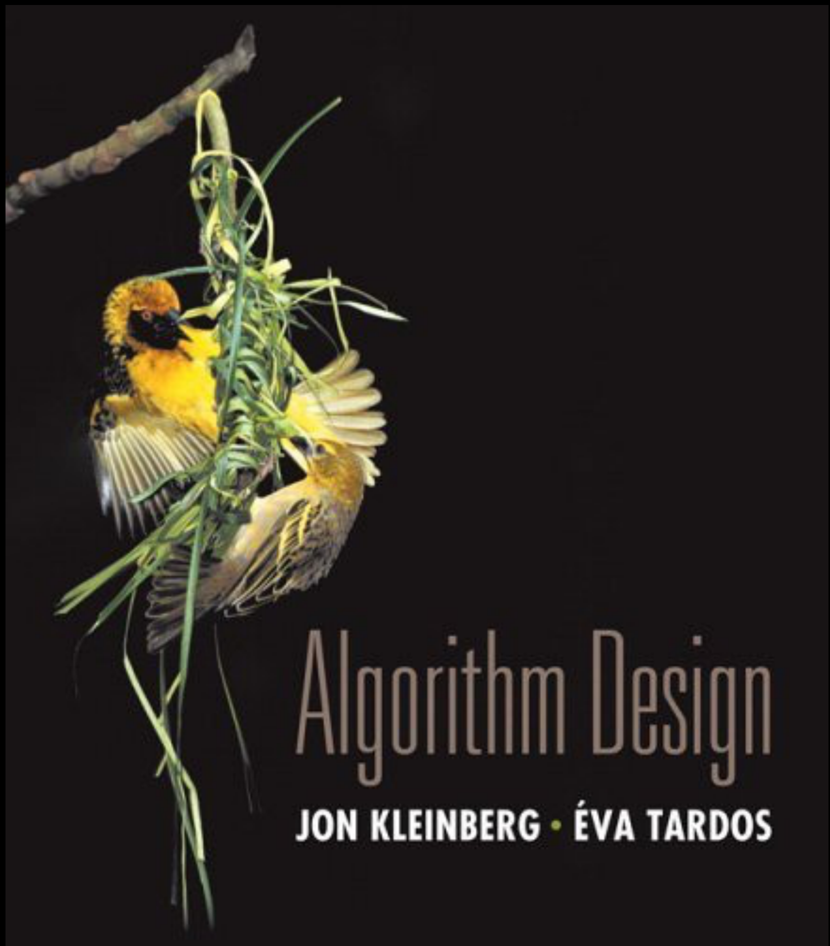


Chapter 6

Dynamic Programming

Slides by Kevin Wayne.

Algorithmic Paradigms

Greedy. Build up a solution incrementally, myopically optimizing some local criterion.

Divide-and-conquer. Break up a problem into 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 overlapping sub-problems, and build up solutions to larger and larger sub-problems.

Dynamic Programming History

Bellman. [1950s] Pioneered the systematic study of dynamic programming.

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, compilers, systems, ….

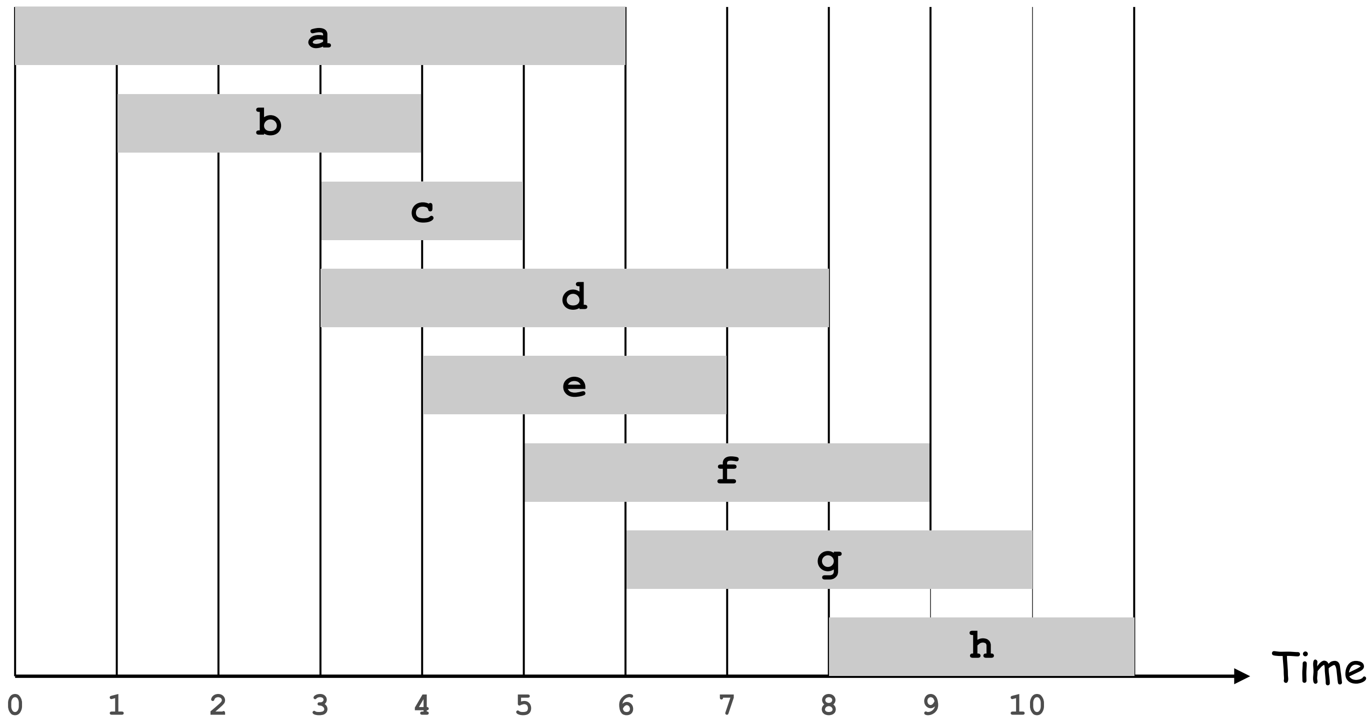
Some famous dynamic programming algorithms.

- 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.

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.

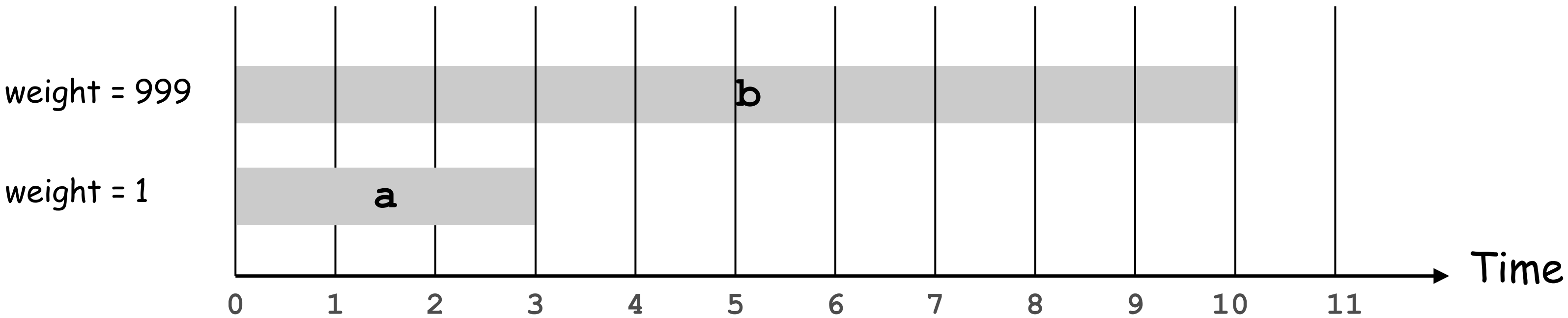


Unweighted Interval Scheduling Review

Recall. Greedy algorithm works if all weights are 1.

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

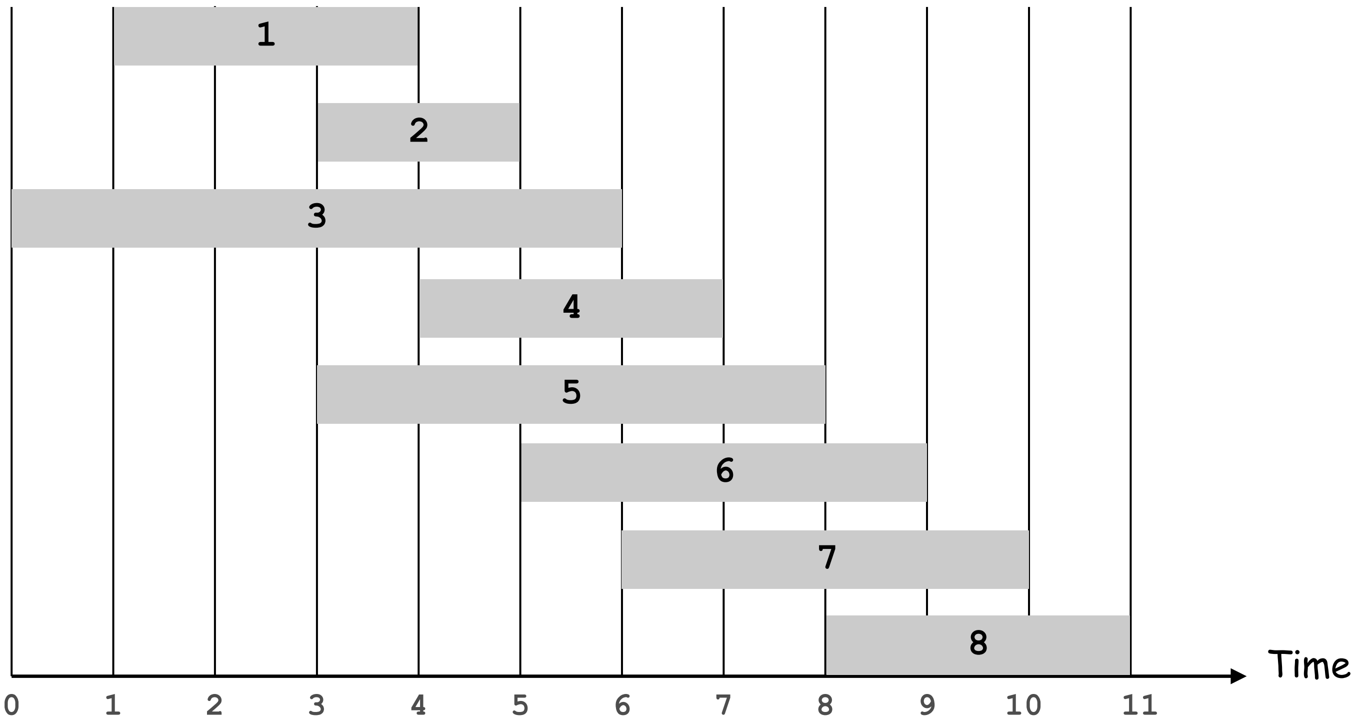
Observation. Greedy algorithm can fail spectacularly if arbitrary weights are allowed.



Weighted Interval Scheduling

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

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



Dynamic Programming: Binary Choice

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

- **Case 1:** *OPT* selects job *j*.
  - collect profit v_j
  - can't use incompatible jobs **{ p(j) + 1, p(j) + 2, ..., j - 1 }**
  - must **include optimal** solution to ***sub-problem*** consisting of remaining compatible jobs **1, 2, ..., p(j)**

optimal substructure

- **Case 2:** *OPT* does not select job *j*.
  - must include optimal solution to ***sub-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

Brute-force 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))
}
```

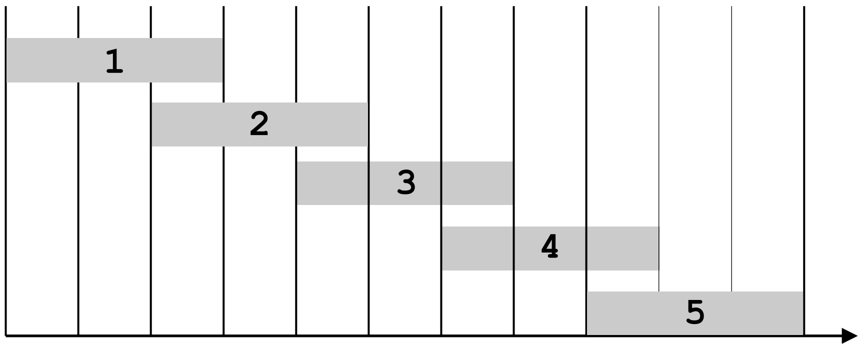
Optimal Cost for the Input of size **n** is computed by function:

Comptute-Opt(j:= n)

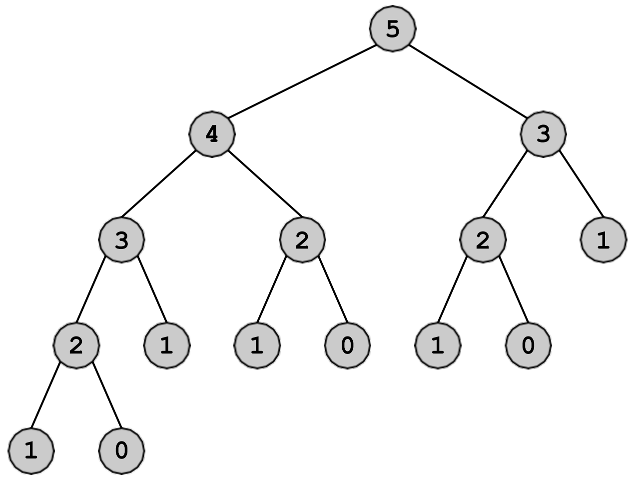
Weighted Interval Scheduling: Brute Force

Observation. **Recursive** algorithm **fails** spectacularly because of **redundant** sub-problems $\Rightarrow$ **exponential** algorithms.

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



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



But.....

Q. How many **<u>different</u>** subproblems we have?

A. Only ***n*** !!!

Q. Can we fix an ***ordering*** to compute them? .....

A. Yes!

Define an Array **M[1,...,n]**, where

M[j] := OPT(j) ≡ Optimum for the sub-problem {1,...,j}

Key-Fact: To compute M[j], we needs only the entries

M[0], M[1], ..., M[j-2], M[j-1] !!

We can apply induction!

Weighted Interval Scheduling:
ITERATIVE dynamic programming algorithm

Bottom-up dynamic programming. *remove recursion.*

```
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)

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

Crucial Issue: Find the correct order for computing the subproblems!

Weighted Interval Scheduling: Finding a Solution

Q. Dynamic programming algorithms computes optimal **value** "only". What if we want the **solution** itself?

A. Do some ***post-processing***.

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

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

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

Remark. O(n) if jobs are pre-sorted by start and finish times.

Weighted Interval Scheduling: Memoization

Memoization: Use Recursion but start a new call **only if** the the required value has not been computed yet.

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 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

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

return M[j]
```

Claim. Memoized version of algorithm takes O(n log n) time.

- Sort by finish time: O(n log n).
- Computing **p(·)**: O(n log n) via sorting by start time (do as homework).
- `M-Compute-Opt(j)`: each invocation takes O(1) time and either
  - **(i) returns an existing value** `M[j]` (when (*) is true!),

  OR
  - **(ii) fills-in one new entry** `M[j]` **and makes two recursive calls**
- Progress measure Φ = # nonempty entries of `M[]`.
  - initially Φ = 0, throughout $\Phi \leq$ **n**.
  - (ii) increases Φ by **1** $\Rightarrow$ at most 2n recursive calls.
- **Hence: Overall running time** of `M-Compute-Opt(n)` is O(n).

▪

Paradigm of Dynamic Programming (Informal Description)

Partition of the initial problem **P(n)** into a set of subproblems

$P_1(n_1), P_2(n_2), \ldots, P_k(n_k)$ such that

- $n_i < n$ for all $i=1,\ldots,k$
- $k = poly(n)$
- $P(n)$ can be computed from $P_1(n_1), P_2(n_2), \ldots, P_k(n_k)$ in poly-time
- There is a natural *ordering* (from *smaller* to *bigger*) of the subproblems so that **recursion** can be applied efficiently.

HOMEWORKS:

1. Give formal definition of the Weighted Interval Scheduling
2. Give an efficient Algorithm that, given any instance $I=\{I_1,\ldots,I_n\}$, computes Predecessor Function **p(j)**, for any **j = 1, …, n**.