Programming in bioinformatics: BioPerl

Dobrica Pavlinušić

http://www.rot13.org/~dpavlin/

Programming and biology
Basic algorithm structures
Programming for biology

- Cultural divide between biologists and computer science
  - use programs, don't write them
  - write programs when there's nothing to use
  - programming takes time
- Focus on interesting, unsolved, problems
- Open Source tools comes as part of the rescue
Reasons for programming

- **Scientific**
  - Quantity of existing data
  - Dealing with new data
  - Automating the automation
  - Evaluating many targets

- **Economic**

... programmers going into biology often have the harder time of it ... biology is subtle, and it can take lots of work to begin to get a handle on the variety of living organisms. **Programmers new to the field sometimes write a perfectly good program for what turns out to be the wrong problem!** -- James Tisdall
Biology

- Science in different mediums
  - in vitro – in glass
  - in vivo – in life
  - in silico – in computer algorithms
- Huge amount of experimental data
  - collected, shared, analyzed
  - biologists forced to relay on computers
Basic programming

Simple basic building blocks which enable us to describe desired behavior (algorithm) to computer

- sequence
- condition
- loop
Why perl?

- well suited to text manipulation tasks
- easy to learn
- CPAN modules, including BioPerl
- rapid prototyping
  - duct tape of Internet
- available on multiple platforms
  - Unix, Linux, Windows, VMS...
- TIMTOWTDI
  - There Is More Than One Way To Do It
int main ()
{
  register char byte, cap;
  for(;read (0, &byte, 1);)
  {
    cap = byte & 32;
    byte &= ~cap;
    byte = ((byte >= 'A') && (byte <= 'Z') ?
             ((byte - 'A' + 13) % 26 + 'A') : byte) | cap;
    write (1, &byte, 1);
  }
}

import java.io.*;
public class rot13 {
  public static void main (String args[])
  {
    int abyte = 0;
    try {
      while((abyte = System.in.read())>=0) {
        int cap = abyte & 32;
        byte &= ~cap;
        byte = ((byte >= 'A') && (byte <= 'Z') ?
                 ((byte - 'A' + 13) % 26 + 'A') : byte) | cap;
        System.out.print(String.valueOf((char)abyte));
      }
    } catch (IOException e) { }
    System.out.flush();
  }
}

#!/usr/bin/perl -p
y/A-Za-z/N-ZA-Mn-za-m/;
Art of programming

• Different approaches
  – take a class
  – read a tutorial book
  – get programming manual and plunge in
  – be tutored by a programmer
  – identify a program you need
  – try all of above until you've managed to write the program
Programming process

- identify inputs
  - data from file or user input
- make overall design
  - algorithm by which program generate output (made out of three simple parts)
- decide how to output results
  - files, graphic
- refine design by specifying details
- write perl code
### IUB/IUPAC codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Nucleic Acid(s)</th>
<th>Code</th>
<th>Amino acid</th>
<th>TLC</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Adenine</td>
<td>A</td>
<td>Alanine</td>
<td>Ala</td>
</tr>
<tr>
<td>C</td>
<td>Cytosine</td>
<td>B</td>
<td>Aspartic acid or Asparagine</td>
<td>Asx</td>
</tr>
<tr>
<td>G</td>
<td>Guanine</td>
<td>C</td>
<td>Cysteine</td>
<td>Cys</td>
</tr>
<tr>
<td>T</td>
<td>Thymine</td>
<td>D</td>
<td>Aspartic acid</td>
<td>Asp</td>
</tr>
<tr>
<td>U</td>
<td>Uracil</td>
<td>E</td>
<td>Glutamic acid</td>
<td>Glu</td>
</tr>
<tr>
<td>M</td>
<td>A or C (amino)</td>
<td>F</td>
<td>Phenylalanine</td>
<td>Phe</td>
</tr>
<tr>
<td>R</td>
<td>A or G (purine)</td>
<td>G</td>
<td>Glycine</td>
<td>Gly</td>
</tr>
<tr>
<td>W</td>
<td>A or T (weak)</td>
<td>H</td>
<td>Histidine</td>
<td>His</td>
</tr>
<tr>
<td>S</td>
<td>C or G (strong)</td>
<td>I</td>
<td>Isoleucine</td>
<td>Ile</td>
</tr>
<tr>
<td>Y</td>
<td>C or T (pyrimidine)</td>
<td>K</td>
<td>Lysine</td>
<td>Lys</td>
</tr>
<tr>
<td>K</td>
<td>G or T (keto)</td>
<td>L</td>
<td>Leucine</td>
<td>Leu</td>
</tr>
<tr>
<td>V</td>
<td>A or C or G</td>
<td>M</td>
<td>Methionine</td>
<td>Met</td>
</tr>
<tr>
<td>H</td>
<td>A or C or T</td>
<td>N</td>
<td>Asparaginie</td>
<td>Asn</td>
</tr>
<tr>
<td>D</td>
<td>A or G or T</td>
<td>P</td>
<td>Proline</td>
<td>Pro</td>
</tr>
<tr>
<td>B</td>
<td>C or G or T</td>
<td>Q</td>
<td>Glutamine</td>
<td>Gln</td>
</tr>
<tr>
<td>N</td>
<td>A or G or C or T (any)</td>
<td>R</td>
<td>Arginine</td>
<td>Arg</td>
</tr>
<tr>
<td>S</td>
<td>Serine</td>
<td>S</td>
<td>Serine</td>
<td>Ser</td>
</tr>
<tr>
<td>T</td>
<td>Threonine</td>
<td>T</td>
<td>Threonine</td>
<td>Thr</td>
</tr>
<tr>
<td>V</td>
<td>Valine</td>
<td>V</td>
<td>Valine</td>
<td>Val</td>
</tr>
<tr>
<td>W</td>
<td>Tryptophan</td>
<td>W</td>
<td>Tryptophan</td>
<td>Trp</td>
</tr>
<tr>
<td>X</td>
<td>Unknown</td>
<td>X</td>
<td>Unknown</td>
<td>Xxx</td>
</tr>
<tr>
<td>Y</td>
<td>Tyrosine</td>
<td>Y</td>
<td>Tyrosine</td>
<td>Tyr</td>
</tr>
<tr>
<td>Z</td>
<td>Glutamic acid or Glutamine</td>
<td>Z</td>
<td>Glutamic acid or Glutamine</td>
<td>Glx</td>
</tr>
</tbody>
</table>
Variables to store data

• Scalars
  – denoted by $sigil
  – store sequence of chars
  – join, substr, translate, reverse

• characters used
  – A, C, G, T – DNA nucleic acid
  – A, C, G, U – RNA
  – N – unknown

• $DNA= 'ATAGTGCCGAGTGATGTAGTA' ;
Transcribing DNA to RNA

#!/usr/bin/perl -w
# Transcribing DNA into RNA

# The DNA
$DNA = 'ACGGGAGGACGGGAAAATTACTACGGCATTAGC';

# Print the DNA onto the screen
print "Here is the starting DNA:\n\n";
print "$DNA\n\n";

# Transcribe the DNA to RNA by substituting all T's with U's.
$RNA = $DNA;

$RNA =~ s/T/U/g;

# Print the RNA onto the screen
print "Here is the result of transcribing the DNA to RNA:\n\n";
print "$RNA\n";

# Exit the program.
exit;
String substitution

Here is the starting DNA:
ACGGGAGGACGGGAAAATTACTACGGCATTAGC

Here is the result of transcribing the DNA to RNA:
ACGGGAGGACGGGAAAAUUACUACGGCAUUAGC

$RNA \sim s/T/U/g$;

replace with

scalar variable
binding operator
substitute operator
modifier (globally)
s/T/U/g

Input data
ACGGGAGGACGGGAAAATTACTACGGCATTAGC

Take 1 char
$\text{char} = 'A'$

$\text{char} \text{ eq} 'T' \quad \text{YES} \quad \rightarrow \quad \text{char} = 'U'$

NO

Output $\text{char}$
A

Output data
ACGGGAGGACGGGAAAUUACUACGGCAUUAGC
#!/usr/bin/perl -w
# Calculating the reverse complement of a strand of DNA

# The DNA
$DNA = 'ACGGGAGGACGGGAAAATTACTACGGCATTAGC';

# Print the DNA onto the screen
print "Here is the starting DNA:

";
print "$DNA\n";

# Make a new (reverse) copy of the DNA
$revcom = reverse $DNA;

print "Reverse copy of DNA:

$revcom\n";

# Translate A->T, C->G, G->C, T->A, s/// won't work!
$revcom =~ tr/ACGT/TGCA/;

# Print the reverse complement DNA onto the screen
print "Here is the reverse complement DNA:
$revcom\n";
exit;
#!/usr/bin/perl -w
# Calculating the reverse complement of a strand of DNA

# read lines from file or STDIN
while ( $DNA = <> ) {
    # remove line ending
    chomp( $DNA );

    # Make a new (reverse) copy of the DNA
    $revcom = reverse $DNA;

    # Translate A->T, C->G, G->C, T->A
    $revcom =~ tr/ACGT/TGCA/;

    # Print the reverse complement DNA onto the screen
    print "$revcom\n";
}

$ cat dna.txt
ACGGGGAGGACGGGAAAATTACTACGGCATTAGC
$ ./03-complement-file.pl dna.txt
GCTAATGCCGTAGTAATTTTCCCCTCGTCCTCCCGT
Introducing @array

- list of ordered elements
  - direct access to element by offset
    $first_element = $array[0];
  - can be created from scalars using split
    @array = split( //, 'ABCD' );
    @array = ( 'A', 'B', 'C', 'D' );
  - can be iterated, extended and consumed at both ends
    $first = shift @array;  # ('B','C','D')
    $last = pop @array;    # ('B','C')
    unshift @array, 'X';   # ('X','B','C')
    push @array, 'Y';      # ('X','B','C','Y')
How about mutations?

- perl provides random number generator
- we want to mutate 10% of nucleotides
  - length of DNA divided by 10
- store mutated DNA in array
- for each mutation
  - find $mutation_position
  - select new $random_nucleotide
  - modify @mutated_DNA
- print out @mutated_DNA as string
#!/usr/bin/perl -w
use strict;
# randomize 10% of nucleotides

my @nucleotides = ( 'A', 'C', 'G', 'T' );

while ( my $DNA = <> ) {
    chomp( $DNA );
    my $DNA_length = length( $DNA );
    warn "DNA has $DNA_length nucleotides\n";
    my $mutations = int( $DNA_length / 10 );
    warn "We will perform $mutations mutations\n";
    my @mutated_DNA = split( //, $DNA );
    for ( 1 .. $mutations ) {
        my $mutation_position = int( rand( $DNA_length ) );
        my $random_position = int( rand( $#nucleotides ) );
        my $random_nucleotide = $nucleotides[ $random_position ];
        $mutated_DNA[ $mutation_position ] = $random_nucleotide;
        warn "mutation on $mutation_position to $random_nucleotide\n";
    }
    warn "$DNA\n";
    print join('', @mutated_DNA),"\n";
}
$ ./05-random.pl dna2.txt | tee dna3.txt
DNA has 33 nucleotides
We will perform 3 mutations
mutation on 16 to A
mutation on 21 to A
mutation on 8 to A
ACGGAAGGACGGAATAATTATGCTAGGCATTAGC
ACGGAGGACGGAAAATTACAACGCTAGGCATTAGC
DNA has 33 nucleotides
We will perform 3 mutations
mutation on 9 to G
mutation on 24 to A
mutation on 12 to A
GCTAATGCTGCTAGTAATTTCCGTCCTCCGTC
GCTAATGCTGCTGTAATTTTCCCCGACCTCCGTC
Introducing %hash

- unordered list of pair elements
  - stores key => value pairs
    \[
    \text{%hash} = ( \text{foo} => 42, \text{bar} => 'baz' );
    \]
  - can fetch all key values or pairs
    \[
    \text{@all\_keys} = \text{keys %hash};
    \text{while } ((\text{$key}, \text{$value}) = \text{each %hash}) \{ \\
    \text{print "$key=$value\n";}
    \}
    \]

- Examples
  - counters
  - lookup tables (mappings)
Let's count nucleotides!

- read input file for DNA line by line
- split DNA into @nucleotides array
- for each $nucleotide increment %count
  - key will be nucleotide code
  - value will be number of nucleotides
  - we don't care about order :-)
- iterate through %count and print number of occurrences for each nucleotide
- same as counting letters in string
#!/usr/bin/perl -w
use strict;
# Count nucleotides in input file

my %count;

while ( my $DNA = <> ) {
    chomp( $DNA );
    # $DNA = “ACGGGAGGACGGGAAAATTACTACGGCATTAAC”

    my @nucleotides = split( //, $DNA );
    # (“A”,“C”,“G”,“G”,“G”,“A”,“G”,“G”,“A”,“C”,“G”,“G”,“G”,“A”,“A”...) 

    foreach my $nucleotide ( @nucleotides ) {
        $count{$nucleotide}++;  # increment by one
    }
}

# %count = ( A => 11, C => 6, G => 11, T => 5 )
while ( my ($nucleotide,$total_number) = each %count ) {
    print "$nucleotide = $total_number
";
}
Unix file handling

```bash
$ cat dna.txt
ACGGGAGGACGGGAAAATTACTACGGCATTAGC
# make new copy
$ cp dna.txt dna2.txt
# append complement of DNA from dna.txt to dna2.txt
$ ./03-complement-file.pl dna.txt >> dna2.txt
# examine current content of file dna2.txt
$ cat dna2.txt
ACGGGAGGACGGGAAAATTACTACGGCATTAGC
GCTAATGCCGTAGTAATTTTCCCGTCCTCCCGT
# count nucleotides in dna.txt
$ ./04-count.pl dna.txt
A = 11
T = 5
C = 6
G = 11
# and again in dna2.txt - do numbers look OK?
$ ./04-count.pl dna2.txt
A = 16
T = 16
C = 17
G = 17
```
my %genetic_code = (
    'TCA' =>'S', 'TCC' =>'S', 'TCG' =>'S', 'TCT' =>'S',
    'TTC' =>'F', 'TTT' =>'F', 'TTA' =>'L', 'TTG' =>'L',
    'TAC' =>'Y', 'TAT' =>'Y', 'TAA' =>'_', 'TAG' =>'_',
    'TGC' =>'C', 'TGT' =>'C', 'TGA' =>'I', 'TGG' =>'W',
    'CTA' =>'L', 'CTC' =>'L', 'CTG' =>'L', 'CTT' =>'L',
    'CCA' =>'P', 'CCC' =>'P', 'CCG' =>'P', 'CCT' =>'P',
    'CAC' =>'H', 'CAT' =>'H', 'CAA' =>'Q', 'CAG' =>'Q',
    'CGA' =>'R', 'CGC' =>'R', 'CGG' =>'R', 'CGT' =>'R',
    'ATA' =>'I', 'ATC' =>'I', 'ATT' =>'I', 'ATG' =>'M',
    'ACA' =>'T', 'ACC' =>'T', 'ACG' =>'T', 'ACT' =>'T',
    'AAC' =>'N', 'AAT' =>'N', 'AAA' =>'K', 'AAG' =>'K',
    'AGC' =>'S', 'AGT' =>'S', 'AGA' =>'R', 'AGG' =>'R',
    'GTA' =>'V', 'GTC' =>'V', 'GTG' =>'V', 'GTT' =>'V',
    'GCA' =>'A', 'GCC' =>'A', 'GGC' =>'A', 'GCT' =>'A',
    'GAC' =>'D', 'GAT' =>'D', 'GAA' =>'E', 'GAG' =>'E',
    'GGA' =>'G', 'GGC' =>'G', 'GGG' =>'G', 'GGT' =>'G',
);
# define subroutine (in separate file together with %genetic_code)
# and store it in module GeneticCode.pm to be reusable

sub codon2aa {
    my ( $codon ) = @_;

    # check does mapping for codon exists
    if ( exists $genetic_code{ $codon } ) {
        # if it does, return amino acid
        return $genetic_code{ $codon };
    } else {
        # if it doesn't exist with error
        die "bad codon: $codon";
    }
}

# now we can use module directly from command line;
perl -MGeneticCode -e "print codon2aa('ACG')"
#!/usr/bin/perl -w
use strict;

# load module (*.pm)
use GeneticCode;

while ( my $DNA = <> ) {
    chomp($DNA);

    my $protein = '';  

    # start at beginning and move by three places through DNA
    for ( my $i = 0; $i <= (length($DNA) - 2); $i += 3 ) {
        # extract single codon starting at position $i
        my $codon = substr( $DNA, $i, 3 );

        # call subroutine from GeneticCode module
        $protein .= codon2aa( $codon );
    }

    print "$protein\n";
}
Decoding DNA proteins

$ cat dna2.txt dna3.txt
ACGGGAGGACGGGAAAATTACTACGGCATTAGC
GCTAATGCCGTAGTAATTTTTCCGTCCTCCCGT
ACGGGAGGACGGGAAAATTACCAACGGCATTAGC
GCTAATGCCGTAGTAATTTTTCCGACCTCCCGT
$ ./06-dna2protein.pl dna2.txt dna3.txt
TGGRENYYGIS
ANAVVIFPSSR
TGGRENYNGIS
ANAVIIIFPTSR
# let's improve our GeneticCode.pm by extending it to DNA2protein.pm

sub DNA2protein {
    my ( $DNA, $offset ) = @_;  
    my $protein = '';  

    # start at $offset and move by three places through DNA  
    for ( my $i=$offset; $i<=(length($DNA)-2-$offset); $i+=3 ) {
        # extract single codon starting at position $i  
        my $codon = substr( $DNA, $i, 3 );  
        # decode codon to amino acid  
        $protein .= codon2aa( $codon );
    }
    # return created protein  
    return $protein;
}

sub revcom {
    my ( $DNA ) = @_;  
    my $revcom = reverse $DNA;  
    $revcom =~ tr/ACGT/TGCA/;  
    return $revcom;
}
#!/usr/bin/perl -w
use strict;

# use module DNA2protein to implement reading frames
use DNA2protein;

while ( my $DNA = <> ) {
    chomp($DNA);
    foreach my $offset ( 0 .. 2 ) {
        print DNA2protein( $DNA, $offset ), "\n";
        print DNA2protein( revcom($DNA), $offset ), "\n";
    }
}

$ ./07-reading-frames.pl dna.txt
TGGRENNYYGIS
ANAVVIFPSSR
REDKITTAL
LMP__FSRPP
GRTGKLLRH___CRSNFPVLP
Review

- Why to pursue biology programming?
- Algorithmic way of thinking
- $Scalars, @arrays and %hashes
- Modules as reusable components made of subroutines
- Combination of small tools with pipes \( (\text{the Unix way}) \)
Find out more...

- James Tisdall: "Beginning Perl for Bioinformatics", O'Reilly, 2001
Questions?

3*7*2