COL351: Slides for Lecture Component 22

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

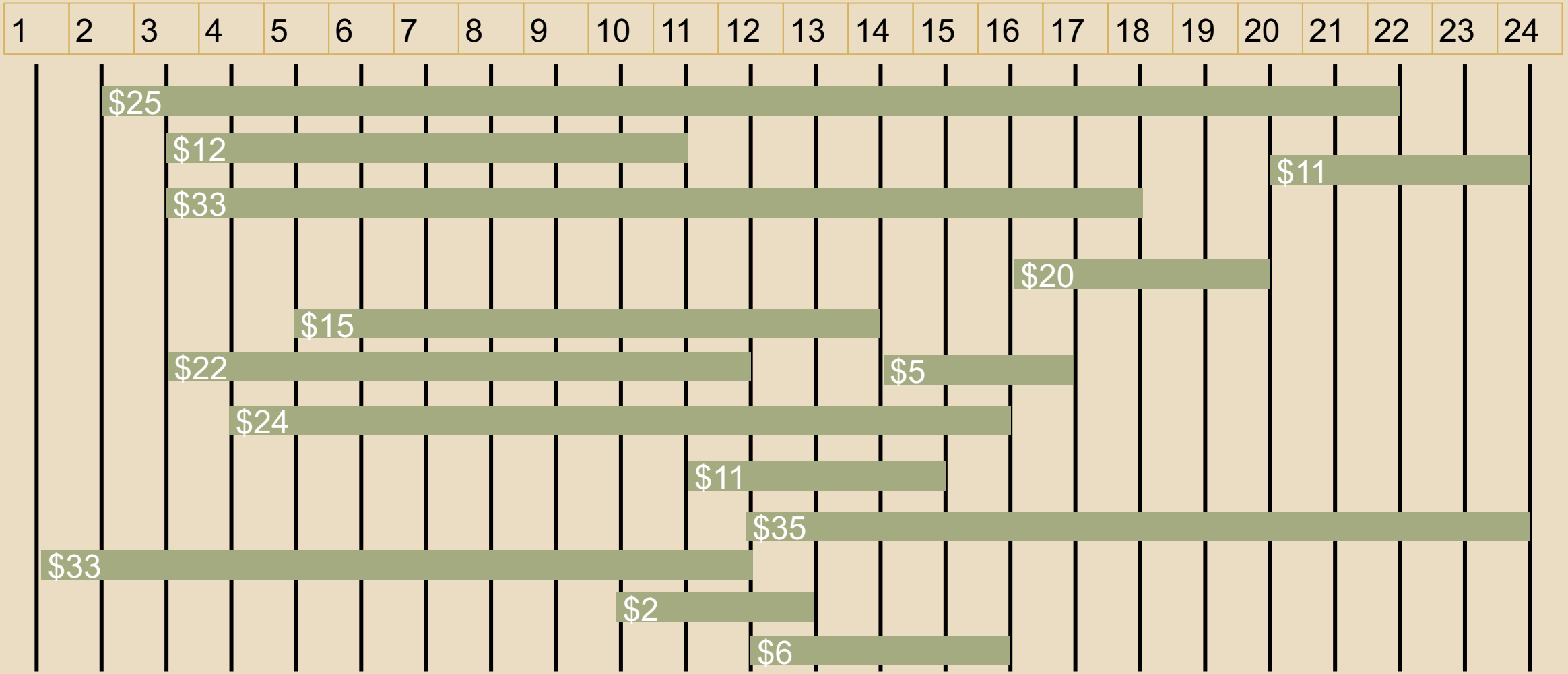
FROM BACKTRACKING TO DYNAMIC PROGRAMMING

- 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 $I_1 \ldots I_n$.
- 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.

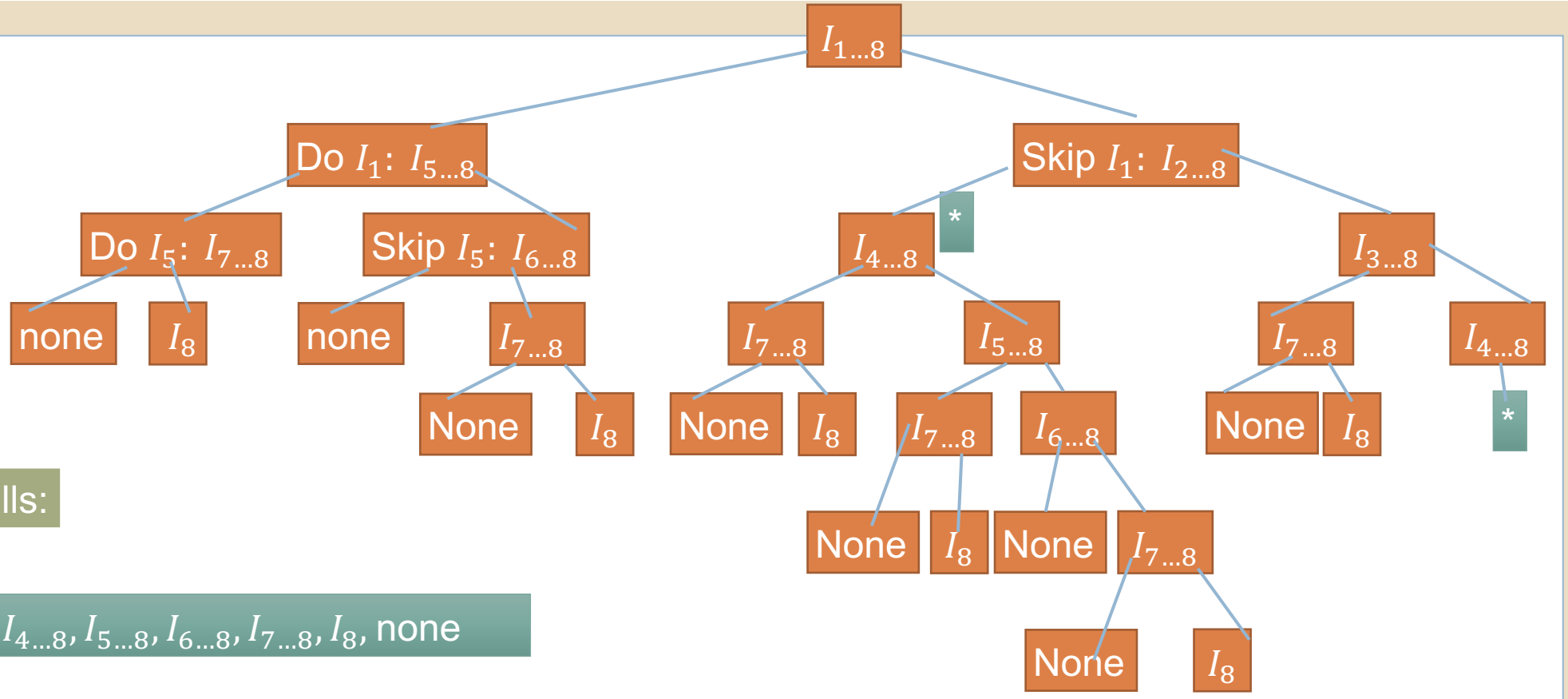
But our goal is a dynamic programming algorithm, so improving the backtracking time is irrelevant.

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:

$I_{1\ldots8}, I_{2\ldots8}, I_{3\ldots8}, I_{4\ldots8}, I_{5\ldots8}, I_{6\ldots8}, I_{7\ldots8}, I_8$, 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 … I_n)?

BTWES (I_K … 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 ... 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

- 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
	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

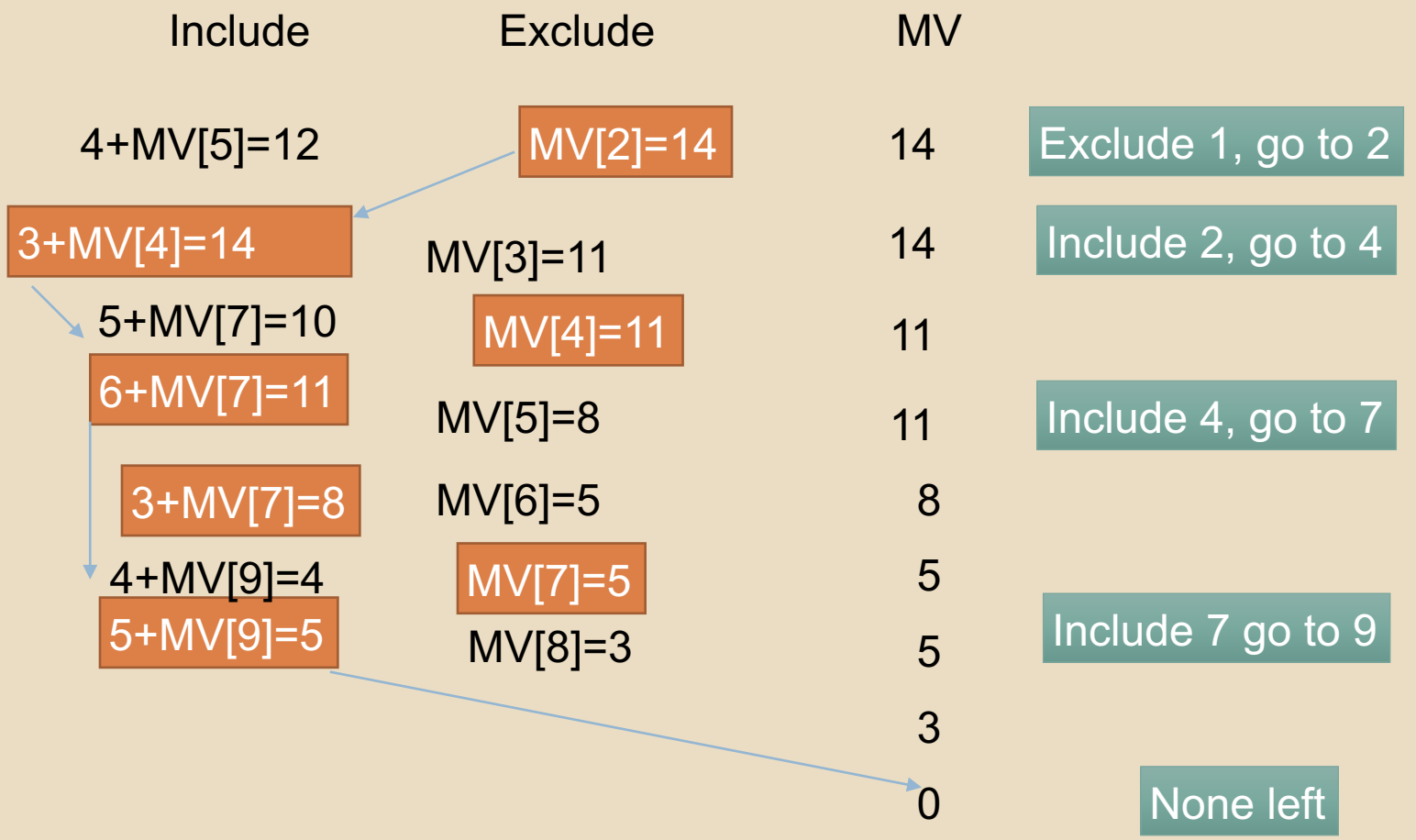
EXAMPLE

- 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



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.