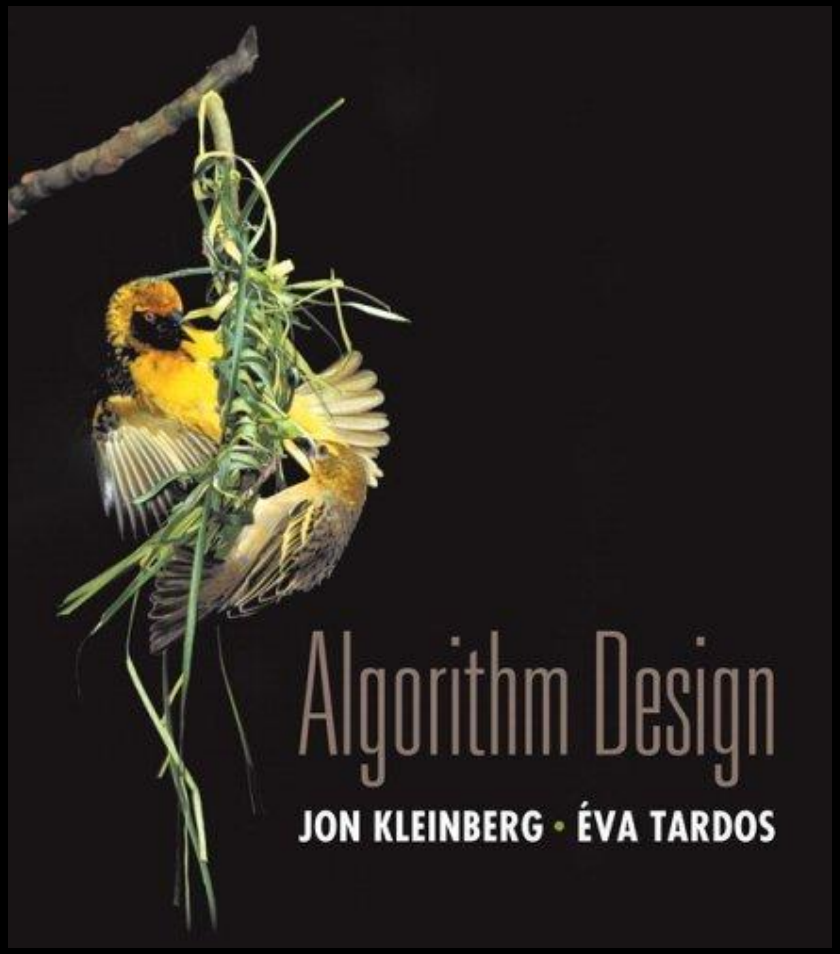


Chapter 6

Dynamic Programming

Algorithmic Paradigms

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

Divide-and-conquer. Break up a problem into two 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. Pioneered the systematic study of dynamic programming in the 1950s.

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, systems, ….

Some famous dynamic programming algorithms.

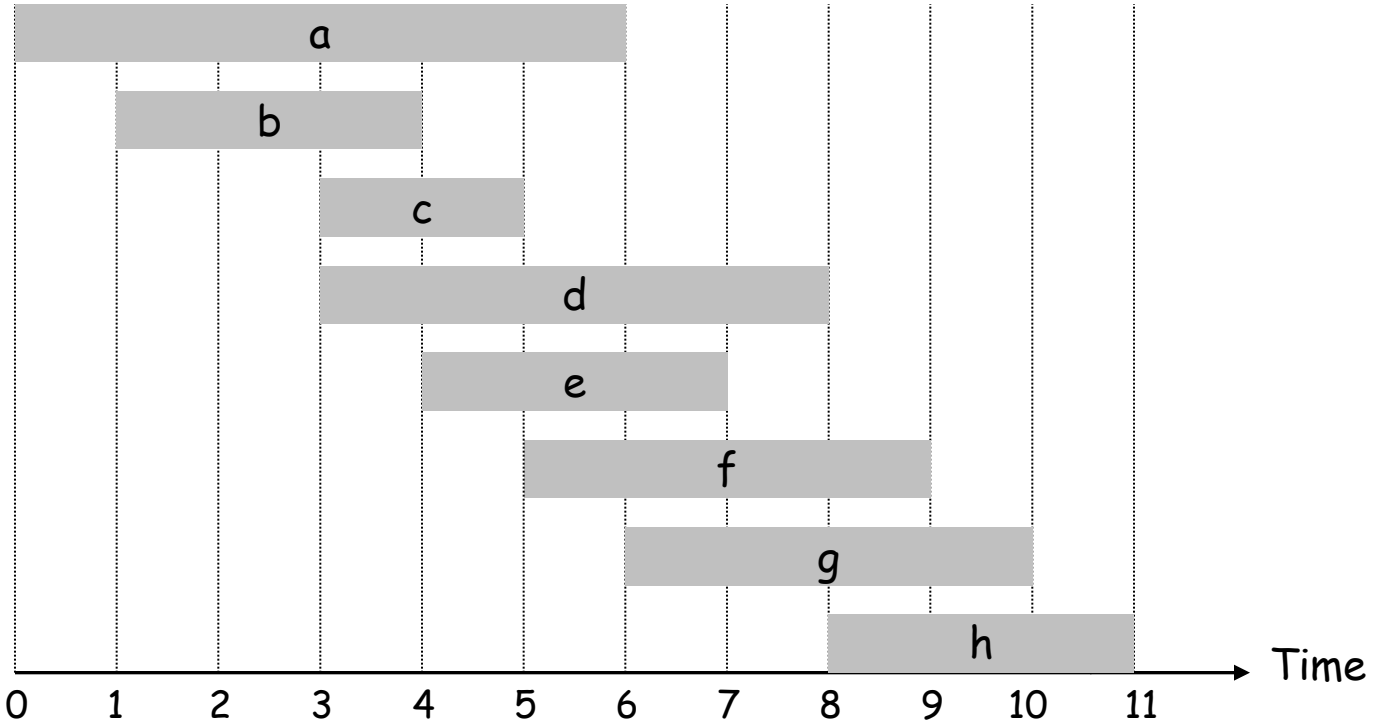
- Viterbi for hidden Markov models.
- Unix diff for comparing two files.
- Smith-Waterman for sequence alignment.
- Bellman-Ford for shortest path routing in networks.
- Cocke-Kasami-Younger for parsing context free grammars.

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.

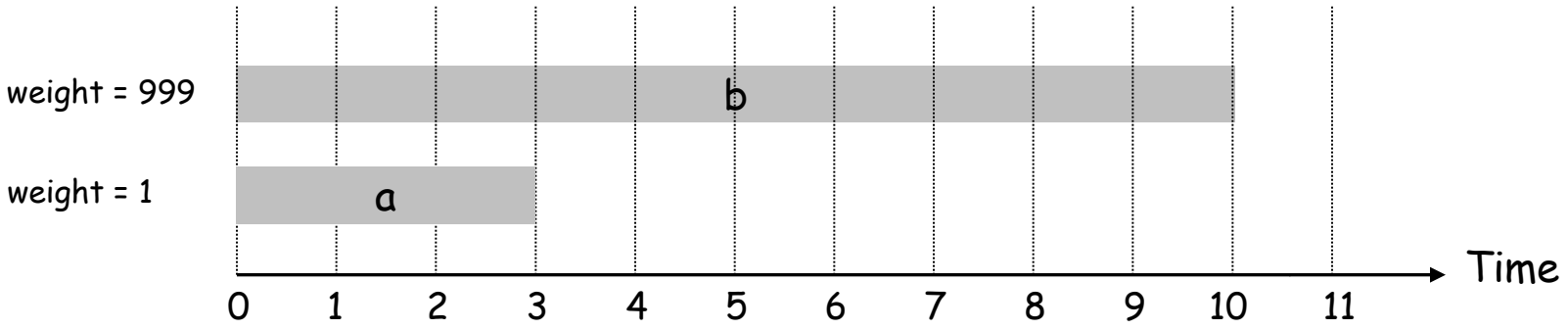


Unweighted Interval Scheduling Review

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.

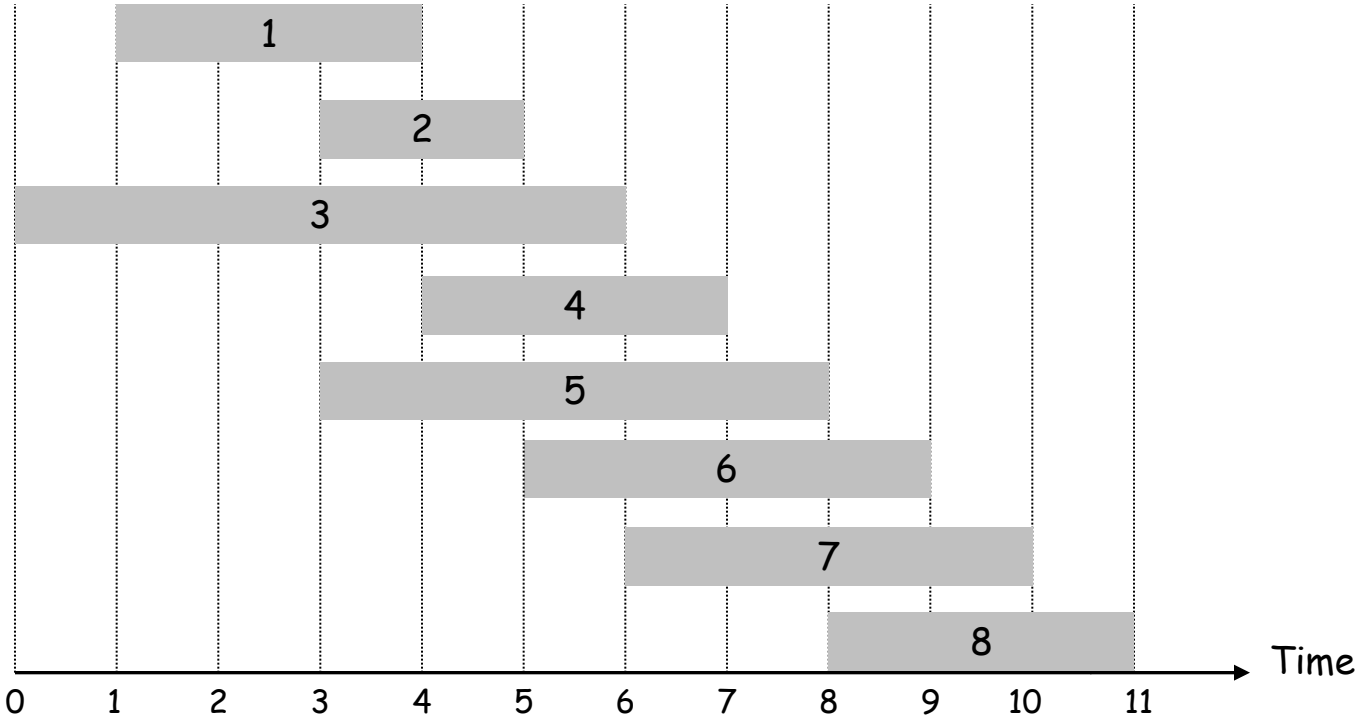


Weighted Interval Scheduling

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.
  - 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

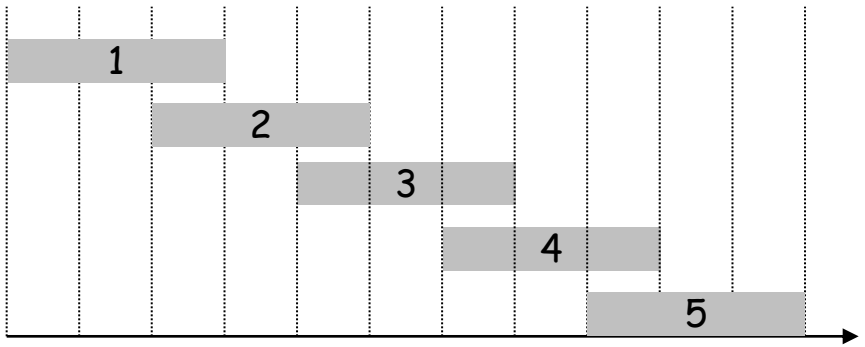
Brute force algorithm.

```
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))
}
```

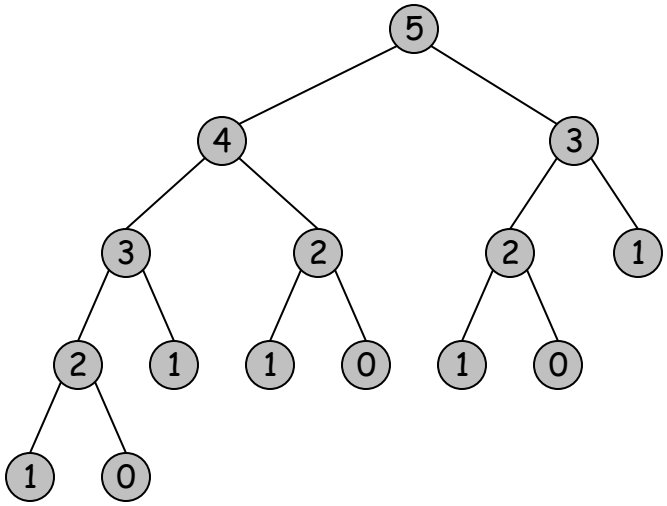
Weighted Interval Scheduling: Brute Force

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 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)

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

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

Weighted Interval Schedule

- Análise Justa

- Complexidade = soma dos custos de todas as chamadas
- Vamos analisar quantas chamadas M-Opt(i) faz
  - Na primeira vez que M-OPT(i) é chamada, ela faz duas chamadas.
  - Nas demais vezes M-OPT(i) não realiza chamadas
  - Portanto, M-OPT(i) faz ao todo 2 chamadas

- O total de chamadas é O(n) e cada chamada tem custo O(1)

- Portanto, o algoritmo gasta O(n).

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

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])
}
```

Maior subsequência crescente

Maior subsequência crescente

Entrada:

A= (a(1), a(2),…., a(n)) uma sequência de números reais distintos.

Objetivo:

Encontrar a maior subsequência crescente A

- Exemplo A= (2, 3, 14, 5, 9, 8, 4)

  - (2,3,8) e (3,14) são subsequências crescentes de tamanho 3

  - A maior subsequência crescente de A é 2,3,5,9

Maior subsequência crescente

- Seja L(i) : tamanho da maior subsequência crescente que termina em a(i) (a(i) pertence a subsequencia)
-
- Exemplo A= (2, 3, 14, 5, 9, 8, 4)
- L(1)=1, L(2)=2, L(3)=3, L(4)=3, L(5)=4,L(6)=4,L(7)=3

- O tamanho da maior subsequência crescente é

$$\max \{ L(1), L(2), \ldots, L(n) \}$$

- Temos a seguinte equação para L(j)

$$L(j) = \max_i \{ 1+L(i) \mid i < j \text{ e } a(j) > a(i) \} \text{ para } j>1$$
$$L(1)=1$$

Maior subsequência crescente: Encontrando o tamanho

```
Input: n, a1,…,an ,

For j=1 to n
  L(j)←1, pre(j)←0
  For i=1 to j-1
        If A(i) < A(j) and 1+L(i)>L(j) then
            L(j) ← 1+L(i)
            pre(j)←i                          %(atualiza predecessor)
        End If
  End For
End For

MSC ← 0
For i=1 to n                                  %(encontra maior L)
  MSC ← max{ MSC, L(i) }
End For
```

- pre(j): é utilizado para guardar o predecessor de j na maior subsequência crescente
- Complexidade $O(n^2)$

Encontrando a subsequencia (Recursivamente)

```
Input: n, L(1),…,L(n),pre(1),…,pre(n),MSC,

j←0
While L(j)<> MSC
   j++
End While
Find_Subsequence(j)

Proc Find_Subsequence(j)
   if j=0
     Return
   else
      Add j to the solution;
      Find_Subsequence(pre(j))
   End if
  Return
```

Complexidade $O(n)$

Encontrando a subsequencia (iterativamente)

```
Input: n, L(1),…,L(n),pre(1),…,pre(n),MSC,

j←0
While L(j)<> MSC
   j++
End While

While j>0
  Add j to the solution;
  j← pre(j)
End While
```

Complexidade $O(n)$

Maior subsequência crescente

- Exercícios

  - Criar uma versão recursiva que permita encontrar o tamanho da maior subsequencia crescente

  * Provar que toda sequencia tem uma subsequência crescente de tamanho $n^{0.5}$ ou uma subsequência decrescente de tamanho $n^{0.5}$

6.4 Knapsack Problem

Knapsack Problem

Knapsack problem.

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

Ex: { 3, 4 } has value 40.

W = 11

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

.

Knapsack Problem: Greedy Attempt

Greedy: repeatedly add item with maximum ratio v_i / w_i.

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

W = 11

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

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!

Dynamic Programming: Adding a New Variable

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

- Case 1: OPT does not select item i.
  - OPT selects best of { 1, 2, …, i-1 } using weight limit w

- Case 2: OPT selects item i.
  - new weight limit = $w - w_i$
  - OPT selects best of { 1, 2, …, i–1 } using this new weight limit

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

Knapsack Problem: Bottom-Up

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

```
Input: n, w_1,…,w_N, v_1,…,v_N

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

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

return M[n, W]
```

Knapsack Algorithm

W + 1 →

n + 1 ↓

	0	1	2	3	4	5	6	7	8	9	10	11
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 polynomial algorithm that produces a feasible solution that has value within 0.01% of optimum. [Section 11.8]

Problema da Mochila

- Exercícios: Escrever pseudo-códigos para

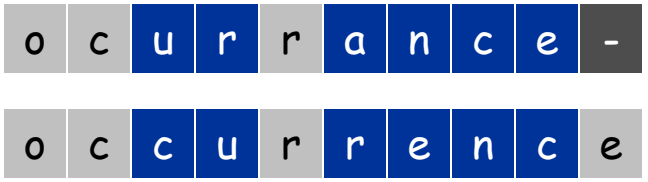
  - Obter os itens que devem ir na mochila dado que o vetor M já está preenchido.
  - Calcular M[n,w] utilizando espaço O(W) em vez de O(nW)
  - Criar uma versão recursiva do algoritmo.

6.6 Sequence Alignment

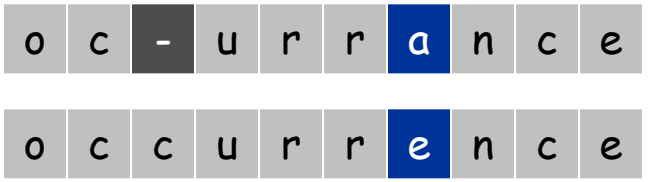
String Similarity

How similar are two strings?

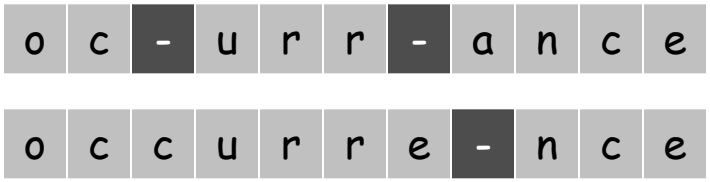
- **ocurrance**
- **occurrence**



6 mismatches, 1 gap



1 mismatch, 1 gap



0 mismatches, 3 gaps

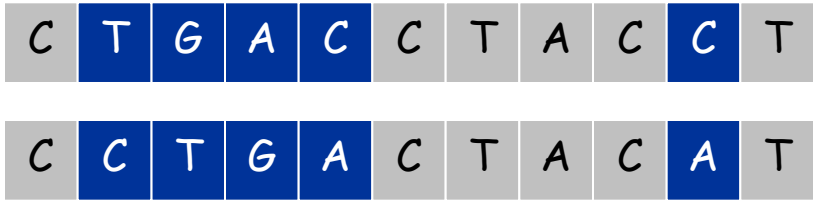
Edit Distance

Applications.

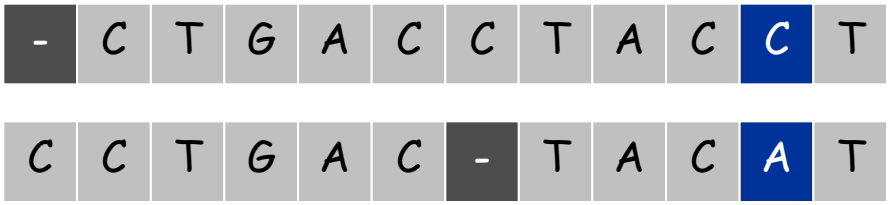
- Basis for Unix diff.
- Speech recognition.
- Computational biology.

Edit distance. [Levenshtein 1966, Needleman-Wunsch 1970]

- Gap penalty δ; mismatch penalty α_{pq}.
- Cost = sum of gap and mismatch penalties.



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



$2\delta + \alpha_{CA}$

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.

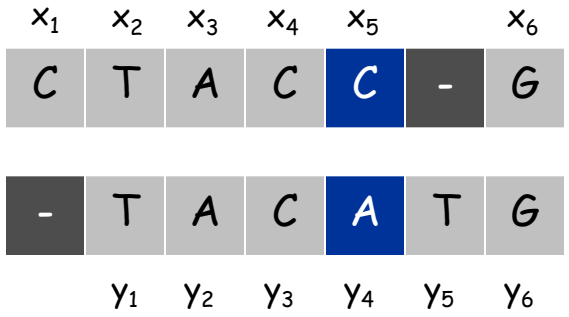
Def. An alignment M is a set of ordered pairs x_i-y_j such that each item occurs in at most one pair and no crossings.

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: $M = x_2$-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 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: 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.

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

Computational biology: $m = n = 100{,}000$. 10 billions ops OK, but 10GB array?

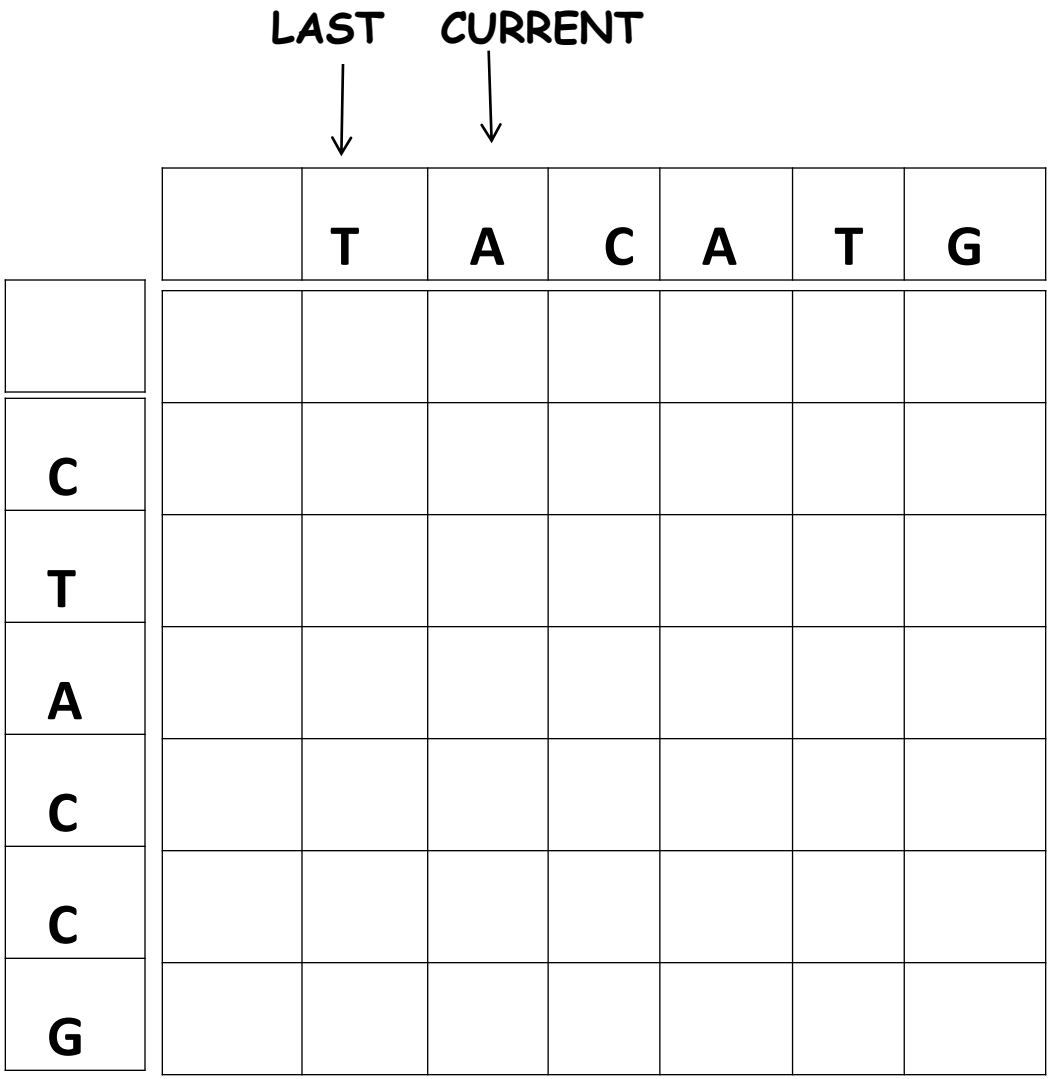
6.7 Sequence Alignment in Linear Space

Sequence Alignment: Value of OPT with Linear Space

```
Sequence-Alignment(m, n, x1x2...xm, y1y2...yn, δ, α) {

   for i = 0 to m
      CURRENT[i] = iδ

   for j = 1 to n
      LAST←CURRENT ( vector copy)
      CURRENT[0]← jδ
      for i = 1 to m
         CURRENT[i] ← min(α[xi, yj] + LAST[i-1],
                          δ + LAST[i],
                          δ + CURRENT[i-1] )
return CURRENT[m]
}
```

- Two vectors of of m positions: LAST e CURRENT
- O(mn) time and O(m+n) space

Sequence Alignment: Value of OPT with Linear Space



Sequence Alignment: Algorithm for recovering the sequence

```
Find_Sequence (i, j, x1x2...xm, y1y2...yn, δ, α) {
   If i=0 or j=0 return
   Else
        If M[i, j] = α[xi, yj] + M[i-1, j-1]
              Add pair xi - yj to the solution
              Return Find_Sequence(i-1,j-1)
        Else If M[i,j]= δ + M[i-1, j]
                Return Find_sequence(i-1,j)
        Else return Find_sequence(i,j-1)
}
```

Analysis. $\Theta(mn)$ space and $O(m+n)$ time

Sequence Alignment: Linear Space

Q. Can we avoid using quadratic 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.

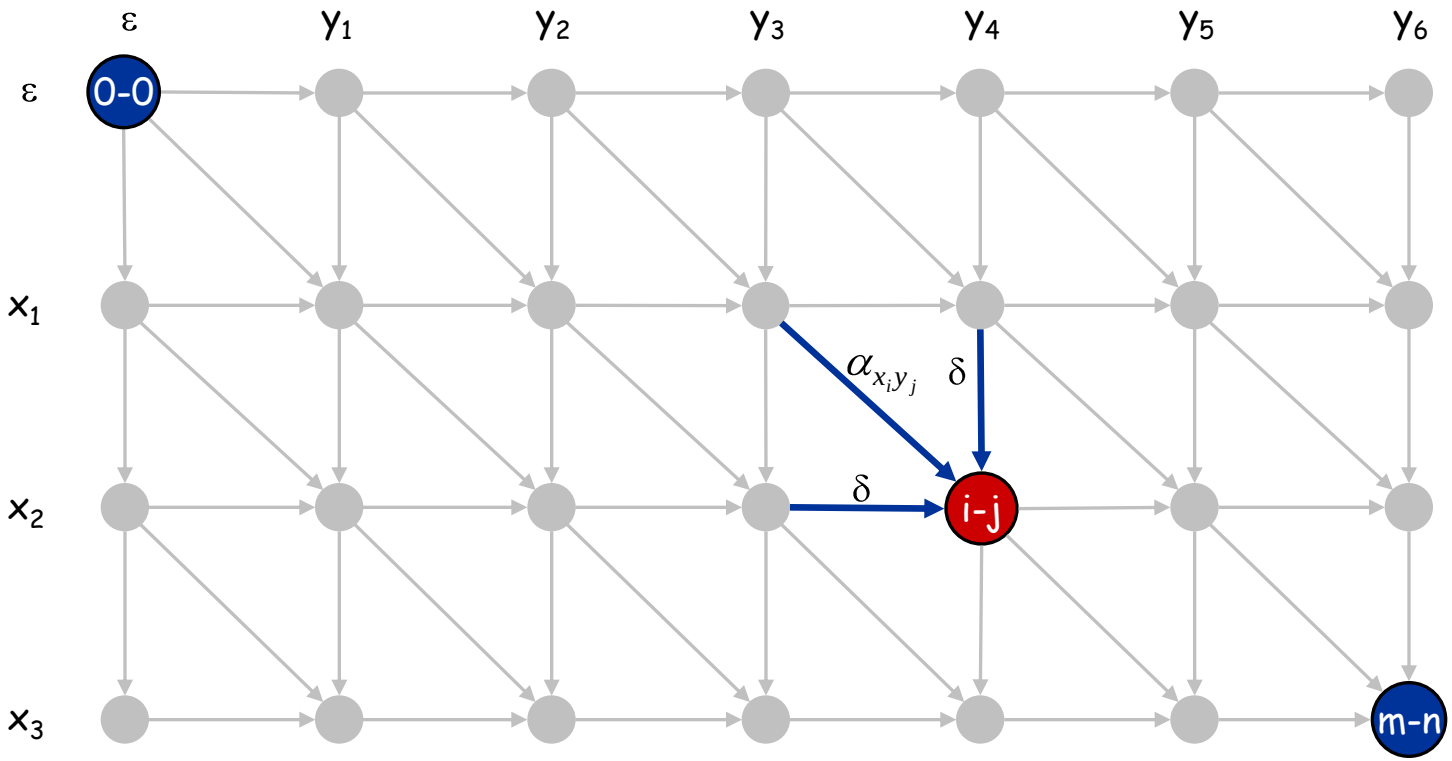
Theorem. [Hirschberg 1975] Optimal alignment in $O(m + n)$ space and $O(mn)$ time.

- Clever combination of divide-and-conquer and dynamic programming.
- Inspired by idea of Savitch from complexity theory.

Sequence Alignment: Linear Space

Edit distance graph.

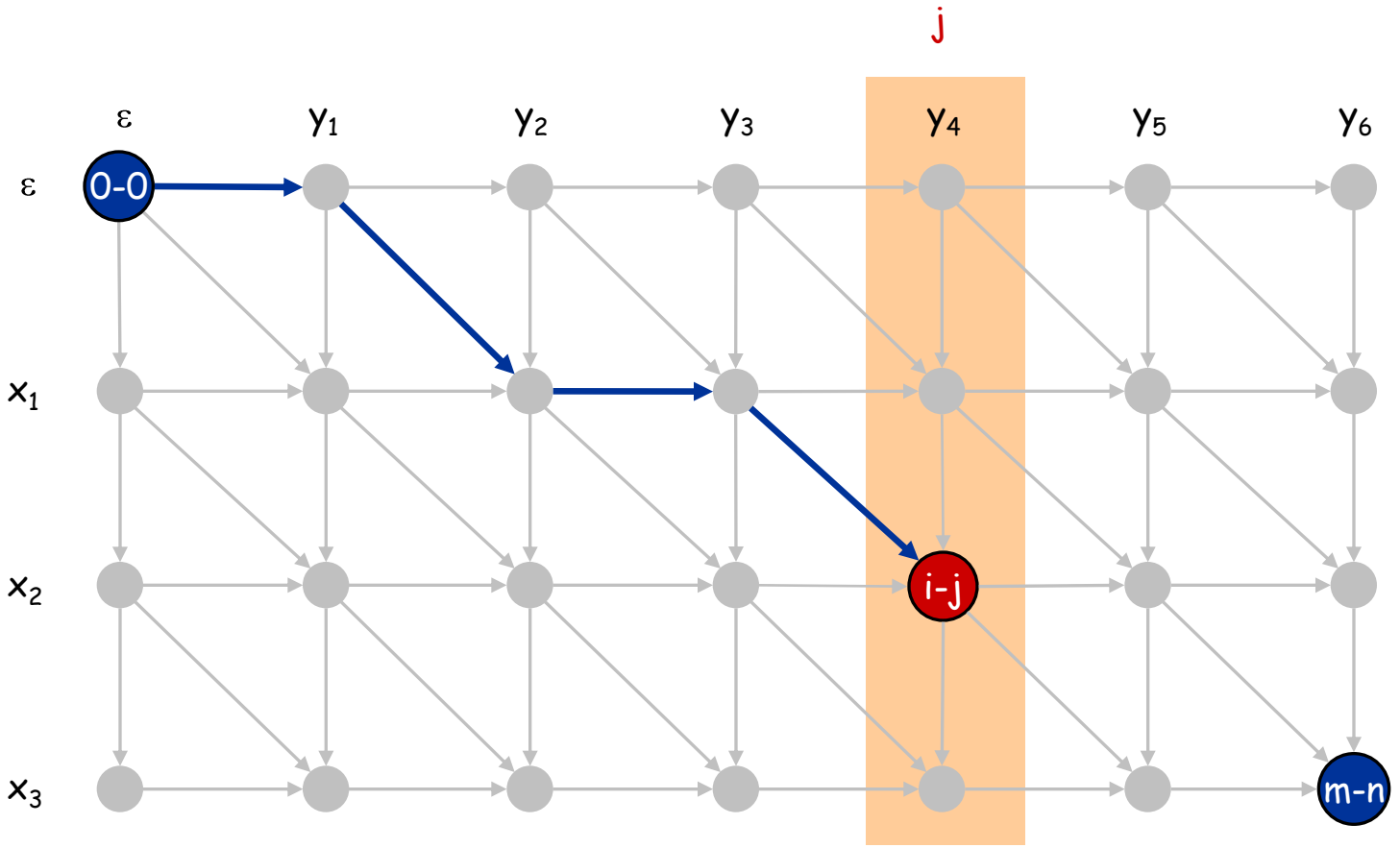
- Let f(i, j) be shortest path from (0,0) to (i, j).
- Observation: f(i, j) = OPT(i, j).



Sequence Alignment: Linear Space

Edit distance graph.

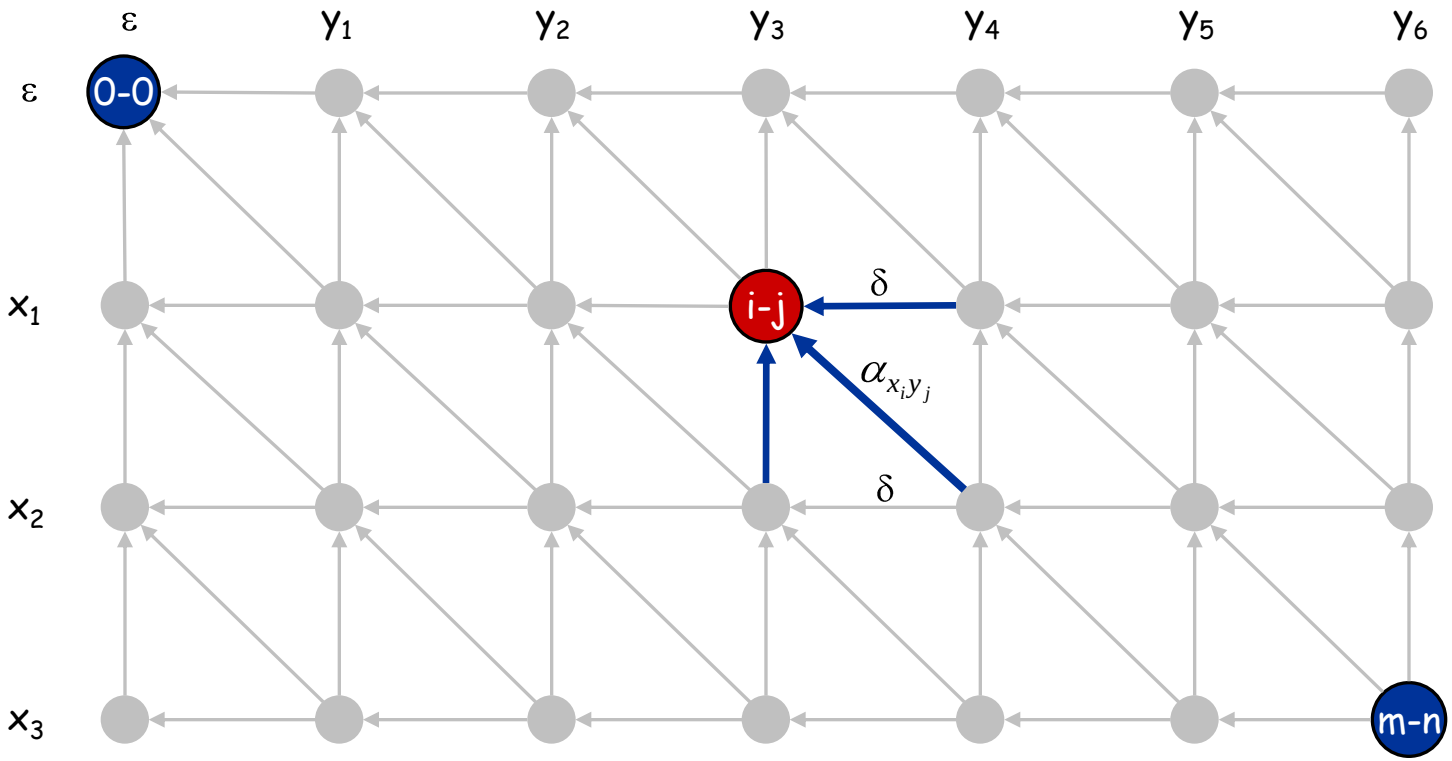
- Let f(i, j) be shortest path from (0,0) to (i, j).
- Can compute f (•, j) for any j in O(mn) time and O(m + n) space.



Sequence Alignment: Linear Space

Edit distance graph.

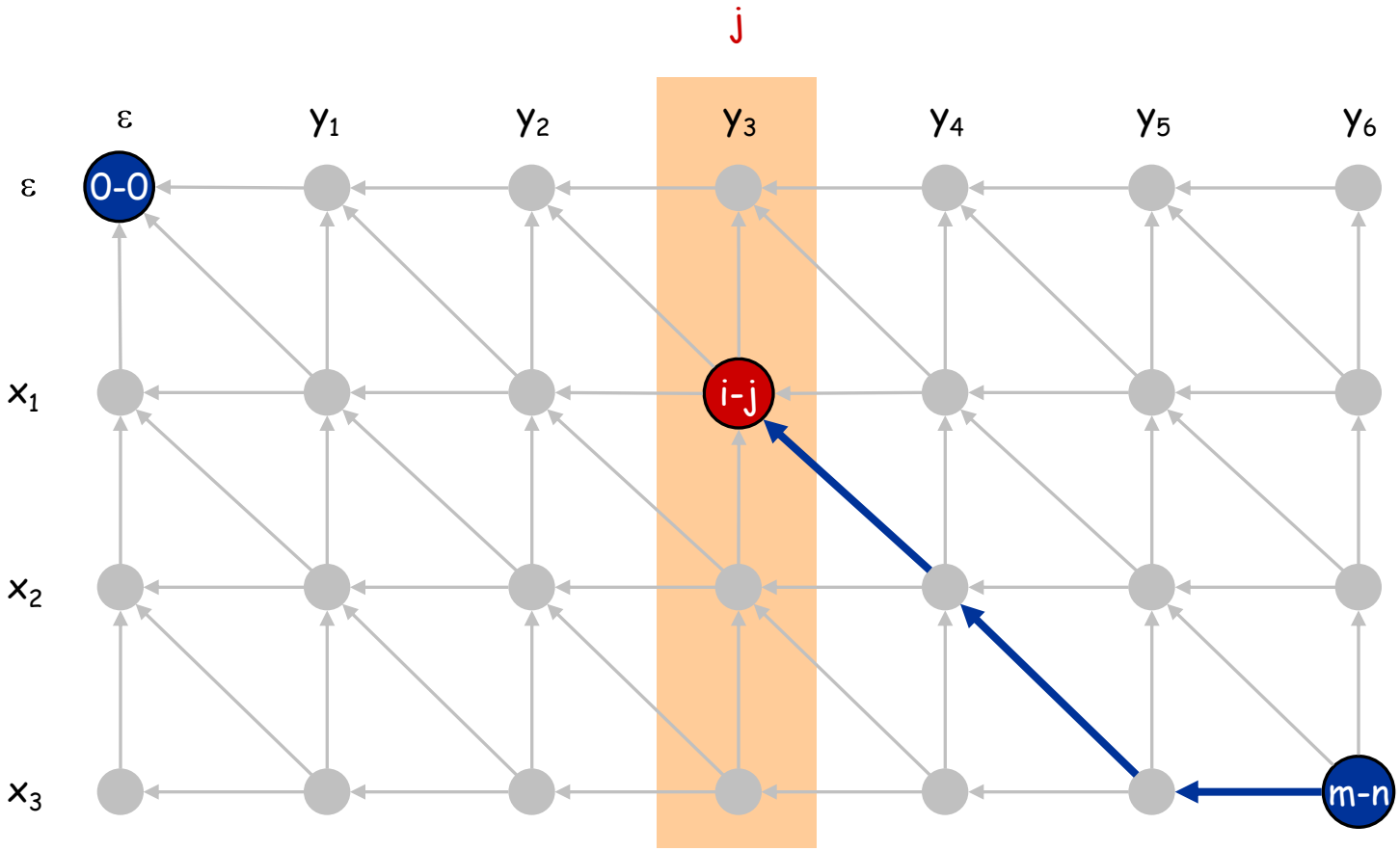
- Let g(i, j) be shortest path from (i, j) to (m, n).
- Can compute by reversing the edge orientations and inverting the roles of (0, 0) and (m, n).



Sequence Alignment: Linear Space

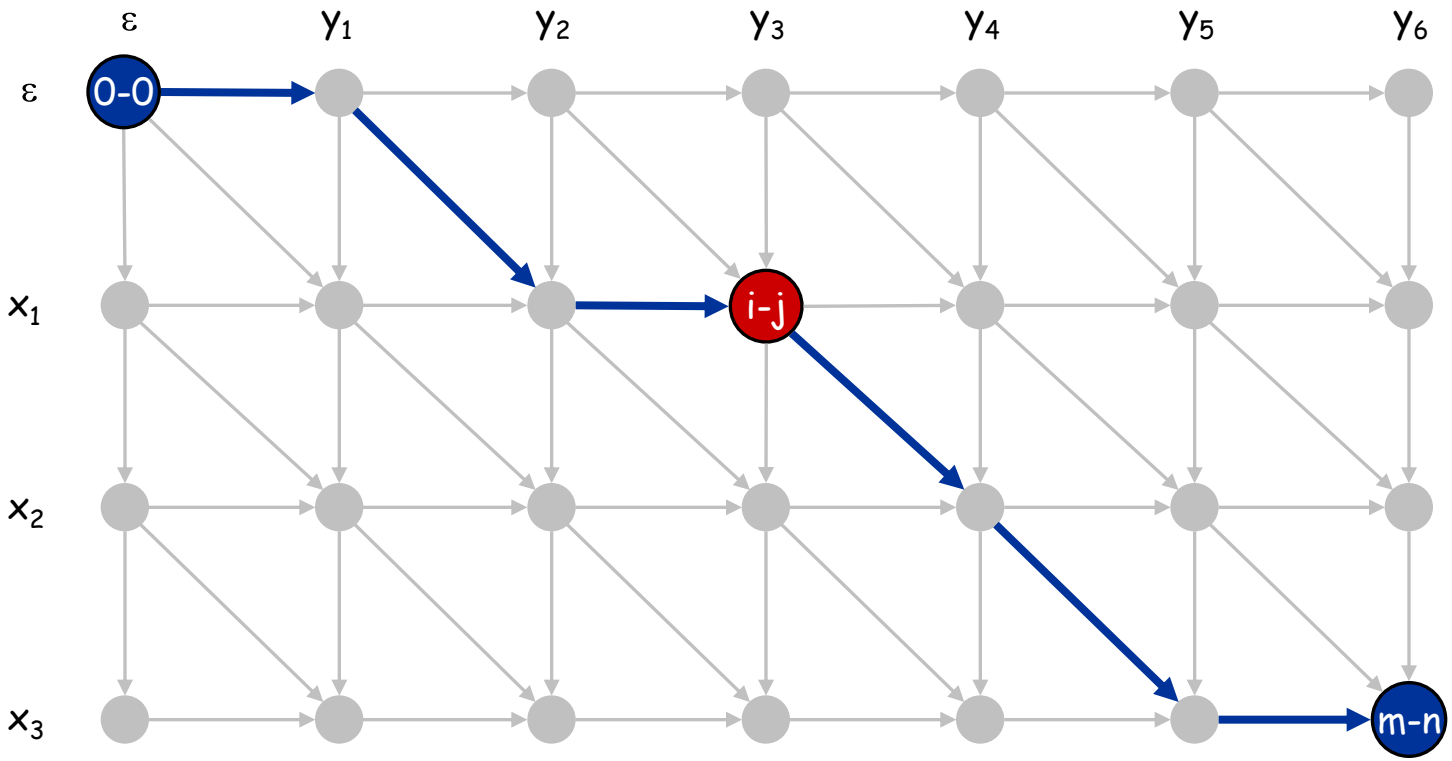
Edit distance graph.

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



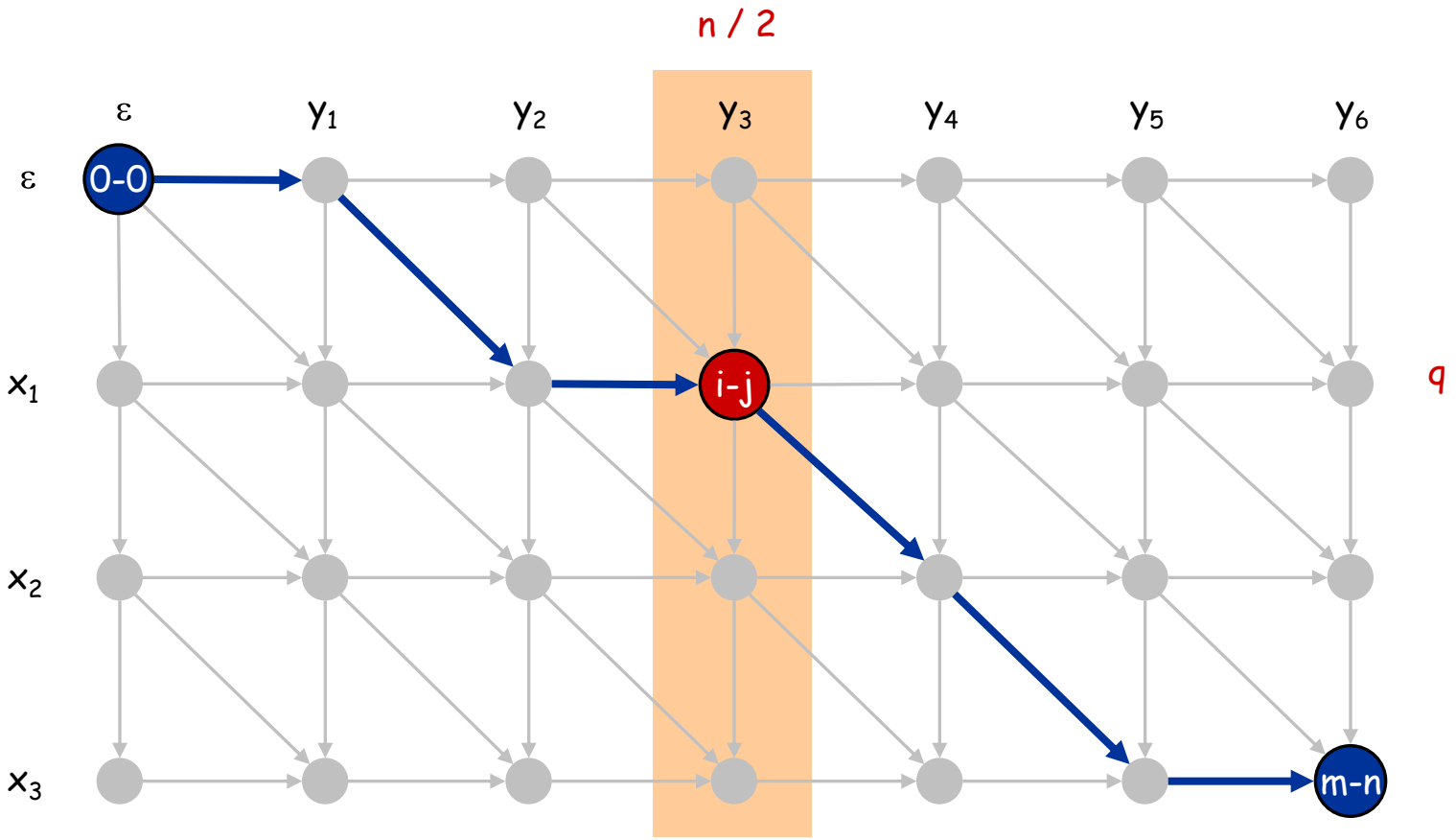
Sequence Alignment: Linear Space

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



Sequence Alignment: Linear Space

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

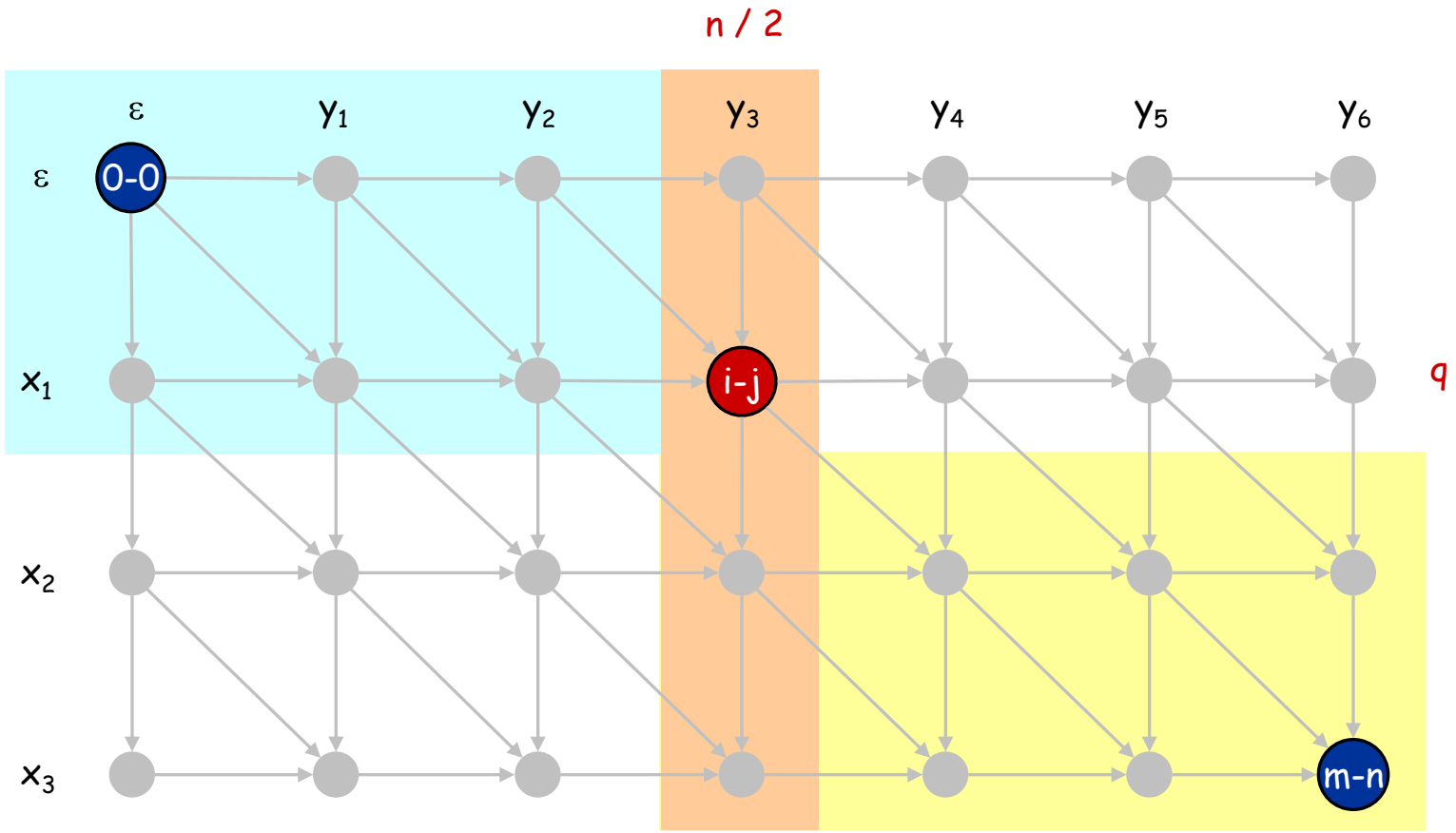


Sequence Alignment: Linear Space

Divide: find some 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.



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. Analysis is not tight because two sub-problems are of size (q, n/2) and (m - q, n/2). In next slide, we save 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) ≤ 2cmn.

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