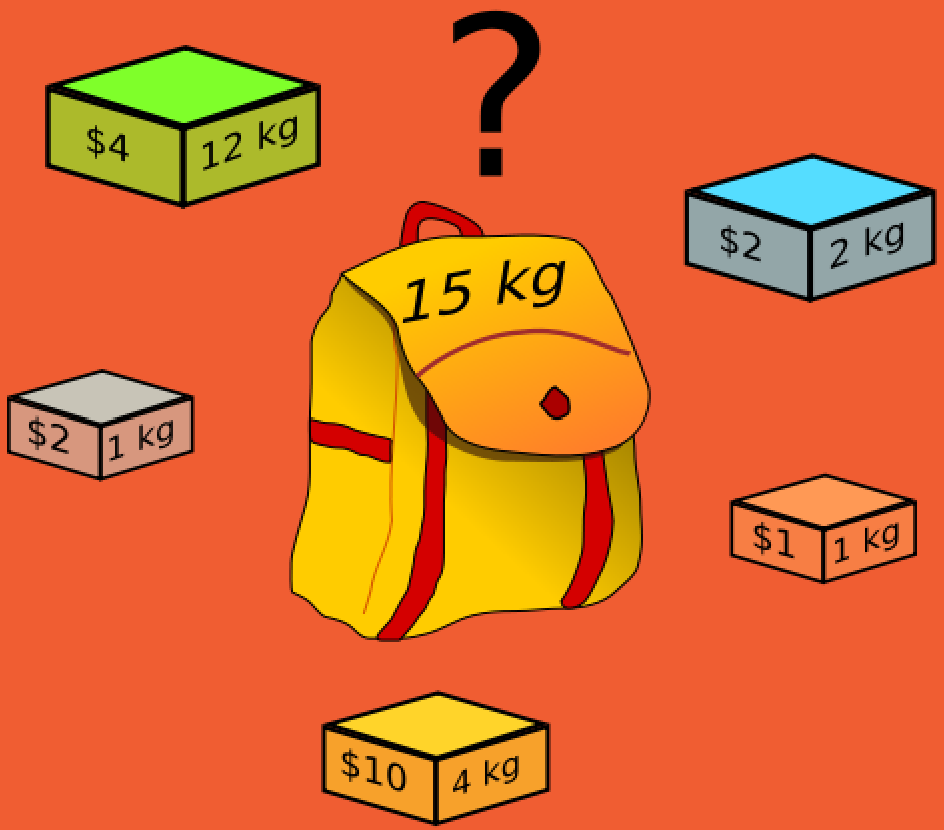
Lecture 7: Dynamic Programming II



THE UNIVERSITY OF SYDNEY

General techniques in this course

- Greedy algorithms [Lecture 3]
- Divide & Conquer algorithms [Lecture 4]
- Sweepline algorithms [Lecture 5]
- Dynamic programming algorithms [Lecture 6 and today]
- Network flow algorithms [18 Sep and 9 Oct]

Algorithmic Paradigms

– Greed. Build up a solution incrementally, optimizing some local criterion.

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

– Planesweep. Sort the geometric input. Define event points, and maintain invariant during sweep.

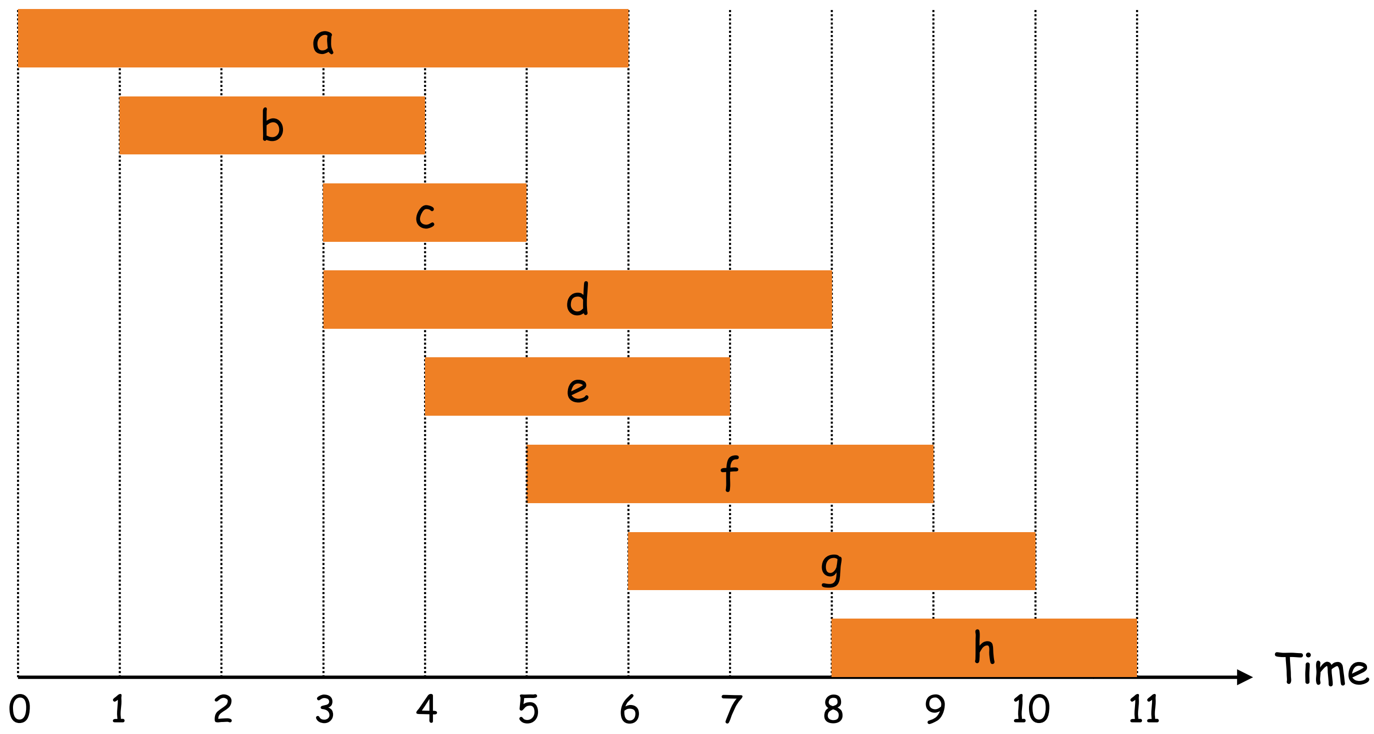
– Dynamic programming. Break up a problem into a series of overlapping sub-problems, and build up solutions to larger and larger sub-problems.

Key steps: Dynamic programming

1. Define subproblems
2. Find recurrences
3. Solve the base cases
4. Transform recurrence into an efficient algorithm

Recap: Weighted Interval Scheduling

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

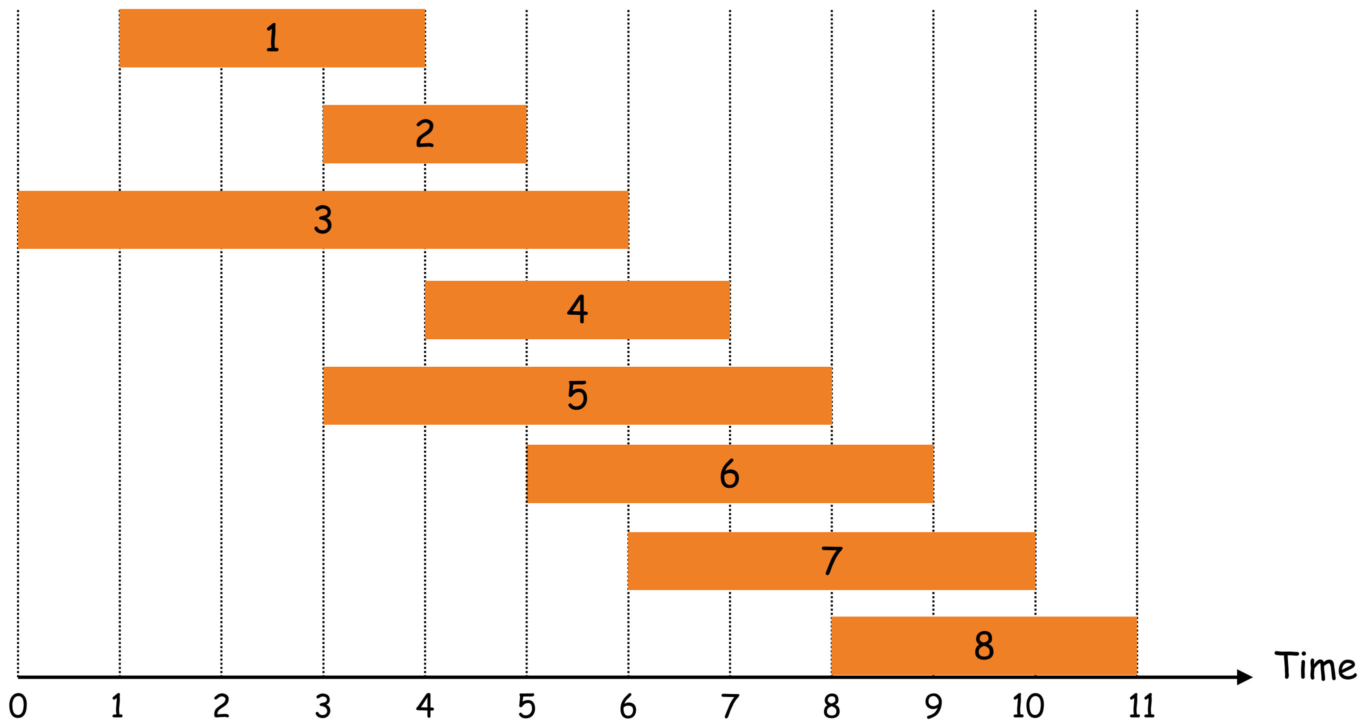


Recap: Weighted Interval Scheduling

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

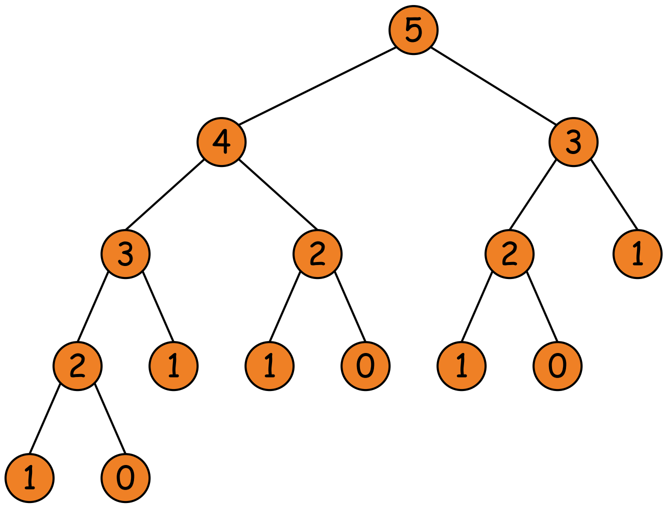
$$OPT(0) = 0$$

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

Done…more or less

Recap: Memoization

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



Could get an exponential number of subproblems!

Recap: Memoization

Instead of recomputing every subproblem store the results of each sub-problem. Unwind recursion.

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

$\downarrow$

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

Time: O(n)

Recap: Knapsack Problem

- **Knapsack problem.**
  - Given n objects and a "knapsack."
  - Item i weighs $w_i > 0$ kilograms and has value $v_i > 0$.
  - Knapsack has capacity of W kilograms.
  - **Goal:** fill knapsack so as to maximize total value.

W = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Recap: Dynamic Programming – Step 1

Step 1: Define subproblems

OPT(i, w) = max profit with subset of items 1, …, i with weight limit w.

Recap: Dynamic Programming – Step 2

Step 2: Find recurrences

- **Case 1:** OPT does not select item i.
  - OPT selects best of { 1, 2, …, i-1 } using weight limit w

- **Case 2:** OPT selects item i.
  - new weight limit = w – w_i
  - OPT selects best of { 1, 2, …, i–1 } using this new weight limit

$$\boxed{OPT(i,w) = \max \{ v_i + OPT(i-1, w-w_i),\ OPT(i-1,w) \}}$$

Recap: Dynamic Programming – Step 2

Step 2: Find recurrences

- **Case 1:** OPT does not select item i.
  - OPT selects best of { 1, 2, …, i-1 } using weight limit w
- **Case 2:** OPT selects item i.
  - new weight limit = w – w_i
  - OPT selects best of { 1, 2, …, i–1 } using this new weight limit

If $w_i > w$

$$OPT(i,w) = OPT\ (i-1,w)$$

otherwise

$$OPT(i,w) = \max \{ v_i + OPT\ (i-1,w-w_i),\ OPT(i-1,w) \}$$

Recap: Dynamic Programming – Step 3

Step 3: Solve the base cases

$$OPT(0,w) = 0$$

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

Done…more or less

Knapsack Problem: Bottom-Up

– **Knapsack.** Fill up an (n+1)-by-(W+1) array.

```
Input: n, w1,…,wN, v1,…,vN

for w = 0 to W
   M[0, w] = 0

for i = 1 to n
   for w = 1 to W
      if (wi > w)
         M[i, w] = M[i-1, w]
      else
         M[i, w] = max {M[i-1, w], vi + M[i-1, w-wi ]}
return M[n, W]
```

Knapsack Algorithm

W + 1

n + 1

	0	1	2	3	4	5	6	7	8	9	10	11
$\emptyset$	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	34	40

OPT: { 4, 3 }
value = 22 + 18 = 40

W = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

6.5 RNA Secondary Structure

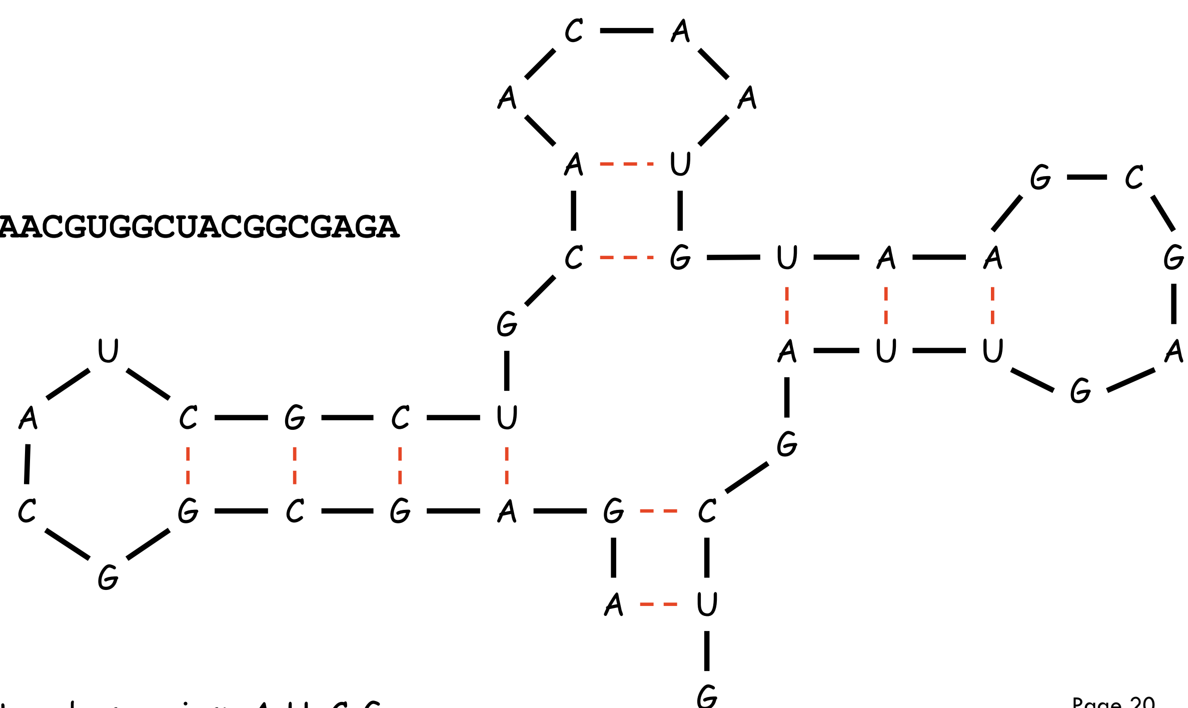
Dynamic programming over intervals

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

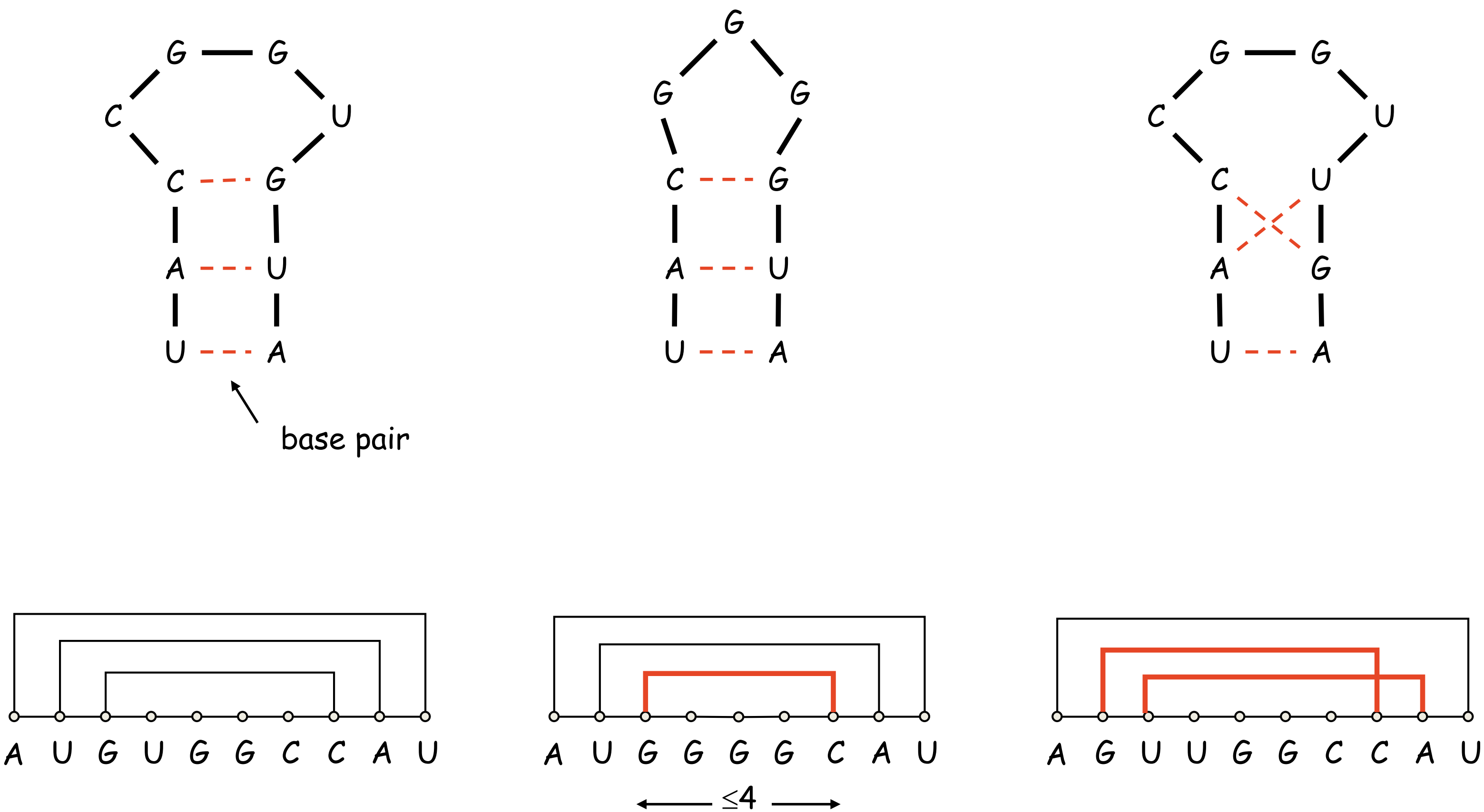
- **Secondary structure.** A set of pairs $S = \{ (b_i, b_j) \}$ that satisfy:
  - [Watson-Crick.] S is a matching and each pair in S is a Watson-Crick complement: A-U, U-A, C-G, or G-C.
  - [No sharp turns.] The ends of each pair are separated by at least 4 intervening bases. If $(b_i, b_j) \in S$, 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



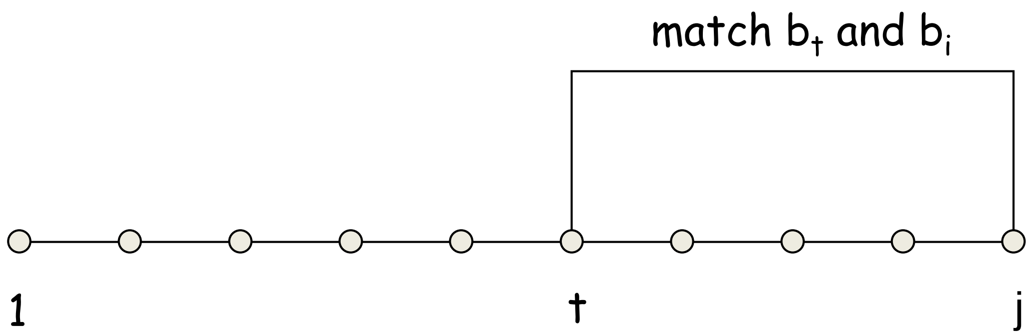
ok

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

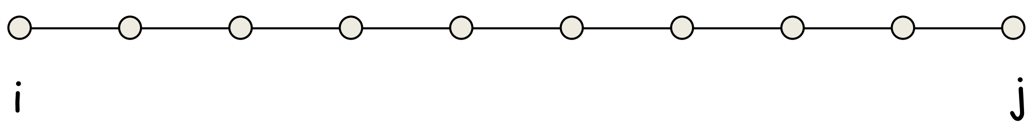
Step 1: Define subproblems

$OPT(i, j)$ = 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

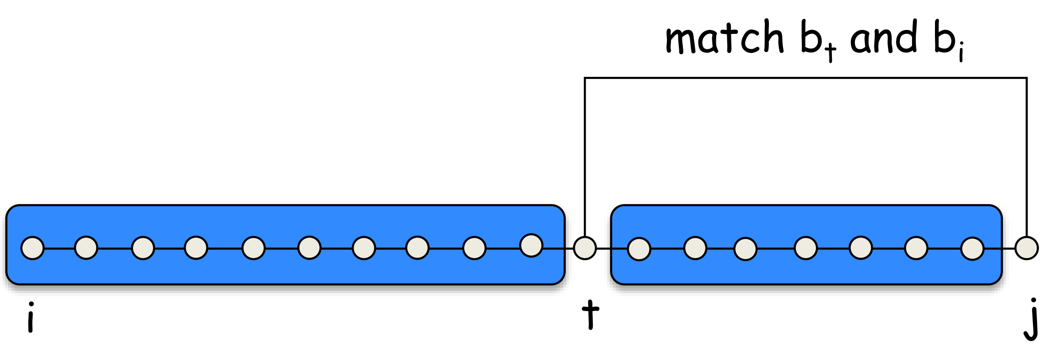


Case 1. Base b_j is not involved in a pair.
$OPT(i, j) = OPT(i, j-1)$

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 $i \leq t < j - 4$.

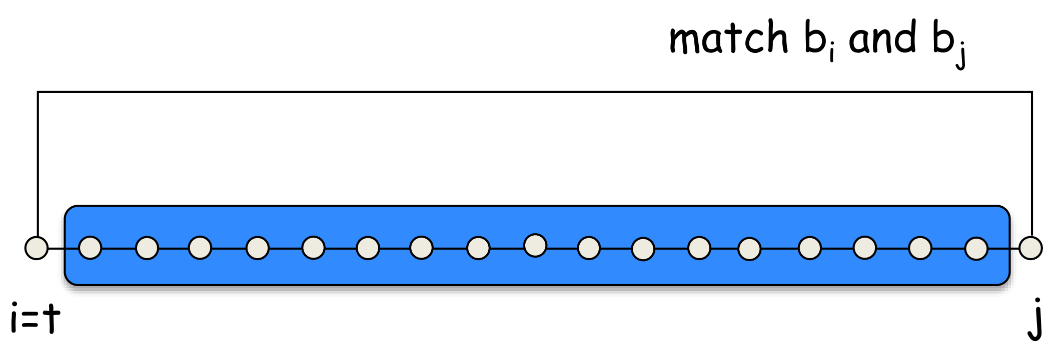
non-crossing constraint decouples resulting sub-problems

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

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 $i \le t < j - 4$.

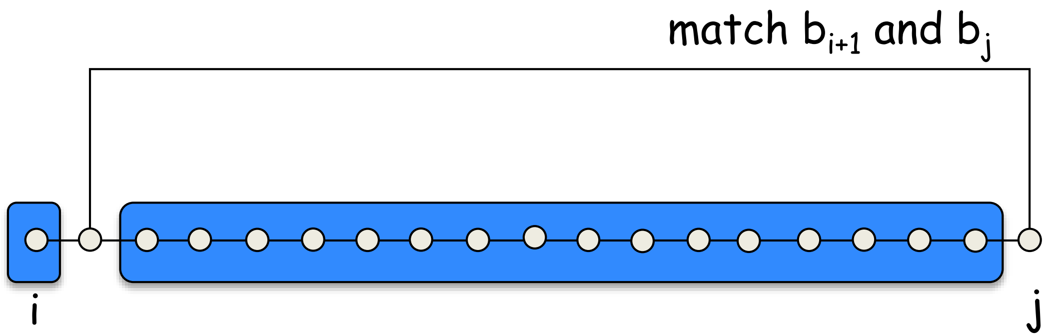
non-crossing constraint decouples resulting sub-problems

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

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 $i \leq t < j - 4$.

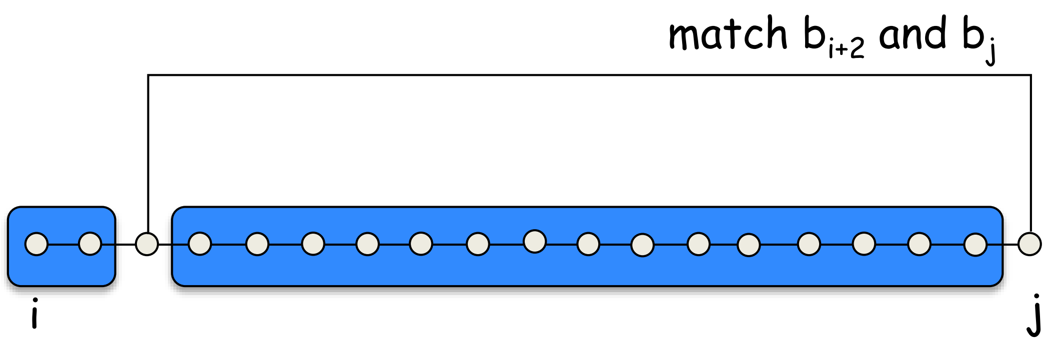
non-crossing constraint decouples resulting sub-problems

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

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 $i \le t < j - 4$.

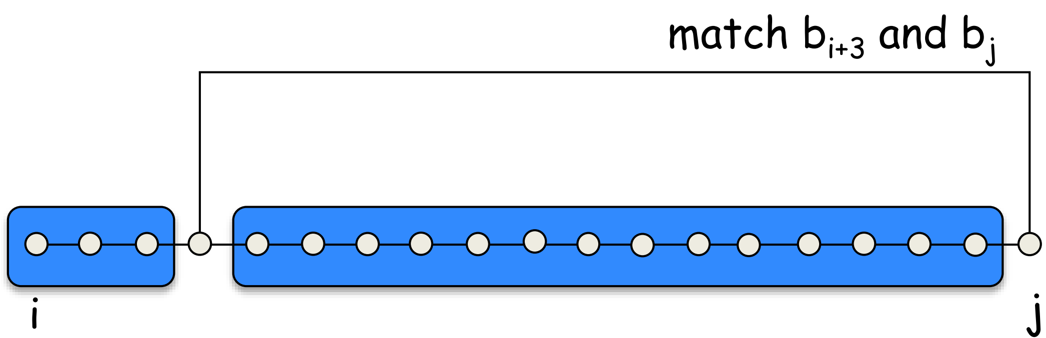
non-crossing constraint decouples resulting sub-problems

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

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

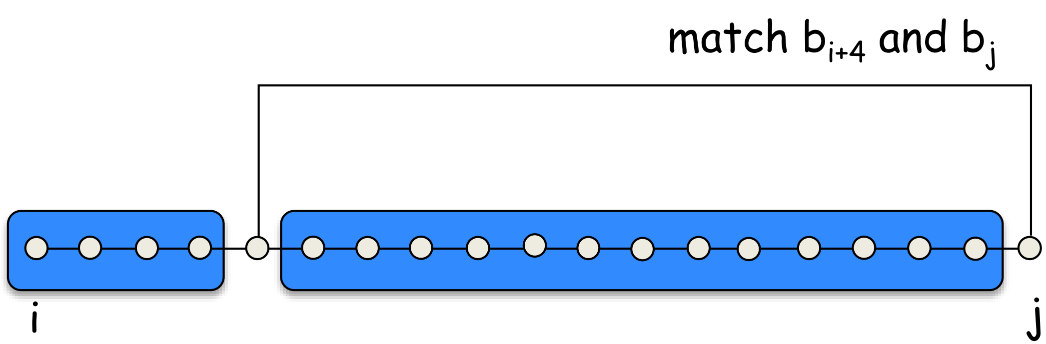
non-crossing constraint decouples resulting sub-problems

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

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 $i \leq t < j - 4$.

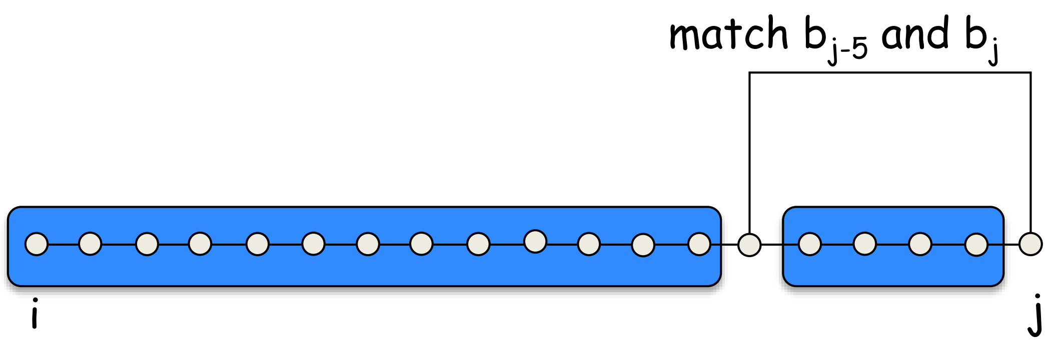
non-crossing constraint decouples resulting sub-problems

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

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 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 $i \le t < j - 4$.
- non-crossing constraint decouples resulting sub-problems
- $\mathrm{OPT}(i, j) = 1 + \max\limits_{i \le t < j-4} \{ \mathrm{OPT}(i, t-1) + \mathrm{OPT}(t+1, j-1) \}$

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 $i \le t < j - 4$.

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

Step 3:

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

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

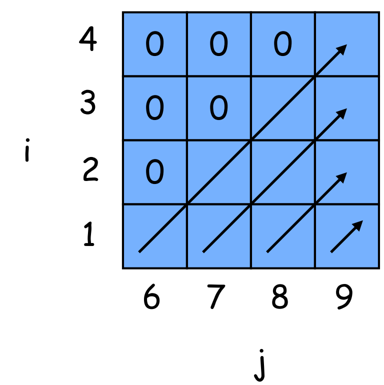
Bottom Up Dynamic Programming Over Intervals

– **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



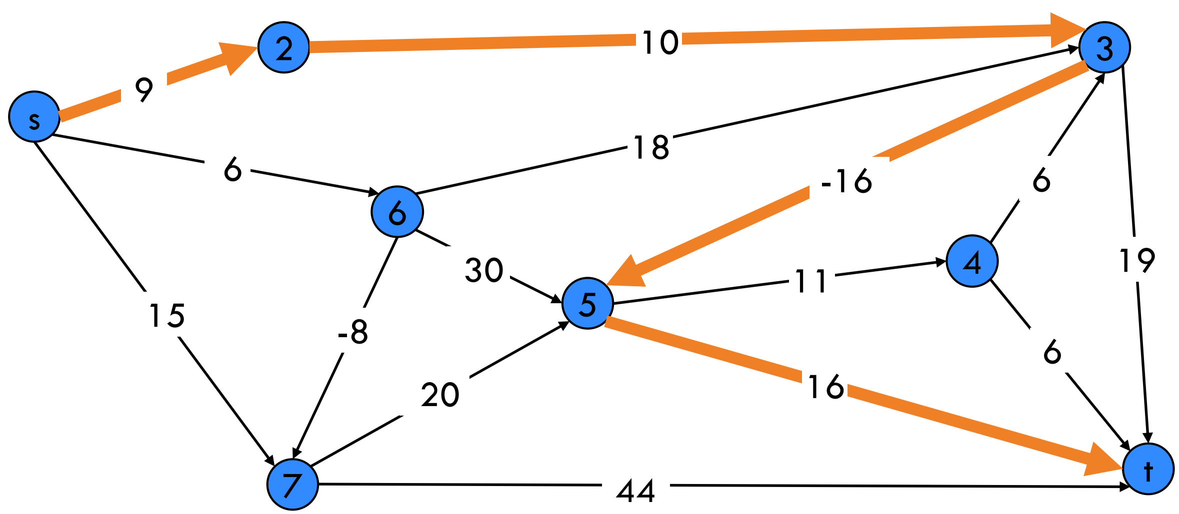
– **Running time:** $O(n^3)$

6.8 Shortest Paths

Shortest Paths

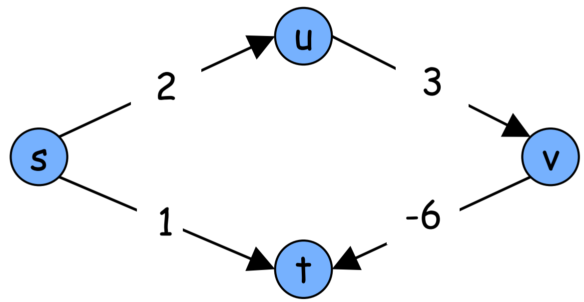
– **Shortest path problem.** Given a directed graph G = (V, E), with edge weights c_{vw}, find shortest path from node s to node t.

allow negative weights

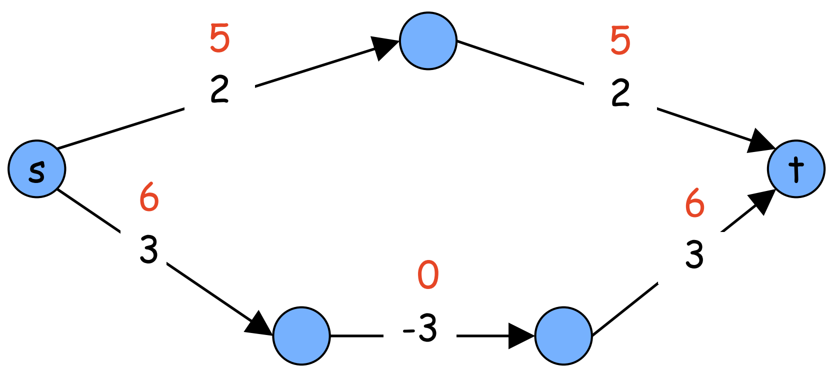


Shortest Paths: Failed Attempts

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

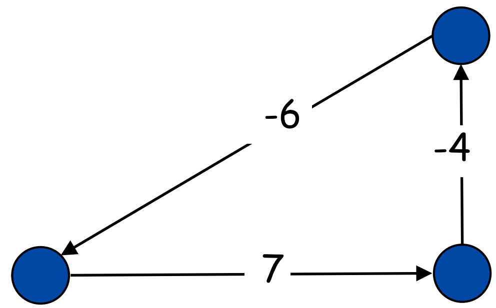


– **Re-weighting.** Adding a constant to every edge weight can fail.

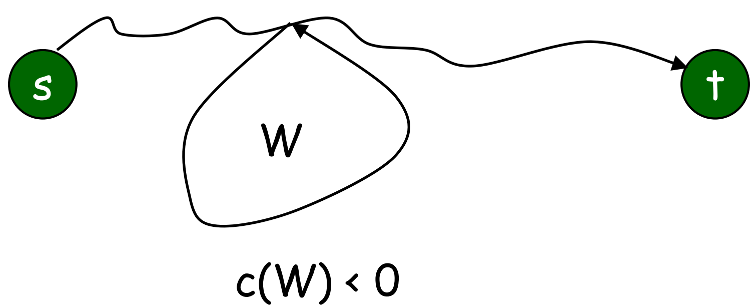


Shortest Paths: Negative Cost Cycles

- **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.

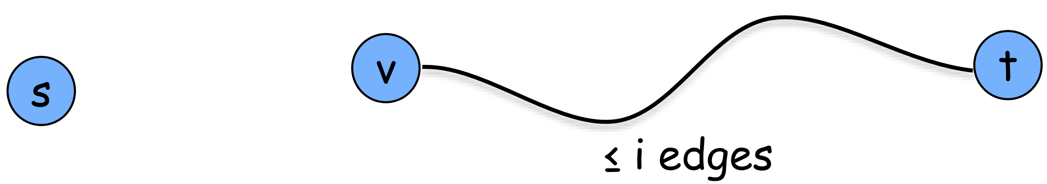


Shortest Paths: Dynamic Programming

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.

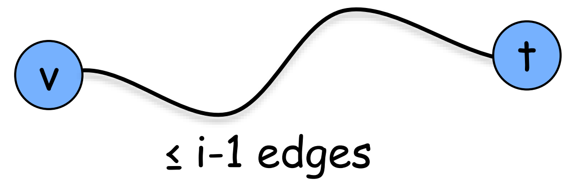


Shortest Paths: Dynamic Programming

Step 2: Find recurrences

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

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

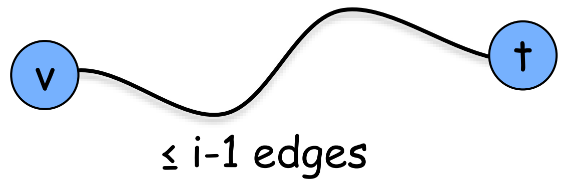


Shortest Paths: Dynamic Programming

Step 2: Find recurrences

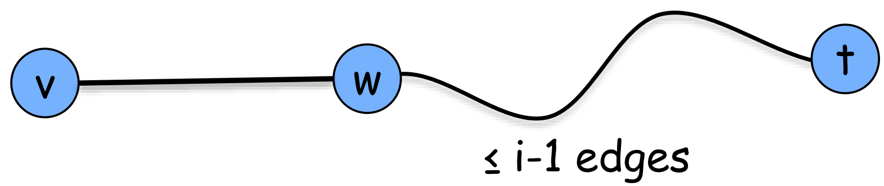
Case 1: P uses at most i-1 edges.

- 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



$$OPT(i,v) = \min\{OPT(i-1,v), \min_{(v,w)\in E} [OPT(i-1,w)+c_{vw}] \}$$

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) = ∞

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

Shortest Paths: 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 }
}
```

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

– **Finding the shortest paths.** Maintain a "successor" for each table entry.

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, $\Theta(m+n)$ space.

Shortest Paths: Practical Improvements

- 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.

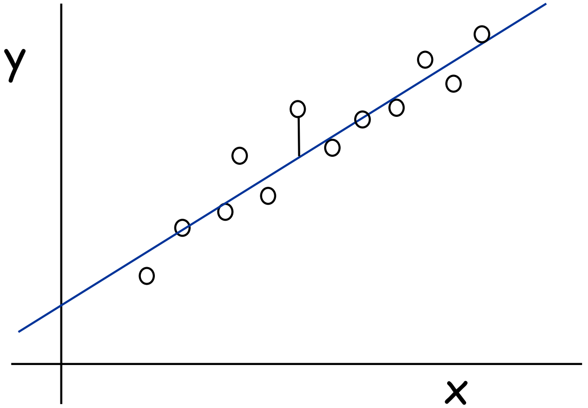
- Overall impact
  - Memory: O(m + n).
  - Running time: O(mn) worst case, but substantially faster in practice.

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

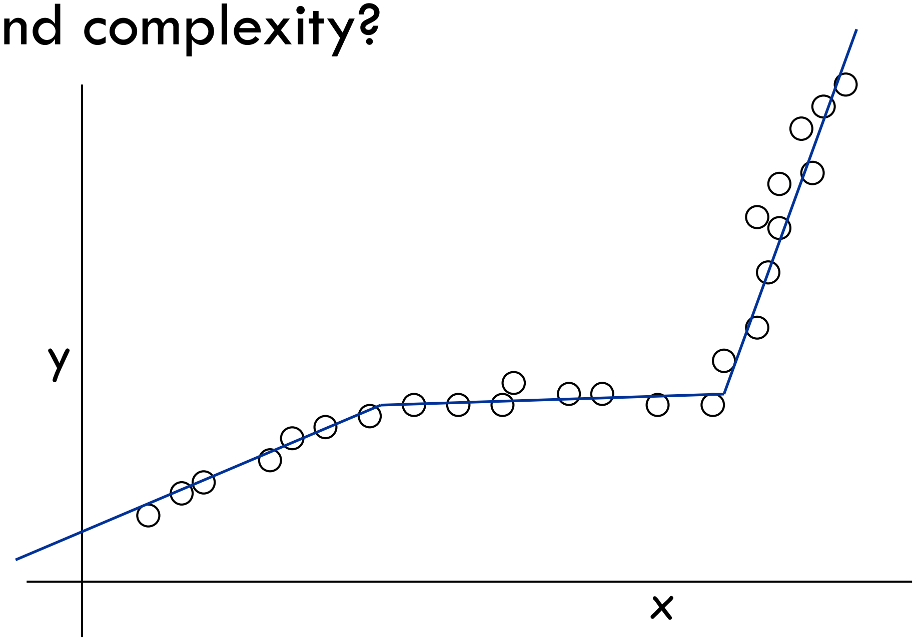
$$a = \frac{n \sum_i x_i y_i - (\sum_i x_i)(\sum_i y_i)}{n \sum_i x_i^2 - (\sum_i x_i)^2}, \quad b = \frac{\sum_i y_i - a \sum_i x_i}{n}$$

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 $f(x)$.

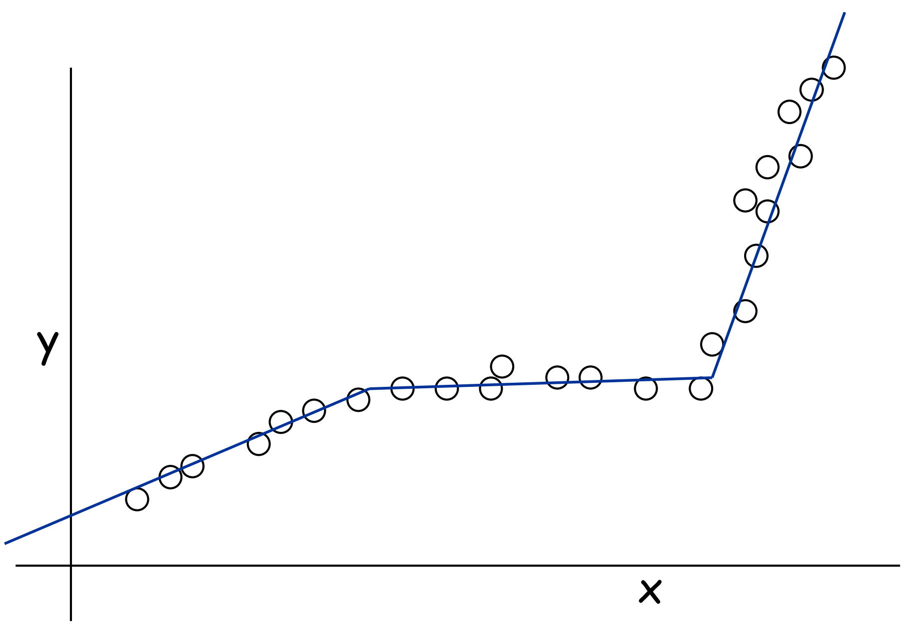
- Question. What's a reasonable choice for f(x) to balance accuracy and complexity?

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 $c > 0$.



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

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

Dynamic Programming: Multiway Choice – Step 3

Step 3: Solving the base cases

$$OPT(0) = 0$$

If $j=0$ then

$$OPT(0) = 0$$

otherwise

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

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

$O(n)$

If j=0 then

$$OPT(0) = 0$$

otherwise

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

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

$O(n)$

Running time: $O(n^3)$

Space: $O(n^2)$

Dynamic Programming Summary I

3 steps:

1. Define subproblems
2. Find recurrences
3. Solve the base cases

4. Transform recurrence into an efficient algorithm [usually bottom-up]

Dynamic Programming Summary II

- **1D dynamic programming**
  - Weighted interval scheduling
  - Segmented Least Squares
  - Maximum-sum contiguous subarray
  - Longest increasing subsequence

- **2D dynamic programming**
  - Knapsack

- **Dynamic programming over intervals**
  - RNA Secondary Structure