Biology & CS

Philip Chan

Biology

- Different levels
  - Evolution
    - organisms over time
  - Ecology
    - interactions among organisms and environment
  - Individual organisms
    - Anatomy, Physiology
  - Cell Biology
    - cells
  - Molecular Biology
    - chemical molecules

Molecular Biology

- DNA
  - Stands for?

Molecular Biology

- DNA
  - Dioxyribonucleic Acid
  - Double helix structure
    - Watson and Crick, 1953
    - Nobel Prize in Physiology or Medicine, 1962

Genome

- Chromosomes
  - inside where?

Genome

- Chromosomes
  - inside the cell nucleus
  - ? pairs

Genome

- Chromosomes
  - inside the cell nucleus
  - 23 pairs (one determines what?)

Genome

- Chromosomes
  - inside the cell nucleus
  - 23 pairs (one determines gender)

Genome

- Chromosomes
  - inside the cell nucleus
  - 23 pairs (one determines gender)
  - contains genetic information
  - copied during cell division
  - made of DNA
  - Gene
    - ?

Genome

- Chromosomes
  - inside the cell nucleus
  - 23 pairs (one determines gender)
  - contains genetic information
  - copied during cell division
  - made of DNA
  - Gene
    - (roughly) segments of DNA that encode proteins
- Genome
  - Human: ? genes

Genome

- Chromosomes
  - inside the cell nucleus
  - 23 pairs (one determines gender)
  - contains genetic information
  - copied during cell division
  - made of DNA
  - Genes
    - (roughly) segments of DNA that encodes proteins
- Genome
  - Human: 20,000-25,000 genes

DNA to Protein

- Transcription
  - DNA -> RNA
- Translation
  - RNA -> Protein

DNA Encoding for Proteins

- DNA
  - Sequence of nucleotides
    - 4 possible nucleotides:
      - Adenine (A), Cytosine (C), Guanine (G), Thymine (T)
        - [Thymine (T) becomes Uracil (U) in RNA]
- Protein
  - Sequence of amino acids
    - 20 possible amino acids
- How many nucleotides are needed to encode one amino acid?

Sequencing Human Genome

- Human Genome Project
  - International (governments/universities)
- Celera Corporation (US)
  - Many short sequences
  - Algorithms to merge them into longer sequences
- Complete genome sequence in ~2003

Why Study the Genome?

- Understanding how genes, proteins, … interact with each other
- Understanding diseases
  - Mistakes in copying DNA
  - Mutations cause changes in DNA

Comparing Genes

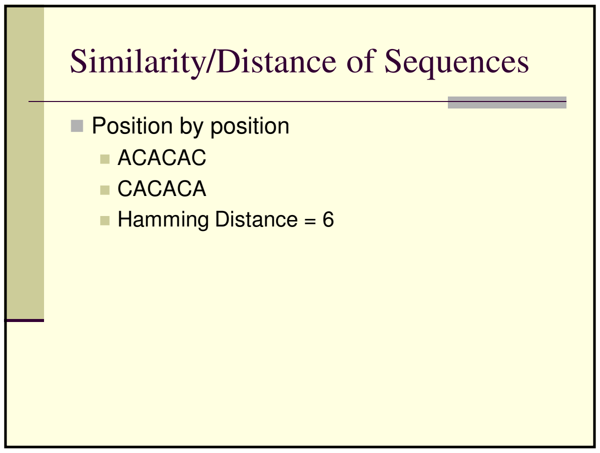
- After a gene is found
  - Biologist might not know its function
  - Find “similarities” with genes of known function

Cancer (1984)

- Cancer-causing gene is similar to a normal growth gene
- Cancer might be caused by a normal growth gene being switched on at the wrong time
- A good gene doing the right thing at the wrong time

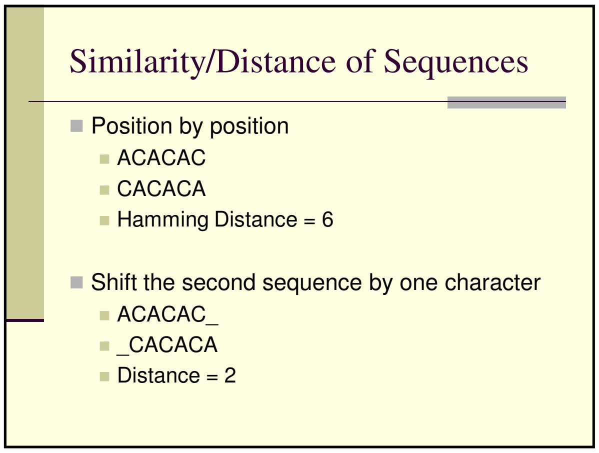
Cystic Fibrosis (1989)

- Cystic Fibrosis is a fatal disease associated with abnormal secretions (clogs in lungs).
- A segment of the Cystic Fibrosis gene is similar to the sequence for ATP binding proteins.
- These proteins affect cell membrane and secretions



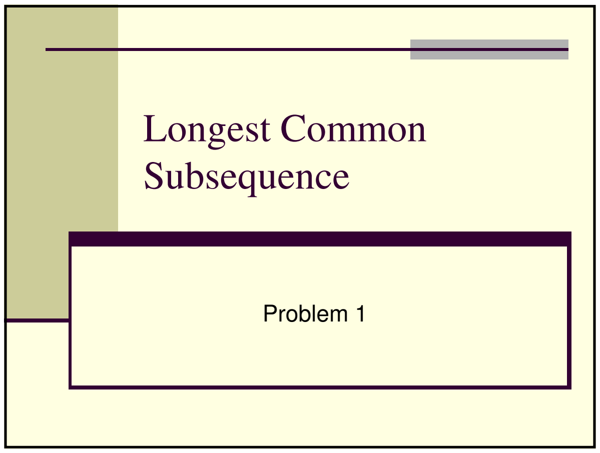
Similarity/Distance of Sequences

- Position by position
  - ACACAC
  - CACACA
  - Hamming Distance = 6



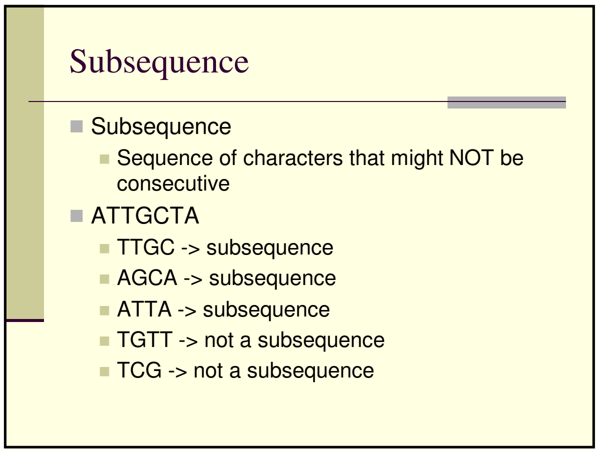
Similarity/Distance of Sequences

- Position by position
  - ACACAC
  - CACACA
  - Hamming Distance = 6
- Shift the second sequence by one character
  - ACACAC_
  - _CACACA
  - Distance = 2



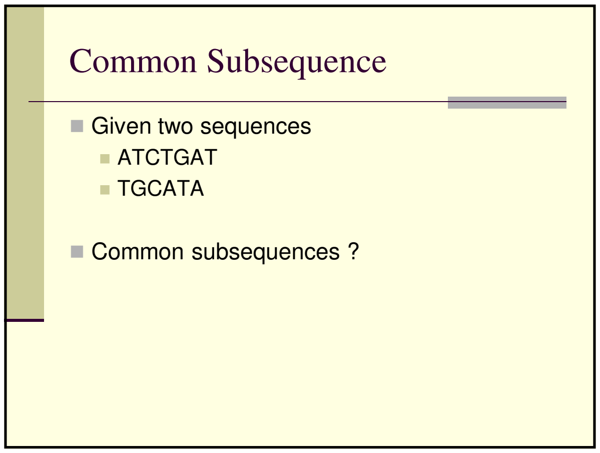
Longest Common Subsequence

Problem 1



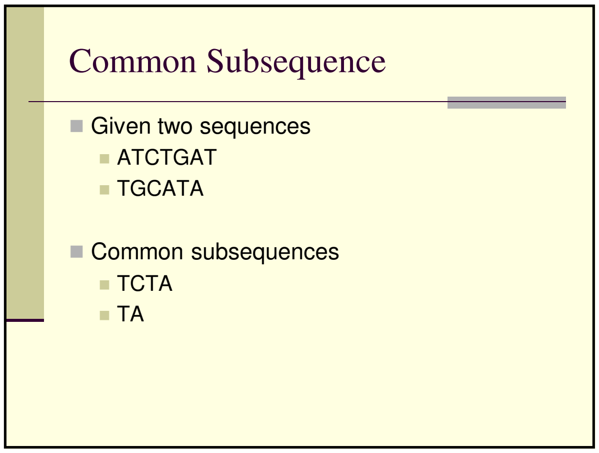
Subsequence

- Subsequence
  - Sequence of characters that might NOT be consecutive
- ATTGCTA
  - TTGC -> subsequence
  - AGCA -> subsequence
  - ATTA -> subsequence
  - TGTT -> not a subsequence
  - TCG -> not a subsequence



Common Subsequence

- Given two sequences
  - ATCTGAT
  - TGCATA
- Common subsequences ?



Common Subsequence

- Given two sequences
  - ATCTGAT
  - TGCATA
- Common subsequences
  - TCTA
  - TA

Longest Common Subsequence (LCS)

- Many different common subsequences
- Want to find the longest
- Length of LCS helps determine similarity of two sequences/genes

Problem Formulation

- Given (input)
  - Two sequences v, w
- Find (output)
  - Longest common substring of v and w (simpler problem)

Algorithm

- Any ideas?

Algorithm 1

- Find common subsequence of length 1
- Find common subsequence of length 2
- …

Algorithm 1

- Find common substring of length 1
- Find common substring of length 2
- …
- What is the time complexity?

Algorithm 1

- Find common substring of length 1
- Find common substring of length 2
- …
- What is the time complexity?
- Are we repeating unnecessary work?

Algorithm 2

- Observation:
  - If common substring of length L+1 exists
    - Common substring of length L must also exists
- Idea?

Algorithm 2

- Observation:
  - If common substring of length L+1 exists
    - Common substring of length L must also exists
- Idea
  - Use common substring of length L to find common substring of length L+1

Algorithm 2

- Observation:
  - If common substring of length L+1 exists
    - Common substring of length L must also exists
- Idea
  - Use common substring of length L to find common substring of length L+1
- Time complexity?

Algorithm ?

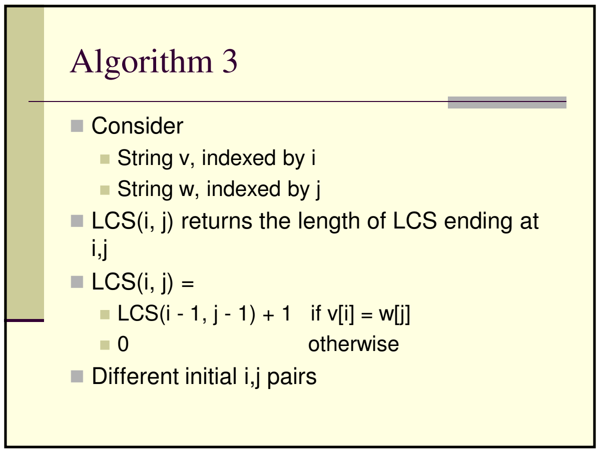
- Tree Search
- What would be the nodes and branches?
- Could recursion help?
- Time complexity?

Algorithm 3

- Consider
  - String v, indexed by i
  - String w, indexed by j
- LCS(i, j) returns the length of LCS ending at i,j

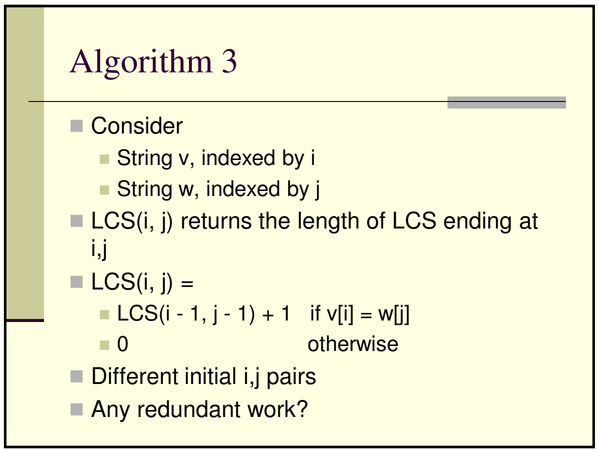
Algorithm 3

- Consider
  - String v, indexed by i
  - String w, indexed by j
- LCS(i, j) returns the length of LCS ending at i,j
- LCS(i, j) =
  - LCS(i - 1, j - 1) + 1 if v[i] = w[j]
  - 0 otherwise



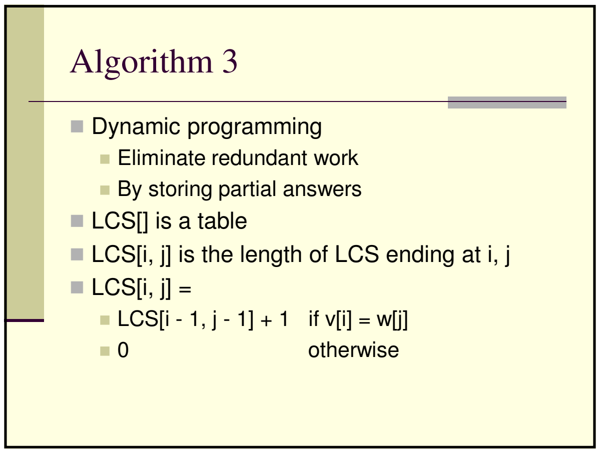
Algorithm 3

- Consider
  - String v, indexed by i
  - String w, indexed by j
- LCS(i, j) returns the length of LCS ending at i,j
- LCS(i, j) =
  - LCS(i - 1, j - 1) + 1   if v[i] = w[j]
  - 0   otherwise
- Different initial i,j pairs



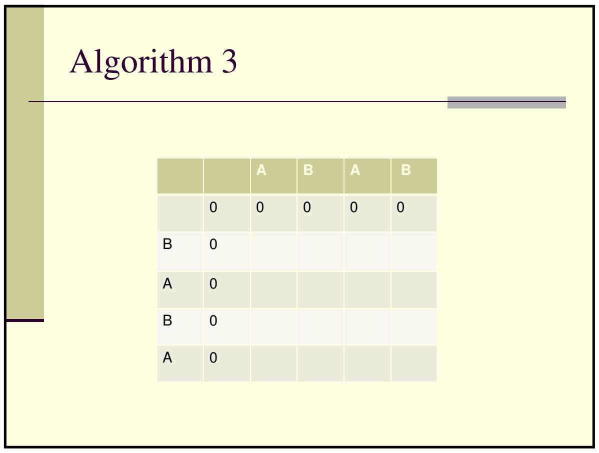
Algorithm 3

- Consider
  - String v, indexed by i
  - String w, indexed by j
- LCS(i, j) returns the length of LCS ending at i,j
- LCS(i, j) =
  - LCS(i - 1, j - 1) + 1   if v[i] = w[j]
  - 0   otherwise
- Different initial i,j pairs
- Any redundant work?



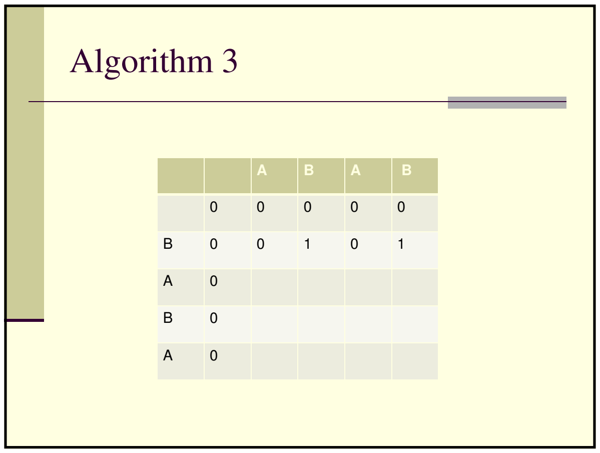
Algorithm 3

- Dynamic programming
  - Eliminate redundant work
  - By storing partial answers
- LCS[] is a table
- LCS[i, j] is the length of LCS ending at i, j
- LCS[i, j] =
  - LCS[i - 1, j - 1] + 1   if v[i] = w[j]
  - 0   otherwise



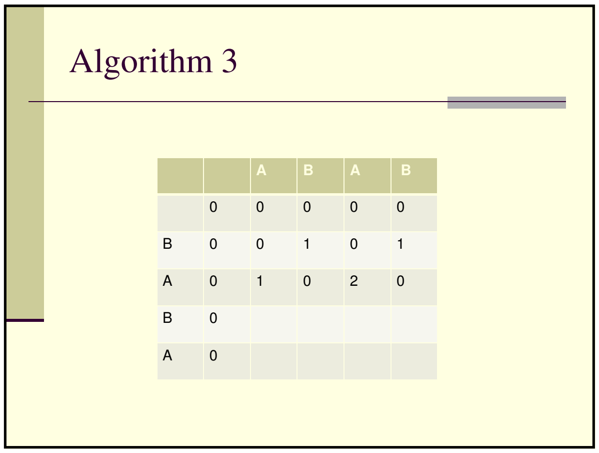
Algorithm 3

		A	B	A	B
	0	0	0	0	0
B	0				
A	0				
B	0				
A	0				



Algorithm 3

		A	B	A	B
	0	0	0	0	0
B	0	0	1	0	1
A	0				
B	0				
A	0				



Algorithm 3

		A	B	A	B
	0	0	0	0	0
B	0	0	1	0	1
A	0	1	0	2	0
B	0				
A	0				

Algorithm 3

		A	B	A	B
	0	0	0	0	0
B	0	0	1	0	1
A	0	1	0	2	0
B	0	0	2	0	3
A	0	1	0	3	0

Algorithm 3

		A	B	A	B
	0	0	0	0	0
B	0	0	1	0	1
A	0	1	0	2	0
B	0	0	2	0	3
A	0	1	0	3	0

Problem Formulation

- Given (input)
  - Two sequences v, w
- Find (output)
  - Longest common subsequence of v and w
  - Skipping character(s) is allowed

Problem

- String editing
  - Transform one string to another by keeping/adding/deleting characters
- Can also be viewed as aligning two strings
- Any ideas?

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

LCS: Example

i coords:	0	0	1	2	3	4	5	5	6	6	7
elements of *v*	--	T	G	C	A	T	--	A	--	C	
elements of *w*	A	T	--	C	--	T	G	A	T	C	
j coords:	0	1	2	2	3	3	4	5	6	7	8

(0,0)→(0,1)→(1,2)→(2,2)→(3,3)→(4,3)→(5,4)→(5,5)→(6,6)→(6,7)→(7,8)

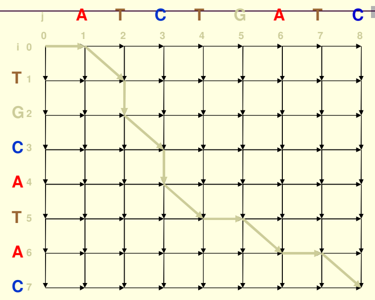
Matches shown in red

positions in *v*: 1 < 3 < 5 < 6 < 7

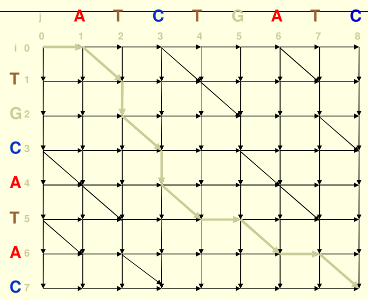
positions in *w*: 2 < 3 < 4 < 6 < 8

Every common subsequence is a path in 2-D grid

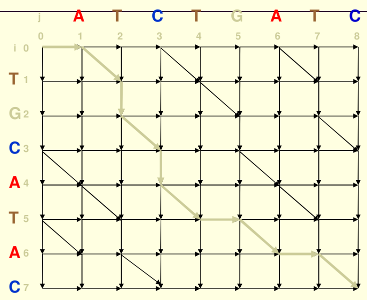
Edit Graph for LCS Problem



Edit Graph for LCS Problem



Edit Graph for LCS Problem

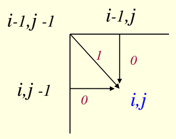


Every path is a common subsequence.

Every diagonal edge adds an extra element to common subsequence

LCS Problem: *Find a path with maximum number of diagonal edges*

Computing LCS



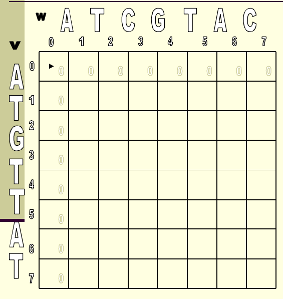
$$s_{i,j} = \text{MAX} \begin{cases} s_{i-1,j} + 0 \\ s_{i,j-1} + 0 \\ s_{i-1,j-1} + 1, \quad \text{if } v_i = w_j \end{cases}$$

Computing LCS

The length of LCS($\boldsymbol{v}_i$,$\boldsymbol{w}_j$) is computed by:

$$s_{i,j} = \max \begin{cases} s_{i-1,j} \\ s_{i,j-1} \\ s_{i-1,j-1} + 1 \text{ if } v_i = w_j \end{cases}$$

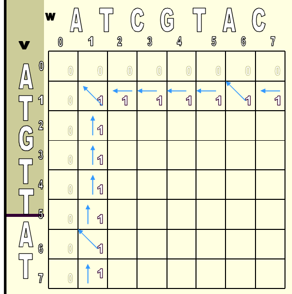
Dynamic Programming Example



Initialize 1st row and 1st column to be all zeroes.

Or, to be more precise, initialize *0th* row and *0th column to be all zeroes.*

Dynamic Programming Example



$$S_{i,j} = \max \begin{cases} S_{i-1,j-1} \leftarrow \text{value from NW +1, if } v_i = w_j \\ S_{i-1,j} \leftarrow \text{value from North (top)} \\ S_{i,j-1} \leftarrow \text{value from West (left)} \end{cases}$$

Dynamic Programming: Backtracing

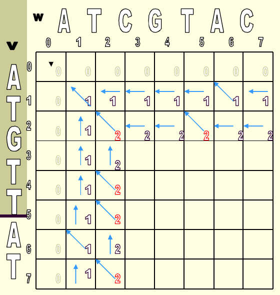
Arrows ← ↖ ↑ show where the score originated from.

↑ if from the top

← if from the left

↖ if $v_i = w_j$

Dynamic Programming Example



Find a match in row and column 2.

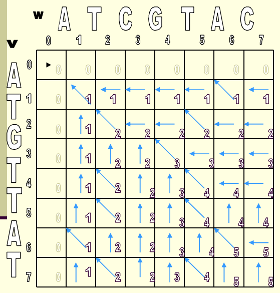
i=2, j=2,5 is a match (T).

j=2, i=4,5,7 is a match (T).

Since $v_i = w_j$, $s_{i,j} = s_{i-1,j-1} + 1$

$s_{2,2} = [s_{1,1} = 1] + 1$
$s_{2,5} = [s_{1,4} = 1] + 1$
$s_{4,2} = [s_{3,1} = 1] + 1$
$s_{5,2} = [s_{4,1} = 1] + 1$
$s_{7,2} = [s_{6,1} = 1] + 1$

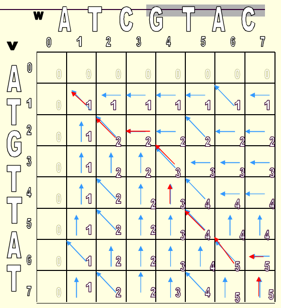
Dynamic Programming Example



Continuing with the dynamic programming algorithm gives this result.

Now What?

- LCS(v,w) created the alignment grid
- Now we need a way to read the best alignment of v and w
- Follow the arrows backwards from sink



LCS Algorithm

```
LCS(v,w)
  for i ← 1 to n
    s_{i,0} ← 0
  for j ← 1 to m
    s_{0,j} ← 0
  for i ← 1 to n
    for j ← 1 to m
                       ⎧ s_{i-1,j}
          s_{i,j} ← max ⎨ s_{i,j-1}
                       ⎩ s_{i-1,j-1} + 1, if v_i = w_j
                      ⎧ “ ↑ “   if  s_{i,j} = s_{i-1,j}
          b_{i,j} ←   ⎨ “ ← “   if  s_{i,j} = s_{i,j-1}
                      ⎩ “ ↖ “   if  s_{i,j} = s_{i-1,j-1} + 1
    return (s_{n,m}, b)
```

Printing LCS: Backtracing

```
PrintLCS(b,v,i,j)
  if i = 0 or j = 0
    return
  if b_{i,j} = “↖“
    PrintLCS(b,v,i-1,j-1)
    print v_i
  else
    if b_{i,j} = “↑“
      PrintLCS(b,v,i-1,j)
    else
      PrintLCS(b,v,i,j-1)
```

LCS Time Complexity

- It takes O(*nm*) time to fill in the *nxm* dynamic programming matrix.
- Why O(*nm*)? The pseudocode consists of a nested “for” loop inside of another “for” loop to set up a *nxm* matrix.

Global Sequence Alignment

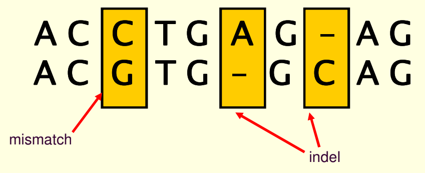
Problem 2

LCS

- simplest form of sequence alignment
  - Calculating sequence similarity
- allows only insertions and deletions (no mismatches).
- score 1 for matches and 0 for indels (insertions/deletions)

Mismatch and Indel

- Indel: insertion/deletion
- Allowing substitution/mismatch



- What do we do with mismatches and indels?

From LCS to Alignment

- penalizing indels and mismatches with negative scores
- Simplest *scoring schema*:
  +1 : **match premium**
  -μ : **mismatch penalty**
  -σ : **indel penalty**
- the resulting score is:

#matches – μ(#mismatches) – σ (#indels)

Global Alignment

- Given (input)
  - Two sequences: v, w
  - Penalty for mismatches and indels
- Find (ouput)
  - Alignment with the maximum score

The Global Alignment Problem

$\uparrow\rightarrow = -\sigma$

$\searrow \begin{cases} = 1 \text{ if match} \\ = -\mu \text{ if mismatch} \end{cases}$

μ : mismatch penalty
σ : indel penalty

$$s_{i,j} = \max \begin{cases} s_{i-1,j-1} + 1 \text{ if } v_i = w_j \\ s_{i-1,j-1} - \mu \text{ if } v_i \neq w_j \\ s_{i-1,j} - \sigma \\ s_{i,j-1} - \sigma \end{cases}$$

Scoring Matrices

To generalize scoring, consider a (4+1) x(4+1) **scoring matrix** δ.

In the case of an amino acid sequence alignment, the scoring matrix would be a (20+1)x(20+1) size. The addition of 1 is to include the score for comparison of a gap character "-".

This will simplify the algorithm as follows:

$$s_{i,j} = max \begin{cases} s_{i-1,j-1} + \delta(v_i, w_j) \\ s_{i-1,j} + \delta(v_i, -) \\ s_{i,j-1} + \delta(-, w_j) \end{cases}$$

Local Alignment

Problem 3

Local Alignments: Why?

- Two genes in different species
  - similar over short conserved regions and
  - dissimilar over remaining regions.
- Example:
  - Homeobox genes have a short region called the *homeodomain* that is highly conserved between species.
  - A global alignment would not find the homeodomain because it would try to align the ENTIRE sequence

Local vs. Global Alignment

- The Global Alignment Problem tries to find the longest path between vertices *(0,0)* and (*n,m*) in the edit graph.
- The Local Alignment Problem tries to find the longest path among paths between **arbitrary vertices** (*i,j*) and (*i', j'*) in the edit graph.

Local vs. Global Alignment (cont'd)

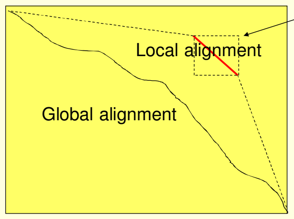
- Global Alignment

```
--T—-CC-C-AGT—-TATGT-CAGGGGACACG—A-GCATGCAGA-GAC
  |   ||  |   ||    |  |  |  |||     ||  |  |  |    |  ||||     |
AATTGCCGCC-GTCGT-T-TTCAG----CA-GTTATG—T-CAGAT--C
```

- Local Alignment—better alignment to find conserved segment

```
                  tccCAGTTATGTCAGgggacacgagcatgcagagac
                     ||||||||||||
aattgccgccgtcgttttcagCAGTTATGTCAGatc
```

Local Alignment: Example



Compute a “mini” Global Alignment to get Local

The Local Alignment Problem

- Given (input)
  - strings **v, w**
  - scoring matrix *δ*
- Find (output)
  - alignment of substrings of **v** and **w** whose alignment score is maximum among all possible alignment of all possible substrings

Algorithm

- Any ideas?

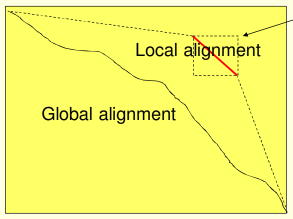
Algorithm

- For each possible starting point
  - Call global alignment
    - (every ending point is considered)
- Time complexity?

Algorithm

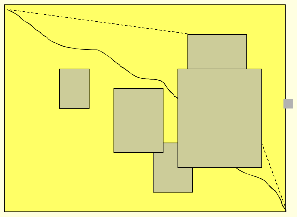
- For each possible starting point
  - Call global alignment
    - (every ending point is considered)
- Time complexity?
  - *O*(n^2) pairs of start and end points
    - Each alignment takes *O*(n^2)
  - Total *O*(n^4)

Local Alignment: Example



Compute a “mini” Global Alignment to get Local

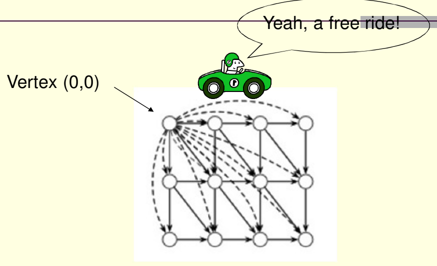
Local Alignment: Running Time



- Long run time $O(n^4)$:

  - In the grid of size $n \times n$ there are $\sim n^2$ vertices (i,j) that may serve as a source.
  - For each such vertex computing alignments from (i,j) to (i',j') takes $O(n^2)$ time.
- This can be remedied by giving free rides

Local Alignment: Free Rides



The dashed edges represent the free rides from (0,0) to every other node. ie, skipping multiple characters in one step.

The Local Alignment Recurrence

- The largest value of $s_{i,j}$ over the whole edit graph is the score of the best local alignment.
- The recurrence:

$$s_{i,j} = \max \begin{cases} 0 \\ s_{i-1,j-1} + \delta(v_i, w_j) \\ s_{i-1,j} + \delta(v_i, -) \\ s_{i,j-1} + \delta(-, w_j) \end{cases}$$

Notice there is only this change from the original recurrence of a Global Alignment

The Local Alignment Recurrence

- The largest value of $s_{i,j}$ over the whole edit graph is the score of the best local alignment.
- The recurrence:

$$s_{i,j} = \max \begin{cases} 0 \\ s_{i-1,j-1} + \delta(v_i, w_j) \\ s_{i-1,j} + \delta(v_i, -) \\ s_{i,j-1} + \delta(-, w_j) \end{cases}$$

Power of ZERO: there is only this change from the original recurrence of a Global Alignment - since there is only one “free ride” edge entering into every vertex

Summary

1. Longest Common Subsequence
   - No penalty on mismatches and indels
2. Global Alignment
   - Penalize mismatches and indels
3. Local Alignment
   - Short highly similarly subsequences