南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

Algorithm Design and Analysis (H)

CS216

Instructor: Shan CHEN (陈杉)

chens3@sustech.edu.cn

(slides edited from Prof. Shiqi Yu)

SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

Dynamic Programming

Algorithmic Paradigms

- **Greedy.** Process the input in some order, and myopically making irrevocable decisions to optimize some underlying criterion.

- **Divide-and-conquer.** Break up a problem into independent subproblems, solve each subproblem, and combine solution to subproblems to form solution to original problem.

fancy name for caching intermediate results for later use

- **Dynamic programming.** Break up a problem into a series of overlapping subproblems, combine solutions to smaller subproblems to form solution to large subproblem.

Dynamic Programming History

- **Richard Bellman.** Pioneered the systematic study of dynamic programming in 1950s.

- **Etymology of "dynamic programming".**
  - Dynamic programming = planning over time.
  - Secretary of Defense had pathological fear to mathematical research.
  - Bellman sought a "dynamic" adjective to avoid conflict.

> "...it's impossible to use the word dynamic in a pejorative sense... Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to."

Reference: Bellman, R. E. *Eye of the Hurricane, An Autobiography.*

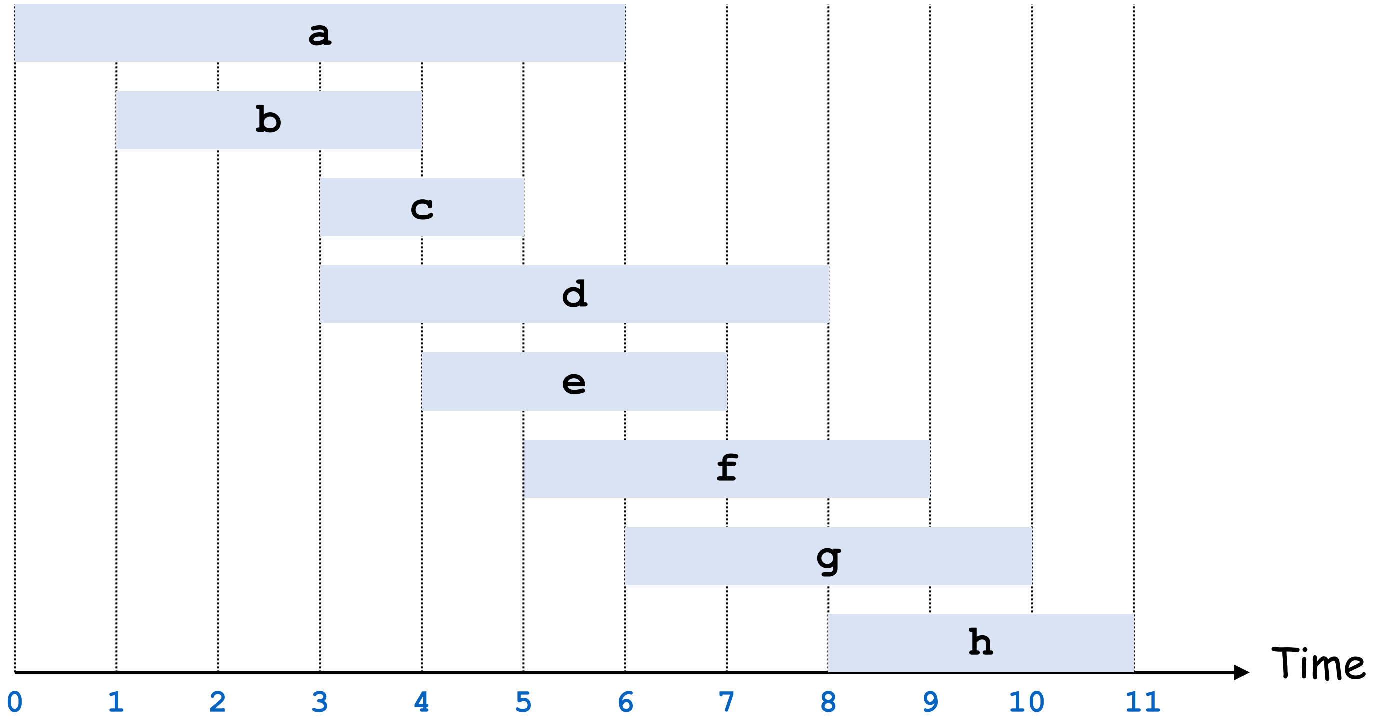
Dynamic Programming Applications

- **Application areas.**
  - Bioinformatics.
  - Control theory.
  - Information theory.
  - Operations research.
  - Computer science: theory, graphics, AI, compilers, systems, ….
- **Some famous dynamic programming algorithms.**
  - Unix diff for comparing two files.
  - Viterbi for hidden Markov models.
  - Needleman-Wunsch/Smith-Waterman for sequence alignment.
  - Bellman-Ford-Moore for shortest path routing in networks.

1. Weighted Interval Scheduling

Weighted Interval Scheduling

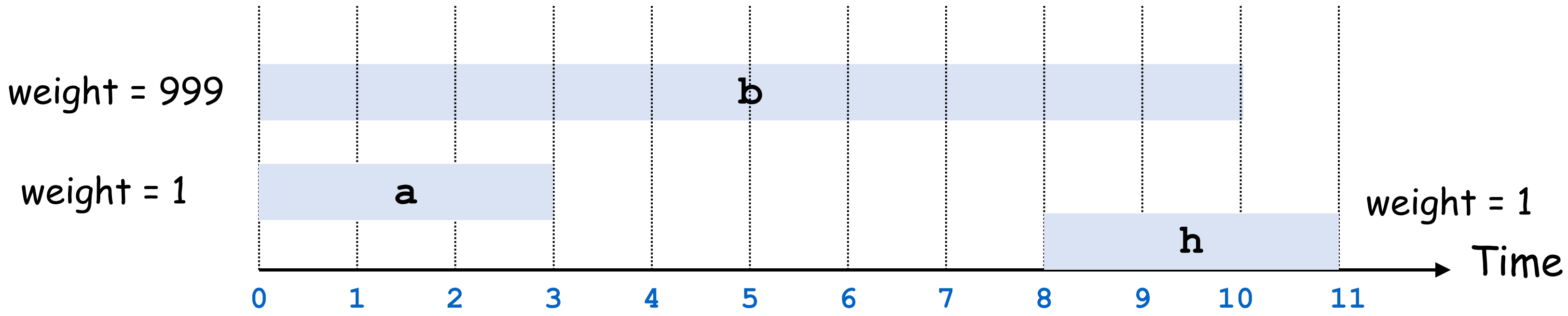
- **Weighted interval scheduling problem.**
  - Job j starts at s_j, finishes at f_j, and has weight/value v_j .
  - Two jobs compatible if they don't overlap.
  - **Goal:** find maximum weight subset of mutually compatible jobs.



Recall: Unweighted Interval Scheduling

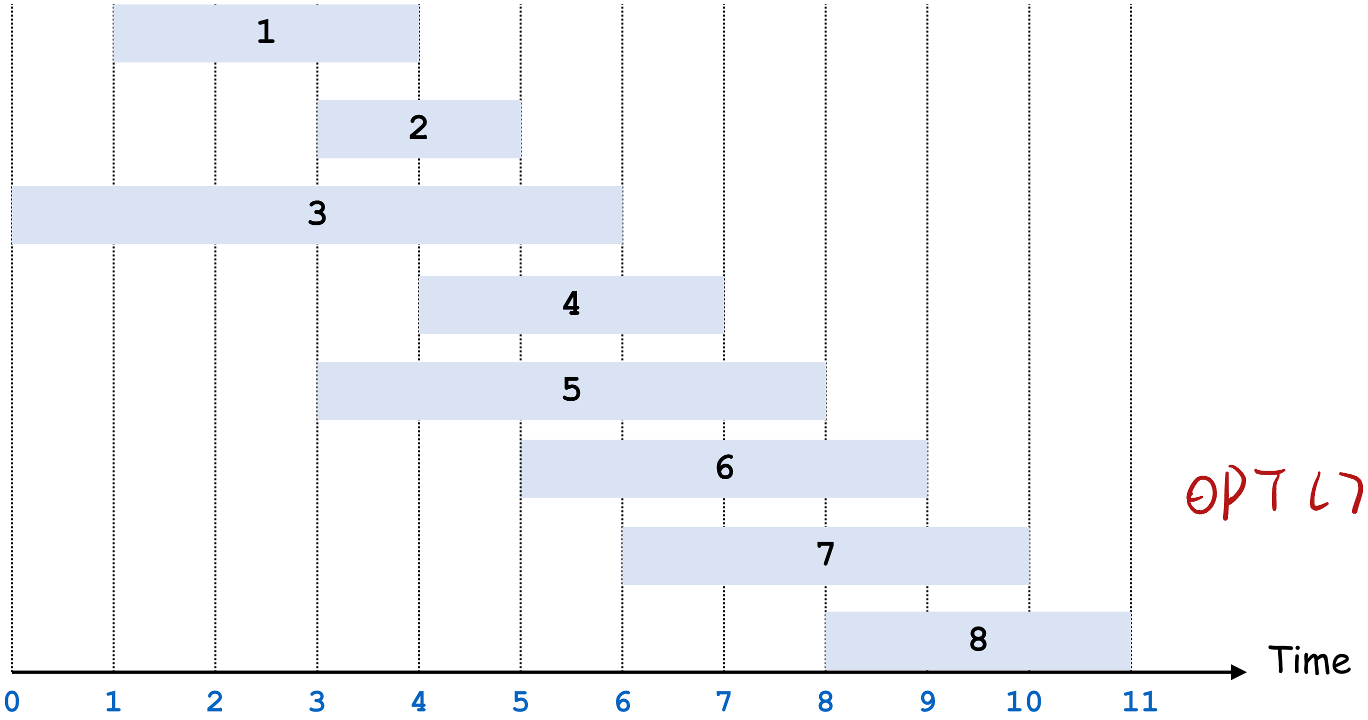
- **Recall.** Greedy algorithm works if all weights are 1.
  - Consider jobs in ascending order of finish time.
  - Add job to subset if it is compatible with previously chosen jobs.

- **Observation.** Greedy algorithm fails spectacularly for weighted version.



Weighted Interval Scheduling

- **Convention.** Order jobs by finish time: $f_1 \leq f_2 \leq . . . \leq f_n$.
- **Def.** p(j) = largest index i < j such that job i is compatible with j.
- Ex: p(8) = 5, p(7) = 3, p(2) = 0.



OPT(7) $\begin{cases} \text{chose 7 + opt(3)} \\ \times\text{chose 7} \rightarrow \text{opt(6)} \end{cases}$

Dynamic Programming: Binary Choice

- **Def.** OPT(j) = max weight of any subset of mutually compatible jobs for subproblem consisting only of jobs 1, 2, ..., j.
- **Goal.** OPT(n) = max weight of any subset of mutually compatible jobs.
- **Bellman equation.**

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\{\, OPT(j-1),\ w_j + OPT(p(j))\,\} & \text{if } j > 0 \end{cases}$$

  - **Case 1**. OPT(j) does not select job j.
    - ✓Must be optimal solution to problem consisting of remaining jobs 1, 2, ..., j - 1.
  - **Case 2.** OPT(j) selects job j.
    - ✓Collect weight v_j. (Can't use incompatible jobs p(j) + 1, p(j) + 2, ..., j – 1.)
    - ✓Must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., p(j).

Weighted Interval Scheduling: Brute Force

- **Brute force algorithm.**

```
Input: n, s1,…,sn , f1,…,fn , v1,…,vn

Sort jobs by finish times and renumber so that f1 ≤ f2 ≤ ... ≤ fn.
Compute p(1), p(2), …, p(n) via binary search

return Compute-Opt(n)


Compute-Opt(j) {
   if (j = 0)
      return 0
   else
      return max{ Compute-Opt(j - 1), vj + Compute-Opt(p(j)) }
}
```

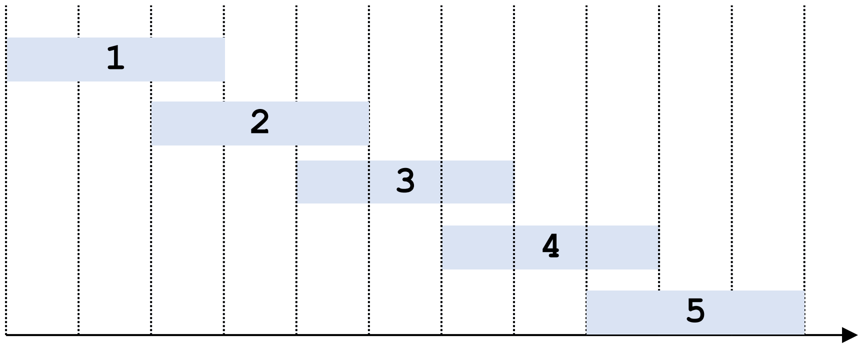
$O(n\log n)$

recursion could be very expensive!

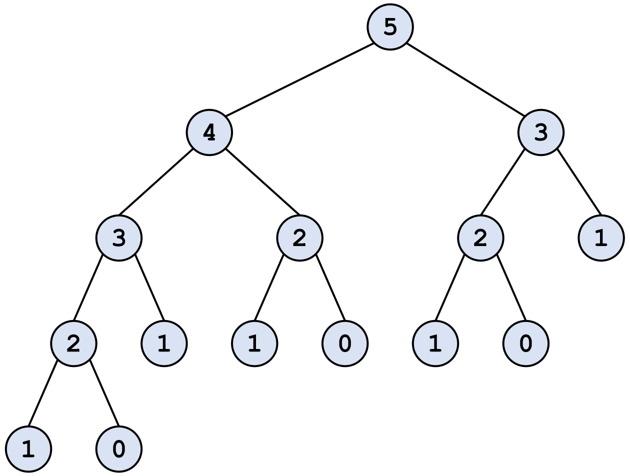
Weighted Interval Scheduling: Brute Force

- **Observation.** Recursive algorithm could be spectacularly slow due to overlapping subproblems $\Rightarrow$ exponential-time algorithm.

- **Ex.** For family of "layered" instances, the number of recursive calls grows like Fibonacci sequence.



$p(1) = 0$, $p(j) = j-2$



depth $\approx$ n, number of recursive calls $\approx 2^n$

Weighted Interval Scheduling: Memorization

- **Memorization.** Cache results of subproblem j in M[j] to avoid solving the same subproblem more than once.
- **Dynamic programming algorithm (top-down).**

```
Input: n, s1,…,sn , f1,…,fn , v1,…,vn

Sort jobs by finish times and renumber so that f1 ≤ f2 ≤ ... ≤ fn.
Compute p(1), p(2), …, p(n) via binary search

M[0] = 0   ← M[] is global array

return M-Compute-Opt(n)

M-Compute-Opt(j) {
   if (M[j] is uninitialized)
      M[j] = max{ M-Compute-Opt(j - 1), vj + M-Compute-Opt(p(j)) }
   return M[j]
}
```

Weighted Interval Scheduling: Running Time

- **Claim.** Memorized version of algorithm takes O(*n log n*) time.
- **Pf.**
  - Sort by finish time: O(*n log n*).
  - Computing p(·) : O(*n log n*) via binary search.

  - `M-Compute-Opt(j)`: each invocation takes O(1) time and either
    - ✓returns an existing value `M[j]`; or
    - ✓initializes `M[j]` and makes 2 recursive calls
      - at most *n* uninitialized `M[]` entries ⇒ at most *2n* recursive calls.
  - Overall running time of `M-Compute-Opt(n)` is O(*n*). ▪

Weighted Interval Scheduling: Finding a Solution

- **Q.** DP algorithm computes optimal value. How to find optimal solution?
- **A.** Do post-processing by calling `Find-Solution(n)`.

```
Find-Solution(j) {
   if (j = 0)
      return empty set
   else if (M[j] > M[j - 1]) → chose job j.
      return {j} ∪ Find-Solution(p(j))
   else                      → × chose job j.
      return Find-Solution(j - 1)
}
```

$M[j] = \max\{ M[j-1], v_j + M[p(j)] \}$

- **Running time.** O(n), because number of recursive calls ≤ n.

Weighted Interval Scheduling: Bottom-Up

- **Dynamic programming algorithm (bottom-up).** Unwind recursion.

```
Input: n, s_1,…,s_n , f_1,…,f_n , v_1,…,v_n

Sort jobs by finish times and renumber so that f_1 ≤ f_2 ≤ ... ≤ f_n.
Compute p(1), p(2), …, p(n) via binary search

M[0] = 0
for j = 1 to n
    M[j] = max{ M[j - 1], v_j + M[p(j)] }

return M[n]
```

previously computed M[] values

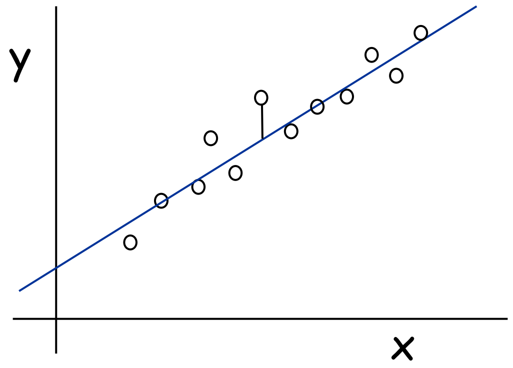
- **Running time.** The bottom-up version takes O(*n log n*) time

2. Segmented Least Squares

Least Squares

- **Least squares.** Foundational problem in statistics.
  - Given n points in the plane: $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$.
  - Find a line $y = ax + b$ that minimizes the sum of the squared errors (SSE).

$$SSE = \sum_{i=1}^{n} (y_i - ax_i - b)^2$$



- **Solution.** Calculus ⇒ min error is achieved when

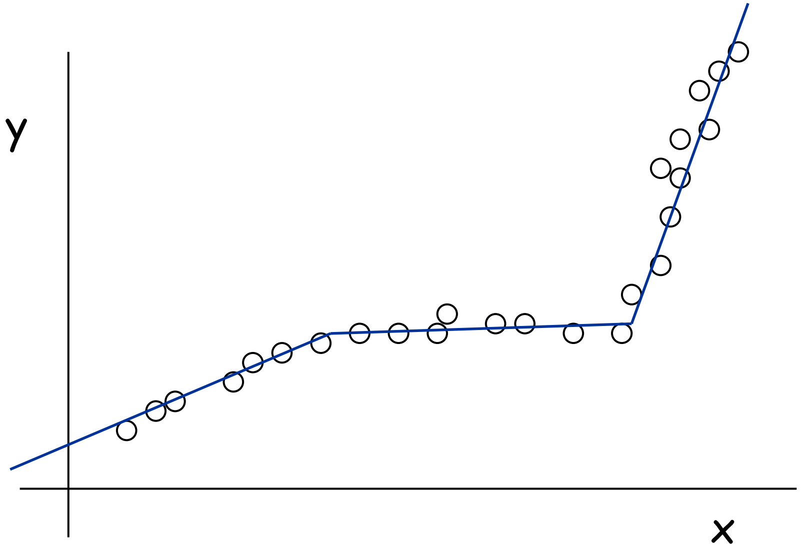
$$a = \frac{n \sum_i x_i y_i - (\sum_i x_i)(\sum_i y_i)}{n \sum_i x_i^2 - (\sum_i x_i)^2}, \quad b = \frac{\sum_i y_i - a \sum_i x_i}{n}$$

Segmented Least Squares

- **Segmented least squares.**
  - Points lie roughly on a sequence of several line segments.
  - Given n points in the plane $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ with $x_1 < x_2 < \ldots < x_n$, find a sequence of lines that minimizes f(x) (x is the problem input).
- **Q.** What's a reasonable choice for f(x) to balance accuracy and parsimony?

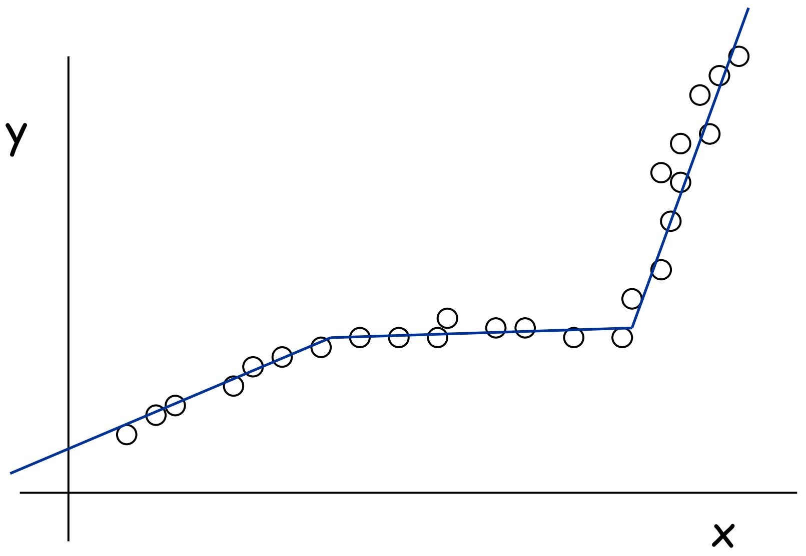
goodness of fit

number of lines



Segmented Least Squares

- **Segmented least squares.**
  - Points lie roughly on a sequence of several line segments.
  - Given n points in the plane $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ with $x_1 < x_2 < \ldots < x_n$, find a sequence of lines that minimizes $f(x) = E + cL$ (for some constant $c > 0$).
    - ✓E = sum of the sums of the squared errors (SSEs) in each segment
    - ✓L = number of lines



Dynamic Programming: Multiway Choice

- **Def.**
  - $OPT(j)$ = minimum cost for points $p_1, p_2, \ldots, p_j$.
  - e_{ij} = minimum SSE for points $p_i, p_{i+1}, \ldots, p_j$.
- **Goal.** OPT(n)
- **To compute OPT(j):**
  - Last segment uses points $p_i, p_{i+1}, \ldots, p_j$ for some $i \le j$.
  - Cost = $e_{ij} + c + OPT(i - 1)$.
- **Bellman equation.**

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \min\limits_{1 \le i \le j} \{ e_{ij} + c + OPT(i-1) \} & \text{if } j > 0 \end{cases}$$

Segmented Least Squares: Algorithm

- **Dynamic programming algorithm (bottom-up).**

```
INPUT: n, p1,…,pn , c

M[0] = 0
for j = 1 to n
   for i = 1 to j
      Compute the minimum SSE eij for the segment pi,…, pj

for j = 1 to n
   M[j] = min 1 ≤ i ≤ j { eij + c + M[i - 1] }

return M[n]
```

$O(n^3)$

$O(n)$

precompute $O(n)$

e_{ij} = combination of $\sum_{k=1}^{i} x_k, \quad \sum_{k=1}^{i} y_k, \quad \sum_{k=1}^{i} x_k^2, \quad \sum_{k=1}^{i} x_k y_k$

- **Remark.** Can be improved to O(n^2) time.
  - ➢ Compute e_{ij} in O(1) time using precomputed cumulative sums.

3. Knapsack Problem

Knapsack Problem

- **Knapsack problem.**
  - Given n objects and a "knapsack".
  - Item i weighs $w_i > 0$ kilograms and has value $v_i > 0$.
  - Knapsack has capacity of W kilograms.
  - **Goal:** fill knapsack to maximize total value of items taken.

- **Greedy approaches fail to work.**
  - repeatedly add item with maximum v_i.
  - repeatedly add item with maximum w_i.
  - repeatedly add item with maximum ratio v_i / w_i.
- **Ex.** Optimal subset for W = 11 is { 3, 4 } with value 40.

#	value	weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Dynamic Programming: False Start

- **Def.** OPT(i) = maximum value within weight capacity W using items 1, …, i.
- **Goal.** OPT(n)
- **To compute OPT(i):**
  - Case 1: OPT(i) does not select item i.
    - ✓OPT(i) selects best of { 1, 2, …, i - 1 }
  - Case 2: OPT(i) selects item i.
    - ✓not sure which other items should be excluded
    - ✓not sure how to relate to OPT(j) for j < i

- **Conclusion.** Need more sub-problems!

Dynamic Programming: Adding a Variable

- **Def.** OPT(i, w) = maximum value within weight limit w using items 1, …, i.
- **Goal.** OPT(n, W)
- **To compute OPT(i, w):**
  - Case 1: OPT(i, w) does not select item i.
    - ✓OPT(i, w) selects best of { 1, 2, …, i - 1 } within weight limit w
  - Case 2: OPT(i) selects item i.
    - ✓OPT(i, w) collects value v_i
    - ✓OPT(i, w) selects best of { 1, 2, …, i - 1 } within weight limit $w - w_i$
- **Bellman equation.**

$$OPT(i,w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1,w) & \text{if } w_i > w \\ \max\{\, OPT(i-1,w),\ v_i + OPT(i-1, w-w_i)\,\} & \text{otherwise} \end{cases}$$

Knapsack Problem: Algorithm

- **Dynamic programming algorithm (bottom-up).**

```
Input: n, W, w1,…,wn, v1,…,vn

for w = 0 to W
   M[0, w] = 0

for i = 1 to n
   for w = 1 to W
      if (wi > w)
         M[i, w] = M[i - 1, w]
      else
         M[i, w] = max { M[i - 1, w], vi + M[i - 1, w - wi ] }

return M[n, W]
```

- **Running time.** O(nW)

Knapsack Algorithm: Demo

W + 1 (columns); n + 1 (rows)

W = 11	0	1	2	3	4	5	6	7	8	9	10	11
ϕ	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	$1_{\{1\}}$	$1_{\{1\}}$	$1_{\{1\}}$	$1_{\{1\}}$	$1_{\{1\}}$	$1_{\{1\}}$	$1_{\{1\}}$	$1_{\{1\}}$	$1_{\{1\}}$	$1_{\{1\}}$	$1_{\{1\}}$
{ 1, 2 }	0	$1_{\{1\}}$	$6_{\{2\}}$	$7_{\{1,2\}}$	$7_{\{1,2\}}$	$7_{\{1,2\}}$	$7_{\{1,2\}}$	$7_{\{1,2\}}$	$7_{\{1,2\}}$	$7_{\{1,2\}}$	$7_{\{1,2\}}$	$7_{\{1,2\}}$
{ 1, 2, 3 }	0	$1_{\{1\}}$	$6_{\{2\}}$	$7_{\{1,2\}}$	$7_{\{1,2\}}$	$18_{\{3\}}$	$19_{\{1,3\}}$	$24_{\{2,3\}}$	$25_{\{1,2,3\}}$	$25_{\{1,2,3\}}$	$25_{\{1,2,3\}}$	$25_{\{1,2,3\}}$
{ 1, 2, 3, 4 }	0	$1_{\{1\}}$	$6_{\{2\}}$	$7_{\{1,2\}}$	$7_{\{1,2\}}$	$18_{\{3\}}$	$22_{\{4\}}$	$24_{\{2,3\}}$	$28_{\{2,4\}}$	$29_{\{1,2,4\}}$	$29_{\{1,2,4\}}$	$40_{\{3,4\}}$
{ 1, 2, 3, 4, 5 }	0	$1_{\{1\}}$	$6_{\{2\}}$	$7_{\{1,2\}}$	$7_{\{1,2\}}$	$18_{\{3\}}$	$22_{\{4\}}$	$28_{\{5\}}$	$29_{\{1,5\}}$	$34_{\{2,5\}}$	$34_{\{2,5\}}$	$40_{\{3,4\}}$

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

optimal subset: { 3, 4 } value = 18 + 22 = 40

$$OPT(i,w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1,w) & \text{if } w_i > w \\ \max\{\, OPT(i-1,w),\ v_i + OPT(i-1,w-w_i) \,\} & \text{otherwise} \end{cases}$$

Knapsack Problem: Remarks

- DP algorithm depends crucially on assumption that weights are integral.

- Running time O(nW) is not polynomial in input size! "pseudo-polynomial"

- Decision version of Knapsack is NP-complete. [Chapter 8]
  - Can a value of at least V be achieved under a restriction of a certain capacity W?

- Knapsack approximation algorithm. There exists a polynomial-time algorithm that produces a feasible solution that has value within 0.01% of optimum. [Section 11.8]

Exercise: Coin Changing

- **Observation.** Greedy algorithm is sub-optimal for US postal denominations: 1, 10, 21, 34, 70, 100, 350, 1225, 1500.
- **Counterexample.** 140¢.
  - Greedy: 100, 34, 1, 1, 1, 1, 1, 1.
  - Optimal: 70, 70.

- **Q.** How to solve this problem via DP?
- **Problem.** Given n coin denominations { d_1, d_2, ..., d_n } and a target value V, find the fewest coins needed to make change for V (or report impossible).

Dynamic Programming: Summary

- **Outline.**
  - Define a collection of subproblems. typically only polynomial number
  - Solution to original problem can be computed from subproblems.
  - Natural ordering of subproblems from "smallest" to "largest" that enables determining a solution to a subproblem from solutions to smaller subproblems.

- **Techniques.**
  - Binary choice: weighted interval scheduling.
  - Multiway choice: segmented least squares.
  - Adding a new variable: knapsack problem.

- **Top-down vs. bottom-up dynamic programming.** Opinions differ.