Design and Analysis of Algorithms
Dynamic Programming

Si Wu

School of CSE, SCUT

cswusi@scut.edu.cn

TA: Wenhao Wu (1565865638@qq.com)

Yi Liu (1337545838@qq.com)

Topics

- **Weighted Interval Scheduling**
- **Segmented Least Squares**
- **Knapsack Problem**

Algorithmic Paradigms

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

Divide-and-conquer. Break up a problem into independent sub-problems, solve each sub-problem, and combine solutions to sub-problems to form solution to original problem.

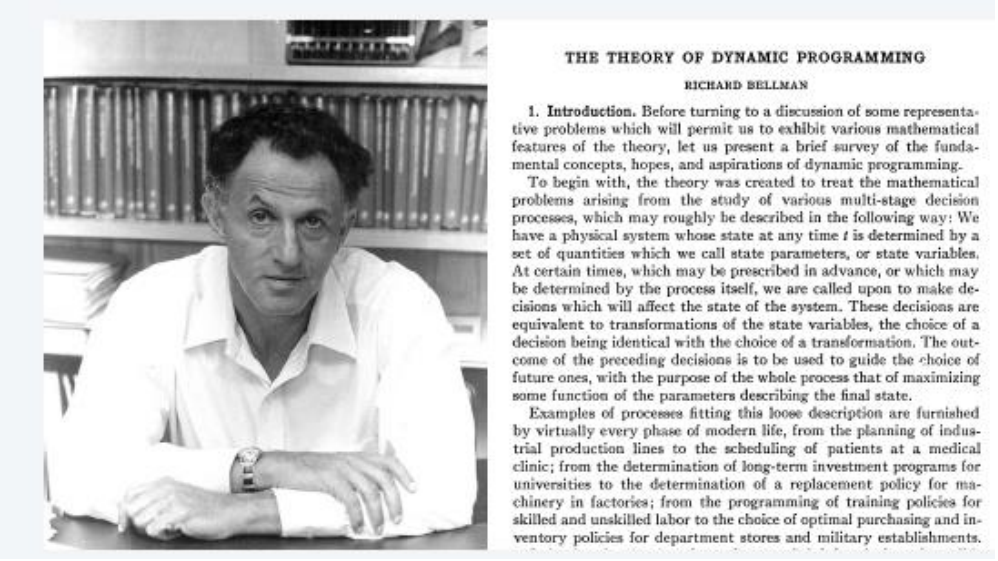
Dynamic programming. Break up a problem into a series of overlapping sub-problems, and build up solutions to larger and larger sub-problems (caching away intermedia results in a table for later reuse).

Dynamic Programming History

Bellman. Pioneered the systematic study of dynamic programming in 1950s.

Etymology.

- Dynamic programming = planning over time.
- Secretary of Defense was hostile to mathematical research.
- Bellman sought an impressive name to avoid confrontation.



THE THEORY OF DYNAMIC PROGRAMMING

RICHARD BELLMAN

1. **Introduction.** Before turning to a discussion of some representative problems which will permit us to exhibit various mathematical features of the theory, let us present a brief survey of the fundamental concepts, hopes, and aspirations of dynamic programming.

To begin with, the theory was created to treat the mathematical problems arising from the study of various multi-stage decision processes, which may roughly be described in the following way: We have a physical system whose state at any time t is determined by a set of quantities which we call state parameters, or state variables. At certain times, which may be prescribed in advance, or which may be determined by the process itself, we are called upon to make decisions which will affect the state of the system. These decisions are equivalent to transformations of the state variables, the choice of a decision being identical with the choice of a transformation. The outcome of the preceding decisions is to be used to guide the choice of future ones, with the purpose of the whole process that of maximizing some function of the parameters describing the final state.

Examples of processes fitting this loose description are furnished by virtually every phase of modern life, from the planning of industrial production lines to the scheduling of patients at a medical clinic; from the determination of long-term investment programs for universities to the determination of a replacement policy for machinery in factories; from the programming of training policies for skilled and unskilled labor to the choice of optimal purchasing and inventory policies for department stores and military establishments.

Dynamic Programming Applications

Areas.

- Bioinformatics.
- Control theory.
- Information theory.
- Operations research.
- Computer science: theory, graphics, AI, compilers, systems,…
- …

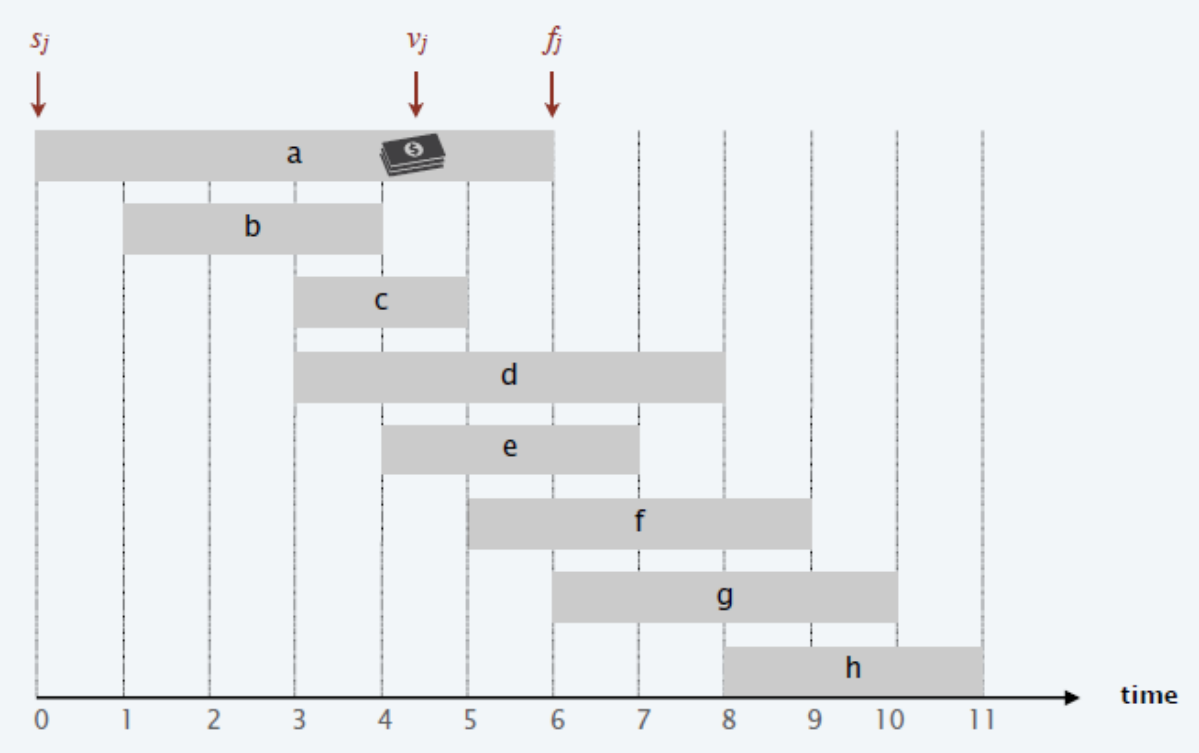
Some famous dynamic programming algorithms.

- Unix diff for comparing two files.
- Viterbi for hidden Markov models.
- De Boor for evaluating spline curves.
- Smith-Waterman for generic sequence alignment.
- Bellman-Ford for shortest patch routing in networks.
- …

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 jots compatible if they don't overlap.
- Goal: find maximum-weight subset of mutually compatible jobs.



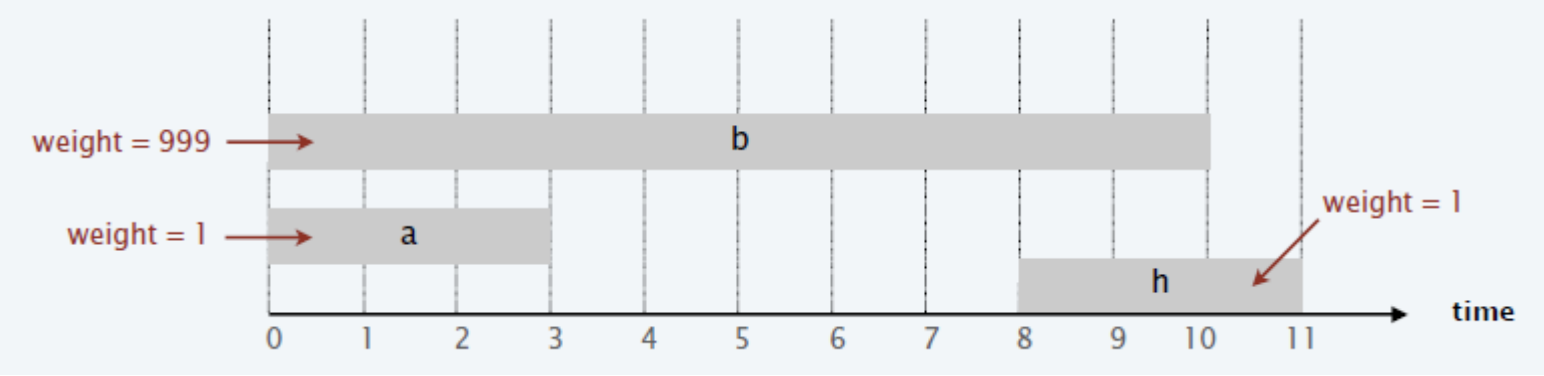
Earliest-Finish-Time First Algorithm

Earliest finish-time first.

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

Recall. Greedy algorithm is correct if all weights are 1.

Observation. Greedy algorithm fails spectacularly for weighted version.

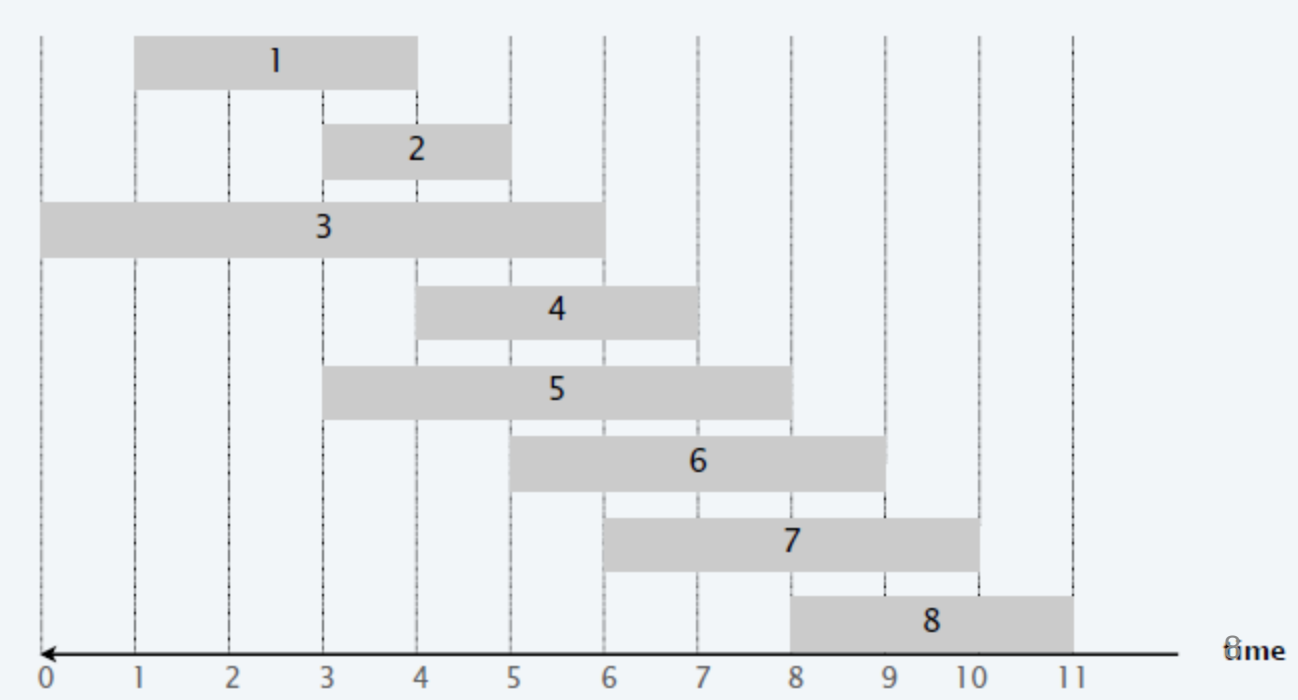


Weighted Interval Scheduling

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

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

Ex. $p(1), p(2), p(3), p(4), p(5), p(6), p(7), p(8)$.



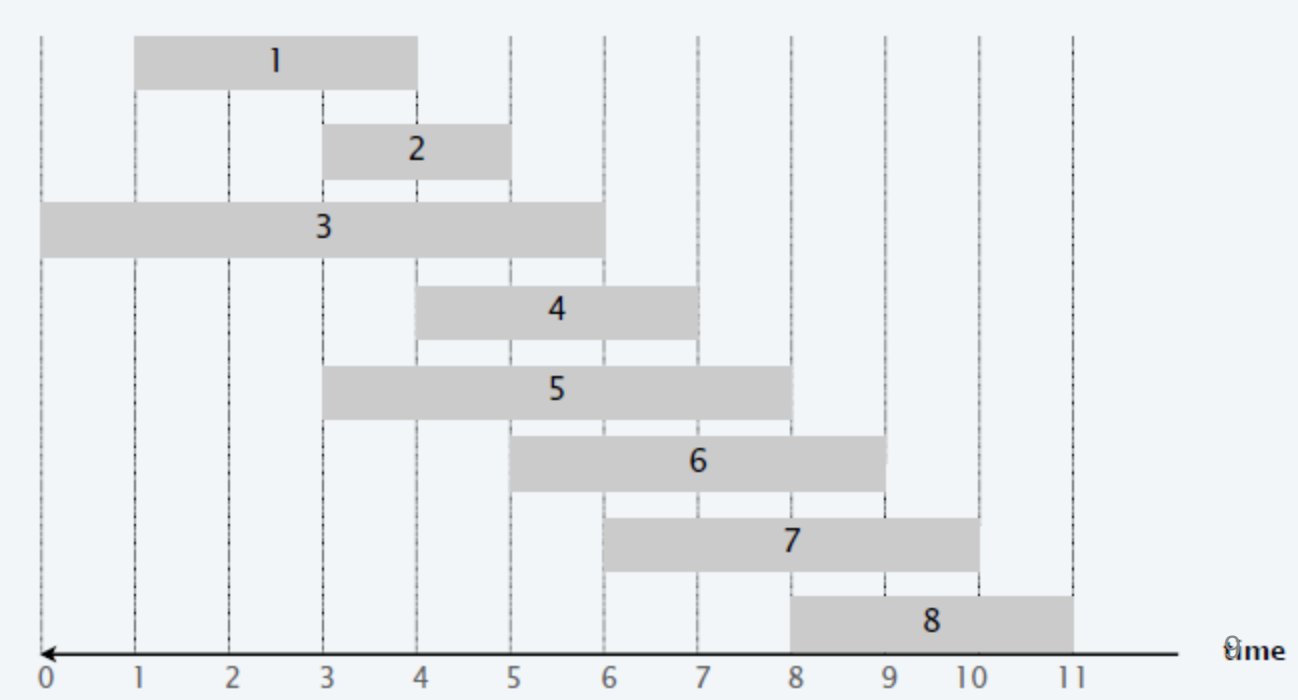
Weighted Interval Scheduling

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

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

Ex.

$p(1) = 0,$
$p(2) = 0,$
$p(3) = 0,$
$p(4) = 1,$
$p(5) = 0,$
$p(6) = 2,$
$p(7) = 3,$
$p(8) = 5.$



Dynamic Programming: Binary Choice

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

Goal. $OPT(n)$ = value of optimal solution to the original problem.

Case 1. $OPT(j)$ selects job j.

- Collect profit v_j.
- Can't use incompatible jobs $\{p(j)+1, p(j)+2, \ldots, j-1\}$.
- Must include optimal solution to problem consisting of remaining compatible jobs 1, 2, …, $p(j)$.

Dynamic Programming: Binary Choice

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

Goal. $OPT(n) =$ value of optimal solution to the original problem.

Case 2. $OPT(j)$ does not selects job j.

- Must include optimal solution to problem consisting of remaining jobs 1, 2, …, $j - 1$.

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

Weighted Interval Scheduling: Brute Force

Brute-Force $(n, s_1, s_2, \ldots, s_n, f_1, f_2, \ldots, f_n, v_1, v_2, \ldots, v_n)$

Sort jobs by finish time so that $f_1 \leq f_2 \leq \cdots \leq f_n$

Compute $p[1], p[2], \ldots, p[n]$.

Return Compute-Opt(n).

Compute-Opt(j)

If $j = 0$

  Return 0.

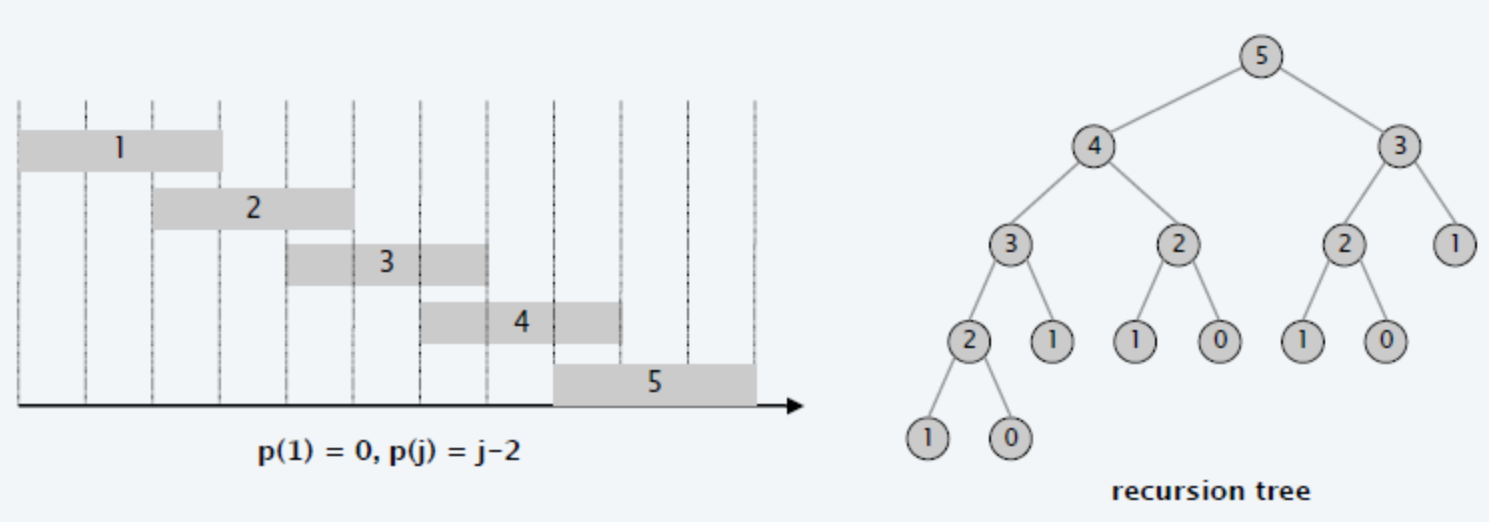
Else

  Return $\max\{ v_j +$ Compute-Opt$(p[j])$, Compute-Opt$(j-1)\}$.

Weighted Interval Scheduling: Brute Force

Observation. Recursive algorithm is spectacularly slow because of overlapping sub-problems -> exponential-time algorithm.

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



Weighted Interval Scheduling: Memorization

Top-down dynamic programming (memorization). Cache result of each sub-problem; lookup as needed.

Top-Down $(n, s_1, s_2, \ldots, s_n, f_1, f_2, \ldots, f_n, v_1, v_2, \ldots, v_n)$

Sort jobs by finish time so that $f_1 \leq f_2 \leq \cdots \leq f_n$

Compute $p[1], p[2], \ldots, p[n]$.

$M[0] \leftarrow 0$.

Return M-Compute-Opt(n).

M-Compute-Opt(j)

If $M[j] = uninitialized$

$M[j] \leftarrow \max\{ v_j +$ M-Compute-Opt$(p[j])$, M-Compute-Opt$(j-1)\}$.

Return $M[j]$

Weighted Interval Scheduling: Finding A Solution

Q. DP algorithm computes optimal value. How to find solution itself?

A. Make a second pass by calling Find-Solution(n).

Find-Solution (j)

If $j = 0$

  Return $\emptyset$

Else If $v_j + M[p[j]] > M[j-1]$

  Return $\{j\} \cup$ Find-Solution $(p[j])$.

Else

  Return Find-Solution $(j-1)$

Weighted Interval Scheduling: Bottom-Up Dynamic Programming

Bottom-up dynamic programming.

Bottom-Up $(n, s_1, s_2, \ldots, s_n, f_1, f_2, \ldots, f_n, v_1, v_2, \ldots, v_n)$

Sort jobs by finish time so that $f_1 \leq f_2 \leq \cdots \leq f_n$

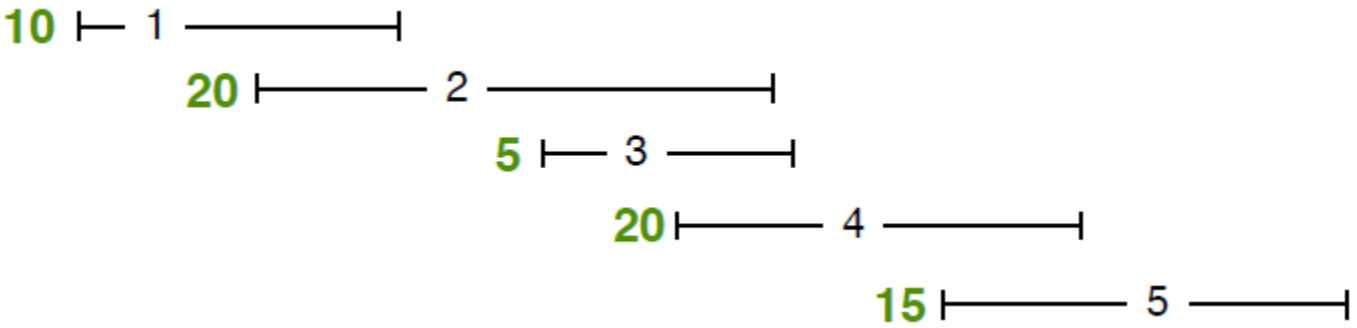
Compute $p[1], p[2], \ldots, p[n]$.

$M[0] \leftarrow 0$.

For $j = 1$ To n

  $M[j] \leftarrow \max\{ v_j + M[p[j]],\ M[j-1] \}$.

Weighted Interval Scheduling: Demo



Bottom-Up $(n, s_1, s_2, \ldots, s_n, f_1, f_2, \ldots, f_n, v_1, v_2, \ldots, v_n)$

Sort jobs by finish time so that $f_1 \leq f_2 \leq \cdots \leq f_n$

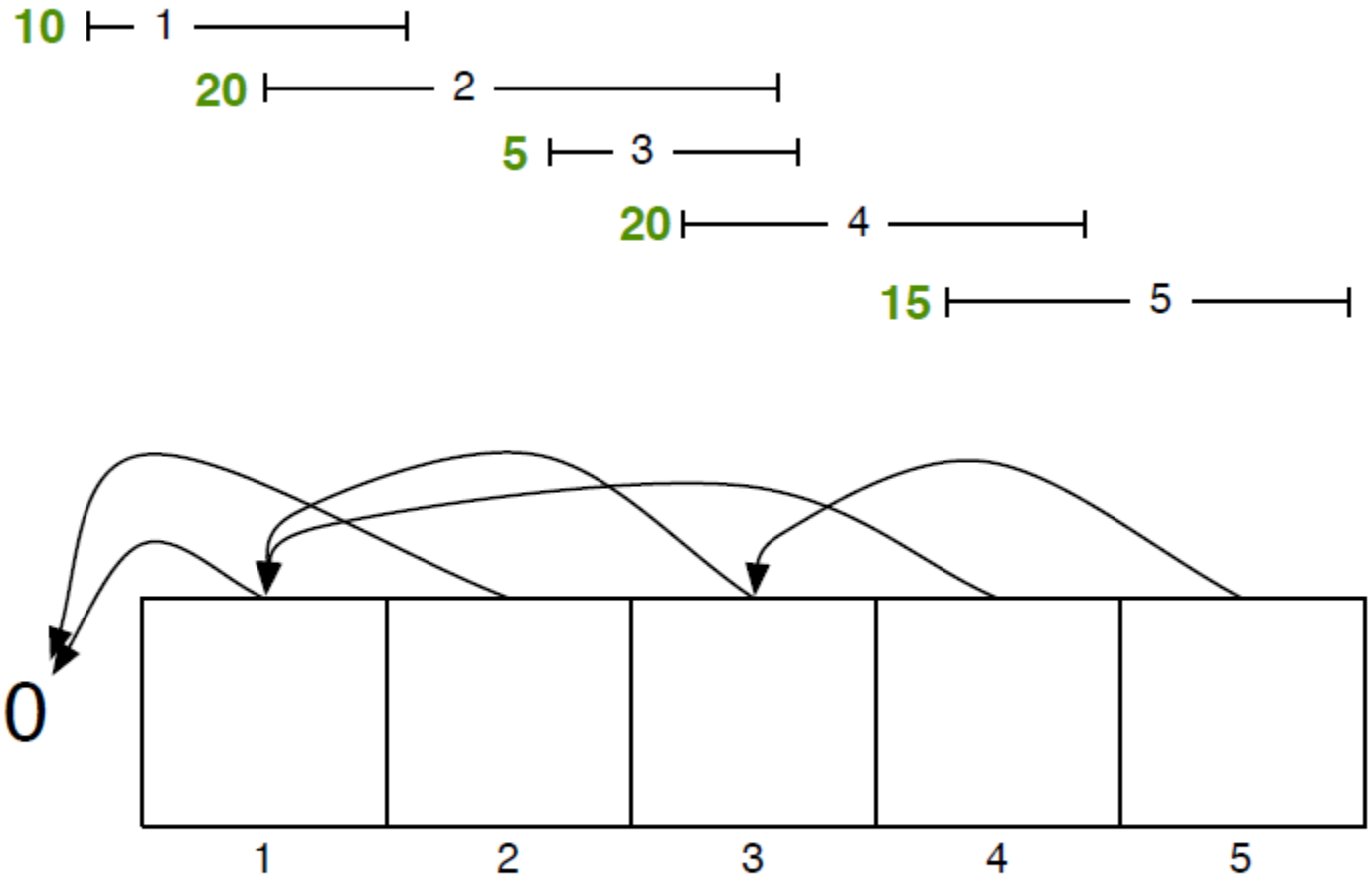
Compute $p[1], p[2], \ldots, p[n]$.

$M[0] \leftarrow 0$.

For $j = 1$ To n

  $M[j] \leftarrow \max\{ v_j + M\big[p[j]\big],\ M[j-1]\}$.

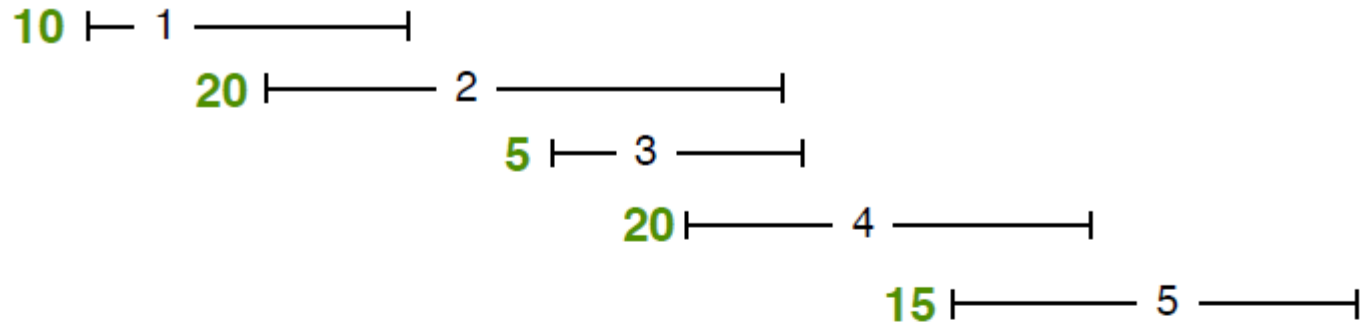
Weighted Interval Scheduling: Demo

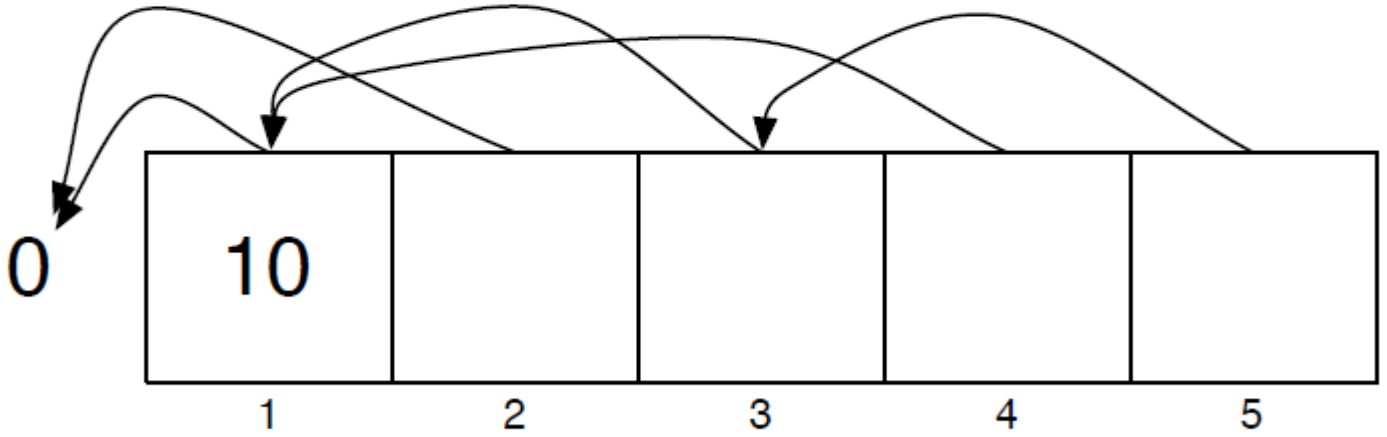


$v_j + M[p(j)]$

$M[j-1]$

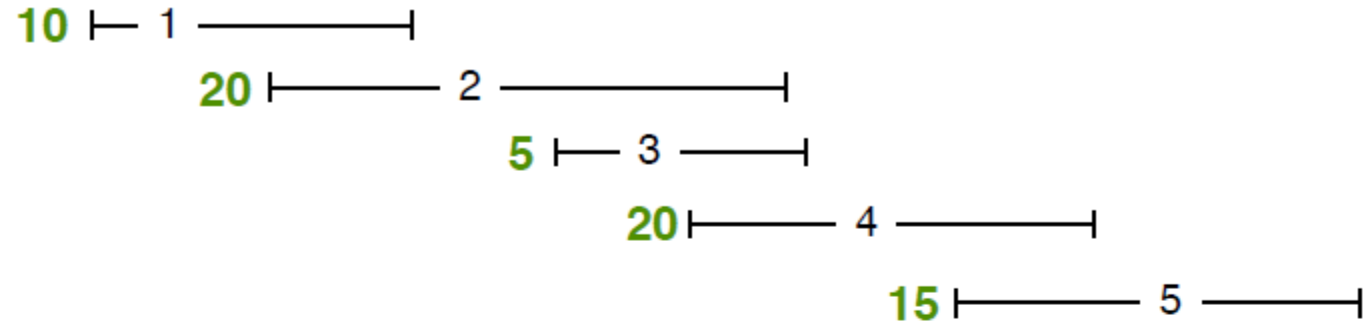
Weighted Interval Scheduling: Demo

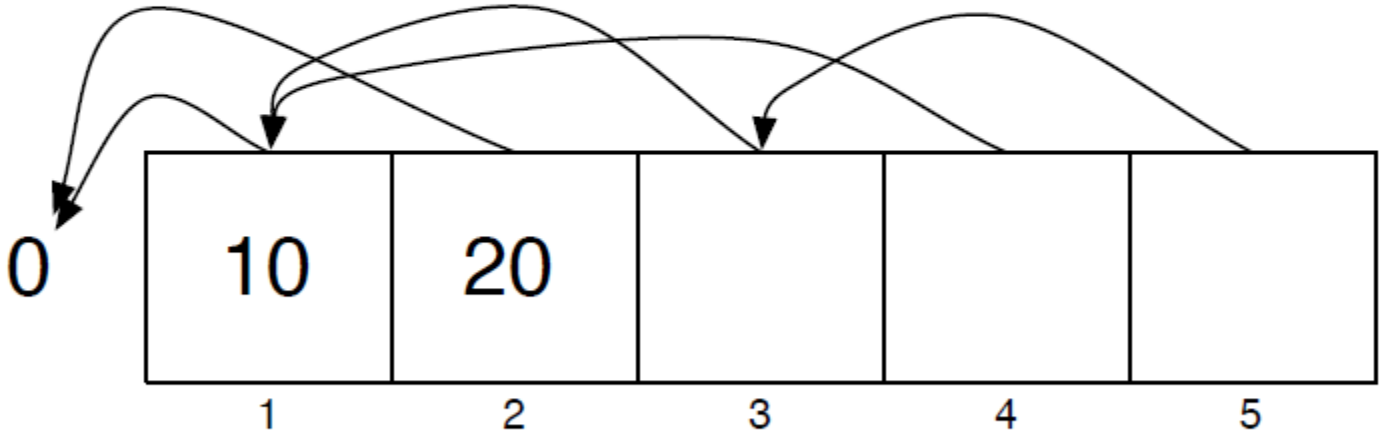




$v_j + M[p(j)]$	10
$M[j-1]$	0

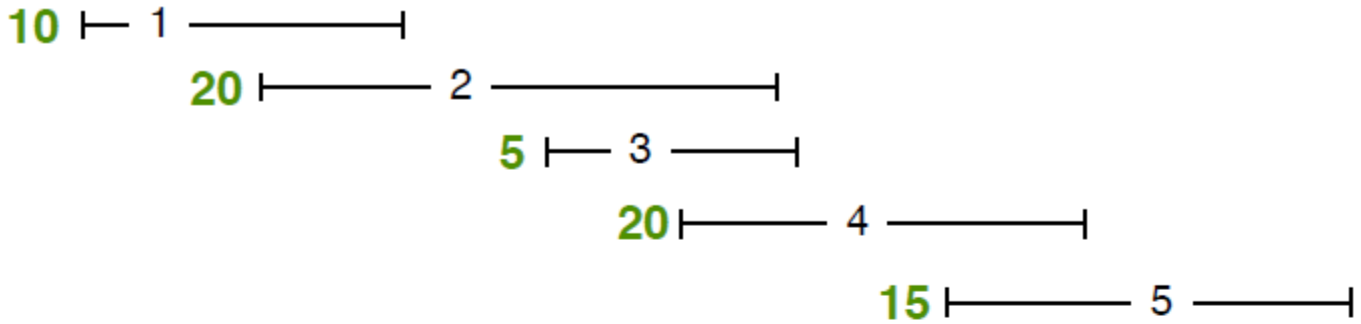
Weighted Interval Scheduling: Demo

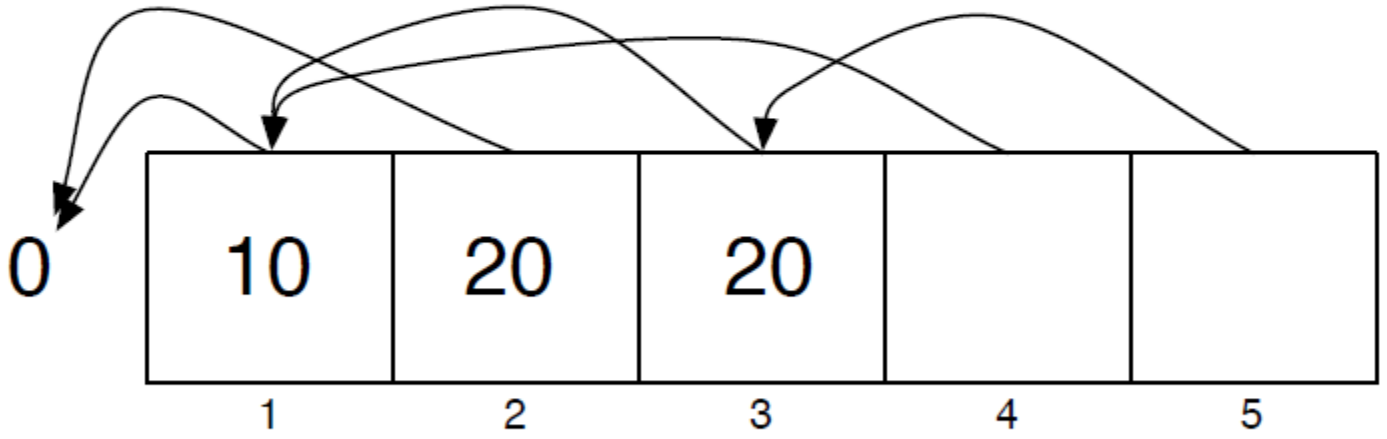




$v_j + M[p(j)]$	10	20
M[j-1]	0	10

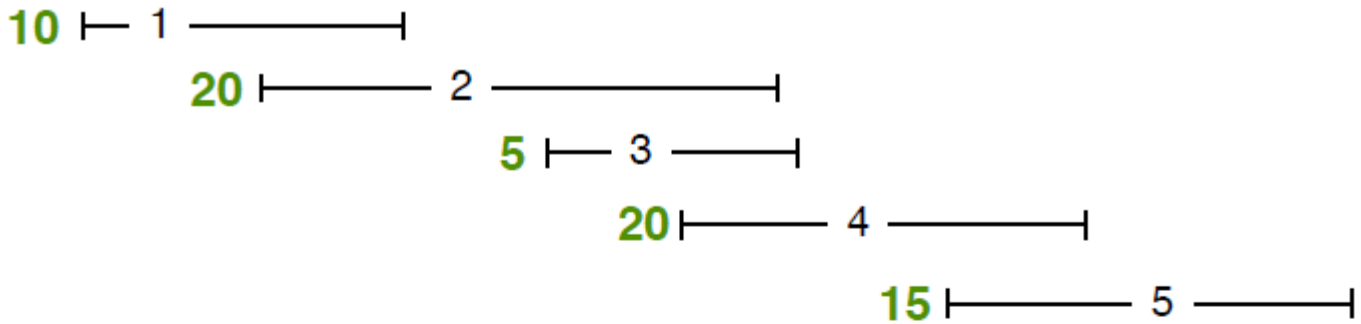
Weighted Interval Scheduling: Demo

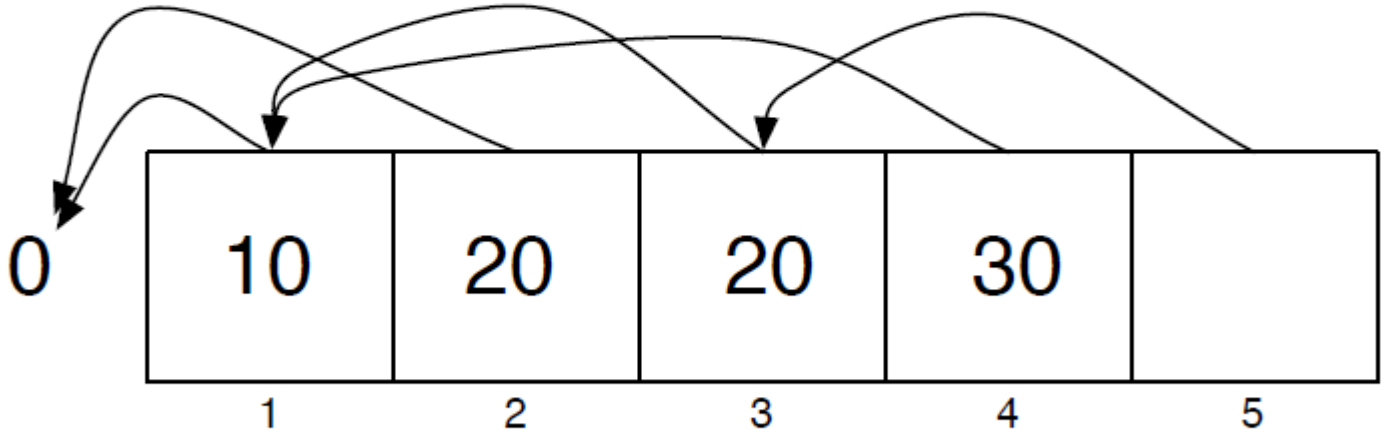




$v_j + M[p(j)]$	10	20	15
$M[j-1]$	0	10	20

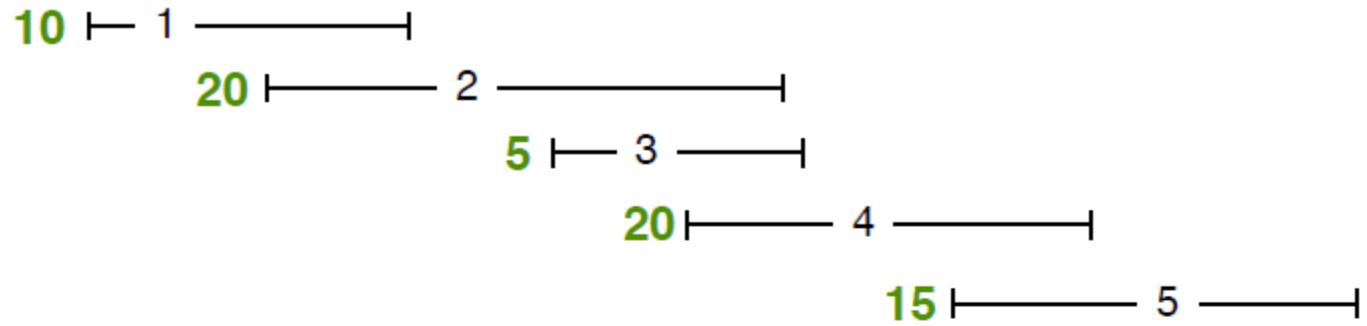
Weighted Interval Scheduling: Demo

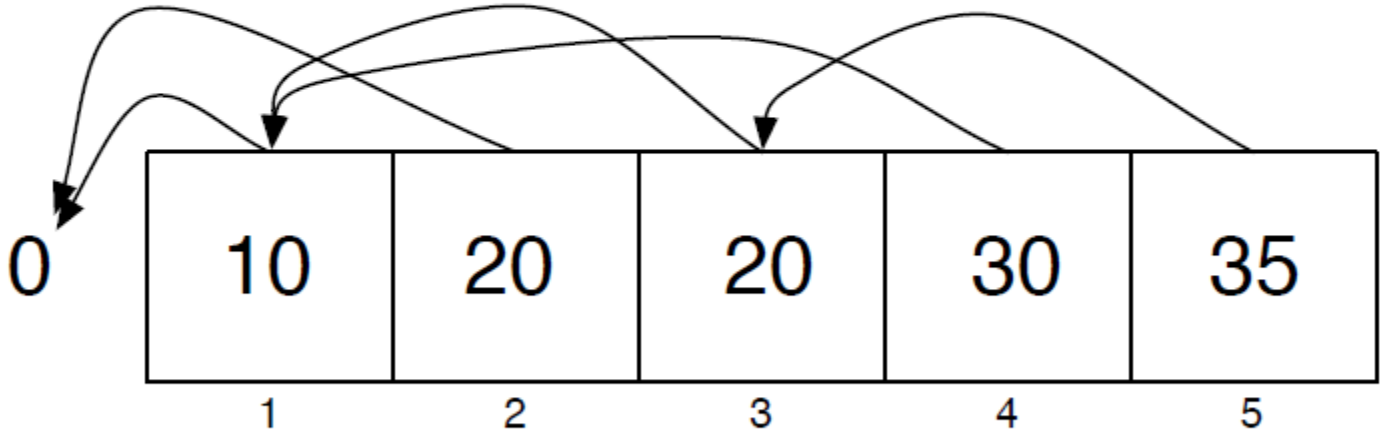




$v_j + M[p(j)]$	10	20	15	30
M[j-1]	0	10	20	20

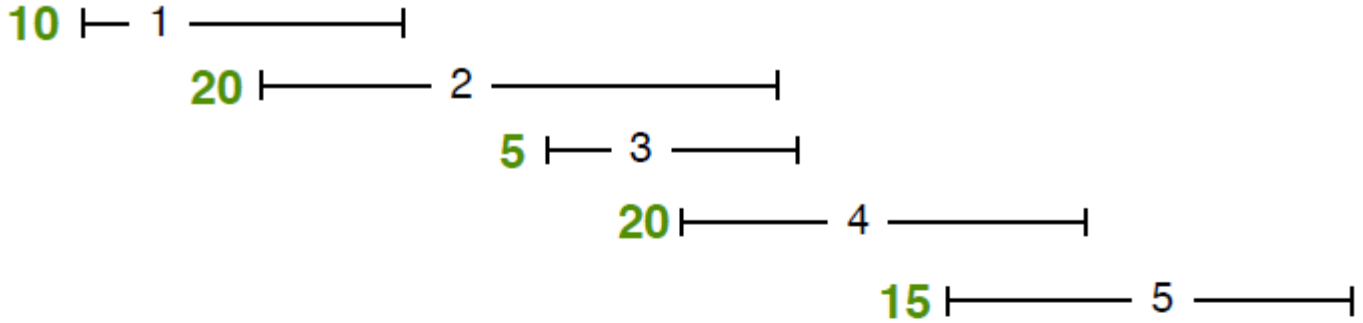
Weighted Interval Scheduling: Demo

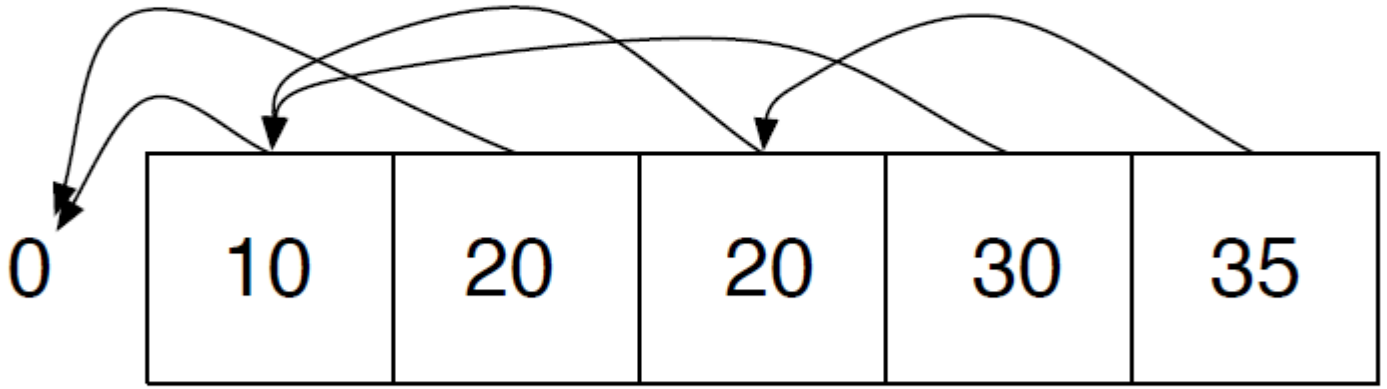




$v_j + M[p(j)]$	10	20	15	30	35
$M[j-1]$	0	10	20	20	30

Weighted Interval Scheduling: Demo



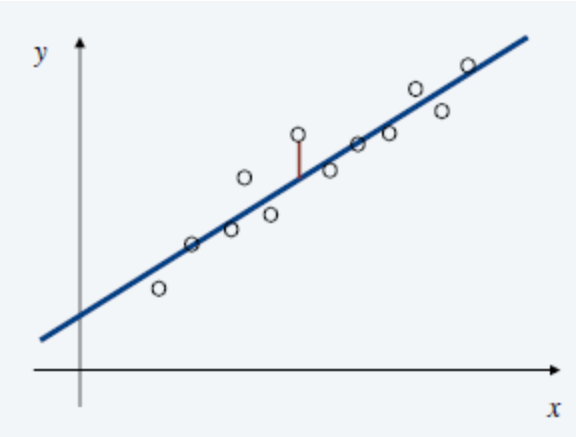


$v_j + M[p(j)]$	10	20	15	30	35
M[j-1]	0	10	20	20	30

Least Squares

Least squares. Foundational problem in statistics.

- Given n points in the plane: $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$.
- Find a line $y = ax + b$ that minimizes the sum of the squared error:
  $SSE = \sum_{i=1}^{n} (y_i - ax_i - b)^2$



Solution. Calculus -> min error is achieved when

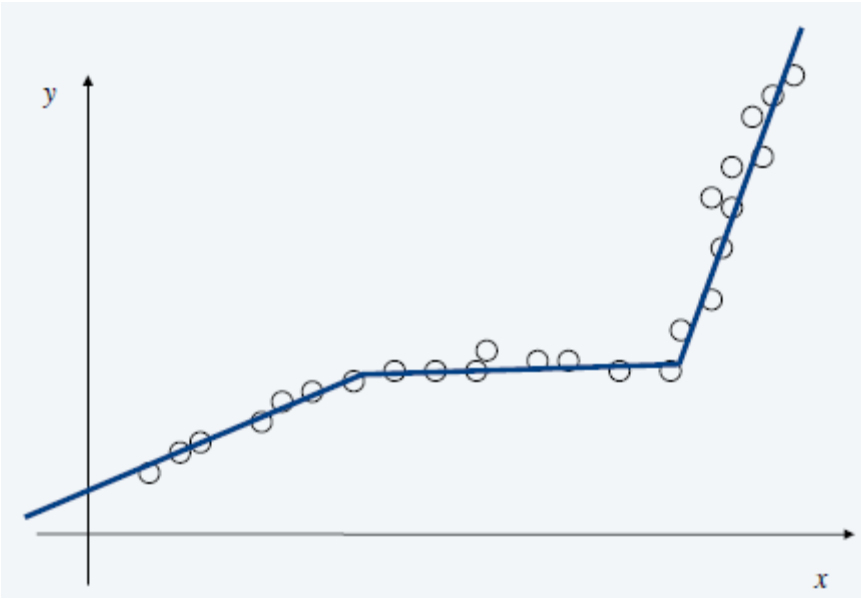
$$a = \frac{n \sum_i x_i y_i - (\sum_i x_i)(\sum_i y_i)}{n \sum_i x_i^2 - (\sum_i x_i)^2}, \; b = \frac{\sum_i y_i - a \sum_i x_i}{n}$$

Segmented Least Squares

Segmented least squares.

- Points lie roughly on a sequence of several line segments.
- Given n points in the plane: $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ with $x_1 \leq x_2 \leq \cdots \leq x_n$, find a sequence of lines that minimizes $f(x)$.

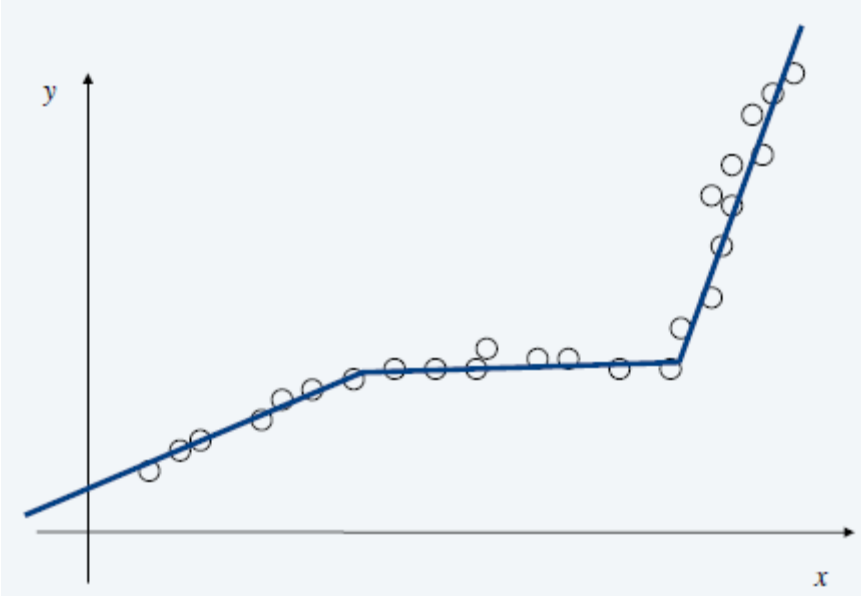
Q. What is a reasonable choice for $f(x)$ to balance accuracy and parsimony?



Segmented Least Squares

Given n points in the plane: $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ with $x_1 \leq x_2 \leq \cdots \leq x_n$, and a constant $c > 0$, find a sequence of lines that minimizes $f(x) = E + cL$:

- $E =$ the sum of the sums of the squared errors in each segment.
- $L =$ the number of lines.



Dynamic Programming: Multiway Choice

Notation.

- $OPT(j)$ = minimum cost for points $p_1, p_2, \ldots, p_j$.
- $e(i, j)$ = minimum sum of squares for points $p_i, p_{i+1}, \ldots, p_j$.

To compute $OPT(j)$.

- Last segment uses points $p_i, p_{i+1}, \ldots, p_j$ for some i.
- Cost = $e(i, j) + c + OPT(i-1)$.

$$OPT(j) = \begin{cases} 0 & if\ j = 0 \\ \min\limits_{1 \le i \le j} \{e(i, j) + c + OPT(i-1)\} & otherwise \end{cases}$$

Segmented Least Squares Algorithm

Segmented-Least-Squares $(n, p_1, p_2, \ldots, p_n, c)$

For $j = 1$ To n

  For $i = 1$ To j

    Compute the least squares $e(i, j)$ for the segment $p_i, p_{i+1}, \ldots, p_j$.

$M[0] \leftarrow 0$.

For $j = 1$ To n

  $M[j] \leftarrow \min_{1 \le i \le j} \{e(i, j) + c + OPT(i - 1)\}$.

Return $M[n]$.

Segmented Least Squares Analysis

Theorem. The dynamic programming algorithm solves the segmented least squares problem in $O(n^3)$ time.

Pf.

- Bottleneck = computing $e(i, j)$ for $O(n^2)$ pairs.
- $O(n)$ per pair using formula.

$$a = \frac{n \sum_i x_i y_i - (\sum_i x_i)(\sum_i y_i)}{n \sum_i x_i^2 - (\sum_i x_i)^2}, b = \frac{\sum_i y_i - a \sum_i x_i}{n}$$

Knapsack Problem

- Given n items and a "Knapsack".
- Item i weights $w_i > 0$ and has value $v_i > 0$.
- Knapsack has weight capacity of W.
- Goal: pack knapsack so as to maximize total value.

Ex. {1,2,5} has value 35 and weight 10.
Ex. {3,4} has value 40 and weight 11.
Ex. {3,5} has value 46 but exceeds weight limit.

i	v_i	w_i
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

knapsack instance
(weight limit W = 11)

Greedy by value. Repeatedly add item with maximum v_i.
Greedy by weight. Repeatedly add item with minimum w_i.
Greedy by ratio. Repeatedly add item with maximum ratio v_i / w_i.

Observation. None of greedy algorithms is optimal.

Dynamic Programming: False Start

Def. $OPT(i) =$ max-profit subset of items 1,2, … i.
Goal. $OPT(n)$.

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

Case 2. $OPT(i)$ selects item i.
- Selecting 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 1,2, … i with weight limit w.

Goal. $OPT(n, W)$.

Case 1. $OPT(i, w)$ does not select item i.

- $OPT(i, w)$ selects best of {1, 2, …, $i - 1$} using weight limit w.

Case 2. $OPT(i, w)$ selects item i.

- Collect value v_i.
- New weight limit = $w - w_i$.
- $OPT(i, w)$ selects best of {1, 2, …, $i - 1$} using this new weight limit.

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

Knapsack Problem: Bottom-Up Dynamic Programming

Knapsack $(n, W, w_1, w_2, \ldots, w_n, v_1, v_2, \ldots, v_n)$

For $w = 0$ To W

  $M[0, w] \leftarrow 0.$

For $i = 1$ To n

  For $w = 0$ To W

    If $w_i > w$

      $M[i, w] \leftarrow M[i-1, w].$

    Else

      $M[i, w] \leftarrow \max\{M[i-1, w], v_i + M[i-1, w-w_i]\}.$

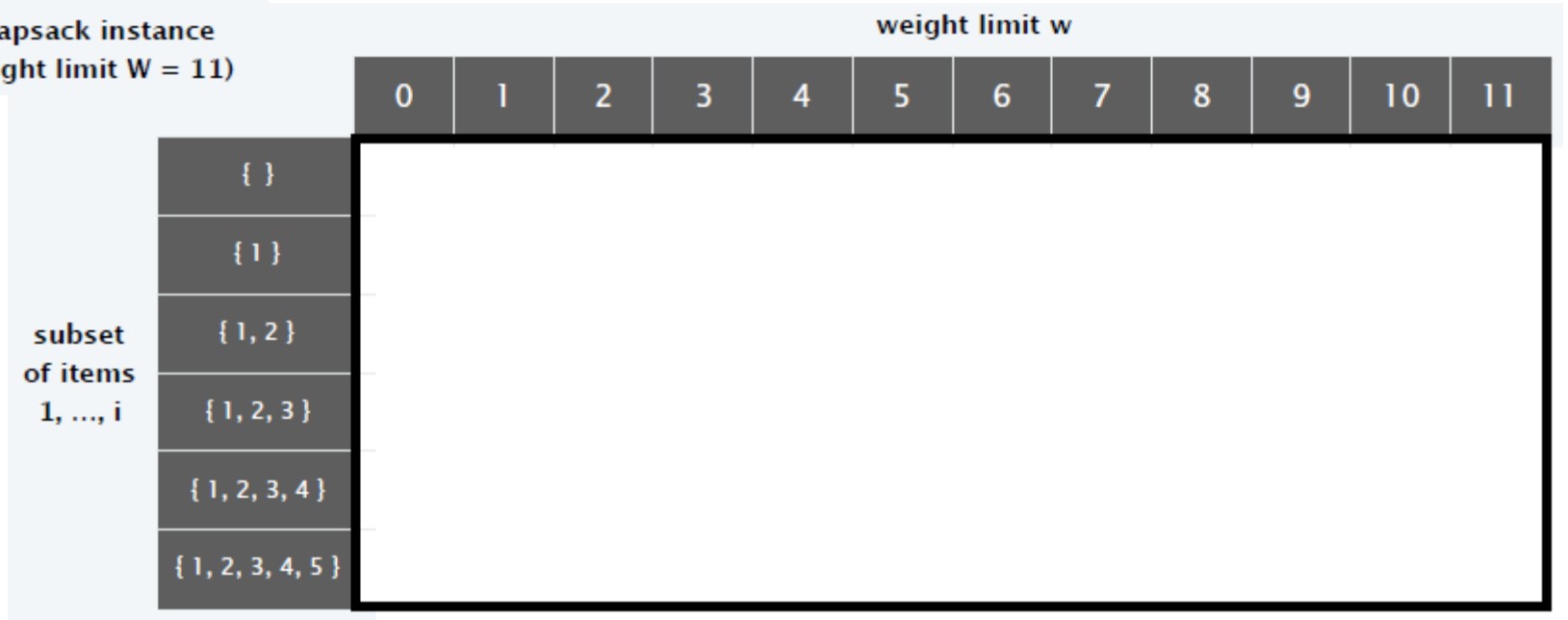
Return $M[n, W].$

Knapsack Problem: Bottom-Up Dynamic Programming Demo

i	v_i	w_i
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

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

knapsack instance (weight limit W = 11)



Knapsack Problem: Bottom-Up Dynamic Programming Demo

i	v_i	w_i
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

knapsack instance (weight limit W = 11)

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

subset of items 1, ..., i \ weight limit w	0	1	2	3	4	5	6	7	8	9	10	11
{ }	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(i, w) = max-profit subset of items 1, ..., i with weight limit w.