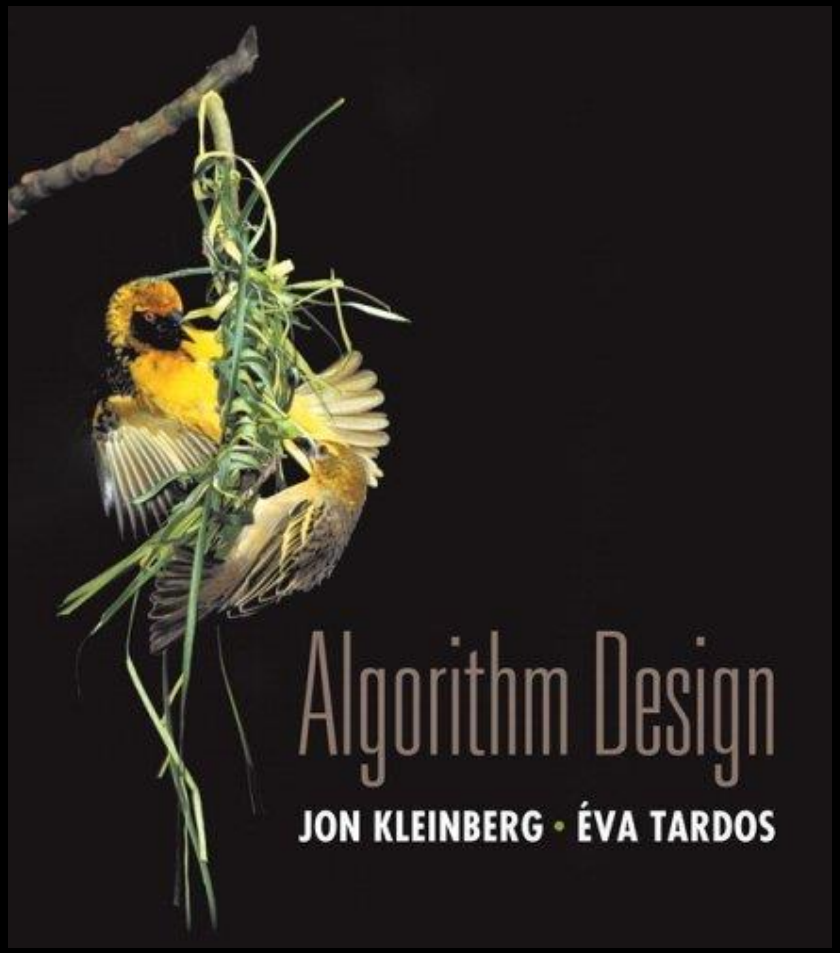


Chapter 6

Dynamic Programming

Slides by Kevin Wayne.

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.

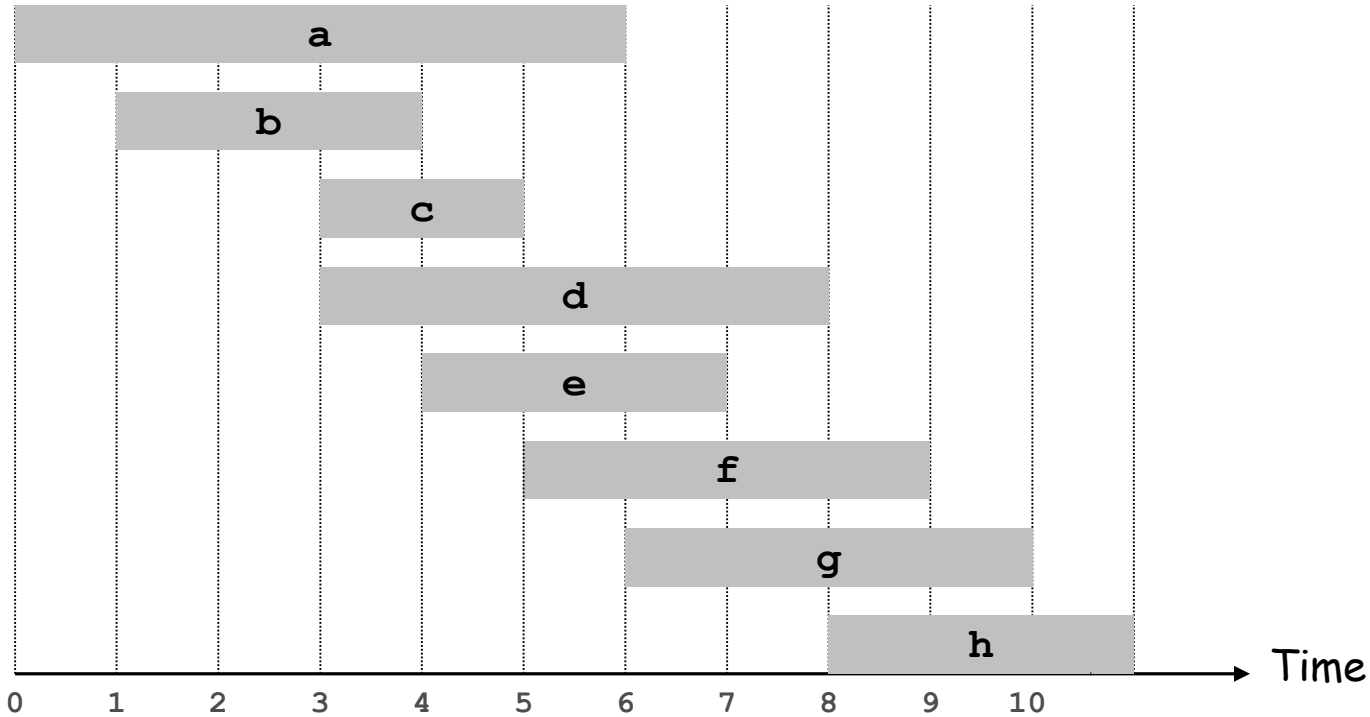
Memoing and tabling in languages such as Clojure and XSBProlog embody efficient computation inspired by dynamic programming

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.

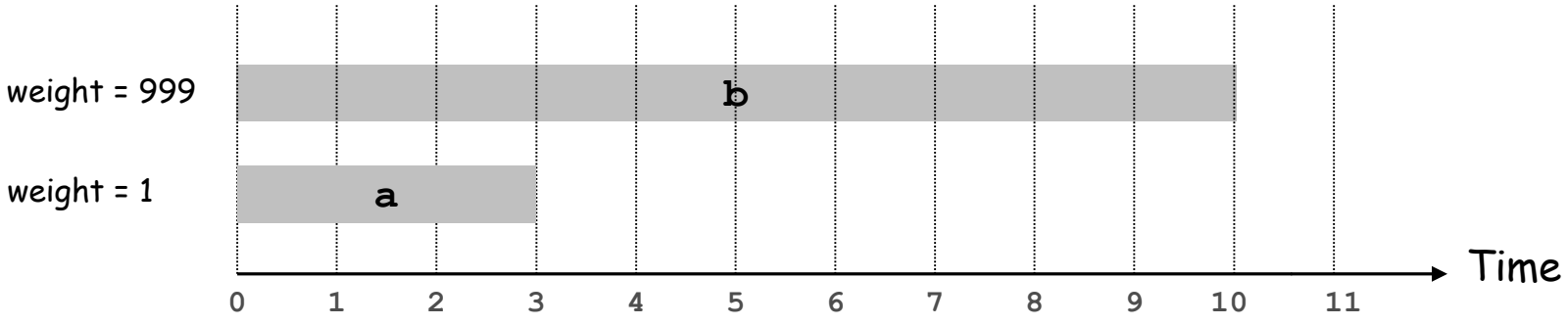


Unweighted Interval Scheduling Review

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.

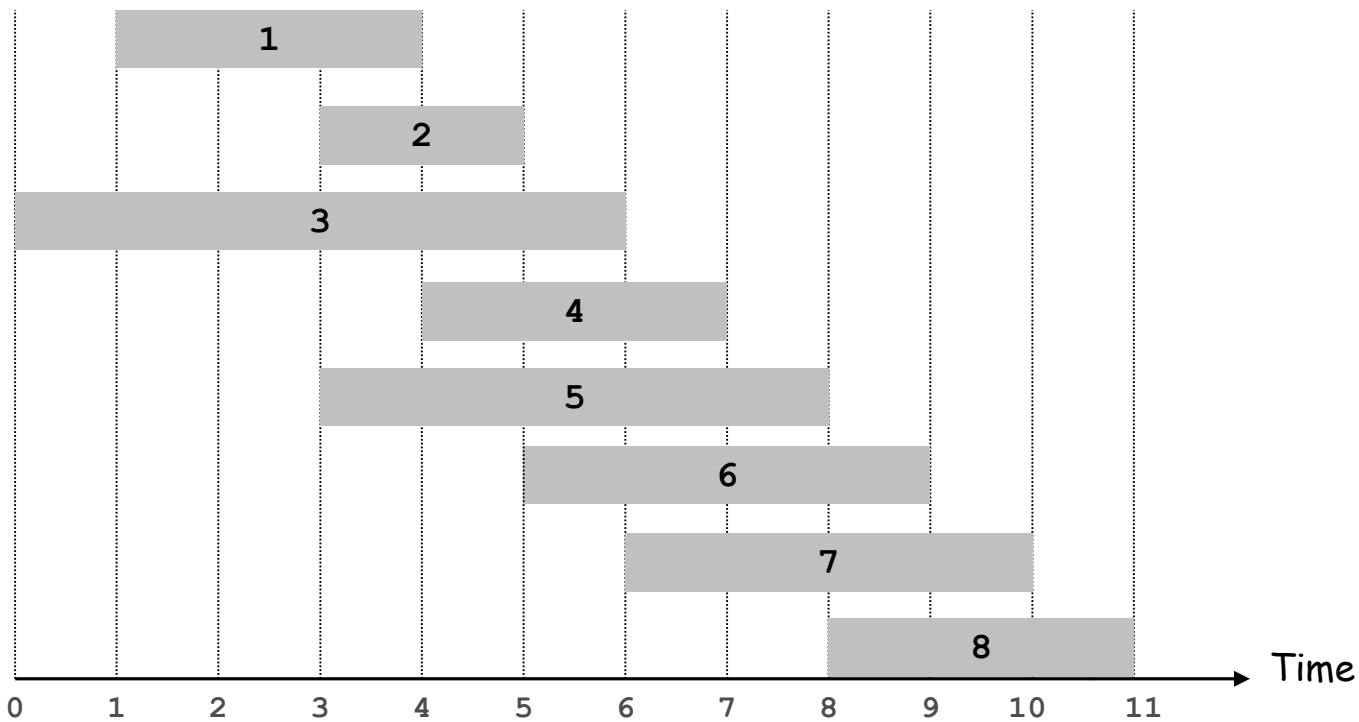


Weighted Interval Scheduling

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.



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.
  - collect profit v_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

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

Weighted Interval Scheduling: Brute Force

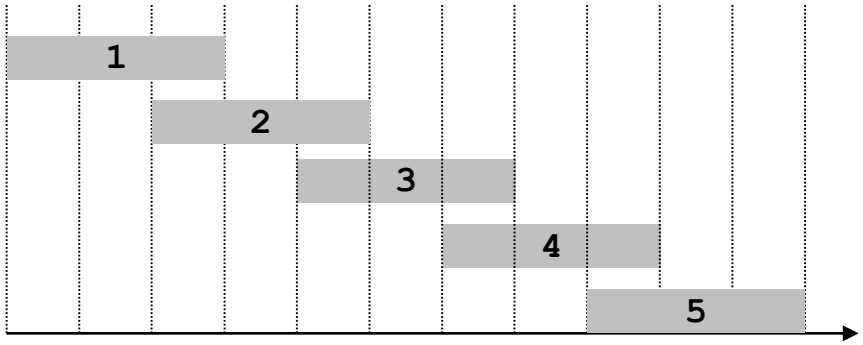
Brute force algorithm.

```
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))
}
```

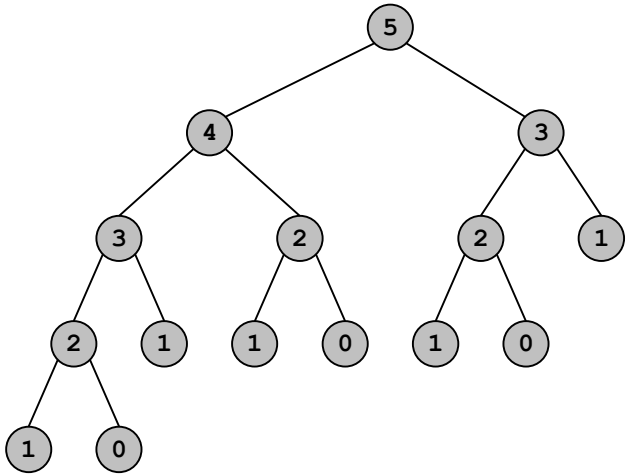
Weighted Interval Scheduling: Brute Force

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.



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



Weighted Interval Scheduling: Memoization

Memoization. Store results of each sub-problem in a cache; lookup (rather than recompute) 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
M[0] = 0

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

global array

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 Φ = # 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-Compute-Opt(n)` is $O(n)$. ▪

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

Weighted Interval Scheduling: Bottom-Up

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])
}
```

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 $\leq n \Rightarrow O(n)$.

Essence of Dynamic Programming

An efficient approach to generate optimal solution exploiting recursive definition

- Generate and test approach is intractable for combinatorial optimization problems.
- Recursive definitions prune the search space, laying the foundation for tractable solutions.
  - Top-down implementation of recursion
    - Naive approach to recursion: Exponential
    - Memoing or table-driven approach: Polynomial
  - Bottom-up implementation using iteration

    . Polynomial [Usually, linear, quadratic or cubic.]

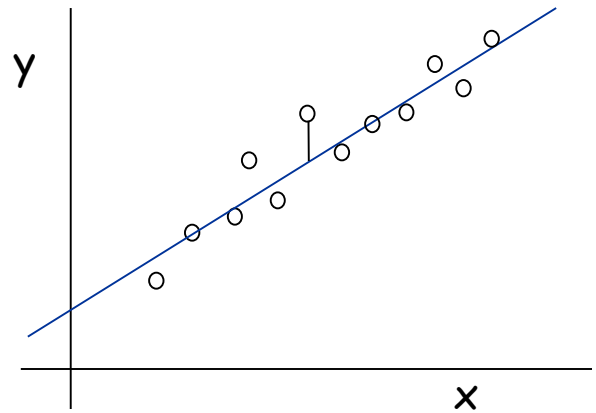
6.3 Segmented Least Squares

Segmented Least Squares

Least squares.

- Foundational problem in statistic and numerical analysis.
- Given n points in the plane: $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$.
- 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

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

$$\frac{d}{da} SSE = 0 \qquad \frac{d}{db} SSE = 0$$

$$\frac{d}{db}\left(\sum_{i=1}^{n} (y_i - a x_i - b)^2\right)$$

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

$$= 0$$

$$\boxed{\sum_{i=1}^{n} y_i - a \sum_{i=1}^{n} x_i - bn = 0} \quad (2)$$

$$n\sum_{i=1}^{n} x_i y_i - n a \sum_{i=1}^{n} x_i^2$$

$$= \sum_{i=1}^{n} x_i \sum_{i=1}^{n} y_i - a\left(\sum_{i=1}^{n} x_i\right)^2$$

$$\boxed{a = \frac{n\sum_{i=1}^{n} x_i y_i - \left(\sum_{i=1}^{n} x_i\right)\left(\sum_{i=1}^{n} y_i\right)}{n\sum_{i=1}^{n} x_i^2 - \left(\sum_{i=1}^{n} x_i\right)^2}}$$

[No Title]

$$\boxed{b = \frac{1}{n}\left(\sum_{i=1}^{n} y_i - a\sum_{i=1}^{n} x_i\right)}$$

Need to substitute a from previous slide to get b entirely in terms of points.

$$\frac{d}{da} SSE = 0 \qquad \frac{d}{db} SSE = 0$$

$$\frac{d}{da} SSE = \frac{d}{da}\left(\sum_{i=1}^{n} (y_i - a x_i - b)^2\right)$$

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

$$= 0$$

$$\boxed{\sum_{i=1}^{n} x_i y_i - a \sum_{i=1}^{n} x_i^2 - b \sum_{i=1}^{n} x_i = 0} \quad (1)$$

Linear Least Square Fit : Summary

: Calculate the slope of the line using the formula:

$$m = \frac{\sum xy - \frac{(\sum x)(\sum y)}{n}}{\sum x^2 - \frac{(\sum x)^2}{n}}$$

where n is the total number of data points.

: Compute the y-intercept of the line by using the formula:

$$b = \bar{y} - m\bar{x}$$

where $\bar{y}$ and $\bar{x}$ are the mean of the x- and y-coordinates of the data points

Segmented Least Squares

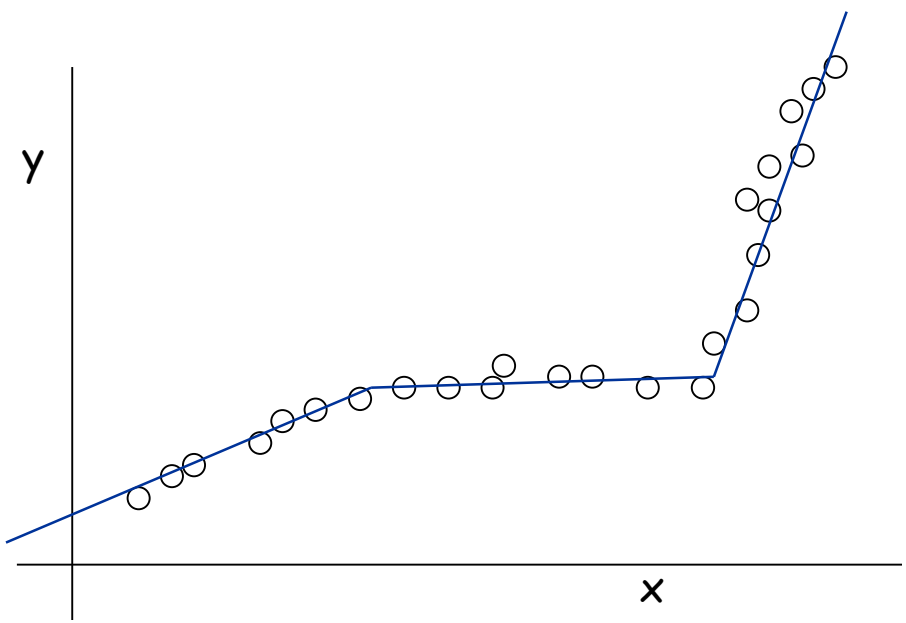
Segmented least squares.

- 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's a reasonable choice for f(x) to balance accuracy and parsimony?

goodness of fit

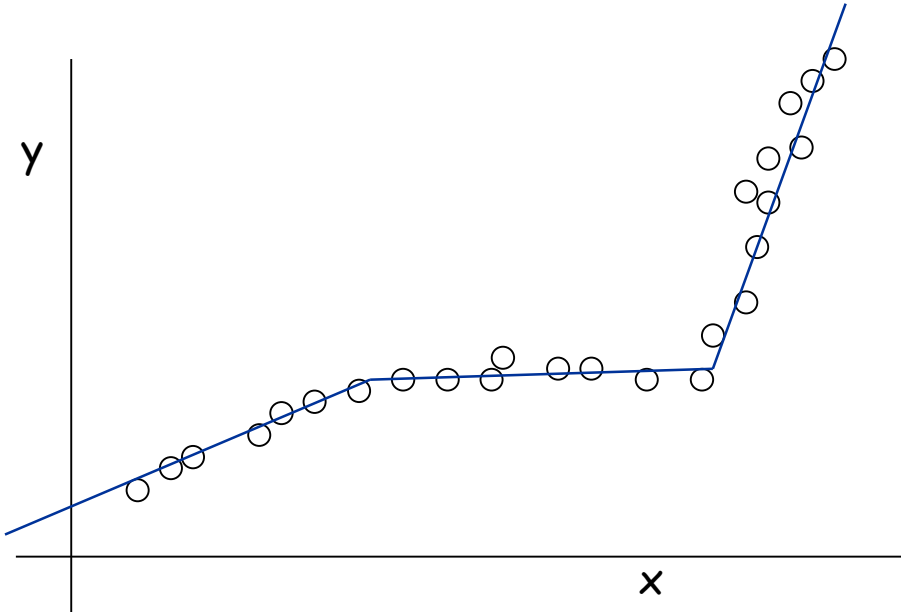
number of lines



Segmented Least Squares

Segmented least squares.

- Points lie roughly on a sequence of several line segments.
- Given n points in the plane $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ with
- $x_1 < x_2 < \ldots < x_n$, 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 $p_1, p_{i+1}, \ldots, p_j$.
- 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).

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

Segmented Least Squares: Algorithm

```
INPUT: n, p1,…,pN , c

Segmented-Least-Squares() {
   M[0] = 0
   for j = 1 to n
      for i = 1 to j
         compute the least square error eij for
         the segment pi,…, pj

   for j = 1 to n
      M[j] = min 1 ≤ i ≤ j (eij + 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: repeatedly add item with maximum ratio v_i / w_i.

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

Knapsack Problem : A Greedy Attempt

- 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.

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)

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

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

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

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 the greedy algorithms is 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.
  - 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 that make explicit weight constraint!

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

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

Knapsack Problem: Bottom-Up

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

```
Input: n, W, w1,…,wN, v1,…,vN

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

for i = 1 to n
   for w = 1 to W
      if (wi > w)
         M[i, w] = M[i-1, w]
      else
         M[i, w] = max {M[i-1, w], vi + M[i-1, w-wi ]}

return M[n, W]
```

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	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."
- Decision version of Knapsack is NP-complete. [Chapter 8]

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

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.

weights are integers between 1 and W

Pf.

- 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. [CHAPTER 8]
- There exists a poly-time algorithm that produces a feasible solution that has value within 1% of optimum. [SECTION 11.8]

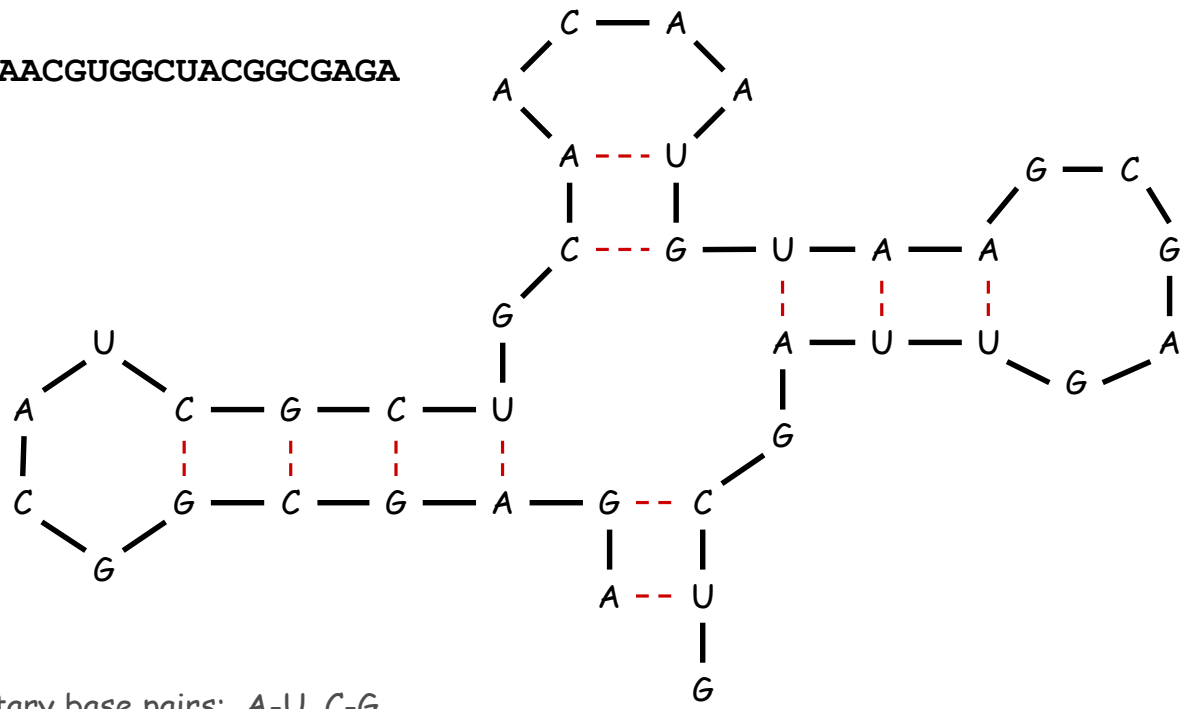
6.5 RNA Secondary Structure

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 $i < k < j < l$.

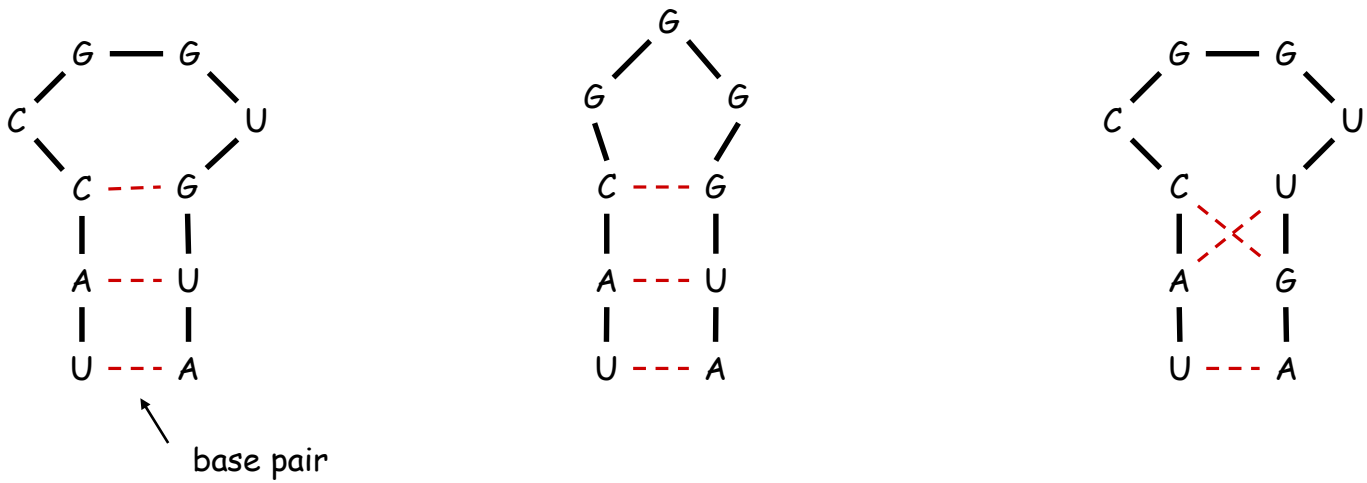
Free energy. Usual hypothesis is that an RNA molecule will form the secondary structure with the optimum total free energy.

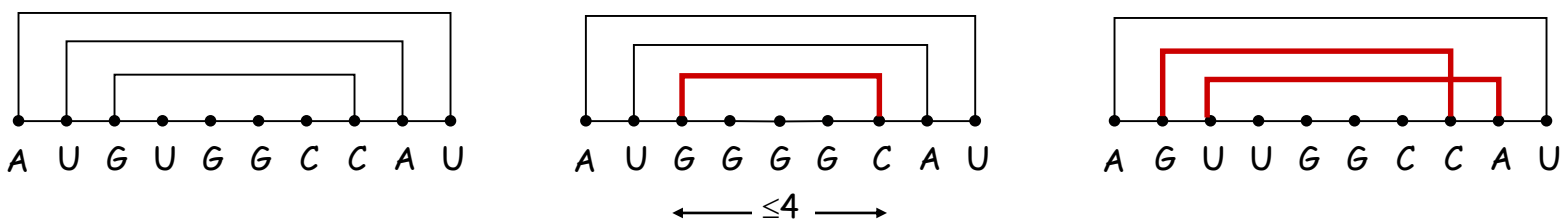
approximate by number of base pairs

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: Examples

Examples.

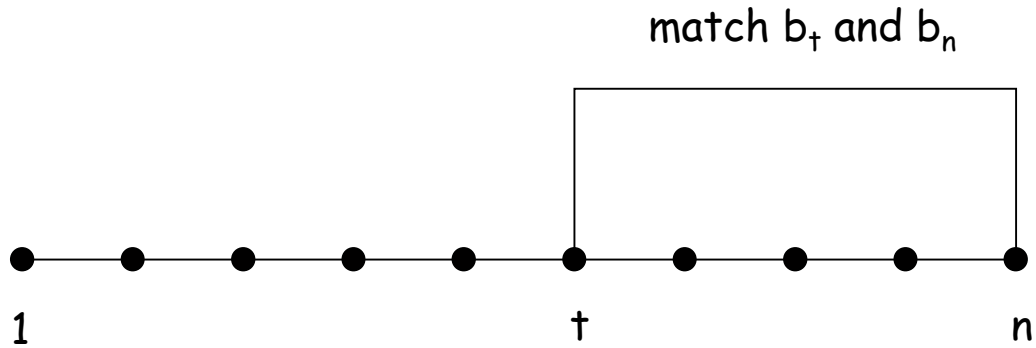




ok sharp turn crossing

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: $b_{t+1} b_{t+2} \ldots b_{n-1}$. ← need more sub-problems

Dynamic Programming Over Intervals

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

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

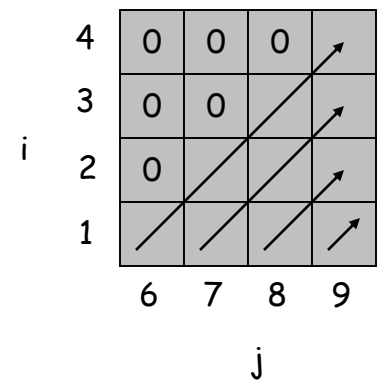
take max over t such that $i \leq t < j-4$ and b_t and b_j are Watson-Crick complements

Remark. Same core idea in CKY algorithm to parse context-free grammars.

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]
                          using recurrence
   return M[1, n]
}
```



Running time. The dynamic programming algorithm solves the RNA secondary substructure problem in $O(n3)$ time and $O(n2)$ space.

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

CKY parsing algorithm for context-free grammar has similar structure

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

6.6 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

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

$2\delta + \alpha_{CA}$

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.

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: $M = x_2\text{-}y_1,\ x_3\text{-}y_2,\ x_4\text{-}y_3,\ x_5\text{-}y_4,\ x_6\text{-}y_6$.

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

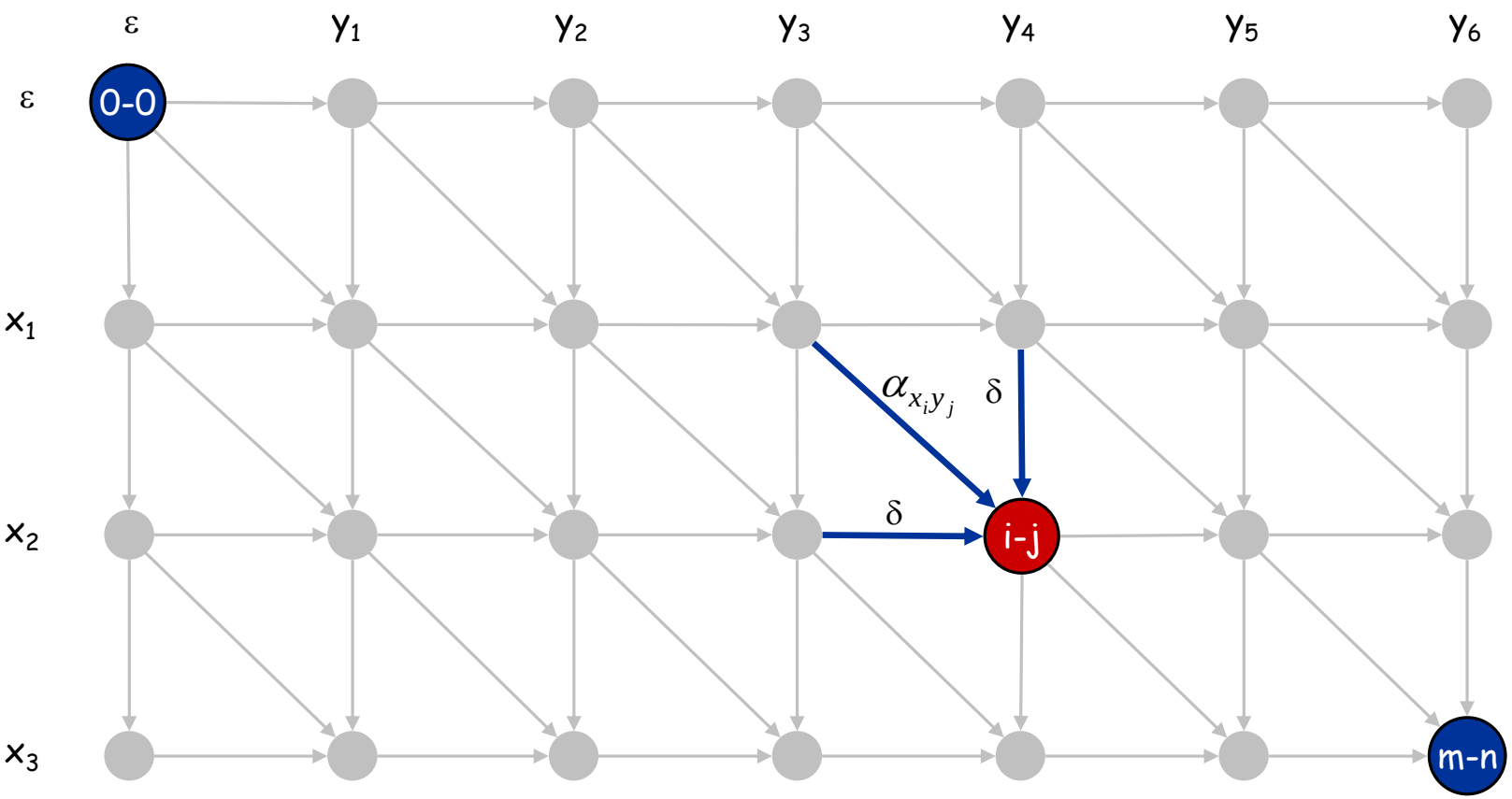
Def. OPT(i, j) = min cost of aligning strings $x_1\, x_2 \ldots x_i$ and $y_1\, y_2 \ldots y_j$.

- **Case 1:** OPT matches x_i-y_j.
  - pay mismatch for x_i-y_j + min cost of aligning 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 $y_1\, y_2 \ldots y_{j-1}$

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

Sequence Alignment: Linear Space

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



Sequence Alignment: Algorithm

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

		A	T	C	G	T	A	C
	0	1	2	3	4	5	6	7
	0	0	1	2	3	4	5	6	7
A	1	1							
T	2	2							
G	3	3							
T	4	4							
T	5	5							
A	6	6							
T	7	7							

α: 1 (mismatch) or 0 (match)

δ: 1

Initialization: 0th row:δj and 0th column:δi

Sequence Alignment Example

		A	T	C	G	T	A	C
	0	1	2	3	4	5	6	7
	0	0	1	2	3	4	5	6	7
A	1	1	0	1	2	3	4	5	6
T	2	2	1	0	1	2	3	4	5
G	3	3	2	1	1	1	2	3	4
T	4	4	3	2	2	2	1	2	3
T	5	5	4	3	3	3	2	2	3
A	6	6	5	4	4	4	3	2	3
T	7	7	6	5	5	5	4	3	3

α: 1 (mismatch) or 0 (match)

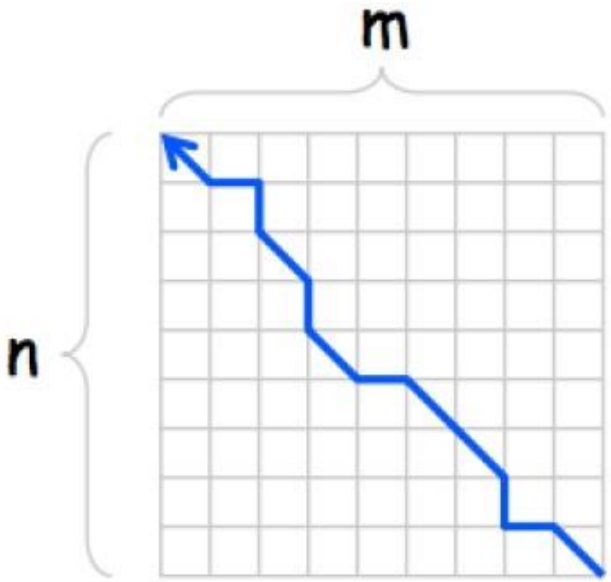
δ: 1

the rest of the matrix:

```
for i = 1 to m
 for j = 1 to n
  M[i,j]=min(α[x_i, y_j]+M[i-1,j-1],
             δ + M[i-1, j],
             δ + M[i, j-1])
```

Constructing the alignment

Backtrack from m,n



Correct: (m,n) => (n,m)

Sequence Alignment Example

		A	T	C	G	T	A	C
	0	1	2	3	4	5	6	7
	0	1	2	3	4	5	6	7
A 1	1	0	1	2	3	4	5	6
T 2	2	1	0	1	2	3	4	5
G 3	3	2	1	1	1	2	3	4
T 4	4	3	2	2	2	1	2	3
T 5	5	4	3	3	3	2	2	3
A 6	6	5	4	4	4	3	2	3
T 7	7	6	5	5	5	4	3	3

Constructing an Alignment

Backtrack from m,n

ATCGTAC
ATGTTAT

IS IT UNIQUE?

Sequence Alignment Example

		A	T	C	G	T	A	C
	0	1	2	3	4	5	6	7
A 1	1	0	1	2	3	4	5	6
T 2	2	1	0	1	2	3	4	5
G 3	3	2	1	1	1	2	3	4
T 4	4	3	2	2	2	1	2	3
T 5	5	4	3	3	3	2	2	3
A 6	6	5	4	4	4	3	2	3
T 7	7	6	5	5	5	4	3	3

Constructing an Alignment

Backtrack from m,n

ATCGTAC
ATGTTAT

IS IT UNIQUE? NO:

ATCG-TAC
AT-GTTAT

A human protein aligned to a fruit fly protein

PAX6_HUMAN aligned against PAX6_DRO

```
5   HSGVNQLGGVFVNGRPLPDSTRQKIVELAHSGARPCDISRILQVSNGCVS    54
    |||||||||||||.||||||||||||||||||||||||||||||||||||
57  HSGVNQLGGVFVGGRPLPDSTRQKIVELAHSGARPCDISRILQVSNGCVS   106

55  KILGRYYETGSIRPRAIGGSKPRVATPEVVSKIAQYKRECPSIFAWEIRD   104
    |||||||||||||||||||||||||||.||||||:||||||||||||||||
107 KILGRYYETGSIRPRAIGGSKPRVATAEVVSKISQYKRECPSIFAWEIRD   156

105 RLLSEGVCTNDNIPSVSSINRVLRNLASEKQQMGA---------------   139
    |||.|.||||||||||||||||||||||::|:|...
157 RLLQENVCTNDNIPSVSSINRVLRNLAAQKEQQSTGSGSSSTSAGNSISA   206

155 -----------SWGTR---PGWYPGTSVPGQPTQ----------------   174
               ||..|   ..||| ||:...|..
307 NHQALQQHQQQSWPPRHYSGSWYP-TSLSEIPISSAPNIASVTAYASGPS   355

175 ------------------------------------DGCQQQE---GGGE   185
                                        ||.|..|   |.||
356 LAHSLSPPNDIESLASIGHQRNCPVATEDIHLKKELDGHQSDETGSGEGE   405

186 NTNSISSNGEDSDEAQMRLQLKRKLQRNRTSFTQEQIEALEKEFERTHYP   235
    |:|..:||..:::.|.||.||||||||||||.:||::||||||||||
406 NSNGGASNIGNTEDDQARLILKRKLQRNRTSFTNDQIDSLEKEFERTHYP   455
```

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.

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

Computational biology: m = n = 100,000.

10 billions ops OK, but 10GB array?

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.

6.7 Sequence Alignment in Linear Space

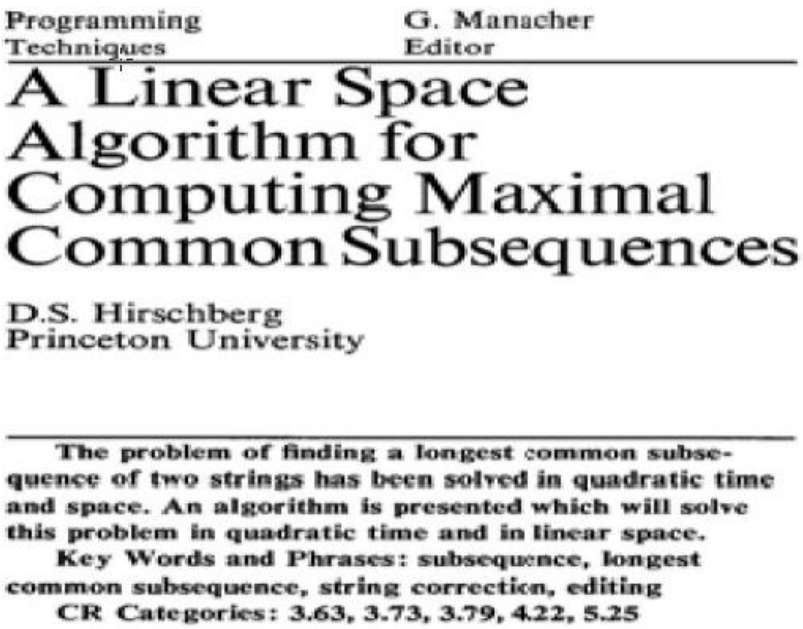
Sequence Alignment: Linear Space

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

- Compute $OPT(i, \cdot)$ from $OPT(i-1, \cdot)$.
- 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.
- Inspired by idea of Savitch from complexity theory.



Programming Techniques

G. Manacher Editor

A Linear Space Algorithm for Computing Maximal Common Subsequences

D.S. Hirschberg
Princeton University

The problem of finding a longest common subsequence of two strings has been solved in quadratic time and space. An algorithm is presented which will solve this problem in quadratic time and in linear space.

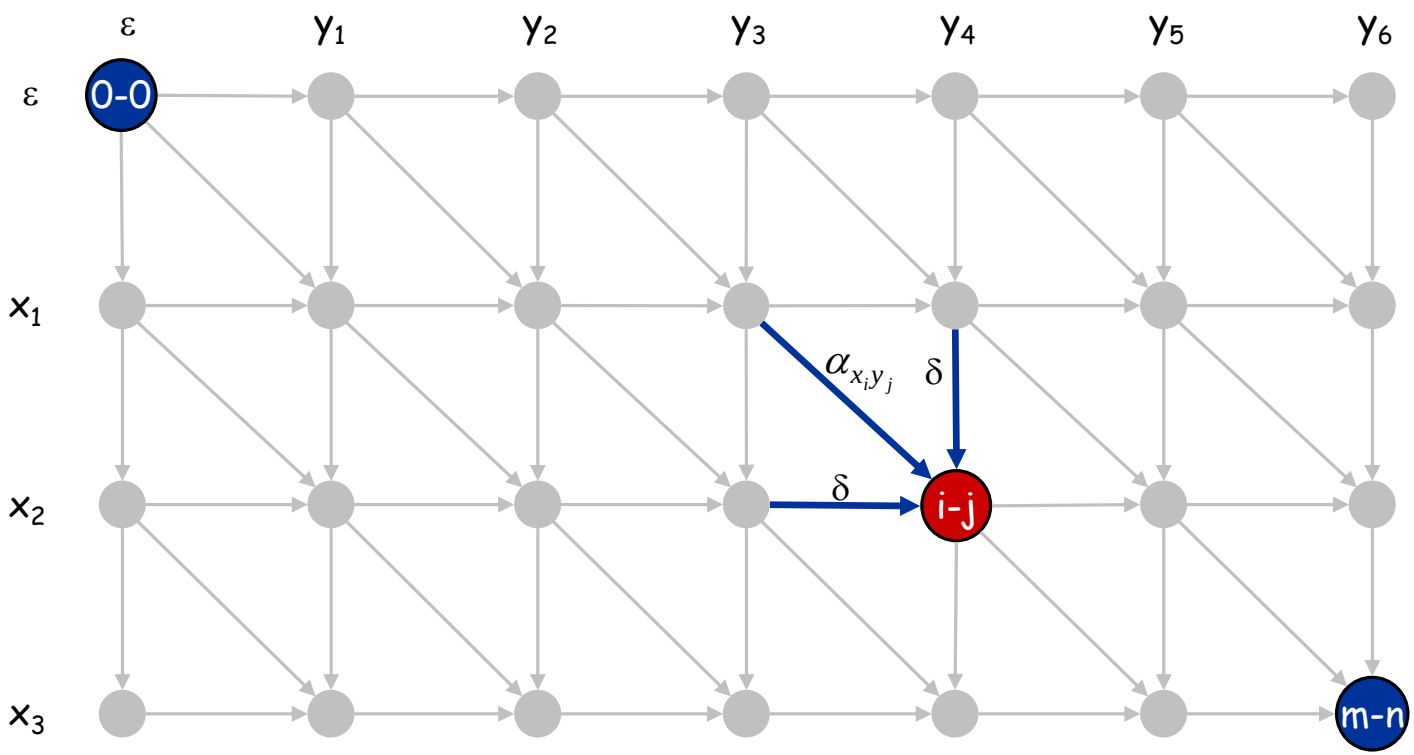
Key Words and Phrases: subsequence, longest common subsequence, string correction, editing

CR Categories: 3.63, 3.73, 3.79, 4.22, 5.25

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).



Hirschberg's algorithm

Edit distance graph.

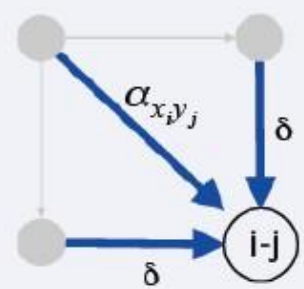
- Let $f(i,j)$ be shortest path from (0,0) to (i,j).
- Lemma: $f(i,j) = OPT(i,j)$ for all i and j.

Should also include
f(0,i) = i and f(0,j) = 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,

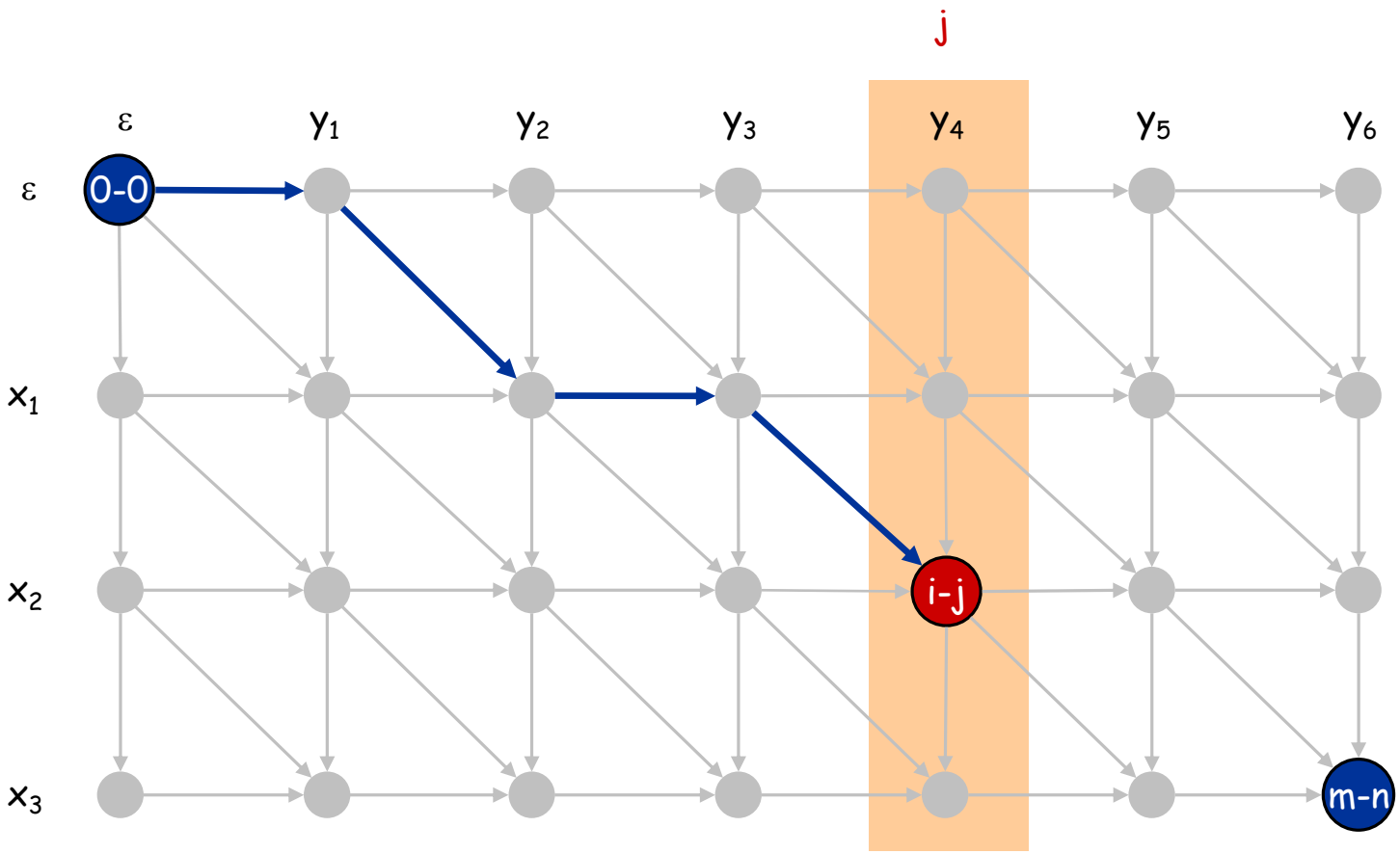
$$
\begin{aligned}
f(i,j) &= \min\{\alpha_{x_i y_j} + f(i-1,j-1),\ \delta + f(i-1,j),\ \delta + f(i,j-1)\} \\
&= \min\{\alpha_{x_i y_j} + OPT(i-1,j-1),\ \delta + OPT(i-1,j),\ \delta + OPT(i,j-1)\} \\
&= OPT(i,j) \ \blacksquare
\end{aligned}
$$



Sequence Alignment: Linear Space

Edit distance graph.

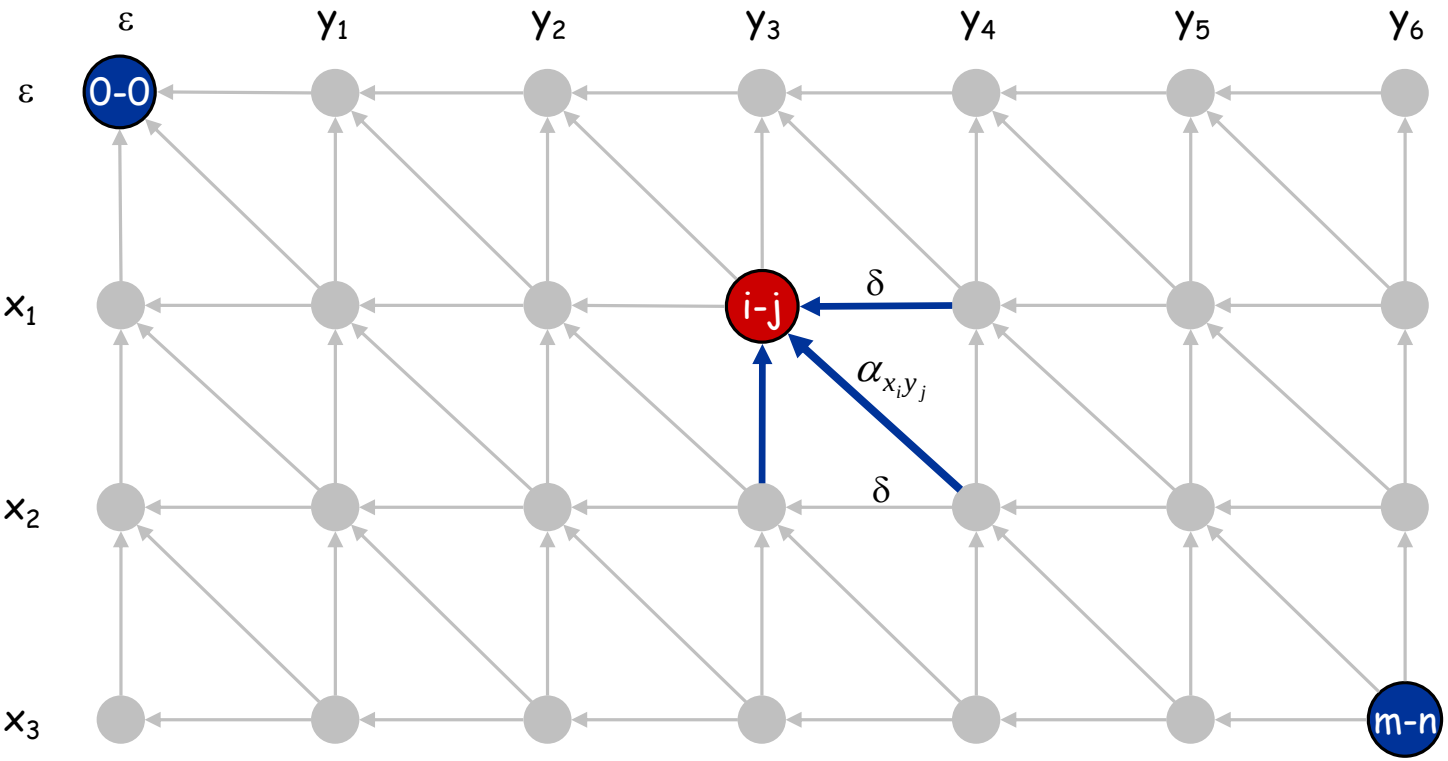
- Let f(i, j) be shortest path from (0,0) to (i, j).
- Can compute f (•, j) for any j in O(mn) time and O(m + n) space.



Sequence Alignment: Linear Space

Edit distance graph.

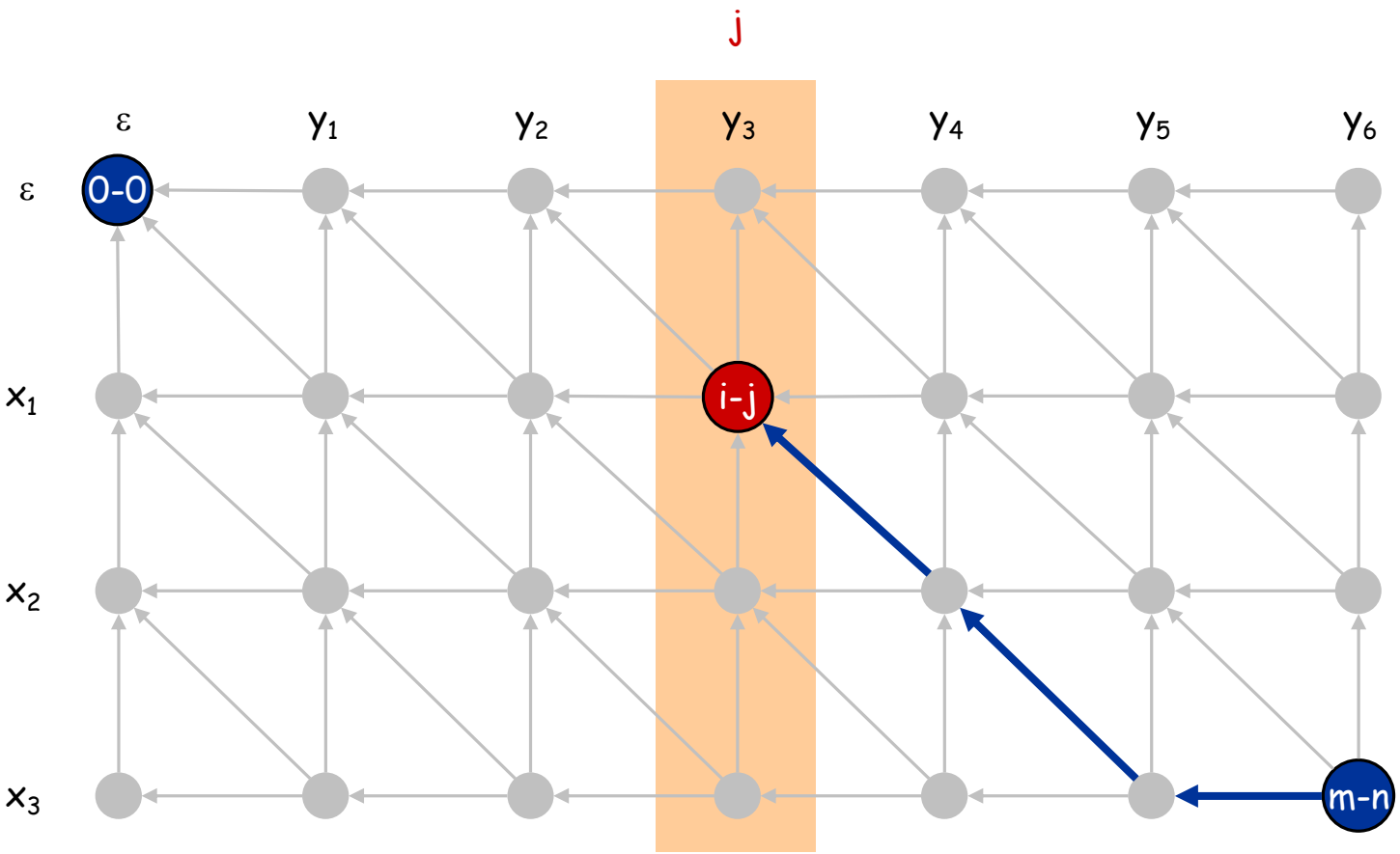
- Let g(i, j) be shortest path from (i, j) to (m, n).
- Can compute by reversing the edge orientations and inverting the roles of (0, 0) and (m, n)



Sequence Alignment: Linear Space

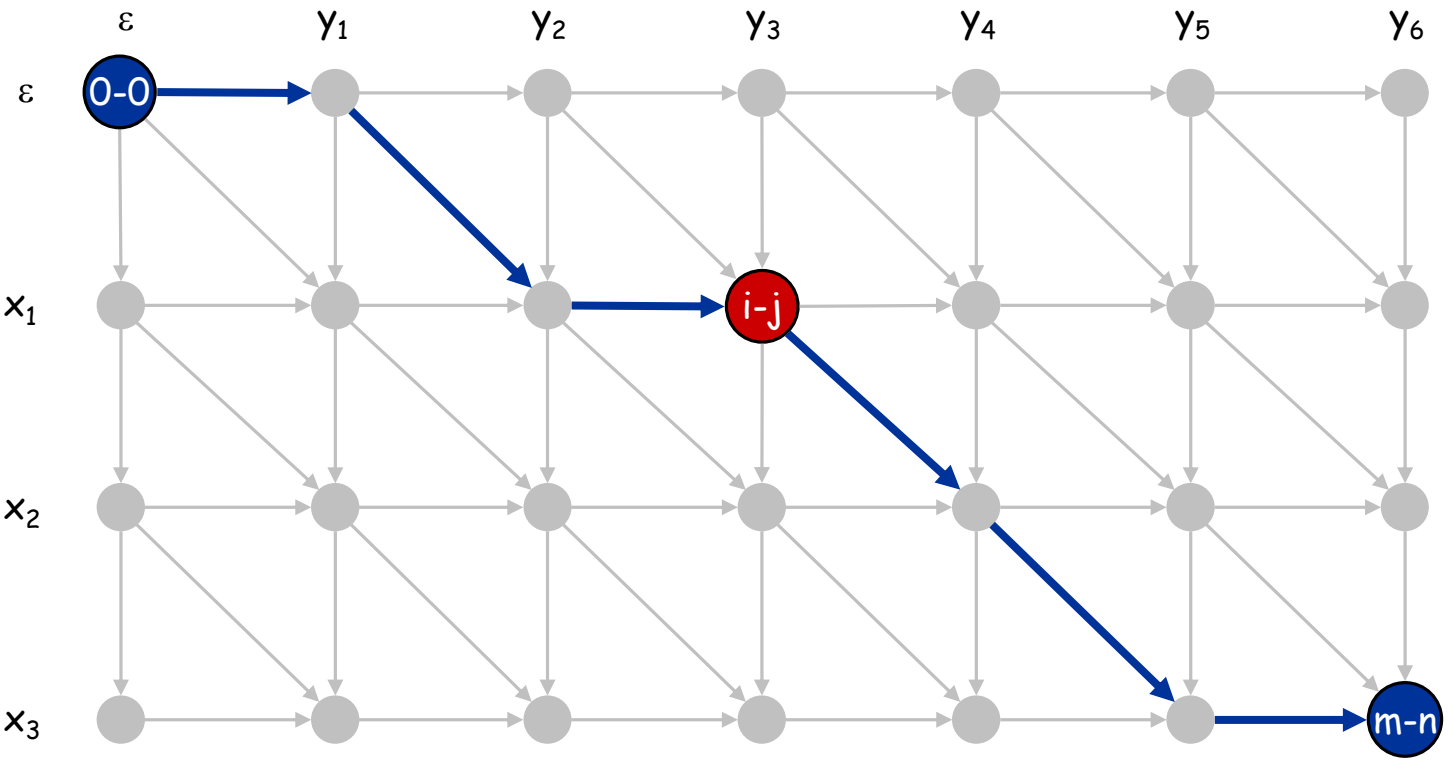
Edit distance graph.

- 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.

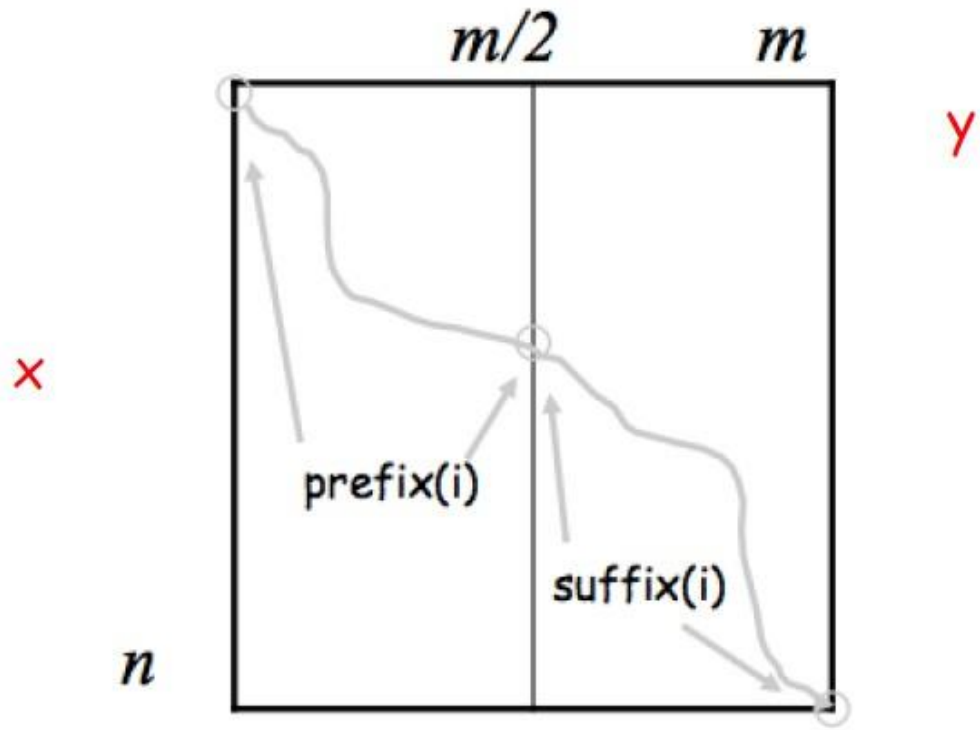


Sequence Alignment: Linear Space

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



Crossing the Middle Line



Correct: (m,n) => (n,m)

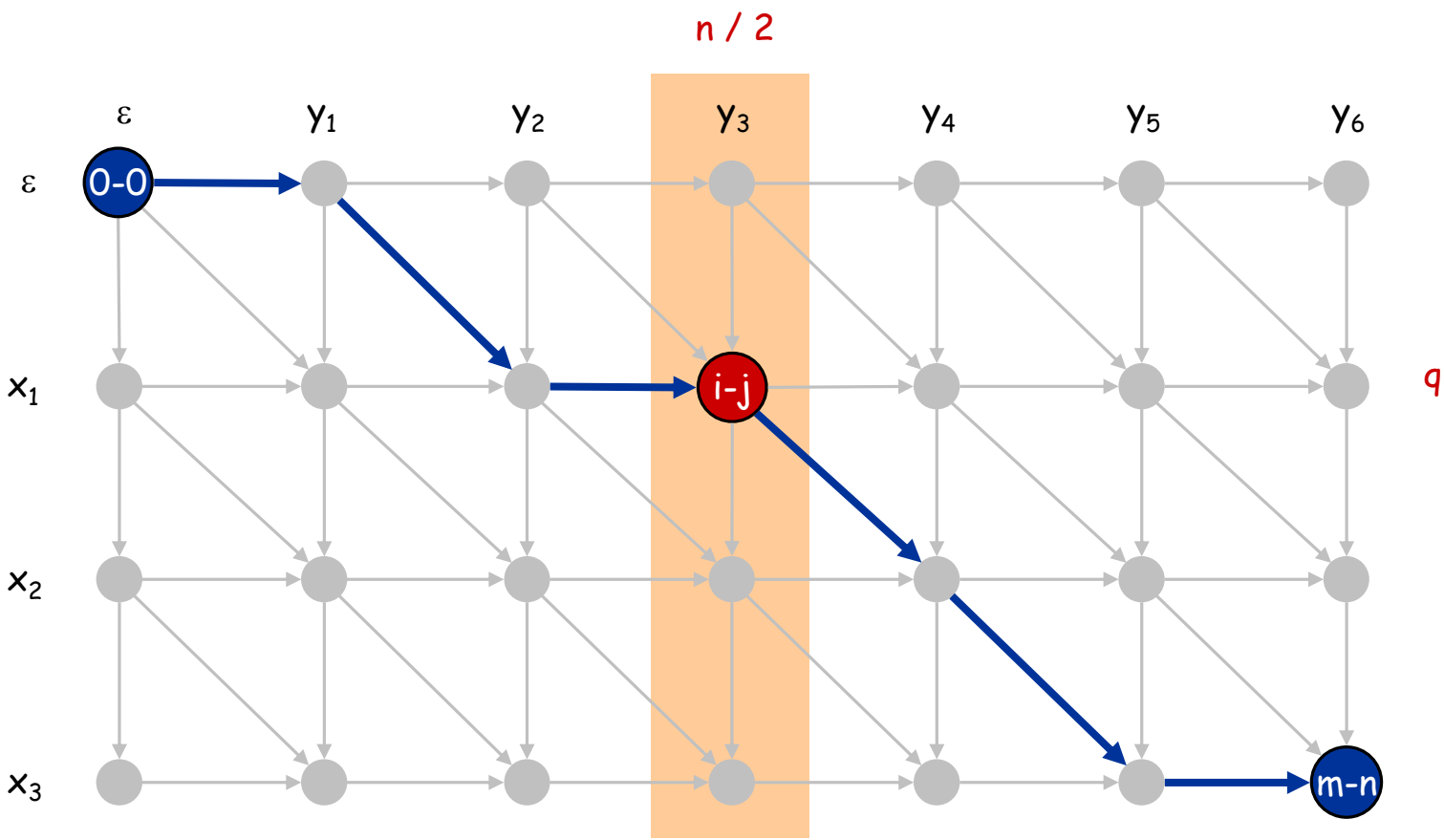
score(*i*) = prefix(*i*) + suffix(*i*)

prefix(*i*): score of the optimal alignment of a length m/2 prefix of y to a prefix of x (takes a path from (0,0) to (*i*,*m*/2))

suffix(i): score of the optimal alignment of a length m/2 suffix of y to a suffix of x (takes a path from (*i*,*m*/2) to *(n,m)*)

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).

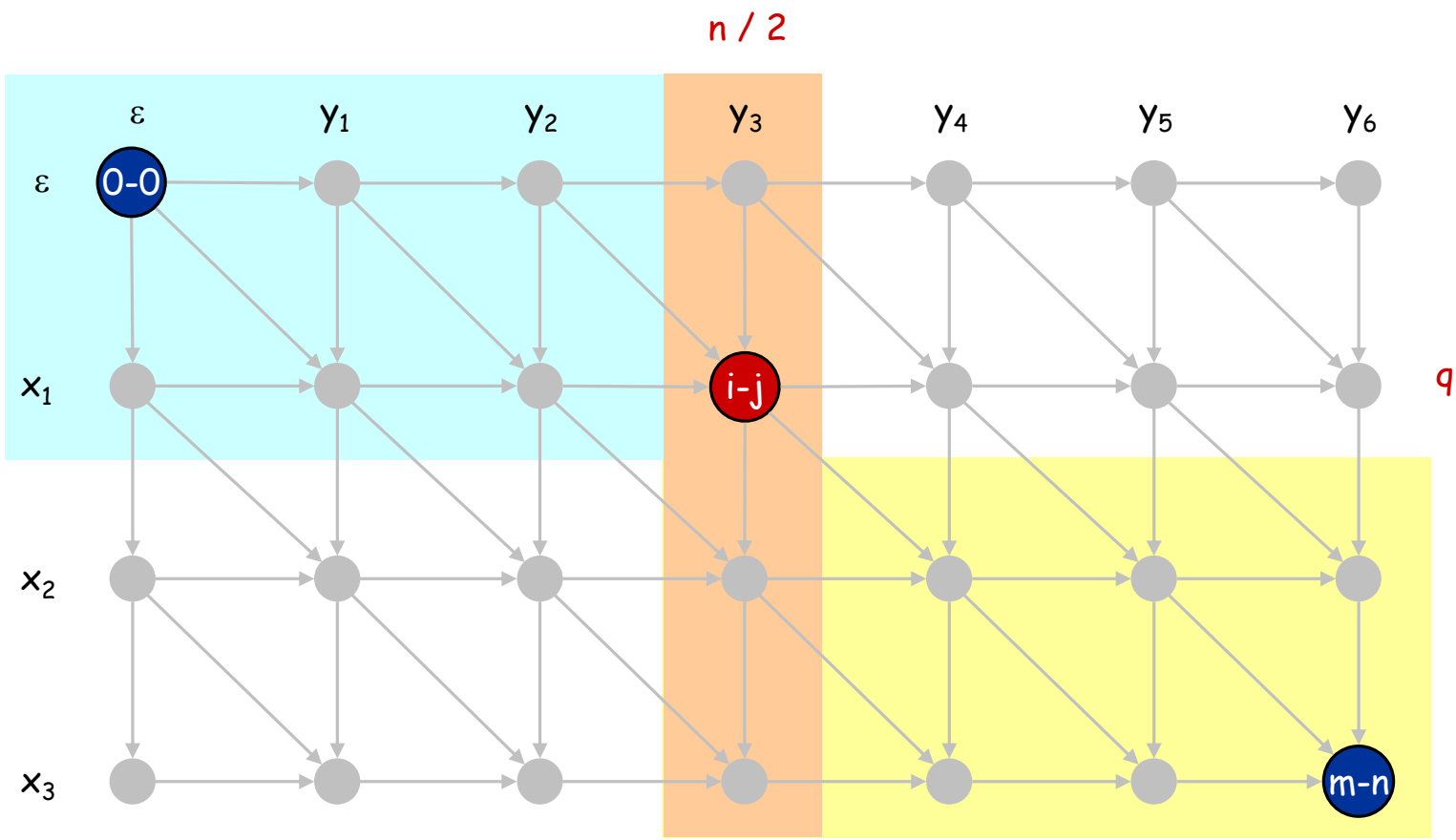


Sequence Alignment: Linear Space

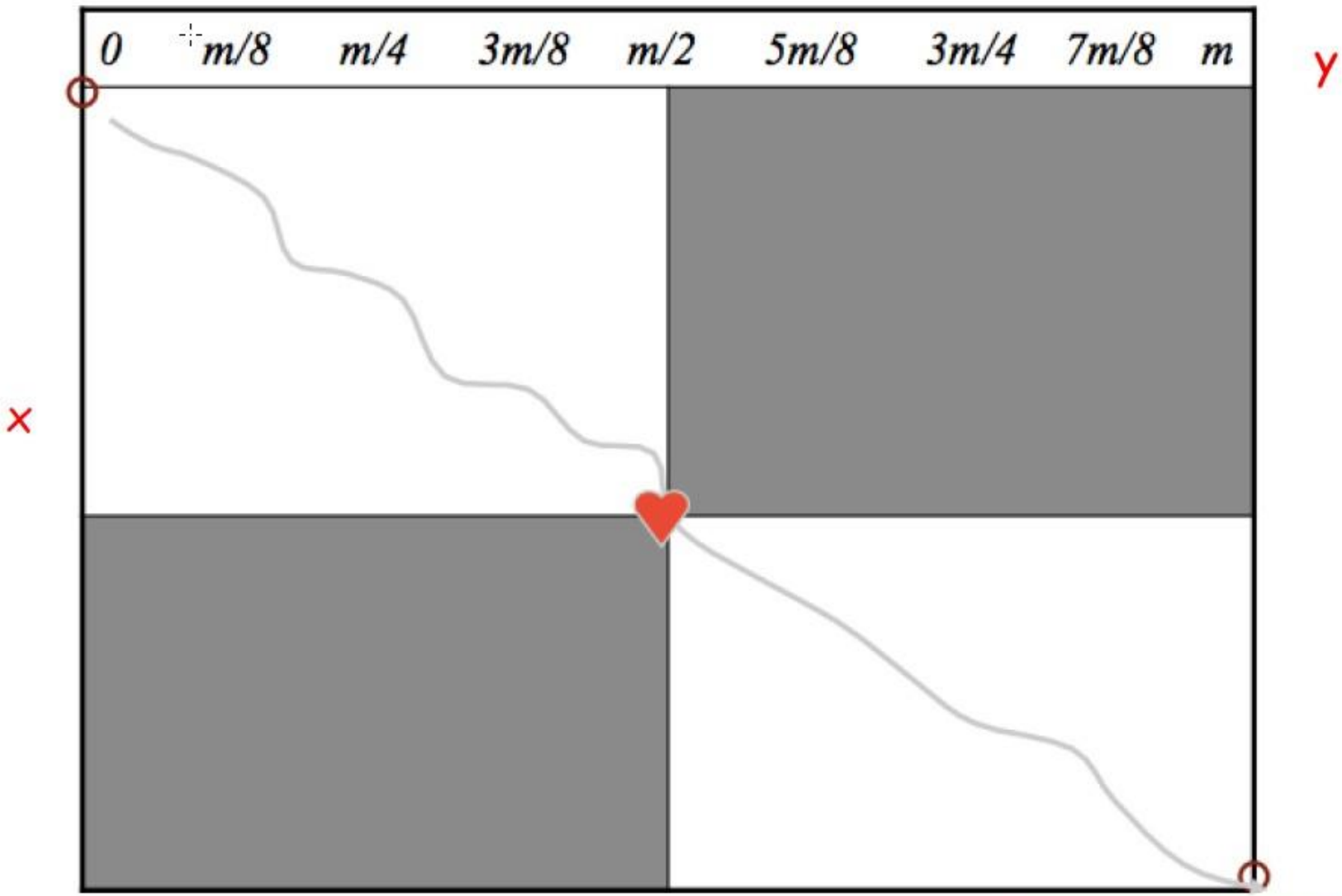
Divide: find index q that minimizes f(q, n/2) + g(q, n/2) using DP.

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

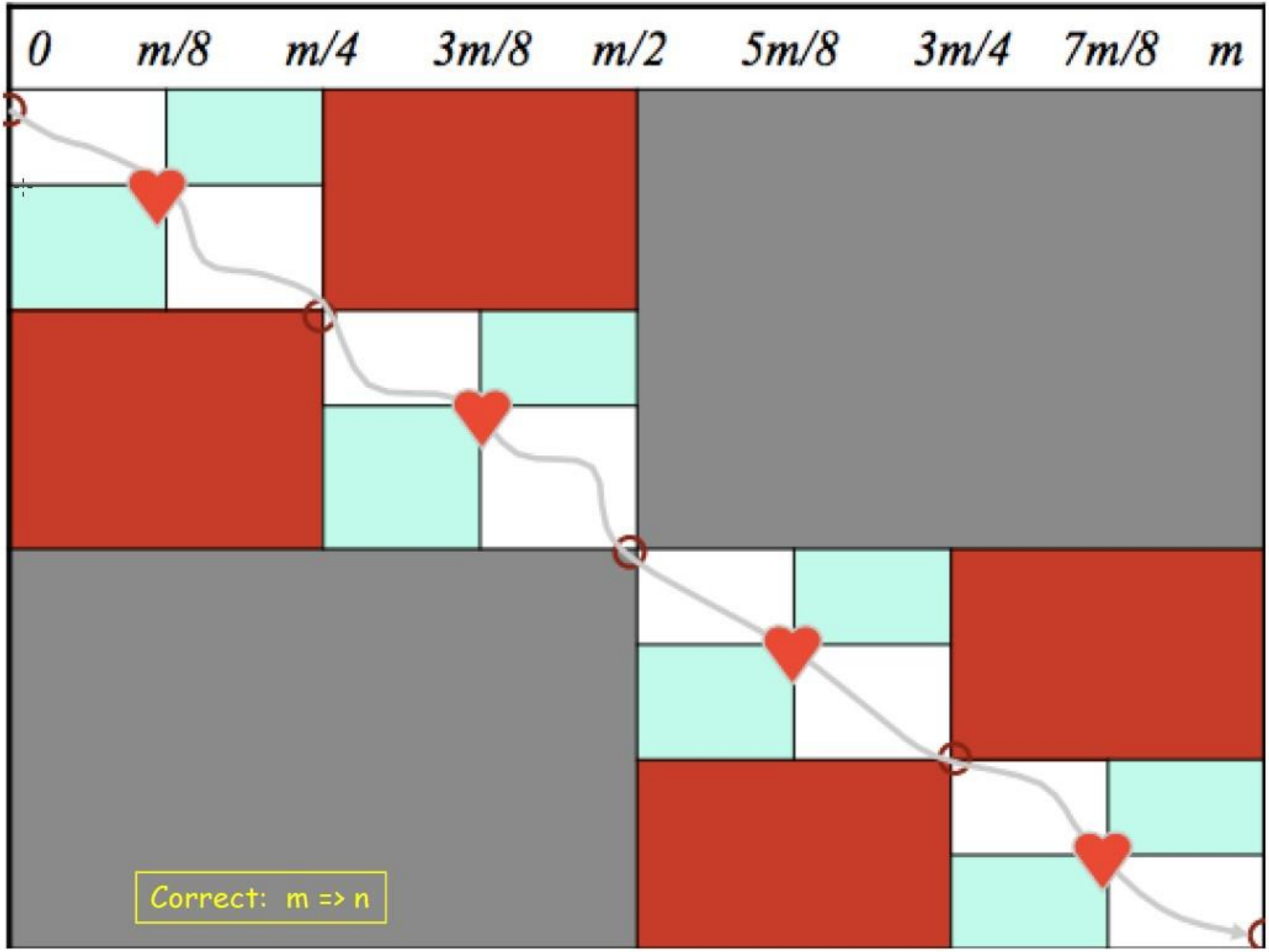
Conquer: recursively compute optimal alignment in each piece.



Finding the Middle Point



Correct: m => n


0
m/8
m/4
3m/8
m/2
5m/8
3m/4
7m/8
m
Correct: m => n

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. Fortunately, the analysis is not tight because two sub-problems are of size (q, n/2) and (m - q, n/2). In next slide, we shave off the **log n** factor.

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:

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

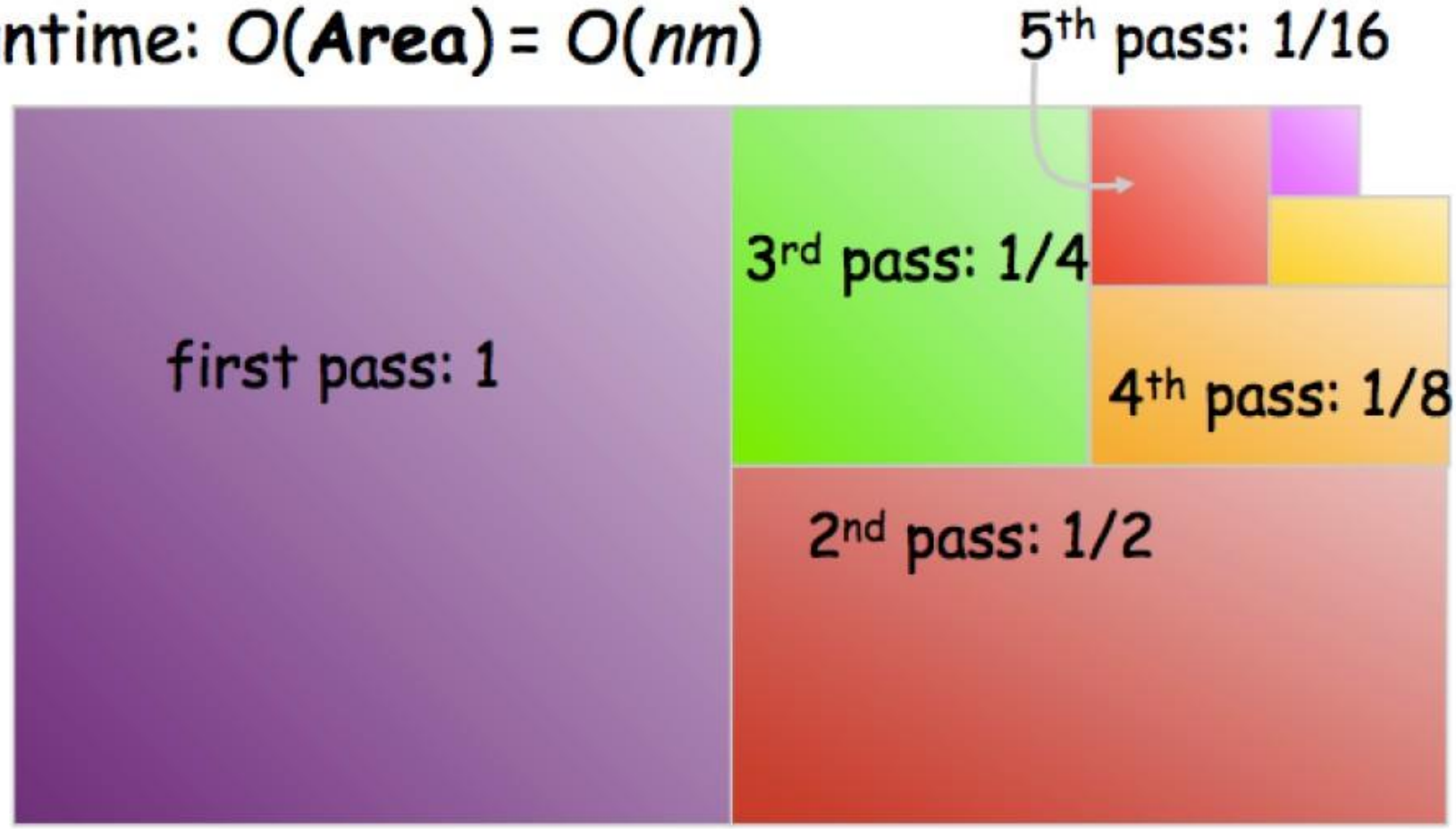
- Base cases: m = 2 or n = 2.
- Inductive hypothesis: T(m, n) $\leq$ 2cmn.

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

Geometric Reduction At Each Iteration

$1 + ½ + ¼ + \ldots + (½)^k \leq 2$

- Runtime: $O(\textbf{Area}) = O(nm)$



Overall Summary of Match(x[1,m], y[1,n])

- Match half-prefix of y with all prefixes of x.
  - Compute $f(., \frac{1}{2}n)$
- Match half-suffix of y with all suffixes of x.
  - Compute $g(., \frac{1}{2}n)$
- Both these together require O(mn) time and O(m+n) space.
- Find optimal value of j (call it q) that minimizes

  $f(j, \frac{1}{2}n) + g(j, \frac{1}{2}n)$. [say: x_q paired with $y_{\frac{1}{2}n}$]
- Now solve sub-problems
  - Match(x[1,q-1],y[1, $\frac{1}{2}$n]) and Match(x[q+1,m],y[$\frac{1}{2}$n+1,n])

$$T(m,n) = T(q, \tfrac{1}{2}n) + T(m - q, \tfrac{1}{2}n) + O(mn)$$

- Cumulative complexity with recursion still requires O(mn) time (reusing O(m+n) space.)