CO7100/CO7200

Algorithms for Bioinformatics

22nd January – 27th March 2018

Thomas Erlebach
Dept. of Informatics
University of Leicester
Office: CS F20
Phone: 0116-2523411
e-mail: te17@leicester.ac.uk

Motivation

- Processing biological data often requires complex computations on large data volumes.
- Efficient algorithms are needed to solve such processing tasks.
- This module teaches the basic concepts of algorithms in the context of bioinformatics.
    - Design principles
    - Analysis of algorithms
    - Algorithm implementation in Java
    - Probabilistic models

Topics

- String matching
- Pairwise sequence alignment
- Hidden Markov models
- Restriction site mapping
- Multiple sequence alignment
- Phylogenetic trees
- Genome rearrangement

Literature

- **[DEKM] R. Durbin, S. Eddy, A. Krogh, G. Mitchison: Biological sequence analysis – Probabilistic models of proteins and nucleic acids. Cambridge University Press, 1998.**
- **[JP] N.C. Jones, P.A. Pevzner: An introduction to bioinformatics algorithms. MIT Press, 2004.**
- [CLRS] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein: Introduction to Algorithms, Second Edition. MIT Press, 2001.
- [G] D. Gusfield: Algorithms on Strings, Trees, and Sequences. Cambridge University Press, 1997.

Assessment

- CO7100: 50% Coursework, 50% Exam
  CO7200: 40% Coursework, 60% Exam
- Coursework (see module webpage for dates):
  - 2 problem-based worksheets:
    - Hand in to me at the beginning of the first Tuesday class on the day of the deadline
  - 1 programming assignment:
    - Submit by 11.59pm on the day of the deadline, departmental hand-in system

  The three pieces of coursework contribute 30%, 30% and 40% to the coursework mark.
- Additional unassessed worksheets

Introduction to algorithms

Algorithms

- Algorithm: set of rules for solving a problem in a finite number of steps
- "al-kawarizmi", a ninth-century Persian mathematician
- machine independent and language independent
- described in natural language or **pseudo-code**
- can be implemented in any programming language, e.g. Java

Pseudocode: Assignments

- Assignment statements:

```
$i \leftarrow 5$
$j \leftarrow i + 7$
```

  Assign the value on the right-hand side to the variable on the left-hand side.

- Array access:

```
$A[1] \leftarrow 8$
$A[2] \leftarrow 5$
$i \leftarrow 1$
**print** $A[i]$
```

  An array is a vector of variables of the same type.

Pseudocode: Conditionals

- if-statements:

```
if $i = 0$ then
  print "i is zero"

or

if $i = 0$ then
  print "i is zero"
else if $i > 0$ then
  print "i is positive"
else
  print "i is negative"
```

Execute commands based on whether a condition is true.

Pseudocode: for-Loops

- for-loop:

```
$s \leftarrow 0$
**for** $i \leftarrow 5$ **to** $10$ **do**
    $s \leftarrow s + i$
**print** $s$
```

  The body of the for-loop is executed for $i = 5$, $i = 6$, . . . , $i = 10$.

  This for-loop outputs 45 (it calculates $5 + 6 + 7 + 8 + 9 + 10$).

Pseudocode: while-Loops

- while-loop:

```
$i \leftarrow 0$
while $i * i \leq 10000$ do
    print $i * i$
    $i \leftarrow i + 1$
```

The body of the while-loop is executed as long as the while-condition holds.

If the while-condition is false initially, the body of the loop is not executed at all.

Pseudocode: Functions

- Function definition:

```
**Algorithm** Fibonacci($n$)
**Input:** *positive integer n*
**Output:** *n-th Fibonacci number*
$F[1] \leftarrow 1$
$F[2] \leftarrow 1$
**for** $i \leftarrow 3$ **to** $n$ **do**
    $F[i] \leftarrow F[i-1] + F[i-2]$
**return** $F[n]$
```

  Defines a function `Fibonacci` that takes a number n as parameter and computes the n-th Fibonacci number.

- Calling a function:

```
**print** Fibonacci(10)
```

Aspects of Algorithms

- Modelling the problem
  (often: statistical considerations)
- Designing an algorithm
  - divide-and-conquer, dynamic programming, greedy, backtracking, etc.
- Analysing the algorithm (worst case, average case)
- Algorithm implementation
- Experimental evaluation

Time complexity

- Running-time of the algorithm
- Want: general statement, not specific to machine, implementation and input
- Time complexity measured as number of elementary steps, specified as function of input size (or other parameters)
- Constant factors are usually ignored and the big-oh notation is used.
- Running-time $O(n^2)$ can mean $0.5 \cdot n^2 - 7$ or $20n^2 + 10n + 3$, for example.

Big-oh notation

- Formally, for functions f and g that map natural numbers to natural numbers, we say that f is $O(g)$ if there are constants $n_0 \in \mathbb{N}$ and $c \in \mathbb{N}$ such that $f(n) \leq c \cdot g(n)$ for all n larger than n_0.
- Intuitively, "f is $O(g)$" means "f is smaller or equal to g multiplied by an appropriate factor, for all n that are larger than a certain value n_0."
- $0.5n^2 - 7$ is $O(n^2)$ because $0.5n^2 - 7 \leq n^2$ for all n. (choose $c = 1$ and $n_0 = 1$)
- $20n^2 + 10n + 3$ is $O(n^2)$ because for $n \geq 10$:

  $$20n^2 + 10n + 3 \leq 20n^2 + n^2 + n^2 = 22n^2$$

  (choose $c = 22$ and $n_0 = 10$)

Worst-case time complexity

- Running-time of an algorithm as a function of the size or of other parameters of the input.
- Example: If the input is a string, it is natural to consider the running-time as a function of the length of the string.
- Worst-case analysis: we want a function f such that the algorithm makes at most $f(n)$ steps on **any** input of size n.
- If we find such an f and if that f is $O(g)$, we say that the algorithm has running-time (or time complexity) $O(g)$.

Classes of running-times

- linear running-time: $O(n)$ or $O(n+m)$
- quadratic running-time: $O(n^2)$ or $O(nm)$
- polynomial running-time: $O(n^k)$ for some constant k
- logarithmic running-time: $O(\log n)$
- exponential running time: $O(c^n)$ for some constant $c > 1$

Space complexity

- Similar to time complexity, one can also consider the amount of memory space an algorithm needs on inputs of a certain size.
- If a program's memory usage exceeds the main memory, swapping occurs and performance deteriorates.
- Therefore, the space complexity of an algorithm can also be very important.

NP-Hardness

- Many computational problems are either polynomial-time solvable or NP-hard.
- For NP-hard problems, no polynomial-time algorithm is known, and there is strong evidence (but no proof yet) that no such algorithm exists.
- Best known algorithms for NP-hard problems have exponential time complexity.
- Often impossible to compute optimal solutions for large instances of NP-hard problems in reasonable time.
- NP stands for "non-deterministic polynomial time".
- A problem that is NP-hard and can be solved non-deterministically in polynomial time is called NP-complete.

String matching

Strings

- String: sequence of symbols over a fixed-size alphabet Σ.
- $\Sigma = \{A, G, C, T\}$ for DNA sequences, $\Sigma =$ "set of amino acids" for protein sequences.
- String X of length m consists of symbols $X[0], X[1], \ldots,$ $X[m-1]$.
- $X[i..j]$ is substring of X from position i to j
- $X[0..i]$: prefix, $X[i..m-1]$: suffix
- proper prefix or suffix if not the whole string

String matching problem

- Input: two strings (the text T of length n and the pattern P of length m)
- Output: Does P occur in T? (If yes, output the positions of all occurrences of P in T.)
- P occurs in T at position i if $P = T[i..i+m-1]$
- Running-time of string matching algorithm: function of n and m.

A first approach

Algorithm 1: Naive String Matching

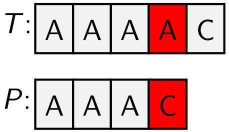
Input: pattern P, text T
Output: all occurrences of P in T

```
$m \leftarrow P$.length
$n \leftarrow T$.length
for $i \leftarrow 0$ to $n - m$ do
    $f \leftarrow$ true
    for $j \leftarrow 0$ to $m - 1$ do
        if $T[i + j] \neq P[j]$ then
            $f \leftarrow$ false
            break
    if $f =$ true then
        print "pattern occurs in position" $i$
```

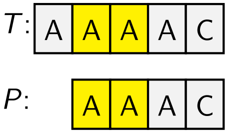
Analysis

- Outer for-loop: $n - m + 1$ iterations.
- Number of iterations of the inner for-loop is between 1 and m.
- In the worst case, number of steps is proportional to $(n - m + 1)m$.
- Thus, running-time is $O(nm)$.
- Worst-case running-time is achieved if, e.g., $T =$ AAA...AAA and $P =$ AAA...AA and $m \leq n/2$.
- Okay for small inputs ($n =$1,000, $m =$10), bad for large inputs ($n =$1,000,000, $m =$100,000).

Idea for improvement



(a)



(b)

- When checking for an occurrence in position 0, a mismatch in position 3 is detected, see (a).
- When checking for an occurrence in position 1, the two symbols in position 1 and 2 are checked again, see (b). But we already know that they match!
- Idea: Avoid such redundant work ➠ Knuth-Morris-Pratt algorithm

Avoiding redundant work

- Assume we have found that the first q symbols of P match T starting at position i, but the $q+1$-th symbol does not match.
- We want to determine the next possible starting position (after i) at which P could possibly occur in T.
- We also want to avoid comparing symbols that we already know a second time.
- For this, we can exploit the knowledge that the first q symbols of P match T from position i.

The prefix function

- Define prefix function π for pattern P as follows:

  For $q = 1, 2, \ldots, m$, $\pi[q]$ is the largest value of k such that $k < q$ and $P[0..k-1]$ is a suffix of $P[0..q-1]$.

- For $k = 0$, $P[0..k-1]$ is the empty string.
- $\pi[q]$ is the length of the longest proper suffix of $P[0..q-1]$ that is also a prefix of P.

How does it help?

- Assume that q symbols of P match T from position i to $j = i + q - 1$, while there is a mismatch at position $j + 1$.
- What is the next potential match?
  The one where P is aligned to T in such a way that its first $\pi[q]$ symbols match positions $j - \pi[q] + 1$ to j.
- Next thing to check is whether the next symbol of the pattern matches the symbol in position $j + 1$ of the text.
- In this way we avoid comparing symbols of the text up to position j a second time.

Algorithm 2: KMP Algorithm

Input: pattern P, text T
Output: all occurrences of P in T

```
$m \leftarrow P.\texttt{length}$
$n \leftarrow T.\texttt{length}$
$\pi \leftarrow \texttt{Compute-Prefix-Function}(P)$
$q \leftarrow 0$
for $i \leftarrow 0$ to $n-1$ do
    while $q > 0$ and $P[q] \neq T[i]$ do
        $q \leftarrow \pi[q]$
    if $P[q] = T[i]$ then
        $q \leftarrow q+1$
    if $q = m$ then
        print "pattern occurs in position" $i - m + 1$
        $q \leftarrow \pi[q]$
```

Running-time of KMP

- Prefix function can be computed in $O(m)$.
- for-loop has n iterations, and each iteration has a constant number of steps, except for the while-loop.
- Each iteration of the while-loop reduces q, and q is increased at most n times by one.
- Therefore, total work in while-loop is $O(n)$.
- Total running-time is $O(m+n)$, linear.

Algorithm 3: Compute-Prefix-Function(P)

Input: pattern P
Output: prefix function π for P

```
$m \leftarrow P.\texttt{length}$
$\pi[1] \leftarrow 0$
$k \leftarrow 0$
for $q \leftarrow 2$ to $m$ do
    while $k > 0$ and $P[k] \neq P[q-1]$ do
        $k \leftarrow \pi[k]$
    if $P[k] = P[q-1]$ then
        $k \leftarrow k + 1$
    $\pi[q] \leftarrow k$
```

Sequence alignment

Motivation

- Fundamental task in sequence analysis: determine whether two given sequences are related.
- Usually done by aligning the sequences (or parts thereof) and scoring the alignment.
- Assume that sequences diverged from a common ancestor via mutations (insertions, deletions, substitutions).
- Example of alignment of human alpha and beta globin, taken from [DEKM]:

```
HBA_HUMAN  GSAQVKGHGKKVADALTNAVAHVDDMPNALSALSDLHAHKL
           G+ +VK+HGKKV  A+++++AH+D++ +++++LS+LH  KL
HBB_HUMAN  GNPKVKAHGKKVLGAFSDGLAHLDNLKGTFATLSELHCDKL
```

Gapped alignments

- Gaps (“–” symbols) model insertions or deletions.
- Alignment of human alpha globin and leghaemoglobin from yellow lupin, taken from [DEKM]:

```
HBA_HUMAN   GSAQVKGHGKKVADALTNAVAHV---D--DMPNALSALSDLHAHKL
            ++ ++++H+ KV   + +A  ++          +L+ L+++H+ K
LGB2_LUPLU NNPELQAHAGKVFKLVYEAAIQLQVTGVVVTDATLKNLGSVHVSKG
```

- This is a structurally plausible alignment.
- But be careful: Sometimes spurious high-scoring alignments between proteins with completely different function and structure are found.

Global alignment

Global alignment of strings S and T:

- Possibly insert gaps into both strings, resulting in strings S' and T' of same length (consisting of real symbols and gap symbols)
- Write S' and T' above each other, aligning corresponding symbols.
- Reasonable assumption: S' and T' do not have gap symbols in the same position.
- Compute score of alignment based on aligned pairs of real symbols (residues) and gap penalties.

Scoring model

- Score an alignment by adding up scores for each pair of aligned real symbols and scores for each gap.
- Defined using **substitution matrix** and **gap penalties**.
- Probabilistic interpretation: score corresponds to the logarithm of the relative likelihood that the sequences are related, compared to being unrelated. (We come back to this later.)
- Interpretation assumes that mutations at different positions (sites) of the sequences occur independently. (Reasonable approximation for DNA and protein sequences.)

Substitution scores

- Specifies score $s(x, y)$ for each pair (x, y) of aligned symbols.
- Consider probability of observing x and y in related and in unrelated sequences: probability p_{xy} for related sequences, probability $q_x q_y$ for unrelated sequences.
- The **odds ratio** is: $\frac{p_{xy}}{q_x q_y}$.
- To obtain an additive scoring function, we use the **log-odds ratio**: $s(x, y) = \log \frac{p_{xy}}{q_x q_y}$.
- Add $s(x, y)$ for all pairs (x, y) of aligned symbols when computing score of an alignment (for alignments without gaps).

Substitution matrix

- The scores $s(x, y)$ can be arranged in a matrix; entry in position i, j represents the score of aligning the i-th symbol with the j-th symbol of the alphabet.
- This matrix is called **substitution matrix** or **score matrix**.
- For amino acids, 20×20 matrix.
- Popular examples of substitution matrices for protein sequences are the BLOSUM and PAM matrices.
- Entries are usually scaled and rounded to integers for computational efficiency.

BLOSUM50 matrix

	A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V
A	5	-2	-1	-2	-1	-1	-1	0	-2	-1	-2	-1	-1	-3	-1	1	0	-3	-2	0
R	-2	7	-1	-2	-4	1	0	-3	0	-4	-3	3	-2	-3	-3	-1	-1	-3	-1	-3
N	-1	-1	7	2	-2	0	0	0	1	-3	-4	0	-2	-4	-2	1	0	-4	-2	-3
D	-2	-2	2	8	-4	0	2	-1	-1	-4	-4	-1	-4	-5	-1	0	-1	-5	-3	-4
C	-1	-4	-2	-4	13	-3	-3	-3	-3	-2	-2	-3	-2	-2	-4	-1	-1	-5	-3	-1
Q	-1	1	0	0	-3	7	2	-2	1	-3	-2	2	0	-4	-1	0	-1	-1	-1	-3
E	-1	0	0	2	-3	2	6	-3	0	-4	-3	1	-2	-3	-1	-1	-1	-3	-2	-3
G	0	-3	0	-1	-3	-2	-3	8	-2	-4	-4	-2	-3	-4	-2	0	-2	-3	-3	-4
H	-2	0	1	-1	-3	1	0	-2	10	-4	-3	0	-1	-1	-2	-1	-2	-3	2	-4
I	-1	-4	-3	-4	-2	-3	-4	-4	-4	5	2	-3	2	0	-3	-3	-1	-3	-1	4
L	-2	-3	-4	-4	-2	-2	-3	-4	-3	2	5	-3	3	1	-4	-3	-1	-2	-1	1
K	-1	3	0	-1	-3	2	1	-2	0	-3	-3	6	-2	-4	-1	0	-1	-3	-2	-3
M	-1	-2	-2	-4	-2	0	-2	-3	-1	2	3	-2	7	0	-3	-2	-1	-1	0	1
F	-3	-3	-4	-5	-2	-4	-3	-4	-1	0	1	-4	0	8	-4	-3	-2	1	4	-1
P	-1	-3	-2	-1	-4	-1	-1	-2	-2	-3	-4	-1	-3	-4	10	-1	-1	-4	-3	-3
S	1	-1	1	0	-1	0	-1	0	-1	-3	-3	0	-2	-3	-1	5	2	-4	-2	-2
T	0	-1	0	-1	-1	-1	-1	-2	-2	-1	-1	-1	-1	-2	-1	2	5	-3	-2	0
W	-3	-3	-4	-5	-5	-1	-3	-3	-3	-3	-2	-3	-1	1	-4	-4	-3	15	2	-3
Y	-2	-1	-2	-3	-3	-1	-2	-3	2	-1	-1	-2	0	4	-3	-2	-2	2	8	-1
V	0	-3	-3	-4	-1	-3	-3	-4	-4	4	1	-3	1	-1	-3	-2	0	-3	-1	5

Example

- Alignment:

```
ADC
RNC
```

- BLOSUM50 score is:

$$s(\mathrm{A},\mathrm{R}) + s(\mathrm{D},\mathrm{N}) + s(\mathrm{C},\mathrm{C}) = -2 + 2 + 13 = 13$$

Gap penalties

- Standard method: define gap penalty $\gamma(g)$ for gaps of length g.
- **Linear gap penalty:** $\gamma(g) = -gd$
  for some constant d
- **Affine gap penalty:** $\gamma(g) = -d - (g-1)e$
  for constants d and e, where d is the **gap-open** penalty and e is the **gap-extension** penalty ($e \leq d$).
- For use with BLOSUM50, $d = 8$ in the linear case and $d = 12$, $e = 2$ in the affine case are reasonable choices.

Example

- Alignment:

```
AR---N
DRAACN
```

- Score using BLOSUM50:

$$s(\mathrm{A},\mathrm{D})+s(\mathrm{R},\mathrm{R})+\gamma(3)+s(\mathrm{N},\mathrm{N}) = -2+7+\gamma(3)+7 = 12+\gamma(3).$$

- With $\gamma(g) = -g \cdot 8$, the score of the alignment is $12 - 3 \cdot 8 = 12 - 24 = -12$.
- With $\gamma(g) = -12 - (g-1) \cdot 2$, the score of the alignment is $12 - 12 - 2 \cdot 2 = -4$.

Alignment algorithms

- For two given strings, we want to compute the (gapped) alignment of highest score.
- Trying out all possible alignments is infeasible (too many).
- Faster algorithms can be obtained using the algorithm design principle of **dynamic programming**:
  Compute optimal solutions for subproblems and then combine these to obtain the optimal solution for the original problem.
- Dynamic programming usually means filling in a matrix (whose entries are solutions to subproblems) and then reading off the solution from the matrix (traceback).

Needleman-Wunsch algorithm

- Compute global alignment of given strings X and Y with largest score (for given substitution matrix and linear gap penalties).
- Let X be of length n and Y be of length m.
- Positions in X (in Y) are indexed from 0 to $n-1$ (to $m-1$).
- Basic idea: compute optimal alignment scores for all pairs of strings $X[0..i-1]$ and $Y[0..j-1]$ for $0 \leq i \leq n$, $0 \leq j \leq m$.

Equations for subproblems

- Let $F(i,j)$ be the score of an optimal alignment between $X[0..i-1]$ and $Y[0..j-1]$. Then:

$$
\begin{aligned}
F(0,0) &= 0 \\
F(i,0) &= -id \\
F(0,j) &= -jd \\
F(i,j) &= \max \begin{cases} F(i-1,j-1) + s(X[i-1], Y[j-1]) \\ F(i-1,j) - d \\ F(i,j-1) - d \end{cases}
\end{aligned}
$$

Justification

- The last equation says that the largest possible score for aligning $X[0..i-1]$ and $Y[0..j-1]$ is obtained in one of three ways:
  - $X[i-1]$ is aligned to $Y[j-1]$, and to the left of this aligned pair of symbols we use an optimal alignment of $X[0..i-2]$ and $Y[0..j-2]$.
  - $X[i-1]$ is aligned with a gap symbol inserted into Y, and to the left of this we use an optimal alignment of $X[0..i-2]$ and $Y[0..j-1]$.
  - $Y[j-1]$ is aligned with a gap symbol inserted into X, and to the left of this we use an optimal alignment of $X[0..i-1]$ and $Y[0..j-2]$.
- This covers all possible cases.

Analysis

- If the entries of the matrix F are computed in order of increasing row and column indices (e.g., row by row), the three entries on the right-hand side of the equations have already been computed when $F(i,j)$ is computed.
- We can compute all entries of F in time $O(mn)$.
- The entry $F(n,m)$ gives us the score of the optimal alignment.
- How do we get the actual optimal alignment (not only its score)? Traceback!
- Total running-time is $O(nm)$.

Storing traceback information

- Remember for each $F(i,j)$ which of the three expressions on the right-hand side of the equation gave the maximum.
- For this, use a second matrix P with:

$$P(i,j) = \begin{cases} \mathrm{D} & \text{if } F(i,j) = F(i-1,j-1) + s(X[i-1], Y[j-1]) \\ \mathrm{T} & \text{if } F(i,j) = F(i-1,j) - d \\ \mathrm{L} & \text{if } F(i,j) = F(i,j-1) - d \end{cases}$$

  (D for diagonal, T for top, L for left)
- $P(i,j) = \mathrm{D}$ means: align $X[i-1]$ and $Y[j-1]$
  $P(i,j) = \mathrm{T}$ means: align $X[i-1]$ with '–'
  $P(i,j) = \mathrm{L}$ means: align '–' with $Y[j-1]$

Constructing the alignment

- $P(n, m)$ tells us the symbols in the last position of the optimal alignment.
- Continue with either $P(n-1, m-1)$ or $P(n, m-1)$ or $P(n-1, m)$, depending on $P(n, m)$; that entry tells us the symbols in the next-to-last position.
- Continue until we reach $P(0, 0)$.
- This process is called **traceback**.

Algorithm 4: Needleman-Wunsch

```
$n \leftarrow X$.length; $m \leftarrow Y$.length
for $i = 0$ to $n$ do
    for $j = 0$ to $m$ do
        if $i = 0$ and $j = 0$ then  $F(i,j) \leftarrow 0$
        else if $i = 0$ then
            $F(i,j) \leftarrow -jd$
            $P(i,j) \leftarrow \mathrm{L}$
        else if $j = 0$ then
            $F(i,j) \leftarrow -id$
            $P(i,j) \leftarrow \mathrm{T}$
        else
            $F(i,j) \leftarrow \max\{F(i-1,j-1) + s(X[i-1], Y[j-1]),$
                    $F(i-1,j) - d, F(i,j-1) - d\}$
            set $P(i,j)$ to D, T, or L depending on which
                    term yielded the maximum
return $F(n,m)$
```

Algorithm 5: Traceback

```
$n \leftarrow X$.length; $m \leftarrow Y$.length
$X' \leftarrow$ empty string; $Y' \leftarrow$ empty string; $i \leftarrow n$; $j \leftarrow m$
while $i + j > 0$ do
    if $P(i,j) = \mathrm{D}$ then
        insert $X[i-1]$ in the beginning of $X'$
        insert $Y[j-1]$ in the beginning of $Y'$
        $i \leftarrow i-1$; $j \leftarrow j-1$
    else if $P(i,j) = \mathrm{T}$ then
        insert $X[i-1]$ in the beginning of $X'$
        insert - in the beginning of $Y'$
        $i \leftarrow i-1$
    else
        insert - in the beginning of $X'$
        insert $Y[j-1]$ in the beginning of $Y'$
        $j \leftarrow j-1$
return $(X', Y')$
```

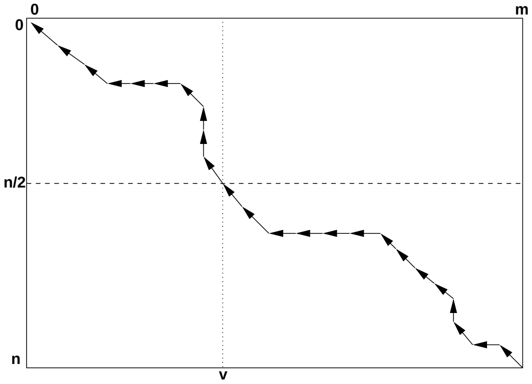
Space usage

- The Needleman-Wunsch algorithm has time complexity $O(nm)$ (or $O(n^2)$ if we assume that n and m are roughly the same).
- The algorithm uses quadratic space $O(nm)$ for storing F and P.
- For very long sequences, the space usage can become a problem.
- Thus, we are interested in **linear-space** algorithms with running-time $O(nm)$ for sequence alignment.

Linear-space sequence alignment

- If we only want the optimal score, we can modify Needleman-Wunsch so that it stores only the current row and the previous row of F and P. This already gives a linear-space algorithm.
- But to construct the optimal alignment, it seems that we need the whole P matrix.
- We can get around this problem by using the **divide-and-conquer** principle:
  Divide the problem into two smaller ones, solve these recursively, and then combine the solutions to get a solution for the original problem.

Divide and conquer



- Find column v such that optimal traceback path passes through entry (u, v), where $u = n/2$.
- Compute optimal alignment of $X[0..u-1]$ and $Y[0..v-1]$, and of $X[u..n-1]$ and $Y[v..m-1]$, recursively.

Finding the middle cell

- Compute $V(i,j)$ for each $i \geq u$ and $0 \leq j \leq m$:
  $V(i,j) = t$ if the traceback from $P(i,j)$ visits $P(u,t)$.
- $V(i,j) = \begin{cases} j & \text{if } i = u, \\ V(i',j') & \text{if } i > u,\ F(i,j) \text{ stems from } F(i',j') \end{cases}$
- $V(n,m)$ gives us the desired value v (the column of row $u = n/2$ through which the traceback from $P(n,m)$ passes).
- Can compute $F(n,m)$ and $V(n,m)$ in linear space, $O(m)$, by storing only current and previous row.
- Thus, total space requirement of resulting algorithm is linear.

Running-time

- Denote the running-time for input strings of length m and n by $T(n, m)$.
- Recurrence relation:

$$T(n,m) \leq \begin{cases} O(m), & \text{if } n \text{ is bounded by a constant} \\ hnm + T(n/2, v) + T(n/2, m-v), & \text{otherwise} \end{cases}$$

- Here, hnm (with suitable h) is the time to compute v and to combine the two optimal alignments obtained from the recursive calls.
- Solving this recurrence yields that $T(n, m)$ is $O(nm)$.
- Thus, we have an algorithm for optimal global alignments with linear gap penalties that uses linear space $O(n + m)$ and has time complexity $O(nm)$.

General gap penalties

- Arbitrary gap penalty $\gamma(g)$ (assuming $\gamma(g_1) + \gamma(g_2) \leq \gamma(g_1 + g_2)$)
- Idea: Consider all possible gap starting points when computing $F(i,j)$:

$$F(i,j) = \max \begin{cases} F(i-1,j-1) + s(X[i-1], Y[j-1]) \\ \max\{F(k,j) + \gamma(i-k) \mid 0 \leq k \leq i-1\} \\ \max\{F(i,k) + \gamma(j-k) \mid 0 \leq k \leq j-1\} \end{cases}$$

- Running-time $O(n^3)$ (if n and m are roughly the same), space complexity $O(n^2)$

Affine gap penalties

- $\gamma(g) = -d - (g-1)e$
- To get $O(nm)$ algorithm, keep three matrices:
    - $M(i,j)$ is the largest score of an alignment between $X[0..i-1]$ and $Y[0..j-1]$ for which $X[i-1]$ is aligned to $Y[j-1]$.
    - $I_X(i,j)$ is the largest score of an alignment between $X[0..i-1]$ and $Y[0..j-1]$ for which $X[i-1]$ is aligned to a gap.
    - $I_Y(i,j)$ is the largest score of an alignment between $X[0..i-1]$ and $Y[0..j-1]$ for which $Y[j-1]$ is aligned to a gap.
- Maximum of $M(n,m)$, $I_X(n,m)$ and $I_Y(n,m)$ gives score of optimal alignment.

Computation

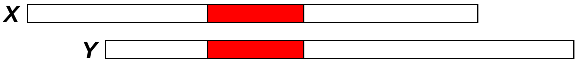
$$
\begin{aligned}
M(i,j) &= \max \begin{cases} M(i-1,j-1) + s(X[i-1], Y[j-1]) \\ I_X(i-1,j-1) + s(X[i-1], Y[j-1]) \\ I_Y(i-1,j-1) + s(X[i-1], Y[j-1]) \end{cases} \\
I_X(i,j) &= \max \begin{cases} M(i-1,j) - d \\ I_X(i-1,j) - e \end{cases} \\
I_Y(i,j) &= \max \begin{cases} M(i,j-1) - d \\ I_Y(i,j-1) - e \end{cases}
\end{aligned}
$$

(Assuming insertions not followed directly by deletions.)

Local alignment

Local alignments

- So far: align X to Y (global alignment)
- Now: identify substrings X' and Y' that have a high-scoring alignment:



- Useful if two protein sequences may share a common domain, or when comparing extended sections of genomic DNA sequence.
- Part of a sequence can be under strong enough selection to preserve detectable similarity, while the rest accumulates noise.

Problem definition

Local alignment problem:

- Given: two strings X and Y.
- Compute substrings X' of X and Y' of Y so that the largest-scoring alignment of X' and Y' has maximum score.

As scoring model, we use a substitution matrix and linear gap penalties.

Solvable via dynamic programming:

Smith-Waterman algorithm

Dynamic programming matrix

- Let n be length of X, m be length of Y.
- Let $X_i = X[0..i-1]$ denote the prefix of X of length i; X_0 is the empty string. (Define Y_j analogously.)
- Define dynamic programming matrix F:

  $F(i,j)$ is the largest score of an alignment between any suffix X' of X_i and any suffix Y' of Y_j.
- Once F has been computed, the largest value $F(i,j)$ gives the score of the best local alignment (and traceback from (i,j) can construct it).

Smith-Waterman Algorithm

Equations for computing F:

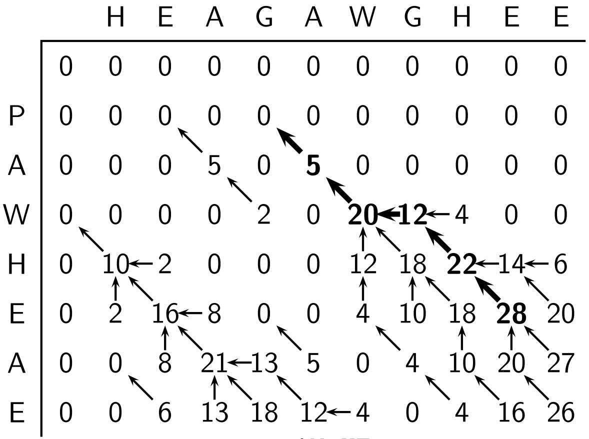
$$
\begin{aligned}
F(0,0) &= 0 \\
F(i,0) &= 0 \\
F(0,j) &= 0 \\
F(i,j) &= \max \begin{cases} 0 \\ F(i-1,j-1) + s(X[i-1], Y[j-1]) \\ F(i-1,j) - d \\ F(i,j-1) - d \end{cases}
\end{aligned}
$$

Difference from Needleman-Wunsch: if a matrix entry is negative, we replace it by 0 (corresponding to X' and Y' being the empty string).

Summary of Smith-Waterman

- Can compute all values $F(i,j)$ by filling the matrix row by row.
- Create second matrix P in which $P(i,j)$ tells us which of the four possible expressions gave the maximum in the computation of $F(i,j)$.
- In the end, find largest entry of matrix F; assume it is $F(i_0,j_0)$.
- This means score of best local alignment is $F(i_0,j_0)$, achieved by an alignment of a suffix of X_{i_0} and a suffix of Y_{j_0}.
- To construct the alignment (and X', Y'), use P matrix for traceback starting at $P(i_0,j_0)$ and ending at first (i_1,j_1) with $F(i_1,j_1)=0$.

Example



		H	E	A	G	A	W	G	H	E	E
	0	0	0	0	0	0	0	0	0	0	0
P	0	0	0	0	0	0	0	0	0	0	0
A	0	0	0	5	0	**5**	0	0	0	0	0
W	0	0	0	0	2	0	**20**	**12**	4	0	0
H	0	10	2	0	0	0	12	18	**22**	14	6
E	0	2	16	8	0	0	4	10	18	**28**	20
A	0	0	8	21	13	5	0	4	10	20	27
E	0	0	6	13	18	12	4	0	4	16	26

Best local alignment (score 28):

```
AW-HE
AWGHE
```

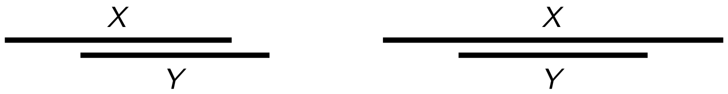
Remarks

- Algorithm can be adapted to affine gap penalties or general gap penalties.
- Subtle issues concerning suitable scoring models for local alignments:
    - Expected score for the alignment of a pair of random symbols must be negative.
    - Otherwise, longer alignments will tend to have larger scores than shorter ones, even if they are not biologically meaningful.
    - Then the algorithm would give a global or nearly global alignment as the best local alignment.

Other alignment variants

Dynamic programming solves also many other alignment problems (simply adapt the equations), e.g.:

- **Repeated matches**: Find different local alignments of parts of one sequence with non-overlapping parts of the other sequence (e.g., find many copies of a repeated domain or motif in a protein).
- **Overlap matches**:



Essentially a global alignment, but do not penalise overhanging ends (useful for DNA fragments).

Heuristic alignment algorithms

- Dynamic programming algorithms compute optimal alignments (exact algorithms), but have running-time at least $O(nm)$.
- When checking a sequence database of size n for alignments with a new sequence X of length m, running-time $O(nm)$ can mean several hours or days.
- **Heuristic** alignment algorithms do not guarantee optimal alignments, but are much faster and often useful in practice.
- Two popular packages: BLAST and FASTA

BLAST

- BLAST (Basic Local Alignment Search Tool) package
- Collection of programs for comparing gene and protein sequences against others in public databases.
- http://www.ncbi.nlm.nih.gov/BLAST/
- Basic idea: good local alignments are likely to include short stretches of identities or very high scoring matches.
- Initially look for such short stretches and use them as ‘seeds’ in search of a good local alignment.

BLAST Algorithm

- BLAST first makes a list of all ‘neighbourhood words’ of a fixed length (by default, 3 for protein sequences, 11 for nucleic acids) that would match the query sequence somewhere with a score higher than a threshold.
- It then scans through the database, and whenever it finds a word in this set, it starts a **hit extension** process to extend the possible match in both directions.
- Initial versions of BLAST produced only alignments without gaps, but more recent versions can also compute gapped alignments.

FASTA

- http://fasta.bioch.virginia.edu/
- Multistep approach to finding high-scoring local alignments between two sequences:
  - Starts from exact short word matches.
  - Finds maximal scoring ungapped extensions.
  - Combines them into gapped alignments.
- Highest-scoring gapped alignments are finally realigned using modified dynamic programming.

FASTA Details

- In first step, uses lookup table to locate all identically matching words of length *ktup* between the two sequences (*ktup* is typically 1 or 2 for protein sequences and 4 or 6 for DNA).
- Then looks for diagonals with many such matches.
- In second step, considers best diagonals and extends the short matches into maximal scoring ungapped regions (possibly joining several short matches).
- In the third step, combines several ungapped regions into gapped regions.

Significance of scores

Significance of scores

- We know how to compute high-scoring local or global alignments between two given sequences.
- How can we assess the significance of the score of an alignment?
  - How do we know if the produced alignment is a biologically meaningful alignment giving evidence for a homology, or just the best alignment between two entirely unrelated sequences?
- Questions can be addressed using statistical considerations.
- We will discuss the Bayesian approach and outline a second approach.
- Consider **only ungapped alignments** here.

Probability Theory

- $\Pr[A]$: probability of event A
- $\Pr[A, B]$: probability of event A and event B occurring together
- $\Pr[A \mid B] = \dfrac{\Pr[A, B]}{\Pr[B]}$: conditional probability of event A, assuming that B has occurred
- $\Pr[A, B] = \Pr[A \mid B] \cdot \Pr[B]$

Bayes' Theorem

- Bayes' Theorem:

$$\Pr[A \mid B] = \frac{\Pr[B \mid A] \cdot \Pr[A]}{\Pr[B]}$$

- Marginal probability:

$$\Pr[A] = \sum_i \Pr[A, B_i] = \sum_i \Pr[A \mid B_i] \cdot \Pr[B_i],$$

where the B_i are disjoint events covering all possible outcomes

The Bayesian approach

- Consider two probabilistic models for generating an alignment:
  - Model M (match): related sequences
  - Model R (random): unrelated sequences
- Let given sequences be

$$X = x_1 x_2 \cdots x_n$$

and

$$Y = y_1 y_2 \cdots y_n$$

- In the end, we would like to know the probability that the sequences X and Y are related:

$$\Pr[M \mid X, Y]$$

The random model

- Model R
- Assume that each symbol c has a probability q_c of appearing in any position of any of the two sequences.
- Probability of observing X and Y is:

$$\Pr[X, Y \mid R] = q_{x_1} q_{y_1} q_{x_2} q_{y_2} \cdots q_{x_n} q_{y_n} = \prod_{i=1}^{n} q_{x_i} q_{y_i}$$

The match model

- Model M
- Assume that for every pair of symbols c and d, there is a probability p_{cd} of observing c aligned to d.
- Probability of observing X aligned to Y is:

$$\Pr[X, Y \mid M] = p_{x_1 y_1} p_{x_2 y_2} \cdots p_{x_n y_n} = \prod_{i=1}^{n} p_{x_i y_i}$$

Recall our scoring model

- We used the log-odds ratio S to motivate our scoring model (log is to base e here):

$$\begin{aligned} S &= \log \frac{\Pr[X, Y \mid M]}{\Pr[X, Y \mid R]} \\ &= \log \frac{\prod_{i=1}^{n} p_{x_i y_i}}{\prod_{i=1}^{n} q_{x_i} q_{y_i}} \\ &= \sum_{i=1}^{n} \log \frac{p_{x_i y_i}}{q_{x_i} q_{y_i}} \end{aligned}$$

This is why we defined $s(x, y) = \log \frac{p_{xy}}{q_x q_y}$.

What we actually want

- Probability that X and Y are related: $\Pr[M \mid X, Y]$.
- For calculation, must assume *a priori* probability $\Pr[M]$ for related sequences and $\Pr[R]$ for unrelated sequences.
- Using Bayes' Theorem:

$$\begin{aligned}
\Pr[M \mid X, Y] &= \frac{\Pr[X, Y \mid M] \Pr[M]}{\Pr[X, Y]} \\
&= \frac{\Pr[X, Y \mid M] \Pr[M]}{\Pr[X, Y \mid M] \Pr[M] + \Pr[X, Y \mid R] \Pr[R]} \\
&= \frac{\Pr[X, Y \mid M] \Pr[M] / (\Pr[X, Y \mid R] \Pr[R])}{1 + \Pr[X, Y \mid M] \Pr[M] / (\Pr[X, Y \mid R] \Pr[R])}
\end{aligned}$$

Relation to scoring model

- We have:

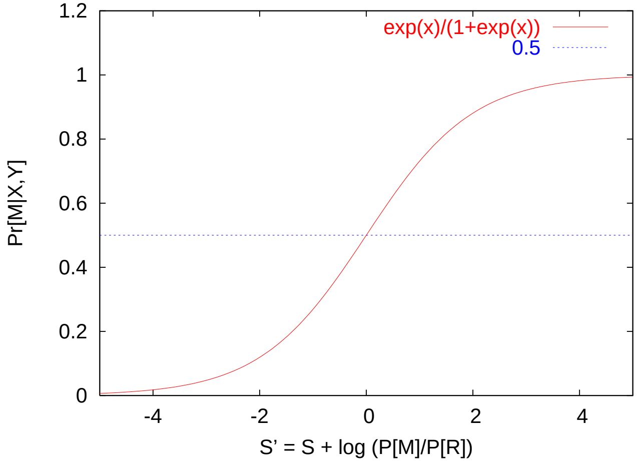
$$\Pr[M \mid X, Y] = \frac{\Pr[X, Y \mid M] \Pr[M] / (\Pr[X, Y \mid R] \Pr[R])}{1 + \Pr[X, Y \mid M] \Pr[M] / (\Pr[X, Y \mid R] \Pr[R])}$$

- With $S = \log \frac{\Pr[X,Y|M]}{\Pr[X,Y|R]}$ we get:

$$\log \frac{\Pr[X, Y \mid M] \cdot \Pr[M]}{\Pr[X, Y \mid R] \cdot \Pr[R]} = S + \log \frac{\Pr[M]}{\Pr[R]}$$

- With $S' = S + \log \frac{\Pr[M]}{\Pr[R]}$: $\Pr[M \mid X, Y] = \frac{e^{S'}}{1 + e^{S'}} = \sigma(S')$ where $\sigma(x) = \frac{e^x}{1+e^x}$ (**logistic** function).

Logistic function



- $\sigma(S')$ is 0.5 for $S' = 0$ and greater than 0.5 for $S' > 0$.

Conclusion from Bayesian approach

- It is meaningful to use a new score of $S' = S + \log \frac{\Pr[M]}{\Pr[R]}$ for an alignment, where S is the score defined via the log-odds ratio.
- Note that $\log \frac{\Pr[M]}{\Pr[R]}$ is usually negative, as $\Pr[M] < \Pr[R]$ in most cases.
- To see whether the two sequences X and Y are likely to be related, compare S' with 0.

Example

- Assume we look for alignments of X to strings in a database of N strings.
- If we expect that X will match only one of the N strings in the database, this corresponds to $\Pr[M] = 1/N$ and $\Pr[R] = 1 - 1/N$.
- In this case, $\log \frac{\Pr[M]}{\Pr[R]} = \log \frac{1/N}{1-1/N} \approx \log \frac{1}{N} = -\log N$.
- Hence, should **subtract** $\log N$ from the log-odds score of each alignment between X and a database string, and check for positive values.
- Equivalently, only alignments with a log-odds score S of more than $\log N$ should be considered significant.

Extreme value distribution

- Alternative approach to assessing significance of alignment scores.
- Consider scores for alignments of query string X with strings in database of N strings.
- Let M_N be the maximum of the alignment scores of X with N random strings (a random variable).
- Distribution of M_N can sometimes be approximated using *extreme value distribution*:

$$\Pr[M_N \leq x] \simeq \exp(-KNe^{\lambda(x-\mu)})$$

Using the second approach

- Let optimal alignment of X with a string Y in the database have score S.
- If probability that M_N is greater or equal to S is small (say, at most 0.05 or 0.01), conclude that X and Y are likely to be related.
- Probability that M_N is greater or equal to S can be approximated using extreme value distribution.

Deriving score parameters

Deriving score parameters

- Basis of our scoring models is log-odds ratio, using:

$$s(x,y) = \log \frac{p_{xy}}{q_x q_y}$$

- Estimate p_{xy}, q_x, q_y from known alignment data:
  - q_x: observed frequency of x
  - p_{xy}: observed frequency of x aligned to y in confirmed alignments of related sequences
- Difficulties:
  - Representative sample of sequences + alignments?
  - Alignments of 'related' sequences depend much on whether the sequences have diverged recently or a long time ago (leading to different values of p_{xy}).

PAM matrices

- PAM: Point Accepted Mutations
- Dayhoff, Schwartz and Orcutt (1978)
- Idea:
    - Obtain substitution data from alignments between very similar proteins
    - Extrapolate this information to longer evolutionary distances.

PAM matrices: Details

- Use alignments of neighbouring protein sequences in a phylogenetic tree to estimate the probabilities $\Pr[b \mid a, 1]$ that a is substituted by b after divergence time 1.
- Divergence time 1: expected number of substitutions 1%
- Then calculate $\Pr[b \mid a, t]$ for $t = 1, 2, \ldots$ (probability that a is substituted by b after divergence time t)
- For each t, scores $s(a, b \mid t)$ are defined by:

$$s(a, b \mid t) = \log \frac{\Pr[b \mid a, t] q_a}{q_a q_b} = \log \frac{\Pr[b \mid a, t]}{q_b}$$

- Resulting substitution matrices: PAMt
- Most widely used matrix is PAM250 (scaled by $3/\log 2$).

Weaknesses of PAM matrices

- Scores for larger values of t are calculated from those for $t = 1$ without taking into account the differences between short time substitutions and long term ones.
- But:
  - Short time substitutions are dominated by amino acid substitutions that arise from single base changes in codon triples.
  - Long term substitutions show all kinds of codon changes.
- Therefore, it is appropriate to use different scoring models depending on the expected divergence of the sequences we are comparing.

BLOSUM matrices

- After introduction of PAM, databases with multiple alignments of more distantly related proteins have been made.
- Can use these to derive score matrices more directly.
- Popular example: BLOSUM matrices, introduced by Henikoff and Henikoff in 1992.
- BLOSUM matrices were derived from a set of aligned, ungapped regions from protein families in the BLOCKS database.

BLOSUM: Details

- Cluster sequences from each block, putting two sequences in the same cluster when the percentage of identical residues exceeds $L\%$.
- Calculate frequencies A_{ab} of observing residue a in one cluster aligned to b in another cluster (normalised by the sizes of the clusters).
- Estimate q_a and p_{ab} using

$$q_a = \frac{\sum_b A_{ab}}{\sum_{cd} A_{cd}} \qquad \text{and} \qquad p_{ab} = \frac{A_{ab}}{\sum_{cd} A_{cd}}.$$

- Define scores as $s(a,b) = \log \frac{p_{ab}}{q_a q_b}$ (scaled and rounded to integers)

Popular BLOSUM matrices

- Two popular BLOSUM matrices are:
  - **BLOSUM50**
    - good for gapped alignments
    - obtained for $L = 50\%$
    - scaled by $3/\log 2$
  - **BLOSUM62**
    - good for ungapped alignments
    - obtained for $L = 62\%$
    - scaled by $2/\log 2$
- Lower values of L correspond to longer evolutionary time, and are applicable for more distant searches.

Power of DNA sequence comparison

- After a new gene is found, biologists usually have no idea about its function.
- A common approach to inferring the function of a newly sequenced gene is to use alignment algorithms to find similarities with genes whose function is known.
- Examples of important discoveries (from [JP]):
  - Link between cancer-causing genes and normal growth genes.
  - Elucidating the nature of cystic fibrosis.

Cancer-causing genes

- In 1984, scientists compared the newly discovered cancer-causing ν-sis oncogene with known genes.
- Surprising finding: cancer-causing gene matches a normal gene involved in growth and development, called platelet-derived growth factor (PDGF).
- As a result, scientists became suspicious that cancer might be caused by a normal growth gene being switched on at the wrong time.

Cystic fibrosis

- Cystic fibrosis is a fatal disease associated with abnormal secretions.
- Diagnosed in children at a rate of 1 in 3900.
- A defective gene causes the body to produce abnormally thick mucus that clogs the lungs and leads to lifethreatening infections.
- More than 10 million Americans are unknown and symptomless carriers of the defective cystic fibrosis gene.
- Each time two carriers have a child, 25% chance that child will have cystic fibrosis.

Cystic fibrosis discovery

- In 1989, search for the cystic fibrosis gene had been narrowed to a region of 1 million nucleotides on the chromosome 7.
- When that region was sequenced, biologists compared it against a database of known genes.
- Found similarities between some segment within the region and a known gene that codes for adenosine triphosphate (ATP) binding proteins. These proteins span the cell membrane multiple times as part of the ion transport channel.
- This is a plausible function for a cystic fibrosis gene, as the disease involves sweat secretions with abnormally high sodium content.