

Bio 595

Computers in Biomedical Research Fall 2003 Slides for Monday, December 1

Ex19 rand function

Built-in function 'scalar' returns the number of elements in an array.

The code in a key line is `$verbs[int(rand(scalar @verbs))]`

So, the line equivalent is `$verbs[int(rand(7))]`

The 'rand(7)' function will return a value greater than 0 and less than 7.

The 'int' function discards the fractional part and returns the integer, perhaps a 3 here, so `$verbs[3]` returns the fourth element in the array (numbered from the first element subscripted zero)

The code could also use an attribute of Perl in which functions are 'chained' without the parentheses:

```
$verbs[int rand scalar @verbs]
```

And `$verbs[rand @verbs]` does the same because subscripts are automatically converted to integers

Simulating DNA mutation

Algorithm design:

Select a random position in a DNA sequence

Choose a nucleotide at random

Substitute into random position in the DNA strand.

Select a position in the string at random:

#randomposition: a subroutine to select a position

```
Sub randomposition {
```

```
  my($string) = @_;
```

```
  # return a position between 0 and length-1
```

```
  return int(rand(length($string)));
```

```
}
```

test this subroutine (ex20)

```
#!/usr/bin/perl -w
# Test the subroutine 'randomposition'
my $dna = 'ACGTCGATCGCCGATCGGACATGTTTA';
# now seed the random number generator
srand(time|$$);
for (my $I=0; $I<20; ++$I) {
  print randomposition($dna), " ";
}
print "\n";
exit;

sub randomposition {
  my ($string)=@_;
  return int rand length $string;
}
```

choose a random nucleotide

```
#!/usr/bin/perl -w
# ex21: test the randomnucleotide subroutine
my @nucleotides = ('A', 'C', 'G', 'T');
srand(time|$$);
for (my $I=0; $I < 20; ++$I) {
  print randomnucleotide(@nucleotides), " ";
}
print "\n";
exit;
# here's the subroutine to pick a random nucleotide
sub randomnucleotide {
  my(@nucs)=@_;
  #scalar returns size of the array
  #array elements are numbered 0 to size-1
  return $nucs[rand @nucs];
}
```

Subroutine to place the random nucleotide

```
# a subroutine to perform a mutation
sub mutate {
  my($dna)=@_;
  my(@nucleotides) = ('A', 'C', 'G', 'T');
  # pick a random position in the DNA
  my($position)=randomposition($dna);
  # pick a random nucleotide
  my($newbase)=randomnucleotide(@nucleotides);
  # now insert the nucleotide into the DNA
  # in $dna at $position change 1 character to $newbase
  Substr($dna,$position,1,$newbase);
  Return $dna
}
```

Alternate design to 'randomnucleotide'

Instead of passing @nucleotides array to randomnucleotide subroutine, place it (declare it) within that subroutine:

```
# randomnucleotide
sub randomnucleotide {
    my(@nucs) = ('A', 'C', 'G', 'T');
    return $nucs[rand @nucs];
}
```

Now the function has no arguments to pass. It's called like this:

```
$randomnucleotide = randomnucleotide();
So, combine these subroutines in a final program (ex22)
```

Generating random DNA

This is actually a useful undertaking: BLAST, for example, depends on random DNA properties for doing sequence similarity scoring and statistics.

Write a program to create a specified number of segments, with minimum and maximum lengths specified.

General design: create a subroutine and pass the necessary arguments: First, imagine the subroutine, then figure out the code (and subroutines you can recycle).

```
@random_DNA= make_random_DNA_set
($minimum_length, $maximum_length, $size_of_set);
Examine ex23 and its extensive comments...
```

Analyzing DNA (ex24)

Finally, here's a program that collects some statistics. The question posed, is: what percentage bases are the same between two random DNA sequences?

We could easily calculate this without writing a program, but the resultant program could also be applied to DNA sequences to be compared.

In this example, generate random DNA chains of the same length; then compare all pairs of DNA and for each pair, calculate the % positions having the same nucleotides. Then calculate the average of all percentages.

Comments about ex24

A main call in the program:

```
@random_DNA = make_random_DNA_set(10,10,10)
```

Note that it wasn't necessary to initialize variables for minimum and maximum length; just pass the desired numeric values

Another main call in the program uses a "nested loop":

```
# run through all pairs of sequences
for (my $k=0; $k < scalar @random_DNA -1; ++$k) {
    for (my $l = $k +1; $l < scalar @random_DNA; ++$l) {
        # calculate and save percentages
        $percent = matching_percentage($random_DNA[$k],
            $random_DNA[$l]);
        Push (@percentages, $percent);
    }
}
```

More comments about ex24

The nested loop here selects all combinations of two (or more) elements in a set.

The nested loop looks at $n * (n-1) / 2$ pairs of sequences. Looking at all possible pairs of sequences can get out of hand in a hurry, since the number of pairs you need to compare rises with the square of the number 'n'.

Note that at the start, sequence 0 (indexed by \$k) is paired with sequences 1, 2, 3...9 (indexed by \$l). At the end, sequence 8 is paired with 9. Note that the 'scalar @random_DNA' gives the number of elements in the array.

Hashes

Perl has three major types of data, scalars, arrays and hashes. A hash allows data to be found quickly using a key. For example:

Suppose we have a hash called %periodic_table. Note that the hash is identified by the prefix '%'. Then

```
$element = $periodic_table{"boron"};
```

The scalar 'boron' is the key, and the scalar definition that is returned is the value. The hash changes its leading character to a dollar sign when a single element is accessed – the value returned from a hash lookup is a scalar value. Note hashes use curly brackets [whereas arrays use square brackets].

Assign a value to a key: \$element('boron')= "Symbol = B, Atomic number = 5, atomic weight = 11";