Objectives

Dynamic Programming: Computational Biology Applications

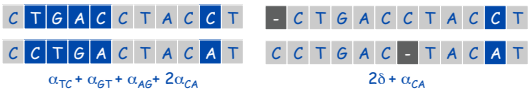
- Sequence Alignment in Linear Space

Edit Distance

[Levenshtein 1966, Needleman-Wunsch 1970]

- Gap penalty: δ
- Mismatch penalty: α_{pq}
  - If p and q are the same, then mismatch penalty is 0
- **Cost** = sum of gap and mismatch penalties

Parameters allow us to tweak cost



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

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

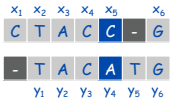
The pair x_i-y_j and $x_{i'}$-$y_{j'}$ *cross* if i < i', but j > j'.

Sequence Alignment Example

X = CTACCG

Y = TACTG

Solution: M = x_2-y_1 , x_3-y_2, x_4-y_3, x_5-y_4 , x_6-y_6



What is the cost of M?

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

Recall: mismatch penalty is 0 if x_i and y_j are the same

Sequence Alignment Case Analysis

Consider last character of strings X and Y: x_M and y_N

- Case 1: x_M and y_N are aligned
- Case 2: x_M is not matched
- Case 3: y_N is not matched

Sequence Alignment Cost Analysis

Consider last character of strings X and Y: x_M and y_N

- Case 1: x_M and y_N are aligned
  - Pay mismatch for x_M-y_N + min cost of aligning rest of strings
  - OPT(M, N) = α_{XmYn} + OPT(M-1, N-1)
- Case 2: x_M is not matched
  - Pay gap for x_M + min cost of aligning rest of strings
  - OPT(M, N) = δ + OPT(M-1, N)
- Case 3: y_N is not matched
  - Pay gap for y_N + min cost of aligning rest of strings
  - OPT(M, N) = δ + OPT(M, N-1)

Sequence Alignment: Problem Structure

Gaps for remainder of Y

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

Gaps for remainder of X

Sequence Alignment: Analysis

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

   for i = 1 to m                          O(mn)
      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]
```

Observation: to calculate the current value, we only need the row above us and the entry to the left

SEQUENCE ALIGNMENT IN LINEAR SPACE

Sequence Alignment: O(m) Space

Collapse into an m x 2 array

- M[i,0] represents previous row; M[i,1] -- current

```
Space-Efficient-Alignment(m, n, x1x2...xm, y1y2...yn, δ, α)
   for i = 0 to m        # initialize first row
      M[i, 0] = iδ
   for j = 1 to n
      M[0, 1] = jδ       # first gap

      for i = 1 to m
           M[i, 1] = min(α[xi, yj] + M[i-1, 0],
                         δ + M[i, 0],
                         δ + M[i-1, 1])
      for i = 1 to m     # copy current row into previous
           M[i, 0] = M[i, 1]
   return M[m, 1]
```

Any drawbacks?

Sequence Alignment: O(m) Space

Collapse into an m x 2 array

- M[i,0] represents previous row; M[i,1] -- current

```
Space-Efficient-Alignment(m, n, x1x2...xm, y1y2...yn, δ, α)
   for i = 0 to m
      M[i, 0] = iδ
   for j = 1 to n
      M[0, 1] = jδ

      for i = 1 to m
           M[i, 1] = min(α[xi, yj] + M[i-1, 0],
                         δ + M[i, 0],
                         δ + M[i-1, 1])
      for i = 1 to m
           M[i, 0] = M[i, 1]
   return M[m, 1]
```

Finds optimal value but won't be able to find the alignment

Why Do We Care About Space?

For English words or sentences, probably doesn't matter

Matters for Biological sequence alignment

- Consider: 2 strings with 100,000 symbols each
  - Processor can do 10 billion primitive operations
  - BUT dealing with a 10 GB array

Sequence Alignment: Linear Space

Can we avoid using quadratic space?

- Optimal value in O(m) space and O(mn) time.
  - Compute OPT(i, •) from OPT(i-1, •)
  - BUT, 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

Recall Our Example

X = bait Y = boot

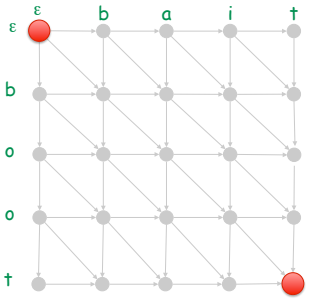
α = 1, for vowel mismatch
α = 2, for other mismatches
δ = 2

j →

i=4 ↓

		b	a	i	t
	0	2	4	6	8
b	2	0	2	4	6
o	4	2	1	3	5
o	6	4	3	2	4
t	8	6	5	4	2

Mapping to a Graph Problem

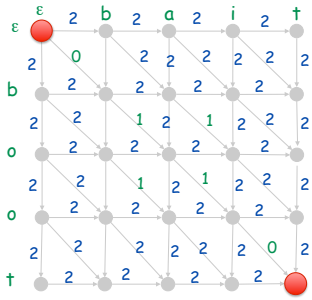


α = 1, for vowel mismatch
α = 2, for other mismatches
δ = 2

Horizontal and vertical edges cost δ
Diagonal edges cost α

Goal:
Find shortest path from top-left to bottom-right

Mapping to a Graph Problem



α = 1, for vowel mismatch
α = 2, for other mismatches
δ = 2

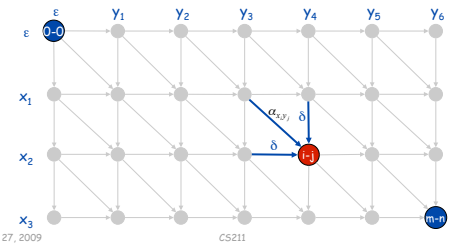
Horizontal and vertical edges cost δ
Diagonal edges cost α

Goal:
Find shortest path from top-left to bottom-right

Sequence Alignment: Forward

Edit distance graph

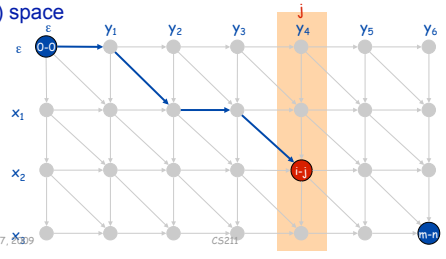
- Let f(i, j) be shortest path from (0,0) to (i, j)
- Observation: f(i, j) = OPT(i, j)



Sequence Alignment: Forward

Edit distance graph

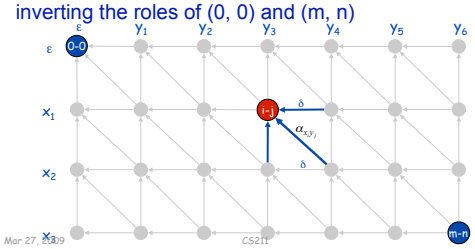
- Let f(i, j) be shortest path from (0,0) to (i, j)
- Can compute f (•, j) for any j in O(mn) time and O(m + n) space



Sequence Alignment: Backward

Edit distance graph

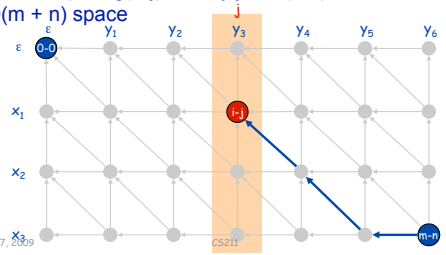
- Let g(i, j) be shortest path from (m, n) to (i, j)
- Can compute by reversing the edge orientations and inverting the roles of (0, 0) and (m, n)



Sequence Alignment: Backward

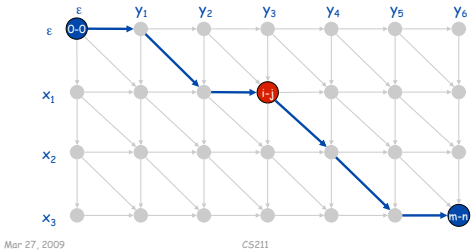
Edit distance graph

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



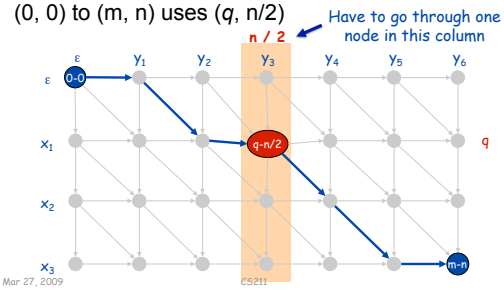
Sequence Alignment: Linear Space

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



Sequence Alignment: Linear Space

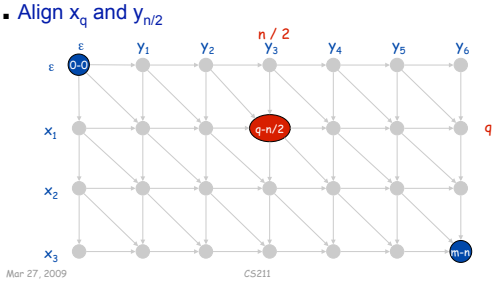
Observation 2. Let *q* be an index that minimizes f(*q*, n/2) + g(*q*, n/2). Then, the shortest path from (0, 0) to (m, n) uses (*q*, n/2)



Sequence Alignment: Linear Space

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

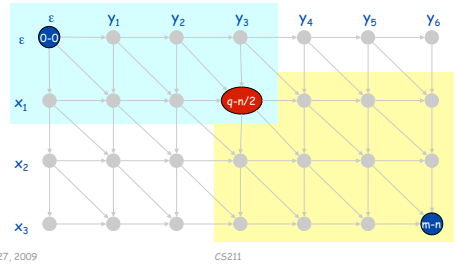
- Align x_q and $y_{n/2}$



Sequence Alignment: Linear Space

Conquer: recursively compute optimal alignment in each piece

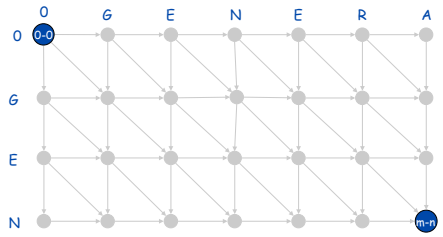
- Reuse working space from one recursive call to next



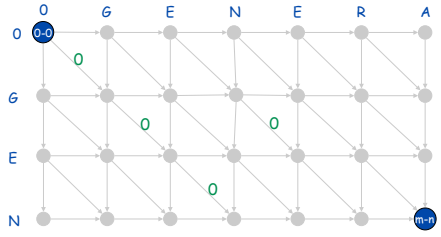
Divide and Conquer Sequence Alignment

```
P contains node on shortest corner-to-corner path
Divide-and-Conquer-Alignment (X, Y)
   m = length of X
   n = length of Y
   if m <= 2 or n <= 2
     compute optimal alignment using Alignment(X, Y)
     return
   Space-Efficient-Alignment(X, Y[1:n/2])
   Backward-Space-Efficient-Alignment(X, Y[n/2+1:n])
   q = index that minimizes f(q, n/2) + g(q, n/2)
   add(q, n/2) to P
   Divide-and-Conquer-Alignment(X[1:q],Y[1:n/2])
   Divide-and-Conquer-Alignment(X[q:m],Y[(n/2):n])
   return P
```

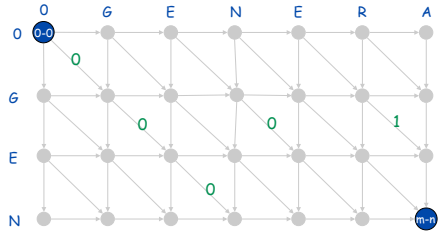

Example



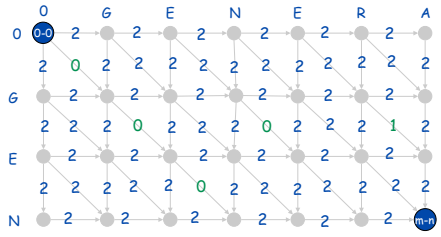
Example



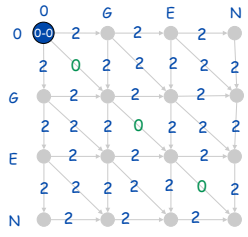
Example



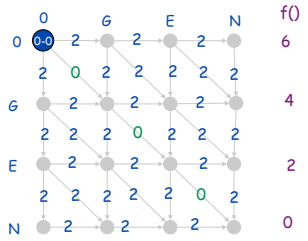
Example



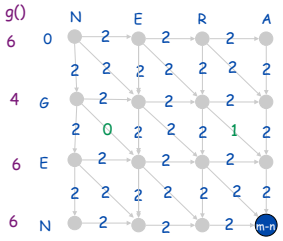
Space-efficient alignment: Left



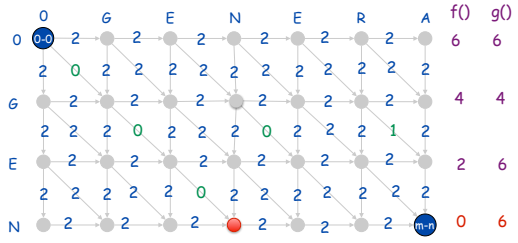
Space-efficient alignment: Left



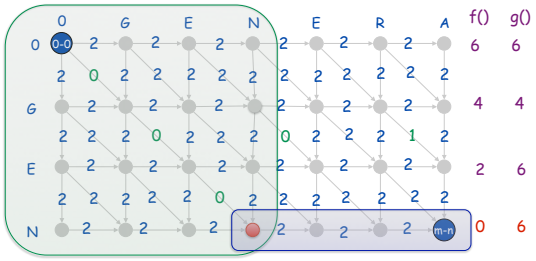
Backwards Space Efficient



Example



Example



Divide and Conquer Sequence Alignment: Analysis

```
P contains node on shortest corner-to-corner path
Divide-and-Conquer-Alignment (X, Y)
   m = length of X
   n = length of Y
   if m <= 2 or n <= 2
     compute optimal alignment using Alignment(X, Y)
     return
   Space-Efficient-Alignment(X, Y[1:n/2])
   Backward-Space-Efficient-Alignment(X, Y[n/2+1:n])
   q = index that minimizes f(q, n/2) + g(q, n/2)
   add(q, n/2) to P
   Divide-and-Conquer-Alignment(X[1:q],Y[1:n/2])
   Divide-and-Conquer-Alignment(X[q:m],Y[(n/2):n])
   return P
```

What is the recurrence relation?

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).

$$T(m,n) \le 2T(m, n/2) + O(mn) \Rightarrow T(m,n) = O(mn \log n)$$

Remark. Analysis is not tight because two sub-problems are of size (q, n/2) and (m - q, n/2).

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)

Recurrence Relation:

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

Solve using substitution:

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

Review: Problem Set 4

Recurrence Relations

1a: T(n) = 5T(n/2)+O(n)

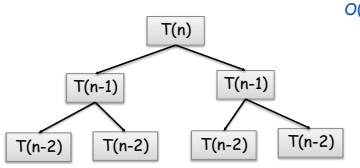
Using formula we figured out $O(n^{\log_2 q})$ for q > 2:

$O(n^{\log_2 5})$

1b. T(n) = 2T(n-1) + O(1)

Unrolling recurrence

Most common mistake: O(n)

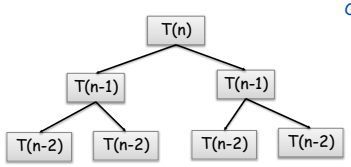


How many levels?
How many problems at bottom level?
What is the cost of each problem?

1b. T(n) = 2T(n-1) + O(1)

Unrolling recurrence

Most common mistake: O(n)



Recurrence: Like Fibonacci sequence

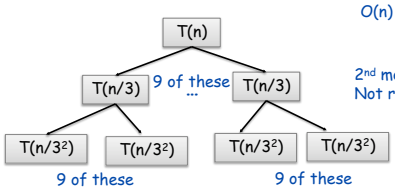
How many levels? n-1
How many problems at bottom level? 2^n
What is the cost of each problem? c O(1)

Therefore, bottom level is 2^n and $T(n) \in O(2^n)$

1c. T(n) = 9T(n/3) + O(n^2)

Unrolling the recurrence

Most common mistake: O(n)



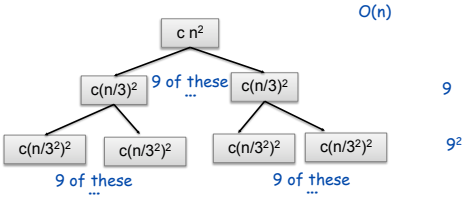
2nd most common mistake: Not reducing problem

How many levels?
How many problems at each level?

1c. T(n) = 9T(n/3) + O(n^2)

Unrolling the recurrence

Most common mistake: O(n)

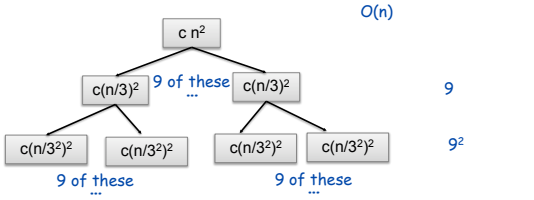


How many levels? $\log_3 n$
How many problems at each level? 9^i
What is the cost of each level?

1c. T(n) = 9T(n/3) + O(n^2)

Unrolling the recurrence

Most common mistake: O(n)



How many levels? $\log_3 n$
How many problems at each level? 9^i
What is the cost of each level? cn^2

$O(n^2 \log_3 n)$

Problem Set 6

1 Standard DP Problem

1 DP Problem *and* program implementing algorithm

- Goal: "pretty print" text so that the *slack* between text and right margin is minimized
  - Slack: number of spaces between text and right margin
- Write a Python program that *pretty prints* text given a maximum length of a line
- Template Python program, test files on Course web site

Unix Details

Python program (pp) must be *executable*

- `chmod u+x pp`
  - Make pp executable by user

Data file and example output are in tgz format

- Tar , gzip
- Need to unzip and extract file

Unix Details

Python program (pp) must be *executable*

- `chmod u+x pp`
  - Make pp executable by user

Data file and example output are in tgz format

- Tar , gzip
- Need to unzip and extract file
- `tar xfz neruda.tgz`