Lezione 13

Bioinformatica

Mauro Ceccanti[‡] e Alberto Paoluzzi[†]

[†] Dip. Informatica e Automazione – Università “Roma Tre”
[‡] Dip. Medicina Clinica – Università “La Sapienza”

Lecture 13: Alignment of sequences
- Sequence alignment
- Dot Matrix of two sequences
- Introduction to dynamic programming
- Longest common subsequence (LCS) problem

Sommario

Lecture 13: Alignment of sequences
- Sequence alignment
- Dot Matrix of two sequences
- Introduction to dynamic programming
- Longest common subsequence (LCS) problem

Background

Biomolecules are strings from a restricted alphabet

- Let Σ be an alphabet, a non-empty finite set.
- Elements of Σ are called symbols or characters.
- A string (or word) over Σ is any finite sequence of characters from Σ.
- For example, if $\Sigma = \{0, 1\}$, then 0101 is a string over Σ

Background

Biomolecules are strings from a restricted alphabet

DNA alphabet Length=4

- 4 nucleotides

Protein alphabet Length=20

- 20 amino acids

Shape determines function

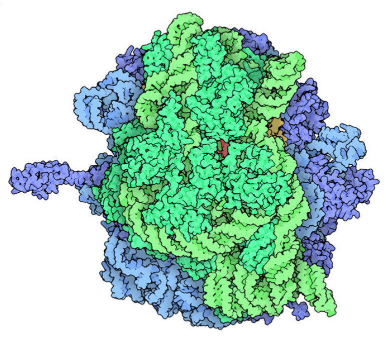
- Protein is a string (sequence of amino acids)
- Proteins do not stay linear in space
- Folding happens
- Folding determines overall 3-D shape
- Shape determines function

```
RIBOSOME =
"MARIAGVEIPRNKRVDVALTYIYG␣
    IGKARAKEALEKTGINPATRVK␣
    DLTEAEVVRLREYVENTWKLE␣
    GELRAEVAANIKRLMDIGCYR␣
    GLRHRRGLPVRGQRTRTNAR␣
    TRKGPRKTVAGKKKAPRK␣ . . . "
```

After solving the structures of the individual small and large subunits, the next step in ribosome structure research was to determine the structure of the whole ribosome. This work is the culmination of decades of research, which started with blurry pictures of the ribosome from electron microscopy, continued with more detailed cryoelectron micrographic reconstructions, and now includes many atomic structures. These structures are so large that they don't fit into a single PDB file–for instance, the structure shown here was split into PDB entries 2wdk and 2wdl.

Shape determines function

- Protein is a string (sequence of amino acids)
- Proteins do not stay linear in space
- Folding happens
- Folding determines overall 3-D shape
- Shape determines function



In 2000, structural biologists Venkatraman Ramakrishnan, Thomas A. Steitz and Ada E. Yonath made the first structures of ribosomal subunits available in the PDB, and in 2009, they each received a Nobel Prize for this work.

Sequence ⇒ Structure ⇒ Function

bbbbbb

- the amino acids in a protein sequence interact locally and establish hydrogen (and even covalent) bounds
- the interaction folds the protein in space and gives it a 3D structure
- the 3D structure determines the protein function
- each protein within the body has a specific function

Sequence alone does not reveal structure

Much less function ... So?

Nature does not solve the same problem twice (usually)

- Short sequence with a specific function (or shape) is called a domain
- The same domain appears in multiple proteins
- If we find the same domain in multiple proteins that provides a clue to function and/or structure

Sequence is easier to get than structure or function

How biologists study proteins

- To study the 3D structure of proteins is hard and expensive (NMR, x-ray crystallography)
- Analogously, the discovery of function through laboratory (in-vitro) and animal (in-vivo) experiments is difficult
- Therefore, few (tens of) thousands of proteins are understood in detail
- Many (i.e. millions) are known only by sequence

SEQUENCE ALIGNMENT SCENARIO

sequence of a new protein with unknown function

- Biologist discovers the sequence of a new protein with unknown function
- If sequence can be associated with a known protein sequence we have a clue about structure and/or function
- Vast quantities of sequence, structure, function info is deposited into public databases
- The new sequence should be compared to the database to find the more similar domains

Main Alignment Methods

- Dot Matrix
- Dynamic Programming
- BLAST, FASTA

Sommario

Lecture 13: Alignment of sequences

- Sequence alignment
- Dot Matrix of two sequences
- Introduction to dynamic programming
- Longest common subsequence (LCS) problem

Similarity of Sequences as homology of structures

bbbbbbb

- Locating regions of similarity between two DNA or protein sequences
- Provide a lot of information about the function and structure of the query sequence
- Similarity of sequences indicates homology
- Two structures are called homologous if they represent corresponding parts of organisms which are built according to the same body plan
- The existence of corresponding structures in different species is explained by derivation from a common ancestor

Similarity relation

matrix picture of sequence similarity

A picture of the similarity of two sequences X, Y can be given by the graph of the similarity relation $S \subseteq X \times Y$ such that:

$$x_i \, S \, y_j \equiv (x_i, y_j) \in S \Longleftrightarrow x_i = y_j$$

By the way, the interesting part of the similarity relation S is given by its reflexive subsets

$$S_{i,j,k} = \{(x_i, y_j) \,|\, x_{i+\ell} = y_{j+\ell}, \quad \ell = 0, \ldots, k\}$$

with starting point (i, j) and length k

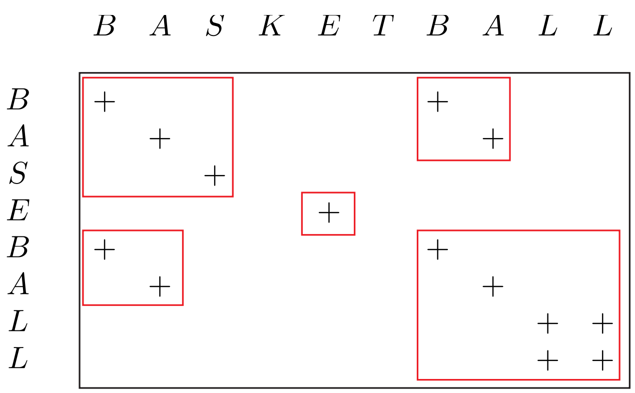
Similarity relation

matrix picture of sequence similarity

$$\begin{array}{c|cccccccccc|}
 & B & A & S & K & E & T & B & A & L & L \\
\hline
B & + & & & & & & + & & & \\
A & & + & & & & & & + & & \\
S & & & + & & & & & & & \\
E & & & & & + & & & & & \\
B & + & & & & & & + & & & \\
A & & + & & & & & & + & & \\
L & & & & & & & & & + & + \\
L & & & & & & & & & + & + \\
\hline
\end{array}$$

Similarity relation

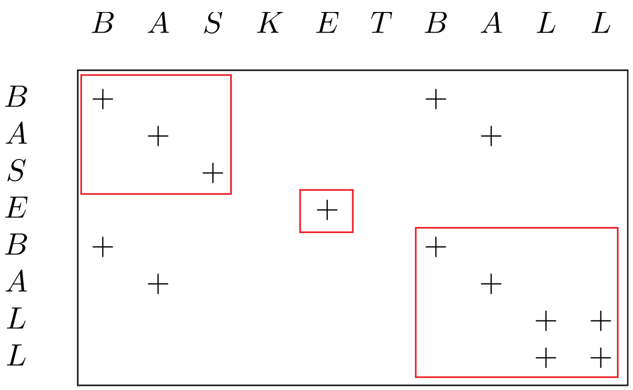
matrix picture of sequence similarity



$$\begin{array}{c|cccccccccc}
 & B & A & S & K & E & T & B & A & L & L \\
\hline
B & + & & & & & & + & & & \\
A & & + & & & & & & + & & \\
S & & & + & & & & & & & \\
E & & & & & + & & & & & \\
B & + & & & & & & + & & & \\
A & & + & & & & & & + & & \\
L & & & & & & & & & + & + \\
L & & & & & & & & & + & +
\end{array}$$

Similarity relation

drop out the reflexive subset that are non maximal[1]



$$\begin{array}{c|cccccccccc}
 & B & A & S & K & E & T & B & A & L & L \\
\hline
B & + & & & & & & + & & & \\
A & & + & & & & & & + & & \\
S & & & + & & & & & & & \\
E & & & & & + & & & & & \\
B & + & & & & & & + & & & \\
A & & + & & & & & & + & & \\
L & & & & & & & & & + & + \\
L & & & & & & & & & + & +
\end{array}$$

[1] if we (i.e. that are contained within another reflexive subset)

Similarity relation

finally project the maximal reflexive subrelations in one (or both) starting sequence

getting the Longest Common Subsequence

$$\boxed{B\quad A\quad S}\,\boxed{E}\,\boxed{B\quad A\quad L\quad L}$$

Sommario

Lecture 13: Alignment of sequences

- Sequence alignment
- Dot Matrix of two sequences
- Introduction to dynamic programming
- Longest common subsequence (LCS) problem

Introduction to dynamic programming

Bellman optimality principle

> *Principle of Optimality: An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.*
>
> Richard Bellman, 1957. *Dynamic Programming.* Princeton University Press, Princeton, NJ.

Optimal substructure

necessary condition

necessary condition for optimality associated with the mathematical optimization method known as dynamic programming

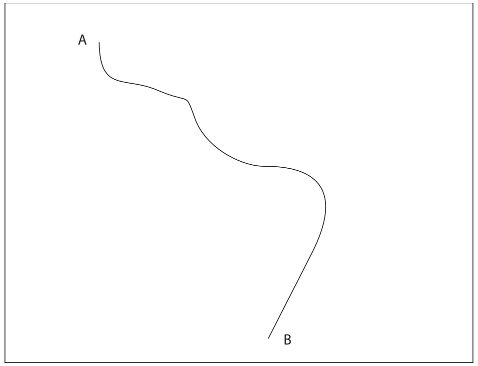
It breaks a dynamic optimization problem into simpler subproblems

In computer science, a problem that can be broken apart like this is said to have optimal substructure

Optimal substructure

a global optimal policy

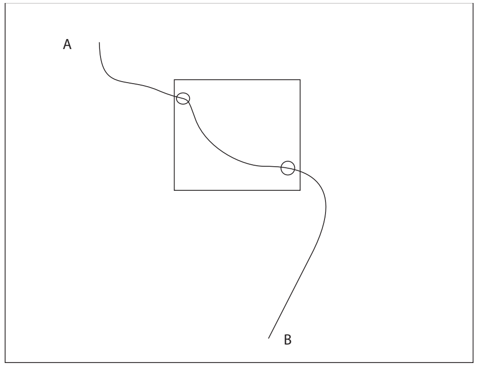
The (optimal) solution of a problem with optimal substructure is made by composition of (optimal) solutions to subproblems, each having in turn optimal substructure



Optimal substructure

a global optimal policy

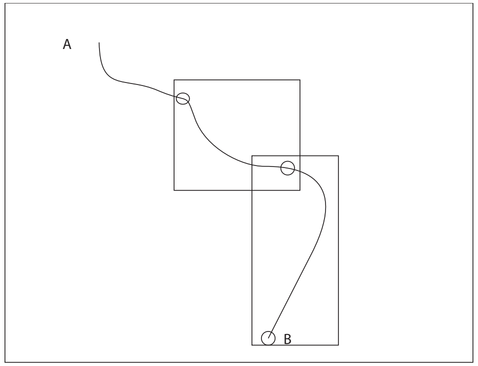
The (optimal) solution of a problem with optimal substructure is made by composition of (optimal) solutions to subproblems, each having in turn optimal substructure



Optimal substructure

a global optimal policy

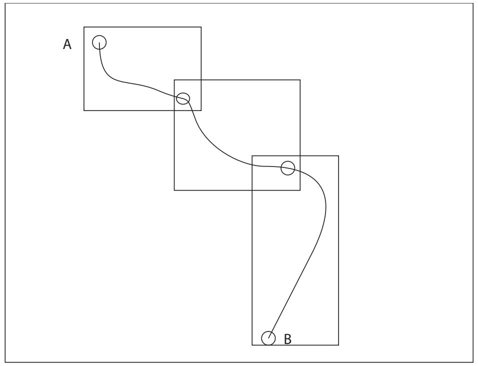
The (optimal) solution of a problem with optimal substructure is made by composition of (optimal) solutions to subproblems, each having in turn optimal substructure



Optimal substructure

a local optimal policy

The (optimal) solution of a problem with optimal substructure is made by composition of (optimal) solutions to subproblems, each having in turn optimal substructure



Sommario

Lecture 13: Alignment of sequences

- Sequence alignment
- Dot Matrix of two sequences
- Introduction to dynamic programming
- Longest common subsequence (LCS) problem

Longest common subsequence

LCS function defined

Let $X, Y \in Seq$ be the sequences to compare, and X_i, Y_j be the subsequences of their first i, j characters, respectively.
The integer function

$$LCS : Seq \times Seq \to Nat$$

gives the integer length of longest common subsequence of any two (sub)sequences, as follows:

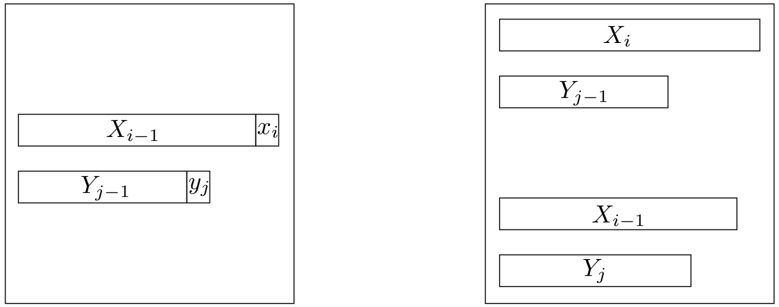
$$LCS(X_i, Y_j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ LCS(X_{i-1}, Y_{j-1}) + 1 & \text{if } x_i = y_j \\ \max(LCS(X_i, Y_{j-1}), LCS(X_{i-1}, Y_j)) & \text{if } x_i \neq y_j \end{cases}$$

Longest common subsequence

LCS function defined

$x_i = y_j$

$LCS(X_i, Y_j) = LCS(X_{i-1}, Y_{j-1}) + 1$



$x_i \neq y_j$

$LCS(X_i, Y_j) = \max(LCS(X_i, Y_{j-1}), LCS(X_{i-1}, Y_j))$

Recursive implementation

just write down in Python the recursive equations above

```
def cls(X,Y):
    i,j = len(X),len(Y)
    if i == 0 or j == 0: return 0
    elif X[i-1] == Y[j-1]: return cls(X[:i-1],Y[:j-1])+1
    else: return max(cls(X[:i],Y[:j-1]),cls(X[:i-1],Y[:j]))
```

```
print cls("BASKETBALL","BASEBALL") ≡ 8
```

OK !

```
print cls("ABRACADABRA","SUPERCALIFRAGILISTICESPIRALIDOSO")
```

VERY long execution time ... WHY ?

... because of recursion nonlinearity

the execution time is exponential with the sequence lengths

a recursion is said linear if the definition right-hand side contains at most one recursive function call

- nonlinear recursion: $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$ complexity: $O(2^n)$

```
def binomial(n,k):
    if k == 0 or n == k: return 1
    else: return binomial(n-1,k) + binomial(n-1,k-1)
```

- linear recursion: $\binom{n}{k} = \binom{n-1}{k-1} \times \frac{n}{k}$ complexity: $O(n)$

```
def binomial(n,k):
    if k == 0 or n == k: return 1
    else: return binomial(n-1,k-1) * n / k
```

Memoization technique

In computing, “memoization” is an optimization technique used primarily to speed up computer programs by having function calls avoid repeating the calculation of results for previously-processed input

- This technique of saving values that have already been calculated is frequently used
- Memoization is a means of lowering a function’s time cost in exchange for space cost; that is, memoized functions become optimized for speed in exchange for a higher use of computer memory space.
- An efficient LCS procedure requires: saving the solutions to one level of subproblem in a table so that the solutions are available to the next level of subproblems.

Length of the Longest Common Subsequence

computing the function *LCS* : *Seq* × *Seq* → *Nat* with memoization

```
def LCS(X, Y):
    m,n = len(X),len(Y)
    # An (m+1) times (n+1) matrix
    C = [[0] * (n+1) for i in range(m+1)]
    for i in range(1, m+1):
        for j in range(1, n+1):
            if X[i-1] == Y[j-1]:
                C[i][j] = C[i-1][j-1] + 1
            else:
                C[i][j] = max(C[i][j-1], C[i-1][j])
    return C
```

Usage example — LCSfunction

```
>>> X = "AATCC"
>>> Y = "ACACG"
>>> m = len(X)
>>> n = len(Y)
>>> C = LCS(X, Y)
```

```
>>> print C
[[0, 0, 0, 0, 0, 0],
 [0, 1, 1, 1, 1, 1],
 [0, 1, 1, 2, 2, 2],
 [0, 1, 1, 2, 2, 2],
 [0, 1, 2, 2, 3, 3],
 [0, 1, 2, 2, 3, 3]]
```

Usage example — LCSfunction

```
>>> X = "ATGGCCTGGAC"
>>> Y = "ATCCGGACC"
>>> m = len(X)
>>> n = len(Y)
>>> C = LCS(X, Y)
```

```
>>> print C
[[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [0, 1, 2, 2, 2, 2, 2, 2, 2, 2],
 [0, 1, 2, 2, 2, 3, 3, 3, 3, 3],
 [0, 1, 2, 2, 2, 3, 4, 4, 4, 4],
 [0, 1, 2, 3, 3, 3, 4, 4, 5, 5],
 [0, 1, 2, 3, 4, 4, 4, 4, 5, 6],
 [0, 1, 2, 3, 4, 4, 4, 4, 5, 6],
 [0, 1, 2, 3, 4, 5, 5, 5, 5, 6],
 [0, 1, 2, 3, 4, 5, 6, 6, 6, 6],
 [0, 1, 2, 3, 4, 5, 6, 7, 7, 7],
 [0, 1, 2, 3, 4, 5, 6, 7, 8, 8]]
```

Reading out an LCS

Backtracking on the table from the lower-right corner

```
def backTrack(C, X, Y, i, j):
     if i == 0 or j == 0:
         return ""
     elif X[i-1] == Y[j-1]:
         return backTrack(C, X, Y, i-1, j-1) + X[i-1]
     else:
         if C[i][j-1] > C[i-1][j]:
             return backTrack(C, X, Y, i, j-1)
         else:
             return backTrack(C, X, Y, i-1, j)
```

Usage example — backTrack function

```
>>> X = "AATCC"
>>> Y = "ACACG"
>>> m = len(X)
>>> n = len(Y)
>>> C = LCS(X, Y)
```

```
>>> print "Some␣LCS:␣'%s'" % backTrack(C, X, Y, m, n)
Some LCS: 'AAC'
```

Usage example — backTrack function

```
>>> X = "ATGGCCTGGAC"
>>> Y = "ATCCGGACC"
>>> m = len(X)
>>> n = len(Y)
>>> C = LCS(X, Y)
```

```
>>> print "Some␣LCS:␣'%s'" % backTrack(C, X, Y, m, n)
Some LCS: 'ATCCGGAC'
```

Reading out all LCSs

```
def backTrackAll(C, X, Y, i, j):
    if i == 0 or j == 0:
        return set([""])
    elif X[i-1] == Y[j-1]:
        return set([Z + X[i-1]
               for Z in backTrackAll(C, X, Y, i-1, j-1)
                   ])
    else:
        R = set()
        if C[i][j-1] >= C[i-1][j]:
            R.update(backTrackAll(C, X, Y, i, j-1))
        if C[i-1][j] >= C[i][j-1]:
            R.update(backTrackAll(C, X, Y, i-1, j))
        return R
```

Usage example — backTrackAll function

```
>>> X = "AATCC"
>>> Y = "ACACG"
>>> m = len(X)
>>> n = len(Y)
>>> C = LCS(X, Y)
```

```
>>> print "All␣LCSs:␣%s" % backTrackAll(C, X, Y, m, n)
All LCSs: set(['ACC', 'AAC'])
```

Usage example — backTrackAll function

```
>>> X = "ATGGCCTGGAC"
>>> Y = "ATCCGGACC"
>>> m = len(X)
>>> n = len(Y)
>>> C = LCS(X, Y)
```

```
>>> print "All␣LCSs:␣%s" % backTrackAll(C, X, Y, m, n)
All LCSs: set(['ATCCGGAC'])
```