Potes enim videre in hac margine, qualiter hoc operati fuimus, scilicet quod iunximus primum numerum cum secundo, videlicet 1 cum 2; et secundum cum tercio; et tercium cum quarto; et quartum cum quinto, et sic deinceps. . . .

[You can see in the margin here how we have worked this; clearly, we combined the first number with the second, namely 1 with 2, and the second with the third, and the third with the fourth, and the fourth with the fifth, and so forth. . . .]

— Leonardo Pisano, *Liber Abaci* (1202)

Those who cannot remember the past are condemned to repeat it.

— Jorge Agustín Nicolás Ruiz de Santayana y Borrás,
The Life of Reason, Book I: Introduction and Reason in Common Sense (1905)

You know what a learning experience is?
A learning experience is one of those things that says,
"You know that thing you just did? Don't do that."

— Douglas Adams, *The Salmon of Doubt* (2002)

From Jeff Erickson, alogrithms.wtf

Lecture 4:
Dynamic Programming I

William Umboh
School of Computer Science

THE UNIVERSITY OF SYDNEY

General techniques in this course

- Greedy algorithms [W2]
- Divide & Conquer algorithms [W3]
- Dynamic programming algorithms [W4-5]
- Network flow algorithms [W6-7]

Algorithmic Paradigms

- **Greedy.** Build up a solution incrementally, myopically optimizing some local criterion.

- **Divide-and-conquer.** Break up a problem into two sub-problems, solve each sub-problem *independently*, and combine solution to sub-problems to form solution to original problem.

- **Dynamic programming.** Break up a problem into a series of *overlapping* sub-problems, and build up solutions to larger and larger sub-problems.

Dynamic Programming Applications

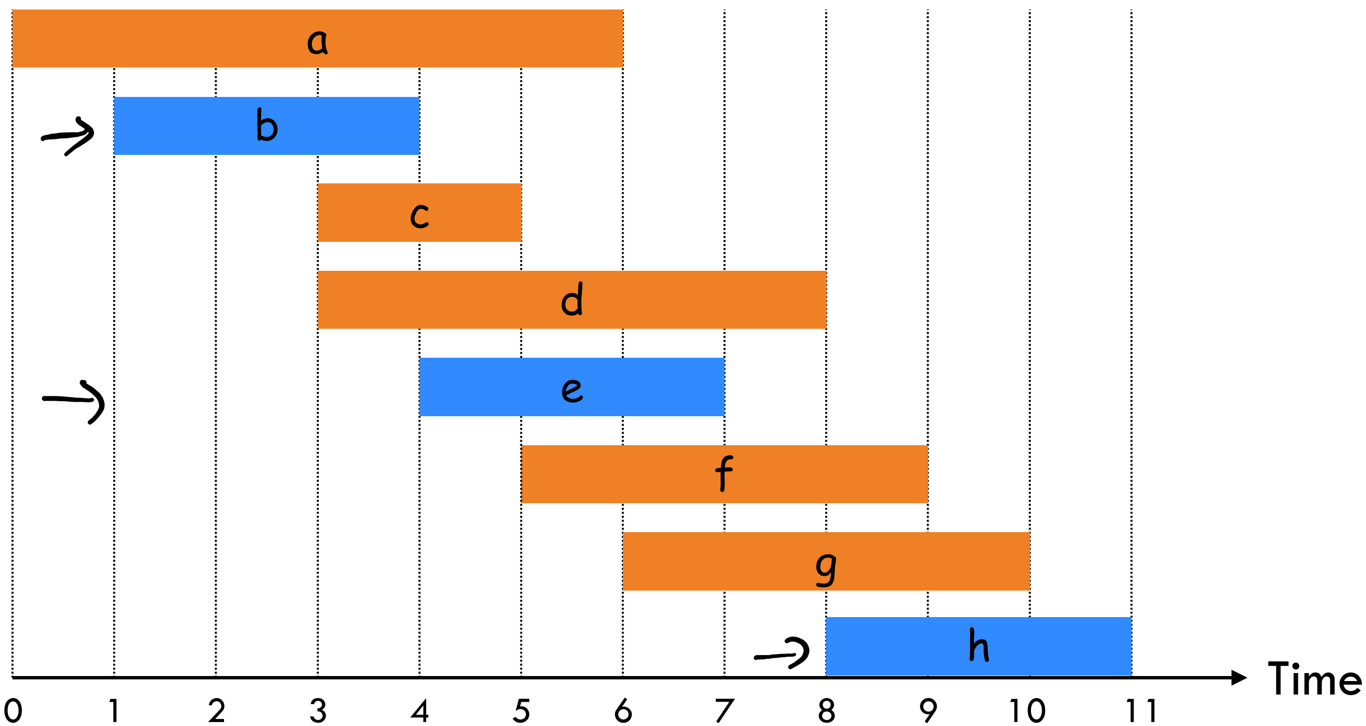
- Areas.
  - Bioinformatics.
  - Control theory.
  - Information theory.
  - Operations research.
  - Computer science: theory, graphics, AI, systems, ….

- Some famous dynamic programming algorithms.
  - Viterbi for hidden Markov models.
  - Unix diff for comparing two files.
  - Smith-Waterman for sequence alignment.
  - Bellman-Ford for shortest path routing in networks.
  - Cocke-Kasami-Younger for parsing context free grammars.

6.1 Weighted Interval Scheduling

Recall Interval Scheduling (Lecture 2)

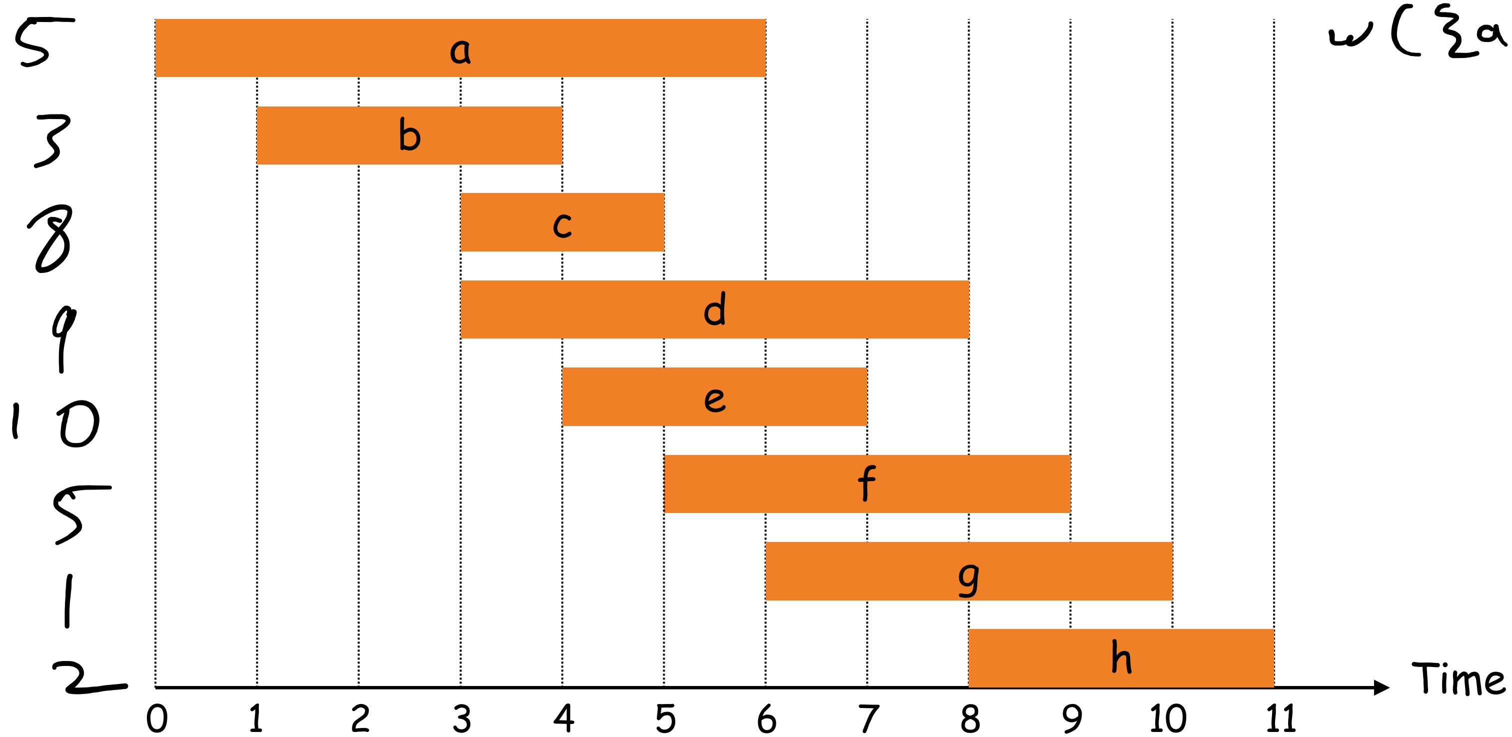
- Interval scheduling.
  - **Input:** Set of n jobs. Each job i starts at time s_i and finishes at time f_i.
  - Two jobs are compatible if they don't overlap in time.
  - **Goal:** find maximum subset of mutually compatible jobs.
  - There exists a greedy algorithm (Earliest Finish Time)



Weighted Interval Scheduling

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

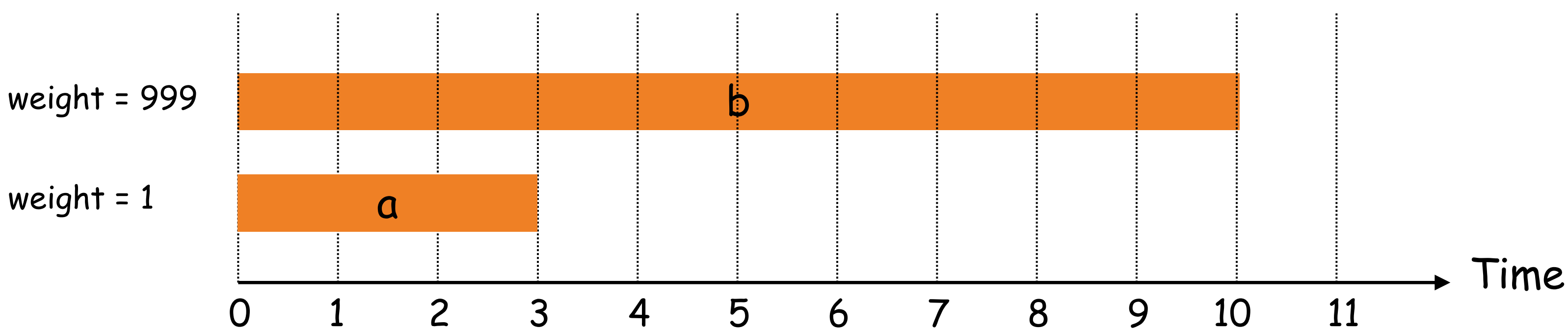
$w(\{a, g\})=6$



Unweighted Interval Scheduling Review

- 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 can fail if arbitrary weights are allowed.



Key steps: Dynamic programming

Formulate the problem recursively.

1. Define subproblems
2. Find recurrence relating subproblems
3. Solve the base cases

Similar to what we did for D&C

Transform recurrence into an efficient algorithm

Recursive formulation of MCS

Maximum contiguous subarray (MCS) in A[1..n]

- Three subproblems:
  a) MCS in A[1..n/2]
  b) MCS in A[n/2+1..n]
  c) MCS that spans A[n/2, n/2 + 1]

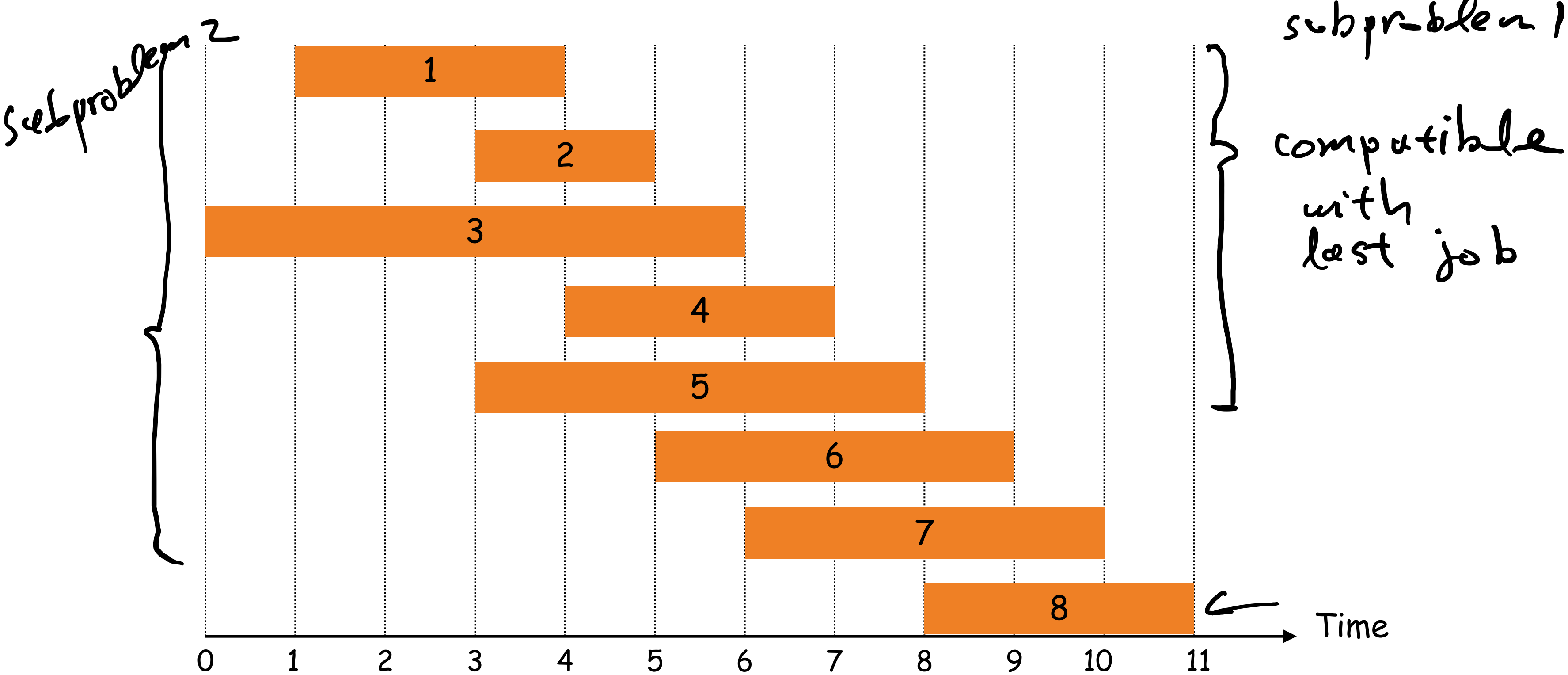
  MCS of A[1..n] = max of the optimal of the subproblems

Weighted Interval Scheduling

Observation 1: OPT either includes last job or not.

- If it does, then it also includes the optimal solution for the remaining jobs compatible with last job
- Else, it is the optimal solution for remaining jobs

Observation 2: Subproblems are prefixes



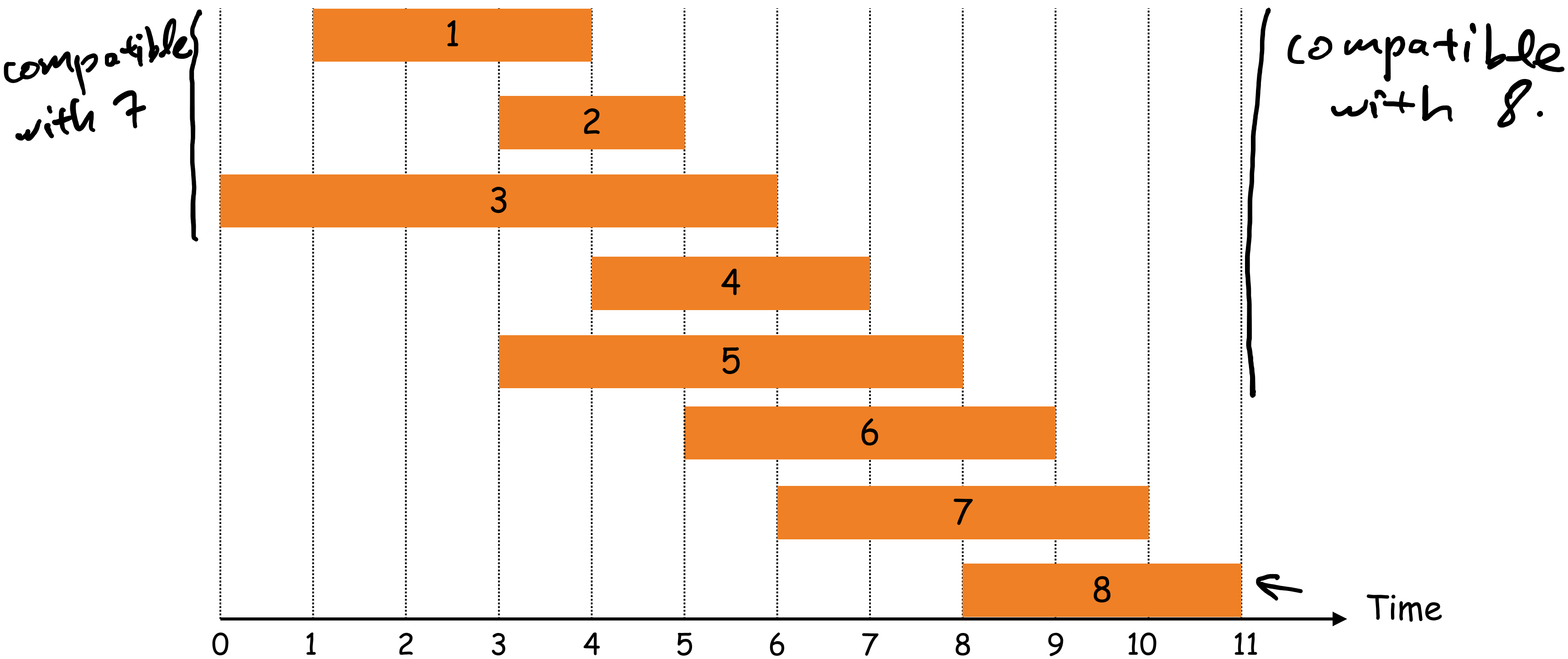
Dynamic Programming: Weighted Interval Scheduling

Notation. Label jobs by finishing time: $f_1 \leq f_2 \leq \ldots \leq f_n$.

Def. $p(j)$ = largest index $i < j$ such that job i is compatible with j.

All jobs $k < p(j)$ are compatible with j.

Ex: $p(8) = 5$, $p(7) = 3$, $p(2) = 0$.



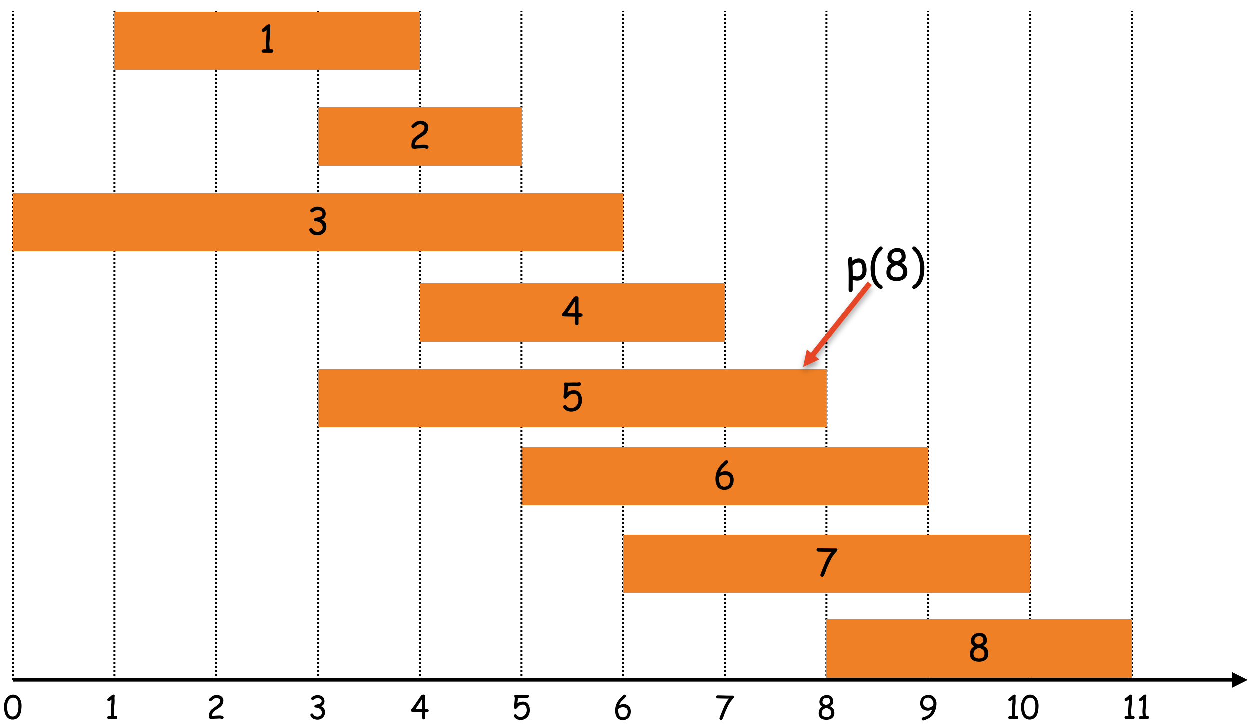
Dynamic Programming: Weighted Interval Scheduling

Notation. Label jobs by finishing time: $f_1 \leq f_2 \leq \ldots \leq f_n$.

Def. p(j) = largest index i < j such that job i is compatible with j.

Step 1: Define subproblems

OPT(j) = value of optimal solution to the subproblem consisting of job requests 1, 2, ..., j.

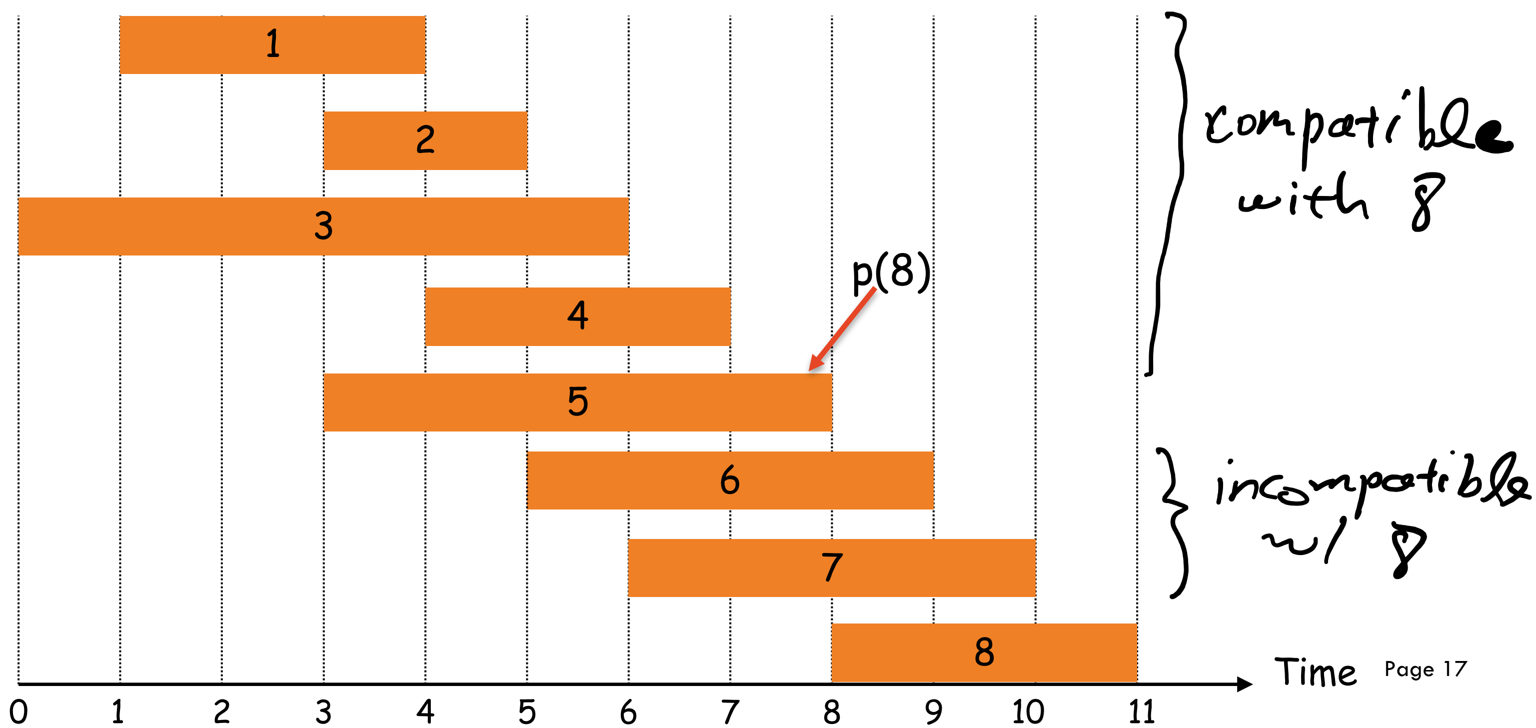


Dynamic Programming: Weighted Interval Scheduling

OPT(j)

Step 2: Find recurrences

- **Case 1:** OPT selects job 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)



Dynamic Programming: Weighted Interval Scheduling

Step 2: Find recurrences

- **Case 1:** OPT selects job 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)

$$\boxed{OPT(j) = \underbrace{v_j + OPT(p(j))}_{\text{Case 1}}}$$

weight

Dynamic Programming: Weighted Interval Scheduling

Step 2: Find recurrences

- **Case 1:** OPT selects job 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)

- **Case 2:** OPT does not select job j.
  - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., j-1

$$\boxed{\mathrm{OPT}(j) = \underbrace{v_j + \mathrm{OPT}(p(j))}_{\text{Case 1}}}$$

Dynamic Programming: Weighted Interval Scheduling

Step 2: Find recurrences

- **Case 1:** OPT selects job 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)
- **Case 2:** OPT does not select job j.
  - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., j-1

Ex $OPT(8) = \max\{v_8 + OPT(5), OPT(7)\}$.

$$\boxed{OPT(j) = \max\{\underbrace{v_j + OPT(p(j))}_{\text{Case 1}}, \underbrace{OPT(j-1)}_{\text{Case 2}}\}}$$

Dynamic Programming: Weighted Interval Scheduling

Step 3: **Solve the base cases**

OPT(0) = 0

can be 0

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

why not OPT(1) = v_1 ?

Done…more or less

Weighted Interval Scheduling: Naïve Recursion

– Naïve recursion algorithm.

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

Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.

Compute p(1), p(2), …, p(n)

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

return Compute-Opt(n)
```

Weighted Interval Scheduling: Naïve Recursion

– Naïve recursion algorithm.

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

Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.

Compute p(1), p(2), …, p(n)

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

return Compute-Opt(n)
```

Running time: $T(n) = T(n-1) + T(p(n)) + O(1) = ?$

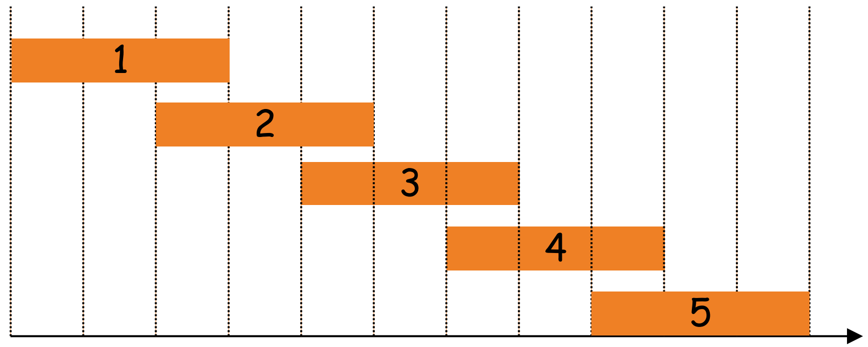
Weighted Interval Scheduling: Naïve Recursion

Observation. Recursive algorithm is slow because of exponential recursive calls $\Rightarrow$ exponential algorithms.

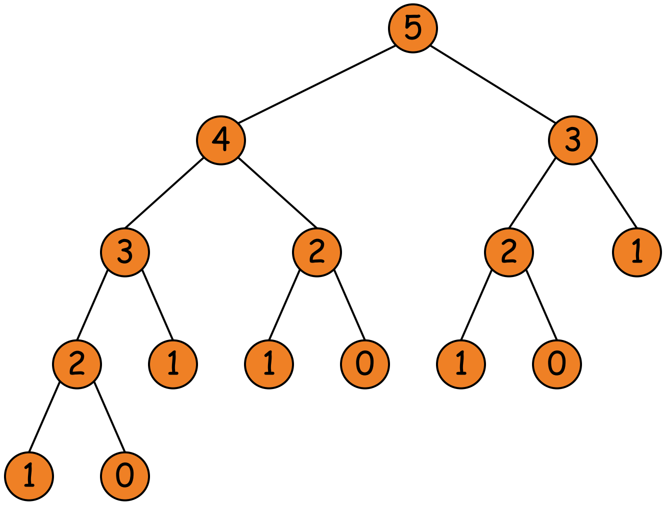
Example. Number of recursive calls for family of "layered" instances grows like Fibonacci sequence:

$T(n) = T(n-1) + T(n-2) + c$

$p(n) = n-2$



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



Exponential recursive calls
aka multiply and surrender

Not memorization!

Weighted Interval Scheduling: Memoization

Memoization. Store results of each sub-problem; lookup when needed.

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

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

for j = 1 to n
   M[j] = empty

M[0] = 0

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

return Compute-Opt(n)
```

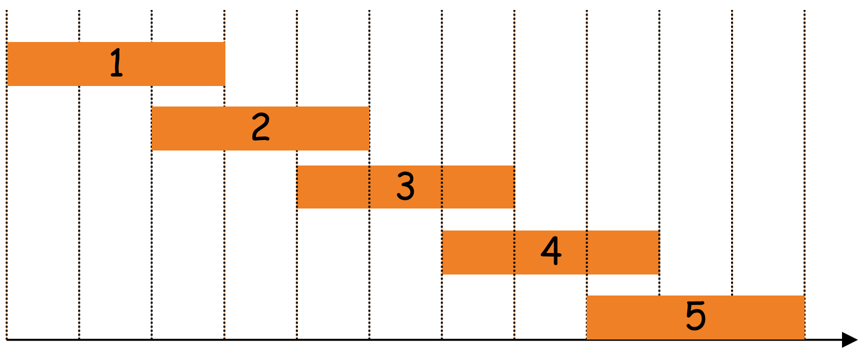
stores values for each subproblem.

Preprocessing

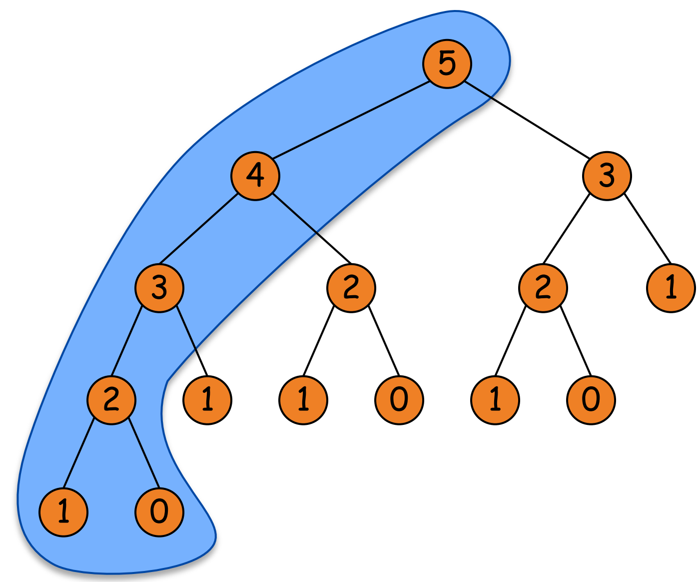
Running time: O(n log n)

Weighted Interval Scheduling: Running Time

Claim. Memoized version of algorithm takes O(n log n) time.



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



Remark: O(n) if jobs are pre-sorted by start and finish times.

Weighted Interval Scheduling: Running Time

Claim. Memoized version of algorithm takes O(n log n) time.

- Sort by finish time: O(n log n).
- Computing p(·) : O(n) after sorting by start time.

- `Compute-Opt(j)`: each call takes O(1) time because it either
  - (i) returns an existing value `M[j]`
  - (ii) fills in one new entry `M[j]` and makes two new recursive calls

- Overall time is O(1) times the number of calls to `Compute-Opt(j)`.

- Progress measure K = # nonempty entries of `M[]`.
  - initially K = 1 and $K \leq n + 1$
  - Case (ii) increases K by 1 $\Rightarrow$ at most 2n recursive calls.

- Overall running time of `Compute-Opt(n)` is O(n). ▪

Remark: O(n) if jobs are pre-sorted by start and finish times.

Weighted Interval Scheduling: Bottom-Up

- Bottom-up dynamic programming. Unwind recursion.
- This is the style we will use in rest of lectures
- Key insight: M[i] only depends on values M[k] for k < i

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

Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.

Compute p(1), p(2), …, p(n)

Iterative-Compute-Opt {
   M[0] = 0
   for j = 1 to n
      M[j] = max(vj + M[p(j)], M[j-1])
   return M[n]
}
```

Weighted Interval Scheduling: Finding a Solution

Question. Dynamic programming algorithm computes optimal value.

What if we want the solution itself?
Answer. Do some post-processing.

```
Run Compute-Opt(n)  → array M
Run Find-Solution(n)

Find-Solution(j) {
   if (j = 0)
      output nothing
   else if (vj + M[p(j)] > M[j-1])   ← picked job j
      print j
      Find-Solution(p(j))
   else
      Find-Solution(j-1)
}
```

\# of recursive calls $\leq n \Rightarrow O(n)$.

Maximum-sum contiguous subarray

Given an array A[ ] of n numbers, find the maximum sum found in any contiguous subarray

A zero-length subarray has maximum 0

Example:

1	-2	7	5	6	-5	5	8	1	-6

Maximum-sum contiguous subarray

Given an array A[ ] of n numbers, find the maximum sum found in any contiguous subarray

A zero-length subarray has maximum 0

Example:

1	-2	7	5	6	-5	5	8	1	-6

Divide-and-conquer algorithm

Maximum contiguous subarray (MCS) in A[1..n]

- Three cases:
  a) MCS in A[1..n/2]
  b) MCS in A[n/2+1..n]
  c) MCS that spans A[n/2, n/2 + 1]

- (a) & (b) can be found **recursively**

- (c) can be found in two steps
  - Consider MCS in A[1..n/2] ending in A[n/2].
  - Consider MCS in A[n/2+1..n] starting at A[n/2+1].
  - Sum these two maximum

Dynamic programming

Step 1: Define subproblems

OPT(i) = value of optimal subarray ending at i (possibly empty)

Dynamic programming algorithm

Example 1:	6 -3 -2 3 -1 2
OPT[1] = 6	<u>6</u>
OPT[2] = 3	<u>6 -3</u>
OPT[3] = 1	<u>6 -3 -2</u>
OPT[4] = 4	<u>6 -3 -2 3</u>
OPT[5] = 3	<u>6 -3 -2 3 -1</u>
OPT[6] = 5	<u>6 -3 -2 3 -1 2</u>

$\boxed{\text{OPT[i] – optimal solution ending at i}}$

Dynamic programming algorithm

Example 2:	-2	5	-1	-5	3	-1	2
OPT[1] = 0	-2						
OPT[2] = 5	-2	5					
OPT[3] = 4	-2	5	-1				
OPT[4] = 0	-2	5	-1	-5			
OPT[5] = 3	-2	5	-1	-5	3		
OPT[6] = 2	-2	5	-1	-5	3	-1	
OPT[7] = 4	-2	5	-1	-5	3	-1	2

OPT[i] – value of optimal solution ending at i

Dynamic programming algorithm

Example 2:	-2	5	-1	-5	3	-1	2
OPT[1] = 0	-2						
OPT[2] = 5	-2	5					
OPT[3] = 4	-2	5	-1				
OPT[4] = 0	-2	5	-1	-5			
OPT[5] = 3	-2	5	-1	-5	3		
OPT[6] = 2	-2	5	-1	-5	3	-1	
OPT[7] = 4	-2	5	-1	-5	3	-1	2

Step 2: Find recurrences

2 cases:

(1) A[i] is not included in the optimal solution ending at i, i.e. it's the empty array

(2) A[i] is included. In this case, the optimal solution ending at i extends optimal solution ending at i - 1

$$\boxed{OPT[i] = \max\{OPT[i-1]+A[i], 0\}}$$

Dynamic programming algorithm

Step 3: Solve the base cases

Why can't we just take A[1]?

$$OPT[1] = \max(A[1], 0)$$

$OPT[0] = 0$ also suffices.

$$OPT[i] = \begin{cases} \max(A[1], 0) & \text{if } i=1 \\ \max\{OPT[i-1]+A[i], 0\} & \text{if } i>1 \end{cases}$$

Pseudo Code

OPT[i] – optimal solution ending at i

$OPT[1] = \max(A[1], 0)$

for i = 2 to n do

  $OPT[i] = \max(OPT[i-1]+A[i], 0)$ } O(n)

$MaxSum = \max_{1\leq i\leq n} OPT[i]$ } O(n)

Total time: O(n)

Longest increasing subsequence

Given a sequence of numbers X[1..n] find the longest increasing *subsequence* (i_1, i_2, …, i_k), that is a subsequence where numbers in the sequence are increasing.

5 2 8 6 3 6 9 7

Longest increasing subsequence

Given a sequence of numbers X[1..n] find the longest increasing *subsequence* $(i_1, i_2, \ldots, i_k)$, that is a subsequence where numbers in the sequence are increasing.

5 2 8 6 3 6 9 7

Longest increasing subsequence

Step 1: Define subproblems

- L[i] = length of the longest increasing subsequence that ends at i, including i itself
- L[1] = 1 (base case)

5 2 8 6 3 6 9 7

- Example:

{5} L[1] = 1   {2,6} L[4] = 2   L[7] = 4

{2} L[2] = 1   {2,3} L[5] = 2   L[8] = 4

{2,8} L[3] = 2   L[6] = 3

Longest increasing subsequence

Step 1: Define subproblems

- L[i] = length of the longest increasing subsequence that ends at i, including i itself
- L[1] = 1 (base case)

5 2 8 6 3 6 9 7

Step 2: Define recurrence

X[i] is in LIS ending at i, by definition, so it must extend the LIS ending at some j < i with X[j] < X[i]

$$L[i] = \max_{0<j<i} \{ L[j] + 1 \mid X[j] < X[i] \}$$

n times

n

Running time: ?

Note: In python, L[i] = max([L[j] + 1 for j in range(1, i) with X[j] < X[i]])

Longest increasing subsequence

Step 1: Define subproblems

- L[i] = length of the longest increasing subsequence that ends at i, including i itself
- L[1] = 1 (base case)

5 2 8 6 3 6 9 7

Step 2: Define recurrence

X[i] is in LIS ending at i, by definition, so it must extend the LIS ending at some j < i with X[j] < X[i]

$$L[i] = \max_{0<j<i} \{ L[j] + 1 \mid X[j] < X[i] \}$$

n times

n

such that

Running time: $O(n^2)$

Can we do better?

O(n log n) possible

6.4 Knapsack

A 1998 study of the Stony Brook University Algorithm Repository showed that, out of 75 algorithmic problems, the knapsack problem was the 18th most popular and the 4th most needed after kd-trees, suffix trees, and the bin packing problem.

First mentioned by Mathews in 1897.
"Knapsack problem" by Dantzig in 1930.

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 so as to maximize total value.
- **Example:** { 3, 4 } has value 40.

W = 11

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

- **Greedy:** repeatedly add item with maximum ratio v_i / w_i. ("best bang for buck")
- Ex: { 5, 2, 1 } achieves only value = 35 $\Rightarrow$ greedy not optimal.

Dynamic Programming: False Start

- **Definition.** OPT(i) = max profit subset of items 1, …, i.

  - **Case 1:** OPT does not select item i.
    - OPT selects best of { 1, 2, …, i-1 }

  - **Case 2:** OPT selects item i.
    - accepting item i does not immediately imply that we will have to reject other items
    - without knowing what other items were selected before i, we don't even know if we have enough room for i

Conclusion: Need subproblems with more structure!

Dynamic Programming: Adding a New Variable

Step 1: Define subproblems

OPT(i, w) = max profit subset of items 1, …, i
with weight limit w.

i = 5
w = 11

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

Dynamic Programming: Adding a New Variable

Step 2: Find recurrences

$OPT(i, w)$

- **Case 1:** OPT does not select item i.
  - OPT selects best of { 1, 2, …, i-1 } using weight limit w

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

i = 5
w = 11

$$OPT[i,w] = \underbrace{OPT[i-1,w]}_{\text{case 1}}$$

Dynamic Programming: Adding a New Variable

Step 2: Find recurrences

- **Case 1:** OPT does not select item i.
  - OPT selects best of { 1, 2, …, i-1 } using weight limit w
- **Case 2:** OPT selects item i.
  - new weight limit = $w - w_i$
  - OPT selects best of { 1, 2, …, i–1 } using this new weight limit

$$OPT[i,w] = \quad \underbrace{OPT[i-1,w]}_{\text{case 1}},\ \underbrace{v_i + OPT[i-1,w-w_i]}_{\text{case 2}}$$

Dynamic Programming: Adding a New Variable

Step 2: Find recurrences

$OPT(i, w) =$ optimal of $\{1, \dots, i\}$ w/ weight $\le w$

- **Case 1:** OPT does not select item i.
  - OPT selects best of { 1, 2, …, i-1 } using weight limit w
- **Case 2:** OPT selects item i.
  - new weight limit = $w - w_i$
  - OPT selects best of { 1, 2, …, i–1 } using this new weight limit

$$OPT[i,w] = \max\{\underbrace{OPT[i-1,w]}_{\text{case 1}},\ \underbrace{v_i + OPT[i-1,w-w_i]}_{\text{case 2}}\}$$

Dynamic Programming: Adding a New Variable

Step 3: Solve the base cases

$$OPT[0,w] = 0$$

Dynamic Programming: Adding a New Variable

- **Base case:** OPT[0,w] = 0

- **Case 1:** OPT does not select item i.
  - OPT selects best of { 1, 2, …, i-1 } using weight limit w

- **Case 2:** OPT selects item i.
  - new weight limit = $w - w_i$
  - OPT selects best of { 1, 2, …, i–1 } using new weight limit $w - w_i$

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

Knapsack Algorithm Recurrence: Example

OPT(i,w)

W + 1 →

n + 1 ↓

	0	1	2	3	4	5	6	7	8	9	10	11
∅	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, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	34	40

40

7+28

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

W = 11

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

Knapsack Problem: Bottom-Up

$M[i, w] = OPT(i, w)$

- **Knapsack.** Fill up an (n+1)-by-(W+1) array.

```
Input: n, 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]
```

Knapsack Algorithm: Bottom-Up Example

W + 1

n + 1

	0	1	2	3	4	5	6	7	8	9	10	11
∅	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, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	34	40

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

W = 11

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

Knapsack Algorithm

W + 1

n + 1

	0	1	2	3	4	5	6	7	8	9	10	11
∅	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, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	34	40

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

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

W = 11

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

Knapsack Problem: Running Time

If $W = 2^n$
time $= \Omega(2^n)$
input size $= O(n^2)$

What is input size?

Input size = O(n log W)

- **Running time:** Θ(nW).
  - Not polynomial in input size!
  - "Pseudo-polynomial" : polynomial in size of numbers not their bit length
  - Decision version of Knapsack is NP-complete.

- Knapsack approximation algorithm. There exists a polynomial algorithm (w.r.t. n) that produces a feasible solution that has value within 0.01% of optimum.

Dynamic Programming Summary

- Dynamic programming = smart recursion
- Recipe.
  - Characterize structure of problem step
  - Recursively define value of optimal solution.
  - Compute value of optimal solution.
  - Construct optimal solution from computed information.

- Dynamic programming techniques.
  - Binary choice: weighted interval scheduling.
  - Adding a new variable: knapsack.

Viterbi algorithm for HMM also uses DP to optimize a maximum likelihood tradeoff between parsimony and accuracy

- Top-down vs. bottom-up: different people have different intuitions.