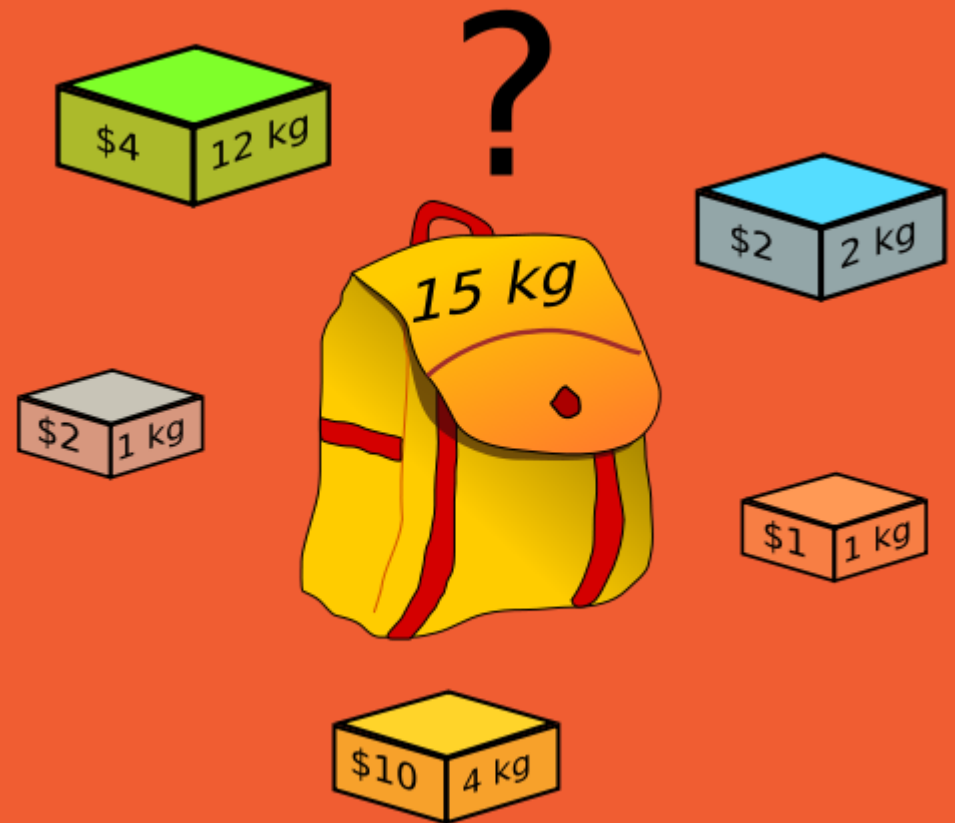


Lecture 7: Dynamic Programming II



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]

Weighted Interval Scheduling

Knapsack Problem

RNA Secondary Structure

Shortest Paths

Bellman-Ford

Segmented Least Square

Algorithmic Paradigms

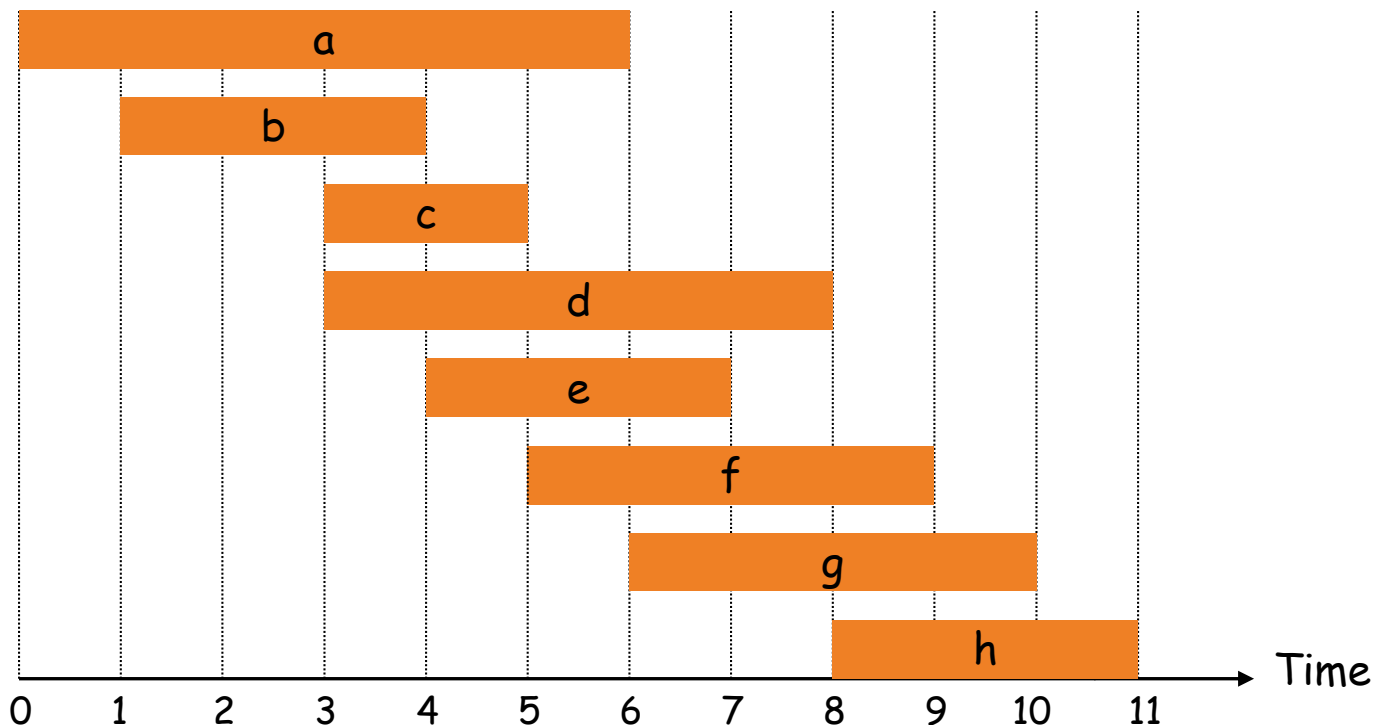
- **Greed.** Build up a solution incrementally, optimizing some local criterion.
- **Divide-and-conquer.** Break up a problem into two sub-problems, solve each sub-problem independently, and combine solution to sub-problems to form solution to original problem.
- **Planesweep.** Sort the geometric input. 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.

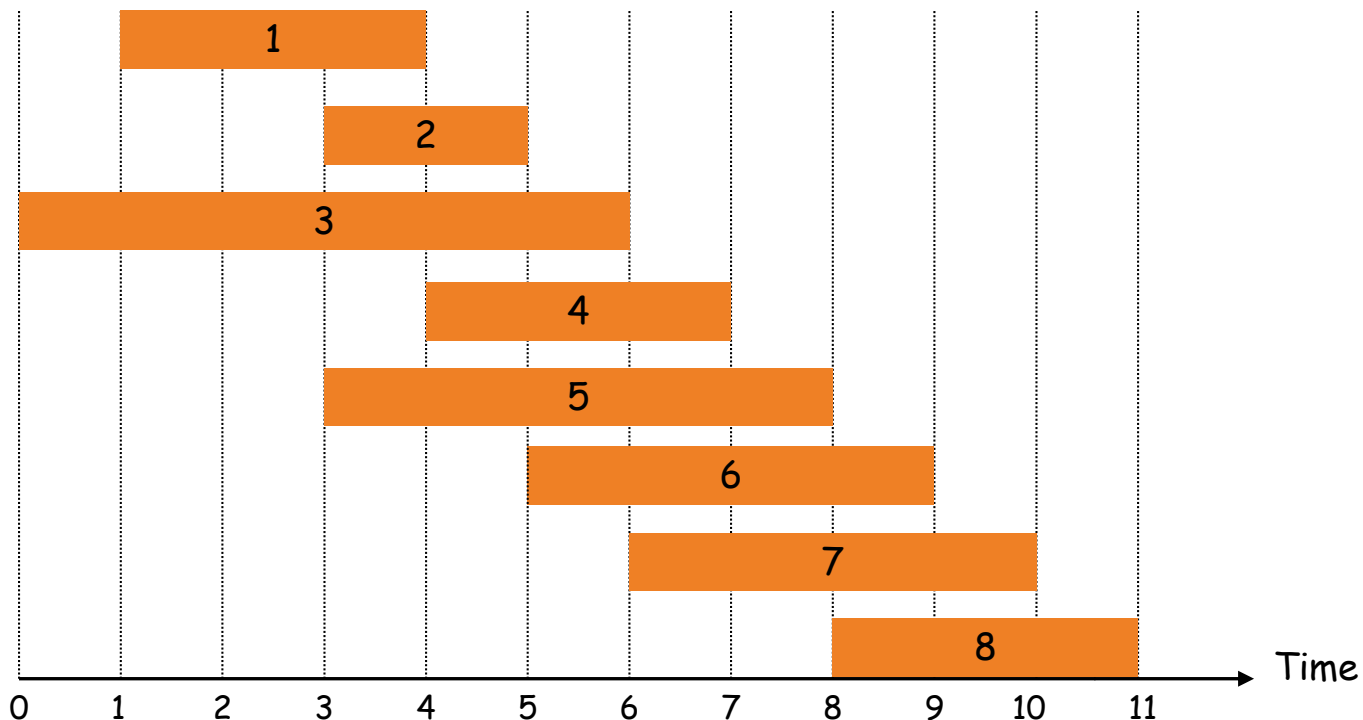


Recap: Weighted Interval Scheduling

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

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

$\text{OPT}(j)$ = 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 $\{ p(j) + 1, p(j) + 2, \dots, j - 1 \}$
- must include optimal solution to problem consisting of remaining compatible jobs $1, 2, \dots, p(j)$

因为已经选择了job j ,
所以只能考虑 $P(j)$ 之前的了。

– **Case 2:** OPT does not select job j .

- must include optimal solution to problem consisting of remaining compatible jobs $1, 2, \dots, j-1$

因为没有选择job j , 所以
要考虑 $j-1$ 及之前的jobs

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

Recap: Dynamic Programming – Step 3

Step 3: Solve the base cases

$$OPT(0) = 0$$

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

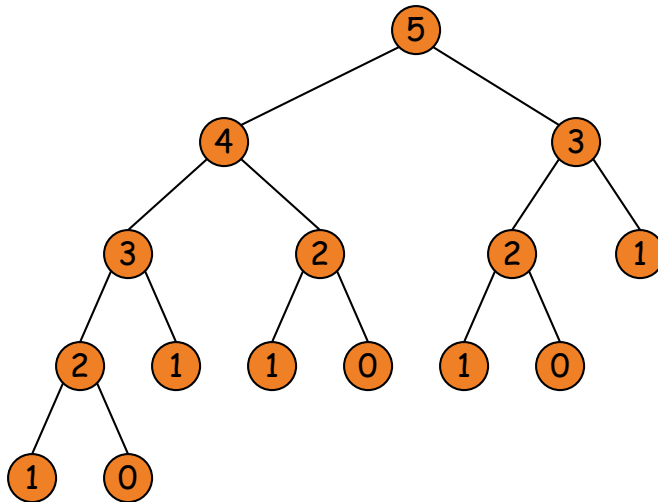


Done...more or less

Recap: Memoization

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

□



Could get an exponential number of subproblems!

Recap: Memoization

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

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



□

```
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

$\text{OPT}(i, w)$ = 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 $\{1, 2, \dots, i-1\}$ using weight limit w
- **Case 2:** OPT selects item i .
 - new weight limit = $w - w_i$
 - OPT selects best of $\{1, 2, \dots, i-1\}$ using this new weight limit

Formula

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

Recap: Dynamic Programming – Step 2

Step 2: Find recurrences

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

If $w_i > w$ 如果物品 i 的质量比背包承重还要大的话，忽略它。

$$\text{OPT}(i, w) = \text{OPT}(i-1, w)$$

otherwise

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

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

6.5 RNA Secondary Structure

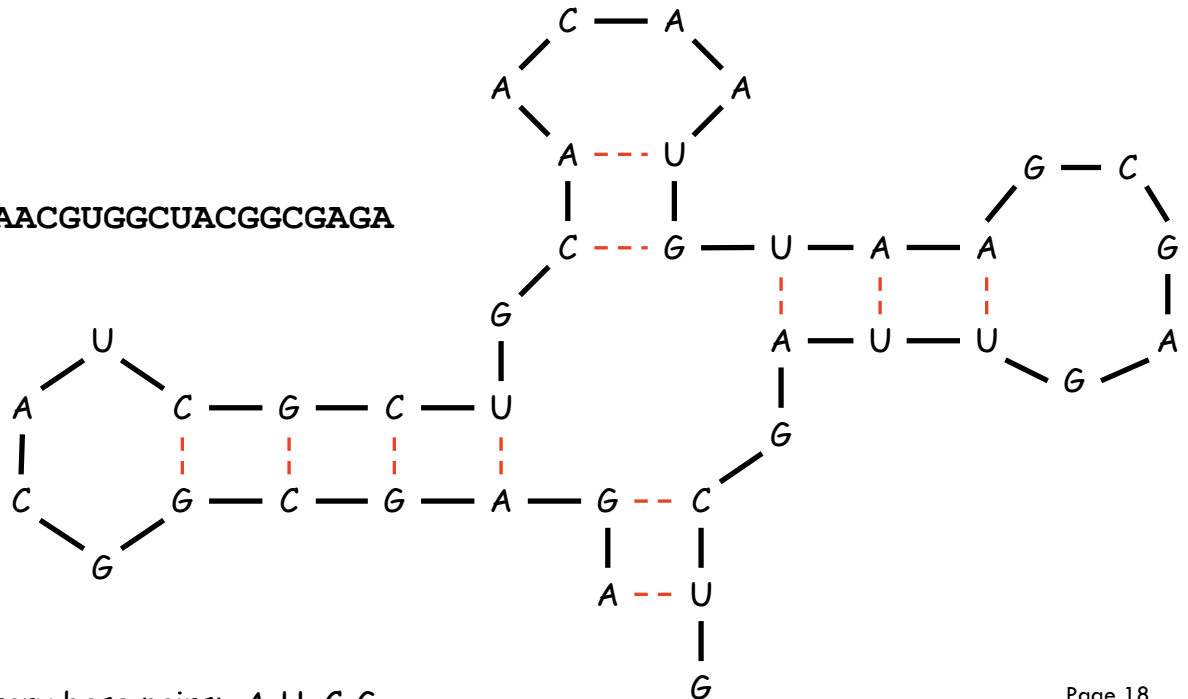
Time Complexity: $O(n^3)$

Space Complexity: $O(n^2)$

RNA (Ribonucleic acid) Secondary Structure


- **RNA.** String $B = b_1b_2\dots b_n$ 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



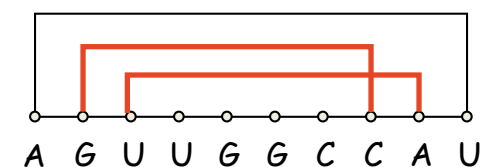
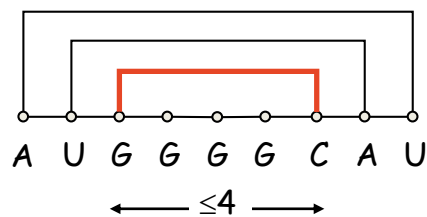
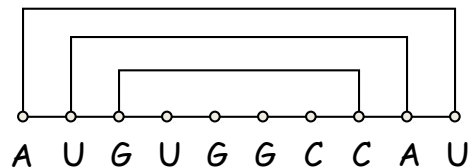
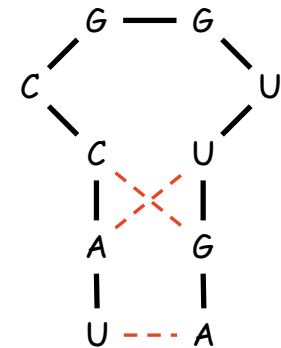
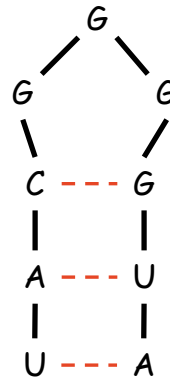
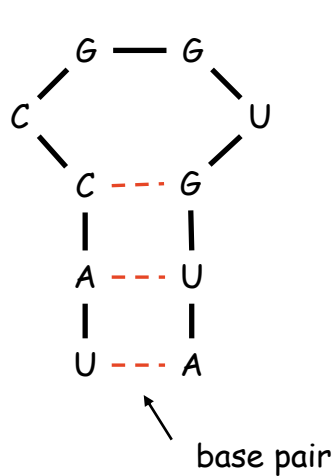
RNA Secondary Structure

- **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.



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

RNA Secondary Structure: Examples



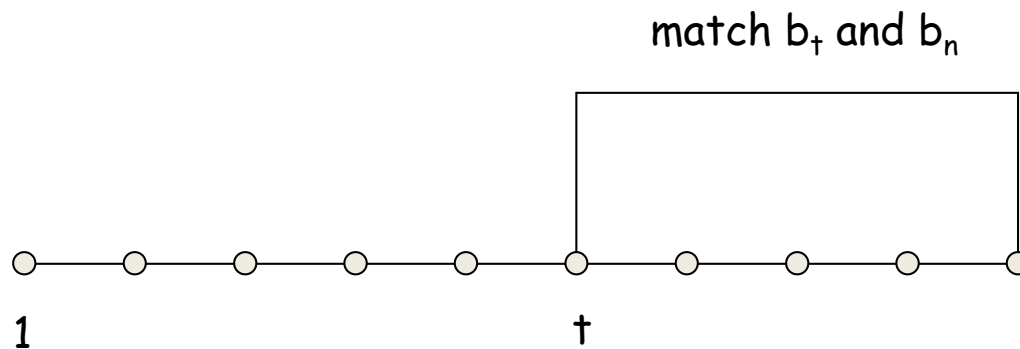
ok

sharp turn

crossing

RNA Secondary Structure: Subproblems

- **First attempt (Step 1).** $\text{OPT}(i) =$ maximum number of base pairs in a secondary structure of the substring $b_1 b_2 \dots b_i$.



- **Difficulty (in Step 2).** Results in two sub-problems.
 - Finding secondary structure in: $b_1 b_2 \dots b_{t-1}$. $\leftarrow \text{OPT}(t-1)$
 - Finding secondary structure in: $b_{t+1} b_{t+2} \dots b_{i-1}$. \leftarrow need more sub-problems

Dynamic Programming Over Intervals – Step 1

Step 1: Define subproblems

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

Dynamic Programming Over Intervals – Step 2

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

Step 2: Find recurrences

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

- $\text{OPT}(i, j) = \text{OPT}(i, j-1)$

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

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

Dynamic Programming Over Intervals – Step 3

Step 3: Solve the base cases

If $i \geq j - 4$ then

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

Dynamic Programming Over Intervals – 3 Steps

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

Step 2:

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

- $\text{OPT}(i, j) = \text{OPT}(i, j - 1)$

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

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

Step 3:

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

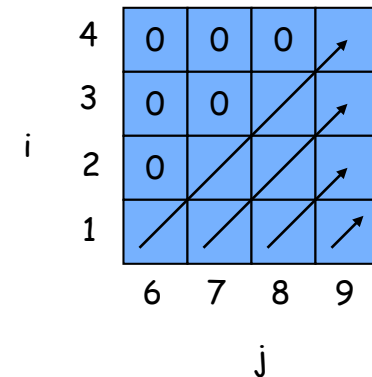
- $\text{OPT}(i, j) = 0$ by no-sharp turns condition.

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

Bellman-Ford Algorithm:

The Bellman-Ford algorithm correctly computes the shortest path in a graph even when the graph contains edges having negative weight.

With

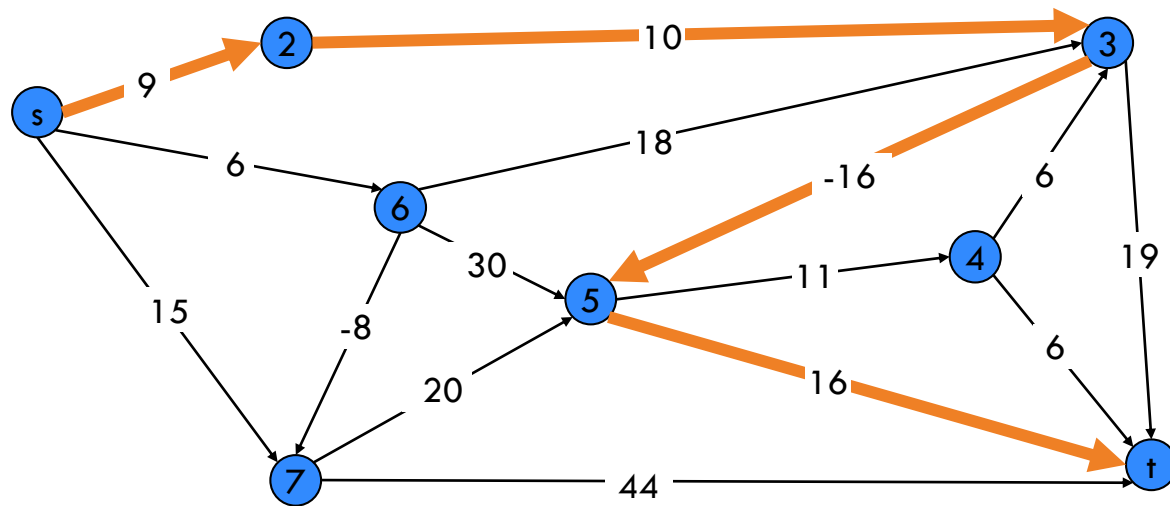
Time Complexity $O(nm)$

Space Complexity $O(n+m)$

Shortest Paths

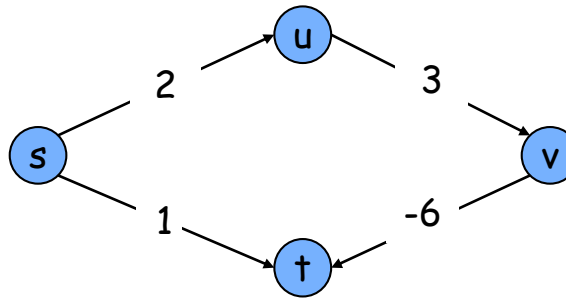
- **Shortest path problem.** Given a directed graph $G = (V, E)$, with edge weights c_{vw} , find shortest path from node s to node t .

allow negative weights

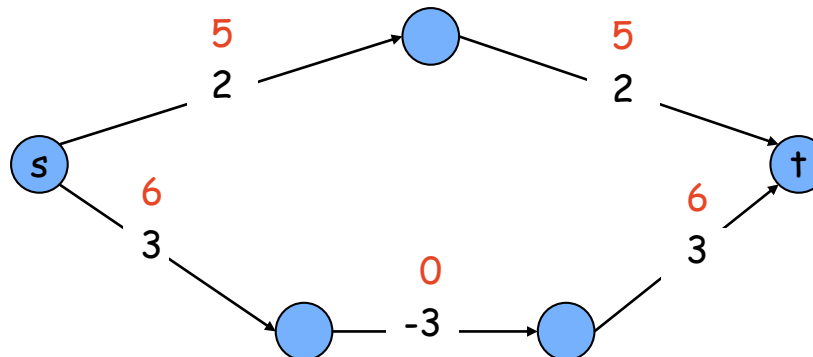


Shortest Paths: Failed Attempts

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

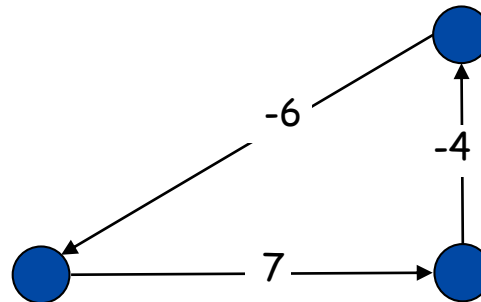


- **Re-weighting.** Adding a constant to every edge weight can fail.

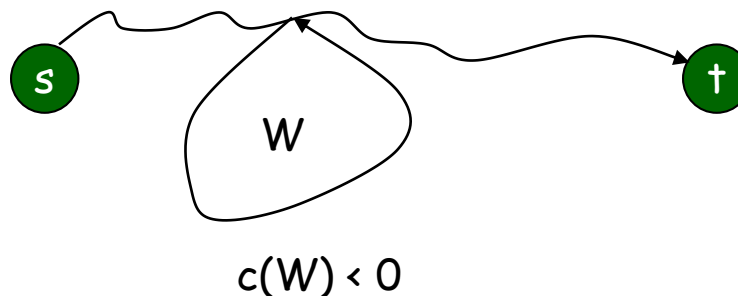


Shortest Paths: Negative Cost Cycles

- **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.

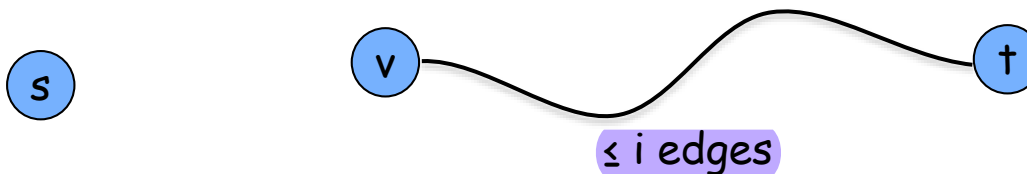


Shortest Paths: Dynamic Programming

Problem: Find shortest path from s to t

Step 1: Define subproblems

$\text{OPT}(i, v)$ = length of shortest v - t path P using at most i edges.

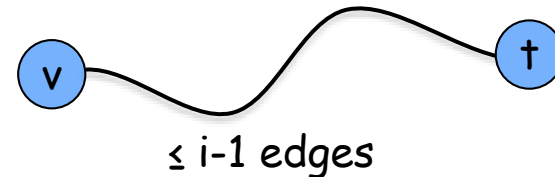


Shortest Paths: Dynamic Programming

Step 2: Find recurrences

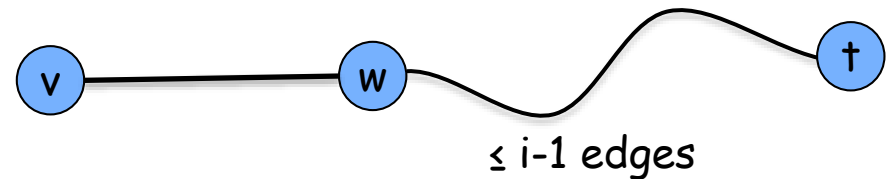
Case 1: P uses at most $i-1$ edges.

- $\text{OPT}(i, v) = \text{OPT}(i-1, v)$



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



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

Shortest Paths: Dynamic Programming

Step 3: Solve the base cases

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

Shortest Paths: Dynamic Programming

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

Step 2:

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

- $\text{OPT}(i, v) = \text{OPT}(i-1, v)$

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

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

Shortest Paths: Implementation

```
Shortest-Path( $G, t$ ) {  
    foreach node  $v \in V$   
         $M[0, v] \leftarrow \infty$   
     $M[0, t] \leftarrow 0$   
  
    for  $i = 1$  to  $n-1$   
        foreach node  $v \in V$   
             $M[i, v] \leftarrow M[i-1, v]$   
            foreach edge  $(v, w) \in E$   
                 $M[i, v] \leftarrow \min \{ M[i, v], M[i-1, w] + c_{vw} \}$   
}
```

- **Analysis.** $\Theta(mn)$ time, $\Theta(n^2)$ space.
- **Finding the shortest paths.** Maintain a "successor" for each table entry.

Bellman-Ford: Efficient Implementation

Time

Complexity:

$O(nm)$

```
Push-Based-Shortest-Path( $G, s, t$ ) {  
  foreach node  $v \in V$  {  
     $M[v] \leftarrow \infty$   
     $\text{successor}[v] \leftarrow \emptyset$  }  
  
   $M[t] = 0$   
  for  $i = 1$  to  $n-1$  {  
    foreach node  $w \in V$  {  
      if ( $M[w]$  has been updated in previous iteration) {  
        foreach node  $v$  such that  $(v, w) \in E$  {  
          if ( $M[v] > M[w] + c_{vw}$ ) {  
             $M[v] \leftarrow M[w] + c_{vw}$   
             $\text{successor}[v] \leftarrow w$   
          }  
        }  
      }  
    }  
    If no  $M[w]$  value changed in iteration  $i$ , stop.  
  }  
}
```

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

Shortest Paths: Practical Improvements

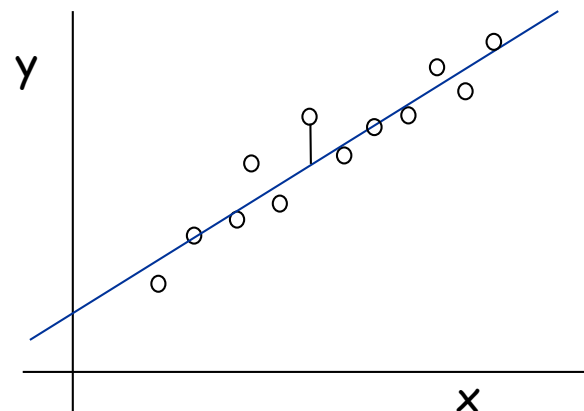
- Practical improvements.
 - Maintain only one array $M[v]$ = shortest v - t path that we have found so far.
 - No need to check edges of the form (v, w) unless $M[w]$ changed in previous iteration.
- **Theorem:** Throughout the algorithm, $M[v]$ is length of some v - t path, and after i rounds of updates, 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: $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$.
 - Find a line $y = ax + b$ that minimizes the sum of the squared error:

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



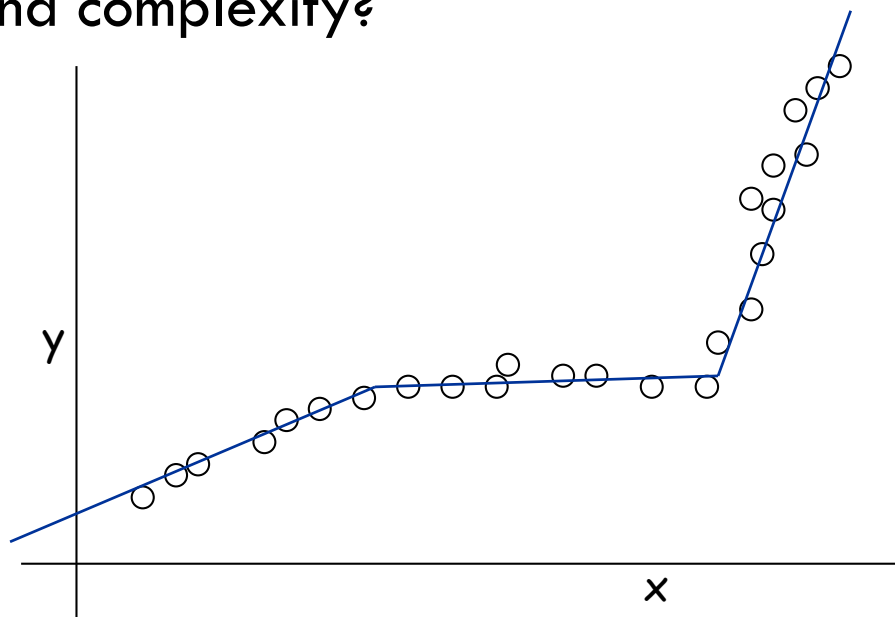
- **Solution.** Calculus \Rightarrow min error is achieved when

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

Segmented Least Squares

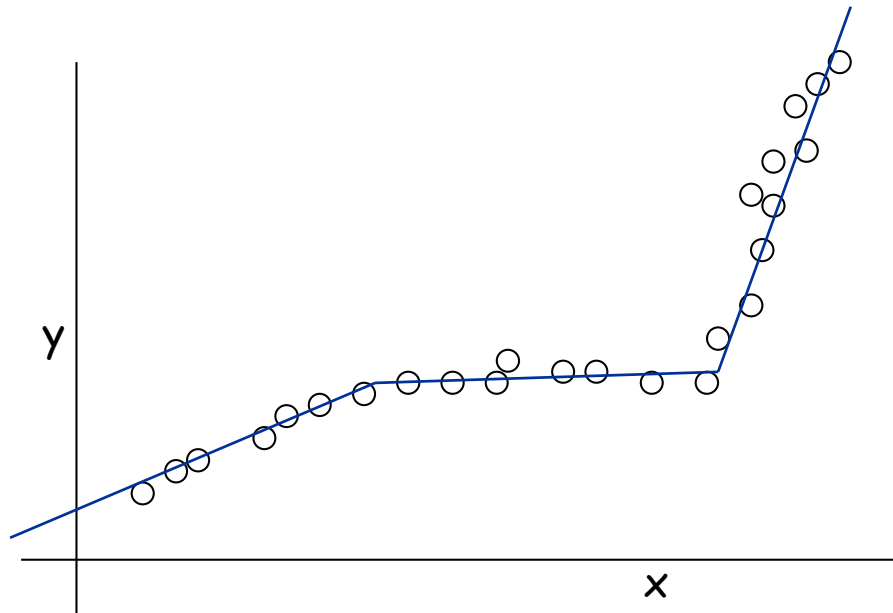
- Segmented least squares.
 - Points lie roughly on a sequence of several line segments.
 - Given n points in the plane $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$ with
 - $x_1 < x_2 < \dots < x_n$, find a sequence of lines that minimizes $f(x)$.
- **Question.** What's a reasonable choice for $f(x)$ 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 $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$ with $x_1 < x_2 < \dots < x_n$, 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

$\text{OPT}(j)$ = minimum cost for points p_1, p_2, \dots, p_j .

Dynamic Programming: Multiway Choice – Step 2

Notations:

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

Step 2: Finding recurrences

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

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

Dynamic Programming: Multiway Choice – Step 3

Step 3: Solving the base cases

$$\text{OPT}(0) = 0$$

If $j=0$ then

$$\text{OPT}(0) = 0$$

otherwise

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

Segmented Least Squares: Algorithm

INPUT: $n, (p_1, \dots, p_n), c$

Segmented-Least-Squares() {

$M[0] = 0$

 for $j = 1$ to n

 for $i = 1$ to j

 compute the least square error e_{ij} for
 the segment p_i, \dots, p_j

 for $j = 1$ to n

$M[j] = \min_{1 \leq i \leq j} (e_{ij} + c + M[i-1])$

 return $M[n]$

}

$O(n^2)$
iterations

$O(n)$

$O(n)$
iterations

$O(n)$

If $j=0$ then

$$\text{OPT}(0) = 0$$

otherwise

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

Segmented Least Squares: Algorithm

```
INPUT:  $n, (p_1, \dots, p_n), c$ 

Segmented-Least-Squares() {
   $M[0] = 0$ 
   $O(n^2)$  iterations {
    for  $j = 1$  to  $n$ 
      for  $i = 1$  to  $j$ 
        compute the least square error  $e_{ij}$  for ←  $O(n)$ 
        the segment  $p_i, \dots, p_j$ 
  }
   $O(n)$  iterations {
    for  $j = 1$  to  $n$ 
       $M[j] = \min_{1 \leq i \leq j} (e_{ij} + c + M[i-1])$  ←  $O(n)$ 
  }
  return  $M[n]$ 
}
```

Running time: $O(n^3)$

Space: $O(n^2)$

Dynamic Programming Summary I

3 steps:

1. Defining subproblems
2. Finding recurrences
3. Solving the base cases

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