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

|   | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|   | $25| $12| $93| $15| $22| $11| $20| $15| $22| $11| $12| $33| $5 | $1 | $11| $33| $2 | $6 | $33| $2 | $6 | $33| $2 | $6 |

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: \( s_1 < f_1 \leq s_2 < f_2 \leq \cdots \leq s_k \leq f_k \)
- **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.

\[
\text{BTWES}(I_1 \ldots I_n):
\]
\[
\begin{align*}
\text{If } n=0 & \text{ return } 0 \\
\text{If } n=1 & \text{ return } V_1 \\
\text{Exclude} & := \text{BTWES}(I_2 \ldots I_n) \\
J & := 2 \\
\text{Until } (J > n \text{ or } s_j > f_{i}) & \text{ do:} \\
& J++ \\
\text{Include} & := V_j + \text{BTWES}(I_j \ldots I_n) \\
\text{Return } & \text{Max(Include, Exclude)}
\end{align*}
\]

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

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

Distinct calls:
CHARACTERIZE CALLS MADE

All of the recursive calls BTWES makes are to arrays of the form $I_{K...n}$, 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 $(I_{K...n})$ for all $K = 1 ... n$ 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 $(I_{K...n})$
- We'll use MV[n+1] to hold the best weight for the empty list, 0.
- So K ranges from 1 to n+1.
What happens when we run BTWES ($I_{K} \ldots I_{n}$)?

BTWES ($I_{K} \ldots I_{n}$)
- If $K=n+1$ return 0
- If $K=n$ return $V_{n}$
- Exclude:= BTWES($I_{K+1} \ldots I_{n}$)
  - $J:=K+1$
  - Until ($J > n$ or $s_{J} > f_{K}$) do:
    - $J++$
  - Include:= $V_{K} + $ BTWES($I_{J} \ldots I_{n}$)
- Return Max(Include, Exclude)

SIMULATE RECURSION ON SUBPROBLEM

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)

REPLACE RECURSION WITH ARRAY/MATRIX

Recall: MV[$K$] is the solution to the subproblem ($I_{K} \ldots I_{n}$)
INVERT TOP-DOWN RECURSION ORDER TO GET BOTTOM UP ORDER

BTWES \( (l_k \ldots l_n) \)
  If K=\( n+1 \) return 0
  If K=\( n \) return \( V_n \)
  Exclude:= BTWES\( (l_{K+1} \ldots l_n) \)
  J:=K+1
  Until (J > n or \( s_j > f_k \)) do:
    J++
  Include:= \( V_k + BTWES(l_j \ldots l_n) \)
  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\[l_1, \ldots, l_n\]
  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 \( (l_k, \ldots, l_n) \)
And \( MV[\( n+1 \)]=0 \)
### 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 |  |  
I6= (6,11), V6=4 |  |  
I7 = (9,13), V7=5 |  |  
I8= (10,12), V8=3 |  |  

### 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 |  |  
I6= (6,11), V6=4 |  |  
I7 = (9,13), V7=5 |  |  
I8= (10,12), V8=3 |  |  

\[
\text{MV}[7] = 3 + \text{MV}[7] = 8 \\
\text{MV}[6] = 5 \\
\text{MV}[7] = 4 + \text{MV}[9] = 4 \\
\text{MV}[7] = 5 \\
\text{MV}[8] = 5 + \text{MV}[9] = 5 \\ 
\text{MV}[8] = 3 \\
\]
**EXAMPLE**

Include | Exclude | MV
--- | --- | ---
I7 = (9,13), V7=5 | 5+MV[9]=5 | MV[8]=3
I8 = (10,12), V8=3 | | 

Best set: 2.4,7, Total value: 3+6+5=14

**TRACING FORWARDS**

Include | Exclude | MV
--- | --- | ---
I7 = (9,13), V7=5 | 5+MV[9]=5 | MV[8]=3
I8 = (10,12), V8=3 | | 

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_i\}...I_n\}]

\begin{align*}
&\text{MV}[n+1] := 0 \\
&\text{MV}[n] := V_n \\
&\text{FOR } K = n-1 \text{ down to } 1 \text{ do:} \\
&\quad \text{Exclude} := \text{MV}[K+1] \\
&\quad J := K + 1 \\
&\quad \text{Until } (J > n \text{ or } s_J > f_K) \text{ do:} \\
&\quad \quad J++ \\
&\quad \text{Include} := V_K + \text{MV}[J] \\
&\quad \text{MV}[K] := \text{Max}(\text{Include}, \text{Exclude}) \\
&\text{Return } \text{MV}[1]
\end{align*}
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[\ldots 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

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 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.