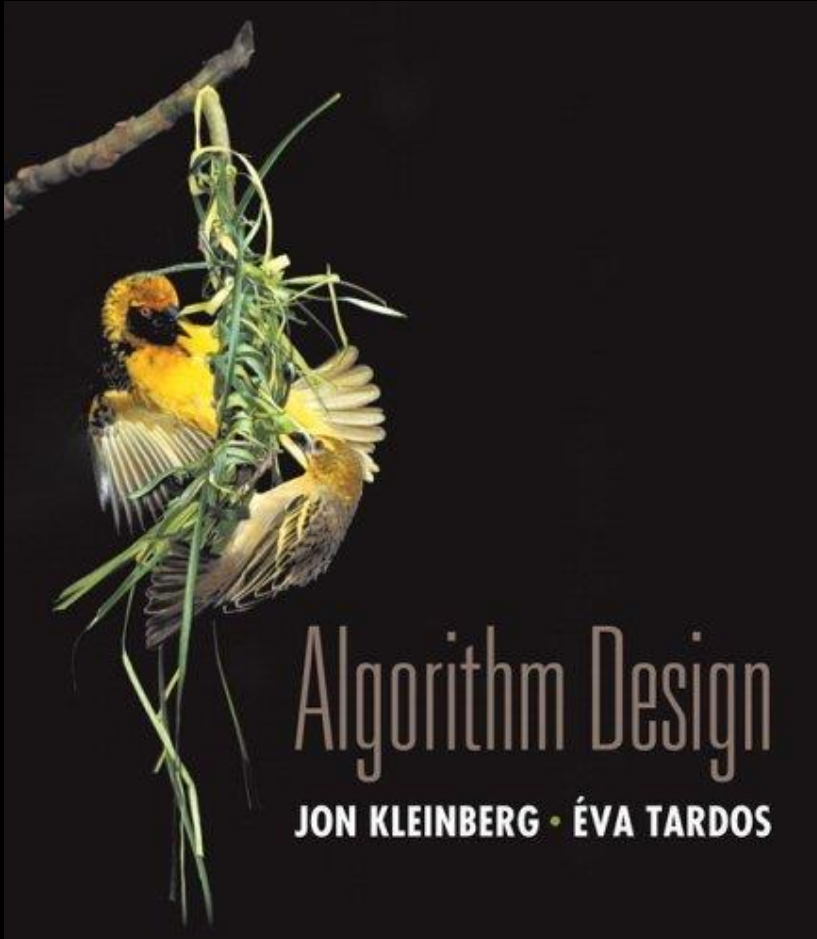


Chapter 6

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 sub-problems with repetitions, and build up solutions to larger and larger sub-problems.

Optimal substructure

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.

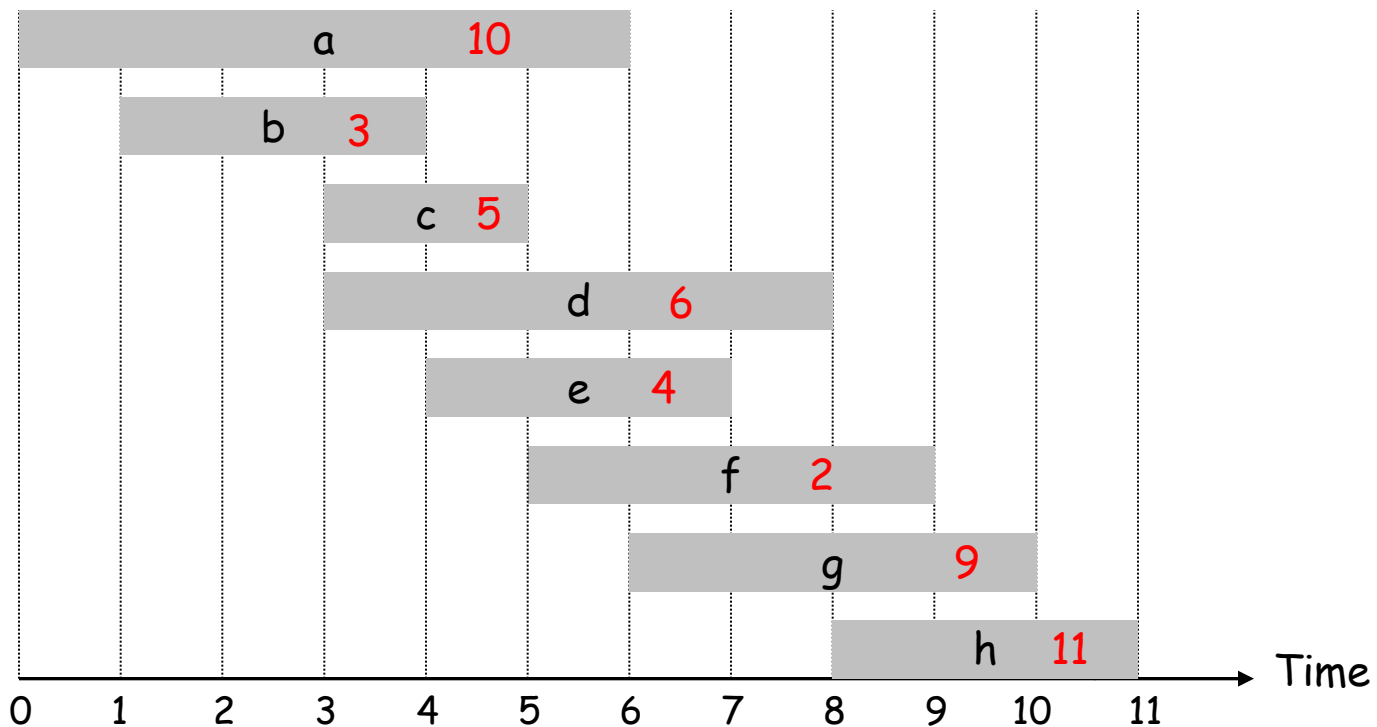
- Viterbi for hidden Markov models.
- Unix diff for comparing two files.
- DNA sequence comparison.
- Bellman-Ford for shortest path routing in networks.
- Cocke-Kasami-Younger for parsing context free grammars.

6.1 Weighted Interval Scheduling

Weighted Interval Scheduling

Weighted interval scheduling problem.

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

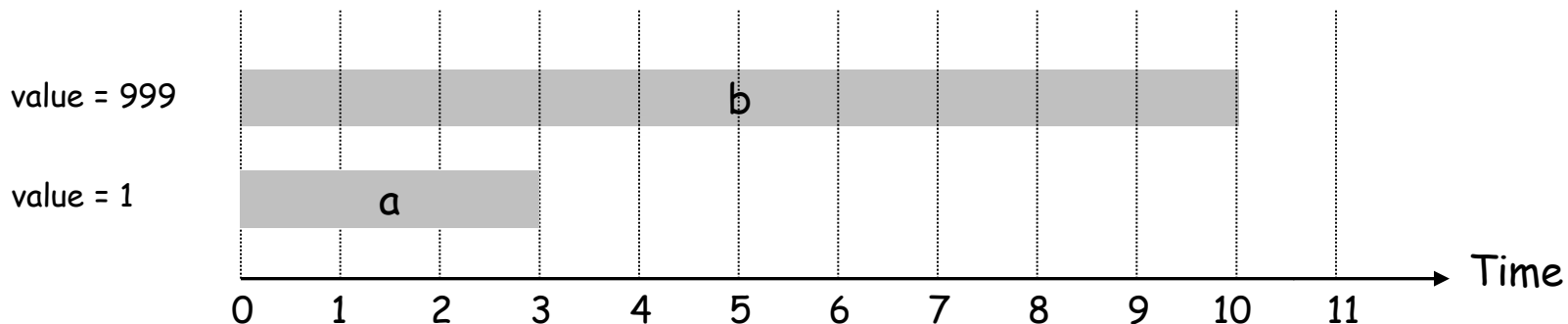


Unweighted Interval Scheduling Review

Recall. Greedy algorithm works if all values are 1.

- Consider jobs in ascending order of finish time.
- Add job to solution if it is compatible with previously chosen jobs.

Observation. Greedy algorithm can fail spectacularly if arbitrary values are allowed.



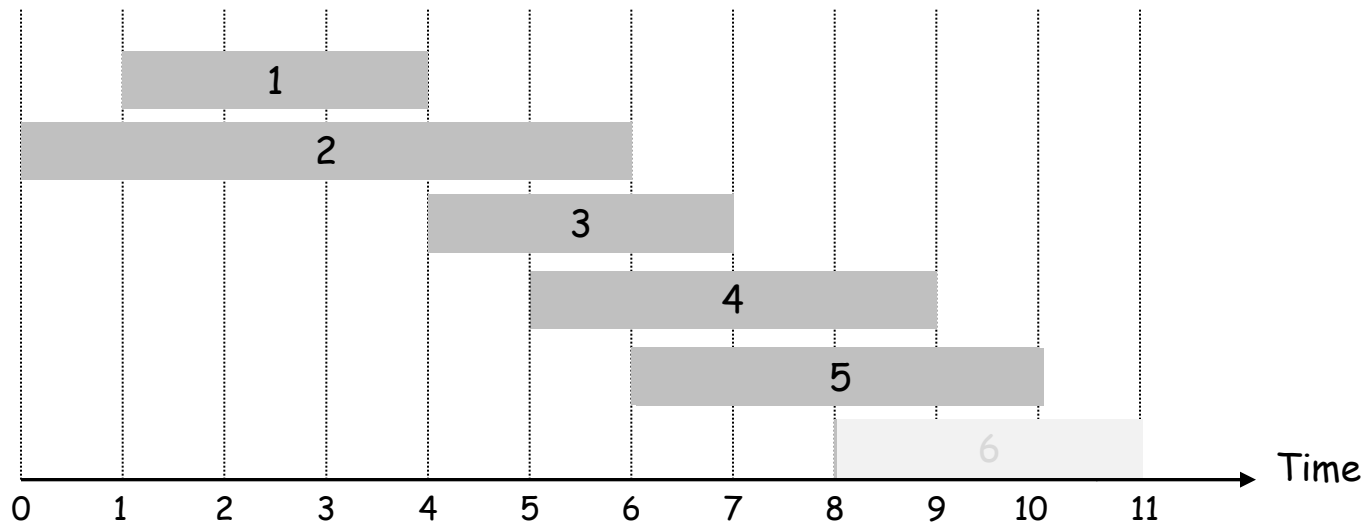
Weighted Interval Scheduling

Dynamic Programming solution:

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

Suppose we want to find optimal solution involving just jobs **1,2,...,5**

Need to decide whether to **include job 5** or to **not include job 5**



Weighted Interval Scheduling

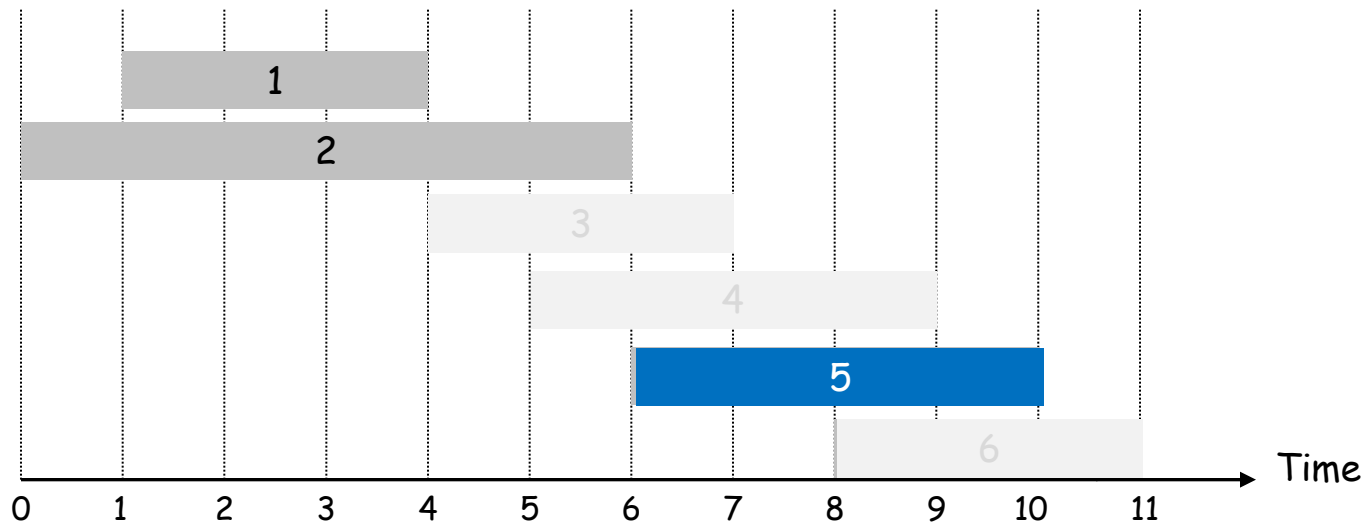
Dynamic Programming solution:

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

Suppose we want to find optimal solution involving just jobs **1,2,...,5**

Need to decide whether to **include job 5** or to **not include job 5**

1. **If include** job 5 => also select optimally among jobs **1,2**



Weighted Interval Scheduling

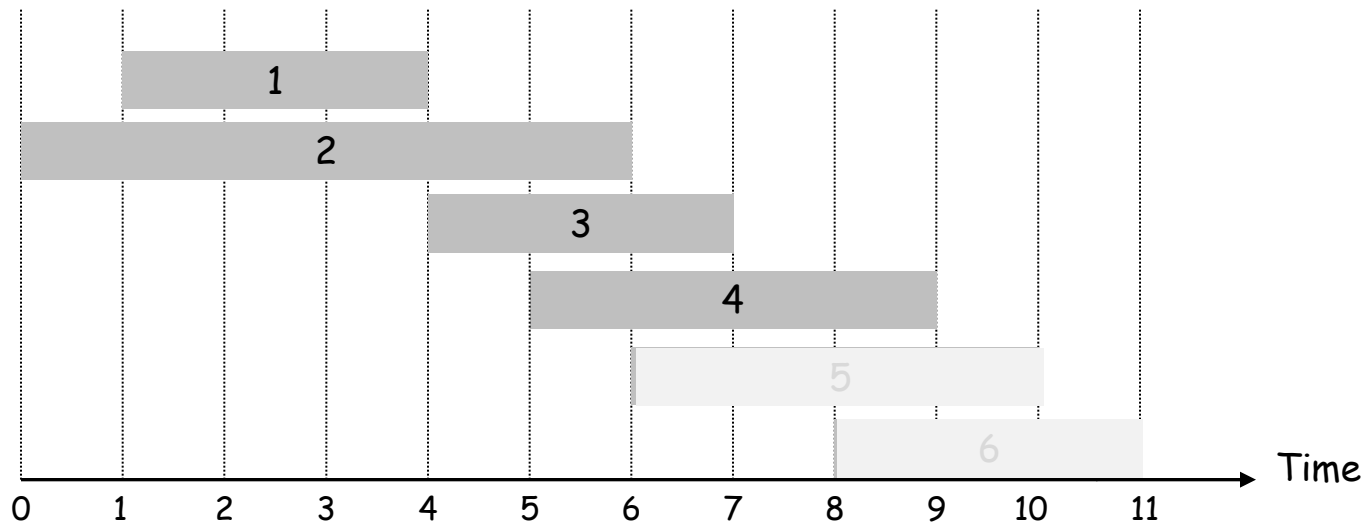
Dynamic Programming solution:

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

Suppose we want to find optimal solution involving just jobs **1,2,...,5**

Need to decide whether to **include job 5** or to **not include job 5**

1. **If include** job 5 \Rightarrow also select optimally among jobs **1,2**
2. **If do not include** job 5 \Rightarrow select optimally among jobs **1,...,4**.

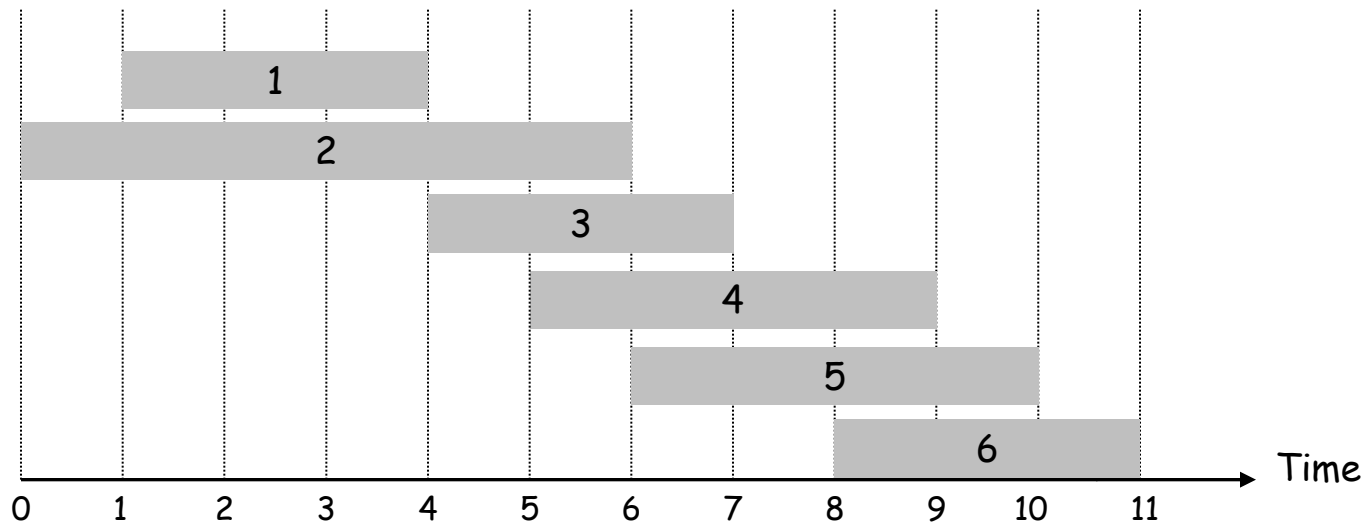


Weighted Interval Scheduling

More generally, define

$\text{lastCompat}(j) = \text{largest index } i < j \text{ such that job } i \text{ is compatible with } j.$

Ex: $\text{lastCompat}(5) = 2$, $\text{lastCompat}(4) = 1$, $\text{lastCompat}(1) = 0$



Weighted Interval Scheduling

More generally, define

$\text{lastCompat}(j)$ = largest index $i < j$ such that job i is compatible with j .

$\text{OPT}(j)$ = value of optimal solution to the problem consisting of jobs $1, 2, \dots, j$.

To compute $\text{OPT}(j)$ we have two options:

- Case 1: Solution **includes** job j .
 - can't use incompatible jobs $\{\text{lastCompat}(j) + 1, \dots, j - 1\}$
 - must include optimal solution to problem consisting of remaining compatible jobs **$1, 2, \dots, \text{lastCompat}(j)$** (=OPT(lastCompat(j)))
- Case 2: Solution does **not include** job j .
 - must include optimal solution to problem consisting of remaining compatible jobs **$1, 2, \dots, j-1$** (=OPT(j-1))

Pick the best option

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

Weighted Interval Scheduling

More generally, define

$\text{lastCompat}(j)$ = largest index $i < j$ such that job i is compatible with j .

$\text{OPT}(j)$ = value of optimal solution to the problem consisting of jobs $1, 2, \dots, j$.

Optimal substructure

To compute $\text{OPT}(j)$ we have

- Case 1: Solution **includes** job j .
 - can't use incompatible jobs $\{\text{lastCompat}(j) + 1, \dots, j - 1\}$
 - must include optimal solution to problem consisting of remaining compatible jobs $1, 2, \dots, \text{lastCompat}(j)$ ($=\text{OPT}(\text{lastCompat}(j))$)
- Case 2: Solution does **not include** job j .
 - must include optimal solution to problem consisting of remaining compatible jobs $1, 2, \dots, j-1$ ($=\text{OPT}(j-1)$)

Pick the best option

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

Weighted Interval Scheduling: Brute Force

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

We can use this expression to compute the optimal value $OPT(n)$

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 $\text{lastCompat}(1), \text{lastCompat}(2), \dots, \text{lastCompat}(n)$

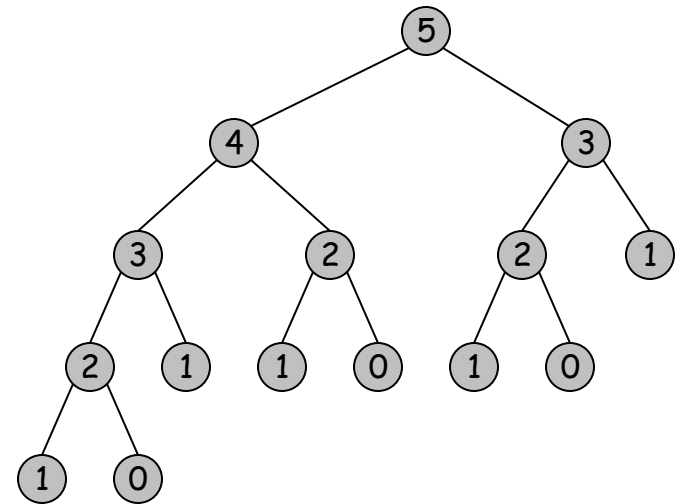
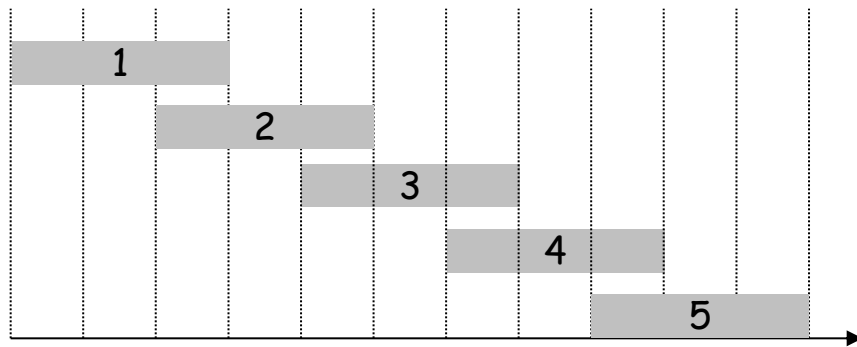
Return $\text{Compute-Opt}(n)$

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

Weighted Interval Scheduling: Brute Force

Observation. This brute force algorithm may take **exponential** time because of **redundant sub-problems**

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



Q: Any ideas on how to decrease running time?

Weighted Interval Scheduling: DP I - Memoization

Dynamic Programming I - 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  $\text{lastCompat}(1), \text{lastCompat}(2), \dots, \text{lastCompat}(n)$ 
```

```
 $M[0] = 0$  ← global array, want to have  $M[j] = \text{OPT}(j)$ 
```

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

```
     $M[j] = \text{empty}$ 
```

```
Run M-Compute-Opt( $n$ )
```

```
-----
```

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

```
    if ( $M[j]$  is empty)
```

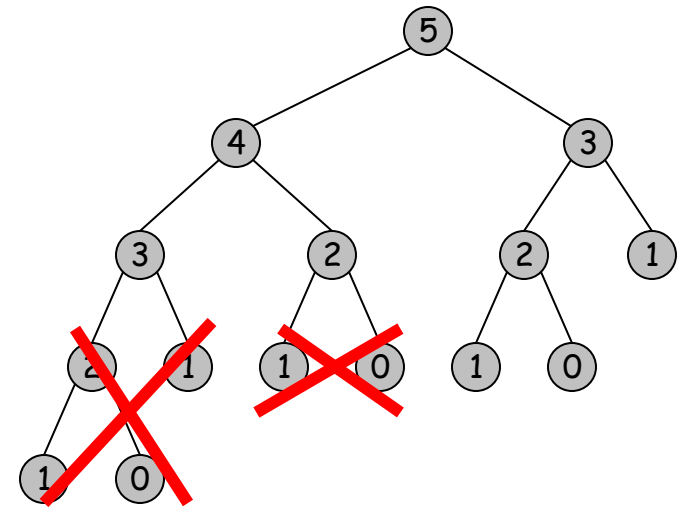
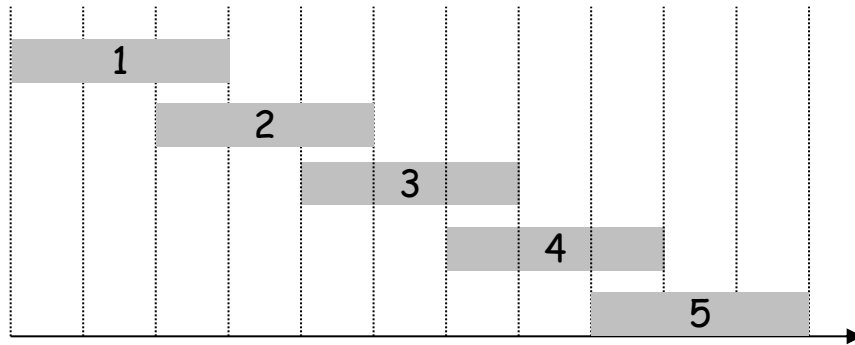
```
         $M[j] = \max(w_j + \text{M-Compute-Opt}(\text{lastCompat}(j)), \text{M-Compute-Opt}(j-1))$ 
```

```
    return  $M[j]$ 
```

```
}
```

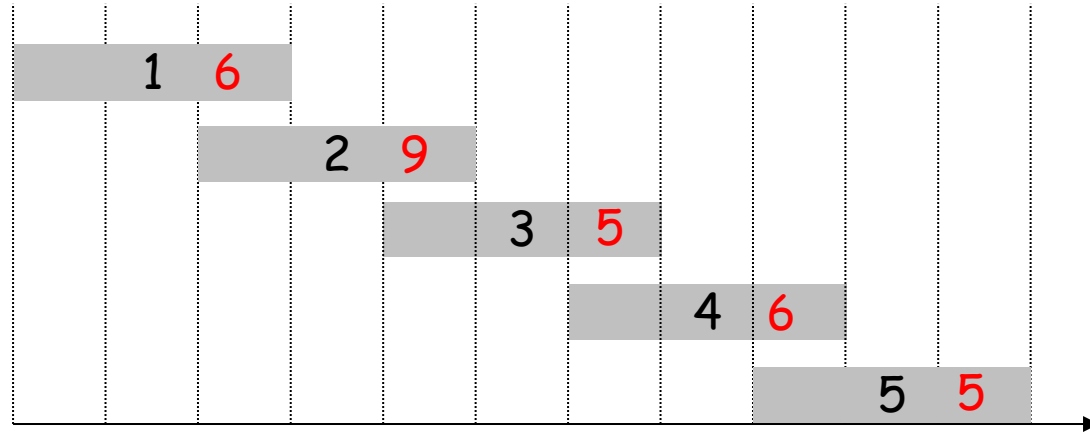

Weighted Interval Scheduling: Brute Force

With memoization:



Weighted Interval Scheduling: DP 1 - Memoization

Ex: Run the **memoization** algorithm on the following instance



$M =$

--	--	--	--	--	--

Weighted Interval Scheduling: DP 1 - Memoization

- Analysis of running time: similar to DFS
(need to add time to sort and compute lastCompat, we focus on rest)
- "Running time" = sum of costs of all calls $M\text{-Compute-Opt}(j)$, $j=0,\dots,n$
- Let us analyze $M\text{-Compute-Opt}(j)$ for a fixed j
 - The first time $M\text{-Compute-Opt}(j)$ is called, it makes two recursive calls
 - After the first time, it does not make any calls
 - So over the whole execution, $M\text{-Compute-Opt}(j)$ makes 2 calls
- Since $j=1,\dots,n$, the total number of calls is $O(n)$
- Each call takes constant time
- So the algorithm takes time $O(n)$.

Weighted Interval Scheduling: DP 2 - Bottom-Up

Dynamic Programming 2 - Bottom-up: Fill table M in **order** $M[0], M[1], \dots$

- When we try to fill $M[j]$ we already have all the information needed, namely $M[\text{lastCompat}(j)]$ and $M[j-1]$

```
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  $\text{lastCompat}(1), \text{lastCompat}(2), \dots, \text{lastCompat}(n)$ 
```

```
Iterative-Compute-Opt {
```

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

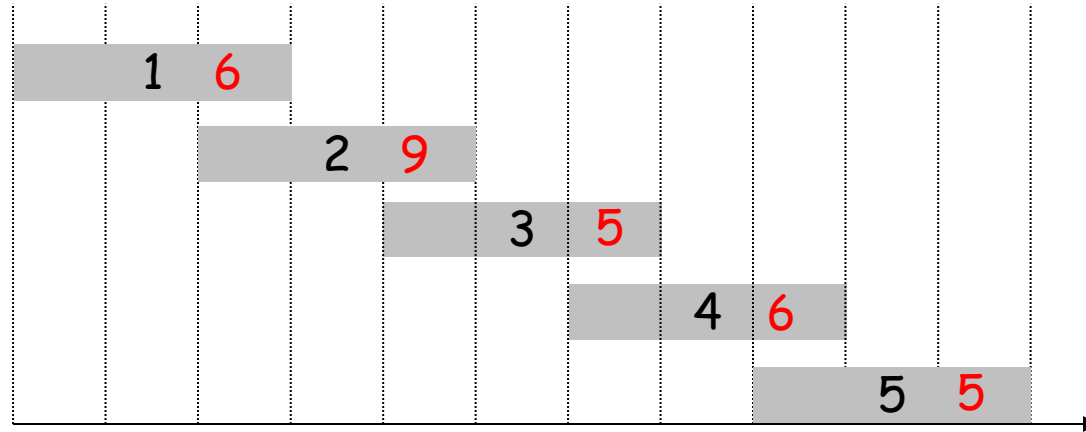
```
    for  $j = 1$  to  $n$ 
```

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

```
}
```

Weighted Interval Scheduling: DP 2 - Bottom-Up

Ex: Run the **bottom-up** algorithm on the following instance

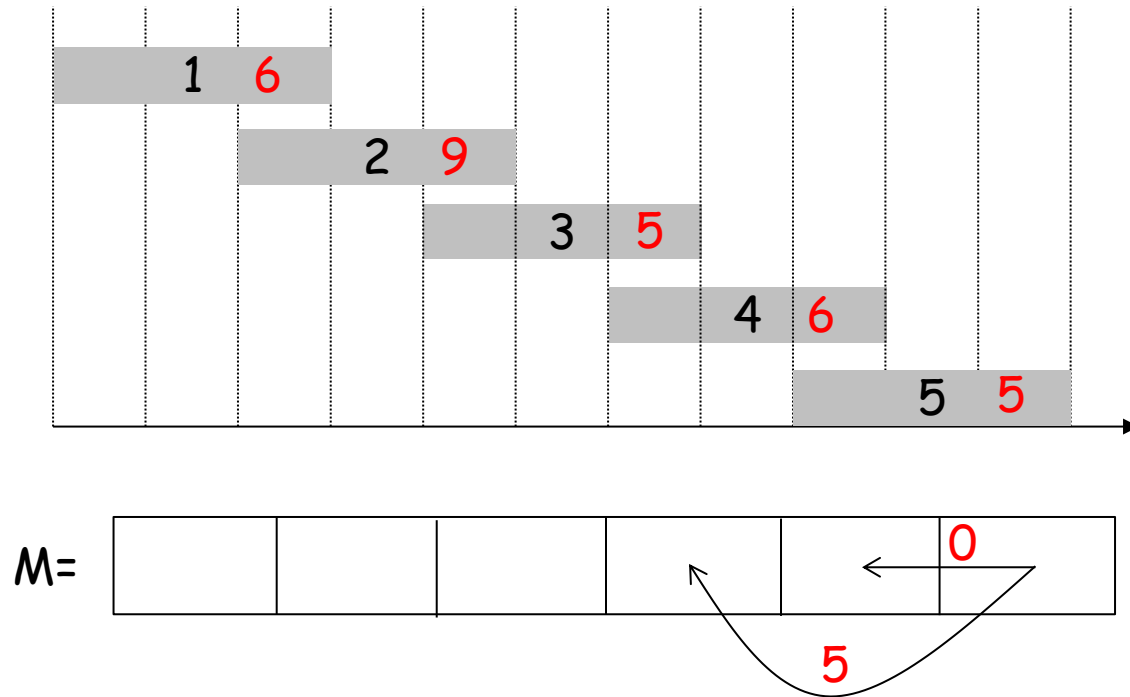


$M =$

--	--	--	--	--	--

Weighted Interval Scheduling: DP 3 - Shortest path

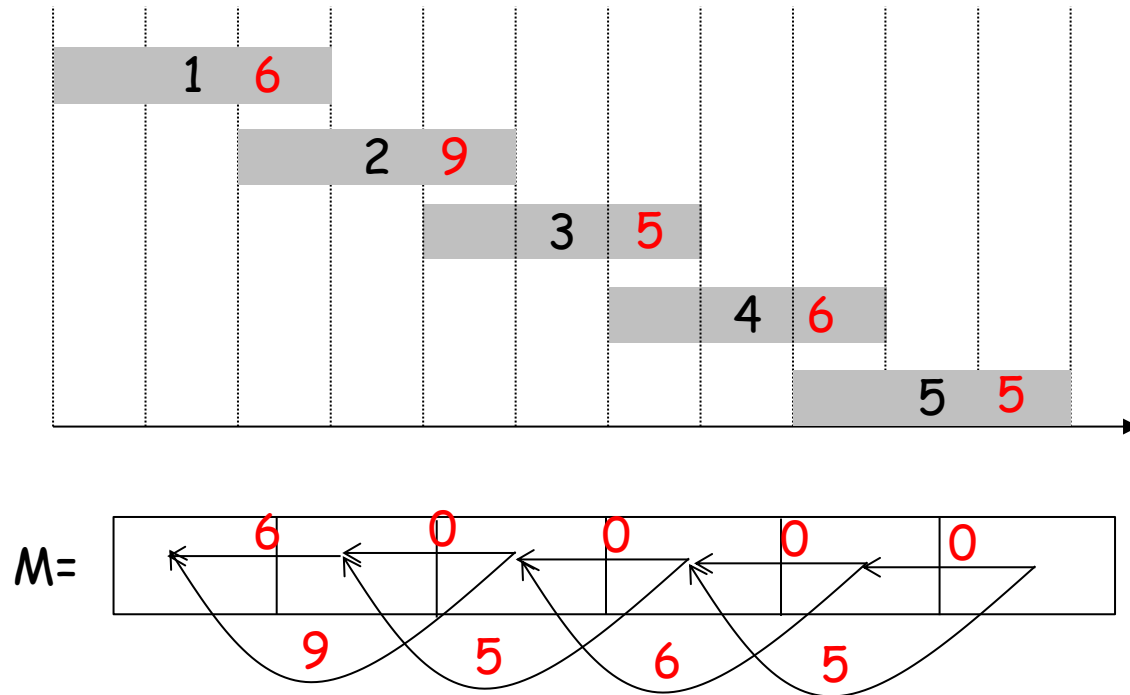
Remark: Every dynamic programming algorithm can also be seen as a shortest/longest path problem on a DAG



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

Weighted Interval Scheduling: DP 3 - Shortest path

Remark: Every dynamic programming algorithm can also be seen as a shortest/longest path problem on a DAG



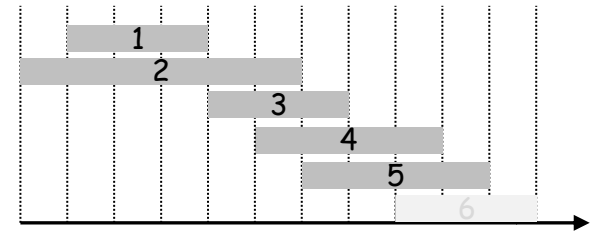
$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\{v_j + OPT(\text{lastCompat}(j)), OPT(j-1)\} & \text{otherwise} \end{cases}$$

How does Dynamic Programming solution looks like?

1) Break problem into **sub-problems**

Sub-problem i : consider only tasks $1, \dots, i$

$OPT(i)$ = optimal value of sub-prob i
= best subset of tasks $1, \dots, i$



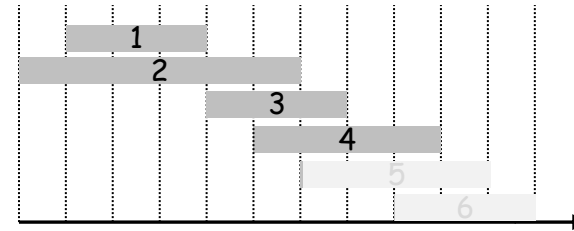
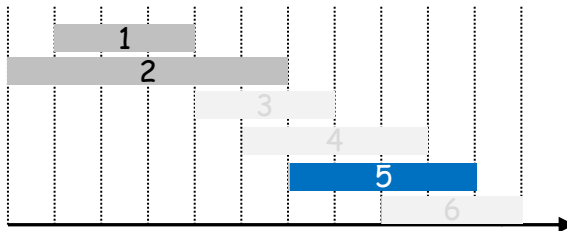
2) To solve sub-problem, **use smaller sub-problems**

Either:

include 5

does not

**Optimal
substructure**

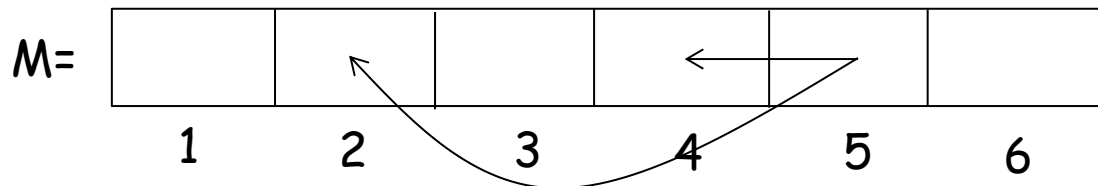


$$OPT(5) = \max\{ \text{val}(5) + OPT(2), \quad OPT(4) \}$$

How does Dynamic Programming solution looks like?

$$\text{OPT}(5) = \max\{ \text{val}(5) + \text{OPT}(2), \text{OPT}(4) \}$$

- 3) Create **table to store optimal** value of each sub-problem.
Fill up table in **starting from smallest** sub-problems
(so always have information needed)



Maior subsequência crescente

Maior subsequência crescente

Entrada:

$A = (a(1), a(2), \dots, a(n))$ uma sequência de números reais distintos.

Objetivo:

Encontrar a maior subsequência **crescente** de A

Exemplo

- $A = (2, 3, 14, 5, 9, 8, 4)$
- $(2, 3, 8)$ e $(3, 5, 9)$ são subsequências crescentes de tamanho 3
- As maiores subsequências crescentes de A são $(2, 3, 5, 9)$ e $(2, 3, 5, 8)$

Maior subsequência crescente

Obs: Note que não funciona tentar uma estratégia gulosa que **concatena** a_j a sequência atual se possível, e se não possível **reinicia** a sequência

Ex: (2,5,3,4)

Iteração 1: Mantem seq 2

Iteração 2: Mantem seq 2, 5

Iteração 3: Não pode concatenar 3 a essa sequência, reinicia: mantém seq 3

Iteração 4: Mantem seq 3,4 ← não é maior subseq crescente!

Tinha que ter visto que deveria guardar a subsequência 2 pra concatenar com o 3...

Difícil saber disso de antemão...

Maior subsequência crescente

Q: Sub-problemas?

- Seja $L(j)$: tamanho da maior subsequência crescente que **termina em $a(j)$** ($a(j)$ **pertence** a subsequência)
- Exemplo $A = (2, 3, 14, 5, 9, 8, 4)$
- $L(1)=1, L(2)=2, L(3)=3, L(4)=3, L(5)=4, L(6)=4, L(7)=3$
- O tamanho da maior subsequência crescente é

$$\max \{ L(1), L(2), \dots, L(n) \}$$

- Temos a seguinte equação para $L(j)$:
[Tente para $L(6)$ com $A = (2, 3, 14, 5, 9, 8, 4)$]

$$L(j) = \max_i \{ 1+L(i) \mid i < j \text{ e } a(i) < a(j) \}, \text{ para } j > 1$$
$$L(1) = 1$$

Maior subsequência crescente

Q: Sub-problemas?

- Seja $L(j)$: tamanho da maior subsequência crescente que **termina em $a(j)$** ($a(j)$ **pertence** a subsequência)
- Exemplo $A = (2, 3, 14, 5, 9, 8, 4)$
- $L(1)=1, L(2)=2, L(3)=3, L(4)=3, L(5)=4, L(6)=4, L(7)=3$
- O tamanho da maior subsequência crescente é

$$\max \{ L(1), L(2), \dots, L(n) \}$$

- Temos a seguinte equação para $L(j)$:
[Tente para $L(6)$ com $A = (2, 3, 14, 5, 9, 8, 4)$]

$$L(j) = \max\{1, \max_i \{ 1+L(i) \mid i < j \text{ e } a(i) < a(j) \}, \text{ para } j > 1 \}$$

$$L(1) = 1$$

Considera a opção de colocar $a(j)$ sozinho na sequência

Maior subsequência crescente

Algoritmo (versão bottom-up):

```
M(1) ← 1, pre(1) ← 0
For j=2 to n
  MAX ← -infinity, MAX_index ← 0
  For i=1 to j-1 //obtem MAX ← max, { 1+L(i) | i < j e a(j) > a(i) }
    If a(i) < a(j) and 1+M(i) > MAX then
      MAX ← 1+M(i)
      MAX_index ← i
  If MAX > 1 //seta L(j) ← max{1, MAX}
    M(j) ← MAX
    pre(j) ← MAX_index
  Else
    M(j) ← 1
    pre(j) ← 0

MSC ← max{M(1), M(2), ..., M(n)} // encontra maior L(j)
```

[Fazer traço pra exemplo A = (4, 2, 3, 5) com pre(.)]

Complexidade: $O(n^2)$

Maior subsequência crescente: encontrando a **subsequencia**

Q: Algoritmo anterior calcula **tamanho** da maior subsequencia crescente.
Como encontrar a **subsequencia em si**? Ideia: BFS/DFS/Dijkstra

A: Mantem $pre(i)$ para guardar qual das opcoes para $a(j)$ foi escolhida
Para essa problema, $pre(i)=0$ indica que $a(j)$ foi colocado sozinho,
 $pre(i)>0$ indica que selecionamos o item $a(pre(i))$ para ser o anterior
a $a(j)$ na sequencia

Ultimo elemento da seq 'e o que tem maior valor $L(j)$; percorre pre , etc.

```
Input: n, L(1), ..., L(n), pre(1), ..., pre(n), OPT
```

```
ultimo ← indice j de maior L(j)
```

```
Find_Subsequence(ultimo)
```

```
Proc Find_Subsequence(j)
```

```
  If j=0
```

```
    Return
```

```
  else
```

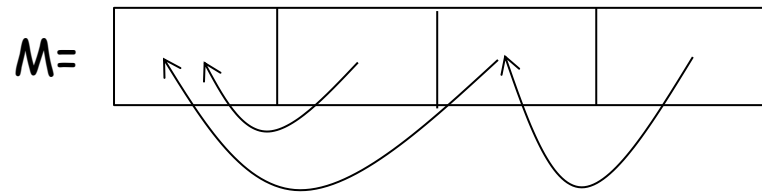
```
    Add j to the solution;
```

```
    Find_Subsequence(pre(j))
```


Maiores subseqüência crescente: encontrando a subseqüência

Ex: (2,5,3,4)

Setas indicando pre's:



Maior subsequência crescente

Q: Porque temos que forçar em $L(j)$ a sequência **terminar em $a(j)$** **(incluindo-o)?**

A: Senão não funciona

Ex: (6,7,4,5)

[Problema: $\tilde{L}(3)$ termina em "7", não pode ser usado em $\tilde{L}(4)$]

Maior subsequência crescente

Exercício: Escreva a versão da programação dinâmica com memoização para resolver esse problema

Exercise: Placing billboards

Placing billboards

Exercise: You need to decide where to put multiple advertisement on a highway of M kms.

- There are n possible places where you can place an advertisement given by x_1, x_2, \dots, x_n in $[0, M]$.
- Placing an advertisement at x_i gives value r_i .
- You cannot put two advertisements at distance < 5 kms from each other.
- **Goal:** Find best set of places to put advertisement.

Ex: $M=20$, $\{x_1, x_2, x_3, x_4\} = \{6, 7, 12, 14\}$, and $\{r_1, r_2, r_3, r_4\} = \{5, 6, 5, 1\}$.

One optimal solution is to put advertisement at x_1 and x_3

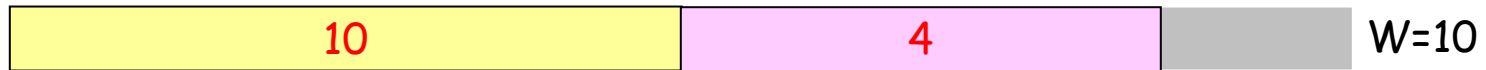
Solve this problem using dynamic programming

6.4 Knapsack Problem

Knapsack Problem

Knapsack problem.

- Given n objects and a backpack of **size W**
- Item i has **size $w_i > 0$** and has **value $v_i > 0$** .
- Sizes are **integers**.
- **Goal:** pick set of items that fit in the backpack and maximize total value.



Ex: { 3, 4 } has value 40.

W = 11

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

Knapsack Problem: Greedy Attempt

Greedy: repeatedly add item with maximum ratio v_i / w_i .

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

.

W = 11	Item	Value	Size
	1	1	1
	2	6	2
	3	18	5
	4	22	6
	5	28	7

Dynamic Programming: False Start

Q: Sub-problems?

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 which 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 occupation limit w.

Q: What is a recursive expression for $OPT(i, w)$?

- Case 1: OPT does not select item i .
 - OPT selects best of $\{ 1, 2, \dots, i-1 \}$ using size limit w
- Case 2: OPT selects item i (if there is space, namely $w \geq w_i$)
 - new size limit = $w - w_i$
 - OPT selects best of $\{ 1, 2, \dots, i-1 \}$ using this new size 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

Dynamic programming. Fill up an n -by- W array to compute $OPT(i,w)$

Q: In which order should we fill this array?

A: Start with $OPT(0, 0)$, then $OPT(0, 1)$, $OPT(0, 2)$...; then $OPT(1,0)$, $OPT(1,2)$,...

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

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

Knapsack Problem: Running Time

Running time. $\Theta(n W)$.

- **Not polynomial** in input size! The input size is $\sim(\log W + n)$
- "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]

Knapsack Problem: Obtaining Optimal Items

$OPT(n, W)$ only gives the maximum **value** we can obtain, but not which **items** we should choose to obtain such value

To get items: For each subproblem $OPT(i, w)$ maintain $pre(i, w)$ indicating which cell of the table was used to compute $OPT(i, w)$:

$pre(i, w)$ is either $(i-1, w)$ or $(i-1, w-w_i)$

Chose to **add**
item i in subproblem $(i-1, w)$

Chose **not** to add
item i in subproblem $(i-1, w)$

```
for i = 1 to n
  for w = 1 to W
    if ( $w_i > w$ )
       $M[i, w] = M[i-1, w]$ 
       $pre[i, w] = (i-1, w)$ 
    else
      if  $M[i-1, w] > v_i + M[i-1, w-w_i]$ 
         $M[i, w] = M[i-1, w]$ 
         $pre[i, w] = (i-1, w)$ 
      else
         $M[i, w] = v_i + M[i-1, w-w_i]$ 
         $pre[i, w] = (i-1, w-w_i)$ 
```

Knapsack Problem: Obtaining Optimal Items

Then follow the pre's to find the set of items that should be included

```
Get_items
  itemsPicked = []
  i = n          //starting subproblem: the whole thing
  w = W
  itemsPicked = []

  While True
    If pre[i,w] = (i-1,w-wi) //chose to add item i
      itemsPicked.append(i)
    Else //chose not to add item I
      [do nothing]
      (i,w) = pre[i,w] //follow the pre

  retrun itemsPicked
```

Knapsack Problem

Exercise: Implement the algorithm for solving the knapsack problem and returning the optimal set of items to be chosen.

Exercise: A moving consulting company

A moving consulting company

Exercise: You have a small consulting company. Your clients are mostly in Rio and Sao Paulo.

- In each month it can run its business either from a Rio office or Sao Paulo office.
- In month i you have **cost R_i** if run from Rio, and **S_i** if run from Sao Paulo
- If you run the business from one city at month i and another at month $i+1$, then you need to spend a fixed cost **M** for moving costs.
- **Goal:** Given n months, decide where your office should be in every month to minimize total cost

Ex: $M = 10$, $\{R_1, R_2, R_3, R_4\} = \{1, 3, 20, 30\}$, $\{S_1, S_2, S_3, S_4\} = \{50, 20, 2, 4\}$

Optimal solution is [Rio, Rio, SP, SP], with cost $1+3+2+4+10=20$

1. Show that the strategy of running the office from the city with smallest costs in each month does not minimize the total cost
2. Solve this problem using dynamic programming [Find actual solution]

6.6 Sequence Alignment

String Similarity

How similar are two strings?

- `ocurrance`
- `occurrence`

Idea: Use gaps to align the strings,
count #mismatches

(gaps = inserted/deleted character)

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

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

6 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

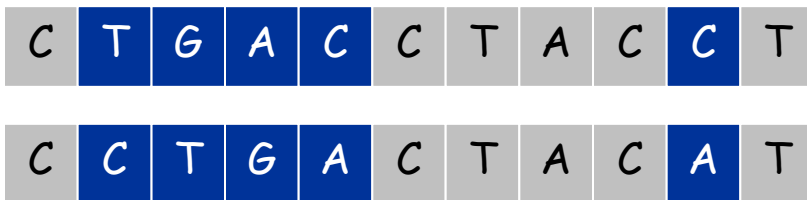
Applications.

- Basis for Unix diff.
- Auto correction, spell checking
- Computational biology.

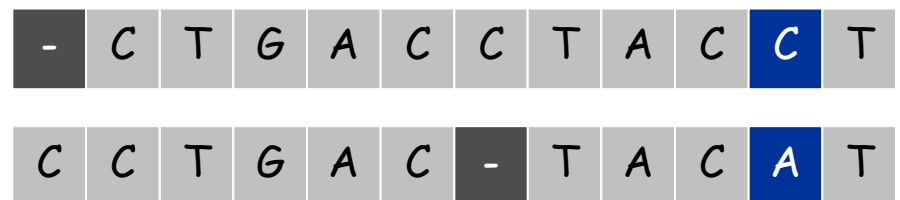
Edit distance. [Levenshtein 1966, Needleman-Wunsch 1970]

- Gap penalty δ ; penalty α_{pq} for matching letter p with q (usually 0 if $p=q$)
- Cost: sum of gap and matching penalties.

Goal: Given two strings $X = x_1 x_2 \dots x_m$ and $Y = y_1 y_2 \dots y_n$ find alignment of minimum cost.



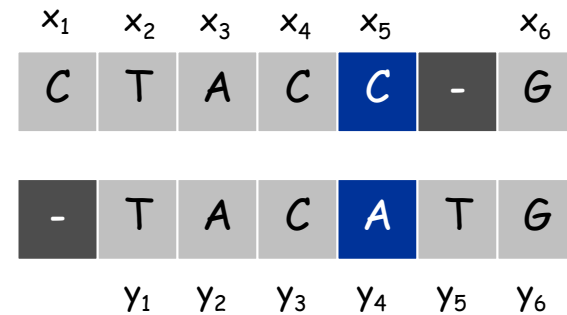
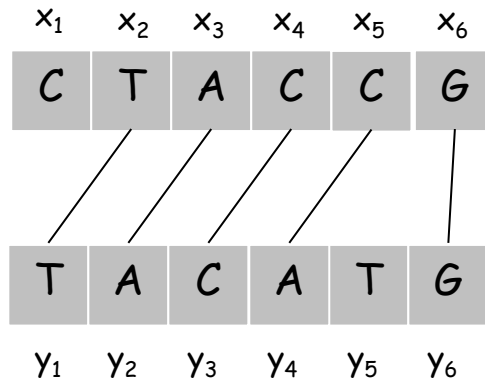
$$\alpha_{TC} + \alpha_{GT} + \alpha_{AG} + 2\alpha_{CA}$$



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

Sequence Alignment

We can this as a **matching** problem:

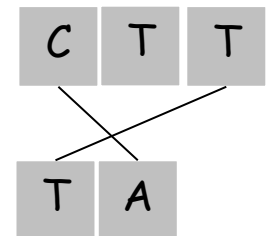


Unmatched is equivalent to matched to a **gap**

Cost of matching:

- Pay gap penalty δ for each unmatched letter
- Pay matching penalty α_{pq} for each matched pair

Crucial property: Valid matching does not have **crossings** (otherwise cannot represent using gaps)



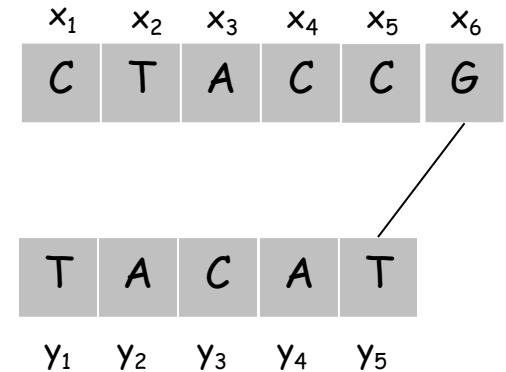
Sequence Alignment: Optimal substructure

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

Q: How can we write $OPT(i, j)$ in a recursive way, in terms of smaller subproblems?

Option 1: solution matches last characters x_i and y_j

- Best to do is pay penalty α_{x_i, y_j} + best matching of rest $x_1 \dots x_{i-1}$ and $y_1 \dots y_{j-1}$ ($=OPT(i-1, j-1)$)



Option 2: does not match last characters.

Then one of them has to be **unmatched**, otherwise there is crossing

- **Option 2a:** leave x_i unmatched

- Best is to pay gap for x_i + best matching of rest $x_1 \dots x_{i-1}$ and $y_1 \dots y_j$ ($=OPT(i-1, j)$)

- **Option 2b:** leave y_j unmatched

- Best is to pay gap for y_j + best matching of rest $x_1 \dots x_i$ and $y_1 \dots y_{j-1}$ ($=OPT(i, j-1)$)

Sequence Alignment: Optimal substructure

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

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[i, 0] =  $i\delta$   
  for j = 0 to n  
    M[0, j] =  $j\delta$   
  
  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: $m = n = 1.000.000$. 1 trillions ops OK, but 1 Tb array?

6.7 Sequence Alignment in Linear Space

Sequence Alignment: Value of OPT with Linear Space

```
Sequence-Alignment(m, n,  $x_1x_2\dots x_m$ ,  $y_1y_2\dots y_n$ ,  $\delta$ ,  $\alpha$ ) {  
  
  for i = 0 to m  
    CURRENT[i] =  $i\delta$   
  end for  
  
  for j = 1 to n  
    LAST ← CURRENT      % vector copy  
    CURRENT[0] ←  $j\delta$   
    for i = 1 to m  
      CURRENT[i] ← min( $\alpha[x_i, y_j] + \text{LAST}[i-1]$ ,  
                        $\delta + \text{LAST}[i]$ ,  
                        $\delta + \text{CURRENT}[i-1]$  )  
    end for  
  end for  
  return CURRENT[m]
```

- Two vectors of of m positions: LAST e CURRENT
- $O(mn)$ time and $O(m+n)$ space

Sequence Alignment: Value of OPT with Linear Space

