Not common enough optimisations

Jacinta Richardson
Perl Training Australia
<jarich@perltraining.com.au>

Where do we start?

```
sub strings {
   my ($start, $end) = @_;
   my $full_string = '';
   foreach my $str ($start .. $end) {
      my $temp = $full_string . $str;
      $full_string = $temp;
   }
   return $full_string;
}

sub numbers {
   my ($start, $end) = @_;
   my $sum = 0;
   foreach my $num ($start .. $end) {
      my $temp = $sum + $num;
      $sum = $temp;
   }
   return $sum;
}
```

```
# perl -d:DProf some_program.pl
# dprofpp

Total Elapsed Time = 1.165982 Seconds
  User+System Time = 1.175982 Seconds
Exclusive Times
%Time ExclSec CumulS #Calls sec/call Csec/c  Name
91.6   1.078  1.078   1000   0.0011 0.0011  main::numbers
9.35   0.110  0.110      1   0.1100 0.1100  main::strings
0.00       - -0.000      1        -      -  strict::import
0.00       - -0.000      1        -      -  strict::bits
0.00       - -0.000      1        -      -  main::BEGIN
```

Remove strings() and save **9.35%**

Improve numbers() by 25% and save

22.9%!

```
sub numbers {
   my ($start, $end) = @_;
   my $sum = 0;
   foreach my $num ($start .. $end) {
      my $temp = $sum + $num;
      $sum = $temp;
      $sum += $num;
   }
   return $sum;
}
```

```
Total Elapsed Time = 0.652972 Seconds
  User+System Time = 0.652972 Seconds
Exclusive Times
%Time ExclSec CumulS #Calls sec/call Csec/c  Name
 80.7   0.527  0.527   1000   0.0005 0.0005  main::numbers
 16.8   0.110  0.110      1   0.1100 0.1100  main::strings
 0.00       - -0.000      1        -      -  strict::bits
 0.00       - -0.000      1        -      -  strict::import
 0.00       - -0.000      1        -      -  main::BEGIN
```

0.527 vs 1.078 seconds total!


Monsoon
SALE
SALE

```
use Benchmark qw(cmpthese);

cmpthese(0, {
        slow => sub {
                my ($start, $end) = (2000, 4000);
                my $sum = 0;
                foreach my $num ($start .. $end) {
                        my $temp = $sum + $num;
                        $sum = $temp;
                }
                return $sum;
        },
        fast => sub {
                my ($start, $end) = (2000, 4000);
                my $sum = 0;
                foreach my $num ($start .. $end) {
                        $sum += $num;
                }
                return $sum;
        },
});
```

```
# perl benchmark.pl
        Rate slow fast
slow   908/s   -- -51%
fast 1844/s 103%   --
```

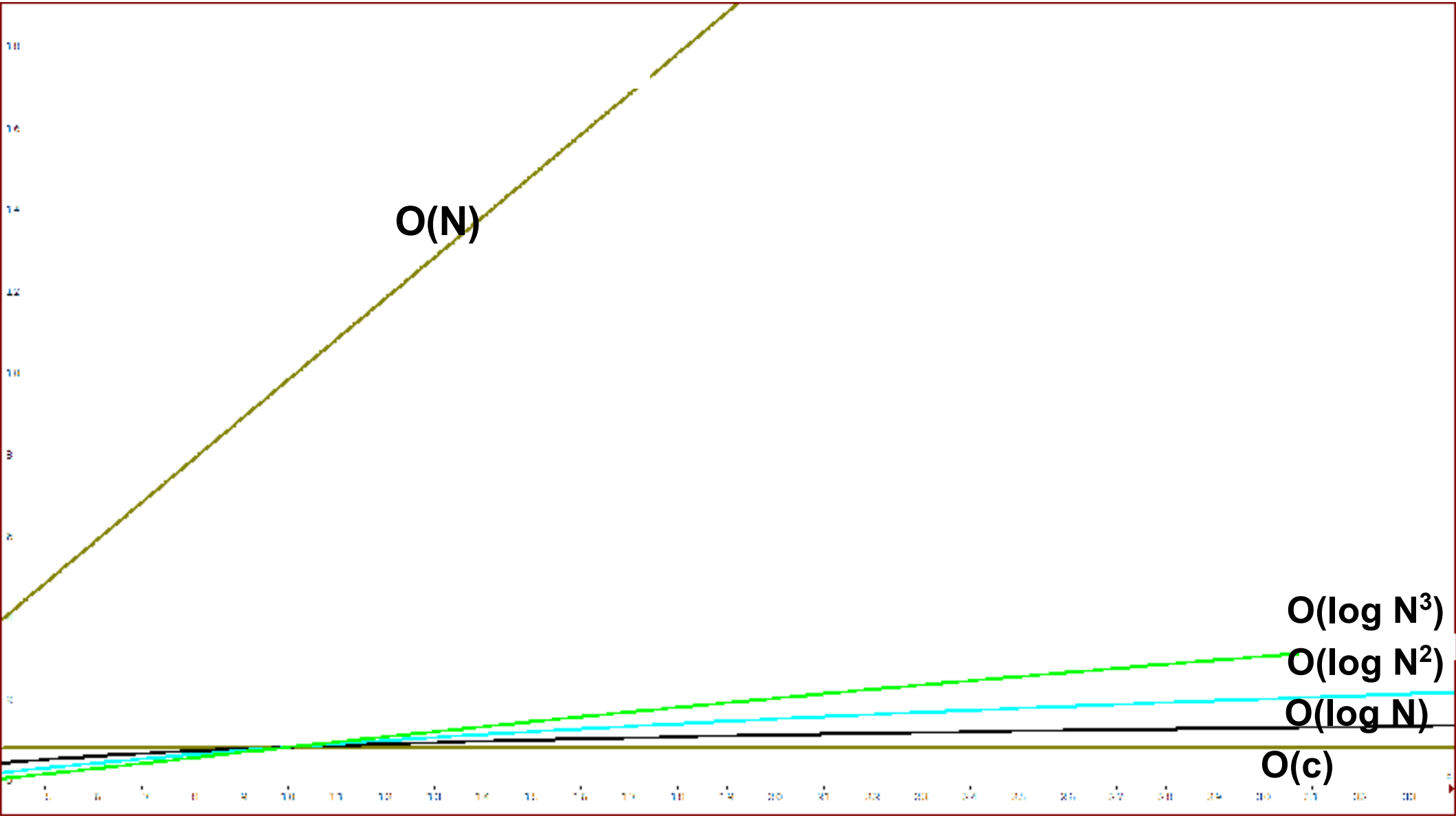

FIAT

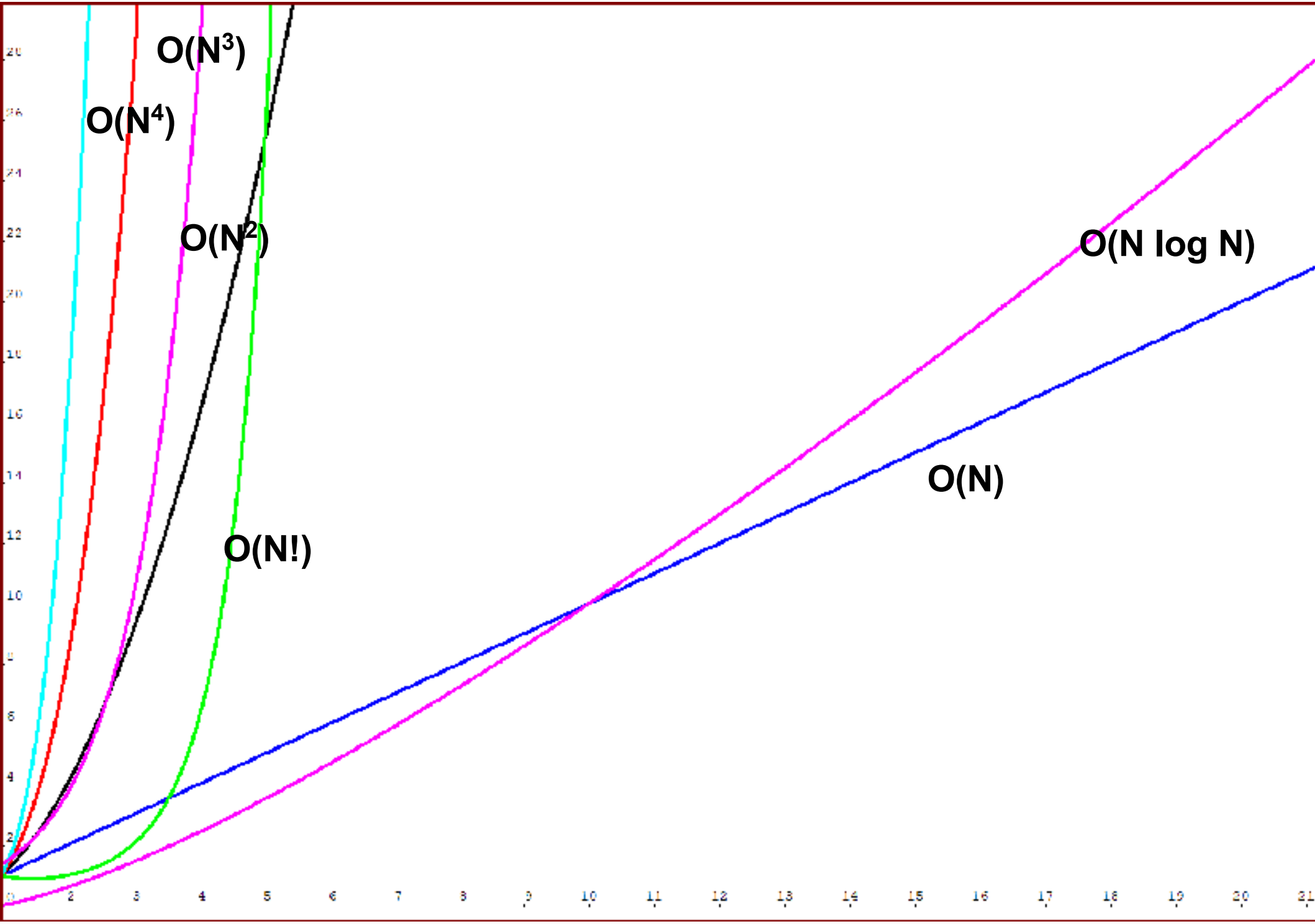
Big-O notation

$O(f(n)) \equiv$ order of $f(n)$

- $O(c)$ Constant
- $O(\log n)$ Logarithmic
- $O((\log n)^c)$ Poly-logarithmic
- $O(n)$ Linear
- $O(n \log n)$ Quasi-linear
- $O(n^c)$ $c>1$ Polynomial
- $O(c^n)$ Exponential
- $O(n!)$ Factorial

N	log n	n	n^2	n^3	n!
1	0.0	1	1	1	1
2	0.3	2	4	8	2
3	0.5	3	9	27	6
4	0.6	4	16	64	24
5	0.7	5	25	125	120
6	0.8	6	36	216	720
7	0.8	7	49	343	5040
8	0.9	8	64	512	40320
9	1.0	9	81	729	362880
10	1	10	100	1000	3628800
100	2	100	10000	1000000	9.33E+157
1000	3	1000	1000000	1000000000	
10000	4	10000	100000000	1000000000000	


O(N)
O(log N^3)
O(log N^2)
O(log N)
O(c)


O(N^3)
O(N^4)
O(N^2)
O(N log N)
O(N)
O(N!)

$cN + dN^2$ => $O(N^2)$

$N^4 + N!$ => $O(N!)$

$2N$ => $O(N)$

$500N$ => $O(N)$

```
my @list1 = (10 .. 100);
my @list2 = (50 .. 200);

# Walk over @list1
foreach my $element (@list1) {
   # Check in @list2 to see if it's there
   foreach my $item (@list2) {
      if($item eq $element) {
         print "$item is in both lists\n";
      }
   }
}
```

```
for each N
  for each M
    do something
```

$$O(NM) \Rightarrow O(N^2)$$

```
my @list1 = (10 .. 100);          # length N
my @list2 = (50 .. 200);          # length M

# Create a hash of all the values in
# @list1: O(N)
my %in_list1;
foreach my $element (@list1) {
   $in_list1{$element} = ();
}

# Walk over each element in @list2 and
# compare against the hash: O(M)
foreach my $item (@list2) {
   if(exists $in_list1{$item}) {
      print "$item is in both lists\n";
   }
}
```

$$O(M+N) \Rightarrow O(2N)$$
$$\Rightarrow O(N)$$

```
use Fatal qw(open);
open(my $addresses, "<", "addresses.txt");

# Get all the suburbs
my %suburbs;
while(<$addresses>) {
   my ($business, $street_address, $suburb) = split("\t");
   $suburbs{$suburb} = 1;
}
close $addresses;

# Print out suburbs and corresponding businesses
foreach my $curr_suburb ( keys %suburbs ) {
   print "<h2>$curr_suburb</h2>\n";

   open(my $addresses, "<", "addresses.txt");
   while(<$addresses>) {
      my ($business, $street_addr, $suburb) = split("\t");
      next unless $suburb eq $curr_suburb;

      # print business details...
   }
}
```

Lines in file: N
Total distinct suburbs: M

$$O(N + NM) \Rightarrow O(NM)$$
$$\Rightarrow O(N^2)$$

(even if we're getting the suburbs from somewhere else!)

```
open(my $addresses, "<", "addresses.txt");

# Get all the suburbs
my %suburbs;
while(<$addresses>) {
   my ($business, $street_addr, $suburb)=split("\t");
   push @{$suburbs{$suburb}}, {
      business_name => $business,
      street_address => $street_addr,
   };
}
close $addresses;

# Print out suburbs and corresponding businesses
foreach my $curr_suburb (keys %suburbs ) {
   print "<h2>$curr_suburb</h2>\n";

   foreach my $business (@$suburbs{$curr_suburb}) {
      # print business details
   }
}
```

Lines in file: N
Total distinct suburbs: M
Businesses per suburb: L

(note: M*L = N)

```
O(N + ML)  => O(N + N)
           => O(2N)
           => O(N)
```

```
# Get all the suburbs
my %suburbs;
while(<$addresses>) {
   my ($business, $street_addr, $suburb)=split("\t");
   push @{$suburbs{$suburb}} = {
      business_name => $business,
      street_address => $street_address,
   };
}

...
    foreach my $business (@$suburbs{$curr_suburb}) {
      # print business details
      print $business->{business_name};
      print $business->{street_address};
   }
```

```
%suburbs = (
    Preston => [
        {
            'street_address' => '645 High Street',
            'business_name' => 'Snap Printing'
        },
        {
            'street_address' => '780 High Street',
            'business_name' => 'Great Pie Place'
        }
    ],
    Coburg => [
        {
            'street_address' => '104 Elizabeth Street',
            'business_name' => 'Perl Training Australia'
        }
    ]
);
```

N^2 solutions are...

- Obvious
- Easy to write
- Easy to verify
- Slow

```
# Determine distances for all customers
my %customers;
while(<$customers>)
{
   my ($postcode, $name, $address, $suburb, $phone)
      = split("|", $_, 2);
   my $distance = calculate_g_circle_distance($postcode);

   # Add postcode and distance information
   $customers{$postcode}{distance} = $distance;

   my $customer = {
         name    => $name,
         address => $address,
         suburb  => $suburb,
         phone  => $phone,
   };

   push @{$customers{$postcode}{customers}}, $customer;
}
```

```
while(<$customers>)
{
   my ($postcode, $name, $address, $suburb, $phone)
      = split("|", $_, 2);
   my $distance = calculate_g_circle_distance($postcode);

   # Add postcode and distance information
   $customers{$postcode}{distance} = $distance;


   ...
```

```
# Determine distances for all customers
my %distances;
while(<$customers>)
{
   my ($postcode, $name, $address, $suburb, $phone)
      = split("|", $_, 2);

   my $distance;
   if(exists $distances{$postcode}) {
      $distance = $distances{$postcode};
   }
   else {
      $distance = calculate_g_circle_distance($postcode);
   }
   ....
}
```

With best wishes for A Merry Christmas and A Happy New Year

From - Ian Campbell.

```
use Memoize;
memoize('calculate_g_circle_distance');

while(<$customers>)
{
   my ($postcode, $name, $address, $suburb, $phone)
      = split("|", $_, 2);
   my $distance = calculate_g_circle_distance($postcode);

   # Add postcode and distance information
   $customers{$postcode}{distance} = $distance;

   ...
```

```
# Determine distances for all customers
# ... as before

# Do something with customer information
foreach my $postcode (keys %customers) {

   # Skip if too far away
   next if $customers{$postcode}{$distance} > 100;

   ...
}
```

```
# Determine distances for all customers
my %customers;
while(<$customers>)
{
   my ($postcode, $name, $address, $suburb, $phone)
      = split("|", $_, 2);
   my $distance = calculate_g_circle_distance($postcode);

   next unless $distance < 100;

   # Add postcode and distance information
   $customers{$postcode}{distance} = $distance;

   my $customer = {
         name    => $name,
         address => $address,
         suburb  => $suburb,
         phone   => $phone,
   };

   push @{$customers{$postcode}{customers}}, $customer;
}
```

```
while(my $id = <$client_ids>) {
   my $customer = get_customer_details($id);

   # Generate mail details
   my $mail_label   = $customer->generate_addr_label();
   my $letter_start = $customer->generate_letter_start();
   my $letter       = $letter_start
                    . $letter_text
                    . $letter_foot;

   # Don't print anything if they're an old customer
   my $last_seen    = $customer->days_since_last_order();
   if($last_seen > $tollerance) {
      next;
   }

   # Print
   print_label($mail_label);
   print_letter($letter);
}
```

```
while(my $id = <$client_ids>) {
   my $customer = get_customer_details($id);

   # Don't print anything if they're an old customer
   my $last_seen   = $customer->days_since_last_order();
   if($last_seen > $tollerance) {
      next;
   }

   # Generate mail details
   my $mail_label   = $customer->generate_addr_label();
   my $letter_start = $customer->generate_letter_start();
   my $letter       = $letter_start
                    . $letter_text
                    . $letter_foot;

   # Print
   print_label($mail_label);
   print_letter($letter);
}
```


WILLIAM MURPHY'S
ELECTRICALLY POWERED
SCENIC RAILWAY.

```
# Break up weblog into subject
# weblogs
foreach my $subject (@many_subjects)
{
   system("grep $subject $web_log
            > $subject.log");
}
```

```
# Break up weblog into subject weblogs
foreach my $subject (@many_subjects) {
    open(my $sub_fh, ">", "$subject.log");
    open(my $web_fh, "<", $web_log);

    # Copy all subject lines into $sub_fh
    while(<$web_fh>) {
       if(/$subject/) {
           print {$sub_fh} $_;
       }
    }
    close $sub_fh;
    close $web_fh;
}
```

```
# Open filehandles for each subject
my %filehandles;
foreach my $subject (@many_subjects) {
      open my $fh, “>”, $subject;
      $filehandles{$subject} = $fh;
}

# Walk over weblog and copy to appropriate
# subject log
open(my $web_fh, “<”. $web_log);
while(<$web_fh>) {
     my ($subject) = m/s(\d{3}-\d{3})/;
     if(exists $filehandles{$subject}) {
          print {$filehandles{$subject}} $_;
     }
}

# close filehandles
```

```
open my $in_fh, "<", $super_dooper_big_file;
open my $out_fh, ">", $sorted_file;

# Read in lines and sort
my @lines = <$in_fh>;
my @sorted = sort @lines;

# Store sorted lines in file
print {$out_fh} @sorted;

close $in_fh;
close $out_fh;
```

```
# Unix sort
system("sort", "-o $sorted_file",
       $super_dooper_big_file);

# Win32 sort
system("sort", $super_dooper_big_file,
       "/O $sorted_file");
```

Code grows over time

=> Code is organic

Code is NOT organic

Premature Optimisation = Bad!

Writing smarter, faster code = Good!

Optimisation hints

- Determine the Big-O for your code
- Memoize
- Throw data away
- Leave loops early
- Utilise system() wisely

Old code

- Profile
- Benchmark
- Apply all the previous hints
- Then (only if required) perform more optimisations

Credits

- See notes page for images to see individual photo credits/licenses
-
-

Not common enough optimisations

Jacinta Richardson
Perl Training Australia
<jarich@perltraining.com.au>

Good morning everyone.

I'm Jacinta, and I work for Perl Training Australia. In my job I get to see a lot of other people's code, and so I'm here to talk to you a bit about the problems I've seen.

Most of the code in this talk is based on real examples, although simplified.

Before I get started, I just want to emphasise that this talk is about effective use of tools. I'm talking about not doing this:

Photo: Johnny Magnusson.

from Wikipedia
http://en.wikipedia.org/wiki/Image:Toolbox.jpg

Citing:
http://www.logodesignweb.com/stockphoto

with your code. There are much more efficient ways to store your toilet paper.

I'm also not talking about...

Picture: rlr77
http://flickr.com/photos/boricua/401520525/

over optimisation

or..

Picture: mindgraph
http://flickr.com/photos/johann-in-london/11089

adding new features...

Won-Tolla
http://flickr.com/photos/wontolla/180208695/
CC By NC SA

I'm talking about improving how you write code right now, and helping you use that to improve the code you have to maintain.

Real optimisation can come later; if it proves necessary.

so...

Picture: ashooo
http://flickr.com/photos/42831233@N00/433218726/

Where do we start?

where do we start? I think it's best to consider the tools we can use when we first realise our code is too slow. Let's consider a problem

```
sub strings {
  my ($start, $end) = @_;
  my $full_string = '';
  foreach my $str ($start .. $end) {
    my $temp = $full_string . $str;
    $full_string = $temp;
  }
  return $full_string;
}

sub numbers {
  my ($start, $end) = @_;
  my $sum = 0;
  foreach my $num ($start .. $end) {
    my $temp = $sum + $num;
    $sum = $temp;
  }
  return $sum;
}
```


Consider this code. Two subroutines, both which are making the same mistake. For the first we're trying to join a number of elements together in a string. But we're copying the existing string, plus the new element into a temporary variable, and then copying that back into our storage variable!

In the second we want to add a bunch of numbers together. We're doing the calculation, storing that result in a temporary variable and copying it back.

If we can only improve one of these subroutines, which do we pick? Strings? Numbers? Who thinks we might not have enough information? Best perhaps to do a little...

profiling.

Photo: Wikipedia
http://en.wikipedia.org/wiki/Image:Criminal_cas

```
# perl -d:DProf some_program.pl
# dprofpp

Total Elapsed Time = 1.165982 Seconds
  User+System Time = 1.175982 Seconds
Exclusive Times
%Time ExclSec CumulS #Calls sec/call Csec/c  Name
91.6   1.078  1.078   1000   0.0011 0.0011  main::numbers
9.35   0.110  0.110      1   0.1100 0.1100  main::strings
0.00       - -0.000      1        -      -  strict::import
0.00       - -0.000      1        -      -  strict::bits
0.00       - -0.000      1        -      -  main::BEGIN
```


Let's look at what this is saying. Wow, the strings method takes a really long time per call. But it's only being called once. Numbers on the other hand is super-fast, but it's being called heaps of times, and that means it's taking up 91% of our program execution time.

This gives us a hint on how we could improve our program

Remove strings() and save **9.35%**

Improve numbers() by 25% and save

22.9%!

Even if we completely removed strings() we'd only save 9% of our time. If we can make even a slight improvement to numbers() we can speed up our program greatly!

```
sub numbers {
   my ($start, $end) = @_;
   my $sum = 0;
   foreach my $num ($start .. $end) {
      my $temp = $sum + $num;
      $sum = $temp;
      $sum += $num;
   }
   return $sum;
}
```


If we remove the temporary variable, but otherwise keep the code the same, do we see an improvement?

```
Total Elapsed Time = 0.652972 Seconds
  User+System Time = 0.652972 Seconds
Exclusive Times
%Time ExclSec CumulS #Calls sec/call Csec/c  Name
 80.7   0.527  0.527   1000   0.0005 0.0005  main::numbers
 16.8   0.110  0.110      1   0.1100 0.1100  main::strings
 0.00       - -0.000      1        -      -  strict::bits
 0.00       - -0.000      1        -      -  strict::import
 0.00       - -0.000      1        -      -  main::BEGIN
```

0.527 vs 1.078 seconds total!

We've more than halved the time per call in numbers, and correspondingly decreased the overall program time.

The exclusive time spent in numbers is only half a second rather than 1!

All subroutines half price!

Profiling is a great way to identify areas in your existing code that need improvement. But what if you want to compare a few solutions without changing your working code. Here we can use...

Image by Jovike, flickr:
http://flickr.com/photos/jvk/344219061/

bench marking. ;)

Photo by dailyjoe.
http://flickr.com/photos/dailyjoe/441020930/

```
use Benchmark qw(cmpthese);

cmpthese(0, {
        slow => sub {
                my ($start, $end) = (2000, 4000);
                my $sum = 0;
                foreach my $num ($start .. $end) {
                        my $temp = $sum + $num;
                        $sum = $temp;
                }
                return $sum;
        },
        fast => sub {
                my ($start, $end) = (2000, 4000);
                my $sum = 0;
                foreach my $num ($start .. $end) {
                        $sum += $num;
                }
                return $sum;
        },
});
```


In Perl we use the benchmark module, like this:

compare our slow subroutine... with our fast subroutine. And our results look like:

```
# perl benchmark.pl
       Rate slow fast
slow  908/s   -- -51%
fast 1844/s 103%   --
```


this. Just as we saw earlier, not using the temporary variable, makes the subroutine twice as fast.

Now, as I said, these are essential for optimising existing code. But I want to help you write code that already starts out fast. Without any cool too-early-optimisations that your peers will point and laugh at.

It's about measurements.

Picture: emilgh
http://www.flickr.com/photos/emilgh/42509423

Big-O notation

Big-O notation is a way of measuring how something scales. Even the most inefficient algorithms tend to finish quickly enough for small sets of data. It's when you have LOTS of data that your algorithm choice becomes important.

$O(f(n)) \equiv$ order of $f(n)$

Big O notation measures the order of a function. How fast the execution speed increases compared to the increase in our data set, which we'll call N.

• O(c)	Constant
• O(log n)	Logarithmic
• O((log n)^c)	Poly-logarithmic
• O(n)	Linear
• O(n log n)	Quasi-linear
• O(n^c) c>1	Polynomial
• O(c^n)	Exponential
• O(n!)	Factorial

21

Here are some common orders, listed from most desirable to least desireable. O(c) is constant time. It takes the same time to complete the operation regardless of the size of the data set. For example looking things up in a hash – it doesn't matter if there are 5 or 500,000 elements in the hash – it takes about the same time.

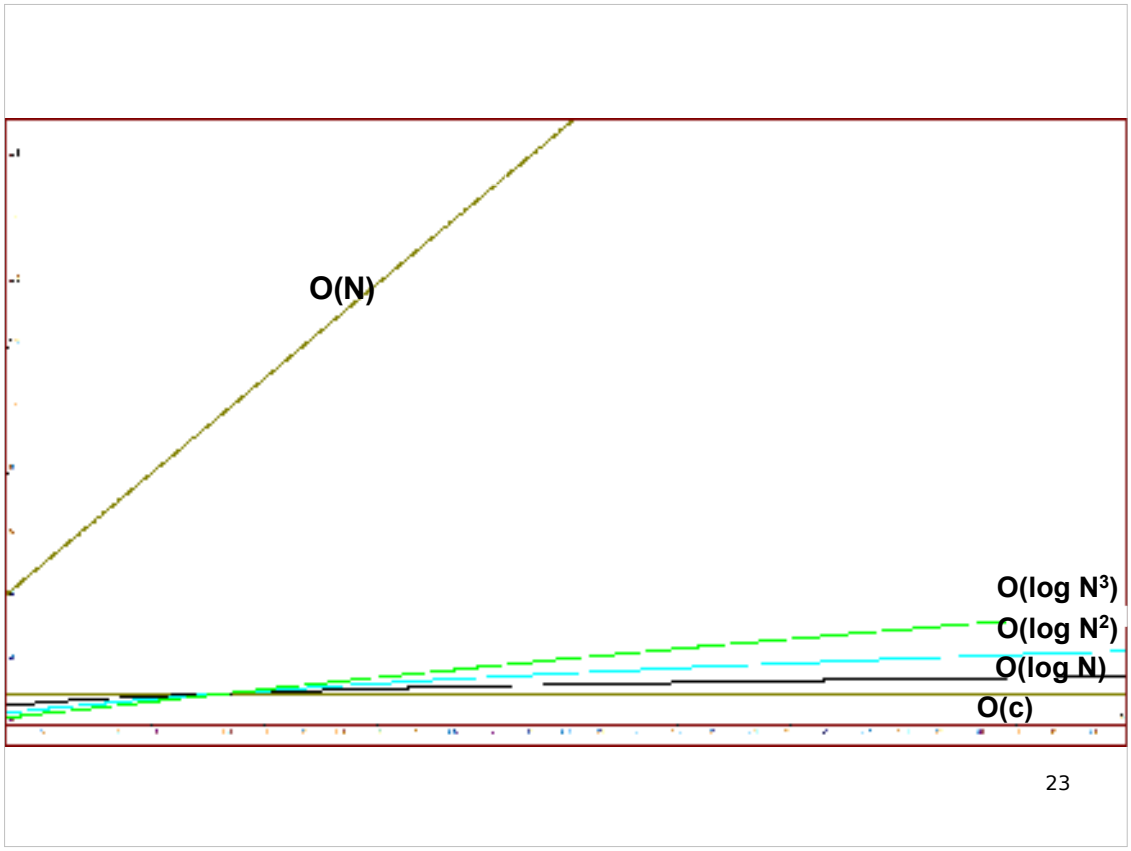
A binary search over an ordered list is log-n. Deciding whether a number is prime using the AKS primality test is log n^c

Anything that involves walking over a list is order n

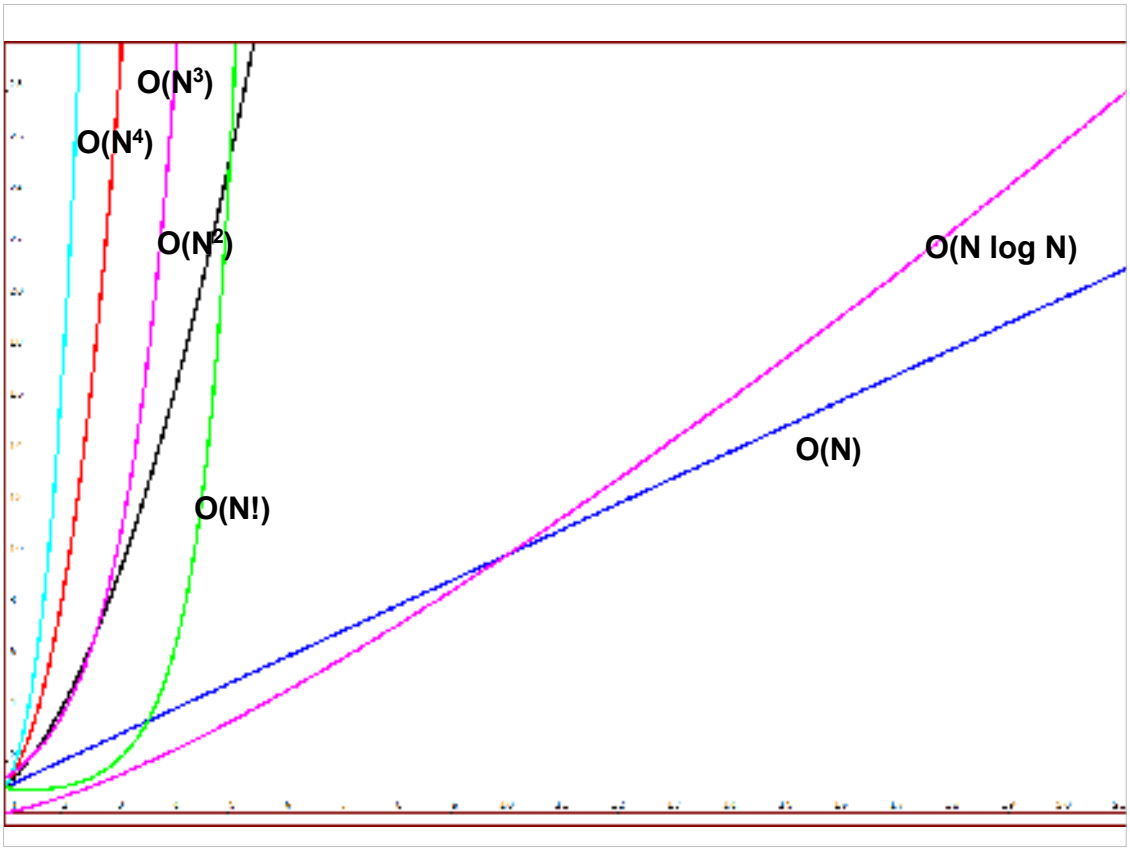
Good sort algorithms such as merge sort, quick sort and heap sort are n log n.
Insertion sort is n^2.

N	log n	n	n^2	n^3	n!
1	0.0	1	1	1	1
2	0.3	2	4	8	2
3	0.5	3	9	27	6
4	0.6	4	16	64	24
5	0.7	5	25	125	120
6	0.8	6	36	216	720
7	0.8	7	49	343	5040
8	0.9	8	64	512	40320
9	1.0	9	81	729	362880
10	1	10	100	1000	3628800
100	2	100	10000	1000000	9.33E+157
1000	3	1000	1000000	1000000000	
10000	4	10000	100000000	1000000000000	

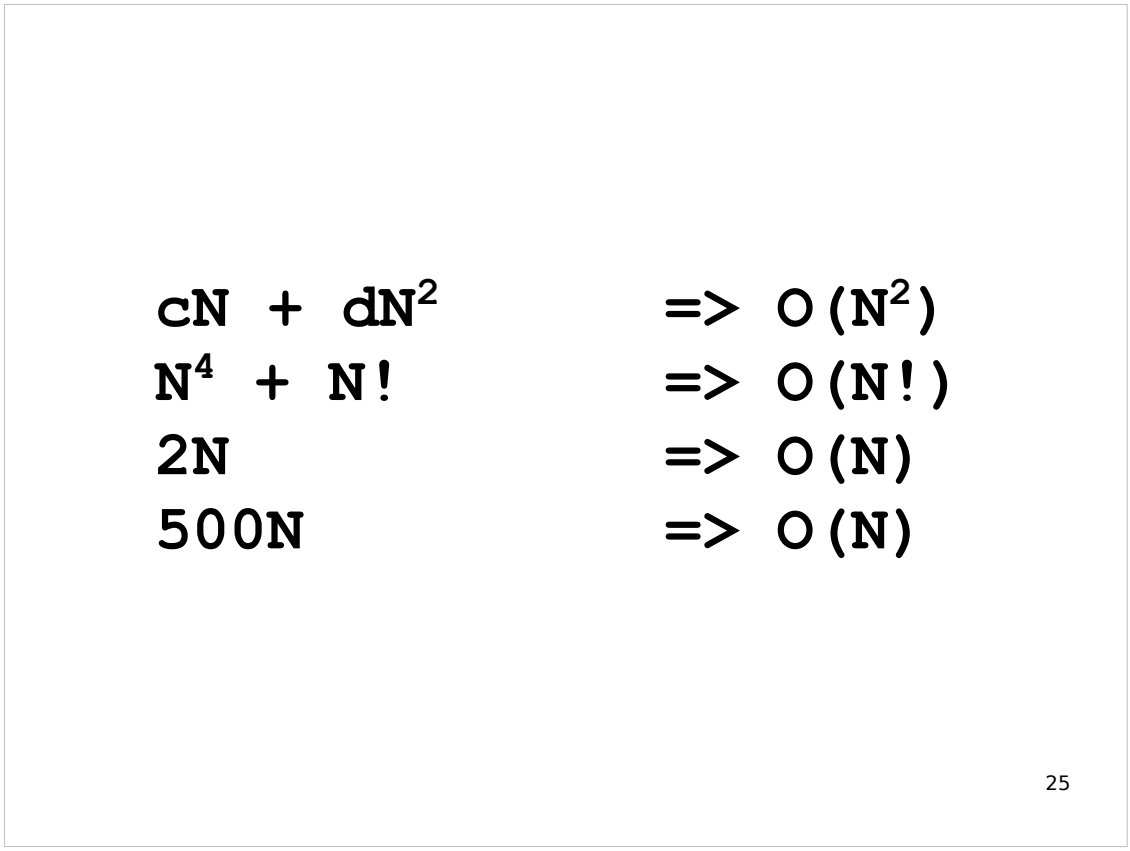
This table shows how these functions grow for some values of N. Imagine each of these is a second of run time. You can see that for N squared, we reach a minute after N grows to eight units. N cubed only takes 4 units and while n! is faster thre, it's much much slower after 6 units. With Order (N) we reach a minute when we have 60 units, but log n only when we have 10 million!



Here's the same kind of information in graphical form. Here we have O(N) and then logn cubed, squared, log n and constant time.



This has our higher order values. You can see that nlogn grows similarly to order N, and that the other ones race out of bounds at a tremendous speed.



It's important to understand that O-notation is about the order of an operation. Thus we look at the fastest growing part. In this case, we just need to consider the grown of N squared

Although N ^ 4 grows really quickly, and is bigger than N! for really small N, N! grows the fastest overall, so we just concentrate on that.

2 N is still Order N. That two isn't changing, and although it makes a difference, we still know that for each increase in N; there is a specific linear increase in our execution time.

500 N is also Order N. For large N, that 500 probably isn't going to make much of a difference. N is better than 500N for small

```
my @list1 = (10 .. 100);
my @list2 = (50 .. 200);

# Walk over @list1
foreach my $element (@list1) {
  # Check in @list2 to see if it's there
  foreach my $item (@list2) {
    if($item eq $element) {
      print "$item is in both lists\n";
    }
  }
}
```


Let's look at some code.

Here we're looking for the intersection of two lists (that is the elements both lists have in common). First we look through the first list and then search for each item there in the second list... It's a straight forward implementation.

```
for each N
  for each M
    do something
```

$O(NM) \Rightarrow O(N^2)$

27

Let's look at the Big-O for this problem.

Consider the case where both lists are about the same size. That makes it's O N squared. For each increase in in either list, we have to do lots of extra work.

Is there a better way?

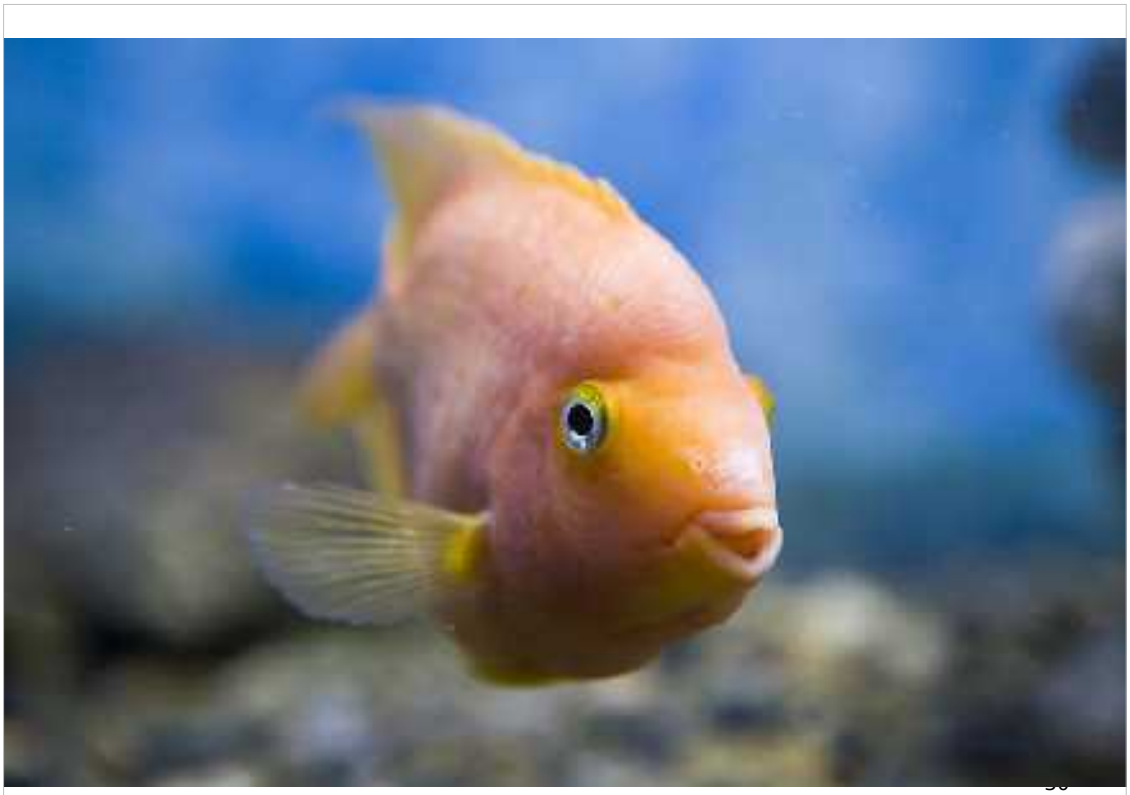
```
my @list1 = (10 .. 100);          # length N
my @list2 = (50 .. 200);          # length M

# Create a hash of all the values in
# @list1: O(N)
my %in_list1;
foreach my $element (@list1) {
  $in_list1{$element} = ();
}

# Walk over each element in @list2 and
# compare against the hash: O(M)
foreach my $item (@list2) {
  if(exists $in_list1{$item}) {
    print "$item is in both lists\n";
  }
}
```


This code solves the same problem. First we create a hash to store all the values in the first list. This operation involves looking at each element once, so that's order N.

Then we walk over each element in our second list and see if it exists in the hash. This operation involves looking at each element once, so that's order M.

```
O(M+N) => O(2N)
       => O(N)
```


So our problem is Order N. For an increase in either list, we only do one extra operation. This code will run many times faster than the previous example!



happy code!

Picture: PixelFixer
http://flickr.com/photos/pixelfix/1359497270/

CC

```
use Fatal qw(open);
open(my $addresses, "<", "addresses.txt");

# Get all the suburbs
my %suburbs;
while(<$addresses>) {
   my ($business, $street_address, $suburb) = split("\t");
   $suburbs{$suburb} = 1;
}
close $addresses;

# Print out suburbs and corresponding businesses
foreach my $curr_suburb ( keys %suburbs ) {
   print "<h2>$curr_suburb</h2>\n";

   open(my $addresses, "<", "addresses.txt");
   while(<$addresses>) {
      my ($business, $street_addr, $suburb) = split("\t");
      next unless $suburb eq $curr_suburb;

      # print business details...
   }
}
```


Here's a slightly more complex example. In this code we're opening a file of customer addresses; walking over that list and recording all the suburbs.

Once we've done that, we walk through all of our suburbs; open the file again, and print out each business in that suburb!

(You might laugh, but this is based on a real life example)

So what's the Big-O for this?

Lines in file: N
Total distinct suburbs: M

$$O(N + NM) \Rightarrow O(NM)$$
$$\Rightarrow O(N^2)$$

(even if we're getting the suburbs from somewhere else!)

Even if we remove the initial gathering of suburbs and just walk through a list of desired suburbs, it's still O N squared.

```
open(my $addresses, "<", "addresses.txt");

# Get all the suburbs
my %suburbs;
while(<$addresses>) {
   my ($business, $street_addr, $suburb)=split("\t");
   push @{$suburbs{$suburb}}, {
      business_name => $business,
      street_address => $street_addr,
   };
}
close $addresses;

# Print out suburbs and corresponding businesses
foreach my $curr_suburb (keys %suburbs ) {
   print "<h2>$curr_suburb</h2>\n";

   foreach my $business (@$suburbs{$curr_suburb}) {
      # print business details
   }
}
```


I don't want you to think that whenever you see nested loops that it's O N squared. It probably is; but you should still look more carefully.

This is an alternate solution. Here we open the file, and then for each entry we add the business details to an array of hashes. I'll show what this ends up looking like in a moment.

Then we walk through all of our suburbs, and print out the businesses we've stored.

Lines in file: N
Total distinct suburbs: M
Businesses per suburb: L

(note: M*L = N)

O(N + ML) => O(N + N)
=> O(2N)
=> O(N)

Now our solution is Order N!

```
# Get all the suburbs
my %suburbs;
while(<$addresses>) {
   my ($business, $street_addr, $suburb)=split("\t");
   push @{$suburbs{$suburb}} = {
      business_name => $business,
      street_address => $street_address,
   };
}

...
    foreach my $business (@$suburbs{$curr_suburb}) {
      # print business details
      print $business->{business_name};
      print $business->{street_address};
   }
```


So let's have a look at that code again. How are we achieving this?

Our complex data structure is a hash of arrays of hashes. To create it manually we'd write something like:

```
%suburbs = (
    Preston => [
        {
            'street_address' => '645 High Street',
            'business_name' => 'Snap Printing'
        },
        {
            'street_address' => '780 High Street',
            'business_name' => 'Great Pie Place'
        }
    ],
    Coburg => [
        {
            'street_address' => '104 Elizabeth Street',
            'business_name' => 'Perl Training Australia'
        }
    ]
);
```


this.

Create a suburbs hash, for each suburb, we record an array of included businesses; for which we're currently storing street address and business name.

We could also throw other data in here, if we wanted, such as business contact name, phone number etc.

Anything! rather than to read that file again!

One of the most important things you can ever do to improve the speed of your code is to learn how to use complex data structures. There are two reasons why the original code for my last example was written the way it was. One reason is that the original author thought references were too hard, so she didn't even consider solutions which used them.

Once you've understood complex data structures, climbing this cliff face is more like:

Picture: nothing
http://flickr.com/photos/nothing/6817165/

this! :)

Picture: bprm2
http://flickr.com/photos/landscape_photography

N^2 solutions are...

- Obvious
- Easy to write
- Easy to verify
- Slow

The other reason the code's author wrote an N-squared solution was because n-squared solutions are usually the first ones that come to mind!

They're obvious. Easy to write. Easy to verify... They'd be perfect. IF they weren't so slow!

Picture: SeanMack
http://en.wikipedia.org/wiki/Image:Snail_climbin
GFDL

The final thing I should say about this solution is that we're making a memory-time trade-off. For really really large files of customers it may not be possible to load all the information into memory. Even so, we can avoid an N squared solution by using temporary files.

In the more normal case, memory is so plentiful on machines these days that typically it's a lot cheaper to take up all the memory for a few minutes than for the process to run for a week.

Picture: NASA
http://en.wikipedia.org/wiki/Image:Wild_Pig_KSC

Public domain

My next optimisation tip relates to throwing stuff away.

Picture: Daniel Candido
http://en.wikipedia.org/wiki/Image:Lixo.jpg
Public domain

```
# Determine distances for all customers
my %customers;
while(<$customers>)
{
   my ($postcode, $name, $address, $suburb, $phone)
      = split("|", $_, 2);
   my $distance = calculate_g_circle_distance($postcode);
   # Add postcode and distance information
   $customers{$postcode}{distance} = $distance;

   my $customer = {
         name    => $name,
         address => $address,
         suburb  => $suburb,
         phone   => $phone,
   };

   push @{$customers{$postcode}{customers}}, $customer;
}
```


This code is fairly similar to the previous one. We read in all of our data, calculate the great circle distance (basically the distance between two suburbs based on their longitude and latitude) and then store it and the customer information.

This is a hash of hashes where one of the hash elements is an array.

Can anyone spot the optimisation problem with this code?

```
while(<$customers>)
{
   my ($postcode, $name, $address, $suburb, $phone)
      = split("|", $_, 2);
   my $distance = calculate_g_circle_distance($postcode);

   # Add postcode and distance information
   $customers{$postcode}{distance} = $distance;

   ...
```


We're calculating the great circle distance again and again each time we look at the same suburb!

Worse, this particular calculation involves a lot of work.

```
# Determine distances for all customers
my %distances;
while(<$customers>)
{
   my ($postcode, $name, $address, $suburb, $phone)
      = split("|", $_, 2);

   my $distance;
   if(exists $distances{$postcode}) {
      $distance = $distances{$postcode};
   }
   else {
      $distance = calculate_g_circle_distance($postcode);
   }
   ....
}
```


The obvious solution to this is to keep a record of the suburbs and distances we've already calculated and check it for each new suburb.

This does have a flaw though. You need your record to be accessible for all scopes in which the subroutine is called!

But there's an alternative!

We can give Perl a memo!

Picture: Tony Corsini
http://en.wikipedia.org/wiki/Image:Burns_on_Ay
Public domain

```
use Memoize;
memoize('calculate_g_circle_distance');

while(<$customers>)
{
   my ($postcode, $name, $address, $suburb, $phone)
      = split("|", $_, 2);
   my $distance = calculate_g_circle_distance($postcode);
   # Add postcode and distance information
   $customers{$postcode}{distance} = $distance;

   ...
```


Memoize works on subroutines which are functions in the mathematical sense. That is, when given the same input, the output is always the same.

Thus we can tell Perl to remember any return value from calculate g circle distance and then we can just leave the rest of our code untouched!

The next trick, is to travel less far.

Picture: HORIZON
http://flickr.com/photos/horizon/119907433/

```
# Determine distances for all customers
# ... as before

# Do something with customer information
foreach my $postcode (keys %customers) {

   # Skip if too far away
   next if $customers{$postcode}{$distance} > 100;

   ...
}
```


In the previous code we built up a data structure containing postcodes, distances and all the relevant data. Here, in another loop, we're now ignoring all of those which are too far away.

This is much more work than we need to do!

```
# Determine distances for all customers
my %customers;
while(<$customers>)
{
   my ($postcode, $name, $address, $suburb, $phone)
      = split("|", $_, 2);
   my $distance = calculate_g_circle_distance($postcode);

   next unless $distance < 100;

   # Add postcode and distance information
   $customers{$postcode}{distance} = $distance;

   my $customer = {
         name    => $name,
         address => $address,
         suburb  => $suburb,
         phone   => $phone,
   };

   push @{$customers{$postcode}{customers}}, $customer;
}
```


If we know we don't want customers which are too far away, we can discard them at the start, saving us time and memory.

Now we're fast

Photo: Fmickan
http://en.wikipedia.org/wiki/Image:Feldhase.jpg
GFDL

```
while(my $id = <$client_ids>) {
   my $customer = get_customer_details($id);

   # Generate mail details
   my $mail_label   = $customer->generate_addr_label();
   my $letter_start = $customer->generate_letter_start();
   my $letter       = $letter_start
                      . $letter_text
                      . $letter_foot;

   # Don't print anything if they're an old customer
   my $last_seen   = $customer->days_since_last_order();
   if($last_seen > $tollerance) {
      next;
   }

   # Print
   print_label($mail_label);
   print_letter($letter);
}
```


Code tends to grow overtime. A collection of "quick fixes" which solve the problem of the moment.

Let's look at this code which sends out newsletters to a company's clients. First we get our customer's details, we generate the letter and down here we print it out.

A few years ago, the company decided that sending newsletters to clients they hadn't heard of for years was expensive. So the programmer put in this test.

Can you spot the problem?

```
while(my $id = <$client_ids>) {
   my $customer = get_customer_details($id);

   # Don't print anything if they're an old customer
   my $last_seen   = $customer->days_since_last_order();
   if($last_seen > $tollerance) {
      next;
   }

   # Generate mail details
   my $mail_label   = $customer->generate_addr_label();
   my $letter_start = $customer->generate_letter_start();
   my $letter       = $letter_start
                    . $letter_text
                    . $letter_foot;

   # Print
   print_label($mail_label);
   print_letter($letter);
}
```


Generating the newsletter sounds expensive!

We can actually short-circuit this loop much earlier than we were!

The next optimisation has to do with calling system.

Picture: chalkie colour circles
http://flickr.com/photos/chalkie_circle2000/270
CC NC

```
# Break up weblog into subject
# weblogs
foreach my $subject (@many_subjects)
{
  system("grep $subject $web_log
          > $subject.log");
}
```


This is bad (but unfortunately rather common).

```
# Break up weblog into subject weblogs
foreach my $subject (@many_subjects) {
    open(my $sub_fh, ">", "$subject.log");
    open(my $web_fh, "<", $web_log);

    # Copy all subject lines into $sub_fh
    while(<$web_fh>) {
       if(/$subject/) {
           print {$sub_fh} $_;
       }
    }
    close $sub_fh;
    close $web_fh;
}
```


This is worse. we're losing readability for no significant time gains. (We might avoid some out-of-memory problems along the way though).

What's the Order of this code?

```
# Open filehandles for each subject
my %filehandles;
foreach my $subject (@many_subjects) {
      open my $fh, ">", $subject;
      $filehandles{$subject} = $fh;
}

# Walk over weblog and copy to appropriate
# subject log
open(my $web_fh, "<". $web_log);
while(<$web_fh>) {
     my ($subject) = m/s(\d{3}-\d{3})/;
     if(exists $filehandles{$subject}) {
           print {$filehandles{$subject}} $_;
     }
}

# close filehandles
```


Here we turn the problem inside out. First we open a bunch of filehandles and store them in a hash. Then we walk over the weblog once and look at each line for a subject. Once we find that, we print it to the correct filehandle.

This is O(N),

it'll fly.

Picture: U.S. Navy photo by Ensign John Gay
http://en.wikipedia.org/wiki/Image:FA-18_Hornet
Public Domain

```
open my $in_fh, "<", $super_dooper_big_file;
open my $out_fh, ">", $sorted_file;

# Read in lines and sort
my @lines = <$in_fh>;
my @sorted = sort @lines;

# Store sorted lines in file
print {$out_fh} @sorted;

close $in_fh;
close $out_fh;
```


Using system isn't bad. In fact sometimes it's a much better solution. In this code we open a super dooper big file, read it into memory, sort it and then print it out.

This is fine for small files. But what if the file is 60gig and we only have 4 gig of memory in the machine? That's a whole lot of swapping.

Your operating system's “sort” is probably going to be faster

```
# Unix sort
system("sort", "-o $sorted_file",
       $super_dooper_big_file);

# Win32 sort
system("sort", $super_dooper_big_file,
       "/O $sorted_file");
```


Using system does introduce portability issues, which need to be considered when using such an approach.

Code grows over time

=> Code is organic

Code cruft grows over time, leading some people to suggest that code is organic.

Code is NOT organic

It isn't.

You code doesn't have to grow. And if you find yourself maintaining slow, inefficient code, rip the cruft out!

Code can't cry.

Picture: Lawerence Whittemore
http://flickr.com/photo_zoom.gne?id=145786038&size=m

Premature Optimisation = Bad!

Writing smarter, faster code = Good!

64

So, finally, I'd like to remind you that premature optimisation is bad, but writing your code in a smarter, faster way is great!

Optimisation hints

- Determine the Big-O for your code
- Memoize
- Throw data away
- Leave loops early
- Utilise system() wisely

My hints have been:

....

Old code

- Profile
- Benchmark
- Apply all the previous hints
- Then (only if required) perform more optimisations

And before fixing old code, I recommend that you ...

Using these methods should make your code run much faster, giving you more time to...

relax.

Picture: Tommy Wong
http://www.flickr.com/photos/gracewong/18328
CC

Credits

- See notes page for images to see individual photo credits/licenses
-
-