CHALMERS UNIVERSITY OF TECHNOLOGY

Examination in Bioinformatics, MVE360

Monday 14 March 2016, 08:30-12:30

Examiner:	Graham Kemp (telephone 772 54 11, room 6475 EDIT) The examiner will visit the exam room at 09:30 and 11:30.
Results:	Will be published by 6 April 2016 at the latest.
Exam review:	See course web page for time and place: http://www.cse.chalmers.se/edu/year/2016/course/MVE360/
Grades:	Grades are normally determined as follows: ≥ 48 for grade 5; ≥ 36 for grade 4; ≥ 24 for grade 3.
Help material:	A Chalmers-approved calculator is permitted. English language dictionaries are allowed.

Specific instructions:

- Check that you have received:
  - question paper (4 pages)
  - Perl reference guide (4 pages)
  - HMM formula collection (3 pages)
- Please answer in English where possible. You may clarify your answers in Swedish if you are not confident you have expressed yourself correctly in English.
- Begin the answer to each question on a new page.
- Write clearly; unreadable = wrong!
- Fewer points are given for unnecessarily complicated solutions.
- Indicate clearly if you make any assumptions that are not given in the question.
- Write the page number and question number on every page.

Question 1. 4 p

a) Consider a certain human protein and the corresponding proteins (orthologs) in a set of other mammals. In addition, consider the corresponding mRNAs encoding all these proteins. What can you say about the rate of amino acid change in the protein as compared to the rate of nucleotide change in the mRNA? If there is a difference, provide at least one explanation for this difference.
(2p)

b) Consider the nucleotide sequence alignment below (four sequences).

```
A. GAAAGGGCG
B. GAGTGGTCA
C. CAGCGGTCG
D. GAAGGGGCG
```

What is a likely phylogenetic tree according the "maximum parsimony" principle? Motivate your answer.
(2p)

Question 2. 4 p

Assuming a match score of 2, a mismatch score of -1 and a gap score of -2, derive the score matrix for a *local* alignment of "CCTC" and "CTCCT".

In this case, what is the score of an optimal local alignment?
How many alignments have this optimal score?
What are these alignments?

(4p)

Question 3. 6 p

a) Draw a suffix trie and a suffix tree for the string "AGTAG".
(3p)

b) Find the Burrows-Wheeler Transform for the string "AGTAG".
(3p)

Question 4. 3 p

What output is printed when the following program is run?

```
#!/usr/bin/perl -w

$s = "CCUCCUCCC"; if ( $s =~ /CU(.*)$/ )           { print "a) $1\n"; }
$s = "CCUCCUCCC"; if ( $s =~ /U(.+)C/ )            { print "b) $1\n"; }
$s = "CCUCCUCCC"; if ( $s =~ /(..)(.*)\1(.*)\1/ )  { print "c) $1,$2,$3\n"; }

$s = "CCUCCUCCC"; $s =~ s/.U./U/g     ; print "d) $s\n";
$s = "CCUCCUCCC"; $s =~ s/U.*U/T/     ; print "e) $s\n";
$s = "CCUCCUCCC"; $s =~ s/(...)/\1T/g ; print "f) $s\n";
```

(3p)

Question 5. 13 p

A "cuckoo RNA" is an RNA molecule that contains a stem loop structure where the loop contains the motif CCUCCUCCC. The following sequence (in FASTA format) contains two "cuckoo stem loops":

```
>RNA example
CUGCGAGGUCCCACCUCCUCCCGGGACCUAGCAGCAGCG
GCGACGCCUCUCCUCCUCCCUGCUGUCGCCGU
```

The structure of this RNA molecule is as follows:

a) Write a Perl program that reads a FASTA file whose name is specified on the command line and prints a message indicating whether the sequence read contains the substring "CCUCCUCCC".
(3p)

b) Extend your program so that it tests whether the sequence contains a "cuckoo stem loop". The program should look for the substring CCUCCUCCC where the 10 nucleotides immediately before this substring include (at least) five consecutive nucleotides whose reverse complement is present within the 10 nucleotides that follow CCUCCUCCC.
(Looking at the stem loop at the left of the figure, the 10 nucleotides before the substring are CGAGGUCCCA and the 10 nucleotides that follow are GGGACCUAGC. A subsequence of 7 nucleotides and its reverse complement are underlined.)
(7p)

c) Modify your program so that it counts how many "cuckoo stem loops" are present in the sequence.
(You do not have to rewrite the entire program — just write the new code and any parts of the solution to (b) that are changed.)
(3p)

Question 6. 8 p

a) What is a hidden Markov model? How can it be used to model phenomena we observe in the world around us? How is it used in Bioinformatics? Give at least 2 examples of phenomena that can be described by an HMM (Do not include gene prediction as an example!).
(3p)

b) Suppose you want to identify genes in a long genome, describe an HMM for this task. Draw the model, describe state space and parameters. Explain mathematically how the parameter training is done.
(2p)

c) What are pair HMMs? What are pair HMMs used for? What are the differences in relation to a standard HMM?
(3p)

Question 7. 8 p

Imagine that you are a bioinformatician working in a lab whose current project is to sequence a fish species that has been recently discovered and therefore never sequenced before. The species has been called *Danio dimidiatus*. Your colleagues have sequenced and assembled the *Danio dimidiatus* genome and determined that its closest relative is the species *Danio rerio*, the zebrafish, whose genome is well studied and fully annotated. Your task is to annotate the *Danio dimidiatus* genome.

Describe in detail how you would annotate the new genome. Describe all the steps you need to take. Include the model description, state space, parameters. Describe how the parameters are trained.

(8p)

Question 8. 14 p

a) In a metagenomics sequencing experiment the Illumina HiSeq 2500 machine is used for sequencing. After demultiplexing you retrieve two fastq files per sample from the sequencing company. The first four lines in each of the files `sample1_1.fastq` and `sample1_2.fastq` are shown below.

sample1_1.fastq

```
@HWI-ST1035_0090:7:1101:1655:1998#GACCCA/1
TCTGTTTGAGCAAAAATTAAAGCAAGTGGTGCCTGTATATGGTATCGAGCTTAAATTG
+HWI-ST1035_0090:7:1101:1655:1998#GACCCA/1
ccceeggggggifhiiiiiiiiiiiiiibfhghihiigfhiiii^fgiiheghiiiiiii
```

sample1_2.fastq

```
@HWI-ST1035_0090:7:1101:1655:1998#GACCCA/2
AGGTTCCCTGATTCTTTACCGTTCCATGGAACGTCTTGGCCATCATCATTGACCAATT
+HWI-ST1035_0090:7:1101:1655:1998#GACCCA/2
eeeddgfgggghiiiiiiiiigiiiiiiiiiihihffiiiiiiiiiihiefhihiidefg
```

i) Explain how the two reads shown here are related.
(3p)

ii) Explain what information is contained in the fourth line of each file.
(3p)

b) In *shotgun metagenomics* we are often interested in identifying what functions the microbial community is capable of. Describe briefly the steps involved in finding genes or functions in the metagenome, starting from the raw sequence data.
(5p)

c) In RNA-seq, when aligning reads to a reference genome sequence you can use a mapper built for genome sequencing applications such as BWA or Bowtie2 if you are working with bacterial species. For eukaryotes it is preferable to use a mapper specially designed for RNA sequencing such as Tophat or Star. Explain why.
(3p)

Perl 5 Reference Guide (Extract)

Literals

Numeric:
- **123**
- **1_234**
- **123.4**
- **5E-10**
- **0xff** (hex)
- **0377** (octal)

String:
- **'abc'**
  literal string, no variable interpolation or escape characters, except **\'** and ****. Also: **q/abc/**. Almost any pair of delimiters can be used instead of **/.../**.
- **"abc"**
  Variables are interpolated and escape sequences are processed. Also: **qq/abc/**.
  Escape sequences: **\t** (Tab), **\n** (Newline), **\r** (Return), **\f** (Formfeed), **\b** (Backspace), **\a** (Alarm), **\e** (Escape), **\033** (octal), **\x1b** (hex), **\c[** (control)
  \l and **\u** lowercase/uppercase the following character. **\L** and **\U** lowercase/uppercase until a **\E** is encountered. **\Q** quote regular expression characters until a **\E** is encountered.
- **'**COMMAND**'**
  evaluates to the output of the COMMAND. Also: **qx/**COMMAND**/**.

Array:
- **(1, 2, 3)**. **()** is an empty array.
- **(1..4)** is the same as **(1,2,3,4)**,
  likewise **('a'..'z')**.
- **qw/foo bar .../** is the same as **('foo','bar',...)**.

Array reference:
- **[1,2,3]**

Hash (associative array):
- **(**KEY1**,** VAL1**,** KEY2**,** VAL2**,...)**
  Also **(**KEY1**=>** VAL1**,** KEY2**=>** VAL2**,...)**

Hash reference:
- **{**KEY1**,** VAL1**,** KEY2**,** VAL2**,...}**

Code reference:
- **sub {**STATEMENTS**}**

Filehandles:
- **<STDIN>**, **<STDOUT>**, **<STDERR>**, **<ARGV>**, **<DATA>**.
  User-specified: HANDLE, **$**VAR.

Globs:
- **<**PATTERN**>** evaluates to all filenames according to the pattern. Use **<${**VAR**}>** or **glob $**VAR to glob from a variable.

Here-Is:
- **<<**IDENTIFIER
  Shell-style "here document."

Special tokens:
- **__FILE__**: filename; **__LINE__**: line number; **__END__**: end of program; remaining lines can be read using the filehandle **DATA**.

Variables

$var	a simple scalar variable.
$var[28]	29th element of array **@var**.
$p = \@var	now **$p** is a reference to array **@var**.
$$p[28]	29th element of array referenced by **$p**. Also, **$p->[28]**.
$var[-1]	last element of array **@var**.
$var[$i][$j]	**$j**th element of the **$i**th element of array **@var**.
$var{'Feb'}	one value from hash (associative array) **%var**.
$p = \%var	now **$p** is a reference to hash **%var**.

$$p{'Feb'}	a value from hash referenced by **$p**. Also, **$p->{'Feb'}**.
$#var	last index of array **@var**.
@var	the entire array; in a scalar context, the number of elements in the array.
@var[3,4,5]	a slice of array **@var**.
@var{'a','b'}	a slice of **%var**; same as **($var{'a'},$var{'b'})**.
%var	the entire hash; in a scalar context, **true** if the hash has elements.
$var{'a',1,...}	emulates a multidimensional array.
('a'...'z')[4,7,9]	a slice of an array literal.
PKG**::**VAR	a variable from a package, e.g., **$pkg::var**, **@pkg::ary**.
\OBJECT	reference to an object, e.g., **\$var**, **\%hash**.
***NAME**	refers to all objects represented by NAME. ***n1 = *n2** makes **n1** an alias for **n2**. ***n1 = $n2** makes **$n1** an alias for **$n2**.

You can always use a **{**BLOCK**}** returning the right type of reference instead of the variable identifier, e.g., **${...}**, **&{...}**. **$$p** is just a shorthand for **${$p}**.

Operators

******	Exponentiation
+ - * /	Addition, subtraction, multiplication, division
%	Modulo division
& | ^	Bitwise AND, bitwise OR, bitwise exclusive OR
>> <<	Bitwise shift right, bitwise shift left
|| &&	Logical OR, logical AND
.	Concatenation of two strings
x	Returns a string or array consisting of the left operand (an array or a string) repeated the number of times specified by the right operand

All of the above operators also have an assignment operator, e.g., **.=**

->	Dereference operator
****	Reference (unary)
! ~	Negation (unary), bitwise complement (unary)
++ --	Auto-increment (magical on strings), auto-decrement
== !=	Numeric equality, inequality
eq ne	String equality, inequality
< >	Numeric less than, greater than
lt gt	String less than, greater than
<= >=	Numeric less (greater) than or equal to
le ge	String less (greater) than or equal to
<=> cmp	Numeric (string) compare. Returns -1, 0, 1.
=~ !~	Search pattern, substitution, or translation (negated)
..	Range (scalar context) or enumeration (array context)
?:	Alternation (if-then-else) operator
,	Comma operator, also list element separator. You can also use **=>**.
not	Low-precedence negation
and	Low-precedence AND
or xor	Low-precedence OR, exclusive OR

All Perl functions can be used as list operators, in which case they have very high or very low precedence, depending on whether you look at the left or the right side of the operator. Only the operators **not**, **and**, **or** and **xor** have lower precedence.

A "list" is a list of expressions, variables, or lists. An array variable or an array slice may always be used instead of a list.

Parentheses can be added around the parameter lists to avoid precedence problems.

Statements

Every statement is an expression, optionally followed by a modifier, and terminated by a semicolon. The

semicolon may be omitted if the statement is the final one in a BLOCK.

Execution of expressions can depend on other expressions using one of the modifiers **if**, **unless**, **while** or **until**, for example:

```
EXPR1 if EXPR2 ;
EXPR1 until EXPR2 ;
```

The logical operators **||**, **&&** or **?:** also allow conditional execution:

```
EXPR1 || EXPR2 ;
EXPR1 ? EXPR2 : EXPR3 ;
```

Statements can be combined to form a BLOCK when enclosed in **{}**. Blocks may be used to control flow:

```
if (EXPR) BLOCK [ [ elsif (EXPR) BLOCK ... ] else BLOCK ]
unless (EXPR) BLOCK [ else BLOCK ]
[ LABEL: ] while (EXPR) BLOCK [ continue BLOCK ]
[ LABEL: ] until (EXPR) BLOCK [ continue BLOCK ]
[ LABEL: ] for (EXPR; EXPR; EXPR) BLOCK
[ LABEL: ] foreach VAR² (LIST) BLOCK
[ LABEL: ] BLOCK [ continue BLOCK ]
```

Program flow can be controlled with:

goto LABEL	Continue execution at the specified label.
last [LABEL]	Immediately exits the loop in question. Skips continue block.
next [LABEL]	Starts the next iteration of the loop.
redo [LABEL]	Restarts the loop block without evaluating the conditional again.

Special forms are:

```
do BLOCK while EXPR ;
do BLOCK until EXPR ;
```

which are guaranteed to perform BLOCK once before testing EXPR, and

```
do BLOCK
```

which effectively turns BLOCK into an expression.

Structure Conversion

pack TEMPLATE, LIST
: Packs the values into a binary structure using TEMPLATE.

unpack TEMPLATE, EXPR
: Unpacks the structure EXPR into an array, using TEMPLATE.

TEMPLATE is a sequence of characters as follows:

a / **A**	ASCII string, null- / space-padded
b / **B**	Bit string in ascending / descending order
c / **C**	Native / unsigned char value
f / **d**	Single / double float in native format
h / **H**	Hex string, low / high nybble first
i / **I**	Signed / unsigned integer value
l / **L**	Signed / unsigned long value
n / **N**	Short / long in network (big endian) byte order
s / **S**	Signed / unsigned short value
u / **P**	Uuencoded string / pointer to a string
v / **V**	Short / long in VAX (little endian) byte order
x / **@**	Null byte / null fill until position
X	Backup a byte

Each character may be followed by a decimal number that will be used as a repeat count; an asterisk (*****) specifies

all remaining arguments. If the format is preceded with **%**N, **unpack** returns an N-bit checksum instead. Spaces may be included in the template for readability purposes.

String Functions

chomp LIST²
: Removes line endings from all elements of the list; returns the (total) number of characters removed.

chop LIST²
: Chops off the last character on all elements of the list; returns the last chopped character.

crypt PLAINTEXT, SALT
: Encrypts a string.

eval EXPR²
: EXPR is parsed and executed as if it were a Perl program. The value returned is the value of the last expression evaluated. If there is a syntax error or runtime error, an undefined string is returned by **eval**, and **$@** is set to the error message. See also **eval** in section Miscellaneous.

index STR, SUBSTR [, OFFSET]
: Returns the position of SUBSTR in STR at or after OFFSET. If the substring is not found, returns **-1** (but see **$[** in section Special Variables).

length EXPR²
: Returns the length in characters of the value of EXPR.

lc EXPR
: Returns a lowercase version of EXPR.

lcfirst EXPR
: Returns EXPR with the first character lowercase.

quotemeta EXPR
: Returns EXPR with all regular expression metacharacters quoted.

rindex STR, SUBSTR [, OFFSET]
: Returns the position of the last SUBSTR in STR at or before OFFSET.

substr EXPR, OFFSET [, LEN]
: Extracts a substring of length LEN out of EXPR and returns it. If OFFSET is negative, counts from the end of the string. May be assigned to.

uc EXPR
: Returns an uppercased version of EXPR.

ucfirst EXPR
: Returns EXPR with the first character uppercased.

Array and List Functions

delete $HASH**{**KEY**}**
: Deletes the specified value from the specified hash. Returns the deleted value unless HASH is **tie**d to a package that does not support this.

each %HASH
: Returns a 2-element array consisting of the key and value for the next value of the hash. Entries are returned in an apparently random order. After all values of the hash have been returned, a null array is returned. The next call to **each** after that will start iterating again.

exists EXPR²
: Checks if the specified hash key exists in its hash array.

grep EXPR, LIST

grep BLOCK LIST
: Evaluates EXPR or BLOCK for each element of the LIST, locally setting **$_** to refer to the element. Modifying **$_** will modify the corresponding element from LIST. Returns the array of elements from LIST for which EXPR returned **true**.

join EXPR, LIST
: Joins the separate strings of LIST into a single string with fields separated by the value of EXPR, and returns the string.

keys %HASH
: Returns an array with all of the keys of the named hash.

map EXPR, LIST

map BLOCK LIST
: Evaluates EXPR or BLOCK for each element of the LIST, locally setting **$_** to refer to the element. Modifying **$_** will modify the corresponding element from LIST. Returns the list of results.

pop @ARRAY
: Pops off and returns the last value of the array.

push @ARRAY, LIST

Pushes the values of LIST onto the end of the array.

reverse LIST
: In array context, returns the LIST in reverse order. In scalar context: returns the first element of LIST with bytes reversed.

scalar @ARRAY
: Returns the number of elements in the array.

scalar %HASH
: Returns a **true** value if the hash has elements defined.

shift [@ARRAY]
: Shifts the first value of the array off and returns it, shortening the array by 1 and moving everything down. If @ARRAY is omitted, shifts **@ARGV** in main and **@_** in subroutines.

sort [SUBROUTINE] LIST
: Sorts the LIST and returns the sorted array value. SUBROUTINE, if specified, must return less than zero, zero, or greater than zero, depending on how the elements of the array (available to the routine as **$a** and **$b**) are to be ordered. SUBROUTINE may be the name of a user-defined routine, or a BLOCK.

splice @ARRAY, OFFSET [, LENGTH [, LIST]]
: Removes the elements of @ARRAY designated by OFFSET and LENGTH, and replaces them with LIST (if specified). Returns the elements removed.

split [PATTERN [, EXPR[2] [, LIMIT]]]
: Splits a string into an array of strings, and returns it. If LIMIT is specified, splits into at most that number of fields. If PATTERN is also omitted, splits at the whitespace. If not in array context, returns number of fields and splits to @_. See also Search and Replace Functions.

unshift @ARRAY, LIST
: Prepends list to the front of the array, and returns the number of elements in the new array.

values %HASH
: Returns a normal array consisting of all the values of the named hash.

Regular Expressions

Each character matches itself, unless it is one of the special characters **+ ? . * ^ $ ( ) [ ] { } |** \. The special meaning of these characters can be escaped using a \.

.	matches an arbitrary character, but not a newline unless it is a single-line match (see **m//s**).
(...**)**	groups a series of pattern elements to a single element.
^	matches the beginning of the target. In multiline mode (see **m//m**) also matches after every newline character.
$	matches the end of the line. In multiline mode also matches before every newline character.
[... **]**	denotes a class of characters to match. **[^** ... **]** negates the class.
(... **|** ... **|** ... **)**	matches one of the alternatives.
(?# TEXT **)**	Comment.
(?: REGEXP **)**	Like **(**REGEXP**)** but does not make back-references.
(?= REGEXP **)**	Zero width positive look-ahead assertion.
(?! REGEXP **)**	Zero width negative look-ahead assertion.
(? MODIFIER **)**	Embedded pattern-match modifier. MODIFIER can be one or more of **i**, **m**, **s**, or **x**.

Quantified subpatterns match as many times as possible. When followed with a **?** they match the minimum number of times. These are the quantifiers:

+	matches the preceding pattern element one or more times.
?	matches zero or one times.
*****	matches zero or more times.
{N**,**M**}**	denotes the minimum N and maximum M match count. **{**N**}** means exactly N times; **{**N**,}** means at least N times.

A \ escapes any special meaning of the following character if non-alphanumeric, but it turns most alphanumeric characters into something special:

\w	matches alphanumeric, including _, **\W** matches non-alphanumeric.
\s	matches whitespace, **\S** matches non-whitespace.
\d	matches numeric, **\D** matches non-numeric.
\A	matches the beginning of the string, **\Z** matches the end.
\b	matches word boundaries, **\B** matches non-boundaries.
\G	matches where the previous **m//g** search left off.
\n, **\r**, **\f**, **\t**	etc. have their usual meaning.

\w, **\s** and **\d** may be used within character classes, **\b** denotes backspace in this context.

Back-references:

\1 ... **\9**	refer to matched subexpressions, grouped with **()**, inside the match.
\10	and up can also be used if the pattern matches that many subexpressions.

See also **$1** ... **$9**, **$+**, **$&**, **$`**, and **$'** in section Special Variables.

With modifier **x**, whitespace can be used in the patterns for readability purposes.

Search and Replace Functions

[EXPR **=~**] [**m**] **/**PATTERN**/** [**g**] [**i**] [**m**] [**o**] [**s**] [**x**]
: Searches EXPR (default: **$_**) for a pattern. If you prepend an **m** you can use almost any pair of delimiters instead of the slashes. If used in array context, an array is returned consisting of the subexpressions matched by the parentheses in the pattern, i.e., **($1,$2,$3,...)**.
Optional modifiers: **g** matches as many times as possible; **i** searches in a case-insensitive manner; **o** interpolates variables only once. **m** treats the string as multiple lines; **s** treats the string as a single line; **x** allows for regular expression extensions.
If PATTERN is empty, the most recent pattern from a previous match or replacement is used.
With **g** the match can be used as an iterator in scalar context.

?PATTERN**?**
: This is just like the **/**PATTERN**/** search, except that it matches only once between calls to the **reset** operator.

[**$**VAR **=~**] **s/**PATTERN**/**REPLACEMENT**/** [**e**] [**g**] [**i**] [**m**] [**o**] [**s**] [**x**]
: Searches a string for a pattern, and if found, replaces that pattern with the replacement text. It returns the number of substitutions made, if any; if no substitutions are made, it returns **false**.
Optional modifiers: **g** replaces all occurrences of the pattern; **e** evaluates the replacement string as a Perl expression; for any other modifiers, see **/**PATTERN**/** matching. Almost any delimiter may replace the slashes; if single quotes are used, no interpretation is done on the strings between the delimiters, otherwise the strings are interpolated as if inside double quotes.
If bracketing delimiters are used, PATTERN and REPLACEMENT may have their own delimiters, e.g., **s(foo)[bar]**. If PATTERN is empty, the most recent pattern from a previous match or replacement is used.

[**$**VAR **=~**] **tr/**SEARCHLIST**/**REPLACEMENTLIST**/** [**c**] [**d**] [**s**]
: Translates all occurrences of the characters found in the search list with the corresponding character in the replacement list. It returns the number of characters replaced. **y** may be used instead of **tr**.
Optional modifiers: **c** complements the SEARCHLIST; **d** deletes all characters found in SEARCHLIST that do not have a corresponding character in REPLACEMENTLIST; **s** squeezes all sequences of characters that are translated into the same target character into one occurrence of this character.

pos SCALAR
: Returns the position where the last **m//g** search left off for SCALAR. May be assigned to.

study [**$**VAR[2]]
: Study the scalar variable **$**VAR in anticipation of performing many pattern matches on its contents before the variable is next modified.

Input / Output

In input/output operations, FILEHANDLE may be a filehandle as opened by the **open** operator, a predefined filehandle (e.g., **STDOUT**) or a scalar variable that evaluates to the name of a filehandle to be used.

<FILEHANDLE**>**
: In scalar context, reads a single line from the file opened on FILEHANDLE. In array context, reads the whole file.

< >
: Reads from the input stream formed by the files specified in **@ARGV**, or standard input if no arguments were supplied.

close FILEHANDLE
: Closes the file or pipe associated with the filehandle.

open FILEHANDLE [, FILENAME]
: Opens a file and associates it with FILEHANDLE. If FILENAME is omitted, the scalar variable of the same name as the FILEHANDLE must contain the filename.
The following filename conventions apply when opening a file.

"FILE" open FILE for input. Also "<FILE".
">FILE" open FILE for output, creating it if necessary.
">>FILE" open FILE in append mode.
"+<FILE" open FILE with read/write access (file must exist).
"+>FILE" open FILE with read/write access (file truncated).
"|CMD" opens a pipe to command CMD; forks if CMD is -.
"CMD|" opens a pipe from command CMD; forks if CMD is -.
FILE may be &FILEHND in which case the new filehandle is connected to the (previously opened) filehandle FILEHND. If it is &=N, FILE will be connected to the given file descriptor. **open** returns **undef** upon failure, **true** otherwise.
Returns a pair of connected pipes.

print [FILEHANDLE] [LIST[2]]
Equivalent to **print** FILEHANDLE **sprintf** LIST.

Special Variables

The following variables are global and should be localized in subroutines:

$_	The default input and pattern-searching space.
$.	The current input line number of the last filehandle that was read.
$/	The input record separator, newline by default. May be multicharacter.
$,	The output field separator for the print operator.
$"	The separator that joins elements of arrays interpolated in strings.
$\	The output record separator for the print operator.
$#	The output format for printed numbers. Deprecated.
$*	Set to 1 to do multiline matching within strings. Deprecated, see the **m** and **s** modifiers in section Search and Replace Functions.
$?	The status returned by the last `...` command, pipe **close** or **system** operator.
$]	The perl version number, e.g., 5.001.
$[	The index of the first element in an array, and of the first character in a substring. Default is 0. Deprecated.
$;	The subscript separator for multidimensional array emulation. Default is "\034".
$!	If used in a numeric context, yields the current value of **errno**. If used in a string context, yields the corresponding error string.
$@	The Perl error message from the last **eval** or **do** EXPR command.
$:	The set of characters after which a string may be broken to fill continuation fields (starting with ^) in a format.
$0	The name of the file containing the Perl script being executed. May be assigned to.
$$	The process ID of the currently executing Perl program. Altered (in the child process) by **fork**.
$<	The real user ID of this process.
$>	The effective user ID of this process.
$(	The real group ID of this process.
$)	The effective group ID of this process.
$^A	The accumulator for **formline** and **write** operations.
$^D	The debug flags as passed to perl using -D.
$^F	The highest system file descriptor, ordinarily 2.
$^I	In-place edit extension as passed to Perl using -i.
$^L	Formfeed character used in formats.
$^P	Internal debugging flag.
$^T	The time (as delivered by **time**) when the program started. This value is used by the file test operators -M, -A and -C.
$^W	The value of the -w option as passed to Perl.
$^X	The name by which the currently executing program was invoked.

The following variables are context dependent and need not be localized:

$%	The current page number of the currently selected output channel.
$=	The page length of the current output channel. Default is 60 lines.
$-	The number of lines remaining on the page.
$~	The name of the current report format.
$^	The name of the current top-of-page format.
$|	If set to nonzero, forces a flush after every write or print on the currently selected output channel. Default is 0.
$ARGV	The name of the current file when reading from <>.

The following variables are always local to the current block:

$&	The string matched by the last successful pattern match.
$`	The string preceding what was matched by the last successful match.
$'	The string following what was matched by the last successful match.
$+	The last bracket matched by the last search pattern.
$1...$9...	Contain the subpatterns from the corresponding sets of parentheses in the last pattern successfully matched. $10... and up are only available if the match contained that many subpatterns.

Special Arrays

@ARGV	Contains the command-line arguments for the script (not including the command name).
@EXPORT	Names the methods a package exports by default.
@EXPORT_OK	Names the methods a package can export upon explicit request.
@INC	Contains the list of places to look for Perl scripts to be evaluated by the **do** FILENAME and **require** commands.
@ISA	List of base classes of a package.
@_	Parameter array for subroutines. Also used by **split** if not in array context.
%ENV	Contains the current environment.
%INC	List of files that have been included with **require** or **do**.
%OVERLOAD	Can be used to overload operators in a package.
%SIG	Used to set signal handlers for various signals.

HMM formula collection

Standard HMMs

Hidden Markov Process: $X_1^T = X_1, \dots, X_T$
Observed process: $Y_1^T = Y_1, \dots, Y_T$

State space: $S = \{s_1, \dots, s_N\}$
Initial probabilities: $\pi_i = P(X_1 = i),\ i = 1, \dots, N$
Transition probabilities: $a_{ij} = P(X_{t+1} = j | X_t = i), i, j \in S$
Emission probabilities: $b_j(Y_t | Y_1^{t-1}) = P(Y_t | X_t = j, Y_1^{t-1})$

$P(X_1^T, Y_1^T) = \pi_{X_1} b_{X_1}(Y_1) \prod_{t=2}^{T} a_{X_{t-1}, X_t} b_{X_t}(Y_t | Y_1^{t-1})$

Forward Algorithm

$\alpha_i(t) = P(Y_1^t, X_t = i)$

Initialization: $\alpha_i(0) = \pi_i, i = 1, \dots, N$

Tabular computation: $\alpha_i(t) = \sum_{j \in S} \alpha_j(t-1) a_{ji} b_i(Y_t | Y_1^{t-1}), i = 1, \dots, N, t = 1, \dots, T$

Termination: $\alpha_i(T+1) = \sum_{j \in S} \alpha_j(T) a_{ji}, i = 1, \dots, N$

Backward Algorithm

$\beta_i(t) = P(Y_{t+1}^T | Y_1^t, X_t = i)$

Initialization: $\beta_i(T+1) = 1, i = 1, \dots, N$

Tabular computation: $\beta_i(t) = \sum_{j \in S} \beta_j(t+1) a_{ij} b_j(Y_{t+1} | Y_1^t), i = 1, \dots, N, t = T, T-1, \dots, 0$

Termination: none

The Viterbi Algorithm

$\delta_i(t) = \max_{X_1^{t-1}} P(Y_1^t, X_1^{t-1}, X_t = i)$

Initialization: $\delta_i(0) = \pi_i, i = 1, \dots, N$

Tabular computation: $\delta_i(t) = \max_{1 \le j \le N} \delta_j(t-1) a_{ji} b_i(Y_t | Y_1^{t-1}), i = 1, \dots, N, t = 1, \dots, T$

$$\psi_i(t) = \operatorname{argmax}_{1 \le j \le N} \delta_j(t-1) a_{ji} b_i(Y_t | Y_1^{t-1}), i = 1, \dots, N, t = 1, \dots, T$$

Termination: $\delta_i(T+1) = \max_{1 \le j \le N} \delta_j(T) a_{ji}, i = 1, \dots, N$

$$\psi_i(T+1) = \operatorname*{argmax}_{1 \le j \le N} \delta_j(T) a_{ji}, i = 1, \dots, N$$

Traceback: $X_{T+1}^* = \psi_i(T+1)$

$$X_t^* = \psi_{X_{t+1}^*}(t+1), t = T, T-1, \dots, 0$$

The Baum-Welch Algorithm

$\gamma_i(t) = P(X_t = i|Y_1^T) = \frac{\alpha_i(t)\beta_i(t)}{\sum_{j\in S}\alpha_j(t)\beta_j(t)}, i = 1, \dots, N, t = 1, \dots, T$

$\xi_{ij}(t) = P(X_t = i, X_{t+1} = j|Y_1^T), i = 1, \dots, N, t = 1, \dots, T$

Initialization: pick arbitrary model parameters $\{\pi_i, a_{ij}, b_j : i, j = 1, \dots, N\}$

Recursion:

1. E-phase: calculate the forward and the backward algorithms for the current parameter settings.
2. M-phase: produce new parameter estimates using the re-estimation formulas below.

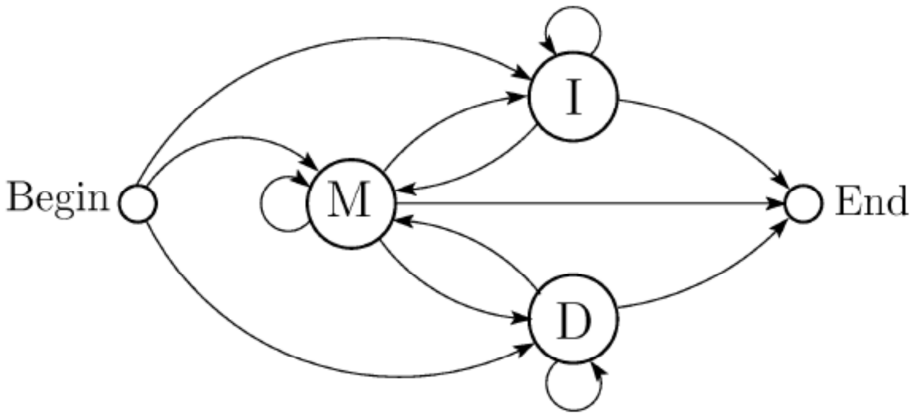
Re-estimation formulas:

$\hat{\pi}_i = \gamma_i(1)$

$$\hat{a}_{ij} = \frac{\sum_{t=1}^{T-1}\xi_{ij}(t)}{\sum_{t=1}^{T-1}\gamma_i(t)}$$

$$\hat{b}_j(c) = \frac{\sum_{t=1, Y_t=c}^{T-1}\gamma_j(t)}{\sum_{t=1}^{T-1}\gamma_j(t)}$$

Pair HMMs



State durations:

$$(d_l, e_l) = \begin{cases} (1,1) \text{ if } X_l = M \\ (1,0) \text{ if } X_l = I \\ (0,1) \text{ if } X_l = D \end{cases}$$

$$p_l = \sum_{k=1}^{L} d_k, \qquad q_l = \sum_{k=1}^{L} e_k$$

$p_0 = q_0 = 0, p_L = T, q_L = U$

The Viterbi algorithm for PHMMs

Initialization: $\delta_i(0,0) = \pi_i$

$\delta_i(t,0) = \delta_i(0,u) = 0, t > 0, u > 0$

Tabular computation:

$$\delta_M(t,u) = b_M(Y_t, Z_u | Y_1^{t-1}, Z_1^{u-1}) \cdot \max \begin{cases} \delta_M(t-1,u-1)a_{MM} \\ \delta_I(t-1,u)a_{IM} \\ \delta_D(t,u-1)a_{DM} \end{cases}$$

$$\delta_I(t,u) = b_I(Y_t, -|Y_1^{t-1}, Z_1^{u}) \cdot \max \begin{cases} \delta_M(t-1,u-1)a_{MI} \\ \delta_I(t-1,u)a_{II} \end{cases}$$

$$\delta_D(t,u) = b_D(-, Z_u | Y_1^{t}, Z_1^{u-1}) \cdot \max \begin{cases} \delta_M(t-1,u-1)a_{MD} \\ \delta_D(t,u-1)a_{DD} \end{cases}$$

Termination:

$$\delta_{\mathrm{M}}(T+1,U+1) = \max \begin{cases} \delta_M(T,U)a_{MM} \\ \delta_I(T,U+1)a_{IM} \\ \delta_D(T+1,U)a_{DM} \end{cases}$$

$$\delta_I(T+1,U+1) = \max \begin{cases} \delta_M(T,U)a_{MI} \\ \delta_I(T,U+1)a_{II} \end{cases}$$

$$\delta_I(T+1,U+1) = \max \begin{cases} \delta_M(T,U)a_{MD} \\ \delta_D(T+1,U)a_{DD} \end{cases}$$