

# Chapter 6

## Dynamic Programming



Slides by Kevin Wayne.  
Copyright © 2005 Pearson-Addison Wesley.  
All rights reserved.

## Algorithmic Paradigms

**Greed.** 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 History

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

## 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, 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.

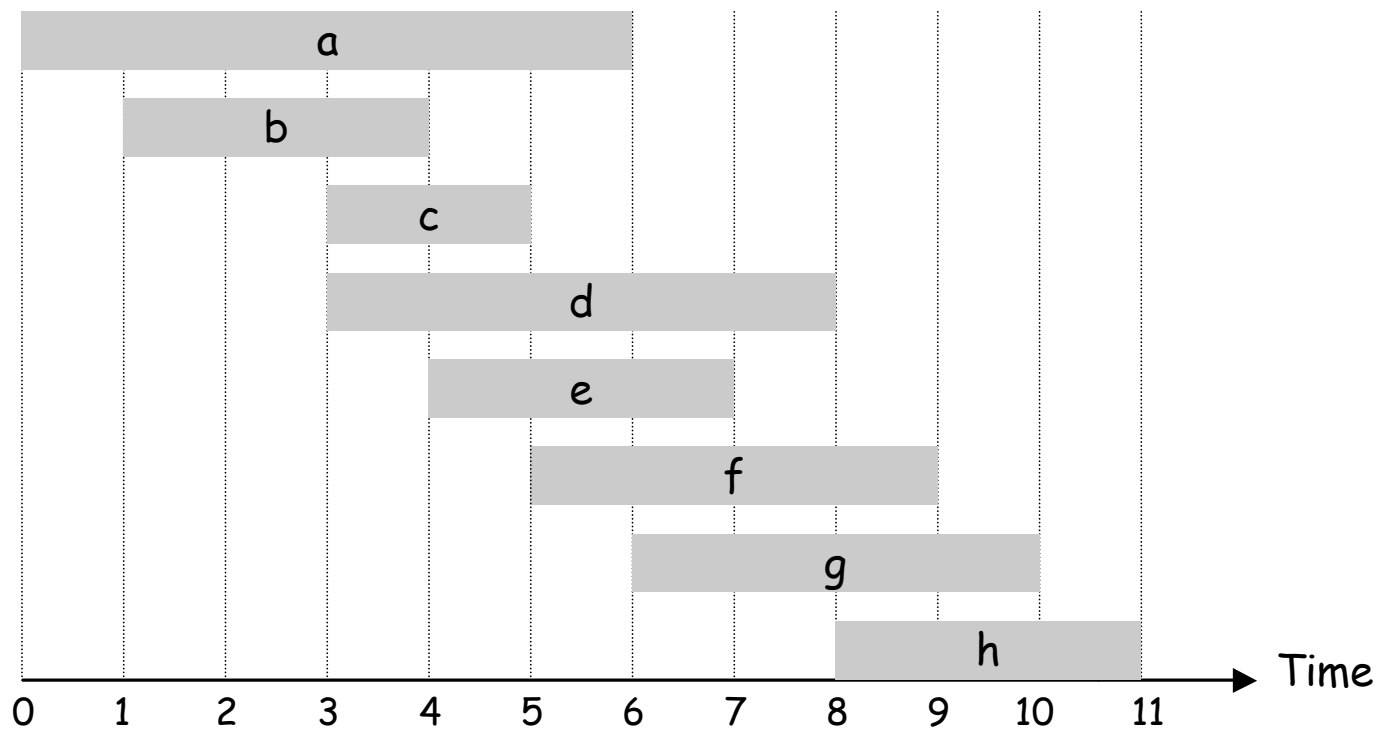
## 6.1 Weighted Interval Scheduling

---

# 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.

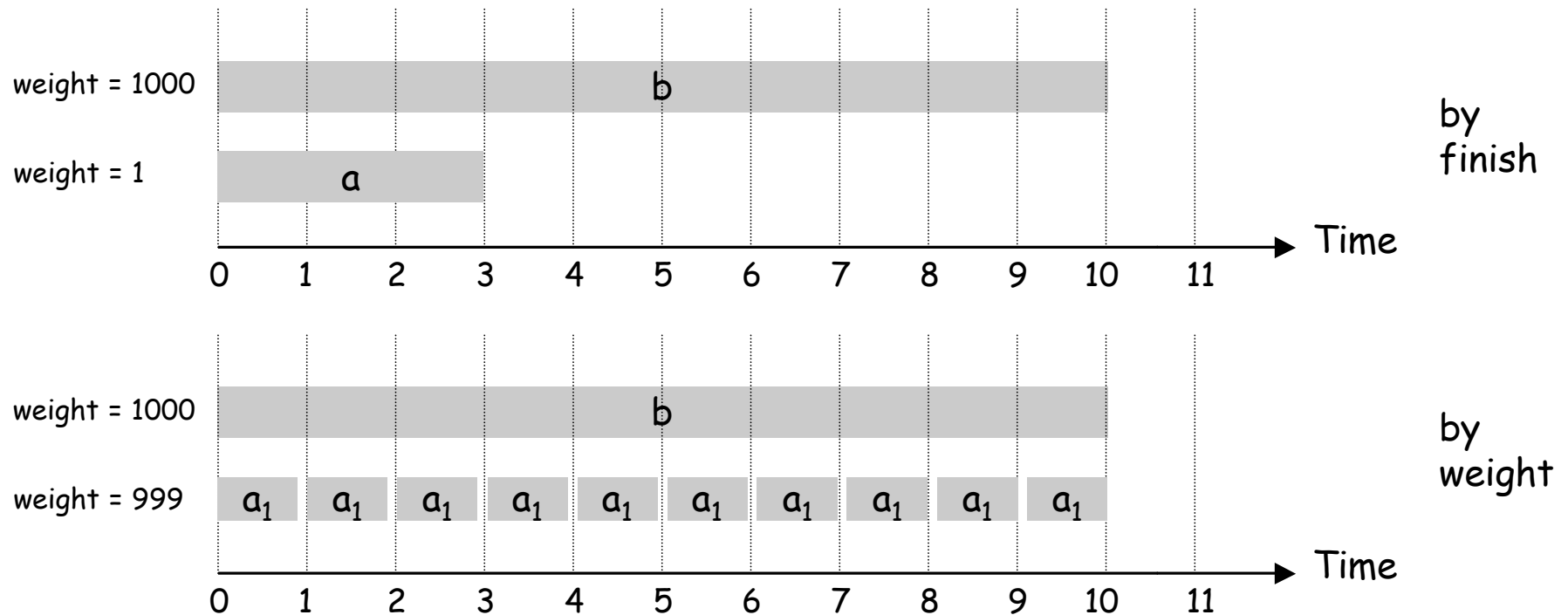


## Unweighted Interval Scheduling Review

**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 spectacularly if arbitrary weights are allowed.

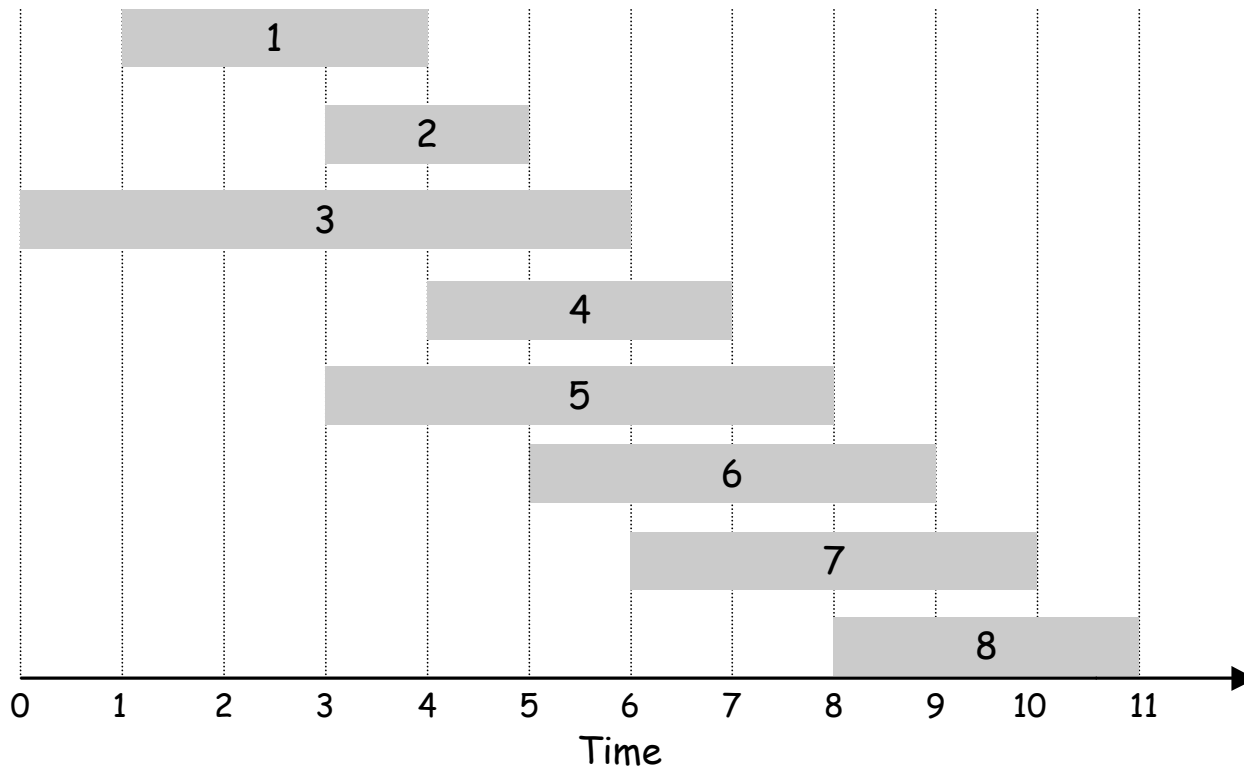


# Weighted Interval Scheduling

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



j	p(j)
0	-
1	0
2	0
3	0
4	1
5	0
6	2
7	3
8	5



## 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  $j$ .
  - 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.

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

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

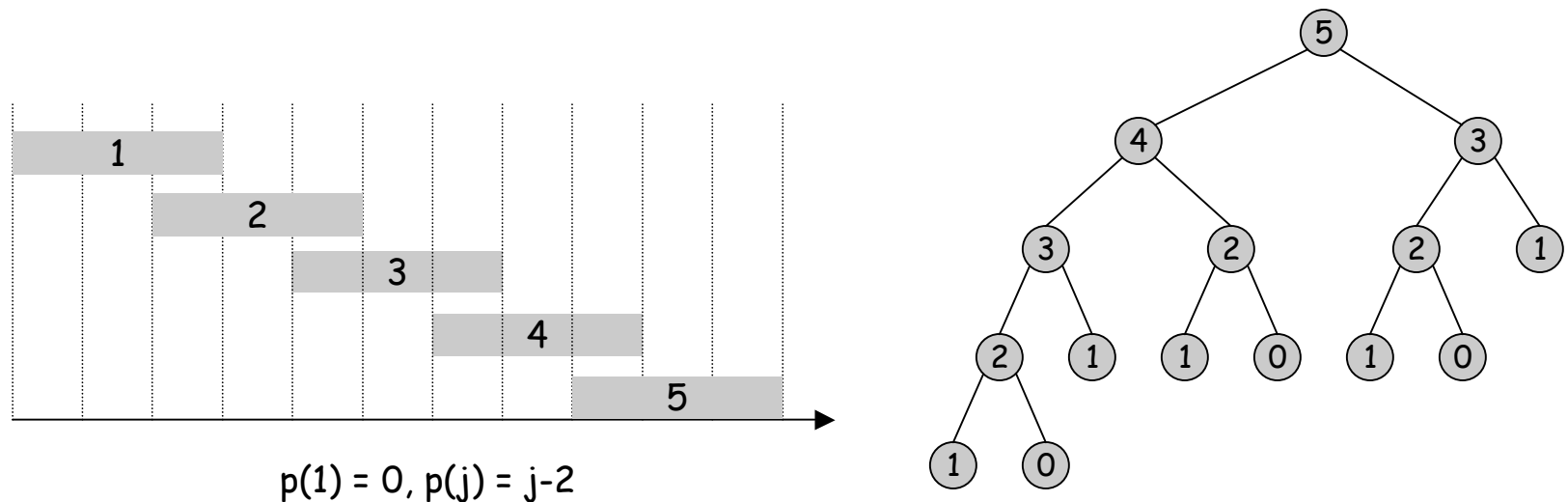
```
Compute  $p(1), p(2), \dots, p(n)$ 
```

```
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.



## Weighted Interval Scheduling: Memoization

**Memoization.** Store results of each sub-problem in a cache; lookup as needed.

```
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  $p(1), p(2), \dots, p(n)$ 
```

```
for  $j = 1$  to  $n$ 
```

```
     $M[j] = \text{empty}$  ← global array
```

```
 $M[0] = 0$ 
```

```
M-Compute-Opt( $j$ ) {
```

```
    if ( $M[j]$  is empty)
```

```
         $M[j] = \max(w_j + \text{M-Compute-Opt}(p(j)), \text{M-Compute-Opt}(j-1))$ 
```

```
    return  $M[j]$ 
```

```
}
```

## 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  $p(\cdot)$ :  $O(n)$  after sorting by start time.
- $M\text{-Compute-Opt}(j)$ : each invocation takes  $O(1)$  time and either
  - (i) returns an existing value  $M[j]$
  - (ii) fills in one new entry  $M[j]$  and makes two recursive calls
- Progress measure  $\Phi = \#$  nonempty entries of  $M[\ ]$ .
  - initially  $\Phi = 0$ , throughout  $\Phi \leq n$ .
  - (ii) increases  $\Phi$  by 1  $\Rightarrow$  at most  $2n$  recursive calls.
- Overall running time of  $M\text{-Compute-Opt}(n)$  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.

```
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  $p(1), p(2), \dots, p(n)$ 
```

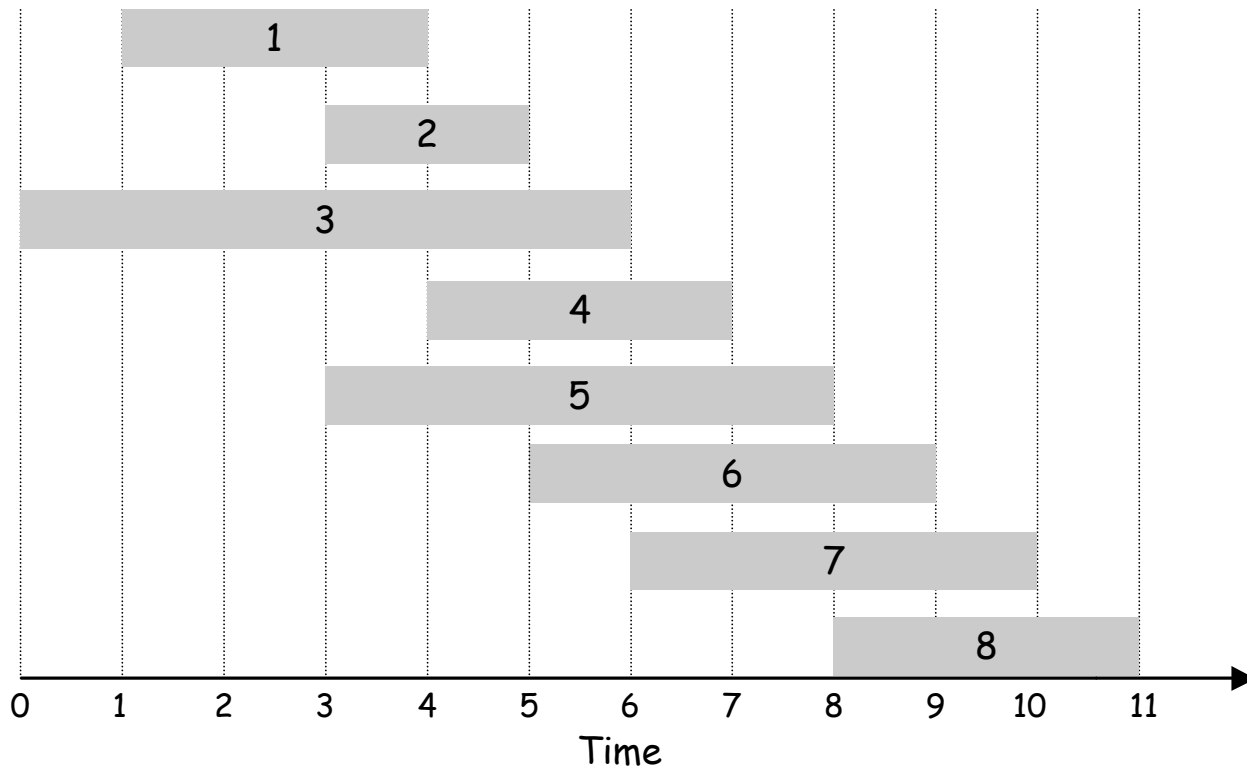
```
Iterative-Compute-Opt {  
    M[0] = 0  
    for j = 1 to n  
        M[j] = max( $v_j + M[p(j)]$ , M[j-1])  
}
```

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



j	v <sub>j</sub>	p <sub>j</sub>	opt <sub>j</sub>
0		-	
1		0	
2		0	
3		0	
4		1	
5		0	
6		2	
7		3	
8		5	

## Weighted Interval Scheduling: Finding a Solution

Q. Dynamic programming algorithms computes optimal value. What if we want the solution itself?

A. Do some post-processing.

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

Find-Solution(j) {
    if (j = 0)
        output nothing
    else if ( $v_j + M[p(j)] > M[j-1]$ )
        print j
        Find-Solution(p(j))
    else
        Find-Solution(j-1)
}
```

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



## 6.4 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 so as to maximize total value.

Ex: { 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  $v_i / w_i$ .

Ex: { 5, 2, 1 } achieves only value = 35  $\Rightarrow$  greedy not optimal.

## Dynamic Programming: False Start

**Def.**  $OPT(i)$  = max profit subset of items  $1, \dots, i$ .

- Case 1:  $OPT$  does not select item  $i$ .
  - $OPT$  selects best of  $\{ 1, 2, \dots, i-1 \}$
- 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 more sub-problems!

## Dynamic Programming: Adding a New Variable

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

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

## Knapsack Problem: Bottom-Up

Knapsack. Fill up an  $n$ -by- $W$  array.

```
Input:  $n, w_1, \dots, w_N, v_1, \dots, v_N$ 

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

for  $i = 1$  to  $n$ 
    for  $w = 1$  to  $W$ 
        if ( $w_i > w$ )
             $M[i, w] = M[i-1, w]$ 
        else
             $M[i, w] = \max \{M[i-1, w], v_i + M[i-1, w-w_i]\}$ 

return  $M[n, W]$ 
```

# Knapsack Algorithm

←————— W + 1 —————→

		0	1	2	3	4	5	6	7	8	9	10	11
n + 1 ↓	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, 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: { 4, 3 }  
value = 22 + 18 = 40

W = 11

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

```

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 ]}
    
```

## Knapsack Problem: Running Time

Running time.  $\Theta(nW)$ .

- Not polynomial in input size!
- "Pseudo-polynomial."
- Decision version of Knapsack is NP-complete. [Chapter 8]

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

# String Similarity

How similar are two strings?

- **ocurrance**
- **occurrence**

o	c	u	r	r	a	n	c	e	-
o	c	c	u	r	r	e	n	c	e

5 mismatches, 1 gap

o	c	-	u	r	r	a	n	c	e
o	c	c	u	r	r	e	n	c	e

1 mismatch, 1 gap

o	c	-	u	r	r	-	a	n	c	e
o	c	c	u	r	r	e	-	n	c	e

0 mismatches, 3 gaps



# Edit Distance

## Applications.

- Basis for Unix diff.
- Speech recognition.
- Computational biology.

Edit distance. [Levenshtein 1966, Needleman-Wunsch 1970]

- Gap penalty  $\delta$ ; mismatch penalty  $\alpha_{pq}$ .
- Cost = sum of gap and mismatch penalties.

C T G A C C T A C C T

C C T G A C T A C A T

$$\alpha_{TC} + \alpha_{GT} + \alpha_{AG} + 2\alpha_{CA}$$

- C T G A C C T A C C T

C C T G A C - T A C A T

$$2\delta + \alpha_{CA}$$



## Sequence Alignment: Problem Structure

**Def.**  $OPT(i, j)$  = min cost of aligning strings  $x_1 x_2 \dots x_i$  and  $y_1 y_2 \dots y_j$ .

- Case 1:  $OPT$  matches  $x_i$ - $y_j$ .
  - pay mismatch for  $x_i$ - $y_j$  + min cost of aligning two strings  $x_1 x_2 \dots x_{i-1}$  and  $y_1 y_2 \dots y_{j-1}$
- Case 2a:  $OPT$  leaves  $x_i$  unmatched.
  - pay gap for  $x_i$  and min cost of aligning  $x_1 x_2 \dots x_{i-1}$  and  $y_1 y_2 \dots y_j$
- Case 2b:  $OPT$  leaves  $y_j$  unmatched.
  - pay gap for  $y_j$  and min cost of aligning  $x_1 x_2 \dots x_i$  and  $y_1 y_2 \dots y_{j-1}$

$$OPT(i, j) = \begin{cases} j\delta & \text{if } i = 0 \\ \min \begin{cases} \alpha_{x_i y_j} + OPT(i-1, j-1) \\ \delta + OPT(i-1, j) \\ \delta + OPT(i, j-1) \end{cases} & \text{otherwise} \\ i\delta & \text{if } j = 0 \end{cases}$$

## Sequence Alignment: Algorithm

```
Sequence-Alignment(m, n,  $x_1x_2\dots x_m$ ,  $y_1y_2\dots y_n$ ,  $\delta$ ,  $\alpha$ ) {  
  for i = 0 to m  
    M[0, i] =  $i\delta$   
  for j = 0 to n  
    M[j, 0] =  $j\delta$   
  
  for i = 1 to m  
    for j = 1 to n  
      M[i, j] = min( $\alpha[x_i, y_j] + M[i-1, j-1]$ ,  
                    $\delta + M[i-1, j]$ ,  
                    $\delta + M[i, j-1]$ )  
  
  return M[m, n]  
}
```

**Analysis.**  $\Theta(mn)$  time and space.

English words or sentences:  $m, n \leq 10$ .

Computational biology:  $m = n = 100,000$ . 10 billions ops OK, but 10GB array?