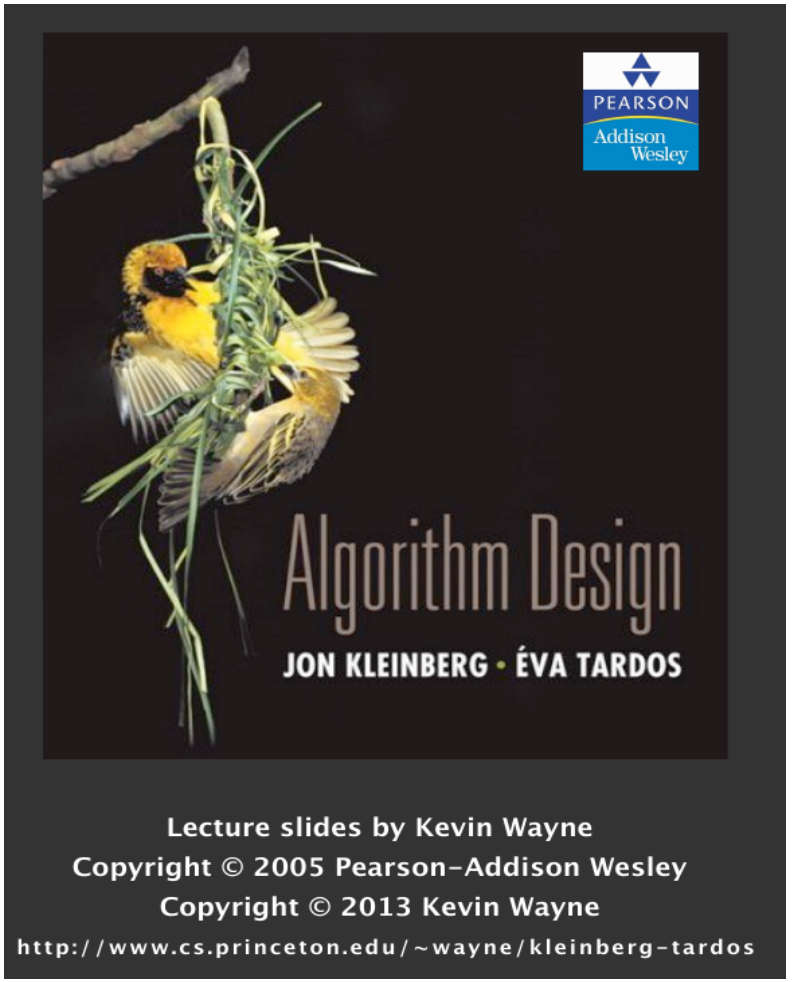
# Dynamic Programming

*Although there is no chapter dedicated to Dynamic Programming in AIMA, this technique is at the core of several fundamental techniques: Reinforcement Learning, Hidden Markov Models, Language Processing, and Search Algorithms*




*Credits: this week most material borrowed from Kevin Wayne's texbook*

*Last modified on 2018/10/14 by f.maire@qut.edu.au*

# Reading for this week

- Page 342 of AIMA
  - Section marked Dynamic Programming (DP)
  - Looking a the index of the textbook (page 1104), you will realise that DP is pervasive in AI

Russell and Norvig Textbook:
*Artificial Intelligence, a Modern Approach*
*3rd* edition

# Today's Menu

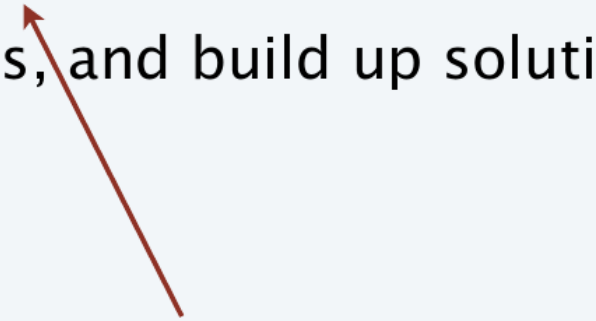
- Algorithm paradigms
- A bit of history
- Examples
  - weighted interval scheduling problem
  - segmented least squares
  - knapsack problem
  - string edit distance (alignment)

# Algorithmic paradigms

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

Divide-and-conquer. Break up a problem into independent subproblems, solve each subproblem, and combine solution to subproblems to form solution to original problem.

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



# Dynamic programming history

Bellman. Pioneered the systematic study of dynamic programming in 1950s.

Etymology.

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

THE THEORY OF DYNAMIC PROGRAMMING

RICHARD BELLMAN

1. **Introduction.** Before turning to a discussion of some representative problems which will permit us to exhibit various mathematical features of the theory, let us present a brief survey of the fundamental concepts, hopes, and aspirations of dynamic programming.

To begin with, the theory was created to treat the mathematical problems arising from the study of various multi-stage decision processes, which may roughly be described in the following way: We have a physical system whose state at any time $t$ is determined by a set of quantities which we call state parameters, or state variables. At certain times, which may be prescribed in advance, or which may be determined by the process itself, we are called upon to make decisions which will affect the state of the system. These decisions are equivalent to transformations of the state variables, the choice of a decision being identical with the choice of a transformation. The outcome of the preceding decisions is to be used to guide the choice of future ones, with the purpose of the whole process that of maximizing some function of the parameters describing the final state.

Examples of processes fitting this loose description are furnished by virtually every phase of modern life, from the planning of industrial production lines to the scheduling of patients at a medical clinic; from the determination of long-term investment programs for universities to the determination of a replacement policy for machinery in factories; from the programming of training policies for skilled and unskilled labor to the choice of optimal purchasing and inventory policies for department stores and military establishments.

# Dynamic programming applications

Areas.

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

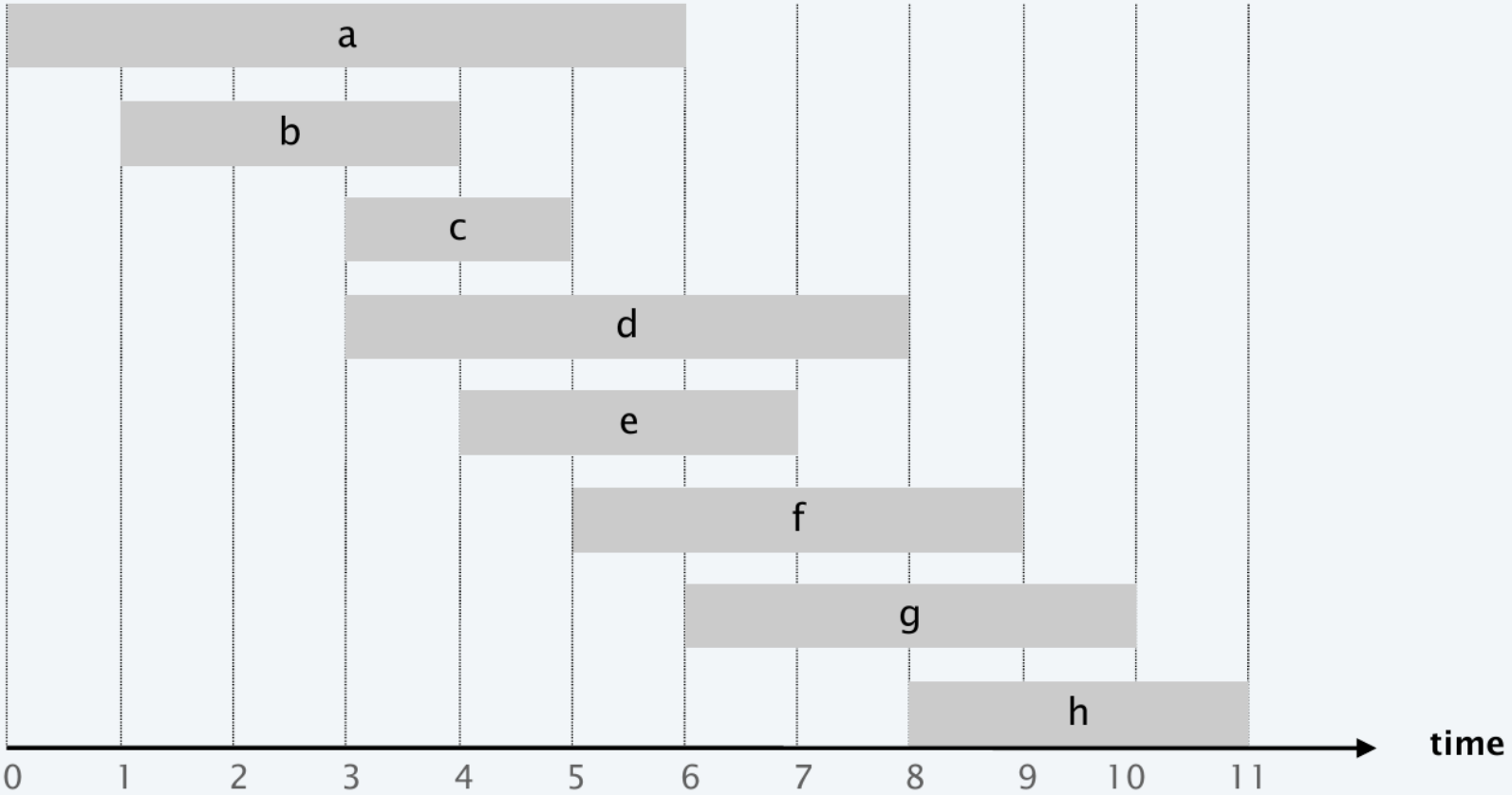
Some famous dynamic programming algorithms.

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

# 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.



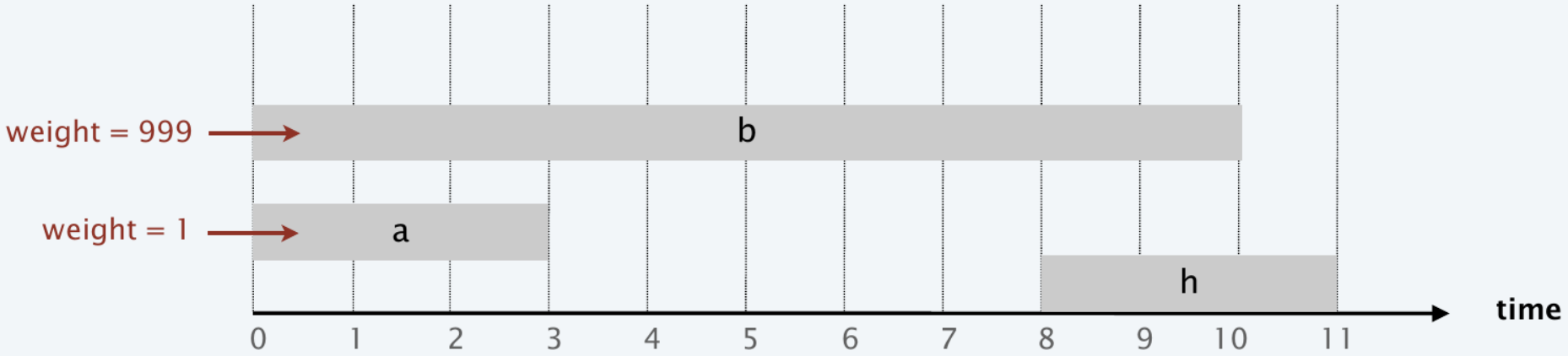
# Earliest-finish-time first algorithm

Earliest finish-time first.

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

Recall. Greedy algorithm is correct if all weights are 1.

Observation. Greedy algorithm fails spectacularly for weighted version.

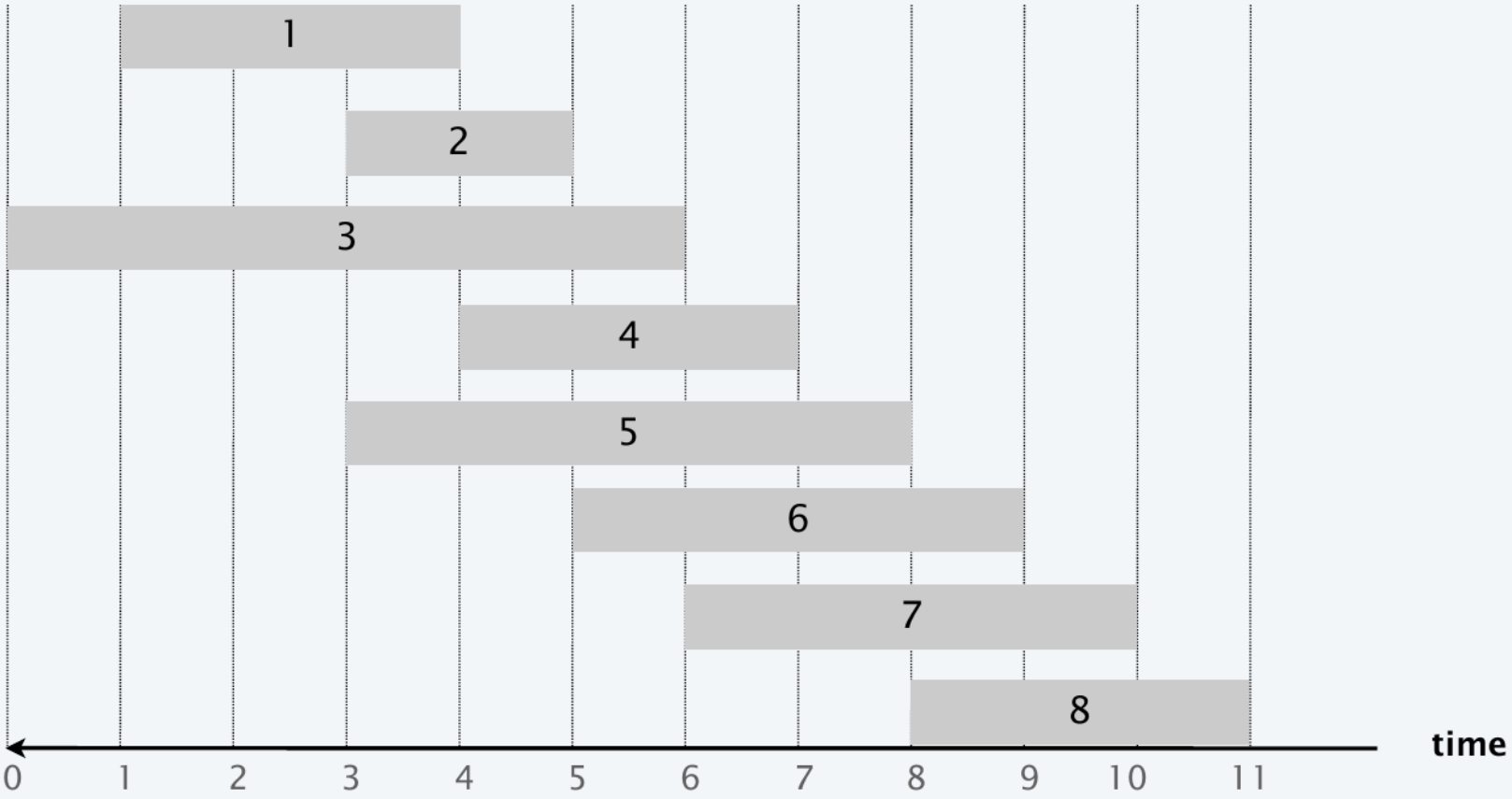


# Weighted interval scheduling

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. $p(8) = 5, p(7) = 3, p(2) = 0.$



# Dynamic programming: binary choice

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

Case 1. $OPT$ selects job $j$.

- Collect profit $v_j$.
- Can't use incompatible jobs $\{ p(j) + 1, p(j) + 2, \ldots, j - 1 \}$.
- Must include optimal solution to problem consisting of remaining compatible jobs $1, 2, \ldots, p(j)$.

optimal substructure property
(proof via exchange argument)

Case 2. $OPT$ does not select job $j$.

- Must include optimal solution to problem consisting of remaining compatible jobs $1, 2, \ldots, j - 1$.

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

# Weighted interval scheduling: brute force

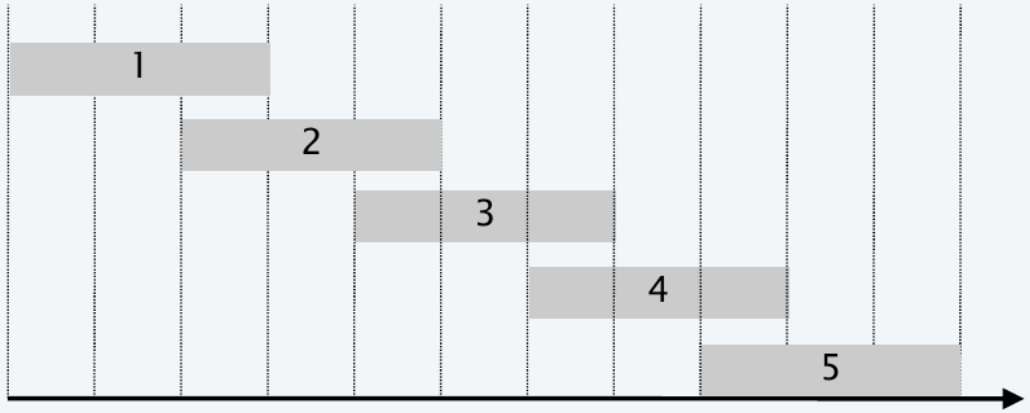
```
Input:  n, s[1..n], f[1..n], v[1..n]
Sort jobs by finish time so that f[1] ≤ f[2] ≤ … ≤ f[n].
Compute p[1], p[2], …, p[n].

Compute-Opt(j)
if j = 0
   return 0.
else
   return max(v[j] + Compute-Opt(p[j], Compute-Opt(j-1))).
```

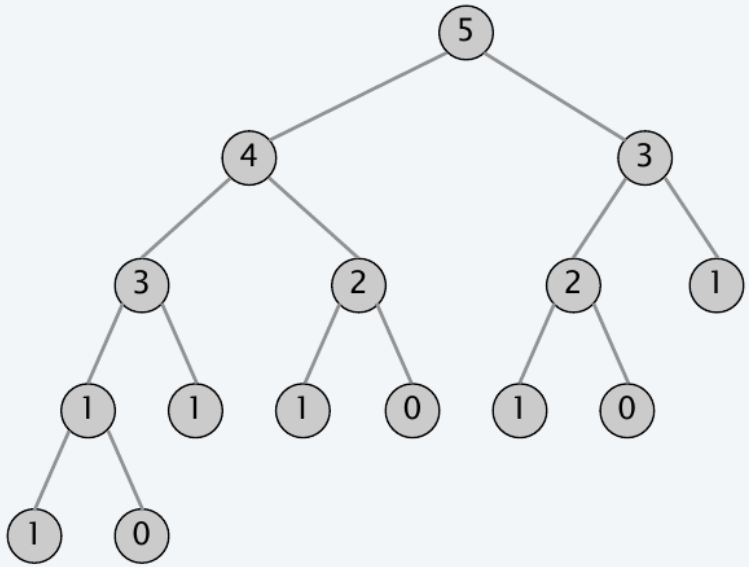
# Weighted interval scheduling: brute force

Observation. Recursive algorithm fails spectacularly because of redundant subproblems $\Rightarrow$ exponential algorithms.

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




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




recursion tree

# Weighted interval scheduling: memoization

Memoization. Cache results of each subproblem; lookup as needed.

```
Input:  n, s[1..n], f[1..n], v[1..n]
Sort jobs by finish time so that f[1] ≤ f[2] ≤ … ≤ f[n].
Compute p[1], p[2], …, p[n].

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

M-Compute-Opt(j)
---------------------------------------------------------
if M[j] is empty
   M[j] ← max(v[j] + M-Compute-Opt(p[j]), 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 $p(\cdot)$ : $O(n \log n)$ via sorting by start time.

- M-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 $\Phi$ by $1 \Rightarrow$ at most $2n$ recursive calls.

- Overall running time of M-COMPUTE-OPT($n$) is $O(n)$. ▪

Remark. $O(n)$ if jobs are presorted by start and finish times.

# Weighted interval scheduling: finding a solution

Q. DP algorithm computes optimal value. How to find solution itself?

A. Make a second pass.

```
Find-Solution(j)
if j = 0
   return ∅.
else if (v[j] + M[p[j]] > M[j-1])
   return { j } ∪ Find-Solution(p[j]).
else
   return Find-Solution(j-1).
```

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

# Weighted interval scheduling: bottom-up

Bottom-up dynamic programming. Unwind recursion.

BOTTOM-UP $(n, s_1, \ldots, s_n, f_1, \ldots, f_n, v_1, \ldots, v_n)$

Sort jobs by finish time so that $f_1 \leq f_2 \leq \ldots \leq f_n$.

Compute $p(1), p(2), \ldots, p(n)$.

$M[0] \leftarrow 0$.

FOR j = 1 TO $n$

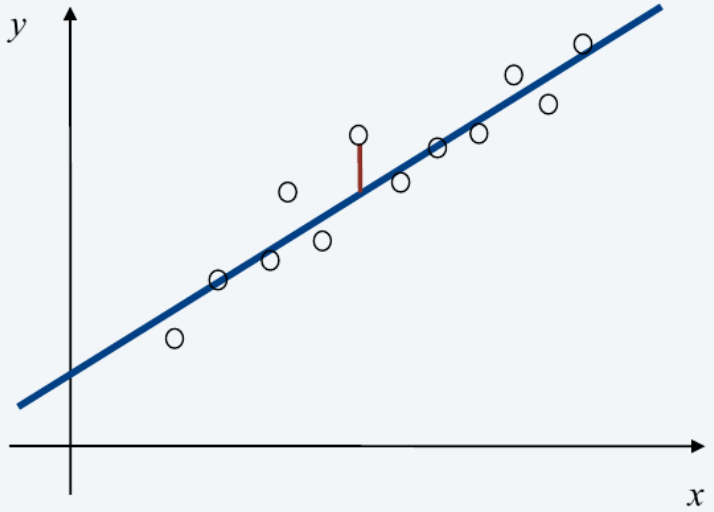
$M[j] \leftarrow \max \{ v_j + M[p(j)], M[j-1] \}$.

# Least squares

Least squares. Foundational problem in statistics.

- Given $n$ points in the plane: $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$.
- Find a line $y = ax + b$ that minimizes the sum of the squared error:

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




Solution. Calculus $\Rightarrow$ min error is achieved when

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

# Segmented least squares

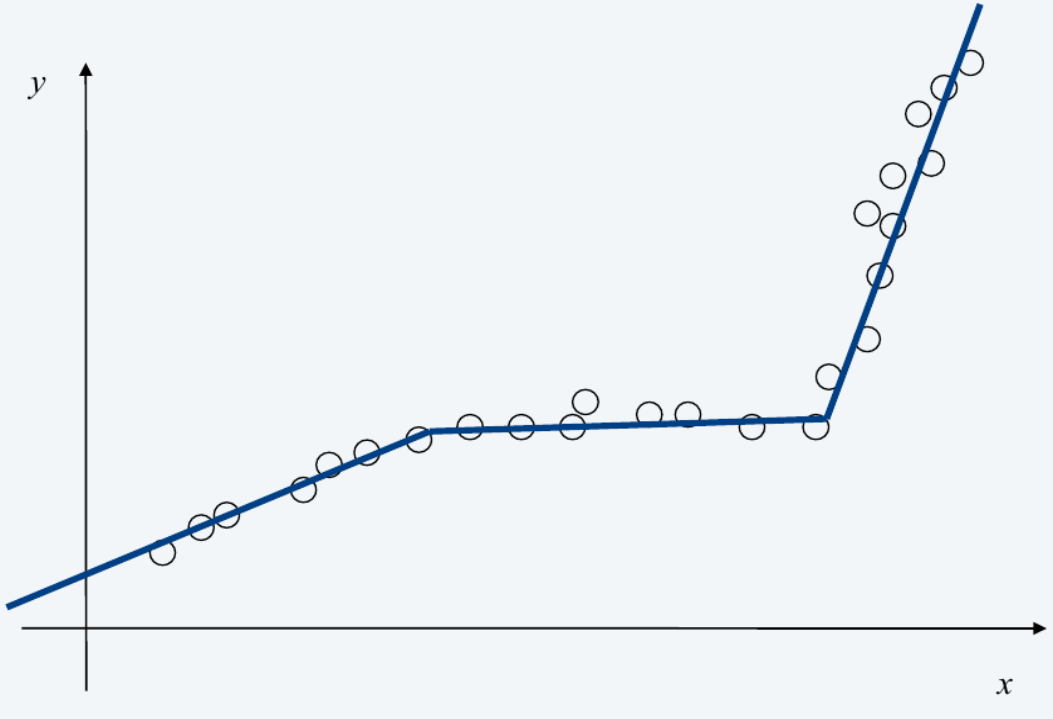
## Segmented least squares.

- Points lie roughly on a sequence of several line segments.
- Given $n$ points in the plane: $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ with $x_1 < x_2 < \ldots < x_n$, find a sequence of lines that minimizes $f(x)$.

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

goodness of fit

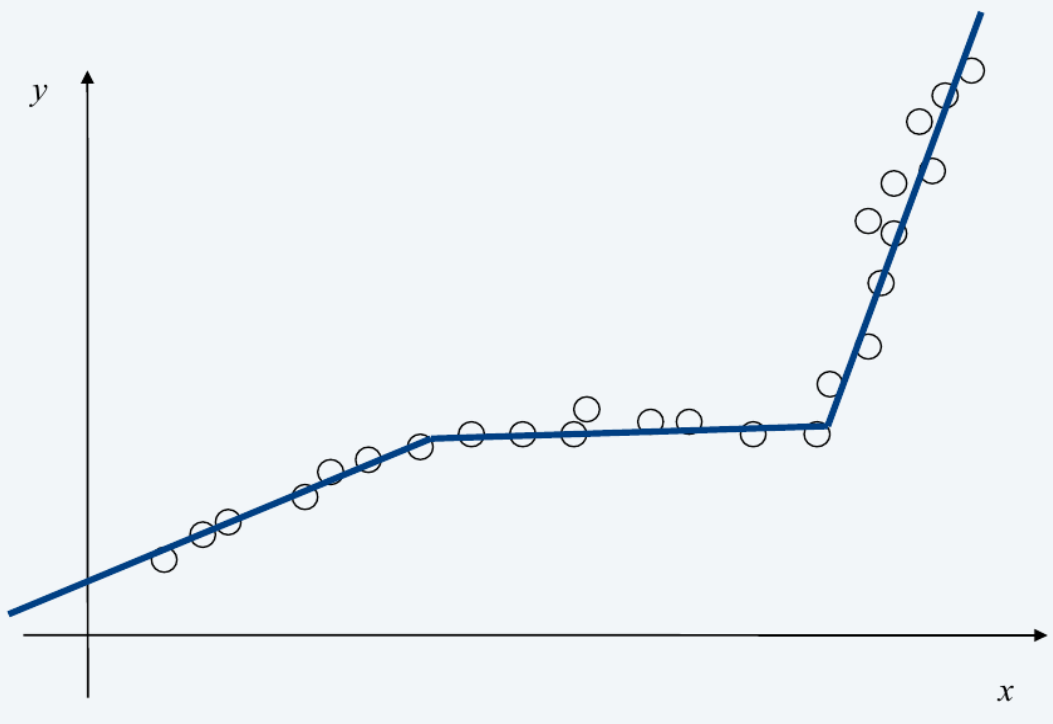
number of lines



# Segmented least squares

Given $n$ points in the plane: $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ with $x_1 < x_2 < \ldots < x_n$ and a constant $c > 0$, find a sequence of lines that minimizes $f(x) = E + c\,L$:

- $E$ = the sum of the sums of the squared errors in each segment.
- $L$ = the number of lines.



# Dynamic programming: multiway choice

## Notation.

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

## To compute $OPT(j)$:

- Last segment uses points $p_i, p_{i+1}, \ldots, p_j$ for some $i$.
- Cost = $e(i, j) + c + OPT(i-1)$. ← optimal substructure property (proof via exchange argument)

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \min\limits_{1 \le i \le j} \left\{ e(i,j) + c + OPT(i-1) \right\} & \text{otherwise} \end{cases}$$

# Segmented least squares algorithm

SEGMENTED-LEAST-SQUARES $(n,\ p_1,\ \ldots,\ p_n\ ,\ c)$

FOR j = 1 TO $n$

  FOR i = 1 TO $j$

    Compute the least squares $e(i, j)$ for the segment $p_i, p_{i+1}, \ldots, p_j$.

$M[0] \leftarrow 0.$

FOR $j = 1$ TO $n$

  $M[j] \leftarrow \min_{1 \le i \le j} \{ e_{ij} + c + M[i-1] \}.$

RETURN $M[n]$.

# Segmented least squares analysis

Theorem. [Bellman 1961] The dynamic programming algorithm solves the segmented least squares problem in $O(n^3)$ time and $O(n^2)$ space.

Pf.

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

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

Remark. Can be improved to $O(n^2)$ time and $O(n)$ space by precomputing various statistics. How?

# Knapsack problem

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

Ex. $\{ 1, 2, 5 \}$ has value 35.

Ex. $\{ 3, 4 \}$ has value 40.

Ex. $\{ 3, 5 \}$ has value 46 (but exceeds weight limit).

| $i$ | $v_i$ | $w_i$ |
|---|---|---|
| 1 | 1 | 1 |
| 2 | 6 | 2 |
| 3 | 18 | 5 |
| 4 | 22 | 6 |
| 5 | 28 | 7 |

**knapsack instance (weight limit W = 11)**

Greedy by value. Repeatedly add item with maximum $v_i$.

Greedy by weight. Repeatedly add item with minimum $w_i$.

Greedy by ratio. Repeatedly add item with maximum ratio $v_i / w_i$.

Observation. None of greedy algorithms is optimal.

# Dynamic programming: false start

Def. $OPT(i)$ = max profit subset of items $1, \ldots, i$.

Case 1. $OPT$ does not select item $i$.

- $OPT$ selects best of $\{ 1, 2, \ldots, i-1 \}$.

optimal substructure property
(proof via exchange argument)

Case 2. $OPT$ selects item $i$.

- Selecting 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 subproblems!

# Dynamic programming: adding a new variable

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

Case 1. $OPT$ does not select item $i$.

- $OPT$ selects best of $\{ 1, 2, \ldots, i-1 \}$ using weight limit $w$.

Case 2. $OPT$ selects item $i$.

optimal substructure property
(proof via exchange argument)

- New weight limit $= w - w_i$.
- $OPT$ selects best of $\{ 1, 2, \ldots, i-1 \}$ 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 $(n, W, w_1, \ldots, w_n, v_1, \ldots, v_n)$

FOR $w = 0$ TO $W$

  $M[0, w] \leftarrow 0$.

FOR $i = 1$ TO $n$

  FOR $w = 1$ TO $W$

  IF $(w_i > w)$ $M[i, w] \leftarrow M[i-1, w]$.

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

RETURN $M[n, W]$.

# Knapsack problem: bottom-up demo

| $i$ | $v_i$ | $w_i$ |
|---|---|---|
| 1 | 1 | 1 |
| 2 | 6 | 2 |
| 3 | 18 | 5 |
| 4 | 22 | 6 |
| 5 | 28 | 7 |

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

**weight limit w**

**subset of items 1, …, i**

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| { } | 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(i, w) = max profit subset of items 1, …, i with weight limit w.**

# Knapsack problem: running time

Theorem. There exists an algorithm to solve the knapsack problem with $n$ items and maximum weight $W$ in $\Theta(n\,W)$ time and $\Theta(n\,W)$ space.

Pf.

weights are integers between 1 and W

- Takes $O(1)$ time per table entry.
- There are $\Theta(n\,W)$ table entries.
- After computing optimal values, can trace back to find solution: take item $i$ in $OPT(i, w)$ iff $M[i, w] < M[i-1, w]$. ▪

## Remarks.

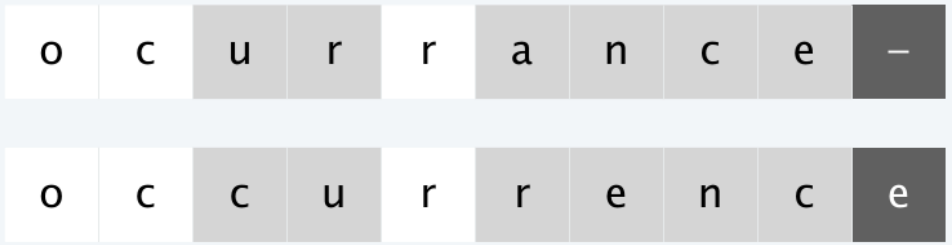
- Not polynomial in input size! ← "pseudo-polynomial"
- Decision version of knapsack problem is NP-COMPLETE.

**pseudo-polynomial time** if running time is polynomial in the numeric value of the input, but is exponential in the length of the input – the number of bits required to represent it.

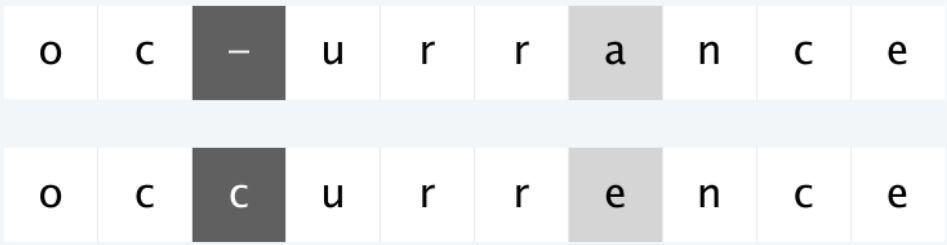
# String similarity

Q. How similar are two strings?

Ex. ocurrance and occurrence.




**6 mismatches, 1 gap**




**1 mismatch, 1 gap**


**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.


**cost = $\delta + \alpha_{CG} + \alpha_{TA}$**

Applications. Unix diff, speech recognition, computational biology, ...

# Sequence alignment

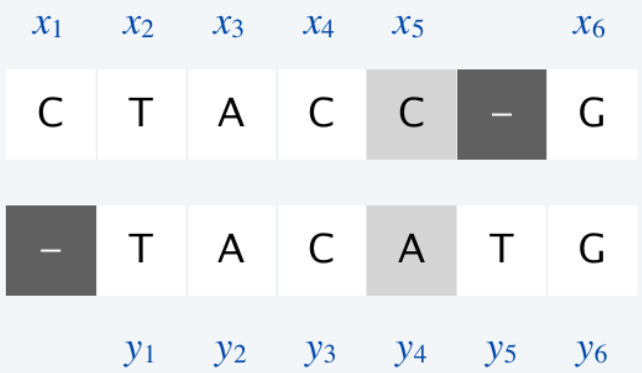
Goal. Given two strings $x_1 x_2 \ldots x_m$ and $y_1 y_2 \ldots y_n$ find min cost alignment.

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.

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

Def. The cost of an alignment $M$ is:

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




**an alignment of CTACCG and 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

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

Case 1. $OPT$ matches $x_i - y_j$.

Pay mismatch for $x_i - y_j$ + min cost of aligning $x_1 x_2 \ldots x_{i-1}$ and $y_1 y_2 \ldots y_{j-1}$.

Case 2a. $OPT$ leaves $x_i$ unmatched.

Pay gap for $x_i$ + 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 $y_j$ + min cost of aligning $x_1 x_2 \ldots x_i$ and $y_1 y_2 \ldots y_{j-1}$.

optimal substructure property
(proof via exchange argument)

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

# Sequence alignment: algorithm

SEQUENCE-ALIGNMENT $(m, n, x_1, \ldots, x_m, y_1, \ldots, y_n, \delta, \alpha)$

FOR $i = 0$ TO $m$

$M[i, 0] \leftarrow i\,\delta$.

FOR $j = 0$ TO $n$

$M[0, j] \leftarrow j\,\delta$.

FOR $i = 1$ TO $m$

FOR $j = 1$ TO $n$

$$M[i,j] \leftarrow \min \{\ \alpha[x_i, y_j] + M[i-1, j-1],$$
$$\delta + M[i-1, j],$$
$$\delta + M[i, j-1]).$$

RETURN $M[m, n]$.

## Sequence alignment: analysis

Theorem. The dynamic programming algorithm computes the edit distance (and optimal alignment) of two strings of length $m$ and $n$ in $\Theta(mn)$ time and $\Theta(mn)$ space.

Pf.

- Algorithm computes edit distance.
- Can trace back to extract optimal alignment itself. ▪

Q. Can we avoid using quadratic space?

A. Easy to compute optimal value in $O(mn)$ time and $O(m + n)$ space.

- Compute $\mathrm{OPT}(i, \bullet)$ from $\mathrm{OPT}(i - 1, \bullet)$.
- But, no longer easy to recover optimal alignment itself.

# Dynamic programming summary

## Outline.

- Polynomial number of subproblems.
- Solution to original problem can be computed from subproblems.
- Natural ordering of subproblems from smallest to largest, with an easy-to-compute recurrence that allows one to determine the solution to a subproblem from the solution to smaller subproblems.

## Techniques.

- Binary choice: weighted interval scheduling.
- Multiway choice: segmented least squares.
- Adding a new variable: knapsack problem.

# Non-examinable complement

- Hirschberg's algorithm is a clever modification of the Needleman-Wunsch Algorithm which still takes O(nm) time, but needs only O(min{n,m}) space.

- The remaining slides are not examinable, and are only provided as a complement for interested people.

# Hirschberg's algorithm

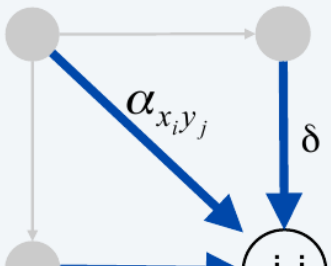
Edit distance graph.

- Let $f(i, j)$ be shortest path from $(0,0)$ to $(i, j)$.
- Lemma: $f(i, j) = OPT(i, j)$ for all $i$ and $j$.

Pf of Lemma. [ by strong induction on $i + j$ ]

- Base case: $f(0, 0) = OPT(0, 0) = 0$.
- Inductive hypothesis: assume true for all $(i', j')$ with $i' + j' < i + j$.
- Last edge on shortest path to $(i, j)$ is from $(i - 1, j - 1)$, $(i - 1, j)$, or $(i, j - 1)$.
- Thus,

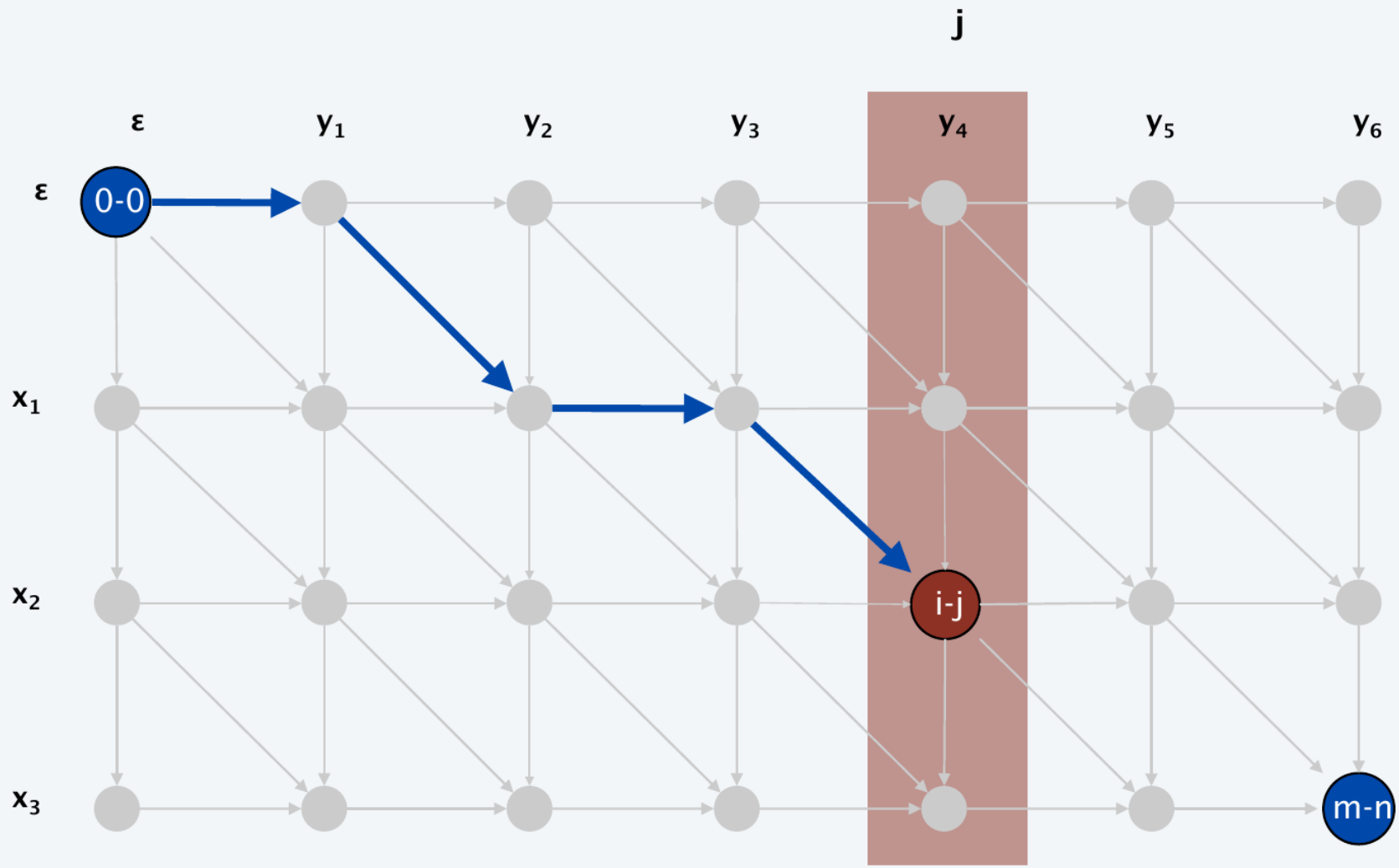
$$
\begin{aligned}
f(i, j) &= \min\{\alpha_{x_i y_j} + f(i-1, j-1),\ \delta + f(i-1, j),\ \delta + f(i, j-1)\} \\
&= \min\{\alpha_{x_i y_j} + OPT(i-1, j-1),\ \delta + OPT(i-1, j),\ \delta + OPT(i, j-1)\} \\
&= OPT(i, j) \quad \blacksquare
\end{aligned}
$$



# Hirschberg's algorithm

## Edit distance graph.

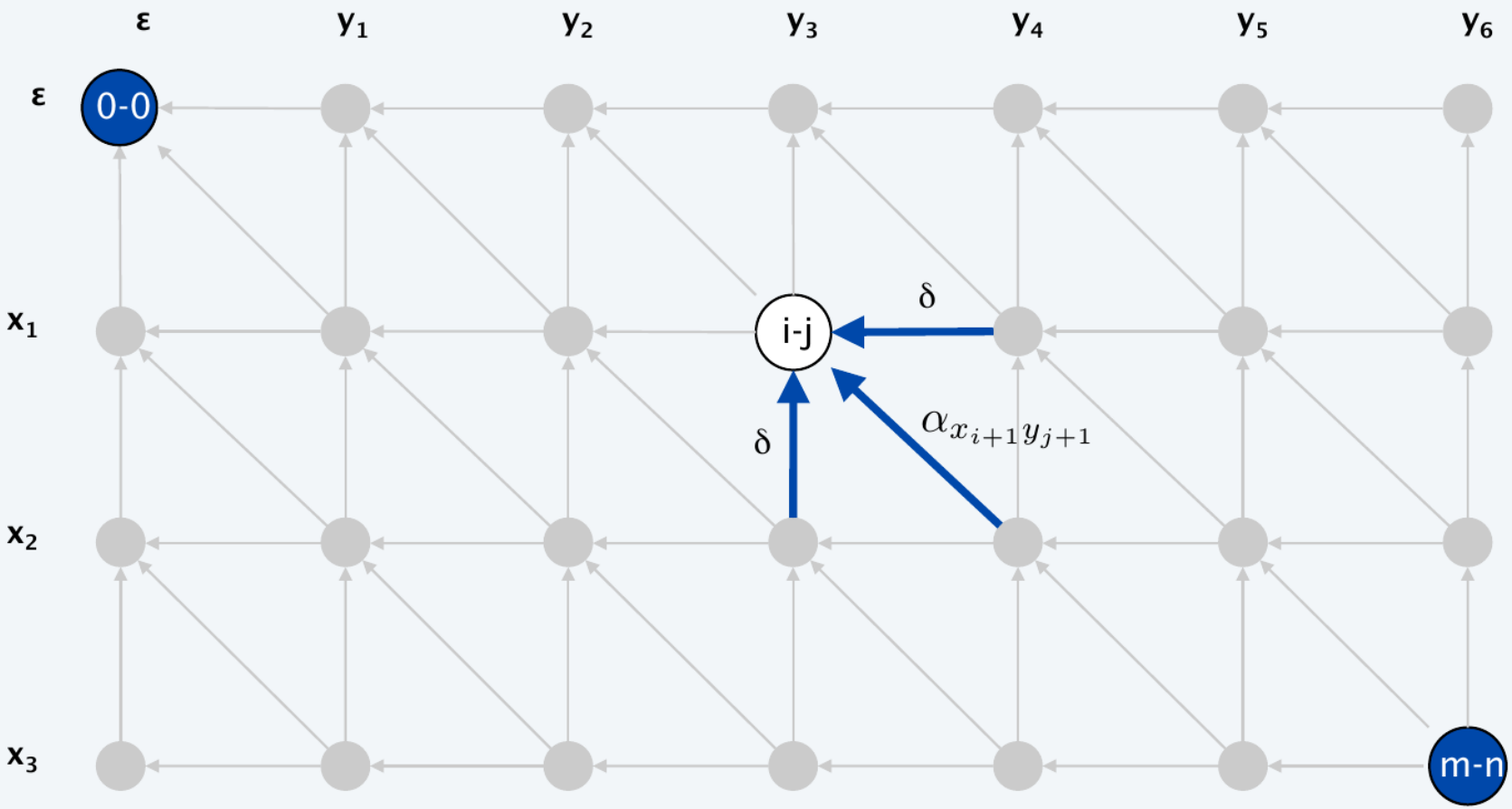
- Let $f(i, j)$ be shortest path from $(0,0)$ to $(i, j)$.
- Lemma: $f(i, j) = OPT(i, j)$ for all $i$ and $j$.
- Can compute $f(\bullet, j)$ for any $j$ in $O(mn)$ time and $O(m + n)$ space.



# Hirschberg's algorithm

## Edit distance graph.

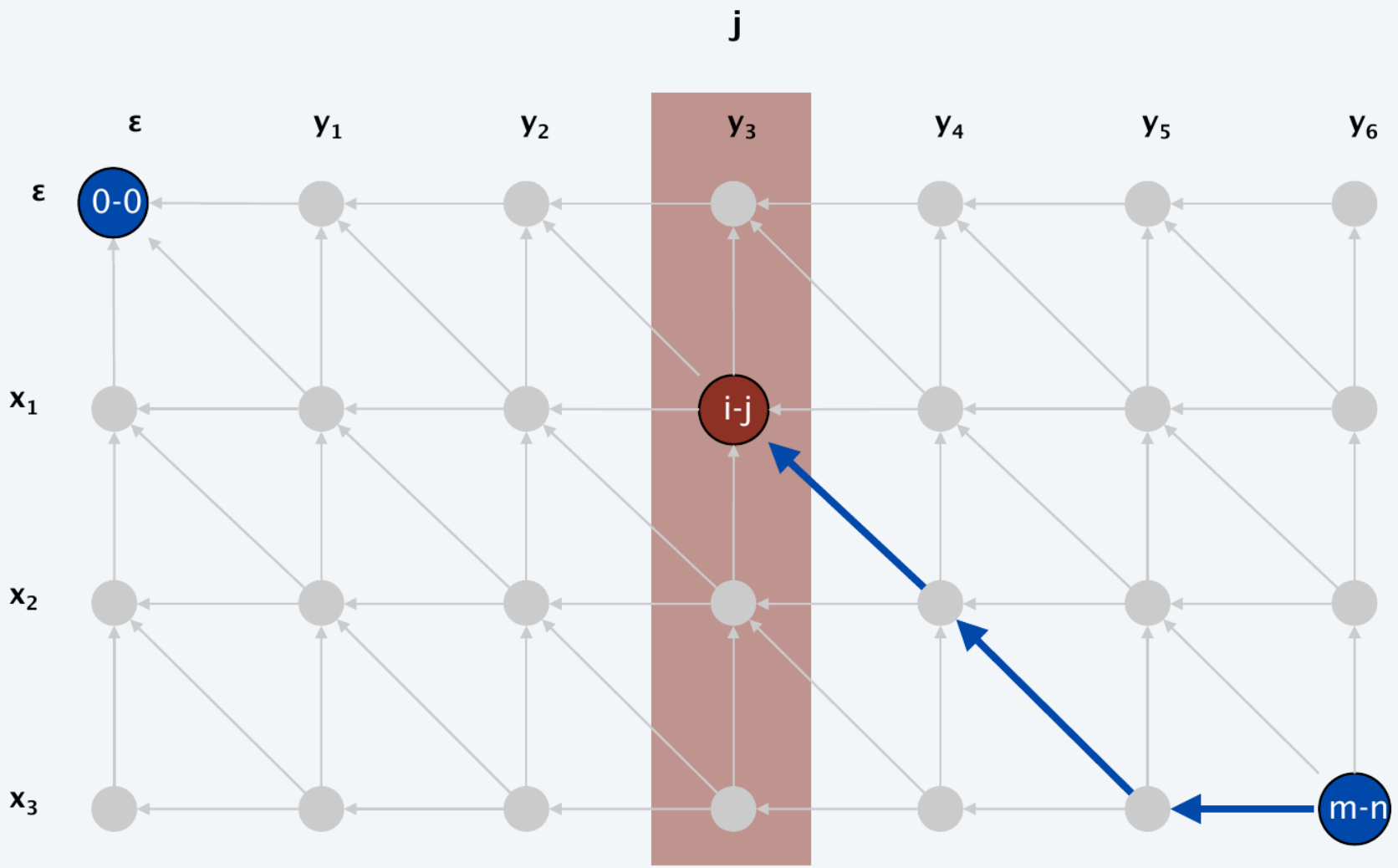
- Let $g(i, j)$ be shortest path from $(i, j)$ to $(m, n)$.
- Can compute by reversing the edge orientations and inverting the roles of $(0, 0)$ and $(m, n)$.



# Hirschberg's algorithm

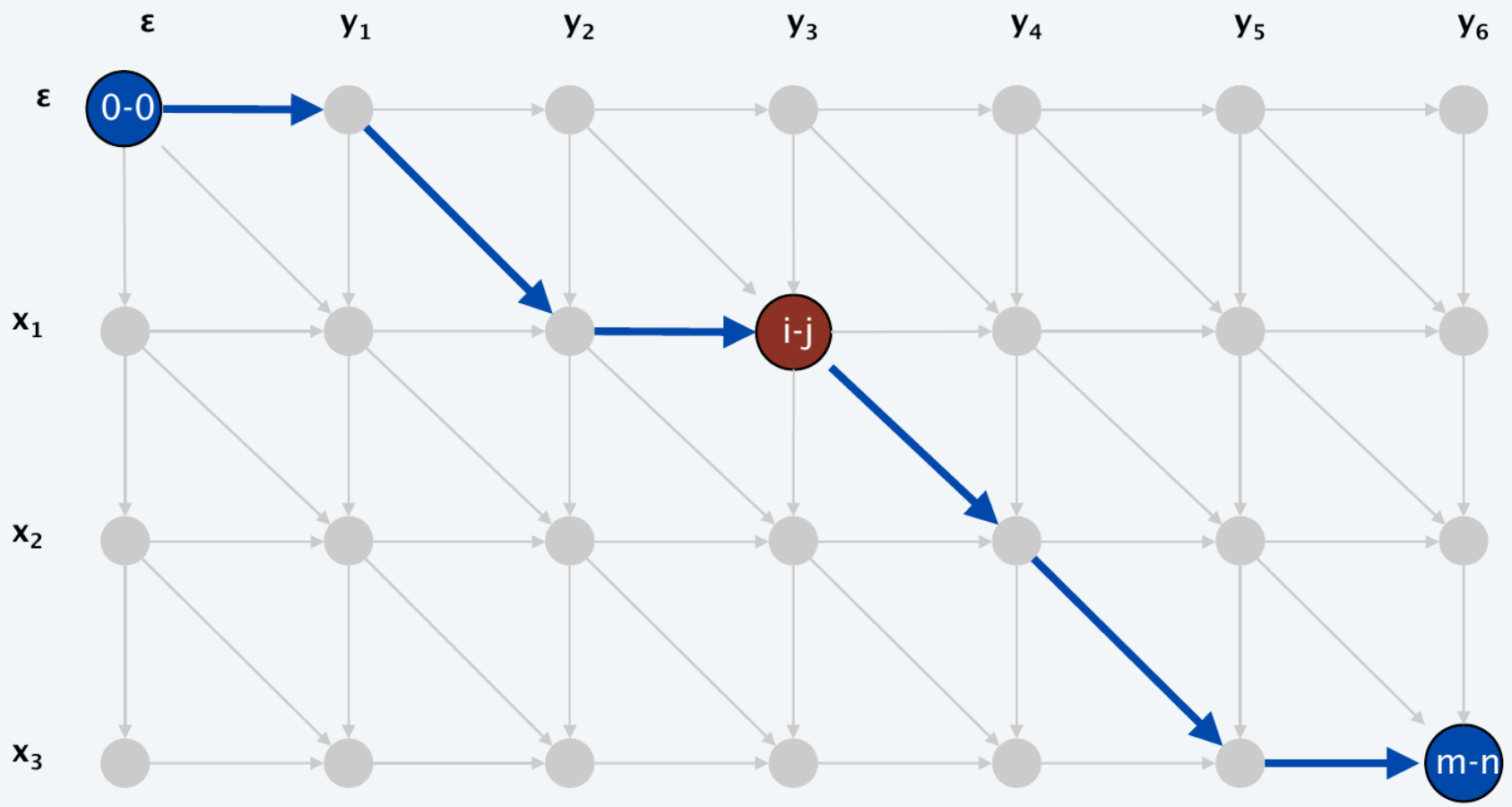
## Edit distance graph.

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



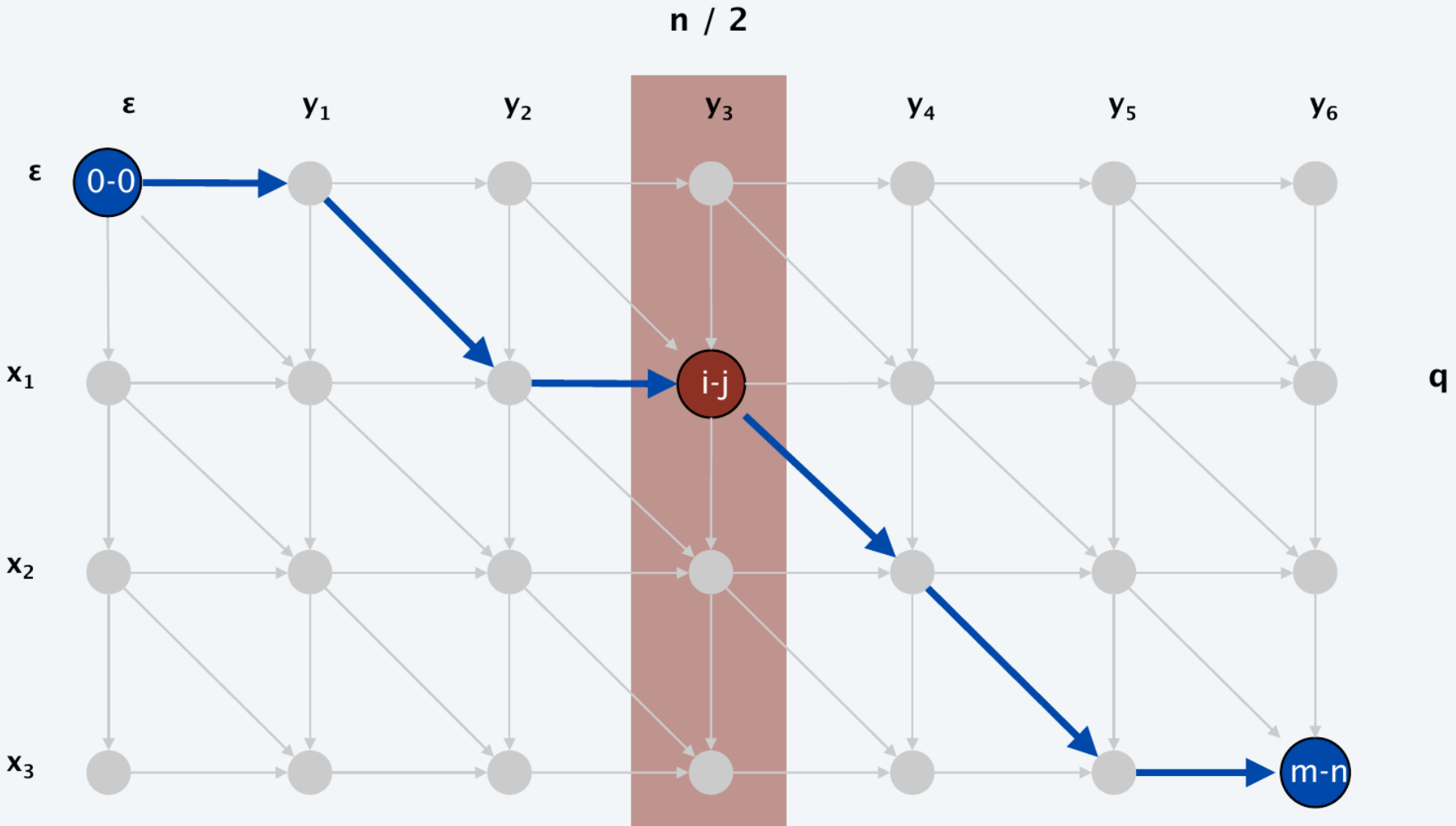
# Hirschberg's algorithm

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



# Hirschberg's algorithm

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



# Hirschberg's algorithm

Divide. Find index $q$ that minimizes $f(q, n/2) + g(q, n/2)$; align $x_q$ and $y_{n/2}$.

Conquer. Recursively compute optimal alignment in each piece.