

Dynamic Programming

Algorithmic Paradigms

- **Greedy**. Build up a solution incrementally, myopically optimizing some local criterion.
- **Divide-and-Conquer**. Break up a problem into non-overlapping (independent) sub-problems, solve each sub-problem, and then combine these solutions of sub-problems to form the solution to original problem.
- **Dynamic Programming**. Break up a problem into a series of **overlapping** sub-problems, and then combine solutions of smaller sub-problems to form the solution to a larger sub-problem.

Dynamic Programming History

- **Bellman**. Pioneered the systematic study of dynamic programming in the 1950s.
- **Etymology**: It is the history of words, their origins, and how their form and *meaning* have changed over time. The *etymology* of [a word]" *means* the origin of the particular word.
 - Dynamic Programming = Planning over time.
 - Secretary of Defense was hostile to mathematics 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"

Dynamic Programming Applications

- Application Areas:

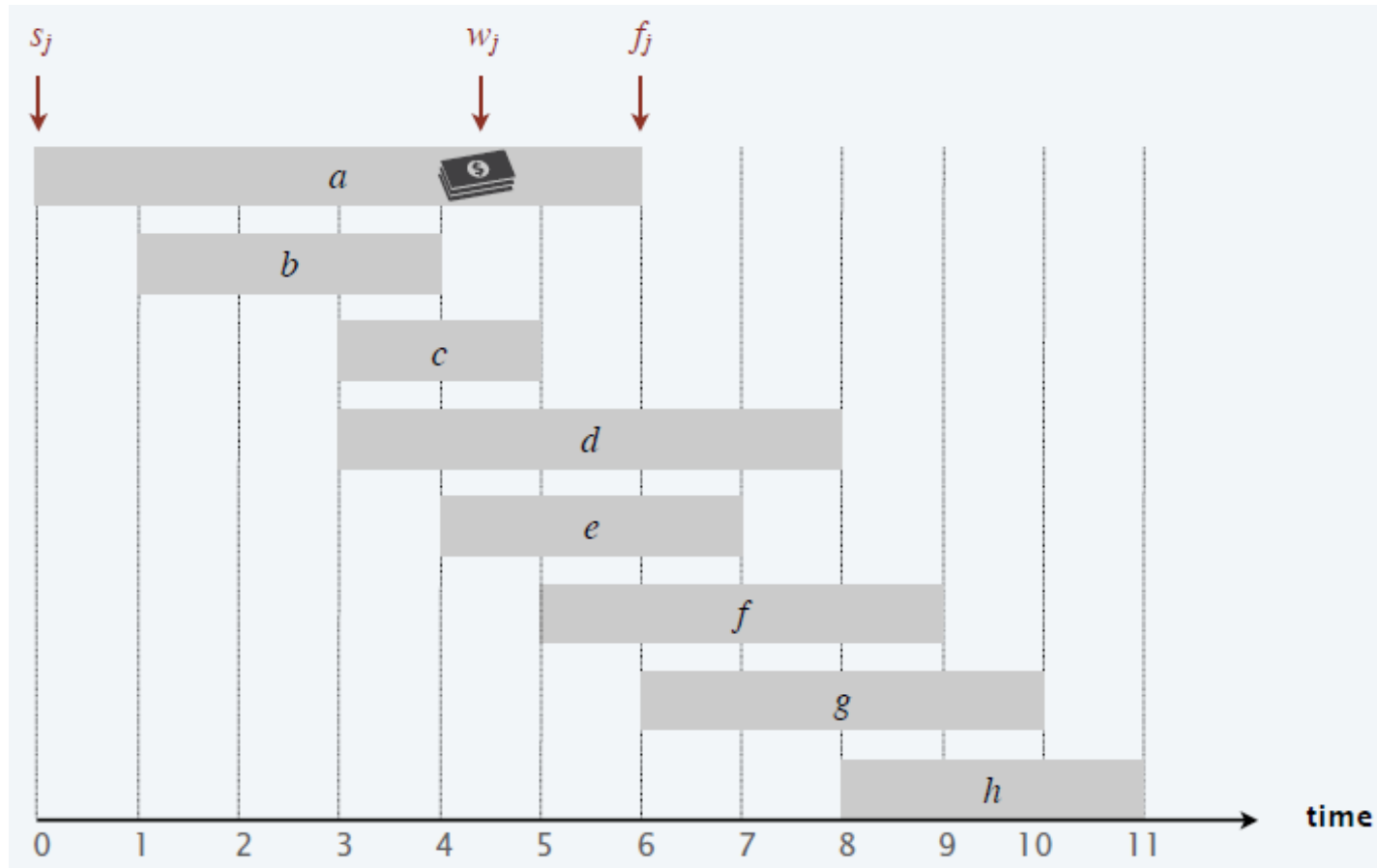
- **Bioinformatics.**
- Control Theory.
- Information Theory.
- Operations Research.
- **Computer Science: AI, Compilers, Graphics, Systems, Theory,**

- Some Famous Dynamic Programming Algorithms:

- Viterbi for Hidden Markov Models.
- Unix diff for Comparing Two Files.
- Knuth–Plass for word wrapping text in
- Needleman-Wunsch/Smith-Waterman TEX sequence Alignment.
- **Bellman-Ford-Moore for Shortest Path Routing in Networks.**
- Cocke-Kasami-Younger for Parsing Context Free Grammars.

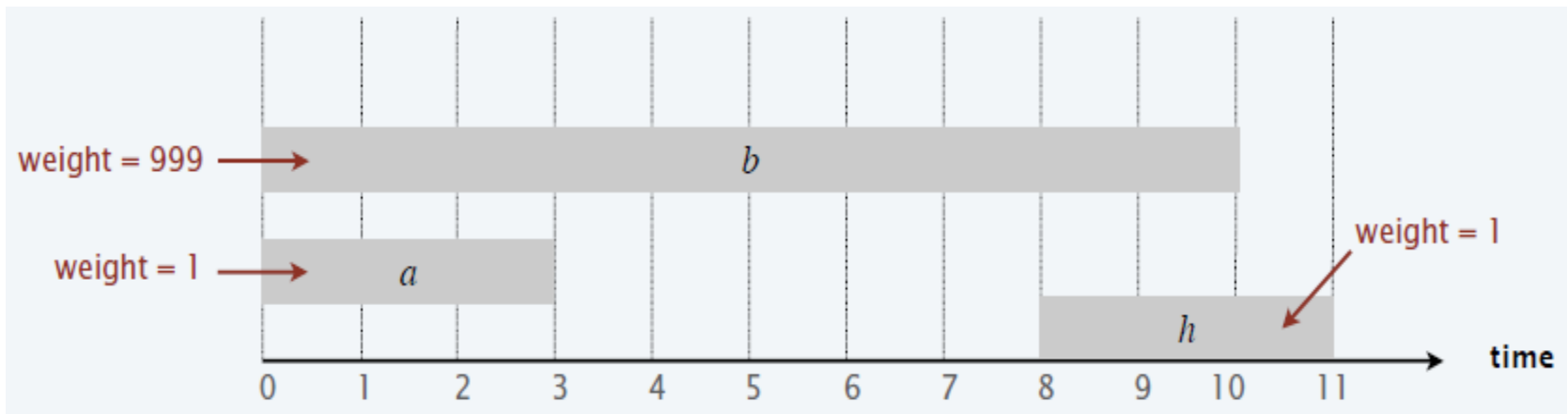
Weighted Interval Scheduling

- **Weighted Interval Scheduling Problem.**
 - Job j starts at s_j , finishes at f_j , and has weight or value $w_j > 0$.
 - Two jobs are **Compatible** if they don't overlap.
 - **Goal:** To find the **Maximum Weight** subset of Mutually Compatible Jobs.



Unweighted Interval Scheduling (Earliest-Finish-Time First Algorithm)

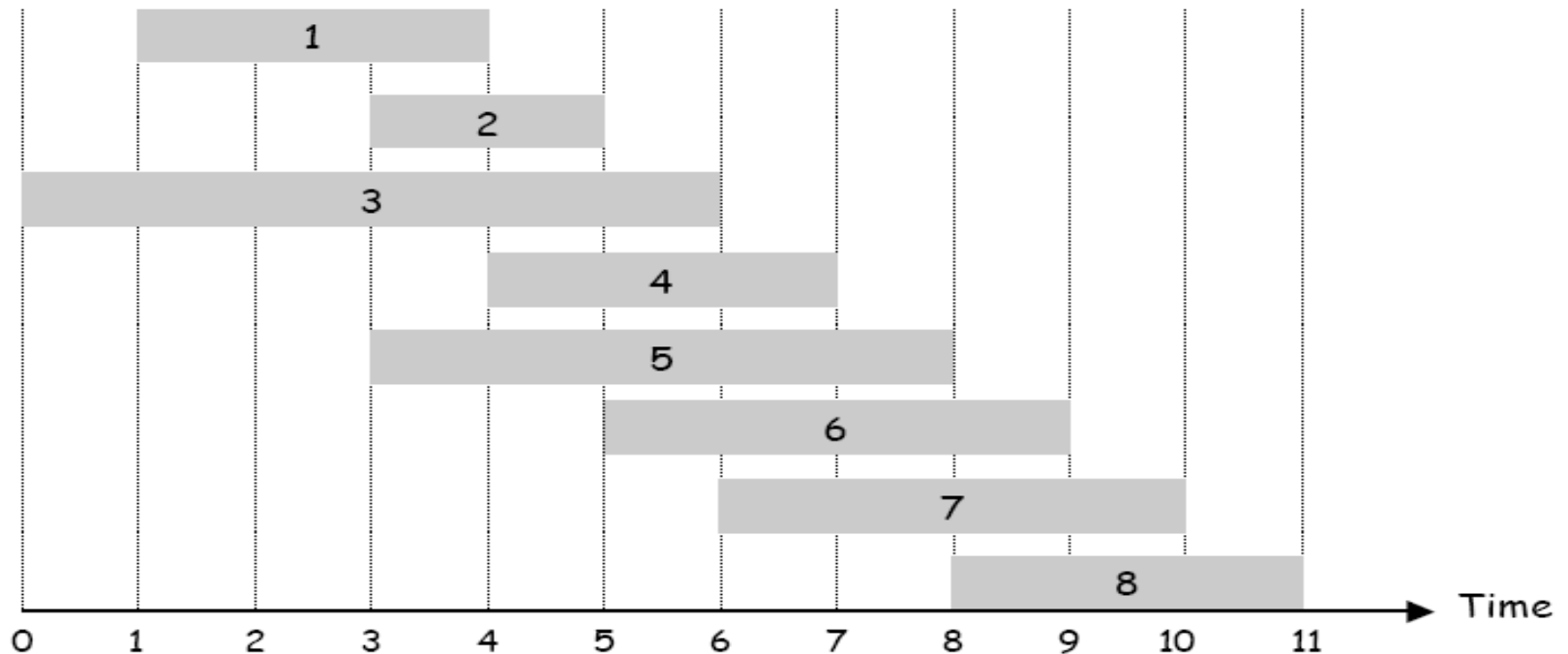
- Earliest-Finish-Time First.
 - Consider jobs in ascending order of finish time.
 - Add job to subset if it is compatible with previously chosen jobs
- Recall. Greedy Algorithm works if all weights are 1.
- Observation. Greedy Algorithm fails spectacularly for weighted version i.e. if arbitrary weights are allowed.



Weighted Interval Scheduling

- **Notation.** Label jobs by finishing time: $f_1 \leq f_2 \leq \dots \leq f_n$.
- **Def.** $p(j)$ = Largest index $i < j$ such that job i is compatible with j .
- **Ex:** $p(8) = 5$, $p(7) = 3$, $p(2) = 0$.

i is rightmost interval that ends before j begins



Dynamic Programming: Binary Choice

Def. $OPT(j)$ = max weight of any subset of mutually compatible jobs for subproblem consisting only of jobs $1, 2, \dots, j$.

Goal. $OPT(n)$ = max weight of any subset of mutually compatible jobs.

Case 1. $OPT(j)$ does not select job j .

- Must be an optimal solution to problem consisting of remaining jobs $1, 2, \dots, j-1$.

Case 2. $OPT(j)$ selects job j .

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

↖ ↗
optimal substructure property
(proof via exchange argument)

Bellman equation.
$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max \{ OPT(j-1), w_j + OPT(p(j)) \} & \text{if } j > 0 \end{cases}$$

Weighted Interval Scheduling: Brute Force

BRUTE-FORCE ($n, s_1, \dots, s_n, f_1, \dots, f_n, w_1, \dots, w_n$)

Sort jobs by finish time and renumber so that $f_1 \leq f_2 \leq \dots \leq f_n$.

Compute $p[1], p[2], \dots, p[n]$ via binary search.

RETURN COMPUTE-OPT(n).

COMPUTE-OPT(j)

IF ($j = 0$)

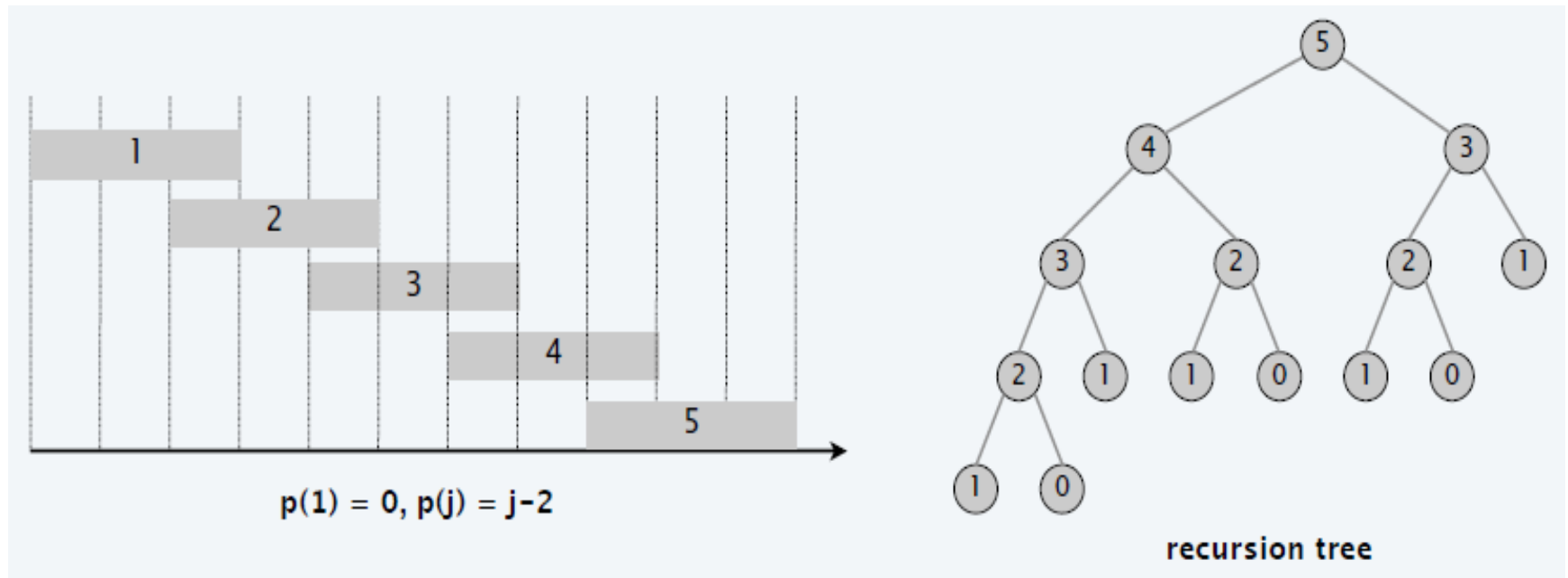
 RETURN 0.

ELSE

 RETURN $\max \{ \text{COMPUTE-OPT}(j-1), w_j + \text{COMPUTE-OPT}(p[j]) \}$.

Weighted Interval Scheduling: Brute Force

- **Observation.** Recursive algorithm is spectacularly slow because of overlapping sub-problems \Rightarrow exponential-time algorithm.
- **Ex.** Number of recursive calls for family of "layered" instances grows like Fibonacci sequence.



Weighted Interval Scheduling: Memoization

- **Top-Down Dynamic Programming (Memoization):** It is an Optimization Technique used mainly to speed up programs by storing the results of expensive function calls and returning the cached result ($M[j]$) when the same inputs occur again, thus avoids solving the sub-problem j more than once. Store results of each sub-problem j in a cache ($M[j]$); lookup table as needed.

TOP-DOWN($n, s_1, \dots, s_n, f_1, \dots, f_n, w_1, \dots, w_n$)

Sort jobs by finish time and renumber so that $f_1 \leq f_2 \leq \dots \leq f_n$.

Compute $p[1], p[2], \dots, p[n]$ via binary search.

$M[0] \leftarrow 0$.  global array

RETURN M-COMPUTE-OPT(n).

M-COMPUTE-OPT(j)

IF ($M[j]$ is uninitialized)

$M[j] \leftarrow \max \{ \text{M-COMPUTE-OPT}(j-1), w_j + \text{M-COMPUTE-OPT}(p[j]) \}$.

RETURN $M[j]$.

Weighted Interval Scheduling: Running Time

- **Claim.** Memoized version of algorithm takes $O(n \log n)$ time.
- **Proof.**
 - Sort by finish time: $O(n \log n)$ via Merge-Sort.
 - Computing $p(j)$ for each j : $O(n \log n)$ via Binary Search
 - M-Compute-Opt(j): each invocation takes $O(1)$ time and either
 - (i) returns an initialized value $M[j]$
 - (ii) initializes $M[j]$ and makes two recursive calls
 - Progress measure $\Phi = \#$ initialized entries among $M[1, \dots, n]$.
 - initially $\Phi = 0$, throughout $\Phi \leq n$.
 - (ii) increases Φ by 1 $\Rightarrow \leq 2n$ recursive calls (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.

Automated Memoization

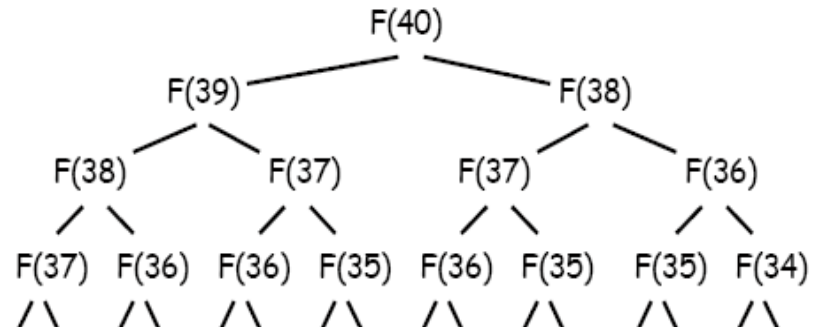
- **Automated Memoization.** Many functional programming languages (e.g., Lisp) have built-in support for memoization.
- **Question:** Why not in imperative languages (e.g., Java)?

```
(defun F (n)
  (if
    (<= n 1)
    n
    (+ (F (- n 1)) (F (- n 2))))))
```

Lisp (efficient)

```
static int F(int n) {
  if (n <= 1) return n;
  else return F(n-1) + F(n-2);
}
```

Java (exponential)



Weighted Interval Scheduling: Finding Solution

- **Question:** Dynamic programming algorithms computes optimal value. What if we want the solution itself?
- **Answer:** Do some Post-processing i.e. Make a second pass by calling Find-Solution(n).

FIND-SOLUTION(j)

IF ($j = 0$)

 RETURN \emptyset .

ELSE IF ($w_j + M[p[j]] > M[j-1]$)

 RETURN $\{j\} \cup \text{FIND-SOLUTION}(p[j])$.

ELSE

 RETURN FIND-SOLUTION($j-1$).

$$M[j] = \max \{ M[j-1], w_j + M[p[j]] \}.$$

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

Weighted Interval Scheduling: Bottom-Up

- Bottom-up Dynamic Programming: Unwind Recursion.

BOTTOM-UP($n, s_1, \dots, s_n, f_1, \dots, f_n, w_1, \dots, w_n$)

Sort jobs by finish time and renumber so that $f_1 \leq f_2 \leq \dots \leq f_n$.

Compute $p[1], p[2], \dots, p[n]$.

$M[0] \leftarrow 0$.

previously computed values

FOR $j = 1$ TO n

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

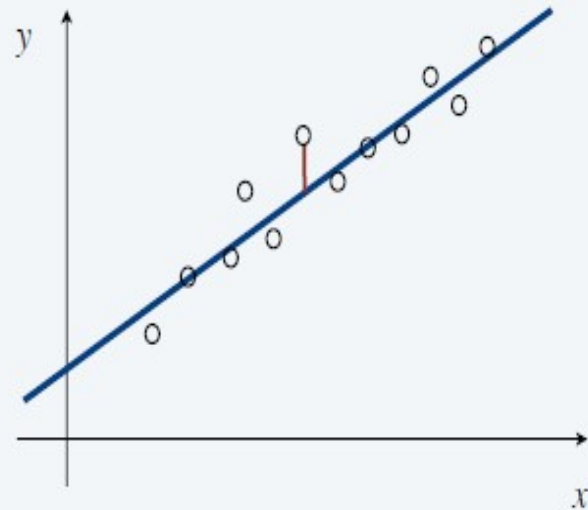
Running time. The bottom-up version takes $O(n \log n)$ time.

Least Squares

- Least Squares.

- Foundational problem in statistics and numerical analysis.
- Given n points in the plane: $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$.
- Find a line $y = ax + b$ that minimizes the sum of the squared error (SSE):

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



Solution. Calculus \Rightarrow minimum error is achieved when

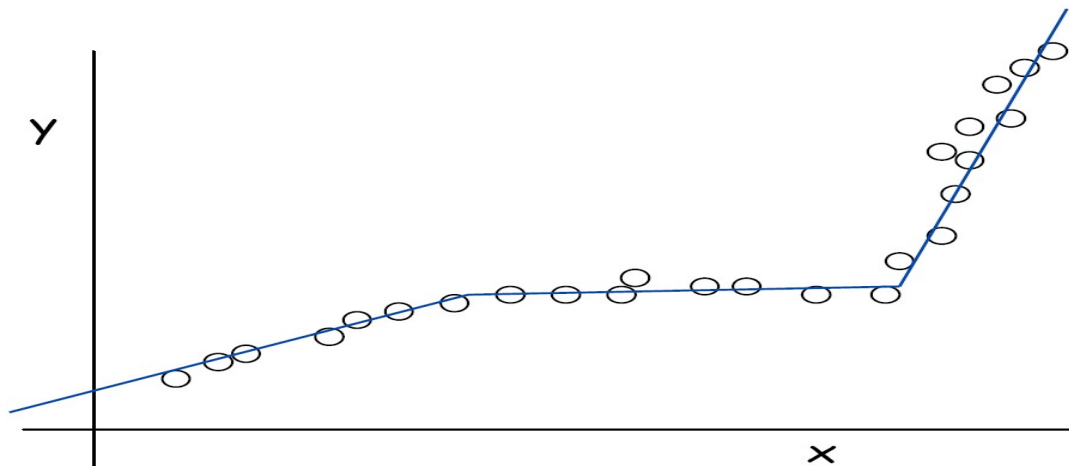
$$a = \frac{n \sum_i x_i y_i - (\sum_i x_i)(\sum_i y_i)}{n \sum_i x_i^2 - (\sum_i x_i)^2}, \quad b = \frac{\sum_i y_i - a \sum_i x_i}{n}$$

Segmented Least Squares

- Segmented Least Squares.
 - Points lie roughly on a sequence of several line segments.
 - Given n points in the plane $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$ with
 - $x_1 < x_2 < \dots < x_n$, find a sequence of lines that minimizes $f(x)$.
- Q. What's a reasonable choice for $f(x)$ to balance accuracy (goodness of fit) and parsimony (number of lines)?

Goal. Minimize $f(x) = E + c L$ for some constant $c > 0$, where

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




Dynamic Programming: Multiway Choice

Notation.

- $OPT(j)$ = minimum cost for points p_1, p_2, \dots, p_j .
- e_{ij} = SSE for for points p_i, p_{i+1}, \dots, p_j .

To compute $OPT(j)$:

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

Bellman equation.

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \min_{1 \leq i \leq j} \{ e_{ij} + c + OPT(i - 1) \} & \text{if } j > 0 \end{cases}$$

Segmented Least Squares: Algorithm

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

FOR $j = 1$ TO n

FOR $i = 1$ TO j

Compute the SSE e_{ij} for the points p_i, p_{i+1}, \dots, p_j .

$M[0] \leftarrow 0$.

FOR $j = 1$ TO n

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

previously computed value



RETURN $M[n]$.

Segmented least squares analysis

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

Pf.

- Bottleneck = computing SSE e_{ij} for each i and j .

$$a_{ij} = \frac{n \sum_k x_k y_k - (\sum_k x_k)(\sum_k y_k)}{n \sum_k x_k^2 - (\sum_k x_k)^2}, \quad b_{ij} = \frac{\sum_k y_k - a_{ij} \sum_k x_k}{n}$$

- $O(n)$ to compute e_{ij} .
 - There are $O(n^2)$ pairs of ij for which we need to compute e_{ij} ; each e_{ij} computation takes $O(n)$ time. Thus all e_{ij} values can be computed in $O(n^3)$ time. For storing all e_{ij} values, OPT array can be filled in $O(n^2)$ time.

Remark. Can be improved to $O(n^2)$ time.

- For each i : precompute cumulative sums $\sum_{k=1}^i x_k$, $\sum_{k=1}^i y_k$, $\sum_{k=1}^i x_k^2$, $\sum_{k=1}^i x_k y_k$.
- Using cumulative sums, can compute e_{ij} in $O(1)$ time.

Dynamic programming: two variables

Def. $OPT(i, w)$ = optimal value of knapsack problem with items $1, \dots, i$, subject to weight limit w .

Goal. $OPT(n, W)$.

Case 1. $OPT(i, w)$ does not select item i .

possibly because $w_i > w$

- $OPT(i, w)$ selects best of $\{1, 2, \dots, i-1\}$ subject to weight limit w .

Case 2. $OPT(i, w)$ selects item i .

optimal substructure property
(proof via exchange argument)

- Collect value v_i .
- New weight limit $= w - w_i$.
- $OPT(i, w)$ selects best of $\{1, 2, \dots, i-1\}$ subject to new weight limit.

Bellman equation.

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

Knapsack Problem

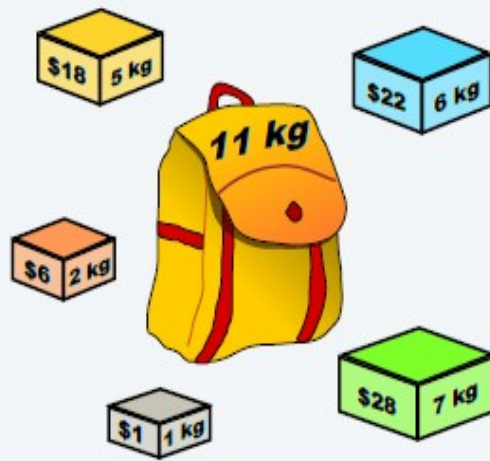
Goal. Pack knapsack so as to maximize total value of items taken.

- There are n items: item i provides value $v_i > 0$ and weighs $w_i > 0$.
- Value of a subset of items = sum of values of individual items.
- Knapsack has weight limit of W .

Ex. The subset $\{1, 2, 5\}$ has value \$35 (and weight 10).

Ex. The subset $\{3, 4\}$ has value \$40 (and weight 11).

Assumption. All values and weights are integral.



Creative Commons Attribution-Share Alike 2.5
by Dake

i	v_i	w_i
1	\$1	1 kg
2	\$6	2 kg
3	\$18	5 kg
4	\$22	6 kg
5	\$28	7 kg

knapsack instance
(weight limit $W = 11$)

weights and values
can be arbitrary
positive integers

Knapsack problem: bottom-up dynamic programming

KNAPSACK($n, W, w_1, \dots, w_n, v_1, \dots, v_n$)

FOR $w = 0$ **TO** W

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

FOR $i = 1$ **TO** n

FOR $w = 0$ **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 $M[n, W].$

previously computed values



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

Knapsack problem: bottom-up dynamic programming demo

i	v_i	w_i
1	\$1	1 kg
2	\$6	2 kg
3	\$18	5 kg
4	\$22	6 kg
5	\$28	7 kg

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

		weight limit w											
		0	1	2	3	4	5	6	7	8	9	10	11
subset of items 1, ..., i	{ }	0	0	0	0	0	0	0	0	0	0	0	0
	{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
	{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
	{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
	{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
	{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	35	40

$OPT(i, w)$ = optimal value of knapsack problem with items 1, ..., i , subject to weight limit w

Knapsack problem: running time

Theorem. The DP algorithm solves the knapsack problem with n items and maximum weight W in $\Theta(n W)$ time and $\Theta(n W)$ space.

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:
 $OPT(i, w)$ takes item i iff $M[i, w] > M[i - 1, w]$. ■

weights are integers
between 1 and W

Remarks.

- Algorithm depends critically on assumption that weights are integral.
- Assumption that values are integral was not used.
- **Running time.** $\Theta(n W)$.
 - Not polynomial in input size!
 - "Pseudo-polynomial."
 - Decision version of Knapsack Problem is NP-complete Problem.
- **Knapsack Approximation Algorithm.** There exists a polynomial algorithm that produces a feasible solution that has value within **0.01% of Optimum Solution.**

Sequence Alignment Problem

String similarity

Q. How similar are two strings?

Ex. occurrence and occurance.

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

Sequence Alignment Problem

Edit distance

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

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

assuming $\alpha_{AA} = \alpha_{CC} = \alpha_{GG} = \alpha_{TT} = 0$

Applications. Bioinformatics, spell correction, machine translation, speech recognition, information extraction, ...

Spokesperson confirms	senior government	adviser was found
Spokesperson said	the senior	adviser was found

Sequence Alignment Problem

BLOSUM matrix for proteins

In bioinformatics, the BLOSUM (BLOCKS SUBstitution Matrix) matrix is a substitution matrix used for sequence alignment of proteins. BLOSUM matrices are used to score alignments between evolutionarily divergent protein sequences using local alignments.

	A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V
A	7	-3	-3	-3	-1	-2	-2	0	-3	-3	-3	-1	-2	-4	-1	2	0	-5	-4	-1
R	-3	9	-1	-3	-6	1	-1	-4	0	-5	-4	3	-3	-5	-3	-2	-2	-5	-4	-4
N	-3	-1	9	2	-5	0	-1	-1	1	-6	-6	0	-4	-6	-4	1	0	-7	-4	-5
D	-3	-3	2	10	-7	-1	2	-3	-2	-7	-7	-2	-6	-6	-3	-1	-2	-8	-6	-6
C	-1	-6	-5	-7	13	-5	-7	-6	-7	-2	-3	-6	-3	-4	-6	-2	-2	-5	-5	-2
Q	-2	1	0	-1	-5	9	3	-4	1	-5	-4	2	-1	-5	-3	-1	-1	-4	-3	-4
E	-2	-1	-1	2	-7	3	8	-4	0	-6	-6	1	-4	-6	-2	-1	-2	-6	-5	-4
G	0	-4	-1	-3	-6	-4	-4	9	-4	-7	-7	-3	-5	-6	-5	-1	-3	-6	-6	-6
H	-3	0	1	-2	-7	1	0	-4	12	-6	-5	-1	-4	-2	-4	-2	-3	-4	3	-5
I	-3	-5	-6	-7	-2	-5	-6	-7	-6	7	2	-5	2	-1	-5	-4	-2	-5	-3	4
L	-3	-4	-6	-7	-3	-4	-6	-7	-5	2	6	-4	3	0	-5	-4	-3	-4	-2	1
K	-1	3	0	-2	-6	2	1	-3	-1	-5	-4	8	-3	-5	-2	-1	-1	-6	-4	-4
M	-2	-3	-4	-6	-3	-1	-4	-5	-4	2	3	-3	9	0	-4	-3	-1	-3	-3	1
F	-4	-5	-6	-6	-4	-5	-6	-6	-2	-1	0	-5	0	10	-6	-4	-4	0	4	-2
P	-1	-3	-4	-3	-6	-3	-2	-5	-4	-5	-5	-2	-4	-6	12	-2	-3	-7	-6	-4
S	2	-2	1	-1	-2	-1	-1	-1	-2	-4	-4	-1	-3	-4	-2	7	2	-6	-3	-3
T	0	-2	0	-2	-2	-1	-2	-3	-3	-2	-3	-1	-1	-4	-3	2	8	-5	-3	0
W	-5	-5	-7	-8	-5	-4	-6	-6	-4	-5	-4	-6	-3	0	-7	-6	-5	16	3	-5
Y	-4	-4	-4	-6	-5	-3	-5	-6	3	-3	-2	-4	-3	4	-6	-3	-3	3	11	-3
V	-1	-4	-5	-6	-2	-4	-4	-6	-5	4	1	-4	1	-2	-4	-3	0	-5	-3	7

Sequence Alignment Problem

Dynamic programming: quiz 1

What is edit distance between these two strings?

P A L E T T E

P A L A T E

Assume gap penalty = 2 and mismatch penalty = 1.

- A. 1
- B. 2
- C. 3
- D. 4
- E. 5

Sequence Alignment Problem

Sequence alignment

Goal. Given two strings $x_1 x_2 \dots x_m$ and $y_1 y_2 \dots y_n$, find a min-cost alignment.

Def. An **alignment** M is a set of ordered pairs $x_i - y_j$ such that each character appears 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:

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

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

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

Sequence alignment: problem structure

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

Goal. $OPT(m, n)$.

Case 1. $OPT(i, j)$ matches $x_i - y_j$.

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

Case 2a. $OPT(i, j)$ leaves x_i unmatched.

Pay gap for x_i + min cost of aligning $x_1 x_2 \dots x_{i-1}$ and $y_1 y_2 \dots y_j$.

Case 2b. $OPT(i, j)$ leaves y_j unmatched.

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

optimal substructure property
(proof via exchange argument)

Bellman equation.

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

Sequence Alignment Problem

Sequence alignment: bottom-up algorithm

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

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

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

already computed

RETURN $M[m, n].$

Sequence Alignment Problem

Sequence alignment: traceback

		P	A	L	A	T	E
	0	2	4	6	8	10	12
P	2	0	2	4	6	8	10
A	4	2	0	2	4	6	8
L	6	4	2	0	2	4	6
E	8	6	4	2	1	3	4
T	10	8	6	4	3	1	3
T	12	10	8	6	5	3	2
E	14	12	10	8	7	5	3

P	A	L	E	T	T	E
P	A	L	-	A	T	E

1 mismatch, 1 gap

Sequence Alignment Problem

Sequence alignment: analysis

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

Pf.

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

Theorem. [Backurs–Indyk 2015] If can compute edit distance of two strings of length n in $O(n^{2-\epsilon})$ time for some constant $\epsilon > 0$, then can solve SAT with n variables and m clauses in $\text{poly}(m) 2^{(1-\delta)n}$ time for some constant $\delta > 0$.

Edit Distance Cannot Be Computed
in Strongly Subquadratic Time
(unless SETH is false)*

Arturs Backurs[†]
MIT

Piotr Indyk[‡]
MIT

↖ which would disprove SETH
(strong exponential time hypothesis)

Sequence Alignment Problem

Dynamic programming: quiz 2

It is easy to modify the DP algorithm for edit distance to...

- A. Compute edit distance in $O(mn)$ time and $O(m + n)$ space.
- B. Compute an optimal alignment in $O(mn)$ time and $O(m + n)$ space.
- C. Both A and B.
- D. Neither A nor B.

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

Sequence Alignment Problem

Sequence alignment in linear space

Theorem. [Hirschberg] There exists an algorithm to find an optimal alignment in $O(mn)$ time and $O(m + n)$ space.

- 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

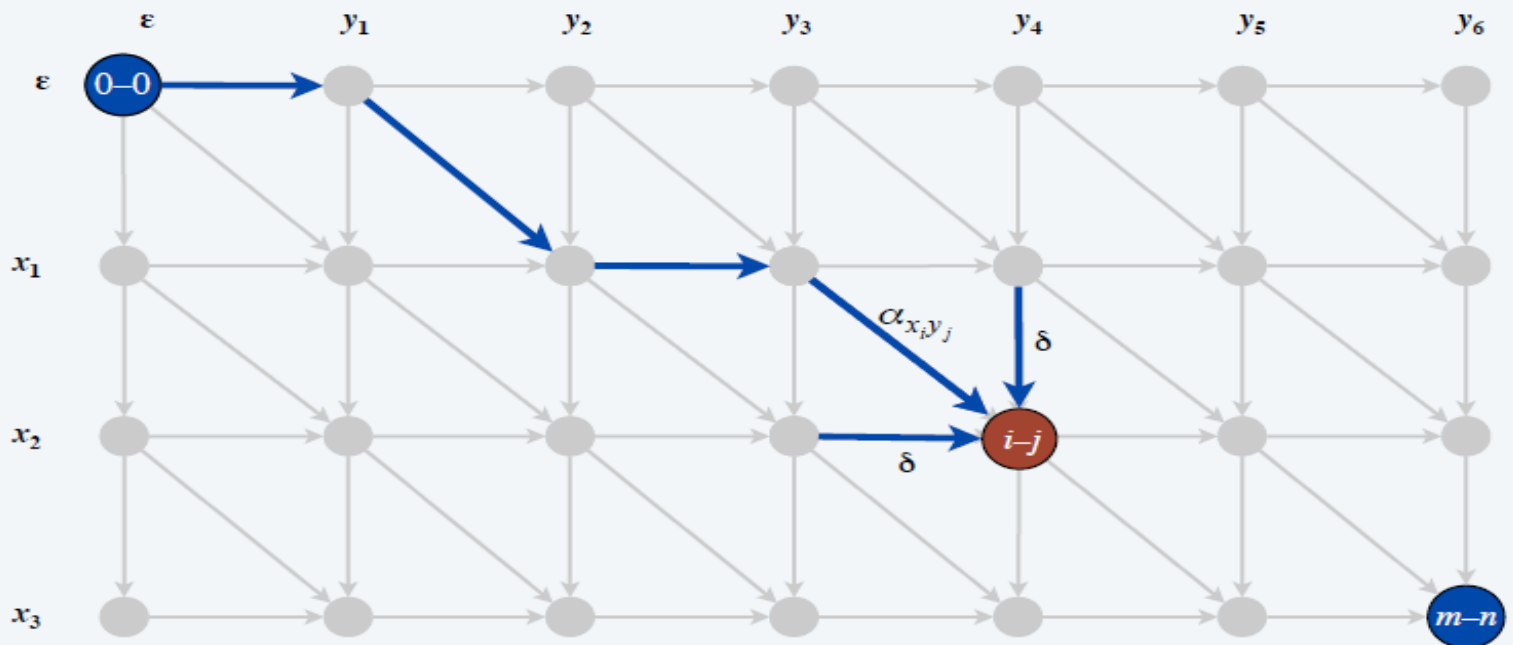


Sequence Alignment Problem

Hirschberg's algorithm

Edit distance graph.

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



Sequence Alignment Problem

Hirschberg's algorithm

Edit distance graph.

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

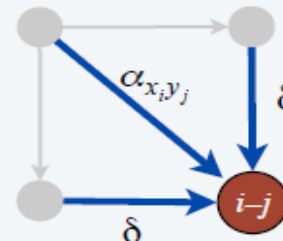
Pf of Lemma. [by strong induction on $i + j$]

- 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,

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

inductive
hypothesis

Bellman
equation

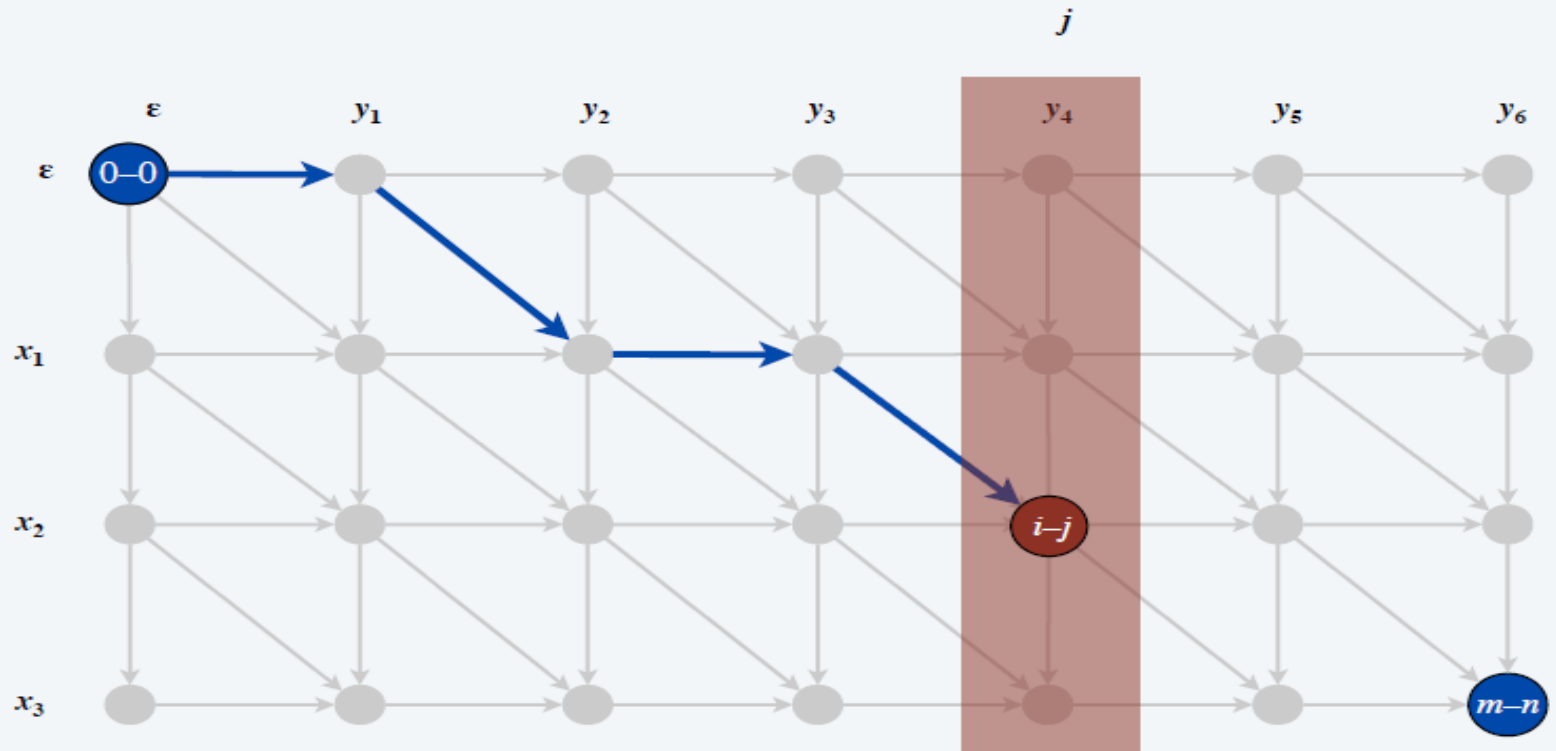


Sequence Alignment Problem

Hirschberg's algorithm

Edit distance graph.

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

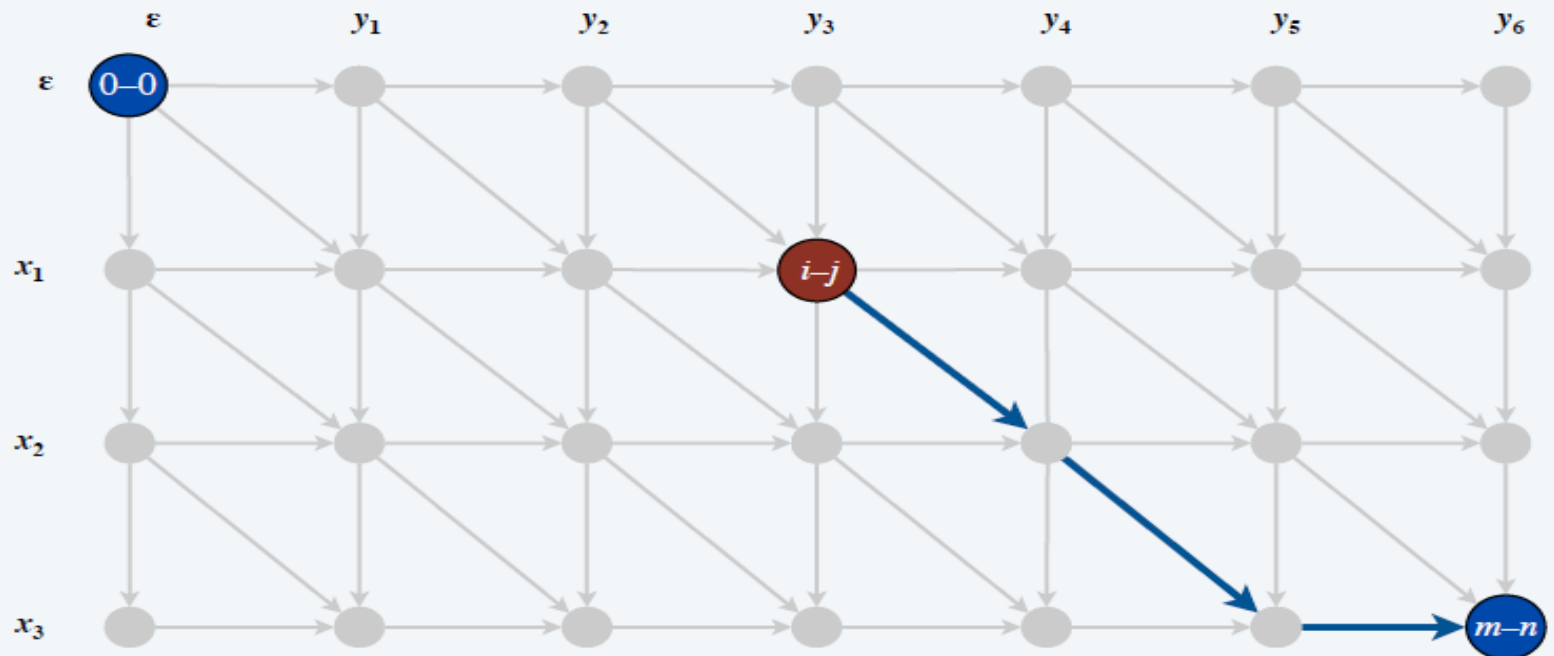


Sequence Alignment Problem

Hirschberg's algorithm

Edit distance graph.

- Let $g(i, j)$ denote length of shortest path from (i, j) to (m, n) .

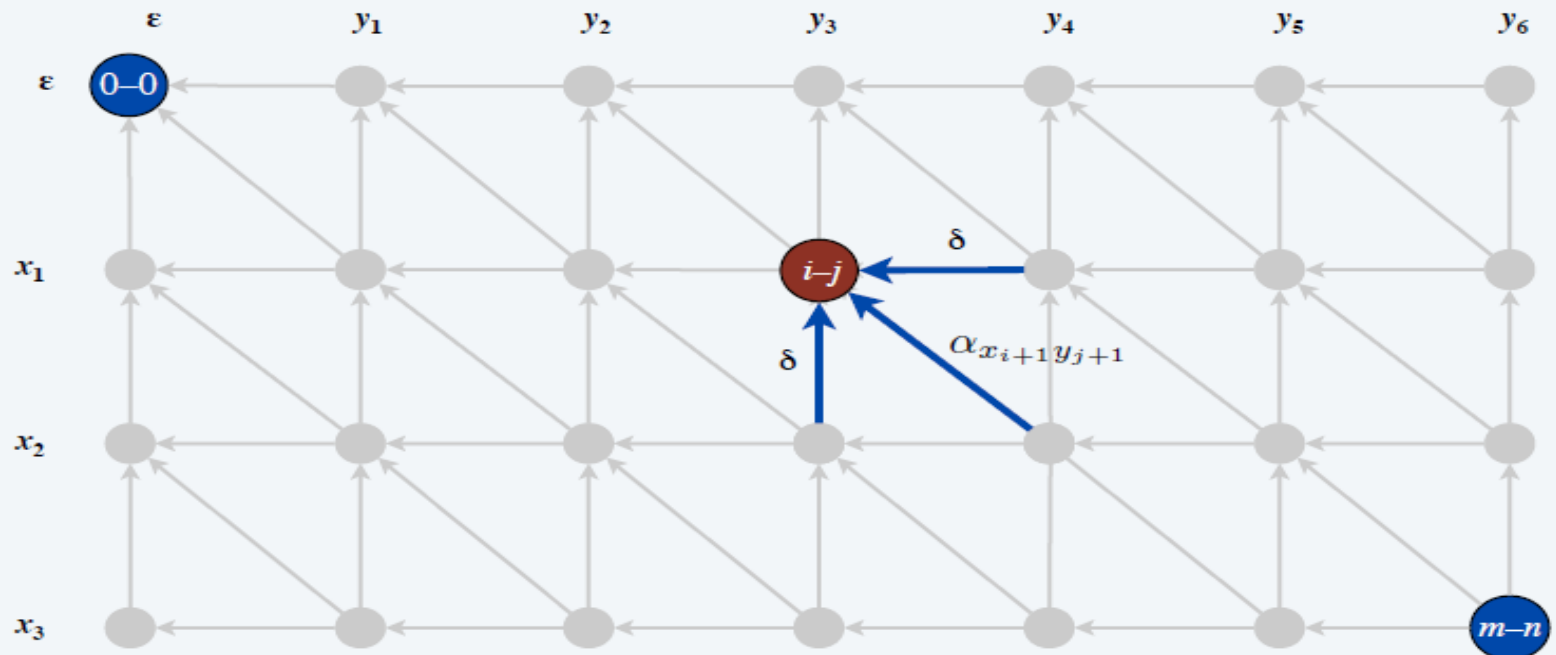


Sequence Alignment Problem

Hirschberg's algorithm

Edit distance graph.

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

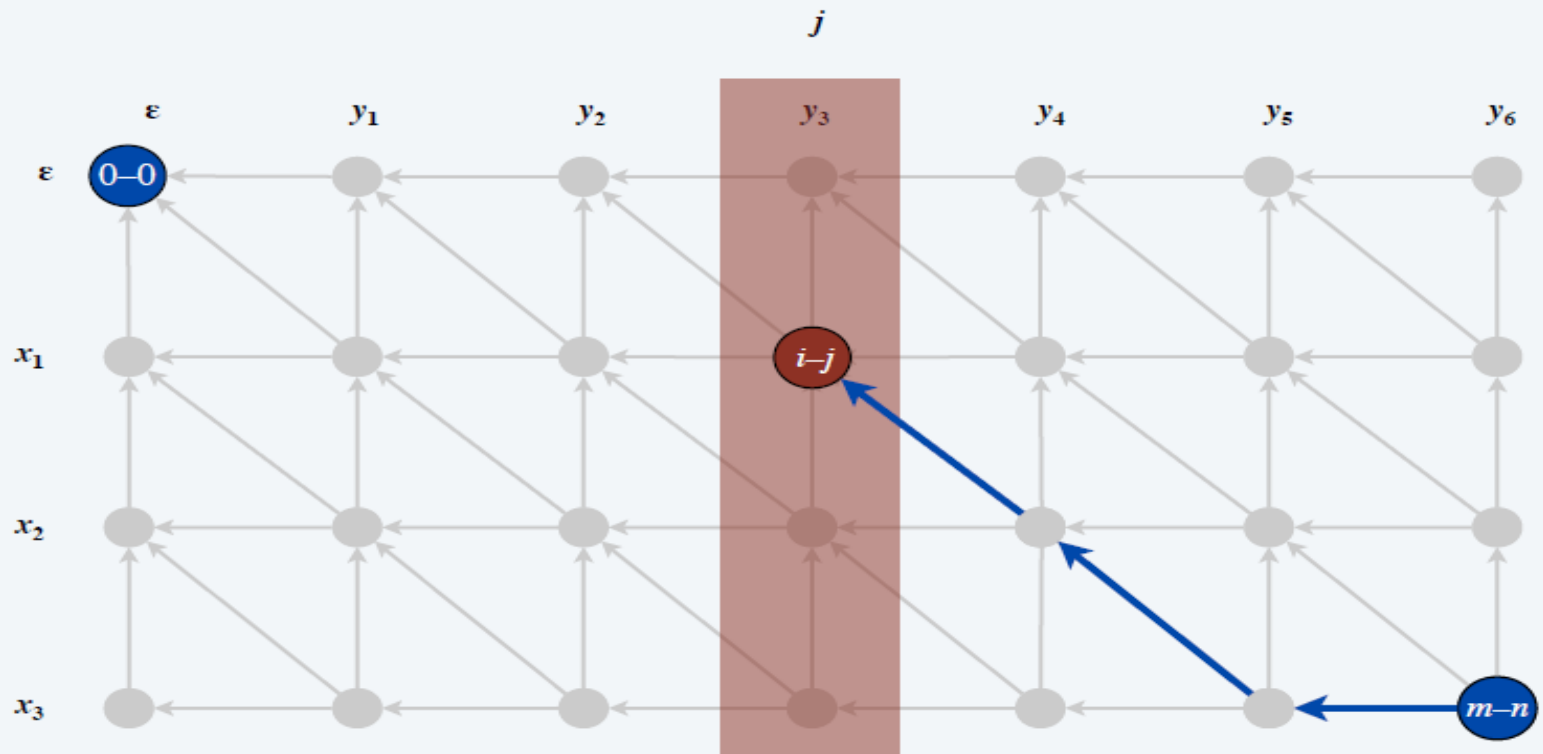


Sequence Alignment Problem

Hirschberg's algorithm

Edit distance graph.

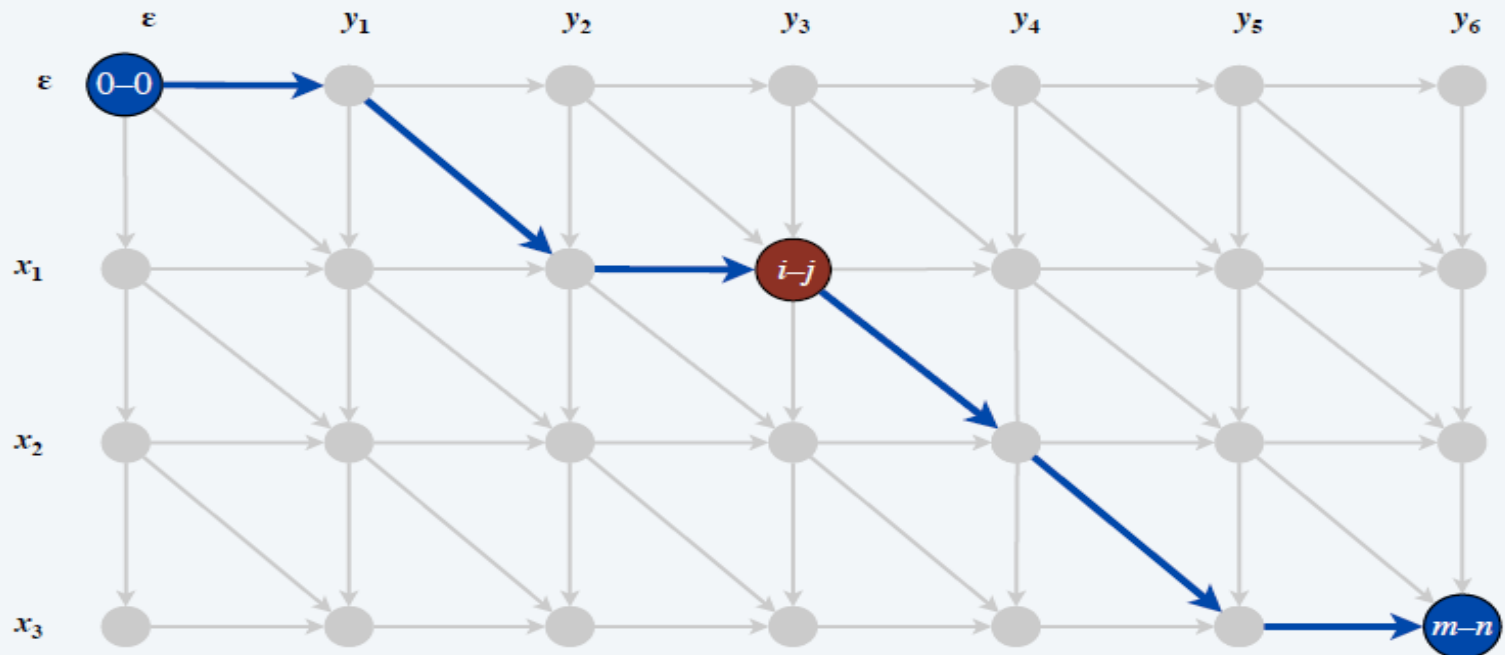
- Let $g(i, j)$ denote length of shortest path from (i, j) to (m, n) .
- Can compute $g(\cdot, j)$ for any j in $O(mn)$ time and $O(m + n)$ space.



Sequence Alignment Problem

Hirschberg's algorithm

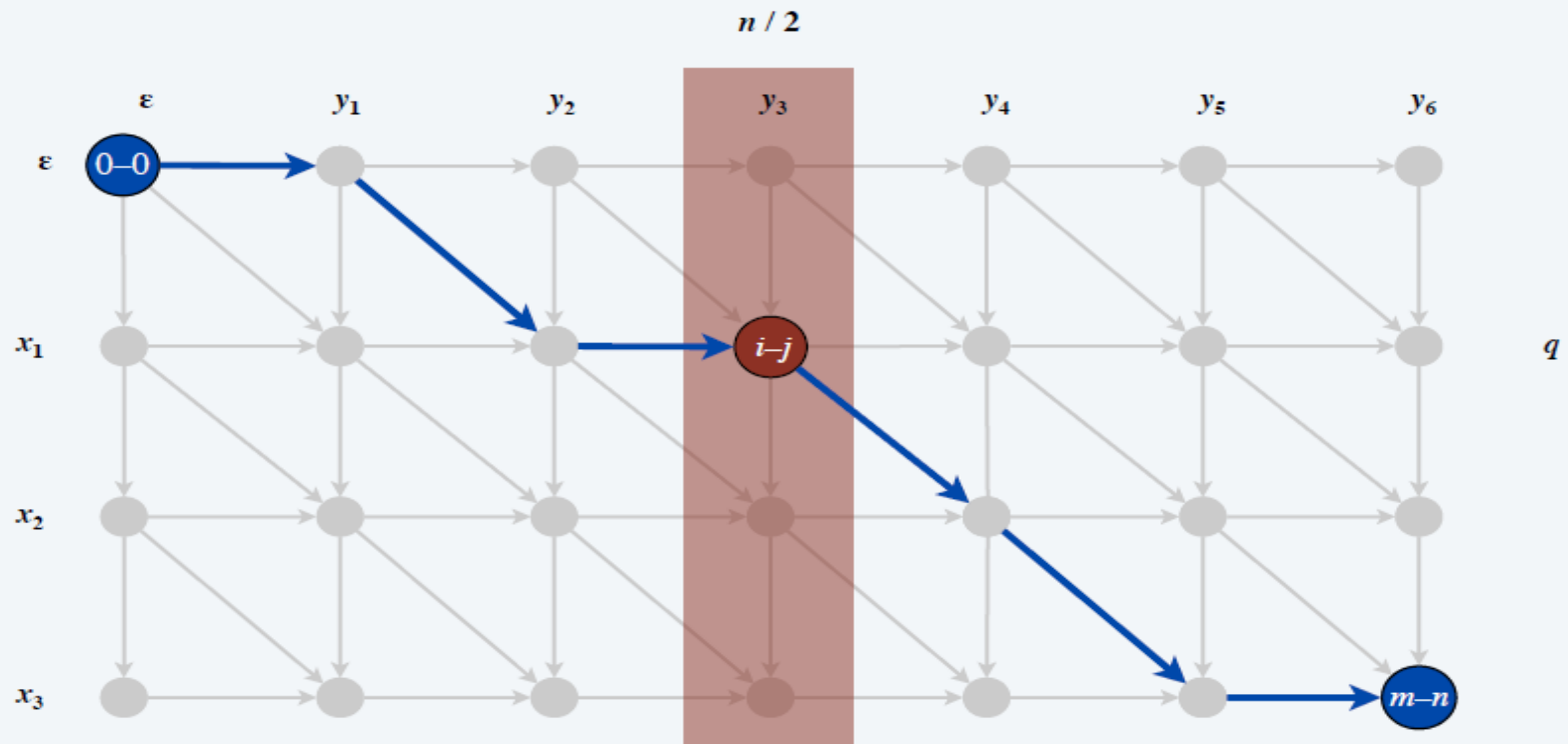
Observation 1. The length of a shortest path that uses (i, j) is $f(i, j) + g(i, j)$.



Sequence Alignment Problem

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) that uses $(q, n/2)$.

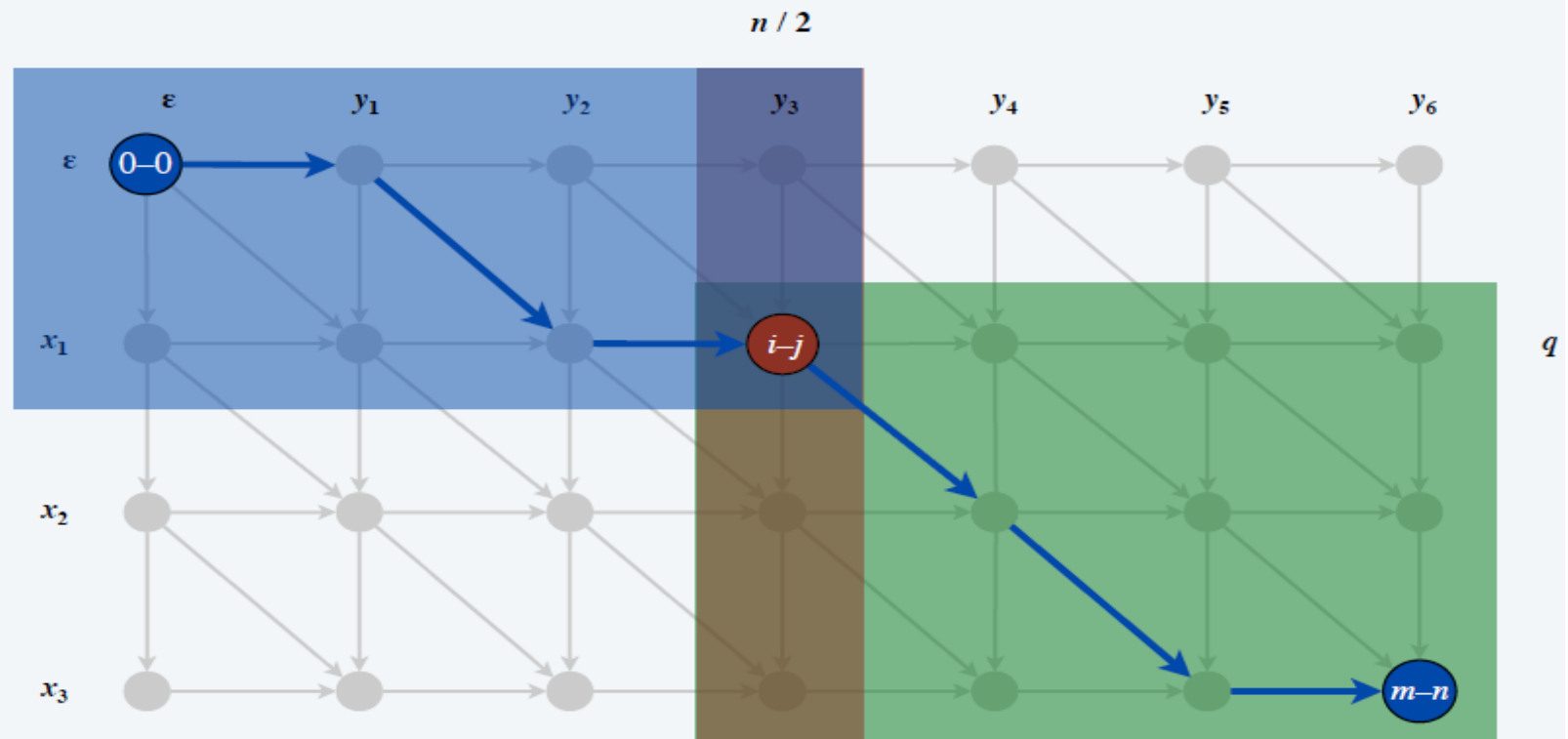


Sequence Alignment Problem

Hirschberg's algorithm

Divide. Find index q that minimizes $f(q, n/2) + g(q, n/2)$; save node $i-j$ as part of solution.

Conquer. Recursively compute optimal alignment in each piece.



Sequence Alignment Problem

Hirschberg's algorithm: space analysis

Theorem. Hirschberg's algorithm uses $\Theta(m + n)$ space.

Pf.

- Each recursive call uses $\Theta(m)$ space to compute $f(\cdot, n/2)$ and $g(\cdot, n/2)$.
- Only $\Theta(1)$ space needs to be maintained per recursive call.
- Number of recursive calls $\leq n$. ■

Hirschberg's algorithm: running time analysis warmup

Theorem. Let $T(m, n) = \max$ running time of Hirschberg's algorithm on strings of lengths at most m and n . Then, $T(m, n) = O(m n \log n)$.

Pf.

- $T(m, n)$ is monotone nondecreasing in both m and n .
- $T(m, n) \leq 2 T(m, n/2) + O(m n)$
 $\Rightarrow T(m, n) = O(m n \log n)$.

Remark. Analysis is not tight because two subproblems are of size $(q, n/2)$ and $(m - q, n/2)$. Next, we prove $T(m, n) = O(m n)$.

Sequence Alignment Problem

Hirschberg's algorithm: running time analysis

Theorem. Let $T(m, n)$ = max running time of Hirschberg's algorithm on strings of lengths at most m and n . Then, $T(m, n) = O(mn)$.

Pf. [by strong induction on $m + n$]

- $O(mn)$ time to compute $f(\cdot, n/2)$ and $g(\cdot, n/2)$ and find index q .
- $T(q, n/2) + T(m - q, n/2)$ time for two recursive calls.
- Choose constant c so that:
$$T(m, 2) \leq cm$$
$$T(2, n) \leq cn$$
$$T(m, n) \leq cmn + T(q, n/2) + T(m - q, n/2)$$
- Claim. $T(m, n) \leq 2cmn$.
- Base cases: $m = 2$ and $n = 2$.
- Inductive hypothesis: $T(m', n') \leq 2cm'n'$ for all (m', n') with $m' + n' < m + n$.

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

inductive hypothesis 