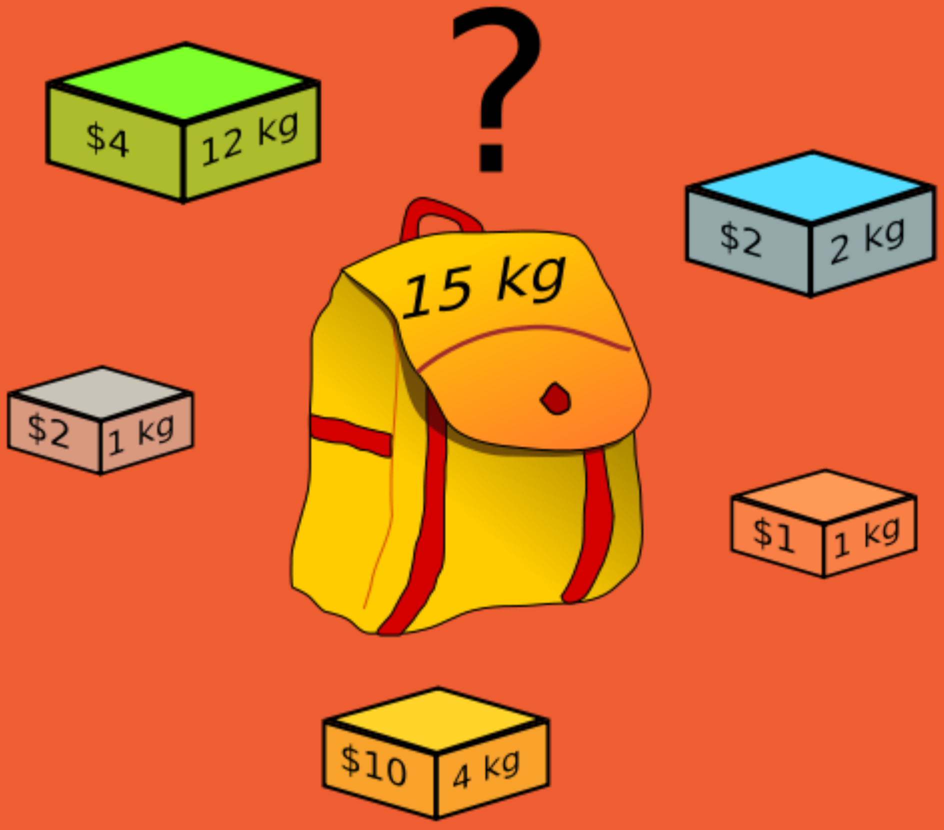
Lecture 5: Dynamic Programming II



William Umboh
School of Computer Science

THE UNIVERSITY OF SYDNEY

Next Assignment and Quiz

- Quiz 4 out today and due 11 Apr
- A2 out next Thursday and due 21 Apr 23:59

Main Idea: Dynamic Programming

Recurrence equation relating optimal solution in terms optimal solutions to smaller subproblems

Given optimal solutions to smaller subproblems, how to construct solution to original problem?

Key steps: Dynamic programming

Formulate the problem recursively.

1. Define subproblems
2. Find recurrence relating subproblems
3. Solve the base cases

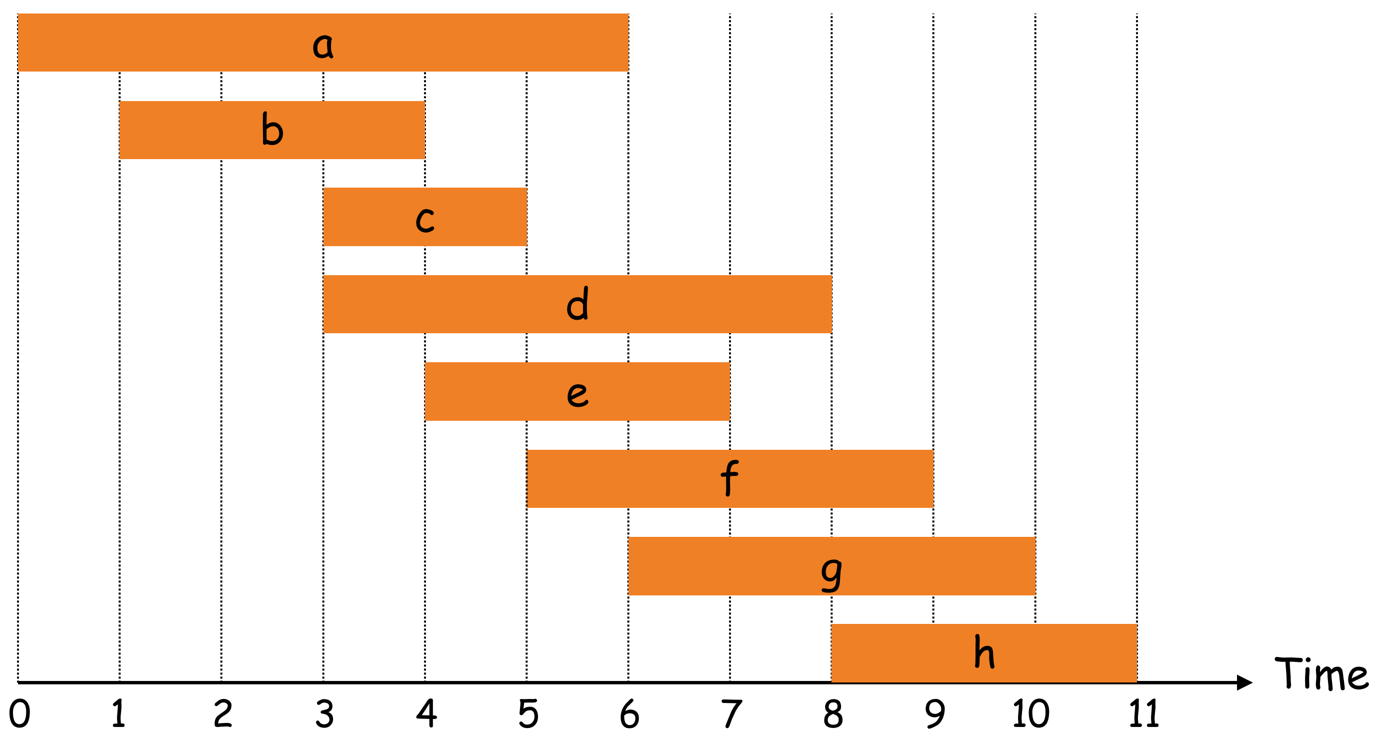
Similar to what we did for D&C

Transform recurrence into an efficient algorithm

- Data structure to store solutions to subproblems
- Evaluation order of subproblems

Recap: Weighted Interval Scheduling

- Job j starts at s_j, finishes at f_j, and has weight v_j.
- Two jobs compatible if they don't overlap.
- Goal: find maximum weight subset of mutually compatible jobs.

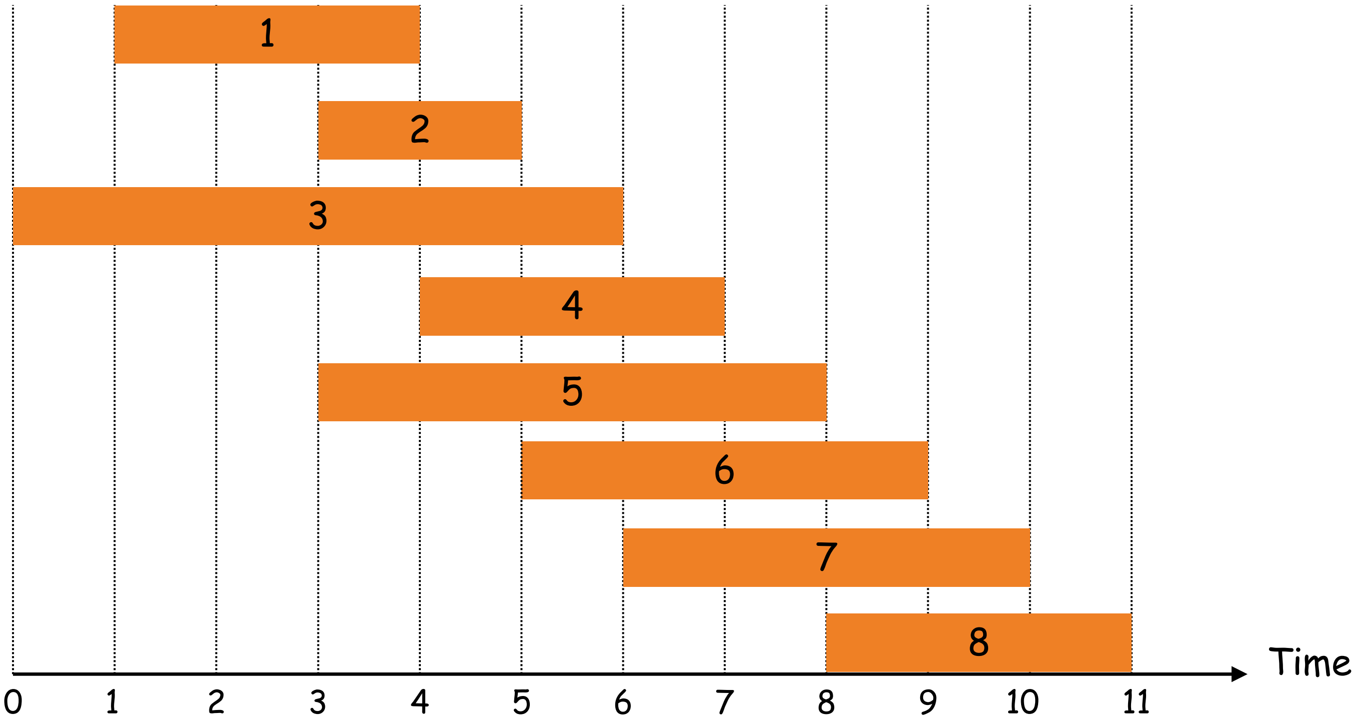


Recap: Weighted Interval Scheduling

Notation. Label 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.



Recap: Dynamic Programming – Step 1

Step 1: Define subproblems

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

Recap: Dynamic Programming – Step 2

Step 2: Find recurrences

- **Case 1:** OPT 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)

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

$$OPT(j) = \max \{\underbrace{v_j + OPT(p(j))}_{\text{Case 1}}, \underbrace{OPT(j-1)}_{\text{Case 2}}\}$$

Recap: Dynamic Programming – Step 3

Step 3: **Solve the base cases**

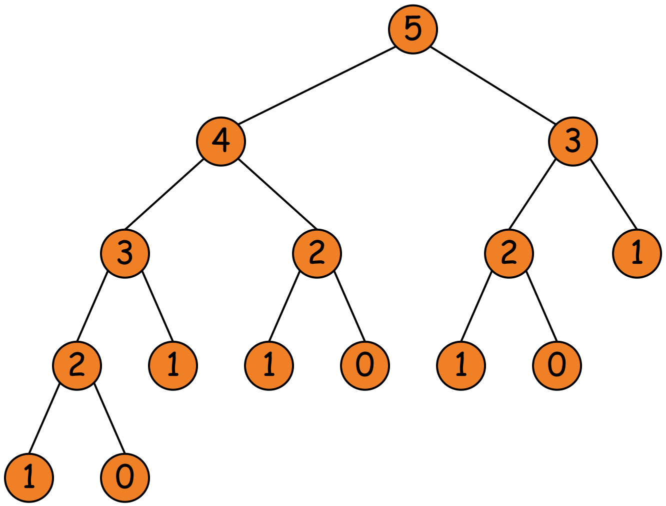
$$OPT(0) = 0$$

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

Done…more or less

Recap: Naïve Recursion is Exponential

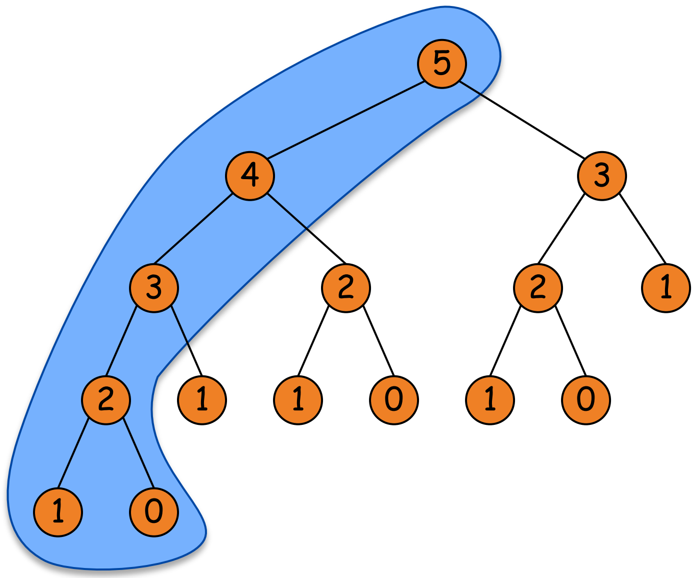
$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max \{ v_j + OPT(p(j)),\ OPT(j-1) \} & \text{otherwise} \end{cases}$$



Could get an exponential number of subproblems!

Recap: Memoization

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



Could get an exponential number of subproblems!

Memoization: Instead of recomputing every subproblem store the results of each sub-problem.

Recap: Bottom-up

Bottom-up Dynamic Programming. Unwind recursion.

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

$\downarrow$

Value of Optimal solution = OPT[n]

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

$\underbrace{p(j)}_{<j}$ $\underbrace{j-1}_{<j}$

Time: O(n)

Recap: Finding a Solution

Question. Dynamic programming algorithm computes optimal value.

What if we want the solution itself?
Answer. Do some post-processing.

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

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

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

Longest Common Subsequence

Given two sequences X[1..n] and Y[1..m], find the longest subsequences X' of X and Y' of Y such that X' = Y'.

<u>Example 1</u> X = BANANAS

Y = KATANA

LCS(X,Y) = AANA

<u>Example 2</u> X = DYNAMIC

Y = PROGRAMMING

LCS(X,Y) = AMI

Longest Common Subsequence: Matching Definition

Given two sequences X[1..n] and Y[1..m], find the longest non-crossing matching between elements of X and elements of Y

can only match elements that are the same.

<u>Example 1</u> X = BANANAS
Y = KATANA

LCS(X,Y) = AANA. Matching: X[2] – Y[2], X[4] – Y[4], X[5] – Y[5], X[6] – Y[6].

X = AB

Y = ST

LCS = ""

<u>Example 2</u> X = DYNAMIC
Y = PROGRAMMING

LCS(X,Y) = AMI. Matching: X[4] – Y[6], X[5] – Y[7], X[6] – Y[8],

Note: if crossings allowed, then can add X[3] – Y[10].

Longest Common Subsequence: Applications

- Measures similarity of two strings
- Bioinformatics
- Merging in version control

LCS Dynamic Programming: Step 1

Step 1: Define subproblems (first try)

OPT(i) = length of LCS(X[1..i], Y[1..i]).

If X[i] = Y[i], then can match them and recurse on LCS(X[1..i-1], Y[1..i-1])

But what if X[i] ≠ Y[i]?

Does not work if $m \neq n$

[illegible]

LCS Dynamic Programming: Step 1

Step 1: Define subproblems

OPT(i,j) = length of LCS(X[1..i], Y[1..j]).

Subproblems: finding LCS of prefixes of X and Y

<u>Example</u>

X = BANANAS

Y = KATANA

OPT(6, 4) = length of LCS(BANANA, KATA) = 2

LCS Dynamic Programming: Step 2

Notations:

– OPT(i,j) = length of LCS(X[1..i], Y[1..j]).

Step 2: Finding recurrences

– Case 1: X[i] ≠ Y[j].
  – Leave X[i] unmatched and take LCS(X[1..i-1], Y[1..j]). OR
  – Leave Y[j] unmatched and take LCS(X[1..i], Y[1..j-1]).
  – OPT(i,j)= max{OPT(i-1,j), OPT(i,j-1)}

Example

X[1..5] = BANAN

Y[1..6] = KATANA

X[5] unmatched: LCS = LCS(X[1..4], Y[1..6]) = ANA

Y[6] unmatched: LCS = LCS(X[1..5], Y[1..5]) = AAN

LCS Dynamic Programming: Step 2

Notations:

– OPT(i,j) = length of LCS(X[1..i], Y[1..j]).

Step 2: Finding recurrences

– Case 1: X[i] ≠ Y[j].
  – Leave X[i] unmatched and take LCS(X[1..i-1], Y[1..j]). OR
  – Leave Y[j] unmatched and take LCS(X[1..i], Y[1..j-1]).
  – OPT(i,j)= max{OPT(i-1,j), OPT(i,j-1)}

Example

X[1..5] = BANAN

Y[1..6] = KATANA

X[5] unmatched: LCS = LCS(X[1..4], Y[1..6]) = ANA

Y[6] unmatched: LCS = LCS(X[1..5], Y[1..5]) = AAN

LCS Dynamic Programming: Step 2

Notations:

- OPT(i,j) = length of LCS(X[1..i], Y[1..j]).

Step 2: **Finding recurrences**

- Case 1: X[i] ≠ Y[j].
  - Leave X[i] unmatched and take LCS(X[1..i-1], Y[1..j]). OR
  - Leave Y[j] unmatched and take LCS(X[1..i], Y[1..j-1]).
  - OPT(i,j)= max{OPT(i-1,j), OPT(i,j-1)}

Example

X[1..3] = BAN

Y[1..6] = KATANA

X[3] unmatched: LCS(X[1..2], Y[1..6]) = A

Y[6] unmatched: LCS(X[1..3], Y[1..5]) = AN

LCS Dynamic Programming: Step 2

Notations:

- OPT(i,j) = length of LCS(X[1..i], Y[1..j]).

Step 2: **Finding recurrences**

- Case 1: X[i] ≠ Y[j].
  - Leave X[i] unmatched and take LCS(X[1..i-1], Y[1..j]). OR
  - Leave Y[j] unmatched and take LCS(X[1..i], Y[1..j-1]).
  - OPT(i,j)= max{OPT(i-1,j), OPT(i,j-1)}

<u>Example</u>

X[1..3] = BAN

Y[1..6] = KATANA

X[3] unmatched: LCS(X[1..2], Y[1..6]) = A

Y[6] unmatched: LCS(X[1..3], Y[1..5]) = AN

LCS Dynamic Programming: Step 2

Notations:

– OPT(i,j) = length of LCS(X[1..i], Y[1..j]).

Step 2: Finding recurrences

– Case 2: X[i] = Y[j].
  – Match X[i] to Y[j] and take LCS(X[1..i-1], Y[1..j-1]) + X[i] OR
  – Leave X[i] unmatched or Y[j] unmatched
  – OPT(i,j) = max{OPT(i-1, j-1) + 1, OPT(i-1, j), OPT(i, j-1)

Example

X[1..6] = BANANA

Y[1..6] = KATANA

Match X[6] – Y[6]: LCS = LCS(X[1..5], Y[1..5]) + A = AANA

X[6] unmatched: LCS = LCS(X[1..5], Y[1..6]) = AAN

Y[6] unmatched: LCS = LCS(X[1..6], Y[1..5]) = AAN

LCS Dynamic Programming: Step 3

Step 3: Solving the base cases

OPT(i,0) = 0 for all i, OPT(0,j) = 0 for all j

$$OPT(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ \max\{OPT(i-1,j-1)+1, OPT(i,j-1), OPT(i-1,j)\} & \text{if } X[i] = Y[j] \\ \max\{OPT(i,j-1), OPT(i-1,j)\} & \text{if } X[i] \neq Y[j] \end{cases}$$

LCS Dynamic Programming: Algorithm

```
INPUT: n, m, X[1..n], Y[1..m]

for j = 0 to m
   M[0,j] = 0
for i = 0 to n
   M[i,0] = 0

for i = 1 to n
   for j = 1 to m
      if (X[i] = Y[j])
         M[i,j] = max(M[i-1,j-1] + 1, M[i-1,j], M[i,j-1])
      else
         M[i,j] = max(M[i-1,j], M[i,j-1])

return M[n,m]
```

base cases

recursive cases

$$OPT(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ \max\{OPT(i-1,j-1)+1, OPT(i,j-1), OPT(i-1,j)\} & \text{if } X[i] = Y[j] \\ \max\{OPT(i,j-1), OPT(i-1,j)\} & \text{if } X[i] \neq Y[j] \end{cases}$$

LCS Dynamic Programming: Analysis

```
INPUT: n, m, X[1..n], Y[1..m]

for j = 0 to m
   M[0,j] = 0
for i = 0 to n
   M[i,0] = 0

for i = 1 to n
   for j = 1 to m
      if (X[i] = Y[j])
         M[i,j] = max(M[i-1,j-1] + 1, M[i-1,j], M[i,j-1])
      else
         M[i,j] = max(M[i-1,j], M[i,j-1])

return M[n,m]
```

O(m)

O(n)

O(n + m)

O(nm)

Running time: O(nm)

Space: O(nm)

See 3927 Lecture 4 for Sequence Alignment

6.5 RNA Secondary Structure

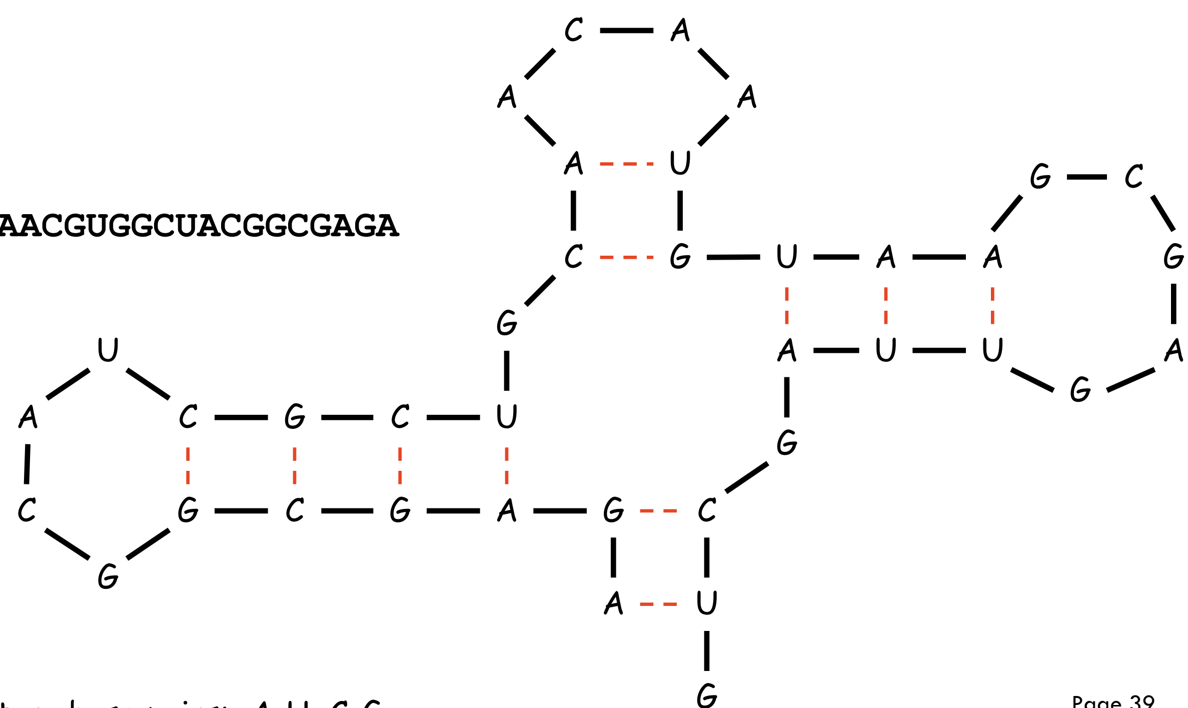
Dynamic programming over intervals

RNA (Ribonucleic acid) Secondary Structure

- **RNA.** String $B = b_1 b_2 \ldots b_n$ over alphabet { A, C, G, U }.

- **Secondary structure.** RNA is single-stranded so it tends to loop back and form base pairs with itself. This structure is essential for understanding behavior of molecule.

Ex: GUCGAUUGAGCGAAUGUAACAACGUGGCUACGGCGAGA



complementary base pairs: A-U, C-G

RNA Secondary Structure

- **Secondary structure.** A set of pairs $S = \{ (b_i, b_j) \}$ that satisfy:
  - [Watson-Crick.] S is a matching and each pair in S is a Watson-Crick complement: A-U, U-A, C-G, or G-C.
  - [No sharp turns.] The ends of each pair are separated by at least 4 intervening bases. If $(b_i, b_j) \in S$, then $i < j - 4$.
  - [Non-crossing.] If (b_i, b_j) and (b_k, b_l) are two pairs in S, then we cannot have $i < k < j < l$.

- **Free energy.** Usual hypothesis is that an RNA molecule will form the secondary structure with the optimum total free energy.

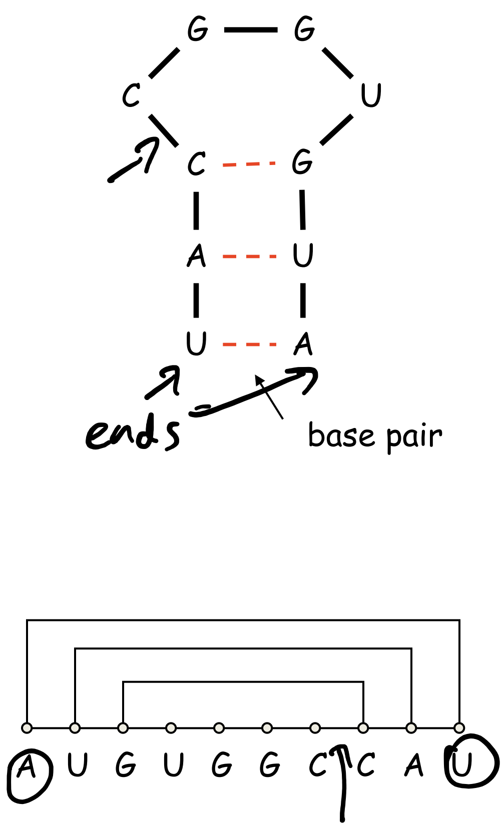
approximated by number of base pairs

- **Goal.** Given an RNA molecule $B = b_1 b_2 \ldots b_n$, find a secondary structure S that maximizes the number of base pairs.

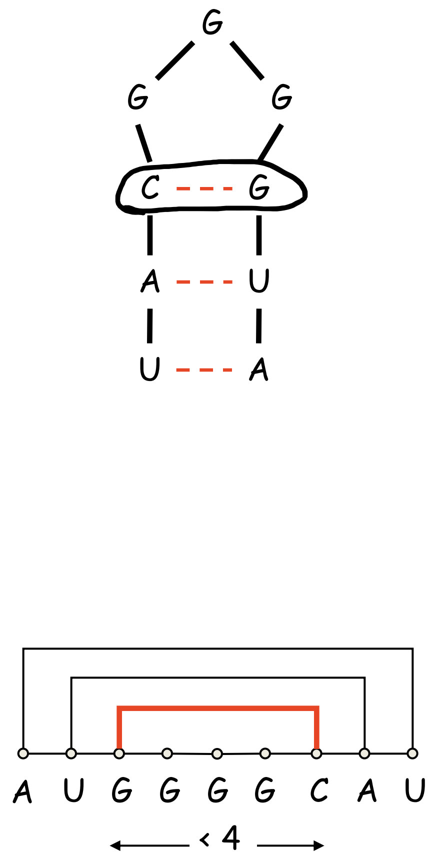
RNA Secondary Structure: Examples

Pairs (C, G)
(A, U)
(U, A)

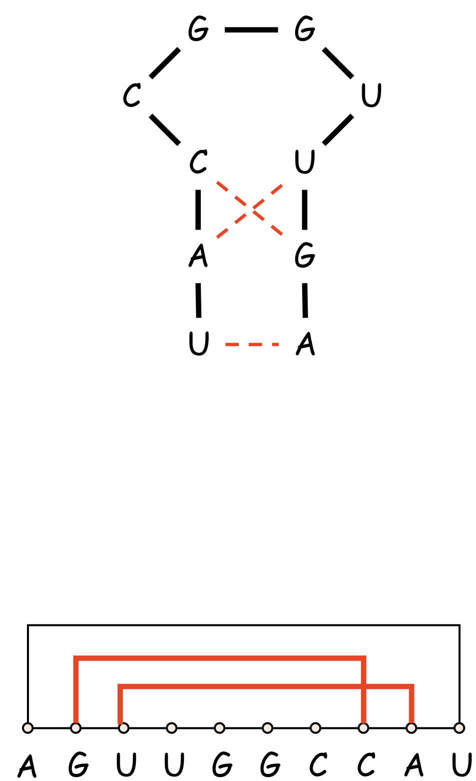
ε is an end of pair (c, g)



ok



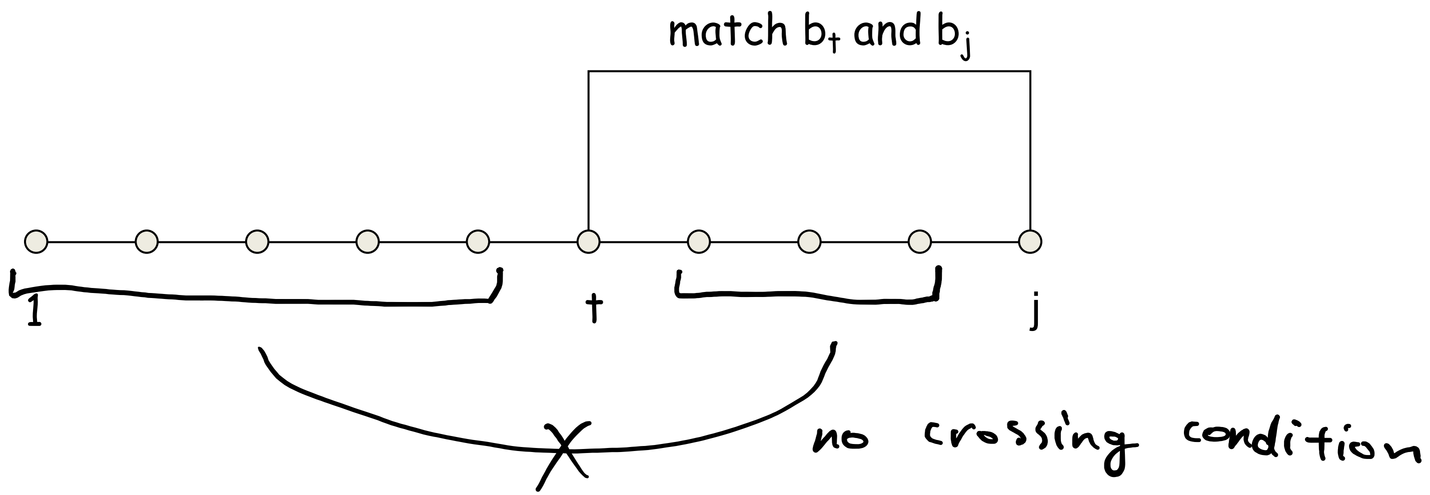
sharp turn



crossing

RNA Secondary Structure: Subproblems

- **First attempt (Step 1).** OPT(j) = maximum number of base pairs in a secondary structure of the substring $b_1 b_2 \ldots b_j$.



- **Difficulty (in Step 2).** Results in two sub-problems.
  - Finding secondary structure in: $b_1 b_2 \ldots b_{t-1}$. ← OPT(t-1)
  - Finding secondary structure in: $b_{t+1} b_{t+2} \ldots b_{j-1}$. → ??

need more sub-problems

Dynamic Programming Over Intervals – Step 1

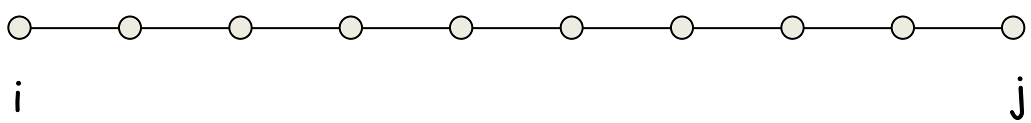
Step 1: Define subproblems

OPT(i, j) = maximum number of base pairs in a secondary structure of the substring $b_i b_{i+1} \ldots b_j$.

Dynamic Programming Over Intervals – Step 2

Notation. OPT(i, j) = maximum number of base pairs in a secondary structure of the substring $b_i b_{i+1} \ldots b_j$.

Step 2: Find recurrences



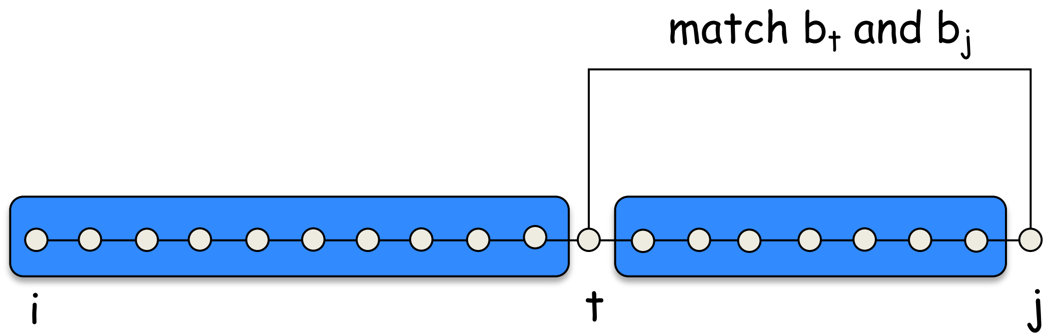
Case 1. Base b_j is not involved in a pair.

OPT(i, j) = OPT(i, j-1)

Dynamic Programming Over Intervals – Step 2

Notation. OPT(i, j) = maximum number of base pairs in a secondary structure of the substring $b_i b_{i+1} \ldots b_j$.

Step 2: Find recurrences



Case 2. Base b_j pairs with b_t for some $i \le t < j - 4$.

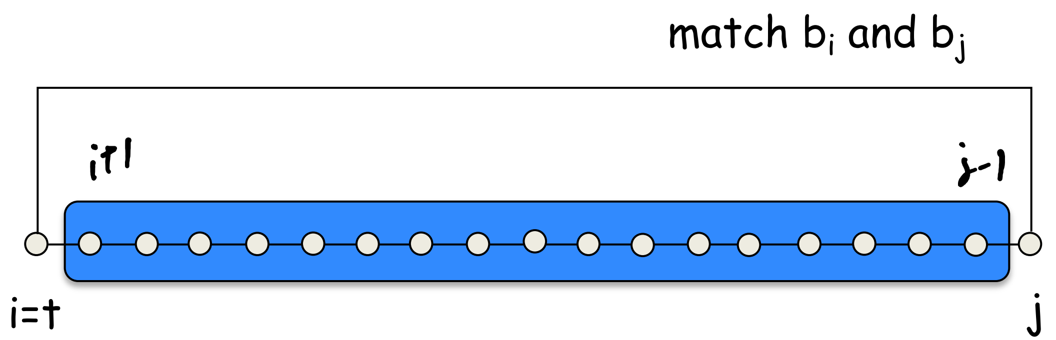
non-crossing constraint decouples resulting sub-problems

$$OPT(i, j) = 1 + \max_{i \le t < j-4} \{ OPT(i, t-1) + OPT(t+1, j-1) \}$$

Dynamic Programming Over Intervals – Step 2

Notation. OPT(i, j) = maximum number of base pairs in a secondary structure of the substring $b_i b_{i+1} \ldots b_j$.

Step 2: Find recurrences



no sharp turns

Case 2. Base b_j pairs with b_t for some $i \le t < j - 4$.

non-crossing constraint decouples resulting sub-problems

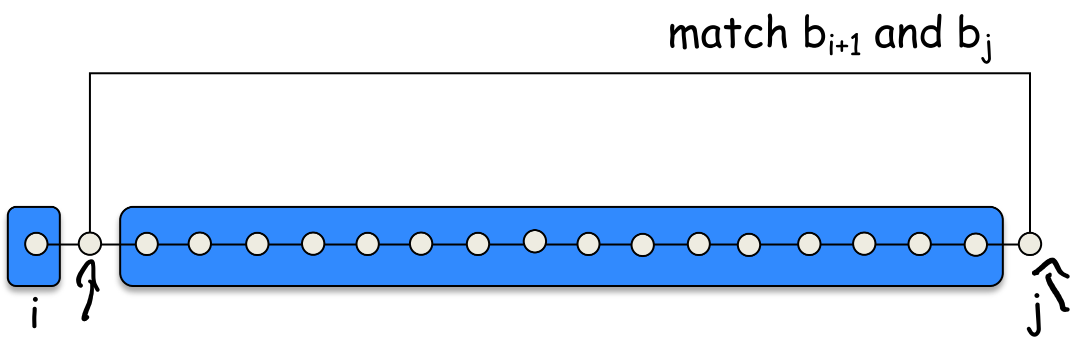
$$OPT(i, j) = 1 + \max_{i \le t < j-4} \{ OPT(i, t-1) + OPT(t+1, j-1) \}$$

Dynamic Programming Over Intervals – Step 2

Notation. $OPT(i, j)$ = maximum number of base pairs in a secondary structure of the substring $b_i b_{i+1} \ldots b_j$.

if i, j matched $OPT(i, j) = 1 + OPT(i+1, j-1)$

Step 2: Find recurrences if $i+1, j$ matched $OPT(i,j) = 1 + OPT(i,i) + OPT(i+2, j-1)$



Case 2. Base b_j pairs with b_t for some $i \le t < j - 4$.

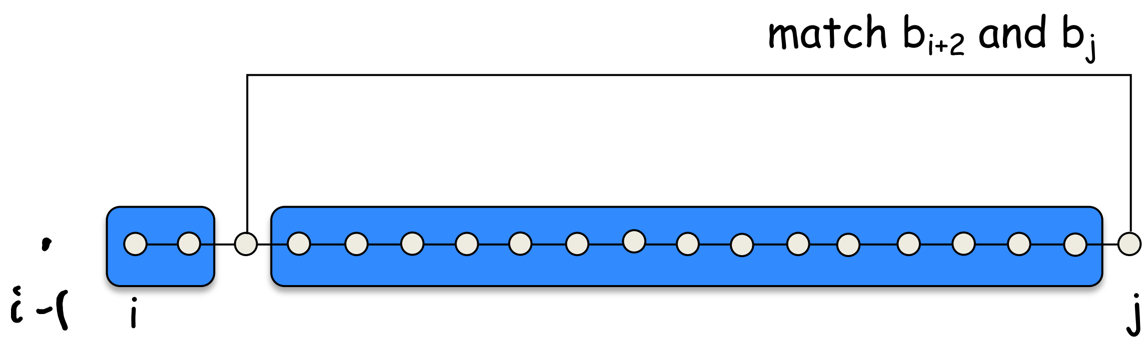
non-crossing constraint decouples resulting sub-problems

$$OPT(i, j) = 1 + \max_{i \le t < j-4} \{ OPT(i, t-1) + OPT(t+1, j-1) \}$$

Dynamic Programming Over Intervals – Step 2

Notation. OPT(i, j) = maximum number of base pairs in a secondary structure of the substring $b_i b_{i+1} \ldots b_j$.

Step 2: Find recurrences



Case 2. Base b_j pairs with b_t for some $i \leq t < j - 4$.

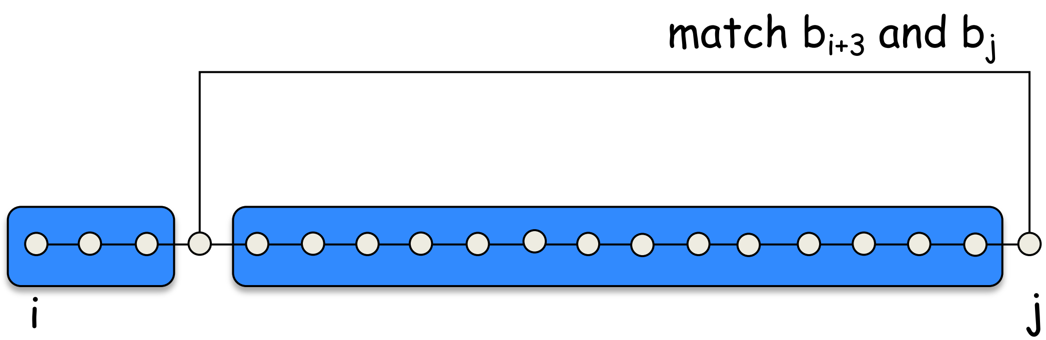
non-crossing constraint decouples resulting sub-problems

$$OPT(i, j) = 1 + \max_{i \leq t < j-4} \{ OPT(i, t-1) + OPT(t+1, j-1) \}$$

Dynamic Programming Over Intervals – Step 2

Notation. OPT(i, j) = maximum number of base pairs in a secondary structure of the substring $b_i b_{i+1} \ldots b_j$.

Step 2: Find recurrences



Case 2. Base b_j pairs with b_t for some $i \leq t < j - 4$.

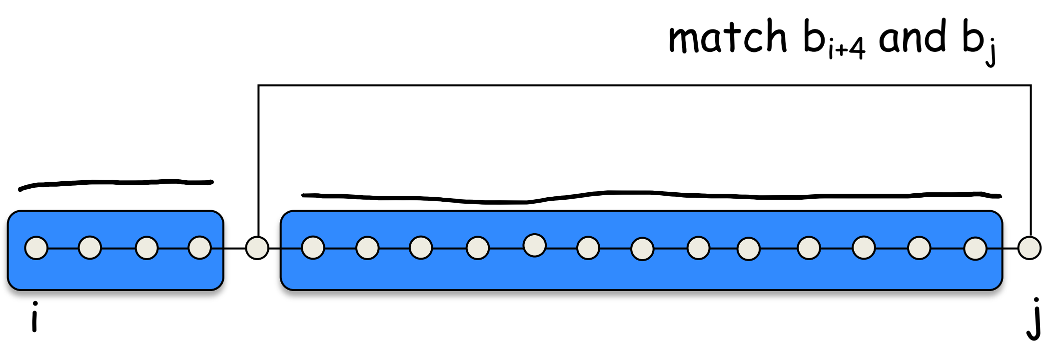
non-crossing constraint decouples resulting sub-problems

$$OPT(i, j) = 1 + \max_{i \leq t < j-4} \{ OPT(i, t-1) + OPT(t+1, j-1) \}$$

Dynamic Programming Over Intervals – Step 2

Notation. OPT(i, j) = maximum number of base pairs in a secondary structure of the substring $b_i b_{i+1} \ldots b_j$.

Step 2: Find recurrences



Case 2. Base b_j pairs with b_t for some $i \leq t < j - 4$.

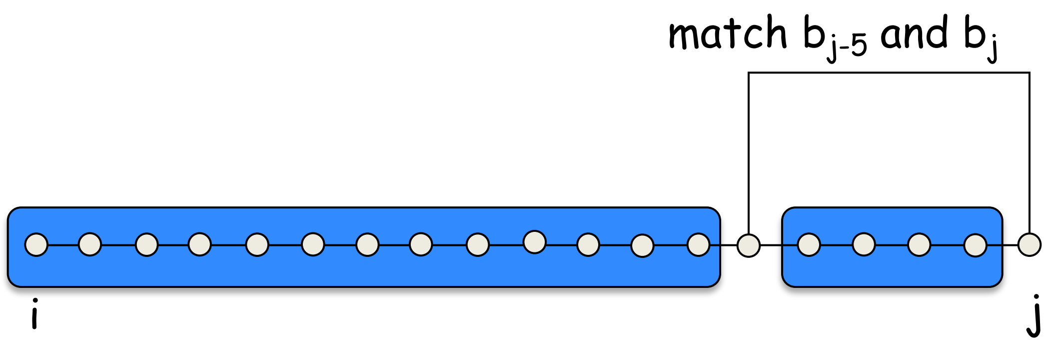
non-crossing constraint decouples resulting sub-problems

$$OPT(i, j) = 1 + \max_{i \leq t < j-4} \{ OPT(i, t-1) + OPT(t+1, j-1) \}$$

Dynamic Programming Over Intervals – Step 2

Notation. OPT(i, j) = maximum number of base pairs in a secondary structure of the substring $b_i b_{i+1} \ldots b_j$.

Step 2: Find recurrences



Case 2. Base b_j pairs with b_t for some $i \le t < j - 4$.

non-crossing constraint decouples resulting sub-problems

$$OPT(i, j) = 1 + \max_{i \le t < j-4} \{ OPT(i, t-1) + OPT(t+1, j-1) \}$$

Dynamic Programming Over Intervals – Step 2

Notation. $OPT(i, j)$ = maximum number of base pairs in a secondary structure of the substring $b_i b_{i+1} \ldots b_j$.

Step 2: Find recurrences

Case 1. Base b_j is not involved in a pair.

- $OPT(i, j) = OPT(i, j-1)$

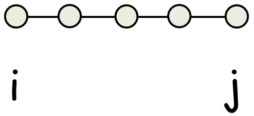
Case 2. Base b_j pairs with b_t for some $i \leq t < j - 4$.

- non-crossing constraint decouples resulting sub-problems
- $OPT(i, j) = 1 + \max_{i \leq t < j-4} \{ OPT(i, t-1) + OPT(t+1, j-1) \}$

Step 3: Find base cases ?

Dynamic Programming Over Intervals – Step 3

Step 3: Solve the base cases



If $i \geq j - 4$ then

$OPT(i, j) = 0$ by no-sharp turns condition.

Dynamic Programming Over Intervals – 3 Steps

Step 1: $OPT(i, j)$ = maximum number of base pairs in a secondary structure of the substring $b_i b_{i+1} \ldots b_j$.

Step 2:

Case 1. Base b_j is not involved in a pair.

- $OPT(i, j) = OPT(i, j-1)$

Case 2. Base b_j pairs with b_t for some $i \leq t < j - 4$.

- non-crossing constraint decouples resulting sub-problems
- $OPT(i, j) = 1 + \max_{i \leq t < j-4} \{ OPT(i, t-1) + OPT(t+1, j-1) \}$

Step 3:

Base case. If $i \geq j - 4$.

- $OPT(i, j) = 0$ by no-sharp turns condition.

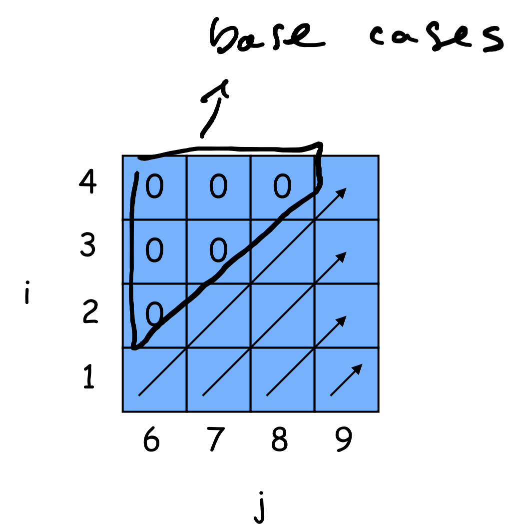
Bottom-Up Dynamic Programming Over Intervals

$$\mathrm{OPT}(i, j) = \max\{\mathrm{OPT}(i, j-1),\ 1 + \max_{i \le t < j-4} \{ \mathrm{OPT}(i, t-1) + \mathrm{OPT}(t+1, j-1) \}$$

- **Question:** What order to solve the sub-problems?
- **Answer:** Do shortest intervals first.

```
RNA(1,n) {
   for k = 5, 6, …, n-1
      for i = 1, 2, …, n-k
         j = i + k
         Compute OPT[i,j]

   return OPT[1,n]
}
```



using the recurrence

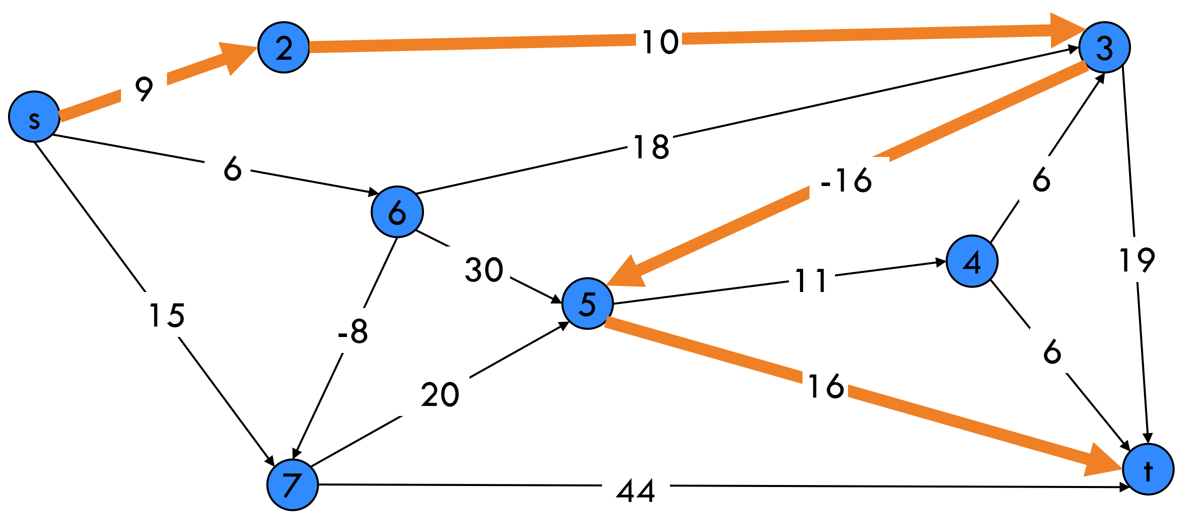
- **Running time:** $O(n^3)$

6.8 Shortest Paths

Shortest Paths

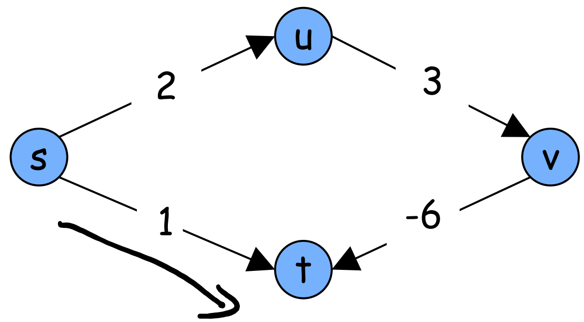
- **Shortest path problem.** Given a directed graph G = (V, E), with edge weights c_{vw}, find shortest path from node s to node t.

allow negative weights

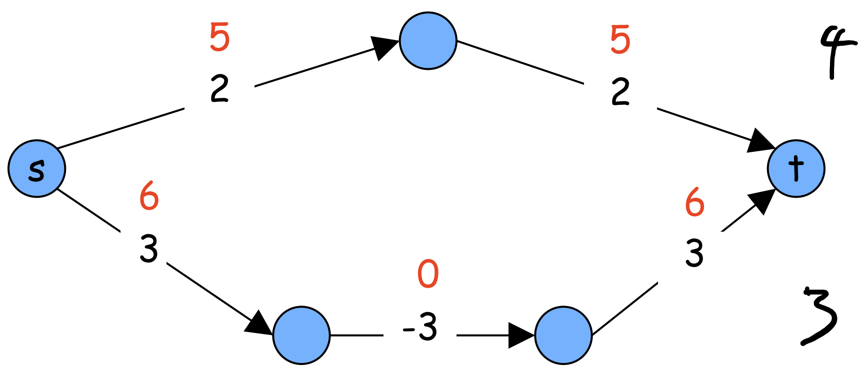


Shortest Paths: Failed Attempts

- **Dijkstra.** Can fail if negative edge costs.



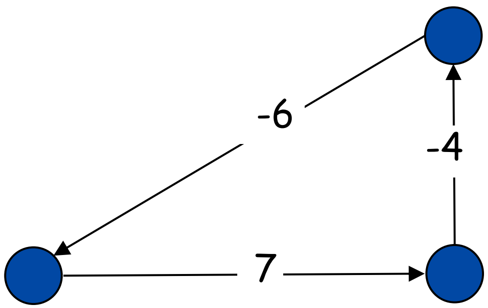
- **Re-weighting.** Adding a constant to every edge weight can fail.



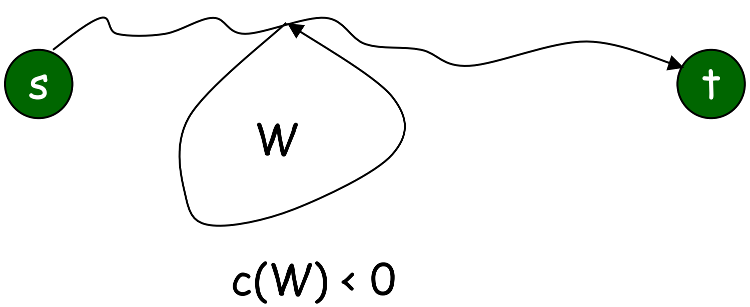
Paths with more edges are penalized more

Shortest Paths: Negative Cost Cycles

- **Negative cost cycle.**



- **Observation.** If some path from s to t contains a negative cost cycle, there does not exist a shortest s-t path; otherwise, there exists one that is simple and thus has at most n – 1 edges.

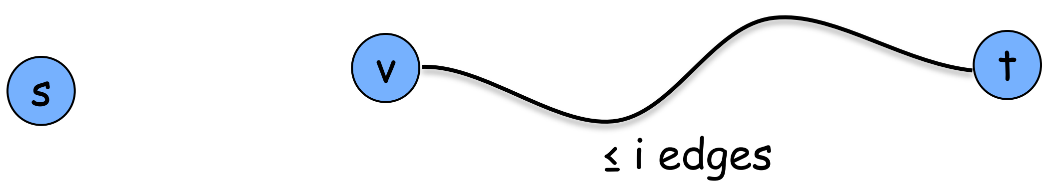


Shortest Paths: Dynamic Programming

Problem: Find shortest path from s to t

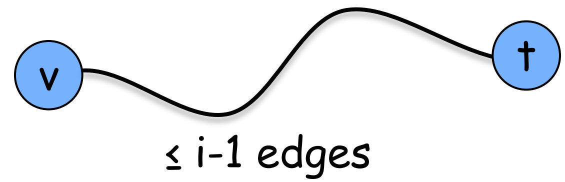
Step 1: Define subproblems

OPT(i, v) = length of shortest v-t path P using at most i edges.



Shortest Paths: Dynamic Programming

Step 2: Find recurrences



Case 1: P uses at most i-1 edges.

- OPT(i, v) = OPT(i-1, v)

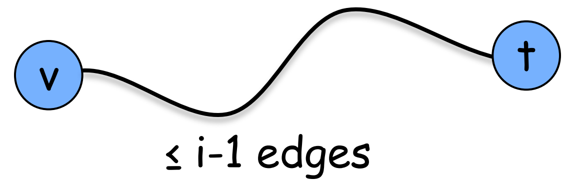
Shortest Paths: Dynamic Programming

$OPT(n-1, s)$
$=$ length of shortest s-t path

Step 2: Find recurrences

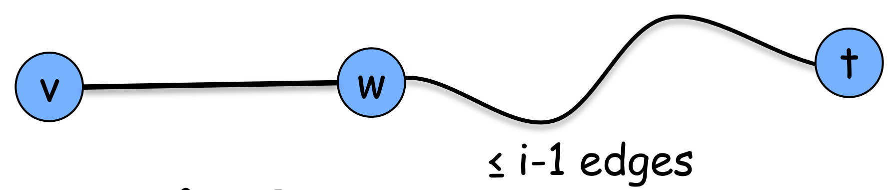
Case 1: P uses at most i-1 edges.

- OPT(i, v) = OPT(i-1, v)



Case 2: P uses exactly i edges.

- if (v, w) is first edge, then OPT uses (v, w), and then selects best w-t path using at most i-1 edges



$$OPT(i,v) = \min\left\{\overbrace{OPT(i-1,v)}^{\text{case 1}},\ \overbrace{\min_{(v,w)\in E}\left[\underline{OPT(i-1,w)+c_{vw}}\right]}^{\text{case 2}}\right\}$$

Shortest Paths: Dynamic Programming

Step 3: Solve the base cases

$$OPT(0,t) = 0 \text{ and } OPT(0,v \neq t) = \infty$$

Shortest Paths: Dynamic Programming

Step 1: OPT(i, v) = length of shortest v-t path P using at most i edges.

Step 2:

Case 1: P uses at most i-1 edges.

- OPT(i, v) = OPT(i-1, v)

Case 2: P uses exactly i edges.

- if (v, w) is first edge, then OPT uses (v, w), and then selects best w-t path using at most i-1 edges

Step 3: OPT(0,t) = 0 and OPT(0,v≠t) = ∞

$$OPT(i,v) = \begin{cases} 0 & \text{if } i=0 \text{ and } v=t \\ \infty & \text{if } i=0 \text{ and } v\neq t \\ \min\{OPT(i-1,v),\ \min\limits_{(v,w)\in E} [OPT(i-1,w)+c_{vw}]\} & \text{otherwise} \end{cases}$$

base cases

recursive case.

n = # vertices
m = # edges

Shortest Paths: Implementation

M = O(n²)

```
Shortest-Path(G, t) {
   foreach node v ∈ V
      M[0, v] ← ∞
   M[0, t] ← 0

   for i = 1 to n-1
      foreach node v ∈ V
         M[i, v] ← M[i-1, v]
         foreach edge (v, w) ∈ E
            M[i, v] ← min { M[i, v], M[i-1, w] + c_vw }
}
```

O(n) time.

O(n) iterations

O(m) iterations

Space used by algorithm in addition to input

- **Analysis.** $\Theta(mn)$ time, $\Theta(n^2)$ working space.

- **Finding the shortest paths.** Maintain a "successor" for each table entry. Successor(i,v) = next vertex on shortest v-t path with at most i edges.

Shortest Paths: Efficient Implementation

```
Shortest-Path(G, t) {
   foreach node v ∈ V
      M[0, v] ← ∞
   M[0, t] ← 0

   for i = 1 to n-1
      foreach node v ∈ V
         M[i, v] ← M[i-1, v]
         foreach edge (v, w) ∈ E
            M[i, v] ← min { M[i, v], M[i-1, w] + c_vw }
}
```

In iteration i, only need M[i-1, *] values

- **Analysis.** Θ(mn) time, Θ(n) working space.
- **Finding the shortest paths.** Maintain a "successor" for vertex. In the i-th iteration, Successor(v) = next vertex on shortest v-t path with at most i edges.

Bellman-Ford: Efficient Implementation

```
Push-Based-Shortest-Path(G, s, t) {
   foreach node v ∈ V {
      M[v] ← ∞
      successor[v] ← ∅ }

   M[t] = 0
   for i = 1 to n-1 {
      foreach node w ∈ V {
      if (M[w] has been updated in previous iteration) {
         foreach node v such that (v, w) ∈ E {
            if (M[v] > M[w] + c_vw) {
               M[v] ← M[w] + c_vw
               successor[v] ← w
            }
         }
      }
      If no M[w] value changed in iteration i, stop.
   }
}
```

Analysis: $\Theta(mn)$ time, $\Theta(n)$ working space.

Shortest Paths: Practical Improvements

$OPT(i, w)$

$OPT(i-1, \cdot)$

- Practical improvements
  - Maintain only one array M[v] = shortest v-t path that we have found so far.
  - No need to check edges of the form (v, w) unless M[w] changed in previous iteration.

- Theorem: Throughout the algorithm, M[v] is length of some v-t path, and after i rounds of updates, the value M[v] is no larger than the length of shortest v-t path using $\leq$ i edges.

- Overall impact
  - Working space: O(n).
  - Total space (including input): O(m+n)
  - Running time: O(mn) worst case, but substantially faster in practice.

Key steps: Dynamic programming

Formulate the problem recursively.

1. Define subproblems
2. Find recurrence relating subproblems
3. Solve the base cases

Similar to what we did for D&C

Transform recurrence into an efficient algorithm

- Data structure to store solutions to subproblems
- Evaluation order of subproblems

Dynamic Programming Summary II

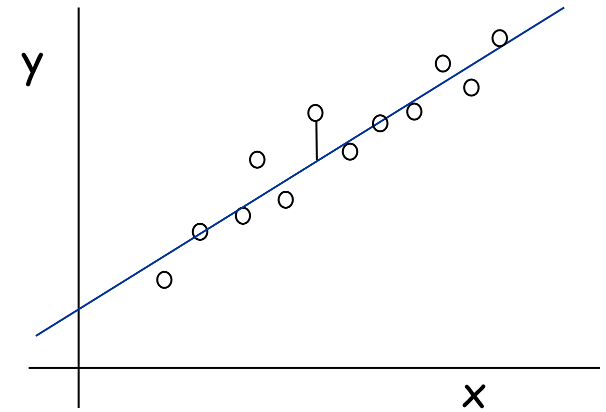
- **1D dynamic programming**
  - Weighted interval scheduling
  - Segmented Least Squares (self-study, not assessed)
  - Maximum-sum contiguous subarray
  - Longest increasing subsequence
- **2D dynamic programming**
  - Knapsack
  - Shortest path
  - Longest common subsequence
- **Dynamic programming over intervals**
  - RNA Secondary Structure

6.3 Segmented Least Squares

Segmented Least Squares

- Least squares.
  - Foundational problem in statistic and numerical analysis.
  - 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 $\Rightarrow$ min error is achieved when

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

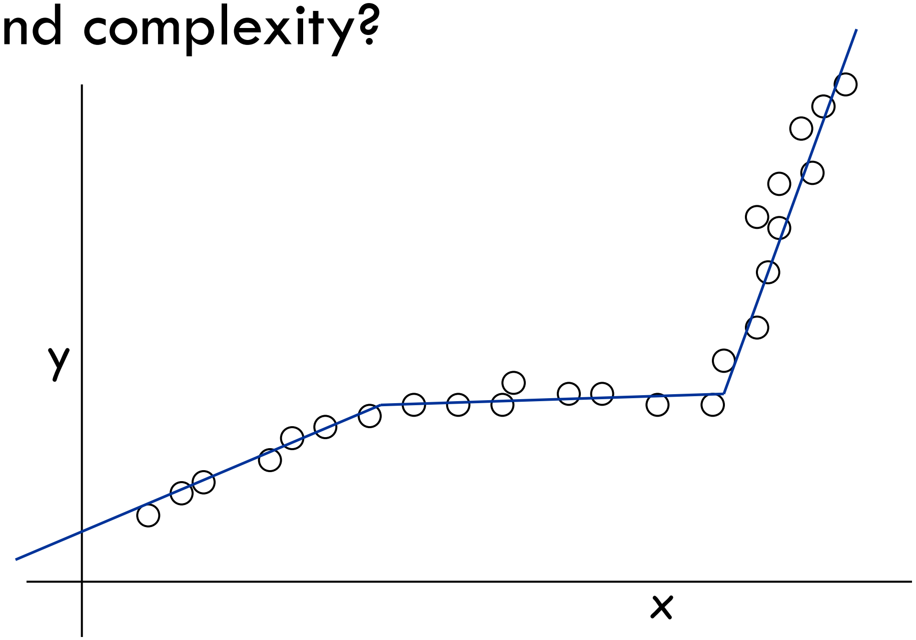
$\leftarrow$ O(n)

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
  - $x_1 < x_2 < \ldots < x_n$, find a sequence of lines that minimizes $f(x)$.

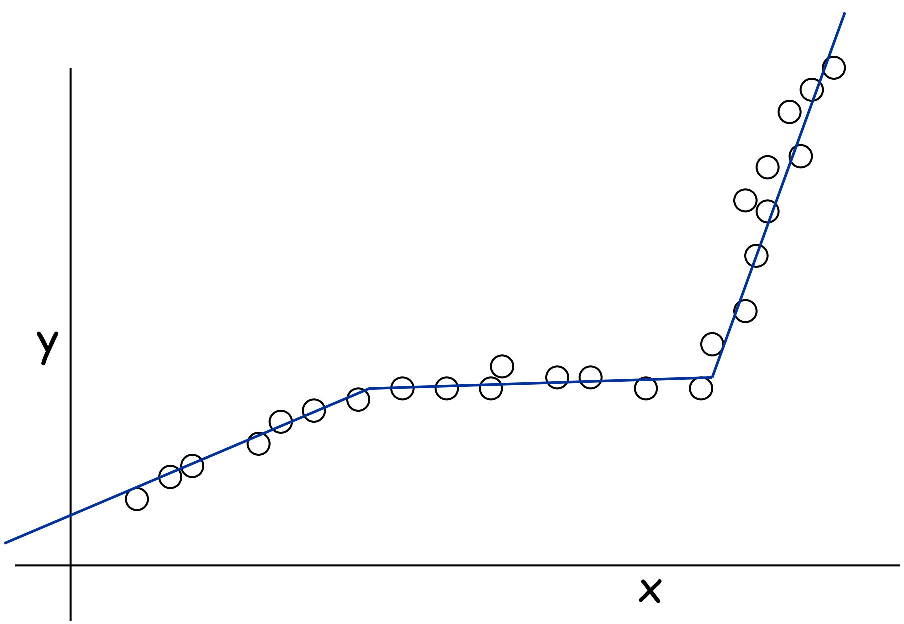
- Question. What's a reasonable choice for $f(x)$ to balance accuracy and complexity?

number of lines



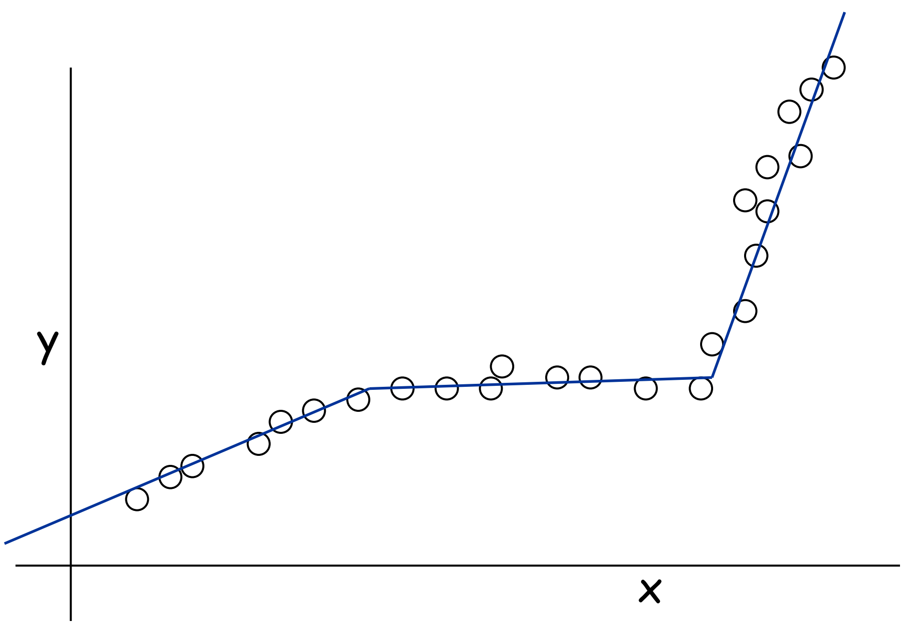
Segmented Least Squares

- 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 $x_1 < x_2 < \ldots < x_n$, find a sequence of lines that minimizes:
    - the sum of the sums of the squared errors E in each segment
    - the number of lines L
  - Tradeoff function: $E + c \cdot L$, for some constant $c > 0$.



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 $x_1 < x_2 < \ldots < x_n$, find a partition into segments that minimizes:
    - the sum of the sums of the squared errors E in each segment
    - the number of segments L
  - Tradeoff function: $E + c \cdot L$, for some constant $c > 0$.



Dynamic Programming: Multiway Choice – Step 1

Step 1: Define subproblems

$OPT(j)$ = minimum cost for points $p_1, p_2, \ldots, p_j$.

Dynamic Programming: Multiway Choice – Step 2

Notations:

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

Step 2: Finding recurrences

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

$$\boxed{OPT(j) = \min_{1 \leq i \leq j} \{ e(i,j) + c + OPT(i-1) \}}$$

Dynamic Programming: Multiway Choice – Step 3

Step 3: Solving the base cases

OPT(0) = 0

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

Segmented Least Squares: Algorithm

```
INPUT: n, (p₁,…,pₙ), c

Segmented-Least-Squares() {
   M[0] = 0
   for j = 1 to n
      for i = 1 to j
         compute the least square error eij for
         the segment pi,…, pj

   for j = 1 to n
      M[j] = min 1 ≤ i ≤ j (eij + c + M[i-1])

   return M[n]
}
```

$O(n^2)$ iterations

$O(n)$

$O(n)$ iterations

$O(n)$

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

Segmented Least Squares: Algorithm

```
INPUT: n, (p1,…,pn), c

Segmented-Least-Squares() {
   M[0] = 0
   for j = 1 to n
      for i = 1 to j
         compute the least square error eij for
         the segment pi,…, pj

   for j = 1 to n
      M[j] = min 1 ≤ i ≤ j (eij + c + M[i-1])

   return M[n]
}
```

$O(n^2)$ iterations

$O(n)$ iterations

$O(n)$

$O(n)$

Running time: $O(n^3)$

Space: $O(n^2)$