DNA

DNA: sequence of base-pairs (bp):
(A, C, T, G)

Human Genome
about $3 \times 10^9$ bps divided into
46 chromosomes with between
5 $\times 10^7$ and 25 $\times 10^7$ bps each
Each chromosome is a sequence of base-pairs

DNA is used to generate proteins:

DNA $\xrightarrow{\text{transcription}}$ mRNA $\xrightarrow{\text{translation}}$ Protein

Proteins

Proteins: sequence of Amino Acids
{gly, trp, cys, ...} (20 of them)

Each DNA bp triple (a "codon") encodes 1 amino acid
Since there are 64 possible codons, this is a many
to one mapping. Some triples have special
meanings, e.g. EOF.

Chromosomes are partitioned into genes each of which
codes a protein. Some regions of the chromosome
do not code anything (intergene DNA).

gene 1  gene 2  gene 3

Form and Function

The Amino Acid sequence determines the protein’s 3d
structure. The structure is also be affected by
the environment.

- The primary structure refers to the amino acid
sequence.
- The secondary structure refers to general
configuration into alpha helixes and beta sheets
- The tertiary structure refers to the full 3d
structure

Protein’s 3d structure determines its function.
Some Goals in Molecular Biology

1. Extract and compare genome sequence for various organisms.
2. Determine what proteins they code.
3. Determine structure and purpose of coded proteins.

Goals 2. and 3. can often be aided by matching genome or protein sequences to previously studied sequences.

Use to:
- study and cure genetic diseases
- design drugs
- study evolution
- understand molecular processes

Example of MS

Multiple Sclerosis is a disease in which the immune system attacks the myelin sheaths of nerve cells.

Conjecture: The immune system T-cells incorrectly identify the myelin sheaths as a virus or bacteria from an earlier infection.

This was tested by comparing the proteins sequences of myelin sheaths with a database of viral and bacterial proteins.

The ones that matched were tested in the laboratory to see if they were attacked by the T-cells. Some were.

Approximate Matching and Sequence Alignment

How similar are:
actagtctac
cgacgtcgata

Allow for:
- mutations: x to y
- insertions: _ to y
- deletions: x to _

“indels”

e.g.
acta_gtc__tac
_ _ _ _ _ _
_cgacgtcgata_

1 mutation
3 insertions
2 deletions

Applications of matching and alignment

Used in many ways in computational biology:
- Sequencing (finding the sequence of DNA or Proteins for a particular individual or species)
- Physical mapping (locating unique tags in DNA)
- Database searches (does this DNA match any other DNA?)
- Evolutionary trees (how similar are two species?)

Before talking about the general matching problem in computational Biology, we will talk about a closely related, but simpler problem: longest common subsequence (LCS)
**Longest Common Subsequence**

**Subsequence** \( (\subseteq) \): Any subset of the elements of a sequence that maintain relative order

e.g. \( A = a_1a_2a_3a_4a_5a_6 \)
\( A' = a_2a_4a_5 \) (a subsequence of \( A \))
\( A'' = a_2a_5a_6 \) (not a subsequence of \( A \))

**Longest Common Subsequence** \( (LCS) \):

\( LCS(A,B) = C \), where \( C \subseteq A, C \subseteq B, |C| \) is maximized

\( e.g. \ A = a \ b \ a \ c \ d \ a \ c \)
\( B = c \ a \ d \ c \ d \ c \)
\( C = a \ c \ d \ c \quad |C| = 4 \)

**Minimum Edit Distance**

**Minimum Edit Distance** \( D(A,B) \): minimum \# of insertions and deletions needed to change A to B.

\( e.g. \ A = a \ b \ a \ c \ d \ a \ c \)
\( B = c \ a \ d \ c \ d \ c \)

**Claim:** \( D(A,B) = |A| + |B| - 2|LCS(A,B)| \)

**Proof outline:**

\( C = CS(A,B) \) (some common subsequence)
\( A - C = \text{deletions} \), \( B - C = \text{insertions} \)
\#deletions = \(|A| - |C|\) \quad \#insertions = \(|B| - |C|\)

This reduction works both ways, hence equality.

**Applications**

**Unix diff:**
Find Minimum Edit Distance and print edits
GNU diff based on algorithm by Eugene Myers, who was VP of Informatics at Celera
Used heavily in version control systems to only store edits that have been made to previous version.

**Computational biology:**
This will use generalizations of LCS, but the algorithms will work with slight modifications.

**Outline**

Will work our way up to the GNU diff algorithm
- Recursive solution
- Memoized solution
- Dynamic programming
- Memory efficient solution
- Myers/Ukkonen algorithm
Recursive Solution

\[
D(A, \text{empty}) = |A| \quad \text{delete}(A)
\]
\[
D(\text{empty}, B) = |B| \quad \text{insert}(B)
\]
\[
D(A:x, B:x) = D(A, B)
\]
\[
D(A:a, B:b) = \min(1 + D(A:a, B), 1 + D(A, B:b))
\]

insert(b) delete(a)

Note that this just returns the edit distance, but it is easy to include the edits.
Can work from start or end (here from end).
Why does it work?
What is the running time?

Memoized Solution

int ED(int i, int j) {
    if (M[i,j] != -1) return M[i,j];
    if (i==0) r = j;
    if (j==0) r = i;
    if (A[i] == B[j])
        r = ED(i-1,j-1);
    else
        r = min(1 + ED(i-1,j), 1 + ED(i,j-1));
    return M[i,j] = r;
}

M[1..n,1..m] = -1;
ED(n,m);

Dynamic programming

for i = 1 to n
    M[i,1] = i;
for j = 1 to m
    M[1,j] = j;
for i = 1 to n
    for j = 1 to m
        if (A[i] == B[j])
            M[i,j] = M[i-1,j-1];
        else
            M[i,j] = 1 + min(M[i-1,j],M[i,j-1]);
**Example**

```
<table>
<thead>
<tr>
<th>B</th>
<th>a</th>
<th>t</th>
<th>c</th>
<th>a</th>
<th>c</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>t</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>c</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>a</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>t</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
```

→ insert

↓ delete

**Common**

Note: can be filled in any order as long as the cells to the left and above are filled.

Can follow path back through matrix to construct edits.

**Space Efficient Solution**

```
for i = 1 to n
for j = 1 to m
if (A[i] == B[j])
    R[j] = Rprev[j-1];
else
    R[j] = 1 + min(Rprev[j],R[j-1]);
Rprev[0..m] = R[0..m]
```

Requires only $O(m)$ space.

What is the problem?

**Space Efficient Solution**

For each entry in a row past $n/2$ keep track of which column it comes from in row $n/2$.

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>t</td>
<td>c</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>t</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>c</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>a</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>t</td>
<td>4</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
```

The function $ED'(A,B)$ returns column:

**Space efficient solution**

Now solve recursively:

```
function RecED(A,B) =
k = ED'(A,B)
RecED(A[1..n/2],B[1..k]) ++ RecED(A[n/2..n],B[k..m])
```

$k = ED'(A,B)$
**Space Efficient Solution**

**Time:**

\[ T(n,m) = T(n/2,k) + T(n/2,m-k) + O(nm) \]

\[ = O(nm) \]

**Space:**

\[ S(n,m) = O(m) \text{ for } ED' \text{ and } O(m + n) \text{ for result} \]

\[ = O(n+m) \]

---

**Space Efficient Solution: Notes**

We used divide-and-conquer to save space, not time.

We allowed for redundant work, but that’s OK!

---

**Improving Time Bounds**

For many applications (e.g. diff), the difference between strings tends to be small. Can we do something better for this case?

This idea was exploited by Myers and Ukkonen (independently) in 1985. Now the basis for GNU Diff.

An example of an algorithms whose runtime is output sensitive.

---

**Viewing as a Graph**

Edit distance can be expressed as the problem of finding the shortest path in the following graph:

How many vertices will Dijkstra’s algorithm visit as a function of \( n \), \( m \) and \( d \)?
**Dijkstra’s algorithm (review)**

Single source \((v_s)\) single destination \((v_d)\) shortest path

- \(d(v) = 1\) for all \(v = v_s\)
- \(d(v_s) = 0\)
- \(v_s\) inserted into \(Q\)
- while Min(Q) \(\neq v_d\)
  - \(v = \text{deleteMin}(Q)\)
  - for each neighbor \(v'\) of \(v\)
    - \(d(v') = \min(d(v'), d(v) + \text{weight}(v,v'))\)
    - \(v'\) inserted into \(Q\), or replace if already in there

Takes \(|E|\) inner iterations, each taking \(O(\log |V|)\) time.
Can be improved to \(O(|E| + |V|\log|V|)\) time.

**Bounding Visited Vertices**

**Theorem:** \(D_{ij}, |j-i|\)

**Proof:** every step away from the diagonal is along a horizontal or vertical edge. Each such edge contributes 1, so the total \(D\), must be at least the distance from the diagonal

**Corollary:** Dijkstra’s algorithm visits at most \(\min(n,m)^*2d\) vertices.

**Bounding time**

Priority Queue can be kept in constant time per operation (basically a modified breadth-first search), so total time is \(O(nm)\). In practice this can be much less than \(O(nm)\).

**Optimizing Space**

Use recursive trick from before

**Problem:** We do not know \(d\) ahead of time. Scanning row by row will not work without \(d\).
Need to bound size of frontier.
**Increasing Band-Widths**

Start with band of $|m-n|$ and double on each step until large enough

Visits at most twice as many vertices as it should.