

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 "^k"
#              unix> ./whogrep.pl "^[^ ]{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, "litb.pdb") ||
    die "Can't open litb.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 ( /^ATOM.* CA .* A / ) {
        print OUT_A substr($_, 30, 16), "\n";
    } elsif ( /^ATOM.* CA .* B / ) {
        print OUT_B substr($_, 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 ( /^>/ ) {
        $sequence .= uc($_);
    }
}
$n = length($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_likelihood_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.
#

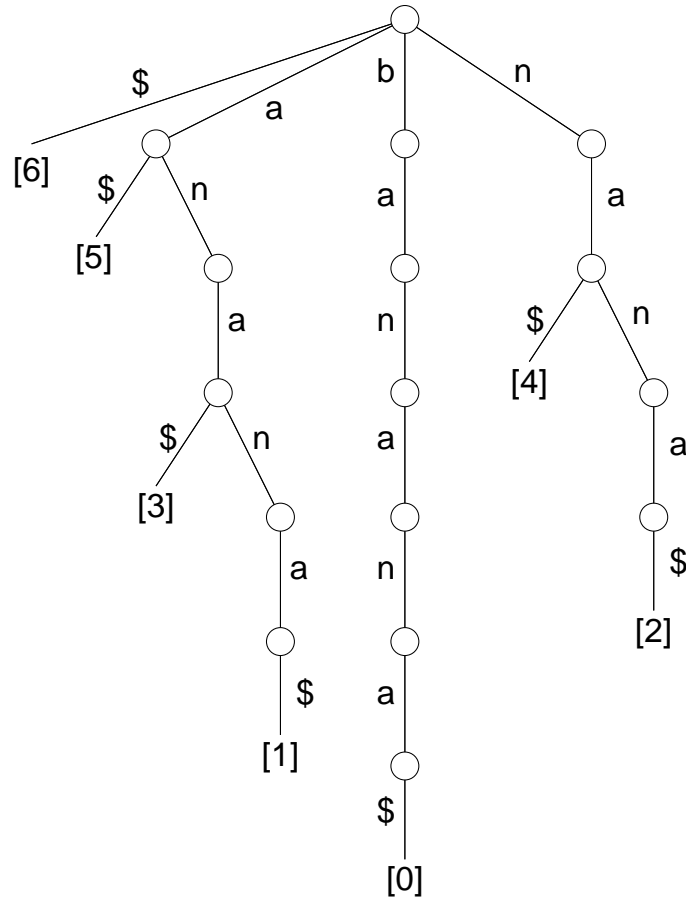
$sequence = "";
while ( <> ) {
    chomp;
    if ( /^>/ ) {
        $sequence .= uc($_);
    }
}
```

Suffix tree

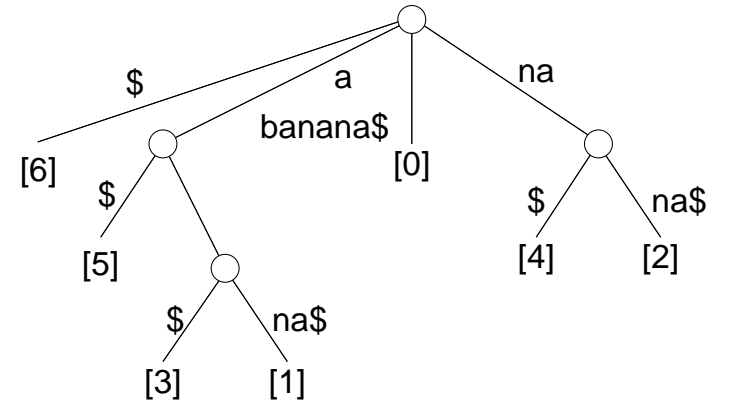
All suffixes

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

Suffix trie



Suffix tree



Suffix array

Sort the suffixes:

```
6 : $  
5 : a$  
3 : ana$  
1 : anana$  
0 : banana$  
4 : na$  
2 : nana$
```

Suffix array is (6, 5, 3, 1, 0, 4, 2)

Requires less space than a suffix tree.

Can use binary search to look for a substring (first and/or last occurrence).

Burrows-Wheeler transform (BWT)

All rotations of the initial string ('banana\$'):

```
$banana  
a$banan  
na$bana  
ana$ban  
nana$ba  
anana$b  
banana$
```

Sorted, the Burrows-Wheeler Matrix (BWM):

```
$banana  
a$banan  
ana$ban  
anana$b  
banana$  
na$bana  
nana$ba
```

The last column gives the Burrows-Wheeler transform: annb\$aa

Reversing BWT with LF mapping

First and last columns from BWM.

Character in column L is immediately before character in column F.

Start with first row, which has \$ in column F.

F	L	The character before	is
\$	a_0		
a_0	n_0		
a_1	n_1		
a_2	b_0		
b_0	\$		
n_0	a_1		
n_1	a_2		

“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.