# Homework 1

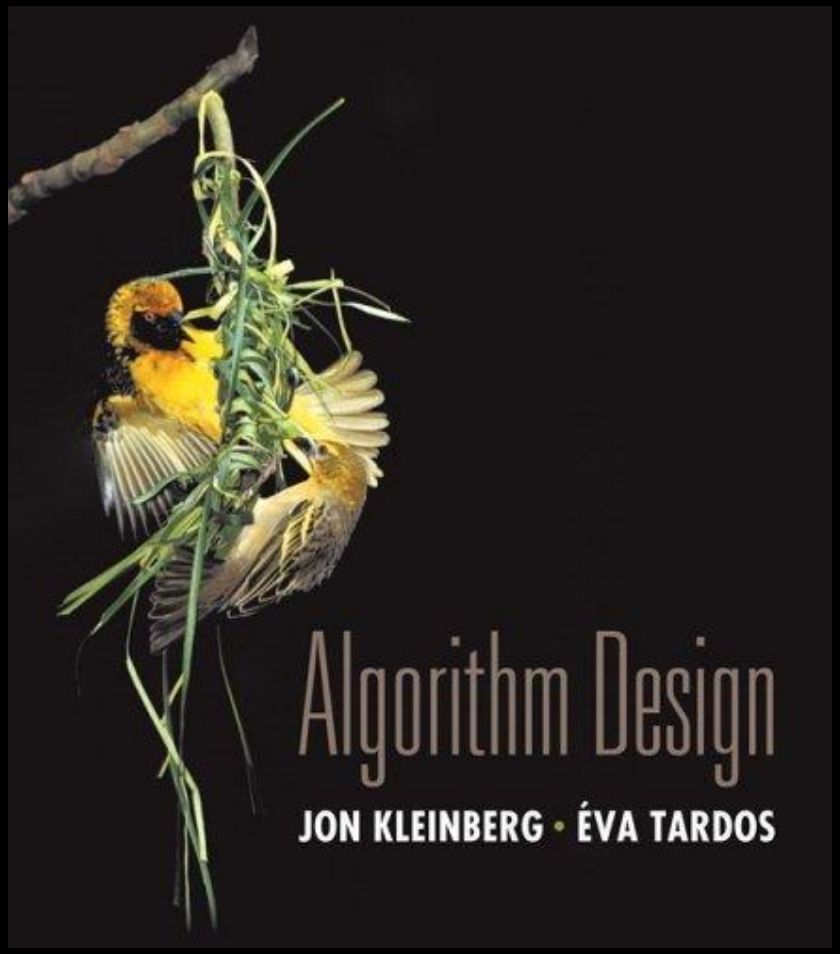
## Homework 1

注册 gradescope first

- Covers chapters 2, 4, 5, 6
- To be released today in Blackboard
- Due: 11:59pm, Mar. 16
- No late homework will be accepted!


## Submission

- Register at gradescope.com using your university email and course code D56P67. Make sure that your FULL NAME is your Chinese name.
- Save your homework solutions in a PDF or image file
- Upload your file to Gradescope and match your solution to each problem




# Chapter 6

## Dynamic Programming

# Algorithmic Paradigms

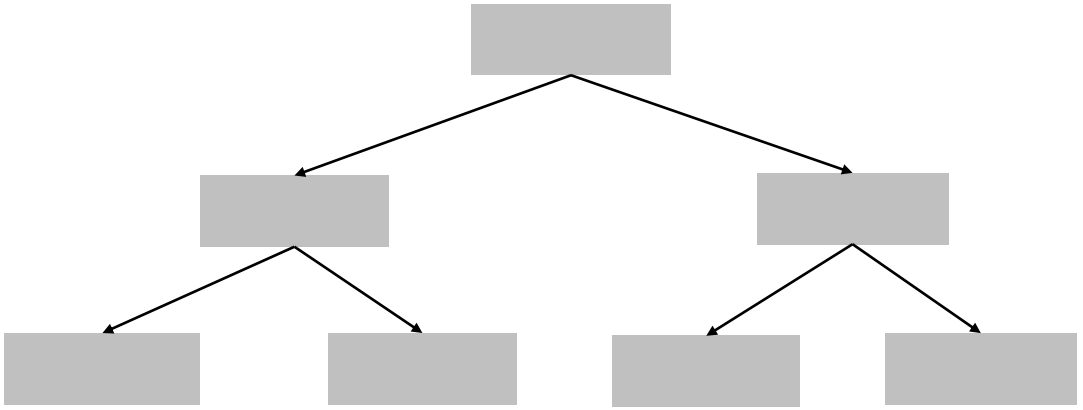
Greed. Build up a solution incrementally, myopically optimizing some local criterion.

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

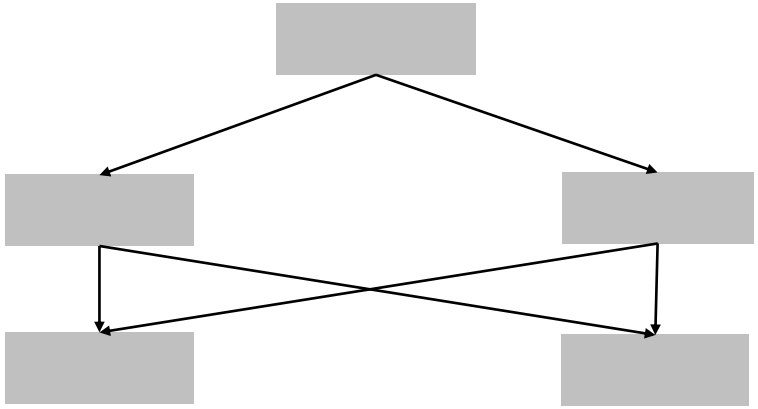
Dynamic programming. Break up a problem into a series of overlapping sub-problems, and build up solutions to larger and larger sub-problems.

# Divide-and-conquer VS. Dynamic programming

Divide-and-conquer



Dynamic programming

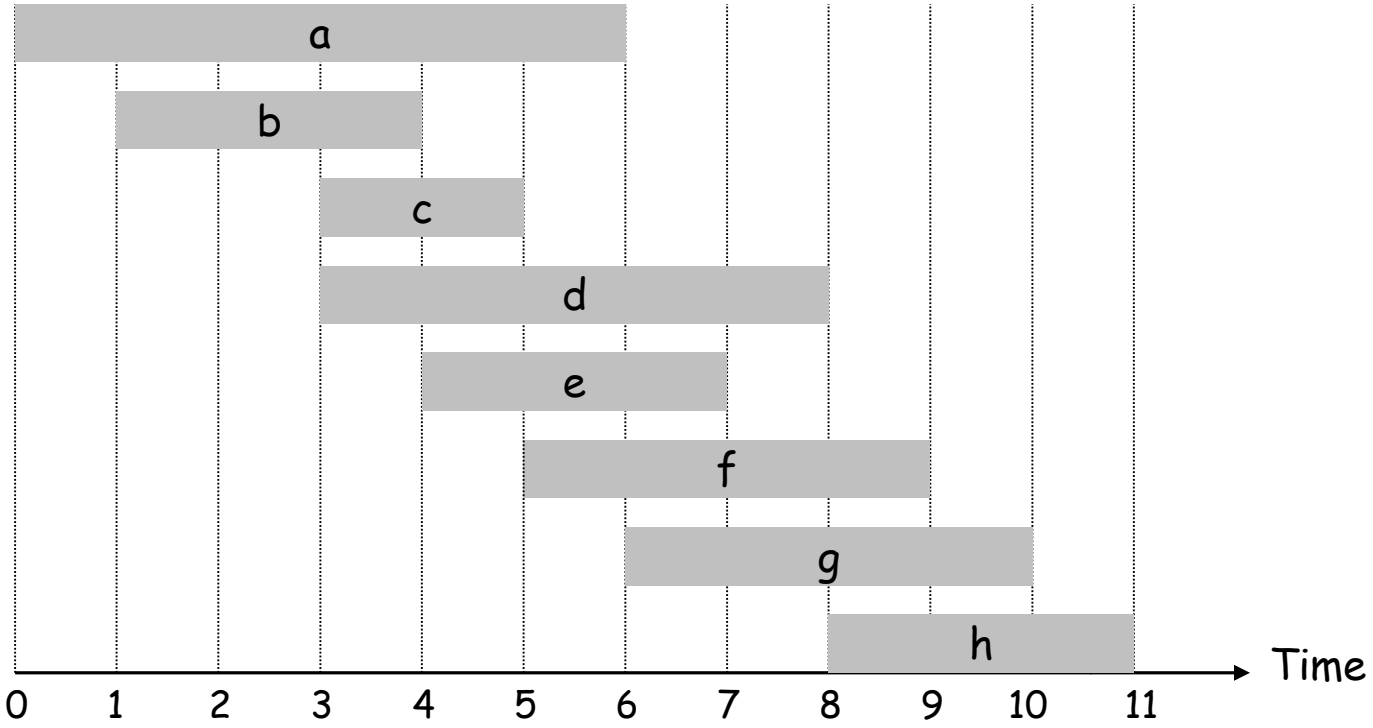


# 6.1 Weighted Interval Scheduling

# 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.
- Goal: find maximum weight subset of mutually compatible jobs.



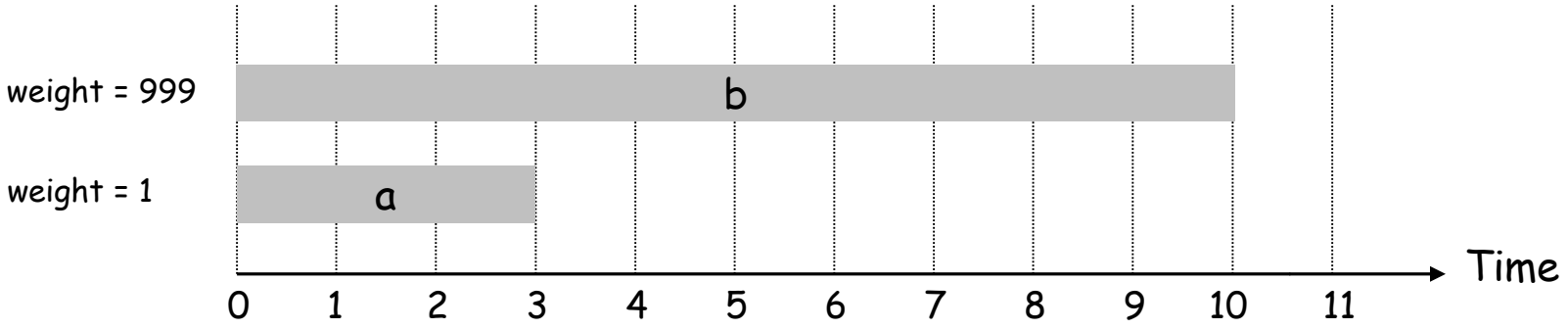
# Unweighted Interval Scheduling Review

greedy for not weight

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.



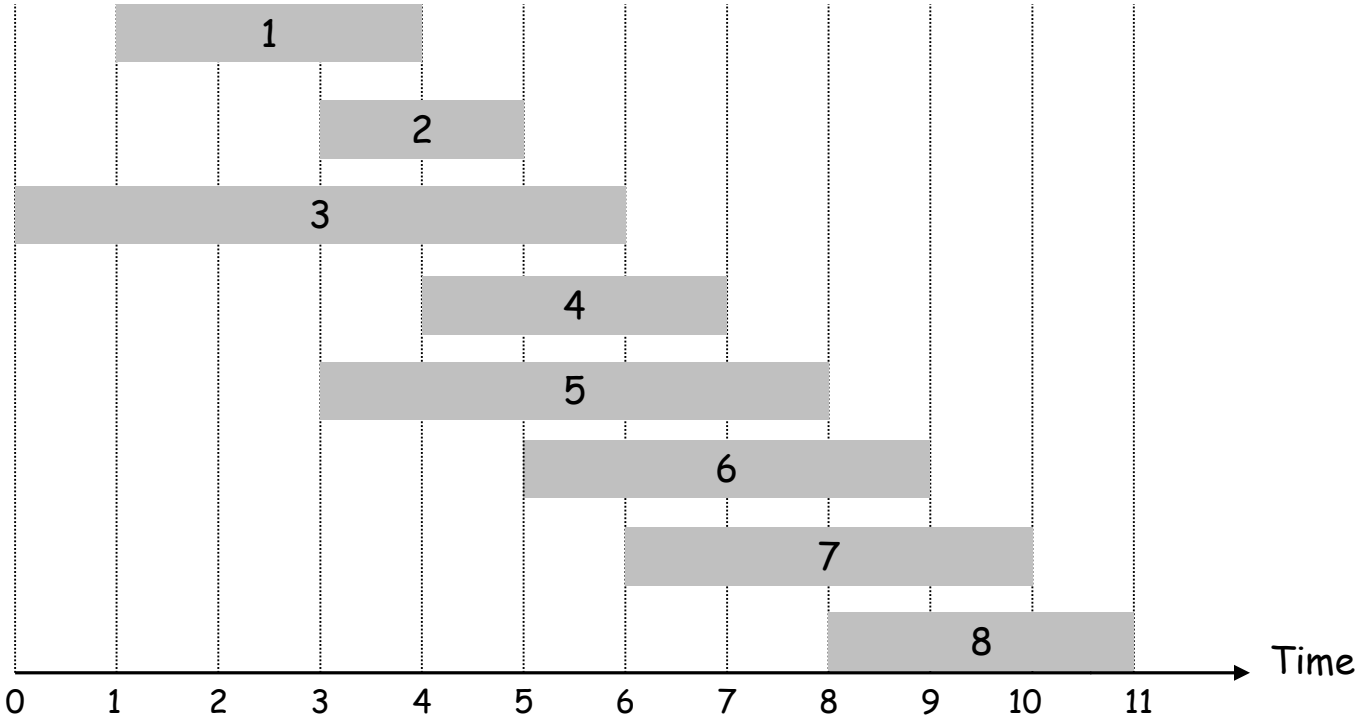
# Weighted Interval Scheduling

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

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.

p(8) 8之前与8无冲突而job中index最大的



# Dynamic Programming: Binary Choice

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

- 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)

optimal substructure

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

打掉 j

无工作可选

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

Case 1 Case 2

# Weighted Interval Scheduling: Brute Force

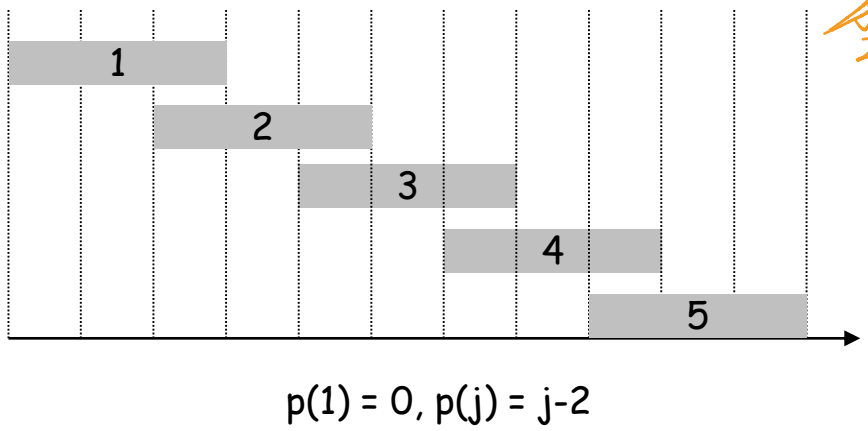
Brute force algorithm.

```
Input: n, s1,…,sn , f1,…,fn , v1,…,vn

Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.

Compute p(1), p(2), …, p(n)

Compute-Opt(j) {
   if (j = 0)
      return 0
   else
      return max(vj + Compute-Opt(p(j)), Compute-Opt(j-1))
}
```

# Weighted Interval Scheduling: Brute Force

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

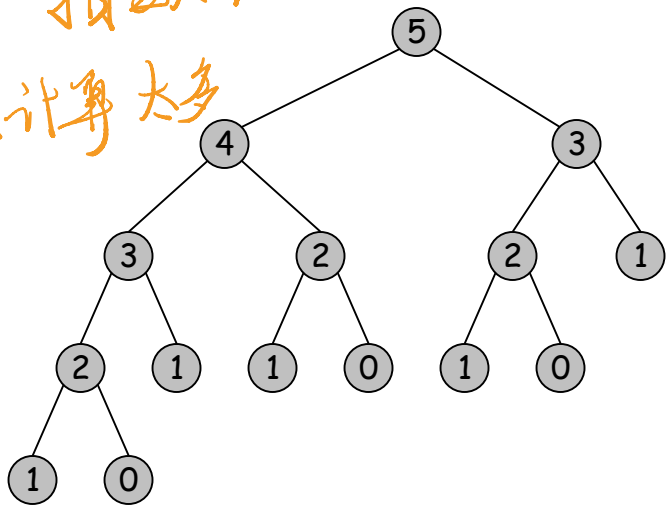
Ex. Number of recursive calls for family of "layered" instances grows like Fibonacci sequence.




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

指数级 complexity

重复计算太多



# Weighted Interval Scheduling: Memoization

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

```
Input: n, s1,…,sn , f1,…,fn , v1,…,vn

Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.
Compute p(1), p(2), …, p(n)

for j = 1 to n
   M[j] = empty   ← global array
M[0] = 0

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

↘ brute force. 求 n个 P. O(n²)
可以降

# 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 $p(\cdot)$ : $O(n)$ after sorting by start time ← how?

- `M-Compute-Opt(j)`: $O(n)$
  - Each entry `M[j]` is computed only once
  - The computation of `M[j]` invokes `M-Compute-Opt` twice

Remark. $O(n)$ if jobs are pre-sorted by start and 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.

我们 DP 以后.

后处理.

```
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)
}
```

2 cases

- # of recursive calls ≤ n ⇒ O(n).

this way. top-down.

自顶向下.

# Weighted Interval Scheduling: Bottom-Up

another way. 自底向上.

Bottom-up dynamic programming. Unwind recursion.

```
Input: n, s1,…,sn , f1,…,fn , v1,…,vn

Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.

Compute p(1), p(2), …, p(n)

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

从0开始. increase to n.

Top-down vs. bottom-up

only compute that we need

- Top-down: May skip unnecessary sub-problems
- Bottom-up: Save the overhead in recursion

缺点：递归 sometimes is expensive.

从0等到n

# 6.4 Knapsack Problem

# Knapsack Problem

classic 背包 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.

# 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.

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

W = 11

∴ 贪心法不 work. 这样就能证明 greedy 不 work ?

Greedy: ..先放最值钱的, 5→2→1. 35 not opt

- repeatedly add item with maximum value $v_i$.
- repeatedly add item with maximum weight $w_i$. 按 weight, same. 35. not opt
- repeatedly add item with maximum ratio $v_i / w_i$. 仍然选5.

Greedy not optimal!

# Dynamic Programming: False Start

Def. OPT(i) = max profit subset of items 1, …, i.

- Case 1: OPT does not select item i.
  - OPT selects best of { 1, 2, …, i-1 }

- Case 2: OPT selects item i.
  - How shall we enforce the weight limit?

Conclusion. Shall specify the remaining weight capacity in OPT

# Dynamic Programming: Adding a New Variable

2 parameters

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

少剩余容量

- 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

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

# Knapsack Problem: Bottom-Up

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

```
Input: n, w_1,…,w_N, v_1,…,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]
```

# Knapsack Algorithm

W + 1 →

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|

n + 1 ↓

| |
|---|
| $\phi$ |
| { 1 } |
| { 1, 2 } |
| { 1, 2, 3 } |
| { 1, 2, 3, 4 } |
| { 1, 2, 3, 4, 5 } |

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

W = 11

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

# Knapsack Algorithm

W + 1 →

n + 1 ↓

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\phi$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| { 1 } | | | | | | | | | | | | |
| { 1, 2 } | | | | | | | | | | | | |
| { 1, 2, 3 } | | | | | | | | | | | | |
| { 1, 2, 3, 4 } | | | | | | | | | | | | |
| { 1, 2, 3, 4, 5 } | | | | | | | | | | | | |

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

W = 11

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

# Knapsack Algorithm

W + 1 →

bottom-up. 从左到右，上到下填

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 | 34 | 40 |

Case 2

Case 2

Case 1

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

W = 11

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

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

# Knapsack Algorithm: Top-down

省掉一些多余计算

W + 1 →

n + 1 ↓

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\phi$ | | 0 | 0 | 0 | 0 | 0 | 0 | | 0 | 0 | 0 | 0 |
| { 1 } | | | 1 | 1 | 1 | 1 | 1 | | | 1 | | 1 |
| { 1, 2 } | 0 | | | | 7 | 7 | 7 | | | | | 7 |
| { 1, 2, 3 } | | | | | 7 | 18 | | | | | | 25 |
| { 1, 2, 3, 4 } | | | | | 7 | | | | | | | 40 |
| { 1, 2, 3, 4, 5 } | | | | | | | | | | | | 40 |

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

W = 11

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

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

# Knapsack Problem: Running Time

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

存储W. need $\log W$.

W 相对 input size ($\log W$) 是指数级.

- Not polynomial in input size!
- "Pseudo-polynomial."
- Decision version of Knapsack is NP-complete. [Chapter 8]

Chap 11. find 任意精度

不保证，但逼近最优解.

Knapsack approximation algorithm. There exists a polynomial algorithm that produces a feasible solution that has value within 0.01% of optimum. [Section 11.8]

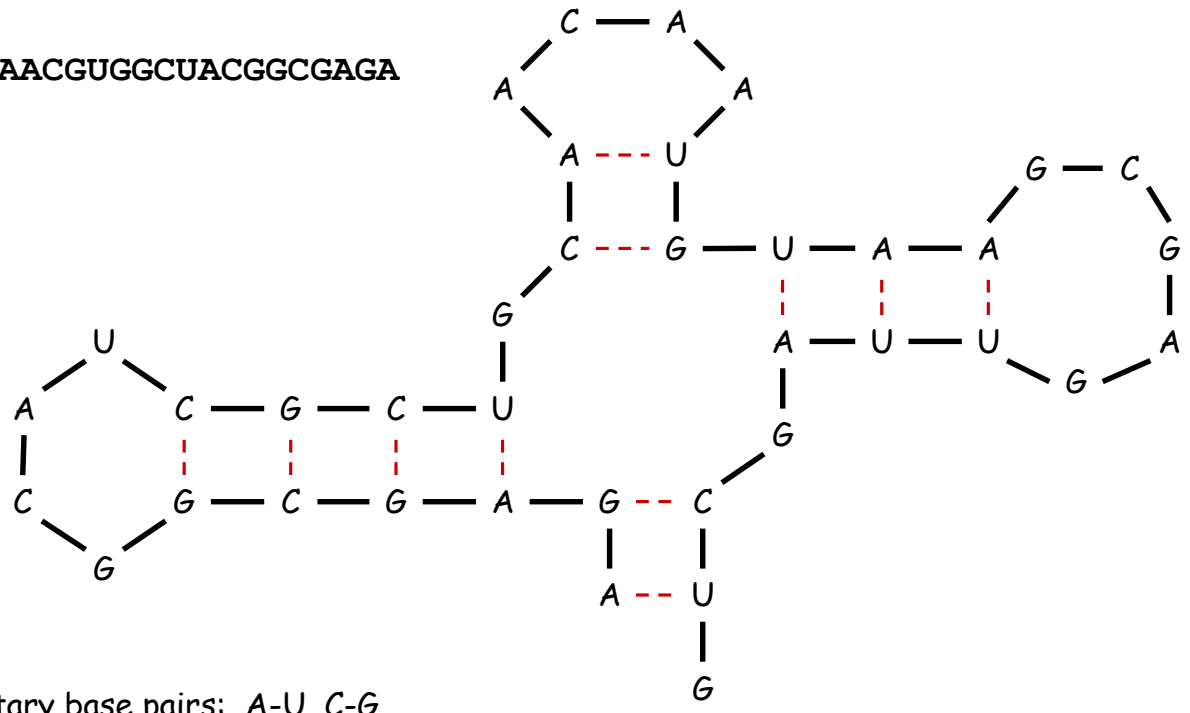
# 6.5 RNA Secondary Structure

# RNA 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:

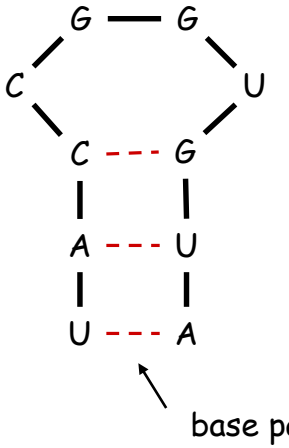
- [Watson-Crick.] S is a matching and each pair in S is a Watson-Crick complement: A-U, U-A, C-G, or G-C.
- [No sharp turns.] The ends of each pair are separated by at least 4 intervening bases. If $(b_i, b_j) \in S$, then $i < j - 4$.
- [Non-crossing.] If $(b_i, b_j)$ and $(b_k, b_l)$ are two pairs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

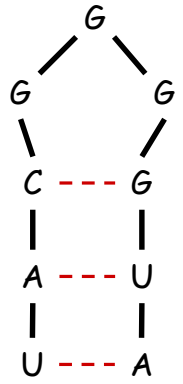
两个能对应的碱基之间至少need 4个碱基.

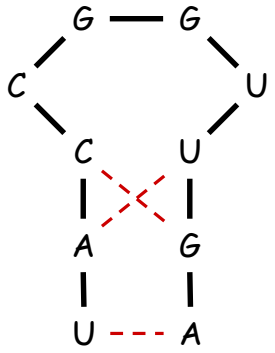
next slide ex. crossing

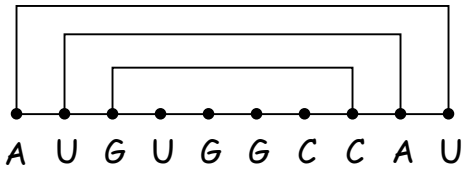
# RNA Secondary Structure: Examples

## Examples.

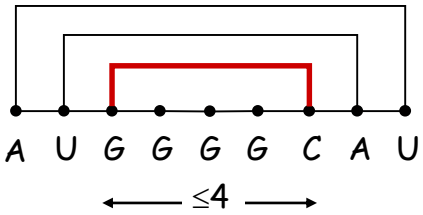


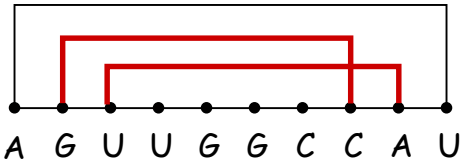




ok




sharp turn




crossing

# RNA Secondary Structure

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 $B = b_1 b_2 \ldots b_n$, find a secondary structure $S$ that maximizes the number of base pairs.

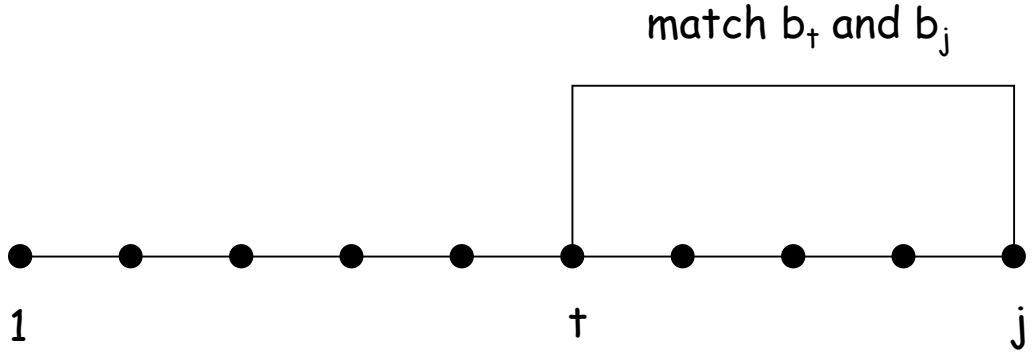
goal. max pairs.

# RNA Secondary Structure: Subproblems

OPT → 前j个碱基可max pairs

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

one index not enough




**Difficulty.** 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

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

index i — j

- If $i \geq j - 4$.
  - OPT(i, j) = 0 by no-sharp turns condition.

must crossing.

- If $i < j-4$: take max of two cases
  - Case 1. Base $b_j$ is not involved in a pair.

    OPT(i, j-1)

key thinking.
divide case, involve or not.

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

    Non-crossing constraint decouples resulting sub-problems

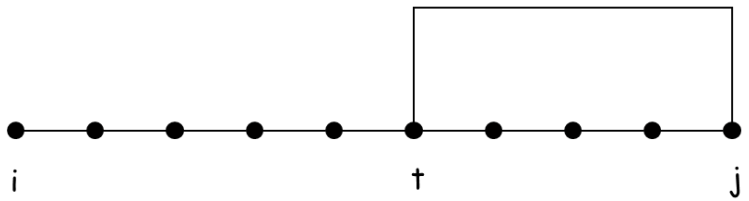
$$1 + \max_t \{ OPT(i, t-1) + OPT(t+1, j-1) \}$$

t and j, pairs

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

从 i 到 j-4. 遍历. t

t like a 分界点.



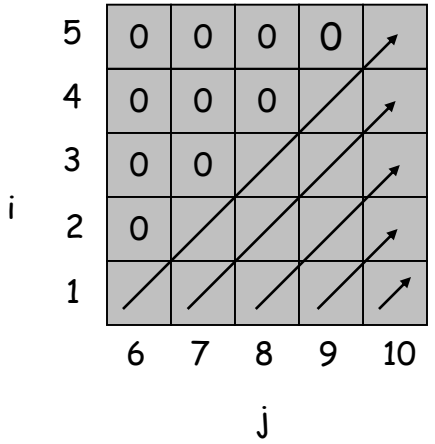
# Bottom Up Dynamic Programming Over Intervals

Q. What order to solve the sub-problems?

A. Do shortest intervals first.

why 0? ∵ $j - i \leq 4$.

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

    return M[1, n]
}
```




2. for, compute t need to traverse

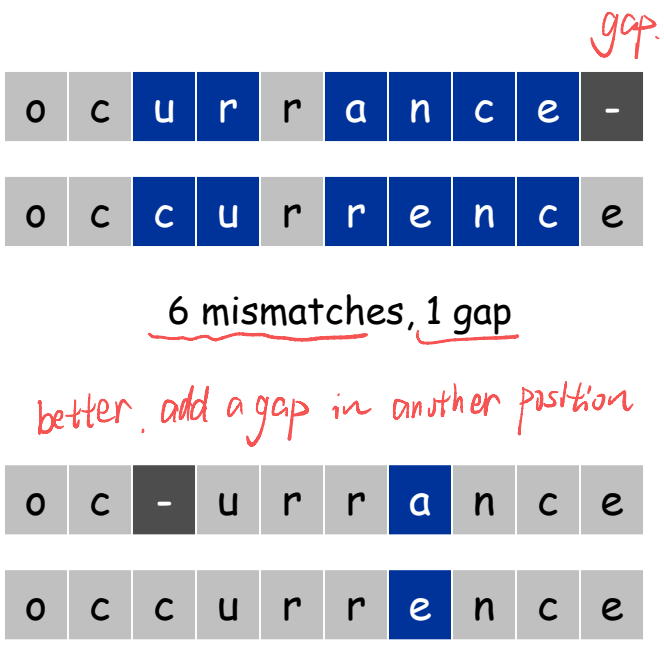
Running time. $O(n^3)$.

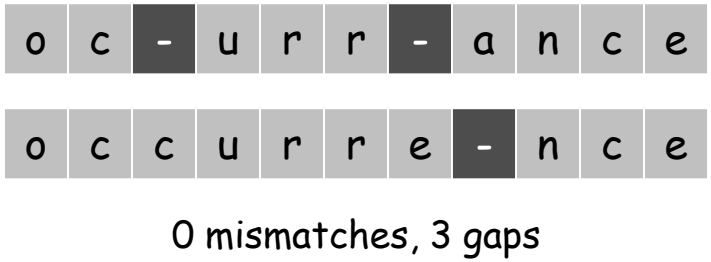
# 6.6 Sequence Alignment

# String Similarity

## How similar are two strings?

- ocurrance
- occurrence

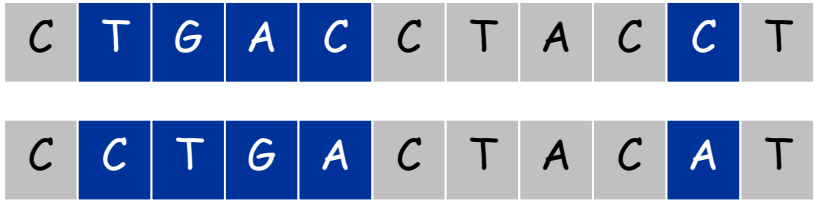


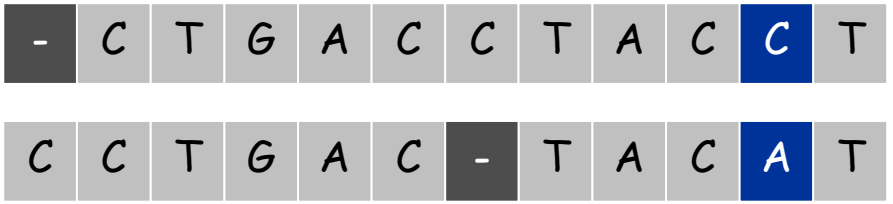
# Edit Distance

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

- Gap penalty $\delta$; mismatch penalty $\alpha_{pq}$.
- Cost = sum of gap and mismatch penalties.
- Edit distance = min cost




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




$2\delta + \alpha_{CA}$

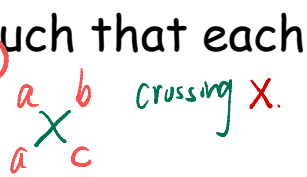
**Applications.**

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

# Sequence Alignment

**Goal:** Given two strings $X = x_1 x_2 \ldots x_m$ and $Y = y_1 y_2 \ldots y_n$ 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.




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

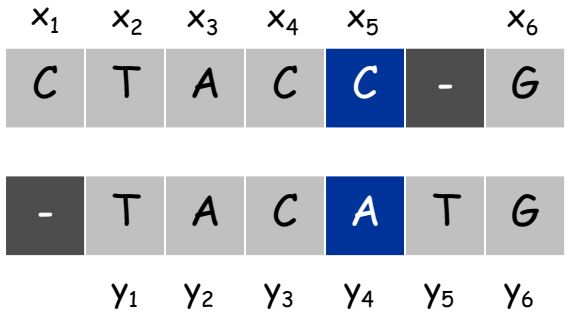
Def. of crossing ↗

**Def.** The cost of an alignment

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

**Ex:** An alignment of CTACCG vs. TACATG.

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



# Sequence Alignment: Problem Structure

first i alphabet of x.

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

first j alphabet of y

- Case 1: OPT matches $x_i$-$y_j$.
  - pay mismatch for $x_i$-$y_j$ + min cost of aligning two strings $x_1 x_2 \ldots x_{i-1}$ and $y_1 y_2 \ldots y_{j-1}$

- Case 2a: OPT leaves $x_i$ unmatched.
  - pay gap for $x_i$ and min cost of aligning $x_1 x_2 \ldots x_{i-1}$ and $y_1 y_2 \ldots y_j$

- Case 2b: OPT leaves $y_j$ unmatched.
  - pay gap for $y_j$ and min cost of aligning $x_1 x_2 \ldots x_i$ and $y_1 y_2 \ldots y_{j-1}$

cost: gap - $\delta$
miss. $\alpha$.

$$OPT(i, j) = \begin{cases} j\delta & \text{if } i = 0 \quad \text{base case} \\ \min \begin{cases} \alpha_{x_i y_j} + OPT(i-1, j-1) & \text{case 1} \\ \delta + OPT(i-1, j) & \text{case 2a} \\ \delta + OPT(i, j-1) & \text{case 2b} \end{cases} & \text{otherwise} \\ i\delta & \text{if } j = 0 \quad \text{base case} \end{cases}$$

if match. $\alpha_{x_i y_j} = 0$

# Sequence Alignment: Algorithm

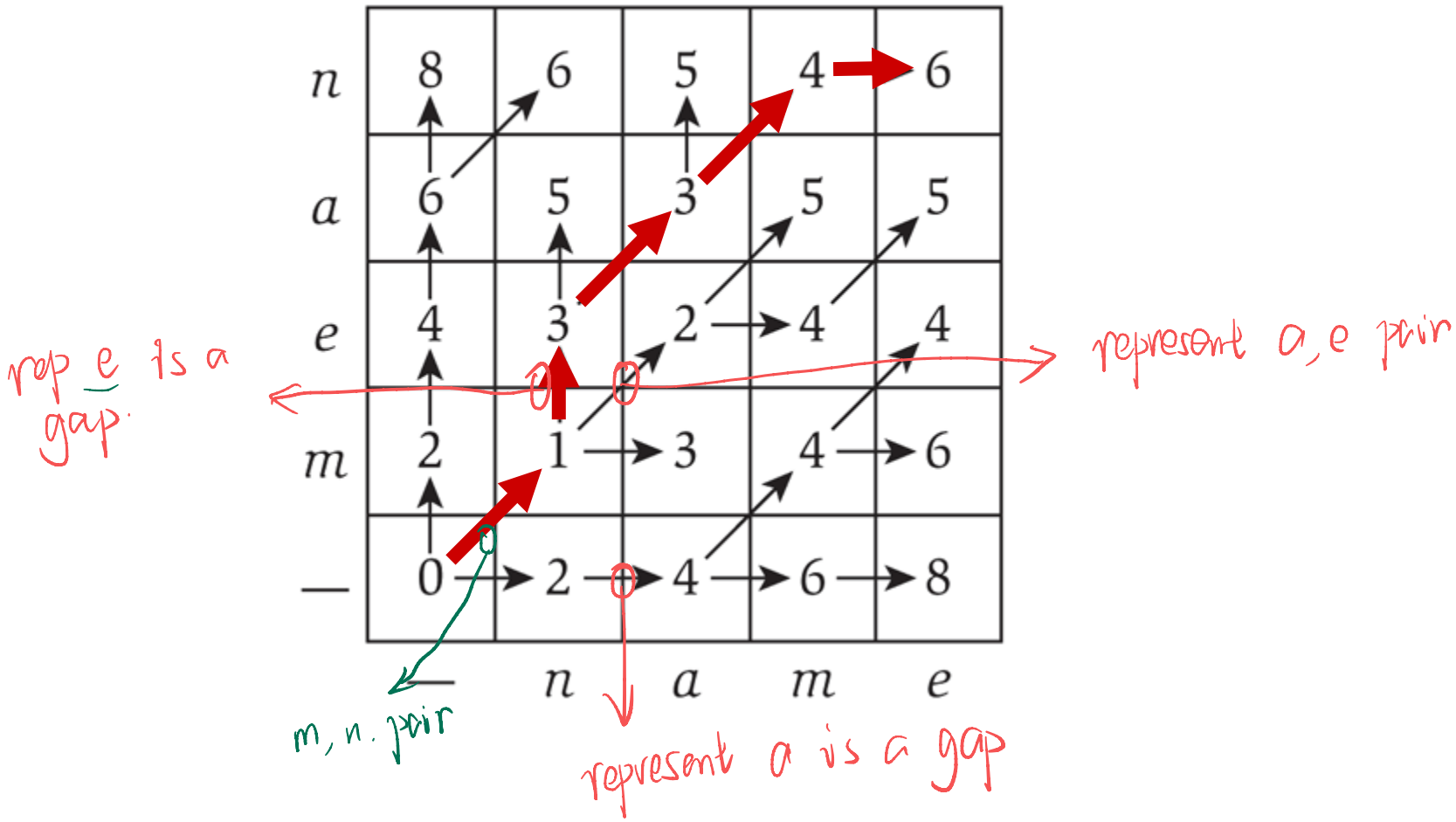
```
Sequence-Alignment(m, n, x1x2...xm, y1y2...yn, δ, α) {
   for i = 0 to m
      M[0, i] = iδ
   for j = 0 to n
      M[j, 0] = jδ

   for i = 1 to m
      for j = 1 to n
         M[i, j] = min(α[xi, yj] + M[i-1, j-1],
                       δ + M[i-1, j],
                       δ + M[i, j-1])
   return M[m, n]
}
```

base case

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

# Sequence Alignment: Example

gap penalty = 2



# Sequence Alignment: Algorithm

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

English words or sentences:

- $m, n \leq 30$. ← OK

Computational biology:

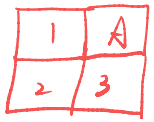
- $m = n = 100{,}000$
- 10 billions ops is OK, but 10GB array is quite large

# 6.7 Sequence Alignment in Linear Space

# Sequence Alignment: Linear Space

Q. Can we avoid using quadratic space?

fill array. 自左下向右下.




compute A. only need info. of 1.2.3

Easy. Optimal cost in O(m + n) space and O(mn) time.

- Compute OPT(i, •) from OPT(i-1, •).
- No longer a simple way to recover alignment itself.

无法通过箭头找出optimal alignment.

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

- Clever combination of divide-and-conquer and dynamic programming.

DP

# Sequence Alignment: Linear Space

## Edit distance graph.

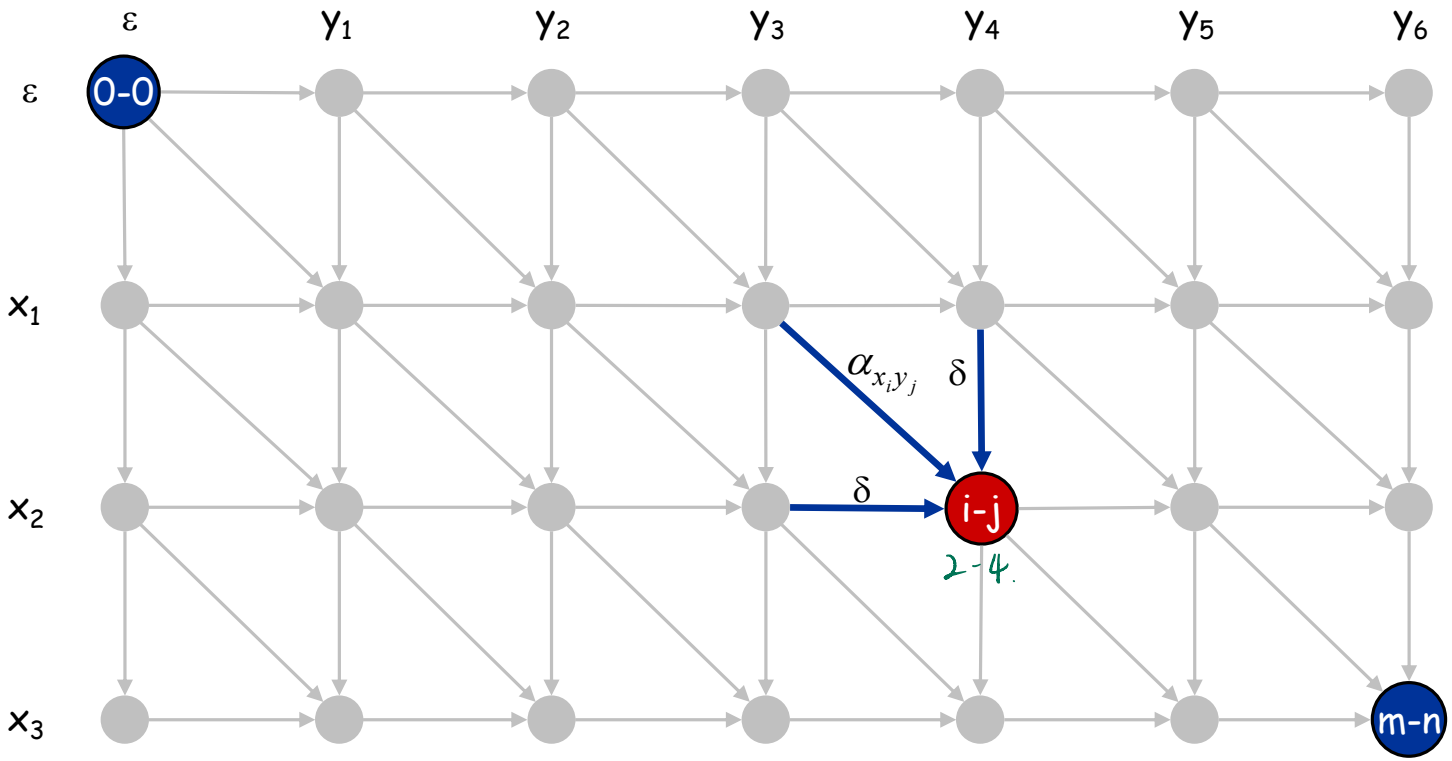
Def. of f(i, j).

- Let f(i, j) be shortest path from (0,0) to (i, j).
- Observation: f(i, j) = OPT(i, j).

path. (0,0) to (i, j). 就是 a alignment

node represent the value of OPT

sum of edge's weight on the path is the cost of the alignment



# Sequence Alignment: Linear Space

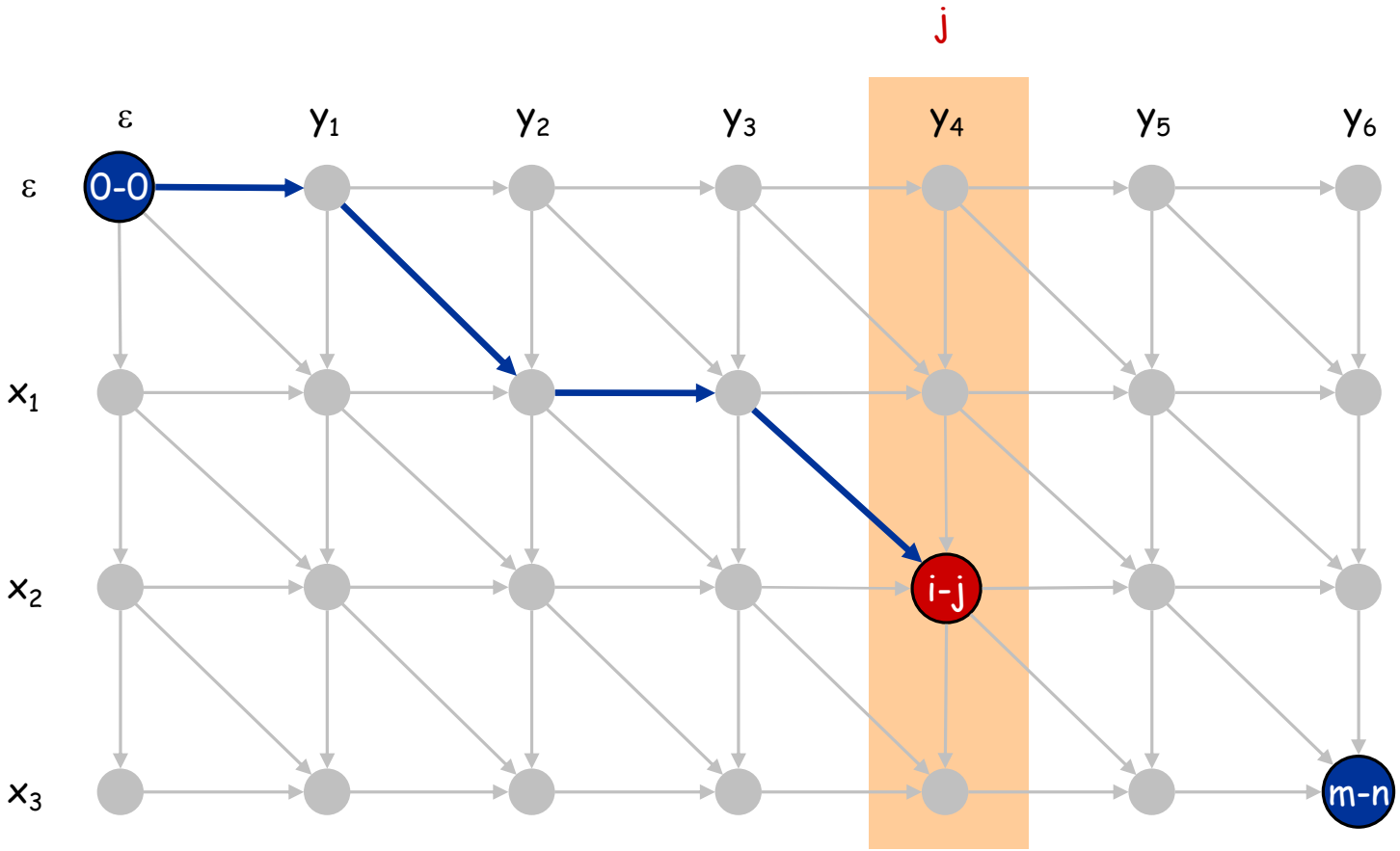
## Edit distance graph.

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

算完一列就删掉.

all value of f function in column j

只保留 info. of 2 columns



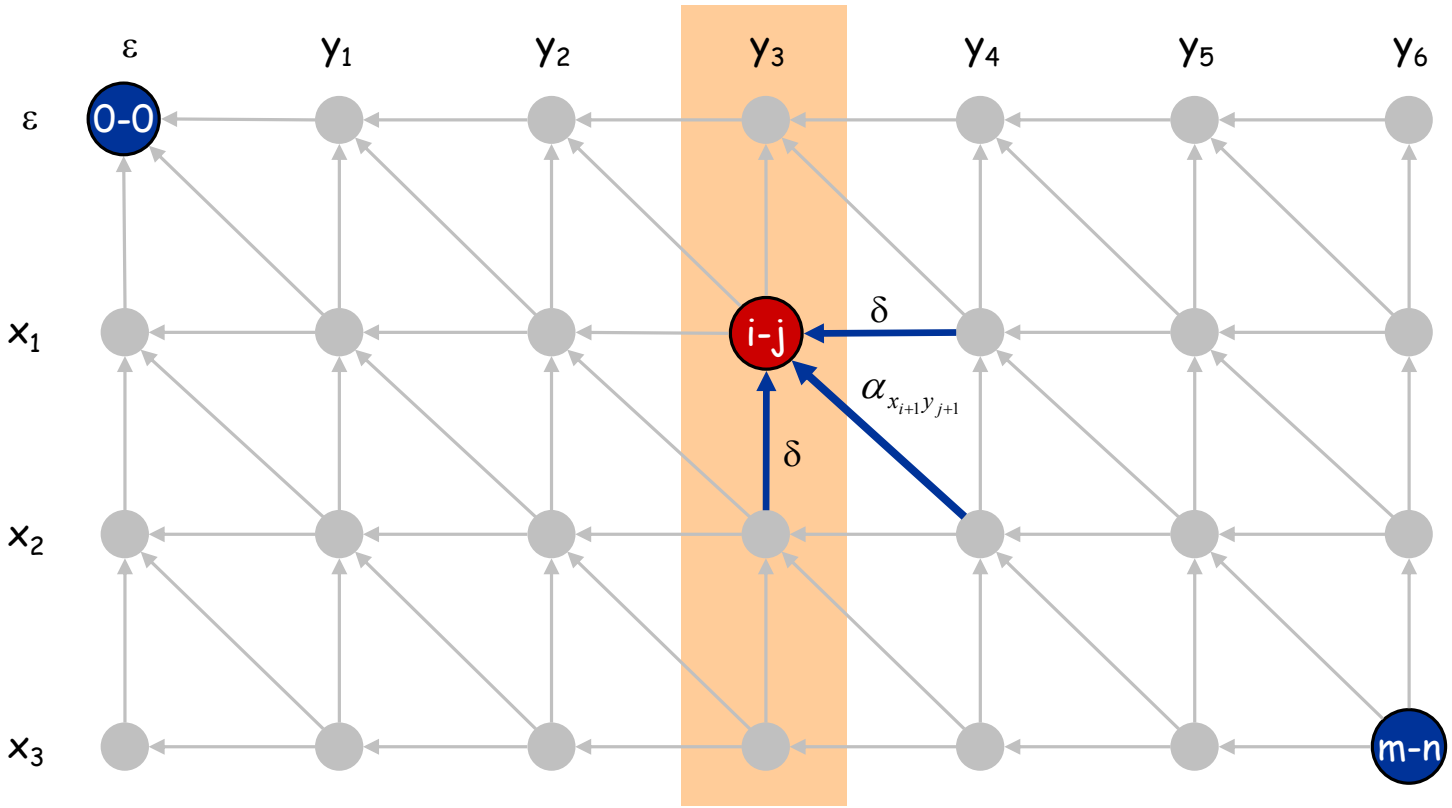
# Sequence Alignment: Linear Space

## Edit distance graph.

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

反向

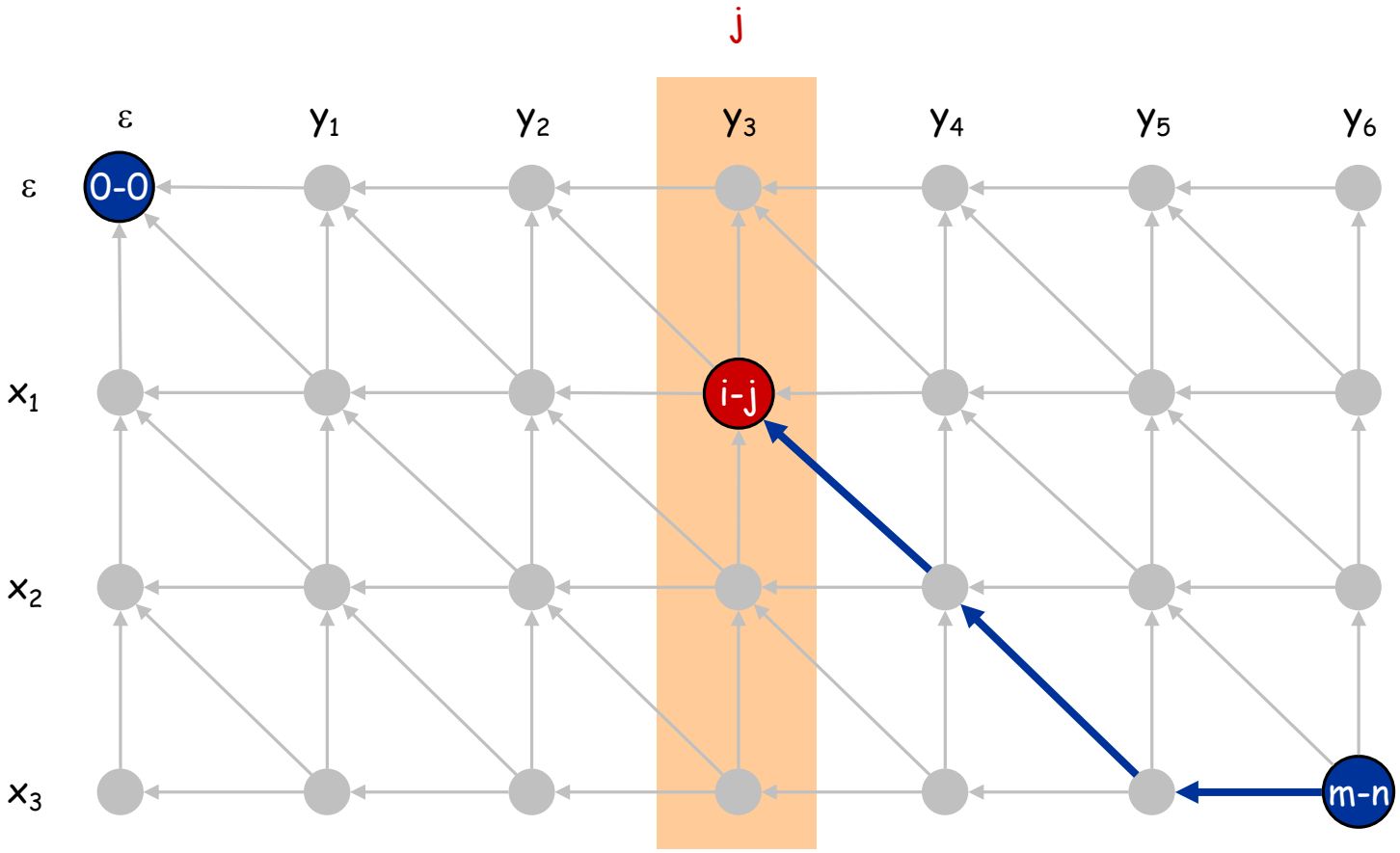
算完一列就删掉



# Sequence Alignment: Linear Space

Edit distance graph.

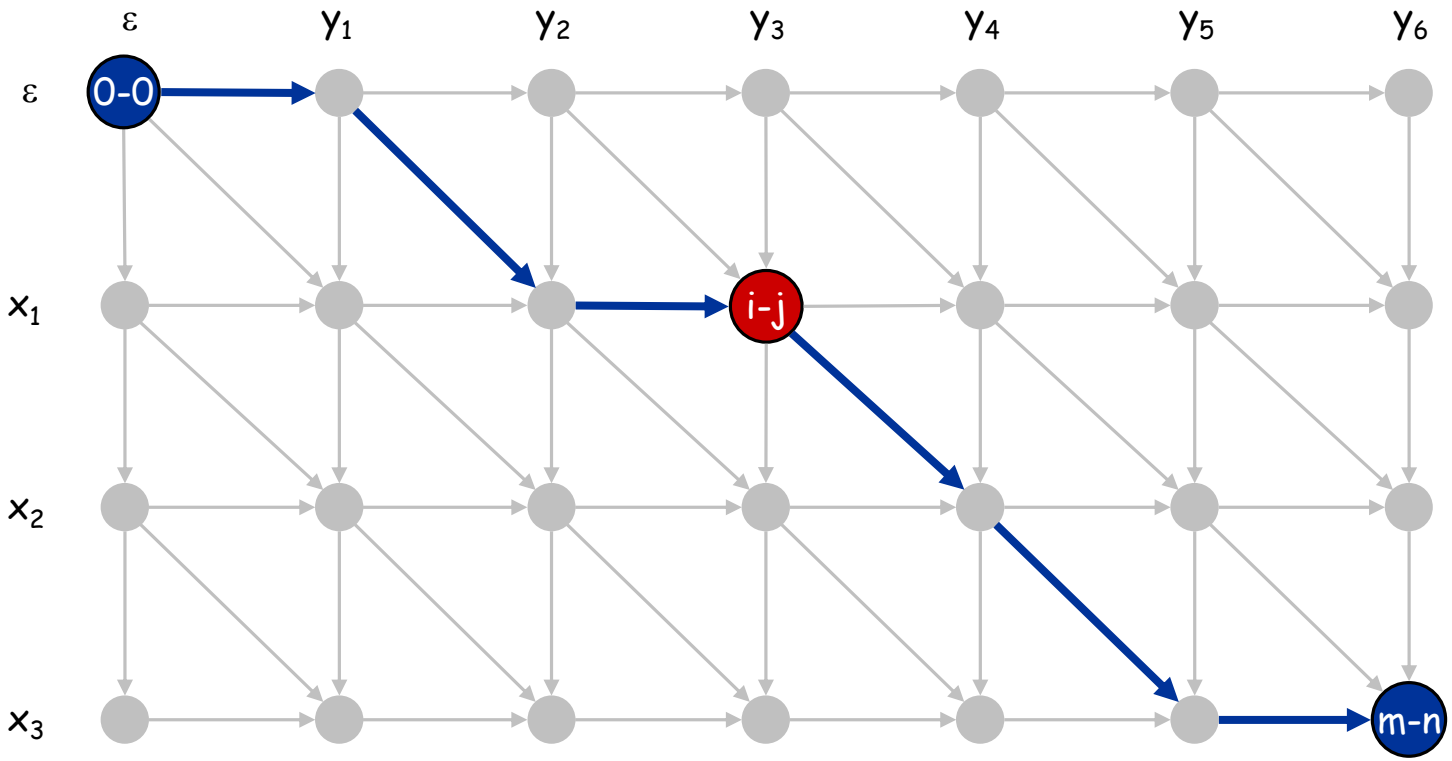
- Let g(i, j) be shortest path from (i, j) to (m, n).
- Can compute g(•, j) for any j in O(mn) time and O(m + n) space.



# Sequence Alignment: Linear Space

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

meaning: shortest path from (0,0) to (m, n), and must through (i, j).

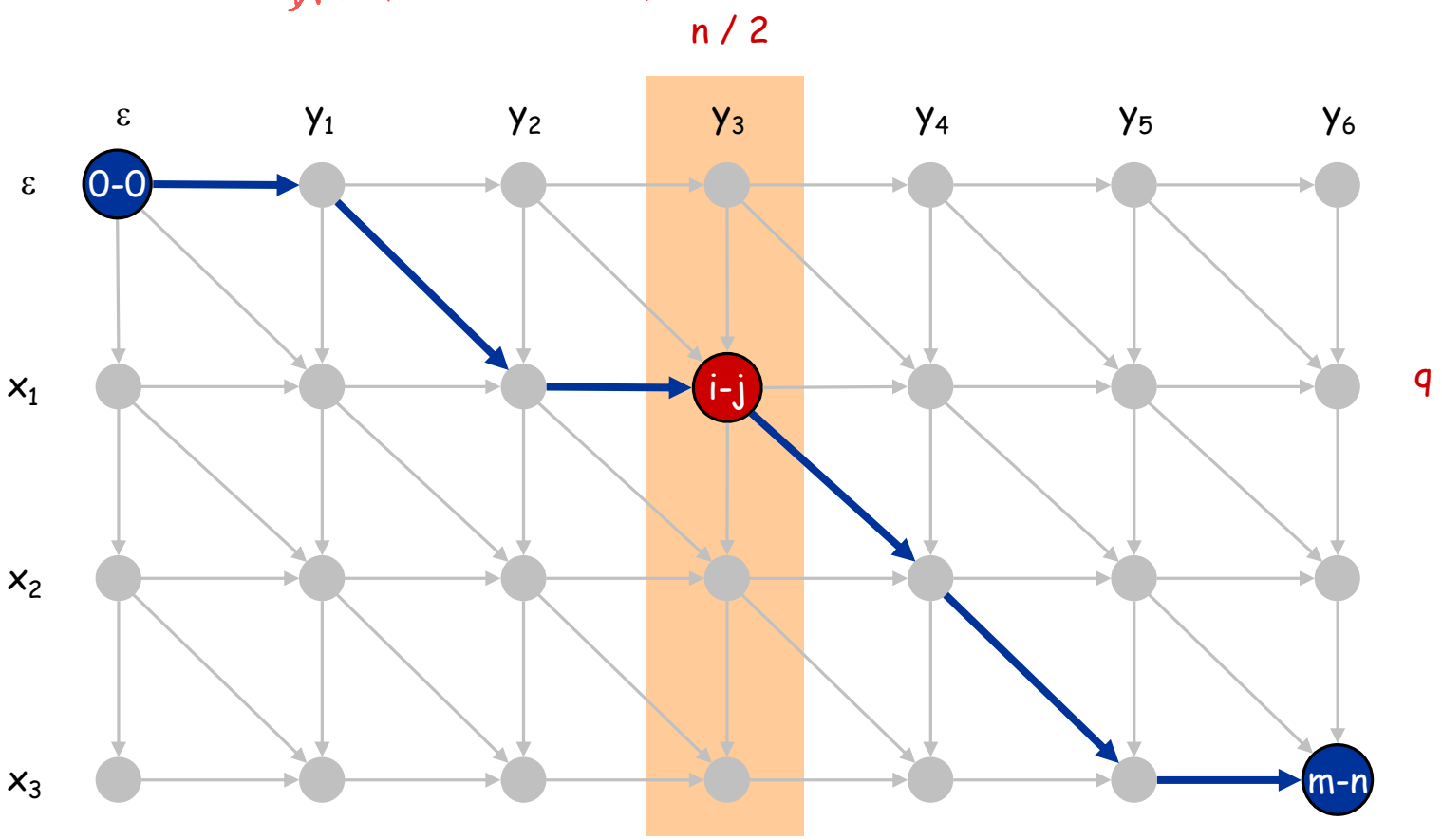


# Sequence Alignment: Linear Space

**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).

(0,0) – (m,n) 必然经过 one point in column n/2

Divide and Conger

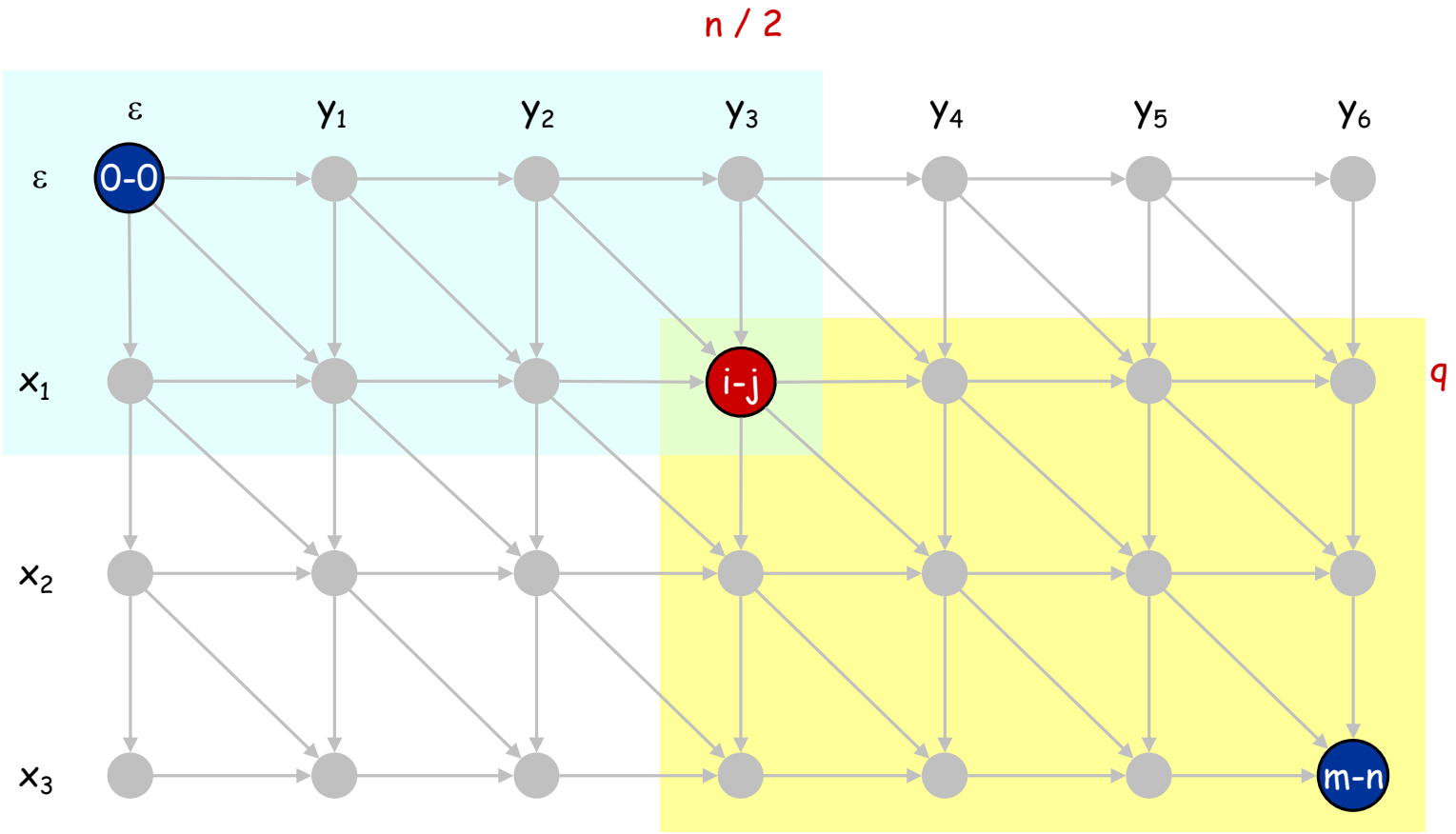


# Sequence Alignment: Linear Space

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

- Do alignment at $(x_q, y_{n/2})$.

Conquer: recursively compute optimal alignment in each piece.



# 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( ·, n/2) and g ( ·, 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:

2 columns, 2 rows. $c_1, c_2, c_3$ → 3 const. let $c = \max\{c_1, c_2, c_3\}$

$$
\begin{aligned}
T(m, 2) &\le cm \\
T(2, n) &\le cn \\
T(m, n) &\le cmn + T(q, n/2) + T(m-q, n/2)
\end{aligned}
$$

- Claim: T(m, n) ≤ 2cmn
  - Base cases: m = 2 or n = 2.
  - Inductive hypothesis: T(m', n') ≤ 2cm'n' with m'<m and n'<n

hypothesis.

$$
\begin{aligned}
T(m,n) &\le T(q,n/2) + T(m-q,n/2) + cmn \\
&\le 2cqn/2 + 2c(m-q)n/2 + cmn \quad \text{according to hypothesis} \\
&= cqn + cmn - cqn + cmn \\
&= 2cmn
\end{aligned}
$$

∴ T(m,n) ~ O(mn)

same as not use divide and conquer

# 6.8 Shortest Paths

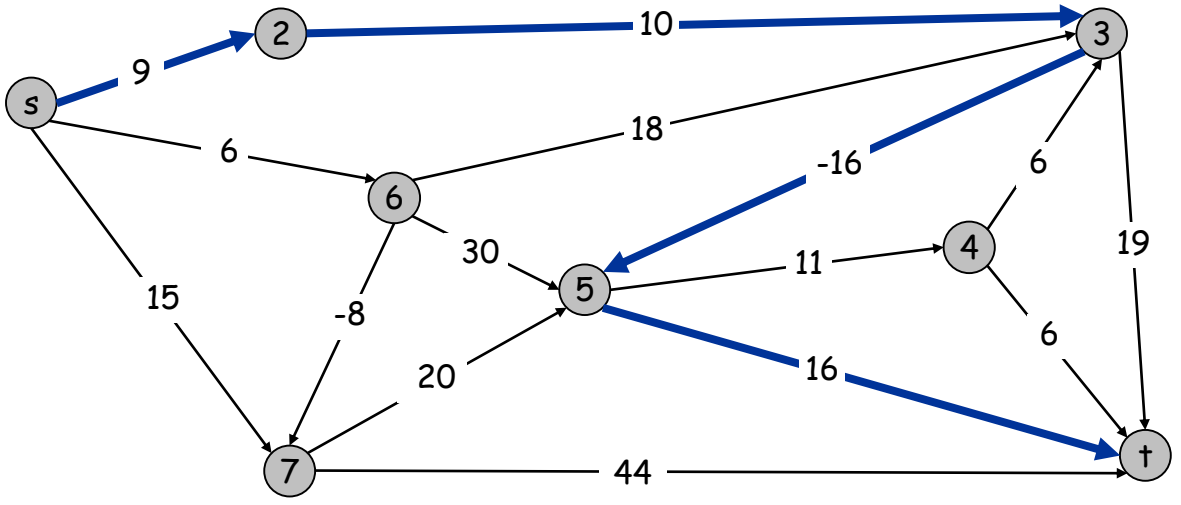
# Shortest Paths

**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

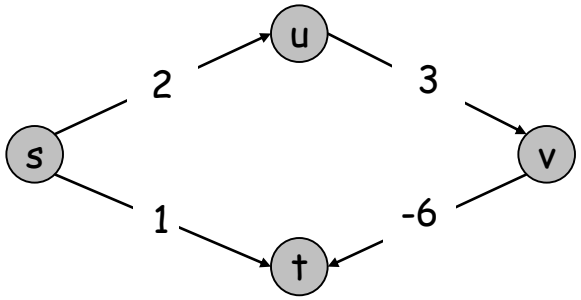
dij don't work while there are negative weights

**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.

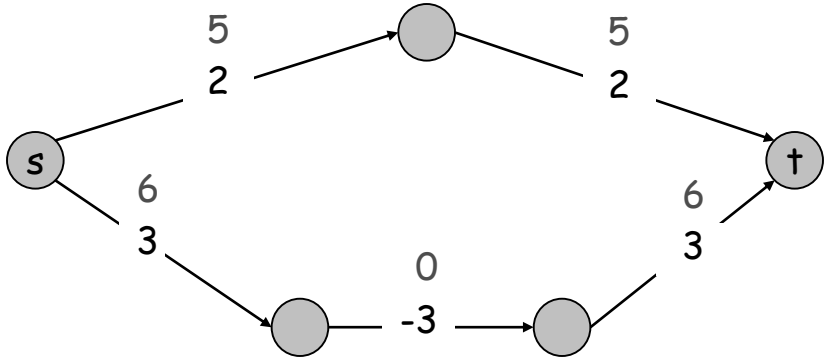


# Shortest Paths: Failed Attempts

Dijkstra. Can fail if negative edge costs.



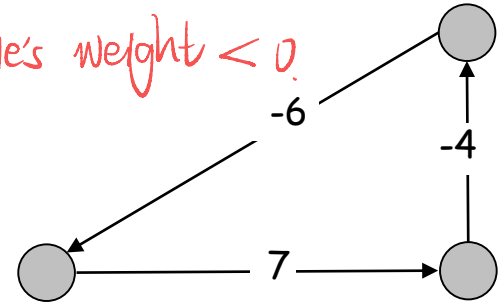

Re-weighting. Adding a constant to every edge weight can fail.



# Shortest Paths: Negative Cost Cycles

Negative cost cycle.

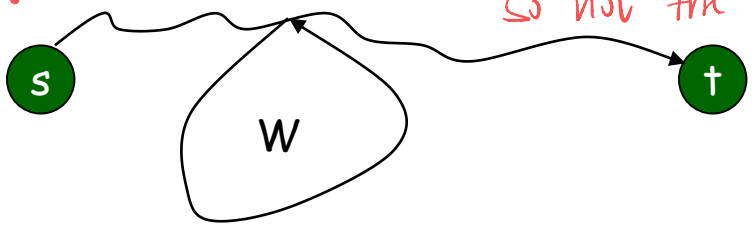
Def. of 负环: sum of edge's weight < 0.




Sum = -3 < 0

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.

through negative cycle infinitely, the weight is lower and lower, so not the shortest path




c(W) < 0

# 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 may use 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

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

**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, s, 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 }

    return M[n-1, s]
}
```

case 1.

case 2. need to traverse edge

★. need to think about it.

原点

edge. node.

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

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

# Shortest Paths: Improvements

```
Shortest-Path(G, s, 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 }

    return M[n-1, s]
}
```

Practical improvements.

- Maintain only one array M[v] = shortest v-t path that we have found so far.

目前为止我们所 v-t 的 shortest path 长度.

no longer limit edge.

# Shortest Paths: Improvements

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

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

    return M[s]
}
```

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.

M[w] changed, 才更新 M[v]

# Bellman-Ford: Efficient Implementation

```
Push-Based-Shortest-Path(G, s, t) {
    foreach node v ∈ V
        M[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
        If no M[w] value changed in iteration i, stop.
    }

    return M[s]
}
```

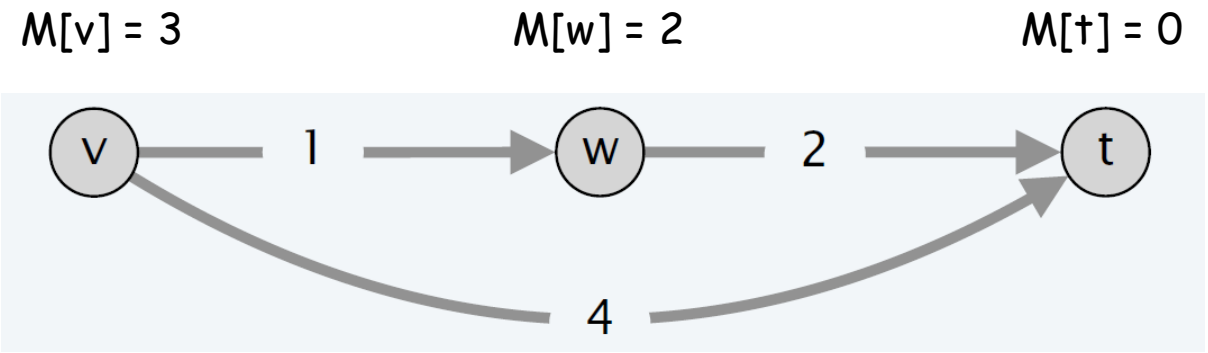
Analysis. only need to maintain M[v]

- O(n) extra space
- Time: O(mn) worst case, but substantially faster in practice.

# Bellman-Ford: Efficient Implementation

Claim. Throughout the algorithm, M[v] is length of some v-t path, and after i rounds of updates, the value M[v] is the length of shortest v-t path using $\leq$ i edges.

Counter-example:




if nodes w considered before node v,
then M[v] = 3 after 1 pass

# Bellman-Ford: Efficient Implementation

Claim. 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.

# 6.9 Distance Vector Protocol

# Distance Vector Protocol

Communication network.

- Nodes ≈ routers.
- Edges ≈ direct communication link.
- Cost of edge ≈ delay on link. ← naturally nonnegative

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.

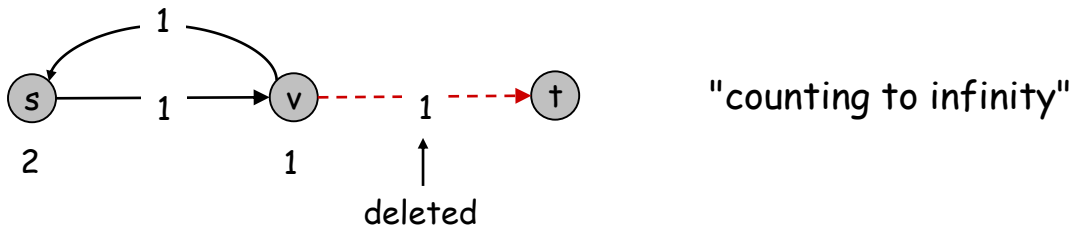
# Distance Vector Protocol

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.

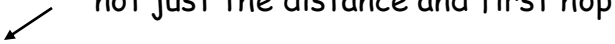
Caveat. Edge costs may change during algorithm (or fail completely).



# Path Vector Protocols

Link state routing.

- Each router also stores the entire path.
- Avoids "counting-to-infinity" problem and related difficulties.
- Requires significantly more storage.

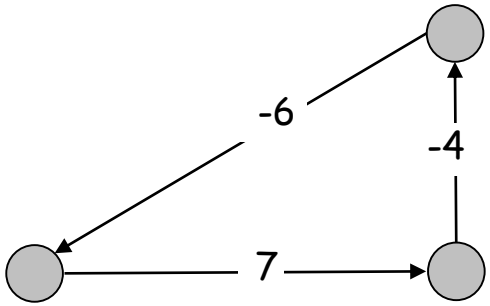



Ex. Border Gateway Protocol (BGP), Open Shortest Path First (OSPF).

# 6.10 Negative Cycles in a Graph

# Detecting Negative Cycles

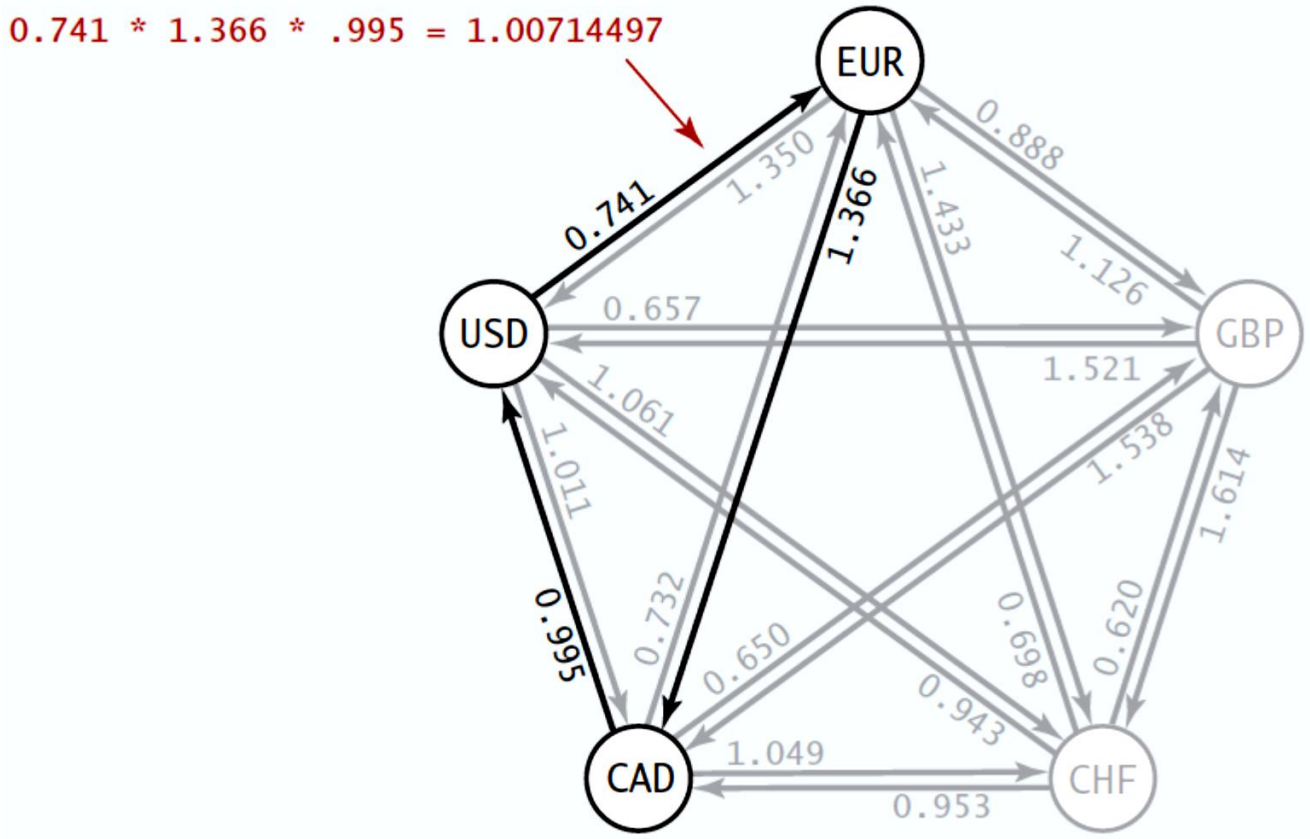
Negative cycle detection problem. Given a digraph $G = (V, E)$, with edge weights $c_{vw}$, find a negative cycle (if one exists).



# Detecting Negative Cycles: Application

Currency conversion. Given n currencies and exchange rates between pairs of currencies, is there an arbitrage opportunity?

Remark. Fastest algorithm very valuable!

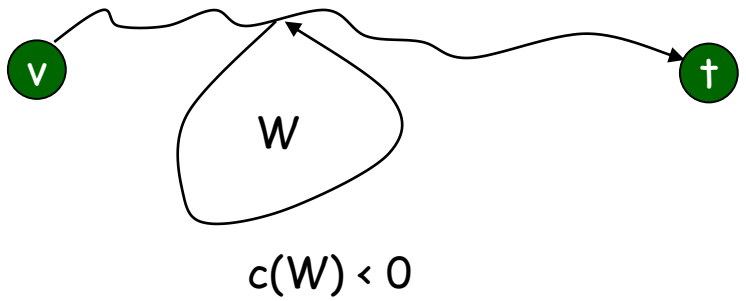


# Detecting Negative Cycles

Lemma. If OPT(n,v) = OPT(n-1,v) for all v, then there is no negative cycle with a path to t.

Pf. (by contradiction)

- OPT(n,v) = OPT(n-1,v) $\Rightarrow$ OPT(i,v) = OPT(n-1,v) for $i \geq n$
- But negative cycle in a path implies that OPT(i,v) always decreases as i increases

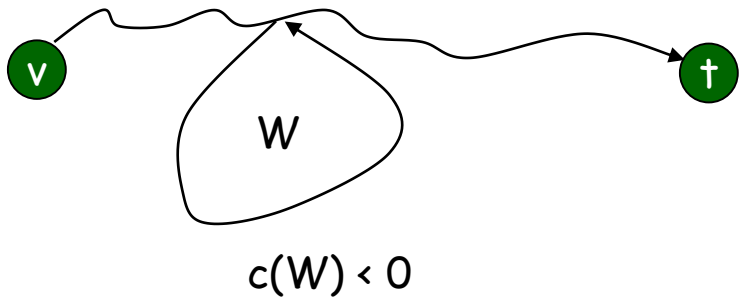


# Detecting Negative Cycles

Lemma. If OPT(n,v) < OPT(n-1,v) for some node v, then (any) shortest path from v to t contains a cycle W. Moreover W has negative cost.

Pf. (by contradiction)

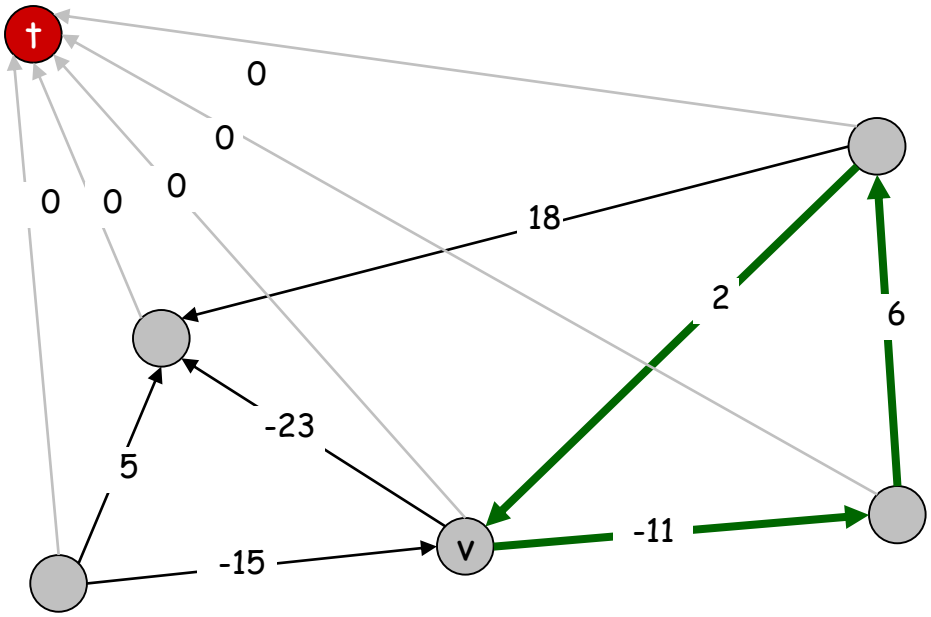
- Since OPT(n,v) < OPT(n-1,v), we know the shortest v-t path P has exactly n edges.
- By pigeonhole principle, P must contain a directed cycle W.
- Deleting W yields a v-t path with < n edges $\Rightarrow$ W has negative cost.



# Detecting Negative Cycles

Theorem. Can detect negative cost cycle in O(mn) time.

- Add new node t and connect all nodes to t with 0-cost edge.
- Check if OPT(n, v) = OPT(n-1, v) for all nodes v.
  - if yes, then no negative cycles
  - if no, then extract cycle from shortest path from v to t



# Dynamic Programming: Chapter Summary

# Dynamic Programming

## Basic idea

- Polynomial number of sub-problems with a natural ordering from smallest to largest.
- Optimal solution to a sub-problem can be constructed from optimal solutions of smaller sub-problems.
- Sub-problems are overlapping!

## Guideline

- Define the sub-problems
  - OPT(…)
- Write down the recursive formulas
  - Ex: OPT(i) = max(f(OPT(j)), g(OPT(k)), …), j,k < i
- Compute the formulas either bottom-up or top-down

# Dynamic Programming

## Algorithms

- Weighted interval scheduling
  - 1D array; binary choice
- Knapsack
  - 2D array; adding a new variable (weight limit)
- RNA secondary structure
  - 2D array: intervals
- Sequence Alignment
  - 2D array: prefix alignment
- Sequence Alignment in Linear Space
  - Combination of divide-and-conquer and dynamic programming
- Shortest path with negative edges
  - (Bellman-Ford) 2D array: shortest path with edge number $\leq i$
- Distance Vector Protocol
- Negative Cycle Detection