CS 381

Divide-and-conquer

Divide a problem into a fixed number of independent subproblems, solve each subproblem recursively independently, and combine solutions to the subproblems into a solution.

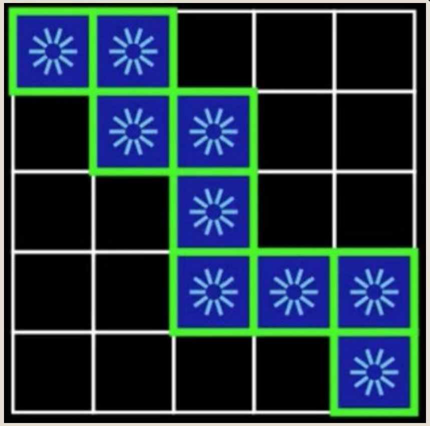
Dynamic programming (DP)

Break a problem into a series of overlapping subproblems, and build up solutions to larger and larger subproblems.

- Solves optimization problems by combining optimum solutions for subproblems.
- A subproblem is only computed once.

Why is it called Dynamic Programming?

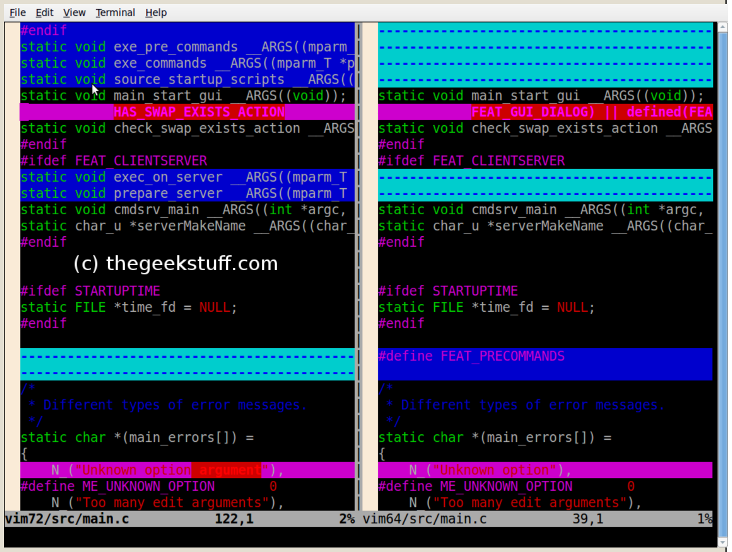
- Originally used by Richard Bellman in the 1950's.
- The name was chosen to make the technique less mathematical sounding
  - Bellman felt this was better for getting funding for the work
- Bellman-Ford DP algorithm for single source shortest paths problem



About Dynamic Programming

- DP computes the results to many optimal subsolutions
  - Computed results are stored in a table (entries are never recomputed)
  - A DP algorithm does not know which subsolutions will be used in the optimum solution
- DP algorithm is based on a recursive formulation of optimality

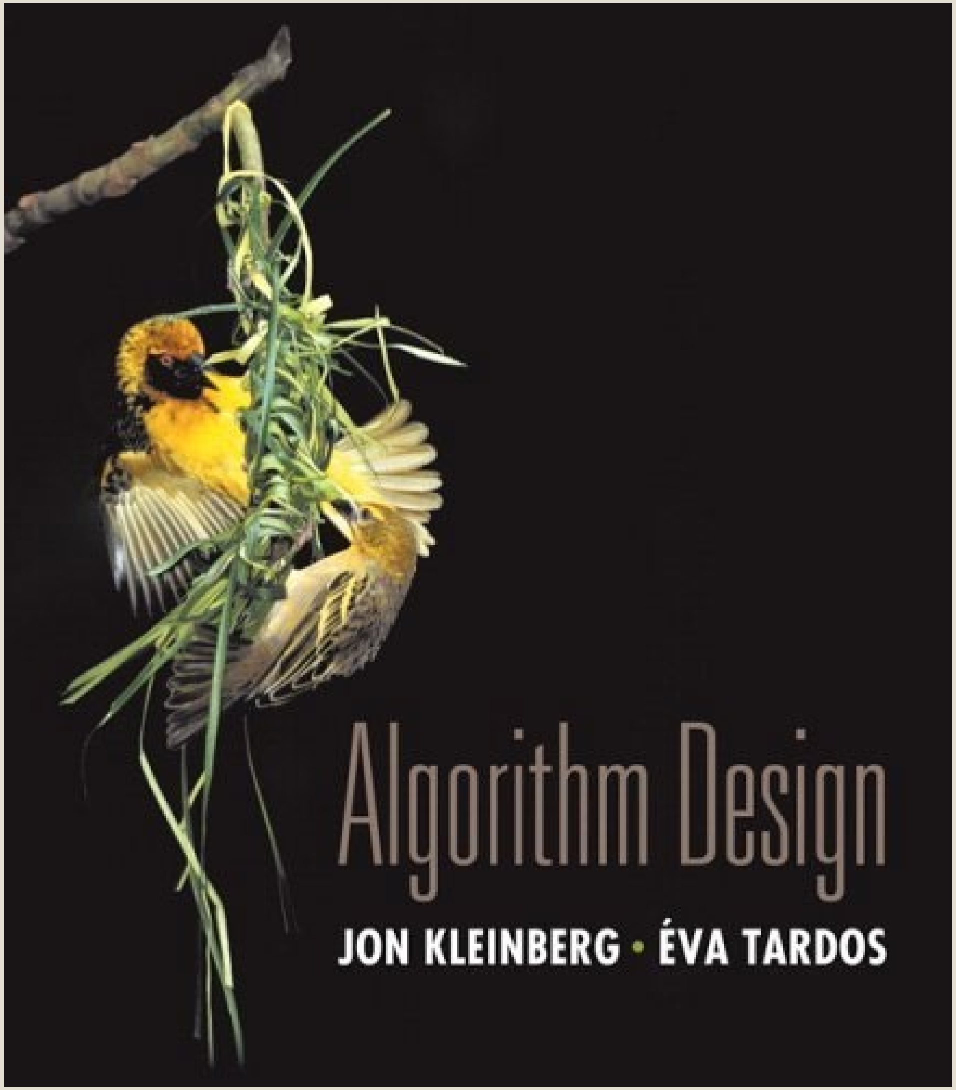
Famous dynamic programming algorithms



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

Steps for designing a DP algorithm

1. Characterize the structure of an optimal solution
2. *Recursively define the value of an optimal solution in terms of optimum subsolutions (Need to decide on the parameters of the recurrence)*
3. Compute the subsolution entries (never re-compute).
4. Construct an optimal solution from the computed entries and other information.



Chapter 6

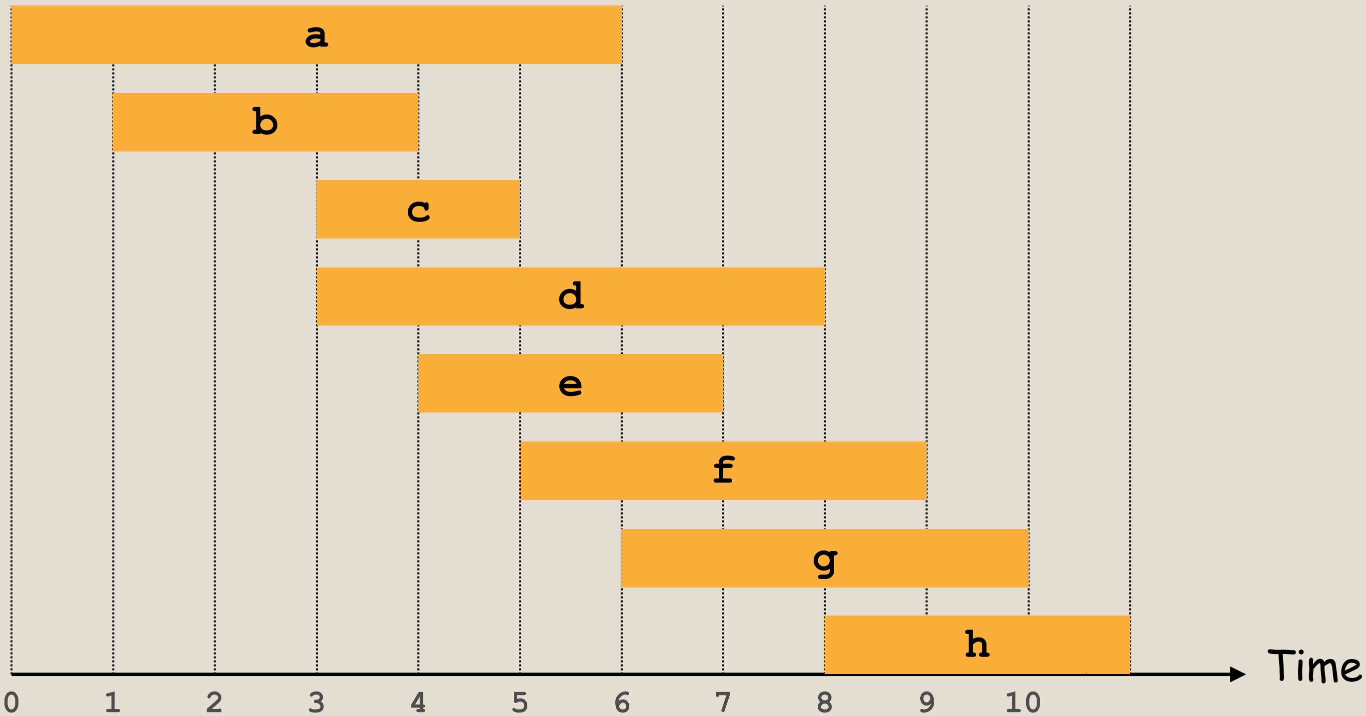
Dynamic Programming

Weighted Interval Selection (adapted)

Slides by Kevin Wayne.

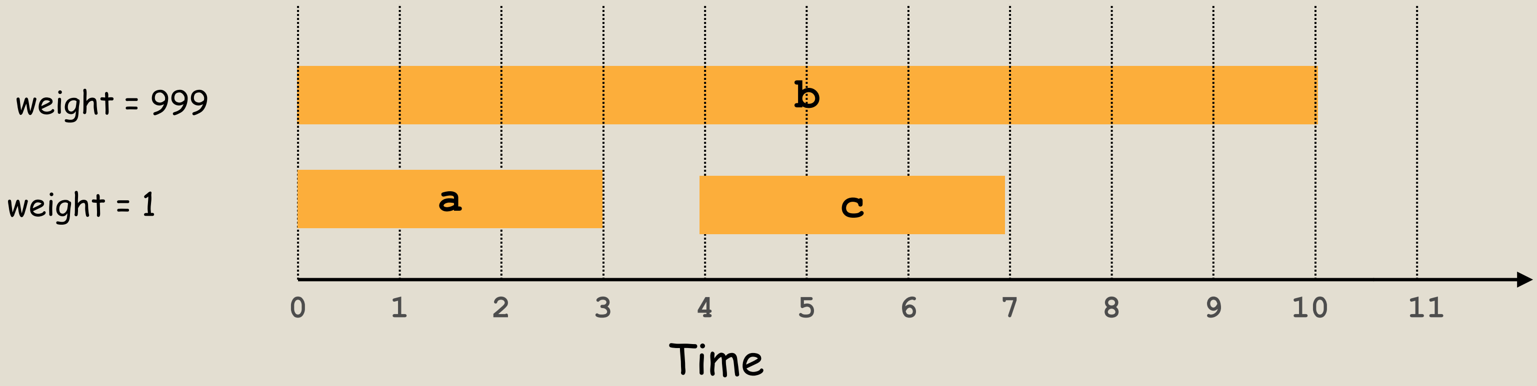
Weighted Interval Scheduling

- Jobs $\{1, \ldots, n\}$. Each job j has
  - start time s_j
  - finish time f_j
  - value (weight) w_j.
- Two jobs are ***compatible*** iff they don't overlap.
- ***Goal:*** find maximum ***weight*** subset of mutually compatible jobs.



Unweighted Interval Scheduling

- Unweighted: each weight is 1
- Goals: maximize the ***number*** of intervals selected
- Greedy algorithm fails when intervals have weight

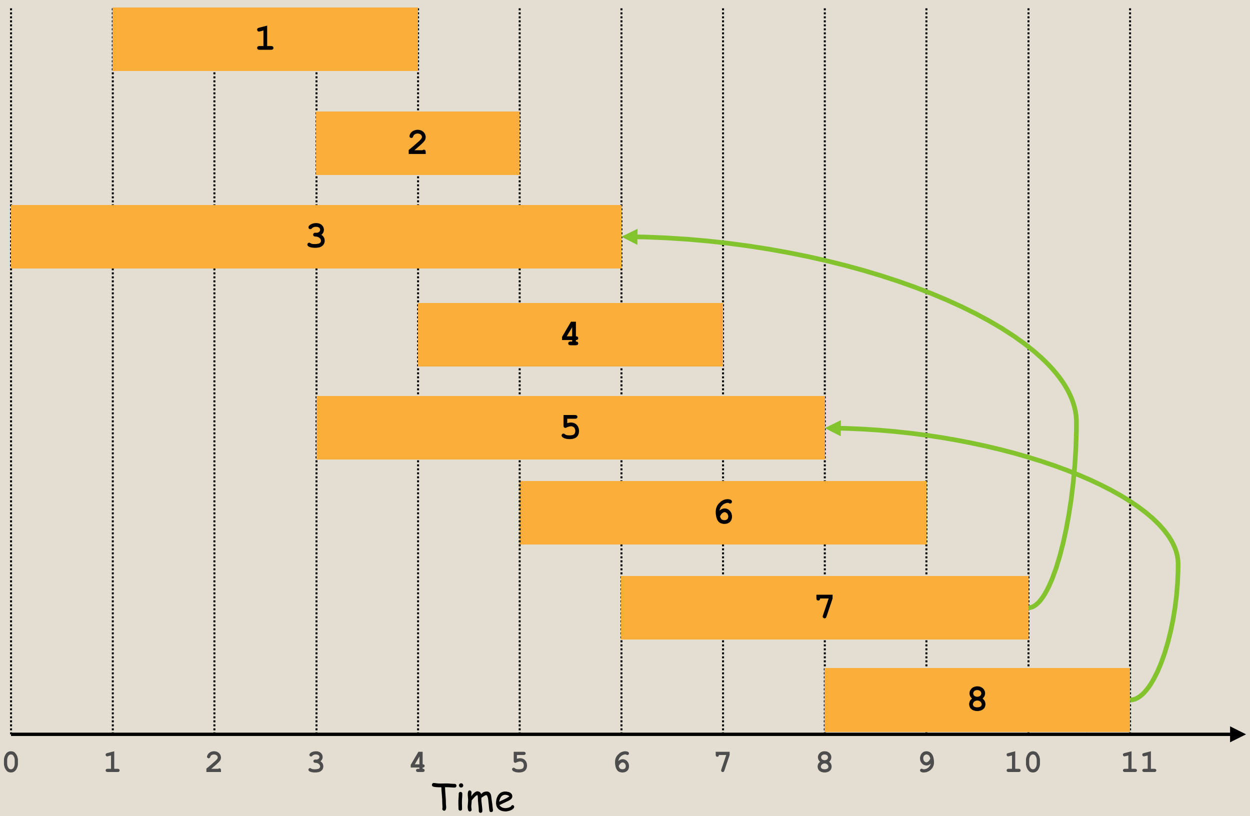


Weighted Interval Scheduling

Label jobs by finishing time: $f_1 \le f_2 \le \ldots \le f_n$ (needs sorting)

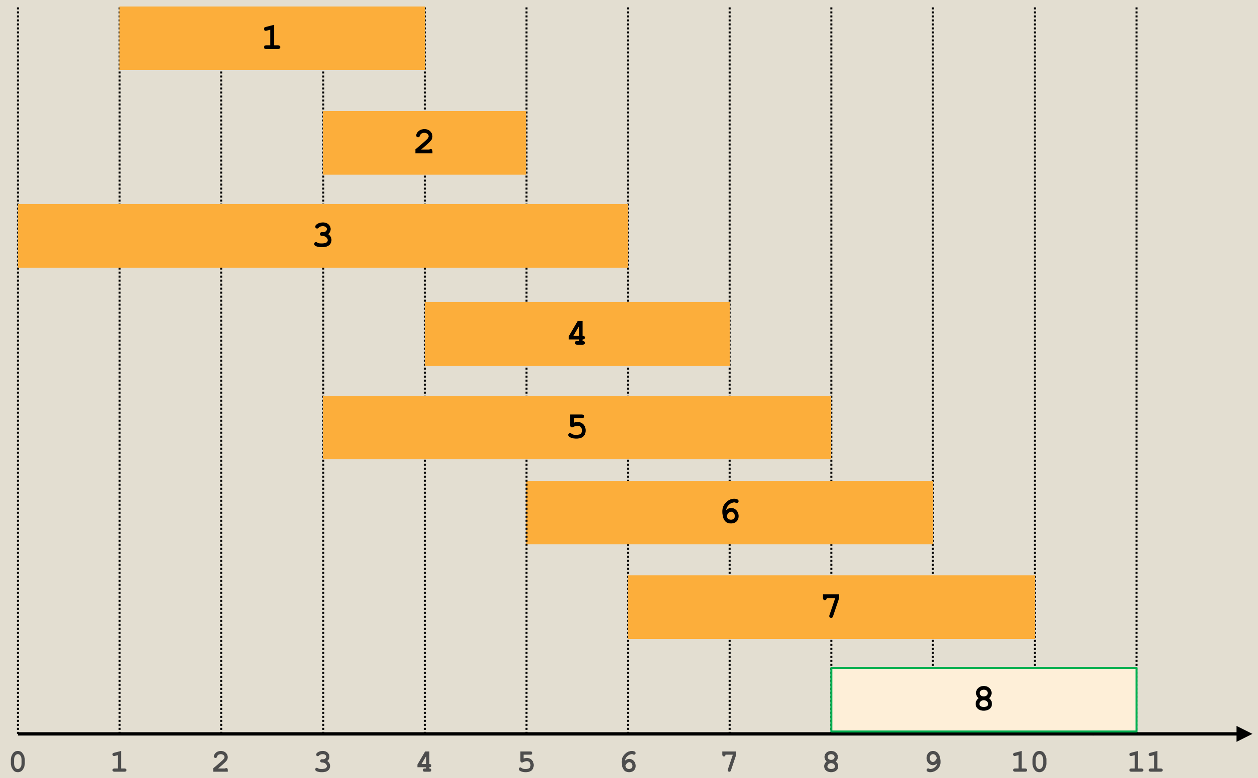
Definition: $p(j) := $ largest index $i < j$ such that job i is compatible with j.

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



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

Compute the p-entries: O(n log n) time



$s_8 = 8$

$f_8 = 11$

Use sorted f-values to Binary Search for largest f-value ≤ 8

f-value belongs to interval 5; hence, p(8) =5

Dynamic Programming: Binary Choice is common

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

Case 1: OPT selects job n.

Dynamic Programming: Binary Choice is common

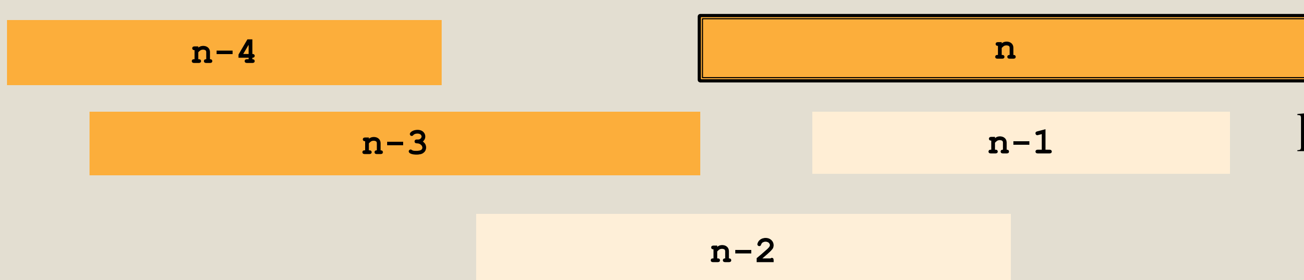
OPT(n) = value of optimal solution to the problem consisting of

jobs 1, 2, …, n.

Case 1: OPT selects job n.

- collect profit w_n
- eliminate incompatible jobs $\{ p(n) + 1, p(n) + 2, \ldots, n - 1 \}$
- must include optimal solution to problem consisting of remaining compatible

  jobs 1, 2, …, p(n)

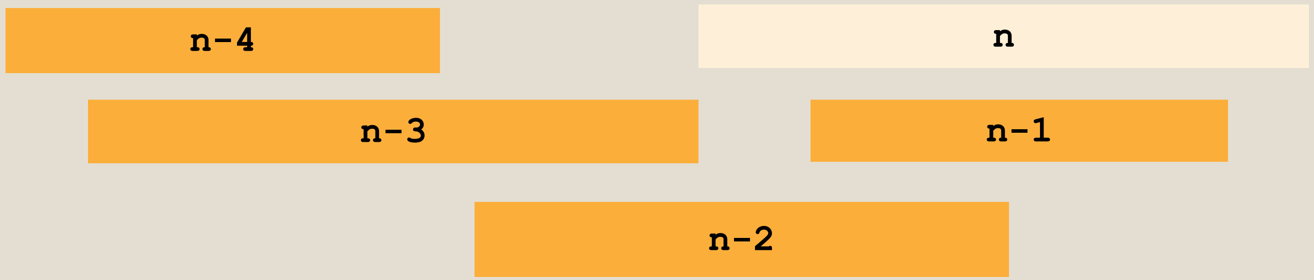


p(n)=n-3

Dynamic Programming: Binary Choice

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

Case 2: OPT does not select job n.



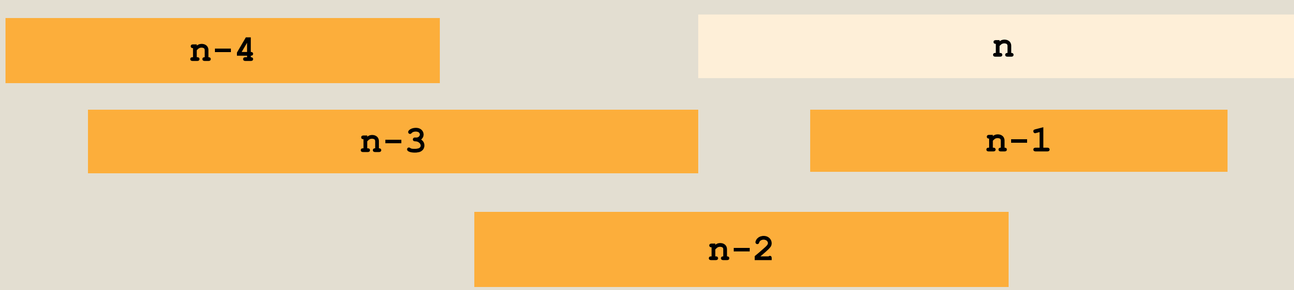
Dynamic Programming: Binary Choice

OPT(n) = value of optimal solution to the problem consisting of

jobs 1, 2, …, n.

Case 2: OPT does not select job n.

- must include optimal solution to problem consisting of remaining compatible jobs 1, 2, …, n-1



Dynamic Programming: Binary Choice

OPT(j) = value of optimal solution to the problem consisting of

jobs 1, 2, …, j.

optimal substructure

Case 1: OPT selects job j.

- collect profit w_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)

Case 2: OPT does not select job j.

- must include optimal solution to problem consisting of remaining compatible jobs 1, 2, …, j-1

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

Weighted Interval Scheduling

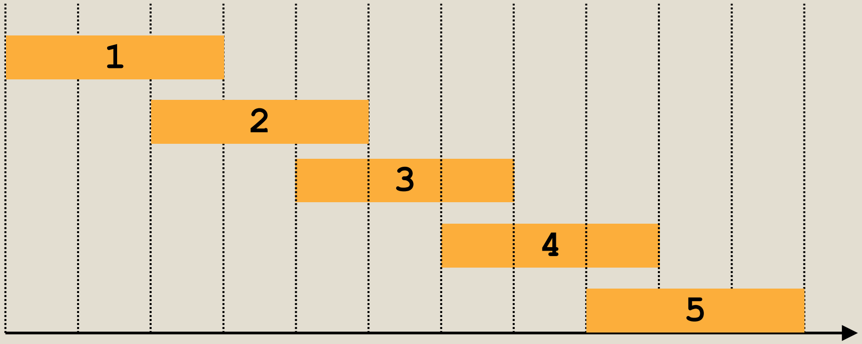
```
Input: n, s1,…,sn , f1,…,fn , w1,…,wn

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

Compute p(1), p(2), …, p(n)

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

p(n) = largest index i < n such that job i is compatible with job n.

$$T(n) = T(n-1) + T(p(n)) + O(1)$$

$$T(1) = 1$$

Weighted Interval Scheduling

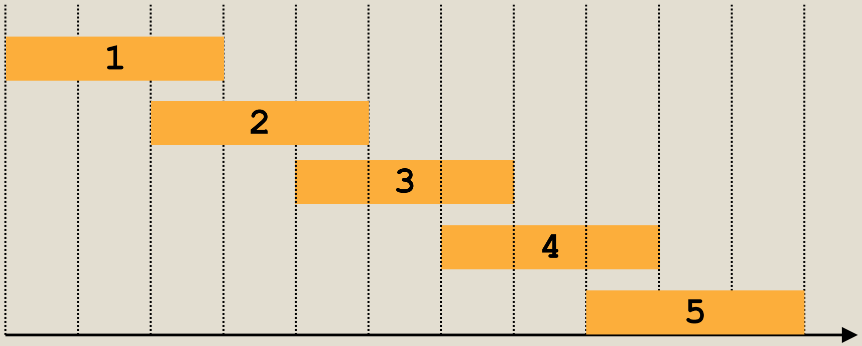


$$T(n) = T(n-1) + T(n-2) + 1$$
$$T(1) = 1$$

p(1) = 0, p(j) = j-2, … p(5)=3

Q: Is this polynomial time?

Weighted Interval Scheduling: Brute Force

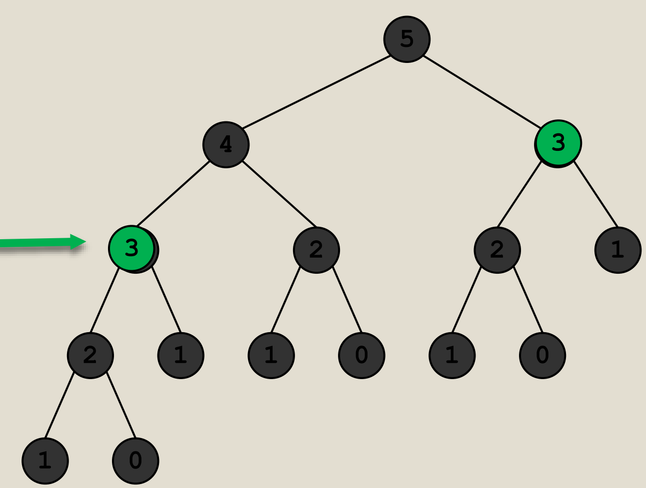


$$T(n) = T(n-1) + T(n-2) + 1$$
$$T(1) = 1$$

p(1) = 0, p(j) = j-2, … p(5)=3

No. Recursive algorithm has exponential time (because of repeated computations)

Key Insight: Do we need to repeat this computation?



Weighted Interval Scheduling: Bottom-Up

Bottom-up iterative dynamic programming.

```
Input: n, s_1,…,s_n , f_1,…,f_n , w_1,…,w_n

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

Compute p(1), p(2), …, p(n)

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

Weighted Interval Scheduling: Memoization

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

```
Input: n, s_1,…,s_n , f_1,…,f_n , w_1,…,w_n

Sort jobs by finish times so that f_1 ≤ f_2 ≤ ... ≤ f_n.
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(w_j + M-Compute-Opt(p(j)),
                   M-Compute-Opt(j-1))
   return M[j]
}
```

Weighted Interval Scheduling: Running Time

DP algorithm takes O(n log n) time.

- Sort by finish time: O(n log n)
- Computing p(·) : O(n log n) via binary search into –f-values.

M-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
- Overall running time is O(n).

Weighted Interval Scheduling: Finding a Solution

Q. DP algorithm computes optimal value. How do we get the solution itself?

A. Do some post-processing.

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

Find-Solution(j) {
   if (j = 0)
      output nothing
   else if (wj + 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).