Dynamic Programming

Knapsack $[(val_1, wt_1), \ldots, (val_n, wt_n)]$, W

Pick a subset $S \subseteq [n]$

maximizing $\sum_{i \in S} val_i$

subject to $\sum_{i \in S} wt_i \leq W$

Suggestion: greedy using val/wt as score.

Counterexample: $W = 10$

$[(35,5), (35,5), (60,6)]$

val/wt	7	7	10

$OPT(lst, W) \overset{want}{=}$ max value solving the problem

$$= \begin{cases} OPT(lst[2..n], W) \text{ if } wt_1 > W \\ \max(val_1 + OPT(lst[2..n], W - wt_1), \text{ // take item 1} \\ \quad OPT(lst[2..n], W)) \quad \text{// don't take it} \end{cases}$$

Base case: len = 1 $OPT = \begin{cases} val_1 & \text{if } wt_1 \leq W \\ 0 & \text{if } wt_1 > W \end{cases}$

Proof of correctness: By induction on len(lst).

Base case: len = 1. (Clear)

Inductive Hyp: $OPT(lst, w)$ gives correct optimum value for all lists of length n.

Inductive Step: Suppose $len(lst) = n+1$.

Case 1: $wt_1 > W$. In this case, item 1 can't be part of any valid solution b/c it weighs too much, so

optimum = optimum($lst[2..n+1], W$)

= OPT($lst[2..n+1], W$) (by IH)

Case 2: $wt_1 \leq W$.

2a: Suppose optimum sol'n includes item 1.

$\Rightarrow$ optimum value must be val_1 + optimum value on the rest, where total wt $\leq W$

total wt = wt_1 + (wt of the rest of the optimal subset)

$= val_1 + OPT(lst[2..n+1], W - wt_1)$ (by IH)

2b: Suppose optimum soln doesn't include item 1. Then optimum(lst, W)

= optimum$(lst[2..n+1], W) \overset{IH}{=} OPT(lst[2..n+1], W)$

Therefore, if $wt_1 \leq W$

optimum = max(2a, 2b)

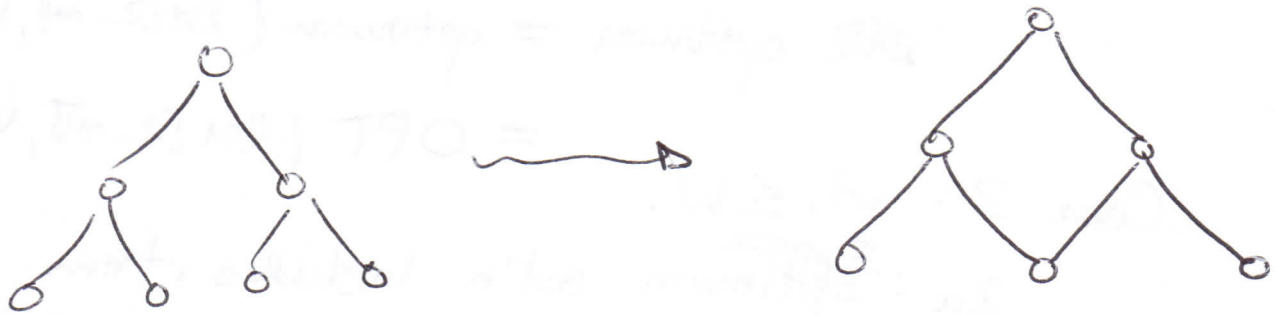
$= \max\{val_1 + OPT(lst[2..n+1], W - wt_1), OPT(lst[2..n+1], W)\}$

$= OPT(lst, W)$ (by construction).

∴ By induction, OPT(lst, W) correct for all lists.

Overlapping subproblem property

You're asking to solve the same subproblem more than once.



recursion tree → recursion DAG

One trick: **memoization**

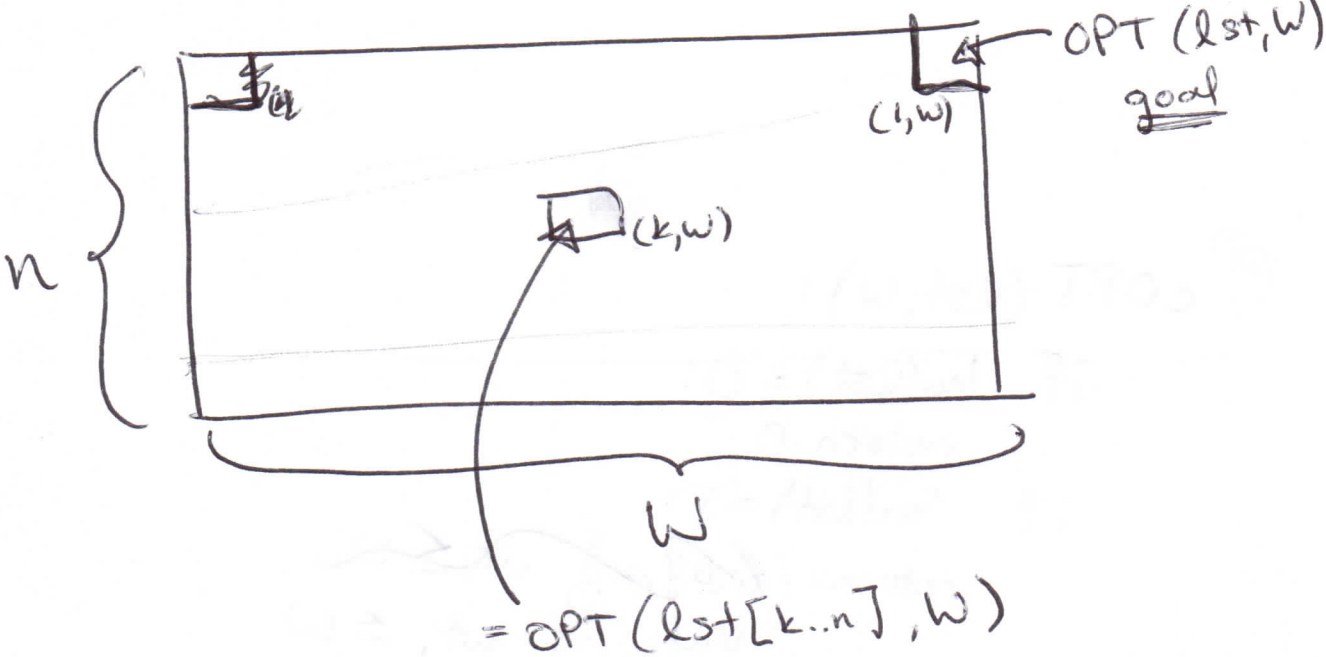
f(x)

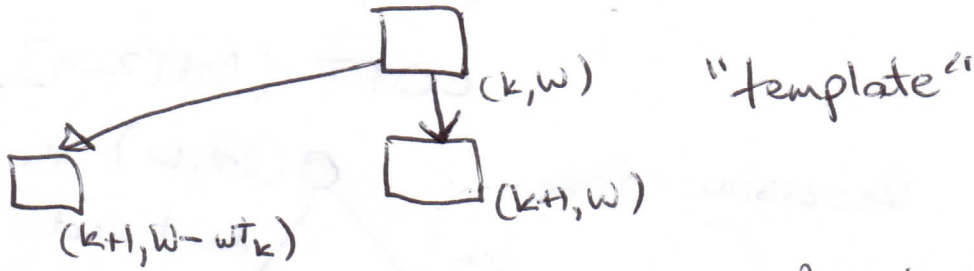
```
memo-f(x):
    is x ∈ table?
        if so, return table(x)
    if not
        table.insert(x, f(x))
        return table(x).
```

memo-cOPT will be correct + avoid duplicating work

Could have written $OPT(k, W) = OPT(lst[k..n], W)$

"Dyn. Prog. table"



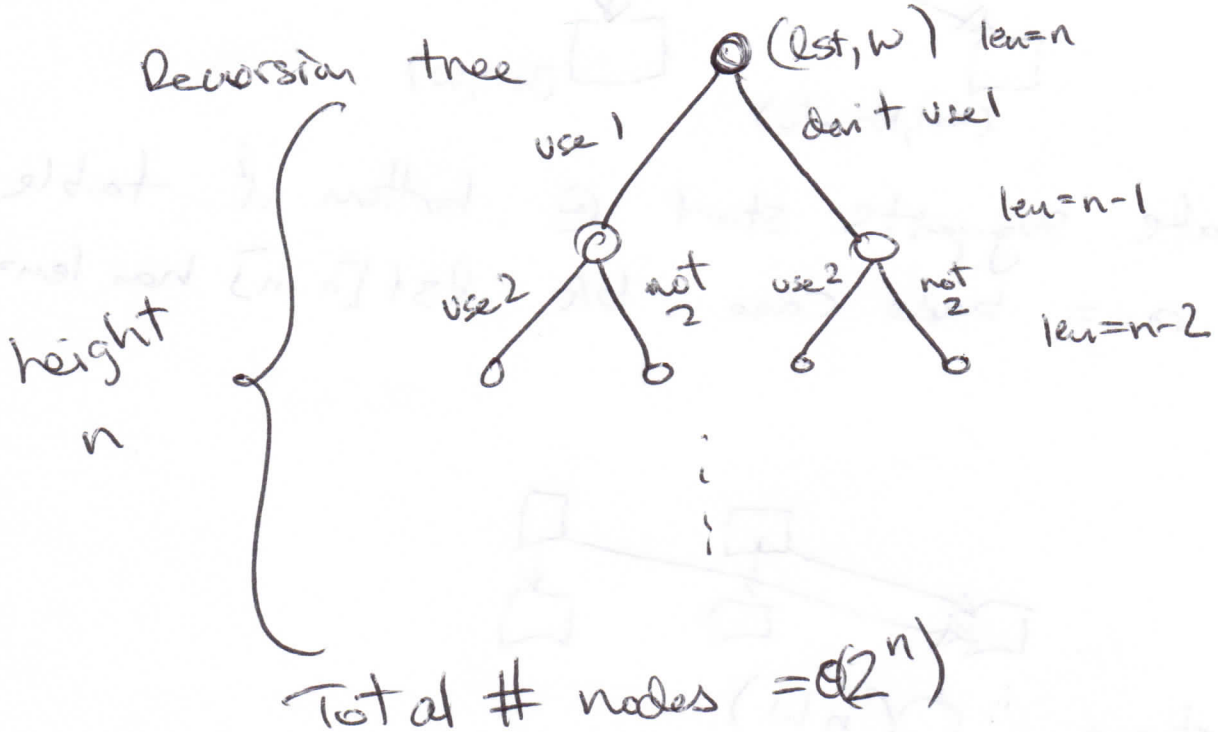


Template suggests start @ bottom of table

Bottom row = base case b/c lst[n..n] has len = 1.

Run time $O(nW)$

```
cOPT(lst, W):
    if len(lst) = 0:
        return 0
    if len(lst) = 1:
        return val_1 if wt_1 ≤ W
               0     otherwise
    return max( val_1 + cOPT(lst[2..n], W - wt_1),
                cOPT(lst[2..n], W) )
```



Total # nodes = $O(2^n)$

allocate a $n \times W$ table

```
for w = 1 to W                    // fill in bottom row
    if wt_n ≤ w                   // base case
        table(n, w) = val_n
    else
        table(n, w) = 0
```

$O(W)$ steps

```
for i = n-1 down to 1             // bottom-up
    for all w ∈ {1, ..., W}
        if wt_i > w
            table(i, w) = table(i+1, w)
        else
            table(i, w) = max( val_i + table(i+1, w - wt_i),
                               table(i+1, w) )
return table(1, W)
```

$O(nW)$ steps

Fibonacci $F_n = F_{n-1} + F_{n-2}$ $F_0 = F_1 = 1$

```
fib(n):
    //base case
    else
        return fib(n-1) + fib(n-2).
```

Don't

Instead, use an array of length n ← 1D Dyn. Prog. table

Recap:

REMEMBER BASE CASE

1. Find the recurrence relation.
2. Figure out the Dyn. Prog. table
   - size
   - # dimensions
3. Figure out template.
4. Use template to determine the order in which to fill in the table.
5. Fill in, in order from (4), using simple for loops + recurrence from (1).

(6. Work back thru table to compute solution)