Introduction to Perl: Eighth Lecture

Duncan C. White (d.white@imperial.ac.uk)

Dept of Computing,
Imperial College London

January 2014

The main topic for today is **data structures on demand**, by means of **program transformations** that guarantee to preserve correctness:

- In languages such as Haskell, data structures are very easy to use (lists and tuples built-in) and define (recursive data types).
- In languages like C, building data structures seems hard (which is why you should build a toolkit), so you tend to only build data structures for the macro-scale.
- In Perl, data structures are even easier to use than in Haskell - so simple that building optimal data structures - and changing them when you change your mind - becomes a useful programming technique on scales right down to a single function.
- I call this the **Agile Data Structures** approach.

We'll also talk briefly about testing, benchmarking and profiling, and then wrap up the course.

- While writing a single function, you often write code that *computes a single answer*. Sometimes you can *transform* this code by building a data structure enabling you to *lookup all answers of that kind*.
- For example, given an unsorted array of numbers:

```
@array = ( 17, 5, 3, 17, 2, 5, 7, 6, 6, 10, 3 );
```

- Consider finding unique values from such an array. We might write the following naive code (**eg1**):

```
# build @uniq, an array of all unique elements of @array
my @uniq;
foreach my $i (0..$#array)              # foreach index i in @array
{
        # count how many elements array[j] (i!=j) are the same as array[i]
        my $count = 0;
        foreach my $j (0..$#array)
        {
                $count++ if $i != $j && $array[$i] == $array[$j];
        }
        # unique if $count == 0
        push @uniq, $array[$i] if $count == 0;
}
```

- This is very C-style code! index based, unclear, 13 lines long, could harbour bugs. Worse still, it's $O(N^2)$.

- Our first transformation is to notice that we can eliminate the `$i != $j` test, and compare the count with one not zero (**eg2**):

```
# build @uniq, an array of all unique elements of @array
my @uniq;
foreach my $i (0..$#array)              # foreach index i in @array
{
        # how many elements array[j] are the same as array[i] (inclusive)
        my $count = 0;
        foreach my $j (0..$#array)
        {
                $count++ if $array[$i] == $array[$j];
        }
        # unique if $count == 1 (array[i] itself)
        push @uniq, $array[$i] if $count == 1;
}
```

- Next, notice that we no longer use indices *i* and *j* separately from *array*[*i*] and *array*[*j*], so we can now loop over the values (**eg3**):

```
# build @uniq, an array of all unique elements of @array
my @uniq;
foreach my $x (@array)
{
        # how many elements y are the same as x (including x)?
        my $count = 0;
        foreach my $y (@array)
        {
                $count++ if $x == $y;
        }
        # unique if $count == 1 (x itself)
        push @uniq, $x if $count == 1;
}
```

- Our next transformation is to notice that the inner loop can be replaced with a call to *grep* (**eg4**). Recall that *grep* constructs a list, and assigning that list to a scalar `$count` delivers the number of elements in the list:

```
# build @uniq, an array of all unique elements of @array
my @uniq;
foreach my $x (@array)
{
        # how many elements are the same as x (including x)?
        my $count = grep { $_ == $x } @array;

        # unique if $count == 1 (x itself)
        push @uniq, $x if $count == 1;
}
```

- All the above transformations have improved the clarity of the code, we're much more confident that this is correct now. However, still $O(N^2)$ - because *grep* (and *map*) count as $O(N)$.
- But now we make a simple observation: Over the course of the foreach loop, we calculate the frequency of *every array element.*
- So why not pre-calculate the element frequencies ahead of time? This suggests a new data structure:

```
my %freq;                  # array element -> frequency of that element
```

- To populate `%freq` we write:

```
my %freq;
foreach my $x (@array)
{
  $freq{$x}++;
}
```

- Once we have `%freq` our code is:

```
my @uniq;
foreach my $x (@array)
{
  push @uniq, $x if $freq{$x} == 1;
}
```

- Bringing this all together, this gives **eg5**, which is clearly $O(N)$!
- Next, the `%freq` building code may be more idiomatically written:

```
my %freq; map { $freq{$_}++ } @array;      # build array element -> frequency of that element
```

  (or `$freq{$_}++ for @array` which many prefer.)
- Finally, we notice that the main loop is another grep:

```
my @uniq = grep { $freq{$_} == 1 } @array; # build @uniq, all unique elements of @array
```

- These two lines are the heart of our final clear simple $O(N)$ version **eg6**. Compare this to our original 13 line $O(N^2)$ **eg1**!
- Of course, we had to allocate a modest extra amount of space for the frequency hash. But it's definitely worth it!

- Now, suppose we actually wanted an array of the *distinct non-unique* values instead. Non-unique values (ignoring *distinct*) are easy, simply change `freq == 1` to `freq > 1`:

```
my %freq; map { $freq{$_}++ } @array;
my @nonuniq = grep { $freq{$_} > 1 } @array;
```

- However, this includes each non-unique element many times.
- For example, if `@array = (1,1,1,2,2)` then `@nonuniq = (1,1,1,2,2)` whereas *distinct* suggests that we wanted `@nonuniq = (1,2)`.
- To remove duplicates from `@nonuniq`, we can use a standard *turn it into a set and extract the keys* idiom:

```
my %set = map { $_ => 1 } @nonuniq;
@nonuniq = keys %set;
```

- Recall that `keys %set` delivers the keys in an unpredictable order. We could say `sort keys %set`, but our code would become $O(NlogN)$.
- An $O(N)$ alternative - that delivers the distinct values in the order they were present in the original array - is to replace the *set of all items in the array* with a *set of all items seen so far* (**eg7**):

```
my %freq; map { $freq{$_}++ } @array;    # build element -> frequency hash
my %seen;                                # what elements have we already seen?
my @nonuniq =                            # build distinct non-unique elements
        grep { $freq{$_} > 1 && ! $seen{$_}++ } @array;
```

- Finally, after building and using `%freq`, suppose we realised that other parts of the program need to locate all the *positions* in the original array `@array` at which a specific value appeared.
- We need a different temporary data structure:

```
my %indexlist;       # array element -> list of positions in original array
```

- Recall that the array contains:

```
@array = ( 17, 5, 3, 17, 2, 5, 7, 6, 6, 10, 3 );
```

  Our desired `%indexlist` comprises:

```
17 => [0, 3],            2  => [4],
6  => [7, 8],            7  => [6],
5  => [1, 5],            3  => [2, 10],
```

- To build `%indexlist` we might write naive code (**eg8**):

```
# initialize all 'inner' array refs to [], maybe several times each
foreach my $value (@array)
{
  $indexlist{$value} = [];
}
# can now freely push positions onto @{$indexlist{$value}}
foreach my $index (0..$#array)
{
  my $value = $array[$index];
  my $aref  = $indexlist{$value};
  push @$aref, $index;
}
```

- In fact, the first loop is not needed because Perl **auto-vivifies** array and hash references when needed, as this snippet shows:

```
my $ref = undef;
@$ref = (1,2,3);
print "@$ref\n"'
```

- So that gives us:

```
# push positions onto @{$indexlist{$value}} freely
foreach my $index (0..$#array)
{
  my $value = $array[$index];
  my $aref  = $indexlist{$value};
  push @$aref, $index;
}
```

- `$value` is only used once, fold it in:

```
foreach my $index (0..$#array)
{
  my $aref = $indexlist{$array[$index]};
  push @$aref, $index;
}
```

- Writing the `foreach` loop as a procedural `map`, we end up with the following more idiomatic version:

```
my %indexlist;
map { my $aref = $indexlist{$array[$_]}; push @$aref, $_ } 0..$#array;
```

- If you're happy to push it one stage further, fold `$aref` in too:

```
my %indexlist;
map { push @{$indexlist{$array[$_]}}, $_ } 0..$#array;
```

- Now, given that `$freq{$v} == @{$indexlist{$v}}`, ie. `$v`'s frequency is the length of `$v`'s position list, do we need to keep `%freq`?
- A minimalist would remove `%freq`, to avoid redundancy. Our uniqueness detector would then be:

```
my @uniq = grep { @{$indexlist{$_}} == 1 } @array;
```

- Personally, I'd keep both - and build them together (**eg9**):

```
my( %indexlist, %freq );
map { $freq{$array[$_]}++; push @{$indexlist{$array[$_]}}, $_; } 0..$#array;
```

- Let's pause for a moment and take stock of what we've done:
  - In a series of very small example programs (each $<$ 20 lines long)..
  - We've shown how to **gradually transform** low level algorithmic code, into **shorter, clearer, more obviously correct** code...
  - Using **temporary data structures** (*scaffolding*) and **higher-order functions** such as `grep` and `map`...
  - To make the **original problem much easier to solve**..
  - Sometimes even making the code **faster** and **more efficient**.
- This is a sufficiently rare combination of good characteristics that it's worth celebrating, noting that it's only possible because Perl makes building optimal data structures so simple.

- Please note that this technique isn't only appropriate on the small scale - let's scale it up. We said that we were working inside functions, let's make that explicit now:

```
#
# @uniq = unique_values( @array ):
#       Deliver all non-repeated values from @array
#       in the SAME ORDER they were present in @array
#
fun unique_values( @array )
{
        my %freq; map { $freq{$_}++ } @array;      # array element -> frequency
        my @uniq = grep { $freq{$_} == 1 } @array; # @uniq, unique elements
        return @uniq;
}

#
# @nonuniq = distinct_nonunique_values( @array ):
#       Deliver all repeated (non-unique) values from @array
#       once each (i.e. distinct), in the SAME ORDER as they
#       were first found in @array
#
fun distinct_nonunique_values( @array )
{
        my %freq; map { $freq{$_}++ } @array;    # array element -> frequency
        my %seen;                                # elements we've already seen
        my @nonuniq = grep                       # distinct non-unique elements
                { $freq{$_} > 1 && ! $seen{$_}++ } @array;
        return @nonuniq;
}
```

- Plus a bonus function (and a test case, giving **eg10**):

```
#
# @distinct = distinct_values( @array ):
#       Deliver all distinct values from @array,
#       in the SAME ORDER as first found in @array.
#
fun distinct_values( @array )
{
        my %seen;                                    # elements already seen
        my @distinct = grep { ! $seen{$_}++ } @array;# distinct elements
        return @distinct;
}
```

- In reality, there'd be many more such functions, some building and using `%indexlist` instead of, or as well as, `%freq`.
- Although there's nothing wrong with building `%freq` and friends independently each time we need them, we might wonder whether we should break such code out:

```
#
# %freq = build_freq_hash( @array ):
#       Build a frequency hash of the elements of @array, i.e. a hash
#       mapping each element (key) to the frequency of that element in @array,
#
fun build_freq_hash( @array )
{
        my %freq; map { $freq{$_}++ } @array;    # array element -> frequency
        return %freq;
}
```

- Now replace that code fragment in other functions with calls:

```
my %freq = build_freq_hash( @array );
```

- Having `build_freq_hash()` available as a separate function opens up the possibility of **prolonging the lifetime** of `%freq`. Perhaps someone will call both `unique_values()` and `distinct_nonunique_values()` with the same array, so why calculate `%freq` twice?
- Perhaps the caller should do the following:

```
my %freq = build_freq_hash( @array );
my @uniq = unique_values( \%freq, \@array );
my @nonuniq = distinct_nonuniq_values( \%freq, \@array );
```

- Or, if the order of elements is unimportant, just pass in `%freq`:

```
my %freq = build_freq_hash( @array );
my @uniq = unique_values( %freq );
my @nonuniq = distinct_nonuniq_values( %freq );
```

- In the latter case, as well as `build_freq_hash()` above, we'd have:

```
#
# @uniq = unique_values( %freq ):
#       Deliver all non-repeated values from a %freq hash
#       in an undetermined order
#
fun unique_values( %freq )
{
        my @uniq = grep { $freq{$_} == 1 } keys %freq;
        return @uniq;
}
```

- Plus the remaining functions, rewritten to take `%freq`:

```
#
# @nonuniq = distinct_nonunique_values( %freq ):
#       Deliver all repeated (non-unique) values from %freq
#       in an undetermined order
#
fun distinct_nonunique_values( %freq )
{
        my %seen;                               # elements we've already seen
        my @nonuniq = grep                      # distinct non-unique elements
                { $freq{$_} > 1 && ! $seen{$_}++ } keys %freq;
        return @nonuniq;
}

#
# @distinct = distinct_values( %freq ):
#       Deliver all distinct values from %freq
#       in an undetermined order
#
fun distinct_values( %freq )
{
        return keys %freq;
}
```

- Adding a test case gives us **eg11**.
- Note the much simpler `distinct_values()` implementation now that we don't care about the order - also note how we changed the comments for each function to say "in an undetermined order".

- Perl has several unit testing modules, the simplest is called `Test::Simple`, but we'll take a quick look at it's big brother `Test::More`.
- First of all, the basic concept of testing is that you already know what the correct (expected) answer is!
- `Test::More` has many test functions, we only need three:
    - `plan tests => N`: How many tests are there in total?
    - `use_ok('module_name')`: Can the given module be successfully loaded?
    - `is($got, $expected, $testdescription)`: Tests that the string `$got` (usually generated from a function you wish to test), is the same as the expected string `$expected`, printing out the given test description.
- What shall we test? How about our frequency/unique/distinct values functions, turned into a module `frequtils`.
- A minimum test might first check that we can load the module:

```
use Test::More;

plan tests => 2;                        # how many tests?
use_ok( 'frequtils' );                  # first test.. load module?
```

- Followed by:

```
#
# my $str = format_hash( %hash ):
#       Format a given hash into a string in a predictable
#       order and format.  we've chosen comma separated
#       key:value pairs, sorted by key
#
fun format_hash( %hash )
{
        my @k = sort keys %hash;
        return join( ",", map { "$_:$hash{$_}" } @k );
}

my @array = (1,2,1,3);
my $input = "1,2,1,3";
my $expected = "1:2,2:1,3:1";

my %freq = build_freq_hash( @array );
my $output = format_hash( %freq );

is( $output, $expected,                 # second test.. right result?
        "build_freq_hash($input)=$output" );
```

- This forms **eg12**. Running it, we get output:

```
1..2
ok 1 - use frequtils;
ok 2 - build_freq_hash(1,2,1,3)=1:2,2:1,3:1
```

- Let's check that the test framework is working, by adding `$output .= ",6:1"` just before the is..

- As expected, now we get something scarier:

```
1..2
ok 1 - use frequtils;
not ok 2 - build_freq_hash(1,2,1,3)=1:2,2:1,3:1,6,1
#   Failed test 'build_freq_hash(1,2,1,3)=1:2,2:1,3:1,6,1'
#   at ./eg12 line 36.
#          got: '1:2,2:1,3:1,6,1'
#     expected: '1:2,2:1,3:1'
# Looks like you failed 1 test of 2.
```

- Scaling this up to more tests of `build_freq_hash()`, we need to generalise how tests are represented:

```
my @freqtests = (               # formatted strings ("input output" pairs)
        "1        1:1",
        "2        2:1",
        "1,2      1:1,2:1",
        "1,2,1    1:2,2:1",
        "1,2,1,2  1:2,2:2",
        "1,2,1,3  1:2,2:1,3:1",
);

plan tests => 1 + @freqtests;   # how many tests?
use_ok( 'frequtils' );          # first test.. load module?
```

- Need to write new code to parse the strings, split the CSV input array apart, call `build_freq_hash()`, and check the results as before:

- This is simply (**eg13**):

```
foreach my $teststr (@freqtests)
{
        my( $input, $expected ) = split( /\s+/, $teststr, 2 );
        my @array = split(/,/, $input);
        my %freq = build_freq_hash( @array );
        my $output = format_hash( %freq );
        is( $output, $expected, "build_freq_hash($input)=$output" );
}
```

- Running it, we get output:

```
1..7
ok 1 - use frequtils;
ok 2 - build_freq_hash(1)=1:1
ok 3 - build_freq_hash(2)=2:1
ok 4 - build_freq_hash(1,2)=1:1,2:1
ok 5 - build_freq_hash(1,2,1)=1:2,2:1
ok 6 - build_freq_hash(1,2,1,2)=1:2,2:2
ok 7 - build_freq_hash(1,2,1,3)=1:2,2:1,3:1
```

- Suppose we wish to generalise further: allow each test to specify which function to test, via a 3rd field:

```
my @tests = (                   # formatted strings ("type input output")
        "freq 1        1:1",    # build_freq_hash() tests
        "freq 1,2,1,3  1:2,2:1,3:1",
        "dist 1,2,1,3  1,2,3",  # distinct_values() tests
        "uniq 1,2,1,3  2,3",    # unique_values() tests
        "dnu 1        _",       # distinct_nonunique_values() tests
        "dnu 1,2,1,2  1,2",
);
```

- Next, we extend the parser to extract the 3rd field, and support a special syntax '_' for when the output is blank:

```
foreach my $teststr (@tests)
{
        my( $type, $input, $expected ) = split( /\s+/, $teststr, 3 );
        $expected = '' if $expected eq "_";
        my @array = split(/,/, $input);
        # to be continued
}
```

- Now, we must choose what action to take based on `$type`. Let's use coderefs and data-driven programming:

```
my %testtype = (  # type -> [coderef, funcname]
        'freq' => [ \&wrap_freq,     'build_freq_hash' ],
        'uniq' => [ \&wrap_uniq,     'unique_values' ],
        'dnu'  => [ \&wrap_nonuniq,  'distinct_nonunique_values' ],
        'dist' => [ \&wrap_distinct, 'distinct_values' ],
);
```

- To use this data structure, we carry on in the `foreach my $teststr (@tests)` body (from `# to be continued`):

```
        ...
        # to be continued
        my( $testfunc, $funcname ) = @{$testtype{$type}};
        my $output   = $testfunc->(@array);
        is( $output, $expected, "$funcname($input)=$output" );
}
```

- This only leaves the definitions of the four wrap functions. Here's `wrap_freq()`:

```
#
# $str = wrap_freq( @array ):
#       call build_freq_hash( @array ) and then build
#       and return a predictable (sorted) representation
#       of the result to compare against, as a string
#
fun wrap_freq( @array )
{
        my %freq = build_freq_hash(@array);
        return format_hash( %freq );
}
```

- The other 3 are left for you to find in the example tarball.
- This is **eg14** - run it, we get output:

```
1..25
ok 1 - use frequtils;
...
ok 4 - build_freq_hash(1,2)=1:1,2:1
...
ok 13 - distinct_values(1,2,1,3)=1,2,3
...
ok 19 - unique_values(1,2,1,3)=2,3
...
ok 23 - distinct_nonunique_values(1,2,1)=1
```

- Perl has a module called `Benchmark`, with a partially OO interface and a procedural interface.
- A `Benchmark->new` object returns the current time, use it as (**eg15**):

```
use Benchmark;
my $t0 = Benchmark->new;                              # start

# ... put your code here ...
my $x = 100; for(my $i=0; $i<100000000; $i++ ) { $x++; }

my $t1 = Benchmark->new;                              # stop
my $ts = timestr( timediff($t1, $t0) );
print "the code took: $ts\n";
```

- Given several alternative algorithms whose efficiency you want to compare, use the procedural interface (**eg16**) to run and report:

```
use Benchmark qw(:all);
my $duration = shift @ARGV || 4;

timethese( -$duration,          # run for at least duration CPU seconds
{
        'x++'  => sub { my $x = 100; $x++ },
        'x+=1' => sub { my $x = 100; $x += 1 },
});
```

- There's another example (**eg17**) using a different benchmark function, `$benchmark_object = countit( $time, $coderef )`, to do more flexible benchmarking. Left for you to investigate.

- Perl has several profiling modules, most obviously one called `Devel::DProf`. Run your program (**eg17** let's say) with:

```
perl -d:DProf eg17
```

- Your program will run a bit slower than usual, then when it finishes, you'll find the `tmon.out` file, containing the profiling data.
- Now run the dprof post-processor, `dprofpp tmon.out`. This will produce a table of where time was spent:

```
Total Elapsed Time = 7.974714 Seconds
  User+System Time = 7.864714 Seconds
Exclusive Times
%Time ExclSec CumulS #Calls   sec/call Csec/c  Name
 48.9   3.853  3.853    74888   0.0001 0.0001  main::On2_uniq
 48.5   3.817  3.817   465357   0.0000 0.0000  main::On_uniq
 7.03   0.553  0.600   266829   0.0000 0.0000  Benchmark::new
 3.38   0.266  8.635      140   0.0019 0.0617  Benchmark::runloop
 1.27   0.100  7.769   540385   0.0000 0.0000  Benchmark::__ANON__
 0.60   0.047  0.047   266829   0.0000 0.0000  Benchmark::mytime
...
```

- Note that you wouldn't normally profile a Benchmark run..
- Once you know the hotspots, you can consider selectively optimizing them. As in any language, repeated profiling and optimization passes can give dramatic speedups.

- Perl features such as:
  - *typeglobs* - manipulating symbol tables.
  - *Autoloading* - defining a subroutine `AUTOLOAD` which handles missing subroutines!
  - *Compile time* vs *run time* distinctions, `BEGIN` and `END` blocks.
  - Writing Perl code on the fly via `eval`.
  - Perl one-liners.
- Using the *Perl debugger* (**perldoc perldebug** and **perldoc perldebtut**).
- *Perl and graphics* - building GUIs using `Tk` or `Gtk`, visualizing directed graphs via `GraphViz` and it's friends, constructing image files via `GD` (useful for CGI programs generating dynamic images).
- *Parser generators* using Perl - especially the awesome yacc-like module `Parse::RecDescent`.
- *Perl threads* - semaphores, thread queues etc.
- *Interfacing external C libraries into Perl* via `XS` or `Inline::C`, embedding a Perl interpreter in other programs, eg. Apache and mod_perl. Plus lots lots more.... Perl 6, Parrot..

- Checkout the Extra Notes document on my website, contains material that didn't fit in the main lectures. New this year: a slide on Moose, an alternative OO system for Perl.
- O'Reilly's site `http://www.perl.com/` (*The Perl Resource*) is a wonderful source of Perl information, containing links to a multitude of Perl information.
- Our old friend **CPAN**, found at `http://www.cpan.org/`.
- The wonderful *Perl Journal* at `http://tpj.com/` which started out as a quarterly paper journal and recently changed to a monthly e-zine in PDF format, still on subscription.
- *The Perl Directory* at `http://www.perl.org/` is a directory of links to other Perl information and news.
- *The Perl Monks* at `http://www.perlmonks.org/` is a forum-based discussion site for all matters Perlish.
- That's all folks! Enjoy your Perl programming - and remember the Perl motto: *There's More Than One Way To Do It!*
- And they're all really good fun!