Dynamic Programming

Algorithm Design 6.1, 6.2, 6.3

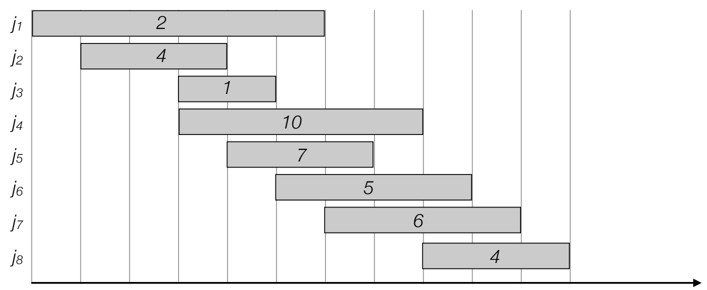
Thank you to Kevin Wayne for inspiration to slides

Applications

- In class (today and next time)

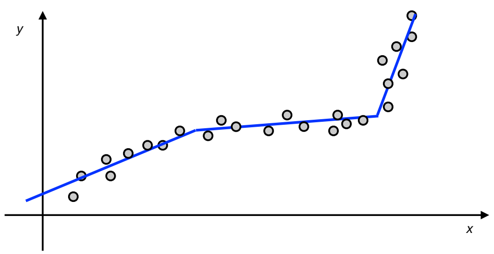
Applications

- In class (today and next time)
  - Weighted interval scheduling
    - Set of weighted intervals with start and finishing times
    - Goal: find maximum weight subset of non-overlapping intervals



Applications

- In class (today and next time)
  - Weighted interval scheduling
  - Segmented least squares
    - Given n points in the plane find a small sequence of lines that minimizes the squared error.



Applications

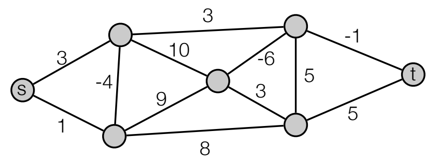
- In class (today and next time)
  - Weighted interval scheduling
  - Segmented least squares
  - Sequence alignment
    - Given two strings A and B how many edits (insertions, deletions, relabelings) is needed to turn A into B?

```
A C A A G T C        A C A A - G T C
- C A T G T -        - C A - T G T -

1 mismatch, 2 gaps    0 mismatches, 4 gaps
```


Applications

- In class (today and next time)
  - Weighted interval scheduling
  - Segmented least squares
  - Sequence alignment
  - Shortest paths with negative weights
    - Given a weighted graph, where edge weights can be negative, find the shortest path between two given vertices.



Applications

- In class (today and next time)
  - Weighted interval scheduling
  - Segmented least squares
  - Sequence alignment
  - Shortest paths with negative weights
- Some other famous applications
  - Unix diff for comparing 2 files
  - Vovke-Kasami-Younger for parsing context-free grammars
  - Viterbi for hidden Markov models
  - ….

Dynamic Programming

- Greedy. Build solution incrementally, optimizing some local criterion.

- Divide-and-conquer. Break up problem into independent subproblems, solve each subproblem, and combine to get solution to original problem.

- Dynamic programming. Break up problem into overlapping subproblems, and build up solutions to larger and larger subproblems.
  - Can be used when the problem have “optimal substructure”:
    - *Solution can be constructed from optimal solutions to subproblems*
    - *Use dynamic programming when subproblems overlap.*

Computing Fibonacci numbers

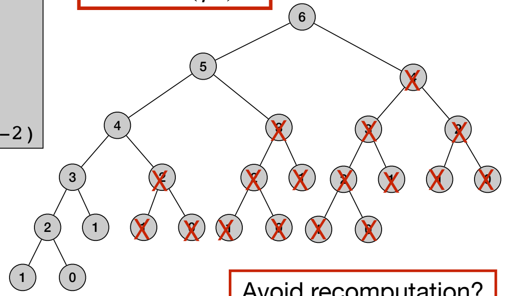
- Fibonacci numbers:

$$F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{otherwise} \end{cases}$$

- First try:

```
Fib(n)
if n = 0
  return 0
else if n = 1
  return 1
else
  return Fib(n-1) + Fib(n-2)
```

time $\Theta(\phi^n)$



Avoid recomputation?

Memoized Fibonacci numbers

- Fibonacci numbers:

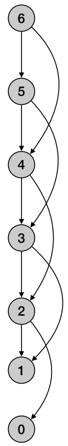
$$F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{otherwise} \end{cases}$$

- Remember already computed values:

```
for j=1 to n
  F[j] = null
Mem-Fib(n)

Mem-Fib(n)
if n = 0
  return 0
else if n = 1
  return 1
else
  if F[n] is empty
    F[n] = Mem-Fib(n-1) + Mem-Fib(n-2)
  return F[n]
```

time $\Theta(n)$



Bottom-up Fibonacci numbers

- Fibonacci numbers:

$$F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{otherwise} \end{cases}$$

- Remember already computed values:

```
Iter-Fib(n)
F[0] = 0
F[1] = 1
for i = 2 to n
  F[n] = F[n-1] + F[n-2]
return F[n]
```

time $\Theta(n)$

space $\Theta(n)$

Bottom-up Fibonacci numbers - save space

- Fibonacci numbers:

$$F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{otherwise} \end{cases}$$

- Remember last two computed values:

```
Iter-Fib(n)
previous = 0
current = 1
for i = 1 to n
  next = previous + current
  previous = current
  current = next
return current
```

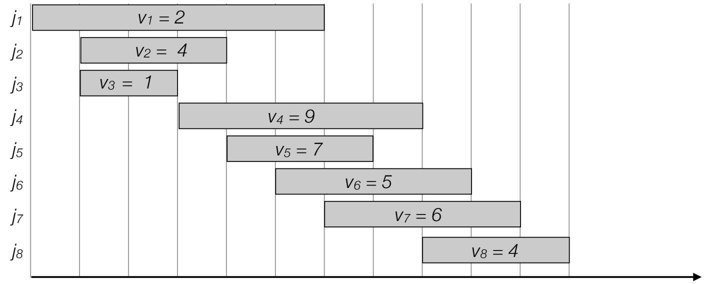
time $\Theta(n)$

space $\Theta(1)$

Weighted Interval Scheduling

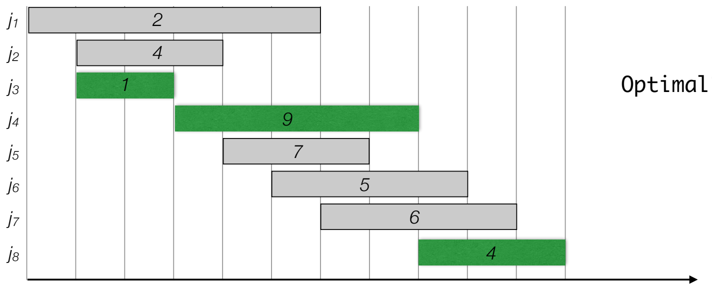
Weighted interval scheduling

- Weighted interval scheduling problem
  - n jobs (intervals)
  - Job *i* starts at s_i, finishes at f_i and has weight/value v_i.
  - Goal: Find maximum weight subset of non-overlapping (compatible) jobs.



Weighted interval scheduling

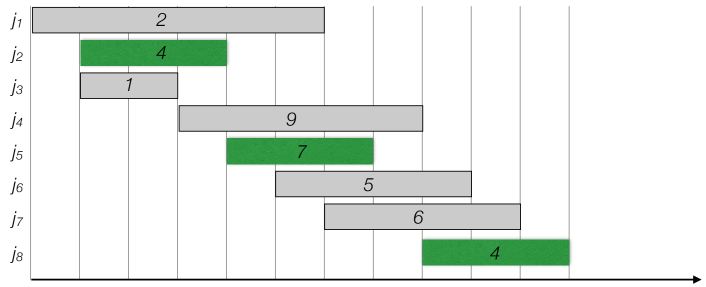
- Weighted interval scheduling problem
  - n jobs (intervals)
  - Job *i* starts at s_i, finishes at f_i and has weight/value v_i.
  - Goal: Find maximum weight subset of non-overlapping (compatible) jobs.



Optimal?

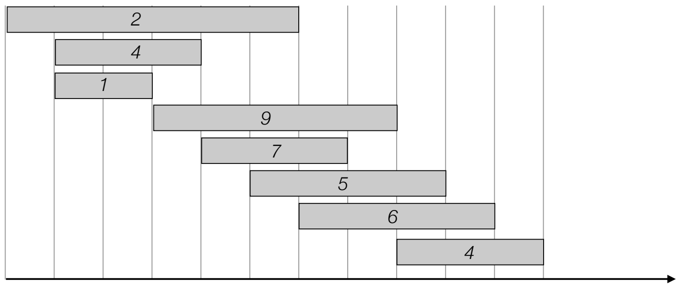
Weighted interval scheduling

- Weighted interval scheduling problem
  - n jobs (intervals)
  - Job *i* starts at s_i, finishes at f_i and has weight/value v_i.
  - Goal: Find maximum weight subset of non-overlapping (compatible) jobs.



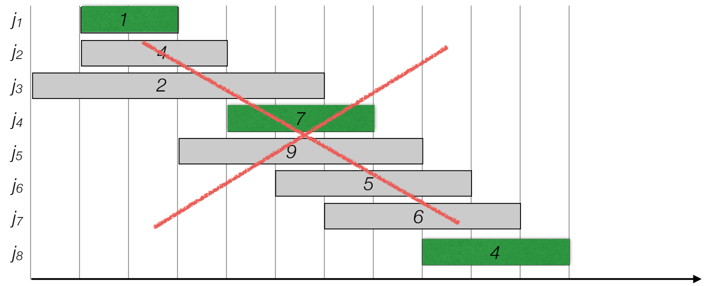
Weighted interval scheduling

- Label/sort jobs by finishing time: $f_1 \leq f_2 \leq \ldots \leq f_n$



Weighted interval scheduling

- Label/sort jobs by finishing time: $f_1 \leq f_2 \leq \ldots \leq f_n$
- ~~Greedy?~~

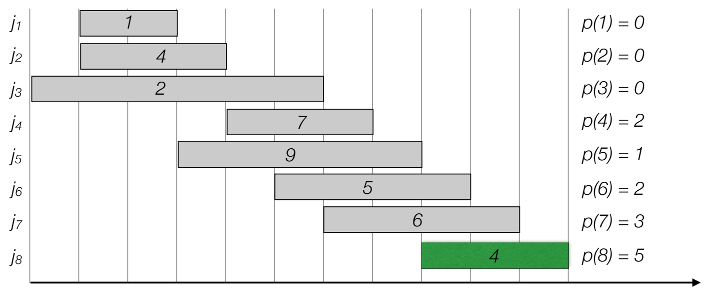


Weighted interval scheduling

- Label/sort jobs by finishing 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.
- Optimal solution OPT:
  - Case 1. OPT selects last job

    OPT = v_n + *optimal solution to subproblem on 1,…,p(n)*
  - Case 2. OPT does not select last job

    OPT = optimal solution to subproblem on 1,…,n-1



Weighted interval scheduling

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

    OPT(j) = v_j + *optimal solution to subproblem on 1,…,p(j)*
  - Case 2. OPT(*j*) does not select job j

    OPT = optimal solution to subproblem 1,…j-1

- Recursion:

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

Weighted interval scheduling: brute force

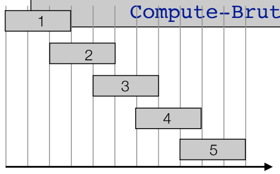
$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\{v_j + OPT(p(j)), OPT(j-1)\} & \text{otherwise} \end{cases}$$

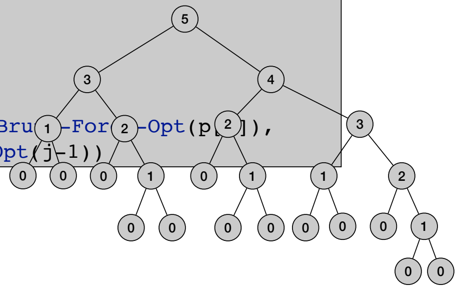
```
Input: n, s[1..n], f[1..n], v[1..n]

Sort jobs by finish time so that f[1] ≤ f[2]≤ … ≤ f[n]
Compute p[1], p[2], …, p[n]
Compute-BruteForce-Opt(n)

Compute-Brute-Force-Opt(j)
if j = 0
  return 0
else
  return max(v[j] + Compute-Brute-Force-Opt(p[j]),
       Compute-Brute-Force-Opt(j-1))
```

time $O(2^n)$





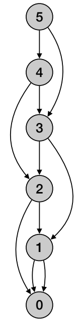
Weighted interval scheduling: memoization

```
Input: n, s[1..n], f[1..n], v[1..n]

Sort jobs by finish time so that f[1] ≤ f[2]≤ … ≤ f[n]
Compute p[1], p[2], …, p[n]

for j=1 to n
  M[j] = null
M[0] = 0.
Compute-Memoized-Opt(n)

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



- Running time O(n log n):
  - Sorting takes O(n log n) time.
  - Computing p(n): O(n log n) - use log n time to find each p(i).
  - Each subproblem solved once.
  - Time to solve a subproblem constant.
- Space O(n)

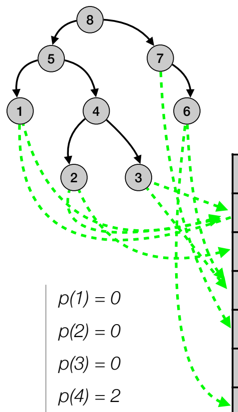
Weighted interval scheduling: memoization

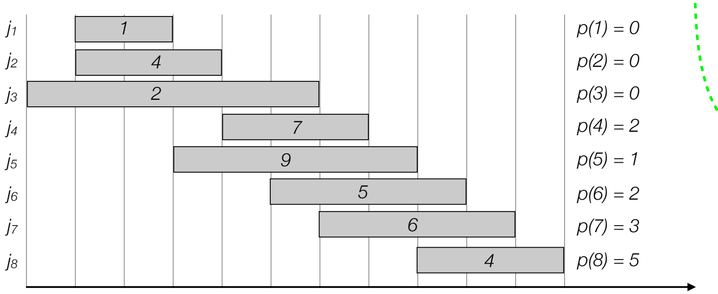
```
Input: n, s[1..n], f[1..n], v[1..n]

Sort jobs by finish time 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.
Compute-Memoized-Opt(n)

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





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

i	M[i]
0	0
1	1
2	4
3	4
4	11
5	11
6	11
7	11
8	15

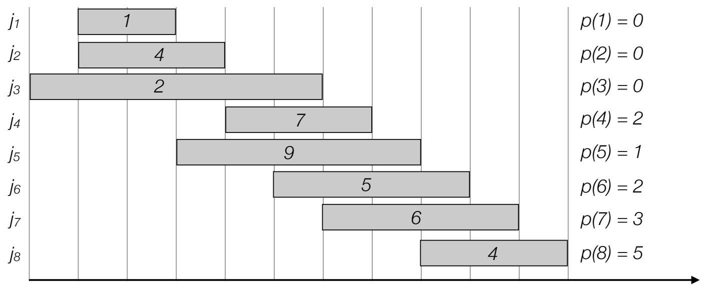
Weighted interval scheduling: memoization

```
Input: n, s[1..n], f[1..n], v[1..n]

Sort jobs by finish time 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.
Compute-Memoized-Opt(n)

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



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

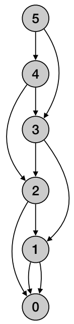
i	M[i]
0	0
1	1
2	4
3	4
4	11
5	11
6	11
7	11
8	15

Weighted interval scheduling: bottom-up

```
Compute-Bottom-Up–Opt(n, s[1..n], f[1..n], v[1..n])

Sort jobs by finish time so that f[1] ≤ f[2]≤ … ≤ f[n]
Compute p[1], p[2], …, p[n]

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



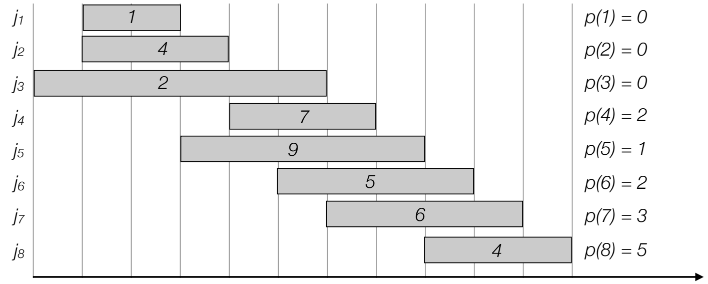
- Running time O(n log n):
  - Sorting takes O(n log n) time.
  - Computing p(n): O(n log n)
  - For loop: O(n) time
    - Each iteration takes constant time.
- Space O(n)

Weighted interval scheduling: bottom-up

```
Compute-Bottom-Up–Opt(n, s[1..n], f[1..n], v[1..n])

Sort jobs by finish time so that f[1] ≤ f[2]≤ … ≤ f[n]
Compute p[1], p[2], …, p[n]

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

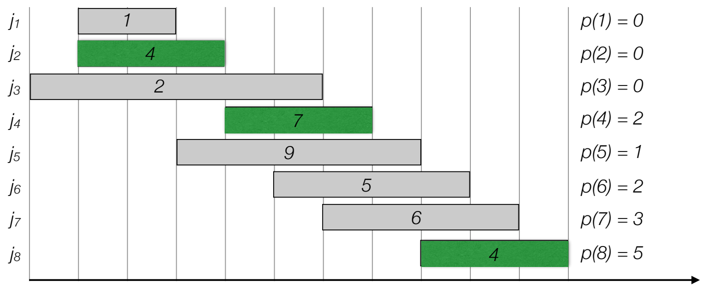


i	M[i]
0	0
1	1
2	4
3	4
4	11
5	11
6	11
7	11
8	15

Weighted interval scheduling: find solution

```
Find-Solution(j)
if j=0
  Return emptyset
else if M[j] > M[j-1]
  return {j} U Find-Solution(p[j])
else
  return Find-Solution(j-1)
```

Solution = 8 , 4 , 2



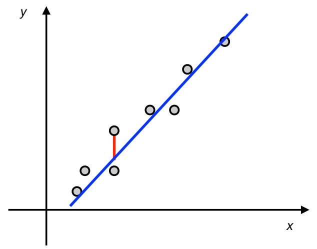
i	M[i]
0	0
1	4
2	4
3	4
4	11
5	11
6	11
7	11
8	15

Segmented Least Squares

Least squares

- Least squares.
  - Given n points in the plane: (x_1,y_1), (x_2,y_2), …, (x_n,y_n).
  - Find a line $y = ax + b$ that minimizes the sum of the squared error:

$$SSE = \sum_{i=1}^{n} (y_i - ax_i - b)^2$$

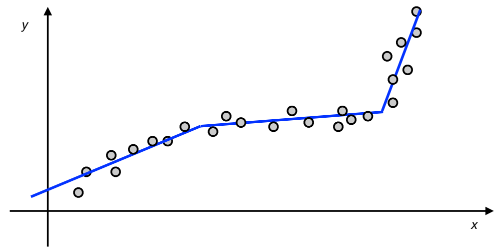


- Solution. Calculus => minimum error is achieved when

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

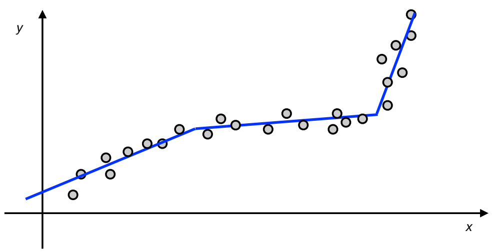
Segmented least squares

- Segmented least squares
  - Points lie roughly on a *sequence* of line segments.
  - Given n points in the plane (x_1,y_1), (x_2,y_2), …, (x_n,y_n).
  - Find a sequence of lines that minimizes some function $f(x)$.
- What is a good choice for $f(x)$ that balance accuracy and number of lines?



Segmented least squares

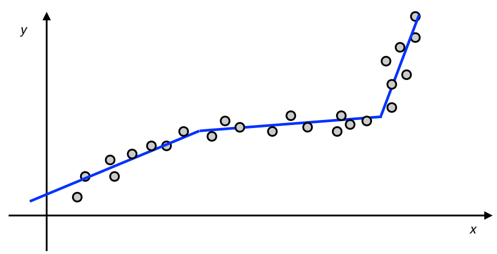
- Segmented least squares. Given n points in the plane (x_1,y_1), (x_2,y_2), …, (x_n,y_n) and a constant $c > 0$ find a sequence of lines that minimizes $f(x) = E + cL$:
  - E = sum of sums of the squared errors in each segment.
  - L = number of lines



Dynamic programming: multiway choice

- 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 = $e(i,j) + c + OPT(i-1)$.

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



Segmented least squares algorithm

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

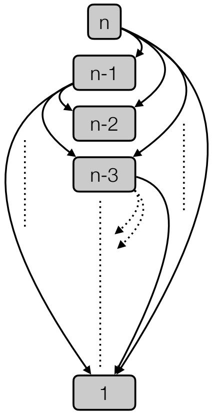
```
Segmented-least-squares(n, p₁, p₂, …,pₙ,c)

for j=1 to n
  for i=1 to j
    Compute the least squares e(i,j) for the segment
    pᵢ, pᵢ₊₁, …,pⱼ.

M[0] = 0.
for j=1 to n
  M[j] = ∞
  for i=1 to j
    M[j] = min(M[j],e(i,j) + c + M[i-1])

Return M[n]
```


Subproblem dag



Segmented least squares algorithm

- Time.
  - O(n^3) for computing $e(i,j)$ for O(n^2) pairs (O(n) per pair).
  - O(n^2) for computing M.
  - Total O(n^3)
- Space
  - O(n^2).

```
Segmented-least-squares(n, p₁, p₂, …,pₙ,c)

for j=1 to n
  for i=1 to j
    Compute the least squares e(i,j) for the segment
    pᵢ, pᵢ₊₁, …,pⱼ.

M[0] = 0.
for j=1 to n
  M[j] = ∞
  for i=1 to j
    M[j] = min(M[j],e(i,j) + c + M[i-1])

Return M[n]
```


Dynamic programming

- **First formulate the problem recursively.**
  - Describe the *problem* recursively in a clear and precise way.
  - Give a recursive formula for the problem.
- **Bottom-up**
  - Identify all the subproblems.
  - Choose a memoization data structure.
  - Identify dependencies.
  - Find a good evaluation order.
- **Top-down**
  - Identify all the subproblems.
  - Choose a memoization data structure.
  - Identify base cases.