

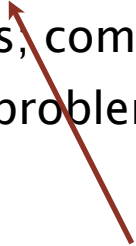
# Algorithmic paradigms

---

**Greed.** Process the input in some order, myopically making irrevocable decisions.

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

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



fancy name for  
caching intermediate results  
in a table for later reuse

# Dynamic programming history

---

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

## Etymology.

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



## THE THEORY OF DYNAMIC PROGRAMMING

RICHARD BELLMAN

1. **Introduction.** Before turning to a discussion of some representative problems which will permit us to exhibit various mathematical features of the theory, let us present a brief survey of the fundamental concepts, hopes, and aspirations of dynamic programming.

To begin with, the theory was created to treat the mathematical problems arising from the study of various multi-stage decision processes, which may roughly be described in the following way: We have a physical system whose state at any time  $t$  is determined by a set of quantities which we call state parameter<sup>2</sup>, or state variables. At certain times, which may be prescribed in advance, or which may be determined by the process itself, we are called upon to make decisions which will affect the state of the system. These decisions are equivalent to transformations of the state variables, the choice of a decision being identical with the choice of a transformation. The outcome of the preceding decisions is to be used to guide the choice of future ones, with the purpose of the whole process that of maximizing some function of the parameters describing the final state.

Examples of processes fitting this loose description are furnished by virtually every phase of modern life, from the planning of industrial production lines to the scheduling of patients at a medical clinic; from the determination of long-term investment programs for universities to the determination of a replacement policy for machinery in factories; from the programming of training policies for skilled and unskilled labor to the choice of optimal purchasing and inventory policies for department stores and military establishments.

# Dynamic programming applications

---

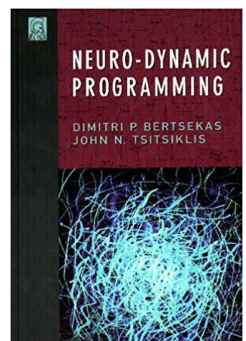
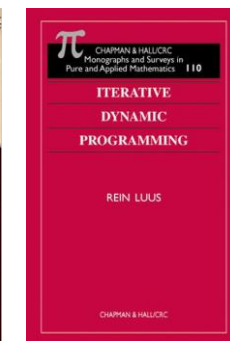
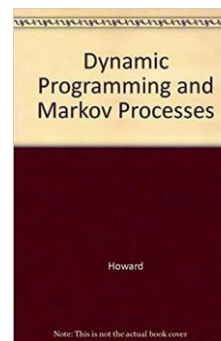
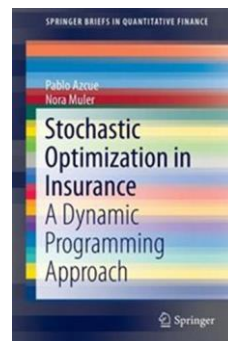
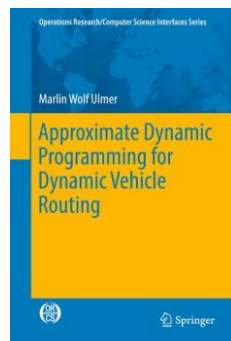
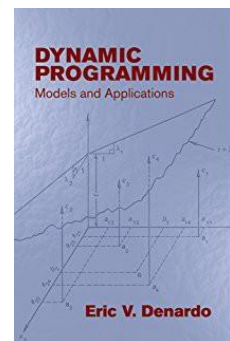
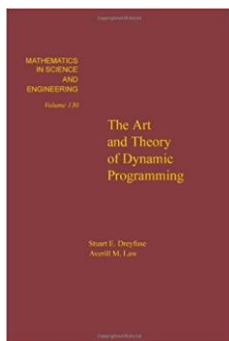
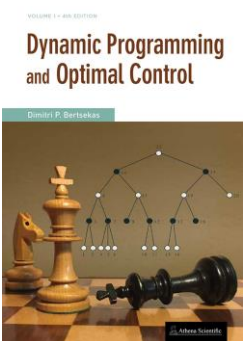
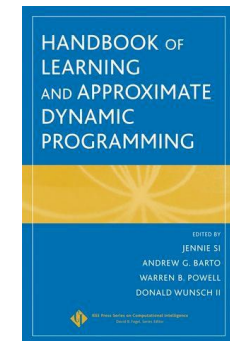
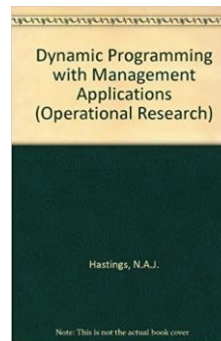
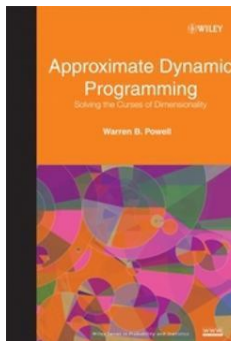
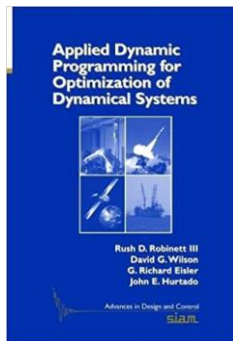
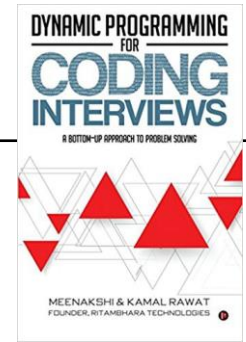
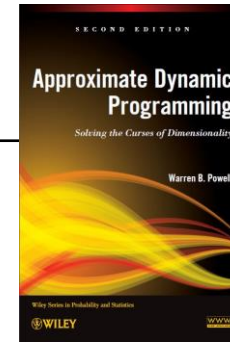
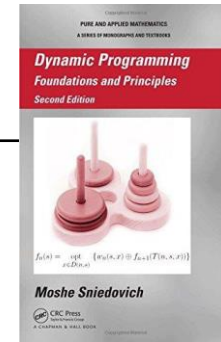
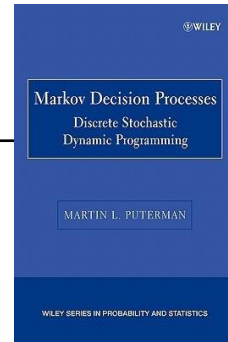
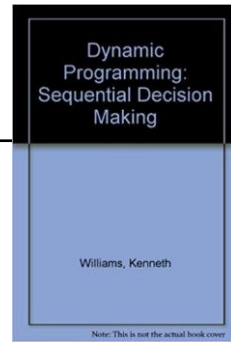
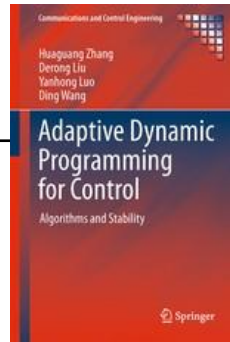
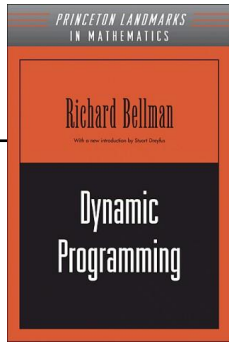
## Application areas.

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

## Some famous dynamic programming algorithms.

- Avidan-Shamir for seam carving.
- Unix diff for comparing two files.
- Viterbi for hidden Markov models.
- De Boor for evaluating spline curves.
- Bellman-Ford-Moore for shortest path.
- Knuth-Plass for word wrapping text in  $T_1X$ .
- Cocke-Kasami-Younger for parsing context-free grammars.
- Needleman-Wunsch/Smith-Waterman for sequence alignment.

# Dynamic programming books



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

- Dynamic Programming

# Algorithmic Paradigms

## Dynamic Programming

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

**Divide-and-conquer.** Break up a problem into 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 History

**Bellman.** [1950s] Pioneered the systematic study of dynamic programming.

## Etymology.

- . Dynamic programming = planning over time.
- . Secretary of Defense was hostile to mathematical research.
- . Bellman sought an impressive name to avoid confrontation.

"it's impossible to use dynamic in a pejorative sense"  
"something not even a Congressman could object to"

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

# Dynamic Programming Applications

## Areas.

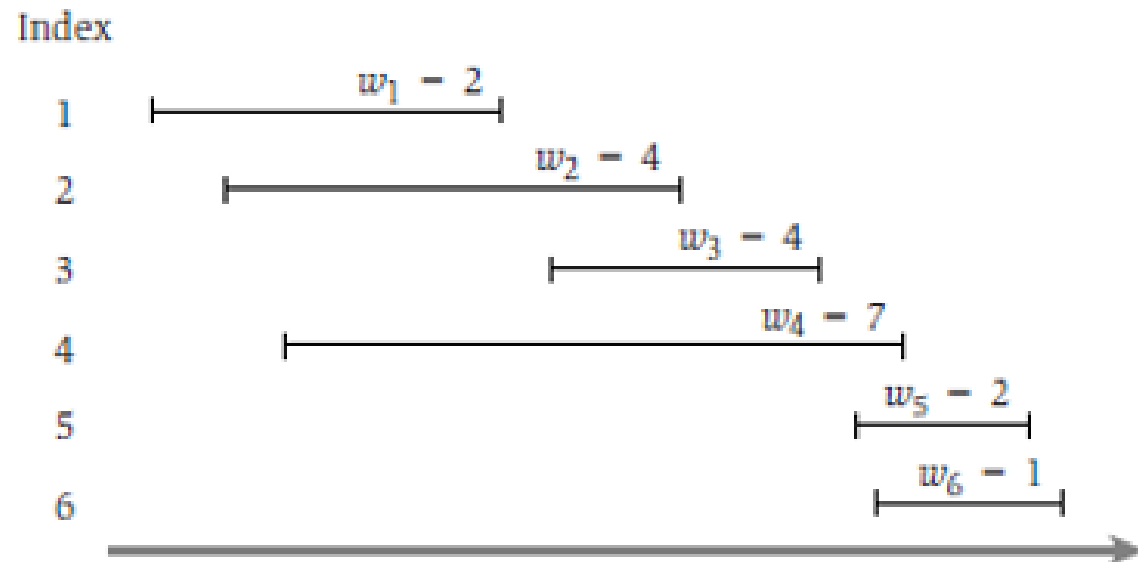
- . Bioinformatics.
- . Control theory.
- . Information theory.
- . Operations research.
- . Computer science: theory, graphics, AI, compilers, systems, ....

## Some famous dynamic programming algorithms.

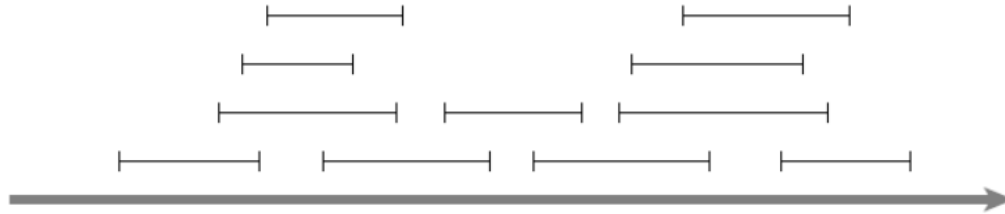
- . Unix diff for comparing two files.
- . Viterbi for hidden Markov models.
- . Smith-Waterman for genetic sequence alignment.
- . Bellman-Ford for shortest path routing in networks.
- . Cocke-Kasami-Younger for parsing context free grammars.



## 6.1 Weighted Interval Scheduling



# When weight is 1 for all the interval “greedy” is an optimal solution

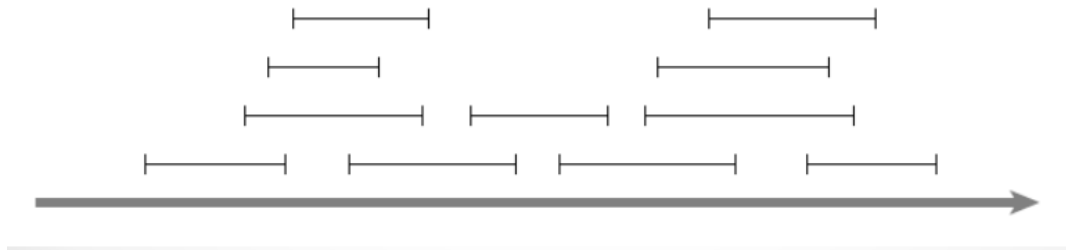


More formally, there will be  $n$  requests labeled  $1, \dots, n$ , with each request  $i$  specifying a start time  $s_i$  and a finish time  $f_i$ . Naturally, we have  $s_i < f_i$  for all  $i$ . Two requests  $i$  and  $j$  are *compatible* if the requested intervals do not overlap: that is, either request  $i$  is for an earlier time interval than request  $j$  ( $f_i \leq s_j$ ), or request  $i$  is for a later time than request  $j$  ( $f_j \leq s_i$ ). We'll say more generally that a subset  $A$  of requests is compatible if all pairs of requests  $i, j \in A$ ,  $i \neq j$  are compatible. The goal is to select a compatible subset of requests of maximum possible size

**Use a simple rule** to select a first request  $i_1$ . Once a request  $i_1$  is accepted, we reject all requests that are not compatible with  $i_1$ . We then select the next request  $i_2$  to be accepted, and again reject all requests that are not compatible with  $i_2$ . We continue in this fashion until we run out of requests.

# Greedy rule

“Earliest finish time” is a greedy rule that produce an optimal solution



# Interval scheduling: earliest-finish-time-first algorithm

EARLIEST-FINISH-TIME-FIRST ( $n, s_1, s_2, \dots, s_n, f_1, f_2, \dots, f_n$ )

**SORT** jobs by finish times and renumber so that  $f_1 \leq f_2 \leq \dots \leq f_n$ .

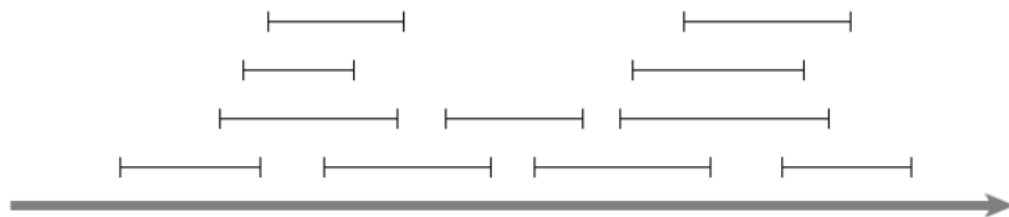
$S \leftarrow \emptyset$ .  $\longleftarrow$  set of jobs selected

**FOR**  $j = 1$  **TO**  $n$

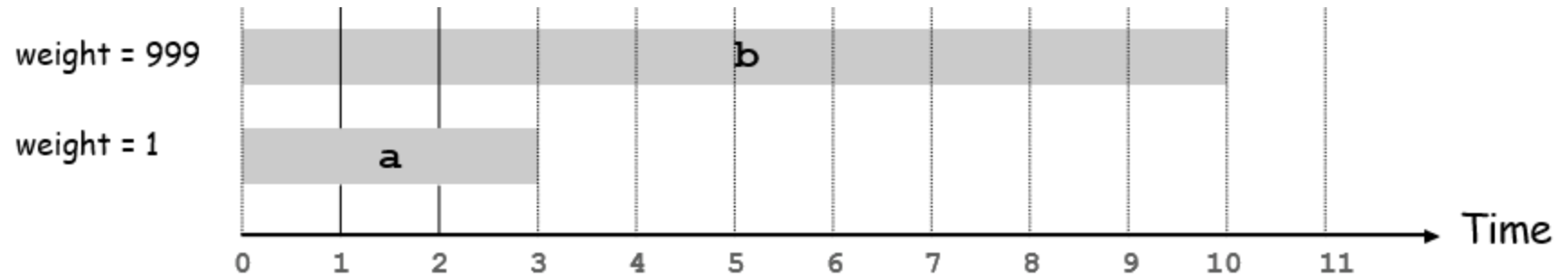
**IF** (job  $j$  is compatible with  $S$ )

$S \leftarrow S \cup \{ j \}$ .

**RETURN**  $S$ .



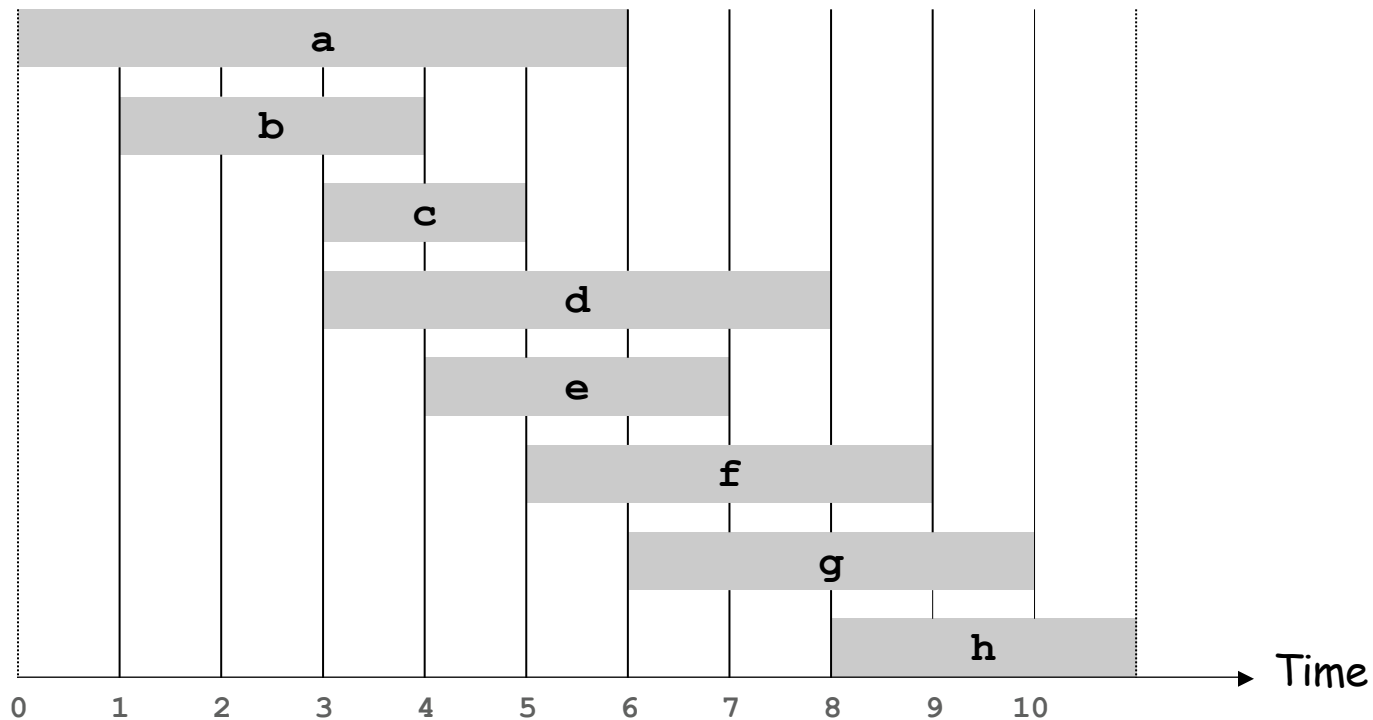
Try to use the same rule for the weighted version



# Weighted Interval Scheduling

## Weighted interval scheduling problem.

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

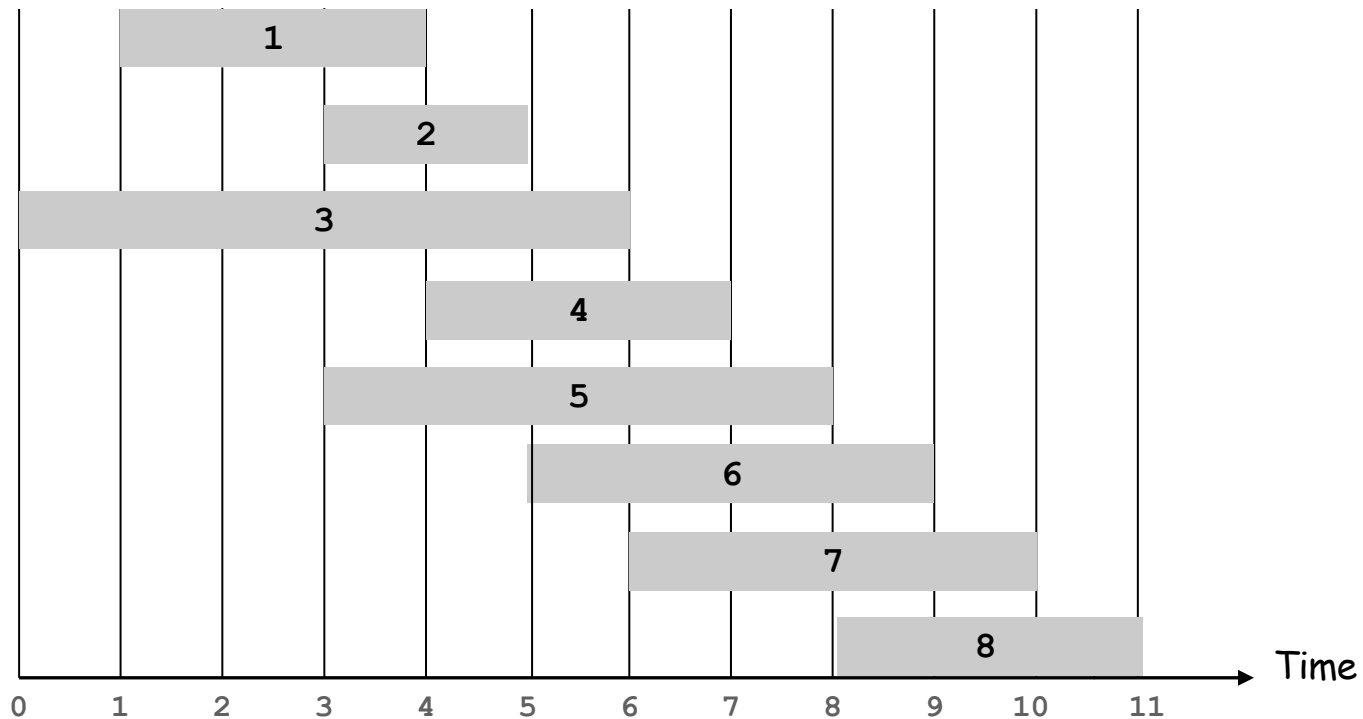


# Weighted Interval Scheduling

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

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

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



# Dynamic Programming: Binary Choice

**Notation.**  $OPT(j)$  = value of optimal solution to the problem consisting of job requests  $1, 2, \dots, j$ .

- Case 1:  $OPT$  selects job  $j$ .
  - collect profit  $v_j$
  - can't use incompatible jobs  $\{ p(j) + 1, p(j) + 2, \dots, j - 1 \}$
  - must include optimal solution to problem consisting of remaining compatible jobs  $1, 2, \dots, p(j)$
- Case 2:  $OPT$  does not select job  $j$ .
  - must include optimal solution to problem consisting of remaining compatible jobs  $1, 2, \dots, j-1$

↖ optimal substructure



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



# Weighted Interval Scheduling: Brute Force

## Brute force algorithm - Recursive

**Input:**  $n, s_1, \dots, s_n, f_1, \dots, f_n, v_1, \dots, v_n$

**Sort** jobs by finish times so that  $f_1 \leq f_2 \leq \dots \leq f_n$ .

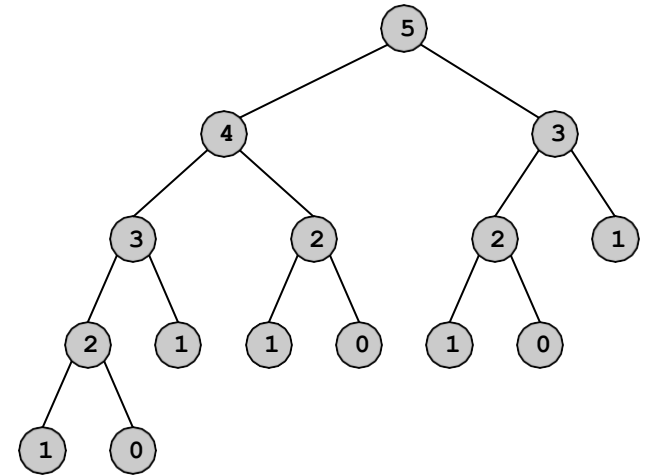
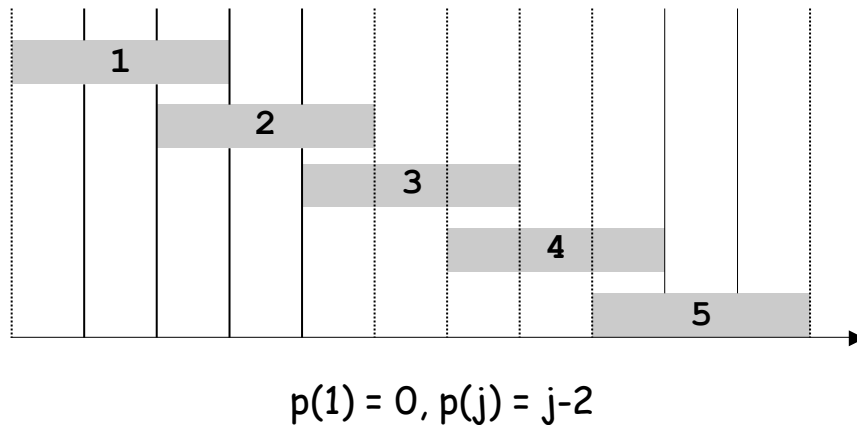
**Compute**  $p(1), p(2), \dots, p(n)$

```
Compute-Opt(j) {  
    if (j = 0)  
        return 0  
    else  
        return max( $v_j + \text{Compute-Opt}(p(j))$ ,  $\text{Compute-Opt}(j-1)$ )  
}
```

# Weighted Interval Scheduling: Brute Force

**Observation.** Recursive algorithm fails spectacularly because of redundant sub-problems  $\Rightarrow$  exponential algorithms.

**Ex.** Number of recursive calls for family of "layered" instances grows like Fibonacci sequence.



```
return max(vj + Compute-Opt(p(j)), Compute-Opt(j-1))
```

# Weighted Interval Scheduling: Bottom-Up

Bottom-up dynamic programming. Unwind recursion.

**Input:**  $n, s_1, \dots, s_n, f_1, \dots, f_n, v_1, \dots, v_n$

**Sort** jobs by finish times so that  $f_1 \leq f_2 \leq \dots \leq f_n$ .

**Compute**  $p(1), p(2), \dots, p(n)$

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

## Fill out the Table

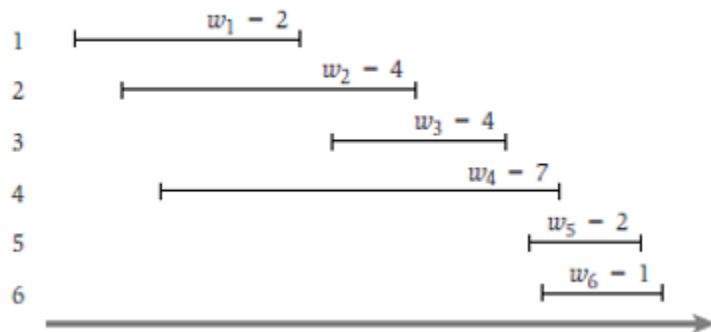
**Input:**  $n, s_1, \dots, s_n, f_1, \dots, f_n, v_1, \dots, v_n$

**Sort** jobs by finish times so that  $f_1 \leq f_2 \leq \dots \leq f_n$ .

**Compute**  $p(1), p(2), \dots, p(n)$

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

Index



$p(1) =$   
 $p(2) =$   
 $p(3) =$   
 $p(4) =$   
 $p(5) =$   
 $p(6) =$   
 $p(7) =$

1							
2							
3							
4							
5							
6							

# Dynamic programming solution

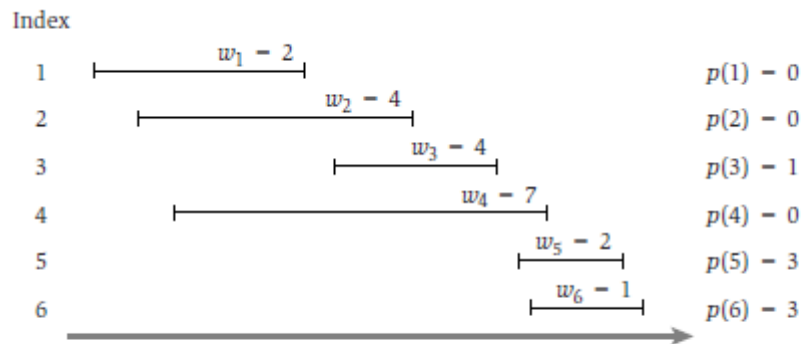
**Input:**  $n, s_1, \dots, s_n, f_1, \dots, f_n, v_1, \dots, v_n$

**Sort** jobs by finish times so that  $f_1 \leq f_2 \leq \dots \leq f_n$ .

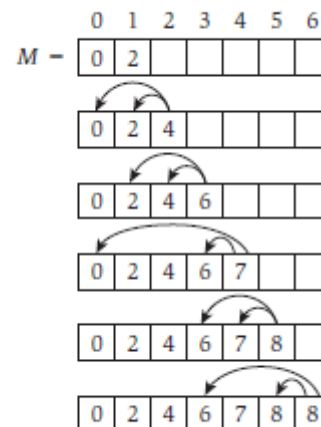
**Compute**  $p(1), p(2), \dots, p(n)$

```

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



(a)



(b)

## Subset Sums problem

### The problem

In the scheduling problem we consider here, we have a single machine that can process jobs, and we have a set of requests  $\{1, 2, \dots, n\}$ . We are only able to use this resource for the period between time 0 and time  $W$ , for some number  $W$ . Each request corresponds to a job that requires time  $w_i$  to process. If our goal is to process jobs so as to keep the machine as busy as possible up to the “cut-off”  $W$ , which jobs should we choose?

### Formally

More formally, we are given  $n$  items  $\{1, \dots, n\}$ , and each has a given nonnegative weight  $w_i$  (for  $i = 1, \dots, n$ ). We are also given a bound  $W$ . We would like to select a subset  $S$  of the items so that  $\sum_{i \in S} w_i \leq W$  and, subject to this restriction,  $\sum_{i \in S} w_i$  is as large as possible. We will call this the *Subset Sum Problem*.

$$\text{OPT}(i, w) = \max_S \sum_{j \in S} w_j,$$

## Example

:  $W = 6$ , items  $w_1 = 2, w_2 = 2, w_3 = 3$

$\text{OPT}(n, W) =$

$\text{OPT}(3, 6) =$

## Subset sums Recurrence

- $\text{OPT}(i, w) =$
- If  $n \notin \mathcal{O}$ , then  $\text{OPT}(n, W) = \text{OPT}(n - 1, W)$ , since we can simply ignore item  $n$ .
  - If  $n \in \mathcal{O}$ , then  $\text{OPT}(n, W) = w_n + \text{OPT}(n - 1, W - w_n)$ , since we now seek to use the remaining capacity of  $W - w_n$  in an optimal way across items  $1, 2, \dots, n - 1$ .

**(6.8)** *If  $w < w_i$  then  $\text{OPT}(i, w) = \text{OPT}(i - 1, w)$ . Otherwise*

$$\text{OPT}(i, w) = \max(\text{OPT}(i - 1, w), w_i + \text{OPT}(i - 1, w - w_i)).$$

---

Subset-Sum( $n, W$ )

  Array  $M[0 \dots n, 0 \dots W]$

  Initialize  $M[0, w] = 0$  for each  $w = 0, 1, \dots, W$

  For  $i = 1, 2, \dots, n$

    For  $w = 0, \dots, W$

      Use the recurrence (6.8) to compute  $M[i, w]$

    Endfor

  Endfor

  Return  $M[n, W]$

---



**(6.8)** If  $w < w_i$  then  $\text{OPT}(i, w) = \text{OPT}(i - 1, w)$ . Otherwise  

$$\text{OPT}(i, w) = \max(\text{OPT}(i - 1, w), w_i + \text{OPT}(i - 1, w - w_i)).$$

Subset-Sum( $n, W$ )

Array  $M[0 \dots n, 0 \dots W]$

Initialize  $M[0, w] = 0$  for each  $w = 0, 1, \dots, W$

For  $i = 1, 2, \dots, n$

For  $w = 0, \dots, W$

Use the recurrence (6.8) to compute  $M[i, w]$

Endfor

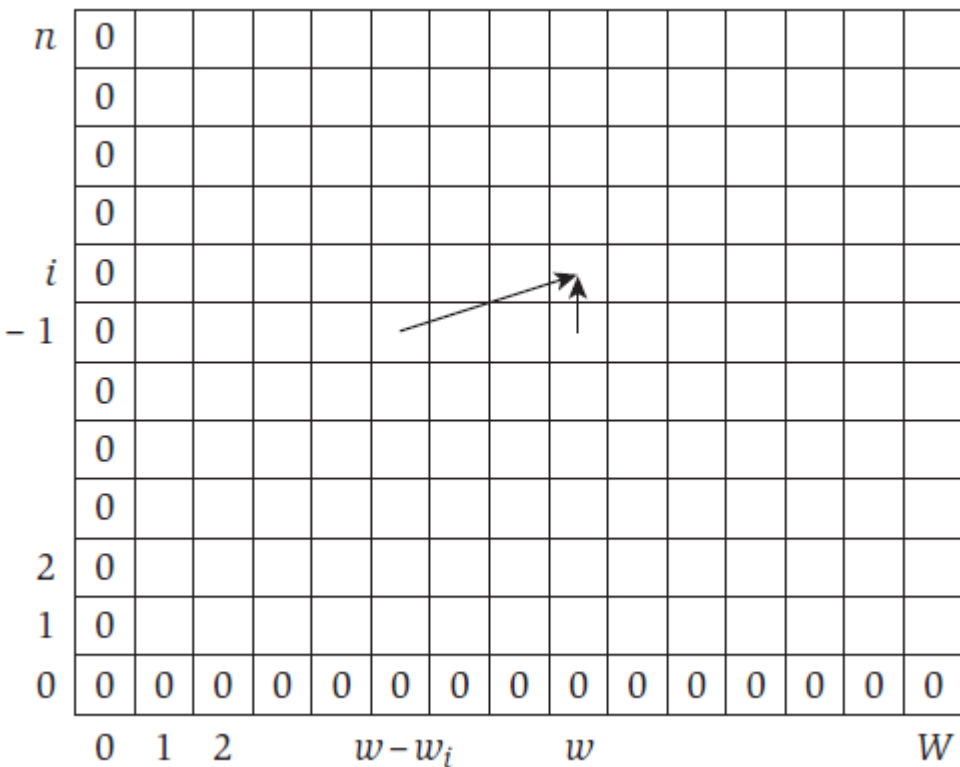
Endfor

Return  $M[n, W]$

$W = 6$ , items  $w_1 = 2, w_2 = 2, w_3 = 3$

$\text{OPT}(n, W) =$

$\text{OPT}(3, 6) =$



# Knapsack problem

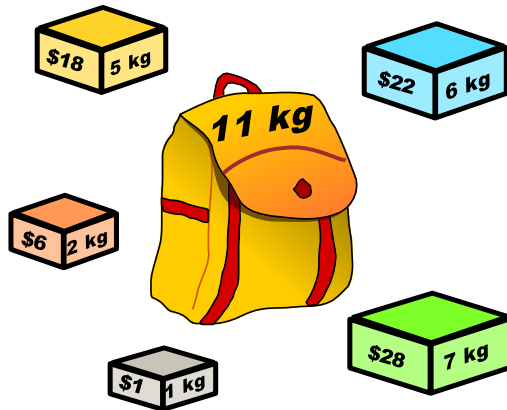
**Goal.** Pack knapsack so as to maximize total value of items taken.

- There are  $n$  items: item  $i$  provides value  $v_i > 0$  and weighs  $w_i > 0$ .
- Value of a subset of items = sum of values of individual items.
- Knapsack has weight limit of  $W$ .

**Ex.** The subset  $\{ 1, 2, 5 \}$  has value \$35 (and weight 10).

**Ex.** The subset  $\{ 3, 4 \}$  has value \$40 (and weight 11).

**Assumption.** All values and weights are integral.



Creative Commons Attribution-Share Alike 2.5  
by Dake

$i$	$v_i$	$w_i$
1	\$1	1 kg
2	\$6	2 kg
3	\$18	5 kg
4	\$22	6 kg
5	\$28	7 kg

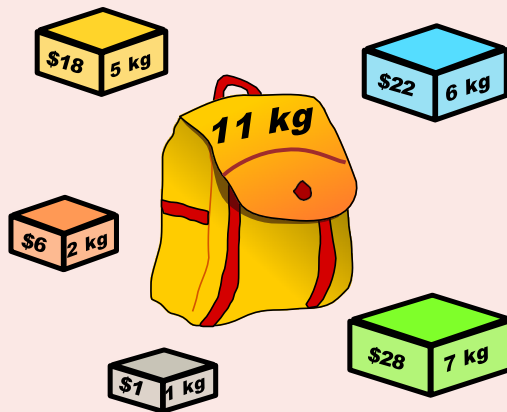
weights and values  
can be arbitrary  
positive integers

**knapsack instance**  
**(weight limit  $W = 11$ )**



## Which algorithm solves knapsack problem?

- A. Greedy-by-value: repeatedly add item with maximum  $v_i$ .
- B. Greedy-by-weight: repeatedly add item with minimum  $w_i$ .
- C. Greedy-by-ratio: repeatedly add item with maximum ratio  $v_i / w_i$ .
- D. None of the above.



Creative Commons Attribution-Share Alike 2.5  
by Dake

$i$	$v_i$	$w_i$
1	\$1	1 kg
2	\$6	2 kg
3	\$18	5 kg
4	\$22	6 kg
5	\$28	7 kg

**knapsack instance**  
**(weight limit  $W = 11$ )**

# Dynamic programming: two variables

---

**Def.**  $OPT(i, w)$  = optimal value of knapsack problem with items  $1, \dots, i$ , subject to weight limit  $w$ .

**Goal.**  $OPT(n, W)$ .

**Case 1.**  $OPT(i, w)$  does not select item  $i$ .

← possibly because  $w_i > w$

- $OPT(i, w)$  selects best of  $\{ 1, 2, \dots, i - 1 \}$  subject to weight limit  $w$ .

**Case 2.**  $OPT(i, w)$  selects item  $i$ .

↖ ↙ optimal substructure property  
(proof via exchange argument)

- Collect value  $v_i$ .
- New weight limit =  $w - w_i$ .
- $OPT(i, w)$  selects best of  $\{ 1, 2, \dots, i - 1 \}$  subject to new weight limit.

28

**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: bottom-up dynamic programming

---

**KNAPSACK**( $n, W, w_1, \dots, w_n, v_1, \dots, v_n$ )

---

**FOR**  $w = 0$  **TO**  $W$

$M[0, w] \leftarrow 0.$

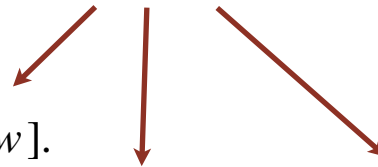
**FOR**  $i = 1$  **TO**  $n$

**FOR**  $w = 0$  **TO**  $W$

**IF** ( $w_i > w$ )  $M[i, w] \leftarrow M[i-1, w].$

**ELSE**  $M[i, w] \leftarrow \max \{ M[i-1, w], v_i + M[i-1, w - w_i] \}.$

previously computed values



**RETURN**  $M[n, W].$

---

29

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

# Knapsack problem: bottom-up dynamic programming demo

$i$	$v_i$	$w_i$
1	\$1	1 kg
2	\$6	2 kg
3	\$18	5 kg
4	\$22	6 kg
5	\$28	7 kg

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

		weight limit w											
		0	1	2	3	4	5	6	7	8	9	10	11
subset of items 1, ..., i	{ }												
	{1 }												
	{1, 2 }												
	{1, 2, 3 }												
	{1, 2, 3, 4 }												
	{1, 2, 3, 4, 5 }												

$OPT(i, w)$  = optimal value of knapsack problem with items 1, ..., i, subject to weight limit w

# Knapsack problem: running time

---

**Theorem.** The DP algorithm solves the knapsack problem with  $n$  items and maximum weight  $W$  in  $\Theta(n W)$  time and  $\Theta(n W)$  space.

**Pf.**

- Takes  $O(1)$  time per table entry.
- There are  $\Theta(n W)$  table entries.
- After computing optimal values, can trace back to find solution:  
 $OPT(i, w)$  takes item  $i$  iff  $M[i, w] > M[i - 1, w]$ . ▀

← weights are integers  
between 1 and  $W$

**Remarks.**

- Algorithm depends critically on assumption that weights are integral.
- Assumption that values are integral was not used.





## Subset Sums problem: Recursion

$W = 6$ , items  $w_1 = 2$ ,  $w_2 = 2$ ,  $w_3 = 3$

3							
2							
1							
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

Initial values

3							
2							
①	0	0	2	2	2	2	2
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

Filling in values for  $i = 1$

3							
②	0	0	2	2	4	4	4
1	0	0	2	2	2	2	2
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

Filling in values for  $i = 2$

3							
2	0	0	2	3	4	5	5
1	0	0	2	2	4	4	4
0	0	0	2	2	2	2	2
	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

Filling in values for  $i = 3$

# Knapsack problem: bottom-up dynamic programming demo

$i$	$v_i$	$w_i$
1	\$1	1 kg
2	\$6	2 kg
3	\$18	5 kg
4	\$22	6 kg
5	\$28	7 kg

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

		weight limit $w$											
		0	1	2	3	4	5	6	7	8	9	10	11
subset of items $1, \dots, i$	{ }	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	35	40

$OPT(i, w)$  = optimal value of knapsack problem with items 1, ..., i, subject to weight limit  $w$