No Wednesday OH 6-9pm

Midterm 6/1

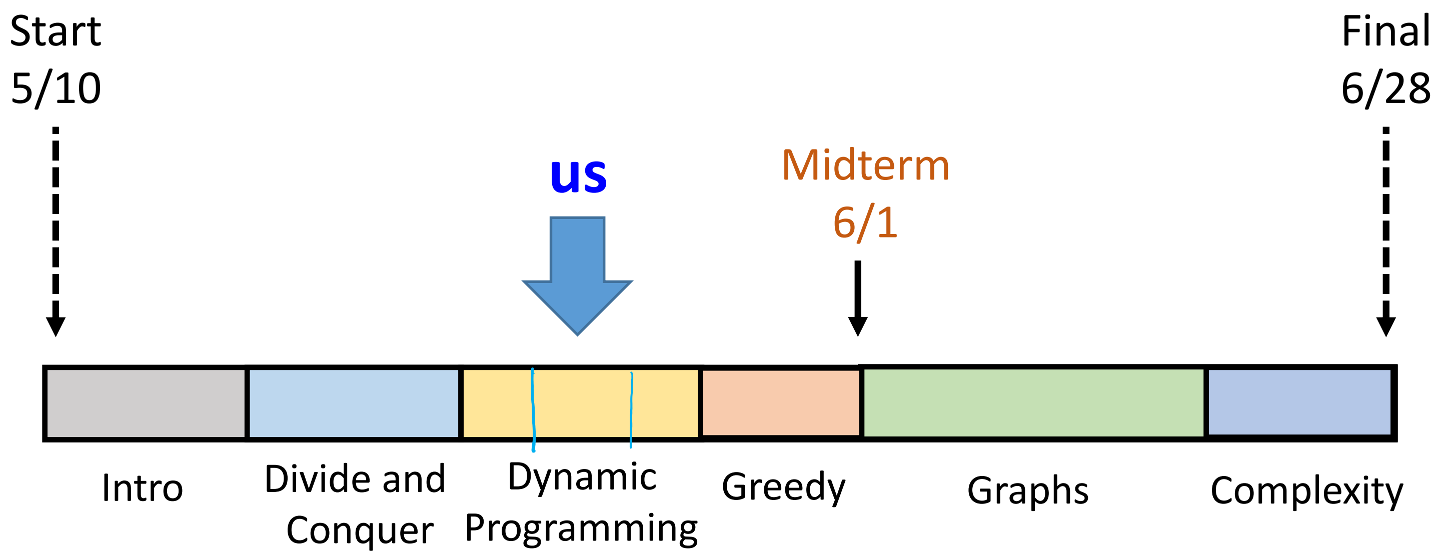
CS3000: Algorithms & Data
Drew van der Poel

Lecture 9

- Dynamic Programming: Weighted Interval Scheduling (Finish)
- Dynamic Programming: Knapsack

May 24, 2021

Outline



Last class: dynamic programming: Weighted Interval Scheduling

Next class: dynamic programming: Segmented Least Squares

Weighted Interval Scheduling

- How can we optimally schedule a resource?
  - This classroom, a computing cluster, …

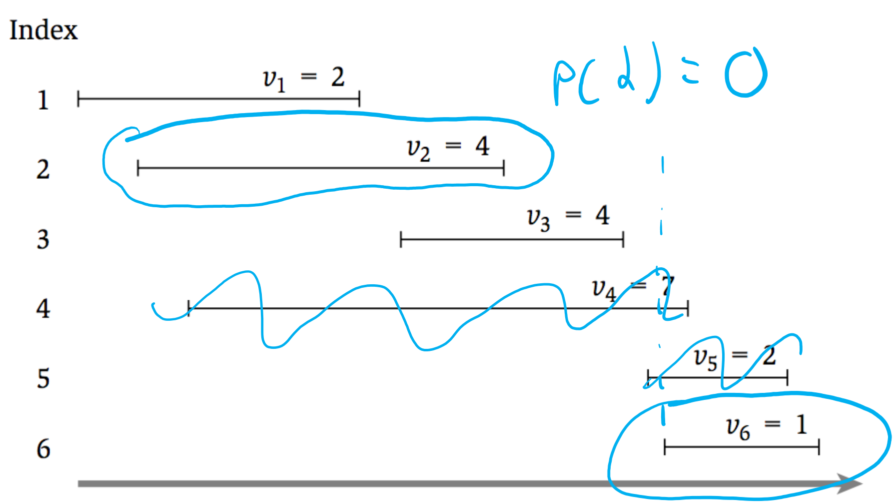
- **Input:** n intervals (s_i, f_i) each with value v_i
  - Assume intervals are sorted so $f_1 < f_2 < \cdots < f_n$
- **Output:** a compatible schedule S **maximizing** the total value of all intervals
  - A **schedule** is a subset of intervals $S \subseteq \{1, \dots, n\}$
  - A schedule S is **compatible** if no $i, j \in S$ overlap
  - The **total value** of S is $\sum_{i \in S} v_i$

A Recursive Formulation

Which is better?

- the optimal solution for $\{1, \dots, 5\}$
- $\{6\}$ + the optimal solution for $\{1, \dots, 3\}$

OPT(2)



A Recursive Formulation: Subproblems & Recurrence

a value, not a set of intervals

- **Subproblems:** Let $OPT(i)$ be the ***value* of the optimal schedule** using only the intervals $\{1, \ldots, i\}$ $(OPT(i) = value(O_i))$

- **Case 1:** Final interval is not in O_i $(i \notin O_i)$
  - Then O_i must be the optimal solution for $\{1, \ldots, i-1\}$
- **Case 2:** Final interval is in O_i $(i \in O_i)$
  - Assume intervals are sorted so that $f_1 < f_2 < \cdots < f_n$
  - Let $p(i)$ be the largest j such that $f_j < s_i$
  - Then O_i must be i + the optimal solution for $\{1, \ldots, p(i)\}$

$2 \le i \le n$

- $OPT(i) = \max\{\underbrace{OPT(i-1)}_{\text{case 1}}^{\text{i out}}, \underbrace{v_i + OPT(p(i))}_{\text{case 2}}^{\text{i in}}\}$

- $OPT(0) = 0, OPT(1) = v_1$

recurrence

Interval Scheduling: Top Down

```
// All inputs are global vars
M ← empty array, M[0] ← 0, M[1] ← v₁
FindOPT(n):
  if (M[n] is not empty): return M[n]
  else:
    M[n] ← max{FindOPT(n-1), vₙ + FindOPT(p(n))}
    return M[n]
```

- What is the running time of **FindOPT(n)**?

n-1 elems. filled

each fill → 2 rec. calls

Total: (n-1) 2 → $\Theta(n)$

Interval Scheduling: Bottom Up

```
// All inputs are global vars
FindOPT(n):
  M[0] ← 0, M[1] ← v₁
  for (i = 2,…,n):
    M[i] ← max{M[i-1], vᵢ + M[p(i)]}
  return M[n]
```

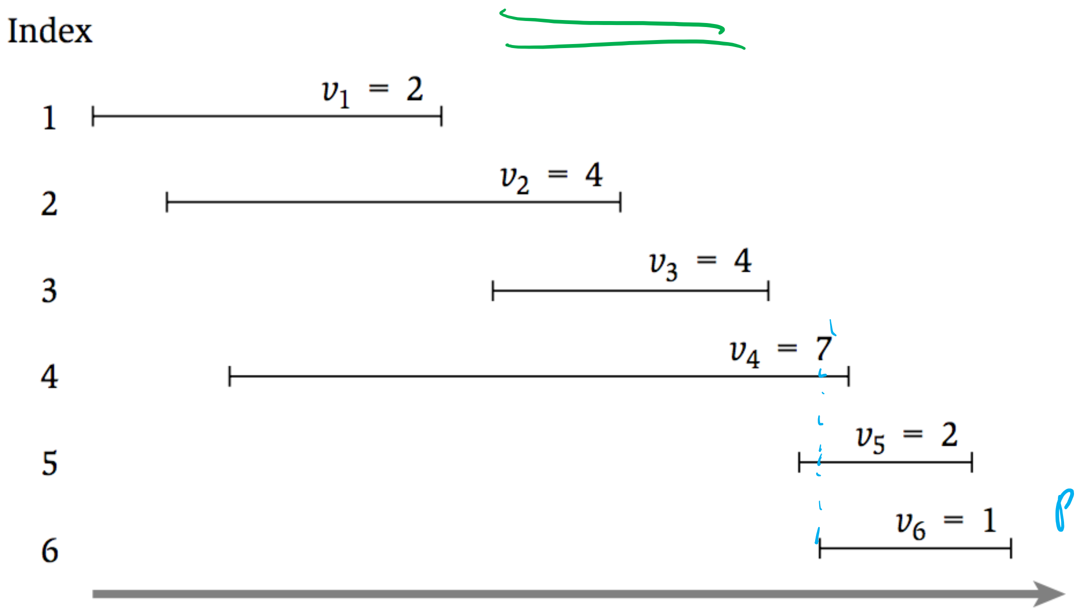
4 M[0]

- What is the running time of **`FindOPT(n)`**?

n-1 loops → $\Theta(n)$ total r/t

Finding the Optimal Solution

- But we want a schedule, not a value!



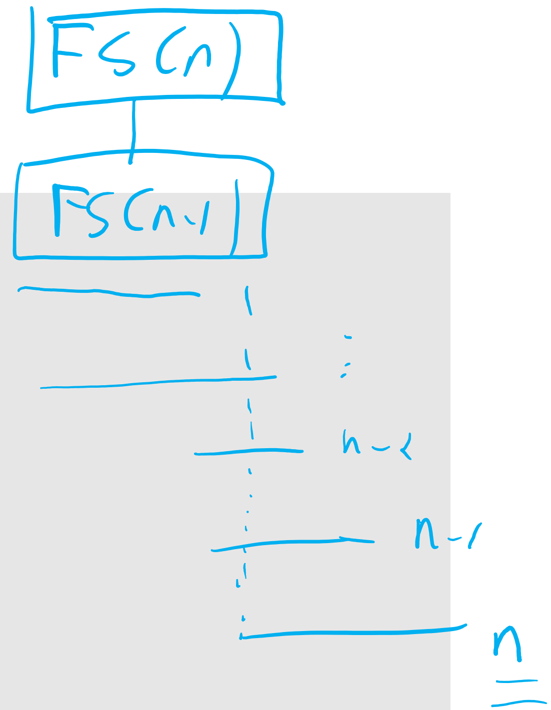
$P(6) = 3$

M[0]	M[1]	M[2]	M[3]	M[4]	M[5]	M[6]
0	2	4	6	7	8	8

Dynamic Programming Recipe

- **Recipe:**
  - (1) identify a set of **subproblems**
  - (2) relate the subproblems via a **recurrence**
  - (3) find an **efficient implementation** of the recurrence (top down or bottom up)
  - (4) **reconstruct the solution** from the DP table

Finding the Optimal Solution




```
// All inputs are global vars
FindSched(M,n):
  if (n = 0): return ∅
  elseif (n = 1): return {1}
  elseif (v_n + M[p(n)] > M[n-1]):
    return {n} + FindSched(M,p(n))
  else:
    return FindSched(M,n-1)
```

- What is the running time of **FindSched(n)**?

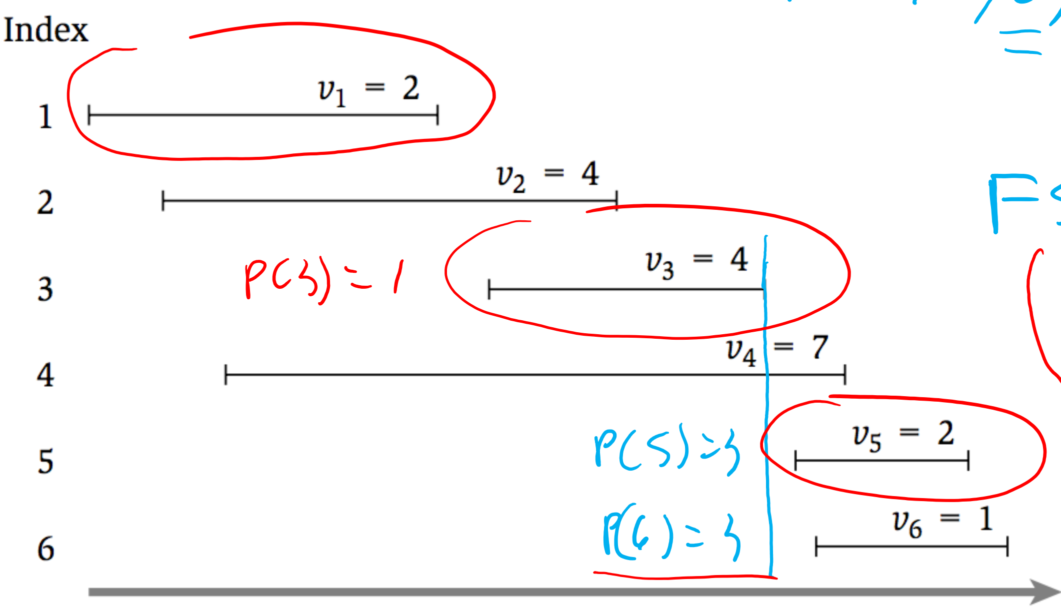
each rec. call $\rightarrow O(1)$

# of rec. calls $\leq n-1$

$\boxed{\text{Total: } O(n)}$

Finding the Optimal Solution

$n = 6$



$FS(M, 6)$: $M[5]$ vs. $v_6 + M[p(6) = 3]$

$8 > 1 + 6$

$FS(M, 5)$: $M[4]$ vs $v_5 + M[p(5) = 3]$

$7 < 2 + 6$

$FS(M, 3)$

$M[2]$ vs. $v_3 + M[p(3) = 1]$

$4 < 4 + 2$

$FS(M, 1)$

Schedule: $\{5, 3, 1\}$

M =

M[0]	M[1]	M[2]	M[3]	M[4]	M[5]	M[6]
0	2	4	6	7	8	8

How much space is used?

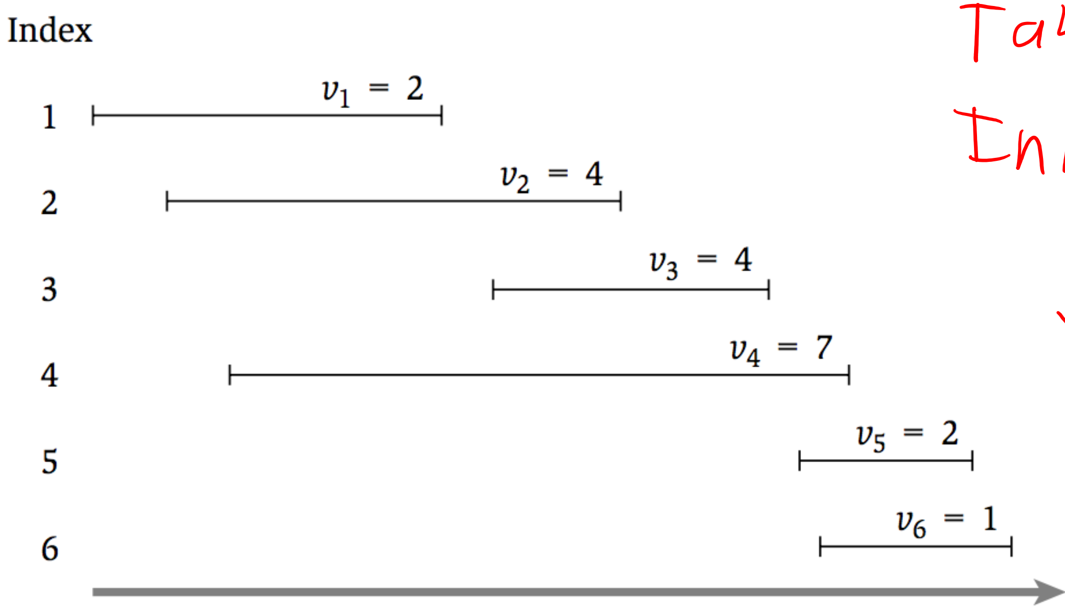


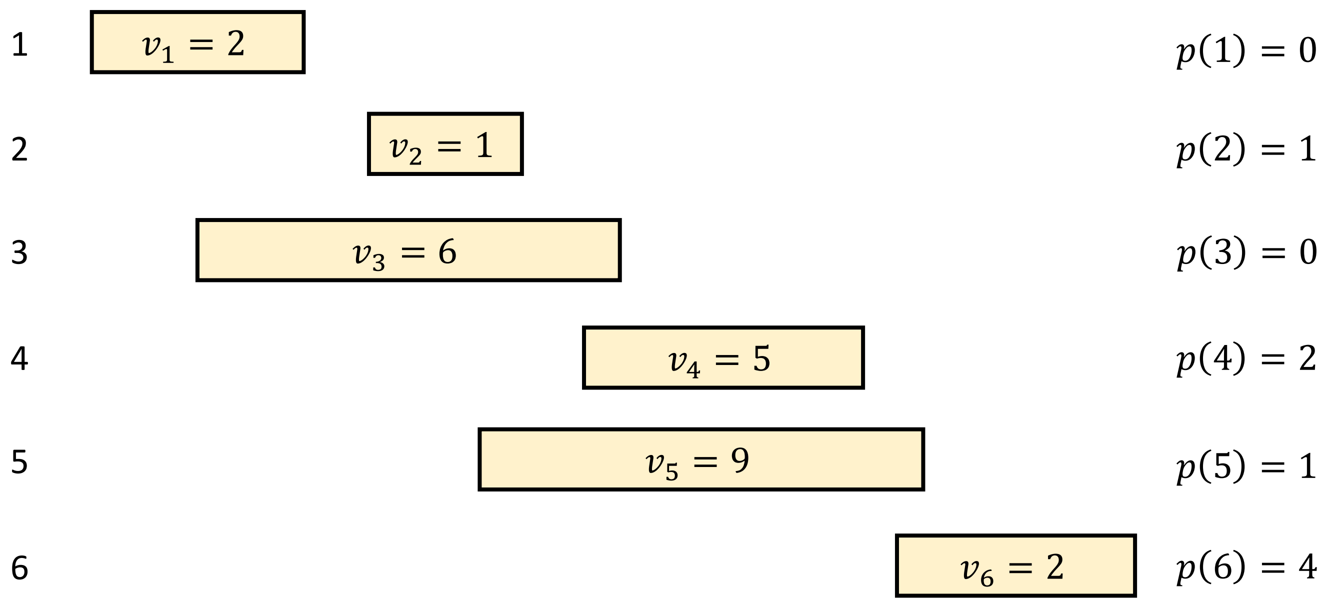
Table: $1 \times (n+1) \rightarrow O(n)$

Inputs: n, each $O(1)$

Total: $O(n)$

M[0]	M[1]	M[2]	M[3]	M[4]	M[5]	M[6]
0	2	4	6	7	8	8

Now You Try



M[0]	M[1]	M[2]	M[3]	M[4]	M[5]	M[6]

Dynamic Programming Recap

- Express the optimal solution as a **recurrence**
  - Identify a small number of subproblems 1
  - Relate the optimal solution on subproblems 2
- Efficiently solve for the **value** of the optimum 3
  - Simple implementation is exponential time, but top-down and bottom-up are linear time
    - Top-Down: recursive, store solution to subproblems
    - Bottom-Up: iterate through subproblems in order
- Find the **solution** using the table of **values** 4

Dynamic Programming Recap

- **Recipe:**

  (1) identify a set of **subprblems** $\to$ OPT(i)

  TODAY: ≥ 1 variables

  (2) relate the subproblems via a **recurrence**

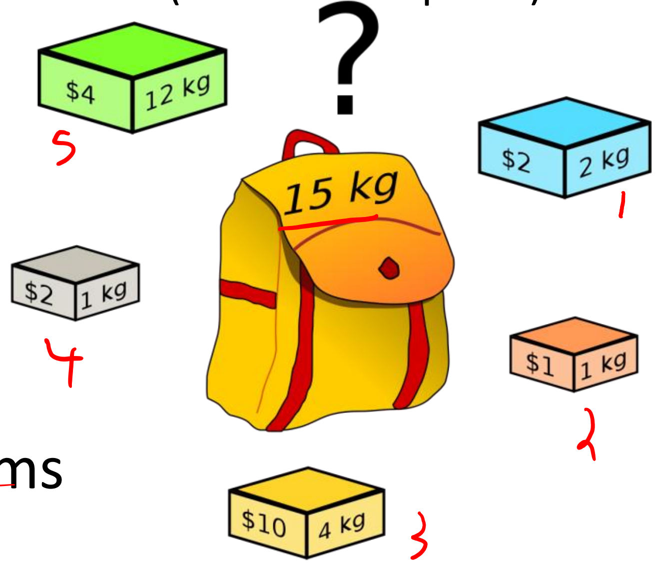
  (3) find an **efficient implementation** of the recurrence (top down or bottom up)

  (4) **reconstruct the solution** from the DP table

Knapsack

The Knapsack Problem

(source: Wikipedia)



- **Input:** n items for your knapsack
  - value v_i and a weight $w_i \in \mathbb{N}$ for n items
  - capacity of your knapsack $T \in \mathbb{N}$
- **Output:** the most valuable subset of items that fits in the knapsack
  - Subset $S \subseteq \{1, \ldots, n\}$
  - Value $V_S = \sum_{i \in S} v_i$ as large as possible
  - Weight $W_S = \sum_{i \in S} w_i$ at most T

$S = \{1, 2, 3\}, \; V_S = 13, \; W_S = 7 \leq 15 = T$

- **Want:** $\mathbf{argmax}_{S \subseteq \{1,\ldots,n\}} V_S$ s.t. $W_S \leq T$

- **(SubsetSum:** $v_i = w_i$,
- **TugOfWar:** $v_i = w_i, T = \frac{1}{2}\sum_i v_i$)

$n = 5 \qquad T = 15$

$v_1 = 2 \qquad w_1 = 2$

$v_2 = 1 \qquad w_2 = 1$

$v_3 = 10 \qquad w_3 = 4$

$v_4 = 2 \qquad w_4 = 1$

$v_5 = 4 \qquad w_5 = 12$

- Problems: counting students, stable matching, sorting, n-digit mulitiplication, array searching, selection, weighted interval scheduling, segmented least squares, **knapsack**

- Alg. techniques: divide & conquer, dynamic programming

- Analysis: asymptotic analysis, recursion trees, Master Thm.

- Proof techniques: (strong) induction, contradiction

Do we really need DP?

Items with large $\frac{v_i}{w_i}$ seem like good choices…

Ex. $T = 8$, $(v_1 = 6, w_1 = 5), (v_2 = 4, w_2 = 4), (v_3 = 4, w_3 = 4)$

$\frac{V_1}{W_1} = \frac{6}{5}$ $\frac{V_2}{W_2} = 1$, $\frac{V_3}{W_3} = 1$

- Strategy 1: Repeatedly pick items that fit with largest $\frac{v_i}{w_i}$

$S = \{1\}$ $T = 8 - 5 = 3$

$V_S = 8 > 6$

- Is this optimal? NO - $S = \{2, 3\}$ $W_S = 8 \leq 8 = T$

Knapsack – what to do with n-th item?

Want: $\mathbf{argmax}_{S \subseteq \{1,\ldots,n\}} V_S$ s.t. $W_S \leq T$

- **Case 1:** $n \notin O_n$

- **Case 2:** $n \in O_n$

Knapsack - subproblems

- Let $O_n \subseteq \{1, \ldots, n\}$ be the **optimal** subset of items given the first *n* items
- **Case 1:** $\mathrm{n} \notin O_n$

  $O_n =$ O_{n-1} w/ same capacity

- **Case 2:** $n \in O_n$

  $O_n =$ $\{n\} \cup O_{n-1}$ w/ capacity = previous cap. $- w_n$

Knapsack - recurrence

$0 \le j \le n$

$0 \le S \le T$

a value

- Let $\mathbf{OPT}(\boldsymbol{j}, \boldsymbol{S})$ be the **value** of the optimal subset of items $\{1, \ldots, j\}$ in a knapsack of size S
- **Case 1:** $j \notin O_{j,S}$

$$OPT(j, S) = OPT(j-1, S)$$

- **Case 2:** $j \in O_{j,S}$

$$OPT(j, S) = OPT(j-1, S-w_j) + v_j$$

Knapsack - recurrence

- Let $\mathbf{OPT}(\boldsymbol{j}, \boldsymbol{S})$ be the **value** of the optimal subset of items $\{1, \ldots, j\}$ in a knapsack of size S
- **Case 1:** $j \notin O_{j,S}$
  - $OPT(j, S) = OPT(j-1, S)$
- **Case 2:** $j \in O_{j,S}$
  - $OPT(j, S) = v_j + OPT(j-1, S-w_j)$

Recurrence:

$$\mathrm{OPT}(j, S) = \begin{cases} \max\left(OPT(j-1, S), OPT(j-1, S-w_j) + v_j \right) & \text{if } w_j \le S \\ OPT(j-1, S) & \text{else } w_j > S \end{cases}$$

Base Cases:

$\mathrm{OPT}(j, 0) = 0$

$\mathrm{OPT}(0, S) = 0$

Knapsack - recurrence

1

- Let $\mathbf{OPT}(\boldsymbol{j}, \boldsymbol{S})$ be the **value** of the optimal subset of items $\{1, \dots, j\}$ in a knapsack of size S
- **Case 1:** $j \notin O_{j,S}$
  - $OPT(j, S) = OPT(j-1, S)$
- **Case 2:** $j \in O_{j,S}$
  - $OPT(j, S) = v_j + OPT(j-1, S-w_j)$

2

Recurrence:

$$\mathrm{OPT}(j, S) = \begin{cases} \max\{OPT(j-1, S), v_j + OPT(j-1, S-w_j)\} & S \geq w_j \\ OPT(j-1, S) & S < w_j \end{cases}$$

Base Cases:

$$\mathrm{OPT}(j, 0) = \mathrm{OPT}(0, S) = 0$$

Knapsack ("Bottom-Up")

```
// All inputs are global vars
FindOPT(n,T):
  M[0,S] ← 0, M[j,0] ← 0

  for (j = 1,…,n):
    for (S = 1,…,T):
      if (w_j > S): M[j,S] ← M[j-1,S]
      else: M[j,S] ← max{M[j-1,S],v_j + M[j-1,S-w_j]}

  return M[n,T]
```

O(1) each

runtime: O(nT)

nT iterations → # of loops

each loop → O(1)

Ask the Audience

Space: $O(nT) \sim (T+1) \times (n+1)$

entries in DP table

dominates

- Input: $T = 8, n = 3$
  - $w_1 = 2\ ,\ v_1 = 4$
  - $w_2 = 3\ ,\ v_2 = 5$
  - $w_3 = 5\ ,\ v_3 = 8$

j

items									
3	0	0	4	5	5	9	9	12	$\boxed{13}$
2	0	0	4	5	5	9	9	9	9
1	0	0	$\boxed{4}$	4	4	4	4	4	4
0	0	0	0	0	0	0	0	0	0
-	0	1	2	3	4	5	6	7	8

capacities (S)

$$\mathrm{OPT}(j,S) = \begin{cases} \max\{OPT(j-1,S), v_j + OPT(j-1,S-w_j)\} & \text{If } S \geq w_j \\ OPT(j-1,S) & \text{If } S < w_j \end{cases}$$

Filling the Knapsack

- Let $\boldsymbol{O_{j,S}}$ be the **optimal subset of items** $\{1, \ldots, j\}$ in a knapsack of size S
- **Case 1:** $j \notin O_{j,S}$
  - Use opt. solution for items 1 to j-1 in a knapsack of size S
- **Case 2:** $j \in O_{j,S}$
  - Use j + opt. solution for items 1 to j-1 in a knapsack of size $S - w_j$

Filling the Knapsack

```
// All inputs are global vars
// M[0:n,0:T] contains solutions to subproblems
FindSol(M,n,T):
  if (n = 0 or T = 0): return Ø
  else:
    if (w_n > T): return FindSol(M,n-1,T)
    else:
      if (M[n-1,T] > v_n + M[n-1,T-w_n]):
        return FindSol(M,n-1,T)
      else:
        return {n} + FindSol(M,n-1,T-w_n)
```

Knapsack Wrapup

- Can solve knapsack problems in time/space $O(nT)$
  - **Recipe:**

    (1) identify a set of **subproblems**

    (2) relate the subproblems via a **recurrence**

    (3) find an **efficient implementation** of the recurrence (top down or bottom up)

    (4) **reconstruct the solution** from the DP table

DP Practice

Problem 2. *Dynamic Programming*

The dark lord Sauron loves to destroy the kingdoms of Middle Earth. But he just can't catch a break, and is always eventually defeated. After a defeat, he requires three epochs to rebuild his strength and once again rise to destroy the kingdoms of Middle Earth. In this problem, you will help Sauron decide in which epochs to rise and destroy the kingdoms of Middle Earth.

The input to the algorithm consists of the numbers $x_1, \ldots, x_n$ representing the number of kingdoms in each epoch. If Sauron rises in epoch i then he will destroy all x_i kingdoms, but will not be able to rise again during epochs $i+1, i+2$, or $i+3$. We call a set $S \subseteq \{1, \ldots, n\}$ of epochs *valid* if it satisfies this constraint that $|i-j| \geq 4$ for all $i, j \in S$, and its *value* is $\sum_{i \in S} x_i$. You will design an algorithm that outputs a valid set of epochs with the maximum possible value.

Example: Suppose there are $(1,7,8,2,6,3)$ kingdoms of Middle Earth in epochs $1, \ldots, 6$. Then the optimal set of epochs for Sauron to rise up and destroy the kingdoms of Middle Earth is $S = \{2,6\}$, during which he destroys 10 kingdoms, 7 in the 2nd epoch and 3 in the 6th epoch.

Using DP…

*** describe the set of subproblems you consider**

*** give a recurrence expressing the solution to each subproblem in terms of the solution to smaller subproblems**

***sketch pseudocode of your algorithm & give the runtime**

***describe how you would recover the solution (epochs) if asked**

DP Practice