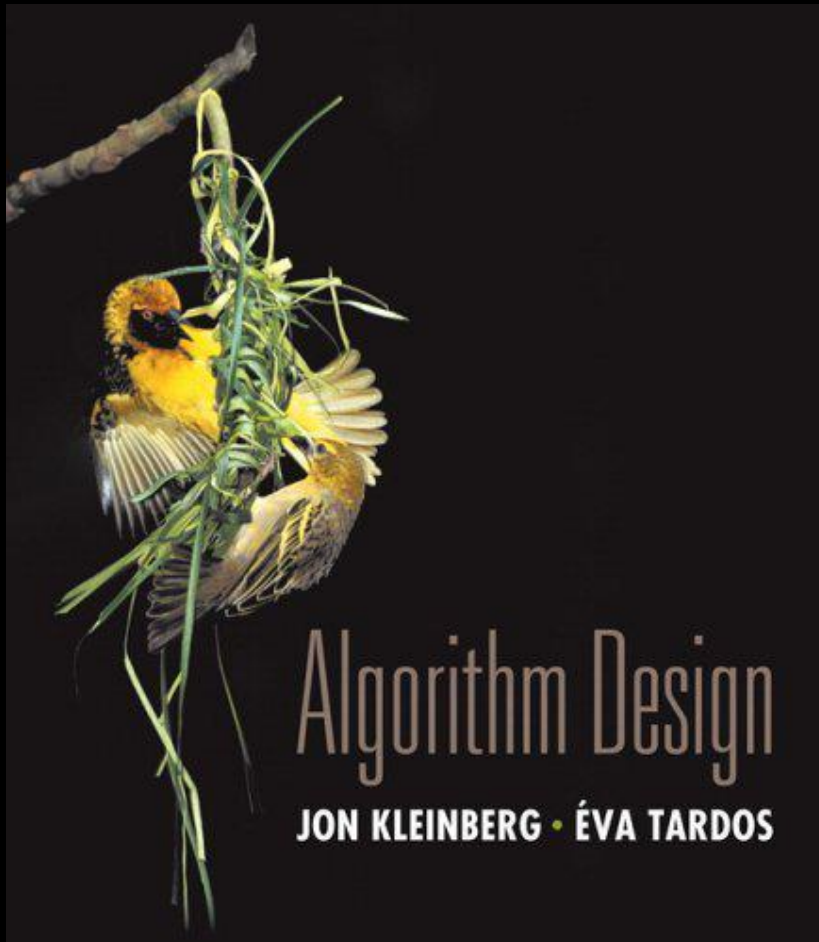


Chapter 6

Dynamic Programming



Slides by Kevin Wayne.
Copyright © 2005 Pearson-Addison Wesley.
All rights reserved.

Algorithmic Paradigms

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

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

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

Dynamic Programming History

Bellman. [1950s] Pioneered the systematic study of dynamic programming.

Etymology.

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

"it's impossible to use dynamic in a pejorative sense"
"something not even a Congressman could object to"

Reference: Bellman, R. E. *Eye of the Hurricane, An Autobiography*.

Dynamic Programming Applications

Areas.

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

Some famous dynamic programming algorithms.

- Unix diff for comparing two files.
- Viterbi for hidden Markov models.
- Smith-Waterman for genetic sequence alignment.
- Bellman-Ford for shortest path routing in networks.
- Cocke-Kasami-Younger for parsing context free grammars.

6.1 Weighted Interval Scheduling

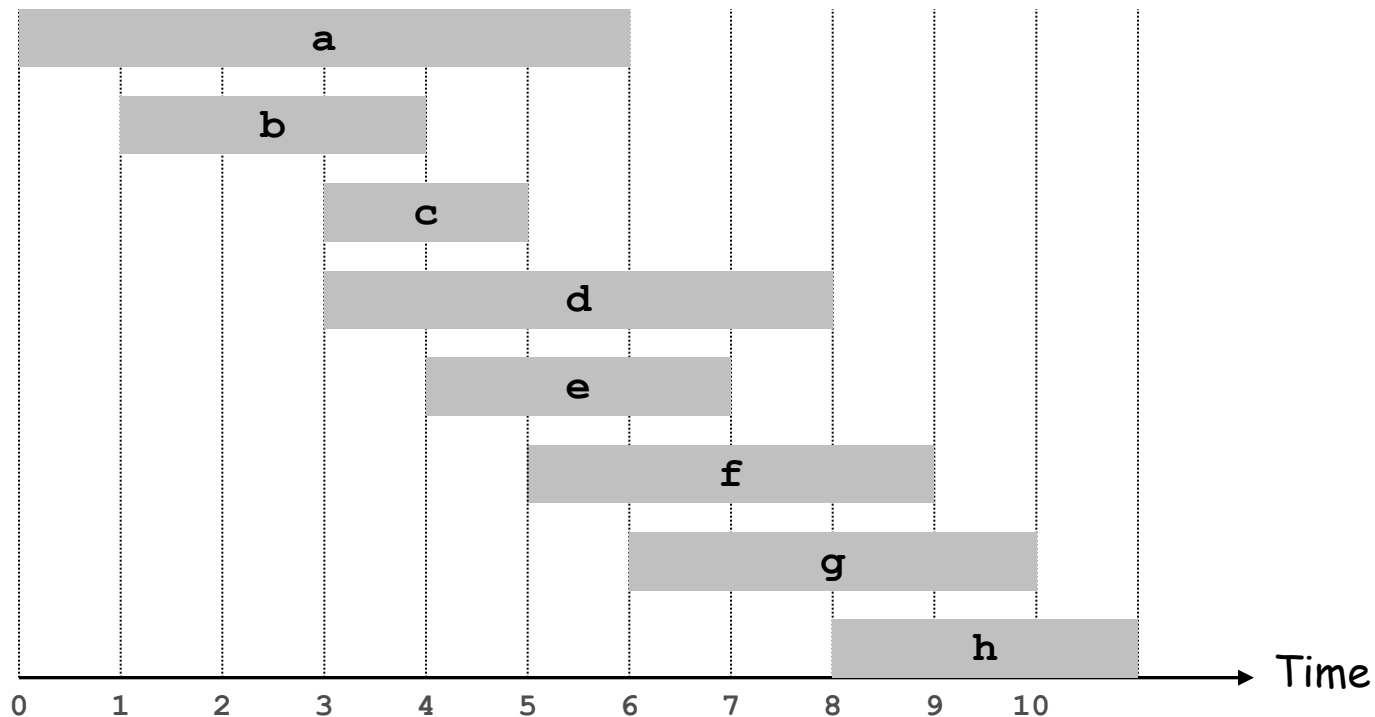
Weighted Interval Scheduling

Weighted interval scheduling problem.

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

maximum subset of mutually compatible jobs

Interval Scheduling

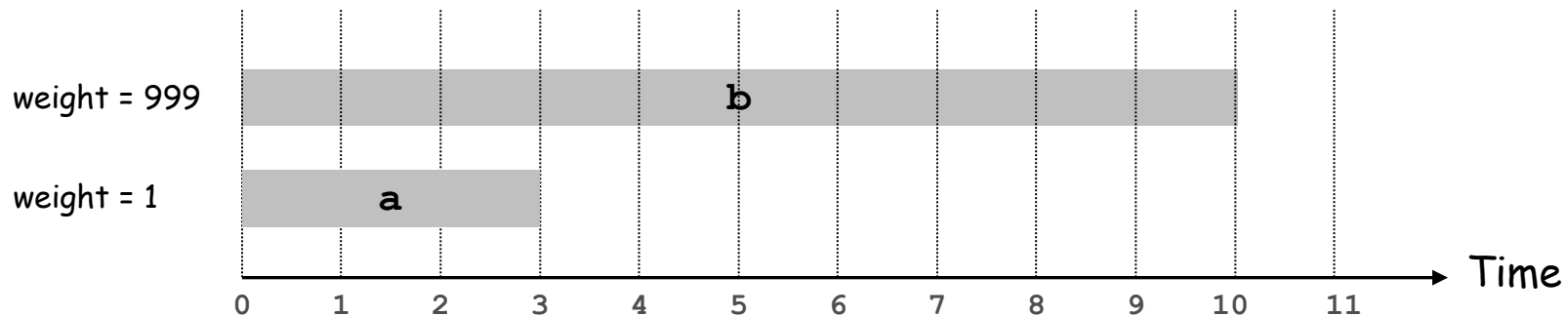


Unweighted Interval Scheduling Review

Recall. Greedy algorithm works if all weights are 1.

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

Observation. Greedy algorithm can fail spectacularly if arbitrary weights are allowed.

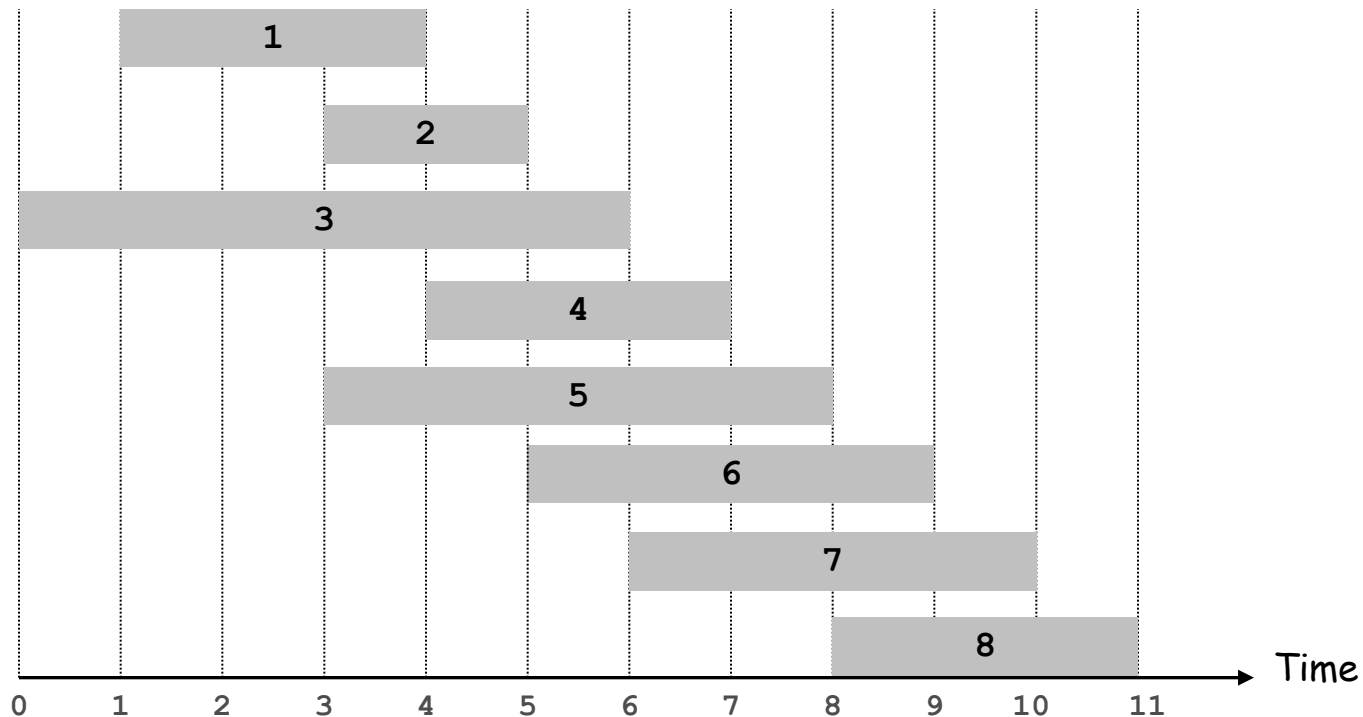


Weighted Interval Scheduling

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



Dynamic Programming: Binary Choice

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

- Case 1: $OPT(j)$ selects job j.
 - collect profit v_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, ..., $p(j)$
- Case 2: $OPT(j)$ does not select job j.
 - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., j-1

↖
↙
optimal substructure

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

Dynamic Programming: optimal solution

From the previous slide, we can observe that Request j belongs to an optimal solution on the set $\{1, 2, \dots, j\}$ if and only if

$$v_j + \text{OPT}(p(j)) \geq \text{OPT}(j - 1)$$

Weighted Interval Scheduling: Brute Force

Brute force algorithm.

Input: $n, s_1, \dots, s_n, f_1, \dots, f_n, v_1, \dots, v_n$

Sort jobs by finish times so that $f_1 \leq f_2 \leq \dots \leq f_n$.

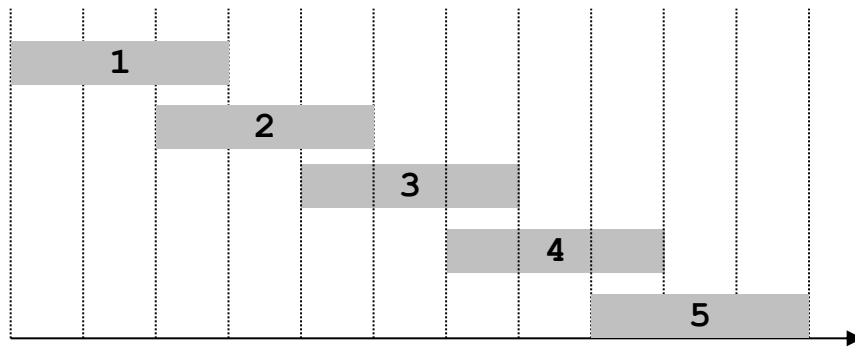
Compute $p(1), p(2), \dots, p(n)$

```
Compute-Opt(j) {  
    if (j = 0)  
        return 0  
    else  
        return max( $v_j + \text{Compute-Opt}(p(j))$ ,  $\text{Compute-Opt}(j-1)$ )  
}
```

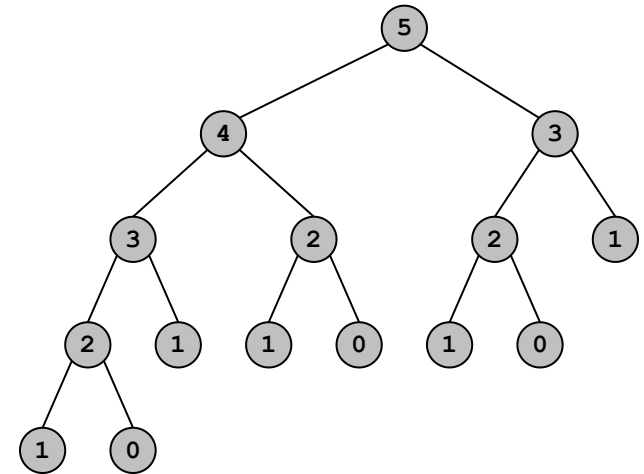
Weighted Interval Scheduling: Brute Force

Observation. Recursive algorithm fails spectacularly because of redundant sub-problems \Rightarrow exponential algorithms.

Ex. Number of recursive calls for family of "layered" instances grows like Fibonacci sequence ($F(0)=1$, $F(1)=1$, $F(n)=F(n-1)+F(n-2)$ ($n \geq 2$, $n \in \mathbb{N}^*$)).



$$p(1) = 0, p(j) = j-2$$



$$\text{Max}((v_j + \text{OPT}(p(j))) , \text{OPT}(j - 1)) \longrightarrow \text{Max}((v_j + \text{OPT}(j-2)) , \text{OPT}(j - 1))$$



Weighted Interval Scheduling: Memoization

Memoization. Store results of each sub-problem in a cache; lookup as needed.

Input: $n, s_1, \dots, s_n, f_1, \dots, f_n, v_1, \dots, v_n$

Sort jobs by finish times so that $f_1 \leq f_2 \leq \dots \leq f_n$.

Compute $p(1), p(2), \dots, p(n)$

for $j = 1$ to n

$M[j] = \text{empty}$

$M[0] = 0$

 global array

M-Compute-Opt(j) {

if ($M[j]$ is empty)

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

return $M[j]$

}

Weighted Interval Scheduling: Running Time

Claim. Memoized version of algorithm takes $O(n \log n)$ time.

- Sort by finish time: $O(n \log n)$.
- Computing $p(\cdot)$: $O(n \log n)$.
- $M\text{-Compute-Opt}(j)$: each invocation takes $O(1)$ time and either
 - (i) returns an existing value $M[j]$
 - (ii) fills in one new entry $M[j]$ and makes two recursive calls
- Progress measure $\Phi = \#$ nonempty entries of $M[\]$.
 - initially $\Phi = 0$, throughout $\Phi \leq n$.
 - (ii) increases Φ by 1 \Rightarrow at most $2n$ recursive calls.
- Overall running time of $M\text{-Compute-Opt}(n)$ is $O(n)$. ▪

Remark. $O(n)$ if jobs are pre-sorted by finish times.

Weighted Interval Scheduling: Finding a Solution

Q. Dynamic programming algorithms computes optimal value.
What if we want the solution itself?

A. Do some post-processing.

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

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

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

Weighted Interval Scheduling: Bottom-Up

Bottom-up dynamic programming. Unwind recursion.

Backtracking



```
Input:  $n, s_1, \dots, s_n, f_1, \dots, f_n, v_1, \dots, v_n$ 
```

```
Sort jobs by finish times so that  $f_1 \leq f_2 \leq \dots \leq f_n$ .
```

```
Compute  $p(1), p(2), \dots, p(n)$ 
```

```
Iterative-Compute-Opt {  
     $M[0] = 0$   
    for  $j = 1$  to  $n$   
         $M[j] = \max(v_j + M[p(j)], M[j-1])$   
}
```

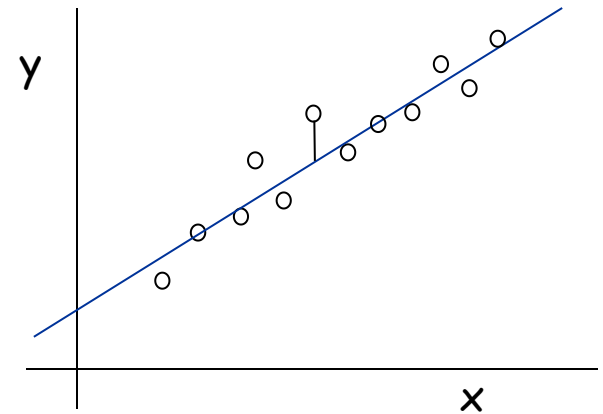

6.3 Segmented Least Squares

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

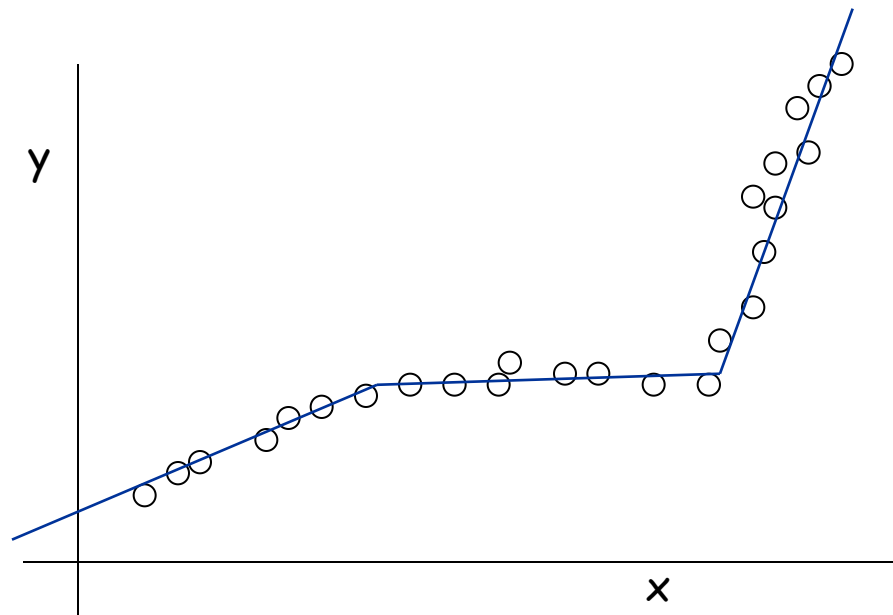
Segmented least squares.

- 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 $f(x)$.

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

↑
number of lines

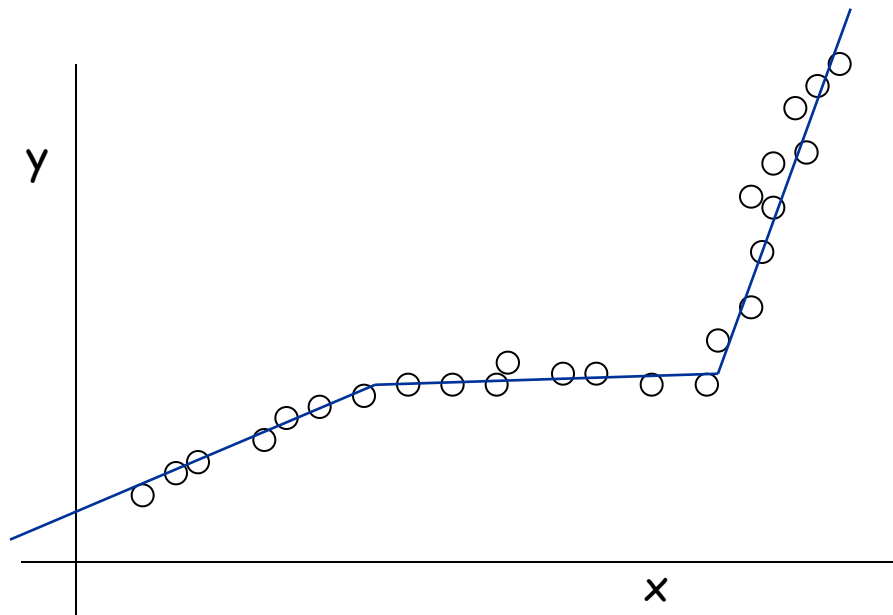
↑
goodness of fit



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: $E + c L$, for some constant $c > 0$.



Dynamic Programming: Multiway Choice

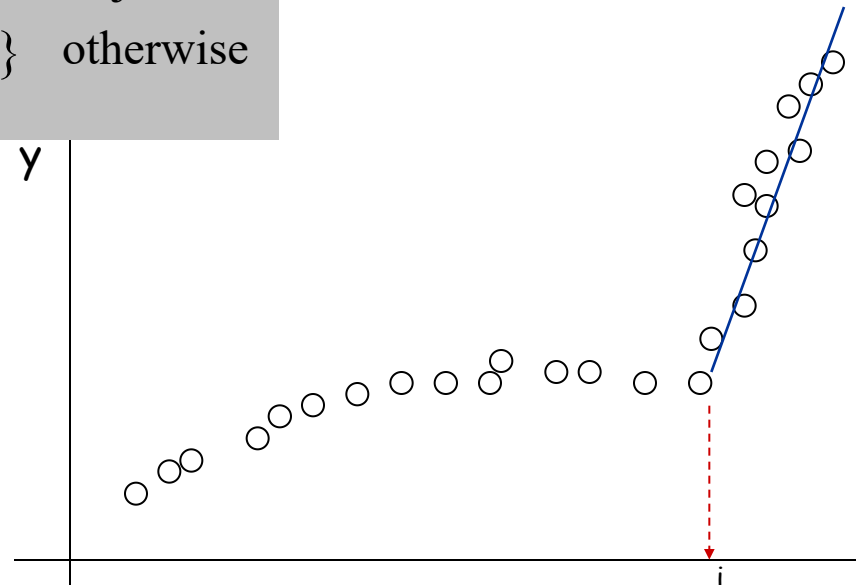
Notation.

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

To compute $OPT(j)$:

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


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



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]  
}
```

Running time. $O(n^3)$.  can be improved to $O(n^2)$ by pre-computing various statistics

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

6.4 Knapsack Problem

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.

Ex: { 3, 4 } has value 40.

$$W = 11$$

#	value	weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Greedy 1: repeatedly add item with maximum value v_i

Greedy 2: repeatedly add item with maximum ratio v_i / w_i .

Ex: { 5, 2, 1 } achieves only value = 35 \Rightarrow greedy not optimal.

Dynamic Programming: False Start

Def. $\text{OPT}(i)$ = max profit subset of items $1, \dots, i$.

- Case 1: $\text{OPT}(i)$ does not select item i .
 - $\text{OPT}(i)$ selects best of $\{ 1, 2, \dots, i-1 \}$
- Case 2: $\text{OPT}(i)$ selects item i .
 - accepting item i does not immediately imply that we will have to reject other items
 - without knowing what other items were selected before i , we don't even know if we have enough room for i

Conclusion. Need more sub-problems!

Dynamic Programming: Adding a New Variable

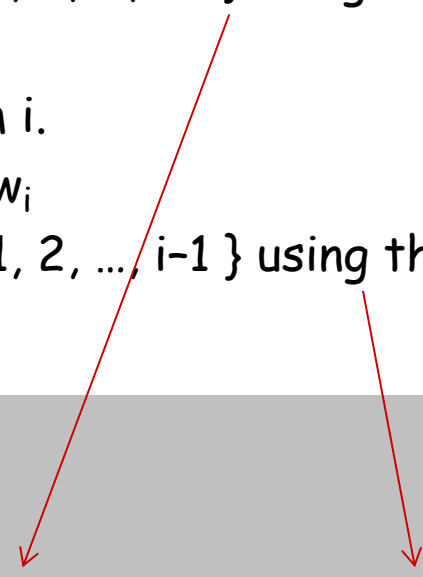
Def. $\text{OPT}(i, w)$ = max profit subset of items $1, \dots, i$ with weight limit w .

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

Dynamic Programming: Adding a New Variable

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

- 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


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

Knapsack Problem: Bottom-Up


Knapsack. Fill up an n -by- W array.

```
Input:  $n, W, w_1, \dots, w_N, v_1, \dots, v_N$ 
```

```
for  $w = 0$  to  $W$ 
```

```
   $M[0, w] = 0$ 
```

$M[i, 0] = 0$



```
for  $i = 1$  to  $n$ 
```

```
  for  $w = 1$  to  $W$ 
```

```
    if  $(w_i > w)$ 
```

```
       $M[i, w] = M[i-1, w]$ 
```

```
    else
```

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

```
return  $M[n, W]$ 
```

Backtracking



Knapsack Algorithm

$$W + 1$$

		0	1	2	3	4	5	6	7	8	9	10	11
<div> <div>n + 1</div> <div></div> <div></div> <div></div> <div></div> <div></div> </div>	ϕ	0	0	0	0	0	0	0	0	0	0	0	0
	$\{1\}$	0	1	1	1	1	1	1	1	1	1	1	1
	$\{1, 2\}$	0	1	6	7	7	7	7	7	7	7	7	7
	$\{1, 2, 3\}$	0	1	6	7	7	18	19	24	25	25	25	25
	$\{1, 2, 3, 4\}$	0	1	6	7	7	18	22	24	28	29	29	40
	$\{1, 2, 3, 4, 5\}$	0	1	6	7	7	18	22	28	29	34	35	40

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

$$W = 11$$

Knapsack Algorithm

		W + 1 →											
		0	1	2	3	4	5	6	7	8	9	10	11
n + 1 ↓	ϕ	0	0	0	0	0	0	0	0	0	0	0	0
	{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
	{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
	{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
	{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
	{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	35	40

Input: $n, W, w_1, \dots, w_N, v_1, \dots, v_N$

for $w = 0$ to W

$M[0, w] = 0$

$M[i, 0] = 0$

for $i = 1$ to n

for $w = 1$ to W

if $(w_i > w)$

$M[i, w] = M[i-1, w]$

else

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

return $M[n, W]$

$W = 11$

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

Knapsack Algorithm

$W + 1$ →

$n + 1$
↓

	0	1	2	3	4	5	6	7	8	9	10	11
ϕ	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	35	40

(A red arrow points from 'i=1' to the first row of the table, and a red box highlights the row for { 1 }.)

Input: $n, W, w_1, \dots, w_N, v_1, \dots, v_N$

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

```
for i = 1 to n
```

```
    for w = 1 to W
        if ( $w_i > w$ )
            M[i, w] = M[i-1, w]
        else
            M[i, w] = max {M[i-1, w],  $v_i + M[i-1, w-w_i]$ }
```

```
return M[n, W]
```

$W = 11$

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

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	35	40

Diagram annotations:
 - A red arrow labeled $i=2$ points to the row {1, 2}.
 - A blue arrow points from the code block to the cell $M[2, 1]$.
 - A red box highlights the row {1, 2} from column 2 to 11.
 - A red box highlights the cell $M[2, 1]$.
 - A red box highlights the cell $M[2, 2]$.
 - A red box highlights the cell $M[2, 3]$.
 - A red box highlights the cell $M[2, 4]$.
 - A red box highlights the cell $M[2, 5]$.
 - A red box highlights the cell $M[2, 6]$.
 - A red box highlights the cell $M[2, 7]$.
 - A red box highlights the cell $M[2, 8]$.
 - A red box highlights the cell $M[2, 9]$.
 - A red box highlights the cell $M[2, 10]$.
 - A red box highlights the cell $M[2, 11]$.

Input: $n, W, w_1, \dots, w_N, v_1, \dots, v_N$

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

```
for i = 1 to n
```

```
    for w = 1 to W
        if ( $w_i > w$ )
            M[i, w] = M[i-1, w]
        else
            M[i, w] = max {M[i-1, w],  $v_i + M[i-1, w-w_i]$ }
```

```
return M[n, W]
```

$W = 11$

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

Knapsack Algorithm

$W + 1$ →

$n + 1$ ↓

	0	1	2	3	4	5	6	7	8	9	10	11
ϕ	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	35	40

Note: Red arrows from i=3 point to the values 1, 6, 7, 7, 18, 19, 24, 25 in the row {1, 2, 3}. A blue box highlights the sub-problem state (i=3, w=4) with value 7.

Input: $n, W, w_1, \dots, w_N, v_1, \dots, v_N$

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

for $i = 1$ to n

for $w = 1$ to W
 if $(w_i > w)$
 $M[i, w] = M[i-1, w]$
 else
 $M[i, w] = \max \{M[i-1, w], v_i + M[i-1, w-w_i]\}$

return $M[n, W]$

$W = 11$

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

Knapsack Algorithm

$W + 1$ →

$n + 1$ ↓

	0	1	2	3	4	5	6	7	8	9	10	11
ϕ	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	35	40

Note: Red arrows indicate the path for item 4 (i=4) from the bottom-right cell (40) back to the cell (18) at (i-1, w-w₄).

Input: $n, W, w_1, \dots, w_N, v_1, \dots, v_N$

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

```
for i = 1 to n
```

```
    for w = 1 to W
        if ( $w_i > w$ )
            M[i, w] = M[i-1, w]
        else
            M[i, w] = max {M[i-1, w],  $v_i + M[i-1, w-w_i]$ }
```

```
return M[n, W]
```

$W = 11$

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

Knapsack Algorithm

$W + 1$ →

$n + 1$ ↓

	0	1	2	3	4	5	6	7	8	9	10	11
ϕ	0	0	0	0	0	0	0	0	0	0	0	0
{1}	0	1	1	1	1	1	1	1	1	1	1	1
{1, 2}	0	1	6	7	7	7	7	7	7	7	7	7
{1, 2, 3}	0	1	6	7	7	18	19	24	25	25	25	25
{1, 2, 3, 4}	0	1	6	7	7	18	22	24	28	29	29	40
{1, 2, 3, 4, 5}	0	1	6	7	7	18	22	28	29	34	35	40

Note: Red arrows indicate the calculation path for the bottom-right cell (i=5, W=11). A blue box highlights the subproblem (i=4, W=6) used in the calculation.

Input: $n, W, w_1, \dots, w_N, v_1, \dots, v_N$

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

```
for i = 1 to n
```

```
    for w = 1 to W
        if ( $w_i > w$ )
            M[i, w] = M[i-1, w]
        else
            M[i, w] = max {M[i-1, w],  $v_i + M[i-1, w-w_i]$ }
```

```
return M[n, W]
```

$W = 11$

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

Knapsack Algorithm

		W + 1 →											
		0	1	2	3	4	5	6	7	8	9	10	11
n + 1 ↓	ϕ	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

Knapsack Problem: Running Time

Running time. $\Theta(n W)$.

- Not polynomial in input size!
- "Pseudo-polynomial."

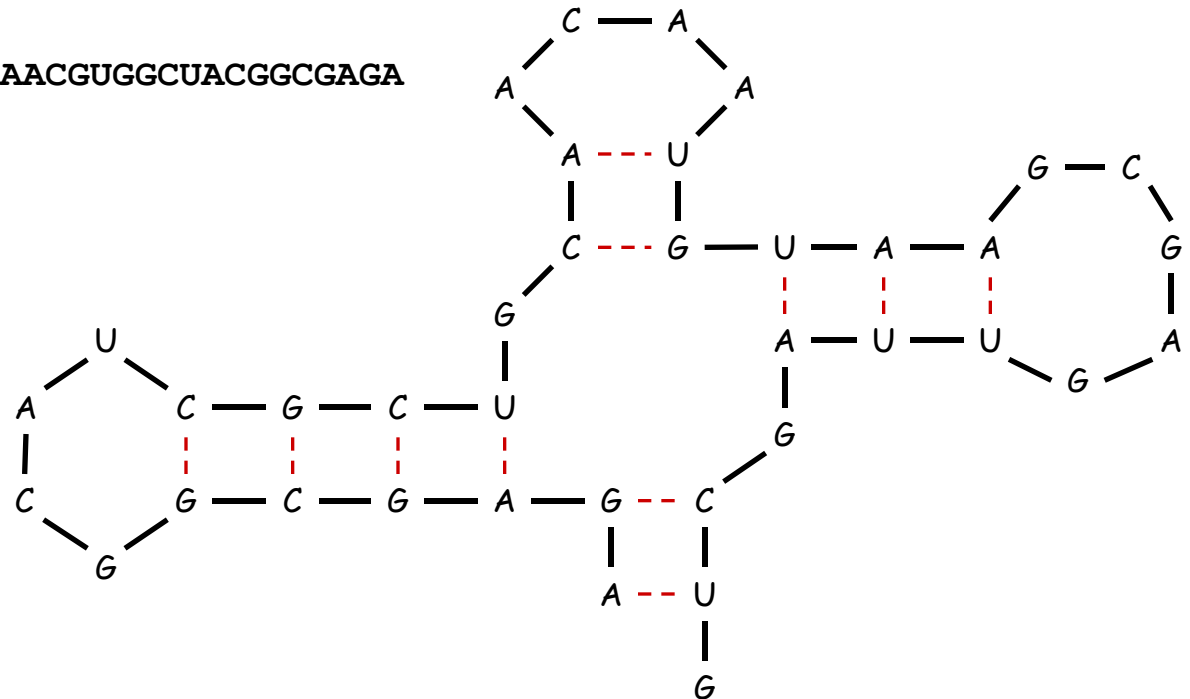
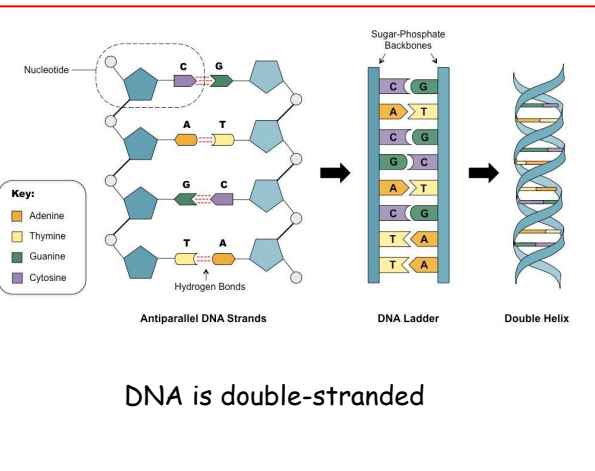
6.5 RNA Secondary Structure

RNA 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



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 $i < k < j < l$.

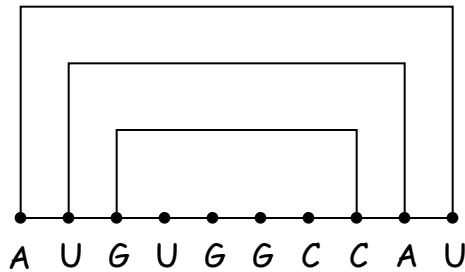
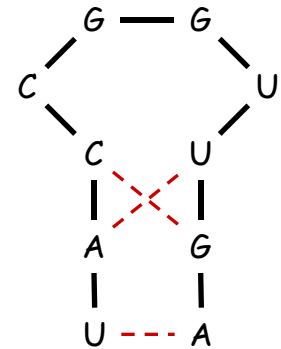
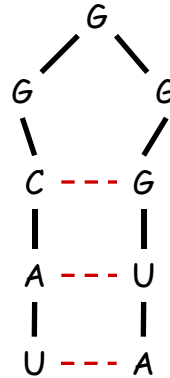
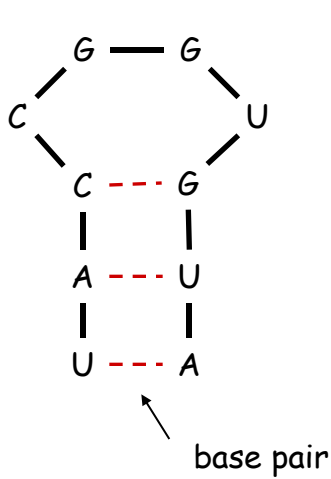
Free energy. Usual hypothesis is that an RNA molecule will form the secondary structure with the optimum total free energy.

↖
approximate by number of base pairs

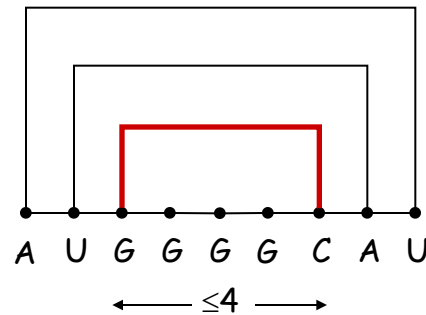
Goal. Given an RNA molecule $B = b_1b_2\dots b_n$, find a secondary structure S that maximizes the number of base pairs.

RNA Secondary Structure: Examples

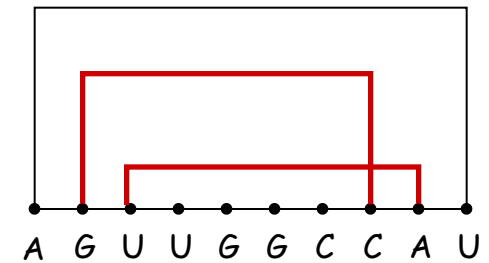
Examples.



ok



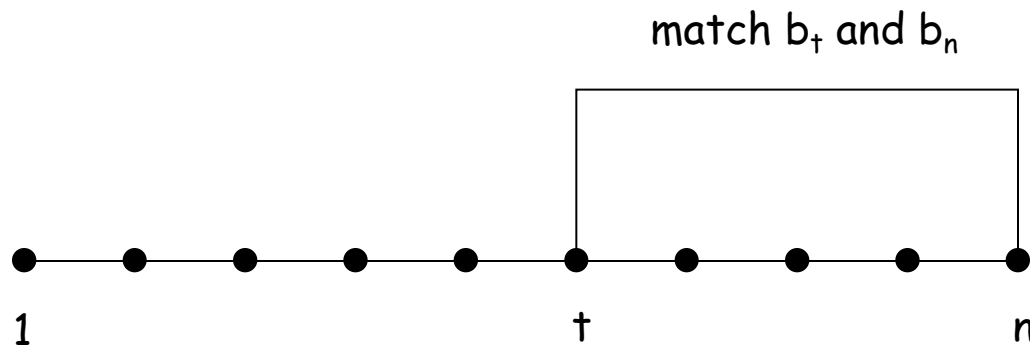
sharp turn



crossing

RNA Secondary Structure: Subproblems

First attempt. $\text{OPT}(j)$ = maximum number of base pairs in a secondary structure of the substring $b_1b_2\dots b_j$.



Difficulty. Results in two sub-problems.

- Finding secondary structure in: $b_1b_2\dots b_{t-1}$. $\leftarrow \text{OPT}(t-1)$
- Finding secondary structure in: $b_{t+1}b_{t+2}\dots b_{n-1}$. \leftarrow need more sub-problems

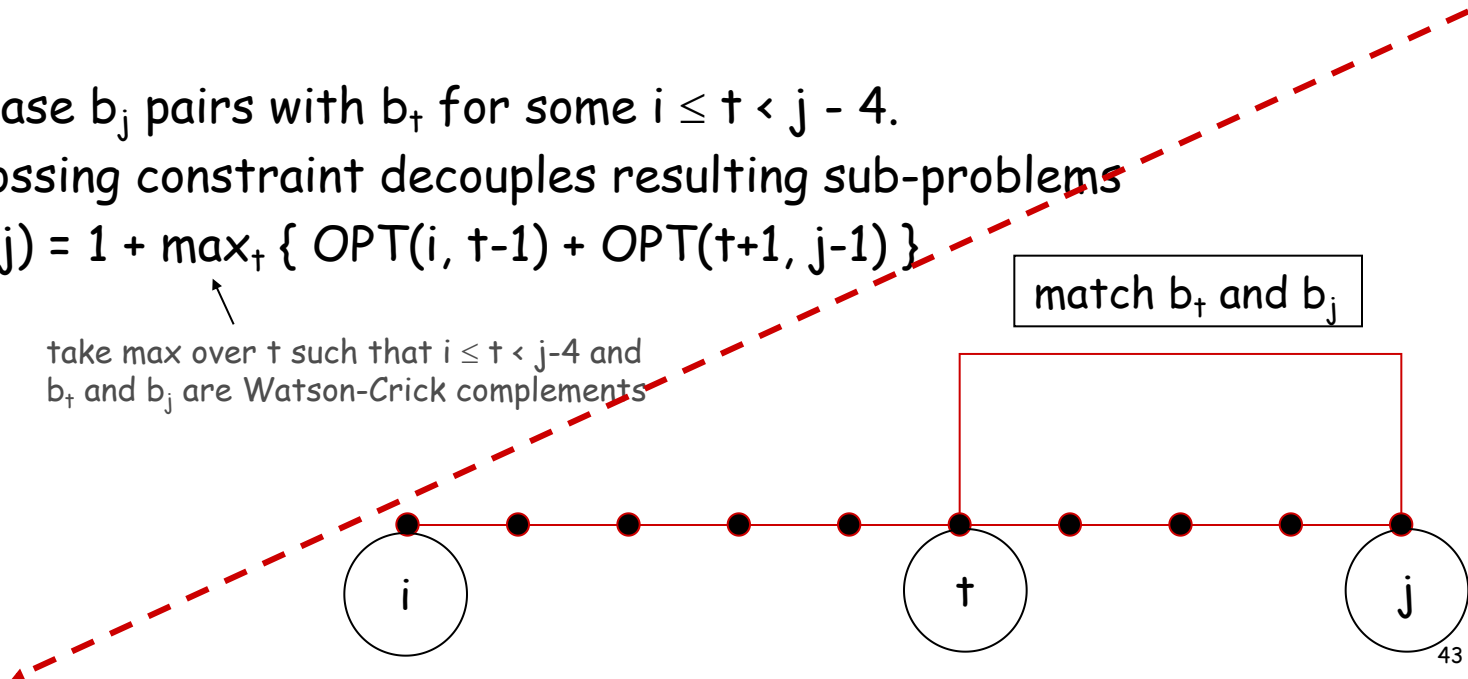
Dynamic Programming Over Intervals

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

- Case 1. If $i \geq j - 4$.
 - $OPT(i, j) = 0$ by no-sharp turns condition.
- Case 2. Base b_j is not involved in a pair.
 - $OPT(i, j) = OPT(i, j-1)$
- Case 3. Base b_j pairs with b_t for some $i \leq t < j - 4$.
 - non-crossing constraint decouples resulting sub-problems
 - $OPT(i, j) = 1 + \max_t \{ OPT(i, t-1) + OPT(t+1, j-1) \}$

take max over t such that $i \leq t < j-4$ and b_t and b_j are Watson-Crick complements

match b_t and b_j



Bottom Up Dynamic Programming Over Intervals

Q. What order to solve the sub-problems?

A. Do shortest intervals first.

```
RNA( $b_1, \dots, b_n$ ) {  
  for  $k = 5, 6, \dots, n-1$   
    for  $i = 1, 2, \dots, n-k$   
       $j = i + k$   
      Compute  $M[i, j]$   
  
  return  $M[1, n]$   
}
```

using recurrence

$O(n)$

$O(n)$

$O(n)$

i

4	0	0	0	↗
3	0	0	↗	↗
2	0	↗	↗	↗
1	↗	↗	↗	↗
	6	7	8	9

j

Running time. $O(n^3)$.

Dynamic Programming Summary

Recipe.

- Characterize structure of problem.
- Recursively define value of optimal solution.
- Compute value of optimal solution.
- Construct optimal solution from computed information.

Dynamic programming techniques.

- Binary choice: weighted interval scheduling.
- Multi-way choice: segmented least squares.
- Adding a new variable: knapsack.
- Dynamic programming over intervals: RNA secondary structure.

Viterbi algorithm for HMM also uses DP to optimize a maximum likelihood tradeoff between parsimony and accuracy

Top-down vs. bottom-up: different people have different intuitions.

6.6 Sequence Alignment

String Similarity

How similar are two strings?

- **ocurrence**
- **occurrence**

o	c	u	r	r	a	n	c	e	-
o	c	c	u	r	r	e	n	c	e

6 mismatches, 1 gap

o	c	-	u	r	r	a	n	c	e
o	c	c	u	r	r	e	n	c	e

1 mismatch, 1 gap

o	c	-	u	r	r	-	a	n	c	e
o	c	c	u	r	r	e	-	n	c	e

0 mismatches, 3 gaps

Edit Distance

Applications.

- Basis for Unix diff.
- Speech recognition.
- Computational biology.

Edit distance. [Levenshtein 1966, Needleman-Wunsch 1970]

- Gap penalty δ ; mismatch penalty α_{pq} .
- Cost = sum of gap and mismatch penalties.

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

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

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

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

$$\alpha_{TC} + \alpha_{GT} + \alpha_{AG} + 2\alpha_{CA}$$

$$2\delta + \alpha_{CA}$$

Sequence Alignment

Goal: Given two strings $X = x_1 x_2 \dots x_m$ and $Y = y_1 y_2 \dots y_n$ find alignment of minimum cost.

Def. An **alignment** M is a set of ordered pairs $x_i - y_j$ such that each item occurs in at most one pair and no crossings.

Def. The pair $x_i - y_j$ and $x_{i'} - y_{j'}$ **cross** if $i < i'$, but $j > j'$.

$$\text{cost}(M) = \underbrace{\sum_{(x_i, y_j) \in M} \alpha_{x_i y_j}}_{\text{mismatch}} + \underbrace{\sum_{i: x_i \text{ unmatched}} \delta + \sum_{j: y_j \text{ unmatched}} \delta}_{\text{gap}}$$

Ex: CTACCG **vs.** TACATG.

Sol: $M = x_2 - y_1, x_3 - y_2, x_4 - y_3, x_5 - y_4, x_6 - y_6$.

x_1	x_2	x_3	x_4	x_5		x_6
C	T	A	C	C	-	G

	y_1	y_2	y_3	y_4	y_5	y_6
-	T	A	C	A	T	G

Sequence Alignment: Problem Structure

Def. $OPT(i, j)$ = min cost of aligning strings $x_1 x_2 \dots x_i$ and $y_1 y_2 \dots y_j$.

- Case 1: OPT aligns x_i - y_j .
 - pay mismatch for x_i - y_j + min cost of aligning two strings $x_1 x_2 \dots x_{i-1}$ and $y_1 y_2 \dots y_{j-1}$
- Case 2a: OPT leaves x_i unmatched.
 - pay gap for x_i and min cost of aligning $x_1 x_2 \dots x_{i-1}$ and $y_1 y_2 \dots y_j$
- Case 2b: OPT leaves y_j unmatched.
 - pay gap for y_j and min cost of aligning $x_1 x_2 \dots x_i$ and $y_1 y_2 \dots y_{j-1}$

$$OPT(i, j) = \begin{cases} j\delta & \text{if } i = 0 \\ \min \begin{cases} \alpha_{x_i y_j} + OPT(i-1, j-1) \\ \delta + OPT(i-1, j) \\ \delta + OPT(i, j-1) \end{cases} & \text{otherwise} \\ i\delta & \text{if } j = 0 \end{cases}$$

Sequence Alignment: Algorithm

```
Sequence-Alignment( $m, n, x_1x_2\dots x_m, y_1y_2\dots y_n, \delta, \alpha$ ) {  
  for  $i = 0$  to  $m$   
     $M[i, 0] = i\delta$   
  for  $j = 0$  to  $n$   
     $M[0, j] = j\delta$   
  
  for  $i = 1$  to  $m$   
    for  $j = 1$  to  $n$   
       $M[i, j] = \min(\alpha[x_i, y_j] + M[i-1, j-1],$   
                     $\delta + M[i-1, j],$   
                     $\delta + M[i, j-1])$   
  
  return  $M[m, n]$   
}
```

Analysis. $\Theta(mn)$ time and space.

English words or sentences: $m, n \leq 10$.


Computational biology: $m = n = 100,000$. 10 billions ops OK, but 10GB array?


6.7 Sequence Alignment in Linear Space

Sequence Alignment: Linear Space

Q. Can we avoid using quadratic **space**?


Easy. Optimal **value** in $O(m + n)$ space and $O(mn)$ time.

- Compute $\text{OPT}(i, \cdot)$ from $\text{OPT}(i-1, \cdot)$.  needs two rows, current one and the previous one
- No longer a simple way to recover alignment itself.

 Remaining issue is to recover alignment

Theorem. [Hirschberg 1975] Optimal **alignment** in $O(m + n)$ space and $O(mn)$ time.

- Clever combination of divide-and-conquer and dynamic programming.
- Inspired by idea of **Savitch** from complexity theory.


if a **nondeterministic Turing machine** can solve a problem using $f(n)$ space,
an ordinary **deterministic Turing machine** can solve the same problem in
the square of that space bound

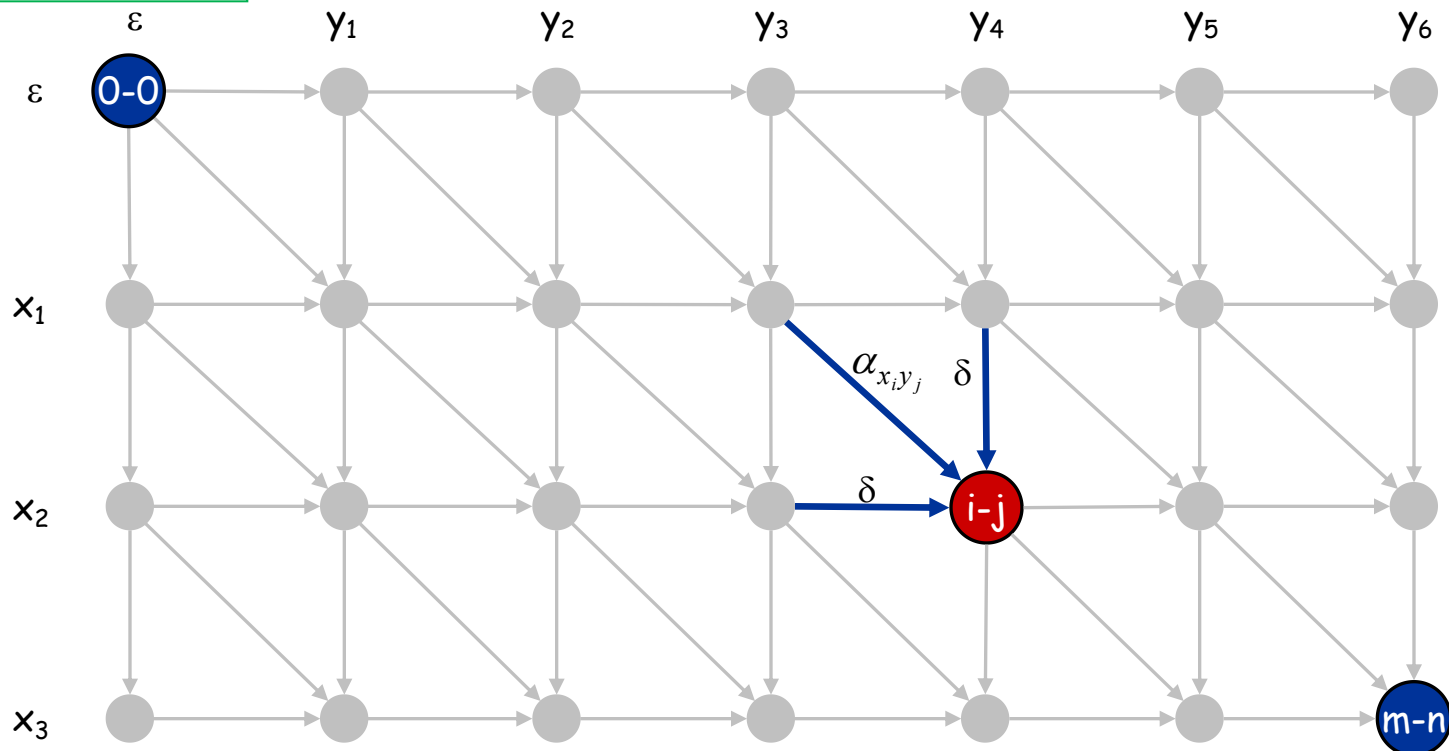
Sequence Alignment: Linear Space

Edit distance graph.

- Let $f(i, j)$ be shortest path from $(0,0)$ to (i, j) .
- Observation: $f(i, j) = \text{OPT}(i, j)$.

the value of the optimal alignment is the length of the shortest path in G_{XY} from $(0, 0)$ to (m, n) .

Shortest corner-to-corner path



Proof. We can easily prove this by induction on $i+j$. When $i+j=0$, we have $i=j=0$, and indeed $f(i, j) = \text{OPT}(i, j) = 0$.

Now consider arbitrary values of i and j , and suppose the statement is true for all pairs (i', j') with $i' + j' < i + j$. The last edge on the shortest path to (i, j) is either from $(i-1, j-1)$, $(i-1, j)$, or $(i, j-1)$. Thus we have

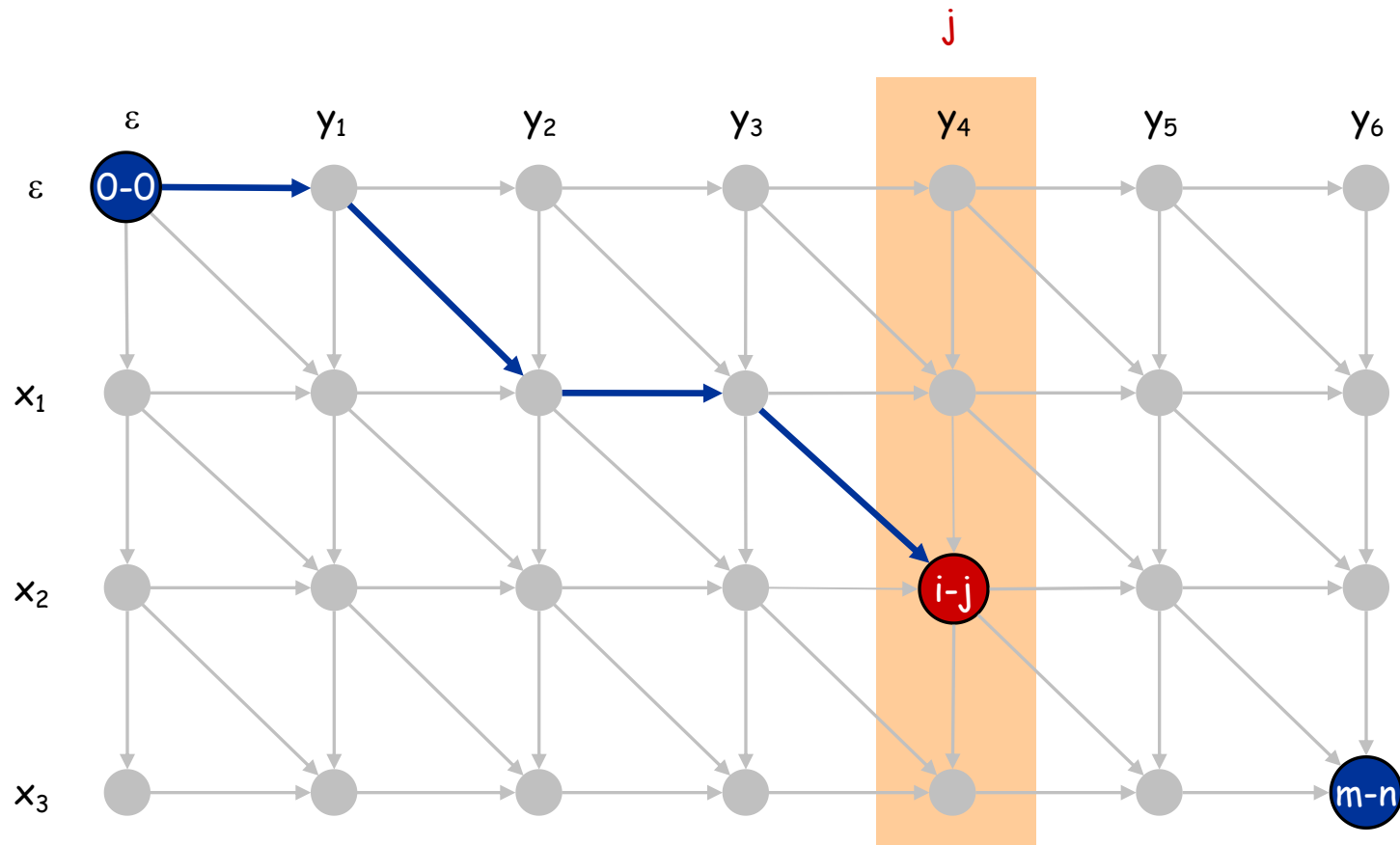
$$\begin{aligned} f(i, j) &= \min[\alpha_{x_i y_j} + f(i-1, j-1), \delta + f(i-1, j), \delta + f(i, j-1)] \\ &= \min[\alpha_{x_i y_j} + \text{OPT}(i-1, j-1), \delta + \text{OPT}(i-1, j), \delta + \text{OPT}(i, j-1)] \\ &= \text{OPT}(i, j), \end{aligned}$$

where we pass from the first line to the second using the induction hypothesis, and we pass from the second to the third using (6.16). ■

Sequence Alignment: Linear Space

Edit distance graph.

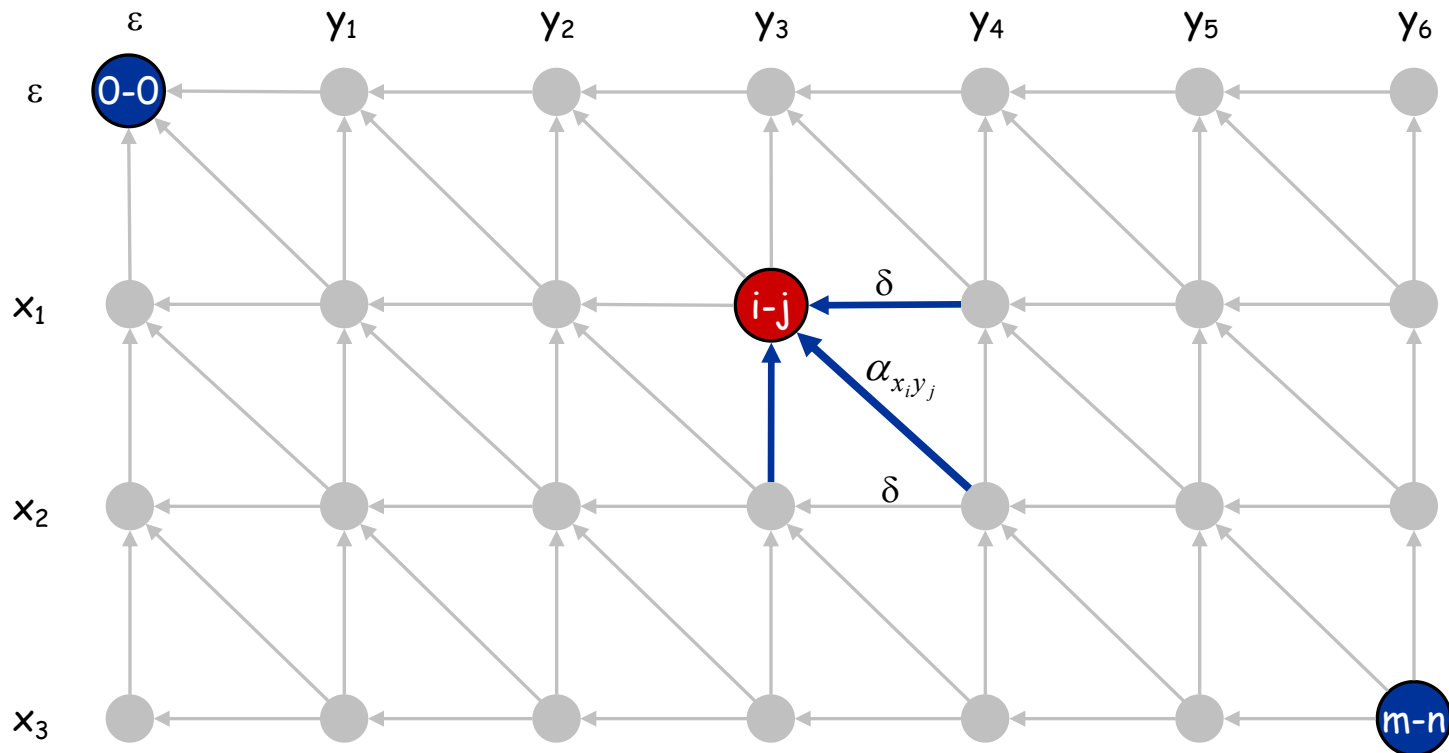
- Let $f(i, j)$ be shortest path from $(0,0)$ to (i, j) .
- Can compute $f(\cdot, j)$ for any j in $O(mn)$ time and $O(m + n)$ space.



Sequence Alignment: Linear Space

Edit distance graph.

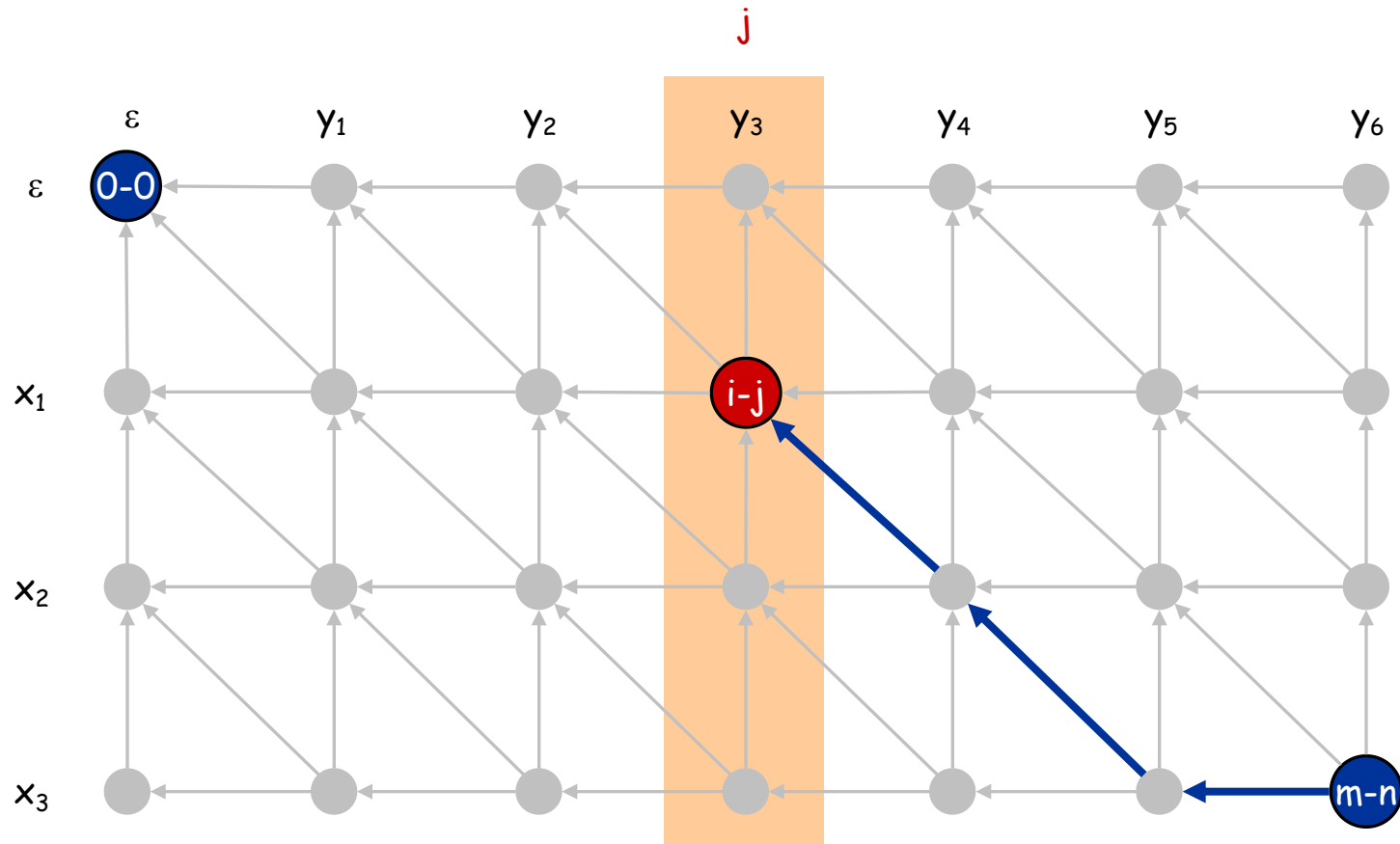
- Let $g(i, j)$ be shortest path from (i, j) to (m, n) .
- Can compute by reversing the edge orientations and inverting the roles of $(0, 0)$ and (m, n)



Sequence Alignment: Linear Space

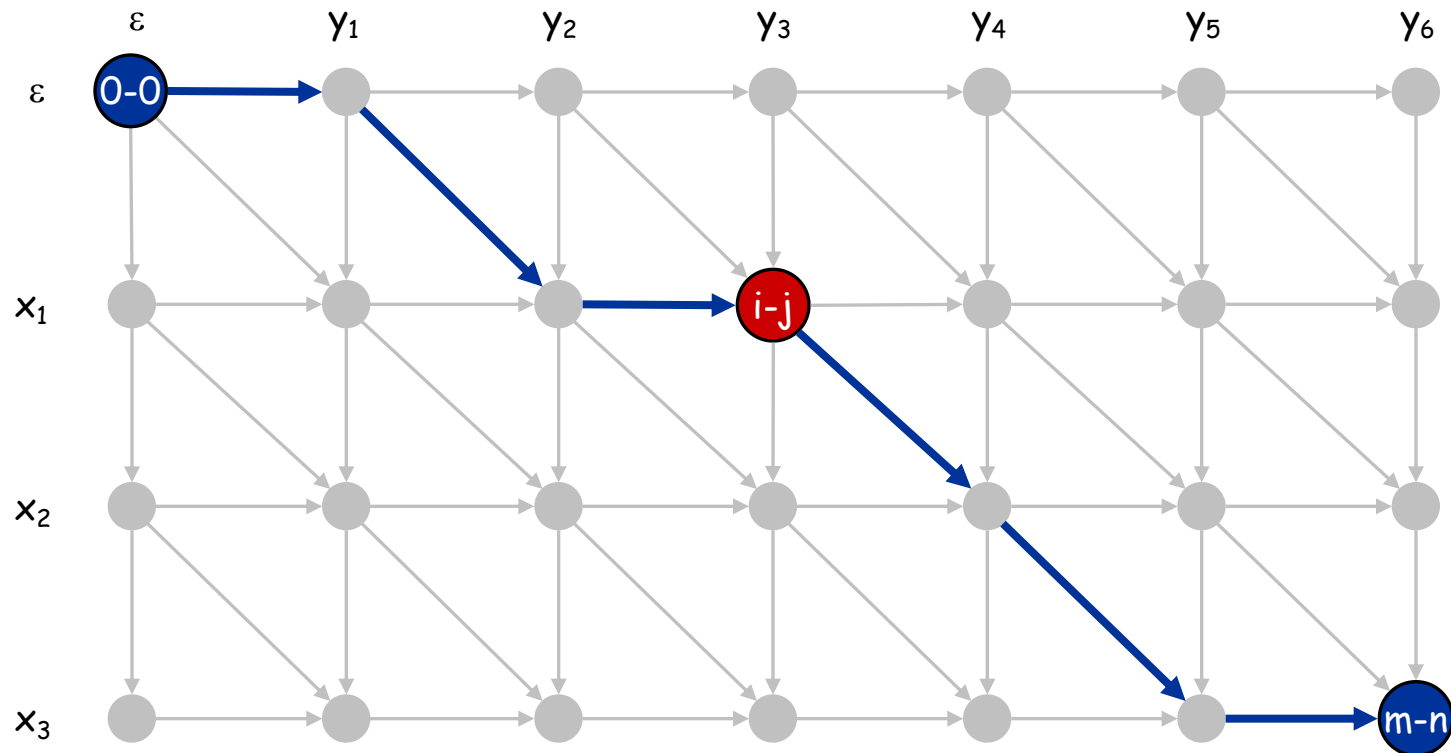
Edit distance graph.

- Let $g(i, j)$ be shortest path from (i, j) to (m, n) .
- Can compute $g(\cdot, j)$ for any j in $O(mn)$ time and $O(m + n)$ space.



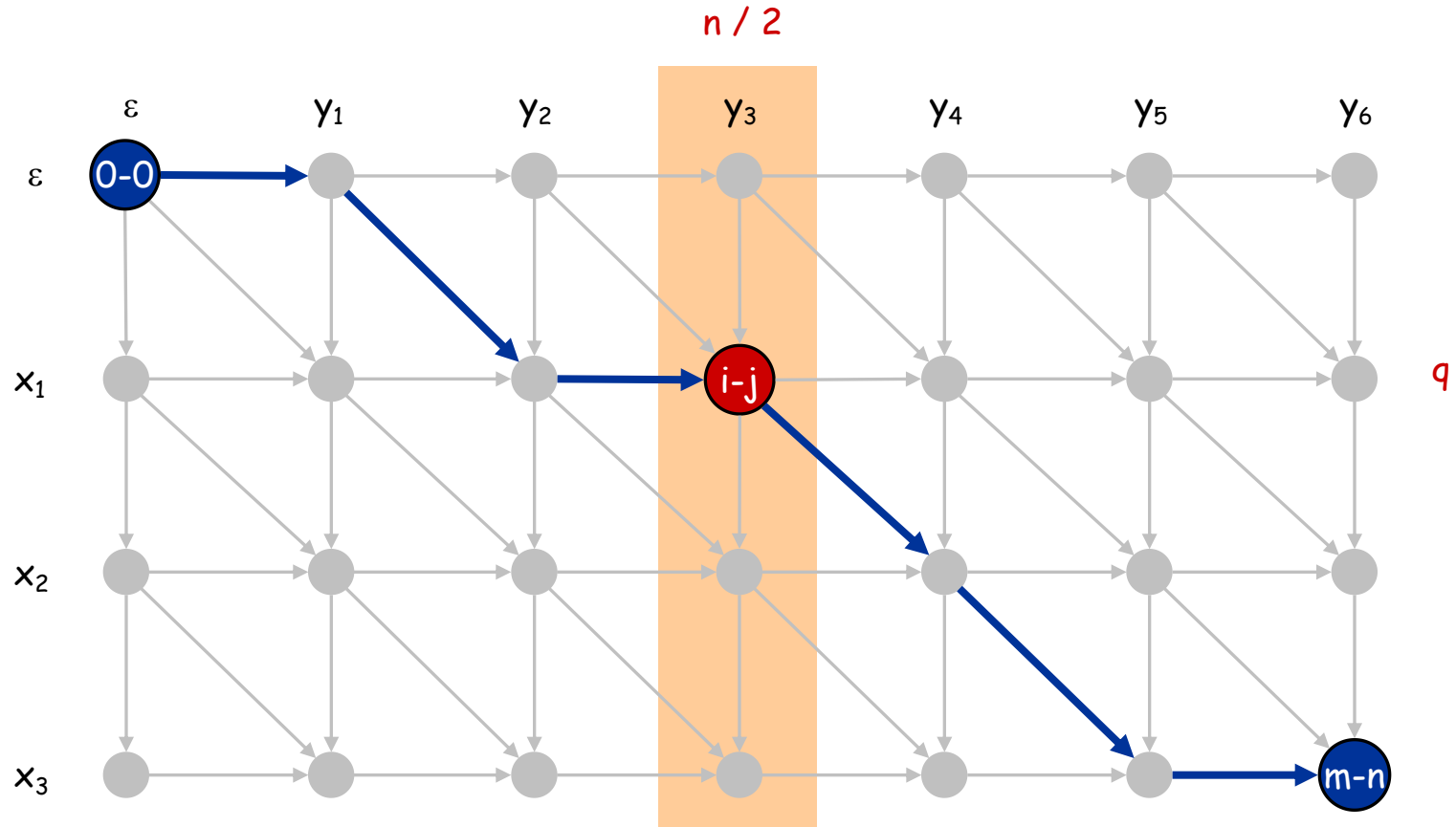
Sequence Alignment: Linear Space

Observation 1. The cost of the shortest path that **uses** (i, j) is $f(i, j) + g(i, j)$.



Sequence Alignment: Linear Space

Observation 2. let q be an index that minimizes $f(q, n/2) + g(q, n/2)$. Then, the shortest path from $(0, 0)$ to (m, n) uses $(q, n/2)$.



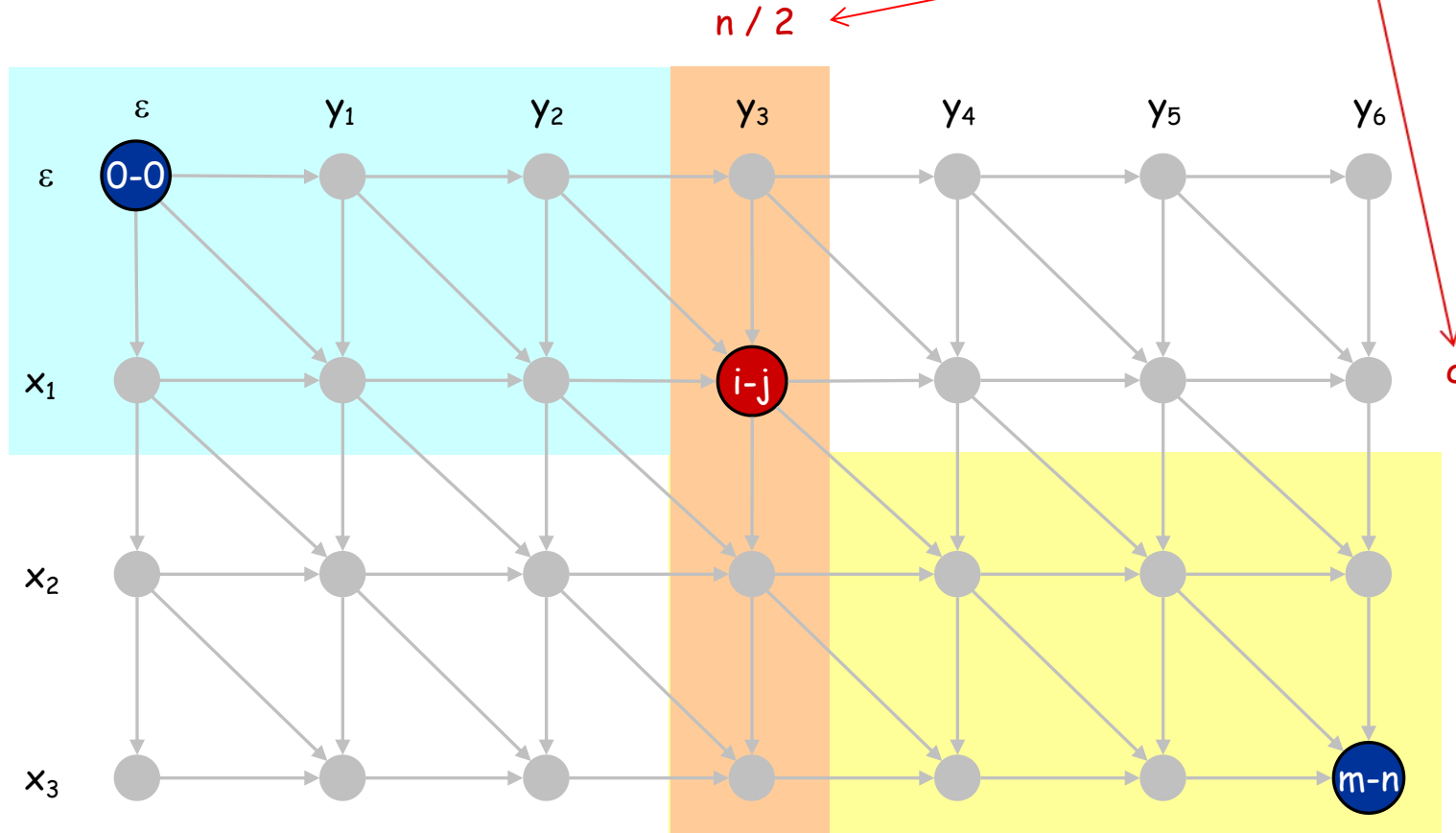
Sequence Alignment: Linear Space

Divide: find index q that minimizes $f(q, n/2) + g(q, n/2)$ using DP.

- Align x_q and $y_{n/2}$.

Conquer: recursively compute optimal alignment in each piece.

alignment ← recursively record the pair → maximum records ($m+n$) ← Record the $(q, n/2)$



Sequence Alignment: Running Time Analysis Warmup

Theorem. Let $T(m, n)$ = max running time of algorithm on strings of length at most m and n . $T(m, n) = O(mn \log n)$.

$$T(m, n) \leq 2T(m, n/2) + O(mn) \Rightarrow T(m, n) = O(mn \log n)$$

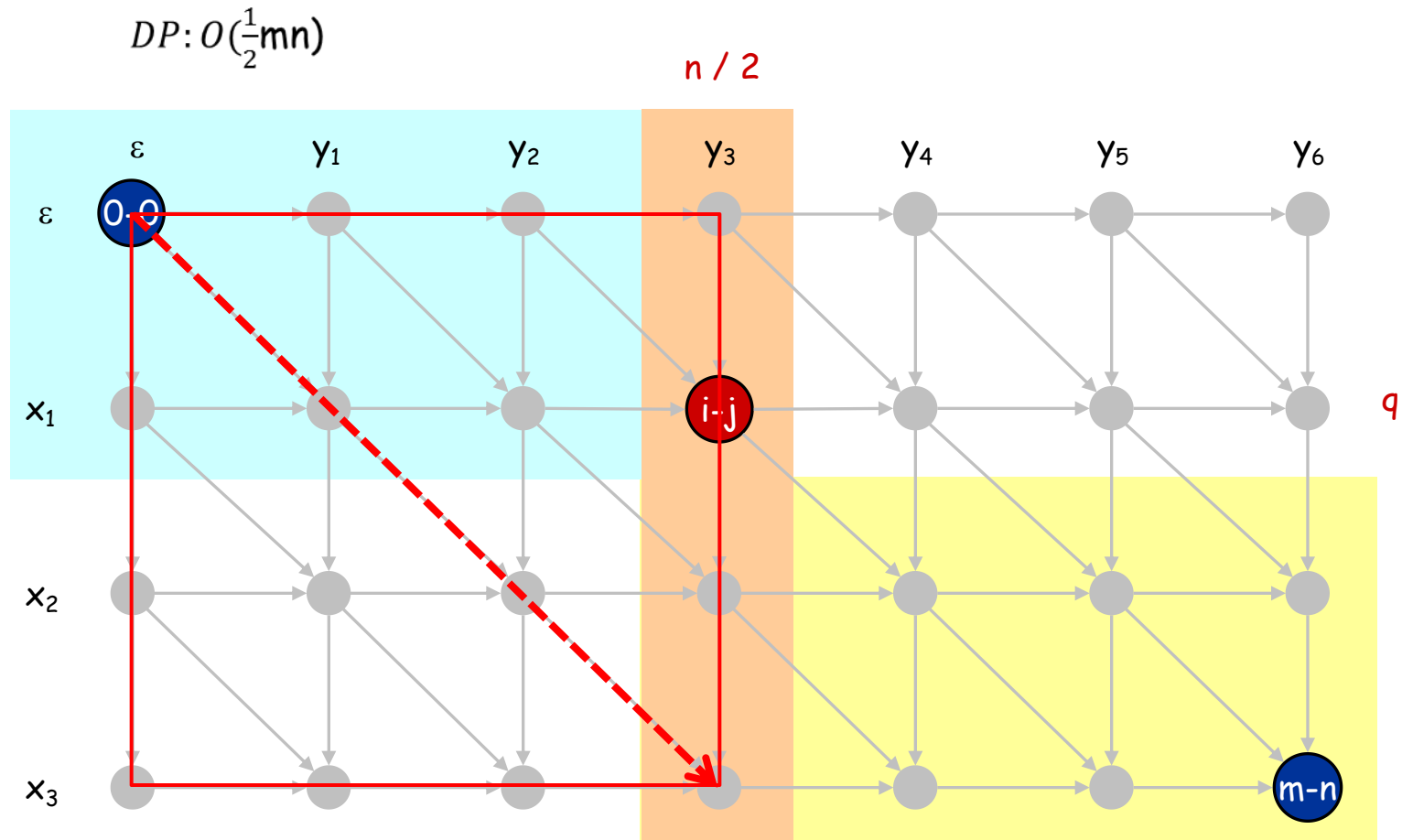
Remark. Analysis is not tight because two sub-problems are of size $(q, n/2)$ and $(m - q, n/2)$. In next slide, we save $\log n$ factor.

Sequence Alignment: Linear Space

Divide: find index q that minimizes $f(q, n/2) + g(q, n/2)$ using DP.

- Align x_q and $y_{n/2}$.

Conquer: recursively compute optimal alignment in each piece.

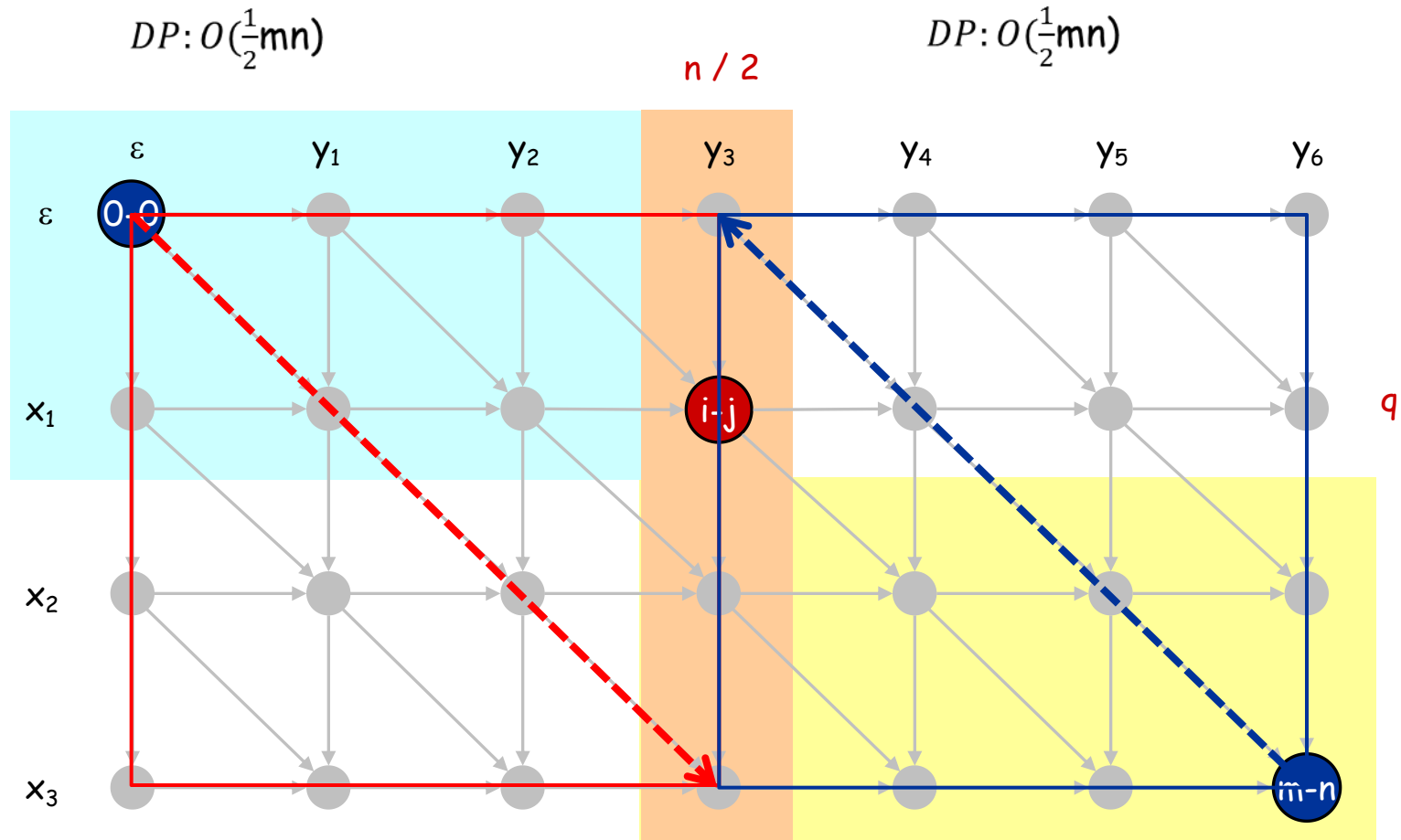


Sequence Alignment: Linear Space

Divide: find index q that minimizes $f(q, n/2) + g(q, n/2)$ using DP.

- Align x_q and $y_{n/2}$.

Conquer: recursively compute optimal alignment in each piece.



Sequence Alignment: Linear Space

Divide: find index q that minimizes $f(q, n/2) + g(q, n/2)$ using DP.

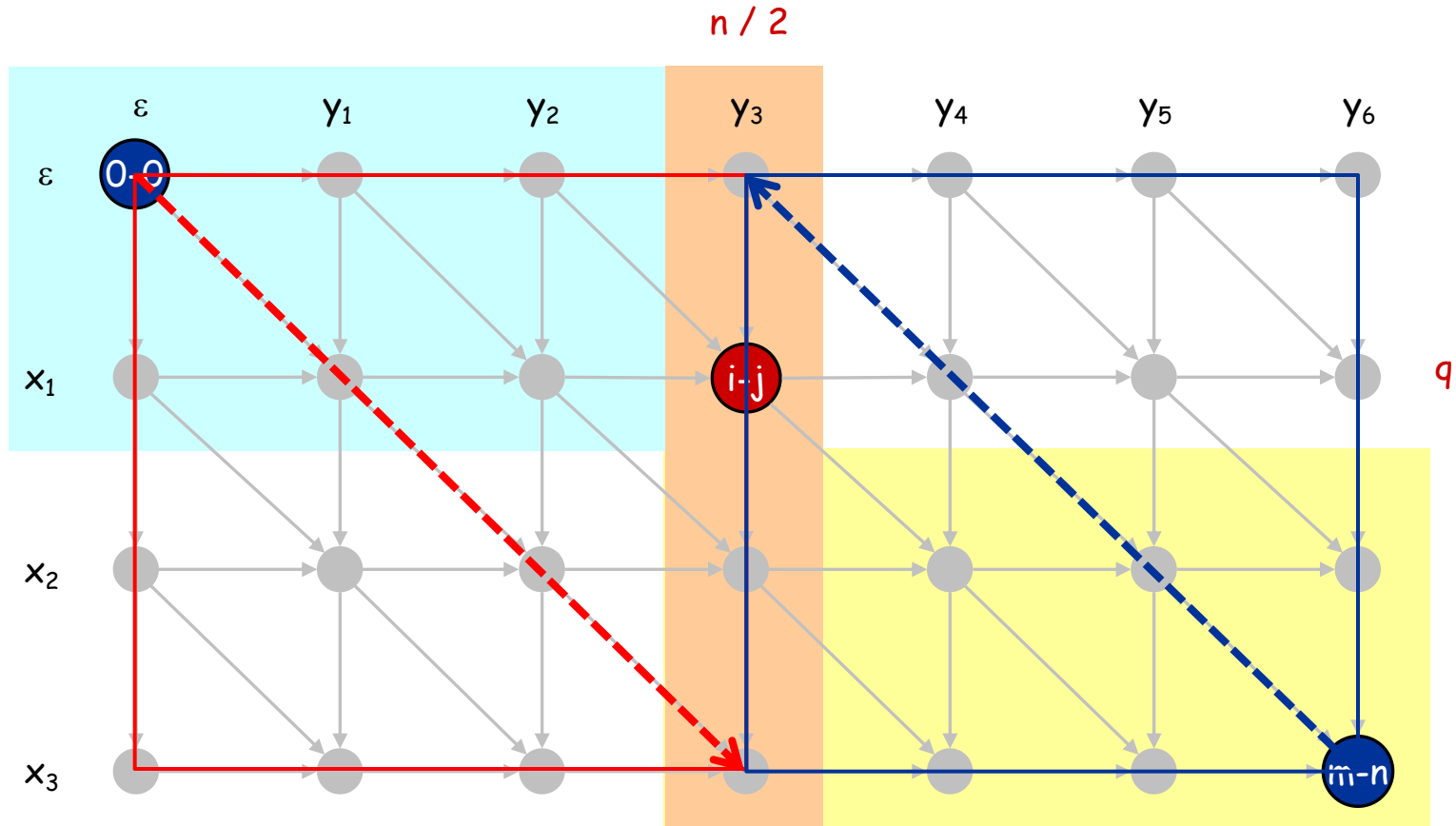
- Align x_q and $y_{n/2}$.

Conquer: recursively compute optimal alignment in each piece.

$$DP: O(\frac{1}{2}mn)$$

$$\text{Find index } q: O(n)$$

$$DP: O(\frac{1}{2}mn)$$



Sequence Alignment: Running Time Analysis

Theorem. Let $T(m, n)$ = max running time of algorithm on strings of length m and n . $T(m, n) = O(mn)$.

Pf. (by induction on n)

- $O(mn)$ time to compute $f(\cdot, n/2)$ and $g(\cdot, n/2)$ and find index q .
- $T(q, n/2) + T(m - q, n/2)$ time for two recursive calls.
- Choose constant c so that:

$$T(m, 2) \leq cm$$

$$T(2, n) \leq cn$$

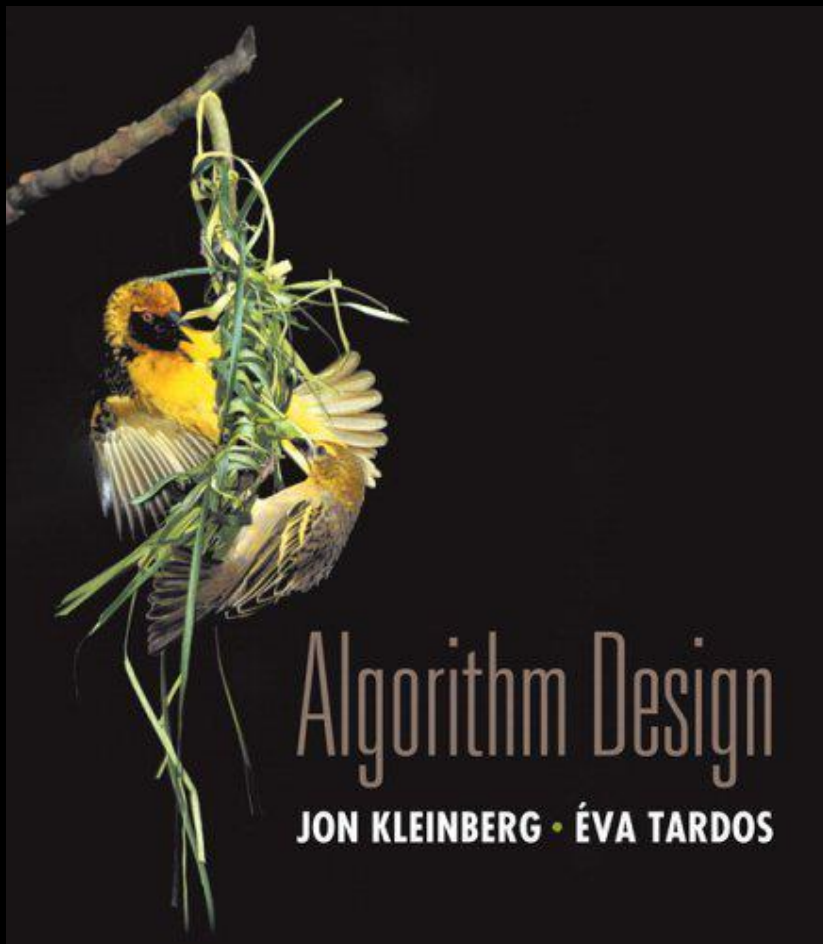
$$T(m, n) \leq cmn + T(q, n/2) + T(m - q, n/2)$$

- Base cases: $m = 2$ or $n = 2$.
- Inductive hypothesis: $T(m', n') \leq 2cm'n'$.

$$\begin{aligned} T(m, n) &\leq T(q, n/2) + T(m - q, n/2) + cmn \\ &\leq 2cq n/2 + 2c(m - q)n/2 + cmn \\ &= cq n + cmn - cq n + cmn \\ &= 2cmn \end{aligned}$$

Chapter 6

Dynamic Programming



Slides by Kevin Wayne.
Copyright © 2005 Pearson-Addison Wesley.
All rights reserved.

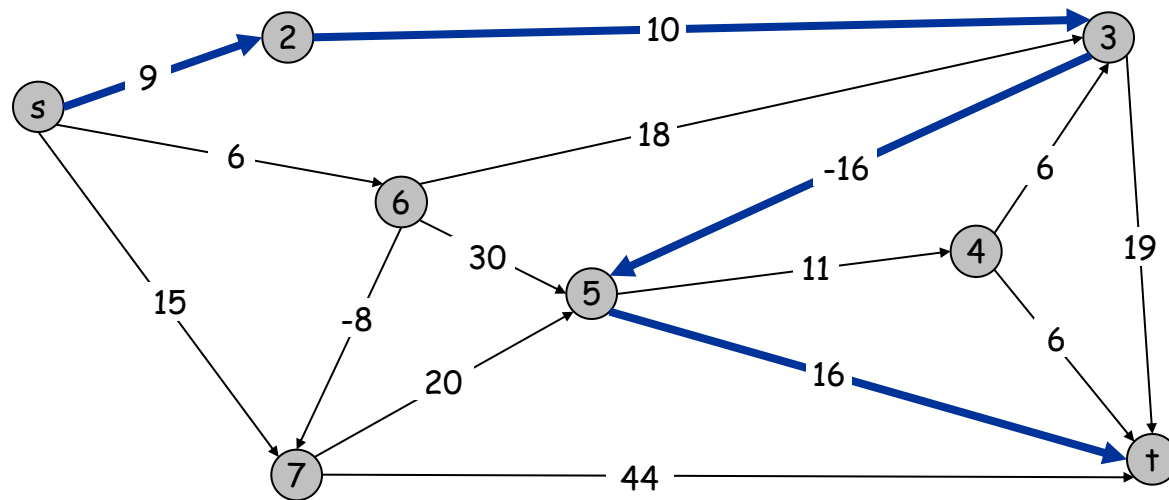
6.8 Shortest Paths

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 .

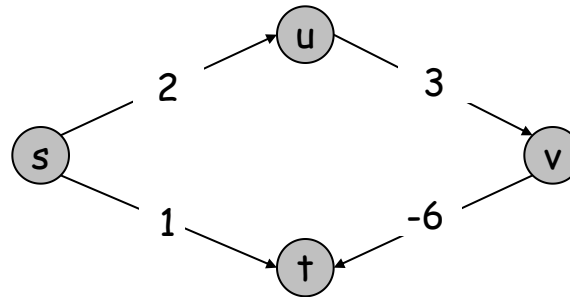
↖ allow negative weights

Ex. Nodes represent agents in a financial setting and c_{vw} is cost of transaction in which we buy from agent v and sell immediately to w .

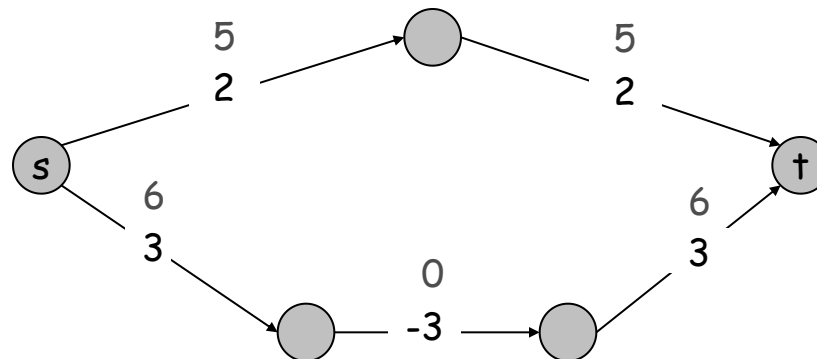


Shortest Paths: Failed Attempts

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

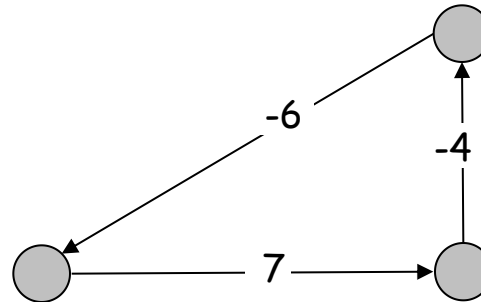


Re-weighting. Adding a constant to every edge weight can fail.

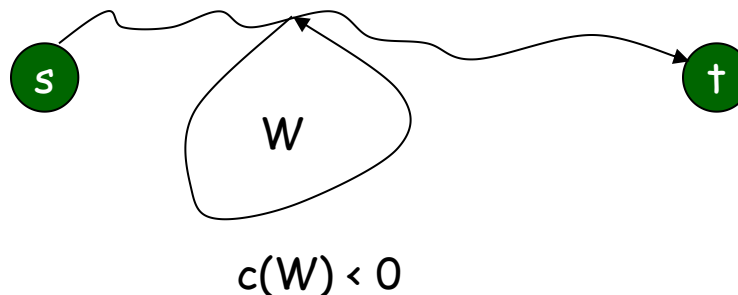


Shortest Paths: Negative Cost Cycles

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

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

- 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

$$OPT(i, v) = \begin{cases} 0 & \text{if } i = 0 \\ \min \left\{ OPT(i-1, v), \min_{(v, w) \in E} \{ OPT(i-1, w) + c_{vw} \} \right\} & \text{otherwise} \end{cases}$$

Remark. By previous observation, if no negative cycles, then $OPT(n-1, v)$ = length of shortest v - t path.

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] + cvw }  
}
```

$O(mn)$

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

$$O\left(n \sum_{v \in V} n_v\right) \quad \sum_{v \in V} n_v = m.$$

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

each edge leaves exactly one of the nodes in V

Shortest Paths: Practical Improvements

Practical improvements.

- Maintain only one array $M[v]$ = shortest v - t path that we have found so far ($i : 1, 2, \dots n-1$).
- 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.

Bellman-Ford: Efficient Implementation

```
Push-Based-Shortest-Path( $G, s, t$ ) {  
  foreach node  $v \in V$  {  
     $M[v] \leftarrow \infty$   
     $\text{successor}[v] \leftarrow \phi$   
  }  
  
   $M[t] = 0$   
  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.  
  }  
}
```

6.9 Distance Vector Protocol

Distance Vector Protocol

Communication network.

- Node \approx router.
- Edge \approx direct communication link.
- Cost of edge \approx delay on link. \leftarrow naturally nonnegative, but Bellman-Ford used anyway!

Dijkstra's algorithm. Requires global information of network.

Bellman-Ford. Uses only local knowledge of neighboring nodes.

Synchronization. We don't expect routers to run in lockstep. The order in which each `foreach` loop executes is not important. Moreover, algorithm still converges even if updates are **asynchronous**.

each time a node w experiences an update to its $M[w]$ value, it becomes "active" and eventually notifies its neighbors of the new value

not only a single destination node, but n destination nodes

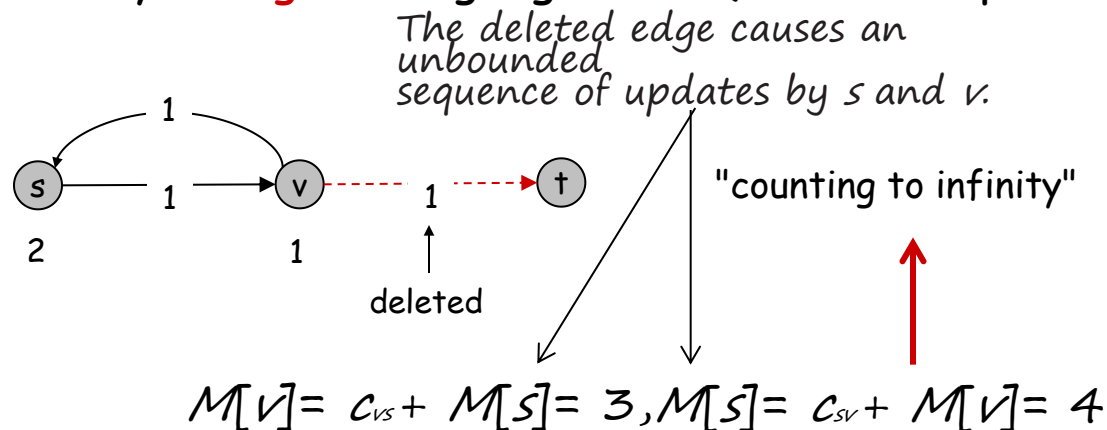
Distance **vector** protocol.

- Each router maintains a vector of shortest path lengths to every other node (distances) and the first hop on each path (directions).
- Algorithm: each router performs n separate computations, one for each potential destination node.
- "Routing by rumor."

Ex. RIP, Xerox XNS RIP, Novell's IPX RIP, Cisco's IGRP, DEC's DNA Phase IV, AppleTalk's RTMP.

Caveat. Edge costs may **change** during algorithm (or fail completely).

By using Bellman-Ford



Path Vector Protocols

Link state routing.

- Each router also stores the entire path.
- Based on Dijkstra's algorithm.
- Avoids "counting-to-infinity" problem and related difficulties.
- Requires significantly more storage.

not just the distance and first hop



Ex. Border Gateway Protocol (BGP), Open Shortest Path First (OSPF).