whogrep.pl

```
#!/usr/bin/perl

#
# file:           whogrep.pl
# purpose:        run the UNIX ’who’ command and print lines matching
#                 pattern given as the first command line argument.
#                 e.g.
#                     unix> ./whogrep.pl "ˆk"
#                     unix> ./whogrep.pl "ˆ[ˆ ]{4} "
#

open(INPUT, "who |");

while ( <INPUT> ) {
        if ( /$ARGV[0]/ ) {
                print;
        }
}
```

get_uniprot.pl

lynx is a general purpose browser for the World Wide Web.
The command:

```
unix> lynx -source URL
```

writes the HTML source of the document identified by the given URL to standard output.

```
#!/usr/bin/perl

open(INPUT,
     "lynx -source http://www.expasy.org/uniprot/$ARGV[0].txt |");

while ( <INPUT> ) {
        print;
}
```

draw_mol.pl (1)

```
#!/usr/bin/perl

open(PDB_FILE, "1itb.pdb") ||
        die "Can’t open 1itb.pdb: $!\n";

open(OUT_A, ">chainA") || die "Can’t open xy_A: $!\n";
open(OUT_B, ">chainB") || die "Can’t open xy_B: $!\n";

while ( <PDB_FILE> ) {
        if ( /ˆATOM.* CA .* A / ) {
                print OUT_A substr($_, 30, 16), "\n";
        } elsif ( /ˆATOM.* CA .* B / ) {
                print OUT_B substr($_, 30, 16), "\n";
        }
}
```

draw_mol.pl (2)

```
close OUT_A;
close OUT_B;

open(GNUPLOT, "| gnuplot > picture.eps");

print GNUPLOT <<EOF;
set terminal postscript eps
plot "chainA" with lines, "chainB" with lines
exit
EOF
```

We can then view the file picture.eps with a PostScript previewer, such as gv.

longest_sub.pl

```
#!/usr/bin/perl
$sequence = "";
while ( <> ) {     # Read sequence in FASTA format
        chomp;
        if ( /ˆ[ˆ>]/ ) {
                $sequence .= uc($_);
        }
}
$n = length($sequence);
$reverse_complement = reverse($sequence);
$reverse_complement =˜ tr/ACGT/TGCA/;
$_ = $sequence . "X" . $reverse_complement;
while ( $n > 0 ) {
        if ( /(.{$n}).*X.*\1/ ) {
                print $1, "\n";
                $n = 0;
        } else {
                $n--;
        }
}
```

All the same

Can test whether all of the characters match with the first character of the string:

```
/^(.)\1*$/
```

- The first character of the string matches with (.)
- This is followed by zero or more occurrences of the same character.
- The $ ensures that there are no other characters in the string.

cpg_islands.pl (incomplete)

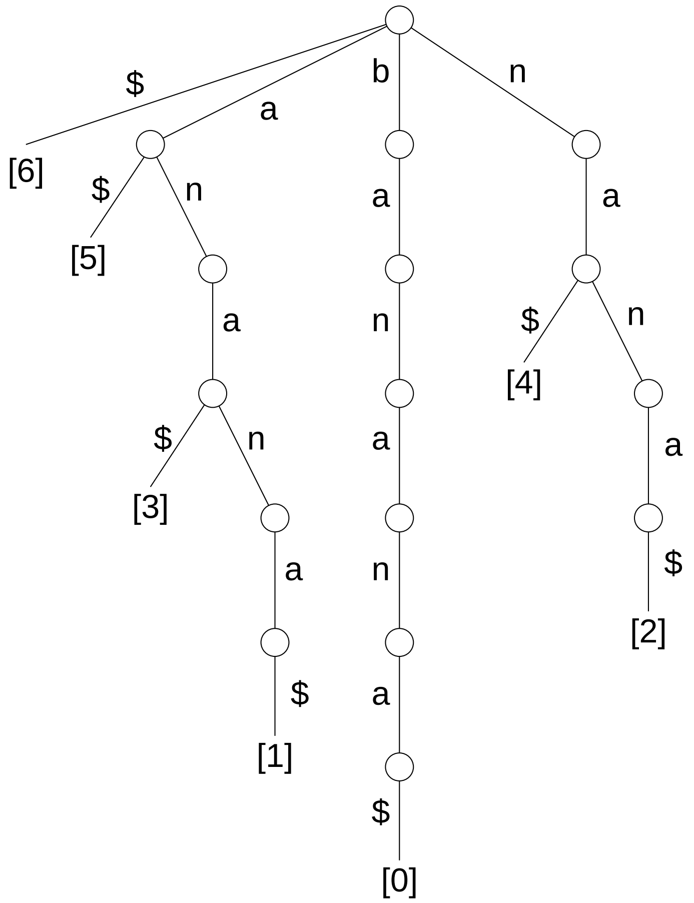
```
#!/usr/bin/perl

%log_likelyhood_ratio = (
  "AA" => -0.740, "AC" => 0.419, "AG" => 0.580, "AT" => -0.803,
  "CA" => -0.913, "CC" => 0.302, "CG" => 1.812, "CT" => -0.685,
  "GA" => -0.624, "GC" => 0.461, "GG" => 0.331, "GT" => -0.730,
  "TA" => -1.169, "TC" => 0.573, "TG" => 0.393, "TT" => -0.679 );

#
# Read sequence in FASTA format.
#

$sequence = "";
while ( <> ) {
        chomp;
        if ( /^[^>]/ ) {
                $sequence .= uc($_);
        }
}
```

Suffix tree

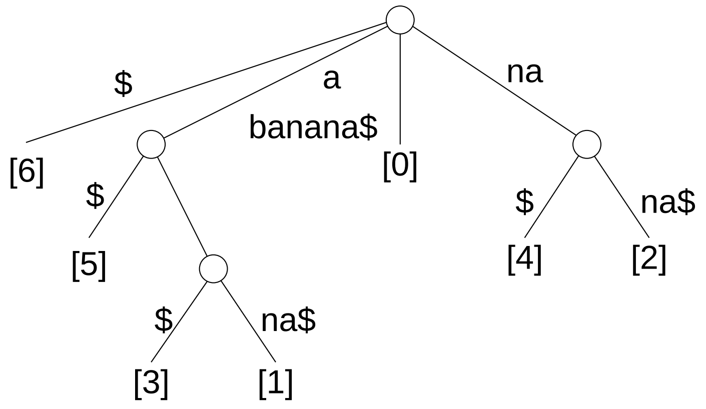
All suffixes

banana$	[0]
anana$	[1]
nana$	[2]
ana$	[3]
na$	[4]
a$	[5]
$	[6]

Suffix trie



Suffix tree



Sum of pairs score for a multiple sequence alignment

- could imagine generalising substitution matrix to N-dimensions, but is there good data to determine reliable scores?
- one alternative approach is to use the sum of pairs

Compute the sum of column scores, where each column score is:

$$\sum_{i<j} s(a_{i,} a_j)$$

where a_i and a_j are the residues in that column from sequences i and j, and $s(x, y)$ is a score taken from a substitution matrix (e,g, from the BLOSUM or PAM families).

This score is simple to compute, but a drawback is that it assumes that all sequences in the set are separated by the same evolutionary distance.

Multiple sequence alignment

Dynamic programming

- in principle this could be done, using the sum of pairs approach for scoring matches/mismatches
- possible for a few short sequences
- not practical for many long sequences

Progressive alignment

- perform pairwise alignment between all pairs of sequences
- construct a guide tree based on distances between each pair
- add sequences into the multiple alignment in the order given by the guide tree
- “once a gap, always a gap”

Progressive alignment

```
Sequence 1: MGLPKSFVSM
Sequence 2: MGVPKTFVSM
Sequence 3: MGVPKTFVASM
Sequecne 4: MGGLPKSYAVSM

1: MGLPKSFVSM       1: MGLPKSFV-SM       1: M-GLPKS-FVSM
   || || |||| (2)      || || || || (3)      | |||||  ||| (3)
2: MGVPKTFVSM       3: MGVPKTFVASM       4: MGGLPKSYAVSM

                    2: MGVPKTFV-SM       2: M-GVPK-TFVSM
                       |||||||| || (1)      | | ||   ||| (5)
                    3: MGVPKTFVASM       4: MGGLPKSYAVSM

                                         3: M-GVPKTFVASM
                                            | | ||    || (5)
                                         4: MGGLPKSYAVSM

Sequence 1: M-GLPKSFV-SM
Sequence 2: M-GVPKTFV-SM
Sequence 3: M-GVPKTFVASM
Sequence 4: MGGLPKSYAVSM
```

“How Perl Saved the Human Genome Project” (Lincoln Stein)

http://www.bioperl.org/wiki/How_Perl_saved_human_genome

Perl has been the solution of choice for genome centers whenever they need to exchange data, or to retrofit one center’s software module to work with another center’s system.

So Perl has become the software mainstay for computation within genome centers as well as the glue that binds them together. Although genome informatics groups are constantly tinkering with other "high level" languages such as Python, Tcl and recently Java, nothing comes close to Perl’s popularity. How has Perl achieved this remarkable position?

“How Perl Saved the Human Genome Project” (Lincoln Stein)

1. Perl is remarkably good for slicing, dicing, twisting, wringing, smoothing, summarizing and otherwise mangling text. Although the biological sciences do involve a good deal of numeric analysis now, most of the primary data is still text: clone names, annotations, comments, bibliographic references. Even DNA sequences are textlike. Interconverting incompatible data formats is a matter of text mangling combined with some creative guesswork. Perl’s powerful regular expression matching and string manipulation operators simplify this job in a way that isn’t equalled by any other modern language.

“How Perl Saved the Human Genome Project” (Lincoln Stein)

2. Perl is forgiving. Biological data is often incomplete, fields can be missing, or a field that is expected to be present once occurs several times (because, for example, an experiment was run in duplicate), or the data was entered by hand and doesn’t quite fit the expected format. Perl doesn’t particularly mind if a value is empty or contains odd characters. Regular expressions can be written to pick up and correct a variety of common errors in data entry. Of course this flexibility can be also be a curse.

3. Perl is component-oriented. Perl encourages people to write their software in small modules, either using Perl library modules or with the classic Unix tool-oriented approach. External programs can easily be incorporated into a Perl script using a pipe, system call or socket. The dynamic loader introduced with Perl5 allows people to extend the Perl language with C routines or to make entire compiled libraries available for the Perl interpreter.

“How Perl Saved the Human Genome Project” (Lincoln Stein)

4. Perl is easy to write and fast to develop in. The interpreter doesn’t require you to declare all your function prototypes and data types in advance, new variables spring into existence as needed, calls to undefined functions only cause an error when the function is needed. The debugger works well with Emacs and allows a comfortable interactive style of development.

5. Perl is a good prototyping language. Because Perl is quick and dirty, it often makes sense to prototype new algorithms in Perl before moving them to a fast compiled language. Sometimes it turns out that Perl is fast enough so that of the algorithm doesn’t have to be ported; more frequently one can write a small core of the algorithm in C, compile it as a dynamically loaded module or external executable, and leave the rest of the application in Perl.

“How Perl Saved the Human Genome Project” (Lincoln Stein)

6. Perl is a good language for Web CGI scripting, and is growing in importance as more labs turn to the Web for publishing their data.

My experience in using Perl in a genome center environment has been extremely favorable overall. However I find that Perl has its problems too. Its relaxed programming style leads to many errors that more uptight languages would catch. For example, Perl lets you use a variable before its been assigned to, a useful feature when that’s what you intend but a disaster when you’ve simply mistyped a variable name. Similarly, it’s easy to forget to declare make a variable used in a subroutine local, inadvertently modifying a global variable.