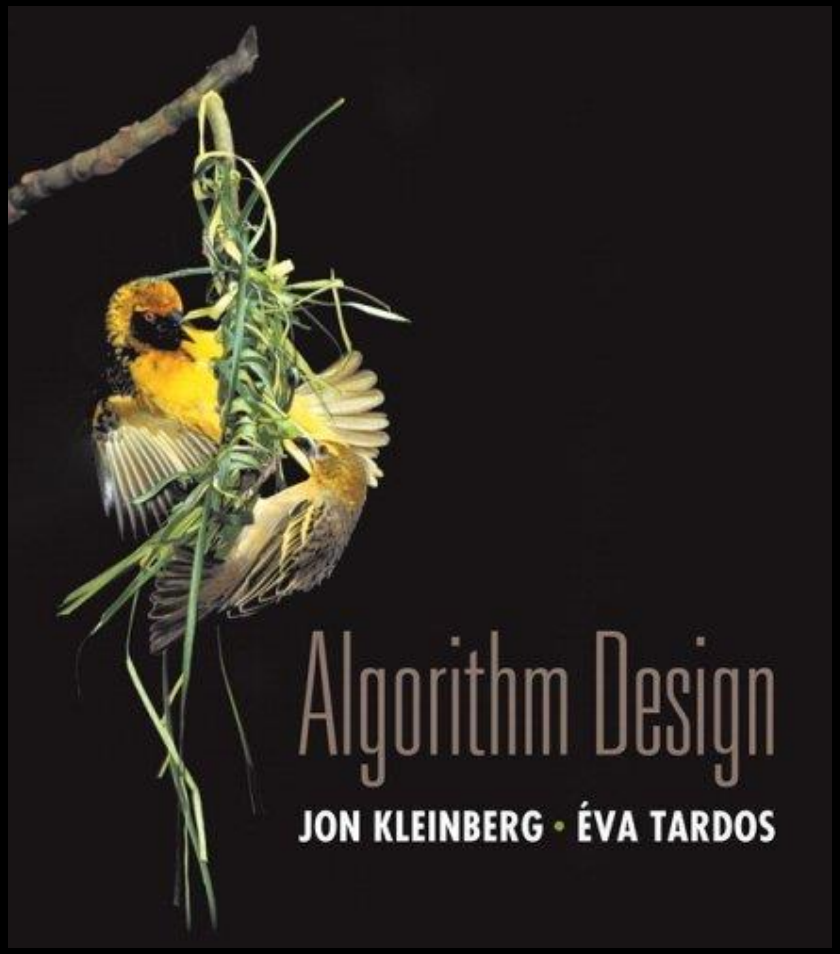


Chapter 6

Dynamic Programming

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.

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.

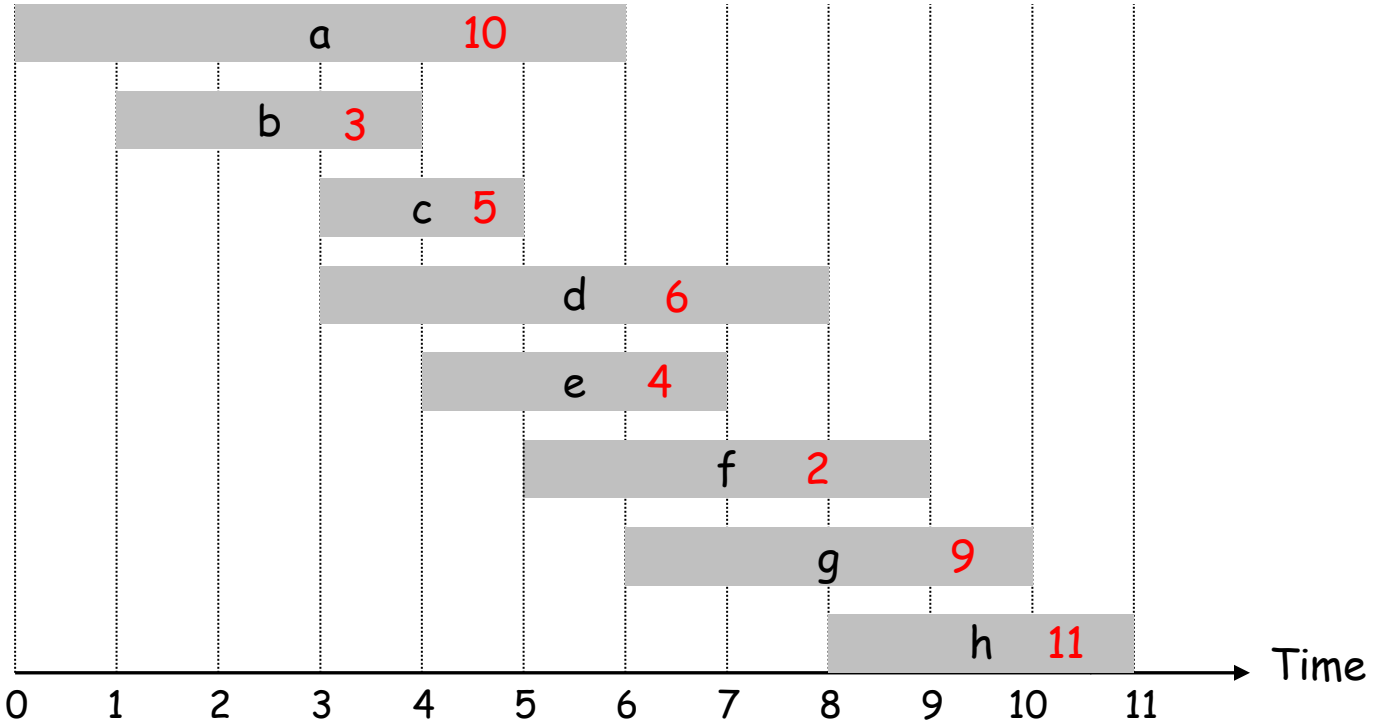
- Viterbi for hidden Markov models.
- Unix diff for comparing two files.
- DNA sequence comparison.
- Bellman-Ford for shortest path routing in networks.
- Cocke-Kasami-Younger for parsing context free grammars.

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

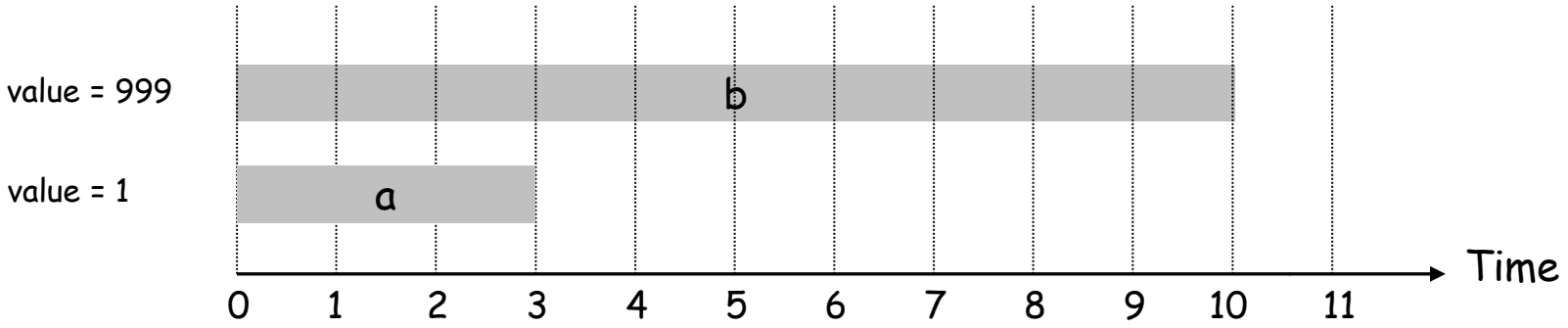


Unweighted Interval Scheduling Review

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.

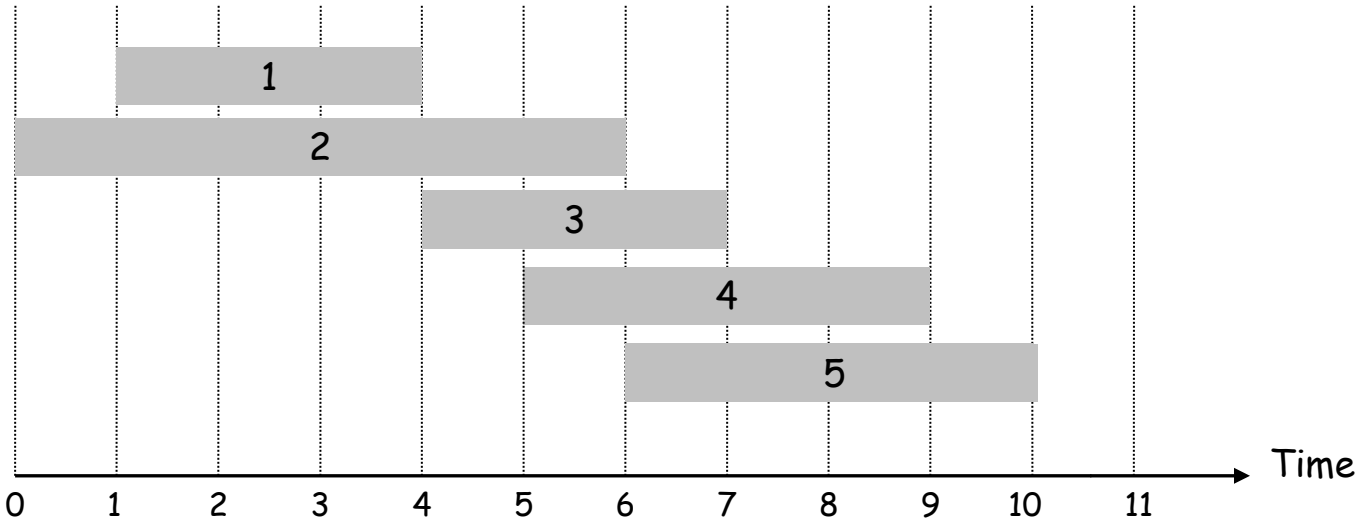


Weighted Interval Scheduling

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

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



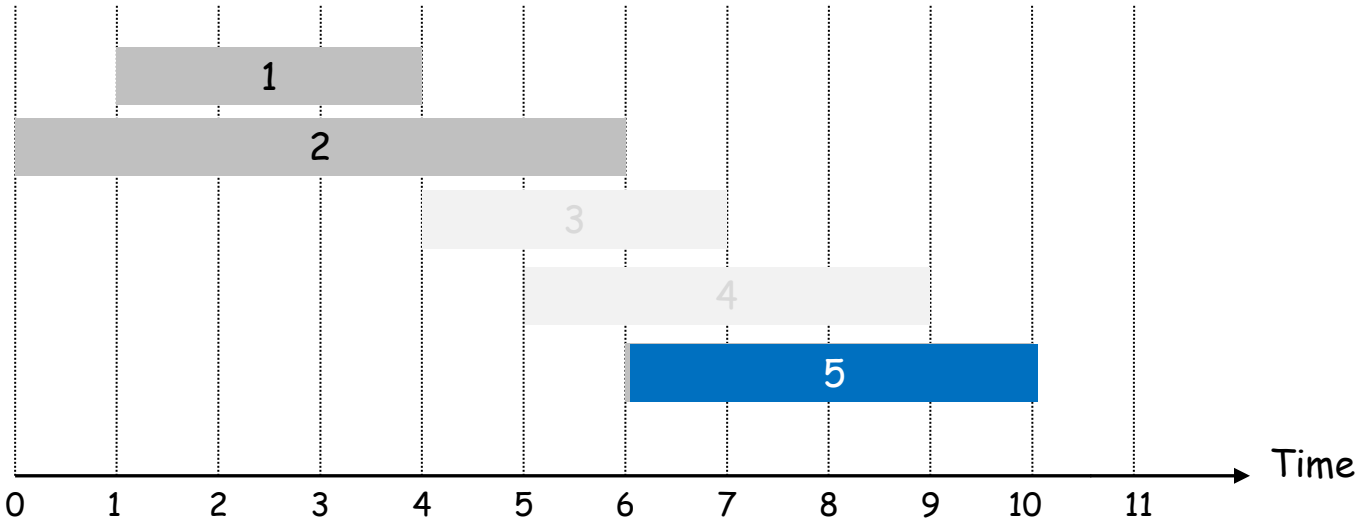
Weighted Interval Scheduling

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

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



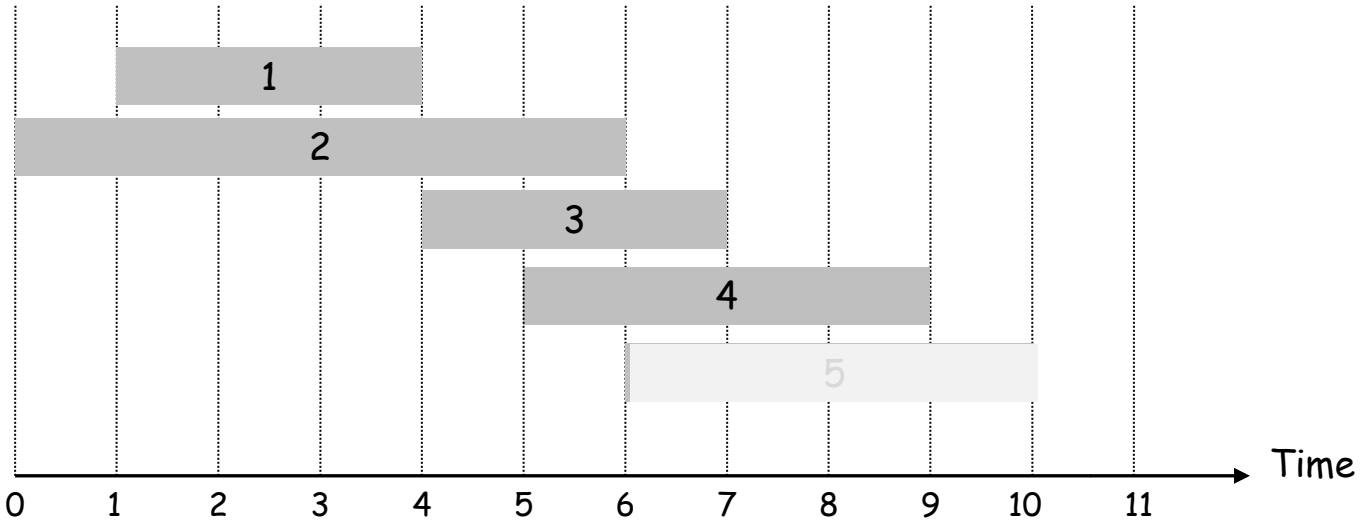
Weighted Interval Scheduling

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

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
2. If do not include job 5 => select optimally among jobs 1,..,4

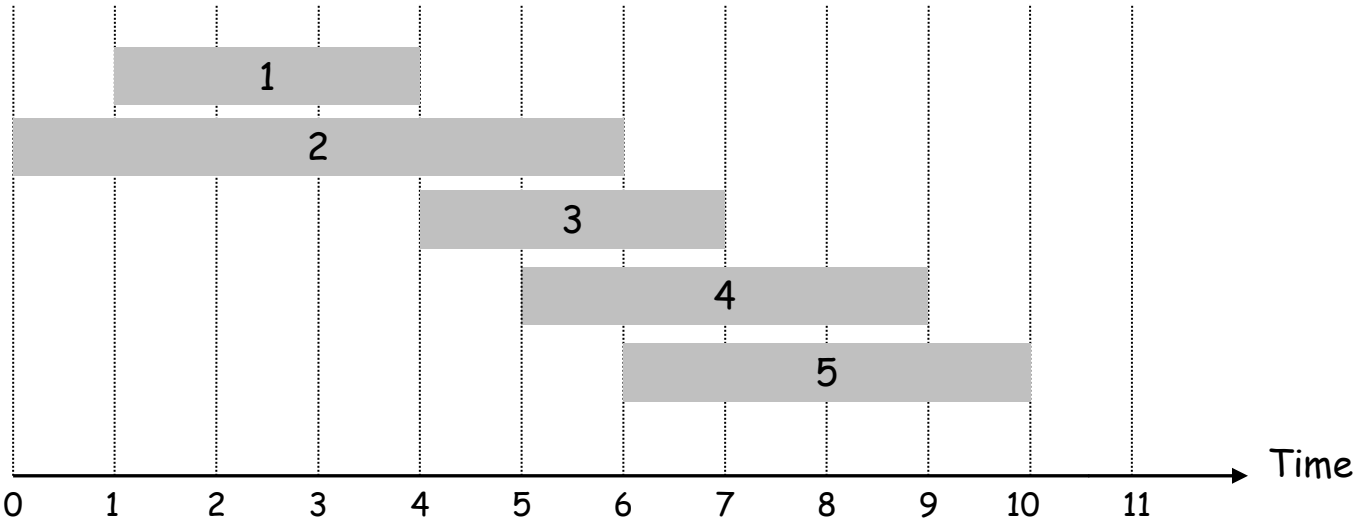


Weighted Interval Scheduling

More generally, define

lastCompat(j) = largest index i < j such that job i is compatible with j

Ex: lastCompat(5) = 2, lastCompat(4) = 1, lastCompat(1) = 0



Weighted Interval Scheduling

More generally, define

lastCompat(j) = largest index i < j such that job i is compatible with j

OPT(j) = value of optimal solution to the problem consisting of jobs 1, 2, ..., j

To compute OPT(j) we have two options:

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

Pick the best option

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

Weighted Interval Scheduling: Brute Force

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

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

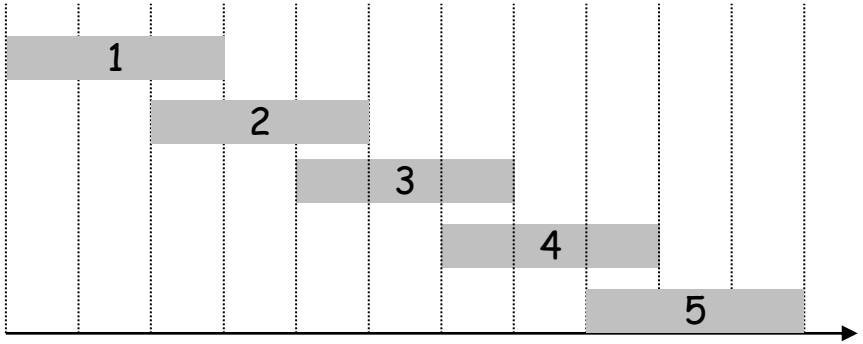
Brute force algorithm.

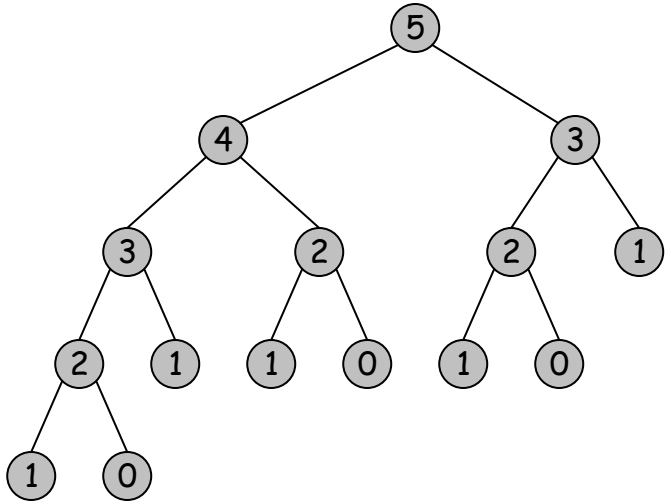
```
Input: s1,…,sn , f1,…,fn , v1,…,vn

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

Return Compute-Opt(n)

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

Weighted Interval Scheduling: Brute Force

Observation. This brute force algorithm takes exponential time because of redundant sub-problems

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

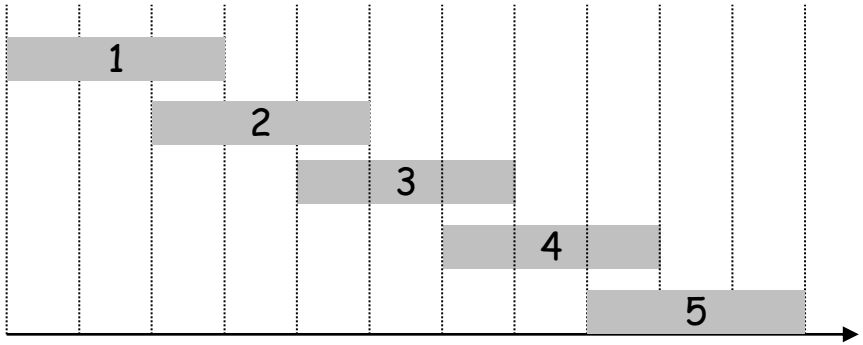


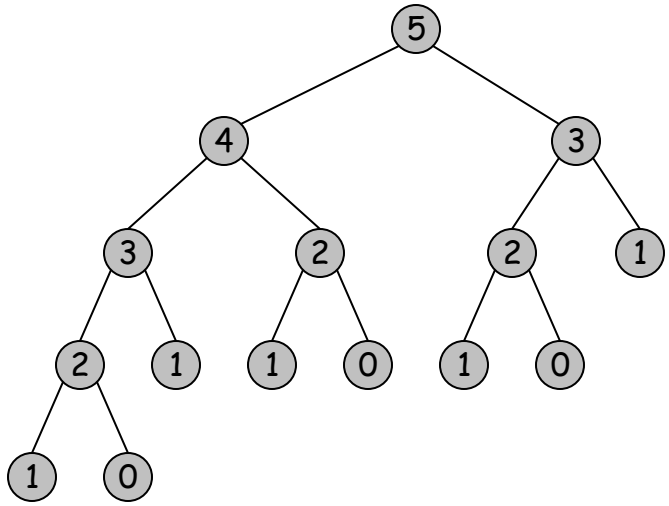


Weighted Interval Scheduling: Brute Force

Observation. This brute force algorithm takes 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?

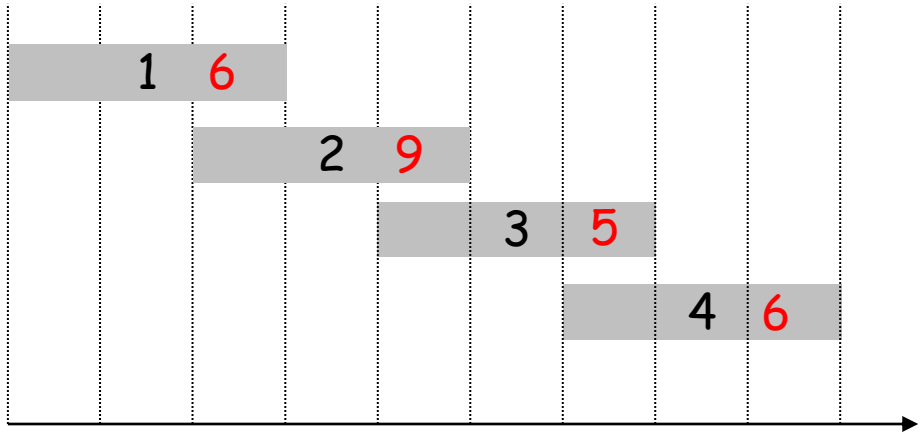
Weighted Interval Scheduling: DP 1 - Memoization

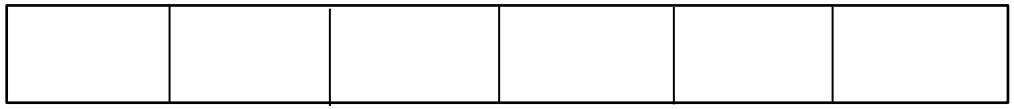
Dynamic Programming I - Memoization. Store results of each sub-problem in a cache; lookup as needed.

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

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

                    ← global array, want to have M[j]=OPT(j)
M[0] = 0
for j = 1 to n
  M[j] = empty
Run M-Compute-Opt(n)

-----------------
M-Compute-Opt(j)
  if (M[j] is empty)
    M[j] = max(vj + M-Compute-Opt(lastCompat(j)), M-Compute-Opt(j-1))
  return M[j]
```

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 (we will ignore the time to sort and compute lastCompat to focus on main part)

- Running time = sum of costs of all calls M-Compute-Opt(j), j=0,…,n

- Let us analyze M-Compute-Opt(j) for a fixed j
  - The first time M-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-Compute-Opt(j) makes 2 calls

- Since j=1,…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

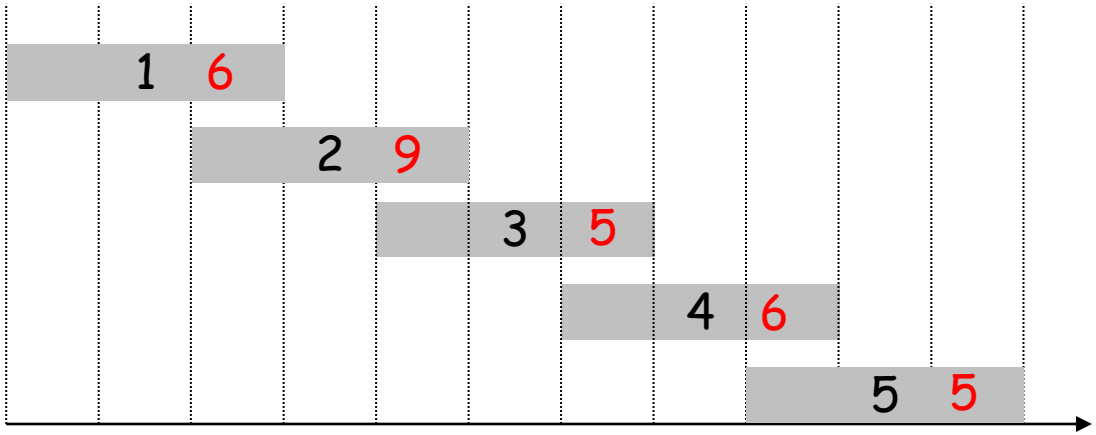
Dynamic Programming 2 - Bottom-up: Fill table M in order M[0], M[1],…

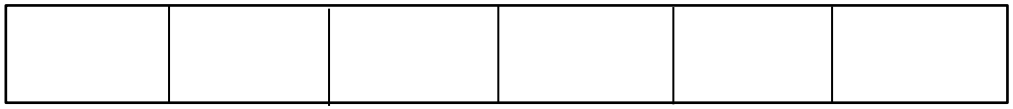
- When we try to fill M[j] we already have all the information needed, namely M[lastCompat(j)] and M[j-1]

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

Sort jobs by finish times so that f_1 ≤ f_2 ≤ ... ≤ f_n.
Compute lastCompat(1), lastCompat(2), …, lastCompat(n)

M = [] * n
M[0] = 0

for j = 1 to n
   M[j] = max(v_j + M[lastCompat(j)], M[j-1]) //M[j] = OPT(j)
```

Weighted Interval Scheduling: DP 2 – Bottom-Up

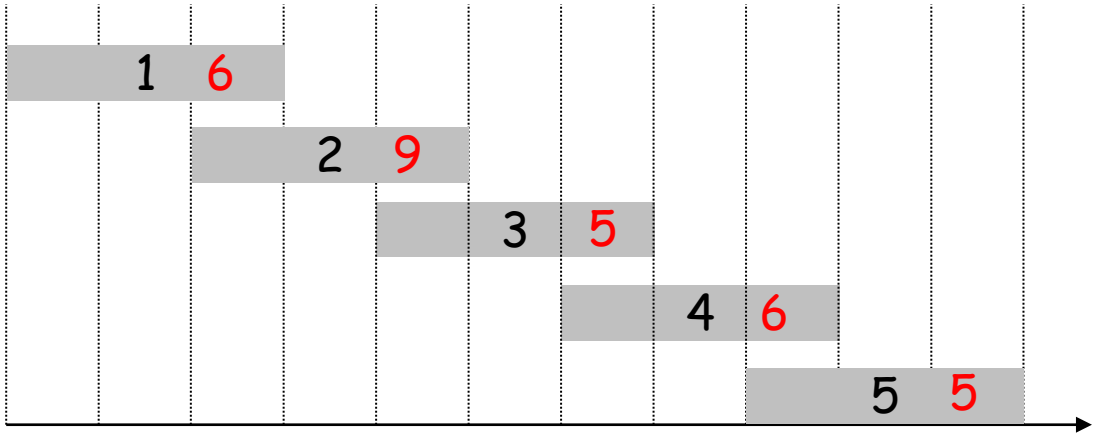
Ex: Run the bottom-up algorithm on the following instance

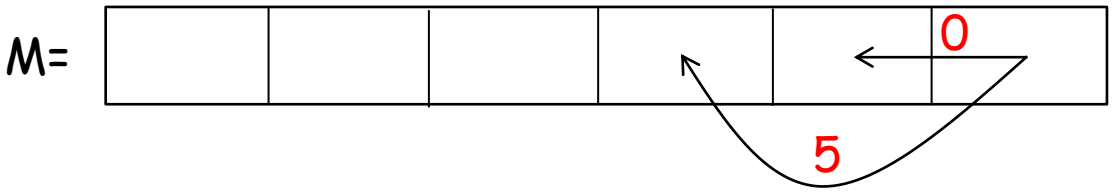


M= 

Weighted Interval Scheduling: DP 3 – Shortest path

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

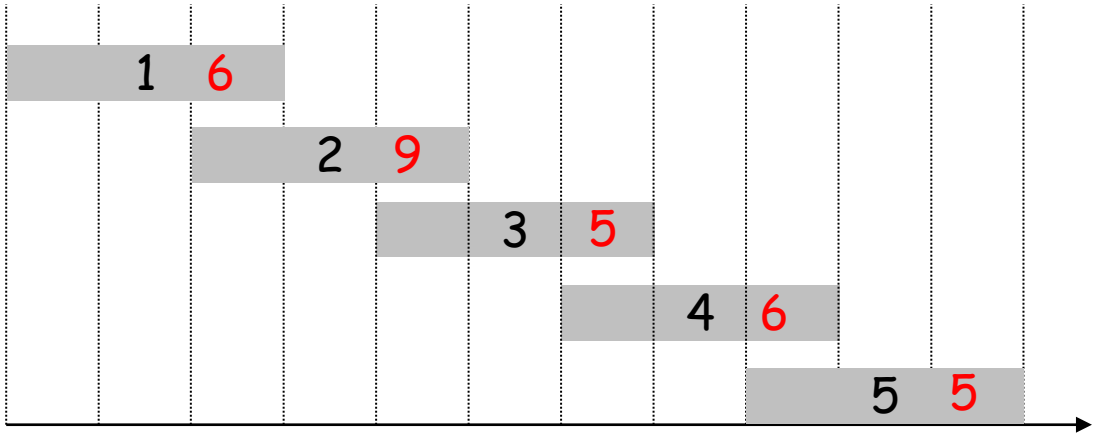


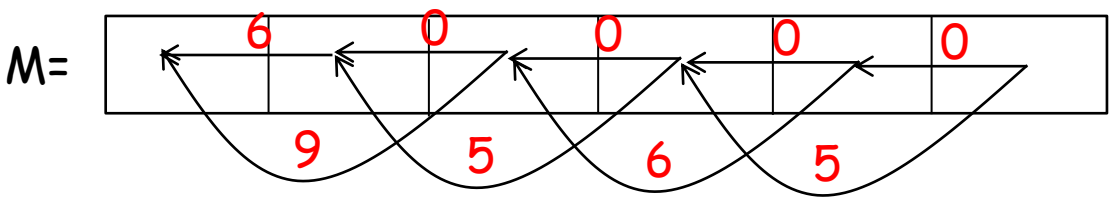


$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\{v_j + OPT(lastCompat(j)), OPT(j-1)\} & \text{otherwise} \end{cases}$$

Weighted Interval Scheduling: DP 3 – Shortest path

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





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

Weighted Interval Scheduling: Finding a Solution

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

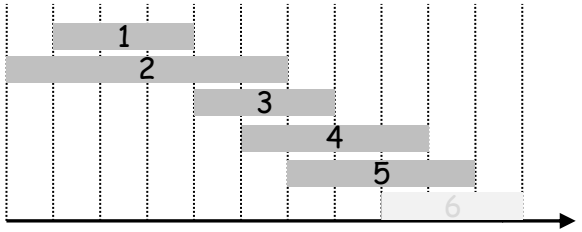
A: Parent

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

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

M = [] * n
M[0] = 0

for j = 1 to n
   M[j] = max(vj + M[lastCompat(j)], M[j-1]) //M[j] = OPT(j)
```

How does Dynamic Programming solution looks like?

1) Break problem into sub-problems

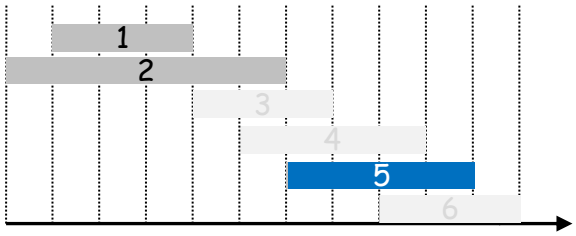
```
Sub-problem i: consider only tasks 1,…,i

OPT(i) = optimal value of sub-prob i
       = best subset of tasks 1,…,i
```

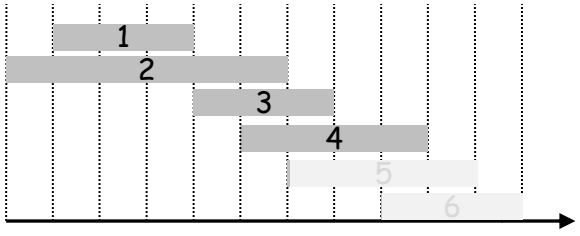


Optimal substructure

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

Decision: include 5 | does not



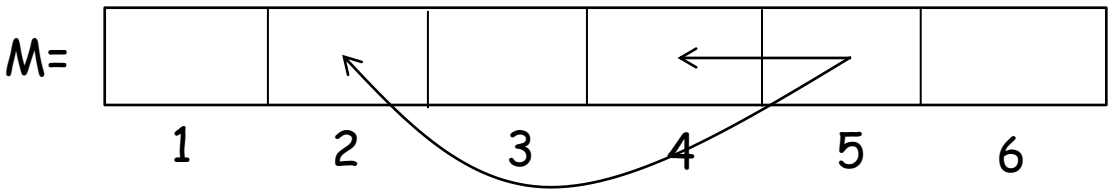



```
OPT(5) = max{ val(5) + OPT(2),    OPT(4)}
```

How does Dynamic Programming solution looks like?

```
OPT(5) = max{ val(5) + OPT(2),     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: 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

Q: Sub-problemas?

Maior subsequência crescente

Q: Sub-problemas?

OPT(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)
OPT(1)=1, OPT(2)=2, OPT(3)=3, OPT(4)=3, OPT(5)=4, OPT(6)=4, OPT(7)=3

O tamanho da maior subsequência crescente é

$$\max \{ OPT(1), OPT(2), \ldots, OPT(n) \}$$

Maior subsequência crescente

Pra calcular OPT(j), consideramos a decisão:

Quem 'e o item anterior ao A[j] na sequencia crescente?

Olhando pra todas possibilidade para essa decisão temos a seguinte equação para OPT(j):

$$OPT(j) = \max_i \{ 1 + OPT(i) \mid i < j \text{ e } A[i] < A[j] \}, \text{ para } j > 1$$
$$OPT(1) = 1$$

[Tente para OPT(6) com A= (2, 3, 14, 5, 9, 8, 4)]

Maior subsequência crescente

Pra calcular OPT(j), consideramos a decisão:

Quem 'e o item anterior ao A[j] na sequencia crescente?

Olhando pra todas possibilidade para essa decisão temos a seguinte equação para OPT(j):

$$OPT(j) = \max\{1, \max_i \{ 1 + OPT(i) \mid i < j \text{ e } A[i] < A[j] \}, \text{ para } j>1 \}$$
$$OPT(1) = 1$$

[Tente para OPT(6) com A= (2, 3, 14, 5, 9, 8, 4)]

Maior subsequência crescente

$$OPT(j) = \max\{1, \max_i \{ 1 + OPT(i) \mid i < j \text{ e } A[i] < A[j] \}, \text{ para } j>1 \}$$
$$OPT(1) = 1$$

```
Input: A

M = [] * n
M[1] = 1
for j=2 to n
  MAX = -infty
  for i=1 to j-1 // faz MAX <- max_i {1+OPT(i) | i < j e A[j] > A[i]}
    if A[i] < A[j] then
      if 1+M[i] > MAX
        MAX = 1+M[i]
  M[j] = max{ 1, MAX } // faz OPT(j) = max{1, MAX}

return max{M[1], M[2], …, M[n]}  // retorna maior sequencia
```

[Fazer traço pra exemplo $A = (4, 2, 3, 5)$]

Complexidade: $O(n^2)$

Maior subsequência crescente

Exercicio: Escreva a versão da programação dinâmica com memoização para resolver esse problema de maior subsequencia crescente

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 x1, x2, .., xn in [0, M]
- Placing an advertisement at xi gives value ri
- You cannot put two advertisements at distance $\leq$ 5kms from each other
- **Goal**: Find best set of places to put advertisement

Ex: M=20, {x1,x2,x3,x4} = {6, 7, 12, 14}, and {r1,r2,r3,r4} = {5,6,5,1}
One optimal solution is to put advertisement at x1 and x3

Solve this problem using dynamic programming

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?

OPT(i) = max profit subset of items 1, …, i

Decision: select or not item i

- Case 1: does not select item i
  - make best selection of items { 1, 2, …, i-1 }

- Case 2: select 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

Better:

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

Decision: select or not item i

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

- Case 1: does not select item i
  - make best selection of items { 1, 2, …, i-1 } using occupation limit w

- Case 2: select item i
  - Get value v_i and occupies size w_i out of occupation limit w
  - Selects best of { 1, 2, …, i–1 } that fits in remaining space w - w_i

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

Knapsack Problem: Bottom-Up

Algorithm. Fill up an n-by-W array to compute all 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: w_1,…,w_N, v_1,…,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 →

n + 1 ↓

	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, 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

Obtaining the optimal items

OPT(n,W) gives the optimal value that we can get in the instance

Q: How can we get the optimal items that should actually be selected?

Obtaining the optimal items

Idea: (Like BSF, DFS, Dijstra) Use parent(i,w) to see where the optimal value OPT(i,w) came from

Then starting from (n,W), go to parent, parent, etc. to figure out items

```
for w = 0 to W
   M[0, w] = 0
   parent[0, w] = empty

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

Obtaining the optimal items

Idea: (Like BSF, DFS, Dijstra) Use parent(i,w) to see where the optimal value OPT(i,w) came from

Then starting from (n,W), go to parent, parent, etc. to figure out items

```
(continued)
#start at (n,W), run using the parents

bestItems = []

i = n
w = W
While parent[i,w] != empty
   #best option at this point was to take the item
   If parent[i,w] = (i-1, w - w_i)
       bestItems.append(i)
   Elseif parent[i,w] = (i-1, w)
       #do not want to use item i

Return bestItems
```

Knapsack Problem: Running Time

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

- Not polynomial in input size! The input size is $(\log W + n)$
- "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 Kleinberg-Tardos book]

Knapsack Problem

Exercise: Write down a pseudo-code to give what are the items in the optimal solution (the previous algorithm only gives the value of the optimal solution)

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 Ri if run from Rio, and Si 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, {R1, R2, R3, R4} = {1, 3, 20, 30}, {S1, S2, S3, S4} = {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

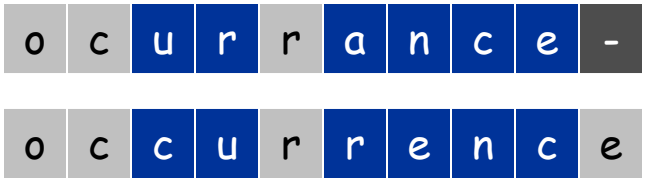
2. Solve this problem using dynamic programming [Find actual solution]

Edit Distance

Edit Distance

How similar are two strings?

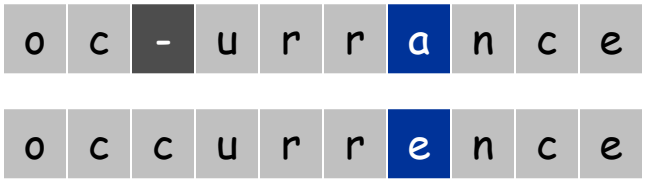
- ocurrance
- occurrence



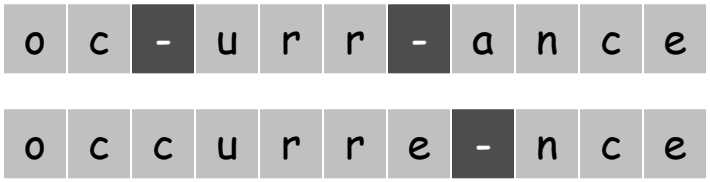
6 substitutions, 1 insertion

Idea: Count minimum number of operations needed to transform one string into the other

- Insertion
- Deletion
- Character substitution



1 insertion, 1 substitution



2 insertion, 1 del

Edit Distance

Problem: Given two strings x=x1,…,xn and y=y1,…,ym compute the minimum number of operations to transform x into y

x =	o	c	u	r	r	a	n	c	e
y =	o	c	c	u	r	r	e	n	c	e

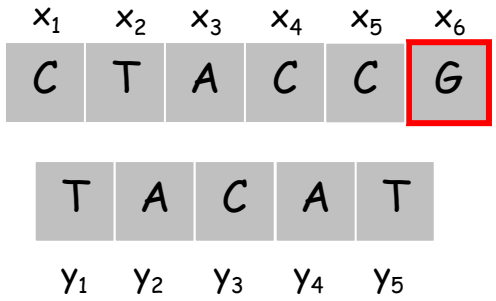
Applications.

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

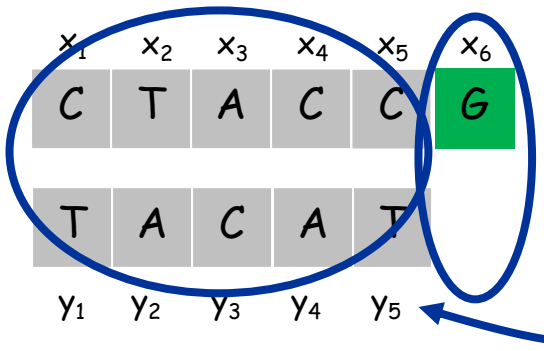
Edit Distance: Optimal substructure

Subproblems: OPT(i, j) = min cost of transforming $x_1\ x_2\ \ldots\ x_i$ into $y_1\ y_2\ \ldots\ y_j$.

Recursive relationship: Think of decision of which operation to apply to the end of the first string



Option 1: Delete letter x_i

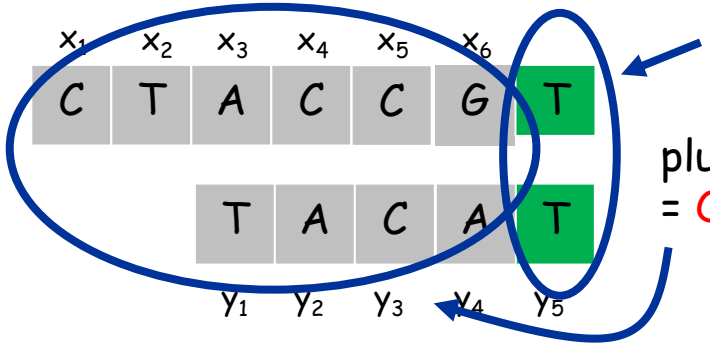


pay for 1 deletion

plus cost of transforming the rest = OPT(i-1,j)

Edit Distance: Optimal substructure

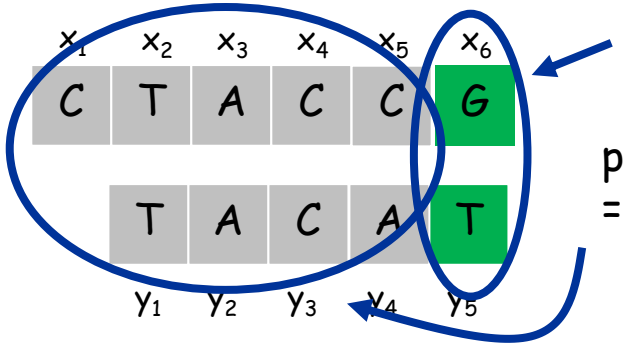
Option 2: Insert letter y_j after x_i



pay for 1 insertion

plus cost of transforming the rest
= OPT(i,j-1)

Option 3: If $x_i \neq y_j$, substitute x_i for y_j

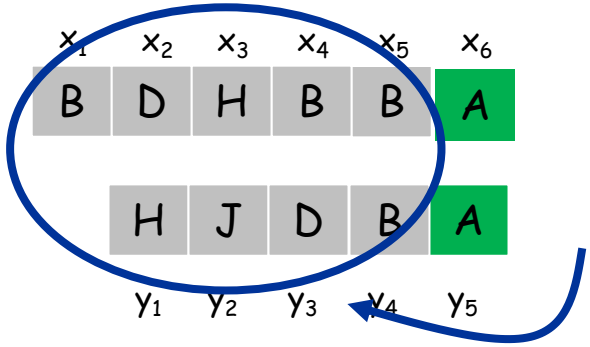


pay 1 substitution

plus the cost of transforming the rest
= OPT(i-1,j-1)

Edit Distance: Optimal substructure

Option 4: If $x_i = y_j$, just "match" x_i and y_j



just pay the cost of transforming the rest
= OPT(i-1,j-1)

Edit Distance: Optimal substructure

So we have the following recursive relationship:

If $x_i \neq y_j$

$$OPT(i,j) = \min\{ 1 + OPT(i-1,j),\ 1 + OPT(i,j-1),\ 1 + OPT(i-1,j-1) \}$$

Else ($x_i = y_j$)

$$OPT(i,j) = \min\{ 1 + OPT(i-1,j),\ 1 + OPT(i,j-1),\ OPT(i-1,j-1) \}$$

With boundary cases

$OPT(i,0) = i$ for all i
$OPT(0,j) = j$ for all j

Notice OPT(m,n) gives the edit distance between the whole strings

Edit Distance: Algorithm

```
EditDistance(x, y)
   for i = 0 to m
      M[i, 0] = i
   for j = 0 to n
      M[0, j] = j

   for i = 1 to m
      for j = 1 to n
         if x[i] != y[j]
            M[i, j] = min(1 + M[i-1,j], 1 + M[i,j-1], 1+ M[i-1,j-1])
         else
            M[i, j] = min(1 + M[i-1,j], 1 + M[i,j-1], M[i-1,j-1])

   return M[m, n]
```

Complexity: O(mn) time