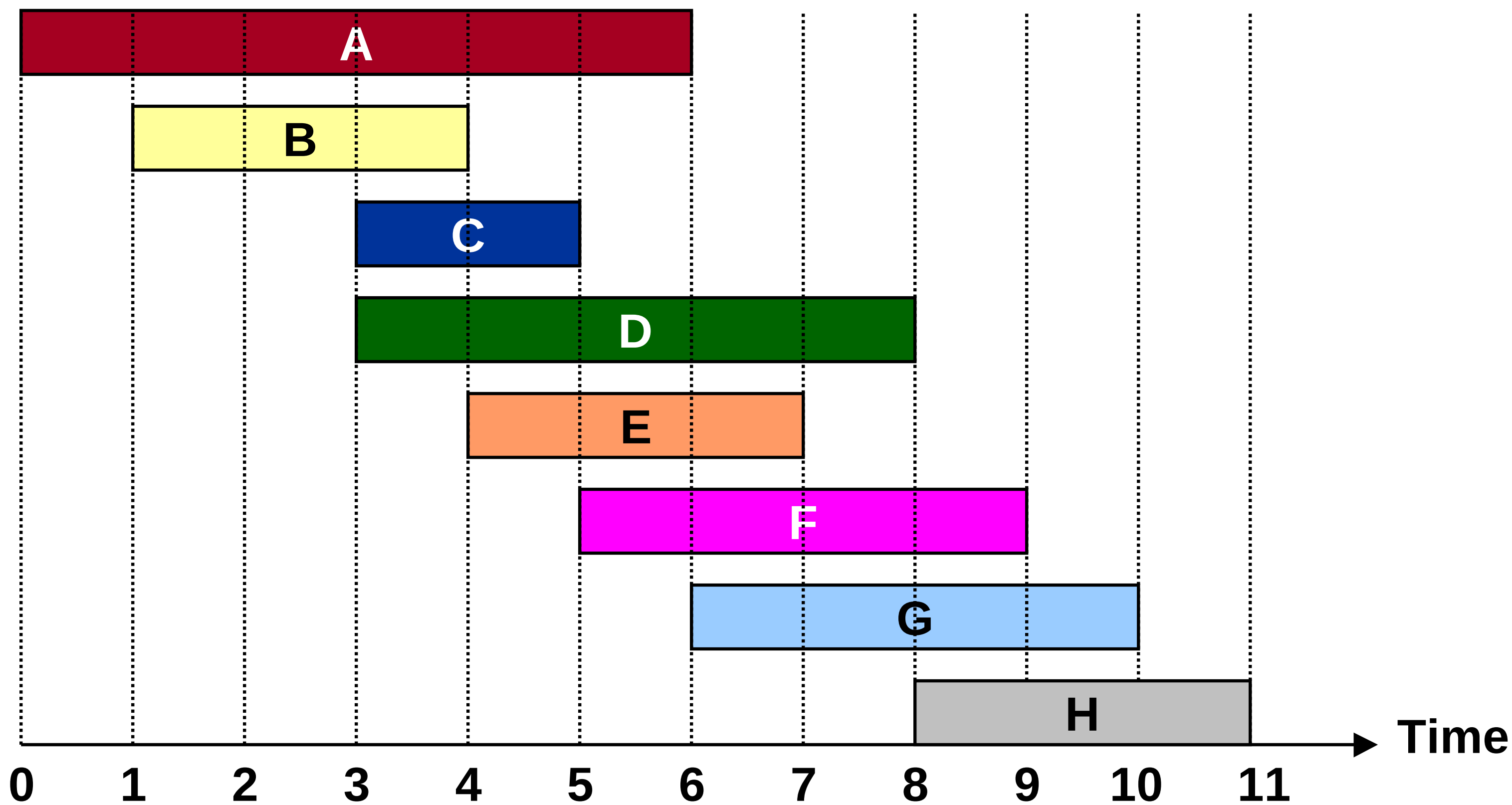
Dynamic Programming

Weighted Activity Selection

Weighted activity selection problem (generalization of CLR 17.1).

- **Job requests 1, 2, … , N.**
- **Job j starts at s_j, finishes at f_j , and has weight w_j .**
- **Two jobs compatible if they don't overlap.**
- **Goal: find maximum weight subset of mutually compatible jobs.**



Activity Selection: Greedy Algorithm

Recall greedy algorithm works if all weights are 1.

Greedy Activity Selection Algorithm

Sort jobs by increasing finish times so that $f_1 \le f_2 \le \dots \le f_N$.

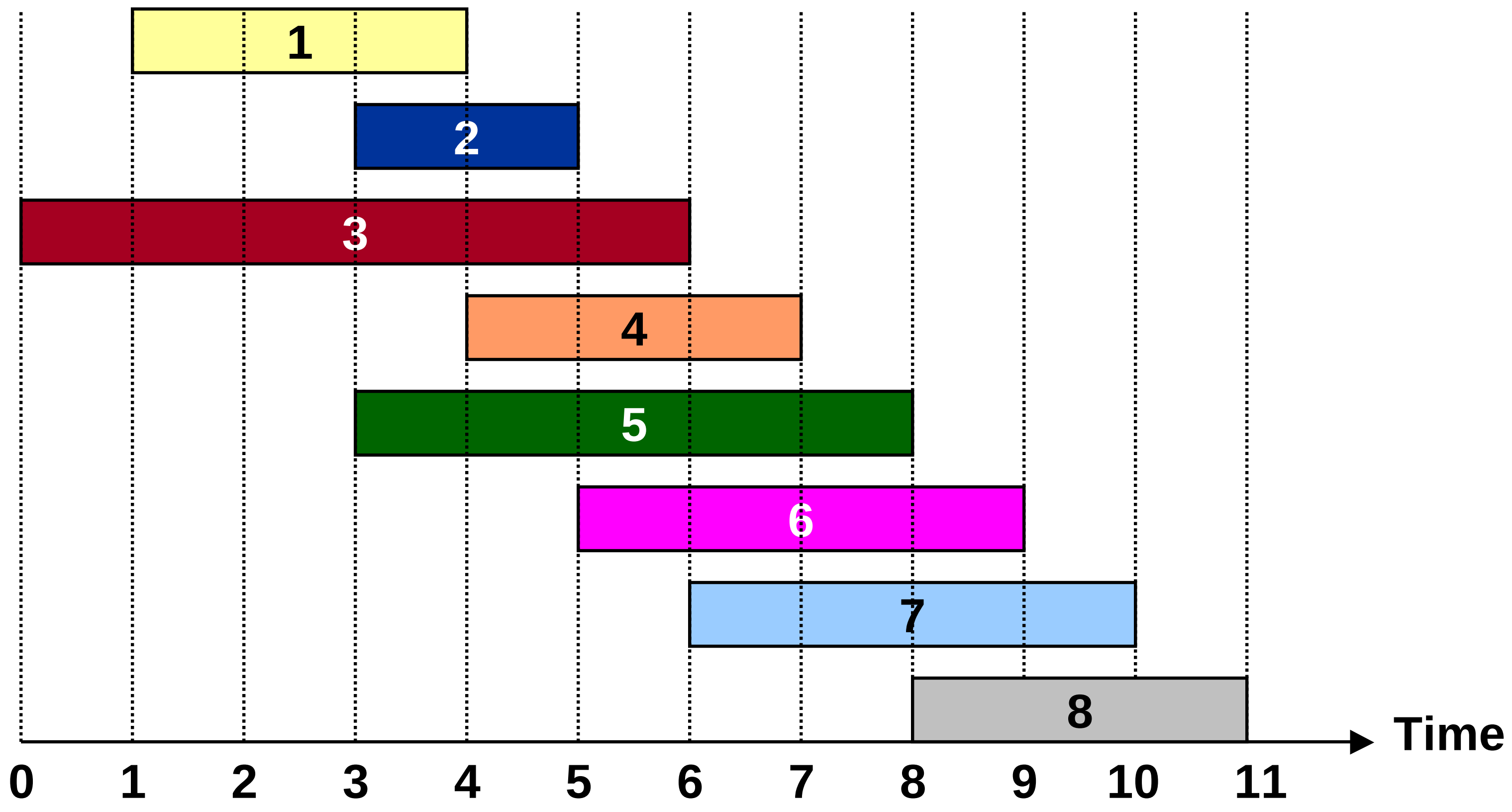
```
S = ϕ
FOR j = 1 to N
   IF (job j compatible with A)
      S ← S ∪ {j}
RETURN S
```

S = jobs selected.

Weighted Activity Selection

Notation.

- Label jobs by finishing time: $f_1 \le f_2 \le . . . \le f_N$.
- Define q_j = largest index i < j such that job i is compatible with j.
  - $q_7 = 3$, $q_2 = 0$



Weighted Activity Selection: Structure

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

- Case 1: OPT selects job j.
  - can't use incompatible jobs { q_j + 1, q_j + 2, . . . , j-1 }
  - must include optimal solution to problem consisting of remaining compatible jobs { 1, 2, . . . , q_j }
- Case 2: OPT does not select job j.
  - must include optimal solution to problem consisting of remaining compatible jobs { 1, 2, . . . , j - 1 }

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

Weighted Activity Selection: Brute Force

Recursive Activity Selection

INPUT: N, $s_1,\ldots,s_N$, $f_1,\ldots,f_N$, $w_1,\ldots,w_N$

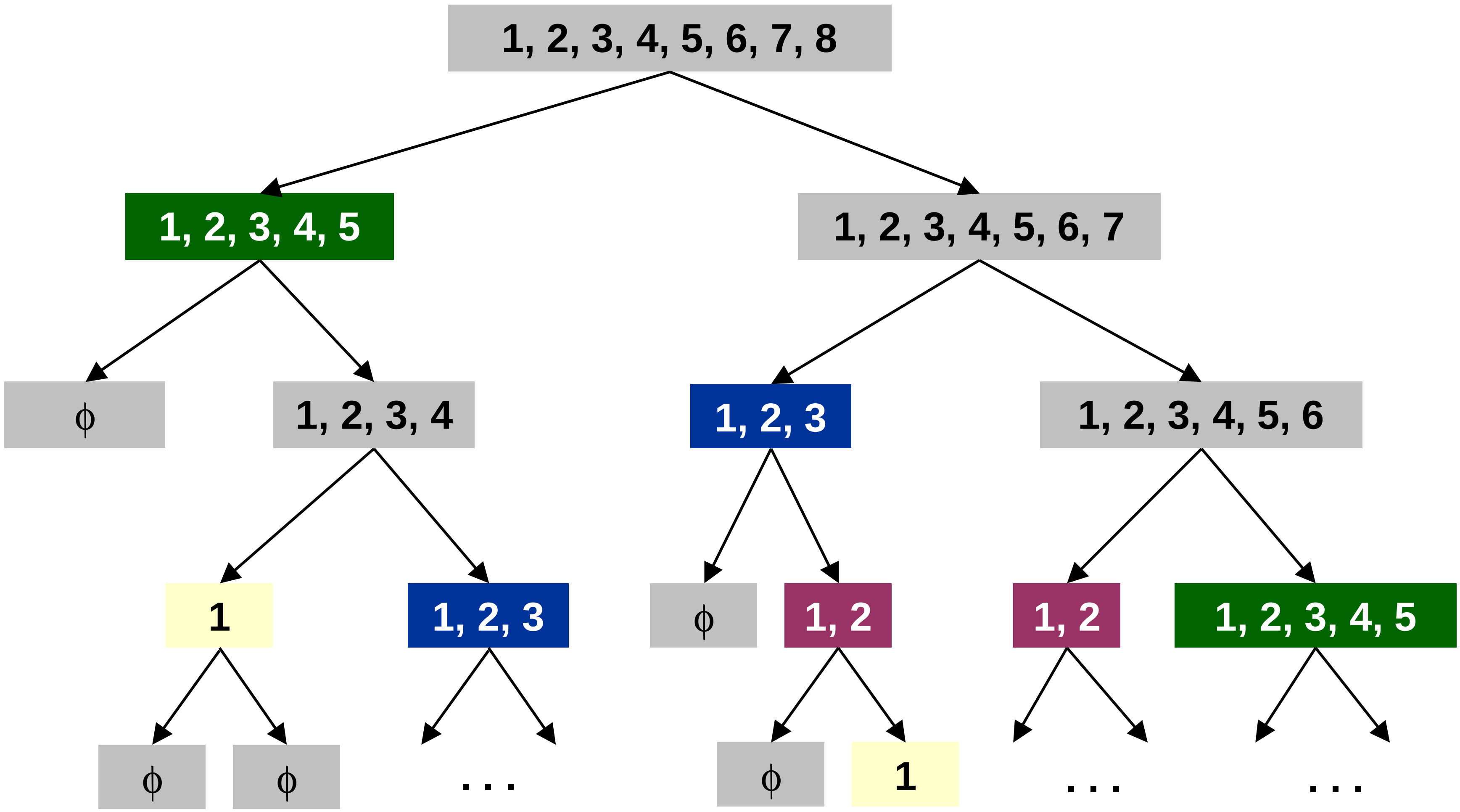
Sort jobs by increasing finish times so that
$f_1 \leq f_2 \leq \ldots \leq f_N$.

Compute q_1, q_2 , ... , q_N

```
r-compute(j) {
   IF (j = 0)
      RETURN 0
   ELSE
      return max(w_j + r-compute(q_j), r-compute(j-1))
}
```

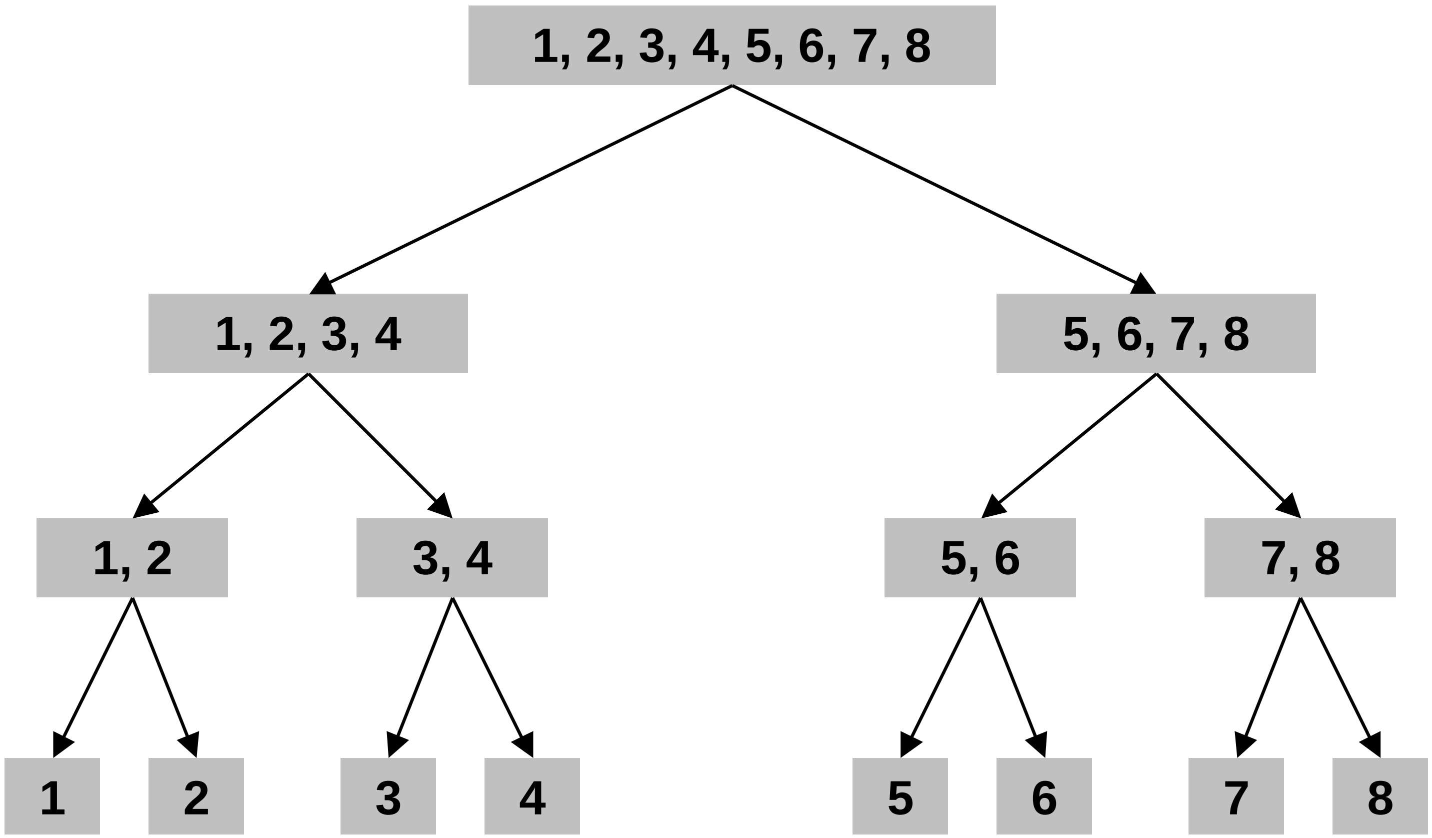
Dynamic Programming Subproblems

Spectacularly redundant subproblems $\Rightarrow$ exponential algorithms.



Divide-and-Conquer Subproblems

Independent subproblems $\Rightarrow$ efficient algorithms.



Weighted Activity Selection: Memoization

Memoized Activity Selection

```
INPUT: N, s_1,…,s_N , f_1,…,f_N , w_1,…,w_N

Sort jobs by increasing finish times so that
f_1 ≤ f_2 ≤ ... ≤ f_N.

Compute q_1, q_2 , ... , q_N

Global array OPT[0..N]
FOR j = 0 to N
   OPT[j] = "empty"

m-compute(j) {
   IF (j = 0)
      OPT[0] = 0
   ELSE IF (OPT[j] = "empty")
      OPT[j] = max(w_j + m-compute(q_j), m-compute(j-1))
   RETURN OPT[j]
}
```

Weighted Activity Selection: Running Time

Claim: memoized version of algorithm takes O(N log N) time.

- **Ordering by finish time: O(N log N).**
- **Computing q_j : O(N log N) via binary search.**

- **`m-compute(j)`: each invocation takes O(1) time and either**
  - **(i) returns an existing value of `OPT[]`**
  - **(ii) fills in one new entry of `OPT[]` and makes two recursive calls**

- **Progress measure Φ = # nonempty entries of `OPT[]`.**
  - ✐ **Initially $\Phi = 0$, throughout $\Phi \leq N$.**
  - ✐ **(ii) increases Φ by 1 $\Rightarrow$ at most 2N recursive calls.**

- **Overall running time of `m-compute(N)` is O(N).**

Weighted Activity Selection: Finding a Solution

`m-compute(N)` determines value of optimal solution.

- Modify to obtain optimal solution itself.

Finding an Optimal Set of Activities

```
ARRAY: OPT[0..N]
Run m-compute(N)

find-sol(j) {
   IF (j = 0)
      output nothing
   ELSE IF (w_j + OPT[q_j] > OPT[j-1])
      print j
      find-sol(q_j)
   ELSE
      find-sol(j-1)
}
```

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

Weighted Activity Selection: Bottom-Up

Unwind recursion in memoized algorithm.

Bottom-Up Activity Selection

```
INPUT: N, s_1,…,s_N , f_1,…,f_N , w_1,…,w_N

Sort jobs by increasing finish times so that
f_1 ≤ f_2 ≤ ... ≤ f_N.

Compute q_1, q_2 , ... , q_N

ARRAY: OPT[0..N]
OPT[0] = 0

FOR j = 1 to N
   OPT[j] = max(w_j + OPT[q_j], OPT[j-1])
```

Dynamic Programming Overview

Dynamic programming.

- **Similar to divide-and-conquer.**
  - solves problem by combining solution to sub-problems
- **Different from divide-and-conquer.**
  - sub-problems are not independent
  - save solutions to repeated sub-problems in table

Recipe.

- **Characterize structure of problem.**
  - optimal substructure property
- **Recursively define value of optimal solution.**
- **Compute value of optimal solution.**
- **Construct optimal solution from computed information.**

Top-down vs. bottom-up.

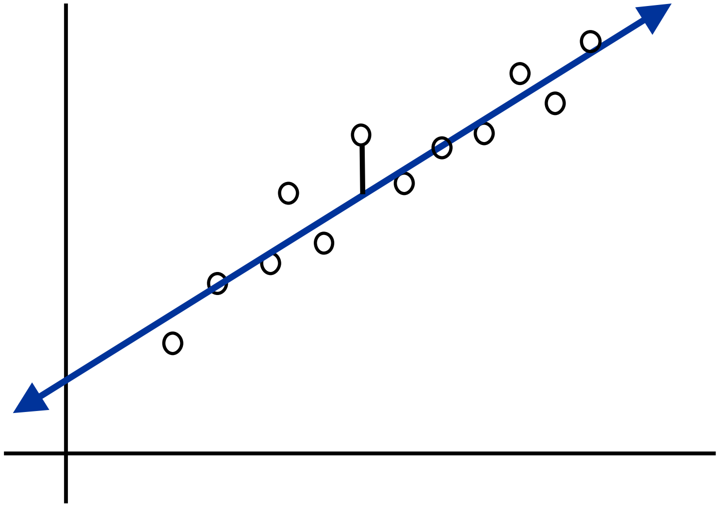
- **Different people have different intuitions.**

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:**

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



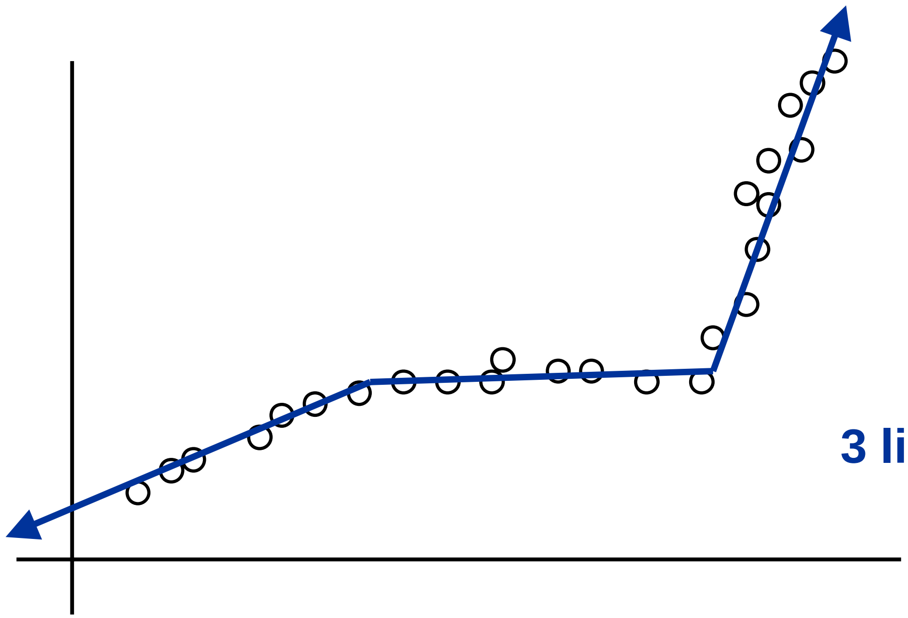
- **Calculus $\Rightarrow$ min error is achieved when:**

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

Segmented Least Squares

Segmented least squares.

- **Points lie roughly on a sequence of 3 lines.**
- **Given N points in the plane $p_1, p_2, \ldots, p_N$, find a sequence of lines that minimize:**
  - **the sum of the sum of the squared errors E in each segment**
  - **the number of lines L**
- **Tradeoff function: e + c L, for some constant c > 0.**



3 lines better than one

Segmented Least Squares: Structure

Notation.

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

Optimal solution:

- Last segment uses points $p_i, p_{i+1}, \ldots, p_j$ for some i.
- Cost = e(i, j) + c + OPT(i-1).

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

New dynamic programming technique.

- Weighted activity selection: binary choice.
- Segmented least squares: multi-way choice.

Segmented Least Squares: Algorithm

Bottom-Up Segmented Least Squares

```
INPUT: N, p1,…,pN , c

ARRAY: OPT[0..N]
OPT[0] = 0

FOR j = 1 to N
   FOR i = 1 to j
      compute the least square error e[i,j] for
      the segment pi,..., pj

   OPT[j] = min1 ≤ i ≤ j (e[i,j] + c + OPT[i-1])

RETURN OPT[N]
```

Running time:

- Bottleneck = computing e(i, n) for $O(N^2)$ pairs, O(N) per pair using previous formula.
- $O(N^3)$ overall.

Segmented Least Squares: Improved Algorithm

A quadratic algorithm.

- Bottleneck = computing e(i, j).
- $O(N^2)$ preprocessing + O(1) per computation.

$$a_{ij} = \frac{n \sum_{k=i}^{j} x_k y_k - \left(\sum_{k=i}^{j} x_k\right)^2 \left(\sum_{k=i}^{j} y_k\right)^2}{n \sum_{k=i}^{j} x_k^2 - \left(\sum_{k=i}^{j} x_k\right)^2}$$

$$b_{ij} = \frac{\sum_{k=i}^{j} y_k - a \sum_{k=i}^{j} x_k}{n}$$

$$n_{ij} = j - i + 1$$

$$xs_k = \sum_{k=1}^{i} x_k \qquad ys_k = \sum_{k=1}^{i} y_k$$

$$xxs_k = \sum_{k=1}^{i} x_k^2 \qquad yys_k = \sum_{k=1}^{i} y_k^2$$

$$xy_k = \sum_{k=1}^{i} x_k \, y_k$$

Preprocessing

$$\sum_{k=i}^{j} x_k = xs_j - xs_{i-1}$$

$$\begin{aligned} e(i,j) &= \sum_{k=i}^{j} (y_k - a x_k - b)^2 \\ &= (yys_j - yys_{i-1}) + \cdots \end{aligned}$$

Knapsack Problem

Knapsack problem.

- Given N objects and a "knapsack."
- Item i weighs $w_i > 0$ Newtons and has value $v_i > 0$.
- Knapsack can carry weight up to W Newtons.
- Goal: fill knapsack so as to maximize total value.

v_i / w_i (↓)

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

W = 11

Greedy = 35: { 5, 2, 1 }

OPT value = 40: { 3, 4 }

Knapsack Problem: Structure

OPT(n, w) = max profit subset of items {1, . . . , n} with weight limit w.

- **Case 1: OPT selects item n.**
  - **new weight limit = $w - w_n$**
  - **OPT selects best of {1, 2, . . . , n – 1} using this new weight limit**
- **Case 2: OPT does not select item n.**
  - **OPT selects best of {1, 2, . . . , n – 1} using weight limit w**

$$OPT(n,w) = \begin{cases} 0 & \text{if } n = 0 \\ OPT(n-1,w) & \text{if } w_n > w \\ \max\{OPT(n-1,w),\ \ v_n + OPT(n-1,w-w_n)\} & \text{otherwise} \end{cases}$$

New dynamic programming technique.

- **Weighted activity selection: binary choice.**
- **Segmented least squares: multi-way choice.**
- **Knapsack: adding a new variable.**

Knapsack Problem: Bottom-Up

Bottom-Up Knapsack

```
INPUT: N, W, w_1,…,w_N, v_1,…,v_N

ARRAY: OPT[0..N, 0..W]

FOR w = 0 to W
   OPT[0, w] = 0

FOR n = 1 to N
   FOR w = 1 to W
      IF (w_n > w)
         OPT[n, w] = OPT[n-1, w]
      ELSE
         OPT[n, w] = max {OPT[n-1, w], v_n + OPT[n-1, w-w_n ]}

RETURN OPT[N, W]
```

Knapsack Algorithm

W + 1 →

N + 1 ↓

Weight Limit	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	35	40

Item	Value	Weight
1	1	1
2	6	2
3	8	5
4	22	6
5	28	7

Knapsack Problem: Running Time

Knapsack algorithm runs in time O(NW).

- **Not polynomial in input size!**
- **"Pseudo-polynomial."**
- **Decision version of Knapsack is "NP-complete."**
- **Optimization version is "NP-hard."**

Knapsack approximation algorithm.

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

Sequence Alignment

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

5 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: Applications

Applications.

- **Spell checkers / web dictionaries.**
  - `ocurrance`
  - `occurrence`
- **Computational biology.**
  - `ctgacctacct`
  - `cctgactacat`

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

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

Edit distance.

- **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

$2\delta + \alpha_{CA}$

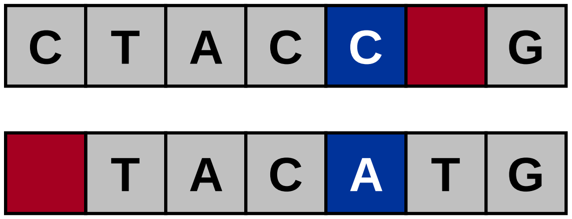
Sequence Alignment

Problem.

- Input: two strings $X = x_1\ x_2\ .\ .\ .\ x_M$ and $Y = y_1\ y_2\ .\ .\ .\ y_N$.
- Notation: {1, 2, . . . , M} and {1, 2, . . . , N} denote positions in X, Y.
- Matching: set of ordered pairs (i, j) such that each item occurs in at most one pair.
- Alignment: matching with no crossing pairs.
  - if $(i, j) \in M$ and $(i', j') \in M$ and $i < i'$, then $j < j'$

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

- Example: CTACCG vs. TACATG.
  - M = { (2,1) (3,2) (4,3), (5,4), (6,6) }



- Goal: find alignment of minimum cost.

Sequence Alignment: Problem Structure

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 (i, j).**
  - pay mismatch for (i, 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 m unmatched.**
  - pay gap for 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 n unmatched.**
  - pay gap for j and min cost of aligning $x_1\ x_2 \ldots x_i$ and $y_1\ y_2 \ldots y_{j-1}$

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

Sequence Alignment: Algorithm

O(MN) time and space.

Bottom-Up Sequence Alignment

```
INPUT: M, N, x1x2...xM, y1y2...yN, δ, α

ARRAY: OPT[0..M, 0..N]

FOR i = 0 to M
   OPT[0, i] = iδ
FOR j = 0 to N
   OPT[j, 0] = jδ

FOR i = 1 to M
   FOR j = 1 to N
      OPT[i, j] = min(α[xi, yj] + OPT[i-1, j-1],
                      δ + OPT[i-1, j],
                      δ + OPT[i, j-1])
RETURN OPT[M, N]
```

Sequence Alignment: Linear Space

Straightforward dynamic programming takes $\Theta(MN)$ time and space.

- English words or sentences $\Rightarrow$ may not be a problem.
- Computational biology $\Rightarrow$ huge problem.
  - $M = N = 100,000$
  - 10 billion ops OK, but 10 gigabyte array?

Optimal value in $O(M + N)$ space and $O(MN)$ time.

- Only need to remember OPT(i - 1, •) to compute OPT(i, •).
- Not clear how to recover optimal alignment itself.

Optimal alignment in $O(M + N)$ space and $O(MN)$ time.

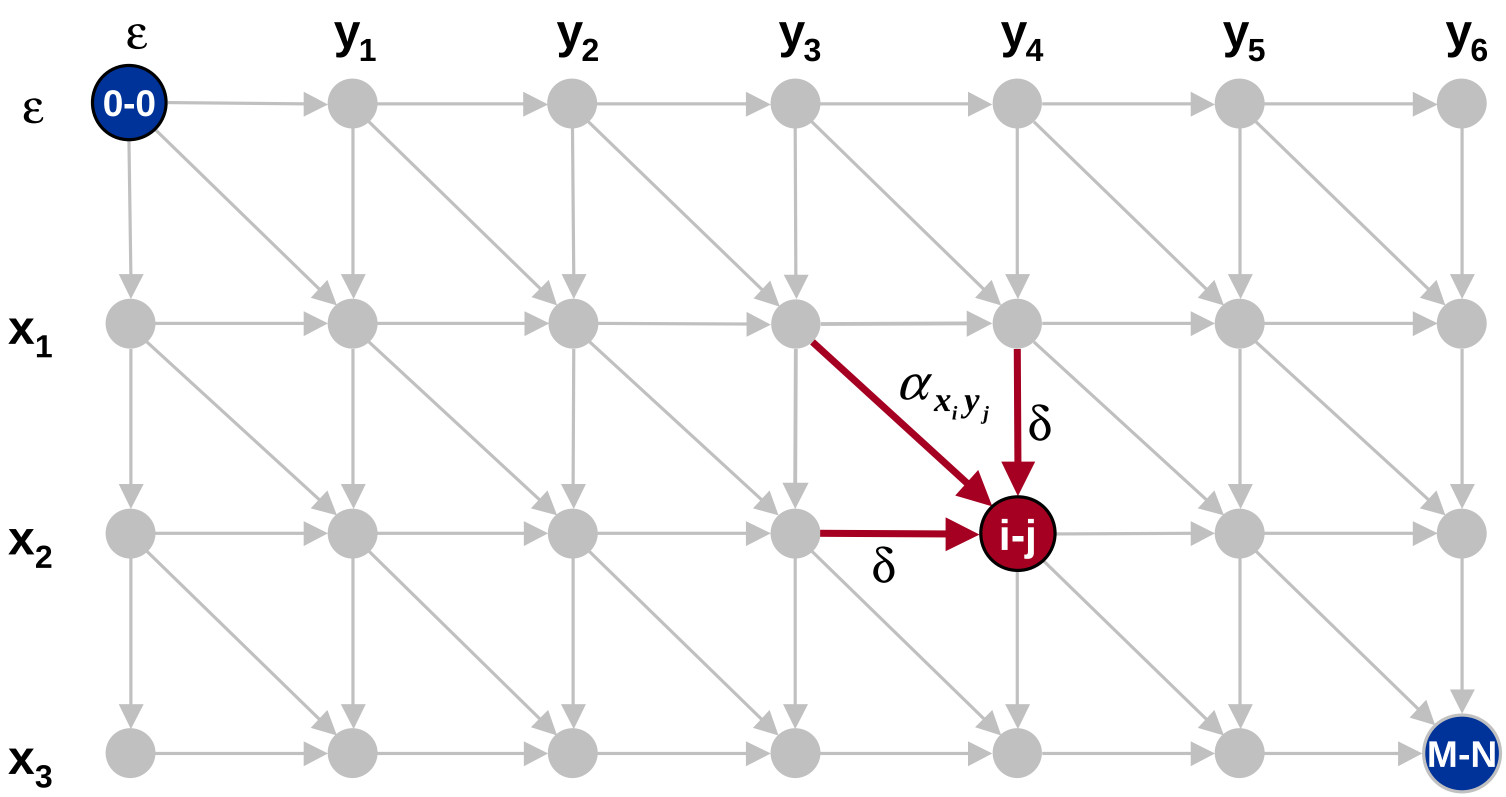
- Clever combination of divide-and-conquer and dynamic programming.

Sequence Alignment: Linear Space

Consider following directed graph (conceptually).

- **Note: takes Θ(MN) space to write down graph.**

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

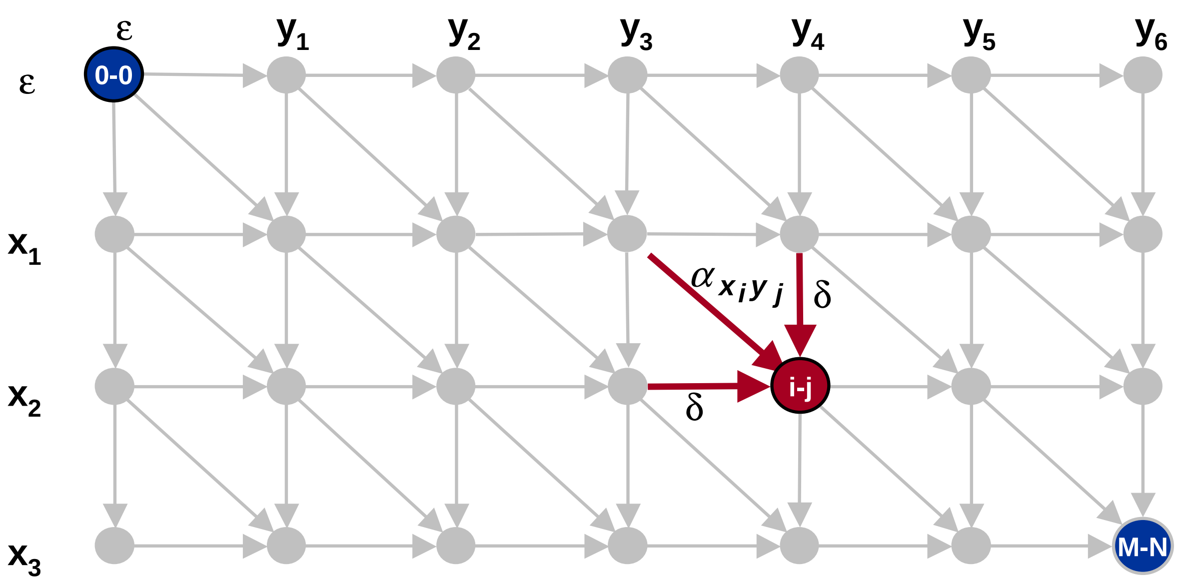


Sequence Alignment: Linear Space

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

- Base case: f(0, 0) = OPT(0, 0) = 0.
- Inductive step: assume f(i', j') = OPT(i', j') for all i' + j' < i + j.
- Last edge on path to (i, j) is either from (i-1, j-1), (i-1, j), or (i, j-1).

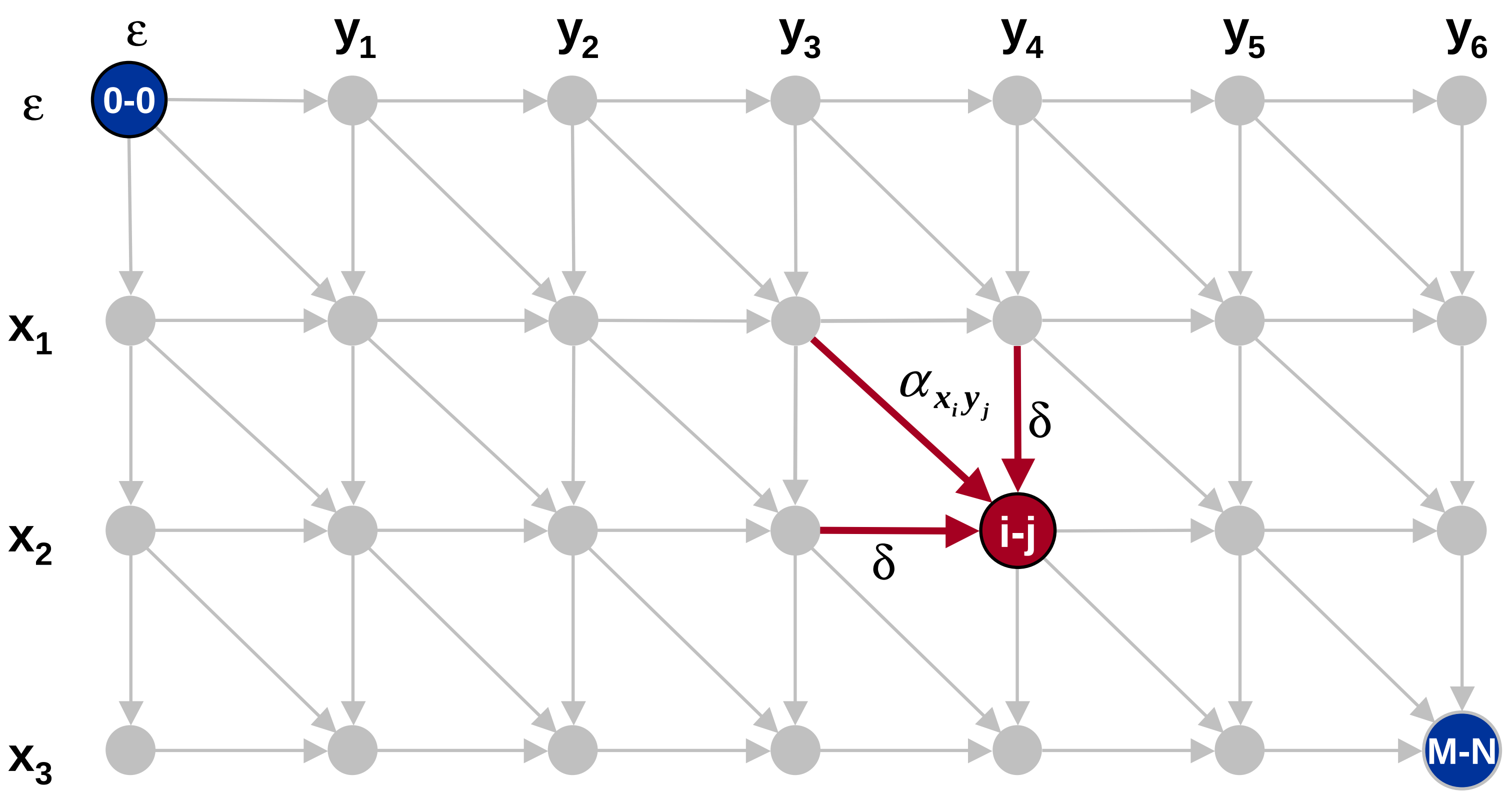
$$\begin{aligned} f(i,j) &= \min\ \{\alpha_{x_i y_j} + f(i-1, j-1),\ \delta + f(i-1, j),\ \delta + f(i, j-1)\} \\ &= \min\ \{\alpha_{x_i y_j} + OPT(i-1, j-1),\ \delta + OPT(i-1, j),\ \delta + OPT(i, j-1)\} \\ &= OPT(i,j) \end{aligned}$$



Sequence Alignment: Linear Space

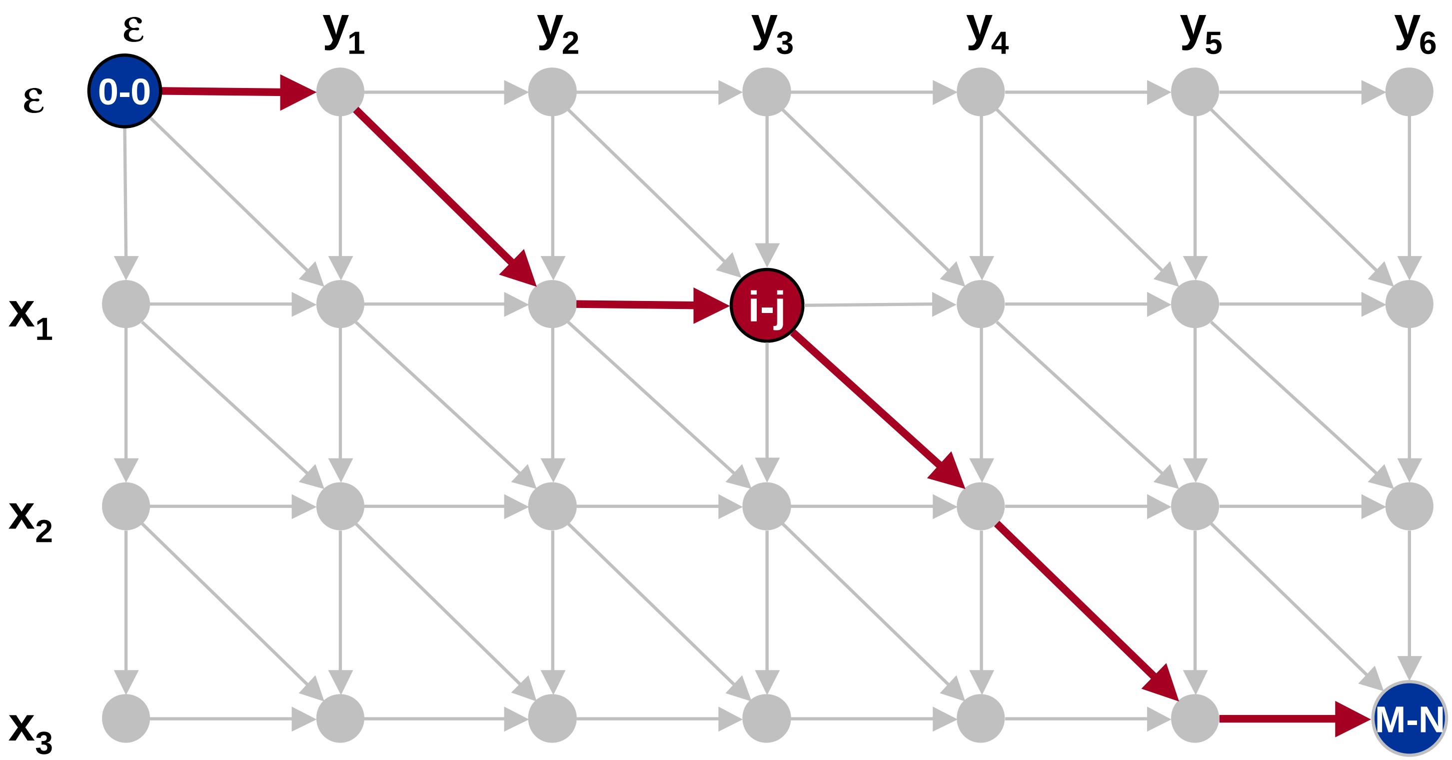
Let g(i, j) be shortest path from (i, j) to (M, N).

- Can compute in O(MN) time for all (i, j) by reversing arc orientations and flipping roles of (0, 0) and (M, N).



Sequence Alignment: Linear Space

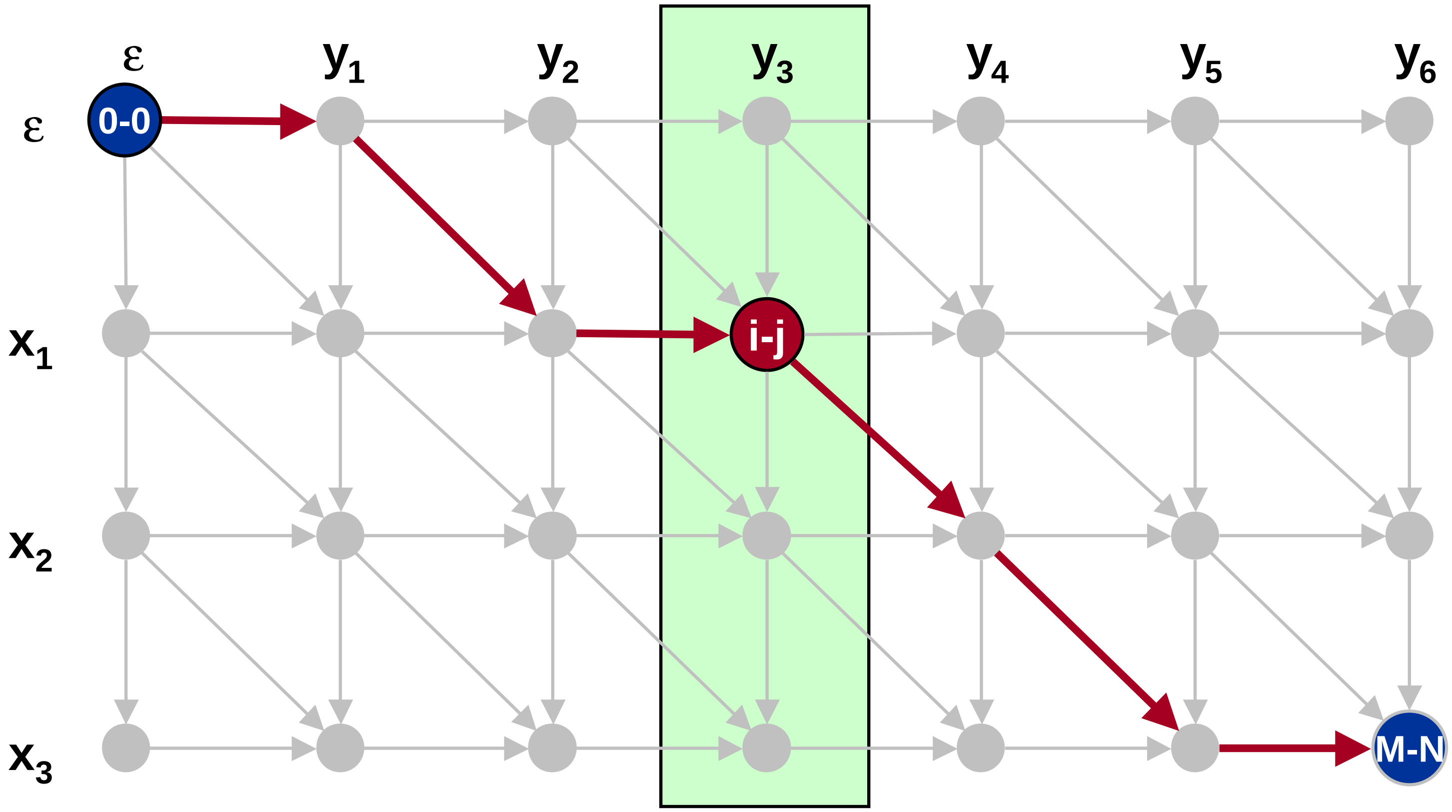
Observation 1: the cost of the shortest path that uses (i, j) is f(i, j) + g(i, j).



Sequence Alignment: Linear Space

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

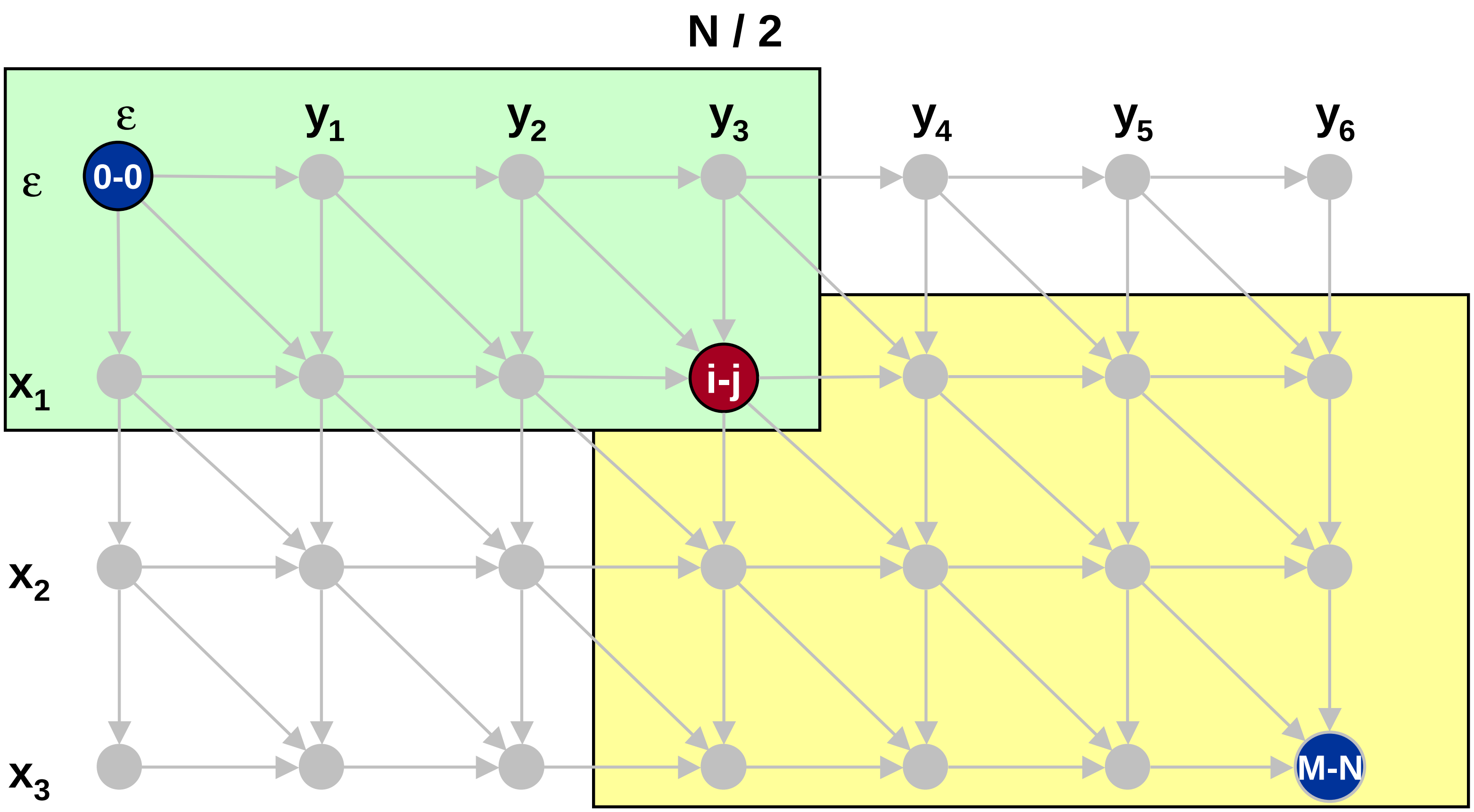
Observation 2: let q be an index that minimizes f(q, N/2) + g(q, N/2). Then, the shortest path from (0, 0) to (M, N) uses (q, N/2).



Sequence Alignment: Linear Space

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

Conquer: recursively compute optimal alignment in each "half."



Sequence Alignment: Linear Space

T(m, n) = max running time of algorithm on strings of length m and n.

Theorem. T(m, n) = O(mn).

- O(mn) work to compute f (• , n / 2) and g (• , n / 2).
- O(m + n) to find best index q.
- T(q, n / 2) + T(m - q, n / 2) work to run recursively.
- Choose constant c so that:

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

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

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