Dynamic Programming

Cormen et. al. IV 15

Dynamic Programming Applications

Areas

- Bioinformatics
- Control theory
- Operations research

Some famous dynamic programming algorithms

- Unix diff for comparing two files
- Smith-Waterman for (DNA) sequence alignment
- Bellman-Ford for shortest path routing in networks

Motivating Example: Fibonacci numbers

$F(1) = F(2) = 1$

$F(n) = F(n-1) + F(n-2) \quad n>2$

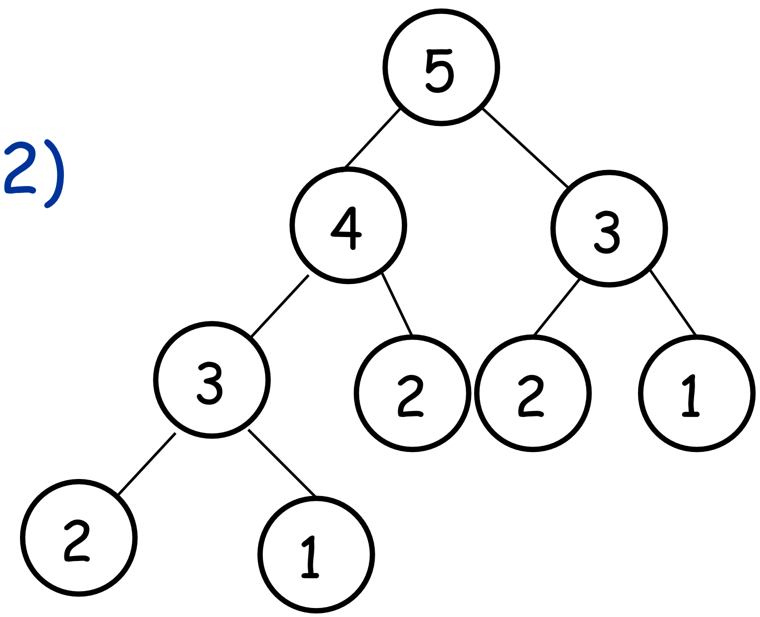
Fibonacci numbers

F(1) = F(2) = 1
F(n) = F(n-1) + F(n-2)   n>2

Simple recursive solution:

```
def fib(n) :
  if n<=2: return 1
  else: return fib(n-1) + fib(n-2)
```

What is the size of the call tree?

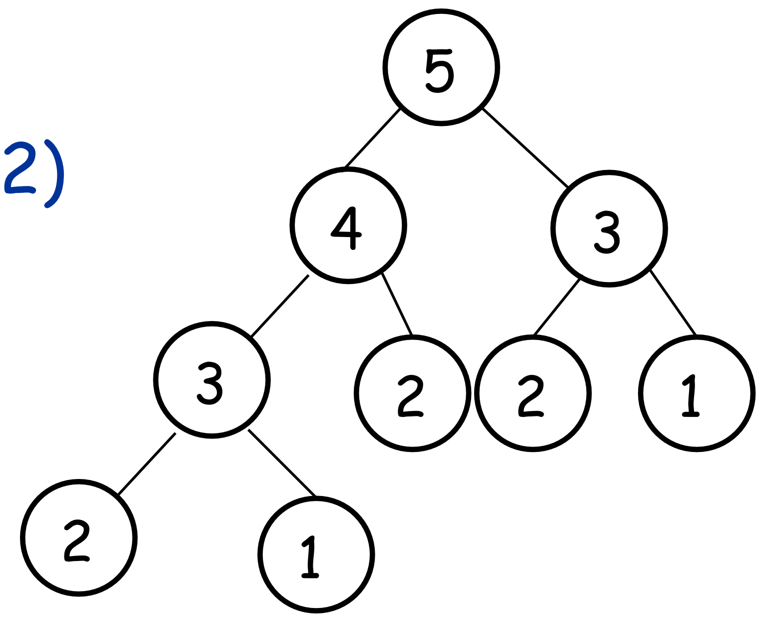


Fibonacci numbers

$F(1) = F(2) = 1$
$F(n) = F(n-1) + F(n-2) \quad n>2$

Simple recursive solution:

```
def fib(n) :
   if n<=2: return 1
   else: return fib(n-1) + fib(n-2)
```

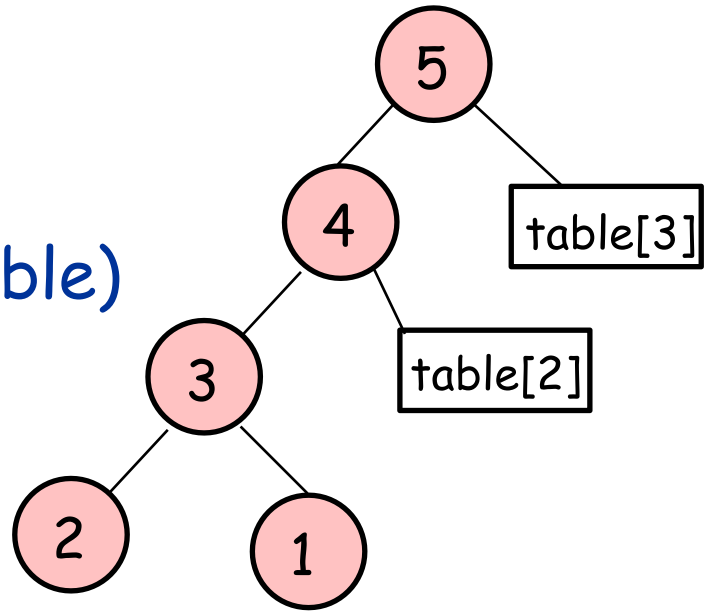


Problem: exponential call tree

Can we avoid it?

Efficient computation using a memo table

```
def fib(n, table) :
    # pre: n>0, table[i] either 0 or contains fib(i)
    if n <= 2 :
        return 1
    if table[n] > 0 :
        return table[n]
    result = fib(n-1, table) + fib(n-2, table)
    table[n] = result
    return result
```



We use a memo table, never computing the same value twice. How many calls now? O(n)

Can we do better?

Look ma, no table

```
def fib(n) :
  if n<=2 : return 1
  a,b = 1
  c = 0
  for i in range(3, n+1) :
      c = a + b
      a = b
      b = c
   return c
```

Compute the values "bottom up"
Avoid the table, only store the previous two
same O(n) time complexity, constant space.

Only keeping the values we need.

Optimization Problems

In optimization problems a set of **choices** are to be made to arrive at an optimum, and sub problems are encountered.

This often leads to a **recursive** definition of a solution. However, the recursive algorithm is often **inefficient** in that it solves the **same sub problem many times**.

Dynamic programming avoids this repetition by solving the problem **bottom up** and **storing** sub solutions, that are (still) needed.

Dynamic vs Greedy, Dynamic vs Div&Co

Compared to Greedy, there is **no predetermined local choice** of a sub solution, but a solution is chosen by computing a set of alternatives and **picking the best**.

Another way of saying this is: Greedy only needs ONE best solution.

Dynamic Programming **builds on** the recursive definition of a divide and conquer solution, but **avoids re-computation** by storing earlier computed values, thereby often saving orders of magnitude of time.

Fibonacci: from exponential to linear

Dynamic Programming

Dynamic Programming has the following steps

- Characterize the **structure** of the problem, i.e., show how a larger problem can be solved using solutions to sub-problems

- **Recursively** define the optimum

- Compute the optimum **bottom up**, **storing** values of sub solutions

- Construct the optimum from the **stored data**

Optimal substructure

Dynamic programming works when a problem has **optimal substructure**: we can construct the optimum of a larger problem from the optima of a "small set" of smaller problems.

- small: polynomial

Not all problems have optimal substructure. Travelling Salesman Problem (TSP) does not have optimal substructure.

Weighted Interval Scheduling

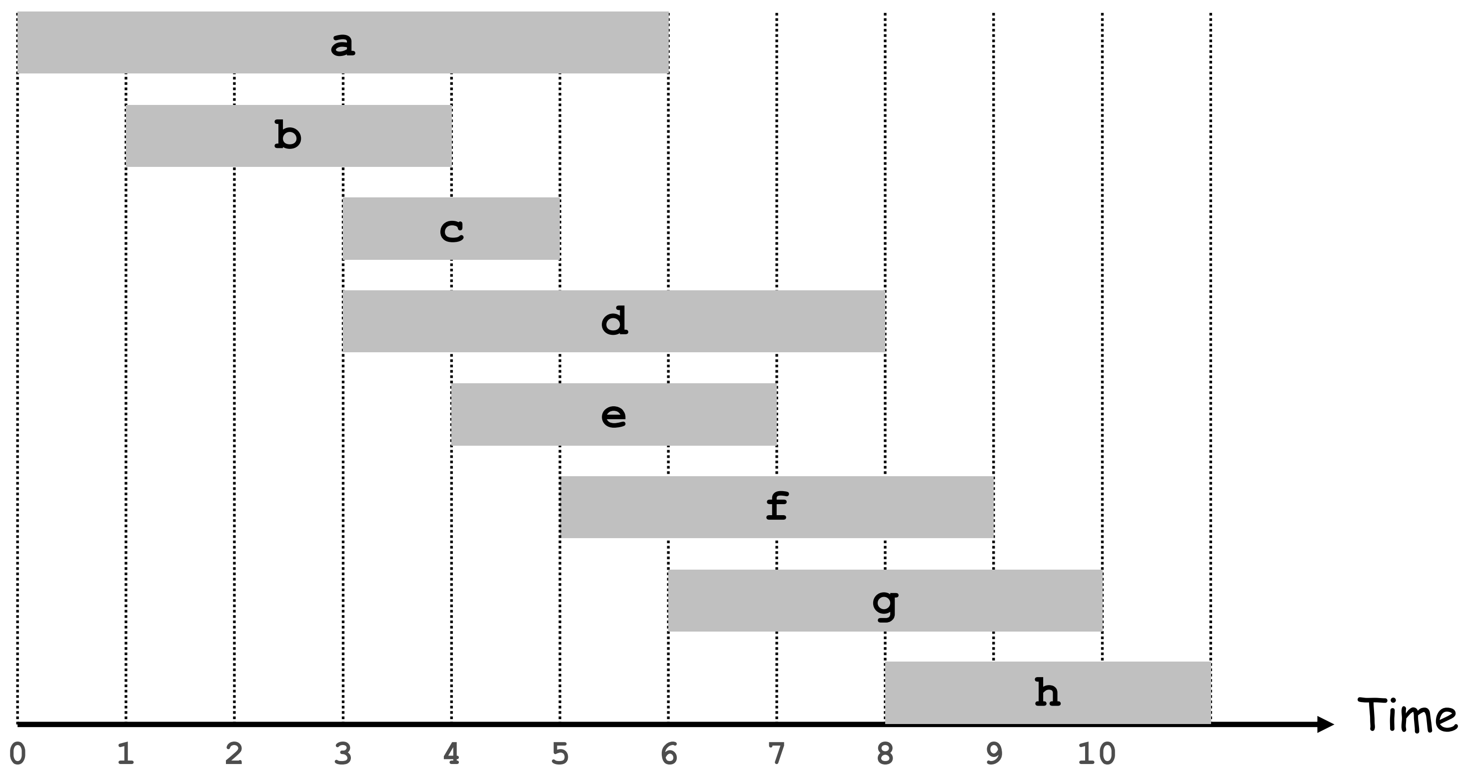
We studied a greedy solution for the interval scheduling problem, where we searched for the maximum number of compatible intervals.

If each interval has a weight and we search for the set of compatible intervals with the maximum sum of weights, no greedy solution is known.

Weighted Interval Scheduling

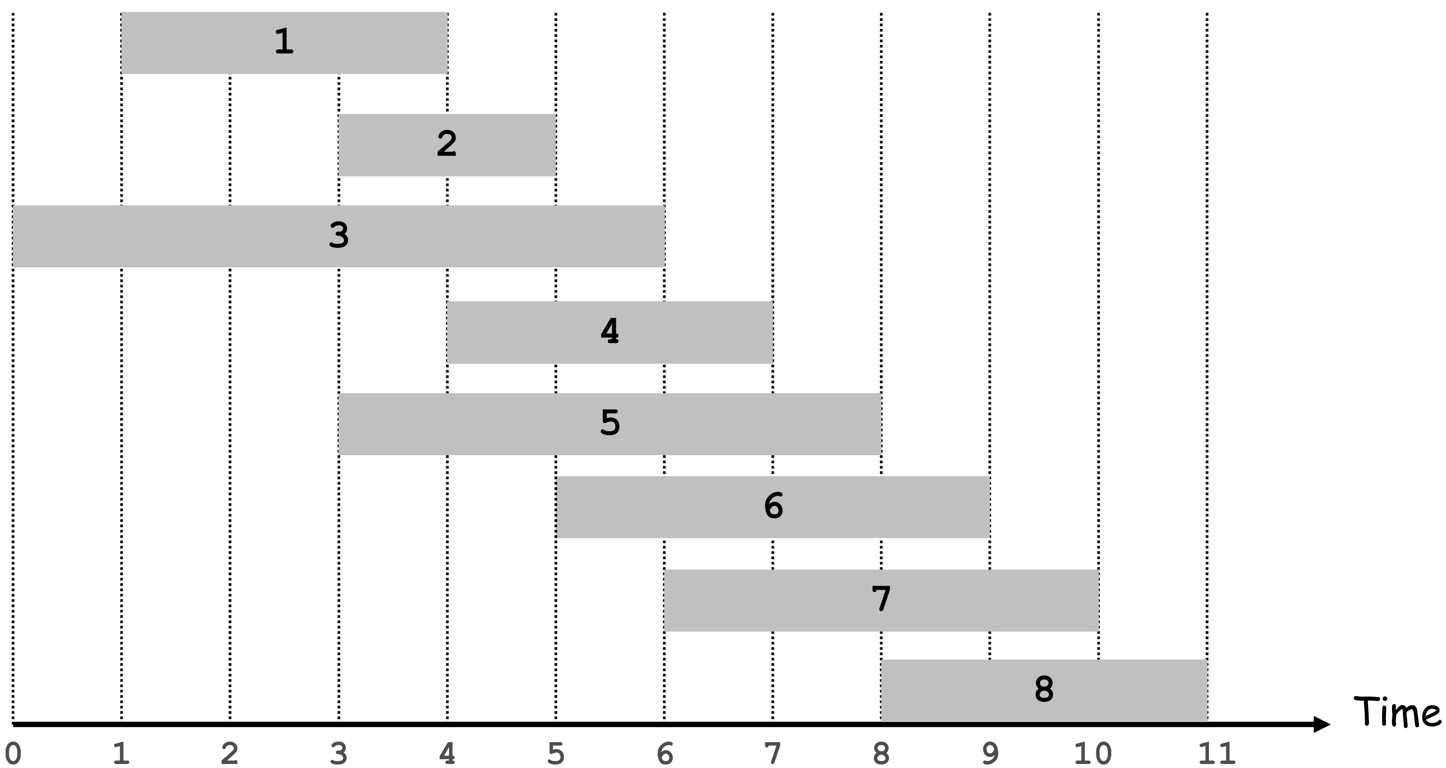
Weighted interval scheduling problem.

- 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 compatible jobs.



Weighted Interval Scheduling

Assume jobs sorted by finish time: $f_1 \leq f_2 \leq \ldots \leq f_n$.
p(j) = largest index i < j such that job i is compatible with j,
in other words: p(j) is j's **latest predecessor**; p(j) = 0 if j has no predecessors. Example: p(8) = 5, p(7) = 3, p(2) = 0.
Using p(j) can you think of a recursive solution?



Recursive (either / or) Solution

Notation. OPT(j): optimal value to the problem consisting of job requests 1, 2, ..., j.

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

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

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

Either / or recursion

This is very often a first recursive solution method:

- either some item is in and then there is some consequence
- or it is not, and then there is another consequence, e.g. knapsack, see later slides:

Here: for each job j
  either j is chosen
    - add v_j to the total value
    - consider p_j next

  or it is not
    - total value does not change
    - consider j-1 next

Weighted Interval Scheduling: Recursive Solution

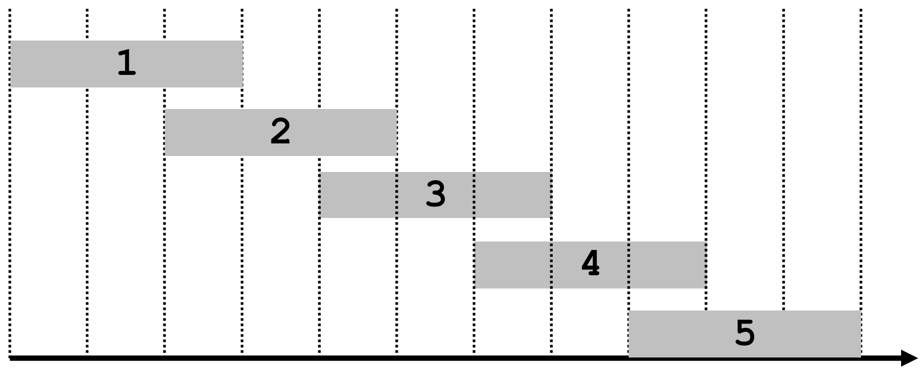
```
input: s_1,…,s_n , f_1,…,f_n , v_1,…,v_n

sort jobs by finish times so that f_1 ≤ f_2 ≤ ... ≤ f_n.

compute p(1), p(2), …, p(n)

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

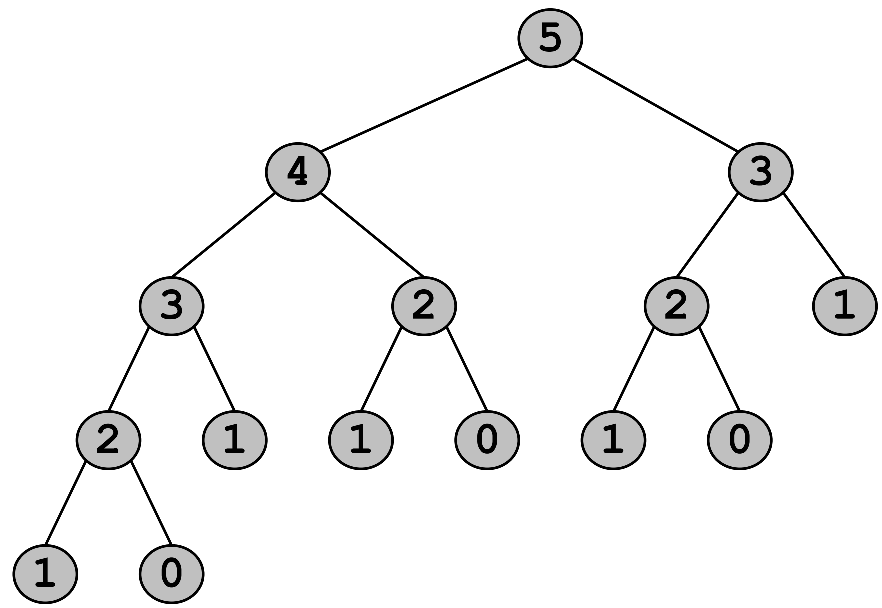
What is the size of the call tree here?
How can you make it big, e.g. exponential?

Analysis of the recursive solution

Observation. Recursive algorithm considers exponential number of (redundant) sub-problems.

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





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

Code on previous slide becomes Fibonacci: opt(j) calls opt(j-1) and opt(j-2)

Weighted Interval Scheduling: Memoization

Memoization. Store results of each sub-problem in a cache; look up as needed.

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

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

for j = 1 to n
   M[j] = empty
M[0] = 0

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

Global array

Weighted Interval Scheduling: Running Time

Claim. Memoized version of `M-Compute-Opt(n)` takes O(n log n) time.

- `M-Compute-Opt(n)` fills in all entries of M ONCE in constant time
- Since M has n+1 entries, this takes O(n)

- But we have sorted the jobs

- So Overall running time is O(n log n).

Weighted Interval Scheduling: Finding a Solution

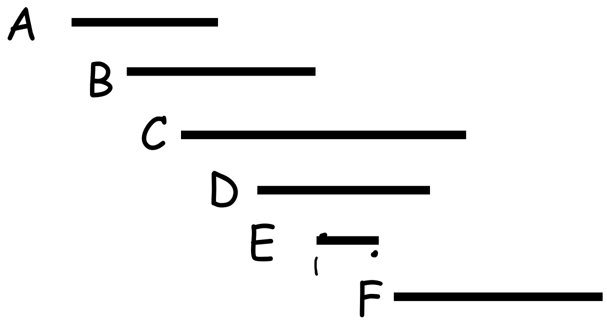
Question. Dynamic programming computes optimal value. What if we want the choice vector determining which intervals are chosen.

Answer. Do some post-processing, walking BACK through the dynamic programming table.

```
Run Dynpro-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)
}
```

Do it, do it: Recursive

	S	F	V
A	1	5	7
B	2	9	8
C	4	13	3
D	6	12	5
E	9	10	10
F	11	15	1



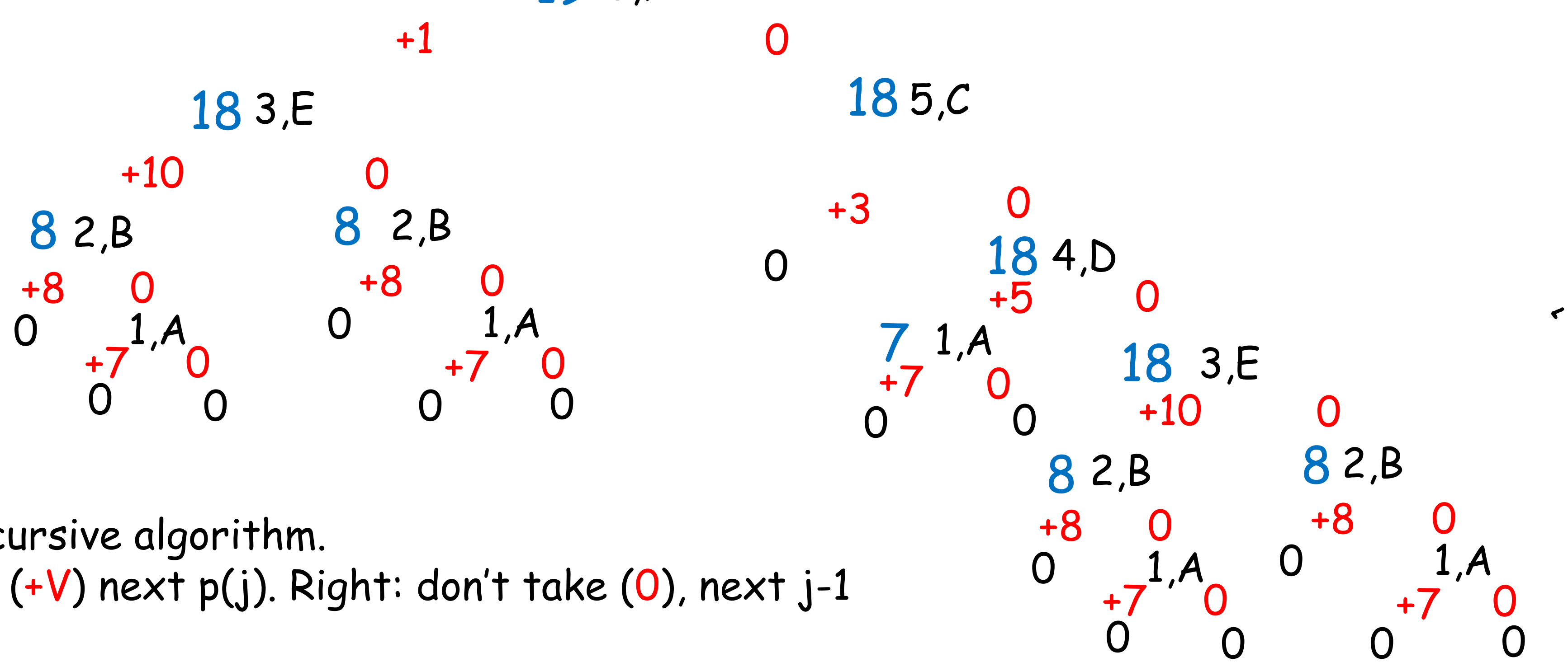
Sort in F order

1 A
2 B
3 E
4 D
5 C
6 F

Determine p array

1,A: 0
2,B: 0
3,E: 2,B
4,D: 1,A
5,C: 0
6,F: 3,E

6,F + 3,E + 2,B = 19



Do the recursive algorithm.
Left: take (+V) next p(j). Right: don't take (0), next j-1

Up: edge: add,
node: take the max

Weighted Interval Scheduling: Bottom-Up

Bottom-up **dynamic programming**, **build a table.**

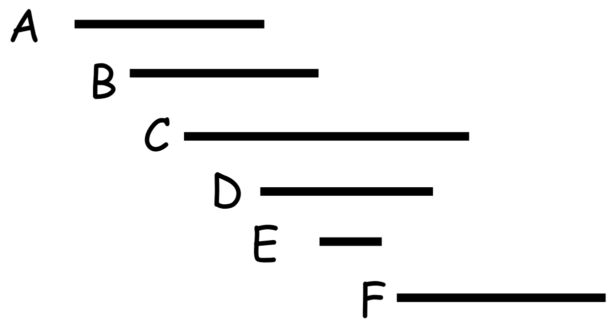
```
input: n, s1,…,sn , f1,…,fn , v1,…,vn

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

compute p(1), p(2), …, p(n)

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

By going in bottom up order M[p(j)] and M[j-1] are present when M[j] is computed. This takes O(nlogn) for sorting and O(n) for Compute, so O(nlogn)

Do it, do it: Dynamic Programming

	S	F	V
A	1	5	7
B	2	9	8
C	4	13	3
D	6	12	5
E	9	10	10
F	11	15	1

Draw Intervals



Sort in F order

1 A
2 B
3 E
4 D
5 C
6 F

Determine p array

1,A: 0
2,B: 0
3,E: 2,B
4,D: 1,A
5,C: 0
6,F: 3,E

```
M[0] = 0
   for j = 1 to n
      M[j] = max(vj + M[p(j)],
                 M[j-1])
```

Create M table

0	7	8	18	18	18	19
0	1,A	2,B	3,E	4,D	5,C	6,F

Walk back to determine choices

6,F: take gets you 19, don't gets you 18, so take F
3,E: take gets you 18, don't gets you 8, so take E
2,B: take gets you 8, don't gets you 0, so take B

Computing the p array

Claim. Memoized version of `M-Compute-Opt(n)` takes O(n log n) time.

- `M-Compute-Opt(n)` fills in all entries of M ONCE in constant time
- Since M has n+1 entries, this takes O(n)

- But we have sorted the jobs

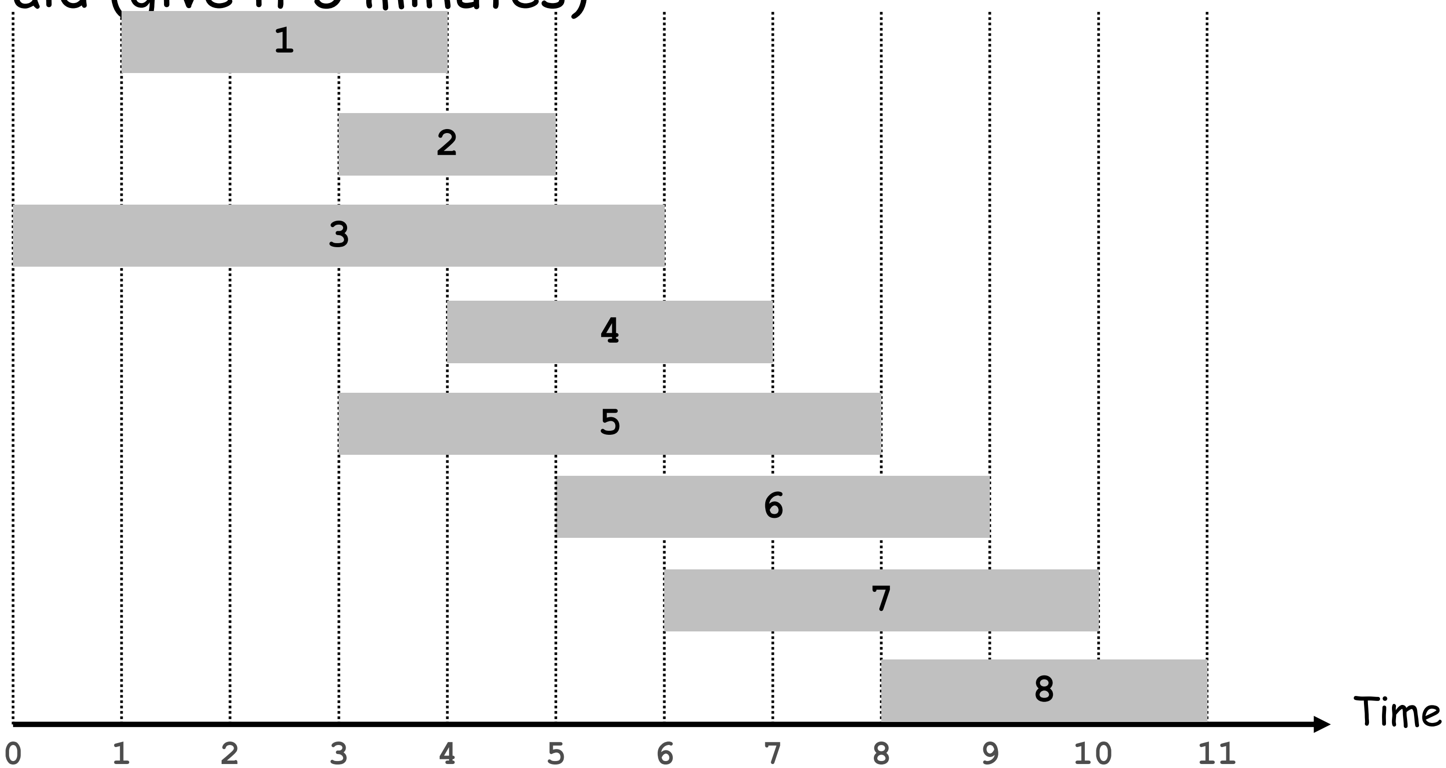
- So Overall running time is O(n log n).

Computing the latest-predecessor array

Visually, it is “easy” to determine p(i), the largest index i < j such that job i is compatible with j. For the example below:

p[1…8] = [0, 0, 0, 1, 0, 2, 3, 5]

How about an algorithm? Or even as a human, try it without the visual aid (give it 5 minutes)



Computing the latest-predecessor array

Visually, it is "easy" to determine p(i), the largest index i < j such that job i is compatible with j. For the example below:

p[1…8] = [0, 0, 0, 1, 0, 2, 3, 5]

How about an algorithm? Or even as a human, try it without the visual aid (give it 5 minutes)

Activity	A1	A2	A3	A4	A5	A6	A7	A8
Start (s)	1	3	0	4	3	5	6	8
Finish (f)	4	5	6	7	8	9	10	11
p								

Time

Computing the latest-predecessor array

Spoiler alert:

1. Treat all the start and finish times as "events" and sort them in increasing order (resolve ties any way, as long as all the f events are before the s events)
2. Have global variables LFSF and ILFSF (for "Latest_Finish_So_Far," and "Index_of_LFSF")
3. Process events in order as follows:
   a. If it is a finish event, f_i then update LFSF and ILFSF
   b. If it is a start event, s_i then set p(i) to ILFSF

Evnt	LFSF	ILFSF	p(x)=y
s3	0	0	p(3)=0
s1	0	0	p(1)=0
s2	0	0	p(2)=0
s5	0	0	p(5)=0
f1	4	1	
s4	4	1	p(4)=1
f2	5	2	
s6	5	2	p(6)=2
f3	6	3	
s7	6	3	p(7)=3
f4	7	4	
f5	8	5	
s8	8	5	p(8)=5
f6	9	6	
f7	10	7	
f8	11	8	

Discrete Optimization Problems

Discrete Optimization Problem (S,f)

- S:
  - Set of solutions of a problem, satisfying some constraint
- $f : S \rightarrow R$
  - Cost function associated with feasible solutions
- Objective: find an optimal solution x_{opt} such that

  $f(x_{opt}) \leq f(x)$ for all x in S (minimization)

  or $f(x_{opt}) \geq f(x)$ for all x in S (maximization)

- Ubiquitous in many application domains
  - planning and scheduling
  - VLSI layout
  - pattern recognition
  - bio-informatics

Knapsack Problem

Knapsack problem.

- Given n objects and a "knapsack" of capacity W
- Item i has a weight $w_i > 0$ and value or profit $v_i > 0$.
- Goal: fill knapsack so as to maximize total value.

What would be a Greedy solution?

repeatedly add item with maximum v_i / w_i ratio …

Does Greedy work?

Capacity W = 7, Number of objects n = 3
w = [5, 4, 3]
v = [10, 7, 5] (ordered by v_i / w_i ratio)

Either / or Recursion for Knapsack Problem

Notation: OPT(i, w) = optimal value of max weight subset that uses items 1, …, i with weight limit w.

- Case 1: item i is not included:
  - OPT includes best of { 1, 2, …, i-1 } using weight limit w
- Case 2: item i is included, if it can be included: w_i <= w
  - new weight limit = w – w_i
  - OPT includes best of { 1, 2, …, i–1 } using weight limit w-w_i

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

Knapsack Problem: Dynamic Programming

Knapsack. Fill an n+1 by W+1 array.

```
Input: n, W, weights w1,…,wn,
             values v1,…,vn

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

for i = 1 to n
 for w = 0 to W
  if wi > w :
     M[i, w] = M[i-1, w]
  else :
     M[i, w] = max (M[i-1, w],
                    vi + M[i-1, w-wi ])
return M[n, W]
```

Do it for:

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

W = 11

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 }												
{ 1, 2 }												
{ 1, 2, 3 }												
{ 1, 2, 3, 4 }												
{ 1, 2, 3, 4, 5 }												

W = 11

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

Knapsack Algorithm

W + 1 →

n + 1 ↓

	0	1	2	3	4	5	6	7	8	9	10	11
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 }												
{ 1, 2, 3 }												
{ 1, 2, 3, 4 }												
{ 1, 2, 3, 4, 5 }												

At 1,1 we can fit item 1 and from then on, all we have is item 1

W = 11

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

Knapsack Algorithm

W + 1 →

n + 1 ↓

	0	1	2	3	4	5	6	7	8	9	10	11
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 }												
{ 1, 2, 3, 4 }												
{ 1, 2, 3, 4, 5 }												

At 2,2 we can either not take item 2 (value 1 (previous row[2]) or we can take item 2 (value 6 previous row[0]+ 6)

At 2,3 we can either not take item 2 (value 1) or we can take item 2 and item 1 (value 7).

From then on we can fit both items 1 and 2 (value 7)

W = 11

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

Knapsack Algorithm

W + 1 →

n + 1 ↓

	0	1	2	3	4	5	6	7	8	9	10	11
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 }												
{ 1, 2, 3, 4, 5 }												

From 3,0 to 3,4 we cannot take item 3.
At 3,5 we can either not take item 3 (value 7) or we can take item 3 (value 18)
At **3,6** we can either not take item 3 (value 7) or we can take item 3 (value 19),
etc.,.

W = 11

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

Knapsack Problem: Dynamic Programming

Knapsack. Find the set of items in the solution.

```
Input: n, W, M, weights w1,…,wn,
              values v1,…,vn

for i = 1 to n:
  S[i] = 0

j = W

for i = n downto 1
  if  M[i, j] > M[i-1, j]] then:
     S[i] = 1
     j -= w[i]

return S
```

Do it for:

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

$W = 11$

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: 40

How do we find the objects in the optimum solution?

Walk back through the table!!

W = 11

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

Knapsack Algorithm

W + 1 →

n + 1 ↓

	0	1	2	3	4	5	6	7	8	9	10	11
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: 40

n=5 Don't take object 5 (7+28=35 < 40)

W = 11

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

Knapsack Algorithm

W + 1 →

n + 1 ↓

	0	1	2	3	4	5	6	7	8	9	10	11
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: 40
n=5 Don't take object 5
n=4 Take object 4 (18+22=40>25)

W = 11

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

Knapsack Algorithm

W + 1 →

n + 1 ↓

	0	1	2	3	4	5	6	7	8	9	10	11
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: 40
n=5 Don't take object 5
n=4 Take object 4
n=3 Take object 3

and now we cannot take anymore,
so choice set is {3,4},
choice vector is [0,0,1,1,0]

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. $\Theta(nW)$.

- Not polynomial in input size!
  - W can be exponential in n

- Decision version of Knapsack is NP-complete. [Chapter 34 CLRS]

Knapsack approximation algorithm.

- There exists a poly-time algorithm that produces a feasible solution that has value within 0.01% of optimum.