# CS240 Algorithm Design and Analysis

## Lecture 5

## Dynamic Programming

Quan Li

Fall 2025

2025.09.25

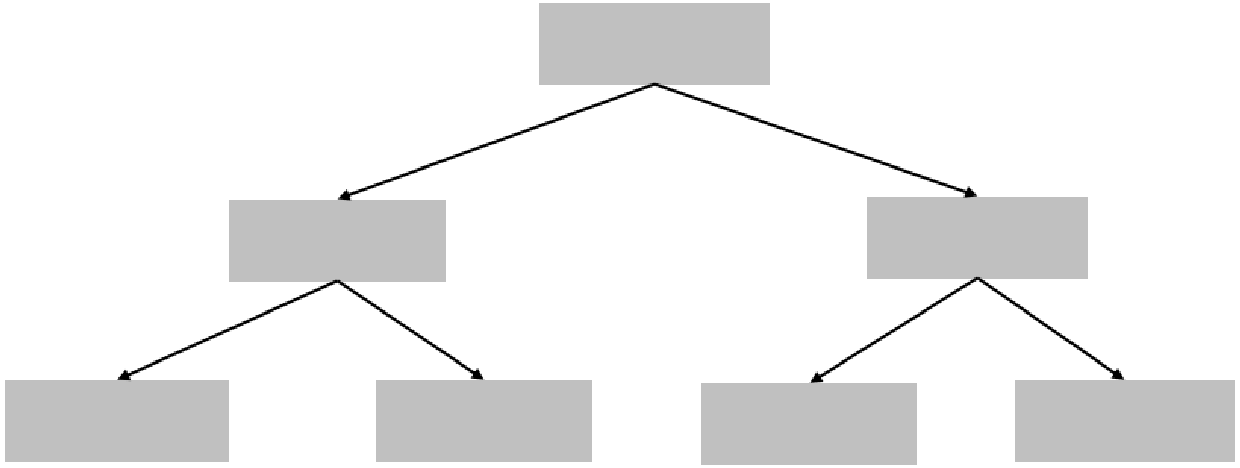
# Algorithmic Paradigms



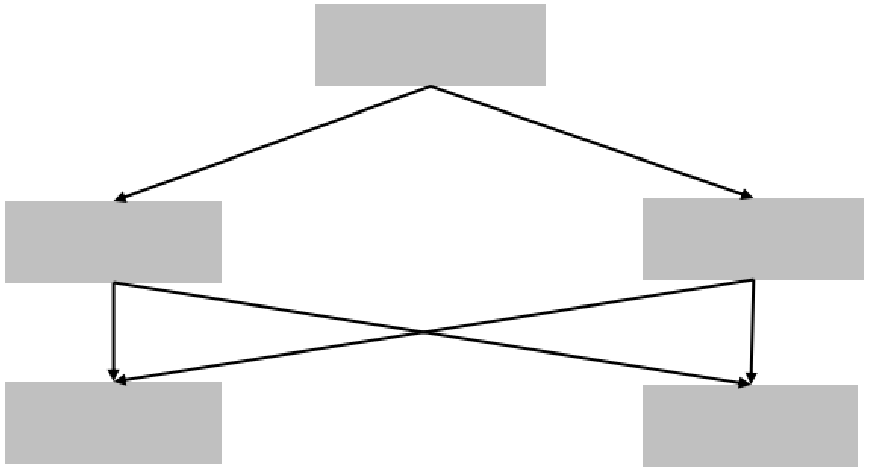
- **Greed.** Build up a solution incrementally, myopically optimizing some local criterion
- **Divide-and-conquer.** Break up a problem into a few sub-problems, solve each sub-problem independently and recursively, and combine solution to sub-problems to form solution to original problem
- **Dynamic 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

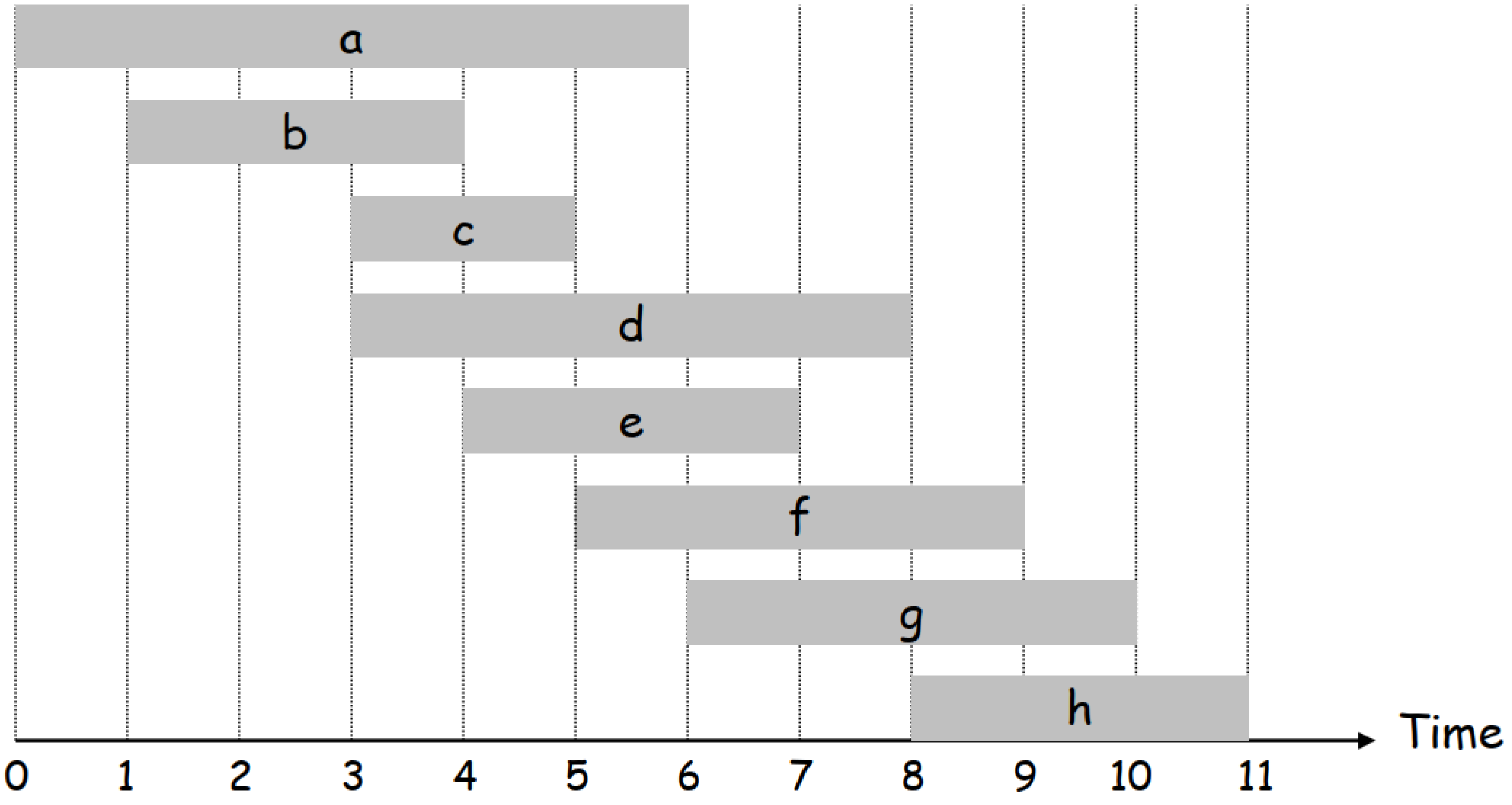


Dynamic programming



# Weighted Interval Scheduling

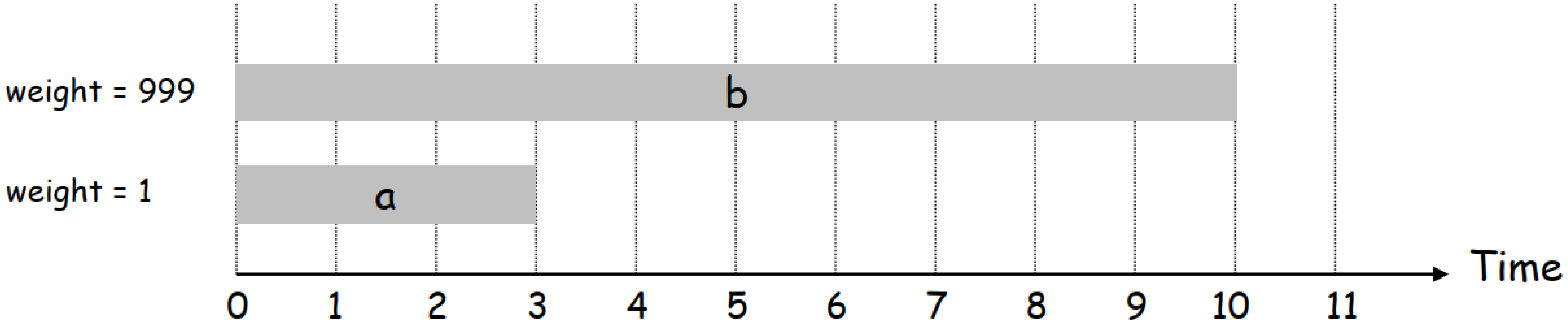
- Weighted interval scheduling problem
  - Job j starts at $s_j$, finishes at $f_j$, and has weight or value $v_j$
  - Two jobs **compatible** if they don't overlap
  - Goal: find maximum **weight** subset of mutually compatible jobs



# Unweighted Interval Scheduling Review

- **Recall**. Greedy algorithm works if all weights are 1
  - Consider jobs in ascending order of finish time
  - Add jobs to subset if it is compatible with previously chosen jobs

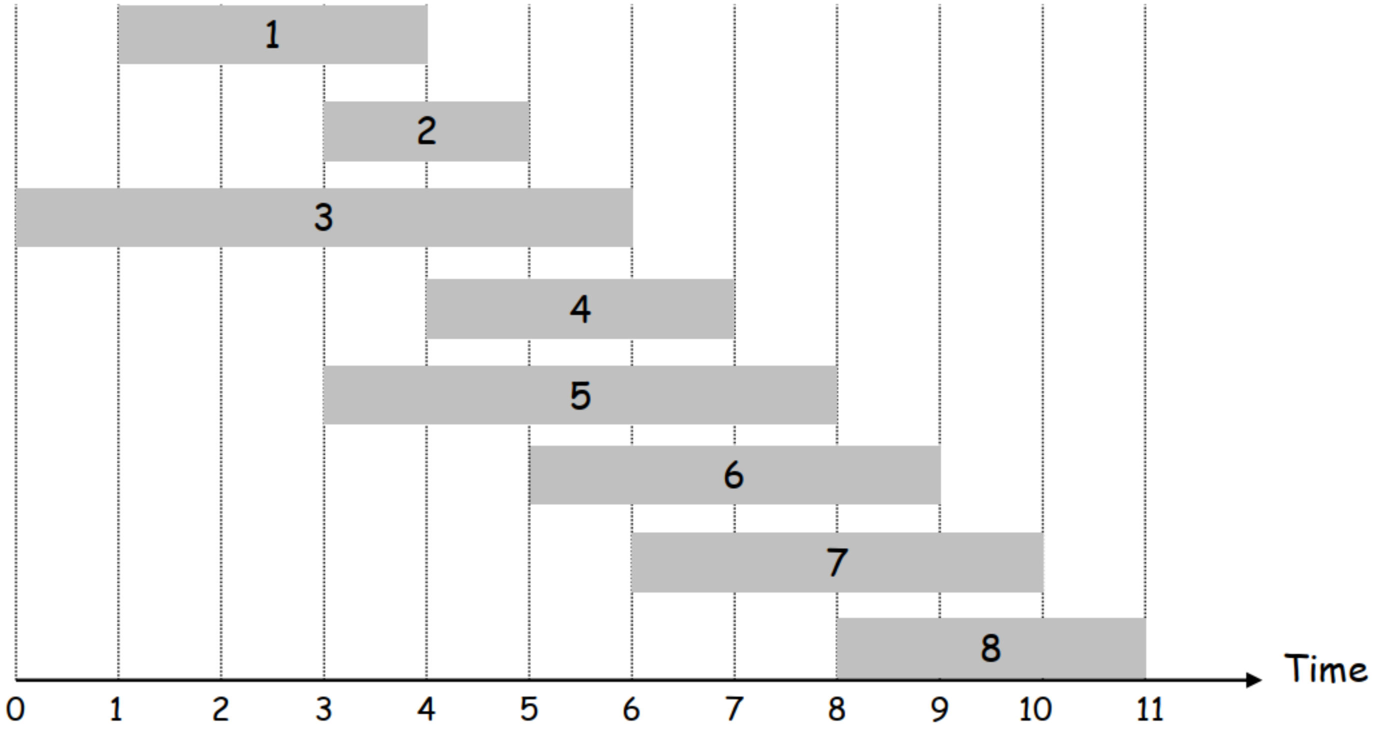
- **Observation**. Greedy algorithm can fail spectacularly if arbitrary weights are allowed



# Weighted Interval Scheduling

- **Notation.** Label jobs by finishing time: $f_1 <= f_2 <= \ldots <= f_n$
- **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



# 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
    - Cannot use incompatible jobs {p(j) + 1, p(j) + 2, …, j-1}
    - Must include optimal solution to problem consisting of remaining compatible jobs 1, 2, …, p(j)
  - Case 2: OPT does not select job j
    - Must include optimal solution to problem consisting of remaining compatible jobs 1, 2, …, j-1

Optimal substructure

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

Case 1 Case 2

立志 成才 报国 裕民

# Weighted Interval Scheduling: Brute Force

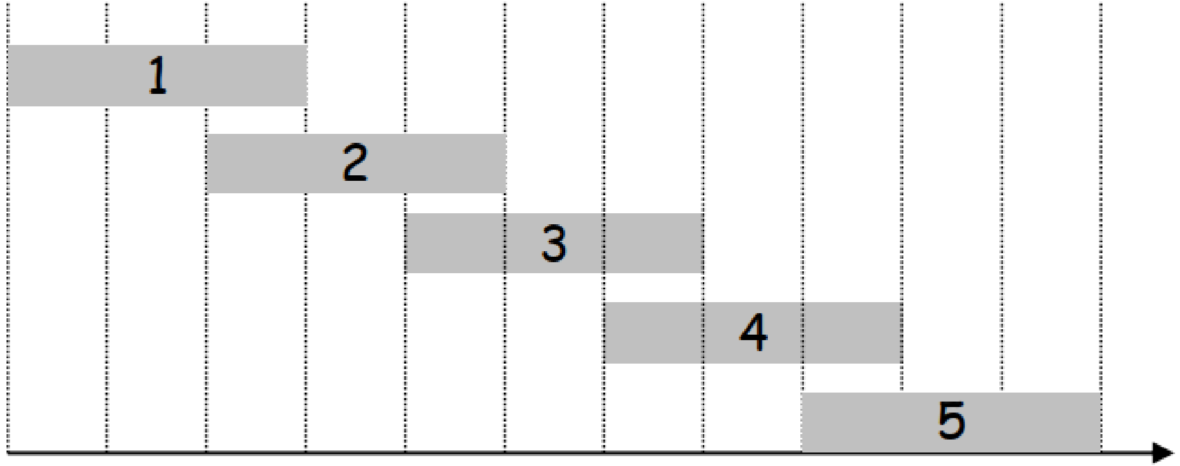
- **Brute force algorithm**

```
Input: n, s_1,…,s_n , f_1,…,f_n , v_1,…,v_n

Sort jobs by finish times so that f_1 ≤ f_2 ≤ ... ≤ f_n.

Compute p(1), p(2), …, p(n)

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

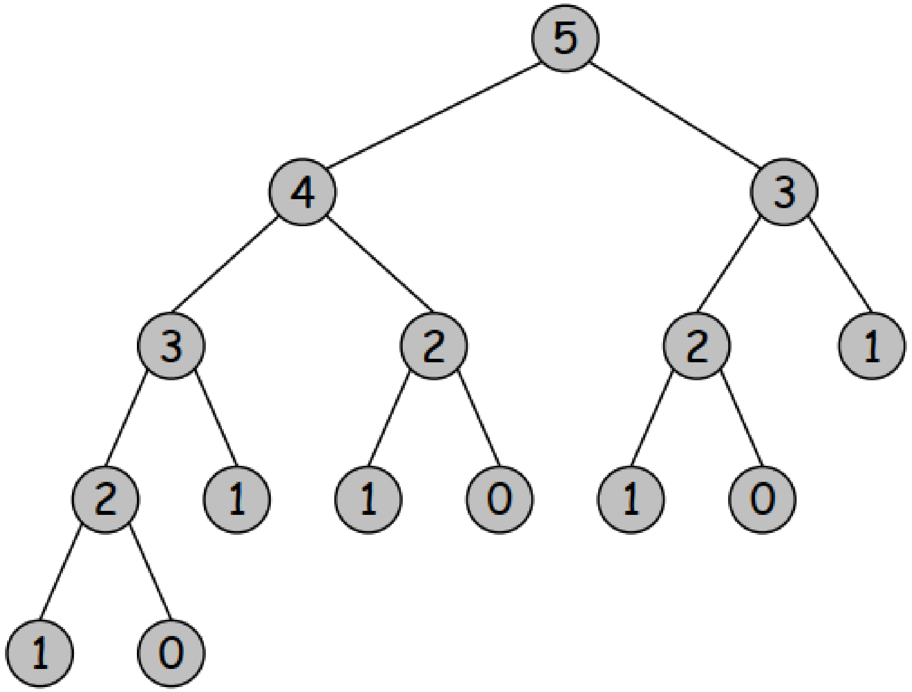
# Weighted Interval Scheduling: Brute Force

- **Observation.** Recursive algorithm fails spectacularly because of redundant sub-problems → exponential algorithms

- **Ex.** Number of recursive calls for family of “layered” instances grows like Fibonacci sequence




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



# Weighted Interval Scheduling: Memoization



- **Memoization. Store results of each sub-problem in 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]
}
```

# Weighted Interval Scheduling: Running Time

- **Claim.** Memorized version of algorithm takes O(nlogn) time
  - Sort by finish time: O(nlogn)
  - Computing p(·): 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

上海科技大学 ShanghaiTech University

- **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 (vj + M[p(j)] > M[j-1])
      print j
      Find-Solution(p(j))
   else
      Find-Solution(j-1)
}
```

# of recursive calls <= n → O(n)

立志成才报国裕民

# Weighted Interval Scheduling: Bottom-Up

- **Bottom-up dynamic programming.** Unwind recursion

```
Input: n, s_1,…,s_n , f_1,…,f_n , v_1,…,v_n

Sort jobs by finish times so that f_1 ≤ f_2 ≤ ... ≤ f_n.

Compute p(1), p(2), …, p(n)

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

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

# 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

# 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

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

- **Greedy:**
  - Repeatedly add item with maximum value $v_i$
  - Repeatedly add item with maximum weight $w_i$
  - Repeatedly add item with maximum ration $v_i/w_i$

**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



- **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, …, 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

n + 1

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

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

# 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 } | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| { 1, 2 } | 0 | 1 | 6 | 7 Case 2 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 |
| { 1, 2, 3 } | 0 | 1 | 6 | 7 | 7 | 18 | 19 | 24 | 25 | 25 Case 2 | 25 | 25 |
| { 1, 2, 3, 4 } | 0 | 1 | 6 | 7 | 7 | 18 | 22 | 24 | 28 | 29 | 29 | 40 Case 1 |
| { 1, 2, 3, 4, 5 } | 0 | 1 | 6 | 7 | 7 | 18 | 22 | 28 | 29 | 34 | 34 | 40 |

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

W = 11

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

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

# Knapsack Algorithm: Top-down

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.** $\Theta(nW)$
  - Not polynomial in input size!
  - "Pseudo-polynomial."
  - Decision version of Knapsack is NP-complete

- **Knapsack approximation algorithm.** There exists a polynomial algorithm that produces a feasible solution that has value within 0.01% of optimum.

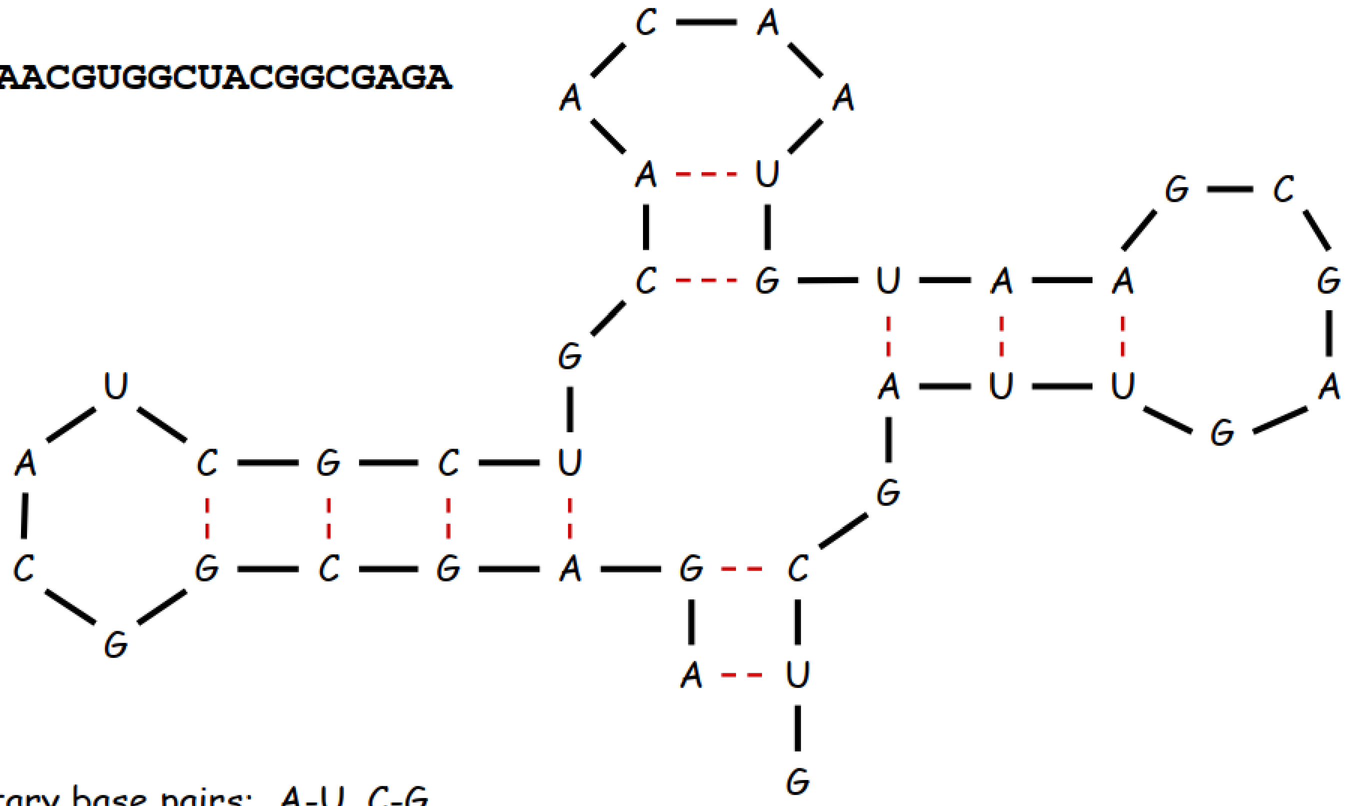
# RNA Secondary Structure

# 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

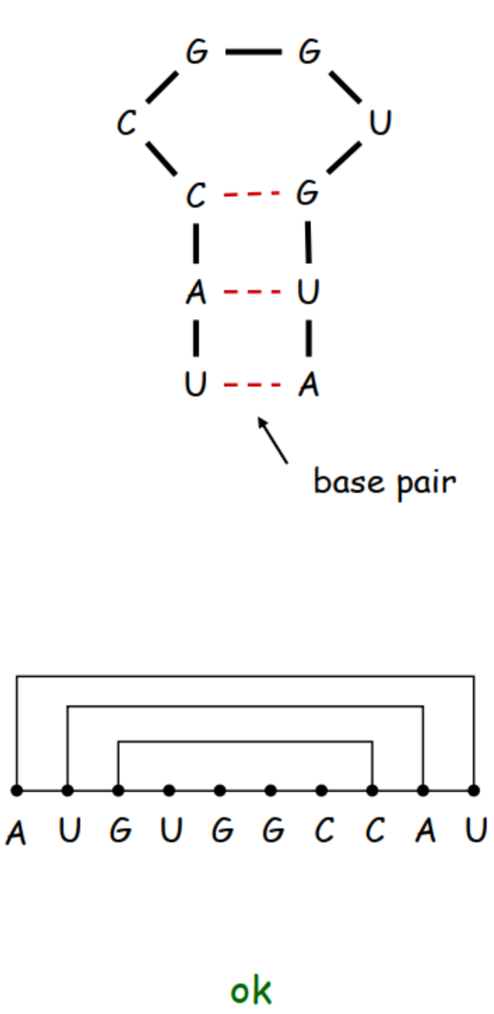
# RNA Secondary Structure

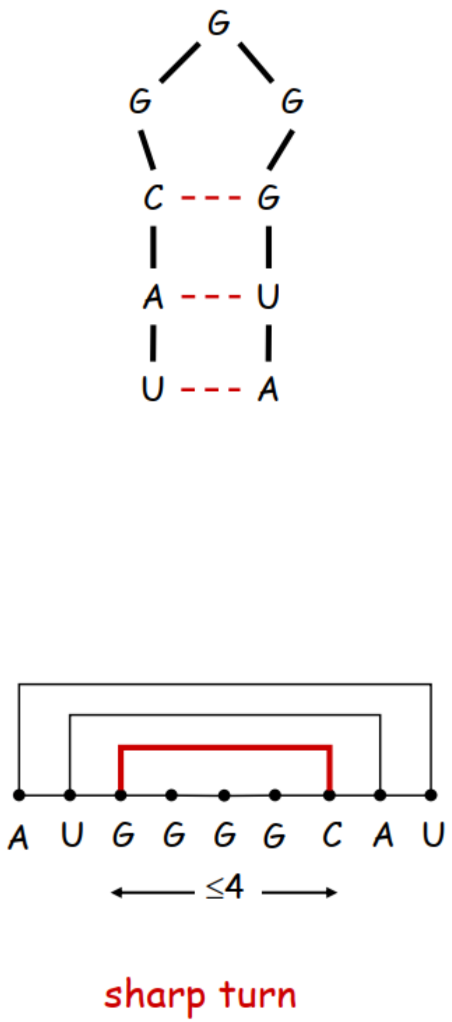
- **Secondary structure.** A set of pairs S = { $(b_i, b_j)$} that satisfy:
  - [Watson-Crick]. S is matching and each pair in S is a Watson-Crick complement: A-U, U-A, C-G, or G-C
  - [No sharp turns]. The ends of each pair are separated by at least 4 intervening bases. If $(b_i, b_j) \in S$, then $i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

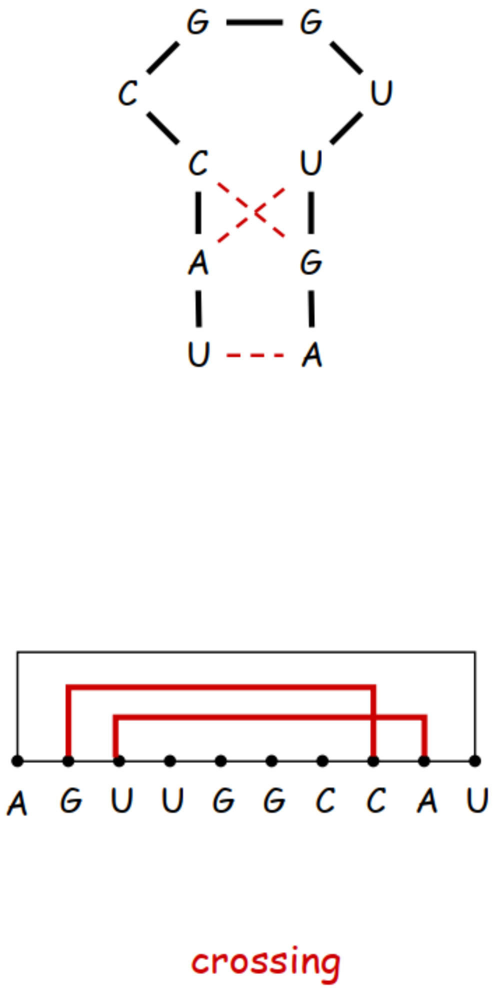
# RNA Secondary Structure: Examples



- **Examples**



# RNA Secondary Structure



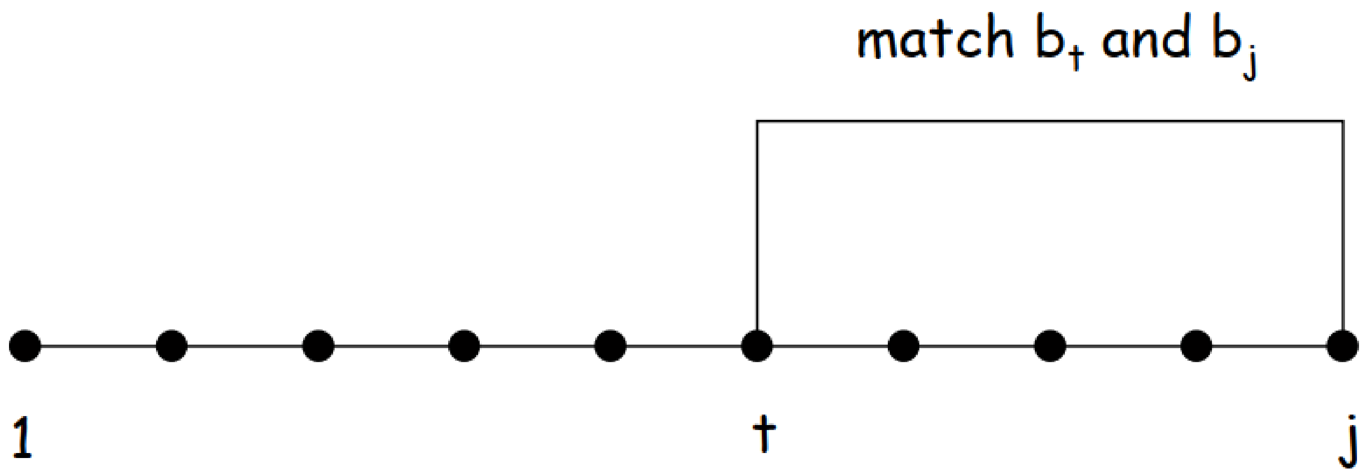
- **Free energy.** Usual hypothesis is that an RNA molecule will form the secondary structure with the optimum total free energy

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

# RNA Secondary Structure: Subproblems

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




- **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 $b_i b_{i+1} \ldots b_j$

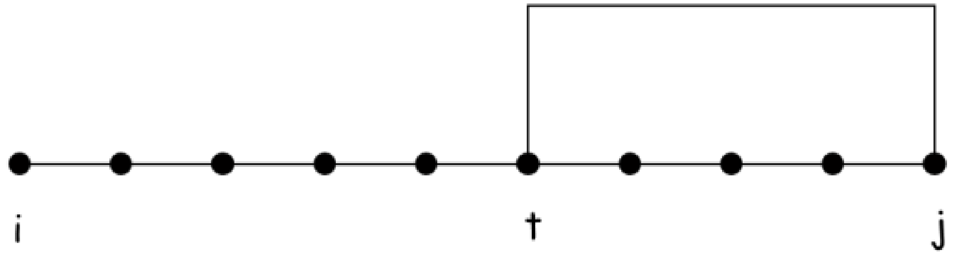
- If i >= j – 4
  - OPT(i, j) = 0 by no-sharp turns condition

- If i < j – 4: take max of two cases
  - Case 1. Base $b_j$ is not involved in a pair
    - OPT(i, j – 1)
  - Case 2. Base $b_j$ pairs with $b_t$ for some i <= t < j – 4

    Non-crossing constraint decouples resulting sub-problems

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

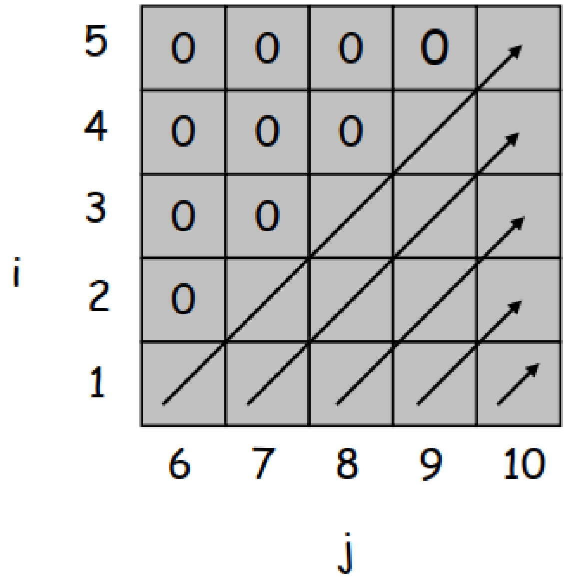
take max over t such that $i \le t < j-4$ and $b_t$ and $b_j$ are Watson-Crick complements



# Bottom Up Dynamic Programming Over Intervals

- **Q.** What order to solve the sub-problems?
- **A.** Do shortest intervals first

```
RNA(b1,…,bn) {
   for k = 5, 6, …, n-1
      for i = 1, 2, …, n-k
         j = i + k
         Compute M[i, j]

   return M[1, n]
}
```




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

# Sequence Alignment

# String Similarity



- How similar are two strings?
  - ocurrance
  - 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

- **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

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

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

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

$2\delta + \alpha_{CA}$

- **Applications**
  - Basis for Unix diff
  - Speech recognition
  - Computational biology

# Sequence Alignment



- **Goal.** Given two strings $X = x_1x_2 \ldots x_m$ and $Y = y_1y_2 \ldots y_n$ 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.** 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 = $x_2 - y_1, x_3 - y_2, x_4 - y_3, x_5 - y_4, x_6 - y_6$

| | | | | | | |
|---|---|---|---|---|---|---|
| $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ | | $x_6$ |
| C | T | A | C | C | - | G |
| - | T | A | C | A | T | G |
| | $y_1$ | $y_2$ | $y_3$ | $y_4$ | $y_5$ | $y_6$ |

# Sequence Alignment: Problem Structure

上海科技大学 ShanghaiTech University

- **Def.** OPT(i, j) = min cost of aligning strings $x_1x_2\ldots x_i$ and $y_1y_2\ldots y_j$
- Case 1: OPT matches $x_i - y_j$
  - pay mismatch for $x_i - y_j$ + min cost of aligning two strings $x_1x_2\ldots x_{i-1}$ and $y_1y_2\ldots y_{j-1}$
- Case 2a: OPT leaves $x_i$ unmatched
  - pay gap for $x_i$ and min cost of aligning $x_1x_2\ldots x_{i-1}$ and $y_1y_2\ldots y_j$
- Case 2b: OPT leaves $y_j$ unmatched
  - pay gap for $y_j$ and min cost of aligning $x_1x_2\ldots x_i$ and $y_1y_2\ldots y_{j-1}$

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

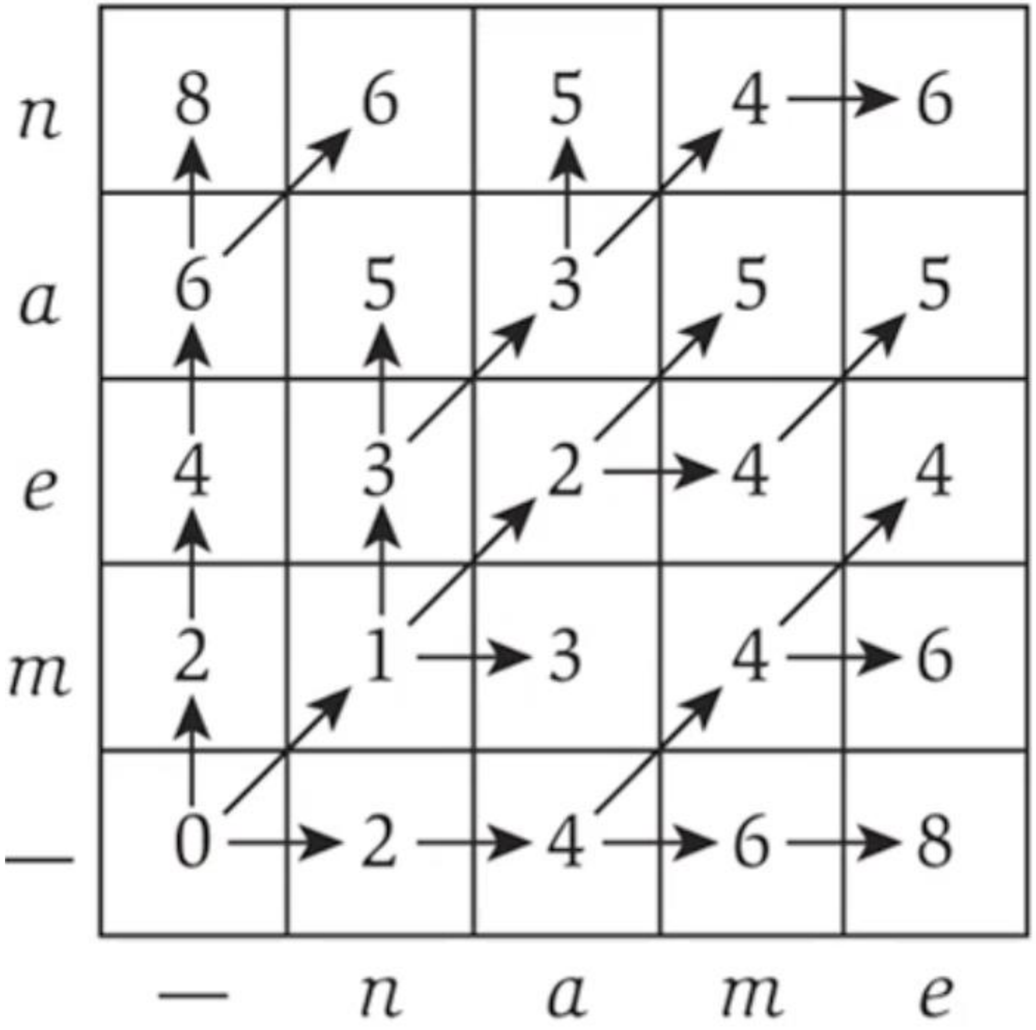
立志成才报国裕民

# Sequence Alignment: Algorithm

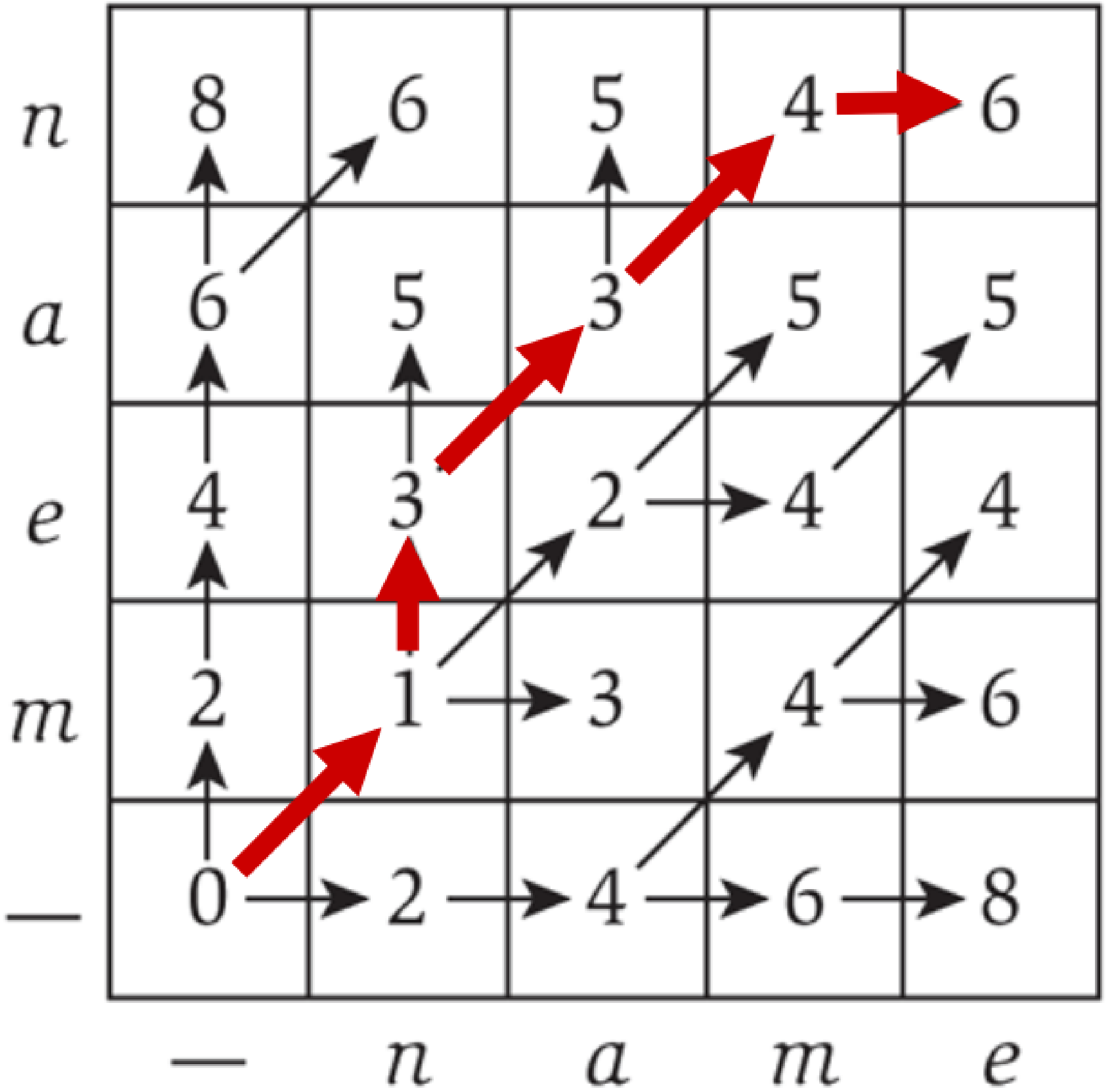
```
Sequence-Alignment(m, n, x1x2...xm, y1y2...yn, δ, α) {
   for i = 0 to m
      M[0, i] = iδ
   for j = 0 to n
      M[j, 0] = jδ

   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]
}
```

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

# Sequence Alignment: Example



# Sequence Alignment: Example



# Sequence Alignment: Algorithm

- **Analysis:** $\Theta(mn)$ time and space
- **English words or sentences:**
  - m, n <= 30. ← OK

- **Computational biology**
  - m = n = 100,000
  - 10 billions ops is OK, but 10GB array is quite large

# Sequence Alignment: Linear Space



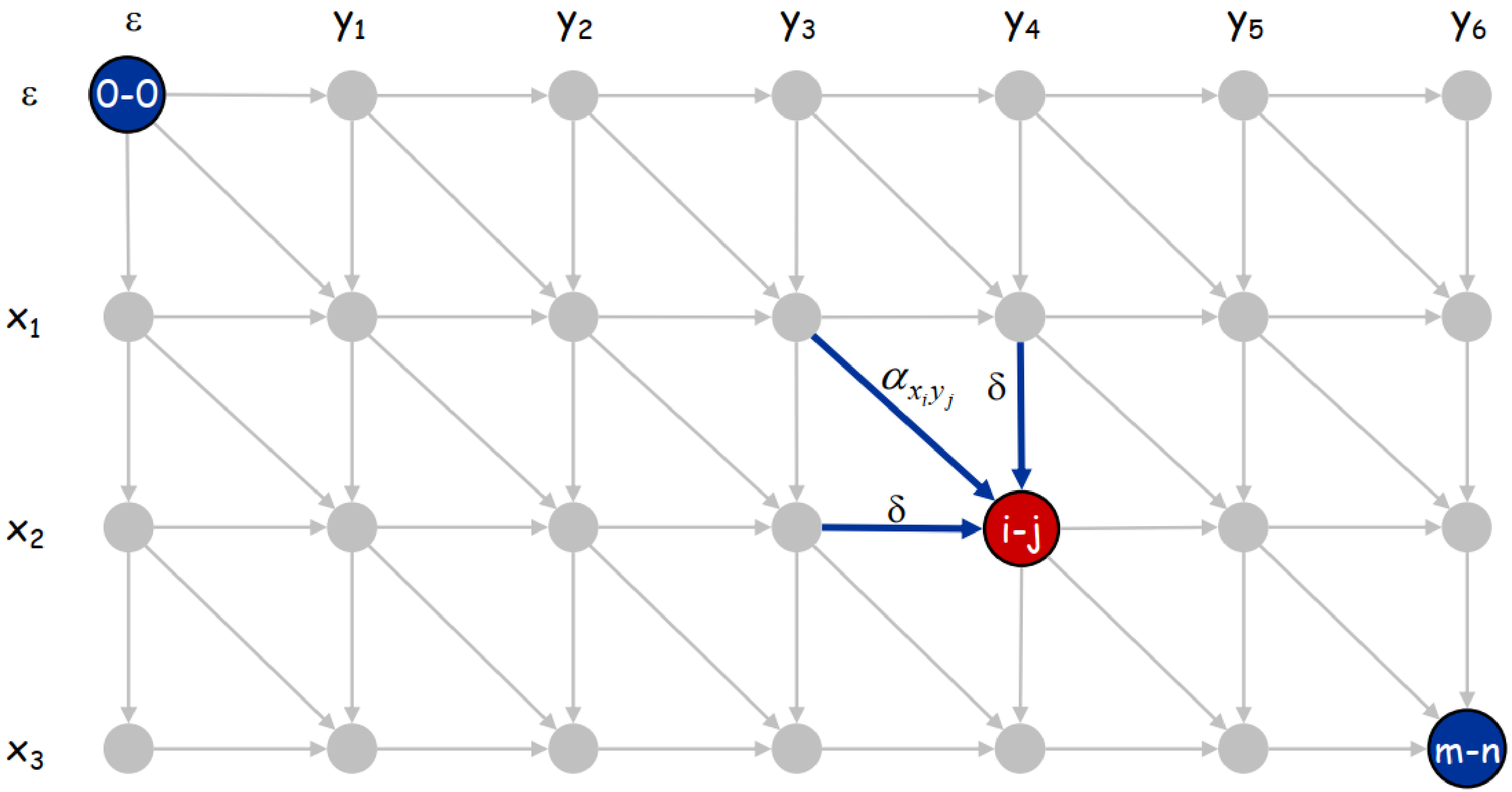
- **Q.** Can we avoid using quadratic **space**?
- **Easy.** Optimal **cost** in O(m + n) space and O(mn) time
  - Compute OPT(i, ·) from OPT(i-1, ·)
  - No longer a simple way to recover alignment itself

- **Theorem.** [Hirschberg 1975] Optimal **alignment** in O(m + n) space and O(mn) time
  - Clever combination of divide-and-conquer and dynamic programming

# Sequence Alignment in Linear Space

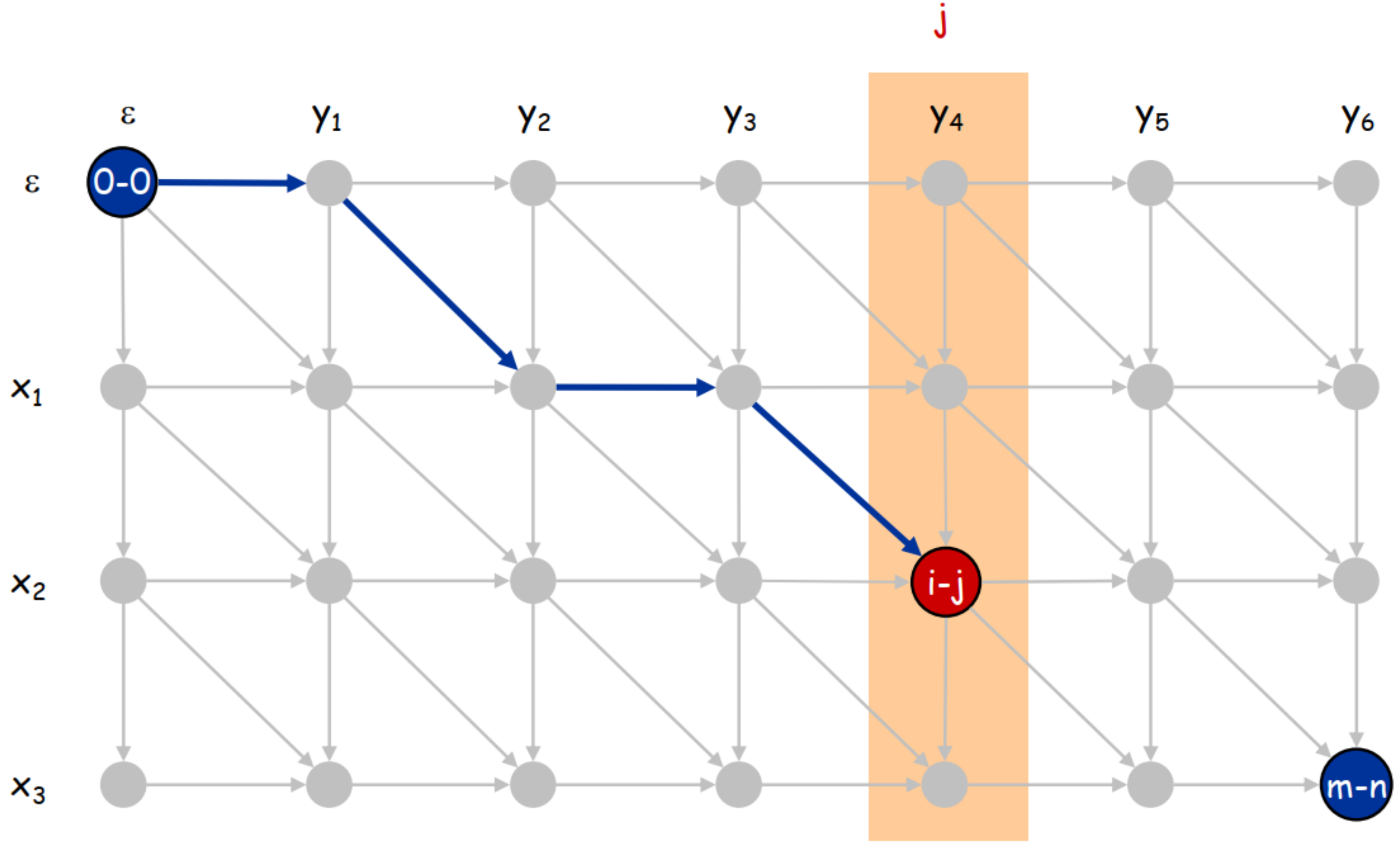
# Sequence Alignment: Linear Space

- **Edit distance graph**
  - Let f(i, j) be shortest path from (0, 0) to (i, j)
  - Observation: f(i, j) = OPT(i, j)



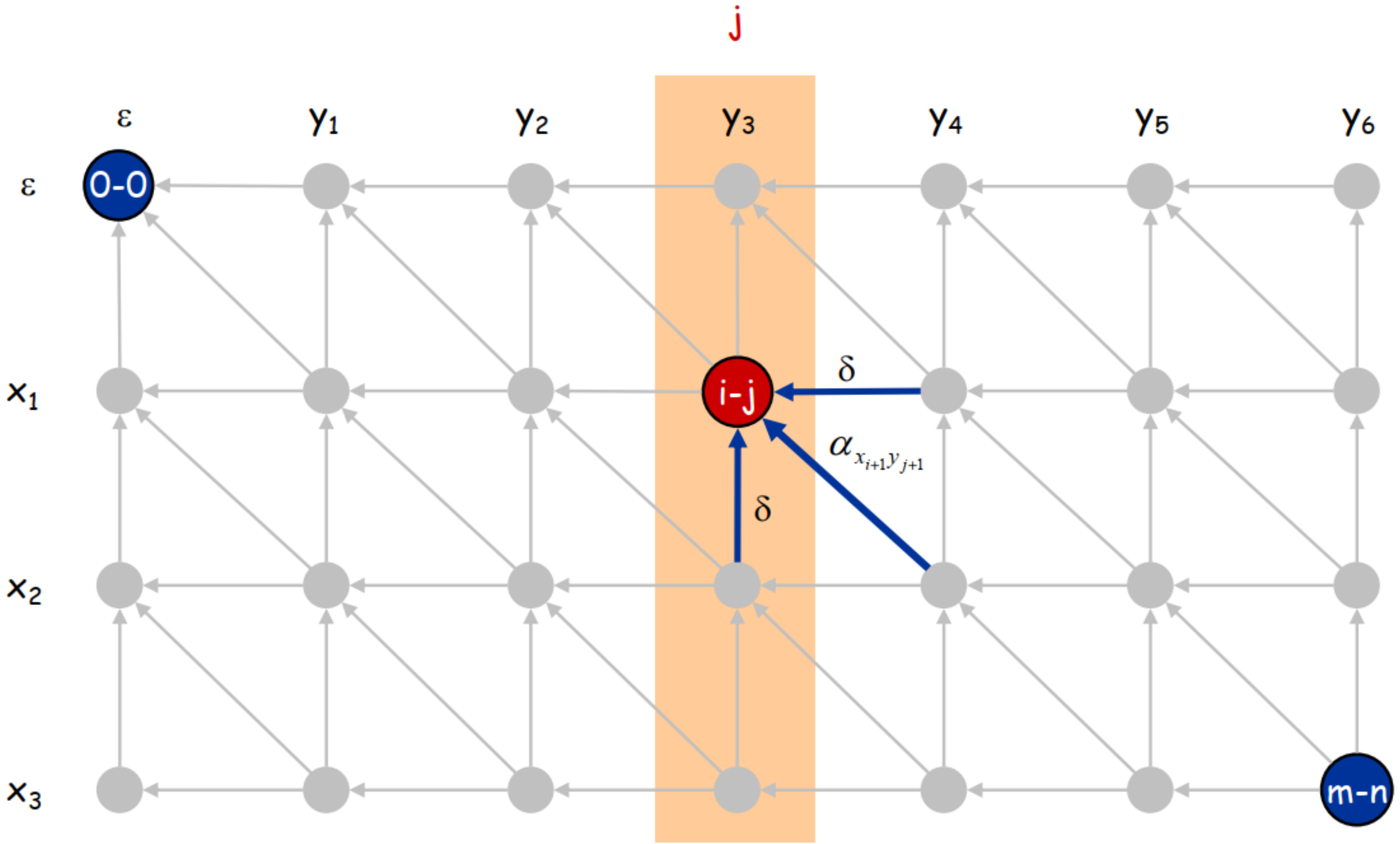
# Sequence Alignment: Linear Space

- **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



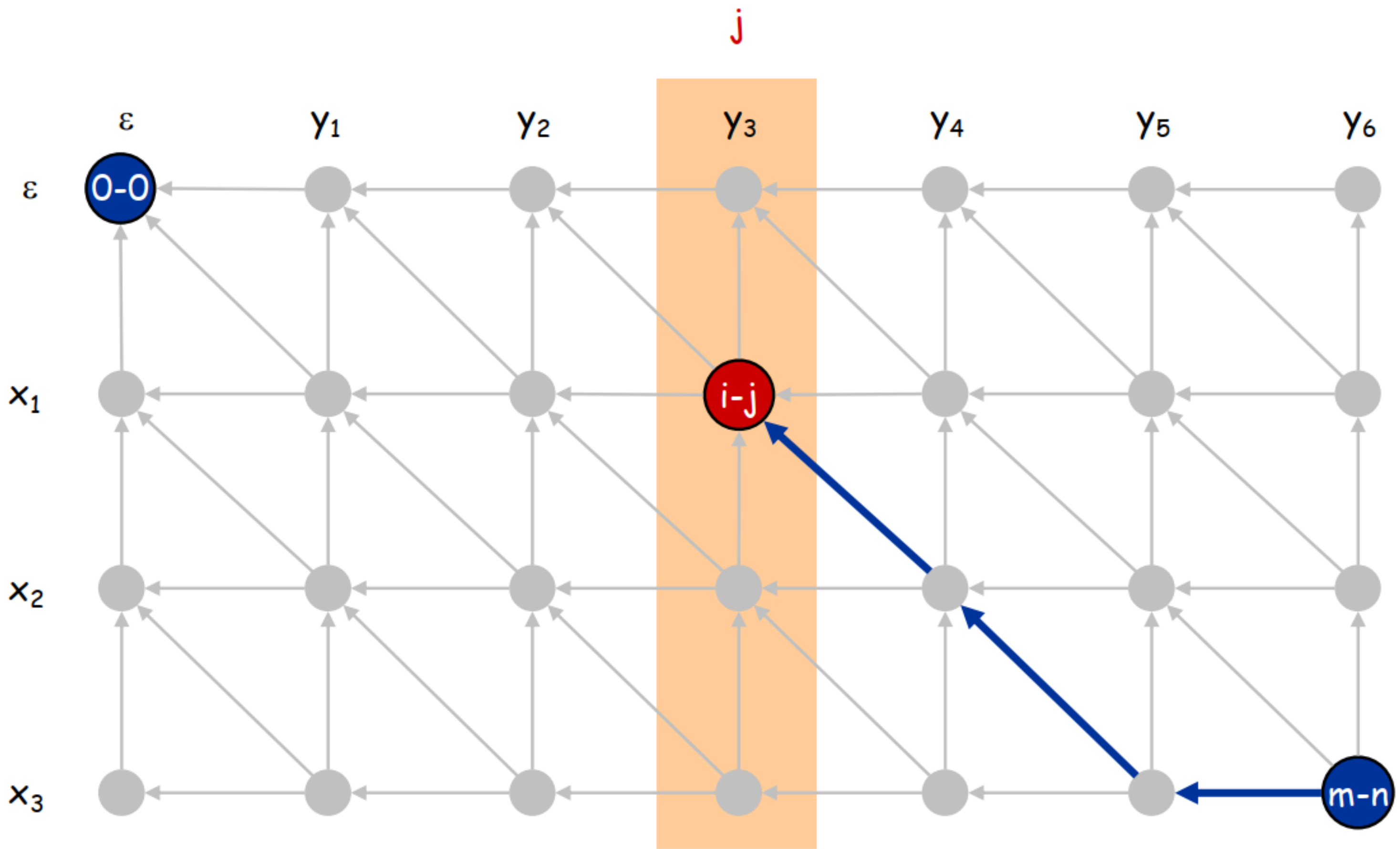
# Sequence Alignment: Linear Space

- **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)



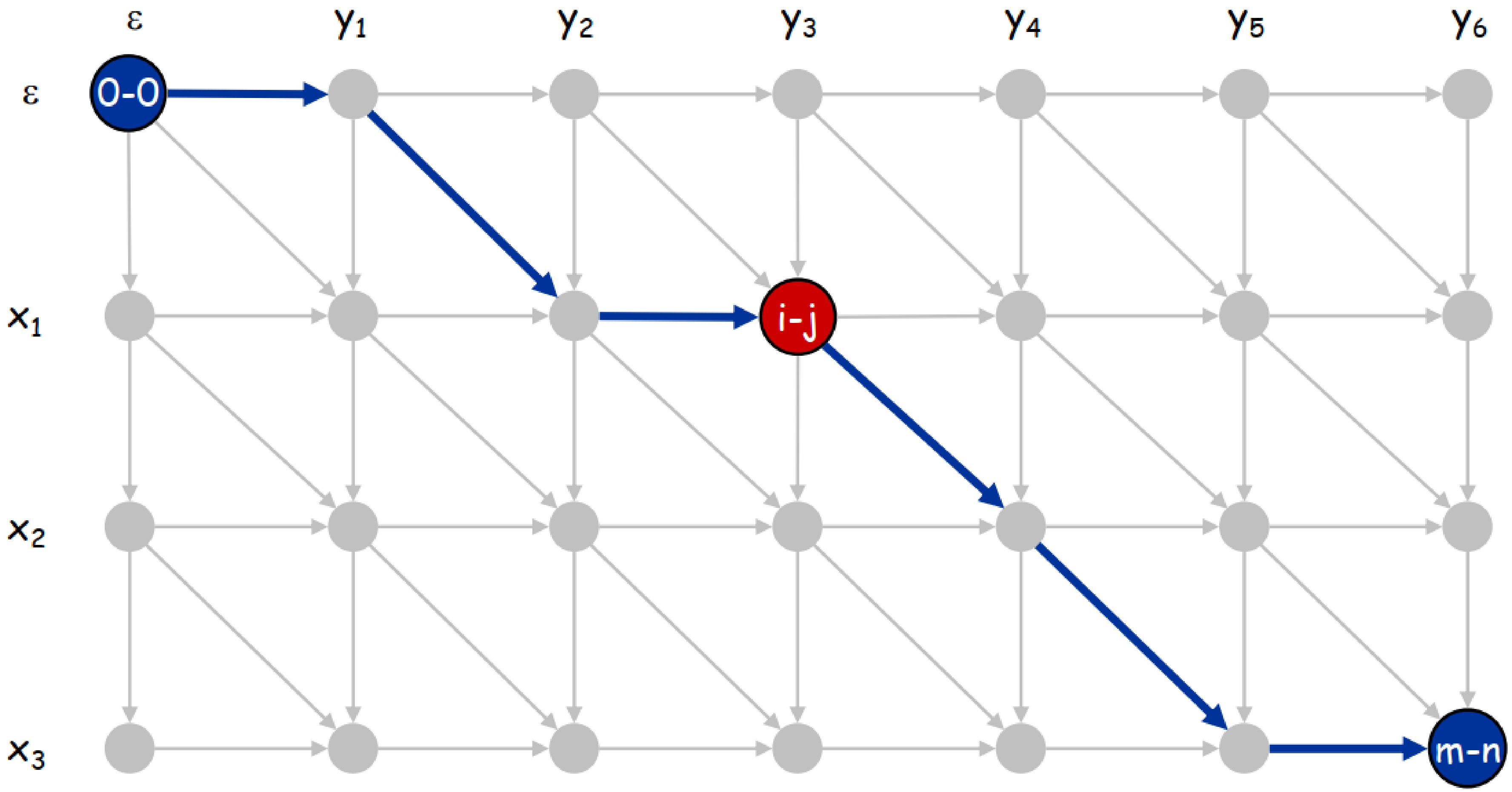
# Sequence Alignment: Linear Space

- **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



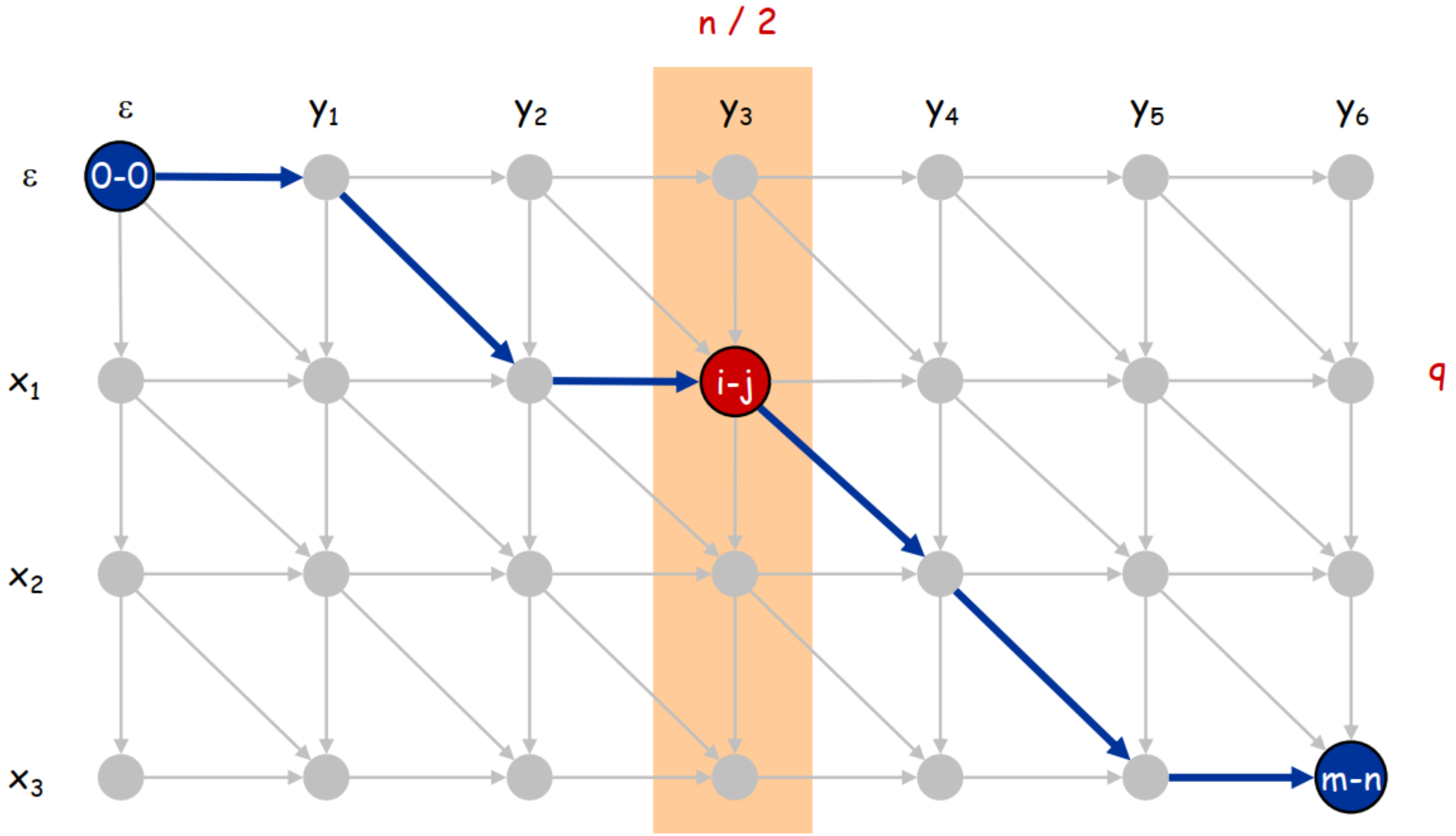
# Sequence Alignment: Linear Space

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



# Sequence Alignment: Linear Space

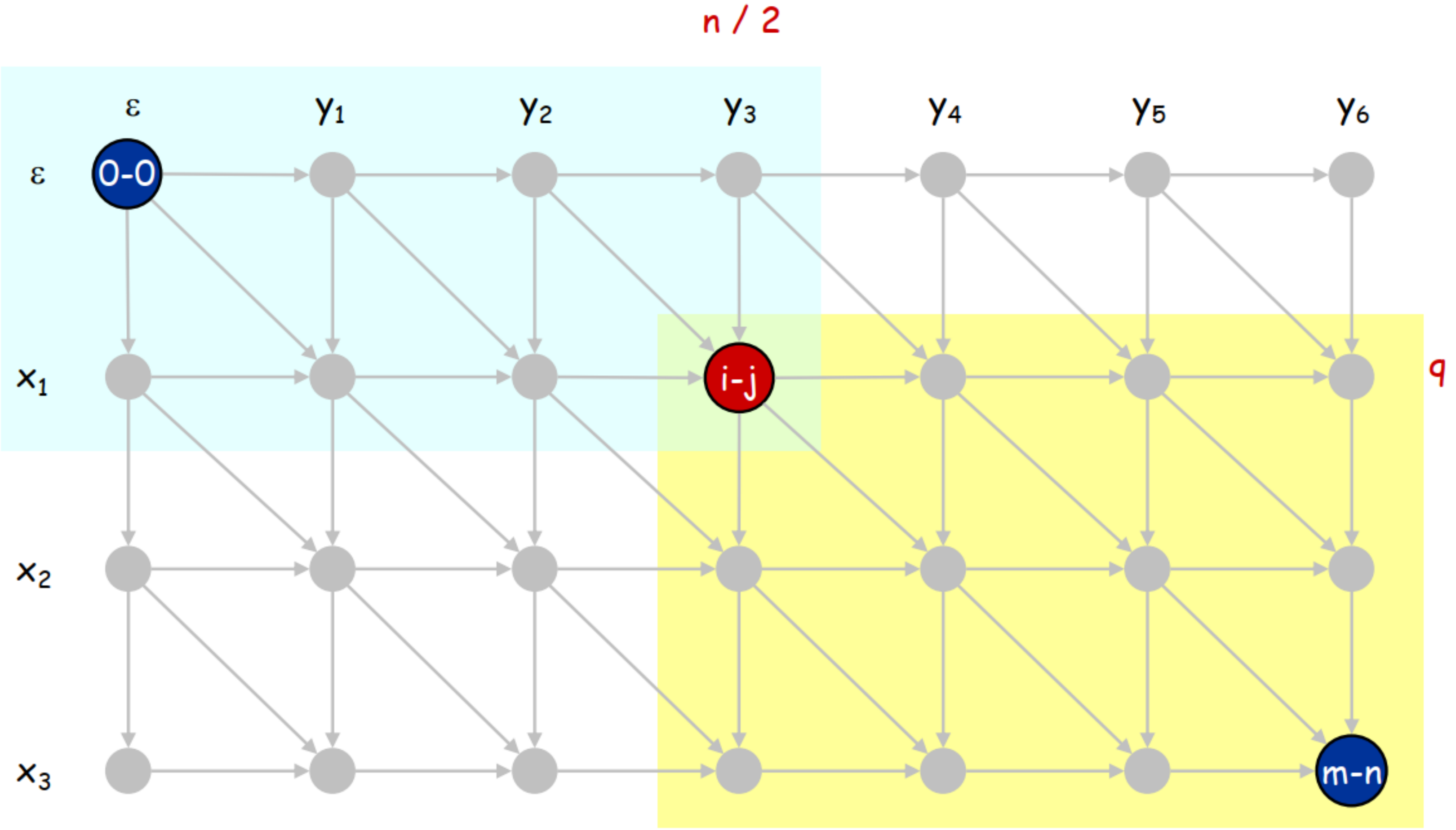
- **Observation 2.** Let q be an index that minimized f(q, n/2) + g(q, n/2)
- Then, the shortest path from (0, 0) to (m, n) uses (q, n/2)



# Sequence Alignment: Linear Space



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

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

- 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

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

# Next Time:
# Dynamic Programming (Cont.)