Dynamic Programming:

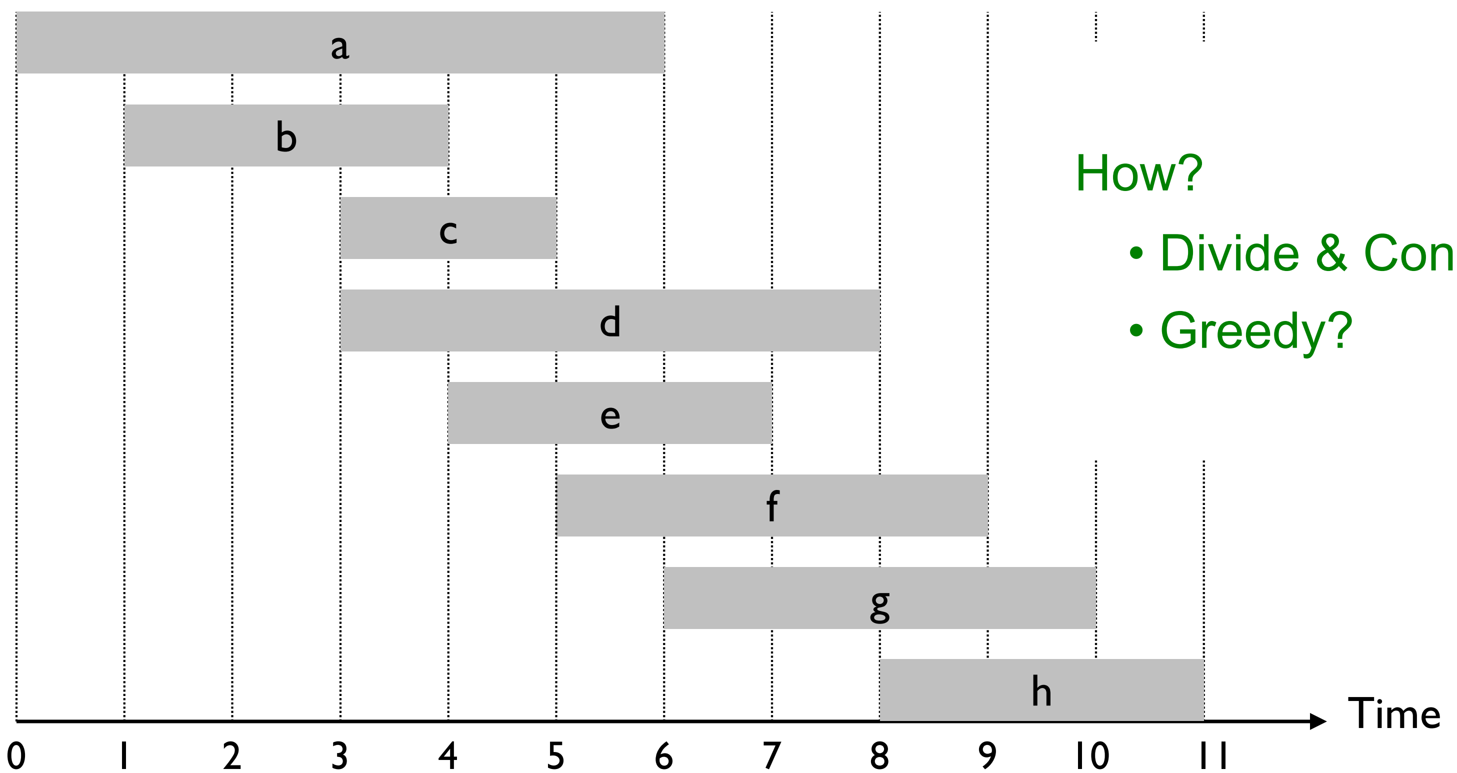
Interval Scheduling and Knapsack

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.



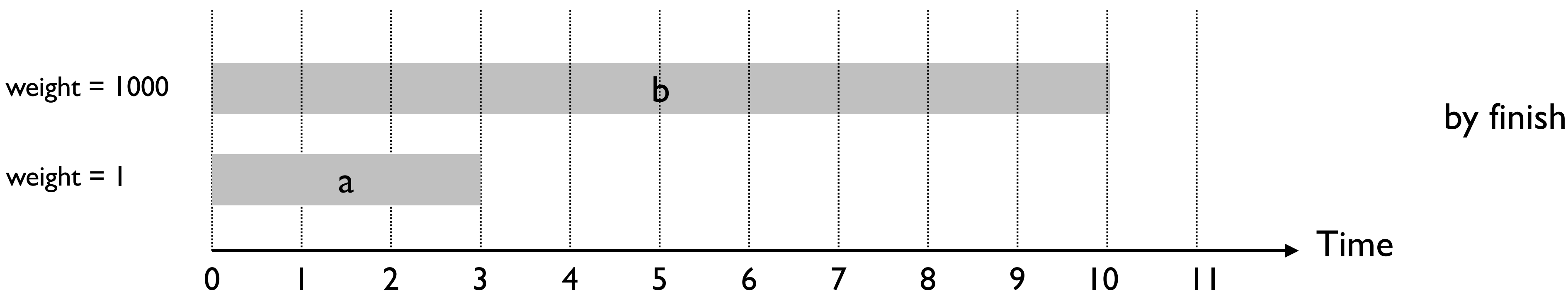
How?

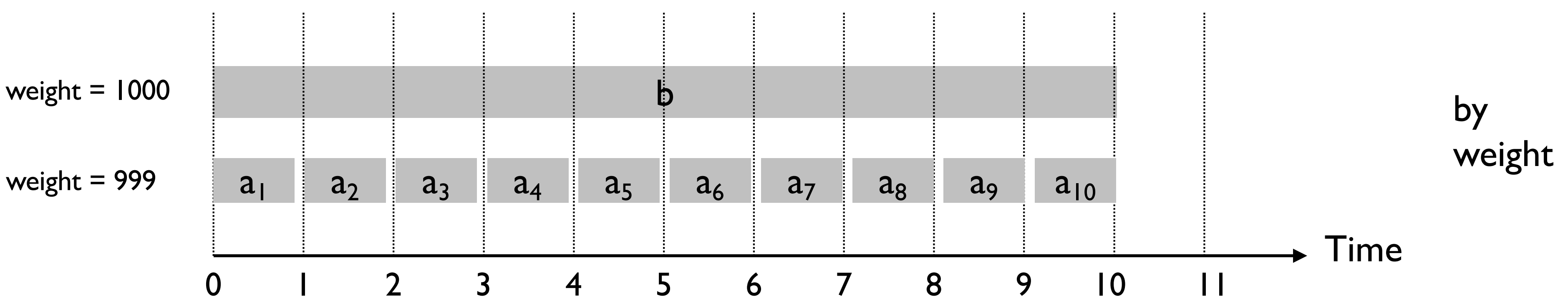
- Divide & Conquer?
- Greedy?

Unweighted Interval Scheduling Review

Recall. Greedy algorithm works if all weights are 1.
- Consider jobs in ascending order of finish time.
- Keep job if compatible with previously chosen jobs.

Observation. Greedy fails spectacularly with arbitrary weights.





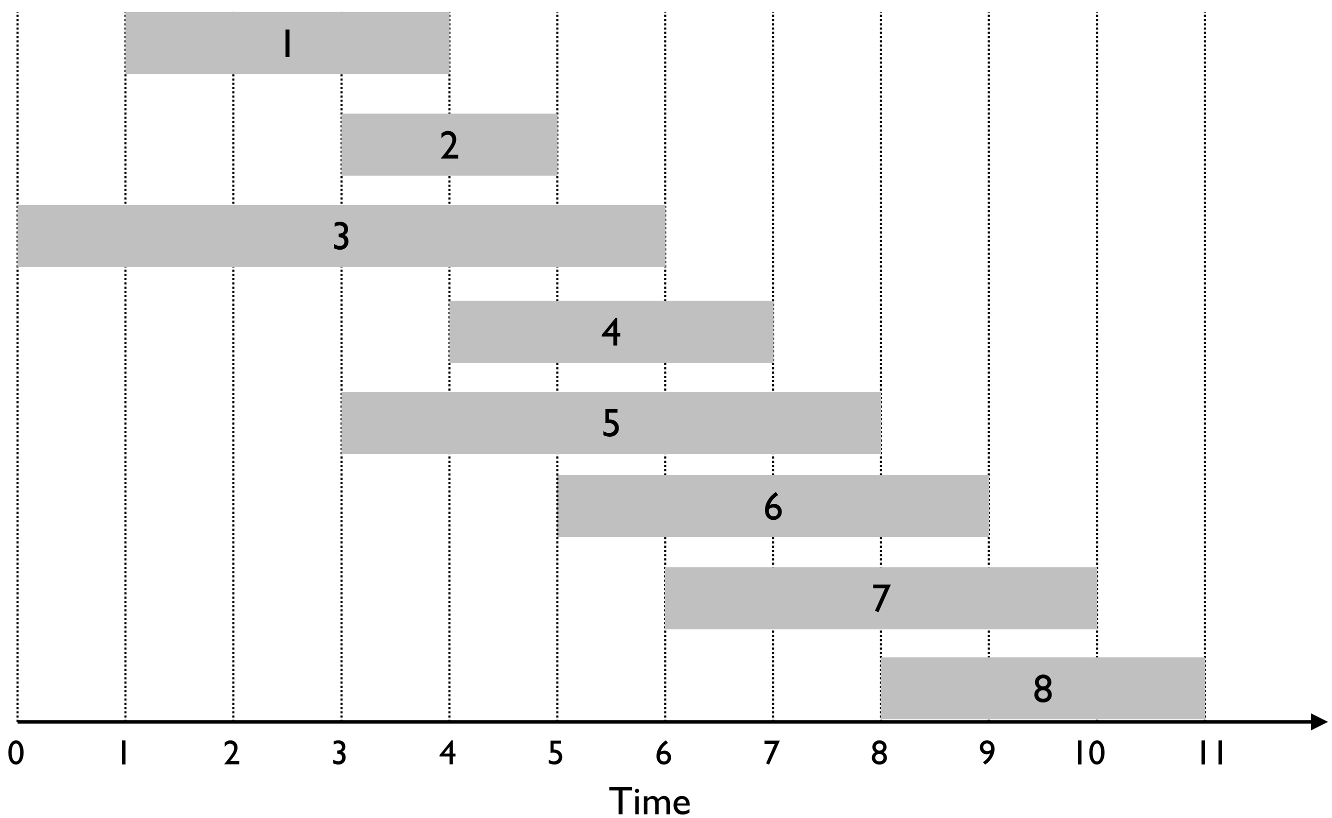
Exercises: by "density" = weight per unit time? Other ideas?

Weighted Interval Scheduling

Notation. Label jobs by finishing time: $f_1 \leq f_2 \leq \ldots \leq f_n$.
Def. $p(j)$ = largest $i < j$ such that job i is compatible with j.

"p" suggesting (last possible) "predecessor"

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, ..., j.

key idea: binary choice

- Case 1: Optimum selects job j.
  - 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)

principle of optimality

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

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

Weighted Interval Scheduling: Brute Force Recursion

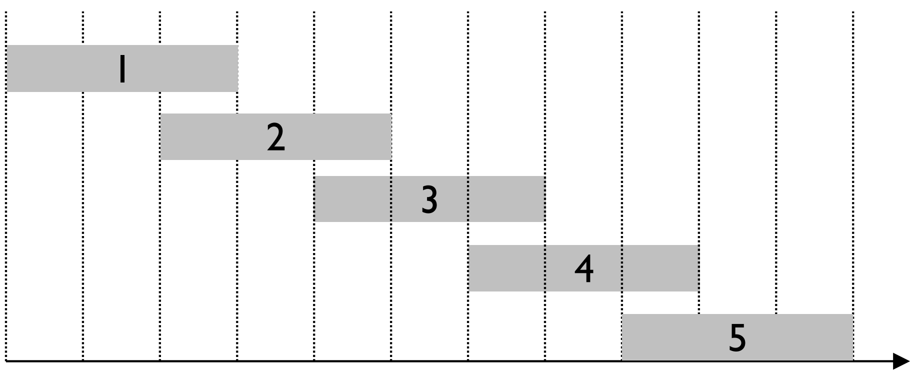
Brute force recursive algorithm.

```
Input: n, 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))
}
```

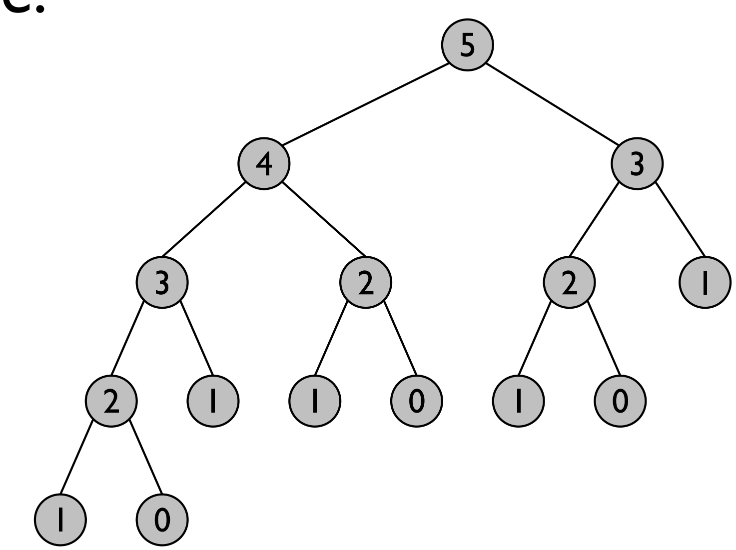
Weighted Interval Scheduling: Brute Force

Observation. Recursive algorithm is correct, but spectacularly slow because of redundant sub-problems $\Rightarrow$ exponential time.

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



p(1) = p(2) = 0; p(j) = j-2, j ≥ 3



Weighted Interval Scheduling: Bottom-Up

Bottom-up dynamic programming. Unwind recursion.

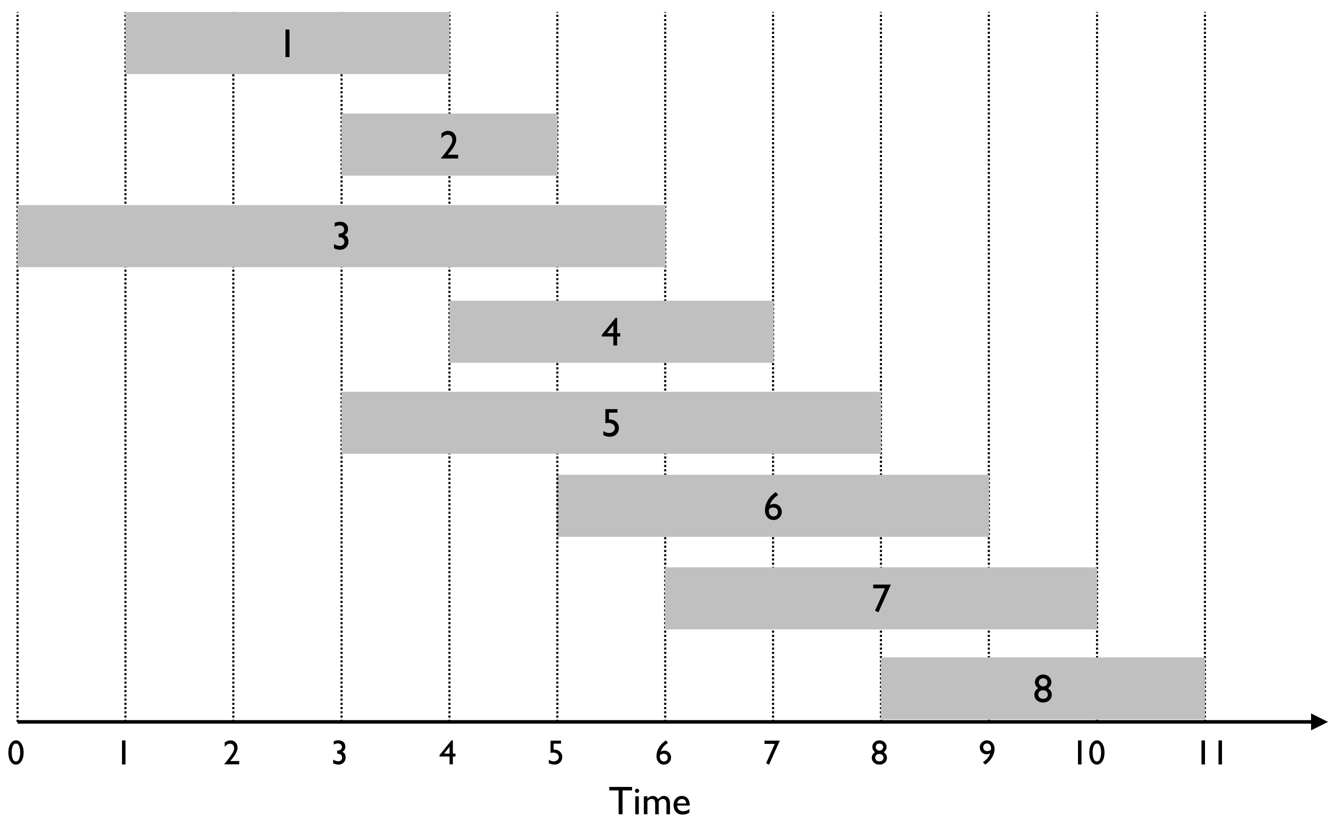
```
Input: n, s1,…,sn , f1,…,fn , v1,…,vn

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

Compute p(1), p(2), …, p(n)

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

Output OPT[n]
```

Claim: OPT[j] is value of optimal solution for jobs 1..j
Timing: Loop is O(n); sort is O(n log n); what about p(j)?

Weighted Interval Scheduling

Notation. Label jobs by finishing time: $f_1 \leq f_2 \leq \ldots \leq f_n$.
Def. p(j) = largest i < j such that job i is compatible with j.

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

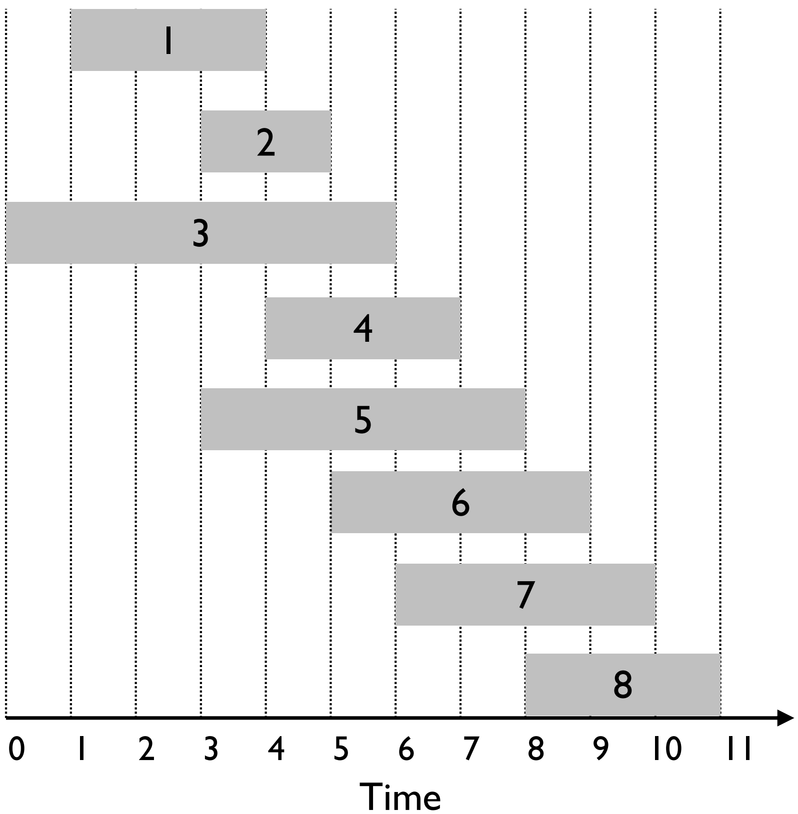


j	vj	pj	optj
0	-	-	0
1		0	
2		0	
3		0	
4		1	
5		0	
6		2	
7		3	
8		5	

Weighted Interval Scheduling Example

Label jobs by finishing time: $f_1 \leq f_2 \leq \ldots \leq f_n$.
p(j) = largest i < j s.t. job i is compatible with j.

Exercise: try other concrete examples:
If all vj=1: greedy by finish time ➛ 1,4,8
what if v2 > v1?, but < v1+v4?
v2>v1+v4, but v2+v6 < v1+v7, say? etc.



j	pj	vj	max(v_j+opt[p(j)], opt[j-1]) =	opt[j]
0	-	-		0
1	0	2	max(2+0, 0) =	2
2	0	3	max(3+0, 2) =	3
3	0	1	max(1+0, 3) =	3
4	1	6	max(6+2, 3) =	8
5	0	9	max(9+0, 8) =	9
6	2	7	max(7+3, 9) =	10
7	3	2	max(2+3, 10) =	10
8	5	?	max(?+9, 10) =	?

Exercise: What values of v8 cause it to be in/ex-cluded from opt?

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 – "traceback"

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

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

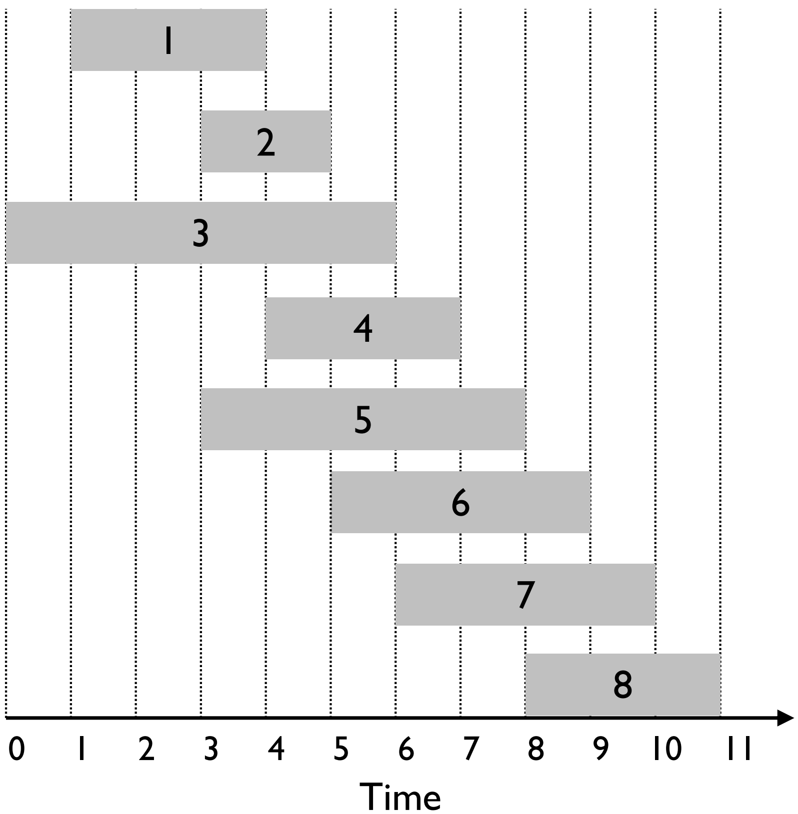
the condition determining the max when computing OPT[ ] (points to the `else if` condition)

the relevant sub-problem (points to `Find-Solution(p(j))` and `Find-Solution(j-1)`)

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

Weighted Interval Scheduling Example

Label jobs by finishing time: $f_1 \leq f_2 \leq . . . \leq f_n$.
p(j) = largest i < j s.t. job i is compatible with j.

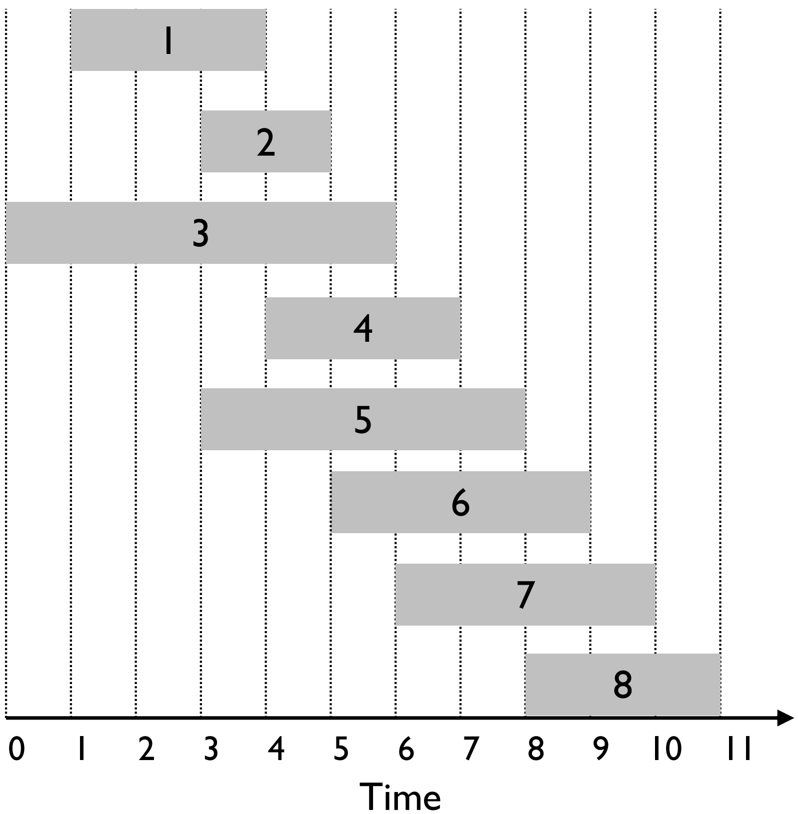


j	pj	vj	max(v_j+opt[p(j)], opt[j-1]) =	opt[j]
0	-	-	-	0
1	0	2	max(2+0, 0) =	2
2	0	3	max(3+0, 2) =	3
3	0	1	max(1+0, 3) =	3
4	1	6	max(6+2, 3) =	8
5	0	9	max(9+0, 8) =	9
6	2	7	max(7+3, 9) =	10
7	3	2	max(2+3, 10) =	10
8	5	2	max(2+9, 10) =	11

V8 = 2 is *in*cluded; opt solution is v8+v5

Weighted Interval Scheduling Example

Label jobs by finishing time: $f_1 \leq f_2 \leq \ldots \leq f_n$.
p(j) = largest i < j s.t. job i is compatible with j.



j	pj	vj	max(v_j+opt[p(j)], opt[j-1]) =	opt[j]
0	-	-	-	0
1	0	2	max(2+0, 0) =	2
2	0	3	max(3+0, 2) =	3
3	0	1	max(1+0, 3) =	3
4	1	6	max(6+2, 3) =	8
5	0	9	max(9+0, 8) =	9
6	2	7	max(7+3, 9) =	10
7	3	2	max(2+3, 10) =	10
8	5	.1	max(0.1+9, 10) =	10

V8 = 0.1 is *excluded*; opt solution is v6+v2

Sidebar: why does job ordering matter?

It' s *Not* for the same reason as in the greedy algorithm for unweighted interval scheduling.

Instead, it's because it allows us to consider only a small number of subproblems (O(n)), vs the exponential number that seem to be needed if the jobs aren' t ordered (seemingly, *any* of the 2^n possible subsets might be relevant)

Don' t believe me? Think about the analogous problem for weighted *rectangles* instead of intervals… (I.e., pick max weight non-overlapping subset of a set of axis-parallel rectangles.) Same problem for squares or circles also appears difficult.

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: maximize total value without overfilling knapsack

Ex: { 3, 4 } has value 40.

W = 11

Item	Value	Weight	V/W
1	1	1	1
2	6	2	3
3	18	5	3.60
4	22	6	3.66
5	28	7	4

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

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

[NB greedy is optimal for "fractional knapsack": take #5 + 4/6 of #4]

Dynamic Programming: False Start

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

- Case 1: OPT does not select item i.
  - OPT selects best of { 1, 2, …, i-1 }

binary choice

- 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.

Still Using Binary Choice

- Case 1: OPT does not select item i.
  - OPT selects best of { 1, 2, …, i-1 } using weight limit w

Still principle of optimality

- Case 2: OPT selects item i.
  - new weight limit = w – w_i
  - OPT selects best of { 1, 2, …, i–1 } using 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

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

```
Input: n, w1,…,wn, v1,…,vn

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

for i = 1 to n
   for w = 1 to W
      if (wi > w)
         OPT[i, w] = OPT[i-1, w]
      else
         OPT[i, w] = max {OPT[i-1, w], vi + OPT[i-1, w-wi]}

return OPT[n, W]
```

(Correctness: prove it by induction on i & 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	35	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 (w_i > w)
  OPT[i, w] = OPT[i-1, w]
else
  OPT[i, w] = max{OPT[i-1,w],v_i+OPT[i-1,w-w_i]}
```

Knapsack Problem: Running Time

Running time. Θ(n W).

- If W is "small' this is fine, but in worst case…
- Not polynomial in input size! ("W" takes only $\log_2 W$ bits)
- Called "Pseudo-polynomial"
- Knapsack is NP-hard. [Chapter 8]

Knapsack approximation algorithm [Section 11.8].

Good News: There exists a polynomial time algorithm that produces a feasible solution (i.e., satisfies weight-limit constraint) that has value within 0.01% (or any other desired factor ε) of optimum.

Bad News: as ε goes down, polynomial goes up.