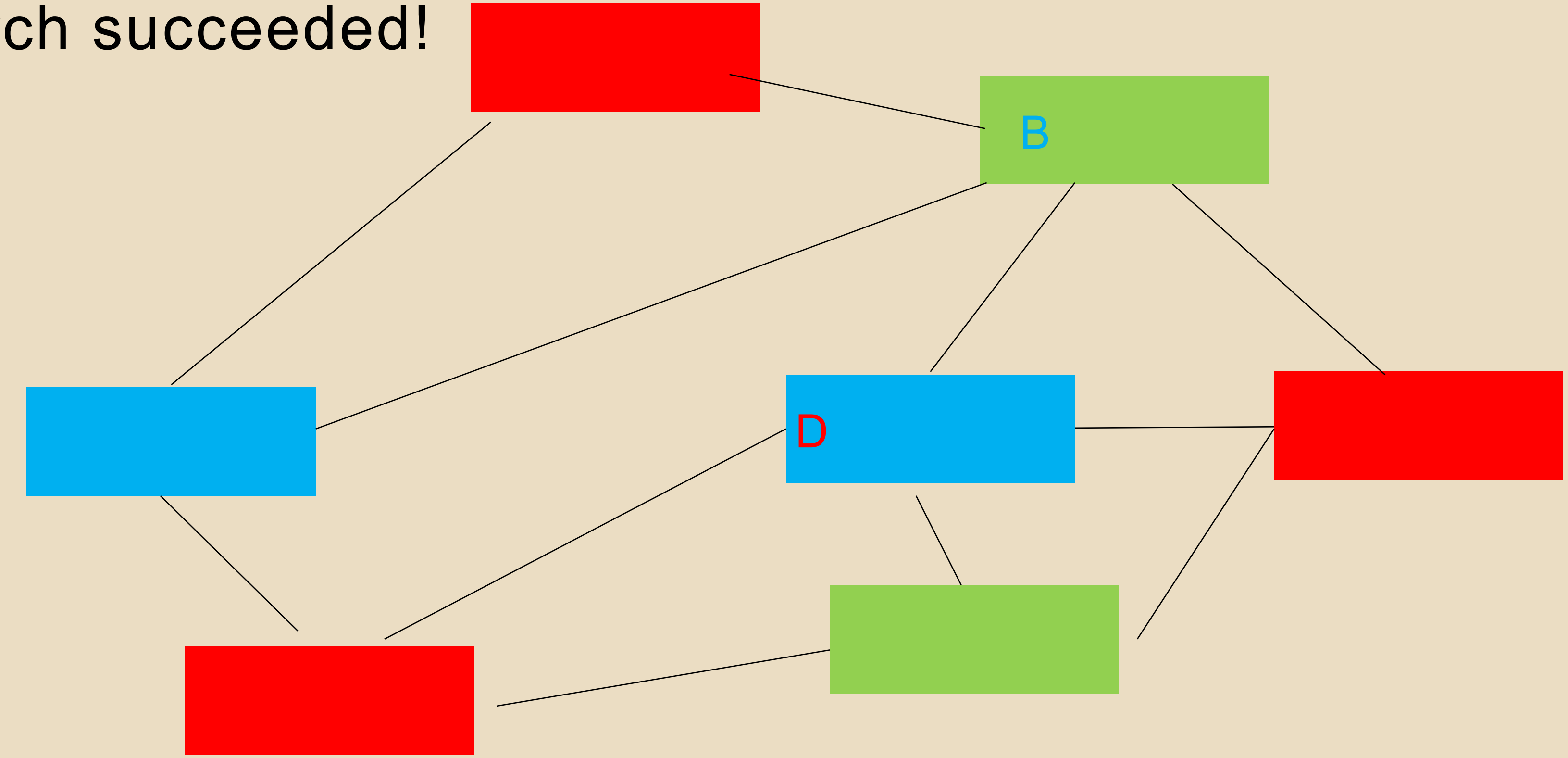
# CASE 2: D

- F must be red



# CASE 2: D

- Search succeeded!



# PARTIAL INFORMATION

- This approach used partial information about the previous solution to generalize the problem so we could solve it recursively

- CL(u)= list of possible colors for vertex u. Initially, CL(u) is

all three colors, but we'll delete colors as we make recursive calls

The list 3-coloring problem, L3C(G,CL), adds the constraints that C(u) must be in CL(u)

# BACKTRACKING ALGORITHM

L3C(G,CL)

- If |V|=0 return True
- If there is any v with |CL(v)|=0 return False
- If there is a v with CL(v)={c}, then :
-     Delete c from CL(u) for each neighbor u of v
-     Return L3C(G-{v},CL)
- If all vertices v have |CL(v)|=3, then pick some v and
-     Delete R from CL(u) for each neighbor u of v
-     Return L3C(G-{v},CL)

# BACKTRACKING ALGORITHM CONTINUED

- Remaining case: the smallest size of CL(u) is 2
- Pick v with CL(v)={c_1,c_2}
- Let CL_1 be CL(u),

  except that we delete c_1 from CL(u) for neighbors u of v
- Let CL_2 be CL(u),

  except that we delete c_2 from CL(u) for neighbors u of v
- IF L3C(G-{v}, CL_1)= True: return True
- Return L3C(G-{v}, CL_2)

# BACKTRACKING TIME ANALYSIS

- Here, exhaustive search is O($3^n$) time, because there are three possible colors per vertex

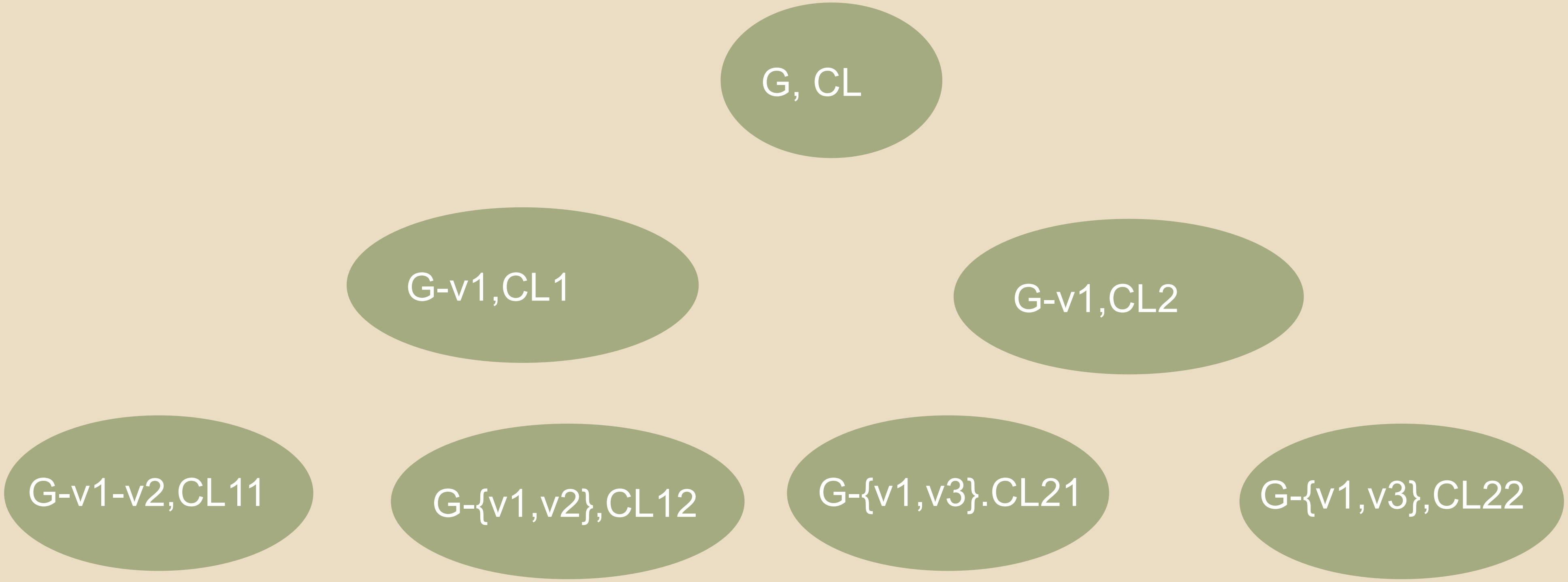
- General way to bound BT algorithms: View recursions as forming tree based on sub-calls

Time = number of leaves

If every node in tree makes at most f= fan-out recursive calls, then

The number of leaves = O($f$^{$depth$})

# THIS ALGORITHM



# TIME ANALYSIS CONT

- Depth ≤ n-1, because graph decreases every rec. call
- Fan-out =2 , because at most two rec. calls

- So time is $O(2^n)$
- Still exponential, but much better than $3^n$ for moderate sized
- inputs

# TOWARDS DYNAMIC PROGRAMMING

- Dynamic Programming = Backtracking + Memoization
- Memoization = store and re-use, like the Fibonacci algorithm from first class

Two simple ideas, but easy to get confused if you rush:
1. Where is the recursion?
(It disappears into the memoization, like the Fib. example did.)
2. 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

# COL702: Backtracking and Dynamic Programming

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

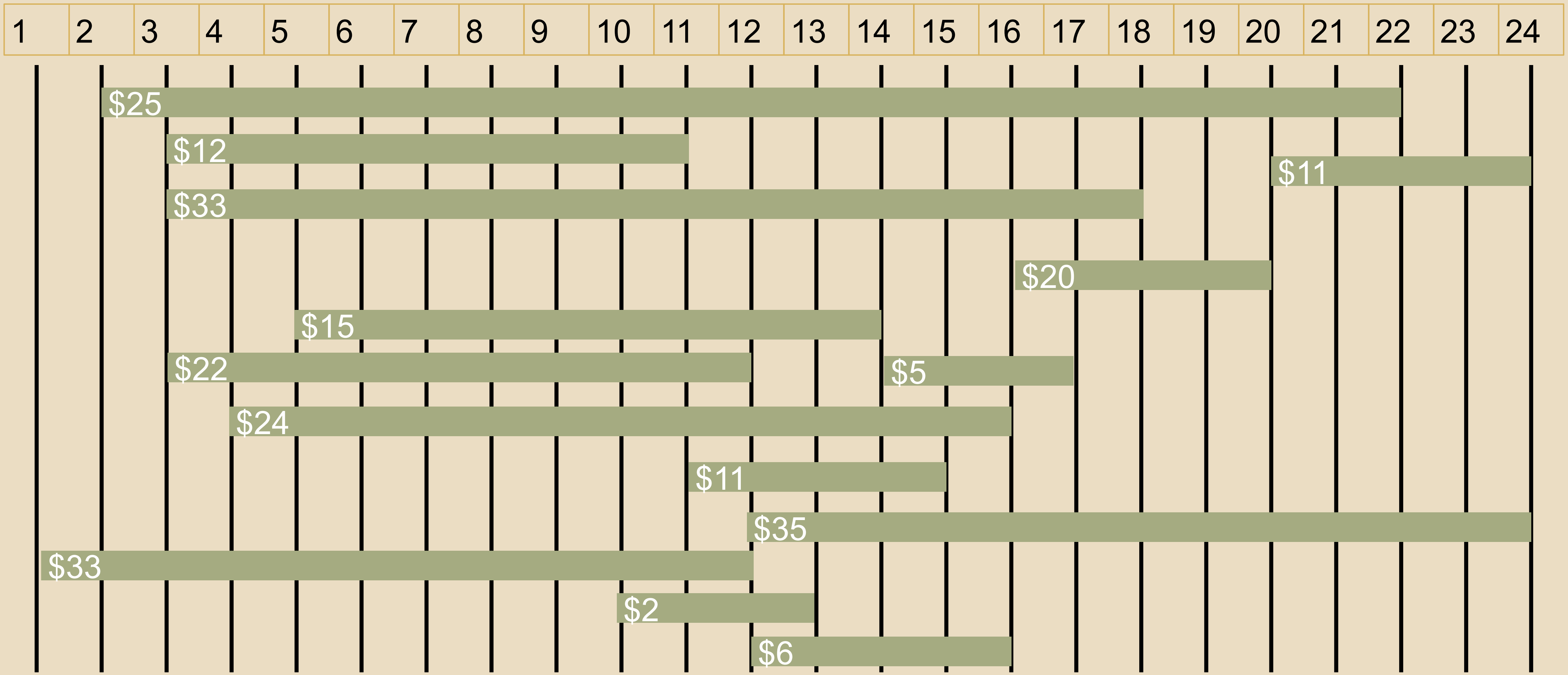
# FROM BACKTRACKING TO DYNAMIC PROGRAMMING

- 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."

# WEIGHTED EVENT SCHEDULING



# 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 $I_1 \dots I_n$.
- Pick first of these: $I_1$.
- Should we include $I_1$ or not? Try both possibilities.

```
BTWES (I_1 … 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.

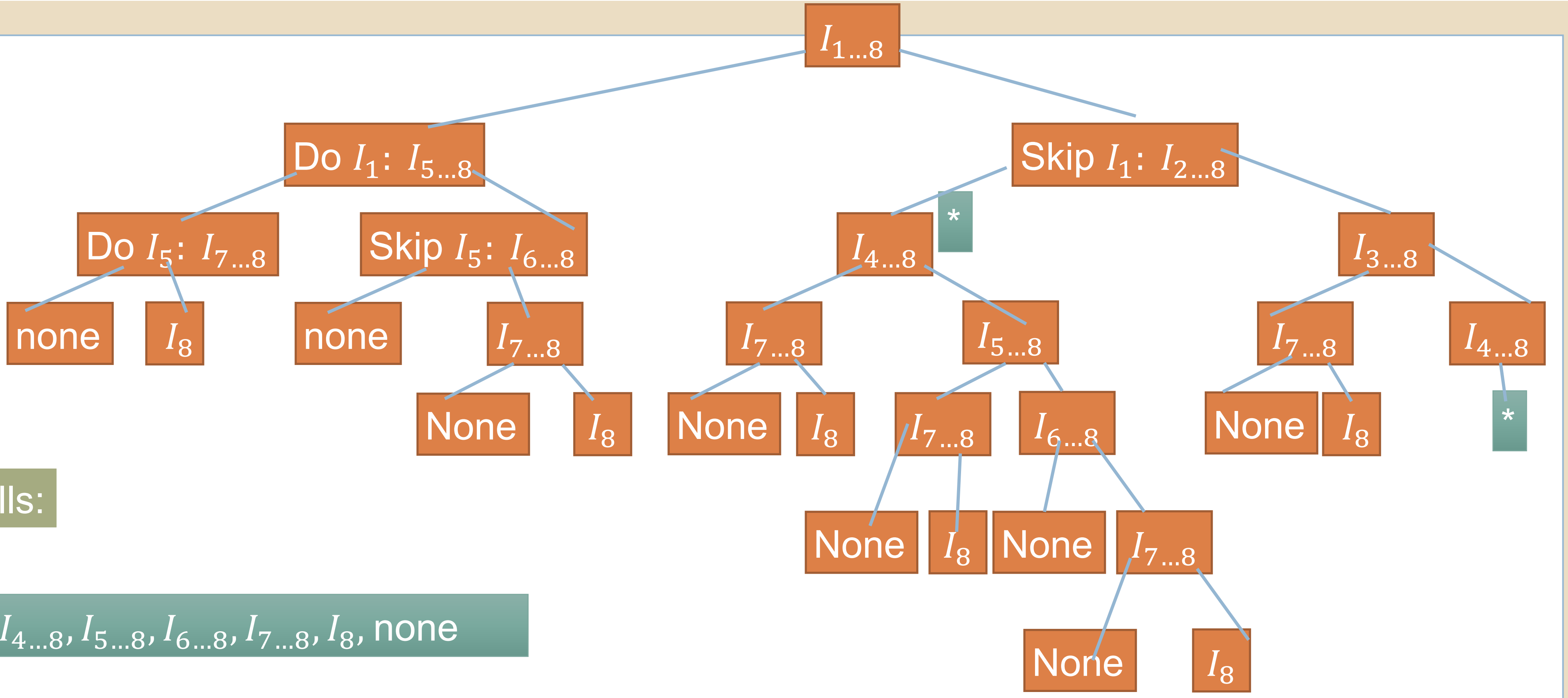
But our goal is a dynamic programming algorithm, so improving the backtracking time is irrelevant.

# 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:

$I_{1\dots8}, I_{2\dots8}, I_{3\dots8}, I_{4\dots8}, I_{5\dots8}, I_{6\dots8}, I_{7\dots8}, I_8$, none

# CHARACTERIZE CALLS MADE

All of the recursive calls BTWES makes are to arrays of the form

$I_{K\ldots n}$, 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 ($I_K \ldots I_n$)

    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)

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 $(I_K, \ldots, I_n)$

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

- 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 |
|---|---|---|
| 4+MV[5]=12 | MV[2]=14 | 14 |
| 3+MV[4]=14 | MV[3]=11 | 14 |
| 5+MV[7]=10 | MV[4]=11 | 11 |
| 6+MV[7]=11 | MV[5]=8 | 11 |
| 3+MV[7]=8 | MV[6]=5 | 8 |
| 4+MV[9]=4 | MV[7]=5 | 5 |
| 5+MV[9]=5 | MV[8]=3 | 5 |
| | | 3 |
| | | 0 |

# TRACING FORWARDS

- 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 | |
|---|---|---|---|
| 4+MV[5]=12 | MV[2]=14 | 14 | Exclude 1, go to 2 |
| 3+MV[4]=14 | MV[3]=11 | 14 | Include 2, go to 4 |
| 5+MV[7]=10 | MV[4]=11 | 11 | |
| 6+MV[7]=11 | MV[5]=8 | 11 | Include 4, go to 7 |
| 3+MV[7]=8 | MV[6]=5 | 8 | |
| 4+MV[9]=4 | MV[7]=5 | 5 | |
| 5+MV[9]=5 | MV[8]=3 | 5 | Include 7 go to 9 |
| | | 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

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

# 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 $a[i_1],\ldots,a[i_k]$ such that $i_1<\ldots<i_k$ and $a[i_1]<\ldots<a[i_k]$.

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 $a[i_1],\dots,a[i_k]$ such that $i_1<\dots<i_k$ and $a[i_1]<\dots<a[i_k]$.

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

*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

- **ALO**H**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

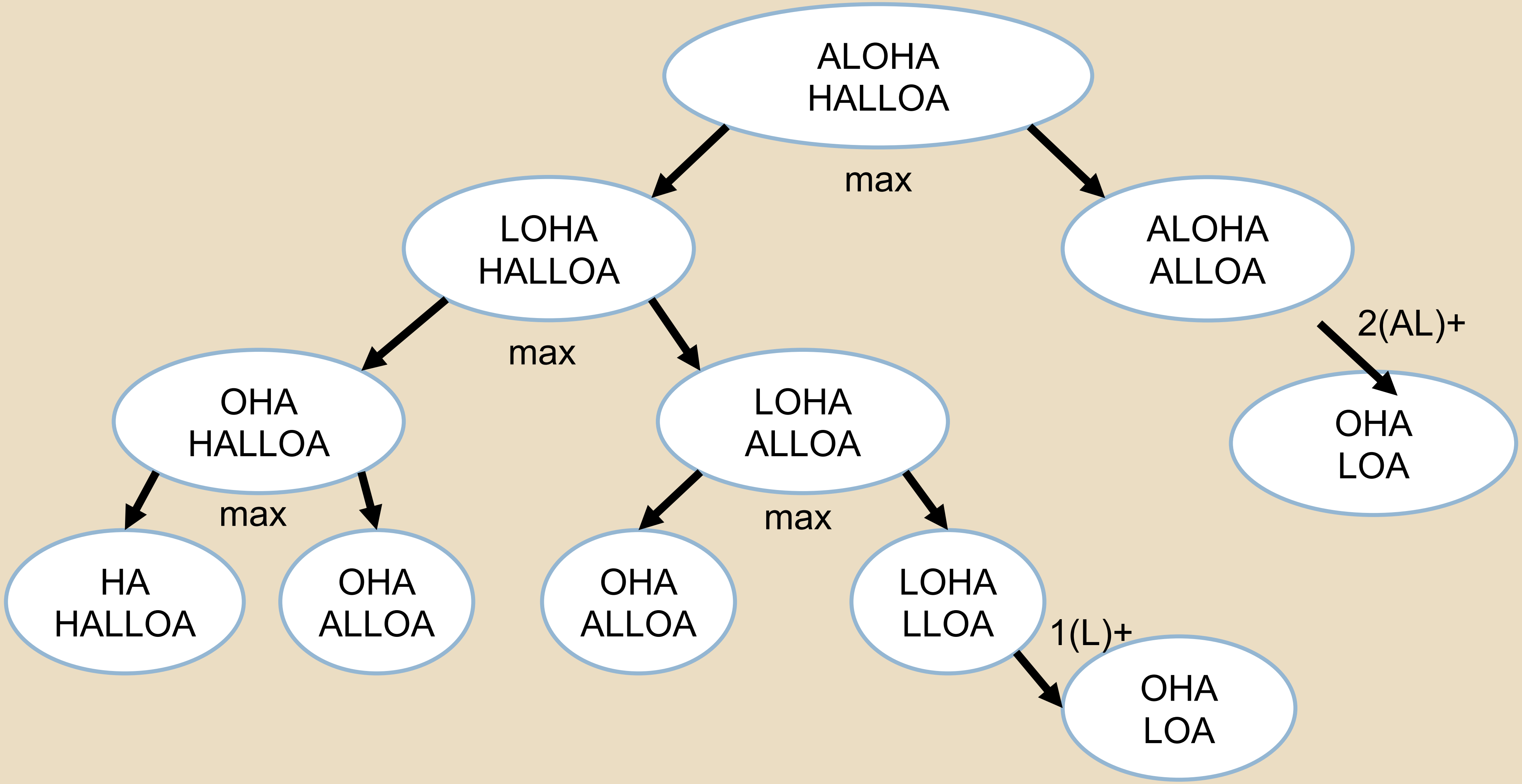
- ALOHA or LOHA
  ALLOA HALLOA

- 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
HA
HALLOA
OHA
ALLOA
max
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 |

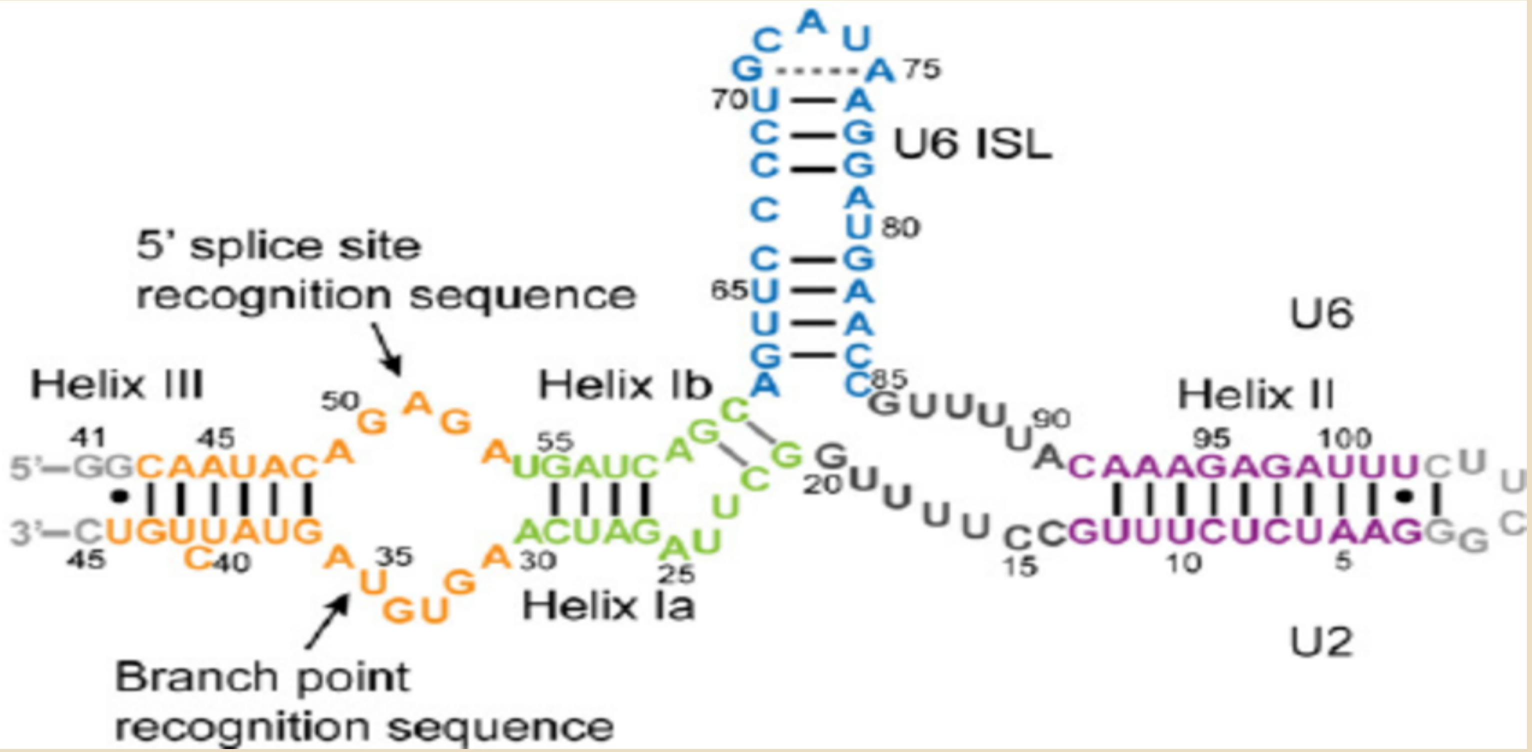
# EXAMPLE

| | 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

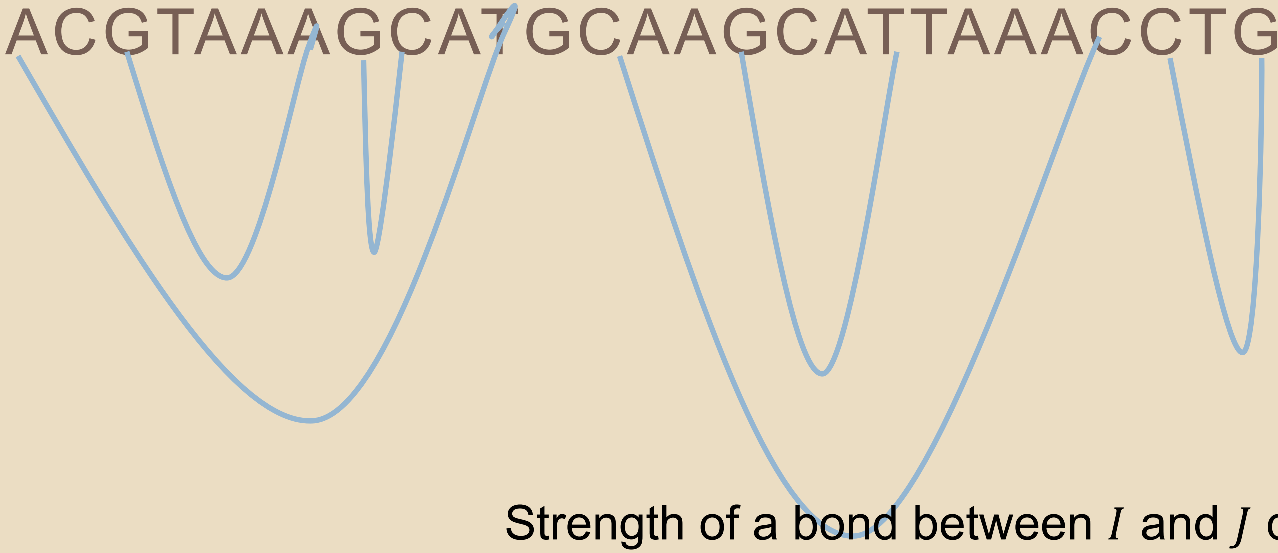



RNA folds back on itself, forming chemical bonds between amino acids in the sequence

# 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 $w_1 \dots w_n$, 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($w_1 \ldots w_n$)
- Initialize MS[$1 \ldots n, 0 \ldots n$]
- 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*