Algorithms in Bioinformatics

Jim Tisdall

Programming for Biology

Lecture Notes

Suggested Reading

```
Mastering Algorithms with Perl
        by Orwant, Hietaniemi, and Macdonald
        (An excellent algorithms text with implementations in Perl)

Introduction to Algorithms
        by Cormen et al.
        (This is the standard modern text)

Writing Efficient Code
        by Jon Bentley
        (Hard to find.  Great book.)

Introduction to Automata Theory, Languages, and Computation
        by Hopcroft and Ullman
        (The standard, mathematical textbook for theoretical computer science.)

Computers and Intractability: A Guide to the Theory of NP-Completeness
        by Gary and Johnson
        (Very well written.)

Network Programming with Perl
        by Lincoln Stein
        (Client-server network programming.)

An Introduction to Parallel Algorithms
        by Joseph Jaja
        (For the next generation of computers.)
```

Programming for Biology

Jim Tisdall, James.Tisdall -- at -- DuPont.com
Last modified: Wed Oct 14 16:14:01 EDT 2009

Time and Space and Algorithms

A program's use of time and space depends on the **algorithms** and associated **data structures** used to solve a problem.

Typically there are many *algorithms* (ways to solve a problem in a computer.) Some ways use less time and/or less space than other ways. Finding the good ways is the study of the design and analysis of algorithms.

An **algorithm** is the design or idea of a computation. It can be expressed in terms of a specific computer program, or more informally as in *pseudocode*.

A **data structure** is the form of the computation as it proceeds. A great deal of biological data is organized into **two-dimensional tables in relational databases**. Relational database tables are the standard workhorse for storing data in biology, and are useful in a surprising number of situations.

It's important to know, however, that **often the best algorithm will use some other data structure** such as a doubly-linked list or a tree, for example. Such data structures might better represent graph structures, gene networks, evolutionary relationships, and so on. And, such data structures may be used in sometimes surprising ways to speed up a computation.

The **space** of an algorithm is just the amount of computer memory it uses.

The **time** of an algorithm is usually given as a function on the size of the input. So if the input is of size n, the algorithm might take time n^2. So, for instance, if you gave such an algorithm a hundred genes, it would take about 10000 units of time to run; if you gave it ten thousand genes, it would take 100000000 units of time to run.

Time is roughly estimated according to the number of basic operations performed by your program as it runs. Basic operations are adding, concatenating two strings, printing, etc. The overall structure of the program is what is important, not an actual prediction of exactly how many seconds the program will take.

System building and knowing what can be computed

We are primarily interested in building software to achieve easily computed, but useful, results. We will not delve into the study of algorithms in any depth in this course. But it can easily happen that you may want to compute something that is hard to compute in a week, or a year, or even at all. This is a practical problem, and it's important to know what you can do about it.

The idea is that **there are limits to what can be computed.** These limits take two main forms: **intractability** and **undecidability**.

The main point:
MANY PROBLEMS CANNOT BE COMPUTED
but it's possible to get "pretty good" answers for many of them

How algorithms are measured

Algorithms are typically classified by how fast they perform on inputs of varying sizes, by giving their speed as a function of the size of the input. The size of the input is usually called **n**.

Say for example that an algorithm gets an input of size **n**, and then just to write the answer it must write an output in space of size $\mathbf{2^n}$. (The amount of space that an algorithm uses is one way to establish a lower bound for how much time the algorithm takes to complete.) Then we say the algorithm's time complexity is *"order of 2 to the n"*, written in a shorthand called **big Oh** notation as

$O(\mathbf{2^n})$.

This way of measuring an algorithm is called *time complexity*.

Examples:
$O(\mathbf{2^n})$ computations: *intractable* (e.g. *exponential*) is *bad*
$O(\mathbf{n^2})$ computations: *tractable* (e.g. *polynomial*) is *good*
O(**5n**) computations: *tractable* (e.g. *linear*) is *great*
O(**log(n)**) computations: *tractable* (e.g. *logarithmic*) is *amazing*

If the size of the input **n** is 3, then all methods take a short amount of time -- 8 and 9 and 15 and about 1, respectively.

But if the size of the input **n = 100** , then **log(n)** is about 6, **5n** is 500, and $\mathbf{n^2}$ is 10,000 which is still not bad. However, $\mathbf{2^n}$ is bigger than the number of atoms in the universe. (And is the universe really finite? Oh well ... who's counting?)

Intractability

Intractability means that a problem cannot be computed in a reasonable amount of time. Many biological problems are intractable.

Example: in phylogeny we learn that there are many possible trees that can be built, and that the number of possible trees grows exponentially as you increase the number of taxa and as you increase the evolutionary time under discussion.

To find the best solution in an exponentially-growing space, such as the space of all possible evolutionary trees, often requires examining each possibility, and so may take an exponentially-growing time. Problems that have this property (very loosely defined here) are called
NP
(for non-deterministic polynomial time), and certain canonical such problems are called **NP-complete**.

NP-complete problems are all essentially interchangeable; that is, they all come down to essentially the same problem. The prototypical NP-complete problem is the

TRAVELING SALESMAN PROBLEM:
given a set of cities and the distances between them, what is the shortest route a traveling salesman can take to visit each one?

By the time you get to about 30 cities, the number of possible routes cannot be computed in your lifetime; by the time you reach about 60 cities, there are more possible routes than there are atoms in the universe. And we don't know a better way to find the best route than to look at each one.

An aside: no one has proved that NP-complete problems *must* require looking at each individual possibility. If you could find a *polynomial-time* algorithm for any NP-complete problem, you would be the most famous computer scientist/mathematician around, and would surely win a Nobel prize. Few people believe it will be done, but it's been an open problem for many years, and no one yet can prove that it can't be done. This is called the **P =? NP** problem.

The practical implications:

If you have a lot of data for your problem, and the problem is in NP, then you have **no practical solution to find the best, optimal answer** except on very small data sets.

But the good news is: there are **approximation algorithms** that will give you a **very good answer** in a reasonable amount of time, even if it's not the optimal answer. Such approximation algorithms underlie many of the practical approaches to such problems as phylogeny, sequence assembly, and many other problems in bioinformatics.

Undecidable problems

Less likely to be a problem for the practical bioinformatics programmer, but something to be aware of, is that there are **problems for which no solution is possible.**

These problems are called *undecidable*, and they were first demonstrated by Alan Turing and others in the 1930s.

Here's the most famous undecidable problem: the

HALTING PROBLEM
Write a program that can scan any other program and decide if it will eventually halt, or if it will go on forever without coming to a stop.

In other words, write a virus checker for nonhalting programs.

As an example of such a nonhalting "virus", here's a perl program that goes on forever (until you stop it):

```
while(1) {}
```

That looks easy to recognize. But we can *prove* that no program can be written that would catch *all* such non-halting programs.

The fact that such an easily-described problem as the HALTING PROBLEM has no solution is, when you think about it, a very deep and profound statement about the limits of human knowledge. But, nevertheless, and of a certainty, we all play on.

Programming for Biology Last modified: Mon Oct 20 23:14:38 EDT 2008

Using Less Time

The Art and Science of Algorithm Design

You can divide knowledge into two types: *procedural knowledge* and *declarative knowledge*.

Declarative knowledge is a collection of facts. (E.g., Watson's great textbook "The Molecular Biology of the Gene")

Procedural knowledge is knowledge of *how to do things*, and is the kind of knowledge captured by computer algorithms. Procedural knowledge has been growing immensely since (programmable digital) computers brought the ability to specify how to do something -- that is, to formulate an *algorithm* -- to the very center of our economic, scientific, and cultural lives.

Algorithms are discovered by a combination of mathematics and art and science and luck and training and talent. Much of what we do on computers relies on the accumulated procedural knowledge -- algorithms -- of our culture.

A good algorithm is more important than a good computer

Finding a better algorithm can be much more important than getting a better, faster computer.

For the following examples I created a set of random DNA that I'll use as my "promoters". I include the code here. (We'll return to this code later in the lecture).

```
#
# Main program -- make promoters from random DNA
#

srand();

$dna = make_random_DNA(1000000);
open(DNA, ">genomic_data") or die;
print DNA $dna;

@promoters = make_random_DNA_set(10, 5000);
open(PROMOTERS, ">promoters") or die;
print PROMOTERS join("\n",@promoters),"\n";

exit 0;

#
# Subroutines
#

# Make a string of random DNA of specified length.
sub make_random_DNA {

    my($length) = @_;
    my $dna;

    for (my $i=0 ; $i < $length ; ++$i) {
        $dna .= randomnucleotide(  );
    }

    return $dna;
}

# Make a set of random DNA
sub make_random_DNA_set {

    my($length, $size_of_set) = @_;
    my $dna;
    my @set;

    # Create set of random DNA
    for (my $i = 0; $i < $size_of_set ; ++$i) {

        $dna = make_random_DNA ( $length );
        push( @set, $dna );
    }

    return @set;
}

# Select at random one of the four nucleotides
sub randomnucleotide {

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

    return randomelement(@nucleotides);
}

sub randomelement {

    my(@array) = @_;

    return $array[rand @array];
}
```

Consider this fragment of perl code, written to find a set of short sequences in a genome ("findpromoters0"):

```
# Read the promoter data from a file
open(PROMOTERS, "promoters") or die "a horrible death: $!";
my @promoters = <PROMOTERS>;

# Look for each occurence of each promoter in the genome
foreach my $promoter (@promoters) {
        chomp $promoter;

        # Read the genome data from a file
        open(GENOME, "genome_data") or die "a horrible death: $!";
        my $genome = <GENOME>;

        while($genome =~ /$promoter/g) {
                # $-[0] prints the location of the find
                #print "$promoter $-[0]\n"; exit;
                $db{$promoter} = $-[0];
        }
}
```

Now this code is good perl. It is syntactically correct, and it will produce the correct output. It will run, and in the end you will print out all the locations of the sequence.

Let's see how long it takes to run:

```
-bash-3.00$ date; perl findpromoters0; date
Thu Oct 20 14:28:06 EDT 2005
Thu Oct 20 14:28:48 EDT 2005
-bash-3.00$
```

Okay, so 42 seconds isn't bad! But wait ... what if we had the entire human genome, and a million tags? I'll let you do the math, or the experiment, but it takes too long.

So we try to make it faster. How? Well, we notice that for each tag, we're reading in the entire genome from the disk. Let's rewrite the code so that it only reads the genome in once (findpromoters1):

```
# Read the genome data from a file
open(GENOME, "genome_data") or die "a horrible death: $!";
my $genome = <GENOME>;

# Read the promoter data from a file
open(PROMOTERS, "promoters") or die "a horrible death: $!";
my @promoters = <PROMOTERS>;

# Look for each occurence of each promoter in the genome
foreach my $promoter (@promoters) {
        chomp $promoter;
        while($genome =~ /$promoter/g) {
                # $-[0] prints the location of the find
                #print "$promoter $-[0]\n"; exit;
                $db{$promoter} = $-[0];
        }
}
```

And the time for that is:

```
-bash-3.00$ date; perl findpromoters1; date
Thu Oct 20 14:30:46 EDT 2005
Thu Oct 20 14:31:05 EDT 2005
-bash-3.00$
```

>From 42 seconds to 19 seconds -- sweet!

But can we do better? Notice that for each promoter, we're scanning through the entire genome. So we're scanning through the entire genome 5000 times.

Is there a way we can scan through the entire genome just once? Yes, and here is one solution:

```
# Read the genome data from a file
open(GENOME, "genome_data") or die "a horrible death: $!";
my $genome = <GENOME>;

# Read the promoter data from a file
open(PROMOTERS, "promoters") or die "a horrible death: $!";
foreach $promoter (<PROMOTERS>) {
        chomp $promoter;
        $promoters{$promoter} = 1;
}

# Look for each occurence of each promoter in the genome
my $genomelength = length($genome);
for($i = 0; $i < $genomelength - 10 + 1; ++$i) {
        my $subsequence = substr($genome, $i, 10);

        # Now we just look in the hash to see if this subsequence is a promoter
        if($promoters{$subsequence}) {
                $db{$promoter} = $i;
        }
}
```

and we run a timing on it to get ("findpromoters2"):

```
-bash-3.00$ date ; perl findpromoters2 ; date
Thu Oct 20 15:42:15 EDT 2005
Thu Oct 20 15:42:16 EDT 2005
-bash-3.00$
```

That's one second, maybe less.

And so we've achieved a 43-fold speedup in our program. What was taking, say, two days to compute, now takes an hour. We couldn't have achieved that speedup going to a super expensive computer (well, maybe a cluster, which we'll discuss later.)

And so we see that finding a better algorithm is the best way to get good performance.

What, exactly, did we do? We eliminated unnecessary work. We eliminated the repetitive reading in of the genome data from the disk; and we eliminated multiple scanning through the genome data.

These are the kinds of things that you can often find in the first version of a working program. So don't neglect the important step of editing your code after you get a working draft.

Programming for Biology Last modified: Mon Oct 20 23:14:38 EDT 2008

Using Less Space

Here is the main problem of space in bioinformatics:
Very large strings will swamp the main memory on your computer.

(Main memory, or RAM, is where your computer holds a running program; it is much smaller than the memory on your disks.)

When a program on your computer starts to use up too much main memory, its performance starts to degrade. The program will first enlist a portion of disk space to hold the part of the running program that it can no longer fit. This is called **swapping**.

But when a program starts swapping, which involves a lot of writing and reading to and from hard disk, it can get increasingly slow. The program may even start **thrashing**, that is, repeatedly writing and reading large amounts of data between main memory and hard disk. A program that is thrashing is going really slow, and it's slowing down the whole computer and other programs, too.

Take this snippet of code that calls `get_chromosome`:

```
my $chromosome1 = getchromosome(1);
```

When `getchromosome(1)` returns the data from human chromosome 1 to be stored in `$chromosome1`, the program uses 100Mb of memory.

Operating on the chromosome may use additional memory. For instance, in perl, when you do a regular expression search, you often want to save the successful match by using parentheses that set the special variables `$1`, `$&`, and so on.

```
$chromosome =~ /AA(GAGTC*T)/;
my $pattern = $1;
```

But once you use these special variables, the inner workings of perl require the use of considerable additional memory by your program. And you may make copies of all or part of the chromosome.

Your resulting code may be clear, straightforward to understand, and correct -- all good and proper things for code to be -- but the amount of memory usage may still seriously slow down your program.

Motto: **copying large strings is slow and takes up large amounts of memory**

Editing for Space

Often, a program that barely runs at all and takes many hours of clogging up the computer, can be rewritten to run more quickly by rewriting the algorithm so that it uses only a small fraction of the memory. It will fit into less memory, and also run a lot faster.

Use references to save space

There's one easy way to cut down on the number of big strings in a program.

Normally (without using references) a subroutine makes copies of the values passed into it, and it makes copies of the values returned from it.

References allow subroutines to avoid the string copying.

When we pass a reference to a string as an argument to a subroutine, we don't pass a copy of the string -- we pass a reference to the string, which takes almost no additional space.

And when the subroutine ends, whatever we've done with the string is immediately available to the calling program, without having to use the `return` function, which would also copy the string.

In our example:

```
load_chromosome( 1, \$chromosome1 );
```

This new subroutine has two arguments. The `1` indicates that we want the biggest human chromosome, chromosome 1.

The second argument is a reference to a scalar variable. Inside the subroutine, the reference is most likely used to initialize an argument `$chromref`, which is a reference to the genomic data. And then, in the subroutine, the DNA data is put into the dereferenced string:

```
sub load_chromosome {
        my($chromnumber, $chromref) = @_;

        ...(omitted)...

        $$chromref = <CHROMOSOME1>
}
```

It is not necessary to return the whole chromosome from the subroutine, which would make a copy of it. The value is passed by the reference *out* of the subroutine.

Using references is also a great way to pass a large amount of data *into* a subroutine without making copies of it. In this case, however, the fact that the subroutine can change the contents of the referenced data is something to watch out for.

The rule of thumb is: **if you don't need two copies of the data, you can use references**.

Managing Memory with Buffers

One of the most efficient ways to deal with very large strings is to deal with them a little at a time.

Here is an example of a program that searches an entire chromosome for a particular 12-base pattern, using very little memory.

When searching for any regular expression in a chromosome, it's hard to see how you could avoid putting the whole chromosome in a string. But very often there's a limit to the size of what you're searching for. In this program, I'm looking for the 12-base pattern "ACGTACGTACGT."

I'm going to read the chromosome data into memory just a line or two at a time, search for the pattern, and then *reuse* the memory to read in the next line or two of data.

The extra programming work involves:

First, keeping track of how much of the data has been read in, so I can report the locations in the chromosome of successful searches.

Second, making sure I search across line breaks as well as within lines of data from the input file.

The following program reads in a FASTA file searches for my pattern in any amount of DNA--a whole chromosome, a whole genome, a year's worth of Solexa data, even all known genetic data, while using only a small amount of main memory.

```
$ perl  find_fragment  human.dna
```

For testing purposes I made a very short FASTA DNA file, `human.dna`, which contains:

```
>human dna: ACGTACGTACGT appears at positions 10, 40, and 98
AAAAAAAAAACGTACGTACGTCCGCGCGCGCGCGCGCACGTACGTACG
TGGGGGGGGGGGGGGCCCCCCCCCGGGGGGGGGGGAAAAAAAAACG
TACGTACGTTTTTTTTTTTTTTTTTTTTTTTTTT
```

Here's the code for the program `find_fragment`:

```
#!/usr/bin/perl

use warnings;
use strict;

# $fragment: the pattern to search for
# $fraglen:  the length of $fragment
# $buffer:   a buffer to hold the DNA from the input file
# $position: the position of the buffer in the total DNA

my($fragment, $fraglen, $buffer, $position) = ('ACGTACGTACGT', 12, '', 0);

# The first line of a FASTA file is a header and begins with '>'
my $header = <>;

# Get the first line of DNA data, to start the ball rolling
$buffer = <>;
chomp $buffer;

# The remaining lines are DNA data ending with newlines
while(my $newline = <>) {

    # Add the new line to the buffer
    chomp $newline;
    $buffer .= $newline;

    # Search for the DNA fragment, which has a length of 12
    # (Report the character at string position 0 as being at position 1,
    # as usual in biology)
    while($buffer =~ /$fragment/gi) {
        print "Found $fragment at position ", $position + $-[0] + 1, "\n";
    }

    # Reset the position counter (will be true after you reset the buffer, next)
    $position = $position + length($buffer) - $fraglen + 1;

    # Discard the data in the buffer, except for a portion at the end
    # so patterns that appear across line breaks are not missed
    $buffer = substr($buffer, length($buffer) - $fraglen + 1, $fraglen - 1);
}
```

Here's the output of running the command
`perl find_fragment human.dna`:

```
 Found ACGTACGTACGT at position 10
 Found ACGTACGTACGT at position 40
 Found ACGTACGTACGT at position 98
```

How the Code Works

I want to search for the fragment even if it is broken by new lines, so I'll have to look at least at two lines at a time. I get the first line, and in the while loop that follows I'll start by adding more lines to the buffer.

Then the while loop starts reading in the next lines of the FASTA file. The newline character is removed with `chomp` and the new line is added to the `$buffer`.

Then comes the short while loop that does the regular expression pattern match of the `$fragment` in the `$buffer`.

When the fragment is found the program simply prints out the fragment's position. The variable `$position` holds the position of the beginning of the buffer in the total DNA.

I also add 1, because biologists always say that the first base in a sequence of DNA is at position 1, whereas Perl says that the first character in a string is at position 0. So I add 1 to the Perl position to get the biologist's position.

The last two lines of code reset the buffer. First we eliminate the beginning (already searched) of the buffer, and then we adjust the `$position` counter accordingly. The buffer is shortened so that it just keeps the part at the very end that might be part of a pattern match that spans the newlines.

The program manages to search the entire genome for the fragment, while keeping at most two lines' worth of DNA in `$buffer`, It performs very quickly, compared to a program that reads in a whole genome and blows out the memory in the process.

When You Should Bother

Programs may be developed on one computer, but run on very different computers.

A space-inefficient program might well work fine on your computer, but not work well at all when you run it on another computer with less main memory installed. Or, it might work fine on the fly genome, but start thrashing when you try it on the human genome.

If you know you'll be dealing with large data sets, like genomes, take the amount of space your program uses as an important constraint when designing and coding. Then you won't have to go back and redo the entire program when a large amount of DNA gets thrown at the program.

Data Compression

In Perl, as in any programming system, the size of the data that the program uses is an absolute lower bound on how fast the program can perform.

Each base is typically represented in a computer language as one ASCII character taking one 8-bit byte, so 3 gigabases equals 3 gigabytes. Of course, you could represent each of the four bases using only 2 bits, so considerable compression is possible; but such space efficiency is not commonly employed. When it is, you can pack 4 times as much data into a given space (for nucleotides, that is.)

```
A 00
C 01
G 10
T 11
```

If you want an exercise, try using perl functions `pack` and `vec` to compress DNA sequence data to 4 bases per byte.

Programming for Biology Last modified: Mon Oct 20 23:14:38 EDT 2008

Profiling

You saw earlier an easy way on Unix to see how long a program takes:

```
date; perl findpromoters1; date
```

This prints the time, then immediately runs the program, and then immediately prints the time again.

Perl has several much more detailed ways to examine the performance of a program.

I'll just show you one of them, called **`DProf`**. `DProf` reports on various aspects of your program's performance.

The most valuable report is probably the summary by subroutine.

By seeing which subroutines are taking the most time, you can narrow your re-editing of the program to just those subroutines, and quickly make the improvements where they count the most.

For demonstration, I'm going to use a program with a few subroutines; namely, the `makerandom` program we used earlier to make random DNA genomic sequence and putative DNA binding sites.

First you have to load the `Devel::Prof` module in your program. You do this by adding the `-d:DProf` command-line argument. Then when your program runs, the module makes counts of many things in the program. Your program will take a bit longer to run, but you'll collect valuable statistics on its performance.

So one can simply run the program as usual, adding the command-line argument. When it's done, it will have created a file called `tmon.out` in my directory. I then run the `dprofpp tmon.out` program to see the results of the profile of my program:

```
$ perl -d:DProf makerandom
$ dprofpp tmon.out
Total Elapsed Time = 5.464274 Seconds
  User+System Time = 5.354274 Seconds
Exclusive Times
%Time ExclSec CumulS #Calls sec/call Csec/c  Name
 72.2   3.870  7.594 105000   0.0000 0.0000  main::randomnucleotide
 69.5   3.725  3.725 105000   0.0000 0.0000  main::randomelement
 33.7   1.807  9.402   5001   0.0004 0.0019  main::make_random_DNA
 0.22   0.012  0.525      1   0.0125 0.5250  main::make_random_DNA_set
$
```

If I wanted to speed this program up, I'd head straight for the `randomelement` and `randomnucleotide` subroutines to see what I might be able to tweak in them, since my analysis shows that they take almost all the time in the program.

`DProf` has many options, but this is how I almost always use it, as it's simple and tells me what I need to know.

Some older perls might not have `DProf` installed, in which case you have to do something like this: (you may need root permission):

```
$ perl -MCPAN -e shell

cpan shell -- CPAN exploration and modules installation (v1.7601)
ReadLine support enabled

cpan> install Devel::DProf
CPAN: Storable loaded ok
Going to read /root/.cpan/Metadata
  Database was generated on Wed, 19 Oct 2005 22:01:03 GMT
Devel::DProf is up to date.

cpan> quit
Lockfile removed.
$
```

In this case perl reported that the `Devel::DProf` module was already installed with the latest version; if not, it would have installed it.

You know, I wonder if I can speed up my `makerandom` program. Let's look at it. Hmmm. I did try a few things out: let's see how the new program `makerandom2` behaves:

```
$ perl -d:DProf makerandom2
$ dprofpp tmon.out
Total Elapsed Time =  1.27999 Seconds
  User+System Time =  1.27999 Seconds
Exclusive Times
%Time ExclSec CumulS #Calls sec/call Csec/c  Name
 96.8   1.240  1.240   5001   0.0002 0.0002  main::make_random_DNA
 0.78   0.010  0.050      1   0.0100 0.0500  main::make_random_DNA_set
$
```

Cool! From over 5 seconds to a little over 1 second. A five-fold speedup!

How did I do it? Here's the new version:

```
srand();

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

$dna = make_random_DNA(1000000);
open(DNA, ">genomic_data") or die;
print DNA $dna;
```

```
@promoters = make_random_DNA_set(10, 5000);
open(PROMOTERS, ">promoters") or die;
print PROMOTERS join("\n",@promoters),"\n";

# Make a string of random DNA of specified length.
sub make_random_DNA {

    my($length) = @_;
    my $dna;

    for (my $i=0 ; $i < $length ; ++$i) {
        $dna .= $nucleotides[rand @nucleotides];
    }

    return $dna;
}

# make_random_DNA_set
sub make_random_DNA_set {

    my($length, $size_of_set) = @_;
    my $dna;
    my @set;

    # Create set of random DNA
    for (my $i = 0; $i < $size_of_set ; ++$i) {

        # make a random DNA fragment
        $dna = make_random_DNA ( $length );

        # add $dna fragment to @set
        push( @set, $dna );
    }

    return @set;
}
```

First, I moved the line

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

out of a subroutine and up to the top of the program. This way the array doesn't have to get reinitialized each time the program is called.

But much more importantly, I eliminated two subroutine calls entirely, and put their functionality directly into the lines of code that were calling them. First I axed `randomelement` by putting its functionality directly into the calling subroutine `randomnucleotide`: from

```
sub randomnucleotide {

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

    return randomelement(@nucleotides);
}

sub randomelement {

    my(@array) = @_;

    return $array[rand @array];
}
```

to

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

sub randomnucleotide {

    return $nucleotides[rand @nucleotides];
}
```

and finally I eliminated `randomnucleotide` by putting its code directly into the calling program: from

```
        $dna .= randomnucleotide(  );
```

to

```
        $dna .= $nucleotides[rand @nucleotides];
```

In short, I eliminated two subroutine calls that were each being called 105000 times, and that made a significant speedup. Usually, you're more likely to try to improve a subroutine than to eliminate it, but as you see eliminating a subroutine can on occasion have big payoffs.

The book by Bentley "Writing Efficient Code" discusses such "tricks" in entertaining and useful detail.

So I hope you're convinced that `DProf` is worthwhile. There are other profiling methods available in Perl too, and you might want to explore them.

Programming for Biology Last modified: Mon Oct 20 23:14:38 EDT 2008

There are different ways to think of parallel processing.

Parallel Algorithms

One kind of parallel processing actually uses the specific topology of the interconnections between the CPUs to implement new kinds of algorithms. This kind of parallel processing is fascinating and gives you very fast programs, but is *way* beyond the scope of this lecture or this course. But I thought you'd like to know that it exists.

In this hard-core parallel algorithms work, you might work on special computers (e.g. "grids", "butterfly networks") or even on purely theoretical models of parallel computation, and you design algorithms to run on those types of parallel computers.

Parallel Processing on Networks and Clusters

More common is this scenario: say you are doing 40 tasks, one after the other, and each one takes an hour. It will take your working week to finish the tasks.

Now let's say you figure out a way to do all the tasks simultaneously, and each one still takes an hour. You'll now finish the tasks, all of them, in one hour instead of one week.

One kind of parallel processing is just like this example. That's the kind of parallelism I'll talk about here, in terms of networks and clusters and threads. You simply divide your program up into parts that can be performed simultaneously, and then you run each part on its own CPU. Not all problems can be divided up like this, but those that can (say running a million blast searches) can get big speedups fairly easily.

Network Programming

One of the most successful forms of multi-processor computing has been *network programming*.

Network programming involves connecting two or more computers by a communications line and implementing a protocol that enables them to exchange information.

The development of computer networks began in earnest in the 1950s, and the various networks were interconnected by the *internet* (from *inter*connected *net*works) beginning in the late 1970s.

The protocols supported by the internet gradually expanded, until the protocols known as the *web* (or "world wide web") became widely popular beginning around 1990.

It is quite possible to program several computers to interact, using the several programming interfaces to the protocols that are available from such languages as perl.

Perl has supported these protocol interfaces since the beginning. I can speak from personal experience that it's a lot of fun to build a useful network service in this way. (In 1992 I was searching all of Genbank with regular expressions in about 35 seconds, by distributing the job with a network service written entirely in perl.)

I recommend the book "Network Programming with Perl" by Lincoln Stein if you're interested in these techniques.

Threads

Threads are different from, but related to, multiprocessing. Threads are multiple execution paths built into one process, that share resources like global variables, signals, and such. You can have a multithreading program that runs on a single processor; or, if you're running on a multiprocessor (it's common to have from 2 to around 24 processors on a given machine) the threads may be executed on different processors, giving you the advantage of parallelism.

Threads are a capability that is built into an operating system (or not, as the case may be.) If your operating system supports threads, and your programming language gives you access to them, then you can use them in your program.

If you're interested in threads, you want to use the "threads" (not "Threads") module:

```
use threads;
```

I'm going to skip the examples of threads programs: see me if you're interested.

Clusters

Clusters are multiple CPUs joined in a simple network. They are typically used to take a program that must compute the same way over many inputs, and run the program on all the CPUs, dividing the input up between them.

If you have access to a (usually) Linux cluster where you work, take the time to find out how to submit programs to it.

In a recent job I had, I had to do three computation-intensive calculations over several genomes. Each one took a week or two to finish when running on a single computer. On the Linux cluster, they all finished within a small number of hours, and using that precomputation I was able to carry my search for novel genes to a successful conclusion.

This Linux cluster has about 450 CPUs, and is a fairly big one. But it's quite straightforward -- you could do it yourself -- to buy 10 or 20 inexpensive Linux boxes and construct a Linux cluster that can speed up your large-scale, repetitive computations by 10 or 20 times.

Programming for Biology *Last modified: Mon Oct 20 23:14:38 EDT 2008*

Using Less Space

Here is the main problem of space in bioinformatics:
Very large strings will swamp the main memory on your computer.

(Main memory, or RAM, is where your computer holds a running program; it is much smaller than the memory on your disks.)

When a program on your computer starts to use up too much main memory, its performance starts to degrade. The program will first enlist a portion of disk space to hold the part of the running program that it can no longer fit. This is called **swapping**.

But when a program starts swapping, which involves a lot of writing and reading to and from hard disk, it can get increasingly slow. The program may even start **thrashing**, that is, repeatedly writing and reading large amounts of data between main memory and hard disk. A program that is thrashing is going really slow, and it's slowing down the whole computer and other programs, too.

Take this snippet of code that calls `get_chromosome`:

```
my $chromosome1 = getchromosome(1);
```

When `getchromosome(1)` returns the data from human chromosome 1 to be stored in `$chromosome1`, the program uses 100Mb of memory.

Operating on the chromosome may use additional memory. For instance, in perl, when you do a regular expression search, you often want to save the successful match by using parentheses that set the special variables `$1`, `$&`, and so on.

```
$chromosome =~ /AA(GAGTC*T)/;
my $pattern = $1;
```

But once you use these special variables, the inner workings of perl require the use of considerable additional memory by your program. And you may make copies of all or part of the chromosome.

Your resulting code may be clear, straightforward to understand, and correct -- all good and proper things for code to be -- but the amount of memory usage may still seriously slow down your program.

Motto: **copying large strings is slow and takes up large amounts of memory**

Editing for Space

Often, a program that barely runs at all and takes many hours of clogging up the computer, can be rewritten to run more quickly by rewriting the algorithm so that it uses only a small fraction of the memory. It will fit into less memory, and also run a lot faster.

Use references to save space

There's one easy way to cut down on the number of big strings in a program.

Normally (without using references) a subroutine makes copies of the values passed into it, and it makes copies of the values returned from it.

References allow subroutines to avoid the string copying.

When we pass a reference to a string as an argument to a subroutine, we don't pass a copy of the string -- we pass a reference to the string, which takes almost no additional space.

And when the subroutine ends, whatever we've done with the string is immediately available to the calling program, without having to use the `return` function, which would also copy the string.

In our example:

```
 load_chromosome( 1, \$chromosome1 );
```

This new subroutine has two arguments. The `1` indicates that we want the biggest human chromosome, chromosome 1.

The second argument is a reference to a scalar variable. Inside the subroutine, the reference is most likely used to initialize an argument `$chromref`, which is a reference to the genomic data. And then, in the subroutine, the DNA data is put into the dereferenced string:

```
 sub load_chromosome {
         my($chromnumber, $chromref) = @_;

         ...(omitted)...

         $$chromref = <CHROMOSOME1>
 }
```

It is not necessary to return the whole chromosome from the subroutine, which would make a copy of it. The value is passed by the reference *out* of the subroutine.

Using references is also a great way to pass a large amount of data *into* a subroutine without making copies of it. In this case, however, the fact that the subroutine can change the contents of the referenced data is something to watch out for.

The rule of thumb is: **if you don't need two copies of the data, you can use references**.

Managing Memory with Buffers

One of the most efficient ways to deal with very large strings is to deal with them a little at a time.

Here is an example of a program that searches an entire chromosome for a particular 12-base pattern, using very little memory.

When searching for any regular expression in a chromosome, it's hard to see how you could avoid putting the whole chromosome in a string. But very often there's a limit to the size of what you're searching for. In this program, I'm looking for the 12-base pattern "ACGTACGTACGT."

I'm going to read the chromosome data into memory just a line or two at a time, search for the pattern, and then *reuse* the memory to read in the next line or two of data.

The extra programming work involves:

First, keeping track of how much of the data has been read in, so I can report the locations in the chromosome of successful searches.

Second, making sure I search across line breaks as well as within lines of data from the input file.

The following program reads in a FASTA file searches for my pattern in any amount of DNA--a whole chromosome, a whole genome, a year's worth of Solexa data, even all known genetic data, while using only a small amount of main memory.

```
$ perl  find_fragment  human.dna
```

For testing purposes I made a very short FASTA DNA file, `human.dna`, which contains:

```
>human dna: ACGTACGTACGT appears at positions 10, 40, and 98
AAAAAAAAAACGTACGTACGTCCGCGCGCGCGCGCGCACGTACGTACG
TGGGGGGGGGGGGGGCCCCCCCCCGGGGGGGGGGGAAAAAAAAAACG
TACGTACGTTTTTTTTTTTTTTTTTTTTTTTTTT
```

Here's the code for the program `find_fragment`:

```
#!/usr/bin/perl

use warnings;
use strict;

# $fragment: the pattern to search for
# $fraglen:  the length of $fragment
# $buffer:   a buffer to hold the DNA from the input file
# $position: the position of the buffer in the total DNA

my($fragment, $fraglen, $buffer, $position) = ('ACGTACGTACGT', 12, '', 0);

# The first line of a FASTA file is a header and begins with '>'
my $header = <>;

# Get the first line of DNA data, to start the ball rolling
$buffer = <>;
chomp $buffer;

# The remaining lines are DNA data ending with newlines
while(my $newline = <>) {

    # Add the new line to the buffer
    chomp $newline;
    $buffer .= $newline;

    # Search for the DNA fragment, which has a length of 12
    # (Report the character at string position 0 as being at position 1,
    # as usual in biology)
    while($buffer =~ /$fragment/gi) {
        print "Found $fragment at position ", $position + $-[0] + 1, "\n";
    }

    # Reset the position counter (will be true after you reset the buffer, next)
    $position = $position + length($buffer) - $fraglen + 1;

    # Discard the data in the buffer, except for a portion at the end
    # so patterns that appear across line breaks are not missed
    $buffer = substr($buffer, length($buffer) - $fraglen + 1, $fraglen - 1);
}
```

Here's the output of running the command
`perl find_fragment human.dna`:

```
 Found ACGTACGTACGT at position 10
 Found ACGTACGTACGT at position 40
 Found ACGTACGTACGT at position 98
```

How the Code Works

I want to search for the fragment even if it is broken by new lines, so I'll have to look at least at two lines at a time. I get the first line, and in the while loop that follows I'll start by adding more lines to the buffer.

Then the while loop starts reading in the next lines of the FASTA file. The newline character is removed with `chomp` and the new line is added to the `$buffer`.

Then comes the short while loop that does the regular expression pattern match of the `$fragment` in the `$buffer`.

When the fragment is found the program simply prints out the fragment's position. The variable `$position` holds the position of the beginning of the buffer in the total DNA.

I also add 1, because biologists always say that the first base in a sequence of DNA is at position 1, whereas Perl says that the first character in a string is at position 0. So I add 1 to the Perl position to get the biologist's position.

The last two lines of code reset the buffer. First we eliminate the beginning (already searched) of the buffer, and then we adjust the `$position` counter accordingly. The buffer is shortened so that it just keeps the part at the very end that might be part of a pattern match that spans the newlines.

The program manages to search the entire genome for the fragment, while keeping at most two lines' worth of DNA in `$buffer`, It performs very quickly, compared to a program that reads in a whole genome and blows out the memory in the process.

When You Should Bother

Programs may be developed on one computer, but run on very different computers.

A space-inefficient program might well work fine on your computer, but not work well at all when you run it on another computer with less main memory installed. Or, it might work fine on the fly genome, but start thrashing when you try it on the human genome.

If you know you'll be dealing with large data sets, like genomes, take the amount of space your program uses as an important constraint when designing and coding. Then you won't have to go back and redo the entire program when a large amount of DNA gets thrown at the program.

Data Compression

In Perl, as in any programming system, the size of the data that the program uses is an absolute lower bound on how fast the program can perform.

Each base is typically represented in a computer language as one ASCII character taking one 8-bit byte, so 3 gigabases equals 3 gigabytes. Of course, you could represent each of the four bases using only 2 bits, so considerable compression is possible; but such space efficiency is not commonly employed. When it is, you can pack 4 times as much data into a given space (for nucleotides, that is.)

```
A 00
C 01
G 10
T 11
```

If you want an exercise, try using perl functions `pack` and `vec` to compress DNA sequence data to 4 bases per byte.

Programming for Biology Last modified: Mon Oct 20 23:14:38 EDT 2008